TRABAJO FIN DE GRADO

Animaciones físicas mediante aprendizaje por refuerzo

Physical animations using reinforcement learning

Autores

Sergio Abreu García y Daniel Álvarez Castro

Grado en Desarrollo de Videojuegos

Facultad de Informática, Universidad Complutense de Madrid

Dirigido por

Antonio A. Sánchez Ruiz-Granados

Convocatoria de junio, curso 2020/21

Resumen

En la mayoría de videojuegos, las animaciones de los personajes están predefinidas, los artistas las crean durante el proceso de producción y así se quedan, cómo una serie de movimientos inmutables que se realizan al margen de la física del mundo en el que se encuentran. Las animaciones basadas en físicas[10] vienen a solucionar este problema. Son igualmente creadas por artistas, pero dentro del juego las ejecutan personajes cuyos cuerpos están simulados mediante físicas realistas. Es decir, se simula cada parte del cuerpo cómo un objeto físico diferente, y se conectan entre ellas mediante articulaciones capaces de aplicar fuerzas, imitando el comportamiento de un cuerpo real. Aplicando las fuerzas adecuadas se puede hacer que estos personajes ejecuten las animaciones previamente creadas por los artistas. Esto consigue personajes que, no sólo pueden ver sus animaciones modificadas de manera natural ante perturbaciones externas, sino que también pueden interactuar con su entorno de forma realista[5]. El movimiento natural que se consigue con esta técnica produce en el jugador una sensación de realismo e inmersión que sería imposible con animaciones tradicionales. Sin embargo, simular un personaje de esta forma presenta dos grandes problemas: el equilibrio y la estabilidad. Si simplemente se reprodujesen estas animaciones tal cual fueron creadas, el personaje no se mantendría de pie y perdería su trayectoria ante la más mínima perturbación. Esto se puede solucionar aplicando fuerzas artificiales que aseguren una postura concreta, pero así se pierde gran parte del realismo propio de esta técnica de animación. Una solución mejor es utilizar aprendizaje automático para que el personaje aprenda a estabilizarse aplicando fuerzas en sus articulaciones, intentando que al hacerlo su postura se parezca lo máximo posible a la de la animación [21]. Esto es aún una frontera del desarrollo de videojuegos. En este trabajo exploramos el uso de aprendizaje por refuerzo para resolver la tarea del equilibrio en animaciones físicas. Para ello utilizamos *Unity*[28], uno de los motores de videojuegos más extendidos en la industria, con lo cual buscamos también comprobar cómo de accesibles son estas técnicas a día de hoy para los estudios de desarrollo.

Palabras clave: Animaciones físicas, animaciones en videojuegos, aprendizaje por refuerzo profundo, aprendizaje automático, Unity.

Abstract

In most videogames, character animations are predefined, artists make them during the production process and so they stay, as an immutable series of movements that are played apart from the physics of the world they're in. Physics-based animations[10] try to solve that problem. They're made by artists alike, but inside the game they're played by characters whose bodies are being simulated with realistic physics. Namely, each part of the body is simulated as a different physical object, and then they're connected by joints that can apply forces, mimicking the behavior of a real body. Applying the right forces one can make those characters play the animations previously made by the artists. And so you can achieve characters that not only can their animations be modified by external perturbations, but they can also interact with their environment in a more realistic manner[5]. The movement attained with this technique provokes a sensation of realism and immersion in the player that would be impossible with traditional animation techniques. Nevertheless, physically simulating a character this way presents two big problems: balance and stability. If animations were just played as they were made, the character wouldn't be able to stand or it would lose its trajectory when facing even the smallest perturbation. This can be fixed by applying artificial forces that ensure a specific pose, but at the cost of losing much of this technique's realism. A better solution is to use machine learning so the character can learn to balance applying forces at its joints, trying to reach a pose as close as possible to the animation[21]. This is still a frontier in game development. In this project, we explore the use of reinforcement learning to solve the task of balance in physics-based animations. For that we use *Unity*[28], one of the most renowned videogame engines, with which we also aim to test the accessibility of these techniques to game studios as of today.

Keywords: physics-based animation, animation in videogames, deep reinforcement learning, machine learning, Unity.

Índice general

Índice de figuras

Índice de cuadros

1. Introducción

1.1. Motivación

Una de las cosas que más llama la atención de un videojuego son sus animaciones [7]. Cómo los personajes se mueven, cómo interactúan con el mundo y entre ellos. Es algo que, como estamos tan acostumbrados a ver en el mundo real, notamos de inmediato cuando lo vemos representado en la pantalla. Cualquiera, sin saber nada de videojuegos, puede discernir entre buenas y malas animaciones a simple vista.

Los animadores, en un trabajo que requiere de muchas horas, habilidad y atención al detalle, se encargan precisamente de esto, de que el movimiento de los personajes sea fluido y resulte natural a ojos del jugador. En los últimos años se ha popularizado mucho la técnica de captura de movimiento (*motion capture*) [25], que consiste en digitalizar los movimientos de un actor, obteniendo animaciones que luego se pueden reproducir digitalmente, como se puede ver en la figura 1.1. Así se consigue un movimiento mucho más realista que con técnicas de animación tradicionales, aunque ello supone un gran coste para los estudios de videojuegos.

Sea cual sea el método utilizado para crear animaciones, todas ellas presentan un problema difícil de solventar, y es la falta de adaptabilidad a diferentes situaciones. Por mucho dinero y tiempo que tenga una compañía, nunca podrá crear animaciones todas las posibles interacciones que un personaje podría tener en un entorno, puesto que éstas son infinitas. Por ejemplo, si un personaje tiene que sentarse en una silla, la animación puede presentar infinitas variaciones dependiendo de la dirección desde la que se aproxime, el tamaño de la silla, la altura del respaldo, la uniformidad del terreno, la animación que estuviese realizando previamente, etc.

Para solventar este problema se utilizan técnicas de generación procedimental de animaciones. Éstas consisten en generar animaciones en tiempo real en base al entorno y la interacción que el personaje esté realizando [3]. En el caso de la silla, un algoritmo de generación procedimental de animaciones podría tomar en consideración todas esas variables y generar una animación que se adapte a la situación.

Pero incluso las animaciones generadas procedimentalmente siguen presentando otro gran problema, y es que son reproducidas al margen de la simulación física. La gran mayoría de videojuegos con animaciones utilizan también un motor de físicas que simula el mundo alrededor de los personajes, pero las animaciones quedan al margen de esta simulación. Por ejemplo, aunque el brazo de un personaje colisione con una mesa, la animación no se verá afectada, lo cual da lugar a una experiencia inconsistente que rompe, en parte, la inmersión del jugador.

La falta de conexión entre las animaciones y el mundo físico no resulta tan evidente a ojos de cualquiera, pero sigue siendo un aspecto a mejorar en la mayoría de juegos. Es por este motivo que se comenzó a experimentar con la simulación física de personajes, donde cada parte del cuerpo es un cuerpo sólido distinto, que están conectados entre ellos mediante articulaciones. Estos personajes se dieron a conocer cómo

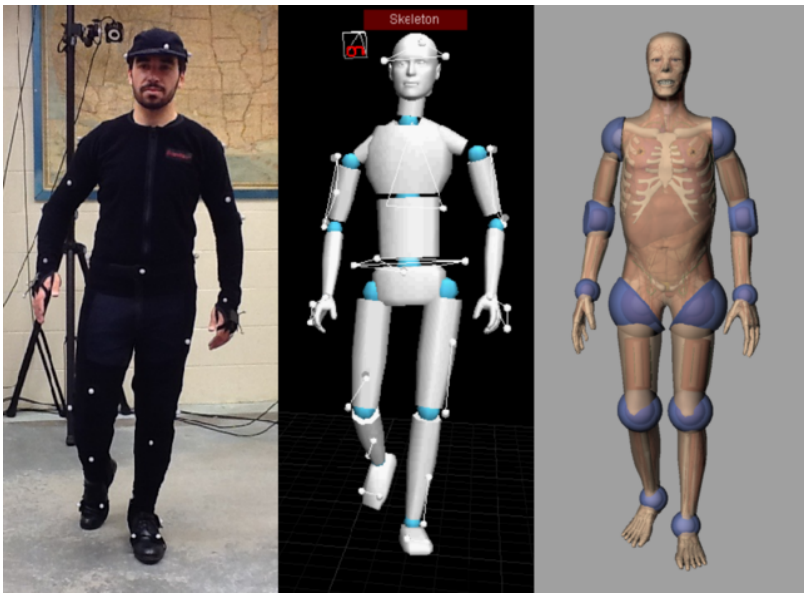

Figura 1.1: *Motion Capture*

Figura 1.2: *Ejemplo de personaje ragdoll en Unreal Engine*

ragdolls, del inglés 'muñeco de trapo', que es a lo que recuerda su comportamiento inerte [1], cómo se puede ver en la figura 1.2.

El siguiente paso resulta natural, animar *ragdolls* aplicando fuerzas en cada una de sus articulaciones, y es aquí donde empieza nuestro trabajo. La idea es, partiendo de un modelo humanoide, crear algunas animaciones básicas y configurarlo a su vez como *ragdoll*.

Por un lado tendremos un modelo animado de la forma tradicional, al cual denominaremos referencia, y que haremos invisible (aunque seguirá moviéndose). Por otro lado tendremos el *ragdoll*, cuyo objetivo será imitar el movimiento de la referencia, pero aplicando fuerzas en cada una de sus articulaciones. Para ello deberá no sólo aprender a igualar la pose de referencia en cada momento, sino a hacerlo manteniendo el equilibrio y adaptándose a los cambios que pueda presentar el entorno. En la figura 1.3 podemos ver ambos cuerpos en funcionamiento.

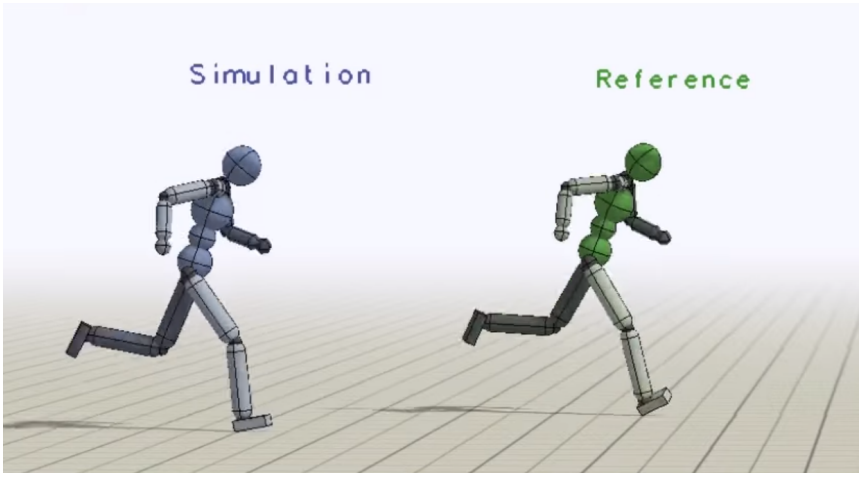

Figura 1.3: *Comparación entre referencia y ragdoll del artículo DeepMimic [21]*

Éste es un problema que ya se ha resuelto antes, pero la falta de robustez y la impredecibilidad de los modelos obtenidos limita su implementación en juegos comerciales. Nuestro objetivo es investigar esta tecnología y ver hasta dónde podemos llevarla utilizando el motor de videojuegos *Unity* [28]. Al ser este uno de los motores más utilizados en la industria, podremos también observar cómo de accesibles son estas técnicas para los estudios de desarrollo a día de hoy, así cómo las dificultades que presenta su uso.

1.1.1 Retos

Se trata de un problema muy complejo y que requiere de conocimientos avanzados tanto del funcionamiento de las físicas cómo de aprendizaje por refuerzo. Hay muy poca información al respecto, y la mayoría se concentra en forma de artículos académicos muy complejos realizados por grupos de expertos en la materia.

Por un lado, tenemos una red neuronal muy grande que necesita varias horas para entrenarse, por lo que probar cualquier cambio o posible mejora de la misma es un proceso que toma mucho tiempo, lo que limita en gran medida la velocidad a la que podemos realizar avances. Además, las técnicas de aprendizaje por refuerzo que utilizamos no son para nada sencillas, por lo hay que saber qué cambios hacer en el modelo para no perder el tiempo entrenando con una mala configuración.

Por otro lado, los resultados no dependen sólo del aprendizaje por refuerzo, sino que también se ven influenciados por las características del entorno. Ciertos parámetros como la gravedad del mundo, la frecuencia de actualización, el peso del modelo o la forma que tienen sus extremidades afectan en gran medida a cómo el agente tiene que actuar. Aquí entra en juego el conocimiento el motor de físicas y su funcionamiento interno, y saber configurarlo es tan importante como saber configurar la parte de aprendizaje por refuerzo.

Esto añade una capa de dificultad extra que hace difícil saber, ante los problemas que surgen durante el proyecto, si su origen se encuentra en el diseño de la red, en el

diseño del entorno, o en ambos.

En cuanto al aprendizaje por refuerzo, estamos ante un sistema de control continuo que posee muchos grados de libertad, 38 en el caso de nuestro modelo. Además, el movimiento de una sola articulación repercute mucho más allá de ella misma. Por ejemplo, un movimiento inadecuado del hombro puede generar una velocidad muy grande en la mano; o un pequeño error al girar el pie puede hacer que el personaje se caiga. Esto hace que sea difícil obtener con precisión el resultado que se busca, y a menudo nos hemos encontrado con inestabilidad en el proceso de aprendizaje y resultados con movimientos poco naturales muy lejos de los deseados.

1.1.2 Aplicaciones

Las tecnologías que utilicemos y nuestros potenciales resultados tienen evidentes aplicaciones en el desarrollo de videojuegos. Prueba de ello es el extendido uso de la generación procedimental de contenido, que permite a los desarrolladores dotar a sus juegos de una cantidad casi infinita de contenido único en el contexto en el que la utilicen; ya sea en generación de escenarios, eventos u otros elementos del juego [14]. La generación procedimental de animaciones ha comenzado a tomar en los últimos años un papel igualmente importante en la industria. Cada vez son más los juegos que implementan este tipo de técnicas, y esta tendencia seguirá hasta convertirse en un estándar más de la creación de videojuegos.

Sin embargo, sus aplicaciones van mucho más allá del entretenimiento, pues el uso de aprendizaje por refuerzo para control motor está fuertemente relacionado con la robótica, campo en el que también se están desarrollando técnicas similares [17] de manera paralela. Aunque esto va más allá del alcance de este proyecto, también nos resulta sumamente interesante, pues podría dar lugar a una nueva generación de robots capaces de moverse por cualquier terreno y recuperarse de cualquier perturbación.

Figura 1.4: Robot Atlas de *Boston Dynamics*, uno de los mayores referentes de control motor en robótica.

1.2. Objetivos

El objetivo final del proyecto sería conseguir desarrollar un agente que fuera capaz de seguir animaciones complejas, y que además tenga una resistencia considerable a perturbaciones externas. Sin en embargo, con este objetivo en el horizonte, hemos primero de cumplir objetivos intermedios escalonados:

1. **Investigar y aprender sobre las técnicas usadas para resolver este tipo de problemas** en trabajos relacionados. Esta es una parte fundamental del proyecto ya que sentará las bases teóricas sobre las que luego realizaremos toda la parte práctica.
2. **Preparar un entorno de pruebas funcional y consistente**, con un ragdoll físico bien desarrollado con el que seamos capaces de llevar a cabo las pruebas y entrenamientos.
3. El siguiente objetivo sería conseguir un agente que sea capaz de **seguir una animación que consista en mantenerse de pie en equilibrio**, de la forma más natural y estática posible.
4. A continuación nuestro objetivo es conseguir que nuestro agente sea capaz de **seguir animaciones sin desplazamiento**, es decir, que involucren algún movimiento del modelo pero que no requieran que este se desplace por la escena.
5. Acto seguido, intentaremos conseguir que el agente sea capaz de **seguir animaciones con desplazamiento**, con especial interés en que fuera capaz de andar naturalmente hacía delante, habiendo aprendido a mantener el equilibrio, seguir la animación y a desplazarse al mismo tiempo.

1.3. Planificación y metodología

Habiendo decidido como debería ser la división del trabajo para alcanzar los distintos objetivos, proponemos una posible planificación. Representada visualmente en la figura 1.5. Las primeras semanas las dedicaremos por completo a la investigación de trabajos relacionados y a la preparación del entorno. Los tres meses siguientes nos centraremos en la parte más difícil del trabajo, y por ende la más larga, obtener un primer agente capaz de quedarse de pie en equilibrio estático. Conseguido esto, dedicaremos alrededor de un mes a realizar los cambios oportunos para que ese modelo funcione también con animaciones dinámicas. Finalmente, en los dos últimos meses intentaremos que el modelo funcione también para animaciones dinámicas con desplazamiento.

Junto a nuestro director hemos acordado reuniones cada dos semanas durante todo del período en el que se desarrolla el TFG hasta la convocatoria de junio. Estas reuniones se harán utilizando *Google Meet* y tendrán como objetivo informar al director de los progresos desde la anterior reunión, además de plantear unos objetivos que intentar alcanzar para la próxima. Durante el último mes decidimos una reunión semanal para tener una crítica más constante de la memoria y poder obtener un mejor resultado final.

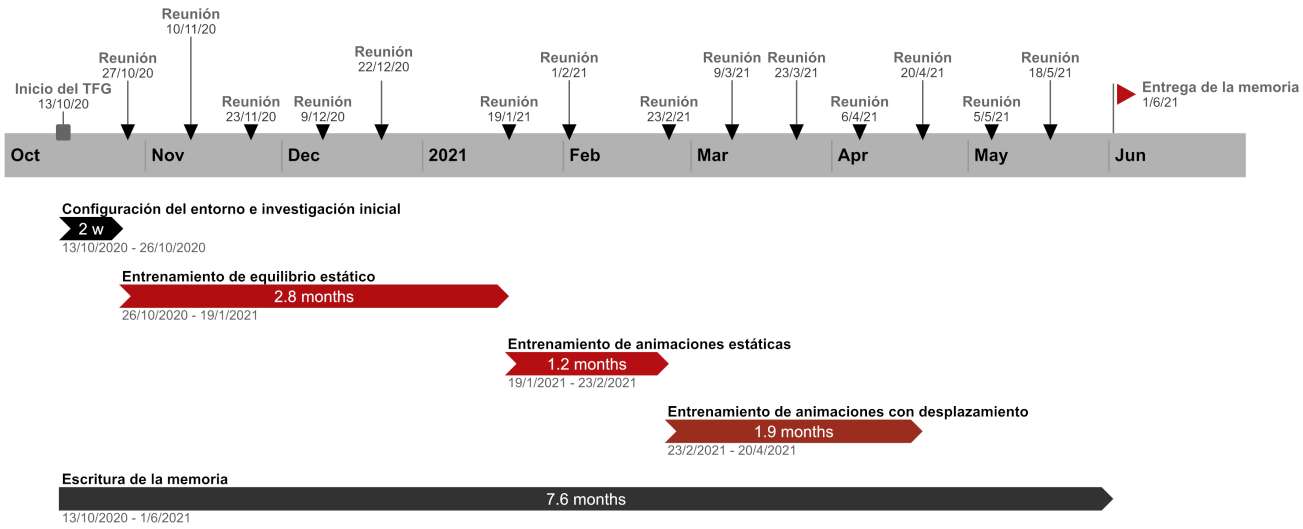

Figura 1.5: Línea de tiempo

Como sistema de control de versiones hemos utilizado *GitHub*, y como manera de compartir documentos *Google Drive*, el cual al principio también se utilizó para la escritura de esta memoria para posteriormente migrar a *Overleaf*.

[1]. [2]. [3].

Dado que sólo somos dos integrantes, mantener una comunicación diaria sobre el proyecto fue todo lo que necesitamos para organizarnos, acordando de la manera más razonable la carga de trabajo que le correspondía a cada uno en cada situación.

1.4. Estructura de la memoria

Esta memoria consta de 6 capítulos, cuyo contenido se describe a continuación:

En este capítulo 1 planteamos el problema al que nos vamos a enfrentar: por qué es importante, cuales son las dificultades que presenta, qué retos y objetivos tenemos, y cómo vamos a organizarnos para conseguirlos.

En el capítulo 2 se recopilan todos conocimientos teóricos sobre aprendizaje por refuerzo que hemos usado a lo largo del proyecto, así cómo los diferentes algoritmos que tenemos disponibles y sus fundamentos teóricos.

[1] www.github.com
[2] drive.google.com
[3] www.overleaf.com

En el capítulo 3 se describen cada uno de los diferentes elementos principales en los que se sostiene la parte práctica del proyecto, como el funcionamiento de la simulación física que usaremos o los funcionamientos de la herramienta de aprendizaje por refuerzo.

En el capítulo 4 explicamos el modelo final que hemos conseguido y las configuración utilizadas en todos los aspectos relevantes del proyecto, así como las decisiones de diseño que nos han llevado a ellas.

El capítulo 5 es una recopilación estructurada de todos los experimentos que hemos hecho a lo largo de los meses, dónde además explicamos las dificultades que nos hemos ido encontrando por el camino y cómo las hemos solventado.

El capítulo 6 expone las conclusiones principales que hemos obtenido tras el desarrollo del proyecto, acompañadas de una posibles ideas de cual podría ser el trabajo futuro para este proyecto.

1.5. Código fuente

Tanto el código fuente como todos los recursos que hemos utilizado pueden encontrarse en el siguiente repositorio github.com/sergioabreu-g/physical-animations bajo la licencia de uso Apache 2.0.

2. Aprendizaje por refuerzo y videojuegos

2.1. Introducción al aprendizaje por refuerzo

Aunque los orígenes del aprendizaje por refuerzo (o aprendizaje reforzado) se remontan a los comienzos de la inteligencia artificial, y a día de hoy sigue siendo una frontera de investigación en el campo del aprendizaje automático.

El aprendizaje por refuerzo RL es un campo del aprendizaje automático que estudia la habilidad de un agente de software para tomar decisiones según unos estímulos positivos y/o negativos llamados recompensas. La asociación entre recompensas y las acciones es lo que determina el comportamiento del agente.

En cada paso, el agente recibe cierta información del entorno, lo que se conoce como estado. Por ejemplo, un robot que aprende a mantenerse de pie podría recibir información de sus sensores para saber su grado de inclinación en los diferentes ejes, qué partes de su cuerpo están en contacto con el suelo, o qué rotación tiene en cada una de sus articulaciones; eso sería el estado. A continuación, el agente elige qué acción tomar ante ese estado. El robot podría tomar la decisión de, si detecta demasiada inclinación hacia delante, poner un pie al frente para frenarse; esa sería la acción. Ésta produce un cambio en el entorno, es decir, que se genera un nuevo estado con el que el agente podrá escoger una nueva acción y repetir el ciclo, como se puede ver en la figura 2.1.

Al mismo tiempo que se genera un nuevo estado, se genera también una recompensa. Esto es, un valor obtenido de una función arbitraria que define cómo de buena ha sido la actuación del agente. Para el robot, podríamos definir una función que diese un valor cercano a uno si el robot está de pie, y cercano a cero si está tumbado; utilizando las mediciones de inclinación de los sensores. Esta función de recompensa es la

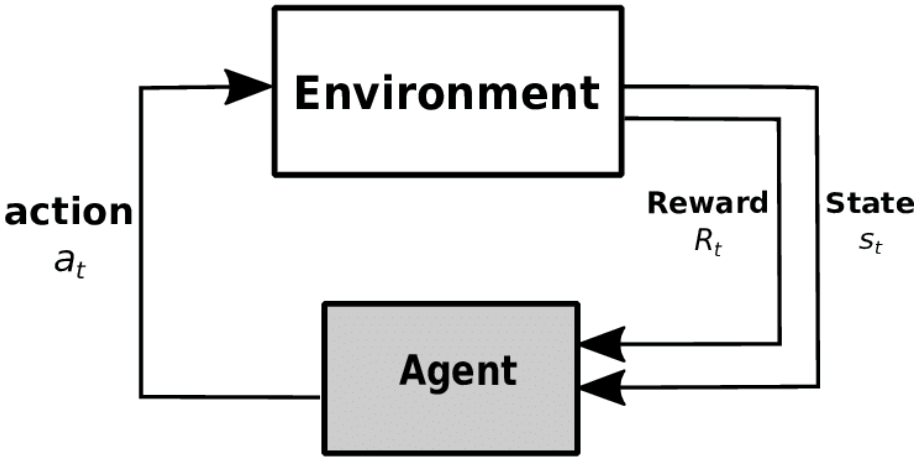

Figura 2.1: Aprendizaje por refuerzo

clave del aprendizaje por refuerzo, pues gracias a ella el agente puede evaluar cada una de sus acciones y aprender de ellas, modificando su comportamiento para potenciar aquellas acciones que maximicen la recompensa.

Algunos elementos clave de un problema de RL son:

- **Entorno**: realidad en la que opera el agente.
- **Estado**: situación actual del entorno.
- **Observación**: información parcial sobre el estado.
- **Acción**: respuesta del agente ante un entorno dado.
- **Trayectoria**: una trayectoria es una secuencia de pares [estado, acción] que describen la evolución del entorno y el agente según las decisiones que va tomando este último.
- **Recompensa**: valor que define cómo de bueno es el estado en el que se encuentra el agente, y que éste usa para buscar un comportamiento óptimo.
- **Política**: función que asocia a cada posible estado de un problema una distribución probabilística de acciones, de las cuales el agente escoge una en cada paso.
- **Valor**: recompensa futura que el agente espera recibir por alcanzar un estado específico o por realizar una acción desde éste.

La diferencia entre esta técnica de aprendizaje automático y el aprendizaje supervisado es que la primera no necesita ejemplos de entrada/salida previamente clasificados para poder aprender, sino que es el propio agente el que, guiándose por las recompensas obtenidas, busca los comportamientos óptimos. Esto hace que definir una buena función de recompensa sea uno de los pasos más importantes a la hora de obtener un modelo exitoso.

2.1.1 Proceso de decisión de Markov (MDP)

En RL, el entorno suele formularse como un proceso de decisión de Markov (MDP), un marco matemático que permite formalizar y describir el problema. Un MDP no es más que un conjunto de estados unidos por unas transiciones que suceden con una probabilidad concreta.

Para entender bien qué es un MDP, es importante primero conocer **la propiedad de Markov** [31], que dice lo siguiente:

"Dado el presente, el futuro es independiente del pasado."

