



Escuela
Politécnica
Superior

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 Lizán

Juan Antonio Puchol García

Justificación y Objetivos

Después de haber analizado y creado distintos tipos de videojuegos en la carrera y habiendo adquirido ya una base de cómo diseñar un videojuego, creo que ha llegado el momento de realizar uno con un motor gráfico mucho más potente como es **Unreal Engine 4** con el objeto de exprimir al máximo estas tecnologías que están en auge actualmente, ya que con *Unreal Engine 4* se pueden obtener unos resultados de calidad y muy vistosos aprovechando al máximo los gráficos de este motor de juego.

Un parte importante de este proyecto es usar estos motores gráficos cada vez más para conseguir dominar esta tecnología, lo cual es una meta muy importante para un Ingeniero Multimedia.

No obstante, el objetivo general de este proyecto es conseguir realizar un juego de calidad utilizando uno de los motores gráficos comerciales, como en este caso es *Unreal Engine 4*, y aprender a sacarle el máximo partido a esta tecnología con el fin de conseguir un producto final acabado y de calidad.

Dedicatoria

En primer lugar me gustaría dedicar este Trabajo de Fin de Grado a mi familia, que ha sido la que siempre ha estado a mi lado, la que me ha ayudado en los momentos duros y me animaba a continuar, y es que gracias a ellos he llegado hasta aquí.

En segundo lugar le dedico este proyecto a todos aquellos compañeros de la carrera que me han acompañado durante estos 4 años y que han hecho que sean unos 4 años increíbles llenos de risas y buenos momentos. Pero en especial a mi grupo de ABP José Luis, Enrique, Toni y Julio con los que forme Paradox Studios, sin olvidar a aquellos con los que más unido he estado este último año y medio de carrera que son Stoycho, Nerea, Julia y Catherine. Por supuesto mención de honor para el compañero de carrera, compañero de piso y cocinero Alfonso. Y aunque este último año no he estado mucho con ellos no podía olvidarme de “Adri Sin An” y el “psao” de Samu.

Por ultimo aunque no menos importante a toda la “Casika” que más que amigos han sabido portarse como una gran familia durante todos los años que seguimos juntos, y a los que considero grandes amigos.

Con este trabajo termina una de las mejores etapas de mi vida, en la cual he conocido a muchas personas increíbles que no hubiera conocido en otras circustancias. De mi paso por la universidad no solo me llevo el titulo de Ingenierio Multimedia me llevo una gran cantidad de experiencias vividas, muchas risas, momentos de tensión y agobios durante las entregas, pero sobre todo una gran cantidad de amigos.

¡Muchas gracias a TODOS!

**“Programar es una de las pocas cosas
del mundo en la que te sientas y puedes
crear algo totalmente nuevo de la nada.”**

-Mark Zuckerberg

Resumen

Durante los últimos años ha aumentado mucho la demanda de videojuegos de modo que lo que antes era cosa de unos pocos “gammers” ahora es un fenómeno habitual en casi todo el mundo. La modernización de los dispositivos móviles y el gran número de juegos para estos han propiciado el éxito de los videojuegos hasta el punto de que esta industria factura más que el cine y el sonido juntos. Evidentemente, el aumento de la demanda ha traído consigo un incremento de la oferta lo que ha redundado en una mayor calidad de los videojuegos que compiten entre sí por el favor del público.

Si los primeros videojuegos se reducían a unas pocas interacciones y eran muy sencillos gráficamente; actualmente, en cambio, los videojuegos son considerados verdaderas obras de arte ya que son capaces de contar una historia como lo haría una película, con la diferencia de que el jugador se convierte en el protagonista de esta. Por esa razón, el presente proyecto realizado en *Unreal Engine 4* pretende situar al jugador como el protagonista de una pequeña historia y potenciar tanto la *jugabilidad* como la calidad visual.

Índice de contenido

1. Introducción	26
2. Marco Teórico o Estado del Arte.....	28
2.1 ¿Qué es un videojuego?.....	28
2.2 Referencias	30
2.2.1 Modo Zombis de <i>Call of Duty</i>.....	30
2.2.1.5 <i>Dead Island</i>	33
2.2.1.6 <i>Left 4 Dead 2</i>	33
2.2.1.7 <i>Resident Evil VII</i>	34
3. Objetivos.....	37
3.1 Objetivo general del Trabajo de Fin de Grado	37
3.2 Desglose de Objetivos	37
4. Metodología	39
4.1 Metodología de desarrollo	39
4.1.1 Metodologías Ágiles	39
4.1.2 Kanban	39
4.2 Gestión de proyecto	40
4.2.1 <i>Toggl</i> y la compatibilidad con <i>Trello</i>.....	41
4.3 Control de versiones	44
5. Cuerpo del trabajo	46
5.1 Análisis y elección de herramientas.....	46
5.1.1 Motor de videojuego	46
5.1.1.1 Conceptos clave	46
5.1.1.2 Motores de videojuegos actuales.....	47
5.1.1.2.1 <i>Cry Engine 3</i>	50
5.1.1.2.2 <i>Unreal Engine 4</i>	50
5.1.1.2.3 <i>Unity 3D</i>	52
5.1.1.3 Elección del motor de juego.....	53
5.1.2 Otras herramientas.....	54
5.1.2.1 Modelado 3D	54
5.1.2.1.1 <i>Blender</i>	54
5.1.2.1.2 <i>3DS Max</i>	55
5.1.2.1.3 <i>Maya</i>	55

5.1.2.2 Texturizado	56
5.1.2.2.1 <i>Mudbox</i>	56
5.1.2.2.2 <i>Substance Painter</i>	56
5.1.2.3 Elección de herramientas.....	57
5.2 Documento de Diseño del Videojuego (GDD)	58
5.2.1 El juego en términos generales	58
5.2.1.1 Resumen de argumento	58
5.2.1.2 Conjunto de características.....	59
5.2.1.3 Género.....	59
5.2.1.4 Clasificación PEGI	59
5.2.1.4.1 ¿Qué es PEGI?.....	59
5.2.1.4.2 ¿Qué entendemos por clasificación?.....	60
5.2.1.4.3 ¿Cómo se mide la clasificación?.....	60
5.2.1.4.4 Clasificación PEGI de Postwar: <i>Hopeless Humanity</i>	61
5.2.1.5 Resumen del flujo de juego.....	62
5.2.1.6 Apariencia del juego	63
5.2.1.7 Ámbito.....	63
5.2.1.7.1 Lugares en los que se desarrolla el videojuego.....	63
5.2.1.7.2 Número de niveles	63
5.2.1.7.3 Número de NPC's.....	63
5.2.1.7.4 Número de habilidades	64
5.2.2 Jugabilidad y mecánicas.....	64
5.2.2.1 Jugabilidad	64
5.2.2.1.1 ¿Qué es la jugabilidad?..	64
5.2.2.1.2 Objetivos del juego	65
5.2.2.1.3 Progresión	65
5.2.2.1.4 Acciones del personaje.....	66
5.2.2.1.5 Controles del juego	66
5.2.2.2 Mecánicas.....	66
5.2.2.2.1 Movimiento y cámara en primera persona.....	67
5.2.2.2.2 Disparar y recargar.....	68
5.2.2.2.3 Comprar armas	68
5.2.2.2.4 Mejorar armas	69
5.2.2.2.5 Comprar munición	69
5.2.2.2.6 Power-ups	69
5.2.2.2.6.1 Resistencia	69
5.2.2.2.6.2 Recarga rápida.....	70
5.2.2.2.6.2 Revivir	70
5.2.2.2.7 Abrir puertas.....	70
5.2.2.3 Rejugar y salvar	70
5.2.2.4 <i>Ranking</i>	71
5.2.3 Historia y personajes	71

5.2.3.1 Historia	71
5.2.3.2 Personajes.....	72
5.2.3.2.1 Protagonista	72
5.2.3.2.2 Zombis.....	72
5.2.4 Nivel de juego.....	72
5.2.4.1 Inteligencia Artificial.....	72
5.2.4.2 Escenario	73
5.2.4.3 Música y sonidos.....	74
5.2.4.4 Puntos de <i>respawn</i>	74
5.2.5 Interfaz	75
5.2.5.1 Menús.....	75
5.2.5.1.1 Menú principal.....	76
5.2.5.1.2 Menú de pausa	77
5.2.5.1.3 Menú fin de partida	77
5.2.5.1.3.1 Sin Ranking.....	77
5.2.5.1.3.2 Con <i>Ranking</i>	78
5.2.5.1.4 Pantalla de <i>ranking</i>	79
5.2.5.1.5 Menú de opciones.....	79
5.2.5.2 HUD	80
5.2.5.3 Cámara	82
5.3 Desarrollo e implementación	84
5.3.1 Unreal Engine 4 y su sistema de <i>scripting</i> visual Blueprints	84
5.3.1.1 La arquitectura de <i>Unreal Engine 4</i>	84
5.3.1.1.1 Conceptos base de la arquitectura de <i>Unreal Engine 4</i>	84
5.3.1.1.1.1 <i>UObjects</i> y <i>Actors</i>	84
5.3.1.1.1.2 <i>Gameplay</i> y las clases básicas.....	84
5.3.1.1.1.3 Representación y control de jugadores.....	85
5.3.1.1.1.3.1 Representación de jugadores en el mundo	85
5.3.1.1.1.3.1.1 Pawn.....	85
5.3.1.1.1.3.1.2 Character	85
5.3.1.1.1.3.2 Formas de controlar a un jugador del mundo.....	85
5.3.1.1.1.3.2.1 Controller	85
5.3.1.1.1.3.2.2 PlayerController	85
5.3.1.1.1.3.2.3 AIController	86
5.3.1.1.1.4 Mostrando información a los jugadores	86
5.3.1.1.1.4.1 HUD.....	86
5.3.1.1.1.4.2 Cámara	86
5.3.1.1.1.5 Manejando información del juego.....	86
5.3.1.1.1.5.1 GameMode.....	86
5.3.1.1.1.5.2 GameState	87
5.3.1.1.1.5.3 PlayerState	87
5.3.1.1.1.6 Cómo se relacionan estas clases.....	87

5.3.1.1.2 Scripting visual <i>Blueprints</i>	88
5.3.1.1.2.1 ¿Qué es un <i>Blueprint</i> ?.....	89
5.3.1.1.2.2 ¿Qué contiene un <i>Blueprint</i> ?.....	89
5.3.1.1.2.2.1 Componentes (<i>Components Window</i>).....	89
5.3.1.1.2.2.2 Constructor (<i>Construction Script</i>).....	90
5.3.1.1.2.2.3 Event Graph	92
5.3.1.1.2.2.4 Funciones	92
5.3.1.1.2.2.5 Variables.....	92
5.3.1.1.2.2.6 Macros	92
5.3.1.1.2.3 Tipos de <i>Blueprints</i>	94
5.3.1.1.2.3.1 <i>Blueprint Class</i>	94
5.3.1.1.2.3.2 <i>Level Blueprint</i>	94
5.3.1.1.2.3.3 <i>Data-Only Blueprint</i>	94
5.3.1.1.2.3.4 <i>Blueprint Interface</i>	94
5.3.1.1.2.3.5 <i>Blueprint Macro Library</i>	95
5.3.1.1.3 Renderizado y gráficos	95
5.3.1.1.3.1 Luces.....	96
5.3.1.1.3.1.1 Static lights	99
5.3.1.1.3.1.2 Stationary lights	100
5.3.1.1.3.1.3 Movable lights	100
5.3.1.1.3.2 Materiales.....	101
5.3.1.1.3.3 Postprocesado (<i>PostProcess Volume</i>)	103
5.3.1.1.4 Audio y sonido	104
5.3.1.1.4.1 <i>Sound Cue Editor</i>	104
5.3.1.1.4.2 <i>Ambient Sound Actor</i>	105
5.3.1.1.4.3 <i>Sound Attenuation</i>	105
5.3.1.1.5 Animaciones	106
5.3.1.1.5.1 <i>Blending animation (Blend Spaces)</i>	106
5.3.1.1.5.2 <i>Animation Blueprint</i>	108
5.3.1.1.5.3 <i>Animation Retargeting (Different Skeletons)</i>	108
5.3.1.1.5.4 Notificación en animación.....	109
5.3.2 Desarrollo de mecánicas de juego.....	109
5.3.2.1 <i>Blueprint First Person Character</i> (personaje principal).....	109
5.3.2.1.1 Componentes	110
5.3.2.1.2 Variables.....	111
5.3.2.1.2 Funciones	114
5.3.2.1.3 Análisis de las funciones importantes en detalle	118
5.3.2.1.3.1 Apuntar.....	118
5.3.2.1.3.2 Equipar arma.....	120
5.3.2.1.3.3 Cambiar de arma	122
5.3.2.1.3.4 Disparar	123
5.3.2.1.3.5 Recibir daño	124

5.3.2.1.3.5.1 Restar vida.....	125
5.3.2.1.3.5.2 Regenerar vida.....	126
5.3.2.1.4 Animaciones	127
5.3.2.2 Armas	131
5.3.2.2.1 WeaponDrop Blueprint (armas en la pared).....	131
5.3.2.2.1.1 Componentes.....	131
5.3.2.2.1.2 Variables	131
5.3.2.2.1.3 Funciones.....	132
5.3.2.2.1.3.1 <i>ConstructionScript</i> (Constructor de clase).....	132
5.3.2.2.1.3.2 Inicializar	133
5.3.2.2.1.3.3 Entrar en zona de compra	134
5.3.2.2.1.3.4 Salir de zona de compra	136
5.3.2.2.1.3.5 Interactuar.....	136
5.3.2.2.1.3.6 Comprar arma.....	137
5.3.2.2.1.3.7 Comprar munición	138
5.3.2.2.1.3.8 Comprobar si ya hemos comprado el arma	139
5.3.2.2.1.3.9 Información compra arma y munición	140
5.3.2.2.2 Weapon Blueprint (arma equipada).....	140
5.3.2.2.2.1 Interfaz (<i>Weapon Interface</i>).....	140
5.3.2.2.2.2 Variables	141
5.3.2.2.2.3 Funciones.....	142
5.3.2.2.2.3.1 Constructor (<i>Construction Script</i>).....	142
5.3.2.2.2.3.2 Inicializar	143
5.3.2.2.2.3.3 Disparar	144
5.3.2.2.2.3.3.1 <i>InputAction Fire</i>	144
5.3.2.2.2.3.3.2 <i>Event Shoot</i>	145
5.3.2.2.2.3.3.3 <i>Create Projectil</i>	145
5.3.2.2.2.3.3.3.1 Función <i>BalaTransform</i>	148
5.3.2.2.2.3.3.3.2 Función <i>GetRaycastPos</i>	149
5.3.2.2.2.3.3.3.3 Funcion <i>DondeImpacto</i>	150
5.3.2.2.2.3.3.3.4 <i>Bullet Projectil Actor</i>	151
5.3.2.2.2.3.4 Dejar de disparar (<i>UnFireEvent</i>).....	152
5.3.2.2.2.3.5 Recargar	152
5.3.2.2.2.3.5.1 Función <i>GetAnimRecarga</i>	156
5.3.2.2.2.3.6 Mejorar Arma	157
5.3.2.2.2.3.6.1 Mejorar Daño	158
5.3.2.2.2.3.6.2 Mejorar Munición Máxima	158
5.3.2.2.2.3.6.3 Mejorar Capacidad Cargador	159
5.3.2.2.2.3.6.4 Mejorar Cadencia	159
5.3.2.2.2.3.6.5 Mejorar Tiempo Recarga	159
5.3.2.3 Interactive Objects (<i>Power-ups</i>)	160
5.3.2.3.1 <i>InteractiveParent Blueprint</i> (clase padre objetos interactivos)	160
5.3.2.3.1.1 Componentes.....	161
5.3.2.3.1.2 Variables	161

5.3.2.3.1.2 Funciones.....	162
5.3.2.3.1.2.1 <i>ConstructionScript</i>	162
5.3.2.3.1.2.2 Inicializar	162
5.3.2.3.1.2.3 Entrar en zona de interacción.....	163
5.3.2.3.1.2.4 Salir de zona de interacción.....	163
5.3.2.3.1.2.5 <i>InputAction</i> Interactuar.....	164
5.3.2.3.1.2.6 Función GetPrecioActual.....	164
5.3.2.3.2 Clases hijas <i>Interactive Objects</i>	165
5.3.2.3.2.1 Puerta	165
5.3.2.3.2.1.1 Construction Script.....	165
5.3.2.3.2.1.2 Función Interactúo	165
5.3.2.3.2.2 Power-up resistencia.....	166
5.3.2.3.2.2.1 Construction Script	166
5.3.2.3.2.2.2 Función Interactúo	167
5.3.2.3.2.3 Power-up revivir	167
5.3.2.3.2.3.1 Construction Script	167
5.3.2.3.2.3.2 Función Interactúo	168
5.3.2.3.2.4 Power-up tiempo recarga.....	168
5.3.2.3.2.4.1 Construction Script	168
5.3.2.3.2.4.2 Función Interactúo	169
5.3.2.3.2.5 Máquinas mejorar armas	169
5.3.2.3.2.5.1 Construction Script.....	169
5.3.2.3.2.5.2 Función Interactúo	170
5.3.2.3.2.5.2 Función GetPrecioActual	170
5.3.2.4 Enemigo.....	171
5.3.2.4.1 <i>Zombi Blueprint</i>	171
5.3.2.4.1.1 Componentes	171
5.3.2.4.1.2 Variables	173
5.3.2.4.1.3 Funciones.....	173
5.3.2.4.1.3.1 Evento <i>BeginPlay</i>	174
5.3.2.4.1.3.2 Atacar	174
5.3.2.4.1.3.3 Recibir impacto bala.....	176
5.3.2.4.1.3.4 Restar vida	177
5.3.2.4.1.3.5 Restar vida piernas.....	178
5.3.2.4.1.3.6 Activar <i>Ragdoll</i>	178
5.3.2.4.1.3.7 Activar sonido	178
5.3.2.4.1.3.8 Desactivar sonido	178
5.3.2.4.1.3.9 Detectar si el sonido ha terminado	179
5.3.2.4.2 <i>Animation Blueprint</i>	179
5.3.2.4.2.1 Variables	179
5.3.2.4.2.2 <i>Event Graph</i>	180
5.3.2.4.2.2.1 <i>Update</i>	180

5.3.2.4.2.2 Notificación ataque zombi.....	180
5.3.2.4.2.3 Notificación <i>ragdoll</i>	180
5.3.2.4.2.3 State Machine (<i>Animation</i>)	181
5.3.2.4.2.4 <i>Notify Animation</i>	182
5.3.2.4.2.4 Notificación en animación de ataque (zombi de pie)	182
5.3.2.4.2.4 Notificación en animación de ataque (zombi en el suelo)	182
5.3.2.4.2.4 Notificación en animación muerte 1	183
5.3.2.4.2.4 Notificación en animación muerte 2	184
5.3.2.4.2.4 Notificación en animación muerte con el zombi en el suelo.....	184
5.3.2.4.3 IA.....	185
5.3.2.5 Blueprint Partida	187
5.3.2.5.1 Variables.....	188
5.3.2.5.1 <i>Spawn</i> enemigos.....	189
5.3.2.5.1 Comienzo de nueva ronda.....	192
5.3.2.5.1 Ranking.....	193
5.3.3 Desarrollo Menús y HUD.....	195
5.3.3.1 Menús.....	196
5.3.3.1.1 Menú Principal.....	196
5.3.3.1.2 Pantalla Ranking	198
5.3.3.2 HUD	199
5.3.3.2.1 HUD Blueprint.....	199
5.3.3.2.2 Widget (BasicAmmo).....	202
5.3.4 Arte en el videojuego.....	205
5.3.4.1 Gráficos 2D.....	205
5.3.4.1.1 Botones	206
5.3.4.1.2 Símbolos máquinas de mejora.....	206
5.3.4.1.3 Símbolos power-ups.....	208
5.3.4.2 Modelado 3D	209
5.3.4.2.1 Tubo de luz.....	210
5.3.4.2.2 Maquina de mejorar arma	211
5.3.4.2.3 Monitor de doble pantalla	211
5.3.4.3 Optimización modelos 3D.....	212
5.3.4.4 Texturas y materiales	216
5.3.4.5 Arreglando UVs de modelos.....	221
5.3.5 Creación de nivel de juego	227
5.3.5.1 Diseño del nivel	227
5.3.5.2 Sonido e iluminación	232
6. Conclusiones.....	235
7. Anexo: Escenario	237
8. Bibliografía.....	268

9. Créditos	273
--------------------------	------------

Índice de Figuras

Figura 1: Battlefield como ejemplo de FPS	29
Figura 2: Nazi Zombis – Call of Duty: World at War	31
Figura 3: Zombis – Call of Duty Black Ops.....	31
Figura 4: Zombis - Call of Duty Black Ops 2	32
Figura 5: Zombis - Call of Duty: Black Ops 3	32
Figura 6: Dead Island	33
Figura 7: Left 4 Dead 2	34
Figura 8: Resident Evil 7.....	35
Figura 9: Resident Evil 7 - Ambientación.....	35
Figura 10: Resident Evil 7 - Ambientación 2	36
Figura 11: Trello del proyecto	41
Figura 12: Email semanal de Toggl	42
Figura 13: Toggl del proyecto	43
Figura 14: Ramas de GitHub del proyecto	44
Figura 15: Commits a rama específica (Apuntar).....	45
Figura 16: Llevando versiones funcionales de otras ramas a Master.....	45
Figura 17: Motores de juegos actuales	47
Figura 18: Tabla motores de juego actuales (1/2).....	48
Figura 19: Tabla motores de juego actuales (2/2).....	49
Figura 20: Logo Cry Engine.....	50
Figura 21: Logo Unreal Engine 4	51
Figura 22: Blueprint vs Kismet.....	51
Figura 23: Logo Unity.....	52
Figura 24: Logo Blender	54
Figura 25: Logo 3DS Max.....	55
Figura 26: Logo Maya.....	55
Figura 27: Logo Mudbox	56
Figura 28: Logo Substance Painter	56
Figura 29: Descriptores clasificación PEGI	61
Figura 30: Clasificación PEGI de PostWar:Hopeless Humanity	62
Figura 31: Flujo de pantallas del juego	62
Figura 32: Controles de Postwar: Hopeless Humanity (teclado)	66
Figura 33: Controles de Postwar: Hopeless Humanity (mando)	66
Figura 34: Foto del jugador en el escenario con un arma equipada	68
Figura 35: Vista aérea del escenario	73

Figura 36: Menú del juego en Photoshop.....	76
Figura 37: Menú de pausa en juego	77
Figura 38: Pantalla muerte sin ranking	78
Figura 39: Pantalla muerte con ranking	78
Figura 40: Menú ranking en Photoshop	79
Figura 41: Menú opciones en Photoshop	80
Figura 42: HUD información constante	81
Figura 43: HUD información dinámica	82
Figura 44: Camara FPS	82
Figura 45: Relacion entre clases	88
Figura 46: Contruction Script del arma AK-47	91
Figura 47: Función inicializar al que llaman los constructores de las armas	91
Figura 48: Ejemplo de Macro.....	93
Figura 49: Ejemplo llamada a macro desde evento personalizado	93
Figura 50: Ejemplo Spot Light	96
Figura 51: Ejemplo Point Light	96
Figura 52: Ejemplo Directional Light	97
Figura 53: Ejemplo Sky Light	97
Figura 54: Ejemplo de una sala del proyecto con iluminación Spot Light y Point Light.....	98
Figura 55: Ejemplo 1 iluminación únicamente con Spot Light sacada del proyecto	98
Figura 56: Ejemplo iluminación únicamente con Spot Light sacada del proyecto	99
Figura 57: Material simple y su aplicación en el proyecto.....	101
Figura 58: Material complejo (simulación de agua)	102
Figura 59: Aplicación de material complejo en el proyecto	103
Figura 60: Ejemplo Sound Cue - Imagen sacada de la documentación oficial de Unreal Engine 4	105
Figura 61: Ejemplo blending animation del proyecto	107
Figura 62: Componentes Bluerpint personaje	110
Figura 63: Tabla de variables del personaje	114
Figura 64: Tabla de funciones del personaje	118
Figura 65: Función apuntar	118
Figura 66: Grafica TimeLine (interpolación entre transformaciones)	120
Figura 67: Función equipar arma.....	120
Figura 68: Ejemplo de socket en el editor de Ureal Engine 4.	121
Figura 69: InputAction cambiar de arma	122
Figura 70: InputAction disparar.....	123
Figura 71: InputAction recibir daño.....	124

Figura 72: Función restar vida	125
Figura 73: Función suma vida	126
Figura 74: Animation Starter Pack de Unreal.....	127
Figura 75: Retargeting Animation del proyecto	128
Figura 76: Animaciones en primera y tercera persona del proyecto.....	129
Figura 77: Corrigiendo animación desde el editor de Unreal.....	130
Figura 78: Animation Blueprint personaje	130
Figura 79: Componentes WeaponParentDrop Blueprint.....	131
Figura 80: Table variables WeaponParentDrop.....	132
Figura 81: Constructor AKDrop	133
Figura 82: Constructor UMPDrop	133
Figura 83: Funcion inicializar WeaponParentDrop	134
Figura 84: WeaponParentDrop evento entrar en zona de compra	134
Figura 85: Mostrando información compra munición arma.....	135
Figura 86: Mostrando información comprar arma	136
Figura 87: WeaponParentDrop evento salir de zona de compra.....	136
Figura 88: InputAction Interactuar WeaponDrop	137
Figura 89: Función comprar arma.....	138
Figura 90: Función comprar munición.....	139
Figura 91: Función que comprueba si ya hemos comprado un arma	139
Figura 92: Función información compra (izquierda) y función información compra munición (derecha).....	140
Figura 93: Tabla variables WeaponParent Bluerpint	142
Figura 94: Constructor AK.....	143
Figura 95: Constructor M60	143
Figura 96: Función inicializar arma	143
Figura 97: Input Action Fire (evento disparar).....	144
Figura 98: Event Shoot de la clase AK	145
Figura 99: Function CreateProjectil (parte 1/2).....	146
Figura 100: Function CreateProjectil (parte 2/2)	147
Figura 101: Función BalaTransform.....	149
Figura 102: Función getRayCastPos.....	150
Figura 103: Función DondeImpacto	151
Figura 104: Bullet Projectil Actor	152
Figura 105: InputAction Recargar desde la clase FirstPersonCharacter	153
Figura 106: Función recargar (parte 1/4)	153
Figura 107: Función recargar (parte 2/4)	154

Figura 108: Función recargar (parte 3/4)	155
Figura 109: Función recargar (parte 4/4)	155
Figura 110: Función getAnimRecarga	156
Figura 111: Función mejorar daño.....	158
Figura 112: Función mejorar munición máxima	158
Figura 113: Función mejorar capacidad cargador	159
Figura 114: Función mejorar cadencia.....	159
Figura 115: Función mejorar tiempo recarga	160
Figura 116: Componentes InteractiveParent	161
Figura 117: Tabla variables InteractiveParent Blueprint.....	162
Figura 118: Función inicializar InteractiveParent Bluerpint	162
Figura 119: Evento entrar zona para interactuar	163
Figura 120: Evento salir de zona de interactuar	163
Figura 121: InputAction Interactuar	164
Figura 122: Función getPrecioActual	164
Figura 123: Constructor Puerta Blueprint	165
Figura 124: Función AbrirPuerta.....	166
Figura 125: Constructor power-up resistencia.....	167
Figura 126: Función Interactúo (clase Power-upResistencia)	167
Figura 127: Constructor power-up revivir.....	168
Figura 128: Función Interactúo (clase Power-upRevivir)	168
Figura 129: Constructor power-up tiempo recarga	168
Figura 130: Función Interactúo (clase Power-upTiempoRecarga)	169
Figura 131: Constructor maquina mejorar cadencia	169
Figura 132: Constructor maquina mejorar capacidad cargador	170
Figura 133: Función Interactúo maquina mejorar cadencia	170
Figura 134: Función Interactúo maquina mejorar capacidad cargador	170
Figura 135: Función getPrecioActual (común a todas las máquinas de mejora).....	170
Figura 136: Enemigo.....	171
Figura 137: Compoenentes Zombi Blueprint	172
Figura 138: Tabla variables Zombi Blueprint	173
Figura 139: Evento atacar	174
Figura 140: Ejemplo del ataque de un zombi	176
Figura 141: Evento recibir impacto	177
Figura 142: Función RestaVida de la clase Zombi	177
Figura 143: Funcion RestaVidaPiernas.....	178
Figura 144: Tabla variables Animation Blueprint	180

Figura 145: Evento Update Animation Blueprint	180
Figura 146: Máquina de estados animación	181
Figura 147: Notificación animación ataque (zombi de pie)	182
Figura 148: Notificación animación ataque (zombi en el suelo)	183
Figura 149: Notificación en animación de muerte 1	183
Figura 150: Notificación en animación de muerte 2	184
Figura 151: Notificación en animación muerte zombi en el suelo	185
Figura 152: Vista del NavMeshVolume en el escenario	186
Figura 153: Behaviour Tree Zombi	187
Figura 154: Tabla variables Partida Blueprint	189
Figura 155: Ejemplo IASpawn y ZonaBP	190
Figura 156: Evento BeginPlay (clase ZonaBP)	190
Figura 157: Evento BeginOverlap (clase ZonaBP).....	191
Figura 158: Función NuevaZona (clase Partida)	191
Figura 159: Función SpawnZombie (clase Partida).....	192
Figura 160: Evento ZombiMuerto clase Partida.....	192
Figura 161: Función ComienzaRonda clase Partida	193
Figura 162: Función ActualizaRanking (parte 1/4)	193
Figura 163 Función ActualizaRanking (parte2/4) somos los primeros.....	194
Figura 164: Función ActualizaRanking (parte3/4) somos segundos.....	194
Figura 165: Función ActualizaRanking (parte 3/4) somos terceros.....	195
Figura 166: Menú principal (sin ocultar nada)	196
Figura 167: Menú principal que vemos en el juego	197
Figura 168: Evento llamado al pulsar el botón ranking desde el menú principal	197
Figura 169: Menú ranking	198
Figura 170: Pantalla ranking final de juego (desde editor)	199
Figura 171: Creación del Widget del HUD	200
Figura 172: Tabla variables HUD Blueprint	201
Figura 173: Tabla funciones HUD Blueprint	202
Figura 174: Widget HUD	202
Figura 175: Función encargada de actualizar la sangre de la pantalla respecto a la vida del jugador	203
Figura 176: Tabla funciones BasicAmmo Widget.....	204
Figura 177: Botón aceptar	206
Figura 178: Botón aceptar con hover	206
Figura 179: Botón empezar partida.....	206
Figura 180: Botón empezar partida hover	206

Figura 181: Símbolo mejorar cadencia	206
Figura 182: Símbolo mejorar daño	207
Figura 183: Símbolo mejorar munición arma.....	207
Figura 184: Símbolo mejorar munición máxima	207
Figura 185: Símbolo mejorar tiempo recarga	208
Figura 186: Símbolos power-up	208
Figura 187: Ejemplo power-up titan	209
Figura 188: Modelo de luz en 3DSMax	210
Figura 189: Modelo de luz in game	210
Figura 190: Modelo máquina mejorar arma en 3DSMax.....	211
Figura 191: Modelo máquina mejorar arma en Unreal Engine 4	211
Figura 192: Modelo monitor con doble pantalla en 3DSMax	212
Figura 193: Modelos monitor con doble pantalla in game.....	212
Figura 194: Modelo toalla sin optimizar	213
Figura 195: Modelo toalla optimizado	214
Figura 196: Modelo cojín sin optimizar	214
Figura 197: Modelo cojín después de optimizar	215
Figura 198: Modelo de cepillo de dientes antes y después de ser optimizado	215
Figura 199: Ejemplo mapa difuso.....	216
Figura 200: Ejemplo mapa de normales	216
Figura 201: Ejemplo mapa especular	217
Figura 202: Ejemplo mapa de oclusión	217
Figura 203: Material suelo de ducha	218
Figura 204: Aplicación del material de ducha	219
Figura 205: Valla metalica sin mascara	219
Figura 206: Mapa difuso desde Photoshop	220
Figura 207: Creación de máscara desde Photoshop	220
Figura 208: Máscara aplicada al objeto.....	220
Figura 209: Modelos con UVs malos.....	221
Figura 210: Mostrando los UVs desde Unreal	222
Figura 211: Mostrando UVs (malos) desde 3DS Max	222
Figura 212: Mostrando UV arreglado desde 3DS Max.....	223
Figura 213: Mostrando UV arreglado desde Unreal	223
Figura 214: Modelo de inodoro con UVs arreglados.....	224
Figura 215: Visualizacion del modelo sin UV	224
Figura 216: Muestra del UV desde Unreal.....	225
Figura 217: Creación del UV desde 3DS Max	225

Figura 218: Visualización del UV creado en Unreal	226
Figura 219: Visualizacion del modelo con el UV ya creado	226
Figura 220: Uso del obstáculo central.....	228
Figura 221: Primer plano del escenario.....	229
Figura 222: Primer boceto del escenario	229
Figura 223: Comienzo del vestuario	230
Figura 224: Resultado final vestuario	230
Figura 225: Comienzo habitaciones.....	231
Figura 226: Resultado final habitaciones	231
Figura 227: Ejemplo sonidos en el escenario	232
Figura 228: Ejemplo iluminación sala de control	233
Figura 229: Ejemplo iluminación exterior	233
Figura 230: Ejemplo 2 iluminación exterior.....	234
Figura 231: Captura de Postwar: Hopeless Humanity InGame número 1.....	237
Figura 232:Captura de Postwar: Hopeless Humanity InGame número 2	238
Figura 233: Captura de Postwar: Hopeless Humanity InGame número 3.....	238
Figura 234: Captura de Postwar: Hopeless Humanity InGame número 4.....	239
Figura 235: Captura de Postwar: Hopeless Humanity InGame número 5.....	239
Figura 236: Captura de Postwar: Hopeless Humanity InGame número 6.....	240
Figura 237: Captura de Postwar: Hopeless Humanity InGame número 7.....	240
Figura 238: Captura de Postwar: Hopeless Humanity InGame número 8.....	241
Figura 239: Captura de Postwar: Hopeless Humanity InGame número 9.....	241
Figura 240: Captura de Postwar: Hopeless Humanity InGame número 10.....	242
Figura 241: Captura de Postwar: Hopeless Humanity InGame número 11.....	242
Figura 242: Captura de Postwar: Hopeless Humanity InGame número 12.....	243
Figura 243: Captura de Postwar: Hopeless Humanity InGame número 13.....	243
Figura 244: Captura de Postwar: Hopeless Humanity InGame número 14.....	244
Figura 245: Captura de Postwar: Hopeless Humanity InGame número 15.....	244
Figura 246: Captura de Postwar: Hopeless Humanity InGame número 16.....	245
Figura 247: Captura de Postwar: Hopeless Humanity InGame número 17.....	245
Figura 248: Captura de Postwar: Hopeless Humanity InGame número 18.....	246
Figura 249: Captura de Postwar: Hopeless Humanity InGame número 19.....	246
Figura 250: Captura de Postwar: Hopeless Humanity InGame número 20.....	247
Figura 251: Captura de Postwar: Hopeless Humanity InGame número 21.....	247
Figura 252: Captura de Postwar: Hopeless Humanity InGame número 22.....	248
Figura 253: Captura de Postwar: Hopeless Humanity InGame número 23.....	248
Figura 254: Captura de Postwar: Hopeless Humanity InGame número 24.....	249

Figura 255: Captura de Postwar: Hopeless Humanity InGame número 25.....	249
Figura 256: Captura de Postwar: Hopeless Humanity InGame número 26.....	250
Figura 257: Captura de Postwar: Hopeless Humanity InGame número 27.....	250
Figura 258: Captura de Postwar: Hopeless Humanity InGame número 28.....	251
Figura 259: Captura de Postwar: Hopeless Humanity InGame número 29.....	251
Figura 260: Captura de Postwar: Hopeless Humanity InGame número 30.....	252
Figura 261: Captura de Postwar: Hopeless Humanity InGame número 31.....	252
Figura 262: Captura de Postwar: Hopeless Humanity InGame número 32.....	253
Figura 263: Captura de Postwar: Hopeless Humanity InGame número 33.....	253
Figura 264: Captura de Postwar: Hopeless Humanity InGame número 34.....	254
Figura 265: Captura de Postwar: Hopeless Humanity InGame número 35.....	254
Figura 266: Captura de Postwar: Hopeless Humanity InGame número 36.....	255
Figura 267: Captura de Postwar: Hopeless Humanity InGame número 37.....	255
Figura 268: Captura de Postwar: Hopeless Humanity InGame número 38.....	256
Figura 269: Captura de Postwar: Hopeless Humanity InGame número 39.....	256
Figura 270: Captura de Postwar: Hopeless Humanity InGame número 40.....	257
Figura 271: Captura de Postwar: Hopeless Humanity InGame número 41.....	257
Figura 272: Captura de Postwar: Hopeless Humanity InGame número 42.....	258
Figura 273: Captura de Postwar: Hopeless Humanity InGame número 43.....	258
Figura 274: Captura de Postwar: Hopeless Humanity InGame número 44.....	259
Figura 275: Captura de Postwar: Hopeless Humanity InGame número 45.....	259
Figura 276: Captura de Postwar: Hopeless Humanity InGame número 46.....	260
Figura 277: Captura de Postwar: Hopeless Humanity InGame número 47.....	260
Figura 278: Captura de Postwar: Hopeless Humanity InGame número 48.....	261
Figura 279: Captura de Postwar: Hopeless Humanity InGame número 49.....	261
Figura 280: Captura de Postwar: Hopeless Humanity InGame número 50.....	262
Figura 281: Captura de Postwar: Hopeless Humanity InGame número 51.....	262
Figura 282: Captura de Postwar: Hopeless Humanity InGame número 52.....	263
Figura 283: Captura de Postwar: Hopeless Humanity InGame número 53.....	263
Figura 284: Captura de Postwar: Hopeless Humanity InGame número 54.....	264
Figura 285: Captura de Postwar: Hopeless Humanity InGame número 55.....	264
Figura 286: Captura de Postwar: Hopeless Humanity InGame número 56.....	265
Figura 287: Captura de Postwar: Hopeless Humanity InGame número 57.....	265
Figura 288: Captura de Postwar: Hopeless Humanity InGame número 58.....	266
Figura 289: Captura de Postwar: Hopeless Humanity InGame número 59.....	266
Figura 290: Captura de Postwar: Hopeless Humanity InGame número 60.....	267

1. Introducción

Aunque existe un debate acerca de quién creó el primer videojuego, es el “*Tennis for Two*” el considerado primer videojuego y data del año 1958. El juego constaba de una línea horizontal que representaba el campo de juego y otra pequeña línea vertical que era la red. Los jugadores tenían que dirigir la bola y golpearla para lo cual se usaba un oscilador a modo de mando y un monitor conectado a una computadora analógica. Desde este inicio han pasado poco más de cincuenta años y el cambio en la industria del videojuego ha sido brutal, tanto que ha logrado alcanzar en apenas medio siglo de historia no solo el estatus de medio artístico sino también el favor de un público cada vez más numeroso.

Hasta hace unos pocos años los videojuegos eran el entretenimiento de una minoría, pero a medida que la tecnología ha ido avanzando y la calidad ha aumentado, se han ido haciendo más populares sobre todo entre los jóvenes, responsables en parte de la evolución de los videojuegos y de la revolución tecnológica que estos han vivido en los últimos tiempos.

Ahora bien, 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 se han vuelto accesibles para un público que no para de crecer. 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 ellos.

Por otra parte, las mejoras en los videojuegos también han supuesto una evolución en sus géneros y sus objetivos, que no se limitan solo a contar una historia sino que amplían su aplicación a distintos campos de la enseñanza. Actualmente hay una gran lista (deportes, estrategias, aventuras, disparos,...).

El género principal de Postwar:Hopeless Humanity 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.

Asimismo, este proyecto introduce una estética que se basa en el género “*Survival Horror*”, caracterizado 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.

Esta memoria 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 Postwar: Hopeless Humanity.
- **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 encontrará toda la información tanto del videojuego como de los personajes, historia, mecánicas, diseño de niveles, etc.; por último, 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

Con el fin de facilitar la comprensión del presente proyecto, es necesario aclarar conceptos que están en la base del mismo, conceptos como el de videojuego o el de género FPS, pues no tiene mucho sentido exponer el desarrollo de un videojuego de disparos en primera persona si no se tienen bien claros ambos términos. Además también se analizarán algunos videojuegos de este estilo que han servido como referencia a la hora de desarrollar Postwar: Hopeless Humanity.

2.1 ¿Qué es un videojuego?

Es difícil dar una única respuesta a esta pregunta, puesto que con la evolución de los videojuegos su definición también ha ido cambiando, como demuestran estas definiciones 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 vídeo 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. 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 tras año, una de las principales industrias del arte y entretenimiento”

“Se define como Videojuego (también conocidos como Juego de Vídeo) a toda aplicación o *software* que ha sido creado con el fin del entretenimiento”

Si bien las definiciones son bastante parecidas unas a otras, hay aspectos en los que no existe consenso. Por ejemplo, en la última definición dice que *un videojuego es creado con el fin del entretenimiento* y esta definición puede que fuese correcta antes, pero ahora los videojuegos y géneros de estos han variado tanto que hay videojuegos hechos únicamente para el aprendizaje, es decir, su fin no es el entretenimiento sino la transmisión de conocimientos.

Por otra parte, en la primera definición 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 entretenir sino el de enseñar.

Por último, si nos centramos en la segunda definición y nos fijamos 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 experimentado los videojuegos que, más allá de que algunos sigan siendo puro entretenimiento, otros muchos son capaces de narrar historias de forma creativa de la misma manera que lo haría una película o un libro, añadiendo además que el jugador se convierte en el protagonista de la historia y el que interactúa y avanza en el progreso de una trama por lo que está logrando una mayor inmersión, de ahí que los videojuegos no solo son reconocidos como entretenimiento sino también como arte.

Por lo que respecta al concepto de género de FPS cabe decir, en primer lugar, que el género juego de disparos engloba una gran cantidad de subgéneros con la característica común de permitir que el jugador controle a un personaje que, por norma general, posee/accede a un arma que puede ser disparada a voluntad. En segundo lugar, dispone de una cámara en primera persona que le proporciona una vista del mundo desde la perspectiva del personaje protagonista. Y juntando ambos rasgos 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 podría ser *Battlefield*.



Figura 1: Battlefield como ejemplo de FPS

2.2 Referencias

Aunque la estética y esencia del juego es original, para lograrlas me he basado en algunos videojuegos populares que se parecen al que quiero realizar, con tal fin he tomado como punto de partida otros trabajos que he estudiado tanto mecánica como estéticamente. Son los siguientes:

- 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, y sus posteriores versiones, son la principal referencia de Postwar: Hopeless Humanity, ya que es un FPS en el que tienes que sobrevivir el máximo número de rondas ante continuas oleadas de zombis que intentarán 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 puede luchar contra zombis alemanes, estadounidenses, rusos y asiáticos, aunque se encuentran en diferentes mapas y modos.

Hasta cuatro jugadores deben sobrevivir a infinitas hordas de ataques zombis. Se ganan puntos por matar o dañar zombis y por la reparación de las barreras a través de las cuales 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 en el número de rondas y la partida se acaba cuando todos los jugadores son abatidos o exterminados por los zombis, que se vuelven más fuertes y rápidos a medida que superan las rondas lo que obliga 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*, que son muy efectivos en distintas circunstancias.



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



Figura 3: Zombis – Call of Duty Black Ops

En este juego se incluyen varias novedades entre las que sobresalen: un motor gráfico mejorado que proporciona el doble de zombis en una partida y dos nuevos modos de juego.

Uno de estos modos de juego es **TranZit** (una historia cooperativa ambientada en un mundo en el cual los jugadores tendrán que viajar a diferentes zonas montados en un autobús fortificado). El otro es **Pena**. Este admite hasta ocho jugadores que formarán dos equipos de cuatro componentes cada uno. Los dos equipos han de sobrevivir por separado y deben sacarles ventaja a sus adversarios, es decir, 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



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

2.2.1.5 *Dead Island*

Dead Island es un videojuego de rol en *action* y un *survival horror* en un mundo abierto desarrollado por *Techland*. Se centra en el reto de la supervivencia en una isla infectada por zombis.



Figura 6: *Dead Island*

El juego comienza con una noche de fiesta en la que ocurren cosas extrañas a las cuales no les 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...). Al intentar huir del hotel, recibimos la ayuda de unos supervivientes, momento en el que nos convertimos en víctimas de un ataque zombi. Aquí comienza la historia, justo cuando te cuentan que por alguna extraña razón somos inmunes al virus que ha convertido a las personas en zombis. Desde este instante, el objetivo del juego pasa a ser la supervivencia y la huida 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*. Se trata de un *shooter* en primera persona ambientado en medio de un apocalipsis zombi. El mundo está infectado y solo quedan cuatro supervivientes que, cooperando entre ellos, buscarán escapar de los escenarios devastados por esta plaga generalizada. La clave se encuentra en la palabra **cooperación**, ya que la naturaleza de *Left 4 Dead* y esta segunda parte invitan a jugar y disfrutar al máximo con tres compañeros más, entre los que deberán acabar con miles de zombis.



Figura 7: Left 4 Dead 2

La acción 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. Todo esto sucede cuando han transcurrido tres semanas del primer brote del virus que sacude los Estados Unidos y después de que la CEDA haya endurecido las cuarentenas de las ciudades y pueblos infectados por el virus y ayude a los supervivientes a escapar de ese apocalipsis zombi.

2.2.1.7 Resident Evil VII

Resident Evil VII es un videojuego de *survival horror* desarrollado por la empresa *Capcom*. Se trata del 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 en tanto que juegos de supervivencia y dista un poco del objetivo principal de Postwar: Hopeless Humanity que es un FPS en que tendrás que sobrevivir el máximo número de rondas posibles, constituye una referencia importante para él tanto en la estética y como en la psicología del jugador, aspectos en los que me quiero basar.

La ambientación tétrica de este juego, con lugares pocos iluminados y una inquietante música de fondo, produce una permanente tensión y recrea muy acertadamente el clima en el que quiero que el jugador de Postwar: Hopeless Humanity se sienta envuelto. Además el uso de la cámara en primera persona contribuye a crear **una atmósfera pavorosa que atenaza al jugador**. Con estos recursos se consigue la sensación de terror de una manera muy inteligente al no abusar de sustos fáciles, de giros de cámara en los que te encuentras algo inesperado o de estridentes sonidos gratuitos; en su lugar, crea tensión de una forma mucho más sutil mediante la iluminación y el sonido.



Figura 8. Resident Evil 7



Figura 9: Resident Evil 7 - Ambientación



Figura 10: Resident Evil 7 - Ambientación 2

En el relato, *Ethan Winters* es atraído por un mensaje de su esposa *Mia*, desaparecida desde hace tres años, a una plantación abandonada en Dulvey, Louisiana, donde explorando una casa aparentemente abandonada Ethan encuentra a *Mia* encarcelada en el sótano. *Mia* está poseída por una fuerza desconocida y, cuando tratan de escapar de allí, ella ataca a *Ethan*, que 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 género FPS para PC. Este juego se desarrollará con el motor de videojuegos *Unreal Engine 4* y tanto la programación de mecánicas como de IA y la lógica del juego se realizarán exclusivamente con el sistema de *Scripting Visual Blueprints*.

Asimismo se realizará un análisis previo de videojuegos similares para obtener información acerca del mercado en este tipo de juegos, se realizará un diseño del videojuego que quedará plasmado en el GDD, se analizarán los diferentes motores de videojuegos y qué características aporta *Unreal Engine* para la realización de videojuegos. Al mismo tiempo, se crearán recursos para el proyecto y se utilizarán otros de terceros, siempre y cuando las licencias *Creative Commons* lo permitan.

3.2 Desglose de Objetivos

Los objetivos específicos del proyecto son los siguientes:

1. Diseñar 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. Crear un sistema de menús y HUD con el sistema de *Scripting Visual Blueprints*.
4. Realizar un pequeño estudio de los distintos juegos de este género en el mercado.
5. Despues de haber realizado durante el último curso un motor gráfico propio, poder utilizar un motor gráfico profesional como *Unreal Engine 4* para intentar exprimirlo al máximo gráficamente.
6. Explotar las ventajas del motor gráfico *Unreal Engine* para conseguir un acabado gráfico vistoso y, en la medida de lo posible, profesional.
7. Estudiar las ventajas y limitaciones que presenta *Unreal Engine* con respecto a otros motores gráficos.
8. Analizar las distintas herramientas que aporta *Unreal Engine* para el desarrollo de un videojuego de calidad.
9. Investigar y comprender el sistema de IA de *Unreal Engine* puesto que, si el juego va a ser de un jugador contra la IA, resulta necesario entender cómo funciona el sistema de IA que usa *Unreal Engine* para así poder diseñar y desarrollar la IA de Postwa: Hopeless Humanity.
10. Generar un documento de diseño para el videojuego (GDD, *Game Design Document*).
11. Crear modelados para un videojuego.
12. Texturizar los modelos realizados.

13. Importar *Assets* a *Unreal Engine* y utilizarlos correctamente.
14. Añadir sonidos al videojuego.

4. Metodología

La metodología empleada en el presente proyecto se basa en las metodologías ágiles y, dentro de estas, me he decantado por *Kanban* dadas las ventajas que he apreciado ahí. Para la gestión del proyecto, la herramienta seleccionada ha sido *Trello* debido a su capacidad para administrar de modo eficiente las tareas. Asimismo, he elegido la aplicación *Toggl* por su eficaz gestión del tiempo así como por su compatibilidad con *Trello*. Por último, Git es el *software* empleado para el control de versiones.

4.1 Metodología de desarrollo

4.1.1 Metodologías Ágiles

Por definición, las metodologías ágiles son aquellas que permiten adaptar la forma de trabajo a las condiciones del proyecto para alcanzar mayor flexibilidad y rapidez en la respuesta, y para adecuar 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 ágiles (*Scrum*, *Extreme Programming*, *Kanban*,...), con sus respectivas ventajas e inconvenientes, por lo que conviene elegir la metodología que mejor se adapte al proyecto y al equipo.

La metodología por la que nos hemos decantado para este proyecto es *Kanban* dado que es una metodología fácil de usar y de aplicar al proyecto, está pensada para equipos pequeños (en este caso el “equipo” es de una sola persona) y además ya había trabajado con ella anteriormente con muy buenos resultados. Igualmente, permite añadir nuevas tareas con rapidez y de un solo vistazo aporta gran información sobre las tareas que se están realizando, las que faltan y las que ya están hechas.

4.1.2 Kanban

El objetivo de *Kanban* es gestionar de manera general la realización de las tareas de un proyecto. Las principales reglas de *Kanban* son las siguientes:

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

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. Estos datos se escriben en *post-it* que se pegan en la pizarra y contienen información variada, como una descripción y la estimación de la duración de la tarea.

El objetivo de esta visualización es ayudar a gestionar el trabajo de forma eficiente distinguiendo entre aquel que está por realizar y el que está en curso 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 cuatro tareas en desarrollo, como máximo una en pruebas, etc. A esto se añade que a menudo 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 y 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 unidad de medida más conocida de *Kanban* es el *lead time*, normalmente se suele utilizar también otra importante: el *cycle time*. El *cycle time* computa el tiempo que se tarde en completar una tarea desde el inicio del trabajo hasta el final del mismo. Si con el *lead time* se mide lo que ven los clientes, lo que esperan, con el *cycle time* se mide más el rendimiento del proceso.

4.2 Gestión de proyecto

A pesar de que en los apartados anteriores cuando hablábamos de *Kanban* se han usado los términos *post-it* y pizarras, para la gestión del proyecto se van a utilizar herramientas virtuales, por lo tanto los *post-it* pasarán a ser tarjetas y las pizarras, tableros virtuales.

La gestión del proyecto se ha realizado con la herramienta gratuita *Trello* que se define como “*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*”.

El primer paso es la creación de las columnas que representan los distintos estados por los que puede pasar una tarjeta (tarea) y son estas:

- **TO DO:** aparecen las tareas que están por hacer y que 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 ese momento.
- **Pruebas:** se recogen las tarjetas ya desarrolladas pero que están en espera de realizar pruebas para comprobar que funcionan perfectamente antes de poder dejarlas ya en la columna de realizadas (*done*).
- **Done:** en esta columna se almacenan las tarjetas ya finalizadas y sobre las que no se tendrá que volver a trabajar.
- **Descartado:** aquí se acumulan aquellas tareas que se hayan descartado y ya no estén previstas para realizarse en el proyecto, ya sea por cuestión de tiempo o por cualquier otro motivo.
- **Ideas Futuras:** donde se colocan aquellas tareas que supondrían una mejora del juego pero que son totalmente prescindibles y, por tanto, solo se realizarán si hay tiempo suficiente.

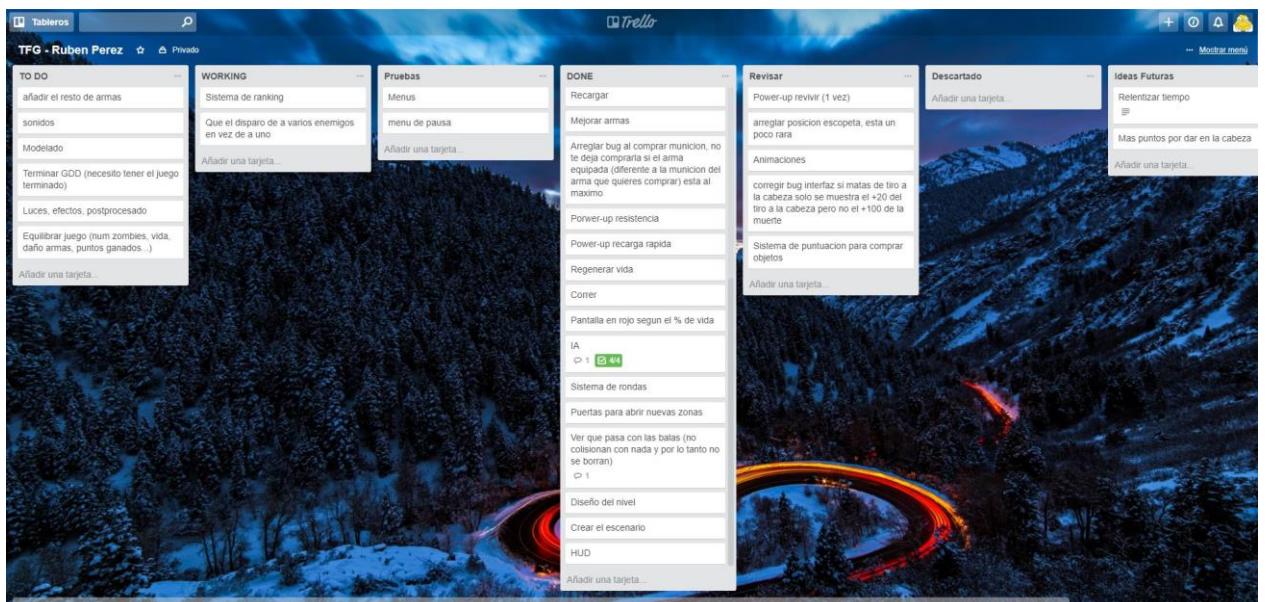


Figura 11: Trello del proyecto

4.2.1 Toggl y la compatibilidad con Trello

Toggl es 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. Su funcionamiento es tan sencillo como registrar las tareas, clasificarlas con etiquetas y pulsar el botón de inicio y de final cada vez que empezamos o finalizamos una, de esa manera disponemos de un informe con el tiempo exacto dedicado a cada tarea del proyecto.

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

Sin embargo, lo más importante de esta herramienta es su compatibilidad con *Trello* dado que existe una extensión en *Chrome* que posibilita la integración de ambas herramientas y así, desde el propio *Trello*, se puede iniciar el temporizador de *Toggl* para una tarea específica con lo que se puede tener un control total sobre las tareas que se están realizando así como del tiempo que se está invirtiendo en ellas.

Además *Trello* permite establecer fechas de vencimiento, es decir, fijada una fecha de vencimiento, cuando esta llega ves el tiempo real que dedicaste a esa tarea, lo que posibilita el 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 qué has dedicado esas horas (véase figura 12).

The screenshot shows an email from Toggl Support with the subject "Tasks tracked last week (2017-07-17 - 2017-07-23)". The body of the email contains a table of tracked tasks and their durations:

Task Description	Duration
ajustar mirilla, arreglar raycast, cambiar raycast si apuntas o no	41:48:46
Apuntar con el arma	41:48:46
arreglar animaciones personaje	04:30:08
Arreglar bug al comprar munición, no te deja comprarla si el arma equipada (diferente a la munición del arma que quierés comprar) esta al máximo	02:46:18
cambiar personaje y animación	03:11:05
Correr	00:17:28
Disparar proyectiles, controlar recarga, muzzle...	04:26:38
Hacer el cambio de arma y equipárlas	00:56:43
IA	10:49:18
Mejorar armas	01:26:17
Pantalla en rojo segun el % de vida	01:11:42
power-ups	05:41:10
prueba con proyecto 3era persona	00:54:50
Regenerar vida	03:08:19
Ver que pasa con las balas (no colisionan con nada y por lo tanto no se borran)	00:32:41
	00:53:40
	01:22:29

At the bottom of the email, there is a note: "Thanks for enrolling with Toggl! Toggl Team If you don't want to receive the weekly report please click [here](#)".

Figura 12: Email semanal de Toggl

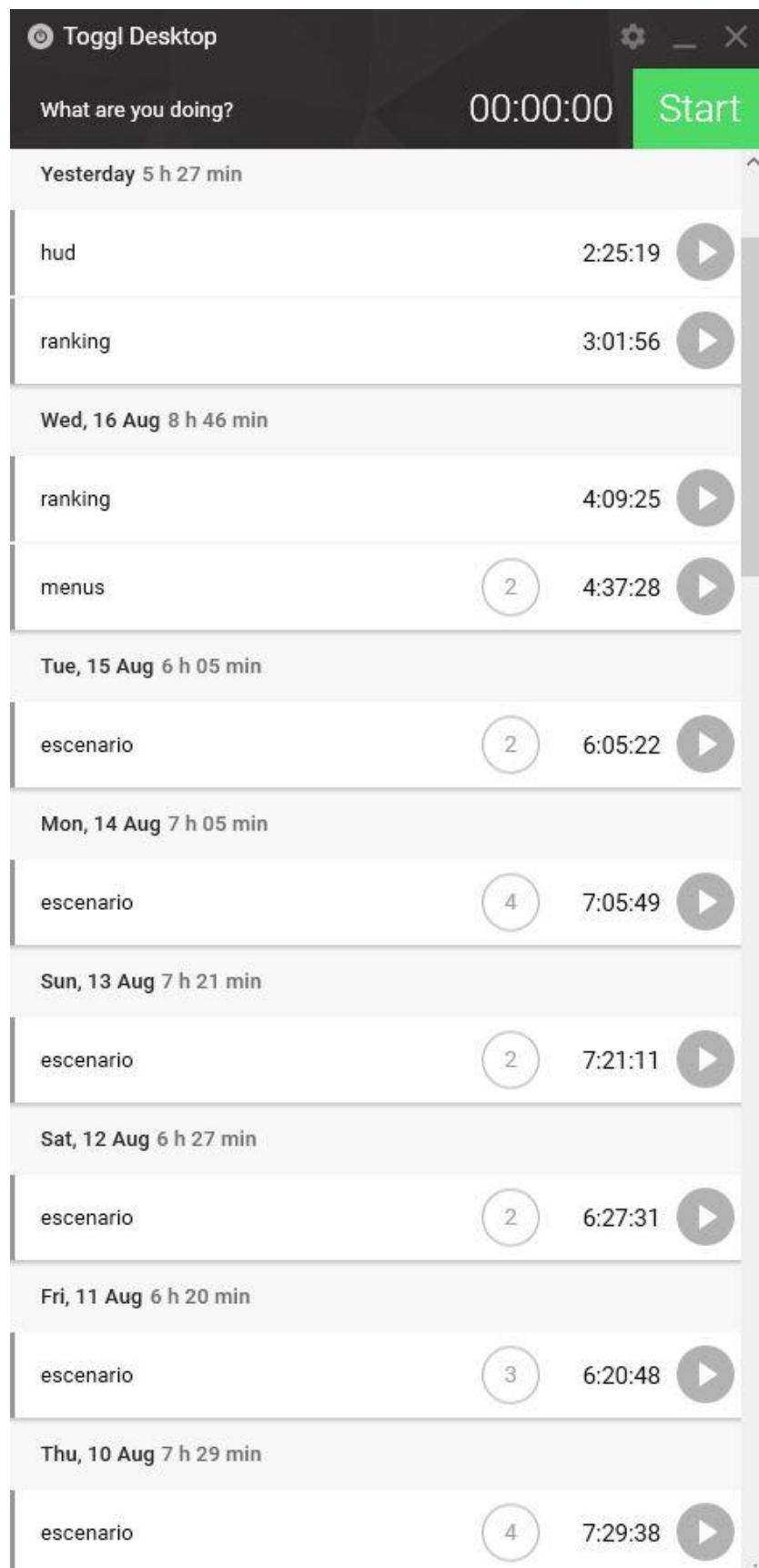


Figura 13: Toggl del proyecto

4.3 Control de versiones

Git es un control de versiones para grandes proyectos. Su empleo constituye una herramienta de gran utilidad que evita problemas en un futuro (y algún que otro susto), puesto que admite la recuperación de versiones anteriores en todo momento y así se trabaja con la tranquilidad de saber que si un día el proyecto deja de funcionar por algo siempre tenemos versiones anteriores que podemos recuperar.

Es importante el hecho de manejar varias versiones a la vez, pues mediante **Git** podemos crear diferentes ramas en las que desarrollar las distintas mecánicas o funcionalidades. Esto permite aislar cada mecánica del resto y facilita el trabajo, ya que nos centramos 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, la solución es tan sencilla como descartar esa versión, que se encuentra aislada en otra rama, sin que se dañe el resto del proyecto.

Este sistema de trabajo que se desarrolla en diferentes ramas y luego se llevan versiones funcionales a master es el que he aplicado y las siguientes imágenes del estado de *GitHub* permitirán entender su funcionamiento de manera más fácil.

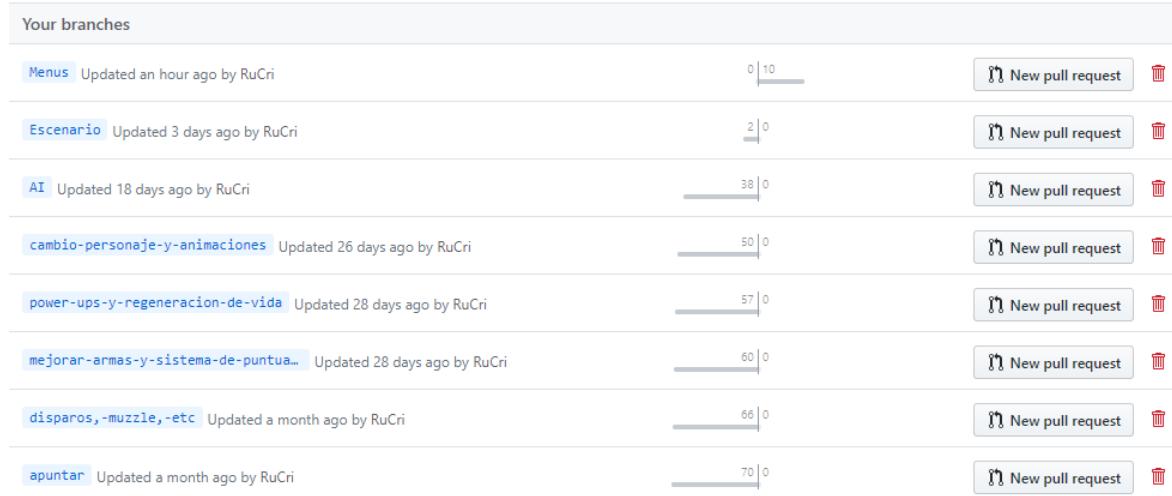


Figura 14: Ramas de GitHub del proyecto

Un plus que aporta usar Git es que te familiarizas con él y, dado que actualmente se emplea en la gran mayoría de empresas y que te pedirán que utilices Git vayas donde vayas, supone una ventaja si ya lo conoces.

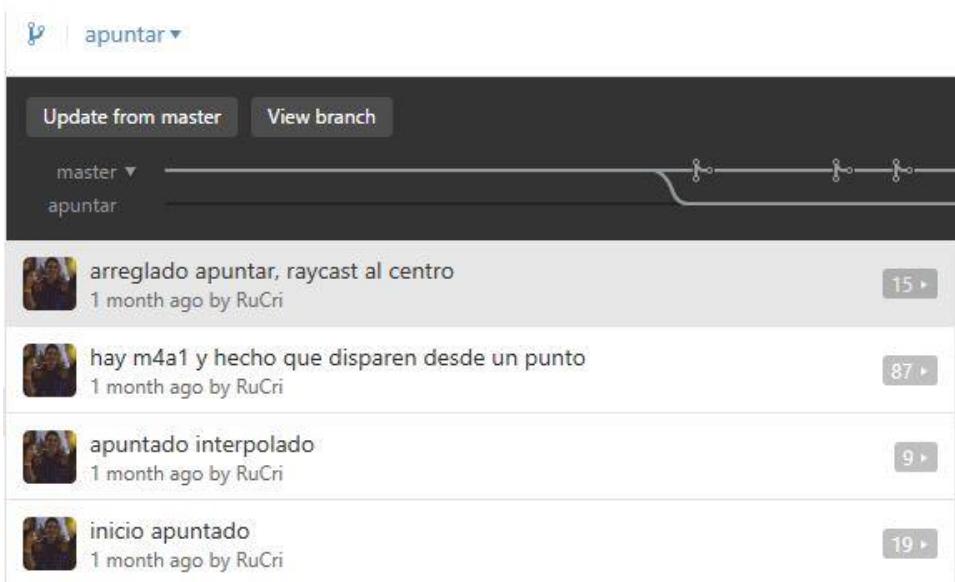


Figura 15: Commits a rama específica (Apuntar)

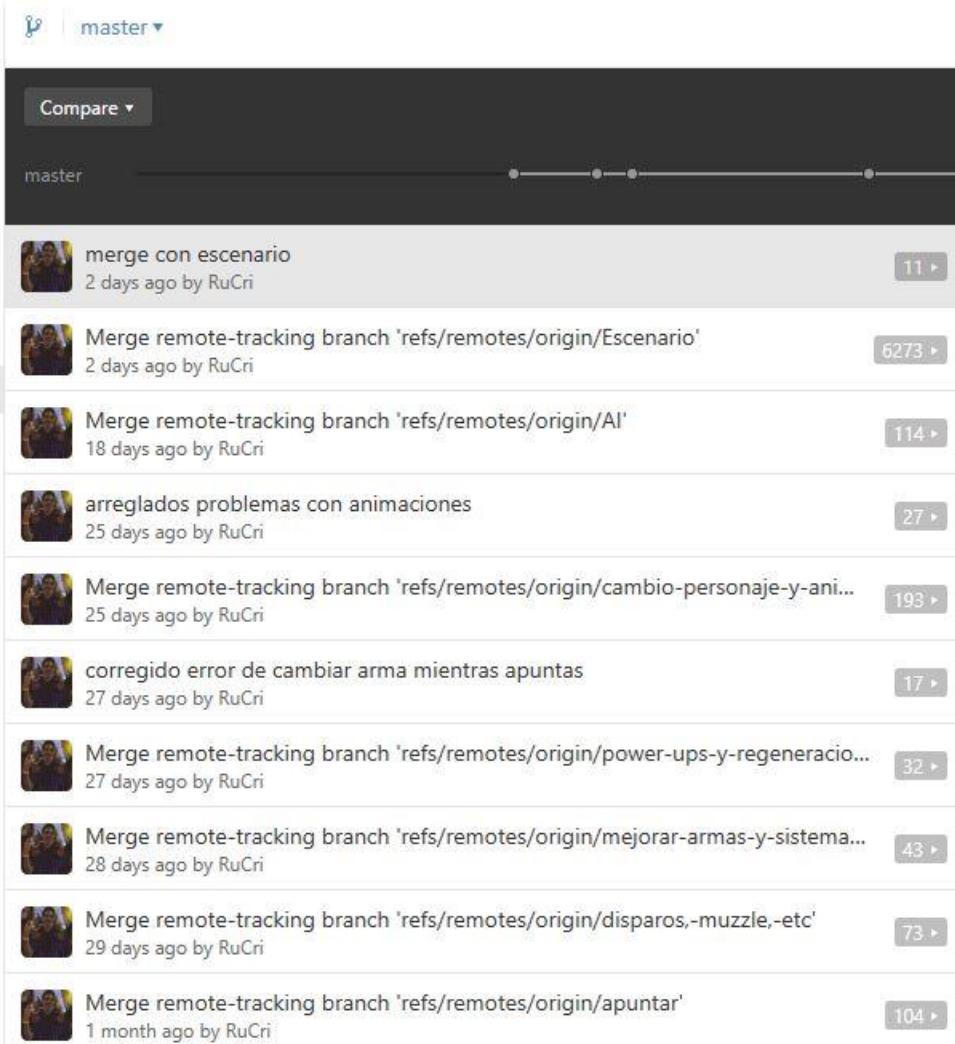


Figura 16: Llevando versiones funcionales de otras ramas a Master

5. Cuerpo del trabajo

Este bloque representa la parte sustancial del documento donde se describe con detalle el proceso completo del desarrollo del videojuego. Básicamente su propósito es responder a tres cuestiones sobre las que se vertebra el trabajo, que son:

1. La primera parte intenta responder al “**¿Con qué se va a desarrollar?**”. En esta parte se habla sobre las herramientas que se usarán para desarrollar el videojuego, un pequeño análisis de estas herramientas y cuáles han sido las seleccionadas.
2. La segunda parte intentará aclarar el “**¿Qué se va a desarrollar?**”, por ello en esta parte se incluye un **GDD (Game Design Document)**, en el que se plasmarán todos los aspectos del videojuego a desarrollar.
3. La 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 y elección de herramientas

Las herramientas disponibles para la creación de un videojuego son variadas pero no todas nos resultan igual de útiles, por tanto es necesario analizar las existentes con el fin de seleccionar aquellas que nos parezcan más adecuadas a nuestro propósito.

