



Escuela
Politécnica
Superior

Futuro Imperfecto

Grado en Ingeniería Multimedia



Trabajo Fin de Grado

Autor:

David Segarra Rodríguez

Tutor/es:

Miguel Ángel Lozano



Universitat d'Alacant
Universidad de Alicante

Junio 2018

Contenido

1.	Justificación y Objetivos	3
2.	Dedicatoria.....	4
3.	Citas	5
4.	Índice de figuras	6
5.	Cuerpo del documento.....	8
5.1	Introducción	8
5.2	Estado del arte.....	8
5.3	Estudio de mercado.....	9
5.4	Objetivos	14
5.5	Diseño e implementación: GDD (Documento de desarrollo del videojuego) .	15
5.5.1	Presentación del documento	15
5.5.2	Introducción del juego	16
5.5.3	Mecánicas de juego	26
5.5.4	Niveles	30
5.5.5.	Interfaz.....	34
5.5.6.	IA	37
5.6	Metodología y planificación del proyecto	39
5.6.1	Resumen de la Fase 1	40
5.6.2	Resumen de la fase 2	41
5.6.3	Resumen de la fase 3	42
5.6.4	Resumen de la fase 4	43
5.7	Implementación	43
5.7.1	Arquitectura del juego general	53
5.7.2	Jugador	54
5.7.3	NPC	59
5.7.4	Elementos del escenario	67
5.7.5	Animaciones	68
6.	Conclusiones.....	71
7.	Bibliografía y referencias	72
8.	Anexos.....	74
8.1	Recursos.....	74

1. Justificación y Objetivos

El por qué hacer un videojuego como trabajo de fin de grado es un motivo simple, me gustan los juegos. Decidí que era la mejor opción ya que, me daba libertad a la hora de enfocar este proyecto y decidir cómo iba a ser el producto final. Por otra parte, también pondría a prueba mi capacidad de tomar decisiones para añadir, modificar o eliminar características de dicho proyecto, así como, poner a prueba todo lo aprendido en el grado. En cuanto al uso del motor gráfico de *Unreal Engine* ([Epic Games, s.f.](#)) fue porque vi que es una herramienta potente que te permite diseñar y crear juegos mucho más completos que en otros motores, tener un resultado vistoso y también se adaptaba mejor al tipo de juego que voy a hacer, un *Hack and slash*

Los objetivos que me quiero lograr son:

- Crear un producto final completo y robusto
- Que el producto final sea disfrutable como divertido para el jugador
- Mejorar todo lo aprendido durante el grado
- Aprender una nueva tecnología usada en el desarrollo de videojuegos (*Unreal Engine 4*)
- Desarrollar un proyecto de ingeniería completo

2. Dedicatoria

Quiero agradecer a mis padres, por hacer esto realidad y apoyarme durante todos estos años en el grado; también agradecérselo a mi tutor, Miguel Ángel, por haberme guiado y enseñado cómo seguir adelante y a mi novia Estíbaliz, también por apoyarme y animarme en los momentos duros, a mi senpai Alberto Salieto, un gran amigo y compañero del grado que me ha ayudado y enseñado muchas cosas a lo largo del curso y a mis profesores del Master de la CoCo School, Francisco Millan e Ignacio Sastre, que han proporcionado feedback sobre el proyecto y me han aconsejado.

3. Citas

«Si algo puede salir mal, probablemente saldrá mal»

Edward A. Murphy Jr.

4. Índice de figuras

<i>Figura 1: Ejemplo de QTE en God of War</i>	10
<i>Figura 2: Ejemplo de tienda de mejora del Devil May Cry 3</i>	11
<i>Figura 3: Ejemplo de mejora del arma del Devil May Cry 3</i>	11
<i>Figura 4: Ejemplo del Modo Katana del Metal Gear Rising</i>	12
<i>Figura 5: Ejemplo del Modo Katana del Metal Gear Rising</i>	13
<i>Figura 6: Ejemplo del tiempo bruja del Bayonetta</i>	13
<i>Figura 7: Calificación el juego</i>	18
<i>Figura 8: Ejemplo de diseño de nivel del Mirror's Edge</i>	19
<i>Figura 9: Ejemplo de diseño de nivel del Mirror's Edge</i>	20
<i>Figura 10: Ejemplo de arte visual del anime Psycho Pass</i>	21
<i>Figura 11: Ejemplo de arte visual del anime Psycho Pass</i>	21
<i>Figura 12: Jugador</i>	22
<i>Figura 13: Robot</i>	23
<i>Figura 14: Enemigo duro</i>	24
<i>Figura 15: Jefe</i>	25
<i>Figura 16: Controles de teclado</i>	28
<i>Figura 17: Controles de mando</i>	29
<i>Figura 18: Diseño del nivel tutorial</i>	31
<i>Figura 19: Diseño del nivel tutorial</i>	32
<i>Figura 20: Diseño del nivel 1</i>	32
<i>Figura 21: Diseño del nivel 1</i>	33
<i>Figura 22: Menú de juego</i>	34
<i>Figura 23: Menú controles</i>	34
<i>Figura 24: Pantalla de carga</i>	35
<i>Figura 25: Interfaz del jugador</i>	35
<i>Figura 26: Diagrama de los menus</i>	36
<i>Figura 27: Ejemplo de BP</i>	44
<i>Figura 28: Ejemplo de tipos de variables</i>	45
<i>Figura 29: Ejemplos de nodos de control</i>	46
<i>Figura 30: Ejemplo de Custom Event</i>	47
<i>Figura 31: Ejemplo de función</i>	48
<i>Figura 32: Ejemplo de función internamente</i>	49
<i>Figura 33: Ejemplo de programación en C++</i>	50
<i>Figura 34: Ejemplo de código C++</i>	51
<i>Figura 35: Diagrama de bloques del juego</i>	53
<i>Figura 36: Diagrama de clases del jugador</i>	54
<i>Figura 37: Diagrama de flujo del movimiento</i>	55
<i>Figura 38: Esquema sistema de combos jugador</i>	56
<i>Figura 39: Diagrama de clases de los NPC</i>	59
<i>Figura 40: Ejemplo del navmesh</i>	61
<i>Figura 41: Ejemplo de behavior tree</i>	62

<i>Figura 42: Esquema básico de la máquina de estados</i>	63
<i>Figura 43: Ejemplo de blackboard</i>	64
<i>Figura 44: Ejemplo de sensor de visión</i>	65
<i>Figura 45: Diagrama del sistema de combos enemigo</i>	66
<i>Figura 46: Diagrama de los elementos del escenario</i>	67
<i>Figura 47: Máquina de estados de las animaciones del jugador</i>	68
<i>Figura 48: Herramienta de retarget de animaciones</i>	70

5. Cuerpo del documento

5.1 Introducción

Durante todo el documento veremos cómo se desarrolla un videojuego completo, en este caso del género *Hack 'n' Slash* que se enfoca su jugabilidad en el combate cuerpo a cuerpo. Para este juego se utilizará lo último en tecnología como, por ejemplo, *Unreal Engine*. Este motor gráfico nos permitirá contar con grandes avances para facilitarnos la creación de juegos.

A continuación, veremos cómo se organiza este documento. Veremos los diferentes tipos de motor que hay en el mercado, qué juegos de este género han triunfado más o han aportado grandes cambios a este género, y para acabar, profundizaremos sobre cómo está creado este videojuego, desde lo que nos proporciona el motor hasta lo que se ha implementado; se irá indagando en cómo se ha ido diseñando este trabajo de ingeniería para crear un videojuego desde cero y cómo se estructura un proyecto desde su gestión hasta su producción y finalización pasando por distintas etapas de desarrollo.

5.2 Estado del arte

En el mundo del desarrollo de videojuegos cada vez se está usando más motores gráficos existentes más que, utilizar tus recursos en crear tu propio motor. Pocas empresas son las que hacen esto último ya que, se está compitiendo con, por ejemplo, *Unreal Engine* de *Epic Games* o *Unity* de *Unity Technologies* que tienen un equipo de desarrollo dedicado a mejorar el motor e implementar mejoras.

Muchas empresas ya desarrollaron su propio motor gráfico en su momento para su uso privado o comercial como *Frostbite* de EA, pero, al contrario que los motores anteriores,

la licencia de este es pago. Es cierto que, *Epic Games* y *Unity Technologies* reciben royalties si publicas un juego con su motor y ganas más de cierta cantidad establecida.

Hoy en día, muchas empresas usan más estos motores gratuitos para crear sus juegos ya sean por su versatilidad a la hora de crear un videojuego o por ser empresas que no tiene presupuesto para pagar licencias de otros motores privados. Al final, la tecnología más adecuada para desarrollar tu videojuego ya sea como empresa o como individuo, es la que más se adapta a lo que quieras crear. Es cierto que, puedes hacer cualquier juego con cualquier motor, pero, cada motor tiene unas especificaciones más enfocadas para crear un tipo de videojuego u otro. Hay que saber que motor nos puede beneficiar más para gestionar bien los recursos.

Sobre los motores anteriores, hay algunos que están enfocados, o adaptados, a otras plataformas. *Unity*, por ejemplo, está muy bien adaptado para crear juegos en móviles que *Unreal*, aunque se pueden crear juegos para esa plataforma. En cambio, *Unreal* está muy enfocado para juegos en PC o consola y que son en primera persona, pero, ya lleva muchos años siendo adaptado para poder hacer cualquier tipo de juego con este motor para poder ganar versatilidad y más usuarios. *Unity* también está enfocado al desarrollo de videojuegos para PC y hay muchos ejemplos de ellos y de distintos géneros. El motor de *Unity* es una herramienta muy potente, tanto como *Unreal*, si se sabe gestionar sus recursos y optimizarlos.

5.3 Estudio de mercado

Ver en que destacan los distintos juegos del género al cuál voy a enfocarme es importante para diseñar e implementar mejores mecánicas para atraer al jugador, es por eso, que hay que analizar bien a los distintos juegos para poder diferenciar al tuyo del resto.

Después de un estudio de los distintos juegos más icónicos que tiene este género he ido analizando lo que les diferenciaba de los demás. Hay muchos juegos de este género, como de los demás, y me he querido centrar en los más importantes para mi o que creo que más han influenciado más en el género o han conseguido destacar y perdurar más en el tiempo.

Empezamos por la saga de *God of War*. Esta saga de videojuegos nació en 2005 con su primer juego para la *Play Station 2* desarrollado por el *SCE Santa Monica Studio*. Lo que hizo destacar a este juego sobre sus competidores es un control del personaje muy gratificante y divertido como también la introducción de un nuevo concepto en los videojuegos: *los quick time event* o eventos donde el jugador tiene que pulsar una serie de botones para conseguir su objetivo. *Santa Monica* usó esto como una mecánica para ejecutar ataques definitivos sobre los jefes finales, enemigos duros y también los comunes para recompensar al jugador por más puntos.



Figura 1: Ejemplo de QTE en God of War

Continuamos con otra saga de videojuegos llamada *Devil May Cry* creada por *Capcom* en 2001 siendo una de las pioneras en sentar las nuevas bases de este género para la generación de esa época. En lo que destacaba era en un sistema de combos distinto al de los demás basado en el tiempo de pulsado entre botones, dependiendo del lapso ejecutabas distintos ataques. También tenía un sistema de puntuaciones por niveles y un sistema de mejora del personaje enfocado a las distintas armas del juego, creando así distintas maneras de personalizar tu estilo de juego.



Figura 2: Ejemplo de tienda de mejora del Devil May Cry 3



Figura 3: Ejemplo de mejora del arma del Devil May Cry 3

En las anteriores imágenes podemos ver cómo funciona el sistema de mejora del *Devil May Cry* donde podemos seleccionar el arma que queramos y mejorarle cualquier parámetro que tenga.

Seguimos con la desarrolladora de *Platinum Games*, quienes han hecho muchos juegos de este género que han sido muy importantes en el tiempo por tener unas mecánicas muy pulidas que responden muy bien a lo que hace el jugador. Entre estos juegos podemos destacar la saga de *Bayonetta* creada en 2009 que, como todos los juegos de este género hechos por ellos presentan un gran control del personaje, se diferencian del resto por tener una mecánica de que cuando se esquiva un ataque del enemigo, el tiempo se ralentiza para los enemigos permitiéndonos atacar más fácilmente. También tienen el *Metal Gear Rising* creado en 2013 que destaca en un gran control del personaje que se nos permite bloquear casi cualquier tipo de ataque permitiéndonos contraatacar y presenta una de las mecánicas más vistosas: “el modo *katana*” llamado así por los desarrolladores que consiste en tener total libertad a la hora de realizar cortes con tu katana permitiéndonos cortar por donde queramos y realizar ataques.



Figura 4: Ejemplo del Modo Katana del Metal Gear Rising

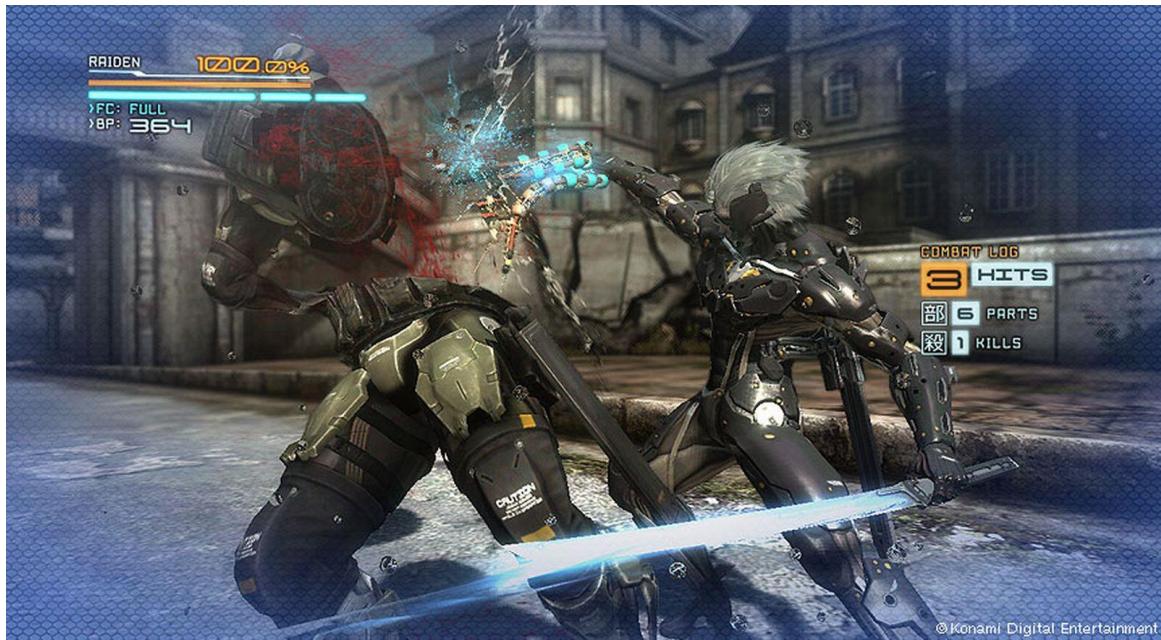


Figura 5: Ejemplo del Modo Katana del Metal Gear Rising

En estas imágenes podemos ver cómo funciona el modo katana del *Metal Gear Rising*. Puedes cortar en cualquier dirección y si cortas en el punto débil del enemigo puedes recuperar toda la vida con lo que le extraes.



Figura 6: Ejemplo del tiempo bruja del Bayonetta

En esta última imagen es del juego *Bayonetta* dónde se ve como se activa el *tiempo bruja* y el tiempo va más lento lo cual, esto te permite ejecutar más combos sobre un enemigo.

Una vez visto todos estos juegos que son los más icónicos de este género podemos ir viendo cuál de ellos ha sido los que más han funcionado o inspirado a otros juegos. A grandes rasgos, todos han aportado algo al género, unos más que otros, que les han hecho destacar y marcar tendencias que otros han seguido cómo: el sistema de mejora del personaje, *quick time events*, sistemas de combos avanzados, etc....

Otras de las demás mecánicas expuestas son más típicas y acotadas a esos juegos por cómo es el personaje principal y forman parte de sus características y de cómo es él que no tendría mucho sentido imitarlas en otro tipo de juegos, sino más bien, adaptarlas al juego en cuestión.

En resumen, todos los juegos nombrados anteriormente han aportado bastante a la industria que se han podido permitir sacar secuelas de ellos e incluso sacar juegos del mismo género.

5.4 Objetivos

Los objetivos que cumplir con este proyecto son:

- Crear un producto de calidad y que funcione.
- Crear un proyecto de ingeniería.
- Aprender a usar nuevas tecnologías que se usan en el mercado
- Mejorar las capacidades de programación y organización de un proyecto

- Mejorar las capacidades de programación de videojuegos

Objetivos del videojuego:

- Crear una arquitectura para el videojuego en *Unreal*.
- Implementar una IA acorde al estilo del videojuego.
- Implementar las mecánicas base del género.
- Integrar las animaciones correspondientes para cada actor del juego.
- Implementar una interfaz gráfica para gestionar el juego.
- Agregar los modelos correspondientes para el juego.

5.5 Diseño e implementación: GDD (Documento de desarrollo del videojuego)

5.5.1 Presentación del documento

En esta parte del documento veremos cómo se estructura un videojuego siguiendo la estructura del GDD (documento de desarrollo de un videojuego) que sigue una serie de pautas dónde hay que definir qué características presenta el videojuego, como se va a estructurar las mecánicas del jugador, que elementos va a tener la IA, cómo se va a mover

e interactuar el jugador con el escenario y los distintos elementos que lo componen: objetos, paredes, enemigos, etc.

También se describirá cómo serán los distintos enemigos, los menús que conformarán el juego y la interfaz gráfica que verá el jugador.

5.5.2 Introducción del juego

El videojuego **Futuro Imperfecto** se desarrollará usando el motor *Unreal Engine* y será para la plataforma PC. Aquí se irán especificando los distintos apartados que se implementarán en el juego y que sirva como guía para el desarrollo.

Futuro Imperfecto es un juego que está basado en distintos tipos de juego del mercado como, por ejemplo: *Bayonetta*, donde tendrá distintos elementos típicos de la saga; *Nier Automata*, dónde cogerá distintos elementos *hack 'n' slash*: el uso de combos cuerpo a cuerpo, el movimiento del personaje será en 3D donde podrá moverse libremente en un escenario, con límites establecidos, donde tendrá que ir avanzando para acabar el nivel. En algunos momentos del nivel, no todos, la cámara podrá cambiar de perspectiva para dar distintos enfoques.

5.5.2.1 Concepto del juego

Futuro Imperfecto es un videojuego donde controlamos a nuestro pobre protagonista que vive en una sociedad distópica dónde el gobierno ha creado una organización de asesinos que eliminan a los criminales. Un día, nuestro protagonista es fijado como un criminal sin haber realizado ningún acto de ese estilo, ¿será un fallo del sistema o habrá algo más?

