



Centro Universitario de la Defensa en la Escuela Naval Militar

TRABAJO FIN DE GRADO

*Implementación de las reglas de ataque y compromiso sobre un
escenario de combate táctico mediante realidad virtual*

Grado en Ingeniería Mecánica

ALUMNO: Rafael Mira Criado

DIRECTORES: Francisco Troncoso Pastoriza
Xavier Núñez Nieto

CURSO ACADÉMICO: 2024-2025

UniversidadeVigo



Centro Universitario de la Defensa en la Escuela Naval Militar

TRABAJO FIN DE GRADO

*Implementación de las reglas de ataque y compromiso sobre un
escenario de combate táctico mediante realidad virtual*

Grado en Ingeniería Mecánica
Intensificación en Tecnología Naval
Cuerpo General

Universida_{de}Vigo

RESUMEN

La irrupción de la Industria de Cuarta Generación ha supuesto una transformación significativa en la manera en que las Fuerzas Armadas (FAS) afrontan sus procesos de adiestramiento y preparación. En este nuevo paradigma, la digitalización y el uso de tecnologías avanzadas permiten optimizar y modernizar procedimientos tradicionales, dotando a las unidades de una mayor eficiencia y versatilidad. Entre las tecnologías emergentes que se sitúan en el núcleo de esta revolución, la Realidad Virtual (RV) ocupa un lugar destacado, permitiendo la creación de entornos virtuales inmersivos de alta fidelidad donde se pueden replicar de manera precisa y controlada situaciones de combate real. En el marco de la llamada Armada 4.0, la RV se ha posicionado como una herramienta clave para el desarrollo de capacidades tácticas y la mejora de la toma de decisiones en situaciones críticas. Este Trabajo Fin de Grado se ha centrado precisamente en dicha línea de trabajo, abordando el diseño e implementación de un entorno virtual urbano de combate, enfocado al entrenamiento militar, utilizando RV y simulación mediante inteligencia artificial avanzada. La integración de estas tecnologías permite al usuario experimentar de manera inmersiva escenarios bélicos complejos, maximizando la eficacia del adiestramiento y fortaleciendo la operatividad de las fuerzas.

PALABRAS CLAVE

Realidad Virtual, *Blueprints*, *Unreal Engine*, *Rules of Engagement*, Simulación de combate

AGRADECIMIENTOS

Antes de nada, quiero expresar mi más sincero agradecimiento a mis tutores, Don Francisco Troncoso Pastoriza y Don Xavier Núñez Nieto, por su incansable paciencia, dedicación y apoyo a lo largo de este proyecto. Su guía constante ha sido un pilar fundamental durante el desarrollo de este trabajo.

Asimismo, deseo agradecer a mi camareta 217, y en especial a Antonio Guerrero Vázquez y Francisco Javier Pavón Jarava, por haber sido un remanso de alegría y desconexión en los momentos más intensos de estudio. Su apoyo y compañerismo han hecho más llevaderas las largas jornadas de esfuerzo y las cortas noches de descanso.

Por último, y no por ello menos importante, quiero agradecer profundamente a mi familia, que ha estado a mi lado en todo momento. Gracias por confiar en mí, por animarme cuando más lo necesitaba y por ser el mayor motor de motivación a lo largo de este camino.

CONTENIDO

Contenido	1
Índice de Figuras	3
1 Introducción.....	9
1.1 Descripción del apartado.....	9
1.2 Antecedentes	9
1.3 Motivación	11
1.4 Objetivos	11
1.5 Estructura del trabajo	12
2 Estado del arte	13
2.1 Descripción del apartado.....	13
2.2 Conceptos generales de la RV	13
2.2.1 Realidad virtual, aumentada y mixta	13
2.2.2 Tipos y características de realidad virtual.....	14
2.2.3 Realidad virtual aplicada al ámbito militar.....	16
2.3 Combate en población.....	19
2.3.1 Características y retos del MOUT	19
2.3.2 Control del territorio y defensa en áreas urbanas	21
2.3.3 Reglas de enfrentamiento y ámbito de aplicación	25
2.4 Programación	28
2.4.1 Concepto general de la programación en UE	28
2.4.2 Tipos y características.....	29
2.4.3 Otras funcionalidades y servicios	31
2.4.4 Inteligencia artificial.....	32
2.5 Software y equipos.....	34
2.5.1 Software utilizado.....	34
2.5.2 Equipos y material	36
3 Desarrollo	38
3.1 Descripción del apartado.....	38
3.2 Escenario	39
3.2.1 Concepto del escenario	39
3.2.2 Creación del escenario desde UE.....	40
3.2.3 Modelado y creación del escenario final	41
3.3 Creación de enemigos y soldados del equipo del personaje	43

