

# Postwar: Hopeless Humanity. Desarrollo de un videojuego en Unreal Engine 4.



Grado en Ingeniería Multimedia

## Trabajo Fin de Grado

Autor:

Rubén Pérez Cristo

Tutor/es:

Francisco José Mora Lizan

Juan Antonio Puchol García



## Justificación y Objetivos

Después de hacer distintos tipos de videojuegos en la carrera y ya teniendo una base de como diseñar un videojuego creo que ha llegado el momento de poder realizar uno con un motor gráfico mucho más potente como es **Unreal Engine 4** y poder así exprimir al máximo estas tecnologías que están en auge actualmente.

Con Unreal Engine 4 se pueden obtener unos resultados de calidad y muy vistosos, aprovechando al máximo los gráficos que se pueden conseguir con este motor de juego.

Una parte importante de este proyecto es que estos motores gráficos se usan cada vez más, por lo que conseguir dominar una tecnología como esta sería un factor muy importante para un Ingeniero Multimedia.

El objetivo general de este proyecto es conseguir realizar un juego de calidad utilizando los motores gráficos comerciales como en este caso Unreal Engine 4 y aprender a sacarle el máximo partido a esta tecnología, consiguiendo un producto final acabado y de calidad.



## **Dedicatoria**



## Resumen

Durante los últimos años ha aumentado mucho los consumidores de videojuegos, lo que antes era cosa de unos pocos “gammers” ahora es habitual en casi todo el mundo. La modernización de los dispositivos móviles y el gran número de juegos para estos ha conseguido que la gente se conciencie mejor acerca de los videojuegos. Esto ha llegado a tal punto que la industria de videojuegos factura más que el cine y el sonido juntos.

El aumento de la demanda de videojuegos trae consigo un aumento de oferta de estos y por lo cual un aumento de calidad de los videojuegos buscando competir contra el resto.

Los primeros videojuegos se reducían a unas pocas interacciones y eran muy sencillos gráficamente. Actualmente, en cambio, los videojuegos son considerados verdaderas obras ya que son capaces de contar una historia como lo haría una película, con la diferencia de que tú te sientes el protagonista de esta.

De esta manera este proyecto realizado en Unreal Engine 4 pretende situar al jugador como el protagonista de una pequeña historia, aunque se centra más en la jugabilidad y la calidad visual.



# Índice de contenido

<b>Justificación y Objetivos.....</b>	<b>3</b>
<b>Dedicatoria.....</b>	<b>5</b>
<b>Resumen.....</b>	<b>7</b>
<b>Índice de contenido .....</b>	<b>9</b>
<b>Índice de Figuras .....</b>	<b>15</b>
<b>1. Introducción.....</b>	<b>19</b>
<b>2. Marco Teórico o Estado del Arte.....</b>	<b>21</b>
<b>2.1 ¿QUÉ ES UN VIDEOJUEGO?.....</b>	<b>21</b>
<b>2.2 REFERENCIAS .....</b>	<b>23</b>
<b>2.2.1 Modo Zombis de Call of Duty.....</b>	<b>23</b>
2.2.1.1 Nazi Zombies - Call of Duty: World at War .....	24
2.2.1.2 Zombies - Call of Duty: Black Ops .....	25
2.2.1.3 Zombies - Call of Duty: Black Ops II .....	25
2.2.1.4 Zombies - Call of Duty: Black Ops III .....	26
2.2.1.5 Dead Island .....	26
2.2.1.6 Left 4 Dead 2.....	27
2.2.1.7 Resident Evil VII .....	28
<b>3. Objetivos .....</b>	<b>31</b>
<b>3.1 OBJETIVO GENERAL DEL TRABAJO DE FIN DE GRADO.....</b>	<b>31</b>
<b>3.2 DESGLOSE DE OBJETIVOS .....</b>	<b>31</b>
<b>4. Metodología .....</b>	<b>33</b>
<b>4.1 METODOLOGÍA DE DESARROLLO.....</b>	<b>33</b>
4.1.1 Metodologías Agiles .....	33
4.1.2 Kanban .....	33
<b>4.2 GESTIÓN DE PROYECTO .....</b>	<b>34</b>
4.2.1 Toggl y la compatibilidad con Trello.....	35
<b>4.3 CONTROL DE VERSIONES .....</b>	<b>38</b>
<b>5. Cuerpo del trabajo .....</b>	<b>40</b>
<b>5.1 ANÁLISIS, EXPLICACIÓN Y ELECCIÓN DE HERRAMIENTAS .....</b>	<b>40</b>
5.1.1 Motor de videojuego .....	40
5.1.1.1 Conceptos clave.....	40

5.1.1.2 Motores de videojuegos actuales .....	41
5.1.1.2.1 Cry Engine 3 .....	44
5.1.1.2.2 Unreal Engine 4 .....	44
5.1.1.2.3 Unity 3D .....	46
5.1.1.3 Elección del motor de juego .....	47
<b>5.1.2 Otras herramientas.....</b>	<b>48</b>
5.1.2.1 Modelado 3D .....	48
5.1.2.1.1 Blender.....	48
5.1.2.1.2 3DS Max .....	49
5.1.2.1.3 Maya .....	49
5.1.2.2 Texturizado .....	50
5.1.2.2.1 Mudbox.....	50
5.1.2.2.2 Substance Painter .....	50
5.1.2.3 Elección de herramientas .....	51
<b>5.2 DOCUMENTO DE DISEÑO DEL VIDEOJUEGO (GDD) .....</b>	<b>52</b>
<b>5.2.1 El juego en términos generales .....</b>	<b>52</b>
5.2.1.1 Resumen de argumento .....	52
5.2.1.2 Conjunto de características .....	53
5.2.1.3 Género.....	53
5.2.1.4 Clasificación PEGI.....	53
5.2.1.4.1 ¿Qué es PEGI? .....	53
5.2.1.4.2 ¿Qué entendemos por clasificación? .....	54
5.2.1.4.3 ¿Cómo se mide la clasificación?.....	54
5.2.1.4.4 Clasificación PEGI de NOMBRE DE JUEGO AQUI .....	55
5.2.1.5 Resumen del flujo de juego .....	55
5.2.1.6 Apariencia del juego .....	56
5.2.1.7 Ámbito .....	56
5.2.1.7.1 Lugares en los que se desarrolla el videojuego.....	56
5.2.1.7.2 Número de niveles.....	56
5.2.1.7.3 Número de NPC's.....	56
5.2.1.7.4 Número de habilidades.....	57
<b>5.2.2 Jugabilidad y mecánicas .....</b>	<b>57</b>
5.2.2.1 Jugabilidad .....	57
5.2.2.1.1 ¿Qué es la jugabilidad? .....	57
5.2.2.1.2 Objetivos del juego .....	58
5.2.2.1.3 Progresión.....	58
5.2.2.1.4 Acciones del personaje .....	59
5.2.2.1.5 Controles del juego .....	59
5.2.2.2 Mecánicas .....	59
5.2.2.2.1 Movimiento y cámara en primera persona .....	59
5.2.2.2.2 Disparar y recargar.....	60
5.2.2.2.3 Comprar armas .....	60

5.2.2.2.4 Mejorar armas .....	61
5.2.2.2.5 Comprar munición .....	61
5.2.2.2.6 Power-ups.....	62
5.2.2.2.6.1 Resistencia .....	62
5.2.2.2.6.2 Recarga rápida .....	62
5.2.2.2.6.2 Revivir .....	62
5.2.2.2.7 Abrir puertas.....	62
5.2.2.3 Rejugar y salvar.....	63
5.2.2.4 Ranking .....	63
<b>5.2.3 Historia y personajes .....</b>	<b>63</b>
5.2.3.1 Historia .....	64
5.2.3.2 Personajes .....	65
5.2.3.2.1 Protagonista (TODO ponerle nombre).....	65
5.2.3.2.2 Zombis .....	65
<b>5.2.4 Nivel de juego .....</b>	<b>65</b>
5.2.4.1 Inteligencia Artificial .....	65
5.2.4.2 Escenario .....	66
5.2.4.3 Música y sonidos.....	66
5.2.4.4 Puntos de respawn .....	67
<b>5.2.5 Interfaz .....</b>	<b>68</b>
5.2.5.1 Menús.....	68
5.2.5.1.1 Menú principal .....	69
5.2.5.1.2 Menú de pausa .....	70
5.2.5.1.3 Menú fin de partida .....	70
5.2.5.1.3.1 Sin Ranking .....	70
5.2.5.1.3.2 Con Ranking .....	71
5.2.5.1.4 Pantalla de ranking .....	72
5.2.5.1.5 Menú de opciones .....	72
5.2.5.2 HUD .....	73
5.2.5.1 Cámara.....	75
<b>5.3 DESARROLLO E IMPLEMENTACIÓN.....</b>	<b>77</b>
<b>5.3.1 Entendiendo Unreal Engine 4 y su sistema de scripting visual Blueprints.....</b>	<b>77</b>
5.3.1.1 La arquitectura de Unreal Engine 4 .....	77
5.3.1.1.1 Conceptos base de la arquitectura de Unreal Engine 4 .....	77
5.3.1.1.1.1 UObjects y Actors.....	77
5.3.1.1.1.2 Gameplay y las clases básicas .....	77
5.3.1.1.1.3 Representación y control de jugadores .....	78
5.3.1.1.1.3.1 Representando jugadores en el mundo .....	78
5.3.1.1.1.3.1.1 Pawn.....	78
5.3.1.1.1.3.1.2 Character.....	78
5.3.1.1.1.3.2 Formas de controlar a un jugador del mundo.....	78
5.3.1.1.1.3.2.1 Controller .....	78

5.3.1.1.3.2.2 PlayerController .....	79
5.3.1.1.3.2.3 AIController .....	79
5.3.1.1.4 Mostrando información a los jugadores .....	79
5.3.1.1.4.1 HUD .....	79
5.3.1.1.4.2 Cámara .....	79
5.3.1.1.5 Manejando información del juego .....	79
5.3.1.1.5.1 GameMode .....	80
5.3.1.1.5.2 GameState .....	80
5.3.1.1.5.3 PlayerState .....	80
5.3.1.1.6 Como se relacionan estas clases .....	81
5.3.1.1.2 Scripting visual Blueprints .....	81
5.3.1.1.2.1 ¿Qué es un Blueprint? .....	82
5.3.1.1.2.2 ¿Qué contiene un Blueprint? .....	82
5.3.1.1.2.2.1 Componentes (Components Window) .....	82
5.3.1.1.2.2.2 Constructor (Construction Script) .....	83
5.3.1.1.2.2.3 Event Graph .....	85
5.3.1.1.2.2.4 Funciones .....	86
5.3.1.1.2.2.5 Variables .....	86
5.3.1.1.2.2.6 Macros .....	86
5.3.1.1.2.3 ¿Qué tipos de Blueprints hay? .....	87
5.3.1.1.2.3.1 Blueprint Class .....	87
5.3.1.1.2.3.2 Level Blueprint .....	88
5.3.1.1.2.3.3 Data-Only Blueprint .....	88
5.3.1.1.2.3.4 Blueprint Interface .....	88
5.3.1.1.2.3.5 Blueprint Macro Library .....	89
5.3.1.1.3 Renderizado y gráficos .....	89
5.3.1.1.3.1 Luces .....	90
5.3.1.1.3.1.1 Static lights .....	93
5.3.1.1.3.1.2 Stationary lights .....	94
5.3.1.1.3.1.3 Movable lights .....	94
5.3.1.1.3.2 Materiales .....	95
5.3.1.1.3.3 Postprocesado (PostProcess Volume) .....	97
5.3.1.1.4 Audio y sonido .....	98
5.3.1.1.4.1 Sound Cue Editor .....	98
5.3.1.1.4.2 Ambient Sound Actor .....	99
5.3.1.1.4.3 Sound Attenuation .....	100
5.3.1.1.5 Animaciones .....	100
5.3.1.1.5.1 Blending animation (Blend Spaces) .....	101
5.3.1.1.5.2 Animation blueprint .....	102
5.3.1.1.5.3 Animation Retargeting (Different Skeletons) .....	103
5.3.1.1.5.4 Notificación en animación .....	103
<b>5.3.2 Desarrollo de mecánicas de juego .....</b>	<b>103</b>

5.3.2.1 Blueprint First Person Character (personaje principal).....	104
5.3.2.1.1 Componentes.....	104
5.3.2.1.2 Variables .....	105
5.3.2.1.2 Funciones.....	109
5.3.2.1.3 Funciones importantes en detalle .....	113
5.3.2.1.3.1 Apuntar .....	113
5.3.2.1.3.2 Equipar arma.....	115
5.3.2.1.3.3 Cambiar de arma.....	117
5.3.2.1.3.4 Disparar.....	118
5.3.2.1.3.5 Recibir daño .....	119
5.3.2.1.3.5.1 Restar vida.....	120
5.3.2.1.3.5.2 Regenerar vida .....	122
5.3.2.1.4 Animaciones .....	122
5.3.2.2 Armas.....	126
5.3.2.2.1 WeaponDrop Blueprint (armas en la pared).....	127
5.3.2.2.1.1 Componentes.....	127
5.3.2.2.1.2 Variables .....	127
5.3.2.2.1.3 Funciones .....	128
5.3.2.2.1.3.1 ConstructionScript (Constructor de clase).....	128
5.3.2.2.1.3.2 Inicializar .....	129
5.3.2.2.1.3.3 Entrar en zona de compra .....	130
5.3.2.2.1.3.4 Salir de zona de compra .....	132
5.3.2.2.1.3.5 Interactuar .....	132
5.3.2.2.1.3.6 Comprar arma .....	133
5.3.2.2.1.3.7 Comprar munición .....	134
5.3.2.2.1.3.8 Comprobar si ya hemos comprado el arma .....	135
5.3.2.2.1.3.9 Información compra arma y munición .....	136
5.3.2.2.2 Weapon Blueprint (arma equipada) .....	136
5.3.2.2.2.1 Interfaz (Weapon Interface).....	136
5.3.2.2.2.2 Variables .....	137
5.3.2.2.2.3 Funciones .....	139
5.3.2.2.2.3.1 Constructor (Construction Script) .....	139
5.3.2.2.2.3.2 Inicializar .....	139
5.3.2.2.2.3.3 Disparar .....	140
5.3.2.2.2.3.3.1 InputAction Fire.....	140
5.3.2.2.2.3.3.2 Event Shoot.....	141
5.3.2.2.2.3.3.3 Create Projectil.....	142
5.3.2.2.2.3.3.3.1 Función BalaTransform .....	144
5.3.2.2.2.3.3.3.2 Función GetRaycastPos.....	145
5.3.2.2.2.3.3.3.3 Funcion DondelImpacto.....	146
5.3.2.2.2.3.3.4 Bullet Projectil Actor .....	147
5.3.2.2.3.4 Dejar de disparar .....	148
5.3.2.2.3.5 Recargar .....	148

5.3.2.2.2.3.6 Mejorar Arma.....	148
5.3.2.2.2.3.6.1 Mejorar Daño .....	148
5.3.2.2.2.3.6.2 Mejorar Munición Máxima.....	148
5.3.2.2.2.3.6.3 Mejorar Capacidad Cargador.....	148
5.3.2.2.2.3.6.4 Mejorar Cadencia .....	148
5.3.2.2.2.3.6.5 Mejorar Tiempo Recarga.....	148
5.3.2.3 Interactuable Objects (Power-ups).....	148
5.3.2.4 Enemigo .....	148
5.3.2.5 Blueprint Partida.....	148
<b>5.3.3 Desarrollo Menús y HUD.....</b>	<b>149</b>
<b>5.3.4 Arte 3D en el videojuego.....</b>	<b>149</b>
<b>5.3.5 Diseño de nivel de juego (escenario, iluminación, sonidos) .....</b>	<b>149</b>
<b>6. Bibliografía .....</b>	<b>150</b>

# Índice de Figuras

Figura 1: Battlefield como ejemplo de FPS .....	23
Figura 2: Nazi Zombis – Call of Duty: World at War.....	24
Figura 3: Zombis – Call of Duty Black Ops .....	25
Figura 4: Zombis - Call of Duty Black Ops 2.....	26
Figura 5: Zombis - Call of Duty: Black Ops 3.....	26
Figura 6: Dead Island .....	27
Figura 7: Left 4 Dead 2 .....	28
Figura 8: Resident Evil 7.....	29
Figura 9: Resident Evil 7 - Ambientación.....	29
Figura 10: Resident Evil 7 - Ambientación 2.....	30
Figura 11: Trello del proyecto.....	35
Figura 12: Email semanal de Toggl .....	36
Figura 13: Toggl del proyecto.....	37
Figura 14: Ramas de GitHub del proyecto.....	38
Figura 15: Commits a rama específica (Apuntar) .....	39
Figura 16: Llevando versiones funcionales de otras ramas a Master .....	39
Figura 17: Motores de juegos actuales .....	41
Figura 18: Tabla motores de juego actuales (1/2) .....	42
Figura 19: Tabla motores de juego actuales (2/2) .....	43
Figura 20: Logo Cry Engine.....	44
Figura 21: Logo Unreal Engine 4.....	45
Figura 22: Blueprint vs Kismet .....	45
Figura 23: Logo Unity.....	46
Figura 24: Logo Blender .....	48
Figura 25: Logo 3DS Max .....	49
Figura 26: Logo Maya.....	49
Figura 27: Logo Mudbox .....	50
Figura 28: Logo Substance Painter .....	50
Figura 29: Descriptores clasificación PEGI .....	54
Figura 30: Clasificación PEGI de PostWar:Hopeless Humanity .....	55
Figura 31: Flujo de pantallas del juego .....	55
Figura 32: Menú del juego en Photoshop.....	69
Figura 33: Menú de pausa en juego .....	70
Figura 34: Pantalla muerte sin ranking .....	71
Figura 35: Pantalla muerte con ranking .....	71

Figura 36: Menú ranking en Photoshop .....	72
Figura 37: Menú opciones en Photoshop .....	73
Figura 38: HUD información constante .....	74
Figura 39: HUD información dinámica .....	75
Figura 40: Camara FPS .....	75
Figura 41: Relacion entre clases.....	81
Figura 42: Contruction Script del arma AK-47.....	84
Figura 43: Función inicializar al que llaman los constructores de las armas .....	85
Figura 44: Ejemplo de Macro.....	87
Figura 45: Ejemplo llamada a macro desde evento personalizado.....	87
Figura 46: Ejemplo Spot Light.....	90
Figura 47: Ejemplo Point Light.....	90
Figura 48: Ejemplo Directional Light .....	91
Figura 49: Ejemplo Sky Light.....	91
Figura 50: Ejemplo de una sala del proyecto con iluminación Spot Light y Point Light. ....	92
Figura 51: Ejemplo 1 iluminación únicamente con Spot Light sacada del proyecto .....	92
Figura 52: Ejemplo iluminación únicamente con Spot Light sacada del proyecto .....	93
Figura 53: Material simple y su aplicación en el proyecto.....	96
Figura 54: Material complejo (simulación de agua) .....	96
Figura 55: Aplicación de material complejo en el proyecto.....	97
Figura 56: Ejemplo Sound Cue - Imagen sacada de la documentación oficial de Unreal Engine 4	99
Figura 57: Ejemplo blending animation del proyecto .....	102
Figura 58: Componentes Bluerpint personaje .....	105
Figura 59: Tabla de variables del personaje .....	109
Figura 60: Tabla de funciones del personaje.....	113
Figura 61: Función apuntar .....	113
Figura 62: Grafica TimeLine (interpolación entre transformaciones) .....	115
Figura 63: Función equipar arma .....	115
Figura 64: Ejemplo de socket en el editor de Unreal Engine 4.....	116
Figura 65: InputAction cambiar de arma .....	117
Figura 66: InputAction disparar .....	118
Figura 67: InputAction recibir daño.....	119
Figura 68: Función restar vida .....	121
Figura 69: Función suma vida.....	122
Figura 70: Animation Starter Pack de Unreal .....	123
Figura 71: Retargeting Animation del proyecto.....	124
Figura 72: Animaciones en primera y tercera persona del proyecto .....	124

Figura 73: Corrigiendo animación desde el editor de Unreal.....	125
Figura 74: Animation Blueprint personaje.....	126
Figura 75: Componentes WeaponParentDrop Blueprint.....	127
Figura 76: Table variables WeaponParentDrop .....	128
Figura 77: Constructor AKDrop .....	<b>¡Error! Marcador no definido.</b>
Figura 78: Constructor UMPDrop.....	<b>¡Error! Marcador no definido.</b>
Figura 79: Funcion inicializar WeaponParentDrop.....	130
Figura 80: WeaponParentDrop evento entrar en zona de compra.....	130
Figura 81: Mostrando información compra munición arma .....	131
Figura 82: Mostrando información comprar arma .....	132
Figura 83: WeaponParentDrop evento salir de zona de compra .....	132
Figura 84: InputAction Interactuar WeaponDrop .....	133
Figura 85: Función comprar arma .....	134
Figura 86: Función comprar munición.....	135
Figura 87: Función que comprueba si ya hemos comprado un arma .....	135
Figura 88: Función información compra (izquierda) y función información compra munición (derecha).....	136
Figura 89: Tabla variables WeaponParent Bluerpint .....	138
Figura 90: Contructor AK-47 .....	<b>¡Error! Marcador no definido.</b>
Figura 91: Constructor M60 .....	<b>¡Error! Marcador no definido.</b>
Figura 92: Función inicializar arma .....	140
Figura 93: Input Action Fire (evento disparar) .....	141
Figura 94: Event Shoot de la clase AK .....	142
Figura 95: Function CreateProjectil (parte 1).....	142



## 1. Introducción

Aunque existe un debate acerca de quién creó el primer videojuego, es el “**Tennis for Two**” el considerado primer videojuego y se creó en el año 1958. El juego constaba de una línea horizontal que era el campo de juego y otra pequeña línea vertical que era la red. Los jugadores tenían que dirigir la bola y golpearla. Para ello se usaba un oscilador a modo de mando y un monitor conectado a una computadora analógica.

Desde este inicio tan solo han pasado un poco más de 50 años y el cambio en la industria del videojuego ha sido brutal, tanto que ha logrado alcanzar en apenas medio siglo de historia el estatus de medio artístico.

Hasta hace unos pocos años los videojuegos era el entretenimiento de solo por unos pocos, pero a medida que iba avanzando la tecnología y aumentando la calidad de estos se han ido haciendo más populares a medida sobre todo entre los jóvenes. La evolución de los videojuegos ha llegado a tal punto que la innovación tecnológica ha sido impulsada también por estos.

La aparición de los *smartphones* ha supuesto un antes y un después en la industria de los videojuegos, ya que gracias a estos, los videojuegos están al alcance de todos. De esta manera juegos como **Candy Crush Saga** o **Pokemon Go** han conseguido atraer al mundo de los videojuegos a adultos que antes no estaban interesados en los videojuegos.

La evolución de los videojuegos también trajo una evolución en sus géneros y sus objetivos, siendo estos muy variados como desde contar una historia hasta enseñar a un usuario. Actualmente hay una gran lista de géneros.

El género principal de NOMBRE DEL JUEGO AQUÍ es el tan conocido **FPS** (First Person Shooter) o en español **juego de disparos en primera persona**. En este género de videojuegos el jugador observa el mundo desde la perspectiva del personaje protagonista.

La estética de este proyecto se basa en el género de “**Survival Horror**”, que se caracteriza por situar al jugador en algún escenario lúgubre, claustrofóbico, perturbador y, por lo general, escasamente iluminado, del que poco o nada conoce y por el que debe transitar con algún pretexto que puede ser tan simple como la supervivencia mientras es acechado por criaturas o entes hostiles de aspecto terrorífico por ejemplo zombis.

Este documento consta de las siguientes partes:

- **Marco Teórico:** donde se tratan conceptos claves para el entendimiento del proyecto, así como ejemplos y referencias de videojuegos en los que se basa NOMBRE DE JUEGO AQUÍ.
- **Objetivos:** aquí se describen el objetivo general y uno a uno todos los objetivos específicos del proyecto.
- **Metodología:** un pequeño análisis de las metodologías más utilizadas y la metodología elegida, además de las herramientas utilizadas para la gestión del proyecto.
- **Cuerpo de trabajo:** en esta parte encontraremos en primer lugar un análisis de los distintos motores de videojuegos, el motor elegido y el porqué de esta elección. A continuación estará el Documento de Diseño del Videojuego (GDD, *Game Design Document*), donde se encontrara toda la información acerca del videojuego como personajes, historia, mecánicas, diseño de niveles, etc. Por ultimo en esta parte estará todo el desarrollo e implementación del videojuego.
- **Conclusiones:** aparecen mis conclusiones sobre todo el trabajo realizado en este proyecto.
- **Anexo:** aparecerán imágenes acerca del resultado final del videojuego.

## 2. Marco Teórico o Estado del Arte

Este bloque sirve un poco para ponernos en situación sobre lo que se va a desarrollar en este proyecto. Para ello se definirá que es un videojuego y que es el género de **FPS** ya que no tiene mucho sentido explicar el desarrollo de un videojuego de disparos en primera persona si no se tiene bien claro que es un videojuego y que es el género FPS.

Además también vamos a analizar algunos videojuegos de este estilo que han servido como referencia a la hora de desarrollar NOMBRE DE JUEGO AQUÍ.

### 2.1 ¿Qué es un videojuego?

Es un poco difícil dar una única solución a esta pregunta, pues con la evolución de los videojuegos su definición también ha ido cambiando. Aquí tenemos varios ejemplos del concepto de videojuego:

**“Un videojuego es una aplicación interactiva** orientada al entretenimiento que, a través de ciertos mandos o controles, permite simular experiencias en la pantalla de un televisor, una computadora u otro dispositivo electrónico.

Los videojuegos se diferencian de otras formas de entretenimiento, en que deben ser interactivos; es decir, los usuarios deben involucrarse activamente con el contenido.”

“Un videojuego o juego de video es un juego electrónico en el que una o más personas interactúan, por medio de un controlador, con un dispositivo que muestra imágenes de vídeo.<sup>1</sup> Este dispositivo electrónico, conocido genéricamente como «plataforma», puede ser una computadora, una máquina arcade, una videoconsola o un dispositivo portátil como por ejemplo un teléfono móvil.

**Los videojuegos son año por año, una de las principales industrias del arte y entretenimiento”**

“Se define como Videojuego (también conocidos como Juego de Video) a todo aplicación o Software que ha sido creado con el fin del entretenimiento”

Si bien las definiciones son bastante parecidas unas a otras hay cosas en las que no están totalmente de acuerdo. Por ejemplo:

En la última definición dice que un videojuego es creado con el fin del entretenimiento, esta definición puede que fuese correcta antes pero ahora los videojuegos y géneros de estos han variado mucho, y por lo tanto hay videojuegos hechos únicamente para el aprendizaje, es decir su fin no es el entretenimiento sino el aprendizaje.

En la primera definición en cambio dice que un videojuego permite simular experiencias, por lo tanto si buscamos simular experiencias como puede ser una simulación de vuelo, cuya definición en Wikipedia es: “Un simulador de vuelo es un sistema que intenta replicar, o simular, la experiencia de pilotar una aeronave de la forma más precisa y realista posible. Los diferentes tipos de simuladores de vuelo **van desde videojuegos hasta réplicas de cabinas en tamaño real**” volvemos al caso anterior donde el fin de este videojuego (simulación de vuelo) no es el de entretener sino el de enseñar.

Por ultimo si nos centramos en la segunda definición, fijándonos en la frase que está en negrita podemos observar que los videojuegos no son solo entretenimiento sino que están considerados incluso como obras de arte.

Esto se debe a la evolución que han sufrido los videojuegos ya que para algunos siguen siendo puro entretenimiento pero la verdad es que son capaces de narrar historias de la misma manera que lo haría una película o un libro, añadiendo además que tú eres el protagonista de la historia y el que interactúa y avanza por está logrando una mayor inmersión, por esto los videojuegos no son solo reconocidos como entretenimiento sino también como arte.

Una vez hemos definido que es un videojuego vamos a definir que es el género de FPS.

En primer lugar el género juego de disparos engloban una gran cantidad de subgéneros que tienen la característica común de permitir controlar un personaje que, por norma general, dispone de un arma que puede ser disparada a voluntad.

Por otra parte la cámara en primera persona es una vista que se emplea en los videojuegos en la cual el mundo se ve desde la perspectiva del personaje protagonista.

Juntando estas 2 cosas obtenemos el género de disparos en primera persona más conocido como FPS por su nombre en inglés (First Person Shooter). Un ejemplo de este tipo de juego puede ser el Battlefield.

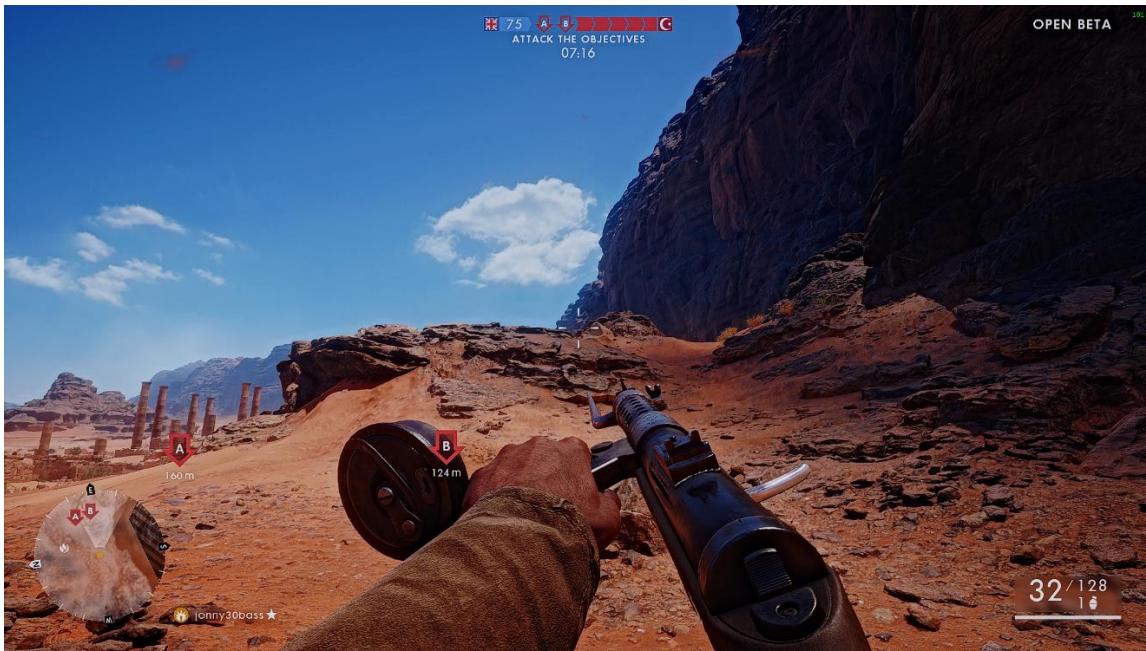


Figura 1: Battlefield como ejemplo de FPS

## 2.2 Referencias

Aunque la estética y esencia del juego es propia para conseguir esta me basaré en algunos videojuegos populares que se parecen al que quiero realizar. Para tener claro que es lo que quiero conseguir he estudiado tanto mecánica como estética de los siguientes videojuegos:

- Modo zombis de Call of Duty
  - Nazi Zombies - Call of Duty: World at War
  - Zombies – Call of Duty: Black Ops
  - Zombies – Call of Duty: Black Ops 2
  - Zombies – Call of Duty: Black Ops 3
- Dead Island
- Left 4 dead 2
- Resident Evil VII

### 2.2.1 Modo Zombis de Call of Duty

Este modo de juego es la principal referencia de NOMBRE DEL JUEGO AQUÍ, ya que es un FPS en el que tienes que sobrevivir el máximo número de rondas ante continuas oleadas zombis que intentaran matarte. El número de munición es finito y tendrás que encargarte de ir reponiendo munición o ir cambiando de arma.

Es un modo de juego exclusivo de Treyarch que tuvo su primera aparición en el videojuego Call of Duty: World at War. Gracias a su popularidad, este minijuego regresó en Call of Duty: Black Ops, Call of Duty: Black Ops II y En Call Of Duty:Black Ops III. Consiste en masacrar hordas infinitas de zombis que aparecen por rondas. Dependiendo de la ubicación de cada mapa, se pueden luchar contra zombis alemanes, estadounidenses, rusos y asiáticos, ya obstante se encuentran en diferentes mapas y modos.

Hasta cuatro jugadores deben sobrevivir infinitas hordas de ataques zombis, ganando puntos por matar o dañar zombis y por la reparación de las barreras a través de la cual estos llegan al escenario. Estos puntos pueden ser utilizados para comprar armas y *Perk-a-Colas* (Bebidas que dan habilidades especiales) en el proceso, o desbloquear nuevas zonas y activar otros objetos especiales.

No hay un límite de número de rondas, la partida se acaba cuando todos los jugadores son abatidos o matados por los zombis. Éstos se vuelven más fuertes y rápidos a medida que pasan las rondas, obligando a que los jugadores tomen decisiones tácticas y administren los puntos que ganan. En algunas ocasiones, cuando matas a los zombis estos dejan una especie de poderes más conocido como "*Power-Ups*" los cuales son muy efectivos en distintas circunstancias y pueden salvarte de un apuro.

### 2.2.1.1 Nazi Zombies - Call of Duty: World at War



Figura 2: Nazi Zombies – Call of Duty: World at War

### **2.2.1.2 Zombies - Call of Duty: Black Ops**



*Figura 3: Zombis – Call of Duty Black Ops*

### **2.2.1.3 Zombies - Call of Duty: Black Ops II**

En este nuevo juego se incluyen varias novedades y alguna mejora como por ejemplo un motor gráfico mejorado que será capaz de tener el doble de zombis en una partida. Entre las novedades destacan 2 nuevos modos de juego que son:

**TranZit:** que se trata de una historia cooperativa ambientada en un mundo en el cual los jugadores tendrán que viajar a diferentes zonas montados en un autobús fortificado.

**Pena:** este nuevo modo alberga hasta ocho jugadores que formaran 2 equipos de 4 componentes cada uno. Los dos equipos tienen que sobrevivir por separado y sacarles ventaja a sus adversarios, es decir ya no solo luchas por sobrevivir contra los zombis sino que ahora también luchas por ganar al equipo rival.



Figura 4: Zombis - Call of Duty Black Ops 2

#### 2.2.1.4 Zombies - Call of Duty: Black Ops III



Figura 5: Zombis - Call of Duty: Black Ops 3

#### 2.2.1.5 Dead Island

Dead Island es un videojuego de rol en acción y un survival horror en un Mundo abierto desarrollado por Techland. Se centra en el reto de la supervivencia en una isla infectada por zombies.



Figura 6: Dead Island

El juego comienza con una noche de fiesta donde empiezan a suceder cosas extrañas a las cuales no le damos importancia debido al alcohol. A la mañana siguiente nos despertamos en una habitación de hotel en mitad del caos (ascensores que no van, cortes eléctricos, gente saltando al vacío...). En el intento por huir del hotel recibiremos la ayuda de unos supervivientes cuando caemos víctimas de un ataque zombi. Aquí empieza la historia real, cuando te cuentan que por alguna extraña razón somos inmunes al virus que ha convertido a las personas en zombis, el objetivo del juego pasa a ser la supervivencia y huir de la isla.

### 2.2.1.6 Left 4 Dead 2

Left 4 Dead 2 es un videojuego de disparos en primera persona cooperativo de tipo survival horror creado por la compañía *Valve Software*.

Es un shooter en primera persona ambientado en medio de un apocalipsis zombi. El mundo está infectado y sólo quedan cuatro supervivientes que, intentando cooperar entre ellos, buscan escapar de los devastados escenarios por esta plaga generalizada. La clave se encuentra en la palabra cooperación, ya que la naturaleza de Left 4 Dead y esta segunda parte los lleva a ser jugados y disfrutados al máximo con tres compañeros más, entre los que se deberá acabar con miles de zombis.



Figura 7: Left 4 Dead 2

Tres semanas después del primer brote del virus que sacude U.S.A., la infección se traslada a los Estados Unidos. La CEDA ha endurecido las cuarentenas de las ciudades y pueblos infectados por el virus y ayuda a los supervivientes a escapar de ese apocalipsis zombi.

Nuestra historia comienza en la azotea de un edificio de Savannah, en la que cuatro supervivientes deben atravesar un centro comercial, un embotellamiento de vehículos, una feria cerrada, unos pantanos, una azucarera, un pueblo medio inundado y, finalmente, la ciudad destrozada de Nueva Orleans.

### 2.2.1.7 Resident Evil VII

Resident Evil VII es un videojuego de survival horror desarrollado por la empresa *Capcom*. Es el undécimo título de la serie principal de Resident Evil y, a diferencia de los otros juegos de la franquicia, este es en primera persona.

Aunque este juego se aleja un poco de los anteriores que eran juegos de supervivencia como tal, y dista un poco del objetivo principal de NOMBRE DE JUEGO AQUÍ que es un FPS en que tendrás que sobrevivir el máximo número de rondas posibles también es una referencia importante ya que la estética y psicología de este juego son en las que me quiero basar.

La ambientación de este juego con lugares poco iluminados, música de fondo y que te hacen estar en tensión en todo momento es el clima en el que quiero que el jugador de NOMBRE DE JUEGO

AQUÍ se sienta envuelto. El uso de la cámara en primera persona permite crear **una atmósfera terrorífica que atenaza al jugador**. Además, plasma la sensación de terror de una manera muy inteligente. No abusa de sustos fáciles, de giros de cámara en los que te encuentras algo que no te esperabas o de estridentes sonidos porque sí. En su lugar, crea la sensación de tensión de una forma mucho más sutil. Juega muy bien con la iluminación y con el sonido.



Figura 8: Resident Evil 7

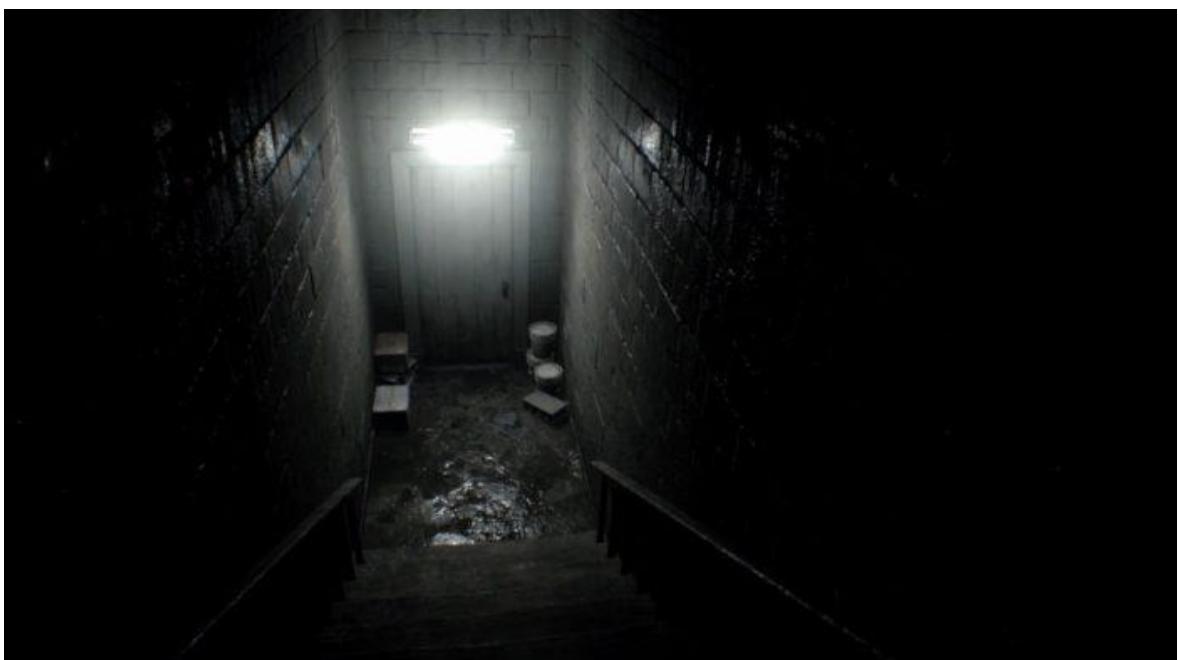


Figura 9: Resident Evil 7 - Ambientación



*Figura 10: Resident Evil 7 - Ambientación 2*

Ethan Winters es atraído a una plantación abandonada en Dulvey, Louisiana por un mensaje de su esposa Mia, que ha estado desaparecida durante 3 años. Explorando una casa aparentemente abandonada, Ethan encuentra a Mia encarcelada en el sótano. Durante su escape, Mia está poseída por una fuerza desconocida y ataca a Ethan. Ethan tendrá que seguir investigando con la ayuda de Zoe a quien conoce a través de una llamada telefónica para averiguar qué está pasando.

### **3. Objetivos**

#### **3.1 Objetivo general del Trabajo de Fin de Grado**

El objetivo de este proyecto es la realización de un videojuego 3D de genero FPS para PC. Este juego se desarrollara con el motor de videojuegos Unreal Engine 4 y tanto la programación de mecánicas, como de IA y la lógica del juego se hará exclusivamente con el sistema de **Scripting Visual Blueprints**.

Además se realizara un análisis previo de videojuegos similares para obtener información acerca del mercado en este tipo de juegos, se realizara un diseño del videojuego que quedara plasmado en el GDD, se analizarán los diferentes motores de videojuegos y que características aporta Unreal Engine para la realización de videojuegos. También se crearan recursos para el proyecto y se utilizaran otros de terceros siempre y cuando las licencias Creative Commons lo permitan.

#### **3.2 Desglose de Objetivos**

En el apartado anterior veíamos una vista general de los objetivos, en este apartado lo detallaremos en objetivos más específicos.

1. Diseñar e implementar un videojuego utilizando tecnologías actuales y potentes como Unreal Engine 4.
2. Implementar lógica, mecánicas e IA con el sistema de Scripting Visual Blueprints.
3. Creación de un sistema de menús y HUD con el sistema de Scripting Visual Blueprints.
4. Realizar un pequeño estudio para ver los distintos juegos de este género en el mercado.
5. Después de haber realizado un motor gráfico propio poder utilizar un motor gráfico profesional como Unreal Engine 4 para intentar exprimirlo al máximo gráficamente.
6. Estudiar y analizar las ventajas y limitaciones que presenta Unreal Egine con respecto a otros motores gráficos.
7. Aprender cuales son las distintas herramientas que aporta Unreal Engine para poder desarrollar un videojuego de calidad.
8. Investigar y aprender el sistema de IA de Unreal Engine, puesto que el juego va a ser de un jugador contra la IA es necesario entender cómo funciona el Sistema de IA que usa Unreal Engine para poder así diseñar e implementar la IA de NOMBRE DE JUEGO AQUÍ.
9. Generar un documento de diseño para el videojuego (GDD, Game Design Document).
10. Crear modelados para un videojuego.
11. Texturizar los modelos realizados.
12. Importar y utilizar correctamente Assets a Unreal Engine.

13. Otro objetivo importante para mí es la parte gráfica, los acabados, puesto que con el motor gráfico elegido puedes obtener unos acabados bastante buenos quiero enfocarme mucho en que el juego se desarrolle en un escenario vistoso. Así que quiero aprovechar todas las ventajas que te aporta un motor gráfico potente como Unreal Engine para obtener unos acabados profesionales.
14. Añadir sonidos al videojuego.

## 4. Metodología

En este apartado se abarcan los aspectos de metodología de desarrollo (donde se hablara sobre las metodologías agiles y la metodología elegida para este proyecto) y gestión de proyectos (donde se hablara sobre cómo se gestiona el proyecto con esta metodología y las herramientas utilizadas).

En este apartado también se hablara aunque muy breve del control de versiones utilizado para este proyecto.