Por una parte, lo más importante es el motor de juego de ahí que se imponga un análisis exhaustivo del mismo, sin olvidarnos de otras herramientas necesarias para la realización del videojuego como pueden ser los programas de modelado y texturizado. Por otra parte, 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. que también requieren de un análisis detallado.

5.1.1 Motor de videojuego

5.1.1.1 Conceptos clave

Antes de estudiar los diferentes motores de videojuegos actuales, aclararemos algunos términos esenciales que constituyen el punto de partida del trabajo.

Software: 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 *Game Engine*. Es un *software* diseñado para la creación, desarrollo y representación de videojuegos. La función básica de un motor es proveer al videojuego de un motor de *renderizado* para los gráficos 2D y 3D, de un **motor físico** (encargado de detectar colisiones, realizar cálculos como velocidad, etc), de **sonidos, scripting** (programación, ya sea 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 existe una gran cantidad de motores de videojuegos, razón por la que es impensable analizarlos todos, por tanto 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 · Kinética · 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 dos imágenes) están tomadas de la Wikipedia [y](#) muestran con más detalle los motores de juegos actuales.

Nombre	Lenguaje de programación	Scripting	Múltiples - Cross-Platform	SDL	Orientación 2D/3D	Plataforma
3D Rad	C#	AngelScript	✗ No	✗ No	✓ 3D	Windows
Adventure Game Studio	C++	AGScript	✓ 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 ¹⁸	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	Windows
Core3D ¹⁹	Objective-C		✓ Sí	✓ Sí	✓ 3D	Windows Linux OS X iOS
CRM32Pro SDK ²⁰	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 ²¹	C++	Lua	✓ Sí	✗ No	✗ 2D	Windows, OS X, Android, iOS, Flash
Duality ²²	C#	Plugin-based	✗ No	✗ No	✗ 2D	Windows
Eclipse Origins ²³	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 ²⁴	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 Xbox One PlayStation 4 PlayStation 3 PlayStation Vita HTML5 Tizen
GamePlay3D ²⁵	C++	Lua	✓ Sí	✗ No	✓ 3D	Windows Linux OS X iOS BlackBerry 10 Android
Gamvas	JavaScript	JavaScript	✓ Sí	✗ No	✗ 2D	HTML5
Grit ²⁶	C++	Lua	✓ Sí	✗ No	✓ 3D	
Haa's Game Engine (HGE) ²⁷	C++	C, Go	✓ Sí	✓ Sí	✗ 2D	Windows Linux OS X
HGamer3D ²⁸	Haskell			✗ No	✓ 3D	
Horde3D ²⁹	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)

id Tech 3	C	Game Data [PK3]	✓ Sí	✗ No	✓ 3D	Windows Linux OS X
id Tech 4	C++	via DLLs	✓ Sí	✗ No	✓ 3D	Windows Linux OS X
IndieLib	C++		✓ Sí	✓ Sí	✗ 2.5D	Windows Linux OS X
ioquake3	C		✓ Sí	✓ Sí	✓ 3D	Windows Linux OS X
IwGame Engine	C++		✓ Sí	✗ No	✗ 2D	Windows Linux OS X iOS Android
Jake2	Java		✓ Sí	✗ No	✓ 3D	Cross-platform
JGame	Java, Actionscript 3		✓ Sí	✗ No	✗ 2D	J2ME Android
jMonkeyEngine	Java		✓ Sí	✓ Sí	✓ 3D	Cross-platform
Jogre	Java		✓ Sí	✗ No	✗ 2D	Cross-platform
JPCT and JPCT-AE	Java		✓ Sí	✗ No	✓ 3D	Java Android
Kobold2D	Objective-C	Lua	✓ Sí	✗ No	✗ 2D	OS X iOS
Libgdx	Java		✓ Sí	✗ No	✓ 3D	Windows Linux OS X iOS Android HTML5
Linderdaum Engine	C++	C#, LinderScript	✓ Sí	✗ No	✓ 3D	Windows OS X iOS Android BlackBerry 10
LOVE	Lua	Lua	✓ Sí	✓ Sí	✗ 2D	Windows Linux OS X
LWJGL	Java		✓ Sí	✓ Sí	✓ 3D	
Maratis	C++	Lua	✓ Sí	✗ No	✓ 3D	Windows Linux OS X iOS Android
melonJS	Javascript	Javascript	✓ Sí	✗ No	✗ 2.5D	HTML5
Moai SDK	C++	Lua	✓ Sí	✓ Sí	✗ 2D	Windows OS X iOS Android
Multiverse Foundation	Jython and Java	Python	✗ No	✗ No	✓ 3D	
Nebula Device	C++	Java, Python, Lua, Tcl/Tk	✓ Sí	✗ No	✗ 2.5D	Windows Linux
NetGore	C#		✓ Sí	✗ No	✗ 2D	Windows Linux
NME	Haxe		✓ Sí	✗ No	✗ 2D	Windows Linux OS X iOS Android BlackBerry
nxPascal	Object Pascal	Delphi, Lazarus	✓ Sí	✗ No	✓ 3D	
OpenSimulator	C#	LSL	✓ Sí	✗ No	✓ 3D	
ORX	C/C++	Custom	✓ Sí	✓ Sí	✗ 2.5D	Windows Linux Mac OS X iOS Android
Oxygine	C++		✓ Sí	✓ Sí	✗ 2D	Windows Linux Mac OS X iOS Android
Panda3D	C++	Python	✓ Sí	✗ No	✓ 3D	Windows Linux OS X iOS
PixelLight	C++	AngelScript, Lua, Python, Javascript/V8	✓ Sí	✗ No	✓ 3D	Windows Linux Android
PLIB	C++		✓ Sí	✓ Sí	✓ 3D	Windows Linux OS X
Polycode	C++	Lua	✓ Sí	✓ Sí	✓ 3D	Windows Linux OS X
PowerBite PackApp Engine	Ninguno	Ninguno	✓ Sí	✗ No	✗ No	Windows
Pyrogenesis	C++	JavaScript	✓ Sí	✓ Sí	✓ 3D	Windows
Retribution Engine	C++		✗ No	✗ No	✓ 3D	Windows
SFML	C++		✓ Sí	✗ No	✗ 2D	
Sge2d	C		✓ Sí	✓ Sí	✗ 2D	cross-platform
Source	C++		✓ Sí	✓ Sí	✗ 2D	Windows
Spring	C++, Java/JVM, Lua, Python		✓ Sí	✓ Sí	✓ 3D	Windows Linux OS X
StepMania	C++	Lua	✓ Sí	✗ No	✓ 3D	cross-platform
StormEngineC	JavaScript	JavaScript	✓ Sí	✗ No	✓ 3D	HTML5
Stratagus	C	Lua	✓ Sí	✓ Sí	✗ 2D	Linux
Torque3D	C++	TorqueScript	✗ No	✗ No	✓ 3D	Windows Linux OS X
Turbulence	TypeScript	JavaScript	✓ Sí	✗ No	✓ 3D	HTML5
Unity3D	C#	C#, JavaScript, Boo	✓ Sí	✓ Sí	✓ 3D	Windows Linux OS X Android
Unreal Engine	C++	C++, UnrealScript	✓ Sí	✓ Sí	✓ 3D	Windows Linux OS X
WiMi5	HTML5	JavaScript, HTML5	✓ Sí	✗ No	✓ 2D	Windows Linux OS X cross-platform
Wire3D	C++		✓ Sí	✗ No	✓ 3D	Windows Wi
WorldForge	C++	Lua(client), Python(server)	✓ Sí	✓ Sí	✓ 3D	
ZenGL	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 de *software* Crytek. En su origen se trataba de un motor de demostración para la empresa Nvidia y, dado su 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* fueron adquiridos por la distribuidora de videojuegos *Ubisoft*.



Figura 20: Logo Cry Engine

Este motor es muy potente: posee una gran capacidad y potencia tanto en aspectos gráficos como en el cálculo de físicas, además de que consigue un increíble realismo en la iluminación de escenas en tiempo real. Sin embargo, presenta un grave inconveniente: una gran curva de aprendizaje que te exige invertir muchas horas para empezar a “dominar” el programa.

Otra ventaja de *CryEngine* es su modelo actual de negocio basado en “paga lo que quieras”. Esta opción de *royalties* libres permite obtener un producto de calidad con bajo coste (o coste 0), a diferencia del modelo anterior de 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 al final en el caso de que el juego tuviera un gran éxito.

5.1.1.2.2 Unreal Engine 4

El 5 de noviembre del 2009 Epic Games publicó una versión gratuita de *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

Unreal Engine 4 incluye mejoras con el objeto de facilitar la producción de videojuegos. Cabe destacar la incorporación de 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: antes se usaba el lenguaje *UnrealScript* (utilizado únicamente en los motores de *Unreal*), pero luego este ha sido sustituido por C++ lo que supone una gran mejora, ya que antes era difícil el comienzo en *Unreal* dado que al tiempo que aprendías un nuevo lenguaje te tenías que adaptar a un motor de juego que no es sencillo de manejar.

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

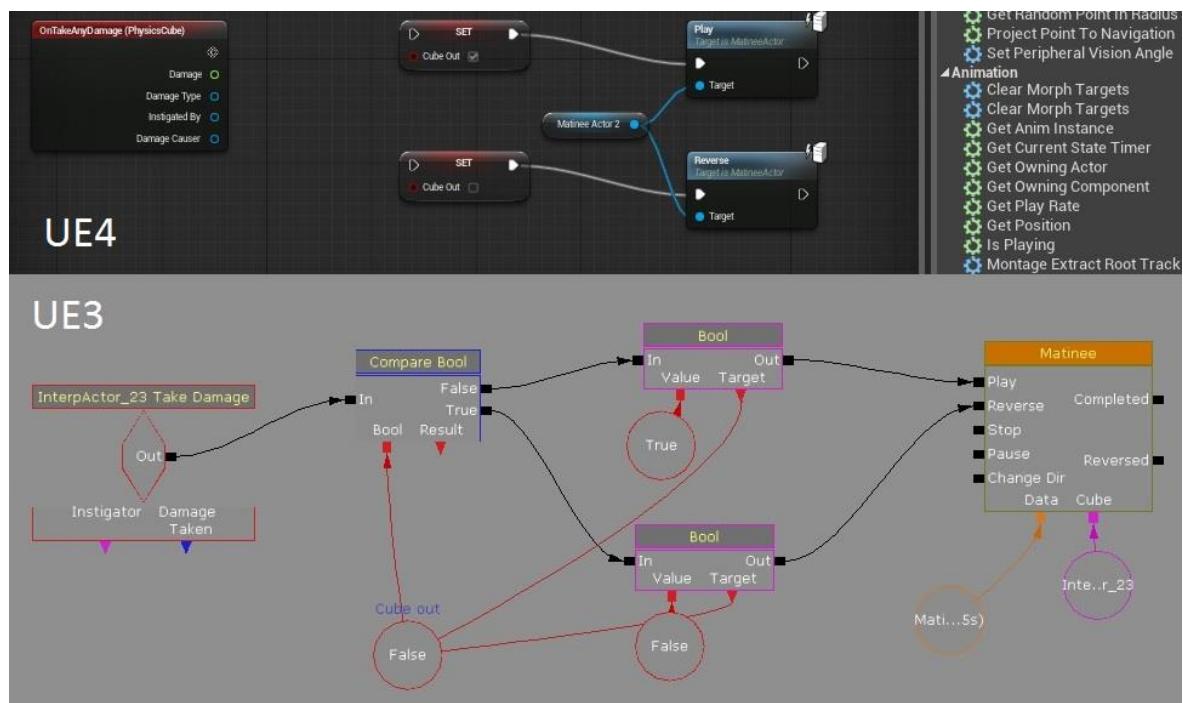


Figura 22: Blueprint vs Kismet

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*, porque permite a un desarrollador crear un juego para una gran cantidad de plataformas usando el mismo código.



Figura 23: Logo Unity

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

Otros aspectos importantes de *Unity* son, en primer lugar, que permite programar *scripts* tanto en C# como en *JavaScript* y, en segundo lugar, que es compatible con las 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 desventaja de *Unity* radica en sus dos licencias: una gratuita y una de pago, con el agravante de 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. Antes de tomar una determinación 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 valorar estos aspectos, 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*.

Una razón que avala nuestra elección es que, como el juego va a ser un FPS y *Unreal* surgió en un primer lugar para la creación de juegos FPS, este motor de juego se adapta perfectamente a las mecánicas que se quieren implementar.

Asimismo, la documentación sobre él es muy extensa, con gran cantidad de artículos, libros, y tutoriales por toda la web, a lo que cabe añadir 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 además de que se dispone de una gran cantidad de videotutoriales en *Youtube*.

Otra razón importante para su elección es el precio: *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\$ americanos.

Finalmente, una razón determinante para decantarnos por este motor es que como usuario ya estaba familiarizado con él al haber realizado dos proyectos con *Unreal Engine 4* y, aunque estos proyectos

fueran mucho más pequeños, esto me ha facilitado su manejo, sobre todo del sistema de *scripting* visual (*Blueprint*).

5.1.2 Otras herramientas

Una vez tomada la decisión más importante, que era la de elegir el motor de juego, a continuación procede escoger algunos de los programas que se utilizarán 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 y algunos programas de modelado 3D son los siguientes.

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.
- 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, de efectos especiales y de 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, la edición de mallas poligonales y la 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, la escultura no destructiva y el 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

Permite importar modelos 3D de otros programas para poder texturizarlos y 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

Ya que el modelado y texturizado no son objetivos prioritarios del proyecto, la elección de las herramientas está condicionada sobre todo por el conocimiento previo del *software* y el ahorro de tiempo que esto supone. Por lo tanto, las herramientas que voy a utilizar son *3DS Max* y *Mudbox* debido a que ya he trabajado con ellas anteriormente y a que las diferencias con las otras opciones no son relevantes, además cualquiera de las seleccionadas hace posible el modelar y texturizar algún objeto 3D.

Hay que recordar que, aunque tanto *Mudbox* como *3DS Max* son programas de pago, voy a usar la licencia de estudiante que me permite disponer del 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 utiliza como guía durante todo el proceso de creación del juego. En este documento se plasman todos los aspectos esenciales para el desarrollo del videojuego: la historia, los personajes, mecánicas, objetivo del juego, inteligencia artificial, diseño de niveles, etc.

El nombre elegido para el videojuego es **PostWar: Hopeless Humanity**.

5.2.1 El juego en términos generales

5.2.1.1 Resumen de argumento

La acción transcurre en Corea del Norte país obsesionado con expandir su poder y conquistar nuevos territorios afines al régimen de coreano. Las relaciones internacionales, ya deterioradas, no posibilitan una salida negociada del conflicto. El temor cunde entre una población que se prepara para lo peor. Se impone un clima de crispación, miedo y desesperación en los países que empiezan a desplegar su armamento nuclear.

El mundo está dividido ahora en dos regiones inmersas en una gran guerra nuclear que hacen del planeta entero una víctima de los 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, consciente de la gravedad de la situación, lucha por su supervivencia enfrentándose contra los que antes eran personas y que ahora, debido a la elevada radiación, se han convertido en muertos vivientes. Hemos sobrevivido gracias a que estábamos en un refugio nuclear desde donde dirigíamos los bombarderos, por ello conservamos la esperanza de encontrar más supervivientes aliados que estén 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 en el juego debido a la cámara en primera persona y a 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

Postwar: Hopeless Humanity se clasificaría dentro del género **FPS** ya que, como en 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 disponemos de armas que podremos usar para acabar con los enemigos.

El objetivo de este juego FPS es sobrevivir el mayor tiempo posible; sin embargo, a pesar de la influencia del género **survival**, no se podría considerar dentro de este propiamente. Por otra parte, si bien es cierto que la temática y la ambientación pretenden amedrentarnos e inquietarnos, a lo cual contribuyen los sonidos, luces y sombras propias de un lugar 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 de videojuegos por edades donde se informa a los padres o a cualquier consumidor del producto de la edad apropiada según el contenido del juego sin tener en

cuenta el nivel de dificultad de este. La etiqueta PEGI también 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.*”

En consecuencia, la clasificación de un juego confirma que es adecuado para jugadores que han cumplido una determinada edad. Así pues, un juego PEGI 7 solo será adecuado para quienes tengan 7 o más años de edad y un juego PEGI 18 solo 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 que aparecen en el reverso de los estuches y aclaran los motivos principales por los que un juego ha obtenido una categoría de edad concreta. Existen ocho descriptores:

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

Figura 29: Descriptores clasificación PEGI

5.2.1.4.4 Clasificación PEGI de Postwar: Hopeless Humanity

La clasificación de Postwar: Hopeless Humanity es de tipo PEGI 18 porque se ajusta a la descripción sobre PEGI 18 que se da en la web oficial de PEGI y que reproducimos a continuación:

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

El siguiente diagrama de flujo muestra los diferentes estados por los que se puede pasar en el juego.

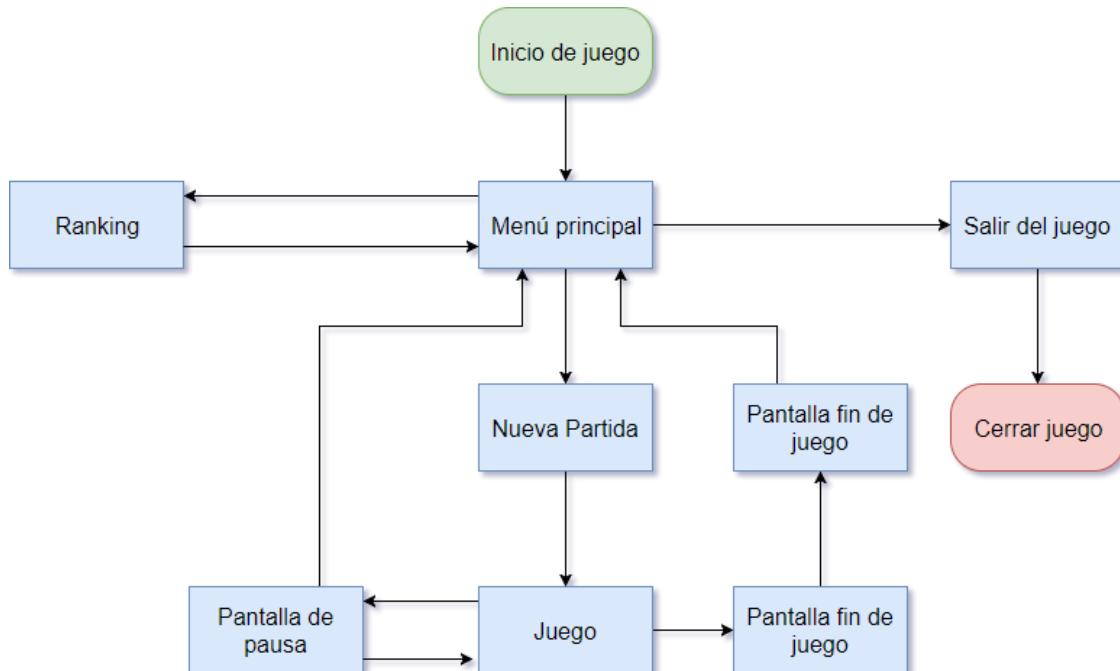


Figura 31: Flujo de pantallas del juego

Este flujo de juego consiste en:

1. Iniciar el juego: abre directamente en el menú principal.
2. En el menú principal hay tres opciones:
 - a. Nueva partida: ejecuta el juego y empieza la partida.
 - b. *Ranking*: permite ver los tres jugadores que más rondas han aguantado, el número de rondas que sobrevivieron y los zombis que mataron.
 - c. Salir del juego: cierra la aplicación.
3. Cuando comienza una partida se ejecuta el juego y mientras se juega podemos acceder al menú de pausa donde disponemos de dos opciones:

- a. Reanudar partida: permite seguir la partida en el estado en el que se dejó.
 - b. Volver al menú principal.
4. Si el jugador muere, se llega a una pantalla de fin de juego en la que se muestran las rondas sobrevividas y los zombis exterminados. Además si se está entre los mejores, permite introducir un nombre para guardar nuestra puntuación en el ranking.

5.2.1.6 Apariencia del juego

La apariencia y ambientación del juego tiene que ser realista. Es por ello que el escenario está basado en una base militar o mejor dicho en las diferencias estancias que podríamos encontrarnos en una base militar.

El diseño del nivel es especialmente cargado y muy detallista, en cualquiera de las salas podemos encontrarnos con una gran cantidad de detalle consiguiendo una buena ambientación.

La historia sucede por la noche por lo tanto toda la iluminación del escenario proceden de fuentes artificiales, como pueden ser lámparas, luces del techo, algún proyector, etc.

5.2.1.7 Ámbito

5.2.1.7.1 Lugares en los que se desarrolla el videojuego

Dentro de una base militar, el personaje acaba de salir de tu refugio desde donde dirigía los bombardeos nucleares. Lo envuelve el ambiente oscuro, silencioso, 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 un único nivel con un objetivo: 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 avanza y aumenta su dificultad surge un mayor número de enemigos, que serán más resistentes.

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

5.2.1.7.4 Número de habilidades

Como el protagonista es un alto mando militar, no posee habilidades especiales pero sí dispone de las mecánicas básicas como son disparar, apuntar y moverse, además incorpora mejoras o **power-ups** que le permiten soportar golpes, recargar más rápido y tener una vida extra, de manera que cuando lo maten la primera vez podrá seguir jugando.

5.2.2 Jugabilidad y mecánicas

Dos aspectos esenciales en todo videojuego son tanto la jugabilidad como las mecánicas de juego, de las que depende en gran medida el éxito del mismo. Para el desarrollo del primer aspecto profundizaré en el concepto de jugabilidad así como en los objetivos del juego, la progresión, los controles, las mecánicas, el sistema de *ranking*,... A continuación expondré las mecánicas del juego que incluyen disparar, recargar,...

5.2.2.1 Jugabilidad

5.2.2.1.1 ¿Qué es la jugabilidad?

Aunque no existe una definición única 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 entretenir de forma satisfactoria y creíble ya sea solo o en compañía de otros jugadores*” - José 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*).

Por su parte, la Wikipedia nos ofrece 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.”

Podemos observar, por tanto, que aun careciendo de una definición unánime de jugabilidad, podemos entenderla como **la manera que tiene un jugador de interactuar con el juego**. Otro aspecto entorno al cual sí hay consenso en relación con ella es que se trata de **lo más importante de un juego**, tanto es así que si en algo coincide la mayor parte de los usuarios es que un juego que sea muy bueno gráficamente pero con *jugabilidad* nula no sería un buen juego. 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

En una base militar arrasada por una bomba nuclear que acabó con la vida tal como se conocía, somos uno de los supervivientes, o quizás el único, del ataque nuclear. **Nuestro objetivo es entonces nuestra propia supervivencia** amenazada por hordas de zombis. Aun así tenemos la esperanza de que haya algún sobreviviente más en un refugio similar al nuestro, por ello luchamos por sobrevivir con el anhelo de encontrar algún superviviente.

5.2.2.1.3 Progresión

La progresión de Postwar: Hopeless Humanity es lineal, por tanto se basa en la habilidad que vamos acumulando a medida que avanzamos para que la supervivencia del jugador sea un reto que a la vez sea divertido.

De esta forma, las primeras rondas serán muy sencillas, con muy pocos enemigos que mueren casi instantáneamente, pero conforme se va avanzando en el tiempo la dificultad aumenta. Esta dificultad viene marcada por el incremento del número de enemigos y su resistencia; cuanto mayor sea el número de rondas, mayor será el número de enemigos y más nos costará abatirlos.

La progresión es constante y paulatina de una ronda a la siguiente, con el fin de que el aumento de la dificultad entre ronda y ronda sea casi imperceptible. Paralelamente se va ampliando el tamaño del mapa: al principio el mapa no estará disponible en su totalidad, sino que se jugará en una pequeña parte de este pero, a medida que vayamos adquiriendo experiencia y eliminando enemigos, podremos desbloquear distintas partes del mapa.

5.2.2.1.4 Acciones del personaje

El control de nuestro personaje es el mismo que el de los juegos FPS, o sea que puedes moverte libremente por el escenario, saltar, disparar, recargar, interactuar con algunos objetos (como abrir las puertas, desbloquear nuevas partes del escenario o comprar los *power-ups*). Además podrás acercar la cámara para 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

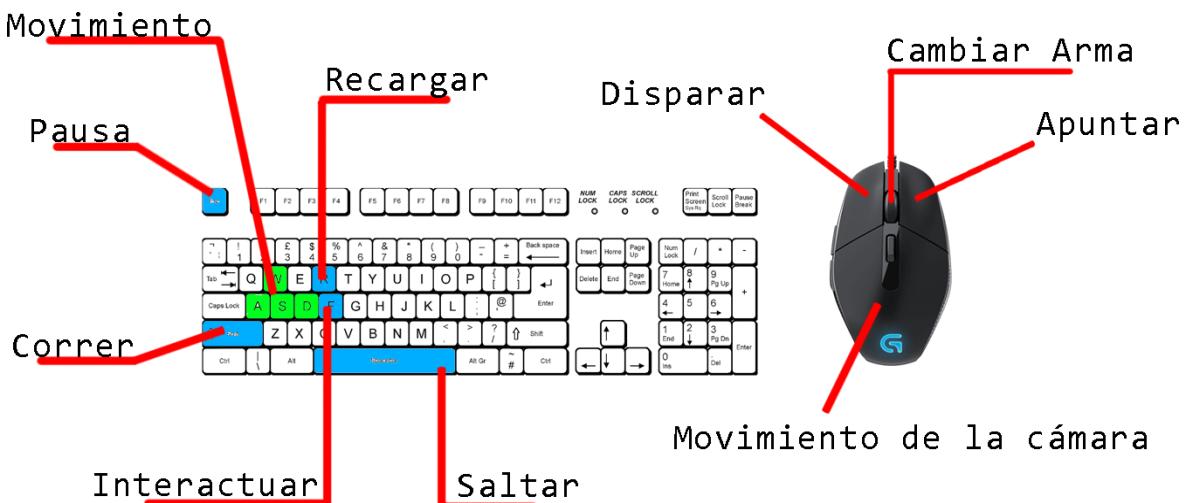


Figura 32: Controles de Postwar: Hopeless Humanity (teclado)

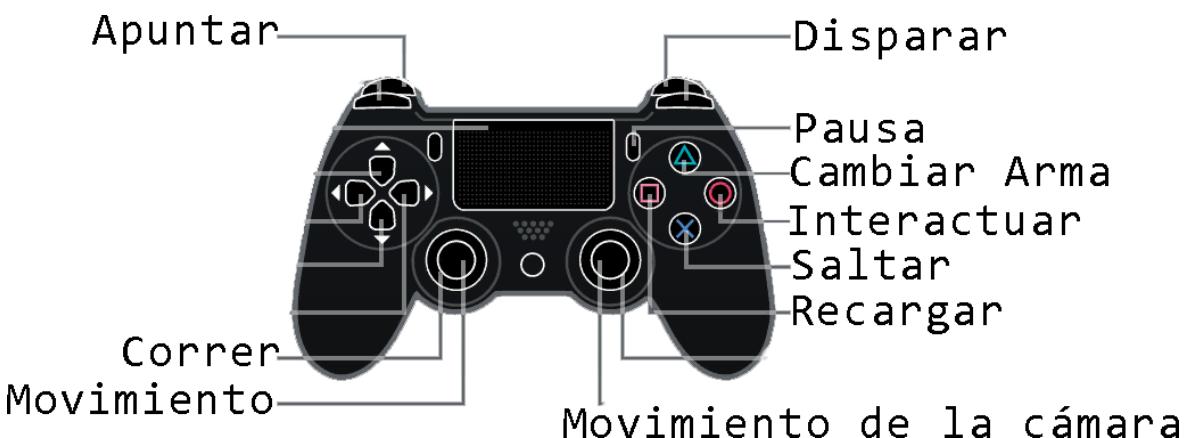


Figura 33: Controles de Postwar: Hopeless Humanity (mando)

5.2.2.2 Mecánicas

Las mecánicas de juego son reglas que generan juegos en busca del disfrute del jugador, por eso producen una cierta adicción y compromiso por parte de los usuarios al plantearles retos.

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

Por lo tanto, **la mecánica y la jugabilidad dependen una de la otra** hasta el punto de que con una mecánica muy simple se pueden conseguir juegos muy divertidos y que enganchen al jugador. Por el contrario, con un gran número de mecánicas complejas que aburran al usuario podemos llegar a acabar con la jugabilidad de un juego.

5.2.2.2.1 Movimiento y cámara en primera persona

Como en cualquier juego FPS, un elemento imprescindible es la cámara en primera persona. Esta ayuda a la inmersión del jugador en el juego al permitirle ver el mundo a través de los ojos de protagonista, por lo que aquel verá únicamente el escenario y el arma con la que se esté jugando.

El movimiento del jugador es lo bastante ligero como para facilitarle el juego sin dar la sensación de lentitud y, a la vez, es lo suficientemente lento como para que sienta que no está corriendo sino andando, todo ello sin que disminuya la presión de que los zombis pueden alcanzarlo. También puede correr durante un cierto tiempo limitado, aunque mientras corre no se permite disparar ni recargar. Por último, el movimiento será libre en el eje de las X y las Y, asimismo es posible dar un salto en el eje de las Z.

La cámara podrá rotar 360° en el eje de las Y pero solo 90° en el eje de las X, es decir, podrá mirar libremente a su alrededor, girarse completamente y ver su espalda; sin embargo, verticalmente no podrá sobrepasar la posición de la cabeza ni de los pies.



Figura 34: Foto del jugador en el escenario con un arma equipada

5.2.2.2 Disparar y recargar

Otra característica fundamental de un juego FPS es la posibilidad de disparar. En este caso el jugador contará con un arma inicial de escasas prestaciones. Tendrá, por tanto, que sobrevivir con esta arma hasta que supere las exigencias necesarias para poder avanzar a otra parte del escenario donde desbloqueará una nueva arma. Cabe añadir que 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 (que significa que ya estará cargada) y además tendrá que tener munición suficiente, es decir, 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 mediante los puntos que ganamos al eliminar enemigos. Si no tenemos puntos suficientes para pagar un arma, no podremos comprarla. Las nuevas armas estarán disponibles en ciertas partes del escenario, permanecerán bloqueadas al principio del juego y serán accesibles a medida que avanzamos en el número de rondas.

Cuando compramos un arma, esta se añade a “nuestro armamento”, que solo podrá disponer de un máximo de dos armas, de forma que si cuando compramos un arma ya tenemos dos el arma que

tengamos equipada en ese momento se sustituirá por el arma comprada; entonces perderemos el arma equipada y obtendremos como nueva arma la comprada. Al comprar un arma dispondremos de toda su munición.

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 se pagan 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:** aumenta 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 mejore la capacidad del cargador la munición máxima del arma seguirá siendo 200 balas pero el tamaño del cargador será de 20 balas.
- **Munición máxima:** aumenta 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 particularidad de que si ya tienes el arma únicamente repondrás munición a un coste mucho menor que si compraras el arma de nuevo.

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

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.

5.2.2.2.6.1 Resistencia

Se podrá comprar la mejora de aumento de resistencia a los golpes de los enemigos para aguantar más, aunque solo se podrá comprar un aumento, es decir, esta mejora no es acumulable, por lo que comprar dos no te hace doblemente resistente.

5.2.2.6.2 Recarga rápida

Con esta mejora se puede recargar más rápido, por lo tanto disminuirá notablemente el tiempo de recarga de las armas. Este *power-up* combinado con unas buenas mejora en el tiempo de recarga del arma (compradas en las máquinas para mejorar armas) pueden ayudar mucho a acabar con grandes hordas de zombis en un tiempo superreducido.

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

5.2.2.6.2 Revivir

Estará disponible un único *power-up* de revivir. Al comprarlo se dispone de una vida extra, o lo que es lo mismo, si nos matan los enemigos reviviremos de nuevo una única vez. Al revivir 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.

5.2.2.7 Abrir puertas

El jugador tendrá la posibilidad de abrir puertas para ir ampliando el escenario y así acceder a diferentes armas y mejoras. Las puertas tendrán un coste de puntos por tanto, antes de abrir una puerta, es necesario eliminar una cantidad de enemigos suficiente para obtener los puntos que costará abrir la puerta.

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 pues esto rompería el estilo de supervivencia que se quiere crear; por lo tanto, si se abandona a mitad de una partida, se pierde todo el progreso y, cuando se vuelva a iniciar el juego, se tendrá que volver a empezar desde la primera ronda.

Lo que se pretende con esto es mantener ese clima de tensión en el que se encuentra el jugador después de estar sobreviviendo el mayor número de rondas posibles. Si se guardaran partidas y se retomaran otro día, por ejemplo, se perdería toda la inmersión del jugador así como todo lo que le hubiera costado llegar hasta el punto en el que se encontrara en ese momento. Por el contrario, al no permitir guardar la partida el jugador sabe que tiene que sobrevivir el máximo tiempo posible sin vuelta atrás y sin descansos (sí hay descansos en el menú de pausa pero no suponen el abandono de la partida sino un receso para 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 tres mejores jugadores, sus nombres, los zombis que mataron y, lo más importante, hasta qué ronda sobrevivieron.

Al acabar una partida se comprueba si la puntuación es suficiente para entrar en el *ranking*. De ser así, saldrá una pantalla en la que se podrá introducir el nombre del jugador.

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 funcionar. El mundo se divide en dos bloques: los que están bajo la influencia y el dominio de Corea del Norte y los que se oponen a ella.

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. La diplomacia internacional ha fracasado en su intento de alcanzar un acuerdo y el temor empieza a apoderarse de una población pesimista acerca de su futuro. Aumenta el clima de crispación, temor y desesperación entre los países que ven como única salida el recurso a su armamento nuclear.

Ambas partes se preparan para la guerra, las pruebas de misiles de largo alcance son continuas cada día, pero “un fallo” en el lanzamiento de un misil nuclear hace que este acabe estrellándose en un territorio enemigo del régimen coreano y destruya allí todo atisbo de vida. A este suceso se responde con otro ataque que desencadena lo que podemos llamar la Tercera Guerra Mundial.

Esta nueva guerra es muy diferente a las anteriores puesto que ya no son los soldados los que tienen el protagonismo sino que son los misiles nucleares los que se encargan de arrasar los territorios enemigos. Estos misiles son controlados desde unos refugios nucleares creados exclusivamente para este fin y situados en distintas partes del mundo desde donde se lanzan ataques indiscriminados que envuelven al planeta entero en una gran guerra nuclear. Cuando cesan los ataques de ambos bandos, se descubren las graves consecuencias: la pervivencia 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 advertimos la gravedad de la situación: sin presencia de vida humana, nuestro objetivo ahora es la

lucha por nuestra supervivencia enfrentándonos contra unos muertos vivientes, que es en lo que se han convertido los que antes eran humanos, a causa de la exposición a la elevada radiación de la guerra. Con todo, aún 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

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. Sin embargo, cuando salimos del refugio, nos damos cuenta de 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 las bombas nucleares se convirtió en zombi, tanto amigos, como compañeros del ejército o enemigos.

5.2.4 Nivel de juego

Postwar: Hopeless Humanity solo tiene un nivel en el que tendrás que sobrevivir el mayor tiempo posible. Este nivel no es accesible en su totalidad desde el principio tal como detallaremos en el apartado “Escenario”.

5.2.4.1 Inteligencia Artificial

La inteligencia artificial de Postwar: Hopeless Humanity es más bien sencilla por la razón de 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 incremento de enemigos y su resistencia, que obstaculizan cada vez más que sigamos con vida. Por lo tanto, la IA se resume en perseguir al jugador y atacarle cuerpo a cuerpo cuando esté cerca de nosotros.

Por el contrario, he puesto más énfasis en el diseño del número de enemigos que pueda haber simultáneamente para complicar el juego sin caer en el exceso, con especial atención a los puntos

de *respawn* (puntos donde aparecen los enemigos), que estarán repartidos por diferentes partes del escenario. Ahora bien, si un espacio del escenario está cerrado no podrán aparecer enemigos en puntos de *respawn* que estén en él, cuestión que se desarrollará más profundamente en el apartado de “**Puntos de *respawn***”.

5.2.4.2 Escenario

Como se ha dicho, el nivel es único: una base militar bastante moderna aunque dañada por la guerra. Se trata entonces de un ambiente conocido pero totalmente cambiado: ahora está desierto, la luz es débil y titila debido a las bajadas y subida de tensión. Las habitaciones son muy variadas: desde estancias muy amplias que te harán sentir inseguro por la dimensión de las mismas, hasta estrechos pasillos que te generarán una gran tensión al sentirte rodeado y sin escapatoria.

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

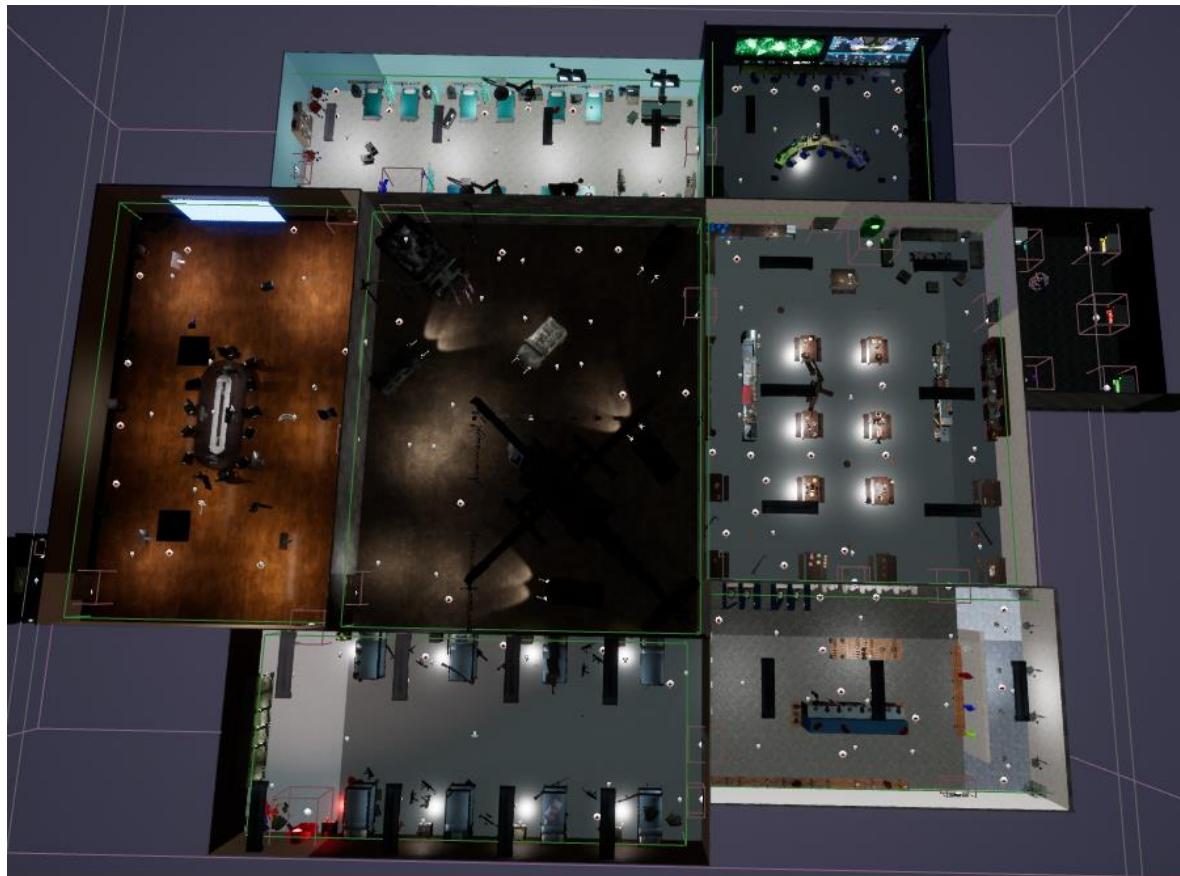


Figura 35: Vista aérea del escenario

5.2.4.3 Música y sonidos

La música y los sonidos son un componente muy importante en cualquier videojuego pues un buen diseño sonoro es capaz de conseguir una inmersión total del jugador y de situarlo en el contexto de una forma realista.

A pesar de que no es la parte más elaborada en el juego, siempre hay que ser cuidadosos con los sonidos gracias a los cuales se puede influir en las sensaciones del jugador, crearle momentos de tensión o, por el contrario, momentos de relajación, serenidad y ausencia de peligro por medio de una música suave, por ejemplo.

Por esa razón en Postwar: Hopeless Humanity habrá varios tipos de sonido:

- **Sonidos y música ambiental:** sonidos o música que se escucharán de fondo durante todo el juego y cuyo objetivo es crear una atmósfera que posicione al jugador dentro del propio juego.
- **Efecto de sonido (SFX):** sonidos que se reproducen en un sitio concreto y que son el resultado de alguna acción, por ejemplo un disparo, la apertura de una puerta, la mejora de un arma, etc.
- **UI Sounds** (sonidos en la *interfaz* de usuario): 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 los que puede *respawnear* (aparecer) un enemigo.

La ubicación de los puntos de *respawn* como tal no es una decisión del todo importante, lo fundamental es el uso que se hace de ellos y las limitaciones que se les imponen (tiempo entre *respawn* y *respawn*, aleatoriedad, etc.). Los puntos de *respawn* estarán repartidos por todo el escenario, lo cual supone un problema de inicio porque el escenario al comenzar el juego no está disponible en su totalidad sino solo 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 indicará que en ese punto no podrá aparecer ningún enemigo. A medida que vamos desbloqueando nuevas partes del escenario, también se van activando los puntos de *respawn* de esos lugares del escenario. Ahora bien, 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 es directamente proporcional al número de ronda en la que nos encontramos: cuanto mayor sea el número de ronda, mayor será la dificultad y, por lo tanto, menor el tiempo de aparición entre enemigo y enemigo.

Asimismo, la cantidad de enemigos que aparecen también depende de la ronda. Al empezar la ronda habrá de golpe una oleada de enemigos cuya cantidad corresponde al resultado de una fórmula en la que el factor importante es el número de rondas.

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

La idea entonces es enfrentarse a una primera horda de zombis en cada ronda y, mientras estás intentando sobrevivir a ella, que vayan apareciendo zombis continuamente con un intervalo fijo de tiempo. Tanto el tiempo de aparición 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 ha aparecido el número máximo de zombis de esa ronda, estos dejarán de reaparecer y de esta manera cuando los mates ya no aparecerá ninguno más lo que supondrá el paso a la siguiente ronda.

5.2.5 Interfaz

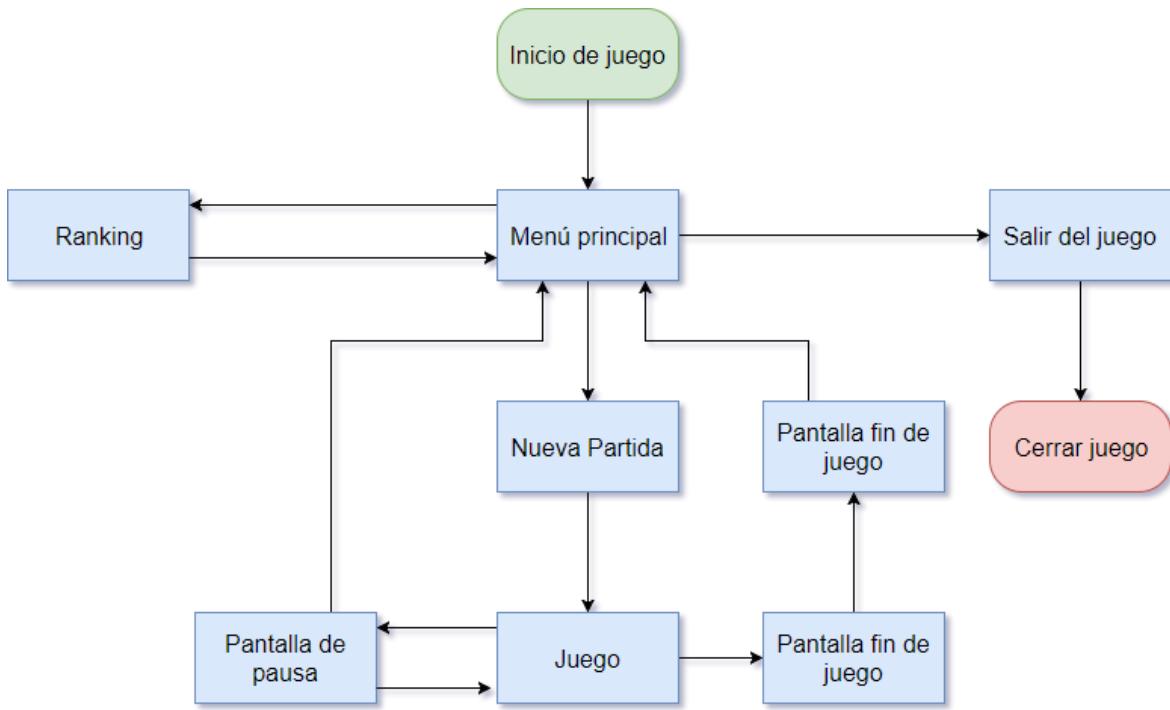
Una *interfaz* de usuario (UI) es el medio por el que el este puede comunicarse con un equipo (o el juego en este caso) y comprender los puntos de contacto entre ambos. Es decir, es la manera en la que podemos comunicarnos con el juego y la forma en la que el juego se comunica con nosotros, por ejemplo,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

A continuación expondré los componentes de la interfaz del juego Postwar: Hopeless Humanity.

5.2.5.1 Menús

Los menús permiten navegar a través del juego y modificar algunas opciones como la resolución, el acceso al *ranking* o los créditos, empezar una nueva partida o salir del juego.

Para poder explicar los menús de una forma más clara, reproducimos a continuación la foto del apartado 5.2.1.5. con el **Resumen del flujo de juego**:



5.2.5.1.1 Menú principal



Figura 36: Menú del juego en Photoshop

Este es el menú principal, o sea, lo primero que se ve cuando se ejecuta el juego, y que permite:

- 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 interrumpe el juego. En él podremos retomar la partida tal cual la habíamos dejado o regresar al menú principal.



Figura 37: Menú de pausa en juego

5.2.5.1.3 Menú fin de partida

Hay dos tipos de menú de final de partida: uno para cuando mueres con puntuación suficiente para entrar en el *ranking*, otro para 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 38: Pantalla muerte sin ranking

5.2.5.1.3.2 Con Ranking

En esta pantalla aparecerá un cuadro en el que escribir el nombre con el que luego se figura en el *ranking* así como un botón de aceptar que validará la información y llevará de vuelta al menú principal.

Además también muestra información de la partida, como el número de muertes y la ronda a la que hemos llegado, datos que se registrarán luego en el *ranking* junto al nombre que hemos introducido.

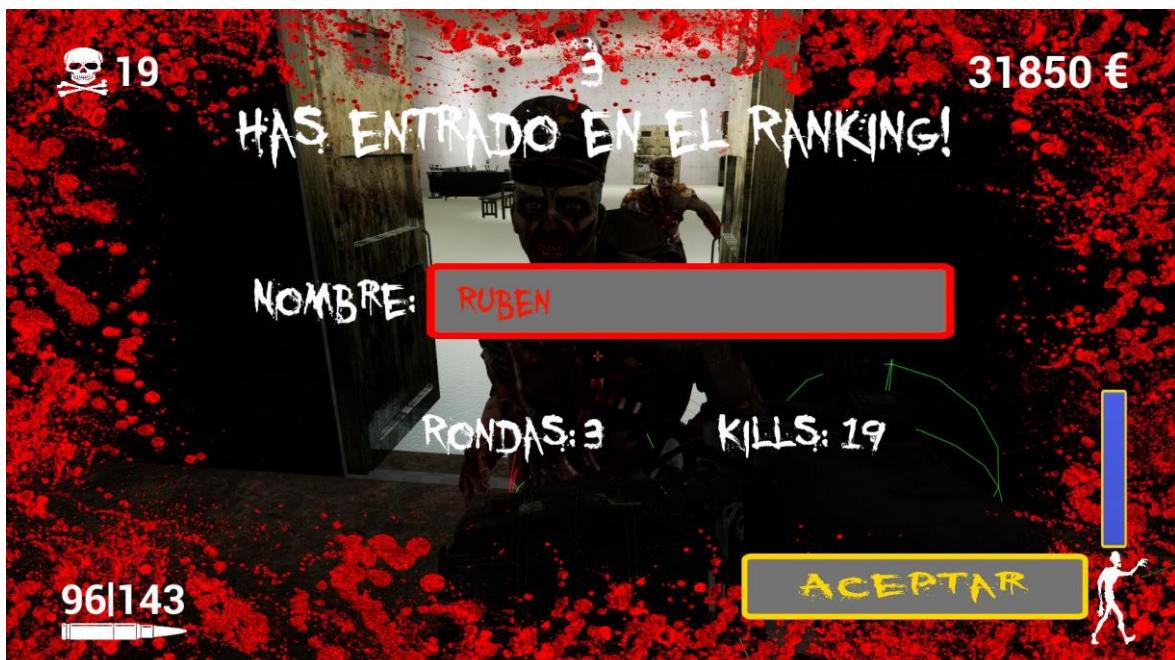


Figura 39: Pantalla muerte con ranking

5.2.5.1.4 Pantalla de *ranking*

En esta pantalla se muestran los tres jugadores que hayan obtenido mejor puntuación. Esta información se guarda entre partidas y se mantiene intacta aunque se cierre el juego. Incluirá el nombre, el número de rondas alcanzado y el número de zombis eliminados, además de un botón de “exit” con el que regresar al menú principal.

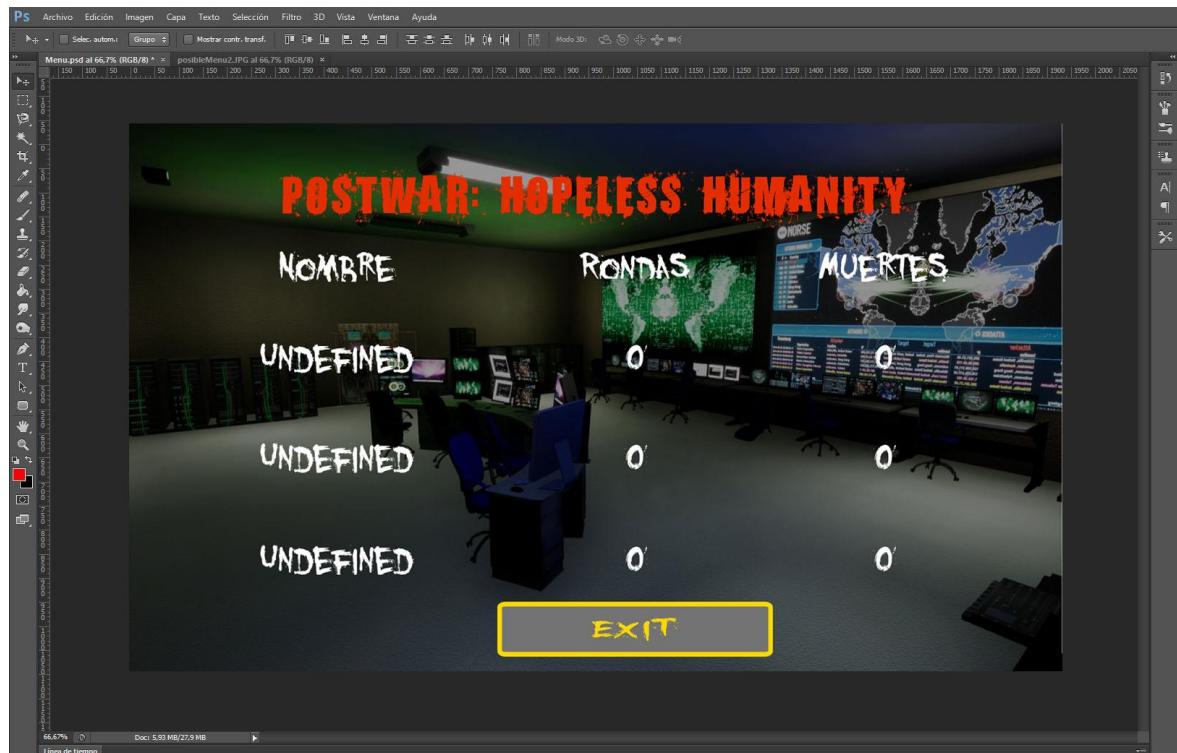


Figura 40: Menú ranking en Photoshop

5.2.5.1.5 Menú de opciones

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

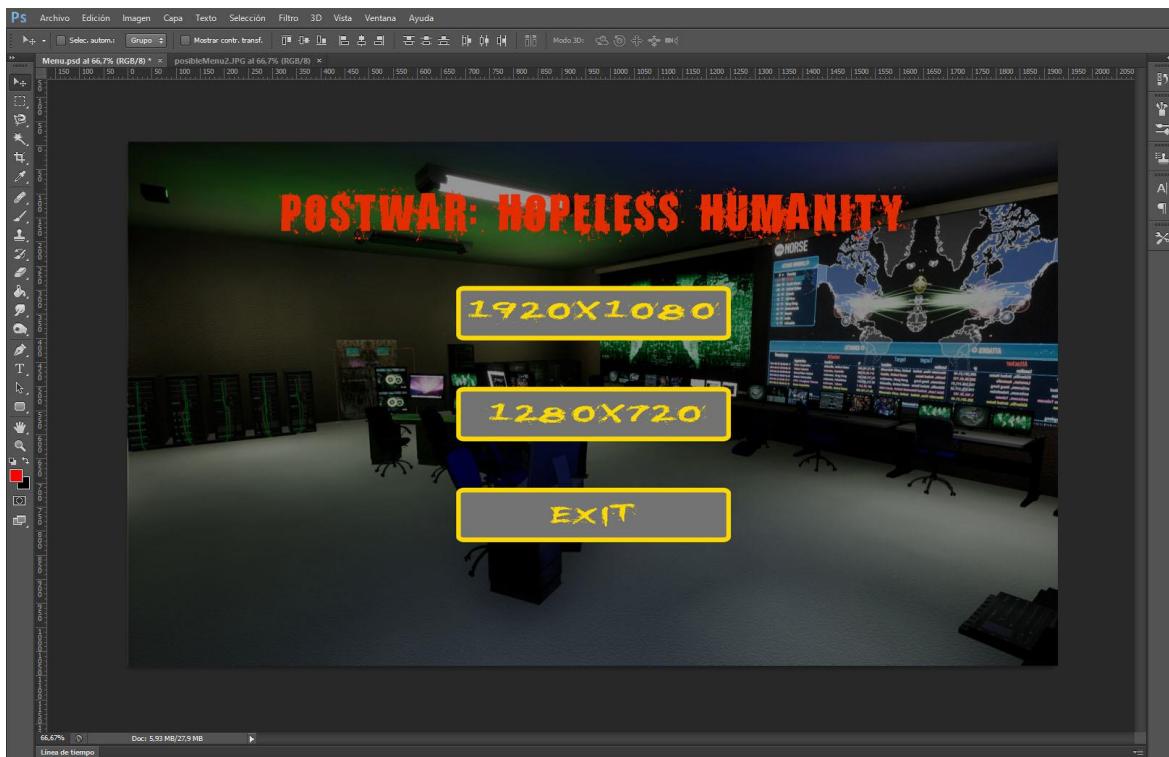


Figura 41: Menú opciones en Photoshop

5.2.5.2 HUD

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

En el HUD de *Postwar* se diferencian dos partes:

- Información que se muestra en todo momento (**Figura 42**):
 - Zombis exterminados (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).

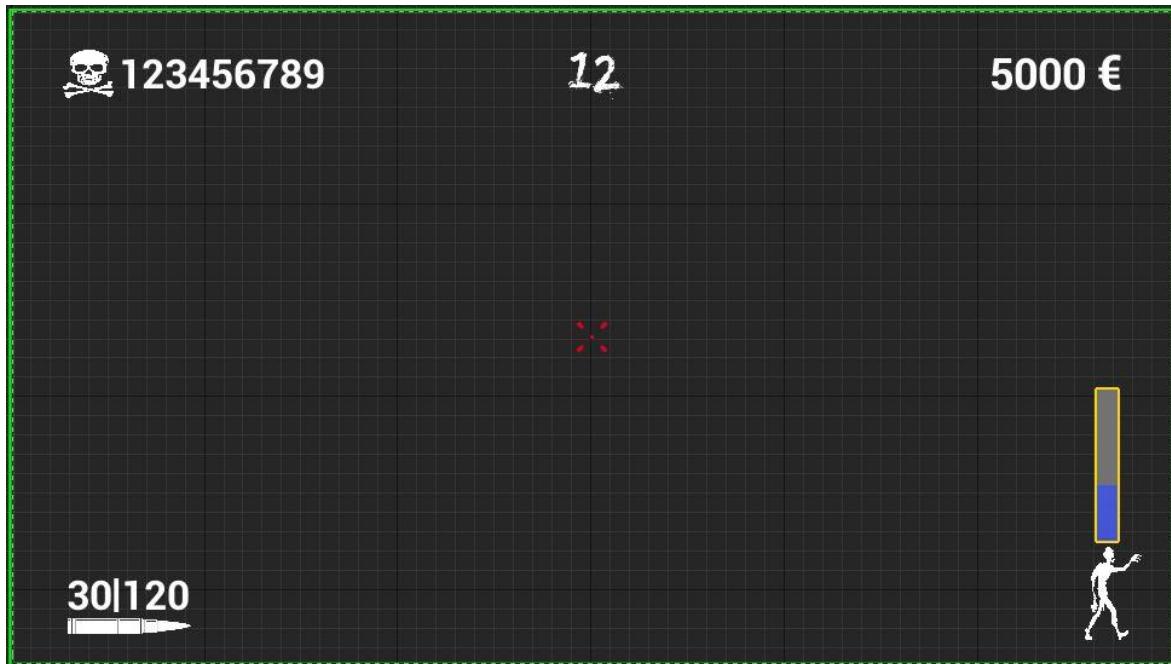


Figura 42: HUD información constante

- Información que aparece por algún evento concreto (**Figura 43**):
 - Sangre alrededor de la pantalla: va apareciendo dependiendo de la vida restante. Su intensidad es inversamente proporcional a lo que nos quede de vida, 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: se mostrará cuando estemos reviviendo.
 - Cambio en la puntuación (arriba derecha): figurará cuando haya una variación en la puntuación que podrá ser positiva, por ejemplo, al matar algún enemigo o por impacto en la cabeza, y será de color verde; o bien, podrá ser negativa, de color rojo, cuando por ejemplo compremos algún arma, alguna mejora, munición, abramos una puerta, etc.
 - Información sobre objeto *interactuable* (en este caso “M4a1 300€”): este mensaje se verá cuando estemos cerca de algún objeto con el que podamos interactuar y mostrará la información del objeto y su precio, por ejemplo si nos acercamos a una puerta pondría “Puerta 900€”.



Figura 43: HUD información dinámica

5.2.5.3 Cámara

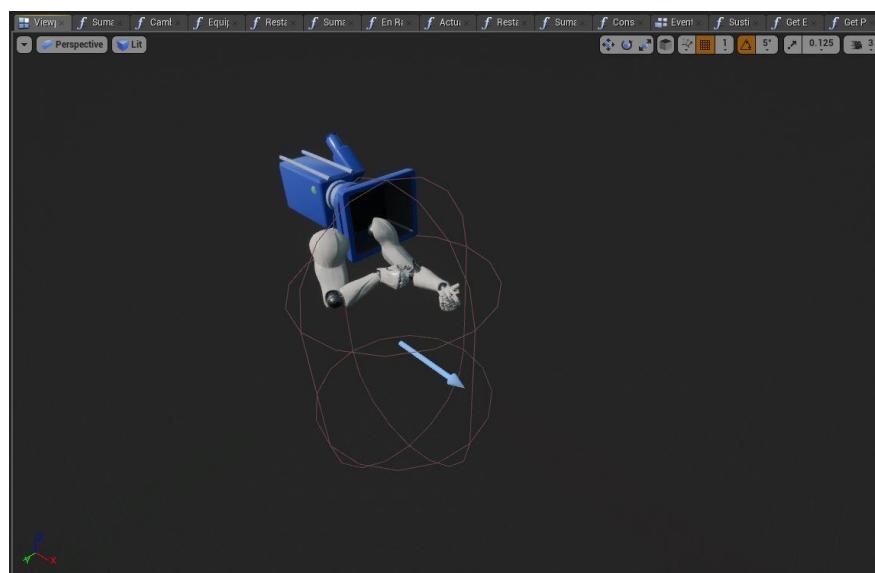


Figura 44: Camara FPS

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

Los elementos destacados son la propia cámara a partir de la cual podremos ver el juego, una cápsula de colisión que reproduce el tamaño del cuerpo del personaje para que podamos colisionar contra obstáculos u enemigos de la misma forma que lo haría un cuerpo y los brazos que, aunque al principio

no sostienen nada, en el juego se encargarán de sujetar las armas cuando las compremos, además, para asegurarnos de que estas están en el lugar correcto, se han añadido unos *sockets* al esqueleto de los brazos con el fin de que, cuando compremos una arma, esta se coloque en la posición de ese *socket* y siga su movimiento, de esta manera si el brazo se mueve el arma también lo hace.

5.3 Desarrollo e implementación

5.3.1 Unreal Engine 4 y su sistema de *scripting* visual Blueprints

Tal como explicamos en el apartado 5.1.1., el motor de juego elegido para el desarrollo de Postwar: Hopeless Humanity es *Unreal Engine 4* cuya arquitectura analizamos seguidamente junto con 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, qué es y para qué se usa, se verán conceptos básicos como que 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 a los actores. Sin embargo, mientras los objetos son instancias de clases que heredan de *UObject*, 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 y pueden experimentar transformaciones de posición, escala y rotación, mientras que un Objeto no puede colocarse en el mundo. Otra diferencia es que los actores están preparados para la réplica en un juego *online* (*networking support*); por tanto, la diferencia fundamental entre los *UObjects* y los *Actors* es que cualquier actor puede posicionarse en el mundo y los *UObjects*, no. 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. En consecuencia, 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, y dotarlos de un sistema que los 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*), cámaras para jugadores o clases como *GameMode*, *GameState* y *PlayerState* que establecen las reglas del juego, informan del estado en el que este se encuentra así como de la progresión de los jugadores sobre este.

5.3.1.1.3 Representación y control de jugadores

5.3.1.1.3.1 Representación de jugadores en el mundo

Existen dos formas de introducir actores que puedan ser controlados por cualquier jugador o por la IA, 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; puede ser poseído por un *Controller* y realizar acciones propias de cada uno. Ahora bien, 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 usamos el ***FirstPersonCharacter*** al ser un FPS (juego de disparos en primera persona), pero también se podría emplear un ***ThirdPersonCharacter*** para juegos en tercera persona.

Los *character* vienen con una cápsula de colisión integrada que se encarga de simular la colisión de un cuerpo humanoide, además disponen de un componente de movimiento por defecto que nos permite realizar acciones básicas como movernos y saltar desde el principio; por último, un *character* también está dotado de la capacidad para replicar su movimiento *online* y para 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 a un *pawn* y controlar sus acciones. Hay dos tipos de *controller*: el ***PlayerController*** y ***AIController***.

5.3.1.1.3.2.2 PlayerController

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

5.3.1.1.3.2.3 AIController

El *AIController* es el otro tipo de controlador, con la diferencia de que 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 ofrece información al usuario sin necesidad de que este cambie su punto de vista. Por lo tanto, el HUD, como ya dijimos en el apartado HUD del GDD, es una *interfaz* 2D que se muestra sobre la pantalla y que ofrece al jugador información importante sobre el estado del juego. Los HUD típicos aportan información sobre vida, puntuación, munición, la mirilla con la que se apunta, etc.

5.3.1.1.4.2 Cámara

La cámara sería lo que en la vida real se corresponde con nuestros ojos; en consecuencia, cada *PlayerController* dispone de una cámara que es gestionada 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 divide en tres apartados: el *GameMode*, *GameState* y *PlayerState*. Estos trabajan conjuntamente de manera que un *GameMode* incluye un *GameState* y a su vez este contiene un *array* de *PlayerState*.

5.3.1.1.5.1 GameMode

GameMode existe solo en el servidor y se encarga de controlar cómo 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. Por otra parte, se

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

5.3.1.1.15.2 GameState

GameState existe tanto en el servidor como en todos los clientes. Su cometido es controlar los aspectos comunes a todos los jugadores. Por ejemplo, en un mundo se encarga de asegurar que todos los objetos estén en las mismas posiciones para todos los clientes o garantiza que si en un juego alguien abre una puerta el *GameState* se mantenga abierta para el resto de clientes. En consecuencia, *GameState* contiene información compartida por todos, como la lista de jugadores, la puntuación total del equipo, las misiones que se cumplen en un mundo abierto, etc.

