



VNiVERSiDAD D SALAMANCA

CAMPUS DE EXCELENCIA INTERNACIONAL

Práctica 2 Unreal: “The Escapist”

Animación Digital

Grado en Ingeniería Informática

Luis Blázquez Miñambres, 70910465Q

Contenido

1.	Introducción	3
2.	Manual de Juego	4
3.	Desarrollo principal del Juego principal	5
	a. Importación de personaje y movimiento	5
	b. Grafo de animaciones	7
	c. HUD.....	10
	d. Menú principal.....	12
4.	Características	14
	a. Personaje principal	14
	i. Sockets y equipar/desequipar arma.....	14
	ii. Recoger el arma	18
	iii. Muerte del personaje	19
	b. Arma.....	20
	i. Propiedades y características	20
	ii. Zoom.....	26
	c. Enemigos.....	29
	i. Movimiento y patrulla	29
	ii. Aplicar daño.....	32
	d. Mapa	34
	e. Objetos recogibles	37
	i. Vida y munición	37
5.	Bibliografía	39

1. Introducción

El juego que he realizado es un *shooter* en tercera persona en la que el jugador principal debe encontrar una manera de escapar de una prisión combatiendo a varios enemigos por el camino con un arma de fuego.

El aspecto de la interfaz principal quedaría de la siguiente manera, donde la cámara que sigue al personaje está un poco movida a la derecha de este debido al sistema de disparos que explicaremos más adelante.



Imagen 1.1: Interfaz principal del juego

Podemos observar otros detalles, que ya explicaré más adelante, como la cruz del centro cuando el arma esta equipada o la cantidad de munición que nos queda y tenemos en cada instante, así como la barra de salud del personaje.

Otro detalle como introducción al resto de apartados es indicar las entradas de teclado y ratón que he añadido al juego para simplificar la tarea a la hora de crear los blueprints con eventos de teclado y ratón. Así como utilizar un elemento de tipo *Structure* (en *ThirdPersonCharacter > MyCharacter > Weapons*) para que me fuera más fácil acceder a las propiedades de la clase a la que pertenece el objeto del arma principal del personaje principal.

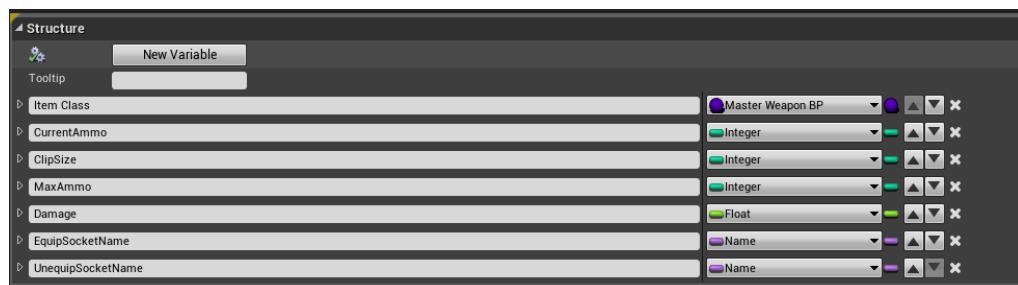


Imagen 1.2: Estructura del arma del personaje principal

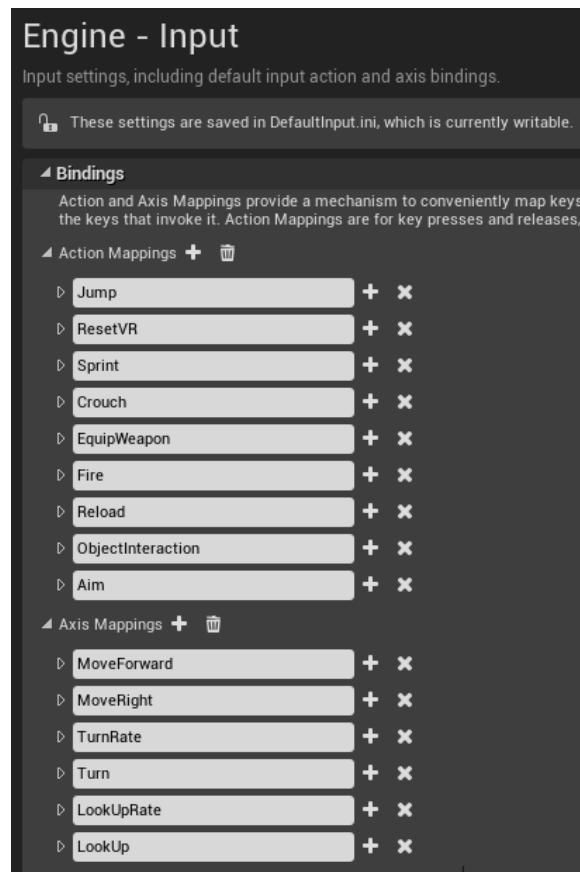


Imagen 1.3: Entradas del juego

2. Manual de Juego

Aunque los controles del juego se pueden observar en el menú del juego, aquí los dejaré reflejados. En concreto:



Disparar arma: Botón Izquierdo del ratón



Apuntar con arma: Botón derecho del ratón



Mover al personaje: Teclas W, A, S, D



Interactuar (Pulsar un botón, recoger objeto, ...): Tecla E



Equipar o desequipar el arma: Tecla F



Saltar: Barra espaciadora



Recargar munición: Tecla R



Sprint: Tecla Mayus Izquierda

3. Desarrollo principal del Juego principal

A continuación, explicaré brevemente el desarrollo de las características básicas del juego desde las fases más primerizas hasta llegar al juego final, antes de entrar en las características más detalladas.

a. Importación de personaje y movimiento

Tanto para la importación del aspecto del personaje principal y de los enemigos, así como sus respectivas animaciones, las importé desde la página web “Mixamo” ya que intenté importarlas desde otros portales, pero fue imposible debido a la capacidad de almacenamiento que suponían.

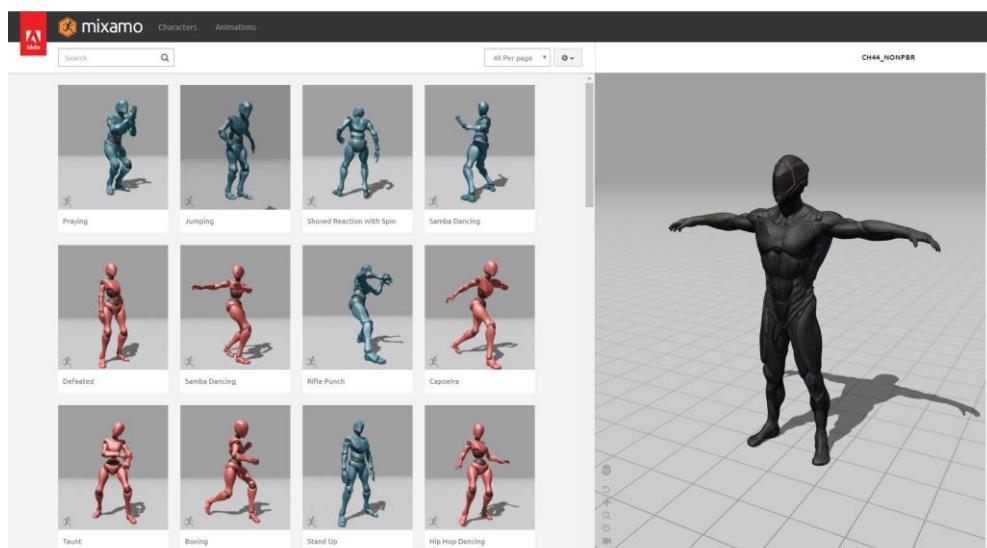


Imagen 3.a.1: Aspecto de la página web Mixamo

En primer lugar, cree un proyecto vacío con los ajustes por defecto para un juego en tercera persona que trae, entre otras cosas, un personaje y un escenario por defecto como el que se puede ver en la imagen 3.a.2, dejando los blueprints que venían por defecto con respecto a las animaciones (ajuste de la velocidad, del salto, etc). A partir de aquí cogí al personaje del escenario y le cambié el *Skeletal Mesh* al de mi personaje que había importado.



Imagen 3.a.2: Personaje principal con skeletal mesh del personaje de Mixamo

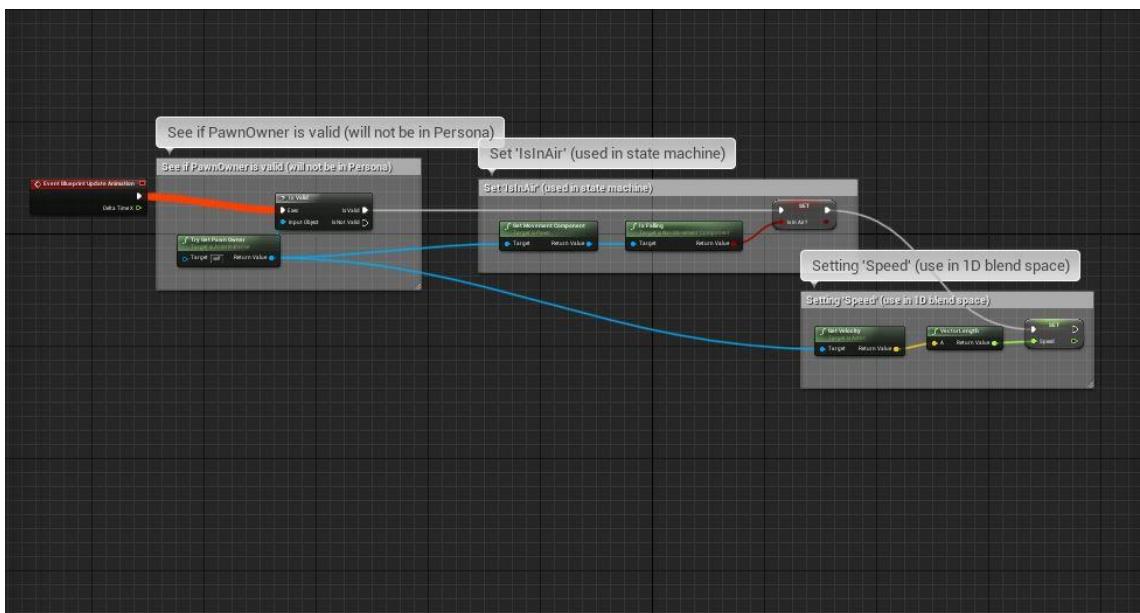


Imagen 3.a.3: Blueprints del AnimGraph del personaje por defecto en tercera persona

A continuación, ajuste una de las propiedades del personaje, el movimiento, a través de una variable en el *EventGraph* de mi personaje. Además de ajustar la acción de Sprint (mientras se mantenga la tecla Mayus Izquierda), aumentando el valor de la propiedad ya intrínseca en el personaje (*Max Walk Speed*) del *ThirdPersonCharacter* consigiendo aumentar la velocidad del personaje e ir cambiando entre las acciones de caminar y correr.

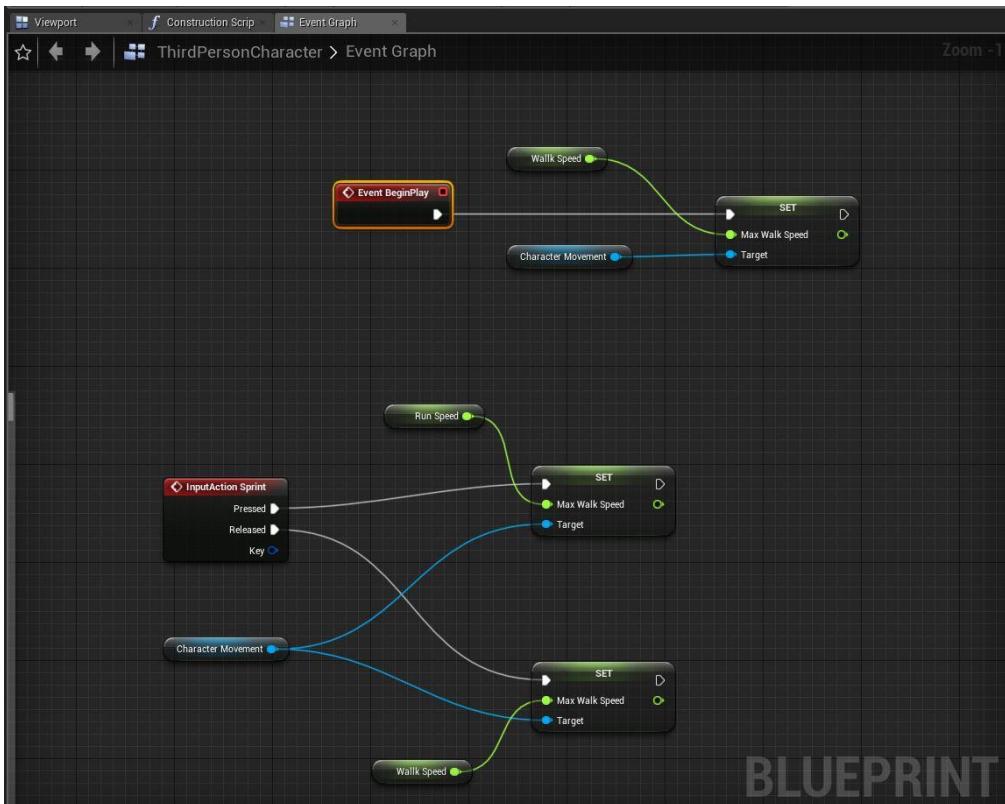


Imagen 3.a.4: Blueprints para la acción de movimiento – correr y andar

b. Grafo de animaciones

Posteriormente, añadí el conjunto de animaciones (que se pueden encontrar en *ThirdPersonBP > MyCharacter > Animations*) principales para caminar, correr y saltar del personaje en el apartado *MyCharacter_Animation > AnimGraph > Locomotion* del personaje.

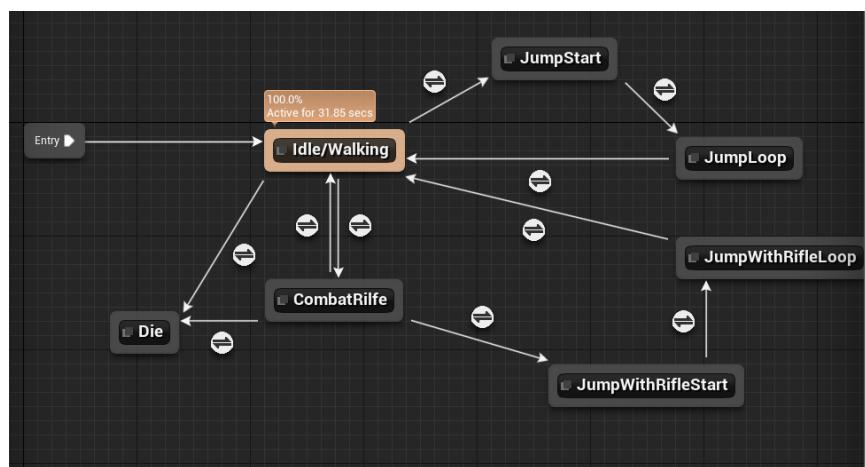


Imagen 3.b.1: AnimGraph del personaje

Principalmente hay que decir que tendremos tres estados principalmente:

- *Idle/Walking*: este estado tendrá las animaciones de caminar y correr, agrupadas en un Blend Tree, con un solo eje (X) en función de la velocidad. A continuación, explicaré el contenido del Blend Space de movimiento donde se puede ver el único eje X que he mencionado y tres puntos que indican las tres animaciones que se ejecutarán cuando la velocidad del personaje sea 0 (animación *Idle o parado*), cuando sea 250 (*caminar*) y cuando sea máxima 500 o más (*correr*).

