Universitat de Girona
Escola Politècnica Superior

Treball final de grau

Estudi: Grau en Disseny i Desenvolupament de Videojocs

Títol: Generació d'animacions de personatges per jocs de lluita mitjançant xarxes neuronals

Document: Memòria

Alumne: Joel Pérez Abad

Tutor: Gustavo Ariel Patow
Departament: Informàtica, matemàtica aplicada i estadística (IMAE)
Àrea: Llenguatges i sistemes informàtics

Convocatòria (mes/any) Setembre 2021

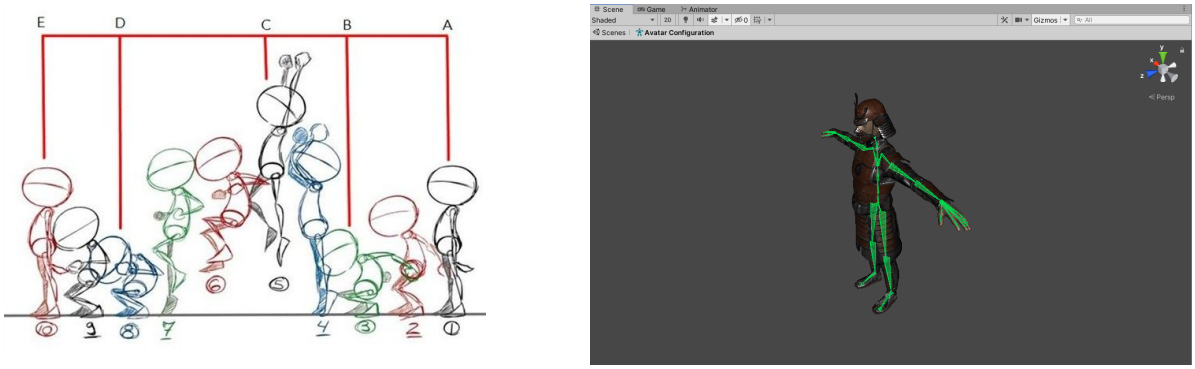
Continguts

1 Introducció

La indústria dels videojocs abasta molts àmbits i disciplines, algunes més tecnològiques i d'altres més artístiques. Dissenyadors de videojocs, programadors, guionistes, productors, enginyers d'àudio, són uns dels pocs perfils que treballen en qualsevol empresa de gran envergadura, sent capaços de crear històries inoblidables i entreteniment del més divertit.

Si indaguem en l'apartat artístic, un aspecte molt important i interessant és **l'animació de personatges**. Aquesta disciplina, encara que sembli estrany, no té l'origen en els videojocs. De fet, els considerats primers videojocs, *Bertie the Brain* i el famós *Pong*, no tenen cap mena de personatge animat. Els primers els trobem en el cinema, més concretament en las pel·lícules col·loquialment conegudes com "d'animació". No es fins als anys 80 on podem començar a trobar personatges com *Pac-Man* o *Donkey Kong* i *Jumpan* (més tard anomenat Mario) amb moviments simulant l'obrir i tancar d'una boca o el moviment dels braços quan es desplacen.

Sigui dibuixant fotogrames en 2D (vegeu Figura 1.1a) o fent moure un "esquelet" d'un objecte en 3D (vegeu Figura 1.1b), les tècniques més empleades requereixen un gran esforç si es vol aconseguir realisme, tant en termes de feina de l'artista (complexitat), com del temps requerit. Actualment algunes grans empreses utilitzen tècniques més avançades com és la captura de moviment o *Motion Capture*, però no està gens estandarditzada i la producció és cara (s'ha de contractar actors, tenir un espai adequat...).



(a) Exemple procés animació clàssica en 2D [1]

(b) Exemple procés d'animació clàssica en 3D

Figura 1.1: Exemples de mètodes de animació

Sortosament, hi ha una alternativa factible a l'animació clàssica: **l'animació procedural**, on són algorismes els que defineixen com s'ha de moure un personatge. En aquest cas es descriu el moviment de manera algorítmica mitjançant una sèrie de regles que permeten controlar com s'aniran modificant els diferents paràmetres de l'objecte o personatge al llarg del temps (parts com cames, potes, braços...). Si això ho ajuntem amb una IA o intel·ligència artificial, podem arribar a crear sistemes on, amb l'objectiu de complir certes tasques, la IA pot definir paràmetres inicials per a l'animació procedural, llevant fins i tot l'entrada de dades del propi programador.

En ser una forma d'animar enfocada en la programació i no tant en les capacitats artístiques del creador, he volgut aprendre a utilitzar-la per a poder crear animacions el més realistes possibles, ja que la inversió de temps és inferior a la manera considerada com a tradicional i així, com a dissenyador de videojocs, poder dedicar més temps als altres apartats que té un videojoc.

1.1 Motivacions

Les motivacions per a dur a terme un treball d'aquest tipus venen per **motius personals** i les **necessitats professionals**.

Pel que fa als motius personals, la motivació radica en la falta d'experiència en animació tradicional. Trobar altres mètodes que fossin enfocats al que se'm dona millor (programació) ha estat un dels meus principals objectius des que vaig realitzar les meves primeres animacions.

Pel que fa als motius professionals, aprendre diferents tècniques d'animar personatges m'ajudarà a obrir el ventall de possibilitats a l'hora d'enfrontar-me al repte de l'animació. Com més recursos tingui a la meva disposició, més fàcil se'm farà poder resoldre aquests reptes.

1.2 Propòsit

Com s'ha comentat, un cop vist que aprendre a realitzar de manera procedural animacions de personatges és de gran utilitat, el propòsit principal d'aquest treball és crear un escenari/prototip on hi hagi un personatge que estigui animat amb algun mètode procedural i crear una intel·ligència artificial que li proporcioni les dades necessàries. Per ser més concrets, la idea és crear un escenari on un personatge ataqui a un altre amb una espasa i el defensor haurà de defensar-se movent la seva espasa de forma procedural.

Per dur-ho a terme s'ha considerat dos elements: **Cinemàtica inversa** i **xarxes neuronals**.

Per saber on mourà l'espasa defensora, es farà servir una intel·ligència artificial formada per un model de xarxa neuronal, la qual rebrà dades de l'atacant i algunes dades del defensor, les processarà i donarà la posició on hauria d'anar l'espasa per defensar-se de l'atac. Llavors, fent servir cinemàtica inversa (*Inverse Kinematics*), mourem els braços del defensor tot creant una animació procedural de moviment de braços.

1.3 Objectius

El principal objectiu del treball és crear una intel·ligència artificial basada en una xarxa neuronal capaç de fer moure una espasa d'un personatge que es defensa d'un atacant, tot creant una animació de defensa fent servir cinemàtica inversa.

Per dur-ho a terme, es realitzaran les següents tasques:

- Estudiar i analitzar el funcionament d'una xarxa neuronal, en concret, les xarxes que proporciona l'eina ML-Agents.
- Crear entorns per fer *reinforcement learning* per a la xarxa neuronal. És començarà des d'entorns i simulacions més senzilles fins a més complexes. Per exemplificar-ho, es començarà amb simples moviments en un únic eix, fins a moviments més avançats, adequats pel seu ús en videojocs.
- Entrenament en aquests entorns i comprovació de les dades extretes (fent les modificacions pertinents, mirant si l'entrenament funciona).
- Lligar els resultats de l'animació procedural amb cinemàtica inversa.
- Perfeccionament de la xarxa i anàlisis dels resultats.
- Arrodoniment i finalització de la documentació del treball.

També recalcar que, en saber prèviament la complexitat de la cinemàtica inversa i les xarxes neuronals, s'espera com a resultat una animació simple, és a dir, en cap moment es busca crear una animació complexa i espectacular capaç d'igualar el treball d'un animador professional.

1.4 Estructura del document

Aquest document s'ha organitzat en **14 apartats**, que són els següents:

1. **Introducció, motivacions, propòsit i objectius del projecte:** Introducció al treball on s'expliquen les raons i motivacions per les quals s'ha triat aquesta tipologia de treball, els objectius i propòsits que es vol assolir en finalitzar i com s'organitza la memòria.
2. **Estudi de viabilitat:** Explicació dels paràmetres que farien possible el desenvolupament del projecte juntament amb els recursos necessaris.
3. **Metodologia:** Esment de la metodologia de desenvolupament que se seguirà, la seva estructura i una justificació de per què de la selecció de la metodologia seguida.
4. **Planificació:** Definició de l'estratègia seguida per arribar als objectius plantejats. Es descriu breument el pla de treball, les tasques planificades, el temps estimat i final i els resultats esperats de cada tasca.
5. **Marc de treball i conceptes previs:** Descripció de conceptes previs necessaris pel seguiment i enteniment global del treball.
6. **Requisits del sistema:** Descripció dels requisits funcionals i no funcionals que han de complir el sistema i les funcionalitats que es volen obtenir. També s'especifica les característiques de l'equip on s'han realitzat les proves.
7. **Estudis i decisions:** Descripció del maquinari, les llibreries i el programari utilitzats durant el desenvolupament del projecte i el perquè de la seva utilització.
8. **Anàlisi i disseny del sistema:** Estudi i anàlisi de les necessitats del projecte, on s'explicaran els requisits, i es farà una explicació dels elements més importants sense mostrar la implementació.
9. **Implementació i proves:** Explicació de com s'ha implementat el prototip amb els detalls més importants, i com s'han anant solucionant els problemes sorgits.
10. **Resultats:** Descripció amb detall del procés de desenvolupament que s'ha calgut dur a terme per implantar el sistema desenvolupat.
11. **Conclusions:** Exposició del grau d'assoliment comparant amb l'objectiu plantejat, així com una crítica dels resultats obtinguts.
12. **Treball futur:** Esment de possibles ampliacions, millores o treballs futurs que es poden realitzar.
13. **Bibliografia:** Llistat de les referències utilitzades per desenvolupar el projecte.
14. **Manual d'usuari i d'instal·lació:** Explicació de com s'ha d'utilitzar i instal·lar l'aplicació en un ordinador.

2 Estudi de viabilitat

Al dur a terme aquest treball s'ha volgut plantejar de tal manera que les despeses siguin mínimes. Per aquesta raó, tant el *software* com el *hardware* utilitzat, o bé són gratuïts o ja es disposaven, i per tant, no afegeixen cap despesa extra. També es farà servir recursos tot respectant els drets d'autor (Llei de Propietat Intel·lectual), és a dir, només s'utilitzarà material d'ús lliure.

2.1 Recursos tècnics

S'ha fet servir el següent *hardware* per a la realització del treball:

- Ordinador portàtil **MSI GP75 Leopard 9SF:**
  - Targeta gràfica (*Graphics Processor Unit* o GPU): NVIDIA GeForce RTX 2070
  - Processador (*Central Processor Unit* o CPU): Intel Core i7-9750H 2.60GHz
  - Memòria (*Random Acces Memory* o RAM): 16GB DDR IV
- Ordinador de sobretaula:
  - Targeta gràfica (*Graphics Processor Unit* o GPU): NVIDIA GeForce GTX 1080
  - Processador (*Central Processor Unit* o CPU): Intel Core i7-8700K 3.7GHz
  - Memòria (*Random Acces Memory* o RAM): 8GB DDR III

L'ordinador portàtil s'ha fet servir principalment per a realitzar l'entrenament de la xarxa neuronal, mentre que amb l'ordinador de sobretaula s'ha realitzat la creació de l'animació procedural. Aquestes dues tasques s'han realitzat en paral·lel, és a dir, mentre s'entrenava la xarxa en el terminal esmentat, en l'altra es programava el funcionament de les animacions.

2.2 Recursos humans

Al ser un treball tecnològic, es necessitessen diversos perfils per a la seva realització, els quals es descriuen a continuació:

1. **Animador digital:** Encarregat de programar i tractar l'apartat de l'animació procedural. Haurà de relacionar les dades que proporciona la xarxa neuronal i crear, amb el mètode de cinemàtica inversa, les animacions del personatge que es defensa.
2. **Programador:** Encarregat de programar i dissenyar l'entorn d'entrenament de la xarxa neuronal i anar polint els resultats per a millorar les dades que es proporcionen a l'animador.
3. **Cap de projecte:** Encarregat de coordinar i supervisar que el treball es realitzi en la metodologia indicada (explicada en l'Apartat 3: Metodologia)

A causa que el projecte ha estat realitzat per una mateixa persona, els dos primers perfils han estat desenvolupats per la meva persona. En canvi, pel que fa al perfil de cap de projecte, l'ha dut a terme el tutor.

2.3 Requisits tecnològics

El *software* utilitzat pel recursos tècnics han estat els següents:

- **Sistema operatiu (ambdós ordinadors):** Windows 10 Home
- **Motor de videojocs:** Unity versió 2020.1.13f1 amb IDE Visual Studio 2020
- **Eina xarxes neuronals:** ML-Agents release 12 (plug-in per a Unity)

La seva utilització i explicació bàsica està explicat en l'Apartat 7: Estudis i decisions.

2.4 Cost econòmic

Com s'ha esmentat prèviament, el cost extra del treball és nul, però si haguéssim de fer un pla de costos (estimació) quedaria de la següent manera:

1. **Costos de maquinaria i *software***
    - Recursos tècnics = 1553 € (MSI GP75 Leopard 9SF) + 1020 € (Ordinador sobretaula) = **2573 €**
    - Requisits tecnològics = **105 €** (Llicencia Windows 10 Home)
    - **Total: 2678 €**
2. **Costos humans**
    - Sous perfils (estimacions):

      Cap de projecte = 26 €/h | Animador digital: 15 €/h | Programador: 20 €/h

      Taula de costos (Figura 2.1):

Tasca	Perfil	Hores	Cost
Planificació projecte	Cap de projecte	125	3250 € (26 €/h)
Estudi motor Unity (1)	Programador	32	640 € (20 €/h)
Estudi motor Unity (2)	Animador digital	40	600 € (15 €/h)
Estudi Inverse Kinematics	Animador digital	22	330 € (15 €/h)
Estudi intel·ligència artificial	Programador	12	240 € (20 €/h)
Estudi xarxes neuronals	Programador	15	300 € (20 €/h)
Estudi ML-Agents	Programador	37	740 € (20 €/h)
Implementació Agent	Programador	18	360 € (20 €/h)
Implementació xarxa neuronal	Programador	132	2640 € (20 €/h)
Entrenament xarxa neuronal	Programador	211	4220 € (20 €/h)
Implementació Inverse Kinematics	Animador digital	35	525 € (15 €/h)
Documentació	Cap de projecte	213	5538 (26 €/h)
			TOTAL COST: 19.383 €

Figura 2.1: Taula de costos

3. **Cost final**

   Els costos finals estimats (sumatori de costos) són **22.061 €**

2.5 Conclusions viabilitat

Primer de tot, comentar que la resta de programari que no s'ha esmentat en l'anterior apartat és perquè és d'ús gratuït.

Pel que fa als costos humans, en el cas hipotètic d'una contractació real en un projecte professional, no és del tot realista envers amb el que s'ha pogut observar a la taula anterior. Moltes hores han estat invertides en provar noves tecnologies i mètodes dels quals es desconeixia el funcionament ni és tenia un coneixement previ. Per aquesta raó, aquestes hores es veurien reduïdes en gran manera.

Com a conclusió, obviant els costos addicionals pel funcionament dels recursos tècnics, es pot dir que el projecte és del tot factible, ja que es disposa de tot recurs necessari pel correcte desenvolupament.

3 Metodologia

Per a la realització d'aquest treball, s'ha analitzat les diferents metodologies de desenvolupament *software* i s'ha arribat a la conclusió que cap arriba a encaixar al 100% amb els objectius i procés en ment. Per aquesta raó, s'ha decidit juntament amb el tutor del projecte utilitzar una metodologia personalitzada lleugerament inspirada en la *Waterfall*. L'estructura és la següent (Figura 3.1):

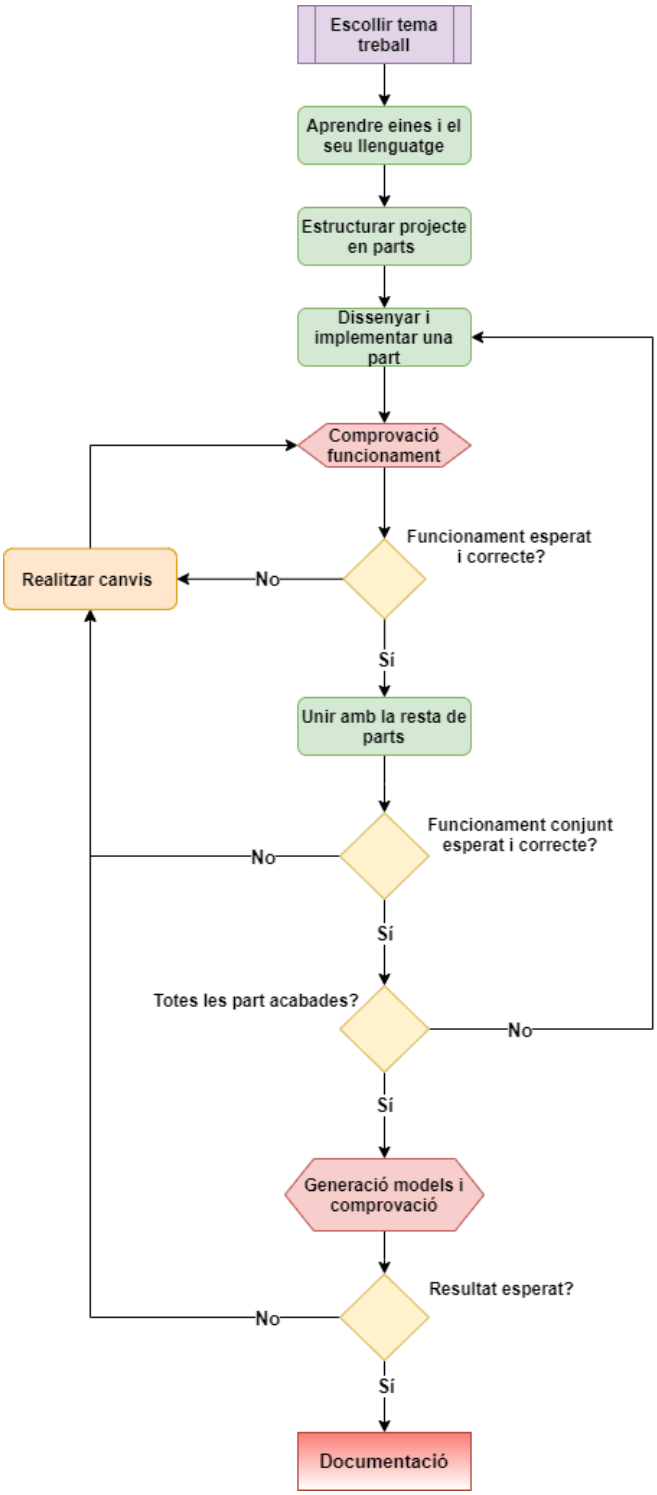


Figura 3.1: Diagrama metodologia personalitzada a seguir

Consisteix a dividir el projecte en seccions o mòduls i organitzar en el temps el desenvolupament, en temps d'implementació i de correcció. Es diu que està inspirat en la metodologia *Waterfall* perquè només passem a la següent secció quan tenim la certesa que l'anterior funciona correctament. Això es fa així per cerciorar-nos que els errors que apareguin són de la nova secció.

Descripció de la metodologia per passos:

0. Escollir tema del treball a realitzar.
1. Aprendre a fer funcionar les eines necessàries i a programar en el seu llenguatge.
2. Estructurar el projecte en diferents seccions per a poder treballar concurrentment.
3. Dissenyar i implementar una secció seguint l'ordre establert en l'anterior pas.
4. Comprovar el correcte funcionament de la nova secció desenvolupada.
   (a) Si no funciona com s'esperava, es realitzen els canvis que es considerin i es torna a comprovar.
   (b) Si el seu funcionament és l'esperat, mirarem d'unir-ho amb les altres seccions.
5. Unir la nova secció amb la resta.
   (a) Si tot funciona correctament, mirarem si encara queden seccions per desenvolupar.
      i. Si no queda cap secció a desenvolupar, passarem a la generació de models.
      ii. Si encara queden seccions, tornarem al pas del disseny i implementació d'una part.
   (b) Si alguna cosa no acaba d'encaixar, mirem de nou què falla i realitzarem els canvis necessaris, tornant així a comprovar per separat el funcionament de la secció.
6. Generació de diferents models i comprovació del correcte funcionament.
   (a) Si tot funciona com s'espera, passem al pas de la documentació.
   (b) Si hi ha algun error o apartat que no funcioni del tot bé, tornarem a fer canvis i a comprovar que aquests no influeixin en la seva secció individual ni al conjunt. També, com a resultar esperat, parlem de si l'escenari és el que volem (personatge 3D amb una espasa que es defensa d'un altre personatge 3D amb una espasa), ja que la idea del projecte és començar a treballar en conceptes més senzills (simples cilindres i cubs) fins a tenir el que es vol com a objectiu final.
7. Finalització d'una memòria sobre el projecte, la qual és començada a l'inici.

4 Planificació

4.1 Inicis i problemes

L'inici del projecte data del mes de **setembre de 2020**, quan es van realitzar les primeres reunions amb el tutor. En elles, es discutia els temes que podria tractar i des de quin punt de vista. Finalment, a finals d'aquell mes es va tenir clar la visió general de la temàtica a tractar. A partir d'octubre de 2020, vaig començar a buscar informació relacionada amb l'animació procedural i les xarxes neuronals.

Primerament, la recerca d'informació va anar enfocada en el programa informàtic ***Blender***, una eina pel tractament d'objectes 3D (modelatge, il·luminació, renderitzat, animació i creació de gràfics tridimensionals), i en la biblioteca de xarxes neuronal ***Keras***. Aquesta recerca va ser molt important perquè va assentar les bases del que seria el projecte. Tot i això, dos mesos després es va trobar l'eina ***ML-Agents***, la qual vam considerar millor i més efectiva per dur a terme la idea general. Per tant, considerem que l'inici del treball tal com es veurà implementat (motor Unity + ML-Agents) comença el **desembre de 2020**.

Darrerament, tot i que estava pensat entregar el treball a la **primera convocatòria (juny 2021)**, la realització d'entrenaments de la intel·ligència artificial i la redacció d'aquesta memòria es van allargar més del que es va pensar originalment, provocant així que l'entrega es realitzés a la **segona convocatòria (setembre 2021)**.

4.2 Planificació inicial

Un cop es va tenir la idea final, es va crear un primer pla de treball amb les diferents tasques a realitzar.

Tasques a realitzar:

0. Aprendre Blender i Keras (Estudi animacions procedurals i xarxes neuronals).
1. Aprendre a utilitzar ML-Agents.
2. Aprendre a implementar animacions procedurals al motor de Unity.
3. Dissenyar els diferents escenaris pels agents de la intel·ligència artificial de ML-Agents.
4. Utilitzar en aquests escenaris l'aprenentatge de ML-Agents.
5. Seleccionar el millor escenari (el més avançat) i lligar l'agent amb les animacions procedurals.
6. Realització de la documentació (memòria) del treball realitzat al llarg del desenvolupament.

Es va marcar com a objectiu entregar el treball en la primera convocatòria, al juny de 2021.

En el diagrama de Gantt de la Figura 4.1 es pot veure el pla inicial:

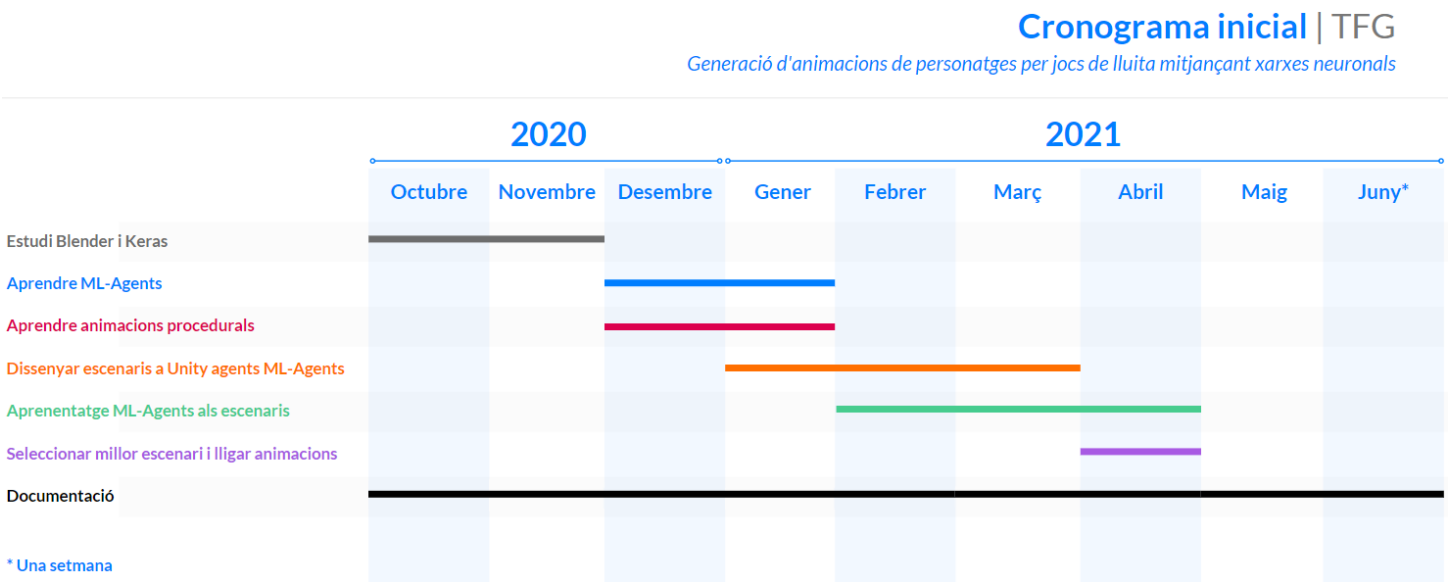


Figura 4.1: Diagrama de Gantt previst

4.3 Planificació final

Com en tot projecte, el que es té planejat no sempre s'arriba a complir. Com ha estat la planificació final es pot observar en el diagrama de la Figura 4.2:

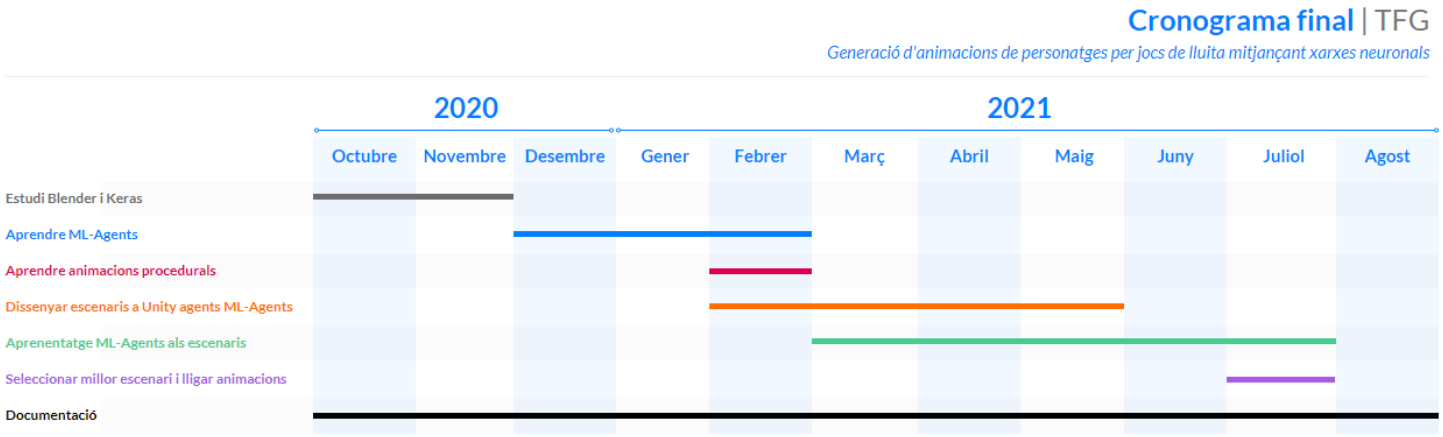


Figura 4.2: Diagrama de Gantt final

4.4 Problemes amb les tasques

El principal problema que hi ha hagut ha estat la durada de les tasques. En totes, el temps estimat a estat inferior al temps finalment invertit. Això es a causa de la complexitat del treball i en diversos problemes que han succeït que es descriuen a continuació:

- **Aprenentatge ML-Agents:** Pel que fa a l'aprenentatge de ML-Agents, s'ha hagut de tornar a la documentació sovint a repassar el seu funcionament, ja que els primers dissenys dels escenaris no funcionaven del tot bé. Per aquesta raó, aquesta tasca s'ha **allargat en un mes** d'anar repassant conceptes.
- **Aprenentatge animacions procedurals:** Pel que fa a les animacions procedurals, es va començar més tard a estudiar, però el temps dedicat a l'estudi principal en cinemàtica inversa es va **reduir un mes**, ja que la complexitat i implementació no era tan elevada com s'estimava.
- **Disseny escenaris a Unity agents ML-Agents:** El disseny de diferents escenaris es va **allargar dos mesos** perquè els més senzills es van tardar més a crear, dilatant així la creació de l'escenari final amb un agent complex, tal com es volia en l'objectiu del projecte (personatge 3D amb una espasa que es defensa d'un altre personatge 3D amb una espasa).
- **Aprenentatge ML-Agents als escenaris:** En dilatar-se el temps en tenir l'escenari final, el temps en fer aprendre a la xarxa neuronal va **augmentar en dos mesos**, en el qual es va treballar exclusivament en l'aprenentatge de l'escenari final.
- **Seleccionar millor escenari i lligar animacions:** En tenir més tard l'agent ja en funcionament, el lligam amb la cinemàtica inversa es va desplaçar en el temps.

5 Marc de treball i conceptes previs

5.1 Motors de videojocs

La creació de videojocs és definida per entorns de desenvolupament adaptats a les seves necessitats, anomenats **motors de videojocs** o ***game engines***. Els desenvolupadors els utilitzen per a construir videojocs per a consoles i altres tipus de sistemes, com ordinadors o telèfons mòbils. Les funcionalitats bàsiques que solen oferir inclouen un motor de renderitzat o *renderer* per a gràfics 2D o 3D, un motor de física o detecció de col·lisions (i resposta a les col·lisions), inclusió de so, habilitació de programació, animació, intel·ligència artificial, xarxes, transmissió de dades, gestió de memòria, *threading*, suport de localització (traducció), i suport de vídeo per a cinemàtiques.

En alguns casos se'ls denominen com a "agents intermedi" o ***middleware*** perquè, igual que en el sentit comercial del terme, ofereixen una plataforma de programari flexible i reutilitzable que proporciona tota la funcionalitat bàsica necessària, res més treure-la de la "caixa", per a desenvolupar una aplicació de joc, reduint al mateix temps els costos, la complexitat i el temps de comercialització, factors tots ells crítics en l'altament competitiva indústria dels videojocs. Igual que altres tipus de *middleware*, els motors de videojocs solen oferir **abstracció de plataformes**, és a dir, permet que el mateix joc s'executi en diverses plataformes (incloses les consoles de jocs i els ordinadors personals) amb pocs canvis, o cap, en el codi font del videojoc.

5.2 Animació procedural: Inverse Kinematics

Una figura humanoide animada en 3D es modela amb un esquelet de segments rígids connectats amb articulacions, anomenat **"cadena cinemàtica"**. Les equacions cinemàtiques de la figura humanoide defineixen la relació entre els angles de les seves articulacions i la postura o configuració.

El mètode de **cinemàtica inversa**, o ***Inverse Kinematics***, és un procés matemàtic que consisteix a calcular els paràmetres variables de les articulacions necessaris per a situar el final d'una cadena cinemàtica en una posició i orientació determinades en relació amb l'inici de la cadena. Per a il·lustrar i exemplificar, seria calcular la posició i rotació del genoll i la cintura si aixequem el peu del terra, tal com es mostra en la Figura 5.1.

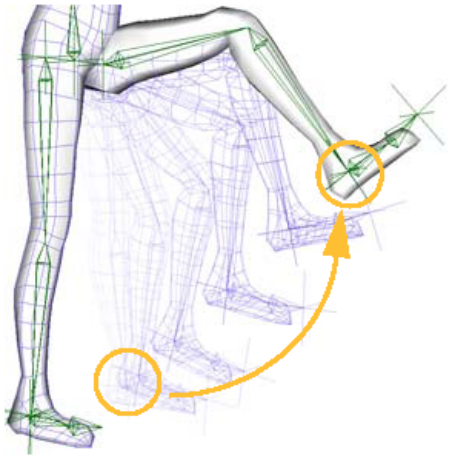


Figura 5.1: Representació del moviment calculat mitjançant cinemàtica inversa [2]

La cinemàtica inversa és important per a la programació de jocs i l'animació en 3D, on s'utilitza per a connectar físicament als personatges en el món. Sovint és més fàcil per als dissenyadors, artistes i animadors definir la configuració espacial d'un conjunt o figura movent peces, o braços i cames, en lloc de manipular directament els angles de les articulacions. Per això, la cinemàtica inversa és més utilitzada per a posicionar figures i personatges.

5.3 Machine learning

Machine learning, o aprenentatge automàtic, és una branca de la intel·ligència artificial destinada a crear sistemes dedicats a la **identificació i l'aprenentatge de patrons** determinats per una quantitat de paràmetres.

La màquina no aprèn per si mateixa, sinó un algorisme de la seva programació, que es modifica amb la constant entrada de dades en la interfície, i que pot, d'aquesta manera, predir escenaris futurs o prendre accions de manera automàtica segons unes certes condicions. A mesura que l'algorisme ingereix dades d'entrenament, és possible produir models més precisos basats en aquestes dades. Un model d'aprenentatge automàtic és la sortida d'informació que es genera quan s'entrena l'algorisme amb dades. Després de l'entrenament, en proporcionar un model amb una entrada, se li donarà una sortida.

En la Figura 5.2, es pot observar de manera senzilla i esquemàtica el funcionament bàsic d'una màquina d'aprenentatge automàtic:

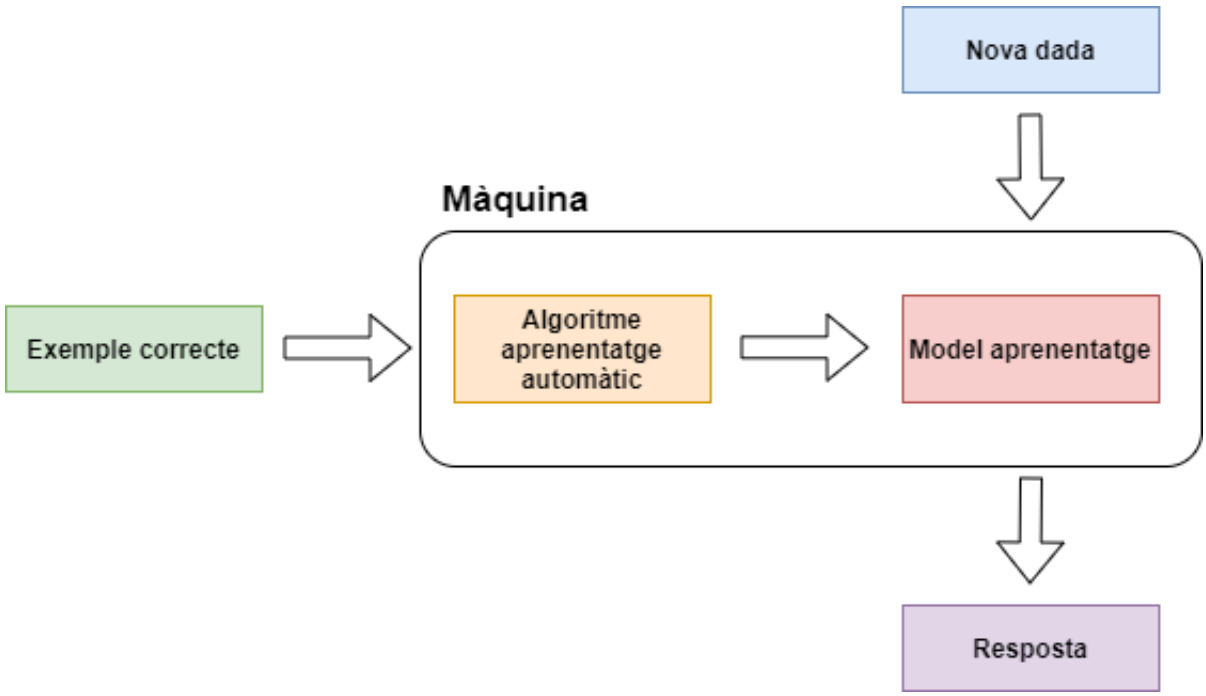


Figura 5.2: Diagrama aprenentatge automàtic

5.3.1 Algorismes de reinforcement learning

Els algorismes d'aprenentatge per reforç o *reinforcement learning* són un dels tipus d'algorismes de l'aprenentatge automàtic. Tenen en compte dades que reben del que anomenem com a **"entorn"** i la tasca de tractar les dades les duu a terme l'anomenat **"agent"**. Aquest tractament s'anomena "acció", que influeix directament en l'entorn tot fent canviar l'estat d'aquest.

Agent

L'agent, per aprendre, ha de tenir alguna forma per poder saber com està influint les seves accions en l'estat de l'entorn, ja que no hi ha cap instructor que el guiï. Aquest element s'anomena **"recompensa"** o **"reforç"**, i sol ser un valor numèric intrínsec a l'algorisme que indica que tan bé o malament ha estat la resposta al tractament de les dades. L'objectiu de l'agent és aprendre una **"política"** o ***"policy"*** (llista de probabilitats de realitzar una acció quan s'està en un estat) òptima, o gairebé òptima, que maximitzi la **"funció de recompensa"**, és a dir, l'obtenció de la millor recompensa final.

Entorn

L'entorn pot ser **determinista**, és a dir, a grans trets, la mateixa acció en el mateix estat condueix al mateix estat següent, o **no determinista** (o **estocàstic**), és a dir, si l'agent realitza una acció en un determinat estat, el següent estat resultant de l'entorn no té per què ser sempre el mateix: existeix una probabilitat que sigui un determinat estat o un altre, fent que sigui més difícil la tasca de trobar la política òptima.

En la Figura 5.3 es mostra la interacció entre agent i entorn d'un sistema d'aprenentatge automàtic.

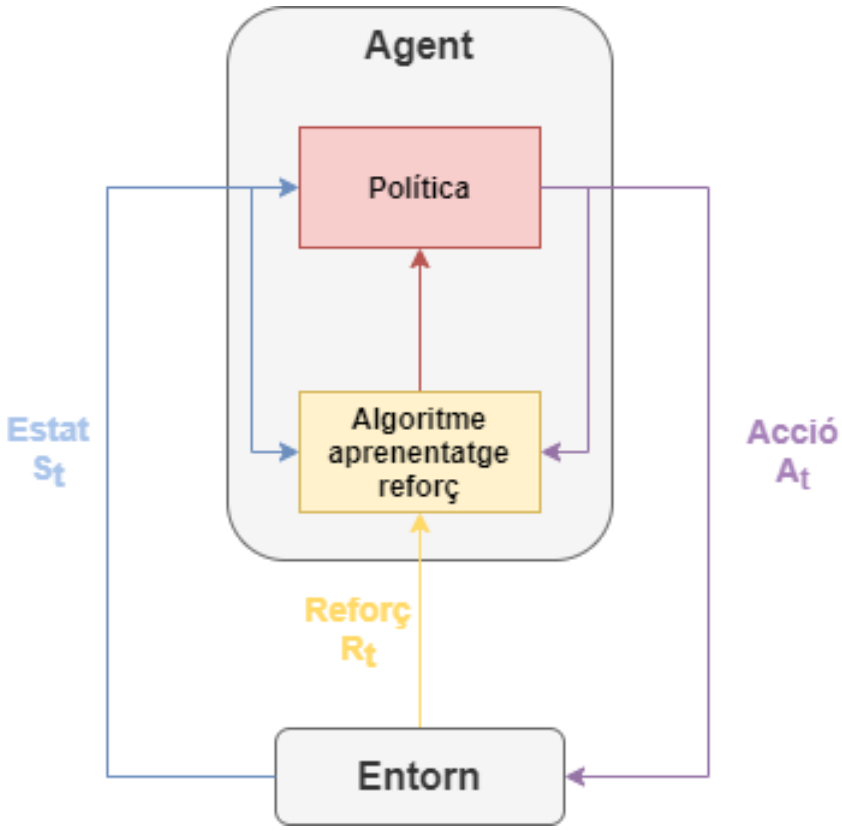


Figura 5.3: Diagrama simplificat algorismes aprenentatge per reforç

Primera distinció: Model-Based RL o Model-Free RL

En l'aprenentatge per reforç, el problema sol formular-se matemàticament com un **procés de decisió de Markov** (*Markov Decision Process* o MDP). Un MDP és una manera de representar la "dinàmica" de l'entorn, és a dir, la forma en què l'entorn reaccionarà a les possibles accions que l'agent pugui realitzar en un estat determinat. Més concretament, un MDP està dotat d'una **funció de transició** (o model de transició), que és una funció que, donat l'estat actual de l'entorn i una acció, dóna com a resultat una probabilitat de passar a qualsevol dels següents estats. Una funció de recompensa (explicada anteriorment) també està associada a un MDP. Intuïtivament, la funció de recompensa produeix una recompensa, donat l'estat actual de l'entorn (i, possiblement, una acció realitzada per l'agent i el següent estat de l'entorn). En conjunt, les funcions de transició i recompensa solen denominar-se **model d'entorn**, i quan un algoritme en té, se li anomena ***Model-Based RL*** o **"aprenentatge de reforç basat en un model"**.

No obstant això, sovint no tenim el MDP, és a dir, no tenim les funcions de transició i recompensa. Per tant, no podem estimar una política a partir del MDP, perquè és desconegut. En absència d'aquestes funcions, és a dir, quan el MDP és desconegut, per a estimar la política òptima, l'agent necessita interactuar amb l'entorn i observar les respostes. Haurà d'estimar una política reforçant les seves creences sobre la dinàmica de l'entorn (enfocament de "prova i error"). Amb el temps, començarà a entendre com respon l'entorn a les seves accions, i així podrà estimar la política òptima. Aquests algorismes amb absència del MDP se'ls hi anomena ***Model-Free RL*** o **"aprenentatge de reforç lliure de model"**.

Model-Free RL: On-Policy learning o Off-Policy learning

En curt, els algorismes d'optimització de la política o *Policy Optimization* són algorismes que avaluen i milloren la mateixa política que s'està utilitzant per a seleccionar les accions. Cada actualització només utilitza les dades recollides mentre s'actua segons la versió més recent de la política, és a dir, estima la rendibilitat dels parells estat-acció suposant que se segueixi la política actual. A part de *Policy Optimization* també se'ls coneix com a *On-Policy learning*.

En canvi, els algorismes de *Q-Learning* o *Off-Policy learning* avaluen i milloren una política diferent a la política que s'utilitza per a la selecció d'accions. Cada actualització pot utilitzar les dades recollides en qualsevol moment durant l'entrenament, independentment de com l'agent estava triant explorar l'entorn quan es van obtenir les dades.

Per posar un exemple, si s'està en l'estat $\mathbf{S_t}$ i es realitza una acció $\mathbf{A_t}$ que ens porta a l'estat $\mathbf{S'_t}$, s'actualitzarà el valor del parell estat-acció basant-se en la millor acció possible que es pugui realitzar en $\mathbf{S'_t}$, o basant-se en una acció segons la seva política actual? El primer mètode d'elecció es *off-policy* i el segon, *on-policy*.

L'esquema de la Figura 5.4 mostra un desglossament de diferents algoritmes segons el tipus.

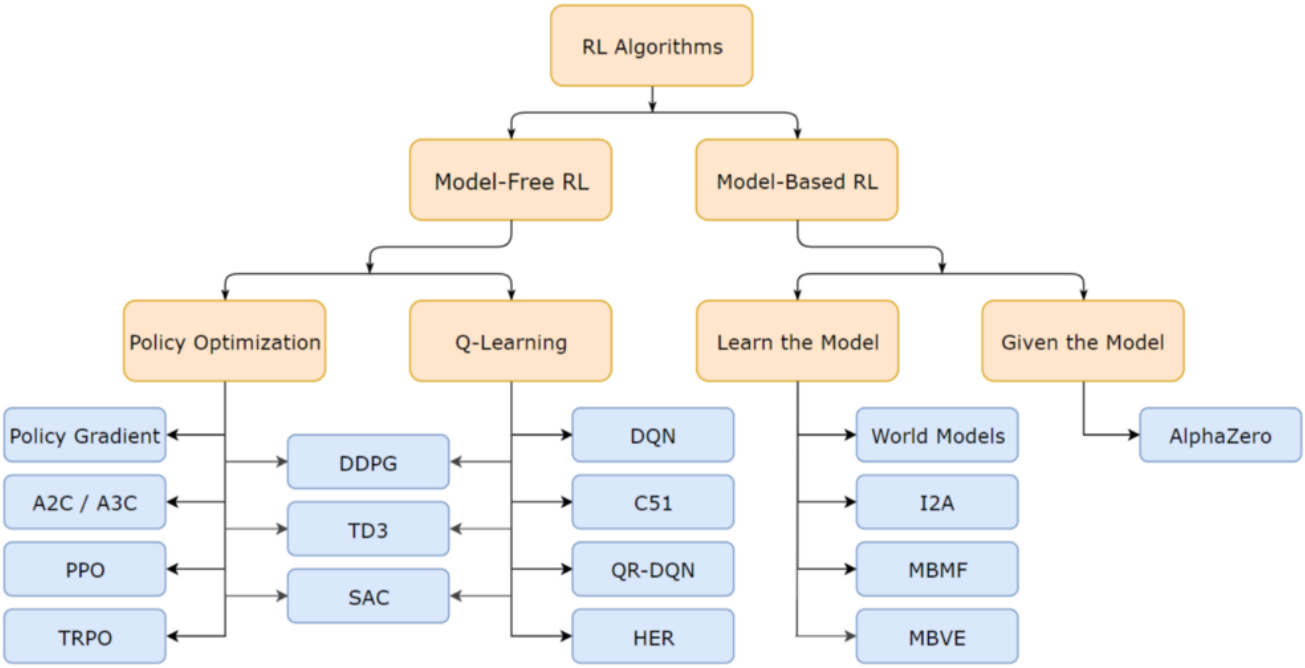


Figura 5.4: Classificació algorismes aprenentatge de reforç [3]

Millorar la política i entropia

Quan l'agent està aprenent una política i una acció retorna una recompensa positiva per a un estat, pot ocórrer que l'agent utilitzi sempre aquesta acció en el futur perquè sap que li ha produït alguna recompensa positiva. Podria existir una altra acció que produeixi una recompensa molt major, però l'agent mai la provarà perquè es limitarà a explotar el que ja ha après. Això significa que l'agent pot quedar-se embussat en un **òptim local** (o màxim local) per no explorar el comportament d'altres accions i no trobar mai **l'òptim global** (o màxim global).

Per evitar-ho, els algorismes fan servir l'entropia de la política. L'entropia, en l'aprenentatge per reforç, es refereix a la previsibilitat de les accions d'un agent, és a dir, la certesa que té la política sobre quina acció produirà la recompensa més gran acumulada a llarg termini. Com més gran sigui l'entropia, més aleatòries seran les accions d'un agent, ja que al no tenir clar quines accions són les millors, la probabilitat d'escollir diverses accions en comptes de només una augmenta.

El procés d'aprenentatge conduirà naturalment a la disminució de l'entropia de la política. Això és raonable, ja que si esperem un comportament intencionat i coordinat, llavors aquest comportament serà naturalment menys aleatori que la política original.