### 4.1 Metodología de desarrollo

#### 4.1.1 Metodologías Agiles

Por definición, las metodologías ágiles son aquellas que permiten adaptar la forma de trabajo a las condiciones del proyecto, consiguiendo flexibilidad y rapidez en la respuesta para adaptar el proyecto y su desarrollo a las circunstancias específicas del entorno, como por ejemplo cambios en las funcionalidades de un proyecto.

Hay una gran variedad de metodologías agiles (Scrum, Extreme Programming, Kanban,...), cada una tiene sus ventajas e inconvenientes, por lo que se trata de elegir la metodología que mejor se adapte al proyecto y al equipo.

La metodología elegida para este proyecto es Kanban ya que es una metodología fácil de usar y aplicar al proyecto, está pensada para equipos pequeños (en este caso el “equipo” es de 1 sola persona) y además ya se había trabajado con ella anteriormente y ha dado muy buenos resultados. Además una parte importante es que permite añadir nuevas tareas rápidamente y de un solo vistazo da gran información sobre las tareas que se están realizando, las que faltan por hacer y las que ya están hechas.

#### 4.1.2 Kanban

El objetivo de Kanban es gestionar de manera general como se van completando las tareas de un proyecto. Las principales reglas de Kanban son las siguientes:

- **Visualizar el trabajo y flujo de trabajo:**

Kanban se basan en el desarrollo incremental, dividiendo el trabajo en partes. Una de las principales aportaciones es que utiliza técnicas visuales para ver la situación de cada tarea. El trabajo se divide en partes, normalmente cada una de esas partes se escribe en un post-it y se pega en una pizarra. Los post-it suelen tener información variada, como una descripción y la estimación de la duración de la tarea.

La pizarra tiene tantas columnas como estados por los que puede pasar la tarea por ejemplo, en espera de ser desarrollada, en análisis, en desarrollo, en periodo de prueba, etc. El objetivo de esta visualización es que quede claro el trabajo a realizar y en que está trabajando cada persona con un simple vistazo a la pizarra.

- **Determinar el límite de trabajo en curso (WIP):**

Kanban dice que el trabajo en curso o WIP que son sus siglas en inglés (*Work In Progress*) debería estar limitado, por tanto el número máximo de tareas que se pueden realizar en cada fase debe ser algo conocido por ejemplo, como máximo 4 tareas en desarrollo, como máximo 1 en pruebas, etc. A esto se añade otra idea que sucede muy a menudo y es que para empezar con una nueva tarea alguna otra tarea previa debe haber finalizado.

- **Medir el tiempo en completar una tarea (*lead time*):**

El tiempo que se tarda en terminar cada tarea se debe medir, a ese tiempo se le llama *lead time*. El lead time cuenta desde que se añade una tarjeta hasta que se hace la entrega.

Aunque la métrica más conocida del Kanban es el *lead time*, normalmente se suele utilizar también otra métrica importante: el *cycle time*. El *cycle time* mide el tiempo en completar una tarea desde su inicio cuando comienza el trabajo hasta el final de este. Si con el *lead time* se mide lo que ven los clientes, lo que esperan, y con el *cycle time* se mide más el rendimiento del proceso.

## 4.2 Gestión de proyecto

Aunque en los apartados anteriores cuando hablábamos de Kanban se ha usado los términos post-it y pizarras, para la gestión del proyecto se van a utilizar herramientas virtuales por lo tanto los post-it pasaran a ser tarjetas y las pizarras pasaran a ser tableros virtuales.

La gestión del proyecto se ha realizado con la herramienta gratuita Trello, que como el mismo se define “*Trello es un gestor de tareas que permite el trabajo de forma colaborativa mediante tableros (board) compuestos de columnas (llamadas listas) que representan distintos estados. Se basa en el método Kanban para gestión de proyectos, con tarjetas que viajan por diferentes listas en función de su estado*”.

Para empezar se crearon las distintas columnas que representan los distintos estados por los que puede pasar una tarjeta (tarea), estas son:

- **TO DO:** aquí están las tareas que están por hacer, seguirán en este estado hasta que se empiece a trabajar sobre ellas y cambien al siguiente estado Working.
- **Working:** en esta columna se encuentran las tareas sobre las que se está trabajando en este momento.

- **Pruebas:** aquí estarán las tarjetas ya desarrolladas pero que están a esperar de realizar pruebas para comprobar que funcionan perfectamente antes de poder dejarlas ya en la columna de realizadas (*done*).
- **Done:** en esta columna estarán las tarjetas ya finalizadas y sobre las que no se tendrá que volver a trabajar.
- **Descartado:** aquí se encontraran aquellas tareas que se hayan descartado y ya no estén previstas para realizarse en el proyecto ya sea por temas de tiempo o cualquier otro motivo.
- **Ideas Futuras:** donde se colocan aquellas tareas que supondrían mejora del juego pero que son totalmente prescindibles y que por lo tanto solo se harán si hay tiempo suficiente.

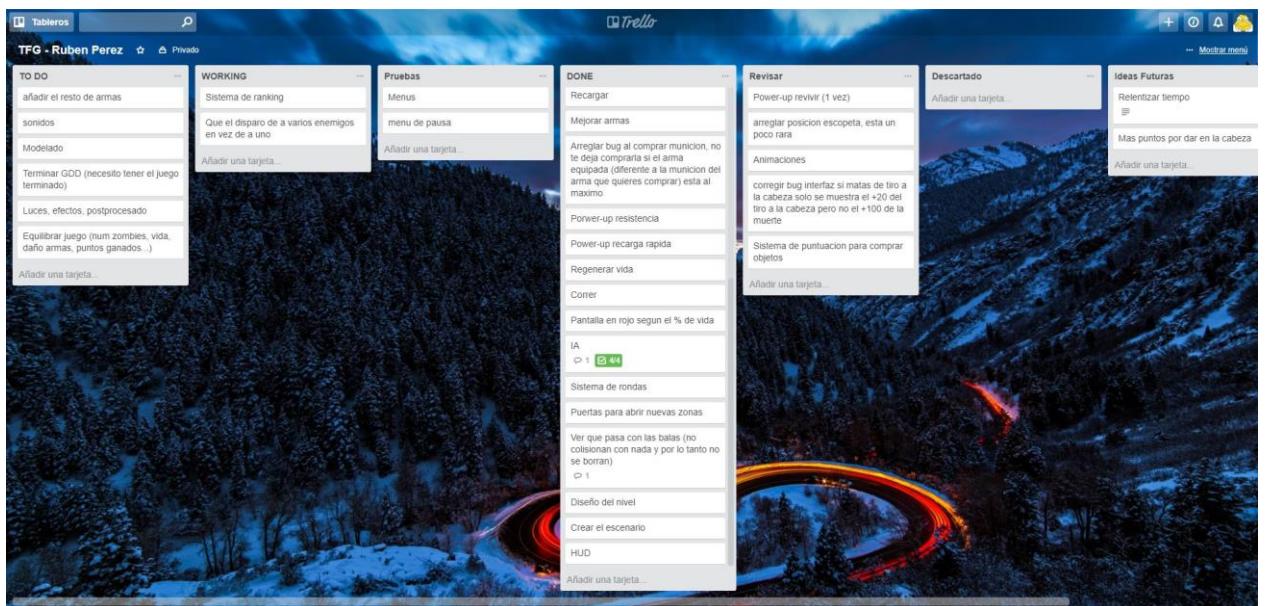


Figura 11: Trello del proyecto

#### 4.2.1 Toggl y la compatibilidad con Trello

Se trata de una aplicación que nos ayuda a gestionar el tiempo que dedicamos a cada proyecto, algo muy útil si trabajamos durante el día en varias tareas diferentes cuyo esfuerzo tenemos que medir. Solo tenemos que registrar las tareas, clasificarlas con etiquetas y pulsar el botón de inicio y de final cada vez que empezamos o finalizamos una, permitiendo de esa manera tener un informe con el tiempo exacto dedicado a cada tarea del proyecto.

Toggl también incluye funcionalidades para equipos de trabajo. Podemos invitar a nuestro equipo para que participen de nuestras tareas, ayudando así a determinar quién trabajó en qué y cuánto tiempo dedicó a cada actividad.

Lo más importante de esta herramienta es que es compatible con Trello, existe una extensión en Chrome que te permite integrar ambas herramientas pudiendo así desde el propio Trello iniciar el

temporizador de toggl para una tarea específica. De esta manera se puede tener un control total sobre las tareas que se están realizando y el tiempo que se está invirtiendo en ellas.

Además en Trello se puede establecer fechas de vencimiento lo que se puede complementar muy bien con esta herramienta, ya que puedes establecer una fecha de vencimiento y llegada la fecha ver el tiempo real que dedicaste a esa tarea, lo que te permite saber si hay problemas de planificación o de productividad.

Otro punto fuerte de esta aplicación es que te envía correos semanales con un resumen detallado de las horas invertidas y a que has dedicado esas horas (véase figura 12).



Figura 12: Email semanal de Toggl

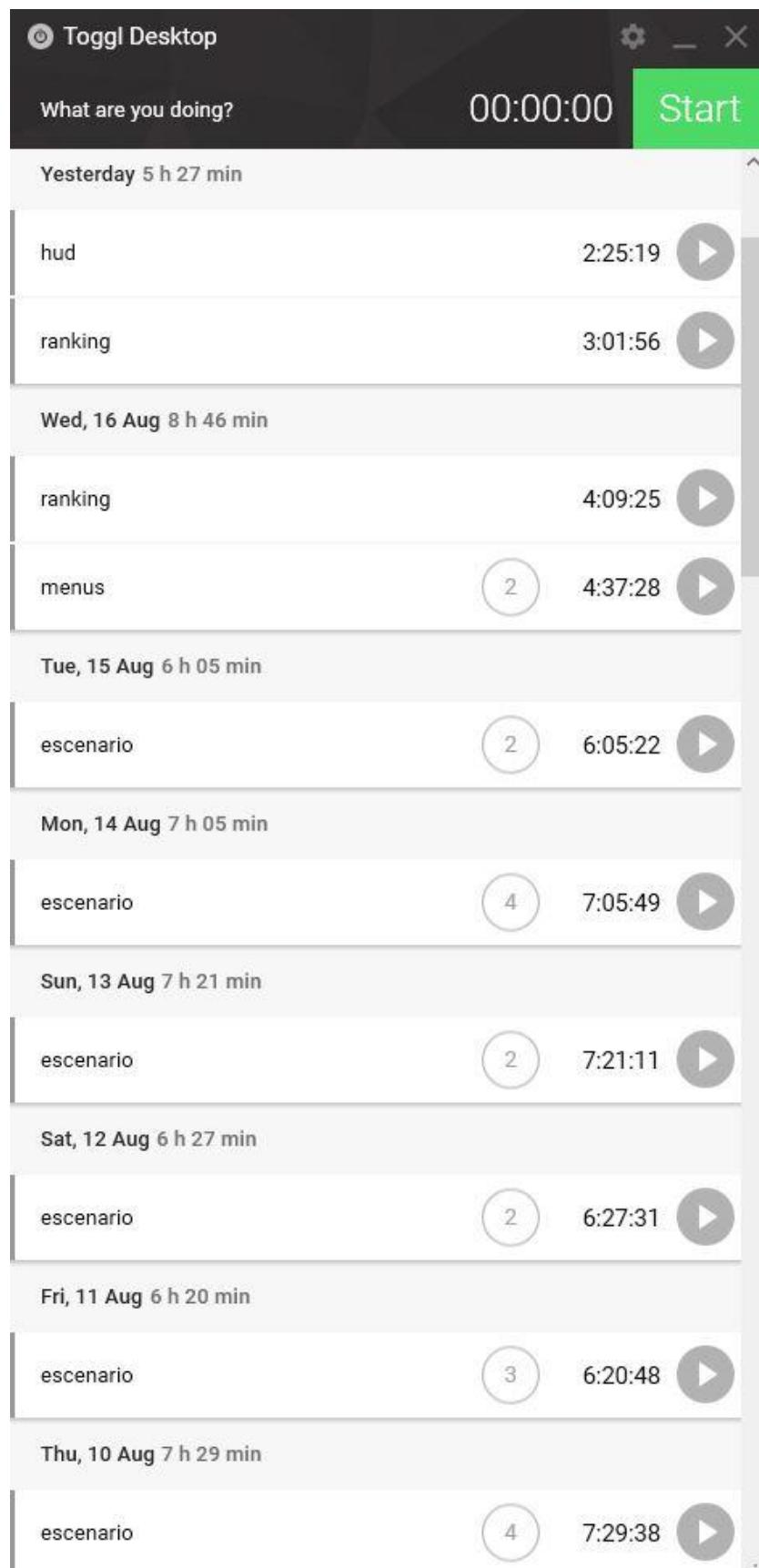


Figura 13: Toggl del proyecto

## 4.3 Control de versiones

Para este proyecto se ha utilizado **Git** como control de versiones ya que es muy importante utilizar un control de versiones para grandes proyectos. De esta manera nos evitamos problemas en un futuro (y algún que otro susto), permitiendo recuperar versiones anteriores en todo momento y así trabajar con tranquilidad sabiendo que si un día el proyecto deja de funcionar por algo siempre tenemos versiones anteriores que podemos recuperar.

Otra parte importante de esto es el hecho de poder trabajar en varias versiones a la vez, mediante **Git** podemos crear diferentes ramas en las que desarrollar las distintas mecánicas o funcionalidades, permitiendo aislar cada mecánica del resto y facilitando así el trabajo (ya que nos permite centrarnos únicamente en lo que estamos desarrollando). Una vez terminadas y comprobadas las mecánicas se incluirán en el proyecto, así podemos trabajar con distintas versiones a la vez y luego juntarlas. Si en algún momento una nueva mecánica o funcionalidad corrompe el proyecto es tan sencillo como descartar esa versión que se encuentra aislada en otra rama manteniendo así el resto del proyecto intacto.

Siguiendo esta filosofía de ir desarrollando en diferentes ramas y luego llevando versiones funcionales a master es como se ha ido trabajando, unas imágenes del estado de GitHub permitirán entender esto de manera más fácil.

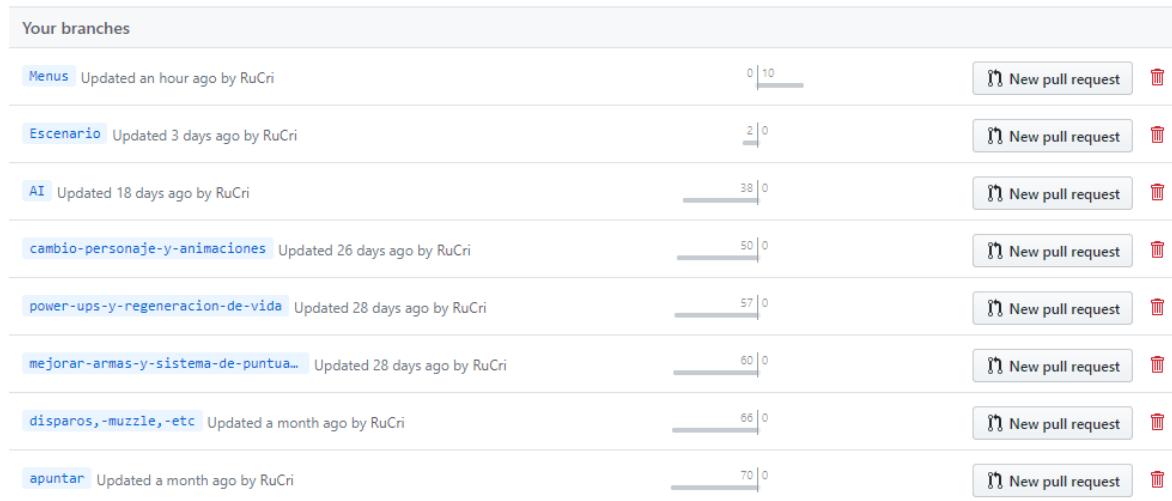


Figura 14: Ramas de GitHub del proyecto

Además de lo comentado antes, un plus que tiene usar Git es el hecho de acostumbrarte a este ya que actualmente se usa en la gran mayoría de empresas y te pedirán que uses Git vayas donde vayas, así que mucho mejor si ya lo conoces.

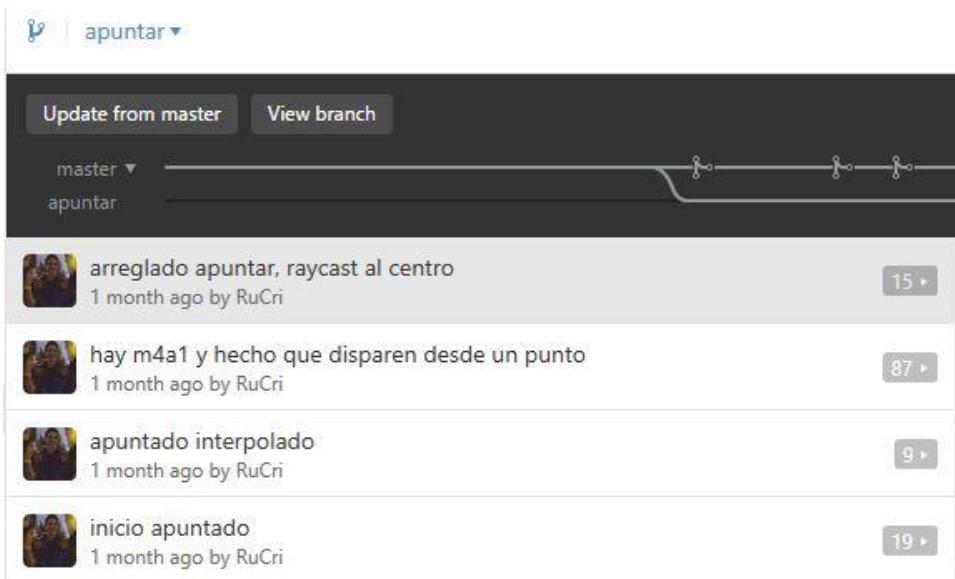


Figura 15: Commits a rama específica (Apuntar)

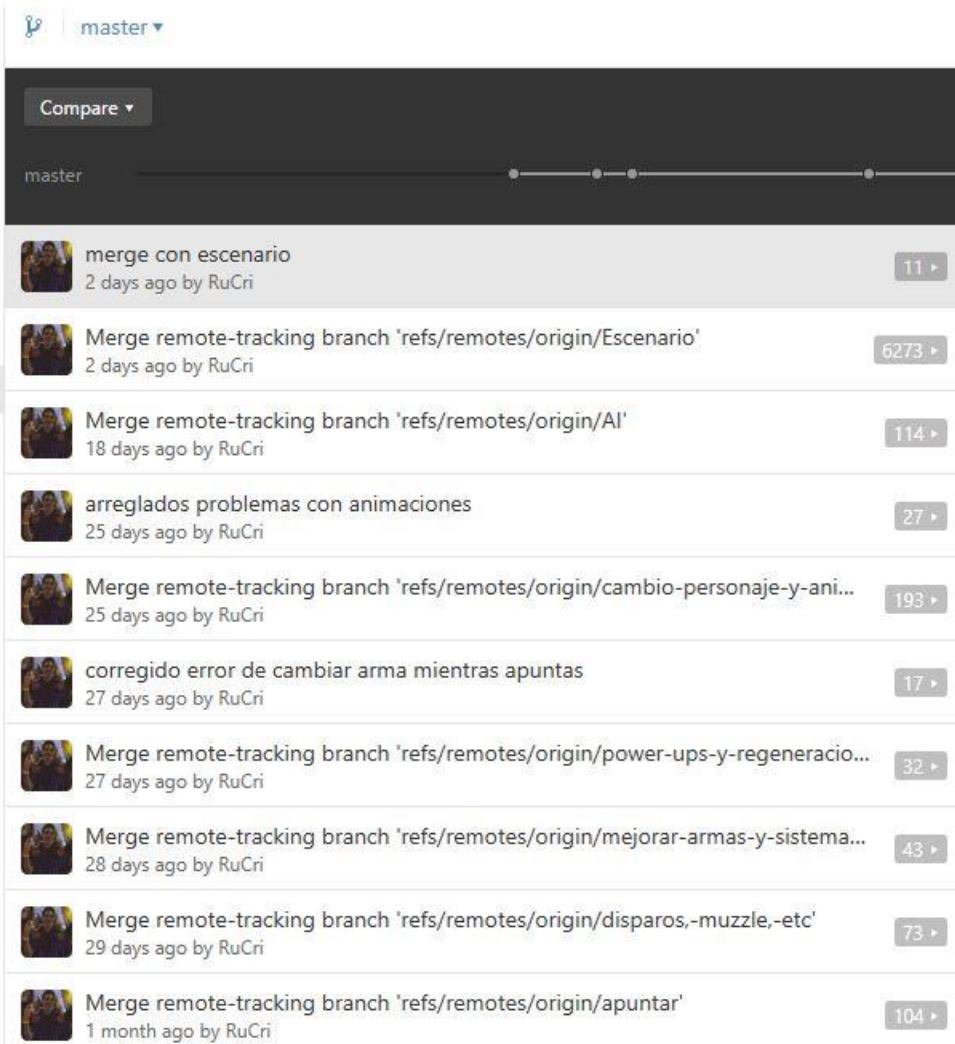


Figura 16: Llevando versiones funcionales de otras ramas a Master

## 5. Cuerpo del trabajo

En este bloque es la parte principal del documento, donde se describe con detalle el proceso completo del desarrollo del videojuego. Aunque antes de entrar en el desarrollo necesitamos saber que se va a desarrollar y con qué herramientas. Por esto este bloque se encuentra dividido en tres partes, que son:

1. La primera parte habla sobre las herramientas que se usarán para desarrollar el videojuego, un pequeño análisis de las distintas herramientas posibles y las herramientas seleccionadas. Con esta parte se intenta responder al “¿Con qué se va a desarrollar?”
2. La segunda parte intentará aclarar el “¿Qué se va a desarrollar?”, por eso esta parte es un **GDD (Game Design Document)**, en el que se plasmarán todos los aspectos del videojuego a desarrollar.
3. En esta tercera parte corresponde al “¿Cómo se ha desarrollado?”, ya que está dedicada exclusivamente al desarrollo e implementación del videojuego. No solo a la realización del videojuego con Unreal Engine sino a todo aquello que ha hecho falta para hacer el videojuego, como pueden ser texturas, modelos 3D, sonidos, etc.

### 5.1 Análisis, explicación y elección de herramientas

En este bloque se van a analizar las distintas herramientas disponibles para la creación de un videojuego y la explicación del porqué se ha elegido uno y no otros.

Como lo más importante es el motor de juego es en este en el que se va a hacer un análisis más exhaustivo, pero también se hablará sobre otras herramientas necesarias para la realización del videojuego como pueden ser los programas de modelado y texturizado. Ya que aunque el juego se haya realizado en Unreal Engine 4 para la creación de *assets* (recursos) se han necesitado muchas herramientas adicionales como edición de imágenes, creación y edición de modelados 3D, edición de audio, etc.

#### 5.1.1 Motor de videojuego

##### 5.1.1.1 Conceptos clave

Antes de analizar cuáles son los diferentes motores de videojuegos actuales es necesario saber qué es un motor de videojuegos.

**Software:** resumidamente un software es un conjunto de programas, instrucciones, reglas informáticas y rutinas que permiten a un ordenador realizar determinadas tareas.

**Motor de juego:** más conocido por su nombre en inglés como *Game Engine*. Un *Game Engine* es un software diseñado para la creación, desarrollo y representación de videojuegos. La funcionalidad básica de un motor es proveer al videojuego de un motor de renderizado para los gráficos 2D y 3D, **motor físico** (encargado de detectar colisiones, realizar cálculos como velocidad, etc), **sonidos, scripting** (programación) ya sea tanto en forma visual o de código, **animación, inteligencia artificial, redes** (motor de red), **streaming, administración de memoria** y un **escenario gráfico** (motor gráfico).

### 5.1.1.2 Motores de videojuegos actuales

Actualmente hay una gran cantidad de motores de videojuegos y es impensable analizarlos todos, por lo que nos vamos a centrar en los más populares y más usados hoy en día en la industria del videojuego como son Unreal Engine 4, Unity 3D y CryEngine.

V · T · E	Game engines (list)		[hide]
Source port · First-person shooter engine (list) · Tile engine · Game engine recreation (list) · Game creation system			
Free and open-source	2D	Adventure Game Studio · Beats of Rage · Cocos2d · Flixel · KiriKiri · libGDX · Moai · OHRRPGCE · OpenFL · ORX · Pygame · Ren'Py · Stratagus · Thousand Parsec · VASSAL · Xconq	
	2.5D	Aleph One · Cube Engine	
	3D	Away3D · Blender Game · Cafu · Crystal Space · Cube 2 Engine · Delta3D · Dim3 · GamePlay · GLScene · Horde3D · Irrlicht · id Tech (1 2 3 4) · JMonkey · OGRE · Open Wonderland · Panda3D · Papervision3D · Platinum Arts Sandbox Free 3D Game Maker · PlayCanvas · PLIB · Quake (II) · Torque 3D	
	Mix	Allegro · Construct Classic · Godot · Lightweight Java Game Library · Spring · Wintermute Engine	
Proprietary	2D	Construct 2 · Corona · Clickteam Fusion · GameMaker: Studio · GameSalad · M.U.G.E.N · NScripter · RPG Maker · Southpaw · Stencyl · UbiArt Framework · Vicious · Virtual Theatre · V-Play · Zillions of Games	
	3D	4A · Amazon Lumberyard · Anvil · Bork3D · C4 · Chrome · Clausewitz Engine · Creation · CryEngine · Crystal Tools · Decima · Diesel · EAGL · EGO · Elflight · Enforce · Enigma · Essence · Flare3D · Fox · Frostbite · Geo-Mod · GoldSrc · HeroEngine · HydroEngine · id Tech (5 6) · Ignite · IW · Jade · Kinetica · LS3D · LithTech · Luminous Studio · LyN · Marmalade · Mizuchi · MT Framework · Outerra · Panta Rhei · PhyreEngine · Plasma · Q · Real Virtuality · REDEngine · Refractor · Riot · RAGE · SAGE · Serious · Shark 3D · ShiVa · Silent Storm · Snowdrop · Source (2) · Titan · TOSHI · Truevision3D · Unigine · Unity · Unreal · Vision · Visual3D · XnGine · X-Ray · YETI · Zero	
	Mix	Gamebryo · Hybrid Graphics · Kaneva Game Platform · Metismo	
Historical	BRender · Build · Dark · Doom · Game-Maker · GameMaker · Garry Kitchen's GameMaker · Genesis3D · Genie · Gold Box · Filmation · Freescape · INSANE · Jedi · MADE · Pie in the Sky · RenderWare · SCUMM · Sim RPG Maker · Sith · Voxel Space · Wolfenstein 3D		
Proprietary middleware	Euphoria · Gameware · GameWorks · Havok · iMUSE · Kynapse · SpeedTree · Xaitment · FaceGen		

Figura 17: Motores de juegos actuales

Tanto esta tabla como la siguiente (dividida en 2 imágenes) están sacadas de Wikipedia. Las imágenes siguientes muestran más en detalle los motores de juegos actuales.

Nombre	Lenguaje de programación	Scripting	Plataformas Múltiples - Cross-Platform	SDL	Orientación 2D/3D	Plataforma
3D Rad	C#	AngelScript	✗ No	✗ No	✓ 3D	Windows
Adventure Game Studio	C++	AGSScript	✓ Sí	✗ No	✗ 2D	Windows Linux
Aleph One	C++	Lua, Marathon markup language	✓ Sí	✓ Sí	✗ 2.5D	Windows Linux OS X
Allegro library	C	Ada, C++, C#, D, Lisp, Lua, Mercury, Pascal, Perl, Python, Scheme	✓ Sí	✓ Sí	✗ 2D	Windows Linux OS X DOS
Angel2D <sup>g</sup>	C++	Lua	✓ Sí	✗ No	✗ 2D	Windows Linux OS X iOS
Ardor3D	Java		✓ Sí	✗ No	✓ 3D	cross-platform
Axiom Engine	C#		✓ Sí	✗ No	✓ 3D	Windows Linux OS X Solaris
Blender	C++	Python	✓ Sí	✓ Sí	✓ 3D	Windows Linux OS X Solaris
Build engine	C		✗ No	✗ No	✗ 2.5D	Windows Linux OS X DOS
Cafu Engine	C++	Lua	✓ Sí	✗ No	✓ 3D	Windows Linux OS X
ClanLib	C++		✓ Sí	✓ Sí	✗ 2.5D	Windows Linux OS X
Cocos2d	C++, Python, Objective-C	JavaScript, Java	✓ Sí	✗ No	✗ 2D	Windows Linux OS X iOS
Construct Classic	Event Based		✗ No	✗ No	✗ 2D	Windows
Core3D <sup>g</sup>	Objective-C		✓ Sí	✓ Sí	✓ 3D	Windows Linux OS X iOS
CRM3DPro SDK <sup>g</sup>	C/C++		✓ Sí	✓ Sí	✗ 2D	Windows Linux OS X
Crystal Space	C++	Java, Perl, Python	✓ Sí	✓ Sí	✓ 3D	Windows Linux OS X
Cube	C++		✓ Sí	✓ Sí	✓ 3D	Windows Linux OS X
Cube 2	C++	Cubescript	✓ Sí	✓ Sí	✓ 3D	Windows Linux OS X
Delta3d	C++	Python	✓ Sí	✗ No	✗ 2.5D	cross-platform
Dim3	C++	JavaScript	✓ Sí	✗ No	✓ 3D	cross-platform
DimensioneX Multiplayer Engine	Java	Java, VBscript	✓ Sí	✗ No	✗ 2.5D	cross-platform
Div GO: Games Online	HTML5 Javascript PHP	DIV Games Studio	✓ Sí	✗ No	✓ 2D ✓ 3D	Windows Linux OS X
Dreemchest <sup>g</sup>	C++	Lua	✓ Sí	✗ No	✗ 2D	Windows, OS X, Android, iOS, Flash
Duality <sup>g</sup>	C#	Plugin-based	✗ No	✗ No	✗ 2D	Windows
Eclipse Origins <sup>g</sup>	Visual Basic 6		✗ Exclusivo para Windows	✗ No	✗ 2D	Windows
ENIGMA	C++	EDL	✓ Sí	✗ No	✗ 2D	Windows Linux OS X
Env3D	Java		✓ Sí	✗ No	✓ 3D	cross-platform
Exult	C++		✓ Sí	✓ Sí	✗ 2D	Windows Linux OS X BSD
FLARE <sup>g</sup>	C++		✓ Sí	✓ Sí	✗ 2D	Windows Linux OS X BSD
Flexible Isometric Free Engine	C++	Python	✓ Sí	✓ Sí	✗ 2D	Windows Linux OS X BSD
Flixel	ActionScript		✓ Sí	✗ No	✗ 2D	
GameKit (OgreKit)	C++	Lua	✓ Sí	✗ No	✓ 3D	Windows Linux OS X android
GameMaker: Studio	Delphi	GML	✓ Sí	✗ No	✗ 2D	Windows Linux OS X android iOS Xbox 360 PlayStation 4 PlayStation 3 PlayStation Vita HTML5 Tizen
GamePlay3D <sup>g</sup>	C++	Lua	✓ Sí	✗ No	✓ 3D	Windows Linux OS X iOS BlackBerry 10 Android
Canvas	JavaScript	JavaScript	✓ Sí	✗ No	✗ 2D	HTML5
Grit <sup>g</sup>	C++	Lua	✓ Sí	✗ No	✓ 3D	
Haaf's Game Engine (HGE) <sup>g</sup>	C++	C, Go	✓ Sí	✓ Sí	✗ 2D	Windows Linux OS X
HGamer3D <sup>g</sup>	Haskell		✗ No	✗ No	✓ 3D	
Horde3D <sup>g</sup>	C++		✓ Sí	✗ No	✓ 3D	Windows
HPL 1 engine	C++	AngelScript	✓ Sí	✓ Sí	✓ 3D	Windows Linux OS X
id Tech 1 (Doom)	C	ACS	✓ Sí	✗ No	✗ 2.5D	Windows Linux OS X
id Tech 1 (Quake)	C	QuakeC	✓ Sí	✗ No	✓ 3D	Windows Linux OS X
id Tech 2	C	C	✓ Sí	✗ No	✓ 3D	Windows Linux OS X

Figura 18: Tabla motores de juego actuales (1/2)

<a href="#">id Tech 3</a>	C	Game Data [PK3]	✓ Sí	✗ No	✓ 3D	Windows Linux OS X
<a href="#">id Tech 4</a>	C++	via DLLs	✓ Sí	✗ No	✓ 3D	Windows Linux OS X
<a href="#">IndieLib</a>	C++		✓ Sí	✓ Sí	✓ 2.5D	Windows Linux OS X
<a href="#">ioquake3</a>	C		✓ Sí	✓ Sí	✓ 3D	Windows Linux OS X
<a href="#">IwGame Engine</a>	C++		✓ Sí	✗ No	✗ 2D	Windows Linux OS X iOS Android
<a href="#">Jake2</a>	Java		✓ Sí	✗ No	✓ 3D	Cross-platform
<a href="#">JGame</a>	Java, Actionscript 3		✓ Sí	✗ No	✗ 2D	J2ME Android
<a href="#">jMonkeyEngine</a>	Java		✓ Sí	✓ Sí	✓ 3D	Cross-platform
<a href="#">Jogre</a>	Java		✓ Sí	✗ No	✗ 2D	Cross-platform
<a href="#">JPCT and JPCT-AE</a>	Java		✓ Sí	✗ No	✓ 3D	Java Android
<a href="#">Kobold2D</a>	Objective-C	Lua	✓ Sí	✗ No	✗ 2D	OS X iOS
<a href="#">Libgdx</a>	Java		✓ Sí	✗ No	✓ 3D	Windows Linux OS X iOS Android HTML5
<a href="#">Linderdaum Engine</a>	C++	C#, LinderScript	✓ Sí	✗ No	✓ 3D	Windows OS X iOS Android BlackBerry 10
<a href="#">LOVE</a>	Lua	Lua	✓ Sí	✓ Sí	✗ 2D	Windows Linux OS X
<a href="#">LWJGL</a>	Java		✓ Sí	✓ Sí	✓ 3D	
<a href="#">Maratis</a>	C++	Lua	✓ Sí	✗ No	✓ 3D	Windows Linux OS X iOS Android
<a href="#">melonJS</a>	Javascript	Javascript	✓ Sí	✗ No	✓ 2.5D	HTML5
<a href="#">Moai SDK</a>	C++	Lua	✓ Sí	✓ Sí	✗ 2D	Windows OS X iOS Android
<a href="#">Multiverse Foundation</a>	Python and Java	Python	✗ No	✗ No	✓ 3D	
<a href="#">Nebula Device</a>	C++	Java, Python, Lua, Tcl/Tk	✓ Sí	✗ No	✓ 2.5D	Windows Linux
<a href="#">NetGore</a>	C#		✓ Sí	✗ No	✗ 2D	Windows Linux
<a href="#">NME</a>	Haxe		✓ Sí	✗ No	✗ 2D	Windows Linux OS X iOS Android BlackBerry
<a href="#">nxPascal</a>	Object Pascal	Delphi, Lazarus	✓ Sí	✗ No	✓ 3D	
<a href="#">OpenSimulator</a>	C#	LSL	✓ Sí	✗ No	✓ 3D	
<a href="#">ORX</a>	C/C++	Custom	✓ Sí	✓ Sí	✓ 2.5D	Windows Linux Mac OS X iOS Android
<a href="#">Oxygine</a>	C++		✓ Sí	✓ Sí	✗ 2D	Windows Linux Mac OS X iOS Android
<a href="#">Panda3D</a>	C++	Python	✓ Sí	✗ No	✓ 3D	Windows Linux OS X iOS
<a href="#">PixelLight</a>	C++	AngelScript, Lua, Python, Javascript/V8	✓ Sí	✗ No	✓ 3D	Windows Linux Android
<a href="#">PLIB</a>	C++		✓ Sí	✓ Sí	✓ 3D	Windows Linux OS X
<a href="#">Polycode</a>	C++	Lua	✓ Sí	✓ Sí	✓ 3D	Windows Linux OS X
<a href="#">PowerBite PackApp Engine</a>	Ninguno	Ninguno	✓ Sí	✗ No	✗ No	Windows
<a href="#">Pyrogenesis</a>	C++	JavaScript	✓ Sí	✓ Sí	✓ 3D	Windows
<a href="#">Retribution Engine</a>	C++		✗ No	✗ No	✓ 3D	Windows
<a href="#">SFML</a>	C++		✓ Sí	✗ No	✗ 2D	
<a href="#">Sge2d</a>	C		✓ Sí	✓ Sí	✗ 2D	cross-platform
<a href="#">Source</a>	C++		✓ Sí	✓ Sí	✗ 2D	Windows
<a href="#">Spring</a>	C++, Java/JVM, Lua, Python		✓ Sí	✓ Sí	✓ 3D	Windows Linux OS X
<a href="#">StepMania</a>	C++	Lua	✓ Sí	✗ No	✓ 3D	cross-platform
<a href="#">StormEngineC</a>	JavaScript	JavaScript	✓ Sí	✗ No	✓ 3D	HTML5
<a href="#">Stratagus</a>	C	Lua	✓ Sí	✓ Sí	✗ 2D	Linux
<a href="#">Torque3D</a>	C++	TorqueScript	✗ No	✗ No	✓ 3D	Windows Linux OS X
<a href="#">Turbulence</a>	TypeScript	JavaScript	✓ Sí	✗ No	✓ 3D	HTML5
<a href="#">Unity3D</a>	C#	C#, JavaScript, Boo	✓ Sí	✓ Sí	✓ 3D	Windows Linux OS X Android
<a href="#">Unreal Engine</a>	C++	C++, UnrealScript	✓ Sí	✓ Sí	✓ 3D	Windows Linux OS X
<a href="#">WiMi5</a>	HTML5	JavaScript, HTML5	✓ Sí	✗ No	✓ 2D	Windows Linux OS X cross-platform
<a href="#">Wire3D</a>	C++		✓ Sí	✗ No	✓ 3D	Windows Wii
<a href="#">WorldForge</a>	C++	Lua(client), Python(server)	✓ Sí	✓ Sí	✓ 3D	
<a href="#">ZenGL</a>	Pascal	C, C++	✓ Sí	✗ No	✗ 2D	GNU/Linux Windows, Mac OS X iOS Android

Figura 19: Tabla motores de juego actuales (2/2)

### 5.1.1.2.1 Cry Engine 3

**CryEngine** es un motor de juego creado por la empresa alemana desarrolladora de software Crytek, originalmente un motor de demostración para la empresa Nvidia, que al demostrar un gran potencial se implementa por primera vez en el videojuego Far Cry, desarrollado por la misma empresa creadora del motor. El 30 de marzo de 2006, la totalidad de los derechos de *CryEngine* son adquiridos por la distribuidora de videojuegos Ubisoft.



Figura 20: Logo Cry Engine

Este motor es muy potente, tiene una gran capacidad y potencia tanto en aspectos gráficos como en el cálculo de físicas, además del increíble realismo en la iluminación de escenas en tiempo real. Pero presenta un gran inconveniente y es que tiene una gran curva de aprendizaje, que te hará invertir una gran cantidad de horas para empezar a “dominar” el programa.

Otra gran ventaja de CryEngine es que anteriormente tenía un modelo de negocio basado en pagos mensuales sin ningún tipo de royalty que daba la opción de crear juegos y ponerlos en el mercado sin la preocupación de tener que pagar en el caso de que el juego tenga un gran éxito. Pero actualmente el modelo de negocio es “paga lo que quieras” y con los royalties libres con lo cual es una buena opción para un producto de calidad y con bajo coste (o coste 0).

### 5.1.1.2.2 Unreal Engine 4

El 5 de noviembre del 2009 Epic Games publicó una versión gratuita del Unreal Engine 3: El Unreal Development Kit (**UDK**) para permitir a grupos de desarrolladores amateur realizar juegos con el Unreal Engine 3. Unreal Engine 4 surge entonces como el sucesor del UDK.



Figura 21: Logo Unreal Engine 4

Como sucesor UE4 (Unreal Engine 4) trae importantes mejoras con el objetivo de mejorar la facilidad de producción de videojuegos. Cabe destacar grandes cambios para el usuario entre el UDK y Unreal Engine 4, y es que aunque UE4 no es fácil de usar su sencillez respecto al UDK es muy notoria. Algunos de estos cambios son:

El lenguaje de scripting en UE4 ha cambiado, anteriormente se usaba el lenguaje UnrealScript (utilizado únicamente en los motores de Unreal), este lenguaje ha sido completamente sustituido por C++. Esto es una gran mejora ya que antes era difícil el comienzo en Unreal teniendo que aprender un nuevo lenguaje a la vez que te adaptas a un motor de juego que no es sencillo de usar.

UE4 dice adiós al Kismet que era el sistema de scripting visual que usaba en UE3 y UDK, este es sustituido por el Blueprint que es una versión avanzada y más intuitiva que Kismet con el que puedes crear un juego sin escribir código.

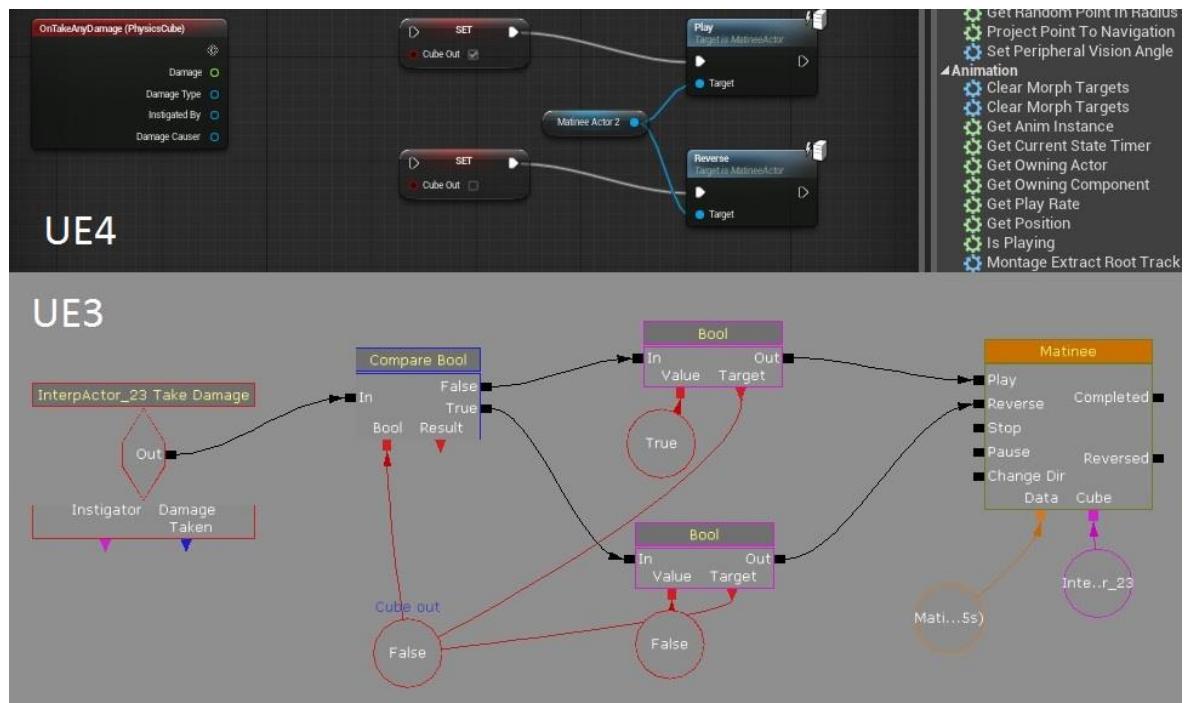


Figura 22: Blueprint vs Kismet

### 5.1.1.2.3 Unity 3D

**Unity** es un motor de videojuego multiplataforma creado por Unity Technologies. Unity está disponible como plataforma de desarrollo para Microsoft Windows, OS X, Linux. La plataforma de desarrollo tiene soporte de compilación con diferentes tipos de plataformas y esto tal vez es uno de los puntos más fuertes de Unity y es que permite a un desarrollador crear un juego para una gran cantidad de plataformas usando el mismo código.