5.5.2.2 Características principales

El juego se basa en estos apartados:

- **Mecánicas fluidas:** las mecánicas básicas del juego tienen que ser agradables para el jugador y que se sienta cómodo al saltar, disparar y realizar ataques cuerpo a cuerpo.
- **Frenético:** cuando el jugador ponga en práctica las mecánicas de combate, tiene que sentir que se sientan satisfactorias a la hora de luchar.

5.5.2.3 Género

Futuro Imperfecto es una mezcla de varios tipos de géneros:

- **Acción:** se pondrán a prueba la velocidad, el tiempo de reacción, la destreza y capacidad de leer los patrones de los enemigos. El jugador tendrá que usar los distintos elementos a su disposición para poder superar las distintas fases del juego.
- **RPG:** tendremos distintos poderes que conseguiremos al derrotar al jefe de cada zona que pasemos. Este poder está relacionado con el poder del jefe en cuestión.
- **Hack ‘n’ slash:** nuestro personaje dispone de un ataque cuerpo a cuerpo que podrá combinar para realizar distintos combos. Combo de ataques débiles o combos entre ataques débiles y fuertes.

5.5.2.4 Propósito y público objetivo

Este juego está enfocado a un público que le gusten los juegos de acción y aventuras con una jugabilidad fluida y entretenida.



Figura 7: Calificación el juego

El juego está calificado para mayores de 12 años por el tema de que contiene violencia, ya que, el jugador golpea a sus enemigos con una espada al igual que los enemigos también golpean al jugador

5.5.2.5 Estilo visual y personajes

El estilo visual que se ha pensado adoptar una temática futurista de estilo plano y simple. Esto significa que, comparado con otros estilos futuristas más cargados de detalles, el objetivo de este es hacer un estilo simple en cuanto a los componentes de los escenarios usando el blanco o colores fríos en los objetos.

Hay otros juegos dónde se usa este tipo de arte porque quieren dar un mensaje visual de limpieza y bienestar, aunque luego no sea así. En otros juegos se usan un estilo mucho más detallado y cargado para que se vea más espectacular.

Para este juego se han ido analizando y visto distintos estilos de arte de otros juegos hasta encontrar algo parecido a lo que se quería, uno de esos juegos es *Mirror's Edge*, juego de ambientación futurista que usa este tipo de diseño:



Figura 8: Ejemplo de diseño de nivel del *Mirror's Edge*

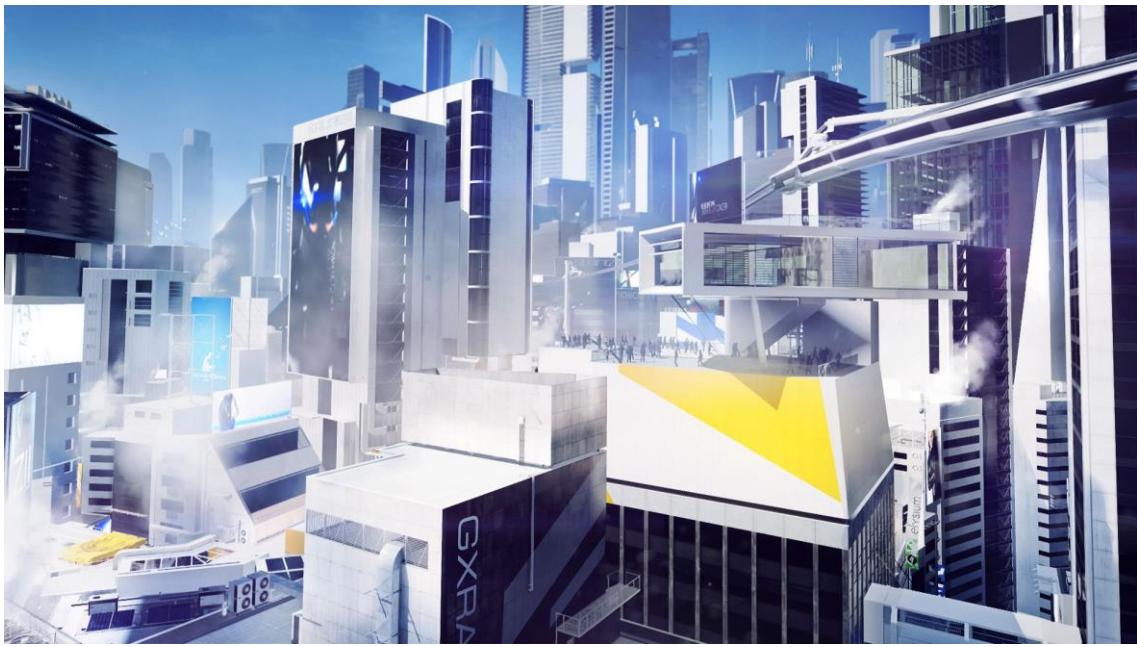


Figura 9: Ejemplo de diseño de nivel del Mirror's Edge

Como se muestran en estas dos imágenes, este es uno de los tipos de arte que se quiere lograr para los escenarios del juego y los distintos complementos de éste.

Otro del tipo de arte analizado es del anime *Psycho Pass*, ambientado en un futuro próximo, que hace poco le han sacado un juego estilo *visual novel*, pero, su arte y los edificios también es parte del estilo que se quiere captar en el videojuego, estilo liso y limpio con figuras rectas e industriales acompañados algunos de los edificios con bordes redondeados perfectos.



Figura 10: Ejemplo de arte visual del anime Psycho Pass

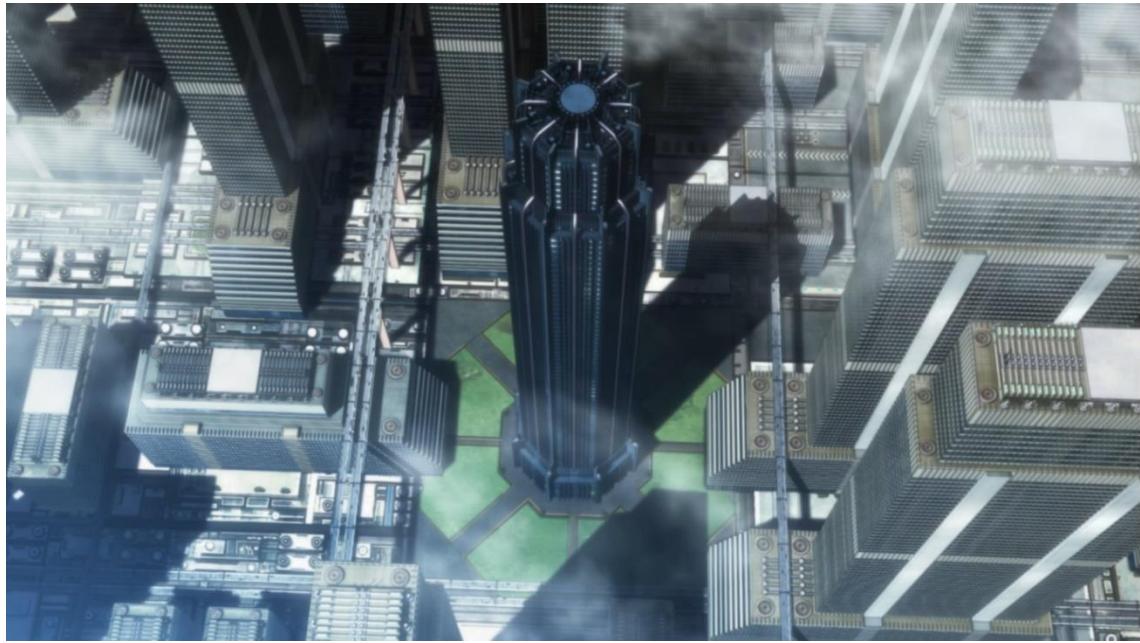


Figura 11: Ejemplo de arte visual del anime Psycho Pass

Como podemos ver en estas dos imágenes, este estilo también se adapta a lo que se pide del juego o cómo se quiere lograr que se vea el estilo visual del juego.

5.5.2.5.1 Jugador

Este es el personaje principal del juego. Tiene dos tipos de armas, una ligera y otra pesada, para causar daños a sus enemigos y que ejecutan distintos combos.

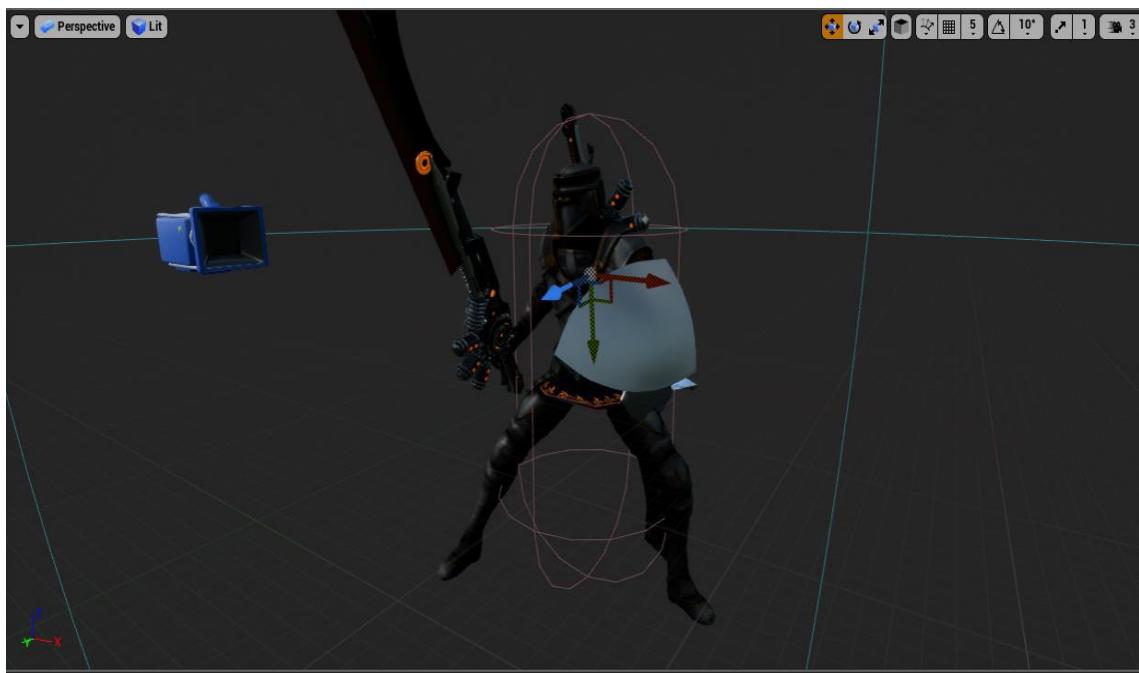


Figura 12: Jugador

5.5.2.5.2 Robot

El robot es el tipo de enemigo más común del juego. Puede realizar distintos combos y tiene una vida moderada.

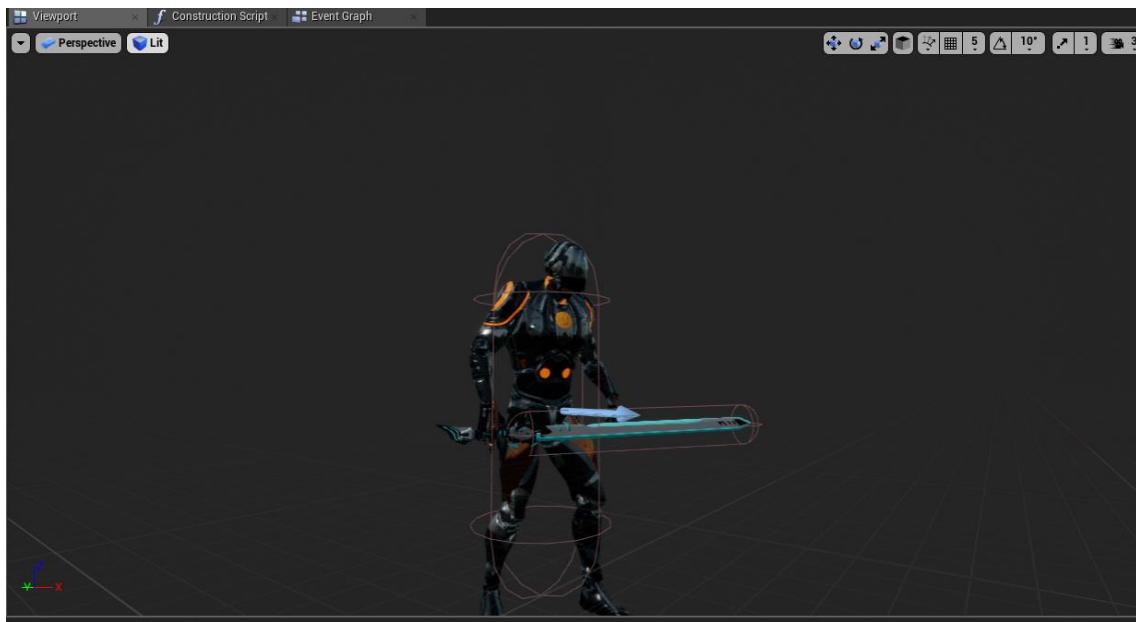


Figura 13: Robot

5.5.2.5.3 Enemigo duro

El enemigo duro es el que tiene más vida, a excepción del jefe final, aunque es el enemigo que menos abunda en el juego. Sale pocas veces debido a su alta vida y a sus ataques contundentes



Figura 14: Enemigo duro

5.5.2.5.4 Jefe

El jefe del nivel es el enemigo más fuerte. Tiene más ataques que los demás enemigos y mucha más vida que ellos.

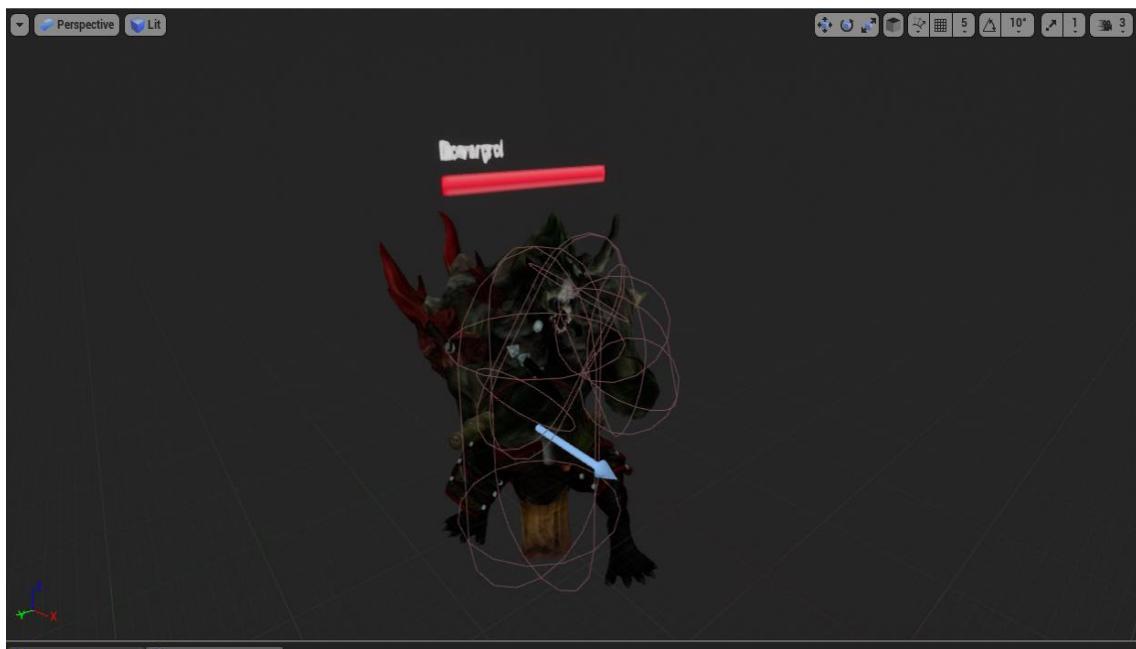


Figura 15: Jefe

5.5.2.6 Alcance

Este videojuego está enfocado a gente que quiera eliminar muchos enemigos y pasar un buen rato.

5.5.3 Mecánicas de juego

Aquí se irán especificando las distintas mecánicas que tendrá el juego y demás elementos relacionados a lo largo de los distintos apartados.

5.5.3.1 Jugabilidad

Mecánicas en 3D donde el jugador podrá moverse libremente por el espacio asignado, saltar, interactuar con los enemigos y objetos del escenario. La base de estas mecánicas será del tipo *hack 'n' slash*, donde el jugador tendrá una serie de ataques para derrotar a los enemigos.

El jugador también podrá cambiar de modo de ataque, posee un arma distancia que cambia el estilo de juego, de atacar a corta distancia a mayor. Este ataque tendrá sus propios combos distintos a los de cuerpo a cuerpo y el jugador podrá cambiar de estilo cuando más lo necesite. Ambos modos de juego poseen una barra que se llena, una vez esté completa, el jugador podrá relazar un ataque más potente de ese estilo que causará más daño a los enemigos. Una vez usa ese poder, tendrá que esperar unos segundos para volver a llenar la barra. La barra se va cargando con el uso de los combos, si cambias de estilo, perderás el progreso de esta barra. Si estás durante unos segundos sin hacer combos esta barra irá decreciendo, también si recibes daño de los enemigos.

5.5.3.2 Flujo del juego

El modo de juego principal es de un jugador, donde se nos irá contando una historia y tendremos que recorrer una serie de niveles semi-lineales (estos niveles no serán completamente una línea recta en lo relacionado al diseño de nivel, sino que, el jugador tendrá momentos donde habrá más caminos o lugares grandes donde hacer varias cosas cómo resolver algún pequeño *puzzle*)

El jugador dispondrá de todos los combos que puede hacer y no tendrá que desbloquear ninguno (es posible que se cambie según el desarrollo del juego) y las mejoras que podrá comprar o desbloquear serán de salud, energía del guante de magnetismo y/o mejoras de daño de ataque. Estas mejoras se podrán conseguir buscando por el escenario o comprándolas antes de elegir la misión que quieras hacer, este sistema nos permitirá ir mejorando a nuestro personaje para que sea más fuerte y nos sea más fácil el juego.

Para pasar de un nivel a otro, el jugador tendrá que buscar el teletransportador que le lleve a la zona del jefe final que estará protegido por un campo de energía que tendrá que desactivar buscando unas esferas de energía, excepto en el tutorial.

5.5.3.4 Movimiento y físicas

El movimiento del personaje no está regido por ninguna fuerza física sino por, velocidades; aunque si colisiona con un objeto al cual se le ha aplicado una fuerza, éste se verá afectado por ella.

Los enemigos son afectados por la fuerza magnética que crea el jugador para ser lanzados por los aires.

5.5.3.4.1 Interacción entre elementos

El jugador podrá interactuar con los distintos objetos que habrá en el escenario que se especificarán en este apartado:

- Podrá colisionar con distintos elementos del escenario ya sean paredes, los propios enemigos, cajas, elementos decorativos del escenario, puertas.
- Habrá objetos los cuales el jugador podrá interactuar de ellos de formas distintas. Hay puertas que podrá abrir las para pasar a otras zonas, paneles que permiten

acceso a zonas más importantes u objetos que podrá romper que tienen objetos consumibles o importantes para la historia.