5.4 Xarxes neuronals

Dintre de l'aprenentatge automàtic hi ha un camp molt interessant anomenat ***deep learning*** o **"aprenentatge profund"**. Són un grup d'algorismes que utilitzen diverses capes d'aprenentatge per a extreure progressivament característiques de major nivell a partir de l'entrada de dades brutes. Aquestes capes fan referència a l'estructura més usada en aquests algorismes, les **xarxes neuronals.**

Els algorismes de xarxes neuronals per a l'aprenentatge automàtic s'inspiren en l'arquitectura i la dinàmica de les xarxes de neurones del cervell humà, intentant replicar el seu funcionament. Òbviament, hi ha certes limitacions. No obstant això, el principi fonamental és el mateix: les xarxes neuronals artificials aprenen canviant les connexions entre les neurones.

5.4.1 Funcionament neurona

La unitat base d'una xarxa neuronal és la **neurona**. Consisteixen en **entrades** (com a sinapsis), que es multipliquen per **pesos**. Aquests resultats, es sumen juntament amb un **biaix**. Finalment, a través d'una funció no lineal coneguda com a ***Transfer function*, funció d'activació o funció de transferència,** es calcula la sortida de la neurona artificial (a vegades en funció d'un cert llindar). Aquesta funció s'utilitza per convertir una entrada amb valors impredictibles en una sortida predictible. Podem observar en la Figura 5.5 un diagrama que mostra els passos que segueix la informació que rep la neurona fins que dona un dada resultant.

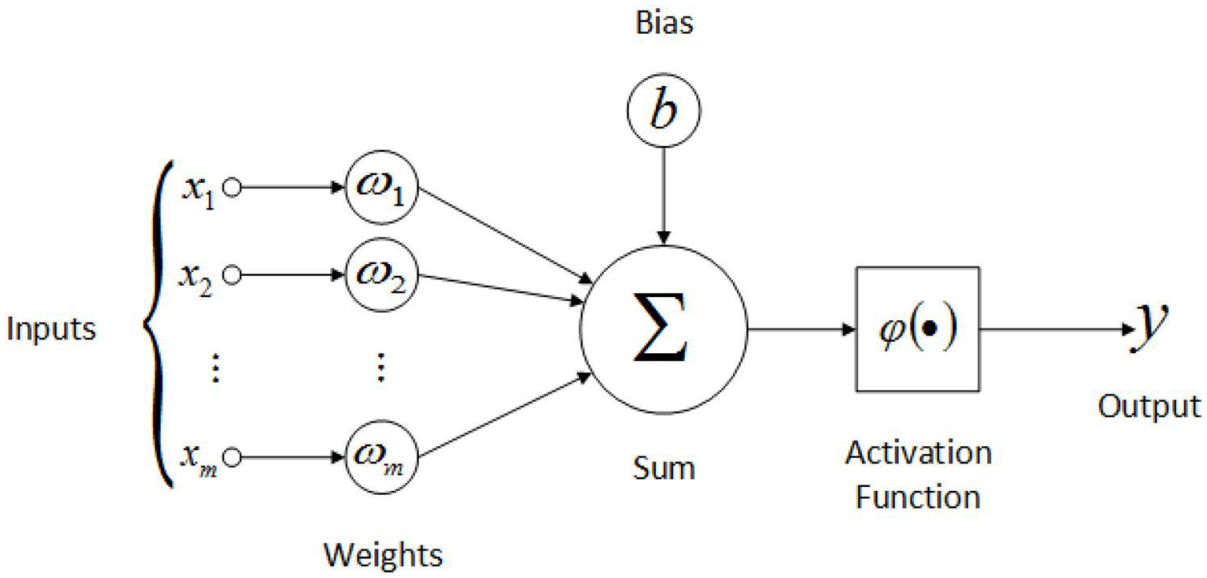


Figura 5.5: Diagrama del funcionament d'una neurona artificial [4]

Els pesos assignats a cada entrada representen el flux d'informació. Ajustant-los podem obtenir la sortida que desitgem per a entrades específiques. Però quan tenim centenars o milers de neurones, es bastant complicat calcular a mà tots els pesos necessaris. Per això hi ha algorismes que permeten ajustar els pesos per a obtenir la sortida desitjada de la xarxa. Aquest procés d'ajust dels pesos es diu **aprenentatge** o **entrenament**. L'entrenament comença amb pesos aleatoris, i l'objectiu és ajustar-los per a rebre el resultat buscat.

5.4.2 Combinació de neurones

L'agrupació i connexió entre diferents neurones es coneix com a **xarxa neuronal**. Aquesta està dividida en **capes**, agrupacions de neurones connectades amb les neurones de les capes immediatament anterior i posterior. Hi ha tres tipus de capes: **la capa d'entrada** (rep les dades externes), **la capa de sortida** (produeix el resultat final), i **les capes ocultes** (entre la d'entrada i la de sortida, podent haver-hi cap o infinites). Els patrons de connexió i maneres d'agrupar les capes defineixen els diferents tipus de xarxes.

En la Figura 5.6, podem observar un exemple de xarxa, formada per **n** dades d'entrada (*inputs*) rebuts per la capa d'entrada, una capa oculta formada per dos capes de **m** i **b** neurones cada una, i una capa de sortida formada per **v** neurones i, per tant, **v** dades de sortida (*outputs*).

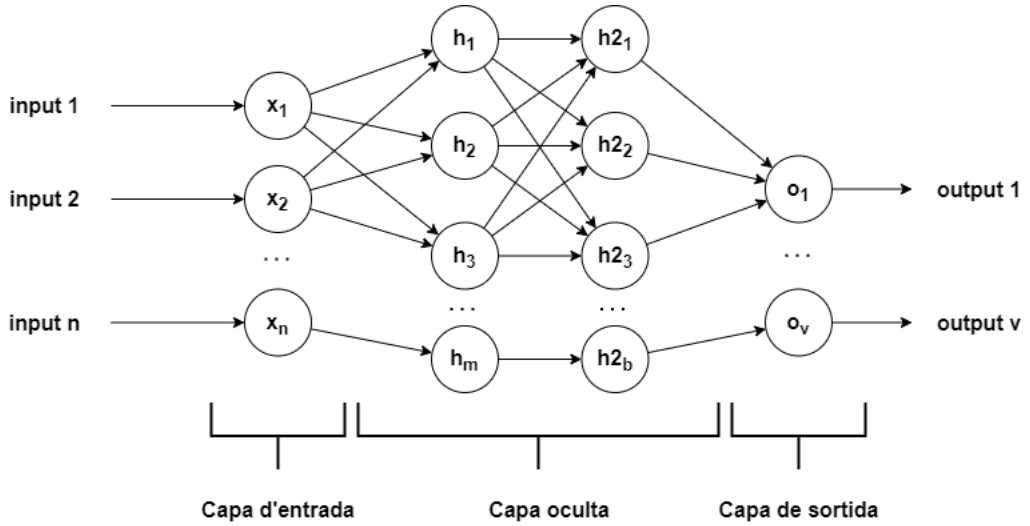


Figura 5.6: Exemple estructura de xarxa neuronal

Les variables que determinen l'estructura d'aquesta xarxa i les variables que determinen com s'entrena la xarxa (per exemple, la taxa d'aprenentatge) s'anomenen **hiperparàmetres**.

6 Requisits del sistema

Per a realitzar els objectius proposats en aquest projecte es necessitarà que el sistema que es crearà compleixi una sèrie de requisits.

6.1 Requeriments funcionals

Funcionalitats del sistema que es crearà per a complir amb els objectius:

1. L'agent aprendrà a moure un objecte (espasa) per a defensar un personatge 3D de un altre que mourà una espasa cap a ell.
2. El personatge defensor, controlat per l'agent d'intel·ligència artificial, mourà els braços tot seguint la espasa per simular que l'està movent ell.
3. L'usuari final podrà accedir a un prototip on podrà executar diferents animacions d'atac per observar els moviments de defensa de l'altre personatge.

6.2 Requeriments no funcionals

Com ja s'ha explicat amb anterioritat, per a la realització del treball s'ha treballat amb el motor de videojocs Unity i amb el plug-in ML-Agents. Per tant, els requeriments no funcionals que es requereixin estan lligats a ells.

6.2.1 Requisits hardware

Per a desenvolupar el projecte es necessita complir amb els requisits mínims que demana el programa de Unity (Figura 6.1).

Minimum requirements	Windows	macOS	Linux (Support in Preview)
Operating system version	Windows 7 (SP1+) and Windows 10, 64-bit versions only.	High Sierra 10.13+	Ubuntu 20.04, Ubuntu 18.04, and CentOS 7
CPU	X64 architecture with SSE2 instruction set support	X64 architecture with SSE2 instruction set support	X64 architecture with SSE2 instruction set support
Graphics API	DX10, DX11, and DX12-capable GPUs	Metal-capable Intel and AMD GPUs	OpenGL 3.2+ or Vulkan-capable, Nvidia and AMD GPUs.
Additional requirements	Hardware vendor officially supported drivers	Apple officially supported drivers	Gnome desktop environment running on top of X11 windowing system, Nvidia official proprietary graphics driver or AMD Mesa graphics driver. Other configuration and user environment as provided stock with the supported distribution (Kernel, Compositor, etc.)
	For all operating systems, the Unity Editor is supported on workstations or laptop form factors, running without emulation, container or compatibility layer.		

Figura 6.1: Taula requisits mínims Unity [5]

6.2.2 Requisits *software*

Llista del programari necessari:

- Unity (versió 2019.4 o posterior)
- Python (3.6.1 o posterior)
- ML-Agents (release 12 o posterior)

7 Estudis i decisions

7.1 Unity

Unity, comunament conegut com Unity3D (veure logotip Figura 7.1), és un motor de videojocs de llicència gratuïta (si els ingressos o els fons reunits per projectes són inferiors a 100 mil USD en els últims 12 mesos) i un entorn de desenvolupament integrat (*Integrated Development Environment* o IDE) per a crear mitjans interactius, normalment videojocs. És famós per la ràpida capacitat de creació de prototips i l'adaptabilitat a tota classe de projectes.

Figura 7.1: Logotip Unity

La primera versió de Unity (1.0.0) va ser creada per David Helgason, Joachim Davant i Nicholas Francis a Dinamarca el 6 de juny de 2005. L'objectiu era crear un motor de videojocs assequible amb eines professionals per als desenvolupadors aficionats i, al mateix temps, "democratitzar la indústria del desenvolupament de videojocs".

Quan es va llançar originalment, Unity només estava disponible per a Mac OS X, i els desenvolupadors només podien desplegar les seves creacions en unes poques plataformes. Les versions més actuals són compatibles amb Windows 7 i Windows 10, Mac OS X, i Linux (Ubuntu), i ofereix almenys una dotzena de plataformes de destí.

Per què Unity d'entre tots el motors?

Actualment, hi ha diversos motors gratuïts molt competents, com Unreal Engine, Godot, CryEngine, etc. Tot i això, per a la realització de cinemàtica inversa i entrenament d'agents es va trobar que Unity ho tenia millor implementat (implementació a alt nivell i molt ben documentada).

7.1.1 Llenguatge de programació

En anteriors versions, Unity podia funcionar amb scripts escrits en C#, UnityScript i Boo. Actualment, només es pot programar en **llenguatge C#**.

C# (pronunciat "si sharp" en anglès) és un llenguatge de programació modern, basat en objectes i amb seguretat de tipus, que permet als desenvolupadors crear molts tipus d'aplicacions segures i sòlides que s'executen en l'ecosistema **.NET**. Té les arrels en la família de llenguatges C, i, per tant, als programadors de C, C++, Java i Javascript els resultarà familiar immediatament.

C# és un llenguatge de programació orientat a objectes. Aquest proporciona construccions de llenguatge per a admetre directament aquests conceptes, per la qual cosa es tracta d'un llenguatge natural en el qual crear i usar components de programari.

Òbviament, com que no hi ha alternatives, s'ha hagut de realitzar el treball en aquest llenguatge.

7.1.2 Entorn de desenvolupament

Els *scripts* es poden modificar en qualsevol editor que es tingui instal·lat. Així i tot, Unity té integrat un entorn de desenvolupament, *Visual Studio* (Figura 7.2), que s'instal·la per defecte.

Un entorn de desenvolupament integrat (*Integrated Development Environment* o IDE) és un programa informàtic que proporciona eines i facilitats pel desenvolupament d'altres programes informàtics.

Té característiques que ajuden a l'escriptura i edició de codi. Aquestes inclouen, entre d'altres:

- **Editor de codi font:** editor de text que ajuda a escriure el codi de programari amb funcions com el ressaltat de la sintaxi amb indicacions visuals, l'autocompletatge específic del llenguatge i la comprovació d'errors a mesura que s'escriu el codi.
- **Automatització de compilacions locals:** eines que automatitzen tasques senzilles i iteratives com a part de la creació d'una compilació local del programari per l'ús per part del desenvolupador, com la compilació del codi font en un codi binari, l'empaquetatge del codi binari i l'execució de proves automatitzades.
- **Depurador:** programa que serveix per a provar altres programes i mostrar la ubicació d'un error en el codi original de manera gràfica.

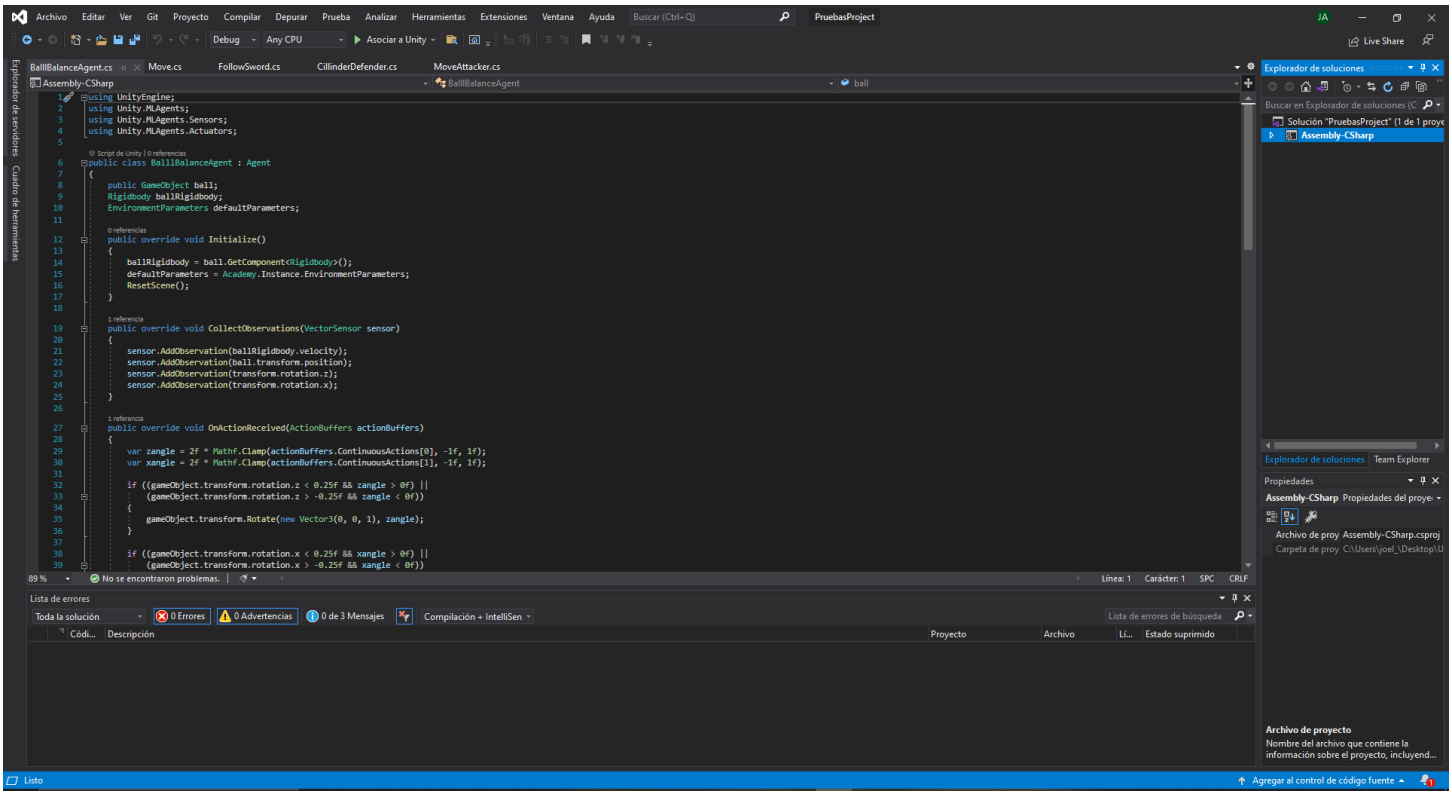


Figura 7.2: Interfície Visual Studio

7.1.3 Conceptes bàsics

Escenes: Les escenes (Figura 7.3) són el lloc on es treballa amb el contingut en Unity. Són elements que contenen tot o part d'un videojoc o aplicació. Es pot construir un videojoc simple en una sola escena, mentre que per a un de més complex, es podria utilitzar una escena per nivell, cadascuna amb els entorns propis, personatges, obstacles, decoracions i HUD. Es pot crear qualsevol nombre d'escenes en un projecte.

GameObjects: Els actius d'aquestes escenes s'anomenen *GameObjects*, els quals poden representar multitud de coses (personatges, escenaris, llum...).

GameObjects indispensables: Un actiu GameObject important és la càmera. Crea una imatge des d'un punt de vista particular en l'escena. El resultat es renderitza en la pantalla o es captura com una textura. Les càmeres són essencials per a mostrar el videojoc al jugador. Es poden personalitzar, programar i controlar per a aconseguir qualsevol mena d'efecte imaginable juntament amb un altre GameObject molt important, la llum. Aquest GameObject indica la intensitat, direcció i color de la llum de l'escena. Pot haver-hi diferents GameObjects amb diferents components de llum per crear l'efecte buscat.

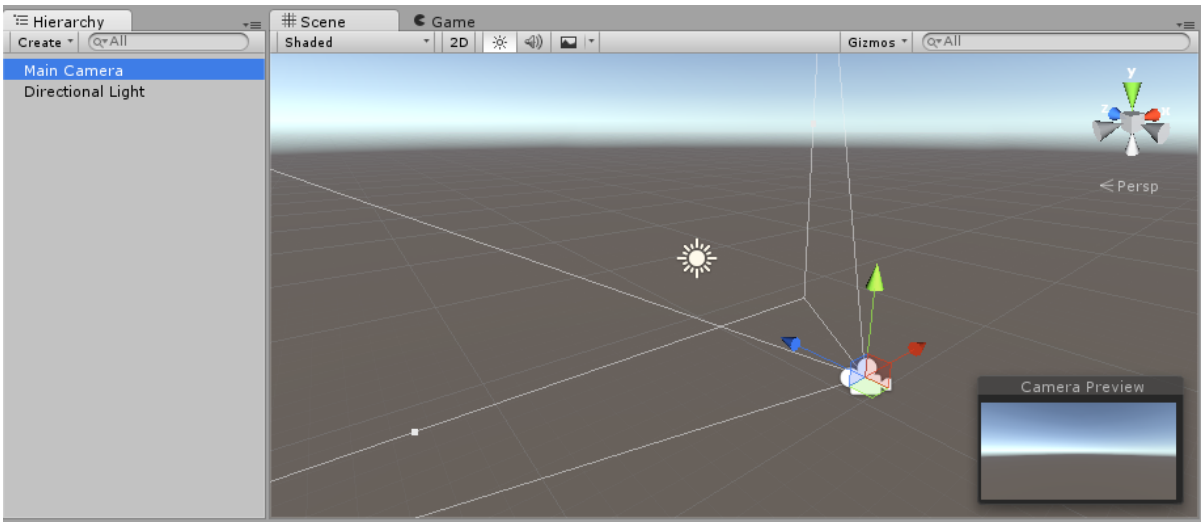


Figura 7.3: Exemple d'escena per defecte de Unity [5]

Prefabs: El sistema *Prefab* de Unity permet crear, configurar i emmagatzemar un GameObject complet amb tots els components, valors de propietats i GameObjects fills com un recurs reutilitzable. El recurs Prefab actua com una plantilla des de la qual es pot crear noves instàncies Prefab en una escena.

Quan es vol reutilitzar un GameObject configurat d'una manera particular, com un personatge no jugador (PNJ), un accessori o una peça d'escenari, en múltiples llocs de l'escena, o a través de múltiples escenes en el projecte, es pot utilitzar un Prefab. Això és millor que simplement copiar i enganxar el GameObject, perquè el sistema de Prefab permet mantenir automàticament totes les còpies sincronitzades per si es fan canvis.

Tags: Un *Tag* és una paraula de referència que es pot assignar a un o més GameObjects. Ajuden a identificar GameObjects per a propòsits de programació. Asseguren que no es necessita afegir manualment GameObjects com a referència, estalviant així temps quan s'està usant el mateix codi de script en múltiples GameObjects.

Els Tags són útils pels Colliders (explicats al següent Apartat 7.1.4: Classes), ja que és molt senzill comprovar el Tag del GameObject amb el qual s'ha produït una col·lisió.

Layers: Les *Layers* a Unity defineixen quins GameObjects poden interactuar amb diferents característiques i entre si. S'utilitza, per exemple, per la detecció de col·lisions per capes, fent que un GameObject col·lideixi amb un altre GameObject que està configurat a una capa o capes específiques.

Namespaces: Els *Namespaces* són una col·lecció de classes a les quals es fa referència utilitzant un prefix triat en el nom de la classe. Se sol fer referència al principi del script de la següent manera: *usign* + nomDelNamespace (Ex. using UnityEngine).

Utilitzades en els scripts d'aquest projecte: UnityEngine, UnityEngine.UI, System.Collections (integrades a Unity), i Unity.MLAgents, Unity.MLAgents.Sensors, i Unity.MLAgents.Actuators (pertanyents a ML-Agents).

7.1.4 Classes

A la figura de la següent pàgina, la Figura 7.4, es pot observar un diagrama de classes que representa alguns elements bàsics de Unity que s'utilitzaran durant el projecte. Es mostra només els atributs i mètodes rellevants, no tots els que tenen.

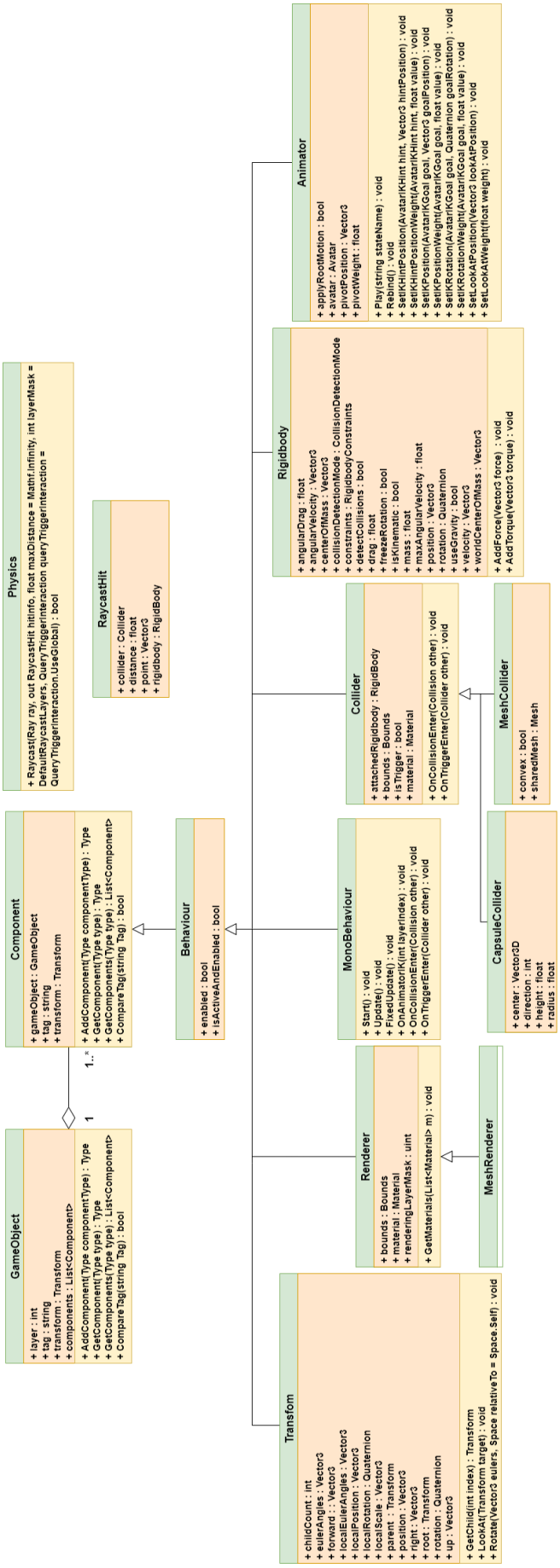


Figura 7.4: Diagrama de classes UML de Unity

A continuació, hi ha descrites cada una de les classes anteriors:

- **GameObject [Figura 7.5]** - Els GameObjects són els objectes fonamentals en Unity que representen als personatges, accessoris i escenaris. Actuen com a contenidors per als Components que implementen funcionalitats. Hi ha uns d'especials que se'ls anomena *Empty*. Són GameObjects sense cap component extra que serveixen principalment com a contenidor, és a dir, un GameObject com a pare en la jerarquia d'un grup de GameObjects.

GameObject		
Nom propietat	**Tipus**	**Descripció**
layer	int	La capa on es troba l'objecte del joc.
tag	string	L'etiqueta d'aquest objecte de joc.
transform	Transform	El component Transform adjunt a aquest objecte.
components	List<Component>	Llista de components que té l'objecte. Mínim té el transform.

GameObject			
Nom mètode	**Retorn**	**Paràmetres**	**Descripció**
AddComponent	Type	Type componentType	Afegeix un component de classe *Type* al GameObject.
GetComponent	Type	Type type	Retorna el component de *Type* type si el GameObject té un adjunt, nul si no en té.
GetComponents	List<Type>	Type type	Retorna tots els components de *Type* type si el GameObject té un adjunt, nul si no en té.
CompareTag	Bool	String tag	Retorna *true* si el GameObject està etiquetat com a tag, *false* altrament.

Figura 7.5: Classe "GameObject"

- **Component [Figura 7.6]** - Atributs que defineixen el comportament del GameObject. El component bàsic i no esborrable és el *Transform*, que representa la posició, escala i l'orientació de l'objecte.

Component		
Nom propietat	**Tipus**	**Descripció**
gameObject	GameObject	GameObject al que està adjuntat.
tag	string	L'etiqueta del GameObject adjuntat.
transform	Transform	El component Transform adjuntat al gameObject.

Component			
Nom mètode	**Retorn**	**Paràmetres**	**Descripció**
AddComponent	Type	Type componentType	Afegeix un component de classe *Type* al GameObject adjuntat.
GetComponent	Type	Type type	Retorna el component de *Type* type si GameObject adjuntat té un adjunt, nul si no en té.
GetComponents	List<Type>	Type type	Retorna tots els components de *Type* type si GameObject adjuntat té un adjunt, nul si no en té.
CompareTag	Bool	String tag	Retorna *true* si el GameObject adjuntat està etiquetat com a tag, *false* altrament.

Figura 7.6: Classe "Component"

- **Physics [Figura 7.7]** - Classe que implementa les propietats de la física global i mètodes d'ajuda.

Physics			
Nom mètode	**Retorn**	**Paràmetres**	**Descripció**
RayCast	bool	Ray ray , out RaycastHit hitInfo, float maxDistance = Mathf.Infinity, int layerMask = DefaultRaycastLayers, QueryTriggerInteraction queryTriggerInteraction = QueryTriggerInteraction.UseGlobal	Retorna *true* quan el raig interseca qualsevol collider, en cas contrari *false*. ray: El punt de partida i la direcció del raig. maxDistance: La distància màxima a la qual el raig ha de buscar col·lisions. layerMask: LayerMask que s'utilitza per a ignorar selectivament certs Colliders en llançar un raig. queryTriggerInteraction: Especifica si aquest *query* ha de colpejar als Triggers.

Figura 7.7: Classe "Physics"

- **RaycastHit [Figura 7.8]** - Classe que implementa l'estructura utilitzada per obtenir informació d'un llançament de raigs (*raycast*).

RaycastHit		
Nom propietat	**Tipus**	**Descripció**
collider	Collider	El collider que ha col·lidit amb el raig.
distance	Float	La distància des de l'origen del raig fins al punt d'impacte.
point	Vector3	El punt d'impacte en l'espai del món on el raig va colpejar el Collider.
rigidBody	RigidBody	El RigidBody del collider que ha estat colpejat. Si el Collider no està unit a un RigidBody, llavors és nul.

Figura 7.8: Classe "RaycastHit"

- **Behaviour [Figura 7.9]** - Classe de la que deriven molts components. Implementa l'activació o desactivació.

Behaviour		
Nom propietat	**Tipus**	**Descripció**
enabled	Bool	*True* si el GameObject està habilitat, *false* altrament. Deshabilitar permet no deixar accedir al GameObject sense haver de destruir-lo, permetent tornar-lo a activar.
isActiveAndEnabled	Bool	Si un GameObject està *enabled* i té un script habilitat llavors retornarà true, *false* altrament.

Figura 7.9: Classe "Behaviour"

- **Transform [Figura 7.10]** - El component bàsic i no esborrable de tot GameObject, que representa la posició, l'escala i l'orientació del objecte.

Transform		
Nom propietat	**Tipus**	**Descripció**
childCount	Int	El nombre de fills que té el pare Transform.
eulerAngles	Vector3	La rotació com a angles d'Euler, en graus.
forward	Vector3	Vector en l'eix Z normalitzat en coordenades de món.
localEulerAngles	Vector3	La rotació com a angles d'Euler en graus respecte a la rotació del Transform del pare.
localPosition	Vector3	Posició espacial en relació amb el Transform del pare.
localRotation	Quaternion	La rotació relativa a la rotació del Transform del pare en Quaternion.
localScale	Vector3	L'escala relativa al GameObject pare.
parent	Transform	El pare en la jerarquia de l'escena.
position	Vector3	La posició espacial en coordenades de món.
right	Vector3	Vector en l'eix X normalitzat en coordenades de món.
root	Transform	Transform més alt de la jerarquia a l'escena.
rotation	Quaternion	Quaternion que emmagatzema la rotació en coordenades de món.
up	Vector3	Vector en l'eix Y (vector normal) normalitzat en coordenades de món.

Transform			
Nom mètode	**Retorn**	**Paràmetres**	**Descripció**
GetChild	Transform	Int index	Retorna el *Transform* del fill nombre *index*.
LookAt	Void	Transform target	Gira el Transform perquè el vector *forward* apunti a la posició actual de *target*.
Rotate	Void	Vector3 euler, Space relativeTo = Space.Self	Utilitza Transform.Rotate per a girar el GameObject en una direcció *euler*. També es determina si rotar el GameObject localment al GameObject o relatiu a coordenades de món.

Figura 7.10: Classe "Transform"

- **Renderer [Figura 7.11]** - Un *renderer* és el que fa que un objecte aparegui en la pantalla. Agrega textures de mapa de bits o textures per procediments, llums, mapatge topològic i posició relativa a l'escena, i es visualitza tot per la càmera per a la qual s'està fent el render.

Renderer		
Nom propietat	**Tipus**	**Descripció**
bounds	Bounds	El volum delimitador del renderitzador.
material	Material	Primer Material instanciat assignat al renderitzador.
renderingLayerMask	uint	Capa de renderitzat en la qual viu aquest renderitzador.

Renderer			
Nom mètode	**Retorn**	**Paràmetres**	**Descripció**
GetMaterials	void	List<Material> m	Retorna a *m* tots els materials instanciats d'aquest objecte.

Figura 7.11: Classe "Renderer"

- **MonoBehaviour [Figura 7.12]** - Classe base de la que deriva tot script de Unity i moltes de les classes. Implementa funcions comunes i indexa variables bàsiques per a qualsevol GameObject.

MonoBehaviour			
Nom mètode	**Retorn**	**Paràmetres**	**Descripció**
Start	Void	-	Es crida quan s'habilita un script just abans que es cridi a qualsevol dels mètodes Update per primera vegada. Mètode editable segons script.
Update	Void	-	Mètode que es crida a cada fotograma, si el MonoBehaviour està activat. Mètode editable segons script.
FixedUpdate	Void	-	Mètode que es crida independent de la velocitat de fotogrames per segon del joc, per als càlculs de física. Té la freqüència del sistema de física; es crida a velocitat fixa. Mètode editable segons script.
OnAnimatorIK	Void	Int layerIndex	Cridat pel component Animator immediatament abans d'actualitzar el sistema IK intern, indicant la layerIndex en la qual es diu al crida l'IK. Aquest callback pot ser usat per a establir les posicions dels objectius IK i els respectius pesos. Mètode editable segons script.
OnCollisionEnter	Void	Collision other	Cridat quan el Collider/RigidBody ha començat a tocar un altre Collider/RigidBody *other*. La classe *Collision* conté informació sobre els punts de contacte, la velocitat d'impacte, etc. Mètode editable segons script.
OnTriggerEnter	Void	Collider other	Quan un GameObject col·lideix amb un altre GameObject, Unity crida OnTriggerEnter, amb *other* com el Collider involucrat. Ocorre en la funció FixedUpdate quan dos GameObjects col·lideixen. Els Colliders involucrats no sempre estan en el punt de contacte inicial. Tots dos GameObjects han de contenir un component Collider. Un ha de tenir Collider.isTrigger habilitat, i contenir un Rigidbody. Mètode editable segons script.

Figura 7.12: Classe "MonoBehaviour"

- **Collider [Figura 7.13]** - Defineix la forma d'un GameObject per als propòsits de les col·lisions físiques. Un collider, que és invisible, pot tenir la mateixa forma exacta que el mesh del GameObject (*Mesh Collider*) o pot tenir colliders en formes primitives (cub, esfera i càpsula).

Collider		
Nom propietat	**Tipus**	**Descripció**
attachedRigidbody	RigidBody	El RigidBody al qual està unit el Collider.
bounds	Bounds	El volum que delimita en coordenades món del Collider.
isTrigger	bool	*True* si el collider és un activador, *false altrament*.
material	Material	El material utilitzat pel Collider.

Collider			
Nom mètode	**Retorn**	**Paràmetres**	**Descripció**
OnCollisionEnter	Void	Collision other	Cridat quan el Collider/RigidBody ha començat a tocar un altre Collider/RigidBody *other*. La classe *Collision* conté informació sobre els punts de contacte, la velocitat d'impacte, etc. Mètode editable segons script.
OnTriggerEnter	Void	Collider other	Quan un GameObject col·lideix amb un altre GameObject, Unity crida OnTriggerEnter, amb *other* com el Collider involucrat. Ocorre en la funció FixedUpdate quan dos GameObjects col·lideixen. Els Colliders involucrats no sempre estan en el punt de contacte inicial. Tots dos GameObjects han de contenir un component Collider. Un ha de tenir Collider.isTrigger habilitat, i contenir un Rigidbody. Mètode editable segons script.

Figura 7.13: Classe "Collider"

- **CapsuleCollider [Figura 7.14]** - *Collider* format per dos semi-esferes ajuntades per un cilindre.

CapsuleCollider		
Nom propietat	**Tipus**	**Descripció**
center	Vector3D	El centre de la càpsula, mesurat en l'espai local de l'objecte.
direction	int	La direcció de la càpsula.
height	float	L'alçada de la càpsula mesurada en l'espai local de l'objecte.
radius	float	El radi de l'esfera, mesurat en l'espai local de l'objecte.

Figura 7.14: Classe "CapsuleCollider"

- **MeshCollider [Figura 7.15]** - *Collider* que agafa la primitiva gràfica 3D del objecte (*Mesh*) i amb el Mesh Renderer renderitza en la posició definida pel component Transform del GameObject.

MeshCollider		
Nom propietat	**Tipus**	**Descripció**
convex	bool	Utilitza un Collider convex de la Mesh.
sharedMaterial	Mesh	L'objecte Mesh utilitzat per a la detecció de col·lisió.

Figura 7.15: Classe "MeshCollider"

- **RigidBody [Figura 7.16]** - Component principal que permet el comportament físic d'un GameObject. Amb un *Rigidbody*, l'objecte respondrà immediatament a la gravetat, i si un o més components Collider són també agregats, el GameObject és mourà si rep col·lisions.

  Ja que un component Rigidbody pren el control del moviment del GameObject al qual està unit, no s'hauria d'intentar moure'l des d'un script canviant les propietats del Transform com a posició i rotació. En canvi, s'hauria d'aplicar forces per a empènyer el GameObject i deixar que el motor de física calculi els resultats.

RigidBody		
Nom propietat	**Tipus**	**Descripció**
angularDrag	float	L'arrossegament angular de l'objecte. Serveix per alentir la rotació d'un objecte. Com més alt sigui l'arrossegament, més alenteix la rotació.
angularVelocity	Vector3	El vector de velocitat angular mesurat en radians per segon.
centerOfMass	Vector3	El centre de massa relatiu a l'origen del Transform.
collisionDetectionMode	CollisionDetectionMode	Tipus de detecció de col·lisions: Discrete, Continuous, Continuous Dynamic i Continuous Speculative
constraints	RigidbodyConstraints	Controla quins graus de llibertat es permeten per a la simulació d'aquest Rigidbody.
detectCollisions	bool	Detecció de col·lisions *true* o *false*.
drag	float	L'arrossegament de l'objecte. S'utilitza per frenar un objecte. Com més alt sigui l'arrossegament, més alenteix l'objecte.
freezeRotation	bool	Controla si la física canviarà la rotació de l'objecte. Si *true*, la rotació no es modifica amb la simulació física.
isKinematic	bool	Controla si la física afecta al RigidBody. Si *true*, les forces, col·lisions o articulacions no afectaran més RigidBody. Estarà sota el control total d'animacions o via script canviant transform.position. Els cossos cinemàtics també afecten el moviment d'uns altres rigidBodies a través de col·lisions o articulacions.
mass	float	La massa del RigidBody. Els objectes de massa més alta empenyen més els objectes de massa més baixa quan col·lideixen.
maxAngularVelocity	float	La velocitat angular màxima del RigidBody mesurada en radians per segon.
position	Vector3	La posició del RigidBody.
rotation	Quaternion	La rotació del RigidBody.
useGravity	bool	Controla si la gravetat afecta aquest RigidBody. Si és *false*, el cos rígid es comportarà com a l'espai exterior.
velocity	Vector3	Vector de velocitat del RigidBody. Representa la taxa de canvi de la posició del Rigidbody.
worldCenterOfMass	Vector3	El centre de massa del RigidBody en coordenades món.

RigidBody			
Nom mètode	**Retorn**	**Paràmetres**	**Descripció**
AddForce	Void	Vector3 force	Afegeix una força al RigidBody. Aquesta força s'aplica contínuament al llarg de la direcció del vector *force*.
AddTorque	Void	Vector3 torque	Afegeix un parell de torsió (torque) al RigidBody.

Figura 7.16: Classe "RigidBody"

- **Animator [Figura 7.17]** - Component que s'utilitza per assignar-li un *AnimatorController*, un diagrama de transaccions (o màquina d'estats) on cada estat realitza una animació i, en cas de que es donin unes condicions, canviar d'estat per realitzar una altra animació.

Animator		
Nom propietat	**Tipus**	**Descripció**
applyRootMotion	bool	El *root motion* és l'efecte on la Mesh sencera d'un objecte s'allunya del punt de partida, però aquest moviment és creat per la mateixa animació en lloc de canviar la posició "Transform".
avatar	Avatar	L'avatar és un determinat disseny d'un model animat dividit en parts. Si es un humanoide, té les cames, els braços, el cap i el cos.
pivotPosition	Vector3	Posició actual del pivot. El pivot és el punt més estable entre el peu esquerre i el dret de l'avatar.
pivotWeight	float	Obté el pes del pivot. Per a un valor de 0, el peu esquerre és el punt més estable. Per a un valor de 1, el peu dret és el punt més estable.

Animator			
Nom mètode	**Retorn**	**Paràmetres**	**Descripció**
Play	Void	string stateName	Reprodueix l'animació/l'estat *stateName*.
Rebind	Void	-	Reenllaça totes les propietats animades i les dades de la malla amb l'Animator.
SetIKHintPosition	Void	AvatarIKHint hint, Vector3 hintPosition	Estableix la posició *hintPosition* a seguir pel sistema IK del element de l'avatar *hint*.
SetIKHintPositionWeight	Void	AvatarIKHint hint, float value	Estableix el pes d'un *hint* IK a *value* (0 = animació original abans de IK, 1 = segueix la *hintPosition*).
SetIKPosition	Void	AvatarIKGoal goal, Vector3 goalPosition	Estableix la posició objectiu, *goalPosition*, d'una part IK, *goal*. Una part IK és una posició i rotació objectiu per a una part específica del cos. Unity calcula com moure la part cap a l'objectiu.
SetIKPositionWeight	Void	AvatarIKGoal goal, float value	Estableix el pes d'un *goal* IK a *value* (0 = animació original abans de IK, 1 = segueix la *goalPosition*).
SetIKRotation	Void	AvatarIKGoal goal, Quaternion goalRotation	Estableix la rotació objectiu, *goalRotation*, d'una part IK, *goal*.
SetIKRotationWeight	Void	AvatarIKGoal goal, float value	Estableix el pes de rotació d'un *goal* IK a *value* (0 = animació original abans de IK, 1 = segueix la *goalRotation*).
SetLookAtPosition	Void	Vector3 lookAtPosition	Estableix la posició a observar, *lookAtPosition*, per l'IK del cap.
SetLookAtWeight	Void	float weight	Estableix el pes del IK del cap a *weight* (0 = animació original abans de IK, 1 = segueix la *lookAtPosition*).

Figura 7.17: Classe "Animator"

7.1.5 Interfície

La Figura 7.18 mostra la interfície del motor Unity:

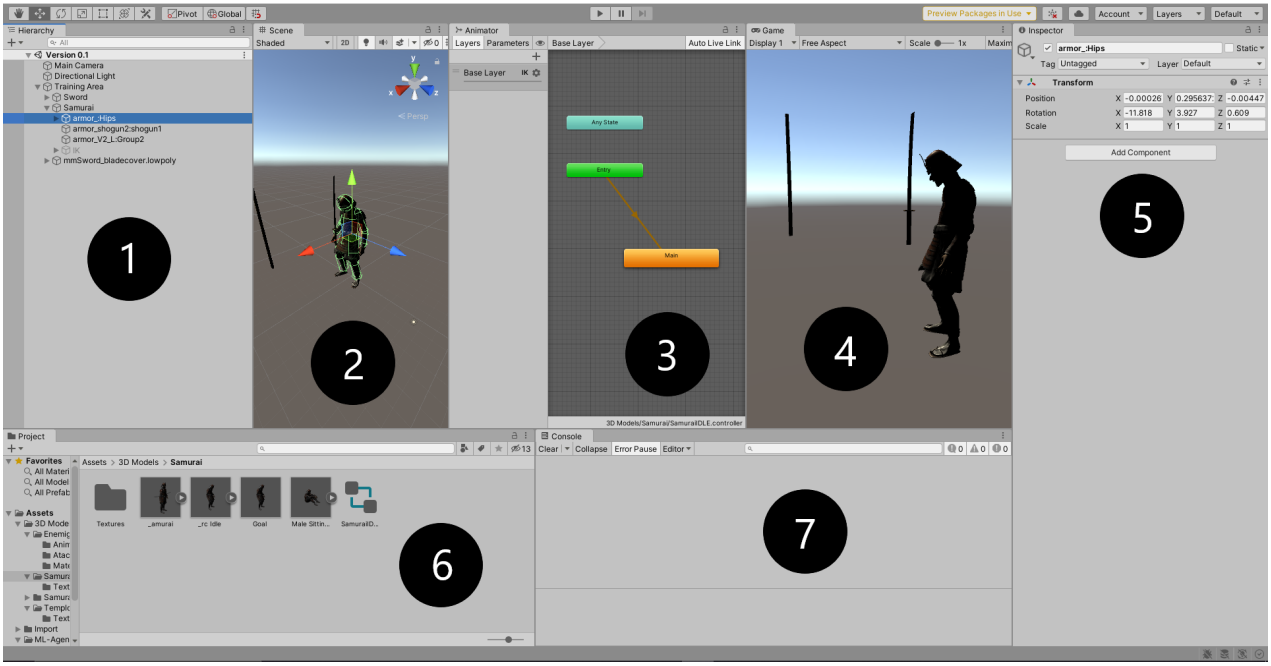


Figura 7.18: Interfície de Unity

Descripció:

1. **Finestra de jerarquies:** representació en text de la jeràrquica de cada GameObject en l'escena actual. Cada element de l'escena té una entrada en la jerarquia, per la qual cosa les dues finestres estan intrínsecament vinculades. La jerarquia revela l'estructura de com els GameObjects s'uneixen entre si.

2. **Finestra vista escena:** permet navegar i editar visualment una escena. La vista d'escena pot mostrar una perspectiva 3D o 2D, en funció del tipus de projecte que es treballi.

3. **Finestra d'animació:** permet crear i modificar "Clips d'animació" directament dins de Unity. A més d'animar el moviment, l'editor també li permet animar variables de materials i components i augmentar els clips d'animació amb esdeveniments d'animació, funcions en punts específics al llarg de la línia de temps.

4. **Finestra vista de joc:** simula l'aspecte del joc renderitzat final a través de les càmeres de l'escena. En fer clic al botó Reprodueix de la barra d'eines (barra superior de la finestra general), comença la simulació.

5. **Finestra Inspector:** permet veure i editar totes les propietats (components) del GameObject seleccionat. Com que els diferents tipus de GameObjects tenen diferents conjunts de propietats, el disseny i el contingut del fitxer Inspector canvia de finestra cada vegada que seleccioneu un GameObject diferent.

6. **Finestra del Projecte:** mostra la biblioteca d'actius disponibles per utilitzar-los al projecte. Quan s'importen recursos al projecte, apareixen aquí.

7. **Finestra consola**: mostra errors, advertiments i altres missatges generats per Unity.

7.1.6 Cinemàtica inversa a Unity

El sistema que controla les animacions a Unity, anomenat ***Mecanim***, té implementat una sèrie de funcions que permeten realitzar cinemàtica inversa amb un personatge que sigui **humanoide**, és a dir, dues cames i dos braços, i que tingui un **"Avatar"** correctament configurat (conjunt jeràrquic de parts interconnectades anomenades ossos o *bones* que col·lectivament formen l'esquelet o *rig* amb el qual s'anima movent aquestes parts, ja que la malla o pell del personatge segueix a aquestes parts. Vegeu Figura 1.1b).

Explicat de manera senzilla, utilitzant el callback *OnAnimatorIK()* podem establir el que s'anomena com a **pes** o ***weight*** de les cames i els braços cap a un **objectiu** o ***goal***, mitjançant les següents funcions explicades anteriorment: **SetIKPositionWeight, SetIKRotationWeight, SetIKPosition, SetIKRotation, SetLookAtPosition**. El pes estableix un valor en el rang [0..1] per a determinar que tan lluny ha d'estar la posició inicial del braç/cama i la de l'objectiu. Per exemple, si el valor del pes de la mà dreta en direcció a una paret és 1, es calcula com queda la mà intentant arribar a l'objectiu i com queda modificada la cadena cinemàtica restant (colze i espatlla).

L'humanoide de la Figura 7.19 és un exemple que proporciona Unity. Com es pot observar, la mà dreta té com a objectiu el cilindre. Com que aquest s'ha mogut cap el seu pit, el sistema a calculat que el plegament del colze, tot creant una postura versemblant.