Figura 23: Logo Unity

Este motor tiene la fama de ser el más sencillo para empezar a realizar videojuegos, es muy utilizado y aunque ha estado considerado durante un buen tiempo como motor para desarrollar juegos exclusivamente móviles, esto cambia a partir de Unity 5 donde la mejora de los gráficos y optimizaciones permiten crear juegos con gráficos realistas. De esta manera entra a competir con motores como Unreal Engine 4 y CryEngine.

Otros aspectos importantes de Unity es en primer lugar es que permite programar scripts tanto en C# como en JavaScript. En segundo lugar hay que destacar que es compatible con principales aplicaciones de modelado y animación (al igual que UE4) como Maya, Blender, 3DS Max, lo que facilita la creación y utilización de assets (recursos).

La parte mala de Unity viene con licencias, y es que tiene 2 licencias una gratuita y una pro de pago. El inconveniente es que en la versión gratuita no tenemos acceso a todas las características del motor mientras que en otro como UE4 sí que tenemos acceso a todas las características desde el principio.

### **5.1.1.3 Elección del motor de juego**

La elección del motor no ha resultado una decisión fácil, ya que de él depende la mayor parte del desarrollo de un videojuego. Para tomar una decisión se han tenido en cuenta una serie de aspectos importantes como:

- Conocimientos previos del motor
- Documentación
- Precio
- Comunidad de usuarios activa
- Posibilidad de mecánicas
- Potencial gráfico
- Curva de aprendizaje

Tras estudiar estos puntos se ha llegado a la conclusión de que Unreal Engine 4 es el motor de juego más apropiado para el desarrollo de este proyecto. Ya que aunque gráficamente la potencia de CryEngine es insuperable y su precio es nulo, la curva de aprendizaje de este motor es un gran inconveniente sumado a que la comunidad de usuarios activa es más bien reducida en comparación con otros como Unreal Engine 4.

Además el juego va a ser un FPS y Unreal surgió en un primer lugar para la creación de juegos FPS por lo tanto el motor de juego se adapta perfectamente a las mecánicas que se quieren implementar.

En cuanto a la documentación esta es muy extensa, hay una gran cantidad de artículos, libros, y tutoriales por toda la web, además de que cuenta con una de las comunidades de usuarios más activa, con un foro oficial en el que expertos responden a las dudas en un margen de tiempo reducido y una gran cantidad de videotutoriales en Youtube.

Otra parte importante es su precio, y es que Unreal junto con todas sus herramientas es totalmente gratuito y solo tendremos que pagar un 5% de nuestros beneficios si superamos la cantidad de 3000\$ dólares americanos.

Finalmente el aspecto que acabo de convencerme para la elección de este motor es que ya había realizado 2 proyectos con él, y aunque estos proyectos fueran mucho más pequeños que este me han servido para tener una pequeña base y no ser nuevo en el sistema de scripting visual (**Blueprint**).

## 5.1.2 Otras herramientas

Una vez ya se ha tomado la decisión más importante que era la de elegir el motor de juego a continuación se va a escoger algunos de los programas que se utilizaran para la creación del contenido del juego (assets).

### 5.1.2.1 Modelado 3D

En gráficos 3D por ordenador, el modelado 3D es el proceso de desarrollo de una representación matemática de cualquier objeto tridimensional (ya sea inanimado o vivo) a través de un software especializado. Al producto se le llama modelo 3D.

Se han estudiado los siguientes programas de modelado 3D.

#### 5.1.2.1.1 Blender

Es un programa informático multiplataforma, dedicado especialmente al modelado, iluminación, renderizado, animación y creación de gráficos tridimensionales. Se desarrolla como Software Libre, con el código fuente disponible. Su uso es completamente **gratuito**.



Figura 24: Logo Blender

Características principales:

- Software libre, gratuito y multiplataforma
- Potente y versátil
- Importa y exporta gran cantidad de formatos 3D
- Soporte gratuito vía [blender3d.org](http://blender3d.org)
- Manual multilenguaje en línea
- Una comunidad activa bastante grande
- Múltiples plugins también gratuitos que expanden las posibilidades del programa
- Compatible con Unity, Unreal y CryEngine
- Posibilidad de usar y modificar el código fuente.

### **5.1.2.1.2 3DS Max**

Es un programa de creación de gráficos y animación 3D desarrollado por Autodesk.



*Figura 25: Logo 3DS Max*

Características:

- Multiplataforma
- Interoperabilidad con Photoshop y After Effects
- Posibilidad de realizar animaciones 3D gracias a sus herramientas de animación.
- Capacidad de modelado mediante mecanismos sofisticados.
- Gran cantidad de herramientas y primitivas geométricas.
- Una comunidad muy extensa con gran cantidad de tutoriales.

El mayor inconveniente de este software es que no es gratuito. Su licencia cuesta 200 euros mensuales, aunque hay disponibles licencias gratuitas para estudiantes.

### **5.1.2.1.3 Maya**

Maya es un programa informático dedicado al desarrollo de gráficos 3D por ordenador, efectos especiales y animación.



*Figura 26: Logo Maya*

Se caracteriza por su potencia y las posibilidades de expansión y personalización de su interfaz y herramientas. MEL (Maya Embedded Language) es el código que forma el núcleo de Maya y gracias al cual se pueden crear scripts y personalizar el paquete. El programa posee diversas herramientas para modelado, renderización, simulación de ropa y cabello, dinámicas (simulación de fluidos) y sobre todo ofrece una gran facilidad para la animación.

El inconveniente al igual que en 3DS Max es que es un software de pago. Su licencia cuesta 242 € mensuales, aunque al igual que el anterior dispone de una licencia gratuita para estudiantes.

### 5.1.2.2 Texturizado

#### 5.1.2.2.1 Mudbox

Mudbox es un software de modelado 3D, texturado y pintura digital, actualmente desarrollado por Autodesk. Aunque se define como un software de modelado 3D este programa es más comúnmente utilizado para el texturizado de modelos 3D realizados previamente con otros programas como 3DS Max.



Figura 27: Logo Mudbox

La interfaz de usuario de Mudbox se encuentra en un ambiente 3D que permite la creación de cámaras móviles y personalizables, edición de mallas poligonales y subdivisión de objetos. Mudbox puede importar y exportar archivos .obj, .fbx, .bio, así como su propio formato, .mud. Permite el uso de capas 3D para una visualización rápida del diseño, escultura no destructiva y soporte para un elevado número de polígonos.

#### 5.1.2.2.2 Substance Painter

Substance Painter es un software de pintura en 3D que le permite texturizar, renderizar y exportar su trabajo. Es compatible con los principales motores de juegos como Unity y Unreal Engine.



Figura 28: Logo Substance Painter

Es bastante parecido a Mudbox, permite importar modelos 3D de otros programas para poder texturizarlos, y luego exportar el mapa de texturas para poder incluirlo en otros programas como puede ser en mi caso Unreal Engine.

En cuanto a las características son similares a las de Mudbox.

### **5.1.2.3 Elección de herramientas**

Una vez analizadas las herramientas anteriores toca elegir con cuales de ellas vamos a trabajar.

Ya que el modelado y texturizado no es uno de los objetivos principales del proyecto y no se le va a dar mucha importancia la elección de estas herramientas viene dada en mayor medida por el conocimiento previo del software y el tiempo que tendría que invertir en aprender otro software.

Por lo tanto las herramientas que voy a utilizar son 3DS Max y Mudbox debido a que ya he trabajado con ellas anteriormente y puesto que las diferencias con las otras no son muy importantes y cualquiera de las seleccionadas me sirve para el objetivo que es el de modelar y texturizar algún objeto 3D he decidido basarme en los conocimientos previos sobre el programa para la elección de este.

Hay que recordar que aunque tanto Mudbox como 3DS Max son programas de pago, voy a usar la licencia de estudiante por lo que puedo usar el programa de forma gratuita.

## 5.2 Documento de Diseño del Videojuego (GDD)

Un documento de diseño de videojuego más conocido como GDD por sus siglas en inglés (Game Design Document) es un documento creado por los desarrolladores (todos en conjunto, artistas, programadores, diseñadores...) que se utilizará como guía durante todo el proceso de desarrollo del juego. En este documento se plasman todos los aspectos importantes para el desarrollo del videojuego como pueden ser la historia, los personajes, mecánicas, objetivo del juego, inteligencia artificial, diseño de niveles, etc.

Por lo tanto en este bloque se encontrara todo lo relacionado al diseño del videojuego en su apartado teórico.

El nombre elegido para el videojuego es TODO NOMBRE DE VIDEOJUEGO AQUÍ

TODO LOGO DEL VIDEOJUEGO AQUÍ

### 5.2.1 El juego en términos generales

#### 5.2.1.1 Resumen de argumento

Nos encontramos en un ambiente donde Corea del Norte sigue con su afán de expandir su poder y conquistar nuevos territorios afines al régimen de coreano, las relaciones internacionales ya deterioradas no sirven de nada, no hay posibilidad de negociación y el temor empieza a brotar en la población que se preparan para lo peor. En este clima de crispación, temor y desesperación los países empiezan su armamento nuclear.

El mundo dividido ahora en 2 regiones se ve envuelto en una gran guerra nuclear, el planeta entero es víctima de distintos ataques nucleares de ambos bandos, cuando estos cesan se descubre la gravedad del asunto. La existencia de la humanidad está en peligro.

Nosotros somos un alto mando militar que viendo la gravedad de la situación lucha por su supervivencia, contra lo que antes eran personas, pero que ahora debido a la radiación tan elevada se han convertido en muertos vivientes.

Hemos sobrevivido gracias a que estábamos en un refugio nuclear desde donde dirigíamos a los bombarderos, por ello tenemos la esperanza de encontrar más supervivientes aliados que estuvieran en la misma situación que nosotros.

Nuestro objetivo entonces está bien claro ¡SOBREVIVIR!

### **5.2.1.2 Conjunto de características**

Las principales características de este juego muchas de ellas compartidas con la gran mayoría de juegos de este género (FPS) son:

- Cámara en primera persona
- Sensación de inmersión que te hace sentir dentro del juego debido a la cámara en primera persona y la ambientación de este.
- Gran ambientación
- Gráficos realistas
- Mecánicas sencillas (las típicas de un FPS)
- Desarrollado con Unreal Engine 4
- Armas que existen en la realidad (para conseguir un mayor realismo)
- Gran cantidad de enemigos

### **5.2.1.3 Género**

TODO NOMBRE DE JUEGO AQUÍ se clasificaría dentro del género **FPS** ya que como el resto de juegos de este género el mundo se ve a través de una cámara en primera persona que representa la vista del protagonista y tenemos armas que podremos usar para ir acabando con los enemigos.

Aunque es un FPS el objetivo de este juego es sobrevivir el mayor tiempo posible, por lo tanto tendría un toque de *survival* aunque no se podría considerar dentro de este género.

De la misma forma que aunque la temática y ambientación pretende ponerte los pelos de punta y hacerte sentir inquieto, jugando con los sonidos, luces y sombras para intentar estar en un entorno un poco terrorífico este juego no podría considerarse tampoco dentro del género de *Survival Horror*.

### **5.2.1.4 Clasificación PEGI**

#### **5.2.1.4.1 ¿Qué es PEGI?**

PEGI es el sistema de clasificación por edades para los videojuegos donde se informa a los padres o cualquier consumidor del producto de la edad apropiada para el contenido del juego pero sin tener en cuenta el nivel de dificultad de este.

La etiqueta PEGI indica la edad mínima necesaria para poder jugar a un juego.

### 5.2.1.4.2 ¿Qué entendemos por clasificación?

Como dice PEGI en su página web “*La clasificación por edad es un sistema destinado a garantizar que el contenido de los productos de entretenimiento, como son las películas, los vídeos, los DVD y los juegos de ordenador, sea etiquetado por edades en función de su contenido. Las clasificaciones por edades orientan a los consumidores (especialmente a los padres) y les ayudan a tomar la decisión sobre si deben comprar o no un producto concreto.*”

Por lo tanto la clasificación de un juego confirma que es adecuado para jugadores que han cumplido una determinada edad. Así pues, un juego PEGI 7 sólo será adecuado para quienes tengan 7 o más años de edad y un juego PEGI 18 sólo será apto para adultos mayores de 18 años. La clasificación PEGI tiene en cuenta la idoneidad de la edad de un juego, no su nivel de dificultad.

### 5.2.1.4.3 ¿Cómo se mide la clasificación?

La clasificación se mide mediante una serie de descriptores. Los descriptores que aparecen en el reverso de los estuches indican los motivos principales por los que un juego ha obtenido una categoría de edad concreta. Existen ocho descriptores:

	<b>Lenguaje soez</b>	El juego contiene palabrotas
	<b>Discriminación</b>	El juego contiene representaciones discriminatorias, o material que puede favorecer la discriminación
	<b>Drogas</b>	El juego hace referencia o muestra el uso de drogas
	<b>Miedo</b>	El juego puede asustar o dar miedo a niños
	<b>Juego</b>	Juegos que fomentan el juego de azar y apuestas o enseñan a jugar
	<b>Sexo</b>	El juego contiene representaciones de desnudez y/o comportamientos sexuales o referencias sexuales
	<b>Violencia</b>	El juego contiene representaciones violentas
	<b>En línea</b>	El juego puede jugarse en línea

Figura 29: Descriptores clasificación PEGI

#### 5.2.1.4.4 Clasificación PEGI de NOMBRE DE JUEGO AQUI

La clasificación de NOMBRE DE JUEGO AQUÍ es de tipo PEGI 18, la descripción sobre PEGI 18 que se da en la web oficial de PEGI es la siguiente:

**PEGI 18-** “*La clasificación de adulto se aplica cuando el nivel de violencia alcanza tal grado que se convierte en representación de violencia brutal o incluye elementos de tipos específicos de violencia. La violencia brutal es el concepto más difícil de definir, ya que en muchos casos puede ser muy subjetiva pero, por lo general, puede definirse como la representación de violencia que produce repugnancia en el espectador”*



Figura 30: Clasificación PEGI de PostWar:Hopeless Humanity

#### 5.2.1.5 Resumen del flujo de juego

A continuación se muestra un diagrama de flujo de los diferentes estados por el que puede pasar el juego.

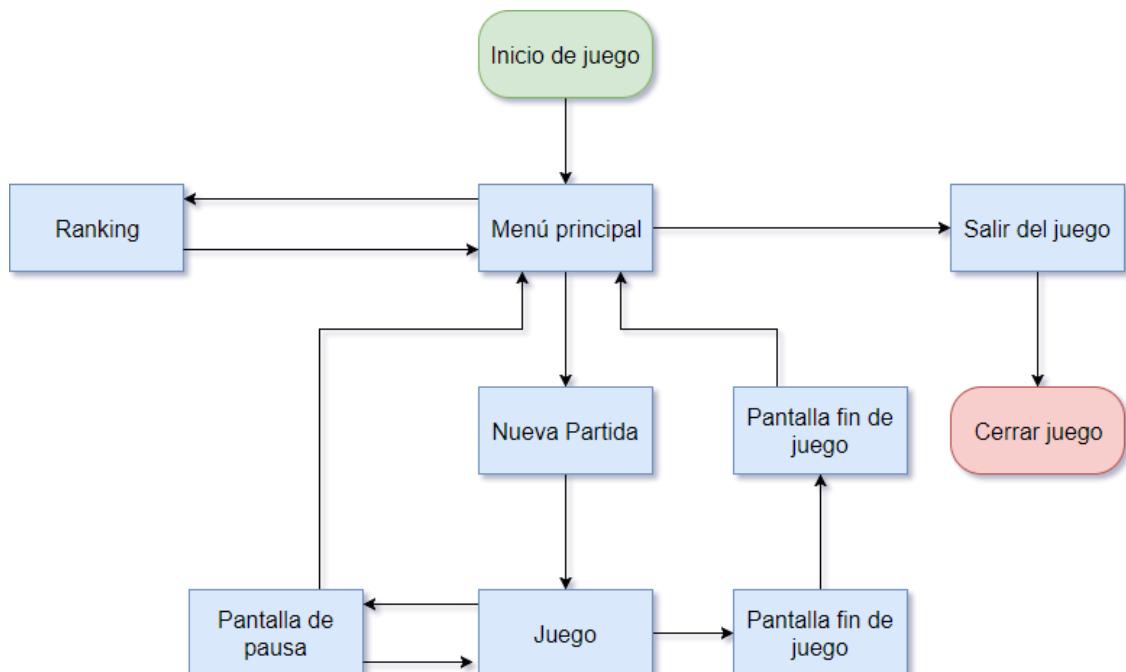


Figura 31: Flujo de pantallas del juego

Este flujo de juego consiste en:

1. Iniciar el juego, el cual se abrirá directamente en el menú principal.
2. Estando en el menú principal tenemos 3 opciones:
  - a. Nueva partida que ejecutara el juego y se podrá empezar a jugar.
  - b. Ranking que permite ver los 3 jugadores que más rondas han aguantado, el número de rondas que sobrevivieron y los zombis que mataron.
  - c. Salir del juego que cerrara la aplicación.
3. Cuando comenzamos una partida se ejecuta el juego. Mientras jugamos podemos acceder al menú de pausa donde disponemos de 2 opciones:
  - a. Reanudar partida que te permitirá seguir la partida en el estado en el que la dejaste.
  - b. Volver al menú principal.
4. Si el jugador muere nos lleva a una pantalla de fin de juego, en la que se mostrarán las rondas sobrevividas y los zombis matados. Además si estamos entre los mejores nos dejará introducir un nombre para guardar nuestra puntuación en el ranking.

### **5.2.1.6 Apariencia del juego**

TODO

### **5.2.1.7 Ámbito**

#### **5.2.1.7.1 Lugares en los que se desarrolla el videojuego**

Te encuentras dentro de una base militar, acabas de salir de tu refugio donde dirigías los bombardeos nucleares, y te encuentras que un ambiente oscuro, silencioso, en mitad de una base militar desierta o eso parece...

#### **5.2.1.7.2 Número de niveles**

Puesto que es un juego de supervivencia solo hay 1 único nivel, el objetivo es aguantar vivo el mayor tiempo posible moviéndote por el escenario y eliminando a los enemigos.

#### **5.2.1.7.3 Número de NPC's**

Los NPC de este juego son infinitos. Estos irán apareciendo una y otra vez mientras continuemos con vida. Al principio no aparecerán muchos a la vez, pero a medida que el juego vaya avanzando se irá complicando, y por lo tanto aparecerán un mayor número de enemigos y serán más resistentes.

Como único enemigo estará el Zombi que te hará daño al contacto.

#### **5.2.1.7.4 Número de habilidades**

Tu eres un alto mando militar, por lo que no tienes una serie de habilidades como son disparar, apuntar y moverte.

Aunque no tengas habilidades especiales en el nivel habrá una serie de mejoras o “**power-ups**” que te permitirán aguantar más golpes, recargar más rápido y tener una vida extra (cuando te maten la primera vez podrás seguir jugando).

### **5.2.2 Jugabilidad y mecánicas**

En este apartado se intentara dar una definición a la jugabilidad, definir los objetivos del juego, la progresión, los controles, mecánicas, sistema de ranking, etc.

#### **5.2.2.1 Jugabilidad**

##### **5.2.2.1.1 ¿Qué es la jugabilidad?**

Aunque no hay una definición exacta de jugabilidad, a lo largo de los años se ha intentado responder a esta pregunta de la manera más acertada posible. Algunas de las definiciones de jugabilidad son las siguientes:

“*El conjunto de propiedades que describen la experiencia del jugador ante un sistema de juego determinado, cuyo principal objetivo es divertir y entretener de forma satisfactoria y creíble ya sea solo o en compañía de otros jugadores*” - Jose Luis González Sánchez, Natalia Padilla Zea y Francisco L. Gutiérrez (del libro *From Usability to Playability*).

“*Las estructuras de interacción del jugador con el sistema del juego y con otros jugadores en el juego*” - Staffan Björk

“*Una serie de decisiones interesantes*” – Sid Meier (game designer).

La Wikipedia nos deja la siguiente definición “*La jugabilidad es un término empleado en el diseño y análisis de juegos que describe la calidad del juego en términos de sus reglas de funcionamiento y de su diseño como juego. Se refiere a todas las experiencias de un jugador durante la interacción con sistemas de juegos. La definición estricta de jugabilidad sería aquello que hace el jugador.*”

Como podemos ver no hay una definición clara de **jugabilidad** pero podemos definirla como la **manera que tiene un jugador de interactuar con el juego**.

Aunque el concepto sea difícil de definir hay una cosa que está clara sobre la jugabilidad y es que **es lo más importante de un juego**. Esto es que es algo en lo que coinciden la mayor parte de los usuarios, un juego que sea muy bueno gráficamente pero con jugabilidad nula no sirve de nada. En cambio, un juego cuya jugabilidad sea muy buena aunque sus gráficos sean pobres puede ser considerado un buen juego. Si el usuario se entretiene y se divierte la jugabilidad habrá cumplido con su objetivo.

### 5.2.2.1.2 Objetivos del juego

Estamos en una base militar arrasada por una bomba nuclear que acabo con la vida tal y como se conocía de lo que quedó expuesta a ella. Somos uno de los supervivientes o quizás el único del ataque nuclear. Nuestro objetivo es entonces nuestra propia supervivencia amenazada por hordas de zombis.

Tenemos la esperanza de que haya algún sobreviviente más que se encontrara en un refugio similar al nuestro, por ello nuestro protagonista lucha por sobrevivir e intentar encontrar algún superviviente.

### 5.2.2.1.3 Progresión

La progresión de NOMBRE DE JUEGO AQUÍ es lineal, se basa por tanto en la habilidad adquirida hasta el momento con el objetivo de que la supervivencia del jugador sea un reto pero que a la vez sea divertido.

De esta forma las primeras rondas serán muy sencillas, muy pocos enemigos y que mueren casi instantáneos, pero a medida que se va avanzando en el tiempo la dificultad va aumentando. Esta dificultad viene marcada por el número de enemigos y su resistencia, cuanto mayor sea el número de rondas mayor será el número de enemigos y más nos costara matarlos.

La progresión por tanto debe ser constante y que no se note un cambio muy brusco de una ronda a la siguiente, sino que el cambio de dificultad entre ronda y ronda sea casi imperceptible pero que se note el aumento de dificultad con el paso del tiempo.

Otra parte importante es la progresión respecto al tamaño del mapa. Al principio el mapa no estará disponible en su totalidad, sino que se jugara en una pequeña parte de este. A medida que vayamos cogiendo experiencia y eliminando enemigos podremos desbloquear distintas partes del mapa.

TODO GRAFICA O TABLA CON LOS VALORES DE RESISTENCIA DE LOS ENEMIGOS EN FUNCION DE LA RONDA, LOS ENEMIGOS TOTALES EN FUNCION DE LA RONDA Y LOS ENEMIGOS QUE PUEDEN APARECER AL MISMO TIEMPO EN FUNCION DE LA RONDA.

#### **5.2.2.1.4 Acciones del personaje**

El control de nuestro personaje es el mismo que el de los juegos FPS, puedes moverte libremente por el escenario, saltar, disparar, recargar, interactuar con algunos objetos como las puertas para abrirlas y desbloquear nuevas partes del escenario o comprar los *power-ups*. Además podrás acercar la cámara para poder apuntar con la mira del arma. Se volverá a hablar de esto más profundamente en el apartado Mecánicas.

#### **5.2.2.1.5 Controles del juego**

TODO

#### **5.2.2.2 Mecánicas**

En este apartado daremos una definición de que son las mecánicas de juego, y cuáles son las diferentes mecánicas de TODO NOMBRE DE JUEGO AQUÍ

Las mecánicas de juego son una serie de reglas que intentan generar juegos que se puedan disfrutar, que generen una cierta adicción y compromiso por parte de los usuarios, al aportarles retos.

La interacción de varias mecánicas de juego en un juego determina la complejidad y el nivel de interacción de un jugador dentro del juego. Esta complejidad e interacción junto con el escenario y los recursos disponibles determinan el equilibrio de juego.

**Por lo tanto la mecánica y la jugabilidad depende una de la otra.** Con una mecánica muy simple se pueden hacer juegos muy divertidos y que enganchen al jugador. Por el contrario con un gran número de mecánicas complejas podemos llegar a acabar con la jugabilidad de un juego.

#### **5.2.2.2.1 Movimiento y cámara en primera persona**

Como cualquier juego FPS y ya que es una característica necesaria para pertenecer a este género la cámara es en primera persona. Esto ayuda a la inmersión del jugador que se siente dentro del juego. La cámara por tanto en primera persona te permite ver el mundo a través de los ojos de protagonista por lo que tu veras únicamente el escenario y el arma con la que se esté jugando.

El movimiento del jugador es lo bastante ligero como para no frustrarte a la hora de jugar ni que de la sensación de lentitud, pero a la vez lo suficientemente lento como para saber que no estas corriendo sino andando y tener la presión de que los zombis pueden alcanzarte.

También podrás correr durante un cierto tiempo limitado, aunque mientras corres no podrás disparar ni recargar.

En cuanto al movimiento será libre en el eje de las X y las Y. Pudiendo dar un salto en el eje de las Z.

La cámara podrá rotar en los 360° en el eje de las Y pero solo 90° en el eje de las X, es decir podrás mirar libremente a tu alrededor girarte completamente y ver tu espalda, pero sin embargo verticalmente no podremos sobre pasar la posición de la cabeza ni de los pies.

#### **TODA FOTO DEL PLAYER EN EL ESCENARIO CON UN ARMA EQUIPADA**

##### **5.2.2.2 Disparar y recargar**

Al igual que el apartado anterior otra característica imprescindible de un juego FPS es la posibilidad de disparar. En este caso el jugador contara con un arma inicial que será la peor de todas. Tendrá por tanto que sobrevivir con esta arma hasta que obtenga los requisitos necesarios para poder avanzar a otra parte del escenario donde se desbloqueara una nueva arma.

Cada arma tendrá unas estadísticas diferentes (cadencia de tiro, tiempo de recarga, daño...).

Para poder recargar un arma el cargador de esta no podrá estar lleno (ya estará cargada) y además tendrás que tener munición suficiente, si te quedan 10 de munición y el cargador es de 30 solo podrás recargar 10 balas. Por lo tanto si te quedas sin munición no podrás recargar y por ende no podrás disparar.

##### **5.2.2.3 Comprar armas**

A medida que vayamos avanzando en el juego podremos comprar nuevas armas. Las nuevas armas estarán disponibles en ciertas partes del escenario que estarán bloqueadas al principio del juego y que serán accesibles a medida que avanzamos en el número de rondas.

#### **TODA FOTO DE UNA PARED DONDE PUEDES COMPRAR UN ARMA**

Cuando compramos un arma esta se añade a “nuestro armamento” en el que podremos tener un máximo de 2 armas.

Si cuando vamos a comprar un arma ya tenemos dos, el arma que tengamos equipada en ese momento se sustituirá por el arma comprada, perdiendo entonces el arma equipada y obteniendo como nuevo arma la comprada.

Al comprar un arma obtenemos toda su munición.

Las armas se compraran mediante los puntos que ganamos al eliminar enemigos. Si no tenemos puntos suficientes para pagar un arma no podremos comprarla.

#### **5.2.2.2.4 Mejorar armas**

Para mejorar las armas habrá una especie de máquina en la última sala del escenario en la que pagaras puntos por mejorar el arma equipada en el momento en el que se interactúa con la máquina.

Se permite mejorar distintos aspectos del arma (hasta un máximo establecido):

- **Capacidad del cargador:** se aumentara el número de balas que contiene el cargador equipado del arma pero no la munición máxima de esta. Por ejemplo si la munición máxima de un arma es de 200 balas y su cargador es de 10 balas cuando se mejora la capacidad del cargador la munición máxima del arma seguirá siendo 200 balas pero el tamaño del cargador será de 20 balas.
- **Munición máxima:** se aumentara el número de munición máxima que se obtiene al comprar munición pero no la capacidad máxima del cargador.
- **Daño:** aumenta el daño que hace el arma.
- **Tiempo de recarga:** disminuye el tiempo de recarga.
- **Cadencia de disparo:** disminuye el tiempo entre disparo y disparo.

#### **5.2.2.2.5 Comprar munición**

La munición se comprará en el mismo sitio donde se compra el arma, con la única diferencia de que si ya tienes el arma únicamente repondrás munición en vez de comprar el arma de nuevo a un coste mucho menor.

Cuanto más mejorada este un arma más cara será su munición.

**TODA FOTO DE QUE SI YA TIENES EL ARMA SOLO COMPRAS MUNICION**

### 5.2.2.2.6 Power-ups

Los *power-ups* también conocidos como poderes especiales, son objetos que al instante benefician o añaden capacidades adicionales para el personaje del juego, como una mecánica de juego.

TODO FOTO DE LOS POWER-UPS (LAS MAQUINAS EN LAS QUE LOS COMPRAS)

#### 5.2.2.2.6.1 Resistencia

Se podrá comprar la mejora de aumentar tu resistencia a los golpes de los enemigos para poder aguantar más.

Un aspecto importante es que solo podrás comprar 1, es decir no es acumulable, por lo que comprar 2 no te hace doblemente resistente.

#### 5.2.2.2.6.2 Recarga rápida

Con esta mejora podrás recargar más rápido como su propio nombre indica, por lo tanto se disminuirá notablemente el tiempo de recarga de las armas. Este *power-up* combinado con unas buenas mejoras en el tiempo de recarga del arma puede ayudarte mucho a acabar con grandes hordas de zombis en los que no te da tiempo casi ni a recargar.

Al igual que la resistencia este potenciador es único, es decir, no es acumulable pero sí que puede **combinar con otro *power-up*** como el de la resistencia.

#### 5.2.2.2.6.2 Revivir

Estará disponible un único *power-up* de revivir. Cuando lo compramos tendremos una vida extra, es decir si nos matan los enemigos reviviremos de nuevo una única vez.

Cuando revivimos perderemos todo lo que teníamos menos el arma equipada en ese momento. Por lo que tendremos que volver a comprar un arma secundaria, y los potenciadores de recarga rápida y resistencia.

Al igual que los anteriores este potenciador es combinable con el de resistencia y recarga rápida.

### 5.2.2.2.7 Abrir puertas

El jugador tendrá la posibilidad de abrir puertas para ir ampliando el escenario y poder tener acceso a diferentes armas y mejoras.

Las puertas tendrán un coste de puntos por lo que antes de abrir una puerta necesitaras eliminar a una cantidad de enemigos suficiente para obtener los puntos que costara abrir la puerta.

TODO FOTO DE CUANDO NO PUEDES ABRIR LA PUERTA AUN

TODO FOTO DE LA PUERTA YA ABIERTA

### **5.2.2.3 Rejugar y salvar**

Por el tipo de juego y el objetivo del mismo no se podrá guardar la partida a mitad ya que esto rompería un poco el estilo de supervivencia que se quiere crear. Por lo tanto si te sales a mitad de una partida perderás todo el progreso y cuando vuelvas a iniciar el juego tendrás que volver a empezar desde la primera ronda.

Lo que se pretende con esto es no perder ese clima de tensión en el que se encuentra el jugador después de estar sobreviviendo el mayor número de rondas posibles, ya que si guardas partidas y la retomas al día siguiente por ejemplo hemos perdido toda la inmersión del jugador así como todo lo que le costó llegar hasta el punto en el que se encuentra en ese momento. Mientras que si no permitimos guardar la partida el jugador sabe que tiene que sobrevivir el máximo tiempo posible sin vuelta atrás y sin descansos (si hay descansos el menú de pausa pero no son descansos de dejar la partida y retomarla en otro momento).

### **5.2.2.4 Ranking**

El juego cuenta con un sistema de ranking en el que se podrá ver cuáles son los 3 mejores jugadores. En él se podrá ver el nombre del jugador, los zombis que mató y lo más importante hasta que ronda sobrevivió.

Al acabar una partida se comprobará si tu puntuación es suficiente para entrar en el ranking, de ser así saldrá una pantalla en la que podrás introducir tu nombre.

TODO FOTO DE RANKING

## **5.2.3 Historia y personajes**

### 5.2.3.1 Historia

Es el año 2020, año en el que parece que la diplomacia deja de existir, el mundo queda dividido en 2 partes los que están bajo el mando de Corea del Norte y los que se oponen a su dominio.

Esta división surge a raíz del intento desesperado y continuo de Corea del Norte por expandir su poder y su régimen en los territorios vecinos. Las relaciones internacionales ya deterioradas no sirven de nada, no hay posibilidad de negociación y el temor empieza a brotar en la población que se preparan para lo peor. En este clima de crispación, temor y desesperación los países empiezan su armamento nuclear.

Ambas partes están preparadas para la guerra, las pruebas de misiles de largo alcance son continuas cada día. “Un fallo” en la prueba de un misil nuclear hace que acabe estrellándose contra un territorio en contra del régimen coreano y destruyendo así todo atisbo de vida. Ante este suceso se decide responder con otro ataque lo que desencadena lo que podemos llamar la 3era Guerra Mundial.

Esta nueva guerra muy diferente al resto puesto que ya no son los soldados los que tienen el protagonismo sino los misiles nucleares ahora los que se encargan de arrasar con todos los territorios enemigos.

Estos misiles son controlados desde diferentes partes del mundo, desde unos refugios nucleares creados exclusivamente para este fin, para resistir a las bombas enemigas al mismo tiempo que se pueden seguir lanzando ataques aliados.

El mundo dividido ahora en 2 regiones se ve envuelto en una gran guerra nuclear, el planeta entero es víctima de distintos ataques nucleares de ambos bandos, cuando estos cesan se descubre la gravedad del asunto. La existencia de la humanidad está en peligro.

Nosotros somos un alto mando militar superviviente gracias a que estábamos en uno de estos refugios nucleares dentro de una base militar desde donde dirigíamos los ataques. Al salir del refugio vemos la gravedad de la situación y es que sin presencia de vida humana nuestro objetivo ahora es la lucha por nuestra supervivencia, contra lo que antes eran personas, pero que ahora debido a la radiación tan elevada se han convertido en muertos vivientes.

Tenemos la esperanza de encontrar más supervivientes aliados que estuvieran en otros refugios dirigiendo el resto de ataques, pero no podremos averiguar esto si acaban con nosotros lo zombis.

Nuestro objetivo entonces está bien claro ¡SOBREVIVIR!

### **5.2.3.2 Personajes**

#### **5.2.3.2.1 Protagonista (TODO ponerle nombre)**

Nosotros ocupamos un alto cargo del ejército y dada nuestra elevada experiencia en tecnología nos asignan la tarea de dirigir los misiles nucleares de nuestro territorio.

Cuando salimos del refugio nos damos cuenta que la guerra ya no importa, que lo importante ahora es luchar por sobrevivir.

#### **5.2.3.2.2 Zombis**

Los zombis son los enemigos que tendremos que ir eliminando para poder avanzar en el juego. No tienen un perfil específico ya que toda persona alcanzada por la bomba nuclear se convirtió en zombi, tanto amigos, como compañeros del ejército o enemigos.

### **5.2.4 Nivel de juego**

TODO NOMBRE DE JUEGO AQUÍ solo tiene un nivel puesto que es un juego de supervivencia tendrás que sobrevivir en este nivel el mayor tiempo posible. Aunque solo tenga un nivel este no es accesible en su totalidad desde el principio. Esto se detallara mejor en el apartado escenario.

#### **5.2.4.1 Inteligencia Artificial**

La inteligencia artificial de TODO NOMBRE DE JUEGO AQUÍ es más bien sencilla, ya que este tipo de juegos no necesita una IA (Inteligencia Artificial) muy compleja para ser divertidos. La diversión recae en el aumento de dificultad derivado del aumento de enemigos y su resistencia haciendo cada vez más difícil que sigamos con vida.

Por lo tanto la IA se resume en perseguir al jugador y atacarle cuerpo a cuerpo cuando estén cerca de nosotros. Lo que sí que habrá es un buen diseño en cuanto al número de enemigos que puedan haber al mismo tiempo para hacer el juego difícil pero no imposible, además de los puntos de *respawn* (puntos donde aparecen los enemigos) que estarán repartidos por diferentes partes del escenario, pero teniendo en cuenta que si una parte del escenario está cerrada no podrán aparecer enemigos en puntos de *respawn* que estén en ella, pero de esto se habla más profundamente en el apartado de “Puntos de respawn”.

### 5.2.4.2 Escenario

Como se ha dicho al principio de este apartado el nivel es único, y se trata de una base militar bastante moderna aunque estropeada por la guerra.

Nos encontramos entonces un ambiente conocido aunque totalmente cambiado, ya que ahora está desierto o al menos de vida humana, la luz es débil y titila debido a las bajadas y subida de tensión. Las habitaciones son muy variadas, desde habitaciones muy amplias que te harán sentir inseguro por la amplitud de la misma hasta estrechos pasillos que te generaran una gran tensión al sentirte rodeado y sin escapatoria.

A todo esto hay que sumarle también el hecho de que hay evidencias claras de que la base militar ha sido usada por personas recientemente, hecho que junto con todo lo anterior provocan al jugador un sentimiento de tensión e inseguridad que le permitirá verse inmerso en el juego.

TODO FOTOS DEL ESCENARIO

### 5.2.4.3 Música y sonidos

La música y los sonidos es una parte muy importante en cualquier videojuego pues un buen diseño sonoro es capaz de conseguir una inmersión total del jugador y hacerle sentirse dentro del propio juego gracias a los sonidos envolventes o a los sonidos desde un determinado punto que se van atenuando con la distancia.

Aunque no es la parte más importante en el juego siempre hay que ser cuidadosos con los sonidos ya que gracias a estos se puede influir en las sensaciones del jugador, crearle momentos de tensión o por el contrario momentos de relajación donde te sientes seguro y fuera de peligro con una música suave por ejemplo.

De esta manera en TODO NOMBRE DE JUEGO AQUÍ habrá varios tipos de sonido:

- **Sonidos y música ambiental:** serán los sonidos o música que se escuchará de fondo durante todo el juego cuyo objetivo es crear una atmósfera que posicione al jugador dentro del propio juego.
- **Efecto de sonido (SFX):** son aquellos sonidos que se reproducen en un sitio concreto y que son el resultado de alguna acción, por ejemplo un disparo, abrir una puerta, mejorar un arma, etc.
- **UI Sounds** (sonidos en la interfaz de usuario): son aquellos sonidos que se reproducen en las pantallas de menú, como por ejemplo al pulsar un botón.

#### **5.2.4.4 Puntos de respawn**

Entendemos por punto de *respawn* aquellos lugares del escenario en el que puede *respawnear* (aparecer) un enemigo.

Los puntos de *respawn* como tal no es una decisión del todo importante, lo importante es el uso que se hace de ellos y las limitaciones que se les pone (tiempo entre *respawn* y *respawn*, aleatoriedad, etc.).

Los puntos de *respawn* estarán repartidos por todo el escenario, lo cual supone ya un problema de inicio y es que el escenario al comenzar el juego no está disponible en su totalidad por lo que solo jugaremos en una pequeña parte de este.

Debido a esto los puntos de *respawn* podrán estar activos o inactivos. Un punto de *respawn* activo significa que en él puede aparecer un enemigo en cualquier momento del juego, por el contrario uno inactivo indicara que en ese punto no podrá reaparecer ningún enemigo.

A medida que vamos desbloqueando nuevas partes del escenario también se van activando los puntos de *respawn* de esas partes del escenario.

El sitio donde aparece un enemigo es aleatorio, elegirá un punto de *respawn* al azar entre todos los activos en ese momento.

La velocidad con la que aparecen los enemigos dependerá de la ronda en la que nos encontremos, cuanto mayor sea el número de ronda mayor será la dificultad y por lo tanto menor el tiempo de reaparición entre enemigo y enemigo.

La cantidad de los enemigos que aparecen por tanto depende de la ronda, al empezar la ronda aparecerán de golpe una oleada de enemigos cuya cantidad corresponde al resultado de una formula en la que el factor importante es el número de rondas.

**TODO PONER FORMULA DE CANTIDAD DE ZOMBIES QUE APARECEN**

Una vez que han aparecido la cantidad inicial de zombis, aparecerán nuevos zombis en el mapa cada cierto tiempo. Este tiempo también depende de un factor fijo y la ronda en la que nos encontramos.

## TODO FORMULA DE RESPAWN DE ZOMBIES SEGÚN EL TIEMPO

La idea entonces es tener una primera horda de zombis en cada ronda y mientras estas intentando sobrevivir a ella que vayan apareciendo zombis continuamente con un intervalo fijo de tiempo. Tanto el tiempo de reaparición como la cantidad dependerán de la ronda.

De esta manera tendremos por ejemplo que en la ronda 10 el número total de zombis será de 120, de los cuales 30 aparecerán al principio de la ronda y los otros 90 aparecerán cada 0.5 segundos en un punto de *respawn*.

Si ya han aparecido el número máximo de zombis de esa ronda estos dejaran de reaparecer, de esta manera cuando los mates ya no aparecerá ninguno más y será el paso a la siguiente ronda.

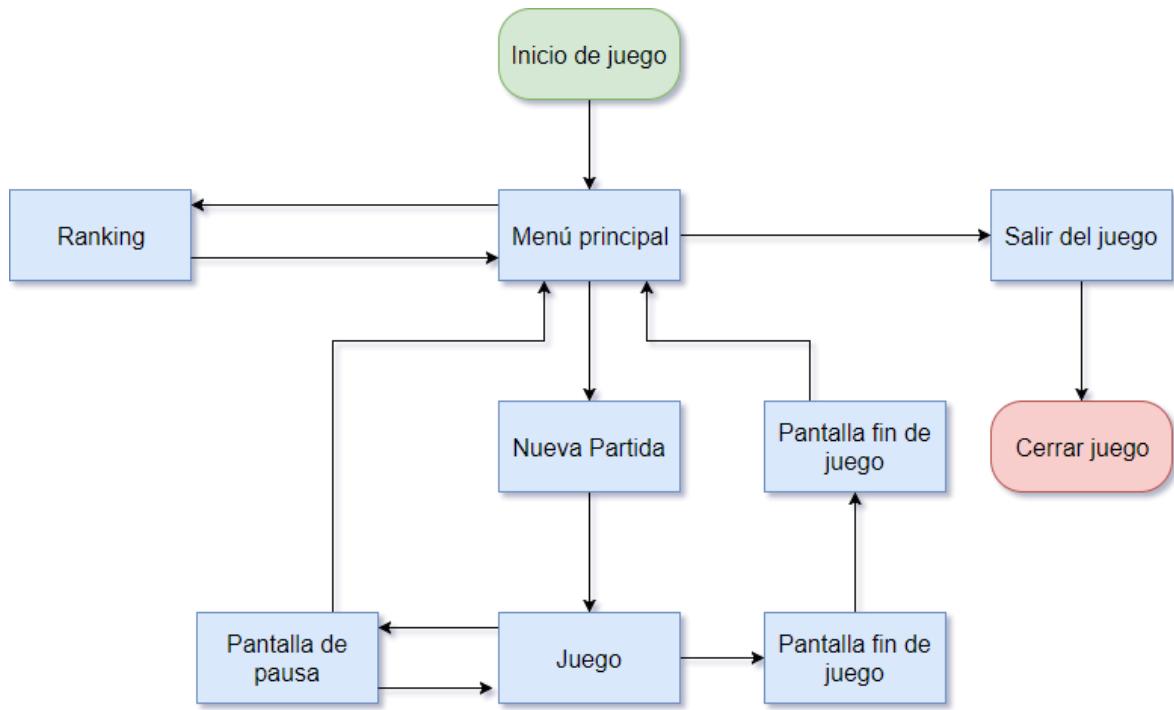
### 5.2.5 Interfaz

Una interfaz de usuario (UI) es el medio con el que el usuario puede comunicarse con un equipo (o el juego en este caso) y comprende los puntos de contacto entre el usuario y el equipo. Es decir, es la manera en la que podemos comunicarnos con el juego y la forma en la que el juego se comunica con nosotros, por ejemplo, mostrándonos información mediante el HUD del que hablaremos más adelante.

