UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

DESARROLLO DE UN VIDEOJUEGO DE PELEA CON PERSONAJE ADVERSARIO CONTROLADO POR INTELIGENCIA ARTIFICIAL

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

NICOLÁS ANDRÉS ESCUDERO MUÑOZ

PROFESOR GUÍA:
FRANCISCO GUTIÉRREZ FIGUEROA

MIEMBROS DE LA COMISIÓN:
IVÁN SIPIRAN MENDOZA
JUAN BARRIOS NUÑEZ

SANTIAGO DE CHILE
2021

Resumen

Diversos géneros de videojuegos han sido introducidos al campo de la inteligencia artificial, pero uno de estos que aún no ha sido explorado es el género de pelea. Con personajes controlados por computadora que son casi invencibles o demasiado malos, este género se puede beneficiar altamente por la aplicación de modelos de inteligencia artificial a sus personajes controlados por computadora. De esta forma, al no existir muchas aplicaciones notables de esto, este proyecto va a resolver el problema de la falta de aplicación de modelos de inteligencia artificial en videojuegos de pelea. Esto se va a resolver aplicando modelos de inteligencia artificial a un personaje controlado por computadora dentro de un juego de pelea simplificado construido por el alumno.

La construcción se hace dentro de Unity utilizando la librería ML-Agents para construir y entrenar modelos de inteligencia artificial que controlan personajes controlados por computadora y se enfrentan con jugadores humanos. Se construyeron dos personajes, uno controlado por el modelo de inteligencia artificial y otro por el jugador humano, junto con una arena adecuada para un videojuego de pelea. El entrenamiento de los modelos se hace dentro de Unity conectando el ambiente de juego con el entrenador virtual de ML-Agents tomando acciones realizadas por ambos jugadores como valores de observación y ejecutando movimientos como respuesta.

La validación se realizó con una prueba de concepto con 15 participantes que probaban el juego y luego respondían un cuestionario. El feedback fue bastante diverso debido a que el juego construido era un videojuego de pelea simplificado y el modelo de inteligencia artificial era muy defensivo. Pero se cumplieron los objetivos de construcción del juego y aplicación de un modelo de inteligencia artificial a un jugador controlado por computadora.

En conclusión, debido al resultado y opiniones obtenidas por el cuestionario existe trabajo futuro para mejorar tanto la experiencia del jugador como la calidad de los modelos de inteligencia artificial asociados. Este trabajo se deja propuesto para una futura iteración.

A mi familia, que a pesar de que me caí en el camino, jamás perdieron la fe en mi.

Agradecimientos

Agradezco a mi profesor guía por tenerme toda la paciencia del mundo, a mi familia y amigos por estar apoyándome durante toda esta etapa y finalmente, a María José Berger por prestarme su computador para poder terminar mi proyecto.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

En la última década, la tecnología se ha expandido de forma exponencial y se ha adentrado en todos los ámbitos de la vida cotidiana. Uno de ellos es la industria del entretenimiento, donde se dejó de requerir elementos o espacios físicos para poder acceder a música, material audiovisual, juegos, etc.

Hasta agosto del 2020, se estimaba que existían 3.1 billones de consumidores juegos de vídeo, lo que corresponde al 40 % de la población mundial. Es tanta su presencia en la sociedad que en los últimos años los videojuegos se han vuelto la industria del entretenimiento más lucrativa del mundo, con un valor de $USD 145.7 mil millones en 2019, en comparación con $ 42.5 mil millones en ganancias de taquilla y $ 20.2 mil millones en música. [1]

Este auge descontrolado, donde incluso jugar videojuegos se ha vuelto una profesión, ha traído con sigo nuevas exigencias y estándares para quienes consumen esta actividad, por lo que se ha vuelto necesario innovar en cómo se desarrollan videojuegos que puedan satisfacer las necesidades cambiantes de sus usuarios.

Dentro de los múltiples géneros de videojuegos que existen, uno que se ha mantenido presente desde el inicio de esta industria es el de los juegos de pelea (*fighting games* en inglés). En 2018, este género representó el 7.8 % de juegos vendidos ese año [2].

Otra área tecnológica que ha crecido exponencialmente en los últimos años es la de la inteligencia artificial (desde ahora IA) y aprendizaje de máquinas (desde ahora ML por sus siglas en inglés). Gracias a su gran rango de aplicaciones, tanto en la academia como en la industria, ha ganado gran popularidad, integrándose en diversos campos desde aplicaciones de mapas o avisos publicitarios hasta vehículos que son capaces de conducirse solos.

Esta tecnología también se ha puesto a disposición del entretenimiento con ejemplos como generadores de texto, escritores automatizados de historias[3] y videojuegos. En este último, uno de sus principales exponentes es el modelo de inteligencia artificial (desde ahora MIA)

[1] https://www.wepc.com/news/video-game-statistics/
[2] https://www.wepc.com/news/video-game-statistics/
[3] https://www.deepstory.ai

creado por **OpenAI**[4] y aplicado al popular juego competitivo **Dota 2**[5], logrando vencer al equipo coronado campeón mundial ese año. [6]. Otro ejemplo emblemático en el rubro es el modelo de la inteligencia artificial desarrollada por **DeepMind**[7] que es capaz de vencer a un 99.8 % de los jugadores[8] del juego de estrategia en tiempo real **Starcraft 2**[9].

A pesar de los resultados preliminares positivos obtenidos por la aplicación de modelos de inteligencia artificial en el ámbito de los videojuegos, su desarrollo no se ha extendido a nuevos géneros.

Un género que podría verse beneficiado por la incorporación de un MIA es el de juegos de pelea. Particularmente, en la implementación los jugadores controlados por computadora que este tipo de juegos poseen cuando se desea jugar de forma individual (*Single Player* en inglés). En la actualidad, la aptitud de estos personajes está poco adaptada a la experiencia de cada usuario y el comportamiento suele ser extremo, es decir, un adversario que es muy fácil de derrotar o un adversario que es prácticamente invencible. Esto significa una curva de destreza poco óptima para el jugador, pues su oponente difícilmente actúa a su misma capacidad.

Ante esto, surge la necesidad de un MIA capaz de adecuar el oponente controlado por computadora al nivel de destreza del usuario, con el objetivo de que este pueda tener una experiencia de juego más desafiante y entretenida.

1.1. Problema Abordado

Como se mencionó previamente, a pesar de sus diferencias, existe un factor común entre la mayoría de los videojuegos de pelea: no utilizan MIAs de forma efectiva, ya sea porque las reacciones de éstas son demasiado rápidas o porque exhiben comportamientos que son explotables por un jugador humano, haciendo enfrentarse a ellas, un trabajo trivial o extremadamente tedioso.

Un ejemplo del comportamiento inadecuado de la aplicación de un MIA a personajes controlados por computadora (desde ahora PCC) es el juego de pelea clásico **Street Fighter 2** donde la computadora abusaba de ventajas injustas para ganar sus partidas[10]. Uno de los exponentes con mejores resultados de la aplicación de un MIA a sus PCC es el juego **Killer Instinct** (2013). Este incluía modo de juego llamado **Shadow Mode** que, apoyándose en el MIA **Shadow AI** [7], luego de 3 entrenamientos, era capaz de adaptarse al estilo del jugador, representando "pelear contra su propia sombra".

[4]https://openai.com/

[5]https://www.dota2.com/home

[6]https://www.theverge.com/2019/4/13/18309459/openai-five-dota-2-finals-ai-bot-competition-og-e-sports-the-international-champion

[7]https://deepmind.com/

[8]https://www.theverge.com/2019/10/30/20939147/deepmind-google-alphastar-starcraft-2-research-grandmaster-level

[9]https://starcraft2.com/es-es/

[10]Video demostrativo de comportamiento abusivo de PCC usando un MIA: https://www.youtube.com/watch?v=laUAgEUunsI - demostración de AI abusiva

La aplicación efectiva de un MIA a los PCC podría traer múltiples beneficios para los jugadores, entrenando sus capacidades de forma personalizada según sus fortalezas y debilidades.

Dado lo anterior, en este trabajo de titulo se abordó el problema de los MIA y su poca aplicación a videojuegos de pelea.

1.2. Objetivos

1.2.1. Objetivo General

El objetivo general de este trabajo de título es desarrollar un videojuego de pelea, donde el personaje adversario está controlado por un modelo de IA, y buscar explorar cómo impacta esto en la experiencia de juego del usuario.

1.2.2. Objetivos Específicos

Para lograr el objetivo planeado se proponen los siguientes objetivos específicos:

1. Desarrollar una plataforma funcional que cumpla con las características mínimas de jugabilidad y robustez de un juego de pelea comercial. Estas características siendo: darle la capacidad de moverse y atacar a los jugadores, una arena que contenga bordes y piso sólido y finalmente que al atacar los personajes inflijan daño a su adversario con la funcionalidad dentro del juego que si la vida de alguno de estos llega a 0, el juego termina.
2. Integrar un modelo de inteligencia artificial encargado del comportamiento de los personajes controlados por computadora.
3. Configurar los parámetros para que el PCMIA tome una actitud ofensiva o defensiva dependiendo de los inputs del jugador.

1.3. Solución Desarrollada

Para abordar este problema se decidió crear un videojuego del género de pelea en 2D. Este juego fue una versión simplificada de un juego de pelea tradicional permitiendo solo dos botones de movimiento (solamente horizontal) y uno de ataque (el golpe), la idea era crear un ambiente para el entrenamiento de la inteligencia artificial y otro para realizar pruebas con usuarios.

Para el desarrollo del juego se utilizó el motor para la creación de juegos **Unity** junto con la librería asociada de **MLAgents** para crear la inteligencia artificial. Se eligió este motor

por ser más amigable para usuarios nuevos y personas con poca experiencia, además de la familiaridad del tesista con esta plataforma. Con respecto a la librería, esta fue escogida porque no existe otra opción similar dentro de *Unity*.

El comportamiento esperado del PCMIA es uno humano, que logre reaccionar a las acciones del jugador humano de acuerdo a la situación en la que se encuentre y que gracias a esto el jugador piense dos veces si esta jugando contra una computadora.

Para validar el desarrollo se realizaron pruebas con 15 usuarios que posteriormente respondieron una encuesta con respecto a su nivel de satisfacción con el juego.

1.4. Estructura del Documento

El resto del documento se estructura de la siguiente manera: en el capítulo 2 se presentan conceptos necesarios para la comprensión de los modelos de inteligencia artificial y otras herramientas de desarrollo utilizados en el proyecto, comparándolo con otras alternativas comerciales descartadas.

En el capítulo 3 se plantea y analiza la solución propuesta y las herramientas utilizadas mientras que en el capítulo 4 se describe el desarrollo del juego y la integración de las librerías necesarias para utilizar un MIA.

En el capítulo 5 se muestra un ejemplo guiado para comprender el uso de la plataforma y los resultados de la validación hecha por usuarios. Finalmente, en el capítulo 6 se encuentran las conclusiones y trabajo futuro del proyecto.

Capítulo 2

Marco Teórico

2.1. Juegos de Pelea

Dentro de los videojuegos el género de peleas consiste en un juego que generalmente enfrenta a dos jugadores que eligen a personajes y dentro de una arena pelean hasta que uno de los dos pierde toda su vida o se acaba el tiempo.

El videojuego de pelea en el que está basado este proyecto es *Street Fighter*, un juego de pelea en dos dimensiones que cumple con los requerimientos de robustez buscado en este tipo de juego, los personajes son capaces de moverse, de recibir daño, de infligir daño y si la vida de uno de estos llega a cero el juego termina.

Este juego también permite la creación de una versión simplificada, lo que es parte del objetivo del desarrollo del a plataforma.

2.2. Glosario

- **Adversario:** Un adversario es alguien que compite con otros que aspiran a un mismo objetivo o la superioridad en algo. Para un videojuego de pelea el adversario va a ser el oponente al que se tiene que enfrentar para poder ganar el juego.

- **Arena:** La arena se va a referir al espacio donde los jugadores en un videojuego de pelea se enfrentan. Para el caso del proyecto este va a ser un espacio de dos dimensiones con un piso sólido y bordes definidos para que los jugadores no puedan salir de ella.

- **Movimiento:** El movimiento en un videojuego de pelea se refiere al desplazamiento, ya sea vertical u horizontal del que son capaces los jugadores. Para motivos del proyecto se va a referir solamente al desplazamiento vertical del jugador.

- **Inteligencia Artificial:** Se refiere a los sistemas o las máquinas que imitan la inteligencia humana para realizar tareas y que tienen la capacidad de mejorar iterativamente

a partir de la información que recopilan.[1]

- **Aprendizaje reforzado:** Se trata de una forma de optimización basada en datos. La máquina aprende a partir de su propia experiencia interactuando con el entorno hasta dar con el comportamiento ideal. A partir de la información disponible, emprenderá acciones que repetirá y "reforzará" según las recompensas que obtenga.[2]
- **Política de acción:** Es la forma en que el agente se va a comportar de acuerdo a la información percibida del ambiente.
- **Gradient descent:** Es un algoritmo de optimización de primer orden usado para encontrar un mínimo local de una función diferencial.

2.3. Herramientas de desarrollo de videojuegos

Una de las principales labores de este trabajo de título fue desarrollar una plataforma en donde se puedan utilizar MIA. Como el foco es el género de peleas en videojuegos, se necesitó utilizar herramientas para la construcción de un juego que cumpliera los requerimientos de la plataforma.

A continuación se van a presentar las dos plataformas que fueron consideradas para la creación del juego en el proyecto.

2.3.1. Unity

Unity es un motor de juegos desarrollado por *Unity Tecnologies*, anunciado y lanzado en junio del 2005 como un motor exclusivo para el Mac OS-X. Originalmente diseñado para la creación de juegos móviles, el motor ha sido gradualmente expandido para poder soportar distintas plataformas y ha logrado abarcar la reputación de ser una plataforma fácil de usar para desarrolladores principiantes y popular para el desarrollo de juegos independientes.[3]

Este motor está construido a base de **scripts** que son segmentos de código basados en el lenguaje C#

2.3.2. Unreal Engine 4

Unreal Engine (desde ahora UE) es un motor de juego desarrollado por *Epic Games*, desarrollado inicialmente para el género de disparos en primera persona en computador ha ido expandiendo su uso en varios géneros de juegos tridimensionales llegando incluso a la industria del cine y la televisión. Su versión más reciente *Unreal Engine 4* ha sido usado para desarrollar una gran variedad de juegos hasta la fecha.