3.3.1 Creación inicial de BP_Player y BP_EnemyBase	43
3.3.2 Introducción a los BT y BB	45
3.3.3 Patrullas y estados del enemigo	49
3.3.4 Introducción a la percepción del entorno del enemigo	52
3.3.5 Introducción al Enviroment Query Sistem	57
3.3.6 Sistema de salud	64
3.3.7 División de enemigos	70
3.3.8 Combate y comunicación del enemigo	94
3.3.9 Creación de equipos.....	96
3.4 Integración en un escenario para combate poblacional en VR	97
4 Resultados	102
4.1 Descripción del apartado.....	102
4.2 Escenario.....	102
4.3 Funciones de enemigos	103
4.4 Integración en un escenario para aplicar procedimiento MOUT	114
5 Conclusiones y líneas futuras	118
5.1 Descripción del apartado.....	118
5.2 Conclusiones	118
5.2.1 Diseño del escenario	118
5.2.2 Programación e implementación de acciones	118
5.2.3 Integración de la RV en el escenario	119
5.2.4 Conclusión final.....	119
5.3 Líneas futuras	119
5.3.1 A corto plazo.....	119
5.3.2 A largo plazo.....	120
6 Bibliografía.....	121
Anexo I: Implicaciones Sociales, y/o Económicas, y/o Ambientales	125
Anexo II: Reflexiones Éticas y Sociales	127
Anexo III: Bloques de código del AIController	129
Anexo IV: Bloques de código del sistema de daño	135
Anexo IV: Bloques de código de enemigos	138
Anexo V: Bloques de código de Behavior Trees	147

ÍNDICE DE FIGURAS

Figura 1-1 Evolución de la Industria a lo largo de la historia [3].	10
Figura 1-2 <i>Starlink</i> como medio de comunicaciones empleados en buques militares [4].	11
Figura 2-1 Ejemplos de realidad virtual no inmersiva [7].	15
Figura 2-2 Ejemplos de realidad virtual semi-inmersiva [8].	15
Figura 2-3 Ejemplo de realidad virtual inmersiva [10].	15
Figura 2-4 <i>Kriegsspiel</i> , el juego de mesa que entrenó a los prusianos [11].	16
Figura 2-5 Simulador Multi-Vehículo desarrollado por Simumak [15].	17
Figura 2-6 Simulador PC-21 de la AGA [18].	17
Figura 2-7 Centro de Adiestramiento de Seguridad Interior en Rota [19].	18
Figura 2-8 Simulador de armas pesada de la Guardia Nacional de Nueva Jersey [21].	19
Figura 2-9 Infraestructuras del campo de prácticas de combate en población de Figueirido [24].	20
Figura 2-10 Zona destinada a combate urbano en San Gregorio [23].	21
Figura 2-11 Movimiento en exterior por pelotón [25].	21
Figura 2-12 Ejemplo cobertura continua en exteriores [25].	22
Figura 2-13 Cordón exterior e interior [25].	22
Figura 2-14 Ejemplo de cómo hacer y como no hacer entrada a una habitación [25].	23
Figura 2-15 Forma correcta y estipulada de limpiar una habitación una vez se entra [25].	24
Figura 2-16 Forma correcta de actuar frente a una esquina [25].	24
Figura 2-17 Manera de acceder a un pasillo en forma de “L” [25].	25
Figura 2-18 Jerarquía del requerimiento de las REN a nivel general [26].	26
Figura 2-19 Soldado del Ejército de Tierra dando órdenes a un perro de raza Pastor Belga [27].	27
Figura 2-20 Planteamiento estructurada de la aplicación de las ROE [26].	28
Figura 2-21 Ejemplo de programación con BP en UE [30].	29
Figura 2-22 Interfaz de UE al seleccionar nuevo BP (Autoría propia).	30
Figura 2-23 Escenario generado con Lumen [32].	31
Figura 2-24 Ejemplo de <i>MetaHuman</i> [33].	32
Figura 2-25 Interfaz de la página <i>Quixel MegaScans</i> [34].	32
Figura 2-26 NPC [36].	33
Figura 2-27 Ejemplo de un BT en UE [40].	34
Figura 2-28 Escenario de <i>Fortnite</i> y logotipo de <i>Epic Games</i> [41] [42].	35
Figura 2-29 Interfaz de la sala general de control de <i>Steam VR Home</i> [45].	35
Figura 2-30 Pantalla de ordenador, ratón y teclado del equipo de trabajo (Autoría propia).	36
Figura 2-31 Visor HTC Focus Vision y mandos de control [48].	37