#### 5.2.5.1 Menús

Los menús nos permitirán navegar a través del juego, pudiendo por tanto modificar algunas opciones como la resolución, acceder al ranking o los créditos, empezar una nueva partida o salir del juego.

Para poder explicar los menús de una forma más clara se va a recuperar la foto del apartado 5.2.1.5 Resumen del flujo de juego:



### 5.2.5.1.1 Menú principal



Figura 32: Menú del juego en Photoshop

Este es el menú principal, es lo primero que se ve cuando se ejecuta el juego, en él se podrá:

- Empezar una partida
- Acceder al menú de opciones
- Acceder al ranking
- Acceder a los créditos
- Cerrar la aplicación.

### 5.2.5.1.2 Menú de pausa

Este menú aparecerá cuando el jugador pause el juego. En el podremos continuar con la partida tal cual la habíamos dejado o volver al menú principal.



Figura 33: Menú de pausa en juego

### 5.2.5.1.3 Menú fin de partida

Hay dos tipos de menú de final de partida, uno es cuando mueres con puntuación suficiente como para entrar en el ranking, mientras que el otro es cuando mueres y no has llegado a la puntuación necesaria para entrar en el ranking.

#### 5.2.5.1.3.1 Sin Ranking

En esta pantalla aparecerá el mensaje de que has muerto y un botón para poder volver al menú principal.



Figura 34: Pantalla muerte sin ranking

#### 5.2.5.1.3.2 Con Ranking

En esta pantalla aparecerá un cuadro donde nos dejara escribir nuestro nombre que es el que luego saldrá en el ranking, un botón de aceptar que validara la información y nos llevara de vuelta al menú principal.

Además también se nos muestra información de la partida como el número de muertes y la ronda a la que hemos llegado que son los datos que aparecerán luego en el ranking junto al nombre que hemos introducido.

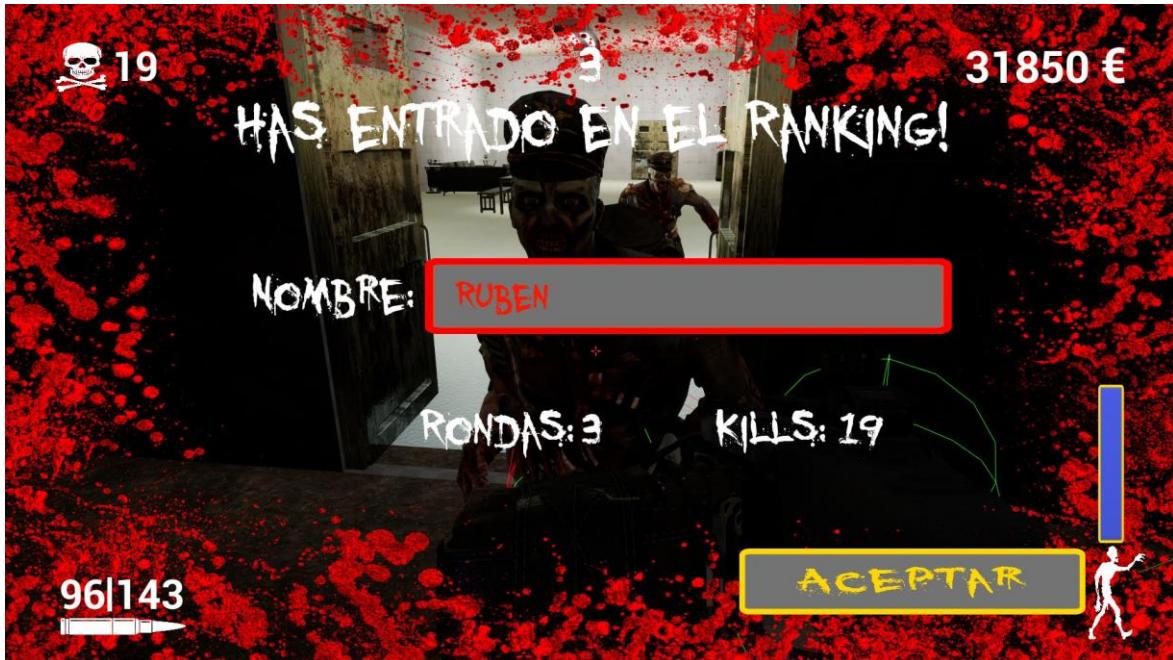


Figura 35: Pantalla muerte con ranking

### 5.2.5.1.4 Pantalla de ranking

En esta pantalla aparecerán los 3 jugadores que hayan obtenido mejor puntuación. Esta información se guarda entre partidas, manteniéndose intacta aunque se cierre el juego.

Aparecerá el nombre, el número de rondas al que has conseguido llegar y el número de zombis que has matado. Además de un botón de “exit” con el que regresas al menú principal.

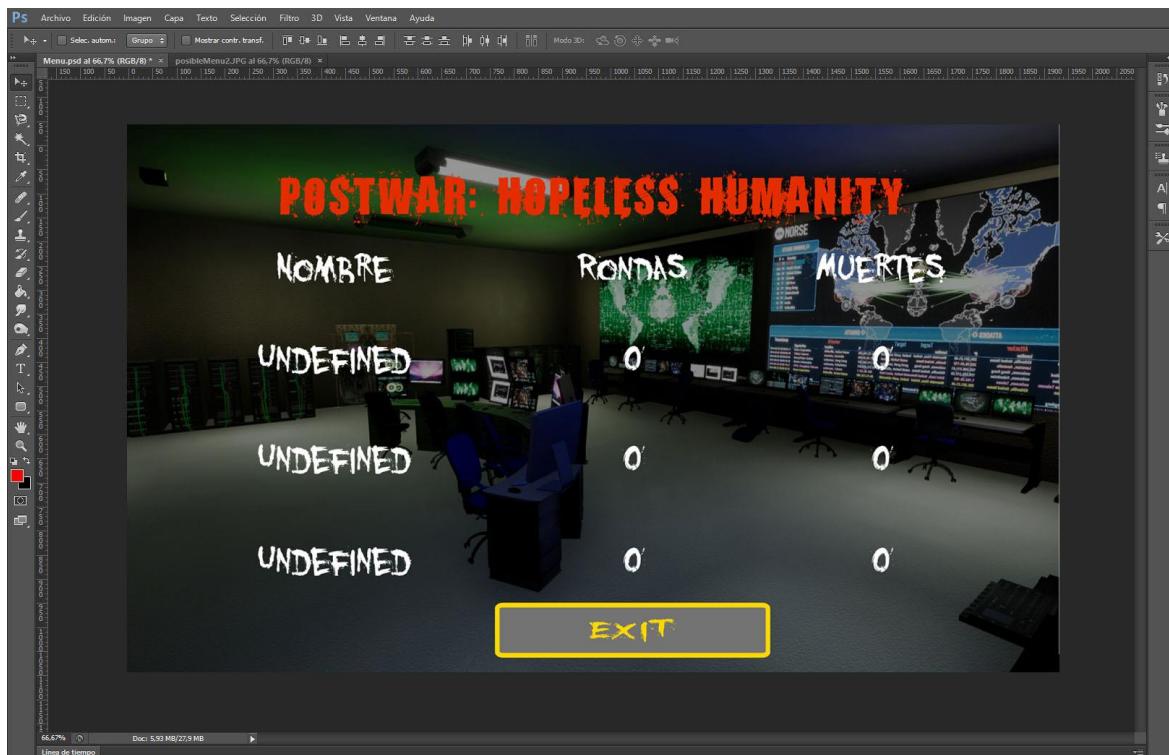


Figura 36: Menú ranking en Photoshop

### 5.2.5.1.5 Menú de opciones

En el menú de opciones podrás ajustar la resolución del juego.

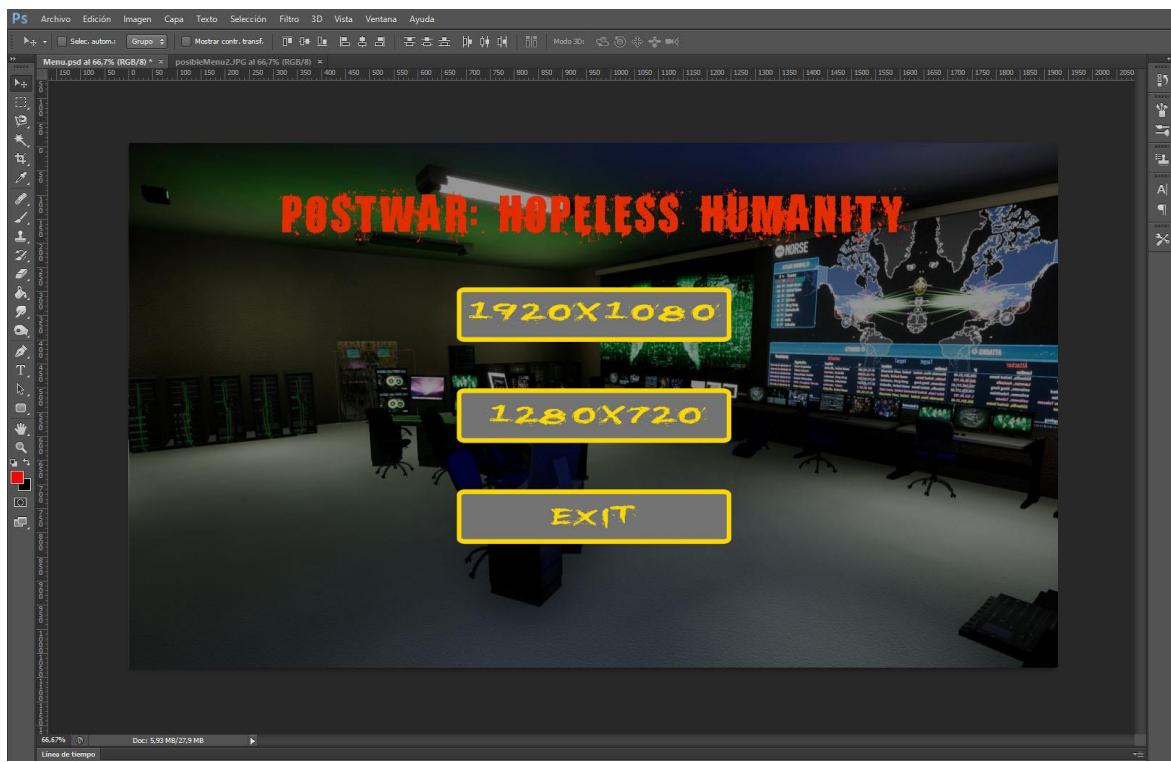


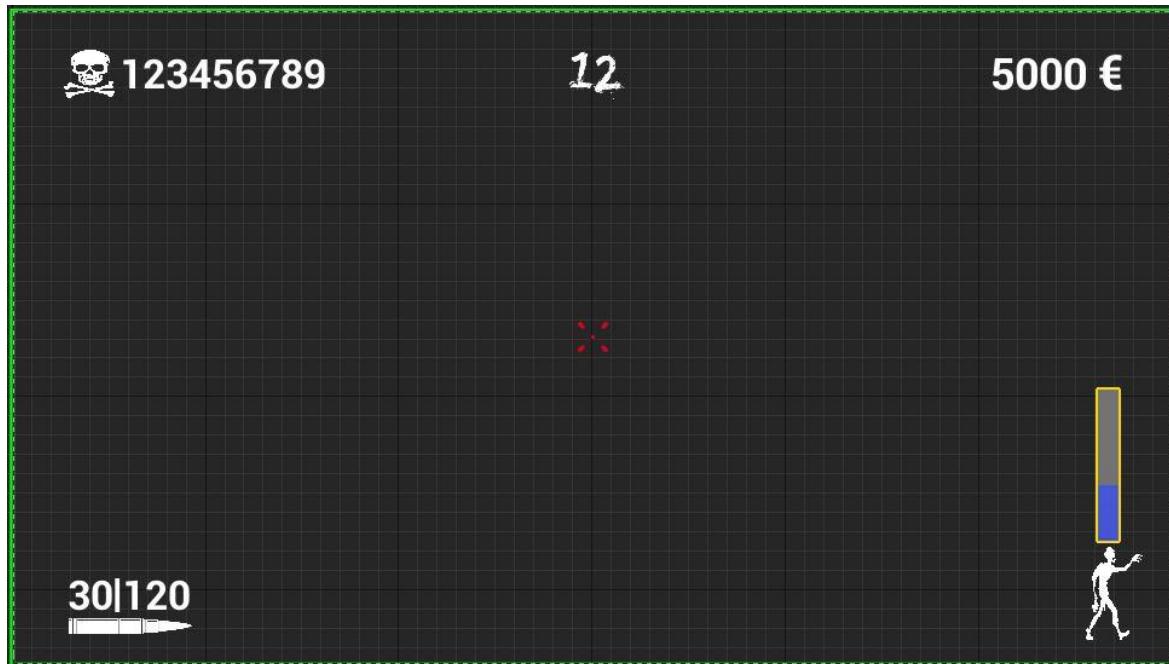
Figura 37: Menú opciones en Photoshop

### 5.2.5.2 HUD

El HUD (*heads-up-display*) es información útil sobre el juego que se le muestra al jugador dibujándola en 2D sobre la pantalla.

En el HUD de Postwar se puede diferenciar en 2 partes.

- Información que se muestra en todo momento (**Figura 38**):
  - Zombis matados (arriba izquierda)
  - Ronda actual (arriba en el centro)
  - Dinero actual (arriba derecha)
  - Munición del cargador y munición total (abajo izquierda)
  - Cantidad de zombis restantes mediante una barra de progreso (abajo derecha)
  - Mirilla (en el centro de la pantalla)



*Figura 38: HUD información constante*

- Información que aparece por algún evento concreto (**Figura 39**):
  - Sangre alrededor de la pantalla, va apareciendo dependiendo de nuestra vida restante, su intensidad es inversamente proporcional a nuestra vida restante, es decir, con 100% de vida tendrá 0 de opacidad (no se ve), mientras que con 20% de vida tendrá 0.8 de opacidad (será casi opaco).
  - Mensaje de recarga, aparecerá cuando se esté recargando el arma.
  - Mensaje de reviviendo aparecerá cuando nos estemos reviviendo.
  - Cambio en la puntuación (arriba derecha) aparecerá cuando hay una variación en la puntuación pudiéndose positiva (al matar algún enemigo o por impacto en la cabeza) siendo esta de color verde. O bien, podrá ser negativa apareciendo de color rojo, esto se dará cuando comprarnos algún arma, alguna mejora, munición, abrimos una puerta, etc.
  - Información sobre objeto interactuable (en este caso “M4a1 300€”). Este tipo de mensaje se verá cuando estamos cerca de algún objeto con el que podemos interactuar, entonces te mostrara la información del objeto y el precio de este, por ejemplo si nos acercamos a una puerta pondría “Puerta 900€”.



Figura 39: HUD información dinámica

### 5.2.5.1 Cámara

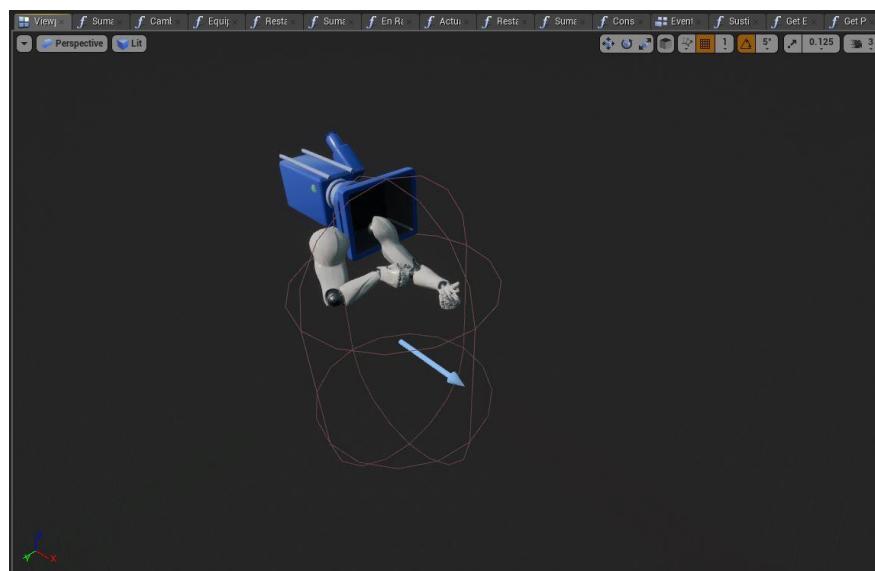


Figura 40: Camara FPS

La cámara de Postwar: Hopeless Humanity es una cámara en primera persona, por lo tanto no tendremos un personaje como tal, sino que solo tendremos los brazos que es lo que se va a ver y una cámara situada en lo que sería la cabeza del personaje, pudiendo simular así que se está viendo el juego desde el punto de vista que la vería el propio jugador.

Como elementos destacan la propia cámara a partir de la cual podremos ver el juego, una capsula de colisión que simula el tamaño del cuerpo del personaje, de manera que podamos colisionar contra obstáculos u enemigos de la misma forma que lo haría un cuerpo.

Los brazos que aunque ahora no sostienen nada, en el juego se encargarían de sujetar las armas cuando las compramos, para asegurarnos que estas están en el lugar correcto se han añadido unos sockets al esqueleto de los brazos, de manera que cuando compremos una arma esta se pondrá en la posición de ese socket y seguirá su movimiento. De esta manera si el brazo se mueve el arma también se mueve con él.

## 5.3 Desarrollo e implementación

### 5.3.1 Entendiendo Unreal Engine 4 y su sistema de scripting visual Blueprints

#### 5.3.1.1 La arquitectura de Unreal Engine 4

En este apartado se hablará sobre la arquitectura de Unreal, que es y para qué se usa, se verán conceptos básicos como qué es un *Pawn* o un *Actor* dentro de Unreal, y otros no tan básicos como los distintos tipos de luces o cómo funcionan los materiales de UE4.

##### 5.3.1.1.1 Conceptos base de la arquitectura de Unreal Engine 4

###### 5.3.1.1.1.1 UObjects y Actors

Tanto los actores como los objetos heredan de la clase UObject, que es la clase base para todos los objetos de Unreal Engine incluyendo por tanto a los actores. Aunque los dos hereden de UObject, los objetos son instancias de clases que heredan de UObject mientras que los actores son instancias de clases que derivan de la clase AActor (que esta a su vez deriva de UObject).

AActor es la clase base para todos los objetos que pueden ser posicionados en el mundo de juego, por tanto de aquí sacamos la diferencia fundamental entre los UObjects y los Actors y es que cualquier actor puede posicionarse en el mundo y tiene transformaciones de posición, escala y rotación, mientras que un Objeto no puede colocarse en el mundo.

La otra diferencia es que los actores están preparados para la réplica en un juego online (**networking support**).

De esta forma los actores pueden ser pensados como entidades completas, mientras que los objetos son partes más especializadas que sirven por ejemplo para definir un actor, su funcionalidad, etc.

Los objetos son más pequeños en memoria que un actor.

###### 5.3.1.1.1.2 Gameplay y las clases básicas

Las clases de gameplay básicas incluyen las funcionalidades necesarias para poder representar jugadores en el mundo tanto aliados como enemigos, además de poder dotar a estos jugadores de

un sistema que lo controle. Este sistema de control puede ser o la entrada del propio jugador o por ejemplo una inteligencia artificial.

También hay otras clases básicas que permiten la creación de HUDs (heads-up display) y cámaras para jugadores.

Por ultimo existen clases como GameMode, GameState y PlayerState que establecen las reglas del juego y permiten saber el estado en el que se encuentra el juego y la progresión de los jugadores sobre este.

### 5.3.1.1.3 Representacion y control de jugadores

#### 5.3.1.1.3.1 Representando jugadores en el mundo

Cuando queremos introducir actores que puedan ser controlados por cualquier jugador o por la IA entonces estamos hablando de *pawn* y si es humanoide de *character*.

##### 5.3.1.1.3.1.1 Pawn

Un pawn es un actor que está preparado para aceptar input de forma fácil, pueden ser poseídas por un *Controller* y realizar acciones propias de cada uno. Un pawn no tiene por qué ser humanoide.

##### 5.3.1.1.3.1.2 Character

Un *Character* es un *pawn* de estilo humanoide. En el caso de este juego usaríamos el **FirstPersonCharacter** al ser un FPS (juego de disparos en primera persona), pero tambien se podría usar un **ThirdPersonCharacter** para juegos en tercera persona.

Los *character* vienen con una capsula de colisión integrada que sería la encargada de simular la colisión de un cuerpo humanoide. Además vienen con un componente de movimiento por defecto el cual nos permite hacer acciones básicas como movernos y saltar desde el principio.

Un *character* también viene preparado para replicar su movimiento online y para que pueda reproducir animaciones de manera sencilla.

#### 5.3.1.1.3.2 Formas de controlar a un jugador del mundo

##### 5.3.1.1.3.2.1 Controller

Un *controller* es un actor no físico (no se puede representar) que puede poseer un *pawn* y por tanto se encarga de controlar sus acciones. Hay 2 tipos de *controller* el **PlayerController** y **AIController**.

#### 5.3.1.1.3.2.2 PlayerController

El PlayerController es el tipo de *controller* que es usado por la persona que está jugando, por tanto es la persona la que a través del PlayerController controla las acciones del *pawn* al que posee.

#### 5.3.1.1.3.2.3 AIController

El AIController es el otro tipo de controlador, si bien el PlayerController era controlado por la persona que está jugando, el AIController es controlado por la máquina, es decir es la Inteligencia Artificial la encargada de controlar las acciones del *pawn* al que posee este AIController.

#### 5.3.1.1.4 Mostrando información a los jugadores

##### 5.3.1.1.4.1 HUD

El HUD (head-up display) es una pantalla transparente que presenta información al usuario de tal forma que este no debe de cambiar su punto de vista para ver dicha información. Por lo tanto el HUD como ya dijimos en el apartado HUD del GDD es una interfaz 2D que se muestra encima de la pantalla y que da al jugador información importante sobre el estado del juego.

Los HUD típicos muestran información sobre vida, puntuación, munición, la mirilla con la que apuntas, etc.

##### 5.3.1.1.4.2 Cámara

La cámara sería lo que en la vida real se correspondería con nuestros ojos, cada PlayerController tiene una cámara.

Las cámaras son gestionadas mediante un **PlayerCameraManager**.

#### 5.3.1.1.5 Manejando información del juego

La información que se maneja sobre un juego, sus reglas, su progreso, etc se dividen en 3, que son el GameMode, GameState y PlayerState.

Aunque este dividido en 3 partes todas trabajan conjuntamente, de manera que un GameMode incluye un GameState y a su vez este tiene un array de PlayerState.

#### 5.3.1.1.1.5.1 GameMode

El GameMode existe solo en el servidor y se encarga de controlar como los jugadores se unen al juego o los *spawn* de los jugadores. Desde el GameMode también se puede parar el acceso al juego de un jugador que se está conectando o saber cuándo un jugador se ha desconectado.

Además se encarga de controlar las reglas del juego y las condiciones de victoria (sería como una especie de árbitro).

#### 5.3.1.1.1.5.2 GameState

El GameState existe tanto en el servidor como en todos los clientes, se encarga de controlar los aspectos comunes a todos los jugadores. Por ejemplo en un mundo se encargaría de asegurar que todos los objetos están en las mismas posiciones para todos los clientes, o que por ejemplo si en un juego alguien abre una puerta el GameState se encarga de que esta esté abierta en el resto de clientes.

El GameState contiene cosas comunes a todos como la lista de jugadores, la puntuación total del equipo, las misiones que se hacen en un mundo abierto, etc.

#### 5.3.1.1.1.5.3 PlayerState

El PlayerState sería individual de cada jugador, y por lo tanto se encarga de la información propia del jugador como puede ser el número de muertes, munición, etc en un juego de disparos. Es decir, un PlayerState guarda la información necesaria para un cliente pero que no necesita saber el servidor y por lo tanto no es necesario que el servidor gaste ciclos en calcular esa información.

Cada jugador tiene un PlayerState y estos pueden replicarse en la red para que todos los clientes estén sincronizados.

#### 5.3.1.1.6 Como se relacionan estas clases

El siguiente grafico sacado de la documentación oficial de Unreal Engine 4 nos muestra cómo se relacionan las clases explicadas anteriormente.

Un juego está compuesto por un GameMode y un GameState. Los jugadores que se unen al juego se representan en este como un PlayerController. El PlayerController contiene cosas básicas para una cámara para poder ver el mundo, un HUD que nos aporta la información del juego y la posibilidad de recibir input para poder manejar al personaje. Además el PlayerController permite poseer un pawn para que puedan ser representados físicamente en el juego.

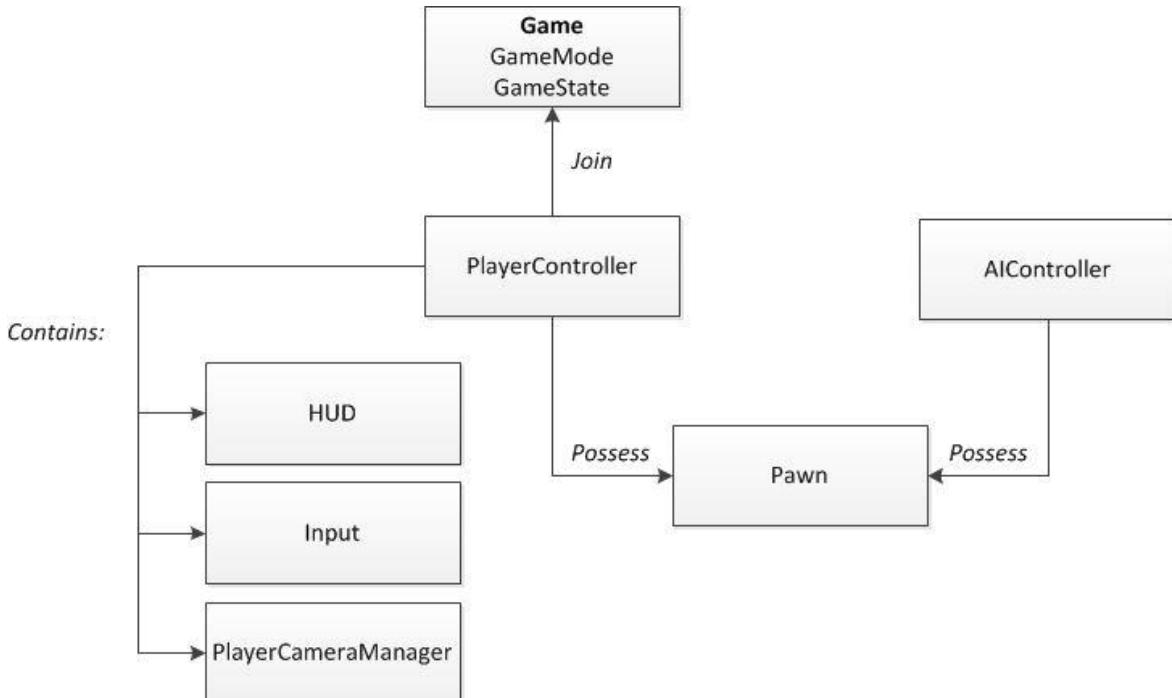


Figura 41: Relacion entre clases

#### 5.3.1.1.2 Scripting visual Blueprints

El Scripting visual Blueprint de Unreal Engine 4 es un sistema visual de programación basado en el concepto de usar nodos y la unión entre estos para crear elementos de gameplay. Como en otros lenguajes de programación se utiliza para definir clases orientadas a objetos (OO).

Por lo general a la mayoría de elementos clases creadas mediante el sistema visual de Blueprints les llamaremos “Blueprint”. Así un Blueprint será entonces una clase dentro del juego como puede ser por ejemplo un arma y tendrá toda la lógica de esta como una función para disparar, una función de recargar, etc.

Este sistema es muy flexible y poderoso ya que permite a los diseñadores utilizar gran cantidad de herramientas y conceptos usados antes únicamente por programadores.

A través del uso de Blueprint los diseñadores pueden crear cualquier elemento del gameplay como por ejemplo las reglas del juego, modificar y crear mallas, crear o modificar materiales, crear cámaras, modificar cámaras, cambiar los controles, añadir nuevos inputs, etc.

#### 5.3.1.1.2.1 ¿Qué es un Blueprint?

Los Blueprints son assets especiales que proporcionan una interfaz basada en nodos intuitiva que mediante la conexión de nodos, eventos, funciones y variables mediante líneas son capaces de crear nuevos tipos de actores, realizar scripting de eventos de nivel, etc. Permitiendo así la creación de un prototipo sin la necesidad de escribir código y desde el propio editor de Unreal Engine.

Una parte importante es que los Blueprints también permiten herencia al igual que se haría en código. De esta manera cuando un **Blueprint padre** es heredado por los **Blueprints hijos** estos heredarían también todas las funciones y variables del padre. Las funciones heredadas pueden ser sobrescritas o no desde los hijos, es decir, si se crea una función en el hijo con el mismo nombre que la función del padre esta será sobrescrita y cuando se llame a esa función se ejecutara la lógica del hijo, pero por el contrario si no se sobrescribe se ejecutara la lógica que había en la función del padre.

#### 5.3.1.1.2.2 ¿Qué contiene un Blueprint?

Los Blueprints ya vienen con algunos componentes por defecto, mientras que otros pueden ser añadidos en función de nuestras necesidades.

El poder añadir componentes a nuestro Blueprint permite que estos puedan ser reutilizados para poder agilizar así el level design. Por ejemplo si tenemos varios objetos con los que podremos interactuar podemos crear un Blueprint padre que contenga una BoxCollision y a partir de este crear los Blueprints hijos que serán los objetos interactivos, de esta forma la lógica de interactuar estará únicamente en la clase padre y se daría al entrar en la BoxCollision. Luego cada Blueprint hijo podría tener su función propia de interactuar.

##### 5.3.1.1.2.2.1 Componentes (Components Window)

Antes de empezar con los componentes hay que tener claro que son. Un componente es una pequeña parte funcional o una pieza con funcionalidad que puede ser añadida a un actor. Un componente no puede existir por sí solo pero cuando se añade a un actor este tendrá acceso al componente pudiendo modificar la funcionalidad de este.

Por ejemplo un Spot Light Component permitirá a tu actor emitir luz como si fuera un Spot Light (luz tipo linterna).

Una vez que ya tenemos claro lo que es un componente ya podemos definir la ventana de componentes (Components Window).

Esta ventana de componentes permite añadir nuevos componentes a nuestro Blueprint desde el propio Blueprint Editor, pudiendo añadir así objetos de colisión como capsulas, o añadir cámaras, partículas, efectos de sonido, etc.

Lo más importante de todo es que estos componentes que añadimos al Blueprint pueden ser referenciados desde propio Blueprint pudiendo así realizar cualquier tipo de acción con ellos como inicializarlos, destruirlos, cambias sus propiedades, responder a eventos.

Los componentes además pueden ser añadidos a variables para que puedan ser accedidos no solo por el Blueprint al que pertenece sino también desde otros Blueprints.

#### 5.3.1.1.2.2 Constructor (Construction Script)

Al igual que en C++ un constructor es una función que inicializa una instancia de su clase, el Constructor Script sería el equivalente a ese constructor, es una función que se llama cuando el Blueprint es creado permitiendo así definir una inicialización. Solo puede haber un Construction Script por Blueprint.

Por esto el Construction Script es una herramienta muy potente ya que contiene un nodo de grafo que es ejecutado al crear una instancia del Blueprint permitiendo ejecutar operaciones de inicializacion, esto es muy útil para acciones como añadir mallas y materiales, o inicializar valores.

Por ejemplo en el caso de este proyecto hay una clase padre Weapon Parent que contiene las variables necesarias para el funcionamiento del arma y para poder manejar todas las armas desde la clase padre, pero es el Contruction Script de cada clase hija la que inicializa estos valores llamando

a una función de la clase padre común a todas las armas, así cada arma en su constructor deberá llamar a esta función para inicializar sus variables. Esto se puede ver mejor en las siguientes fotos:

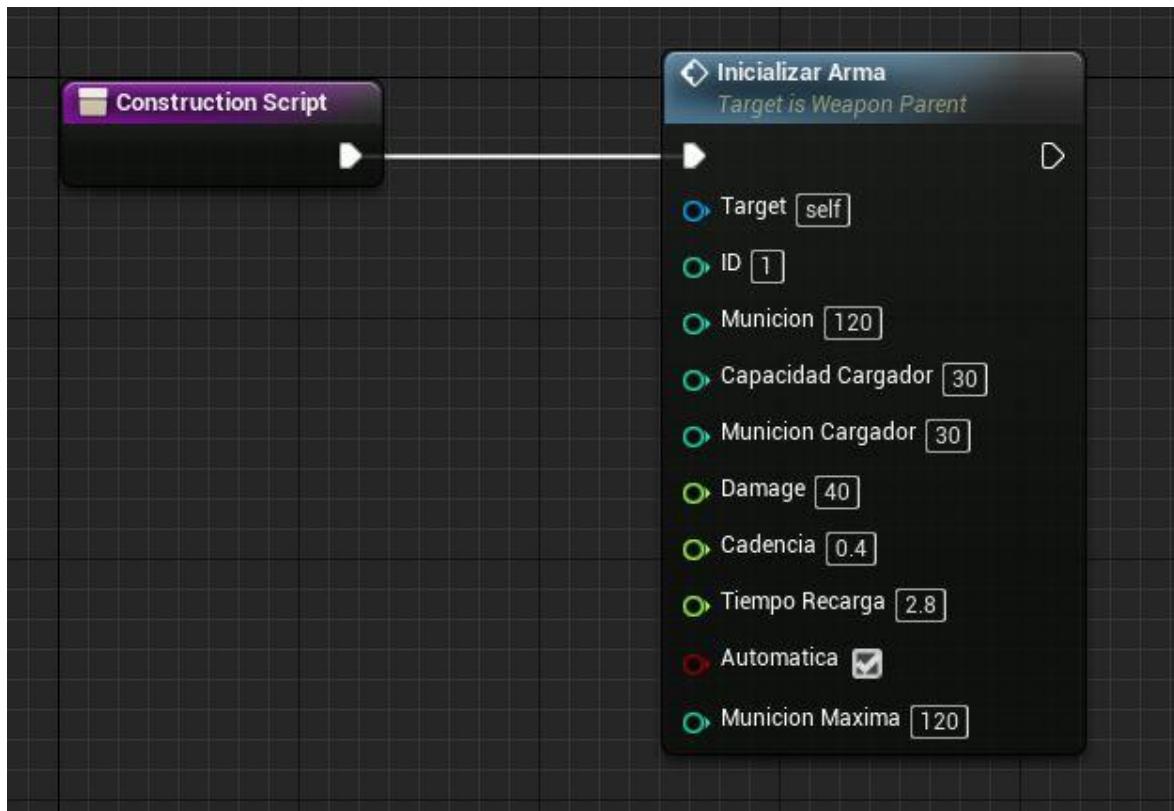


Figura 42: Contruction Script del arma AK-47

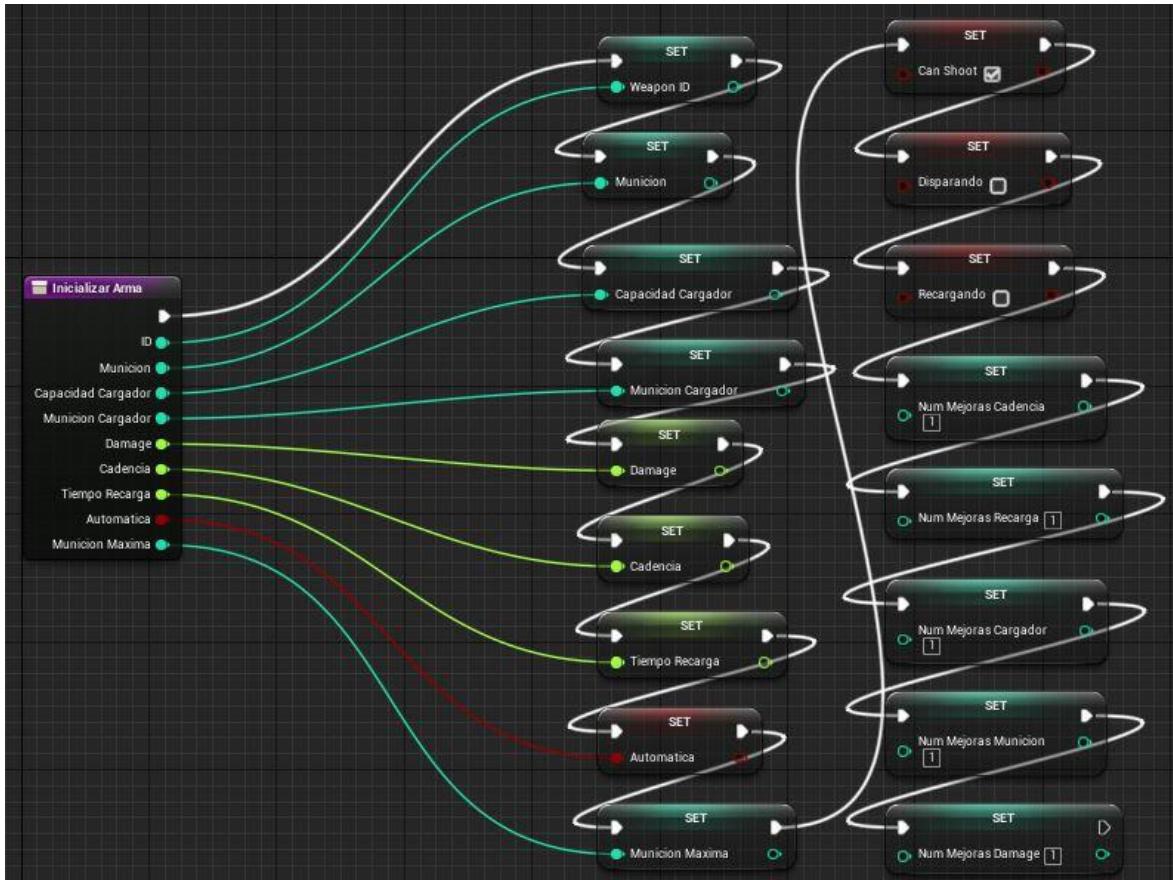


Figura 43: Función inicializar al que llaman los constructores de las armas

### 5.3.1.1.2.2.3 Event Graph

El Event Graph es el grafo general del Blueprint, cada Blueprint ya trae uno creado por defecto aunque por lo general empieza vacío. Los Event Graph se utilizan para realizar acciones en respuesta a eventos de unreal (como el OnComponentBeginOrOverlap), eventos personalizados creados por el propio usuario o incluso entradas de ratón, teclado, un mando, etc.

El Event Graph es usado para añadir funcionalidades comunes a todas las instancias del Blueprint, por ejemplo si al pulsar la F cerca de una puerta esta se abre, si esta función la hacemos en el Event Graph se ejecutara en todas las instancias de este Blueprint, así cada vez que estemos cerca de una puerta (que tenga este Blueprint) y pulsemos F esta se abrirá.

De esta manera los Event Graph son usados añadiendo uno o más eventos que actúan como puntos de entrada y luego se conectan a funciones o variables para conseguir las acciones deseadas.

### 5.3.1.1.2.2.4 Funciones

Las funciones son gráficos de nodos de un determinado Blueprint, que se pueden ejecutar desde el propio Blueprint o ser llamados desde otros. Las funciones tienen un único punto de entrada cuyo nombre es el mismo que el nombre de la función y un solo pin de salida de ejecución, lo que permite que cuando se llama desde otro Blueprint al acabar la función continuemos la lógica del Blueprint que la llamó.

Las funciones pueden ser puras o impuras. Una función pura es aquella que no puede modificar ningún atributo por lo tanto sería usadas por ejemplo para getters. Por el contrario las funciones impuras pueden modificar estados o miembros de la clase.

### 5.3.1.1.2.2.5 Variables

Las variables son las propiedades que guardan valores o referencias a Objects o Actores en el mundo. Podemos acceder a estas propiedades de forma interna a través del Blueprint que las contenga o hacerlas públicas para que puedan ser accedidas desde otros Blueprints.

### 5.3.1.1.2.2.6 Macros

Las macros son parecidas a las funciones, con la diferencia que pueden tener varios pin de entrada y salida de ejecución, de manera que a una macro se puede acceder desde distintas ejecuciones y su salida será el pin de ejecución correspondiente.

Por ejemplo digamos que tenemos 2 if else anidados, en vez de hacer esto en el grafo normal, podemos hacer una macro que contenga los 2 if elses, de manera que la entrada sería la entrada de ejecución mientras que la salida sería la ejecución resultante de los if elses, por ejemplo podría cumplirse el 2do if y se devolvería esta por la macro para continuar con la ejecución en el grafo que se llamó.

Para que quede más claro he hecho una pequeña demostración de una macro con el ejemplo anterior. La primera imagen sería la macro con los 2 if else, y la segunda imagen sería la llamada a la macro desde un evento personalizado del grafo. La macro nos permitiría aislar este código para que no esté en el propio grafo devolviendo la ejecución por el pin que necesitamos y además permitiendo que el código de la macro sea reutilizable.

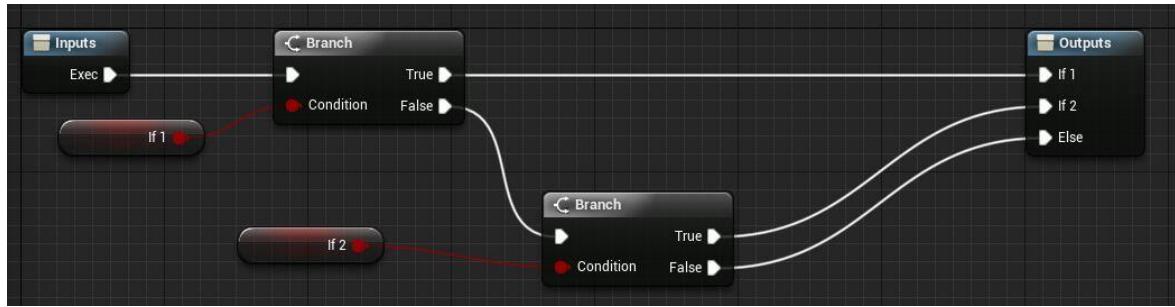


Figura 44: Ejemplo de Macro

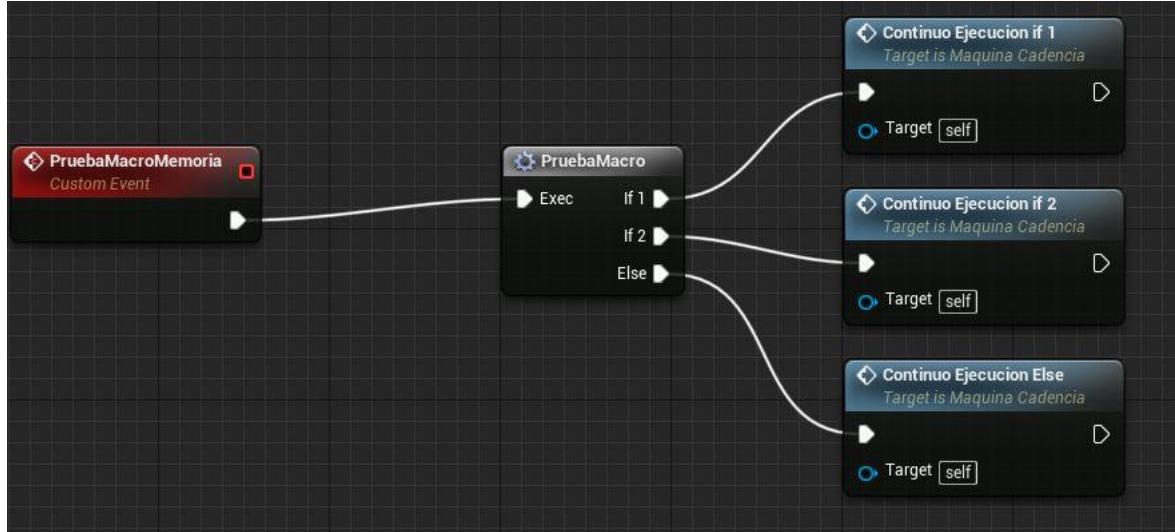


Figura 45: Ejemplo llamada a macro desde evento personalizado

### 5.3.1.1.2.3 ¿Qué tipos de Blueprints hay?

Hay varios tipos de Blueprint con distintas funcionalidades, los 2 más comunes son el Level Blueprint y el Blueprint Class aunque vamos a explicarlos todos brevemente estos 2 son las más importantes.