[1] https://www.uchile.cl/noticias/173079/que-es-y-como-funciona-la-inteligencia-artificial
[2] https://www.iic.uam.es/inteligencia-artificial/aprendizaje-por-refuerzo/
[3] https://techcrunch.com/2019/10/17/how-unity-built-the-worlds-most-popular-game-engine/

2.4. Modelos de IA

En esta sección se van a presentar ejemplos de MIA en juegos populares y que fue los logros que adquirieron para validar su construcción. A su vez, se va a explicar librerías que sirven para la construcción de MIA, primero va a ser una librería que solo puede usada en UE4 seguida de una que solo puede ser utilizada en Unity y que fue la librería escogida para el desarrollo de este proyecto.

2.4.1. OpenAI Five

OpenAI, una organización de investigación de IA, logró construir y entrenar una IA capaz de jugar el juego de estrategia *Dota 2*, llamada *OpenAI Five*, esta IA logro vencer a un equipo de nivel mundial en el juego en un formato de mejor de tres (ganar dos juegos para salir victorioso).

La IA fue desarrollada utilizando un espacio de observación restringido para imitar las mismas condiciones de juego de un jugador humano. El número de acciones validas que podía tomar fue discretizado en un espacio de 170000 acciones con algunas de estas condicionas por elementos del juego (como el tiempo de enfriamiento al utilizar habilidades). En comparación el numero de acciones promedio en juegos tradicionales como el ajedrez o el Go son 35 y 250 respectivamente.[4]

2.4.2. Alphastar

DeepMind, una organización de investigación de IA perteneciente a *google*, construyó **AlphaStar**, una IA que logró llegar al nivel más alto en las clasificaciones del juego, *Gran Master*, limitando las condiciones y acciones de la IA a estándares humanos.

El entrenamiento de esta IA es en base a observar juegos de jugadores profesionales y luego múltiples agentes combaten entre ellos usando un comportamiento explotativo para encontrar y enfocarse en las debilidades de sus oponentes para que así estos puedan aprender de estas falencias en futuras versiones.[5]

2.4.3. MindMaker

MindMaker (desde ahora MM) es un plugin que le permite a juegos y simulaciones dentro de UE4 funcionar como ambientes de entrenamiento para agentes autónomos de aprendizaje de máquinas. El plugin facilita una conexión en red entre el ambiente de entrenamiento dentro de UE y la librería de aprendizaje de máquinas utilizada por el agente para optimizar lo que sea que esté aprendiendo. La librería de aprendizaje de máquinas puede ser código propio

[4]https://openai.com/blog/openai-five/
[5]https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii

hecho en python, en caso de estar desarrollando una herramienta de aprendizaje de máquinas o puede utilizar un motor de aprendizaje ya creado.[6]

2.4.4. ML-Agents

The Unity Machine Learning Agents Toolkit (ML-Agents desde ahora MLA)[7] es un proyecto *open-source* que permite que los juegos y las simulaciones sirvan como entornos para entrenar agentes inteligentes. Se proporcionan implementaciones (basadas en PyTorch) de algoritmos para permitir tanto a desarrolladores de juegos como aficionados capacitar fácilmente agentes inteligentes para una gran variedad de juegos (2D, 3D y VR / AR). Los usuarios también pueden usar la API de Python proporcionada para capacitar a los agentes mediante *reinforced learning* [6], *imitation training* [4], *neuroevolution* [3] u otro método similar. Estos agentes entrenados se pueden usar para múltiples propósitos, incluido el control del comportamiento de NPC (en una variedad de entornos, como multiagente y adversario), pruebas automatizadas de compilaciones de juegos y evaluación de diferentes decisiones de diseño de juegos antes del lanzamiento.

2.4.5. Algoritmo utilizado en el Proyecto

El algoritmo usado en este proyecto se llama *Proximal Policy Optimization* (desde ahora PPO). PPO es un algoritmo de optimización que asegura que las nuevas políticas de acción (desde ahora PA) no esten alejadas de las antiguas. Esto se realiza usando una razón de probabilidad entre la PA nueva y antigua que es aplicada a la funicón objetivo del algoritmo. Esto asegura que la nueva PA se quede dentro de un pequeño intervalo de la antigua PA, lo que en consecuencia asegura que el cambio de comportamiento en intervalos no sea brusco y genere un aprendizaje incorrecto.

Los parámetros utilizados para el entrenamiento son conocidos como **Hyperparametros**.

Hyperparametros

A continuación se presenta una lista de los hyperparametros utilizados para la configuración del ambiente de entrenamiento utilizando el algoritmo PPO:

- **Batch size:** El número de experiencias (Observaciones del agente, acciones y recompensas obtenidas) utilizadas en una iteración de **gradient descent**.
- **Buffer size:** Corresponde al número de experiencias que deben ser recolectadas antes de que se produzca un aprendizaje o actualización al modelo.

[6] https://github.com/krumiaa/MindMaker/#readme
[7] https://github.com/Unity-Technologies/ml-agents#readme

- **Learning rate:** Corresponde a cuán fuerte es cada actualización de gradient descent. Generalmente, se debe reducir si el entrenamiento es inestable y la recompensa no aumenta constantemente.
- **Epsilon:** Corresponde al umbral aceptable de divergencia entre las políticas antiguas y nuevas durante la actualización de gradient descent. Establecer un valor pequeño resulta en actualizaciones más estables, pero también hace el proceso de entrenamiento más lento.
- **Lambd:** Corresponde al parámetro **Lambda** usado para calcular el **Generalized Advantage Estimate (GAE)**. Esta variable se puede ver como cuánto el agente depende del valor estimado actual para calcular el valor estimado actualizado. Un menor valor significa depender más del valor estimado actual, mientras que un valor mayor corresponde a depender de las recompensas obtenidas en el ambiente.
- **Number of Epoch:** Es el número de veces que se pasa por el buffer de experiencia durante *gradient descent*.
- **Learning Rate Schedule:** Corresponde a cómo el porcentaje de aprendizaje decae con el tiempo.
- **Normalize:** Indica si se va a realizar una normalización a los valores del vector de observaciones.
- **Hidden Units:** Este valor corresponde a cuántas unidades existen en cada capa conectada de la red neuronal.
- **Number of Layers:** Corresponde al valor de cuántas capas escondidas van a estar presente después del input de observación.
- **Vis Encode Type:** Corresponde al tipo de codificación usada para las observaciones visuales.
- **Keep Checkpoints:** Este es el número de checkpoints que salvar una vez que ocurre una cierta cantidad de pasos.
- **Max Steps:** Corresponde al número máximo de pasos por los que la simulación pasa durante el proceso de entrenamiento.
- **Time Horizon:** Corresponde a cuánta experiencia por agente es recolectada antes que sea añadida al buffer de experiencia.

2.4.6. Self-Play

Self-Play es un método de entrenamiento en aprendizaje reforzado donde la IA compite contra sí misma. La forma en que esto ocurre es que después de cierto número de pasos (determinado por el usuario) se guarda el modelo de la IA dentro de un stack. Una vez que se tiene un cierto número de elementos (determinado por el usuario) se toma al azar uno de estos modelos antiguos y se enfrenta a la IA actual con este. Cuando el stack se llena, los modelos más antiguos guardados son reemplazados por modelos nuevos.

2.4.7. Parámetros de Self-Play

Similar a los hyperparametros de la configuración de PPO, Self-Play posee sus propios parámetros que son requeridos para activar sus funciones:

- **Save steps:** Indica cada cuántos pasos se va a guardar una copia del modelo. Por ejemplo, si el número es 10000, entonces después de 10000 pasos se va a guardar una copia del modelo actual.
- **Team change:** Indica cada cuántos pasos los objetos dentro del entrenamiento van a cambiar de id de equipo.
- **Swap steps:** Indica cada cuántos pasos se va a reemplazar el modelo contra el cual se está jugando actualmente. Un número mayor nos dice que se va a jugar una cantidad mayor de veces contra un mismo modelo.
- **Window:** Corresponde al tamaño de la ventana donde se guardan los modelos pasados. Con un tamaño de ventana más grande existe una variedad mas amplia y antigua de oponentes contra los cuales entrenar la inteligencia artificial.
- **Play against current self ratio:** Indica la probabilidad del agente de jugar contra sí mismo. Como es un valor probabilístico, su valor va en el rango 0 a 1.
- **Initial elo:** Un sistema alternativo de medición aparte de las recompensas de ambiente, usando múltiples juegos para medir la habilidad relativa de dos jugadores. Este valor indica la cantidad inicial que va a tener la variable previo al inicio del entrenamiento

2.5. Trabajo Relacionado

A continuación se va a presentar una revisión de artículos recientes publicados en literatura específica del área, en donde se va a exponer la importancia de la inteligencia artificial en videojuegos junto con su actual avance.

2.5.1. Experiencia del Usuario relacionada con la Inteligencia Artificial

Denisova y Cairns [1] ejecutaron un experimento para medir la experiencia de usuarios en juegos digitales y cómo esta puede ser influenciada a través de la percepción de elementos que éstos creen estar dentro del juego.

Para poder lograr entender lo que compone la experiencia del jugador en juegos digitales, en primer lugar el juego en sí es un factor crucial para establecer la experiencia. Esto es, ya que jugar un juego de terror es muy distinto a uno de carreras. Otros factores dentro de los juegos que afectan la experiencia del jugador incluyen la música y el nivel de dificultad. Asimismo, otros factores externos al juego también influyen en la experiencia del jugador. Entre estos se incluye el tamaño de la pantalla, iluminación y la personalidad del jugador.

Al existir una gran variedad de elementos que influencian la experiencia del jugador, es difícil saber si estas son generadas por el juego mismo o por las expectativas sobre el juego. Así pues, *reviews* y *ratings* ayudan a cambiar la actitud de un posible jugador antes de que este haya probado el juego. Por otro lado, los jugadores esperan que juegos nuevos posean características y tecnología nuevas, los que son importantes para generar una buena primera impresión del juego.

Basado en lo dicho anteriormente, los autores proponen que es posible alterar la experiencia del jugador sugiriéndole que el juego contiene una característica capaz de afectar el desempeño del jugador. En el contexto del experimento presentado en este artículo, se trata de una inteligencia artificial adaptativa.

El primer experimento fue realizado con un grupo de 21 participantes usando el juego *Don't Starve*. Se les daba cinco minutos para que se acostumbraran al juego y a sus controles y dos sesiones de 20 minutos para realizar el experimento. A todos los participantes se les dio una explicación de lo que era una inteligencia artificial que se adapta antes de comenzar. La primera sesión "contenía" la inteligencia artificial adaptativa y la segunda no. En comparación de la segunda sesión, los participantes se sintieron más inmersos y desconectados del mundo real cuando jugaban en la sesión con la inteligencia artificial adaptativa. Incluso llegaban a decir que la inteligencia artificial hacia toda una diferencia en el juego.

El segundo experimento se realizó con un grupo de 40 personas bajo el mismo ambiente que el primer experimento y con resultados similares.

Este experimento demuestra que elementos tecnológicos novedosos dentro de un juego digital sirven para aumentar el nivel de inmersión y expectativa del jugador generando una mejor experiencia de juego en general, por lo que la correcta implementación de un inteligencia artificial adaptativa en un juego de pelea incrementaría el nivel de inmersión del juego y haría mejor la experiencia del jugador en el juego.

2.5.2. Algoritmos de Inteligencia Artificial Aplicados a Juegos de Pelea

Gajardo, Besoain y Barriga [2] realizaron un estudio acerca de la importancia de la calidad de una inteligencia artificial oponente en un videojuego de pelea. Otros géneros pueden depender de la historia o de fuentes visuales dentro del juego, pero los juegos de pelea son sobre la confrontación y la experiencia con un adversario. A su vez, se exponen algoritmos comunes de comportamiento en videojuegos como soluciones más recientes.

El primer algoritmo que se presenta son las máquinas de estados finitos, FSM desde ahora en adelante. Las FSM son un modelo popular por su bajo requerimiento en conocimientos de programación, referencia visual y estudio constante. Estas están basadas en una máquina hipotética compuesta por uno o más estados y reglas de transición asociadas. Los estados pueden ser accedidos uno a la vez, con la máquina seleccionando los estados accesibles y revisando las condiciones y consecuencias de moverse a otro estado por medio de transiciones predefinidas. Las FSM siguen una lógica sencilla, comienzan desde un estado inicial donde el personaje comienza su su proceso de "toma de decisiones", en el cual considera distintas

posibles transiciones y los estados en los que estas desembocan. Si una de estas se cumple, se pasa al estado adecuado, en caso contrario se mantiene el estado actual.

Otro algoritmo que se presenta son los árboles de comportamiento, *behavior tree*, BT desde ahora. Los BT a diferencia de una FSM no mantiene un estado, sino que toman comportamientos que la inteligencia artificial puede ejecutar si se cumplen las condiciones necesarias. Estos comportamientos se organizan en un árbol jerárquico que controla las decisiones de la inteligencia artificial. El árbol comienza desde una raíz, aquí inicia la búsqueda, pasa a un nodo, y si este no es una hoja, inicia una función compositora.

- **Secuenciador:** Visita a cada hijo, del primero al último. En el caso de que alguno falle en su ejecución retorna el fracaso al padre, de lo contrario si el último hijo de la secuencia es ejecutado sin problemas, se retorna éxito.
- **Selector:** También visita cada nodo hijo en orden, pero la condición para pasar de un nodo a otro es que el primero falle. Pasa por cada uno hasta encontrar un nodo exitoso en cuyo caso retorna éxito, en caso contrario retorna fracaso.

Al llegar a un nodo hoja, este comienza a ser ejecutado, por lo que se retornará en funcionamiento, éxito o fracaso dependiendo del estado. En caso de estar en funcionamiento se reproducirá en cada tic hasta que se retorne éxito o fracaso. La versatilidad del modelo permita crear una inteligencia artificial competente y que pueda aprender automáticamente.

El último algoritmo que se presenta es la búsqueda en árboles de Monte Carlo, MCTS desde ahora. A diferencia de los métodos anteriores que permiten al diseñador especificar comportamientos, MCTS es un algoritmo de búsqueda adversaria para seleccionar los mejores movimientos posibles. El enfoque de MCTS es reproducir posibles victorias, comenzando con una muestra de movimientos aleatorios durante todo el juego entregando valores a los nodos del árbol. Así, los nodos mejor evaluados son más propensos a ser usados en futuras iteraciones. Cada ronda de búsqueda del MCTS consiste en 4 pasos: Selección, Expansión, Simulación y Propagación.

- **Selección:** Se seleccionan los movimientos desde la raíz hasta una hoja, tomando en cuenta estadísticas de victoria de iteraciones pasadas.
- **Expansión:** Si aún no se llega al final del juego, se expande el árbol añadiendo nuevos nodos que equivalen a todos los movimientos válidos.
- **Simulación:** Se seleccionan movimientos al azar, simulando el juego hasta el final.
- **Propagación:** Se utilizan los resultados de la simulación para actualizar estadísticas de los nodos entre el nodo hoja y la raíz.

