UNIVERSIDAD DE ALCÁLA
Escuela Politécnica Superior

Grado en Ingeniería de computadores.

Videojuegos, Inteligencia Artificial y Comportamiento: estudio, integración y demostración.

Autor: Alejandro Rilo Ferreiro
Tutor: Antonio Moratilla Ocaña
Curso académico: 2021-2022

ÍNDICE

UNIVERSIDAD DE ALCÁLA
Escuela Politécnica Superior

Resumen:

En este trabajo, el objetivo principal es el desarrollo de una serie de NPCs que interactúan de diferentes maneras con un personaje controlado por un jugador.
Para esto, hemos utilizado Unity 3D para crear un minijuego RPG en el cual se implementan los diferentes NPCs y el personaje, así como la interacción.
Pero antes de empezar el desarrollo del minijuego, se ha realizado un estudio de las diferentes formas de inteligencias artificiales que existen y cómo son aplicables al mundo de los videojuegos para posteriormente elegir las más adecuadas, así como las que ya ofrece Unity para facilitar el trabajo a los desarrolladores.

Resume:

This project the main goal is the development of some NPCs who interact in different ways with a character controlled by a human player.

To do this, we have used Unity 3D to create a RPG minigame where the NPCs will be implemented, the character controlled by the human player and the interaction between them.

But before start with the development of the minigame, we have investigated about the differents IAs that exists and how they are used on Videogames industry, after that investigation, we have decided to use some of this IAs for some reasons for example there are IAs that Unity provides by himself to make easier the task for the developpers.

Resumen extendido:

En el trabajo como hemos dicho anteriormente, hemos hecho en primer lugar un estudio de las diferentes Inteligencias artificiales existentes como pueden ser diferentes algoritmos de **path-finding** para realizar el reconocimiento de un terreno y saber el camino más corto para ir de un punto a otro evitando obstáculos etc. Como por ejemplo puede ser Djikstra o el más utilizado en el mundo de los videojuegos actualmente el algoritmo A*.
Aparte de algoritmos de **pathfinding** como los mencionados anteriormente, también se estudian en este trabajo los diferentes tipos de inteligencia artificial que existen para dotar de inteligencia a los NPCs a la hora de reaccionar a los diferentes estímulos que le puede ofrecer el entorno como puede ser, por ejemplo, cuando un personaje está quieto y se acerca el personaje a una distancia marcada por el desarrollador, el NPC reaccione a este estímulo persiguiéndole y peleando con él. En este ámbito, podemos encontrar inteligencias artificiales que aprendan de cómo se comporta el jugador y actuar en consecuencia en base al aprendizaje mediante por ejemplo RNA (Red neuronal artificial) y también podemos encontrar IAs que su comportamiento sea a través de árboles de decisión en los cuáles dependiendo de cómo el personaje interactúe con el NPC, así como el uso de máquinas de estado las cuáles funcionan reaccionando a los estímulos del medio ambiente como puede ser el caso en el que un personaje esté patrullando una zona y mientras el personaje no entre en su zona de visión está en alerta patrullando hasta el momento en el que entra en su zona de visión y por tanto detecta a este y cambia su estado pasando a un estado de persecución al personaje lo cual es muy interesante y una de las IAs que más hemos utilizado en este trabajo.
Una vez explicadas las diferentes IAs existentes, en este trabajo se explica cuáles son las más utilizadas en el mundo de los videojuegos y cómo y por qué son las más utilizadas, así como cuáles son las que nos ofrece Unity y las que elegiremos en nuestro proyecto explicando el por qué.
El siguiente apartado del proyecto será la explicación de cada uno de los NPCs desarrollados, explicando cuál es su funcionalidad dentro del juego los diferentes métodos de aplicación de IA utilizados en cada NPC y cómo los hemos implementado en estos, así como las diferentes dificultades encontradas en el desarrollo y cómo hemos hecho para solucionar dichos problemas.
Una vez explicado cada NPC, explicaremos la metodología de trabajo que hemos seguido, explicando cada etapa del proyecto con las fechas y descripción de cada estas. Finalizada la explicación de la metodología y etapas del trabajo, pasaremos a realizar una conclusión del trabajo en la cual explicaremos lo aprendido en el trabajo, así como las dificultades más importantes que hemos encontrado y las posibles mejoras en un futuro que podríamos añadir al proyecto para mejorarlo e intentar hacer de este minijuego un minijuego más profesional y útil de lo que es actualmente.
Para terminar, al final del trabajo se añadirá una bibliografía de los diferentes sitios en los que me he apoyado para el estudio de las IAs y la realización del videojuego con los NPCs y el personaje.

Palabras clave:

NPC, Inteligencia artificial, máquina de estados, árboles de decisión, Script.

Introducción.

En la industria de los videojuegos, siempre ha existido la necesidad de poder contar con ciertos personajes no controlados por ningún jugador que interactuaran con aquellos que, si son controlados por el jugador, que son los llamados NPCs.
La dificultad más grande que se puede encontrar a estos NPCs, es la necesidad de dotar de cierta inteligencia a estos personajes haciendo que estos se adapten de la mejor manera posible a la realidad, de manera que sean capaces de interactuar adaptándose a diferentes situaciones y reaccionando a lo que ocurre a su alrededor.
Y la pregunta es: ¿Cómo podemos hacer para dotar a un personaje de inteligencia si no lo controla nadie?, y ahí es donde entra la Inteligencia artificial o IA.
Y es aquí en el estudio e investigación de los diferentes tipos de IA que existen y cómo aplicarlas en estos personajes llamados NPCs donde vamos a centrar nuestro trabajo.

Objetivos y campo de aplicación.

Los objetivos de este trabajo son los siguientes:

1. Explicar de forma más o menos detallada los diferentes tipos de Inteligencia artificial que existen.
2. Cómo se pueden aplicar estos tipos de IA en la industria de los videojuegos y más en concreto en los NPCs.
3. Conseguir implementar un NPC para demostrar la aplicación de estas metodologías de Inteligencia artificial utilizando el motor de videojuegos Unity3D.

Descripción del proyecto.

Estudio de los diferentes tipos de IAs existentes Algoritmos de PATHFINDING:

En este apartado, vamos a estudiar los diferentes tipos de algoritmo de pathfinding:

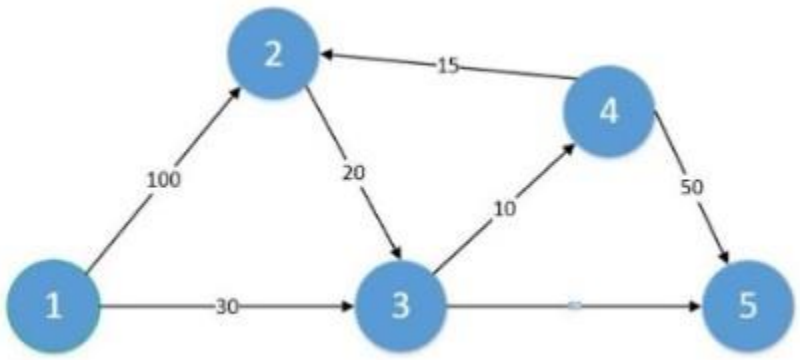
- **Algoritmo de Djikstra o SPF:**

Este algoritmo es un algoritmo de pathfinding que se utiliza para encontrar en un grafo el camino más corto de un vértice al resto de vértices del grafo mediante un sistema de pesos en cada arista del grafo.

Este algoritmo funciona utilizando dos conjuntos de nodos. Un conjunto S que contiene los nodos ya seleccionados y un conjunto C que contiene el resto de nodos que existen en el grafo.

Para explicar este algoritmo, vamos a realizar un ejemplo de manera que se entienda más fácil.

Ejemplo:



En este grafo, lo primero que haremos será elegir nuestro nodo origen que será el nodo 1 e iremos calculando el camino más corto haciendo el resto de los nodos del grafo.

En la primera iteración, lo que haremos será poner una etiqueta al nodo 1 de manera que:

Etiqueta = [Valor acumulado, procedencia] iteración Etiqueta nodo 1 = [0,-] 0

En la segunda iteración, lo que haremos será ir desde el nodo 1 hasta todos los nodos que tiene conectados y tabular esta iteración.

Etiqueta nodo 2 = [100,0]1
Etiqueta nodo 3 = [30, 0]2

Una vez ya hemos llegado a este punto habremos ido a todos los puntos diferentes que podemos ir desde el nodo 1, por tanto, lo que haremos ahora será tabular esta iteración quedando de la siguiente manera.

Nodo	Etiqueta	Estado
1	[0,-]	Final
2	[100,0]	Temporal
3	[30,0]	Final

En los casos de los nodos 1 y 2 el estado en el que quedan será el estado final, ya que son la distancia mínima segura al 100% para llegar a esos dos nodos desde el nodo 1, ya que el nodo 1 es el origen y para cualquier camino alternativo que tengamos para llegar al nodo 3 hay que pasar seguro por el nodo 2 cuyo valor acumulado (100) en la etiqueta es mayor que el valor acumulado que tiene la etiqueta 3 (30) y por tanto 30 es el valor menor posible.

En la tercera iteración, por tanto, lo que haremos será estudiar todos los caminos posibles desde el nodo 3 hasta todos los diferentes puntos que estén conectados a él que son los nodos 4 y 5.

Etiqueta nodo 4 = [40,3]2

Etiqueta nodo 5 = [90,3]2

Una vez hecho esto, pasaremos a tabular la iteración:

Nodo	Etiqueta	Estado
1	[0,-]	Permanente
2	[0,100]	Temporal
3	[0,30]	Permanente
4	[40,3]	Permanente
5	[90,3]	Temporal

El nodo 4 pasa a estado permanente debido a que cualquier otro camino para llegar desde el nodo 1 hasta el nodo 4, conllevaría un mayor valor acumulado.
En la cuarta iteración, lo que haremos será calcular los posibles caminos desde el nodo 4 que son los nodos 2 y 5.

Etiqueta nodo 2 = [55,4]3

Etiqueta nodo 5 = [90,4]3

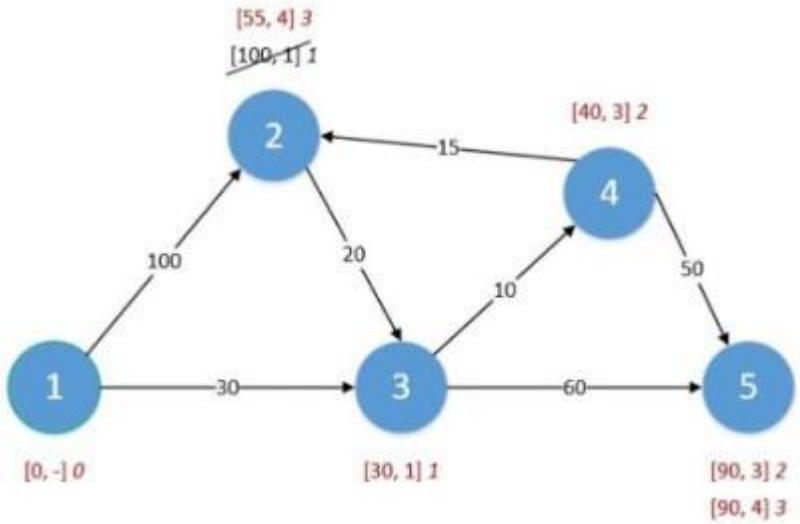
Una vez hecho esto pasamos a tabular la iteración:

Nodo	Etiqueta	Estado
1	[0,-]	Permanente
2	[55,4]	Permanente
3	[0,30]	Permanente
4	[40,3]	Permanente
5	[90,3] o [90,4]	Temporal

En esta tabla, podemos ver que hemos cambiado la etiqueta del nodo 2 ya que la anterior etiqueta era [100,0] por lo que el valor acumulado era 100 y en la nueva etiqueta que es [55,4] el valor acumulado es 55 y al ser menor nos quedamos con la etiqueta [55,4] y como permanente porque es el menor valor acumulado posible en todos los caminos y nos queda por decidir la etiqueta del nodo 5, ya que podemos ver que tanto en la anterior iteración como en esta el valor acumulado es 90.
En la quinta iteración quedaría estudiar los posibles casos desde el nodo 2, que es el nodo 3 el cual ya tiene un estado permanente y por tanto no tiene sentido reevaluar, y el nodo 5 que no tiene destinos disponibles, por tanto, el algoritmo quedaría finalizado y el nodo 5 pasaría a estado permanente con dos posibles caminos óptimos, quedando la tabla así:

Nodo	Etiqueta	Estado
1	[0,-]	Permanente
2	[55,4]	Permanente
3	[0,30]	Permanente
4	[40,3]	Permanente
5	[90,3] o [90,4]	Permanente

Y el grafo quedaría de la siguiente manera:



De esta manera, podríamos saber el camino más corto desde el nodo 1 hasta cualquier nodo del grafo.

Ventajas y desventajas del algoritmo:

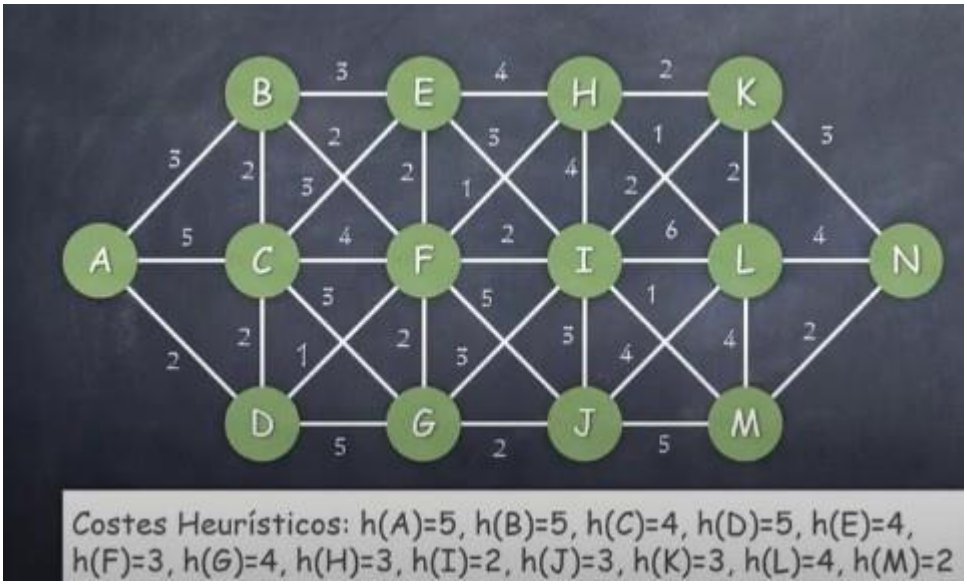
La desventaja más grande de este algoritmo es el problema de escalabilidad, ya que la complejidad de este algoritmo crece de manera bastante grande con el aumento de la cantidad de vértices, aunque tiene otras desventajas como puede ser el no poder trabajar con pesos negativos, así como que es más lento que otros algoritmos como el algoritmo de Dijkstra.
Como ventajas, podemos mencionar que es una búsqueda no informada, es decir, que no necesita conocer el destino para resolver el algoritmo.

- **Algoritmo A*:**

El algoritmo A* es otro algoritmo de pathfinding para descubrir la ruta óptima de un nodo a otro del grafo partiendo de una configuración de inicio y teniendo una configuración destino, es un algoritmo de búsqueda inteligente y eficiente el cuál trabaja con dos listas de nodos, una abierta y otra cerrada y lo que se hace en cada iteración es seleccionar el nodo de la lista abierta con menor coste de evaluación que este coste es igual a la suma del coste operativo(coste desde el inicio) y un coste heurístico que es el coste sobreestimado a la configuración de destino y que se suele calcular normalmente siguiendo un criterio de distancia. Una vez calculado este coste se asigna al nodo y se añade a la lista cerrada. A la vez que se añaden nuevos nodos a la lista abierta y se actualizan los costes.

Coste evaluación f(n) = Coste operativo g(n) + Coste heurístico h(n)

Para explicar mejor este algoritmo, voy a mostrar un ejemplo:



En este grafo, lo que vamos a hacer va a ser calcular el coste de llegar desde el nodo A hasta el nodo N siguiendo el algoritmo A*. Los costes heurísticos en el ejemplo ya estarían calculados siguiendo como criterio para calcularlos por ejemplo el h(A) es 5 ya que de izquierda a derecha hay 5 nodos hasta llegar a N y también se le añade 1 de coste extra en caso de pertenecer el nodo a la fila de arriba o a la de abajo.

Iteración 0:
Lo primero que haremos será crear las dos listas, la lista cerrada y la lista abierta:
En la lista cerrada añadiremos el nodo orígen (nodo A). Y en la lista abierta añadiremos los nodos que sean adyacentes y alcanzables desde el nodo A que son el B, el C y el D con su f(n) o Coste de evaluación.
Lista cerrada: g(A) = 0
Lista abierta: f(B) = g(B) + h(B) = 3 + 5 = 8, f(C) = 5 + 4 = 9, **f(D) = 2 + 5 = 7.**

Iteración 1:
En esta iteración, lo que haremos será repetir el proceso anterior a partir del nodo con menor Coste de evaluación de la lista abierta que es el nodo D.
Por tanto, añadiremos a la lista cerrada el nodo D y añadiremos los nuevos nodos adyacentes a D y alcanzables por D a la lista abierta:
Lista cerrada: g(A) = 0, g(D) = 2
Lista abierta: f(B) = 3 + 5 = 8, f(C) = 4 + 4 = 8, **f(F) = 3 + 3 = 6**, f(G) = 7 + 4 = 11.

Iteración 2:
En la iteración 2, lo que haremos será repetir el proceso de la anterior, cogeremos el nodo F ya que es el que tiene menor coste de evaluación y actualizamos las listas:
Lista cerrada: g(A) = 0, g(D) = 2, g(F) = 3
Lista abierta: f(B) = 3 + 5 = 8, f(C) = 4 + 4 = 8, f(G) = 5 + 4 = 9, f(E) = 5 + 4 = 9, f(H) = 4 + 3
= 7, **f(I) = 5 + 2 = 7,** f(J) = 8 + 3 = 11.

Iteración 3:
En esta iteración en el empate entre el coste f(I) es igual al f(H), elegimos por ejemplo el f(I) y volvemos a realizar el mismo proceso:
Lista cerrada: g(A) = 0, g(D) = 2, g(F) = 3, g(I) = 4
Lista abierta: f(B) = 3 + 5 = 8, f(C) = 4 + 4 = 8, f(G) = 5 + 4 = 9, f(E) = 5 + 4 = 9, f(J) = 7 + 3
= 10, f(J) = 7 + 3 = 10, f(K) = 6 + 2 = 8, f(L) = 10 + 1 = 11, f(M) = 5 + 2 = 7, f(H) = 4 + 3 = 7.

Iteración 4:
En esta iteración tenemos también un empate y en este caso elegimos f(H). Seguimos repitiendo el mismo procedimiento:
Lista cerrada: g(A) = 0, g(D) = 2, g(F) = 3, g(I) = 4, g(H) = 4.
Lista abierta: f(B) = 3 + 5 = 8, f(C) = 4 + 4 = 8, f(G) = 5 + 4 = 9, f(E) = 5 + 4 = 9, f(J) = 7 + 3
= 10, f(K) = 6 + 2 = 8, **f(L) = 5 + 1 = 6**, f(M) = 5 + 2 = 7.

Iteración 5:
Lista cerrada: g(A) = 0, g(D) = 2, g(F) = 3, g(I) = 4, g(H) = 5, g(L) = 5.
Lista abierta: f(B) = 3 + 5 = 8, f(C) = 4 + 4 = 8, f(G) = 5 + 4 = 9, f(E) = 5 + 4 = 9, f(J) = 7 + 3
= 10, f(K) = 6 + 2 = 8, **f(M) = 5 + 2 = 7,** f(N) = 9.

Iteración 6:
Lista cerrada: g(A) = 0, g(D) = 2, g(F) = 3, g(I) = 4, g(H) = 5, g(L) = 5, g(M) = 5.
Lista abierta: f(B) = 3 + 5 = 8, f(C) = 4 + 4 = 8, f(G) = 5 + 4 = 9, f(E) = 5 + 4 = 9, f(J) = 7 + 3
= 10, f(K) = 6 + 2 = 8, **f(N) = 7.**

Iteración 7:
Al haber llegado al punto en el que f(N) es el nodo con menor coste de evaluación significa que ya hemos encontrado el camino más rápido que sería:
El camino sería: **A -> D -> F -> I -> M -> N** y el coste es: 7 .

Ventajas y desventajas del algoritmo:

La principal desventaja de este algoritmo es que necesitas conocer el punto origen y el punto de destino debido a que es una búsqueda informada y necesita de una función heurística que cumpla ciertas condiciones.
Como ventajas, la rapidez y eficiencia de este algoritmo respecto por ejemplo a Dijkstra sería la principal.
Este algoritmo es el que más se utiliza en la industria de los videojuegos y a su vez el utilizado por Unity para saber la manera más rápida para llegar de un punto a otro.

Arboles de decisión.
Otra de las técnicas de inteligencia artificial, es el uso de árboles de decisión, en estos lo que se hace es que el NPC o aquello a lo queramos dotar de IA de una capacidad de actuar de una forma o de otra dependiendo del input que reciba.
Los árboles de decisión están compuestos por los siguientes elementos:

- **Nodos:** Es cada una de las posibles situaciones que se pueden dar.
- **Flechas:** Son los arcos que unen los nodos entre sí.
- **Vectores:** Estos representan la opción por la que se opta entre las diferentes posibildades.
- **Etiquetas:** Unen nodos y flechas y denominan las acciones que se llevan a cabo.

Un ejemplo gráfico de árbol de decisión puede ser el siguiente:

1. Estructura básica de un árbol de decisión.
En el ámbito de los videojuegos, los árboles de decisión se utilizan mucho por ejemplo en las conversaciones de los personajes con un NPC en el cuál este se comporta de una

manera o de otra dependiendo de las respuestas que dé el jugador.
Hay un ejemplo muy claro en el cuál estos árboles de decisión modifican la propia historia del juego de manera importante y este es el Fallout.
En el juego Fallout 3, una de las primeras escenas del juego es el nacimiento de nuestro personaje en el cual se ve el parto y en el momento del parto se pregunta al jugador para que elija su sexo y dependiendo de la elección que hagas la conversación de justo después es diferente a si hubiera elegido chica. Esto es un claro ejemplo de un árbol de decisión:

2. Elección de sexo del personaje al inicio del fallout.
En este caso, el jugador elegirá chico y se puede ver como el mensaje es: "¿Es niño? ¡Es un niño! ¡Tenemos un hijo Catherine! ¡Un niño precioso y sano!" Todo en masculino debido a haber elegido chico.

3. Mensaje después de elegir chico en la imagen anterior.

Otro caso en el que se utilizan los árboles de decisión es en juegos como el ajedrez en el cuál la decisión de la IA será siempre dependiente del movimiento del oponente. Ya que si el jugador por ejemplo el jugador es blancas y empieza moviendo el peón de rey, la partida es completamente diferente en cuanto al árbol que si el jugador mueve cualquier otra pieza del tablero, en estos casos como el número de partidas posible en ajedrez es cincuenta órdenes de magnitud más grande que el número de átomos que hay en el universo lo cual es una barbaridad y hace que sea imposible conseguir estudiar todas las posibles partidas computacionalmente en un tiempo razonable, lo que se hizo por ejemplo en la IA Deep blue que fue una IA desarrollada por IBM la cuál fue capaz de derrotar a Garry Kasparov fue hacer aprender a una IA haciéndola jugar muchas partidas y después de esto aplicar a un árbol de decisión generado un algoritmo de minmax para la toma de decisiones en la partida.

4. Primera versión del Deep Blue de IBM.

Red neuronal artificial (RNA).

Esta implementación de IA es la más complicada de las mencionadas hasta ahora, las RNAs (Red Neuronal Artificial) se basan en intentar resolver los problemas como lo hace el cerebro humano.
Este tipo de IA se basa en un aprendizaje automático en el que el sistema se forma a sí mismo. Para realizar este aprendizaje lo que se hace es intentar minimizar una función de perdida mediante el método de propagación hacia atrás.
Las RNAs se utilizan para cosas como el reconocimiento de voz, reconocimiento facial o detección de enfermedades como el cáncer.
En el ámbito de los videojuegos se podrían utilizar por ejemplo para que un NPC aprenda a como pelea un personaje para ser capaz de pelear mejor contra él prediciendo por ejemplo los movimientos que puede realizar siguiendo un patrón.

Tipos de la IA de las anteriores utilizadas en el trabajo.

En este trabajo, tras estudiar los diferentes tipos de IA, he decidido utilizar los árboles de decisión para el tratamiento de los NPCs cuya función es dialogar con el jugador y reaccionar de una manera o de otra según las respuestas del jugador o según si el jugador por ejemplo ha cumplido una misión o no etc.

Por otro lado, también he utilizado el sistema de máquinas de estado, ya que Unity nos ofrece la posibilidad de dotar de comportamientos según inputs recibidos mediante máquinas de estado de manera bastante sencilla e incluso con una interfaz gráfica a través de la utilización en el objeto del componente AnimatorController. Como ejemplo de máquina de estados en el trabajo tenemos un NPC que es una madre que se comporta como una persona preocupada porque ha perdido a su hijo y una vez que se lo llevamos y matamos a una maga que la tiene atrapada pasa a otro estado en el que está feliz, este comportamiento lo conseguimos utilizando las máquinas de estado.

Para el sistema de pathfinding, Unity ya nos ofrece facilidades para mover un personaje por ejemplo en patrulla de punto a punto de la manera más optima implementando ya de por sí el algoritmo A*.

Descripción detallada de cada uno de los NPCs utilizados.

Los diferentes Personajes creados en este trabajo son los siguientes:

- **PaisanoWarrok:**

El objetivo de este personaje es dialogar con nosotros para preguntarnos si le podemos ayudar con uno de los secuaces de Maw Jaygo y en caso afirmativo si cumplimos con la misión nos dará dinero y si no aceptamos la misión podremos volver a preguntarle para hacer la misión.

- **PaisanoAmigoTabernero:**

El objetivo de este personaje es dialogar con nosotros para en caso de que hayamos vencido al secuaz Warrok de Maw Jaygo nos manda a hablar con su amigo tabernero que nos puede llevar a pelear contra Maw Jaygo. En caso de no haber vencido al Warrok no querrá decirnos nada interesante.

- **Maga:**

Este personaje es un personaje contra el que tendremos que luchar el cuál lanza bolas de fuego para atacarnos, aparecerá cuando llevemos al niño con su madre y nos pida que le ayudemos.

- **Tabernero:**

Personaje que en caso de que hayamos cumplido una serie de misiones, nos dará la opción de teletransportarnos para pelear contra Maw Jaygo (El boss final). Si decidimos teletransportarnos nos pedirá 100 monedas en caso de no tenerlas no podremos teletransportarnos y en caso afirmativo nos teletransportará.

- **Niño:**

Este personaje es un niño asustado porque se ha perdido y no sabe volver a casa con su madre, cuando interactuamos con él nos pide ayuda para llevarle a casa, si aceptamos nos seguirá hasta que le llevemos con su madre.

- **Madre:**

Este NPC es la madre que ha perdido el niño y está preocupada inicialmente debido a que no encuentra a su hijo, si hablamos con ella nos dirá que no encuentra a su hijo. Una vez que le llevamos a su hijo (El niño descrito anteriormente) nos lo agradecerá y nos pedirá que le ayudemos para derrotar a la maga descrita anteriormente ya que no la deja irse de allí.

- **Enemigo básico:**

Los enemigos básicos son varios NPCs que se encuentran patrullando por el mapa y en caso de que entremos en su campo de visión pelearan con nosotros y en caso de matarles nos darán un loot aleatorio.

- **Tendero:**

Este personaje nos ofrece la posibilidad de comprar objetos para hacernos más fuertes o curar a nuestro personaje dependiendo del objeto. En caso de que el **o**bjeto que queremos comprar cuesta más dinero del que tenemos o tenemos el inventario lleno, No nos dejará comprarlo y nos avisará con un mensaje.

- **Maw Jaygo:**

Este personaje es el boss final o el enemigo más difícil y el que cuando lo matamos finaliza el juego. Este personaje tiene diferentes tipos de ataque que realiza según la distancia a la que se encuentra el jugador y aparte también tiene diferentes ataques según en la fase en la que se encuentre. Este NPC pasa de la fase 1 a la fase 2 cuando la vida baja de un valor.

- **Aj:**

Aj es el personaje controlado por el jugador el cuál se puede desplazar por el mapa, interactuar con los diferentes personajes y pelear con los diferentes personajes que son enemigos a este, así como equiparse los objetos que tenga en el inventario o en caso de ser consumibles consumirlos para curarse o hacerse más fuerte.

Otro tipo de NPCs que podríamos mencionar, aunque no son personajes como tal son los objetos los cuáles para conseguirlos puede ser de las siguientes maneras: al derrotar alguno de los enemigos del mapa o comprándolos con dinero en la tienda. Estos existen como iconos 2d en el inventario del personaje y en la tienda y en 3d para cuando matamos a un personaje y aparecen como loot. Estos objetos pueden ser acumulables o no, así como equipables o consumibles y en caso de ser equipables, sólo los podremos equipar en la parte del cuerpo que les corresponda.

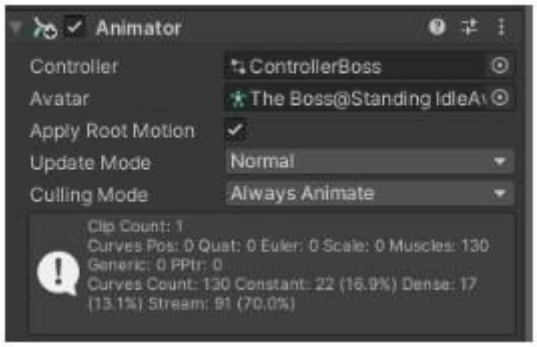
Ahora vamos a pasar a ir describiendo detalladamente cómo están hechos los diferentes personajes sean NPCs o el personaje controlado por el jugador.

- **PaisanoWarrok:**

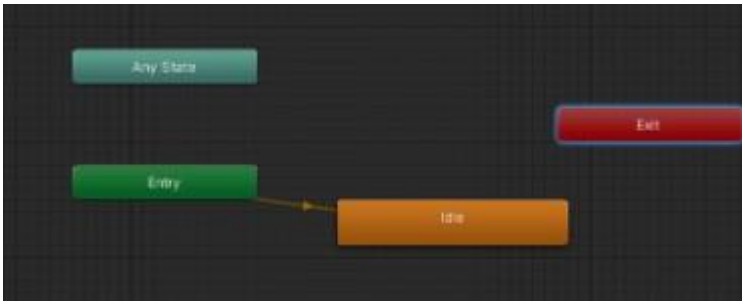
Descripción:
El funcionamiento de este NPC se basa en una máquina de estados y un árbol de decisión que en conjunto consiguen la funcionalidad completa. Aquí podemos ver la máquina de estados que se utiliza para este NPC. En primer lugar, vemos que, al arrancar el juego, el personaje se encontrará en el estado Idle, es decir quieto esperando hasta que un input le haga cambiar de estado. Aunque en el caso de este personaje, sólo tiene este estado y nunca cambiará.
En este personaje la parte interesante está en el árbol de decisión, ya que será el que dotará de inteligencia al personaje.

Componentes:

- Animator:



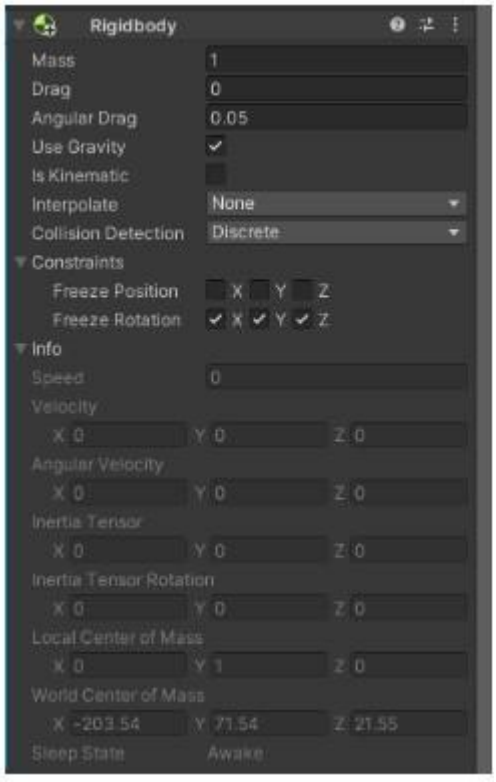
Este componente se ocupa a través del controller **ControllerBoss** de gestionar los diferentes estados en los que puede estar el personaje.
Este controller es una máquina de estados configurada de la siguiente manera:



5. Máquina de estados para el paisano Warrok controller.

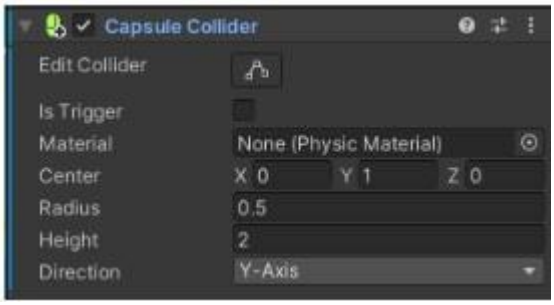
En este caso podemos ver que hay sólo un estado aparte de los ofrecidos por Unity en todos los controles que es el estado Idle y una única transición que va de **Entry -> Idle.**
Como hemos dicho en la descripción anteriormente, esta máquina de estados realmente no aporta una IA como tal al personaje, ya que da igual el input que reciba que siempre se va a mantener en el estado Idle el personaje.

- Rigidbody:



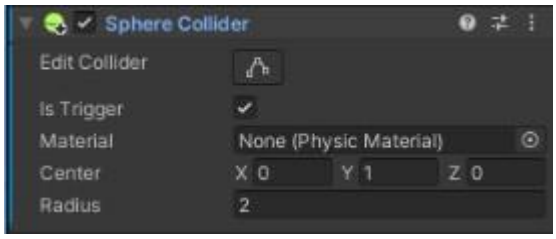
6. Componente Rigidbody del personaje PaisanoWarrok.

- Capsule Collider:



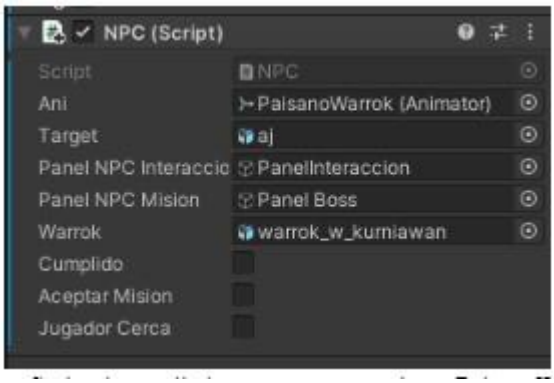
7. Componente capsule collider del personaje PaisanoWarrok.