- Otra mecánica de juego que habrá es un rayo magnético que atrae a los enemigos (que son robot de acero). El jugador podrá atraer a uno (o varios) enemigos para poder eliminarle o lanzarlos por los aires. Este rayo tiene una barra de energía que se irá cargando cuando se cojan baterías o mediante ataques a los enemigos. A los jefes de nivel o enemigos más fuertes, será más difícil poder atraerles o repelerles ya que, necesitan estar debilitados para poder usar el rayo sobre ellos.

5.5.3.4.2 Controles

Los controles están más pensados para jugar con mando, pero se adaptarán lo mejor posible al teclado y ratón.



Figura 16: Controles de teclado

GAMEPAD CONTROLS

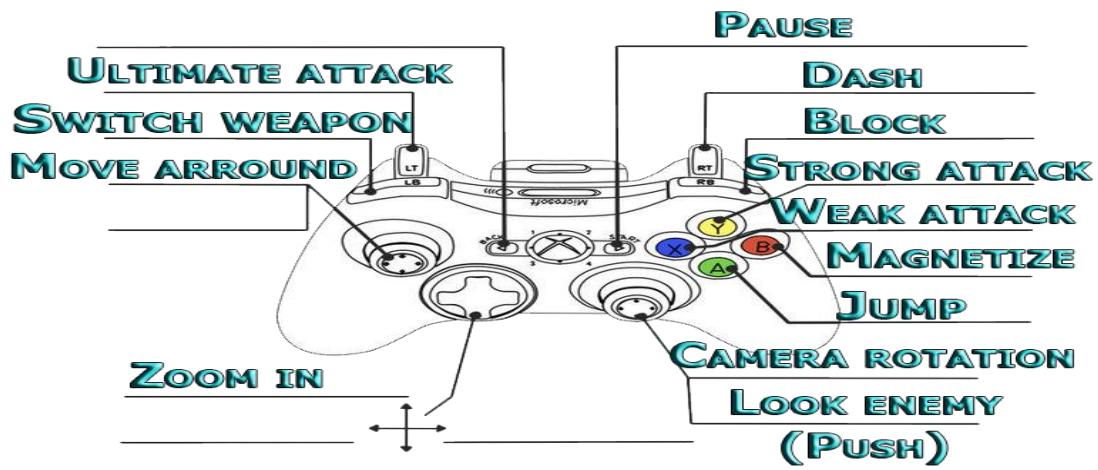


Figura 17: Controles de mando

5.5.3.5 Historia

Los eventos del juego ocurren en el futuro donde la ciudad donde transcurren los hechos está regida por una organización que eliminan a la gente suponen un peligro para ellos. Nuestro protagonista, sin saber por qué, es objetivo de esta organización que envía robots para matarle. Conforme avancemos en la historia veremos porqué ocurre esto y la razón por la que el protagonista va viendo que ha perdido fragmentos de sus recuerdos en momentos concretos de su vida donde ve que no cuadran bien.

La historia empieza con nuestro protagonista haciendo una misión para un equipo perteneciente a esa organización encargados de recopilar información sobre el objetivo a asesinar, aunque no todo iba a salir bien. Justo antes de acabar, es traicionado por su organización y no sabemos por qué, por eso, tendremos que escapar y buscar la razón por la cual somos objetivo de ellos.

5.5.4 Niveles

5.5.4.1 Diseño general del nivel

El esquema base que seguirán los niveles es: una zona principal, dónde el jugador empieza; distintas fases donde hay distintos tipos de enemigos en las que tendrá que ir avanzando; y, por último, la zona del jefe final dónde el jugador tendrá que enfrentarse a un enemigo más fuerte y ganarle.

5.5.4.2 Elementos del nivel

Los niveles tendrán distintos elementos, tanto visuales como objetos interactuables, para que cada uno se sienta algo distinto al anterior. También aparecerán los enemigos de maneras distinta a la de los demás niveles.

5.5.4.3 Diferentes diseños del nivel



Figura 18: Diseño del nivel tutorial



Figura 19: Diseño del nivel tutorial

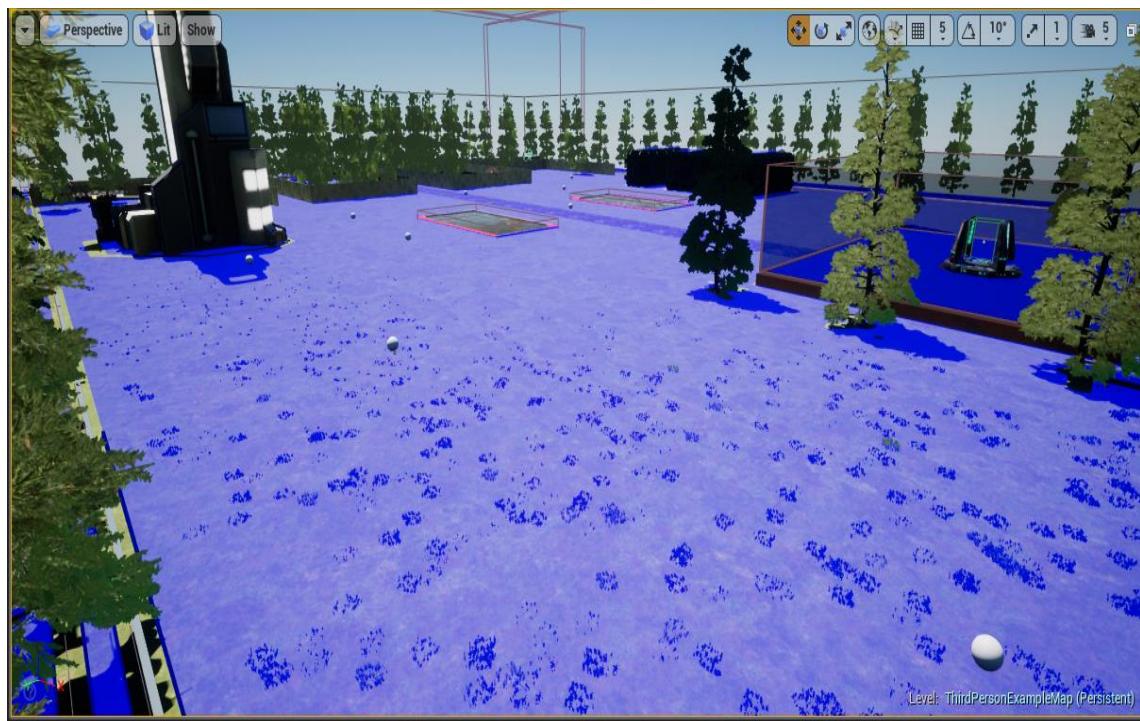


Figura 20: Diseño del nivel 1

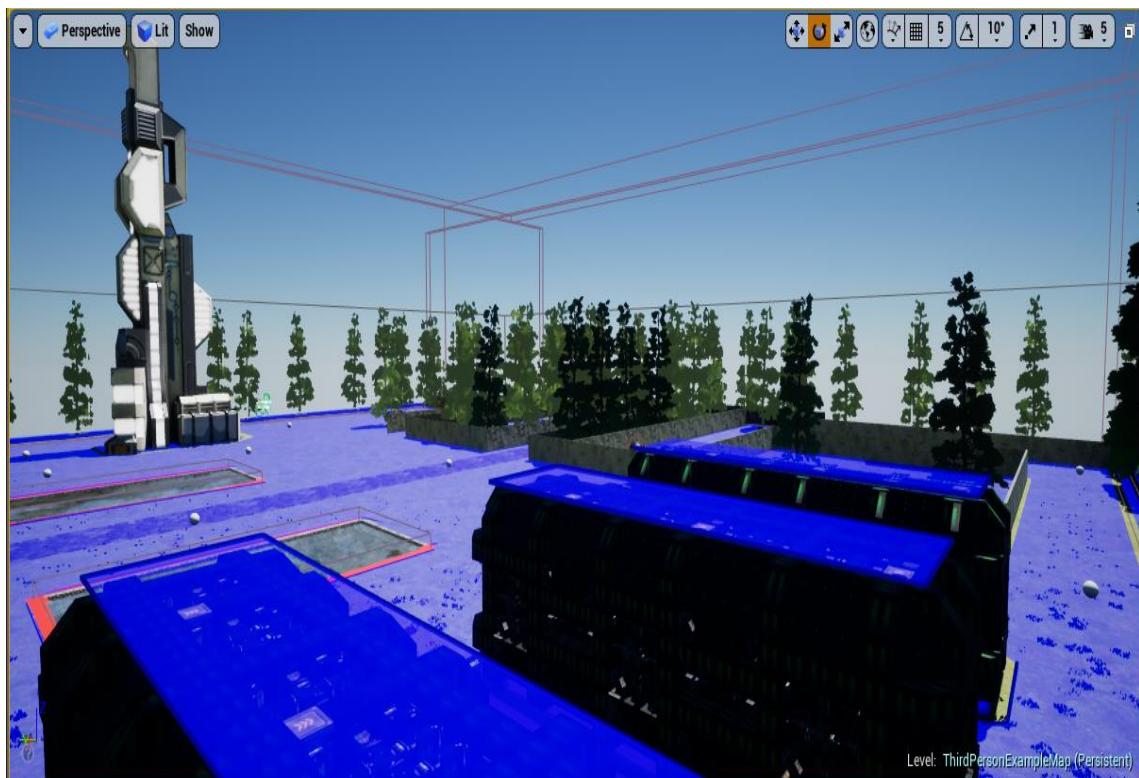


Figura 21: Diseño del nivel 1

5.5.5. Interfaz



Figura 22: Menú de juego

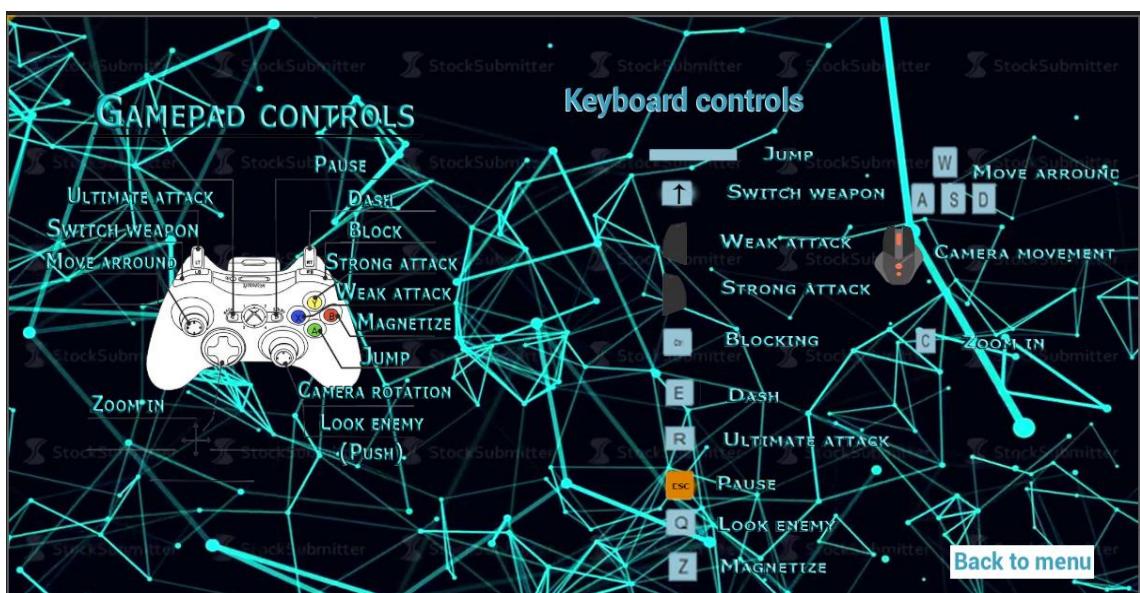


Figura 23: Menú controles



Figura 24: Pantalla de carga

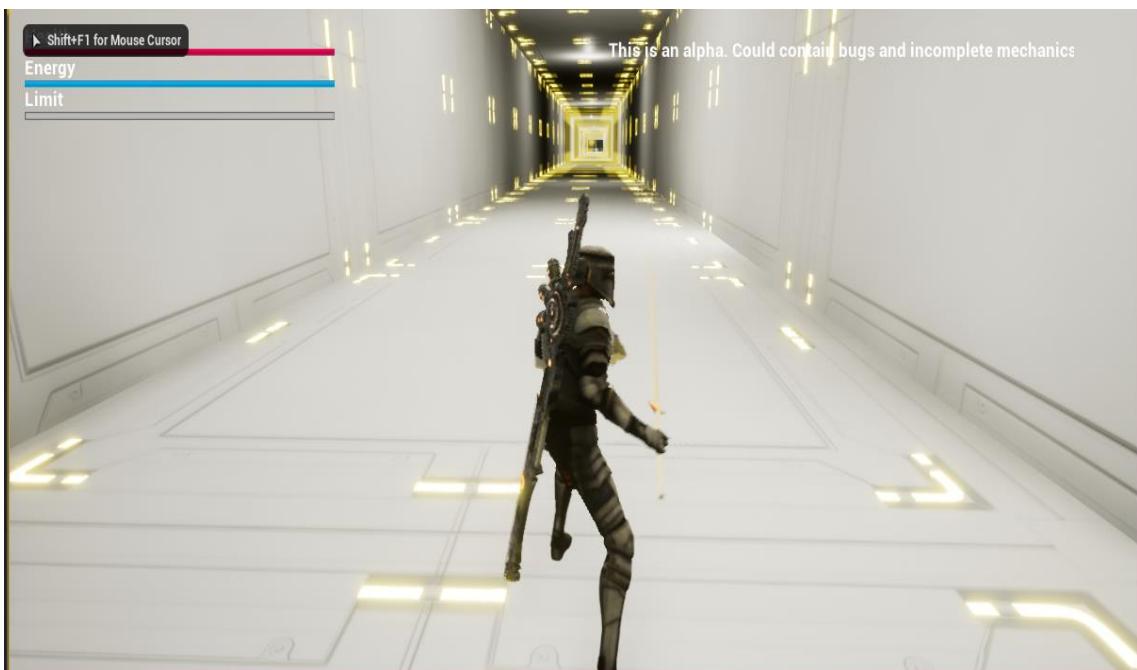


Figura 25: Interfaz del jugador

5.5.5.1 Diagrama de flujo

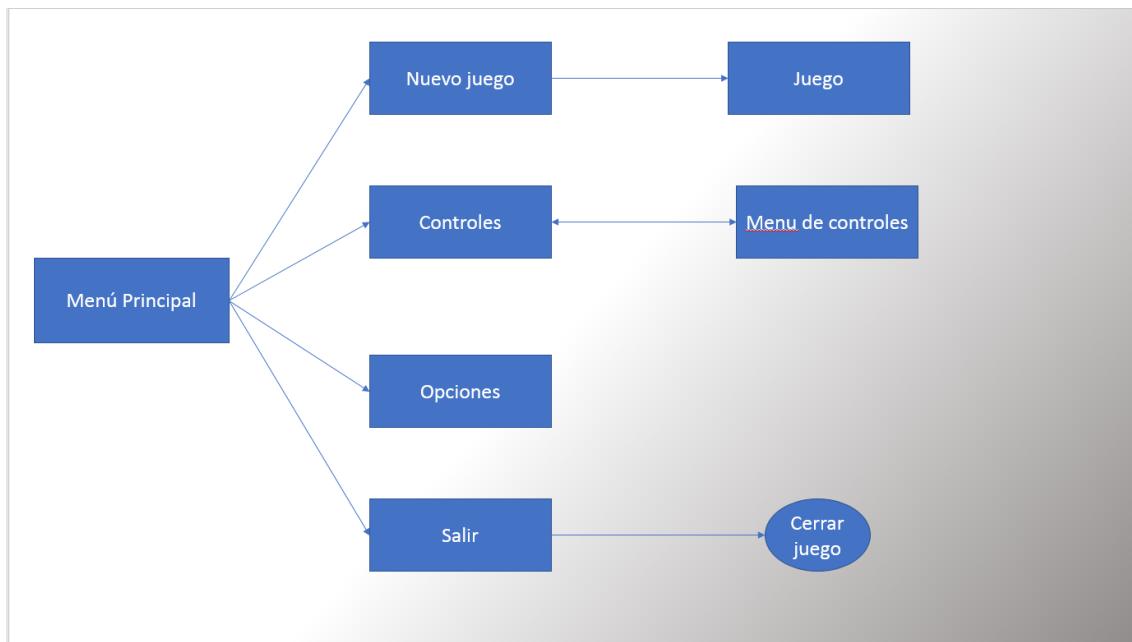


Figura 26: Diagrama de los menús

Desde el menú principal podemos empezar una partida, ver los controles o cerrar el juego y acabar la ejecución del programa. Las transiciones entre menús son instantáneas, pero, cuando se pasa de un nivel a otro hay una pantalla de carga o cuando se empieza un juego nuevo.

5.5.5.2 Cámara

La cámara será en 3º persona donde el jugador podrá rotar sobre el personaje y se irá ajustando, dependiendo de las condiciones del escenario, si está cerca de alguna pared se regula la distancia automáticamente. Si fija a algún enemigo la cámara dejará de ser de movimiento libre y se irá moviendo acorde al elemento fijado.

5.5.6. IA

La IA (Inteligencia Artificial) es uno de los puntos más importantes de este tipo de juegos ya que, tiene que ser desafiante pero entretenida a la vez.

Habrá distintos tipos de enemigos. Están divididos en 3 grandes grupos:

- **Enemigo básico:** Este enemigo será el que más abunde en el juego, en distintas variantes como por ejemplo un enemigo volador, otro normal o con un escudo... Su comportamiento es de patrullar una zona o estar quietos hasta que se encuentran con el jugador. Una vez le ha visto, éste va a por él usando distintos patrones de ataques. Estos ataques estarán sujetados a su visión, distancia del jugador y también a su propia vida.
- **Enemigo fuerte:** Al contrario del enemigo básico, este enemigo es mucho más fuerte e inteligente que él. Serán menos abundantes que estos y tendrán comportamientos más elaborados para que resulten más desafiantes al jugador. También habrá distintos tipos de enemigos: voladores, con espadas, a distancia...

Su set de movimientos estará mucho más completo que el del enemigo básico, un ejemplo es que si está a una distancia media hará un set de ataques distinto que a corta distancia; y si tiene menos vida podrá ejecutar distintos ataques que antes no podía y usarlos junto a los anteriores. Su movimiento por el escenario es muy parecido al de los enemigos normales, se irá moviendo por una ruta o zona hasta que vea al jugador o sea atacado por éste.

- **Enemigo jefe:** Cuando el jugador llegue al final del nivel, tendrá que enfrentarse a un jefe final donde pondrá a prueba todo lo que ha aprendido. Este jefe es mucho

más resistente y fuerte que los demás enemigos. Sus patrones serán más complejos que los otros enemigos e irán cambiando conforme a menos vida tenga. También puede que el enemigo sea más agresivo que en un principio o más precavido.