Martínez-Arellano, Cant y Woods [5] proponen un método de creación de personajes para el juego de pelea *M.U.G.E.N.* que permite crear inteligencias artificiales interesantes, utilizando métodos computacionalmente baratos y que no requieran de un desarrollador experto.

El método propone un algoritmo de programación genética (GP) que refina las estrategias generadas al azar por el computador, generando estrategias mejores usando *tournament selection*.

GP es un método que toma elementos generados o conseguidos aleatoriamente, los ejecuta y compara para ver cuál es más eficiente, hace esto con todos los elementos que posee y una vez terminada la ejecución de cada uno escoge a los que tuvieron mejor desempeño, los combina y realiza todo este proceso nuevamente. Esto ocurre hasta llegar a un límite definido por el usuario o si uno de los elementos creados logra conseguir desempeño igual o mejor a lo deseado. El método de *Tournament Selection* involucra realizar "torneos" entre un conjunto de individuos elegidos al azar pertenecientes a la población de cromosomas del algoritmo genético. Los ganadores de cada torneo son los que poseen mayor puntaje en su función de evaluación y son elegidos para realizar *crossover* que es la combinación de la información de dos individuos para generar un nuevo cromosoma.

Los personajes de AI creados fueron probados contra otros personajes de AI y por 27 jugadores humanos. Se utilizaron 41 personajes creados utilizando GA para el test contra personajes de AI y 4 personajes creados manualmente usando a *Kung Fu Man* como base para realizar comparaciones entre la inteligencia artificial creada y la que existe inherentemente en el juego. Las AI generadas utilizando GA utilizaron distintos porcentajes de mutación, entre 1 % a 20 %, y distintas generaciones máximas, algunas llegando hasta las 100 generaciones. Las conclusiones del experimento entre inteligencias artificiales fue que los resultados entre AI entrenadas usando 100 generaciones no poseían grandes diferencias de las que se entrenaban utilizando 30 y todas tenían un desempeño sobresaliente sobre las 4 inteligencias artificiales basadas en la que está presente por defecto en el motor.

Las pruebas con humanos fueron evaluadas de acuerdo con los siguientes criterios: sus resultados, dificultad percibida y cuán satisfactorio era jugar contra ellos. Se realizaron pruebas con las 41 inteligencias artificiales creadas utilizando GA y las 4 manualmente creadas, cada jugador jugaba un mejor de 3 partidas contra la inteligencia artificial y luego daba su evaluación al respecto. Los resultados indicaban que el nivel de satisfacción y dificultad era mayor con las inteligencias artificiales creadas usando GA y menor con las que tenían de base la preexistente en el juego. Una gran cantidad de usuarios igualaban dificultad con los resultados de sus peleas, mientras que otros igualaban dificultad con satisfacción. A pesar de todo el resultado final indica un mayor nivel de reto y satisfacción con las inteligencias artificiales creadas con GA.

Las principales ventajas de este procedimiento son que no se requiere conocimiento previo de cómo programar las estrategias de los personajes de AI y no es necesario interactuar con el código interno del juego. Adicionalmente, el procedimiento es capaz de crear una gran diversidad de personajes con diferente habilidad estratégica, lo que podría usarse como un punto de inicio para un futuro proceso adaptativo.

Yoshida et al. [8] describen la aplicación de la búsqueda Monte-Carlo en árboles de decisión, MCTS desde ahora, para inteligencia artificial de juegos de pelea. MCTS es una técnica de búsqueda que usa simulaciones estocásticas, proceso no determinista, determinado tanto por las acciones predecibles como por elementos aleatorios. En el pasado se ha utilizado MCTS con buenos resultados en juegos de estrategia por turnos como *Go* o juegos en tiempo real como *Ms. Pac-Man*. La gran diferencia entre este juego y juegos de pelea es que el tiempo de reacción del primero es de 40 ms mientras que en el otro son 16.67 ms, haciendo la aplicación un tanto más compleja.

MCTS construye y utiliza el árbol de juego de acuerdo a los siguientes cuatro pasos: Selección, Expansión, Simulación y Propagación hacia atrás. Estos pasos son repetidos un número específico de veces, una vez terminado la acción dentro del juego ejecutada es representada por el hijo del nodo raíz que fue visitada el mayor número de veces en las repeticiones.

Se evalúa la efectividad en *FightingICE*[8], una plataforma de competencia presente en conferencias de Inteligencia Computacional y Conferencias de Juegos. Se realizaron pruebas contra cinco inteligencias artificiales que fueron ganadoras en el pasado. Los resultados confirman que el MCTS da una búsqueda efectiva para controlar la AI de un juego en la plataforma mencionada, al haber vencido a 4 de sus 5 oponentes en puntaje.

2.5.3. Herramienta para Desarrollar Juego de Pelea

Universal Fighting Engine es un kit de herramientas de Unity[9] para desarrollar juegos de pelea con una variedad de editores de Unity personalizados y fáciles de usar.

Unity es un motor para desarrollar videojuegos desarrollado por *Apple* el 2005, originalmente para el *MAC OS-X* y extendido para soportar otras plataformas el 2018. El motor puede crear juegos 2D, 3D, de realidad virtual y de realidad aumentada.

Universal Fighting Engine viene con una variedad de herramientas específicamente diseñadas para ayudar a pequeñas compañías y desarrolladores independientes a crear sus propios juegos de pelea.

Este es el ejemplo más cercano a lo que se realizó en este proyecto ya que una de las herramientas que posee es un sistema de inteligencia artificial que puede ser personalizado de acuerdo con el gusto del usuario con distintos criterios presentes en la plataforma, esto se puede ver en más detalle en la Figura 2.1. El único problema con esta herramienta es que todas sus funcionalidades están detrás de un pago y no disponibles a todo al público en general, además de que herramientas de este tipo son escasas y mientras más se puedan liberar al mundo, mejor.

[8]http://www.ice.ci.ritsumei.ac.jp/ ftgaic/index.htm
[9]http://www.ufe3d.com/doku.php/start

Figura 2.1: Criterios a personalizar en la inteligencia artificial de Universal Fighting Engine

Capítulo 3

Análisis y diseño de la solución

En este capitulo se describen las principales decisiones involucradas en el diseño del videojuego.

3.1. Herramienta

La herramienta en la cual se desarrolló el programa en la fase de implementación fue elegida en la etapa de diseño de la solución debido a que las decisiones sobre el PCMIA y el modelo que se utilice recaerán directamente en las posibilidades que entregue la herramienta escogida. Debido a esto, sumado a los puntos mencionados en el capítulo 2, la herramienta escogida fue Unity.

3.2. Plataforma de videojuego

Como se mencionó en capítulos previos del presente trabajo de título, el desarrollo del videojuego está inspirado en juegos de pelea clásicos como Street Fighter o Mortal Kombat. Dentro de las características de este tipo de juegos se encuentra estar animado en dos dimensiones (2D) y desarrollarse en un espacio estático donde dos personajes se enfrentan entre sí hasta que uno de ellos se queda sin puntos de vida.

Debido a que se busca integrar el videojuego con un modelo de inteligencia artificial, se descartó la funcionalidad multiplayer que suele tener este tipo de videojuegos. Por ello, todos los enfrentamientos ocurrirán entre el usuario y un personaje controlado por un modelo de inteligencia artificial (desde ahora PCMIA) previamente entrenado.

Figura 3.1: Plataforma de juego

3.2.1. Etapa

El videojuego diseñado mantendrá una dinámica estilo sandbox, es decir, un juego que se reinicia cada vez que uno de los personajes pierde debido a que no cuenta con niveles ni otras características que requieran de la implementación de un menú u otras funcionalidades. El objetivo del juego es reducir la vida del oponente a cero mediante golpes desde poca distancia.

La etapa corresponde a una arena que posee dos límites: uno derecho y otro izquierdo para prevenir que los personajes salgan del área designada. A su vez, se extiende por debajo un *collider* representando el suelo que previene que los luchadores caigan al vacío. La arena es la misma tanto para el entrenamiento como para las peleas entre el usuario y el PCMIA. Un ejemplo de la arena de juego se puede apreciar en la Figura 3.1.

3.2.2. Mecánicas del juego

Debido a la naturaleza simple del juego diseñado, se definieron sólo dos mecánicas: movimiento y ataque:

- **Movimiento:** En esta primera iteración se definieron movimientos únicamente en el eje horizontal, es decir, acercándose o alejándose del oponente. Esto se ejecutará por el usuario quién, utilizando las flechas izquierda y derecha o las teclas A y D en el teclado, podrá mover a su personaje a la izquierda o derecha respectivamente. Análogamente, las heurísticas de movimiento que tendrá el PCMIA utilizará las teclas Z y X respectivamente
- **Golpe:** Utilizando la tecla P del teclado el personaje controlado por el jugador ejecuta un golpe. Este si impacta con el jugador oponente le causa daño y pierde vida. En el

caso del PCMIA, la heurística asociada hace uso de la barra espaciadora (SPACE) para esta acción.

La decisión de asignar controles diferenciados entre el usuario y el PCMIA fue para evitar la interferencia de controladores en caso de requerir realizar pruebas dentro de la plataforma donde ambos tipos de luchadores estén presentes en la escena. Esto se puede visualizar en la figura 3.2.

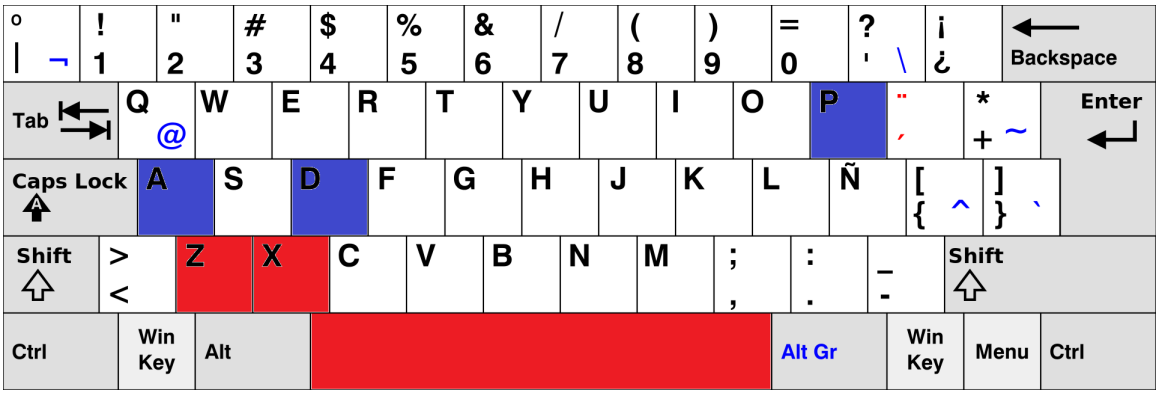

Figura 3.2: Rojo son las teclas que usa el MIA y azul son las teclas que usa el jugador

3.3. Modelo de inteligencia artificial

En etapas tempranas del proyecto se evaluó la construcción de un modelo de inteligencia artificial desde cero para el PCMIA utilizando los scripts en C# con los que funciona Unity. Sin embargo, debido a la falta de experiencia y poca información disponible, esta opción consumiría gran cantidad del tiempo del trabajo de título en desmedro de los otros objetivos.

Es por lo anterior, que se optó por utilizar una de las librerías mencionadas en el capítulo 2: ML-Agents.

3.3.1. MLAgents

Como se mencionó previamente, la librería MLA provee una implementación del algoritmos de aprendizaje reforzado *Proximal Policy Optimization (PPO)*. Este es configurable haciendo uso de los siguientes hiper parámetros.

- **Batch size:** El número de experiencias (Observaciones del agente, acciones y recompensas obtenidas) utilizadas en una iteración de **gradient descent**.
- **Buffer size:** Corresponde al número de experiencias que deben ser recolectadas antes de que se produzca un aprendizaje o actualización al modelo.

- **Learning rate:** Corresponde a cuán fuerte es cada actualización de gradient descent. Generalmente, se debe reducir si el entrenamiento es inestable y la recompensa no aumenta constantemente.
- **Epsilon:** Corresponde al umbral aceptable de divergencia entre las políticas antiguas y nuevas durante la actualización de gradient descent. Establecer un valor pequeño resulta en actualizaciones más estables, pero también hace el proceso de entrenamiento más lento.
- **Lambd:** Corresponde al parámetro **Lambda** usado para calcular el **Generalized Advantage Estimate (GAE)**. Esta variable se puede ver como cuánto el agente depende del valor estimado actual para calcular el valor estimado actualizado. Un menor valor significa depender más del valor estimado actual, mientras que un valor mayor corresponde a depender de las recompensas obtenidas en el ambiente.
- **Number of Epoch:** Es el número de veces que se pasa por el buffer de experiencia durante gradient descent.
- **Learning Rate Schedule:** Corresponde a cómo el learning rate decae con el tiempo.
- **Normalize:** Indica si se va a realizar una normalización a los valores del vector de observaciones.
- **Hidden Units:** Este valor corresponde a cuántas unidades existen en cada capa conectada de la red neuronal.
- **Number of Layers:** Corresponde al valor de cuántas capas escondidas van a estar presente después del input de observación.
- **Vis Encode Type:** Corresponde al tipo de codificación usada para las observaciones visuales.
- **Keep Checkpoints:** Este es el número de checkpoints que salvar una vez que ocurre una cierta cantidad de pasos.
- **Max Steps:** Corresponde al número máximo de pasos por los que la simulación pasa durante el proceso de entrenamiento.
- **Time Horizon:** Corresponde a cuánta experiencia por agente es recolectada antes que sea añadida al buffer de experiencia.

3.4. Personaje controlado por modelo de inteligencia artificial: PCMIA

3.4.1. Entrenamiento: Self-Play en AITraining room

En un juego de pelea, los personajes son iguales en función y forma, es decir, la implementación debe ser simétrica para los jugadores.

Lo anterior vuelve indistinguible a un personaje controlado por un usuario de un PCMIA. Por ello se puede entrenar al PCMIA utilizando versiones previas de sí mismo, estrategia conocida como self-play. La escena utilizada para el entrenamiento del PCMIA contra sí mismo se conoce como AITraining room.

Para poder habilitar Self-Play, los agentes a entrenar deben poseer el mismo espacio de observación y de acción, tener distintas identificación de equipo e incluir los hiperparámetros en el archivo de configuración del entrenador para que los utilice en lugar de aquellos por defecto. Se pueden observar los contenidos del archivo de configuración en la sección 3.4.2.