Esto quiere decir que un estado S_t acumula toda la información relevante para determinar la probabilidades de transición al siguiente estado S_{t+1}, por lo que todos los estados anteriores pueden ser descartados.

$$P[S_{t+1}|S_t] = P[S_{t+1}|S_1...S_t]$$

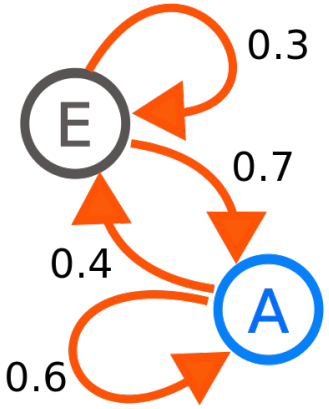

Figura 2.2: Cadena de Markov

Para llegar a entender qué es un MDP y cual es su base matemática, partiremos primero de un concepto más simple llamado **cadena de Markov**; una secuencia de estados aleatorios que cumplen la propiedad de Markov. Matemáticamente, se define cómo una tupla $\langle S, P \rangle$, donde:

- **S** es un conjunto de estados finitos
- **P** es una función probabilística de transición entre estados

En la figura 2.2 se muestra una cadena de Markov de 2 estados, dónde cada nodo es un estado y cada arista guiada la probabilidad de transición de un estado a otro. La matriz de transición del ejemplo anterior es la correspondiente a esta cadena de Markov.

El siguiente paso sería incorporar a una cadena de Markov la información correspondiente a la función de recompensa. Es decir, necesitamos asociar una recompensa a cada estado. Esto se conoce como **proceso de recompensa de Markov** (del inglés *Markov Reward Process*). Matemáticamente, se define como una tupla $\langle S, P, R, \gamma \rangle$ donde:

- **S** es un conjunto de estados finitos.
- **P** es una función probabilística de transición entre estados
- **R** es una función de recompensa.
- γ es un factor de descuento.

El objetivo aquí es **maximizar el retorno** en cada paso, el cual se define como:

$$G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

El factor de descuento γ es un valor entre 0 y 1 qué determina la importancia de las acciones futuras. Como se puede ver en la fórmula, un valor pequeño de γ hará que la recompensa se haga pequeña rápidamente a medida que aumenta k, haciendo que las acciones cercanas a k=0 tengan mucha más influencia en el resultado del sumatorio. Al revés, un valor grande de γ hará que la recompensa decaiga más lentamente al

incrementarse k, por lo que más estados tendrán una influencia significativa sobre el resultado.

Esta variable es muy importante en aprendizaje por refuerzo, puesto que nos permite decidir si queremos un agente que valore más la recompensa a corto o a largo plazo. La recompensa a corto plazo (γ pequeño) es mucho más segura, puesto que se puede obtener casi al instante con muy pocas acciones. Sin embargo, puede que realizando otras acciones diferentes obtenga una mayor recompensa a largo plazo (γ grande), pero quizás suceda algún evento imprevisible entre medias que le impida conseguirlo. En la mayoría de problemas es beneficioso usar valores de γ grandes (superiores a 0.9) para alcanzar políticas óptimas y evitar el estancamiento en máximos locales.

La función de estado-valor asigna a cada estado un valor dependiendo de la recompensa esperada que se pueda obtener partiendo de él.

$$v(s) = E[G_t|S_t = s]$$

Gracias a esta función podemos determinar qué estados son más favorables, lo cual nos servirá luego para elegir qué acciones tomar. El siguiente paso es descomponer la función de valor en dos partes:

- Recompensa inmediata R_t
- El valor descontado del siguiente estado $\gamma(S_t + 1)$

Así obtenemos una nueva ecuación conocida como **la ecuación de Bellman de estados**:

$$v(s) = R_t + \gamma(S_{t+1}) = R_t + \gamma \sum_{s' \in S} P_{ss'} v(s')$$

Con esta ecuación podemos calcular el valor de cada estado de nuestro MDP, aunque hacerlo no es fácil, y en problemas complejos requiere del uso de técnicas como programación dinámica o los algoritmos de Monte Carlo.

Finalmente, un **proceso de decisión de Markov** también contiene las acciones que un agente puede tomar. Es decir, ahora tenemos el control sobre cómo movernos entre estados. Las acciones a tomar desde cada estado pueden dar lugar a un estado siguiente distinto dependiendo de unas probabilidades concretas. Es decir que tomar la acción A en el estado S_a no tiene por qué llevarnos siempre a un mismo estado S_b, podríamos tener un 70 % de probabilidad de terminar en S_b y un 30 % de terminar en S_c. Siguiendo el ejemplo de antes, que el robot ponga un pie al frente no siempre le garantiza evitar caerse, existe una probabilidad tanto de caerse como de no hacerlo.

Al incorporar las acciones surge de manera natural el concepto de **política**. Una política π es una función que asigna acciones a los estados. Es decir, le asigna a cada acción la probabilidad de ser tomada en cada estado. Una política define completamente el comportamiento de un agente. Gracias a este nuevo concepto podemos definir por fin las ecuaciones más relevantes del aprendizaje por refuerzo:

- **La función de estado-valor** (*state-value function*) $v_\pi(s)$ es el retorno esperado

empezando en el estado S y siguiendo la política π. Esta función nos indica cómo de bueno es un estado S si el agente sigue una política π.

$$v_\pi(s) = E_\pi[G_t|S_t = s]$$

- **La función de acción-valor** (*action-value function*) $q_\pi(s, a)$ es el retorno esperado empezando en el estado s, tomando la acción a, y luego siguiendo la política π. Esta función se utiliza para valorar cómo de buena es una acción arbitraria si luego se va a seguir una política π.

$$q_\pi(s, a) = E_\pi[G_t|S_t = s, A_t = a]$$

El objetivo de los algoritmos de aprendizaje por refuerzo es **encontrar una política óptima π_* que maximice el retorno total del agente**.

- La función de estado-valor óptima (*optimal state-value function*) v*(s) es la mejor función de valor de entre todas las políticas. Es el retorno máximo que uno puede extraer del sistema.

$$v_*(s) = max_\pi v_\pi(s)$$

- La función de acción-valor óptima (*optimal action-value function*) $q*(s, a)$ es la mejor función de acción-valor de entre todas las políticas.

$$q_*(s, a) = max_\pi q_\pi(s, a)$$

Optimizando estas ecuaciones podemos ir mejorando nuestra política hasta conseguir resolver nuestro problema de aprendizaje por refuerzo. Para ello se pueden utilizar diferentes métodos, cada uno de los cuales con sus ventajas y desventajas. En las siguiente secciones explicaremos de manera básica los más relevantes para nuestro trabajo y veremos por cuál nos hemos decantado.

2.1.2 Optimización: descenso de gradiente

'Optimización' se refiere a el proceso de maximizar/minimizar una función. En aprendizaje automático, hablamos de maximizar la función de recompensa o de minimizar la función de pérdida [26].

El algoritmo de optimización por excelencia es el llamado descenso de gradiente (*gradient descent* en inglés). El gradiente de una función en un punto indica la dirección de máximo crecimiento de la función en dicho punto. Gráficamente, nos da información sobre la pendiente. Por ejemplo, el gradiente de la siguiente función 'y' en el primer punto es un valor positivo, indicando que en ese punto 'y' aumenta en 4 unidades al aumentar 'x' en 1 unidad.

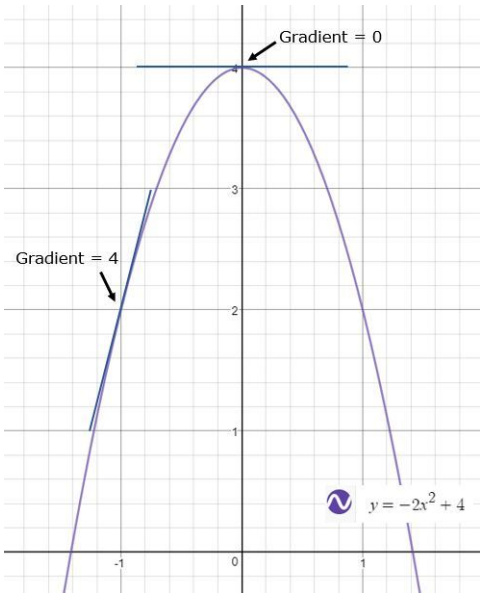

Figura 2.3: Gráfica del gradiente.

Matemáticamente, el gradiente es la derivada parcial de una función 'f' respecto a sus variables y se denota como:

$$\nabla f(x_1, \ldots, x_n) = (\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n})$$

Dado que podemos definir la recompensa como una función de la política, podemos también calcular los gradientes de la recompensa respecto a cada uno de los parámetros de esa política. Es decir, podemos saber cómo varía la recompensa al hacer cambios en la política, por lo que podemos escoger aquellos cambios que maximicen la función de recompensa.

$$\pi^* = \pi + \alpha \nabla R(\pi)$$

Dónde α es el ratio de aprendizaje (*learning rate*), un hiperparámetro que define cuánto debe cambiar la política en cada iteración del descenso de gradiente. Un ratio de aprendizaje muy pequeño hará que el modelo aprenda muy lentamente, mientras que uno muy grande hará que el algoritmo se salte el punto óptimo y no sea capaz de converger.

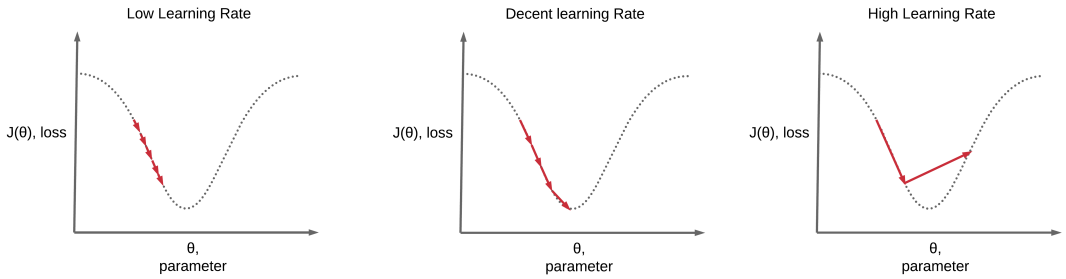

Figura 2.4: Ratio de aprendizaje

Los conceptos de descenso de gradiente y ratio de aprendizaje, aunque sencillos, son parte de la base del aprendizaje automático, y nos permitirán entender más adelante algoritmos de optimización más complejos.

2.1.3 El dilema de la explotación-exploración

Un último concepto muy importante en aprendizaje por refuerzo es el dilema de la explotación-exploración [26]. Cuando intentamos optimizar un modelo, nos encontramos con el problema siguiente: si hacemos que nuestro agente escoja siempre las acciones de mayor valor, con el objetivo de maximizar el retorno, muchos estados nunca serán visitados, puesto que no se escogerán nunca las acciones de menor valor que llevan a ellos. Esto puede llevar a que nuestro algoritmo se estanque en máximos locales y nunca explore un subconjunto de los estados que podrían ser mejores.

Para solucionar este problema se introduce el concepto de **exploración**, que consiste en escoger acciones de manera que, en vez de intentar maximizar el retorno, se busque explorar todos los estados de manera equitativa. El método de exploración más simple es el aleatorio, en dónde la acción que realizará el agente se escoge de manera aleatoria de entre todas las disponibles.

Buscar el equilibrio entre explotación (buscar el retorno máximo) y exploración (tratar de explorar todos los estados) es una de las grandes dificultades del aprendizaje por refuerzo, y se han ideado multitud de algoritmos para intentar solucionarlo.

2.2. Algoritmos de aprendizaje por refuerzo

Aunque el aprendizaje por refuerzo es una técnica de aprendizaje automático muy prometedora, ha estado siempre limitada por su poca aplicabilidad a escenarios reales. Esto es debido a que los algoritmos de RL estaban originalmente pensados para problemas de espacio discreto, sobre los que se pueden definir un número finito de estados y realizar operaciones con ellos. Por ejemplo, obtener la probabilidad de acabar en un estado S_x dado el estado actual S_t y una acción escogida A_t, para así escoger la acción que más nos interese de cara a maximizar una función de recompensa.

Sin embargo, en la mayoría de problemas reales la cantidad de estados suele ser demasiado grande cómo para representarlos todos de manera explícita, o bien los estados son infinitos porque se trabaja con variables continuas. Esta limitación ha hecho que en los últimos años se hayan desarrollado múltiples algoritmos que buscan mitigar estos problemas.

A continuación explicamos brevemente las principales tendencias en el mundo del aprendizaje por refuerzo y las posibles aplicaciones de éstas en nuestro proyecto. Es importante entender que, aunque todos estos métodos pueden implementarse tal cual su descripción académica, normalmente se toman conceptos de todos ellos según convenga, y la línea que los separa se difumina cuando nos vamos a aplicaciones reales.

2.2.1 *Model-free vs Model-based*

Los **algoritmos basados en un modelo** son aquellos que **generan un modelo interno del entorno**, modelando una función de transición entre estados y una función de recompensa. Es decir, que por un lado aprenden cómo se comporta el entorno (función de transición) y por otro cómo de bueno es cada estado o acción (función de recompensa). Esto les permite aprender una aproximación de las dinámicas del entorno sobre el que trabajan, que luego se utilizan para escoger la acción más conveniente en cada situación.

Los algoritmos sin modelo existen en contraposición a los anteriores, y son los más extendidos a día de hoy. Esto se debe, principalmente, a su facilidad de uso y de implementación, pues no necesitan modelar una función de transición, un trabajo que puede ser muy difícil dependiendo de la complejidad del entorno en el que opere el agente.

Los algoritmos sin modelo aprenden directamente una política, es decir que aprenden cómo comportarse sin necesidad de modelar el entorno sobre el que operan. La gran mayoría de proyectos de locomoción basados en aprendizaje por refuerzo, incluido el nuestro, utilizan algoritmos sin modelo.

Una buena manera de diferenciarlos es que los algoritmos con modelo se pueden utilizar para hacer predicciones sobre como evolucionará el entorno, utilizando la función de transición aprendida. Los algoritmos sin modelo sólo pueden escoger acciones utilizando una política aprendida.

2.2.2 Q-Learning

Dentro de los métodos de aprendizaje sin modelo, sin duda el más extendido es ***Q-learning***.

Éste método se basa en el aprendizaje de una función Q, que define cómo de buena es una acción, tal y como veíamos en la explicación de los MDP. Por ejemplo, si tenemos un robot que se puede mover en cuatro direcciones, la función Q nos daría un valor para cada una de las acciones dado el estado actual. Usando estos valores, el robot entonces escogería la acción que más le interese.

Por lo general se escoge la acción de mayor valor, lo que se conoce como una estrategia voraz (*greedy* en inglés). Aquí nos encontramos con el problema de explotación-exploración que explicábamos antes. Escoger siempre la acción de mayor valor puede llevar a nunca explorar algunas acciones que a priori parecen peores, estancándose en máximos locales. La solución a esto suele pasar por utilizar otra estrategia complementaria, generalmente aleatoria, que garantice una exploración más equitativa de espacio de estados.

En problemas simples, los valores de la función Q se almacenan en forma de tabla, donde cada casilla indica el valor de una acción A estando en el estado S. Esta representación es muy sencilla y facilita mucho la implementación, por lo que resulta ideal para problemas con espacios pequeños de estados y acciones, dónde Q-learning

STATES	ACTIONS			
	UP	DOWN	LEFT	RIGHT
...	...	...	...	...
.... .	0.43	0.25	0.12	**0.8**
. ....	-0.5	-0.3	**0.38**	-0.15
:..:	**0.6**	0.18	0	0.53
:'':'	-0.32	**0.75**	0.23	0.49
...	...	...	...	...

Figura 2.5: Representación tabular de la función Q para el juego de la serpiente (*Snake*)

es sin duda el método más extendido. En la figura 2.5 se puede ver un ejemplo de representación tabular de la función Q.

Sin embargo, *Q-learning* presenta un gran problema que dificulta enormemente su utilización para problemas más complejos. Para espacios de estados y acciones más grandes, la representación tabular es inviable, por lo que se necesita una función de aproxime los valores Q. Esta función se implementa generalmente mediante una red neuronal. Aún así *Q-Learning* a menudo presenta comportamientos erráticos si se alcanzan estados que difieren mucho de los visitados durante el entrenamiento. Este problema no es trivial, y ha sido históricamente uno de los mayores impedimentos para la utilización de este y otros muchos algoritmos en aplicaciones reales como la robótica.

Q-learning es un método *off-policy*, es decir que durante el entrenamiento puede utilizar muestras obtenidas en con políticas diferentes a la actual. Esto implica que es un método muy eficiente con las muestras, a diferencia de otros que veremos más adelante.

2.2.3 Aprendizaje por refuerzo profundo

El aprendizaje por refuerzo profundo (**DRL**, por su abreviación en inglés, *Deep Reinforcement Learning*) es el resultado de combinar aprendizaje por refuerzo con aprendizaje profundo.

Muchos de los problemas que se pretenden resolver mediante aprendizaje por refuerzo presentan un MDP tan grande que resulta inviable almacenar los estados y acciones en memoria y realizar operaciones sobre ellos. Mucho menos en el caso de problemas continuos, donde los espacios de estados y acciones son directamente infinitos.

Por este motivo que surge la necesidad de aprendizaje por refuerzo profundo, que **utiliza redes neuronales profundas para aproximar las funciones de valor**. Es decir que en vez de conocer y almacenar el MDP al completo, podemos en su

lugar entrenar a una red neuronal profunda para que aprenda aproximar la función de estado-valor (o la de acción-valor) en base a los ejemplos de entrenamiento que reciba.

Éste concepto es una de las claves en el desarrollo del campo del aprendizaje por refuerzo, y gracias a él se han conseguido resultados increíbles en los últimos años

El ejemplo más evidente es *AlphaGo*, desarrollada por *DeepMind* en 2016 se convirtió en la primera IA capaz de vencer a jugadores profesionales de Go, el juego de mesa más antiguo del mundo. Esto supuso un gran avance en el campo del aprendizaje automático puesto que Go es un juego muy difícil de tratar desde el punto de vista de inteligencia artificial.

Figura 2.6: AlphaGo vs Lee Sedol (2016)

2.2.4 Policy gradient

A diferencia del *Q-learning*, los métodos *policy gradient* no utilizan una función Q para juzgar cómo de buena es cada acción. En vez de eso, **aprenden directamente una política parametrizada**, que puede seleccionar acciones sin necesidad de una función de valor. Esta política se modela como una función parametrizada respecto a $\theta : \pi_\theta(a, s)$. La función de recompensa depende pues de esta política, y se define como [30]:

$$J(\theta) = \sum_{s \in S} d^\pi(s) V^\pi(s) = \sum_{s \in S} d^\pi(s) \sum_{a \in A} \pi_\theta(a|s) Q^\pi(s, a)$$

Donde $d^\pi(s)$ es la distribución estacionaria de la cadena de Markov para π_θ (la distribución de estados bajo la política π). Esto se puede entender mejor si pensamos qué pasaría si pudiésemos viajar a través de la cadena de Markov infinitamente. Entonces la probabilidad de acabar en un estado se estabilizaría en un valor concreto. Esto es lo que se llama probabilidad estacionaria.

$$d^\pi(s) = \lim_{t \longrightarrow \infty} P(s_t = s|s_0, \pi_\theta)$$

Es la probabilidad de acabar en el estado s empezando en s_0 y siguiendo la política π_θ durante t pasos.

Para encontrar los parámetros θ óptimos se utiliza el gradiente de la función de recompensa; $\nabla J(\theta)$. La optimización puede realizarse diferentes métodos, siendo el más simple de ellos el descenso de gradiente. Es importante añadir que, aunque para la optimización puede utilizarse una función de valor, ésta nunca será necesaria para seleccionar acciones, a diferencia de en *Q-Learning*.

La principal ventaja de estos métodos respecto a *Q-Learning* es que consiguen entrenamientos mucho más estables y no necesitan un tratamiento especial para ser usados en problemas de control continuo.

Por contra, su desventaja es que los métodos *policy gradient* son *on-policy*, lo que quiere decir que, durante el entrenamiento, sólo utilizan muestras que hayan sido generadas usando la política actual, con el fin de evitar introducir sesgos durante el aprendizaje. Es decir, cada vez que se actualiza la política, todas las muestras anteriores son descartadas y no se vuelven a utilizar, lo que hace que pierda mucha información útil, por lo que estos métodos son poco eficientes con las muestras.

Policy gradient es una formulación general, por lo que puede aplicarse tanto a algoritmos basados en modelo como sin modelo. Debido a sus características, estos métodos han dado muy buenos resultados en problemas de control continuo complejos, como el control motor en robótica, por lo que resultan especialmente interesantes para nuestro proyecto.

2.2.5 PPO

Proximal Policy Optimization (PPO) [23] es un algoritmo *policy gradient* inspirado en *TRPO (Trust Region Policy Optimization)*, y que viene a solucionar los problemas de convergencia que surgen al utilizar métodos *policy gradient* en problemas muy complejos.

Cómo explicamos en el apartado 2.1.2, cuando queremos optimizar un algoritmo mediante descenso de gradiente surge un problema: necesitamos escoger un ratio de aprendizaje que no sea demasiado grande ni demasiado pequeño. Esto no es trivial, y en muchos casos puede ser imposible encontrar un valor que garantice un aprendizaje estable durante todo el entrenamiento.

La idea principal de PPO es solucionar este problema **restringiendo el cambio máximo que puede realizarse sobre la política** en cada iteración. Para ello introduce una nueva función que denomina *'clipped surrogate objective function'*, que limita la nueva política para que no diverja demasiado de la antigua. Es decir, la nueva política siempre se encontrará en un rango alrededor de la anterior (región de confianza o *trust region*), evitando así cambios bruscos y garantizando la estabilidad en todas las fases del entrenamiento.

El hiperparámetro ϵ sirve para definir esta región de confianza, de manera que la nueva política siempre se encontrará en el rango $[1-\epsilon, 1+\epsilon]$ de la anterior (en el paper original $\epsilon = 0{,}2$).

Para calcular la diferencia entre políticas, podemos calcular el ratio entre la probabilidad de tomar una acción bajo la política actual y la probabilidad de tomar esa misma acción bajo la política anterior.

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta(anterior)}(a_t|s_t)}$$

El valor resultante determinará si la política nueva tiene mayor probabilidad que la anterior de tomar la acción a_t ($r_t > 1$), menor ($r_t < 1$) o igual ($r_t = 1$). Esta información se utiliza luego en la función de pérdida (*loss function*) para calcular la divergencia entre dos políticas y poder satisfacer la restricción de la región de confianza.

TRPO era conceptualmente muy similar a PPO pero utilizaba una medida de divergencia *Kullback–Leibler*, que permite calcular la diferencia entre dos distribuciones probabilísticas, y restringía la actualización de la política fuera de la función objetivo, lo cual implicaba calcular derivadas de segundo orden. Esto hace que su implementación sea más difícil y su ejecución más exigente computacionalmente, por lo que PPO supuso un avance muy significativo.

Por su estabilidad y efectividad para tratar problemas de control continuo, **PPO es el método más extendido en aplicaciones de control motor**, y ha sido el que hemos utilizado durante todo el desarrollo de nuestro proyecto.

2.3. Aprendizaje por refuerzo en videojuegos

A día de hoy el uso de técnicas de aprendizaje por refuerzo en el desarrollo de videojuegos comerciales no son frecuentes. Sin embargo, los videojuegos llevan tiempo siendo uno de los entornos de investigación preferidos para todo tipo de técnicas de inteligencia artificial [24]. Ésta, por su parte, ha sido de vital importancia en el desarrollo de la industria del videojuego, ya que ha sido la base del comportamiento de los NPCs (*Non Playable Character*), lo que ha permitido crear obras mucho más realistas e inmersivas. En la figura 2.7 podemos ver una imagen del juego F.E.A.R., que en 2005 presentó una de las mejores IAs [18] de enemigos nunca vistas, y que aún a día de hoy sigue siendo considerada como tal.

Los videojuegos históricamente siempre han presentado entornos con muchas características favorables para el desarrollo de la inteligencia artificial; un agente (el jugador) sigue un comportamiento específico dentro de un entorno controlado y con unas reglas y objetivos claros. Esto hace de los videojuegos un dominio perfecto dónde los investigadores de inteligencia artificial pueden llevar a cabo sus estudios.

Además, los videojuegos son representaciones de la realidad, por lo que los avances que se realicen en ellos son muy fácilmente extrapolables a situaciones reales. Por ejemplo, si se necesita entrenar a un robot, es mucho más rápido y barato entrenar a una réplica en un videojuego y luego transferir ese conocimiento al robot real para terminar de ajustarlo [29]. Por todo esto, algunas de las implementaciones más conocidas de aprendizaje por refuerzo han sido agentes entrenados para jugar a determinados

Figura 2.7: F.E.A.R. (2005)

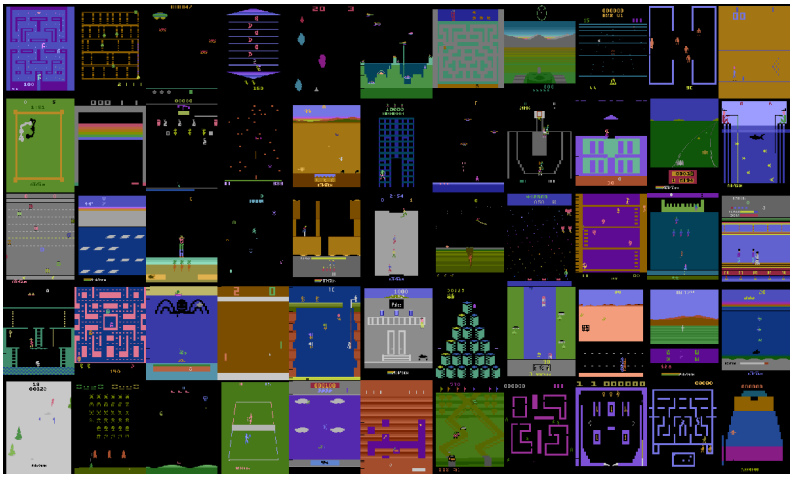

Figura 2.8: Algunos de los juegos de *Arcade Learning Environment (ALE)*

videojuegos.