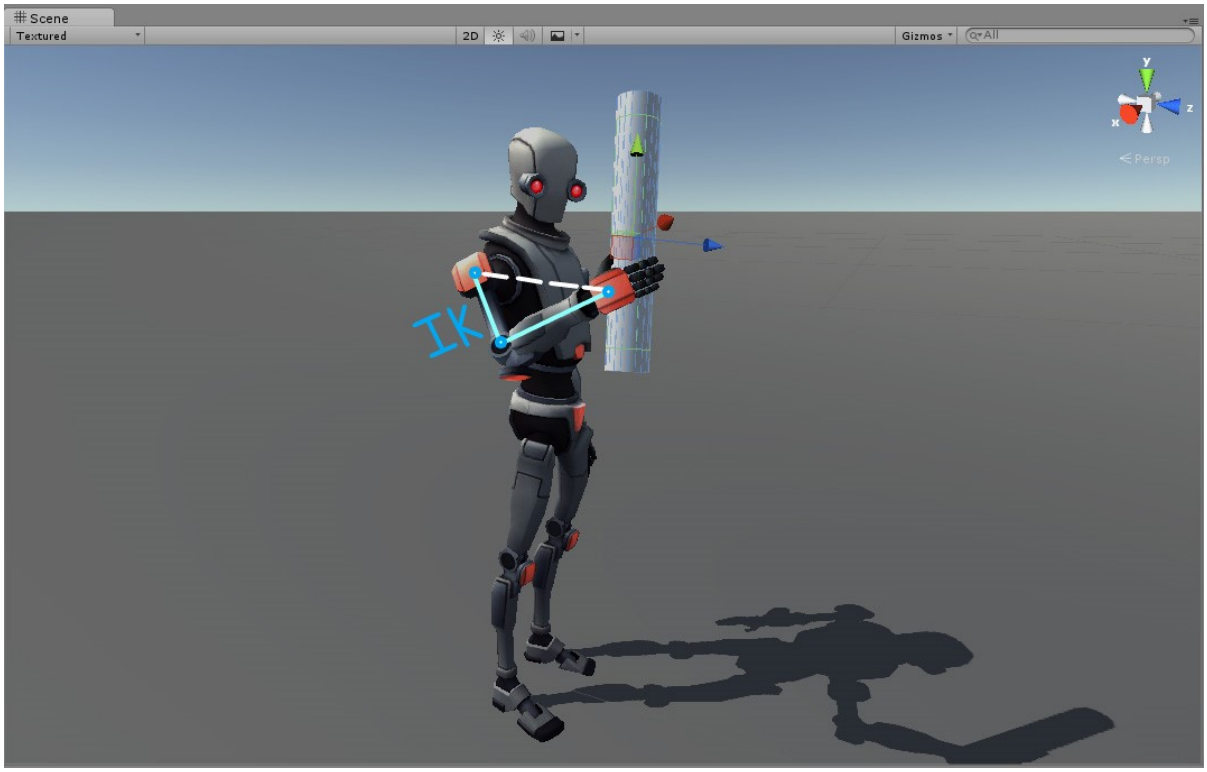


Figura 7.19: Exemple de cinemàtica inversa a Unity

7.2 ML-Agents

Unity Machine Learning Agents Toolkit (ML-Agents Toolkit) és un projecte de codi obert que permet que els videojocs i les simulacions al motor de Unity serveixin com a entorns per a **l'entrenament d'agents intel·ligents**. Els agents es poden entrenar utilitzant l'aprenentatge per reforç, l'aprenentatge per imitació, la neuroevolución o altres mètodes d'aprenentatge automàtic a través d'una API de Python. També proporciona implementacions (basades en PyTorch) d'algorismes d'última generació per entrenar fàcilment agents intel·ligents per a videojocs 2D, 3D i VR/AR. Aquests agents entrenats poden ser utilitzats per a múltiples propòsits, incloent-hi el control del comportament dels personatges no jugables (en una varietat d'escenaris com ara multi-agent i adversaris), proves automatitzades de construccions de videojocs i avaluació de diferents decisions de disseny abans del llançament.

7.2.1 Components

El ML-Agents Toolkit conté **5 components** d'alt nivell:

1. **Learning environment:** Format per l'escena Unity i tots els personatges del joc. L'escena Unity proporciona l'entorn en el qual els agents observen, actuen i aprenen. La manera de configurar l'escena Unity perquè serveixi d'entorn d'aprenentatge depèn realment de l'objectiu. Es pot utilitzar la mateixa escena tant per a l'entrenament com per a provar els agents entrenats.
2. **Python Low-Level API:** Interfície Python de baix nivell per a interactuar i manipular un entorn d'aprenentatge. A diferència de l'entorn d'aprenentatge, l'API de Python no forma part de Unity, sinó que viu fora i es comunica amb Unity a través d'un comunicador. Aquesta API està continguda en un paquet dedicat de Python i és usada pel procés d'entrenament de Python per a comunicar-se amb i controlar l'acadèmia durant l'entrenament. L'acadèmia és un objecte *Singleton* que controla el temps, el reinici i els ajustos d'entrenament/inferència de l'entorn. És l'element que orquestra als agents i el procés de presa de decisions.
3. **External Communicator:** Connector de l'entorn d'aprenentatge amb l'API de baix nivell de Python. Viu dins de l'entorn d'aprenentatge.
4. **Python Trainers:** Conté tots els algorismes d'aprenentatge automàtic que permeten entrenar als agents. Els algorismes estan implementats en Python.
5. **Gym Wrapper:** Una forma comuna en la qual els investigadors d'aprenentatge automàtic interactuen amb entorns de simulació és a través d'un embolcall anomenat *gym*. És un conjunt d'eines per a desenvolupar i comparar algorismes d'aprenentatge per reforç. OpenAI en proporciona un dintre d'un paquet Python dedicat i instruccions per a usar-lo amb algorismes d'aprenentatge automàtic existents.

7.2.2 Funcionament

El *Learning Enviornment* o entorn d'aprenentatge conté **2 components** que ajuden a organitzar l'escena de Unity:

1. **Agent:** Unit a un GameObject de Unity (qualsevol personatge dins d'una escena) i s'encarrega de generar les observacions, realitzar les accions que rep i assignar una recompensa (positiva o negativa) quan és apropiat. Cada Agent està vinculat a un *Behavior*.
2. **Behavior:** Defineix atributs específics de l'agent com el nombre d'accions que pot prendre. Cada Behavior s'identifica de manera única per un camp "Behavior name". Es pot pensar com una funció que rep observacions i recompenses de l'agent i retorna accions. Hi ha de tres tipus: **Aprenentatge**, **Heurística** o **Inferència**. Un "Behavior d'aprenentatge" encara no està definit, sinó que està a punt de ser entrenat. Un "Behavior heurístic" és aquell que es defineix per un conjunt de regles codificades en codi. Un "Behavior d'inferència" és aquell que inclou un fitxer de xarxa neuronal entrenat. En essència, després que un "Behavior d'aprenentatge" estigui entrenat, es converteix en un "Behavior d'inferència".

Cada Learning Enviornment tindrà sempre un agent per a cada personatge de l'escena. Tot i que cada agent ha d'estar vinculat a un Behaviour, és possible que els agents que tenen observacions i accions similars tinguin el mateix Behaviour, però això no significa que en cada cas tinguin valors d'observació i acció idèntics.

El següent esquema (Figura 7.20) mostra aquest comportament d'agent i behaviour.

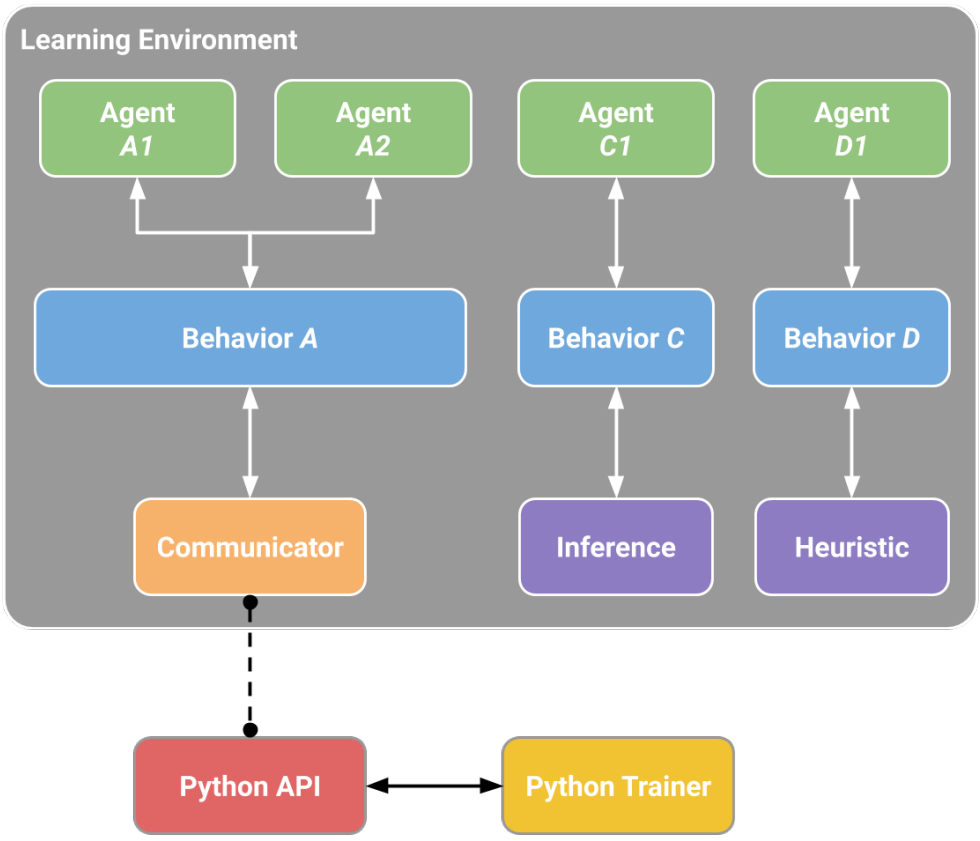


Figura 7.20: Esquema funcionament ML-Agents [6]

A través del que s'anomena com ***Academy*** o **acadèmia** es garanteix que tots els agents estan sincronitzats, a més de controlar la configuració de tot l'entorn. L'esquema d'alt nivell de la Figura 7.21, mostra els agents connectats amb el seu *brain* (behavior amb els inputs i outputs del procés de decisió) i amb l'academy.

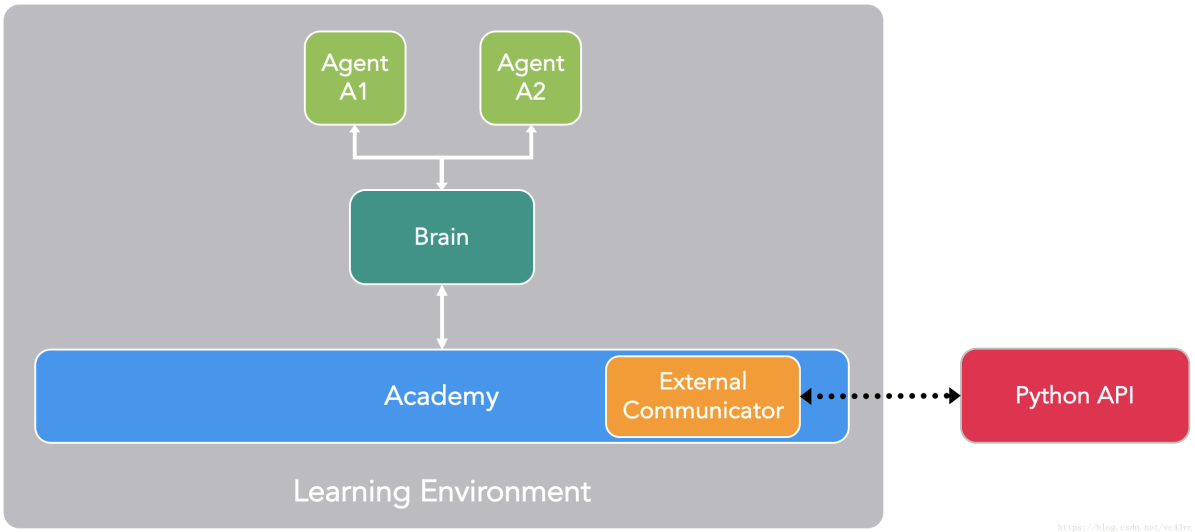


Figura 7.21: Esquema Academy ML-Agents [6]

A continuació, necessitem definir **3 entitats** referents a l'agent a entrenar a l'escenari d'entrenament, anomenat **entorn**:

1. **Observations:** el que l'agent percep de l'entorn. Les observacions poden ser numèriques i/o visuals. Les numèriques mesuren els atributs de l'entorn des del punt de vista de l'agent. Per a la majoria dels entorns interessants, un agent necessitarà diverses observacions numèriques contínues. Les observacions visuals, d'altra banda, són imatges generades per les càmeres connectades a l'agent i representen el que l'agent està veient en un moment determinat. És habitual confondre l'observació d'un agent amb l'estat de l'entorn (o del joc). L'estat de l'entorn representa informació sobre tota l'escena que conté tots els personatges del joc. L'observació dels agents, tanmateix, només conté informació de la qual l'agent és conscient i sol ser un subconjunt de l'estat de l'entorn. Per exemple, l'observació d'un agent pot no incloure informació sobre un enemic amagat que ell desconeix.

2. **Actions:** accions que pot realitzar l'agent. Igual que les observacions, les accions poden ser contínues o discretes en funció de la complexitat de l'entorn i de l'agent. Si l'entorn és un simple món quadriculat en el qual només importa la ubicació del mateix agent, llavors n'hi ha prou amb una acció discreta que prengui un dels quatre valors (nord, sud, est, oest). No obstant això, si l'entorn és més complex i l'agent pot moure's lliurement, és més apropiat utilitzar dues accions contínues (una per a la direcció i una altra per a la velocitat).

3. **Reward signal:** un valor escalar que indica que tan bé ho està fent l'agent. Cal tenir en compte que el *reward signal* no té per què proporcionar-se a cada moment, sinó només quan l'agent realitza una acció que és bona o dolenta. Aquesta és la forma en què es comunica a l'agent els objectius, per la qual cosa han d'establir-se de manera que la maximització de la recompensa generi el comportament òptim desitjat.

Una vegada definides aquestes entitats (els components bàsics d'una tasca d'aprenentatge per reforç, explicats en l'Apartat 5: Marc de treball i conceptes previs), podem entrenar el comportament de l'agent. Això s'aconsegueix simulant l'entorn durant molts assajos en els quals, amb el temps, aprèn quina és l'acció òptima per a cada observació que mesura maximitzant la recompensa futura.

En la terminologia de l'aprenentatge per reforç, el comportament que s'aprèn es diu política, que és essencialment un mapa (òptim) de les observacions a les accions. El procés d'aprenentatge d'una política mitjançant simulacions es denomina fase d'entrenament, mentre que el joc amb un PNJ que utilitza la política apresa es denomina fase d'inferència.

7.2.3 Classes

Les classes de ML-Agents que s'utilitzen en aquest projecte estan integrades en el paquet *ML-Agents package* (com.unity.ml-agents) que s'ha d'instal·lar a Unity (passos a seguir a l'Apartat 14: Manual d'usuari i d'instal·lació) i es fa servir el Namespace corresponent per fer-hi referencies en els scripts. També és important remarcar que aquestes classes descrites a continuació són de la **Release 12**. Per tant, alguns aspectes poden estar desfasats respecte de versions més recents.

Unity.MLAgents

- **Class "Agent"**
  Classe que representa l'agent a entrenar. S'ha de crear com a subclasse en un script per a implementar-lo. Aquest script s'ha d'agregar a un GameObject de l'escena del projecte de Unity, la qual farà d'entorn.

  La classe és una extensió de la classe de Unity MonoBehaviour. Per tant, es poden implementar les funcions estàndard de MonoBehaviour sense cap problema. A més a més, les observacions i accions d'un agent típicament prenen lloc durant la fase FixedUpdate, per tant, es deuria solament utilitzar la funció MonoBehaviour.Update per a propòsits cosmètics.

  Els *steps*: Els agents operen en temps d'*steps*. En cada step, l'agent recol·lecta observacions, les passa a la seva política de presa de decisions, i rep un vector d'acció com a resposta. Per a cada simulació o entrenament, s'hi estableix un màxim de steps, o il·limitats, a la variable MaxStep. Si s'estableix el valor a una estimació raonable del temps que hauria l'agent de prendre per a completar una tasca, llavors els agents que no han tingut èxit en aquest marc de temps es reiniciaran i començaran un nou episodi d'entrenament en lloc de continuar perdent el temps.

  Observacions: Els agents realitzen observacions utilitzant la *interface* (classe abstracta que només pot contenir mètodes i propietats abstractes, és a dir, sense implementar) anomenada ISensor. La API de ML-Agents proporciona implementacions diferents per a diferents tipus de sensors, que s'explicaran en el Namespace Unity.MLAgents.Sensors.

  Decisions: Un agent té disponibles diferents maneres de prendre decisions, és a dir, hi ha diferents polítiques de presa de decisions. Aquestes estan descrites en el la classe *BehaviorParameters*, que fa de component al mateix GameObject de l'agent. Un dels paràmetres, *BehaviorType*, un nombre enter, determina el tipus:

  0. **Default:** utilitza el procés remot (xarxa neuronal en entrenament) per a la presa de decisions. Si no està disponible, utilitzarà la inferència (xarxa neuronal ja entrenada) i si no es proporciona cap model, utilitzarà l'heurística.
  1. **HeuristicOnly:** quan es necessita prendre una decisió, es crida a la funció *Heuristic(ActionBuffers)* de l'agent. La implementació és responsable de proporcionar l'acció apropiada. Implementar-la és útil per fer *debugging*. Per exemple, es pot utilitzar l'entrada del teclat per a seleccionar les accions de l'agent amb la finalitat de controlar manualment el comportament de l'agent.
  2. **InferenceOnly:** les decisions es prenen sempre utilitzant el model entrenat especificat en el component BehaviorParameters.

Per a desencadenar una decisió de l'agent automàticament, s'adjunta el component *DecisionRequester* al GameObject de l'agent. També es pot cridar manualment amb la funció *RequestDecision()* de l'agent. Només és necessari cridar-la quan l'agent està en posició d'actuar sobre la decisió. En molts casos, això serà cada callback del FixedUpdate, però podria ser menys freqüent. Per exemple, un agent que salta en l'entorn només podria prendre una acció quan toca el terra, per la qual cosa podrien transcórrer diversos fotogrames entre una decisió i la necessitat de la següent.

Accions: S'utilitza la funció *OnActionReceived(ActionBuffer)* per a implementar les accions que l'agent pot realitzar, com moure's per a aconseguir un objectiu o interactuar amb l'entorn.

Episodis d'entrenament: Quan es crida a la funció *EndEpisode()* en un agent o l'agent arriba al MaxStep, l'episodi d'entrenament actual acaba. Això vol dir que l'agent ha realitzat una simulació, un entrenament, i ha obtingut un resultat. Es pot reiniciar a l'agent (per realitzar més episodis), o remoure'l de l'entorn, implementant la funció *OnEpisodeBegin()*. Òbviament, per fer un entrenament complet, l'agent haurà de viure milers o milions d'episodis per a que la xarxa neuronal, el model que estem entrenant, aprengui.

Variables i mètodes [Figura 7.22]

Agent		
Nom propietat	**Tipus**	**Descripció**
MaxStep	int	El màxim de steps que pot donar un agent abans de reiniciar-se; o 0 per a steps il·limitats.
CompletedEpisodes	int	Retorna el nombre d'episodis que l'Agent ha completat (o bé s'ha cridat EndEpisode(), o s'ha aconseguit el MaxStep).
StepCount	int	Retorna el comptador d'steps actual (dins de l'episodi actual).

Agent			
Nom mètode	**Retorn**	**Paràmetres**	**Descripció**
CollectObservations	Void	VectorSensor sensor	Recull les observacions vectorials de l'agent per step. Descriu l'entorn actual des de la perspectiva de l'agent. Afegeix observacions vectorials al paràmetre *sensor* cridant als mètodes d'ajuda VectorSensor, explicats a la classe VectorSensor.
EndEpisode	Void	-	Estableix el flag de *done* a true i reinicia l'agent.
Heuristic	Void	ActionBuffers actionsOut	Mètode per a proporcionar una lògica de presa de decisions personalitzada o pel control manual d'un agent amb inputs de Unity. S'assigna un valor de decisió a cada índex de l'array *actionsOut*. El mateix array es reutilitzarà entre steps.
Initialize	Void	-	Realitza una inicialització o configuració única de la instància de l'agent. Es crida una vegada quan l'agent s'activa per primera cop.
OnActionReceived	Void	ActionBuffers actionBuffers	Especificar el comportament de l'agent en cada step, basat en l'acció proporcionada. Les accions poden ser contínues o discretes. S'ha d'especificar quin tipus d'espai d'acció utilitza l'agent, juntament amb la grandària de l'array d'accions, en el component BehaviorParameters associat a l'agent. Quan un agent utilitza l'espai d'acció continu, els valors de la array d'accions són números de punt flotant. S'ha de fixar els valors en el rang [-1,1], per a augmentar l'estabilitat numèrica durant l'entrenament. Quan un agent utilitza l'espai d'accions discretes, els valors de la matriu d'accions són nombres enters que representen una acció específica i discreta.
OnEpisodeBegin	Void	-	Estableix una instància de l'agent al principi d'un episodi.
SetReward	Void	float reward	Reemplaça la recompensa del step actual i actualitza la recompensa de l'episodi en conseqüència.

Figura 7.22: Classe "Agent"

- **Class "DecisionRequester"**
  Component que sol·licita automàticament decisions per a una instància d'Agent a intervals regulars. Proporciona una forma còmoda i flexible d'activar el procés de presa de decisions.

Variables i mètodes [Figura 7.23]

DecisionRequester		
Nom propietat	**Tipus**	**Descripció**
DecisionPeriod	int	La freqüència amb la qual l'agent sol·licita una decisió. Un DecisionPeriod de 5 significa que l'agent sol·licitarà una decisió cada 5 steps.
TakeActionsBetweenDecisions	bool	Indica si l'agent prendrà o no una acció durant els steps en els quals no sol·licita una decisió. No té efecte quan DecisionPeriod s'estableix en 1.

Figura 7.23: Classe "Decision Requester"

Editor a l'Inspector [Figura 7.24]:

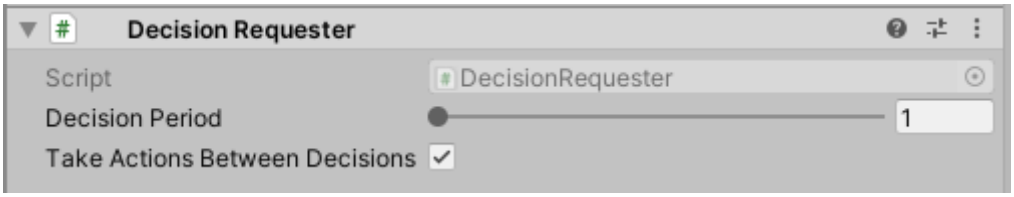


Figura 7.24: Component "Decision Requester"

Unity.MLAgents.Policies

- **Class "BehaviorParameters"**
  En temps d'execució, aquest component genera els objectes de política de l'agent d'acord amb la configuració especificada en l'Editor.

Variables i mètodes [Figura 7.25]

BehaviorParameters		
Nom propietat	**Tipus**	**Descripció**
BehaviorName	string	El nom d'aquest comportament, que s'utilitza com a nom base.
BehaviorType	Enum (int)	El tipus de comportament de l'agent. Descrit en el punt de "Decisions" de la classe Agent.
BrainParameters	BrainParameters	Els BrainParameters associats per a aquest behavior.
InferenceDevice	Enum (string)	Com es realitza la inferència per al model d'aquest agent, per CPU o GPU.
Model	NNModel	El model de xarxa neuronal utilitzat en la manera d'inferència. Quan el valor és nul, s'entrena un.
UseChildSensors	bool	True si s'usen tots els sensors dels components adjunts als GameObjects fills de l'agent.

Figura 7.25: Classe "Behavior Parameters"

Editor al Inspector [Figura 7.26]:

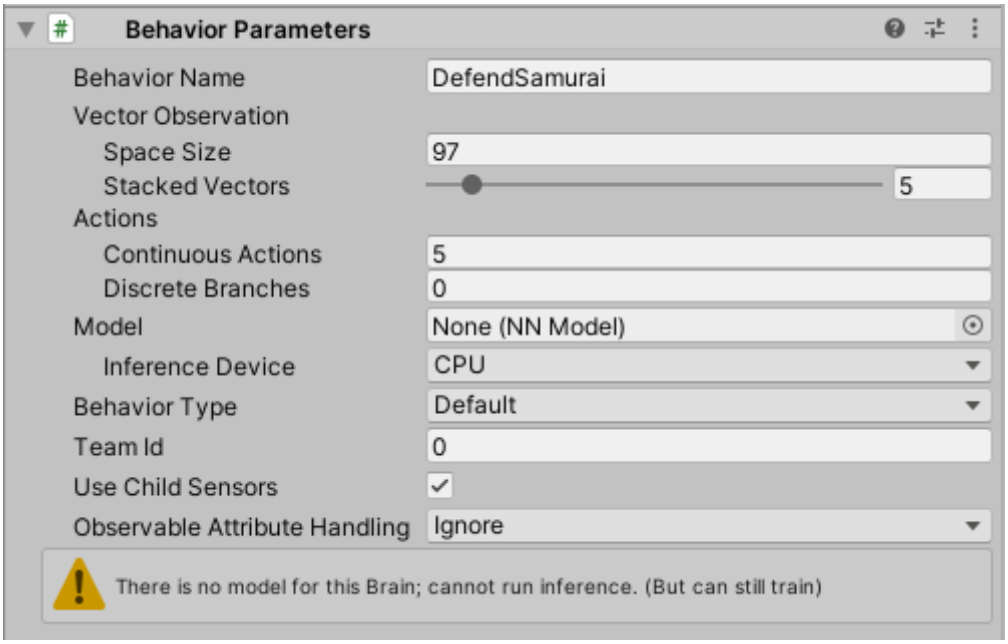


Figura 7.26: Component "Behavior Parameters"

- **Class "Brain Parameters"**
  Defineix quines són les entrades i sortides del procés de decisió.

Variables i mètodes [Figura 7.27]

BrainParameters		
Nom propietat	**Tipus**	**Descripció**
NumStackedVectorObservations	int	L'apilament es refereix a la concatenació de les observacions a través de múltiples fotogrames. Aquest camp indica el nombre de fotogrames a concatenar.
VectorActionSize	int	La grandària de l'espai d'acció. Per a l'espai d'accions contínues: la longitud del vector flotant que representa l'acció. Per a l'espai d'accions discretes: el nombre de branques en l'espai d'accions.
VectorActionSpaceType	Enum (int)	Defineix si l'acció és discreta (0) o contínua (1).
VectorObservationSize	int	La grandària de l'espai d'observació. Un agent crea el vector d'observació en la implementació de CollectObservations(VectorSensor).
NumActions	int	El nombre d'accions especificades.

Figura 7.27: Classe "Brain Parameters"

Unity.MLAgents.Sensors

- **Class "CameraSensor"**
  Un sensor que embolica un objecte Camera per a generar observacions visuals per a un agent.

Variables i mètodes [Figura 7.28]

CameraSensor		
Nom propietat	**Tipus**	**Descripció**
camera	Camera	Objecte Camera des de la qual capturar imatges.
width	int	L'amplada de l'observació visual generada.
height	int	L'alçada de l'observació visual generada.
grayscale	bool	Si s'ha de convertir la imatge generada a escala de grisos o mantenir el color.
name	String	El nom del sensor de la càmera.
compression	SensorCompressionType	La compressió a aplicar a la imatge generada.

Figura 7.28: Classe "Camera Sensor"

SensorCompressionType: Enum int amb valor 0 i 1:

0. **None:** No hi ha compressió. Les dades es conserven com a matrius de floats.
1. **PNG:** Format PNG. Les dades s'emmagatzemaran en format binari.

- **Class "RayPerceptionSensor"**
  Una implementació del sensor que suporta observacions basades en el llençat de raigs.

Variables i mètodes [Figura 7.29]

RayPerceptionSensor		
Nom propietat	**Tipus**	**Descripció**
name	String	Nom del sensor.
rayInput	RayPerceptionInput	Inputs del ray del sensor.

Figura 7.29: Classe "Ray Perception Sensor"

RayPerceptionInput: Struct que conté els elements que defineixen el llençat de raigs.

- **Angles [IReadOnlyList<float>]:** Llista d'angles (en graus) utilitzats per a definir els raigs. 90 graus es consideren "cap endavant" en relació amb l'objecte del joc.
- **CastRadius [float]:** Radio de l'esfera a utilitzar per al spherecasting. Si és 0 o menys, s'utilitzen els raigs en el seu lloc.
- **CastType [Enum (int)]:** Si es realitzen els llançaments en 2D o en 3D. Si val 0, llançament en 2D, utilitzant Physics2D.CircleCast o Physics2D.RayCast. Si val 1, llançament en 3D, utilitzant Physics.SphereCast o Physics.RayCast.
- **DetectableTags [IReadOnlyList<string>]:** Llista de tags que corresponen als tipus d'objectes que l'agent pot veure.
- **EndOffset [float]:** Desplaçament de l'altura final del raig des del centre de l'agent.
- **LayerMask [int]:** Opcions de filtrat pels llançaments.
- **RayLength [float]:** Longitud dels raigs al fer el llençament. Això s'escalarà cap amunt o cap avall en funció de l'escala de la transformació.
- **StartOffset [float]:** Desplaçament inicial de l'alçada del raig des del centre de l'agent.
- **Transform [Transform]:** Transform del GameObject.

- **Class VectorSensor**
  Una implementació de sensors per a observacions vectorials.

Variables i mètodes [Figura 7.30]

VectorSensor		
Nom propietat	**Tipus**	**Descripció**
observationSize	int	Nombre d'observacions del vector.
name	String	Nom del sensor.

Figura 7.30: Classe "Vector Sensor"

Per afegir observacions hi ha el mètode *AddObservation*, el qual permet afegir diferents dades com a observacions. La llista de dades permeses són les següents:

- Quaternion
- Bool
- int
- float
- Vector2
- Vector3

Unity.MLAgents.Actuators

- **Struct ActionBuffer**
  Estructura que embolica les accions contínues i discontínues per a un *IActionReceiver*, una classe interfície que descriu un objecte que pot rebre accions d'una xarxa d'aprenentatge per reforç en particular, i s'utilitza quan es crida la funció OnActionReceived(ActionBuffers). Conté un array de floats, *continuousActions*, o un array de int, *discreteActions*.

7.2.4 Fitxer de configuració .yaml

ML-Agents proporciona una àmplia gamma de mètodes i opcions d'entrenament. Com a tal, execucions específiques poden requerir diferents configuracions d'entrenament i poden generar diferents resultats i estadístiques. Aquesta tasca la dur a terme el **fixter configurador .yaml**. És un arxiu de configuració que conté totes les configuracions i hiperparámetres que s'utilitzaran durant l'entrenament. El conjunt de configuracions i hiperparámetres a incloure en aquest arxiu depèn dels agents de l'entorn i del mètode d'entrenament específic que es desitgi utilitzar.

Quin tipus de fitxer és l'extensió .yaml?

YAML és un format de serialització de dades, és a dir, un format per a emmagatzemar un objecte mitjançant la seva codificació. Al tenir un bon suport en Python, llenguatge en el que està programat ML-Agents, els creadors van decidir que les dades de configuració es farien en un fitxer d'aquest tipus.

Algorisme d'aprenentatge per reforç

Una de les primeres decisions que s'ha de prendre respecte a l'execució d'entrenament és quin algorisme d'aprenentatge per reforç s'utilitzarà: PPO (Proximal Policy Optimization) o SAC (Soft Actor Critic). L'algorisme per defecte és PPO. Aquest és un mètode on-policy que ha demostrat ser més general i estable que molts altres algorismes d'aprenentatge per reforç. A diferència del PPO, SAC és off-policy, cosa que significa que pot aprendre de les experiències recollides en qualsevol moment del passat. A mesura que es recullen les experiències, es col·loquen en un búfer de repetició d'experiències i s'extreuen aleatòriament durant l'entrenament. Ambdós, tot i així, es poden utilitzar per a entorns amb espais d'acció discrets o continus.

Hi ha algunes configuracions d'entrenament que són comuns a tots dos i altres que depenen de l'elecció de l'algoritme.

A les següents pàgines s'expliquen aquestes dades comunes i les particulars del PPO. En aquesta memòria no s'exposen les del SAC perquè no es realitzen entrenaments amb aquest algortime. Si es vol saber quines són s'ha de visitar el GitHub de ML-Agents i anar dins de l'apartat de "docs", i posteriorment a l'apartat de "Training Configuration File"[1].

[1] https://github.com/Unity-Technologies/ml-agents/blob/4feccd22f466b1affd5bf9e8050f1ff2b54a62da/docs/Training-Configuration-File.md

Configuracions comunes
Llista de dades de configuració comunes:

- ***trainer_type*** -> (per defecte = *ppo*)
  El tipus d'algorisme d'aprenentatge per reforç a utilitzar: *ppo* o *sac*

- ***summary_freq*** -> (per defecte = *50000*)
  Nombre d'experiències que s'han de recollir abans de generar i mostrar estadístiques d'entrenament.

- ***time_horizon*** -> (per defecte = *64*)
  Quants steps s'han de recollir per agent abans d'afegir-lo al buffer d'experiència. Quan s'arriba a aquest límit abans de finalitzar un episodi, s'utilitza una estimació de valor per predir la recompensa esperada global a partir de l'estat actual de l'agent. En els casos en què hi hagi recompenses freqüents dins d'un episodi o els episodis siguin prohibitius, pot ser més ideal un nombre menor. Aquest nombre ha de ser prou gran com per captar tot el comportament important dins d'una seqüència d'accions d'un agent.

  Rang típic: *32 - 2048*

- ***max_steps*** -> (per defecte = *500000*)
  Nombre total de steps (és a dir, observació recollida i acció realitzada) que s'han de fer a l'entorn (o en tots els entorns si s'utilitzen múltiples en paral·lel) abans de finalitzar el procés d'entrenament. Si es té diversos agents amb el mateix nom de comportament a l'entorn, tots els passos que han fet aquests agents contribuiran al mateix max_steps.

  Rang típic: *500000 - 10000000*

- ***keep_checkpoints*** -> (per defecte = *5*)
  Nombre màxim de punts de control del model. Els punts de control guarden el model després del nombre de passos especificat per l'opció checkpoint_interval. Un cop assolit el nombre màxim de punts de control, se suprimeix el punt de control més antic quan es desa un punt de control nou.

- ***checkpoint_interval*** -> (per defecte = *500000*)
  El nombre d'experiències recollides entre cada punt de control. Cada punt de control desa l'arxiu .onnx a la carpeta "results/".

- ***init_path*** -> (per defecte = *cap*)
  Inicialitzar a partir d'un model prèviament guardat. S'ha de tenir en compte que l'execució anterior ha d'haver utilitzat les mateixes configuracions d'entrenador que l'execució actual, i haver estat guardada amb la mateixa versió de ML-Agents.

  S'ha de proporcionar la ruta completa a la carpeta on es van guardar els punts de control, per exemple, "./models/run-id/behavior_name". Aquesta opció es proporciona en cas que es desitgi inicialitzar diferents Behaviors de diferents execucions.

- ***threaded*** -> (per defecte = *true*)
  Per defecte, les actualitzacions del model es poden produir mentre es fa un step. Això vulnera lleugerament l'assumpció de la política de PPO a canvi d'una acceleració de l'entrenament. Per mantenir l'estricta política de PPO, es pot desactivar les actualitzacions paral·leles configurant-ho amb *false*. Pel que fa el SAC, no hi ha cap motiu per desactivar-ho.

Hiperparàmetres:

- ***learning_rate*** -> (per defecte = *0.0003*)
  Taxa d'aprenentatge inicial per al descens de gradient (estima numèricament on una funció genera els seus valors més baixos) al mètode d'optimització utilitzat. Correspon a la força de cada pas d'actualització del descens de gradient. Normalment hauria de reduir-se si l'entrenament és inestable i la recompensa no augmenta de manera constant.

  Rang típic: *0.00001 - 0.001*

- ***batch_size***
  Nombre d'experiències en cada iteració del descens de gradient. Sempre ha de ser diverses vegades menor que buffer_size. Si s'utilitza un espai d'acció continu, aquest valor ha de ser gran. Si s'utilitza un espai d'acció discret, aquest valor ha de ser menor.

  Rang típic: (Continu - PPO): *512 - 5120*; (Continu - SAC): *128 - 1024*; (Discret, PPO i SAC): *32 - 512*.

- ***buffer_size*** -> (per defecte = *10240* per a PPO i *50000* per a SAC)
  PPO: Nombre d'experiències a recollir abans d'actualitzar el model de polítiques. Correspon a quantes experiències ha de recollir abans de realitzar qualsevol aprenentatge o actualització del model. Ha de ser diverses vegades major que batch_size. Normalment, un buffer_size major correspon a actualitzacions d'entrenament més estables.
  SAC: La grandària màxima del buffer d'experiències - de l'ordre de milers de vegades major que els episodis, perquè el SAC pugui aprendre tant de les experiències antigues com de les noves.

  Rang típic: PPO: *2048 - 409600*; SAC: *50000 - 1000000*

- ***learning_rate_schedule*** -> (per defecte = *linear* per a PPO i *constant* per a SAC)
  Determina com canvia la taxa d'aprenentatge amb el temps. Per a PPO, es recomana que la taxa d'aprenentatge decaigui fins a max_steps perquè l'aprenentatge convergeixi de forma més estable. No obstant això, per a alguns casos (per exemple, l'entrenament durant un temps desconegut) aquesta característica es pot desactivar. Per al SAC, es recomana mantenir la taxa d'aprenentatge constant perquè l'agent pugui continuar aprenent fins que la seva funció convergeixi de manera natural.

  Linear fa caure la taxa d'aprenentatge linealment, arribant a 0 en max_steps, mentre que *constant* manté la taxa d'aprenentatge constant per a tota l'execució de l'entrenament.

Ajustos de la xarxa neuronal:

- ***hidden_units*** -> (per defecte = *128*)
  Nombre d'unitats en les capes ocultes de la xarxa neuronal. Correspon a quantes unitats hi ha en cada capa totalment connectada de la xarxa neuronal. Per a problemes senzills en els quals l'acció correcta és una combinació directa de les entrades d'observació, ha de ser petit. Per als problemes en els quals l'acció és una interacció molt complexa entre les variables d'observació, ha de ser major.

  Rang típic: *32 - 512*

- ***num_layers*** -> (per defecte = *2*)
  El nombre de capes ocultes de la xarxa neuronal. Correspon a quantes capes ocultes hi ha després de l'entrada de l'observació, o després de la codificació de l'observació visual. Per a problemes senzills, és probable que un menor nombre de capes permeti un entrenament més ràpid i eficient. Per a problemes més complexos de control poden ser necessàries més capes.

  Rang típic: *1 - 3*

- ***normalize*** -> (per defecte = *false*)
  Si s'aplica una normalització a les entrades de l'observació vectorial. Aquesta normalització es basa en la mitjana i la variància del vector d'observació. La normalització pot ser útil en casos amb problemes complexos de control continu, però pot ser perjudicial amb problemes més simples de control discret.

Senyals de recompensa
Cada senyal de recompensa ha de definir almenys dos paràmetres, *strength* i *gamma*.

- ***strength*** -> (per defecte = *1*)
  Factor pel qual es multiplica la recompensa donada per l'entorn. Els rangs típics variaran en funció del senyal de recompensa.

- ***gamma*** -> (per defecte = *0.99*)
  Factor de descompte per a les recompenses futures procedents de l'entorn. Pot considerar-se com la distància a la qual l'agent ha de preocupar-se per les possibles recompenses en el futur. En situacions en les quals l'agent ha d'actuar en el present per a preparar-se per a les recompenses en un futur llunyà, aquest valor ha de ser gran. En els casos en què les recompenses són més immediates, pot ser menor. Ha de ser estrictament menor que 1.

  Rang típic: *0.8 - 0.995*

Configuracions específiques de PPO

- ***beta*** -> (per defecte = *0.005*)
  Força de la regularització de l'entropia, que fa que la política sigui "més aleatòria". Això assegura que els agents explorin adequadament l'espai d'acció durant l'entrenament. Augmentar això assegurarà que es realitzin més accions aleatòries. Ha d'ajustar-se de manera que l'entropia disminueixi lentament juntament amb els augments de la recompensa. Si l'entropia disminueix massa ràpid, augmenta la beta. Si l'entropia disminueix massa lentament, disminueixi beta.

  Rang típic: *0.0001 - 0.01*

- ***epsilon*** -> (per defecte = *0.2*)
  Influeix en la rapidesa amb la qual la política pot evolucionar durant l'entrenament. Correspon al llindar acceptable de divergència entre la política antiga i la nova durant l'actualització per descens de gradient. Si aquest valor és petit, les actualitzacions seran més estables, però també s'alentirà el procés d'entrenament.

  Rang típic: *0.1 - 0.3*

- ***lamd*** -> (per defecte = *0.95*)
  Això pot considerar-se com la mesura en que l'agent es basa en l'estimació de valor actual en calcular una estimació de valor actualitzada. Els valors baixos corresponen a una major confiança en l'estimació del valor actual (el que pot suposar un alt biaix), i els valors alts corresponen a una major confiança en les recompenses reals rebudes en l'entorn (el que pot suposar una alta variància). El paràmetre proporciona un equilibri entre tots dos, i el valor correcte pot conduir a un procés d'entrenament més estable.

  Rang típic: *0.9 - 0.95*

- ***num_epoch*** -> (per defecte = *3*)
  Número de passades a realitzar a través del buffer d'experiència quan es realitza l'optimització per descens de gradient. Com més gran sigui el batch_size, major serà el número de passades a realitzar. Disminuir això assegurarà actualitzacions més estables, a costa d'un aprenentatge més lent.

  Rang típic: *3 - 10*

7.2.5 Python

Python (veure logotip Figura 7.31) és un llenguatge de programació interpretat, multiparadigma i multiplataforma concebut a la fi de la dècada de 1980 per Guido van Rossum en el Centrum Wiskunde & Informatica (CWI) dels Països Baixos com a successor del llenguatge de programació ABC, inspirat en el SETL.

Figura 7.31: Logotip Python

La Ptyhon API i els algoritmes d'entrenament de ML-Agents estan escrits en aquest llenguatge. Tot i que no es toqui en cap part durant aquest projecte, és important comentar-ho, ja que la dificultat hagués estat molt més elevada si s'hagués hagut de realitzar algun wrap entre Unity i alguna llibreria de deep learning, les quals, en la gran majoria, están escrites en Python.

7.2.6 PyTorch

PyTorch (veure logotip Figura 7.32) és una biblioteca de codi obert basat en Python per a realitzar càlculs utilitzant gràfics de flux de dades, la representació subjacent dels models d'aprenentatge profund. Facilita l'entrenament i la inferència en CPUs i GPUs en un escriptori d'ordinador, servidor o dispositiu mòbil.

Figura 7.32: Logotip PyTorch

Disposa d'una interfície en Python molt senzilla per a la creació de xarxes neuronals sense la necessitat d'una llibreria d'alt nivell. Al contrari que altres entorns com Tensorflow, PyTorch treballa amb grafs dinàmics en comptes d'estàtics. Això significa que en temps d'execució es poden anar modificant les funcions i el càlcul del gradient variarà amb elles.

Dins del ML-Agents Toolkit, quan s'entrena el comportament d'un agent, la sortida és un arxiu de model (.onnx) que després es pot associar a un agent. Tret que s'implementi un nou algorisme, l'ús de PyTorch està, en la seva majoria, abstret i entre bastidors.

7.2.7 TensorBoard

El ML-Agents Toolkit guarda estadístiques durant la sessió d'aprenentatge que es poden veure amb una utilitat de TensorFlow anomenada *TensorBoard* (Figura 7.33).

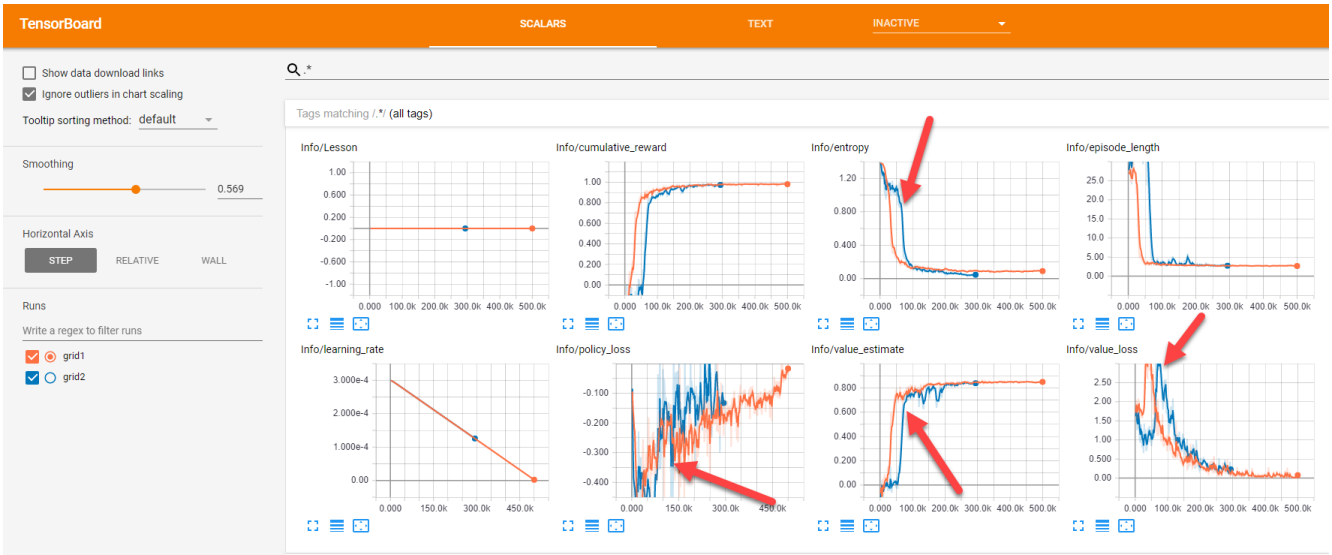


Figura 7.33: Exemple de la representació amb TensorBoard

Les dades que mostre són les següents:

- **Environment/Cumulative Reward:** La recompensa mitjana acumulada de l'episodi sobre tots els agents. Ha d'augmentar durant una sessió d'entrenament reeixida.
- **Environment/Episode Length:** La durada mitjana de cada episodi en l'entorn per a tots els agents.
- **Policy/Entropy:** El grau d'aleatorietat de les decisions del model. Ha de disminuir lentament durant un procés d'entrenament reeixit. Si disminueix massa ràpid, ha d'augmentar-se el hiperparèmetre beta.
- **Policy/Learning Rate:** La grandària del step que dóna l'algorisme d'entrenament mentre busca la política òptima. Ha de disminuir amb el temps.
- **Policy/Extrinsic Reward:** Recompensa mitjana acumulada rebuda de l'entorn per episodi.
- **Policy/Value Estimate:** L'estimació del valor mitjà de tots els estats visitats per l'agent. Ha d'augmentar durant una sessió d'entrenament reeixida.

- **Losses/Policy Loss:** La magnitud mitjana de la funció de pèrdua de la política. Es relaciona amb el grau de canvi de la política (procés de decisió de les accions). La seva magnitud hauria de disminuir durant una sessió d'entrenament reeixida.
- **Losses/Value Loss**: La pèrdua mitjana de l'actualització de la funció de valor. Es correlaciona amb que tan bé el model és capaç de predir el valor de cada estat. Hauria d'augmentar mentre l'agent està aprenent, i després disminuir una vegada que la recompensa s'estabilitza.

7.3 Draw.io

Draw.io (logotip Figura 7.34) és un programari gratuït de diagrames en línia per a fer diagrames de flux, diagrames de procés, organigrames, UML, ER i diagrames de xarxa.