Sumado a lo anterior, Self-Play posee sus propios parámetros que son requeridos para activar sus funcionalidades:

- **Save steps:** Indica cada cuántos pasos se va a guardar una copia del modelo. Por ejemplo, si el número es diez mil, entonces después de diez mil pasos se va a guardar una copia del modelo actual.
- **Team change:** Indica cada cuántos pasos los objetos dentro del entrenamiento van a cambiar de id de equipo.
- **Swap steps:** Indica cada cuántos pasos se va a reemplazar el modelo contra el cual se esta jugando actualmente. Un número mayor nos dice que se va a jugar una cantidad mayor de veces contra un mismo modelo.
- **Window:** Corresponde al tamaño de la ventana donde se guardan los modelos pasados. Con un tamaño de ventana más grande existe una variedad mas amplia y antigua de oponentes contra los cual entrenar la inteligencia artificial.
- **Play against current self ratio:** Indica la probabilidad del agente de jugar contra sí mismo. Como es un valor probabilístico, su valor va en el rango 0 a 1.
- **Initial elo:** Un sistema alternativo de medición aparte de las recompensas de ambiente, usando múltiples juegos para medir la habilidad relativa de dos jugadores. Esta valor indica la cantidad inicial que va a tener la variable previo al inicio del entrenamiento

3.4.2. Valores de Hiperparámetros

```yaml
behaviors:
  FightingAI:
    trainer_type: ppo
    hyperparameters:
      batch_size: 32
      buffer_size: 10240
      learning_rate: 3e-4
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      learning_rate_schedule: linear
    network_settings:
      normalize: false
      hidden_units: 128
```

```
      num_layers: 2
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    keep_checkpoints: 5
    max_steps: 500000
    time_horizon: 64
    summary_freq: 50000
    threaded: false
    self_play:
        save_steps: 2000
        team_change: 10000
        swap_steps: 1000
        window: 20
        play_against_current_self_ratio: 0.5
        initial_elo: 1200
```

Estos fueron los valores que se decidió usar para los hiper parámetros en la fase de diseño, la mayoría de estos valores son los valores por defecto que recomienda la librería de ML-Agents en caso de ser la primera vez que se utiliza la librería y se realiza este tipo de configuración.

3.5. Controladores y recompensas

En aspectos de diseño se crearon dos controladores, uno para la inteligencia artificial y uno para el jugador. Estos controladores estaban encargados tanto de las acciones como de las interacciones de los luchadores. Entre estas funciones estaba el movimiento, el golpe, ser golpeado y devolver el escenario a su condicional inicial en caso de que un jugador se quede sin vida, entre otras funciones. Adicionalmente, se creó un script **Agent** que representa al jugador AI. Este script va incluido en el luchador controlado por la inteligencia artificial y es el que decide que acciones y observaciones tomar a la vez que comunicarse con el controlador para manejar las recompensas distribuidas por las elecciones que toma.

A continuación, en la tabla 3.1 se puede ver la distribución escogida para las recompensas de acuerdo a acciones y eventos que ocurren en base a estas acciones. Se escogió esta distribución pensando en dar un incentivo al MIA para que avance, trate de conectar golpes y gane, en general que presente un comportamiento agresivo.

Tabla 3.1: Tabla de distribución de recompensas en la fase de diseño

Distribución de Recompensas						
	Movimiento	Conectar Golpe	Fallar Golpe	Ser Golpeado	Ganar	Perder
Valor (Float)	+0.1	+1	0	-1	+2	-2

Capítulo 4

Implementación

En este capitulo se describen las decisiones principales tomadas en la implementación de los elementos necesarios para la construcción del juego.

4.1. Plataforma

Como se mencionó en el capítulo 2, el mecanismo para desarrollar en Unity corresponde a construir un conjunto de escenas que poseen una serie de objetos denominados *GameObjects* dentro de ellas.

Cada *GameObject* representa un elemento necesario para la ejecución del juego. A su vez, estos objetos pueden ser modificados utilizando, entre otros componentes, scripts en C#.

A continuación, se detallará la implementación de cada elemento de la plataforma, los cuales fueron separados utilizando el patrón de diseño Modelo-Vista-Controlador (MVC).

4.1.1. Escenas

Acorde al diseño presentado en el capítulo 3, fue necesario implementar dos escenas independientes para el correcto funcionamiento de la plataforma.

La primera escena corresponde al AITraining Room. En ella, se ejecuta la fase de entrenamiento donde el PCMIA se enfrenta contra si mismo. Se definió un número total de entrenamientos, correspondiente al hiperparámetro max_steps, de 500.000 veces.

La segunda escena fue estructurada de forma similar a la anterior, dos personajes enfrentados entre si en un espacio definido, con la diferencia de que solo un jugador corresponde a un PCMIA previamente entrenado y el otro jugador corresponde al usuario. Se puede observar en la figura 4.1 la vista que comparten ambas escenas.

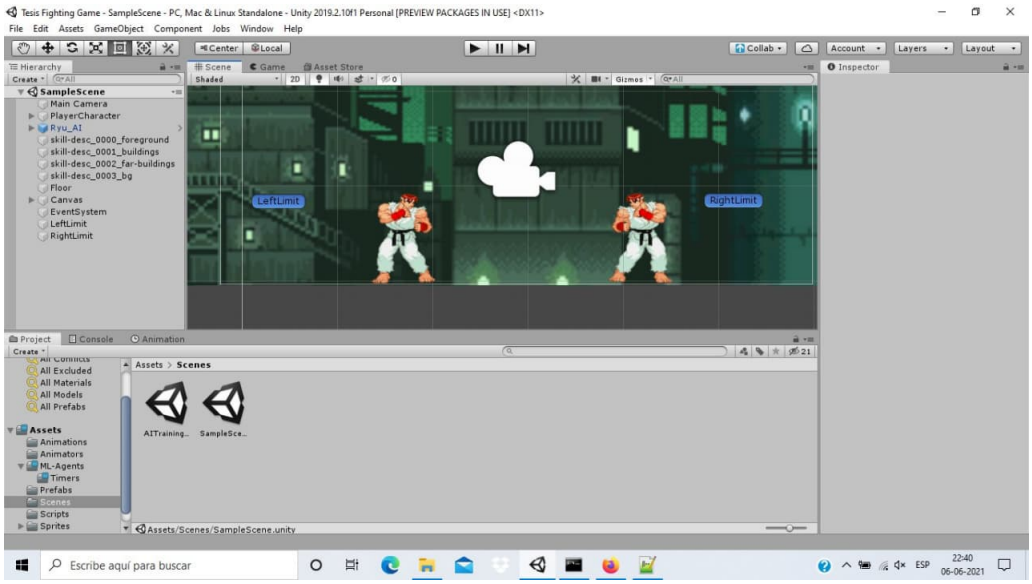

Figura 4.1: Escenas del juego

4.1.2. Luchador

Tanto el personaje utilizado por el usuario como el PCMIA corresponden a una instancia del mismo tipo de personaje, el luchador. Como se puede apreciar en la figura 4.2 el luchador está compuesto de las siguientes componentes:

- **RigidBody2D:** Es una componente que detecta colisión dentro del sistema e impide que dos objetos pasen uno sobre otro. En el caso de este juego se utiliza para que los personajes se mantengan sólidos y no pasen uno entre el otro, así como para que puedan colisionar con las murallas invisibles de la etapa y no salirse de esta.
- **Collider2D:** Existen dos instancias de esta componente dentro del personaje: cilíndrica y circular. Esto se debe a que una sola de estas no era suficiente para poder cubrir el área necesaria del modelo para poder registrar los golpes. La forma de estos colliders varía de acuerdo al tipo de animación que esté realizando el personaje.
- **GameObject:** Se tiene un GameObject dentro del luchador en la forma de su *hitbox*, usando una función dentro del controlador del luchador. La posición del objeto hitbox funciona para crear una circunferencia utilizada al momento de registrar golpes en los enemigos una vez que la hitbox hace contacto con un Collider de tipo AI, para facilitar este tipo de colisiones se opto por usar una circunferencia de radio 0.2 ya que este era el tamaño adecuado relativo a la distancia que el personaje debia acercarse para conectar un golpe.
- **PlayerController (Script):** Utilizado por el jugador humano para administrar las acciones del luchador. Utilizando los scripts de Unity se crea el controlador encargado de todas las acciones e interacciones del personaje.
- **AIController (Script):** Utilizado por la inteligencia artificial para administrar las acciones del luchador, estructurado de manera similar que el script PlayerController.

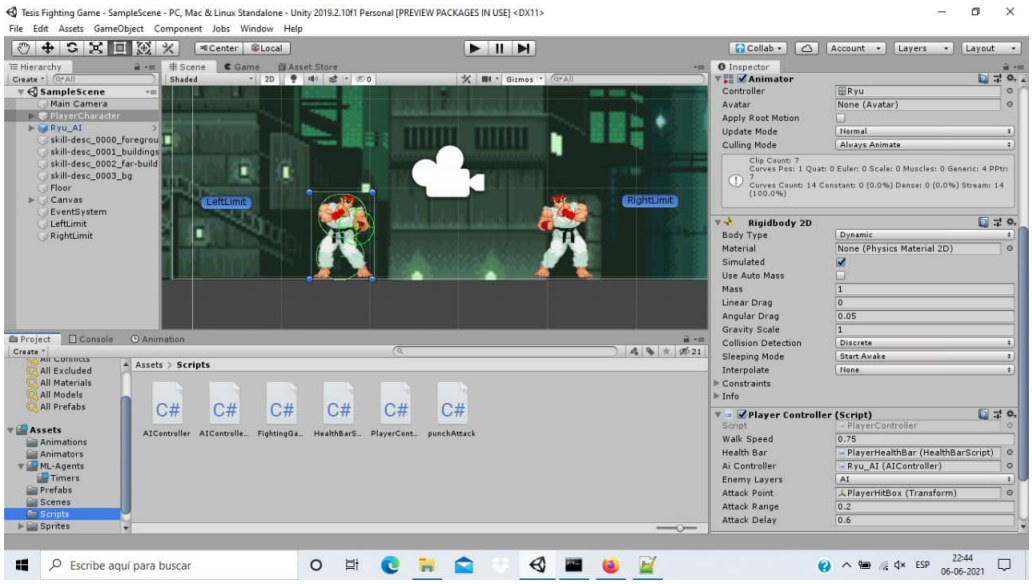

Figura 4.2: Luchador

- **FightingGameAIAgent (Script):** Script utilizado por la inteligencia artificial. Este es el script encargado de controlar el entrenamiento de la AI.

- **Decisión Requester:** Componente necesario para que el luchador controlado por la inteligencia artificial pueda reconocer los inputs del agent al momento de tomar decisiones.

Los componentes de tipo script serán estudiados en detalle en la subsección 4.2

4.1.3. Cámara

Por defecto, al crear una escena en Unity ésta viene con una componente de cámara, porque sin esta no hay forma de que el jugador pueda ver el juego. Se utiliza esta cámara ajustada para que pueda cubrir todo el largo de la arena sin dejar personajes fuera de la vista del usuario.

4.1.4. Límites Laterales

Son los bordes de la arena. En los límites izquierdo y derecho del escenario de juego existen dos GameObjects que solo poseen una componente Collider2D rectangular. Éste funcionan como paredes invisibles que previenen que los luchadores dentro de la arena caigan fuera de la etapa. En la figura 4.1 se pueden ver los límites en la forma de esos rectángulos de color azul.

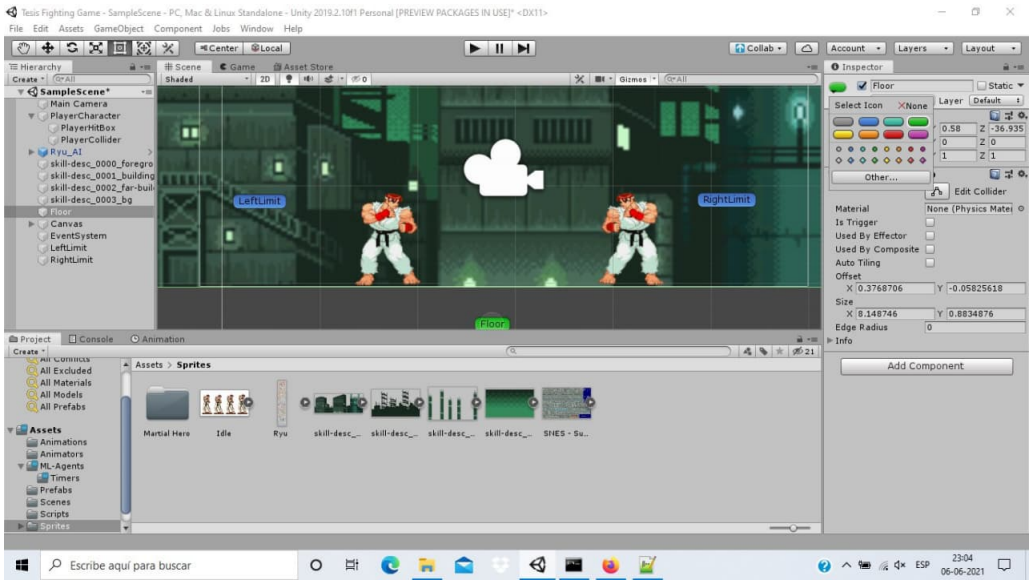

Figura 4.3: Piso

4.1.5. Arena

La arena estáti compuesta por un spritesheet encontrada de internet del sitio itch.io[1] en la sección de fondos gratis. El suelo está compuesto por un collider2D rectangular que impide que los personajes dentro de la arena caigan al vacío. Se puede ver la posición del piso dentro de la escena en la figura 4.3.

4.2. Mecánicas de juego

Al estar el juego restringido a un espacio 2D con el eje Z fijo, los personajes se pueden mover por el eje horizontal. Usando las teclas A y D el jugador puede mover al personaje de izquierda a derecha respectivamente. Esto se debe a que el método **Update** en el script de **PlayerController** tiene una condición que detecta y guarda la dirección del eje al momento de tipresionar un botón de movimiento. Por lo que si se trata de mover al personaje hacia la derecha, este va a devolver un valor positivo y viceversa si se trata de mover a la izquierda. De esta forma se puede diferenciar la dirección del movimiento y ajustar la velocidad del cuerpo y la animación que se debe reproducir de acuerdo a la tecla presionada.

El movimiento del luchador no tiene restricciones salvo en dos situaciones: estar ejecutando un golpe o estar recibiendo un golpe. Esto es para evitar situaciones injustas como poder cancelar ciertas animaciones en movimientos o incluso poder golpear de manera consecutiva. Para poder lograr esto, se puso dentro de las condiciones para poder realizar alguna acción que el luchador no estuviera recuperándose. Existen dos formas de estar recuperándose:

- **Hit stun:** Este es el tipo de recuperación que ocurre cuando el luchador es golpeado. La

[1]https://ansimuz.itch.io/industrial-parallax-background

duración esta alineada al tiempo que tarda en reproducir la animación al ser golpeado más un pequeño offset de tiempo adicional. Esto es para que no se pueda actuar de manera inmediata al momento de salir de la animación.