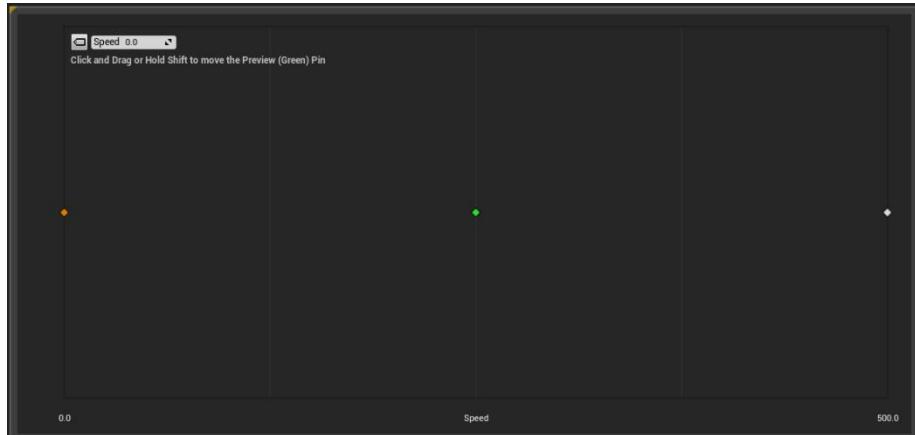


Imagen 3.b.2: AnimGraph del personaje

- *JumpStart y JumpLoop*: la animación de *JumpStart* y *JumpLoop* son la misma con la excepción de que se pasa del estado de caminar al primer estado de saltar mencionado, *JumpStart*, el jugador pulsa la tecla asociada a la acción *Jump*. Y pasa de este estado al segundo estado, *JumpLoop*, mientras esté en el aire. Esto último se puede detectar a través de una función en el Blueprint del personaje llamado *IsInAir*, intrínseca en el personaje por defecto (como se puede comprobar en la imagen 3.a.3). Volviendo al estado de caminar cuando el valor de la función anteriormente mencionada es falso.
- *CombatRifle*: para el movimiento con el arma también utilicé un BlendSpace que dependía de la dirección a la que se movía el personaje y su velocidad, al igual que el estado de movimiento normal.

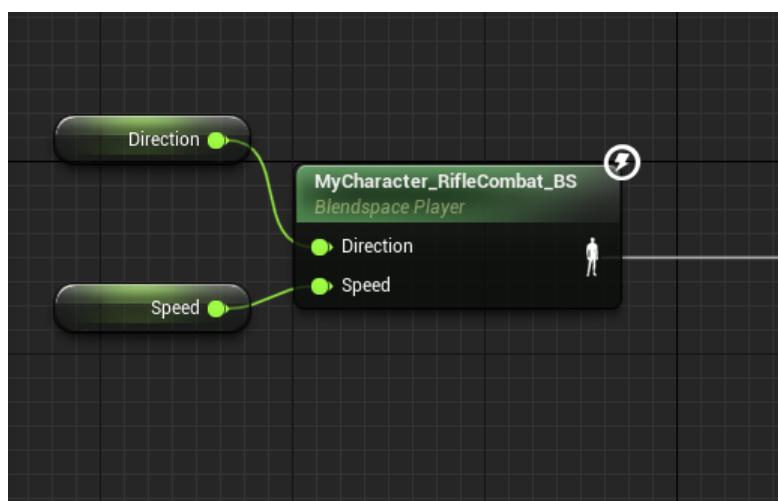


Imagen 3.b.3: BlendSpace dentro del estado *CombatRifle* en el AnimGraph

Por otro lado, el BlendSpace creado para el movimiento del personaje del mapa dependerá de dos ejes, X e Y, indicando la velocidad de movimiento y la dirección a la que apunta el personaje con el arma, respectivamente. Pudiendo caminar hacia atrás con el arma puesta sin que la cámara gire cuando la dirección del personaje es menor o igual a 180.



Imagen 3.b.3: BlendSpace de movimiento con el arma equipada

El efecto para conseguir que, equipada el arma, la cámara se mantenga en la misma dirección permitiendo que el personaje camine hacia atrás sin rotar, se consigue en el *EventGraph* del *ThirdPersonCharacter* con un macro llamada *Orient/Yaw* en la que se determina si el personaje girará sobre si mismo sin el movimiento de cámara (*Use Controller Rotation Yaw*) o si, teniendo el arma equipada, no girará al personaje sino que la orientación se mantendrá fija, permitiendo caminar hacia atrás o hacia los lados apuntando siempre hacia el frente con el personaje. Se puede ver en la función *Pitch/Yaw* en el *EventGraph* del personaje en *MyCharacter_AnimationBlueprint*.

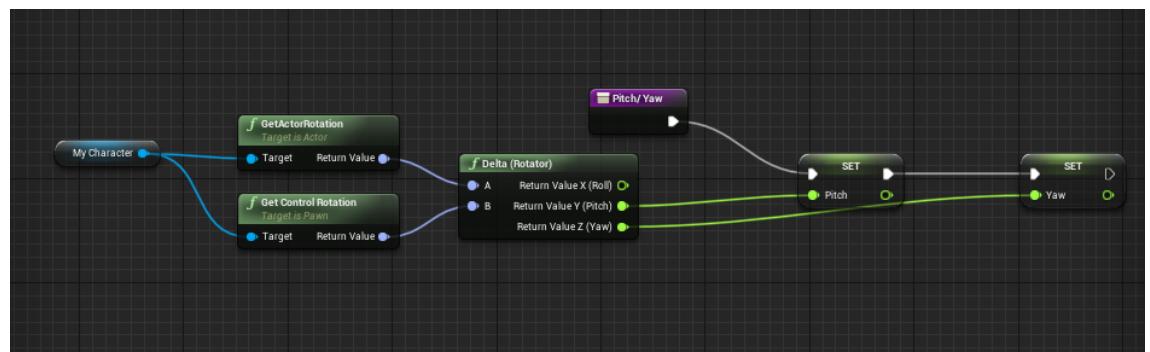


Imagen 3.b.4: Función para asignar la orientación y rotación del personaje en cada momento para las animaciones

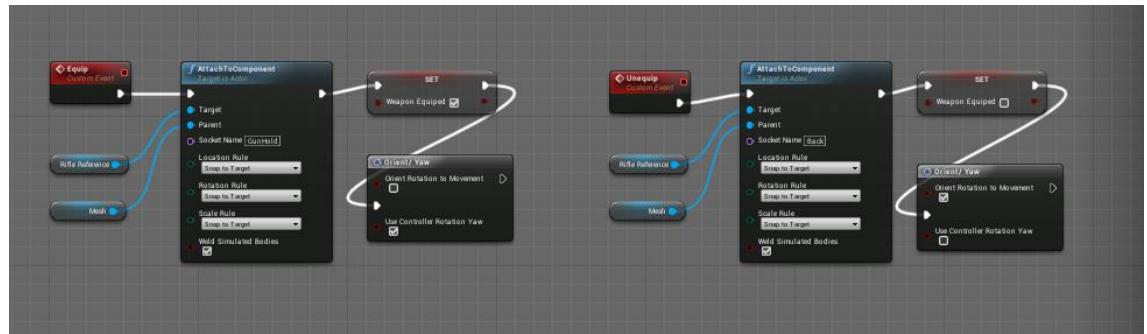


Imagen 3.b.5: Evento de equipar y desequipar el arma en EventGraph del ThirdPersonCharacter (ThirdPersonBP > Blueprints)

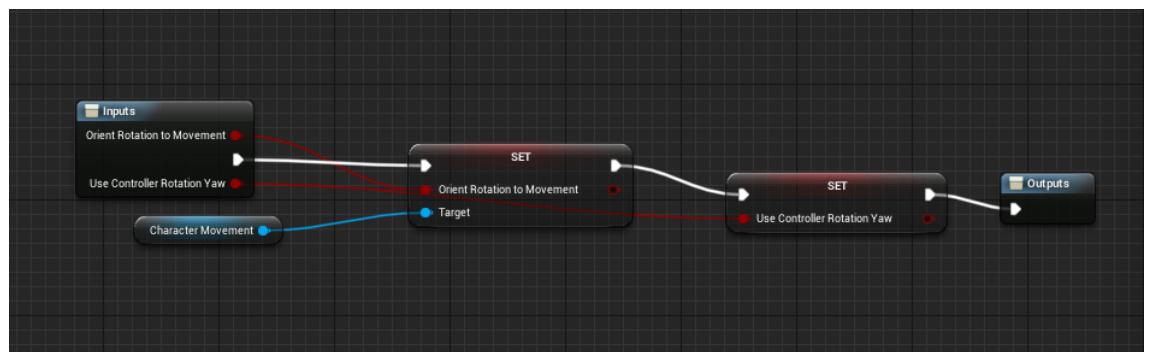


Imagen 3.b.6: Macro para evitar que rote la cámara con el personaje cuando el arma está equipada y siempre mire al frente cambiando la orientación del personaje o bloquearla en movimiento (rectángulo en gris de la imagen 3.c.5)

- [JumpWithRifleStart](#) y [JumpWithRifleLoop](#): de manera análoga a la estructura de las animaciones JumpStart y JumpLoop , con la excepción de que estas animaciones parten del estado CombatRifle (cuando el arma está equipada).
- [Die](#): activa la animación de muerte del personaje cuando la variable *Die* del Blueprint del personaje consigue un valor verdadero, que será cuando su vida sea menor o igual a 0.

c. HUD

En el HUD principal, creamos un *widget* que asociaremos a nuestro personaje del personaje encontraremos, como ya mencionamos anteriormente, la barra de salud, la munición y la cruz de apuntado del arma.

El funcionamiento de este es básico por lo que no me pararé a entrar en muchos detalles al respecto de este. Simplemente, para la barra de salud y los textos que indican la munición máxima (morada) y la munición del cartucho actual (amarillo) creamos un *Binding* o una función con la que gestionar el control de estos elementos del HUD. En concreto, los blueprints para mostrar la cruz de apuntado y la munición si el jugador principal tiene equipada el arma principal o no. Y en cuanto al porcentaje de la barra de vida, se irá actualizando con el valor de la variable *CurrentHealth*, correspondiente a la cantidad de vida del jugador.

Todas estas variables se encuentran declaradas en el *EventGraph* del *ThirdPersonCharacter*.

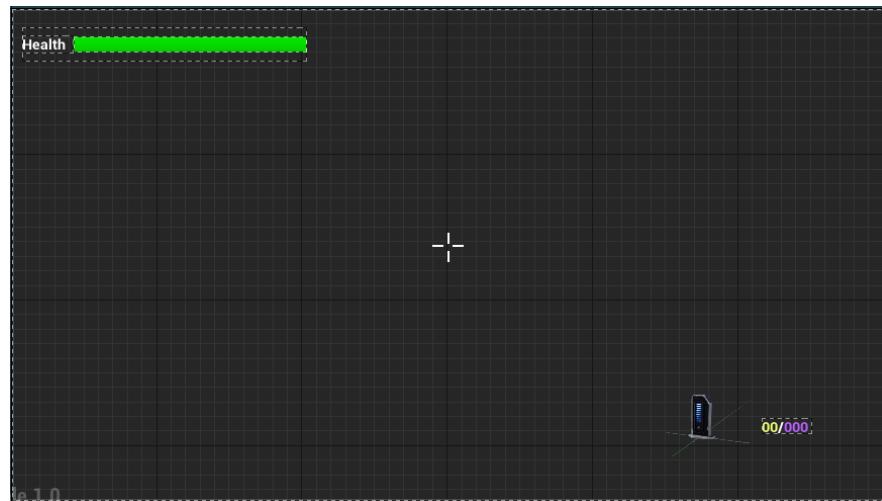


Imagen 3.c.1: HUD principal del jugador

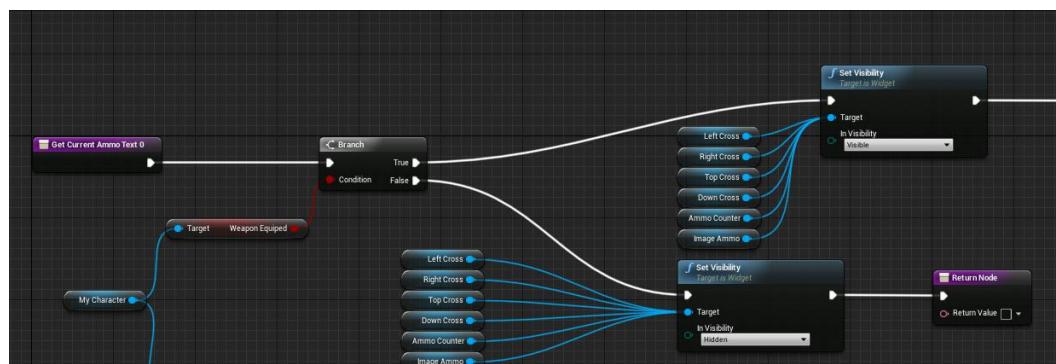


Imagen 3.c.2: Parte del Binding Blueprint donde se muestra el comportamiento del HUD al equipar el arma al personaje

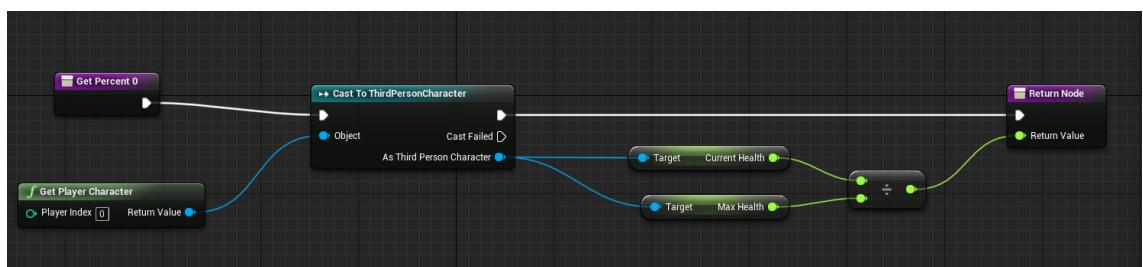


Imagen 3.c.3: Binding Blueprint para la barra de salud

Por último, para que el *widget* se instancie en el juego utilice la función *Create Widget* en el *EventGraph* del *ThirdPersonCharacter* para que se cree nada más iniciar el juego. Para mostrarlo en la pantalla del jugador, se utiliza la salida de esa función y se añade como entrada a la función *Add to Viewport*.

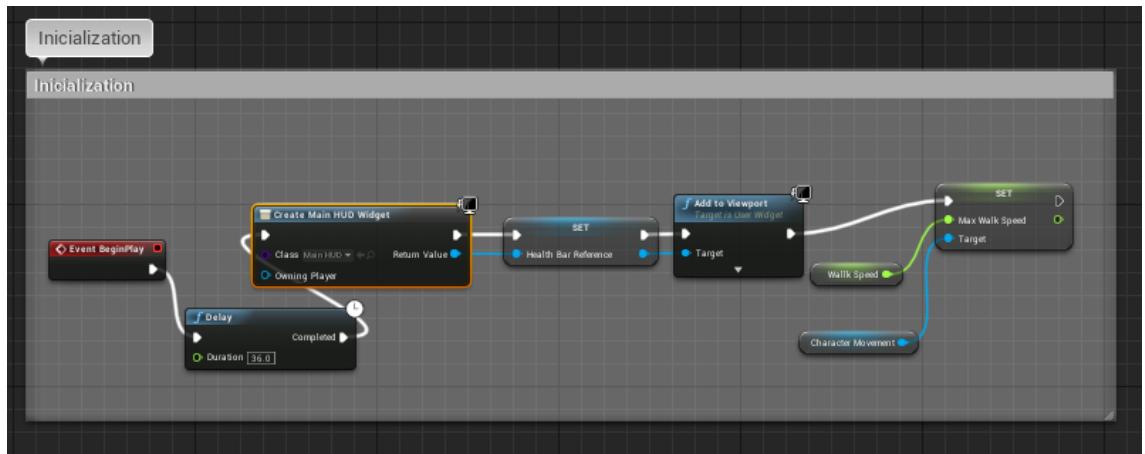


Imagen 3.c.4: Inicialización de parámetros al iniciar el juego (en el EventGraph del ThirdPersonCharacter)

d. Menú principal

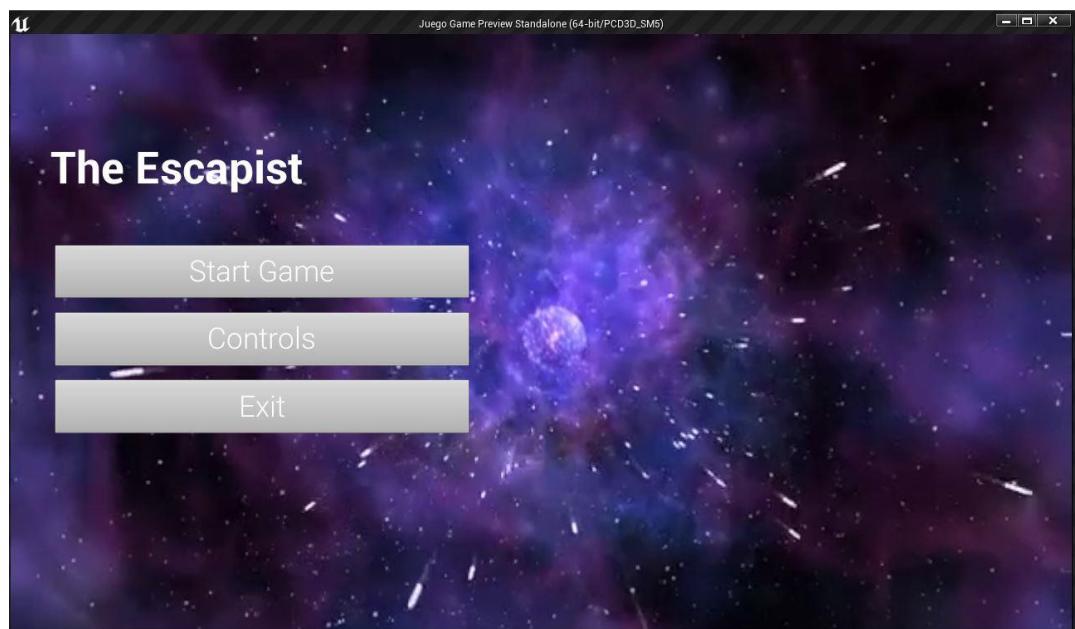


Imagen 3.d.1: Vista del menú principal

En cuanto a la creación del menú principal, he creado otra escena distinta a la escena en la que se encontrará el mapa que contiene el juego principal, y que contendrá un widget con varios botones y un fondo que será una textura creada a partir de un video. Todos estos materiales se pueden encontrar en *Content > Main_Menu*.