5.3.1.1.15.3 PlayerState

PlayerState es específica 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, o lo que es lo mismo, un *PlayerState* registra la información importante para un cliente sin necesidad de que la conozca el servidor ni de gastar ciclos en calcular esa información. Cada jugador dispone de un *PlayerState* y estos pueden replicarse en la red para que todos los clientes estén sincronizados.

5.3.1.1.16 Cómo se relacionan estas clases

El siguiente gráfico extraído de la documentación oficial de *Unreal Engine 4* muestra cómo se relacionan las clases explicadas anteriormente.

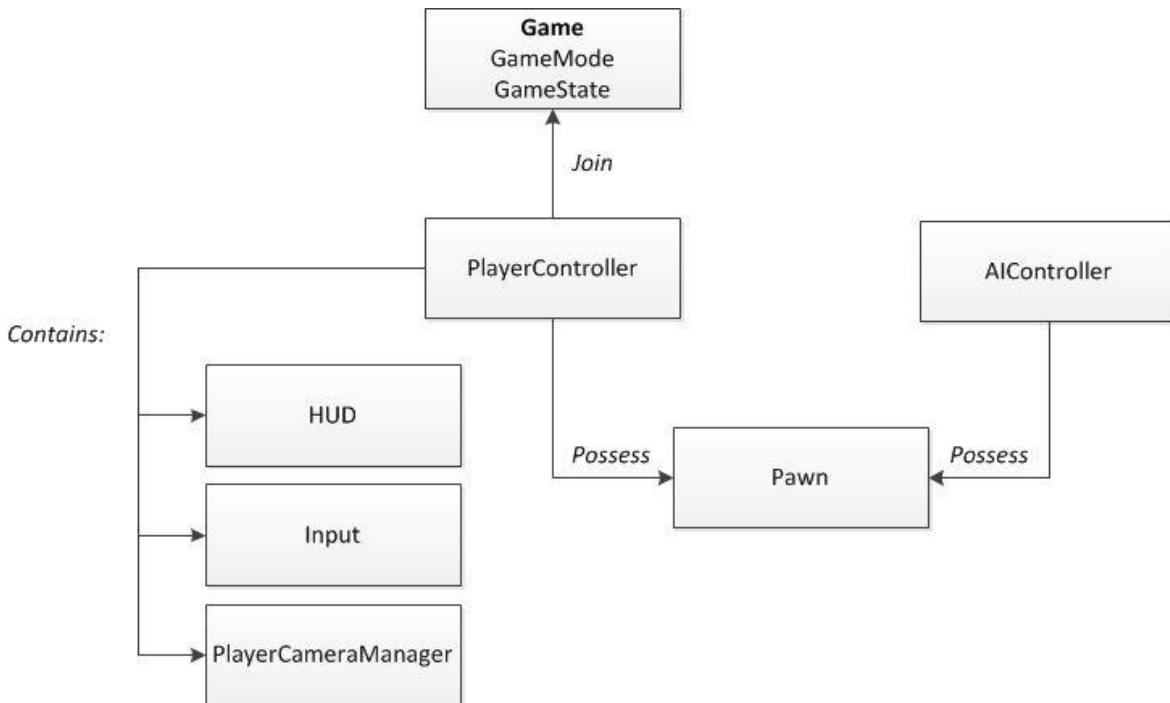


Figura 45: Relacion entre clases

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

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 uso de 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 *Blueprint* se les llama “*Blueprint*”. Así un *Blueprint* es una clase dentro del juego, como puede ser por ejemplo un arma, y posee toda la lógica de esta, como una función para disparar, una función de recargar, etc.

Este sistema es muy flexible y poderoso dado que permite a los diseñadores manejar gran cantidad de herramientas y conceptos reservados antes para programadores. A través del uso de *Blueprint* los diseñadores crean cualquier elemento del *gameplay*, como por ejemplo las reglas del juego, la

modificación y creación de mallas o de materiales, la confección de cámaras o la introducción de modificaciones en ellas, el cambio de los controles, la incorporación de nuevos *inputs*, etc.

5.3.1.1.2.1 ¿Qué es un *Blueprint*?

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

Una peculiaridad de los *Blueprints* radica en que también admiten herencia al igual que se haría en código. De esta manera, cuando un ***Blueprint* padre** es heredado por los ***Blueprints* hijos**, estos heredan 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 ejecutará la lógica del hijo; por el contrario, si no se sobrescribe, se ejecutará la lógica que había en la función del padre.

5.3.1.1.2.2 ¿Qué contiene un *Blueprint*?

Los *Blueprints* ya incluyen algunos componentes por defecto, mientras que otros pueden ser añadidos en función de nuestras necesidades. Incorporar componentes a nuestro *Blueprint* facilita su reutilización para agilizar así el *level design*. Por ejemplo, si tenemos varios objetos con los que 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 ejecutará 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*)

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 y podrá modificar la funcionalidad de este. Por ejemplo, un *Spot Light Component* permitirá a un actor emitir luz como si fuera un *Spot Light* (luz tipo linterna).

Aclarado el concepto *componente*, cabe definir la **ventana de componentes** (*Components Window*).

La ventana de componentes permite añadir nuevos elementos a nuestro *Blueprint* desde el propio *Blueprint Editor* y posibilita la incorporación de objetos de colisión, tales como cápsulas, o agregar cámaras, partículas, efectos de sonido, etc. Y lo más importante de todo es que estos componentes pueden ser referenciados desde el propio *Blueprint* lo que permite realizar cualquier tipo de acción con ellos como inicializarlos, destruirlos, cambiar sus propiedades, responder a eventos... Además los componentes pueden ser incorporados a variables para que accedan a ellos no solo el *Blueprint* al que pertenecen sino también 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. Se trata pues de una función que se llama cuando el *Blueprint* es creado y así permite definir una inicialización. No obstante, conviene aclarar que solo puede haber un *Construction Script* por *Blueprint*. Constituye una herramienta muy potente ya que contiene un nodo de grafo que se ejecuta al crear una instancia del *Blueprint* y permite realizar operaciones de inicialización, lo que 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 el manejo de todas las armas desde la clase padre, pero es el *Construction 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 46: Contruction Script del arma AK-47

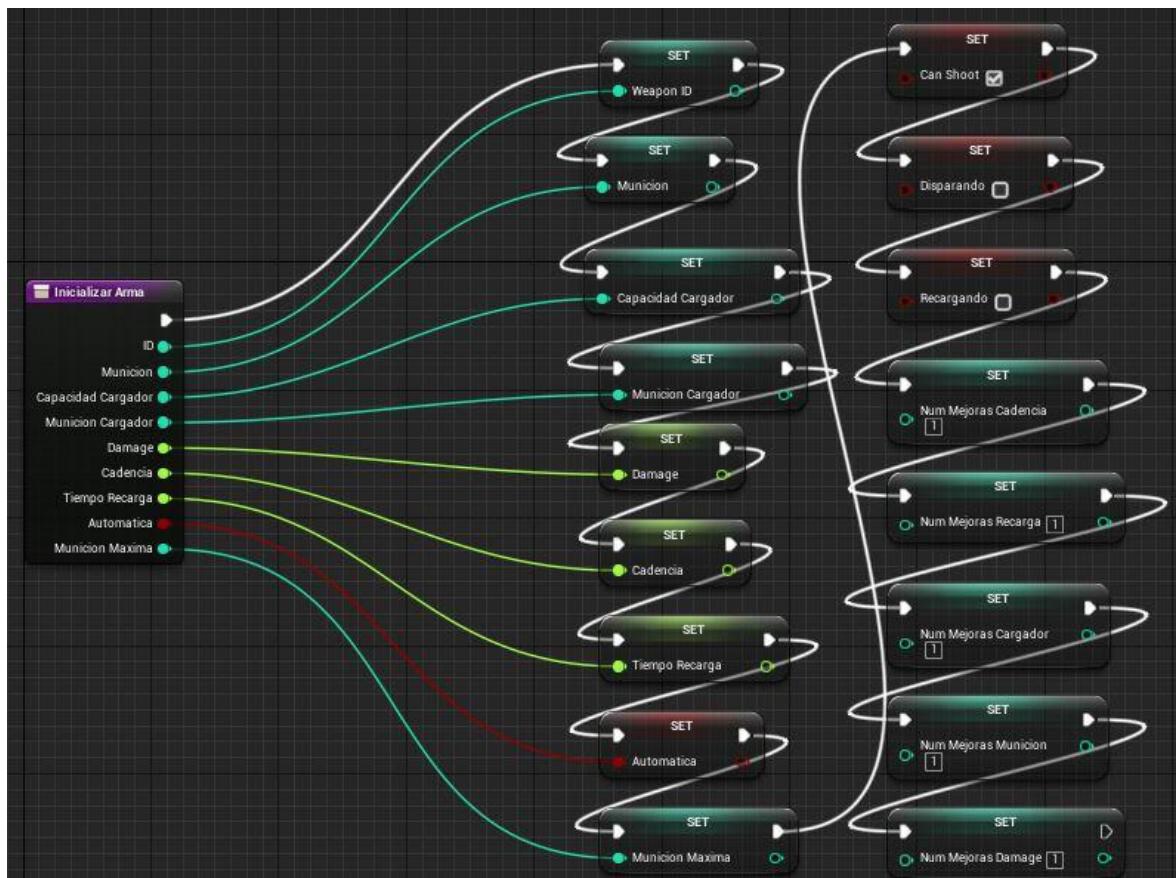


Figura 47: Función inicializar al que llaman los constructores de las armas

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*), a eventos personalizados creados por el propio usuario o incluso a 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, cuando esta función la hagamos en el *Event Graph* se ejecutará 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* añaden 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 usada 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 se pueda acceder desde otros *Blueprints*.

5.3.1.1.2.2.6 Macros

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

En el supuesto de que tengamos dos *if else* anidados, en vez de realizar esto en el grafo normal, podemos optar por una macro que contenga los dos *if elses*, de manera que la entrada sería la de ejecución mientras que la salida sería la ejecución resultante de los dos *if elses*, así podría cumplirse el segundo *if* y se devolvería esta ejecución por la macro para continuar con la ejecución en el grafo que se llamó.

Con el objeto de aclarar lo anterior, a continuación se incluye una pequeña demostración de una macro con el ejemplo citado arriba. La primera imagen corresponde a la macro con los dos *if else*, y la segunda imagen, a la llamada a la macro desde un evento personalizado del grafo. La macro permite aislar este código para que no esté en el propio grafo, devuelve la ejecución por el pin que necesitamos y además permite que el código de la macro sea reutilizable.

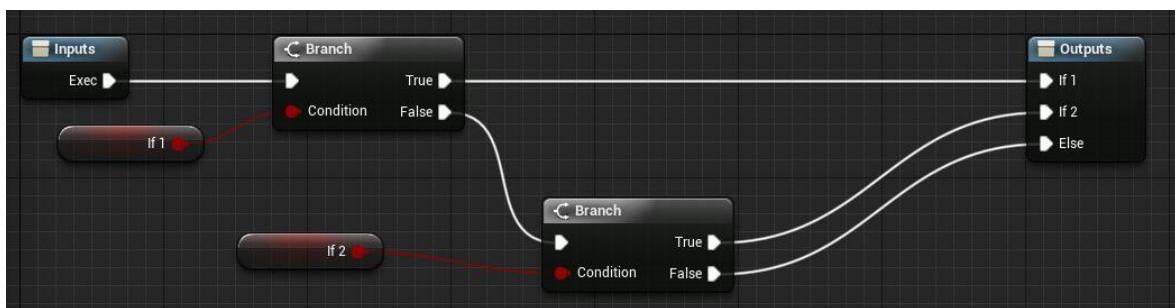


Figura 48: Ejemplo de Macro

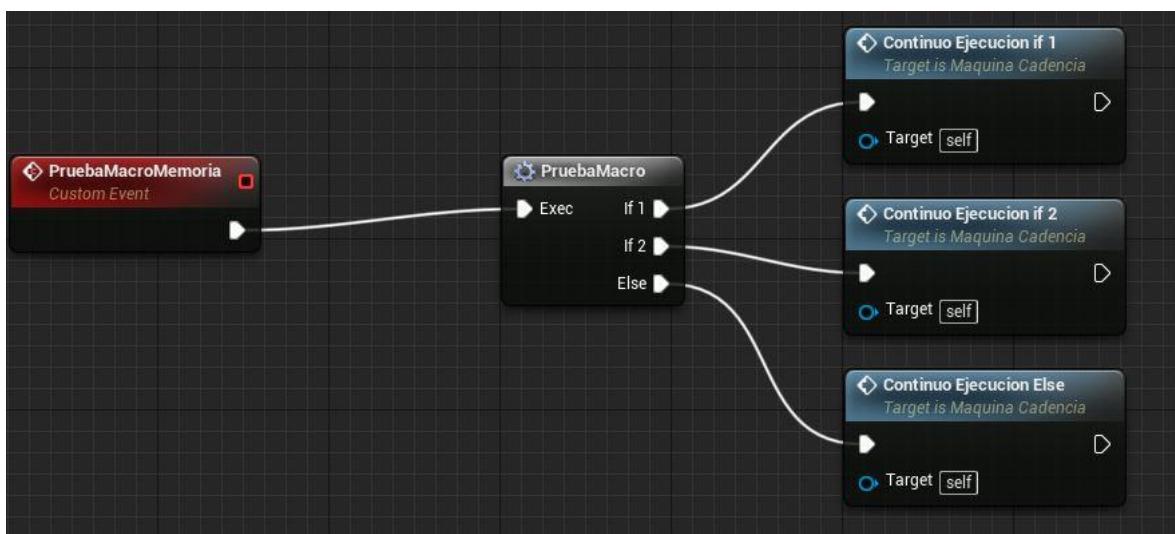


Figura 49: Ejemplo llamada a macro desde evento personalizado

5.3.1.1.2.3 Tipos de Blueprints

Existen varios tipos de *Blueprint* con distintas funcionalidades. Los dos más comunes son *Level Blueprint* y *Blueprint Class*, aunque vamos a explicarlos todos brevemente.

5.3.1.1.2.3.1 Blueprint Class

Blueprint Class es el más común y usado, por lo tanto nos referiremos a él únicamente como *Blueprint*. Se trata de 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 instancia que se comporta de igual manera que lo haría otro *Actor*.

5.3.1.1.2.3.2 Level Blueprint

Level Blueprint es un tipo de *Blueprint* especial que actúa como grafo de nodos global de un determinado nivel. Cada nivel tiene su propio *Level Blueprint* creado por defecto 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 gráficos), variables y componentes heredados de su padre. Estos *Blueprints* permiten cambiar y modificar esas propiedades heredadas pero no añadir nuevos elementos. Son, por lo tanto, un reemplazo de las funcionalidades de las clases padres y 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 tan solo añadiendo código o variables nuevas, así pasan de ser un *Data-Only Blueprint* a ser un *Blueprint Class* que se editará 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 y así incrementará la 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. Otra ventaja 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 se tratara de 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 virtud de este tipo de *Blueprint* es que ofrece un método común de interactuar con objetos totalmente distintos que comparten alguna funcionalidad específica. Si, pongamos por caso, tenemos un árbol y un coche y los dos pueden ser disparados por un arma de fuego y pueden ser dañados, posibilita la creación de un *Blueprint Interface* que contenga una función llamada, por ejemplo, “RecibirDaño”, así tanto el coche como el árbol implementan esta función “RecibirDaño” por lo que, al ser disparados, se envía un mensaje a través de la interfaz que permite tratar estos dos objetos totalmente distintos 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 (para recordar la función de una macro véase el apartado 5.3.1.1.2.2.6 Macros); por consiguiente, un *Blueprint Macro Library* sirve para almacenar distintos tipos de macros con el fin de que estas puedan ser reutilizadas y llamadas desde cualquier *Blueprint*. Es decir, permite que las macros sean comunes y 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 contiene herramientas avanzadas de *DirectX11* para iluminación global, transparencia con iluminación, efectos de post procesado como el *Bloom* y simulación de partículas mediante la GPU utilizando campos vectoriales.

Un aspecto destacable a la hora de *renderizar* en *Unreal Engine 4* es que se utiliza el sombreado diferido, más conocido como *Deferred Shading*, mientras que en *Unreal Engine 3* se usaba el *Forward Lighting*. Esta novedad permite emplear una gran cantidad de luces en la escena con un

coste reducido, ya que el *Deferred Shading* reserva 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 cuatro tipos de luces: *Spot*, *Point*, *Directional* y *Sky*.

Spot Light emite luz desde un punto, pero la luz se proyecta de forma cónica, igual que si se tratara de la luz emitida por una linterna.

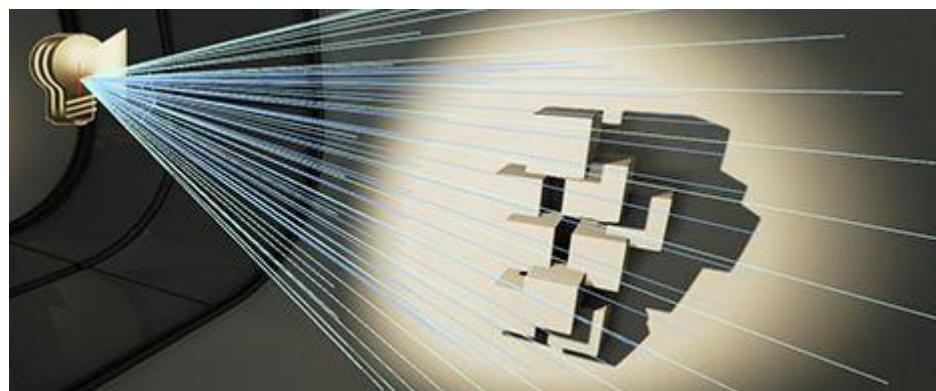


Figura 50: Ejemplo Spot Light

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



Figura 51: Ejemplo Point Light

Directional Light se usa normalmente para simular la luz exterior, o sea, la luz natural emitida por el sol o por cualquier fuente de luz que esté extremadamente lejos; sería la equivalente a luz emitida por el sol.

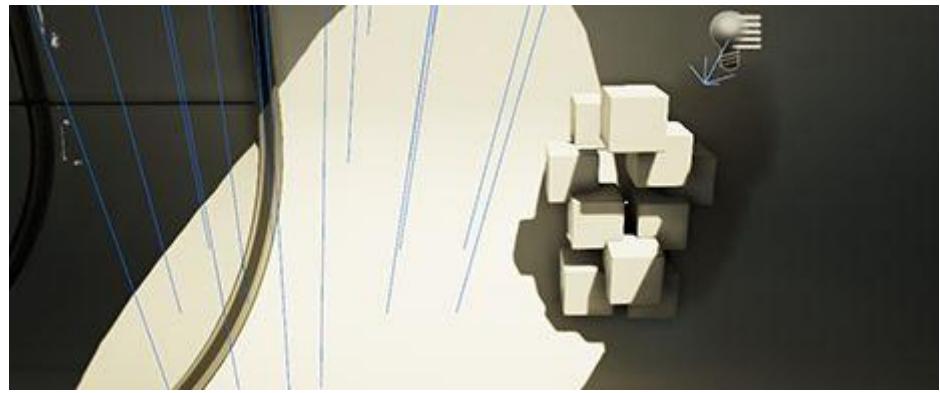


Figura 52: Ejemplo Directional Light

Sky Light aplica una luz general y uniforme a todos los elementos de una escena igual que una luz ambiente o luz indirecta que permite que se distingan los objetos aun en zonas de penumbra.

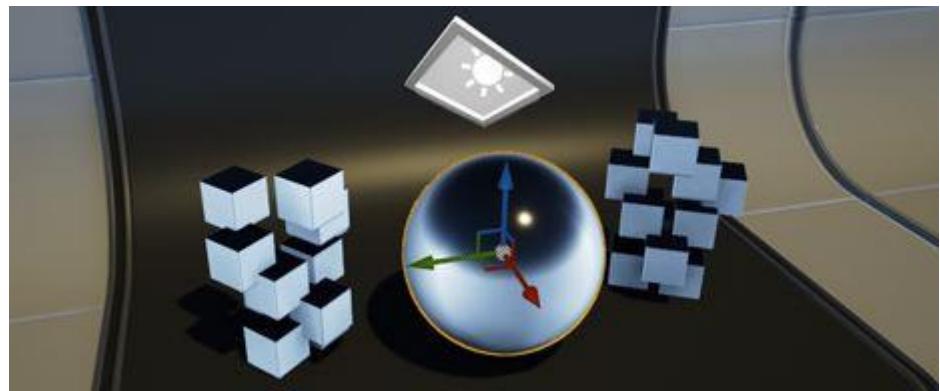


Figura 53: Ejemplo Sky Light

Las cuatro imágenes anteriores están tomadas 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 imperceptibles lo que hace que recaiga toda la importancia de la luz en las *SpotLight* y, sobre todo, en las *PointLight*. A continuación mostramos tres imágenes extraídas del proyecto en las que se ven tanto *SpotLight* como *PointLight*.



Figura 54: Ejemplo de una sala del proyecto con iluminación Spot Light y Point Light.

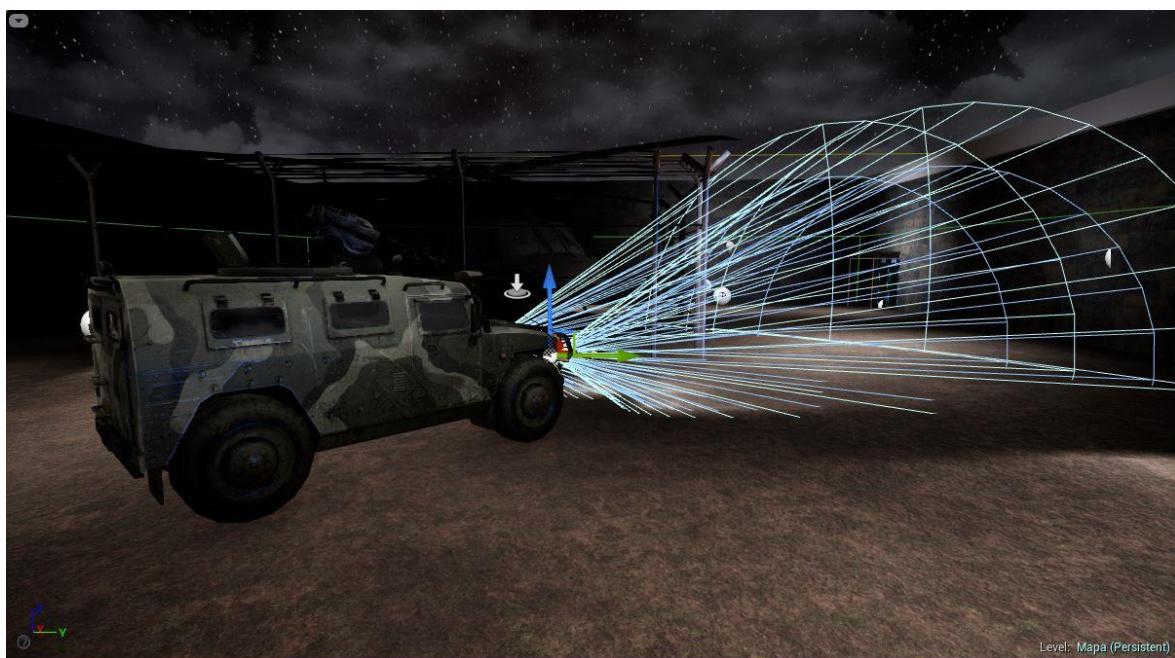


Figura 55: Ejemplo 1 iluminación únicamente con Spot Light sacada del proyecto



Figura 56: Ejemplo iluminación únicamente con Spot Light sacada del proyecto

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

5.3.1.1.3.1.1 Static lights

Las *Static Lights* tienen una calidad media, una mutabilidad reducida y un 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, es preciso definir qué es un mapa de luces, más conocido como **lightmap**. Un *lightmap* es una estructura de datos usada en el *lightmapping*, o lo que es lo mismo, una forma de cachear las superficies almacenando el brillo de las mismas 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 suelen aplicar a los objetos estáticos en aplicaciones 3D en tiempo real, como los videojuegos, puesto que es una forma de calcular la iluminación global con un coste computacional muy bajo.

Las *Static Lights* son luces que no se pueden mover ni modificar 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 Lights*.

Como las *Static Lights* 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 Lights* tienen la mayor calidad, una mutabilidad media y un coste de rendimiento medio. Aunque 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 modificar su posición, por eso la mutabilidad es media. Con todo, estos cambios solo afectan a la iluminación directa, ya que la iluminación indirecta es precalculada y no cambiará. Por último, 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 Lights* poseen una buena calidad, la mutabilidad más alta y un alto coste de rendimiento. Producen iluminación y sombras completamente dinámicas, pueden cambiar su posición, rotación, color, brillo, desvanecimiento, radio y todas las propiedades que posee una luz. La iluminación que producen no es calculada dentro de los *lightmaps* y no admiten iluminación indirecta.

Las *Movable Lights* configuradas para emitir sombras utilizan toda la escena en el cálculo de las sombras dinámicas, por lo que tienen 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 constituyen un elemento fundamental en el proyecto a la hora de conseguir la ambientación o estética deseada, ya que un material no es solo una imagen aplicada a una malla sino que además es capaz de definir el tipo de superficie del objeto al que se le aplica mediante el color, brillo, opacidad, rugosidad, etc. Cuando la luz de la escena incide sobre la superficie de un objeto, el material de dicho objeto determina cómo interactúa la luz con esa superficie, 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 utilizan nodos gráficos llamados **Material Expressions** dentro del **Material Editor**. Cada nodo contiene una parte de código **HLSL** (*High-Level Shading Language*), 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: algunos muy sencillos producen únicamente una imagen de textura (véase figura 57); otros más complejos simulan efectos reales, como el agua (véase figura 58).

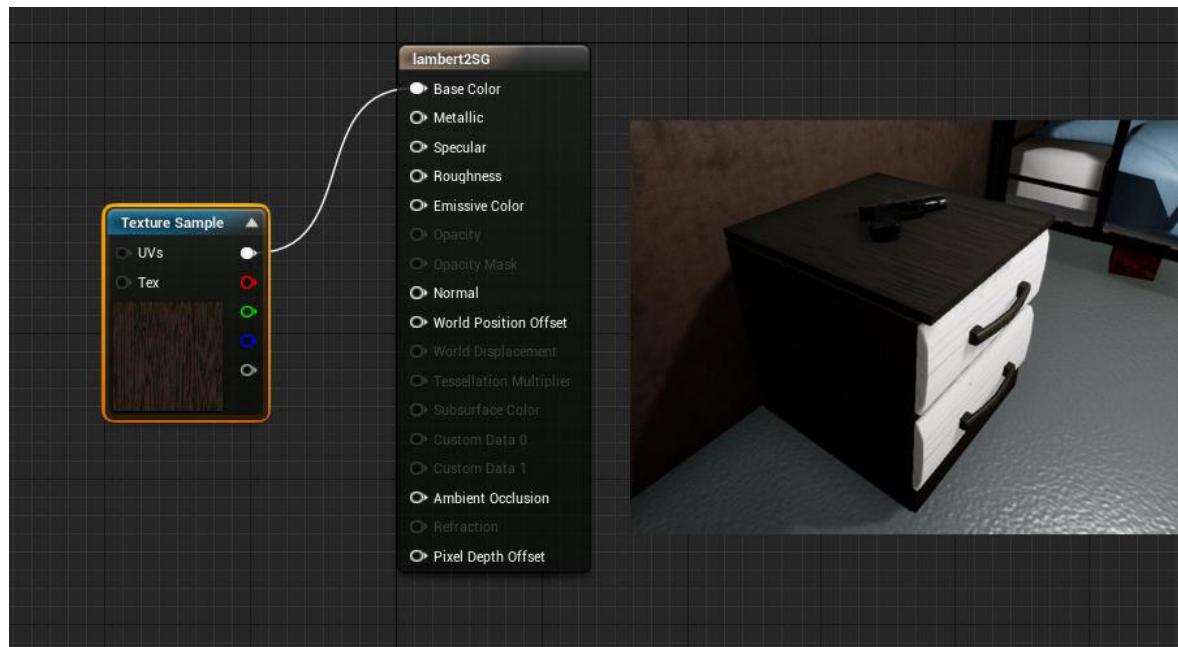


Figura 57: Material simple y su aplicación en el proyecto

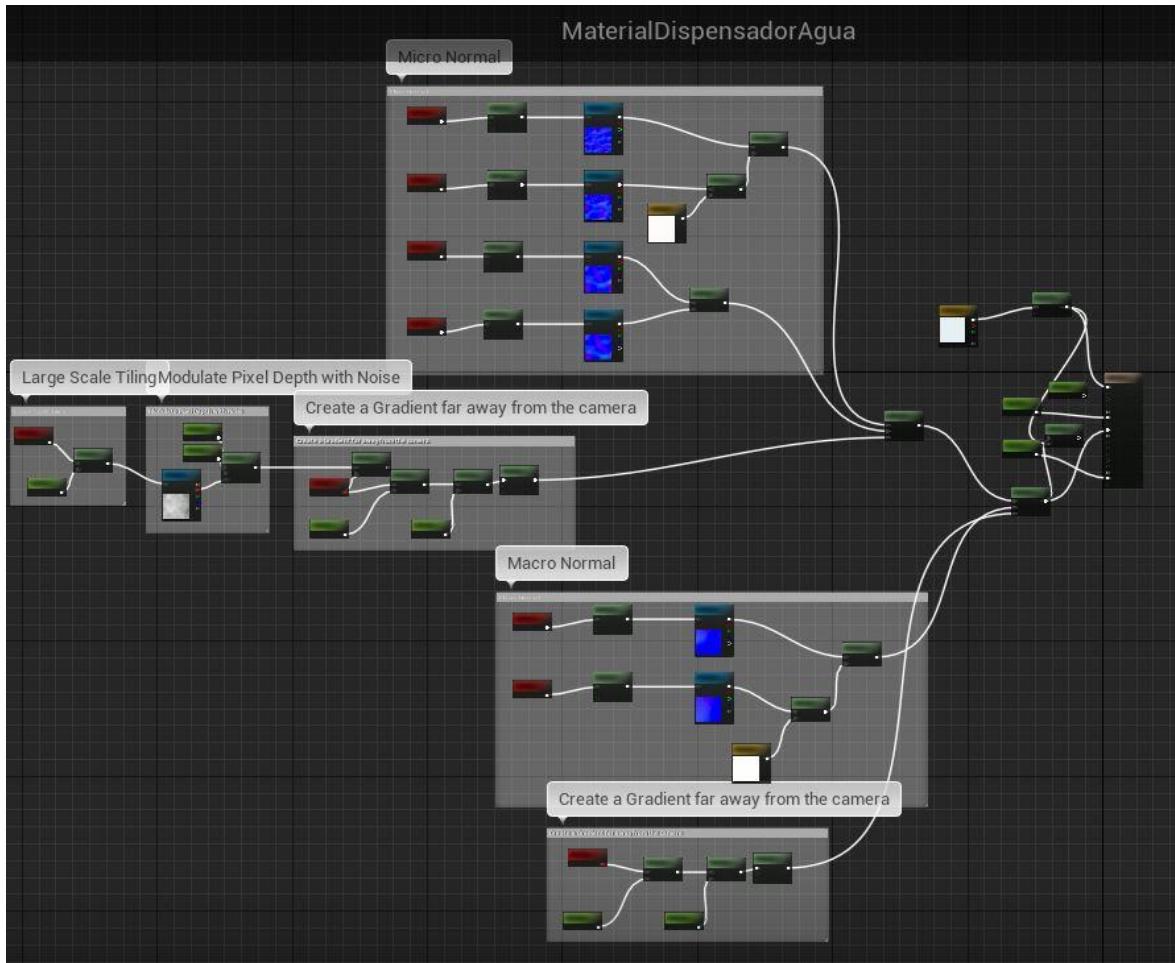


Figura 58: Material complejo (simulación de agua)

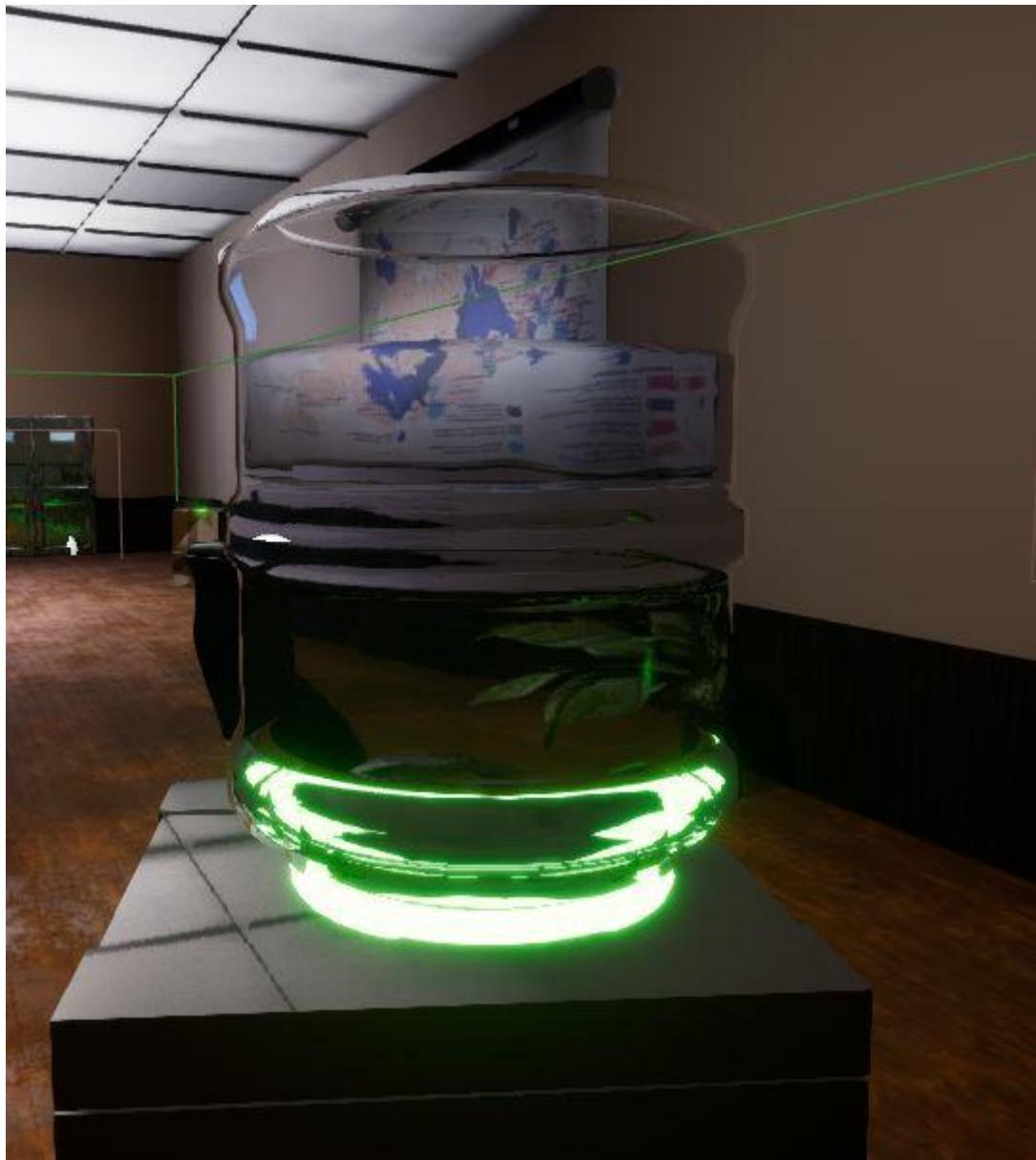


Figura 59: Aplicación de material complejo en el proyecto

Ambos materiales aparecen en este proyecto, tal como se aprecia en la primera foto de la mesita de noche cuyo material presenta una imagen de textura y, en la segunda, donde el agua del bidón tiene una refracción (se puede observar mirando el mapa de atrás) además de que la incidencia de la luz sobre esta va cambiando para dar la sensación de que el agua no está 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. Gracias al postprocesado los

artistas y diseñadores modifican la apariencia general de la escena y añaden efectos como la oclusión ambiental, el *Bloom*, *Blur*, cambio de color, etc.

5.3.1.1.4 Audio y sonido

El sonido es determinante para crear ambientes creíbles así como para permitir la inmersión del usuario en el juego. Abarca desde los sonidos ambientales en el nivel, a los sonidos interactivos de vehículos o armas, diálogos de personajes, etc. El audio en un juego puede intensificar la experiencia en el usuario o destruirla por completo.

Hacer que el audio de un juego suene de manera adecuada es bastante complicado, por ello *Unreal Engine 4* nos proporciona herramientas y características para moldear los sonidos del juego a nuestro antojo y así producir la sensación deseada.

El sistema de audio de *Unreal Engine 4* dispone 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 algunas de sus propiedades básicas como el Volumen o *pitch* y otras propiedades más complejas como la atenuación del sonido **Sound Attenuation**. Además UE4 también permite crear sonidos compuestos llamados **Sound Cues** 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 producir sonidos complejos o editar libremente sonidos existe el **Sound Cue Editor**, que es un editor de nodos gráfico parecido al editor de materiales o al de *Blueprints*.

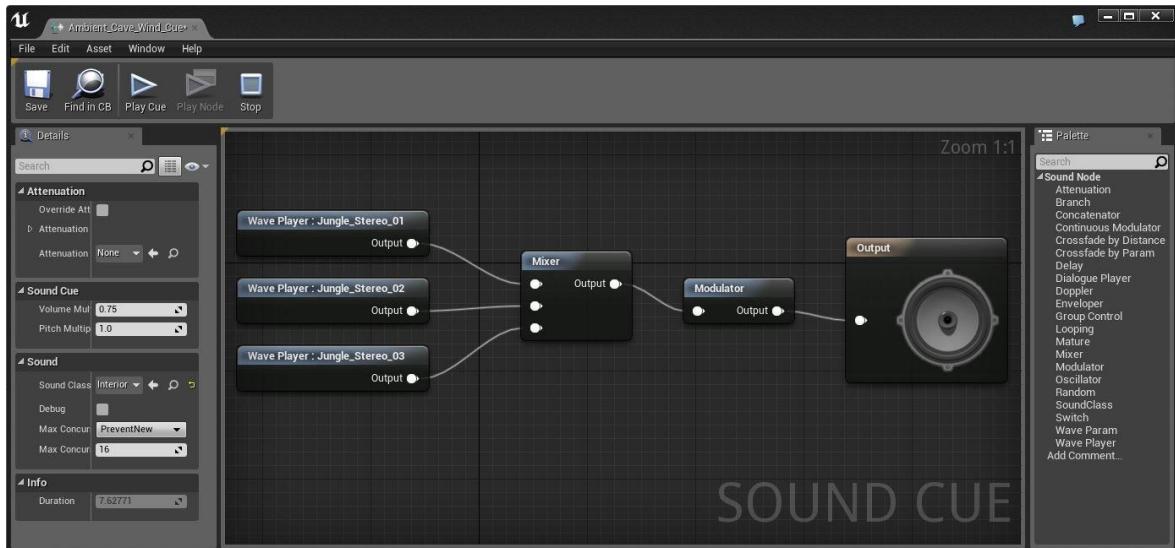


Figura 60: Ejemplo Sound Cue - Imagen sacada de la documentación oficial de Unreal Engine 4

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* y que se puede modificar libremente dentro del propio editor con el fin de añadir 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 por el cual se producen o modifican sonidos ambiente mediante **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** lo que permite 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 sujeto a los mismos cambios en la intensidad del mismo en función de la proximidad a la fuente del sonido (cuanto más cerca estás de la fuente del sonido mayor es la intensidad y viceversa).

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 dos 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 permanezca dentro de este radio el sonido no tendrá ninguna atenuación. En cambio, el *MaxRadius* es el radio máximo al que el sonido puede llegar; cuando el jugador sale de este radio no lo escuchará.

Cuando el jugador se encuentra entre el *MinRadius* y el *MaxRadius*, entonces el volumen del sonido irá desvaneciéndose linealmente desde el 100% (dentro del *MinRadius*) hasta el 0% (fuera del *MaxRadius*). La velocidad a la que ocurre este fundido depende del parámetro ***Distance Algorithm*** y de 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, que puede ser, por ejemplo, una esfera, una caja, un cono o una cápsula.

5.3.1.1.5 Animaciones

Unreal Engine 4 proporciona una gran cantidad de herramientas y de formas para poder trabajar con animaciones. Las cuatro herramientas principales que hemos utilizado para este proyecto son las siguientes:

- ***Skeleton Editor***: 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. Esta herramienta se ha usado por ejemplo en las armas que se vinculan a un *socket* colocado en la mano del esqueleto.
- ***Skeletal Mesh Editor***: permite añadir **LOD's** o asignar materiales a una *Skeletal Mesh*.
- ***Animation Editor***: trabaja 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 crea reglas para elegir qué animaciones reproducir o bien se puede acceder a herramientas más complejas como máquinas de estados para elegir diferentes animaciones.

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

El **blending animation** se puede definir como una mezcla entre dos animaciones, es decir, si estás andando por ejemplo y pasas a estado de correr, no hay un salto de la animación de andar al *frame* siguiente la animación de correr, sino que se produce una especie de transición (*blending*) entre las dos animaciones que permite que el paso de una animación a otra se vea de manera natural y fluida. Para llevarlo a cabo se dispone del **Blend Space**, herramienta de *Unreal* que posibilita la realización de una transición de manera sencilla entre dos o más animaciones. Para materializarlo se puede elegir entre las siguientes opciones:

- **Blend Space ID:** es el más sencillo. Consiste simplemente en mezclar dos o más animaciones pero admite 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 dos valores de entrada, por lo tanto la transición entre las animaciones se refleja en una especie de gráfico y dependerá del valor de los dos parámetros situados en los ejes X e Y. Estos dos valores podrían ser, por ejemplo, velocidad y dirección, lo que permite realizar un *blending* entre la animación de andar y correr dependiendo de la velocidad, pero además se haría otro *blending* entre las animaciones de correr hacia delante o hacia los lados dependiendo de la dirección.

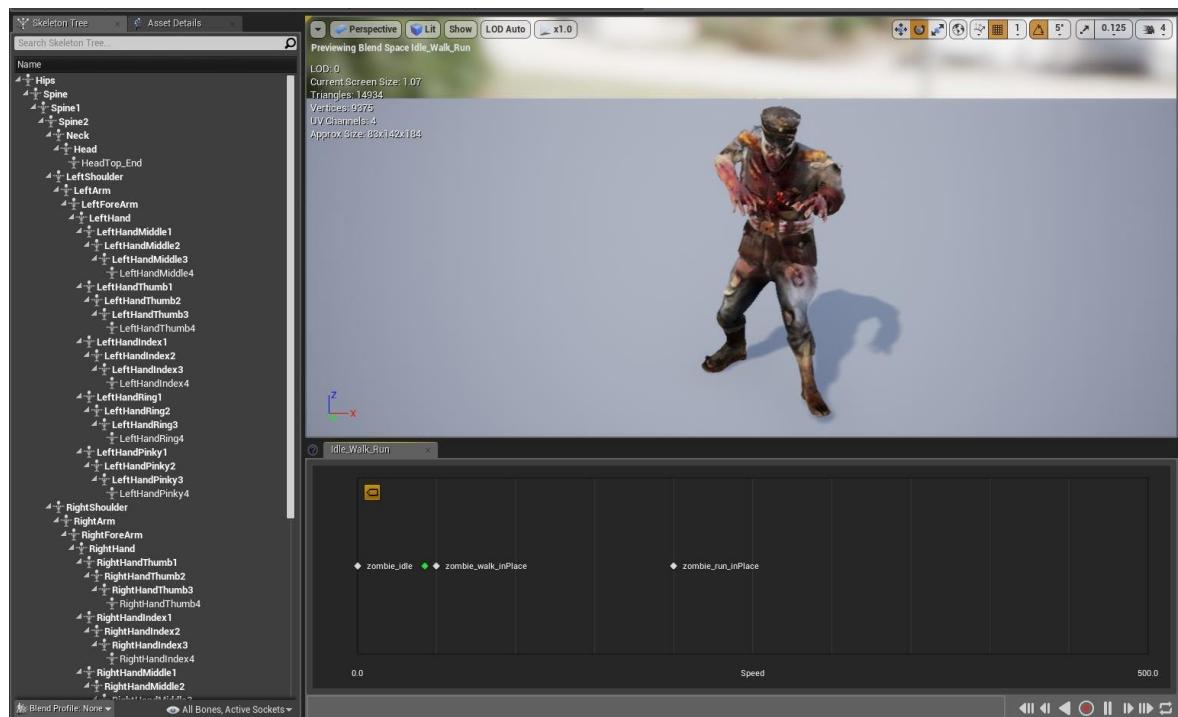


Figura 61: Ejemplo blending animation del proyecto

La imagen anterior muestra el *blending* para las animaciones del zombi que se ha realizado en este proyecto. Se trata de un *blending* entre tres animaciones: la animación *Idle* (cuando el zombi está quieto), la animación de andar y la animación de correr, que corresponden a los tres puntos blancos que se ven en el gráfico. El punto verde es el valor del parámetro que se utiliza para pasar de una a otra, por lo que en este caso estoy 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 empezará a hacer un *blending* entre la animación de andar y de correr.

5.3.1.1.5.2 Animation Blueprint

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

Dentro del *Animation Blueprint Editor* se pueden mezclar animaciones, controlar directamente la animación en algunos huesos determinados de una malla o crear una lógica encargada de seleccionar la animación necesaria en cada momento. Este editor posee una salida final que contiene la pose concreta en cada *frame* en la que se encuentra la malla.

Hay dos componentes fundamentales en el *Animation Blueprint* que se utilizan para crear la animación final en cada fotograma. Son estos:

- **EventGraph:** se encarga de actualizar valores que luego serán necesarios para elegir una animación.
- **AnimGraph:** conduce 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 admite 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** que pasa información de los huesos de un esqueleto a los huesos del otro. Después de seleccionar 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 dos esqueletos.

5.3.1.1.5.4 Notificación en animación

Las notificaciones en animación, conocidas en UE4 como *AnimNotifies* o solo *Notifies*, proporcionan una sistema de configuración de eventos que se producirán en un punto específico de una animación. Las notificaciones se usan normalmente para agregar efectos como sonidos o partículas durante la animación, pero también aceptan notificaciones personalizadas para poder ejecutar la lógica que se desee.

5.3.2 Desarrollo de mecánicas de juego

Las mecánicas de juego que se han realizado para este proyecto y que ya han sido detalladas en el GDD, se han creado mediante la programación visual que ofrece *Unreal (Scripting Visual Blueprint)*.

A priori podría parecer que esta manera visual de programar no va a dar resultado o que este va a ser poco satisfactorio; sin embargo, a medida que nos vamos adentrando en el mundo del *Blueprint* advertimos lo completo que es este sistema visual y las posibilidades que ofrece a la hora de realizar cualquier cosa que te imagines únicamente con la programación visual.

Una vez he aprendido a manejar este sistema de *Blueprints* he tomado conciencia del gran potencial de *Unreal* y de la gran cantidad de operaciones que se pueden desarrollar de una manera relativamente sencilla con este motor.

A partir de ahora, para facilitar la comprensión y con el objetivo de que se puedan diferenciar fácilmente **variables**, **funciones o eventos** y **clases**, les vamos a asignar un color diferente a cada una de ellas de acuerdo con las siguientes instrucciones:

- 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 apoya el resto para obtener información del juego, como puede ser las muertes del *player*, su puntuación, etc. Además es el *Blueprint* del personaje al que controlamos y, por tanto, el que más funciones y variables posee, pues controla toda la lógica del jugador.

A pesar de la extensión de este *Blueprint*, nos disponemos a realizar un análisis detallado aunque solo se mostrará la programación (visual) de aquellos aspectos que sea preciso conocer.

5.3.2.1.1 Componentes

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

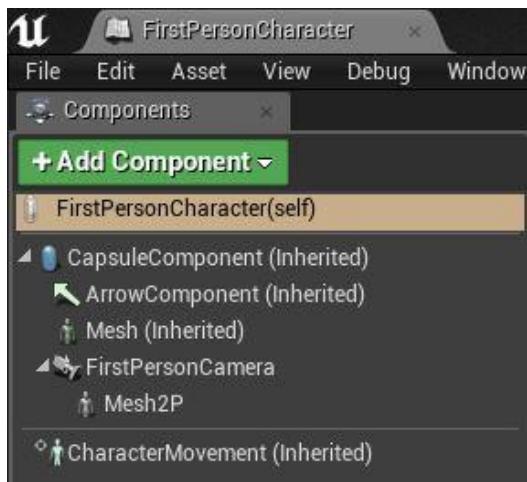


Figura 62: Componentes Blueprint personaje

- **CapsuleComponent:** este componente se crea por defecto en este tipo de *Blueprint*. Se encarga de controlar la colisión de nuestro personaje, por tanto es una cápsula que simularía la colisión de un cuerpo humano.
- **FirstPersonCamera:** este componente de tipo *CameraComponent* equivale 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° cuando no está apuntando y de 35° cuando 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, así como para que mantengan los brazos y el arma que sostienen en una posición constante respecto a la pantalla. Si el juego fuera en tercera persona, usaríamos el **Mesh** creado por defecto de *Unreal* que es hijo de la cápsula.

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

A estos componentes cabe añadir dos más, que son las armas que podremos equiparnos. Estas armas no figuran como componentes por defecto, pues al principio carecemos de arma. Para adquirirla debemos comprarla y solo podemos tener dos al mismo tiempo, por lo que 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 es el *Blueprint* más importante y el que más variables y funciones contiene, tanto es así que la mejor forma de mostrarlas es mediante una tabla.

(Siguiendo con la estética de la portada emplearemos los colores negro y amarillo para el contenido de esta y del resto de tablas.)

FirstPersonCharacter Variables		
Nombre	Tipo	Descripción
GunOffset	Vector	Es una pequeña cantidad que se suma a la posición del arma para obtener el punto exacto desde el que saldría el proyectil que se va a disparar.
BaseTurnRate	Float	Controla la velocidad de la cámara en el eje X.
BaseLookUpRate	Float	Controla la velocidad de la cámara en el eje Y.
UsingMotionControllers?	Boolean	Controla si se están usando mandos especiales para realidad virtual.
Equipada	Weapont Parent	Es la referencia al arma equipada en todo momento, la que recibiría las acciones de disparar, apuntar, recargar, etc.
Arma1	Weapont Parent	Corresponde a la primera arma de tu inventario, que puedes tener equipada o no.
Arma2	Weapont Parent	Corresponde a la segunda arma de tu inventario; puede estar equipada o no.
Apuntando	Boolean	Variable de tipo <i>booleano</i> que es verdadera cuando estás apuntando con la mirilla del arma y falsa en el resto de casos.
PosNormal	Transform	Transformación que da la posición, rotación y escala de los brazos cuando no se está apuntando con el arma.

PosApuntadoAk	<i>Transform</i>	Transformación que da la posición, rotación y escala de los brazos cuando se está apuntando con una AK-47.
PosApuntadoShotGun	<i>Transform</i>	Transformación que da la posición, rotación y escala de los brazos cuando se está apuntando con una escopeta.
PosApuntadoM4	<i>Transform</i>	Transformación que da la posición, rotación y escala de los brazos cuando se está apuntando con una M4A1.
PosApuntadoM60	<i>Transform</i>	Transformación que da la posición, rotación y escala de los brazos cuando se está apuntando con una M60.
PosApuntadoUMP	<i>Transform</i>	Transformación que da la posición, rotación y escala de los brazos cuando se está apuntando con una UMP45.
PosApuntadoG36	<i>Transform</i>	Transformación que da la posición, rotación y escala de los brazos cuando se está apuntando con una G36.
PosApuntadoScarL	<i>Transform</i>	Transformación que da la posición, rotación y escala de los brazos cuando se está apuntando con una Scar-L.
NuevaPosApuntado	<i>Transform</i>	Resultado de la función <code>getPosApuntado</code> , que restituye en esta variable una de las transformaciones anteriores, lo que facilita el código ya que la lógica de elegir una de las posiciones anteriores estará en una función y en el resto del código solo usaremos esa variable.
Puntuación	<i>Integer</i>	Esta variable corresponde con el número de puntos o dinero que tiene el jugador.
VidaActual	<i>Integer</i>	Cantidad de vida actual del jugador. Si es igual a 0 estás muerto.
VidaMáxima	<i>Integer</i>	Valor máximo que puede alcanzar la vida del jugador: por defecto son 10 puntos, aunque con el <i>power-up</i> de

		resistencia se incrementarían hasta los 15.
TiempoRecargaBase	<i>Float</i>	Tiempo base que se añade a cada uno de los tiempos de recargas de las armas.
Revivir	<i>Boolean</i>	Variable <i>booleana</i> que es verdadera cuando has adquirido el <i>power-up</i> de revivir y falsa cuando no lo has comprado.
Reviviendo	<i>Boolean</i>	Variable <i>booleana</i> que dice si estás en estado de revivir.
TimmerHandleRecuperaciónVida	<i>TimmerHandle</i>	Variable usada para reanudar o pausar la recuperación de vida del jugador.
TiempoRegeneraciónVida	<i>Float</i>	Tiempo disponible para pasar sin ser dañado por un enemigo <i>y</i> para poder empezar a recuperar vida automáticamente con el tiempo.
Sprint	<i>Boolean</i>	Variable <i>booleana</i> que es verdadera si estás corriendo y falsa en cualquier otro caso.
CambiandoArma	<i>Boolean</i>	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).
PartidaReference	Partida	Variable que guarda una referencia de un objeto tipo Partida con el que se pueden realizar acciones como actualizar el <i>ranking</i> .
EnPausa	<i>Boolean</i>	Variable <i>booleana</i> verdadera mientras el juego está en el menú de pausa y falsa en cualquier otro caso.
Ranking	<i>Ranking</i>	Variable que guarda una referencia a un objeto de tipo <i>ranking</i> a partir del cual podremos acceder a la información del <i>ranking</i> .

RankingSubClass	<i>SaveGame</i>	Variable que almacena una referencia a un objeto de tipo <i>SaveGame</i> a partir del cual podremos guardar nuevos datos.
Nombre	<i>String</i>	Almacena el nombre del jugador en caso de que haya alcanzado los números necesarios para acceder al <i>ranking</i> .
ZombisMatados	<i>Integer</i>	Cantidad de zombis matados.

Figura 63: Tabla de variables del personaje

5.3.2.1.2 Funciones

Igual que con las variables en el apartado anterior, en este se incluye una tabla explicativa de las funciones que figuran en el *Blueprint* del *FirstPersonCharacter*, donde se explica brevemente la lógica de cada función. Las funciones importantes serán desarrolladas con mayor detalle después de la tabla acompañadas de 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
ConstructionScript	Constructor de la clase, en él se establece la vida máxima y la vida actual del personaje.
GetEquipoID	Función que devuelve un <i>array</i> con los IDs de las armas que tienes en el equipo actualmente. Es usado por ejemplo cuando te acercas a un arma para comprarla, ya que primero compruebas que no esté ya en el equipo.
EquiparArma	Añade un arma al equipo al comprarla. Si ya tienes dos armas en el equipo (este es el máximo), entonces se llamará a la función sustituirArma . Lo veremos en mayor detalle más abajo.
CambiarArma	Realiza el cambio de arma entre las dos armas que tienes en el equipo, es decir, si estás 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 un arma no podrás cambiarla.

SustituirArma	Sustituye el arma equipada por el arma comprada cuando adquieres un arma y ya tienes el equipo lleno.
GetPosApuntado	Devuelve la transformación que corresponde al arma equipada en posición de apuntado.
RestarPuntuacion	Recibe como parámetro de entrada los puntos que se deben restar, los descuenta de la puntuación del jugador y además llama al método restaPuntos del HUD para que aparezca por pantalla la animación de los puntos restados en rojo.
SumarPuntuacion	Recibe como parámetro de entrada los puntos por sumar, los agrega a la puntuación total del jugador y además llama al método sumaPuntos del HUD para que aparezca por pantalla la animación de los puntos sumados en verde.
ComprobarPuntos	Recibe como parámetro de entrada los puntos necesarios para realizar una compra y tiene como parámetro de salida un <i>booleano</i> que será <i>true</i> en caso de que tengas los puntos necesarios o <i>false</i> cuando no tengas los puntos necesarios para hacer la compra.
RestaVida	Esta función se encarga de restar un punto de vida a la actual del jugador cada vez que reciba un impacto. Además tiene otras funciones importantes (como controlar si podemos revivir), que veremos en detalle abajo.
SumaVida	En esta función se llama de manera automática cada X tiempo siendo X el valor de la variable tiempoRegeneracionVida . Suma 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.
EnRanking?	Función que devuelve un <i>booleano</i> que será <i>true</i> si tienes los puntos suficientes para entrar al <i>ranking</i> y falso si no has conseguido los puntos necesarios para entrar al <i>ranking</i> .
ActualizarRanking	Se encarga de actualizar el <i>ranking</i> . Controla en todo momento que si quedas 2º por ejemplo, el que estaba 2º pase a ser el 3º y tú ocupes su lugar.
SumarKill	Aumenta el número de zombis matados de la variable zombisMatados .
Revivo	Revive al personaje una vez que ha muerto y había comprado el <i>power-up</i> de revivir (solo lo revive la primera vez que muere).

	Además esta función se encarga de deshabilitar el <i>input</i> mientras estás reviviendo para que no puedas moverte, de volverlo a activar una vez que ya has revivido y de llamar a PlayAnimationReviviendo (función de HUD). Una vez que hayamos revivido nuestra vida estará al máximo.
RegenerarVida	<i>CustomEvent</i> llama a la función SumaVida . La utilidad de este <i>custom event</i> es que permite el uso de un setTimerByFunctionName sobre él para que sea llamado automáticamente cada cierto tiempo, y haga que la regeneración de vida sea automática.
DejarDeApuntar	<i>CustomEvent</i> para dejar de apuntar y todo lo que ello conlleva. El arma y los brazos pasan a su posición por defecto, la velocidad de movimiento vuelve a la normalidad (nos movemos más lentos mientras apuntamos) y el FOV de la cámara pasaría de 35º a 90º, que es su estado natural.
DejarDeCorrer	<i>CustomEvent</i> para dejar de correr y, por tanto, para que la animación pare de correr y regrese a la velocidad de movimiento de andar. El cambio de animación de correr a la animación <i>Idle</i> se haría con un <i>blending</i> de 0.25 seg.
EventAnyDamage	Evento que se activa cuando recibimos cualquier tipo daño. Como parámetro de entrada tenemos el daño que nos han causado, el cual se lo pasamos a la función RestaVida , que es la responsable de 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).
EventBeginPlay	Evento de <i>Unreal</i> que se llama cuando el juego comienza para ese actor. En este evento se inician atributos necesarios para el juego, como por ejemplo se crea el actor de tipo partida que se encargará de toda la información de la partida. También se comprueba si existe algún archivo con un <i>ranking</i> 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.
InputAction Jump	Llama al método de saltar de <i>Unreal</i> .

<i>InputAction CambioArma</i>	Este evento contiene la lógica del cambio de arma que se explicará abajo en detalle.
<i>InputAction Fire</i>	Cuando pulsamos el botón de disparar llamamos al FireEvent del arma equipada, pero antes de esto se comprueba que no estamos cambiando de arma y si el arma es automática o no. Este evento también se explicará en detalle puesto que la lógica para el disparo de armas automáticas precisa de una foto para su comprensión.
<i>InputAction Recargar</i>	Llama al evento recargar del arma equipada.
<i>InputAction Sprint</i>	<p>En este evento el personaje pasa de andar a correr _ para ello dejará de apuntar (si estaba apuntando) ya que no se puede correr mientras se apunta y dejará de recargar (si estaba recargando) ya que no se puede recargar y correr al mismo tiempo_, se empezará a reproducir la animación de correr y cambiará la velocidad máxima del jugador de 500 a 1000.</p> <p>Para añadir un poco de dificultad se han añadido una serie de limitaciones a esta mecánica:</p> <ul style="list-style-type: none"> • No se puede apuntar mientras corres. Si se está corriendo y se apunta, automáticamente se dejará de correr y se volverá a la velocidad de andar. Por el contrario, si se está apuntando y se echa a correr, se dejará de apuntar automáticamente para empezar a correr. • No se puede recargar mientras corres. Si se está corriendo y se recarga, se dejará de correr automáticamente. Si se está recargando y se empieza a correr, la recarga quedará anulada como si nunca se hubiera intentado recargar el arma. • No se puede correr durante más de tres segundos seguidos. A los tres segundos se para de correr automáticamente y se empezará a andar de nuevo.
<i>InputAction Apuntar</i>	<p>Es el evento que nos permite cambiar entre apuntar desde la cadera o apuntar con la mira del arma.</p> <p>Este evento se explicará en detalle en la sección de Funciones en detalle. (Véase apartado 5.3.2.1.3.1 Apuntar).</p>

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

Figura 64: Tabla de funciones del personaje

5.3.2.1.3 Análisis de las funciones importantes en detalle

Debido al elevado número de funciones que hay en el *Blueprint* de nuestro personaje, las hemos analizado someramente en el apartado anterior sin entrar en cómo están programadas ni añadir imágenes de la lógica de las funciones. Sin embargo, dada la importancia que adquieren algunas de ellas, creo conveniente profundizar en su explicación y añadir imágenes aclaratorias.

Una aclaración en relación con los nodos de las imágenes de las funciones que veremos a continuación es que al tener que alejarnos mucho para mostrar toda la lógica esta se vuelve ilegible, por lo tanto se han comprimido los nodos únicamente para que sean legibles en las imágenes.

5.3.2.1.3.1 Apuntar

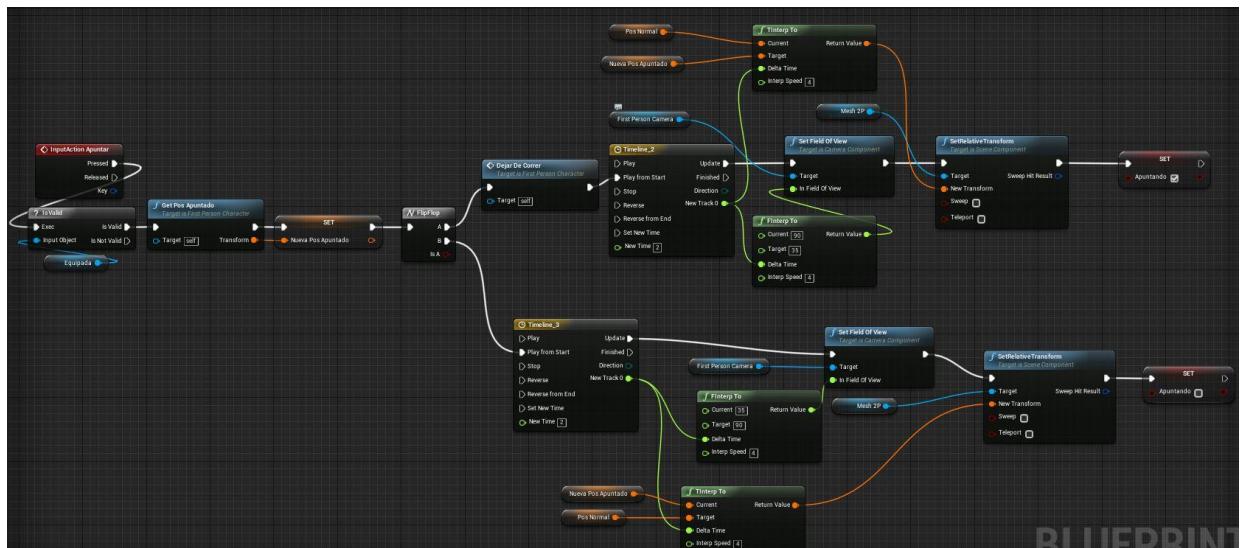


Figura 65: Función apuntar

En esta función, el primer paso es comprobar si el arma equipada es válida, lo que equivaldría a comprobar que el arma equipada no fuera nula si estuviéramos programando en código. Esta

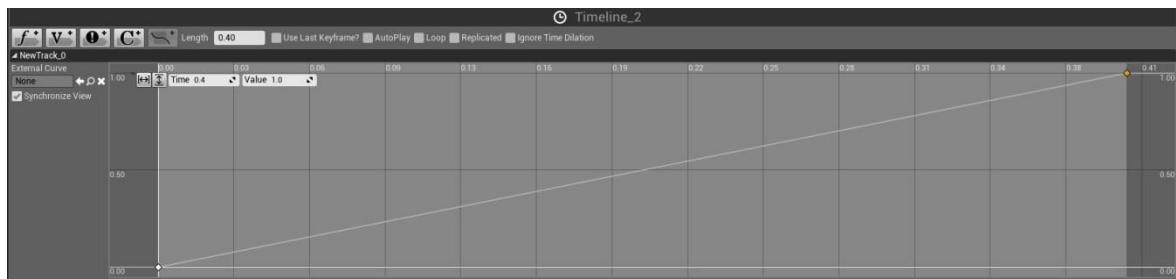
comprobación es necesaria en vista de que el jugador puede pulsar el botón de apuntar incluso cuando aún no tiene arma (al principio de la partida). Si el arma equipada es válida, se continúa con la lógica de la función; si por el contrario no tuviéramos ningún arma equipada, no se haría nada.

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

El siguiente paso es un nodo **Flip/Flop** que actúa como una especie de alternador. La primera vez que entramos a la función se ejecutará la parte de arriba, la segunda vez la de abajo, la tercera la de arriba y así sucesivamente, es decir, va alternando entre las dos salidas de ejecución. La ventaja de usar este Flip/Flop es que no se precisa comprobar si se está apuntando para dejar de hacerlo o viceversa, ya que vamos alternando.

El nodo Flip/Flop tiene dos salidas:

- **Salida A:** Cuando la ejecución continúa por la primera salida del Flip/Flop, entonces pasaremos a estar apuntando. Para ello llamamos a la función **dejar de correr**, que detendrá el *sprint* en caso de que estuviéramos corriendo. Seguidamente se hace una interpolación lineal usando el nodo de tipo **TimeLine** llamado **Timeline_2** (véase figura 66). Este nodo nos devuelve un valor dentro de una interpolación definida en su gráfica que se usa para obtener una transformación intermedia entre las transformaciones origen (**posNormal**) y destino (**nuevaPosApuntado**) 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 dos posiciones de apuntado, también se practica una interpolación del **FOV** (campo de visión) utilizando la misma interpolación definida por la gráfica de **Timeline_2**. Por último, se pone la variable booleana **apuntando** a verdadera.
- **Salida B:** Cuando la ejecución continúa por la segunda salida del Flip/Flop, entonces dejamos de apuntar para volver a la posición normal. Se crea una interpolación entre la posición de apuntado (ahora será la transformación de origen) y la posición normal (ahora la transformación destino). Se establece también una interpolación en el **FOV** desde 35° hasta 90° (su valor normal) y, por último, se coloca la variable **apuntando** a falsa.