#### 5.3.1.1.2.3.1 Blueprint Class

Los Blueprint Class son los más comunes y usados por lo tanto son a los que nos referimos cuando únicamente decimos Blueprint, es un asset que permite añadir funcionalidades a las clases de juego rápidamente.

Los Blueprints son creados dentro del editor de Unreal de forma visual y son guardados como assets (un recurso) en el paquete de contenidos. Este tipo de Blueprint define una nueva clase o tipo de Actor que puede ser colocado dentro de mapas como instancias que se comportan de igual manera que lo haría otro Actor.

### 5.3.1.1.2.3.2 Level Blueprint

El Level Blueprint es un tipo de Blueprint especial que actua como grafo de nodos global de un determinado nivel. Cada nivel tiene su propio Level Blueprint creado por defecto y que puede ser editado desde el propio editor de Unreal. A diferencia del resto de Blueprints no se pueden crear más Level Blueprints a través de la interfaz del editor.

En un Level Blueprint se tiene acceso a todos los actores que estén en dicho nivel.

### 5.3.1.1.2.3.3 Data-Only Blueprint

Un Data-Only Blueprint es un Blueprint Class que contiene solo el código (los nodos graficos), variables y componentes heredados de su padre. Estos Blueprints permiten cambiar y modificar esas propiedades heredadas, pero no permiten añadir nuevos elementos. Son por lo tanto un reemplazo de las funcionalidades de las clases padres y que pueden ser utilizados por los diseñadores para cambiar propiedades o crear elementos con variación.

Los Data-Only Blueprint son editados en un editor compacto pero estos se pueden convertir en Blueprints completos fácilmente únicamente añadiendo código o variables nuevas, dejando de ser un Data-Only Blueprint y pasando a ser un Blueprint Class el cual se editaría en el editor completo.

### 5.3.1.1.2.3.4 Blueprint Interface

Un Blueprint Interface está compuesto de una o más funciones que pueden ser añadidas a otros Blueprints. Cualquier Blueprint que tenga la interfaz debe implementar todas las funciones definidas en esta.

Las funciones de la interfaz pueden dar funcionalidad en cada uno de los Blueprints que las han añadido. Esto es como el concepto de interfaz en programación que permite a diferentes tipos de objetos compartir funciones de una interfaz común.

Una parte importante de los Blueprint Interface es que permiten a los Blueprint enviar y compartir datos con otros Blueprints a través de mensajes a funciones de la interfaz.

Un Blueprint Interface puede crearse desde el editor visual de Unreal de manera sencilla como si fuera un Blueprint Class pero con la diferencia de que este tiene algunas limitaciones como por ejemplo que no permite ni añadir nuevas variables, ni editar grafos, ni añadir componentes.

La mayor ventaja que tiene este tipo de Blueprint es que permiten un método común de interactuar con objetos totalmente distintos que comparten alguna funcionalidad específica. Por ejemplo si tenemos un árbol y un coche y los dos pueden ser disparados por un arma de fuego y pueden ser dañados, se puede crear un Blueprint Interface que contenga una función llamada por ejemplo “RecibirDaño”, de esta manera tanto el coche como el árbol implementarían esta función RecibirDaño, por lo que al ser disparados se envía un mensaje a través de la interfaz pudiendo tratar así dos objetos totalmente distintos como un coche y un árbol de manera común.

#### **5.3.1.1.2.3.5 Blueprint Macro Library**

Como su propio nombre indica es una especie de librería de macros, si recordamos lo que hacía una macro (véase apartado 5.3.1.1.2.2.6 Macros), un Blueprint Macro Library sirve para almacenar distintos tipos de macros para que estas puedan ser reutilizadas y llamadas desde cualquier Blueprint.

Es decir permite que las macros sean comunes y que se puedan utilizar desde cualquier sitio cuando se necesiten.

#### **5.3.1.1.3 Renderizado y gráficos**

El sistema de renderizado de Unreal Engine 4 es nuevo y soporta herramientas avanzadas de DirectX11 que permiten iluminación global, transparencia con iluminación, efectos de post procesado como el Bloom, así como simulación de partículas mediante la GPU utilizando campos vectoriales.

Un aspecto importante es el cambio a la hora de renderizar, en Unreal Engine 4 se utiliza el sombreado diferido más conocido como Deferred Shading mientras que en Unreal Engine 3 se usaba el Forward Lighting.

Esto permite utilizar una gran cantidad de luces en la escena disminuyendo muchísimo su coste ya que el Deferred Shading deja el cálculo de luces para el final y calcula toda la iluminación de la escena a la vez.

### 5.3.1.1.3.1 Luces

En Unreal Engine 4 existen 4 tipos de luces: Spot, Point, Directional y Sky.

Las **Spot Light** emiten luz desde un punto, pero la luz es emitida en forma cónica, el ejemplo más fácil de entender es el de la luz emitida por una linterna.

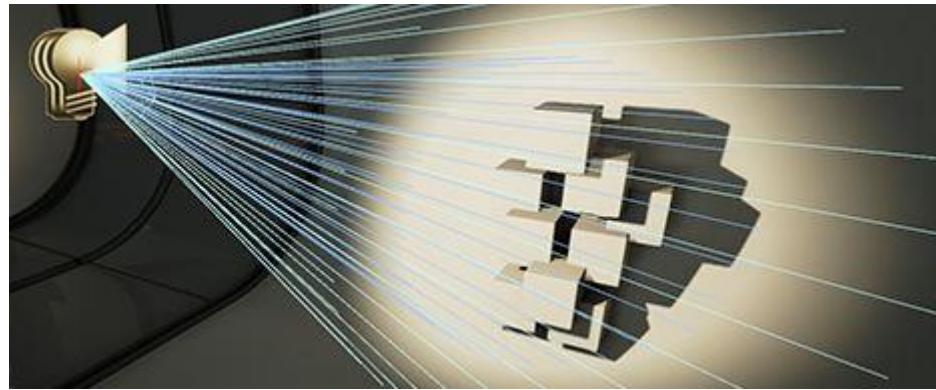


Figura 46: Ejemplo Spot Light

En las **Point Light** la luz es emitida desde un punto y en todas las direcciones la podríamos comparar con la luz emitida desde una bombilla.



Figura 47: Ejemplo Point Light

Las **Directional Light** se usan normalmente para simular la luz exterior, la luz natural emitida por el sol o por cualquier fuente de luz que este extremadamente lejos. Este tipo de luz sería el equivalente a luz emitida por el sol.

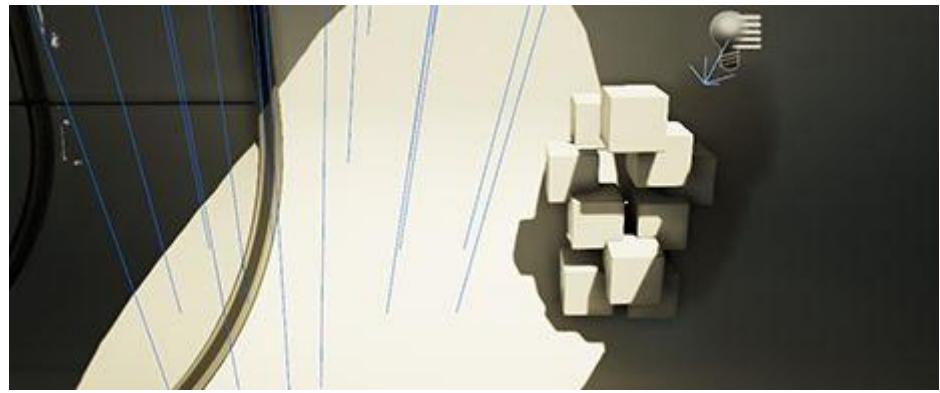


Figura 48: Ejemplo Directional Light

Las **Sky Light** aplican una luz general y uniforme a todos los elementos de una escena, es una especie de luz ambiente o luz indirecta que permite que se distingan los objetos aun en zonas de penumbra.

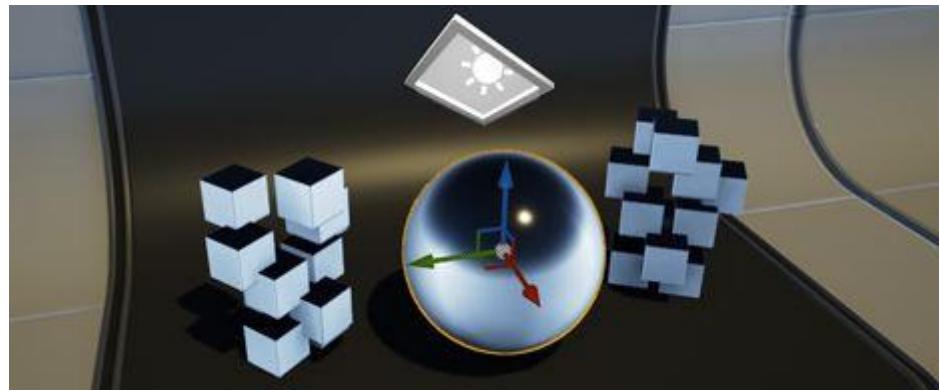


Figura 49: Ejemplo Sky Light

Las 4 imágenes anteriores están sacadas de la documentación oficial de Unreal:

<https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/LightTypes/index.html>

Dado que Postwar: Hopeless Humanity está ambientado en una noche oscura con iluminación exterior prácticamente nula la Directional light y Sky light son casi imperceptible **recayendo así toda la importancia de la luz en las SpotLight y sobre todo en las PointLight**. A continuación 3 imágenes sacadas del proyecto en las que se ven tanto SpotLight como PointLight.



Figura 50: Ejemplo de una sala del proyecto con iluminación Spot Light y Point Light.

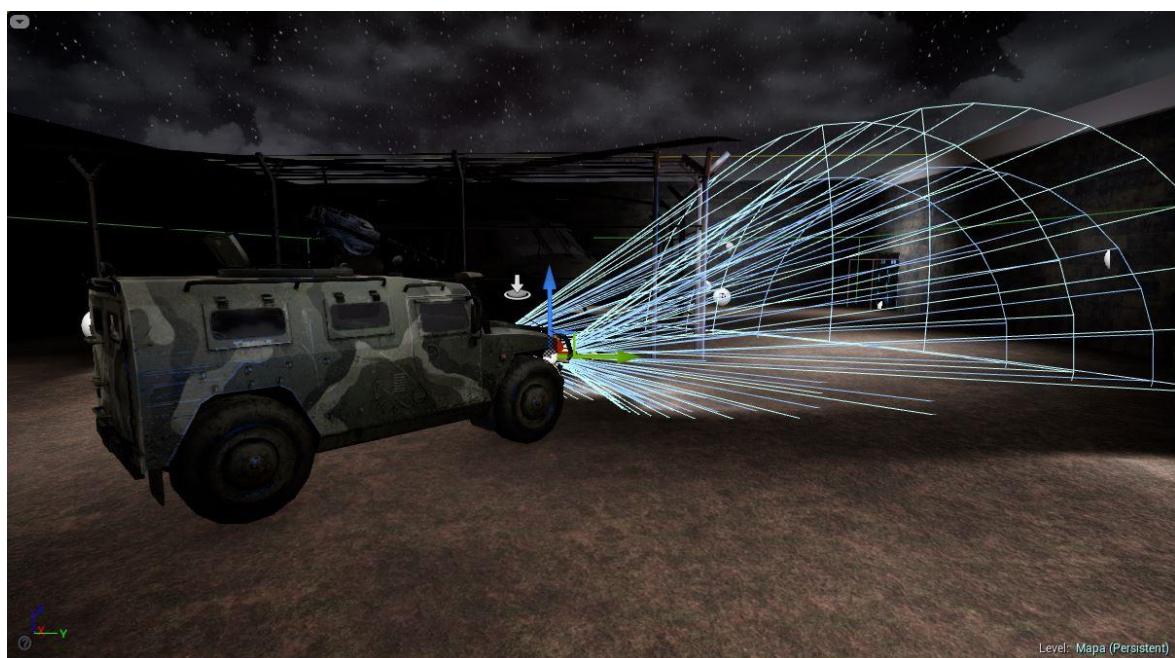


Figura 51: Ejemplo 1 iluminación únicamente con Spot Light sacada del proyecto



Figura 52: Ejemplo iluminación únicamente con Spot Light sacada del proyecto

Aunque estos 4 tipos de luces son bastante distintos todos comparten una propiedad llamada **Mobility**. Existen 3 tipos de **Mobility**: **Satic**, **Stationary** y **Movable**.

#### 5.3.1.1.3.1.1 Static lights

Las Static Light tienen una calidad media, una mutabilidad reducida y el coste de rendimiento más bajo. Debido a esto el principal uso de las luces estáticas es para dispositivos de baja potencia en plataformas móviles.

Antes de entrar en profundidad con las luces estáticas necesitamos definir lo que es un mapa de luces o más bien conocido como **lightmap**. Un lightmap es una estructura de datos usada en el lightmapping, una forma de cachear las superficies almacenando el brillo de las superficies en una textura para poder utilizarla luego en tiempo real. Es decir, se hace un cálculo de cómo afectan las luces a una escena, se preparan con esos cálculos una serie de imágenes y luego, en tiempo de ejecución se les aplica a los objetos para simular la iluminación.

Los lightmaps se sueles aplicar a los objetos estáticos en aplicaciones 3D en tiempo real como los videojuegos, ya que es una forma de calcular la iluminacion global con un coste computacional muy bajo.

Las Static Light son luces que no se pueden mover ni cambiar sus parámetros como intensidad o color durante el juego. Son calculadas solo dentro de los *lightmaps*, y una vez procesadas no tienen impacto en el rendimiento. Debido a que son procesadas una vez para obtener los *lightmaps* los objetos dinámicos no pueden ser afectados por las Static Light.

Como las Static Light solo utilizan *lightmaps* las sombras son calculadas antes del gameplay, lo cual significa que no permiten sombras dinámicas. Por el contrario los objetos estáticos iluminados por esta luz son capaces de producir sombras de área.

#### 5.3.1.1.3.1.2 Stationary lights

**Las Stationary Light tienen la mayor calidad, una mutabilidad media y un coste de rendimiento medio.**

La principal diferencia que tiene este tipo de luces respecto a las estáticas es que estas pueden cambiar la intensidad y el color de la luz durante el juego. Un inconveniente es que no pueden cambiar su posición, por eso la mutabilidad es media. Aunque estos cambios solo afectan a la iluminación directa ya que la iluminación indirecta es precalculada y no cambiará.

Las sombras que producen estas luces son de mayor calidad y se verán afiladas aunque la resolución del *lightmap* sea un poco baja.

#### 5.3.1.1.3.1.3 Movable lights

**Las Movable Light tienen una buena calidad, la mutabilidad más alta y un alto coste de rendimiento.**

Estas luces producen iluminación y sombras completamente dinámicas, pueden cambiar su posición, rotación, color, brillo desvanecimiento, radio y todas las propiedades que posee una luz. Ninguna de la iluminación que producen es calculada dentro de los *lightmaps* y no admiten iluminación indirecta.

Las Movable Light configuradas para emitir sombras usan toda la escena para calcular las sombras dinámicas, teniendo por tanto un coste bastante elevado. Este coste depende del número de mallas afectadas por la luz y del número de polígonos que tengan estas mallas, es decir, una luz móvil con

un radio muy grande que afecte a muchos objetos tendrá un coste computacional mucho mayor que una luz pequeña que afecte solo a unos pocos.

### 5.3.1.1.3.2 Materiales

Un material es un recurso que se puede aplicar a una malla para controlar su apariencia visual en la escena.

Los materiales son una parte fundamental en el proyecto a la hora de intentar conseguir la ambientación o estética deseada, ya que un material no es solo una imagen aplicada a una malla sino que un material es capaz de definir el tipo de superficie del objeto al que se le aplica mediante el color, brillo, opacidad, rugosidad, etc.

Si entramos un poco más en detalle, cuando la luz de la escena incide sobre una superficie de un objeto el material de dicho objeto es el encargado de decir como la luz interactúa con esa superficie, diciendo por ejemplo la cantidad de luz absorbida, reflejada, etc. Estos cálculos se realizan utilizando los datos que hay en el material.

Al igual que los Blueprints los materiales no están escritos mediante código, sino que usan nodos gráficos llamados **Material Expressions** dentro del **Material Editor**. Cada nodo contiene una parte de código **HLSL** (High-Level Shading Language), este es uno de los lenguajes utilizados por ejemplo para hacer shaders en OpenGL, por lo que aunque no seamos consciente de ellos **lo que hacemos cuando creamos un material de manera indirecta es programar shaders**.

Los materiales pueden ser muy variados, pueden haber algunos muy sencillos en los que se use únicamente una imagen de textura (véase figura TODO) o algunos materiales más complejos que intenten simular efectos reales como el agua (véase figura TODO).

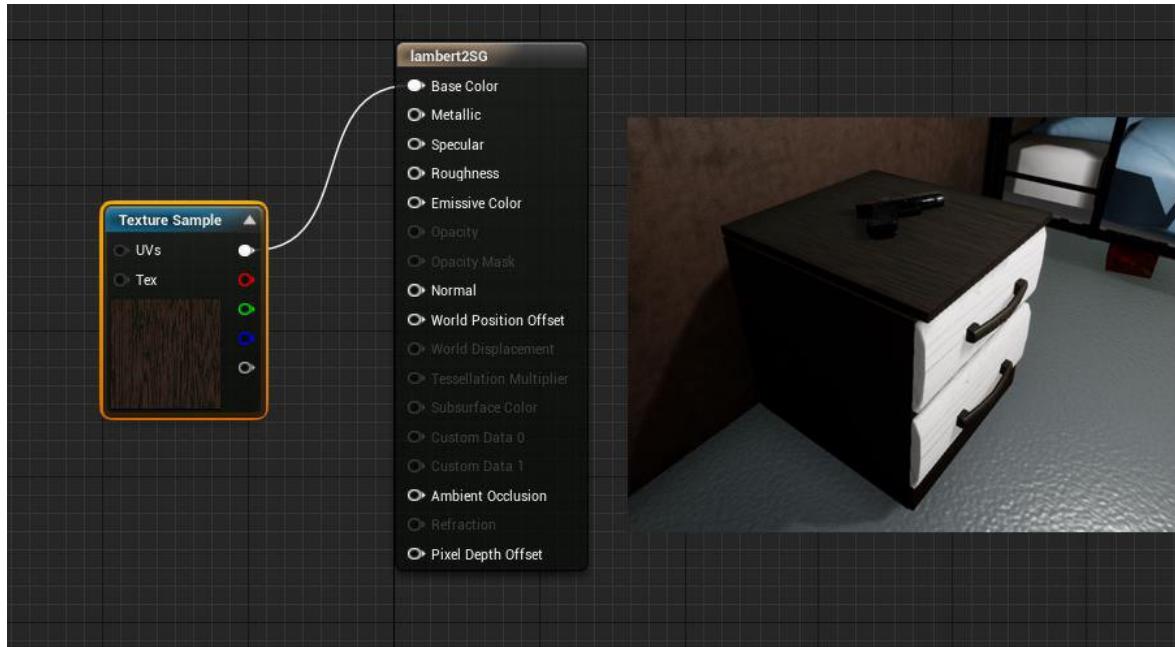


Figura 53: Material simple y su aplicación en el proyecto

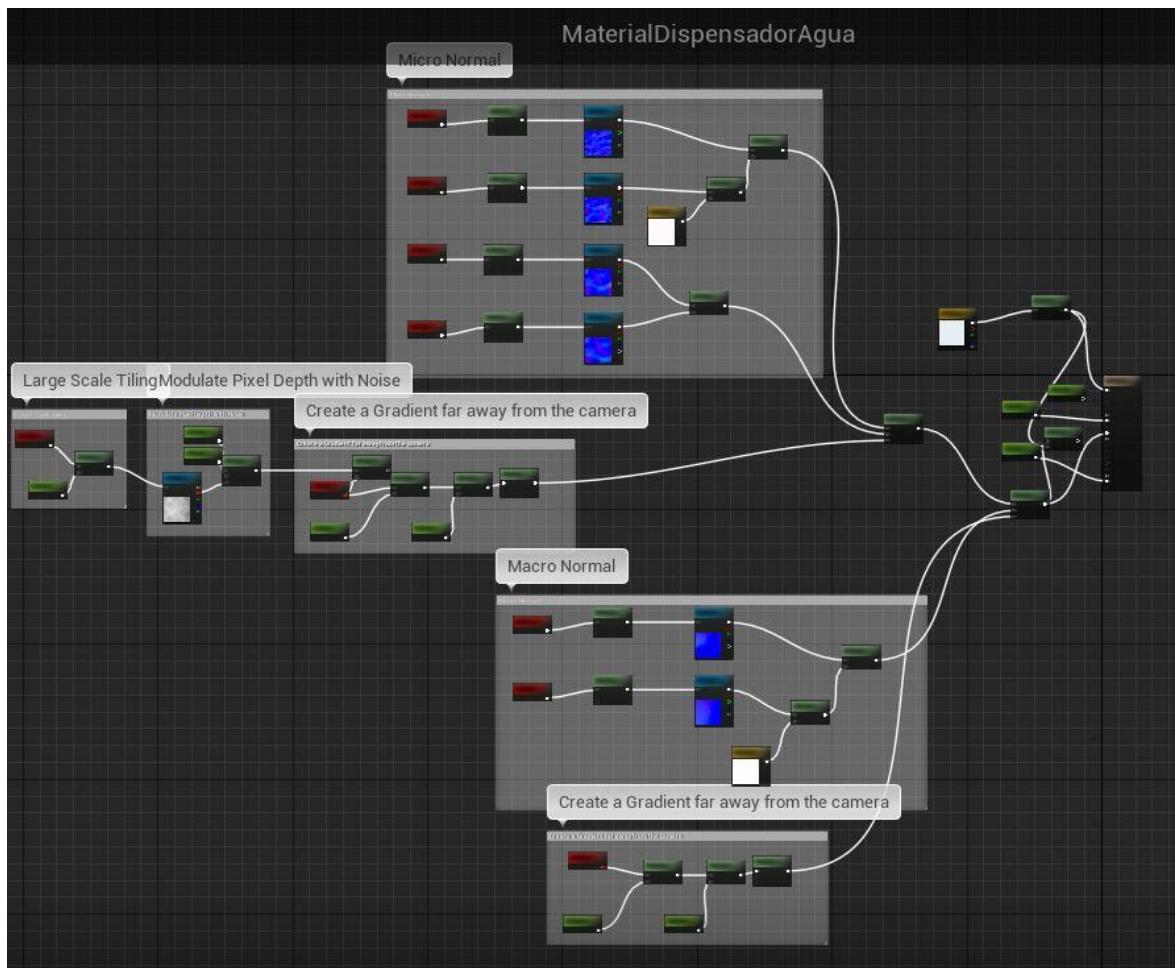


Figura 54: Material complejo (simulación de agua)



Figura 55: Aplicación de material complejo en el proyecto

Ambos materiales pertenecen a este proyecto, podemos ver como hay materiales muy simple con una sola imagen de textura como el de la primera foto de la mesita de noche, y otros materiales más complejos como el agua que esa dentro de ese bidón, la cual tiene una refracción (se puede observar mirando el mapa de atrás), además que la incidencia de la luz sobre esta y las normales del material van cambiando con el tiempo para dar la sensación de que el agua no es estática sino que hay un poco de movimiento en ella.

#### 5.3.1.1.3.3 Postprocesado (PostProcess Volume)

Los PostProcessVolume son una especie de cajas invisibles que permiten añadir una capa de postprocesado que se aplica a todo lo que está dentro de dicha caja. Esto permite a los artistas y diseñadores modificar la apariencia general de la escena y añadir efectos como la oclusión ambiental, el Bloom, Blur, cambio de color, etc.

### 5.3.1.1.4 Audio y sonido

El sonido es uno de los aspectos más importantes para crear ambientes creíbles y permitir la inmersión del usuario en el juego. De los sonidos ambientales en el nivel, a los sonidos interactivos de vehículos o armas, diálogos de personajes, etc. El audio en un juego puede crear una gran experiencia en el usuario o destruirla por completo.

Hacer que el audio de un juego suene como debería sonar es algo bastante complicado, por ello Unreal Engine 4 nos proporciona herramientas y características para poder moldear los sonidos del juego a nuestro antojo y poder darles la sensación deseada.

El sistema de audio de Unreal Engine 4 se compone de varios componentes que trabajan conjuntamente para conseguir la mejor experiencia de audio para el usuario. Al importar un archivo de audio se pueden modificar algunos de sus propiedades básicas como el Volumen o *pitch* o algunas propiedades más complejas como la atenuación del sonido **Sound Attenuation**.

Ademas UE4 también permite crear sonidos compuestos llamados **Sound Cues** y que son modificados en el **Sound Cue Editor**.

#### 5.3.1.1.4.1 Sound Cue Editor

De la misma forma que para crear o modificar materiales existía un editor de materiales para crear sonidos complejos o editar libremente sonidos existe el **Sound Cue Editor**. Este es un editor de nodos gráfico parecido al editor de materiales o al de Blueprints.

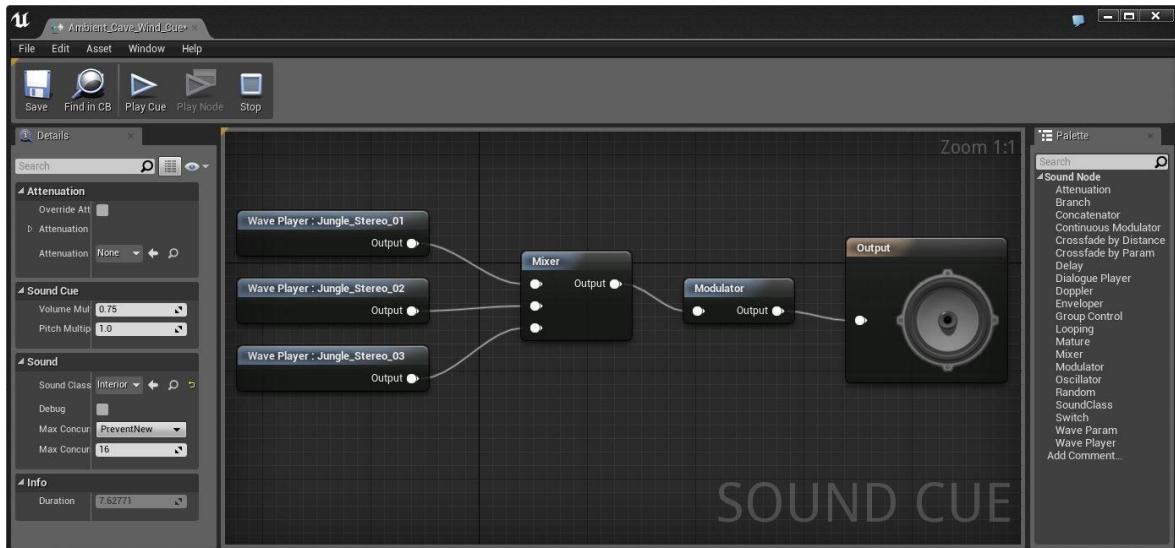


Figura 56: Ejemplo Sound Cue - Imagen sacada de la documentación oficial de Unreal Engine 4

El Sound Cue Editor trabaja con **Sound Waves** y como salida tiene **Sound Cue**.

Un **Sound Wave** es la representación de un archivo de audio importada al Sound Cue Editor, la cual puede ser modificada libremente dentro del propio editor añadiendo efectos de sonidos complejos.

La salida de audio de la combinación de nodos creada en el Sound Cue Editor se guarda como un **Sound Cue**.

Por defecto cada vez que creamos un archivo en el Sound Cue Editor tendrá un valor de salida predeterminado que viene indicado con un altavoz que será el Sound Cue final.

#### 5.3.1.1.4.2 Ambient Sound Actor

Unreal Engine 4 simplifica el proceso mediante el cual puedes producir o modificar sonidos ambiente a partir del uso de **Ambient Sound Actor**, de tal manera que simplemente con arrastrar un Sound Cue o Sound Wave al nivel UE4 lo convertirá en un **Ambient Sound Actor** permitiendo así modificar muchas propiedades a las que antes no teníamos acceso, como por ejemplo el radio de atenuación.

Un Ambient sound Actor se comporta como un sonido del mundo real, escuchándose este más fuerte cuanto más cerca estas de la fuente del sonido, de la misma forma si estamos demasiado lejos no oiremos el sonido.

TODO AÑADIR FOTO DEL ACTOR SOUND AL COMPRAR UN POWER UP

### 5.3.1.1.4.3 Sound Attenuation

El **Sound Attenuation** es la propiedad del sonido que permite bajar el volumen dependiendo de la distancia entre el jugador y el emisor del sonido. El Sound Attenuation funciona con 2 radios, **MinRadius** y **MaxRadius**. Las funcionalidades de estos radios son sencillas pero muy importantes.

El MinRadius es el radio en el que el sonido se escucha al 100% de volumen, mientras el jugador este dentro de este radio el sonido no tendrá ninguna atenuación.

El MaxRadius es el radio máximo al que el sonido puede llegar, cuando el jugador sale de este radio no escuchara el sonido.

Cuando el jugador se encuentra entre el MinRadius y el MaxRadius entonces el volumen del sonido estará atenuado desvaneciéndose linealmente desde el 100% (dentro del MinRadius) y el 0% (fuera del MaxRadius). La velocidad en la que ocurre este fundido depende del parámetro **Distance Algorithm** y la propiedad **DistanceModel** que proporcionan varias curvas de desvanecimiento para controlar el volumen entre radios.

Además también se puede modificar la forma de atenuación mediante el **Attenuation Shape**. Con esta propiedad podemos definir la forma del volumen de atenuación, pudiendo ser esta por ejemplo una esfera, una caja, un cono o una capsula.

### 5.3.1.1.5 Animaciones

Aunque Unreal Engine 4 trae una gran cantidad de herramientas y de formas para poder trabajar con animaciones vamos a hablar solo de las básicas que hemos usado para este proyecto.

Unreal Engine 4 tiene 4 herramientas principales para trabajar con animaciones:

- **Skeleton Editor:** este sirve para examinar los esqueletos y las Skeletal Mesh. En este puedes añadir *sockets* que son una especie de puntos guía a los que después podrás vincular un modelo u actor, esto se ha usado por ejemplo en las armas que se vinculan a un socket colocado en la mano del esqueleto.
- **Skeletal Mesh Editor:** aquí puedes añadir LOD's o asigna materiales a una Skeletal Mesh.

- **Animation Editor:** en el podrás trabajar con secuencias de animación (**Animation Sequences**), **Animation Montages** y **Blend Spaces**. En este editor podremos crear las distintas animaciones de secuencia, los montajes de animación, ajustar tiempos de animaciones, recortar animaciones, mezclarlas y añadir notificaciones a las animaciones.
- **Animation Blueprint Editor:** es el editor final. En este editor podrás crear reglas para elegir que animaciones reproducir o bien también se pueden usar cosas más complejas como máquinas de estados para elegir diferentes animaciones.

#### **5.3.1.1.5.1 Blending animation (Blend Spaces)**

El blending animation sería una mezcla entre dos animaciones, es decir que si estas andando por ejemplo y pasas a estado de correr no hay un salto directo de la animación de andar y al frame siguiente la animación de correr, sino que hay una especie de transición (blending) entre las 2 animaciones lo que permite que el paso de una animación a otra se vea de manera natural y fluida.

Hemos estado hablando del blending entre 2 animaciones pero no de cómo podemos realizarlo, para ello está el **Blend Space**. El **Blend Space** es una herramienta de Unreal que permite hacer una transición de manera sencilla entre 2 o más animaciones. Se puede elegir entre:

- **Blend Space 1D:** es el más sencillo, simplemente es una mezcla entre 2 o más animaciones pero permitiendo un único valor de entrada para la transición entre animaciones como puede ser la velocidad por ejemplo.
- **Blend Space:** es un poco más complejo que el anterior, la diferencia fundamental es que permite 2 valores de entrada, por lo tanto la transición entre las animaciones estarán es una especie de gráfico y dependerán del valor de los 2 parámetros situados en los ejes X e Y. Estos 2 valores podrían ser por ejemplo la velocidad y dirección, así podemos hacer un blending entre la animación de andar y correr dependiendo de la velocidad pero además se haría blending entre las animaciones de correr hacia delante o hacia los lados dependiendo de la dirección.

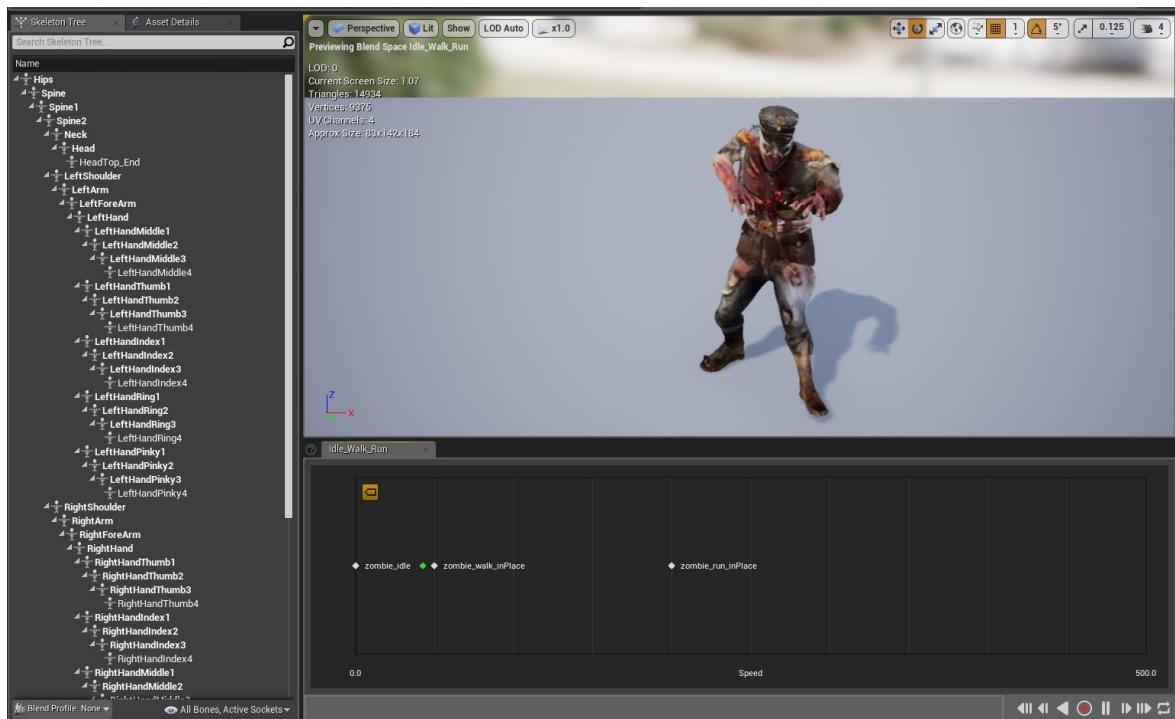


Figura 57: Ejemplo blending animation del proyecto

La imagen anterior muestra el blending para las animaciones del zombi que se ha realizado en este proyecto. Sería un blending entre 3 animaciones, la animación idle (cuando el zombi está quieto), la animación de andar y la animación de correr que seria los 3 puntos blancos que se ven en el gráfico. El punto verde sería el valor del parámetro que se utiliza para pasar de una a otra por lo que en ese caso estaría haciendo un blending de aproximadamente 90% la animación de andar y 10% la animación Idle.

La variable usada es la velocidad, por lo tanto cuando el zombi alcance cierta velocidad empezara a hacer un blending entre la animación de andar y de correr.

TODO revisar estos 2 últimos párrafos igual tiene más sentido que estén en la parte donde explico como he desarrollado las animaciones.

### 5.3.1.1.5.2 Animation blueprint

Un Animation Blueprint es una Blueprint especializado que controla la animación de un Skeletal Mesh. Este al igual que los materiales o el sonido tiene un editor propio llamado **Animation Blueprint Editor** siguiendo la coherencia de Unreal funciona con la unión de nodos gráficos.

Dentro del Animation Blueprint Editor se puede mezclar animaciones, controlar directamente la animación en huesos determinados de una malla o crear una lógica encargada de seleccionar la

animación necesaria en cada momento. Este editor tiene una salida final que sería la pose concreta en cada *frame* en la que se encuentra la malla.

Hay 2 componentes fundamentales en el Animation Blueprint que se utilizan para crear la animación final en cada fotograma. Estos son:

- **EventGraph:** el cual se encarga de actualizar valores que luego serán necesarios para elegir una animación.
- **AnimGraph:** es el encargado de conducir a las máquinas de estado para que elijan la animación correcta dependiendo de los valores ya actualizados en el EventGraph.

#### 5.3.1.1.5.3 Animation Retargeting (Different Skeletons)

**Animation Retargeting** es una característica de Unreal que permite que las animaciones se reutilicen entre los personajes que no comparten el mismo esqueleto.

El proceso de reasignación de animaciones entre personajes que tienen diferentes esqueletos utiliza un recurso llamado **Rig** para pasar información de los huesos de un esqueleto a los huesos del otro. Después de selección el Rig este se compartirá entre el esqueleto origen y el esqueleto destino, de esta manera los huesos de ambos esqueletos coincidirán y las animaciones por tanto podrán ser usadas en los 2 esqueletos.

#### 5.3.1.1.5.4 Notificación en animación

Las notificaciones en animación conocidas en UE4 como **AnimNotifies** o solo **Notifies** proporcionan una manera para configurar eventos que quieras que se produzcan en un punto específico de una animación.

Las notificaciones se usan normalmente para agregar efectos como sonidos o partículas durante la animación, pero también permiten notificaciones personalizadas para poder ejecutar la lógica que se desee.

### 5.3.2 Desarrollo de mecánicas de juego

En este apartado explicaremos las mecánicas de juego que se han realizado para este proyecto y que ya han sido detalladas en el GDD. Hay que decir que todas las mecánicas han sido programadas mediante la programación visual que ofrece Unreal (Scripting Visual Blueprint).

Aunque la primera impresión puede ser que esta manera visual de programar no va a dar resultado o dejaría bastante que desear, a medida que te vas adentrando en el mundo del Blueprint y te vas dando cuenta de lo completo que es este sistema visual te das cuenta que puedes hacer casi cualquier cosa que te imagines únicamente con la programación visual.

Una vez he aprendido a usar este sistema de Blueprints he sido consciente del gran potencial que tiene Unreal y de la gran cantidad de cosas que se podrían hacer de una manera relativamente sencilla con este motor.

Para intentar facilitar la comprensión del lector y que se puedan identificar fácilmente entre variables, funciones o clases vamos a dar color a estas 3. De esta forma se representaran así:

- Nombre de clase en lila. Por ejemplo: **FirstPersonCharacter**.
- Nombre de variable en azul. Por ejemplo: **posApuntadoAk**.
- Nombre de función o evento en verde. Por ejemplo: **getPosApuntado**.

### 5.3.2.1 Blueprint First Person Character (personaje principal)

El Blueprint del First Person Character es el más importante de todos, ya que es el Blueprint central en el que se apoyan el resto para obtener información del juego como puede ser las muertes del player, su puntuación, etc.

Ademas es el Blueprint del personaje al que controlamos y es por esto que es el que más funciones

y variables tiene, ya que tiene que controlar toda la lógica del jugador.

Debido a la extensión de este Blueprint se va a intentar explicar todo pero solo se hará especial hincapié mostrando la programación (visual) en aquello que sea importante saber.

#### 5.3.2.1.1 Componentes

Como se puede observar nuestro Blueprint del personaje no tiene una gran cantidad de componentes, tiene simplemente los necesarios para el desarrollo del juego como son:

- **CapsuleComponent:** este componente se crea por defecto en este tipo de Blueprints. Se encarga de controlar la colisión de nuestro personaje, es una capsula que simularía la colisión de un cuerpo humano.

- **FirstPersonCamera:** este componente de tipo *CameraComponent* es lo que

*Figura 58: Componentes Blueprint personaje*

correspondería a los ojos del personaje. Es la cámara a través de la cual podemos observar todo lo que rodea a nuestro personaje. El campo de visión más conocido como **FOV** (Field Of Vision) es de 90 grados cuando no se está apuntando y de 35 grados cuando se está apuntando (un zoom).

- **Mesh2P:** es un componente de tipo **Skeletal Mesh**, en nuestro caso al ser un juego en primera persona simplemente son los brazos del personaje, y por lo tanto son hijos de la cámara para que sigan el movimiento y la rotación de esta y mantener los brazos y por lo tanto el arma que sostienen en una posición constante respecto a la pantalla. Si el juego fuera en 3era persona entonces usaríamos el **Mesh** creado por defecto de Unreal que es hijo de la capsula.

Al ser un Skeletal Mesh esto nos permite agregar animaciones a esta malla, cosa que hemos usado por ejemplo para una animación de disparo, animaciones de recarga dependiendo del arma a recargar, animación de correr, etc.

**A estos componentes habría que añadir 2 componentes más que serían las armas que tendríamos equipada.** Estas armas no están como componentes por defecto ya que al principio aparecemos sin ningún arma, además de que para adquirirlas hay que ir comprando y solo podemos tener 2 al mismo tiempo, debido a esto los componentes de las armas se irán añadiendo y eliminando dependiendo de las armas que vamos comprando.

### 5.3.2.1.2 Variables

Como ya hemos dicho este era el Blueprint más importante y por lo tanto el que más variables y funciones tiene. El número de variables es tan extenso que creo que la mejor forma de mostrarlas es mediante una tabla.

Siguiendo con la estética de la portada usaremos los colores negro y amarillo para el contenido de esta y el resto de tablas.