- **End lag:** Este tipo de recuperación ocurre cuando se ejecuta la acción de golpear. Esto es porque esta acción es clave para vencer al oponente en el juego, por lo que poner una restricción en la cantidad de golpes por segundo que puede dar el luchador es importante. Si no, la duración del juego sería muy poca y se transformaría en un juego en el que el que golpea primero gana. El tiempo que toma al luchador recuperarse es el mismo que la duración de su animación de golpear más un pequeño offset de tiempo para que no pueda actuar de manera inmediata.

El personaje controlado por la inteligencia artificial posee lógica idéntica en el manejo de situaciones que el del luchador controlado por un ser humano. Esto es porque uno de las principales condiciones para poder usar el algoritmo de **Self-Play** dentro de la librería de ML-Agents es que debe ser activado en un juego con adversarios simétrico.

Por motivos de evitar errores al momento de realizar pruebas en ambientes donde la inteligencias artificial y un jugador humano estén presentes. Los botones usados para poder controlar a un MIA son distintos a las utilizados para controlar un personaje humano.

4.3. Entrenamiento del PCMIA

Como ha sido mencionado anteriormente, el foco de este trabajo de título es la construcción de una inteligencia artificial funcional que pueda ser probada contra un jugador humano. De esta forma el foco del trabajo está puesto en la construcción y el entrenamiento de esta.

4.3.1. Agent

El primer paso para poder construir y entrenar una inteligencia artificial usando MLAgents es crear un script **Agent** importando las dependencias de la librería de MlAgents y extendiendo el script **MonoBehaviour** a la clase Agent. Una vez extendido el script se procede a sobreescribir las siguientes funciones de la super clase Agents

- **Initialize:** Esta función se ejecuta antes del inicio del primer episodio y es usada para fijar valores previo al inicio del entrenamiento. Para el caso de este proyecto se obtiene la posición local dentro de la escena del luchador y se calcula la distancia inicial entre el personaje y su oponente.
- **CollectObservations:** Esta es la función usada para recolectar las observaciones que se usan en la toma de decisiones. Estas observaciones están estructuradas en forma de un vector especial llamado **VectorSensor**. Este vector puede contener valores de distintos tipos como **floats**, **Bools** y otros tipos de vectores (Vector2 y Vector3). Para este proyecto se poseen tres observaciones: la posición del personaje, la posición del oponente y un valor Bool que determina si el oponente esta atacando.

- **OnEpisodeBegin:** Esta función es llamada al inicio de cada nuevo episodio. En este proyecto se utiliza esta función para restablecer los valores de posición y vida de los personajes junto con el valor de la distancia inicial entre ambos luchadores.

- **OnActionRecieved:** Esta es la función que determina qué acción va a tomar la inteligencia artificial en cada paso. La cantidad de acciones presente en el vector de acción que usa esta función depende de los valores puestos en el editor de Unity a la vez de que tipo de acción se va a usar, continua o discreta. Para los fines de este proyecto, se eligió un vector de tamaño 1 de acciones discretas que acepta solo cuatro valores, numerados del 0 al 3 que representan las acciones de moverse a la izquierda, derecha, no moverse y golpear. A su vez, dentro de esta función se calcula y actualiza la distancia entre los jugadores solo en caso de ser menor a la distancia previa.

- **Heuristic:** Finalmente, esta función hace una asignación de qué tecla va a cada acción que toma la inteligencia artificial y las añade al buffer de acciones. En el caso de este proyecto, como fue mencionado en la etapa de diseño, se asigna las teclas Z para el movimiento a la izquierda, X para el movimiento a la derecha y la barra espaciadora para el golpe, con una acción vacía quedando por defecto que representa el no realizar una acción. Todas estos elementos mencionados anteriormente fueron añadidos al buffer de acciones discretas que fueron usados en la función que toma acción.

4.3.2. AIController

A pesar de que el agente también sirve como una especie de controlador para la inteligencia artificial, por motivos de orden y buenas prácticas se separa las funciones de decisión de la inteligencia artificial del ejecutor de estas acciones. Por esta razón, se creó un script que funciona como el controlador de la inteligencia artificial, encargada cuando el agente tome una decisión, este la ejecute. Esta se pueda ejecutar sin generar errores visuales o comportamientos no deseados dentro del juego. Para lograr esto se construyó la función **ExecuteAction()** que de acuerdo a la elección numérica elegida por el agente esta es traducida a la acción "física" dentro del juego bajo ciertas condiciones.

- **Movimiento:** En un juego de pelea se tienen dos luchadores enfrentándose uno al otro viéndose de frente, por lo que los movimientos de estos son como mirar a un espejo. De esta forma, los valores numéricos de las velocidades y las animaciones de cada uno de los luchadores están invertidos de acuerdo al lado en que estén presente en la pantalla. Para lograr esto fue necesario crear identificadores para cada jugador durante el entrenamiento y así poder asignar correctamente los valores de velocidad y la animación a reproducir durante una orden del agente.

- **Restricciones:** De manera similar que con el controlador utilizado en el jugador controlado por una persona, no se pueden realizar acciones si es que se está bajo el estado de *hit stun* o *end lag*. Como fue dicho anteriormente, esto es para poder mantener una simetría dentro del juego y evitar problemas durante el entrenamiento.

4.3.3. Rewards

Para todo tipo de entrenamiento usando ML-Agents es necesario asignar recompensas por ciertas acciones o situaciones que ocurran durante el juego. Estas recompensas son en la forma de valores de tipo **float** que pueden ser positivos o negativos. Esto es para poder incentivar positiva o negativamente a la inteligencia artificial y poder guiar su comportamiento hacia uno deseado aprovechando que siempre quiere maximizar su recompensa.

Como se vio en el capítulo 3, la primera distribución de recompensas fue bajo la idea de un refuerzo positivo con el propósito de construir un MIA que reaccionara a las acciones tomadas por el jugador oponente, realizando maniobras ofensivas y defensivas cuando fuera necesario. De esta forma, tratando de incentivar este comportamiento y que existiera aproximación entre los jugadores. Se asignaron valores positivos a las recompensas dándole una mayor magnitud a las acciones y escenarios relacionados con conectar golpes y ganar el juego.

Los resultados no fueron completamente lo esperado. En vez de pelear, el MIA adoptó un comportamiento enteramente basado en movimiento y evitando el conflicto, ya que ser golpeado junto con perder toda la vida infligen una recompensa negativa sobre la inteligencia artificial. Por lo que para maximizar sus recompensas el MIA trato de evitar golpes y derrotas dándole un foco a pequeñas recompensas en un periodo largo de tiempo a través del movimiento

Se trató de arreglar los valores de recompensa de distintas maneras:

1. Cambiando la magnitud de los valores de recompensa

2. Removiendo/añadiendo valores de recompensa

3. Dándole más valores de recompensas a ciertas acciones

Esto nos llevo a la distribución de valores N°2 y 3 mostrados en la tabla 4.1. El resultado de usar la segunda fila de valores fue un MIA que, de manera similar al primero, priorizaba evadir al oponente pero rara vez trataba de entrar y conectar golpes. Esto se debe a que la forma de maximizar su recompensa era manteniendo el 0 evitando ser golpeado y si la oportunidad se presentaba conectar un golpe.

Finalmente, con los valores de la última fila se generó un MIA que no retrocedía inmediatamente, al disminuir su recompensa por movimientos, golpes fallidos, ser golpeado y perder mantenía su posición y trataba de conectar golpes. Una vez que el jugador se acercaba este empezaba a retroceder mientras seguía tratando de conectar golpes.

Dado el mejor rendimiento de la última distribución de valores se decidió utilizar el MIA creado utilizando estos valores al presentar un comportamiento que se acercaba al esperado.

Tabla 4.1: Tabla de distribución de recompensas en la implementación

Distribución de Recompensas						
	Movimiento	Conectar Golpe	Fallar Golpe	Ser Golpeado	Ganar	Perder
Valores N°1	+0.1	+1	0	-1	+2	-2
Valores N°2	0	+2	0	-1.50	+3	-3
Valores N°3	-0.001	+1	-0.001	-1	+2	-2

4.4. Entrenamiento

Para poder ejecutar el entrenamiento primero se ingresa al ambiente virtual. Una vez dentro se ingresa el comando:

```
mlagents-learn <trainer-config-file> --run-id=<run-identifier>
```

Donde:

- **trainer-config-file** Representa el **path** al archivo de configuraciones .YAML utilizado para definir los hyperparametros del algoritmo de entrenamiento.
- **run-id=<run-identifier>** Representa la identificación para el cerebro que fue creado durante el entrenamiento. Esto permite identificar la carpeta donde fue guardado el entrenamiento una vez este haya finalizado.

Se puede ver un ejemplo del uso de este comando en la cmd de Windows en la figura 4.4 esto muestra un logo de Unity las versiones de la librería de MLAgents que se usan dentro del proyecto y en el ambiente de python. También indica el puerto por el cual se esta conectando con el entrenador y da el mensaje de aprobación para ejecutar el entrenamiento en Unity.

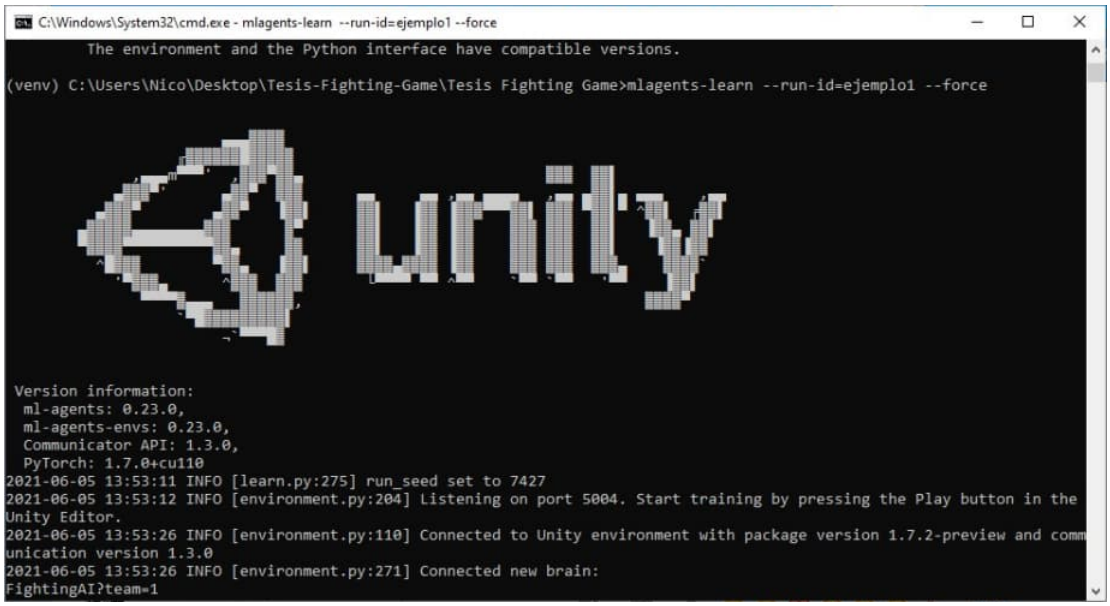

Figura 4.4: Comando de entrenamiento ejecutado en la cmd de windows

Una vez que aparece el mensaje de confirmación en el ambiente virtual se procede a iniciar el entrenamiento en Unity, como se puede ver en la figura 4.5. Lo único que hay que presionar aquí es el botón de play que reproduce la escena presente en la barra superior del centro de la ventana.

Figura 4.5: Reproducción del entrenamiento dentro del ambiente de Unity

4.4.1. Validación del Modelo

Para validar que el entrenamiento haya sido como lo esperado, se validaba el MIA creado jugando múltiples partidas contra él. Para tener una mejor evaluación de su comportamiento se adoptaban distintas estrategias que cambiaban el flujo de juego: aproximarse agresivamente, esperar en la posición original y retroceder.

Como el estado esperado del MIA es uno cuyo comportamiento sea difícil de distinguir del de un jugador humano bajo el contexto de este proyecto, que pudiera realizar acciones de acuerdo a los movimientos del oponente. En caso de no dar signos de agresividad cuando se retrocede o signos defensivos cuando se avanza se realizaba una alteración de los valores de recompensa, cambiando cuánto las acciones y situaciones de la tabla 4.1 le daban o cuestan al MIA. Una vez realizados estos cambios se volvía a empezar el entrenamiento siguiendo los pasos explicados en esta sección.

Pero el resultado final, como fue explicado en más detalle en la sección 4.3.3, es un MIA con un comportamiento muy defensivo que no se aproxima al oponente. En aspectos técnicos se logró implementar un MIA en un personaje dentro del juego y que este controlara y combatiera contra el jugador humano. Por otro lado, el comportamiento es a lo más la mitad de lo que se esperaba originalmente en este proyecto.

Capítulo 5

Validación

En este capítulo se da un ejemplo guiado sobre la prueba de concepto del proyecto. Los ejecutables del proyecto se encuentran en carpetas individuales dentro del servicio de guardado de archivos de *google*, *One Drive*[1][2], estos van a estar presente también dentro del anexo.

Posteriormente, se revisarán algunos resultados de la prueba con usuarios realizada. Los resultados finales de esta prueba se pueden encontrar en el Anexo.

5.1. Ejemplo guiado

5.1.1. Iniciando el Programa

Dentro de la carpeta **GameBuild** (**LinuxGameBuild** y **MacGameBuild** para otras plataformas) hacer click en el ejecutable **Tesis Fighting Game**. Como se puede ver en la figura 5.1 este abre una pantalla de configuración.

En esta pantalla la configuración ideal para jugar el juego es con resolución de 1280x720 con la opción de **Windowed** marcada. Se marca esta opción porque actualmente el juego no cuenta con una forma de cerrarlo internamente, por esto es mejor mantenerlo en modo de ventana para poder apagarlo al terminar.

Como el juego no es demandante gráficamente, se puede dejar las opciones de calidad de gráficos como están. El monitor donde se reproduce el proyecto se deja a preferencia del usuario. Una vez elegidas las opciones de configuración se hace click en el botón de **Play!** que está presente en la esquina inferior derecha para iniciar el proyecto.