- Sphere Collider:



8. Componente Sphere collider del personaje PaisanoWarrok.

- NPC (Script):



9. Script desarrollado para comportamiento PaisanoWarrok.

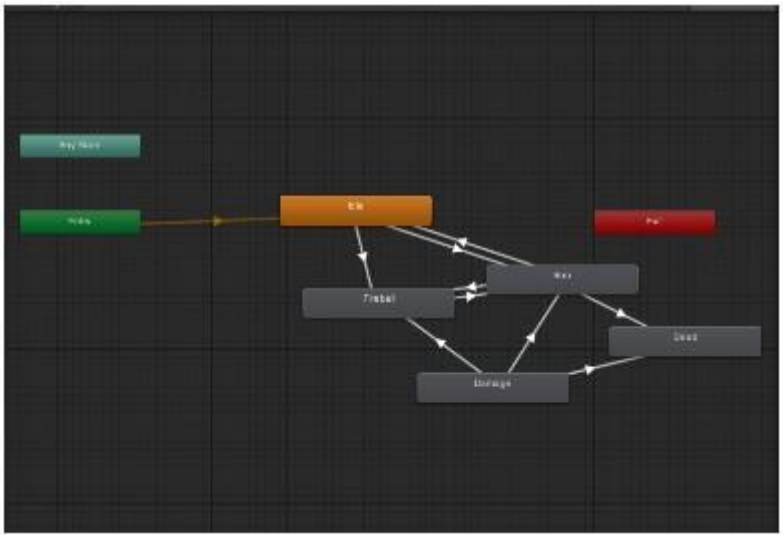
- **Maga:**

Descripción:
La maga es un NPC que sólo aparece cuando llevamos al niño que está perdido con su madre y la madre nos pide ayuda con la maga que no le deja irse de ahí. La IA de este NPC se basa en una máquina de estados. Cuando aparece en el mapa, en un principio está quieto esperando a que el jugador entre en su campo de visión y una vez este entra en su campo de visión se acerca hasta la distancia en la que nos puede atacar con una bola de fuego.

Componentes:

- Animator:

10. Componente Animator del personaje Maga.



11. Controller del personaje Maga.

Lo primero que vamos a hacer para explicar el diagrama es mostrar las diferentes animaciones que van asociadas a cada estado.

Estados:

- Idle:

Como podemos ver en la imagen siguiente, mientras el personaje se encuentre en este estado, se reproducirá la animación idle hasta que
pasemos al siguiente estado y cambie el estado y por tanto la animación. Marcamos la casilla Loop time para que esta animación no
se ejecute una única vez si no que se esté ejecutando constantemente mientras que estemos en este estado.

12. Captura del estado Idle del personaje Maga.

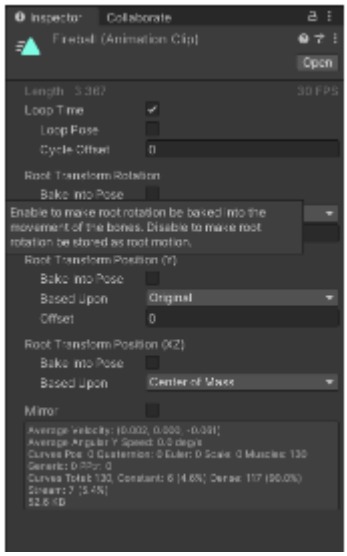
- Run:

En este estado, la animación que se ejecutará será la animación injured run al igual que comentamos en el estado idle, marcaremos la casilla loop time para que se ejecute en bucle hasta que salgamos del estado.

13. Captura del estado Run del personaje Maga.

- Fireball:

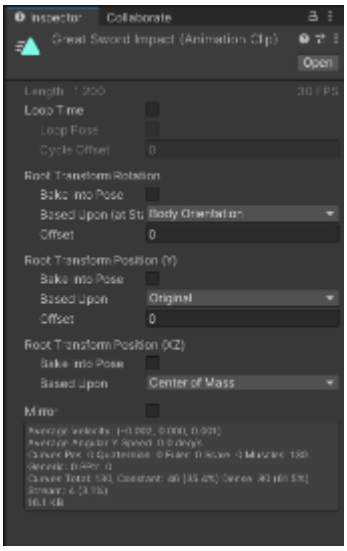
En este estado, se ejecutará la animación fireball y marcaremos como en los anteriores la casilla loop time por las mismas razones que en los estados anteriores.



14. Captura de la animación ejecutada en el estado Fireball del personaje Maga.

- Damage:

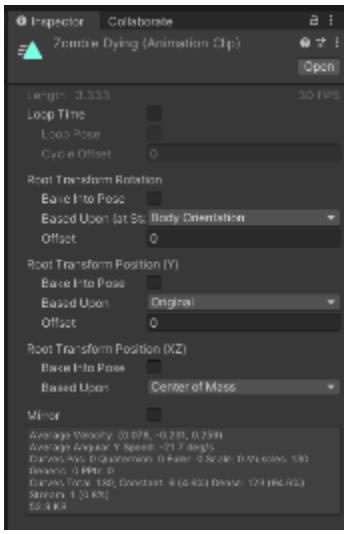
En el estado Damage, la animación ejecutada será la animación Great Sword Impact. En este caso la casilla Loop time queda desmarcada ya que sólo queremos que se reproduzca la animación una vez en el
momento que este reciba un golpe de nuestro personaje.



15. Captura de la animación ejecutada en el estado Damage del personaje Maga.

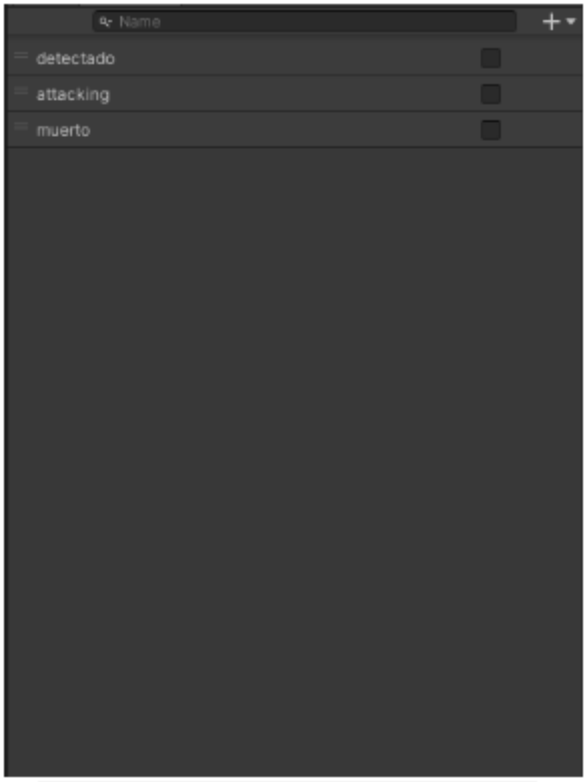
- Dead:

En este estado se ejecutará la animación Zombie Dying. En este caso la casilla loop time también quedará desmarcada porque al ser una
animación de muerte, sólo queremos que se ejecute una vez cuando el personaje muere y no en bucle.



16. Captura de la animación Zombie Dying ejecutada en el estado Dead del personaje Maga.

Para continuar, adjunto una imagen de los distintos parámetros que son los que van a hacer de triggers para que se activen las transiciones de un estado a otro para poder explicar mejor las transiciones.



17. Parámetros del controller del personaje Maga.

Una vez explicados los diferentes estados posibles con sus animaciones y los parámetros que existen en el controller, vamos a pasar a explicar las diferentes transiciones que hay entre estados.

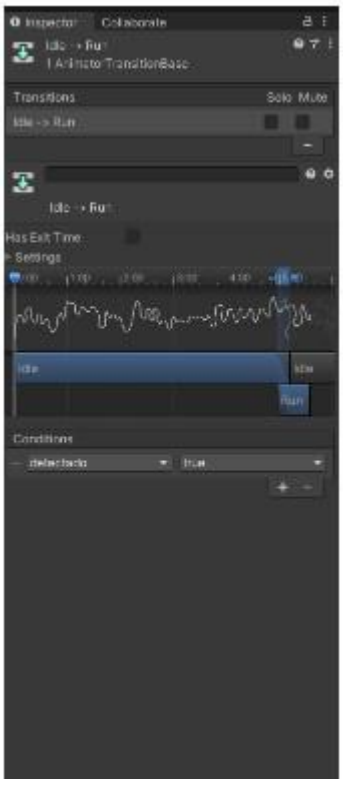
Transiciones:

- Idle -> Run:

Esta transición se ejecutará en el momento en que el valor del parámetro detectado sea igual a true. Esto ocurrirá en el momento en el que nuestro personaje entre en el rango de detección de la maga y gestionado por el script de comportamiento (MagaScript) como podemos ver en el siguiente fragmento de código en el que podemos ver que si la distancia entre el jugador y el NPC es menor que la máxima distancia de detección que hayamos puesto a la Maga que en nuestro caso será 10 el parámetro detectado pasará a ser true y en caso contrario el parámetro detectado pasará a ser false.

18. Captura del script MagaScript para gestión parámetro detectado.

En cuanto a la casilla Has exit time de la transición la dejaremos desmarcada, ya que no queremos obligar a que se tenga que ejecutar toda la animación de Idle para empezar a ejecutarse la animación de Run.



19. Transición del estado Idle -> Run personaje Maga.

- Idle -> Fireball

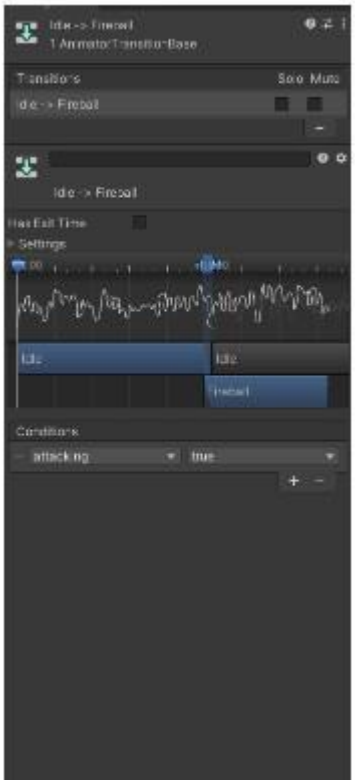
Esta transición, se ejecuta en el momento en el que estando en el estado Idle el parámetro attacking del controller pasa a valer true y por tanto se pasa al estado fireball ejecutandose la animación fireball.
Esta transición se gestiona en el script MagaScript en este fragmento de código que es el caso en el que nos encontremos a una distancia de detección más pequeña de

la mínima distancia a la que se considera detectado al jugador, si el parámetro attacking sigue a true se seguirá ejecutando la transición.

```
if(distancia <= minDist) {
    detectado = false;
    transform.LookAt(rotacion);
    ani.SetBool("attacking", true);
}
```

20. Captura del script MagaScript para gestión parámetro detectado en caso de estar más cerca de la distancia mínima.

En este caso tampoco marcaremos la opción has exit time ya que no queremos que la acción del estado Idle tenga que ser ejecutada completamente para poder iniciar la acción de ataque.



21. Transición del estado Idle -> Fireball personaje Maga.

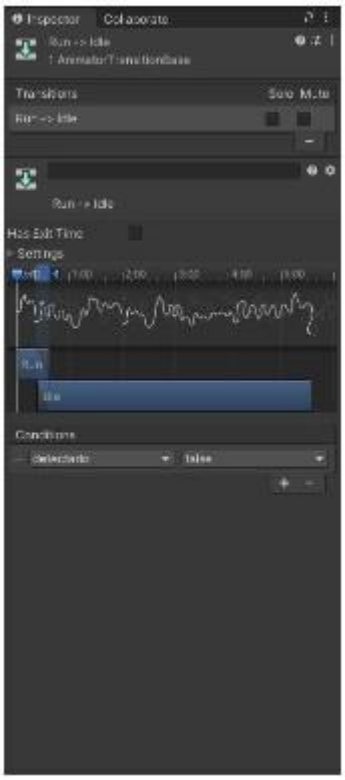
- Run -> Idle.

Esta transición será ejecutada en el momento en el que mientras la maga está corriendo para perseguir a nuestro personaje, este sale de su campo de detección y por tanto se para, pasando del estado Run al estado Idle. El trigger de esta transición, es el paso del parámetro detectado del controlador de true a false. La gestión de este

paso de un estado a otro, también se gestiona en el script MagaScript en este fragmento de código.

22. Captura del script MagaScript para gestión parámetro detectado en caso de estar el jugador a una distancia mayor que la distancia máxima establecida de campo de visión.

En esta transición al igual que en las anteriores no se marcará la opción has exit time, ya que queremos que en el momento que se deje de detectar a nuestro personaje la maga se quede parada y no tenga que terminar la animación de correr.



23. Transición del estado Run -> Idle personaje Maga

- Run -> Fireball

Para que esta transición sea lanzada, lo que tiene que ocurrir es que mientras la maga nos persigue, consigue acercarse a nosotros a la distancia estipulada de ataque que le queramos poner, se activa la transición y lanza el ataque de fireball.

```
        }
        if(distancia <= minDist) {
            detectado = false;
            transform.LookAt(rotacion);
            ani.SetBool("attacking", true);
        }
```

24. Captura del script MagaScript para gestión parámetro attacking pasando a true en caso de estar el jugador a la mínima distancia necesaria para lanzar el ataque.

En este caso, tampoco activaremos la opción Has exit time, por las mismas razones mencionadas anteriormente.

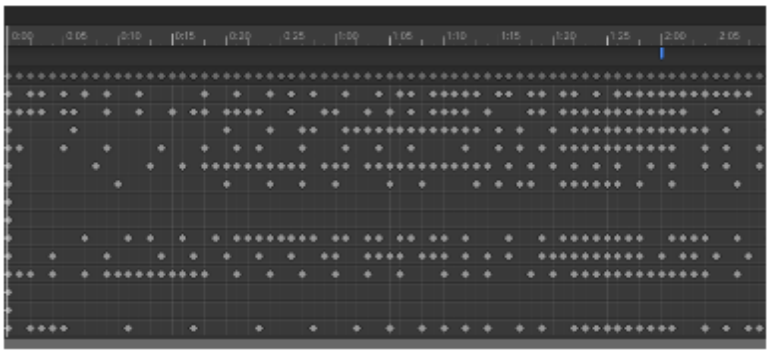
25. Transición del estado Run -> Fireball personaje Maga

- Fireball -> Run

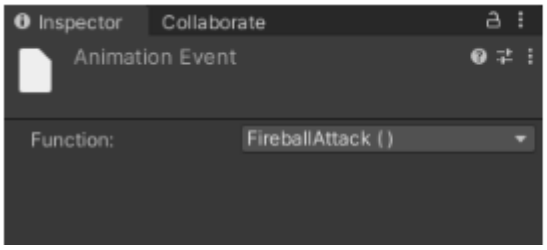
Esta transición se ejecutará en el caso de que después de lanzar el ataque, el personaje siga dentro del campo de detección.
En este caso, como lo que queremos es modificar el parametro attacking sólo una vez que se haya lanzado el ataque, crearemos una función que añadiremos como evento en la animación del lanzamiento de la bola en el momento que la bola es lanzada. Esta gestión se hará tanto desde el script MagaScript como desde la interfaz de Unity en la pestaña Animation para añadir el evento. Captura del script MagaScript para gestión parámetro attacking pasando a false en caso de que la bola ya haya sido lanzada.

```
void FireballAttack() {
    Instantiate(fireball, salidaMagia.position, salidaMagia.rotation);
    ani.SetBool("attacking", false);
}
```

26. Captura del script MagaScript para gestión parámetro attacking pasando a false en caso de que la bola ya haya sido lanzada.



27. Captura de la pestaña animation con el evento añadido en el minuto 2.



28. Captura del evento con la función asignada.

En este caso, sí que marcaremos la opción Has exit time ya que queremos que la animación se ejecute completamente antes de volver al estado Run.

28. Transición del estado Fireball -> Run personaje Maga

- Damage -> Fireball

Esta transición ocurre cuando después de recibir daño, si hp sigue siendo mayor que 0, el parámetro muerto será false y volverá al estado de ataque fireball. Esto se gestiona en el script personaje en un método que, si el tag del objeto al que ataca es el de la maga, se pasa al estado Damage en el controller de la maga y por tanto se ejecuta la animación de recibir daño.

29. Captura del script Personaje para lanzamiento del estado Damage en caso de que el personaje al que atacamos sea la maga (case “Mage”).

Para este caso, también dejaremos marcada la casilla Has exit time ya que queremos que la animación de daño se ejecute por completo antes de volver a atacar.

30. Transición del estado Damage -> Fireball personaje Maga

- Damage -> Run

Esta transición ocurre cuando después de recibir daño, si hp sigue siendo mayor que 0, si el parámetro detectado es igual a true, pasará al estado Run.
Para llegar al estado Damage al igual que en la transición anterior, se gestiona en el script Personaje.
Esta gestión se hace en el script MagaScript en este fragmento de código:

31. Captura del script MagaScript para la gestión del parámetro detectado.

En este caso también dejaremos marcada la opción Has exit time.

32. Transición del estado Damage -> Run personaje Maga

- Damage -> Dead

Si después de ejecutar desde el script personaje la animación de daño el atributo hp de la maga es menor o igual que 0, el parámetro muerto pasará a true y se pasará al estado Dead que ejecutará la animación de muerte.
Esta gestión se hace en este fragmento de código del script MagaScript:

```
if(hp <= 0) {
    hp = 0;
    detectado = false;
    ani.SetBool("detectado", false);
    ani.SetBool("attacking", false);
    ani.SetBool("muerto", true);
    Destroy(this.gameObject.GetComponent<MagaScript>());
    Destroy(this.gameObject.GetComponent<CharacterController>());
    Destroy(this.gameObject, 10);
}
```

33. Captura del script MagaScript para la gestión de la muerte de la maga.

Por las mismas razones que en las anteriores transiciones Damage -> X, dejaremos la casilla Has exit time marcada.

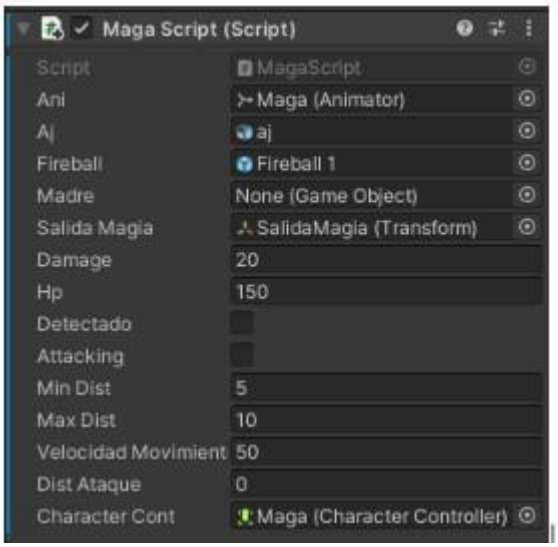
34. Transición del estado Damage -> Dead personaje Maga

- Run -> Dead

Esta transición se ejecuta si mientras estamos en el estado Run, el parámetro muerto pasa a true, la gestión del parámetro muerto es igual que para la anterior transición. En este caso la casilla Has exit time no la marcamos ya que si el personaje muere mientras está corriendo no queremos que termine de ejecutar la acción de correr antes de morir si no que muera en el mismo momento que su vida pase a valer 0.

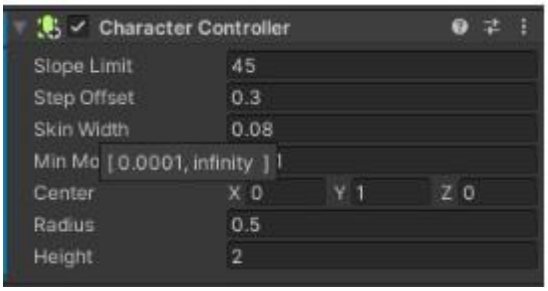
35. Transición del estado Run -> Dead personaje Maga

- **MagaScript:**



36. Script para gestionar el comportamiento de la Maga.

- CharacterController:



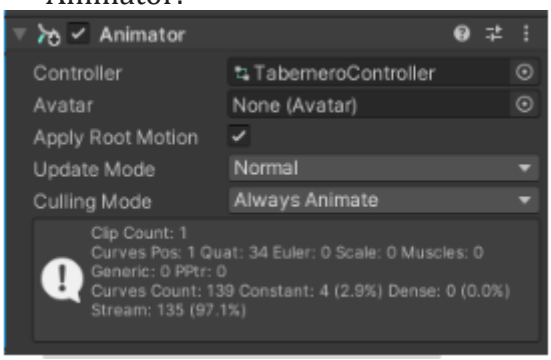
37. Componente CharacterController de la Maga.

- **Tabernero:**

Descripción:
Este personaje al igual que el paisano, basa su IA en un árbol de decisión y su funcionalidad es esperar a que el personaje interactúe con él y cuando esto ocurre, si no vas enviado por un amigo suyo al intentar hablar con él te dice que no le hagas perder el tiempo y en caso de que sí que vayas enviado por su amigo, nos pregunta que si queremos nos lleva a pelear contra Maw Jaygo por cien monedas y si aceptamos y tenemos el dinero suficiente, nos llevará a su ubicación y si aceptamos pero no tenemos el dinero suficiente lo que ocurrirá será que nos dirá que no tenemos dinero y que volvamos cuando lo tengamos. Si decidimos no querer teletransportarnos, lo que hará será cerrarse el mensaje y preguntarnos si queremos hablar con él como cuando nos acercamos. La gestión de la IA del personaje se realiza a través de un árbol de decisión.

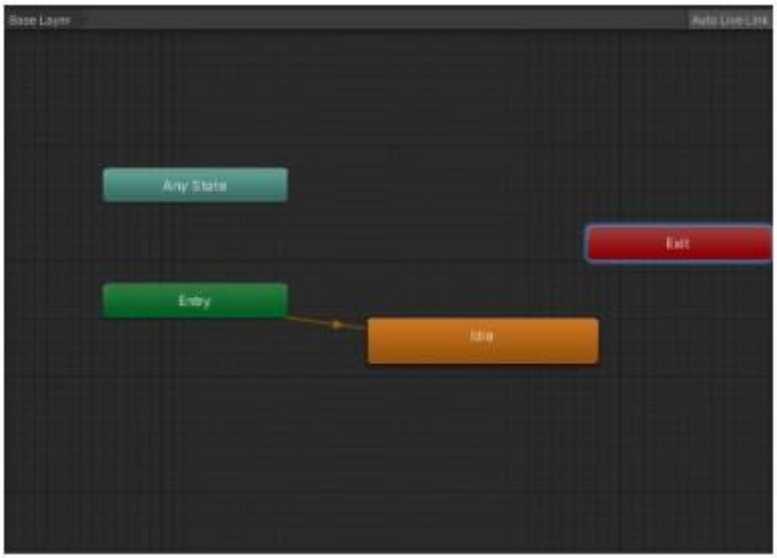
Componentes:

- Animator:



38. Componente Animator del tabernero.

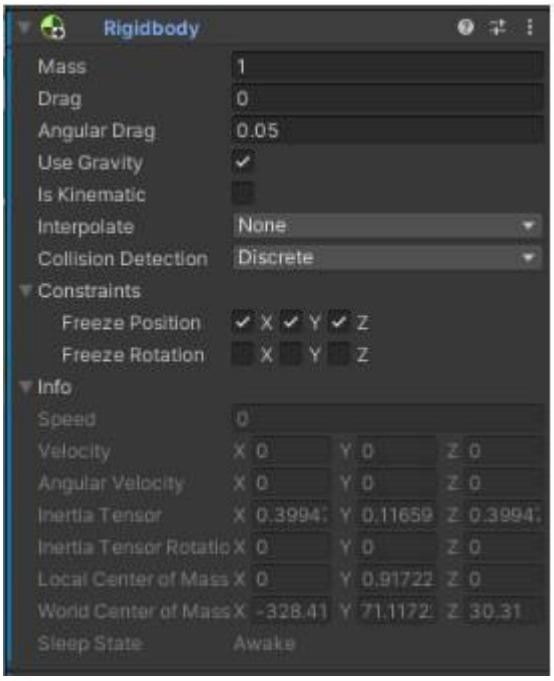
Este componente se ocupa a través del controller TabernеroController de gestionar los diferentes estados en los que puede estar el personaje. Este controller es una máquina de estados configurada de la siguiente manera:



39. Máquina de estados para el TabernеroController.

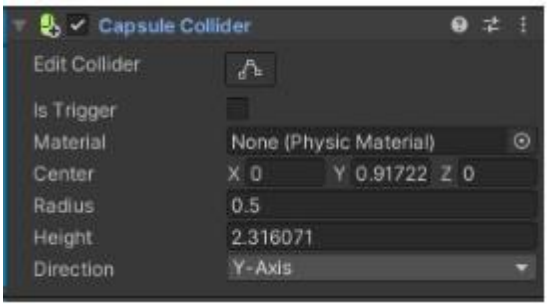
En este caso podemos ver que hay sólo un estado aparte de los ofrecidos por Unity en todos los controles que es el estado Idle y una única transición que va de Entry -> Idle.
Como hemos dicho en la descripción anteriormente, esta máquina de estados realmente no aporta una IA como tal al NPC, ya que da igual el input que reciba que siempre se va a mantener en el estado Idle.

- Rigidbody:



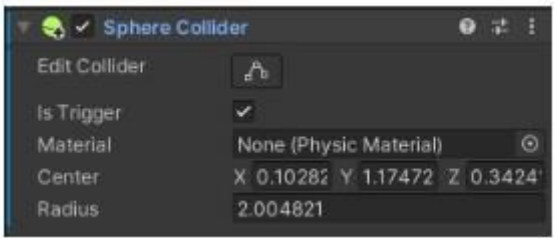
40. Componente Rigidbody del personaje Tabernero.

- Capsule Collider:



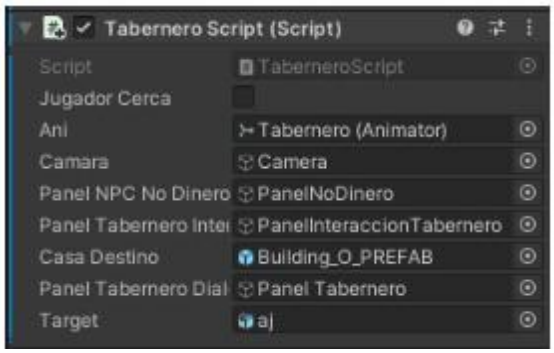
41. Componente capsule collider del personaje Tabernero.

- Sphere Collider:



42. Componente Sphere collider del personaje Tabernero.

- NPC (Script):

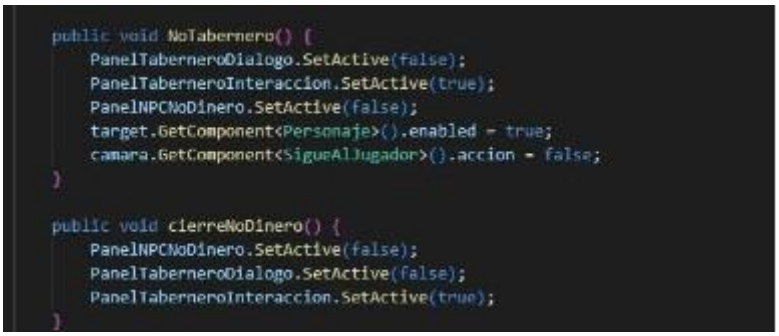


43.Script desarrollado para comportamiento Tabernero.

En este script, se gestiona el árbol de decisión del tabernero.
En este árbol, el NPC empieza quieto esperando a que el jugador se acerque hablar con él, una vez nos acercamos nos da la opción de interactuar con él con el mensaje: "Presiona "X" para hablar".
En este caso en el árbol ya tendríamos dos posibles destinos:

» Si presionamos X, habrá dos posibilidades que hayamos hablado ya con el paisano amigo suyo y por tanto nos ofrezca la posibilidad de teletransportarnos a la ubicación del boss mostrando el siguiente mensaje: "Con que has vencido al Warrok secuaz de Maw Jaygo, Interesante... ¿Te ves listo para pelear con Maw Jaygo, por 100 monedas podría llevarte contra él?". Si en este mensaje le decimos que sí, en caso de tener las 100 monedas nos teletransportará y en caso de no tenerlas nos dirá que no tenemos dinero suficiente. Si le decimos que no, se volverá a mostrar el panel con el mensaje "Presiona "X" para hablar".
La otra posibilidad es si al presionar X, no hemos cumplido la misión que nos manda su amigo y hablado con él nos mostrará el siguiente mensaje y no nos ofrecerá teletransportarnos: "No me hagas perder el tiempo por favor."

44. Captura TabarneroScript gestión del árbol de decisión



```
public void NoTabernero() {
    PanelTabarneroDialogo.SetActive(false);
    PanelTabarneroInteraccion.SetActive(true);
    PanelNPCNoDinero.SetActive(false);
    target.GetComponent<Personaje>().enabled = true;
    camara.GetComponent<SigueAlJugador>().accion = false;
}

public void cierreNoDinero() {
    PanelNPCNoDinero.SetActive(false);
    PanelTabarneroDialogo.SetActive(false);
    PanelTabarneroInteraccion.SetActive(true);
}
```

45.Captura NinoScript para gestión de los botones.

» Si en vez de pulsar X el jugador se aleja del Tabernero el mensaje desaparece.

```
private void OnTriggerEnter(Collider other) {
    if(other.tag == "Player"){
        jugadorCerca = true;
        PanelTabarneroInteraccion.SetActive(true);
    }
}

private void OnTriggerExit(Collider other) {
    if(other.tag == "Player") {
        jugadorCerca = false;
        PanelTabarneroInteraccion.SetActive(false);
        PanelTabarneroDialogo.SetActive(false);
        PanelNPCNoDinero.SetActive(false);
    }
}
```

46. Captura TabarneroScript gestión de triggers del Sphere Collider.

Como vemos en la captura anterior, la gestión de si el jugador está cerca o no se hace a través del Sphere Collider, en este caso lo que haremos será comprobar con OnTriggerEnter si hemos entrado en la Sphere y con OnTriggerExit si hemos salido de la Sphere.

- **Niño:**

Descripción:
El personaje lo podemos encontrar desde el momento en que se inicia el juego y en un primer lugar está en un estado asustado y cuando interactúas con él te pide por favor que lleves a casa con su madre porque se ha perdido, si aceptas la misión, el niño te sigue hasta que llegas hasta el punto en el que se encuentra su madre (Que es el NPC madre que describiremos posteriormente) y una vez le llevas allí deja de seguirte y se queda en ese punto. La IA de este NPC se basa en el uso de un árbol de decisión y una máquina de estados.

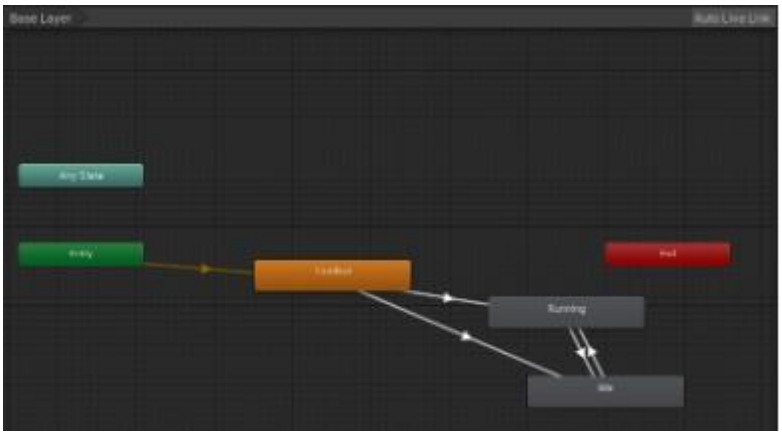
Componentes:

- Animator:

47. Componente Animator del Niño.

Este componente se ocupa a través del controller ControllerNiño de gestionar los diferentes estados en los que puede estar el personaje.

Este controller es una máquina de estados configurada de la siguiente manera:



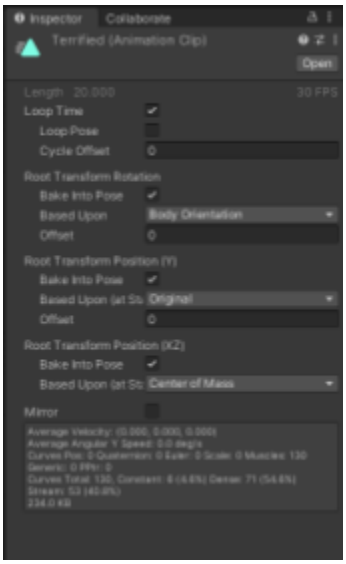
48. Máquina de estados para el ControllerNiño.

Estados:

- Terrified:

Este estado es el estado inicial del NPC Niño, y como vemos en la siguiente imagen, puede pasar al estado Running o Idle dependiendo del valor de ciertos parámetros que mencionaremos posteriormente.

En este estado la animación ejecutada será Terrified:

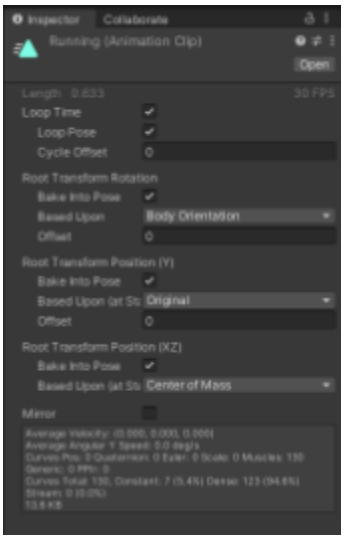


49. Animación Terrified para el NPC Niño.

Podemos ver la casilla Loop time marcada, ya que queremos que mientras estemos en este estado la animación se ejecute en bucle.

- Running:

A este estado podemos llegar tanto desde el estado Idle como desde el estado Terrified, y llegamos a él a través de las transiciones que explicaremos posteriormente. La animación que se ejecutará en este estado es la animación Running.

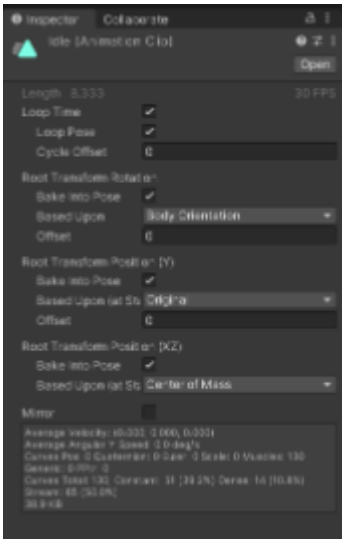


50. Animación running para el NPC Niño.

Podemos ver que está marcada la casilla Loop Time, ya que queremos que mientras estemos en este estado se ejecute todo el rato la animación de correr.