FirstPersonCharacter Variables		
Nombre	Tipo	Descripción

<b>GunOffset</b>	Vector	Es una pequeña cantidad que se suma a la posición del arma para obtener como resultado el punto exacto desde el que saldría el proyectil a disparar.
<b>BaseTurnRate</b>	Float	Controla la velocidad de la cámara en el eje X.
<b>BaseLookUpRate</b>	Float	Controla la velocidad de la cámara en el eje Y.
<b>UsingMotionControllers?</b>	Boolean	Controla si se están usando mandos especiales para realidad virtual.
<b>Equipada</b>	Weapont Parent	Es la referencia al arma equipada en todo momento, la cual recibiría las acciones de disparar, apuntar, recargar, etc.
<b>Arma1</b>	Weapont Parent	Se correspondería al primer arma de tu inventario, la cual puedes tener equipada o no.
<b>Arma2</b>	Weapont Parent	Se correspondería al segundo arma de tu inventario, puede estar equipada o no.
<b>Apuntando</b>	Boolean	Variable de tipo booleano que es verdadera cuando estas apuntando con la mirilla del arma y falsa en el resto de casos.
<b>PosNormal</b>	Transform	Es la transformación que da la posición, rotación y escala de los brazos cuando no se está apuntando con el arma.
<b>PosApuntadoAk</b>	Transform	Es la transformación que da la posición, rotación y escala de los brazos cuando se está apuntando con una AK-47.
<b>PosApuntadoShotGun</b>	Transform	Es la transformación que da la posición, rotación y escala de los brazos cuando se está apuntando con una escopeta.
<b>PosApuntadoM4</b>	Transform	Es la transformación que da la posición, rotación y escala de los brazos cuando se está apuntando con una M4A1.

<b>PosApuntadoM60</b>	Transform	Es la transformación que da la posición, rotación y escala de los brazos cuando se está apuntando con una M60.
<b>PosApuntadoUMP</b>	Transform	Es la transformación que da la posición, rotación y escala de los brazos cuando se está apuntando con una UMP45.
<b>PosApuntadoG36</b>	Transform	Es la transformación que da la posición, rotación y escala de los brazos cuando se está apuntando con una G36.
<b>PosApuntadoScarL</b>	Transform	Es la transformación que da la posición, rotación y escala de los brazos cuando se está apuntando con una Scar-L.
<b>NuevaPosApuntado</b>	Transform	El resultado de la función <code>getPosApuntado</code> , que devolvería en esta variable una de las transformaciones anteriores, facilitando el código ya que la lógica de elegir una de las posiciones anteriores estaría en una función y en el resto del código solo usaremos esta variable.
<b>Puntuación</b>	Integer	Esta variable corresponde con el número de puntos o dinero que tiene el jugador.
<b>VidaActual</b>	Integer	Cantidad de vida actual del jugador, si es igual a 0 estas muerto.
<b>VidaMaxima</b>	Integer	El valor máximo al que puede llegar la vida del jugador, por defecto son 10 puntos, aunque con el power-up de resistencia se incrementarían hasta los 15.
<b>TiempoRecargaBase</b>	Float	Es el tiempo base que se añade a cada uno de los tiempos de recargas de las armas.
<b>Revivir</b>	Boolean	Variable booleana que es verdadera cuando has adquirido el power-up de

		revivir y falsa cuando no lo has comprado.
<b>Reviviendo</b>	Boolean	Variable booleana que dice si estas en estado de revivir.
<b>TimmerHandleRecuperacionVida</b>	TimmerHandle	Variable usada para reanudar o pausar la recuperación de vida del jugador.
<b>TiempoRegeneracionVida</b>	Float	Tiempo que tiene que pasar sin ser dañado por un enemigo para poder empezar a recuperarte vida automáticamente con el tiempo.
<b>Sprint</b>	Boolean	Variable booleana que es verdadera si estas corriendo y falsa en cualquier otro caso.
<b>CambiandoArma</b>	Boolean	Variable verdadera mientras se está cambiando de arma (utilizada por ejemplo para no poder disparar ni apuntar mientras se hace la transición de un arma a otra).
<b>PartidaReference</b>	Partida	Variable que guarda una referencia de un objeto tipo Partida con el que se pueden hacer acciones como actualizar el ranking.
<b>EnPausa</b>	Boolean	Variable booleana que es verdadera mientras el juego está en el menú de pausa y falsa en cualquier otro caso.
<b>Ranking</b>	Ranking	Variable que guarda una referencia a un objeto de tipo ranking a partir del cual podremos acceder a la información del ranking.
<b>RankingSubClass</b>	SaveGame	Variable que almacena una referencia a un objeto de tipo <b>SaveGame</b> a partir del cual podremos guardar nuevos datos.
<b>Nombre</b>	String	Aquí se almacena el nombre del jugador en caso de que haya alcanzado

		los números necesarios para acceder al ranking.
<b>ZombisMatados</b>	Integer	Cantidad de zombis matados.

Figura 59: Tabla de variables del personaje

### 5.3.2.1.2 Funciones

En este apartado se incluye una tabla explicativa de las funciones que hay en el Blueprint del FirstPersonCharacter, donde se explicara brevemente la lógica de cada función.

Las funciones importantes serán explicadas con mayor detalle después de la tabla incluyendo imágenes de la lógica de estas.

**En la tabla de funciones también se incluirán los customEvents y los inputsAction propios que están en el EventGraph de este Blueprint.**

FirstPersonCharacter Functions	
Nombre	Descripción
<b>ConstructionScript</b>	Constructor de la clase, en él se establece la vida máxima y la vida actual del personaje.
<b>GetEquipoID</b>	Función que devuelve un array con los IDs de las armas que tienes en el equipo actualmente. Esto es usado por ejemplo cuando te acercas a un arma para comprarla ya que primero comprueba que no esté ya en el equipo.
<b>EquiparArma</b>	Añade un arma al equipo al comprarla. Si ya tienes 2 armas en el equipo (este es el máximo), entonces se llamará a la función <b>sustituirArma</b> . Lo veremos en mayor detalle más abajo.
<b>CambiarArma</b>	Realiza el cambio de arma entre las 2 armas que tienes en el equipo, es decir si estas usando una AK y en el equipo tienes una UMP pasarías a tener equipada la UMP y el AK pasaría a estar en el equipo. Si solo tienes 1 arma no podrás cambiarla.
<b>SustituirArma</b>	Esta es la función que se encarga de sustituir el arma equipada por la arma comprada cuando compras un arma teniendo ya el equipo lleno.

<b>GetPosApuntado</b>	Devuelve una transformación que corresponde a la transformación de posición de apuntado del arma que se tiene equipada en ese momento.
<b>RestarPuntuacion</b>	Recibe como parámetro de entrada los puntos a restar, los resta a la puntuación del jugador y además llama al método <b>restaPuntos</b> del HUD para que aparezca por pantalla la animación de los puntos restados en rojo.
<b>SumarPuntuacion</b>	Recibe como parámetro de entrada los puntos a sumar, los suma a la puntuación total del jugador y además llama al método <b>sumaPuntos</b> del HUD para que aparezca por pantalla la animación de los puntos sumados en verde.
<b>ComprobarPuntos</b>	Recibe como parámetro de entrada los puntos necesarios para realizar una compra y tiene como parámetro de salida un booleano que será true en caso de que tengas los puntos necesarios o false cuando no tengas los puntos necesarios para hacer la compra.
<b>RestaVida</b>	Esta función se encarga de restar un punto de vida a la vida actual del jugador cada vez que recibe un impacto. Además tiene otras funciones importantes (como controlar si podemos revivir) que veremos en detalle abajo.
<b>SumaVida</b>	En esta función se llama de manera automática cada X tiempo siendo X el valor de la variable <b>tiempoRegeneracionVida</b> , sumando 1 punto de vida cada vez que se llama a la función. Además es la encargada de desactivar la llamada automática a esta función cuando la vida ha alcanzado su valor máximo.
<b>EnRanking?</b>	Función que devuelve un booleano el cual será true si tienes los puntos suficientes para entrar al ranking y falso si no has conseguido los puntos necesarios para entrar al ranking.
<b>ActualizarRanking</b>	Como su nombre indica esta función se encarga de actualizar el ranking, controlando en todo momento que si quedas 2do por ejemplo el que estaba 2do pase a ser el 3ero y tú te quedes en su lugar.
<b>SumarKill</b>	Función muy sencilla que simplemente aumenta el número de zombis matados de la variable <b>zombisMatados</b> .
<b>Revivo</b>	Su función es la de revivir al personaje una vez que ha muerto y había comprado el power-up de revivir (solo lo revive la

	primera vez que muere). Además esta función se encarga de deshabilitar el input mientras estas reviviendo para que no puedas moverte, volverlo a activar una vez que ya has revivido y de llamar a <b>PlayAnimationReviviendo</b> (función de HUD). Una vez que hemos revido nuestra vida estará al máximo.
<b>RegenerarVida</b>	CustomEvent que llama a la función <b>SumaVida</b> . La utilidad de este custom event es que podemos usar un <b>setTimerByFunctionName</b> sobre el para que se llame automáticamente cada cierto tiempo, haciendo así que la regeneración de vida sea automática.
<b>DejarDeApuntar</b>	CustomEvent para dejar de apuntar y todo lo que ello conlleva. El arma y los brazos pasarían a su posición por defecto, la velocidad de movimiento volvería a la normalidad (nos movemos más lentos mientras apuntamos) y el <b>FOV</b> de la cámara pasaría de 35° a 90° que sería su estado natural.
<b>DejarDeCorrer</b>	CustomEvent para dejar de correr y por tanto parar la animación de correr y volver a la velocidad de movimiento de andar. El cambio de animación de correr a la animación Idle se haría con un blending de 0.25 seg.
<b>EventAnyDamage</b>	Evento que se activa cuando recibimos cualquier tipo daño. Como parámetro de entrada tendríamos el daño que nos han causado, este daño se lo pasamos a la función <b>RestaVida</b> que es la que hace toda la lógica del restar vida. Este evento es el encargado de activar que se llame al generar vida cada cierto tiempo. (Lo veremos con más detalle abajo en el apartado 5.3.2.1.3.5 Recibir daño).
<b>EventBeginPlay</b>	Evento de Unreal que se llama cuando el juego comienza para ese actor. En este evento se inicializan cosas necesarias para el juego como por ejemplo se crea el actor de tipo partida que se encargara de toda la información de la partida. También se comprueba si existe algún archivo con un ranking guardado, si existe, lo carga para que podamos usarlo y sobrescribirlo. Si por el contrario no hay ningún archivo de guardado anterior entonces lo crea.
<b>InputAction Jump</b>	Llama al método de saltar de Unreal.

<b>InputAction CambioArma</b>	Este evento contiene la lógica del cambio de arma que se explicará abajo en detalle.
<b>InputAction Fire</b>	Cuando pulsamos el botón de disparar llamaríamos al <b>FireEvent</b> del arma equipada, pero antes de esto se comprobaría que no estamos cambiando de arma y se comprobaría si el arma es automática o no. Este evento también se explicara en detalle puesto que la lógica para el disparo de armas automáticas es necesaria explicar ayudándose de una foto para un mejor entendimiento.
<b>InputAction Recargar</b>	Llama al evento recargar del arma equipada.
<b>InputAction Sprint</b>	En este evento el personaje pasaría de estar andando a estar corriendo, para ello se dejaría de apuntar (si se estaba apuntando) ya que no puedes correr mientras apuntas, se dejaría de recargar (si se estaba recargando) ya que no se puede recargar y correr al mismo tiempo, se empezaría a reproducir la animación de correr y se cambiaría la velocidad máxima del jugador de 500 a 1000. Para añadir un poco de dificultad se han añadido una serie de limitaciones a esta mecánica: <ul style="list-style-type: none"> <li>• <b>No se puede apuntar mientras corres.</b> Si estas corriendo y apuntas automáticamente se dejaría de correr y volveríamos a la velocidad de andar. Por el contrario si estas apuntando y corres se dejaría de apuntar automáticamente para empezar a correr.</li> <li>• <b>No se puede recargar mientras corres.</b> Si estas corriendo y recargas se dejaría de correr automáticamente. Si estas recargando y empiezas a correr la recarga quedaría anulada como si nunca hubiéramos intentado recargar el arma.</li> <li>• <b>No se puede correr durante más de 3 segundos seguidos.</b> A los 3 segundos se para de correr automáticamente y empezaríamos a andar de nuevo.</li> </ul>
<b>InputAction Apuntar</b>	Es el evento que nos permite cambiar entre apuntar desde la cadera o apuntar con la mira del arma. Este evento se explicara en detalle en la sección de Funciones en detalle. (Véase apartado 5.3.2.1.3.1 Apuntar).

<b>InputAction Pause</b>	Evento que pone el juego en estado de pausa, parando por tanto toda la ejecución del juego como el comportamiento de los enemigos. Deshabilita el input del juego para que no podamos movernos y activa el input en modo de interfaz para poder interactuar con el menú de pausa. Crea el widget de pausa y lo añade en la pantalla
--------------------------	---

Figura 60: Tabla de funciones del personaje

### 5.3.2.1.3 Funciones importantes en detalle

Debido a que el número de funciones que hay en el Blueprint de nuestro personaje era bastante elevado hemos hablado un poco de ellas en el apartado anterior pero sin entrar en cómo está programada ni añadir imágenes de la lógica de las funciones. Es por esto que en este apartado vamos a tratar de explicar en detalle aquellas funciones que a mi parecer eran más importantes y de las que se añadirán imágenes.

Un dato a tener en cuenta es que las nodos de las imágenes de las funciones que veremos a continuación están bastante bien organizados, pero como habría que alejarse mucho para que apareciera toda la lógica esta sería ilegible, por lo tanto se han comprimido los nodos únicamente para que sean legibles en las imágenes.

#### 5.3.2.1.3.1 Apuntar

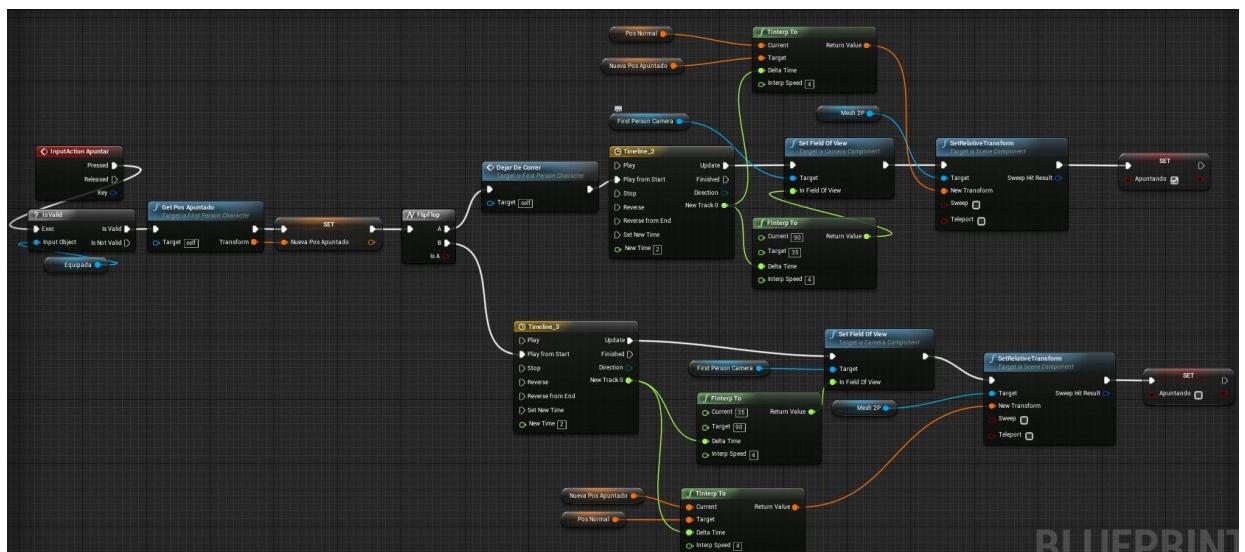


Figura 61: Función apuntar

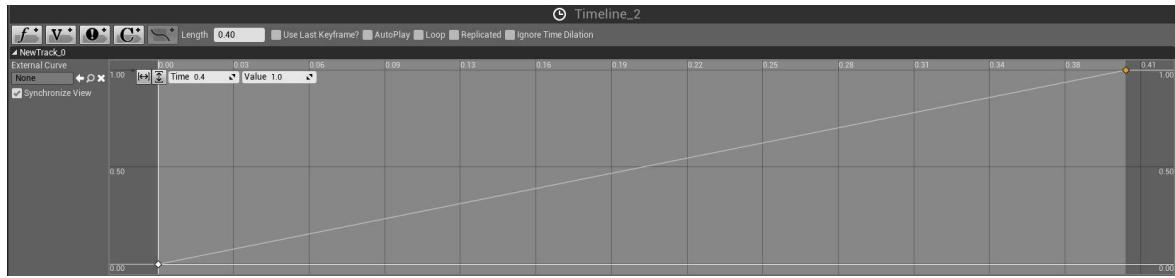
Lo primero que se hace en esta función es comprobar si el arma equipada es válida, que sería lo equivalente a comprobar que el arma equipada no fuera nula si estuviéramos programando en código. Esta comprobación es necesaria ya que el jugador puede pulsar el botón de apuntar incluso cuando aún no tiene arma (al principio de la partida). Si el arma equipada es válida se continua con la lógica de la función, si por el contrario no tuviéramos ningún arma equipada no se haría nada.

El segundo paso sería la llamada a la función **getPosApuntado** que como ya dijimos anteriormente nos devuelve una transformación que es la posición de apuntado del arma que tengamos equipada. Este valor nos lo guardamos en la variable **nuevaPosApuntado** que usaremos luego.

El siguiente paso es un nodo **Flip/Flop** este nodo actuaría como una especie de alternador. La primera vez que entramos a la función se ejecutaría la parte de arriba, la segunda vez la de abajo, la tercera la de arriba y así continuaría, es decir, va alternando entre las 2 salidas de ejecución. La ventaja de usar este Flip/Flop es que no necesitamos hacer comprobaciones de si estoy apuntando entonces dejar de apuntar, o si no estoy apuntando entonces apuntar ya que vamos alternando.

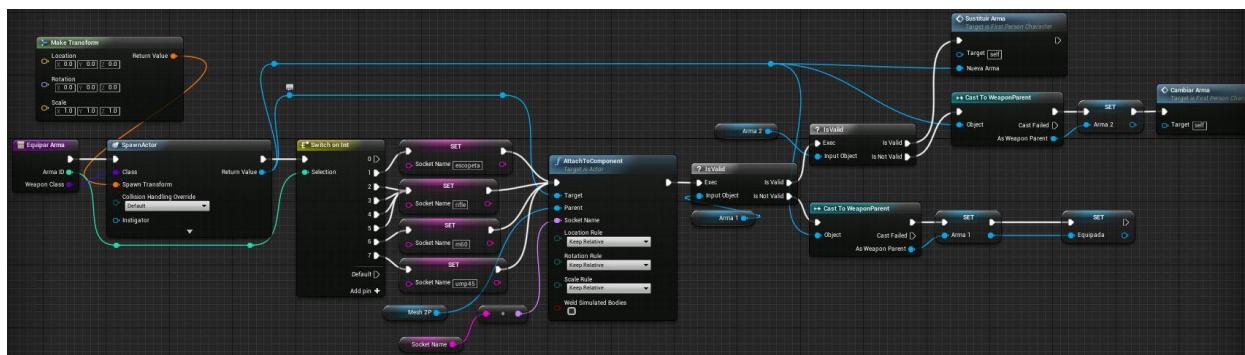
El nodo Flip/Flop tiene 2 salidas:

- **Salida A:** Cuando la ejecución continua por la primera salida del Flip/Flop entonces pasaríamos a estar apuntando. Para ello llamamos a la función dejar de correr, la cual detendría el sprint en caso de que estuviéramos corriendo. Seguidamente se haría una interpolación lineal usando el nodo de tipo **TimeLine** llamado **Timeline\_2** (véase figura TODO). Este nodo nos devuelve un valor dentro de una interpolación definida en su gráfica, ese valor se usa para obtener una transformación intermedia entre las transformaciones origen (**posNormal**) y destino (**nuevaPosApuntado**) que es la que obtuvimos antes como resultado del método **getPosApuntado**. Una vez tenemos esta transformación intermedia se pone como transformación relativa de la malla **Mesh2P** que son los brazos de nuestro personaje (el arma también seguiría el movimiento al estar conectada al socket de la mano). Al mismo tiempo que se está realizando la interpolación entre las 2 posiciones de apuntado también se realiza una interpolación del **FOV** (campo de visión) usando la misma interpolación definida por la gráfica de **Timeline\_2**. Por último se pondría la variable booleana **apuntando** a verdadera.
- **Salida B:** Cuando la ejecución continúa por la segunda salida del Flip/Flop entonces dejaríamos de apuntar para volver a la posición normal. Se haría una interpolación entre la posición de apuntado (ahora sería la transformación de origen) y la posición normal (ahora sería la transformación destino). Se haría también una interpolación en el **FOV** ahora desde 35° hasta 90° (su valor normal) y por último se pondría la variable **apuntando** a falsa.



La foto de arriba es la gráfica de la interpolación lineal que se hace entre las 2 transformaciones de posición de apuntado. Desde 0 (transformación origen) hasta 1 (transformación destino) en 0.4 segundos.

### 5.3.2.1.3.2 Equipar arma



A esta función se le llama cuando compramos un arma, es decir la llamaríamos desde la clase **WeaponDrop**. Cuando llamamos a esta función le pasamos 2 parámetros, un entero que es el ID del arma comprada y una clase de tipo actor que en este caso sería la Clase del arma comprada (AK, M4A1, UMP, etc.).

En el primer paso de esta función crearíamos un actor cuya clase sería la que nos pasaron por parámetro.

El segundo paso es el de vincular el arma (el actor que acabamos de crear) a los brazos del personaje, en un punto específico llamado **socket**. Para ello usamos el nodo **AttachToComponent** que recibe unos varios parámetros de entrada entre ellos están:

- **Target:** es el actor sobre el que se va a hacer el *attach to component*, es decir el actor que se va a unir a otro componente (una malla en este caso). Nuestro target es el arma que hemos creado.

- **Parent:** es el componente sobre el que se va a unir el actor del target. Este componente es nuestro **Mesh2P** que son los brazos del personaje.
- **Socket Name:** es el nombre del socket (es una especie de punto de guía para saber dónde tiene que colocarse el actor) al cual vamos a unir nuestro actor.

Es decir, haciendo un resumen de lo anterior lo que haríamos es unir la nueva arma (target) a los brazos del personaje (parent) en un punto específico de los brazos (socket). Hay 4 sockets ya que no todas las armas pueden estar en la misma posición debido a su tamaño o su forma. El socket es elegido con el nodo **SwitchOnInt** dependiendo del ID del arma. Hay 4 tipos de sockets (rifle, shotgun, m60, ump), por lo tanto si el arma comprada es una UMP45 se pondría en el socket “ump”, pero si es por ejemplo una AK47 o una G36 se unirían con el socket “rifle”.

Por ultimo nos quedaría ver en qué espacio del inventario se colocaría el arma comprada, hay 3 posibilidades:

- No tenemos ningún arma comprada. Se guardaría una referencia al arma comprada en la variable **Arma1**.
- Solo tenemos un arma comprada. Se guardaría una referencia al arma comprada en la variable **Arma2**.
- Tenemos 2 armas compradas y por tanto el inventario está lleno. Se llama al método sustituir arma que se encargaría de ver cuál es el arma que tenemos equipada y sustituir ese arma por la comprada. Es decir, si teníamos equipada el **Arma1** la referencia a la nueva arma comprada se guardaría en la variable **Arma1**. Si por el contrario teníamos equipada el **Arma2**, la referencia a la nueva arma comprada se guardaría ahora en la variable **Arma2**.

Hemos estado hablando de **sockets** pero no de cómo se crean, ya que estos sockets no aparecen por defecto. Estos puntos se pueden añadir en el esqueleto de la malla desde el editor de Unreal. Por ejemplo en la imagen siguiente se puede ver el *socket* utilizado para la posición de la escopeta, cuyo nombre es “shotgun”.



Figura 64: Ejemplo de socket en el editor de Unreal Engine 4.

A la izquierda se pueden ver los huesos de los que depende este *socket*. Esto significa que moviendo cualquiera de estos huesos también se movería el *socket*.

En el centro podemos ver visualmente donde se encuentra el *socket*.

A la derecha podemos ver información útil sobre el socket como el nombre del *socket*, el nombre de su hueso padre o su posición, rotación y escala.

#### 5.3.2.1.3.3 Cambiar de arma

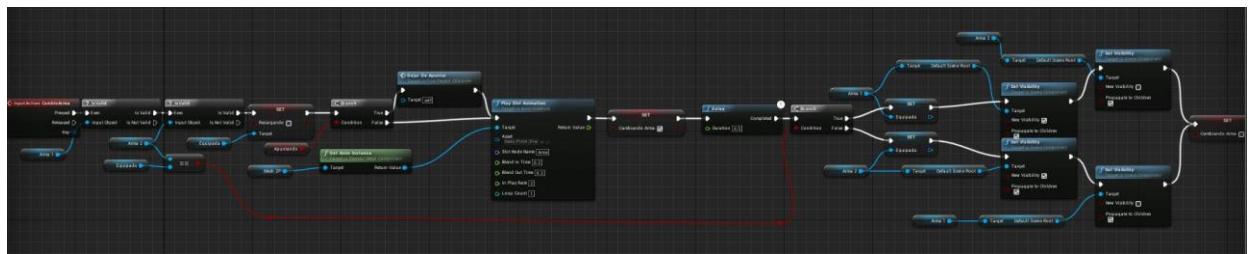


Figura 65: InputAction cambiar de arma

El cambio de arma no es una función sino un evento del tipo *InputAction* el cual se activa pulsando el botón triangulo (si jugamos con mando de ps4) o con la ruleta del ratón hacia arriba o hacia abajo. El porqué de la elección de un evento y no una función es sencillo, para esta función necesitaba un **delay** cosa que no permiten las funciones pero si los eventos.

El primer paso es comprobar que tenemos 2 armas en el inventario, ya que si no tenemos ninguna o solo tenemos una no vamos a poder cambiar de arma.

Una vez que ya hemos comprado que tenemos 2 armas en el inventario tenemos que ver cuál de esas dos es la que tenemos equipada. Teniendo el arma equipada ponemos la variable **recargando** a falso, si no estaba recargando ya estaba en falso con lo cual no cambiaría nada, pero si estaba recargando al poner esta variable a falso se anularía la recarga de ese arma. Seguidamente llamamos a la función **dejarDeApuntar** para que si estábamos apuntando el arma vuelva a su posición original antes de cambiarla.

El tercer paso en el que ya sabemos que no estamos recargando, ni estamos apuntando sería el de cambiar un arma por otra. Para ello reproducimos la animación de cambiar arma, ponemos la variable a **cambiandoArma** a verdadero y esperamos 0.5 segundos.

El último paso sería el de ocultar el arma que teníamos equipada, hacer visible la nueva arma que vamos a tener equipada (la que teníamos antes en el inventario) y volvemos a poner la variable **cambiandoArma** a falso puesto que ya hemos terminado el proceso de cambiar de arma.

El **delay** de 0.5 segundos usado justo después de reproducir la animación de cambiar de arma es para que el cambio de un arma por otra se produzca en la parte de la animación donde los brazos no se ven en la pantalla ocultando así el “efecto mágico” en el que un arma desaparece y aparece la otra.

### 5.3.2.1.3.4 Disparar

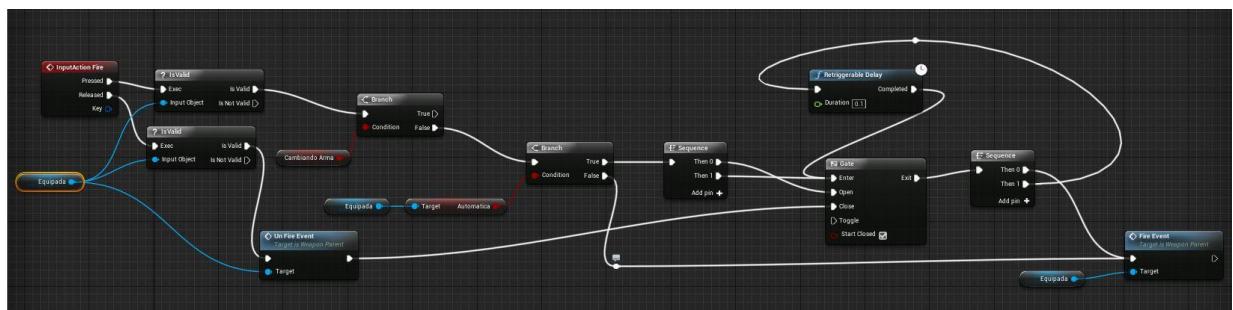


Figura 66: InputAction disparar

Cuando lo vemos por primera vez puede parecer un poco lioso pero es más sencillo de lo que parece. De nuevo, al igual que en el caso anterior, hemos elegido un evento en vez de una función debido a la necesidad de usar un **delay** en concreto un **retriggerableDelay**. Este evento tiene 2 acciones:

- En primer lugar cuando pulsamos el botón vinculado a este evento (gatillo derecho del mando o click izquierdo del raton) comprobaríamos si tenemos un arma equipada, ya que sin arma equipada no podemos disparar. Seguidamente comprobaríamos si estamos en mitad de un cambio de arma, si no estamos cambiando de arma seguiría la ejecución. Después comprobaríamos si el arma equipada es automática o no.
  - **Si no es automática:** se llama a la función **FireEvent** del arma equipada una sola vez.
  - **Si es automática:** se ejecutaría una secuencia, que en primer lugar abre una puerta (**Gate**) permitiendo así que el flujo de ejecución pase a través de ella. En segundo lugar se llamaría a otra secuencia que se encargaría de primero llamar a al **FireEvent** del arma equipada y segundo de volver a entrar por la puerta para que se vuelva de nuevo a esta segunda secuencia consiguiendo así un disparo

automático mientras la puerta este abierta o lo que es lo mismo mientras se mantenga el botón de disparar pulsado.

- En segundo lugar cuando soltamos el botón se llamaría a la función **UnFireEvent** del arma equipada que se encarga de hacer la lógica de dejar de disparar como desactivar el *muzzle*. Después de esto se cerraría la puerta (**Gate**) para que ya no pueda pasar el flujo de ejecución a través de esta (el flujo sigue en bucle desde que pulsamos el botón hasta que lo soltamos y se cierra la puerta permitiendo así el disparo automático).

### 5.3.2.1.3.5 Recibir daño

En este apartado vamos a hablar no solo de cuando nuestro personaje recibe daño, sino de cómo afecta esto en la vida del personaje, es decir, como disminuye o aumenta nuestra vida después de haber sido dañados.

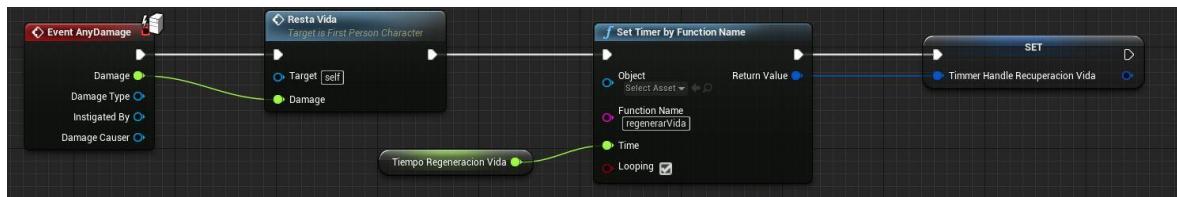


Figura 67: InputAction recibir daño

Aunque el evento **AnyDamage** es bastante sencillo como podemos ver en la foto de arriba, nos viene bien para explicar cómo se lleva a cabo la regeneración de vida de nuestro personaje.

Es importante saber que este evento es llamado automáticamente cada vez que nuestro jugador recibe daño de algún tipo, en el caso de nuestro juego el daño recibido será el del ataque de un zombi.

Lo primero que hace este evento es llamar a la función **RestaVida**

En segundo lugar se llamaría a la función **regenerarVida** pero esta función no se llama directamente, sino que se llama a través del nodo **SetTimerByFunctionName** el cual permite que la recuperación de vida sea automática sin tener que estar controlando nosotros llamar a la función

cada X tiempo para ir regenerando vida. El funcionamiento de **SetTimerByFunctionName** es el siguiente:

Podemos observar (véase figura TODO) que recibe 4 parámetros como entrada, estos 4 parámetros son:

- **Object:** es el objeto objetivo, es decir, el objeto sobre el cual tiene implementada la función a la que vamos a llamar. En este caso esta vacío, cuando esta vacío por defecto el objeto somos nosotros mismos, es decir, la función que vamos a llamar está en el mismo Blueprint desde el cual estamos usando el **SetTimmetByFunctionName**.
- **Function Name:** es el nombre de la función de la cual queremos ejecutar su lógica. En nuestro caso el nombre es “regenearVida”.
- **Time:** cuando llamamos a esta función no se ejecuta la lógica de la función **regenerarVida** inmediatamente, sino que se ejecutaría después del tiempo indicado en este parámetro, en nuestro caso el tiempo que hay que esperar para recuperar vida viene dado por la variable **tiempoRegeneracionVida** y por lo tanto se lo pasamos como parámetro de entrada en **Time**.
- **Looping:** si esta casilla está marcada la función indicada en **FunctionName** se llamará una vez detrás de otra con el intervalo de tiempo indicado en el parámetro **Time**, hasta que se indique lo contrario. Por el contrario si no está seleccionada la casilla **Looping** se llamará a la función una única vez.

Resumiendo lo anterior, y aplicándolo a nuestro caso, si suponemos que la variable **tiempoRegeneracionVida** tiene un valor de 2 entonces, una vez nos han quitado vida (hemos llamado a la función **restaVida**), entonces pasados los 2 segundos se ejecutaría la función **regenerarVida** y se volvería a llamar a esta función automáticamente cada 2 segundos ya que la casilla *looping* está marcada. De esta manera conseguimos que la regeneración de vida a través del tiempo sea automática teniendo que comprobar únicamente cuando hemos llegado al máximo de vida para poder desactivar la llamada automática de la función **regenerarVida** y no seguir sumándonos vida.

Por ultimo nos guardamos la salida del **SetTimerByFunctionName** en la variable **TimmerHandleRecuperacionVida** que es la que nos servirá luego para desactivar la llamada automática a esta función.

#### 5.3.2.1.3.5.1 Restar vida

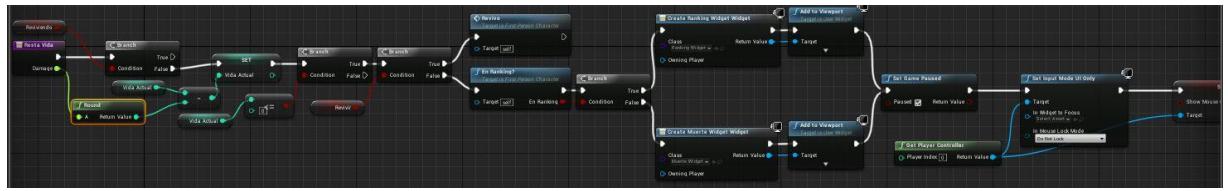


Figura 68: Función restar vida

Aunque esta función es sencilla he decidido explicarla porque en un principio se puede pensar que lo único que hace es restar un punto de nuestra vida actual, pero su funcionalidad es bastante mayor que esa.

En primer lugar comprobamos que no estamos en estado de revivir, ya que mientras estamos reviviendo no pueden hacernos daño.

En segundo lugar esta función tiene un parámetro de entrada que es el daño que nos han causado, pero este parámetro es de tipo *float* mientras que nuestra vida es un *int* por lo tanto tenemos que redondear este daño, ya que puede no ser un entero, dependiendo de la zona de impacto por ejemplo será mayor o menor, y nuestra vida sí que es un entero. Una vez ya tenemos el daño que nos han hecho convertido a un entero entonces restamos a nuestra vida actual este daño, y guardamos este nuevo valor de vida actual en la variable **vidaActual**.

A continuación comprobaríamos si esta vida es menor o igual a 0. Si la vida es mayor que 0 ya no continuaríamos con la ejecución, pero si la vida es menor o igual a 0 significa que hemos muerto por lo tanto comprobaríamos ahora si tenemos disponible el *power-up* de revivir. Si podemos revivir llamamos a la función **Revivo** que contiene la lógica necesaria para revivir a nuestro personaje. Si por el contrario no tenemos disponible la opción de revivir habríamos muerto definitivamente en esta partida.

Una vez que sabemos que hemos muerto entonces tendriámos que comprobar si tenemos los puntos suficientes o no para entrar en el ranking, de esto se encarga la función **EnRanking?**, la cual devuelve un booleano que:

- Si es true significa que hemos conseguido entrar en el ranking y por lo tanto aparecería la pantalla de que hemos entrado en el ranking permitiéndonos introducir un nombre.
- Si es falso aparecería la pantalla final de muerte.

Una vez estamos mostrando una de las dos pantallas disponibles la de muerte o la de ranking se pondría el juego en pausa para que no se ejecutara por debajo, se pondría el modo de input al de interfaz solo para poder clicar en los botones y escribir el nombre en caso de que estemos en la pantalla del ranking. Finalmente mostraríamos el cursor sobre la pantalla.

### 5.3.2.1.3.5.2 Regenerar vida

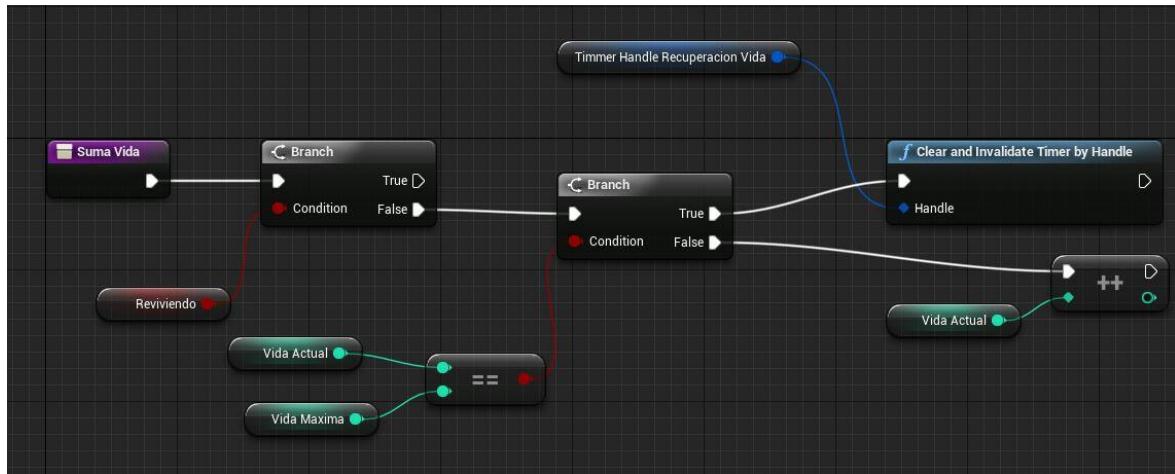


Figura 69: Función suma vida

El evento **regenerarVida** evento al que se llamaba con el nodo **setTimerByFunctionName** del que hemos hablado al principio de este apartado llama a la función **sumaVida** que es la que tiene toda la lógica.

Lo primero que hace esta función es comprobar si estamos reviviendo, ya que si estamos reviviendo no podemos sumarnos vida.

Si no estamos reviviendo comprobamos si la vida actual es igual a la vida máxima, es decir si ya hemos recuperado toda nuestra vida.

- Si hemos alcanzado la vida máxima usariámos la variable **TimmerHandlerRecuperacionVida** para poder parar la llamada automática a la función de regenerar vida. Esto lo haríamos con el nodo **ClearAndInvalidateTimerByHandle** al que le pasamos la variable de tipo **TimeHandle** encargada de llamar a la función **regenerarVida**.
- Si no hemos alcanzado la vida máxima entonces incrementamos nuestra vida actual en 1.

### 5.3.2.1.4 Animaciones

En este apartado vamos a hablar de las animación que tiene nuestro personaje, y como se han añadido estas animaciones al proyecto.

Por defecto cuando usas el Blueprint del FirstPersonCharacter este te viene con el modelo de brazos de Unreal Engine 4 y con algunas animaciones básicas como la animación de Idle, o la animación de saltar, pero estas animaciones no eran suficientes para este proyecto ya que las

animaciones del personaje quedarían un poco pobre por ejemplo sin una animación de recarga de arma.

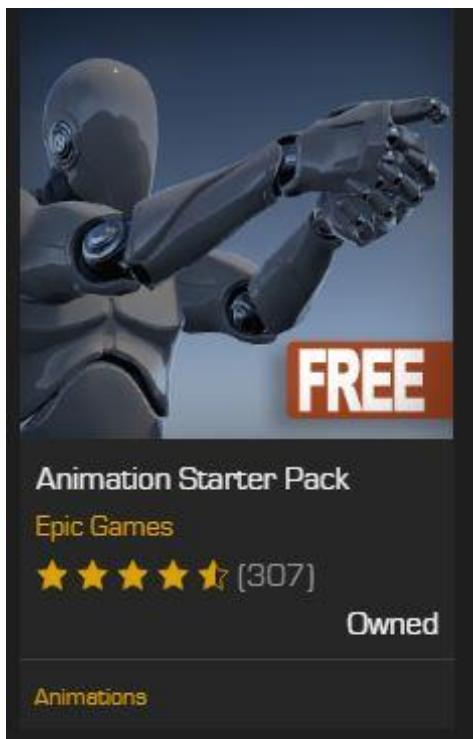


Figura 70: Animation Starter Pack de Unreal

Para conseguir un mayor número de animaciones recurrimos al bazar de Unreal Engine el cual tenía un pack totalmente gratis con animaciones sobre un juego FPS como por ejemplo la de disparar agachado, recargar una escopeta, etc.

El inconveniente de haber usado estas animaciones es que estaban preparadas para un juego en 3<sup>a</sup> persona, por lo tanto el esqueleto y la malla de estas animaciones eran un personaje completo y no solo los brazos de este como estamos usando en el proyecto.

Debido a esto he tenido que adaptar las animaciones del esqueleto en 3<sup>a</sup> persona al esqueleto en 1<sup>a</sup> persona que solo tiene los brazos.

Para esto hemos usado la característica de Unreal Engine 4 llamada **Animation Retargeting** (véase apartado 5.3.1.1.5.3 Animation Retargeting).

Antes de usar el Animation Retargeting hemos tenido que modificar ambos esqueletos, el origen (del starter pack de unreal) y el esqueleto destino (el esqueleto de nuestro personaje, es decir, los brazos) para que estos sean compatibles. Esto lo hemos hecho desde el **Skeleton Editor** estableciendo el tipo de **Rig a Humanoid** (recuadro rojo en la foto de abajo) y haciendo coincidir el nombre de los huesos de ambos (recuadro verde en la foto de abajo).

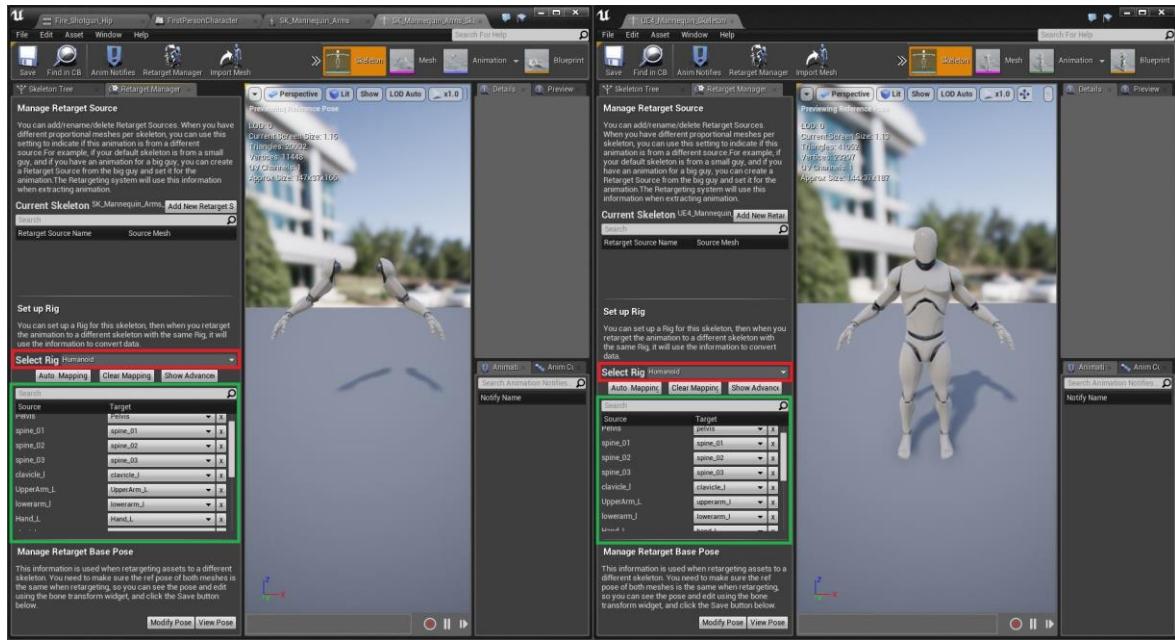


Figura 71: Retargeting Animation del proyecto

Al hacer esto conseguimos duplicar todas las animaciones que teníamos en el esqueleto de origen (derecha) al esqueleto destino (izquierda), teniendo por tanto las animaciones en 3<sup>a</sup> persona (arriba) y en 1<sup>a</sup> persona (abajo) como podemos ver en la siguiente imagen sacada del proyecto.