[1]Windows: https://drive.google.com/drive/folders/1GjwDyPOTh95ZJDqYEq0q7BirSm3KFTKN?usp=sharing
[2]Linux: https://drive.google.com/drive/folders/1259L-E9igbgHCgaw0RIjHv-8XT0ZltVh?usp=sharing

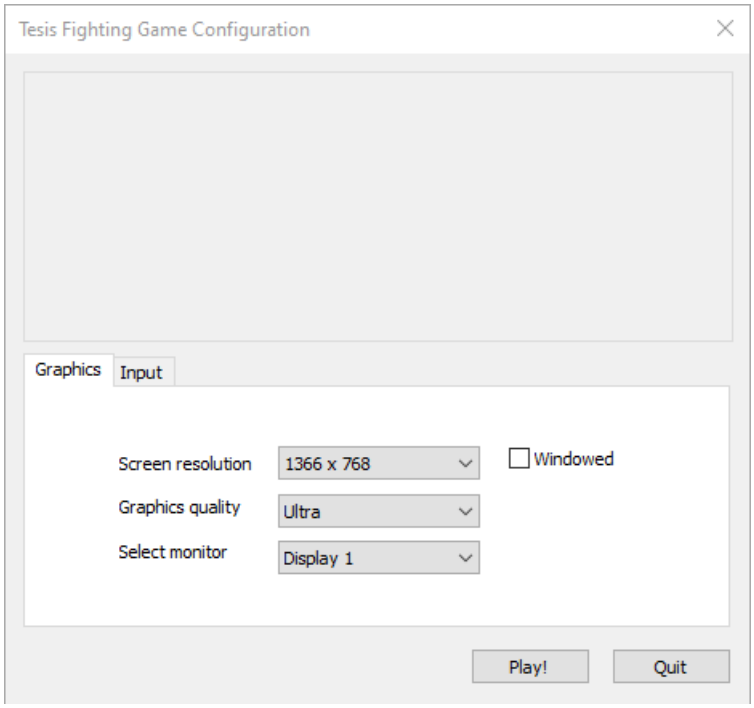

Figura 5.1: Menu de configuraciones de Tesis Fighting Game

5.1.2. Dentro del Juego

El objetivo del juego es vencer al oponente. Para poder realizar esto se debe vaciar la vida del oponente antes de que él vacíe la del jugador. Esto se logra utilizando las opciones de movimiento y ataque que posee el personaje. Estas se pueden ver en las siguientes figuras.

Las figuras 5.2, 5.3 y 5.4 representan el movimiento a la derecha del personaje, el movimiento a la izquierda del personaje y la acción de golpear del personaje respectivamente.

5.1.3. Vencer a la Computadora

El flujo de juego para vencer a la computadora se va a dividir en tres fases. Acercamiento, donde el jugador debe avanzar pero evitando el rango de ataque del oponente. Ofensiva, donde el jugador va a tratar de conectar un golpe sobre el oponente sin recibir contraataque. Y finalmente, retirada, donde después de realizar una ofensiva el jugador va a posicionarse para evitar comportamientos agresivos del oponente.

Una vez terminada esta última fase se va a partir el ciclo desde el principio. La idea es poder llegar a un escenario similar al mostrado en la figura 5.5. Pero en caso contrario y que el jugador sea el que este en riesgo de perder. El juego se reinicia una vez que la vida de alguno de los dos personajes llega a cero. Esto permite retar al PCC las veces necesarias hasta que el jugador se encuentre satisfecho o se rinda.

Figura 5.2: Movimiento a la derecha usando la tecla D

Figura 5.3: Movimiento a la izquierda usando la tecla A

Figura 5.4: Golpe usando la tecla P

Figura 5.5: Escenario antes de la victoria del jugador

5.2. Prueba con usuarios

Con el fin de determinar si se logró construir un PCMIA cuyo comportamiento pueda asemejarse al de un jugador humano, se realizó una prueba con usuarios donde se entregó un ejecutable del juego, para que hicieran una pequeña prueba de cinco a diez minutos y posteriormente respondieran un cuestionario[3].

5.2.1. Metodología

En esta sección se detallan los elementos principales de la prueba con usuarios.

Participantes

Se reclutó a 15 personas para participar en un estudio de usuario cuyo objetivo era determinar si el PCMIA construido en este proyecto poseía comportamiento semejante a un jugador humano. Por este motivo se reclutó gente con distintos grados de experiencias en videojuegos y que estaban en un rango etario de entre 23 a 28 años.

[3]Link Prueba de Concepto: https://docs.google.com/forms/d/e/1FAIpQLSeyPTtdeTAbCQ0vgEauDMqzJKDUn3CG3pV-5Sx9G556F-xfXQ/viewform?usp=sf_link

Materiales

Debido a las restricciones sanitarias asociadas a la pandemia de COVID-19 del año 2020, no fue posible realizar pruebas presenciales del juego. Por ello, se optó por la alternativa de enviar una versión ejecutable de juego con el PCMIA previamente entrenado a los usuarios junto con una lista de instrucciones básicas, información sobre los controles y un cuestionario que debía ser contestado al final de la prueba. El único material necesario para el usuario era tener un computador a mano cuyo sistema operativo fuera Windows o Linux. No se pudo realizar pruebas en Mac, ya que usuarios potenciales informaron que presentaron problemas en ejecutar el programa por motivos actualmente desconocidos, pero bajo investigación.

Procedimiento

Al ser entregado el build correspondiente del juego al usuario junto con el cuestionario, el procedimiento consistió en sesiones cortas de 5 a 10 minutos con el objetivo de derrotar al PCMIA. Se determinó que este período de tiempo era óptimo para la realización de las pruebas al considerar el espacio de atención de una persona entre 20 y 30 años y que el juego era simple. Además, como este se reinicia una vez que uno de los luchadores perdía toda su vida, había completa libertad con respecto a cuánto tiempo el usuario quería pasar en el juego.

Una vez que el usuario termina su sesión de juego, este procedía a contestar el cuestionario anexado al programa de prueba dejando su experiencia y comentarios con respecto al juego.

5.2.2. Resultados

De la prueba con usuarios se recopiló información tanto de la experiencia de juego, como del rendimiento del PCMIA y dificultad del juego. A continuación se van a exponer los resultados de esta información junto con los comentarios más relevantes con el resto presente en el anexo. Más adelante en la siguiente sección se va a realizar una discusión al respecto de los resultados y comentarios adicionales.

Preguntas

Empezando por el rango etario, los jugadores tenían en promedio 24.5 años de edad y un 13.4 % de los jugadores usaba Linux como su sistema operativo mientras que el resto utilizaba Windows. Del universo de personas que participaron en las pruebas, al menos 66.6 % jugaban entre 0 a 10 horas mientras que el resto (33.4 %) jugaba más que esto. Con respecto a la experiencia en juegos de pelea, la gran mayoría del universo estudiado (86.6 %) no jugaba estos juegos o no eran su principal elección de juego a jugar.

En aspecto de entretención, como se puede ver en la figura 5.6 sólo un 13.3 % del universo encontró el juego entretenido y a un 20 % no le causó una gran impresión el juego.

Las principales justificaciones con respecto al factor de entretenimiento del juego gravitan a que el juego es simple. Al carecer de más opciones que moverse y golpear, para los usuarios el juego se volvió rápidamente monótono, por lo que la idea de tratar de construir una plataforma simple por motivos de probar el PCMIA fueron a costa de la experiencia del usuario.

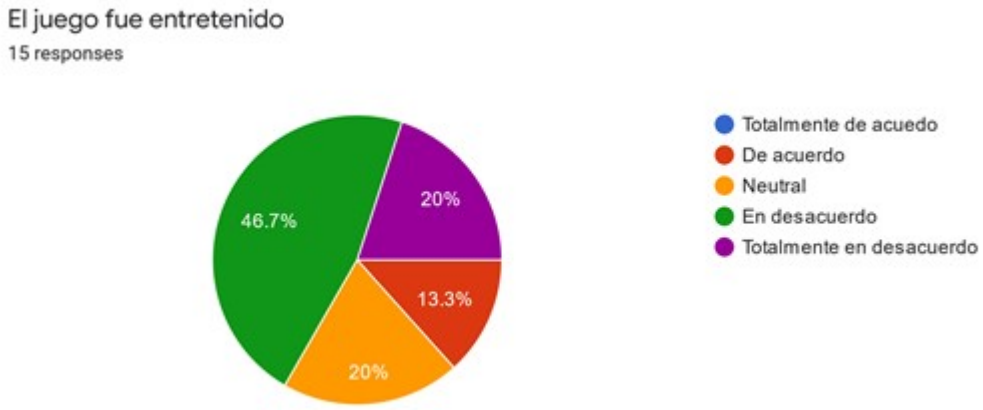

Figura 5.6: Distribución de entretenimiento con el juego

Con respecto a los controles del juego, un 60 % de los participantes encontró los controles cómodos e intuitivos. Esto significa que la distribución de los controles no les causó ningún malestar o sensación ajena. Entre las justificaciones se encontraba la facilidad de manejar al personaje y lo intuitivo que resultaban por la semejanza que poseían con controles más tradicionales de juegos. Se pude ver más detalle de la distribución en la figura 5.7.

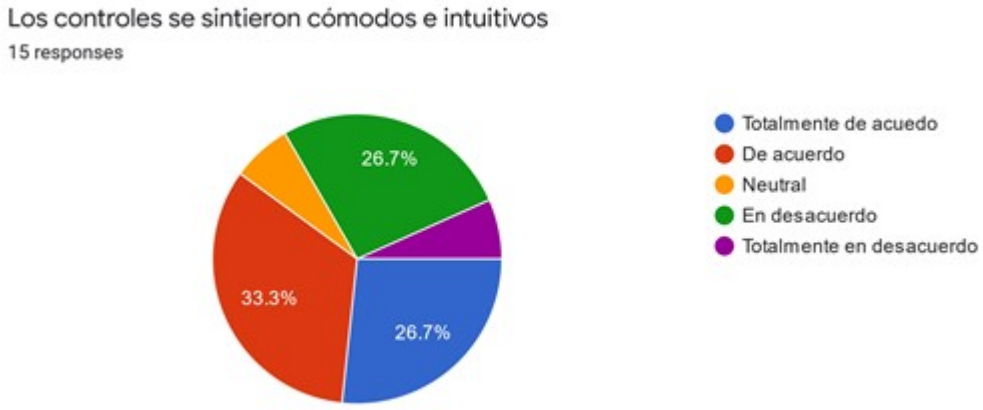

Figura 5.7: Distribución de comodidad de controles

Tratando de obtener información sobre si el juego funcionaba de manera natural, como se puede observar en la figura 5.8, un 40 % del universo de usuarios no se encontraba con una opinión fuerte al respecto. Esto significa que no poseían una referencia acerca del funcionamiento natural de un videojuego de pelea. Repasando las justificaciones de estas preguntas,

hubo una gran cantidad de personas que no entendieron bien la pregunta o respondieron bajo su propia lógica de qué se considera natural.

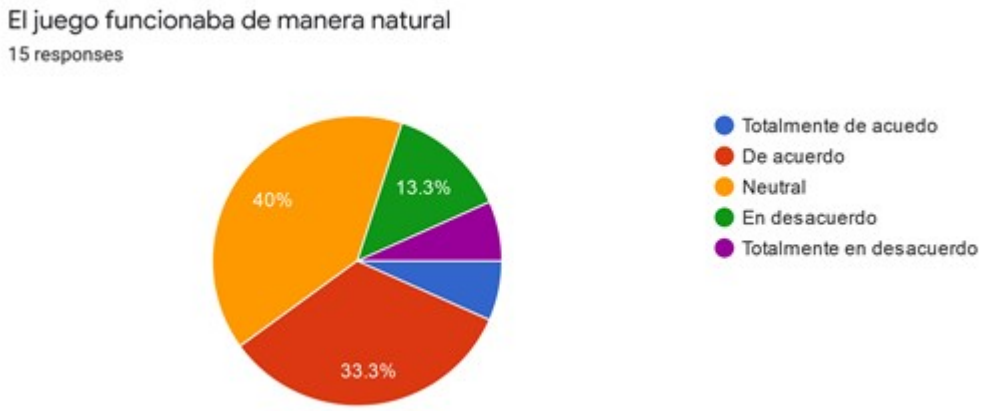

Figura 5.8: Distribución de funcionamiento natural del juego

Con respecto a la inmersividad de la plataforma, como se puede ver en la figura 5.9 un 53.4 % de la muestra de usuarios estuvo en desacuerdo con que la experiencia era inmersiva. Esto significa que el juego no logró captar su atención por períodos largos de tiempo. Viendo las justificaciones al respecto, una gran mayoría de las razones de por qué el juego no era inmersivo era por problemas con la plataforma, porque el juego era demasiado simple o porque faltaban elementos para fomentar inmersividad como música o efectos de sonido. Esto significa que la falta de elementos de ambiente producen un efecto negativo sobre la experiencia de juego del usuario.

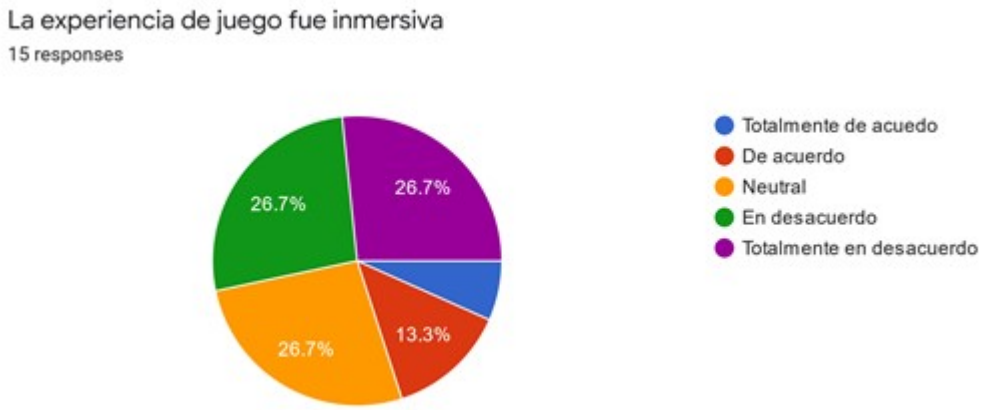

Figura 5.9: Distribución de inmersividad del juego

Viendo la dificultad del juego, un 46.7 % de la muestra de usuarios encontró el juego desafiante, mientras que un 20 % encontró que la dificultad estaba ni muy difícil pero tampoco muy fácil y un 23.3 % no encontró desafiante el juego. Esto se puede ver en más detalle en la figura 5.10. Para este caso, existe una gran variedad de justificaciones con respecto a la

dificultad, algunos atribuyéndolo al PCMIA, otros a la plataforma y finalmente algunos a su capacidad de ganar. Esto demuestra que el factor dificultad dentro de un juego no solo está limitado a cuán buena es la implementación del PCMIA, sino que también la propia habilidad del jugador o las mecánicas del juego.

Figura 5.10: Distribución de comodidad de controles

Finalmente, se va a ver si es que la experiencia de juego fue similar a jugar contra un ser humano. En la figura 5.11, solamente un 20 % de la muestra de usuarios estudiados estuvo de acuerdo con esta afirmación, mientras que un 46.7 % estuvo un desacuerdo y un 33.3 % tomó una posición neutral con respecto al tema. Esto implica que el PCMIA poseía un comportamiento lo suficientemente errático como para derrumbar su semejanza a un jugador humano. Al ver las justificaciones, se ve un comentario en común sobre el comportamiento del PCMIA y su incapacidad de comportarse agresivamente, sino solo estando parado o retrocediendo mientras golpea.