Figura 2-32 Tarjeta Gráfica NVIDIA GeForce RTX (Autoría propia).....	37
Figura 3-1 Esquema general de la consecución del trabajo (Autoría propia).	38
Figura 3-2 Situación del conflicto entre Redland y Greenland (generado por IA).	39
Figura 3-3 Interfaz de entrada de UE y sus distintos <i>templates</i> (Autoría propia).	40
Figura 3-4 <i>Third Person Template</i> por defecto de UE (Autoría propia).	41
Figura 3-5 Plantilla final del escenario (Autoría propia).....	42
Figura 3-6 Apariencia del <i>Static Mesh</i> por defecto del <i>ThirdPersonCharacter</i> (Autoría propia). ..	43
Figura 3-7 Lógica de <i>ABP_Manny</i> que usan los enemigos de la simulación (Autoría propia).	44
Figura 3-8 Función de persecución en el <i>Event Graph</i> del <i>BP_EnemyBase</i> (Autoría propia).....	45
Figura 3-9 <i>BB_EnemyBase</i> con las variables de toda la programación (Autoría propia).	45
Figura 3-10 <i>Event Graph</i> del <i>AIController</i> (Autoría propia).	46
Figura 3-11 BT básico de persecución (Autoría propia).	46
Figura 3-12 <i>BTT_DefaultAttack</i> (Autoría propia).	47
Figura 3-13 <i>Retarget</i> de la animación de ataque del enemigo (Autoría propia).	47
Figura 3-14 Machete que lleva equipado el enemigo en su <i>socket</i> (Autoría propia).	48
Figura 3-15 BT del <i>BP_EnemyBase</i> añadiendo nuevos nodos (Autoría propia).	49
Figura 3-16 Lógica de nodos usados para que el enemigo patrulle en bucle (Autoría propia).	49
Figura 3-17 <i>BPI_EnemyAI</i> y representación visual de la <i>BP_PatrolRoute</i> (Autoría propia).	50
Figura 3-18 Establecimiento de valores de velocidad en <i>BP_EnemyBase</i> (Autoría propia).	51
Figura 3-19 <i>Enum</i> que contiene los estados comunes de enemigos (Autoría propia).....	51
Figura 3-20 BT con <i>AttackingState</i> y <i>PassiveState</i> (Autoría propia).....	52
Figura 3-21 Implementación de la función <i>SetStateAsPassive</i> (Autoría propia).	52
Figura 3-22 Implementación del <i>AIPerception</i> en el <i>AIC_EnemyBase</i> (Autoría propia)	53
Figura 3-23 Visualización de los radios de los sentidos del enemigo [53].	53
Figura 3-24 Primera parte de la lógica de la función <i>CanSenseActor</i> (Autoría propia).....	54
Figura 3-25 Segunda parte de la lógica de la función <i>CanSenseActor</i> (Autoría propia).	54
Figura 3-26 lógica inicial común para asignar funciones a los estímulos (Autoría propia).....	55
Figura 3-27 Función <i>HandleSenseSight</i> (Autoría propia).	55
Figura 3-28 Función <i>GetCurrentState</i> (Autoría propia).....	56
Figura 3-29 Función <i>HandleSenseSound</i> (Autoría propia).	56
Figura 3-30 Función <i>SetStateAsInvestigating</i> (Autoría propia).	56
Figura 3-31 Lógica del estado <i>Investigating</i> implementada en <i>BT_EnemyBase</i> (Autoría propia). .	57
Figura 3-32 <i>BS_Strafing</i> (Autoría propia).....	58
Figura 3-33 Establecimiento de la variable <i>Direction</i> (Autoría propia).	58
Figura 3-34 Establecimiento de la variable <i>Is Focusing Target</i> (Autoría propia).	59
Figura 3-35 Nueva lógica en el estado <i>Idle</i> del <i>Locomotion</i> (Autoría propia).....	59