Para crear la textura que servirá como fondo del menú principal a partir de un vídeo, he creado un objeto *MediaPlayer* en el que he introducido el clip en bucle (*Loop*) y he generado el material y a partir de este último, la textura animada.

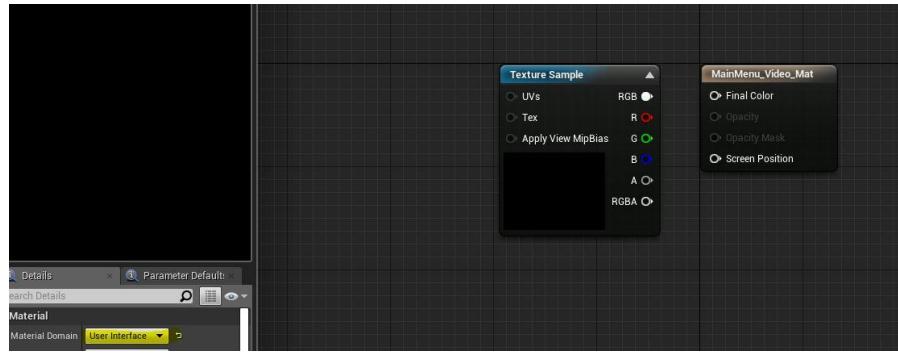
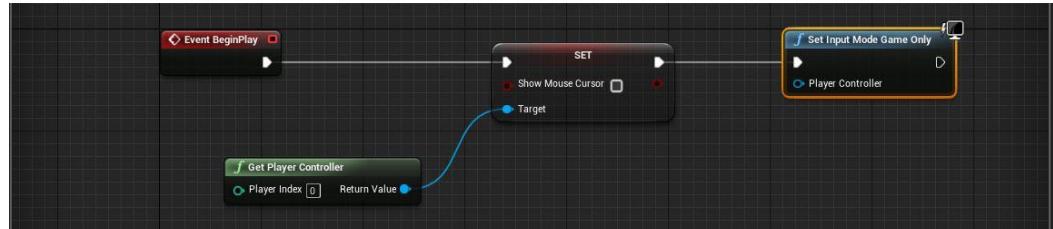


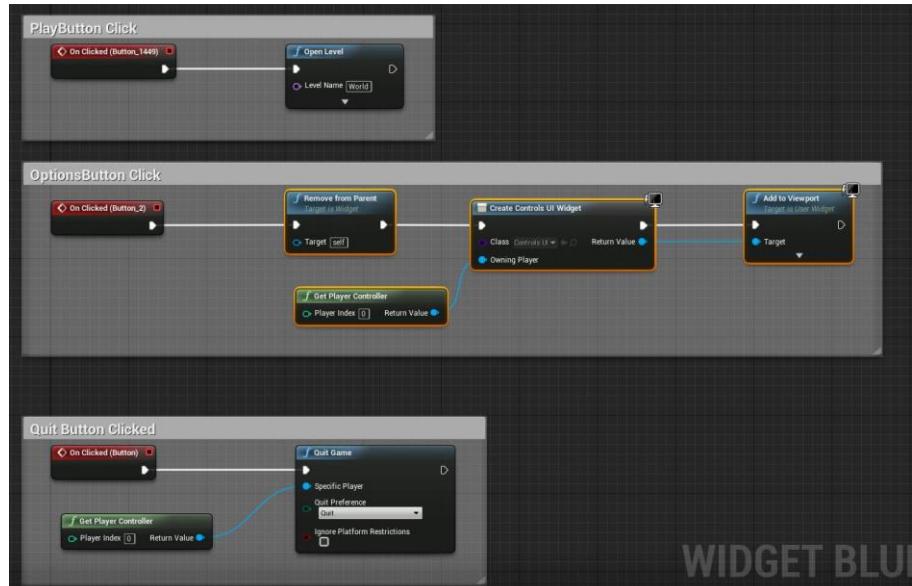
Imagen 3.d.2: Contenido de la textura del material a partir del video

En cuanto al control de los botones, por una parte, permitiremos al jugador hacer uso del puntero del ratón en este nivel para la interacción de los botones, como se puede ver en la imagen 3.d.3.



En cuanto a los botones se mostrará su funcionamiento en el blueprint del nivel (*Level Blueprint*) descrito en la imagen 3.d.4. Básicamente, cuando se seleccione el botón *Start Game* cargará el nivel con el mapa del juego mediante la función *OpenLevel*. El botón *Options* permitirá cargar otro widget distinto a la inicial con la función *Remove from Parent*.

Y, por último, el botón *Exit*, cancelará la ejecución del juego permitiendo salir del mismo.



WIDGET BLUE

Imagen 3.d.4: Control de los botones del menú del juego

Otro detalle importante, es seleccionar en las Preferencias del juego, que el nivel que se cargará por defecto cada vez que iniciemos el juego será el nivel referente al menú principal. Así como para modificarlo, en caso de hacer algún ajuste

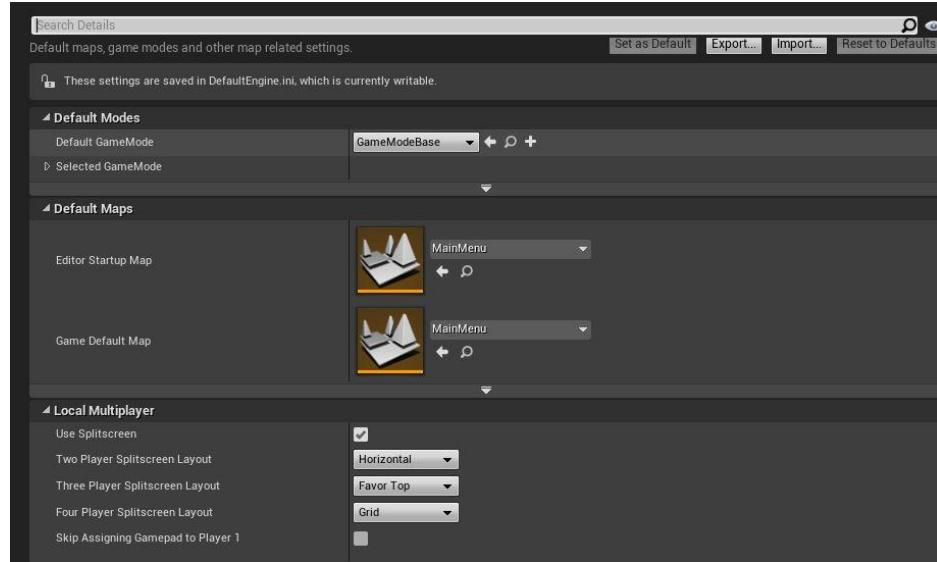


Imagen 3.d.5: Preferencias del juego para indicar el nivel de inicio por defecto

4. Características

A continuación, se detallarán las características de los objetos utilizados en el juego, ya estén relacionados o no con el jugador, como los enemigos, objetos colecciónables, o los mapas creados.

a. Personaje principal

i. Sockets y equipar/desequipar arma

En cuanto al personaje principal detallaremos varias características, el primero referente al *Skeletal Mesh*. Añadí un socket en la espalda del jugado donde irá instanciada el arma del jugador cuando la recoja del suelo (algo que ya explicaremos más adelante).

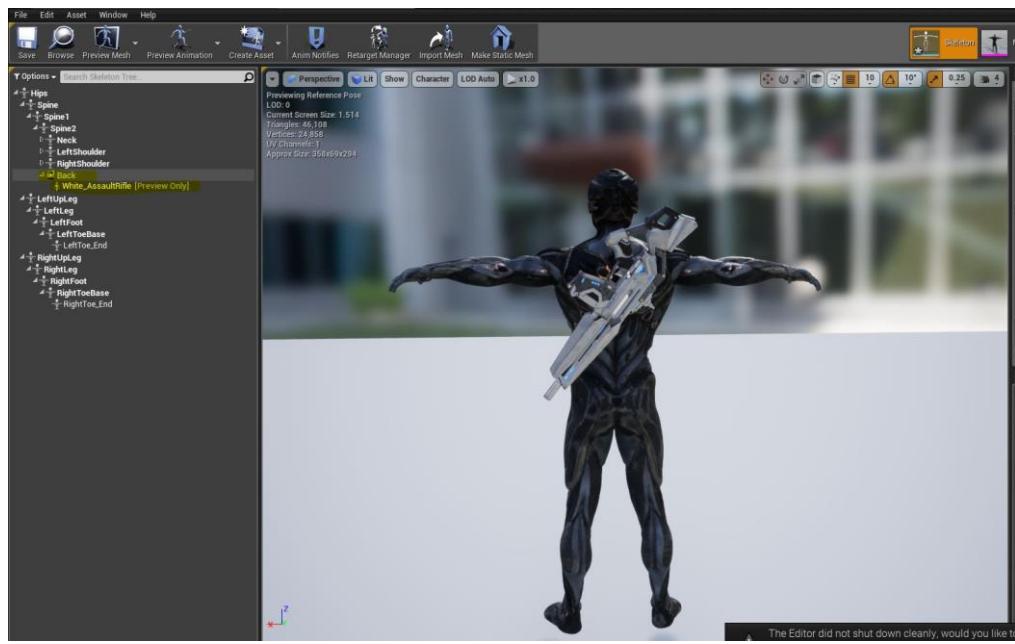


Imagen 4.a.1: Creación de Socket en la espalda del personaje donde irá colocada el arma al cogerla

Haremos lo mismo para asociar el arma a la mano del personaje. Creamos un socket en la mano izquierda y hacemos una preview de cómo se vería el arma con el personaje en la posición en la que tiene equipada el arma, de manera que podemos ajustar la posición del arma y cómo quedará posicionada cuando se instancie en el arma del jugador.



Imagen 4.a.2: Creación de Socket en la mano del personaje donde se asociará el arma

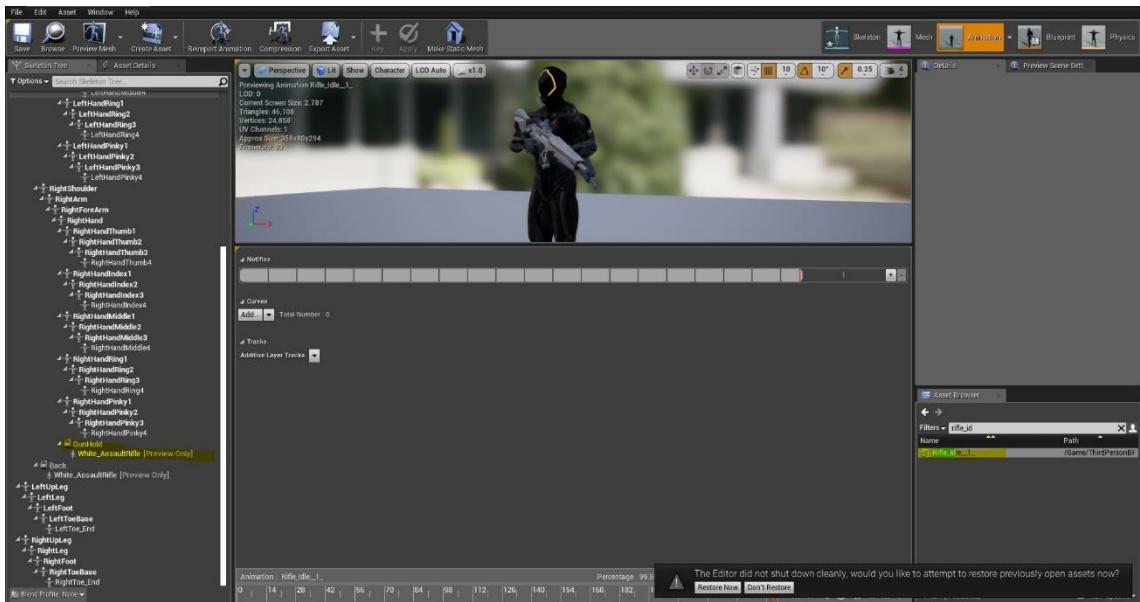


Imagen 4.a.3: Preview de cómo se vería la posición del arma aplicándole la animación de coger el arma estando parado (idle)

En cuanto a la acción de equipar y desequipar el arma, lo primero crearemos un *AnimMontage* que es un objeto del entorno de Unreal que permite añadir otras funcionalidades a una animación (eventos, partículas, etc). Dentro del mismo utilizaremos una animación que podremos llevar a cabo en movimiento para equipar y desequipar el arma de la espalda del personaje.

Esto es, creando un nuevo grupo al Montage llamado *UpperBody* e indicándole que solo queremos utilizar esta característica para la animación, lo cual permitirá que la animación solo se ejecute en la parte superior del esqueleto o *Skeletal Mesh* del personaje mientras que la parte inferior continuará ejecutando la animación del *AnimGraph* del personaje (la de mover si nos equipamos el arma en movimiento).

Posteriormente, y como se muestra en la siguiente imagen lanzaremos un mensaje a un evento *Equip* y a otro *Unequip* en medio de la animación justo cuando pasa la mano de la espalda a la posición del arma para equipar el arma y desequiparla si la animación se hace en orden inverso. Este mensaje hará que se lance un evento de animación en el *EventGraph* del *MyCharacter_Blueprint* que llamará a la función de *Equip* o *Unequip* (dependiendo de la acción que se vaya a realizar) para desinstanciar el arma del socket adherido en la espalda del personaje y adherirla al socket en la mano del personaje si la estamos equipando. O des instanciarla del socket de la mano y devolverla al socket adherido en la espalda para desequipar el arma.