Figura 7.34: Logotip Draw.io

La major part dels diagrames d'aquesta memòria estan realitzats amb aquest programa, ja que és molt intuïtiu, ràpid i, el més important, és gratuït.

7.4 Venngage

Venngage (logotip Figura 7.35) es un programari de disseny gràfic per a crear cartells, informes, infografies i presentacions.

Figura 7.35: Logotip Venngage

Usat per crear els diagrames de gantt de l'Apartat 3: Metodologia, per les mateixes raons pel que s'ha utilitzat el programa Draw.io.

8 Anàlisi i disseny del sistema

8.1 Diagrama de cas d'ús

Hi ha un únic actor: l'usuari. Aquest pot arribar a fer una quantitat limitada d'accions, les quals permeten modificar en gran mesura com s'entrena l'agent i provar el funcionament del resultat. Això es a causa que es proporcionarà el projecte de Unity, el qual permetrà l'edició de la configuració per defecte de l'entrenament. A més a més, com entrenar pot ser costós en temps, l'usuari té a l'abast una petita demo on es pot observar el resultat d'un entrenament complet. Un entrenament està conformat per un conjunt d'animacions de l'atacant i un fitxer de configuració de l'entrenament.

En el diagrama següent, la Figura 8.1, es pot observar la relació entre l'actor i els casos d'ús, i entre casos d'ús. En aquest podem observar la distinció entre tres sistemes: Un executable de Unity amb la demo de la intel·ligència artificial ja entrenada, el projecte de Unity amb tot el codi i el sistema operatiu (terminal i explorador d'arxius) amb el qual fem la connexió entre Unity i els algoritmes de deep learning a la API de baix nivell de Python (explicat a l'Apartat 7: Estudis i decisions, a la Secció 7.2: ML-Agents i Subsecció 7.2.2: Funcionament).

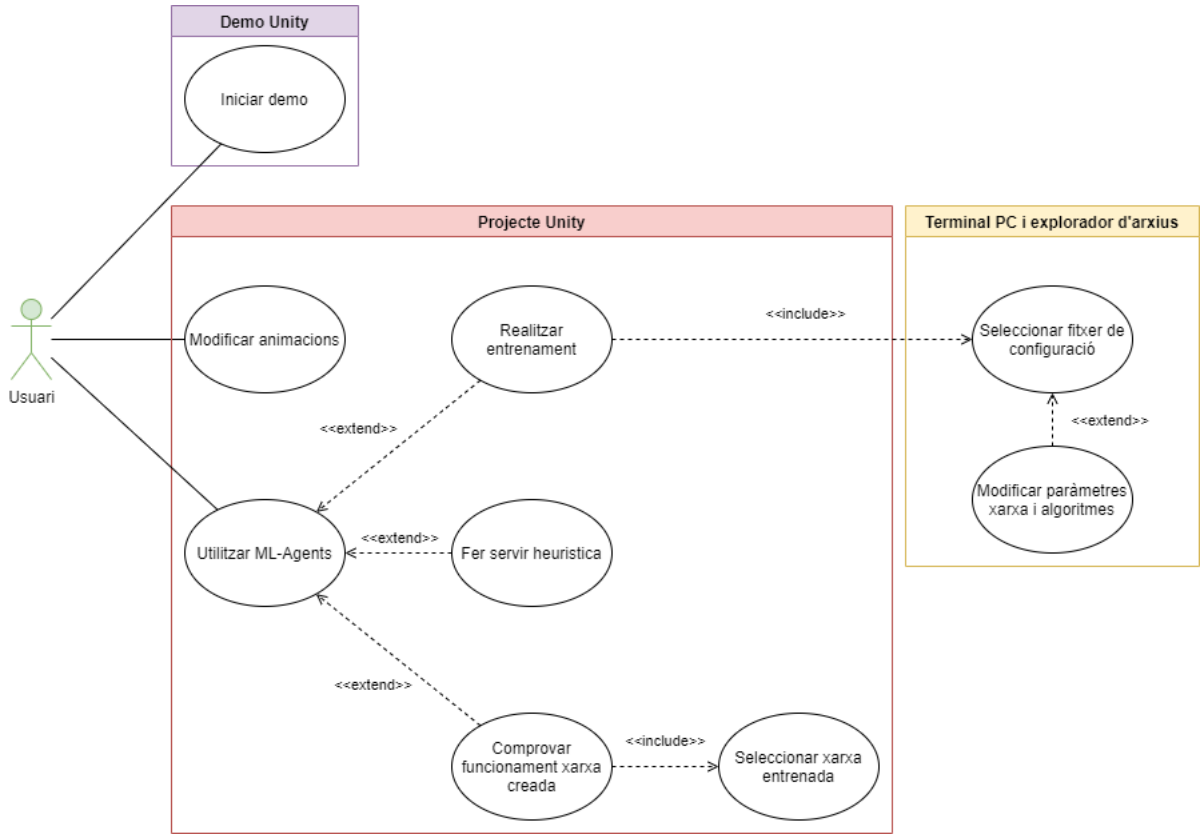


Figura 8.1: Diagrama de casos d'ús de les funcionalitats principals del sistema

8.2 Fitxes de cas d'us

Cas d'ús:	Iniciar demo
Descripció	Executar la *build* de Unity d'un entrenament estès d'un agent amb unes certes animacions i configuració.
Actors	Usuari
Precondició	-
Flux principal	1. Executar l'aplicació anomenada "Demo.exe"
Flux alternatiu	-
Postcondició	La demo s'ha iniciat i, seguint els tutorials, es pot provar diferents animacions per veure com l'agent es defensa.

Figura 8.2: Fitxa cas d'ús "Inciar Demo"

Cas d'ús:	Modificar animacions
Descripció	Canviar una (o més) animació del atacant per una de nova.
Actors	Usuari
Precondició	Tenir les animacions del atacant obtingudes de Mixamo en una carpeta del projecta de Unity
Flux principal	1. Obrir l'animator "Attack controller". 2. Seleccionar qualsevol estat entre "Attack1" fins "Attack15". 3. Assignar a l'atribut "Motion" la variable "Animation clip" de l'animació nova.
Flux alternatiu	-
Postcondició	Animació canviada d'un estat del controlador d'animacions del personatge atacant.

Figura 8.3: Fitxa cas d'ús "Modificar Animacions"

Cas d'ús:	Utilitzar ML-Agents
Descripció	Utilitzar l'apartat d'ML-Agents del projecte de Unity
Actors	Usuari
Precondició	Tenir instal·lat Python (3.6.1 o superior), el paquet de Unity "com.unity.ml-agents" i el paquet Python "mlagents".
Flux principal	1. Obrir el projecte de Unity anomenat "AnimationTFG". 2. Seleccionar l'escena "Training Scene". 3. Seleccionar el GameObject anomenat "SwordDefender". 4. Si es vol entrenar l'agent: 4.1. Realitzar entrenament 5. Altrament si es vol comprovar el funcionament amb l'heurística: 5.1. Fer servir heurística 6. Altrament si es vol veure els resultats d'una xarxa: 6.1. Comprovar funcionament xarxa creada
Flux alternatiu	-
Postcondició	S'ha inicialitzat el procés d'aprenentatge amb resultats diferents depenent del que s'hagi escollit.

Figura 8.4: Fitxa cas d'ús "Utilitzar ML-Agents"

Cas d'ús:	Realitzar entrenament
Descripció	Inicialitzar un entrenament d'un agent
Actors	Usuari
Precondició	Instal·lació correcta de ML-Agents, projecte de Unity obert a l'escena "Training scene" i existeix un arxiu YAML correcte.
Flux principal	1. Seleccionar fitxer de configuració en un terminal 2. Quan aparegui en el terminal el missatge "Start training by pressing the Play button in the Unity Editor", prémer el botó Play en Unity per a iniciar l'entrenament en l'Editor.
Flux alternatiu	1. Prémer les tecles Ctrl + C en el terminal per pausar l'entrenament.
Postcondició	S'inicialitza l'entrenament sota unes condicions i finalitza després del màxim d'steps indicat al arxiu YAML o per una pausa de l'usuari.

Cas d'ús:	Seleccionar fitxer de configuració
Descripció	Selecciona un fitxer YAML de configuració per a dur a terme un entrenament, establint la connexió entre l'entrenador i l'entorn de Unity.
Actors	Usuari
Precondició	Instal·lació correcta de ML-Agents, projecte de Unity obert a l'escena "Training scene" i existeix un arxiu YAML correcte.
Flux principal	1. Obrir una finestra de comandos o terminal. 2. Executar la comanda: "mlagents-learn <trainer-config-file> --run-id=<run-identifier>", on trainer-config-file és la ruta de l'arxiu YAML de configuració, i run-identifier és el nom únic que s'utilitza per a identificar el resultat.
Flux alternatiu	-
Postcondició	S'ha seleccionat un fitxer YAML com a configuració de l'entrenament i s'ha aconseguit inciar la connexió entre Unity i l'entrenador de Python.

Cas d'ús:	Modificar paràmetres xarxa i algoritmes
Descripció	Canvia algun paràmetre de configuració d'un fitxer YAML.
Actors	Usuari
Precondició	Existeix un fitxer YAML correcta i el valor introduït ha de ser vàlid pel paràmetre que es vol cambiar.
Flux principal	1. Obrir fitxer amb un edito de text (notepad, notepad++...) 2. Buscar paràmetre o paràmetres a canviar 3. Escriure en la posició correcta el nou valor del paràmetre 4. Guardar fitxer
Flux alternatiu	-
Postcondició	S'ha editat un fitxer de configuració YAML

Figura 8.5: Fitxes casos d'ús "Realitzar entrenament"

Cas d'ús:	Fer servir heurística
Descripció	A l'editor del projecte de Unity, fer servir l'heurística per provar el funcionament de les accions que podrà dur a terme l'agent.
Actors	Usuari
Precondició	Projecte de Unity obert, cap terminal obert amb la comanda d'entrenament executada i cap xarxa neuronal assignada a la variable "Model" de l'agent.
Flux principal	1. Seleccionar l'escena "Training Scene". 2. Prémer el botó Play en Unity per a iniciar l'entrenament en la finestra de joc. 3. Per finalitzar, tornar a prémer el botó Play en Unity.
Flux alternatiu	-
Postcondició	S'ha pogut comprovar el funcionament de les accions mitjançant l'heurística.

Figura 8.6: Fitxa cas d'ús "Fer servir heurística"

Cas d'ús:	Comprovar funcionament xarxa creada
Descripció	Visualització del comportament d'una xarxa neuronal creada a partir de l'entrenament de l'agent.
Actors	Usuari
Precondició	Tenir una xarxa neuronal (.onn) compatible amb les animacions d'atac que realitzarà l'atacant. Projecte de Unity obert.
Flux principal	1. Seleccionar l'escena "Demo Scene". 2. Seleccionar el GameObject anomenat "SwordDefender". 3. Seleccionar xarxa entrenada 4. Prémer el botó Play en Unity per a iniciar l'entrenament en la finestra de Joc. 5. Per finalitzar, tornar a prémer el botó Play en Unity.
Flux alternatiu	-
Postcondició	S'ha pogut visualitzar el funcionament d'una xarxa neuronal entrenada.

Cas d'ús:	Seleccionar xarxa entrenada
Descripció	Assigna a l'agent una xarxa neuronal que ha estat entrenada per a les animacions d'atac de l'atacant.
Actors	Usuari
Precondició	Tenir una xarxa neuronal (.onn) compatible amb les animacions d'atac que realitzarà l'atacant. Projecte de Unity obert. GameObject "SwordDefender" seleccionat.
Flux principal	1. Assignar a l'atribut "Model" una xarxa neuronal compatible.
Flux alternatiu	-
Postcondició	L'atribut "Model" de l'agent té assignat una xarxa neuronal compatible amb l'entorn d'entrenament.

Figura 8.7: Fitxes casos d'ús "Comprovar funcionament xarxa creada"

8.3 Diagrama de classes

La solució al nostre problema ve donada per la utilització d'un programari ja existent, Unity i ML-Agents. Per aquesta raó, el diagrama de classes resultant alhora de l'anàlisi en un diagrama de casos d'ús i les seves respectives fitxes està format per les classes derivades d'aquest programa. A més a més, l'API de Python de ML-Agents la considerarem una "caixa negra", és a dir, una classe que rep les dades del entorn de Unity i retorna les accions a realitzar per l'agent, i, finalment, una xarxa neuronal a utilitzar per l'animació de defensa. Aquesta consideració ha sorgit a raó de que el funcionament va més enllà del que tracta aquest projecte, ja que les classes d'aquesta API, a part de parlar de la connexió i transmissió de dades entre ella i Unity, també tracta en gran mesura la implementació dels algoritmes d'aprenentatge per reforç i les xarxes neuronals.

Per tant, el diagrama de classes resultant és el següent, Figura 8.8, mostrant les noves classes derivades al projecte de Unity i la connexió amb la API de Python.

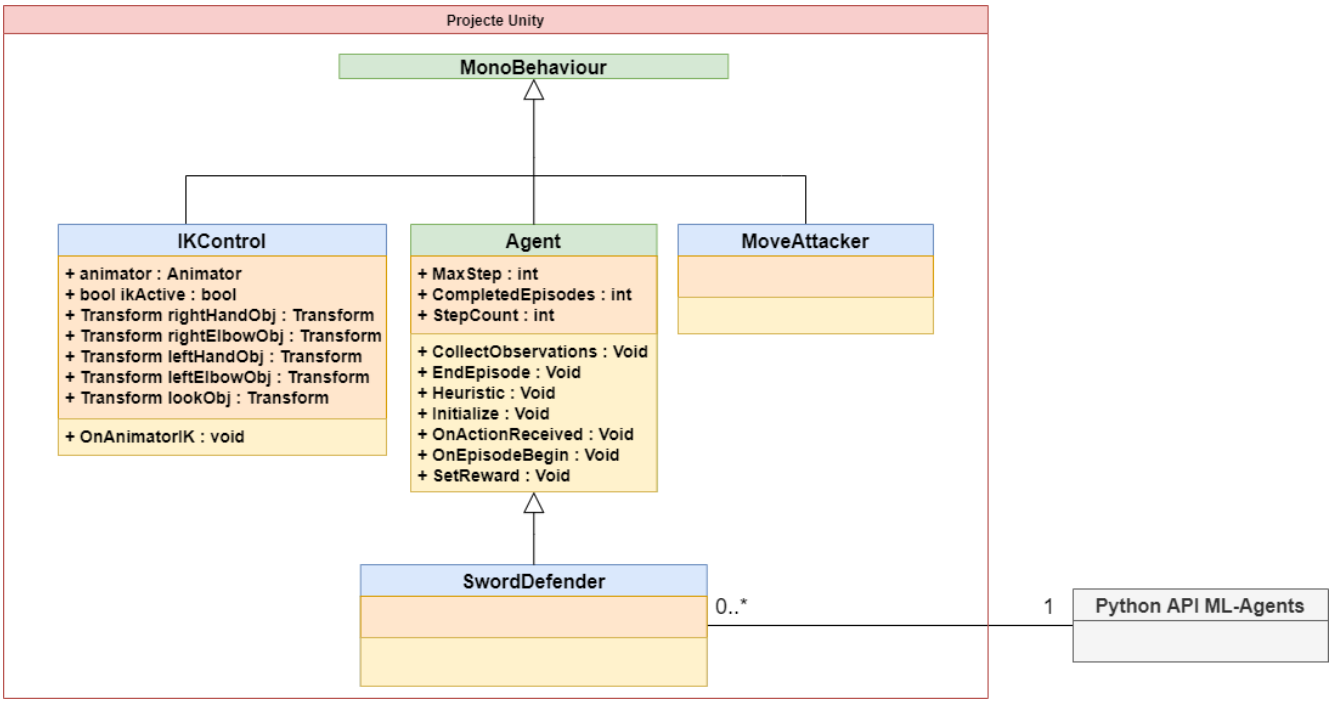


Figura 8.8: Diagrama de classes del projecte

A continuació es farà una breu explicació de les classes:

- **MonoBehaviour:** Classe pare implementada a Unity. Veure Apartat 7: Estudis i decisions, Secció 7.1: Unity, Subsecció 7.1.4: Classes.
- **Agent:** Classe agent pare d'on deriven tots els agents d'un entorn. Veure Apartat 7: Estudis i decisions, Secció 7.2: ML-Agents, Subsecció 7.2.3: Classes.

- **IKControl:** Classe que implementa el funcionament de la cinemàtica inversa sobre el personatge defensor i la seva espasa, l'agent.
  Variables:
  - **public Animator animator**
    Animador del personatge defensor, el qual emmagatzema el Animator Controller i l'avatar.
  - **public bool ikActive**
    Variable de control per activar i desactivar el funcionament de la cinemàtica inversa.
  - **public Transform rightHandObj**
    Transform de l'objectiu de la mà dreta.
  - **public Transform leftHandObj**
    Transform de l'objectiu de la mà esquerra.
  - **public Transform rightElbowObj**
    Transform de l'objectiu parcial del colze dret.
  - **public Transform leftElbowObj**
    Transform de l'objectiu parcial del colze esquerra.
  - **public Transform lookObj**
    Transform de l'objectiu del cap, és a dir, cap a on mirarà el personatge.

  Mètodes:
  - **void OnAnimationIK()**
    Implementació del callback que es cridat just abans d'actualitzar les posicions del sistema de cinemàtica inversa. És utilitzat per a establir els objectius IK i els pesos respectius. La següent Figura 8.9 mostra un exemple de les funcions que ho fan possible.

```
void OnAnimatorIK(int layerIndex)
{
    animator.SetIKPositionWeight(AvatarIKGoal.LeftFoot, leftFootPositionWeight);
    animator.SetIKRotationWeight(AvatarIKGoal.LeftFoot, leftFootRotationWeight);
    animator.SetIKPosition(AvatarIKGoal.LeftFoot, leftFootObj.position);
    animator.SetIKRotation(AvatarIKGoal.LeftFoot, leftFootObj.rotation);
}
```

Figura 8.9: Exemple

- **MoveAttacker:** Classe que s'encarregarà de controlar el moviment de l'atacant.
- **SwordDefender:** Classe de tipus "Agent" que s'encarregarà de tot la lògica de l'agent (accions, observacions, etc).
- **Python API ML-Agents:** Classe "caixa negra" que conté totes les dades i algoritmes de l'aprenentatge per reforç i deep learning (xarxa neuronal, polítiques, recompenses, etc). Rep les dades necessàries dels agents de Unity i retorna accions.

Les classes "MoveAttacker" i "SwordDefender" varien en variables i mètodes depenent del estadi de desenvolupament. Recordar que la idea de desenvolupament és per estadis, és a dir, començar amb entorns senzills (agents poc complexos i accions simplificades) fins arribar a l'entorn amb els dos personatges i les seves respectives espases.

8.4 Interfície d'usuari

La interfície del projecte de Unity és la que proporciona el propi motor, ja que treballem directament amb ell. En canvi, la demo, n'és una de creada, la qual està explicada a l'Apartat 10.2: Demo.

9 Implementació i proves

Com ja s'ha mencionat amb anterioritat, aquest projecte ha estat desenvolupat primerament en escenaris simples. En total s'han dut a terme 3 escenaris diferents, cadascun d'ells amb variacions destinades a arribar a un últim, on tenim als dos personatges combatent.

A continuació, s'explicarà i mostrarà tots els escenaris d'entrenament fins arribar al més avançat.

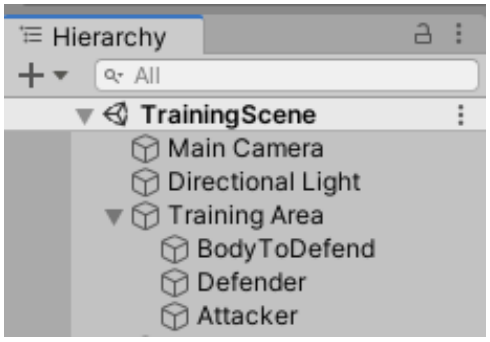
9.1 Primera implementació: Cubs i cilindres - Moviment en un eix

Per començar, es va idear un escenari senzill on l'agent, al no tenir encara el modelat de l'espasa, seria un cilindre, l'objectiu a defendre seria un cub i l'espasa enemiga un altre cilindre. Aquest últim es mourà mitjançant un rigidBody, que mitjançant el motor de física de Unity es mourà en direcció al cub.

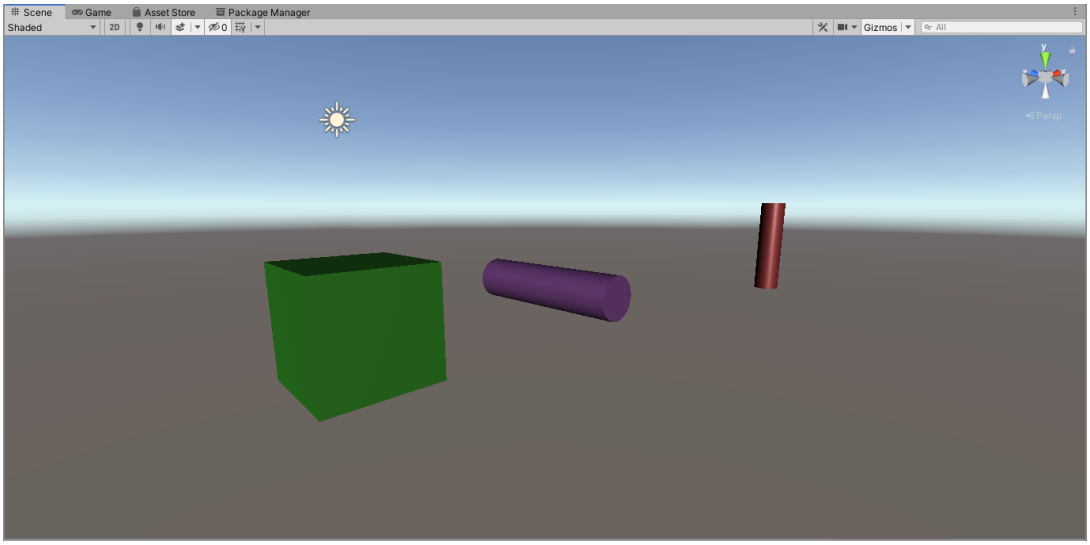
A la Figura 9.1b podem veure un cub verd (anomenat BodyToDefend, amb un Tag = Goal) com a objectiu del atacant, un cilindre morat (anomenat Defender, amb un Tag = Defender) com a defensor del BodyToDefend, i un cilindre vermell (anomenat Attacker) com a atacant de l'objectiu.

La jerarquia la podem analitzar a la Figura 9.1a, on veiem que tots tres objectes son independents i fills d'un GameObject Empty anomenat Training Area. Aquest té el Transform amb rotació (0,0,0) i escala (1,1,1). La seva existència compleix la funció d'independitzar les dades d'observació de l'agent del eix global de l'escena. Així, si modifiquem la posició/rotació/escala del grup, com que les dades són respecte aquest eix, el seu nou eix de referència, el resultat de les accions a la xarxa neuronal sempre seran les mateixes. Això permet que, a l'hora de realitzar els entrenaments, per a accelerar-lo, es pugin multiplicar els Training Area amb tots els fills GameObject.

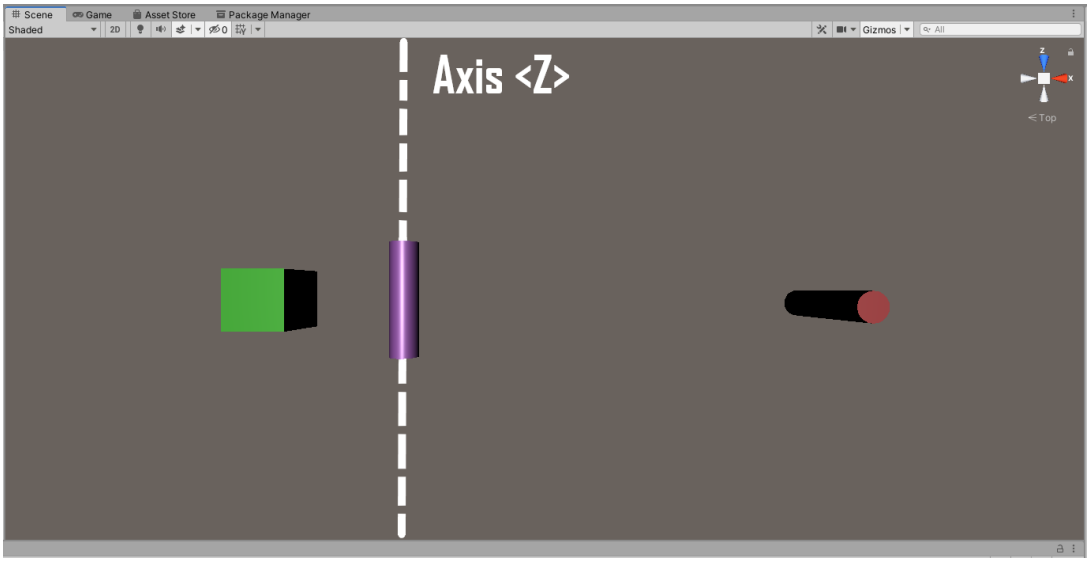
Darrerament, a la Figura 9.1c, està representat amb una línia blanca l'eix pel qual el cilindre defensor es podrà moure.



(a) Jerarquía dels GameObjects



(b) Visió de la finestra "Scene"



(c) Respresentació del moviment possible de l'agent

Figura 9.1: Escena Unity "Training Scene" de la primera implementació

9.1.1 Agent

Namespaces

La classe de l'espasa defensora ("SwordDefender") necessitarà sobreescriure certes funcions tant del propi Unity com de ML-Agents. Per tant, farem servir els *namespaces* de UnityEngine per a fer referencia a les classes de Unity, i Unity.MLAgents, Unity.MLAgents.Sensors i Unity.MLAgents.Actuators pels mètodes de ML-Agents.

A més a més, com s'ha explicat al diagrama de classe de l'Apartat 8: Anàlisi i disseny del sistema, és hereu de la classe Agent. Per indicar això, a l'hora de inicialitzar la classe ho indiquem de la següent manera:

```
public class SwordDefender : Agent
```

Variables

En aquesta primera prova, té les següents variables:

- **public Transform attacker**
  Transform del cilindre atacant. Referenciat mitjançant l'Inspector de Unity.
- **public Transform goal**
  Transform del cub, l'objectiu a protegir de l'agent i d'atacar el cilindre atacant. Referenciat mitjançant l'Inspector de Unity.
- **public float forceMultiplier**
  Número real que multiplica la força aplicada al moviment de l'agent. Valor referenciat mitjançant l'Inspector de Unity.
- **private Rigidbody rBody**
  RigidBody del cilindre defensor.

Fent servir el mètode *Start()* de Unity assignem a la variable rBody el rigidBody del cilindre defensor. Com que aquesta classe/script formarà part del GameObject (com si fos un component) podem fer l'assignació al concepte de *this* de forma implícita, que fa referència al cilindre en la instància actual de la classe. Per tant, el mètode queda de la següent manera (Figura 9.2):

```
// Start is called before the first frame update
Mensaje de Unity | 0 referencias
void Start()
{
    rBody = GetComponent<Rigidbody>();
}
```

Figura 9.2: Primera prova - Mètode *Start()* a la classe "SwordDefender"

Components del GameObject

La figura 9.3 mostra els components que té l'agent.

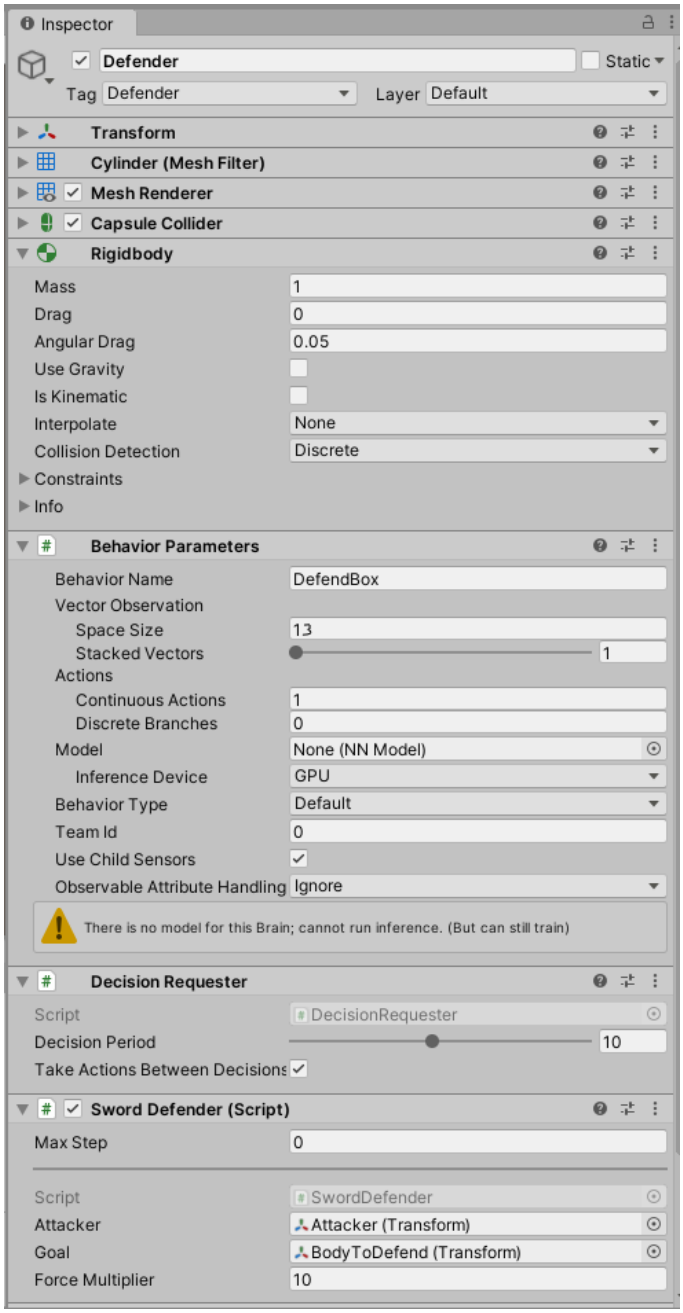


Figura 9.3: Primera prova - Inspector del GameObject "Defender"

En el component *BehaviorParameters* s'ha assignat un nom al comportament, DefendBox, el vector d'observació s'ha establit en 13 variables (sumatori del nombre de variables que s'indicarà en el mètode *CollectObservations(VectorSensor)*), només apilarem 1 vector d'observacions per FixedUpdate (Stacked Vector) i marquem que només podrà fer una acció continua.

Pel que fa el *DecisionRequester*, hi ha assignat que cada 10 steps es demani una decisió (Descision Period), i finalment no hi ha un màxim de steps per episodi (Max Step = 0), ja que l'atacant col·lisionarà inevitablement contra el cub o el defensor.

Inici episodi

Quan un nou episodi comença, posicionem el cilindre atacant a una posició aleatòria dintre d'uns paràmetres respecte del cub objectiu, aturem la velocitat a la que s'estava movent cap el cub en l'anterior episodi i posem endavant del cub el cilindre defensor, tot aturant també el moviment de l'anterior episodi. El codi corresponent a això (Figura 9.4) s'implementa al mètode *OnEpisodeBegin()*, el qual és sobreescrit respecte a la implementació a la classe Agent base.

```
// Set up an Agent instance at the beginning of an episode
1 referencia
public override void OnEpisodeBegin()
{
    // Agent's initial position and velocity
    gameObject.transform.position = new Vector3(goal.position.x + 2.25f, 0, goal.position.z + 0);
    rBody.velocity = Vector3.zero;
    gameObject.transform.localRotation = new Quaternion(0.707105756f, 0, 0, 0.707107902f);
    rBody.angularVelocity = Vector3.zero;


    // Attacker's initial position and velocity
    attacker.position = new Vector3(goal.position.x + 9.5f, 0, Random.Range(goal.position.z - 5f, goal.position.z + 5f));
    attacker.GetComponent<Rigidbody>().velocity = Vector3.zero;
    attacker.transform.localRotation = new Quaternion(0, 0, 0, 1);
    attacker.GetComponent<Rigidbody>().angularVelocity = Vector3.zero;
}
```

Figura 9.4: Primera prova - Mètode *OnEpisodeBegin()* a la classe "SwordDefender"

Observacions

L'agent tindrà la informació de la seva velocitat, posició en l'eix Z, la posició i la velocitat de l'atacant, i la posició del objectiu (el cub). Fem servir el mètode *CollectObservations(VectorSensor)* tot afegint les observacions mitjançant el mètode *AddObservation()*. El codi resultant és mostra a la Figura 9.5.

Cal recordar que les posicions i rotacions són respecte el GameObject Empty anomenat Training Area. Per aquesta raó, quan vulguem fer-hi referencia a aquest tipus de dades fem servir les locals i no les globals.

```
// Collect the vector observations of the agent for the step.
// The agent observation describes the current environment from the perspective of the agent.
1 referencia
public override void CollectObservations(VectorSensor sensor)
{
    // Agent's observations
    sensor.AddObservation(transform.localPosition);
    sensor.AddObservation(rBody.velocity.z);

    // Attacker's observations
    sensor.AddObservation(attacker.localPosition);
    sensor.AddObservation(attacker.GetComponent<Rigidbody>().velocity);

    // Goal's observations
    sensor.AddObservation(goal.localPosition);

}
```

Figura 9.5: Primera prova - Mètode *CollectObservations(VectorSensor)* a la classe "SwordDefender"

Accions

Al només poder moure l'agent en un sol eix, la única acció que tindrà serà del tipus continua. Amb la funció *OnActionReceived(ActionBuffers)*, que és cridada cada cop que l'algoritme respon amb una acció, apliquem el vector que rebem multiplicat pel factor forceMultiplier i apliquem el resultat com una força al rigidBody de l'agent. Si el resultat posicional d'aquesta força va més enllà dels límits calculats a ull, li donem a l'algorisme com a resposta una recompensa de -1, és a dir, que ho ha fet malament. El codi resultant es pot observar a la Figura 9.6.

```
// Allow the agent to execute actions based on the ActionBuffers contents.
1 referencia
public override void OnActionReceived(ActionBuffers actionBuffers)
{
    // Initiate the Vector3 where the received data will be stored
    Vector3 controlSignal = Vector3.zero;

    // The received action is saved in the Z-axis of the previous Vector3
    controlSignal.z = actionBuffers.ContinuousActions[0];

    // A force is applied in the direction of the received action multiplied by the forceMultiplier factor.
    rBody.AddForce(controlSignal * forceMultiplier);

    // If the position after moving the sword with the previuos force is out of range, a reward of -1 is set and the episode ends.
    if(transform.position.z > (5f + goal.position.z) || transform.position.z  < (goal.position.z - 5f))
    {
        SetReward(-1f);
        EndEpisode();
    }
}
```

Figura 9.6: Primera prova - Mètode *OnActionReceived(ActionBuffers)* a la classe "SwordDefender"

Heurística

Mitjançant el mètode *Heuristic(ActionBuffers)*, obtenim la única acció que pot realitzar l'agent i li assignem un input. En aquest cas, l'input de Unity de l'eix horitzontal, que controla les fletxes direccionals laterals del teclat i les tecles alfabètiques "A" i "D". En la figura 9.7 hi ha el codi que ho implementa.

```
// Choose an action for this agent using a custom heuristic.
1 referencia
public override void Heuristic(in ActionBuffers actionsOut)
{
    var continuousActionsOut = actionsOut.ContinuousActions;
    continuousActionsOut[0] = Input.GetAxis("Horizontal");
}
```

Figura 9.7: Primera prova - Mètode *Heuristic(ActionBuffers)* a la classe "SwordDefender"

9.1.2 Atacant

Namespaces

La classe de l'espasa atacant ("MoveAttacker") tindrà només el nameSpace de UnityEngine per a fer referencia a les classes de Unity.

Com es hereu de la classe MonoBehaviour, s'inicialitza la classe de la següent manera:

```
public class MoveAttacker : MonoBehaviour
```

Variables

En aquesta primera prova, té les següents variables:

- **public Transform target**
  Transform del cub objectiu de l'atacant. Referenciat mitjançant l'Inspector de Unity.
- **public GameObject defender**
  GameObject del cilindre defensor. Referenciat mitjançant l'Inspector de Unity.
- **private Rigidbody rBody**
  RigidBody del cilindre atacant.

De la mateixa manera que el rigidBody de l'agent, s'ha utilitzat el mètode *Start()* de Unity per assignar a la variable rBody el rigidBody del cilindre atacant.

Components del GameObject

La Figura 9.8 mostra els components del cilindre atacant.

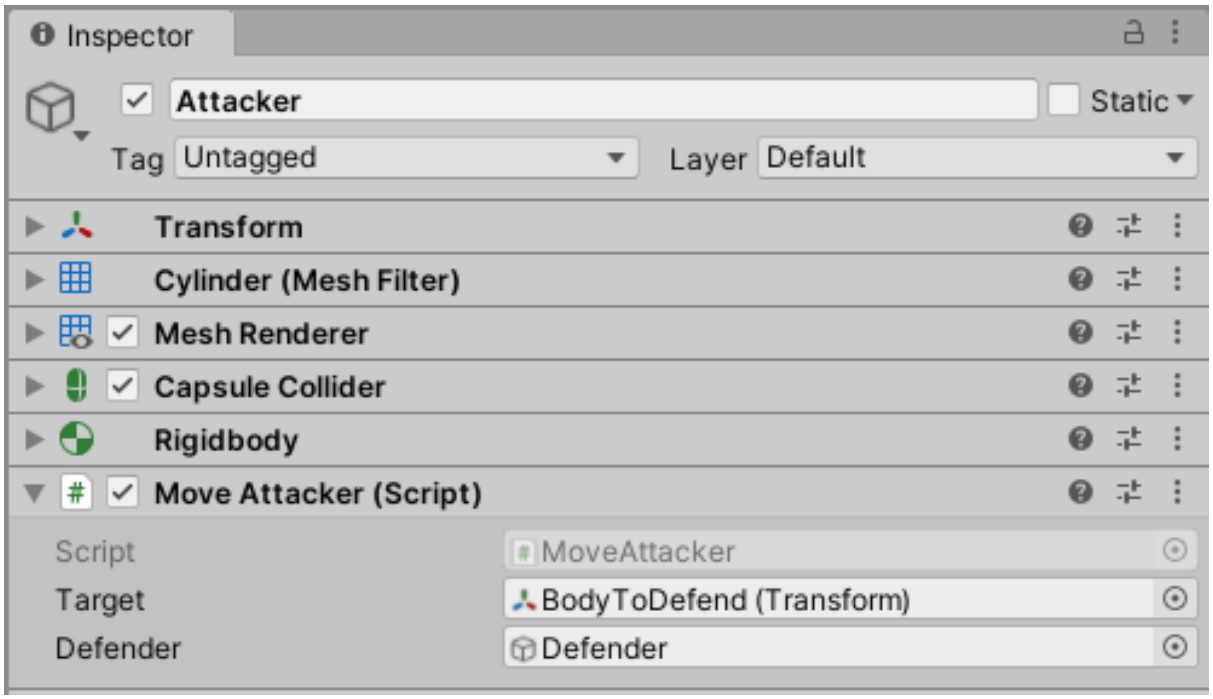


Figura 9.8: Primera prova - Inspector del GameObject "Attacker"

Mètodes

Com que s'ha de moure cap al cub, implementem la funció de Unity *FixedUpdate()* la qual es cridada amb la freqüència d'actualització del càlcul de la física de l'escena. Per tant, apliquem una força al cilindre atacant en direcció al cub mitjançant el rigidBody. En la següent imatge (Figura 9.9) es mostra el mètode usat (*rbody.AddForce()*).

```
// FixedUpdate is called once per physics-frame
Mensaje de Unity | 0 referencias
private void FixedUpdate()
{
    rBody.AddForce((target.position - transform.position) * 1f);
}
```

Figura 9.9: Primera prova - Mètode *FixedUpdate()* a la classe "MoveAttacker"

Posteriorment, fent servir la funció *OnTriggerEnter(Collider)*, comprovem quan hi ha una col·lisió amb el collider. Si l'objecte té com a Tag == "Goal", vol dir que ha col·lidit amb el cub. Per tant, agafant la referencia de l'agent a la classe, establim una recompensa de -1 i acabem l'episodi d'entrenament. En les següents línies de codi (Figura 9.10) es pot veure com està implementat.

```
// OnTriggerEnter happens on the FixedUpdate function when two GameObjects collide.
Mensaje de Unity | 0 referencias
public void OnTriggerEnter(Collider other)
{
    if (other.gameObject.tag == "Goal")
    {
        defender.GetComponent<SwordDefender>().SetReward(-1f);
        defender.GetComponent<SwordDefender>().EndEpisode();
    }
}
```

Figura 9.10: Primera prova - Mètode *OnTriggerEnter(Collider)* a la classe "MoveAttacker"

Finalment, per veure la col·lisió amb el cilindre defensor, es fa servir el mètode *OnCollisionEnter(Collision)*. Mirem si la col·lisió amb un altre rigidBody té com a Tag == "Defender", i si és així, calculem la distancia en l'eix Z entre el punt de la col·lisió i la posició del cilindre defensor. A aquesta distància li restem 1, i el resultat és la recompensa final. D'aquesta manera, quan menys distancia hi hagi del centre del cilindre al punt de la col·lisió, més recompensa obtindrà. El codi de com està implementat es mostra a la Figura 9.11.

```
// OnCollisionEnter is called when this collider/rigidbody has begun touching another rigidbody/collider.
Mensaje de Unity | 0 referencias
public void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.tag == "Defender")
    {
        float distance = Mathf.Abs(collision.GetContact(0).point.z - collision.gameObject.transform.position.z);
        float reward = 1f - distance;

        defender.GetComponent<SwordDefender>().SetReward(reward);
        defender.GetComponent<SwordDefender>().EndEpisode();

    }
}
```

Figura 9.11: Primera prova - Mètode *OnCollisionEnter(Collision)* a la classe "MoveAttacker"

9.1.3 Objectiu

En aquesta primer prova, el cub verd no té cap classe ni comportament més enllà d'estar darrere del cilindre defensor amb un collider. A la següent imatge (Figura 9.12) observem els components.

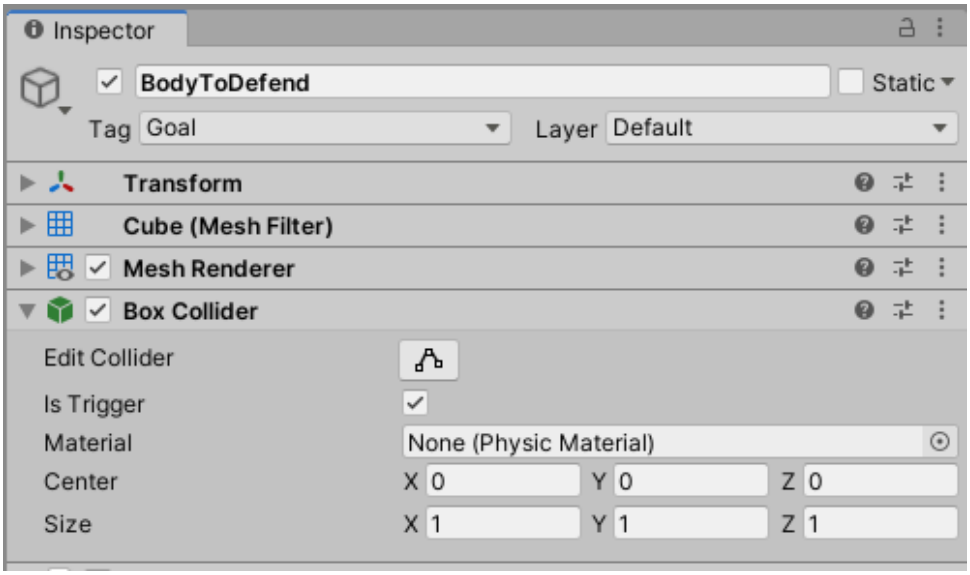


Figura 9.12: Primera prova - Inspector del GameObject "BodyToDefend"

9.1.4 Fitxer de configuració YAML

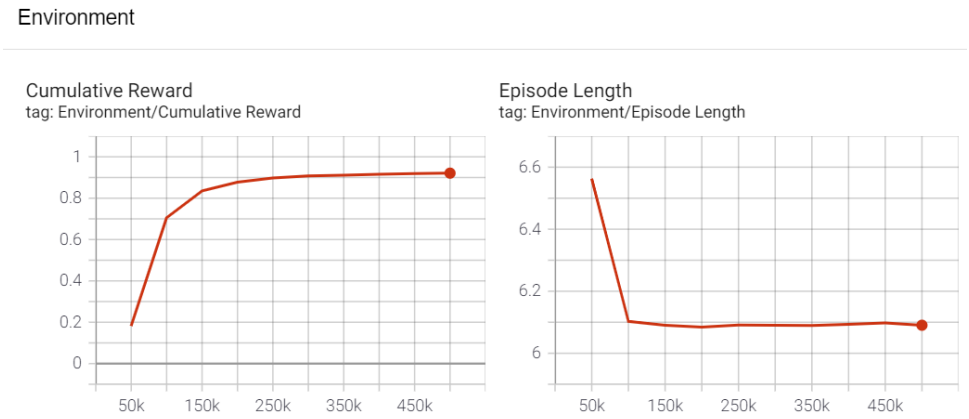
El fitxer YAML d'aquest entorn n'és un de molt senzill amb molts paràmetres amb els valors predeterminats.

```yaml
behaviors:
  DefendBox:
    trainer_type: ppo
    hyperparameters:
      batch_size: 120
      buffer_size: 12000
      learning_rate: 0.0003
      beta: 0.001
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      learning_rate_schedule: linear
    network_settings:
      normalize: true
      hidden_units: 128
      num_layers: 2
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    keep_checkpoints: 5
    max_steps: 500000
    time_horizon: 1000
    summary_freq: 50000
```

Figura 9.13: Primera prova - Paràmetres del fitxer de configuració YAML

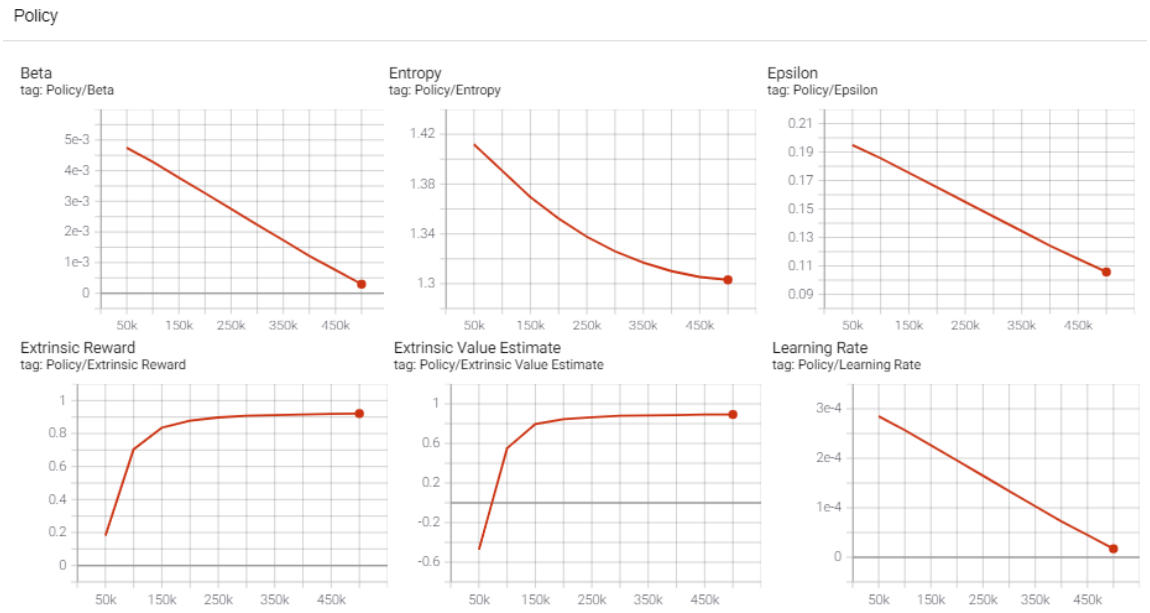
9.1.5 Resultats

Gràfiques dels estadístics de TensorBoard (Figura 9.14) després de l'entrenament complert.