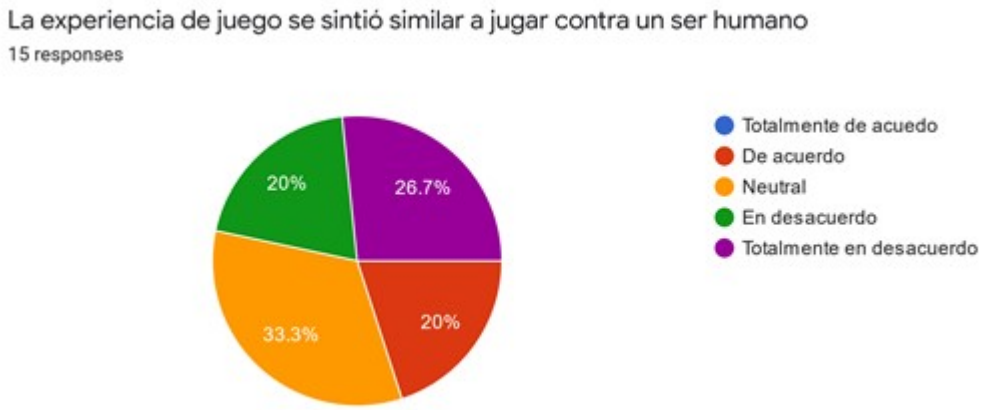

Figura 5.11: Distribución de experiencia contra la AI

Comentarios

La siguiente información fue recopilada de las ultimas dos preguntas del cuestionario **¿Qué nuevos elementos o características me gustaría ver en el juego?** y **¿Qué cambios le realizaría a las características actuales del juego?**:

- En general, el comportamiento del PCMIA se sentía muy poco natural en el sentido humano. Al estar estática en posición o retrocediendo y atacando periódicamente, es como si específicamente se hubiese entrenado a un luchador cobarde.
- El juego es demasiado simple. Se pide agregar más elementos como saltar y bloquear junto con ataques especiales, un sistema de combos y ventanas de hit stun más definidas e intuitivas.
- Además, a pesar de no ser estrictamente parte del gameplay loop, pero sí de la experiencia de usuario, tener efectos de sonido, música y distintas etapas sirve para aumentar el estado de inmersión del usuario.

5.2.3. Discusión

De los resultados obtenidos se puede determinar que el grado en que un usuario disfruta un juego de este género depende explícitamente de las opciones que posee dentro del videojuego junto con el tipo de adversario con el que se encuentra. En una gran mayoría de casos, por la simplicidad de la plataforma desarrollada, la profundidad del juego se ve afectada ya que no existen más opciones que las pocas que se ofrecen. Por casos como estos se ve que la idea técnica de hacer un juego simple para probar un PCMIA contra un ser humano puede afectar el factor de entretenimiento de este.

Con respecto a la plataforma, se obtiene que faltan elementos de inmersión como música y efectos de sonido. Estos elementos no fueron considerados para esta versión del proyecto porque el foco la experiencia jugando contra un PCMIA. Además, como en una gran mayoría de pruebas de usuarios se encontraron bugs o "exploits" con respecto al hit stun de los personajes. Este bug era abusado tanto por los jugadores humanos como por el PCMIA.

Finalmente, el PCMIA no llegó al desempeño máximo del proyecto que era demostrar comportamiento semejante a un ser humano al jugar el videojuego. Sin embargo, se logró dar un reto desafiante a la mayoría de los usuarios utilizando los métodos de entrenamiento descritos en este documento al tener que encontrar oportunidades ofensivas en un jugador claramente defensivo.

Para lograr la creación de un MIA que logre desafiar al 100 % de los usuarios. Se propone como trabajo futuro crear más ataques y movimientos dentro del juego. Específicamente, un ataque en la forma de un proyectil y permitir a los personajes saltar como mínimo. Esto le va a añadir más complejidad al juego, lo que permitirá expandir el rango de observación y las acciones que puede realizar el agente.

Además de esto, modificar valores de los hiper parámetros, específicamente los que están

encargados de la cantidad de pasos antes de la creación de un modelo y el epsilon responsable de que los modelos no presenten diferencias muy drásticas de un paso al otro.

Con estos cambios también se debe distribuir las recompensas obtenidas por acción nuevamente, al crear nuevas opciones de movimiento se debe evaluar qué tipo de recompensas se van a tener que asignar a estas nuevas acciones, cómo esto afecta el comportamiento del MIA y si es que hay que refactorizar las recompensas asignadas a movimientos antiguos.

5.2.4. Limitaciones

En esta sección se va a hablar y reconocer los factores que limitaron los resultados presentados en este proyecto.

Empezando por el número de personas que probaron el juego y contestaron el cuestionario, la principal causa de este problema fue la pandemia global de COVID-19 que esta ocurriendo actualmente. Al no poder realizar un estudio formal en un espacio dedicado con medios de reclutamiento más directos, se tuvo que optar por un método de aproximación más informal utilizando redes sociales.

En el escenario de que la pandemia continúe, la utilización de redes sociales para realizar el estudio va a ser imperativo. Utilizar todas las redes populares e incentivar a usuarios a compartir la prueba de manera agresiva para así causar un efecto de bola de nieve con el numero de usuarios, haciendo por ende una prueba más significativa.

Otro punto qué considerar es con respecto a la pregunta del cuestionario, “El juego funcionaba de manera natural”, esta pregunta causó confusión en las personas al no estar muy claro que significaba “de manera natural” en este contexto.

El contexto para esta pregunta era ”si se sentía como un videojuego de pelea de manera natural”, tanto en aspectos visuales como en forma de jugarlo la idea es que los usuarios sintieran que estaban jugando un juego de pelea. Reformar la pregunta agregando el contexto es una solución a la confusión de los usuarios pero no soluciona la limitación central.

Como el juego consiste de solo una escena y faltan varios elementos que son necesarios para este (musica, efectos de sonido, efectos visuales, etc.), para que el juego pueda ser evaluado en su totalidad como uno del género de pelea. Se propone como solución agregar estos elementos antes de realizar otro estudio formal.

Capítulo 6

Conclusión y Trabajo Futuro

La utilización de MIA en los medios de entretenimiento, más específicamente en los videojuegos, no ha pasado a muchos géneros más allá qué los juegos de estrategia. Uno de los géneros que se beneficiaría más de la aplicación de MIA en sus PCC es el de pelea, al no poseer una buena historia en el comportamiento de sus PCC contra jugadores humanos, siendo estos o muy desafiantes o muy fáciles de enfrentar. Dado este contexto, el problema de la falta de aplicación de MIA en juegos de pelea fue abordado por este trabajo de título.

A continuación se va a dar un pequeño repaso de los objetivos específicos propuestos, además del grado de completitud de cada uno:

1. **Desarrollar una plataforma funcional que cumpla con las características mínimas de jugabilidad y robustez de un juego de pelea:** El objetivo fue completado con éxito, la plataforma cuenta con una arena con limites definidos, barras de vida para los personajes y dos modelos capaces de moverse horizontalmente, atacar con un golpe, ser dañados y perder el juego.
2. **Integrar un modelo de inteligencia artificial encargado del comportamiento de los personajes controlados por computadora:** El objetivo fue completado con éxito, utilizando la librería ML-Agents se logró integrar un MIA a un PCC y que controlara al personaje asignado.
3. **Configurar los parámetros para que el PCMIA tome una actitud ofensiva o defensiva dependiendo de los inputs del jugador:** No se logró cumplir con este objetivo ya que el PCMIA demostraba un comportamiento puramente defensivo ante cualquier acción realizada por el jugador

La cantidad de objetivos específicos logrados fueron dos de tres, por ende se considera el objetivo general de desarrollar un videojuego de pelea, donde el personaje adversario está controlado por un modelo de IA, y buscar explorar cómo impacta esto en la experiencia de juego del usuario como parcialmente logrado.

Entrando a más detalle, las razones de por qué el último objetivo no fue cumplido se debe a diversos factores. Entre ellos el ambiente de entrenamiento, los parámetros usados y

posiblemente la distribución de recompensas no eran las óptimas para un proyecto de este tipo. A su vez, puede que al ir por un videojuego de pelea simplificado haya hecho que el entrenamiento nunca fuera correcto, ya que el ambiente mismo incentivaba el comportamiento mostrado. Por otro lado, la metodología para acercarse al comportamiento esperado funciona, esto se debe a que la primera iteración de PCMIA retrocedía inmediatamente al momento de empezar el juego y trataba de ganar recompensas moviéndose y evitando al oponente. En cambio al cambiar los valores de recompensa y asignando nuevos a ciertas situaciones y acciones se logro un acercamiento considerable al comportamiento esperado.

Una solución a este problema, propuesta como trabajo futuro, para el cumplimiento del tercer objetivo y la completitud total del objetivo general es agregarle complejidad al juego en la forma de tener más opciones de movimiento (saltar, agacharse), darle opciones defensivas (bloquear) y agregar más tipos de ataques (un proyectil y un ataque que tenga más rango pero que sea más lento). Al agregar estas opciones también se propone agrandar el espacio de observación para que estas nuevas opciones sean incluidas en la toma de decisión del PCMIA.

Para lograr una mejor recepción del juego ante los usuarios y que se entienda naturalmente que se trata de un videojuego de pelea se propone agregar elementos básicos para mejorar la calidad de la experiencia de juego. Algunos de estos elementos siendo: musica, voces, efectos de sonido, un contador de victorias y finalmente tiempo, para así darle otra condición de victoria al videojuego.

Finalmente, y siguiendo las propuestas de trabajo futuro, se propone crear otras dinámicas de juego, específicamente un modo PCMIA contra PCMIA donde se podrán probar distintos agentes peleando uno contra otro y se podrá medir su efectividad contra el resto. A su vez, como este se trata de un videojuego de pelea, se propone crear un modo jugador contra jugador para que dos personajes controlados por humanos combatan entre ellos.

Bibliografía

[1] Alena Denisova and Paul Cairns. The placebo effect in digital games: Phantom perception of adaptive artificial intelligence. In *Proceedings of the 2015 Annual Symposium on Computer-Human Interaction in Play*, CHI PLAY '15, page 23–33, New York, NY, USA, 2015. Association for Computing Machinery.

[2] I. Gajardo, F. Besoain, and N. A. Barriga. Introduction to behavior algorithms for fighting games. In *2019 IEEE CHILEAN Conference on Electrical, Electronics Engineering, Information and Communication Technologies (CHILECON)*, pages 1–6, 2019.

[3] Edgar Galván and Peter Mooney. Neuroevolution in deep neural networks: Current trends and future challenges. `https://arxiv.org/abs/2006.05415`, Jun 2020. Last visited: Late 2020.

[4] Zoltán Lorincz. A brief overview of imitation learning. `https://smartlabai.medium.com/a-brief-overview-of-imitation-learning-8a8a75c44a9c`, Sep 2019. Last visited: Late 2020.

[5] G. Martínez-Arellano, R. Cant, and D. Woods. Creating ai characters for fighting games using genetic programming. *IEEE Transactions on Computational Intelligence and AI in Games*, 9(4):423–434, 2017.

[6] Błażej Osiński. What is reinforcement learning? the complete guide. `https://deepsense.ai/what-is-reinforcement-learning-the-complete-guide/`, Jan 2021. Last Visited: Late 2020.

[7] Tommy Thompson. The killer groove: Shadow ai of killer instinct. `https://towardsdatascience.com/killergroove-c4c606601a8a`, Jun 2017. Last Visited: Late 2020.

[8] S. Yoshida, M. Ishihara, T. Miyazaki, Y. Nakagawa, T. Harada, and R. Thawonmas. Application of monte-carlo tree search in a fighting game ai. In *2016 IEEE 5th Global Conference on Consumer Electronics*, pages 1–2, 2016.

Apéndice A

Preguntas Abiertas Cuestionario

A continuación, se presentan las tablas A.1, A.2 y A.3 con las respuestas completas a las preguntas abiertas del cuestionario realizado en la prueba de concepto.

Tabla A.1: Tabla con las respuestas a la pregunta ¿Que nuevos elementos o características me gustaría ver en el juego?

¿Que nuevos elementos o características me gustaría ver en el juego?
Si como ser humano cuenta un niño de 8 años que no sabe jugar, si
-Poder saltar y agacharse, ademas de agregar patadas y combos(?) -Agregar algo para marcar las victorias/derrotas
Agregaría salto y bloqueo al juego. También, agregar que puedan haber ataques altos, bajos y medios.
Me gustaría poder golpear en otra dirección y saltar. También sería entretenido que hubiera ataques especiales, sonidos al atacar y música de fondo.
El adversario tiene una buena estrategia para las acciones que se pueden tomar
Poder elegir el personaje, agregar otros movimientos (saltos, patadas, etc), sonido.
Saltar y agacharse
principalmente más opciones de golpes y movimiento (saltar?). Si la única opción es un puño y moverse de izquierda a derecha relativamente lento, entonces la opción más viable siempre va a ser no moverse y golpear lo más rápido posible
Un menú seria lo mas útil, para que la entrada y salida del juego sea natural. Segundo sería agradable incluir en este un tutorial de las teclas, o un tutorial temporal en la primera partida del juego que desapareciera luego.
Los movimientos de la inteligencia artificial no eran muy naturales.
Un contador para saber cuantas veces perdía o ganaba y poder saltar/defenderme.
Al menos un ataque más (una patada por ejemplo) y la habilidad de saltar.
Música, sonido al efectuar golpes, sonido de acertar golpes, voces, una entrada musical tipo openning
Efectos de sonido, Inteligencia artificial más adecuada y mejor tracking/hitboxes, blocking en el sistema de pelea.
hitboxes, más acciones

Tabla A.2: Tabla con las respuestas a la pregunta ¿Que cambios le realizaría a las características actuales del juego?