Sobre como los enemigos decidirán la distancia adecuada para realizar sus ataques y categorizar dicha distancia se usará la lógica difusa o algún algoritmo que permita realizar la estimación de la distancia más humana o real. Todas las IAs conocerán las limitaciones del escenario y por donde se pueden mover gracias a los algoritmos de *pathfinding* y el *navmesh* correspondiente que calcula las zonas transitables por donde se puede navegar.

Tabla resumen

	Enemigo Básico	Enemigo Fuerte	Jefe final
Sistema de movimiento básico (<i>pathfinding/navmesh</i>)	X	X	X
Sistema de ataque básico (basado en distancias)	X	X	X
Sistema de combos avanzado (basado en la vida restante)	-	X	X
Comportamiento básico	X	X	

Comportamiento avanzado (<i>Behavior tree</i>)			X
---	--	--	---

5.6 Metodología y planificación del proyecto

La planificación del desarrollo del proyecto seguirá una metodología ágil basa en iteraciones, las cuales, durarán entre 2 o 3 semanas, que forman las distintas fases repartidas a lo largo de los meses. Estas iteraciones tendrán objetivos primarios y secundarios dónde cada uno tiene más prioridad que otros. Si por algún motivo no se ha podido completar alguno de los objetivos, se deberá explicar el por qué no se ha conseguido y estudiar si se puede aplazar para otra iteración o ver si se puede eliminar del proyecto o simplificarlo si es posible.

Esta metodología es una de las mejores que se puede aplicar a un proyecto siendo sólo una persona ya que, plantearse objetivos a corto plazo para ver sus resultados es una buena manera de automotivación para seguir adelante con el proyecto y no abandonarlo.

En cuanto al software utilizado para realizar el juego se usará el motor gráfico de *Unreal Engine* en su versión 4.16. Se ha elegido este motor por su gran potencia y versatilidad a la hora de desarrollar todo tipo de juegos como también su gran impacto en el mercado, cada vez más y más juegos se crean con este motor debido a la calidad y fiabilidad del producto desarrollado.

Los modelos del juego serán creados usando el programa de modelado Maya. Es un programa muy usado en el mundo empresarial por su gran dinamismo y versatilidad a la hora de modelar cualquier tipo de objeto del mundo. Crea modelos de gran calidad y permite la exportación a cualquier tipo de motor gráfico de hoy en día.

Para controlar las distintas versiones de los documentos, versiones del juego y demás archivos del proyecto se usará la aplicación de *GitHub*, un software de control de

versiones que permite sincronizar código de manera más optima como también controlar que tipo de archivos se suben como también poder volver a versiones anteriores si ocurre algún error o se pierde información importante.

5.6.1 Resumen de la Fase 1

En la primera fase se propuso como objetivo final conseguir una demo jugable de las mecánicas básicas que iba a tener el juego que son:

- Programar las mecánicas básicas de movimiento
- Programar las mecánicas básicas de ataque
- Diseño de nivel básico
- Diseño básico de menús

Hay objetivos que se marcaron para esta fase que no se han podido cumplir y se han descartado crearlos para sustituirlos por otros gratuitos:

- Texturas de objetos
- Modelados
- Diseño de armas

5.6.2 Resumen de la fase 2

En esta segunda fase se propuso como objetivo principal completar una IA funcional cumpliendo ciertos requisitos que son:

- Movimiento de la IA
- Buscar y localizar al jugador
- Atacar al jugador
- Muerte
- Patrullaje
- Crear una Alpha del juego (una versión jugable con principio y fin)
- Diseñar el nivel tutorial y nivel final

Se retrasaron algunos de los distintos objetivos por falta de tiempo:

- Depurar mecánicas del jugador
- Depurar mecánicas del enemigo
- Modificar los niveles
- Mejorar los tutoriales

Otros objetivos se descartaron por falta de tiempo y/o no estaban relacionados con el objetivo principal del proyecto y se han sustituidos por elementos gratuitos:

- Animaciones de los personajes y enemigos
- Modelado de los personajes y enemigos
- Modelado de armas
- Modelado de elementos del escenario
- Texturas de los diferentes modelos

5.6.3 Resumen de la fase 3

En esta iteración el objetivo principal es completar las tareas retrasadas a lo largo del proyecto que fueran más críticas:

- Depurar mecánicas del jugador
- Depurar mecánicas del enemigo
- Cambiar el diseño de los niveles
- Buscar modelos de los elementos del escenario

5.6.4 Resumen de la fase 4

En la última iteración el objetivo principal es tener el juego ya completo y jugable, a falta de ultimar detalles del juego:

- Mejorar varios aspectos del juego
- Implementación de los menús definitivos
- Adaptar los menús al *gamepad*
- Mejorar el estado de “final de juego”
- Implementar botiquines

5.7 Implementación

Unreal tiene dos maneras diferenciadas de programar un juego. Una de ellas son los *Blueprints* (*nombrados BP para abreviar*) y la otra es C++ adaptado a su motor gráfico, de ambas formas puedes crear un videojuego y se hablará sobre sus características luego.

Los BP es una manera de programar gráfica donde se colocan nodos, que contienen líneas de programación en C++, que ejecutan una serie de acciones. Los BP son una herramienta muy potente que te permite implementar muchas mecánicas de forma rápida y ver su funcionamiento, también son muy usadas para prototipado rápido y de aprendizaje en el motor. Por otro lado, C++, permite crear funciones o características específicas que no existen en los BP y es una herramienta muy versátil; C++ es mucho más rápido a la hora de ejecutarse, por lo que, C++ es mucho más eficiente a la hora de que un juego sea sólido

y estable. A la hora de la verdad, los BP y C++ son usados a la vez, hay cosas que son más rápidas de implementar en un BP que en C++ y viceversa. La fusión de ambos permite tener un juego eficiente y estable para su disfrute.

Una vez introducido cómo se programa con *Unreal*, vamos a ver las características de cada uno. Primero veremos los BP y como son a través de imágenes. Veremos que variables tiene, que elementos podemos crear, cómo funciones o métodos; y como fluye la ejecución del programa.

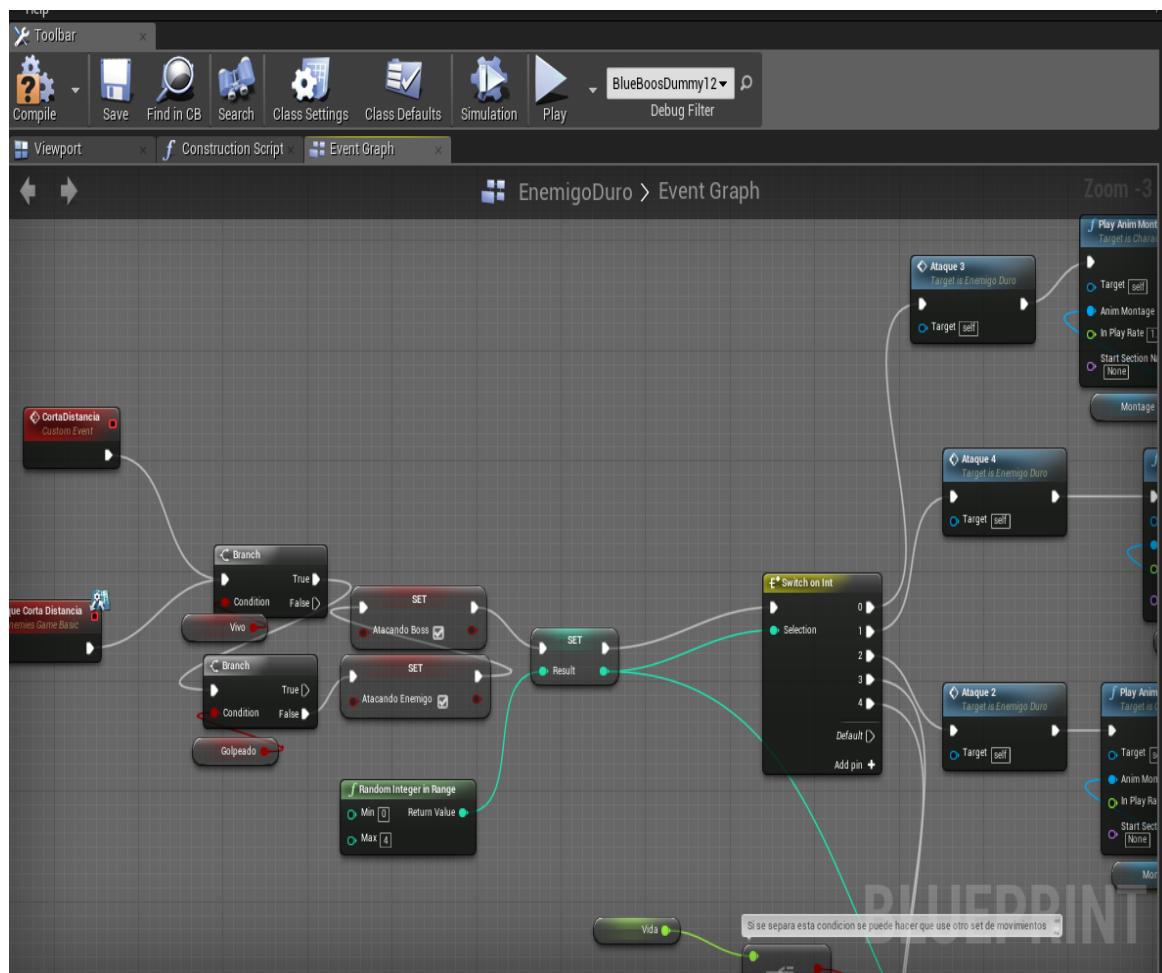


Figura 27: Ejemplo de BP

En los BP podemos crear nodos de distintos colores que pueden tener distintos parámetros de entrada y de salida como también hay distintos tipos de nodos que se usan para diferentes ocasiones como: nodos de control del flujo de la programación, nodos de funciones matemáticas, nodos de control y nodos de utilidades del propio motor. En la figura 27 podemos ver distintos tipos de nodos que se irán explicando. La manera de

conectar estos pines con otros nodos es uniéndolos mediante una línea, ambos nodos tienen que tener variables compatibles y/o una entrada de flujo de ejecución para poder funcionar. Se pueden ir viendo distintos tipos de nodos y de distintos colores que están interconectados entre sí. Hay algunos nodos que necesitan algún tipo de variable para que hagan su función o que estén conectados al flujo de ejecución o que sólo necesitan conectar su salida a otro nodo que reciba el flujo de la ejecución. Estos nodos suelen ser funciones matemáticas o variables para usar su valor.

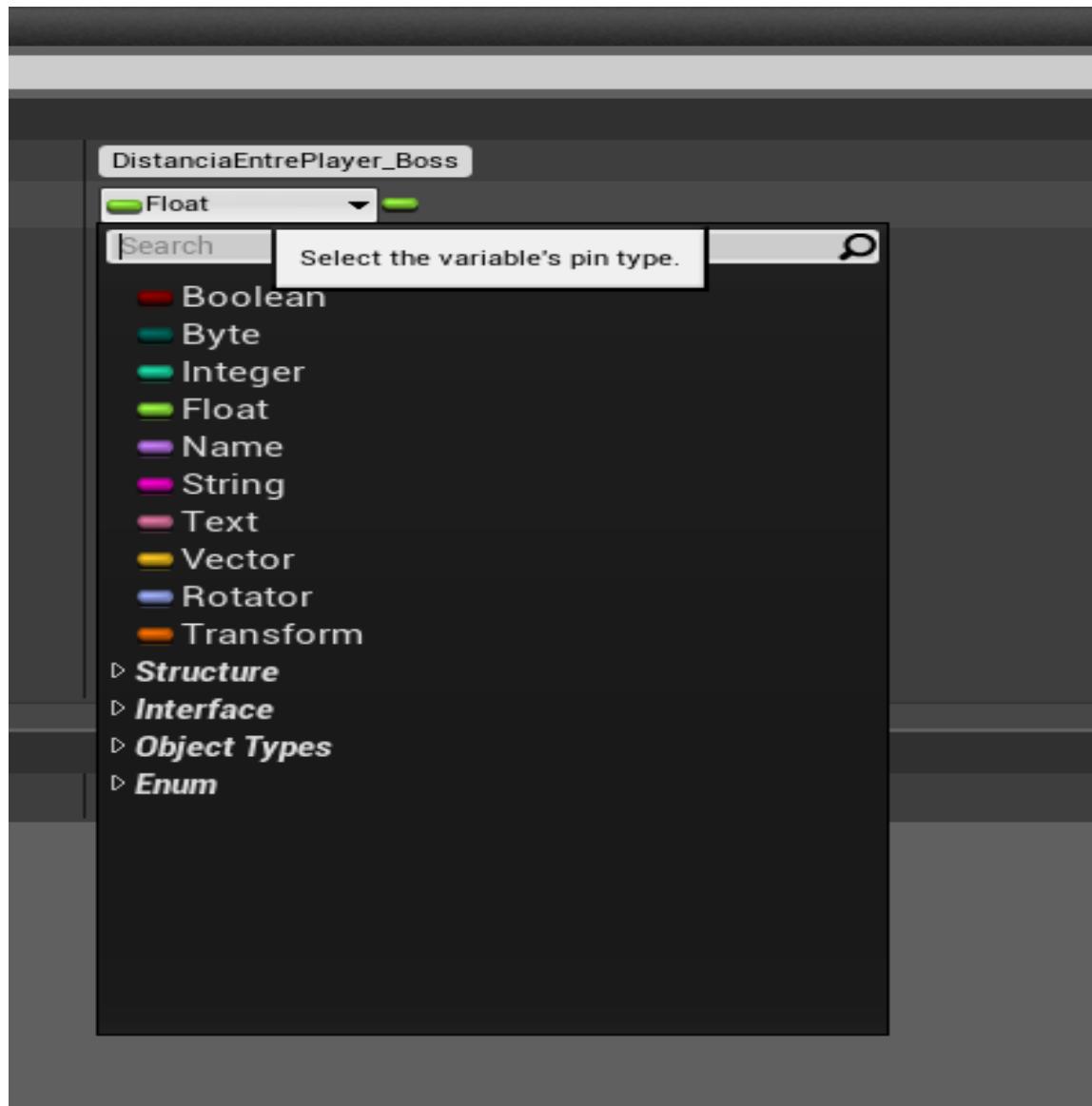


Figura 28: Ejemplo de tipos de variables

Para saber cómo funcionan los BP bien, primero tenemos que ver qué tipo de información necesitan para funcionar y son: las variables. Viendo en la figura 28 podemos ver los tipos de variable que se pueden crear en un BP, desde las primitivas, hasta más complejas como BP propios que tiene *Unreal* o que has creado tú u objetos que tienen su propia estructura. Cada tipo de variable tiene un color identificativo para que sea más fácil reconocerlas visualmente. En caso de las variables primitivas que se ven en la figura 28 son las más usadas, aunque, si es una variable más compleja como los objetos qué están en el grupo de “*Object Types*”, suelen ser de color azul y dependiendo de que atributo necesitemos de ese objeto, una referencia a éste o qué clase de objeto es, comparten el mismo color porque son variables de la misma gama.

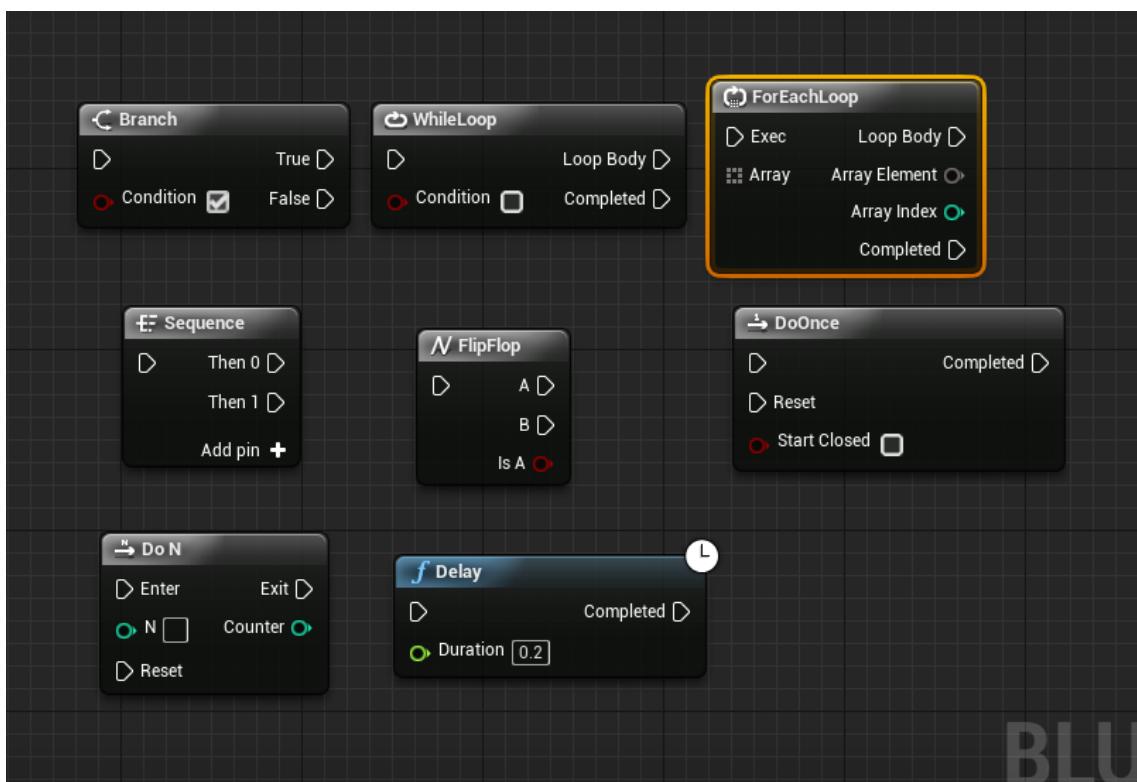


Figura 29: Ejemplos de nodos de control

Una vez visto qué son las variables, vamos a ir viendo el tipo de nodos que existen en *Unreal* como también los nodos especiales para crear métodos usados en nuestros BP. Unos de los tipos de nodos más importantes son los nodos de control y de flujo de ejecución. Estos nodos son encargados de controlar y gestionar como va a ir nuestro flujo de ejecución y como se van a gestionar nuestras variables, esto se debe a que, todo nodo en los BP, que no sea para realizar cálculos matemáticos, tiene un conector de color

blanco que indica el flujo de la ejecución y en muchos momentos, necesitaremos uno de estos nodos para poder controlar como queremos que vaya esta ejecución, podemos decir que este tipo de nodos son los más importantes porque son los más usados. Estos nodos son los típicos que se pueden encontrar en cualquier lenguaje de programación, condiciones, bucles y condiciones múltiples, aunque también, *Unreal* tiene nodos únicos que no están en otros motores como: nodos de única ejecución o de n-ejecuciones, siendo n el número máximo de veces que queremos que se ejecute una parte del código, alternadores (*flip flop*) o secuenciadores (figura 29).

Cada uno de estos nodos se usan para distintas situaciones, pero, todos se usan para controlar el flujo de la ejecución y controlarlo. Dependiendo de lo que se quiera hacer, algunos de estos nodos son más usados que otros.