(a) Estadístics Enviroment

(b) Estadístics Losses



(c) Estadístics Policy

Figura 9.14: Primera prova - Estadístics de TensorBoard

Com s'ha pogut observar, la xarxa resultant arriba a una recompensa mitjana aproximada de 0.9, observable a la figura 9.14a, al gràfic de "Cumulative Reward", més que suficient com per teoritzar que ha après la tasca correctament.

En altres estadístics, com el "Value Loss" (Figura 9.14b), podem observar com el valor comença amb un valor elevat i acaba descendint regularment a 0. Això ens indica que en els primers 150.000-200.000 steps ha après el màxim possible, com podem veure al gràfic de "Cumulative Reward" abans mencionat. A partir d'aquests steps l'aprenentatge és tant baix que la gràfica de "Value Loss" s'estabilitza, tot i que hi ha alguna variació imperceptible en la representació actual (s'hauria d'ampliar el gràfic). Això es causa que hi ha una petita millora dels 250.000 steps fins al final.

A la Figura 9.15 podem observar un exemple de com respon la xarxa neuronal resultant.

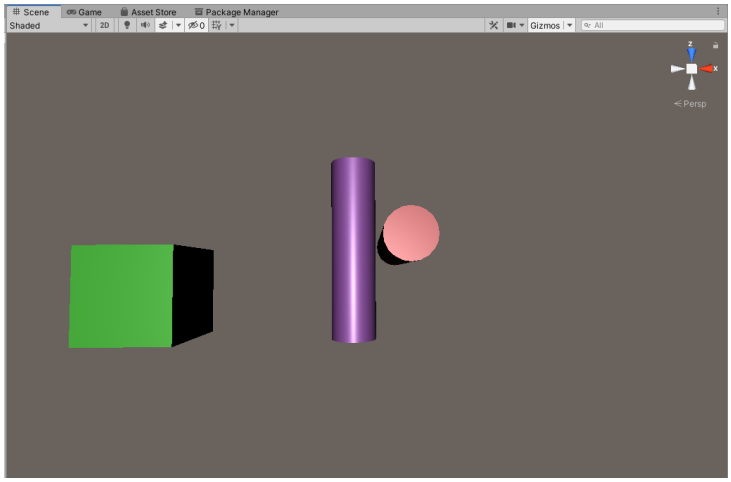


Figura 9.15: Primera prova - Posició final donada per la xarxa neuronal resultant

9.1.6 Problemes i solucions

Límits de moviment
En primer lloc, no es van posar límits d'espai per a que es mogués l'agent. A causa d'això, l'agent tardava molt en trobar l'acció que li donés alguna recompensa si es que la trobava.

Per aquesta raó, és va decidir posar límits. Això va solucionar el problema, ja que un cop codificat, l'agent aprenia a no anar més enllà d'aquests límits i així col·lisionava involuntàriament amb l'atacant, trobant així que li donava una recompensa positiva.

El FixedUpdate()
En segon lloc, en els primers intents d'entrenament no es tenia el coneixement suficient sobre el funcionament de ML-Agents, i el que es realitzaven no donaven el resultat esperat. Això era a causa que el moviment del cilindre atacant el realitzava amb el mètode *Update()* en comptes del *FixedUpdate()*. Aquest simple error feia que l'agent anés desincronitzat amb l'atacant i, per tant, les dades d'observació que obtenia en un moment donat podien no ser correctes. Per entendre-ho millor, si l'agent obtenia les dades en el frame X i l'actualització de la posició de l'atacant era al frame X+1, tenim que l'agent té informació errònia.

Com ja s'ha pogut comprovar, un cop es va canviar de mètode, l'agent va aconseguir aprendre bé el comportament desitjat.

9.2 Segona implementació: Cubs i cilindres - Moviment lliure

En aquesta segona implementació, es va decidir modificar les accions possibles de l'agent per apropar-les al comportament final de l'espasa d'un guerrer.

Pel que fa a l'escena de Unity no hi va haver cap canvi, és a dir, es van mantenir els tres GameObjects.

A la figura 9.16 es mostra els nous eixos que pot fer servir l'agent.

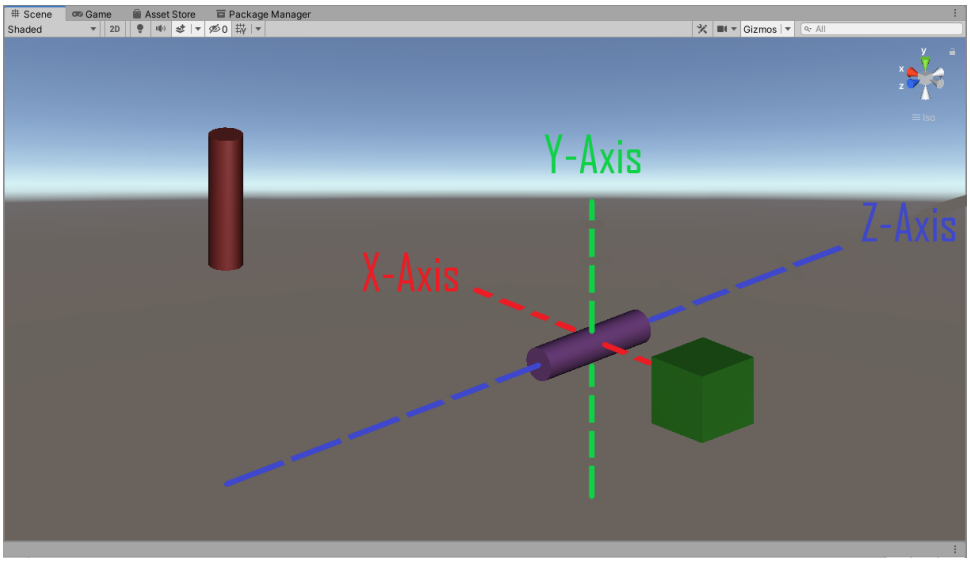


Figura 9.16: Escena Unity "Training Scene" de la segona implementació

9.2.1 Agent

Namespaces

Cap canvi en els Namespaces.

Variables

Es mantenen les mateixes variables que l'anterior implementació.

Components del GameObjet

La figura 9.17 mostra els components que té l'agent.

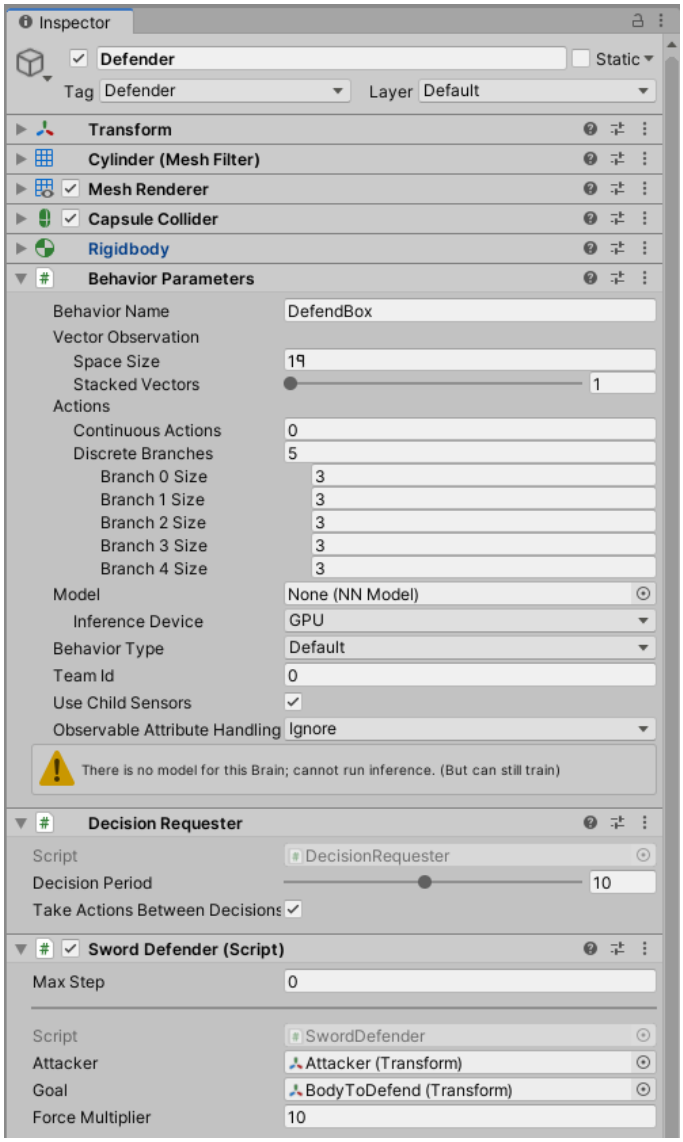


Figura 9.17: Segona prova - Inspector del GameObject "Defender"

En el component *BehaviorParameters* s'ha l'espai del vector d'observacions a 19 (afegim la rotació en quaternions de l'agent i la velocitat completa de l'agent) i s'ha canviat el nombre d'accions. Ara seran 5 accions discretes (moviment lliure, per tant, tres eixos; i rotació, en 2 eixos) amb 3 opcions cada una: positiu, negatiu i neutre.

La resta de paràmetres es mantenen inalterats.

Inici episodi

A diferencia del anterior *OnEpisodeBegin()*, canviem la posició inicial de l'atacant, el qual ara podrà aparèixer també a una alçada aleatòria.

```
// Set up an Agent instance at the beginning of an episode
1 referencia
public override void OnEpisodeBegin()
{
    // Agent's inital positon and velocity
    gameObject.transform.position = new Vector3(goal.position.x + 2.25f, 0, goal.position.z + 0);
    rBody.velocity = Vector3.zero;
    gameObject.transform.localRotation = new Quaternion(0.707105756f, 0, 0, 0.707107902f);
    rBody.angularVelocity = Vector3.zero;

    // Attacker's initial position and velocity
    attacker.position = new Vector3(goal.position.x + 9.5f,
        Random.Range(goal.position.y - 5f, goal.position.x + 5f) ,
        Random.Range(goal.position.z - 5f, goal.position.z + 5f));
    attacker.GetComponent<Rigidbody>().velocity = Vector3.zero;
    attacker.transform.localRotation = new Quaternion(0, 0, 0, 1);
    attacker.GetComponent<Rigidbody>().angularVelocity = Vector3.zero;
}
```

Figura 9.18: Segona prova - Mètode *OnEpisodeBegin()* a la classe "SwordDefender"

Observacions

L'agent ara veurà també la seva pròpia rotació i la velocitat en tots els eixos, fent que ara el vector d'observacions tingui una mida de 19. La Figura 9.19 mostra com queden les observacions de l'agent.

```
// Collect the vector observations of the agent for the step.
// The agent observation describes the current environment from the perspective of the agent.
0 referencias
public override void CollectObservations(VectorSensor sensor)
{
    // Agent's observations
    sensor.AddObservation(transform.localPosition);
    sensor.AddObservation(transform.localRotation);
    sensor.AddObservation(rBody.velocity);

    // ...
```

Figura 9.19: Segona prova - Mètode *CollectObservations(VectorSensor)* a la classe "SwordDefender"

Accions

Com s'ha comentat abans, s'han canviat les accions. Com que volem que l'espasa, que ara és un cilindre, pugi moure's lliurament i rotar alhora per l'entorn, s'ha d'indicar que aquestes seran discretes. Per aquesta raó, dividim en les accions en 5 de concurrents: moviment eix X, moviment eix Y, moviment eix Z, rotació eix X i rotació eix Z. Finalment, cada una d'aquestes accions se li ha d'indicar el nombre de valors que pot prendre. Cadascun pot prendre tres: positiu, negatiu i neutre.

El que s'ha de mirar és quines d'aquestes accions està volent realitzar la xarxa. Per això, comprovem si el valor de cada acció discreta és neutre, positiu o negatiu (0, 1, i 2 respectivament a l'index). Depenent d'això, creem els vectors de moviment i rotació que posteriorment aplicarem al rigidBody com a una força o, en el cas de la rotació, un *torque*.

A la Figura 9.20 es pot veure el codi que implementa l'anteriorment explicat.

```
// Allow the Agent to execute actions based on the ActionBuffers contents.
0 referencias
public override void OnActionReceived(ActionBuffers actionBuffers)
{
    // Get the action index for movement
    int movementX = actionBuffers.DiscreteActions[0];
    int movementY = actionBuffers.DiscreteActions[1];
    int movementZ = actionBuffers.DiscreteActions[2];

    // Get the action index for rotating
    int rotationX = actionBuffers.DiscreteActions[3];
    int rotationZ = actionBuffers.DiscreteActions[4];

    float dirX = 0f;
    float dirY = 0f;
    float dirZ = 0f;

    float rotX = 0f;
    float rotZ = 0f;

    // Look up the index in the movement action list:
    if (movementX == 1) { dirX = -1f; }
    if (movementX == 2) { dirX = 1f; }

    if (movementY == 1) { dirY = -1f; }
    if (movementY == 2) { dirY = 1f; }

    if (movementZ == 1) { dirZ = -1f; }
    if (movementZ == 2) { dirZ = 1f; }

    // Look up the index in the rotation action list:
    if (rotationX == 1) { rotX = -0.1f; }
    if (rotationX == 2) { rotX = 0.1f; }

    if (rotationZ == 1) { rotZ = -0.1f; }
    if (rotationZ == 2) { rotZ = 0.1f; }


    // Create the action's vectors
    Vector3 movementVector = new Vector3(dirX, dirY, dirZ);
    Vector3 rotationVector = new Vector3(rotX, 0f, rotZ);

    // Apply the action results to move the Agent
    rBody.AddRelativeForce(movementVector * forceMultiplier);
    rBody.AddRelativeTorque(rotationVector, ForceMode.VelocityChange);

    // If the position of the Agents surpase one of this limits, the reward will be -1 and the episode will be over
    if (transform.localPosition.x > 2.3f ||
        transform.localPosition.y > 9f || transform.localPosition.y < 5f ||
        transform.localPosition.z > 0.87f || transform.localPosition.z < -0.77f)
    {
        SetReward(-1f);
        EndEpisode();

    }
}
```

Figura 9.20: Segona prova - Mètode *OnActionReceived(ActionBuffers)* a la classe "SwordDefender"

Heurística

Al canviar les accions també canvia la heurística. Ara, per cada acció discreta i per cada valor que pot prendre, haurem de discernir un input. En el codi de la figura 9.21 es pot veure les tecles del teclat per a cada acció.

```
// Choose an action for this agent using a custom heuristic.
1 referencia
public override void Heuristic(in ActionBuffers actionsOut)
{
    var discreteActionsOut = actionsOut.DiscreteActions;

    // Initialate at neutral for each discreta action
    discreteActionsOut[0] = 0;
    discreteActionsOut[1] = 0;
    discreteActionsOut[2] = 0;
    discreteActionsOut[3] = 0;
    discreteActionsOut[4] = 0;

    // Input for the movement in X-Axis
    if (Input.GetKey(KeyCode.Q))
    {
        discreteActionsOut[0] = 1;
    }
    if (Input.GetKey(KeyCode.E))
    {
        discreteActionsOut[0] = 2;
    }
    // Input for the movement in Y-Axis
    if (Input.GetKey(KeyCode.S))
    {
        discreteActionsOut[1] = 1;
    }
    if (Input.GetKey(KeyCode.W))
    {
        discreteActionsOut[1] = 2;
    }
    // Input for the movement in Z-Axis
    if (Input.GetKey(KeyCode.A))
    {
        discreteActionsOut[2] = 1;
    }
    if (Input.GetKey(KeyCode.D))
    {
        discreteActionsOut[2] = 2;
    }
    // Input for the rotation in X-Axis
    if (Input.GetKey(KeyCode.F))
    {
        discreteActionsOut[3] = 1;
    }
    if (Input.GetKey(KeyCode.G))
    {
        discreteActionsOut[3] = 2;
    }
    // Input for the rotation in Z-Axis
    if (Input.GetKey(KeyCode.C))
    {
        discreteActionsOut[4] = 1;
    }
    if (Input.GetKey(KeyCode.V))
    {
        discreteActionsOut[4] = 2;
    }
}
```

Figura 9.21: Segona prova - Mètode *Heuristic(ActionBuffers)* a la classe "SwordDefender"

9.2.2 Atacant

Namespaces

Cap canvi en els namespaces de la classe "MoveAttacker".

Variables

Les variables no han estat modificades ni s'ha afegit cap de nova.

Components del GameObject

Els components s'han mantingut inalterats en aquesta nova implementació respecte a l'anterior.

Mètodes

L'únic canvi que s'ha realitzat ha estat en la fórmula del càlcul de la recompensa per l'agent. Com que ara tant l'agent com l'atacant es poden moure per tot l'entorn, fer una simple resta en l'eix Z ignora la distància en altres eixos. Per aquesta raó, el càlcul es realitza mitjançant el mètode *Distance(a,b)*, implementat a l'estructura Vector3, el qual calcula la distància entre dos punts a l'espai. A més a més, ara volem que tots dos col·lisionin en el centre de cada cilindre (actualment la posició del Transform de cada cilindre). Per tant, calculem, per a cada cilindre la distància al punt de col·lisió, fem la resta de 1 per a cada una i dividim entre dos, ja que ara la recompensa passa a ser de màxim 0.5 per cada centre de cilindre.

```
// OnCollisionEnter is called when this collider/rigidbody has begun touching another rigidbody/collider.
Mensaje de Unity | 0 referencias
public void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.tag == "Defender")
    {
        float distanceDefender = Vector3.Distance(collision.GetContact(0).point, collision.gameObject.transform.position);
        float distanceAttacker = Vector3.Distance(collision.GetContact(0).point, transform.position);

        float rewardDefender = (1f - distanceDefender) / 2;
        float rewardAttacker = (1f - distanceAttacker) / 2;

        float reward = rewardAttacker + rewardDefender;

        defender.GetComponent<SwordDefender>().SetReward(reward);
        defender.GetComponent<SwordDefender>().EndEpisode();

    }
}
```

Figura 9.22: Segona prova - Mètode *OnCollisionEnter(Collision)* a la classe MoveAttacker

9.2.3 Objectiu

No hi ha cap canvi en el GameObject "BodyToDefend" respecte l'anterior prova.

9.2.4 Fitxer de configuració YAML

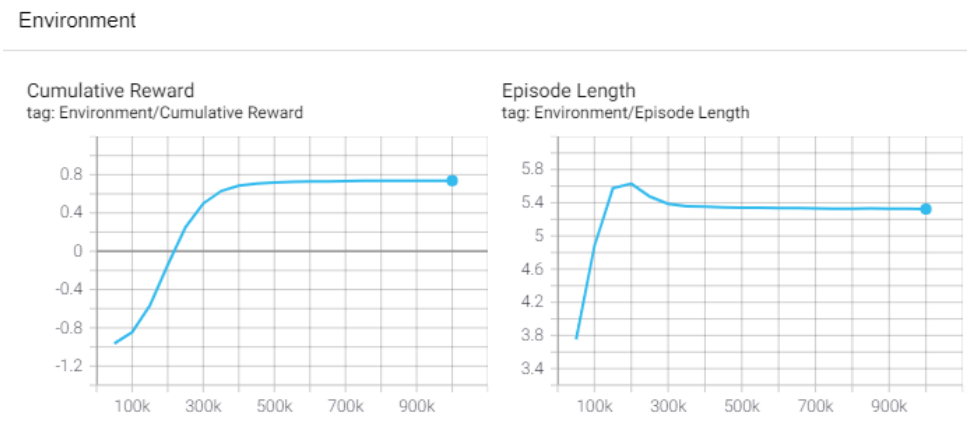
El fitxer YAML serà el mateix que l'anterior. Moltes de les dades predeterminades serveixen per a accions discretes. L'únic canvi que hi ha hagut ha estat el nombre de *max_steps*, ja que al ser un entorn un mica més complexa necessita de més temps d'entrenament per a provar més accions. En la següent figura 9.23 podem observar que aquest paràmetre ara té com a valor 1.000.000 d'steps.

```yaml
behaviors:
  DefendBox:
    trainer_type: ppo
    hyperparameters:
      batch_size: 120
      buffer_size: 12000
      learning_rate: 0.0003
      beta: 0.001
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      learning_rate_schedule: linear
    network_settings:
      normalize: true
      hidden_units: 128
      num_layers: 2
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    keep_checkpoints: 5
    max_steps: 1000000
    time_horizon: 1000
    summary_freq: 50000
```

Figura 9.23: Segona prova - Paràmetres del fitxer de configuració YAML

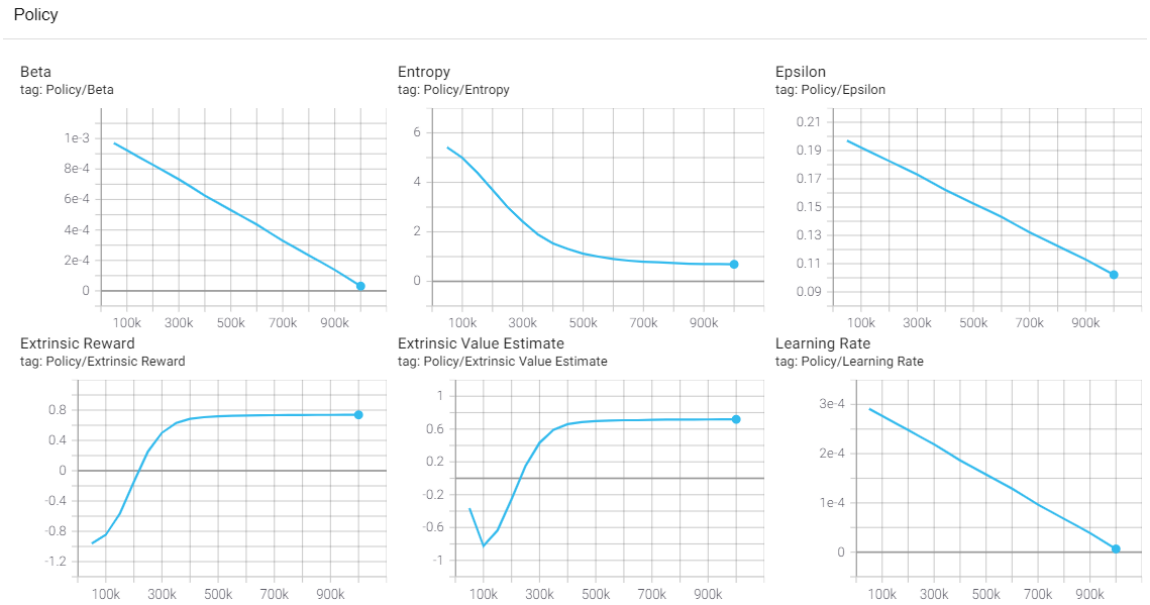
9.2.5 Resultats

Gràfiques dels estadístics de TensorBoard (Figura 9.24) després de l'entrenament complert.



(a) Estadístics Enviroment

(b) Estadístics Losses



(c) Estadístics Policy

Figura 9.24: Segona prova - Estadístics de TensorBoard

En aquesta ocasió, la recompensa mitjana aproximada és de 0.7 sobre 1, observable al primer gràfic de la figura 9.24a. Els decimals que resten per arribar a la recompensa màxima tenen origen en haver dividit en dos la recompensa i que la amplada dels cilindres també afecten a la distància al punt de col·lisió, és a dir, que tot i que la col·lisió fos just al centre, la distància calculada mai seria 0.

Referent a altres gràfiques d'interès, com s'ha comentat en l'anterior implementació, la gràfica de "Value Loss" (Figura 9.24b) ens indica el mateix comportament que en la primera prova. Durant els primers 400.000 steps va aprenent fins que s'estabilitza, deixant així d'augmentar la recompensa.

Per últim, a la Figura 9.25 es pot observar un exemple de com actua la xarxa neuronal resultant.

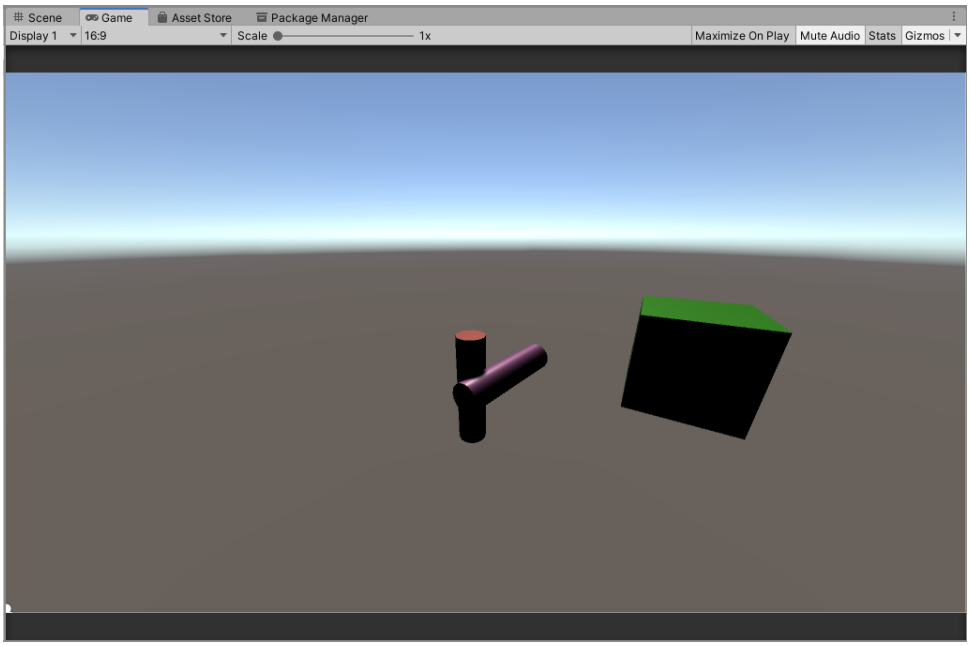


Figura 9.25: Segona prova - Posició final donada per la xarxa neuronal resultant

9.2.6 Problemes i solucions

Accions continues a discontinues

L'únic problema que va sorgir en aquesta implementació va ser la codificació de les accions. En primer instància, es van voler dividir en accions continues en compte de discretes. Per aquesta raó, en alguns entrenaments, l'agent no explorava totes les accions perquè al ser continues, per cada step del entrenament només es movia en algun dels tres eixos o rotava l'objecte, és a dir, la seqüencia d'accions era molt reduïda. Això feia que els resultats no fossin els esperats, amb moviments erràtics i una recompensa mitjana baixa (aproximadament de 0.4 sobre 1).

Finalment, al canviar les accions a discretes, va permetre a l'agent realitzar més d'una acció al mateix step (moure en més d'un eix i rotar).

9.3 Tercera implementació: Samurai amb espasa contra cilindre

Un cop definit quasi tot el comportament final de l'agent, toca canviar el mesh de cada una de les seves parts. Això vol dir, canviar el cub per un personatge i el cilindre defensor per una espasa. A més a més, mitjançant la cinemàtica inversa, es farà que sembli que el moviment de l'agent el dur a terme el personatge defensor.

En la següent figura 9.26 es pot observar el nou entorn: Un personatge que ocupa el lloc del GameObject "BodyToDefend", una espasa que fa de GameObject "Defender" i el cilindre atacant canviat de mida simulant una altra espasa.

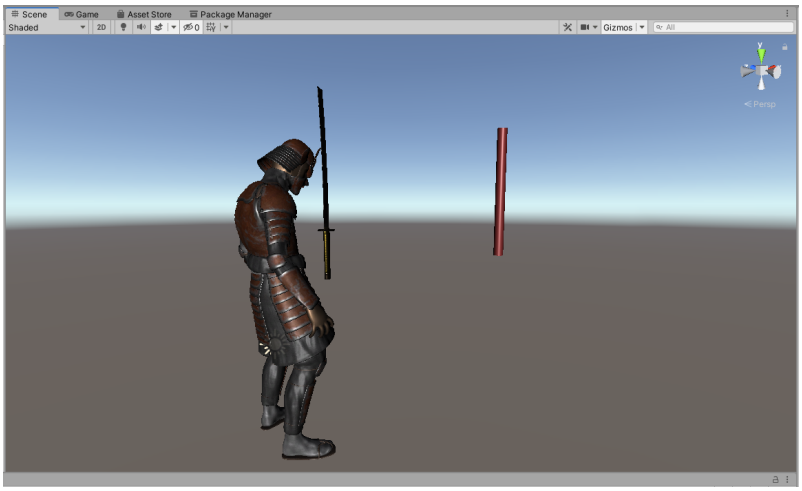


Figura 9.26: Escena Unity "Training Scene" de la tercera implementació

La jerarquia dels GameObjects segueix igual que en les anteriors implementacions.

9.3.1 Agent

Namespaces

Cap canvi en els Namespaces de la classe "SwordDefender".

Variables

Noves variables:

- **public Transform[] goal** (substitueix a l'anterior public Transform goal)
  Array de Transform del personatge defensor, l'objectiu a protegir de l'agent i d'atacar el cilindre atacant. Referenciat mitjançant l'Inspector de Unity.
- **public Transform trainingAreaAxis**
  Transform que fa referència al sistema d'eixos del àrea d'entrenament. Referenciat mitjançant l'Inspector de Unity.

Ara, la posició del defensor no té cap valor, ja que el seu cos és molt complexa. Per tant, es substitueix la variable Transform "goal" per una llista de Transforms, els quals fan referència a les parts individuals del personatge. Aquestes parts formen part de la jerarquia del personatge defensor, ja que aquest té l'esquelet d'animació com a GameObjects. A més a més, per a obtenir la posició relativa al sistema d'eixos que conforma l'àrea d'entrenament, es té una variable que hi fa referència, "trainingAreaAxis".

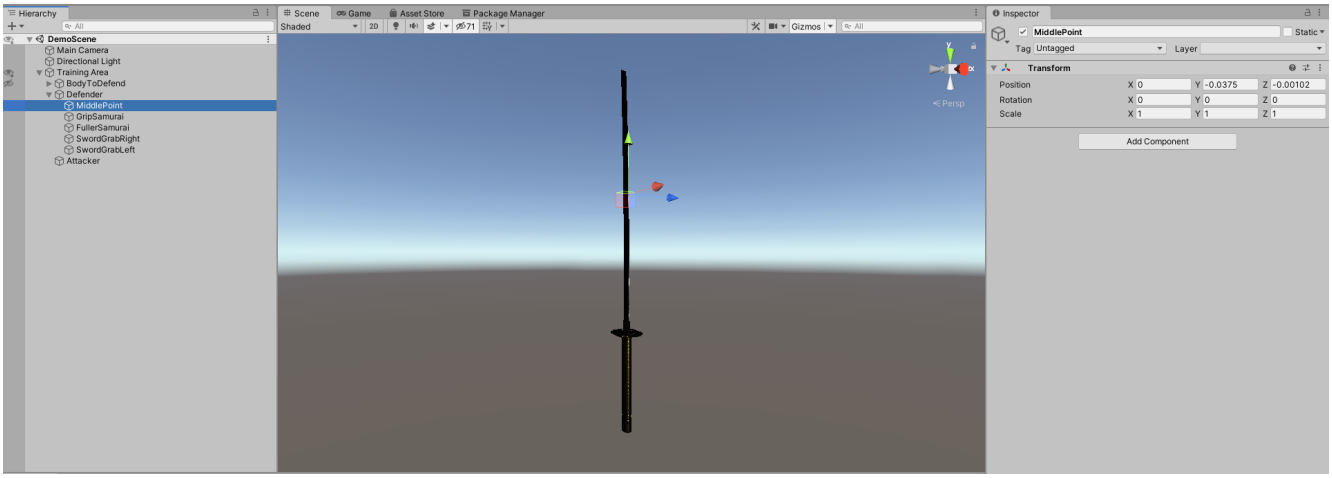
Jerarquia del GameObject

En aquesta implementació i fins la darrera, l'espasa té diferents GameObjects fills.

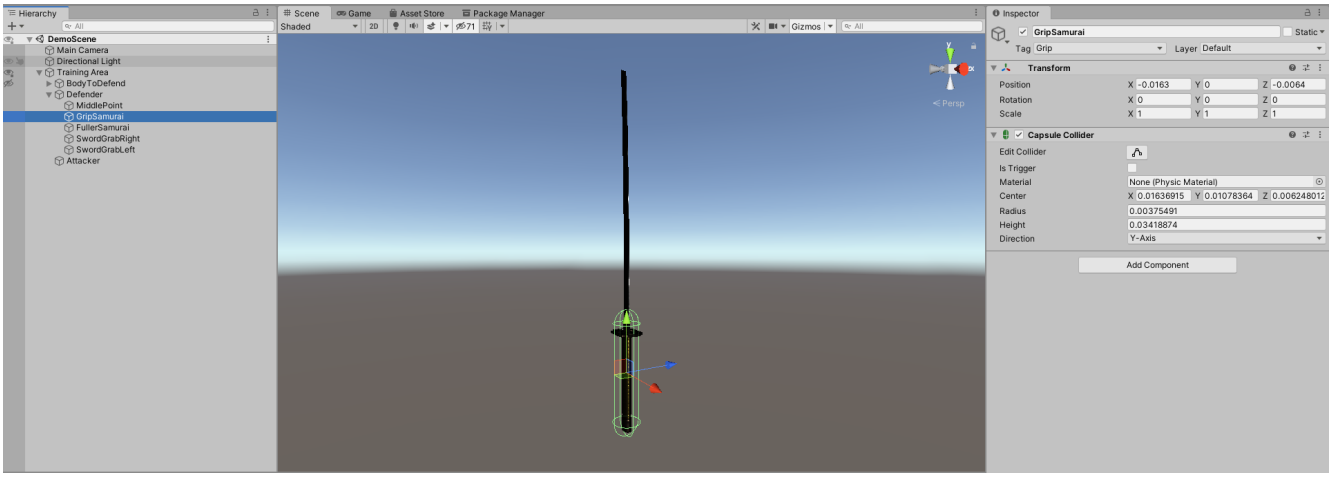
Primerament, un GameObject fill del tipus Empty que està posicionat just al centre de la fulla de l'espasa, anomenat "MiddelPoint" (Figura 9.27a). En les anteriors implementacions, ens bastava amb la posició del cilindre, ja que feia referencia al mig del objecte. Ara, la posició de l'espasa retorna la posició del maneg. Per tant, aquest fill serveix per tenir a l'abast aquesta posició tant important.

En segon lloc, dos fills anomenats "GripSamurai" (Figura 9.27b) i "FullerSamurai" (Figura 9.27c) que conformen els colliders de l'espasa. Al ser un objecte complexe, es va decidir dividir les col·lisions entre la fulla i el maneg de l'espasa. A més a més, per les propietats de pare-fill dels GameObjects, no fa falta crear noves classes per a obtenir les col·lisions. El mètode *OnTriggerEnter(Collider)* es cridat per a qualsevol collider fill o pertinent a ell.

Darrerament, s'ha definit dos GameObject Empty fills anomenats "SwordGrabRight" (Figura 9.27d) i "SwordGrabLeft" (Figura 9.27e). Estan situats en el maneg de l'espasa (un a la part superior i l'altre a l'inferior), i serveixen com a punts objectius per les mans del personatge respectivament per a dur a terme la cinemàtica inversa.

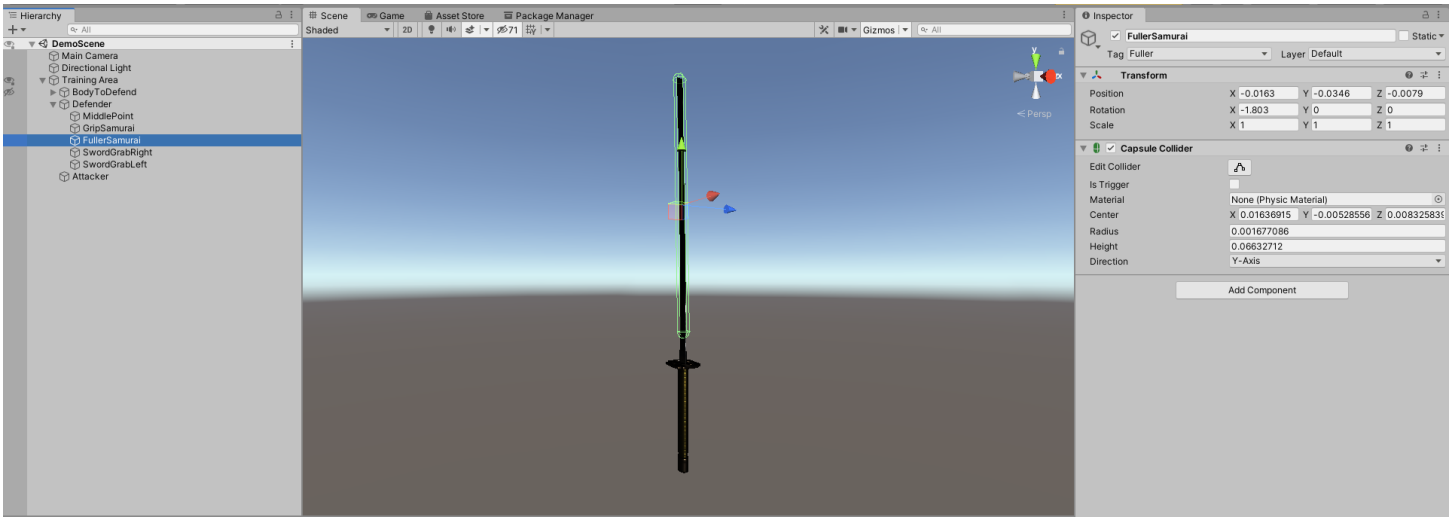


(a) GameObject "MiddlePoint"

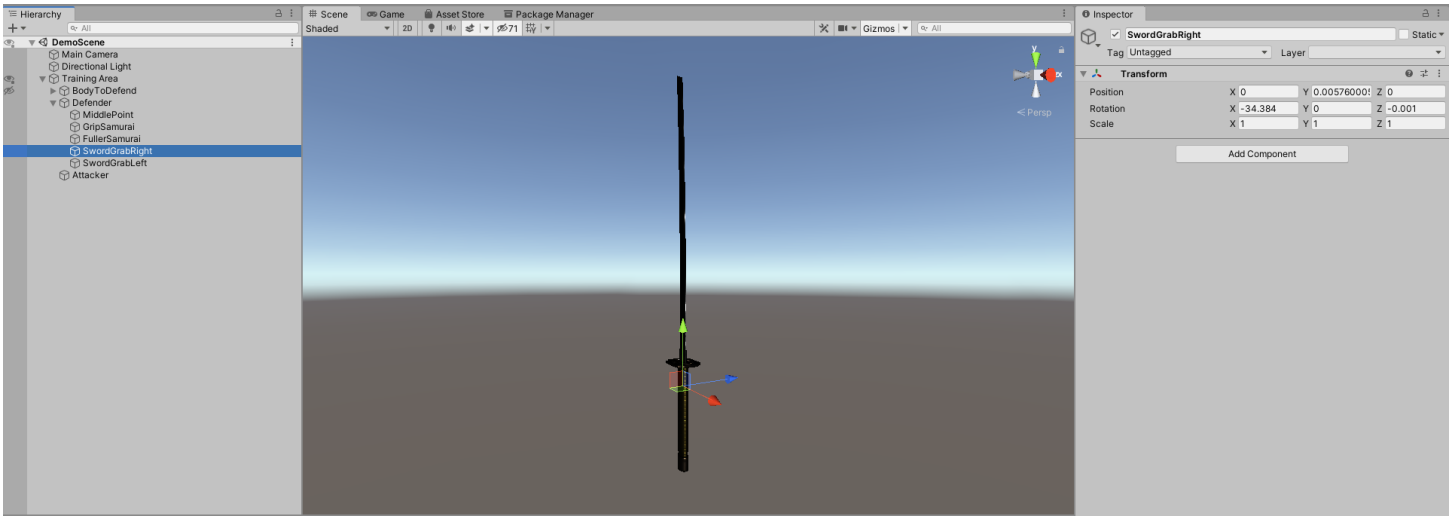


(b) GameObject "GripSamurai"

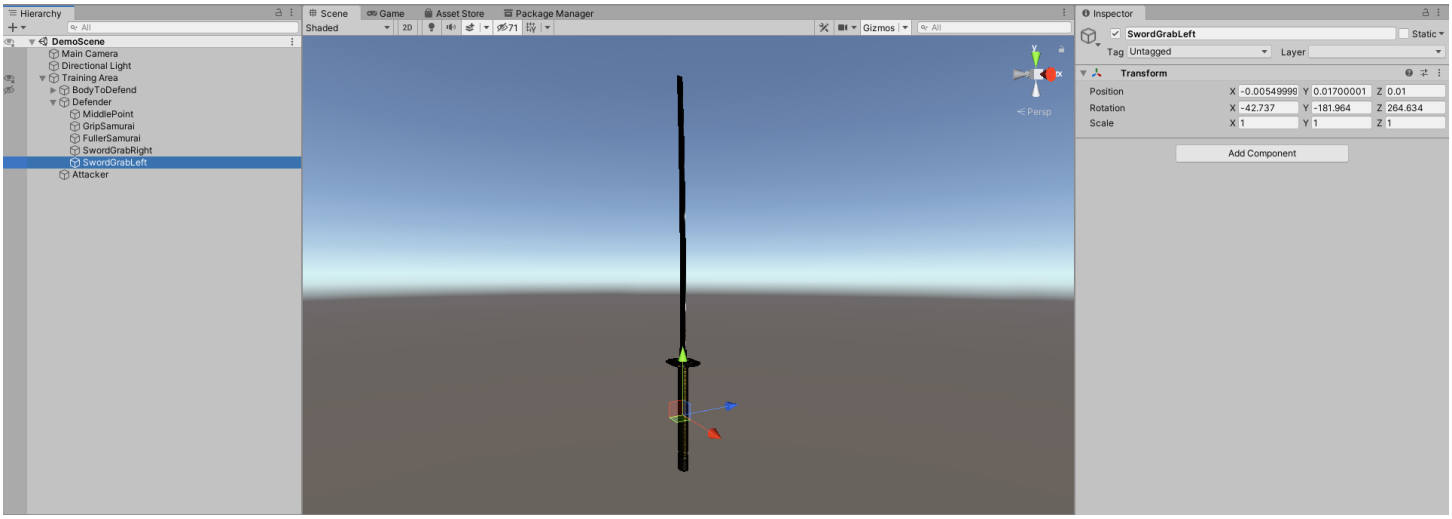
Figura 9.27: Tercera prova - Jerarquia del GameObject "SwordDefender"



(c) GameObject "FullerSamurai"



(d) GameObject "SwordGrabRight"



(e) GameObject "SwordGrabLeft"

Figura 9.27: Tercera prova - Jerarquia del GameObject "SwordDefender" (cont.)

Components del GameObject

La Figura 9.28 mostra els components que té l'agent (només el referents ML.Agents).

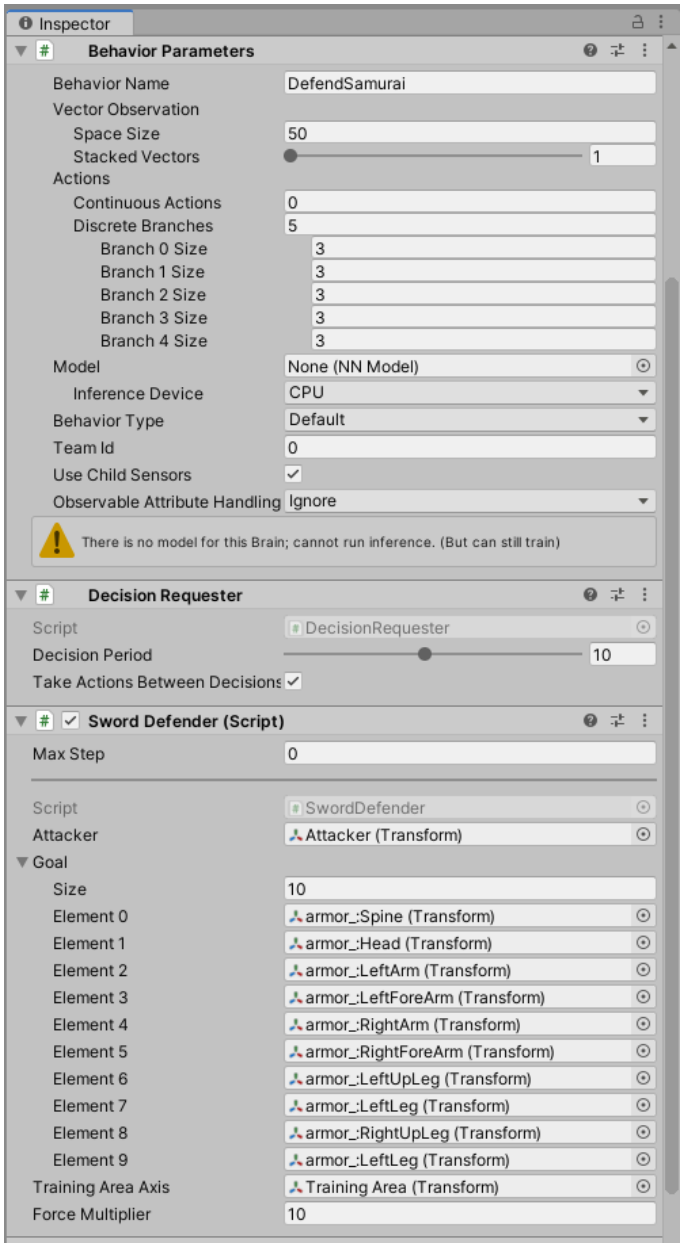


Figura 9.28: Tercera prova - Inspector del GameObject "Defender"

En el component *BehaviorParameters* s'ha assignat un nou nom al comportament, DefendSamurai, el vector d'observació s'ha establit en 50 variables, només apilarem 1 vector d'observacions per FixedUpdate (Stacked Vector) i seguim amb les 5 accions discretes amb 3 opcions per a cada una.

Pel que fa el *DecisionRequester*, es manté igual. On es veu la diferencia és en la col·lecció de Transforms que hi ha en l'script de "SwordDefender" i la referència a l'eix de coordenades de l'àrea d'entrenament, "Training Area Axis".