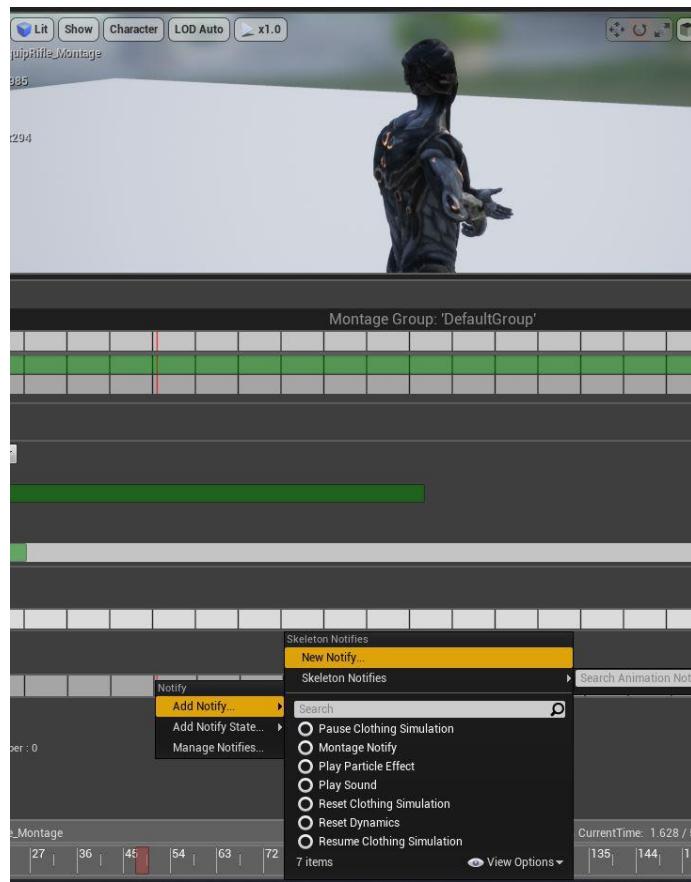


Imagen 4.a.4: Notificación desde el AnimMontage de Equipar/Desequipar el arma para instanciarla en el jugador

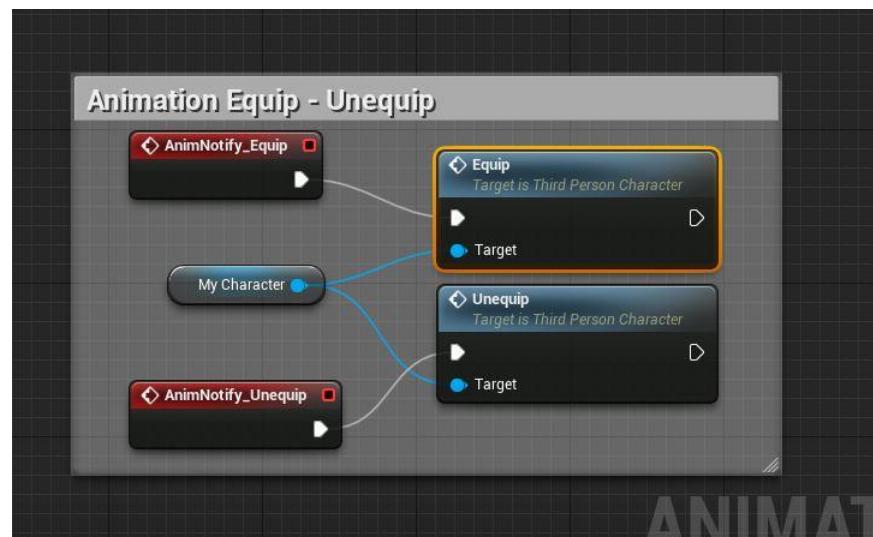


Imagen 4.a.4: Eventos de animación dentro del EventGraph de MyCharacter_Bluperint que llaman a los eventos de Equipar y Desequipar el arma del EventGraph en el ThirdPersonCharacter

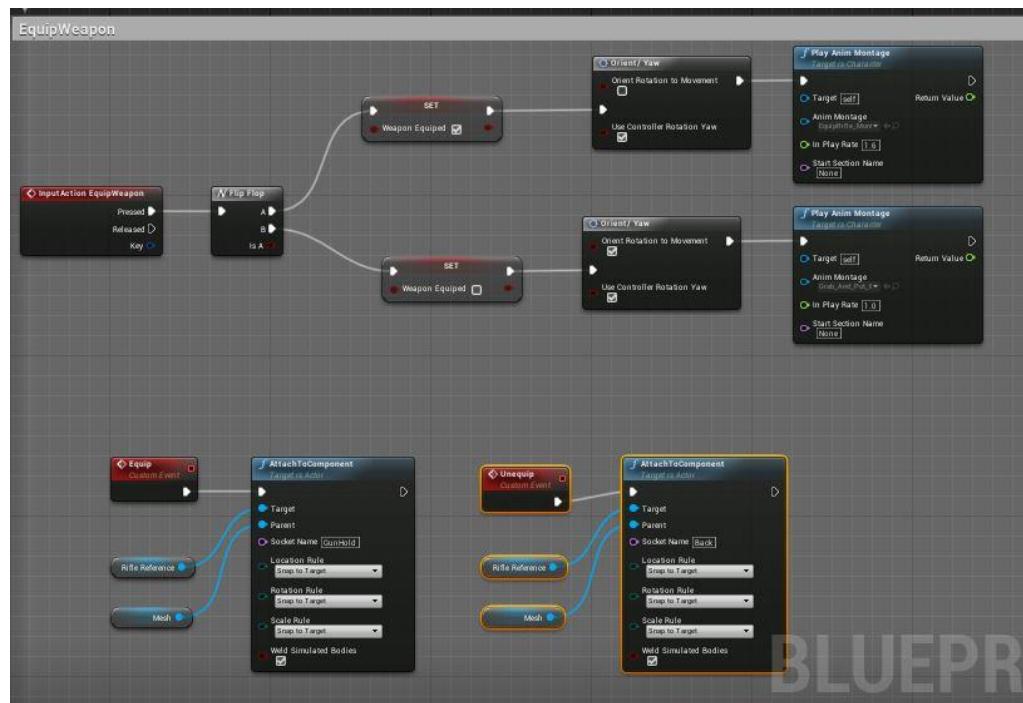


Imagen 4.a.5: Eventos de Equipar y Desequipar el arma del EventGraph en el ThirdPersonCharacter llamados

ii. Recoger el arma

Para recoger el arma no he añadido ninguna animación, pero a continuación detallaré el proceso de recoger el arma y su posterior asociación al socket de la espalda del personaje.

Lo primero añadiremos crearé un blueprint de tipo *Actor* y le añadiré los componentes básicos (Mesh para la forma, Sphere Collision para detectar las colisiones con el objeto, y un Widget que servirá para desplegar un mensaje cuando el jugador se acerque al arma que le dirá como recogerla.

NOTA: el objeto o blueprint del arma se llama *RiflePickup_Item* y se encuentra en *ThirdPerson_BP > MyCharacter > Weapons*. Es un objeto que hereda de *Master_Item_Pickup* ya que mi intención inicial era hacer más de un arma, pero lo acabé descartando. Aunque así logramos una mayor abstracción de las características del objeto.

Posteriormente, una vez añadido a la propiedad Mesh el esqueleto e instancia del arma, que descargué del bazar de la tienda de Epic Games en el paquete "*SciFiWeapLigth*", cuya carpeta contiene todas las texturas, apariencias y físicas del arma.

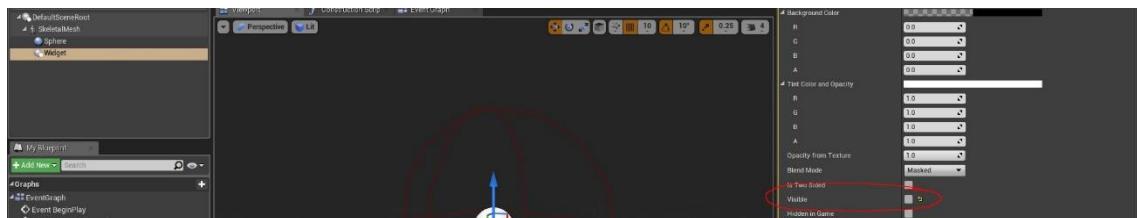


Imagen 4.a.ii.1: Propiedad Visible del widget del objeto del arma desactivado para que solo aparezca cuando se acerque el personaje

En cuanto a los eventos de interacción con el blueprint del arma, caben destacar varias cosas. La primera es que añadí dos eventos para la colisión del personaje con el radio de colisión del arma con los eventos *Begin Overlap* y *End Overlap* para detectar si el personaje está cerca o no y así mostrar el widget con el mensaje indicativo que había mencionado anteriormente.

El otro, el más importante, es cuando se dispara el evento asociado a la acción de interacción con un objeto a través entrada o *input* de teclado (la tecla E), como ya se mostró en la introducción de este informe y que disparará el evento de recoger el objeto del suelo *Pickup* (que se encuentra en el EventGraph del ThirdPersonCharacter).

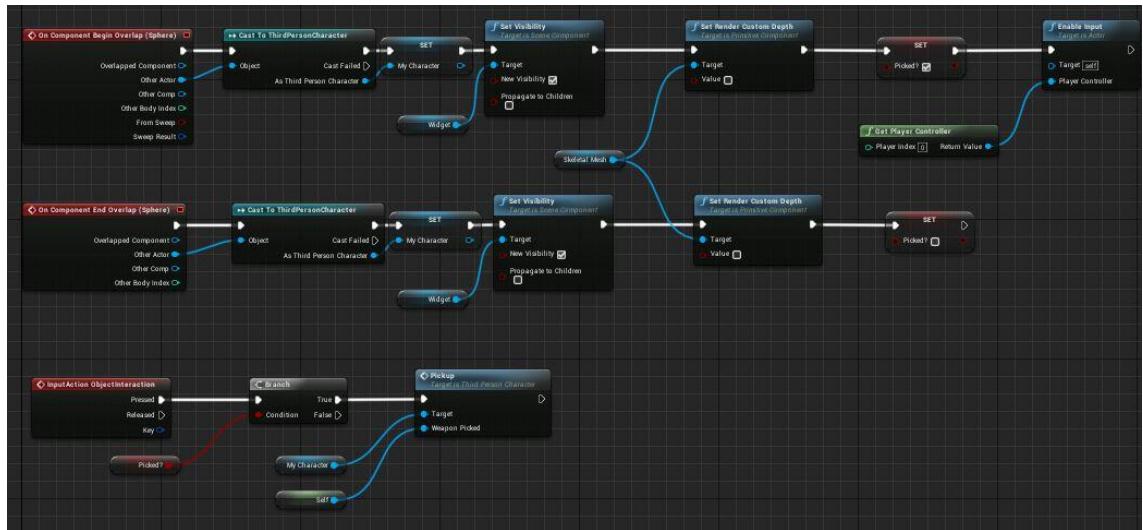


Imagen 4.a.ii.2: Blueprint del objeto creado del arma

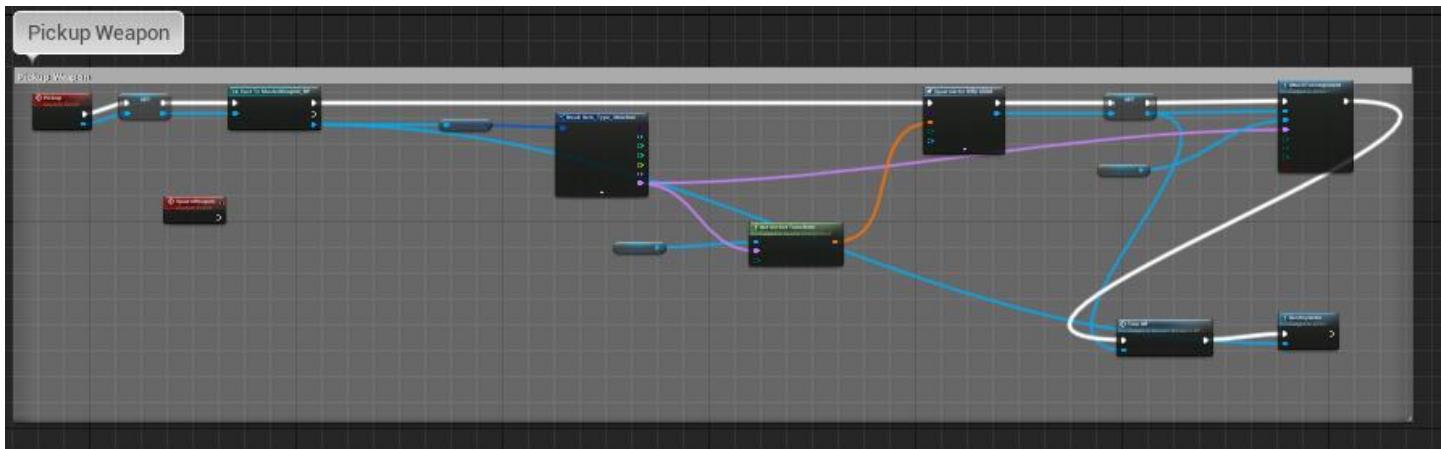


Imagen 4.a.ii.3: Evento de recoger el arma en el EventGraph del ThirdPersonCharacter

iii. Muerte del personaje

En cuanto a la muerte del personaje se mediará por medio de un evento *TakeDamage*, que se disparará cuando el enemigo choque con nosotros y realice la animación de pegar, bajando la vida del jugador y reproduciendo una animación de muerte en el estado *Die* en el *AnimGraph* (como se puede ver en la imagen 3.b.1) en caso de que la variable *IsDeath?* sea cierta y

destruyendo al actor para que los enemigos no vayan a por él ni pueda moverse una vez su vida llegue a 0. Y añadiendo el widget de fin de juego con el número de bajas totales realizadas.

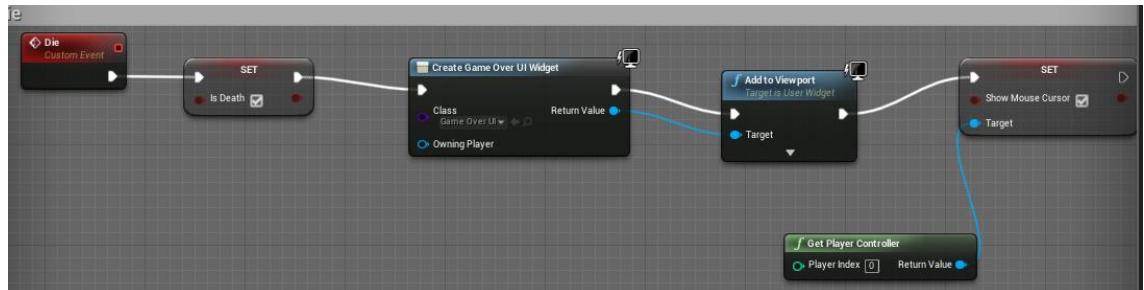


Imagen 4.a.iii.1: Evento de muerte en el EventGraph del ThirdPersonCharacter



Imagen 4.a.iii.2: Widget para la pantalla de muerte

b. Arma

i. Propiedades y características

Para las propiedades del arma principal que usará el personaje, primero crearemos un blueprint *Master_WeaponBP* (en *Content > ThirdPersonBP > Weapons*), que será la clase padre de la que herede el blueprint *Rifle_Child*. Esta abstracción la hice principalmente porque en un principio tenía pensado añadir más de un arma.

Todas las funciones que se mostraran a continuación, a no ser que se especifique lo contrario pertenecen al blueprint *Master_WeaponBP*.

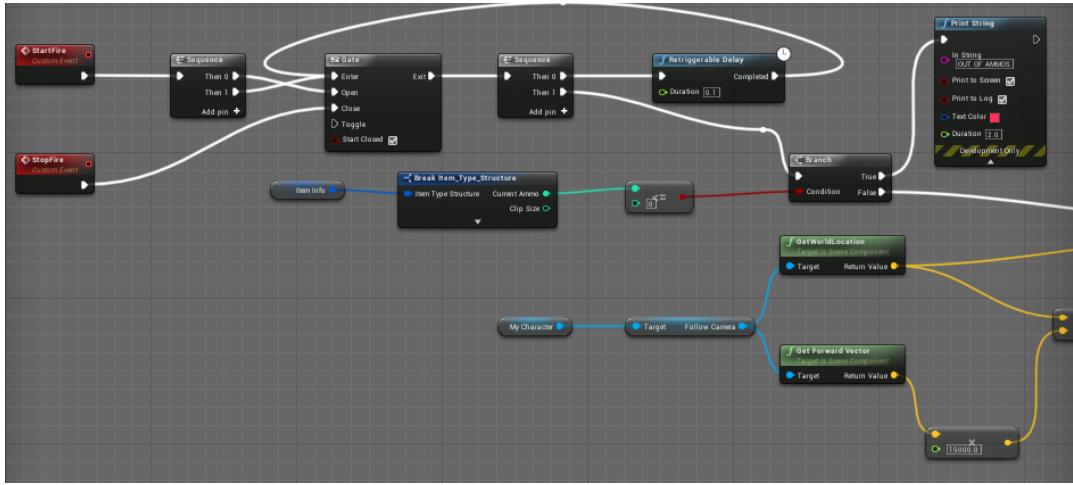


Imagen 4.b.i.0: Obtención de los vectores de traza de la cámara

Dentro del blueprint anterior explicaré varias cosas para tener en cuenta sobre el control del arma y la acción de disparo. Primero, hay que explicar que la acción de disparo se realizará en función de una traza o línea que crearemos en referencia a la cámara del juego. Para ello con las funciones *GetWorldLocation* y *GetForwardVector* conseguiremos la posición de la cámara detrás del jugador y la posición en forma de vector de posiciones que forma una línea invisible hacia el frente de la cámara, que será el punto final de la línea o traza que seguirán los disparos cuando se creen (de ahí que lo multiplique por un número muy grande, dependiendo de la distancia que queramos que lleguen los proyectiles de disparo).

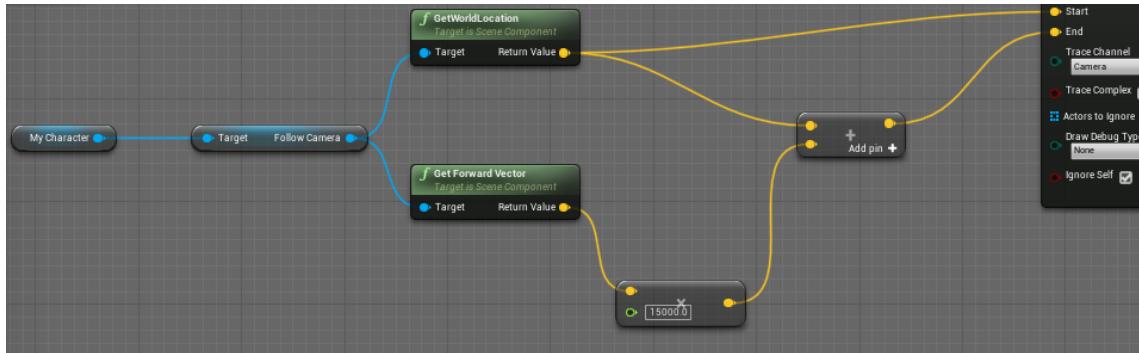


Imagen 4.b.i.1: Obtención de los vectores de traza de la cámara

Para crear la traza o línea utilizaremos la función *LineTraceByChannel* con un vector de posición de entrada y otro de salida que formarán el punto inicial y final de la línea o traza de disparo y como punto de la traza pondremos de referencia a la posición cámara. El valor devuelto será un valor booleano indicando si la línea se ha activado o no.