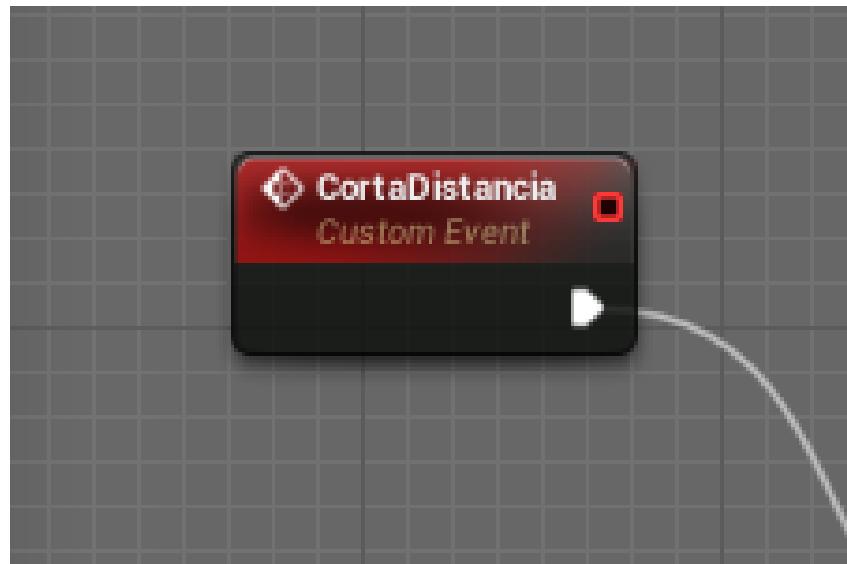


Figura 30: Ejemplo de Custom Event

A continuación, podemos ver un *custom event* (Figura 30), que es un elemento que tiene un funcionamiento parecido a una función en programación de objetos. Este elemento enlaza con los cálculos matemáticos o funcionalidades de nuestro BP y pueden ser llamados desde otros BP para que se ejecuten. Estos eventos pueden recibir variables, como se han visto en la Figura 27 dónde tendrá un pin de color correspondiente al tipo de variable que necesite, que pueden modificar otras variables del mismo BP o usarlas para guardar su información dentro de éste. Estos eventos serán ejecutados cuando se llamen

a su nodo correspondiente de llamada, que será de color azul, y entonces, podrán ejecutar todos los nodos a los cuales estén conectados. Para invocar estos eventos se necesita una referencia al objeto que lo contienen y después llamar a dicho evento.

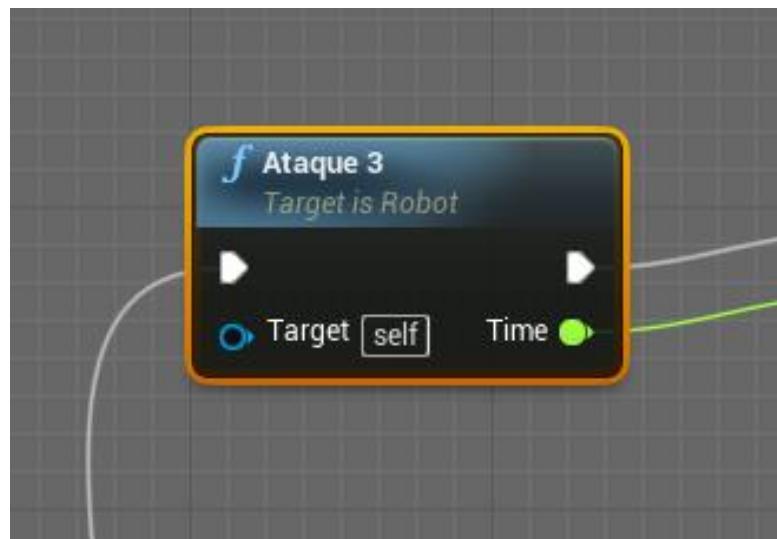


Figura 31: Ejemplo de función

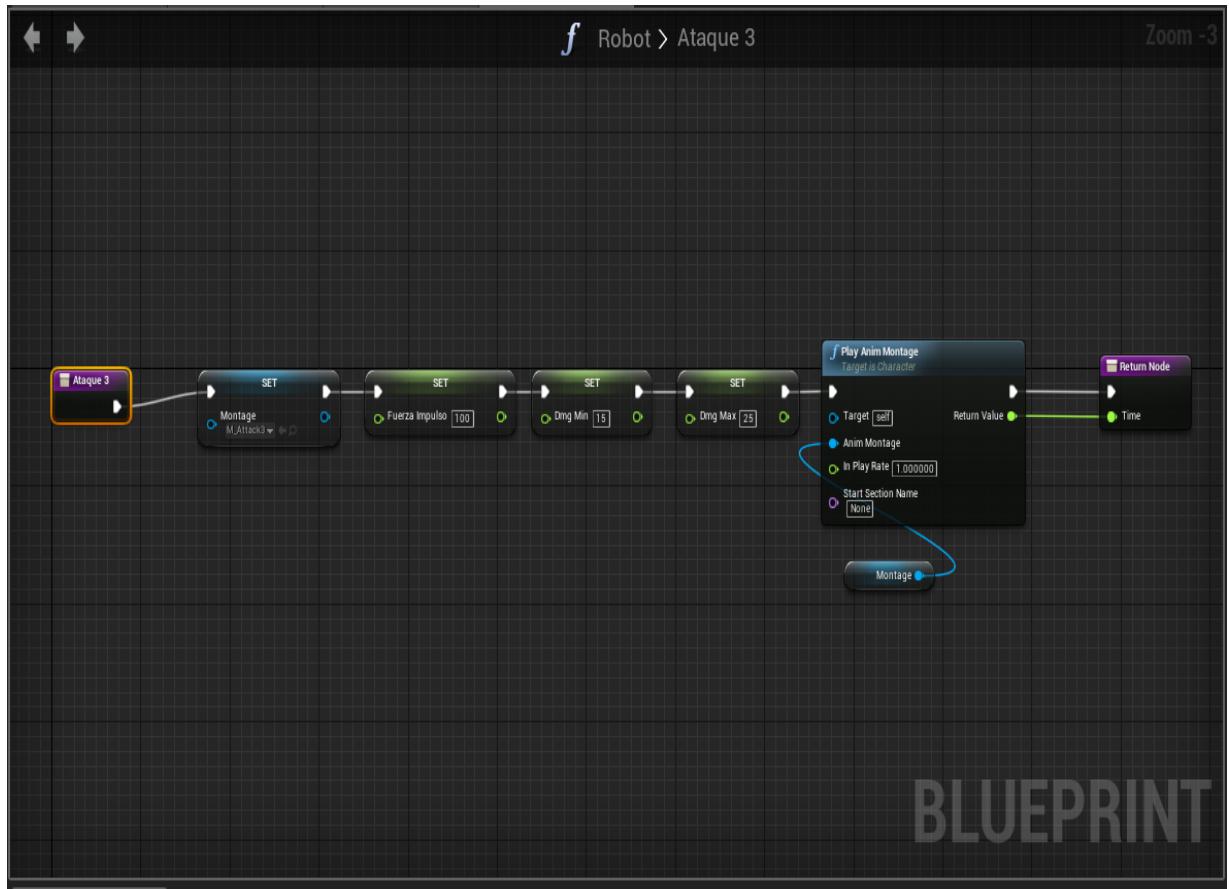


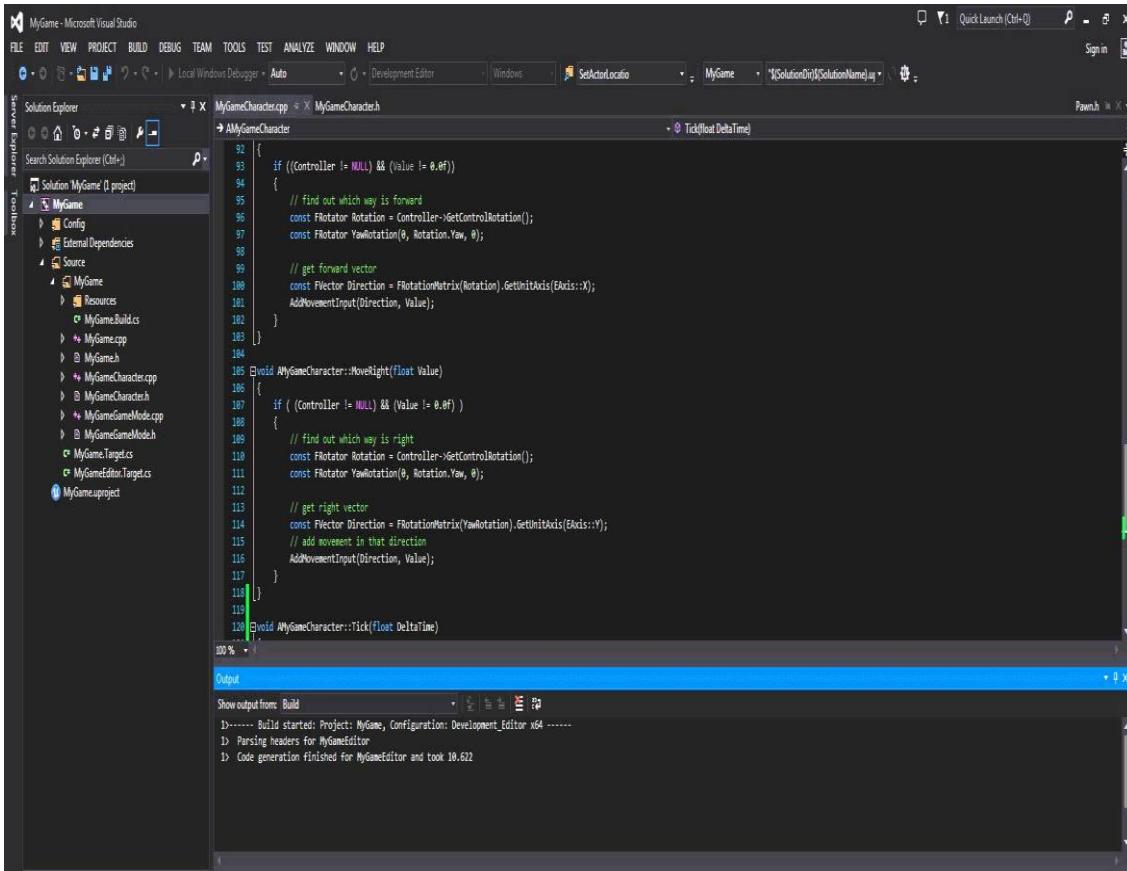
Figura 32: Ejemplo de función internamente

Las funciones en *Unreal* (figura 31) se suelen usar para gestionar cálculos internos del propio BP o funciones que ayudan a los *custom event* (figura 30) para realizar acciones específicas. Al igual que los *custom event*, las funciones pueden recibir cualquier tipo de variable, y también, pueden retornar variables para su posterior uso que se distinguen por el color del pin que tienen. Para llamar a estas funciones se pueden hacer de dos maneras, invocarlas llamando a su nodo de invocación (figura 31) o usando un nodo que puedes hacer que una función se ejecute en bucle cada cierto tiempo, también se puede para esa ejecución llamando al mismo nodo.

Estos son los elementos más importantes de un BP, todos los elementos anteriores comparten en común un output de color blanco que indica el flujo de la programación que se conecta a otro nodo que lo recibe ya que, ese pin especial puede ser tanto de entrada

como de salida. Algunos BP tiene sólo de salida o de entrada o tienen ambos. De esta manera visual se puede ver cómo va a ir el flujo del juego en ese determinado BP.

Una vez hemos descrito que es un BP, que elementos tiene y como funciona, pasamos a ver como es el C++ que ofrece *Unreal*, aunque en este proyecto no se ha hecho nada con C++ debido a que se suele usar para fases más finales del proyecto y en juegos de mayor tamaño con más gente involucrada.



```
MyGame - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ANALYZE WINDOW HELP
Local Windows Debugger Auto Development Editor Windows SetActorLocation MyGame \${SolutionDir}\${SolutionName}.up Pawn.h X
Sign in

Solution Explorer MyGameCharacter.cpp MyGameCharacter.h
Solution MyGame (1 project)
  MyGame
    > Config
    > External Dependencies
    > Source
      > MyGame
        > Resources
        > MyGameBuild.cs
      > MyGame.h
      > MyGameCharacter
      > MyGameCharacter.cpp
      > MyGameCharacter.h
      > MyGameGameMode
      > MyGameGameMode.h
      > MyGameTarget.cs
      > MyGameEditor.Target.cs
      MyGame.project
  100 %

MyGameCharacter.cpp
92 {
93     if ((Controller != NULL) && (Value != 0.0f))
94     {
95         // find out which way is forward
96         const Frotator Rotation = Controller->GetControlRotation();
97         const Frotator YawRotation(0, Rotation.Yaw, 0);
98
99         // get forward vector
100        const FVector Direction = RotationMatrix(Rotation).GetUnitAxis(EAxis::X);
101        AddMovementInput(Direction, Value);
102    }
103 }
104
105 void AMyGameCharacter::MoveRight(float Value)
106 {
107     if ((Controller != NULL) && (Value != 0.0f))
108     {
109         // find out which way is right
110         const Frotator Rotation = Controller->GetControlRotation();
111         const Frotator YawRotation(0, Rotation.Yaw, 0);
112
113         // get right vector
114         const FVector Direction = RotationMatrix(YawRotation).GetUnitAxis(EAxis::Y);
115         // add movement in that direction
116         AddMovementInput(Direction, Value);
117     }
118 }
119
120 void AMyGameCharacter::Tick(float DeltaTime)
121 {
122 }

Output
Show output from: Build
>----- Build started: Project: MyGame, Configuration: Development_Editor x64 -----
> Parsing headers for MyGameEditor
> Code generation finished for MyGameEditor and took 10.622
```

Figura 33: Ejemplo de programación en C++

```

→ AMyGameCharacter
92 {
93     if ((Controller != NULL) && (Value != 0.0f))
94     {
95         // find out which way is forward
96         const FRotator Rotation = Controller->GetControlRotation();
97         const FRotator YawRotation(0, Rotation.Yaw, 0);
98
99         // get forward vector
100        const FVector Direction = FRotationMatrix(Rotation).GetUnitAxis(EAxis::X);
101        AddMovementInput(Direction, Value);
102    }
103 }
104
105 void AMyGameCharacter::MoveRight(float Value)
106 {
107     if ( (Controller != NULL) && (Value != 0.0f) )
108     {
109         // find out which way is right
110         const FRotator Rotation = Controller->GetControlRotation();
111         const FRotator YawRotation(0, Rotation.Yaw, 0);
112
113         // get right vector
114         const FVector Direction = FRotationMatrix(YawRotation).GetUnitAxis(EAxis::Y);
115         // add movement in that direction
116         AddMovementInput(Direction, Value);
117     }
118 }
119
120 void AMyGameCharacter::Tick(float DeltaTime)
...

```

Figura 34: Ejemplo de código C++

Una vez hablado de los BP, vamos a ver cómo va la programación en C++ y como está estructurado, en la figura 34 podemos ver la estructura de la clase de jugador sus métodos de movimiento. Al igual que los BP, en C++ puedes crearte cualquier tipo de variable, ya sea primitiva o de objetos propios de *Unreal* o tuyos, como también crearte funciones que hagan una serie de cálculos o ejecuten instrucciones para cualquier elemento del juego y retornen datos o estructuras que se usarán en otras partes del juego. Se puede ver como está estructurado en C++ el movimiento de la clase jugador y como se calculan esos vectores de movimiento. La manera con la que se enlaza este método con todo el juego es igual que con los BP, lo único que cambia es el modo de programar el juego.

Estas son las dos maneras distintas de programar en *Unreal*. Se ha visto más en profundidad los BP porque son la estructura principal del proyecto y se han trabajado más con ellos que con C++. En los siguientes apartados se irá viendo, por partes, cómo se ha estructurado el juego en distintas categorías viendo lo que se ha usado del motor como lo que se ha implementado.

5.7.1 Arquitectura del juego general

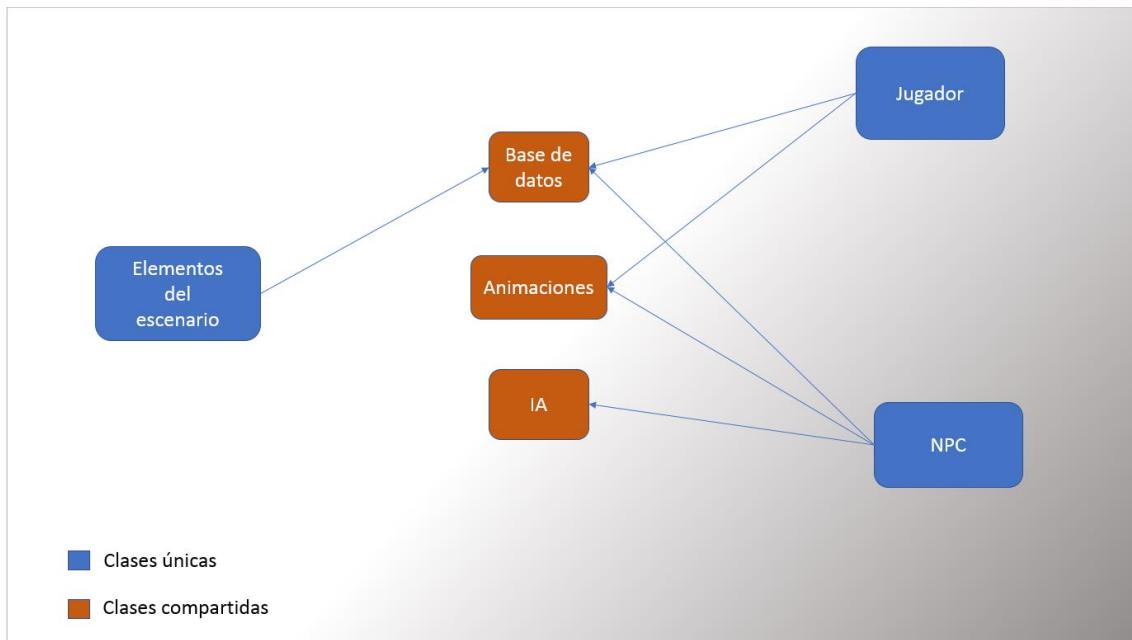


Figura 35: Diagrama de bloques del juego

El juego se compone de distintos elementos que se relacionan entre ellos y tienen guardados distintos elementos para ser usados dentro del juego. Podemos diferenciar dos tipos de bloques que comparten información básica que luego se usan en el juego, más adelante se irá profundizando en los distintos elementos que forman el juego con su diagrama explicativo.

La base de datos corresponde a toda la información de variables que guardan las distintas clases que componen el juego. Esto se refiere a que, cada clase se guarda distinta información que alguna es compartida con otras clases, las variables que necesiten otras clases se accederán a ellas mediante métodos que devuelvan ese valor, nunca se modifica esa variable desde otra clase. Dentro de esta base sólo se dejan las variables de las cuales las demás clases necesiten información de ellas.