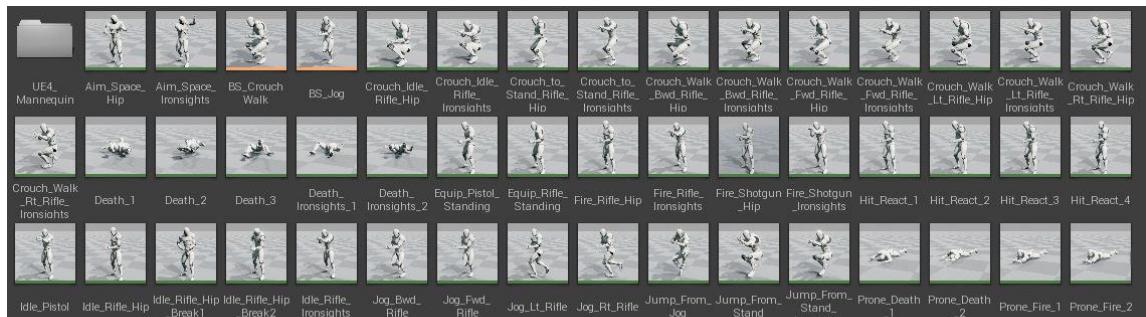
3<sup>a</sup> Persona → 1<sup>a</sup> persona

Figura 72: Animaciones en primera y tercera persona del proyecto

Una vez llegados a este paso ya tendríamos las animaciones listas para usar en el proyecto, pero en el proceso de adaptar animaciones de 3<sup>a</sup> a 1<sup>a</sup> persona estas aparecieron rotadas respecto al hueso central llamado **Root**.

Intente solucionar esto de varias formas como rotando este hueso en las animaciones que aparecían rotadas lo cual no funciono porque al ser el hueso central todos dependen de él y por lo tanto las animaciones originales como Idle y Jump estarían ahora mal rotadas al arreglar la rotación de las nuevas.

Luego intente hacer un **Animation Montaje** de cada animación mal rotada para intentar modificar en esta la rotación de los huesos pero seguimos teniendo el mismo problema.

Por ultimo decidí retocar las animaciones en el propio editor de animaciones añadiendo varios *key frames* de rotación para que aparecieran luego bien rotadas, lo que resultó ser un trabajo muy costoso pues al rotar las animaciones surgieron varios problemas, como por ejemplo que a mitad de animación el brazo avanzaba mucho hacia delante por la propia animación haciendo que el hombro del personaje ocupara la mitad de la pantalla al entrar en el cuadro de la cámara.

Por lo tanto tuve que ir retocando animación a animación viendo en que frames se iba la animación de la cámara o en cuales la atravesaba y ocupaba casi la totalidad de la pantalla. Se puede ver un ejemplo de esto en la siguiente foto donde se ven las gráficas modificadas para conseguir que la animación funcione correctamente.

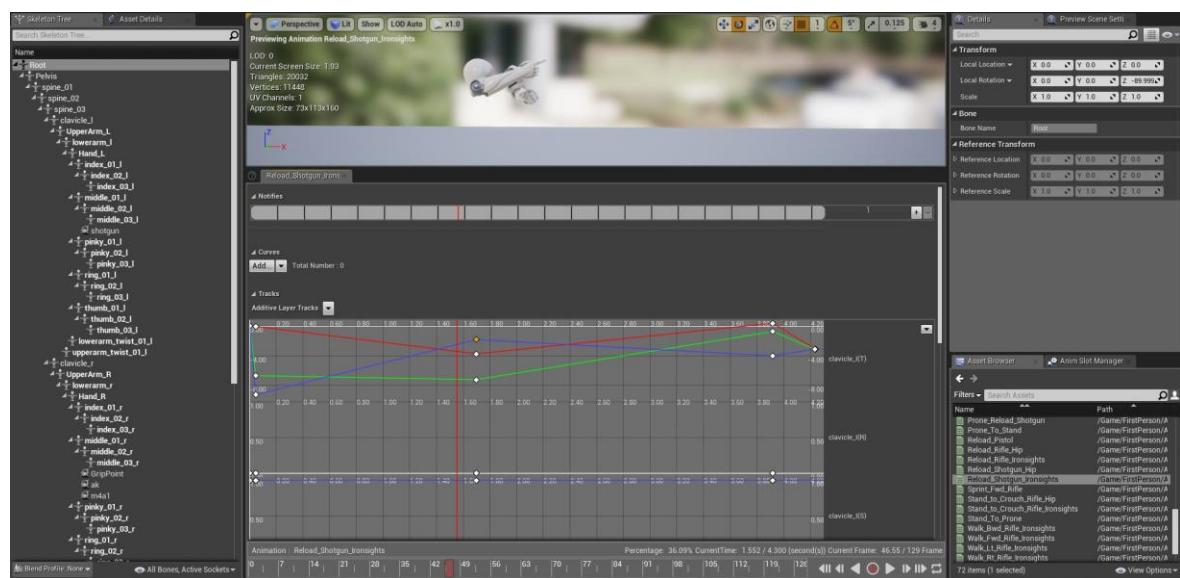


Figura 73: Corrigiendo animación desde el editor de Unreal

Finalmente cuando ya tenemos todas las animaciones listas para ser utilizadas nos toca crear un **Animation Blueprint** encargado de reproducir la animación correcta en cada situación.

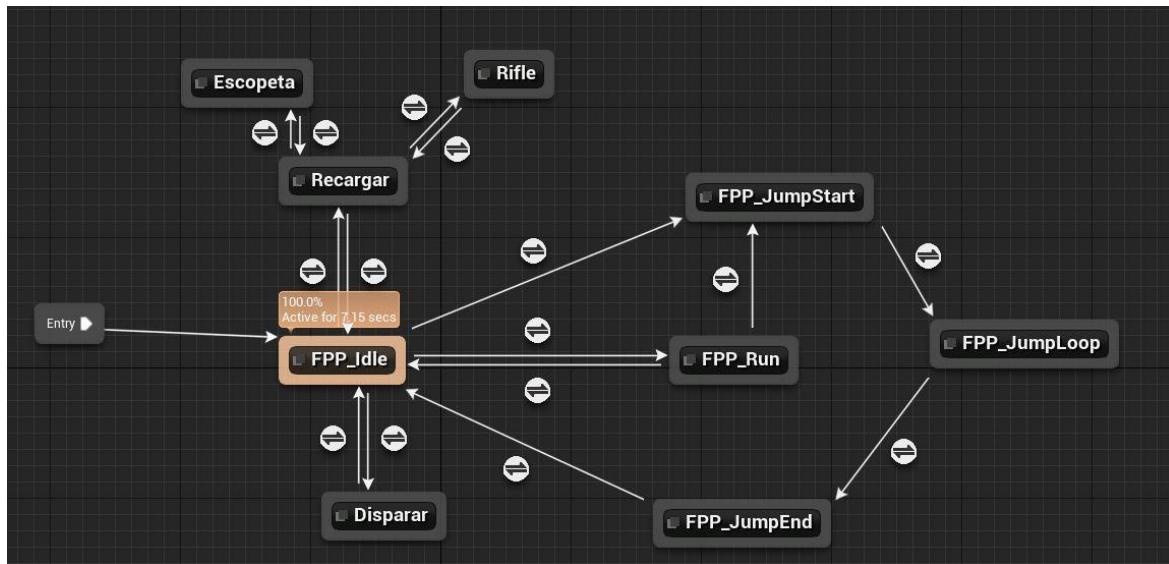


Figura 74: Animation Blueprint personaje

La animación inicial es la de Idle, la cual se reproduce en bucle siempre que el personaje está parado.

Si disparamos se reproduciría la animación de disparar que es un retroceso del arma. Como se puede observar en el grafico no hay una conexión directa entre la animación de correr (**FPP\_Run**) y la animación de disparar, esto significa que no podemos estar corriendo y disparando a la vez, sino que primero nos pararíamos (pasando por la animación idle) y ya disparamos.

Cuando recargamos comprobamos si tenemos la escopeta o un rifle, dependiendo del arma se reproducirá una animación de recarga o la otra.

Cuando saltamos se reproduce la animación **FPP\_JumpStart** que es el inicio del salto, luego pasariamos a reproducir en bucle la animación **FPP\_JumpLoop** que es la animación de nuestro personaje cuando se encuentra en el aire después de un salto. Esta animación se reproduce en bucle hasta que entramos en contacto con el suelo que se reproducirá la animación **FPP\_JumpEnd** y después de esta volveríamos al estado normal (**FPP\_Idle**).

### 5.3.2.2 Armas

En este apartado vamos a explicar la lógica de las armas del juego, como por ejemplo como se ha hecho para poder comprar armas de la pared y equiparla en nuestro inventario.

También se explicara cómo funcionan las armas y algunos aspectos importantes de estas como el disparo, recarga, etc.

### 5.3.2.2.1 WeaponDrop Blueprint (armas en la pared)

Todas las armas que están colgadas en la pared y que podemos comprar heredan de la clase **WeaponParentDrop**. Este Blueprint es el que contiene toda la lógica necesaria para comprar el arma.

#### 5.3.2.2.1.1 Componentes

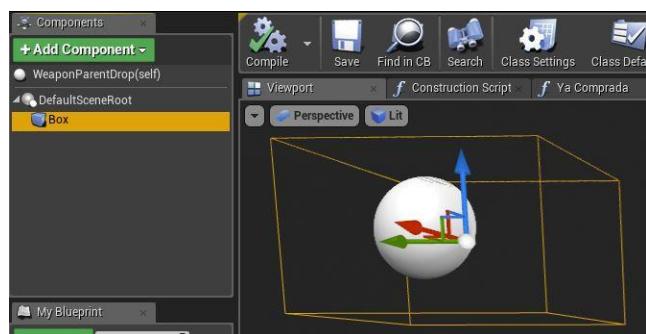


Figura 75: Componentes WeaponParentDrop Blueprint

El único componente que tiene esta clase padre es la caja de colisión (**Box Collision**) que es la que nos va a servir para saber si estamos cerca del arma y por tanto podremos comprar el arma o no.

#### 5.3.2.2.1.2 Variables

WeaponParentDrop Variables		
Nombre	Tipo	Descripción
<b>WeaponID</b>	Int	Número con el ID del arma, este ID se usa luego para saber qué clase de arma tienes que crear, por ejemplo si compramos arma con ID 1 creamos un actor cuya clase es <b>AKDrop</b> .
<b>Comprada</b>	Boolean	Variable booleana que es el resultado de la función <b>YaComprada</b> . Si es true es que ya hemos comprado el arma y si es falso es que aún no hemos comprado ese arma con la que estamos interactuando (hemos entrado en la zona de compra).

<b>precioArma</b>	Int	Es el precio que tenemos que pagar para comprar esa arma.
<b>precioMunicion</b>	Int	Es el precio que tenemos que pagar para comprar munición de esa arma.
<b>Info</b>	String	Es un string con información sobre el arma, la cual se muestra luego por pantalla cuando estamos en zona de compra. Esta información es el nombre y el precio del arma.
<b>InfoAmmo</b>	String	Es un string con información sobre la munición del arma con la que interactuamos y su precio.

Figura 76: Table variables WeaponParentDrop

### 5.3.2.2.1.3 Funciones

Como el número de funciones de esta clase no es muy elevado no vamos a hacer una tabla de estas, sino que vamos a explicarlas directamente.

#### 5.3.2.2.1.3.1 ConstructionScript (Constructor de clase)

El constructor de esta clase está vacío. Como es una clase padre de la que heredan todas las armas que podemos comprar la importancia de inicializar la clase recae sobre el constructor de la clase hijo que sabe los atributos específicos de la clase de arma.

Los constructores de las clases hijas como pueden ser **AKDrop** o **UMPDrop** serían los encargados de inicializar los valores del arma (para comprar) como se muestra a continuación.

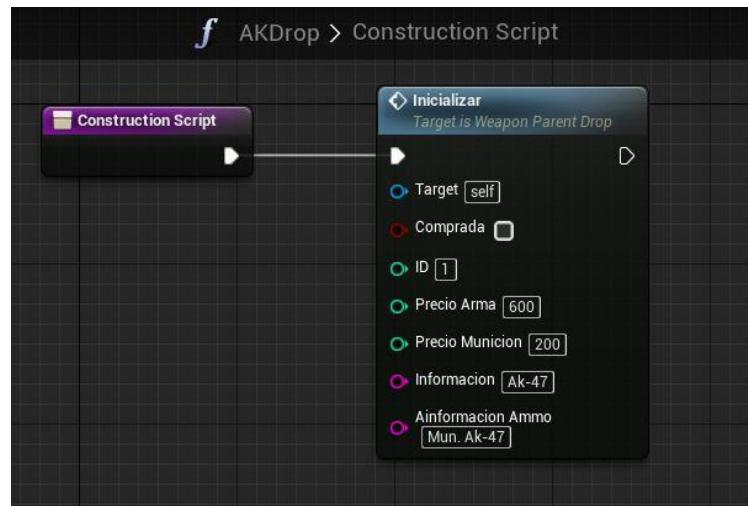


Figura 77: Constructor AKDrop

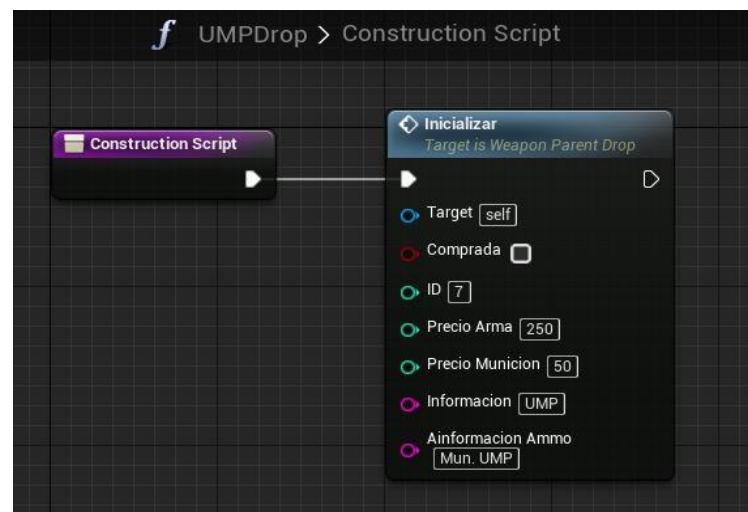


Figura 78: Constructor UMPDrop

### 5.3.2.2.1.3.2 Inicializar

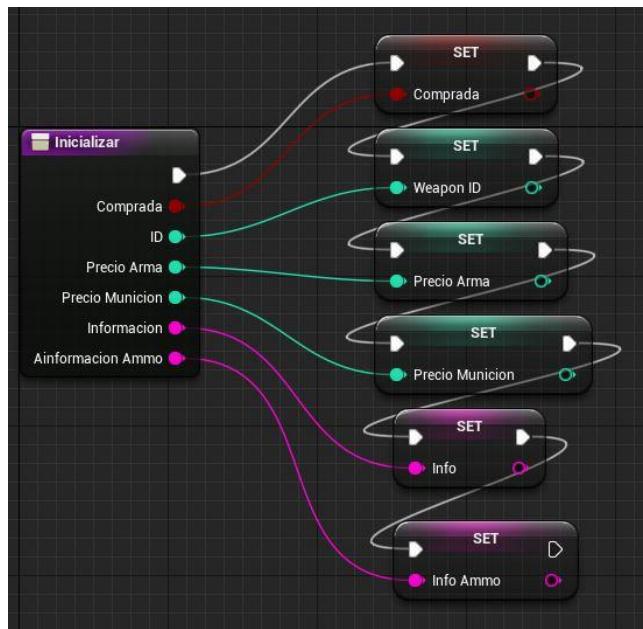


Figura 79: Función inicializar WeaponParentDrop

Esta es la función a la que llaman los constructores de las clases hijas. Su función es bien sencilla, es la de establecer los valores de las variables. La ventaja de esto es que ahora podemos tratar a todas las armas como **WeaponParentDrop** sin importar que tipo de arma sea. Por ejemplo a la hora de mostrar la información cuando estas cerca de un arma no tenemos que mirar en que arma estamos, sino simplemente llamamos a la función que muestra información de la clase padre y esta muestra la información de la variable **info** sin importar que arma sea, habiendo sido esta información establecida correctamente por el constructor de la clase hija al llamar a esta función **Iniciar**.

### 5.3.2.2.1.3.3 Entrar en zona de compra

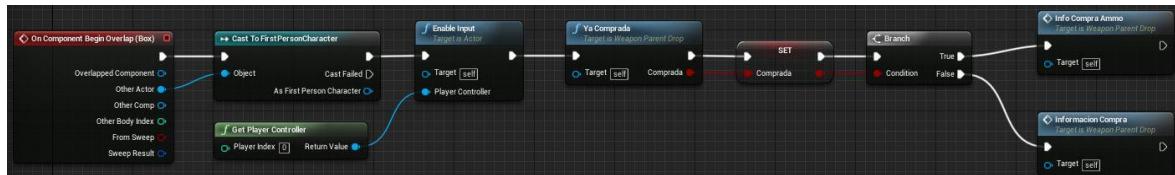


Figura 80: WeaponParentDrop evento entrar en zona de compra

Este evento se encarga de controlar si estamos dentro de la zona de compra de un arma y por tanto podremos interactuar con ella. Para ello detecta si el actor ha entrado dentro de la zona de la **BoxCollision** que tenía como componente. Cuando el player entra en esta zona entonces se ejecuta este evento que se encarga de habilitar el input para esta clase, permitiendo recoger entradas del teclado y que se utilicen eventos de tipo InputAction en este Blueprint.

En segundo lugar comprobaría si el arma ya está comprada llamando a la función **YaComprada** que devuelve un booleano que se almacena en la variable **comprada**.

Si esta variable es verdadera significa que el arma ya ha sido comprada y se llama a la función **InfoCompraAmmo** que se encarga de que aparezca un mensaje con la información sobre la compra de munición.



Figura 81: Mostrando información compra munición arma

Si **comprada** es falso entonces se llamaría a la función **InformacionCompra** que se encarga de mostrar el mensaje en la pantalla con la información de la compra del arma.



Figura 82: Mostrando información compra arma

#### 5.3.2.2.1.3.4 Salir de zona de compra

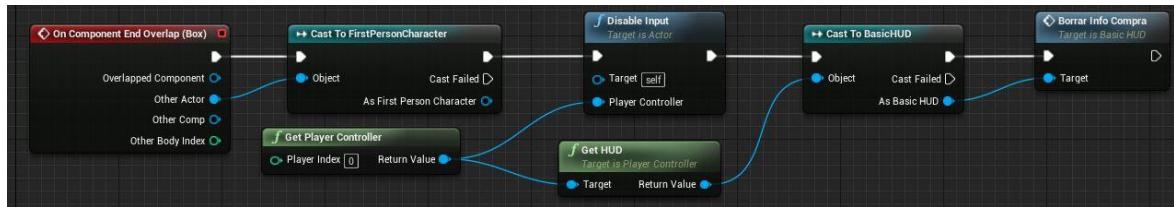


Figura 83: WeaponParentDrop evento salir de zona de compra

Cuando el jugador sale de la zona de compra o lo que es lo mismo sale de la **BoxCollision** del **WeaponDrop** entonces se llama a este evento que se encarga de deshabilitar el input en esta clase para que aunque estés pulsando la F (es el botón con el que se compra) no se llame al evento **InputAction Interactuar** ya impidiendo así la compra del arma si estas fuera de la zona.

Por ultimo llama a una función del HUD llamada **BorrarInfoCompra** que se encarga de borrar la información de compra de la pantalla.

#### 5.3.2.2.1.3.5 Interactuar

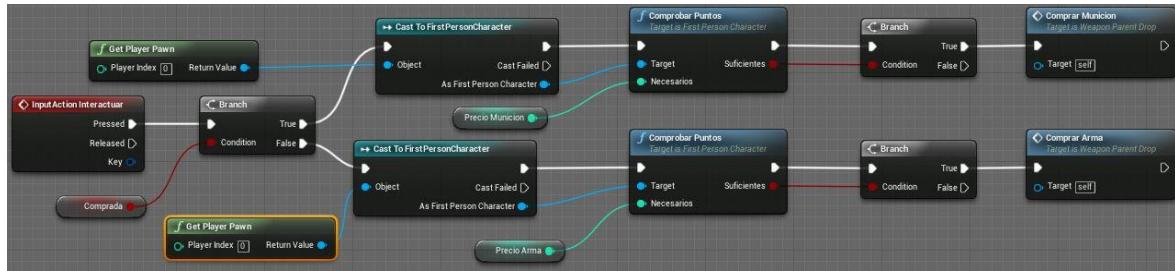


Figura 84: InputAction Interactuar WeaponDrop

Este evento se llama cuando pulsamos la tecla vinculada al evento **Interactuar** (la tecla F del teclado o el botón circulo de un mando de PS4), aunque solo se llamará si estamos dentro de la zona de compra del arma, si estamos fuera aunque pulsemos la tecla de interactuar no pasará nada.

Lo primero que hace este evento es comprobar si el arma con la que estamos interactuando ya está comprada o no, para saber si tenemos que comprar ese arma o solo su munición.

- Si el arma está comprada se llama a la función **ComprobarPuntos** que devuelve un booleano que será true si el jugador tiene los puntos necesarios para comprar el arma y false en caso de que no tenga los puntos necesarios. Si tenemos los puntos necesarios entonces podremos comprar la munición, para ello llamamos a la función **ComprarMunicion**.
- Si el arma no está comprada y tenemos el dinero necesario para comprarla se llama a la función **ComprarArma**.

### 5.3.2.2.1.3.6 Comprar arma

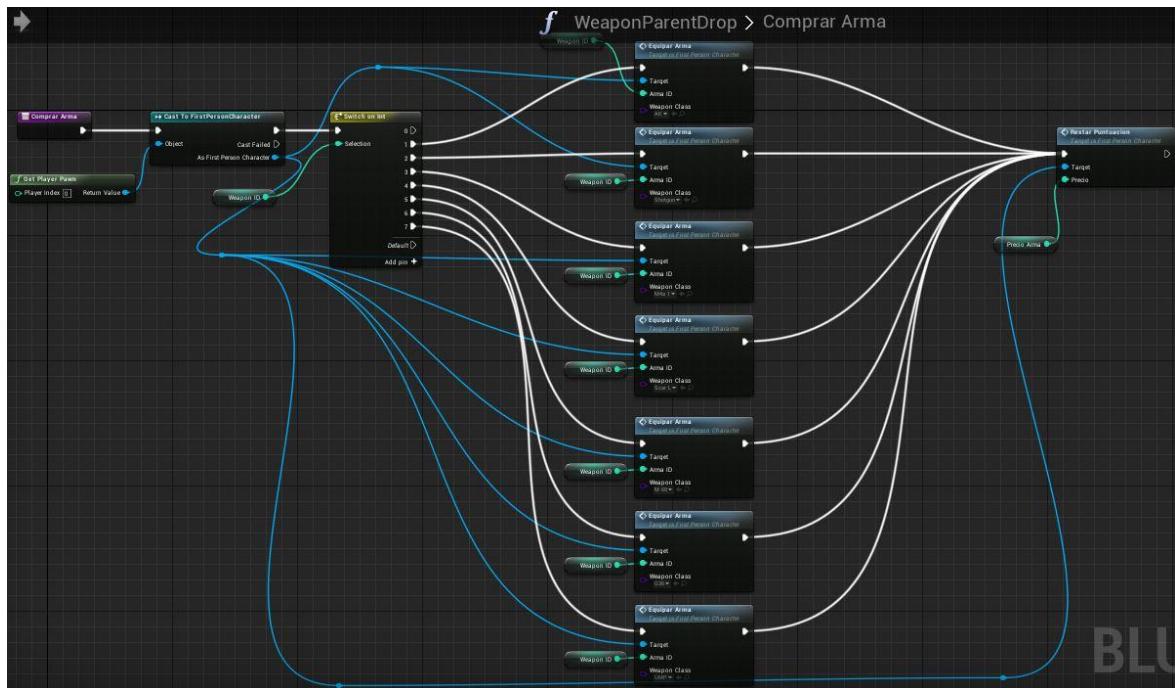


Figura 85: Función comprar arma

Aunque en un primer vistazo parezca compleja esta función es bastante sencilla. En primer lugar cogemos al FirstPersonCharacter para poder llamar a su función **EquiparArma**. Esta función equipar arma necesita recibir como parámetro de entrada el ID del arma y una clase para poder crear luego un actor de esa clase (véase apartado 5.3.2.1.3.2 Equipar arma).

Para pasarlo la clase del arma correcta lo que se hace antes de llamar a la función **EquiparArma** es hacer un **switch** a partir del ID del arma con la que estamos interactuando, y dependiendo de su valor el arma será de una clase u otra. Por ejemplo un arma con ID 3 será una M4A1 mientras que un arma con ID 7 será una UMP.

Por último después de llamar al **EquiparArma** se llama a la función **RestarPuntuacion** del player pasándole el precio del arma que acabamos de comprar.

### 5.3.2.2.1.3.7 Comprar munición

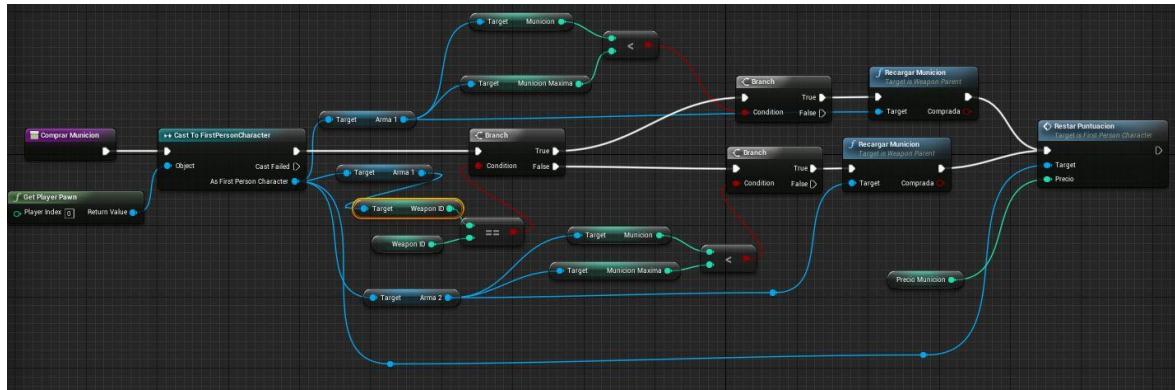


Figura 86: Función comprar munición

Lo primero que hace esta función es comprobar sobre que arma vas a comprar munición, ya que cuando te acercas a comprar munición no tienes por qué tener el arma equipada. Entonces lo primero sería comprobar si el arma del que vamos a comprar munición es la referencia que tenemos en Arma1 o en Arma2 para poder sumar la munición al arma correcta. Esto lo comprobamos con los ID, es decir si el ID del Arma1 coincide con el ID del arma con la que estamos almacenando tenemos que llamar a la función **recargarMunicion** del Arma1, si esos ID no coinciden tenemos que llamar a la función de **recargarMunicion** del Arma2, ya cuando estamos en esta función sabemos seguro que tenemos el arma con la que estamos interactuando en una de las 2 variables.

Una vez que ya sabemos sobre que arma tenemos que trabajar entonces comprobaríamos la munición de esta arma, ya que si la munición está al máximo no se podrá comprar más munición.

Si la munición no está al máximo entonces se llama a la función **recargarMunicion** y seguidamente a la función **RestarPuntuacion** pasándole el precio de la munición que acabamos de comprar.

### 5.3.2.2.1.3.8 Comprobar si ya hemos comprado el arma

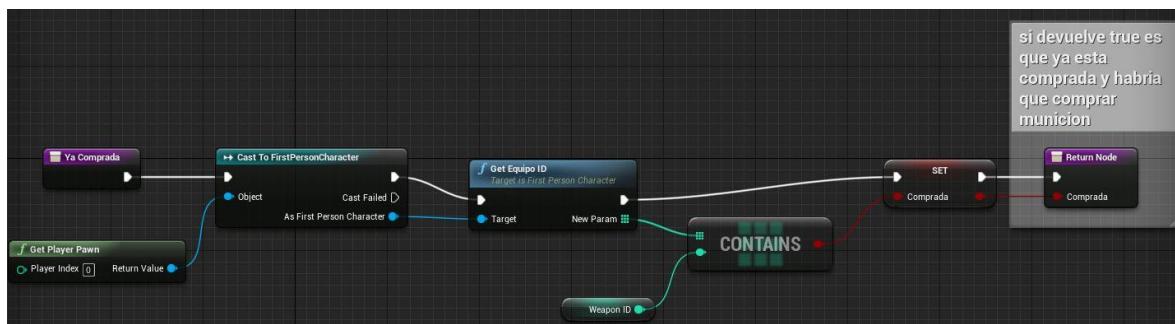


Figura 87: Función que comprueba si ya hemos comprado un arma

Esta función devuelve un booleano que nos dirá si el arma ya ha sido comprada o aún no. Para ello lo que hace es llamar a la función **getEquipmentID** del player, que nos devuelve un array con los ID de las armas que tenemos en el equipo, entonces ahora comprobamos si el ID del arma con la que estamos interactuando esta en este array.

Si esta en el array entonces el arma ya ha sido comprada ya que la tenemos en el equipo, pondremos la variable **comprada** a verdadero y la devolveremos por la función.

Si el ID del arma con la que interactuamos no está en el array del equipo del player significará que no hemos comprado aun ese arma y por tanto establecemos la variable **comprada** a falso y devolveremos valso en la función.

### 5.3.2.2.1.3.9 Información compra arma y munición



Figura 88: Función información compra (izquierda) y función información compra munición (derecha)

Estas 2 funciones las vamos a explicar juntas ya que su funcionalidad es la misma. Pero cambiando la información que se pasa.

Estas funciones llaman a la función del HUD **ActualizaInfoCompra** y le pasan la información de compra en caso de que el arma no esté comprada ya o la información de compra de munición en caso de que ya hayamos comprado el arma.

### 5.3.2.2.2 Weapon Blueprint (arma equipada)

En este apartado va a explicar el Blueprint de la clase arma llamado **WeaponParent** que es el Blueprint del cual heredan todas las armas que podemos utilizar. Muchas de las funciones de este Blueprint no se implementan en la clase padre sino que son sobrescritas en las clases hijas debido a que tienen distintos comportamientos dependiendo del arma, cuando esto ocurra se cogerá de ejemplo la clase **AK** para mostrar un ejemplo de estas funciones en una clase hija.

#### 5.3.2.2.2.1 Interfaz (Weapon Interface)

Todas las armas implementan una interfaz llamada **WeaponInterface** que contiene los métodos necesarios que tienen que ser implementados por cada arma, además de otras funciones a las que luego se les puede enviar un mensaje desde otro Blueprint gracias a esta interfaz como es el caso del evento **Shoot**, el cual se llama median un mensaje a través de la interfaz desde la clase **WeaponParent** ejecutándose así la lógica del **Shoot** del arma que reciba el mensaje que será aquella con la que hemos disparado.

### 5.3.2.2.2 Variables

WeaponParent Variables		
Nombre	Tipo	Descripción
<b>WeaponID</b>	Int	ID del arma, que nos sirve para identificar la clase del arma.
<b>MunicionCargador</b>	Int	Es la munición actual que hay en el cargador del arma.
<b>CapacidadCargador</b>	Int	Cantidad de balas que tiene un cargador, así a la hora de recargar cada arma se recarga en función de su capacidad del cargador, por ejemplo la m60 tendría 100 balas en el cargador mientras que la UMP tendría 30.
<b>MunicionMaxima</b>	Int	Es la munición máxima que se puede tener (fuera del cargador).
<b>Municion</b>	Int	Es la munición actual que disponemos para recargar el arma sin tener en cuenta la munición que ya hay dentro del cargador.
<b>Recargando</b>	Boolean	Variable booleana que nos dice si estamos recargando el arma o no.
<b>Disparando</b>	Boolean	Variable booleana que nos dice si estamos disparando o no.
<b>Automatica</b>	Boolean	Verdadera en caso de que el arma sea automática y falsa cuando no lo es.
<b>CanShoot</b>	Boolean	Esta variable nos dice si estamos preparados para realizar un disparo. Si es

		verdadera podremos disparar, mientras que si es falsa no podremos.
<b>Damage</b>	Float	El daño que causa un impacto del arma.
<b>Cadencia</b>	Float	Es el tiempo que tiene que pasar entre disparo y disparo.
<b>TiempoRecarga</b>	Float	El tiempo que tarda un arma en recargarse.
<b>NumMejorasCadencias</b>	Int	Indica el número de veces que se ha mejorado la cadencia del arma (en una máquina de mejorar cadencia).
<b>NumMejorasRecarga</b>	Int	Indica el número de veces que se ha mejorado el tiempo de recarga del arma (en una máquina de mejorar tiempo de recarga).
<b>NumMejorasCargador</b>	Int	Indica el número de veces que se ha mejorado la capacidad del cargador del arma (en una máquina de mejorar capacidad del cargador).
<b>NumMejorasMunicion</b>	Int	Indica el número de veces que se ha mejorado la capacidad de la munición máxima del arma (en una máquina de mejorar munición maxima).
<b>NumMejorasDamage</b>	Int	Indica el número de veces que se ha mejorado el daño del arma (en una máquina de mejorar daño).
<b>CentrarBala</b>	Vector	Vector que se suma a la posición de la bala cuando se dispara sin apuntar para que esta salga centrada respecto a la pantalla.
<b>CentrarBalaApuntando</b>	Vector	Vector que se suma a la posición de la bala cuando se dispara apuntando para que esta salga centrada respecto a la pantalla.

Figura 89: Tabla variables WeaponParent Blueprint

### 5.3.2.2.2.3 Funciones

#### 5.3.2.2.2.3.1 Constructor (Construction Script)

El constructor de esta clase está vacío. Como es una clase padre de la que heredan todas las armas que podemos utilizar en el juego la importancia de inicializar la clase recae sobre el constructor de las clases hijas que sabe los atributos específicos de la clase de arma que está inicializando.

Los constructores de las clases hijas como pueden ser **AK** o **M60** serían los encargados de inicializar los valores del arma como se muestra a continuación.

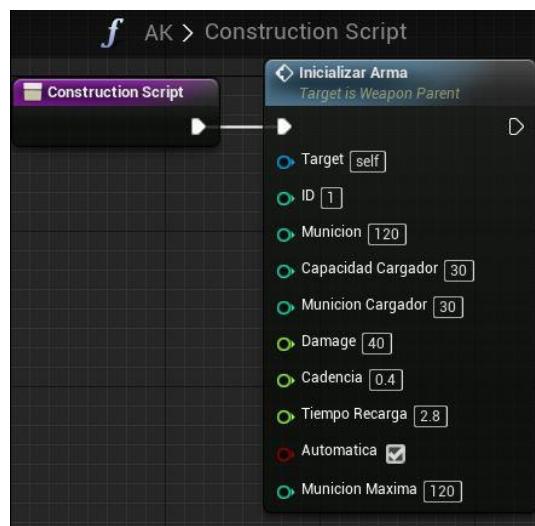


Figura 90: Constructor AK

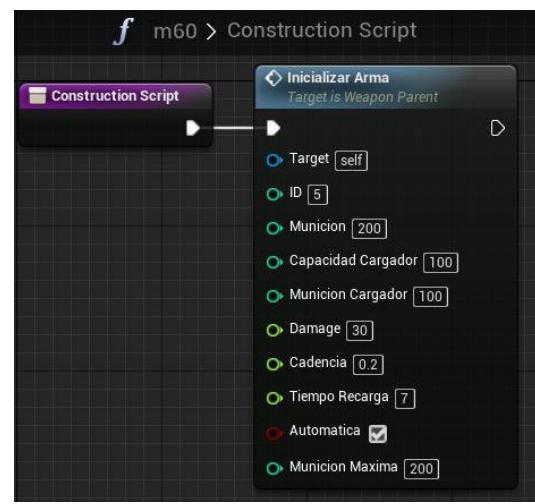


Figura 91: Constructor M60

#### 5.3.2.2.2.3.2 Inicializar

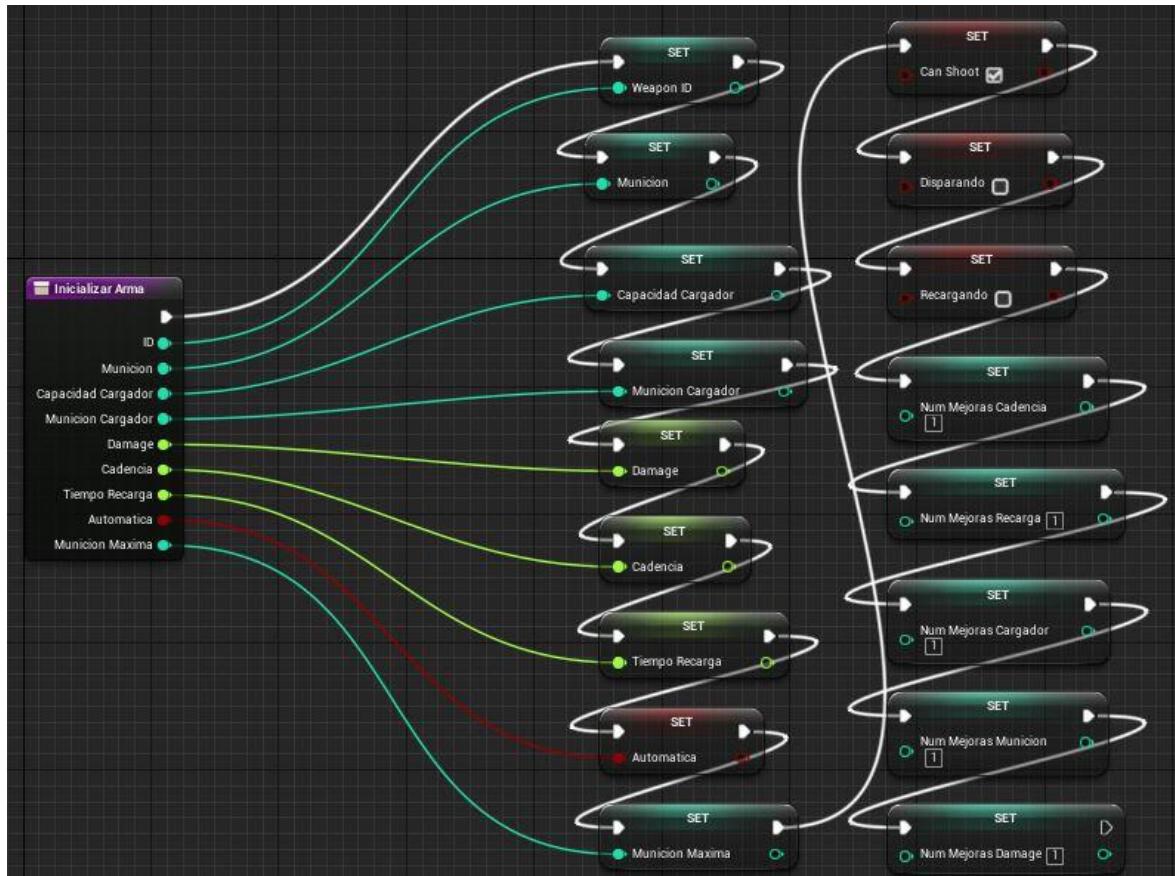


Figura 92: Función inicializar arma

Esta es la función a la que llaman los constructores de las clases hijas. Se encarga de establecer las estadísticas de cada arma. La ventaja de esto es que ahora podemos tratar a todas las armas como **WeaponParent** sin importar que tipo de arma sea. De esta forma lo que guardamos en las variables **Arma1**, **Arma2** y **Equipada** del player son **WeaponParent** pudiendo intercambiar de armas, recargarlas, disparar, etc. Sin la necesidad de saber el tipo de arma que tenemos pues tanto si tenemos una M60 o tenemos una UMP llamamos al método disparar de la clase padre.

### 5.3.2.2.2.3.3 Disparar

Esta función de disparar se divide en varias, además una parte común se hace la clase padre **WeaponParent** y luego otra parte se hace en cada arma ya que es distinta de un arma a otra.

#### 5.3.2.2.2.3.3.1 InputAction Fire

En primer lugar empezamos por el evento **InputAction Fire** que es el que se ejecuta cada vez que pulsamos el botón de disparar.

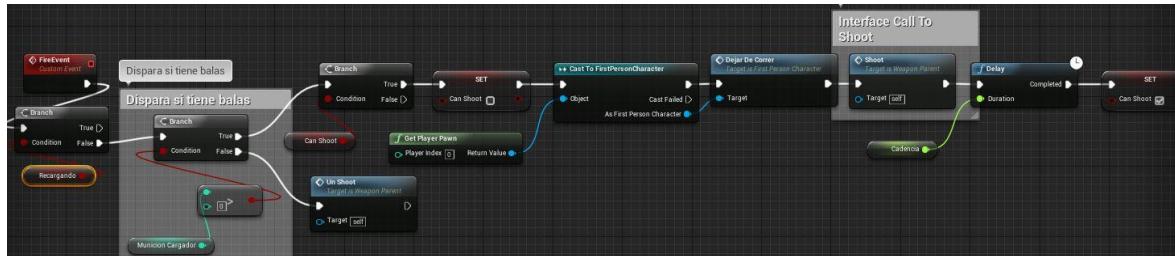


Figura 93: Input Action Fire (evento disparar)

Lo primero que hace este evento es comprobar si estamos recargando, ya que si estamos recargando no podemos disparar. Si no estamos recargando entonces comprobamos si la munición que tenemos en el cargador es mayor que 0, es decir, si nos quedan balas en el cargador con las que disparar. Si no quedan balas se llama el evento **UnShoot**.

Si tenemos balas en el cargador comprobamos ahora que podemos disparar, esta condición viene dada por la variable **CanShoot**, y si podemos disparar entonces cambiamos esta variable a falso ya que estamos en mitad de un disparo y se hará true cuando pase el tiempo necesario entre disparo y disparo.

Una vez que sabemos que podemos disparar llamamos a la función **DejarDeCorrer** ya que no podemos disparar mientras se corre. Seguidamente llamamos al evento **Shoot** del arma, esto lo hacemos gracias a la **WeaponInterface** que implementan todas las armas, así cada arma ahora ejecutará su lógica de disparo. Veremos este evento en detalle en el apartado siguiente.

Esperamos el tiempo necesario para realizar otro disparo que viene dado por la cadencia del arma y entonces ponemos la variable **CanShoot** a verdadero para que la próxima vez que se llame a este evento **Fire** se pueda disparar. Hay que recordar que si el arma es automática a este evento se está llamando continuamente mientras pulsamos el botón de disparar, por eso es necesario esta variable **CanShoot** para que aunque entremos 10 veces en este evento disparemos con la cadencia establecida por el arma.