Existen varias plataformas basadas en videojuegos enfocadas en la investigación de inteligencia artificial. Una de las más relevantes es *Arcade Learning Environment (ALE)* [2]; un entorno que permite a los desarrolladores crear y evaluar agentes que jueguen a juegos de la antigua *Atari 2600*, como los que se pueden ver en la figura 2.8. *OpenAI Gym Retro* tiene el mismo objetivo que ALE aunque se extiende a juegos retro de varias plataformas. *Universe* es otra plataforma de *OpenAI*, pero con la que buscan desarrollar inteligencias artificiales más generales; capaces de jugar a cualquier juego o manejar cualquier aplicación que se les presente.

[1].

En 2017, *Unity Technologies* presentó una tecnología propia llamada *Unity ML-Agents* [27], y que buscaba integrar de manera nativa el aprendizaje automático dentro de su motor de videojuegos. Desde su lanzamiento, han sacado diferentes versiones mejorando muchas de sus capacidades, y a día de hoy es una herramienta de fácil uso

[1] openai.com/blog/gym-retro/

pero con la que se pueden lograr grandes resultados.

A lo largo de los años, han ido surgiendo diversas competiciones que tienen cómo objetivo fomentar el desarrollo de la inteligencia artificial utilizando los videojuegos como base. Destacan especialmente: la *General Video Game AI Competition*, dónde se busca entrenar agentes capaces de aprender a jugar a cualquier juego sin información previa; la *VizDoom competition*, en la cual el objetivo es obtener un agente que juegue a *Doom* recibiendo la imagen como entrada; y las competiciones basadas en *StarCraft* y *StarCraft II* [4], dos juegos muy complejos en los que se requieren agentes capaces de gestionar una gran cantidad de información.

[2]. [3].

Aunque el uso de aprendizaje por refuerzo en juegos comerciales no sea aún una realidad (con algunas excepciones), se está experimentando y tiene un gran potencial para comportamientos de NPCs y, como en nuestro caso, para la creación procedimental de contenido en videojuegos.

2.4. Aprendizaje por refuerzo para control motor

Se han realizado numerosos estudios en relación al aprendizaje por refuerzo para control motor. A continuación resumimos los que nos han parecido más relevantes y que han tenido una mayor influencia en este trabajo.

Los autores de ***DeepMimic*** [21] utilizan aprendizaje por refuerzo profundo para generar movimiento natural en base a múltiples *clips* animados. Es la referencia de mayor importancia para nosotros dado que nuestro trabajo tiene objetivos muy similares. Su objetivo es entrenar a un ragdoll para imitar animaciones predefinidas sin caerse, escogiendo en cada paso los movimientos a realizar con cada una de sus articulaciones. En este caso el agente escoge una pose objetivo en cada momento, el cálculo de las fuerzas necesarias para alcanzar esa pose se delega en otro algoritmo. Además, entrenan otra red neuronal por encima que se encarga de decidir qué animación utilizar como referencia dependiendo del entorno en cada momento.

Para la recompensa, utiliza varias funciones ponderadas mediante sus propios hiperparámetros. La recompensa se divide en dos partes, una de imitación que recompensa al agente por estar en la misma postura que la referencia; y una de objetivo que recompensa al agente por cumplir tareas extra, como alcanzar una velocidad objetivo al caminar.

En este artículo hacen hincapié en tres conceptos:

- **Early termination**: reiniciar la sesión en cuanto el personaje se caiga. Esto evita que el personaje intente alcanzar la pose objetivo desde el suelo, lo que lastra el entrenamiento y evita que aprenda los patrones correctos.

[2] www.gvgai.net
[3] vizdoom.cs.put.edu.pl

- **Initial State Distribution**: iniciar la sesión desde diferentes frames de la animación de manera aleatoria con el fin de aprenderla uniformemente. Quitarlo haría que las partes finales de una animación recibiesen muy poco entrenamiento, dado que dependen de que se aprendan con éxito las partes anteriores.
- **El uso de PIDs (*Proportional Integral Derivative*)** en vez de aplicar torques directamente, con lo que la red neuronal no tiene que aprender las dinámicas de movimiento, sino simplemente elegir una pose objetivo en cada frame. Esto facilita mucho el entrenamiento y da resultados mucho más naturales, dado que le permite a la red focalizarse en aprender cual es el mejor movimiento en cada situación, y no necesita aprender las dinámicas físicas subyacentes necesarias para realizar dicho movimiento. Es decir, se relegan los cálculos físicos en una capa de abstracción diferente, por lo que se simplifica el aprendizaje del modelo.

Dado que tienen el mismo objetivo que nosotros y sus resultados son tan potentes, es el artículo en el que más nos hemos apoyado para la realización de este trabajo.

Otro de los artículos más referenciados sobre este tema es ***Simple Data-Driven Control for Simulated Bipeds*** [11]. Es antiguo y sus resultados ya han sido superados por muchos otros, pero sentó las bases del control de personajes simulados físicamente. No utiliza ni aprendizaje por refuerzo ni aprendizaje automático de ningún otro tipo, por lo que no nos resulta demasiado útil en la práctica. Aún así, su lectura nos resultó interesante, sobre todo para coger confianza con la terminología y las características propias de este tipo de proyectos.

Consta de tres módulos diferenciados. El primero utiliza un controlador PID para seguir una animación objetivo y calcular la diferencia con la simulación. El segundo se especializa en mantener el equilibrio, para lo que utiliza un *'Jacobian Transpose (JT) Control'* que calcula qué torques es necesario aplicar en cada articulación para que la suma de todos ellos sea una fuerza concreta en el centro de masas, a lo que denomina fuerza virtual. El tercero es un sistema que optimiza los parámetros de los dos anteriores para encontrar el mejor modelo, lo hace mediante *'Covariance Matrix Adaption (CMA)'*, un algoritmo evolutivo.

En 2014, el artículo ***Flexible Muscle-Based Locomotion for Bipedal Creatures*** [12] presentaba ideas muy interesantes sobre locomoción y cómo mejorar todavía más el realismo de las animaciones basadas en físicas. Su idea principal era mejorar la naturalidad de este tipo de animaciones recreando la estructura músculo-esquelética de los personajes, en vez de simplemente aplicar fuerzas en las articulaciones. Por ejemplo, en vez mover la pierna aplicando una simple fuerza en la rodilla, lo que hacen es recrear los diferentes músculos que unen ambas partes de la pierna, teniendo en cuenta su tamaño, longitud, puntos de inserción... Así, consiguen un modelo mucho más fiel a la realidad, y eso se nota en la naturalidad de los movimientos que son capaces de generar. En este caso no usan aprendizaje automático de ningún tipo, sino un algoritmo de optimización que es capaz de gestionar en tiempo real la actuación síncrona de cientos de músculos.

Más recientemente, los mismos autores han publicado artículos en los que han seguido iterando sobre las mismas ideas. Destacamos ***Scalable Muscle-actuated Human Simulation and Control*** [16], presentado en 2019 y dónde se centran en

recrear de la manera más realista posible el modelo músculo-esquelético humano y las dinámicas de movimiento que se generan mediante la contracción muscular. Gracias a esto consiguen replicar de manera muy precisa movimientos humanos, y ver cómo estos cambian dependiendo de diferentes factores: el tamaño o longitud de los músculos, el peso de las extremidades, la velocidad de desplazamiento objetivo... Resulta especialmente interesante su capacidad para replicar fallos en músculos o huesos, lo que permite ver qué factores llevan a una persona a a cojear de determinada manera, a desarrollar una forma de andar concreta, o incluso a ver cómo un modelo de prótesis concreto afectaría a las dinámicas de movimiento. En este caso si que usaron un modelo de aprendizaje por refuerzo profundo.

En otro artículo llamado ***Functionality-Driven Musculature Retargeting***[22], consiguieron un algoritmo capaz de transferir la estructura muscular entre cuerpos diferentes manteniendo la funcionalidad. Por ejemplo, son capaces de transferir la musculatura de un cuerpo humano estándar a otro cuerpo con las piernas el doble de largas, con asimetrías en las extremidades, con menor rango de movimiento en las articulaciones... Gracias a esto son pueden crear cuerpos 'exóticos' partiendo de modelos musculo-esqueléticos ya conocidos, lo que les permite también transferir movimientos con la menor pérdida de funcionalidad posible. Para el control del movimiento utilizaron de nuevo aprendizaje por refuerzo profundo.

Por último, mencionamos el artículo ***Learned Motion Matching*** [15], en el que se combina aprendizaje automático con una técnica de animación procedural conocida como *Motion Matching*. Esta última consiste en, partiendo de un conjunto de animaciones, escoger aquella que mejor se ajuste a la situación actual y los objetivos del personaje. Para ello se emplea un algoritmo que busca, en cada instante, entre todas las animaciones disponibles para encontrar la que maximice una serie de variables. Este algoritmo aumenta su coste en tiempo y memoria de manera lineal respecto al tamaño del conjunto de animaciones, lo que limita su escalabilidad.

Para mejorar el problema de la escalabilidad recurren a modelos generativos de movimiento, que han tenido un fuerte desarrollo en los últimos años. Estos utilizan aprendizaje automático para, partiendo de un conjunto de animaciones base, generar animaciones nuevas que se adapten a cada situación. Estos métodos son escalables a grandes conjuntos de datos, pero son difíciles de controlar, requieren de entrenamientos muy largos, y a menudo generan movimientos erráticos o de peor calidad que los de referencia.

Learned Motion Matching mezcla lo mejor de los modelos generativos con lo mejor de *Motion Matching*, para lo cual reemplaza el algoritmo de selección de animaciones de este último con una serie de redes neuronales que comprimen la información relevante de manera eficiente y escalable.

El resultado es un sistema de animaciones que mantiene la calidad de las referencias pero es a su vez escalable a conjuntos de animaciones muy grandes. Aunque en este caso no utilizan animaciones basadas en físicas ni aprendizaje por refuerzo, es un artículo de mucha relevancia, tanto por el realismo de las animaciones generadas como porque son sistemas que sí se están empleando en la industria de manera habitual.

3. Animaciones y simulación física

3.1. Animaciones tradicionales y físicas

Animación es el proceso utilizado para otorgar una sensación de movimiento a imágenes u otros objetos inanimados. Las animaciones, aunque no de manera exclusiva, tienen una relación histórica muy estrecha con los videojuegos, y su uso en estos se remonta a poco después de los orígenes del medio.

De la manera más simplificada posible, una animación digital consiste en una secuencia de imágenes que se reproducen a cierta velocidad, generando la esperada sensación de movimiento, por lo que la tarea de animar tradicionalmente se podría definir como el diseño y creación de todas esas imágenes que forman una animación.

Con el paso del tiempo y los avances tecnológicos del medio la animación digital se ha facilitado y se ha llevado al entorno tridimensional, donde lo que se anima no son simples imágenes bidimensionales, sino modelos en tres dimensiones, con profundidad. Aunque en la mayoría de casos luego se muestran en una pantalla que solo tiene dos dimensiones, representan objetos en tres dimensiones y así se tratan desde el punto de vista del *software*.

Para un animador, animar a un personaje consiste en definir las transformaciones espaciales que sufre cada parte del mismo dentro de un tiempo finito y concreto. Por ejemplo, una animación de andar de un personaje humano no es más que una serie de rotaciones de cada una de las partes del cuerpo alrededor de sus articulaciones.

Para crear animaciones se utilizan programas especializados que permiten asignar secuencias de transformaciones espaciales a las diferentes partes de un modelo tridimensional. Estas animaciones pueden luego exportarse para ser utilizadas por motores de videojuegos u otros fines como el cine de animación.

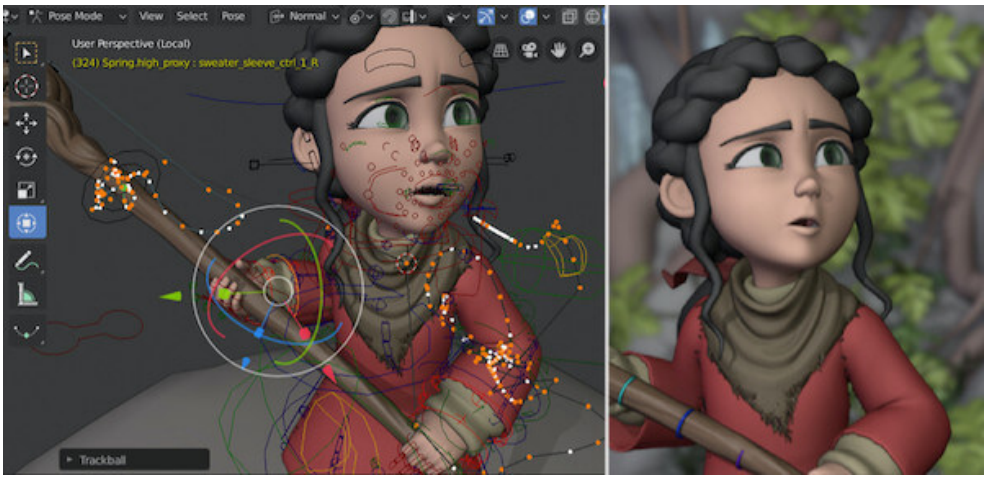

Figura 3.1: Animación en *Blender*

Pero como ya mencionamos en la introducción de este proyecto, nuestro objetivo consiste en generar animaciones basadas en físicas. Las animaciones tradicionales se reproducen como transformaciones espaciales fijas y concretas, tal y cómo las crea el animador. Mientras las animaciones basadas en físicas se reproducen dentro de una simulación física, dónde cada parte móvil está sujeta a las leyes y condiciones físicas de dicha simulación.

De esta forma, un objeto cayendo, que animación tradicional sería una secuencia de imágenes donde el objeto se acerca al suelo, se convierte el objeto siendo atraído por una fuerza gravitatoria hacia el suelo. O un personaje humanoide levantando un brazo, se convierte en esqueleto físico donde es necesario aplicar una fuerza desde el hombro del personaje para generar movimiento.

Estas animaciones basadas en físicas, aunque en la teoría deberían describir los mismos movimientos que las tradicionales, provocan una sensación más inmersiva en el jugador, ya que conllevan sensaciones de inercia y velocidad que son perceptibles para el jugador y resultan más naturales, además de tener la capacidad de interactuar con el entorno de manera realista y reaccionar a perturbaciones. Es decir, ya no estamos ante una secuencia de movimientos inmutable que se reproduce siempre de la misma manera, sino que ahora el movimiento es parte de una simulación, y puede ser perturbado por colisiones, cambios de gravedad, pérdidas de equilibrio...

Nuestra intención es encontrar una forma para convertir estas animaciones tradicionales en animaciones basadas en físicas, pero para ello necesitaremos un repertorio de animaciones tradicionales con las que podamos experimentar esta conversión. Para crear estas animaciones vamos a usar la herramienta de modelado y animación 3D *Blender* [6].

Pero, como acabamos de mencionar, para que se puedan dar estas animaciones basadas en físicas nos es indispensable una herramienta que nos proporciones dicha simulación física. Esta simulación nos la va a otorgar el ya mencionado motor de videojuegos *Unity* [28], con el cual hemos tenido tenido la oportunidad de familiarizarnos durante nuestro paso por la carrera. Internamente, *Unity* utiliza *PhysX* [8], el motor de simulación física en tres dimensiones por excelencia.

3.2. Unity

Unity es un conocido motor de videojuegos multi-plataforma, es decir, una herramienta de creación de videojuegos que cuenta con un conjunto de funcionalidades útiles para el desarrollo de videojuegos, como son: un motor de renderizado en 3D y 2D, un motor de sonido, un motor de físicas, un sistema de *scripting*, un sistema de entrada, etc.

A grandes rasgos, el funcionamiento de Unity es el siguiente. Primero tenemos el concepto de entidad (a lo que Unity llama GameObject), que no es más que cualquier cosa que forme parte del juego: un vehículo, un personaje, un menú, un gestor de sonidos, una luz... Para organizar estos *GameObjects* de manera coherente, aparece entonces el concepto de escena. Una escena es un conjunto de entidades ordenadas bajo una jerarquía concreta, y sirven para diferenciar de manera tajante diferentes partes

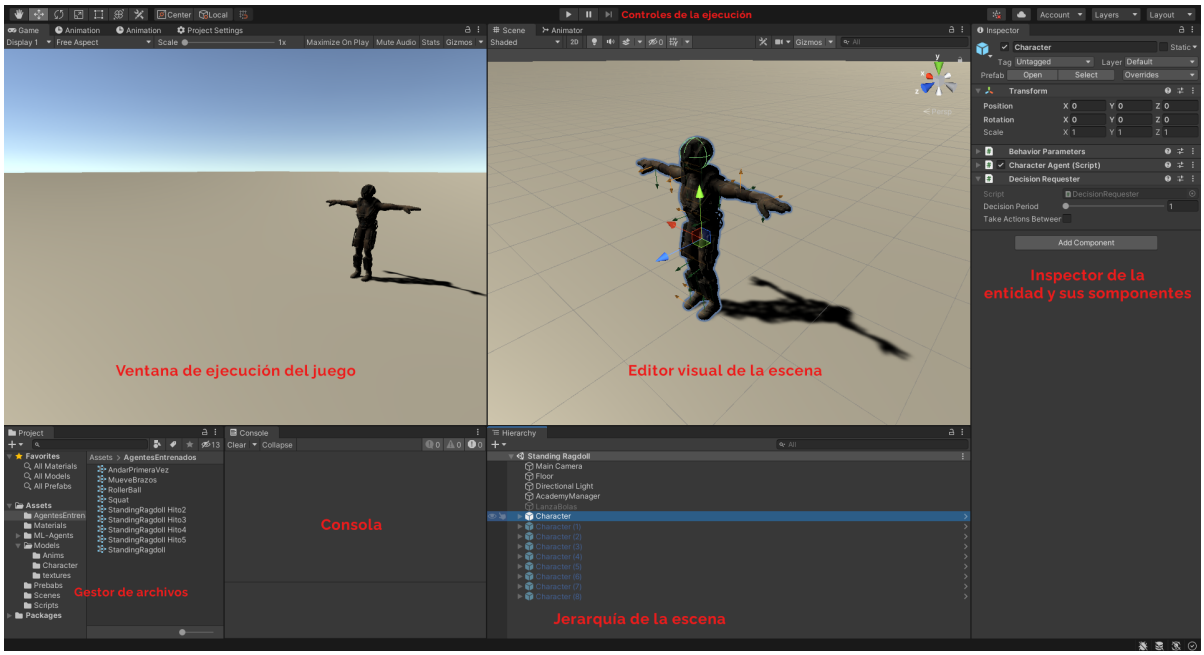

Figura 3.2: Descripción de *Unity*

del juego o aplicación. Por ejemplo, en un juego una escena podría corresponderse al menú inicial, otra al menú de opciones, otra al primer nivel, otra al segundo...

El sistema de *scripting* de Unity es lo que permite programar la funcionalidad de los *GameObjects*, y utiliza una arquitectura por componentes. Cada *GameObjects* solo cuenta por defecto con un componente del tipo *'Transform'* que no se puede eliminar, y que le otorga a la entidad una posición, rotación y escala dentro de la escena. Sin embargo, el desarrollador puede añadir tantos componentes adicionales cómo crea conveniente.

Unity cuenta con una batería de componentes nativos que proporcionan múltiples funcionalidades a nuestros *GameObjects*, pero también se pueden desarrollar componentes propios en forma de '*scripts*', archivos de código a través de los cuales los programadores pueden acceder a la API del motor y crear funcionalidades nuevas con total libertad.

En la figura 3.2 se puede ver cómo está distribuido el entorno de trabajo en *Unity*, aunque esta es sólo una de las posibilidades, ya que el usuario puede mover, cerrar, y escalar las diferentes ventanas como mejor se adapte a su forma de trabajo.

Unity es una herramienta extensa con mucho que ofrecer al desarrollador de videojuegos, pero en el contexto de este proyecto estamos especialmente interesados en las siguientes características:

- **El motor de renderizado 3D**. Ya que en este proyecto vamos a trabajar con modelos y animaciones 3D, necesitamos una forma de que estas puedan aparecer en el monitor de un ordenador. Unity cuenta con un motor de renderizado que nos permite justo eso, proporcionándonos un mundo virtual tridimensional donde podemos desplazar y mostrar en pantalla nuestros modelos.

- **El motor de simulación física**. Proporcionado por la integración en Unity de *PhysX* [8], *Unity* proporciona al desarrollador una simulación física para sus videojuegos. Esta simulación nos resulta indispensable para las animaciones basadas en físicas que pensamos generar como ya mencionábamos en la sección anterior.
- **Sistema de *Plug-Ins* de terceros**. Unity cuenta con la capacidad de agregar herramientas de terceros al propio motor para expandir sus funcionalidades. Esta es la forma por la que vamos a ser capaces de usar técnicas de aprendizaje por refuerzo en el proyecto, a través de un *plug-in* llamado *'ML-Agents'*, creado por los propios desarrolladores de *Unity* en los últimos años.

3.2.1 ML-Agents

Cómo comentamos en la sección anterior, *ML-Agents* es un *plug-in* para el motor *Unity* basado en *PyTorch*, que provee al desarrollador e investigador un conjunto de herramientas para crear entornos de aprendizaje por refuerzo usando el editor de *Unity*, e interactuar con ellos a través de una API basada en *Python*. Todo esto viene incluído con el paquete de *ML-Agents*, que puede instalarse en Unity con muy fácilmente y permite definir entornos de entrenamiento dentro del editor, junto con unos componentes que sirven para controlar el aprendizaje de los agentes. Puede ser usada para entrenar agentes usando RL, aprendizaje por imitación (*Imitation Learning*), neuroevolución y otros métodos de aprendizaje automático.

Una de las características de *ML-Agents* es la posibilidad de elegir entre dos algoritmos de *Deep RL* diferentes para el entrenamiento de los agentes. Estos son PPO [23] y SAC [13], que usan una implementación desarrollada por *OpenAI* e integrada en *PyTorch*.

3.2.2 PyTorch

PyTorch [19] es una librería *open source* de aprendizaje automático desarrollada por *Facebook*, y que ha sido muy utilizada en el mundo de la inteligencia artificial, siendo la base por ejemplo de proyectos tan ambiciosos como el sistema *Autopilot* de *Tesla*. Este tipo de librerías suelen estar programadas en lenguajes de bajo nivel, pero ofrecen una interfaz de más alto nivel para facilitar su uso.

3.3. Cuerpo físico articulado del personaje (*ragdoll*)

Una vez escogido el modelo inicial, necesitamos transformarlo en un ragdoll. Esto es, simular físicamente cada parte de su cuerpo y conectarlas entre ellas. De esta manera, el modelo puede interactuar con el mundo físico: colisionar con objetos, caer por gravedad, ser arrastrado, empujado...

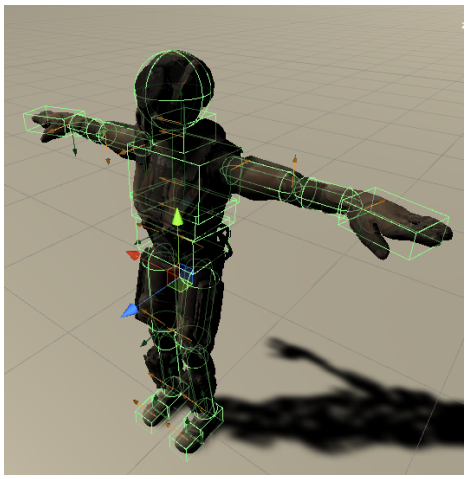

Figura 3.3: Configuración física del ragdoll

Los *ragdolls* son cuerpos inertes que no presentan ningún tipo de funcionalidad, simplemente existen en la simulación física como si de muñecos de trapo se tratasen. Lo que nosotros queremos conseguir se conoce de manera coloquial como *active ragdoll*: un modelo que no solo es simulado físicamente, si no que ejerce fuerzas con sus partes del cuerpo para moverse.

Cuando importas un modelo cualquiera a *Unity*, éste genera un *GameObject* para cada hueso de su esqueleto; brazo, mano, cuello, pie, etc. Es decir, que *Unity* ya interpreta el esqueleto del modelo como un conjunto de objetos relacionados por una jerarquía, lo cual nos será muy útil para configurar nuestro proyecto.

Como lo que queremos es simular este modelo físicamente, lo que hacemos es añadir a cada uno de estos *GameObjects* un *RigidBody* y un *Collider*. El *RigidBody* (cuerpo rígido) es el componente que le permite a nuestro objeto formar parte de la simulación física, mientras que el *Collider* es el componente que define su forma dentro de esta simulación para que pueda así colisionar con el resto de entidades.

Estos dos componentes tenemos que ajustarlos independientemente para cada parte del cuerpo, como se puede ver en la figura 3.3. El *Collider* de un brazo, por ejemplo, deberá ser un cilindro de aproximadamente las mismas dimensiones que el brazo del modelo, para que la simulación física se corresponda con lo que se ve en pantalla. A su vez, ajustamos también la masa del cuerpo rígido para que se corresponda aproximadamente con la distribución de pesos de un ser humano medio, según [9] (ver tabla 3.1).

A continuación, necesitamos que las partes del cuerpo estén conectadas entre sí. Para ello las conectamos utilizando un componente llamado *ConfigurableJoint* en cada una de ellas (excepto en la raíz del cuerpo). Este componente sirve para imponer un conjunto de restricciones entre dos cuerpos rígidos. Por ejemplo, podemos configurarlo para que dos cuerpos físicos no puedan alejarse el uno del otro en ningún eje de movimiento, y solo puedan rotar el uno alrededor del otro en dos ejes. Esto sería una buena representación del comportamiento del antebrazo respecto al brazo, donde ninguno se puede mover sin mover al otro, pero sí que pueden rotar tanto para generar flexión y extensión como pronación y supinación.

Segmento	Hombres	Mujeres	Promedio
Cabeza y cuello	6.94	6.68	6.81
Torso	43.46	42.58	43.02
Brazo	2.71	2.55	2.63
Antebrazo	1.62	1.38	1.5
Mano	0.61	0.56	0.585
Muslos	14.16	14.78	14.47
Pantorrillas	4.33	4.81	4.57
Pies	1.37	1.29	1.33

Cuadro 3.1: Porcentajes de peso corporal total

Así pues, configuramos las *ConfigurableJoints* para simular cada una de las articulaciones del cuerpo humano. Esto no sólo implica bloquear los ejes de movimiento y rotación correspondientes para cada una, sino también configurar los límites de rotación para aquellos ejes en los que se permita el movimiento. Siguiendo el ejemplo del brazo, la flexión y extensión del mismo sólo tiene un rango de unos 150°, no podemos permitir que el antebrazo pueda dar una vuelta completa alrededor del codo.