La foto de arriba corresponde a la gráfica de la interpolación lineal que se practica entre las dos 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

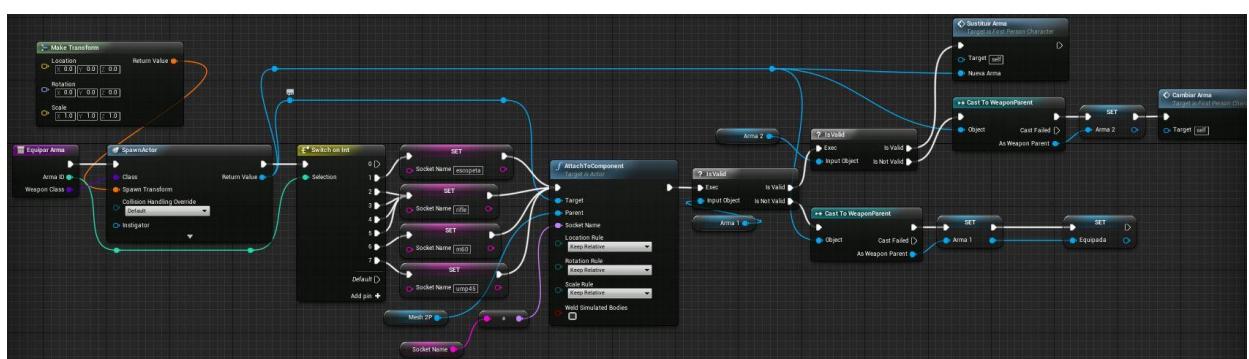


Figura 67: Función equipar arma

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

El primer paso de esta función es crear un actor cuya clase será la que se nos proporcione por parámetro. El segundo paso vincula el arma (el actor que acabamos de crear) a los brazos del personaje en un punto específico llamado *socket* para lo que recurrimos al nodo **AttachToComponent** que recibe varios parámetros de entrada entre los que se encuentran:

- **Target:** actor sobre el que se va a practicar el *attach to component*, es decir, el actor que se va a unir a otro componente (una malla en este caso). Nuestro *target* es la arma que hemos creado.
- **Parent:** 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:** nombre del *socket* que actúa de guía para saber dónde tiene que colocarse el actor al cual vamos a unir el nuestro.

En resumen, unimos la nueva arma (*target*) a los brazos del personaje (*parent*) en un punto específico de los brazos (*socket*). Existen cuatro *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. Los tipos de sockets son rifle, shotgun, m60, ump; por lo tanto, si el arma comprada es una UMP45 se colocará en el socket “ump”, pero si es por ejemplo una AK47 o una G36 se unirá con el socket “rifle”.

Por último, nos queda ver en qué espacio del inventario se colocará el arma comprada, para lo que se dispone de tres posibilidades:

- No tenemos ningún arma comprada. Se guardará una referencia al arma comprada en la variable **Arma1**.
- Solo tenemos un arma comprada. Se grabará una referencia al arma comprada en la variable **Arma2**.
- Tenemos dos armas compradas y por tanto el inventario está lleno. Se llama al método sustituir arma que se encargará de comprobar cuál es el arma que tenemos equipada y sustituirá esa arma por la comprada, o lo que es lo mismo, si tenemos equipada el **Arma1**, la referencia a la nueva arma comprada se guardará en la variable **Arma1**. Si por el contrario está equipada el **Arma2**, la referencia a la nueva arma comprada se guardará ahora en la variable **Arma2**.

Antes de continuar, conviene aclarar que los *sockets* no aparecen por defecto sino que 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 68: Ejemplo de socket en el editor de Unreal Engine 4.

A la izquierda aparecen los huesos de los que depende este *socket*, que se moverá a la par que lo haga cualquiera de esos huesos. En el centro podemos apreciar dónde se encuentra el *socket*. A la derecha

figura información útil sobre el *socket* como su nombre, el nombre de su hueso padre o su posición, rotación y escala.

5.3.2.1.3.3 Cambiar de arma

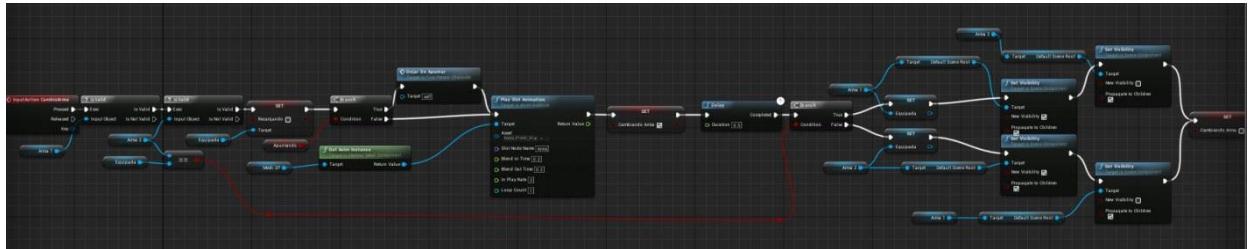


Figura 69: InputAction cambiar de arma

El cambio de arma constituye un evento del tipo *InputAction* que se activa pulsando el botón triángulo (si jugamos con mando de ps4) o con la ruleta del ratón hacia arriba o hacia abajo. Por qué es un evento y no una función se debe a que esta función necesitaría un *delay*, cosa que las funciones no permiten pero los eventos, sí.

Cuando se desea cambiar de arma, el primer paso es comprobar que tenemos dos armas en el inventario, puesto que si no tenemos ninguna o solo tenemos una no podremos cambiar de arma. Hecha esta comprobación, se verifica cuál de las dos es la que tenemos equipada.

Una vez que el arma está equipada, ponemos la variable **recargando** a falso (si no se está recargando ya estará en falso y no cambiará nada, pero si se está recargando al poner esta variable a falso se anulará la recarga de esa arma). Seguidamente se llama a la función **dejarDeApuntar** para que, si se estaba apuntando, el arma vuelva a su posición original antes de cambiarla.

El tercer paso _ sabiendo que no estamos recargando, ni estamos apuntando_ será el de cambiar un arma por otra. Para ello reproducimos la animación de cambiar arma, ponemos la variable **cambiandoArma** a verdadero y esperamos 0.5 segundos.

El último paso consiste en ocultar el arma que tengamos equipada, hacer visible la nueva arma que vamos a equipar (la que teníamos antes en el inventario) y volver a colocar la variable **cambiandoArma** en 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 se produzca en la parte de la animación, donde los brazos no se muestran en la pantalla, de esta forma queda oculto el “efecto mágico” de que un arma desaparece y aparece la otra.

5.3.2.1.3.4 Disparar

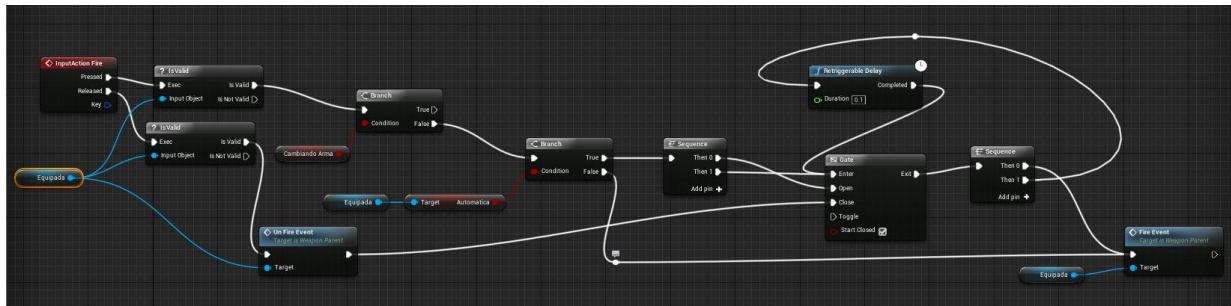


Figura 70: InputAction disparar

A pesar de la apariencia confusa de la imagen, es más sencillo de lo que cabe pensar. 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 contiene dos acciones:

- En primer lugar, cuando pulsamos el botón vinculado a este evento (gatillo derecho del mando o click izquierdo del ratón), comprobamos si tenemos una arma equipada, ya que sin arma equipada no podemos disparar. Seguidamente, verificamos que no nos encontramos en mitad de un cambio de arma y, si es así, continúa la ejecución. Después revisamos si la arma equipada es automática o no, tras lo que se abre una doble opción:
 - **Si no es automática:** se llama a la función *FireEvent* del arma equipada una sola vez.
 - **Si es automática:** se ejecutará una secuencia que, en primer lugar, abre una puerta (*Gate*) que permite que el flujo de ejecución pase a través de ella y, en segundo lugar, llama a otra secuencia que se encargará de llamar al *FireEvent* del arma equipada y, después, de volver a entrar por la puerta para regresar de nuevo a esta segunda secuencia, así se consigue un disparo automático mientras la puerta esté abierta, o lo que es lo mismo, mientras se mantenga el botón de disparar pulsado.
- En segundo lugar, cuando soltamos el botón, se llama a la función *UnFireEvent* del arma equipada que activa la lógica de dejar de disparar como desactivar el *muzzle*. Después de esto, se cerrará la puerta (*Gate*) para bloquear 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, lo que permite el disparo automático).

5.3.2.1.3.5 Recibir daño

En este apartado se abordará no solo cuándo nuestro personaje recibe daño, sino cómo afecta esto en la vida del personaje, es decir, cómo disminuye o aumenta su vida después de haber sido dañados.

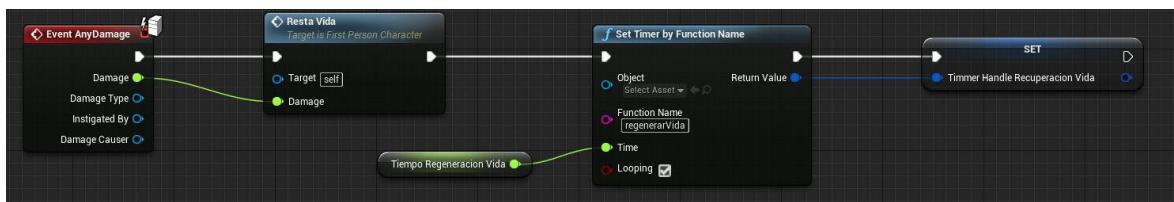


Figura 71: InputAction recibir daño

Como podemos apreciar en la foto de arriba, el evento **AnyDamage** es bastante sencillo.

Por su relación con este evento, pasamos a explicar cómo se lleva a cabo la regeneración de vida del personaje. En primer lugar, conviene saber que este evento es llamado automáticamente cada vez que el personaje recibe daño de algún tipo, en nuestro caso el daño recibido será el del ataque de un zombi. Inmediatamente, este evento llama a la función **RestaVida** y, seguidamente, a la función **regenerarVida**, pero esta función no se llama directamente sino 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.

En el funcionamiento de **SetTimerByFunctionName** podemos observar (véase figura 71) que recibe cuatro parámetros como entrada, que 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 está vacío y, cuando esto ocurre, 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 que deseamos ejecutar su lógica. En nuestro caso el nombre es “**regenerarVida**”.
- **Time:** cuando llamamos a esta función, no se ejecuta la lógica de la función **regenerarVida** inmediatamente, sino que se ejecutará después del tiempo indicado en este parámetro. En nuestro caso el tiempo que hay que esperar para recuperar vida viene dado por la variable

tiempoRegeneraciónVida 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 **tiempoRegeneraciónVida** tiene un valor de 2 entonces, una vez nos han quitado vida (hemos llamado a la función **restaVida**), pasados los 2 segundos se ejecutará la función **regenerarVida** y se volverá 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 así solo tendremos que comprobar cuándo hemos llegado al máximo de vida para poder desactivar la llamada automática de la función **regenerarVida** y no seguir sumándonos vida. Por último, nos guardamos la salida del **SetTimerByFunctionName** en la variable **TimmerHandleRecuperaciónVida**, que es la que nos servirá luego para desactivar la llamada automática a esta función.

5.3.2.1.3.5.1 Restar vida

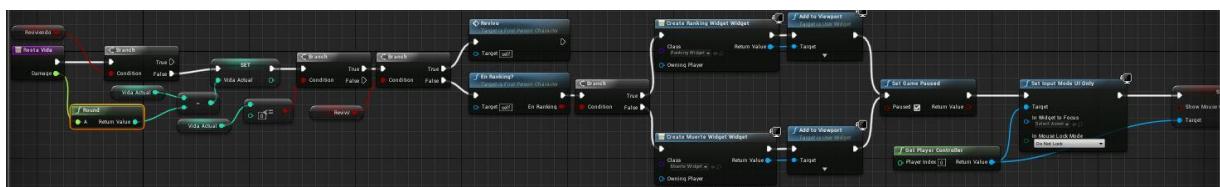


Figura 72: Función restar vida

A pesar de la sencillez de esta función, 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.

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

En segundo lugar, como 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 hemos de redondear este daño, pues puede no ser un entero según la zona de impacto y nuestra vida sí es un entero. En consecuencia, debemos transformar el daño que nos han hecho en un entero, lo

restamos a nuestra vida actual y guardamos este nuevo valor de vida actual en la variable **vidaActual**.

A continuación, comprobamos si esta vida es menor o igual a 0. Si la vida es mayor que 0 ya no continuamos con la ejecución, pero si la vida es menor o igual a 0 significa que hemos muerto y en este caso comprobamos si tenemos disponible el *power-up* de revivir. Si es así, 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, habremos muerto definitivamente en esta partida.

En el supuesto de que hubiéramos muerto, entonces verificamos si disponemos de los puntos suficientes o no para entrar en el *ranking*, de lo que se encarga la función **EnRanking?**, que devuelve un booleano que:

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

Mientras se está mostrando una de las dos pantallas disponibles, la de muerte o la de *ranking*, se pondrá el juego en pausa para que este no se siga ejecutando 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

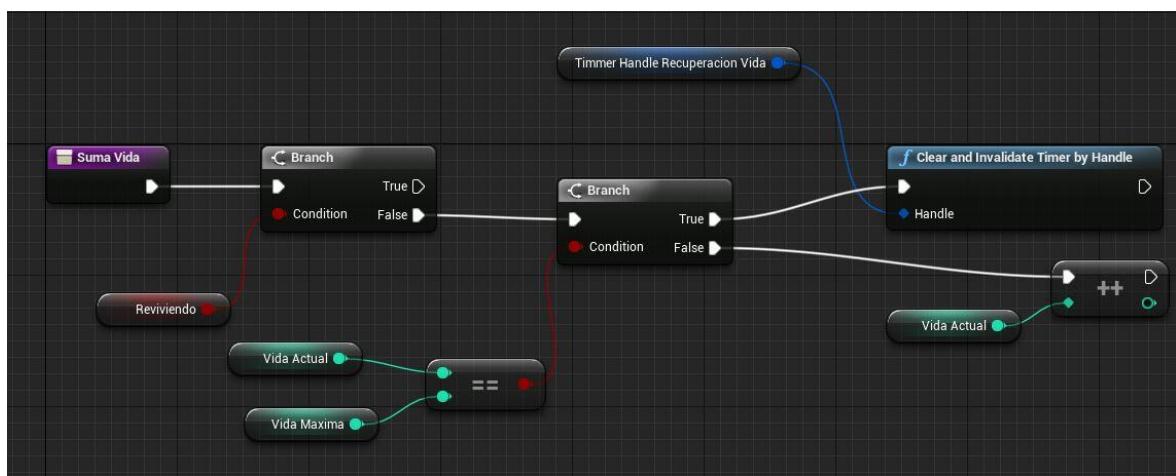


Figura 73: Función suma vida

El evento **regenerarVida**, al que se llamaba con el nodo **setTimerByFunctionName** citado al principio de este apartado, llama a la función **sumaVida**, que es la que contiene toda la lógica.

Lo primero que hace esta función es verificar si estamos reviviendo, dado que si estamos haciéndolo es imposible sumarnos vida. Si no estamos reviviendo, comprobamos si la vida actual es igual a la vida máxima. A continuación se abren dos opciones:

- Si hemos alcanzado la vida máxima, recurriremos a la variable **TimmerHandlerRecuperacionVida** para poder parar la llamada automática a la función de regenerar vida. Esto lo efectuaremos 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 incrementaremos nuestra vida actual en 1.

5.3.2.1.4 Animaciones

Las animaciones de nuestro personaje y el modo mediante el cual se han incorporado al proyecto merecen un análisis detallado.

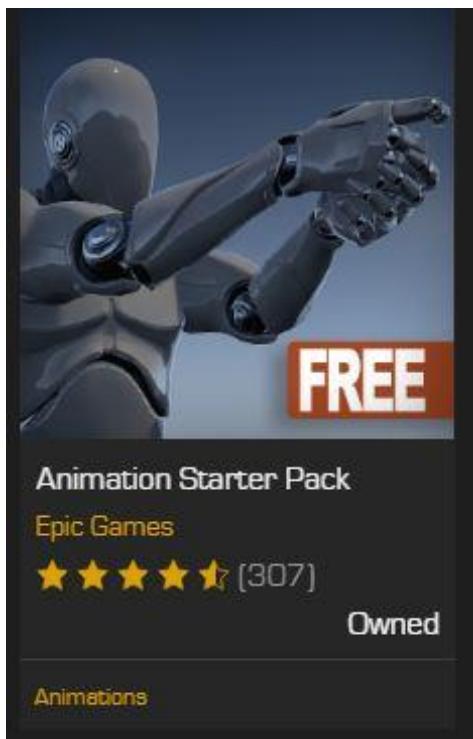


Figura 74: Animation Starter Pack de Unreal

Por defecto, el *Blueprint* del *FirstPersonCharacter* 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 no eran suficientes para un personaje como el de este proyecto que precisa, por ejemplo, de una animación de recarga de arma.

Puesto que era necesario incrementar el número de animaciones, recurrimos al bazar de *Unreal Engine* que incluye un *pack* totalmente gratuito 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 están preparadas para un juego en 3^a persona, por lo tanto el esqueleto y la malla de estas animaciones presentan un personaje completo y no solo los brazos, que es lo que precisamos en el proyecto. En consecuencia, me he visto obligado a adaptar las animaciones del esqueleto en tercera persona al de primera persona para lo que hemos utilizado 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* ha sido necesario 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. 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).

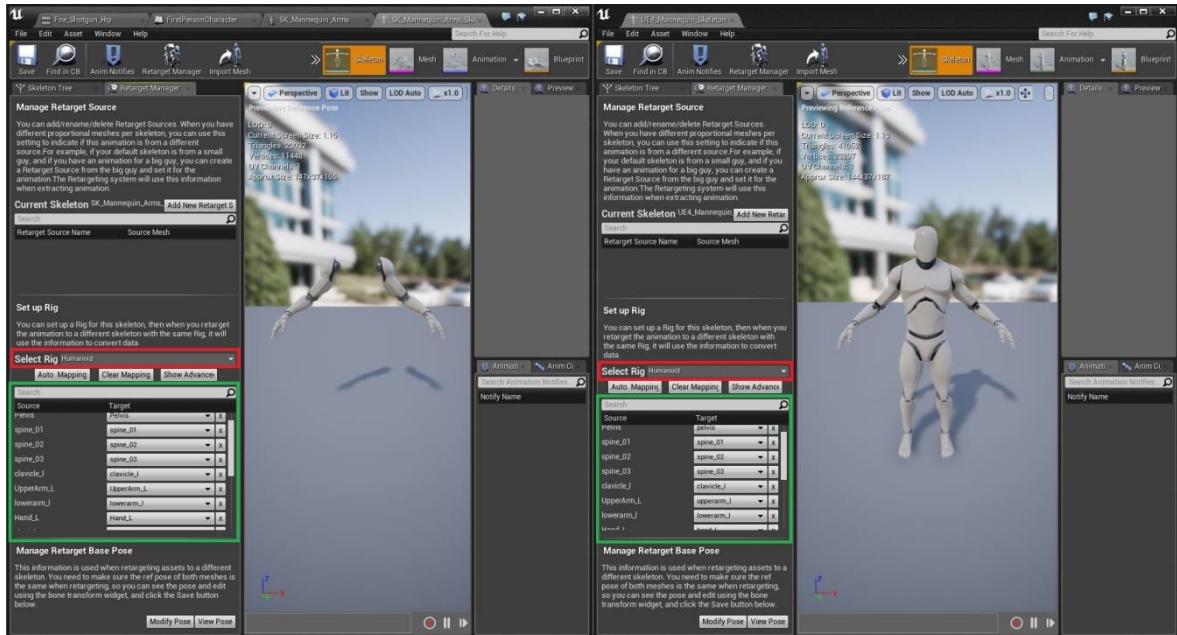


Figura 75: Retargeting Animation del proyecto

Con ello duplicamos todas las animaciones que teníamos en el esqueleto de origen (derecha) en el esqueleto destino (izquierda), por tanto ya tenemos las animaciones en 3^a persona (arriba) y en 1^a persona (abajo) como ilustra la siguiente imagen extraída del proyecto.



Figura 76: Animaciones en primera y tercera persona del proyecto

Llegados a este punto, ya disponemos de las animaciones para ser usadas en el proyecto, pero en el proceso de adaptar animaciones de 3^a a 1^a persona surgió un inconveniente: estas aparecieron rotadas respecto al hueso central llamado ***Root***.

Para intentar solucionar el problema, giré este hueso en las animaciones que aparecían rotadas sin éxito porque, al ser el hueso central, todos dependen de él y animaciones originales como *Idle* y *Jump* aparecían ahora mal rotadas. Luego probé a hacer un ***Animation Montaje*** de cada animación mal rotada para modificar en esta la rotación de los huesos pero tampoco funcionó. Por último, 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 inconvenientes, como por ejemplo que a mitad de la 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.

Así que la forma de resolverlo fue ir retocando animación a animación y comprobando en qué *frames* se iba la animación de la cámara o en cuáles la atravesaba y ocupaba casi la totalidad de la pantalla.

Se puede ver un ejemplo de esto en la siguiente foto donde aparecen las gráficas modificadas para conseguir que la animación funcione correctamente.

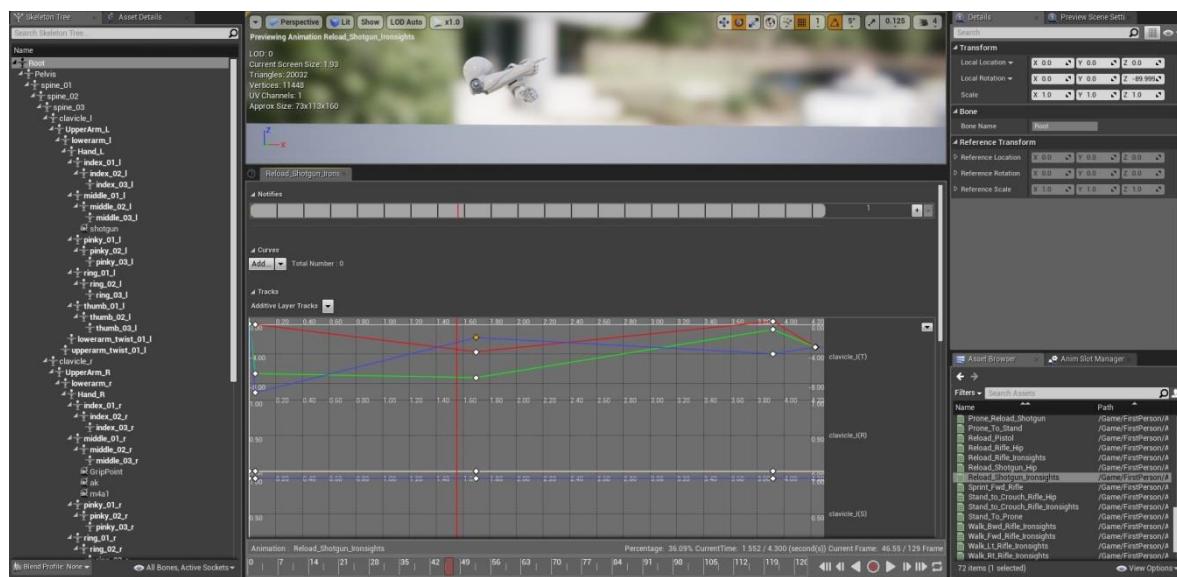


Figura 77: Corrigiendo animación desde el editor de Unreal

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

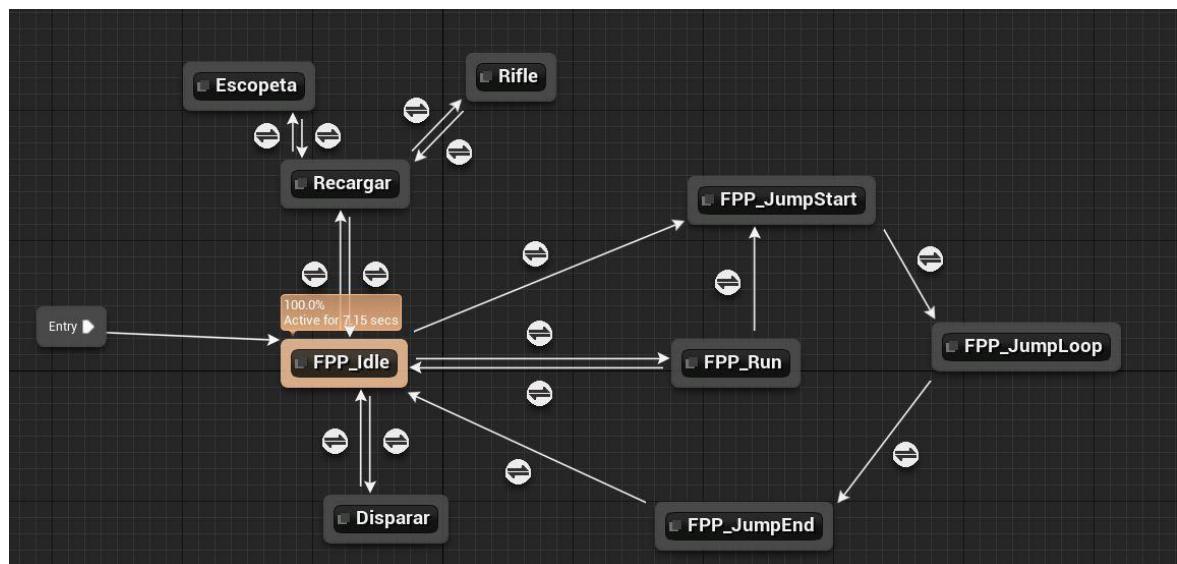


Figura 78: Animation Blueprint personaje

La imagen superior muestra la máquina de estado encargada de elegir una animación y su funcionamiento es el siguiente:

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

Si disparamos se reproducirá la animación de disparar que simula el retroceso del arma. Como se puede observar en el gráfico, 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 luego disparamos.

Cuando recargamos constatamos si el tipo de arma que tenemos es la escopeta o un rifle, puesto que 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 pasamos 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, momento en que se reproducirá la animación **FPP_JumpEnd** y después de esta volveremos al estado normal (**FPP_Idle**).

5.3.2.2 Armas

La lógica de las armas del juego constituye un elemento esencial en un videojuego de este tipo.

5.3.2.2.1 WeaponDrop Blueprint (armas en la pared)

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

5.3.2.2.1.1 Componentes

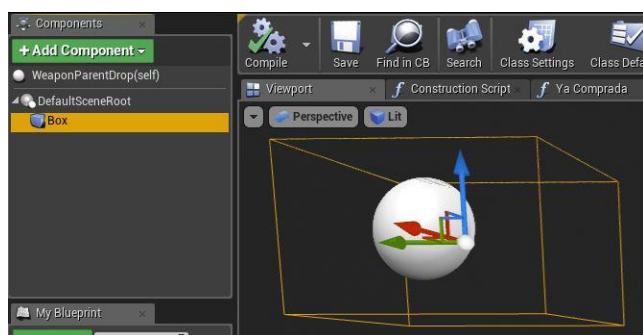


Figura 79: Componentes WeaponParentDrop Blueprint

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

5.3.2.2.1.2 Variables

WeaponParentDrop Variables		
Nombre	Tipo	Descripción
WeaponID	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 AKDrop .
Comprada	Boolean	Variable <i>booleana</i> resultado de la función YaComprada . Si es <i>true</i> es que ya hemos comprado el arma y si es falso es que aún no hemos comprado esa arma con la que estamos interactuando (hemos entrado en la zona de compra).
precioArma	Int	Es el precio que tenemos que pagar para comprar esa arma.
precioMunición	Int	Es el precio que tenemos que pagar para comprar munición de esa arma.
Info	<i>String</i>	Es un <i>string</i> con información sobre el arma, que se muestra luego por pantalla cuando estamos en zona de compra. Esta información comprende el nombre y el precio del arma.
InfoAmmo	<i>String</i>	Es un <i>string</i> con información sobre la munición del arma con la que interactuamos y su precio.

Figura 80: Table variables WeaponParentDrop

5.3.2.2.1.3 Funciones

Como el número de funciones de esta clase no es muy elevado, no es necesaria una tabla, 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**, se encargan de inicializar los valores del arma (para comprar), como se muestra a continuación.

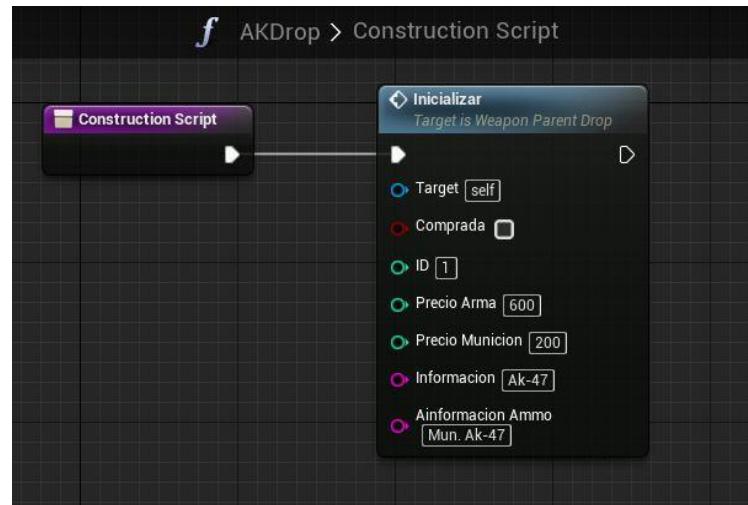


Figura 81: Constructor AKDrop

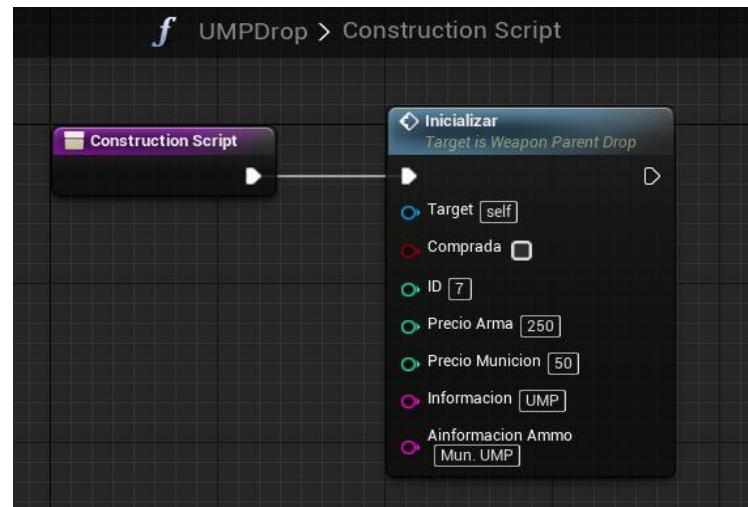


Figura 82: Constructor UMPDrop

5.3.2.2.1.3.2 Inicializar

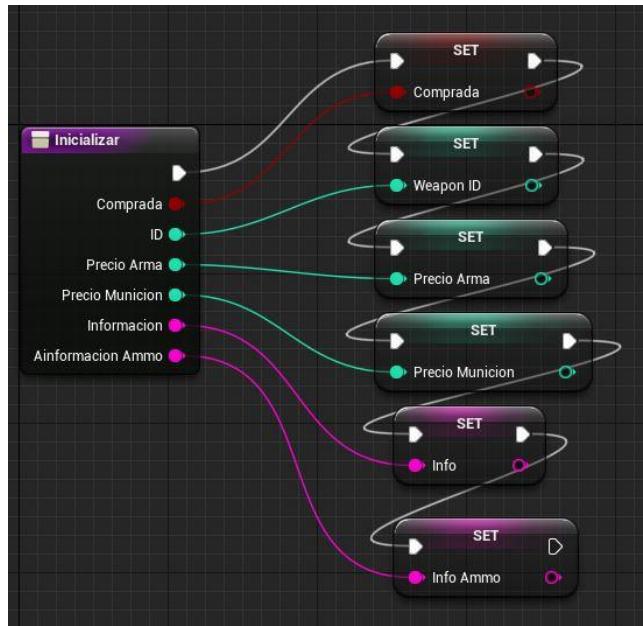


Figura 83: Función inicializar WeaponParentDrop

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

5.3.2.2.1.3.3 Entrar en zona de compra

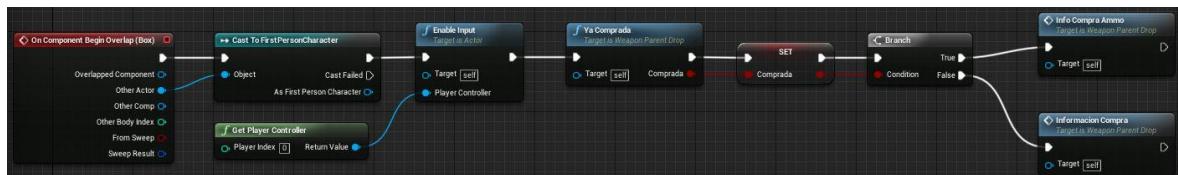


Figura 84: WeaponParentDrop evento entrar en zona de compra

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

A continuación comprueba 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 85: Mostrando información compra munición arma

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



Figura 86: Mostrando información compra arma

5.3.2.2.1.3.4 Salir de zona de compra

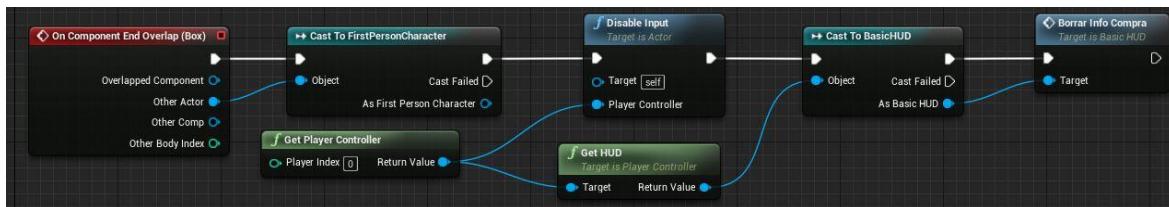


Figura 87: WeaponParentDrop evento salir de zona de compra

Cuando el jugador sale de la zona de compra o, lo que es lo mismo, sale de la **BoxCollision** del **WeaponDrop**, se llama a este evento que deshabilita el *input* en esta clase para que, aunque se pulse la F (es el botón con el que se compra), no se llame al evento **InputAction Interactuar** y así se impide la compra del arma si se está fuera de la zona. Por último, llama a una función del HUD denominada **BorrarInfoCompra** que cancela la información de compra de la pantalla.

5.3.2.2.1.3.5 Interactuar

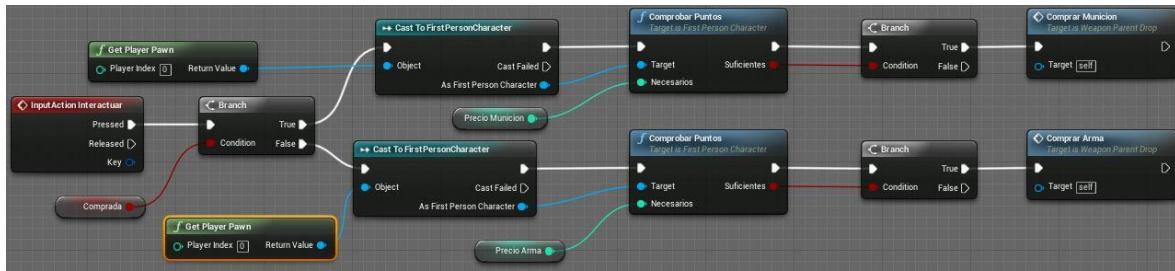


Figura 88: InputAction Interactuar WeaponDrop

Este evento se llama cuando pulsamos la tecla vinculada al evento **Interactuar** (la tecla F del teclado o el botón círculo de un mando de PS4), aunque solo se llamará si estamos dentro de la zona de compra del arma; en el caso contrario, aun pulsando la tecla de interactuar, no sucederá nada.

Lo primero que hace este evento es constatar que el arma con la que estamos interactuando ya está comprada y solo necesita munición o si tenemos que comprarla.

- 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. Con los puntos necesarios entonces podemos comprar la munición; para ello llamamos a la función **ComprarMunición**.
- Si el arma no está comprada y tenemos el dinero necesario, se llama a la función **ComprarArma**.

5.3.2.2.1.3.6 Comprar arma

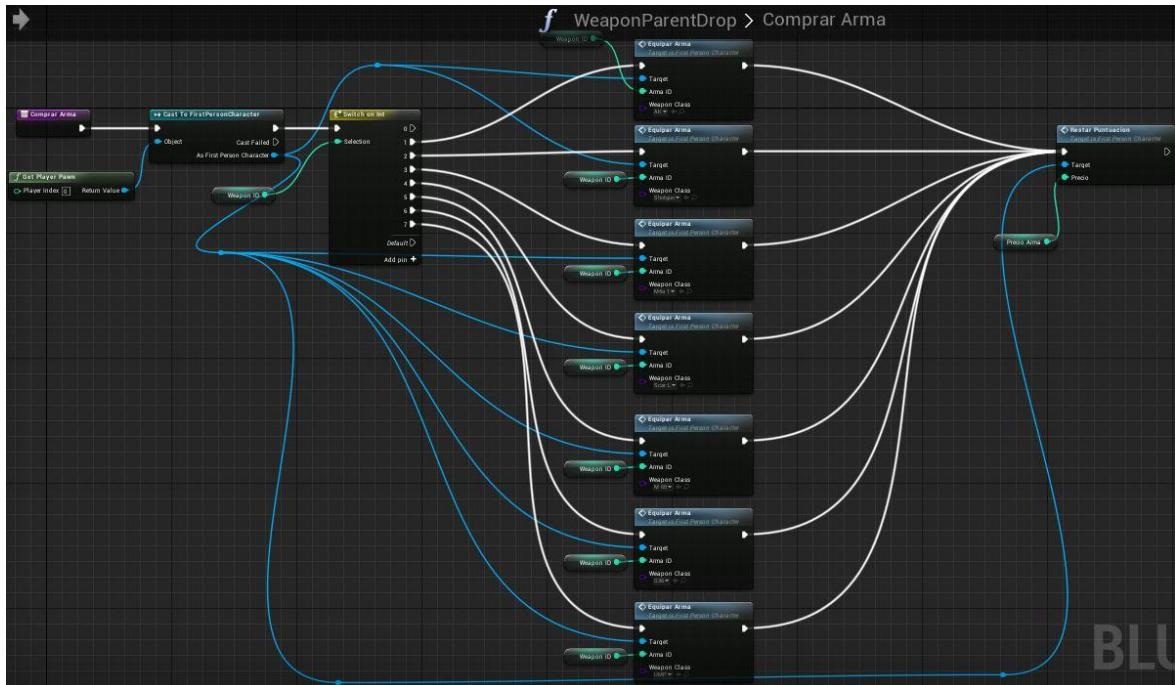


Figura 89: Función comprar arma

Aunque a primera vista parezca compleja, esta función es bastante sencilla, tal como se explica a continuación.

Para comprar un arma, en primer lugar llamamos a la función **EquiparArma** del *FirstPersonCharacter*. Esta función 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 pasarle la clase del arma correcta, antes de llamar a la función **EquiparArma**, se hace 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 a **EquiparArma**, se llama a la función **RestarPuntuación** del *player* que le proporciona el precio del arma que acabamos de comprar.

5.3.2.2.1.3.7 Comprar munición

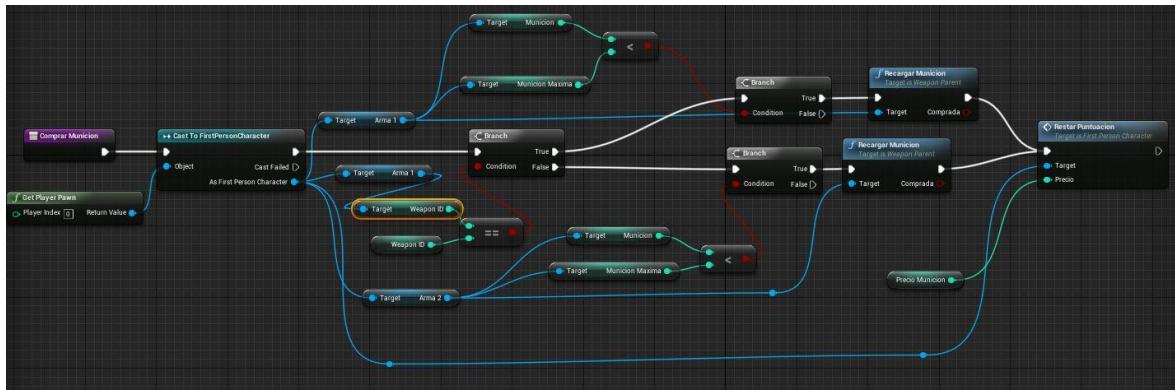


Figura 90: Función comprar munición

Antes de comprar munición, esta función comprueba para qué arma es, puesto que cuando te acercas a comprar munición no es obligatorio tener el arma equipada. Entonces lo primero es conocer la referencia al arma en Arma1 o en Arma2 para sumar la munición al arma correcta. Esto se comprueba con los ID, es decir, si el ID del Arma1 coincide con el ID del arma que se está recargando se llama a la función **recargarMunición** del Arma1; si esos ID no coinciden, se llama a la función de **recargarMunición** del Arma2, y cuando estamos en esta función sabemos seguro que tenemos el arma con la que estamos interactuando en una de las dos variables.

Una vez que conocemos de qué arma se trata, comprobamos la munición de esta arma, pues si la munición está al máximo no se podrá comprar más munición. Si la munición no está al máximo, se llama a la función **recargarMunición** y seguidamente a la función **RestarPuntuación** para pasarle el precio de la munición que acabamos de comprar.

5.3.2.2.1.3.8 Comprobar si ya hemos comprado el arma

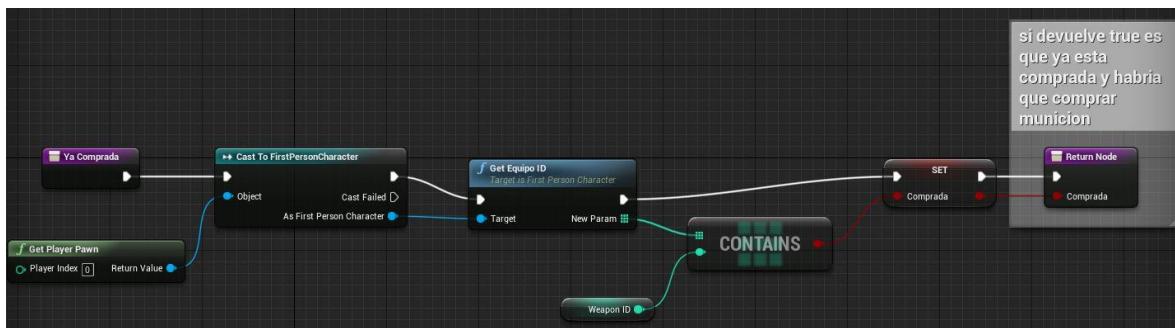


Figura 91: Función que comprueba si ya hemos comprado un arma

Esta función devuelve un booleano que aclarará si el arma ya ha sido comprada o aún no. Para ello llama a la función **getEquipoid** del *player*, que devuelve un *array* con los ID de las armas que se tienen en el equipo, entonces se comprueba si el ID del arma con la que se interactúa aparece en este

array. Si es así, el arma ya ha sido comprada y, puesto que figura en el equipo, se pone la variable **comprada** a verdadero y se devuelve por la función. Si el ID del arma no está en el *array* del equipo del *player*, significará que no se ha comprado aún esa arma y por tanto se establece la variable **comprada** a falso y se devuelve falso como resultado de la función.

5.3.2.2.1.3.9 Información compra arma y munición

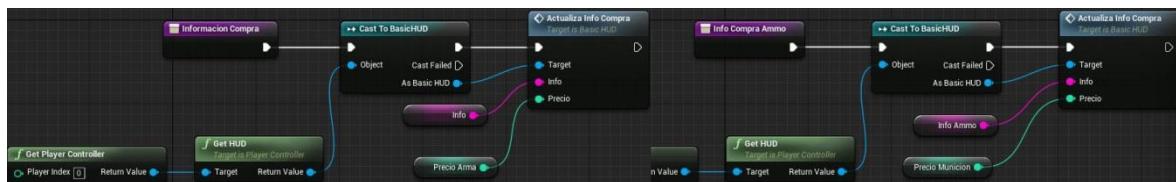


Figura 92: Función información compra (izquierda) y función información compra munición (derecha)

Estas dos funciones las vamos a explicar juntas debido a 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 se haya comprado el arma.

5.3.2.2.2 Weapon Blueprint (arma equipada)

El *Blueprint* de la clase arma llamado **WeaponParent** es el *Blueprint* del que heredan todas las armas que se pueden 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 según el arma. Cuando haya que explicar el comportamiento de una clase hija se escogerá como ejemplo la clase **AK** para ilustrar estas funciones.

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** que se llama mediante un mensaje a través de la interfaz desde la clase **WeaponParent**, así se ejecuta la lógica del **Shoot** del arma que reciba el mensaje, que será aquella con la que se ha disparado.

5.3.2.2.2 Variables

WeaponParent Variables		
Nombre	Tipo	Descripción
WeaponID	Int	ID del arma, que nos sirve para identificar la clase del arma.
MunicionCargador	Int	Es la munición actual que hay en el cargador del arma.
CapacidadCargador	Int	Cantidad de balas que tiene un cargador, así a la hora de recargar cada arma se hace en función de la capacidad del cargador, por ejemplo la m60 tendría 100 balas en el cargador mientras que la UMP tendría 30.
MunicionMáxima	Int	Es la munición máxima que se puede tener (fuera del cargador).
Munición	Int	Es la munición actual de la que disponemos para recargar el arma sin contar la munición que ya hay dentro del cargador.
Recargando	Boolean	Variable booleana que nos dice si estamos recargando el arma o no.
Disparando	Boolean	Variable booleana que nos dice si estamos disparando o no.
Automática	Boolean	Verdadera en caso de que el arma sea automática y falsa cuando no lo sea.
CanShoot	Boolean	Esta variable nos dice si estamos preparados para realizar un disparo. Si es verdadera podremos disparar, mientras que si es falsa no podremos.
Damage	Float	El daño que causa un impacto del arma.
Cadencia	Float	Es el tiempo que transcurre entre disparo y disparo.
TiempoRecarga	Float	El tiempo que tarda un arma en recargarse.
NumMejorasCadencias	Int	Indica el número de veces que se ha mejorado la cadencia del arma (en una máquina de mejorar cadencia).

NumMejorasRecarga	Int	Indica el número de veces que se ha mejorado el tiempo de recarga del arma (en una máquina de mejorar tiempo de recarga).
NumMejorasCargador	Int	Indica el número de veces que se ha mejorado la capacidad del cargador del arma (en una máquina de mejorar capacidad del cargador).
NumMejorasMunición	Int	Indica el número de veces que se ha mejorado la capacidad de la munición máxima del arma (en una máquina de mejorar munición máxima).
NumMejorasDamage	Int	Indica el número de veces que se ha mejorado el daño del arma (en una máquina de mejorar daño).
CentrarBala	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.
CentrarBalaApuntando	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 93: 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 se pueden utilizar en el juego, la importancia de inicializar la clase recae sobre el constructor de las clases hijas que conoce los atributos específicos de la clase de arma que está inicializando.

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

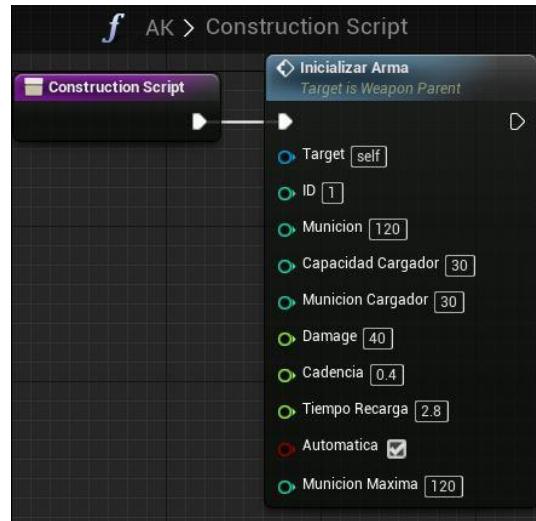


Figura 94: Constructor AK

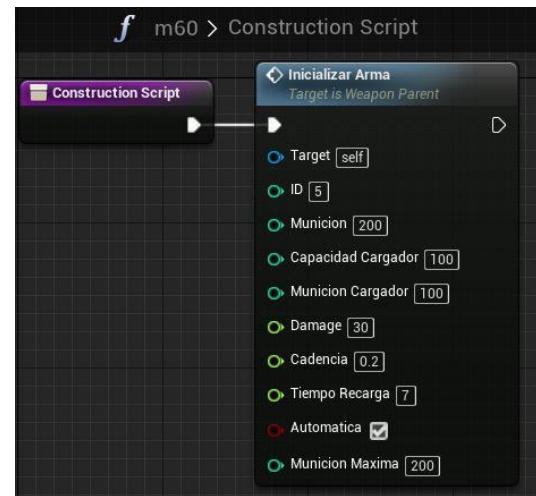


Figura 95: Constructor M60

5.3.2.2.2.3.2 Inicializar

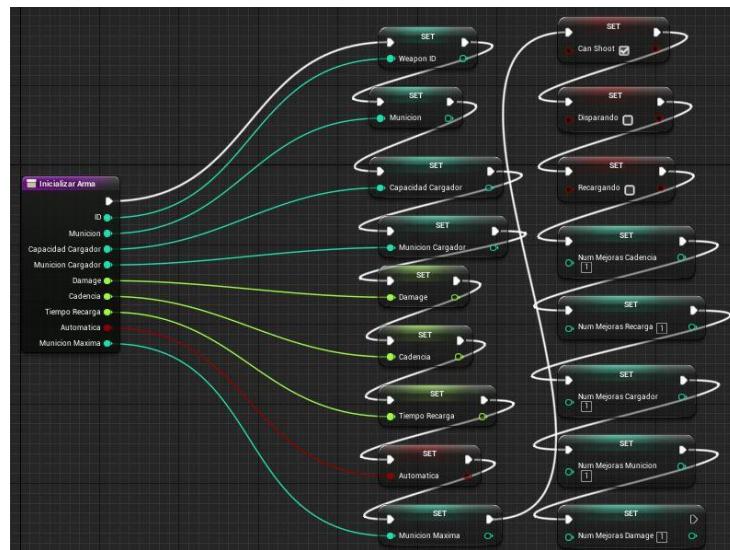


Figura 96: Función inicializar arma

Esta es la función a la que llaman los constructores de las clases hijas. Se encarga de establecer las estadísticas de cada arma y supone la ventaja de que ahora se tratan todas las armas como **WeaponParent** sin importar el tipo de arma sea. De esta forma lo que guardamos en las variables **Arma1**, **Arma2** y **Equipada** del *player* son **WeaponParent** que permite cambiar de armas, recargarlas, disparar, etc. sin necesidad de conocer la clase de arma pues, tanto si se trata de una M60 como de una UMP, se llama al método disparar de la clase padre.

5.3.2.2.2.3.3 Disparar

Esta función se divide a su vez en varias, además de que una parte común se hace en la clase padre **WeaponParent** y luego otra parte específica se hace en cada arma.

5.3.2.2.2.3.3.1 InputAction Fire

El evento **InputAction Fire** se ejecuta cada vez que se pulsa el botón de disparar.

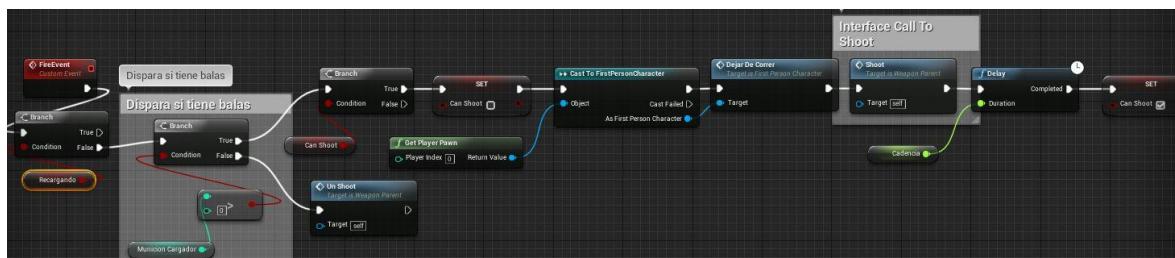


Figura 97: Input Action Fire (evento disparar)

Primero, comprobamos si se está recargando, ya que si es así no es posible disparar. Si no estamos recargando, entonces nos aseguramos de que la munición del cargador es mayor que 0, es decir, si quedan balas en el cargador con las que disparar. Si no quedan, se llama al evento **UnShoot**. Si se dispone de balas en el cargador comprobamos ahora que se puede disparar, esta condición viene dada por la variable **CanShoot**, y en ese caso cambiamos esta variable a falso dado que se está en mitad de un disparo y se hará true cuando pase el tiempo necesario entre disparo y disparo.

Sabiendo que es posible disparar, llamamos a la función **DejarDeCorrer** puesto que no es posible disparar mientras se corre. Seguidamente llamamos al evento **Shoot** del arma 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.

A continuación, 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 llamemos a este evento ***Fire*** se pueda disparar. Hay que recordar que si el arma es automática a este evento se está llamando continuamente mientras se pulsa el botón de disparar, por eso es necesaria esta variable ***CanShoot***, para que aunque entremos 10 veces en este evento se dispare 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**.

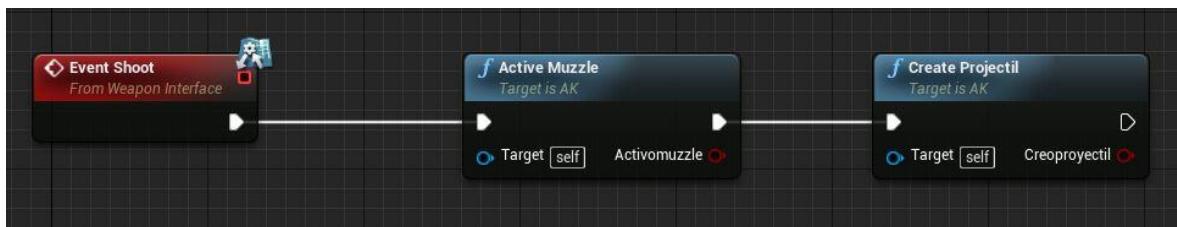


Figura 98: Event Shoot de la clase AK

Únicamente llama a dos funciones cuya lógica es diferente en cada arma: una es la de ***ActiveMuzzle***, encargada de activar el destello del arma cuando corresponda, y la otra es la función ***CreateProjectil***, que contiene toda la lógica del disparo y que veremos en detalle en el apartado siguiente.

5.3.2.2.3.3.3 Create Projectil

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

Debido a la extensión de esta función, la he dividido en dos imágenes. Se ha dejado la función ***getRaycastPos*** en ambas imágenes para facilitar la conexión entre ambas, de modo que el final de la primera foto continúa al inicio de la segunda.

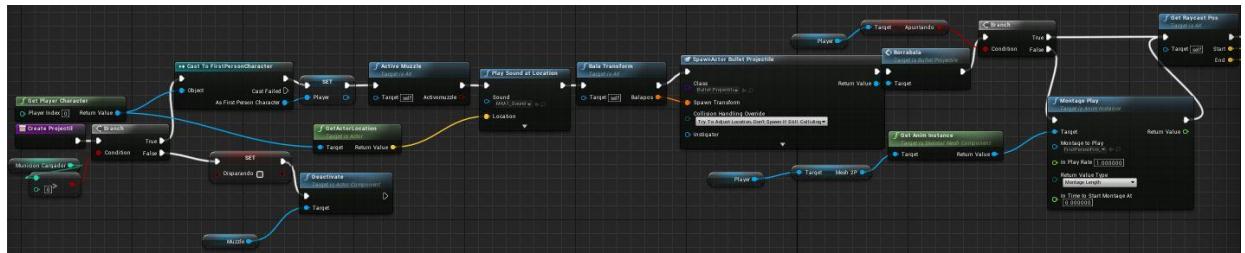


Figura 99: Function CreateProjectil (parte 1/2)

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*. Luego, 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, crearemos 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 encargará 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 porque impedía apuntar debido a que tenías el arma muy cerca de la cámara.

Tanto si hemos reproducido la animación de retroceso como si no 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.

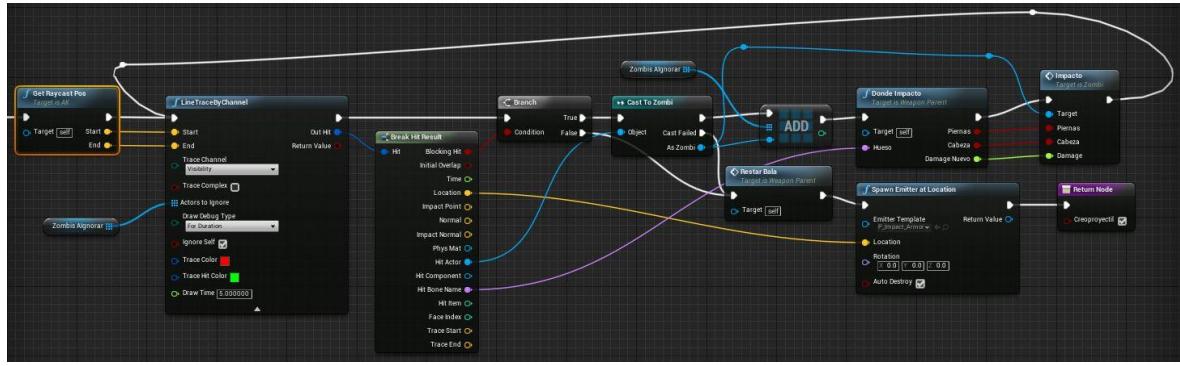


Figura 100: Function CreateProjectil (parte 2/2)

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 dos veces al mismo zombi con el mismo disparo (porque se añade ese zombi a la lista de zombis a ignorar). Esta lista deberá estar vacía cada vez que llamamos a esta función para que la bala pueda volver a impactar sobre cualquier zombi por eso se ha elegido una variable local. 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 en el primer zombi sino que impactan en 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 con el fin de 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:

- **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 impacta el *raycast*.
- **Hit Bone Name:** es el nombre del hueso con el que impacta el *raycast*; nos servirá para saber en qué parte del cuerpo le hemos dado.

Una vez que hemos lanzado el *raycast* y tenemos los resultados que vienen dados en el nodo *Break Hit Result* hay tres 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 llamará 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 volveremos a lanzar un *raycast* puesto que hemos dado a un zombi y puede ser que haya más detrás de él.

Como vemos la bala atraviesa zombis incluso puede 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 que devuelve la transformación que se usará para crear una bala que es un actor de la clase *Bullet Projectil*.

Antes de nada conviene aclarar el componente que se usa en esta función llamado **Sphere**. Se trata de una esfera que se encuentra en el cañón del arma para poder saber en todo momento desde donde se tiene que disparar la bala.

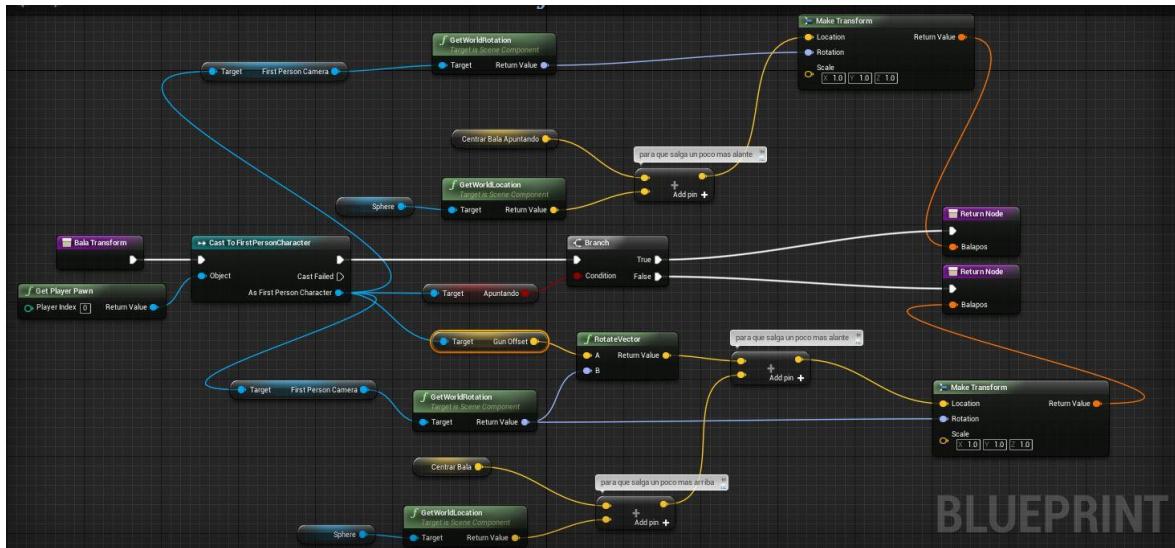


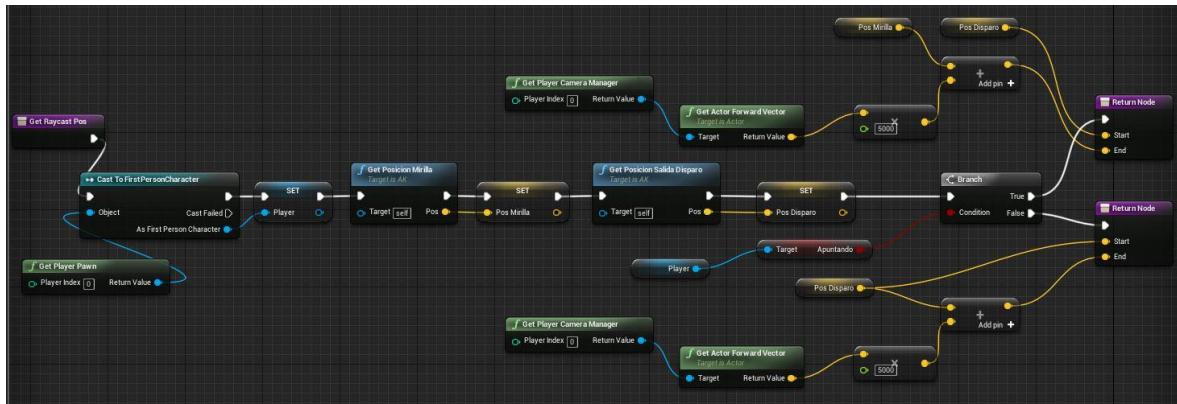
Figura 101: Función BalaTransform

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 de la cámara y la posición es la 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 no está apuntando, entonces obtenemos como rotación la misma que en el supuesto anterior pero la posición ahora cambiará 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 último devolvemos esta transformación.

5.3.2.2.3.3.3.2 Función GetRaycastPos

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

Figura 102: 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 con 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`).

5.3.2.2.2.3.3.3.3 Funcion DondelImpacto

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

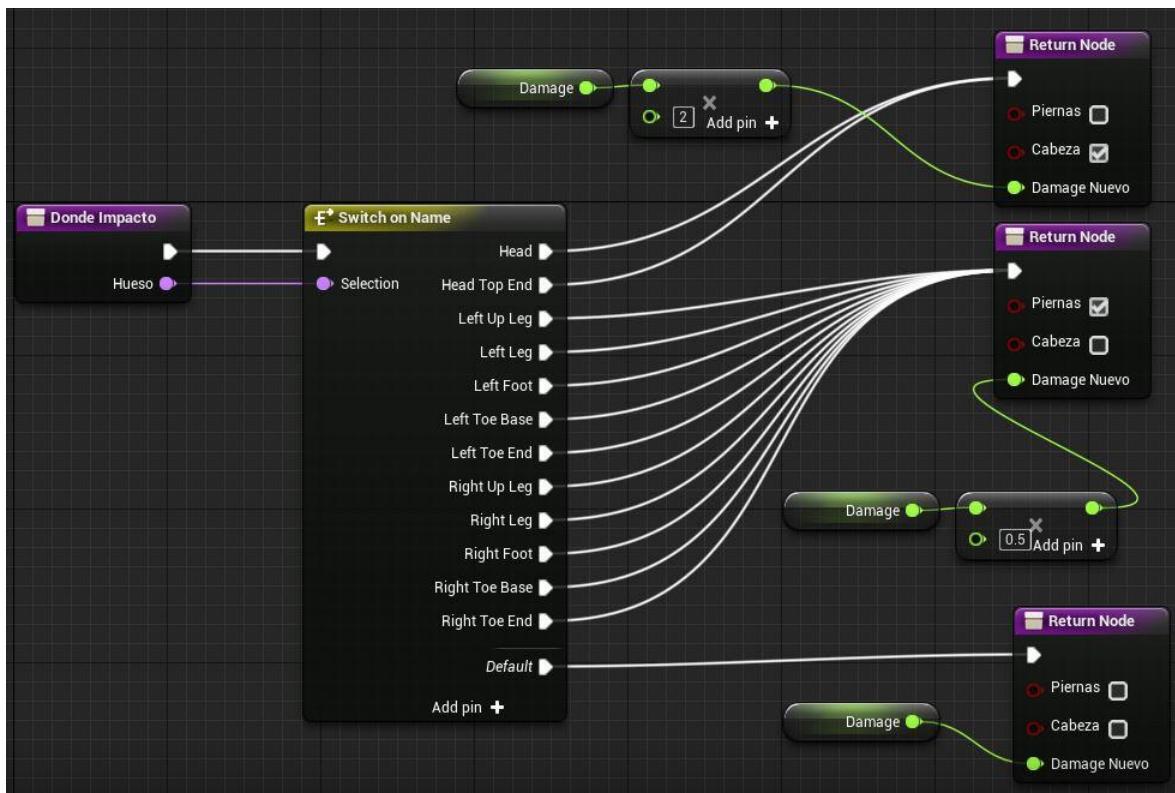


Figura 103: Función DondeImpacto

Esta función recibe por parámetro el nombre del hueso sobre el que impacta la bala, que viene dado por el nodo ***Break Hit Result*** de la función ***createProjectil***. Con este nombre se hace un *Switch* con tres posibles resultados:

- El hueso pertenece a la cabeza, por lo tanto devolvemos tres 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

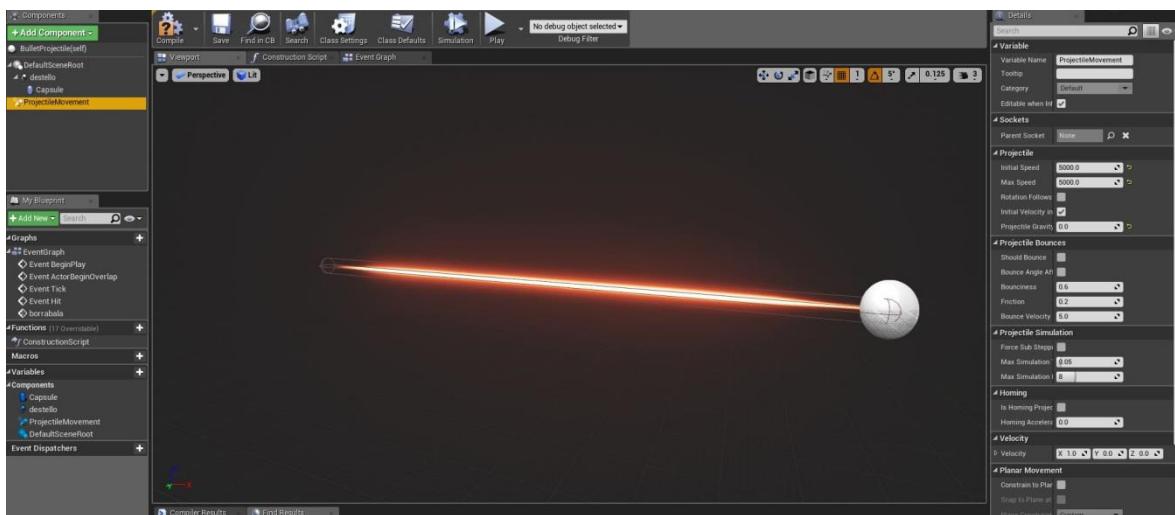


Figura 104: Bullet Projectil Actor

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

5.3.2.2.2.3.4 Dejar de disparar (UnFireEvent)

Este evento **UnFireEvent** se llama en la clase padre cuando se deja de pulsar la tecla de disparar. Únicamente se dedica a llamar al evento **UnShoot** a través de la interfaz **WeaponInterface** para que cada clase hija haga la lógica correspondiente para dejar de disparar. Esta lógica suele ser desactivar el *muzzle* (destello del arma) y poner la variable **disparando** a falsa.