Figura 3-36 Nueva lógica en el estado <i>Walk/Run</i> del <i>Locomotion</i> (Autoría propia).	59
Figura 3-37 Carpeta de EQS y el <i>Query Graph</i> de <i>EQS_Strafe</i> (Autoría propia).	60
Figura 3-38 <i>EQS_Strafe</i> (Autoría propia).	60
Figura 3-39 <i>EQS_Context_AttackTarget</i> (Autoría propia).	61
Figura 3-40 Ramificación del <i>Behavior Tree</i> de la secuencia <i>Strafe</i> (Autoría propia).	61
Figura 3-41 Asignación de <i>Attack Radius</i> y <i>Defend Radius</i> a la BB (Autoría propia).	62
Figura 3-42 <i>BTD_IsAtPositiveDistance</i> (Autoría propia).	62
Figura 3-43 <i>BTT_MoveToIdealRange</i> (Autoría propia).	63
Figura 3-44 Actualización de la ramificación <i>Strafe</i> (Autoría propia).	63
Figura 3-45 <i>EQS_FindCover</i> en el mundo gracias al <i>Testing Pawn</i> (Autoría propia).	64
Figura 3-46 Lógica de <i>Find Cover</i> en el <i>BT_EnemyBase</i> (Autoría propia).	64
Figura 3-47 Carpeta con los elementos del sistema de daño (Autoría propia).	64
Figura 3-48 Funciones de <i>BPI_Damage</i> (Autoría propia).	65
Figura 3-49 Variables de la estructura <i>E_DamageInfo</i> (Autoría propia).	65
Figura 3-50 Enumeradores <i>E_HitResponse</i> y <i>E_TipoDeDaño</i> respectivamente (Autoría propia).	65
Figura 3-51 Función <i>Heal</i> (Autoría propia).	66
Figura 3-52 Macro <i>CanBeDamaged</i> (Autoría propia).	66
Figura 3-53 Inicio de la función <i>TakeDamage</i> (Autoría propia).	67
Figura 3-54 Continuación de la función <i>TakeDamage</i> (Autoría propia).	67
Figura 3-55 Función para hacer daño dentro de <i>BP_Player</i> (Autoría propia).	68
Figura 3-56 Función <i>GetPercent</i> (Autoría propia).	68
Figura 3-57 Lógica para establecer <i>Widget</i> en el mundo (Autoría propia).	69
Figura 3-58 Comienzo de la función <i>Die</i> (Autoría propia).	69
Figura 3-59 Continuación de la función <i>Die</i> (Autoría propia).	70
Figura 3-60 Lógica de <i>DamageResponse</i> (Autoría propia).	70
Figura 3-61 BT final del <i>BP_EnemyBase</i> (Autoría propia).	71
Figura 3-62 <i>BP_EnemyMelee</i> (Autoría propia).	71
Figura 3-63 Continuación de <i>BPI_EnemyAI</i> (Autoría propia).	72
Figura 3-64 Carpeta de BT usados (Autoría propia).	72
Figura 3-65 Uso de una variable para ejecutar todos los BT (Autoría propia).	73
Figura 3-66 Primera parte de <i>CheckIfForgottenSeenActor</i> (Autoría propia).	73
Figura 3-67 Segunda parte de <i>CheckIfForgottenSeenActor</i> (Autoría propia).	74
Figura 3-68 Función <i>HandleForgotActor</i> (Autoría propia).	74
Figura 3-69 Función <i>HandleLostSight</i> (Autoría propia).	74
Figura 3-70 Ramificación <i>GoToLastKnownLocation</i> (Autoría propia).	75