Inici episodi

Els únics canvis en el començament de cada episodi són en les posicions de l'agent i l'atacant. El codi corresponent a això està implementat en la Figura 9.29.

```
// Set up an Agent instance at the beginning of an episode
0 referencias
public override void OnEpisodeBegin()
{
    // Agent's inital positon and velocity
    gameObject.transform.localPosition = new Vector3(1.77999997f, 6.1930995f, 0.180999994f);
    rBody.velocity = Vector3.zero;
    gameObject.transform.localRotation = new Quaternion(-0.707107902f, 0, 0.707105756f, 0);
    rBody.angularVelocity = Vector3.zero;

    // Attacker's initial position and velocity
    attacker.position = new Vector3(goal[0].position.x + 9.5f,
        Random.Range(goal[0].position.y - 5f, goal[0].position.x + 5f) ,
        Random.Range(goal[0].position.z - 5f, goal[0].position.z + 5f));
    attacker.GetComponent<Rigidbody>().velocity = Vector3.zero;
    attacker.transform.localRotation = new Quaternion(0, 0, 0, 1);
    attacker.GetComponent<Rigidbody>().angularVelocity = Vector3.zero;
}
```

Figura 9.29: Tercera prova - Mètode *OnEpisodeBegin()* a la classe "SwordDefender"

Observacions

Com s'ha observat en *BehaviorParameters*, les observacions han augmentat en gran quantitat a causa de la divisió del GameObject "Goal" en 10. El número 50 ve donat per les 10 variables de l'agent, les 6 variables de l'atacant, i les 34 variables del defensor. Aquest últim en té tantes perquè necessitem saber la posició de les parts del cos on pot col·lisionar el cilindre atacant. En concret, l'agent sabrà la posició del cap, els braços (braç i avantbraç) i les cames (cuixa i cama). El codi resultant es troba a la Figura 9.30 de la següent pàgina.

```
// Collect the vector observations of the agent for the step.
// The agent observation describes the current environment from the perspective of the agent.
0 referencias
public override void CollectObservations(VectorSensor sensor)
{
    // Agent's observations
    sensor.AddObservation(transform.localPosition);
    sensor.AddObservation(transform.localRotation);
    sensor.AddObservation(rBody.velocity);

    // Attacker's observations
    sensor.AddObservation(attacker.localPosition);
    sensor.AddObservation(attacker.GetComponent<Rigidbody>().velocity);

    // Goal's observations
    // SPINE
    sensor.AddObservation(trainingAreaAxis.InverseTransformPoint(goal[0].position));

    // HEAD
    sensor.AddObservation(trainingAreaAxis.InverseTransformPoint(goal[1].position));
    sensor.AddObservation(Quaternion.Inverse(goal[1].rotation) * trainingAreaAxis.rotation);

    // LEFT ARM
    sensor.AddObservation(trainingAreaAxis.InverseTransformPoint(goal[2].position));
    sensor.AddObservation(trainingAreaAxis.InverseTransformPoint(goal[3].position));

    // RIGHT ARM
    sensor.AddObservation(trainingAreaAxis.InverseTransformPoint(goal[4].position));
    sensor.AddObservation(trainingAreaAxis.InverseTransformPoint(goal[5].position));

    // LEFT LEG
    sensor.AddObservation(trainingAreaAxis.InverseTransformPoint(goal[6].position));
    sensor.AddObservation(trainingAreaAxis.InverseTransformPoint(goal[7].position));

    // RIGHT LEG
    sensor.AddObservation(trainingAreaAxis.InverseTransformPoint(goal[8].position));
    sensor.AddObservation(trainingAreaAxis.InverseTransformPoint(goal[9].position));

}
```

Figura 9.30: Tercera prova - Mètode *CollectObservations(VectorSensor)* a la classe "SwordDefender"

Les rotacions només s'obtenen del cap del samurai i de l'espasa defensora perquè són els únics en rotar.

Accions

Per limitar millor els moviments de l'espasa i que sembli que l'està movent el samurai, els límits pels quals es pot moure l'agent varien. El codi següent (Figura 9.31) mostra aquests nous límits calculats manualment.

```
    // codi anterior

    // If the position of the Agents surpase one of this limits, the reward will be -1 and the episode will be over
    if (transform.localPosition.x > 2.3f ||
        transform.localPosition.y > 9f || transform.localPosition.y < 5f ||
        transform.localPosition.z > 0.87f || transform.localPosition.z < -0.77f)
    {
        SetReward(-1f);
        EndEpisode();

    }
}
```

Figura 9.31: Tercera prova - Nous límits en el mètode *OnActionReceived(ActionBuffers)* de la classe "SwordDefender"

Heurística

Com que no hi ha canvis en les accions, l'huerística es manté.

9.3.2 Atacant

Namespaces

No hi ha cap canvi en els Namespaces.

Variables

No hi ha cap nova variable.

Components del GameObject

La Figura 9.32 mostra els components del cilindre atacant, ja que ha variat en escala i la referència de la variable "Target" de l'script "MoveAttacker".

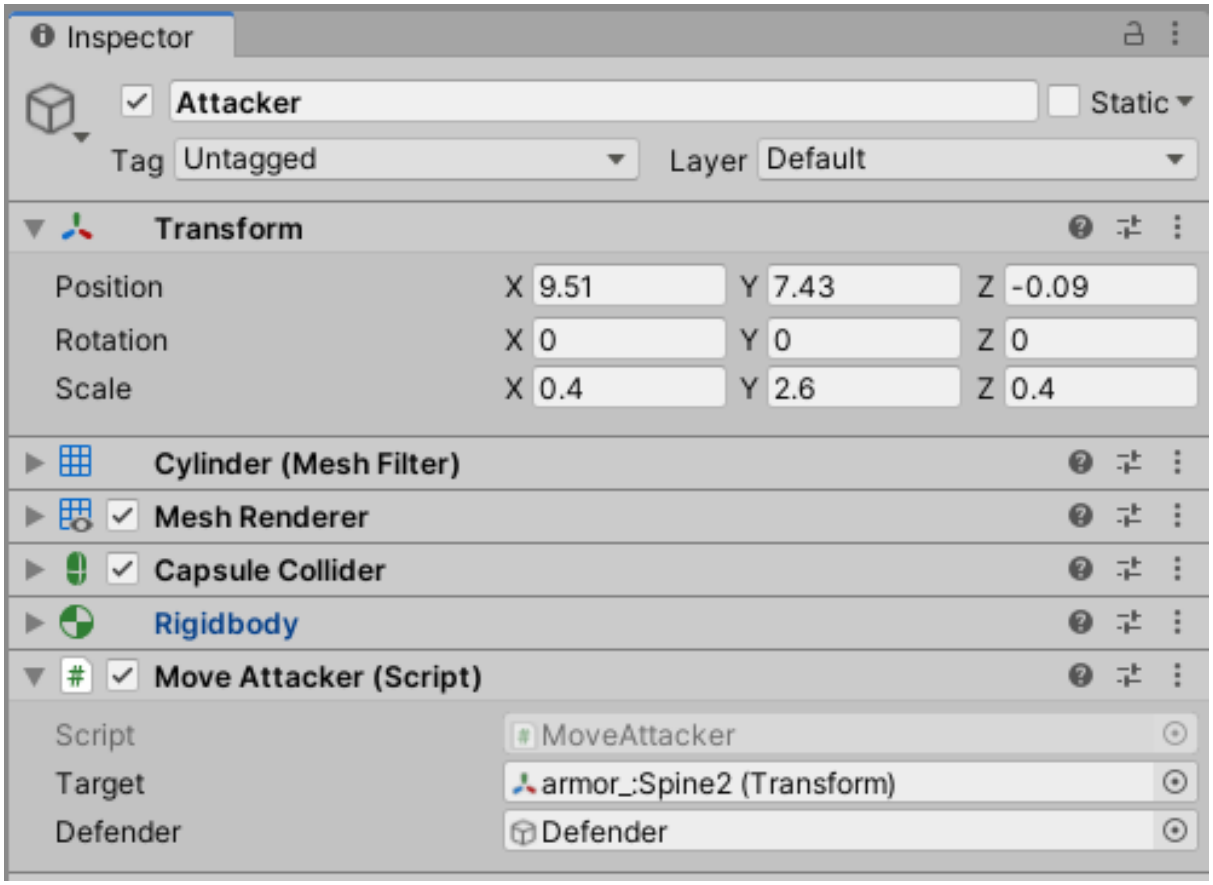


Figura 9.32: Primera prova - Inspector del GameObject "Attacker"

Mètodes

Els colliders de l'agent, al canviar de jerarquia, fan que els mètodes que depenien d'ells variïn. En el cas de *OnCollisonEnter(Collision)* del cilindre atacat, s'ha de modificar el comportament, ja que el GameObject que retorna de la col·lisió és el "Defender", i el que ens interessa és saber si a col·lidit amb "FullerSamurai" o amb "GripSamurai". Per aquesta raó, primerament calcularem la distància entre el punt de col·lisió i tots dos GameObjects. Si està més aprop del maneg vol dir que ha col·lidit amb ell. Per tant, lamentablement no ens interessa i li donem a l'agent una recompensa de -1. En canvi, si està mes aprop el tall de l'espasa, fem el següent càlcul:

$$reward = \frac{\left(1 - \frac{distance}{MAX\_DIS}\right)}{2}$$

On **MAX_DIS** és la distància màxima que hi ha entre el punt mitjà de l'espasa i els extrems, **distance** és la distància entre la col·lisió recent i el punt mitjà de l'espasa, i la divisió per 2 és pel repartiment de la recompensa entre les dues espases.

El punt mitjà de l'espasa defensora en aquest cas és el fill GameObject "MiddlePoint".

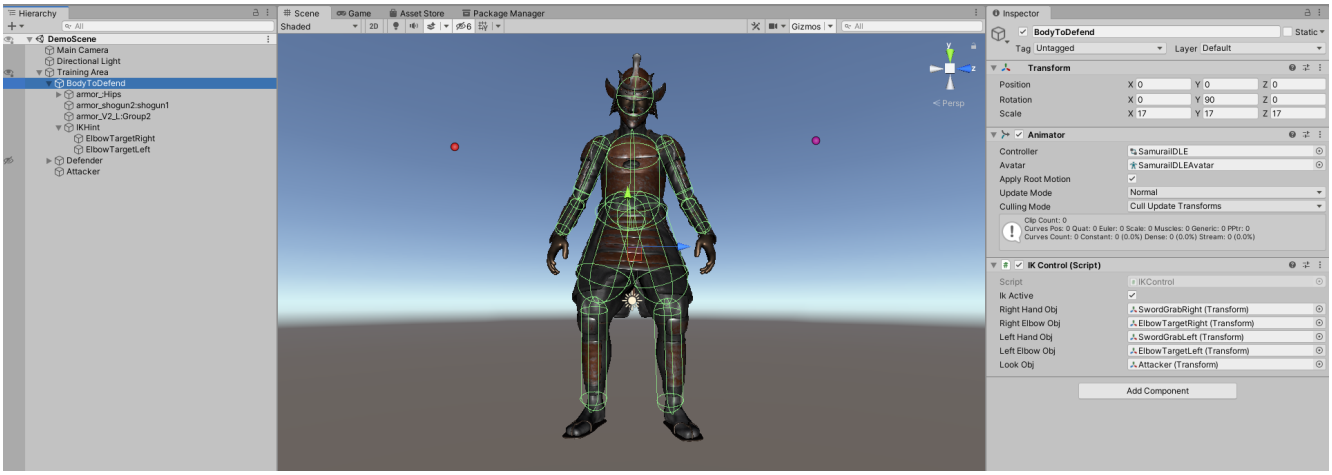
Aquests canvis es mostren implementats en el següent codi (Figura 9.33).

```
// OnCollisionEnter is called when this collider/rigidbody has begun touching another rigidbody/collider.
Mensaje de Unity | 0 referencias
public void OnCollisionEnter(Collision collision)
{
    // Check if the cilinder has collided with the grip or the fuller of the sword
    float distanceGrip = Vector3.Distance(defender.transform.GetChild(1).position, collision.GetContact(0).point);
    float distanceFuller = Vector3.Distance(defender.transform.GetChild(2).position, collision.GetContact(0).point);

    // If it has collided with the grip, bad
    if(distanceGrip <= distanceFuller)
    {
        defender.GetComponent<SwordDefender>().SetReward(-1);
        defender.GetComponent<SwordDefender>().EndEpisode();
    }
    // If it has collided with the fuller, good
    else
    {
        float distanceDefender = Vector3.Distance(collision.GetContact(0).point, collision.gameObject.transform.GetChild(0).position);
        float distanceAttacker = Vector3.Distance(collision.GetContact(0).point, transform.position);

        float rewardDefender = (1f - (Mathf.Clamp(distanceDefender, 0f, 3f) / 3f)) / 2f;
        float rewardAttacker = (1f - (Mathf.Clamp(distanceAttacker, 0f, 3f) / 3f)) / 2f;

        float totalReward = rewardAttacker + rewardDefender;

        defender.GetComponent<SwordDefender>().SetReward(totalReward);
        defender.GetComponent<SwordDefender>().EndEpisode();
    }
}
```

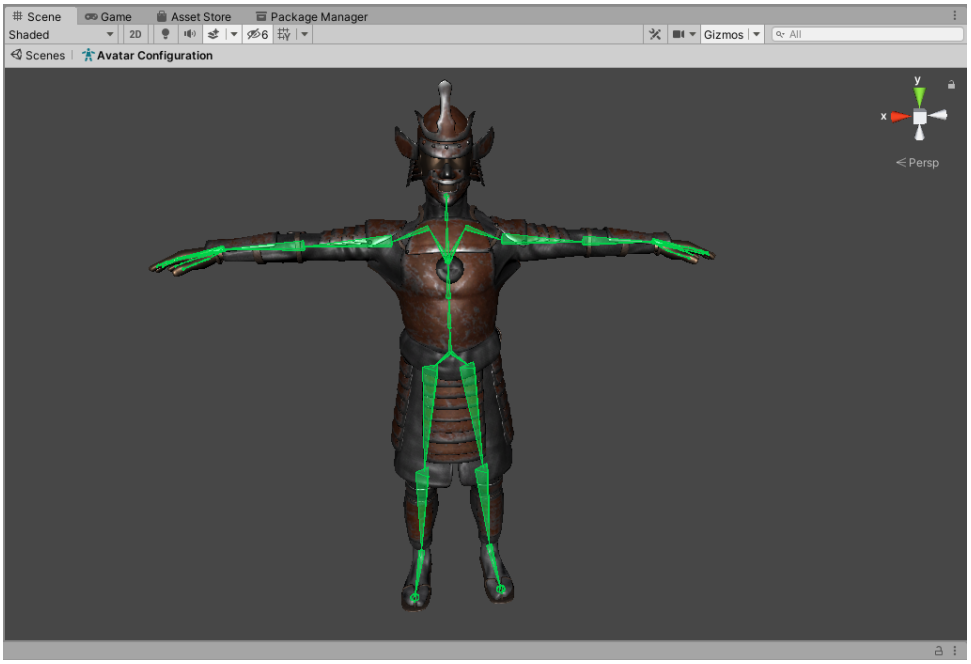
Figura 9.33: Tercer prova - Mètode *OnCollisionAttacker(Collsion) de la classe "MoveAttacker"*

9.3.3 Objectiu

El nou objectiu és un personatge 3D obtingut de manera gratuïta de la pàgina web *Sketchfab* [7]. Aquest li mancava l'esquelet incrustat al model, per tant, es va utilitzar la pàgina web de Mixamo, la qual et permet pujar un model humanoide 3D i fa automàticament el muntatge de l'esquelet sobre el model (Figura 9.34b). A la Figura 9.34a observem com és la jerarquia i com és veu a l'escena.



(a) Visualització dels colliders



(b) Avatar (esquelet) del model 3D

Figura 9.34: Tercera prova - Jerarquia del GameObject "BodyToDefend"

Respecte als diferents fills del "BodyToDefend", en primera instància, hi ha tota la jerarquia de l'esquelet ("armor_:hips"), on als respectius ossos hi ha els colliders amb tag == "Goal". En segona instancia, hi ha els dos mesh del personatge, l'armadura (2armor_V2_L:Group2") i la pell ("armor_shogun2:shogun1"). Per últim, hi ha una agrupació de dos GameObjects ("ElbowTargetRight" i "ElbowTargetLeft") que serveixen com a objectiu pels colzes en la implementació de la cinemàtica inversa.

Cinemàtica inversa

Com s'ha mostrat a l'anterior Apartat 8: Anàlisi i disseny del sistema, al diagrama de classes, es crea una classe que estarà inclosa al GameObject "BodyToDefend", la qual s'encarregarà d'aplicar la cinemàtica inversa a les mans del personatge.

Per fer-ho, primerament s'ha de crear un asset de Unity anomenat "Animator Controller", el qual controla la lògica d'un GameObject animat. Dins d'ell estan els States que estan enllaçades entre si a través de Transitions. Els States són la representació dels clips d'Animació (les animacions per se) en el Animator. En el cas del samurai, aquest estarà en una animació IDLE, és a dir, una animació d'inactivitat (el samurai respira i es mou com si estigués quiet dempeus). Aquesta animació es troba al state "IDLE", l'state del qual no es mourà. A més a més, per a realitzar cinemàtica inversa s'ha d'activar a la pestanya de "Layer Settings".

Tot aquesta explicació es pot veure reflectada en la Figura 9.35, on es pot veure l'state IDLE i l'opció de poder fer cinemàtica inversa activada.

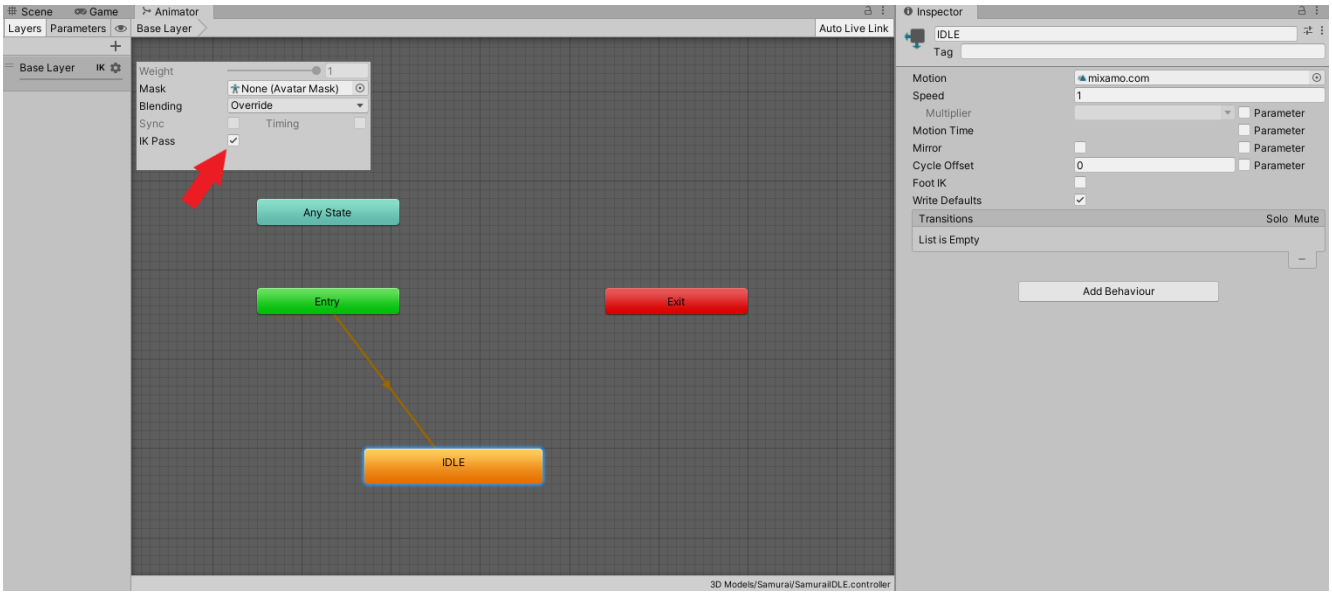


Figura 9.35: Tercer prova - "Animator Controller" del GameObject "BodyToDefend"

De la mateixa manera que l'esquelet del model del samurai es va obtenir de Mixamo, la animació de IDLE també, ja que Mixamo ofereix l'opció d'aplicar animacions a muntatges d'esquelets a models 3D que es pugin a la pàgina.

Un cop tot preparat, es codifica la cinemàtica inversa. Com es pot veure a la Figura 9.34a, els GameObject que tindran que buscar les mans són "SwordGrabRight" i "Sword-GrabLeft", juntament amb "ElbowTargetRight" i "ElbowTargetLeft" pels colzes, i amb el cap seguirà a "Attacker".

El codi de la Figura 9.36 mostra la implementació.

```
Mensaje de Unity | 0 referencias
void Start()
{
    animator = GetComponent<Animator>();
}

// A callback for calculating IK
Mensaje de Unity | 0 referencias
void OnAnimatorIK()
{
    //if the IK is active, set the position and rotation directly to the goal.
    if (ikActive)
    {
        // Set the look target position, if one has been assigned
        if (lookObj != null)
        {
            animator.SetLookAtWeight(1);
            animator.SetLookAtPosition(lookObj.position);
        }

        // Set the right hand target position and rotation and the right elbow target position and rotation, if one has been assigned
        if (rightHandObj != null)
        {
            animator.SetIKPositionWeight(AvatarIKGoal.RightHand, 1f);
            animator.SetIKRotationWeight(AvatarIKGoal.RightHand, 1f);
            animator.SetIKHintPositionWeight(AvatarIKHint.RightElbow, 1f);

            animator.SetIKPosition(AvatarIKGoal.RightHand, rightHandObj.position);
            animator.SetIKRotation(AvatarIKGoal.RightHand, rightHandObj.rotation);
            animator.SetIKHintPosition(AvatarIKHint.RightElbow, rightElbowObj.position);
        }

        // Set the left hand target position and rotation and the left elbow target position and rotation, if one has been assigned
        if (leftHandObj != null)
        {
            animator.SetIKPositionWeight(AvatarIKGoal.LeftHand, 1f);
            animator.SetIKRotationWeight(AvatarIKGoal.LeftHand, 1f);
            animator.SetIKHintPositionWeight(AvatarIKHint.LeftElbow, 1);

            animator.SetIKPosition(AvatarIKGoal.LeftHand, leftHandObj.position);
            animator.SetIKRotation(AvatarIKGoal.LeftHand, leftHandObj.rotation);
            animator.SetIKHintPosition(AvatarIKHint.LeftElbow, leftElbowObj.position);
        }
    }

    //if the IK is not active, set the position and rotation of everything to the original position
    else
    {
        animator.SetIKPositionWeight(AvatarIKGoal.RightHand, 0);
        animator.SetIKRotationWeight(AvatarIKGoal.RightHand, 0);

        animator.SetIKPositionWeight(AvatarIKGoal.LeftHand, 0);
        animator.SetIKRotationWeight(AvatarIKGoal.LeftHand, 0);
        animator.SetLookAtWeight(0);
    }

}
```

Figura 9.36: Tercera prova - Implementació de cinemàtica inversa al GameObject "BodyToDefend"

9.3.4 Fitxer de configuració YAML

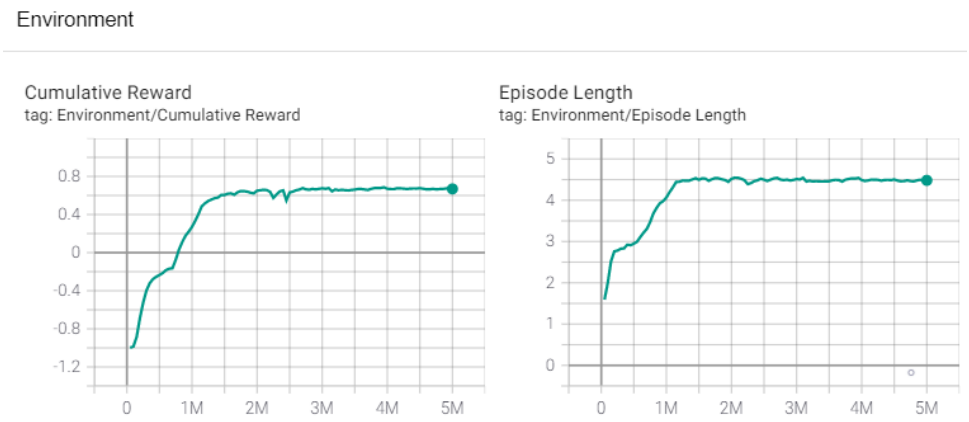
El fitxer YAML s'ha modificat en alguns paràmetres. S'ha canviat el nom del *behavior* a "DefendSamurai", s'ha augmentat a 512 el *batch_size*, és a dir, el número d'experiències recol·lectades abans d'actualitzar el model, s'ha augmentat lleugerament *beta* a 0.005, per a que l'agent explori totes les possibilitats durant l'entrenament, s'ha augmentat la xarxa neuronal a 3 capes ocultes de 512 neurones cada una, i per últim, s'ha augmentat el temps d'entrenament a 5.000.000 d'steps. En la Figura 9.37 podem observar aquests canvis.

```yaml
behaviors:
  DefendSamurai:
    trainer_type: ppo
    hyperparameters:
      batch_size: 512
      buffer_size: 12000
      learning_rate: 0.0003
      beta: 0.005
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      learning_rate_schedule: linear
    network_settings:
      normalize: true
      hidden_units: 512
      num_layers: 3
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    keep_checkpoints: 5
    max_steps: 5000000
    time_horizon: 1000
    summary_freq: 50000
```

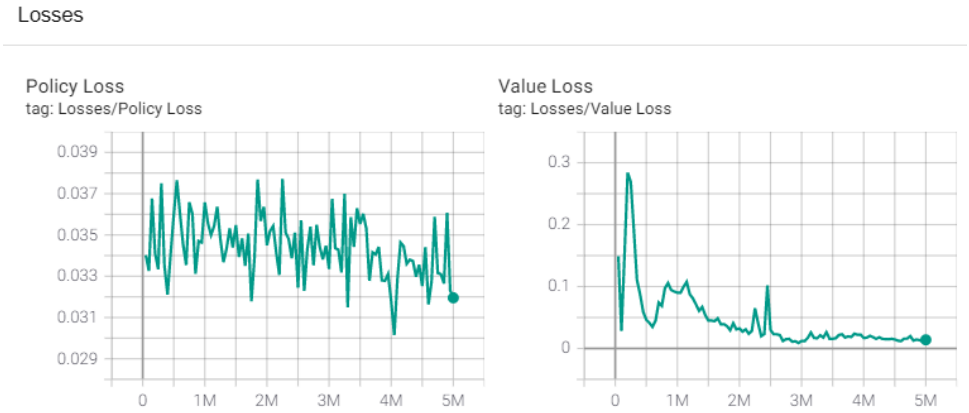
Figura 9.37: Tercera prova - Paràmetres del fitxer de configuració YAML

9.3.5 Resultats

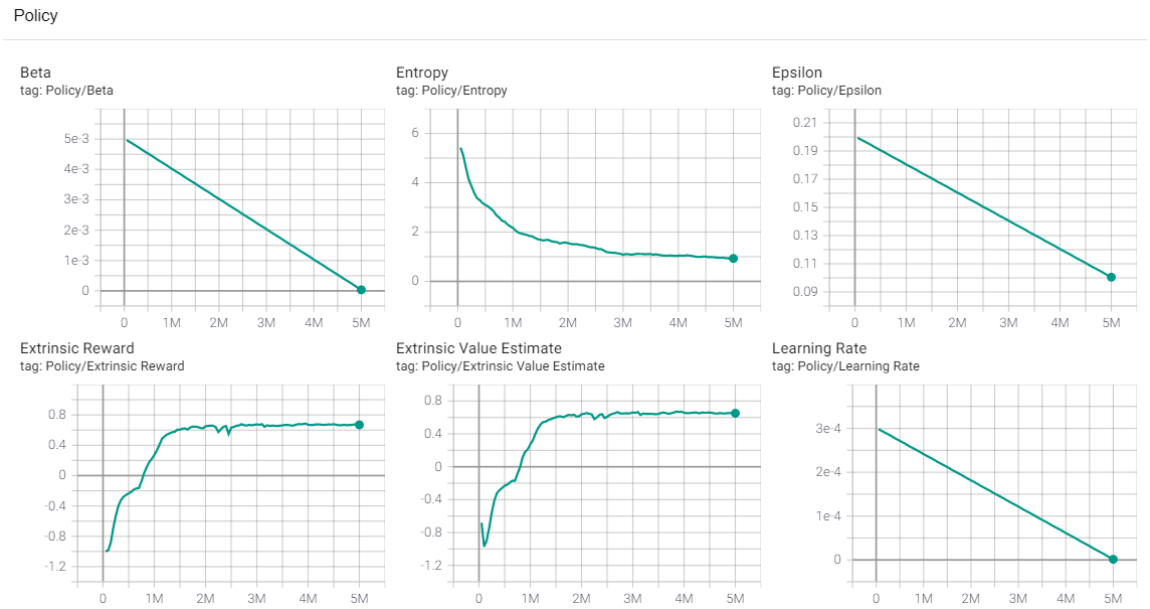
Gràfiques dels estadístics de TensorBoard (Figura 9.38) després de l'entrenament complet.



(a) Estadístics Enviroment



(b) Estadístics Losses



(c) Estadístics Policy

Figura 9.38: Tercera prova - Estadístics de TensorBoard

La mitjana de la recompensa en aquesta implementació és aproximadament de 0.7 sobre 1, podent-se observar a la gràfica "Cumulative Reward" de la Figura 9.38a.

A diferència dels anteriors entrenaments, es pot observar en certs estadístics algunes irregularitats a conseqüència de les noves restriccions per a l'agent. Per exemple, en el mateix "Cumulative reward" podem observar com comença al voltant d'una recompensa de -0.9, el que pot indicar com està trobant en primera instància els límits per on es pot moure. Seguidament comença a rebre recompenses positives fins aproximadament els 1.500.000 steps (observem com aprèn gràcies també als pics de la gràfica de "Value Loss" de la Figura 9.38b). Un cop sembla trobar la màxima recompensa, comença un aprenentatge de prova i error on fa petites variacions per tal de millor. Com es pot observar en l'entropia (Figura 9.38c) i en el "Value Loss" (Figura 9.38b), aconsegueix fer petites millores però al final no arriba a aprendre més.

A la Figura 9.39, podem observar un exemple de com mou l'espasa i la col·lisiona contra el cilindre.

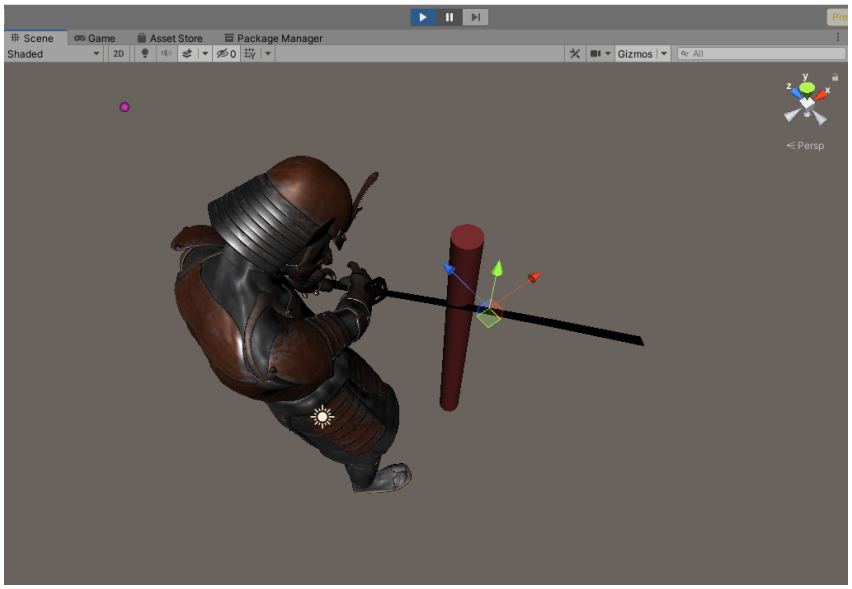


Figura 9.39: Tercera prova - Posició final donada per la xarxa neuronal resultant

9.3.6 Problemes i solucions

Proves amb raytracing

Per simular millor que el defensor mou l'espasa, es va voler afegir una restricció nova que substituís els límits espacials. Es va pensar utilitzar el llançament de raigs. La idea era que des de cada canell del defensor es llances un raig amb una distancia màxima determinada en direcció al maneg de l'espasa. Quan un dels raigs deixés de col·lidir amb el maneg, significaria que la mà ja no arriba a tocar l'espasa, per tant, la recompensa a donar seria de -1.

El problema era que l'aprenentatge no era òptim. Tot i forçar al màxim la distància entre les mans i el maneg, era massa restrictiu i l'agent no arribava a trobar una acció que apropés les espases dintre del rang i en un temps raonable. La mitjana de recompenses arribava a ser de -0.1 sobre 1 en entrenaments de 50 milions de steps, tardant 96 hores. Finalment, es va tornar als límits espacials que ja era suficientment bons per l'objectiu que es busca.

9.4 Quarta implementació: Samurai contra guerrer (Escenari final)

Finalment, en aquesta última implementació s'ha implementat les animacions d'atac, realitzades per un nou GameObject atacant, un model 3D d'un guerrer.

A la Figura 9.40 es pot veure com queda l'entorn final.

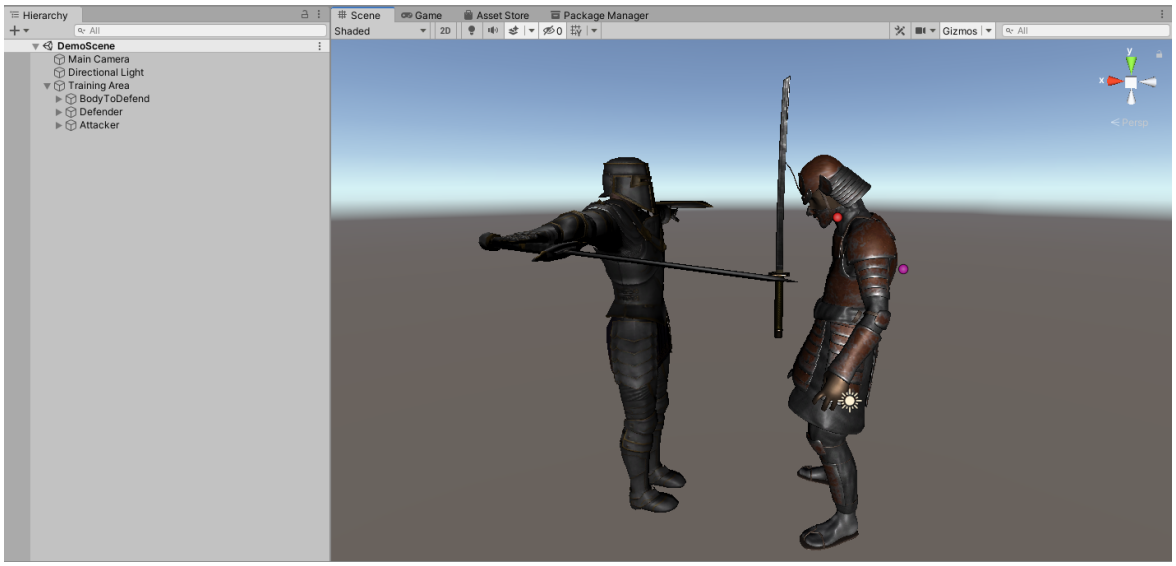


Figura 9.40: Escena Unity "Training Scene" de la quarta implementació

La jerarquia dels GameObjects es manté però canvia la composició del GameObject "Attacker", ja que aquest és més complexa.

9.4.1 Agent

Namespaces

No s'ha afegit ni eliminat cap *Namespace*.

Variables

La variable "Attacker" canvia de referencia a l'espasa enemiga i s'afegeix una altra variable:

- **public Transform attacker**
  Transform del GameObject de l'espasa de l'atacant, "Paladin_j_Nordsttorm_Sword". Conté l'script "MoveAttacker" i el rigidBody de l'espasa. Referenciat mitjançant l'Inspector de Unity.
- **public Transform attackerJoint**
  Transform del GameObject de l'articulació de l'espasa de l'atacant, "Sword_joint". Referenciat mitjançant l'Inspector de Unity.

D'aquesta última s'obté la posició i rotació com a observacions per a l'agent, ja que la posició i rotació de "Paladin_j_Nordsttorm_Sword" no es consistent (el moviment es relatiu a l'articulació de la mà del personatge).

Jerarquia del GameObject

Es manté la configuració de l'espasa.

Components del GameObject

La Figura 9.41 mostra els components que té l'agent (només el referents ML.Agents).

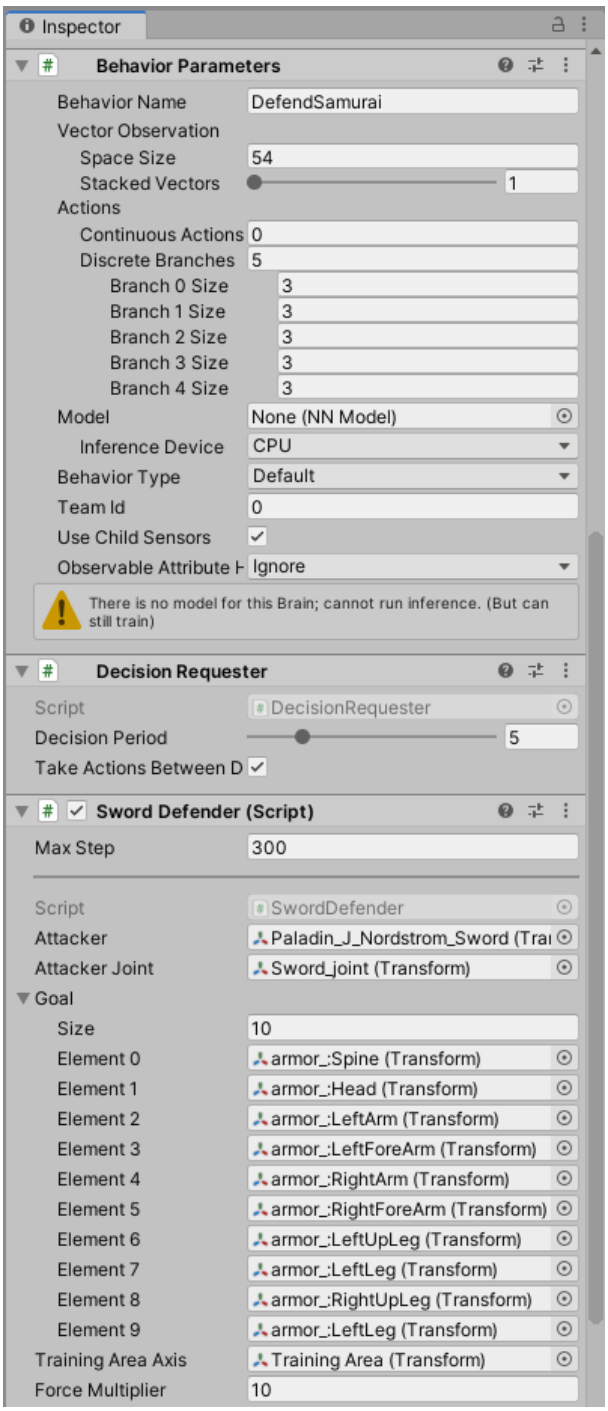


Figura 9.41: Quarta prova - Inspector del GameObject "Defender"

En el component *BehaviorParameters*, el vector d'observació s'ha establit en 54 variables, només apilarem 1 vector d'observacions per FixedUpdate (Stacked Vector) i seguim amb les 5 accions discretes amb 3 opcions per a cada una.

Pel que fa el *DecisionRequester*, augmentem el període de decisions a 5 steps (anteriorment a cada 10).

L'altre canvi ve donat per la variable "Max Step". Fins ara, s'havia indicat que cada episodi tingués una durada màxima indefinida, és a dir, que l'episodi no acabés fins que s'obtingués una recompensa. Ara, si l'agent mou l'espasa de tal manera que l'espasa de l'atacant no col·lisiona amb res, quan passen 300 steps, finalitza l'episodi.

Inici episodi

En aquesta implementació, al haver canviat l'atacant, ara cridarem un mètode propi anomenant *PlayNextAnim()*, el qual s'encarrega de moure i inicialitza l'animació de l'atacant. També mourem a la posició i rotació inicial del rigidBody de l'espasa atacant, per si s'ha mogut al col·lisionar amb l'altra rigidBody de l'espasa defensora.

A la Figura 9.42, es pot veure el codi restant de la inicialització de l'atacant.

```
// Set up an Agent instance at the beginning of an episode
0 referencias
public override void OnEpisodeBegin()
{
    // ... Previous code ...

    // Attacker's initial position and velocity
    attacker.GetComponent<MoveAttacker>().PlayNextAnim();
    attacker.localPosition = new Vector3(-0.735548317f, -1.41793692f, 0.0554279089f);
    attacker.localRotation = new Quaternion(0.000568372896f, 4.47034871e-08f, -5.15475813e-06f, 0.999999881f);
}
```

Figura 9.42: Quarta prova - Mètode *OnEpisodeBegin()* a la classe "SwordDefender"

Observacions

El cilindre ja no existeix. Per tant, ara l'agent voldrà saber la posició de l'espasa atacant i la rotació. De la mateixa manera que amb les parts del cos del samurai, en interessa saber la posició respectiva a l'eix de la "Training Area". Això ho obtenim de la variable "attackerJoint". De la variable "attacker", obtindrem la velocitat a la que va l'espasa que es calcula al script "MoveAttacker".

El codi de la figura 9.43 mostra les noves observacions.

```
// Collect the vector observations of the agent for the step.
// The agent observation describes the current environment from the perspective of the agent.
0 referencias
public override void CollectObservations(VectorSensor sensor)
{
    // ... Previous code ...

    // Attacker's observations
    sensor.AddObservation(trainingAreaAxis.InverseTransformPoint(attackerJoint.position));
    sensor.AddObservation(Quaternion.Inverse(attackerJoint.rotation) * trainingAreaAxis.rotation);
    sensor.AddObservation(attacker.GetComponent<MoveAttacker>().jointVelocity);

    // ... Next Code ...

}
```

Figura 9.43: Quarta prova - Mètode *CollectObservations(VectorSensor)* a la classe "SwordDefender"

Accions

Les accions es mantenen en 5 de discretes. Tot i així, augmentem les restriccions de moviment.

```
// ... Previous code ...

// If the position of the Agents surpase one of this limits, the reward will be -1 and the episode will be over
if (transform.localPosition.x > 2f || transform.localPosition.x < 0.6f ||
    transform.localPosition.y > 9f || transform.localPosition.y < 5f ||
    transform.localPosition.z > 2f || transform.localPosition.z < -1f ||
    transform.rotation.eulerAngles.z < 85f || transform.rotation.eulerAngles.z > 300f)
{
    SetReward(-1f);
    EndEpisode();
}
```

Figura 9.44: Quarta prova - Nous límits en el mètode *OnActionReceived(ActionBuffers)* a la classe "SwordDefender"

Heurística

L'heurística no varia i el moviment en entrenaments heurístics és igual a la implementació prèvia.

9.4.2 Atacant

Namespaces

S'ha afegit el *Namespace* "System.Collections". Aquest implementa diferents mètodes per a realitzar corrutines.

Variables

El Transform "target" és eliminat a conseqüència de que ara l'espasa del atacant es mourà mitjançant una animació, i no mitjançant una força. A més a més, s'afegeixen les següent variables:

- **public Animator attackerAnimator**
  Animator del GameObject de "Attacker". Referenciat mitjançant l'Inspector de Unity.
- **public Transform attackerModel**
  Transform del GameObject "Attacker". Referenciat mitjançant l'Inspector de Unity.
- **public int numAnim**
  Nombre enter que indica l'animació a realitzar.
- **public Transform jointPosition**
  Transform del GameObject "Sword_joint", l'articulació de l'esquelet que mou l'espasa atacant. Referenciat mitjançant l'Inspector de Unity.
- **public Vector3 jointVelocity**
  Vector velocitat a la que es mou l'articulació de l'espasa, "Sword_joint".
- **private bool alreadyHit**
  Boolena que indica si l'espasa ja ha col·lidit amb algun objecte.
- **private Vector3 lastPosition**
  Vector de l'última posició de l'articulació de l'espasa, "Sword_joint".

El mètode *Start()* de Unity canvia, traient la línia de codi del rigidBody i afegint la inicialització de les variables privades.

```
// Start is called before the first frame update
Mensaje de Unity | 0 referencias
private void Start()
{
    numAnim = 0;
    alreadyHit = false;
    lastPosition = jointPosition.position;
}
```

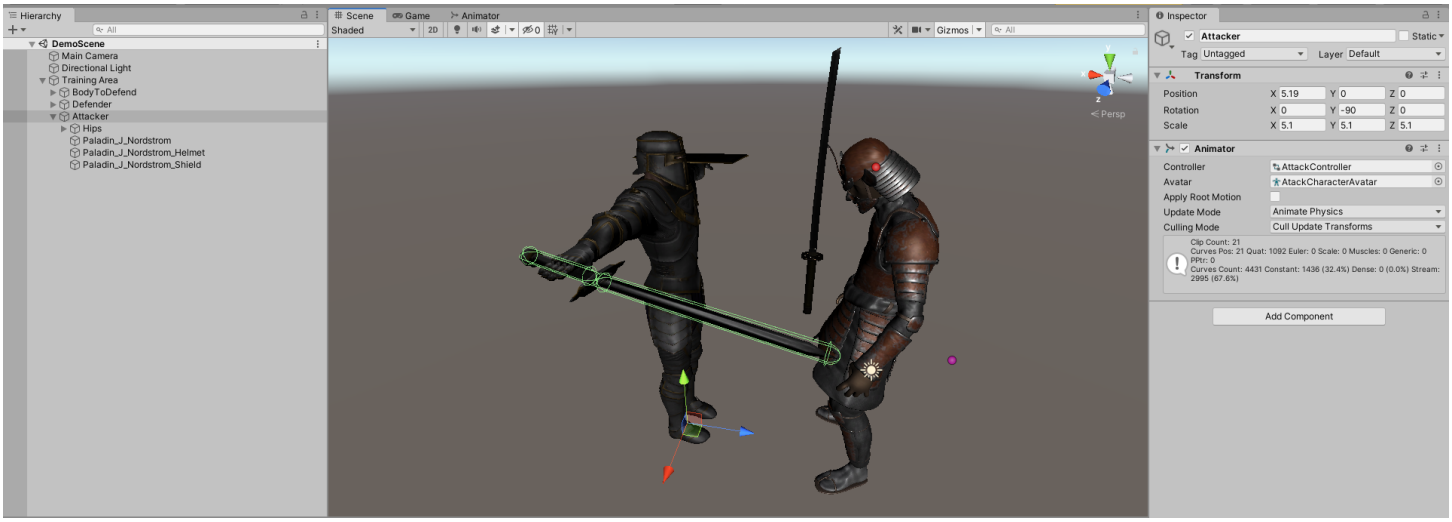
Figura 9.45: Quarta prova - Mètode *Start()* de la classe "MoveAttacker"

Jerarquia del GameObject

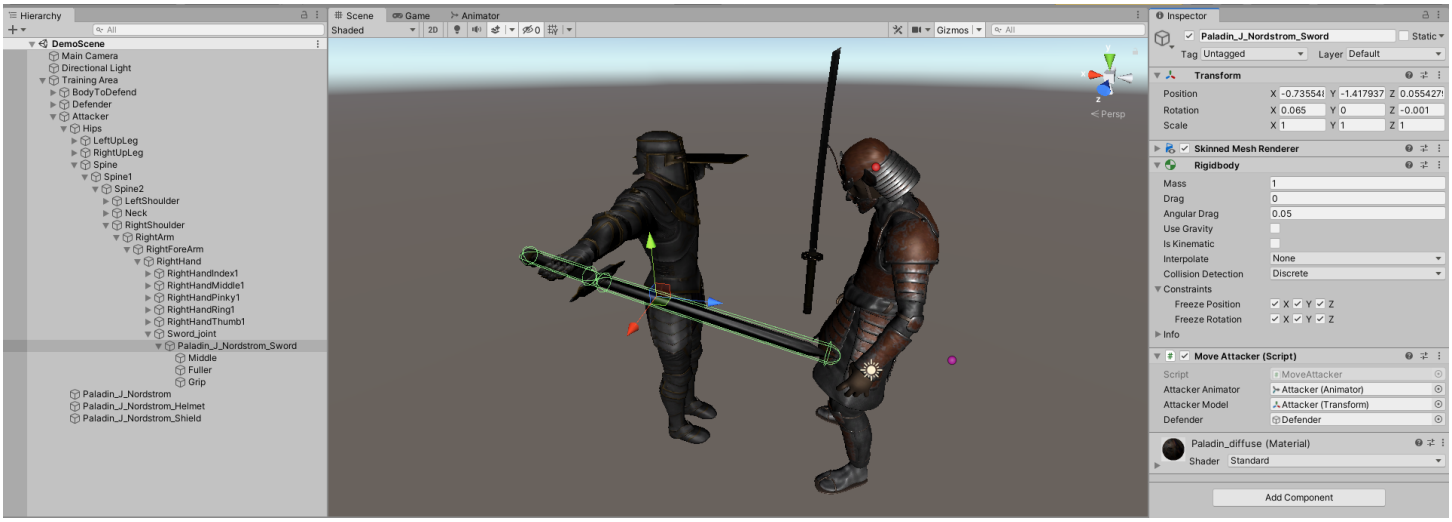
Els fills del GameObject "Attacker" són els mostrats a la Figura 9.46a:

Podem veure la jerarquia sencera de l'esquelet (Hips), i dins d'ella, just a la mà dreta, hi ha l'articulació o *joint* amb el mesh de l'espasa. El GameObject corresponent, "Paladin_J_Nordstrom_Sword" (Figura 9.46b), és el que conté la classe "MoveAttacker" i un rigidBody amb tots els "Constraints" activats, és a dir, que que congela la rotació i el moviment al llarg de tots els eixos del rigidBody (l'espasa ara es mou mitjançant una animació i no el motor de física). Aquest GameObject té tres fills: "Middle" (Figura 9.46c), que marca la posició mitjana de la fulla de l'espasa; "Fuller"(Figura 9.46d), la fulla de l'espasa amb un Tag = "Attacker" i un collider el tipus càpsula; i "Grip" (Figura 9.46e), el maneg de l'espasa amb un Tag = "Grip" i també un collider de tipus càpsula.