5.3.2.2.2.3.5 Recargar

Aunque en un principio se puede pensar que la mecánica de recargar es sencilla y que simplemente es añadir balas al cargador, la función es bastante extensa. Para facilitar su legibilidad se ha dividido en varias imágenes.

Cuando el jugador pulsa la tecla asignada a la recarga, se ejecuta el evento **InputAction Recargar**, que comprueba si tenemos un arma equipada y, si la tenemos, llama a la función **Recargar** de la clase padre (la recarga es común independientemente de que se recargue, por eso la lógica se encuentra en la clase padre).

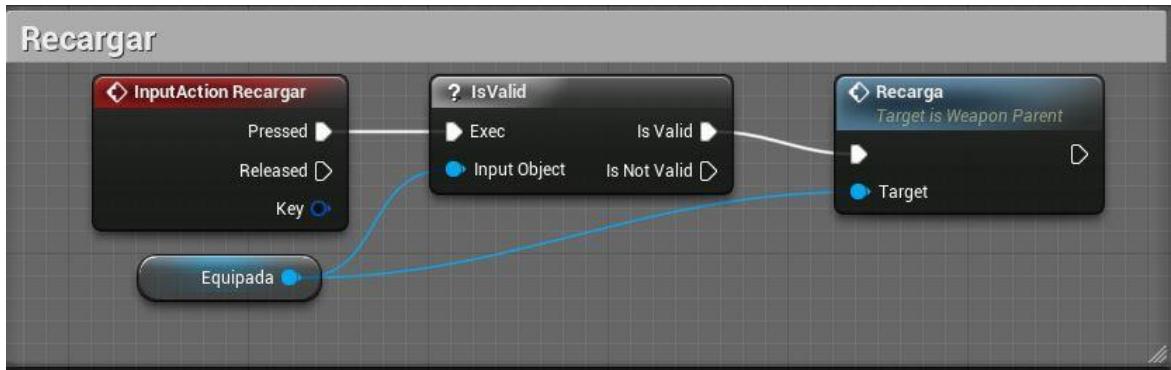


Figura 105: InputAction Recargar desde la clase FirstPersonCharacter

Una vez hemos visto cómo se llama a la función **Recargar** del arma que estamos utilizando en ese momento desde el **FirstPersonCharacter**, vamos a ver ahora la lógica de esta función.

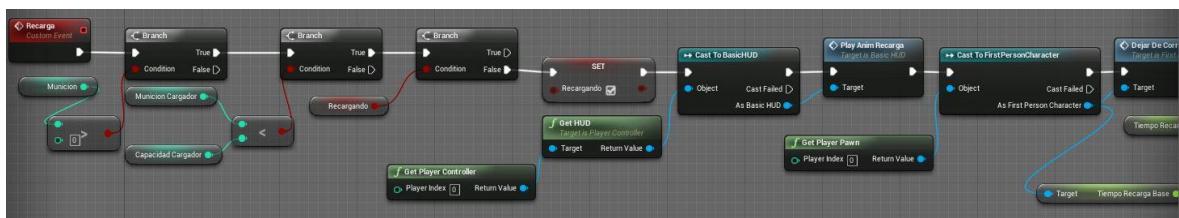


Figura 106: Función recargar (parte 1/4)

Esta función comprueba que nuestra munición es mayor que 0, puesto que si no nos queda munición fuera del cargador no podremos recargar. Recordamos que la variable **munición** almacenaba la cantidad de munición que teníamos fuera del cargador disponible para la recarga.

Si tenemos munición, entonces comprobamos que la munición del cargador (el número de balas que tenemos actualmente en el cargador) sea menor a la capacidad del cargador, ya que se puede dar el caso de que le demos al botón de recargar cuando tengamos el cargador completo. De esta forma solo recargaríamos si el cargador no está completo.

El siguiente paso será asegurarnos de que no estamos recargando, ya que puede ser que le demos varias veces a la tecla de recarga, o que incluso mientras recarga sigamos pulsando la tecla, así si ya está recargando pulsar la tecla de nuevo no tendría ningún efecto.

Una vez hemos hecho estas comprobaciones, llamamos a la función **PlayAnimRecarga** del **HUD**, que se encargará de dibujar el texto “recargando” en la pantalla con una animación sobre este, y llamamos a la función **DejarDeCorrer** del personaje (que hemos obtenido mediante un *casteo* con el nodo **CastToFirstPersonCharacter**).

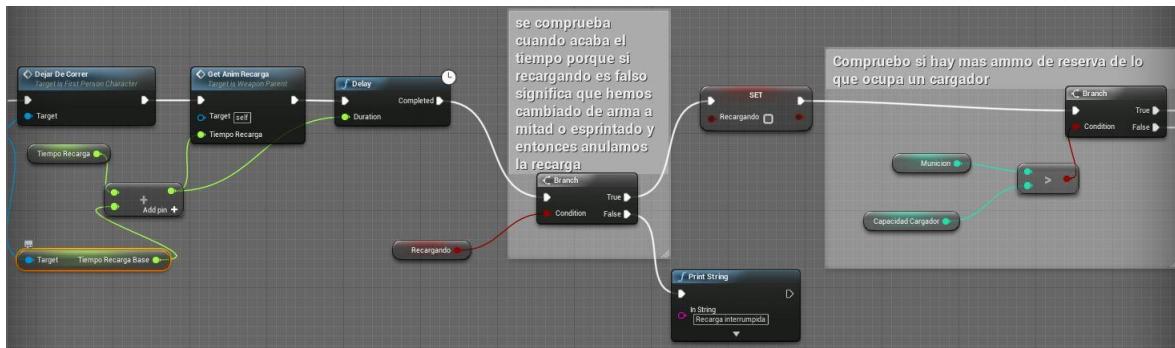


Figura 107: Función recargar (parte 2/4)

Después de llamar a **DejarDeCorrer** (final de la parte 1), llamamos a la función **getAnimRecarga** y le pasamos como parámetro el tiempo base de recarga del jugador más el tiempo de recarga del arma actual, **este resultado será el tiempo total de la recarga**. Esta función, a partir del ID del arma que tengamos, reproducirá la animación correcta y lo hará de manera que la duración de la animación sea la misma que la duración de la recarga, como veremos en el siguiente apartado.

Utilizamos ahora el nodo **Delay** que nos permite pausar la ejecución de este evento el tiempo que le indiquemos en su parámetro de entrada “*Duration*”. El valor que le pasamos será, por tanto, el tiempo total que tardará la recarga. Este **Delay** es lo que necesitamos puesto que no queremos que se recargue el arma hasta que no haya terminado todo el tiempo de la recarga.

Una vez salimos del nodo **Delay** (cuando ha pasado el tiempo indicado), volvemos a comprobar si la variable **recargando** sigue siendo cierta ya que, si mientras estamos recargando corremos o cambiamos de arma, esta variable se hará falsa, por tanto anulará la recarga cuando llegamos a este punto. De esta manera, si hemos anulado la recarga con alguna acción, no recargaremos el arma y es como si no hubiéramos llamado a la función **Recargar**.

Si la variable **recargando** continúa siendo cierta en esta comprobación, significa que no hemos anulado la recarga, por tanto se ha completado la recarga, así que ahora volvemos a poner **recargando** a falso para futuras veces que vayamos a recargar y seguimos la ejecución.

Ahora hemos de hacer las verificaciones necesarias para conocer la cantidad munición que tenemos, si podemos recargar el cargador entero, solo una parte del cargador, si nos sobra munición fuera del cargador después de recargar, etc. Estas comprobaciones se llevan a cabo en las partes 3 y 4 de esta función. Ahora bien, dado que algunas comprobaciones te llevan a la parte 3 y otras a la parte 4, he optado por colocar las dos imágenes seguidas porque es necesario explicarlas conjuntamente.

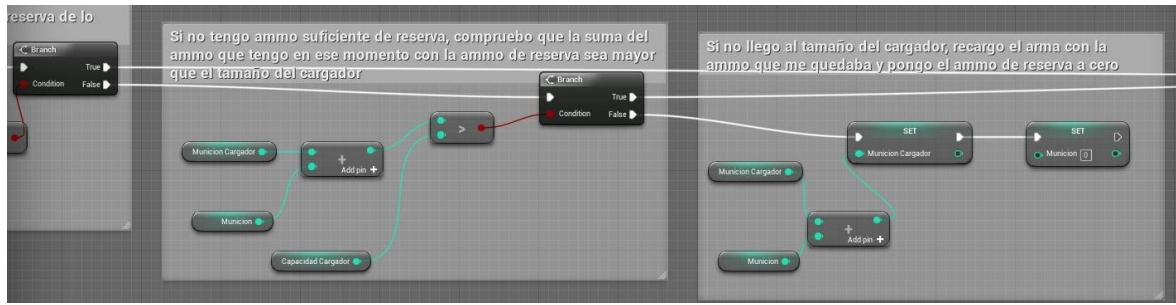


Figura 108: Función recargar (parte 3/4)

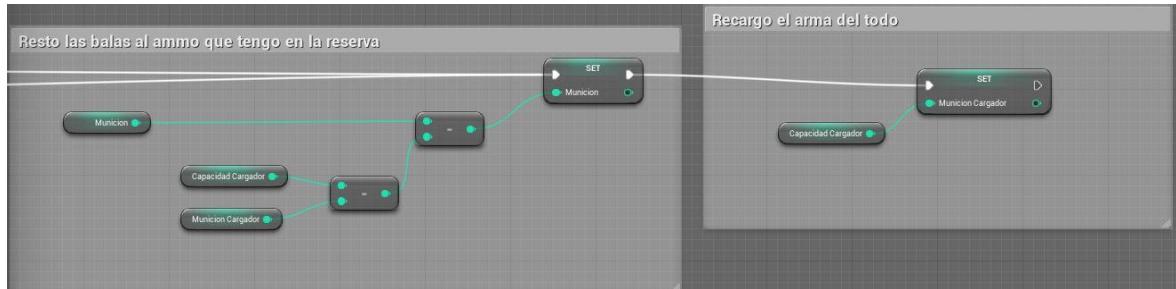


Figura 109: Función recargar (parte 4/4)

Para ello empezamos revisando si la munición (disponible para recarga) es mayor que la capacidad del cargador (**comprobación final de la parte 2**). En este punto se abren dos posibilidades (son las que se ven en el comienzo de la parte 3, el *true* y el *false*):

- ***True - La munición es mayor que la capacidad del cargador (lógica en la parte 4):*** En este caso nos sobrará munición disponible, entonces tenemos que ver cuánta munición nos queda y cuánta hemos usado para recargar el arma. La munición utilizada para recargar el arma será la resta de la capacidad del cargador menos la munición del cargador (**CapacidadCargador – MunicionCargador**). De esta forma, si la capacidad del cargador es de 30 pero teníamos 7 balas ya en el cargador, la munición utilizada para la recarga será de 23 balas. Una vez que conocemos la munición empleada, restamos a nuestra munición disponible la munición utilizada; el resultado será la nueva munición disponible. Por ejemplo, si teníamos 45 balas de munición disponible y la munición utilizada era 23, la nueva munición disponible será de 22. Finalmente, como sabemos que la munición disponible excede la capacidad del cargador, la recarga del cargador será completa, por lo tanto ponemos la munición del cargador al máximo utilizando para ello la capacidad del cargador.
- ***False - La munición es menor o igual a la capacidad del cargador (lógica en el principio de la parte 3):*** Aquí entramos cuando, pongamos por caso, nuestra capacidad de cargador es de 30 balas y nuestra munición es de 25 balas. Cuando nos encontramos en esta situación, se podría pensar que simplemente pasamos las 25 balas al cargador y ya se ha acabado, pero no es tan sencillo, pues esto solo sería correcto si tengo de 0 a 5 balas en el cargador; en

cambio, si recargo cuando tengo por ejemplo 20 balas en el cargador, mi munición (fuera del cargador) sigue siendo de 25 balas (que es menor que la capacidad del cargador) dado que, si pasamos todas nuestras balas al cargador, tendríamos 45 balas en el cargador y esto excede su capacidad máxima de 30 balas. Por todo lo anterior, llegados a este punto tenemos que comprobar si la munición que ya había en el cargador (**MunicionCargador**) sumada a la munición disponible para recarga (variable **munición**) es mayor que la capacidad del cargador. En esta situación se pueden dar dos posibles resultados: que sea mayor y, por lo tanto, se cumple la condición o que no lo sea y que no se cumpla la condición:

- **Se cumple la condición MunicionCargador + Munición > CapacidadCargador (lógica en la parte 4):** Es la misma lógica que la del apartado “*True - La munición es mayor que la capacidad del cargador*”. La munición necesaria para la recarga es menor a la munición disponible, por tanto recargamos el cargador completo, miramos el número de balas utilizadas para esta recarga y se las restamos a la munición disponible.
- **No se cumple la condición MunicionCargador + Munición > CapacidadCargador (lógica en el final de la parte 3):** Cuando esto ocurre, significa que no tengo munición suficiente para completar la recarga entera o bien la munición es la justa para completar la recarga completa. En este supuesto, la munición actual del cargador sería la munición que ya tenía cuando inicié la recarga más toda la munición que tenía disponible (acabamos de comprobar que esta suma no excede la capacidad del cargador). Después ponemos la munición disponible (variable **muncion**) a 0 puesto que la hemos usado toda para recargar el arma.

5.3.2.2.3.5.1 Función GetAnimRecarga

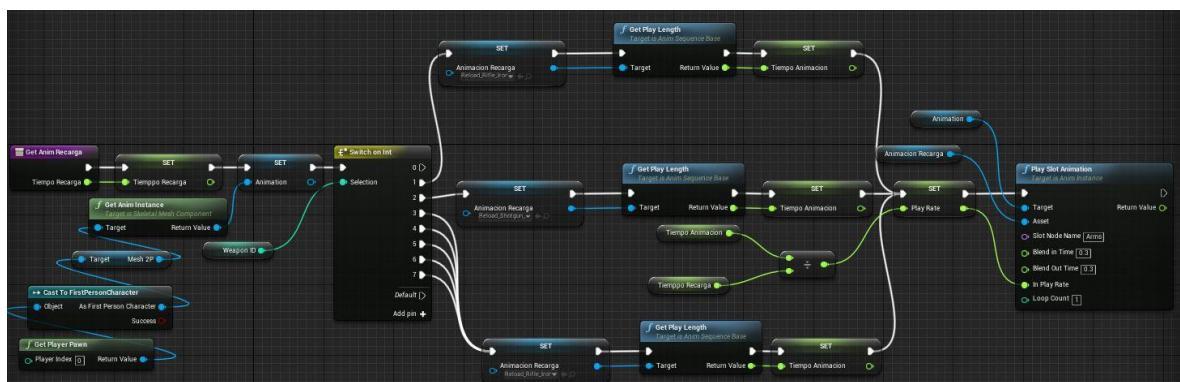


Figura 110: Función getAnimRecarga

Nos guardamos en la variable **TiempoRecarga** el parámetro de entrada que recibimos, que corresponde con el tiempo total que durará la recarga. También nos guardamos en la variable **Animation** una instancia de la animación de la malla **Mesh2P**, que son los brazos del personaje y hacemos un *switch* mediante el nodo **SwitchonInt** mediante el ID del arma y, dependiendo del arma, usaremos una animación de recarga u otra. Nos guardamos la animación que tenemos que usar, obtenemos la duración de esta animación (**getPlayLength**) y nos la guardamos en la variable **tiempoAnimación**.

Una vez que ya disponemos del tiempo de la animación (variable **tiempoAnimacion**) y del tiempo de la recarga (variable **TiempoRecarga**), podemos conocer la velocidad a la que tiene que reproducirse la animación. Esta velocidad es el resultado de dividir **TiempoAnimacion** entre **TiempoRecarga** y nos guardamos este resultado en la variable **PlayRate**.

Finalmente, llamamos al nodo **PlaySlotAnimation** al que le pasamos los siguientes parámetros de entrada:

- **Target:** es la instancia de animación sobre la cual queremos reproducir una animación, por lo tanto la de nuestros brazos del jugador que nos guardamos al principio de la función en la variable **Animation**.
- **Asset:** es el recurso que vamos a utilizar, es decir, la animación que queremos reproducir que se encuentra almacenada en nuestra variable **AnimacionRecarga**.
- **SlotNodeName:** es el espacio sobre el cual vamos a reproducir la animación en el esqueleto, ya que puede tener varios: por ejemplo, en un esqueleto entero puede ser que quieras reproducir una animación únicamente en las piernas.
- **Blend in Time:** es el tiempo de fusión entre la animación que estábamos reproduciendo y nuestra animación de recarga.
- **Blend out Time:** es el tiempo de fusión entre la animación de recarga y la siguiente animación que vayamos a reproducir.
- **In Play Rate:** es la velocidad con la que se reproduce la animación, de manera que un *PlayRate* igual a 1 es velocidad normal, mientras que *PlayRate*=2 sería el doble de velocidad y *PlayRate*=0.5 sería la mitad de velocidad. Le pasamos la velocidad que ya habíamos calculado previamente almacenada en nuestra variable **PlayRate**.
- **Loop Count:** la cantidad de veces que tiene que repetir la animación, en nuestro caso solo se reproducirá una vez.

5.3.2.2.3.6 Mejorar Arma

En este apartado vamos a analizar las diferentes lógicas a la hora de mejorar alguna estadística de un arma. Estas funciones son comunes a todas las armas por la razón de que la lógica es la misma pero cada una utiliza sus estadísticas, por lo tanto estas funciones pertenecen a la clase padre **WeaponParent**.

Estas funciones se llaman cuando interactuamos con la máquina que mejora las armas. Cada máquina llamará a la función de su tipo, así la máquina que mejora el daño llamará a la función **Mejorar Daño**.

5.3.2.2.3.6.1 Mejorar Daño

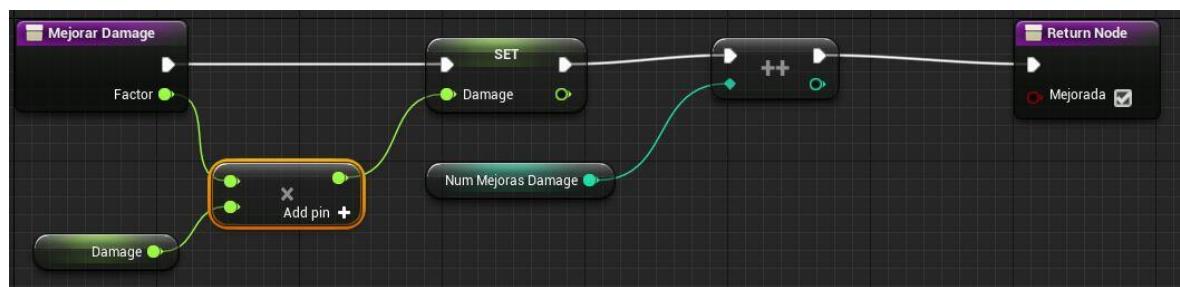


Figura 111: Función mejorar daño

Esta función recibe por parámetro el factor de mejora de daño y lo multiplica por el daño actual del arma. Se establece este resultado como el nuevo daño del arma. Por último, aumentamos en 1 el número de mejoras de daño que se ha hecho para esa arma.

5.3.2.2.3.6.2 Mejorar Munición Máxima

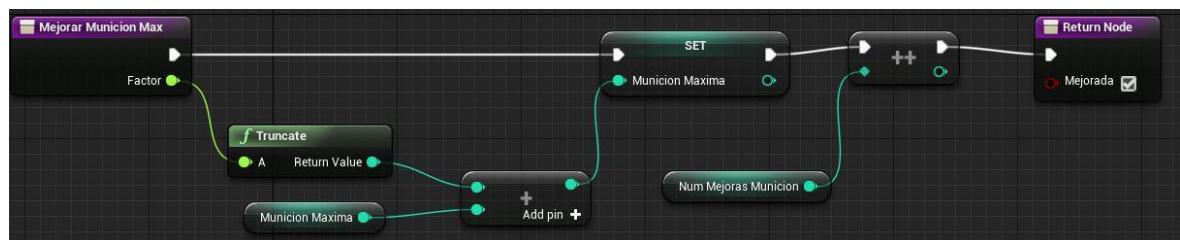


Figura 112: Función mejorar munición máxima

Recibe como parámetro el factor mediante el cual aumenta el número de balas de munición máxima. Tenemos que quedarnos con la parte entera al ser de tipo *float* y la munición máxima es un entero. Una vez tenemos la parte entera, se la sumamos a la munición máxima que podemos tener y aumentamos en uno el número de mejoras de munición máxima de esa arma.

5.3.2.2.3.6.3 Mejorar Capacidad Cargador

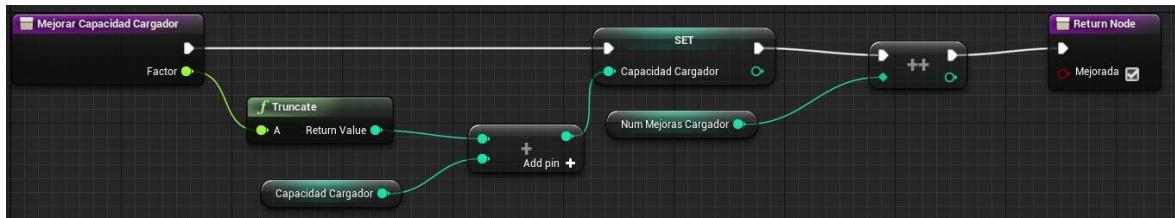


Figura 113: Función mejorar capacidad cargador

Al igual que en la función anterior, nos quedamos con la parte entera del factor de aumento, se lo sumamos a la capacidad del cargador y establecemos este resultado como la nueva capacidad del cargador. Finalmente, aumentamos en uno el número de mejoras de capacidad de cargador del arma que tengamos equipada en el momento en el que interactuamos con la máquina de mejorar capacidad cargador.

5.3.2.2.3.6.4 Mejorar Cadencia

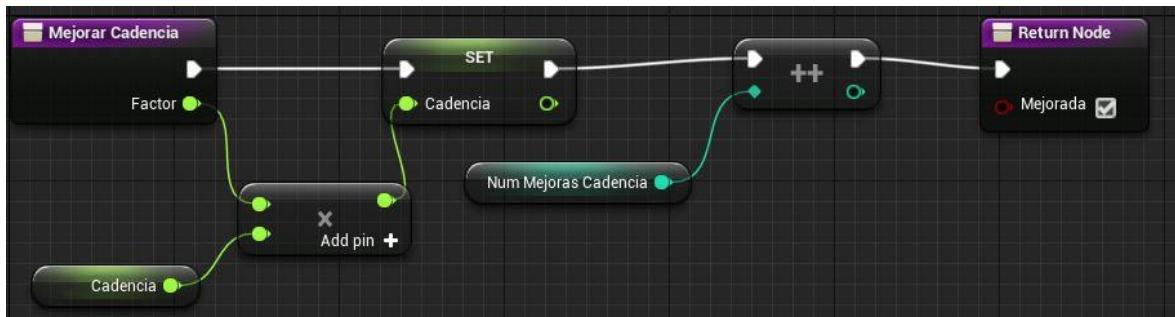


Figura 114: Función mejorar cadencia

Recibimos como parámetro el factor de aumento de cadencia. Este factor será menor a 1, por ejemplo 0.8, de manera que al multiplicarlo por nuestro tiempo de cadencia el resultado sea mejor, así disminuirá el tiempo entre disparo y disparo y aumentará la cadencia del arma. Nos guardamos en la variable **Cadencia** el resultado de la multiplicación del factor de aumento por la cadencia que teníamos. Terminamos aumentando en uno el número de mejora de cadencia para esta arma.

5.3.2.2.3.6.5 Mejorar Tiempo Recarga



Figura 115: Función mejorar tiempo recarga

Al igual que ocurría con la cadencia el factor recibido, aquí será menor a 1 para que disminuya el tiempo de recarga. El nuevo tiempo de recarga será el resultado de la multiplicación del antiguo tiempo de recarga por el factor recibido por parámetro. Finalmente, aumentamos en uno el número de mejoras de recarga para esta arma.

5.3.2.3 Interactive Objects (Power-ups)

Todos los objetos del juego con los que se puede interactuar sin tener en cuenta las armas se considerarán **interactive objects** y, por tanto, heredarán de una clase padre llamada **InteractiveParent**. Esto permite que todos los objetos cuenten con una misma interfaz común y puedan ser tratados todos por igual aunque internamente funcionen de manera completamente diferente.

A continuación se explicará el *Blueprint* de la clase padre y todos los objetos con los que podemos interactuar teniendo en cuenta que, cuando hablamos de *power-ups*, nos estamos refiriendo también a los objetos interactivos.

5.3.2.3.1 InteractiveParent Blueprint (clase padre objetos interactivos)

Esta es la clase padre de la que heredan todos los objetos con los que podemos interactuar. Este *Blueprint* contiene toda la lógica necesaria para poder interactuar con objetos salvo algunas funciones, que son sobrescritas por las clases hijas para que cada una pueda hacer su función aunque sean tratadas todas como si fueran un **InteractiveParent**.

5.3.2.3.1.1 Componentes

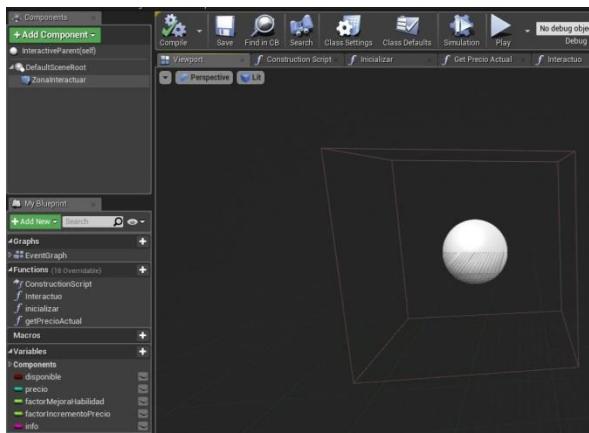


Figura 116: Componentes InteractiveParent

El único componente de esta clase es una caja de colisión llamada **ZonaInteractuar** que nos servirá para saber si estamos dentro de la zona en la cual podemos interactuar con un objeto.

5.3.2.3.1.2 Variables

InteractiveParent Variables		
Nombre	Tipo	Descripción
Disponible	<i>Boolean</i>	Variable que indica si un <i>power-up</i> está disponible o no. En el caso del <i>power-up</i> de revivir, por ejemplo, solo podrás comprarlo una vez, por lo que cuando no está disponible se desactiva la opción de compra aunque estés en su zona de interactuar.
Precio	<i>Int</i>	Indica el precio que costaría interactuar con el objeto.
FactorMejoraHabilidad	<i>Float</i>	Factor por el cual se incrementan las habilidades a las que afecte el objeto con el que estemos interactuando. Por ejemplo, si el factor es de 2 y estamos interactuando con una máquina de daño, significa que nos aumentará el daño del arma en 2.
FactorIncrementaPrecio	<i>Float</i>	Es el factor mediante el cual se incrementa el precio de un objeto interactivo cada vez que interactuamos con él, es decir, si este factor es de 2 cada vez que interactuemos con él nos costará el doble.

Info	<i>String</i>	Es un <i>string</i> con información sobre el <i>power-up</i> que se muestra luego por pantalla cuando estamos en zona de interactuar.
-------------	---------------	---

Figura 117: Tabla variables InteractiveParent Blueprint

5.3.2.3.1.2 Funciones

Aunque la lógica principal de los objetos interactivos recae sobre esta clase padre *InteractiveParent*, algunas de sus funciones son sobrescritas por las clases hijas para algún comportamiento específico.

5.3.2.3.1.2.1 ConstructionScript

Al igual que ocurría en otras clases padres, como *WeaponParentDrop* o *WeaponParent*, el constructor de esta clase padre está vacío, pues son las clases hijas las que inicializan el objeto en su constructor. Estos constructores los veremos individualmente cuando hablemos de cada objeto.

5.3.2.3.1.2.2 Inicializar

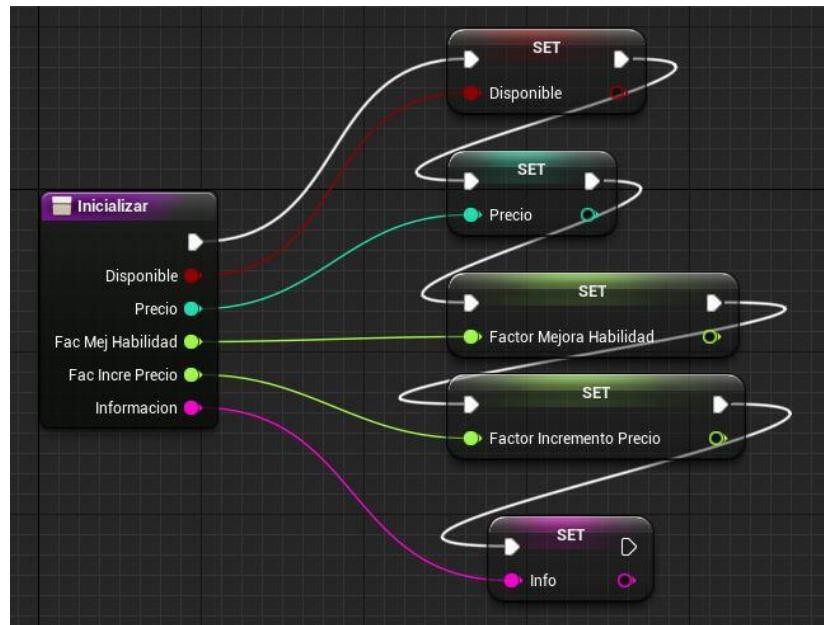


Figura 118: Función inicializar InteractiveParent Bluerpint

Es la función a la que llaman los constructores de las clases hijas. Se encarga de establecer los atributos del objeto.

5.3.2.3.1.2.3 Entrar en zona de interacción

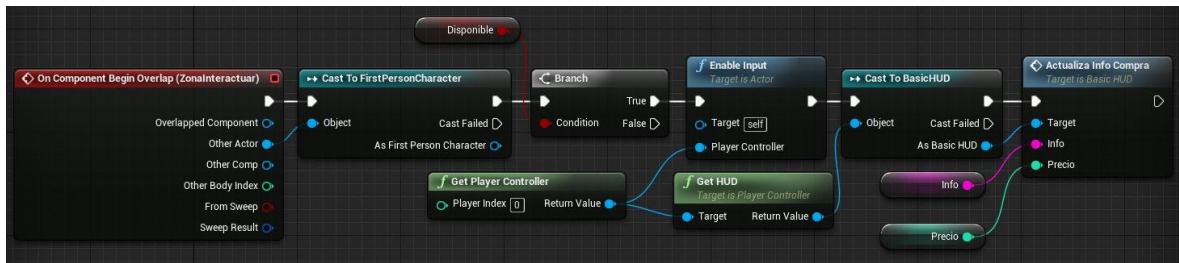


Figura 119: Evento entrar zona para interactuar

Este evento se ejecuta cuando el jugador entra dentro de la caja de colisión llamada **ZonaInteractuar**. Lo primero que hacemos es asegurarnos de que el actor que ha entrado en esta zona es el jugador (clase **FirstPersonCharacter**), porque puede darse que un zombi que te está persiguiendo pase cerca de un *power-up* y active este evento, por lo tanto sin esta comprobación podría desencadenarse un caos ya que un zombi que pase cerca de un power-up lo activaría.

Una vez que sabemos que es el jugador quien está dentro de la zona, constatamos que el *power-up* está disponible. Si lo está, entonces habilitamos el *input* para esta clase y también podemos activar ahora el **InputAction Interactuar** pulsando la tecla de interacción (tecla F del teclado). Por último, llamamos a la función **ActualizaInfoCompra** del **HUD** y le pasamos la información y el precio del objeto para que lo muestre por pantalla.

5.3.2.3.1.2.4 Salir de zona de interacción

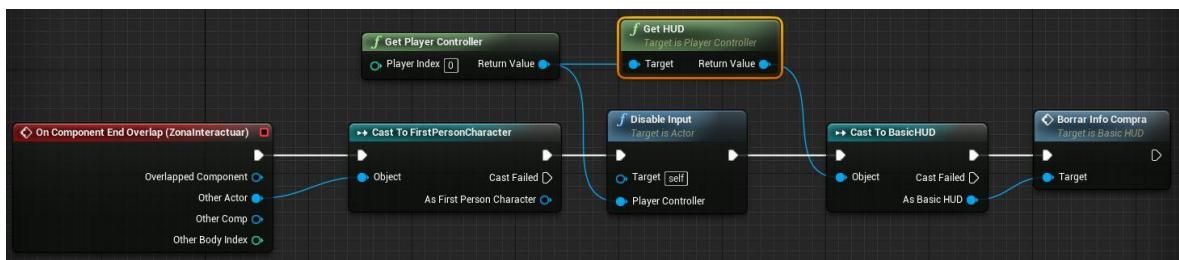


Figura 120: Evento salir de zona de interacción

Este evento es activado cuando un actor sale de la caja de colisión del objeto interactivo llamada **ZonaInteractuar**. Al igual que en el apartado anterior, tenemos que confirmar que el actor que salió de la zona es el jugador, de lo contrario podría ser una bala por ejemplo (la bala también es un actor) la que salió de la zona de interactuar y esta podría deshabilitar la compra aun cuando nosotros siguiéramos dentro de la zona. Después de esto desactivaremos el *Input* en esta clase de manera que,

aunque pulsemos la tecla para interactuar, no se llame al evento **InputAction Interactuar**. Por último, llamamos a la función **BorrarInfoCompra** del **HUD** para que borre la información del objeto que se estaba mostrando por pantalla.

5.3.2.3.1.2.5 InputAction Interactuar

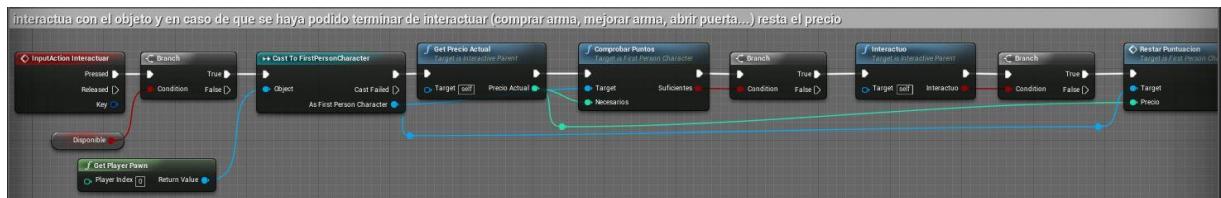


Figura 121: InputAction Interactuar

Este evento se activa cuando, estando dentro de la zona para interactuar de un objeto, pulsamos la tecla de interacción. Primero, comprobamos si el *power-up* con el que estamos interactuando está disponible y si es así llamamos a la función **getPrecioActual**, que nos devolverá el precio que cuesta la interacción con el objeto. Ese precio se lo pasamos como parámetro de entrada a la función **ComprobarPuntos** de la clase **FirstPersonCharacter**, que devolverá un *booleano* verdadero siempre que tengamos los puntos suficientes para interactuar con el objeto. En este caso, llamamos a la función **Interactuo**, que sobrescriben todas las clases hijas ya que cada una tiene una funcionalidad diferente. Por último, si hemos conseguido interactuar con el objeto, cosa que sabemos gracias al *booleano* que devuelve la función **Interactuo**, entonces llamamos a la función **RestarPuntuación** y le transferimos el precio que costó interactuar con el objeto. Esta función se encargará de restar al jugador los puntos que ha utilizado al comprar el *power-up*.

5.3.2.3.1.2.6 Función GetPrecioActual

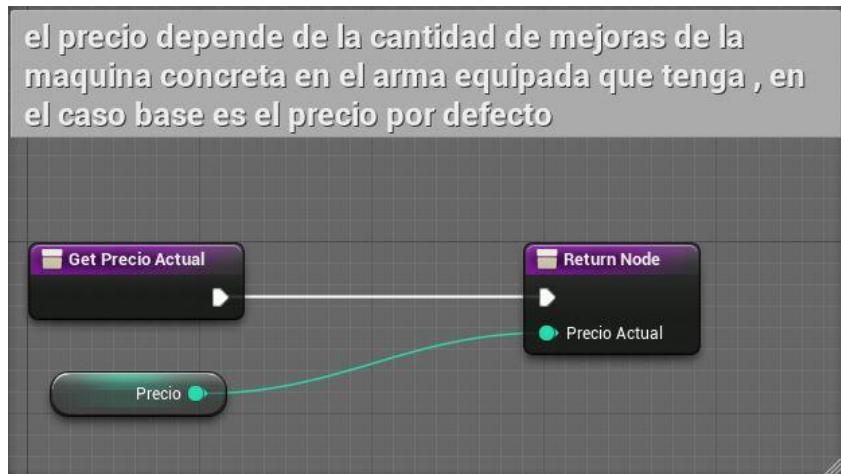


Figura 122: Función getPrecioActual

Esta función devuelve el precio actual del ***interactive object***. Se utiliza para algunos objetos, como por ejemplo las puertas, cuyo precio es siempre el mismo. Otros objetos como las máquinas de mejoras sobrescriben esta función al aumentar su precio con cada compra.

5.3.2.3.2 Clases hijas ***Interactive Objects***

Todos los objetos interactivos que figuran a continuación heredan de la clase padre ***InteractiveParent***. Los componentes de esta clase son la caja de colisión heredada de la clase padre y una malla que representa el objeto físico en el juego.

5.3.2.3.2.1 Puerta

5.3.2.3.2.1.1 Construction Script

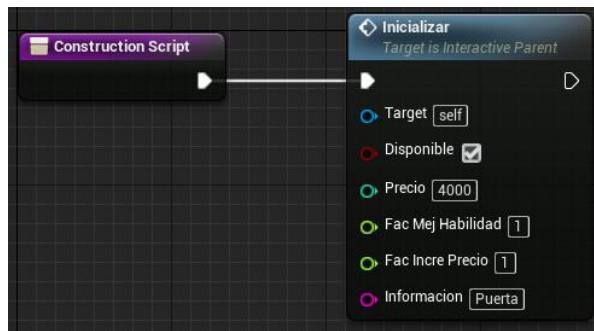


Figura 123: Constructor Puerta Blueprint

En el constructor se llama al método inicializar de la clase padre y le pasa todos los atributos de este objeto.

5.3.2.3.2.1.2 Función Interactúo

La función **Interactúo** de esta clase pone la variable **disponible** a falso para que no se pueda interactuar de nuevo con esta puerta (una vez que se abre ya no se puede cerrar) y llama a la función **AbrirPuerta** que vemos justo debajo:

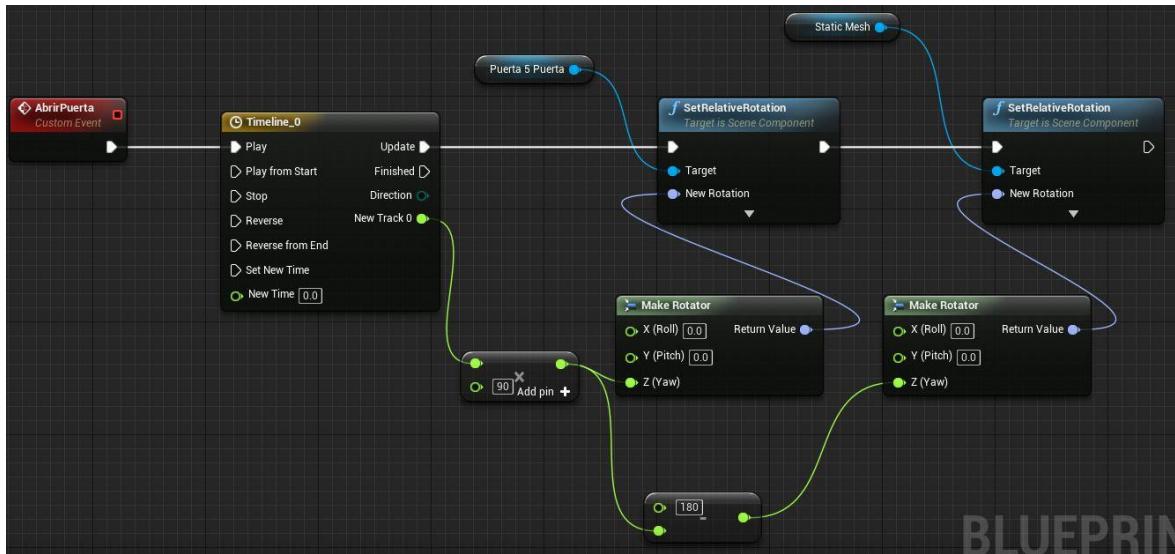


Figura 124: Función AbrirPuerta

Esta función utiliza un nodo del tipo **TimeLine** que nos permite hacer una interpolación lineal definida en una gráfica. La interpolación utilizada va de 0 a 1. Usamos la salida de este nodo (valor de la interpolación en cada *frame*) y la multiplicamos por 90, puesto que son los grados que queremos rotar la puerta. Creamos entonces un vector de rotación con el nodo **MakeRotator**, pero solo le pasamos el valor a la Z, que es en el eje en el que nos interesa girar la puerta (recordamos que el eje Z de *Unreal* es el que va hacia arriba). El resultado de este vector se lo pasamos como parámetro al nodo **setRelativeRotation**, que se encargará de rotar la puerta que recibe en “Target”.

Como podemos observar, se realizan dos rotaciones debido a que las puertas son dobles y se gira cada puerta individualmente. De esta manera la segunda puerta rotará igual que la primera pero girada 180°, por eso la rotación de la primera se la restamos a 180, para que las dos puertas giren en el mismo sentido.

5.3.2.3.2.2 Power-up resistencia

5.3.2.3.2.2.1 Construction Script

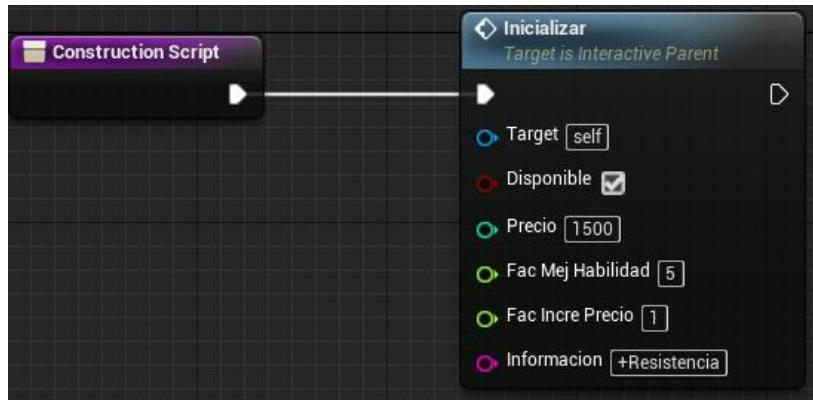


Figura 125: Constructor power-up resistencia

En el constructor se llama al método inicializar de la clase padre que inicializa todas las variables de este objeto.

5.3.2.3.2.2.2 Función Interactúo

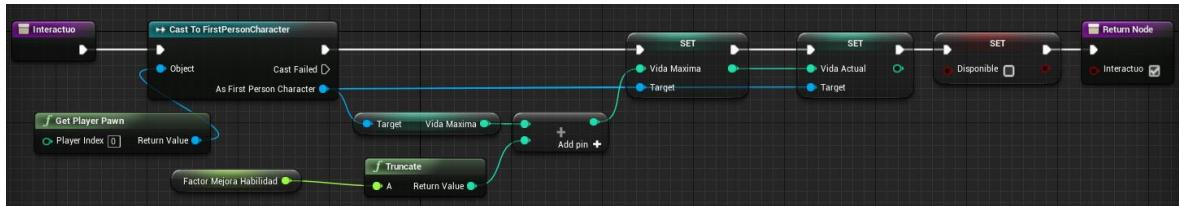


Figura 126: Función Interactúo (clase Power-upResistance)

El objetivo de esta función es aumentar la vida máxima del jugador para que sea más resistente. Para ello cogemos la vida máxima actual del jugador y le sumamos el factor de mejora de habilidad de este objeto. El resultado de esta suma se establece como nueva vida máxima y además se pone la vida actual al máximo (al nuevo máximo de vida), de manera que si estamos con poca vida en el juego también nos curaremos al instante al comprar esta mejora. Por último, ponemos la variable **disponible** a falso.

5.3.2.3.2.3 Power-up revivir

5.3.2.3.2.3.1 Construction Script

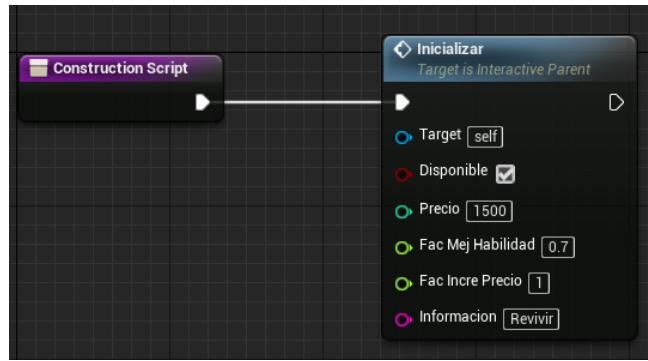


Figura 127: Constructor power-up revivir

5.3.2.3.2.3.2 Función Interactúo

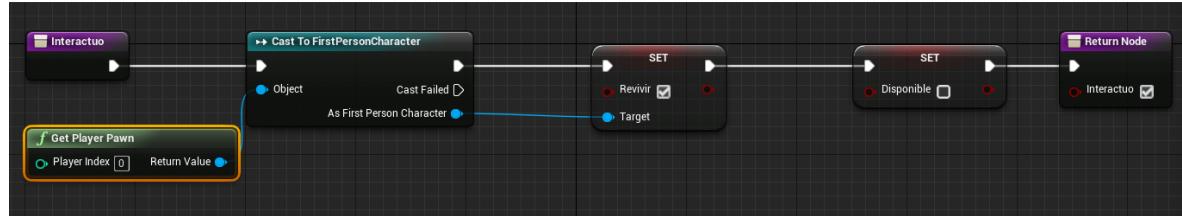


Figura 128: Función Interactúo (clase Power-upRevivir)

El objetivo de esta función es muy sencillo: conceder una segunda oportunidad al jugador, de manera que si muere una vez podrá revivir. Este “revivir” solo funcionará la primera vez que muere el jugador, después de comprar este *power-up*, y solo se podrá comprar una vez durante toda la partida.

En esta función únicamente se establece la variable **Revivir** a verdadera (para poder revivir al jugador) y la variable **disponible** a falsa para que no se pueda volver a comprar.

5.3.2.3.2.4 Power-up tiempo recarga

5.3.2.3.2.4.1 Construction Script

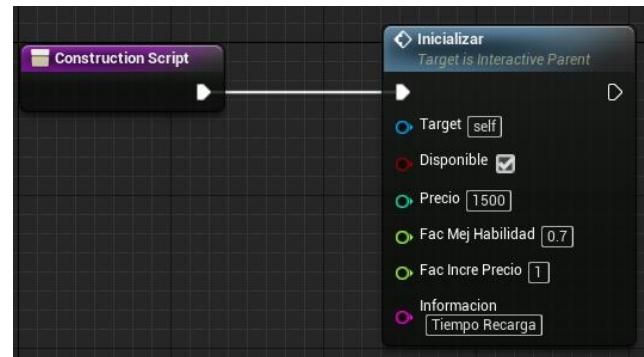


Figura 129: Constructor power-up tiempo recarga

5.3.2.3.2.4.2 Función Interactúo

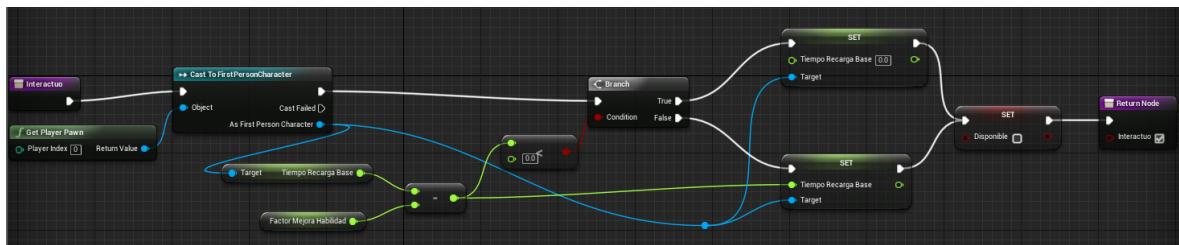


Figura 130: Función Interactúo (clase Power-upTiempoRecarga)

Esta función se encarga de aumentar el tiempo de recarga independientemente del arma que se esté usando, es decir, es una mejora de un atributo del jugador. Para ello tenemos que disminuir el tiempo de recarga base del jugador, que se suma al tiempo de recarga de cualquier arma. A este tiempo base se le resta el factor de mejora de este objeto y, si la resta da un número negativo, el tiempo de recarga base se establece en 0, si no el tiempo de recarga base será el resultado de la resta anterior. Por último, se pone la variable **disponible** a falso para que no se pueda volver a comprar esta mejora.

5.3.2.3.2.5 Máquinas mejorar armas

Puesto que la lógica de las cinco máquinas de mejorar armas existentes es similar, mostraremos únicamente el constructor y la función de interactuar de dos ellas.

Estas máquinas pueden ser compradas más de una vez y su precio aumenta cada vez que se compra, por esto se sobrescribe la función **getPrecioActual** de la clase padre.

5.3.2.3.2.5.1 Construction Script

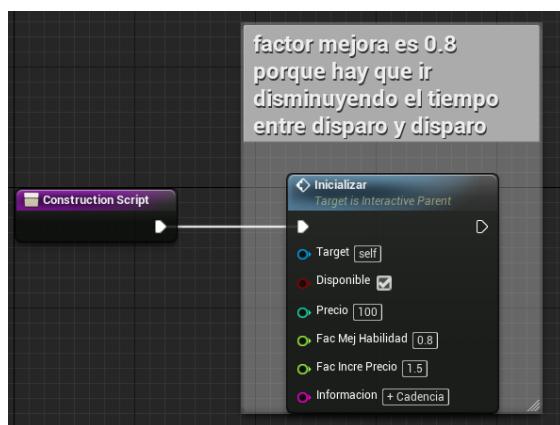


Figura 131: Constructor maquina mejorar cadencia

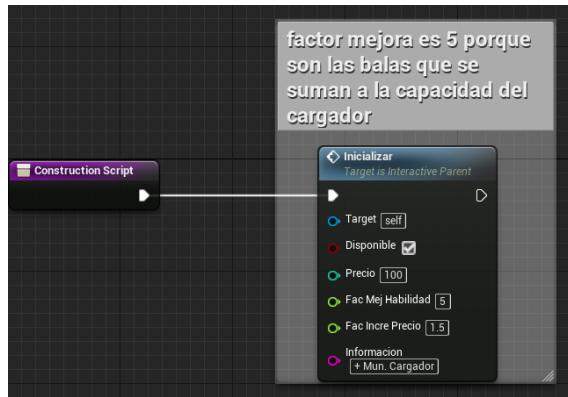


Figura 132: Constructor maquina mejorar capacidad cargador

5.3.2.3.2.5.2 Función Interactúo

Estas funciones se dedican a llamar a la función que mejora el atributo del arma que corresponde con la máquina de mejora. Cuando se mejora un atributo de un arma es únicamente del arma equipada.



Figura 133: Función Interactúo maquina mejorar cadencia



Figura 134: Función Interactúo maquina mejorar capacidad cargador

5.3.2.3.2.5.2 Función GetPrecioActual

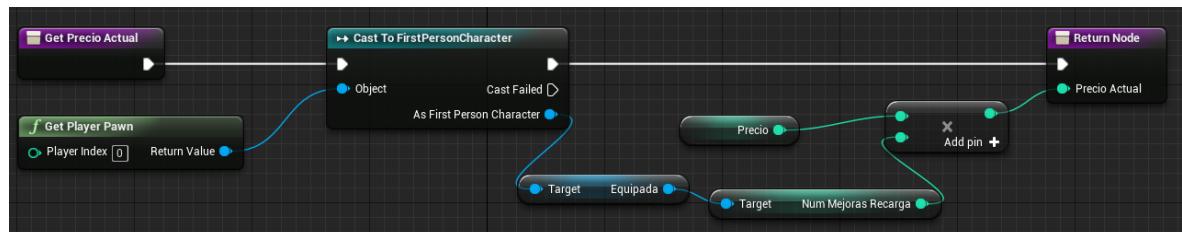


Figura 135: Función getPrecioActual (común a todas las máquinas de mejora)

El precio de una máquina de mejora depende de las veces que hemos comprado esta mejora para el arma actual. Esto quiere decir que, por ejemplo, si tenemos una UMP y un AK en el inventario y hemos mejorado tres veces el daño del AK y ninguna vez el daño de la UMP, cuando nos acercamos a la máquina de mejorar daño con el AK equipada nos indicará que el precio de la compra es tres veces el precio inicial, mientras que si nos acercamos con la UMP el precio de la compra será el precio inicial.

Para calcularlo simplemente multiplicamos el número de mejoras del arma del mismo tipo que la máquina con la que estamos interactuando por el precio inicial.

5.3.2.4 Enemigo

Dado que los enemigos del juego son zombis, conviene explicar el *Blueprint* principal llamado **Zombi**, el *behaviour tree* encargado del comportamiento del zombi.



También hablaremos del *Animation Blueprint* que usa el *skeletal mesh* del zombi y de las notificaciones en las animaciones, ya que los ataques se realizan utilizando esta notificación en mitad de la animación con el objeto de conseguir un mayor realismo y que el zombi ataque justo cuando el brazo está extendido y puede incluso esquivar el ataque si eres lo suficientemente rápido.

En la foto de la izquierda podemos observar al zombi enemigo cuya apariencia es coherente con la historia del videojuego.

Figura 136: Enemigo

5.3.2.4.1 Zombi Blueprint

El *Blueprint* del enemigo de *PostWar: Hopeless Humanity* tiene una serie de componentes, variables y funciones que analizaremos a continuación.

5.3.2.4.1.1 Componentes

Se trata de un *Blueprint* del tipo *pawn*, porque necesita ser controlado por un *Controller* (que pueden ser tanto *Inputs* como la IA). Por este motivo, heredamos una serie de componentes como *CharacterMovement* que ya habíamos visto en el *Blueprint* del *FirstPersonCharacter*. Los componentes de este *Blueprint* son:

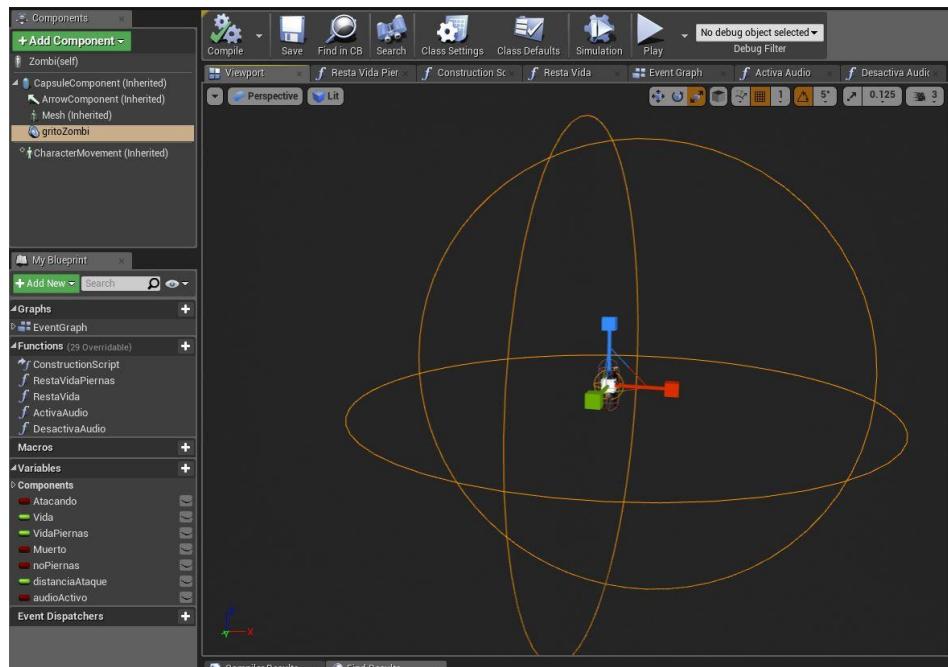


Figura 137: Componentes Zombi Blueprint

- **CapsuleComponent:** es la cápsula que simula la colisión del personaje, como se aprecia en la foto al principio del apartado “Enemigo” (véase figura 136). Esta cápsula se mueve y, dado que el resto de componentes son hijos de esta cápsula, todos los componentes se moverán junto a ella.
- **Mesh:** es la malla esquelética (*skeletal mesh*) utilizada para representar en el mundo este actor. Es el modelo del zombi, que contiene también las animaciones.
- **GritoZombi:** es el sonido que hace el zombi. Este componente figura seleccionado en la imagen de arriba para que se puedan observar los radios de atenuación. Si nos fijamos bien, hay una esfera bastante pequeña en el centro de la esfera grande. Esa esfera pequeña es el radio interior y, si estamos dentro de esa esfera (estamos pegados al zombi), escucharemos el sonido al máximo de volumen. Cuando estamos entre las dos esferas, percibimos el sonido con una atenuación lineal que depende de la distancia a la que nos encontremos. Si salimos de la segunda esfera (la esfera grande), no oiremos nada. Esto es así debido a la gran cantidad de zombis que puede haber al mismo tiempo mientras jugamos y para que solo escuchemos aquellos que estén más cerca. Si no fuera así, la situación devendría un absoluto caos con los gritos de 100 zombis por ejemplo. Esto se puede ver mejor en el siguiente video:

<https://www.youtube.com/watch?v=kMox3qvbykU> realizado únicamente para mostrar este efecto de la atenuación del sonido del grito del zombi.

- **CharacterMovement:** igual que en el Blueprint del jugador (*FirstPersonCharacter*), este componente se encarga de permitir al personaje moverse por el escenario.

5.3.2.4.1.2 Variables

Zombi Variables		
Nombre	Tipo	Descripción
Atacando	<i>Boolean</i>	Nos indica si estamos atacando o no. Será verdadero cuando atacamos y falso cuando no estamos atacando.
Vida	<i>Float</i>	Indica la vida actual del zombi.
VidaPiernas	<i>Float</i>	Indica la vida actual de las piernas del zombi. Si llega a 0, el zombi no podrá caminar y por lo tanto tendrá que ir arrastrándose.
Muerto	<i>Boolean</i>	Variable booleana que indica si un zombi está muerto o no.
NoPiernas	<i>Boolean</i>	Esta variable será verdadera cuando no tiene piernas y falsa cuando aún tiene piernas. Será usada en la máquina de estados de animación para saber si el zombi puede caminar o, por el contrario, tendrá que arrastrarse.
DistanciaAtaque	<i>Float</i>	Es la distancia a partir de la cual un zombi puede atacar. Esta variable es utilizada en el <i>Behaviour Tree</i> para saber cuándo pasar a posición de ataque.
AudioActivo	<i>Boolean</i>	Nos dice si el audio está activo (verdadera) o si el audio no está activo (falsa).

Figura 138: Tabla variables Zombi Blueprint

5.3.2.4.1.3 Funciones

Las funciones del *Blueprint Zombi* incluyen algunas muy sencillas que serán explicadas, pero no se pondrá una foto de ellas.

5.3.2.4.1.3.1 Evento *BeginPlay*

En el evento *BeginPlay* de esta clase se obtiene la vida que posee un zombi accediendo a la referencia de la clase **Partida** que tiene el *player*. Esta vida dependerá del número de ronda en la que nos encontraremos. Establecemos esta vida como vida del zombi y ponemos la vida de las piernas a 1/4 la vida del zombi. De esta manera si el zombi tiene 100 puntos de vida, sus piernas tendrán 25 puntos de vida.

5.3.2.4.1.3.2 Atacar

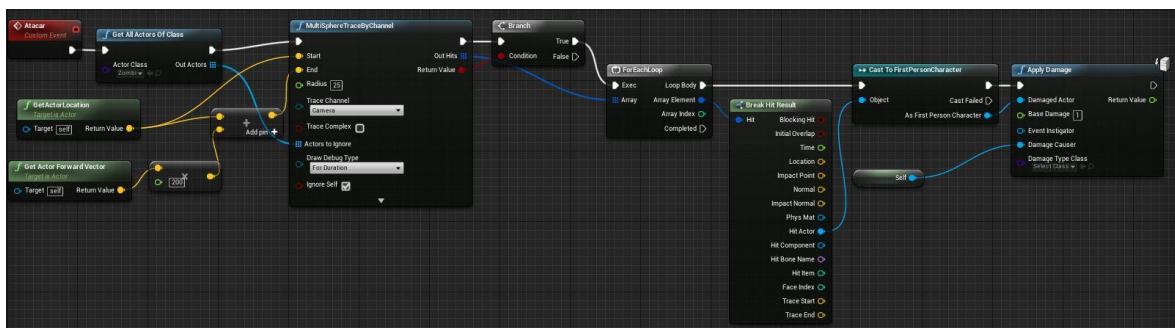


Figura 139: Evento atacar

Cada vez que se llama a este evento el enemigo realizará un ataque.

Si recordamos cómo se hacía el disparo del enemigo, consistía en trazar una línea (*raycast*) con la dirección de la bala para ver con qué actores chocaba. Lo hacíamos con el nodo ***LineTraceByChannel***. Sin embargo, ahora no nos interesa lanzar una línea solo, ya que no es una bala sino un brazo el que efectúa el ataque y, por lo tanto, necesitamos algo con mayor volumen. Lo que lanzamos ahora para comprobar si el ataque ha alcanzado a algún enemigo es una esfera a través de la trayectoria de la línea. Para ello usamos el nodo ***MultiSphereTraceByChannel***. Este nodo recibe varios parámetros de entrada, entre los que se encuentran:

- **Start:** es el punto de inicio desde el que se lanzará la esfera. Este punto será la posición del enemigo, así el ataque empezará siempre desde la posición del enemigo que lo haya lanzado.
- **End:** es el punto final de la línea a través de la cual “viaja” la esfera encargada de detectar los actores con los que impacta este ataque. Dicho de otra manera más sencilla, sería el alcance del ataque. Para calcular este punto sumamos la posición del enemigo con el resultado de multiplicar el vector dirección por 200.
- **Radius:** es el radio de la esfera que estamos usando para detectar las colisiones del ataque.

- **Actors to Ignore:** es una lista con todos los actores a los cuales no afectará el ataque y por tanto con los que la esfera no colisionará. Por este parámetro de entrada le pasamos todos los enemigos que estén vivos, de esta manera los ataques de los zombis no afectarán a otros zombis, que es justo lo que queremos. Para conseguir esto, obtenemos todos los zombis mediante el nodo **GetAllActorOfClass**, que nos devuelve un *array* con todos los actores de la clase elegida (clase **Zombi**), y este *array* es el que le pasamos como parámetro de entrada al nodo **MultiSphereTraceByChannel**.
- **IgnoreSelf:** Marcamos esta casilla para que el ataque ignore al propio actor que lo lanza, acción necesaria puesto que, cuando lanzamos el ataque, lo hacemos desde la posición del enemigo y, por lo tanto, podía detectar colisión consigo mismo.

Este nodo nos da como resultado un *array* con todos los resultados de impacto. Para recorrerlos todos y poder tratarlos individualmente, recurrimos al nodo **ForEachLoop**, le pasamos como parámetro de entrada el *array* con todos los resultados del ataque y nos los divide uno a uno, luego ejecuta el código siguiente a este nodo **ForEachLoop** para cada uno de los resultados.

Entonces, para cada resultado del *for each* tendremos un **BreakHitResult** al igual que lo teníamos cuando disparábamos, aunque en este caso solo nos interesa saber si el actor con el cual ha colisionado es el jugador. Para conocer este dato, seleccionamos la salida **HitActor** que devuelve el actor con el que colisionó y lo *casteamos* a **FirstPersonCharacter** (clase del jugador) y, si el *casteo* es posible, significa que el ataque le dio al jugador así que le enviaremos un mensaje para que este sereste vida.



Figura 140: Ejemplo del ataque de un zombi

La foto de arriba muestra lo explicado en este apartado sobre el ataque del zombi. Las cápsulas corresponden a los ataques que realizaron los zombis desde esa posición. Las rojas y las verdes representan el resultado del recorrido de la esfera a través de la línea con un radio de 25 y un alcance de 200. Los cuadrados en rojo son los puntos de impacto.

Estas cápsulas están solo a modo de *debug* visual y no aparecerán en el juego final, además se ha hecho que perduren en el tiempo para mostrar este efecto.

Si el enemigo está en el suelo, el ataque no será con la mano del zombi sino que intentará morder desde el suelo. Como este ataque es más difícil que te alcance debido a que la distancia de ataque es bastante menor y la velocidad con la que se mueve el zombi también es menor, el daño será mayor.

5.3.2.4.1.3.3 Recibir impacto bala

En la siguiente captura de la lógica del evento **impacto** se han dejado los nodos **PrintString** (utilizados a modo de prueba para saber si funcionaba bien) de manera que sea más fácil comprender cuándo impacta en las piernas, cabeza o torso. En la versión final no estarán.

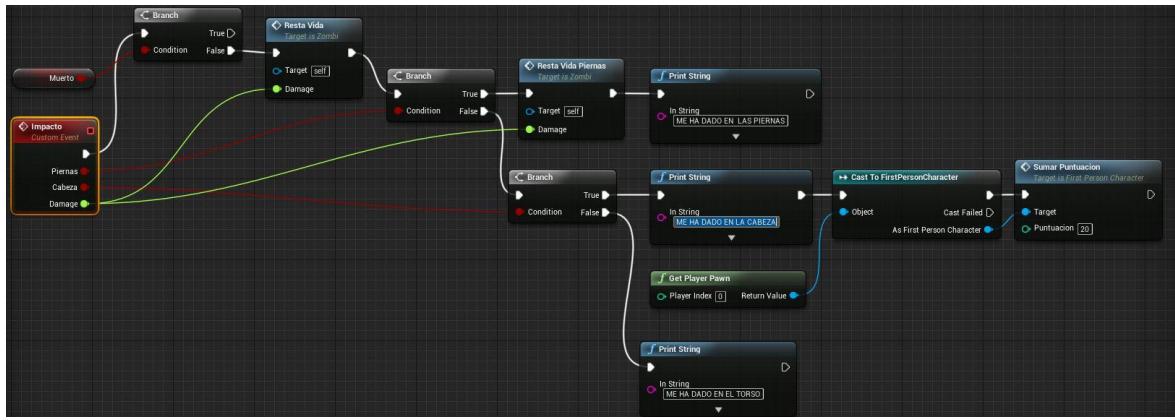


Figura 141: Evento recibir impacto

Este evento se llama cuando ya hemos tratado el impacto de una bala sobre un zombi, sabiendo el daño causado y dónde impactó. Recibimos tres parámetros de entrada, dos *booleanos* y *float*. El primer *booleano*, “Piernas”, será verdadero si el impacto fue en las piernas. El segundo *booleano*, “Cabeza”, será verdadero si el impacto fue en la cabeza. El tercero es el daño que nos han causado.