- Idle:

A este estado podemos llegar tanto desde el estado Terrified como desde el estado Running, la animación que se ejecutará en este estado es la animación Idle.



51. Animación Idle para el NPC Niño.

Transiciones

- Terrified -> Idle

Esta transición será lanzada dependiendo del valor de los parámetros Following y Running. En el momento en que Following valga true y Running valga false, se pasa al estado Idle.

52. Transición Terrified -> Idle ControllerNiño.

- Terrified -> Running.

Esta transición tendrá lugar en el momento que estando en el estado Terrified, los parámetros Following y Running pasan al estado true.

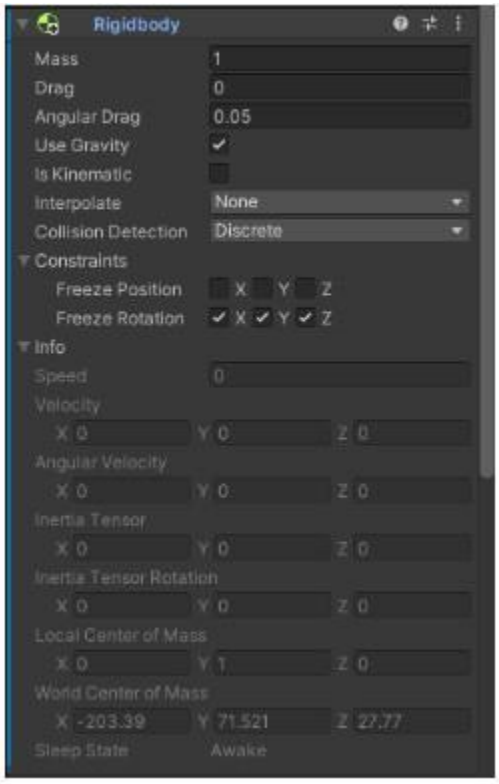
53. Transición Terrified -> Running ControllerNiño.

- Running -> Idle

Esta transición ocurre cuando estando en el estado Running, el parámetro Running pasa a ser false.

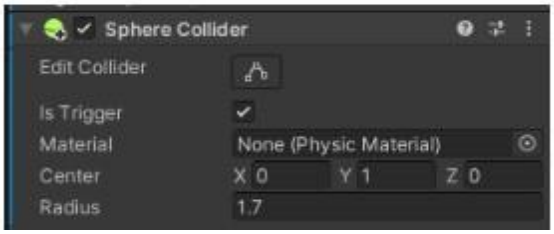
54. Transición Idle-> Running ControllerNiño.

- Rigidbody:



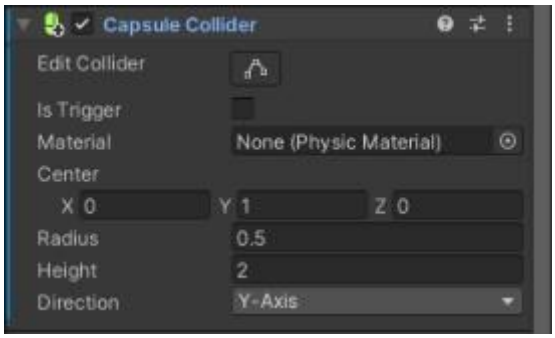
55. Captura Rigidbody NPC Niño.

- Sphere Collider:



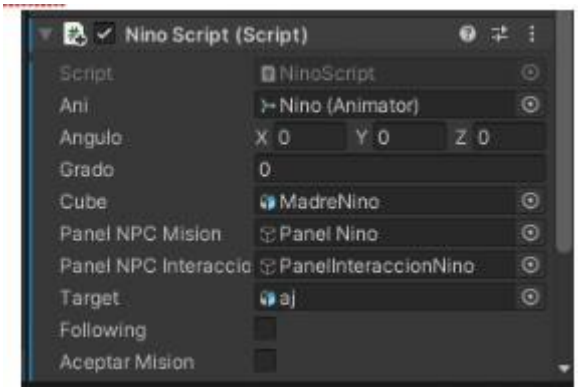
56. Captura Sphere collider NPC Niño.

- Capsule Collider:



57. Captura Capsule collider NPC Niño.

- NinoScript:



58. Captura NinoScript NPC Niño.

En este script, se gestiona el árbol de decisión del niño.
En este árbol, el NPC empieza en quieto aterrorizado esperando a que el jugador se acerque hablar con él, una vez nos acercamos nos da la opción de interactuar con él con el mensaje: "Presiona "X" para hablar".
En este caso en el árbol ya tendríamos dos posibles destinos:

» Si presionamos X, continuaremos el diálogo pasando al siguiente nodo, y se nos mostrará el siguiente mensaje del niño pidiendonos ayuda: "¿Me llevas a casa? Me he perdido". Si en este mensaje le decimos que sí, el niño pasará al estado following cambiando otra vez de nodo y nos seguirá hasta llevarle con su madre. Si decimos que no, volveremos al nodo anterior en el que se nos pide pulsar X para hablar.

59. Captura NinoScript gestión del árbol de decisión.

```
public void No() {
    PanelNPCMision.SetActive(false);
    PanelNPCInteraccion.SetActive(true);
}

public void Si() {
    PanelNPCInteraccion.SetActive(false);
    PanelNPCMision.SetActive(false);
    ani.SetBool("Following", true);
    following = true;
    aceptarMision = true;
    jugadorCerca = false;
}
```

60. Captura NinoScript para gestión de si aceptamos o no la misión.

» Si en vez de pulsar X el jugador se aleja del niño el mensaje desaparece.

```
private void OnTriggerEnter(Collider other) {
    if(other.tag == "Player"){
        jugadorCerca = true;
        if(!aceptarMision){
            PanelNPCInteraccion.SetActive(true);
        }
    }
}

private void OnTriggerExit(Collider other) {
    if(other.tag == "Player") {
        jugadorCerca = false;
        PanelNPCInteraccion.SetActive(false);
        PanelNPCMision.SetActive(false);
    }
}
```

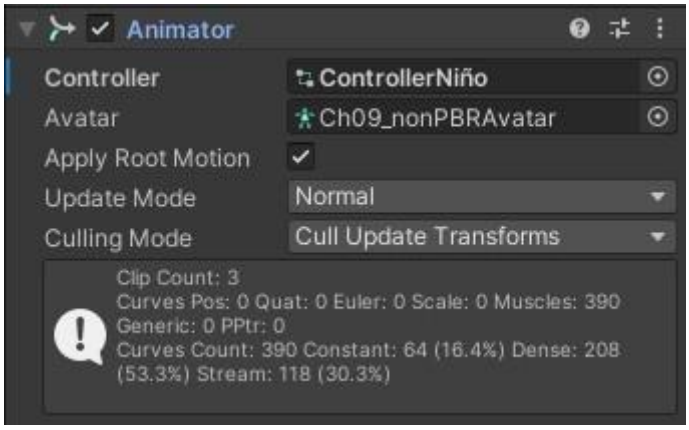
61. Captura NinoScript gestión de triggers del Sphere Collider.

Como vemos en la captura anterior, la gestión de si el jugador está cerca o no se hace a través del Sphere Collider, en este caso lo que haremos será comprobar con OnTriggerEnter si hemos entrado en la Sphere y con OnTriggerExit si hemos salido de la Sphere.

- **Madre:**

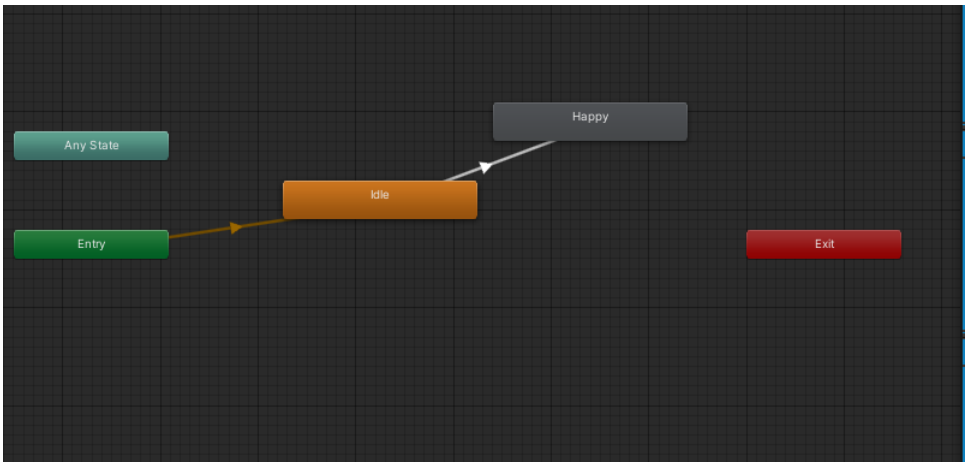
Descripción:
Como hemos dicho anteriormente en la descripción del personaje del niño, al principio la encontraremos asustada y preocupada debido a que no encuentra a su hijo, hasta que le llevamos a su hijo (Que es el niño descrito anteriormente) y una vez le llevamos a su hijo seguirá preocupada debido a que está la maga que no le deja irse de allí y nos pedirá ayuda para deshacernos de ella. Una vez vencemos a la maga, pasará a estar en un estado feliz y nos dará dinero por haberla ayudado. La IA al igual que para el niño se basa en una máquina de estados y un árbol de decisión modelando mediante estos dos métodos de IA el comportamiento del personaje.

Componentes:
- Animator:



62. Componente Animator del niño.

Este componente se ocupa a través del controller ControllerNiño de gestionar los diferentes estados en los que puede estar el personaje. Este controller es una máquina de estados configurada de la siguiente manera:

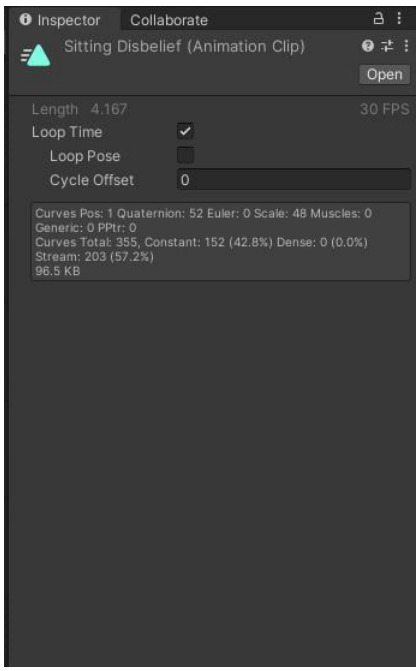


63. Máquina de estados para el MadreController.

Estados:

- Idle:

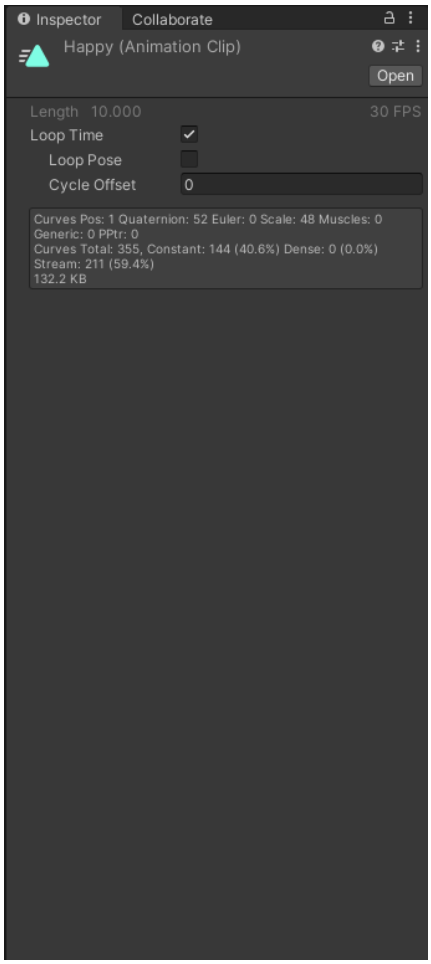
Este estado es el estado inicial del NPC Madre, y como vemos en la siguiente imagen, puede pasar al estado Happy dependiendo del valor de un parámetros que mencionaremos posteriormente.
En este estado la animación ejecutada será Sitting Disbelief:



64. Animación Sitting disbelief para el NPC Madre.

Podemos ver la casilla Loop time marcada, ya que queremos que mientras estemos en este estado la animación se ejecute en bucle.

Happy:
A este estado podemos llegar desde el estado Idle y llegamos a él a través de las transiciones que explicaremos posteriormente. La animación que se ejecutará en este estado es la animación Happy.



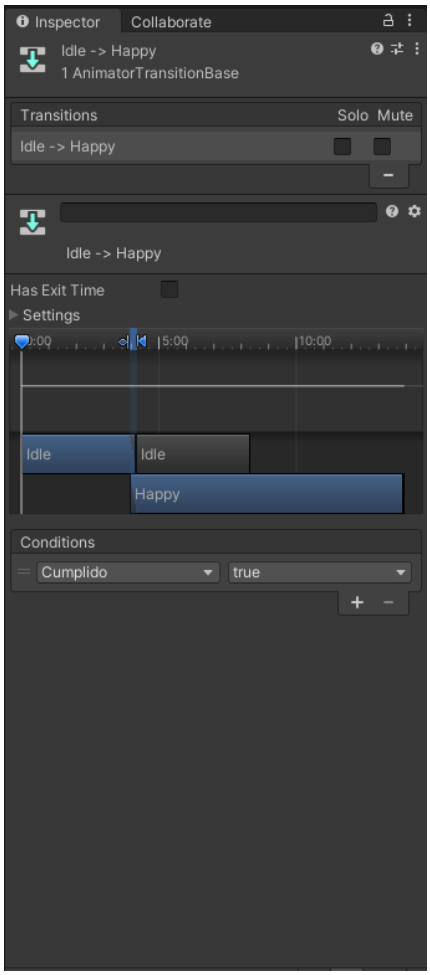
66. Animación Happy para el NPC Madre.

Podemos ver que está marcada la casilla Loop Time, ya que queremos que mientras estemos en este estado se ejecute todo el rato una vez hayamos matado a la maga.

Transiciones:

- Idle -> Happy.

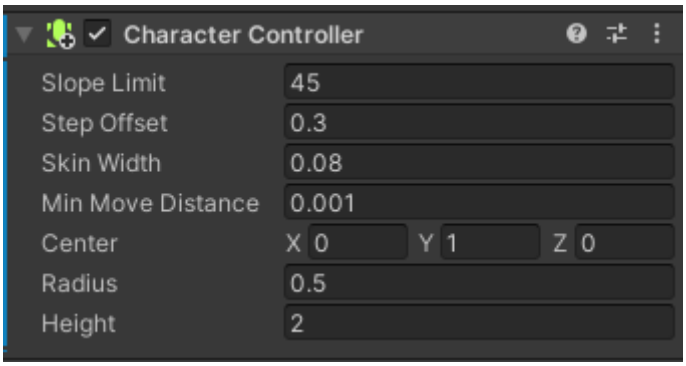
Esta transición depende del parámetro Cumplido que empieza en false, y pasará a true en el caso que hayamos matado a la maga. En el momento que este parámetro pasa a true se activa la transición y cambia de estado.



66. Transición Idle-> Happy NPC Madre.

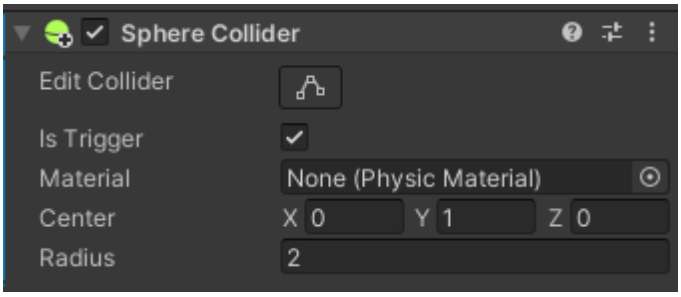
En este caso no marcaremos la casilla Has exit time ya que no queremos esperar a que termine la animación Idle para cambiar de estado.

- Character controller:



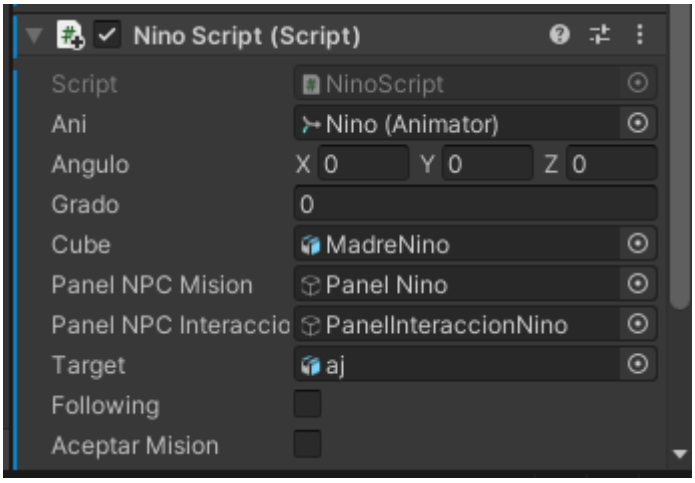
67. Captura CharacterController NPC Madre.

- Sphere Collider:



68. Captura Sphere collider NPC Madre.

- MadreScript:



69. Captura Madre Script NPC Madre.

En este script, se gestiona el árbol de decisión de la madre.
En este árbol, el NPC empieza quieta y preocupada esperando a que el jugador se acerque hablar con él, una vez nos acercamos nos da la opción de interactuar con ella con el mensaje: "Presiona "X" para hablar".
En este caso en el árbol ya tendríamos tres posibles destinos:

» Si presionamos X, continuaremos el diálogo pasando al siguiente nodo, y se nos mostrará el siguiente mensaje de la madre pidiéndonos ayuda: "He perdido a mi hijo por favor ayúdame a encontrarle".

Después de este mensaje sólo se nos da la opción de cerrar el diálogo.

» Si en vez de pulsar X el jugador se aleja de la madre el mensaje desaparece.

```
private void OnTriggerEnter(Collider other) {
    if(other.tag == "Player"){
        jugadorCerca = true;
        if(!aceptarMision){
            PanelNPCInteraccion.SetActive(true);
        }
    }
}

private void OnTriggerExit(Collider other) {
    if(other.tag == "Player") {
        jugadorCerca = false;
        PanelNPCInteraccion.SetActive(false);
        PanelNPCMision.SetActive(false);
    }
}
```

70.Captura MadreScript para la gestión de triggers del Sphere Collider.

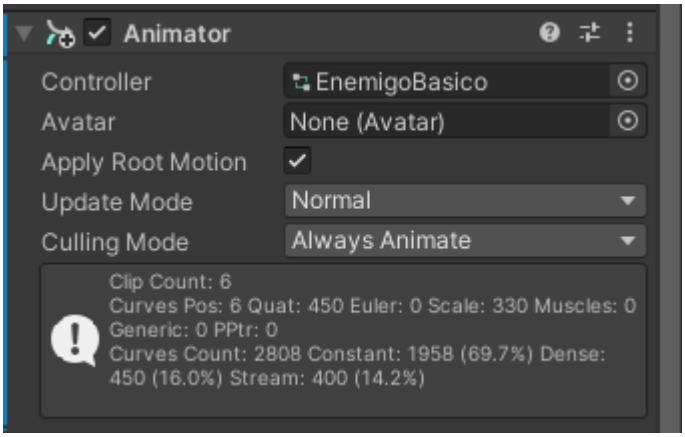
Como vemos en la captura anterior, la gestión de si el jugador está cerca o no se hace a través del Sphere Collider, en este caso lo que haremos será comprobar con OnTriggerEnter si hemos entrado en la Sphere y con OnTriggerExit si hemos salido de la Sphero.

» El último destino posible, es en caso de que hayamos llevado al niño con la madre en el que se muestra el mensaje "Gracias por traerme a mi hijo, pero esta maga no me deja huir. ¡Ayúdame por favor!".
Después de esto, se nos da la opción de cerrar el diálogo y pelear contra la maga.
Desde este nodo, si vencemos a la maga y volvemos a hablar con la madre, nos agradecerá haberla ayudado y nos dará dinero con el siguiente mensaje: "Gracias por ayudarme! Toma aquí tienes 20 Monedas."

- **Enemigo básico:**

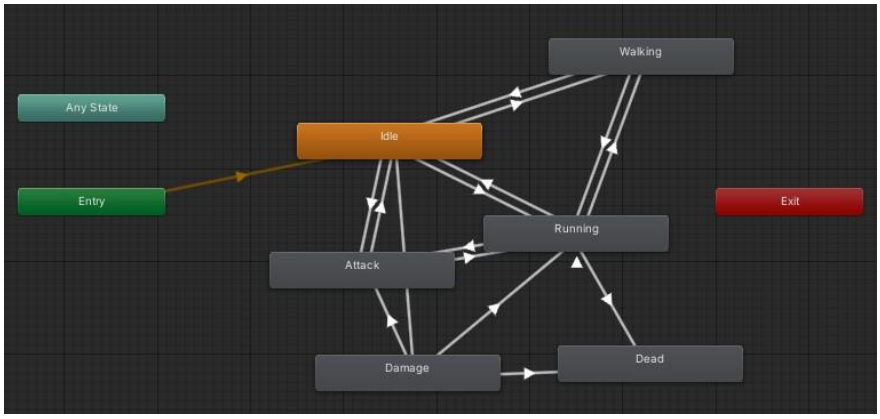
Descripción:
Los enemigos básicos, son enemigos que podemos encontrarnos por el mapa patrullando hasta que nuestro personaje entra en su campo de detección y le empieza a perseguir con la intención de matarle. Si se sale el personaje del campo de detección este nos dejará de perseguir y seguirá patrullando. Una vez matamos a estos personajes, nos dejan un loot aleatorio.

Componentes:
- Animator:

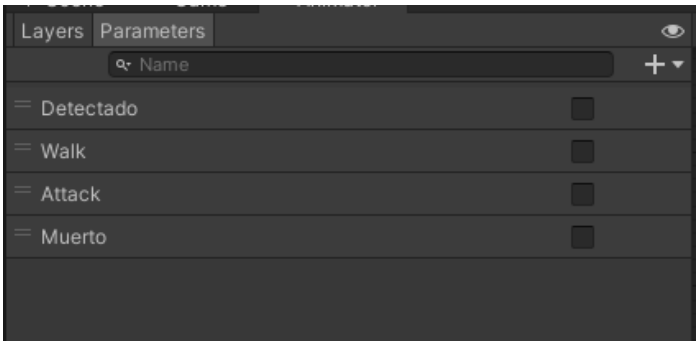


71. Componente Animator de los enemigos básicos.

Este componente a través del controller EnemigoBasico se ocupará a través de una máquina de estados de gestionar los diferentes estados y transiciones en los que puede estar el personaje.



72.Máquina de estados para el controller EnemigoBasico.

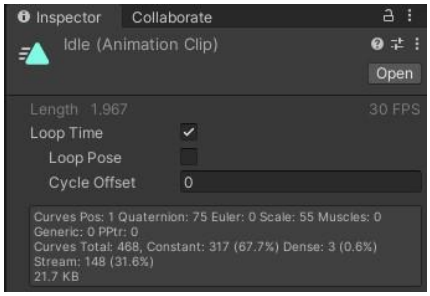


73.Parámetros EnemigoBasico controller.

Estados:

- Idle:

Este estado es el estado inicial de los enemigos básicos, y como vemos en la siguiente imagen, puede pasar a diferentes estados dependiendo del valor de los parámetros que mencionaremos posteriormente.
En este estado la animación ejecutada será Idle:

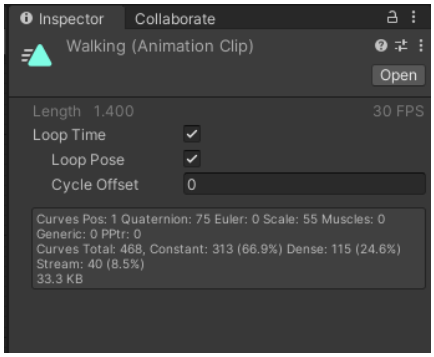


74. Animación Idle para enemigos básicos.

Podemos ver la casilla Loop time marcada, ya que queremos que mientras estemos en este estado la animación se ejecute en bucle.

- Walking:

A este estado podemos llegar desde el estado Idle o desde el estado Running y llegamos a él a través de las transiciones que explicaremos posteriormente. La animación que se ejecutará en este estado es la animación Walking.

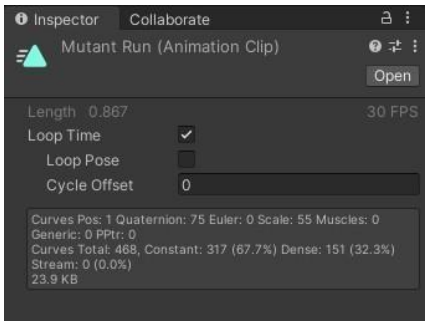


75. Animación Walking para los enemigos básicos.

Podemos ver que está marcada la casilla Loop Time, ya que queremos que mientras estemos en este estado se ejecute todo el rato.

- Running:

A este estado se llega cuando nuestro personaje está en el campo de visión del enemigo básico, y como vemos en la siguiente imagen, puede pasar a diferentes estados dependiendo del valor de los parámetros que mencionaremos posteriormente.
En este estado la animación ejecutada será Mutant Run:

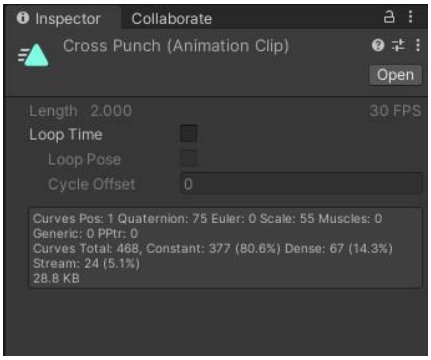


76. Animación Mutant Run para enemigos básicos.

Podemos ver la casilla Loop time marcada, ya que queremos que mientras estemos en este estado la animación se ejecute en bucle.

- Attack:

A este estado podemos llegar desde el estado Idle, desde el estado Running o desde el estado Damage y llegamos a él a través de las transiciones que explicaremos posteriormente. La animación que se ejecutará en este estado es la animación Cross Punch.

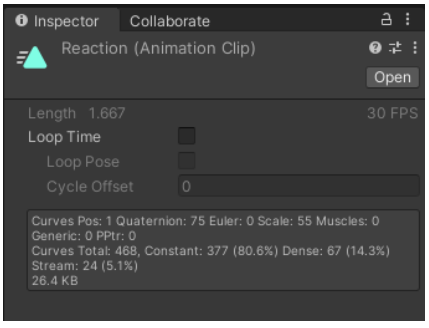


77. Animación Walking para los enemigos básicos.

Podemos ver que está desmarcada la casilla Loop Time, ya que queremos que sólo se ejecute una vez la animación cuando estamos en el estado.

- Damage:

A este estado se llega cuando nuestro personaje golpea a este NPC, y como vemos en la siguiente imagen, puede pasar a diferentes estados dependiendo del valor de los parámetros que mencionaremos posteriormente.
En este estado la animación ejecutada será Reaction:

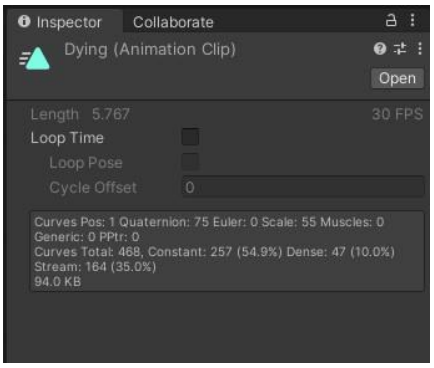


78. Animación Reaction para enemigos básicos.

Podemos ver que está desmarcada la casilla Loop Time, ya que queremos que sólo se ejecute una vez la animación cuando estamos en el estado.

- Dead:

A este estado podemos llegar desde el estado Running o desde el estado Damage, y llegamos a él a través de las transiciones que explicaremos posteriormente. La animación que se ejecutará en este estado es la animación Dying.



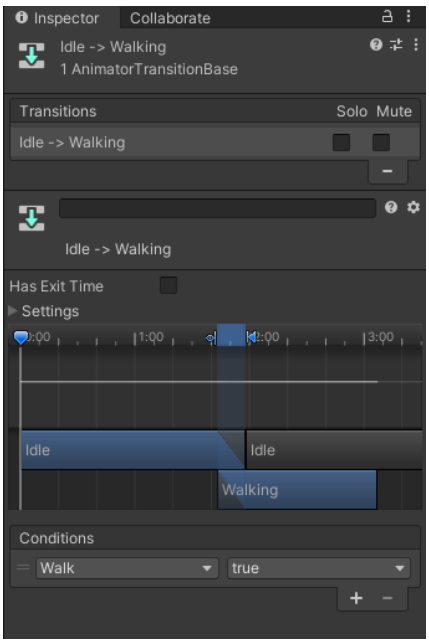
79. Animación Dying para los enemigos básicos.

Podemos ver que está desmarcada la casilla Loop Time, ya que queremos que sólo se ejecute una vez la animación cuando estamos en el estado.

Transiciones:

- Idle -> Walking.

Esta transición depende del parámetro Walk. En el momento que estamos en el estado Idle, si el parámetro Walk es igual a true pasaremos del estado Idle a Walking.

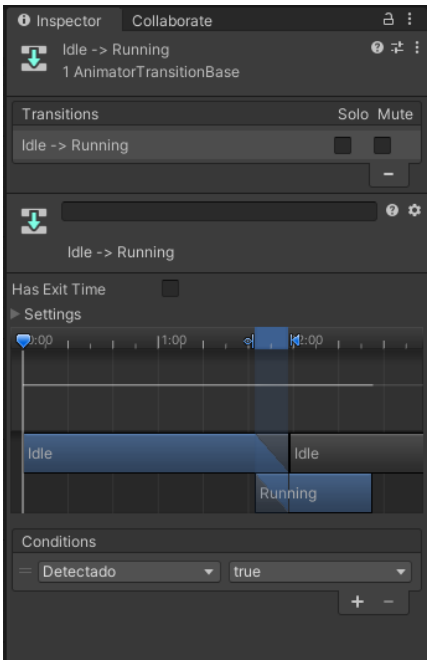


80. Transición Idle-> Walking EnemigoBasico.

En este caso no marcaremos la casilla Has exit time ya que no queremos esperar a que termine la animación Idle para cambiar de estado.

- Idle -> Running.

Esta transición depende del parámetro Detectado. En el momento que estamos en el estado Idle, si el parámetro Detectado es igual a true pasaremos del estado Idle a Running.

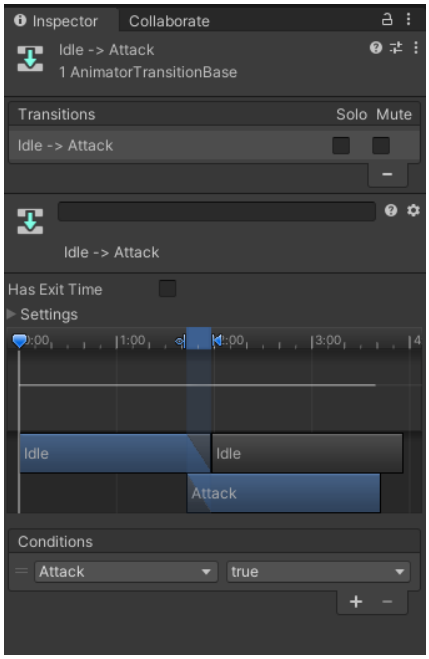


81. Transición Idle-> Running EnemigoBasico.

En este caso no marcaremos la casilla Has exit time ya que no queremos esperar a que termine la animación Idle para cambiar de estado.

- Idle -> Attack.

Esta transición depende del parámetro Attack. En el momento que estamos en el estado Idle, si el parámetro Attack es igual a true pasaremos del estado Idle a Attack.

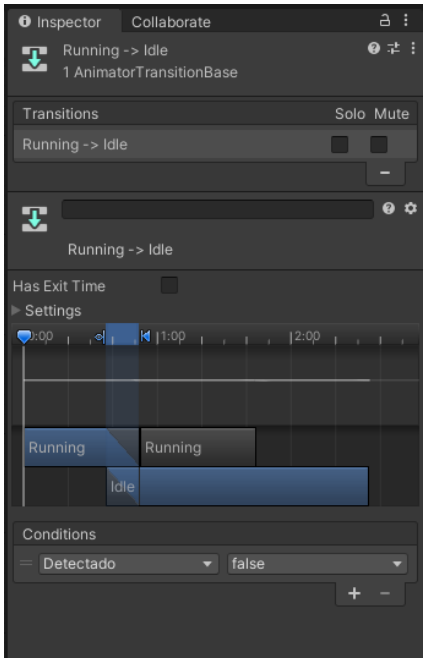


82. Transición Idle-> Attack EnemigoBasico.

En este caso no marcaremos la casilla Has exit time ya que no queremos esperar a que termine la animación Idle para cambiar de estado.

- Running -> Idle.

Esta transición depende del parámetro Detectado. En el momento que estamos en el estado Running, si el parámetro Detectado pasa a false pasaremos del estado Running a Idle.

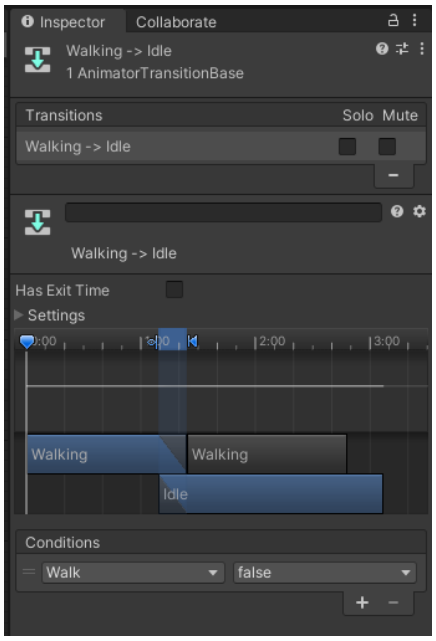


83. Transición Running-> Idle EnemigoBasico.

En este caso no marcaremos la casilla Has exit time ya que no queremos esperar a que termine la animación Running para cambiar de estado.

- Walking -> Idle.

Esta transición depende del parámetro Walk. En el momento que estamos en el estado Walking, si el parámetro Walk pasa a false pasaremos del estado Walking a Idle.

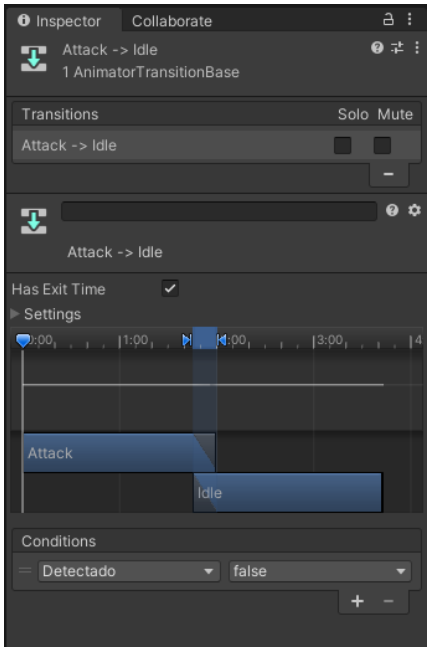


84. Transición Walking-> Idle EnemigoBasico.

En este caso no marcaremos la casilla Has exit time ya que no queremos esperar a que termine la animación Walking para cambiar de estado.