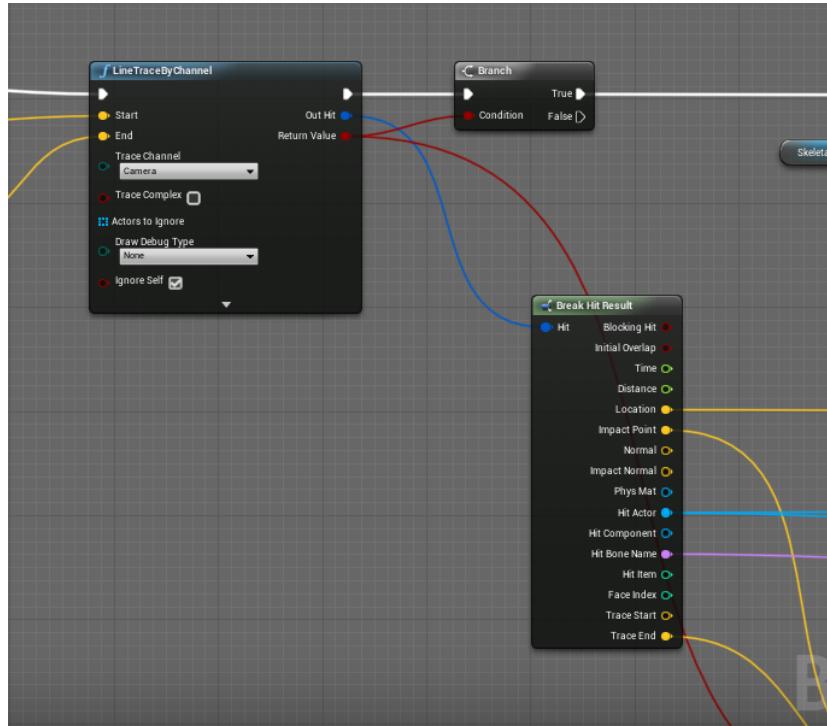


Imagen 4.b.i.2: Línea de trazado para el disparo y función de impacto del disparo

Para ejecutar la opción de disparar el arma, utilizaremos los inputs definidos al principio del proyecto en la imagen 1.3. Cuando el jugador accione el botón asociado a la acción de disparo (*Fire*), mientras lo presione se llamará al evento *Start Fire* siempre y cuando tenga el arma equipada y el arma haya sido recogida del suelo (de ahí la función *IsValid*). En caso de que suelte o libere el botón de acción de disparo, disparará el evento *Stop Fire*.

Ambos eventos son los que comienzan los eventos vistos anteriormente (creación de la traza, impacto del proyectil, etc). Como se puede ver en la imagen 4.b.i.0.

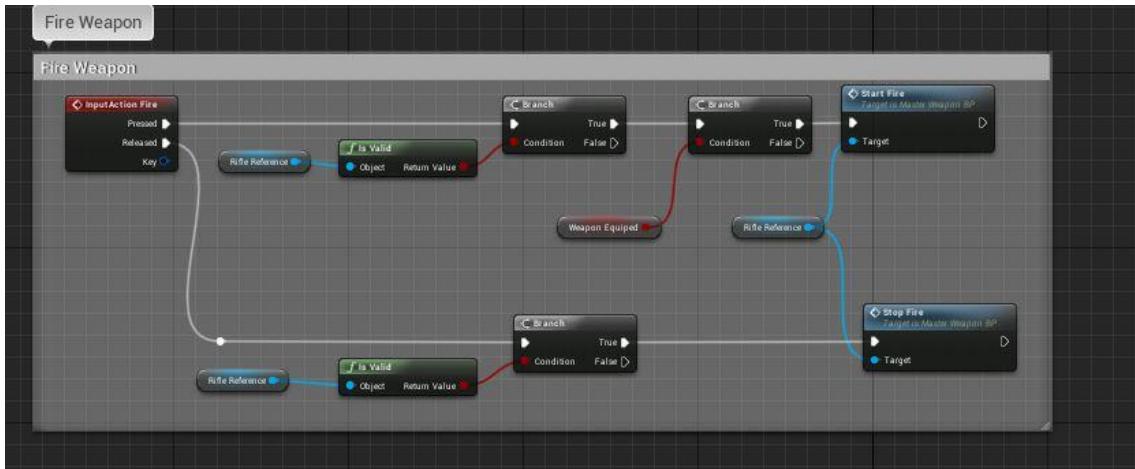


Imagen 4.b.i.3: Evento de Disparo del arma y disparo de los eventos StartFire y StopFire

A continuación, en las siguientes imágenes podemos ver el evento *WeaponReload* para el control de la carga de la munición, el cual básicamente reproducirá un *AnimMontage* con la animación de recargar y hará comprobaciones y actualizaciones con respecto a la cantidad de munición restante, para saber si cuando dejar de efectuar el disparo del arma si el personaje se queda sin munición. Así como toda la configuración necesaria para sacar la cantidad necesaria de la munición máxima para el cartucho actual del arma.

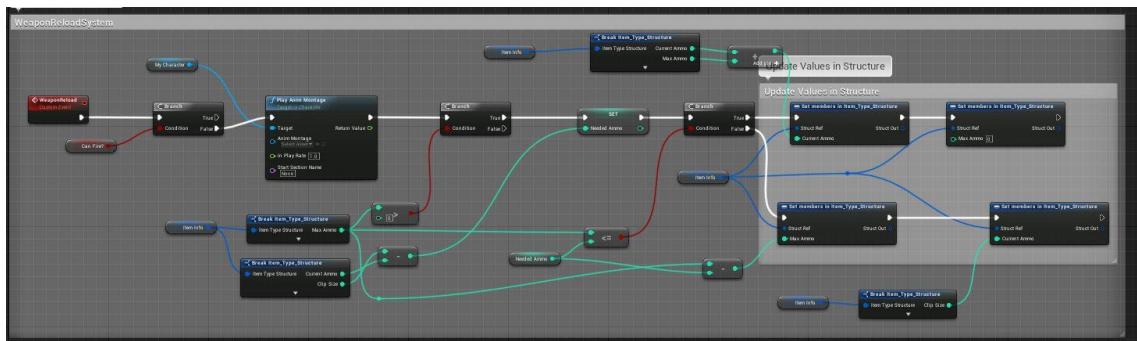


Imagen 4.b.i.4: Evento de Control de carga de munición del arma

El evento *Still Loading* se encargará de todo el procesamiento necesario para comprobar si el arma esta cargándose o no y para saber si hay munición aún para recargar el cartucho y ejecutar las animaciones, acciones, etc. de los eventos *AutoReLoad* y *WeaponReload*.

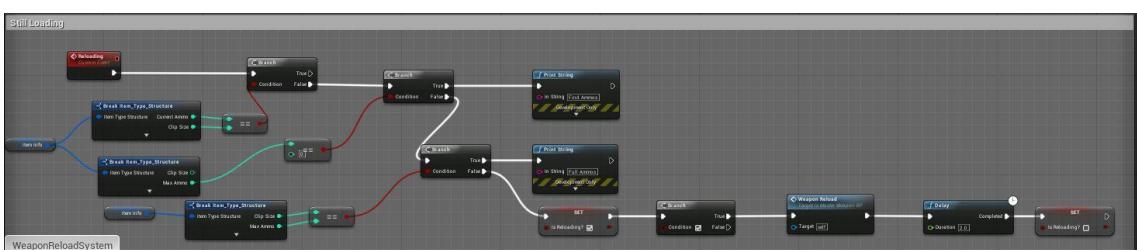


Imagen 4.b.i.5: Evento de control de munición total y máxima

Volviendo a la creación de la línea o traza utilizaremos las funciones *Select Vector*, *Find Look at Rotation* y *Make Transform* para obtener la rotación y evitar que cuando el personaje se gire o cambie de posición los disparos no tengan un comportamiento anómalo disparándose o creándose en otras posiciones, si no que se creen en una posición relativa al movimiento del propio personaje y vayan de frente a donde le personaje está mirando. Con la función *Get Socket Location* indicaremos el socket que crearemos posteriormente en el *Skeleton Mesh* del arma, que será desde donde se origine el fogonazo y desde donde se crearán los proyectiles.

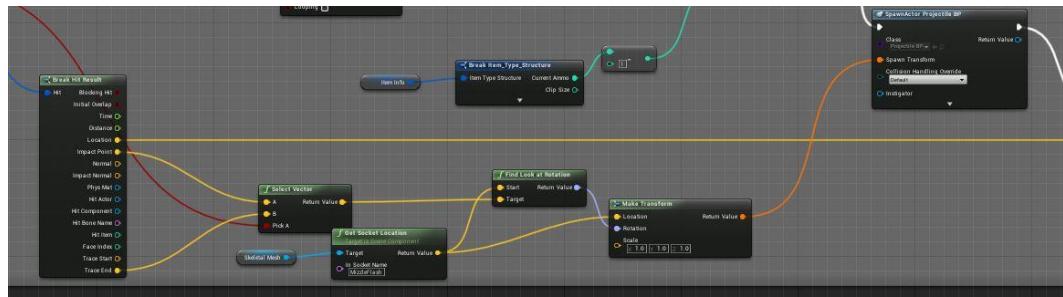


Imagen 4.b.i.6: Generación del proyectil o disparo en la traza

A parte, como se ve en la siguiente imagen, se creará la animación con el fogonazo del arma añadiendo la animación con partículas que ya venía definida en el paquete de armas. Posteriormente, haciendo uso de la estructura descrita en la imagen 1.2, iremos disminuyendo en 1 la munición del cartucho actual del arma hasta que alcance 0, en ese caso se producirá la auto recarga.

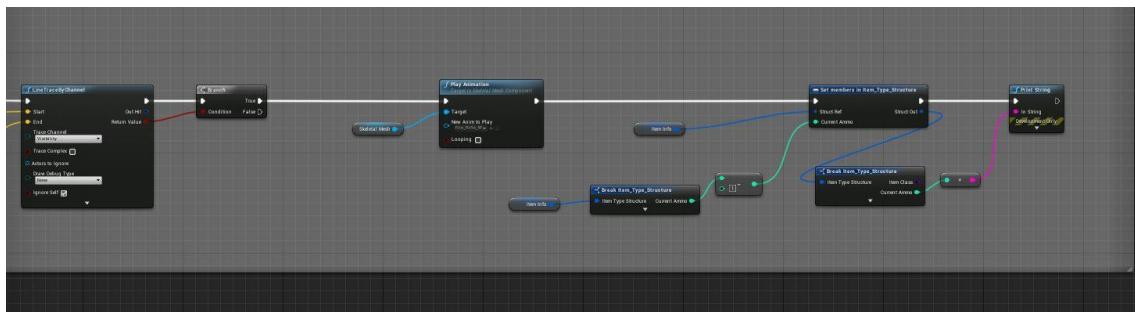


Imagen 4.b.i.7: Disparo y animación de fogonazo en el arma

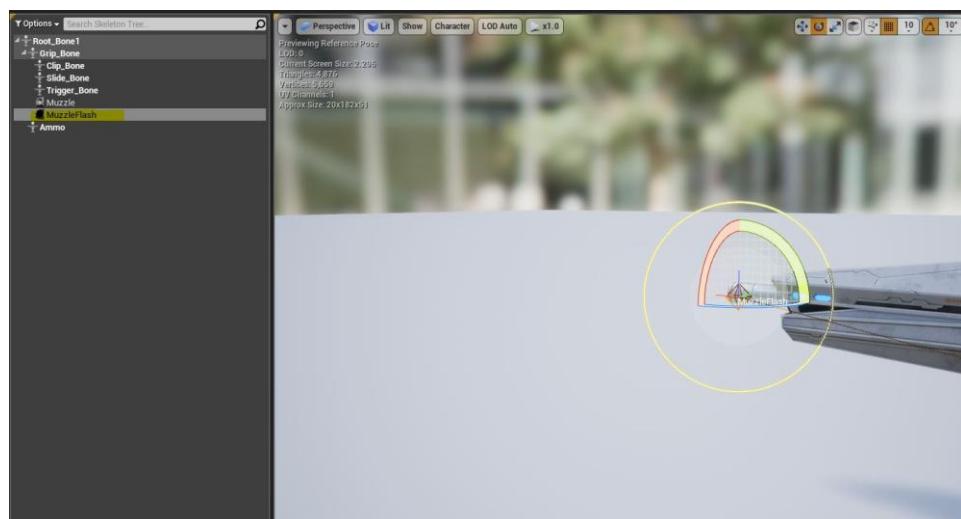


Imagen 4.b.i.8: Creación del socket en el arma de donde se originarán los proyectiles y donde se producirán las partículas, así como la animación de fogonazo

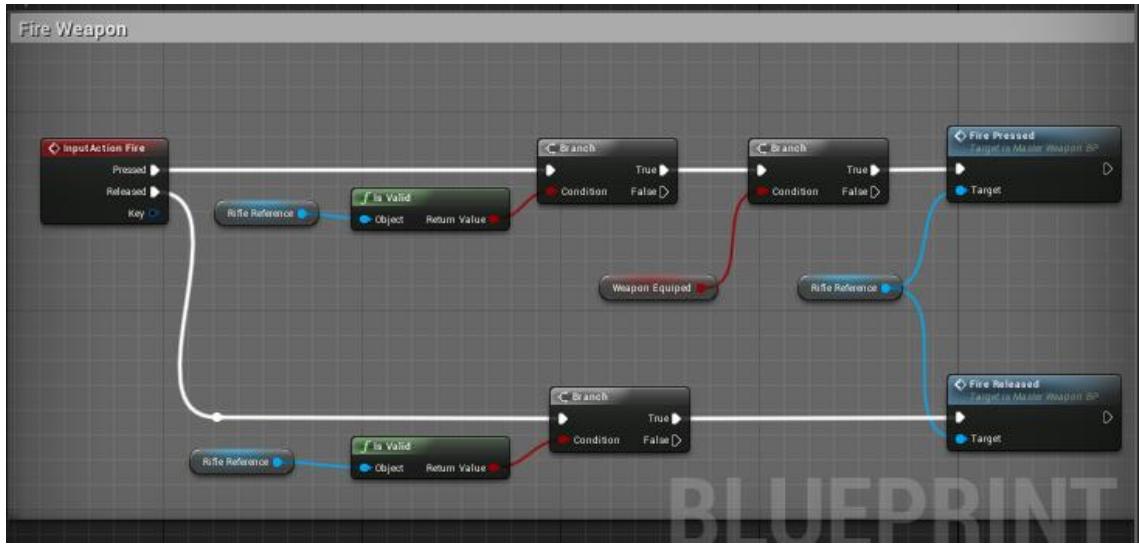


Imagen 4.b.i.8: Input del personaje para disparar desde el EventGraph del ThirdPersonCharacter

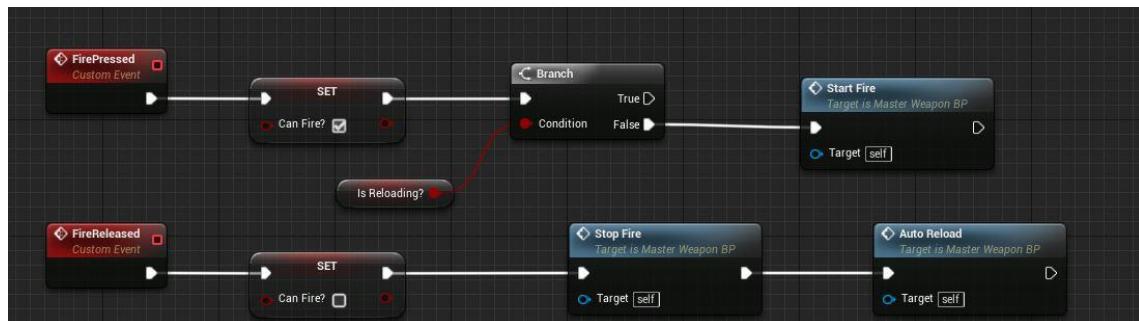


Imagen 4.b.i.9: Disparo de los eventos al dejar de disparar y cuando autorecarga el arma

El evento relativo a la autorecarga se disparará cada vez que se deje de disparar, comprobándose si la munición del cartucho actual del arma es igual a 0, en ese caso se pasará a realizar la animación de recargar y a restarle a la capacidad máxima de la munición la cantidad total del nuevo cartucho.