El primer paso si no hemos muerto ya es llamar a la función **RestaVida** y le pasamos el daño. Esta función se llama al principio de todo independientemente del sitio donde nos hayan dado, tanto si nos dan en el torso, como en las piernas o en la cabeza. Una vez hecho esto, comprobamos si el impacto fue en las piernas, cabeza o torso mediante *if else* anidados (nodos **branch**).

Si el impacto es en las piernas, llamamos a la función **RestaVidaPiernas** y descontará el daño causado a las piernas, puesto que la vida de las piernas es independiente a la vida total del zombi a la que ya se le restó el daño correspondiente. Si el impacto alcanza la cabeza, sumamos 20 puntos al jugador llamando a la función **SumarPuntuación**. Si el impacto lo recibe el torso, entonces no ocurrirá nada especial puesto que ya hemos restado la vida al principio.

5.3.2.4.1.3.4 Restar vida

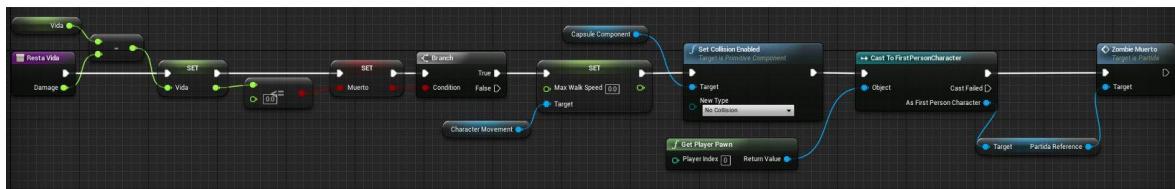


Figura 142: Función RestaVida de la clase Zombi

Esta función recibe por parámetro la cantidad de daño que nos han causado y se la sustraen a la vida actual del zombi. Además comprueba si el zombi ha muerto o no.

Si el zombi muere, se le pone la velocidad a 0 para que, mientras reproduzca la animación de muerte, no siga moviéndose; se le quita la colisión para que no obstruya al resto de zombis y puedan atravesarlo; finalmente, se llama a la función **ZombieMuerto** de la clase **Partida** para notificar que hemos matado un zombi.

5.3.2.4.1.3.5 Restar vida piernas

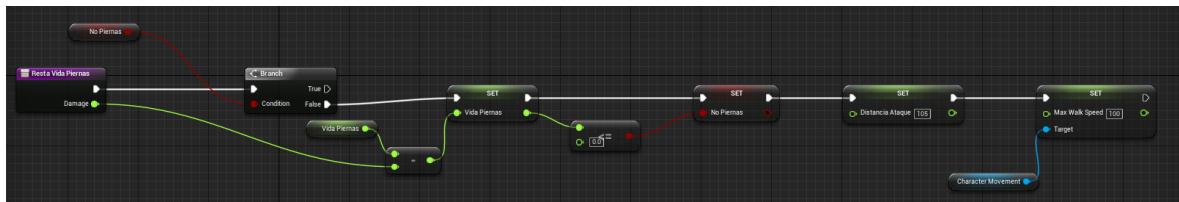


Figura 143: Función RestaVidaPiernas

Recibimos como parámetro de entrada el daño que nos han infligido y restamos este daño a la vida actual de las piernas. Si la vida de las piernas es menor o igual a 0, ponemos la variable **NoPiernas** a *true*, disminuimos la distancia de ataque desde 200 (distancia cuando el zombi está de pie) hasta 105 (distancia cuando el zombi se arrastra), ya que ahora no dará un manotazo sino que te morderá desde el suelo, y también disminuimos su velocidad máxima de 350 a 100, pues es lógico que si se arrastra se mueva mucho más lento que cuando camina.

5.3.2.4.1.3.6 Activar *Ragdoll*

Esta función se llama desde el *EventGraph* del *Animation Blueprint* del zombi y se hace en un punto exacto de la animación de muerte a partir del cual activamos el *ragdoll*, el cuerpo caerá desplomado al suelo y le podremos disparar para desplazarlo. Después de activar el *ragdoll*, se comienza una cuenta de 6 segundos, pasados los cuales se destruye el actor.

5.3.2.4.1.3.7 Activar sonido

Esta función comprueba si el audio del zombi está desactivado para activarlo; en caso contrario, no hace nada.

5.3.2.4.1.3.8 Desactivar sonido

Esta función comprueba si el audio del grito del zombi está activo. Si está activo, lo desactiva; si ya estaba desactivado, no hace nada.

5.3.2.4.1.3.9 Detectar si el sonido ha terminado

El evento **OnAudioFinished (gritoZombi)** se encarga de detectar si el audio ha finalizado. Si lo ha hecho, ponemos la variable **AudioActivo** a falsa. Necesitamos esta función porque puede ser que estés en una zona con muchos zombis o que un zombi te persiga durante más tiempo de lo que dura el sonido del grito del zombi, por lo tanto detectando cuándo termina este sonido podremos volver a activarlo.

5.3.2.4.2 Animation Blueprint

Este *Blueprint*, a diferencia de los anteriores, carece de componentes ya que no se va a representar físicamente en el mundo. En cambio, dispone de dos grafos bien diferenciados: el primero es el **EventGraph** con el que estamos familiarizados. El segundo, en cambio, contiene novedades. Se llama **AnimGraph**, contiene la lógica de las animaciones y tendrá como resultado la pose final del modelo en cada *frame*. Para decidir qué pose elige en cada *frame*, consta de una máquina de estados que veremos más abajo.

5.3.2.4.2.1 Variables

Animation Blueprint Variables		
Nombre	Tipo	Descripción
Velocidad	<i>Float</i>	Es la velocidad a la que se está moviendo el zombi. Se usa para el <i>blending</i> entre las animaciones <i>Idle</i> , <i>Walk</i> y <i>Run</i> .
Atacar	<i>Boolean</i>	<i>Booleano</i> que es cierto cuando el zombi está atacando. Se utiliza en la máquina de estados para saber si se tiene que reproducir la animación de atacar.
SinPiernas	<i>Boolean</i>	Será cierto cuando el zombi no tenga piernas y, por tanto, tendrá que arrastrarse.
HeMuerto	<i>Boolean</i>	Variable <i>booleana</i> que será cierta cuando el zombi haya muerto. Gracias a esta variable sabemos si tenemos que reproducir la animación de muerte
AnimMuerte1	<i>Boolean</i>	Si esta variable es cierta se reproducirá la animación de muerte número 1.

AnimMuerte2	<i>Boolean</i>	Si esta variable es cierta se reproducirá la animación de muerte número 2.
--------------------	----------------	--

Figura 144: Tabla variables Animation Blueprint

5.3.2.4.2.2 Event Graph

En el *EventGraph* de este *Blueprint* encontramos tres eventos.

5.3.2.4.2.2.1 Update

Este es el evento principal de este *Blueprint*. Sin este evento no actuaría la máquina de estados, puesto que esta funciona mediante una serie de condiciones que no se pueden actualizar en la máquina de estados de animación sino que se actualizan aquí.



Figura 145: Evento Update Animation Blueprint

Esta función es bastante sencilla: simplemente se encarga de establecer los valores de las variables para que puedan ser usados en la máquina de estado. Así se utiliza la velocidad del zombi para establecer la variable **Velocidad**. Ocurre lo mismo con el resto de variables, es decir, se utilizan las del zombi para darles valor.

Si el zombi muere, se selecciona una animación de las dos animaciones de muerte disponibles.

5.3.2.4.2.2.2 Notificación ataque zombi

Este evento se encarga de llamar a la función **Atacar** de la clase **Zombi** y se ejecuta cuando llega una notificación del tipo **ZombieAttack** desde la animación de ataque.

5.3.2.4.2.2.3 Notificación ragdoll

Este evento se encarga de llamar a la función **ActivarRagdoll** de la clase **Zombi** y se ejecuta cuando llega una notificación del tipo **Ragdoll** desde cualquiera de las dos animaciones de muerte.

5.3.2.4.2.3 State Machine (Animation)

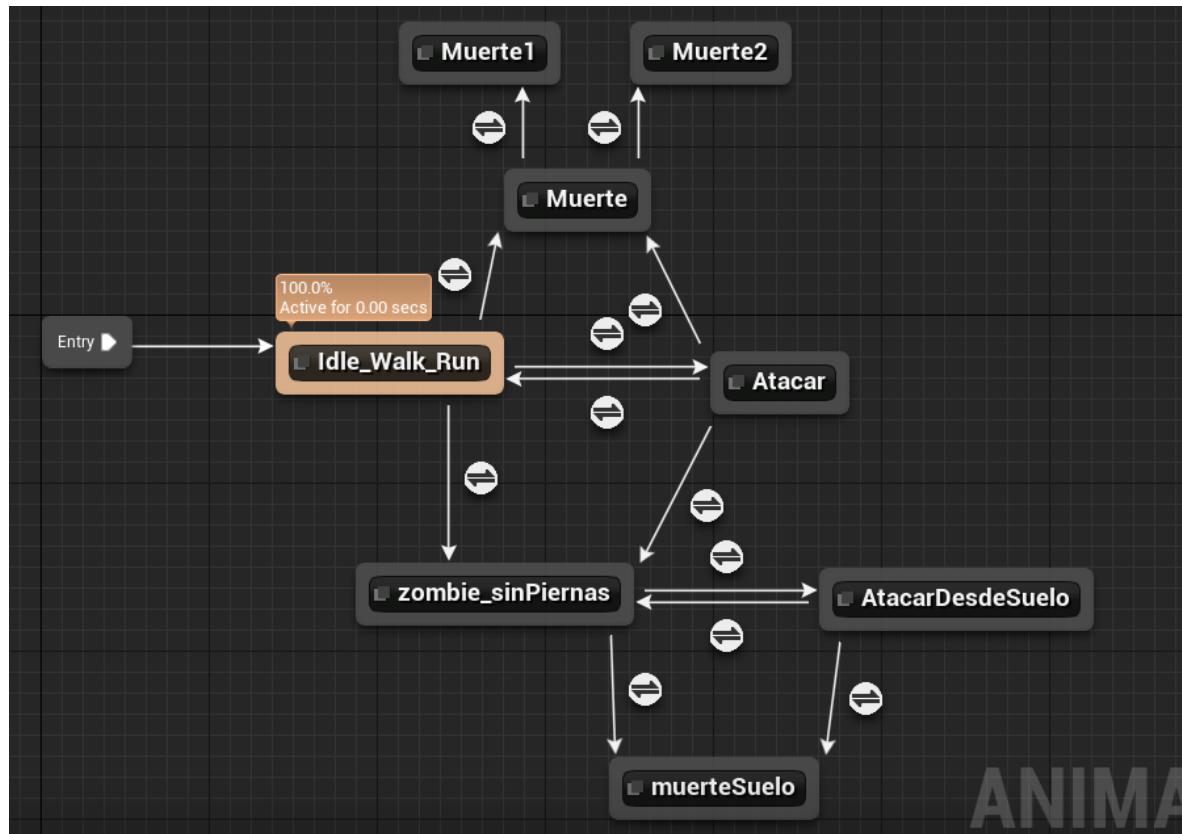


Figura 146: Máquina de estados animación

El nodo **Idle_Walk_Run** se ejecuta por defecto. Utiliza la velocidad del zombi con el propósito de saber si está quieto, andando o corriendo y practica una fusión entre estas animaciones para que, por ejemplo, pueda estar andando rápido y no se vea un salto de la animación de andar a la animación de correr.

Desde este nodo, cuando el zombi ataca pasará al nodo **Atacar**, que activará la animación de ataque. Si el zombi muere, se irá al nodo **Muerte** desde el cual se verá qué animación de muerte de las dos disponibles hay que activar. Cuando el zombi se queda sin piernas, pasaremos al nodo **zombie_sinPiernas**. (Como podemos observar, aquí solo hay una flecha que va desde **Idle_Walk_Run** a **zombie_sinPiernas**, lo que es lógico pues una vez que nos quedamos sin piernas ya no podremos volver a las animaciones de caminar). Si estamos sin piernas y atacamos, se activará la animación de ataque desde el suelo contenida en el nodo **AtacarDesdeSuelo**. (Aquí sí que podemos observar que hay flechas en ambas direcciones, puesto que cuando dejamos de atacar tenemos que volver a la animación de movimiento desde el suelo). Por último, si morimos desde el suelo se lanzaría la animación de muerte desde el suelo.

5.3.2.4.2.4 Notify Animation

Las notificaciones en las animaciones se realizan en momentos concretos en los que se activa la notificación.

5.3.2.4.2.4 Notificación en animación de ataque (zombi de pie)

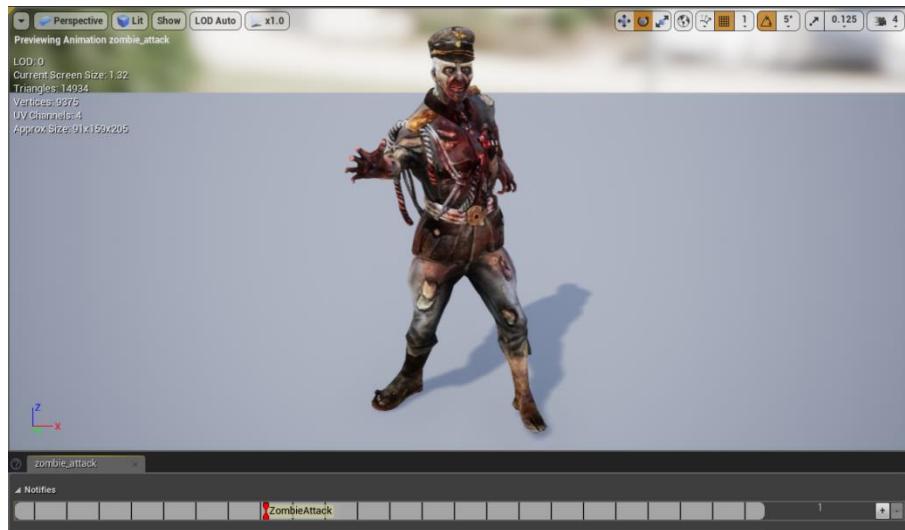


Figura 147: Notificación animación ataque (zombi de pie)

Como podemos observar, esta notificación es activada cuando el brazo del zombi está completamente extendido. Al hacerlo así ganamos en realismo pues, aunque el zombi empiece la animación de ataque, esta no hará daño hasta que llegue a esta parte de la animación y se llame a la función Atacar del zombi.

Otra ventaja de emplear la notificación en la animación de ataque es que no tenemos que estar controlando tiempos de ataque, es decir, no tenemos que usar tiempos para que si me ha atacado no me haga daño hasta que pasen tantos segundos (cosa que pasaría si el zombi hiciera daño por contacto), ya que solo hará daño cuando llegue a este punto. Se llamará a la función de Ataque una única vez por animación, ajustando así la velocidad con la que el zombi puede dañarte a su velocidad de ataque definida por la animación.

5.3.2.4.2.4 Notificación en animación de ataque (zombi en el suelo)

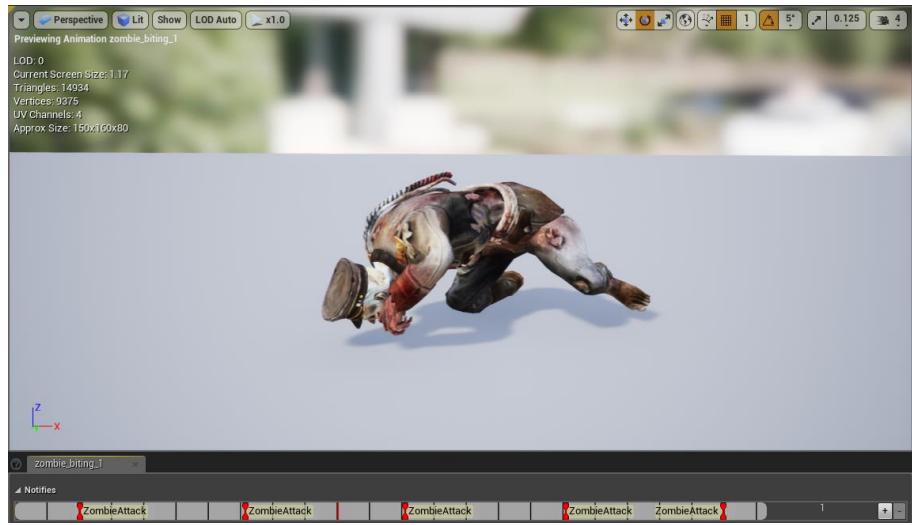


Figura 148: Notificación animación ataque (zombi en el suelo)

Esta animación muestra que hay varias notificaciones. Se ha hecho así porque es más difícil que el zombi te muerda cuando está en el suelo y su distancia de ataque y su velocidad es menor. Si consigue morderte, te hará daño cada 0.5 segundos (intervalos que hay entre una notificación y la siguiente). De esta manera cuanto más tiempo permanezcas al lado del zombi que te está mordiendo, más vida te quita.

Con esta estrategia compensamos la dificultad que existe de que un zombi te muerda desde el suelo y evitamos el gran daño que este causa.

5.3.2.4.2.4 Notificación en animación muerte 1



Figura 149: Notificación en animación de muerte 1

En esta animación el zombi cae hacia atrás agitando los brazos. Mientras está cayendo, activaremos el ***ragdoll***, lo que permite que a partir de ese punto caiga desplomado y lo pueda hacer cada vez de una forma diferente.

5.3.2.4.2.4 Notificación en animación muerte 2



Figura 150: Notificación en animación de muerte 2

En esta animación el zombi cae hacia delante y queda clavado de rodillas en el suelo. En ese preciso momento se activará el ***ragdoll*** para que desde esta posición el zombi pueda caer hacia cualquier lado y podamos dispararle para mover su cuerpo.

5.3.2.4.2.4 Notificación en animación muerte con el zombi en el suelo

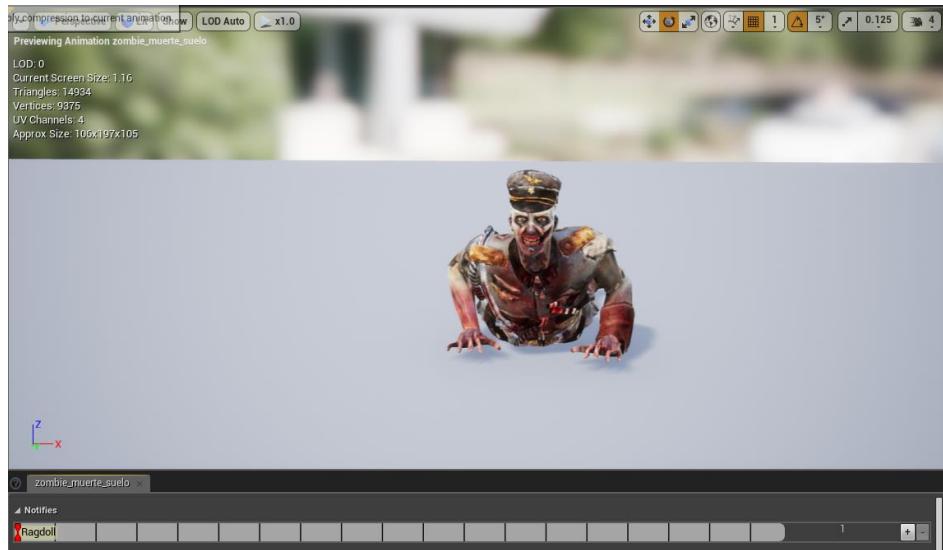


Figura 151: Notificación en animación muerte zombi en el suelo

Cuando estamos en el suelo, no disponemos de ninguna animación de muerte y tampoco podemos utilizar una de las anteriores puesto que el zombi se levantaría para morir y es algo que no deseamos. Para solucionar esto, decidí crear una nueva animación a partir de la animación de movimiento del zombi en el suelo, en la cual se activa el ***ragdoll*** en el primer *frame*. El resultado de esto es que cuando matas a un zombi que está en el suelo cae desplomado instantáneamente.

5.3.2.4.3 IA

Para un correcto funcionamiento de una IA en *Unreal Engine 4* precisamos de cuatro componentes básicos que pueden aumentar según la complejidad de la IA. Estos componentes son:

- ***Behaviour Tree***: Es el componente principal la lógica de la IA. Sería por tanto como el cerebro de esta. Nos permite establecer comportamientos en función de una serie de condiciones cuyo resultado será una acción del personaje que posea este *behaviour tree*.
- ***Blackboard***: Sirve para almacenar variables que utilizaremos luego con el *behaviour tree*. Cada *behaviour tree* tiene su *blackboard*.
- ***Blueprint de tipo Character***: Este *blueprint* sería nuestra clase **Zombi**. Este actor corresponde al personaje que va a ser poseído por el *Controller* de la inteligencia artificial (**AIController**) que se encargará del movimiento del personaje.
- ***Blueprint de tipo AIController***: Este *Blueprint* posee al *Blueprint* del tipo *Character* y es el encargado de su movimiento.

Aunque hayamos definido todos los componentes anteriores, el movimiento de nuestra IA no funcionará hasta que definamos un ***Nav Mesh Bounds Volume*** que se usará para calcular los caminos

por los que el enemigo puede pasar. Esto se aprecia mejor en la imagen de abajo: las zonas en verde corresponden a los lugares por donde podrá pasar un enemigo.



Figura 152: Vista del NavMeshVolume en el escenario

Las zonas en las que hay obstáculos, como pueden ser las camas, las mesas, sillas, etc. no están de color verde lo que significa que el enemigo no podrá atravesarlas sino que tendrá que esquivarlas. De la misma manera no podrá pasar de una sala a otra hasta que no se abra la puerta que permita la transición entre dos salas.

El *Behaviour Tree* de nuestra IA es el siguiente:

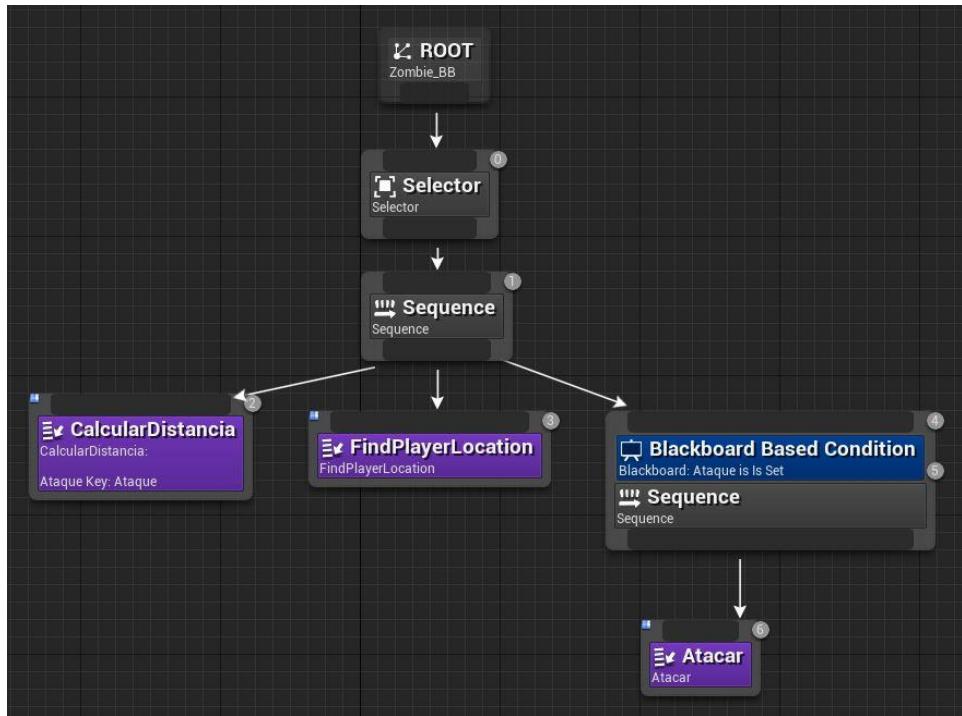


Figura 153: Behaviour Tree Zombi

Es preciso aclarar que los nodos son distintos a los que ya estamos habituados. Los nodos en los cuales se puede incluir lógica son llamados **Task** y son siempre los nodos hojas que no tienen una conexión de salida. Los **Task** son una especie de *Blueprint* en los cuales, al igual que en el resto, podemos definir eventos, funciones, variable, etc.

El primer nodo que se ejecuta es el de **CalcularDistancia**. En él calcularemos la distancia del zombi hasta el jugador. Este nodo se encarga de activar y desactivar el sonido del zombi cuando nos acercamos o alejamos. Además si la distancia es suficiente para el ataque, pondremos la variable **Ataque** a *true* y así permitimos entrar al último nodo cuya condición es que “Ataque is Set” o lo que es lo mismo **Ataque** sea verdadero.

El segundo nodo es el **FindPlayerLocation**. Se encarga de que el zombi se mueva hasta la posición del *player* y calcula el *pathfinding* para poder así evitar obstáculos.

Al último nodo solo entraremos si estamos a la distancia necesaria para realizar un ataque y por lo tanto atacaremos.

5.3.2.5 Blueprint Partida

Este *Blueprint* controla aspectos relevantes de la partida, como por ejemplo las zonas donde pueden o no aparecer enemigos, los enemigos que quedan para pasar de ronda, la ronda en la que nos encontramos, el *ranking*, etc.

Este *Blueprint* necesita de otros *Blueprints* para su funcionamiento, por tanto no se va a explicar como los anteriores sino que **se va a dividir en cuatro apartados: uno para las variables y los otros tres para las funcionalidades de este Blueprint, pero no estarán compuestas por una única función sino por varias, incluso de otros Blueprints.**

5.3.2.5.1 Variables

Partida Blueprint Variables		
Nombre	Tipo	Descripción
RondaActual	Int	Indica la ronda actual en la que nos encontramos.
ZombisPorRonda	Int	Número de zombis que aparecen en cada ronda.
SpawnsDisponibles	Array <i>AISpawn</i>	Es una lista con todos los actores de tipo <i>AISpawn</i> . Estos actores se usan para saber el lugar exacto en el que puede aparecer un zombi.
ZombisVivos	Int	Es el número de zombis que quedan vivos en la ronda actual. Cuando este número es 0 se pasa a la siguiente ronda.
HandlerSpawnZombies	<i>Timmer Handler</i>	Es el manejador para poder pausar la función de <i>respawn</i> de zombis que se llama automáticamente cuando pasamos de ronda.
ZombiesRondaActual	Int	Indica la cantidad de zombis que tienen que aparecer en la ronda actual.
FactorAumentoZombies	Float	Es el factor mediante el cual aumenta el número de zombis en cada ronda.
PrimeraZona	<i>Boolean</i>	Esta variable booleana empieza inicializada a <i>true</i> , y seguirá así hasta que entremos a una zona por primera vez.
ZombisRondaAnterior	Int	La cantidad de zombis que hubo en la ronda anterior.

VidaZombi	<i>Float</i>	Es la vida de los zombis en esta ronda. Esta vida va incrementando en cada ronda.
------------------	--------------	--

Figura 154: Tabla variables Partida Blueprint

5.3.2.5.1 *Spawn* enemigos

El *spawn* de los enemigos planteó un problema importante a la hora de diseñarlo, debido a que me costó un poco hallar una solución a que reaparecieran enemigos de manera aleatoria solo en aquellas salas que ya hubiéramos abierto. Es decir: si solo hemos abierto la sala de reuniones y las habitaciones, los enemigos tendrán entonces que aparecer de manera aleatoria en alguna de estas dos salas, además no podrán aparecer en un punto cualquiera pues, si no se controla esta cuestión, podrían surgir en algún lugar inaccesible lo que dejaría al zombi atascado sin poder moverse, o lo que es peor, sin que podamos matarlo por estar oculto detrás de algo, todo lo cual impediría pasar de ronda. Para resolver este inconveniente he optado por lo siguiente:

He creado una clase **ZonaBP** que representa las distintas zonas del mapa, por lo tanto existe un actor **ZonaBP** en cada habitación del escenario. Este *Blueprint* solo tiene un componente: una caja de colisión del mismo tamaño que la sala a la que pertenece.

A continuación, he concebido otra clase llamada **IASpawn** que representa los puntos exactos en los que un enemigo podrá aparecer, es decir, cada **IASpawn** es un punto en el mapa donde podrá aparecer un enemigo. Este *Blueprint* solo contiene un componente, que es una esfera de colisión necesaria para que se detecte colisión entre este actor y el actor de tipo **ZonaBP** en el que se encuentra.

El *Blueprint* de **IASpawn**, por tanto, no contiene ni funciones, ni variables, solo la esfera de colisión. Su finalidad es que pueda ser arrastrado en el editor a los puntos exactos donde queremos que aparezcan enemigos y luego la ZonaBP se encargará de detectar cuáles son estos puntos, como veremos ahora.

En la siguiente imagen aparece el actor ZonaBP, que es la caja grande de color naranja de la cual solo podemos ver sus bordes, y los actores **IASpawn**, que son las esferas blancas que se encuentran sobre el suelo y que representan los puntos donde aparecerán enemigos.



Figura 155: Ejemplo IASpawn y ZonaBP

Cada zona debe conocer los distintos puntos en los que puede aparecer un enemigo, dicho de otra forma, cada zona es responsable de sus puntos de *spawn*. Para poder añadir estos puntos de *spawn* a las zonas de manera automática, utilizamos el evento ***BeginPlay*** de **ZonaBP**.

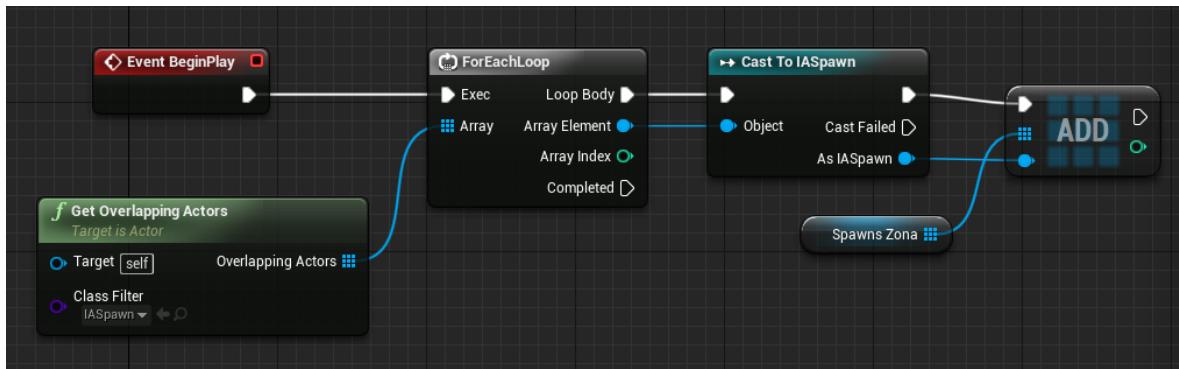


Figura 156: Evento BeginPlay (clase ZonaBP)

Este evento se activa automáticamente cuando comienza el juego para su actor. En el caso de que su actor esté desde el principio de partida, se ejecutará este evento al iniciar el juego. Este evento es el encargado de saber en qué puntos pueden aparecer los zombis dentro de esta zona, para ello usamos el nodo ***GetOverlappingActors***, que te devuelve una lista con todos los actores que están colisionando con nuestra zona. Recorremos esta lista uno a uno para verificar si se trata de un actor de tipo ***IASpawn*** y, en ese caso, lo añadimos a nuestra variable ***SpawnsZona*** donde almacenaremos todos los ***IASpawn*** de una zona, que serán por tanto todos los puntos donde un zombi podrá aparecer en esa zona.

Ahora ya disponemos de los puntos en los cuales pueden aparecer los zombis dentro de cada zona, pero aún ignoramos en cuáles de esas zonas ha estado el jugador, o sea, tenemos todos los puntos de cada zona pero no controlamos todavía que no aparezcan zombis en otros espacios distintos de aquellos a los que el jugador ha accedido. Para obtener este dato lo que se ha hecho es que los puntos donde pueden aparecer los zombis no sean los que se encuentran almacenados en la variable **SpawnsZona** de cada **ZonaBP**, sino que los zombis solo puedan aparecer en aquellos puntos (**IASpawn**) que se encuentran dentro de la variable **SpawnsDisponibles** de la clase **Partida**.

Por lo tanto, ahora hemos de conseguir que los **IASpawn** que estén en **SpawnsDisponibles** sean aquellos que realmente están disponibles, esto es, sean aquellos a los que se ha accedido a su zona. Para alcanzar este objetivo, necesitamos conocer en qué momento accedemos a una nueva habitación y, por lo tanto, activamos una nueva zona. Este dato lo sabemos gracias al componente que tenían los **ZonaBP**, que era una caja de colisión, y, como esta está ajustada a las habitaciones a las que corresponde una zona, cuando abramos una puerta para acceder a otra sala, se detectará colisión con esta caja de colisión y se ejecutará el evento **ActorBeginOverlap**.

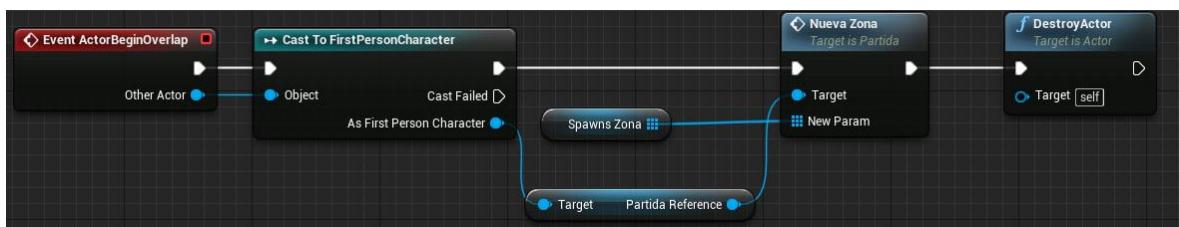


Figura 157: Evento *BeginOverlap* (clase *ZonaBP*)

Al ejecutar este evento nos aseguramos de que la colisión que detectó la caja se produjo con el jugador y no con otro actor, para lo cual usamos un *casteo*. Si el actor que colisionó era el *player* (clase **FirstPersonCharacter**), llamamos a la función **NuevaZona** de la clase **Partida** y le pasamos por parámetro la lista de **IASpawns** de la zona que se almacenaba en la variable **SpawnsZona**.

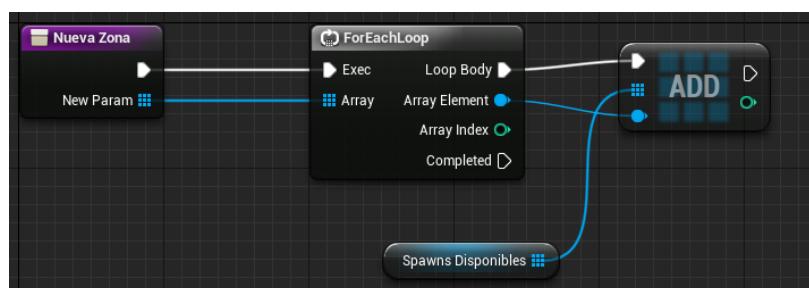


Figura 158: Función *NuevaZona* (clase *Partida*)

En esta función simplemente añadimos los **IASpawn** de la zona a la que acabamos de acceder a la lista de *spawns* disponibles almacenados en la variable **SpawnsDisponibles**, puesto que esa zona ya la hemos activado, por lo tanto los zombis podrán aparecer ahora en los puntos de esa zona. De esta

manera cada vez que visitamos una nueva zona, incorporamos los puntos de *spawn* de esa zona a la lista de puntos de *spawns* disponibles, en la que se encuentran todos los puntos de *spawn* de las zonas activas, así evitamos que los zombis aparezcan en puntos de *spawn* pertenecientes a zonas que no hemos visitado.

Llegados a este punto ya sabemos dónde pueden aparecer los zombis y controlamos que no aparezcan en zonas que no hemos visitado, así que únicamente falta hacer que aparezcan, misión que cumple la función ***SpawnZombie*** de la clase **Partida**.

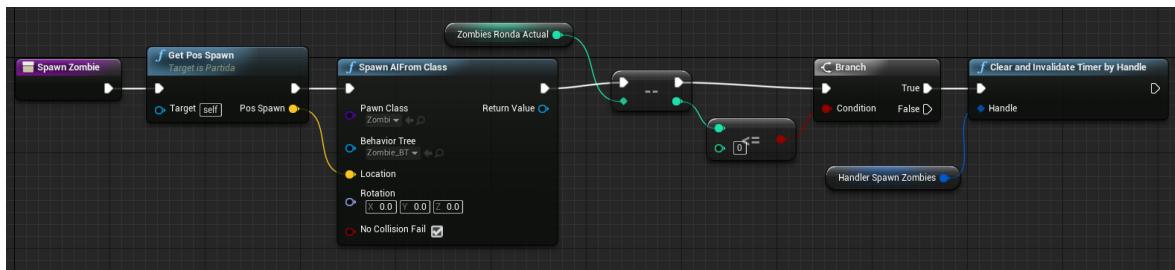


Figura 159: Función *SpawnZombie* (clase *Partida*)

La primera tarea de esta función ***GetPosSpawn*** es elegir un punto, de entre todos los puntos de *spawn* disponibles, seleccionarlo aleatoriamente entre todos los posibles y devolver su posición exacta en el mundo, que es lo que realmente nos interesa para que el zombi aparezca en ese punto.

Además esta función también se encarga de controlar si ya han aparecido todos los zombis que debían hacerlo en esta ronda. Si esto es así, desactivará la llamada automática a esta función (activada en el comienzo de ronda) utilizando el **TimerHandler** de esta función almacenado en la variable **HandleSpawnZombies**.

5.3.2.5.1 Comienzo de nueva ronda

Una nueva ronda comienza cuando ya hemos matado a todos los zombis que aparecieron durante la ronda actual. Para controlar esto cada vez que un zombi muere llama al evento ***ZombieMuerto*** de la clase **Partida**.

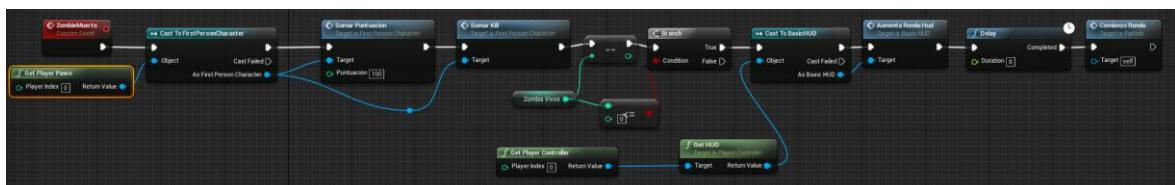


Figura 160: Evento *ZombieMuerto* clase *Partida*

Cuando un zombi muere se ejecuta este evento, el cual se encarga de sumar la puntuación de la muerte al jugador y sumar la muerte al marcador. Además se encarga de comprobar que si era el último zombi vivo (el control de los zombis vivos lo tiene la clase **Partida**) entonces cambiaríamos de ronda, llamando a la función del **HUD** encargada de reproducir la animación del cambio de ronda y llamando a la función **ComienzoRonda** que contiene la lógica del cambio de ronda. A esta última función se llama después de esperar 8 segundos para dar un tiempo de descanso entre ronda y ronda.



Figura 161: Función ComienzoRonda clase Partida

Esta función se encarga de aumentar en 1 la ronda actual, obtiene el número de zombis que habrán en esta ronda llamando a la función **GetNumZombiesRonda** y con el resultado devuelvo por esta función establecemos las variables **ZomBiesRondaActual**, **ZombisVivos** y **ZombisPorRonda**. Por ultimo calculamos la vida que tendrá el zombi en esta nueva ronda y usamos el nodo **SetTimmerByFuncitionName** que como ya vimos activaba la llamada automática a una función cada cierto tiempo, en este caso a la función es la de **SpawnZombie** y el tiempo es de 0.5 segundos consiguiendo así que cada 0.5 segundos aparezca un nuevo zombi en un punto aleatorio del mapa.

5.3.2.5.1 Ranking

Cuando morimos comprobamos si tenemos los puntos necesarios o no mediante la función **EnRanking?**. Si hemos conseguido entrar en el ranking entonces se llamaría a la función **ActualizaRanking** que hará lo siguiente:

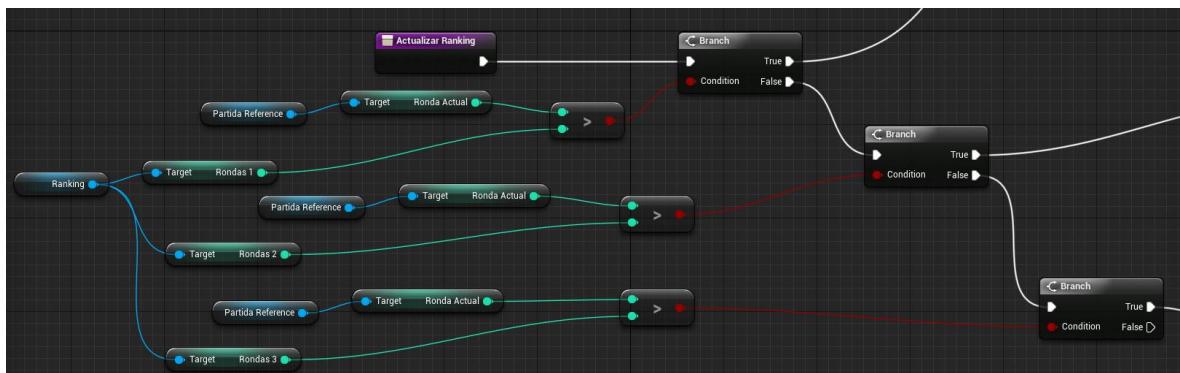


Figura 162: Función ActualizaRanking (parte 1/4)

Lo primero que se hace es comprobar en qué puesto del ranking hemos entrado, para ello comprobamos nuestro número de rondas con el número de rondas de los 3 jugadores que estaban en el ranking. Puede haber 3 casos:

- **Somos los primeros:** tenemos entonces que pasar el jugador del ranking que ocupaba la primera posición a la segunda, y el de la segunda posición a la tercera. Finalmente poner al jugador actual como primero, esto se hace de la siguiente forma:

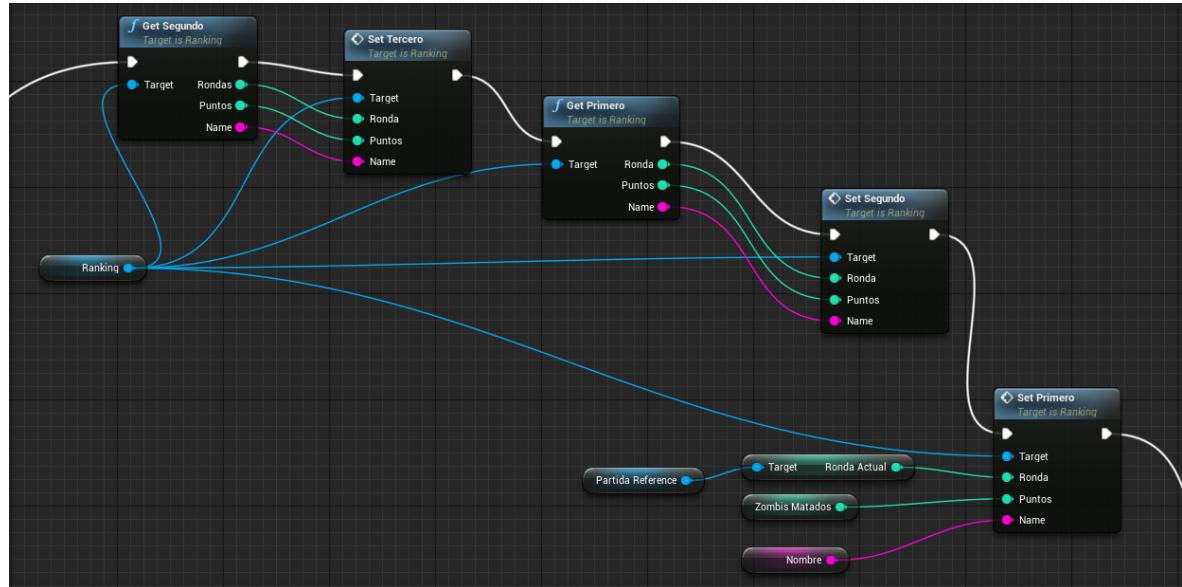


Figura 163 Función ActualizaRanking (parte2/4) somos los primeros

- **Somos los segundos:** en este caso tendríamos que pasar el jugador que era el numero 2 al número 3 y poner al nuevo jugador como segundo.

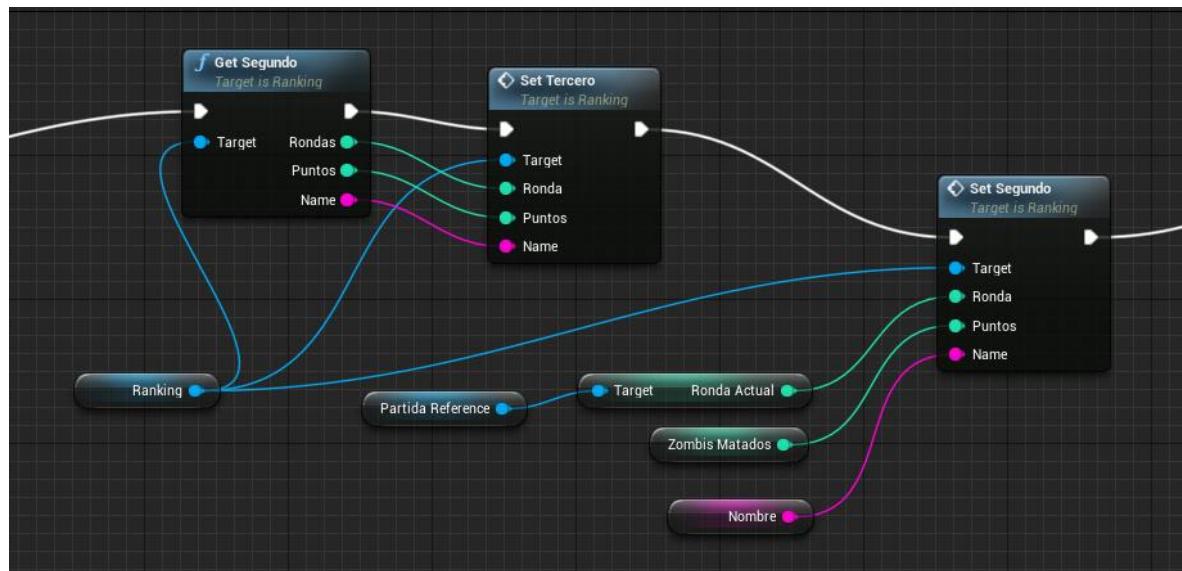


Figura 164: Función ActualizaRanking (parte3/4) somos segundos.

- **Somos terceros:** este es el caso más fácil en el que únicamente tendríamos que sustituir los datos del tercer jugador por los datos del jugador actual.

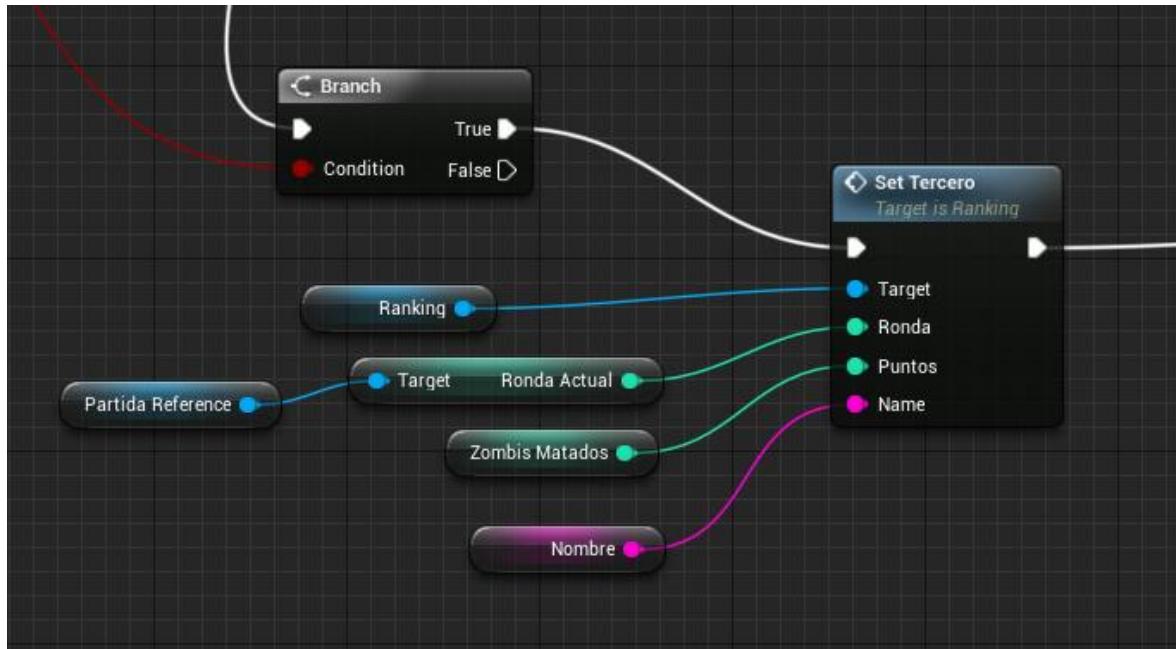


Figura 165: Función ActualizaRanking (parte 3/4) somos terceros.

Finalmente desde cualquiera de estos 3 resultados terminaríamos guardando estos resultados para que perduren entre partidas, aunque salgamos del juego o apaguemos el equipo.

5.3.3 Desarrollo Menús y HUD

Para la creación de interfaces de usuario en Unreal Engine 4 hemos utilizado los **UMG (Unreal Motion Graphics)**, que se basan en la utilización de **Widgets**. Los **Widgets** son una serie de funciones predefinidas (como botones, check box, barras de progreso, etc.) que son usadas para la creación de nuestras interfaces.

Los **Widgets** son editados en un tipo especial de Blueprint llamado **Widget Blueprint**, el cual contiene 2 pestañas:

- **Designer:** esta es la pestaña para diseñar, se tratará de una capa visual de la interfaz. En esta capa podemos añadir los elementos nombrados antes como un botón, personalizar estos elementos y dotarlos de animaciones.
- **Graph:** esta pestaña es a la que estamos acostumbrados ya en el resto de Blueprints, gracias a esta pestaña podemos dotar de lógica a los elementos utilizados en la interfaz, por ejemplo para que cuando pulses el botón de inicio empiece el juego.

5.3.3.1 Menús

En este apartado vamos a hablar de aquellas cosas que me parezcan importante comentar sobre alguna funcionalidad de los menús. Por lo tanto de algunos menús como el menú de opciones, créditos o la pantalla de muerte no se explicaran pues su funcionamiento es sencillo. Otros menús como por ejemplo el menú de inicio se explicara solo una pequeña parte para hacernos una idea de cómo esta implementado el resto pues todo está hecho de manera similar.

5.3.3.1.1 Menú Principal

Nuestro menú principal tendría el siguiente aspecto (sin usar la lógica de la pestaña grafo):

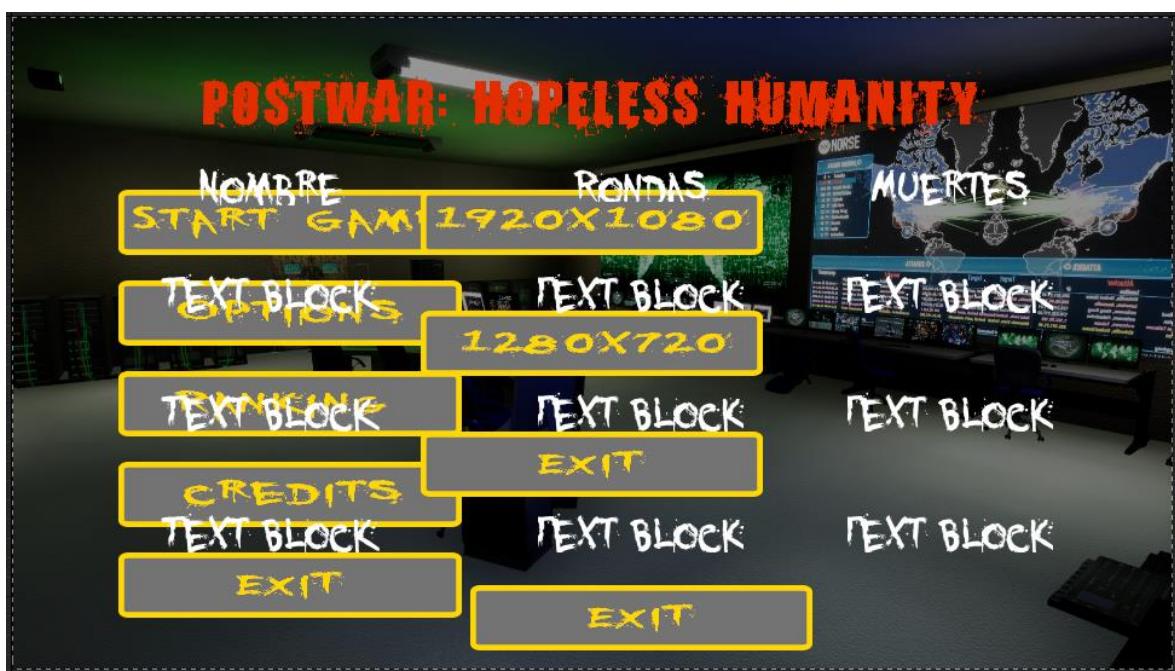


Figura 166: Menú principal (sin ocultar nada)

Como podemos observar este menú tiene en su interfaz toda la información a la que podamos acceder, ya sea por ejemplo el menú de opciones o el ranking. Gracias a la pestaña **Graph** podemos modificar los elementos de la interfaz mediante nodos gráficos como hacíamos en los Blueprints. De esta manera el menú que veríamos al entrar al juego seria el siguiente:



Figura 167: Menú principal que vemos en el juego

A estos botones se le puede añadir efectos, por ejemplo en nuestro caso cuando pasamos por encima de un botón se pondría de color rojo.

Cada botón del menú es vinculado a un evento al cual llama cuando es pulsado. Por ejemplo, si pulsamos el botón “Ranking” entonces tendríamos que ocultar estos botones que vemos en la foto de arriba y mostrar únicamente el ranking. Esto se hace de la siguiente forma:

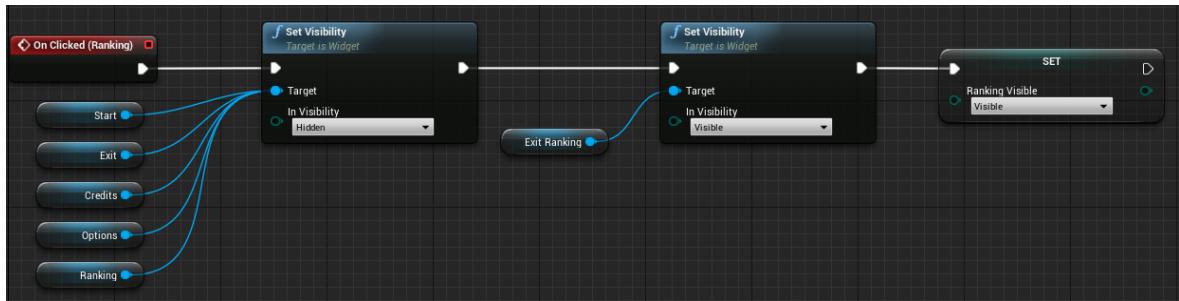


Figura 168: Evento llamado al pulsar el botón ranking desde el menú principal

El resultado de ejecutar ese evento sería el siguiente:



Figura 169: Menú ranking

En el juego sustituiríamos los “Text Block” por los nombres, rondas y muertes de los jugadores que estén en el ranking. Esto se haría leyendo del fichero donde se almacena esta información.

5.3.3.1.2 Pantalla Ranking

Esta es la pantalla que aparece cuando morimos y hemos conseguido los puntos necesarios para entrar al ranking.

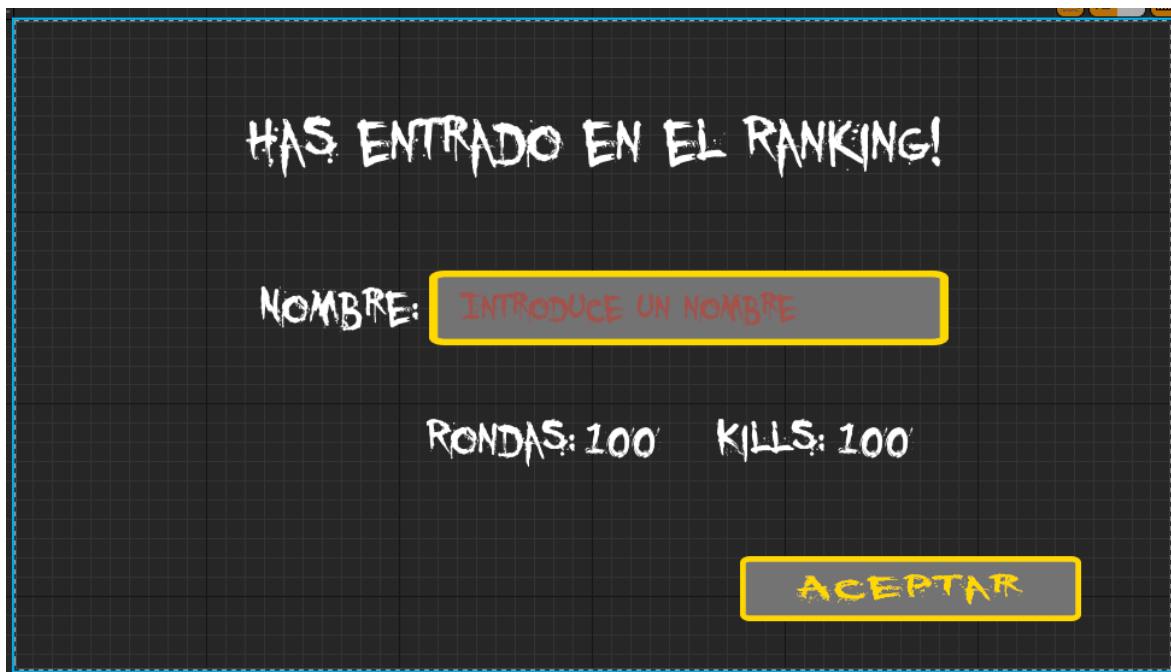


Figura 170: Pantalla ranking final de juego (desde editor)

Como podemos observar en esta pantalla no tenemos una imagen de fondo como teníamos en el menú principal, esto se ha hecho así porque queríamos que se viera el juego de fondo mientras aparece esta pantalla. El resultado de esta pantalla en el juego se puede ver en el apartado “Interfaz” del GDD (exactamente el apartado 5.2.5.1.3.2 Con Ranking) o al final del video de gameplay (enlace en el apartado conclusiones).

Para podernos guardar la información introducida en el cuadro de texto hemos creado un evento que se encarga de guardar este nombre del introducido cuando se pulsa la tecla enter o cuando se pierde el focus de este text box, es decir, cuando clicamos fuera de él.

Finalmente cuando pulsamos el botón aceptar se llamaría a la función **ActualizaRanking** que vimos anteriormente (véase apartado 5.3.2.5.1 Ranking) y cargaríamos de nuevo el menú principal.

5.3.3.2 HUD

Para el HUD hemos creado un Blueprint llamado HUD el cual contiene un widget llamado BasicAmmo.

5.3.3.2.1 HUD Blueprint

Lo primero que haría el Blueprint HUD sería crear el widget, añadirlo a la pantalla para que se pueda visualizar y guardarnos una referencia de este para poder acceder a sus funciones y variables. Podemos ver esto en la imagen inferior:

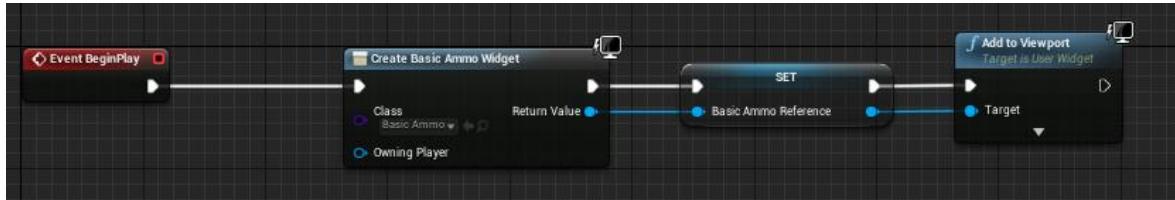


Figura 171: Creación del Widget del HUD

Como este Widget se ha creado dentro de la clase HUD solo él tiene acceso a este. Se podría hacer que el player tuviera una referencia de este Widget pero no se ha hecho así, para intentar separar el WidgetBlueprint del resto de Blueprints normales, por lo tanto el jugador (el **FirstPersonCharacter**) solo tendría acceso a la clase HUD y sus funciones pero no a las funciones de BasicAmmo. De esta forma se han creado una serie de funciones que hacen de “intermediario” entre la interfaz y el jugador para que estas se puedan comunicar teniendo separada la lógica del juego de la lógica de la interfaz.

Las variables de este Blueprint son:

HUD Variables		
Nombre	Tipo	Descripción
PrecioPagado	Text	Es una variable de tipo texto (para que pueda ser imprimida por pantalla) con el precio que hemos pagado por algo. A esta variable se le pone un valor cuando se llama a las funciones RestaPuntos o SumaPuntos desde el juego.
Verde	Booleano	Esta variable booleana indica el color que tiene que tener la variable PrecioPagado cuando la imprimimos por pantalla. De manera que si es verdadera será de color verde y si es falsa de color rojo.
BasicAmmoReference	BasicAmmo	Es una referencia al Widget BasicAmmo.

NumRondaHud	Int	Es el número de ronda que se muestra en el HUD, es necesario para hacer la transición de una ronda a otra y que no cambie el numero automáticamente.
InformacionCompra	Text	Es la variable de tipo texto que almacena la información de compra que se envía desde el juego para que se muestre por pantalla.

Figura 172: Tabla variables HUD Blueprint

Las funciones de este Blueprint son las siguientes:

HUD Funciones	
Nombre	Descripción
BeginPlay	Crea el widget y lo añade al viewport para que se pueda visualizar.
RestaPuntos	Recibe por parámetro el precio que hay que restar, lo convierte en texto y le añade el signo “-“ delante. Establece la variable verde a falso para que este precio se dibuje en rojo y luego llama a la función AnimPuntos que se encarga de reproducir la animación de los puntos. Después de 2 segundos se borra para que deje de aparecer en pantalla.
SumaPuntos	Recibe por parámetro el precio que hay que sumar, lo convierte en texto y le añade el signo “+“ delante. Establece la variable verde a verdadero para que este precio se dibuje en verde y luego llama a la función AnimPuntos que se encarga de reproducir la animación de los puntos. Después de 2 segundos se borra para que deje de aparecer en pantalla.
PlayAnimationRecarga	Llama a la función AnimRecarga del Widget BasicAmmo que se encarga de reproducir la animación del texto que te indica que estas recargando.
AumentaRondaHUD	Llama a la función AnimNumRonda del Widget BasicAmmo , esta animación se hace con el número de ronda que acabamos de superar, después de 4 segundos aumentamos el número de ronda y volvemos a llamar a AnimNumRonda .
PlayAnimationReviviendo	Llama a la función AnimReviviendo para que reproduzca la animación del texto cuando estamos reviviendo.

ActualizarInfoCompra	Esta función recibe 2 parámetros, un string que es la información y un int que es el precio. Con estas 2 variables crea un único string del tipo “Info Precio €” donde Info y Precio son los valores que recibimos por parámetro.
BorrarInfoCompra	Se encarga de borrar la información de compra que se estaba mostrando por pantalla.

Figura 173: Tabla funciones HUD Blueprint

5.3.3.2.2 Widget (BasicAmmo)

El widget creado llamado **BasicAmmo** tendría el siguiente aspecto (mostrando todos sus elementos):



Figura 174: Widget HUD

Esta sería la vista desde la pestaña gráfica del widget, recordamos que también tenemos una pestaña Graph que actúa como Blueprint para este widget. En esta pestaña Graph podremos “bindear” los elementos de este widget a valores del jugador como puede ser la vida, munición, etc. Es decir, lo que hacemos es vincular el valor de estos elementos de la interfaz con otros valores usados en otros Blueprints de manera que si estos cambian también cambiarán los valores de la interfaz.

Todos los elementos de este Widget por tanto exceptuando aquellos que son estáticos (como por ejemplo la imagen de la bala) están vinculados a valores del juego. Como la manera en la que se hacen estos “bindeos” es muy parecida para todos solo vamos a mostrar un ejemplo que va a ser el de la pantalla roja dependiendo de la vida del jugador. La forma de poder hacer estos “bindeos” es

mediante funciones. A continuación explicaremos solo uno para que sirva de ejemplo y mostraremos una tabla con el resto.

La imagen de abajo es la función **SangreHUD** que se encarga de dibujar esta sangre por pantalla cuando el jugador está dañado, además se hace de manera progresiva, es decir, cuanto más dañado estas mayor será la cantidad de sangre en la pantalla.

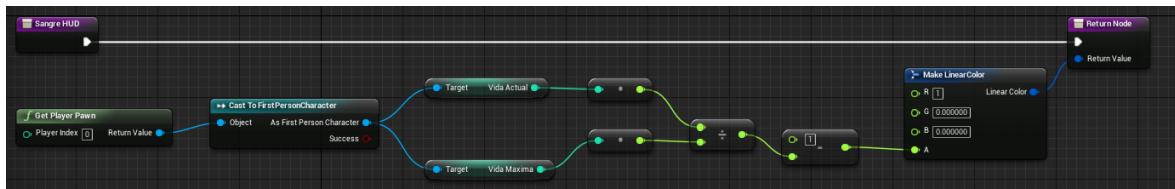


Figura 175: Función encargada de actualizar la sangre de la pantalla respecto a la vida del jugador

En primer lugar obtenemos el player a partir del nodo **GetPlayerPawn** y lo convertimos en el **FirstPersonCharacter** que necesitamos para poder acceder a sus variables. Obtenemos la vida actual del player y la vida máxima, estos 2 valores son enteros y nos interesa sacar un porcentaje por lo tanto tenemos que convertir ambos valores a float. Una vez convertidos dividimos la vida actual entre la vida máxima dándonos como resultado un valor entre 0 o 1. El valor resultante de esta división se le resta a uno y este nuevo resultado será la opacidad de esta imagen que representa la sangre en la pantalla. De esta forma si la vida actual es 10 y la vida máxima es 10 entonces $10/10 = 1$ pero como $1-1=0$ tendríamos que la opacidad final es de 0 que es justo lo que necesitamos porque si estamos con la vida completa esta imagen tiene que ser totalmente invisible. De la misma manera si tenemos 5 puntos de vida la opacidad será de 0.5.