### 5.3.2.2.3.3.2 Event Shoot

Este evento es llamado mediante un mensaje desde el **InputAction Fire** gracias a la implementación de la interfaz común **WeaponInterface** y pertenece a las clases hijas ya que cada una llamará a sus funciones para disparar. En este ejemplo vemos el de la clase **AK**.

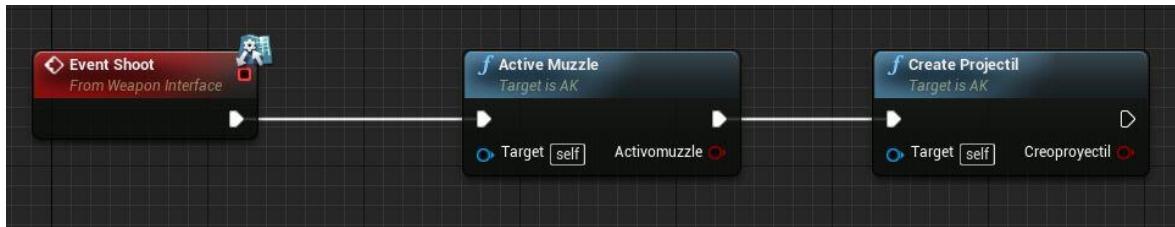


Figura 94: Event Shoot de la clase AK

Este evento únicamente llama a 2 funciones cuya lógica es diferente en cada arma, una es la de **ActiveMuzzle** que se encarga de activar el destello del arma cuando corresponda, y la otra es la función **CreateProjectile** que es la que tiene toda la lógica del disparo y que veremos en detalle en el apartado siguiente.

#### 5.3.2.2.3.3.3 Create Projectile

Esta función es la que contiene toda la lógica del disparo y que cambia dependiendo del tipo de arma. Como antes mostramos el event **shoot** de la clase AK ahora también mostraremos la función **CreateProjectile** de esta clase.

Debido a la extensión de esta función he decidido dividirla en 2 imágenes. Se ha dejado la función **getRaycastPos** en ambas imágenes para intentar de facilitar la conexión entre ambas, sería el final de la primera foto y el inicio de la segunda.

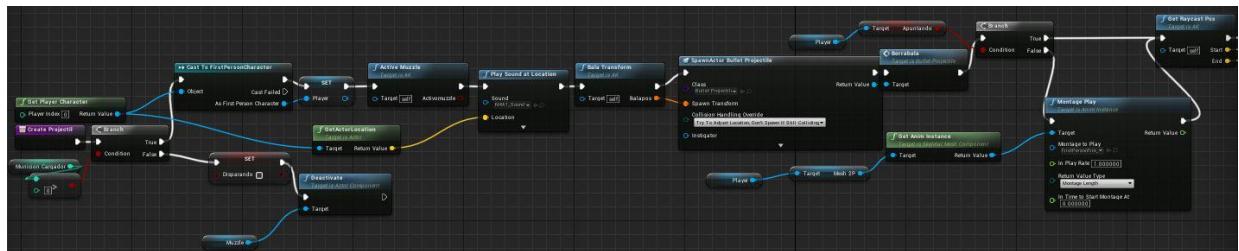


Figura 95: Function CreateProjectile (parte 1/2)

Comprobamos que tenemos balas en el cargador, si no tenemos desactivamos el muzzle y ponemos la variable **disparando** a falsa.

Si tenemos balas nos guardamos una referencia del **FirstPersonCharacter** en la variable **player** ya que vamos a tener que usar la clase **FirstPersonCharacter** varias veces y así nos ahorraremos tener que hacer varios casteos. A continuación activamos el destello del arma y obtenemos la posición del arma respecto al mundo para reproducir el sonido en esta posición. El sonido es diferente para cada arma, es decir, cada arma tiene su sonido real de disparo.

En segundo lugar crearíamos un actor de tipo **Bullet Projectil** pasándole como parámetro la transformación de la bala obtenida en la función **BalaTransform**. Una vez creada la bala llamamos a su método **BorraBala** que se encargara de borrar la bala cuando colisione con algo o bien cuando pase un determinado tiempo y no haya colisionado con nada.

A continuación comprobamos si el player está apuntando o no, para reproducir la animación del retroceso al disparar solo si no está apuntando. Esto se ha hecho así porque era muy molesto cuando apuntabas con la mira y se reproducía esta animación de retroceso impidiéndote apuntar debido a que tenías el arma muy cerca de la cámara.

Tanto si estábamos si hemos reproducido la animación de retroceso como sino llamamos a la función **getRaycastPos** que nos dirá los puntos de inicio y final de *raycast* que tenemos que trazar para el disparo del arma. El resto de función sigue en la parte 2.

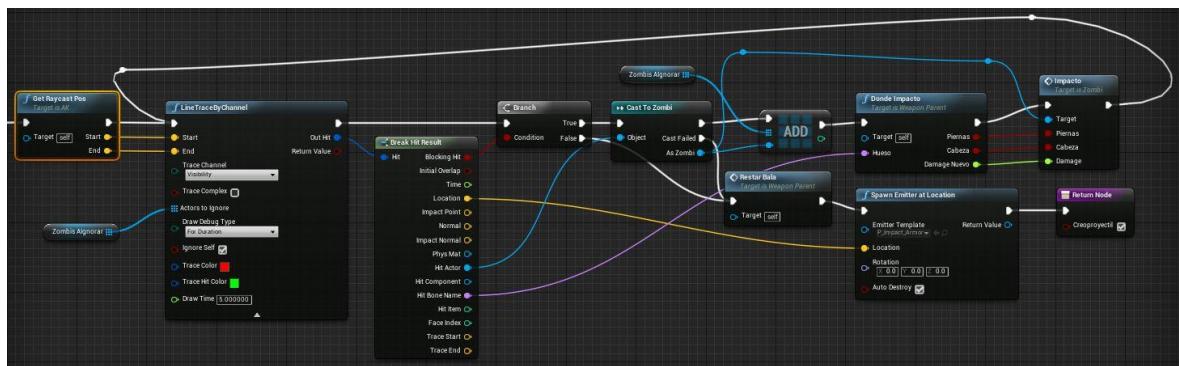


Figura 96: Function CreateProjectil (parte 2/2)

Ahora creamos un *raycast* con el método **LineTraceByChannel** pasándole por parámetro los vectores de inicio y final obtenidos en la función **getRayCastPos**. A esta función le pasamos también una variable local llamada **ZombisAIgnorar**, que es un array con todos los zombis a los que ya hemos golpeado (en este disparo en el que estamos ahora). Nos interesa que esta variable sea local ya que tiene que empezar vacía cada vez que se ejecuta esta función, con el objetivo de que no puedas darle 2 veces al mismo zombi con el mismo disparo (porque se añade ese zombi a la lista de zombis a ignorar) pero que si haces otro disparo esta lista este vacía de nuevo para que puedas volver a darle a cualquier zombi. Esto se ha hecho porque quería que las balas atravesaran a los zombis, así si los consigues poner en fila india las balas no impactan solo al primer zombi sino que impactan a todos los que están en la trayectoria de la bala.

**Para conseguir esto cada vez que la bala impacta contra un zombi lo añade a la lista de zombis a ignorar para que no pueda impactarle de nuevo y se vuelve a lanzar el raycast (como**

si volviéramos a disparar) pero esta vez solo daremos a aquellos zombis a los que la bala no haya impactado ya. Esto se hará mientras la bala siga impactando contra zombis.

Como resultado de la función **LineTraceByChannel** tenemos el nodo **BreakHitResult** del cual nos interesan 4 cosas:

- **Blocking Hit:** es un booleano que será verdadero si el *raycast* ha impactado con algo y falso en caso de que no haya conseguido impactar con nada.
- **Location:** es la posición en el mundo donde impacta el *raycast*.
- **Hit Actor:** es el actor contra el que impactar el *raycast*.
- **Hit Bone Name:** es el nombre del hueso con el que impacta el *raycast*, esto nos servirá para saber en qué parte del cuerpo le hemos dado.

Una vez que hemos lanzado el raycas y tenemos los resultados que vienen dados en el nodo **Break Hit Result** hay 3 opciones:

- **Opción 1:** La bala no ha impactado con nada, por lo tanto se llamaría a la función **RestarBala**.
- **Opción 2:** La bala ha impactado con un actor pero no es un zombi. Se llamaría a **RestarBala** y se crearía una partícula de impacto de bala en el punto de impacto que viene dado por el **Location** del nodo **Break Hit Result**.
- **Opción 3:** La bala ha impactado contra un zombi. Se añade este zombi a la lista de zombis a ignorar para este disparo que se almacenan en la variable **ZombisAIgnorar**. A continuación se llamaría a la función **DondeImpacto** pasándole como parámetro de entrada el **Hit Bone Name** para obtener los detalles del impacto en el zombi y cuando tenemos los resultados se los pasamos a la función **Impacto** del zombi al cual ha impactado la bala. Y volveríamos a lanzar un raycast puesto que hemos dado a un zombi y puede ser que hayan más detrás de él.

Como vemos la bala atravesaría zombis pudiéndole dar a varios zombis a la vez si están uno detrás de otro pero nunca atravesaría otro tipo de objetos como paredes.

#### 5.3.2.2.3.3.3.1 Función BalaTransform

Esta es la función devuelve la transformación que se usara para crear una bala que es un actor de la clase Bullet Projectil.

Antes de empezar hay que aclarar el componente que se usa en esta función llamado **Sphere**. Este componente es una esfera que se encuentra en el cañón del arma para poder saber en todo momento desde donde se tendría que disparar la bala.

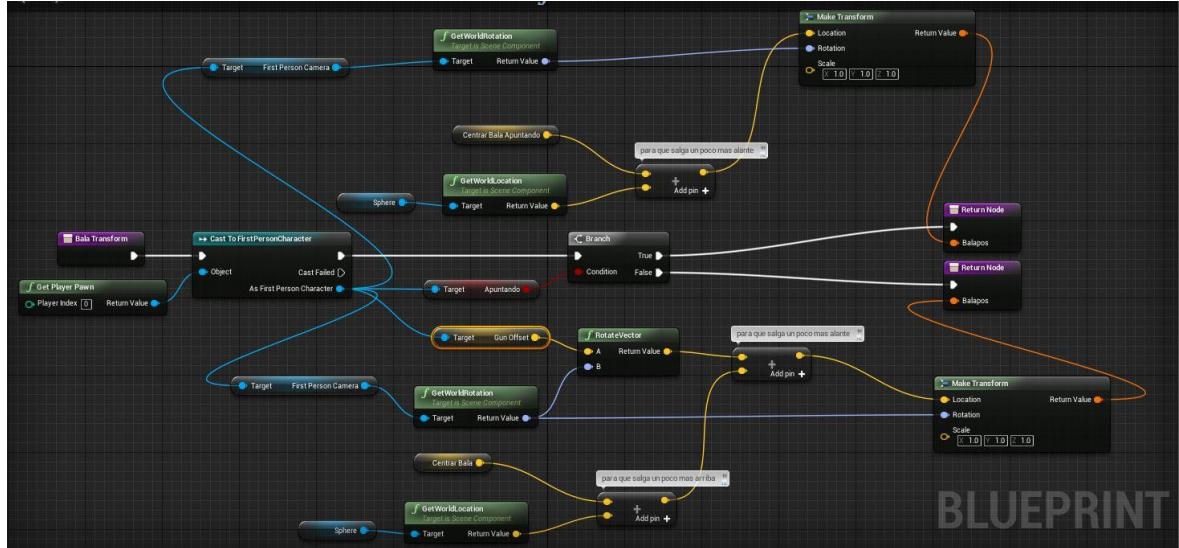


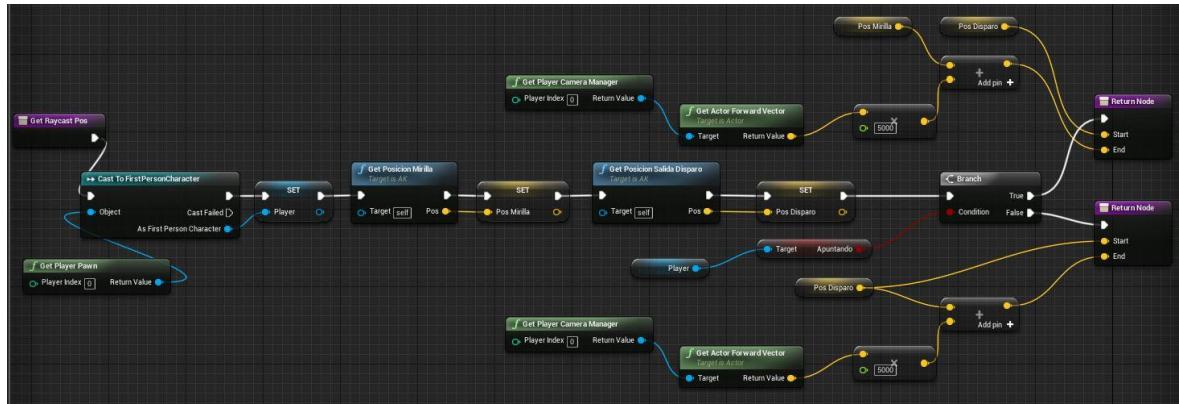
Figura 97: Función *BalaTransform*

En primer lugar se obtiene el player puesto que necesitamos acceder a algunos de sus componentes como la cámara. Comprobamos si el player está apuntando.

- Si está apuntando entonces creamos una transformación donde la rotación es la rotación de la cámara y la posición es la posición de **Sphere** a la que se le suma el vector **CentrarBalaApuntado** para centrar el disparo con respecto a la cámara. Finalmente devolvemos esta transformación.
- Si el player no está apuntando entonces obtenemos como rotación la rotación de la cámara igual que en el caso anterior. La posición ahora cambiaria un poco ya que cogemos el vector del jugador llamado **GunOffset** y lo rotamos usando como rotación la de la cámara, y a este vector resultante le sumamos el vector que resulta de la suma del vector **CentrarBala** con el vector que nos da la posición de **Sphere**. Por ultimo devolvemos esta transformación.

### 5.3.2.2.2.3.3.3.2 Función *GetRaycastPos*

Esta función devuelve 2 vectores, que son el punto inicial y el punto final sobre el cual se tiene que lanzar el raycast.

Figura 98: Función `getRayCastPos`

En primer lugar nos guardamos una referencia del `FirstPersonCharacter` en la variable `player` y llamamos a la función `getPosicionMirilla` que nos da el vector con la posición de la mirilla respecto al mundo. Nos guardamos este vector en la variable `posMirilla`. Llamamos a la función `getPosicionSalidaDisparo` que nos devuelve un vector con la posición del cañón del arma en respecto al mundo y nos lo guardamos en la variable `posDisparo`.

A continuación comprobamos si el jugador está apuntando.

- Si está apuntando entonces nuestro punto de inicio (`start`) del *raycast* será la posición del cañón del arma (variable `posDisparo`) y nuestro punto final (`end`) será el resultado de multiplicar el vector dirección del jugador por 5000 (alcance del disparo) y sumarlo a la posición de la mirilla (almacenada en la variable `posMirilla`).
- Si no está apuntando entonces nuestro punto de inicio del raycast será la posición del cañón del arma. Nuestro punto final será el resultado de multiplicar el vector dirección del jugador por 5000 (alcance del disparo) y sumarlo a la posición del cañón (almacenada en la variable `posDisparo`).

### 5.3.2.2.3.3.3.3 Funcion DondelImpacto

Esta función se utiliza para saber los detalles del impacto de la bala sobre el zombi y luego pasarle estos resultados a la función `Impacto` de la Clase Zombi que se encargara de tratarlos.

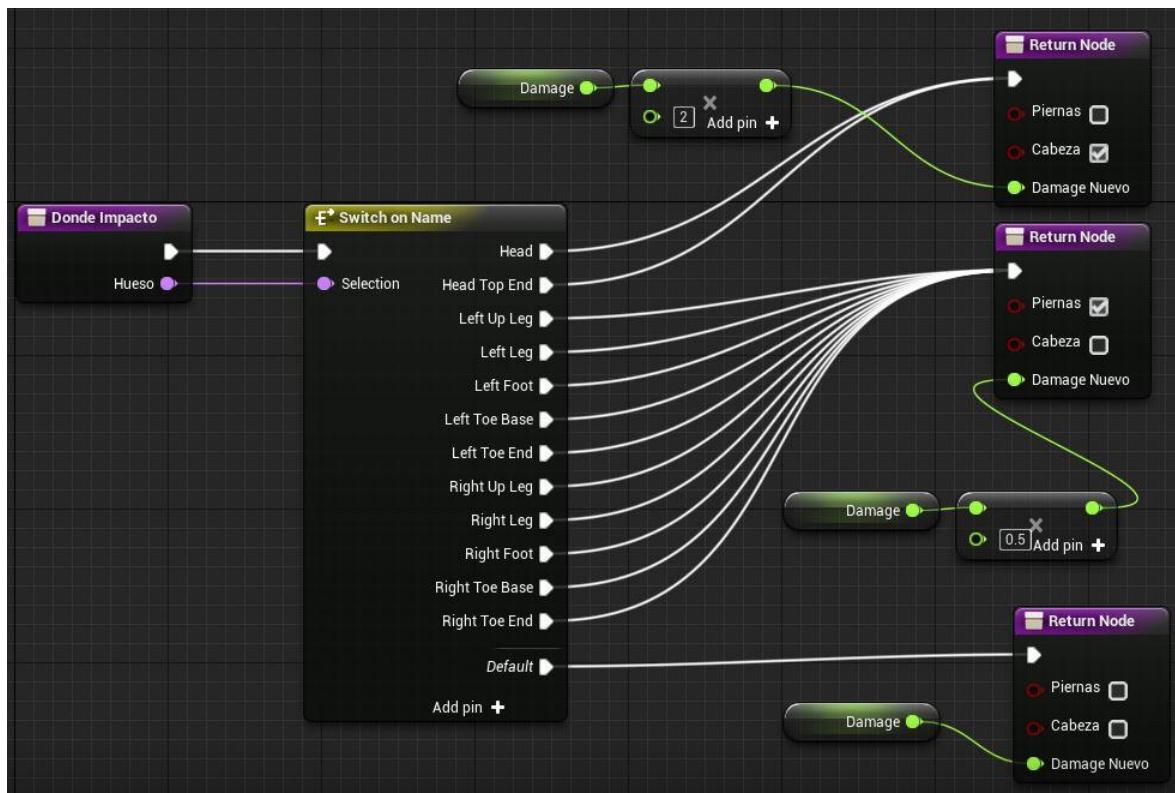


Figura 99: Función DondeImpacto

Esta función recibe por parámetro el nombre del hueso sobre el que impactó la bala que viene dado por el nodo **Break Hit Result** de la función **createProjectil**. Con este nombre se hace un Switch pudiendo obtener 3 resultados:

- El hueso pertenece a la cabeza por lo tanto devolvemos 3 valores que serán falso para el booleano piernas, true para el booleano cabeza y el daño que es el daño del arma pero multiplicada por 2. De esta forma hacemos que los impactos a la cabeza hagan el doble de daño.
- El hueso pertenece a las piernas. En este caso el booleano “Piernas” será verdadero, el booleano “Cabeza” será falso y el daño que causa el impacto de la bala será la mitad que el que causa normalmente. De esta forma hacemos que los impactos en las piernas hagan la mitad de daño.
- El hueso impactado no es ni de la cabeza ni de las piernas. Los dos booleanos “Piernas” y “Cabeza” serán falsos y el daño será el normal que viene dado por la variable **damage** del arma.

#### 5.3.2.2.3.3.3.4 Bullet Projectil Actor

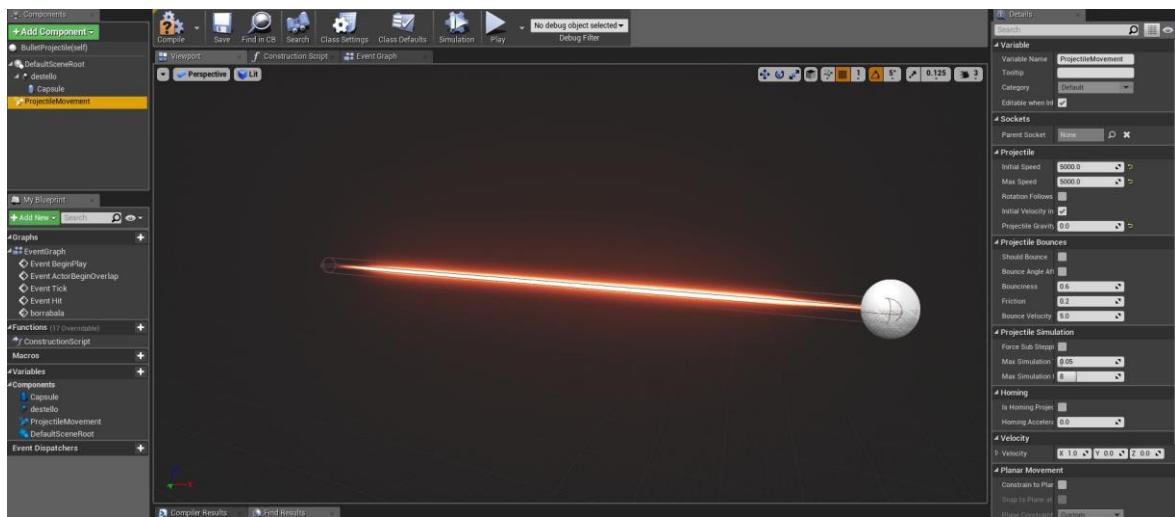


Figura 100: Bullet Projectil Actor

La bala que disparamos es un actor de la clase **Bullet Projectil**. Como se observa en la foto superior es una especie de destello. Sus componentes son:

- **Capsule:** una capsula de colisión.
- **Destello:** es un sistema de partículas que simula el destello de una bala.
- **ProjectileMovement:** es un componente de tipo **Projectil Movement** que permite el movimiento del proyectil.

#### 5.3.2.2.3.4 Dejar de disparar

#### 5.3.2.2.3.5 Recargar

#### 5.3.2.2.3.6 Mejorar Arma

##### 5.3.2.2.3.6.1 Mejorar Daño

##### 5.3.2.2.3.6.2 Mejorar Munición Máxima

##### 5.3.2.2.3.6.3 Mejorar Capacidad Cargador

##### 5.3.2.2.3.6.4 Mejorar Cadencia

##### 5.3.2.2.3.6.5 Mejorar Tiempo Recarga

### 5.3.2.3 Interactuable Objects (Power-ups)

### 5.3.2.4 Enemigo

### 5.3.2.5 Blueprint Partida

**5.3.3 Desarrollo Menús y HUD**

**5.3.4 Arte 3D en el videojuego**

**5.3.5 Diseño de nivel de juego (escenario, iluminación, sonidos)**

## 6. Bibliografía

Los géneros de los videojuegos. Recuperado el 21 de Junio de 2017, del Sitio web de Wikipedia  
[https://en.wikipedia.org/w/index.php?title=List\\_of\\_video\\_game\\_genres&oldid=781933781](https://en.wikipedia.org/w/index.php?title=List_of_video_game_genres&oldid=781933781)

Survival horror. (2017, 9 de abril). *Wikipedia, La enciclopedia libre*. Fecha de consulta: 15:58, junio 21, 2017  
desde [https://es.wikipedia.org/w/index.php?title=Survival\\_horror&oldid=98237658](https://es.wikipedia.org/w/index.php?title=Survival_horror&oldid=98237658).

Tennis for Two. (2016, 4 de mayo). *Wikipedia, La enciclopedia libre*. Fecha de consulta: 16:09, junio 21, 2017

desde [https://es.wikipedia.org/w/index.php?title=Tennis\\_for\\_Two&oldid=90859411](https://es.wikipedia.org/w/index.php?title=Tennis_for_Two&oldid=90859411).

Primer videojuego. (2017, 20 de junio). *Wikipedia, La enciclopedia libre*. Fecha de consulta: 16:09, junio 21, 2017

desde [https://es.wikipedia.org/w/index.php?title=Primer\\_videojuego&oldid=99968828](https://es.wikipedia.org/w/index.php?title=Primer_videojuego&oldid=99968828).

Historia de los videojuegos. (2017, 6 de junio). *Wikipedia, La enciclopedia libre*. Fecha de consulta:

17:55, junio 21, 2017

[https://es.wikipedia.org/w/index.php?title=Historia\\_de\\_los\\_videojuegos&oldid=99668007](https://es.wikipedia.org/w/index.php?title=Historia_de_los_videojuegos&oldid=99668007).

Videojuego. (2017, 20 de junio). *Wikipedia, La enciclopedia libre*. Fecha de consulta: 16:38, junio 22, 2017 <https://es.wikipedia.org/w/index.php?title=Videojuego&oldid=99963794>.

Videojuego (concepto) *Nintendo wiki*. Fecha de consulta: junio 22, 2017  
[http://es.nintendo.wikia.com/wiki/Videojuego\\_\(concepto\)](http://es.nintendo.wikia.com/wiki/Videojuego_(concepto))

Definición de videojuego , Master Magazine Fecha de consulta: junio 22, 2017  
<https://www.mastermagazine.info/termino/7136.php>

Simulador de vuelo. (2017, 28 de mayo). *Wikipedia, La enciclopedia libre*. Fecha de consulta: 16:56, junio 22, 2017

desde [https://es.wikipedia.org/w/index.php?title=Simulador\\_de\\_vuelo&oldid=99443656](https://es.wikipedia.org/w/index.php?title=Simulador_de_vuelo&oldid=99443656).

Primera persona (videojuegos). (2013, 6 de junio). *Wikipedia, La enciclopedia libre*. Fecha de consulta:

18:38, junio 22, 2017

desde [https://es.wikipedia.org/w/index.php?title=Primera\\_persona\\_\(videojuegos\)&oldid=67508046](https://es.wikipedia.org/w/index.php?title=Primera_persona_(videojuegos)&oldid=67508046).

Videojuego de disparos. (2017, 13 de mayo). *Wikipedia, La enciclopedia libre*. Fecha de consulta:

18:38, junio 22, 2017

desde [https://es.wikipedia.org/w/index.php?title=Videojuego\\_de\\_disparos&oldid=99119398](https://es.wikipedia.org/w/index.php?title=Videojuego_de_disparos&oldid=99119398).

Videojuego de disparos en primera persona. (2017, 8 de junio). *Wikipedia, La enciclopedia libre*. Fecha de consulta:

18:39, junio 22, 2017

desde [https://es.wikipedia.org/w/index.php?title=Videojuego\\_de\\_disparos\\_en\\_primera\\_persona&oldid=99705150](https://es.wikipedia.org/w/index.php?title=Videojuego_de_disparos_en_primera_persona&oldid=99705150).

Nazi zombies Call of Duty World at War. *Call of duty Wiki*. Fecha de consulta: junio 22, 2017  
[http://es.callofduty.wikia.com/wiki/Zombies\\_\(modo\)](http://es.callofduty.wikia.com/wiki/Zombies_(modo))

Call of Duty: Black Ops 3. (2017, 6 de mayo). *Wikipedia, La enciclopedia libre*. Fecha de consulta:  
21:07, junio 22, 2017

desde [https://es.wikipedia.org/w/index.php?title=Call\\_of\\_Duty:\\_Black\\_Ops\\_3&oldid=98921762](https://es.wikipedia.org/w/index.php?title=Call_of_Duty:_Black_Ops_3&oldid=98921762)

Call of Duty: Black Ops 2. (2017, 19 de junio). *Wikipedia, La enciclopedia libre*. Fecha de consulta:  
21:07, junio 22, 2017

desde [https://es.wikipedia.org/w/index.php?title=Call\\_of\\_Duty:\\_Black\\_Ops\\_2&oldid=99956102](https://es.wikipedia.org/w/index.php?title=Call_of_Duty:_Black_Ops_2&oldid=99956102)

Call of Duty: Black Ops. (2017, 18 de junio). *Wikipedia, La enciclopedia libre*. Fecha de consulta:  
21:08, junio 22, 2017

desde [https://es.wikipedia.org/w/index.php?title=Call\\_of\\_Duty:\\_Black\\_Ops&oldid=99930817](https://es.wikipedia.org/w/index.php?title=Call_of_Duty:_Black_Ops&oldid=99930817)

Call of Duty: World at War. (2017, 2 de mayo). *Wikipedia, La enciclopedia libre*. Fecha de consulta:  
21:08, junio 22, 2017

desde [https://es.wikipedia.org/w/index.php?title=Call\\_of\\_Duty:\\_World\\_at\\_War&oldid=98817528](https://es.wikipedia.org/w/index.php?title=Call_of_Duty:_World_at_War&oldid=98817528)

Dead Island. (2017, 8 de junio). *Wikipedia, La enciclopedia libre*. Fecha de consulta:  
10:57, junio 23, 2017

desde [https://es.wikipedia.org/w/index.php?title=Dead\\_Island&oldid=99713009](https://es.wikipedia.org/w/index.php?title=Dead_Island&oldid=99713009).

Dead Island. (2017, 8 de junio). *Meristation*. Fecha de consulta:  
10:57, junio 23, 2017 <http://www.meristation.com/xbox-360/dead-island/analisis-juego/1534918>

Left 4 Dead 2. (2017, 8 de junio). *Meristation*. Fecha de consulta:  
10:57, junio 23, 2017 <http://www.meristation.com/xbox-360/left-4-dead-2/analisis-juego/1529629>

Left 4 Dead 2. (2017, 24 de mayo). *Wikipedia, La enciclopedia libre*. Fecha de consulta:  
11:28, junio 23, 2017

desde [https://es.wikipedia.org/w/index.php?title=Left\\_4\\_Dead\\_2&oldid=99356674](https://es.wikipedia.org/w/index.php?title=Left_4_Dead_2&oldid=99356674)

Resident Evil 7: Biohazard. (2017, 22 de junio). *Wikipedia, La enciclopedia libre*. Fecha de consulta:  
11:36, junio 23, 2017

desde [https://es.wikipedia.org/w/index.php?title=Resident\\_Evil\\_7:\\_Biohazard&oldid=100001980](https://es.wikipedia.org/w/index.php?title=Resident_Evil_7:_Biohazard&oldid=100001980)

Kanban: una explicación del método . (2017, 9 de julio). Javier Garzás. Fecha de consulta:  
11:36, junio 23, 2017 desde <http://www.javiergarzas.com/2011/11/kanban.html>

Garzás, J. (2017). *Kanban: una explicación del método*. Javier Garzás. Available at:  
<http://www.javiergarzas.com/2011/11/kanban.html> [Accessed 10 Jul. 2017].

WWWhat's new? - Aplicaciones, marketing y noticias en la web. (2017). *toggl, excelente opción para gestionar nuestro tiempo entre proyectos*. [online] Available at:

<https://wwwwhatsnew.com/2014/03/01/toggl-excelente-opcion-para-gestionar-nuestro-tiempo-entre-proyectos/> [Accessed 10 Jul. 2017].

Anon, (2017). [online] Available at:

<http://www.idera.gob.ar/portal/sites/default/files/TrelloTutorialBasico.pdf> [Accessed 10 Jul. 2017].

Software. (2017, 8 de julio). *Wikipedia, La enciclopedia libre*. Fecha de consulta: 11:45, julio 10, 2017

desde <https://es.wikipedia.org/w/index.php?title=Software&oldid=100337980>.

Motor de videojuego. (2017, 23 de mayo). *Wikipedia, La enciclopedia libre*. Fecha de consulta: 11:59, julio 10, 2017

desde [https://es.wikipedia.org/w/index.php?title=Motor\\_de\\_videojuego&oldid=99337855](https://es.wikipedia.org/w/index.php?title=Motor_de_videojuego&oldid=99337855).

La leyenda de Darwan. (2017). *Cinco cosas buenas y cinco no tan buenas de Unity 3D*. [online] Available at: <https://laleyendadedarwan.es/2016/06/25/cinco-cosas-buenas-y-cinco-no-tan-buenas-de-unity-3d/> [Accessed 10 Jul. 2017].

Unity (motor de juego). (2017, 9 de julio). *Wikipedia, La enciclopedia libre*. Fecha de consulta: 13:56, julio 10, 2017

desde [https://es.wikipedia.org/w/index.php?title=Unity\\_\(motor\\_de\\_juego\)&oldid=100369895](https://es.wikipedia.org/w/index.php?title=Unity_(motor_de_juego)&oldid=100369895).

Unreal Engine. (2017, 9 de mayo). *Wikipedia, La enciclopedia libre*. Fecha de consulta: 13:57, julio 10, 2017

desde [https://es.wikipedia.org/w/index.php?title=Unreal\\_Engine&oldid=98983482](https://es.wikipedia.org/w/index.php?title=Unreal_Engine&oldid=98983482).

CryEngine. (2017, 22 de junio). *Wikipedia, La enciclopedia libre*. Fecha de consulta: 13:57, julio 10, 2017

desde <https://es.wikipedia.org/w/index.php?title=CryEngine&oldid=100003883>.

Modelado 3D. (2017, 24 de abril). *Wikipedia, La enciclopedia libre*. Fecha de consulta: 17:20, julio 10, 2017

desde [https://es.wikipedia.org/w/index.php?title=Modelado\\_3D&oldid=98609591](https://es.wikipedia.org/w/index.php?title=Modelado_3D&oldid=98609591).

Mudbox. (2016, 17 de junio). *Wikipedia, La enciclopedia libre*. Fecha de consulta: 17:26, julio 10, 2017

desde <https://es.wikipedia.org/w/index.php?title=Mudbox&oldid=91763937>.

Autodesk Maya. (2017, 30 de abril). *Wikipedia, La enciclopedia libre*. Fecha de consulta: 18:14, julio 10, 2017

desde [https://es.wikipedia.org/w/index.php?title=Autodesk\\_Maya&oldid=98765232](https://es.wikipedia.org/w/index.php?title=Autodesk_Maya&oldid=98765232).

Autodesk 3ds Max. (2017, 10 de junio). *Wikipedia, La enciclopedia libre*. Fecha de consulta: 18:13, julio 10, 2017

desde [https://es.wikipedia.org/w/index.php?title=Autodesk\\_3ds\\_Max&oldid=99746556](https://es.wikipedia.org/w/index.php?title=Autodesk_3ds_Max&oldid=99746556).

Blender. (2017, 7 de julio). *Wikipedia, La enciclopedia libre*. Fecha de consulta: 18:13, julio 10, 2017

desde <https://es.wikipedia.org/w/index.php?title=Blender&oldid=100314825>.

Jugabilidad. (2017, 2 de mayo). *Wikipedia, La enciclopedia libre*. Fecha de consulta: 10:58, julio 12, 2017

desde <https://es.wikipedia.org/w/index.php?title=Jugabilidad&oldid=98823068>.

Sistema de juego. (2017, 31 de mayo). *Wikipedia, La enciclopedia libre*. Fecha de consulta: 17:57, julio 12, 2017

desde [https://es.wikipedia.org/w/index.php?title=Sistema\\_de\\_juego&oldid=99507704](https://es.wikipedia.org/w/index.php?title=Sistema_de_juego&oldid=99507704).

Studio. (2017). *2. el videojuego*. Es.slideshare.net. Retrieved 12 July 2017, from <https://es.slideshare.net/tongoxcore/2-el-videojuego-7649908>

*toggl, excelente opción para gestionar nuestro tiempo entre proyectos.* (2017). *WWWhat's new? - Aplicaciones, marketing y noticias en la web*. Retrieved 10 July 2017, from <https://wwwwhatsnew.com/2014/03/01/toggl-excelente-opcion-para-gestionar-nuestro-tiempo-entre-proyectos/>

Garzás, J. (2017). *Kanban: una explicación del método*. Javier Garzás. Retrieved 10 July 2017, from <http://www.javiergarzas.com/2011/11/kanban.html>

*Cinco cosas buenas y cinco no tan buenas de Unity 3D.* (2017). *La leyenda de Darwan*. Retrieved 10 July 2017, from <https://laleyendadedarwan.es/2016/06/25/cinco-cosas-buenas-y-cinco-no-tan-buenas-de-unity-3d/>

(2017). Retrieved 10 July 2017, from <http://www.idera.gob.ar/portal/sites/default/files/TrelloTutorialBasico.pdf>

*Gamificación: mecánicas de juego - BrainSINS.* (2017). BrainSINS. Retrieved 12 July 2017, from <https://www.brainsins.com/es/blog/gamificacion-mecanicas-de-juego/3131>

Studio. (2017). *7. mecánicas de juego*. Es.slideshare.net. Retrieved 12 July 2017, from <https://es.slideshare.net/tongoxcore/7-mecnicas-de-juego>

Potenciador. (2016, 12 de marzo). *Wikipedia, La enciclopedia libre*. Fecha de consulta: 19:03, julio 12, 2017

desde <https://es.wikipedia.org/w/index.php?title=Potenciador&oldid=89779206>.

Música de videojuegos. (2017, 4 de mayo). *Wikipedia, La enciclopedia libre*. Fecha de consulta: 11:43, julio 13, 2017

desde [https://es.wikipedia.org/w/index.php?title=M%C3%BAsica\\_de\\_videojuegos&oldid=98867629](https://es.wikipedia.org/w/index.php?title=M%C3%BAsica_de_videojuegos&oldid=98867629).

*TODO cambiar todos los que están arriba de esto*

*Graphs.* (2017). Docs.unrealengine.com. Recuperado 23 August 2017, a partir de <https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/Graphs/index.html>

*Components Window.* (2017). Docs.unrealengine.com. Recuperado 23 August 2017, a partir de <https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/Components/index.html>

*Construction Script.* (2017). [Docs.unrealengine.com](https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/UserConstructionScript/index.html). Recuperado 23 August 2017, a partir de <https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/UserConstructionScript/index.html>

*EventGraph.* (2017). [Docs.unrealengine.com](https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/EventGraph/index.html). Recuperado 23 August 2017, a partir de <https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/EventGraph/index.html>

*Functions.* (2017). [Docs.unrealengine.com](https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/Functions/index.html). Recuperado 23 August 2017, a partir de <https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/Functions/index.html>

*Macros.* (2017). [Docs.unrealengine.com](https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/Macros/index.html). Recuperado 23 August 2017, a partir de <https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/Macros/index.html>

*Types of Blueprints.* (2017). [Docs.unrealengine.com](https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/Types/index.html). Recuperado 23 August 2017, a partir de <https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/Types/index.html>

*Level Blueprint.* (2017). [Docs.unrealengine.com](https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/Types/LevelBlueprint/index.html). Recuperado 23 August 2017, a partir de <https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/Types/LevelBlueprint/index.html>

*Blueprint Macro Library.* (2017). [Docs.unrealengine.com](https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/Types/MacroLibrary/index.html). Recuperado 23 August 2017, a partir de <https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/Types/MacroLibrary/index.html>

*Blueprint Interface.* (2017). [Docs.unrealengine.com](https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/Types/Interface/index.html). Recuperado 23 August 2017, a partir de <https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/Types/Interface/index.html>

*Blueprint Class.* (2017). [Docs.unrealengine.com](https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/Types/ClassBlueprint/index.html#data-onlyblueprint). Recuperado 23 August 2017, a partir de <https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/Types/ClassBlueprint/index.html#data-onlyblueprint>

*Types of Lights.* (2017). [Docs.unrealengine.com](https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/LightTypes/index.html). Recuperado 23 August 2017, a partir de <https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/LightTypes/index.html>

*Light Mobility.* (2017). [Docs.unrealengine.com](https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/LightMobility/index.html). Recuperado 23 August 2017, a partir de <https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/LightMobility/index.html>

*Static Lights.* (2017). [Docs.unrealengine.com](https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/LightMobility/StaticLights/index.html). Recuperado 23 August 2017, a partir de <https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/LightMobility/StaticLights/index.html>

*Stationary lights.* (2017). [Docs.unrealengine.com](https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/LightMobility/StationaryLights/index.html). Recuperado 23 August 2017, a partir de <https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/LightMobility/StationaryLights/index.html>

*Movable Lights.* (2017). [Docs.unrealengine.com](https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/LightMobility/DynamicLights/index.html). Recuperado 23 August 2017, a partir de <https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/LightMobility/DynamicLights/index.html>

*Materials.* (2017). [Docs.unrealengine.com](https://docs.unrealengine.com/latest/INT/Engine/Rendering/Materials/). Recuperado 24 August 2017, a partir de <https://docs.unrealengine.com/latest/INT/Engine/Rendering/Materials/>

*Essential Material Concepts.* (2017). *Docs.unrealengine.com*. Recuperado 24 August 2017, a partir de  
<https://docs.unrealengine.com/latest/INT/Engine/Rendering/Materials/IntroductionToMaterials/index.html>

*Post Process Effects.* (2017). *Docs.unrealengine.com*. Recuperado 24 August 2017, a partir de  
<https://docs.unrealengine.com/latest/INT/Engine/Rendering/PostProcessEffects/>

*Audio and Sound.* (2017). *Docs.unrealengine.com*. Recuperado 24 August 2017, a partir de  
<https://docs.unrealengine.com/latest/INT/Engine/Audio/>

*Audio System Overview.* (2017). *Docs.unrealengine.com*. Recuperado 24 August 2017, a partir de  
<https://docs.unrealengine.com/latest/INT/Engine/Audio/Overview/index.html>

*Sound Cue Editor.* (2017). *Docs.unrealengine.com*. Recuperado 24 August 2017, a partir de  
<https://docs.unrealengine.com/latest/INT/Engine/Audio/SoundCues/Editor/index.html>

*Sound Attenuation.* (2017). *Docs.unrealengine.com*. Recuperado 24 August 2017, a partir de  
<https://docs.unrealengine.com/latest/INT/Engine/Audio/DistanceModelAttenuation/index.html>

*Ambient Sound Actor User Guide.* (2017). *Docs.unrealengine.com*. Recuperado 24 August 2017, a partir de <https://docs.unrealengine.com/latest/INT/Engine/Audio/SoundActors/index.html>

*Animation Blueprint Editor.* (2017). *Docs.unrealengine.com*. Recuperado 24 August 2017, a partir de  
<https://docs.unrealengine.com/latest/INT/Engine/Animation/AnimBlueprints/Interface/index.html>

*Animation Blueprints.* (2017). *Docs.unrealengine.com*. Recuperado 24 August 2017, a partir de  
<https://docs.unrealengine.com/latest/INT/Engine/Animation/AnimBlueprints/index.html>

*Animation Editors.* (2017). *Docs.unrealengine.com*. Recuperado 24 August 2017, a partir de  
<https://docs.unrealengine.com/latest/INT/Engine/Animation/Persona/index.html>

*Animation Notifications (Notifies).* (2017). *Docs.unrealengine.com*. Recuperado 24 August 2017, a partir de <https://docs.unrealengine.com/latest/INT/Engine/Animation/Sequences/Notifies/>

*Animation Retargeting (Different Skeletons).* (2017). *Docs.unrealengine.com*. Recuperado 24 August 2017, a partir de  
<https://docs.unrealengine.com/latest/INT/Engine/Animation/RetargetingDifferentSkeletons/index.html>

*Blend Spaces.* (2017). *Docs.unrealengine.com*. Recuperado 24 August 2017, a partir de  
<https://docs.unrealengine.com/latest/INT/Engine/Animation/Blendspaces/index.html>

*Blend Spaces Overview.* (2017). *Docs.unrealengine.com*. Recuperado 24 August 2017, a partir de  
<https://docs.unrealengine.com/latest/INT/Engine/Animation/Blendspaces/Overview/index.html>

*Blending Animations.* (2017). *Docs.unrealengine.com*. Recuperado 24 August 2017, a partir de  
<https://docs.unrealengine.com/latest/INT/Engine/Animation/AnimationBlending/index.html>