Configurar todos los cuerpos rígidos, sus colisiones y sus articulaciones es un proceso tedioso que lleva bastante tiempo, pero basta con hacerlo una vez y prestar atención a los pequeños errores que se puedan cometer. Es importante terminar esta fase de la configuración con la certeza de que no hay errores, pues así se evita que surjan problemas durante el desarrollo del aprendizaje automático, que pueden ser difíciles de identificar.

Las *ConfigurableJoints* tienen una funcionalidad especial llamada **rotación objetivo (*target rotation*)**. Esto será clave en la realización de nuestro proyecto, puesto que nos permitirá decirle a cada articulación cómo debe rotar y con qué fuerza.

Tras configurar cada parte del cuerpo, obtenemos entonces un modelo con una simulación física realista, sobre el que luego podremos probar nuestros modelos de aprendizaje por refuerzo. La idea es entrenar a una red neuronal que sea capaz de decidir, para un estado de la simulación dado, qué rotación objetivo necesita cada articulación para imitar a una animación predefinida sin perder el equilibrio.

3.4. Simulación física

El problema planteado posee una complejidad física elevada, pues un modelo con tantos grados de libertad exige la interacción en tiempo real de múltiples objetos sólidos conectados entre ellos. Es por esto que resulta fundamental entender cómo el motor físico gestiona estas interacciones, y configurarlo para obtener una simulación precisa y adaptada a nuestras necesidades.

Los dos parámetros clave que condicionan el resultado de la simulación son los siguientes:

- **Fixed step size**: determina cuánto tiempo debe pasar entre dos actualizaciones

de las físicas. Un valor grande acelera la ejecución, a costa de perder precisión en la simulación. En nuestro caso nos interesa usar un valor pequeño, puesto que necesitamos una simulación precisa y estable para facilitarle el aprendizaje al agente.

- **Solver Iterations y Solver Velocity Iterations**: el motor físico funciona por aproximaciones, dado que obtener una solución exacta a las operaciones que calculan las colisiones y las fuerzas resultantes de las mismas sería demasiado exigente a nivel computacional. Con estos parámetros, podemos indicarle al motor cuantas iteraciones queremos que se realicen sobre estos modelos aproximativos, por lo que aumentándolos conseguiremos una mayor precisión.

  Los valores por defecto son demasiado bajos para un modelo como el nuestro, 6 y 1 respectivamente. Haciendo diferentes pruebas, hemos observado que la simulación sólo comienza a ser efectiva cuando subimos ambos valores por encima de 8. Finalmente, dado que la influencia de estos valores en el rendimiento no es especialmente notable, hemos establecido ambos en 13, más que suficiente para poder confiar en que la simulación será precisa.

- **Velocidad angular máxima**: el efecto de este parámetro no es tan evidente, pero es igualmente importante. Cuando se aplican fuerzas a los cuerpos rígidos, estos aumentan su velocidad correspondientemente. Para que la simulación sea estable, el motor físico establece unos valores límite que las velocidades de los objetos no pueden superar. Estos valores son por defecto bajos, al menos para nuestra aplicación. Dado que el motor físico utiliza un modelo aproximativo, en las primeras iteraciones del mismo se pueden alcanzar grandes velocidades, que luego se ven reducidas al ir mejorando la aproximación. Un límite de velocidad angular demasiado bajo limita los valores que se pueden alcanzar en esos primeros cálculos, haciendo que las aproximaciones subsiguientes salgan sesgadas. Hemos comprobado que subir el valor máximo de la velocidad de rotación ayuda a la simulación, aumentando la agilidad y precisión de los movimientos de nuestro ragdoll. El valor por defecto es de 7 rads/s, y nosotros lo hemos establecido en 20 rads/s, valor a partir del cual dejan de observarse diferencias. Sólo la velocidad máxima de rotación es relevante, la lineal no tiene ningún efecto, debido a la naturaleza rotatoria de las articulaciones de nuestro sistema.

Estos parámetros pueden configurarse de manera global para toda la simulación física, o para cada cuerpo físico independientemente. Nosotros hemos optado por la última solución, de manera que hemos añadido 3 variables a nuestro componente principal que se establecen para todos los *RigidBodies* al principio de la simulación. Así se evita una carga de trabajo innecesaria sobre el motor físico en otros cuerpos del entorno que no requieran de una simulación tan precisa.

3.5. PIDs

Uno de los aspectos que hemos visto repetidos en varios artículos es la utilización de PIDs para el control de la fuerza en las articulaciones.

Los **PIDs (*Proportional, Integral, Derivative*)** son pequeños programas que se encargan de ajustar un parámetro de salida según un error de entrada. En nuestro caso, por ejemplo, el error es la diferencia entre la rotación objetivo de la articulación y la rotación actual, mientras que la salida es la fuerza que se debe aplicar en dicha articulación. La idea es que, dependiendo del error y de cómo este varíe en el tiempo, se ajustará la salida debidamente para alcanzar el objetivo en cada momento.

Los PIDs son útiles porque añaden una capa de abstracción entre la física y la red neuronal. Así, la red pasa a encargarse de decidir cuál es la postura óptima del personaje en cada momento, mientras que los PIDs se encargan de aplicar las fuerzas necesarias para alcanzar dicha postura. Esto evita que la red neuronal tenga que aprender a realizar los cálculos físicos que permiten pasar de una postura a otra, simplificando su trabajo y mejorando el rendimiento del sistema [20].

Durante la realización del proyecto, estuvimos varias semanas implementando nuestros propios PIDs en Unity para que se encargaran de elegir la fuerza adecuado en cada eje de cada articulación. Sin embargo, terminamos por utilizar una característica nativa de Unity que vimos que daba mejor resultado, **la rotación objetivo de las *ConfigurableJoints*** que mencionamos anteriormente.

Ésta es como un PID reducido, donde la parte integral se ha eliminado. Tras varios experimentos, vimos que daba resultados muy similares a nuestros PIDs, y en algunos casos incluso mejores. Para compararlas, creamos una escena de prueba en donde eliminamos la red neuronal, y donde a cada rotación objetivo se le asignaba la rotación de la articulación animada. Es decir, el personaje simulado físicamente seguía la animación lo mejor que pudiese.

Haciendo esto descubrimos dos cosas. La primera es la que ya hemos dicho, que la rotación objetivo de las *ConfigurableJoints* conseguía resultados ligeramente mejores a nuestros PIDs. La segunda es que mucho más importante que eso era el tiempo que pasaba entre cada actualización de las físicas, el valor **fixedTimestep** explicado en el apartado anterior. Cuanto más pequeño sea, más veces por segundo se actualizarán las físicas, lo que implica mayor precisión, pero también mayor carga de procesamiento. Disminuyendo este valor, conseguimos que el ragdoll siguiese a la perfección la animación de referencia.

4. Configuración y elementos del agente

A continuación explicamos las decisiones de diseño y configuraciones finales para cada uno de los elementos del proyecto.

4.1. Modelo 3D del personaje

Lo primero que necesitamos es un modelo, y preferiblemente uno gratuito. El primero que usamos lo acabamos descartando tras los primeros experimentos porque presentaba problemas de simetría.

Más adelante encontramos otro que se ajustaba más a nuestras necesidades el cual aparece en la figura 4.1: *Deadspace Inspired Alien* (crédito al usuario *nataliedesign* por la autoría del modelo) a través de la web Sketchfab.

En realidad, aunque estamos hablando de modelo, lo que realmente importa es el esqueleto. El esqueleto es el conjunto de huesos unidos entre sí que definen cómo puede moverse el modelo. El modelo define cómo se ve, mientras que el esqueleto define cómo se mueve. En nuestro caso, es fundamental que este último sea perfectamente simétrico, así cómo que no presente una excesiva complejidad para no incrementar los grados de libertad del sistema innecesariamente. El esqueleto de este modelo puede verse en la figura 4.2

4.2. Implementación del aprendizaje por refuerzo

4.2.1 Proceso de entrenamiento

El proceso de entrenamiento utilizando *ML-Agents* y *Unity* es muy sencillo, se resume en los siguientes pasos:

1. Desde un terminal, lanzamos *ML-Agents* utilizando *Python*, y le indicamos el fichero de configuración a utilizar y el ID del entrenamiento (para diferenciar los entrenamientos entre sí). Una vez ejecutado el comando, *ML-Agents* carga todo lo necesario y abre un canal de comunicación que se queda en espera hasta que ejecutemos nuestra escena de entrenamiento en *Unity*. Cuando iniciamos la ejecución en *Unity*, comienza esta comunicación, dónde el editor se encarga de la simulación mientras que la parte de Python realiza el entrenamiento del modelo.

   Ésta es la forma más sencilla, aunque no la más potente, dado que estamos limitados por la velocidad de ejecución del editor de *Unity*. Para solucionarlo, podemos primero exportar nuestra escena a un ejecutable, y luego indicarle a

Figura 4.1: *Dead Space Inspired Alien* por *nataliedesing*

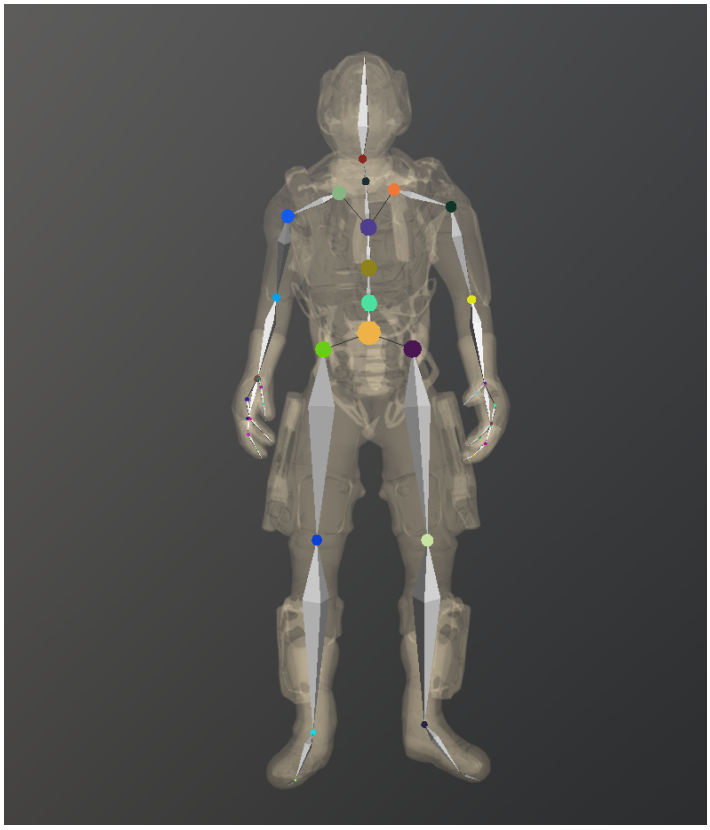

Figura 4.2: Esqueleto del modelo

ML-Agents que utilice dicho ejecutable para el entrenamiento en vez de conectar con el editor de *Unity*. Además, podemos también indicarle que lance varias instancias a la vez para acelerar el entrenamiento, y que no se renderice la parte gráfica para ahorrar recursos.

2. Hecho el paso anterior, *ML-Agents* comienza el entrenamiento. Se llama '**paso**' a cada una de las iteraciones del ciclo de aprendizaje por refuerzo que explicábamos en el capítulo 2. Es decir, en cada paso, el agente: recibe un estado actual y una recompensa del paso anterior, toma una decisión en base a esa información utilizando su política actual, ejecuta esa acción para generar un nuevo estado y una nueva recompensa.

   A un conjunto de pasos se denomina '**episodio**'. Esta separación en episodios es necesaria puesto que en todo entrenamiento de aprendizaje por refuerzo existe lo que se llama condición de terminación. Es decir, llegará un punto en el que no se pueda continuar con el estado actual, o hacerlo resulte infectivo. Por ejemplo, para un agente que aprenda a jugar al ajedrez la condición de terminación sería que la partida termine, en cuyo caso habría que reiniciar el tablero y empezar una nueva partida para seguir entrenando. Cada partida sería un episodio distinto. En nuestro caso, se termina un episodio cada vez que el agente se cae al suelo o se aleja demasiado de la referencia. Las condiciones exactas para determinar cuando esto sucede han ido cambiando entre experimentos, y las explicaremos en detalle en el capítulo siguiente.

3. Finalmente, un entrenamiento termina cuando bien se alcanza un límite de pasos preestablecido o bien el usuario decide finalizarlo manualmente. En algunos casos dejamos que se alcanzase ese límite, cuando veíamos que el entrenamiento estaba siendo bueno, pero mayoritariamente terminábamos nosotros el entrenamiento en cuanto veíamos que no iba bien o que era suficiente para nuestros objetivos del momento. Cuando un entrenamiento termina, *ML-Agents* genera un archivo con los pesos de la red neuronal entrenada. Con este archivo podemos luego en *Unity* utilizar a nuestro agente entrenado y ver los resultados finales.

Por último, decir que durante el proceso de entrenamiento se generan **gráficas de diferentes indicadores** (recompensa acumulada, duración media de los episodios, entropía, pérdida de la política...). Éstas se pueden visualizar en tiempo real gracias a ***TensorBoard***, herramienta para la cual *PyTorch* ofrece integración nativa.

Ésto nos ha permitido ver rápidamente los resultados a medida que realizábamos cambios sobre el modelo o el entorno. Además, cuando un entrenamiento termina, se guardan todas las gráficas generadas durante el mismo, como la que aparece en la figura 4.3 , lo que nos ha sido muy útil para hacer esta memoria, como veremos en el capítulo siguiente.

4.2.2 Configuración de ML-Agents

Como hemos mencionado anteriormente, ML-Agents va a ser nuestra herramienta principal para utilizar técnicas de aprendizaje por refuerzo dentro de *Unity*.

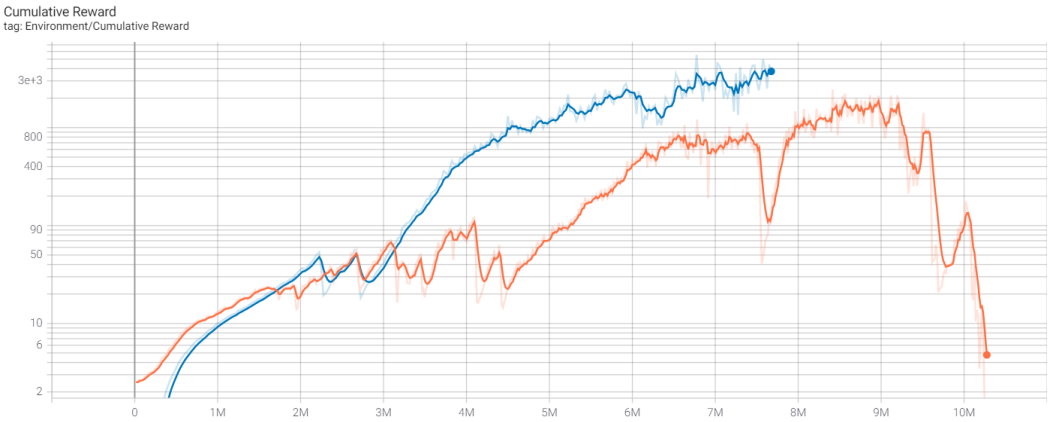

Figura 4.3: Ejemplo de gráfica de la recompensa acumulada

A nivel de usuario, *ML-Agents* presenta 4 elementos principales al programador para interactuar con la herramienta: el vector de observación, el vector de acción, la función de recompensa y el fichero de hiperparámetros.

Para los vectores de observación, acción y recompensa es necesario asignar a la entidad que se va a animar físicamente dos componentes: uno que viene contenido en el paquete *ML-Agents*, llamado *Behavior Parameters*, y otro creado por el usuario, el cual debe heredar de la clase *Agent*. Ambos componentes serán capaces de comunicarse entre sí una vez estén incorporados en el agente. En la figura 4.4 puede verse el componente *Behavior Parameters* y el componente Agente en el editor de Unity.

Behavior Parameters actúa como una conexión intermedia entre el componente creado por el usuario y el entrenador de *ML-Agents*. En este componente hay que especificar el tamaño de nuestros vectores de observación y acción y si el espacio de acciones es continuo o discreto. Aquí también podemos indicarle si queremos, en vez de entrenar al agente, utilizar un modelo ya entrenado.

Por otro lado, en nuestro componente agente, nosotros (el usuario de la herramienta) tenemos que implementar los métodos indicados por *ML-Agents* y que cumplen las siguientes funciones:

- Verificar las condiciones de finalización durante cada episodio.
- Reiniciar el agente y el entorno en cada nuevo episodio.
- Recopilar las observaciones del entorno.
- Ejecutar las acciones del agente.
- Calcular y asignar la recompensa.

4.2.3 Vector de observación

En *ML-Agents*, se llama vector de observación al estado del entorno, es decir toda aquella información que la red neuronal recibe como entrada; **el estado S**.

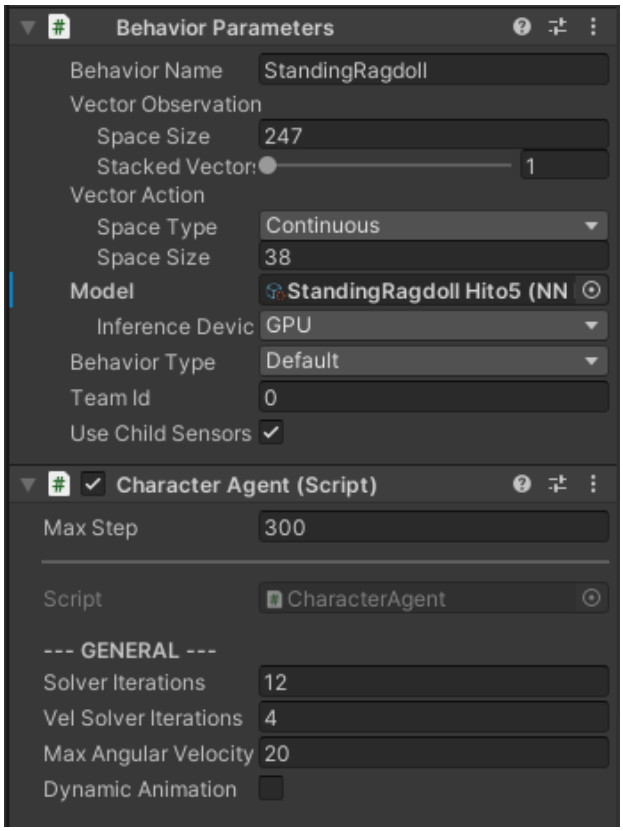

Figura 4.4: Behavior Parameters y Componente Agente en el inspector de Unity

Inicialmente, pensamos que serían necesarios dos tipos de entradas diferentes. Por un lado el estado espacial del cuerpo físico, esto es: la rotación, velocidad angular, velocidad lineal, y posición de todas las partes del cuerpo. Por otro, el estado correspondiente de la referencia. Es decir que recibiría tanto su estado actual (el del cuerpo físico), como su estado objetivo (el de la referencia).

Sin embargo, más adelante nos dimos cuenta de que pasarle como entrada el estado objetivo era completamente innecesario. Solo con el estado del cuerpo físico y una recompensa adecuada, el modelo es capaz de aprender por sí mismo cuáles son las acciones a tomar para alcanzar el estado objetivo. Una vez empezamos los experimentos de animaciones con movimiento, incluimos a mayores una observación que indica el instante en el que se encuentra la animación de referencia. Éste se codifica como un *float* en el rango *[0, 1]*, dónde 0 es el principio de la animación y 1 el final.

También descartamos otros valores de entrada, como un vector de equilibrio que determinaba, usando el centro de masas y el centro de soporte, cómo de desbalanceado estaba el personaje; o un booleano por cada parte del cuerpo que indicaba si ésta estaba o no tocando el suelo. De nuevo, es información que el modelo no necesita para aprender la animación, por lo que fueron descartadas.

Un punto importante a la hora de trabajar con la entrada de este tipo de modelos es que los vectores de medición han de ser transformados a una base común para que el modelo aprenda mejor y sea capaz de generalizar con facilidad. Por lo tanto, todos los vectores se dan relativos a la raíz del modelo, que es la cadera. Así se evita que la

red tenga que trabajar con valores absolutos, que son más difíciles de interpretar. La única excepción son las medidas correspondientes a la propia raíz, que no se pueden dar relativas a sí mismas.

Finalmente, el vector de observación queda así:

- **Rotación** de cada parte del cuerpo físico en espacio local.
- **Velocidad lineal y angular** de cada parte del cuerpo físico en espacio local.
- **Posición relativa** de todas las partes del cuerpo físico respecto a la raíz.
- **Tiempo relativo** de la animación de referencia (solo en animaciones con movimiento)

4.2.4 Vector de acción

En *ML-Agents*, se llama vector de acción al conjunto de valores de salida de la red neuronal.

Los valores del vector de acción indican la rotación objetivo de cada articulación del ragdoll. Cada uno de estos valores es un *float* en el rango [-1, 1], que interpretamos como la rotación relativa de cada eje de cada articulación. Estos valores los convertimos a ángulos de rotación concretos, realizando una transformación lineal a los límites de rotación de cada eje.

Hay que mencionar que originalmente se generaba un *float* para cada eje de cada articulación, independientemente de que éste pudiese moverse o no. Esto lo cambiamos para que el modelo solo tuviese en cuenta aquellos ejes que permitiesen el movimiento. Por ejemplo, la articulación de la rodilla sólo permite movimiento en un eje, por lo tanto se generará un único *float* para cada rodilla.

Vector de acción final:

- **Rotación objetivo de cada eje no bloqueado**, normalizada en el rango [-1, 1].

4.2.5 Recompensa

La recompensa es aquello a lo que hemos dedicado más tiempo y que ha estado sujeto a más fases de experimentación y cambios. Es muy difícil modelar una recompensa adecuada para un problema como éste, dado que hay muchas maneras de interpretarlo y poca información que nos permita decidir de forma contundente cuál de ellas es la mejor.

Recompensa inicial

Inicialmente, como lo que queríamos era simplemente que el agente aprendiese a quedarse de pie, no tuvimos en cuenta la animación de referencia para el cálculo de la

recompensa. Recompensábamos con un *float* en el rango [0, 1] dependiendo de lo bien equilibrado que estuviese el ragdoll. Es decir, de cómo de vertical fuese el vector que va desde su centro de apoyo hasta su centro de masas. Además, si este vector superaba un cierto límite, se terminaba el episodio y se añadía una recompensa negativa grande.

Esta primera recompensa sólo tenía en cuenta el equilibrio, y nos sirvió como punto de partida para iniciar los experimentos y aprender el funcionamiento básico de *ML-Agents*.

Recompensa con sistema de objetivos

Más adelante, para mejorar la recompensa inicial, creamos un sistema de objetivos (o *targets*) encargado de valorar el parecido del ragdoll con la referencia. Los objetivos relacionaban a una parte del cuerpo físico con su referencia animada, y generaban una recompensa según el parecido que hubiese entre ambas. Por ejemplo, en las primeras versiones creamos objetivos para las manos, buscando que el ragdoll no sólo se mantuviese de pie, sino que sostuviese las manos en la misma posición y rotación que la referencia.

Aunque con pocos *objetivos* esta recompensa nos dio algún resultado positivo, en general no terminaba de funcionar del todo bien, y el modelo nunca fue capaz de converger de manera óptima.

Recompensa basada en DeepMimic

Para solucionar los problemas del sistema de objetivos, decidimos cambiar la recompensa a una que ya que había dado resultados anteriormente, la del artículo *DeepMimic* [21].

Esta fórmula 4.1 consta de 4 subrecompensas diferentes, con las cuales se hace una media ponderada por unos pesos que se pueden configurar para cada problema concreto. Así se puede elegir cuánta influencia tiene cada una de ellas sobre el comportamiento del agente.

Todas las subrecompensas tienen como objetivo medir cómo de parecido es el movimiento del ragdoll a la referencia, aunque cada una se centra en un aspecto diferente.

$$R = w_r * r_r + w_v * r_v + w_m * r_m + w_e * r_e \tag{4.1}$$

La primera de ellas es la recompensa de rotación, que se encarga de evaluar cuanto se parecen las rotaciones de las N articulaciones del ragdoll (q_j) a aquellas de la referencia ($\hat{q}_j$). Ésta es la recompensa más importante de todas, puesto que las rotaciones son la base a la hora de imitar una postura. Por ello recibe el mayor peso de todas.

$$r_r = \exp\left[-k_r * \left(\sum_{j=0}^{N} ||q_j - \hat{q}_j||^2\right)\right] \tag{4.2}$$

La segunda valora la diferencia entre la velocidad angular de cada articulación física (v_j) respecto a su referencia ($\hat{v}_j$).

$$r_v = \exp\left[-k_v * (\sum_{j=0}^{N} ||v_j - \hat{v}_j||^2)\right] \tag{4.3}$$

La tercera valora la diferencia de posición entre el centro de masas del ragdoll (m_j) y de la referencia ($\hat{m}_j$), con el objetivo de que encuentre la estrategia de equilibrio adecuada.

$$r_m = \exp\left[-k_m * ||m_j - \hat{m}_j||^2\right] \tag{4.4}$$

Por último, la cuarta recompensa valora la diferencia de posición entre las manos y pies físicos (p_j) y sus referencias ($\hat{p}_j$), con lo que se busca aumentar la estabilidad de las extremidades, que son más sensibles a cambios en la salida. Por ejemplo, un pequeño cambio en el hombro puede provocar un cambio grande de la posición de la mano.

$$r_e = \exp\left[-k_e * (\sum_{j=0}^{N} ||p_j - \hat{p}_j||^2)\right] \tag{4.5}$$

4.2.6 Hiperparámetros

El fichero de configuración de *ML-Agents* se compone de una serie de valores que especifican los hiperparámetros del modelo, y las características del algoritmo de aprendizaje y de la red neuronal que utiliza *ML-Agents* a través de *PyTorch*.

La configuración de los hiperparámetros de cualquier modelo de aprendizaje reforzado es una cuestión compleja. No hay una manera certera de saber qué es lo que funciona y lo que no, más allá de ir probando a realizar cambios sutiles y ver cómo afecta eso al entrenamiento.