Para que el jugador no pueda disparar mientras se esté recargando el cartucho o si se ha quedado sin munición se utilizará la variable *Can Fire?* Que indica si el arma está disponible para disparar o no.

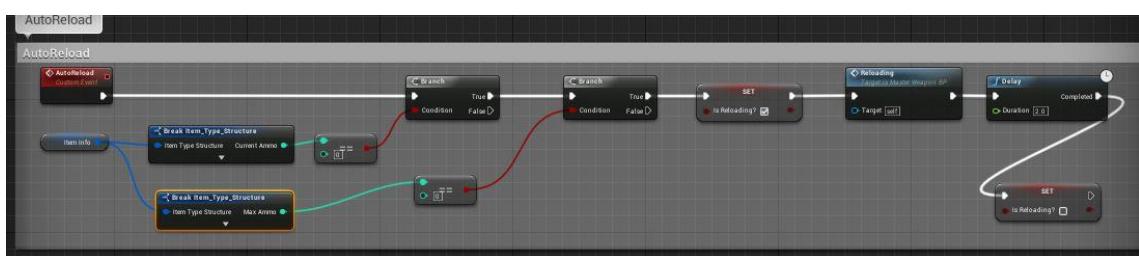


Imagen 4.b.i.10: Evento de autorecarga del arma

Por último, cada vez que instanciamos un proyectil que impacta contra el enemigo, este proyectil queda creado en el mundo indefinidamente. Para

que los proyectiles se destruyan si no impactan contra ningún objeto o actor con físicas, le añado un *Life Span* que es un periodo de vida en segundos que quedará instanciado un objeto antes de ser destruido.

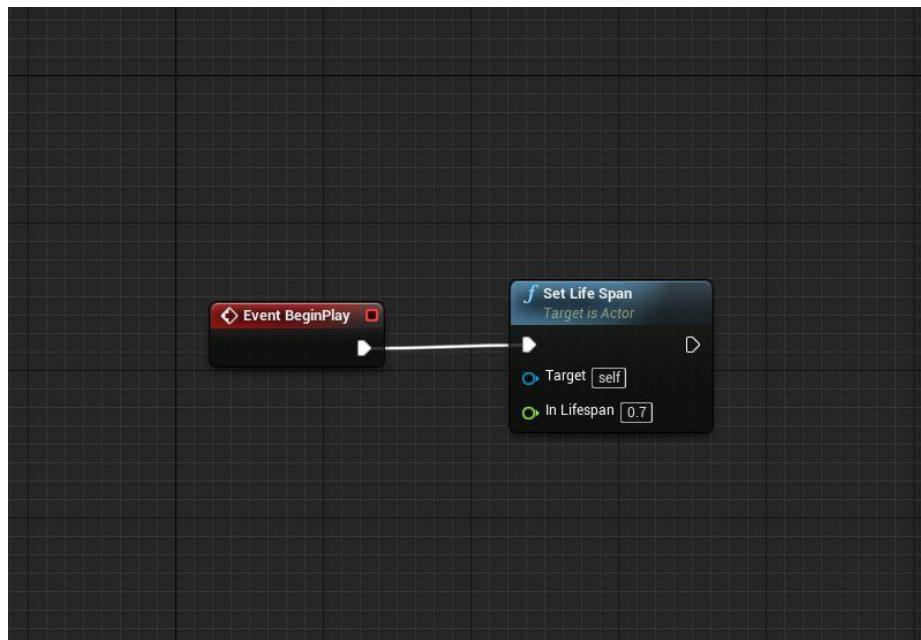


Imagen 4.b.i.11: En Projectile_BP, establecer el tiempo de vida del proyectil

ii. Zoom

He implementado, además de un *AIM Offset* para poder girar la cámara con el arma apuntando y que la animación del personaje mire hacia la dirección y posición hacia la que apunta la cámara en cualquier ángulo, la posibilidad de hacer *zoom* cuando el arma está equipada y tener una mejor visión a la hora de disparar con el jugador mediante el uso de timelines, de manera que el cambio de una posición de cámara a otro se vea más suave y no tan inmediato y forzado.

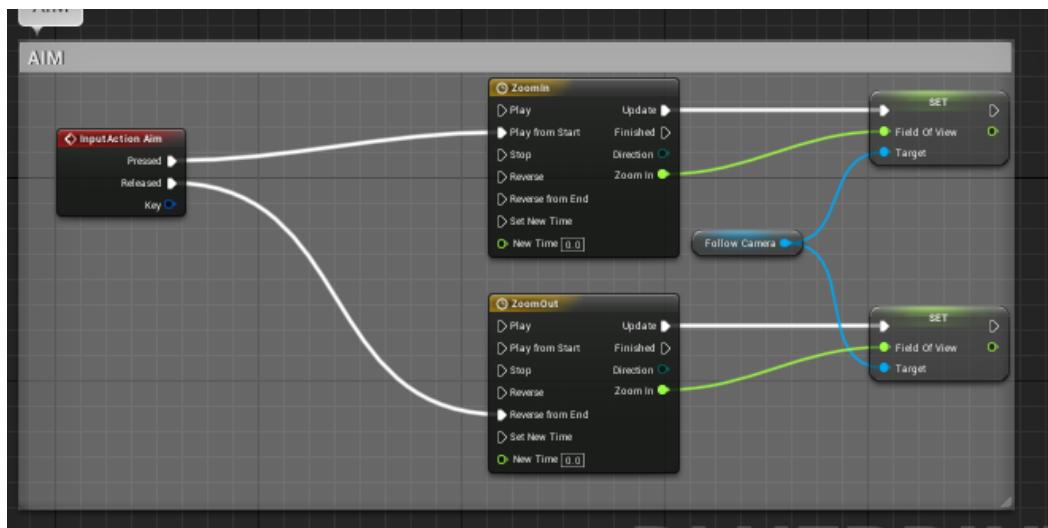


Imagen 4.b.ii.1: Evento de apuntar en el EventGraph del ThirdPersonCharacter

Para ello he creado dos timelines, aunque realmente se trata del mismo, uno para acercar la cámara cuando se pulsa el *Input* de *Aim* y otro para cuando deja de pulsarse. En lo referente a que ambos son el mismo, la función de timeline de *ZoomIn* pasará de un valor de clave y=90 que es el valor por defecto de amplitud de la camara en tiempo x=0, a un valor de clave y=40, que es un valor de amplitud más cercano llevando a cabo la transición en un tiempo de 4 segundos, x = 0.4.

Para el *ZoomOut* utilizaremos exactamente la misma función de timeline pero, como se puede apreciar en la imagen 4.b.ii.1 , cuando dejamos de pulsar el botón de apuntado activamos la función de timeline en orden inverso a como está definida. Es decir, realizando la transición de valor y=40 de vuelta al valor y=90.

La salida de estos timelines será la modificación de la vista de campo de la cámara del juego, con una transición más suave.

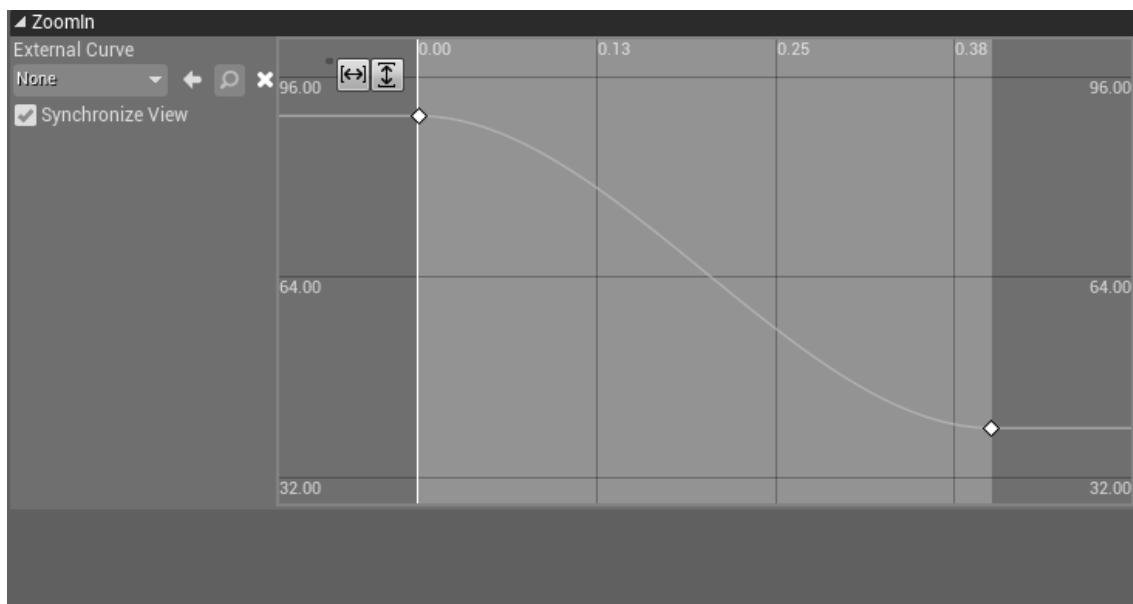


Imagen 4.b.ii.2: Función timeline para aumentar o disminuir la cámara

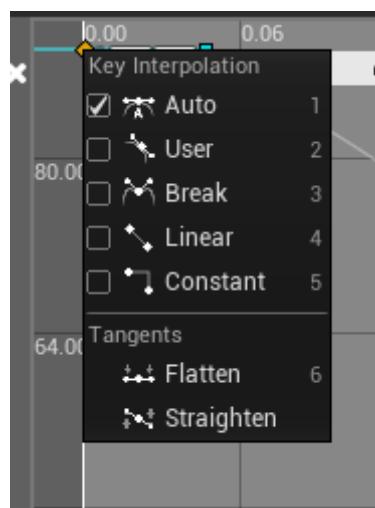


Imagen 4.b.ii.3: Propiedad automática para el cambio de valores que es necesario activar

NOTA: también he hecho uso de los timelines para las puertas del inicio del juego que se abren y se cierran automáticamente cuando el jugador está cerca. El procedimiento de generación de este objeto (en *Content > ThirdPersonBP > Objects > Door_BP*) es, añadirle un componente *Sphere Collision* para saber cuando el personaje está cerca y utilizar sus eventos *BeginOverlap* y *EndOverlap* para abrir o cerrar las puertas automáticamente cuando pasa el personaje. Después, en el EventGraph del Blueprint, utilizando estos eventos, utilizar los timelines para realizar la transición suave de cambio de posición en el eje X de las dos puertas automáticas, dejando las componentes Y y Z con sus valores originales.

Es necesario que esta acción se haga solo una vez cada vez que se disparan los eventos con la función *DoOnce*, de lo contrario las puertas se abrirían y cerrarían en bucle cada vez que el jugador estuviera cerca del radio de colisión con el componente *Spehere Collision*.

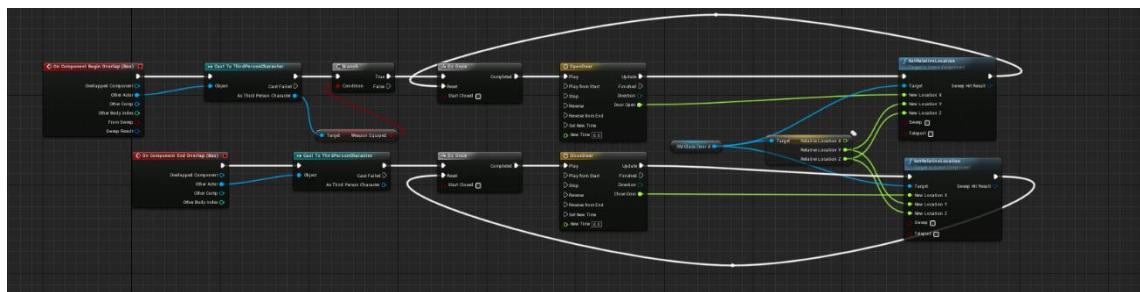


Imagen 4.b.ii.4: Evento del objeto de las puertas automáticas utilizando también Timelines

c. Enemigos

Todo el contenido del enemigo al que se hará referencia se encuentra en la carpeta *Content > ThirdPersonBP > Enemy*.

i. Movimiento y patrulla

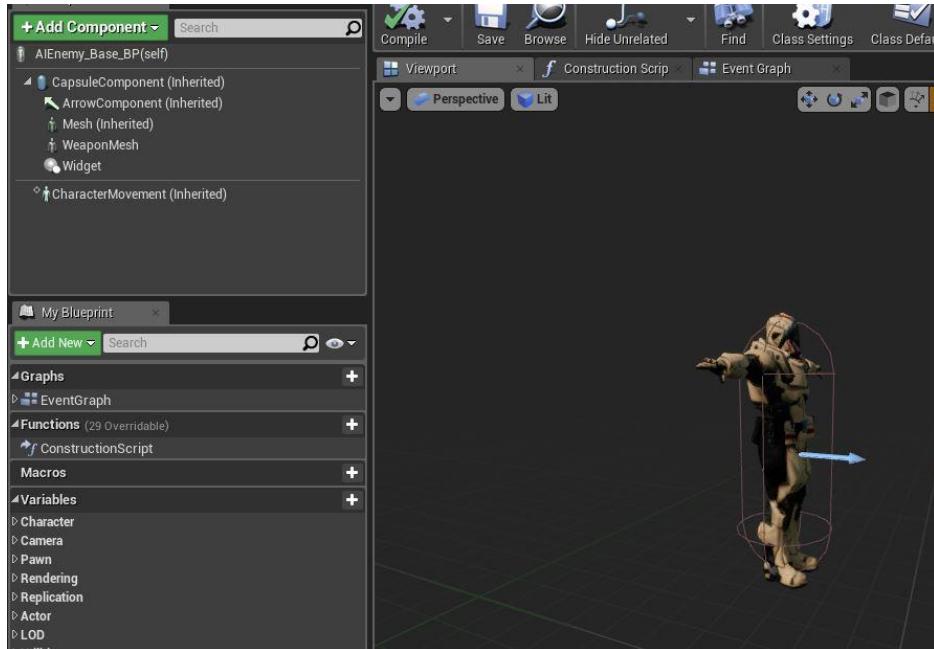


Imagen 4.c.i.1: Componentes del AI_Enemy

Todo el contenido del enemigo al que se hará referencia se encuentra en la carpeta *Content > ThirdPersonBP > Enemy*. Al igual que se hacía referencia en clase y no entraré mucho en detalle, si el enemigo está patrullando entre dos *waypoints*, y ve a través del componente *Pawn Sense* al jugador, lo perseguirá hasta alcanzarlo.

Si lo alcanza, le aplicará un daño bajándole la vida, y ejecutando una de las tres animaciones de ataque de manera aleatoria de su *AnimGraph* como se puede ver en la imagen 4.c.i.9.