Una vez que ya hemos visto un ejemplo de “bindeo” de un valor de la interfaz con un valor del juego vamos a explicar en una tabla el resto de funciones que utilizamos para “bindear” el resto de elementos de la interfaz.

En esta tabla no vamos a tener las columnas comunes de “Nombre” y “Descripción” sino que tendrá las siguientes columnas:

- Nombre: nombre de la función.
- Item: elemento al que representa en la pantalla.
- Bind: variable con la cual está “bindeado” (vinculado) este elemento.

Nombre	Item	Bind
MunicionActual	Munición actual del jugador en el cargador del arma.	MunicionCargador de la clase FirstPersonCharacter .
MunicionTotal	Munición total disponible para recargar el arma.	MunicionCargador de la clase FirstPersonCharacter .
Recarando	Texto que aparece cuando estamos recargando.	Recargando de la clase FirstPersonCharacter .
Puntuacion	Dinero del jugador con el que podemos comprar armas y mejoras.	Puntuación de la clase FirstPersonCharacter .
DibujarPuntosGastados	La cantidad de dinero ganado o gastado que se dibuja debajo del dinero total.	PrecioPagado de la clase HUD .
Reviviendo	Texto que aparece en pantalla cuando estamos reviviendo.	Reviviendo de la clase FirstPersonCharacter .
SangreHUD	Imagen de sangre que aparece cuando estamos dañados.	Calculado mediante la vida actual y la vida máxima del jugador.
ColorPrecioPagado	Color con el cual se dibuja el valor de la función DibujarPuntosGastados .	Verde de la clase HUD .
KillsTotales	Numero de zombis que hemos matado.	ZombisMatados de la clase FirstPersonCharacter .
PorcentajeBarra	Es la barra de progreso que disminuye según la cantidad de zombis que quedan vivos.	Calculada mediante los zombis vivos y los zombis totales de la ronda.
AumentaNumRondaHUD	Numero de ronda	NumRondaHUD de la clase HUD .
InfoCompra	Texto con la información de compra que aparece al acercarnos a un objeto que podemos comprar.	InformacionCompra de la clase HUD .
DibujarMira	Es la mirilla con la que se apunta cuando no miramos a través de la mira del rifle.	Apuntando de la clase FirstPersonCharacter .

Figura 176: Tabla funciones BasicAmmo Widget

5.3.4 Arte en el videojuego

En esta sección se va a ver todo lo relacionado con el arte visual del videojuego. Hay que tener en cuenta que el diseño artístico de este proyecto ha sido influido por los juegos que se usaron de referencia, intentando alcanzar siempre un resultado de calidad parecido.

El diseño gráfico del videojuego ha sido un gran reto para mí y una de las partes más complicadas del proyecto.

En primer lugar porque no soy artista y el objetivo de conseguir un resultado realista con una cantidad de polígonos no muy elevada en los polígonos ha sido todo un reto a la hora de modelar.

Por otro lado el objetivo ha sido siempre el de un escenario realista, pero que además sea vistoso y en el cual el jugador se sienta cómodo para poder jugar, pero sin intentar alejarnos mucho de la realidad ni la historia del juego el cual transcurre en una base militar. Debido a esto se han intentado recrear algunas salas que podríamos encontrar en una base militar, estas salas son:

- Habitaciones
- Comedor
- Vestuario
- Enfermería
- Sala de control
- Sala de reunión
- Exterior
- Sala inicial
- Sala donde se mejoran las armas

Es importante destacar que no todos los elementos que se han utilizado en este proyecto son de elaboración propia. Ocurre lo mismo con las texturas y materiales, algunos de ellos han sido elaboradas por mí pero otros no.

5.3.4.1 Gráficos 2D

Para la realización de gráficos 2D se ha utilizado Photoshop. Entre los gráficos 2D se encuentran los botones para los menús y las imágenes que aparecen tanto en los power-ups como en las máquinas de mejorar las armas. Algunas de estas imágenes se pueden ver a continuación:

5.3.4.1.1 Botones



Figura 177: Botón aceptar



Figura 178: Botón aceptar con hover



Figura 179: Botón empezar partida



Figura 180: Botón empezar partida hover

5.3.4.1.2 Símbolos máquinas de mejora

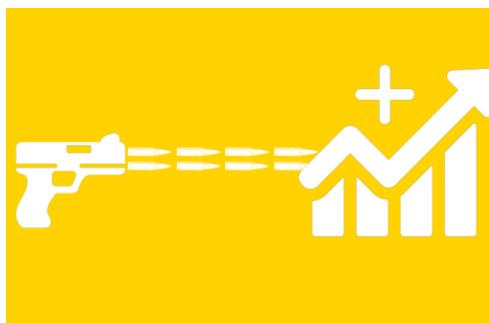


Figura 181: Símbolo mejorar cadencia

Este símbolo es una pistola que dispara 2 balas en vez de una intentando representar así un mayor aumento de cadencia.



Figura 182: Símbolo mejorar daño

Este símbolo indica un aumento en el daño de las balas, representado mediante las propias balas y un hombre fuerte al lado de una gráfica indicando una mejora de esta estadística. Además el color rojo ha sido elegido puesto que el daño o fuerza es comúnmente representado con este color e inconscientemente muchos jugadores simplemente con el color ya lo asocian al daño porque ya están acostumbrados.

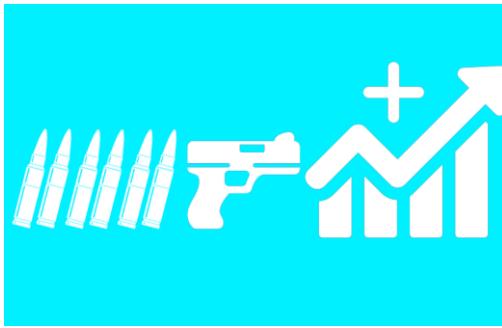


Figura 183: Símbolo mejorar munición arma

En esta imagen se puede ver una gran cantidad de balas al lado de un arma, indicando así un aumento de la capacidad de balas del arma.

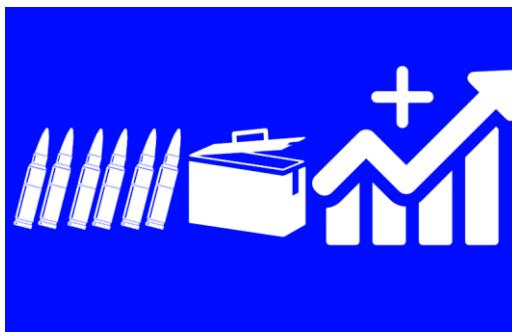


Figura 184: Símbolo mejorar munición máxima

En este símbolo podemos ver una caja de munición y balas a su lado, indicando así un aumento de la capacidad de munición máxima (la munición disponible para recarga).



Figura 185: Símbolo mejorar tiempo recarga

En este símbolo observamos una mano que simula estar moviéndose rápido, unas balas y una caja de munición. Intenta reflejar un aumento en la velocidad de recarga obteniendo balas más rápido de la caja de munición.

5.3.4.1.3 Símbolos power-ups

Estos símbolos están situados en el centro de la imagen y con bastante margen puesto que se aplican a un objeto con zona esférica y no es un objeto plano.

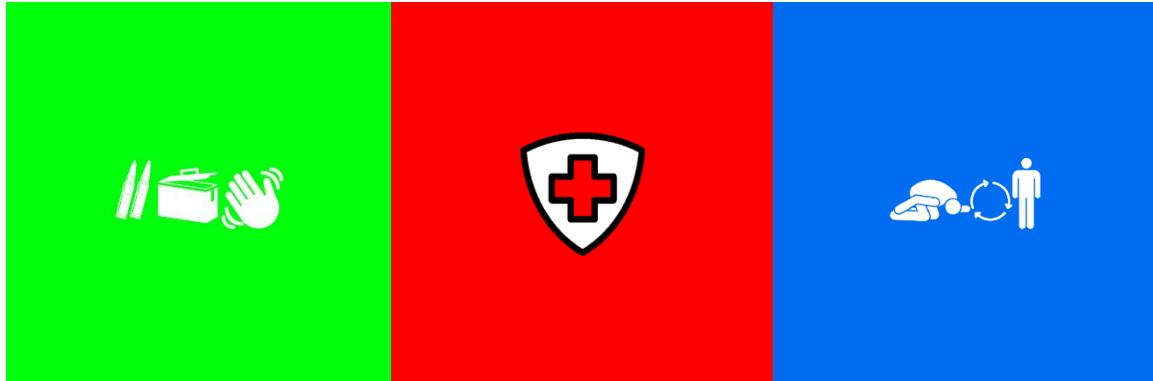


Figura 186: Símbolos power-up

Estos símbolos de izquierda a derecha serían mejorar tiempo de recarga base, aumentar resistencia, revivir.

El símbolo de recarga es parecido al que ya vimos en la sección anterior.

El símbolo de resistencia es una especie de escudo con el símbolo de la cruz roja representando un aumento de resistencia.

El símbolo de revivir intenta explicar que una vez que has caído al suelo (has muerto) puedes volver a ponerte de pie (puedes revivir).

Un ejemplo de cómo se aplica a una máquina de power-up sería el siguiente:



Figura 187: Ejemplo power-up titan

5.3.4.2 Modelado 3D

A continuación vamos a ver varios ejemplos de entre los distintos modelos de elaboración propia. Veremos el modelo tanto en 3DSMax como su resultado final en el juego después de aplicarle materiales.

5.3.4.2.1 Tubo de luz

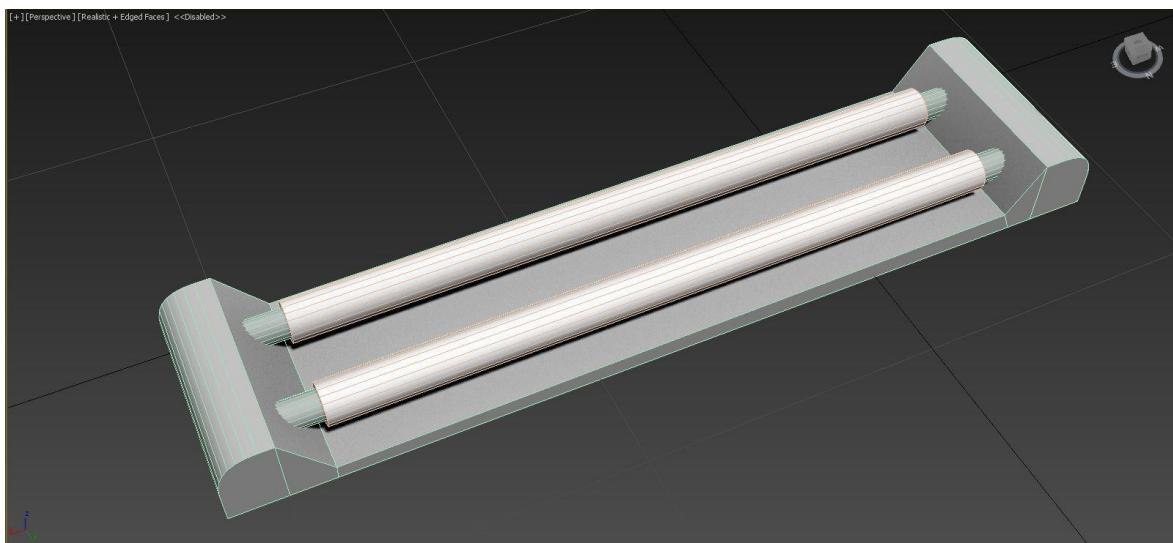


Figura 188: Modelo de luz en 3DSMax



Figura 189: Modelo de luz in game

5.3.4.2.2 Maquina de mejorar arma

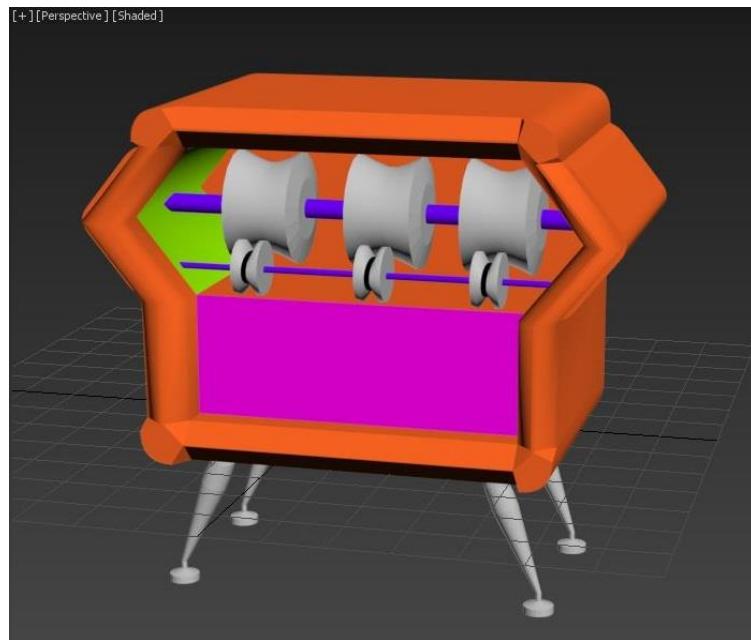


Figura 190: Modelo máquina mejorar arma en 3DSMax



Figura 191: Modelo máquina mejorar arma en Unreal Engine 4

5.3.4.2.3 Monitor de doble pantalla

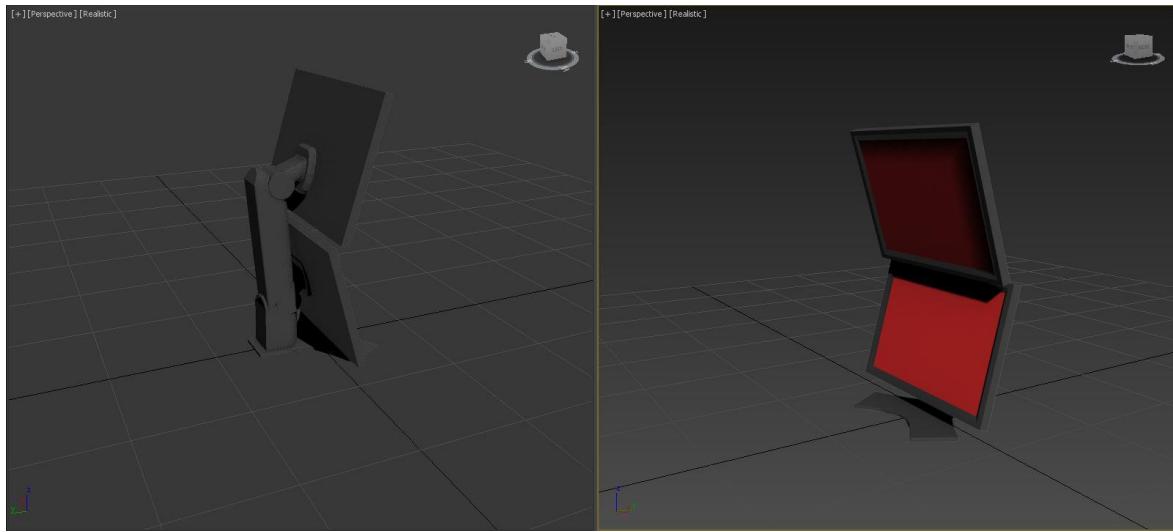


Figura 192: Modelo monitor con doble pantalla en 3DSMax

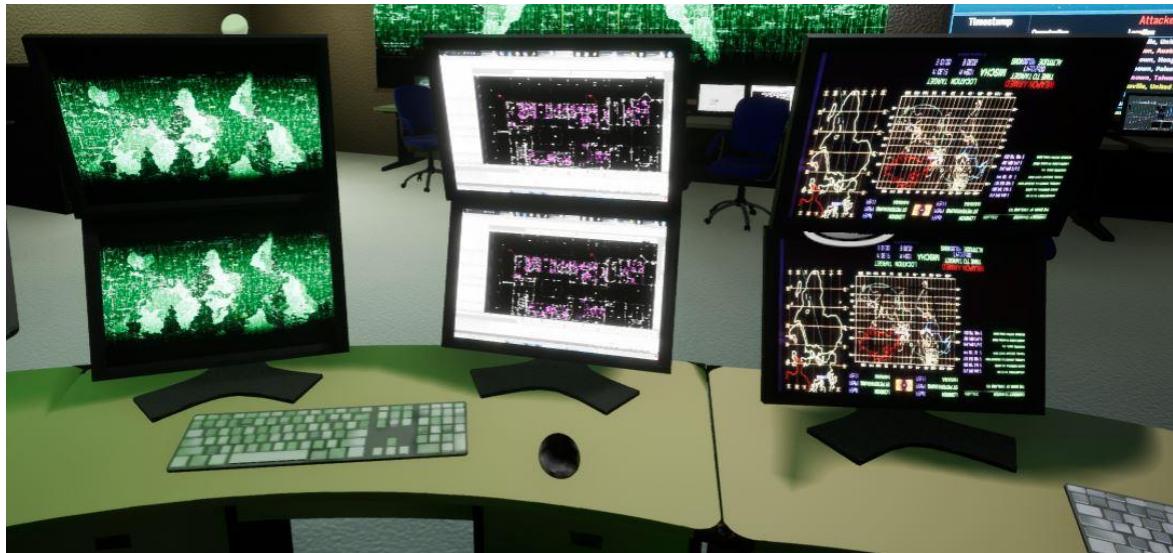


Figura 193: Modelos monitor con doble pantalla in game

5.3.4.3 Optimización modelos 3D

Como ya se dijo al principio del apartado de arte algunos de los modelos utilizados no son de elaboración propia. El problema de usar estos modelos es que muchos de ellos no están preparados para un juego y por lo tanto tienen una gran cantidad de polígonos, es por esto que es necesario optimizarlos.

Prácticamente todos los modelos utilizados que no han sido de elaboración propia han sido optimizados o modificados antes de poder usarse en Unreal. Estas optimizaciones y modificaciones

se han hecho siempre que la licencia lo permita, pues en algunas licencias de modelos prohíben su modificación.

Este trabajo normalmente lleva bastante tiempo, pues muchas veces las herramientas de optimización acaban rompiendo la malla del modelo de manera que tiene que ser reparada manualmente. Además que al optimizar el modelo reduciendo así el número de polígonos hay que volver a hacer los UV.

La optimización de modelos se ha hecho no solo basándose en el número de polígonos del modelo sino también pensando en su impacto en el suelo, de esta manera un objeto que es muy visible y necesita mayor calidad ha sido menos optimizado que otro objeto el cual es pequeño o no se le presta mucha atención como el caso de los cepillos de dientes por ejemplo en los cuales la optimización ha sido mucho mayor a costa perder calidad en el modelo.

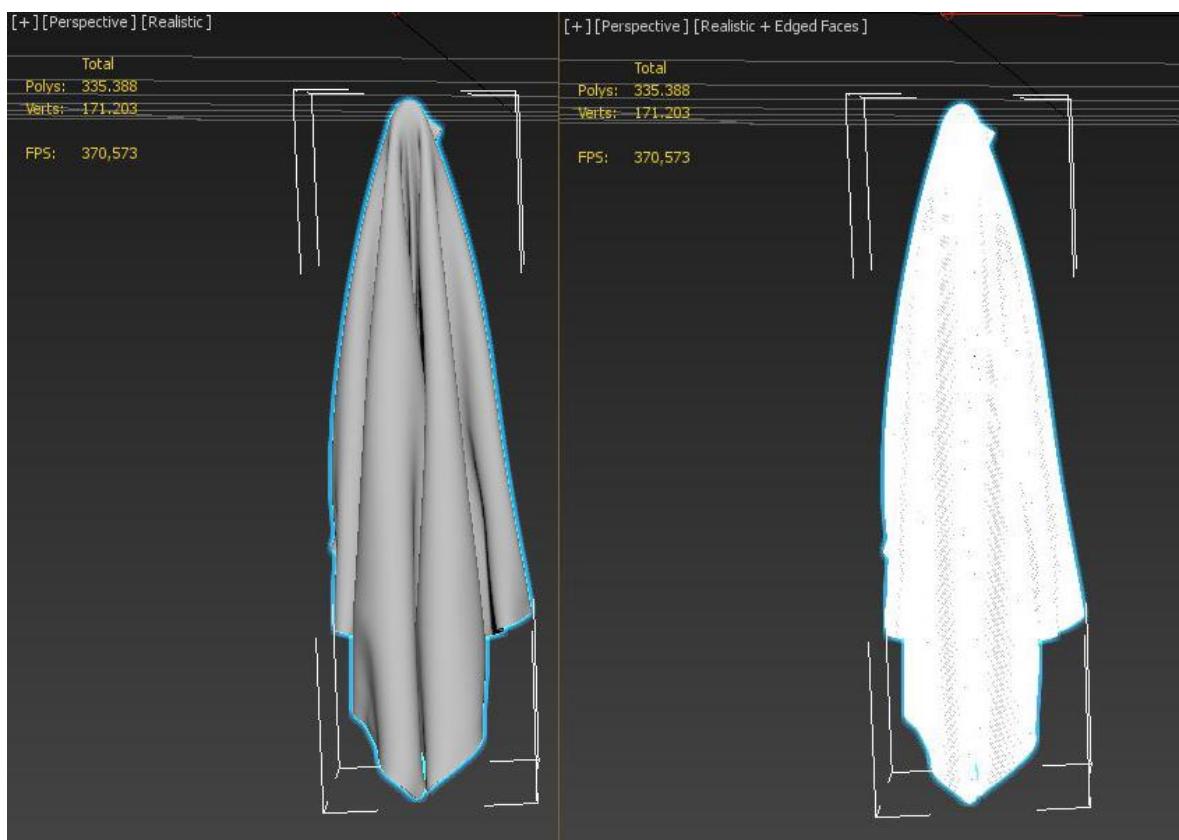


Figura 194: Modelo toalla sin optimizar

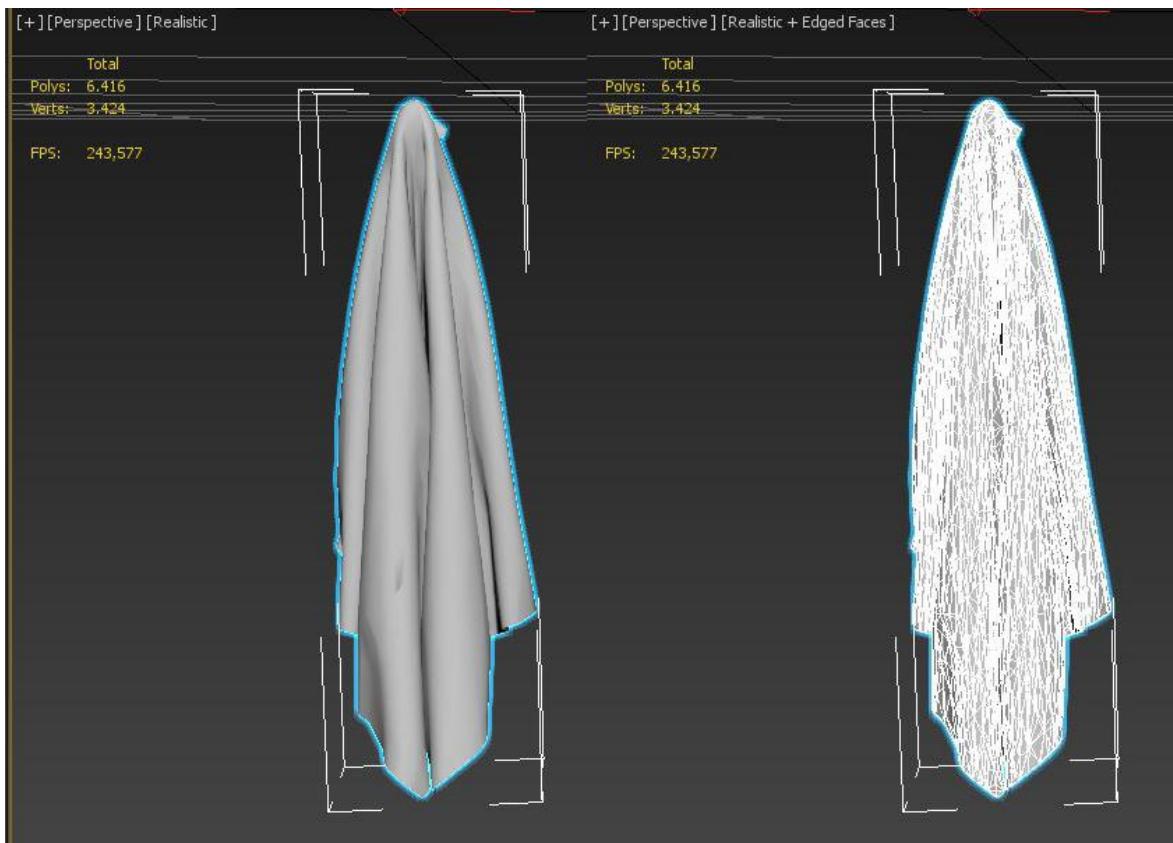


Figura 195: Modelo toalla optimizado

Modelo Toalla	
Número de polígonos inicial	333.388
Número de polígonos final	6.416
Número de vértices inicial	171.203
Número de vértices final	3.424

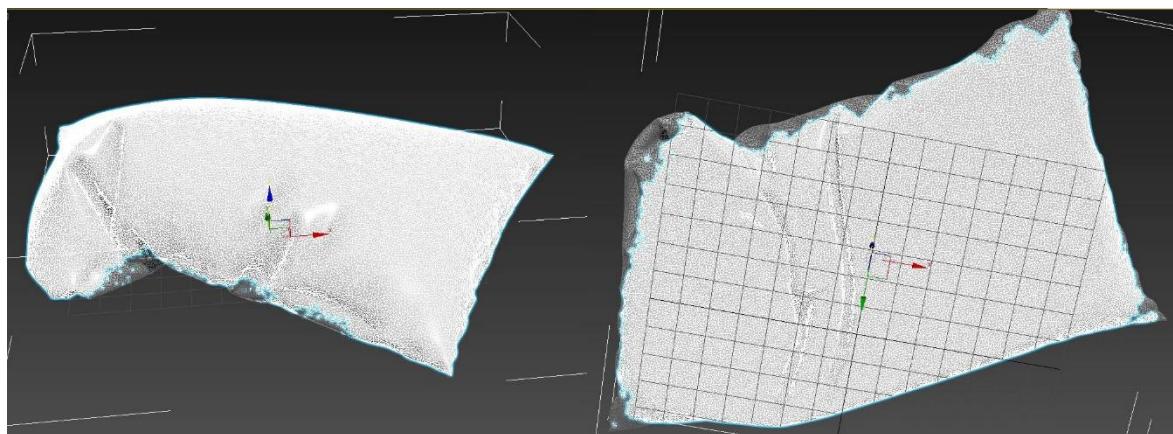


Figura 196: Modelo cojín sin optimizar

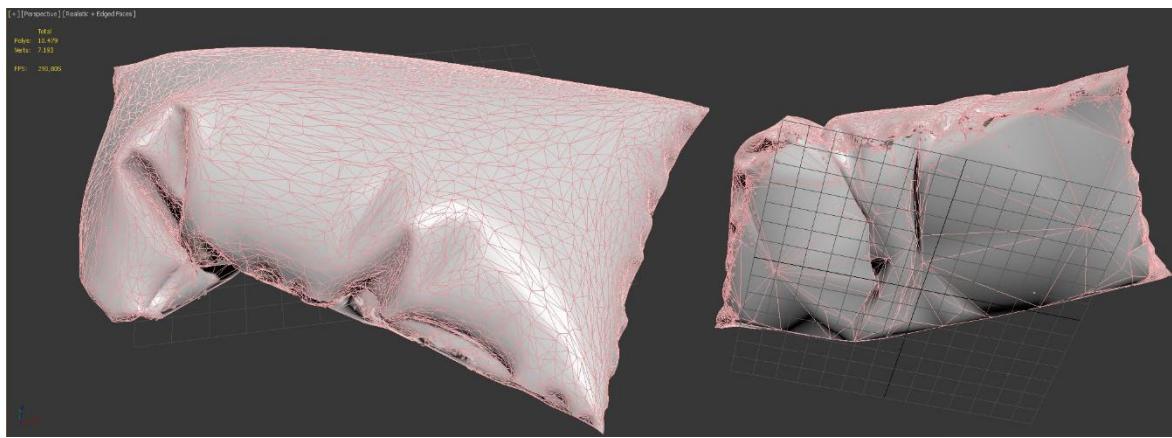


Figura 197: Modelo cojín después de optimizar

Modelo Cojín	
Número de polígonos inicial	111.832
Número de polígonos final	10.479
Número de vértices inicial	60.713
Número de vértices final	7.193

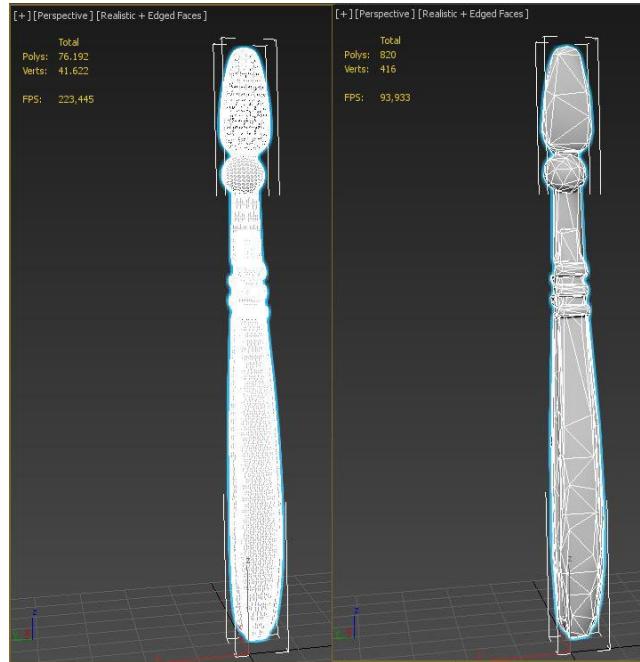


Figura 198: Modelo de cepillo de dientes antes y después de ser optimizado

Modelo Cepillo de dientes	
Número de polígonos inicial	76.192
Número de polígonos final	820

Número de vértices inicial	41.622
Número de vértices final	416

5.3.4.4 Texturas y materiales

Para poder crear los materiales que se han aplicado a los distintos modelos desde el editor de Unreal primero tenemos que tener los mapas de textura de estos materiales. Los mapas de textura más comunes y por tanto más utilizados en el proyecto han sido los siguientes:

- **Mapa difuso:** serían básicamente los colores que tiene el modelo. Es el resultado de aplicar la imagen de textura 2D al modelo en 3D.

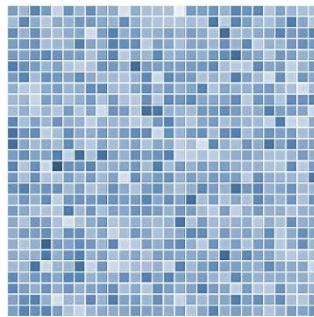


Figura 199: Ejemplo mapa difuso

- **Mapa de normales:** se encargan de simular el relieve de un objeto. Pero no sería un releve real ni produciría sombras, este efecto se consigue gracias a las direcciones de normales que se aplican sobre el modelo. Estas normales son leídas de este mapa de texturas y de ahí viene su nombre.

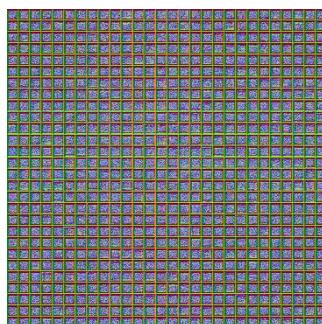


Figura 200: Ejemplo mapa de normales

- **Mapa especular:** se encarga de definir qué partes de un objetos son las que reflejan el brillo y en qué cantidad. Las partes blancas reflejarían el máximo de luz posible mientras que las partes negras no reflejarían luz.

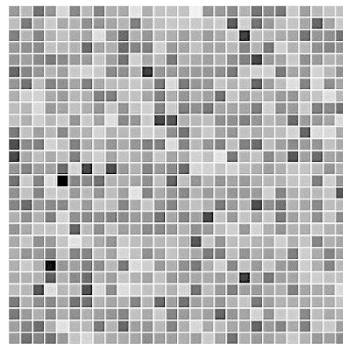


Figura 201: Ejemplo mapa especular

- **Mapa de oclusión:** los mapas de oclusión se utilizan para informar qué partes del modelo deberían recibir alta o baja iluminación.

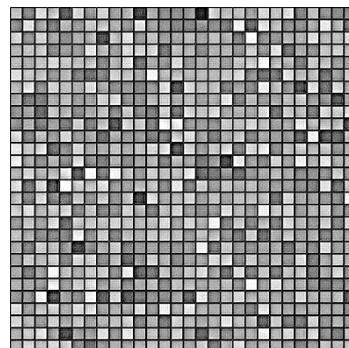


Figura 202: Ejemplo mapa de oclusión

Una vez que tenemos estos mapas de texturas podemos crear un material de la siguiente forma:

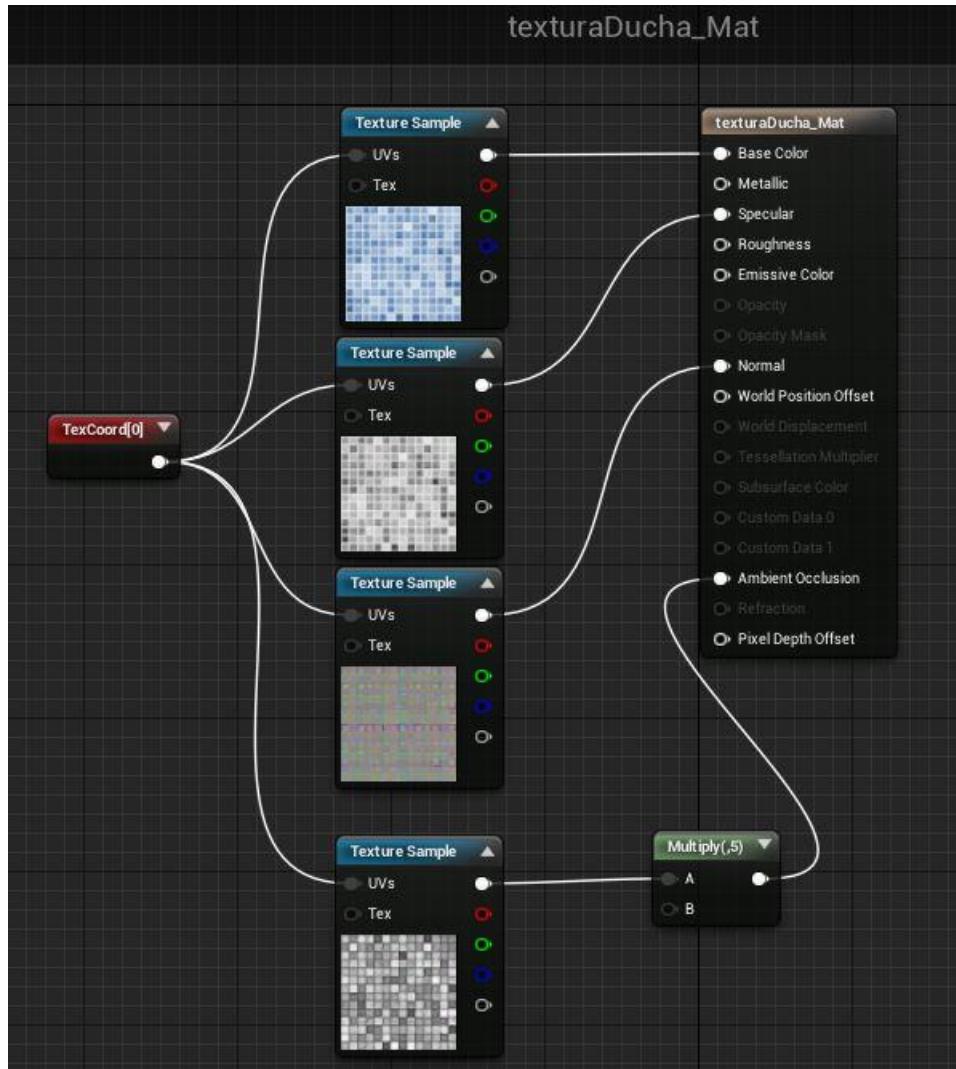


Figura 203: Material suelo de ducha

Su aplicación en el juego quedaría así:

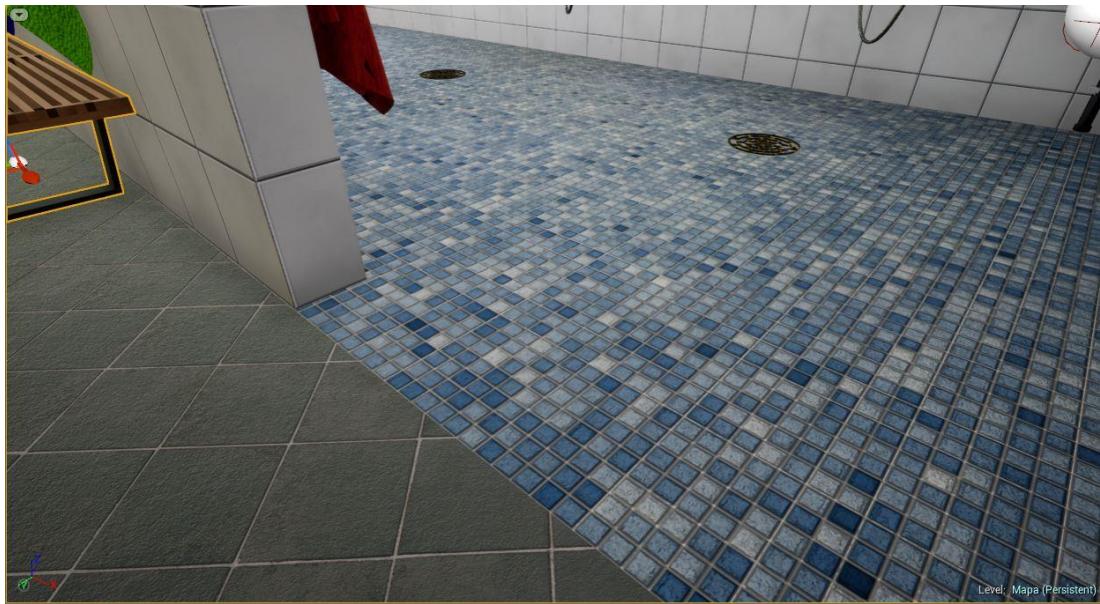


Figura 204: Aplicación del material de ducha

Los mapas de texturas que acabamos de ver son los más utilizados y los que están presente en la mayoría de modelos, pero también hay otros que son usados aunque no con tanta frecuencia, como puede ser una imagen que actué a modo de mascara permitiendo ver a través de ciertas partes del objeto.

Para este proyecto se ha tenido que crear una máscara para el material de los objetos de las vallas metálicas. Esta imagen se ha hecho en Photoshop. El proceso ha sido el siguiente:

Primero teníamos el modelo el cual necesita la máscara:



Figura 205: Valla metalica sin mascara

A partir de su mapa difuso hemos creado la máscara

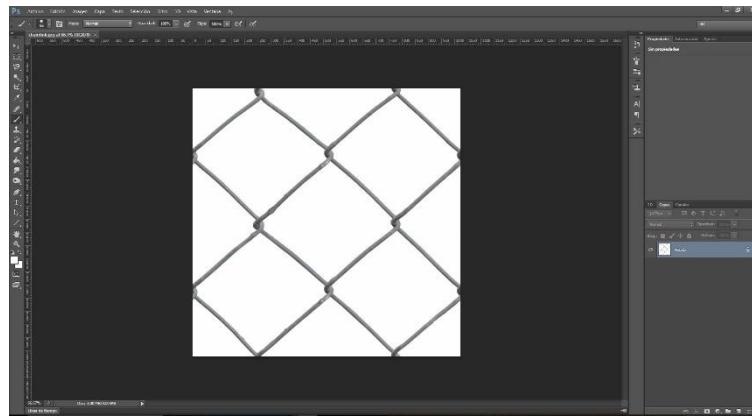


Figura 206: Mapa difuso desde Photoshop

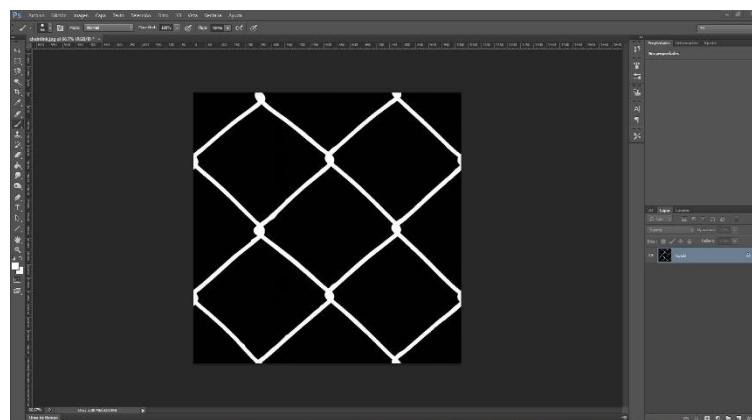


Figura 207: Creación de máscara desde Photoshop

Aplicación de mascara al objeto:



Figura 208: Máscara aplicada al objeto

Una vez que hemos aplicado la máscara ya podemos ver a partir de las partes del objeto que estaban de color negro en la imagen que hace de máscara.

5.3.4.5 Arreglando UVs de modelos

Otro problema que ha surgido por utilizar modelos de terceros es que muchos de ellos o no tenían UVs o al optimizar el modelo se rompían, por lo tanto cuando se intentaba calcular el *lightmap* en el cálculo de la sombra sobre ellos estos modelos quedaban totalmente oscuros.

La solución a esto ha sido volver a hacer los UVs desde 3DS Max. A continuación vamos a ver los 2 casos en los que había que reconstruir los UVs que son:

- El UVs ya no sirve después de modificar la malla del objeto para optimizarla.
- El objeto no tenía UVs.

El primer caso vamos a verlo sobre el inodoro, el cual algunas partes aparecen oscuras puesto que se encuentran fuera del mapping como se verá a continuación.



Figura 209: Modelos con UVs malos

En esta foto se ve como el cálculo de sombra sobre este objeto no se realiza correctamente, hace efectos extraños y además lo que sería la tapa se ve de color negro cuando tendría que ser de color blanca.

En la foto de abajo podemos ver ahora como está el UV de este modelo.

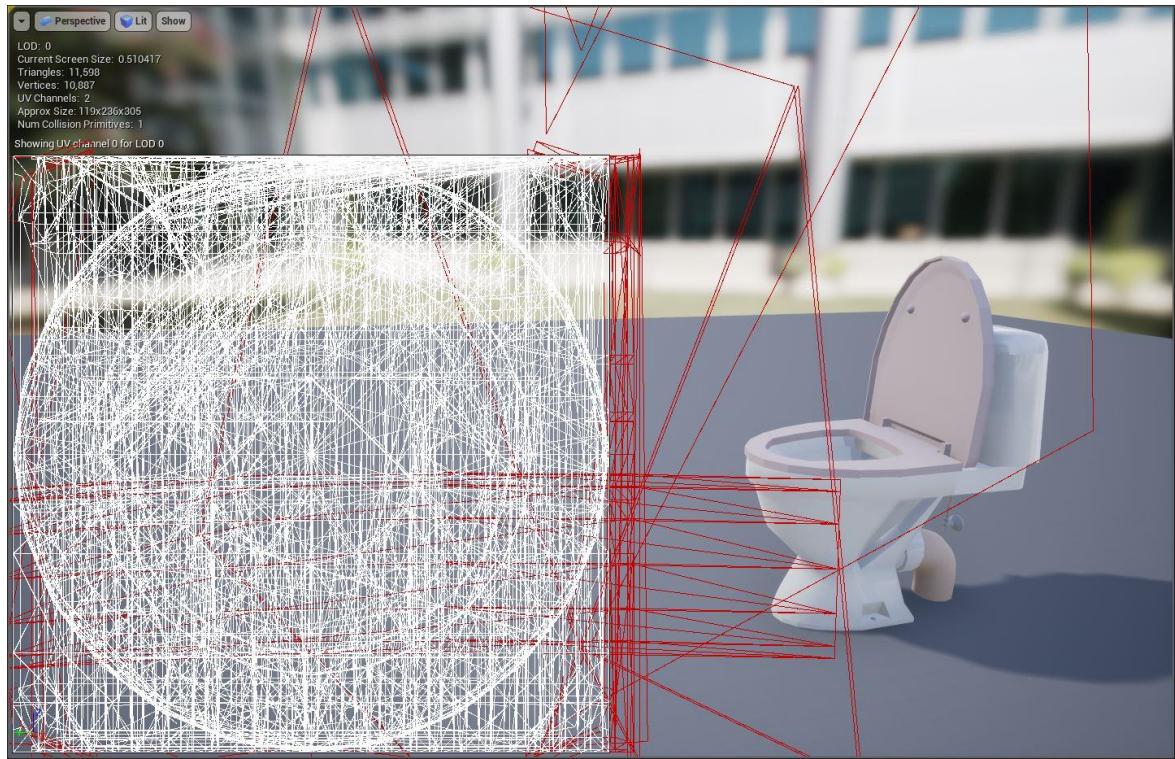


Figura 210: Mostrando los UVs desde Unreal

Como podemos observar hay zonas del modelo que quedan fuera de lo que es el UV produciendo estos efectos extraños y calculando las sombras de manera incorrecta.

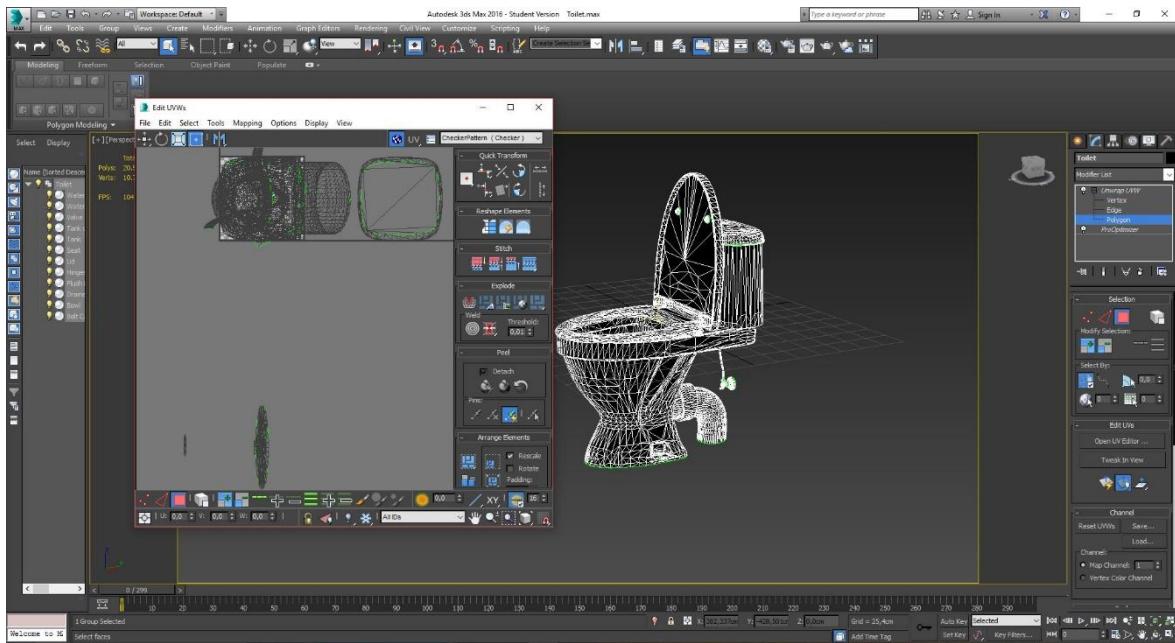


Figura 211: Mostrando UVs (malos) desde 3DS Max

En esta imagen se puede observar el UV pero esta vez desde 3DS Max donde podemos ver ahora de manera más clara que hay partes del objeto fuera del UV.

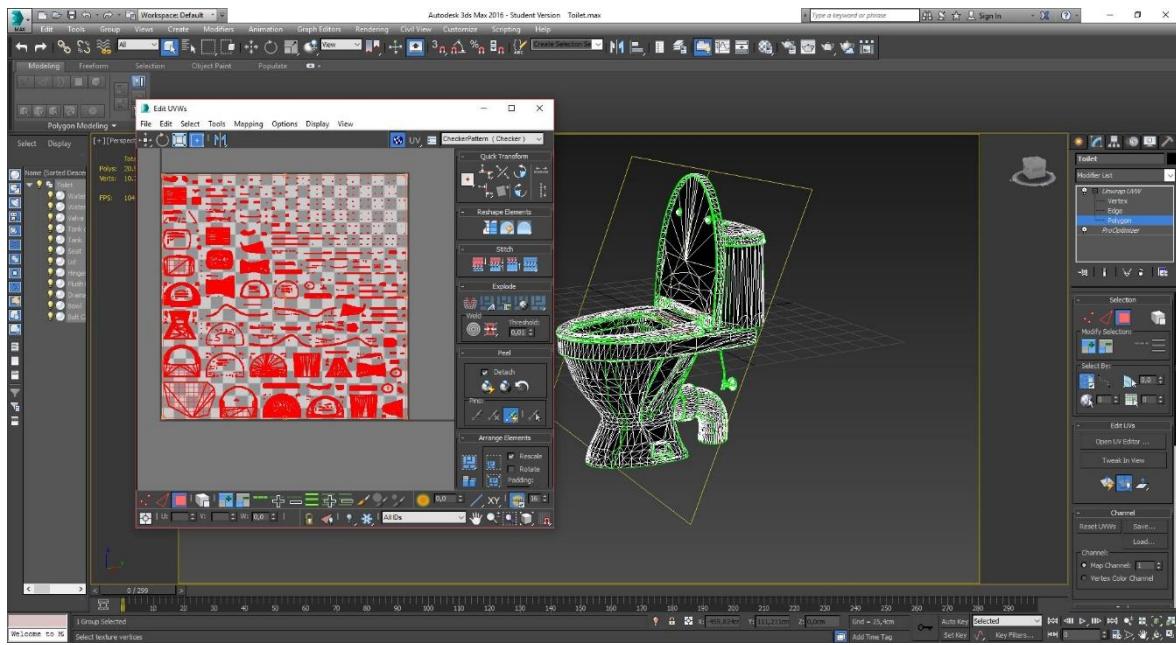


Figura 212: Mostrando UV arreglado desde 3DS Max

En esta imagen se puede ver el UV ya arreglado desde 3DS Max.



Figura 213: Mostrando UV arreglado desde Unreal

Como se puede observar ahora todas las partes del modelo se encuentran dentro del UV permitiendo así una correcta iluminación del mismo como veremos en la siguiente foto.



Figura 214: Modelo de inodoro con UVs arreglados

Una vez que ya hemos visto el primer paso en el que el UV se había roto por haber modificado la malla para optimizarla, vamos a ver ahora el segundo caso que es cuando no tiene ningún UV.

Esta vez vamos a explicarlo todo seguido y luego se mostraran las imágenes del proceso. El modelo sobre el que vamos a trabajar ahora es el de las duchas del vestuario. Las imágenes del proceso son las siguientes:

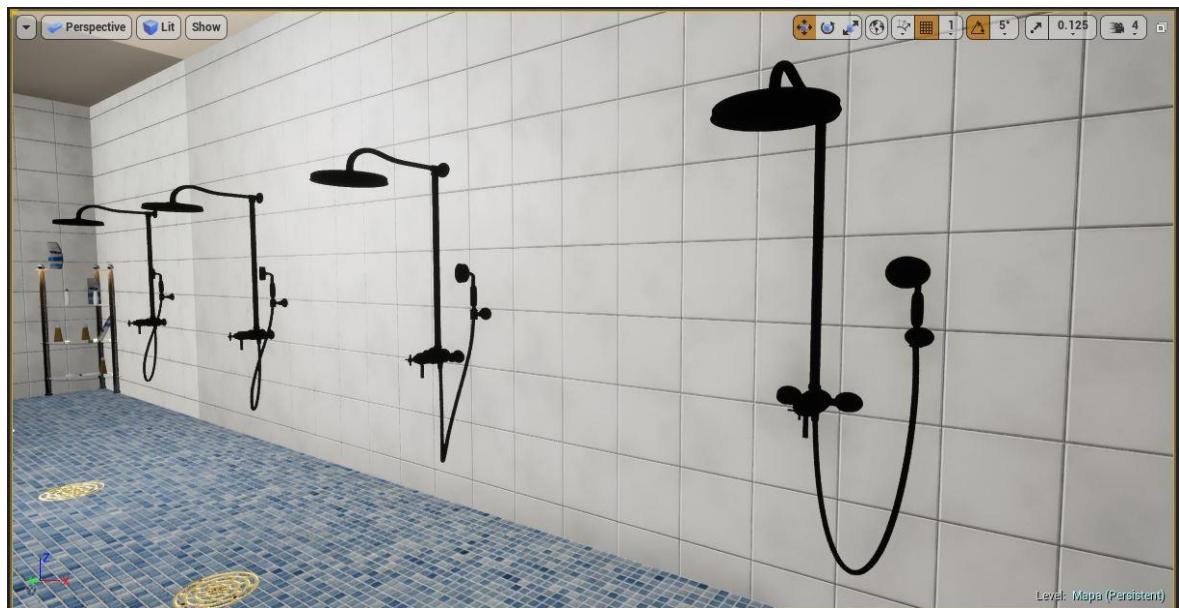


Figura 215: Visualizacion del modelo sin UV

Como se puede ver en esa imagen el objeto aparece completamente negro pues no es posible el cálculo de la luz sobre él.



Figura 216: Muestra del UV desde Unreal.

En esta imagen se puede ver desde el propio editor de mallas de Unreal Engine 4 como el objeto carece de UV.

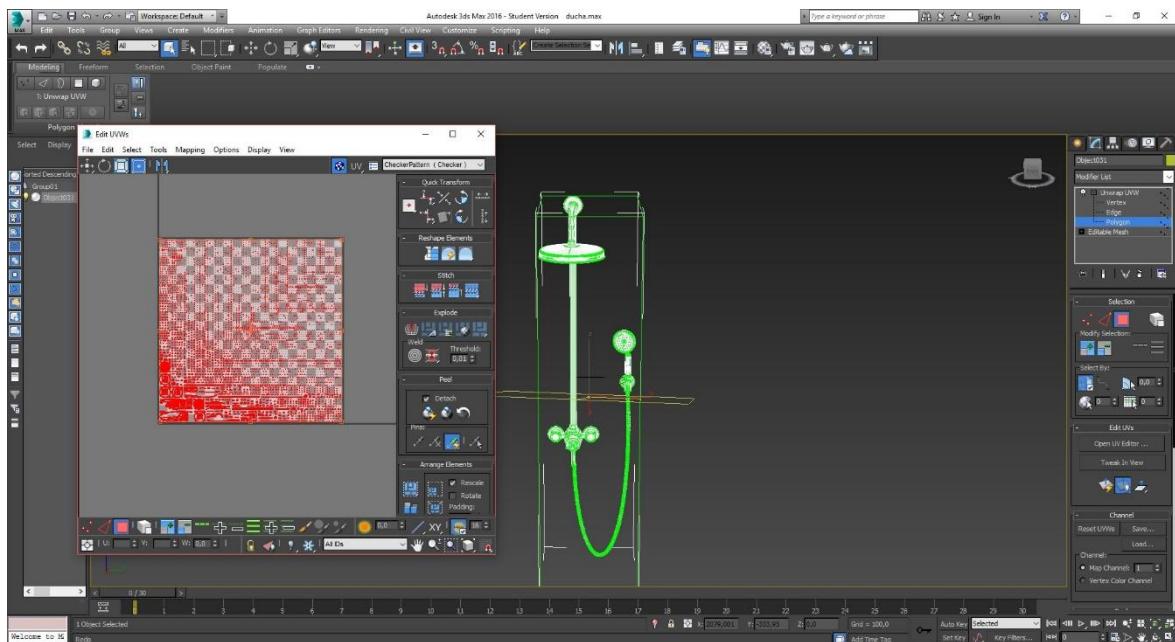


Figura 217: Creación del UV desde 3DS Max

En esta imagen podemos observar el UV que se ha creado para este modelo desde el programa 3DS Max.

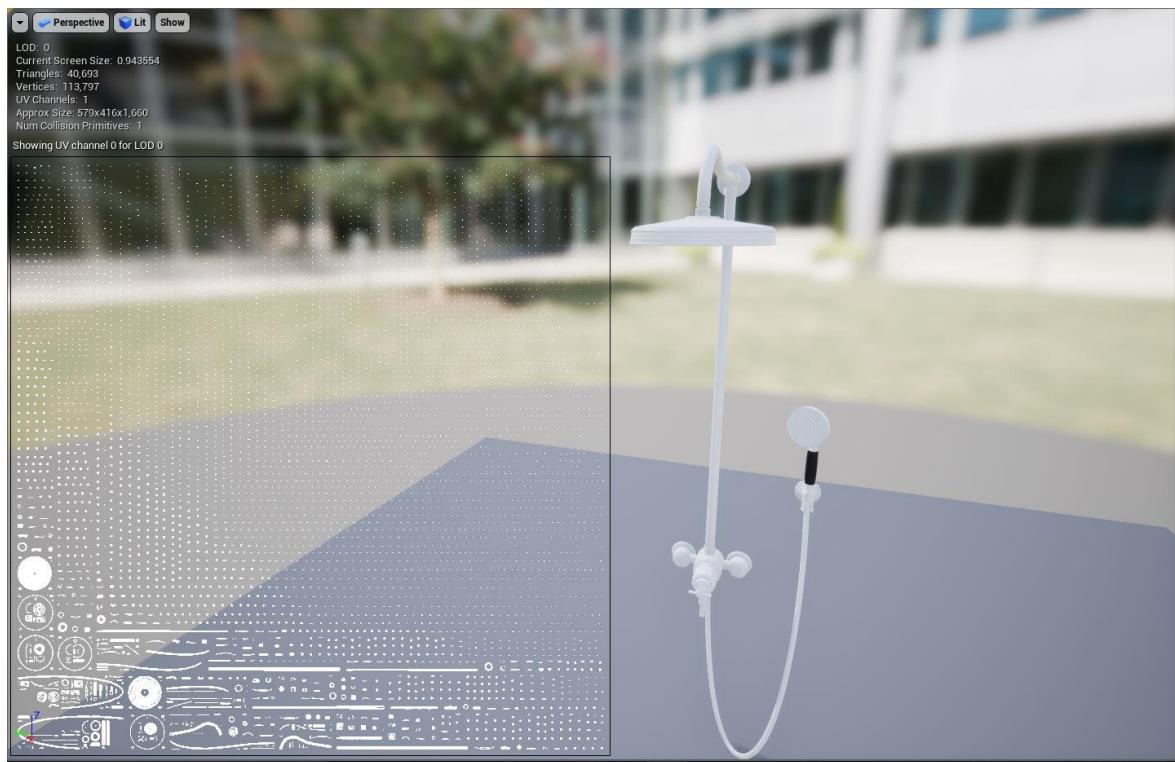


Figura 218: Visualización del UV creado en Unreal

En esta foto podemos ver el UV que hemos creado en el paso anterior pero esta vez desde el propio Unreal.

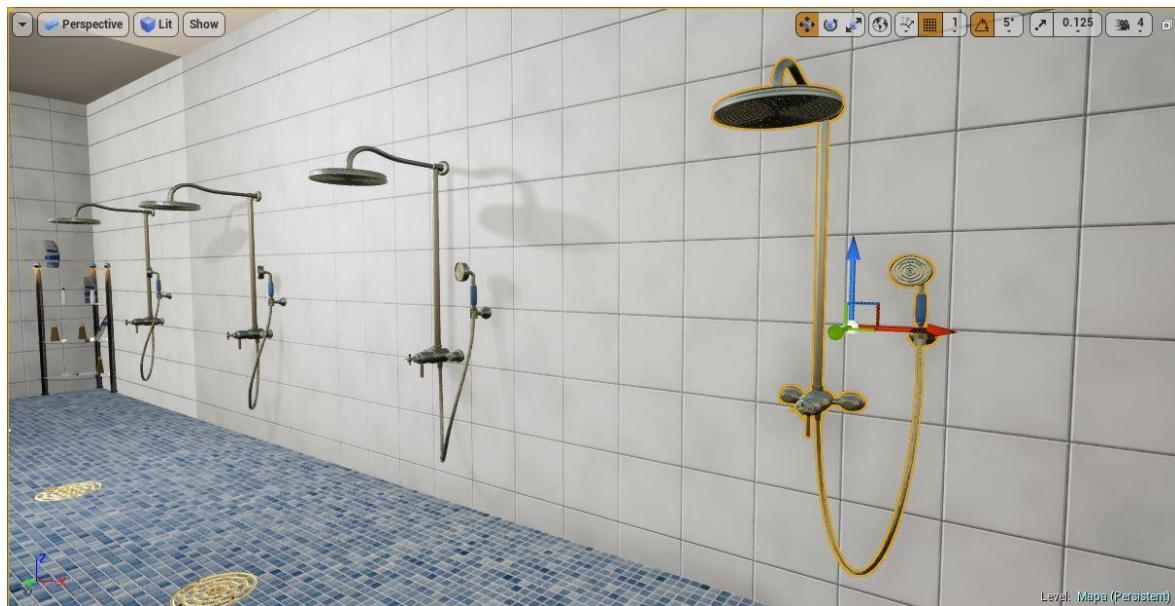


Figura 219: Visualización del modelo con el UV ya creado

Finalmente podemos ver el objeto desde Unreal Engine y esta vez con un correcto cálculo de la iluminación puesto que ahora sí que tiene un UV.

5.3.5 Creación de nivel de juego

En este apartado se va a intentar resumir todo el proceso que se ha llevado para la construcción del escenario.

5.3.5.1 Diseño del nivel

Desde un primer momento queríamos que nuestro escenario fuera una especie de base militar, puesto la historia está ambientada en un conflicto bélico y nuestro personaje se encuentra dentro de una base militar cuando la humanidad llega a su fin.

Aunque la referencia para el escenario han sido bases militares estas no se han recreado a escala 1:1 en nuestro escenario, sino que en todo momento estaba buscando un punto intermedio entre situar al jugador dentro de una base militar pero consiguiendo al mismo tiempo un acabado vistoso y realista en el cual el jugador pueda jugar cómodamente.

Como se puede observar el nivel es muy espaciado, ya que para este tipo de juegos en los que tienes una gran cantidad de enemigos siguiéndote necesitan una gran cantidad de espacio para poder huir de ellos, si esto no fuera así el juego sería muy frustrante pues te quedarías bloqueado en el escenario cuando los enemigos te rodean destruyendo así toda la jugabilidad.

Para evitar que esto ocurra se ha desarrollado el escenario pensando todo momento en la jugabilidad y la mejor experiencia para el jugador, teniendo siempre algún camino alternativo por el que huir en el caso de que te vengan zombis en varias direcciones.

Además de estos se han introducido obstáculos en todas las salas que aunque en primer lugar se pueda pensar que son obstáculos para el jugador o que simplemente son elementos decorativos estos elementos te pueden salvar la vida, puesto que puedes rodear estos elementos haciendo que los enemigos que te persiguen tengan que rodearlo también y como ellos son más lentos ganarías un tiempo clave para poder huir.

Un claro ejemplo de esto es el coche que esta puesto en el exterior situado justo en el centro.



Figura 220: Uso del obstáculo central

El coche seleccionado en la foto nos serviría como obstáculo para poder despistar a los enemigos. Por lo tanto supongamos que estamos en una partida ya avanzada donde hay una gran cantidad de zombis, estos podrían aparecer por cualquier lado por lo tanto podría pasar que vinieran tanto por la puerta rodeada de verde como por la puerta rodeada de rojo.

Si esto sucede **el jugador podría seguir la trayectoria de la flecha azul**, dando una vuelta sobre este obstáculo central con el objetivo de salir por la puerta verde. Al hacer esto los zombis que entraban por la puerta verde te estarán siguiendo dando la vuelta también al coche siguiendo la trayectoria marcada por las líneas en verde. Mientras que los zombis que entran por la puerta roja darán solo media vuelta al coche siguiendo la trayectoria de la flecha roja. Llegados a este punto hemos conseguido reunir los 2 grupos de zombis en uno solo (que seguirían la trayectoria de la flecha blanca) y hemos conseguido despejar la puerta verde por la cual podremos salir ahora teniendo un único grupo de zombi que nos persigue.

De igual manera se han diseñado el resto de salas en las cuales siempre hay un obstáculo que te permite dar la vuelta y fusionar dos grupos de zombis para obtener siempre una vía de escape.

Ahora que ya hemos dejado claro como se ha pensado el diseño del juego vamos a ver algunas imágenes de los primeros bocetos, como estaban las salas en sus primeras fases y el resultado final

pág. 228

de alguna de ellas. Para poder ver mejor todo el resultado final del escenario se ha creado un anexo de arte al final del documento.