5.7.2 Jugador

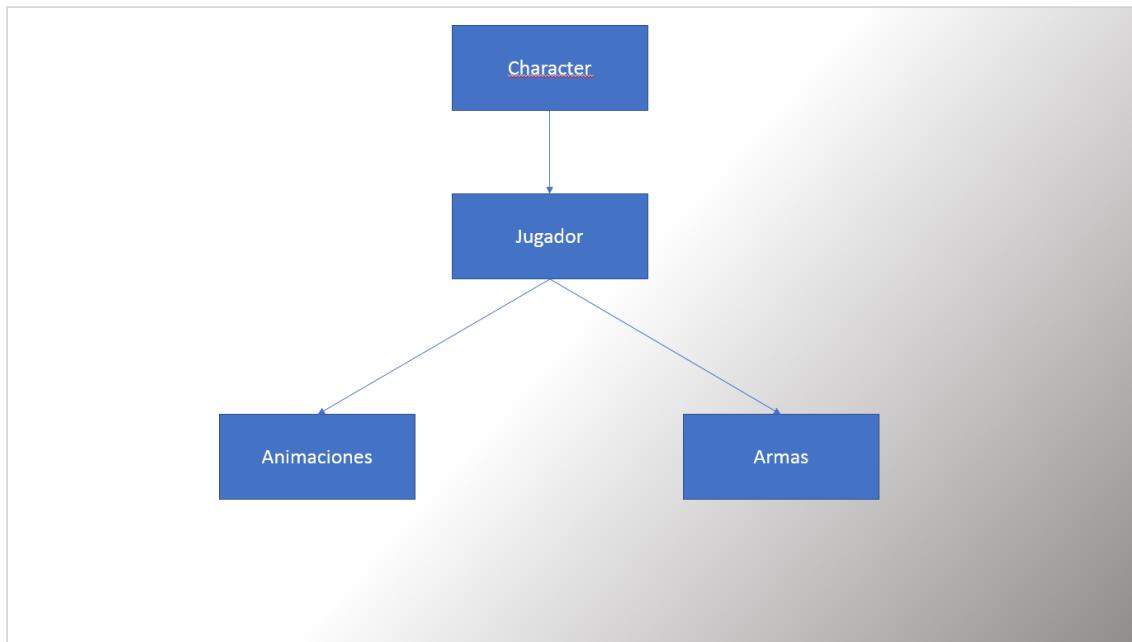


Figura 36: Diagrama de clases del jugador

En este apartado vamos a centrarnos sobre cómo se ha implementado el jugador y como se ha estructurado su jerarquía de herencia adaptándolo al motor. Se irá hablando en orden de cómo se representa la figura 36, de arriba abajo y de izquierda derecha.

Al ser nuestro jugador un personaje que se va a mover mediante velocidades y a interactuar con los distintos elementos del escenario, ya sean paredes, enemigos, objetos, etc. Se ha hecho que herede de una clase prediseñada por el motor que provee de distintos elementos para movernos por el mapa como una cámara, un control donde podemos especificar la velocidad de nuestro personaje cuando anda, salta o se agacha; y así como, un elemento para gestionar las colisiones con el entorno del nivel. Esta clase, también los da la oportunidad de añadirle que maya vamos a mover y cómo se va a gestionar sus animaciones que se hablarán de cómo van y su gestión más adelante.

Una vez visto como se gestiona el padre de nuestro jugador, vamos a ver cómo se ha organizado y estructurado el BP del jugador. En este BP está todo lo que es lógica del juego, en otras palabras, todo lo que tiene que ver con que el personaje pueda moverse, interactuar e información y datos sobre él.

El BP del jugador se ha sido creado totalmente y no se ha usado ningún elemento ya creado por *Unreal*, es verdad que la clase padre tiene elementos que facilitan el movimiento de un personaje por el mapa, se tiene que implementar los eventos que gestionan esos movimientos. Los eventos están adaptados para teclado y ratón para que se puedan usar ambos. Cuando un evento de movimiento se lanza, se comprueba cual de todos los tipos de eventos pueden ser, si se quiere avanzar hacia delante, o hacia atrás, se calcula el vector de la rotación asociado a ese movimiento y se le aplica al jugador. Este vector suele ser el que apunta hacia adelante desde la posición de la cámara. Por otro lado, si se quiere girar se hace exactamente lo mismo excepto que el vector que se calcula es el que forma noventa grados con el vector anterior. De esta manera el jugador puede rotar para girar hacia la derecha o izquierda.

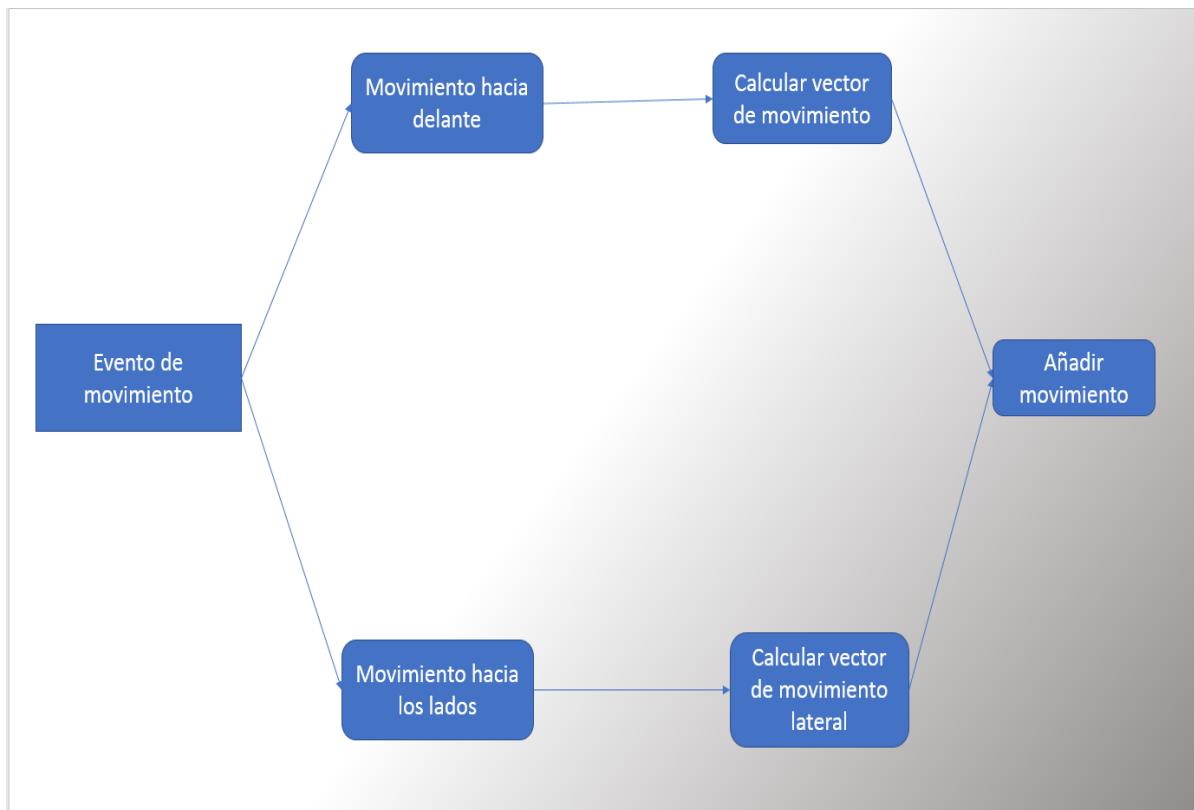


Figura 37: Diagrama de flujo del movimiento

Otro punto importante es que el sistema de ataque del jugador está dentro de este BP, que se gestiona con la cantidad de veces que se ha pulsado uno de los botones de ataque, cómo se puede conectar ataques débiles con fuertes, se puede cortar el combo de los ataques

débiles, aunque no al revés. Este sistema detecta, en el momento de cambio al siguiente ataque, si se ha vuelto a pulsar alguno de los botones de ataque. Si es así, realiza un cambio de ataque hasta que ya no queden más combos por hacer. Cuando los combos se han acabado se reinician todas la variables de control que usa este sistema para poder ejecutar los combos desde el principio. Este sistema es el mismo para los ataques fuertes, lo único que cambia es el evento que recibe.

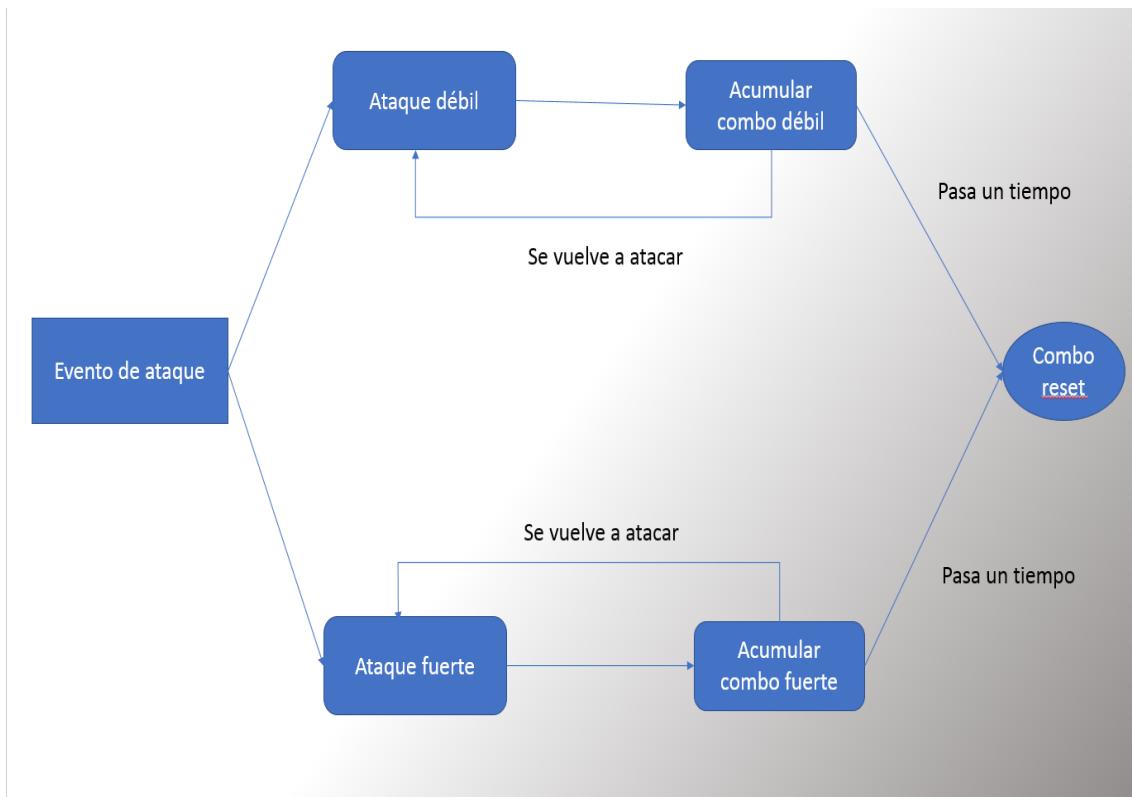


Figura 38: Esquema sistema de combos jugador

Nuestro personaje puede realizar distintas acciones como: esquivar, bloquear, cambiar de arma, fijar al objetivo o usar el magnetismo. Cada una de estas acciones han sido creadas usando distintos elementos que tiene *Unreal* como fuerzas físicas y funciones matemáticas de cálculos de vectores.

Siguiendo el orden anterior, cuando el jugador quiere esquivar, en una de las direcciones posibles, se gestiona si el evento es el que activa esta función del personaje, si es así, se calcula en qué dirección tiene que ir el jugador aprovechando la dirección de movimiento hacia donde se había movido antes y se aplica una fuerza en esa dirección.

Cuando el jugador quiera bloquear algún ataque, si el evento correspondiente corresponde a bloquear y se mantiene en el tiempo, el jugador cambiará de pose y empezará a bloquear el daño que reciba hasta que no pueda más. Mientras bloquea puede moverse al igual que antes, aunque a menor velocidad.

El cambio de arma se gestiona cuando el jugador recibe el evento asociado a esto, si se corresponde, éste se quedará quieto hasta que se cambia de arma. El cambio de arma afecta al tipo de combos que se podrá realizar, aunque el sistema es el mismo explicado anteriormente.

El jugador puede fijar objetivos dentro de un radio de visión que tiene. Cuando se llama a este evento, si tiene algún enemigo cerca suya, si hay más de uno fija al más cercano. Estos cálculos se realizan usando las posiciones del jugador y los enemigos dentro del radio de acción calculado la distancia a la que se encuentra de cada uno de ellos. Si por algún casual se pierde la visión del enemigo fijado, éste deja de fijarse, lo mismo pasa si el enemigo muere.

Por último, está el magnetismo que puede atraer a algunos tipos de enemigos. Si se activa este evento, la cámara se acercará al hombro del jugador dónde podrá apuntar a los enemigos que vea y los podrá atraer hacia él. La manera de atraer a los enemigos es calculando el vector que va desde el enemigo hacia el jugador y, de esta manera, aplicar una fuerza en esa dirección. Después de que pase un tiempo, el enemigo saldrá disparado en sentido contrario. El magnetismo también se puede usar junto al fijar objetivo, si se tiene a un enemigo fijado se puede usar el magnetismo con él sin tener que apuntar.

El evento de gestión de daño del jugador usa un elemento de *Unreal* que te ayuda gestionar el daño que se le puede aplicar a los distintos actores que crees o existan. Este evento se llama cuando el jugador recibe daño que no causa el mismo y se calcula restándole a su vida la cantidad de daño que recibe. Si la vida baja a más de cero, el jugador muere y se acaba el juego.

Otra de las partes importantes que tiene el jugador es que, está formado por animaciones, las cuales se ejecutan y se controlan con un BP caracterizado para ello y qué se referencia en el BP del jugador. Sobre este se hablará más en profundidad más adelante ya que, todos los actores que heredan del BP *character* usan este tipo de BP y el concepto de desarrollo es muy parecido.

El BP de jugador tiene la información referente a las armas que lleva. Esta información consta de que malla va a usar y del daño que puede causar esta arma que, varia dependiendo del tipo de arma. Las armas tienen su propio sistema de colisiones para gestionar a quién le hacen daño y, dependiendo de quién sea, se le aplique este daño si cuando el jugador realiza un ataque y el arma colisiona con algún elemento al que se le puede aplicar daño.

5.7.3 NPC

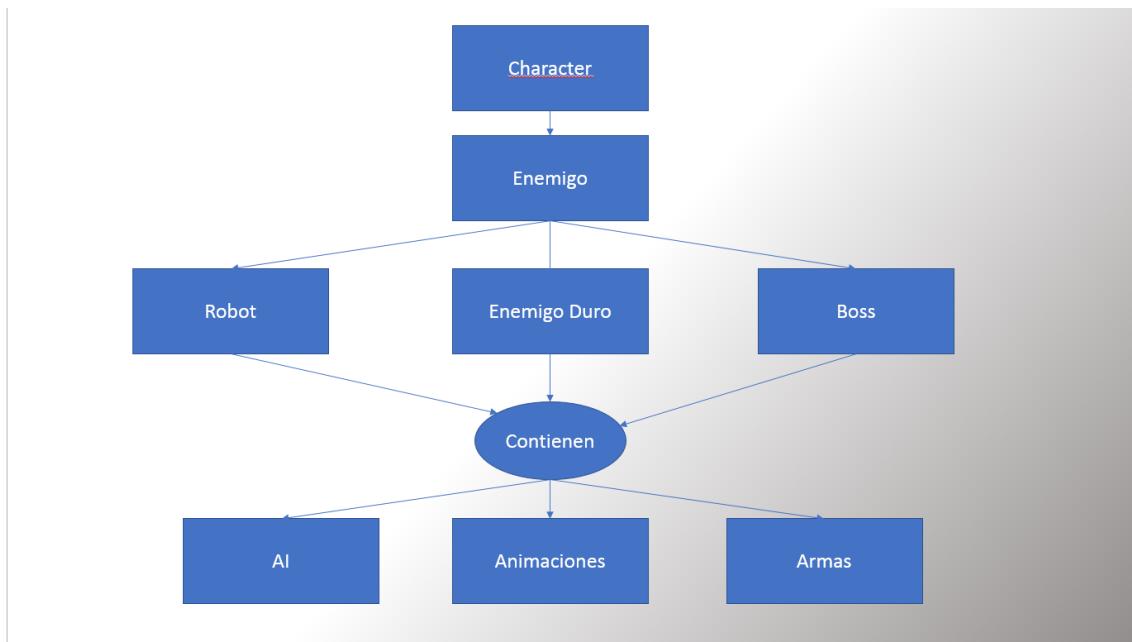


Figura 39: Diagrama de clases de los NPC

Al igual que el jugador, los NPC también heredan del BP *character* que contienen los elementos necesarios para moverte por el escenario. Los NPC tienen una clase general donde tienen guardada la información colectiva que usan todos los tipos de enemigos, esa información son las variables que todos los hijos van a tener y usar, para no ir repitiendo información, o funciones y/o características generales que van a usar todos. Es por eso por lo que este BP tiene pocos métodos implementados y variables porque se han de especificar en sus hijos. En la clase padre están los eventos relacionados con el magnetismo que gestionan las fuerzas de impulso que recibe el enemigo.

Aunque hay distintos tipos de enemigos, todos han sido implementados de la misma manera por heredar del BP parente *Enemigo*. Eso quiere decir qué, los métodos que antes no han sido implementados están creados en sus hijos, aunque son específicos para cada uno. En estos BP se encuentran los métodos del sistema de combos, donde el enemigo puede atacar; también está el gestor de colisiones, tanto de hacer daño como de recibir,

que dependen con qué objeto del escenario colisionen; y así como funciones y variables que sólo se encuentran en los hijos y son necesarias para cálculos internos suyos.

El sistema de recibir daño de los *NPC* funciona de la misma manera que con el jugador, aunque si un enemigo muere el juego no acaba. Se calcula el daño que recibe y se le resta a la vida, si pasa de cero el enemigo muere.

La IA es un apartado en común de todos los *NPC* que se especifica que elementos tienen los enemigos y como se han creado usando algoritmos existentes que han sido adaptados para este proyecto, cómo también, que opciones nos deja *Unreal* para gestionar y crear nuestra IA.

Al igual que el jugador, los enemigos también tienen sus armas, las cuales, tienen sus propias colisiones para interactuar con los objetos del escenario a los cuales pueden hacer daño que es, generalmente, el jugador; también está la información de qué malla van a usar o la cantidad de daño que pueden hacer.