Por ello, aunque hemos conseguido llegar a un modelo que funciona y es capaz de converger de manera estable, es muy posible que si siguiésemos realizando experimentos encontrásemos uno todavía mejor.

Elegimos como algoritmo de aprendizaje **PPO** (*Proximal Policy Optimization*), por ser el recomendado por *Unity* para problemas de control continuo, además de haber sido usado anteriormente en proyectos similares, como vimos en el capítulo 2.

Tanto el '*buffer size*' como el '*learning rate*' los escogimos por prueba y error, tras probar muchas configuraciones diferentes y ver cuales eran las más estables y las que convergían mejor. El '*batch size*' se escoge siempre como un submúltiplo del '*buffer size*'.

trainer_type		ppo			
hyperparameters	**Valor**	**network_settings**	**Valor**	**reward_signals**	**Valor**
batch_size	1000	normalize	true	gamma	0.99
buffer_size	20000	hidden_units	1024	strength	1.0
learning_rate	0.0001	num_layers	2	keep_checkpoints	5
beta	0.0005	num_layers: simple	simple	checkpoint_interval	200000
epsilon	0.2	memory	null	max_steps	50000000
lambd	0.95			time_horizon	1250
num_epoch	2			summary_freq	25000
learning_rate_schedule	linear			threaded	false

Cuadro 4.1: Configuración de hiperparámetros final del agente.

Beta, *epsilon*, *lambda* y *num_epoch* los configuramos siguiendo los rangos recomendados por la propia documentación de *ML-Agents*, y aunque probamos otros valores a lo largo del proyecto, al final vimos que lo que mejor funcionaba era lo recomendado.

En cuanto a la arquitectura de la red, probamos múltiples configuraciones. Al principio usábamos una red de menos neuronas por capa y más capas. Tras varios experimentos vimos que el entrenamiento se comportaba mejor con una más red ancha y menos profunda, así que al final optamos por **2 capas ocultas de 1024 neuronas** cada una. La capa de entrada y de salida no se incluyen aquí, y dependen de los grados de libertad del ragdoll a entrenar.

Gamma define cuán adelante en el tiempo considera el agente la recompensa cuando tiene que tomar una decisión. También experimentamos con diferentes valores, tanto más pequeños como más grandes, pero vimos que con 0.99 obteníamos los mejores resultados.

Por último, el '*time horizon*' define cuántos pasos recolectar antes de añadirlos al *buffer* de experiencias, y *Unity* recomienda que sean los suficientes como para englobar una secuencia relevante del comportamiento del agente. En nuestro caso, el agente toma 250 decisiones por segundo, por lo que 1250 equivaldría a una secuencia de 5 segundos, que es lo suficientemente grande como para cubrir cualquier tipo de animación.

La configuración al final del proyecto de estos hiperparámetros puede verse en la tabla 4.1.

4.3. Componente '*CharacterAgent*'

Explicaremos ahora de manera resumida nuestra implementación y las decisiones de diseño importantes que hemos tomado para agilizar el flujo de trabajo y los experimentos.

La mayor parte de nuestro trabajo se concentra en el *script* ***CharacterAgent***, que se encarga de:

- **Comprobar las condiciones de terminación y reiniciar los episodios** en cada iteración de la simulación física. Si se cumple alguna de las condiciones,

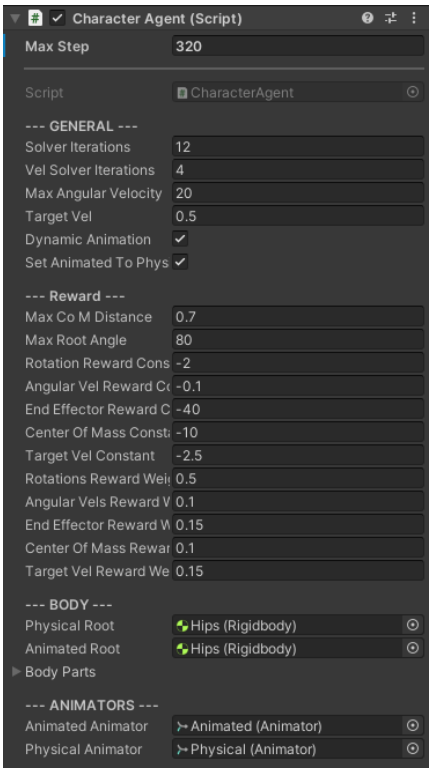

Figura 4.5: Configuración del componente *CharacterAgent* desde *Unity*.

se termina el episodio y se igualan las variables de cada parte del cuerpo a las de la referencia; posición, velocidad lineal y angular, y rotación. Así puede dar comienzo el siguiente episodio utilizando como punto de partida la pose actual de la referencia. Hacemos esto, en lugar de restablecer el ragdoll siempre a la misma pose inicial, para potenciar la exploración de estados que podría costarle mucho alcanzar si partiese siempre de la misma postura.

- **Realizar el cálculo de la recompensa**, según la fórmula 4.1.
- **Recoger las observaciones**, accediendo a las variables de la simulación física necesarias y enviándoselas a *ML-Agents* a través de su *API*.
- **Procesar las acciones generadas por el modelo**. Cada cierto número (configurable) de iteraciones de la simulación física, *ML-Agents* genera un vector de acción utilizando el modelo actual, que *CharacterAgent* transforma a una rotación objetivo en ángulos para cada articulación.

En cuanto a la organización, tenemos declaradas todas las variables de configuración para que sean fácilmente modificables desde el editor de *Unity*, como se puede apreciar en la figura 4.5. Esto incluye principalmente: la **configuración de la simulación física y de la recompensa**, y las **condiciones de terminación de episodios**. Al inicio de la ejecución, el *script* se encarga de aplicar las configuraciones que sean necesarias según estas variables.

Además, buscando agilizar los tiempos de desarrollo, implementamos la función *OnValidate* de *Unity* para automatizar algunos procesos en los que perderíamos mucho tiempo de tener que hacerlos manualmente. Esta función se ejecuta cada vez que se

realiza un cambio sobre el código o las variables del *script*, lo cual es muy útil cuando uno de esos cambios requiere reconfigurar alguna otra parte del proyecto. Por ejemplo, cada vez que modificamos el tamaño del vector de observación o de acción, se actualiza automáticamente la configuración de *ML-Agents*.

CharacterAgent necesita una referencia a cada parte del cuerpo (mano, pecho, pierna...) para el cálculo de la recompensa y las condiciones de terminación. En vez de poner esas referencias a mano, lo cual requeriría bastante trabajo manual, lo tenemos también automatizado para que se detecten automáticamente. Esto es muy útil a la hora de utilizar nuevos personajes o hacer cambios sobre uno ya configurado.

Cada parte del cuerpo dispone, además, de un *script* auxiliar llamado ***BodyPart***, que funciona como capa de abstracción entre la lógica general del agente y el control motor. Para ello facilita funciones genéricas que permiten controlar la rotación objetivo de la articulación y reiniciarla sin necesidad de conocer los detalles concretos de la implementación física. Este *script* también se encarga de guardar la configuración inicial de la articulación y de detectar si esa parte del cuerpo está en contacto con el suelo; aunque ambas cosas dejaron de usarse en las versiones finales del agente.

5. Experimentos y Resultados

Para poder resolver el problema de manera más sencilla, basamos nuestro trabajo en un modelo iterativo. La idea era seleccionar el objetivo más simple, implementarlo con éxito y entonces aumentar la complejidad añadiendo nuevas características, para volver a repetir el proceso hasta alcanzar el resultado buscado.

En este capítulo hablaremos de los diferentes experimentos que hemos realizado para alcanzar estos objetivos del proyecto, describiendo las etapas y resultados parciales que hemos obtenido en cada uno.

5.1. Experimento 1: Equilibrio estático

Una vez hechos todos los preparativos, planteamos como primer objetivo el conseguir un ragdoll que sea capaz de mantenerse de pie ejerciendo fuerza con cada una de sus articulaciones. Ya que nuestro objetivo final es que nuestro agente sea capaz de caminar, ser capaz de quedarse de pie parece un buen punto de partida para luego pasar a animaciones que involucren movimientos.

Además, la capacidad de sostener el equilibrio debería demostrar que el modelo de personaje y su configuración de huesos y articulaciones son capaces de aprender correctamente otras animaciones bípedas.

A continuación exponemos los resultados escalonados que hemos ido obteniendo con el objetivo del equilibrio estático en el horizonte, y explicamos los progresos y el aprendizaje obtenido en cada iteración.

5.1.1 Primera aproximación

Esta primera etapa de la experimentación consistió en familiarizarnos con *ML-Agents* y encontrar una configuración inicial parcialmente funcional.

Función de recompensa

En este experimento la función de recompensa se calcula a través un cálculo de la inclinación del agente y de unas partes del cuerpo del modelo específicas seleccionadas llamadas *targets* que mencionamos brevemente en la sección 4.2.5.

La inclinación se calcula como un valor de 0 a 1, resultando en 1 cuando el ángulo de balanceo del agente (el ángulo formado por la superficie y la línea que describe el agente desde sus pies a su cintura) es 0, y dando 0 cuando el ángulo de balance es de 50 grados o mayor.

$$R_i = 1 - \frac{\log(\alpha + 1)}{\log(Max_\alpha)} \tag{5.1}$$

Hiperparámetros		Configuración de la red	
batch_size	1048	normalize	True
buffer_size	10240	hidden_units	512
learning_rate	0.0005	num_layers	3
beta	0.005	vis_encode_type	simple
epsilon	0.2	memory	None
lambd	0.95		
num_epoch	3		
learning_rate_schedule	linear		

Cuadro 5.1: Hiperparámetros y configuración de la red utilizada.

Los *targets* del modelo físico son comparados uno a uno con sus análogos en el modelo animado en ángulo de rotación y distancia, de forma que cuanto menor sea la diferencia entre estos mayor será la recompensa otorgada.

$$R_t = \frac{\sum_1^n d_i * \alpha_i}{n} \tag{5.2}$$

Los dos cálculos anteriores se multiplican para obtener la recompensa final, de esta forma, si un objetivo tiene un valor muy bajo, la recompensa final será muy pequeña. Esto evita que el agente intente maximizar un objetivo e ignore el otro.

$$R = R_i * R_t \tag{5.3}$$

Por otro lado, las partes del cuerpo del agente tienen un marcador que indica si pueden o no tocar el suelo, de forma que, si durante el cálculo de recompensa se detecta que una parte no autorizada está tocándolo, la recompensa se anula. El objetivo de esto es evitar que el agente intente usar otras partes del cuerpo como apoyo en vez de los pies.

Configuración utilizada

La configuración de hiperparámetros utilizada puede encontrarse en la tabla 5.1.

Entrenamiento

La principal dificultad que encontramos en esta fase fue que el entrenamiento del agente estaba constantemente interrumpido por comportamientos nocivos, como por ejemplo mantenerse sobre una sola pierna, quedarse de rodillas, o sostenerse usando brazos y piernas. Estos comportamientos provocaban que el agente, de manera fortuita, obtuviera un substancial aumento de recompensa con ellos, intentaba replicar este comportamiento, lo que producía un decremento en la recompensa media. Buscar formas de evitar estos comportamientos y mitigar su efecto fue el centro de nuestra atención en estos primeros pasos.

Para intentar evitar esto sólo permitimos el contacto con el suelo mediante los pies, si otra parte del cuerpo entraba en contacto con el suelo el episodio terminaría.

Figura 5.1: Gráfica de recompensa acumulada del entrenamiento

Figura 5.2: Agentes siendo apenas capaces de mantenerse en pie

También incluimos algunas mejoras al modelo físico antes de conseguir estos resultados, como una articulación extra, y un cambio a un material con más fricción en los pies, con el objetivo de que tuviera más control en el contacto con el suelo y evitar que se resbalase.

Como puede verse en la gráfica de la figura 5.1, aunque el agente llegó a ser capaz de alcanzar cierto nivel de recompensa, los entrenamientos anteriores y el que mostramos estuvieron interrumpidos por comportamientos nocivos en los que el agente se atascaba como mencionábamos antes, generando caídas locales de la recompensa.

Comportamiento del agente

Como puede verse en la figura 5.2, el agente es capaz de mantener un equilibrio bípedo, pero corrige su posición constantemente, estando lejos de tener una apariencia natural. Vídeo de ejemplo.

Hiperparámetros		Configuración de la red	
batch_size	2048	normalize	False
buffer_size	20480	hidden_units	650
learning_rate	0.0004	num_layers	3
beta	0.005	vis_encode_type	simple
epsilon	0.2	memory	None
lambd	0.95		
num_epoch	3		
learning_rate_schedule	linear		

Cuadro 5.2: Hiperparámetros y configuración de la red utilizada.

5.1.2 Mejorando el entrenamiento

Cambios realizados

A estas alturas nuestro mayor problema era la poca estabilidad que tiene el entrenamiento, causado por los comportamientos nocivos del agente, por lo que nos centramos en solucionarlo.

Hicimos el cambio de modelo que mencionamos en el apartado 4.1 ya que el modelo anterior era ligeramente asimétrico. Y lo mejoramos ajustando los pesos de las partes del cuerpo del modelo físico según los porcentajes medios de la tabla 3.1, lo que proporcionó al modelo un comportamiento físico más coherente.

También modificamos ligeramente la función de recompensa y realizamos ajustes en los hiperparámetros.

Función de recompensa

En esta versión, la función de recompensa es la misma que la expuesta en el apartado 5.1.1, pero con una modificación que hace que la recompensa otorgada por el ángulo de balanceo pueda llegar a ser negativa. El objetivo de esto es desalentar más drásticamente comportamientos alejados del equilibrio bípedo.

Configuración utilizada

La configuración de hiperparámetros utilizada puede encontrarse en la tabla 5.2.

Entrenamiento

Como puede verse en la gráfica de la figura 5.3, en esta prueba el entrenamiento fue mucho más estable. Las caídas en recompensa seguían estando presentes pero menos frecuentes y menos severas, de las cuales el agente se recuperaba rápidamente.

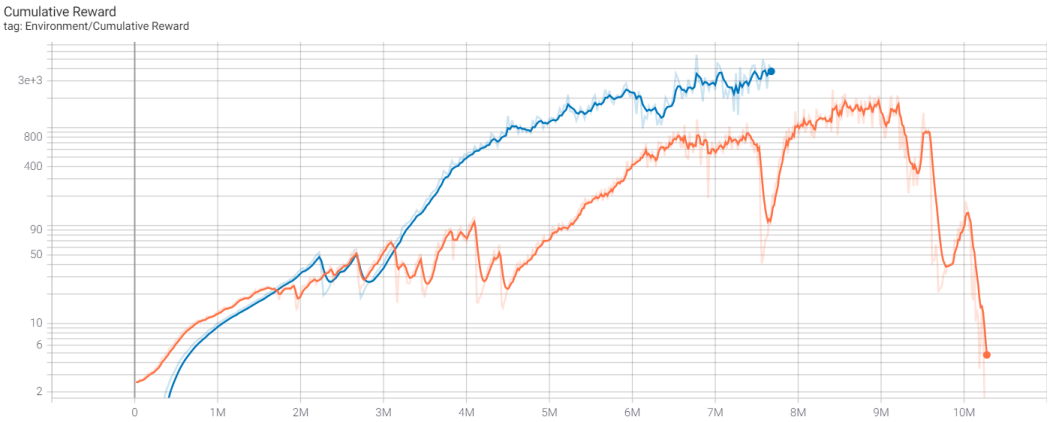

Figura 5.3: Gráfica de recompensa acumulada del entrenamiento. Naranja: Resultado obtenido en las pruebas del apartado 5.1.1. Azul: Resultado de estas pruebas.

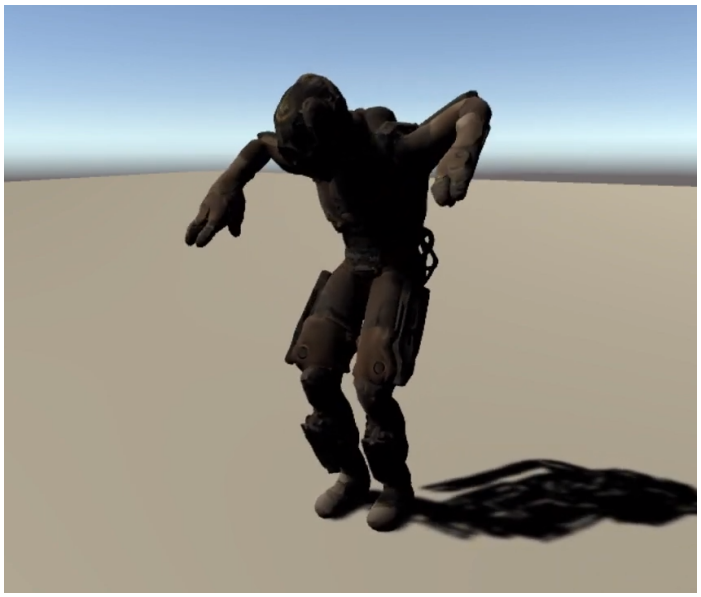

Figura 5.4: El agente sigue teniendo problemas para mantenerse en pie

Comportamiento del agente

Como puede verse en la figura 5.4, en su pico de recompensa el comportamiento de agente apenas cambió, seguía siendo capaz de sostenerse de pie, pero realizando movimientos visibles constantemente. Vídeo de ejemplo.

5.1.3 Entrenamiento estable

Cambios realizados

Nuestro principal problema es que el entrenamiento sigue siendo algo inestable y que no es capaz de alcanzar un nivel suficientemente alto de recompensa como para que sea capaz de mantener una posición relajada en equilibrio.

Los contenidos del vector de observación que involucraban una posición espacial sido convertidos a un espacio local, es decir, la posición se representa desde una posición "padre"de la entidad y no desde la posición global de la escena, para facilitar el aprendizaje. Se utiliza la raíz del cuerpo como base, que en este caso es la cadera del modelo.

La función de recompensa ha sido modificada significativamente.

Reducido el '*decision period*' de 5 a 1 (que es modificable desde uno de los componentes nativos de *ML-Agent* en el editor de *Unity* y simboliza el tiempo que tarda el agente en tomar decisiones). Este valor tan alto estaba impidiendo al agente realizar correcciones sutiles, dado que pasaba demasiado tiempo entre decisión y decisión. Gracias a este cambio el agente es capaz de corregir su postura en intervalos mucho menores y, por tanto, es más estable.

Función de recompensa

La función de recompensa ha sido bastante simplificada. Hemos eliminado la parte de la recompensa que tenía en cuenta cómo de equilibrado estaba el ragdoll (5.1). Ahora sólo se le recompensa por alcanzar las posiciones y rotaciones objetivo (5.2). Esto hace que el entrenamiento converja más rápido.

La otra recompensa era redundante, dado que para alcanzar los objetivos es necesario que se mantenga en equilibrio. También hemos eliminado la parte negativa de la recompensa, tras comprobar que provocaba inestabilidad en los entrenamientos.

Por otro lado, antes se añadía una recompensa negativa si alguna parte del cuerpo que no debía estaba tocando el suelo, ahora esto se ha convertido en un multiplicador que reduce la recompensa ya añadida.

También hemos añadido un multiplicador de recompensa que multiplica por valores mayores la recompensa actual cuanta menos fuerza relativa haga el ragdoll con sus articulaciones. Es decir se le recompensa por hacer el mínimo esfuerzo posible, con lo que pretendemos conseguir eliminar la rigidez de anteriores modelos y conseguir movimientos más naturales.

Configuración utilizada

La configuración de hiperparámetros utilizada puede encontrarse en la tabla 5.3.

Entrenamiento

Como puede verse en la gráfica de la figura 5.5, con esta configuración el entrenamiento resultó totalmente estable. El agente ya no tomaba comportamientos que acabasen reduciendo su recompensa y truncando el progreso anterior.

Hiperparámetros		Configuración de la red	
batch_size	2048	normalize	False
buffer_size	20480	hidden_units	512
learning_rate	0.0003	num_layers	4
beta	0.005	vis_encode_type	simple
epsilon	0.15	memory	None
lambd	0.95		
num_epoch	3		
learning_rate_schedule	linear		

Cuadro 5.3: Hiperparámetros y configuración de la red utilizada.

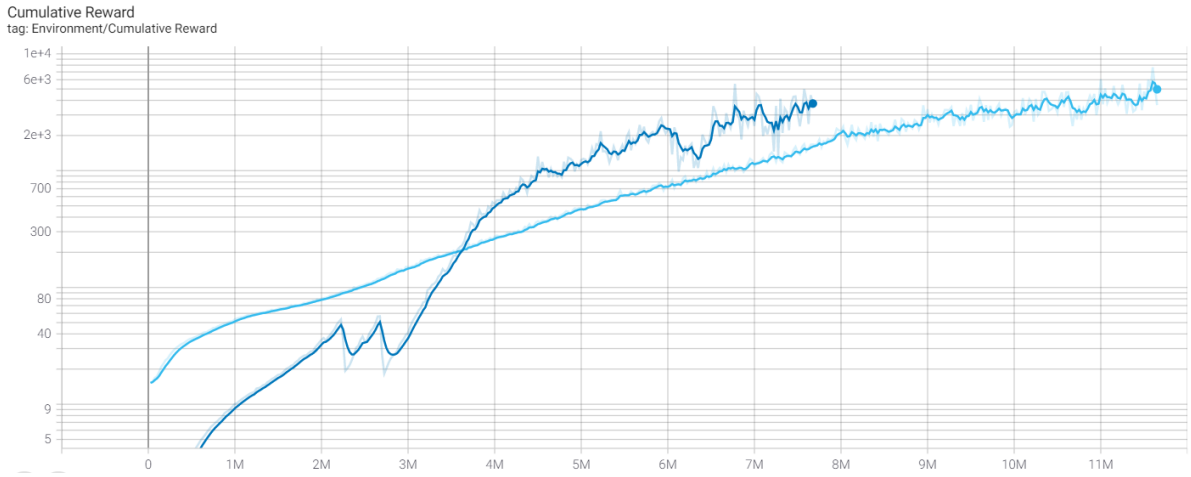

Figura 5.5: Gráfica de recompensa acumulada del entrenamiento. Azul: Resultado obtenido en las pruebas del apartado 5.1.2. Celeste: Resultado de estas pruebas.

Comportamiento del agente

Como puede verse en la figura 5.6, El agente sigue siendo capaz de estar de pie de una manera mucho más estable, pero sigue sin poder quedarse totalmente parado. Vídeo de ejemplo.

5.1.4 Equilibrio natural

Cambios realizados

Teniendo un entrenamiento que es estable pero que no es capaz de alcanzar una recompensa suficientemente alta como para que el agente sea capaz de conseguir un comportamiento estático, solo nos queda intentar mejorar todos los aspectos como podamos para alcanzar este resultado.

Sustituimos la función de recompensa basada en *targets* que estábamos usando hasta ahora por la función de recompensa basada en el la de *DeepMimic* [21] que

Figura 5.6: Agente mucho más estático, pero sigue corrigiendo constantemente su posición

expusimos en el apartado 4.2.5. Esta es conceptualmente muy similar a nuestro sistema de objetivos, pero tiene una base matemática mucho más sólida.

Reconfiguramos el motor de físicas para aumentar la precisión de la simulación, lo cual también implica un mayor coste de recursos. Para ello hemos reducido el *timestep* del motor físico, lo que hace que se actualice más veces por unidad de tiempo.

Reducido el espacio de observaciones a casi la mitad quitando el estado del cuerpo de referencia, ya que el agente no debería necesitar saber donde está la referencia y simplemente puede guiarse por la recompensa obtenida. Los resultados del entrenamiento tras esta simplificación han sido equivalentes, por lo que esas observaciones no estaban aportando nada.

Asignado un número máximo de *steps* por episodio. Esto permite medir con mucha más facilidad el rendimiento del modelo, puesto que ahora hay un límite superior de recompensa. Es decir, sabiendo la recompensa máxima y la obtenida, podemos obtener un porcentaje que nos indica cómo de cerca del comportamiento óptimo está nuestro agente.

Hiperparámetros de la red neuronal reconfigurados para adaptarse a todo el resto de cambios.

Función de recompensa

Para obtener estos resultados utilizamos la función de recompensa de *DeepMimic* expuesta en el apartado 4.2.5 del capítulo anterior, pero sin aun utilizar la subrecompensa que compara las posiciones de manos y pies físicos y animados.

Como mencionamos en dicho apartado, esta recompensa usa unos valores que actúan como pesos para determinar el valor que tiene cada subrecompensa en la re-

Subrecompensa	Valor
Rotación	0.7
Velocidad Angular	0.15
Centro de Masa	0.15

Cuadro 5.4: Pesos de la recompensa

Hiperparámetros		Configuración de la red	
batch_size	1000	normalize	True
buffer_size	20000	hidden_units	1024
learning_rate	0.0001	num_layers	2
beta	0.0005	vis_encode_type	simple
epsilon	0.2	memory	None
lambd	0.95		
num_epoch	2		
learning_rate_schedule	linear		

Cuadro 5.5: Hiperparámetros y configuración de la red utilizada.

compensa total. Los pesos utilizados para conseguir estos resultados aparecen en la tabla 5.4.

Configuración utilizada

La configuración de hiperparámetros utilizada puede encontrarse en la tabla 5.5.

Esta recompensa se calcula y otorga al agente cada vez que este recibe y ejecuta una nueva acción.

Entrenamiento

Como puede verse en la gráfica de la figura 5.7, el entrenamiento transcurre con la misma estabilidad que en el experimento anterior y casi es capaz de alcanzar el máximo de recompensa, al que se acerca de manera asintótica.

Comportamiento del agente

Como puede verse en la figura 5.8, el agente es capaz de mantenerse en equilibrio relativamente parado. Su posición es casi exactamente la misma a la de la animación de referencia, e incluso es capaz de soportar ciertas alteraciones como pequeños golpes o desplazamientos de sus extremidades, aunque en el momento que se aleja mucho de los estados visitados durante el entrenamiento genera movimientos erráticos. Vídeo de ejemplo.