- Attack-> Idle.

Esta transición depende del parámetro Detectado. En el momento que estamos en el estado Attack, si el parámetro Detectado pasa a false pasaremos del estado Attack a Idle.

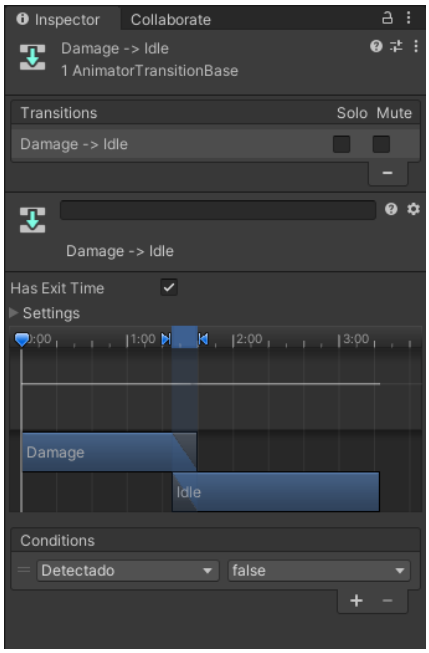


85. Transición Attack-> Idle EnemigoBasico.

En este caso marcaremos la casilla Has exit time ya que queremos esperar a que termine la animación Attack para cambiar de estado.

- Damage-> Idle.

Esta transición depende del parámetro Detectado. En el momento que estamos en el estado Damage, si el parámetro Detectado pasa a false pasaremos del estado Damage a Idle.

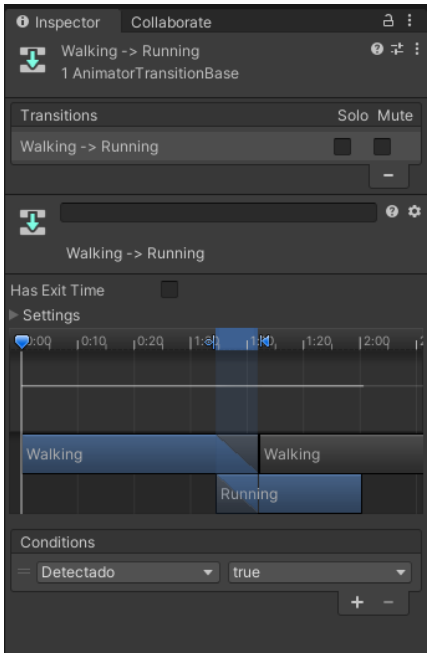


86. Transición Damage-> Idle EnemigoBasico.

En este caso marcaremos la casilla Has exit time ya que queremos esperar a que termine la animación Damage para cambiar de estado.

- Walking-> Running.

Esta transición depende del parámetro Detectado. En el momento que estamos en el estado Walking, si el parámetro Detectado pasa a true pasaremos del estado Walking a Running.

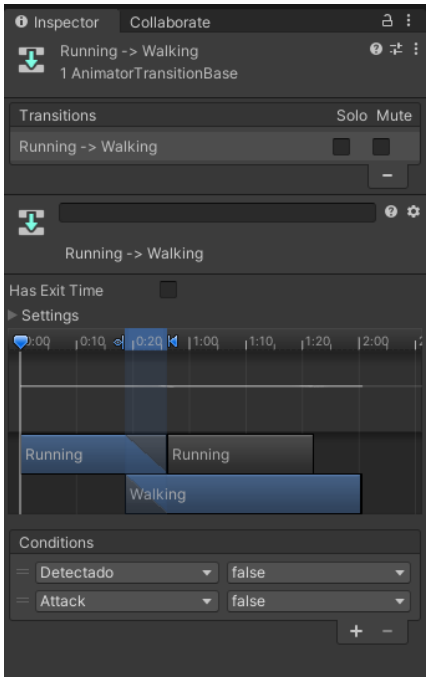


87. Transición Walking-> Running EnemigoBasico.

En este caso no marcaremos la casilla Has exit time ya que no queremos esperar a que termine la animación Walking para cambiar de estado.

- Running-> Walking.

Esta transición depende de los parámetros Detectado y Attack. En el momento que estamos en el estado Running, si el parámetro Detectado pasa a true y Attack es igual a false, pasaremos del estado Running a Walking.

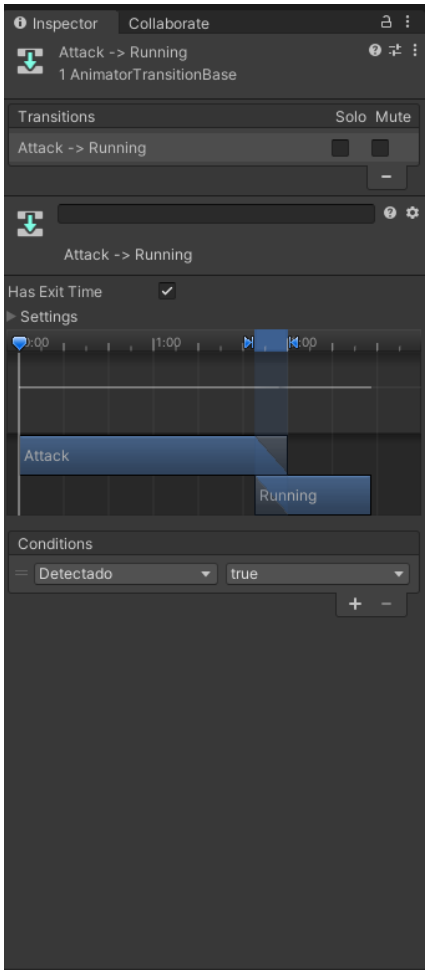


88. Transición Running-> Walking EnemigoBasico.

En este caso no marcaremos la casilla Has exit time ya que no queremos esperar a que termine la animación Running para cambiar de estado.

- Attack-> Running.

Esta transición depende del parámetro Detectado. En el momento que termina el estado Attack, si el parámetro Detectado pasa a true, pasaremos del estado Attack a Running.

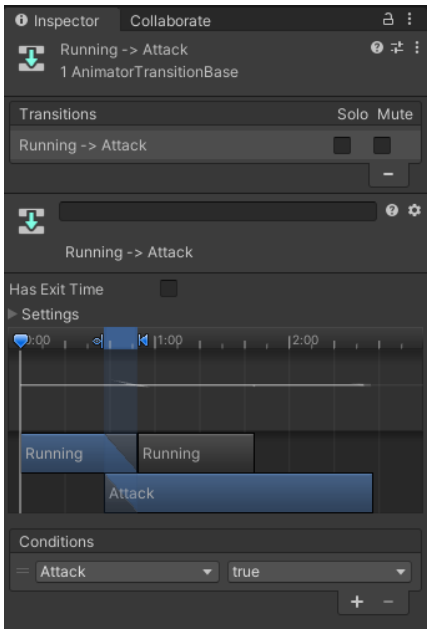


89. Transición Attack-> Running EnemigoBasico.

En este caso no marcaremos la casilla Has exit time ya que no queremos esperar a que termine la animación Attack para cambiar de estado.

- Running-> Attack.

Esta transición depende del parámetro Attack. En el momento que estamos en el estado Running, si el parámetro Attack pasa a true, pasaremos del estado Running a Attack.

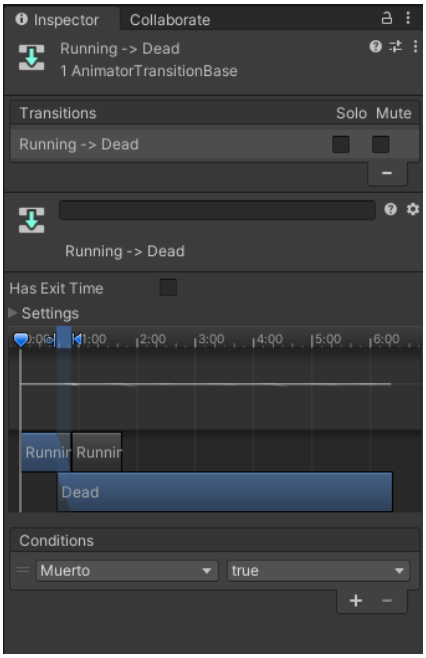


90. Transición Running-> Attack EnemigoBasico.

En este caso no marcaremos la casilla Has exit time ya que no queremos esperar a que termine la animación Running para cambiar de estado.

- Running-> Dead.

Esta transición depende del parámetro Muerto. En el momento que estamos en el estado Running, si el parámetro Muerto pasa a true, pasaremos del estado Running a Dead.

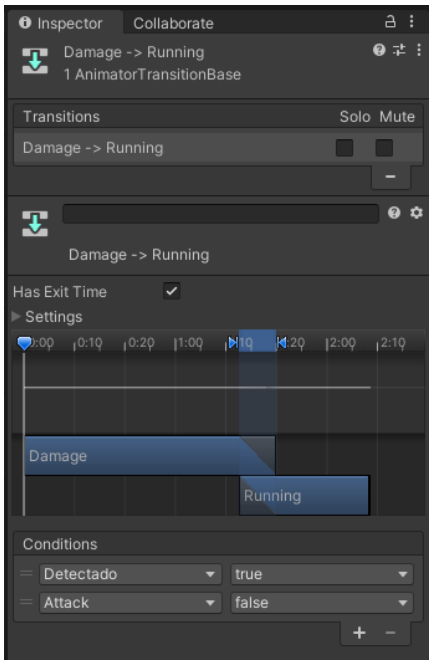


91. Transición Running-> Dead EnemigoBasico.

En este caso no marcaremos la casilla Has exit time ya que no queremos esperar a que termine la animación Running para cambiar de estado.

- Damage-> Running.

Esta transición depende del parámetro Detectado y del parámetro Attack. En el momento que estamos en el estado Damage, si cuando termina la animación damage el parámetro Detectado vale true y Attack false, pasaremos del estado Damage a Running.

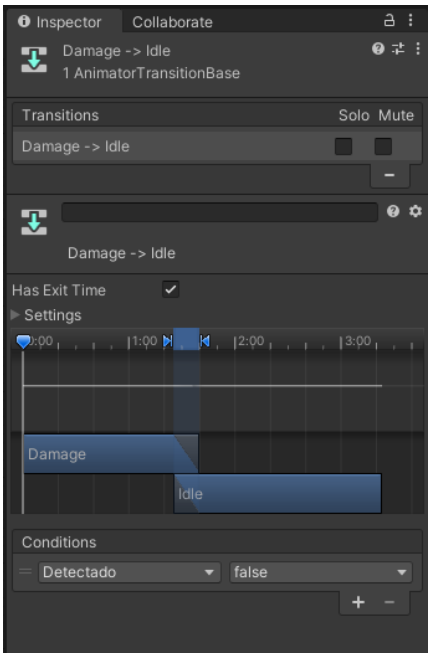


93. Transición Damage-> Idle EnemigoBasico.

En este caso marcaremos la casilla Has exit time ya que queremos esperar a que termine la animación Damage para cambiar de estado.

- Damage-> Attack.

Esta transición depende del parámetro Attack. En el momento que estamos en el estado Damage, si cuando termina la animación damage el parámetro Attack vale true, pasaremos del estado Damage a Attack.

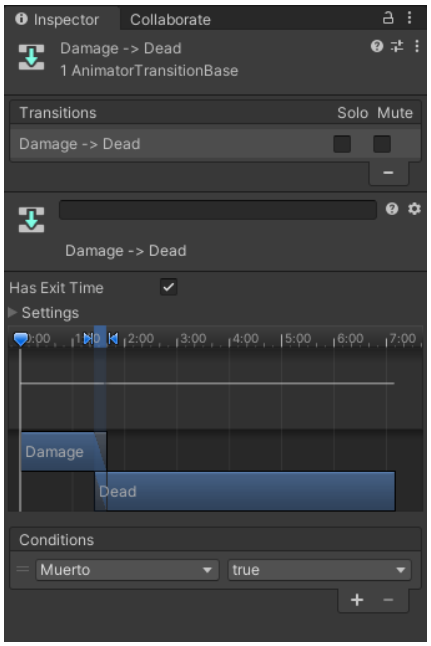


94. Transición Damage-> Attack EnemigoBasico.

En este caso marcaremos la casilla Has exit time ya que queremos esperar a que termine la animación Damage para cambiar de estado.

- Damage-> Dead.

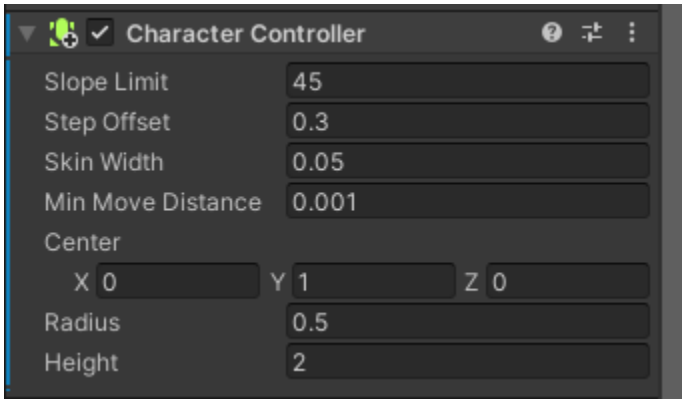
Esta transición depende del parámetro Muerto. En el momento que estamos en el estado Damage, si cuando termina la animación damage el parámetro Muerto vale true, pasaremos del estado Damage a Dead.



95. Transición Damage-> Dead EnemigoBasico.

En este caso marcaremos la casilla Has exit time ya que queremos esperar a que termine la animación Damage para cambiar de estado.

- CharacterController:



96. CharacterController EnemigoBasico.

- NPCPatrulla(Script):

97. Script comportamiento EnemigoBasico.

Este Script se ocupa de implementar todo el comportamiento explicado anteriormente para los parámetros transiciones etc.
El comportamiento de este personaje es estar patrullando por varios puntos del mapa hasta que el jugador entre en su área de detección y en el moment que entra lo que va a hacer es pasar a true el parámetro Detectado activando la transición Walking -> Running o Idle -> Running dependiendo del estado anterior.

```
if(aj != null){
    float distancia = Vector3.Distance(transform.position,aj.transform.position);
    Vector3 rotacion = new Vector3(aj.transform.position.x, transform.position.y, aj.transform.position.z);
    if(distancia < maxDist && distancia > minDist) {
        detectado = true;
        attacking = false;
        ani.SetBool("Walk", false);
    }
    else {
        ani.SetBool("Detectado", false);
        detectado = false;
    }
    if(detectado) {
        transform.LookAt(rotacion);
        ani.SetBool("Detectado", true);
        ani.SetBool("Attack", false);
        if(!attacking){
            characterCont.Move(transform.forward * velocidadMovimiento * Time.deltaTime);
        }
    }
```

```
} else {
    ani.SetBool("Walk", true);
    if(MoveToTarget()) {
        currentPosition++;
        if(currentPosition >= puntosDePatrulla.Length) {
            currentPosition = 0;
        }
    }
}
```

98. Script NPCPatrulla gestión del parámetro detectado dependiendo de si el personaje está en distancia de detección o no.
El método MoveToTarget() lo que hace es ir moviendo el NPC patrullando de un punto a otro establecidos en un array por nosotros.

```
private bool MoveToTarget() {
    bool llegado = false;
    if(movimiento) {
        Vector3 rotacion = new Vector3(puntosDePatrulla[currentPosition].position.x, transform.position.y, puntosDePatrulla[currentPosition].position.z);
        Vector3 distancia = puntosDePatrulla[currentPosition].position - transform.position;
        if(distancia.magnitude < 1f) {
            llegado = true;
            transform.LookAt(rotacion);
            characterCont.Move(transform.forward * 0 * Time.deltaTime);
            ani.SetBool("Walk", false);
            movimiento = false;
            StartCoroutine(tiempoQuieto());
        } else {
            ani.SetBool("Walk", true);
            transform.LookAt(rotacion);
            characterCont.Move(transform.forward * velocidadMovimiento * Time.deltaTime);
            llegado = false;
        }
    }
    return llegado;
}
```

99. Script NPCPatrulla método MoveTo Target.

También se gestiona en este Script el control de cuando ataca el personaje. Esto es cuando entramos en la mínima distancia de detección que es la distancia de ataque, pasamos el parámetro Attack a true activando la transición Running -> Attack.

```
else if(distancia <= minDist) {
    detectado = false;
    transform.LookAt(rotacion);
    ani.SetBool("Detectado", false);
    ani.SetBool("Attack", true);
    attacking = true;
}
```

99. Script NPCPatrulla gestión ataque del NPC.

Los enemigos básicos al morir nos dejan un loot aleatorio que también se genera y gestiona en este script de la siguiente manera:

```
void dejarLoot() {
    int opcion = Random.Range(0,4);
    int indexObjPocion, indexObjArmadura, indexObjHacha, indexObjCarne, indexObjAntiMagia;
    indexObjPocion = db.dataBase.FindIndex(p => p.nombre.Equals("Pocion"));
    indexObjArmadura = db.dataBase.FindIndex(p => p.nombre.Equals("Armadura"));
    indexObjHacha = db.dataBase.FindIndex(p => p.nombre.Equals("Hacha"));
    indexObjCarne = db.dataBase.FindIndex(p => p.nombre.Equals("Carne"));
    indexObjAntiMagia = db.dataBase.FindIndex(p => p.nombre.Equals("Capa anti magia"));
    BaseDatos.ObjetoInventario[] arrayDb = db.dataBase.ToArray();
    switch(opcion) {
        case 0:
            Instantiate(pocion, transform.position + new Vector3(4,0,2), transform.rotation);
            Instantiate(moneda, transform.position, transform.rotation);
            Instantiate(hacha, transform.position + new Vector3(-3,1,2), transform.rotation);
            moneda.GetComponent<ObjetoInv>().cantidad = 1;
            rellenarObjeto(pocion, indexObjPocion);
            rellenarObjeto(hacha, indexObjHacha);
            break;
        case 1:
            Instantiate(pocion, transform.position + new Vector3(2,0,2), transform.rotation);
            Instantiate(moneda, transform.position, transform.rotation);
            moneda.GetComponent<ObjetoInv>().cantidad = 5;
            rellenarObjeto(pocion, indexObjPocion);
            break;
        case 2:
            Instantiate(hacha, transform.position + new Vector3(2,1,2), transform.rotation);
            Instantiate(pocion, transform.position + new Vector3(4,0,2), transform.rotation);
            Instantiate(moneda, transform.position, transform.rotation);
            moneda.GetComponent<ObjetoInv>().cantidad = 3;
            rellenarObjeto(pocion, indexObjPocion);
            rellenarObjeto(hacha, indexObjHacha);
            break;
        case 3:
            Instantiate(moneda, transform.position, transform.rotation);
            Instantiate(pocion, transform.position + new Vector3(4,0,2), transform.rotation);
            moneda.GetComponent<ObjetoInv>().cantidad = 10;
            rellenarObjeto(pocion, indexObjPocion);
            break;
        default:
            break;
    }
}
```

100. Método dejarLoot NPCPatrulla script.

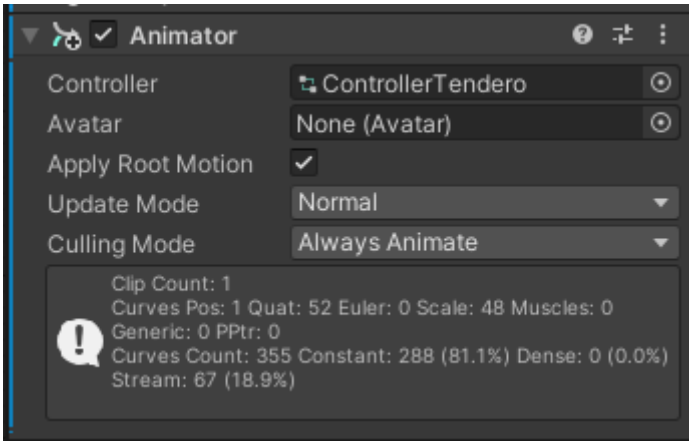
```
private void rellenarObjeto(GameObject objeto, int index) {
    BaseDatos.ObjetoInventario[] arrayDb = db.dataBase.ToArray();
    objeto.GetComponent<ObjetoInv>().cantidad = arrayDb[index].cantidad;
    objeto.GetComponent<ObjetoInv>().id = arrayDb[index].id;
    objeto.GetComponent<ObjetoInv>().precio = arrayDb[index].precio;
    objeto.GetComponent<ObjetoInv>().imagen = arrayDb[index].imagen;
    objeto.GetComponent<ObjetoInv>().acum = arrayDb[index].acum;
    objeto.GetComponent<ObjetoInv>().nombre = arrayDb[index].nombre;
    objeto.GetComponent<ObjetoInv>().descripcion = arrayDb[index].descripcion;
    objeto.GetComponent<ObjetoInv>().type = arrayDb[index].type;
    objeto.GetComponent<ObjetoInv>().parteCuerpo = arrayDb[index].parteCuerpo;
}
```

101. Método rellenarObjeto NPCPatrulla script.

- **Tendero:**

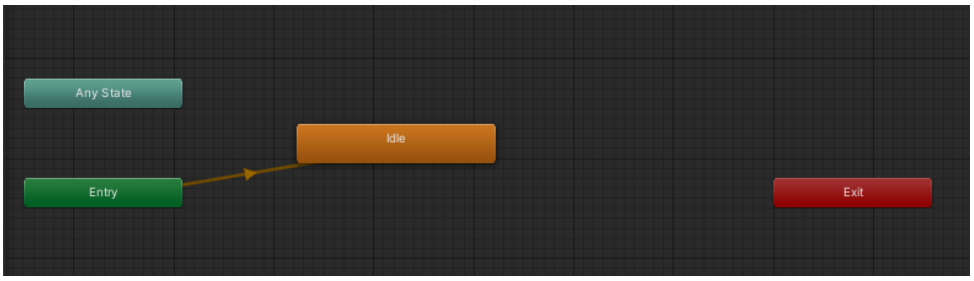
Descripción:
Este personaje es un tendero el cuál se ocupa de vendernos objetos por un precio.

Componentes:
- Animator:



102. Animator Tendero.

Este componente se ocupa a través del controller ControllerTendero de gestionar los diferentes estados en los que puede estar el personaje.
Este controller es una máquina de estados configurada de la siguiente manera:

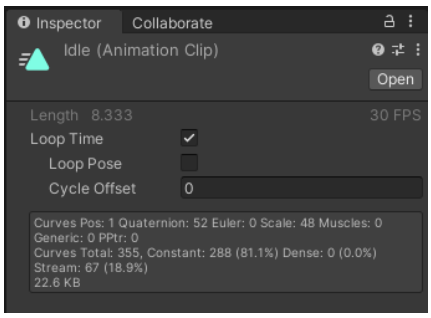


103. Máquina de estados ControllerTendero.

Como se puede ver en la captura en este NPC sólo hay un estado y la IA del personaje no depende de la máquina si no más de un árbol de decisión.

Estados:
- Idle:

El personaje se mantendrá en este estado ejecutando la animación Idle todo el tiempo.

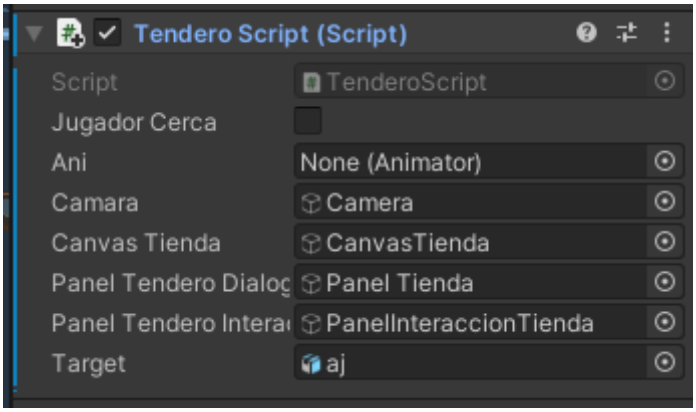


104. Estado Idle del Tendero.

Podemos ver que la casilla Loop Time está marcada debido a que queremos que la animación se ejecute en bucle mientras estemos en este estado.
El árbol de decisión del tendero es el siguiente y explica el comportamiento del NPC. Cuando el personaje se acerca al tendero, se activa el panel con el mensaje “Presione “X” para hablar” y en caso de que se presione la tecla X, se activa otro panel y nos pregunta “¿Desea comprar en la tienda”? En caso de pinchar en el botón Si, se activa la tienda donde podemos comprar y si pinchamos en el botón No, se vuelve a mostrar el mensaje “Presione “X” para hablar” y si el personaje se aleja del tendero, desaparece el mensaje. Esta gestión se realiza desde el componente TenderoScript .

- TenderoScript:

Este componente se ocupa de gestionar todo lo explicado en el apartado anterior como podemos ver en las capturas siguientes:




```
public void Comportamiento_Dialogo() {
    if(Input.GetKeyDown(KeyCode.X) && jugadorCerca) {
        Vector3 posJugador = new Vector3(transform.position.x, target.transform.position.y,transform.position.z);
        target.transform.LookAt(posJugador);
        target.transform.Translate(Vector3.forward * 0 * Time.deltaTime);
        PanelTenderoInteraccion.SetActive(false);
        PanelTenderoDialogo.SetActive(true);
        target.GetComponent<Personaje>().enabled = false;
        camara.GetComponent<SigueAlJugador>().accion = true;
    }
}
```

```
public void NoTienda() {
    PanelTenderoDialogo.SetActive(false);
    PanelTenderoInteraccion.SetActive(true);
    target.GetComponent<Personaje>().enabled = true;
    camara.GetComponent<SigueAlJugador>().accion = false;
}

public void SiTienda() {
    PanelTenderoInteraccion.SetActive(false);
    PanelTenderoDialogo.SetActive(false);
    CanvasTienda.SetActive(true);
    jugadorCerca = false;
}
```

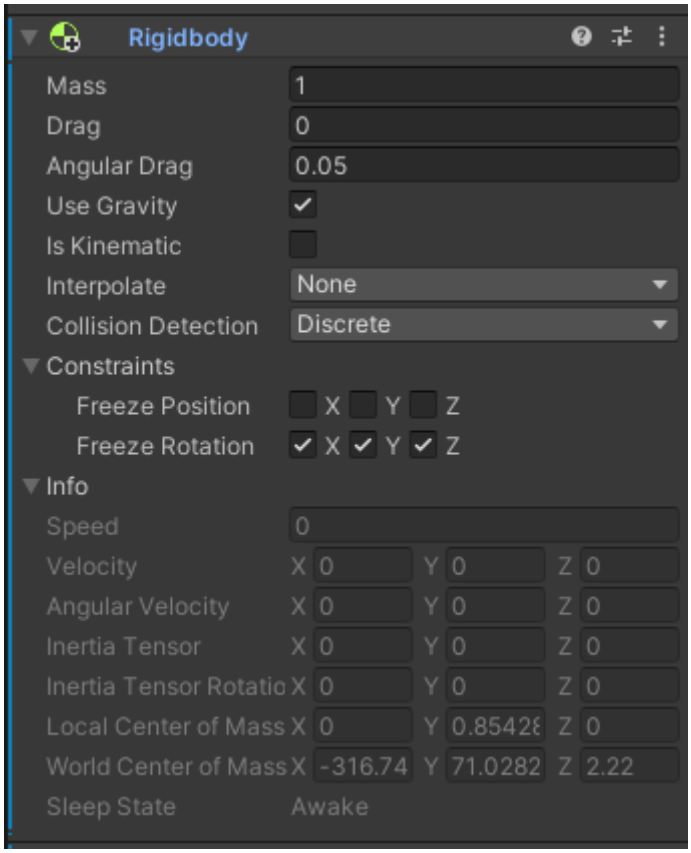
```
private void OnTriggerEnter(Collider other) {

    if(other.tag == "Player"){
        jugadorCerca = true;
        PanelTenderoInteraccion.SetActive(true);
    }
}

private void OnTriggerExit(Collider other) {
    if(other.tag == "Player") {
        jugadorCerca = false;
        PanelTenderoInteraccion.SetActive(false);
        PanelTenderoDialogo.SetActive(false);
    }
}
```

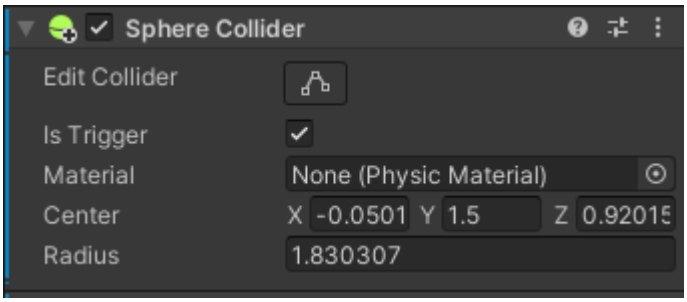
105. Script gestión del comportamiento del tendero.

- Rigidbody:



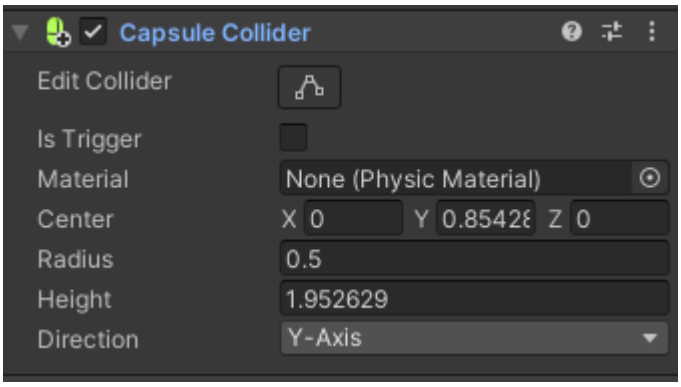
106. Rigidbody tendero.

- Sphere Collider:



107. Sphere Collider tendero.

- Capsule Collider:



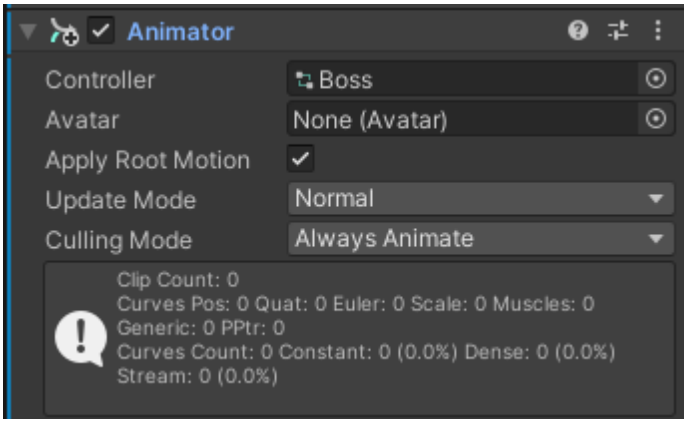
108. Capsule Collider tendero.

- **Maw Jaygo:**

Descripción:
Este personaje es el boss final del juego y sólo aparece en el momento en el que ya hemos cumplido la misión de matar a su secuaz Warrok en la ciudad y llevado este botín al amigo del tabernero y pagado a estas 100 monedas para teletransportarse hasta la ubicación de este. El comportamiento de este personaje será estar quieto esperando a que el personaje del jugador esté a una distancia menor que la distancia de detección que establezcamos y cuando esto ocurre su comportamiento se divide en dos fases:
La fase 1 en la que tiene una serie de ataques dependiendo de la distancia al jugador y de un factor de aleatoriedad y una vez que la vida de este personaje baja de un umbral, se pasa a la fase 2 en la que los tipos de ataque cambian. Una vez muerto este personaje se terminará el juego.

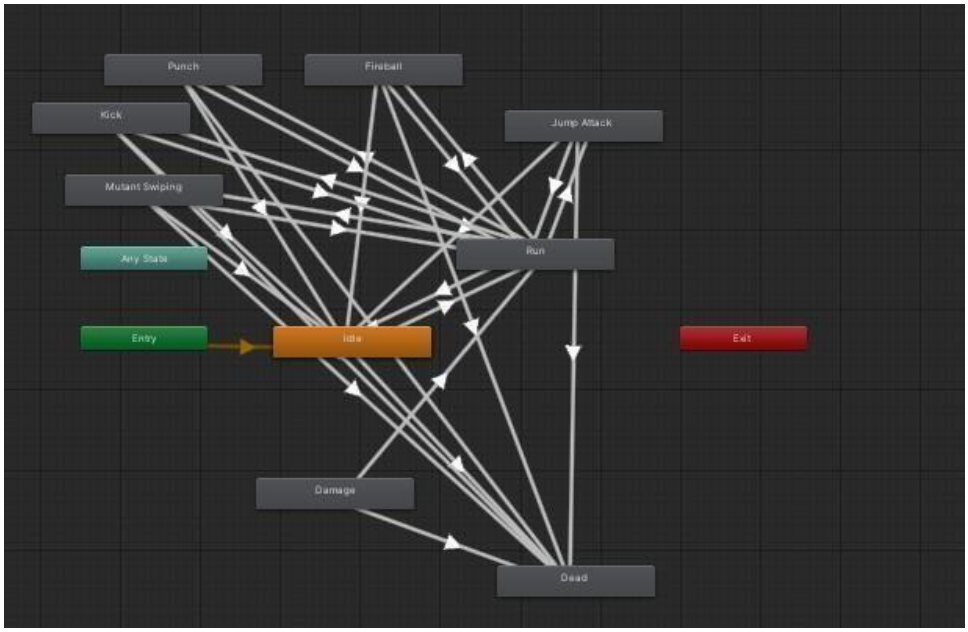
Componentes:

- Animator:



109. Animator del personaje Maw Jaygo.

El comportamiento de este personaje se gestiona mediante una máquina de estados dentro del controller Boss y el script Boss.cs, la máquina de estados del controller Boss es la siguiente:

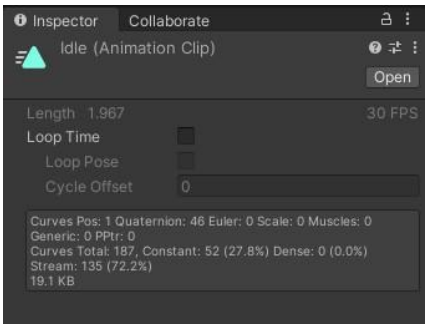


110. Máquina de estados del Boss.

Estados:

- Idle:

Este estado es el estado inicial de los enemigos básicos, y como vemos en la siguiente imagen, puede pasar a diferentes estados dependiendo del valor de los parámetros que mencionaremos posteriormente.
En este estado la animación ejecutada será Idle:

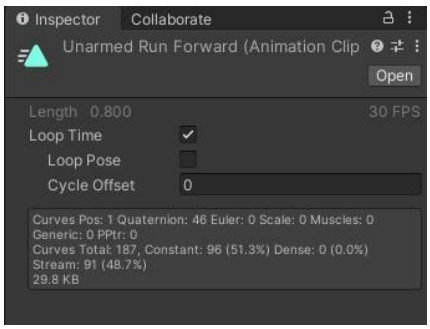


111. Animación Idle para Maw Jaygo.

Podemos ver la casilla Loop time marcada, ya que queremos que mientras estemos en este estado la animación se ejecute en bucle.