5.7.3.1 IA

Unreal nos proporciona distintas de estas herramientas para facilitarnos el trabajo, aunque la gran mayoría de los elementos importantes de la IA se han trabajado con la base de estas herramientas usando distintos algoritmos. La IA es una de las partes más importantes de un juego y *Unreal* proporciona algunos de los elementos que se suelen usar para la IA y facilitar el trabajo quitando carga a los desarrolladores sobre elementos que se tardarían bastante tiempo en adaptar a un juego. Estos elementos son:

- **Navmesh:** *Unreal* tiene un elemento que calcula por nosotros las áreas navegables de nuestros escenarios, haciendo que la IA sepa por donde puede moverse calculando la ruta más corta. Este *navmesh* se puede modificar distintos elementos para modificar el comportamiento de la IA. Este objeto se puede poner sin la necesidad de crear algún BP, simplemente se arrastra hasta la zona del escenario donde se quiere que esté. Tampoco hace falta enlazarlo con la IA ya que, *Unreal*

lo usa por defecto a la hora de que una IA se mueva por el escenario hasta un punto o actor concreto usando la ruta más corta calculada por el BP de la IA

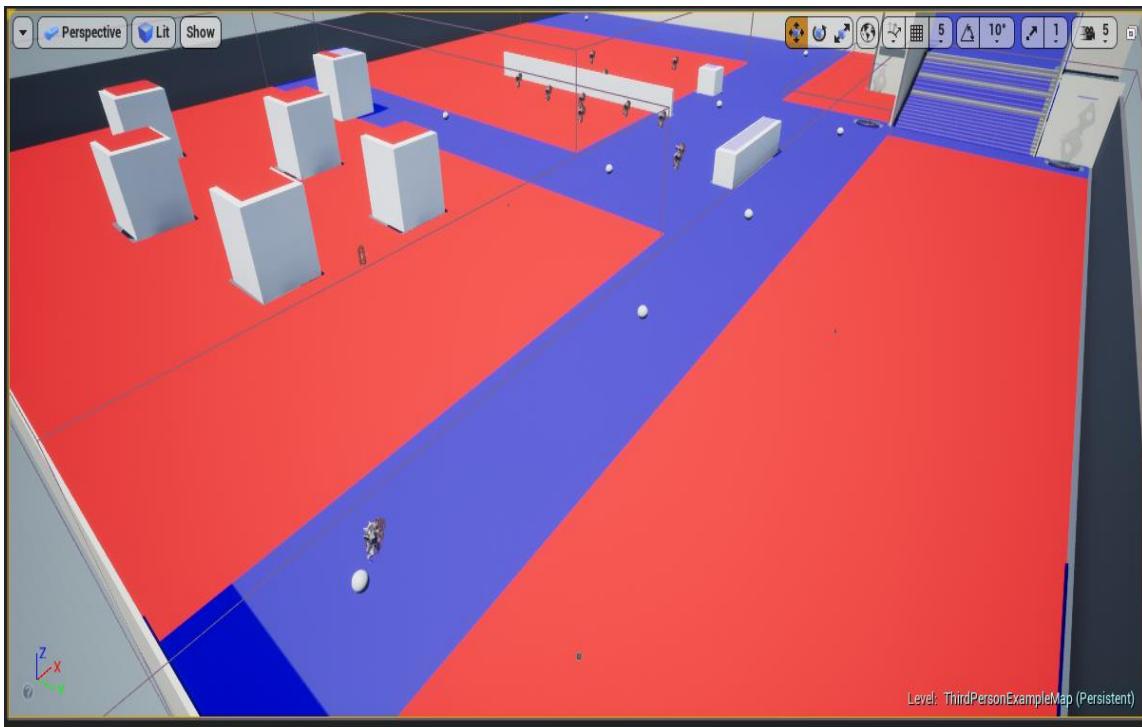


Figura 40: Ejemplo del navmesh

La figura 40 muestra cómo se comporta el *navmesh* de *Unreal* siendo el color azul una zona de más prioridad frente a la roja.

- **Behavior Tree:** *Unreal* nos proporciona una plantilla específica donde podemos crear nuestro árbol de comportamiento para nuestra IA, dónde le podemos decir qué y cómo tiene que actuar, dependiendo de distintos parámetros.

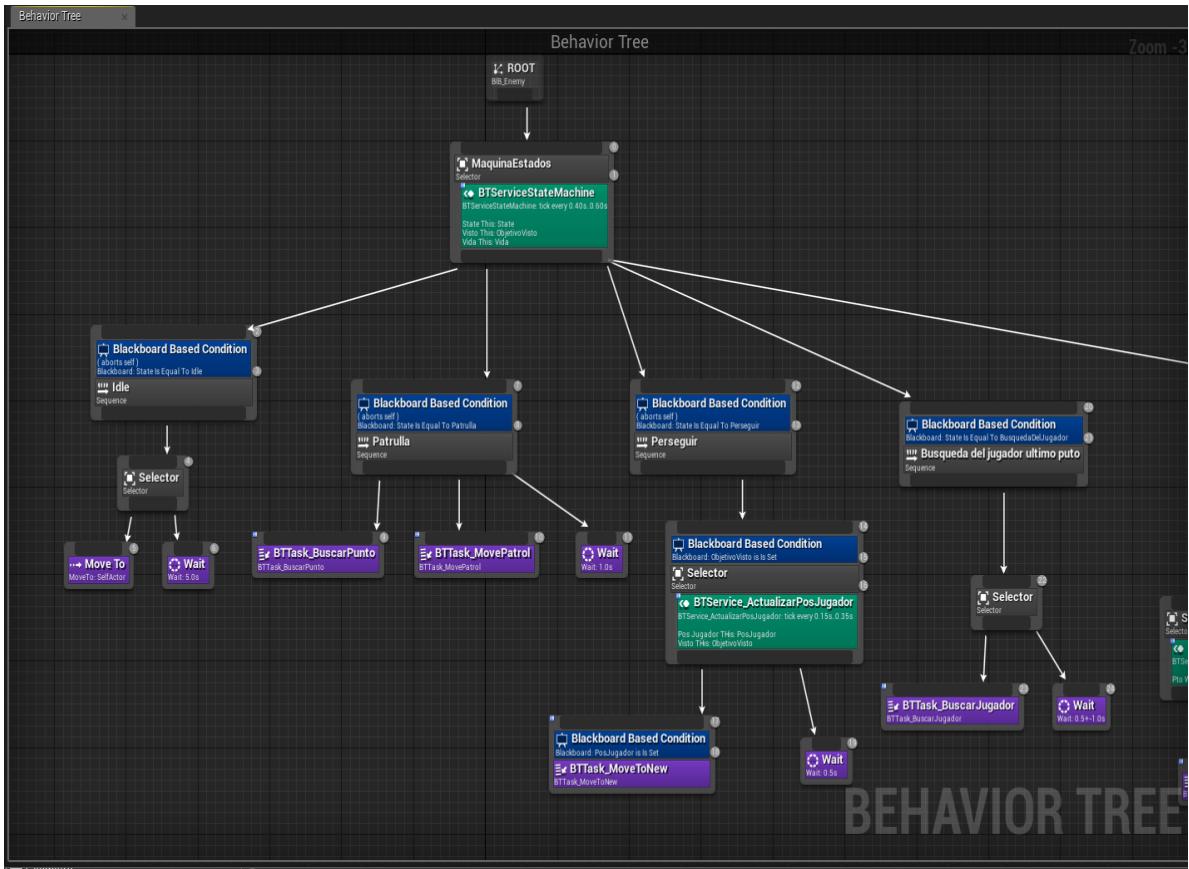


Figura 41: Ejemplo de behavior tree

Se puede ver la toma de decisiones que hará la IA. *Unreal* nos proporciona las herramientas para facilitarnos la creación de este árbol, pero, como se gestionan los distintos estados de la IA y cómo funcionan esos estados internamente se ha tenido que programar. *Unreal* no te proporciona una gestión de estados o tiene pre-creados estados típicos que la IA puede tener, esto tiene que gestionarlo y crearlo el propio programador para su uso en el juego. En el árbol de comportamiento se pueden ver los distintos estados que se han creado para el enemigo que cada uno de ellos se gestionan con una máquina de estados. Esta máquina de estados decide cual de todos los distintos estados ejecutar, dependiendo de una serie de condiciones. Estas condiciones están regidas por si ha visto al jugador, la distancia a la que se encuentra de él y la vida que tiene. Una vez se ha elegido un estado, se podrán realizar distintas acciones que se han implementado.

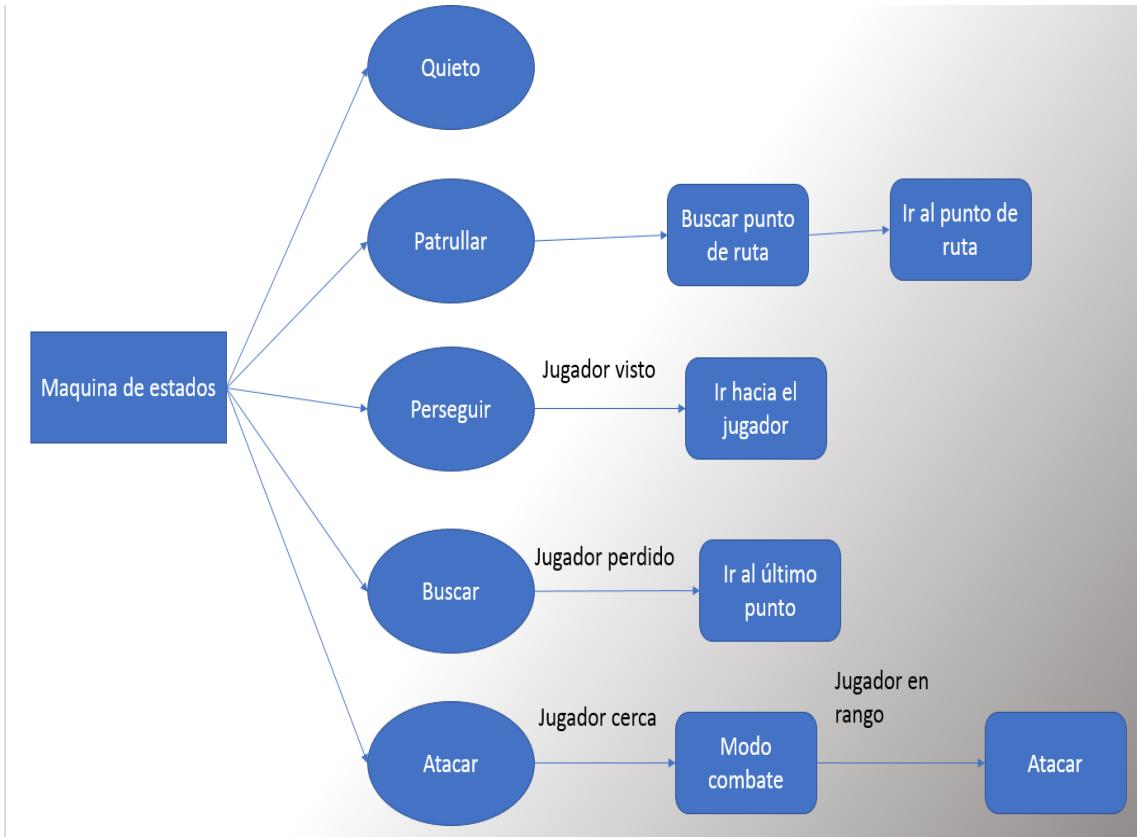


Figura 42: Esquema básico de la máquina de estados

Cada estado es único y puede cortarse si se cumplen las condiciones de otro estado.

Viendo la figura 41, se puede ver un pequeño esquema de cómo va funciona la gestión de la máquina de estados. El estado de patrullar se rige usando el *pathfinding* que se explicará más adelante, perseguir sigue al jugador hasta que lo pierde de vista y si eso ocurre, va al último lugar dónde le ha visto; y por último tenemos el estado de atacar que si está cerca del jugador se pondrá en modo combate. En ese modo puede atacar al jugador cuando este cerca de él.

- **Blackboard** (sistema de memoria): *Unreal* nos facilita un elemento para guardar información que luego usará la IA en el árbol de comportamiento.

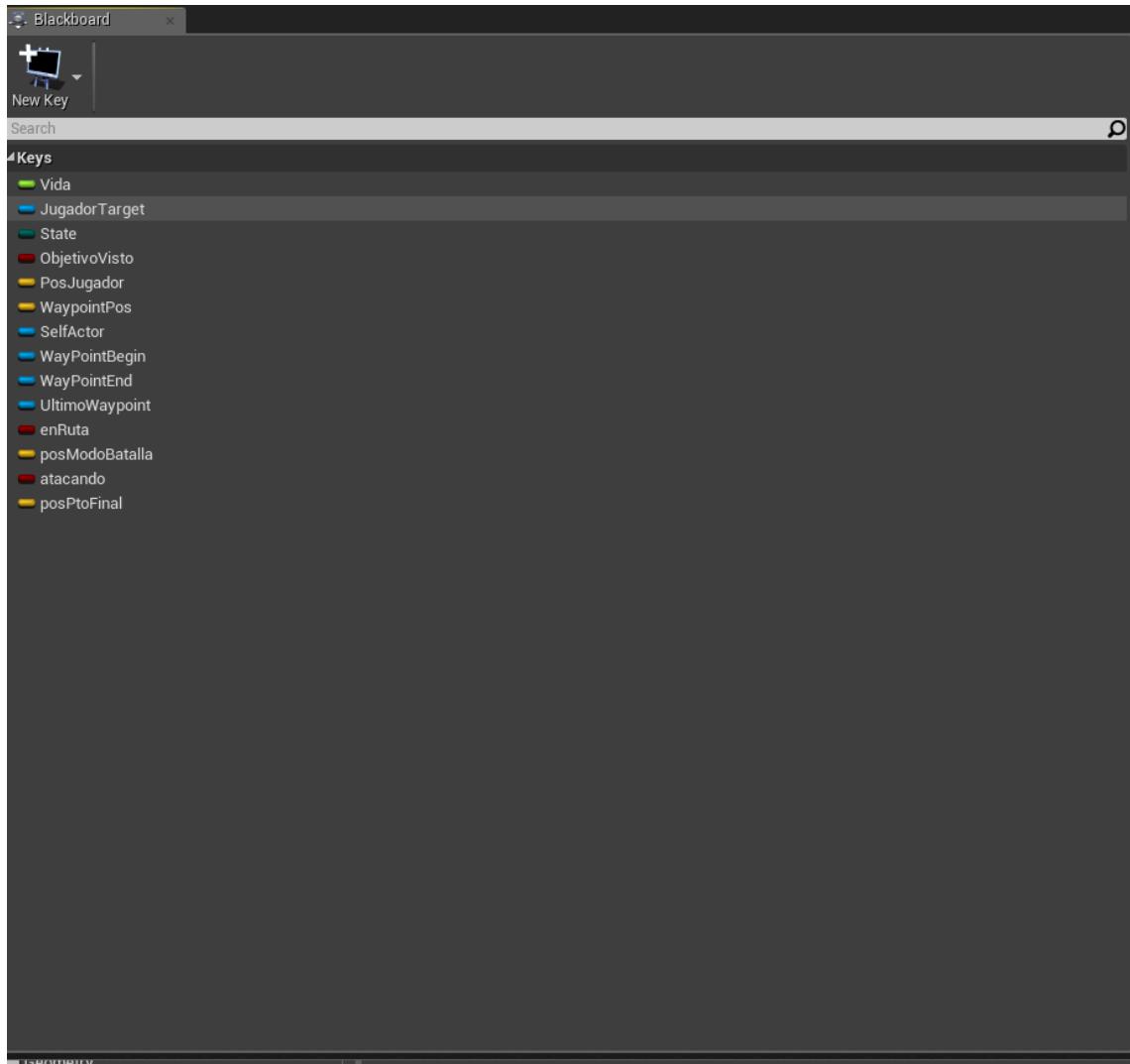


Figura 43: Ejemplo de blackboard

- **AIController** (Sistema de control de la IA): Este elemento nos permite conectar nuestra inteligencia (los distintos elementos nombrados anteriormente) con nuestra lógica del enemigo que está en otro BP. Este elemento nos permite dotar de sensores de vista, oído, olfato, etc... para hacer nuestra IA mucho más inteligente y poder usarlos en los elementos anteriores. Cuando se crea este BP, hereda de otro BP donde se gestiona el uso del *navmesh* y cálculo de la ruta más corta por éste.



Figura 44: Ejemplo de sensor de visión

Con esta imagen se puede ver el sensor de vista aplicado a los enemigos. El área verde indica que estamos en su campo de visión.

Estos son los distintos elementos que ofrece *Unreal* para nuestra IA, todo lo demás ha sido añadido adaptando distintos algoritmos de IA que se han creado a la lógica de *Unreal*. Estos son:

- **Pathfinding:** *Unreal* no proporciona uno como tal, se ha creado uno usando como base los *waypoints* (puntos de rutas) dónde es la propia IA la que elige su destino y, hasta que no llega a ese destino, no vuelve a seleccionar otro. El punto de usar este algoritmo es que se calcula un grafo de distancias donde se sabe a qué distancia está un nodo de otro, una vez se ha calculado todo, la IA busca el nodo más cercano a él hasta que llegue a su destino. Hay muchas maneras de hacer este algoritmo, pero la que se ha usado en el juego usa los nodos visitados. Una vez que la IA ha pasado por ese nodo, no vuelve a visitarlo hasta que acabe el viaje (llega a su destino).
- **Sistema de ataque:** El sistema de ataque de cada IA es distinta en cada juego, por eso *Unreal* no ofrece un standard a la hora de implementar esa mecánica en su

motor, sino que, te da facilidades para que se implemente uno propio. En este caso, es un sistema de combos que el enemigo te ataca cuando esté a cierta distancia. Estos ataques no continúan si el jugador sale de cierto rango. Cada combo del enemigo es distinto y no son iguales, cuando llega a cierto nivel de vida puede usar otros combos más fuertes que antes no podía usar.