Figura 3-71 Ramificación <i>Seeking</i> (Autoría propia).	75
Figura 3-72 <i>EQS_Seeking</i> (Autoría propia).	76
Figura 3-73 Ramificación <i>Finish Seeking</i> (Autoría propia).	76
Figura 3-74 Primera parte función <i>KnifeShortAttack</i> (Autoría propia).	77
Figura 3-75 Segunda parte función <i>KnifeShortAttack</i> (Autoría propia).	77
Figura 3-76 Diferencia de lógica del <i>KnifeLongAttack</i> (Autoría propia).	78
Figura 3-77 Función personalizada <i>SphereDamage</i> (Autoría propia).	78
Figura 3-78 Primera parte función <i>JumpToAttackTarget</i> (Autoría propia).	79
Figura 3-79 Función <i>CalculateFutureActorLocation</i> (Autoría propia).	79
Figura 3-80 Segunda parte función <i>JumpToAttackTarget</i> (Autoría propia).	80
Figura 3-81 Ramificación <i>Attack</i> del <i>BP_EnemyMelee</i> (Autoría propia).	80
Figura 3-82 <i>BTS_UpdateDistanceToTarget</i> (Autoría propia).	81
Figura 3-83 <i>Sequence Attack</i> final del <i>BP_EnemyMelee</i> (Autoría propia).	81
Figura 3-84 Primera parte de <i>BTT_MeleeAttack</i> en <i>BP_EnemyMelee</i> (Autoría propia).	82
Figura 3-85 Segunda parte de <i>BTT_MeleeAttack</i> en <i>BP_EnemyMelee</i> (Autoría propia).	82
Figura 3-86 Tercera parte de <i>BTT_MeleeAttack</i> en <i>BP_EnemyMelee</i> (Autoría propia).	83
Figura 3-87 BT final del <i>BP_EnemyBase</i> (Autoría propia).	83
Figura 3-88 <i>BS_Jog</i> (Autoría propia).	84
Figura 3-89 Acople del arma en <i>BP_EnemyRanged</i> (Autoría propia).	84
Figura 3-90 Acople del arma en <i>BP_EnemyRanged</i> (Autoría propia).	85
Figura 3-91 Secuencia <i>Evade</i> (Autoría propia).	85
Figura 3-92 <i>EQS_FindIdealRange</i> (Autoría propia).	86
Figura 3-93 Primera parte de <i>Attack</i> en <i>BP_EnemyRanged</i> (Autoría propia).	86
Figura 3-94 Segunda parte de <i>Attack</i> en <i>BP_EnemyRanged</i> (Autoría propia).	87
Figura 3-95 <i>BTD_CanSeeTarget</i> (Autoría propia).	87
Figura 3-96 <i>BTT_Heal</i> (Autoría propia).	88
Figura 3-97 <i>BTD_IsHealthBelowWhatItShould</i> (Autoría propia).	88
Figura 3-98 Ramificación de buscar refugio y curarse (Autoría propia).	89
Figura 3-99 BT final del <i>BP_EnemyBase</i> (Autoría propia).	89
Figura 3-100 BT final del <i>BP_FriendTeam</i> (Autoría propia).	90
Figura 3-101 Secuencia <i>Attack</i> para <i>BP_FriendTeam</i> (Autoría propia).	90
Figura 3-102 <i>BP_PatrolEnemy</i> (Autoría propia).	91
Figura 3-103 <i>BP_ShooterLongRangedEnemy</i> (Autoría propia).	91
Figura 3-104 <i>BT_ShooterLongRangeEnemy</i> (Autoría propia).	92
Figura 3-105 <i>Static Mesh</i> de <i>BP_SurrenderEnemy</i> y <i>BP_RunAwayEnemy</i> (Autoría propia).	93
Figura 3-106 Primera parte de la lógica de escapar (Autoría propia).	93