- Run:

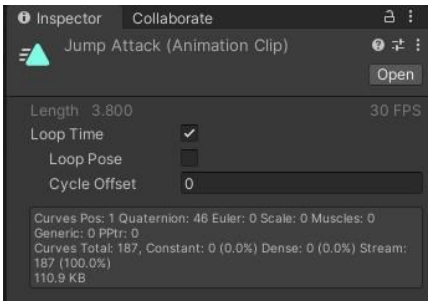
En este estado, la animación que se ejecutará será la animación Unarmed Run Forward al igual que comentamos en el estado idle, marcaremos la casilla loop time para que se ejecute en bucle hasta que salgamos del estado.



112. Captura de la animación Unarmed Run Forward del personaje Maw Jaygo.

- Jump Attack:

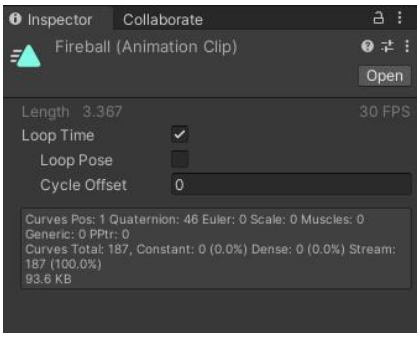
En el estado Jump Attack, se ejecuta la animación Jump Attack dejando marcada la opción Loop Time para que se ejecute en bucle mientras estemos en este estado.



113. Animación Jump Attack del personaje Maw Jaygo.

- Fireball:

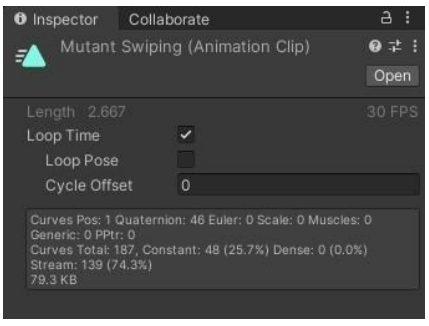
En el estado Fireball, se ejecuta la animación Fireball dejando marcada la opción Loop Time para que se ejecute en bucle mientras estemos en este estado.



114. Animación Fireball del personaje Maw Jaygo.

- Mutant Swiping:

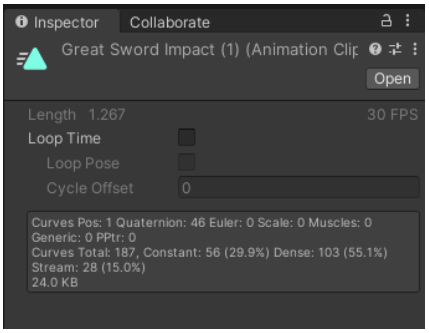
En el estado Mutant Swiping, se ejecuta la animación Mutant Swiping dejando marcada la opción Loop Time para que se ejecute en bucle mientras estemos en este estado.



115. Animación Mutant Swiping del personaje Maw Jaygo.

- Damage:

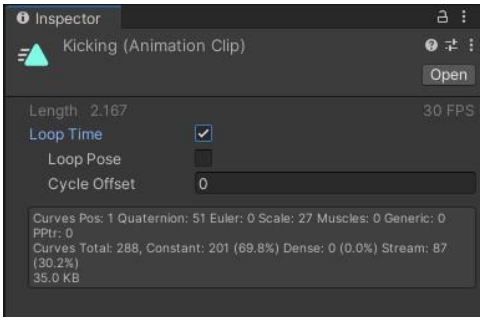
En el estado Damage, se ejecuta la animación Great Sword Impact no se marca la opción Loop Time para que se ejecute sólo una vez.



116. Animación Great Sword Impact del personaje Maw Jaygo.

- Kick:

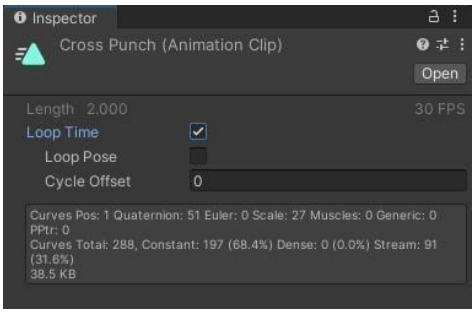
En el estado Kick, se ejecuta la animación Kicking dejando marcada la opción Loop Time para que se ejecute en bucle mientras estemos en este estado.



117. Animación kicking del personaje Maw Jaygo.

- Punch:

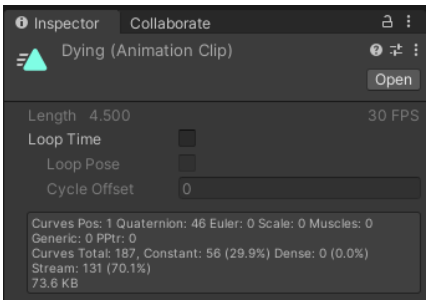
En el estado Punch, se ejecuta la animación CrossPunch dejando marcada la opción Loop Time para que se ejecute en bucle mientras estemos en este estado.



118. Animación Cross punch del personaje Maw Jaygo.

- Dead:

En el estado Dead, se ejecuta la animación Dying y no se marca la opción Loop Time para que se ejecute sólo una vez.

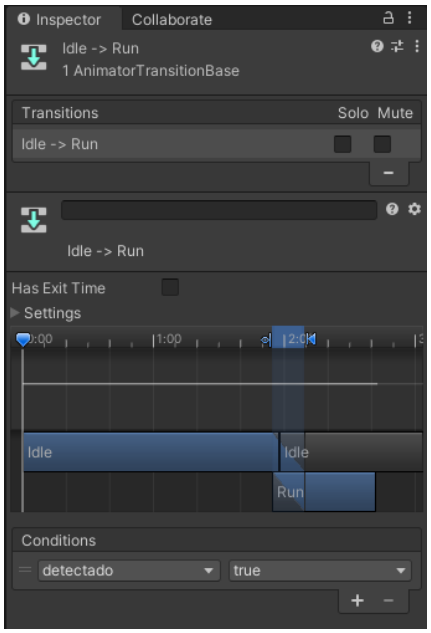


119. Animación Dying del personaje Maw Jaygo.

Transiciones:

- Idle -> Run.

Esta transición depende del parámetro detectado. En el momento que estamos en el estado Idle, si el parámetro detectado es igual a true pasaremos del estado Idle a Run.

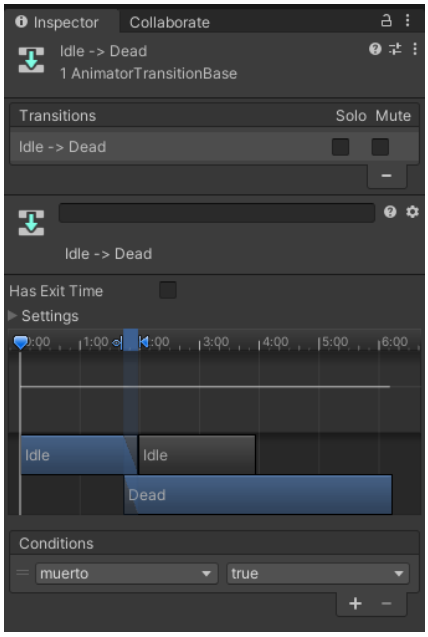


120. Transición Idle-> Run Maw Jaygo.

En este caso no marcaremos la casilla Has exit time ya que no queremos esperar a que termine la animación Idle para cambiar de estado.

- Idle -> Dead.

Esta transición depende del parámetro muerto. En el momento que estamos en el estado Idle, si el parámetro muerto es igual a true pasaremos del estado Idle a Dead.

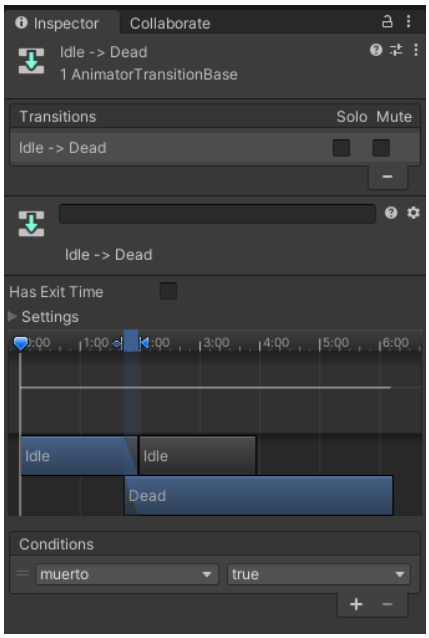


121. Transición Idle-> Dead Maw Jaygo.

En este caso no marcaremos la casilla Has exit time ya que no queremos esperar a que termine la animación Idle para cambiar de estado.

- Run -> Jump Attack.

Esta transición depende del parámetro JumpAttack. En el momento que estamos en el estado Run, si el parámetro JumpAttack es igual a true pasaremos del estado Run a JumpAttack.

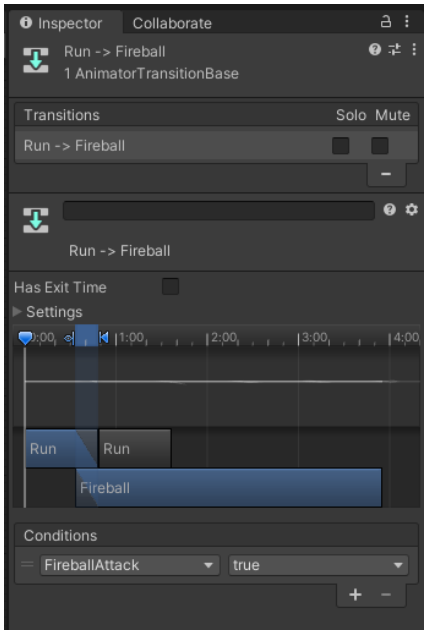


122. Transición Run-> Jump Attack Maw Jaygo.

En este caso no marcaremos la casilla Has exit time ya que no queremos esperar a que termine la animación Run para cambiar de estado.

- Run -> Fireball.

Esta transición depende del parámetro FireballAttack. En el momento que estamos en el estado Run, si el parámetro FireballAttack es igual a true pasaremos del estado Run a FireballAttack.

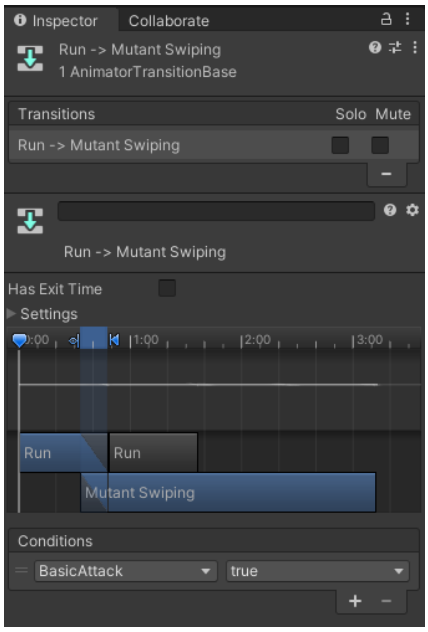


123. Transición Run-> Fireball Maw Jaygo.

En este caso no marcaremos la casilla Has exit time ya que no queremos esperar a que termine la animación Run para cambiar de estado.

- Run -> Mutant Swiping.

Esta transición depende del parámetro BasicAttack. En el momento que estamos en el estado Run, si el parámetro BasicAttack es igual a true pasaremos del estado Run a BasicAttack.

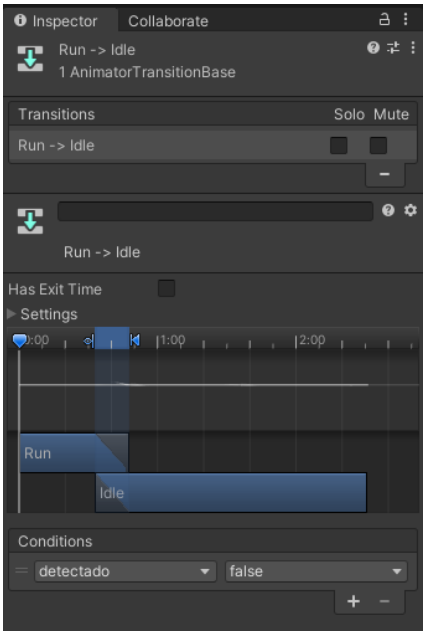


124. Transición Run-> Fireball Maw Jaygo.

En este caso no marcaremos la casilla Has exit time ya que no queremos esperar a que termine la animación Run para cambiar de estado.

- Run -> Idle.

Esta transición depende del parámetro detectado. En el momento que estamos en el estado Run, si el parámetro detectado pasa a false pasaremos del estado Run a Idle.

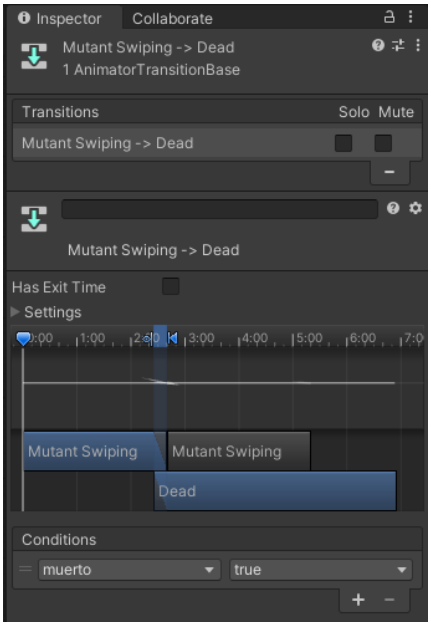


125. Transición Run-> Idle Maw Jaygo.

En este caso no marcaremos la casilla Has exit time ya que no queremos esperar a que termine la animación Run para cambiar de estado.

- Fireball -> Dead.

Esta transición depende del parámetro muerto. En el momento que estamos en el estado Fireball, si el parámetro muerto pasa a true pasaremos del estado Fireball a Dead.

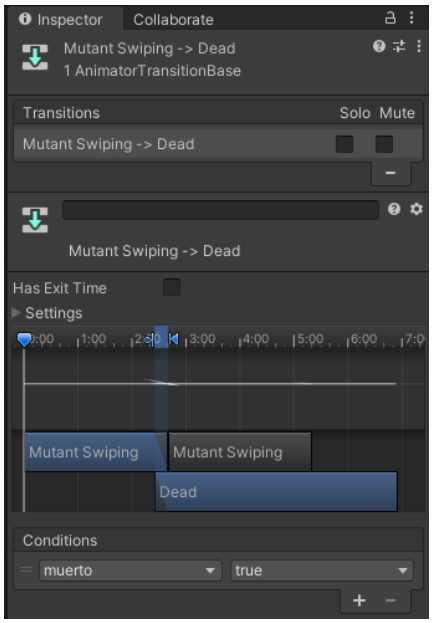


126. Transición Fireball -> Dead Maw Jaygo.

En este caso no marcaremos la casilla Has exit time ya que no queremos esperar a que termine la animación Fireball para cambiar de estado.

- Mutant Swiping -> Dead.

Esta transición depende del parámetro muerto. En el momento que estamos en el estado Mutant Swiping, si el parámetro muerto pasa a true pasaremos del estado Mutant Swiping a Dead.

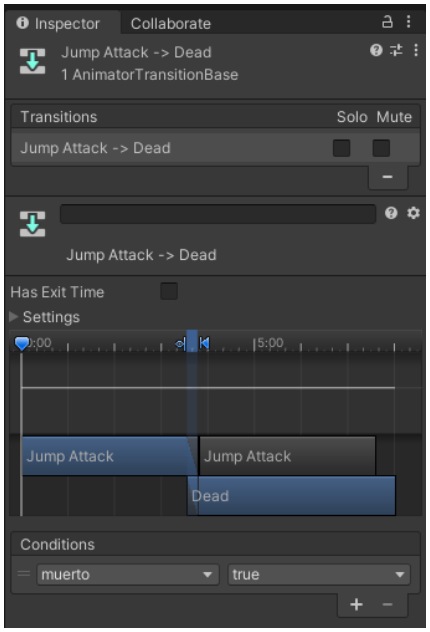


127. Transición Mutant Swiping-> Dead Maw Jaygo.

En este caso no marcaremos la casilla Has exit time ya que no queremos esperar a que termine la animación Mutant Swiping para cambiar de estado.

- Jump Attack -> Dead.

Esta transición depende del parámetro muerto. En el momento que estamos en el estado Jump Attack, si el parámetro muerto pasa a true pasaremos del estado Jump Attack a Dead.

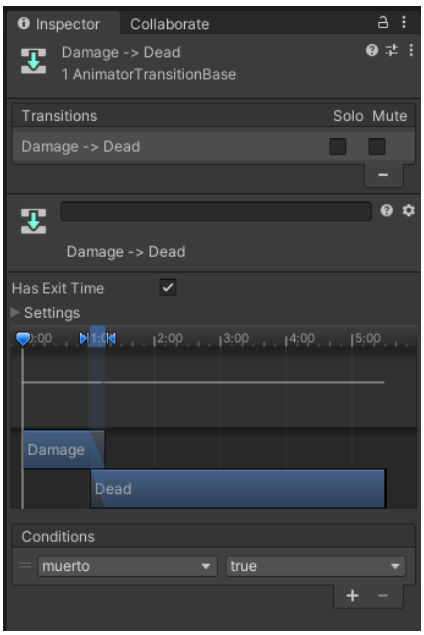


128. Transición Jump Attack -> Dead Maw Jaygo.

En este caso no marcaremos la casilla Has exit time ya que no queremos esperar a que termine la animación Jump Attack para cambiar de estado.

- Damage -> Dead.

Esta transición depende del parámetro muerto. En el momento que termina la animación del estado Damage, si el parámetro muerto pasa a true pasaremos del estado Damage a Dead.

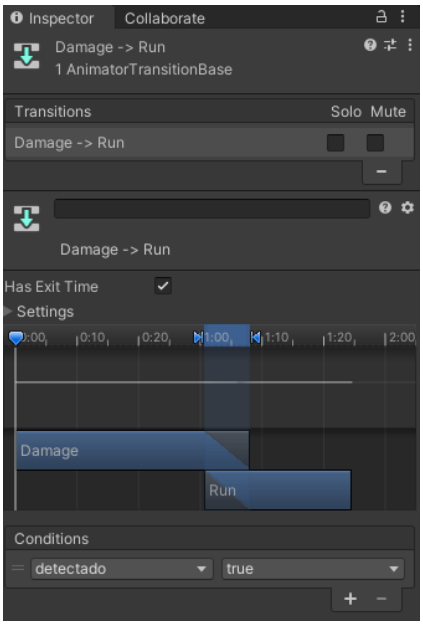


129. Transición Damage -> Dead Maw Jaygo.

En este caso marcaremos la casilla Has exit time ya que queremos esperar a que termine la animación Damage para cambiar de estado.

- Damage -> Run.

Esta transición depende del parámetro detectado. En el momento que termina la animación del estado Damage, si el parámetro detectado es igual a true pasaremos del estado Damage a Run.

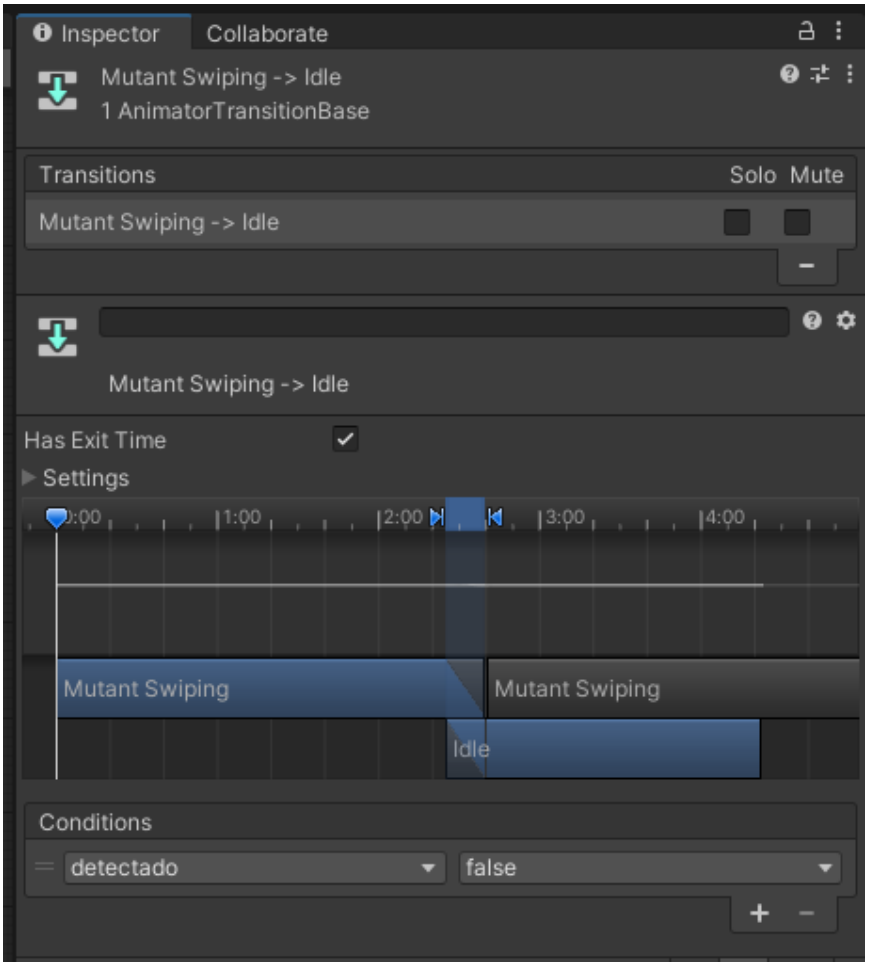


130. Transición Damage -> Run Maw Jaygo.

En este caso marcaremos la casilla Has exit time ya que queremos esperar a que termine la animación Damage para cambiar de estado.

- Mutant Swiping -> Idle.

Esta transición depende del parámetro detectado. En el momento que termina la animación del estado Mutant Swiping, si el parámetro detectado es igual a false pasaremos del estado Mutant Swiping a Idle.

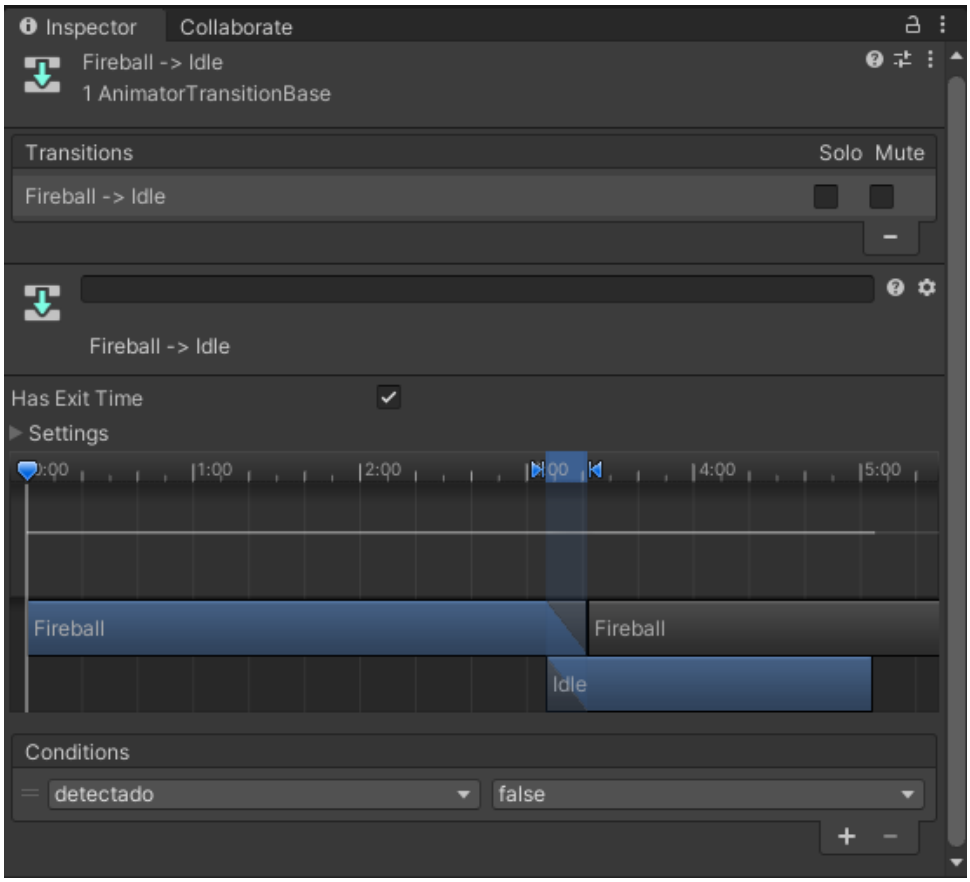


131. Transición Mutant Swiping -> Idle Maw Jaygo.

En este caso marcaremos la casilla Has exit time ya que queremos esperar a que termine la animación Mutant Swiping para cambiar de estado.

- Fireball -> Idle.

Esta transición depende del parámetro detectado. En el momento que termina la animación del estado Fireball, si el parámetro detectado es igual a false pasaremos del estado Fireball a Idle.

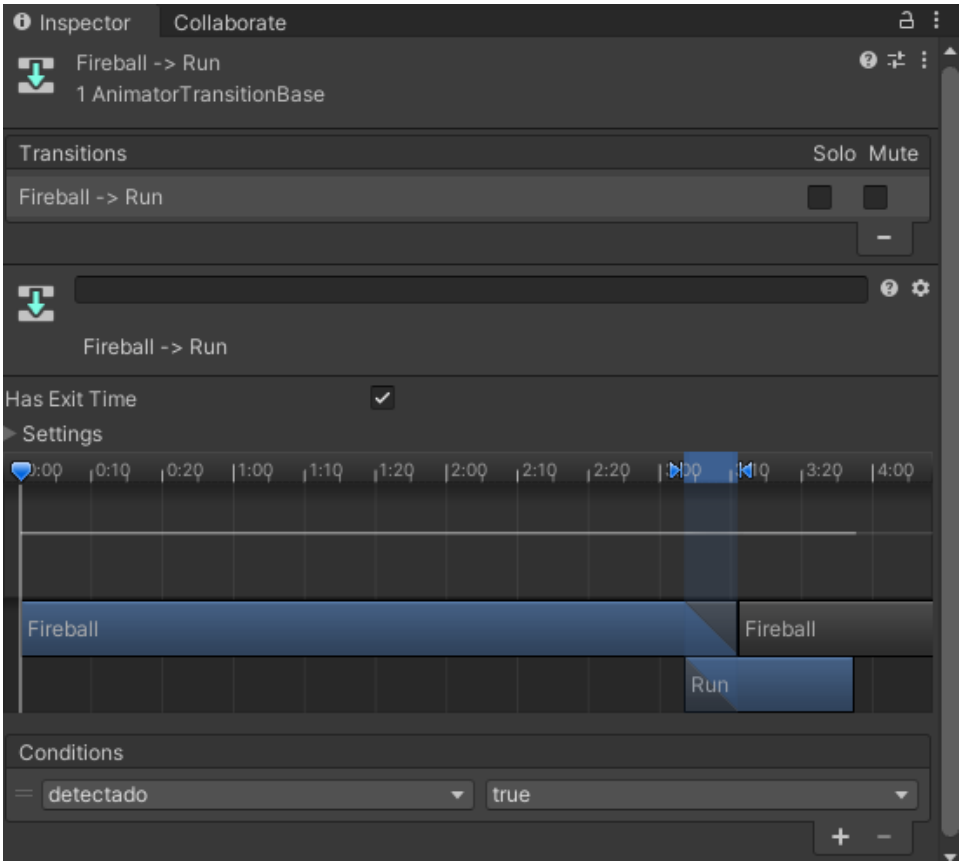


132. Transición Fireball -> Idle Maw Jaygo.

En este caso marcaremos la casilla Has exit time ya que queremos esperar a que termine la animación Fireball para cambiar de estado.

- Fireball -> Run.

Esta transición depende del parámetro detectado. En el momento que termina la animación del estado Fireball, si el parámetro detectado es igual a true pasaremos del estado Fireball a Run.

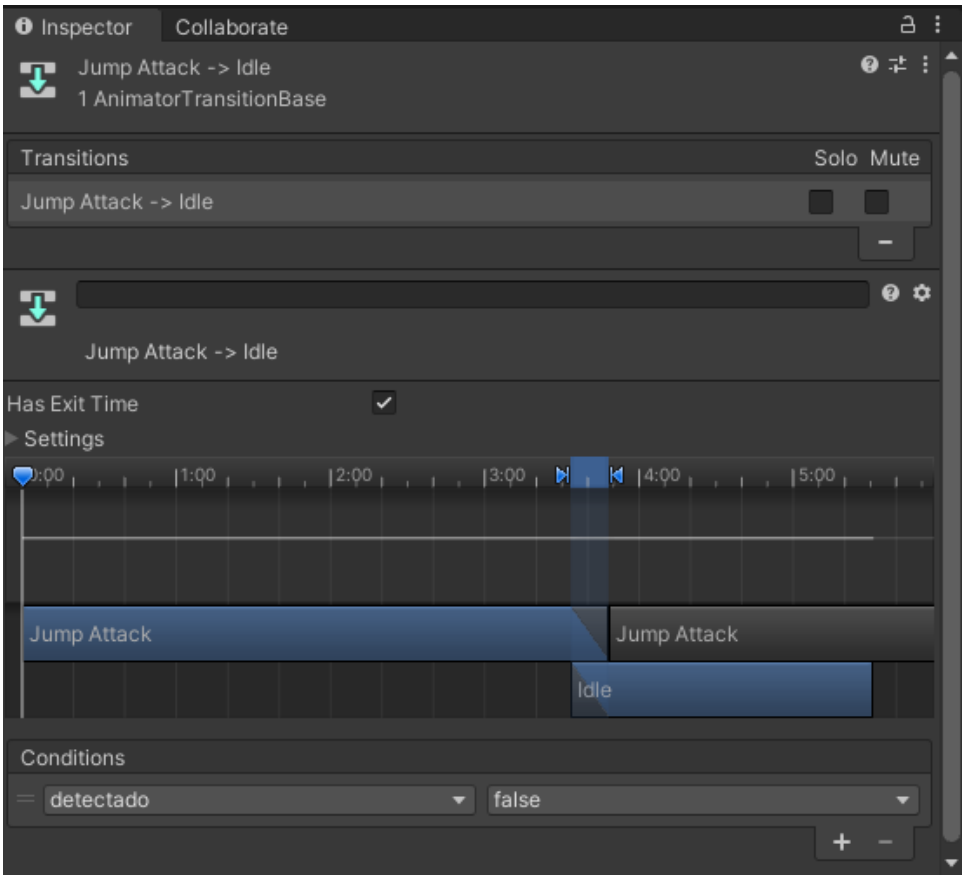


133. Transición Fireball -> Run Maw Jaygo.

En este caso marcaremos la casilla Has exit time ya que queremos esperar a que termine la animación Fireball para cambiar de estado.

- Jump Attack-> Idle.

Esta transición depende del parámetro detectado. En el momento que termina la animación del estado Jump Attack, si el parámetro detectado es igual a false pasaremos del estado Jump Attack a Idle.

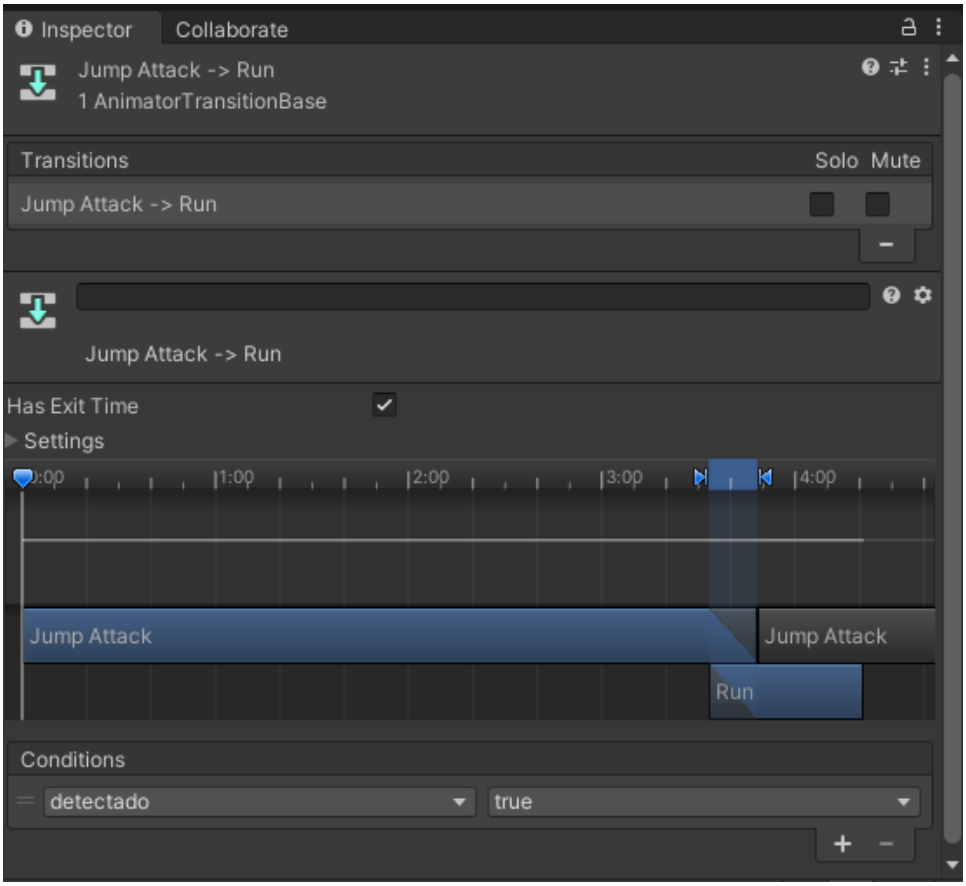


134. Transición Jump Attack -> Idle Maw Jaygo.

En este caso marcaremos la casilla Has exit time ya que queremos esperar a que termine la animación Jump Attack para cambiar de estado.

- Jump Attack-> Run.

Esta transición depende del parámetro detectado. En el momento que termina la animación del estado Jump Attack, si el parámetro detectado es igual a true pasaremos del estado Jump Attack a Run.