Per últim, hi ha els tres meshses de l'armadura: el cos "Paladin_J_Nordstrom", el casc "Paladin_J_Nordstrom_Helmet", i l'escut "Paladin_J_Nordstrom_Shield".

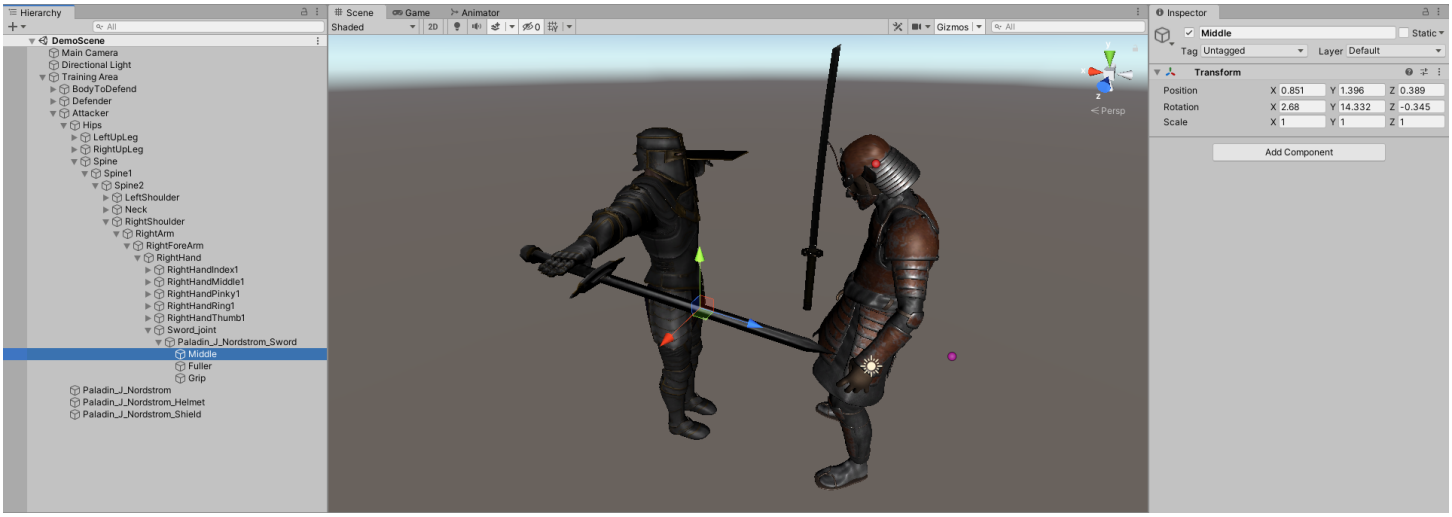


(a) GameObject "Attacker"

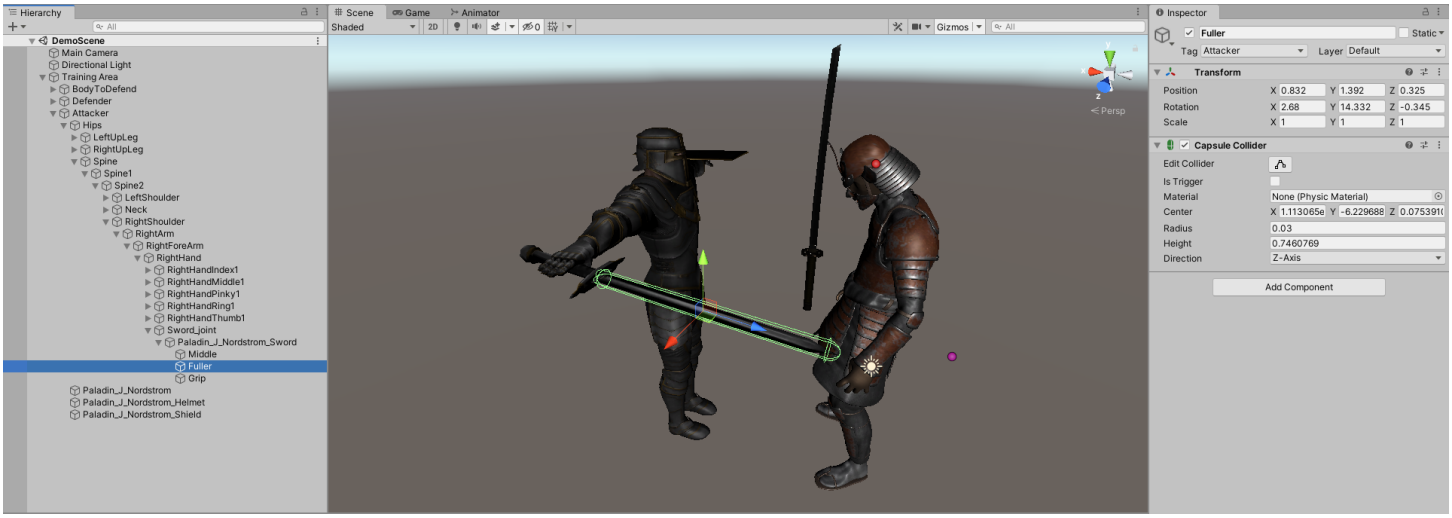


(b) GameObject "Paladin_J_Nordstrom_Sword"

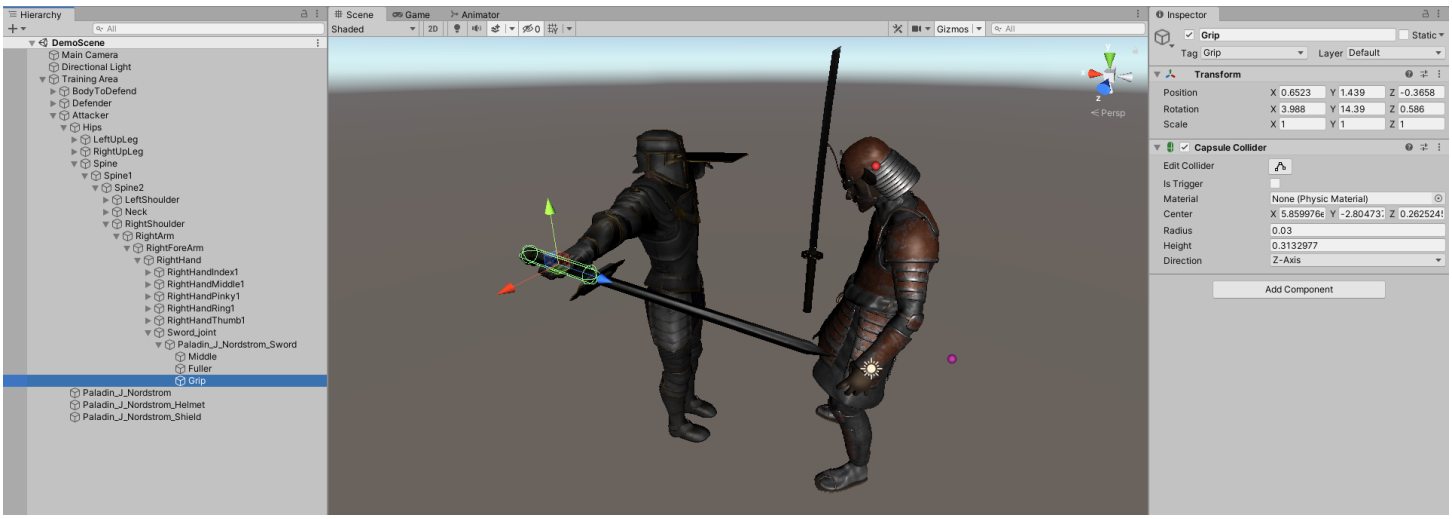
Figura 9.46: Quarta prova - Jerarquia del GameObject "Attacker"



(c) GameObject "Middle"



(d) GameObject "Fuller"



(e) GameObject "Grip"

Figura 9.46: Quarta prova - Jerarquia del GameObject "Attacker" (cont.)

Components del GameObject

El GameObject "Attacker" deixa de tenir l'script de "MoveAttacker" (passa al seu GameObject fill "Paladin_J_Nordstrom_Sword") i se li afegeix un component "Aniamtor". Aquest s'encarrega de tot l'apartat de les animacions.

A la Figura 9.47 es pot observar l'Inspector de l'"Animator Controller" del GameObject "Attacker".

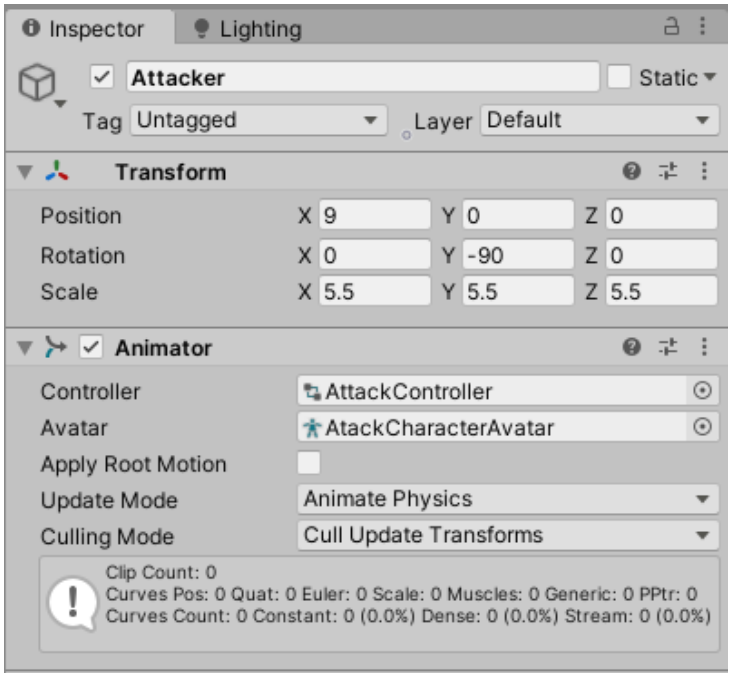


Figura 9.47: Quarta prova - Inspector de "Animator Controller" del GameObject "Attacker"

Hi ha un canvi important a remacar. L'"Update Mode" descriu cada quan s'actualitza l'animació. Com estem treballant amb física, ens interessa establir-ho en el mode "Animate Physics", que actualitza l'animador durant el bucle dels càlculs físics, tenint així el sistema d'animació sincronitzat amb el motor de física.

El grup d'animacions estan guardats com a estats individuals en l'Animator Controller.

A la Figura 9.48 es poden observar totes les animacions disponibles en els diferents States.

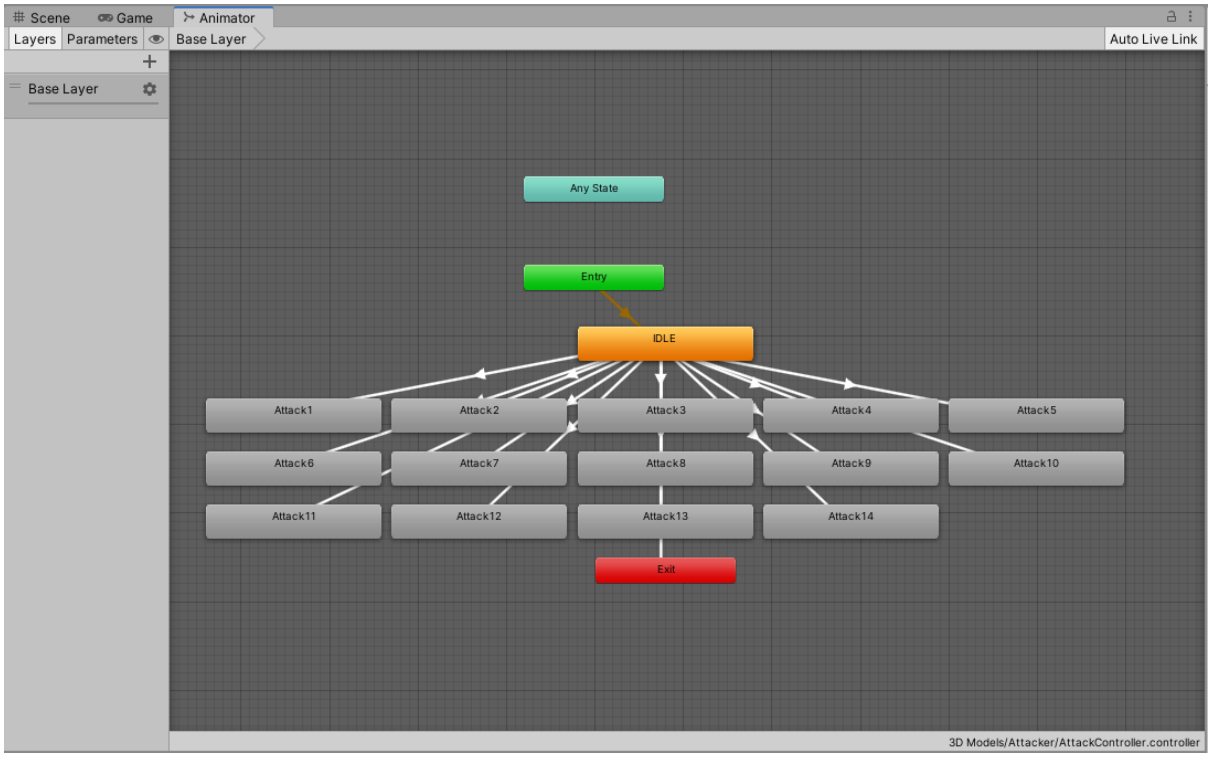


Figura 9.48: Quarta prova - "Animator Controller" del GameObject "Attacker"

Aquesta és una de les pestanyes importants per l'usuari d'aquest sistema, ja que fa referència al cas d'ús "Modificar animacions".

Mètodes

En primer lloc, el nou mètode *PlayNextAnim()* atura la animació en curs amb un mètode del Animator anomenat *Rebind()*. Posteriorment, creem un string anomenat *nameAnim* amb el valor del nom de la següent animació a reproduir (Figura 9.49a), i depenent d'aquest, movem el model sencer del atacant a una distància determinada (Figura 9.49b). A continuació, si era l'ultima animació del conjunt, assignem a "numAnim" un valor de 0 per a tornar a començar per la primera animació. Finalment, reproduïm l'animació de nom *nameAnim* (Figura 9.49c).

```
// Reproduce the next animation for the training
1 referencia
public void PlayNextAnim()
{
    // First, rebind all the animated properties and mesh data with the Animator.
    attackerAnimator.Rebind();

    // Create the information for the next animation
    numAnim++;
    string nameAnim = "Attack" + numAnim;

    // ...
```

(a) Preparar la següent animació

```
// ... Previous code ...

// Move the Attacker according to the animation to play
switch (numAnim)
{
    case 1:
        attackerModel.transform.localPosition = new Vector3(5.8f, 0f, -1.42f);
        attackerModel.transform.rotation = new Quaternion(0, -0.56640625f, 0, 0.824126244f);
        break;
    case 2:
        attackerModel.transform.localPosition = new Vector3(12f, 0f, 0f);
        attackerModel.transform.rotation = new Quaternion(0, -0.707106829f, 0, 0.707106829f);
        break;
    case 3:
        attackerModel.transform.localPosition = new Vector3(7f, 0f, 0f);
        break;
    case 4:
        attackerModel.transform.localPosition = new Vector3(8f, 0f, 0f);
        break;

        // ... Like this until case 14

}

// ...
```

(b) Moure el GameObject "Attacker"

```
    // ...

    // If it's the last animation, begin again
    if (numAnim == 14)
    {
        numAnim = 0;
    }

    // Play the animation
    attackerAnimator.Play(nameAnim);
}
```

(c) Reproduir l'animació

Figura 9.49: Quarta prova - Mètode *PlayNextAnim()* de la classe "MoveAttacker"

En segon lloc, el mètode *OnTriggerEnter(Collider)* (Figura 9.50) se li afegeix una restricció més al comprovar si l'objecte col·lisionat té un Tag == "Goal" o "GoalHand" (s'ha separat els braços de la resta del cos en qüestió de col·lisions). Ara, també s'haurà de comprovar que no hagi col·lisionat l'espasa amb un altre part del cos del samurai o amb l'agent. Per tant, fem servir la variable "alreadyHit" com a controlador. Si es fals, abans de establir la recompensa cambien el seu valor a true.

```
// OnTriggerEnter happens on the FixedUpdate function when two GameObjects collide.
Mensaje de Unity | 0 referencias
public void OnTriggerEnter(Collider other)
{
    if ((other.gameObject.tag == "Goal" || other.gameObject.tag == "GoalHand") && !alreadyHit)
    {
        alreadyHit = true;
        defender.GetComponent<SwordDefender>().SetReward(-1f);
        defender.GetComponent<SwordDefender>().EndEpisode();
        StartCoroutine(WaitEndEpisode());
    }
}
```

Figura 9.50: Quarta prova - Mètode *OnTriggerEnter(Collider)* de la classe "MoveAttacker"

En tercer lloc, implementem la funció *FixedUpdate()* per anar calculant el vector velocitat de l'articulació de l'espasa. El càlcul és simple: fem una resta entre la posició anterior i la posició actual de l'articulació i la dividim pel temps que ha passat amb *Time.fixedDeltaTime*. Finalment, per evitar guardar la velocitat entre la posició anterior i una nova que sigui la d'inici d'una altre animació, mirem que la velocitat no sigui superiro a certs límits.

```
// Called every physics timestep.
Mensaje de Unity | 0 referencias
private void FixedUpdate()
{
    Vector3 actualPosition = jointPosition.position;
    jointVelocity = (actualPosition - lastPosition) / Time.fixedDeltaTime;

    lastPosition = actualPosition;

    if(jointVelocity.x > 100f  || jointVelocity.x < -100f ||
       jointVelocity.y > 100f || jointVelocity.y < -100f ||
       jointVelocity.z > 100f || jointVelocity.z < -100f)
    {
        jointVelocity = Vector3.zero;
    }
}
```

Figura 9.51: Quarta prova - Mètode *FixedUpdate()* a la classe "MoveAttacker"

Un cop s'ha reiniciat l'episodi activem una corrutina, *WaitEndEpisode()*, que fa esperar mig segon abans d'habilitar la llibertat per a col·lisionar (Figura 9.52) de l'espasa. S'ha implementat d'aquesta manera per evitar que en un mateix frame es cridés dos cops aquest mètode per a dos objectes diferents.

```
4 referencias
IEnumerator WaitEndEpisode()
{
    yield return new WaitForSeconds(0.5f);
    alreadyHit = false;
}
```

Figura 9.52: Quarta prova - Mètode *IEnumerator WaitEndEpisode()* de la classe "MoveAttacker"

Finalment, al mètode *OnCollisionEnter(Collision)* se li afegeixen dos noves restriccions: si ja ha hagut una col·lisió i si el GameObject d'aquest col·lisió és la fulla o el maneg de l'espasa atacant. Òbviament, no fem res si ja ha tingut una col·lisió prèvia. Si no en té cap, mirem quin tag té el GameObject de la col·lsió. Si Tag == "Attacker", anem a mirar si ha col·lisionat amb la fulla o el maneg de l'espasa de l'agent (explicat a l'anterior implementació). En canvi, si Tag == "Grip", establim una recompensa de -1 i acabem l'episodi. En qualsevol dels dos casos, un cop es finalitza un episodi, s'inicialitza la corrutina *WaitEndEpisode()*.

A la Figura 9.53 s'ha explicat el codi que implementa la teoria anterior.

```
// OnCollisionEnter is called when this collider/rigidbody has begun touching another rigidbody/collider.
Mensaje de Unity | 0 referencias
public void OnCollisionEnter(Collision collision)
{
    // Check if the collider has been already activated
    if (!alreadyHit)
    {
        // Check which collider has been activated.
        Collider myCollider = collision.GetContact(0).thisCollider;

        // If it has collided with the grip of the attacker's sword, bad
        if (myCollider.gameObject.tag == "Grip")
        {
            alreadyHit = true;
            defender.GetComponent<SwordDefender>().SetReward(-1);
            defender.GetComponent<SwordDefender>().EndEpisode();
            StartCoroutine(WaitEndEpisode());
        }
        // If it has collided with the fuller of the attacker's sword, good
        else if (myCollider.gameObject.tag == "Attacker")
        {
            alreadyHit = true;

            // Check if the cilinder has collided with the grip or the fuller of the sword
            float distanceGrip = Vector3.Distance(defender.transform.GetChild(1).position, collision.GetContact(0).point);
            float distanceFuller = Vector3.Distance(defender.transform.GetChild(2).position, collision.GetContact(0).point);

            // If it has collided with the grip, bad
            if (distanceGrip <= distanceFuller)
            {
                defender.GetComponent<SwordDefender>().SetReward(-1);
                defender.GetComponent<SwordDefender>().EndEpisode();
                StartCoroutine(WaitEndEpisode());
            }
            // If it has collided with the fuller, good
            else
            {
                float distanceDefender = Vector3.Distance(collision.GetContact(0).point, collision.gameObject.transform.GetChild(0).position);
                float distanceAttacker = Vector3.Distance(collision.GetContact(0).point, transform.position);

                float rewardDefender = (1f - (Mathf.Clamp(distanceDefender, 0f, 3f) / 3f)) / 2f;
                float rewardAttacker = (1f - (Mathf.Clamp(distanceAttacker, 0f, 3f) / 3f)) / 2f;

                float totalReward = rewardAttacker + rewardDefender;

                defender.GetComponent<SwordDefender>().SetReward(totalReward);
                defender.GetComponent<SwordDefender>().EndEpisode();
                StartCoroutine(WaitEndEpisode());
            }
        }
    }
}
```

Figura 9.53: Quarta prova - Mètode *OnCollisionEnter(Collision)* de la classe "MoveAttacker"

9.4.3 Objectiu

Els avantbraços ("armor_:LeftForeArm" i "armor_:RightForeArm") canvien de Tag a "GoalHand", per evitar que es tracti la penalització de l'agent en rotar l'espasa fent servir la col·lisió amb els braços.

9.4.4 Fitxer de configuració YAML

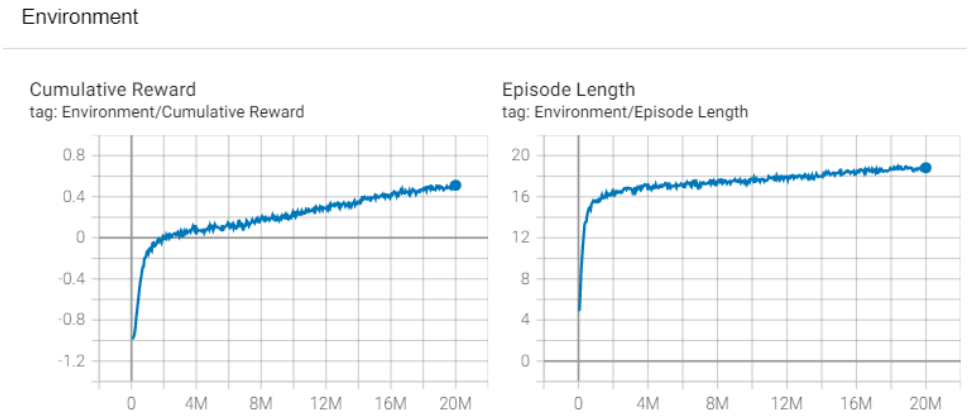
Després de provar diferents configuracions, la millor ha estat la mateixa que en l'anterior prova. La única diferencia és en el número de steps totals de l'entrenament (20.000.000 steps), ja que l'agent necessita de més entrenaments per aprendre els nous límits i com parar els atacs realitzats per les animacions. A la Figura 9.54 podem observar aquests canvis.

```yaml
behaviors:
  DefendSamurai:
    trainer_type: ppo
    hyperparameters:
      batch_size: 512
      buffer_size: 12000
      learning_rate: 0.0003
      beta: 0.005
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      learning_rate_schedule: linear
    network_settings:
      normalize: true
      hidden_units: 512
      num_layers: 3
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    keep_checkpoints: 5
    max_steps: 30000000
    time_horizon: 1000
    summary_freq: 50000
```

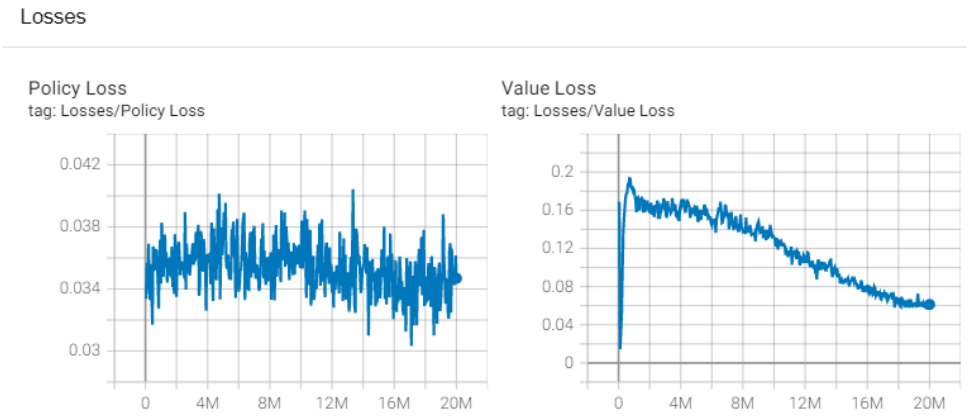
Figura 9.54: Quarta prova - Paràmetres del fitxer de configuració YAML

9.4.5 Resultats

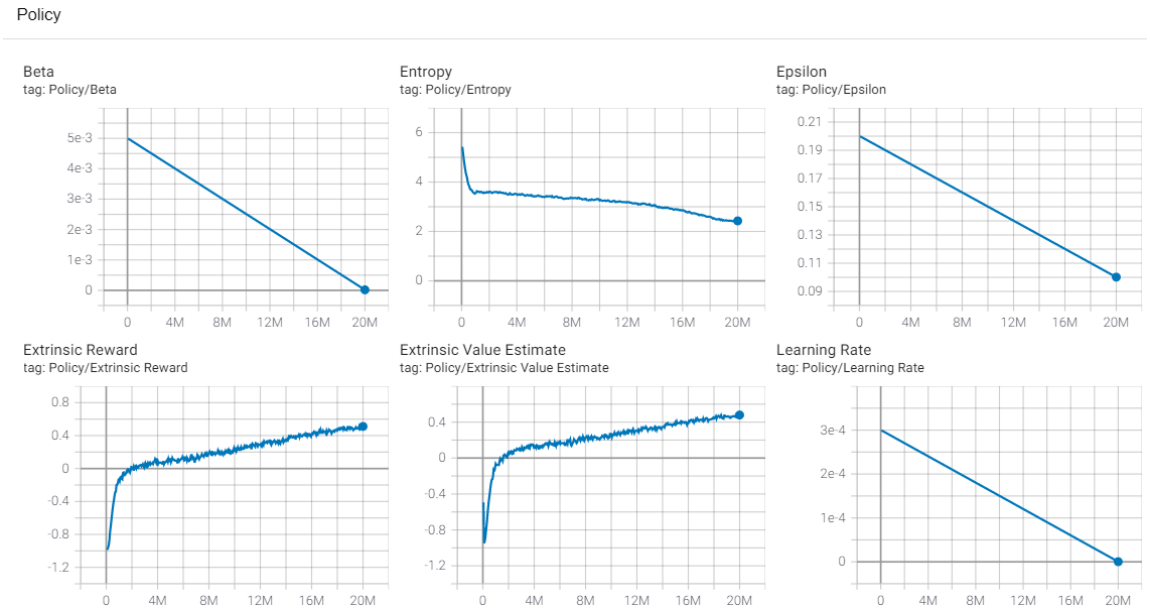
Gràfiques dels estadístics de TensorBoard (Figura 9.55) després de l'entrenament complet.



(a) Estadístics Enviroment



(b) Estadístics Losses



(c) Estadístics Policy

Figura 9.55: Quarta prova - Estadístics de TensorBoard

A la figura 9.55a es pot observar com la mitja de recompensa després de 20 milions d'steps és aproximadament 0.55 sobre 1. Com es pot veure, la tendència indica que aquest valor encara pot millorar. Les dades que ens ho indican són el valor major que 0 a la gràfica "Value Loss" (Figura 9.55b) i a la gràfica "Entropy" (Figura 9.55c).

El problema ve donat en la duració de l'entrenament. Un dels objectius de realitzar d'aquesta manera les animacions era reduir al màxima la duració. En aquest cas, realitzar els anteriors steps esmentats té una duració de 12 hores. Per tant, considerem que el valor ja es suficient. A més a més, com es pot observar a la demo, els moviments són acceptables.

A l'Apartat 10: Resultats, es pot observar unes imatges resultants a l'entrenament i una anàlisis complet d'elles.

9.4.6 Problemes i solucions

Número d'animacions
Les animacions d'atac pensades per a realitzar el projecte formaven un conjunt de 20. A més a més, es volia fer que aquestes comencessin des de diferents posicions, de la mateixa manera que havia funcionat en les anteriors implementacions. Desafortunadament, en certes animacions l'agent tardava un temps bastant elevat (de 8 a 10 hores) en aprendre un moviment amb recompensa positiva. De la mateixa manera, les posicions inicials aleatòries, també hi perjudicaven, fent que el temps s'incrementes i, fins i tot, impedint que l'agent trobés algun moviment amb recompensa positiva. Per aquesta raó, es va decidir eliminar totes aquelles animacions massa complexes i es va codificar una única posició inicial per a cada animació. Això va permetre reduir en gran manera el temps d'entrenament (12 a 16 hores les 14 animacions) i que totes elles tinguessin un moviment vàlid i amb recompensa positiva.

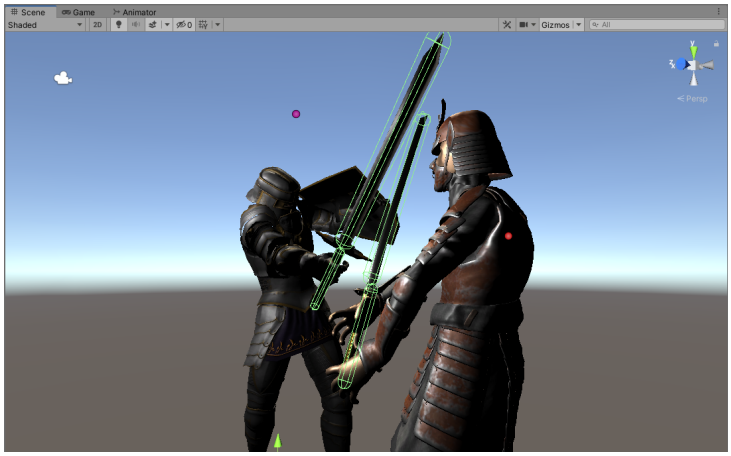
Observacions necessàries
Al moure l'espasa atacant amb les animacions, es va treure la velocitat del rigidBody com a observació en el vector d'observacions de l'agent, ja que ara només seguiria al GameObject sense variar en la velocitat. Al fer-ho, augmentava el temps d'aprenentatge fins el punt que l'agent no aprenia un moviment amb recompensa positiva. Per arreglar-ho, es va decidir fer el càlcul de la velocitat de la manera explicada anteriorment, tornant a tenir el mateix nombre d'observacions i, més important, veient com aprenia de nou amb valors positius a la recompensa.

Velocitat animacions
Tot i tenir a les animacions sincronitzades amb el motor de física, aquestes es movien massa ràpid, fent que l'espasa atacant travesses l'espasa defensora. Això feia que moviments que visualment i heurísticament eren vàlids, com que no detectava la col·lisió, no rebien una recompensa positiva. Per arreglar aquest problema, es va reduir a 0.4 la velocitat de les animacions, creant així més frames intermedis per animació que si col·lidien amb l'espasa defensora.

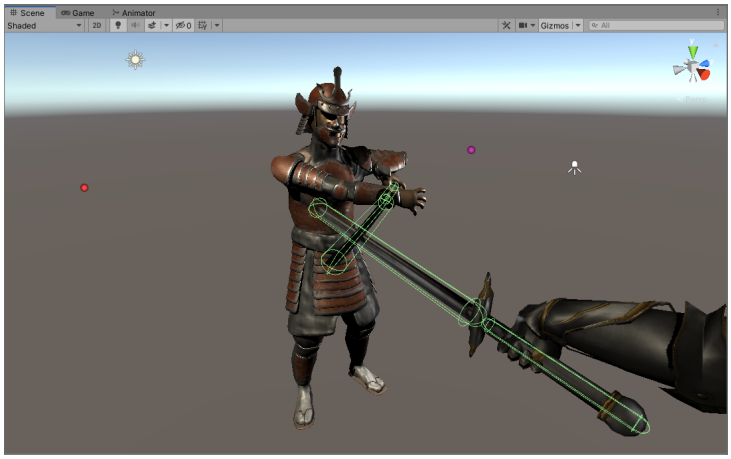
10 Resultats

10.1 Imatges a l'entrenament

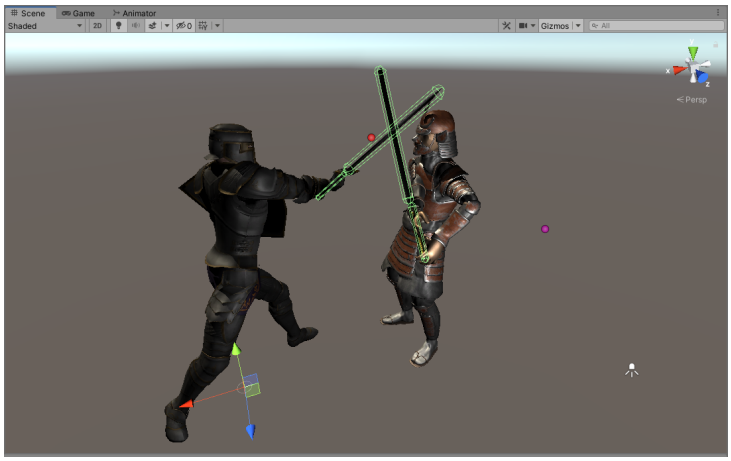
A la Figura 10.1 de les següent pàgines es pot observar els resultats de totes les 14 animacions.



(a) Animació "Attack1"

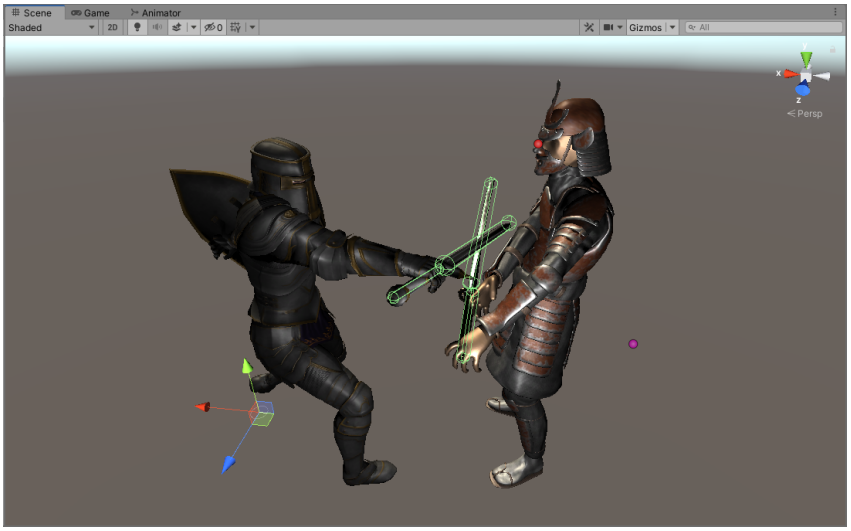


(b) Animació "Attack2"

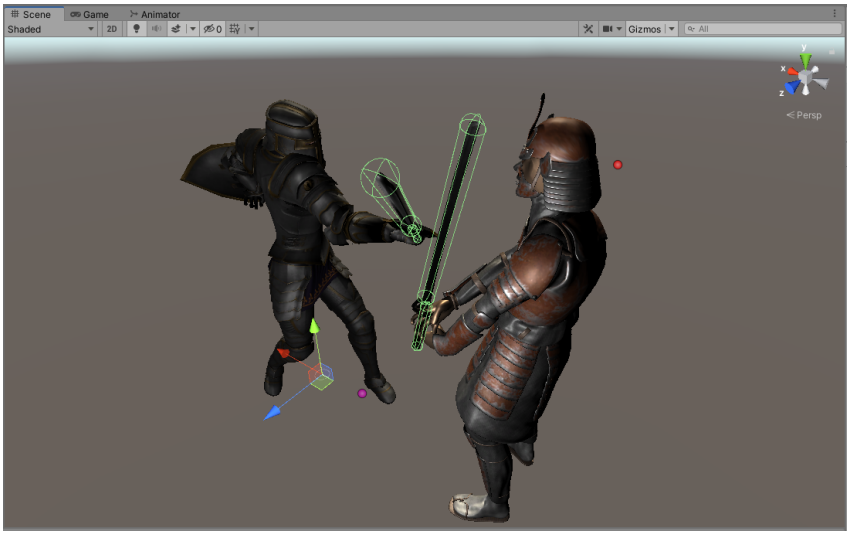


(c) Animació "Attack3"

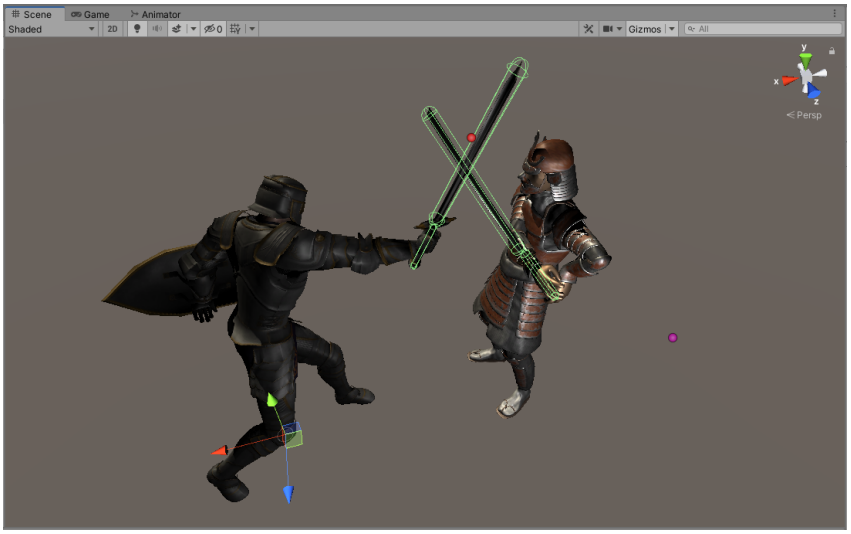
Figura 10.1: Quarta prova - Resultats de l'agent respecte les animacions d'atac



(d) Animació "Attack4"

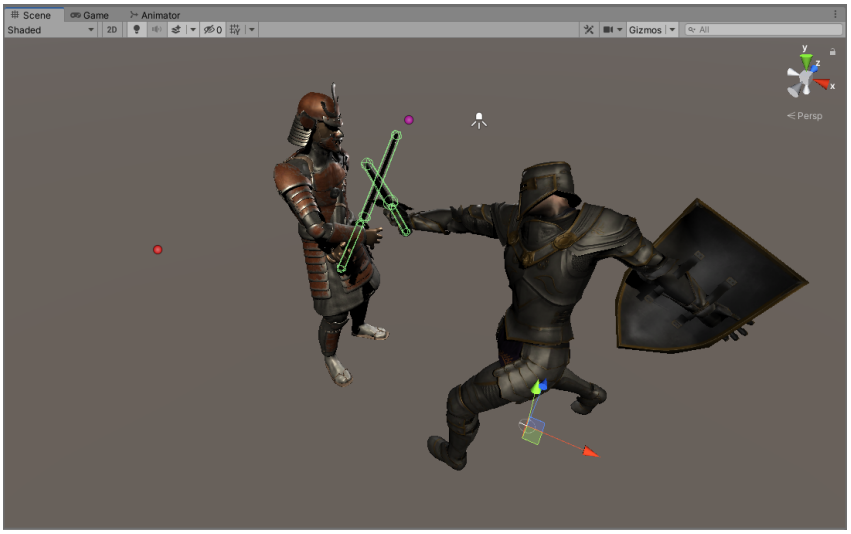


(e) Animació "Attack5"

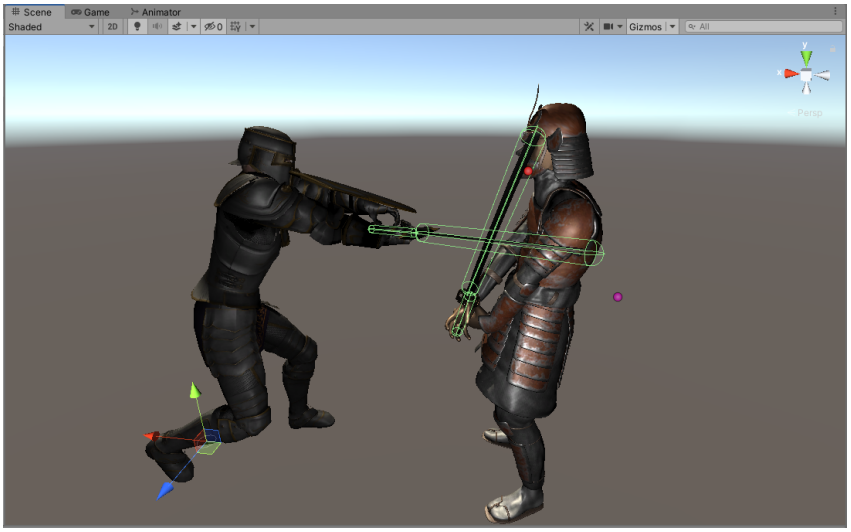


(f) Animació "Attack6"

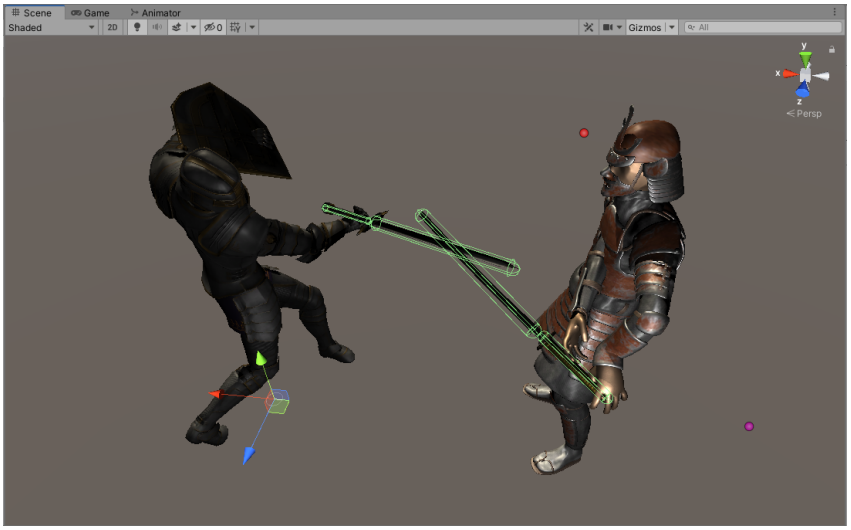
Figura 10.1: Quarta prova - Resultats de l'agent respecte les animacions d'atac (cont.)



(g) Animació "Attack7"

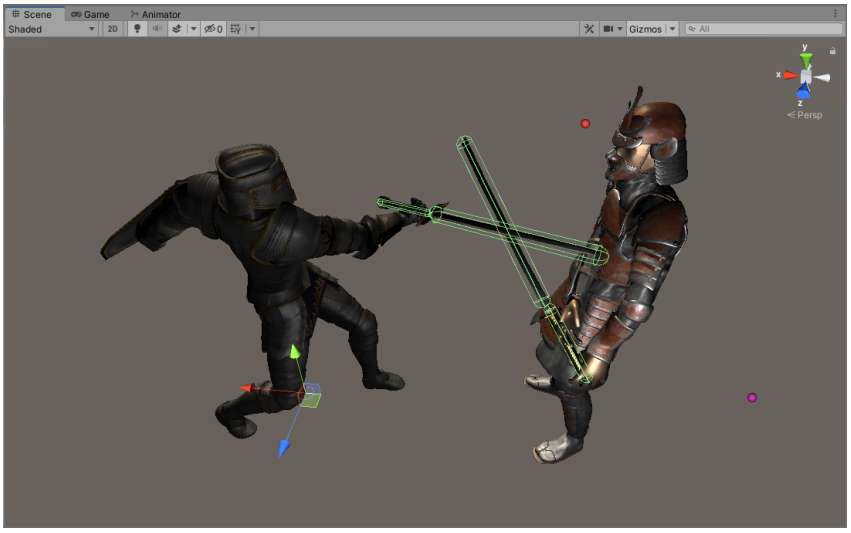


(h) Animació "Attack8"

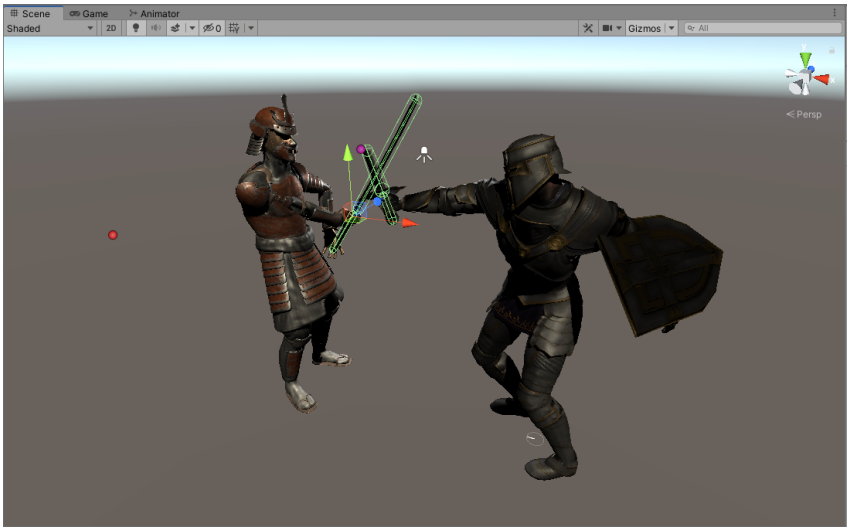


(i) Animació "Attack9"

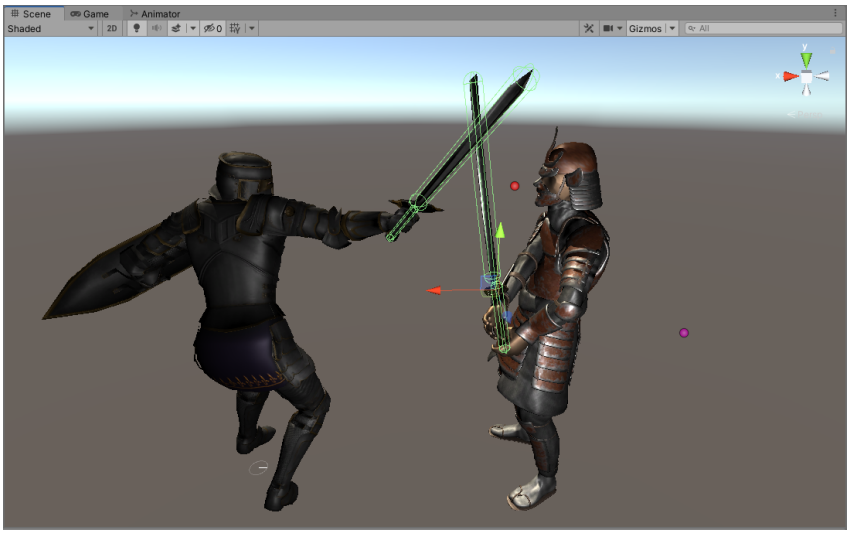
Figura 10.1: Quarta prova - Resultats de l'agent respecte les animacions d'atac (cont.)