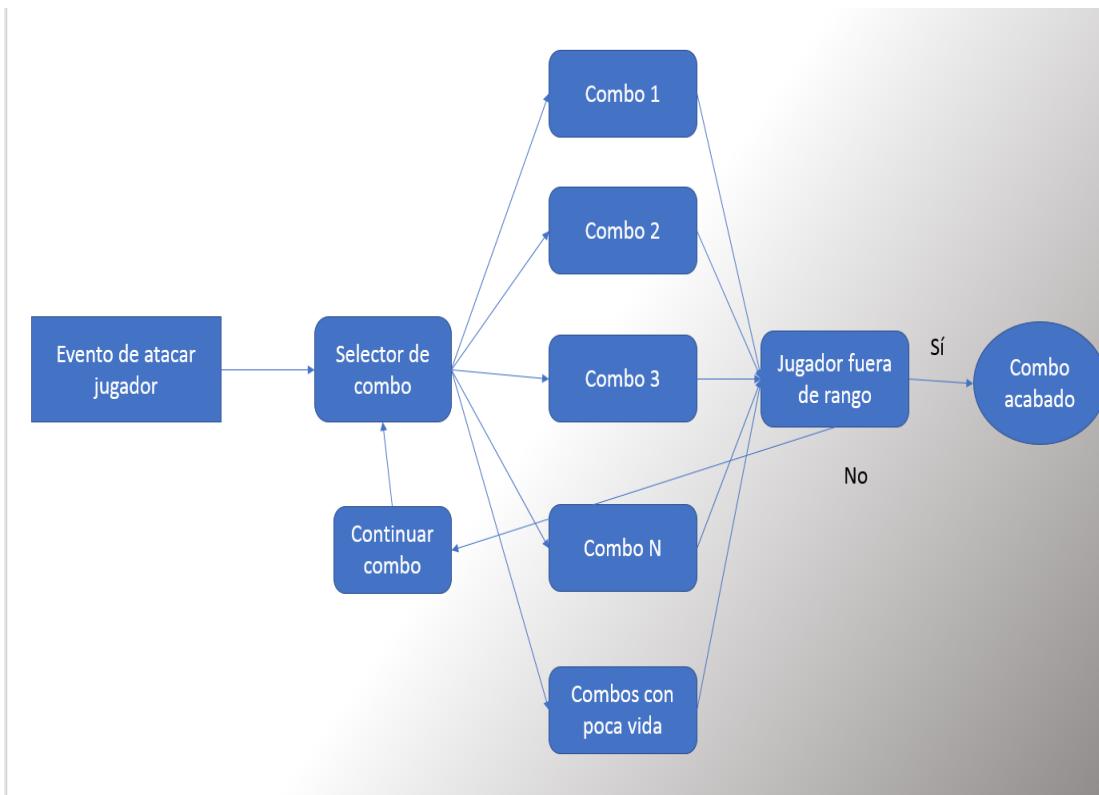


Figura 45: Diagrama del sistema de combos enemigo

- **Comportamiento básico:** Como se ha nombrado anteriormente, *Unreal* nos facilita una herramienta para crear nuestro propio árbol de comportamiento. La IA patrulla, usando el *pathfinding* y los *waypoints* como apoyo, persigue al jugador cuando le ve y si le pierde de vista va al último punto dónde le ha visto y, ataca cuando está a la distancia adecuada. Todos esos elementos se han implementado.

5.7.4 Elementos del escenario

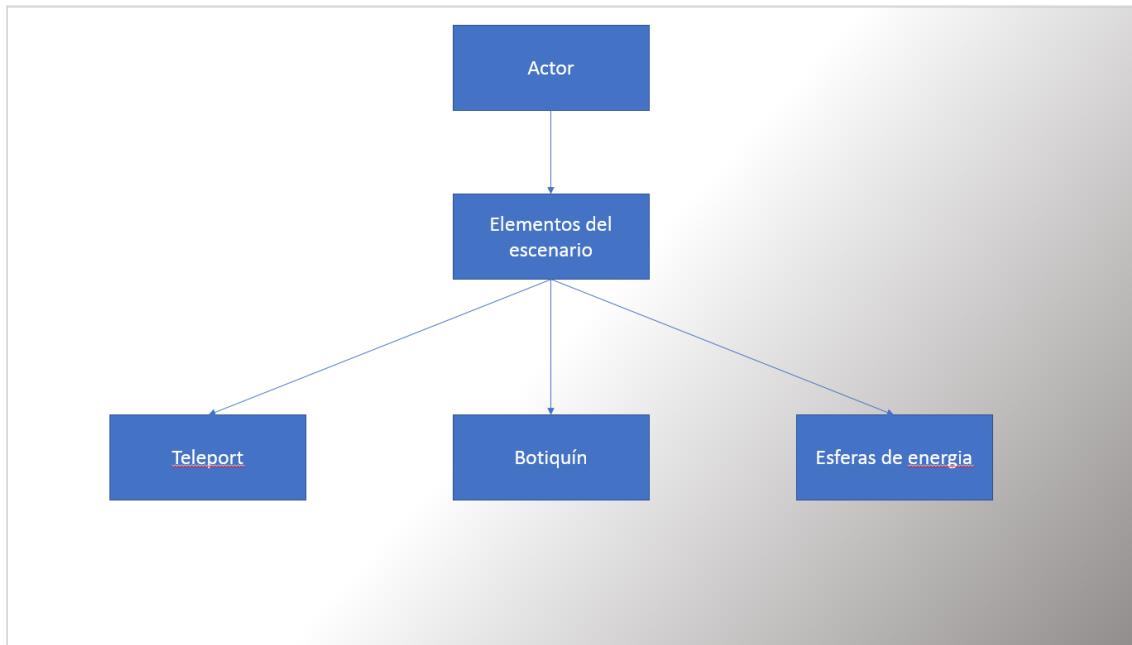


Figura 46: Diagrama de los elementos del escenario

Una parte importante de los escenarios son los elementos con los cuales, el jugador puede interactuar con ellos. Estos elementos son creados a partir del elemento más básico que tiene *Unreal*, qué es un actor. Este actor posee los elementos mínimos para poder existir en el nivel. Al igual que los demás elementos del juego, se ha hecho una estructura dónde los elementos más usados, o comunes, estén en una sola clase; al haber distintos elementos y que cada uno tiene una función distinta en el juego. Empezando por el teletransportador, éste hace que el jugador pueda ir a otro punto del nivel o pasar de zona; el botiquín es usado por el jugador únicamente donde le curará una parte de la vida perdida y, por último, las esferas de energía que son las responsables de que los campos de energía estén activos.

5.7.5 Animaciones

Unreal tiene un BP de animación que gestiona todo lo relacionado con los movimientos que hará el actor para ver en qué momento del juego, y de qué forma, realizar una animación de andar, saltar o atacar que depende de una serie de circunstancias. Estas animaciones se gestionan con una máquina de estados que se tiene que crear y ajustar que varía dependiendo del actor. *Unreal* nos facilita una plantilla donde se pueden poner estados, pero, ver como se gestionan o se estructuran es cosa del desarrollador. Como todas las clases que usan el BP de *character* usan este tipo de objeto, suelen tener una forma en común de implementación aunque, varía dependiendo de las necesidades del personaje y de las funciones que puede hacer dentro del juego, en este caso, las máquinas de estado y la gestión de la comunicación entre su BP correspondiente varían entre el jugador y los enemigos, y dentro de los enemigos, presentan cambios muy pequeños por el cambio de ciertas variables que uno no tiene y otro sí y viceversa.

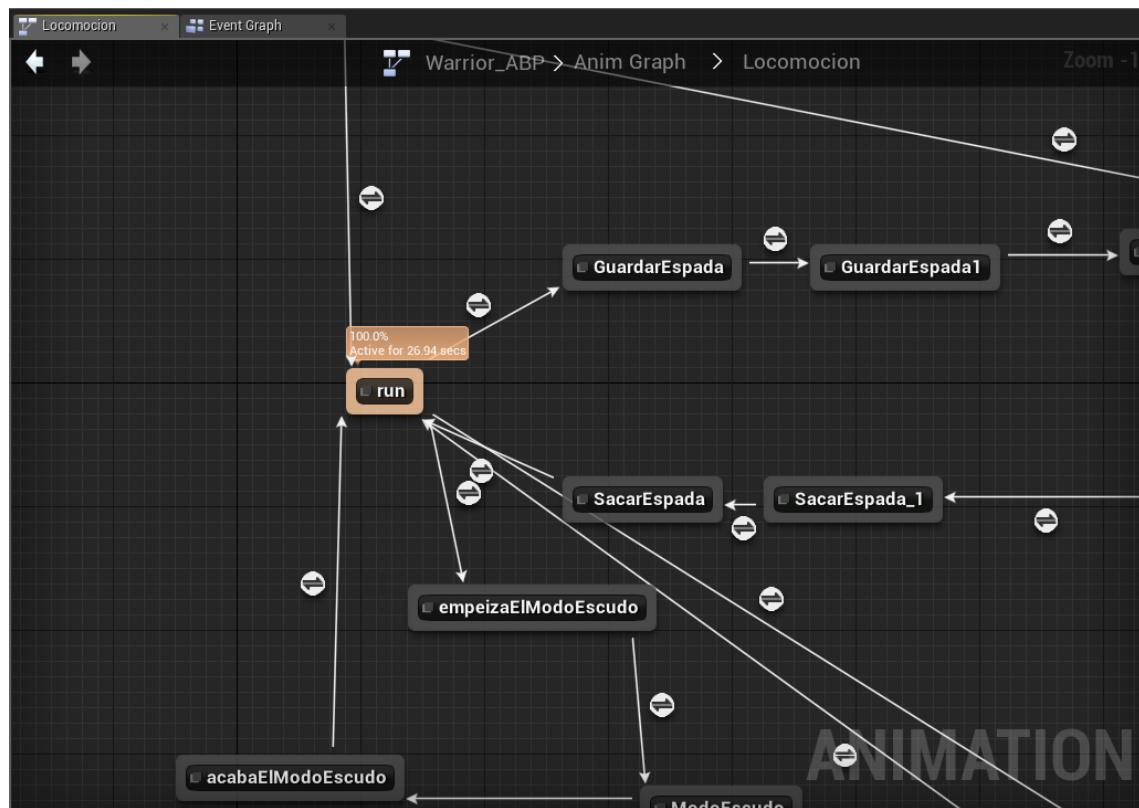


Figura 47: Máquina de estados de las animaciones del jugador

Las animaciones van cambiando según la máquina de estado asociada al BP. Dependiendo de si alguna de las variables que controlan a esta máquina cambian o no, se irán gestionando estas animaciones y ejecutando conforme se satisfagan sus condiciones. Si el personaje salta cambiarán las animaciones a las de salto, si cambia de arma o si se cubre.

Hay una herramienta que tiene *Unreal* que ha facilitado el uso de las animaciones para poder reutilizar en otros esqueletos distintos. Esta herramienta ha facilitado el uso de las animaciones enfocadas a un esqueleto poder usarlas en otro usando un intérprete que traduce qué hueso es el equivalente en el otro esqueleto dónde se quiere pasar la animación. Gracias a este elemento, se pueden reutilizar las animaciones para usarlas en el máximo número de elementos jugables del juego que se mueven por el escenario para no perder tiempo.

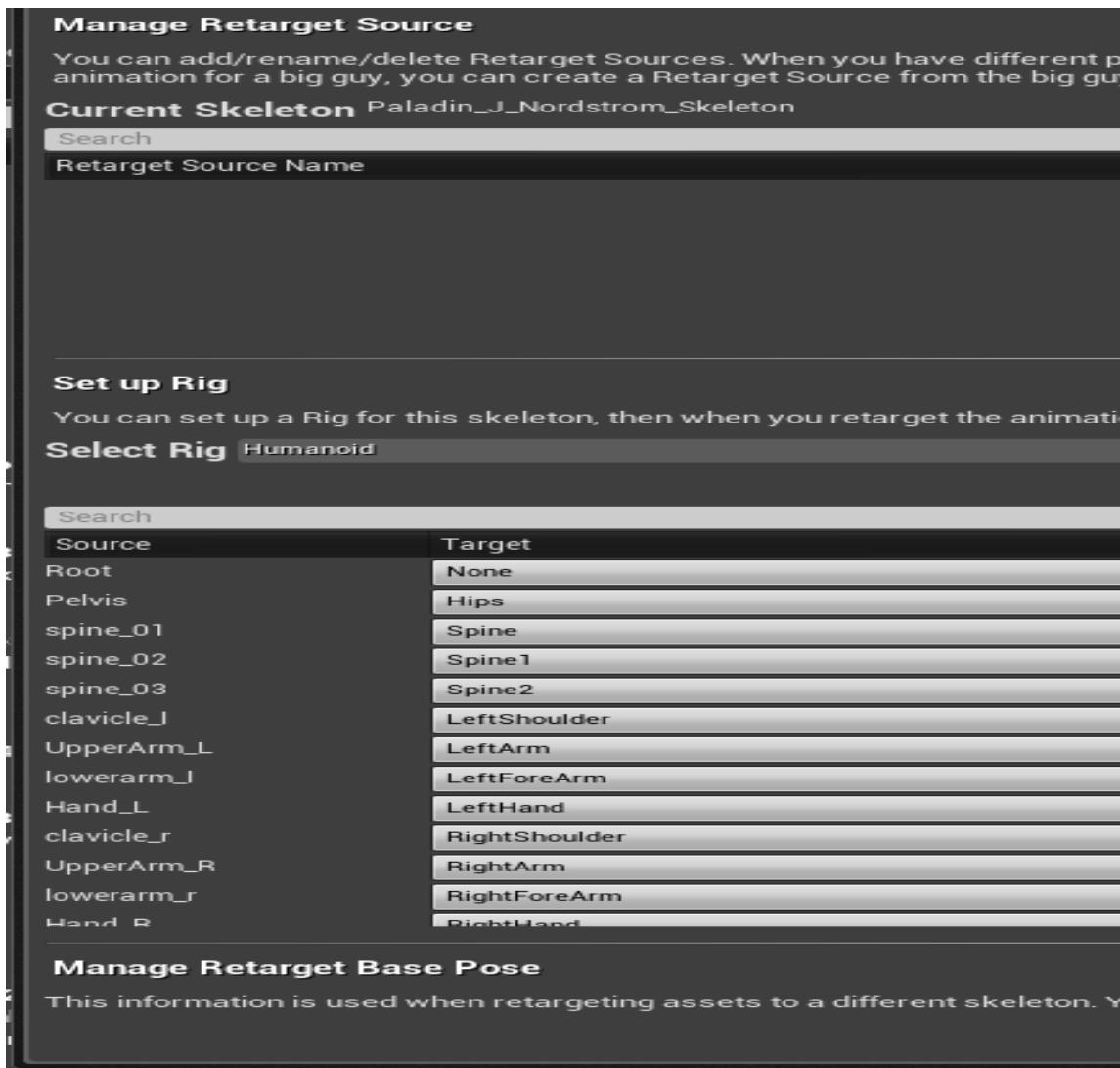


Figura 48: Herramienta de retarget de animaciones

La función principal de esta herramienta es, llevar las transformaciones realizadas en una animación hecha con un esqueleto a otro que no es el mismo esqueleto y el nuevo esqueleto sepa a que huesos afectan esas transformaciones y permitir el rehusó de animaciones hechas para humanoides poder usarlas en esqueletos similares

6. Conclusiones

En general, gracias a este proyecto he aprendido el objetivo con el cuál empecé hacer este proyecto que es: aprender a planificar un juego y a usar un motor gráfico actual que se usa en el mercado laboral. Los resultados han sido satisfactorios, he aprendido a usar el motor desde cero y también he mejorado como programador, es por eso por lo que, gracias a estos conocimientos que he adquirido a lo largo de este proyecto me he dado cuenta de muchas de las partes que se pueden mejorar o enfocar de otras maneras para obtener el mismo resultado esperado.

Sobre el juego he de decir que ha quedado una base sólida para seguir añadiendo más contenido al juego. Es cierto de que me hubiera gustado añadir más niveles al juego como también más tipos de enemigos y añadir más mecánicas al jugador, pero por falta de tiempo y recursos no ha podido ser. Aún así, el producto cumple con los requisitos planteados al principio del proyecto y he podido hacer el juego que quería en un motor que nunca había usado.

Ha sido una experiencia muy satisfactoria porque he tenido que tocar casi todos los apartados que dispone el motor para diseñar un videojuego, tales como el modelado de los distintos objetos, mirar animaciones y gestionarlas, crear los materiales de distintos elementos del escenario, crear efectos de partículas, establecer la distinta iluminación que va a tener el escenario y, como no, programar los distintos elementos para que puedan usarse en el juego. Gracias a esto he podido enfocarme en lo que más me ha gustado e intentar plasmarlo en el juego.

7. Bibliografía y referencias

Referencias orden alfabético

Capcom Co. (s.f.). *Capcom*. Obtenido de <http://www.capcom-europe.com/>

EA Corporation. (s.f.). *Frostbite engine*. Obtenido de <https://www.ea.com/frostbite>

Epic Games. (s.f.). *Epic Games Behavior Tree*. Obtenido de
<https://docs.unrealengine.com/en-US/Engine/AI/BehaviorTrees/HowUE4BehaviorTreesDiffer>

Epic Games. (s.f.). *Epic Games Blackboard*. Obtenido de
<https://docs.unrealengine.com/en-us/Engine/AI/BehaviorTrees/NodeReference/Decorators>

Epic Games. (s.f.). *Epic Games BluePrints*. Obtenido de
<https://docs.unrealengine.com/en-us/Engine/Blueprints/GettingStarted>

Epic Games. (s.f.). *Unreal Engine*. Obtenido de <https://www.unrealengine.com/en-US/what-is-unreal-engine-4>

GitHub. (s.f.). *GitHub*. Obtenido de <https://github.com/>
Opel, M. (s.f.). Obtenido de
https://www.youtube.com/watch?v=g0nM43FOVdk&list=PLFY4PASroVXJq-NzsHi_nvsORUUSk0yuE

Platinum Games INC. (s.f.). *Platinum Games*. Obtenido de
<https://www.platinumgames.com/>

Santa Monica. (s.f.). *God Of War*. Obtenido de <http://godofwar.playstation.com/>

Santa Monica Studio. (s.f.). *Santa Monica Studio*. Obtenido de
<http://sms.playstation.com/>

Tutorial de combate. (s.f.).

Unity Technology. (s.f.). *Unity* . Obtenido de <https://unity3d.com/es>

Unreal Foro. (s.f.). Obtenido de
<https://answers.unrealengine.com/questions/195362/combine-attack-animation-blueprints-help.html>

Valve Co. (s.f.). *Valve NavMesh*. Obtenido de
https://developer.valvesoftware.com/wiki/Navigation_Meshes

Wenderlich, R. (s.f.). *Raywenderlich Pathfinding*. Obtenido de
<https://www.raywenderlich.com/4946/introduction-to-a-pathfinding>

Wikipedia. (s.f.). *Wikipedia Bayonetta*. Obtenido de
<https://es.wikipedia.org/wiki/Bayonetta>

Wikipedia. (s.f.). *Wikipedia Devil May cry*. Obtenido de
[https://es.wikipedia.org/wiki/Devil_May_Cry_\(videojuego\)](https://es.wikipedia.org/wiki/Devil_May_Cry_(videojuego))

Wikipedia. (s.f.). *Wikipedia God of War*. Obtenido de
https://es.wikipedia.org/wiki/God_of_War

Wikipedia. (s.f.). *Wikipedia Hack and Slash*. Obtenido de
https://es.wikipedia.org/wiki/Hack_and_slash

Wikipedia. (s.f.). *Wikipedia Metal Gear Rising*. Obtenido de
https://es.wikipedia.org/wiki/Metal_Gear_Rising:_Revengeance

Wikipedia. (s.f.). *Wikipedia Quick Time Event*. Obtenido de
https://es.wikipedia.org/wiki/Quick_time_event

Wikipedia. (s.f.). *Wikipedia Videojuego*. Obtenido de Videojuego:
<https://es.wikipedia.org/wiki/Videojuego>

YouTube. (s.f.). *YouTube*. Obtenido de <https://youtu.be/d15dPiC9V8Q?t=16m49s>

8. Anexos

Enlace al repositorio: <https://github.com/DavidBrando/FuturoImperfecto>

Enlace a la gestión del proyecto:

<https://github.com/DavidBrando/FuturoImperfecto/projects>

8.1 Recursos