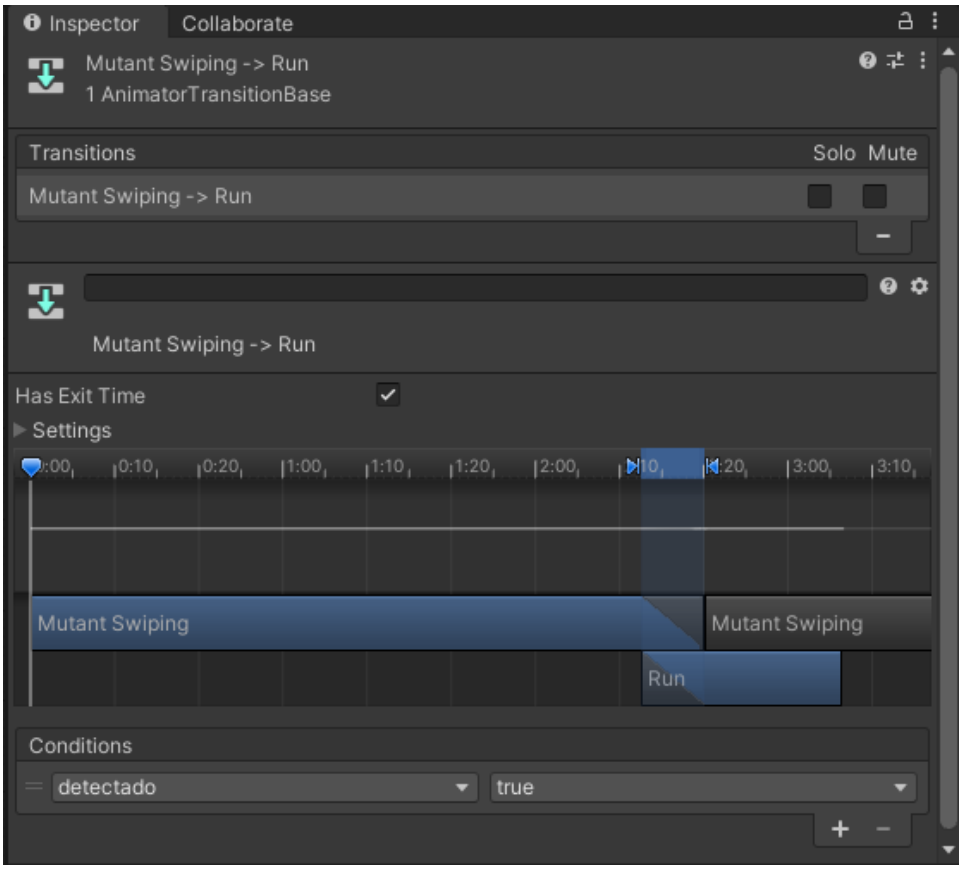


135. Transición Jump Attack -> Run Maw Jaygo.

En este caso marcaremos la casilla Has exit time ya que queremos esperar a que termine la animación Jump Attack para cambiar de estado.

- Mutant Swiping -> Run.

Esta transición depende del parámetro detectado. En el momento que termina la animación del estado Mutant Swiping, si el parámetro detectado es igual a true pasaremos del estado Mutant Swiping a Run.

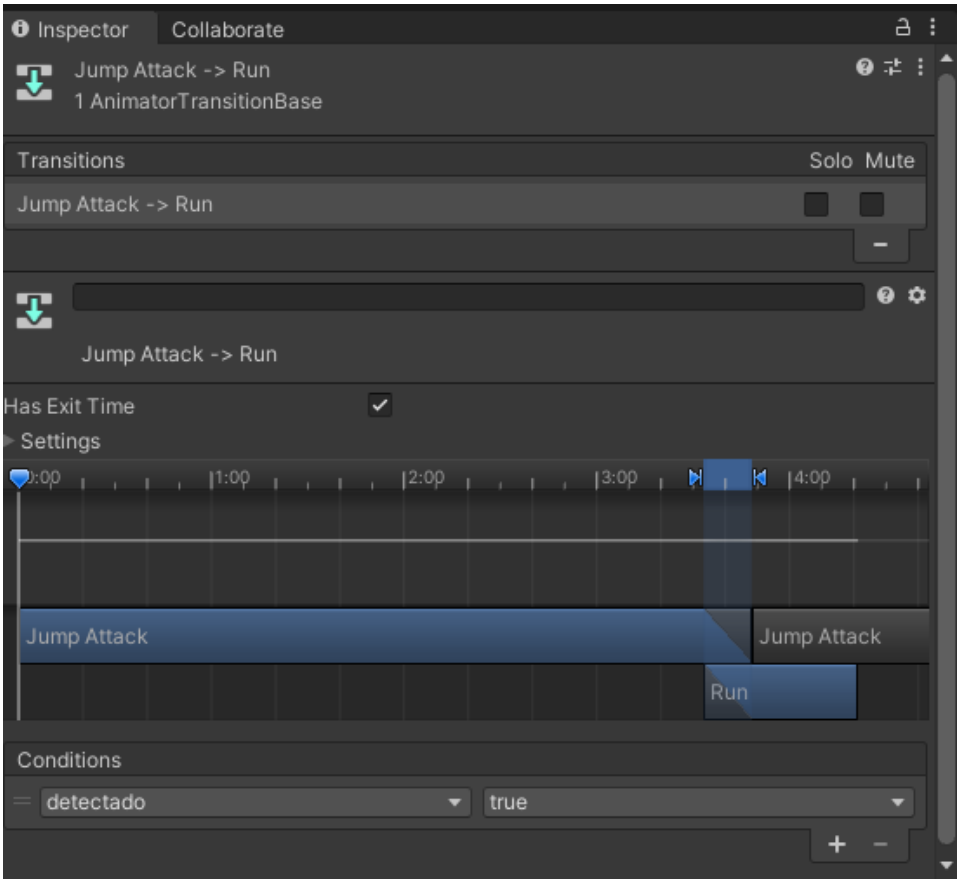


136. Transición Mutant Swiping -> Run Maw Jaygo.

En este caso marcaremos la casilla Has exit time ya que queremos esperar a que termine la animación Mutant Swiping para cambiar de estado.

- Kick -> Run.

Esta transición depende del parámetro detectado. En el momento que termina la animación del estado Kick, si el parámetro detectado es igual a true pasaremos del estado Kick a Run.

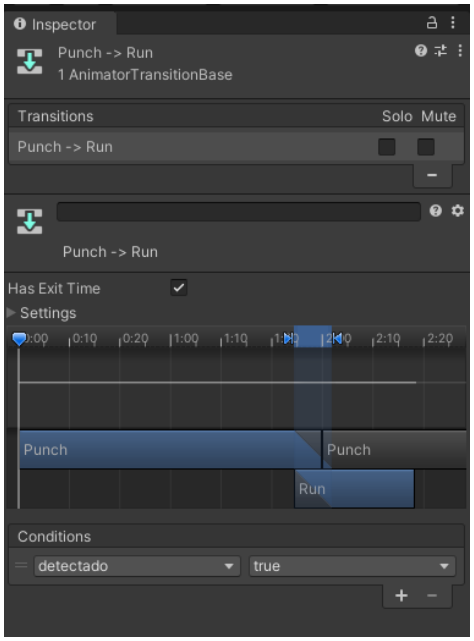


137. Transición Kick -> Run Maw Jaygo.

En este caso marcaremos la casilla Has exit time ya que queremos esperar a que termine la animación Kicking para cambiar de estado.

- Punch -> Run.

Esta transición depende del parámetro detectado. En el momento que termina la animación del estado Punch, si el parámetro detectado es igual a true pasaremos del estado Punch a Run.

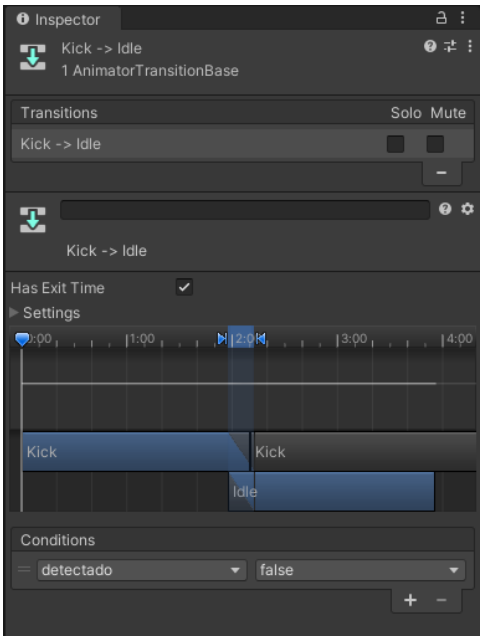


138. Transición Punch-> Run Maw Jaygo.

En este caso marcaremos la casilla Has exit time ya que queremos esperar a que termine la animación Cross Punch para cambiar de estado.

- Kick-> Idle.

Esta transición depende del parámetro detectado. En el momento que termina la animación del estado Kicking, si el parámetro detectado es igual a false pasaremos del estado Kicking a Idle.

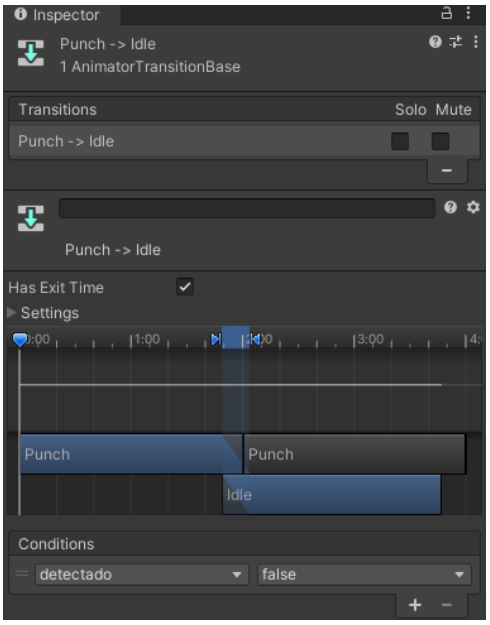


139. Transición Kick -> Idle Maw Jaygo.

En este caso marcaremos la casilla Has exit time ya que queremos esperar a que termine la animación Kicking para cambiar de estado.

- Punch-> Idle.

Esta transición depende del parámetro detectado. En el momento que termina la animación del estado Punch, si el parámetro detectado es igual a false pasaremos del estado Punch a Idle.

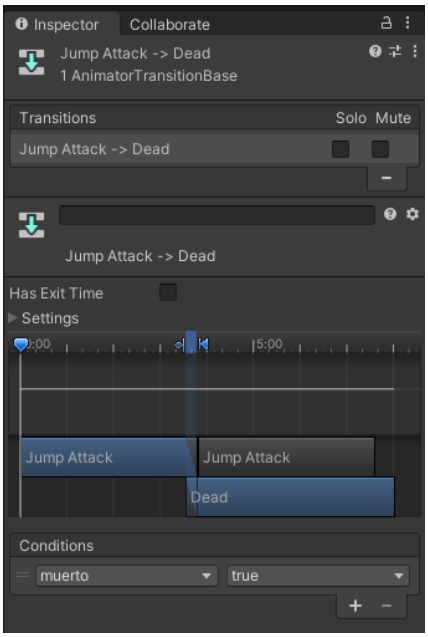


140. Transición Punch-> Idle Maw Jaygo.

En este caso marcaremos la casilla Has exit time ya que queremos esperar a que termine la animación Cross Punch para cambiar de estado.

- Punch -> Dead.

Esta transición depende del parámetro muerto. En el momento que estamos en el estado Pucnh, si el parámetro muerto pasa a true pasaremos del estado Punch a Dead.

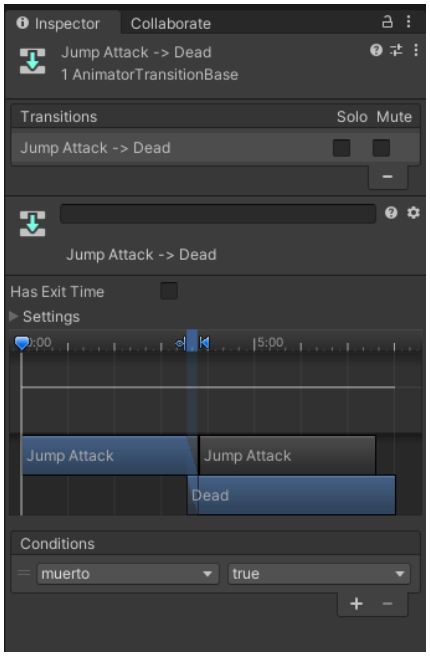


141. Transición Punch -> Dead Maw Jaygo.

En este caso no marcaremos la casilla Has exit time ya que no queremos esperar a que termine la animación Cross Punch para cambiar de estado.

- Kick -> Dead.

Esta transición depende del parámetro muerto. En el momento que estamos en el estado Kick, si el parámetro muerto pasa a true pasaremos del estado Kick a Dead.

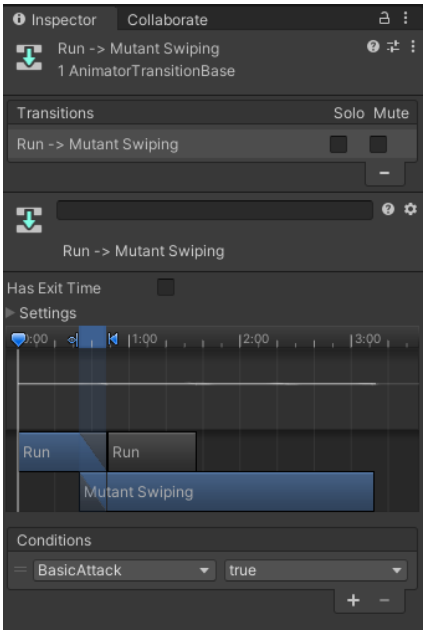


142. Transición Kick -> Dead Maw Jaygo.

En este caso no marcaremos la casilla Has exit time ya que no queremos esperar a que termine la animación Kicking para cambiar de estado.

- Run -> Punch.

Esta transición depende del parámetro BasicAttack2. En el momento que estamos en el estado Run, si el parámetro BasicAttack2 es igual a true pasaremos del estado Run a BasicAttack2.

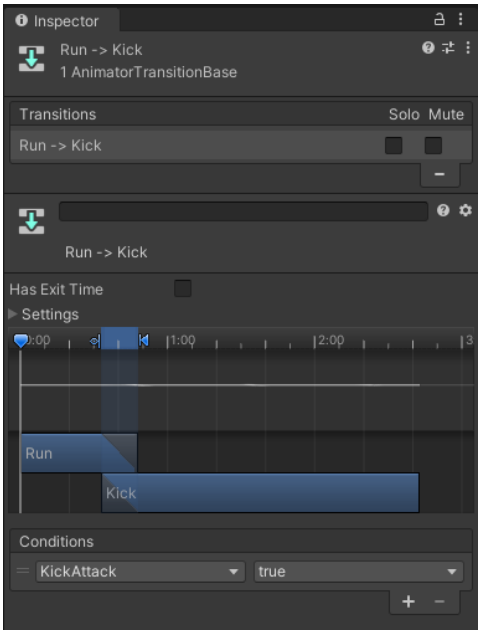


143. Transición Run-> Punch Maw Jaygo.

En este caso no marcaremos la casilla Has exit time ya que no queremos esperar a que termine la animación Run para cambiar de estado.

- Run -> Kick.

Esta transición depende del parámetro KickAttack. En el momento que estamos en el estado Run, si el parámetro KickAttack es igual a true pasaremos del estado Run a KickAttack.



144. Transición Run-> Kick Maw Jaygo.

En este caso no marcaremos la casilla Has exit time ya que no queremos esperar a que termine la animación Run para cambiar de estado.

- Boss (Script):

Este script se ocupa de dotar de IA al boss, modificando los valores de los parámetros del controller detallado anteriormente entre otras cosas.
En primer lugar, lo que hará este script cada vez que se ejecute el update es comprobar si está vivo o no, y en caso afirmativo cambiar el parámetro muerto a true para activar alguna de las transiciones descrita anteriormente para ejecutar la animación de muerte dependiendo del estado en el que se encontrara anteriormente, así como destruir los componentes del personaje y los componentes de la barra de vida etc.

```
public void Comportamiento_Enemigo() {
    if(minHp <= 0) {
        minHp = 0;
        detectado = false;
        attacking = false;
        ani.SetBool("detectado", false);
        ani.SetBool("JumpAttack", false);
        ani.SetBool("BasicAttack", false);
        ani.SetBool("FireballAttack", false);
        ani.SetBool("KickAttack", false);
        ani.SetBool("BasicAttack2", false);
        ani.SetBool("muerto", true);
        dejarLoot();
        Destroy(this.gameObject.GetComponent<Boss>());
        Destroy(this.gameObject.GetComponent<CharacterController>());
        Destroy(barraVida);
        Destroy(fondoBarra);
        txtNombre.SetActive(false);
        Destroy(this.gameObject, 10);
    }
```

145. Script boss comprobación de si el boss está vivo o muerto.

En caso de no estar muerto, generará un número aleatorio que decidirá el ataque que realizará, aunque luego dependerá de factores de distancia y fase. En caso de que el número generado sea 0, el ataque que se ejecutará en caso de estar en distancia de ataque es el ataque en salto(Jump Attack), por lo que pasaremos el valor del parámetro JumpAttack a true y el resto de los parámetros los pasaremos a false.

```
else {
    rutina = Random.Range(0,4);
```

```
switch (rutina)
{
    case 0:
        if(!attacking && !lejos){
            attacking = true;
            ani.SetBool("JumpAttack", true);
            ani.SetBool("BasicAttack", false);
            ani.SetBool("FireballAttack", false);
            ani.SetBool("BasicAttack2", false);
            ani.SetBool("KickAttack", false);
            minDistAtaque = 4;
        }
        break;
```

146. Capturas Script Boss para el ataque en salto.

Si el número aleatorio generado es igual a 1, si la fase en la que se encuentra es la 2 y está a una distancia mayor que para realizar un ataque a melee, se ejecutará el ataque Fireball poniendo el parámetro FireballAttack a true y el resto a false. En el caso de que estemos en cualquiera de las dos fases, pero cerca, lanzaremos el ataque Mutant Swiping, por lo que pasaremos el parámetro BasicAttack a true y el resto a false.

```
case 1:
    if(!attacking){
        if(fase == 2 && lejos) {
            attacking = true;
            minDistAtaque = 6;
            ani.SetBool("FireballAttack", true);
            ani.SetBool("BasicAttack", false);
            ani.SetBool("JumpAttack", false);
            ani.SetBool("BasicAttack2", false);
            ani.SetBool("KickAttack", false);

        } else if(!lejos){
            attacking = true;
            minDistAtaque = 3;
            ani.SetBool("KickAttack", false);
            ani.SetBool("JumpAttack", false);
            ani.SetBool("BasicAttack", true);
            ani.SetBool("FireballAttack", false);
            ani.SetBool("BasicAttack2", false);
        }
    }
    break;
```

147. Capturas Script Boss para el caso en que rutina sea igual a 1.

En el caso en el que el número aleatorio es igual a 2, si nos encontramos en la fase 1 y cerca, haremos un ataque básico por lo que pondremos el parámetro BasicAttack a true, pero si estamos en la fase 2, lanzaremos un ataque básico de la fase 2 por lo que pondremos el parámetro BasicAttack2 a true y en ambos casos el resto de los parámetros de ataque a false.

```
case 2:
    if(!attacking){
        if(fase == 2 && !lejos) {
            minDistAtaque = 4;
            ani.SetBool("BasicAttack2", true);
            ani.SetBool("BasicAttack", false);
            ani.SetBool("JumpAttack", false);
            ani.SetBool("FireballAttack", false);
            ani.SetBool("KickAttack", true);
        } else if(fase == 1 && !lejos) {
            minDistAtaque = 3;
            ani.SetBool("BasicAttack", true);
            ani.SetBool("BasicAttack2", false);
            ani.SetBool("JumpAttack", false);
            ani.SetBool("FireballAttack", false);
            ani.SetBool("KickAttack", false);

        }
    }
    break;
```

148. Capturas Script Boss para el caso en que rutina sea igual a 2.

En el caso en el que el número aleatorio es igual a 3, si nos encontramos en la fase 1 y cerca, haremos un ataque básico por lo que pondremos el parámetro BasicAttack a true pero si estamos en la fase 2 y lejos, lanzaremos la bola de fuego por lo que pondremos el parámetro Fireball a true y en ambos casos el resto de parámetros de ataque a false.

```
case 3:
    if(!attacking){
        if(fase == 2 && lejos) {
            minDistAtaque = 4;
            ani.SetBool("BasicAttack", false);
            ani.SetBool("JumpAttack", false);
            ani.SetBool("FireballAttack", true);
            ani.SetBool("BasicAttack2", false);
            ani.SetBool("KickAttack", false);
            attacking = true;

        } else if(fase == 1 && !lejos) {
            minDistAtaque = 3;
            attacking = true;
            ani.SetBool("BasicAttack", true);
            ani.SetBool("JumpAttack", false);
            ani.SetBool("FireballAttack", false);
            ani.SetBool("BasicAttack2", false);
            ani.SetBool("KickAttack", false);
        }
    }
    break;
```

149. Capturas Script Boss para el caso en que rutina sea igual a 3.

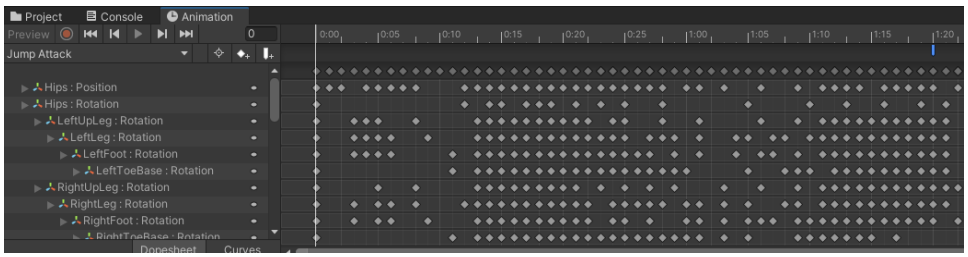
Para el resto de los valores del número aleatorio realizaremos un ataque básico sin tener en cuenta en qué fase estamos.

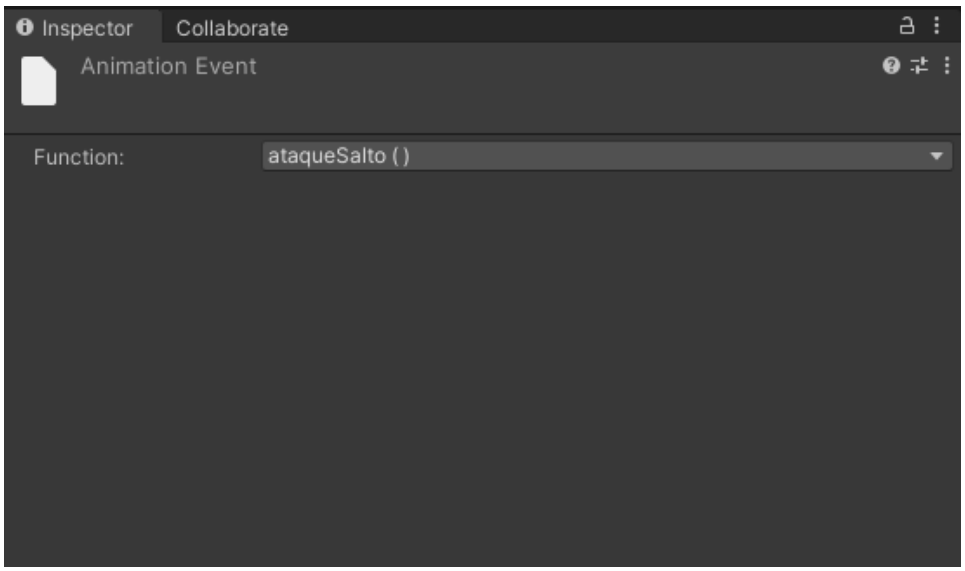
```
default:
    if(!attacking){
        minDistAtaque = 3;
        attacking = true;
        ani.SetBool("BasicAttack", true);
        ani.SetBool("JumpAttack", false);
        ani.SetBool("FireballAttack", false);
        ani.SetBool("BasicAttack2", false);
        ani.SetBool("KickAttack", false);
    }
    break;
```

150. Capturas Script Boss para el caso default.

En este script, también se gestiona los métodos para detectar si hacemos daño al jugador al realizar un ataque. Esta detección se realiza mediante un RaycastHit y un detector situado a la altura de la cadera del NPC y cuando lancemos este método, se lanzará un láser desde el detector hacia adelante una distancia que nosotros pongo y en caso de que choque con un objeto y el objeto con el que choca es el jugador, lanzará la animación de daño del jugador y le quitará la vida dependiendo del ataque. Tenemos un método para cada ataque.

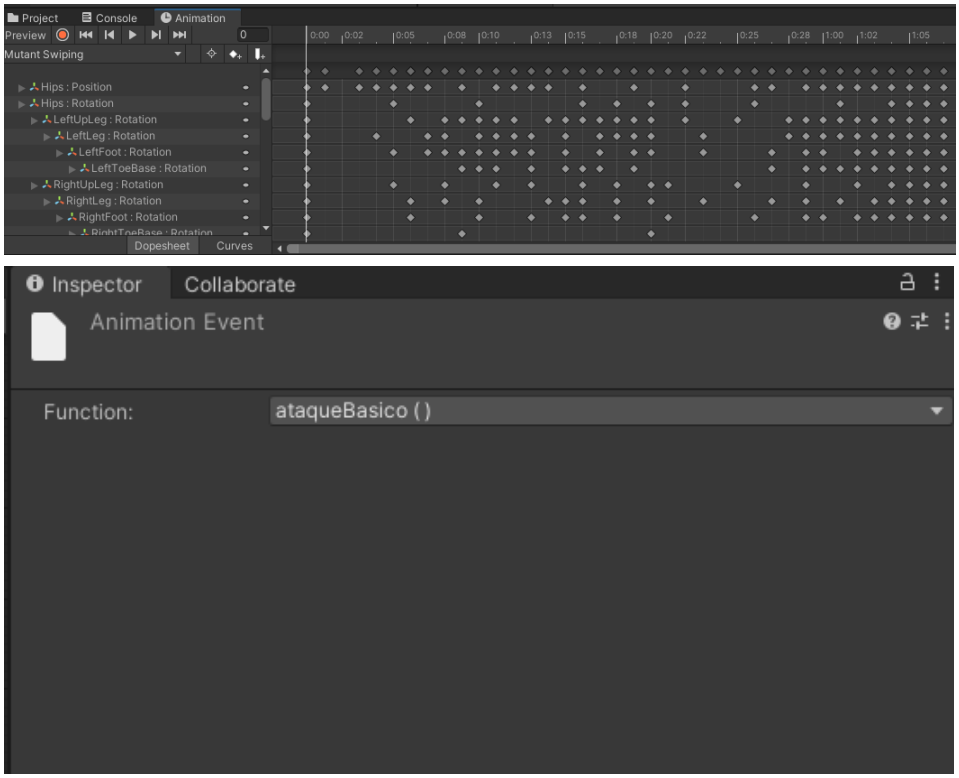
```
void ataqueSalto() {
    RaycastHit hit;
    attacking = true;
    transform.Translate(Vector3.forward * 8 * Time.deltaTime);
    if(Physics.Raycast(detector.position, transform.forward, out hit, minDistAtaque)){
        if(hit.collider.tag == "Player") {
            aj.GetComponent<Personaje>().minHp -= (damageSalto - aj.GetComponent<Personaje>().adDmgReduction);
            hit.collider.gameObject.GetComponent<Personaje>().anima.Play("Damage");
        }
    }
    ani.SetBool("JumpAttack", false);
}
```





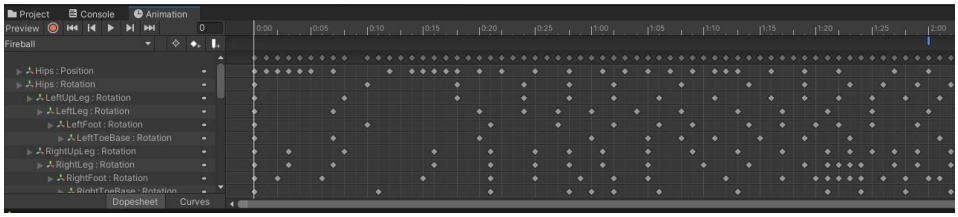
151. Ataque en salto boss.

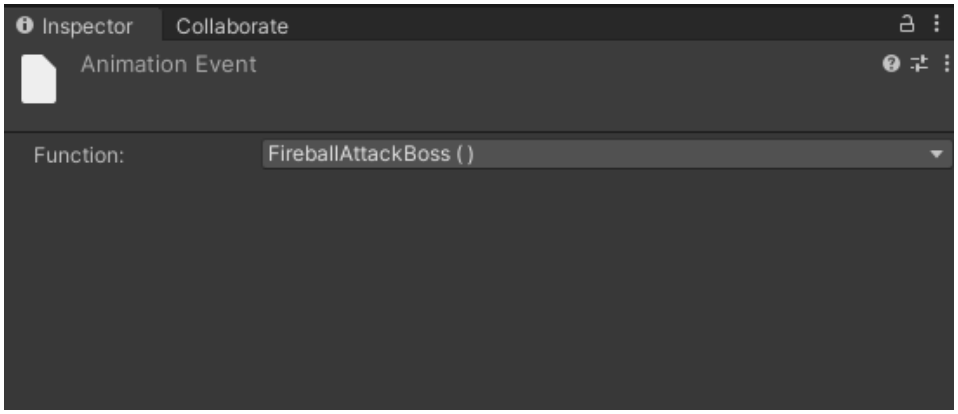
```
void ataqueBasico() {
    RaycastHit hit;
    attacking = true;
    if(Physics.Raycast(detector.position, transform.forward, out hit, minDistAtaque)){
        if(hit.collider.tag == "Player") {
            aj.GetComponent<Personaje>().minHp -= (damageAA - aj.GetComponent<Personaje>().adDmgReduction);
            hit.collider.gameObject.GetComponent<Personaje>().anima.Play("Damage");
        }
    }
    ani.SetBool("BasicAttack", false);
}
```



152. Ataque básico boss.

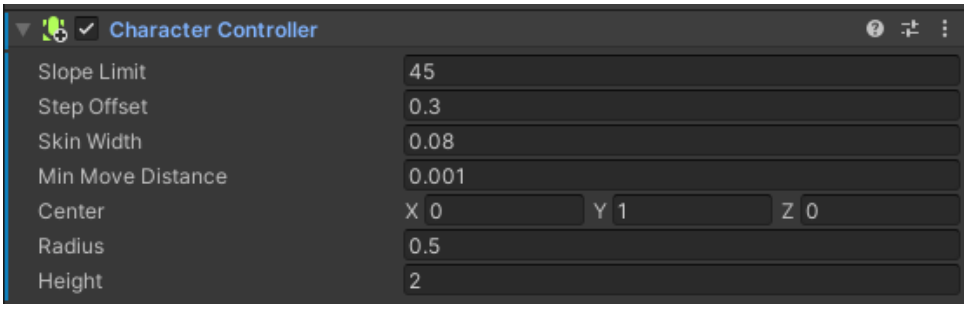
```
void FireballAttackBoss() {
    attacking = true;
    Instantiate(fireball, salidaFireball.position, salidaFireball.rotation);
    ani.SetBool("FireballAttack", false);
}
```





153. Ataque fireball boss.

- CharacterController:

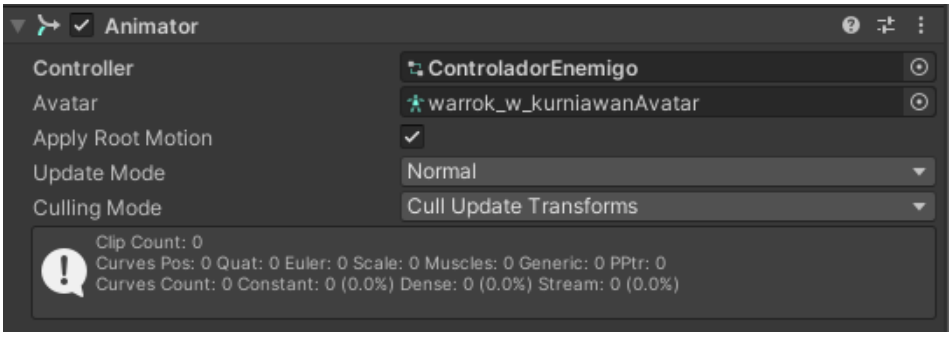


154. Character Controller Boss.

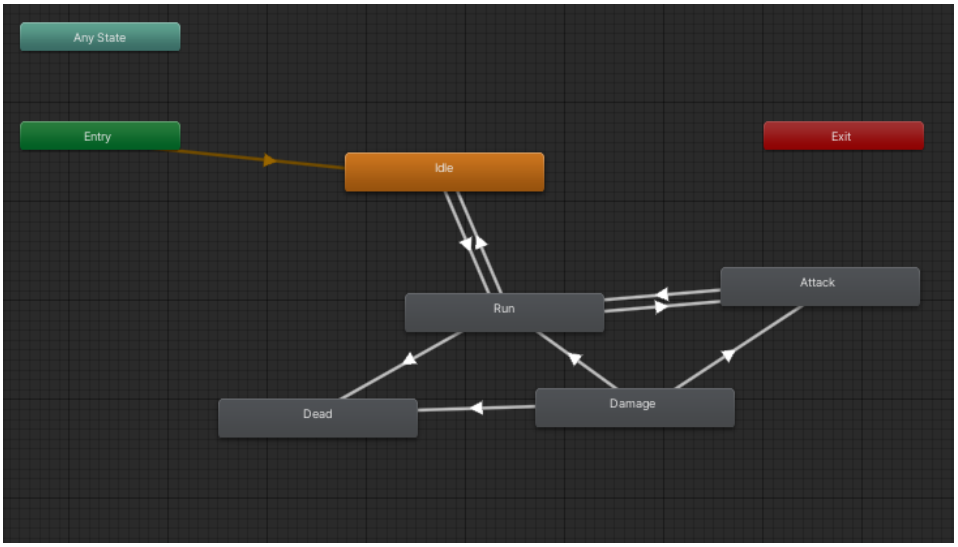
- **Warrok:**

Descripción:
Este personaje aparecerá en el juego una vez que hablemos con el paisano Warrok, el cual nos pedirá que le ayudemos a echar a este. En la "historia" este personaje es un secuaz de Maw Jaygo con el cuál al matarle el jugador ganará un botín especial y con el cuál se le terminará permitiendo ir a pelear contra Maw Jaygo.