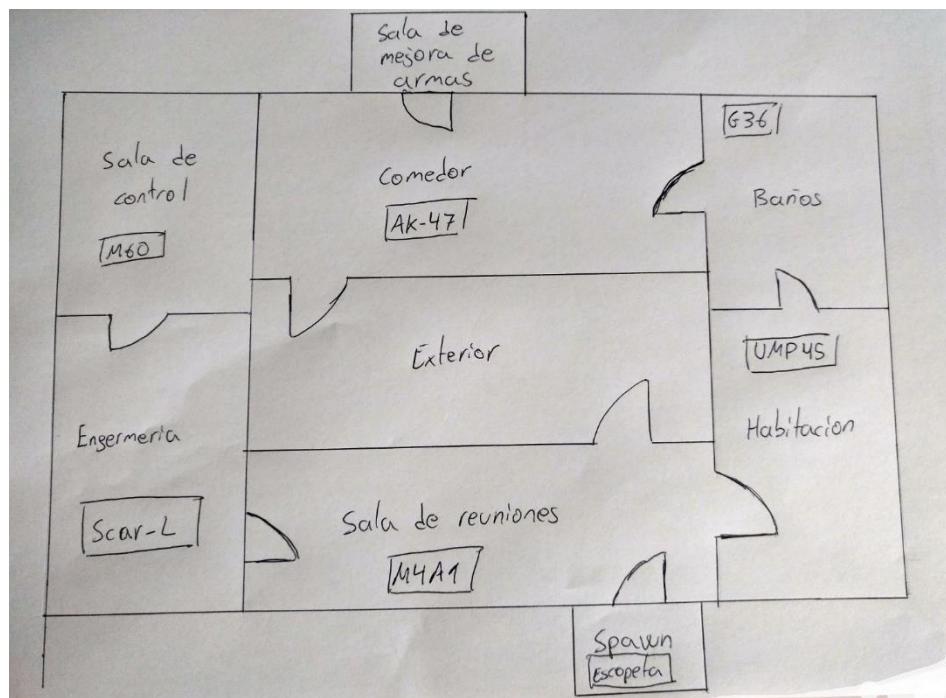


Figura 221: Primer plano del escenario

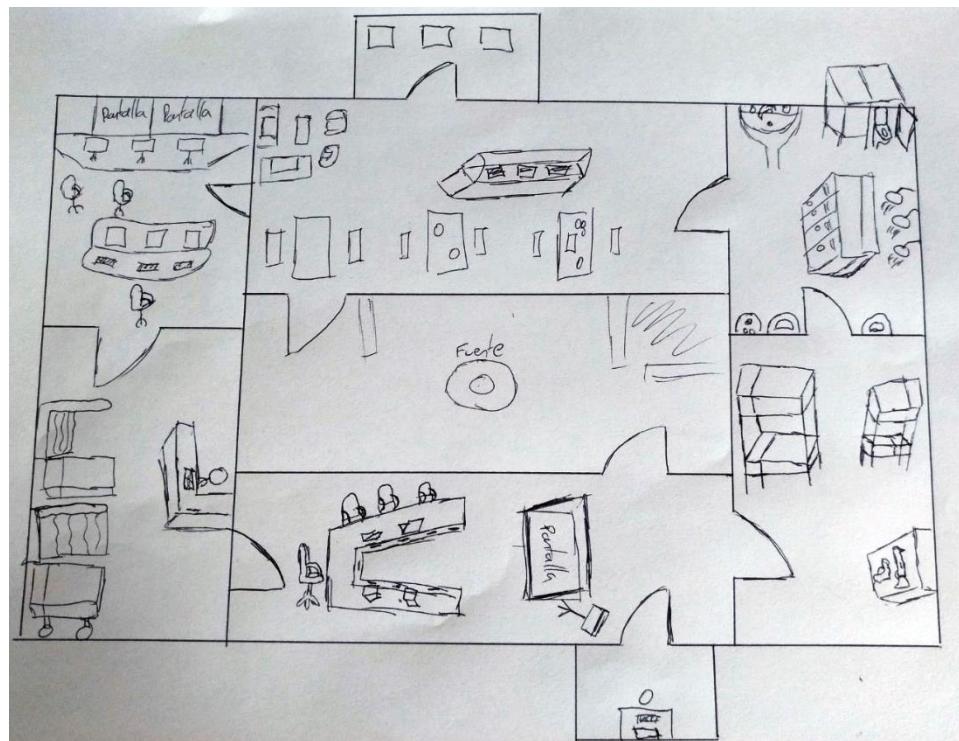


Figura 222: Primer boceto del escenario



Figura 223: Comienzo del vestuario



Figura 224: Resultado final vestuario

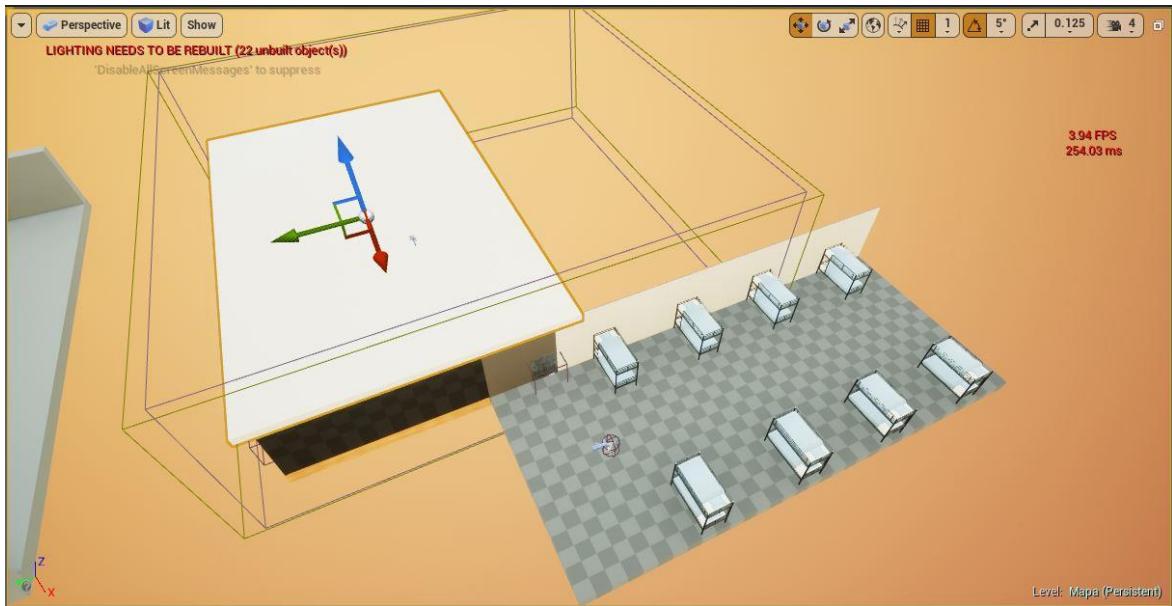


Figura 225: Comienzo habitaciones



Figura 226: Resultado final habitaciones

Se han grabado varios videos mientras se estaba creando el escenario y luego juntados en un único video (por sala) en cámara rápida para poder observar el proceso de creación del nivel. Dos ejemplos son los siguientes:

- Creando el comedor: <https://www.youtube.com/watch?v=vMYZXAcb5E0>
- Creando la enfermería: <https://www.youtube.com/watch?v=gtBwbjdEMuM>

5.3.5.2 Sonido e iluminación

Tanto el sonido como la iluminación de este proyecto han sido cuidados con gran detalle con el objetivo de conseguir una buena calidad final.

Los sonidos añadidos tienen como intención proporcionar feedback al jugador e intentar conseguir una completa inmersión del usuario en el juego. Así por ejemplo cuando un zombi se acerca escucharíamos sus gritos o cuando realizamos una compra o nos acercamos a una maquina escucharíamos sus sonidos.



Figura 227: Ejemplo sonidos en el escenario

En cuanto a la iluminación ha sido elegida en cada una de las salas de manera independiente para poder ajustarse mejor a los requerimientos de la misma, por ejemplo, la enfermería es junto al comedor las salas más iluminadas del juego porque son salas donde se necesita gran cantidad de luz. Mientras que la iluminación de las habitaciones es más tenue.

En la iluminación de la sala de control se ha intentado dar la sensación de que esta sala pequeña y sin mucha luz e iluminada casi en su totalidad por los proyectores que apuntan hacia las pantallas. Para ello aunque hay 3 luces en el techo 2 de estas aparecen apagadas como se puede observar en la imagen de abajo.



Figura 228: Ejemplo iluminación sala de control

Otro caso aparte es por ejemplo el exterior del mapa en el que se ha intentado recrear una escena donde única iluminación existente proviene de las luces de los coches.

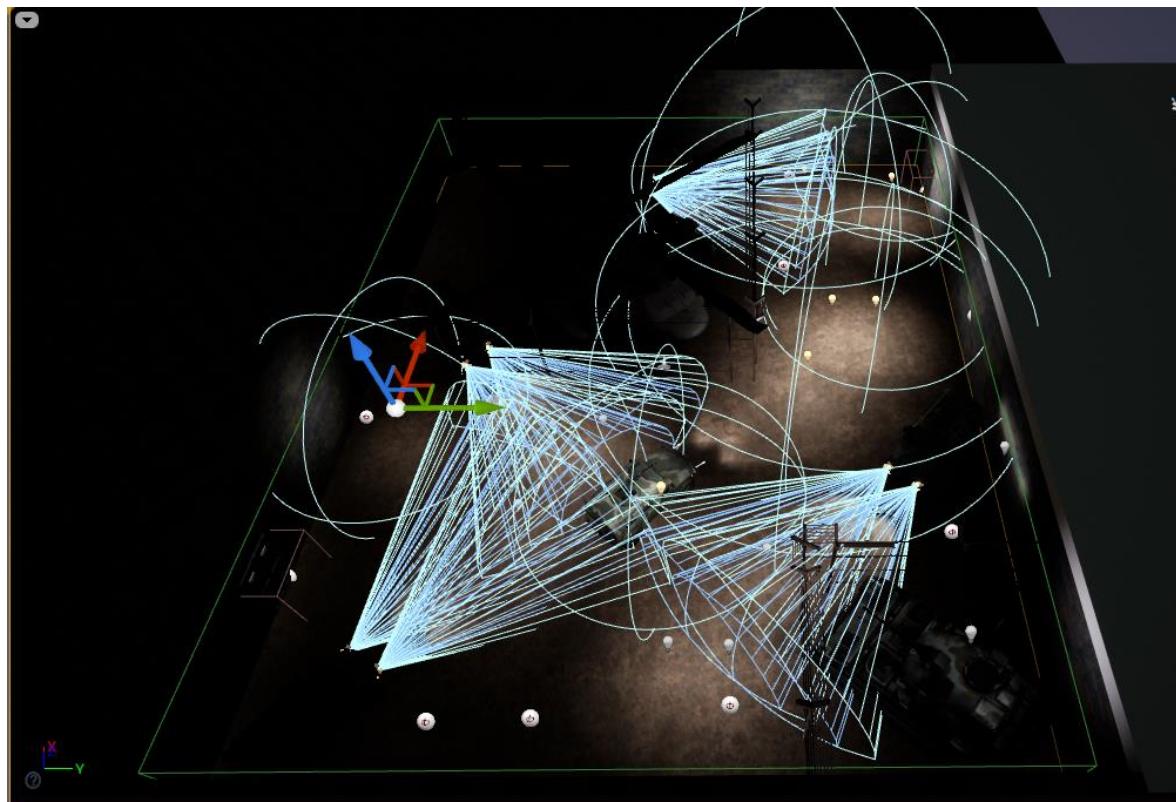


Figura 229: Ejemplo iluminación exterior



Figura 230: Ejemplo 2 iluminación exterior

6. Conclusiones

Después de mucho tiempo de trabajo se ha conseguido el objetivo principal que era el de tener un producto final además de conseguir dominar el motor gráfico de Unreal Engine 4.

Hay que dejar claro que este juego final se ha hecho para un TFG cuyo propósito no ha sido otro que el de aprender a utilizar Unreal Engine 4. Por lo tanto el juego está muy lejos de ser una versión comercializable. A pesar de esto el resultado final ha sido muy satisfactorio pues se ha conseguido un juego acabado el cual puede ser jugado y con una calidad bastante alta.

Por lo tanto hay que valorar la calidad del juego sabiendo que este ha sido desarrollado por un único estudiante de Ingeniería Multimedia y no por un equipo de desarrolladores expertos en este sector.

Haciendo un resumen de lo anterior con este proyecto no solo se han conseguido cumplir todos y cada uno de los objetivos propuestos sino que además se ha obtenido un producto final de gran calidad.

Se ha realizado un análisis de las herramientas disponibles bastante completos, viendo puntos a favor y en contra de cada una y realizando una elección final.

Se ha realizado también un GDD del videojuego en el cual se han detallado todos los aspectos relacionados con el diseño del videojuego. Hay que decir que el videojuego puede ser jugador por cualquier persona (aunque la clasificación PEGI sea para mayor de 18, pero esto es siempre una recomendación).

Aunque se han utilizado modelos de terceros, se han creado una gran cantidad de modelos con otros programas para ser luego importados a Unreal Engine y poder ser utilizado.

La sonorización del juego aunque no es una parte esencial del proyecto ha sido tratada cuidadosamente.

La creación de los menús y HUD se ha realizado mediante Photoshop y el sistema de Scripting Visual Blueprint como se especificó en los objetivos. El resultado de estos es bastante bueno, pues el HUD consta también de animaciones que facilitan la comprensión de lo que está sucediendo a tu alrededor y por tanto enriquecen la experiencia del usuario.

La IA aunque es sencilla es bastante completa, teniendo un Behaviour Tree para definir su comportamiento, animaciones, incluso activación de ragdoll cuando mueren. La máquina de estados que se encarga de escoger animación del enemigo oportuna en cada frame es bastante compleja con lo que hemos conseguido por ejemplo que un enemigo al que le hayas disparado en las piernas tenga que arrastrarse.

Se ha implementado tambien un sistema de ranking que funciona correctamente, característica que veía imprescindible para este juego, ya que es una manera de atraer el usuario intentando que este se supere a sí mismo y a sus amigos, creando un “pique sano” que lleva a seguir jugando este juego para intentar alcanzar la mejor puntuacion.

Cabe destacar que uno de los aspectos a los que más importancia se le ha dado en este proyecto era el de conseguir un escenario realista y con una iluminación de calidad. Creo que este objetivo ha sido cumplido con bastante éxito. Se ha conseguido por tanto un escenario con una gran cantidad de detalles adaptado además a la jugabilidad, creando salas bastante espaciadas en las que el jugador pueda moverse con tranquilidad aun siendo perseguido por un gran número de zombis.

Por último la gran variedad de armas, las distintas maneras en el que estas pueden ser mejoradas, mejorando sus estadísticas individualmente permitiendo adaptarse al estilo de juego de cada jugador, sumado a la gran variedad de formas de recorrer el mapa y rutas a seguir hace que el juego sea divertido y que cada partida sea única y diferente al resto, creando en el usuario la necesidad de seguir jugando.

Esta necesidad de rejugar el juego por parte del usuario se debe tambien a una mejora lineal de tus habilidades a medida que juegas porque empiezas a descubrir que armas son mejores, que recorridos son los más seguros, y sobre todo tus ganas de superar al resto de jugadores o a ti mismo en el ranking del juego.

Se puede ver una muestra del juego en el siguiente video: <https://youtu.be/NPvsn4lw1ek>

7. Anexo: Escenario

En esta sección se muestra el resultado final del escenario del juego. Además se recomienda ver el siguiente video en el cual se hace un recorrido por el escenario pudiendo ver este con mayor detalle y tranquilidad ya que en este video no hay enemigos, se centra en enseñar el escenario final.

<https://www.youtube.com/watch?v=0DVneaDBUo4>

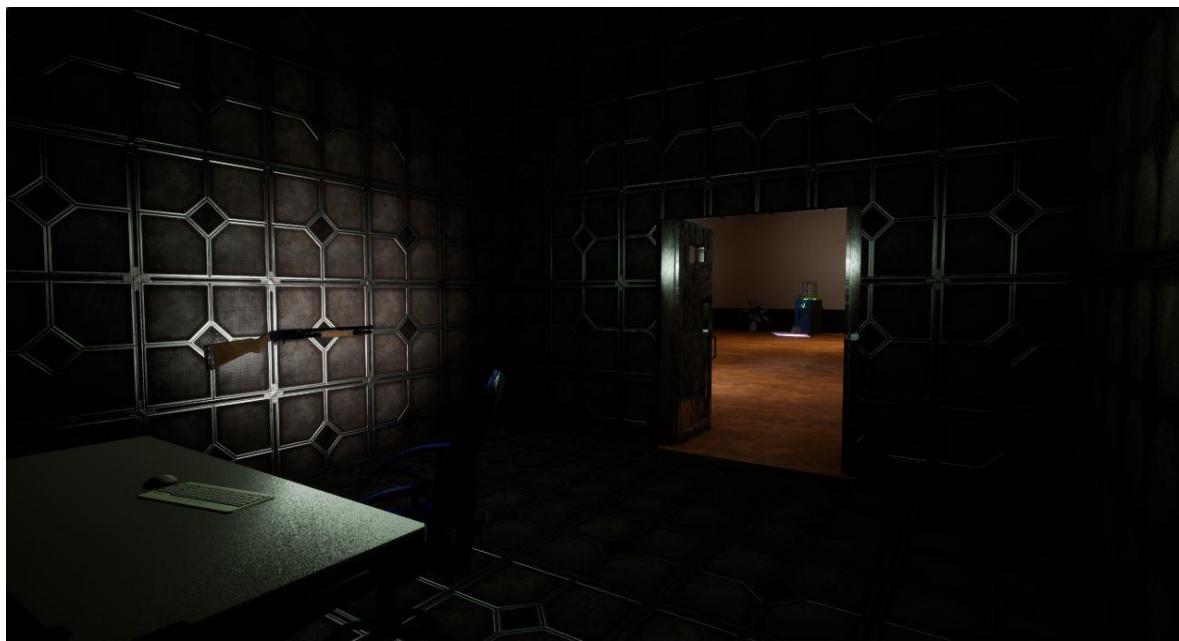


Figura 231: Captura de Postwar: Hopeless Humanity InGame número 1



Figura 232: Captura de Postwar: Hopeless Humanity InGame número 2

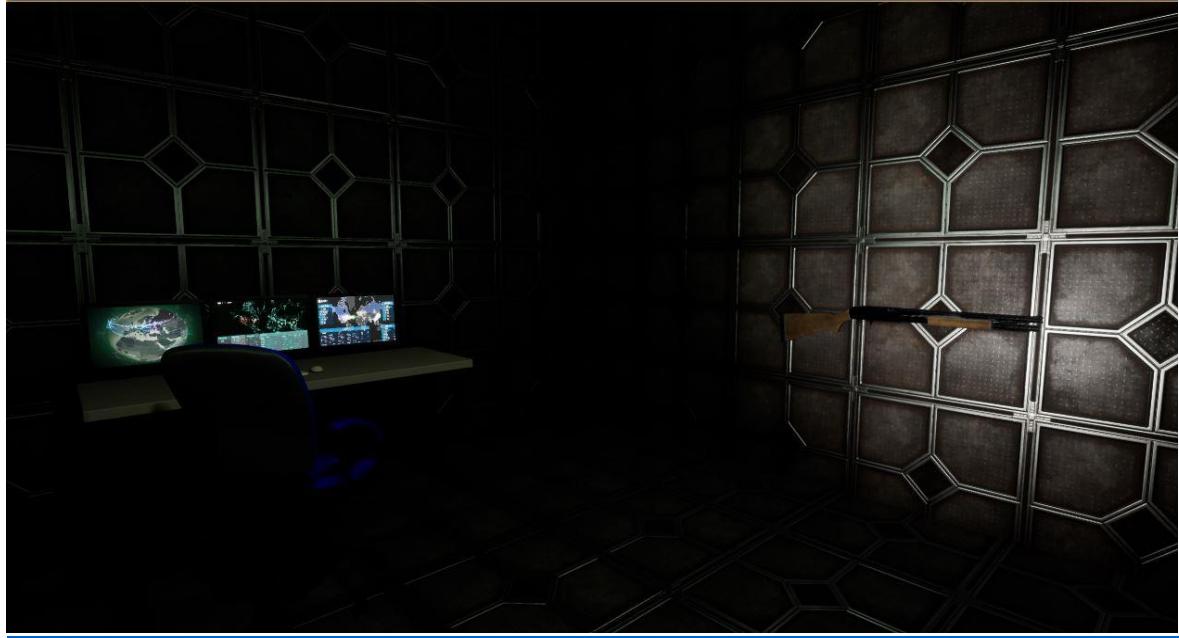


Figura 233: Captura de Postwar: Hopeless Humanity InGame número 3



Figura 234: Captura de Postwar: *Hopeless Humanity* InGame número 4



Figura 235: Captura de Postwar: *Hopeless Humanity* InGame número 5

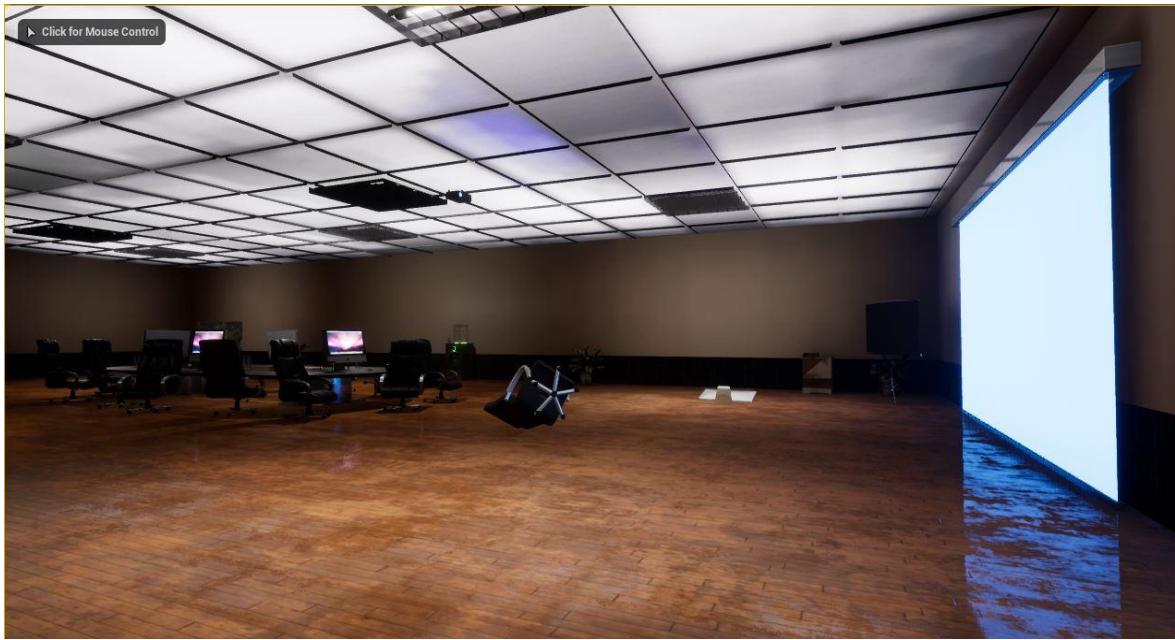


Figura 236: Captura de Postwar: Hopeless Humanity InGame número 6



Figura 237: Captura de Postwar: Hopeless Humanity InGame número 7

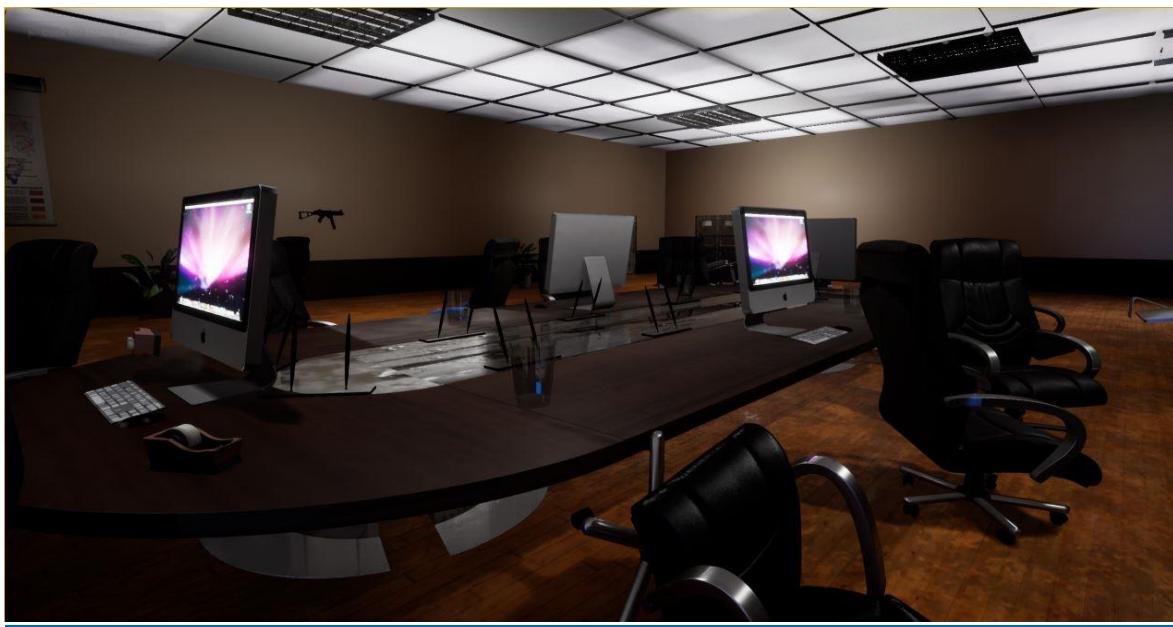


Figura 238: Captura de Postwar: *Hopeless Humanity* InGame número 8

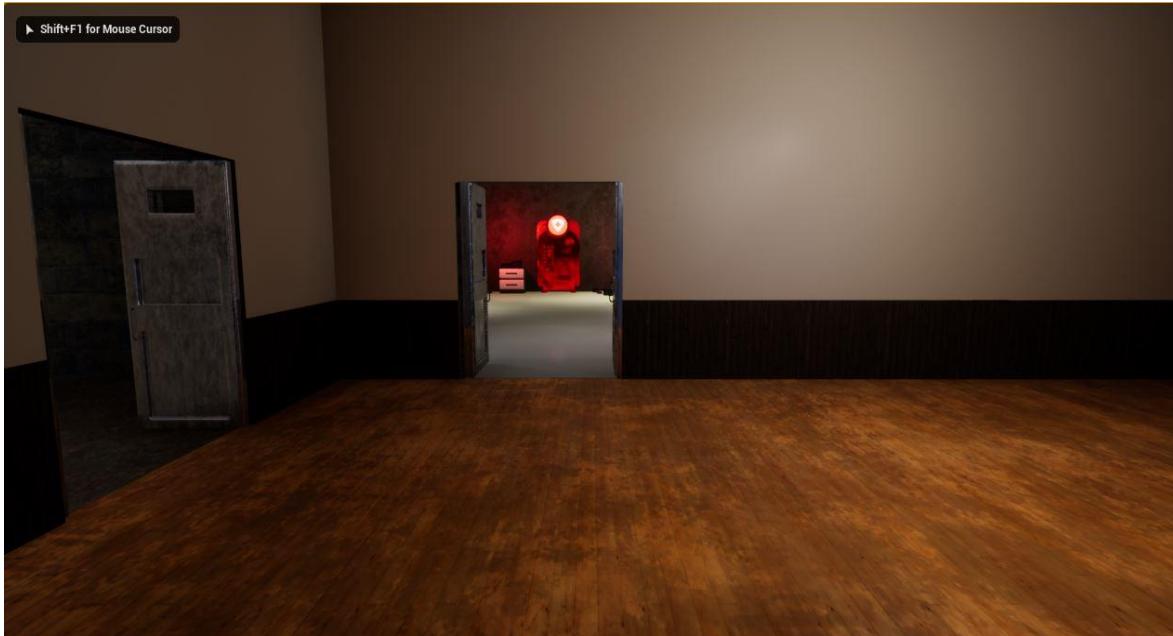


Figura 239: Captura de Postwar: *Hopeless Humanity* InGame número 9



Figura 240: Captura de Postwar: Hopeless Humanity InGame número 10



Figura 241: Captura de Postwar: Hopeless Humanity InGame número 11



Figura 242: Captura de Postwar: Hopeless Humanity InGame número 12



Figura 243: Captura de Postwar: Hopeless Humanity InGame número 13



Figura 244: Captura de Postwar: Hopeless Humanity InGame número 14



Figura 245: Captura de Postwar: Hopeless Humanity InGame número 15



Figura 246: Captura de Postwar: Hopeless Humanity InGame número 16

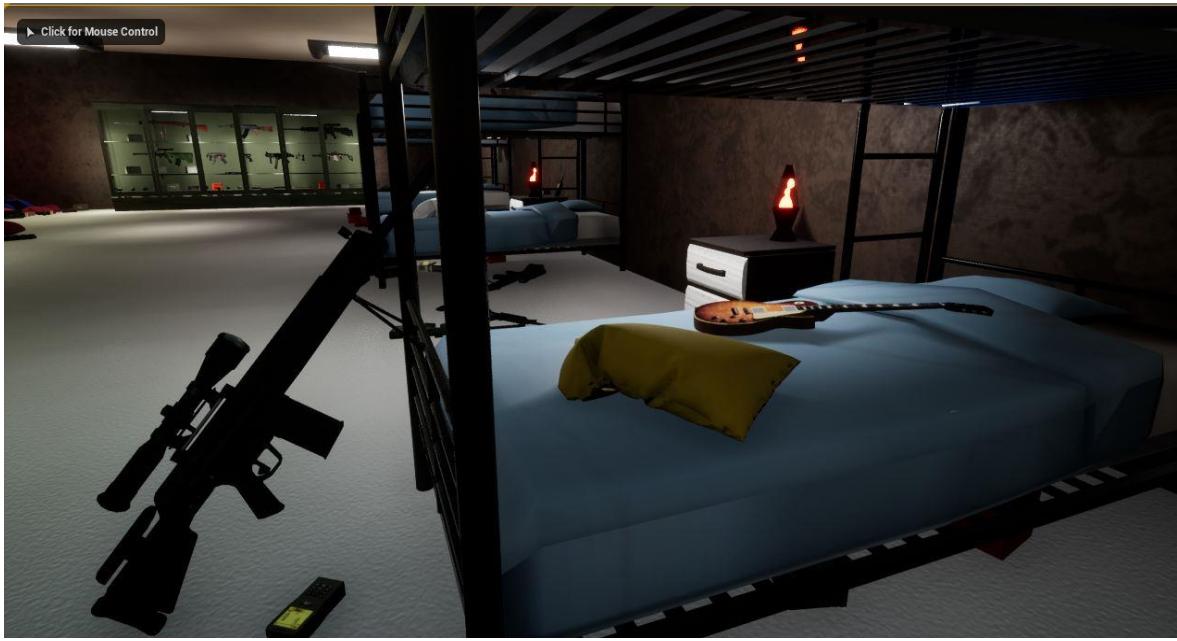


Figura 247: Captura de Postwar: Hopeless Humanity InGame número 17



Figura 248: Captura de Postwar: Hopeless Humanity InGame número 18

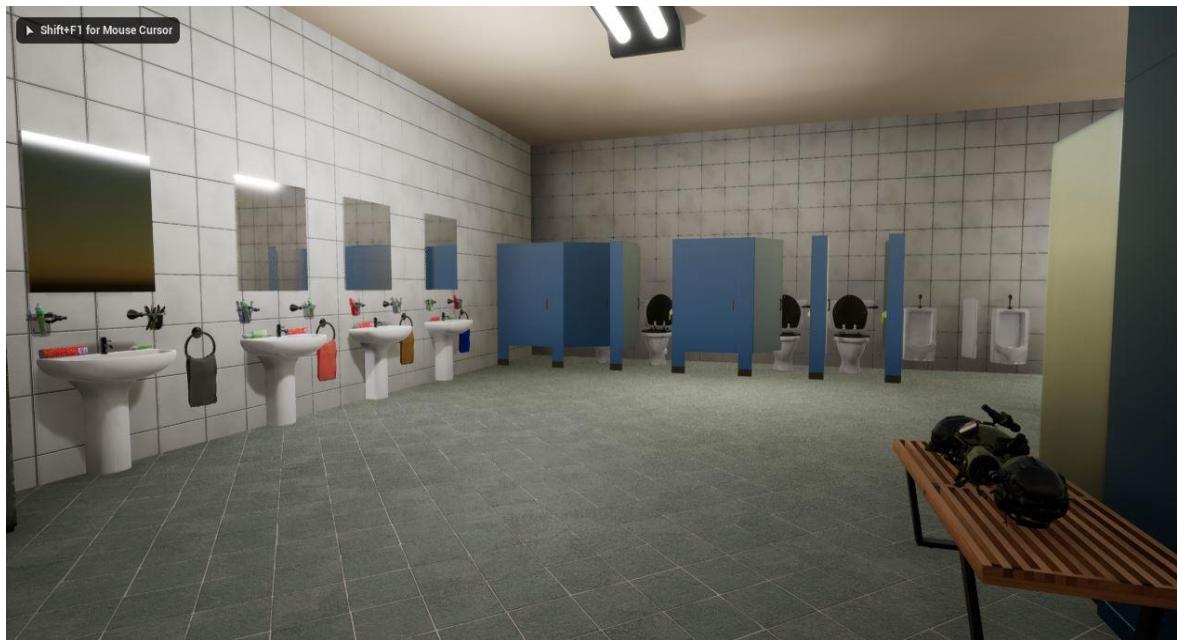


Figura 249: Captura de Postwar: Hopeless Humanity InGame número 19



Figura 250: Captura de Postwar: Hopeless Humanity InGame número 20



Figura 251: Captura de Postwar: Hopeless Humanity InGame número 21



Figura 252: Captura de Postwar: Hopeless Humanity InGame número 22



Figura 253: Captura de Postwar: Hopeless Humanity InGame número 23



Figura 254: Captura de Postwar: Hopeless Humanity InGame número 24



Figura 255: Captura de Postwar: Hopeless Humanity InGame número 25



Figura 256: Captura de Postwar: Hopeless Humanity InGame número 26

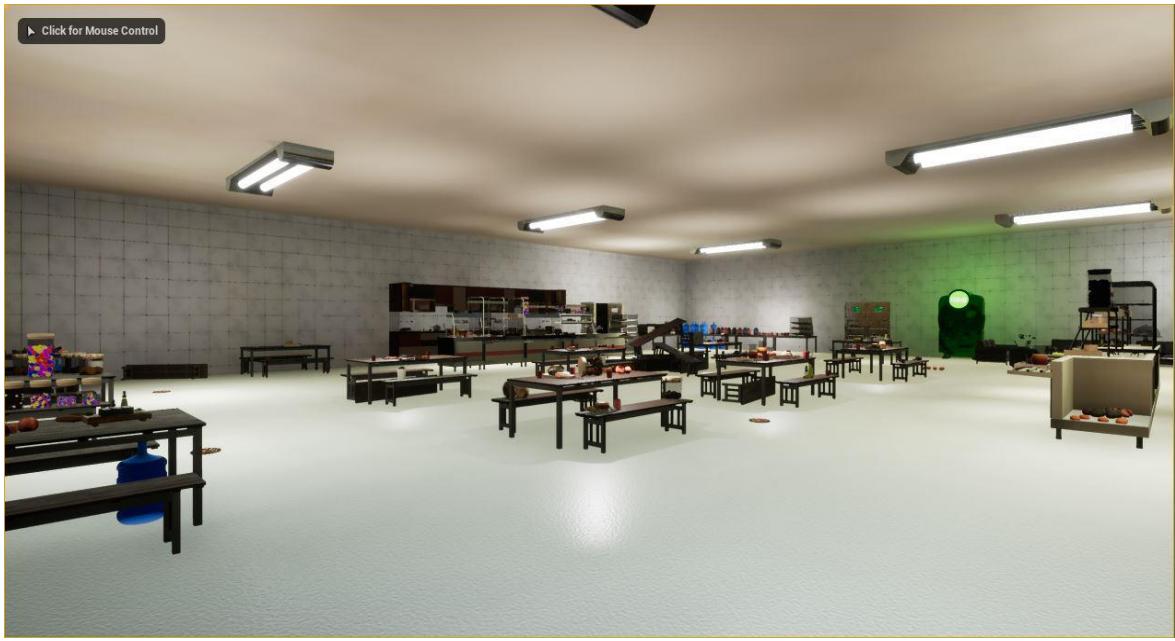


Figura 257: Captura de Postwar: Hopeless Humanity InGame número 27



Figura 258: Captura de Postwar: *Hopeless Humanity* InGame número 28



Figura 259: Captura de Postwar: *Hopeless Humanity* InGame número 29



Figura 260: Captura de Postwar: Hopeless Humanity InGame número 30



Figura 261: Captura de Postwar: Hopeless Humanity InGame número 31



Figura 262: Captura de Postwar: Hopeless Humanity InGame número 32



Figura 263: Captura de Postwar: Hopeless Humanity InGame número 33

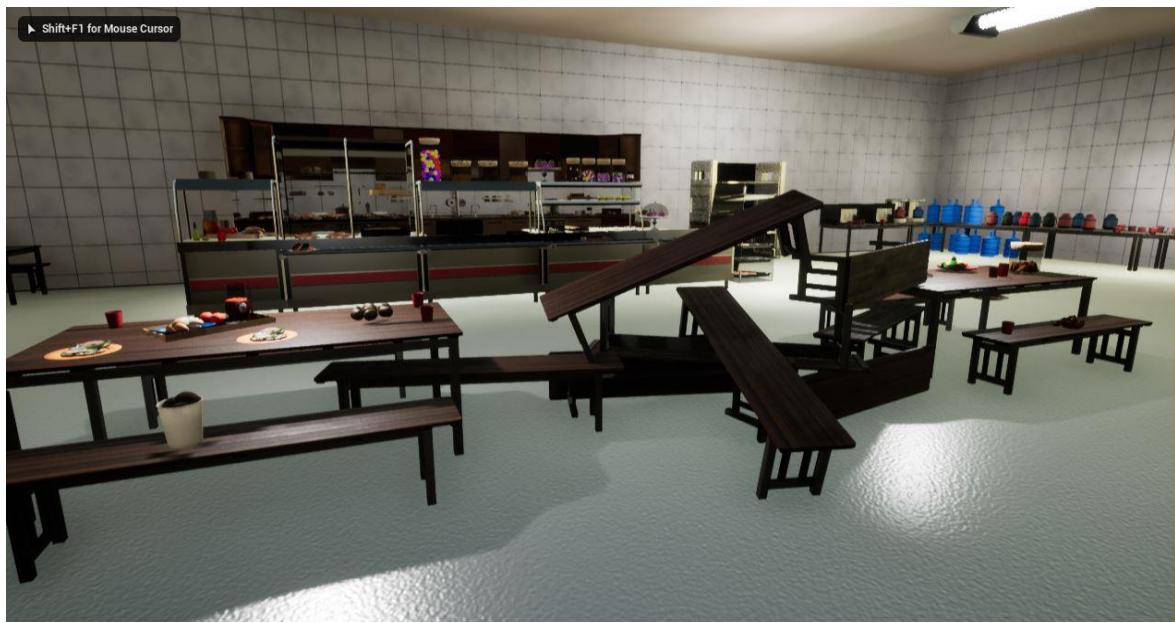


Figura 264: Captura de Postwar: Hopeless Humanity InGame número 34



Figura 265: Captura de Postwar: Hopeless Humanity InGame número 35



Figura 266: Captura de Postwar: Hopeless Humanity InGame número 36

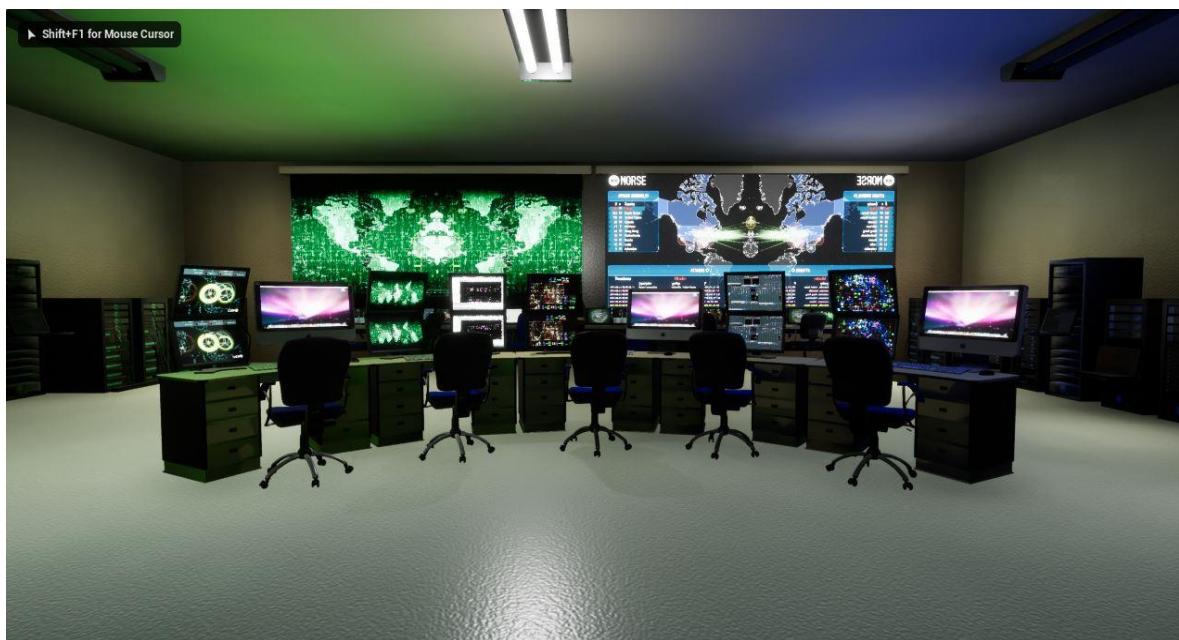


Figura 267: Captura de Postwar: Hopeless Humanity InGame número 37



Figura 268: Captura de Postwar: Hopeless Humanity InGame número 38



Figura 269: Captura de Postwar: Hopeless Humanity InGame número 39



Figura 270: Captura de Postwar: Hopeless Humanity InGame número 40



Figura 271: Captura de Postwar: Hopeless Humanity InGame número 41



Figura 272: Captura de Postwar: Hopeless Humanity InGame número 42



Figura 273: Captura de Postwar: Hopeless Humanity InGame número 43



Figura 274: Captura de Postwar: Hopeless Humanity InGame número 44

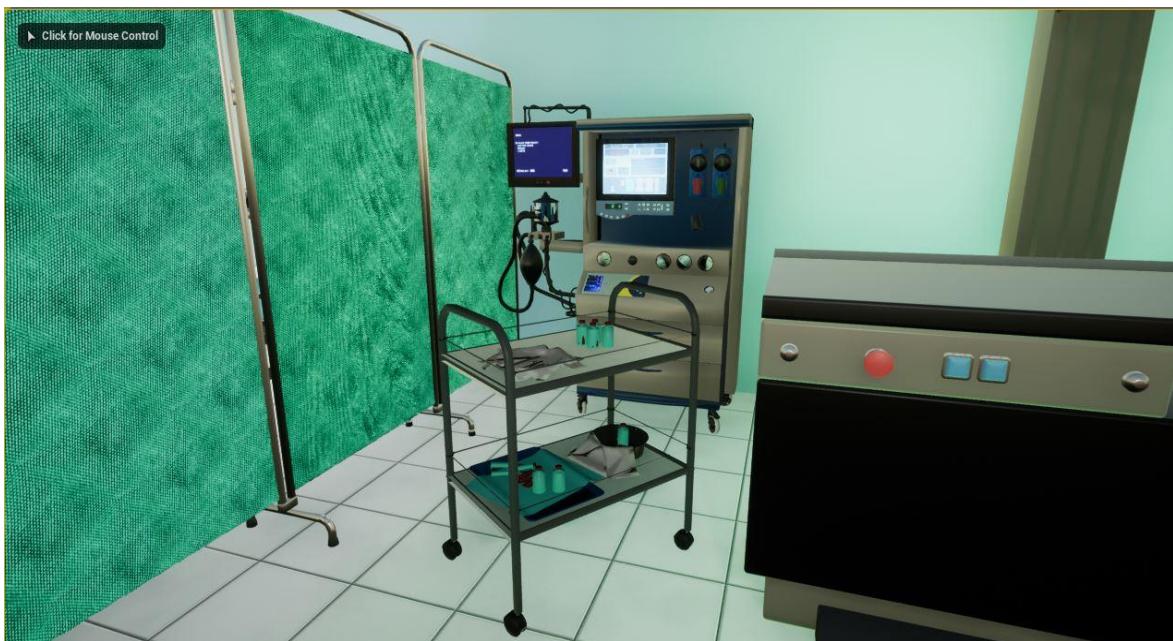


Figura 275: Captura de Postwar: Hopeless Humanity InGame número 45



Figura 276: Captura de Postwar: Hopeless Humanity InGame número 46



Figura 277: Captura de Postwar: Hopeless Humanity InGame número 47



Figura 278: Captura de Postwar: *Hopeless Humanity* InGame número 48



Figura 279: Captura de Postwar: *Hopeless Humanity* InGame número 49

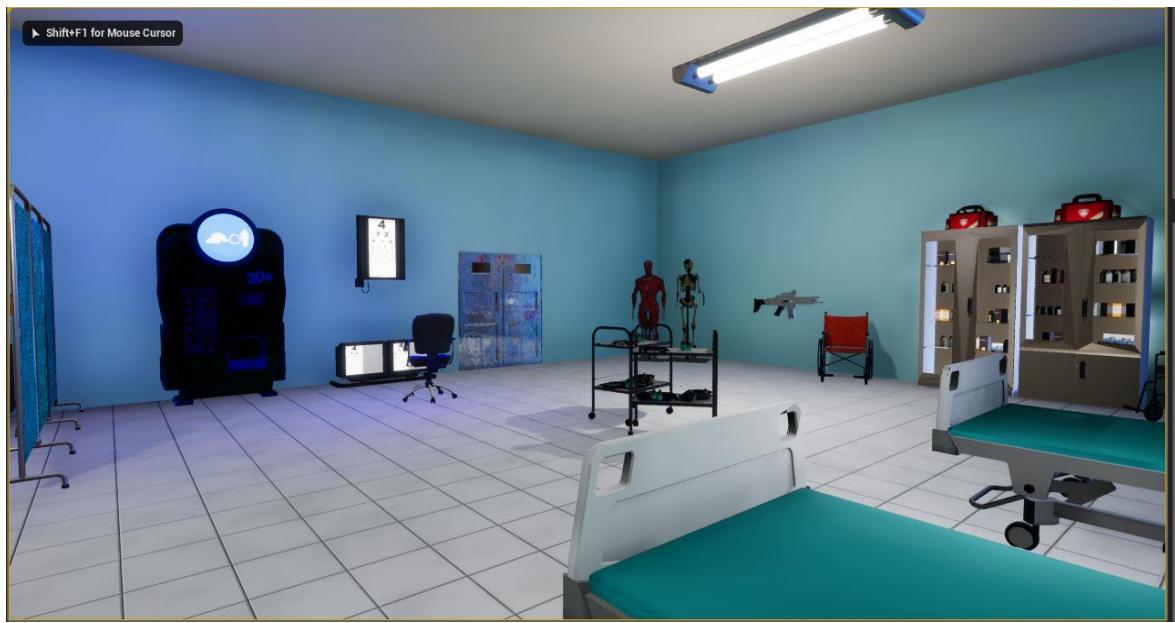


Figura 280: Captura de Postwar: Hopeless Humanity InGame número 50



Figura 281: Captura de Postwar: Hopeless Humanity InGame número 51



Figura 282: Captura de Postwar: Hopeless Humanity InGame número 52



Figura 283: Captura de Postwar: Hopeless Humanity InGame número 53



Figura 284: Captura de Postwar: Hopeless Humanity InGame número 54



Figura 285: Captura de Postwar: Hopeless Humanity InGame número 55



Figura 286: Captura de Postwar: Hopeless Humanity InGame número 56



Figura 287: Captura de Postwar: Hopeless Humanity InGame número 57



Figura 288: Captura de Postwar: Hopeless Humanity InGame número 58

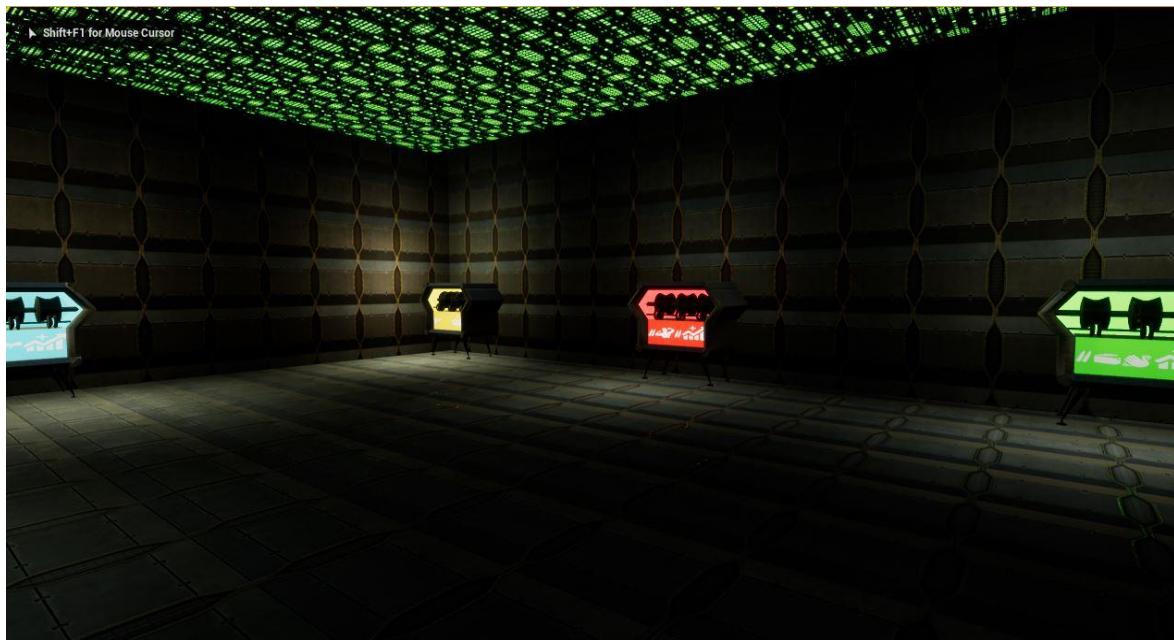


Figura 289: Captura de Postwar: Hopeless Humanity InGame número 59

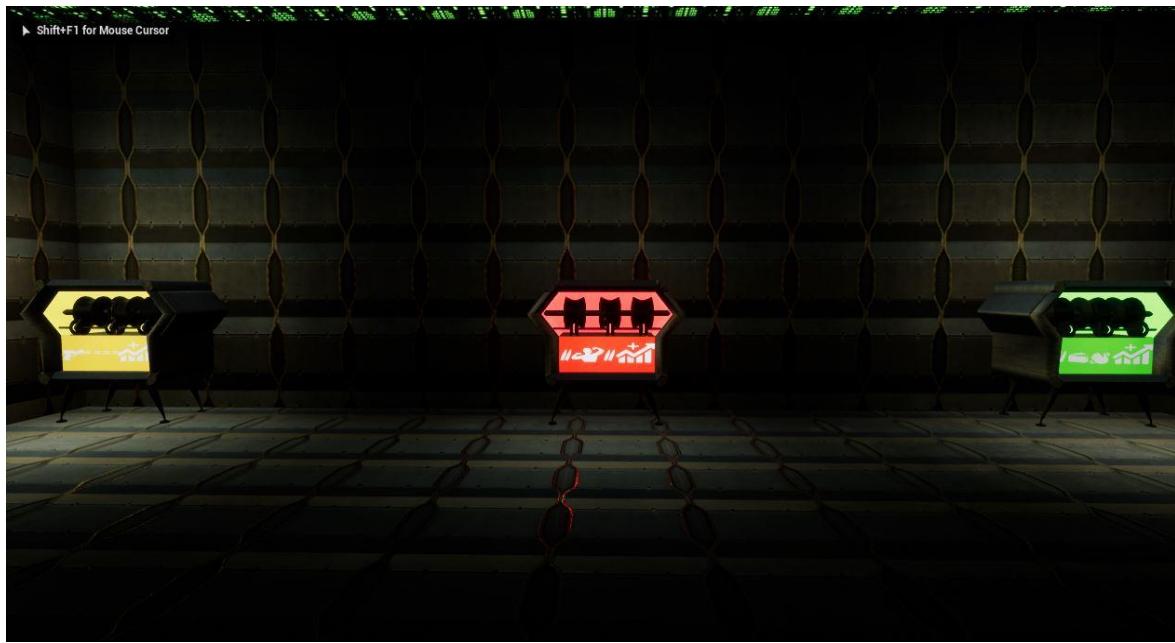


Figura 290: Captura de Postwar: Hopeless Humanity InGame número 60

8. Bibliografía

360, X., & (360), D. (2017). *Supervivencia paradisíaca. Meristation*. Recuperado 23 June 2017, a partir de <http://www.meristation.com/xbox-360/dead-island/analisis-juego/1534918>

Wikipedia. (2017). Recuperado 22 June 2017, a partir de desde
https://es.wikipedia.org/w/index.php?title=Videojuego_de_disparos_en_primera_persona&oldid=9705150

Call of Duty: Black Ops 3. (2017). Es.wikipedia.org. Recuperado 22 June 2017, a partir de
https://es.wikipedia.org/w/index.php?title=Call_of_Duty:_Black_Ops_3&oldid=98921762

Dead Island. (2017). Es.wikipedia.org. Recuperado 23 June 2017, a partir de
https://es.wikipedia.org/w/index.php?title=Dead_Island&oldid=99713009

Historia de los videojuegos. (2017). Es.wikipedia.org. Recuperado 21 June 2017, a partir de
https://es.wikipedia.org/w/index.php?title=Historia_de_los_videojuegos&oldid=99668007.

List of video game genres. (2017). En.wikipedia.org. Recuperado 21 July 2017, a partir de
https://en.wikipedia.org/w/index.php?title=List_of_video_game_genres&oldid=781933781

Survival horror. (2017). Es.wikipedia.org. Recuperado 21 June 2017, a partir de
https://es.wikipedia.org/w/index.php?title=Survival_horror&oldid=98237658.

Tennis for Two. (2017). Es.wikipedia.org. Recuperado 21 June 2017, a partir de
https://es.wikipedia.org/w/index.php?title=Tennis_for_Two&oldid=90859411.

Videojuego. (2017). Es.wikipedia.org. Recuperado 21 June 2017, a partir de
<https://es.wikipedia.org/w/index.php?title=Videojuego&oldid=99963794>

Wikipedia. (2017). Recuperado 22 June 2017, a partir de desde
https://es.wikipedia.org/w/index.php?title=Left_4_Dead_2&oldid=99356674

Resident Evil 7: Biohazard. (2017, 22 de junio). *Wikipedia*. (2017). Recuperado 22 June 2017, a partir de desde
https://es.wikipedia.org/w/index.php?title=Resident_Evil_7:_Biohazard&oldid=100001980

360, X., & (360), L. (2017). *¡Matadlos! ¡A todos!. Meristation*. Recuperado 23 June 2017, a partir de <http://www.meristation.com/xbox-360/left-4-dead-2/analisis-juego/1529629>

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

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

Anon. (2017). Recuperado 10 July 2017, a partir de <http://www.idera.gob.ar/portal/sites/default/files/TrelloTutorialBasico.pdf>

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.

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

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

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.

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>.

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.

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.

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.

Studio. (2017). 2. *el videojuego*. Es.slideshare.net. Recuperado 12 July 2017, a partir de <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*. Recuperado 10 July 2017, a partir de <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. Recuperado 10 July 2017, a partir de <http://www.javiergarzas.com/2011/11/kanban.html>

Cinco cosas buenas y cinco no tan buenas de Unity 3D. (2017). *La leyenda de Darwan*. Recuperado 10 July 2017, a partir de <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. Recuperado 12 July 2017, a partir de <https://www.brainsins.com/es/blog/gamificacion-mecanicas-de-juego/3131>

Studio. (2017). 7. *mecánicas de juego*. Es.slideshare.net. Recuperado 12 July 2017, a partir de <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.

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

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

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

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

9. Créditos

Zombi (<https://www.mixamo.com/#/?page=1&query=zombi&type=Character>) by Mixamo / Free

M60E4 (<https://free3d.com/3d-model/m60e4-79415.html>) by ysup12 / Personal use only License.

UMP (<http://www.cadnav.com/3d-models/model-36014.html>) / Free lincense(Personal and Commercial).

Armchair OSCAR (https://3dsky.org/3dmodels/show/armchair_oscar_1) by 3d-mag / Personal use.

Xerox 3D Model (<https://archive3d.net/?a=download&id=5e1f2d2a>) by Wolf Logan / Personal use.

Screen 3D Model (<https://archive3d.net/?a=download&id=c93a096e>) by Artem Karapetyan / Personal use.

Monitor 3D Model (<https://archive3d.net/?a=download&id=0512e370>) by indcouple / Personal use.

Scotch 3D Model (<https://archive3d.net/?a=download&id=144ead38>) by Wagner Otto / Personal use.

Bed 3D Model (<https://archive3d.net/?a=download&id=c1844281>) by Scharoun / Personal use.

Presentation screen (<https://www.turbosquid.com/3d-models/screen-presentation-max-free/444586>) by TripRay Company / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>)

Table for negotiations Tau (https://3dsky.org/3dmodels/show/stol_dlia_pierieghovorov_tau) by Saturn88 / Personal use.

Projector (<http://www.3dmodelfree.com/models/29930-0.htm>) / Personal use.

UH60 Helicopter (<https://free3d.com/3d-model/uh60-helicopter-47194.html>) by ysup12 / Personal use only.

GAZ-2975 Tiger (<https://free3d.com/3d-model/gaz-2975-tiger-59753.html>) by azlyirnizam / Personal use only

MP5K (<https://free3d.com/3d-model/mp5k-fully-rigged-60158.html>) by yogo361 / Non commercial use

US APC (<https://free3d.com/3d-model/us-apc-24406.html>) by ysup12 / Personal use only License.

Server Rack (<https://free3d.com/3d-model/server-rack-76768.html>) by flevasgr / Non commercial use License.

iMac (<https://free3d.com/3d-model/imac-25593.html>) by storque12 / Personal use only License.

Tactical Flashlight (<https://free3d.com/3d-model/tactical-flashlight-33623.html>) by texasfunk101 / Commercial use License.

Acoustic guitar (<https://free3d.com/3d-model/acoustic-guitar-89551.html>) by Edson / Personal use only.

Lava Lamp (<https://www.turbosquid.com/3d-models/maya-lava-lamp/404512>) by My3DGuy / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

Light (<https://www.turbosquid.com/3d-models/free-lamp-lights-3d-model/532163>) by Essezeta 3d / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

Pillows (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/1044194>) by SatoyaSan / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

Pillow Photorealistic (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/971154>) by TeoRossi / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

Pillow Photorealistic 2 (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/971216>) by TeoRossi / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

Heart Shaped Pillow (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/896901>) by anlu / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

Sniper Rifle KSR-29 (<https://free3d.com/3d-model/sniper-rifle-ksr-29-new-34178.html>) by 3dhaupt / Non comercial use License.

Showcase (<https://archive3d.net/?a=download&id=1807a3b9>) by Frank Evans / Personal use.

Ring towel holder (<http://www.cadnav.com/3d-models/model-35668.html>) / Free (Personal and Commercial).

Towels (<https://archive3d.net/?a=download&id=2d61bb96#>) by Ahmet Tangan / Personal use.

Towel (<https://archive3d.net/?a=download&id=4aaa431b>) by Anton / Personal use.

Toilet paper dispenser (<http://www.cadnav.com/3d-models/model-15629.html>) / Free (Personal and Commercial).

Bathroom corner shelf (<http://www.cadnav.com/3d-models/model-16504.html>) / Free (Personal and Commercial).

Filing Cabinets Locker (<http://www.cadnav.com/3d-models/model-2156.html>) / Free (Personal and Commercial).

Bath towel shelf rack (<http://www.cadnav.com/3d-models/model-27956.html>) / Free (Personal and Commercial).

Stalls (<https://archive3d.net/?a=download&id=984ef7ac>) by Isidoro di Mileto / Personal use.

Shampoo (<https://archive3d.net/?a=download&id=53501fc9>) by Shahrooz Zolghadr / Personal use.

Supermarket cooked food display showcase (<http://www.cadnav.com/3d-models/model-27402.html>) / Free (Personal and Commercial).

Plate of Sandwiches (<http://www.cadnav.com/3d-models/model-32433.html>) / Free (Personal and Commercial).

Sandwich on plate (<http://www.cadnav.com/3d-models/model-31819.html>) / Free (Personal and Commercial)

Hamburgues & french fries (<http://www.cadnav.com/3d-models/model-32512.html>) / Free (Personal and Commercial).

Glass of cognac (<https://www.cgtrader.com/free-3d-models/food/miscellaneous/glass-of-cognac>) by milivoje-zarkovic / Royalty Free License (<https://www.cgtrader.com/pages/terms-and-conditions#royalty-free-license>)

Bottle of purified wáter (<https://www.cgtrader.com/free-3d-models/food/beverage/bottle-of-purified-water>) by mauro-chile / Royalty Free License .

Nutella Chocolate Hazelnut Spread (<https://www.cgtrader.com/free-3d-models/food/miscellaneous/nutella-chocolate-hazelnut-spread>) by Supercigale / Editorial License.

Burger (<https://www.cgtrader.com/free-3d-models/food/miscellaneous/burger-528667a4-fc18-4094-8e4d-2d7dae0686b2>) by andrescmgamma / Royalty Free License.

Bread bread garnish (<https://www.cgtrader.com/free-3d-models/food/miscellaneous/bread-bread-garnish--2>) by edson-lopes / Royalty Free License.

Fruits (<https://www.cgtrader.com/free-3d-models/food/fruit/fruits-594e7e27-827c-453b-bb84-ce714ecf8a19>) by 3dmodelspbry / Royalty Free License.

Soup and spoon (<https://www.cgtrader.com/free-3d-models/food/miscellaneous/soup-and-spoon>) by slatcher / Royalty Free License.

Can beer (<https://www.cgtrader.com/free-3d-models/food/beverage/can-beer>) by lolito / Royalty Free License.

Kithcen set – sweets in jars (<https://www.cgtrader.com/free-3d-models/food/miscellaneous/kitchen-set-sweets-in-jars>) by limen / Royalty Free License.

OFS – soda can (<https://www.cgtrader.com/free-3d-models/food/miscellaneous/ofs-soda-can>) by Ridwan / Royalty Free License.

Bread Free VR / AR / low-poly (<https://www.cgtrader.com/free-3d-models/food/miscellaneous/bread-d27391a5-b572-48cd-b6c1-bb6ddfed74d8>) by crossorder / Editorial License.

Corn Bradroll (<https://www.cgtrader.com/free-3d-models/food/miscellaneous/corn-breadroll>) by marlis / Royalty Free License.

Mornings Donuts And Coffe (<https://www.cgtrader.com/free-3d-models/food/miscellaneous/morning-donuts-and-coffe>) by x-sparta-x / Royalty Free License.

Oil and Vinegar Carafe (<https://www.cgtrader.com/free-3d-models/food/miscellaneous/oil-and-vinegar-carafe-64649e70-0746-4eff-abe1-8bfe769d845c>) by Kike-maldonado-s / Royalty Free License.

Bread Lowpoly Free VR / AR / low-poly (<https://www.cgtrader.com/free-3d-models/food/miscellaneous/bread-lowpoly>) by splitgame / Royalty Free License.

Stew- soup (<https://www.cgtrader.com/free-3d-models/food/miscellaneous/stew-soup>) by c-ferreira / Royalty Free License.

Orange(<https://www.cgtrader.com/free-3d-models/food/fruit/orange-89352190-06fc-4b7e-805b-1d2c95d4942b>) by sultana / Royalty Free License.

Lemon (<https://www.cgtrader.com/free-3d-models/food/fruit/lemon-3d-model-low-poly-vr-ar>) by grader30 / Royalty Free License.

Hamburgues Free VR / AR / low-poly (<https://www.cgtrader.com/free-3d-models/food/miscellaneous/hamburger-755bf2bb-2a1e-46c8-bd6c-7d248ec7f7ef>) by ufukufuk / Royalty Free License.

Plank cheese cheese cutter (<https://www.cgtrader.com/free-3d-models/food/miscellaneous/plank-cheese-cheese-cutter>) by Edson-lopes / Royalty Free License.

Pizza Diavola (<https://www.cgtrader.com/free-3d-models/food/miscellaneous/pizza-diavola>) by fabelar / Royalty Free License.

Bowl of Yakisoba (<https://www.cgtrader.com/free-3d-models/food/miscellaneous/bowl-of-yakisoba-miojo>) by c-ferreira / Royalty Free License.

Donuts (<https://archive3d.net/?a=download&id=734e3fec>) by Rom / Personal use.

Chiken wrap (<https://archive3d.net/?a=download&id=4f04d9ff>) by Jes / Personal use.

Food (<https://archive3d.net/?a=download&id=6c73e683>) by ChiKo / Personal use.

Sushi (<https://archive3d.net/?a=download&id=cc88a94b>) by Kon / Personal use.

Salad (<https://archive3d.net/?a=download&id=72551241>) by Richard / Personal use.

Donuts showcase (<https://archive3d.net/?a=download&id=68094c99>) by Gustav / Personal use.

Bread (<https://archive3d.net/?a=download&id=c4b3cbdd>) by Indrid / Personal use.

Gin with nuts (<https://free3d.com/3d-model/gin-with-nuts-59441.html>) by 3dregenerator / Persona use License.

Male Muscle Anatomy (<http://www.cadnav.com/3d-models/model-36670.html>) / Free (Personal and Commercial).

Human Body Skeleton (<http://www.cadnav.com/3d-models/model-37018.html>) / Free (Personal and Commercial).

Lightweight wheelchair (<http://www.cadnav.com/3d-models/model-33459.html>) / Free (Personal and Commercial).

Hospital room divider screen (<http://www.cadnav.com/3d-models/model-33427.html>) / Free (Personal and Commercial).

Radiography machine (<http://www.cadnav.com/3d-models/model-33423.html>) / Free (Personal and Commercial).

Radiography diagnostic monitor (<http://www.cadnav.com/3d-models/model-33417.html>) / Free (Personal and Commercial).

Medical X-ray monitor (<http://www.cadnav.com/3d-models/model-33416.html>) / Free (Personal and Commercial).

Ultrasound diagnostic machine (<http://www.cadnav.com/3d-models/model-33413.html>) / Free (Personal and Commercial).

Medical radiation equipment (<http://www.cadnav.com/3d-models/model-33409.html>) / Free (Personal and Commercial).

Medical exam stool (<http://www.cadnav.com/3d-models/model-33396.html>) / Free (Personal and Commercial).

Desfibrillator emergency medicine (<http://www.cadnav.com/3d-models/model-33392.html>) / Free (Personal and Commercial).

Hospital beds (<http://www.cadnav.com/3d-models/model-33369.html>) / Free (Personal and Commercial).

Hospital medication cart (<http://www.cadnav.com/3d-models/model-19693.html>) / Free (Personal and Commercial).

Medical machine parts (<http://www.cadnav.com/3d-models/model-19696.html>) / Free (Personal and Commercial).

Eye examination box (<http://www.cadnav.com/3d-models/model-19685.html>) / Free (Personal and Commercial).

Hemodialysis machine (<http://www.cadnav.com/3d-models/model-19475.html>) / Free (Personal and Commercial).

Medince cabinets (<http://www.cadnav.com/3d-models/model-19466.html>) / Free (Personal and Commercial).

Prision Fence Pack (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/600548>) by George Piskas / Royalty Free License.

Banana Colada – Custom Perk Machine (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/1052172>) by madgaz182 / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

Table coffe (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/525162>) by Peter301 / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

Table (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/839096>) by gela motskobili0080 / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

3D ABCD Lounge Chair (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/1172750>) by Designconnected / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

Techno Sofa (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/892324>) by iani_3D / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

Modernbasin (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/340814>) by Roodogg / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

Paramedic Bag (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/834773>) by Cliche Studio / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

Tuba.max (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/226686>) by ARTACTIVE / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

Toothbrush.mac (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/440026>) by Nikola Dechev / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

Tooth Brush (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/448011>) by rusty_hawk / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

Colgate tooth paste (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/463922>) by Nolan Brundson / Editorial uses.

George Nelson Platform Bench (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/1004778>) by optasia3d / Editorial uses.

Locker (basic) (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/505340>) by razor605 / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

Military Case (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/1120427>) by Lekik / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

Box of shotgun shells (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/527502>) by Everett G / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

Granade.max (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/305513>) by SPiTfIRe814 / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

Steyr AUG (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/274610>) by Jonflect / Editorial use allowed. / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

M1 garand (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/759786>) by commander566 / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

Barret m107 (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/588956>) by commander566 / Editorial uses allowed. / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

Flowers (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/726193>) by Sabracon / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

G36 (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/968920>) by Verge Art / Editorial use / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

Gibson Les Paul Guitar – Low Poly (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/654750>) by Riptide099 / Editorial use allowed.

Electric guitar (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/506686>) by jedik18 / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

Radio (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/1127601>) by AndijaAlp / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

Presentation screen (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/444586>) by TripRay Company / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

Scar-L with attachments (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/560208>) by Snail Studios / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

Cell door (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/552899>) by Morty3D / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

M4a1 (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/871182>) by game_ready / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).

AK (<https://www.turbosquid.com/FullPreview/Index.cfm/ID/1080222>) by Sphprott / Royalty Free License (<https://blog.turbosquid.com/royalty-free-license/>).