(j) Animació "Attack10"

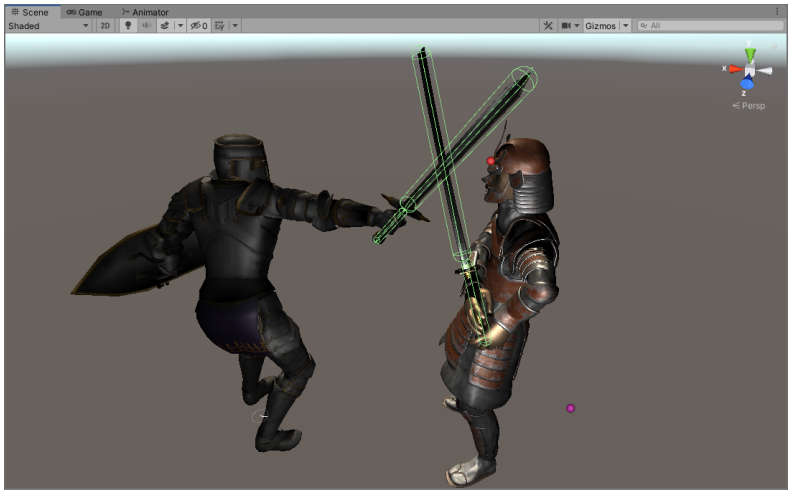


(k) Animació "Attack11"

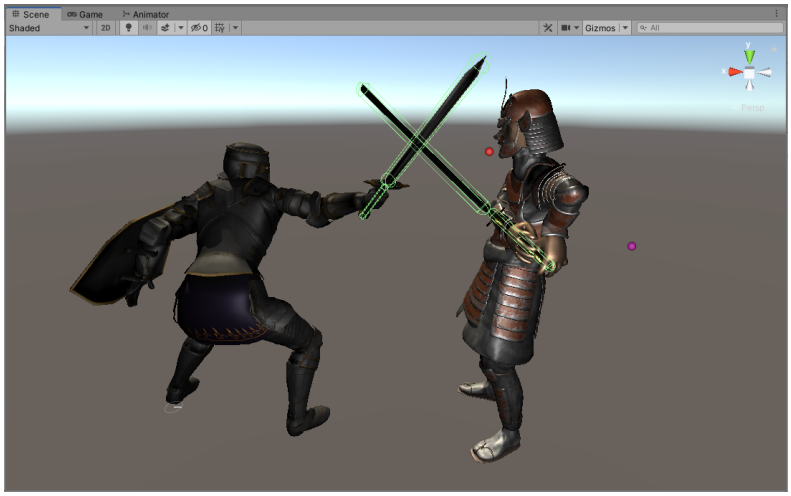


(l) Animació "Attack12"

Figura 10.1: Quarta prova - Resultats de l'agent respecte les animacions d'atac (cont.)



(m) Animació "Attack13"



(n) Animació "Attack14"

Figura 10.1: Quarta prova - Resultats de l'agent respecte les animacions d'atac (cont.)

Les imatges anteriors mostren els resultats amb una mitja entre tots de 0.86 sobre 1, a diferència de l'observat al Tensorboard. Això és degut a que la xarxa neuronal, en algunes instàncies, sobrepassa els límits posicionals, fent que la mitja disminueixi fins el 0.5. Per aquesta raó, a la demo final, en algunes ocasions, l'animació del defensor fallarà. Tot i això, s'ha decidit ignorar-ho perquè, per probabilitat, succeïx pocs cops.

10.1.1 Anàlisi i comparativa amb defenses reals

Com s'havia establit amb anterioritat, les animacions resultants són simples però efectives. La xarxa neuronal busca sempre els moviments més ràpids i efectius, tot i que no sempre ho aconsegueix, ja que en algunes ocasions fa moviments innecessaris. Tot i això, si es fa una comparativa amb la realitat, es pot veure les similituds amb l'art del combat amb espasa.

Els resultats que s'han obtingut es poden comparar amb certes guàrdies o postures defensives de diferents escoles de combat provinents de diferents nacions d'Europa i d'algunes arts marcials japoneses. Aquest breu anàlisis es centrarà en l'escola alemanya, italiana i l'art marcial japonesa anomenada *Kenjutsu*.

Pflug / Posta Breve / Chudan No Kamae

La gran majoria d'animacions són atacs laterals per l'esquerra del defensor. Això fa que l'agent adopti una guàrdia a mitjana alçada, tot movent el braç dret per davant del cos (Figura 10.2).

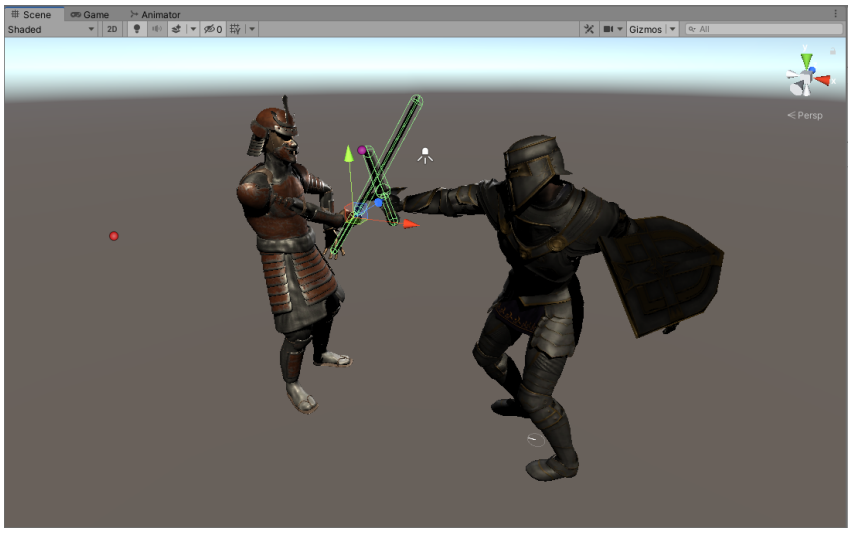


Figura 10.2: Animació "Attack11" a comparar

A l'escola alemanya a aquesta guàrdia s'anomena "Pflug" o "Plow". A la Figura 10.3 es pot veure un parell de persones realitzant-la.

(a) Pflug 1 [8]

(b) Pflug 2 [9]

Figura 10.3: Postura "Pflug"

A l'escola italiana, hi ha una semblant anomenada "Posta Breve", significant "Guàrdia Curta", i al Kenjutsu, una anomenada "Chudan no kamae", significant "Guàrdia de nivell mitjà". La Figura 10.4 mostra la postura japonesa.

Figura 10.4: Postura "Chudan no kamae" [10]

Langort / Posta Bicorno / Te ura gasumi
Hi ha alguns resultats on l'espasa va cap a l'axil·la del defensor, tot apuntant amb l'espasa cap a l'enemic (Figura 10.5).

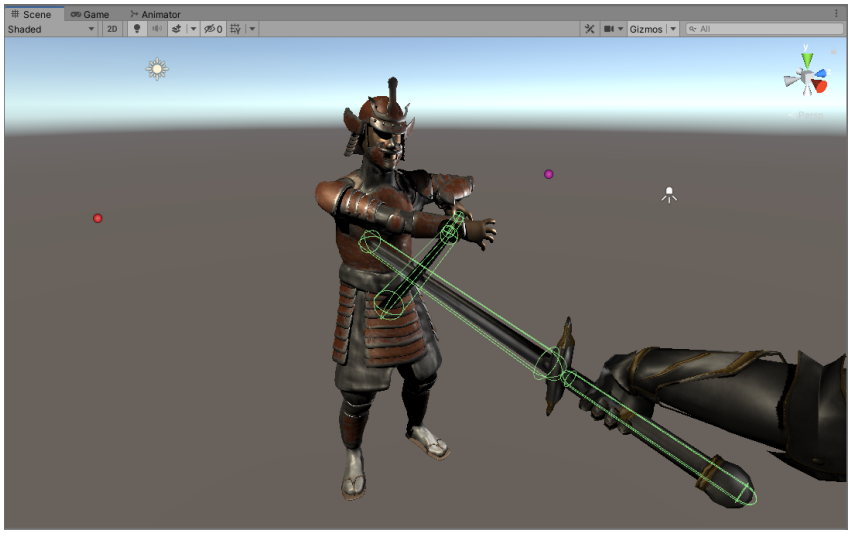
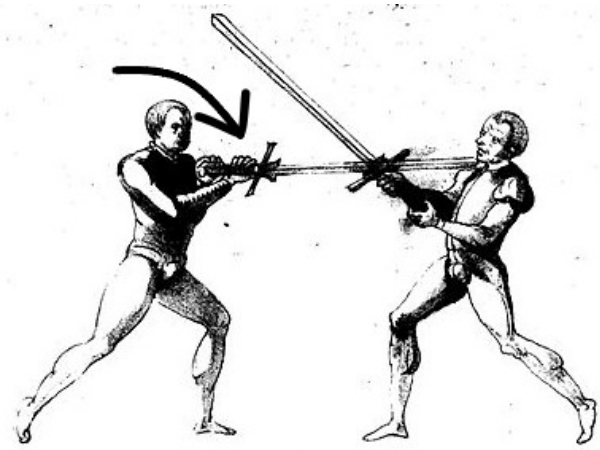


Figura 10.5: Animació "Attack2" a comparar

A l'escola alemanya a aquesta guàrdia s'anomena "Langort". La Figura 10.6 mostra una persona en aquesta postura.

Figura 10.6: Postura "Langort" [8]

A l'escola italiana, es troba la postura "Posta Bicorno". La Figura 10.7 mostra un parell d'exemples de com es veu.



(a) Bicorno 1 [11]

(b) Bicorno 2 [9]

Figura 10.7: Postura "Posta Bicorno"

Finalment, a l'art marcial Kenjutsu hi ha una postura anomenada "Te ura gasumi", la qual té bastant similitud. La Figura 10.8 mostra un dibuix que ho exemplifica.

Figura 10.8: Postura "Te ura gasumi" [12]

Conclusions
Després d'haver vist les semblances i diferencies entre els resultats i les postures reals, es pot concloure que l'objectiu de l'agent és semblant o pot arribar a semblar-se al que es buscava quan es van crear aquestes postures. Sense tenir en compte la certa divergència entre les animacions i les guàrdies, es pot deduir que ha d'haver-hi una connexió entre les dues, és a dir, l'agent troba de manera natural certes maneres de moure i posicionar l'espasa que ja van pensar humans fa segles. Es pot argumentar que la xarxa neuronal no deixa d'imitar un cervell humà i, en aquest cas, intenta imitar el cervell d'aquells esgrimistes de fa centenars de generacions. Tot i això, si realment fos aquest l'objectiu del projecte, es crearien més regles i limitacions que, sense cap dubte, farien que en uns diversos dies el personatge defensor es comportes com un mestre de les arts de la guerra.

10.2 Demo

La demo per a comprovar el funcionament de la xarxa neuronal resultant està formada per un escenari japonès ("Tatami"), unes fulles que cauen d'un cirerer de flor japonès ("Particle System"), una música de fons ("M Audio Source") i una petita brisa ("BA Audio Source"), un canvas ("Canvas") amb els botons necessàries per realitzar les animacions ("Canvas"), i els dos personatges ("Combat Area"), els quals comencen amb els scripts corresponents desactivats.

A la Figura 10.9 es pot observar els GameObjects comentats en l'anterior paràgraf.

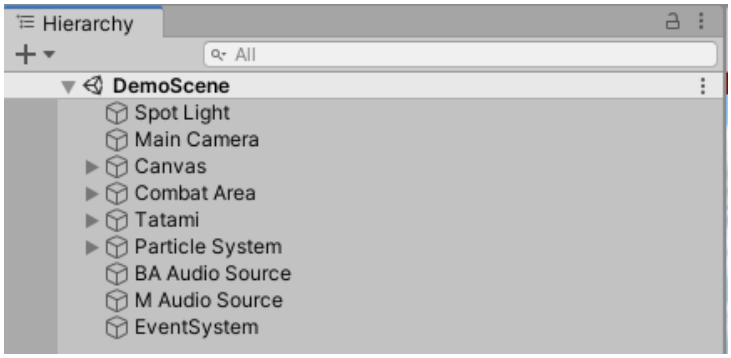


Figura 10.9: Jerarquia de la "Demo Scene"

10.2.1 Escena

Per al funcionament de la demo, s'ha creat un nou script, de nom "HandlerDemo", que s'encarrega d'indicar tant al canvas com als personatges quan iniciar una animació i quan parar. Aquest està adjuntat al canvas de l'escena (podria estar a un GameObject Empty, però es troba més coherent que estigui al canvas, ja que el modifica).

Canvas
Format per dos panells:

1. **StartPanel:** Panell que conté un botó, "StartButton", per iniciar la demo i passar al següent panell i seleccionar una animació a provar.
2. **GamePanel:** Panell que conté un botó, "ExitButton", per tancar l'aplicació, un desplegable, "Dropdown", amb la llista d'animacions d'atac que l'agent pot defensar, i un text, "SelectText" que t'indica que escullis una animació d'aquest desplegable.

La Figura 10.10 mostra els dos panells. Les fonts usades son Harukaze[2] , de Nug's Project, i Kashima Brush[3], de Abo Daniel Studio. Ambdues provenen de la pàgina web "DaFont".

[2] https://www.dafont.com/es/harukaze.font
[3] https://www.dafont.com/es/kashima-brush.font

(a) StartPanel

(b) GamePanel

Figura 10.10: GameObject "Canvas" i el seus panells

Tatami

De la pàgina web Sketchfab s'ha obtingut l'escenari "Flying Island - Low Poly"[4], de "ChojoThePony", el qual ha estat editat per a formar un tatami on combaten els personatges.

A la Figura 10.11 es pot observar l'escenari editat.

[4] https://sketchfab.com/3d-models/flying-island-low-poly-8780067653804460a0c792c91ed2bbf5

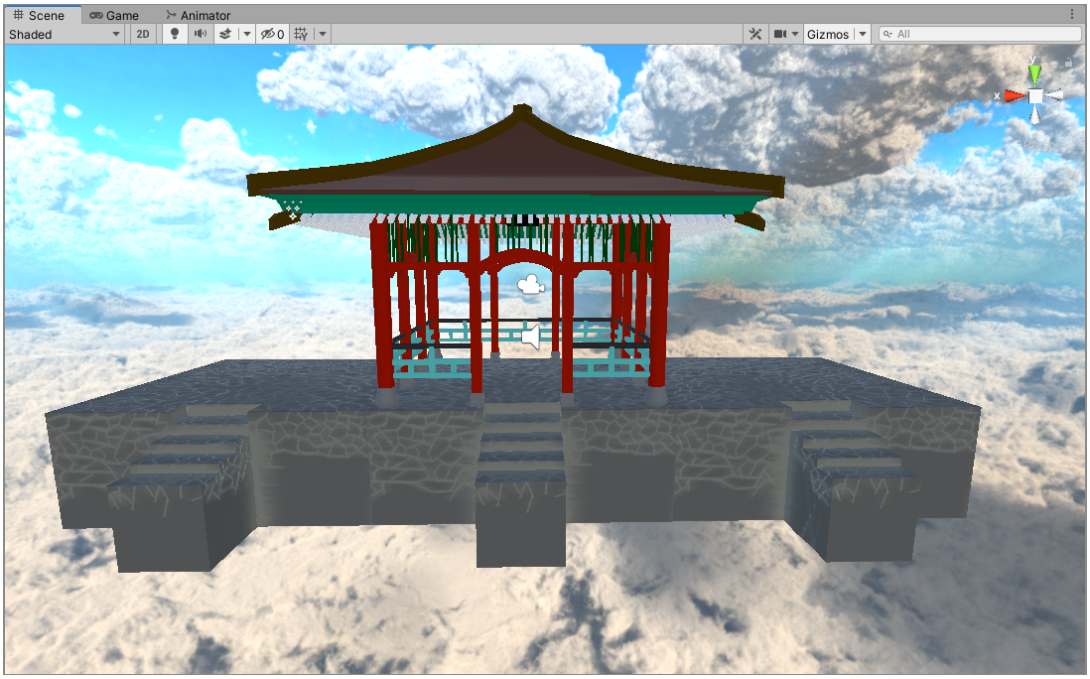


Figura 10.11: "Flying Island - Low Poly" editat

Flors (Particle System)

Mitjançant el sistema de partícules que té Unity, s'ha creat una simulació de la caiguda de flors d'un cirerer japonès. Donant-li gravetat i una mica de vent (tot mitjançant l'Inspector del sistema de particules,) i una textura de flor amb el color correcte, sembla que cauen d'un arbre proper (Figura 10.12).

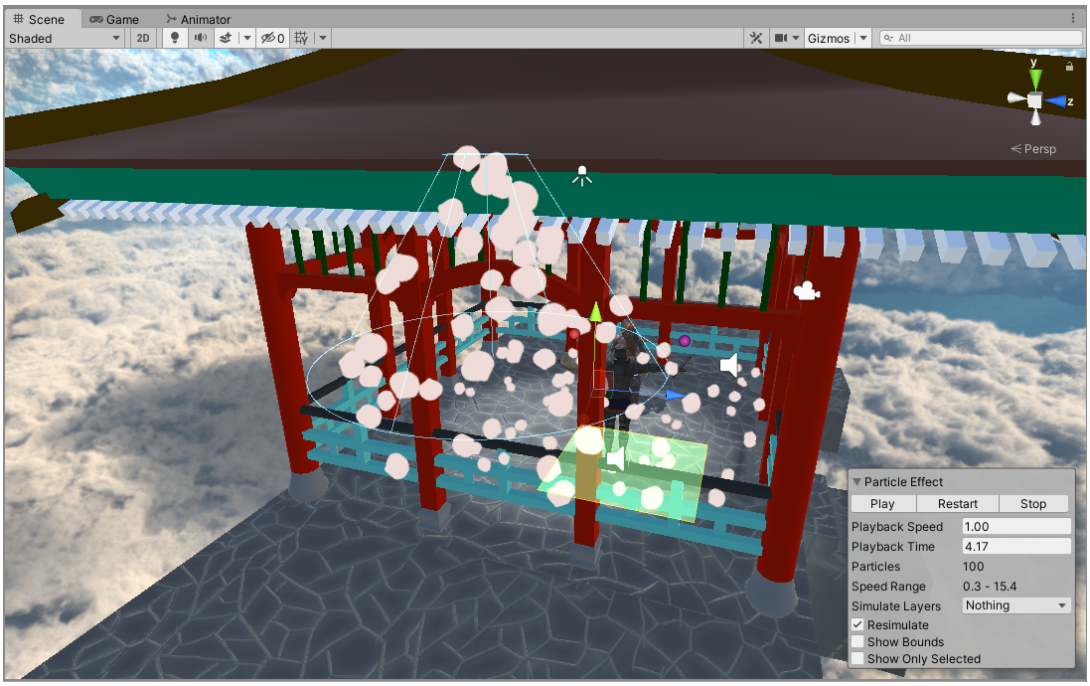
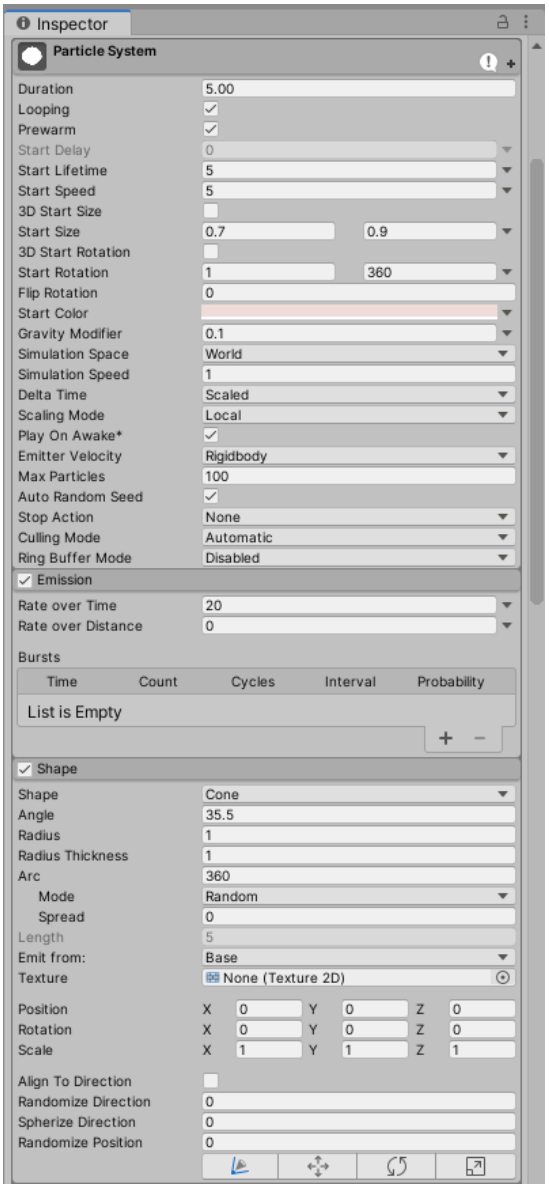


Figura 10.12: Sistema de partícules "Particle System"

La textura de la fulla prové de la pàgina web "textures.com"[5]. La Figura 10.13 mostra la seva forma.

Figura 10.13: Textura "Leaf" del sistema de partícules "Particle System"

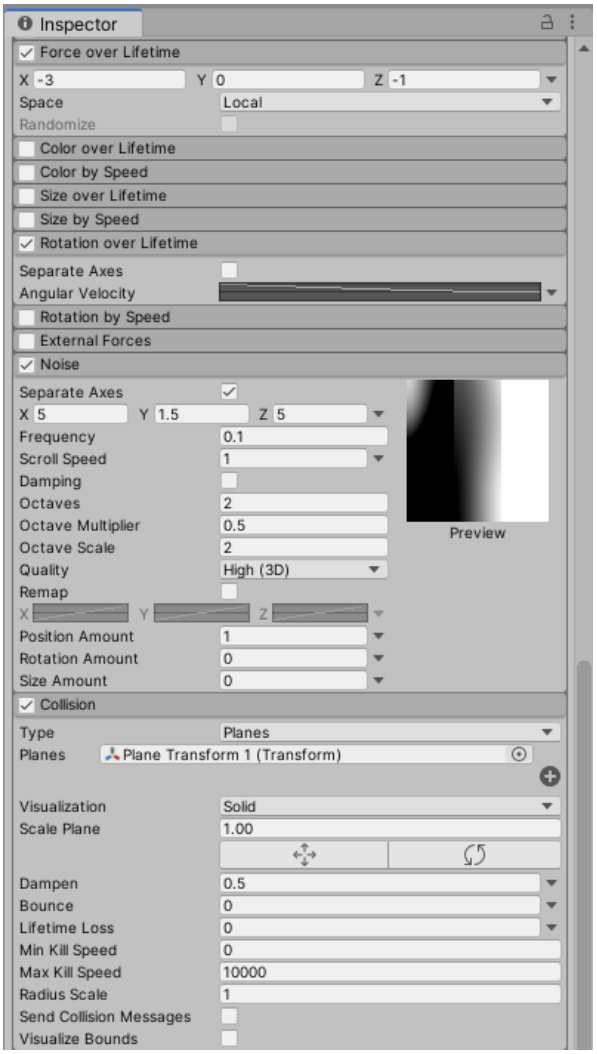
La Figura 10.14 mostra la configuració del sistema de partícules de les fulles.



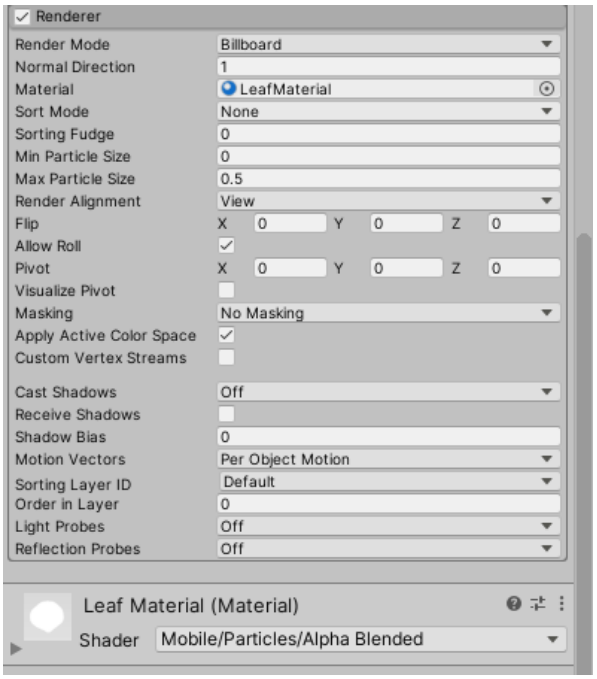
(a) "Particle System" 1

Figura 10.14: Inspector del sistema de partícules "Particle System"

[5] https://www.textures.com/download/image/29264



(b) "Particle System" 2



(c) "Particle System" 3

Figura 10.14: Inspector del sistema de partícules "Particle System" (cont.)

Àudio
S'ha afegit tres sons a l'escena mitjançant el sistema d'àudio de Unity. El component que ho permet, anomenat "Audio Source", ha estat adjuntat a tres GameObjects:

1. **M Audio Source:** La música de fons, Honō no Megami, és creada per Adrian von Ziegler[6].
2. **BA Audio Source:** Efecte de so de vent anomenat "Wind In Pine Trees" provinent de la YouTube Audio Library.
3. **Paladin_J_Nordstrom_Sword:** Al GameObject que té adjuntat l'script "MoveAttacker" se li afegeix un so de col·lisió d'espases per a que soni quan succeeixi una.

Als "Audio Source" se'ls pot assignar una taula mescladora virtual de música anomenada "AudioMixer" per poder tractar el volum i separar per pistes les diferents tipologies d'àudio que hi ha. En el cas d'aquesta demo, en ser tan curta, s'ha decidit no fer cap mena de tractament del so més enllà d'ajuntar-los tots en únic "AudioMixer" de nom "DemoAudioMixer".

Skybox
El fons de l'escenari està format per un material anomenat "DayInTheClouds" que forma part d'uns recursos gratis de la tenda oficial de Unity[7].

Script
L'script "HandlerDemo" té les següent variables:

- **public GameObject panelGame**
  GameObject del panell inicial que conté el botó "StartButton". Referenciat mitjançant l'Inspector de Unity.
- **public GameObject panelStart**
  GameObject del panell amb el botó "ExitButton", el text "SelectText" i el drop-down "Dropdown" de les animacions. Referenciat mitjançant l'Inspector de Unity.
- **public Button startButton**
  Button per a començar a seleccionar una animació. Referenciat mitjançant l'Inspector de Unity.
- **public Button exitButton**
  Button per a sortir de l'aplicació. Referenciat mitjançant l'Inspector de Unity.
- **public Dropdown animationDropdowm**
  Dropdown amb la llista d'animacions per a seleccionar. Referenciat mitjançant l'Inspector de Unity.
- **public GameObject attacker**
  GameObject de l'atacant que té l'script "MoveAttacker", "Paladin_J_Nordstrom_Sword". Referenciat mitjançant l'Inspector de Unity.
- **public GameObject defender**
  GameObject de l'agent que té l'script "SwordDefender", "Defender". Referenciat mitjançant l'Inspector de Unity.

[6] https://www.youtube.com/watch?v=Ix6aZ6q8aBc
[7] https://assetstore.unity.com/packages/2d/textures-materials/sky/skybox-series-free-103633

- **public bool startAnimation**
  Booleà de control per a saber quan realitzar una animació i quan parar.

Mitjançant el Namespace "UnityEngine.UI" obtenim el mètode *onClick.AddListener(method)*, el qual assigna un mètode a executar si es polsa el botó al que fa referència. Pel que fa al drop-down, es fa servir *onValueChanged.AddListener(method(indexDropdown))*, que crida un mètode amb paràmetre l'index al que ha canviat el drop-down. Aquests mètodes s'assignen al mètode *Start()* de MonoBehavior, com es pot veure a la Figura 10.15.

```
// Start is called before the first frame update
Mensaje de Unity | 0 referencias
private void Start()
{
    startAnimation = false;

    // A method is added to be called if the button is pressed
    startButton.onClick.AddListener(StartGame);
    exitButton.onClick.AddListener(ExitGame);

    // A method is added to be called if the drop-down value is changed
    animationDropdown.onValueChanged.AddListener(delegate {
        PlayAnimation(animationDropdown);
    });

}
```

Figura 10.15: Mètode *Start()* a la classe "HandlerDemo"

Quan es pressiona al botó "StartButton", simplement s'amaga el "StartPanel" i es mostra el "GamePanel" per a que es pugi començar a seleccionar animacions (Figure 10.16a). Si es selecciona una animació de la llista del "Dropdown", es desactiva el "GamePanel" i s'activa la xarxa neuronal del defensor juntament amb l'animació de l'atacant seleccionada, vista pel codi com l'índex de l'animació a la llista (Figure 10.16b). Finalment, si es pressiona el botó "ExitButton", es tanca l'aplicació (Figure 10.16c).

```
// Activates the animations drop-down panel
1 referencia
private void StartGame()
{
    panelGame.SetActive(true);
    panelStart.SetActive(false);
}
```

(a) Mètode "StartGame"

```
// Play the animation selected from the drop-down menu.
1 referencia
private void PlayAnimation(Dropdown change)
{
    panelGame.SetActive(false);

    startAnimation = true;

    attacker.GetComponent<MoveAttacker>().enabled = true;
    attacker.GetComponent<MoveAttacker>().numAnim = change.value;

    defender.GetComponent<SwordDefender>().enabled = true;
}
```

(b) Mètode "PlayAnimation"

```
// Quits the aplication
1 referencia
private void ExitGame()
{
    Application.Quit();
}
```

(c) Mètode "ExitGame"

Figura 10.16: Mètodes de *AddListener()* a la classe "HandlerDemo"

Per últim, s'ha creat un mètode, "Restart()" (Figure 10.17), per desactivar l'atacant i el defensor un cop s'hagi acabat l'animació, ja que aquest mètode públic es cridat quan l'espasa enemiga col·lideix amb l'espasa defensora, o si hi algún error, si col·lisiona amb el defensor.

```
// When the animation is over, characters are disabled by disabling their scripts
6 referencias
public void Restart()
{
    panelGame.SetActive(true);
    attacker.GetComponent<MoveAttacker>().enabled = false;
    defender.GetComponent<SwordDefender>().enabled = false;
}
```

Figura 10.17: Mètode "RestartDemo" a la classe "HandlerDemo"

10.2.2 Canvis codi

Tant a la classe "MoveAttacker" com a la de "SwordDefender" s'ha afegit una nova variable, public GameObject **demoHandler**, la qual fa referència al GameObject del canvas on està l'script "HandlerDemo". En els mètodes on es calcula la recompensa per l'agent canvia l'estructura.

Nova estructura: En primer lloc, segueix estant l'assignació del valor de la recompensa, però, seguidament, és comprova si hi ha una referència a aquest GameObject demoHandler. Si hi és, vol dir que estem a la demo, per tant, assignarem la variable pública "startAnimation" de la classe "HandlerDemo" a false. Posteriorment, acabem l'episodi d'inferència i tornem a mirar si hi té una referència diferent a null. Si en té, cridem el mètode "Restart()" de la classe "HanlderDemo" per desactivar als personatges.

La Figura 10.18 mostra el codi de la nova estructura.

```
defender.GetComponent<SwordDefender>().SetReward(totalReward);

if (demoHandler != null)
{
    GetComponent<AudioSource>().Play();
    demoHandler.GetComponent<HandlerDemo>().startAnimation = false;
}

defender.GetComponent<SwordDefender>().EndEpisode();

if (demoHandler != null)
{
    demoHandler.GetComponent<HandlerDemo>().Restart();
}

StartCoroutine(WaitEndEpisode());
```

(a) Canvi exclusiu al mètode *OnCollisionEnter(collision)* de la classe "MoveAttacker"

```
defender.GetComponent<SwordDefender>().SetReward(-1f);

if (demoHandler != null)
{
    demoHandler.GetComponent<HandlerDemo>().startAnimation = false;
}

defender.GetComponent<SwordDefender>().EndEpisode();

if (demoHandler != null)
{
    demoHandler.GetComponent<HandlerDemo>().Restart();
}

StartCoroutine(WaitEndEpisode());
```

(b) Canvi en el codi als mètodes de les classes on s'estableix una recompensa de -1

Figura 10.18: Nova estructura del codi a l'hora d'acabar un episodi

Finalment, hi ha tres altres canvis. El primer és a la classe "MoveAttacker", on s'ha afegit un nou mètode, "PlayIDLE" (Figura 10.19) que reprodueix l'animació IDLE de l'atacant.

```
// Reproduce the "IDLE" animation
1 referencia
public void PlayIDLE()
{
    attackerAnimator.Play("IDLE");
    attackerModel.localPosition = new Vector3(9f, 0, 0);
    attackerModel.rotation = new Quaternion(0, -0.707106829f, 0, 0.707106829f);
}
```

Figura 10.19: Mètode *PlayIDLE()* a la classe "MoveAttacker"

La resta de canvis són a la classe "SwordDefender", on primerament s'ha afegit el mètode de MonoBehavior anomenat *Awake()*, el qual s'activa cada cop que s'habilita la instància del GameObject a la que està adjunta a la classe.

En ell, es fa l'assignació del rigidBody (Figura 10.20), ja que en començar l'script desactivat, és necessari tenir-ho el primer de tot. A vegades, el mètode OnEpisodeBegin() de l'agent era cridat abans del mètode Start(), i per aquesta raó tot el que es feia en aquest mètode es fa ara en l'Awake().

```
// Called when the script instance is being loaded
Mensaje de Unity | 0 referencias
private void Awake()
{
    rBody = GetComponent<Rigidbody>();
}
```

Figura 10.20: Demo - Mètode *Awake* a la classe "SwordDefender"

Com a útlim canvi, al mètode OnEpisodeBegin() (Figura 10.21) es revisa si s'està entrenant o s'està a la demo. Si és la demo, mira si la variable "StartAnimation" de la classe "HandlerDemo" és true. Si no és així, reprodueix l'animació IDLE del personatge atacant, la qual es reproduirà en bucle. En canvi, si la variable té el valor de true, fes reproduir a l'atacant la següent animació que li pertoqui (recordar que des de "HandlerDemo" s'ha modificat la variable "numAnim" de "MoveAttacker" amb l'índex on és a la llista del "Dropdown" l'animació seleccionada).

```
// Set up an Agent instance at the beginning of an episode
0 referencias
public override void OnEpisodeBegin()
{
    // Agent's inital positon and velocity
    gameObject.transform.localPosition = new Vector3(1.77999997f, 6.1930995f, 0.180999994f);
    rBody.velocity = Vector3.zero;
    gameObject.transform.localRotation = new Quaternion(0.707106829f, 0, -0.707106829f, 0);
    rBody.angularVelocity = Vector3.zero;

    // Attacker's initial position and velocity
    if (demoHandler == null)
    {
        attacker.GetComponent<MoveAttacker>().PlayNextAnim();
        attacker.localPosition = new Vector3(-0.735548317f, -1.41793692f, 0.0554279089f);
        attacker.localRotation = new Quaternion(0.000568372896f, 4.47034871e-08f, -5.15475813e-06f, 0.999999881f);
    }
    // If it's the "Demo Scene", play only one animation.
    else
    {
        if (demoHandler.GetComponent<HandlerDemo>().startAnimation)
        {
            attacker.GetComponent<MoveAttacker>().PlayNextAnim();
            attacker.localPosition = new Vector3(-0.735548317f, -1.41793692f, 0.0554279089f);
            attacker.localRotation = new Quaternion(0.000568372896f, 4.47034871e-08f, -5.15475813e-06f, 0.999999881f);
        }
        // If it has already played, play the IDLE animation
        else
        {
            attacker.GetComponent<MoveAttacker>().PlayIDLE();
        }
    }
}
```

Figura 10.21: Demo - Mètode *OnEpisodeBegin()* a la classe "SwordDefender"

11 Conclusions

Per assolir els objectius del projecte, es van marcar unes tasques a realitzar (vegeu Apartat 1.3: Objectius). Aquestes tasques s'han pogut dur a terme malgrat les adversitats que s'han trobat al llarg del desenvolupament.

S'ha estudiat a consciència el funcionament del motor Unity i l'API ML-Agents, aprenent com és el funcionament de les classes i com s'uneixen totes entre elles per formar agents en l'entorn d'un videojoc.

S'ha après sobre tècniques d'intel·ligència artificial, més concretament deep learning i les xarxes neuronals. La informació sobre aquestes tècniques és molt abundant i complexa, i en alguns casos confusa, havent de, en certes ocasions, aprofundir en les bases matemàtiques per a entendre-ho. Aquestes tècniques tenen un ventall molt gran de possibilitats, i en aquest projecte només se n'ha vist una petita part del potencial i funcionament.

S'ha après també el funcionament de la cinemàtica inversa i el seu potencial. Tot i que s'ha aprofundit només en la seva implementació a Unity, s'ha trobat un gran aliat per a treballar amb les animacions.

L'elecció de les eines ha estat bastant senzilla, ja que l'única que hi havia disponible per dur a terme aquest tipus de projectes és l'ML-Agents. Que ja existís una eina com aquesta en la que es poden crear agents controlats per intel·ligencies artificials basades en xarxes neuronals va ser un alleujament de treball en comparació amb les idees prèvies a començar el projecte. Gràcies a la insistència i perseverança del tutor a buscar eines per a Unity, es va aconseguir trobar-la.

S'ha assolit la creació de diferents escenaris amb agents de diversa complexitat fins arribar a l'objectiu final: un agent que controla l'espasa d'un guerrer i es capaç de moure-la mitjançant la cinemàtica inversa amb sentit i coherència fins a parar el cop d'una espasa atacant. Altre cop, gràcies a l'experiència del tutor en aquest món, el plantejament del codi i del funcionament de l'agent ha estat molt més ràpid de realitzar i pensar. La creació dels límits i les restriccions ha estat un treball ardu però finalment s'han trobat uns d'òptims.

Respecta a l'objectiu menys visible, s'ha pogut observar que l'agent aprèn a moure's per a millorar la seva política de recompenses i la majoria d'animacions apreses arriba a aconseguir una mitja de 0.7 sobre 1 de recompensa, fent que les espases col·lideixin just als seus centres. S'ha aconseguit demostrar que gràcies als algoritmes d'aprenentatge per reforç i a les xarxes neuronals podem aconseguir uns resultats que s'apropen al comportament humà.

L'entrenament de les xarxes neuronals és un aspecte molt crític, ja que la potència de l'ordinador i el temps disponible són reduïts (en comparació amb els professionals de la indústria). Molts projectes d'aquesta índole es fan difícils de completar a causa del cost computacional (elevat), fent complicat per als estudiants poder investigar més. A més a més, s'ha d'entendre bé com treballar amb elles perquè es pot perdre molt de temps en modificar els paràmetres sense trobar res de profit.

Com a última reflexió, comentar que els resultats han estat els esperats, una animació ben senzilla i per res pretensiosa, que s'han creat tot i la incertesa del projecte. S'ha aconseguit crear un entorn que gràcies als algoritmes de reforç i deep learning és capaç d'aprendre com moure una espasa i protegir-se d'un atac enemic.

12 Treball futur

El projecte té diferents vies de millora i expansió:

- Perfeccionar el sistema de recompenses per a simular conscientment diferents guàrdies reals.
- Guardar aquests moviments de defensa en clips d'animació i modificar el cos per acompanyar-lo.
- Crear una aplicació on es pugui crear animacions aleatòries d'atac i pugui entrenar un agent per a que es defensi.
- Crear un script per a millor el moviment de la cinemàtica inversa, fent que els colzes es moguin en referència a la posició de l'espasa i que les mans segueixi el maneg de l'espasa, tot simulant que l'està agafant.

13 Bibliografia

- Kucuk, S., & Bingul, Z. (2006). *Robot kinematics: Forward and inverse kinematics.* https://www.researchgate.net/profile/Mohamed_Mourad_Lafifi/post/Is-there-any-method-to-solve-the-multiple-solutions-problem-for-inverse-kinematics-of-a-6-DOF-serial-manipulator/attachment/59d639a579197b8077996da2/AS%3A402906923716613%401473071904615/download/Robot+Kinematics+-++Forward+and+Inverse+Kinematics.pdf
- Unity Technologies. (2021). *Unity documentation: Inverse Kinematics.* https://docs.unity3d.com/Manual/InverseKinematics.html
- Tutorials Point. (2021). *SDLC - Waterfall Model.* https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm
- Ward, J. (29 d'abril de 2008). What is a Game Engine? *Informa PLC.* https://www.gamecareerguide.com/features/529/what_is_a_game_.php
- Michael, T. (2006) *The Discipline of Machine Learning.* http://www.cs.cmu.edu/~tom/pubs/MachineLearning.pdf
- Open AI. (2018). *INTRODUCTION TO RL. Part 1: Key Concepts in RL.* [Consulta: 18 de maig de 2021]. https://spinningup.openai.com/en/latest/spinningup/rl_intro.html
- Open AI. (2018). *INTRODUCTION TO RL. Part 2: Kinds of RL Algorithms.* [Consulta: 18 de maig de 2021]. https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html
- Szepesvári, C. (2010). *Algorithms for reinforcement learning. Synthesis lectures on artificial intelligence and machine learning.* http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.308.549&rep=rep1&type=pdf
- Mehlig, B. (2021) *Machine learning with neural networks: An introduction for scientists and engineers.* https://arxiv.org/pdf/1901.05639.pdf
- Chris, N. (2020). A Beginner's Guide to Neural Networks and Deep Learning. *Pathmind Inc.* https://wiki.pathmind.com/neural-network
- Unity Technologies. (2021). *Unity User Manual 2020.3 (LTS)* https://docs.unity3d.com/Manual/index.html
- Unity Technologies. (2021). *Unity ML-Agents Toolkit* https://github.com/Unity-Technologies/ml-agents/tree/main/docs
- Unity Technologies. (2021). *Unity Manual. About ML-Agents package* https://docs.unity3d.com/Packages/com.unity.ml-agents@1.0/manual/index.html
- Facebook's AI Research lab. (2021). *Pytorch Documentation* https://pytorch.org/docs/stable/index.html
- Python Software Foundation. (2021). *Python Documentation.* https://www.python.org/doc/

Referències

[1] Toon Boom Animation Inc. (2021). *Straight Ahead and Pose-to-Pose Principle.* [Consulta: 11 de maig de 2021].
https://learn.toonboom.com/modules/animation-principles/topic/straight-ahead-and-pose-to-pose-principle

[2] Autodesk Softimage. (2015). *Animating with Inverse Kinematics.* [Consulta: 11 de maig de 2021].
https://download.autodesk.com/global/docs/softimage2013/en_us/userguide/index.html?url=files/ik_AnimatingwithInverseKinematics.htm,topicNumber=d30e191647

[3] OpenAI. (2021). *INTRODUCTION TO RL. Part 2: Kinds of RL Algorithms.* [Consulta: 18 de maig de 2021].
https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html

[4] Bapu Ahire, J. (27 de novembre de 2018). The Artificial Neural Networks Handbook: Part 4. *DZone: Programming & DevOps news, tutorials & tools*
https://dzone.com/articles/the-artificial-neural-networks-handbook-part-4

[5] Unity Technologies. (2021). *Unity documentation: System requirements for Unity 2020 LTS.* [Consulta: 15 de maig de 2021].
https://docs.unity3d.com/Manual/system-requirements.html?_ga=2.11458994.747430549.1498124004-1088614642.1492950587

[6] Unity Technologies. (2021). *ML-Agents Toolkit Overview.* [Consulta: 20 de maig de 2021].
https://github.com/Unity-Technologies/ml-agents/blob/main/docs/ML-Agents-Overview.md

[7] Sketchfab. (2021). *Samurai: 3D Model by MR.Sirapop* [Consulta: 27 d'abril de 2021].
https://sketchfab.com/3d-models/samurai-0ac619b7b276445cb69c1887dd21ede9

[8] Lovric, A. (1 de maig de 2020). The Beginner's Guide to German Longsword, part 5: The four (plus one) basic guards. *Arming Guild.*
https://armingguild.com/the-beginners-guide-to-german-longsword-part-5-the-four-plus-one-basic-guards/

[9] The International Fellowship of Chivalry-Now. (2006). *Discipline of the Sword: Guards.*
http://www.chivalrynow.net/sword/guards.htm#top

[10] Avila, M. (25 de febrer de 2015). Diferencia entre Seigan no kame y Chudan no Kamae. *Curso de Katana.*
https://cursodekatana.com/diferencia-entre-seigan-no-kamae-y-chudan-no-kamae/

[11] Vogtherr, H. (1553). Augsburg State Archive Reichsstadt "Schätze" Nr. 82.
https://commons.wikimedia.org/wiki/File:Reichsstadt_%22Sch%C3%A4tze%22_Nr._82_045v.jpg

[12] Tenshin Shoden Katori Shinto Ryu. (2021). *KISO.*
http://www.katori-shintoryu.fr/kiso.html

14 Manual d'usuari i d'instal·lació

14.1 Instal·lació ML-Agents per realitzar els entrenaments

Passos a realitzar:

1. Descarregar i instal·lar Unity 2018.4 o posterior. Es recomana encaridament que s'instal·li Unity a través del Unity Hub, ja que permet gestionar diverses versions de Unity.
2. Instal·lar Python 3.6.1 o superior. Si s'utilitza Windows, instal·lar la versió x86-64 i no la x86.
3. Instal·lar el paquet *com.unity.ml-agents* directament des del Package Manager registry. S'ha d'habilitar "Preview Packages" en el desplegable "Advanced" per a trobar-ho.
4. Instal·lar el paquet de Python *mlagents*. Aquest implica la instal·lació d'altres paquets de Python dels quals depèn. Es pot tenir problemes d'instal·lació si la màquina té versions anteriors de qualsevol d'aquestes dependències ja instal·lades. A més a més, per a instal·lar mlagents s'ha de realitzar mitjançant l'entorn de Python anomenat *pip3*. Per tant, la comanda a utilitzar en un terminal de Windows és la següent:
   pip3 install mlagents
   Si s'ha instal·lat correctament, s'hauria de poder executar mlagents-learn –help, després de la qual cosa veurà els paràmetres de la línia de comandos que pot utilitzar amb mlagents-learn.
5. Instal·lar PyTorch amb la comanda següent en un terminal de Windows:
   pip3 install torch==1.7.0 -f https://download.pytorch.org/whl/torch_stable.html

   Tenir en compte que, a Windows, és necessita Visual C++ Redistributable de Microsoft si no ho té ja.

Per a qualsevol problema, consultar la pàgina d'instl·lació de ML-Agents:

https://github.com/Unity-Technologies/ml-agents/blob/4feccd22f466b1affd5bf9e8050f1ff2b54a62da/docs/Installation.md