Componentes:
- Animator:



155. Componente Animator del personaje Warrok.



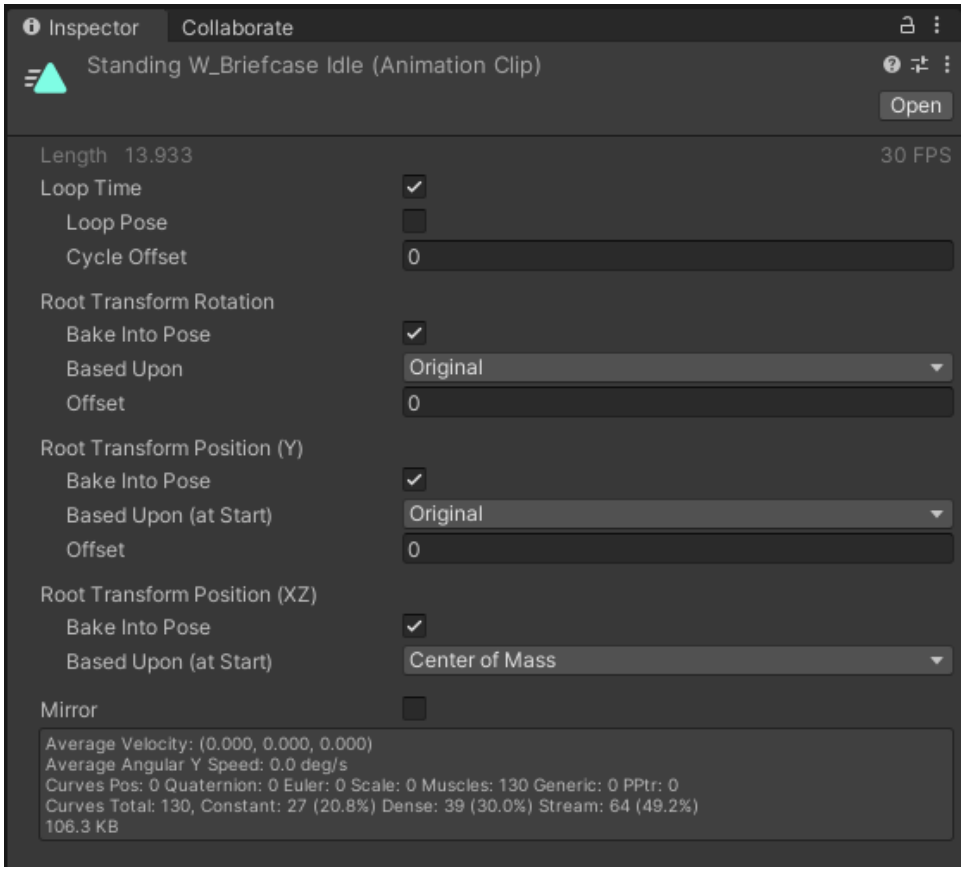
156. Controller del personaje Warrok.

Lo primero que vamos a hacer para explicar el diagrama es mostrar las diferentes animaciones que van asociadas a cada estado.

Estados:

- Idle:

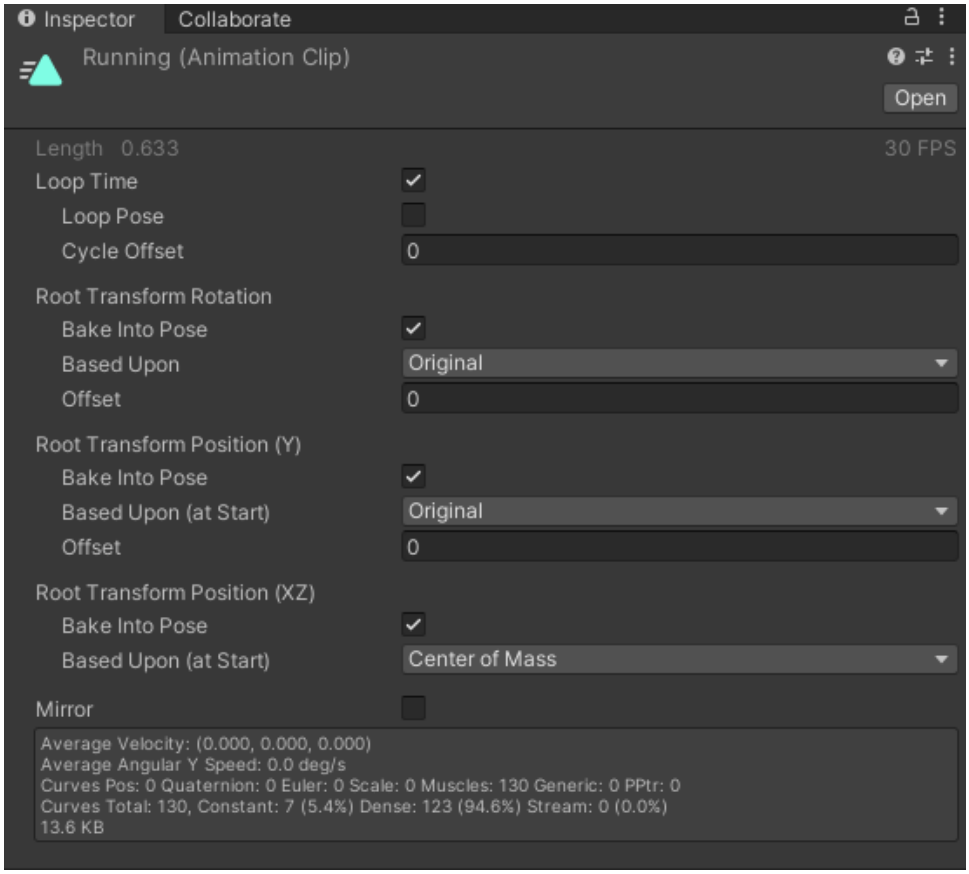
Como podemos ver en la imagen siguiente, mientras el personaje se encuentre en este estado, se reproducirá la animación idle hasta que
pasemos al siguiente estado y cambie el estado y por tanto la animación. Marcamos la casilla Loop time para que esta animación no
se ejecute una única vez si no que se esté ejecutando constantemente mientras que estemos en este estado.



157. Captura del estado Idle del personaje Warrok.

- Run:

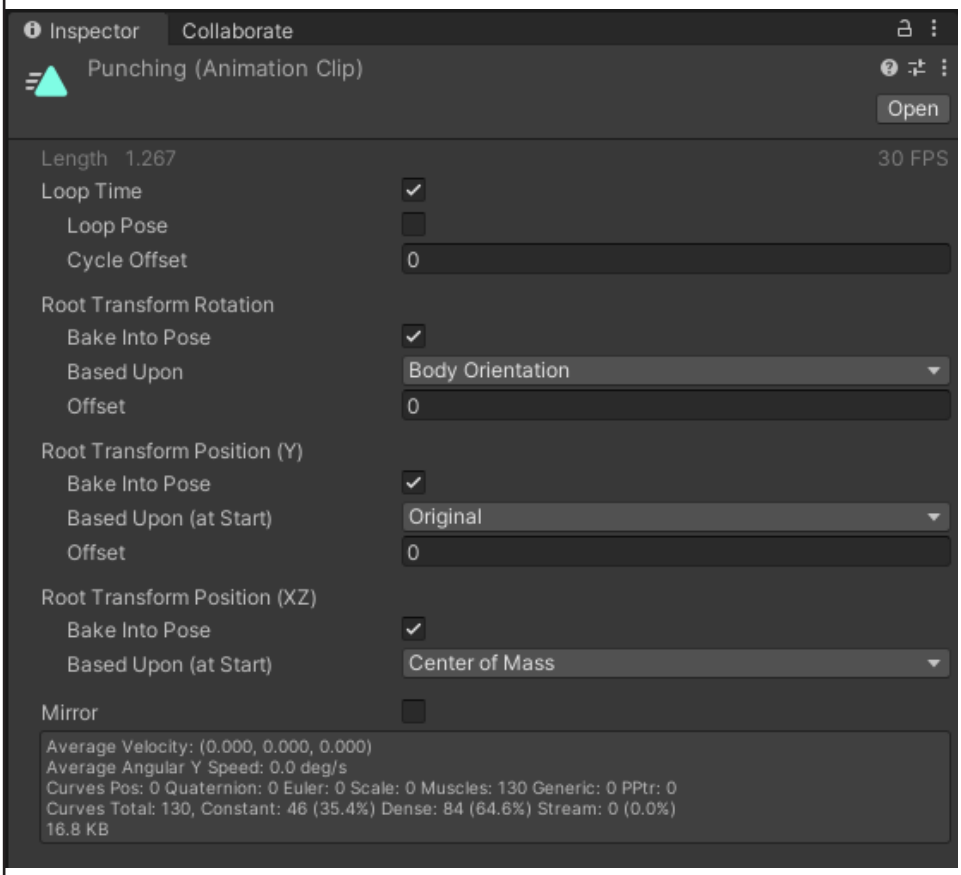
En este estado, la animación que se ejecutará será la animación Running al igual que comentamos en el estado idle, marcaremos la casilla loop time para que se ejecute en bucle hasta que salgamos del estado.



158. Captura de la animación Running del personaje Warrok.

- Attack:

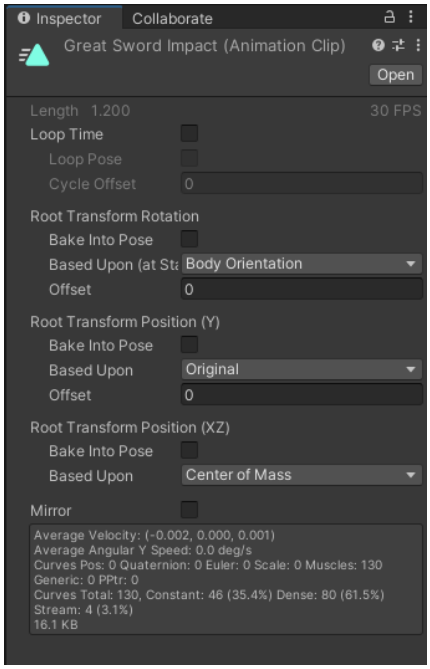
En este estado, se ejecutará la animación Punching y marcaremos como en los anteriores la casilla loop time por las mismas razones que en los estados anteriores.



159. Captura de la animación ejecutada en el estado Attack del personaje Warrok.

- Damage:

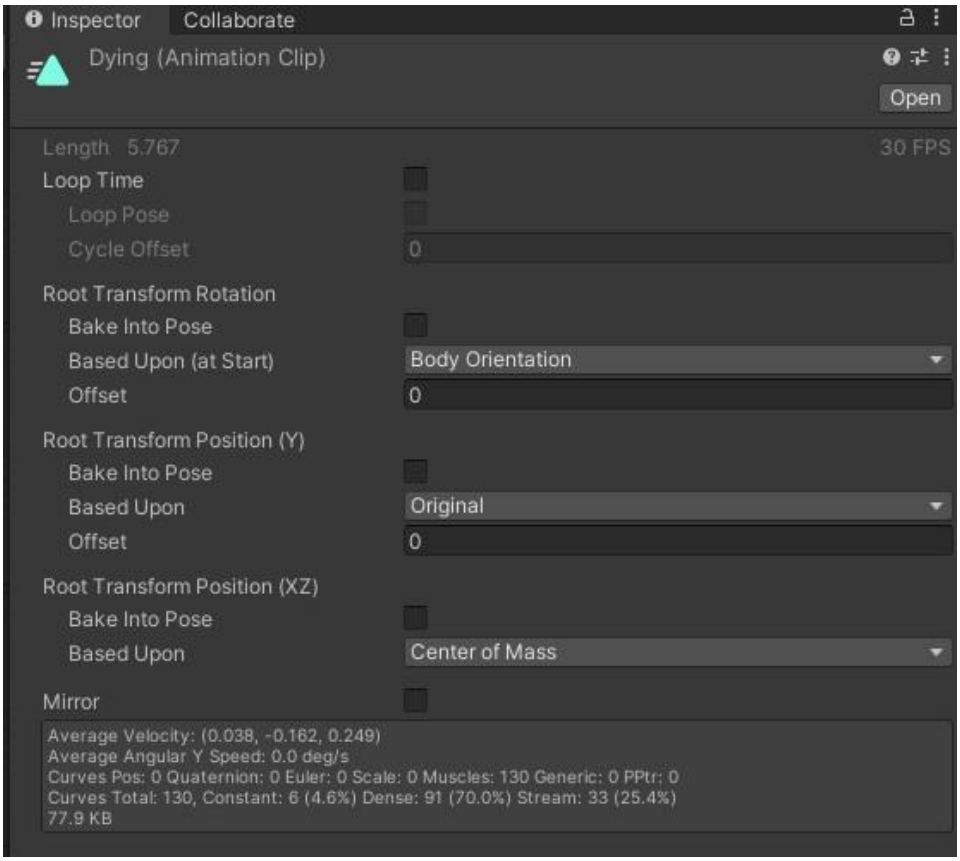
En el estado Damage, la animación ejecutada será la animación Great Sword Impact. En este caso la casilla Loop time queda desmarcada ya que sólo queremos que se reproduzca la animación una vez en el momento que este reciba un golpe de nuestro personaje.



160. Captura de la animación ejecutada en el estado Damage del personaje Warrok.

- Dead:

En este estado se ejecutará la animación Dying. En este caso la casilla loop time también quedará desmarcada porque al ser una animación de muerte, sólo queremos que se ejecute una vez cuando el personaje muere y no en bucle.



161. Captura de la animación Dying ejecutada en el estado Dead del personaje Warrok.

Para continuar, adjunto una imagen de los distintos parámetros que son los que van a hacer de triggers para que se activen las transiciones de un estado a otro para poder explicar mejor las transiciones.

162. Parámetros del controller del personaje Warrok.

Una vez explicados los diferentes estados posibles con sus animaciones y los parámetros que existen en el controller, vamos a pasar a explicar las diferentes transiciones que hay entre estados.

Transiciones :

- Idle -> Run:

Esta transición se ejecutará en el momento en que el valor del parámetro detectado sea igual a true. Esto ocurrirá en el momento en el que nuestro personaje entre en el rango de detección del warrok y gestionado por el script de comportamiento (EnemigoScript) como podemos ver en el siguiente fragmento de código en el que podemos ver que si la distancia entre el jugador y el NPC es menor que la máxima distancia de detección que hayamos puesto al Warrok el parámetro detectado pasará a ser true y en caso contrario el parámetro
detectado pasará a ser false.

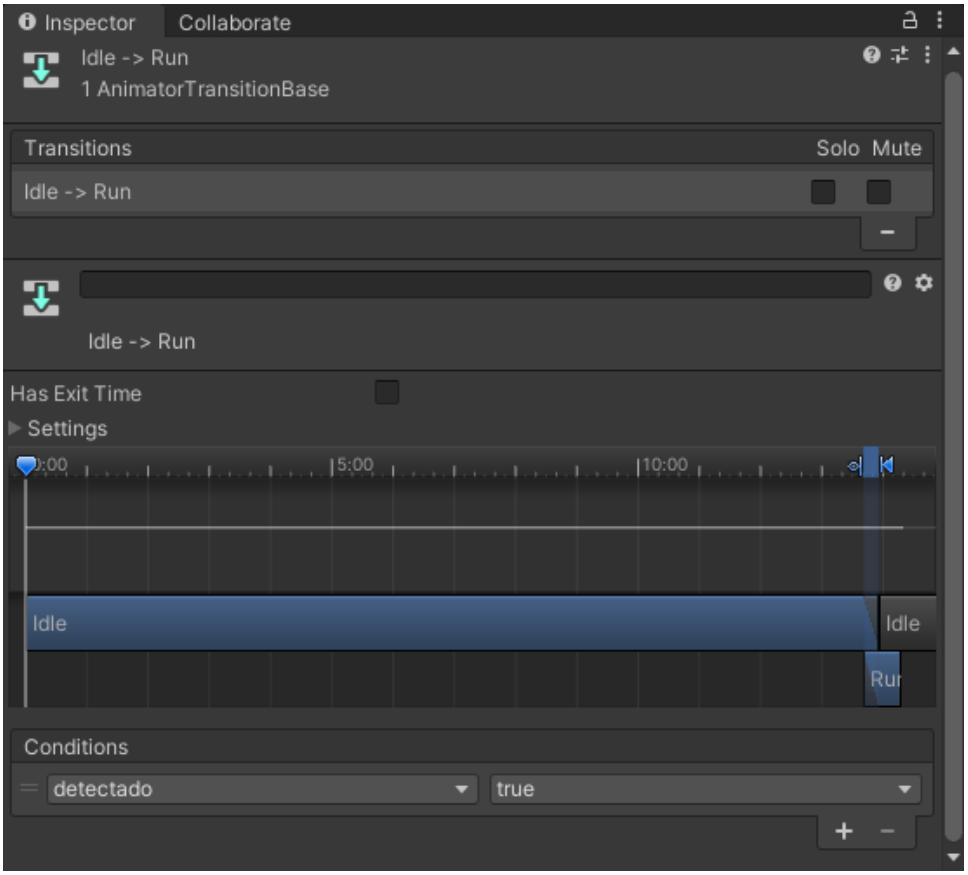
```
if(aj != null) {
    float distancia = Vector3.Distance(transform.position,aj.transform.position);
    if(distancia < maxDist && distancia > minDist) {
        detectado = true;
    }
    else {
        detectado = false;
    }

    Vector3 rotacion = new Vector3(aj.transform.position.x, transform.position.y, aj.transform.position.z);

    if(detectado) {
        transform.LookAt(rotacion);
        ani.SetBool("detectado", true);
        ani.SetBool("attacking", false);
        if(!attacking){
            characterCont.SimpleMove(transform.forward * velocidadMovimiento * Time.deltaTime);
        }
    }
    else {
        ani.SetBool("detectado", false);
    }
```

163. Captura del script EnemigoScript para gestión parámetro detectado.

En cuanto a la casilla Has exit time de la transición la dejaremos desmarcada, ya que no queremos obligar a que se tenga que ejecutar toda la animación de Idle para empezar a ejecutarse la animación de Run.



164. Transición del estado Idle -> Run personaje Warrok.

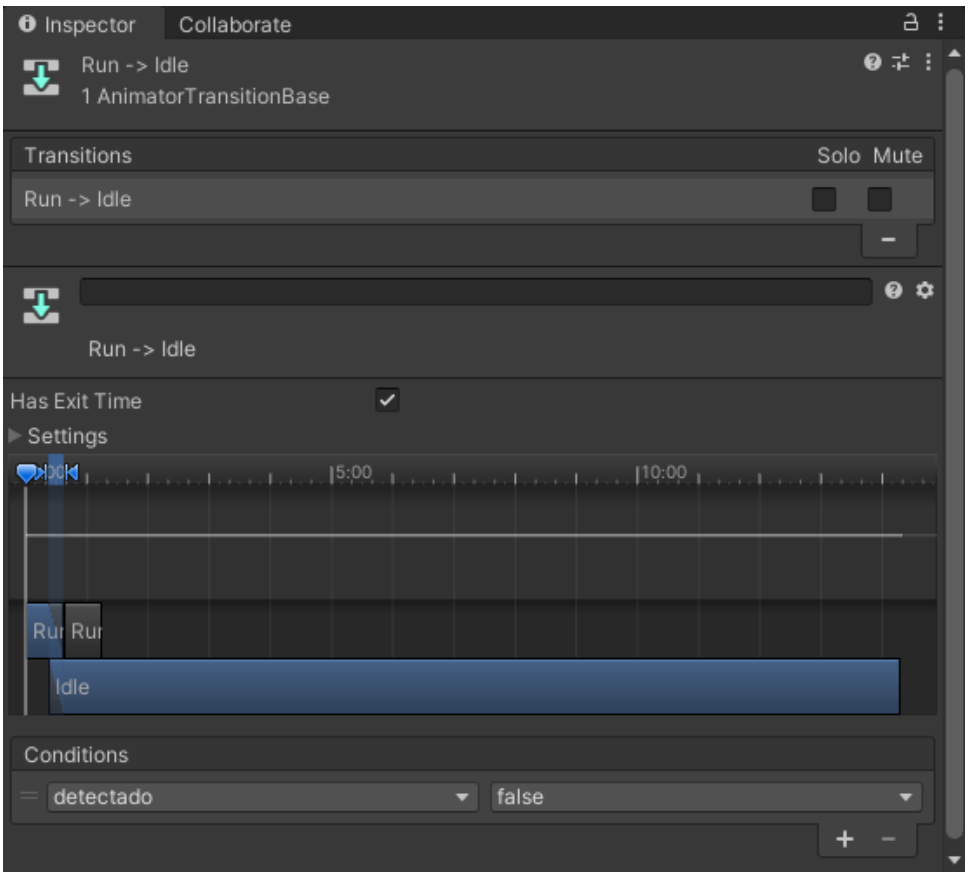
- Run -> Idle.

Esta transición será ejecutada en el momento en el que mientras el warrok está corriendo para perseguir a nuestro personaje, este sale de su campo de detección y por tanto se para, pasando del estado Run al estado Idle.
El trigger de esta transición, es el paso del parámetro detectado del controlador de true a false. La gestión de este paso de un estado a otro, también se gestiona en el script EnemigoScript en este fragmento de código.

```
if(detectado) {
    transform.LookAt(rotacion);
    ani.SetBool("detectado", true);
    ani.SetBool("attacking", false);
    if(!attacking){
        characterCont.SimpleMove(transform.forward * velocidadMovimiento * Time.deltaTime);
    }
}
else {
    ani.SetBool("detectado", false);
}
```

165. Captura del script EnemigoScript para gestión parámetro detectado en caso de estar el jugador a una distancia mayor que la distancia máxima establecida de campo de visión.

En esta transición al igual que en las anteriores no se marcará la opción has exit time, ya que queremos que en el momento que se deje de detectar a nuestro personaje la maga se quede parada y no tenga que terminar la animación de correr.



166. Transición del estado Run -> Idle personaje Warrok

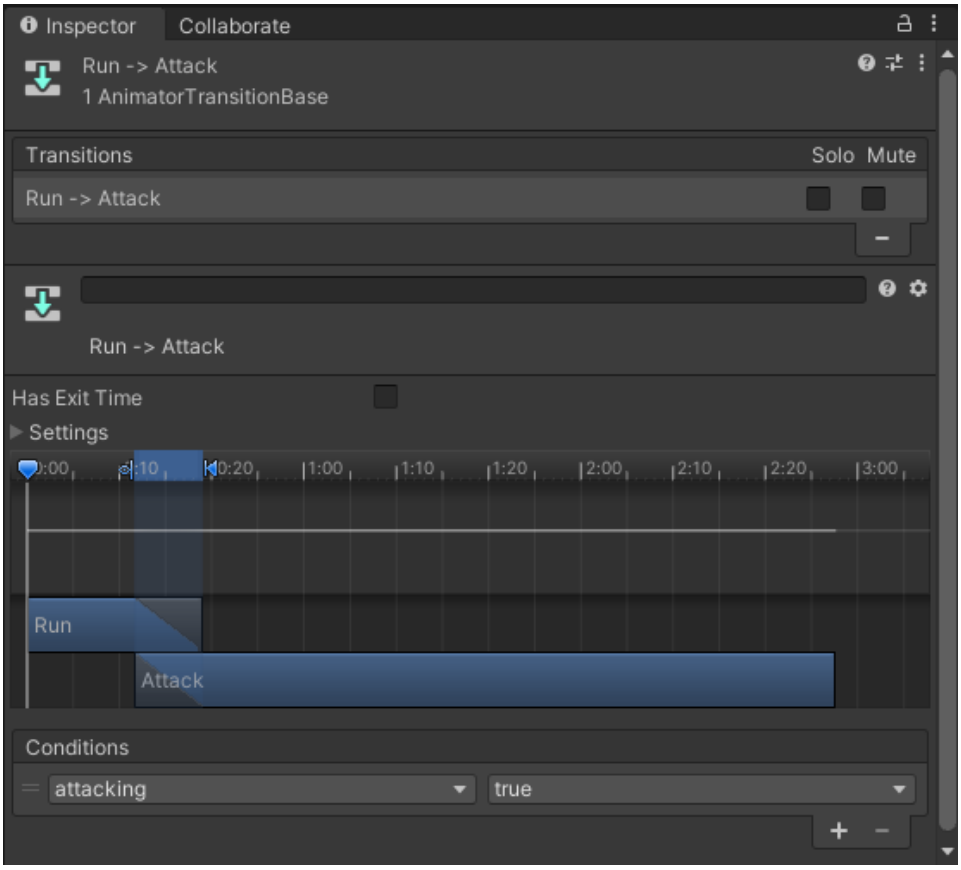
- Run -> Attack

Para que esta transición sea lanzada, lo que tiene que ocurrir es que mientras el warrok nos persigue consiga acercarse a nosotros a la distancia estipulada de ataque que le queramos poner, se activa la transición y lanza el ataque.

```
        }
        if(distancia <= minDist) {
            detectado = false;
            transform.LookAt(rotacion);
            ani.SetBool("attacking", true);
        }
```

167. Captura del script EnemigoScript para gestión parámetro attacking pasando a true en caso de estar el jugador a la mínima distancia necesaria para lanzar el ataque.

En este caso, tampoco activaremos la opción Has exit time, por las mismas razones mencionadas anteriormente.



168. Transición del estado Run -> Attack personaje Warrok

- Attack -> Run:

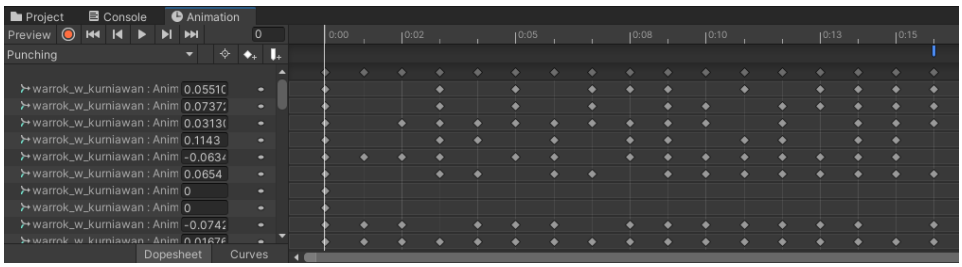
Esta transición se ejecutará en el caso de que después de lanzar el ataque, el personaje siga dentro del campo de detección.
En este caso, como lo que queremos es modificar el parametro attacking sólo una vez que se haya lanzado el ataque, crearemos una función que añadiremos como evento en la animación del ataque en el momento que lanza el puño el warrok. Esta gestión se hará tanto desde el script EnemigoScript como desde la interfaz de Unity en la pestaña Animation para añadir el evento.

```
void ataque() {
    RaycastHit hit;

    if(Physics.Raycast(detector.position, transform.forward, out hit, distAtaque)){
        if(ani.GetCurrentAnimatorStateInfo(0).IsName("Base Layer.Attack")){
            if(hit.collider.tag == "Player") {
                aj.GetComponent<Personaje>().minHp -= (damage - aj.GetComponent<Personaje>().adDmgReduction);
                hit.collider.gameObject.GetComponent<Personaje>().anima.Play("Damage");
            }
        }
    }

}
```

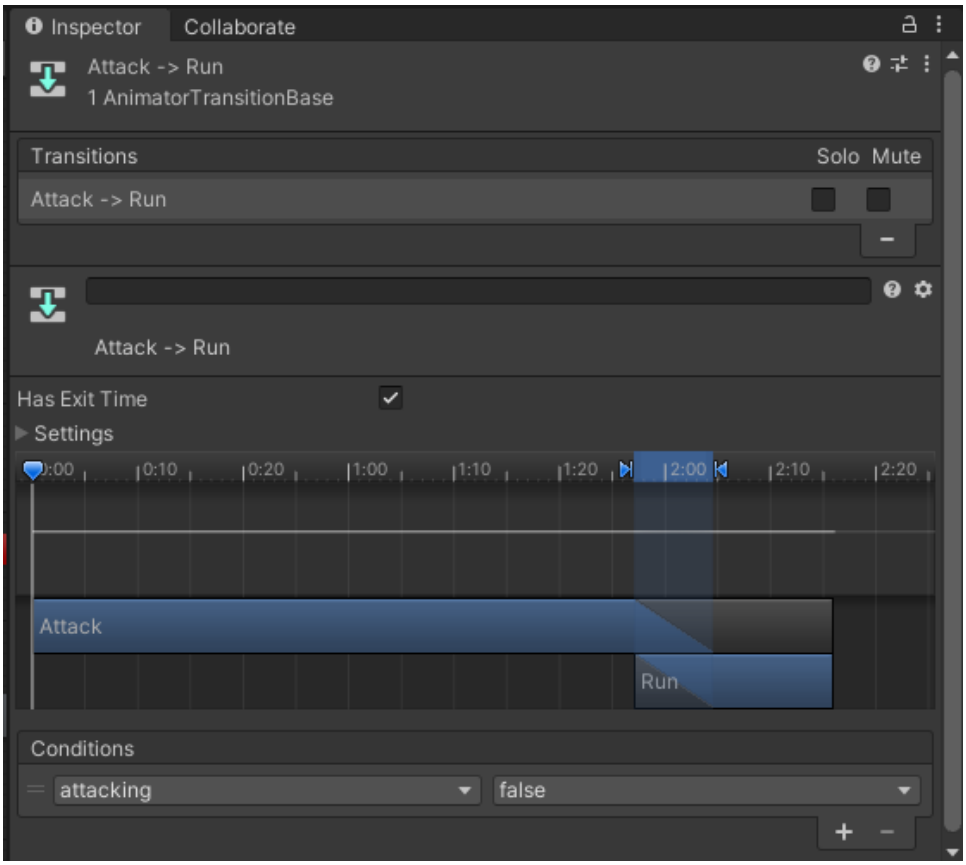
169. Captura del script EnemigoScript para gestión parámetro attacking pasando a false en caso de que el ataque ya ha sido lanzado.



170. Captura de la pestaña animation con el evento añadido en el segundo 16.

171. Captura del evento con la función asignada.

En este caso, sí que marcaremos la opción Has exit time ya que queremos que la animación se ejecute completamente antes de volver al estado Run.



172. Transición del estado Fireball -> Run personaje Maga

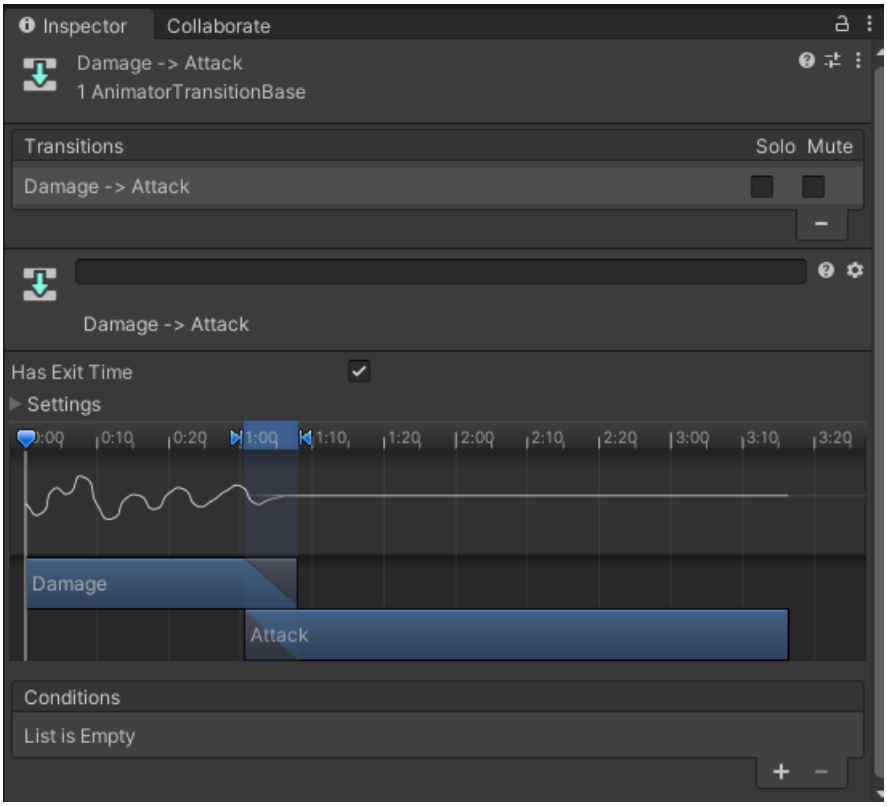
- Damage -> Attack:

Esta transición ocurre cuando después de recibir daño, si hp sigue siendo mayor que 0, el parámetro muerto será false y volverá al estado de ataque Attack. Esto se gestiona en el script personaje en un método que si el tag del objeto al que ataca es el del warrok, se pasa al estado Damage en el controller del warrok y por tanto se ejecuta la animación de recibir daño.

```
void atacando() {
    RaycastHit hit;
        if(anima.GetCurrentAnimatorStateInfo(0).IsName("Base Layer.Boxing")){
            if(Physics.Raycast(detector.position, transform.forward, out hit, 3)){
                switch(hit.collider.tag) {
                    case "Enemigo":
                        hit.collider.gameObject.GetComponent<EnemigoScript>().hp -= damage;
                        hit.collider.gameObject.GetComponent<EnemigoScript>().ani.Play("Damage");
                        break;
                    case "Mage":
                        hit.collider.gameObject.GetComponent<MagaScript>().hp -= damage;
                        hit.collider.gameObject.GetComponent<MagaScript>().ani.Play("Damage");
                        break;
```

173. Captura del script Personaje para lanzamiento del estado Damage en caso de que el personaje al que atacamos sea el Warrok (case “Warrok”).

Para este caso, también dejaremos marcada la casilla Has exit time ya que queremos que la animación de daño se ejecute por completo antes de volver a atacar.