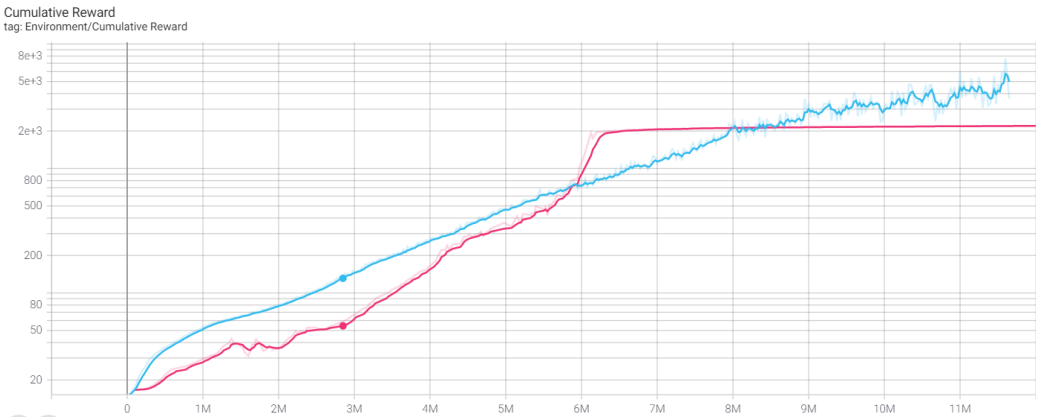

Figura 5.7: Gráfica de recompensa acumulada del entrenamiento. Celeste: Resultado obtenido en las pruebas del apartado 5.1.3. Magenta: Resultado de estas pruebas.

Figura 5.8: Agente siendo capaz de mantener el equilibro sin apenas moverse

5.2. Experimento 2: Animación dinámica

Una vez conseguido el primer objetivo de mantener una pose estática de manera estable, ya podemos empezar a probar el modelo con animaciones, lo cual es a priori bastante más difícil. Ahora no sólo le bastará con encontrar la posición de equilibrio más parecida a la pose de referencia, sino que tendrá que hacerlo mientras se mueve de manera precisa.

Haber conseguido que el agente mantenga el equilibro significa que una gran parte del proceso que configurar el agente para que cumpla este segundo objetivo ya está completada, y cumplir este objetivo debería suponer un buen punto de partida para el siguiente donde el agente no solo deberá moverse, pero también desplazarse de posición.

Subrecompensa	Valor
Rotación	0.7
Velocidad Angular	0.15
Centro de Masa	0.15

Cuadro 5.6: Pesos de la recompensa

Hiperparámetros		Configuración de la red	
batch_size	1000	normalize	True
buffer_size	20000	hidden_units	1024
learning_rate	0.0001	num_layers	2
beta	0.0005	vis_encode_type	simple
epsilon	0.2	memory	None
lambd	0.95		
num_epoch	2		
learning_rate_schedule	linear		

Cuadro 5.7: Hiperparámetros y configuración de la red utilizada.

5.2.1 Moviendo los brazos

La primera animación que probamos fue una animación simple de movimiento de brazos y cabeza, en la cual un brazo se mueve hacia arriba mientras el otro se mueve hacia abajo, mientras la cabeza se mueve de lado a lado.

Cambios realizados

Partimos del agente del apartado 5.1.4, pero le realizamos un pequeño cambio. Dado que la postura cambia con el tiempo, la red necesita una referencia de en qué punto se encuentra la animación cuando vaya a tomar una decisión. Así pues, ahora en el vector de observación también incluimos un *float* de 0 al 1 que indica el estado de la animación, donde 0 es el primer frame y 1 el último. A parte de ese no se realizaron más cambios para conseguir que el agente imitase la animación.

Función de recompensa

Estos resultados se obtuvieron con la función de recompensa de *DeepMimic* [21] y con los pesos que aparecen en la tabla 5.6.

Configuración utilizada

La configuración de hiperparámetros utilizada puede encontrarse en la tabla 5.7.

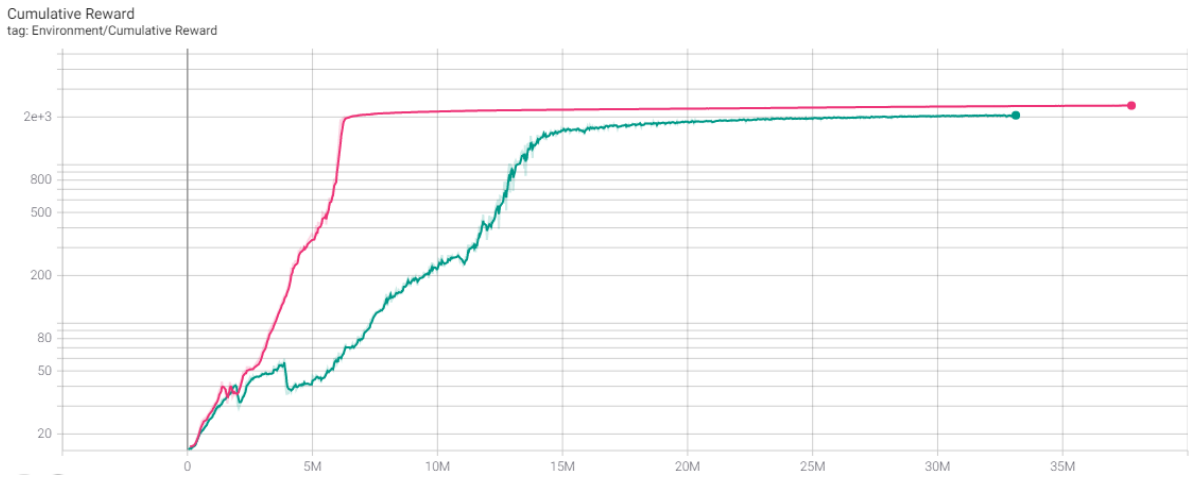

Figura 5.9: Gráfica de recompensa acumulada del entrenamiento. Magenta: Resultado obtenido en las pruebas del apartado 5.1.4. Cian: Resultado de estas pruebas.

Entrenamiento

El entrenamiento fue un éxito en el primer intento y alcanzó un 82 % de la recompensa máxima alcanzable. Probablemente el hecho de que esta animación no involucre un movimiento en las piernas sea el motivo de los rápidos resultados, ya que la animación compromete poco al equilibrio que ya era capaz de mantener.

Como puede verse en la gráfica de la figura 5.9, comparándolo con el entrenamiento que tuvo cuando aprendió a mantener el equilibrio, se puede observar que este entrenamiento fue menos estable y tardó más en converger. Pese a ello, como hemos mencionado antes, fue capaz de acercarse mucho a la recompensa máxima posible, y se estabilizó de la misma forma que el el entrenamiento del equilibrio.

Comportamiento del agente

Como puede verse en la figura 5.10, el ragdoll es casi indistinguible de la referencia, exceptuando pequeñas correcciones que hace con las piernas para estabilizar el movimiento de los brazos.Vídeo de ejemplo.

5.2.2 Sentadillas

A diferencia de la animación anterior, esta animación en la que el modelo animado hace sentadillas, realiza un movimiento de piernas y desplazamiento del tren superior que desafía directamente el equilibrio que alcanzamos en el experimento anterior. Sin embargo, conseguimos que el agente realizase esta animación sin realizar demasiados cambios, aunque nos costó algo más que con el movimiento de brazos.

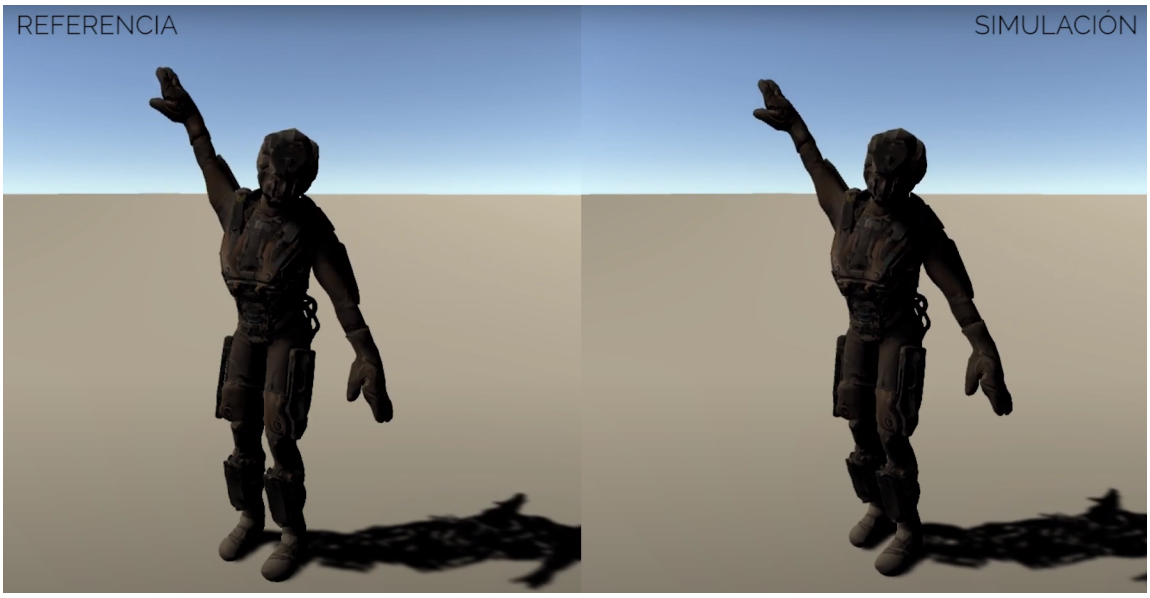

Figura 5.10: Agente ejecutando la animación de mover los brazos

Subrecompensa	Valor
Rotación	0.65
Velocidad Angular	0.1
Centro de Masa	0.1
Posición Extremidades	0.15

Cuadro 5.8: Pesos de la recompensa

Cambios realizados

Los únicos cambios fueron una modificación de la función de recompensa y un pequeño ajuste a los hiperparámetros.

Función de recompensa

Estos resultados se obtuvieron con la función de recompensa de *DeepMimic*, pero a esta altura fue cuando incluimos la subrecompensa que compara las manos y los pies del agente y la referencia. Los pesos utilizados para conseguir estos resultados fueron los siguientes:

El valor de los pesos de las subrecompensas puede ser encontrado en la tabla 5.8.

Configuración utilizada

La configuración de hiperparámetros utilizada puede encontrarse en la tabla 5.9.

Hiperparámetros		Configuración de la red	
batch_size	1000	normalize	True
buffer_size	20000	hidden_units	1024
learning_rate	0.0001	num_layers	2
beta	0.001	vis_encode_type	simple
epsilon	0.2	memory	None
lambd	0.95		
num_epoch	2		
learning_rate_schedule	linear		

Cuadro 5.9: Hiperparámetros y configuración de la red utilizada.

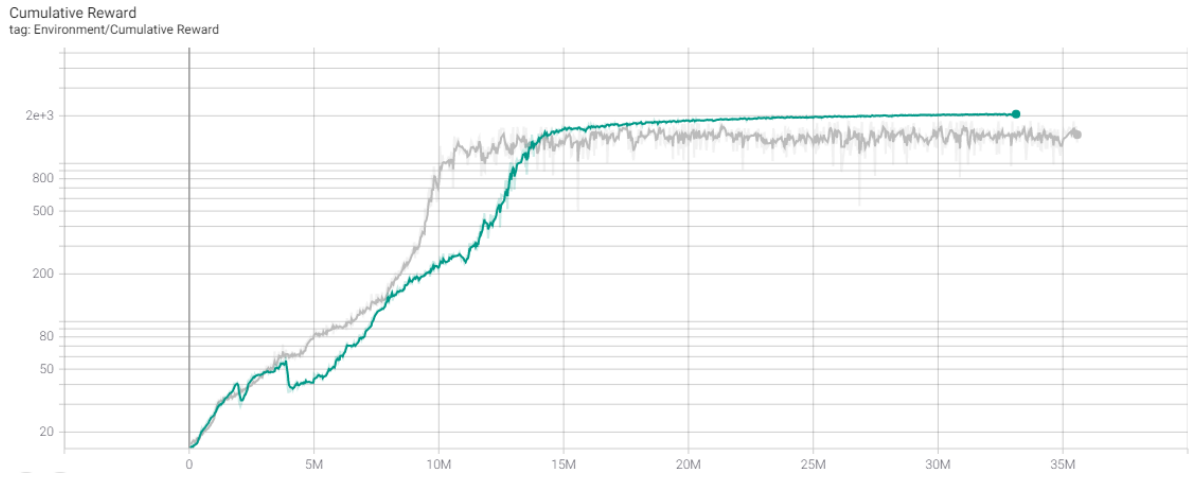

Figura 5.11: Gráfica de recompensa acumulada del entrenamiento. Cian: Resultado obtenido en las pruebas del apartado 5.2.1. Gris: Resultado de estas pruebas.

Entrenamiento

Como puede verse en la gráfica de la figura 5.11, al compararlo con el entrenamiento de la animación de los brazos, el entrenamiento de las sentadillas transcurrió de manera más estable, e incluso consiguió converger más rápido. Sin embargo, no alcanzó un una recompensa tan estable ni tan cercana a la máxima como en los dos experimentos anteriores, lo cual es comprensible si tenemos en cuenta que se trata de un movimiento mucho más complejo.

Comportamiento del agente

Como puede verse en la figura 5.12, aunque en el entrenamiento obtuvo alrededor de un 60 % de la recompensa máxima total, el ragdoll se mueve de manera muy precisa, y es también difícilmente distinguible de la referencia. Vídeo de ejemplo.

Aunque un 60 % pueda parecer poco, hay que tener en cuenta que la animación original seguramente no sea posible de realizar en la manera que fue creada. Es decir, el agente tiene que encontrar el punto medio entre parecerse a la referencia y no caerse,

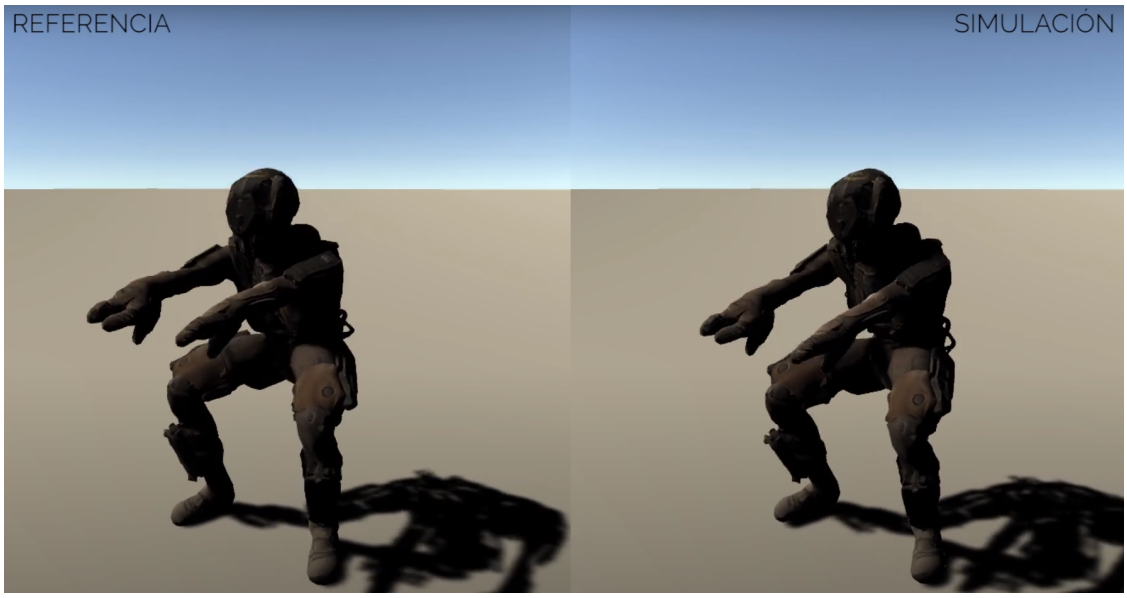

Figura 5.12: Agente siguiendo la animación de sentadilla

y parecerse más de un 60 % seguramente implique perder el equilibrio.

Con estas dos animaciones damos por sentadas las bases del proyecto, y conseguimos los objetivos mínimos que teníamos en mente cuando lo comenzamos.

5.3. Experimento 3: Animación dinámica con desplazamiento

El siguiente y más ambicioso de nuestros objetivos es conseguir que el ragdoll se desplace de manera estable siguiendo una animación dada.

Este experimento es posiblemente el más interesante de todos, ya que conseguirlo no solo significaría tener un agente capaz de generar una animación física que consiga andar físicamente, sino que además significaría que el agente conseguiría aprender a desplazarse por su cuenta, si necesidad de enseñarle explícitamente que se desplace hacia delante.

Para empezar, tendremos que hacer que la referencia se encuentre siempre en la misma posición horizontal que el ragdoll, lo cual no sabemos cómo puede influir en el entrenamiento. De no ser así, el ragdoll intentaría imitar el movimiento quedándose quieto, pues en cuanto se moviese se separaría de la referencia, y no sería recompensado ni por mantener el centro de masas igual al de la referencia ni por mantener manos y pies en la misma posición.

En los primeros intentos que hicimos, los resultados fueron pésimos. El ragdoll no se movía del sitio, y si lo hacía era para moverse lateralmente, en vez de hacia delante. Encontramos que uno de los problemas era que la referencia también se movía con él lateralmente, por lo que durante el entrenamiento el ragdoll veía más fácil dejarse caer hacia los lados que aprender a impulsarse con una pierna hacia delante.

Subrecompensa	Valor
Rotación	0.56
Velocidad Angular	0.07
Centro de Masa	0.07
Posición Extremidades	0.1
Velocidad objetivo	0.2

Cuadro 5.10: Pesos de la recompensa

Una vez solucionamos esto, conseguimos por primera vez que aprendiese a andar, aunque no de la manera que queríamos. El agente desarrolló una manera muy peculiar de moverse, con los brazos en alto y llevando las piernas hacia los lados mucho más que la referencia.

Cambios realizados

La perspectiva de que el agente tuviera que desplazarse involucró unos cambios inmediatos para que el entrenamiento del agente pudiera siquiera contemplar esta posibilidad.

Como acabamos de mencionar, para que el agente fuera capaz de desplazarse tuvimos que "liberar" la posición de la referencia, que hasta ahora había estado fija en un sitio, para que siguiera el movimiento del agente.

Al principio permitimos que la referencia siguiera al agente en cualquier dirección del plano del suelo, hacia delante, detrás, izquierda y derecha. Pero para nuestra sorpresa esto ocasionó una tendencia en el agente de desplazarse lateralmente, probablemente debido a que el movimiento de piernas que usaba para este desplazamiento comprometía menos su equilibro. Para evitar esto, implementamos que la referencia solo pudiera seguir la posición del agente en el eje Z de la escena, es decir, hacia delante o hacia atrás.

Incluimos una nueva subrecompensa para recompensar que el agente alcanzase una velocidad de desplazamiento objetivo con la cadera.

Función de recompensa

Estos resultados se obtuvieron con la función de recompensa de *DeepMimic* [21], pero con una nueva subrecompensa a la que se le atribuye al igual que las otras un peso en la recompensa total, que se encarga de calcular la velocidad a la que se está desplazando el agente en el espacio, y le asigna una recompensa cuan más cerca esté esta velocidad a una estipulada.

El valor de los pesos de las subrecompensas puede ser encontrado en la tabla 5.10.

Hiperparámetros		Configuración de la red	
batch_size	1000	normalize	True
buffer_size	20000	hidden_units	1024
learning_rate	0.0001	num_layers	2
beta	0.001	vis_encode_type	simple
epsilon	0.2	memory	None
lambd	0.95		
num_epoch	2		
learning_rate_schedule	linear		

Cuadro 5.11: Hiperparámetros y configuración de la red utilizada.

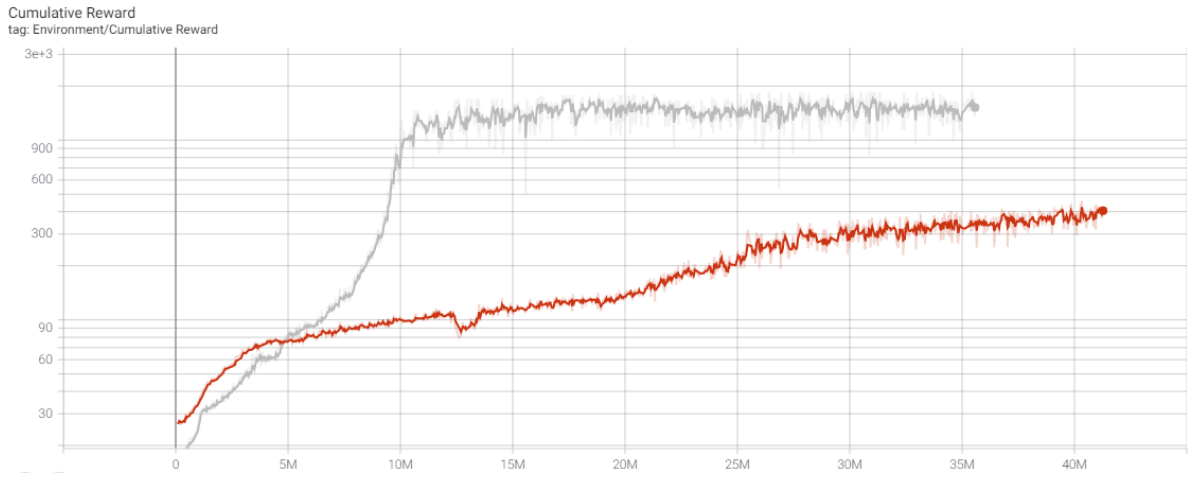

Figura 5.13: Gráfica de recompensa acumulada del entrenamiento. Gris: Resultado obtenido en las pruebas del apartado 5.2.2. Rojo: Resultado de estas pruebas.

Configuración utilizada

La configuración de hiperparámetros utilizada puede encontrarse en la tabla 5.11.

Entrenamiento

Como puede verse en la gráfica de la figura 5.13, aunque el entrenamiento es bastante estable, está muy lejos de la recompensa máxima, y en todo el tiempo que estuvo entrenando (4 horas y media) no fue capaz de acercarse al resultado que obtuvimos con las sentadillas.

Comportamiento del agente

Como puede verse en la figura 5.14, el agente consigue desplazarse hacia delante de manera costosa e irregular y los movimientos que hace están lejos de los que la referencia ejecuta. Vídeo de ejemplo.

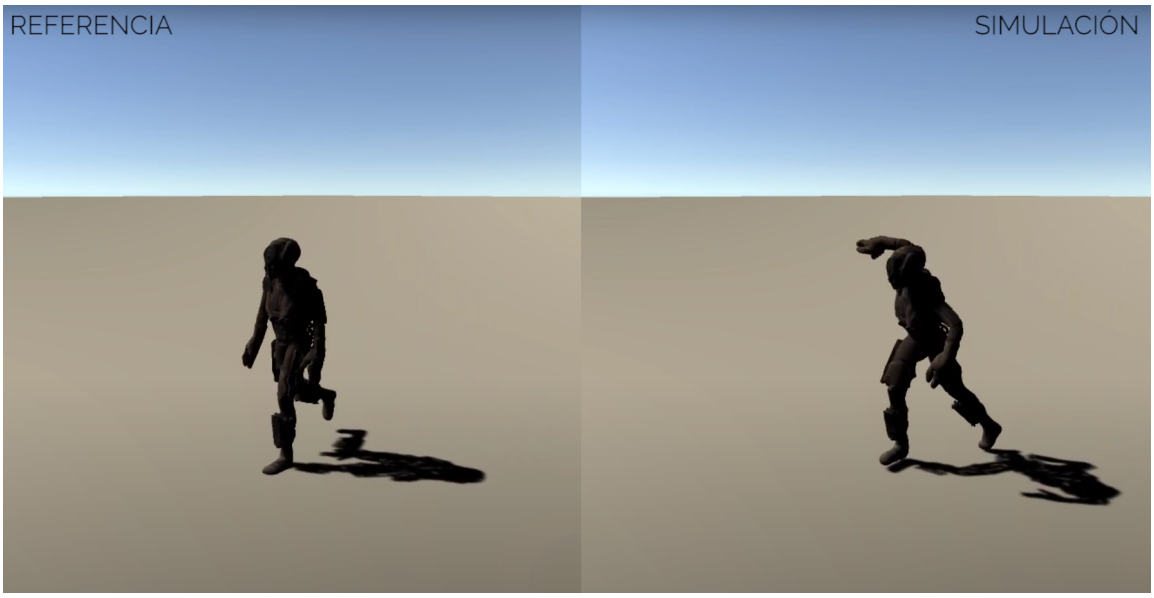

Figura 5.14: Agente consiguiendo desplazarse, pero estando lejos de seguir la animación de referencia

5.4. Conclusiones de los experimentos

Después de finalizar la experimentación podemos sacar las siguientes conclusiones del proceso:

La configuración del entorno de entrenamiento y de la simulación física son tan importantes como los otros conceptos del aprendizaje por refuerzo. Aunque se use una función de recompensa correcta para el agente y las observaciones y acciones sean las adecuadas, si el entorno no está bien planteado de manera lógica y física el agente no será capaz de aprender los comportamientos deseados. Prueba de esto son las diferentes mejorías que hemos obtenido a lo largo de la experimentación mejorando el entorno y no necesariamente el agente.

Por otro lado, los largos tiempos de entrenamiento y la opacidad a lo que está sucediendo en estos hace que sea muy difícil detectar errores y erratas comunes que no sean fallos de concepto. Es conveniente preparar entornos de pruebas o simulaciones para poder corregir estos fallos que son difíciles de percibir durante los entrenamientos del agente, y realizar estas pruebas en busca de ellos cuando el agente no es capaz de entrenar los comportamientos deseados.

6. Conclusiones

Pare este proyecto nos planteamos los siguientes objetivos:

1. Investigar y aprender sobre las técnicas usadas para resolver este tipo de problemas.
2. Preparar un entorno de pruebas funcional y consistente.
3. Conseguir un agente que sea capaz de seguir una animación estática que consista en mantenerse de pie en equilibrio.
4. Conseguir que nuestro agente sea capaz de seguir animaciones con movimiento pero sin desplazamiento.
5. Conseguir que el agente sea capaz de seguir animaciones con movimiento y con desplazamiento, con especial interés en que sea capaz de andar naturalmente hacia delante.

La primera etapa del proyecto consistió en familiarizarnos con los fundamentos del aprendizaje por refuerzo y con las herramientas con las que íbamos a trabajar. Durante el proceso de investigación encontramos trabajos muy relacionados con el nuestro, en concreto el artículo ***DeepMimic*** [21], que se convertiría en una guía e inspiración.