Figura 3-107 Segunda parte de la lógica de escapar (Autoría propia).	93
Figura 3-108 <i>ReserveAttackToken</i> (Autoría propia).	94
Figura 3-109 <i>ReturnAttackToken</i> (Autoría propia).	94
Figura 3-110 Función <i>StoreAttackToken</i> (Autoría propia).	95
Figura 3-111 Función <i>AttackEnd</i> (Autoría propia).	95
Figura 3-112 Función <i>OnSameTeam</i> (Autoría propia).	96
Figura 3-113 Lógica implementada en <i>HandleSenseSight</i> (Autoría propia).	97
Figura 3-114 Escenario integrado (Autoría propia).	97
Figura 3-115 Civil 1 (Autoría propia).	98
Figura 3-116 Civil 2 (Autoría propia).	98
Figura 3-117 <i>Mesh</i> de <i>BP_Player</i> y soldados de operaciones especiales (Autoría propia).	99
Figura 3-118 <i>Mesh</i> de <i>BP_EnemyBase</i> (Autoría propia).	99
Figura 3-119 HK G36E y FN-9 (Autoría propia).	100
Figura 3-120 <i>Widget</i> de inicio (Autoría propia).	101
Figura 3-121 Uso de gafas Focus Vision y escenario dentro de la simulación (Autoría propia).	101
Figura 4-1 Escenario para combate en población con enemigos y civiles (Autoría propia).	102
Figura 4-2 Patrullaje (Autoría propia).	103
Figura 4-3 <i>AIPerception</i> del <i>BP_EnemyBase</i> (Autoría propia).	104
Figura 4-4 <i>Strafing</i> del <i>BP_EnemyMelee</i> (Autoría propia).	104
Figura 4-5 Ataques <i>BP_EnemyMelee</i> (Autoría propia).	105
Figura 4-6 Acción Investigating (Autoría propia).	105
Figura 4-7 <i>Seeking</i> (Autoría propia).	106
Figura 4-8 Uso de <i>NavLinks</i> (Autoría propia).	107
Figura 4-9 Función <i>FindCover</i> y <i>Heal</i> (Autoría propia).	107
Figura 4-10 Recibir y hacer daño (<i>TakeDamage</i>) (Autoría propia).	108
Figura 4-11 <i>BP_Player</i> con arma (Autoría propia).	108
Figura 4-12 <i>EQS_FindIdealRange</i> (Autoría propia).	109
Figura 4-13 <i>BP_ShooterLongRange</i> (Autoría propia).	109
Figura 4-14 <i>BP_FriendTeam</i> (Autoría propia).	110
Figura 4-15 <i>BP_RunAwayEnemy</i> (Autoría propia).	111
Figura 4-16 <i>BP_SurrenderEnemy</i> (Autoría propia).	111
Figura 4-17 Combate conjunto a través de sistema de comunicación (Autoría propia).	112
Figura 4-18 Combate entre enemigos de diferentes equipos (Autoría propia).	113
Figura 4-19 Captura del objetivo de interés (Autoría propia).	113
Figura 4-20 Mapa táctico de un ejercicio (Autoría propia).	114

Figura 4-21 Soldados de operaciones especiales (Autoría propia).	115
Figura 4-22 Civil encubierto contra enemigo de <i>Redland</i> (Autoría propia).	115
Figura 4-23 Francotirador (Autoría propia).	116
Figura 4-24 Infraestructura del <i>BP_TargetEnemy</i> (Autoría propia).	116
Figura 4-25 Civiles por el mapa (Autoría propia).	117
Figura 4-26 Captura de pantalla de lo que el <i>VRPawn</i> ve con las gafas (Autoría propia).	117

1 INTRODUCCIÓN

1.1 Descripción del apartado

En este apartado introductorio se expone la motivación detrás de este proyecto, analizando diferentes aspectos clave. En primer lugar, se analiza desde una perspectiva general y, posteriormente, se profundiza en su aplicación específica dentro del ámbito de las FAS, justificando así su desarrollo. Se presentarán los antecedentes, la relevancia del estudio y los objetivos que se persiguen. Por último, se detallará la estructura adoptada en esta memoria, proporcionando una visión clara del enfoque seguido en la implementación de reglas de ataque y compromiso en un escenario de combate táctico mediante Realidad Virtual (RV).

1.2 Antecedentes

La tecnología 4.0, también conocida como la *Cuarta Revolución Industrial*, representa un hito en la evolución de las tecnologías digitales. Integra sistemas ciberfísicos, inteligencia artificial (IA), *Internet de las Cosas* (IoT) y análisis de datos en tiempo real. Esta cuarta revolución industrial está precedida por la primera de ellas que, en el siglo XVIII, marcó la transición de la producción manual a la mecanizada con la máquina de vapor y el uso del carbón. La segunda, en el siglo XIX, introdujo la electricidad y la producción en masa con la línea de ensamblaje y el motor de combustión. La tercera, en el siglo XX, se basó en la automatización, la informática y la digitalización con el auge de los ordenadores e Internet (Figura 1-1).

El término Industria 4.0 se introdujo en 2011 durante la Feria de Hannover (Alemania), donde se presentó como una estrategia nacional para modernizar y digitalizar las industrias manufactureras. Su desarrollo se basa en los avances de las tres revoluciones industriales previas. La tecnología 4.0 combina estos avances con el poder de la conectividad, creando sistemas interconectados y autónomos capaces de adaptarse y optimizarse por sí mismos [1].

La Industria 4.0 ha transformado significativamente las FAS, impulsando la modernización de sus estructuras y operaciones a través de la digitalización y la automatización. La implementación de tecnologías como la IA, *Big Data* y la computación en la nube ha permitido optimizar la toma de decisiones, mejorar la eficiencia operativa y reducir la carga administrativa en múltiples áreas estratégicas.