174. Transición del estado Damage -> Attack personaje Warrok

- Damage -> Run:

Esta transición ocurre cuando después de recibir daño, si hp sigue siendo mayor que 0, si el parámetro detectado es igual a true, pasará al estado Run.
Para llegar al estado Damage al igual que en la transición anterior, se gestiona en el script Personaje.
Esta gestión se hace en el script EnemigoScript en este fragmento de código:

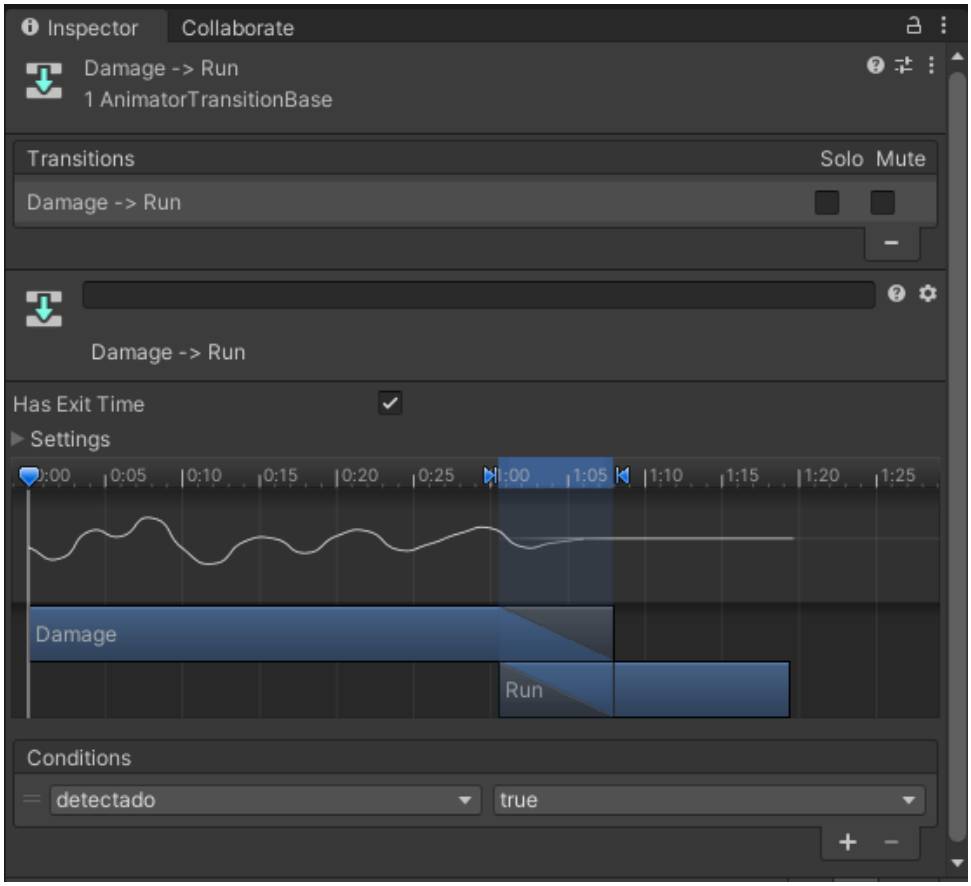
```
if(aj != null) {
    float distancia = Vector3.Distance(transform.position,aj.transform.position);
    if(distancia < maxDist && distancia > minDist) {
        detectado = true;
    }
    else {
        detectado = false;
    }

    Vector3 rotacion = new Vector3(aj.transform.position.x, transform.position.y, aj.transform.position.z);

    if(detectado) {
        transform.LookAt(rotacion);
        ani.SetBool("detectado", true);
        ani.SetBool("attacking", false);
        if(!attacking){
            characterCont.SimpleMove(transform.forward * velocidadMovimiento * Time.deltaTime);
        }
    }
    else {
        ani.SetBool("detectado", false);
    }
```

175. Captura del script EnemigoScript para la gestión del parámetro detectado.

En este caso también dejaremos marcada la opción Has exit time.



176. Transición del estado Damage -> Run personaje Warrok

- Damage -> Dead

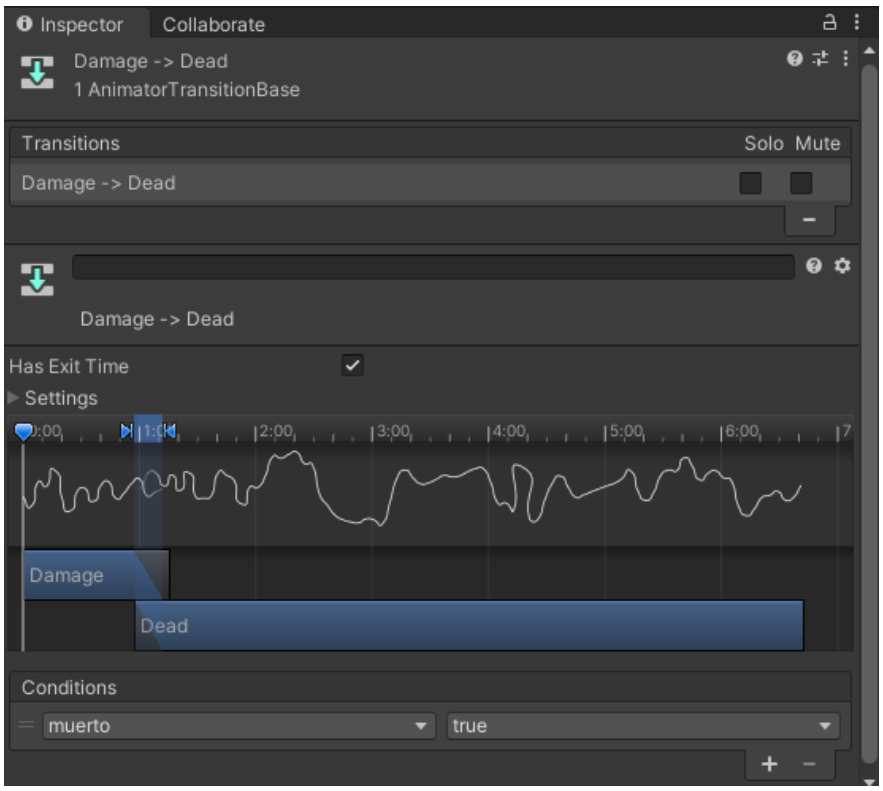
Si después de ejecutar desde el script personaje la animación de daño el atributo hp de la maga es menor o igual que 0, el parámetro muerto pasará a true y se pasará al estado Dead que ejecutará la animación de muerte.

Esta gestión se hace en este fragmento de código del script EnemigoScript:

```
(hp <= 0) {
  hp = 0;
  detectado = false;
  ani.SetBool("detectado", false);
  ani.SetBool("attacking", false);
  ani.SetBool("muerto", true);
  dejarLoot();
  Destroy(this.gameObject.GetComponent<EnemigoScript>());
  Destroy(this.gameObject.GetComponent<CharacterController>());
  Destroy(this.gameObject, 10);
```

177. Captura del script EnemigoScript para la gestión de la muerte del Warrok.

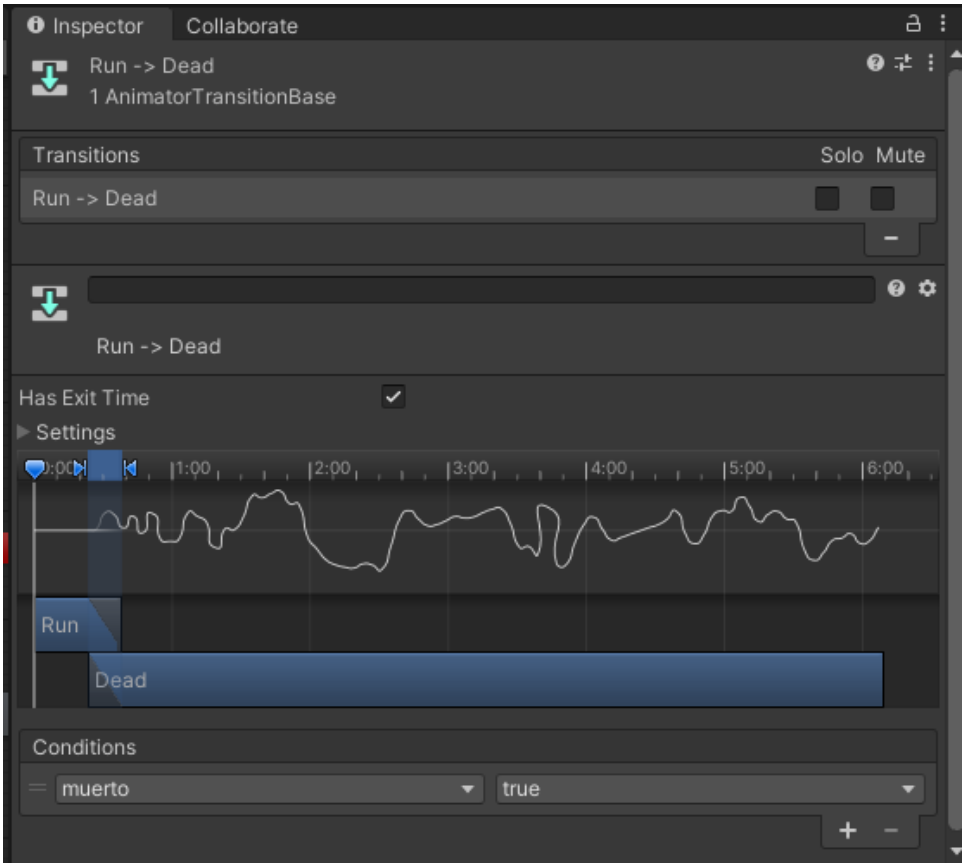
Por las mismas razones que en las anteriores transiciones Damage -> X, dejaremos la casilla Has exit time marcada.



178.Transición del estado Damage -> Dead personaje Maga

- Run -> Dead:

Esta transición se ejecuta si mientras estamos en el estado Run, el parámetro muerto pasa a true, la gestión del parámetro muerto es igual que para la anterior transición. En este caso la casilla Has exit time no la marcamos ya que si el personaje muere mientras está corriendo no queremos que termine de ejecutar la acción de correr antes de morir si no que muera en el mismo momento que su vida pase a valer 0.



179. Transición del estado Run -> Dead personaje Warrok

- **Inventario + tienda + objetos.**

Objetos

Los objetos son un tipo de NPC diferentes a los explicados anteriormente. Para la gestión de estos, lo que hemos hecho en primer lugar es crear un ScriptableObject el cuál va a tener dentro un tipo de objeto llamado ObjetoInventario, y una lista de tipo ObjetoInventario.

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

[CreateAssetMenu(fileName = "BaseDatos", menuName = "TFGJuego/Inventario", order = 1)]
public class BaseDatos : ScriptableObject {
    [System.Serializable]
    public struct ObjetoInventario {
        public string nombre;
        public int id, precio, cantidad;
        public Sprite imagen;
        public Type type;
        public ParteCuerpo parteCuerpo;
        public bool acum;
        public string descripcion;
    }

    public enum Type {
        NA,
        CONSUMIBLE,
        EQUIPABLE
    }

    public enum ParteCuerpo {
        NA,
        CABEZA,
        CUERPO,
        BRAZO,
        PIE
    }
    public List<ObjetoInventario> dataBase;
}
```

180. Script Base de datos.

En nuestro juego, va a haber dos tipos de objetos objetos Equipables y objetos Consumibles. Y dentro de los Equipables, cada objeto podrá ser equipado solamente en una parte del cuerpo del jugador. Aparte de esto, los objetos dependiendo del booleano acum podrán ser o no acumulables.
Aparte de estos atributos, todos los objetos llevan asignado un Script llamado ObjetoInv.

- ObjetoInv:

Este script, se ocupa de dotar de cierto comportamiento a los objetos, como por ejemplo cuando han sido instanciados que puedan ser looteados siendo añadidos en el inventario en caso de que este no esté lleno, además de destruirlo una vez se ejecuta la animación de loot.

Además, tendrá implementados los métodos OnPointerEnter y OnPointerExit que para la gestión del inventario serán muy útiles ya que capturan cuando hemos colocado el ratón encima del objeto y cuando dejamos de tenerlo encima.
Los objetos lo que hacen es mejorar a nuestro personaje en el caso de los equipables y curar vida en caso de los consumibles.

```
// Start is called before the first frame update
void Start()
{
    aj = GameObject.Find("aj");
    PanelObjetoInteraccion = GameObject.Find("Canvas").transform.GetChild(0).transform.GetChild(8).gameObject;
    PanelInteraccionInventarioLleno = GameObject.Find("Canvas").transform.GetChild(0).transform.GetChild(9).gameObject;
    if(cantidadTxt != null) {
        cantidadTxt.transform.SetParent(this.transform);
        cantidadTxt.transform.position = transform.position + offsetCantidad;
    }
    if(description != null) {
        nombreTxt = description.transform.GetChild(0).GetComponent<Text>();
        data = description.transform.GetChild(1).GetComponent<Text>();
        description.SetActive(false);
        if(!description.GetComponent<Image>().enabled) {
            description.GetComponent<Image>().enabled = true;
            nombreTxt.enabled = true;
            data.enabled = true;
        }
    }
}
```

181. Método Start del script ObjetoInv.

```
void Update()
{
    if(Input.GetKey("e") && jugadorCerca && !loot) {
        loot = true;
        PanelObjetoInteraccion.SetActive(false);
        if(!nombre.Equals("Moneda")) {
            bool lleno = aj.gameObject.GetComponent<Personaje>().inventario.AddObjeto(this);
            if(!lleno) {
                aj.gameObject.GetComponent<Personaje>().objetoADestruir = this.gameObject;
                aj.gameObject.GetComponent<Personaje>().anima.Play("Loot");
                if(cantidadTxt != null) {
                    cantidadTxt.text = cantidad.ToString();
                }
            } else {
                PanelInteraccionInventarioLleno.SetActive(true);
                loot = false;
            }
        } else {
            aj.gameObject.GetComponent<Personaje>().objetoADestruir = this.gameObject;
            aj.gameObject.GetComponent<Personaje>().anima.Play("Loot");
        }
    }
    if(cantidadTxt != null){
        if(cantidad == 0 || equipado){
            if(id == 0) {
                Debug.Log("Vacio cantidad: "+id);
            }
            cantidadTxt.text = "";
        }
        else {
            cantidadTxt.text = cantidad.ToString();
            cantidadTxt.transform.SetParent(this.transform);
            cantidadTxt.transform.position = transform.position + offsetCantidad;
        }
        if(transform.parent == Inventario.canvas) {
            description.SetActive(false);
        }
    }
}
```

182. Método Update del script ObjetoInv.

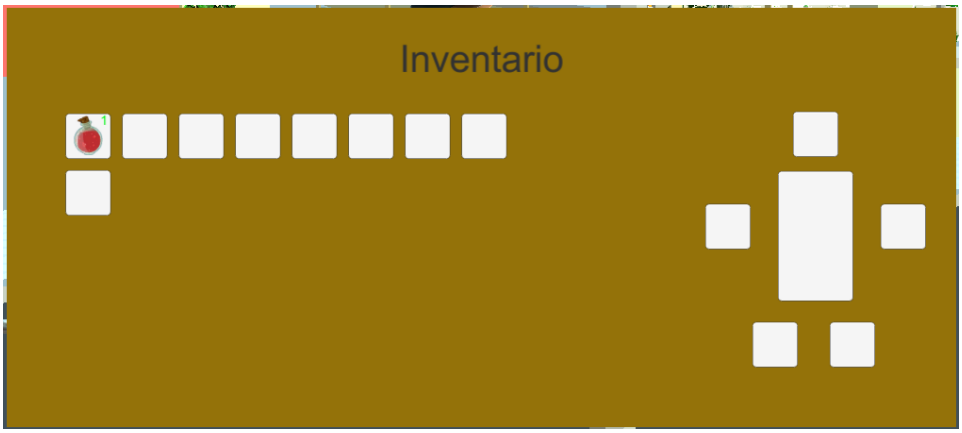
```
public void OnPointerEnter(PointerEventData eventPointer) {
    if(id != -1) {
        description.SetActive(true);
        nombreTxt.text = nombre;
        data.text = descripcion;
        description.transform.position = transform.position + offset;
    }
}

public void OnPointerExit(PointerEventData eventPointer) {
    description.SetActive(false);
}
```

183. Métodos OnPointerEnter y OnPointerExit del script ObjetoInv.

- **Inventario:**

El inventario, es un Canvas en el cuál creamos un contenedor que funciona como grid en el que añadimos una serie de celdas y en él, lo que podemos hacer es almacenar objetos, ver las descripciones de estos poder moverlos de una celda a otra, acumular los objetos que sean iguales y acumulables, consumir los objetos consumibles y además equipar los objetos equipables en las partes del cuerpo que le corresponda al objeto. A continuación, adjunto una captura del inventario.



184. Captura del inventario durante un gameplay.

- **Tienda:**

La tienda es un Canvas al igual que el inventario que nos ofrece una interfaz por la cual podemos comprar diferentes objetos por un precio permitiendonos comprarlo en el caso que tengamos espacio y dinero suficiente para ello, o no permitiendolo en caso contrario.

185. Captura de la tienda durante un gameplay.

Esta es la interfaz de la tienda, y en caso de que queramos comprar un objeto nos preguntará si lo queremos comprar por el dinero que cuesta, y decidiremos si queremos comprarlo o no en caso de que decidamos comprarlo si tenemos dinero suficiente y espacio en el inventario nos dirá objeto comprado correctamente. En el caso de no tener dinero nos dirá que no tenemos dinero suficiente y en caso de que el problema sea el espacio en el inventario nos dirá que el inventario está lleno.

186. Mensaje de pregunta para la compra.

187. Mensaje no hay dinero suficiente.

188. Mensaje compra correcta.

189. Mensaje el inventario está lleno.

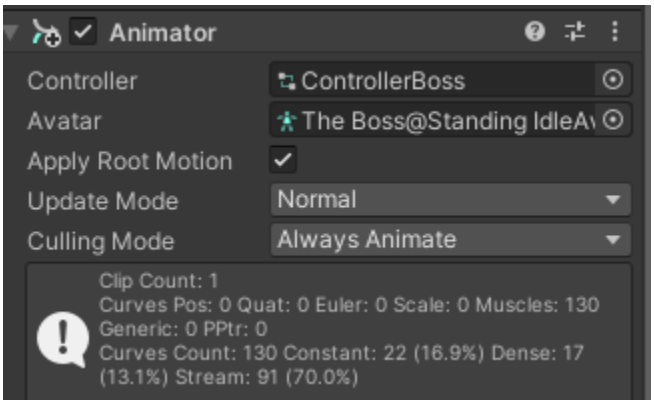
- **PaisanoAmigoTabernero:**

Descripción:
El funcionamiento de este NPC se basa en una máquina de estados y un árbol de decisión que en conjunto consiguen la funcionalidad completa. Aquí podemos ver la máquina de estados que se utiliza para este NPC.

En primer lugar, vemos que, al arrancar el juego, el personaje se encontrará en el estado Idle, es decir quieto esperando hasta que un input le haga cambiar de estado. Aunque en el caso de este personaje, sólo tiene este estado y nunca cambiará.
En este personaje la parte interesante está en el árbol de decisión, ya que será el que dotará de inteligencia al personaje.

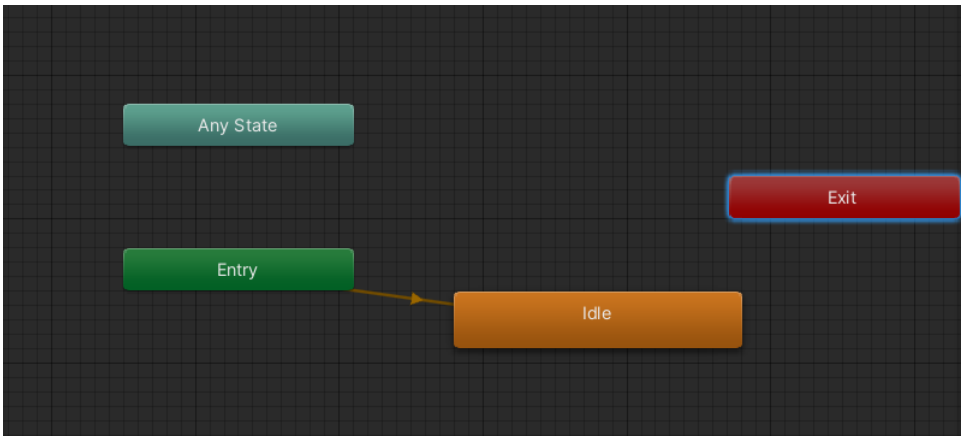
Componentes:

- Animator:



190. Animator PaisanoAmigoTabernero

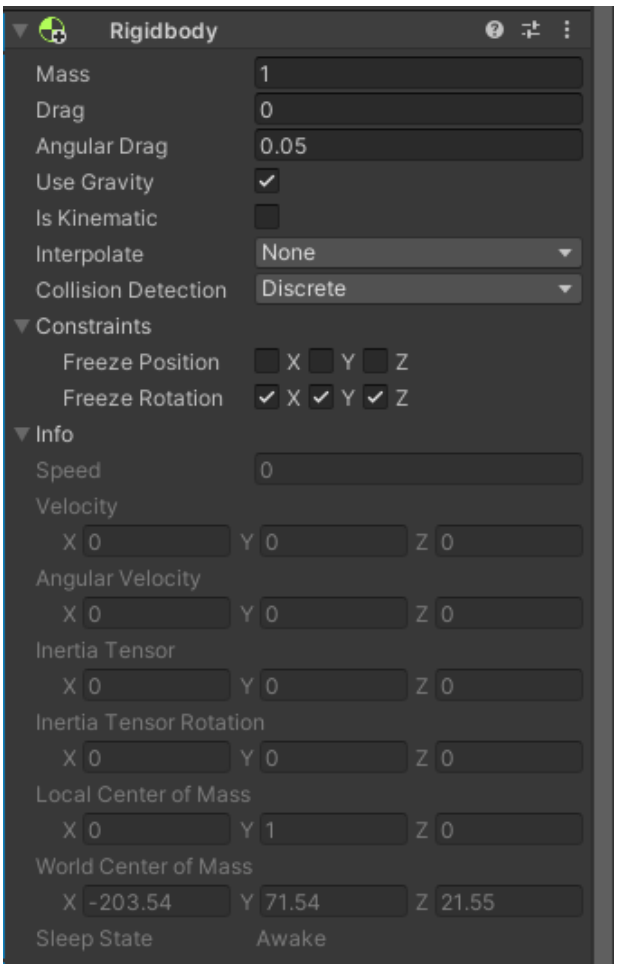
Este componente se ocupa a través del controller ControllerBoss de gestionar los diferentes estados en los que puede estar el personaje.
Este controller es una máquina de estados configurada de la siguiente manera:



191. Máquina de estados para el paisano Amigo Tabernero controller.

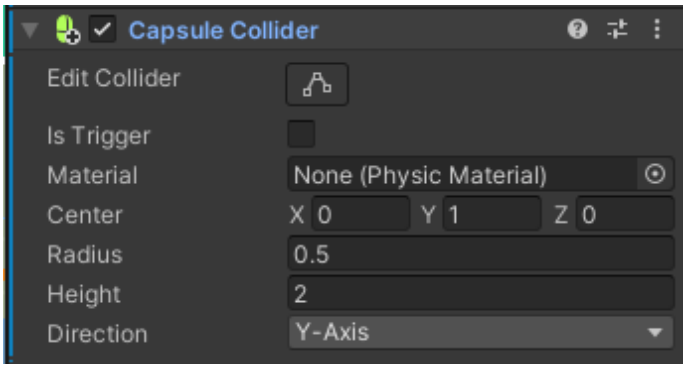
En este caso podemos ver que hay sólo un estado aparte de los ofrecidos por Unity en todos los controles que es el estado Idle y una única transición que va de Entry -> Idle.
Como hemos dicho en la descripción anteriormente, esta máquina de estados realmente no aporta una IA como tal al personaje, ya que da igual el input que reciba que siempre se va a mantener en el estado Idle el personaje.

- Rigidbody:



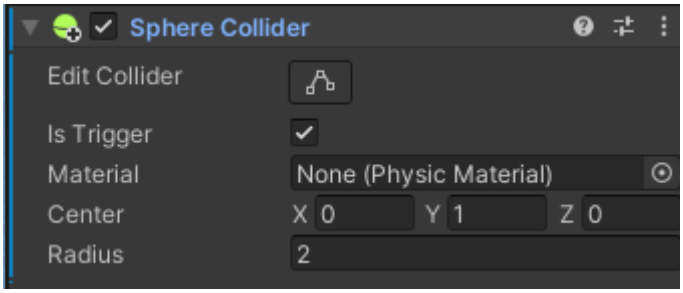
192. Componente Rigidbody del personaje PaisanoAmigoTabernero.

- Capsule Collider:



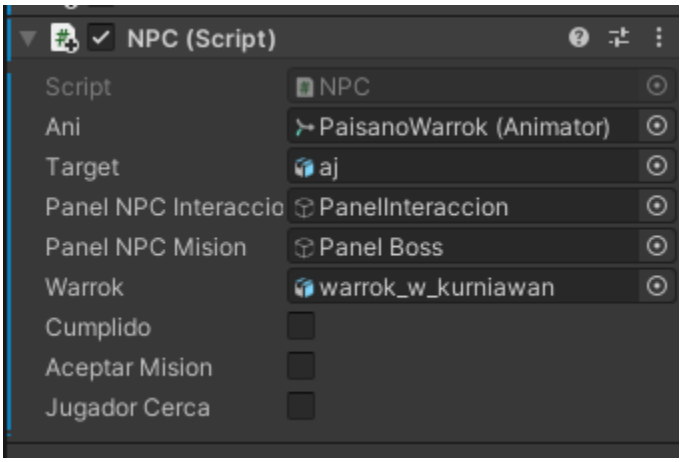
193. Componente capsule collider del personaje PaisanoAmigoTabernero.

- Sphere Collider:



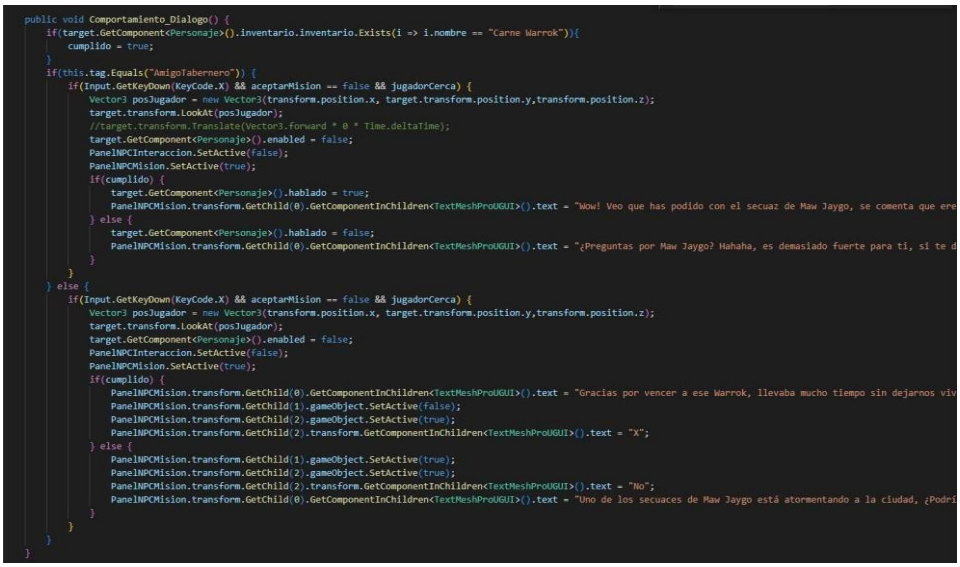
194. Componente Sphere collider del personaje PaisanoAmigoTabernero.

- NPC (Script):



195. Script desarrollado para comportamiento.

PaisanoAmigoTabernero. En este script se desarrolla el árbol de decisión del paisano, este paisano existirá desde el momento que se inicia el juego, pero si vamos a hablar con él antes de haber derrotado al Warrok no querrá hablar con nosotros ya que pensará que somos demasiado débiles y por tanto no quiere que oigamos hablar de Maw Jaygo, en cambio cuando le llevamos la carne de Warrok demostrándole que hemos derrotado al secuaz nos manda a hablar con el tabernero que está al lado.



```
public void Comportamiento_Dialogo() {
    if(target.GetComponent<Personaje>().inventario.inventario.Exists(i => i.nombre == "Carne Warrok")){
        cumplido = true;
    }
    if(this.tag.Equals("AmigoTabernero")) {
        if(Input.GetKeyDown(KeyCode.X) && aceptarMision == false && jugadorCerca) {
            Vector3 posJugador = new Vector3(transform.position.x, target.transform.position.y,transform.position.z);
            target.transform.LookAt(posJugador);
            //target.transform.Translate(Vector3.forward * 0 * Time.deltaTime);
            target.GetComponent<Personaje>().enabled = false;
            PanelNPCInteraccion.SetActive(false);
            PanelNPCMision.SetActive(true);
            if(cumplido) {
                target.GetComponent<Personaje>().hablado = true;
                PanelNPCMision.transform.GetChild(0).GetComponentInChildren<TextMeshProUGUI>().text = "Wow! Veo que has podido con el secuaz de Maw Jaygo, se comenta que ere
            } else {
                target.GetComponent<Personaje>().hablado = false;
                PanelNPCMision.transform.GetChild(0).GetComponentInChildren<TextMeshProUGUI>().text = "¿Preguntas por Maw Jaygo? Hahaha, es demasiado fuerte para ti, si te d
            }
        }
    } else {
        if(Input.GetKeyDown(KeyCode.X) && aceptarMision == false && jugadorCerca) {
            Vector3 posJugador = new Vector3(transform.position.x, target.transform.position.y,transform.position.z);
            target.transform.LookAt(posJugador);
            target.GetComponent<Personaje>().enabled = false;
            PanelNPCInteraccion.SetActive(false);
            PanelNPCMision.SetActive(true);
            if(cumplido) {
                PanelNPCMision.transform.GetChild(0).GetComponentInChildren<TextMeshProUGUI>().text = "Gracias por vencer a ese Warrok, llevaba mucho tiempo sin dejarnos viv
                PanelNPCMision.transform.GetChild(1).gameObject.SetActive(false);
                PanelNPCMision.transform.GetChild(2).gameObject.SetActive(true);
                PanelNPCMision.transform.GetChild(2).transform.GetComponentInChildren<TextMeshProUGUI>().text = "X";
            } else {
                PanelNPCMision.transform.GetChild(1).gameObject.SetActive(true);
                PanelNPCMision.transform.GetChild(2).gameObject.SetActive(true);
                PanelNPCMision.transform.GetChild(2).transform.GetComponentInChildren<TextMeshProUGUI>().text = "No";
                PanelNPCMision.transform.GetChild(0).GetComponentInChildren<TextMeshProUGUI>().text = "Uno de los secuaces de Maw Jaygo está atormentando a la ciudad, ¿Podri
            }
        }
    }
}
```

196. Script válido para ambos paisanos del juego. A modo de conclusión, podríamos decir que el ámbito de las IAs es un ámbito muy complejo y extenso a la vez que apasionante y que unido al ámbito de los videojuegos se pueden hacer cosas inimaginables, como dotar a un personaje no manejado por nadie de un comportamiento humano.
También me gustaría recalcar que es un tema bastante abierto en el cuál a mi modo de ver, una de los cosas más importante para la gente que trabaja en este sector es

Conclusión del desarrollo del trabajo:

la capacidad imaginativa para pensar en qué comportamientos se le puede dar a un personaje así como el lugar que tiene por ejemplo en mi caso dentro de una historia, ya que mediante el uso de las diferentes técnicas de IA mencionadas en el trabajo y las que ahí que no hemos mencionado se puede conseguir prácticamente cualquier tipo de NPC.
También decir que lo más complejo para mí en este trabajo ha sido el conseguir equilibrar la dificultad de los NPCs para no hacerlos ni muy difíciles de vencer ni muy fáciles y me parece que es algo bastante infravalorado a la hora de jugar a los videojuegos y que tiene una importancia muy grande en la jugabilidad a la vez que dificultad en el desarrollo.

Conclusiones:

En esta conclusión, quiero destacar que haciendo este trabajo he aprendido mucho sobre el ámbito del desarrollo de videojuegos que era algo impensable para mí ya que es algo que despertaba mucha curiosidad en mí de cómo se desarrollaban los videojuegos por lo que aparte de la dificultad y el esfuerzo que me ha supuesto este trabajo ha sido una experiencia bastante enriquecedora y gratificante ya que he podido en algunas cosas de mejor y en otras de peor manera solucionar todos los problemas que me he ido encontrando, así como el observar la cantidad de cosas que se pueden conseguir a través de las IAs como por ejemplo la utilización de estas para detección de enfermedades como puede ser el cáncer cosa que desconocía y me parece increíble.
También quiero destacar que la parte que más dificultad me ha generado ha sido la parte de la gestión del inventario.

Posibles mejoras:

En caso de tener más tiempo para poder seguir trabajando sobre este proyecto, me hubiera gustado haber podido hacer el personaje manejado por el humano mucho más completo ya que esto nos abriría un abanico mucho más amplio de posibilidades a la hora de las interacciones con el resto de NPCs y del mapa, que es otro de los puntos que me hubiera gustado poder tratar más, ya que la interacción de los personajes con el mapa en mi trabajo no es la más correcta posible y creo que sería uno de los puntos de mejora más importantes en el trabajo. Otra cosa para mejorar podría ser las interfaces gráficas como por ejemplo las celdas para equipar los objetos al personaje hacerlas de una manera mucho más correcta.

También me hubiera gustado haber podido refactorizar más el código separando ciertas partes por ejemplo en métodos aparte para poder reutilizar ese código y no tener que escribirlo una y otra vez.

Bibliografía:

Departamento de Ingeniería Telemática - UC3M.
http://www.it.uc3m.es/jvillena/irc/practicas/13-14/03.pdf

"Artificial Intelligence in video games Wikipedia".
https://en.wikipedia.org/wiki/Artificial_intelligence_in_video_games

"BUSQUEDA DE CAMINOS EN MAPAS ´ DE VIDEOJUEGOS".
https://e-archivo.uc3m.es/bitstream/handle/10016/23954/TFG_Alvaro_Parra_De_Miguel_2015.pdf?sequence=1&isAllowed=y

"Descripción general del algoritmo A-Star y su análisis de aplicaciones en el desarrollo de juegos - programador clic". programador clic.
https://programmerclick.com/article/64651780962/

"Unity: Desarrollo de una sencilla máquina de estados". Asociación de Estudiantes de Videojuegos.
https://aev.org.es/unity-desarrollo-de-una-sencilla-maquina-de-estados/

Don Pachi. Unity Tutorial de Misiones - Como hacer un sistema de dialogo con NPC. (25 de julio de 2020). [Video en línea]. Disponible:
https://www.youtube.com/watch?v=wY8PhF6xreQ

Don Pachi. Unity Tutorial - Como crear un sistema de misiones. (18 de julio de 2020). [Video en línea]. Disponible:
https://www.youtube.com/watch?v=zq70nwVHC8Y

JoexScript. Unity 3D - Enemigo básico (Rutinas y comportamientos). (9 de mayo de 2021). [Video en línea]. Disponible:
https://www.youtube.com/watch?v=_5pxcUykXcA

Xxlucastron741xX. Como crear un juego RPG o tipo Warcraft. (25 de junio de 2016). [Video en línea]. Disponible:
https://www.youtube.com/watch?v=ZoQJAlY487U

"Detección de distintos tipos de cáncer utilizando RNA".
https://riull.ull.es/xmlui/bitstream/handle/915/2905/Deteccion%20de%20distintos%20ti%20pos%20de%20cancer%20mediante%20Redes%20Neuronales%20Artificiales.pdf?seque%20nce=1&isAllowed=y

UNIVERSIDAD DE ALCÁLA
Escuela Politécnica Superior

Grado en Ingeniería de computadores.

Videojuegos, Inteligencia Artificial y Comportamiento: estudio, integración y demostración.

Autor: Alejandro Rilo Ferreiro
Tutor: Antonio Moratilla Ocaña