El objetivo primordial del proyecto consistía en convertir animaciones tradicionales, que funcionan al margen de una simulación física, en **animaciones basadas en físicas**, donde los movimientos se producen sujetos a una simulación física. Esto significa, que junto con las herramientas para trabajar con aprendizaje reforzado, necesitamos una forma de representar estas animaciones en pantalla, y una herramienta que nos proporcione esta simulación física.

Este proyecto se sostiene sobre un *plug-in* para el motor de videojuegos **Unity** llamado **ML-Agents**. Por un lado, *Unity* nos proporciona la infraestructura necesaria sobre la cual construir el proyecto: el motor de simulación física y el motor de renderizado 3D para poder visualizar a nuestros modelos y animaciones. Por otro lado, el aprendizaje por refuerzo nos lo facilita *ML-Agents*, que consiste en un conjunto de extensiones para *Unity* que permiten al usuario crear y configurar un entorno de aprendizaje por refuerzo dentro del motor. Éste se comunica luego con *PyTorch*, una librería que proporciona las implementaciones de los algoritmos de aprendizaje por refuerzo.

Una vez familiarizados con estos elementos, nos dispusimos a preparar el entorno de pruebas sobre el que luego experimentaríamos. Una de las partes más importantes es la preparación del modelo 3D con forma humanoide que será el cuerpo de nuestro agente. Este fue configurado como ***ragdoll***, es decir, un personaje con cada una de sus partes del cuerpo simuladas físicamente, de forma que cada una de sus articulaciones fueron configuradas por nosotros para que sean coherentes con sus análogas en el cuerpo humano, y se les atribuyó a cada una de las partes del cuerpo un peso que mantuviese una proporcionalidad similar a la humana (ver 6.1).

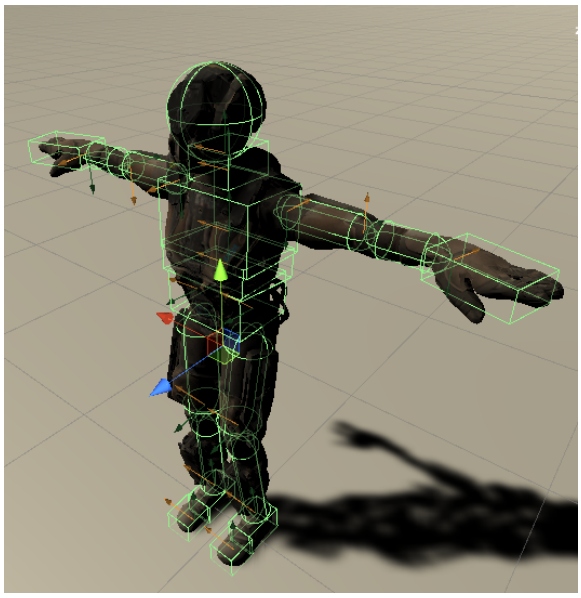

Figura 6.1: Personaje configurado como ragdoll.

Con los conocimientos necesarios en la materia y nuestro modelo físico preparado, comenzamos con el desarrollo y la experimentación enfocados en nuestro primer objetivo; conseguir un agente que sea capaz de mantenerse de manera estática en una postura bípeda erguida. Esto implica que el agente ha de aprender a mantener el equilibrio sobre sus piernas de la manera más natural posible.

El proceso para conseguir este objetivo es la etapa del proyecto en la que se ha requerido invertir más tiempo y que ha involucrado más revisiones. En los primeros compases del desarrollo conseguimos un agente que era capaz de permanecer erguido sobre sus piernas, pero el entrenamiento era lento e irregular. Además, aunque podía estar de pie, el agente necesitaba corregir su posición constantemente para no caerse, lo que estaba lejos de parecerse a la animación de referencia completamente estática. Tras esta primera etapa, nuestros esfuerzos se centraría en corregir estos dos problemas, la estabilidad del entrenamiento y el comportamiento final del agente entrenado.

Realizando numerosas pruebas y cambios en la simulación, la configuración de hiperparámetros, y la función de recompensa, conseguimos que el entrenamiento fuera más rápido y estable. Sin embargo, aunque el comportamiento era superior a los anteriores, no conseguíamos mejorarlo lo suficiente como para que el agente se mantuviese en un equilibrio estático natural, sino que "temblaba" de manera evidente debido a las constantes fuerzas de corrección que necesitaba aplicar para no caerse.

Finalmente, la solución que encontramos fue sustituir la función de recompensa por otra basada en ***DeepMimic*** [21] y modificar la simulación física de forma que se actualizase más veces por segundo (suponiendo una mayor exigencia computacional). Estos cambios, junto con otras modificaciones en los hiperparámetros y el entorno, resultaron en el comportamiento que buscábamos: **un agente capaz de mantener el equilibrio de manera estática y natural**. Además el agente era capaz de seguir este comportamiento con cierta robustez, de forma que soporta ciertos tirones y empujes en diferentes partes del cuerpo, pero pasando a un comportamiento errático cuando estas

perturbaciones lo alejan demasiado de los estados visitados durante el entrenamiento.

Tras eso, dimos paso a la siguiente fase del proyecto, en la que buscamos cumplir el objetivo de que el agente fuera capaz de seguir animaciones con movimiento, pero que no involucrasen desplazarse de su posición. Partiendo del agente que era capaz de mantener el equilibrio, la experimentación para este objetivo dio resultados rápidamente. Fue capaz de seguir tanto una animación en la que solo movía los brazos como otra en la que realizaba sentadillas con todo el cuerpo (ver 6.2). Para ello bastaron unos pocos cambios y ajustes a la configuración que habíamos utilizado para el equilibrio estático,y también era capaz de soportar ciertas perturbaciones, lo cual nos dio a entender que habíamos conseguido un modelo bastante robusto.

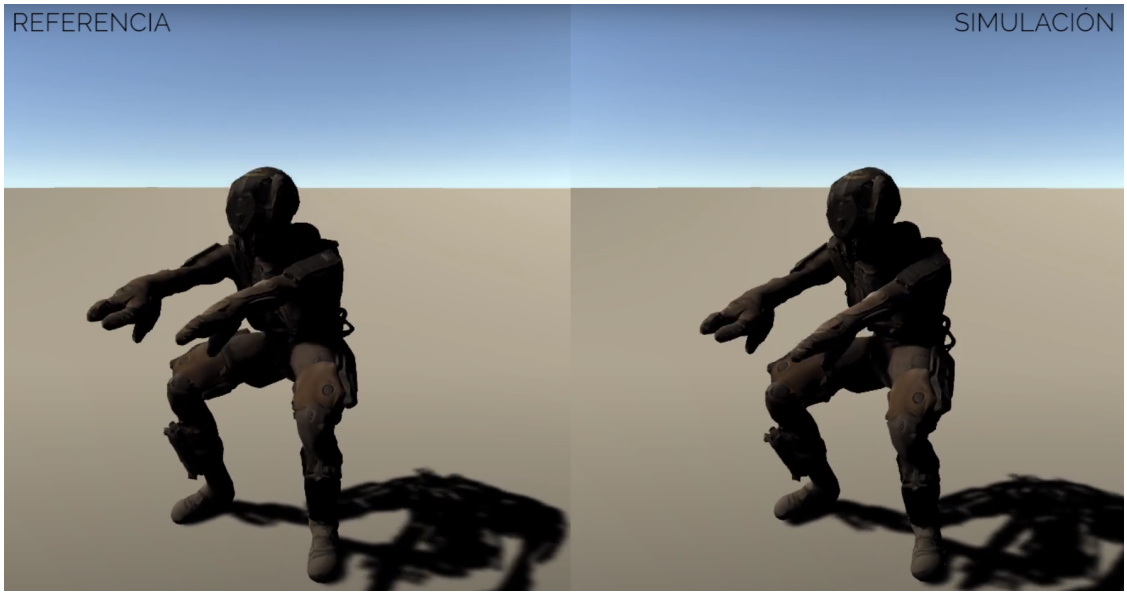

Figura 6.2: Agente siguiendo la animación de sentadilla

Siendo capaz de seguir animaciones estáticas, nos lanzamos al ambicioso objetivo de que el agente fuera capaz de seguir una animación de caminar, la cual implica movimiento y desplazamiento. Sin embargo, tras probar con diferentes cambios y configuraciones, no fuimos capaces de conseguir que el agente siguiese la animación de referencia de manera natural. En el mejor entrenamiento conseguido el agente era capaz de desplazarse hacia delante, pero con movimientos erráticos que estaban lejos de asemejarse a la referencia.

Una demostración de los resultados que hemos ido obteniendo a lo largo del proyecto pueden encontrase en el siguiente vídeo resumen.

No podemos concluir que las animaciones basadas en físicas estén a la altura de otras técnicas de generación procedimental de animaciones, pues el coste en tiempo y esfuerzo de desarrollar un agente como el nuestro es muy elevado. Tras varios meses de desarrollo y experimentación hemos conseguido un modelo capaz de seguir ciertas animaciones, pero que asumimos que es incapaz de ejecutar muchas otras. Por lo que si nuestro objetivo fuese conseguir un personaje animado físicamente con un conjunto de animaciones complejas, su proceso de desarrollo sería demasiado costoso.

Sin embargo, viendo los avances que hemos obtenido, creemos que sería posible

implementar un agente (aunque quizá no en el contexto de *ML-Agents*) lo suficientemente polivalente como para ser capaz de seguir cualquier tipo de animación de referencia. Éste requeriría un proceso de configuración por parte del usuario para adaptarse a distintos modelos 3D, distintas animaciones de referencia, y distintas configuraciones de la simulación física. Dicho agente podría ser extremadamente útil para desarrolladores en el contexto adecuado, y creemos que nuestro trabajo, aunque está muy lejos de presentar estas características, apunta en la dirección del desarrollo de tal herramienta.

Por otro lado, nuestra experiencia con *ML-Agents* nos ha llevado a tener un gran interés por la herramienta. Como nuestro trabajo demuestra, cuando el comportamiento que se quiere obtener de un agente es muy complejo, la dificultad de implementarlo es muy elevada. Sin embargo, nuestro proceso de familiarización con la herramienta nos llevó a observar que cuando el comportamiento objetivo es algo más simple, la configuración de un agente para ejecutarlo es sencilla y los entrenamientos necesarios no son especialmente exigentes.

Por tanto, para comportamientos sencillos, donde la otra opción sería desarrollar una inteligencia artificial que describiese dicho comportamiento, una implementación de un agente con *ML-Agents* puede resultar mucho más rápida, fácil, y dar resultados más fiables y naturales. Esto resulta especialmente interesante en el contexto de los videojuegos, donde una IA predecible puede romper por completo la experiencia de juego.

La accesibilidad de la herramienta y la aproximación al aprendizaje automático que involucra, también despierta nuestro interés en el ámbito docente. Como estudiantes del Grado de Desarrollo de Videojuegos, creemos que haber trabajado con esta herramienta y familiarizarnos con el aprendizaje por refuerzo hubiera sido interesante durante nuestro paso por el grado, y la mayoría de conocimientos necesarios para utilizarla; como *Unity* o la programación en *C#* ya están presentes a lo largo del periodo académico.

6.1. Trabajo futuro

Tras conseguir que el agente sea capaz de seguir la animación de andar de manera natural intentaríamos mejorar el agente para que sea capaz de aprender esa misma animación de andar pero en contextos más variados en el entorno, como distintas superficies, obstáculos e incluso con diferentes direcciones objetivo. El objetivo de esto sería que el agente no solo aprendiese a caminar, sino también desarrollase comportamientos que diesen lugar a animaciones nuevas derivadas de la original. Es decir, que además de conseguir animaciones basadas en físicas también obtendríamos nuevas animaciones generadas procedimentalmente.

Tras ello, el siguiente posible objetivo a seguir sería intentar orientar el proyecto a desarrollar esa posible herramienta que mencionábamos en las conclusiones, es decir, un agente configurable para distintos modelos 3D, animaciones y simulaciones físicas.

Aunque un proyecto así debería ser seccionado en objetivos más manejables. Siendo probablemente uno de éstos el adaptar y experimentar con nuestro agente para

trabajar con modelos 3D distintos al que hemos utilizado y con nuevas animaciones.

Appendices

Glosario

articulación Conexión física entre dos cuerpos sólidos que limita el movimiento y/o la rotación entre ellos. Por ejemplo, se puede conectar la parte de arriba del brazo con el antebrazo mediante una articulación, y sólo permitir el movimiento entre ellos alrededor de un eje y en un rango concreto, imitando así el comportamiento de una articulación real.. vii

centro de masas (CoM) El centro de masa de un sistema es la posición media de todos sus componentes ponderados respectos su masas. La idea fundamental del equilibrio de un cuerpo es mantener su centro de masas justo encima del punto de apoyo, por ello es de suma importancia en nuestro proyecto. Lo llamaremos CoM (*Center of Mass*) de aquí en adelante.. vii

centro de soporte (CoS) Es el punto medio de todos los puntos de apoyo del sistema. En nuestro caso, será generalmente el punto medio de los dos pies de nuestro modelo. Lo llamaremos CoS (*Center of Support*) para abreviar.. vii

frame Los videojuegos, así cómo muchos otros procesos computacionales, constan de un bucle principal que se ejecuta infinitamente mientras la aplicación esté activa. Cada una de las iteraciones que se realiza sobre este bucle genera una nueva imagen para ser mostrada al usuario. Cada una de estas imágenes se conoce como *frame*, aunque el término también se puede referir a la propia iteración del bucle en vez de a la imagen generada. Este término es ampliamente utilizado en el desarrollo de videojuegos, pues es de vital importancia en multitud de aspectos; desde la implementación de algoritmos hasta el aspecto artístico.. vii, 22

GameObject Nombre que *Unity* le da a cada una de las entidades de un juego: un personaje, un vehículo, un emisor de sonido, una luz.... vii, 25

PID Un controlador PID (Proportional Integral Derivative) es un mecanismo de control continuo que se utiliza en multitud de aplicaciones industriales. Su objetivo es hacer que un valor alcance (o se mantenga en) un valor o rango de valores concreto, para lo cual utiliza una señal de feedback que le permite reajustar constantemente su salida. En el apartado 3.6 explicamos en detalle su funcionamiento y la utilidad dentro de este proyecto.. vii, 22

ragdoll Personaje simulado físicamente. Es un conjunto de cuerpos sólidos unidos por articulaciones. Cada cuerpo sólido se corresponde con una parte del cuerpo, por ejemplo la mano es un cuerpo sólido que está unido al antebrazo por una articulación. Aplicando torques en estas articulaciones podemos hacer que el personaje se mueva de determinada forma, pudiendo llegar a replicar animaciones.. vi, vii, 2, 21, 27, 30, 38, 40, 41, 61, 62, 68, 76, 83

referencia Personaje que sigue una animación estándar y al que el cuerpo físico intenta imitar.. vii, 38

vector de acción Datos de salida de la red neuronal que representan la acción A escogida por el modelo ante el estado S. Se trata de la rotación objetivo de cada una de las articulaciones del ragdoll, es decir, la pose objetivo.. vii

vector de observación Datos de entrada de la red neuronal que representan el estado actual S; formado por la posición, rotación y velocidades lineal y angular de cada parte del cuerpo físico.. vii

A. Introduction

A.1. Motivation

One of the most striking things about a video game is its animations[7]. How the characters move, how they interact with the world and with each other. It is something that, as we are so used to seeing in the real world, we immediately notice when we see it represented on the screen. Anyone, without knowing anything about videogames, can discern between good and bad animation at a glance.

Animators, in a job that requires long hours, skill and attention to detail, do just that, making the movement of characters fluid and natural to the player's eye. In recent years, the technique of motion capture has become very popular.[25], which consists of digitalising the movements of an actor, obtaining animations that can then be reproduced digitally, as can be seen in the figure A.1.This achieves a much more realistic movement than traditional animation techniques, although it is very costly for video game studios.

Whatever the method used to create animations, all of them present a difficult problem to solve, and that is the lack of adaptability to different situations. No matter how much money and time a company has, it will never be able to create animations for all the possible interactions a character could have in an environment, as these are infinite. For example, if a character has to sit on a chair, the animation can have infinite variations depending on the direction from which it is approached, the size of the chair, the height of the backrest, the evenness of the terrain, the animation that was previously being performed, etc.

To solve this problem, procedural animation generation techniques are used. These consist of generating animations in real time based on the environment and the interaction that the character is performing [3]. In the case of the chair, a procedural animation generation algorithm could take all these variables into consideration and generate an animation that adapts to the situation.

But even procedurally generated animations still present another major problem, and that is that they are reproduced outside the physics simulation. The vast majority of video games with animations also use a physics engine that simulates the world around the characters, but the animations are left out of this simulation. For example, even if a character's arm collides with a table, the animation is unaffected, resulting in an inconsistent experience that breaks the player's immersion to some extent.

The lack of connection between the animations and the physical world is not so obvious to the naked eye, but it is still an aspect that needs to be improved in most games. It is for this reason that we started experimenting with physical character simulation, where each body part is a separate solid body, which is connected to each other by joints. These characters became known as ragdolls, which is what their inert behaviour is reminiscent of [1], as can be seen in the figure A.2.

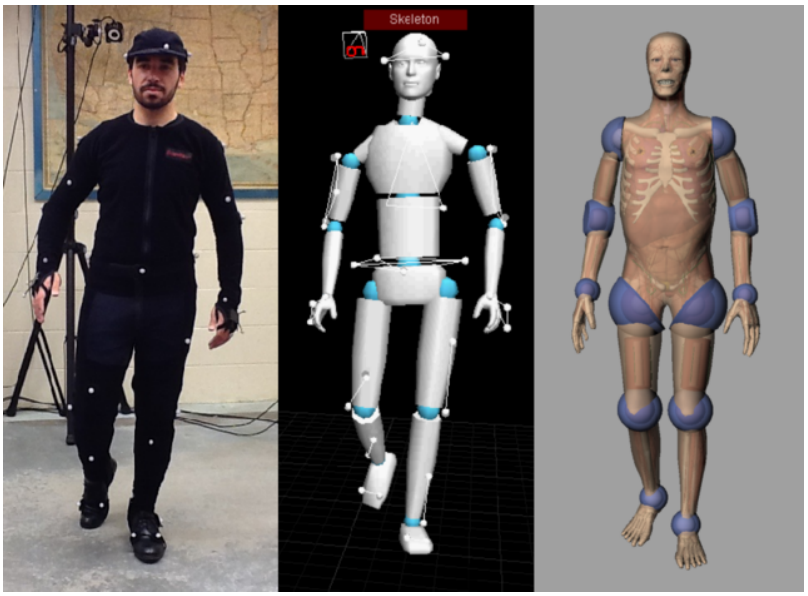

Figure A.1: *Motion Capture*

Figure A.2: *Example of a ragdoll character in Unreal Engine*

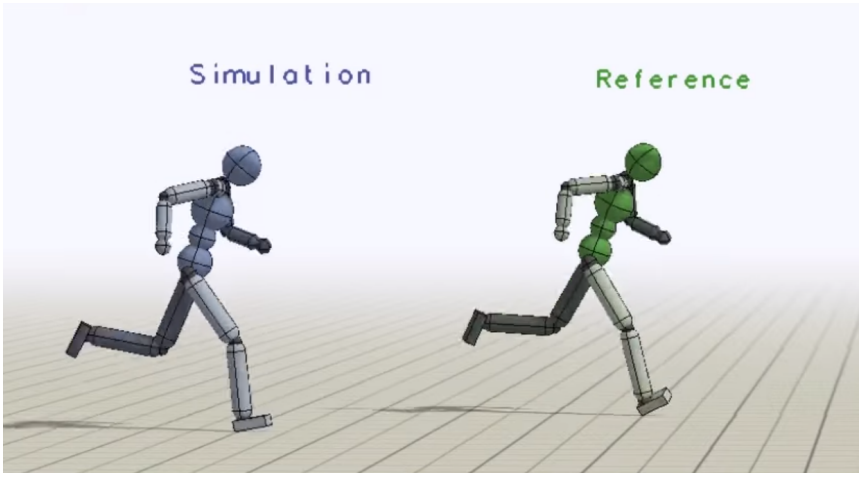

Figure A.3: *Comparison between reference and ragdoll from DeepMimic article [21]*

The next step is natural, to animate ragdolls by applying forces to each of their joints, and this is where our work begins. The idea is, starting from a humanoid model, to create some basic animations and configure it as a ragdoll.

On the one hand we will have an animated model of the traditional form, which we will call the reference, and which we will make invisible (although it will continue to move). On the other hand, we will have the ragdoll, whose objective will be to imitate the movement of the reference, but applying forces to each of its joints. To do this, it will not only have to learn to match the reference pose at all times, but also to do so while maintaining its balance and adapting to the changes that the environment may present. In figure A.3 we can see both bodies in operation.

This is a problem that has been solved before, but the lack of reliability and unpredictability of the models obtained limits their implementation in commercial games. Our goal is to investigate this technology and see how far we can take it using the Unity game engine. [28].As this is one of the most widely used engines in the industry, we can also see how accessible these techniques are for development studios today, as well as the difficulties in using them.

A.1.1 Challenges

This is a very complex problem and requires advanced knowledge of both physics and reinforcement learning. There is very little information on the subject, and most of it is concentrated in the form of very complex academic papers by groups of experts in the field.

On the one hand, we have a very large neural network that takes several hours to train, so testing any changes or possible improvements to it is a time-consuming process, which greatly limits the speed at which we can make progress. In addition, the reinforcement learning techniques we use are by no means simple, so you need to know what changes to make to the model so you don't waste time training with a bad setup.

On the other hand, the results do not depend only on reinforcement learning, but are also influenced by the characteristics of the environment. Parameters such as the gravity of the world, the refresh rate, the weight of the model or the shape of its limbs greatly affect how the agent has to act. This is where knowledge of the physics engine and its inner workings comes into play, and knowing how to configure it is just as important as knowing how to configure the reinforcement learning part.

This adds an extra layer of difficulty that makes it difficult to know, when problems arise during the project, whether their origin lies in the network design, the environment design, or both.

In terms of reinforcement learning, we are dealing with a continuous control system that has many degrees of freedom, 38 in the case of our model. Moreover, the movement of a single joint has repercussions far beyond the joint itself. For example, an inappropriate movement of the shoulder can generate a very large velocity in the hand; or a small error in turning the foot can cause the character to fall. This makes it difficult to get the precise result we are looking for, and we have often encountered instability in the learning process and results with unnatural movements far from the desired ones.

A.1.2 Applications

The technologies we use and our potential outcomes have obvious applications in game development. Proof of this is the widespread use of procedural content generation, which allows developers to endow their games with an almost infinite amount of content unique to the context in which they use it, be it in the generation of scenarios, events or other game elements [14].Procedural animation generation has begun to take on an equally important role in the industry in recent years. More and more games are implementing this type of technique, and this trend will continue to become a standard in video game creation.

However, its applications go far beyond entertainment, as the use of reinforcement learning for motor control is strongly related to robotics, a field in which similar techniques [17] are also being developed in parallel. While this is beyond the scope of this project, it is also of great interest to us, as it could lead to a new generation of robots capable of moving over any terrain and recovering from any disturbance.

A.2. Objectives

The ultimate goal of the project would be to develop an agent that is capable of following complex animations, and also has considerable resistance to external disturbances. However, with this goal on the horizon, we have first to meet intermediate, staggered objectives:

1. **Research and learn about the techniques used to solve this type of problem**in related work. This is a fundamental part of the project as it will lay

Figure A.4: *Boston Dynamics*'s Atlas Robot, one of the most important references for motor control in robotics.

the theoretical foundations on which we will then carry out the practical part of the project.

2. **Set up a functional and consistent testing environment**, with a well-developed physical ragdoll with which we are able to carry out testing and training.
3. The next objective would be to achieve an agent that is able to **follow an animation consisting of standing in balance**, as naturally and statically as possible.
4. Next, our goal is to get our agent to be able to **follow animations without displacement**, i.e. that involve some movement of the model but do not require the model to move around the scene.
5. Next, we will try to get the agent to be able to **follow animations with displacement**, with special interest in the agent being able to walk naturally forward, having learnt to keep balance, follow the animation and move at the same time.

A.3. Planning and methodology

Having decided how the division of labour should look like in order to achieve the different objectives, we propose a possible planning. It is visually represented in the figure A.5. The first few weeks will be devoted entirely to researching related work and preparing the environment. The next three months we will focus on the most difficult and therefore the most time-consuming part of the work, obtaining a first agent capable of standing in static equilibrium. Once this has been achieved, we will spend about a month making the appropriate changes so that this model also works with dynamic animations. Finally, in the last two months we will try to make the model work also for dynamic animations with displacement.

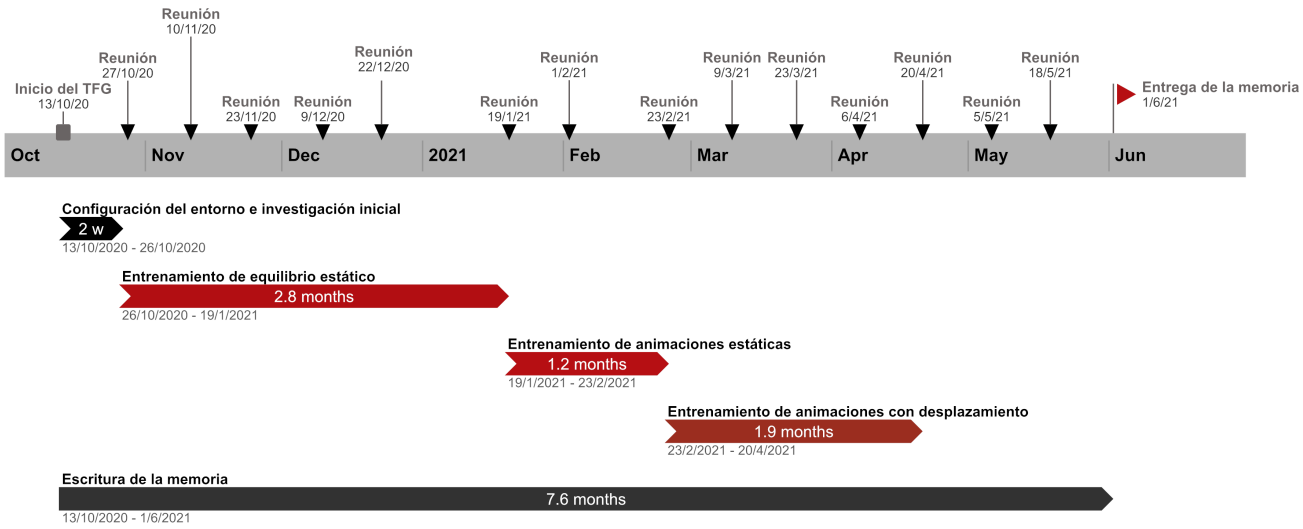

Figure A.5: Timeline

Together with our director, we have agreed to meet every two weeks during the whole period of the TFG until the June call. These meetings will be held using *Google Meet* and will be aimed at informing the director of the progress since the previous meeting, as well as setting goals to try to achieve for the next one. During the last month we decided to have a weekly meeting to have a more constant critique of the report and to be able to obtain a better final result.