Uno de los principales impactos de la Industria 4.0 en las FAS es el desarrollo de sistemas autónomos y robotizados, que incluyen desde drones para reconocimiento y vigilancia hasta vehículos terrestres no tripulados que pueden realizar misiones de exploración o desminado sin poner en riesgo la vida del personal. Estas innovaciones han mejorado significativamente la capacidad operativa de las unidades

militares, proporcionando mayor precisión y rapidez en situaciones de combate y defensa. Además, permiten una mayor cobertura en terrenos hostiles o de difícil acceso, optimizando las misiones de reconocimiento. Su implementación también ha reducido la carga operativa del personal, permitiéndole concentrarse en tareas estratégicas más complejas [2].

La conectividad también ha jugado un papel crucial en la transformación del mundo militar. La introducción de redes de comunicación seguras y en tiempo real ha facilitado la coordinación entre unidades en distintos puntos del campo de batalla. Gracias al IoT, la ciberseguridad avanzada, satélites *Starlink* (Figura 1-2) y nubes de traspaso de datos 5G los buques, vehículos y personal del ejército han optimizado sus procedimientos a la hora de traspasar información táctica. El entrenamiento y la formación del personal militar también han evolucionado con la Industria 4.0. La RV y la realidad aumentada han facilitado la creación de entornos de simulación altamente realistas, donde los soldados pueden practicar maniobras tácticas, entrenarse en el manejo de vehículos y equipos, o enfrentarse a escenarios de combate sin riesgos físicos. Estos métodos permiten un aprendizaje más dinámico y efectivo, reduciendo costos y aumentando la preparación de las tropas [2].

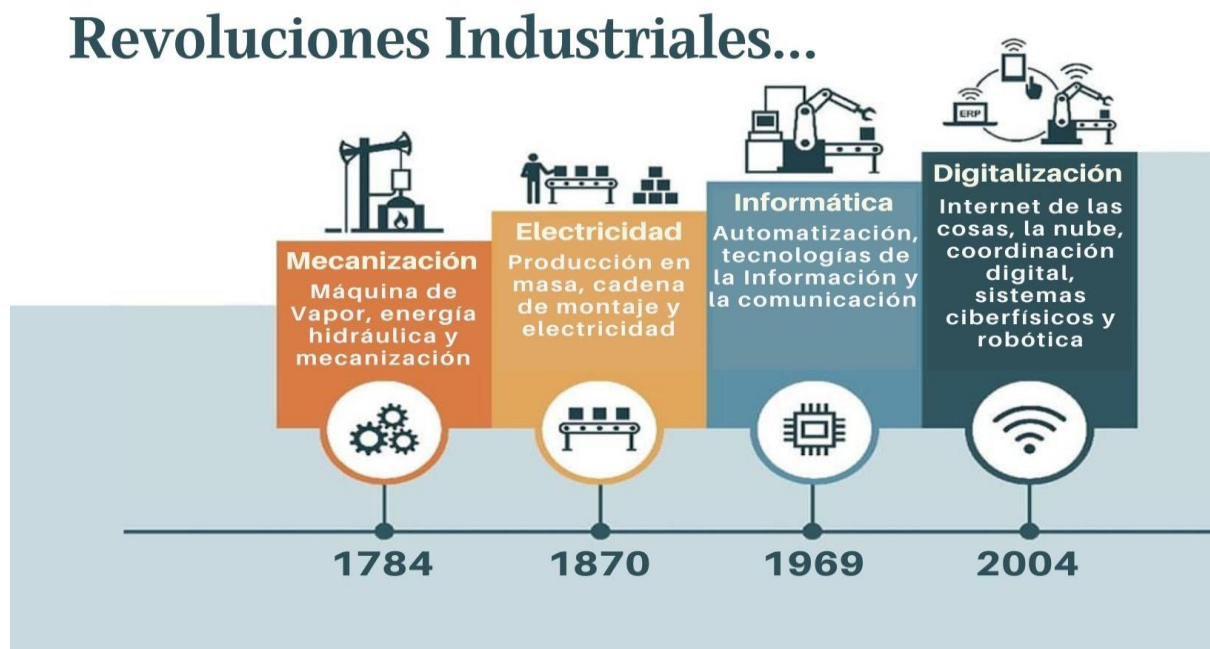


Figura 1-1 Evolución de la Industria a lo largo de la historia [3].