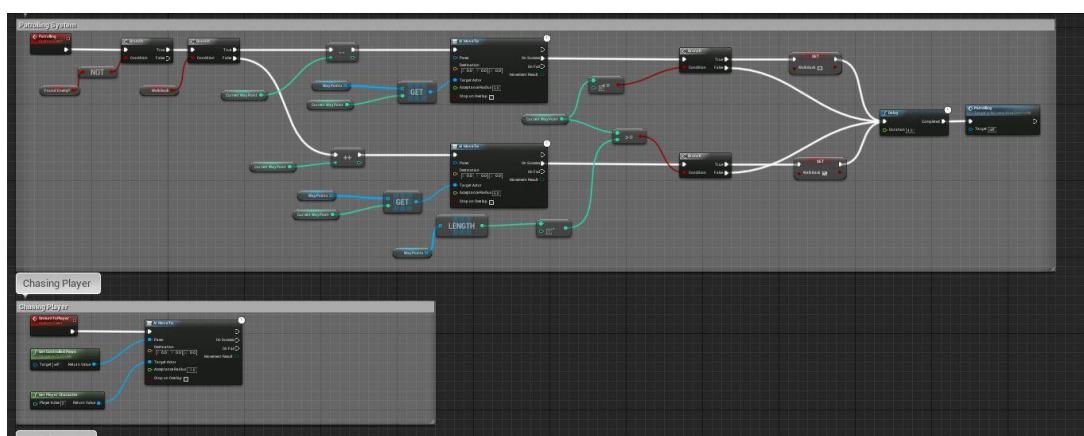


Imagen 4. c.i.2: Movimiento del enemigo

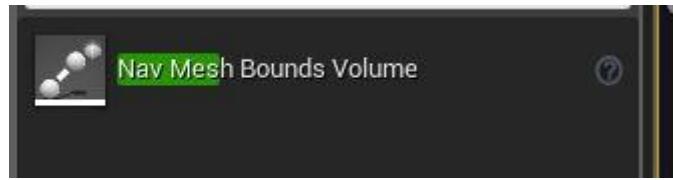


Imagen 4. c.i.3: Creación y adhesión de un Nav Mesh Bounds Volume al juego

Creo un AI Controller donde irán todas las funciones o eventos descritos anteriormente, *FindCharacter* (evento que se lanzará cuando el enemigo tenga en su campo de visión al personaje o se cruce en su camino de patrulla), *Patrolling* (Para mover al enemigo con la función *AIMoveTo* entre los dos waypoints marcados), *ChasingPlayer* (para moverse a través de la función *AIMoveTo* hacia la posición relativa del personaje y alcanzarlo, cogiendo la instancia de posición del personaje principal y pasándosela a esta función)

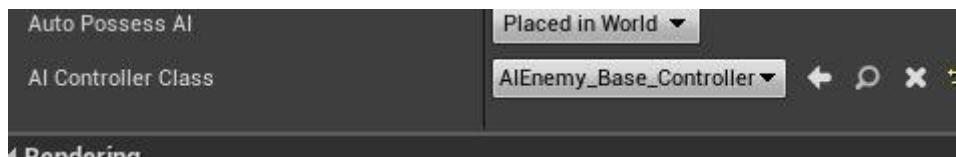


Imagen 4. c.i.4: Añadir el AI Controller del Enemigo

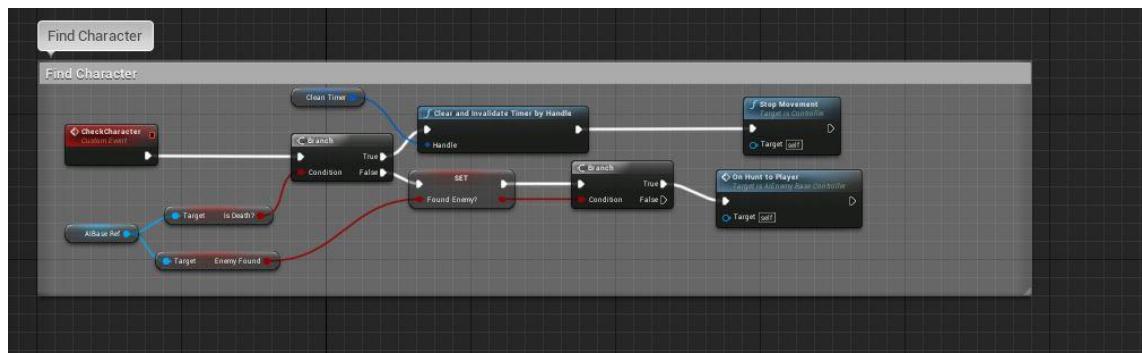


Imagen 4. c.i.5: Movimiento del enemigo cuando encuentra al personaje en el AIEnemy_Base_Controller

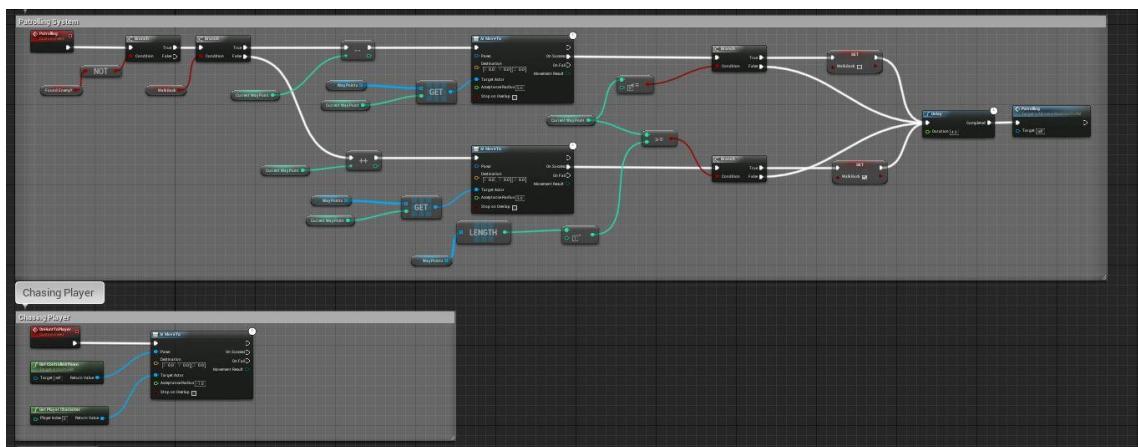


Imagen 4. c.i.6: Movimiento del enemigo cuando persigue al personaje y lo alcanza en el AIEnemy_Base_Controller

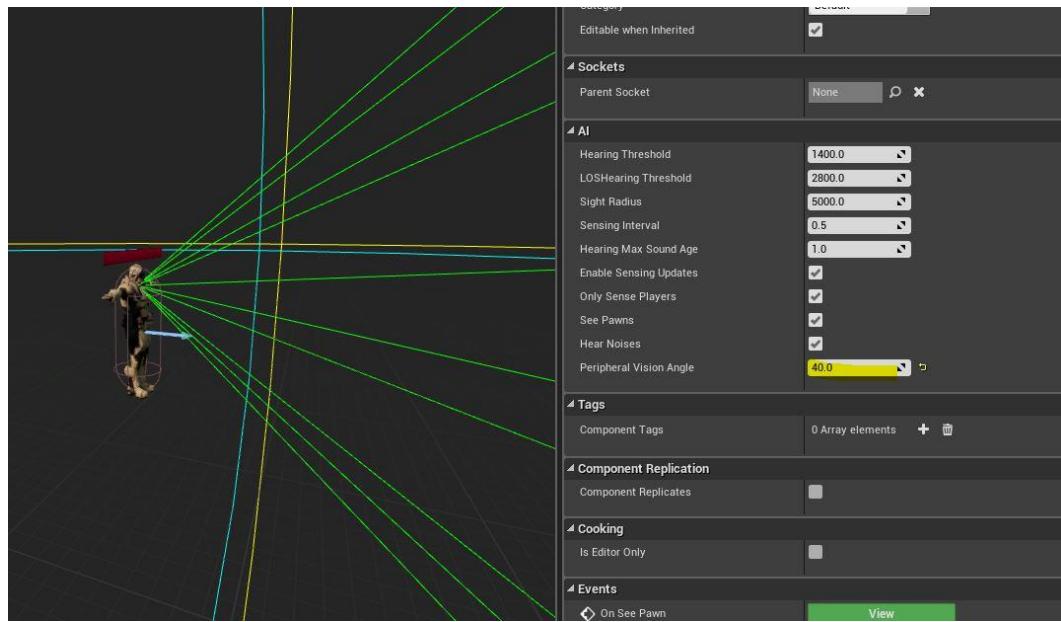


Imagen 4. c.i.7: Indicación del radio de visión del enemigo en el componente Pawn Sense



Imagen 4.c.i.8: Enemigo entre dos waypoints con el Nav Mesh añadido

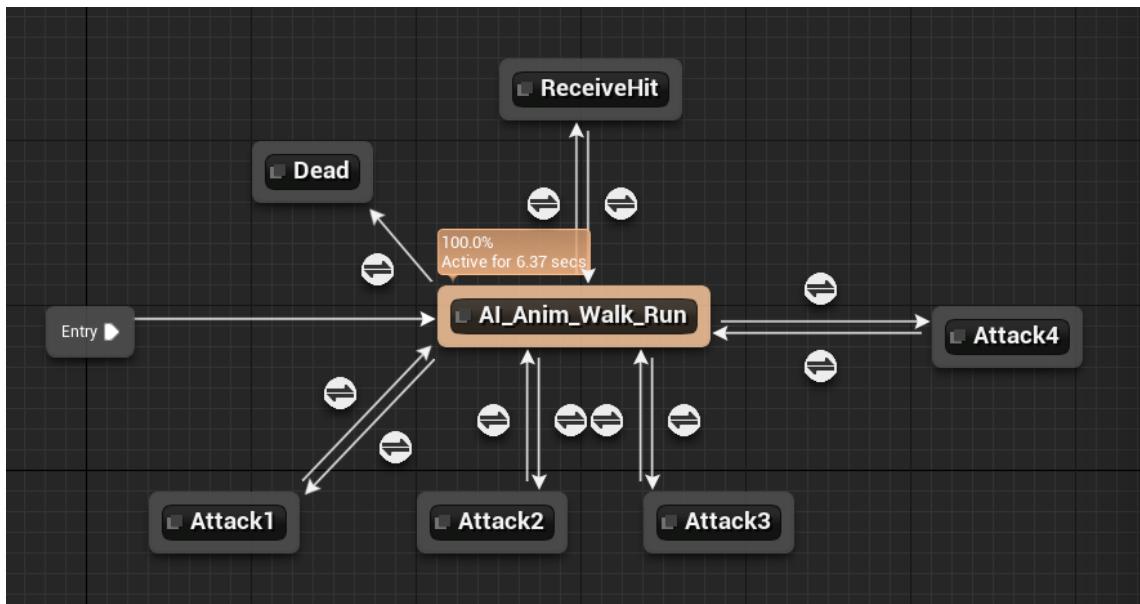


Imagen 4.c.i.9: Grafo de animaciones del enemigo

ii. Aplicar daño

Para aplicar daño a los enemigos, dado que se trata de un combate con armas de fuego (a distancia), lo primero cree un *Blueprint Interface*, que utilizará solo el enemigo cuando reciba un disparo del personaje (el personaje principal no la utilizará porque el enemigo tendrá que dañarle cuando se acerque a él).

Lo primero, en el *AI/EnemyBase_BP*, que es el blueprint principal de control del enemigo, iremos a los ajustes de clase *Class Settings > Interfaces* y añadiremos el componente que acabamos de crear anteriormente.

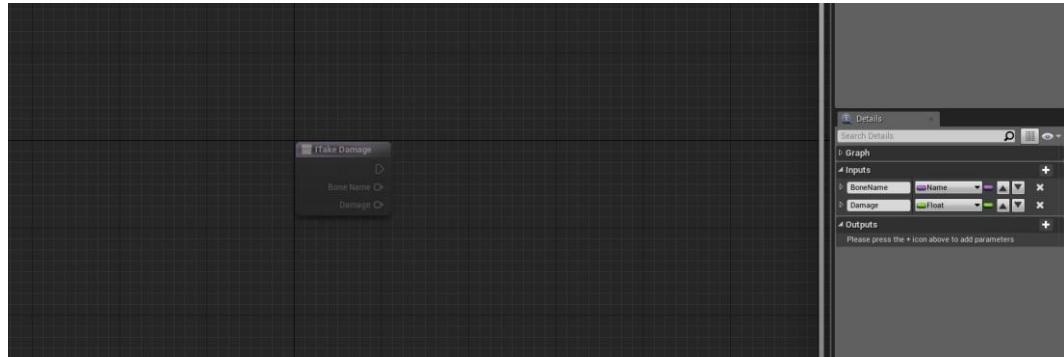


Imagen 4.a.iii.1: Blueprint Interface creado para implementar el daño recibido por los enemigos debido a un disparo

Posteriormente, ajustaremos los valores de colisión del componente *Capsule Collision*. Y en el blueprint de disparo del arma, ya descritos anteriormente, en la función *Break Hit Result* añadiremos una salida al conector *Hit Actor*, que es el actor al que impacta la línea de disparo, que envíe una notificación para que se ejecute el evento del blueprint Interface *TakeDamage* si el actor contra el que impacta es de tipo *AI_EnemyBase_BP*.

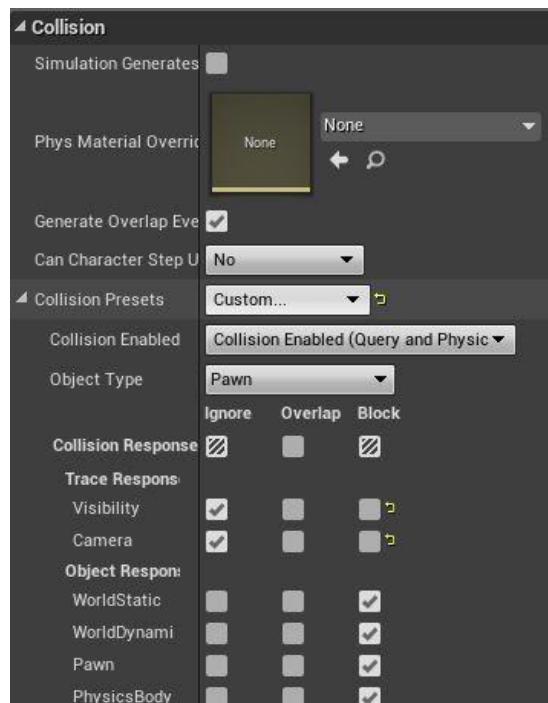


Imagen 4.a..iii.2: Ajustamos la colisión en el Capsule Component del enemigo



Imagen 4.a..iii.3: Blueprint de disparo del personaje principal que cuando impacte con un actor de tipo AI_EnemyBase mandará una notificación para que se dispare el evento

Por último, el evento se disparará quitándole vida al enemigo. Como algo adicional he añadido a través del *Blueprint Interface* una selección por impacto, permitiendo implementar disparos a la cabeza si el disparo impacta contra la cabeza del *Skeletal Mesh* del enemigo, provocando su muerte instantánea. En caso contrario, solo le quitará vida.

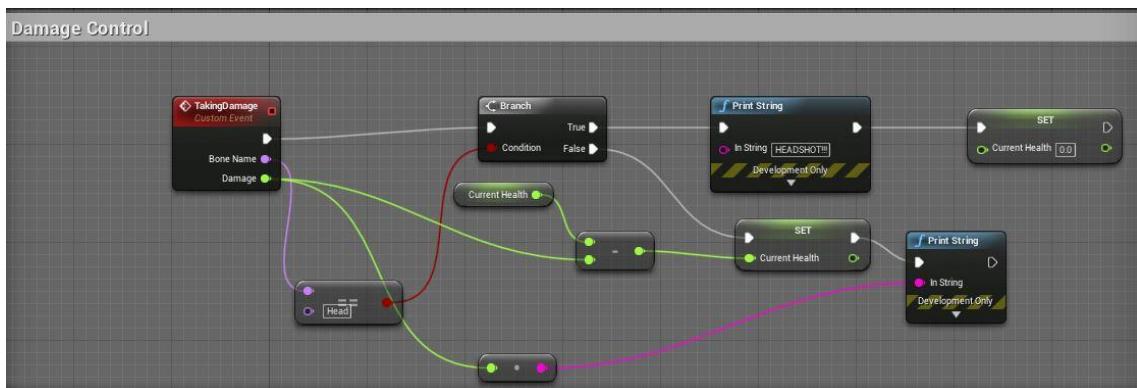


Imagen 4.a..iii.4: Conjunto de funciones de control de daños (I): Se dispara en el blueprint del enemigo el evento de daño

Si el enemigo muere, antes de destruir la instancia del actor, se actualizará el contador de bajas del personaje principal sumándole uno. En caso contrario, solo se activará la variable *Hit*, que en el *AnimGraph* si se detecta activa, reproducirá la animación de impacto.

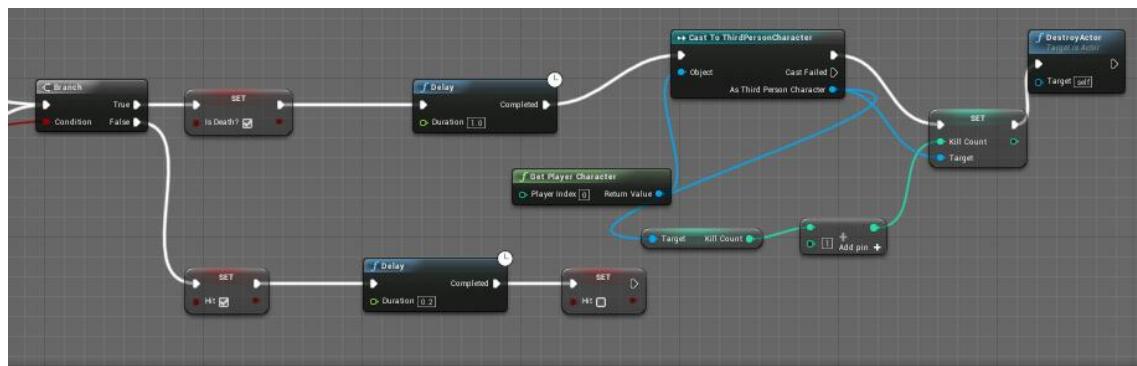


Imagen 4.a..iii.5: Conjunto de funciones de control de daños (II): Incremento del contador de bajas y reproducción de la animación de impacto

d. Mapa

Para el mapa del juego, he utilizado un mapa predefinido de uno de los paquetes del Bazar de la tienda de *Epic Games: Module Sci Fi Season 1 Starter Bundle* con el que pretendía crear el ambiente y temática de ciencia ficción para el desarrollo del juego.