As a version control system we used *GitHub*, and as a way of sharing documents *Google Drive*, which was also used at the beginning for the writing of this report and later migrated to *Overleaf*.

[1] [2] [3]

As there are only two of us, daily communication about the project was all we needed to organise ourselves, agreeing in the most reasonable way on the workload of each of us in each situation.

A.4. Report structure

This report consists of 6 chapters, the contents of which are described below:

[1] www.github.com
[2] drive.google.com
[3] www.overleaf.com

In this chapter 1 we set out the problem we are going to face: why it is important, what difficulties it presents, what challenges and objectives we have, and how we are going to organise ourselves to achieve them.

Chapter 2 compiles all the theoretical knowledge about reinforcement learning that we have used throughout the project, as well as the different algorithms available to us and their theoretical foundations.

Chapter 3 describes each of the different main elements underpinning the practical part of the project, such as the functioning of the physical simulation we will use or the workings of the reinforcement learning tool.

In chapter 4 we explain the final model we have achieved and the configurations used in all relevant aspects of the project, as well as the design decisions that led to them.

Chapter 5 is a structured compilation of all the experiments we have done over the months, where we also explain the difficulties we have encountered along the way and how we have solved them.

Chapter B outlines the main conclusions we have drawn from the development of the project, accompanied by possible ideas for future work on this project.

A.5. Source code

Both the source code and all the resources we have used can be found in the following repository github.com/sergioabreu-g/physical-animations under the Apache 2.0 license.

B. Conclusions

For this project, we set the following goals:

1. To research and learn about the techniques used to solve this kind of problems.
2. To prepare a functional and consistent test environment.
3. To get an agent that is able to follow a static animation that consists of standing in balance.
4. To get the agent to be able to follow animations with movement but without displacement.
5. To get the agent to be able to follow animations with movement and with displacement, with special interest in the agent being able to naturally walk forward.

The first stage of the project was to familiarise ourselves with the basics of reinforcement learning and the tools we were going to work with. During the research process, we came across works closely related to ours, in particular the article **DeepMimic** [21], which would become a guide and inspiration.

The primary goal of the project was to convert traditional animations, which operate outside of a physical simulation, into 'physics-based animations', where movements are produced subject to a physical simulation. This means that, along with the tools to work with reinforcement learning, we need a way to represent these animations on screen, and a tool to provide us with this physical simulation.

This project is based on a *plug-in* for the **Unity** videogame engine called **ML-Agents**. On the one hand, *Unity* provides us with the necessary infrastructure on which to build the project: the physics simulation engine and the 3D rendering engine to visualise our models and animations. On the other hand, reinforcement learning is provided by *ML-Agents*, which consists of a set of extensions for *Unity* that allow the user to create and configure a reinforcement learning environment within the engine. This then communicates with *PyTorch*, a library that provides the implementations of the reinforcement learning algorithms.

Once we were familiar with these elements, we set about preparing the test environment that we would later experiment on. One of the most important parts is the preparation of the 3D humanoid-shaped model that will be the body of our agent. This was configured as a ragdoll, i.e. a character with each of its body parts physically simulated, so that each of its joints were configured by us to be consistent with their analogues in the human body, and each of the body parts was given a weight that maintained a human-like proportionality (see B.1).

With the necessary knowledge on the subject and our physical model ready, we start with development and experimentation focused on our first goal; to get an agent that is able to stand statically in an upright bipedal posture. This implies that the agent has to learn to balance on its legs as naturally as possible.

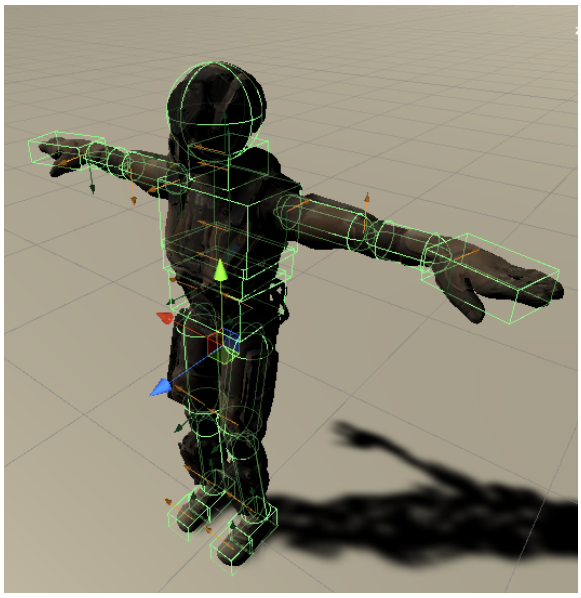

Figure B.1: fig:Character configured as a ragdoll

.

The process of achieving this goal is the most time-consuming and revision-intensive stage of the project. In the early stages of development we achieved an agent who was able to stand upright on his legs, but the training was slow and irregular. Moreover, although it could stand, the agent needed to constantly correct its position in order not to fall down, which was far from resembling the fully static reference animation. After this first stage, our efforts would focus on correcting these two problems, the stability of the training and the final behaviour of the trained agent.

By making numerous tests and changes to the simulation, the hyperparameter settings, and the reward function, we were able to make the training faster and more stable. However, although the behaviour was superior to the previous ones, we were not able to improve it enough to keep the agent in a natural static balance, but rather it "trembled" in an evident way due to the constant correction forces it needed to apply in order not to fall down.

Finally, the solution we came up with was to replace the reward function with one based on **DeepMimic** [21], and modify the physical simulation so that it would update more times per second (assuming a higher computational demand). These changes, along with other modifications to the hyperparameters and environment, resulted in the behaviour we were looking for: **an agent capable of maintaining balance in a static and natural way**. Furthermore, the agent was able to follow this behaviour with some robustness, so that it withstands certain pulls and pushes on different parts of the body, but then switches to erratic behaviour when these perturbations take the agent too far away from the states visited during training.

After that, we moved on to the next phase of the project, in which we sought to achieve the goal of the agent being able to follow animations with movement, but which did not involve moving from its position (no displacement). Starting with the agent being able to maintain its balance, experimentation for this goal quickly yielded

results. He was able to follow both an animation in which he only moved his arms and one in which he performed squats with his whole body (see B.2). A few changes and adjustments to the configuration we had used for static balancing were enough, and it was also able to withstand some perturbations, which suggested that we had achieved a fairly robust model.

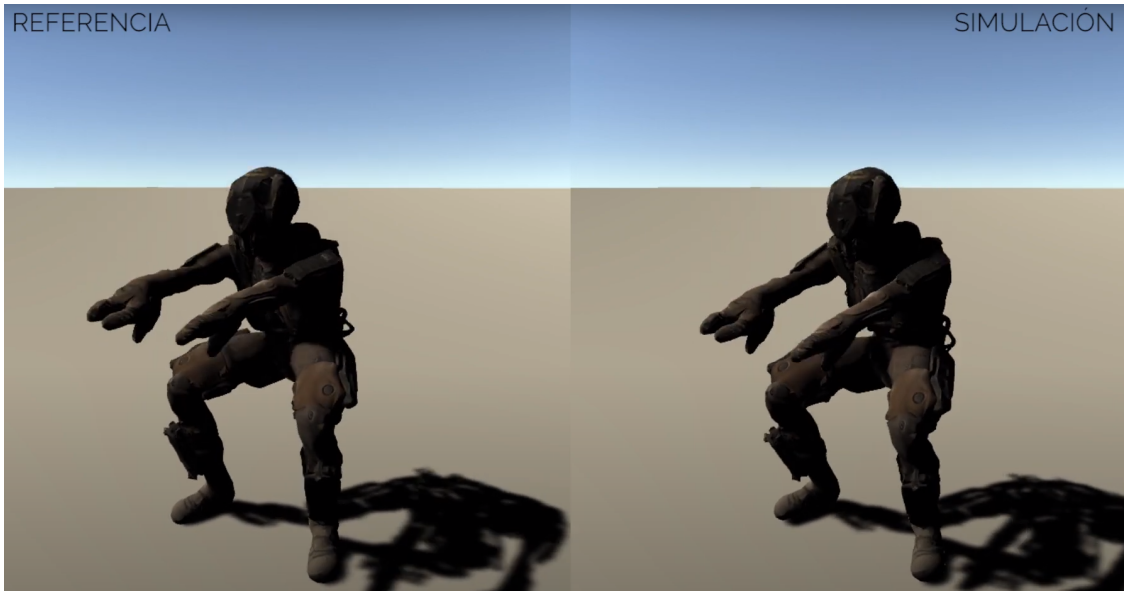

Figure B.2: Agent following the squat animation

Being able to follow static animations, we set ourselves the ambitious goal of making the agent able to follow a walking animation, which involves movement and displacement. However, after trying different changes and configurations, we were not able to get the agent to follow the reference animation in a natural way. In the best training achieved, the agent was able to move forward, but with erratic movements that were far from resembling the reference.

A demonstration of the results we have been obtaining throughout the project can be found in the following summary video.

We cannot conclude that physics-based animations are on a par with other procedural animation generation techniques, as the cost in time and effort to develop an agent like ours is very high. After several months of development and experimentation, we have achieved a model capable of following certain animations, but we assume that it is incapable of executing many others. So if our goal was to achieve a physically animated character with a complex set of animations, the development process would be too costly.

However, looking at the progress we have made, we believe that it would be possible to implement an agent (though perhaps not in the context of *ML-Agents*) that is sufficiently versatile to be able to follow any kind of reference animation. It would require a configuration process by the user to adapt to different 3D models, different reference animations, and different configurations of the physical simulation. Such an agent could be extremely useful for developers in the right context, and we believe that our work, although far from presenting these features, points in the direction of

developing such a tool.

On the other hand, our experience with *ML-Agents* has led us to develop a strong interest in the tool. As our work shows, when the desired behaviour of an agent is very complex, the difficulty of implementing it is very high. However, our familiarisation process with the tool led us to observe that when the target behaviour is somewhat simpler, the configuration of an agent to execute it is straightforward and the necessary training is not particularly demanding.

Therefore, for simple behaviours, where the other option would be to develop an artificial intelligence to describe the behaviour, an agent implementation with *ML-Agents* can be much faster, easier, and give more reliable and natural results. This is especially interesting in the context of videogames, where a predictable AI can completely break the gaming experience.

The accessibility of the tool and the approach to machine learning it involves also arouses our interest in the teaching field. As students of the Videogame Development degree, we believe that having worked with this tool and becoming familiar with reinforcement learning would have been interesting during our time in the degree, and most of the knowledge needed to use it, such as Unity or programming in C#, is already present throughout the academic period.

B.1. Future Work

After getting the agent to be able to follow the walking animation in a natural way, we would try to improve the agent so that it would be able to learn the same walking animation but in more varied contexts in the environment, such as different surfaces, obstacles and even with different target directions. The aim of this would be for the agent not only to learn to walk, but also to develop behaviours that give rise to new animations derived from the original one. In other words, in addition to getting physics-based animations, we would also get new procedurally generated animations

After that, the next possible objective to follow would be to try to orient the project towards developing the possible tool mentioned in the conclusions, i.e. a configurable agent for different 3D models, animations and physical simulations.

However, such a project would have to be broken down into more manageable objectives. Probably one of these would be to adapt and experiment with our agent to work with 3D models other than the one we have used and with new animations.

C. Contribuciones

C.1. Daniel Álvarez Castro

Al comienzo del proyecto, junto con Sergio tuvimos una fase de familiarizarnos con los elementos clave el aprendizaje por refuerzo que íbamos a necesitar para este proyecto, así como explorar las diferentes referencias de trabajos previos relacionados que Sergio había encontrado anteriormente.

Tras eso, mi objetivo consistió en investigar como funcionaba ML-Agents. Nos topamos con ML-Agents antes de dar comienzo al proyecto y ya teníamos decidido que íbamos a intentar trabajar con ello para antes de comenzar el proyecto. Lo primero fue explorar el proceso por el que se instala ML-Agents como un plug-in en Unity y la instalación de los paquetes de Python que necesita la herramienta, que posteriormente retransmitiría a Sergio para que ambos tuviésemos instalada la herramienta.

Después, ML-Agents cuenta con una documentación detallada en la que se incluye una guía de como dar los primeros pasos por la herramienta, que consiste en usar una escena preparada con un agente ya entrenado y reentrenarlo para familiarizarse con como se utiliza la herramienta en el contexto del editor de Unity y como se lanza un entrenamiento.

A continuación, el proceso me llevó a una sección para aprender a como crear un entorno de entrenamiento y a crear un pequeño agente de 0 de manera guiada. Tras completarlo, me dediqué a explorar los diferentes entornos y agentes de ejemplo que vienen incluidos con la herramienta, lo que nos dio una inspiración inicial sobre como podríamos plantear el nuestro.

Con ML-Agents ya preparado me dispuse a buscar un modelo con que el que pudiésemos trabajar, hasta que encontré el que usaríamos en un un principio. Con un modelo a mano me dediqué a configurar sus colisiones y sus articulaciones para convertirlo en el *ragdoll* que necesitábamos. Este primer modelo inicial y esta configuración de articulaciones acabaría siendo sustituida y modificada por Sergio en futuras etapas del proyecto.

Con el entorno de pruebas listo y Sergio trabajando en el componente agente me dispuse a documentar los conceptos sobre el aprendizaje reforzado que habíamos explorado anteriormente, dicha recopilación ahora forma parte de los conceptos discutidos en el capítulo 2. También documenté algunos conceptos sobre ML-Agents que ahora forman parte del contenido del capitulo 4.

Poco después de empezar las pruebas con los entrenamientos, fuimos conscientes de que estos daban muchos problemas de rendimiento en mi máquina mientras en la de Sergio se realizaban sin muchos problemas aparentes, así que desde este punto, Sergio se encargaría del grueso de crear el componente agente y de ejecutar los entrenamientos, mientras yo me encargaba de tareas más auxiliares. De cualquier forma Sergio y yo nos reuníamos a menudo para discutir como mejorar la función de recompensa y otros

aspectos del componente agente.

Cuando decidimos que había que reconfigurar los pesos de las partes del cuerpo del modelos, yo me encargué de encontrar la referencia necesaria para calcular este peso, y de configurar una serie de agentes de prueba con distinto peso máximo para comprobar con cual de ellos obteníamos el mejor resultado.

Cuando empezamos a utilizar la función de recompensa de *DeepMimic* [21], en el componente agente me dediqué a realizar una serie de pruebas para comprobar que las distintas subrecompensas devolvían valores coherentes según el comportamiento del agente.

A mediados del desarrollo, me dispuse a realizar unas pruebas en paralelo con Sergio en las que utilizaríamos un modelo cuadrúpedo con forma perro para probar si nuestro agente funcionaba en dicho modelo. Sin embargo estás pruebas no llevaron a ningún lado, ya que el modelo utilizado era poco funcional, y un avance con el modelo principal desvió nuestra atención de vuelta a este.

Aunque partíamos de unas animaciones simples que desarrolló Sergio para hacer pruebas, cuando nos hicieron falta nuevas animaciones para seguir avanzando, me dediqué a crearlas en Blender. De esto salió la animación de hacer sentadillas, la animación de caminar y una animación de gatear con la que estuvimos experimentando.

Cuando redoblamos nuestra atención en la escritura de la memoria, me dediqué preparar el entorno y el documento en LaTeX que hemos utilizado en esta memoria y a escribir el grueso de los capítulos 1, 3, 4, 5 y 6, los cuales fui editando y expandiendo conforme recibíamos retroalimentación sobre ellos de Antonio. Mientras que Sergio se encargaba del capítulo 2, corrección general e inclusión de referencias.

C.2. Sergio Abreu García

Durante las fases iniciales del proyecto, me encargué de la búsqueda de referencias y trabajos similares a lo que nosotros queríamos hacer. Mucha de la bibliografía referenciada la encontré en este proceso, en el que destaco el artículo *DeepMimic* [21], que ha sido nuestra guía principal durante todo el proyecto. Tanto yo como Daniel, estuvimos los primeros meses del proyecto buscando trabajos en los que poder apoyarnos, viendo cómo hacían las cosas e intentando aprender los fundamentos de aquellos que tenían más experiencia que nosotros.

En esta fase, también buscamos ambos información sobre aprendizaje reforzado y los diferentes algoritmos que existían. Gracias a ello pudimos fundamentar el proyecto en una base teórica más sólida, y escoger las técnicas que más nos favorecían para el tipo de problema que teníamos ante nosotros. Fue así como dimos con *ML-Agents* para *Unity*, y empezamos a encaminar el trabajo hacia algo más concreto.

Un poco más adelante ya comenzamos a separar más el trabajo de cada uno. Mientras Daniel conseguía las herramientas de trabajo y hacía las primeras pruebas con ellas, yo me encargué de profundizar en los proyectos que más nos habían llamado la atención con objetivos similares al nuestro. Así, saqué las primeras ideas de que dirección tomar respecto a cosas más concretas como: función de recompensa, configuración física, espacio de acciones y observaciones...

Profundicé especialmente en el ya mencionado artículo *DeepMimic* [21]. Me gustó por dos motivos: el primero es que algunos de sus objetivos eran prácticamente idénticos a los nuestros; y el segundo es que sus resultados eran muy buenos. Lo leí varias veces y le expliqué a Daniel lo que había aprendido. Por todo ello, lo terminamos utilizando como la referencia principal durante el desarrollo del proyecto.

Cuando ya estábamos más familiarizados tanto con las herramientas como con la base teórica, comencé con la creación de los *scripts* '*CharacterAgent*' y '*Bodypart*', y es ahí dónde se concentran la mayor parte de mis esfuerzos del proyecto. Dado que nuestro proyecto no tiene una gran cantidad de código que gestionar, nos bastó con mantener dos *scripts* bien organizados. Primero creé '*CharacterAgent*', dónde se concentra el código importante de nuestro proyecto. Más adelante, creé otro *script* '*BodyPart*' para liberar a '*CharacterAgent*' de funciones auxiliares que necesitaban ejecutarse sobre cada parte del cuerpo.

En '*CharacterAgent*', me encargué de crear el código y realizar las pruebas pertinentes, dado que mi máquina me permitía ejecutar los entrenamientos mucho más rápido que a Daniel. Así pues, programé las funciones principales: cálculo de recompensa, recogida de observaciones, ejecución de acciones y comprobación de las condiciones de terminación. Mientras que Daniel se encargaba de la creación de este documento y de las explicaciones, yo me dedicaba a realizar cambios sobre el modelo, ejecutar los entrenamientos y guardar los resultados.

También considero importante, aunque no aporte funcionalidad, la organización de las variables de configuración. Organicé las variables en secciones, de manera que desde el editor de *Unity* pudiesen modificarse de manera intuitiva.

Muchas de estas variables, aunque fácilmente accesibles, afectaban a otras partes del proyecto que necesitaban ser reconfiguradas de manera síncrona, lo cual nos hacía perder tiempo en un trabajo tedioso. Para solucionarlo me encargué de implementar una función de ayuda que se encarga de, cada vez que se realiza un cambio en alguna de las variables del archivo, realizar los cambios pertinentes al proyecto para ahorrar trabajo manual.

Además, también implementé otra función de ayuda que se encarga de detectar automáticamente las partes del cuerpo del ragdoll utilizado, y asignarles un componente '*BodyPart*' automáticamente en el caso de que no lo tengan. De nuevo, esto nos ahorró mucho trabajo manual, especialmente cuando realizábamos cambios sobre el esqueleto del modelo.

Me encargué también de realizar los ajustes sobre el entorno y la simulación física, dado que estaba realizando pruebas constantes sobre ellos. Fue por esto que necesité aprender cómo funcionaba la simulación física internamente, así que busqué información sobre el tema que luego nos ayudaría a escribir la memoria. Sabiendo esto, pude realizar los ajustes sobre los diferentes parámetros de configuración de la simulación: frecuencia de actualización, precisión de las aproximaciones, límites de velocidades...

Aunque de la configuración del modelo se encargó principalmente Daniel, también realicé algunos ajustes menores cuando veía que presentaban algún error durante los entrenamientos.

En cuanto a la creación de la memoria, la mayor parte del contenido lo escribió Daniel. Yo me encargué del capítulo de 2, de la introducción, de realizar revisiones generales del documento, y de incluir referencias e imágenes.

D. Bibliografía

[1] Jadon Barnes. Fun with ragdolls 4, 2018. URL https://www.youtube.com/watch?v=oB33HLoceiw.

[2] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, Jun 2013. ISSN 1076-9757. doi: 10.1613/jair.3912. URL http://dx.doi.org/10.1613/jair.3912.

[3] Armin Bruderlin. Procedural motion control techniques for interactive animation of human figures. *Simon Fraser University*, 37(4), 1995.

[4] Michal Certicky, David Churchill, Kyung-Joong Kim, Martin Certicky, and Richard Kelly. Starcraft ai competitions, bots and tournament manager software. *IEEE Transactions on Games*, PP:1–1, 11 2018. doi: 10.1109/TG.2018.2883499.

[5] Simon Clavet. Machine learning summit: Ragdoll motion matching, 2020. URL https://www.youtube.com/watch?v=JZKaqQKcAnw. Last accessed: 08-06-2021.

[6] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation, Stichting Blender Foundation, Amsterdam, 2018. URL http://www.blender.org. Last accessed: 08-06-2021.

[7] Jonathan Cooper. *Game Anim*. CRC Press, 2021.

[8] Nvidia Corporation. Physx, 2021. URL https://github.com/NVIDIAGameWorks/PhysX. Last accessed: 08-06-2021.

[9] Paolo de Leva. Adjustments to zatsiorsky-seluyanov's segment inertia parameters. *Journal of Biomechanics*, 29(9):1223–1230, 1996. ISSN 0021-9290. doi: https://doi.org/10.1016/0021-9290(95)00178-6. URL https://www.sciencedirect.com/science/article/pii/0021929095001786.

[10] Kenny Erleben, Jon Sporring, Knud Henriksen, and Kenrik Dohlman. *Physics-Based Animation (Graphics Series)*. Charles River Media, Inc., USA, 2005. ISBN 1584503807.

[11] T. Geijtenbeek, N. Pronost, and A.F. van der Stappen. Simple Data-Driven Control for Simulated Bipeds. In *Eurographics/ACM SIGGRAPH Symposium on Computer Animation*, pages 211–219, 2012.

[12] Thomas Geijtenbeek, Michiel van de Panne, and A. Frank van der Stappen. Flexible muscle-based locomotion for bipedal creatures. *ACM Transactions on Graphics*, 32(6), 2013.

[13] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018.

[14] Mark Hendrikx, Sebastiaan Meijer, Joeri Velden, and Alexandru Iosup. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP)*, 9, 02 2013. doi: 10.1145/2422956.2422957.

[15] Daniel Holden, Oussama Kanoun, Maksym Perepichka, , and Tiberiu Popa. Learned motion matching. *ACM Trans. Graph*, 40(4), 2020. URL https://static-wordpress.akamaized.net/montreal.ubisoft.com/wp-content/uploads/2020/07/09154101/Learned_Motion_Matching.pdf. Last accessed: 08-06-2021.

[16] Seunghwan Lee, Moonseok Park, Kyoungmin Lee, and Jehee Lee. Scalable muscle-actuated human simulation and control. *ACM Trans. Graph.*, 38(4), 2019. URL http://mrl.snu.ac.kr/research/ProjectScalable/Page.htm.

[17] Rongrong Liu, Florent Nageotte, Philippe Zanne, Michel de Mathelin, and Birgitta Dresp-Langley. Deep reinforcement learning for the control of robotic manipulation: A focussed mini-review. *Robotics*, 10(1), 2021. ISSN 2218-6581. doi: 10.3390/robotics10010022. URL https://www.mdpi.com/2218-6581/10/1/22.

[18] Jeff Orkin. Three states and a plan: The a.i. of f.e.a.r. *Game developers conference*, 2006(6), 2006.

[19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[20] Xue Bin Peng, Glen Berseth, KangKang Yin, and Michiel van de Panne. Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning. *ACM Transactions on Graphics (Proc. SIGGRAPH 2017)*, 36(4), 2017.

[21] Xue Bin Peng, Pieter Abbeel, Sergey Levine, and Michiel van de Panne. Deepmimic: Example-guided deep reinforcement learning of physics-based character skills. *ACM Transactions on Graphics (Proc. SIGGRAPH 2018)*, 37(4), 2018.

[22] Hoseok Ryu, Seungwhan Lee Minseok Kim, Moon Seok park, Kyoungmin Lee, , and Jehee Lee. Functionality-driven musculature retargeting. *Computer Graphics Forum*, 40(1), 2021. URL http://mrl.snu.ac.kr/research/ProjectScalable/Page.htm.

[23] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

[24] Kun Shao, Zhentao Tang, Yuanheng Zhu, Nannan Li, and Dongbin Zhao. A survey of deep reinforcement learning in video games, 2019.

[25] Shubham Sharma, Shubhankar Verma, Mohit Kumar, and Lavanya Sharma. Use of motion capture in 3d animation: Motion capture systems, challenges, and recent trends. In *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*, pages 289–294, 2019. doi: 10.1109/COMITCon.2019.8862448.

[26] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, 2015.

[27] Unity Technologies. Unity ml-agents toolkit documentation. URL https://github.com/Unity-Technologies/ml-agents/tree/main/docs. Last accessed: 08-06-2021.

[28] Unity Technologies. Unity, 2021. URL https://unity.com/. Last accessed: 08-06-2021.

[29] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world, 2017.

[30] Lilian Weng. Policy gradient algorithms. 2018. URL https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html. Last accessed: 08-06-2021.

[31] Ryan Wong. Getting started with markov decision processes: Reinforcement learning, 2018. URL https://towardsdatascience.com/getting-started-with-markov-decision-processes-reinforcement-learning-ada7b4572ffb. Last accessed: 08-06-2021.