¿Que cambios le realizaría a las características actuales del juego?
Le agregaría más controles como saltos, bloqueos y un crouch
-Que el enemigo se mueva por ultimo hacia adelante y hacia atrás ocasionalmente de forma notoria, por que con los golpes que pegaba se sentía como si estuviera vibrando en el lugar. -El fondo es fome, así que también lo cambiaria por muerte o algo así
Le arreglaría el hitstun, y que el movimiento de los personajes sea más rápido.
Cuando resulta casi imposible vencer al enemigo el juego se vuelve menos entretenido, quizás debería tener un umbral máximo de qué tan bueno es el oponente.
Agregar música y acciones (como combos y otros)
Que los movimientos del oponente sean más acordes con la situación.
Que se pudiera saltar y agacharse, que el otro personaje fuese distinto al que es uno
Al recibir daño no debiera poderse tirar un golpe inmediatamente, debiese haber un momento donde el personaje sufre del golpe (hitstun). Quizás aumentar más la velocidad de movimiento serviría, de tal forma de poder moverse y golpear a un enemigo que solo lanza golpes constantemente (moviéndose y golpeando después de un golpe del enemigo y antes que tire otro)
La habilidad de saltar o bloquear, serían las principales de esa forma se podría aplicar un poco mas de estrategia
Agregaría sonido y más variedad de movimiento/ataques
Que el enemigo también se acerque a atacarte
El comportamiento del oponente.
La interface del juego
Las perfeccionaría en cualquier dirección en las que parezcan una mejora sobre lo que hay hoy día, y la verdad existen muchas direcciones.
hitboxes, mejor inteligencia artificial, mecánicas de golpe

Tabla A.3: Tabla con las respuestas a Comentarios y Sugerencias adicionales

Comentarios y Sugerencias adicionales
Seria util saber quien gano al final
No quiero escribirte un par de parrafos así que si quieres saber mi opinión pidemela, trataré de ir punto por punto
Me da la impresión de que la hitbox no coincide con la animación del personaje, algo que habitualmente en los juegos de pelea se considera muy importante.
Suerte

Apéndice B

Código en C#

En este capitulo va a presentar el código del controlador del MIA, el controlador del jugador y finalmente el agente importado de ML-Agents.

B.1. AIControler

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class AIController : MonoBehaviour
{
    [SerializeField]
    private float walkSpeed = 0.75f;
    private Animator animator;
    private float xAxis;
    private float yAxis;
    private Rigidbody2D rgbd;
    private SpriteRenderer spriterenderer;
    private int maxHp = 10;
    private int currentHp;
    public Transform attackPoint;
    public float attackRange = 0.2f;
    public LayerMask enemyLayers;
    public HealthBarScript healthBar;
    public PlayerController playerController;
    public AIController enemyController;
    public FightingGameAIAgent agent;
    private string currentAnimation;
    private bool isAttacking;
    private bool isAttackPressed;
    public bool playerIsAI;
    public bool isPlayer1;
    public bool isPlayer2;

    [SerializeField]
    private float recoveryDelay = 2f;
```

```
32
33      //Animation States
34      const string AI_IDLE = "AI_Idle";
35      const string AI_FORWARD_WALK = "AI_Forward_Walk";
36      const string AI_BACK_WALK = "AI_Back_Walk";
37      const string AI_PUNCH = "AI_Punch";
38      const string AI_HURT = "AI_Hurt";
39      const string AI_KO = "Ryu_KO";
40
41      // Start is called before the first frame update
42      void Start()
43      {
44          currentHp = maxHp;
45          animator = GetComponent<Animator>();
46          rgbd = GetComponent<Rigidbody2D>();
47          spriterenderer = GetComponent<SpriteRenderer>();
48      }
49
50      //Funcion para cambiar animaciones
51      public void ChangeAnimationState(string newAnimation)
52      {
53          if (currentAnimation == newAnimation) return;
54
55          animator.Play(newAnimation);
56          currentAnimation = newAnimation;
57      }
58
59      void AttackComplete()
60      {
61          isAttacking = false;
62      }
63
64      //Metodo que es llamado del enemigo para determinar si este esta
    atacando, usado para las observaciones.
65      public float IsEnemyAttacking()
66      {
67          if (playerIsAI)
68          {
69              if (enemyController.GetIsAttacking())
70              {
71                  return 1f;
72              }
73              else
74              {
75                  return 0;
76              }
77          }
78          else
79          {
80              if (playerController.GetIsAttacking())
81              {
82                  return 1f;
83              }
84              else
85              {
86                  return 0;
```

```
87              }
88          }
89      }
90
91      //Metodo auxiliar de IsEnemyAttacking que retorna la variable
    isAttacking del controlador
92      public bool GetIsAttacking()
93      {
94          return isAttacking;
95      }
96
97      //Funcion para da ar al personaje
98      public void GetDamaged()
99      {
100         isAttacking = true;
101         currentHp--;
102         healthBar.SetHealth(currentHp);
103         ChangeAnimationState(AI_HURT);
104         Invoke("AttackComplete", recoveryDelay);
105         agent.GiveReward(-1f);
106
107         if (playerIsAI) {
108             GiveOponentReward(1f);
109             if (currentHp == 0) {
110                 GiveOponentReward(2f);
111                 agent.GiveReward(-2f);
112                 EndEpisode();
113                 enemyController.EndEpisode();
114             }
115         }
116         else
117         {
118             if (currentHp == 0)
119             {
120                 agent.GiveReward(-1f);
121                 EndEpisode();
122                 playerController.SetUpEpisode();
123             }
124         }
125     }
126
127     //Funci n auxiliar para darle una recompensa al oponente en caso de
    conseguir dar un golpe
128     public void GiveOponentReward(float reward) {
129         enemyController.agent.GiveReward(reward);
130     }
131
132
133     //Funcion usada para ejecutar la accion de la AI desde el agent
134     public void ExecuteAction(string movement) {
135
136         if (movement.Equals("right") && !isAttacking) {
137             MoveRight();
138         }
139         else if (movement.Equals("none") && !isAttacking) {
140             NoMove();
```

```
        }
        else if (movement.Equals("left") && !isAttacking) {
            MoveLeft();
        }
        else if (movement.Equals("punch") && !isAttacking)
        {
            Punching();
            isAttacking = true;
        }



    }

    //Funcion auxiliar que se activa cuando la AI debe realizar un golpe
    void Punching() {

        isAttacking = true;
        ChangeAnimationState(AI_PUNCH);

        //detect enemies in range of attacks
        Collider2D[] hitenemies = Physics2D.OverlapCircleAll(attackPoint.
    position, attackRange, enemyLayers);

        //damage them
        if (hitenemies.Length > 0)
        {
            if (playerIsAI)
            {
                enemyController.GetDamaged();
            }
            else
            {
                playerController.GetDamaged();
            }
        }
        else
        {
            agent.GiveReward(-0.001f);
        }

        Invoke("AttackComplete", recoveryDelay);
    }

    //Funcion para que la AI se mueva hacia adelante,
    void MoveRight() {

        Vector2 vel = new Vector2(0, rgbd.velocity.y);
        vel.x = -walkSpeed;

        rgbd.velocity = vel;
        if (isPlayer2)
        {
            ChangeAnimationState(AI_BACK_WALK);

        }
```

```
196         if (isPlayer1)
197         {
198             ChangeAnimationState(AI_FORWARD_WALK);
199 
200         }
201         if (CalcDistance())
202         {
203             agent.GiveReward(0.001f);
204         }
205         else
206         {
207             agent.GiveReward(-0.001f);
208         }
209     }
210 
211     //Funcion para que la AI nose mueva,
212     void NoMove() {
213 
214         Vector2 vel = new Vector2(0, rgbd.velocity.y);
215         rgbd.velocity = vel;
216         ChangeAnimationState(AI_IDLE);
217         agent.GiveReward(-0.001f);
218     }
219 
220     //Funcion para que la AI se mueva hacia atras,
221     void MoveLeft() {
222 
223         Vector2 vel = new Vector2(0, rgbd.velocity.y);
224         vel.x = walkSpeed;
225         rgbd.velocity = vel;
226         if (isPlayer2)
227         {
228             ChangeAnimationState(AI_FORWARD_WALK);
229         }
230         if (isPlayer1)
231         {
232             ChangeAnimationState(AI_BACK_WALK);
233         }
234         if (CalcDistance())
235         {
236             agent.GiveReward(0.001f);
237         }
238         else
239         {
240             agent.GiveReward(-0.001f);
241         }
242 
243     }
244 
245     bool CalcDistance()
246     {
247         float prevDist = agent.prevDist;
248         float actualDist = agent.actualDist;
249         return (prevDist > actualDist);
250     }
251 
```

```
252     //Funcion auxiliar para crear el circulo de la hitbox
253     void OnDrawGizmosSelected() {
254         Gizmos.color = Color.green;
255         Gizmos.DrawWireSphere(attackPoint.position, attackRange);
256     }
257 
258 
259     public void SetCurrentHp(int value)
260     {
261         currentHp = value;
262     }
263 
264     public int GetMaxHp()
265     {
266         return maxHp;
267     }
268 
269     public void EndEpisode()
270     {
271         agent.EndEpisode();
272     }
273 
274     public int GetHp()
275     {
276         return currentHp;
277     }
278 }
```

B.2. PlayerController

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4 
5  public class PlayerController : MonoBehaviour
6  {
7      [SerializeField]
8      private float walkSpeed = 0.75f;
9      private Animator animator;
10     private float xAxis;
11     private float yAxis;
12     private Rigidbody2D rgbd;
13     private SpriteRenderer spriterenderer;
14     private int maxHp = 10;
15     private int currentHp;
16     public HealthBarScript healthBar;
17     public AIController aiController;
18     public LayerMask enemyLayers;
19     public Transform attackPoint;
20     public float attackRange = 0.2f;
21     private string currentAnimation;
22     private bool isAttacking;
23     private bool isAttackPressed;
```

```
    private Vector3 originalPosition;

    [SerializeField]
    private float attackDelay = 2f;

    //Animation States
    const string PLAYER_IDLE = "Ryu_Idle";
    const string PLAYER_FORWARD_WALK = "Ryu_Forward_Walk";
    const string PLAYER_BACK_WALK = "Ryu_Back_Walk";
    const string PLAYER_PUNCH = "Ryu_Punch";
    const string PLAYER_HURT = "Ryu_Hurt";
    const string PLAYER_KO = "Ryu_KO";

    // Start is called before the first frame update
    void Start()
    {
        originalPosition = this.transform.localPosition;
        currentHp = maxHp;
        animator = GetComponent<Animator>();
        rgbd = GetComponent<Rigidbody2D>();
        spriterenderer = GetComponent<SpriteRenderer>();
    }

    // Update is called once per frame
    void Update()
    {

        //Checking for inputs
        xAxis = Input.GetAxisRaw("Horizontal");
        //space Attack key pressed?
        if (Input.GetKeyDown(KeyCode.P))
        {
            isAttackPressed = true;
        }
    }

    //=====================================================
    // Physics based time step loop
    //=====================================================
    private void FixedUpdate()
    {
        //Check update movement based on input
        Vector2 vel = new Vector2(0, rgbd.velocity.y);

        if (!isAttacking)
        {
            if (xAxis < 0)
            {
                vel.x = -walkSpeed;
            }
            else if (xAxis > 0)
            {
                vel.x = walkSpeed;
            }
            else
            {
```

```
                vel.x = 0;
            }
        }

        if (!isAttacking)
        {
            if (xAxis > 0)
            {
                ChangeAnimationState(PLAYER_FORWARD_WALK);
            }
            else if (xAxis < 0)
            {
                ChangeAnimationState(PLAYER_BACK_WALK);
            }
            else
            {
                ChangeAnimationState(PLAYER_IDLE);
            }
        }

        //assign the new velocity to the rigidbody
        rgbd.velocity = vel;


        //attack
        if (isAttackPressed && !isAttacking)
        {
            isAttackPressed = false;

            if (!isAttacking)
            {
                isAttacking = true;


                ChangeAnimationState(PLAYER_PUNCH);
                //detect enemies in range of attacks
                Collider2D[] hitenemies = Physics2D.OverlapCircleAll(
    attackPoint.position, attackRange, enemyLayers);
                //damage them
                if (hitenemies.Length > 0)
                {
                    aiController.GetDamaged();
                }

                Invoke("AttackComplete", attackDelay);
            }
        }
    }

    void OnDrawGizmosSelected () {
        Gizmos.color = Color.green;
        Gizmos.DrawWireSphere (attackPoint.position, attackRange);
    }

    void AttackComplete()
    {
```

```
        isAttacking = false;
    }

    //=======================================================
    // mini animation manager
    //=======================================================
    void ChangeAnimationState(string newAnimation)
    {
        if (currentAnimation == newAnimation) return;

        animator.Play(newAnimation);
        currentAnimation = newAnimation;
    }

    public void GetDamaged()
    {
        currentHp--;
        healthBar.SetHealth(currentHp);
        ChangeAnimationState(PLAYER_HURT);
        Invoke("AttackComplete", attackDelay);
        ChangeAnimationState(PLAYER_IDLE);
        aiController.agent.GiveReward(1f);
        if(currentHp == 0)
        {
            aiController.EndEpisode();
            SetUpEpisode();
        }
    }

    public void SetUpEpisode(){
        transform.localPosition = originalPosition;
        currentHp = maxHp;
        healthBar.SetHealth(currentHp);
    }

    public bool GetIsAttacking()
    {
        return isAttacking;
    }
}
```

B.3. FightingGameAIAgent

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Unity.MLAgents;
using Unity.MLAgents.Actuators;
using Unity.MLAgents.Sensors;

public class FightingGameAIAgent : Agent {
    [SerializeField] public Transform enemyPlayer;
    [SerializeField] public AIController controller;
```

```
    private Vector3 originalPosition;
    public float prevDist;
    public float actualDist;

    public override void Initialize()
    {
        originalPosition = this.transform.localPosition;
        prevDist = Mathf.Abs(Vector2.Distance(transform.localPosition,
    enemyPlayer.localPosition));
    }


    public override void CollectObservations(VectorSensor sensor){
        sensor.AddObservation(transform.position);
        sensor.AddObservation(enemyPlayer.position);
        sensor.AddObservation(controller.IsEnemyAttacking());
    }

    public override void OnEpisodeBegin(){

        this.transform.localPosition = originalPosition;
        controller.SetCurrentHp(controller.GetMaxHp());
        controller.healthBar.setMaxHealth(controller.GetMaxHp());
        prevDist = Mathf.Abs(Vector2.Distance(transform.localPosition,
    enemyPlayer.localPosition));
    }

    public override void OnActionReceived(ActionBuffers actions){
        string action = "none";
        actualDist = Mathf.Abs(Vector2.Distance(transform.localPosition,
    enemyPlayer.localPosition));

        switch (actions.DiscreteActions[0]) {
            case 0: action = "right"; break;
            case 1: action = "none"; break;
            case 2: action = "left"; break;
            case 3: action = "punch"; break;
        }

        controller.ExecuteAction(action);
        if(prevDist > actualDist)
            prevDist = actualDist;
    }

    public override void Heuristic(in ActionBuffers actionsOut){
        int action = 1;

        if (Input.GetKey(KeyCode.Z)) action = 0;
        if (Input.GetKey(KeyCode.X)) action = 2;
        if (Input.GetKeyDown(KeyCode.Space)) action = 3;


        ActionSegment<int> discreteActions = actionsOut.DiscreteActions;
        discreteActions[0] = action;

    }
```

```
64
65      public void GiveReward(float reward){
66          AddReward(reward);
67      }
68
69  }
```