Cabe destacar que el diseño del mapa es uno que venía por defecto en el paquete al que luego yo, posteriormente, he ido añadiendo y quitando elementos, como habitaciones, iluminación, posiciones de los enemigos, paneles interactivos, ... que no venían incluidos como parte del mapa para reducir el tamaño del proyecto.

Como detalle inicial, voy a describir como realicé los títulos de introducción del juego de manera básica. Lo primero, cree otro widget (en *Content > Main_Menu > Intro_Text_UI*) que se cargaría al inicio del nivel del juego, justo antes de tomar el control con el jugador. Este cuenta con varios textos los cuales fui animando mediante el uso de *Timelines* quitándole la componente Alpha al principio del widget (para que se percibiera invisible) y volviéndosela a añadir paulatinamente pasados unos segundos, dando la sensación de cinematográfica.

Lo mismo apliqué con el fondo negro del widget para que fuera desapareciendo cuando se hubieran desplegado todos los textos.

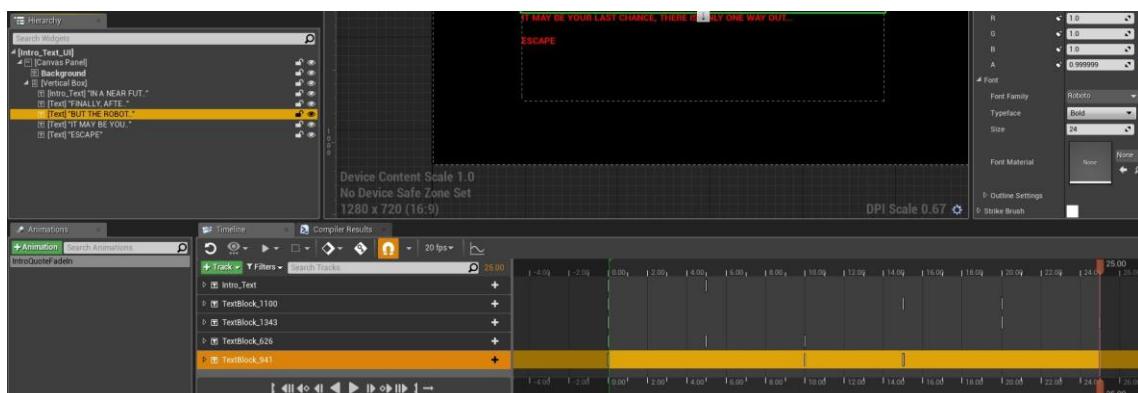


Imagen 4.d.1: Timeline entero de las animaciones sobre los distintos componentes del widget

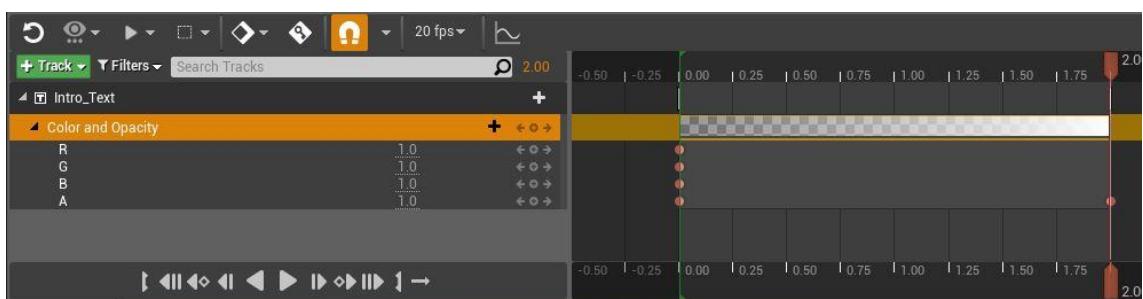


Imagen 4.d.2: Uso de timeline para uno de los textos a animar



Imagen 4.d.1: Vista preliminar del escenario



Imagen 4.d.2: Sala de control para reestablecer la energía

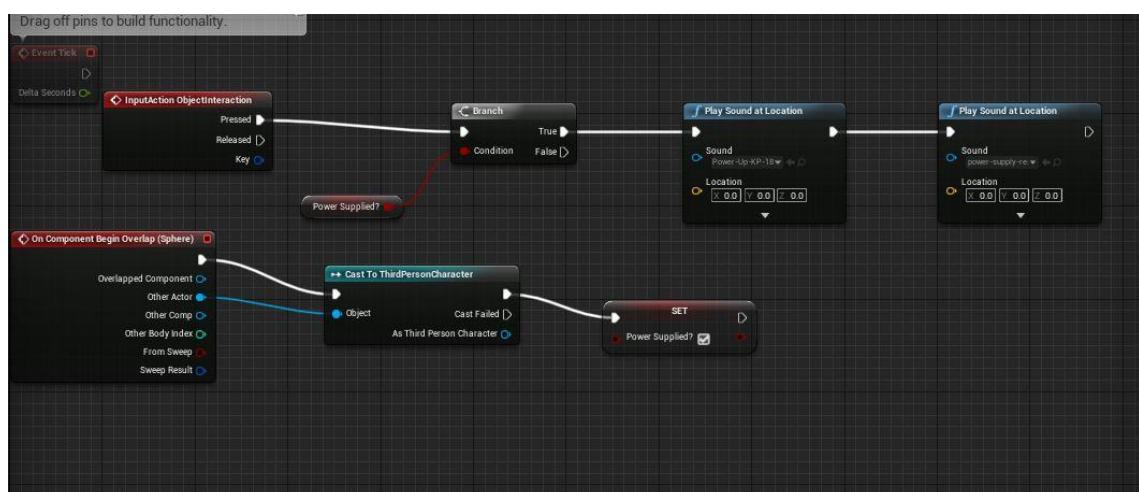


Imagen 4.d.3: Blueprint Actor del objeto de la terminal de la sala de control

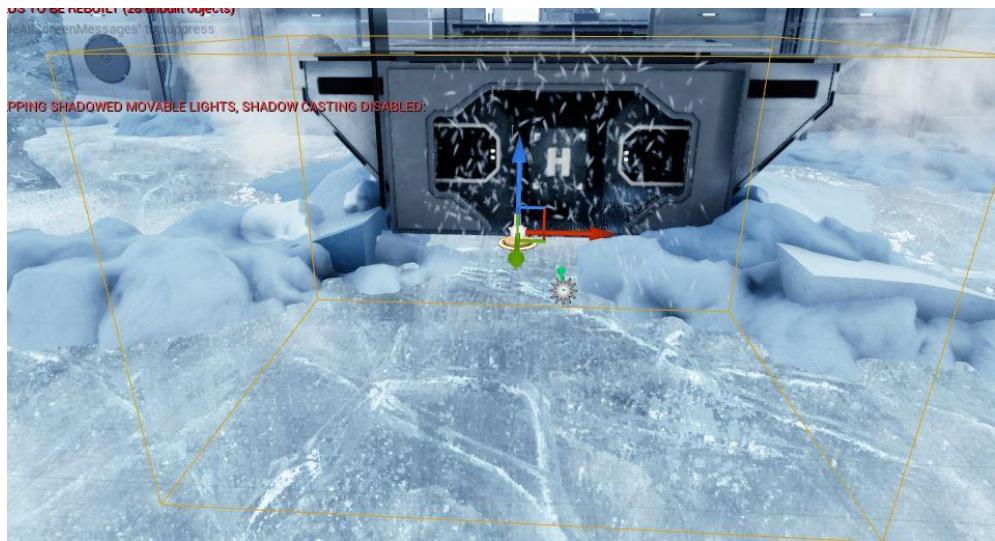


Imagen 4.d.4: Trigger Box (señala el fin del juego)

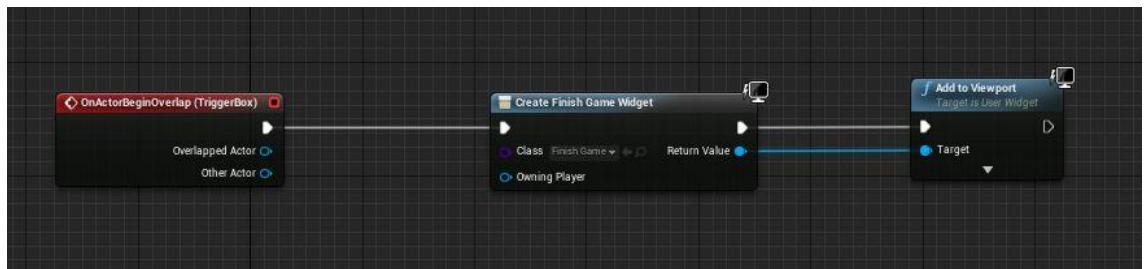


Imagen 4.d.6: Crear la pantalla de fin de juego cuando se choca con el Trigger Box



Imagen 4.d.8: Widget para la pantalla de fin de juego

e. Objetos recogibles

i. Vida y munición

En el juego habrá dos objetos recogibles o colecciónables que se podrán obtener, la salud y la munición. Estos dos objetos, tendrán los mismos componentes básicos, un *Sphere Collision* para detectar las colisiones con el jugador, un *Mesh* para indicar la forma e imagen del objeto y, en el caso de la munición, un *Widget* para indicar con un mensaje la recogida de munición.

Ambos objetos se encuentran en *Content > ThirdPersonBP > Mycharacter > Objects*.

Para implementar por un lado la recogida de salud, al chocar el jugador con el objeto, se comprobará si tiene la salud llena o no. En el primer caso, no se recogerá el objeto y simplemente el jugador chocará con él y no ocurrirá nada. En caso contrario, se llenará la salud máxima del jugador, destruyendo el objeto y reproduciendo un sonido.

Para implementar la recogida de munición ocurriría algo similar, mediante el evento *On Overlap*, permitiendo ser recogido si la munición del arma no está completa y emitiendo un sonido al destruir el actor.



Imagen 4.e.i.1: Static Mesh del objeto de vida



Imagen 4.e.ii.2: Static Mesh del objeto de munición

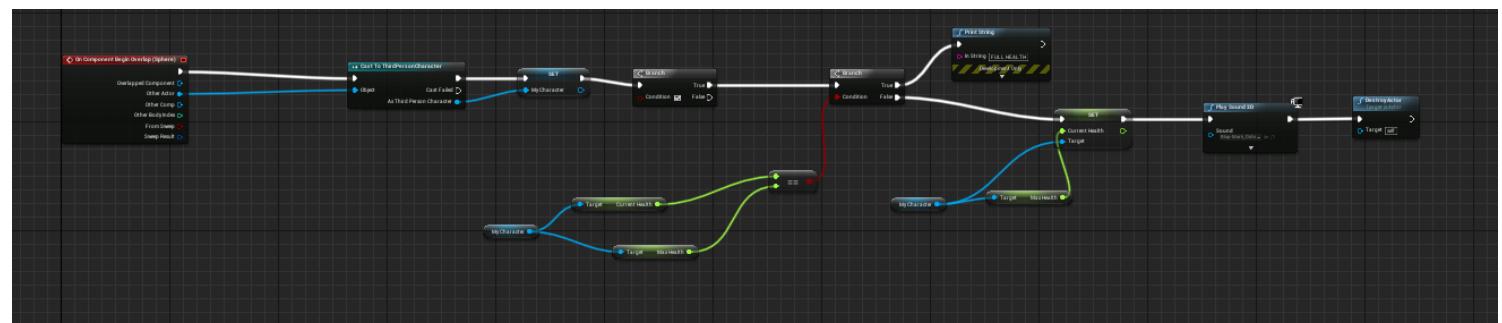


Imagen 4.e.i.3: Blueprint para controlar la salud del jugador en HealthPickup_BP

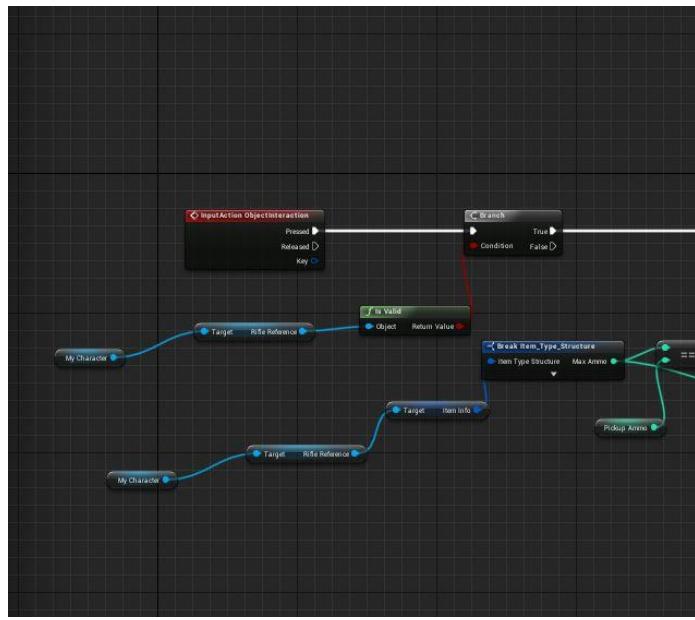


Imagen 4.e.i.4: Implementación del elemento Overlap con la acción que se realizará cuando el jugador choque con el objeto en Rifle_Ammo_Pickup_BP_Child

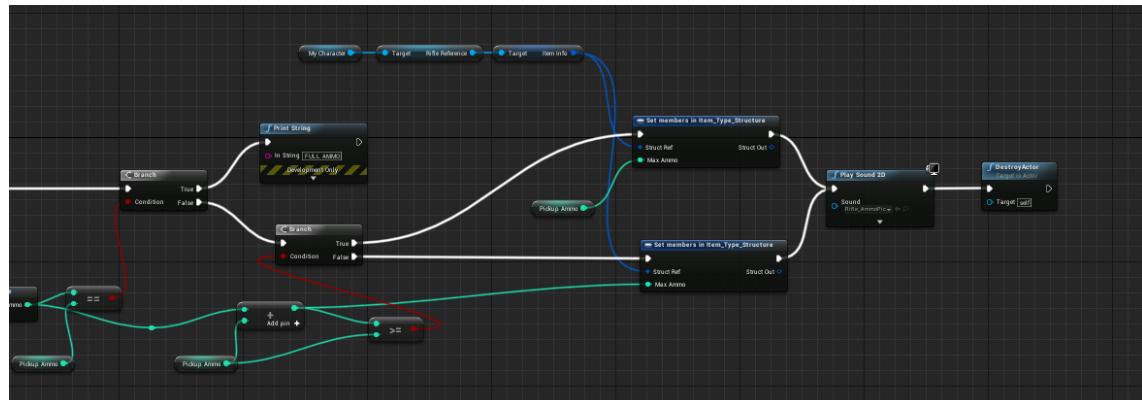


Imagen 4.e.i.5: Implementación del elemento Overlap con la acción que se realizará cuando el jugador choque con el objeto en Rifle_Ammo_Pickup_BP_Child (II)

5. Bibliografía

- Guia Unreal Third Person Shooter. Diseño Shooter (**Varios Videos**), *Youtube*:
https://www.youtube.com/watch?v=QrqKTN-ORzI&list=PLM6ZWbxOglqsCb3dUJRdYoUbyi_zFCOjs
- Música de Fondo Juego, Tron Legacy BSO:
<https://www.youtube.com/watch?v=qRUfcp8Ba6g&list=WL&index=335&t=12s>
- Animaciones de fondo en Widgets:
<https://www.youtube.com/watch?v=1ZrQppQGlok&list=WL&index=333&t=0s>
- Textos de Inicio:
<https://www.youtube.com/watch?v=uxbcVucO59Q&list=WL&index=332&t=0s>
- Conseguir efecto de fogonazo en el arma:
<https://www.youtube.com/watch?v=7OeylCXSdVE&list=WL&index=287&t=0s>
- Método de recarga de armas automática y avanzada:
https://www.youtube.com/watch?v=Pw8FG8-_1gA&list=WL&index=283&t=1260s
- Animaciones con armas:
<https://www.youtube.com/watch?v=S97c34UkLB&list=WL&index=282&t=311s>
- Como crear enemigos IA inteligentes:
<https://www.youtube.com/watch?v=fUtqgy511Bw>
- Crear Menú principal: https://youtu.be/kM_dmNtXj4o