Desde el punto de vista estratégico, la toma de decisiones ha mejorado gracias al uso de IA y análisis de datos en tiempo real. Los sistemas avanzados de gestión de información pueden procesar grandes volúmenes de datos provenientes de múltiples fuentes, permitiendo a los mandos militares anticiparse a amenazas, evaluar riesgos y diseñar estrategias más efectivas en escenarios de conflicto. Si bien los avances tecnológicos ofrecen múltiples beneficios, también presentan desafíos, especialmente en términos de ciberseguridad y ética militar. La dependencia de sistemas digitales y automatizados expone a las FAS a posibles ataques cibernéticos, lo que requiere el desarrollo continuo de estrategias de defensa en este ámbito. Además, el uso de IA en la toma de decisiones militares plantea dilemas sobre el grado de autonomía que deben tener los sistemas automatizados en situaciones de combate.

La Industria 4.0 afecta la forma en que las FAS operan y se preparan para el futuro. La adopción de tecnologías emergentes ha permitido optimizar recursos, mejorar la seguridad y aumentar la capacidad de respuesta ante amenazas globales. Sin embargo, su implementación requiere una adaptación constante a los nuevos desafíos, asegurando un equilibrio entre la innovación tecnológica y la seguridad estratégica [2].



Figura 1-2 Starlink como medio de comunicaciones empleados en buques militares [4].

1.3 Motivación

La importancia de esta industria en el ámbito militar y su relevancia en las FAS han quedado claramente establecidas. Es fundamental continuar mejorando los medios disponibles para el ejército con el fin de simular escenarios de combate que sumerjan al soldado en situaciones realistas, donde su capacidad de reacción y toma de decisiones sea clave para enfrentar posibles escenarios en zonas de operaciones.

En España, el cuerpo de Infantería de Marina ya emplea simuladores de RV para recrear entornos tácticos en los que el soldado debe actuar y tomar decisiones en función de la situación en la que se encuentre. Por ello, el realismo del escenario es un aspecto crucial, ya que debe ofrecer una amplia variedad de situaciones dinámicas en las que cada enfrentamiento implique la aplicación de reglas de combate y protocolos de actuación.

En este contexto, la programación de la IA de los enemigos jugará un papel fundamental en el desarrollo del trabajo. La combinación de una IA avanzada con una inmersión realista en un entorno táctico será la base para una implementación efectiva de las reglas de enfrentamiento en combate.

1.4 Objetivos

Con el objetivo principal de contribuir a la modernización de las FAS y, en particular, al adiestramiento táctico del alumnado de la Escuela Naval Militar, el presente trabajo desarrolla una plataforma de simulación en RV que permite la ejecución de ejercicios de combate urbano mediante la aplicación de inteligencia artificial y reglas de enfrentamiento en un entorno virtual controlado.

Para alcanzar esta meta general, se han establecido los siguientes objetivos complementarios, que, en conjunto, han permitido su consecución:

- Crear una inteligencia artificial avanzada que dota a los enemigos de la capacidad de actuar de manera realista e impredecible, incorporando comportamientos tácticos como búsqueda de cobertura, patrullaje, detección del jugador y reacción a amenazas.
- Desarrollar un entorno de RV funcional y operativo, que permita al usuario experimentar situaciones de combate urbano dinámicas, aplicando los procedimientos doctrinales de adiestramiento y facilitando la toma de decisiones en un entorno seguro.

Adicionalmente, y más allá de lo previsto inicialmente, se ha logrado incorporar una serie de objetivos de valor añadido que refuerzan la calidad del trabajo. Entre ellos destacan la exportación de la lógica a un entorno realista que reproduce un escenario de combate urbano poblacional, y la integración de armamento y enemigos con una estética alineada a las doctrinas de combate y al material empleado habitualmente por las FAS, lo que incrementa de manera significativa la inmersión y el valor formativo de la simulación. También se han incorporado *Widgets* orientativos que guían al usuario durante el trabajo.

1.5 Estructura del trabajo

La estructura que se seguirá durante la memoria del trabajo será la presentada en el índice de contenido:

1. **Introducción y objetivos:** Se describen los detalles previos junto con la motivación del proyecto y sus antecedentes. Se establecen los objetivos que se pretenden alcanzar durante el mismo.
2. **Estado del arte:** Se enfoca en una descripción del ámbito en el que se desarrolla el trabajo, facilitando así la comprensión del desarrollo y los resultados posteriores. Se centra por un lado en el ámbito militar, describiendo aquellos conceptos que resultan interesantes para establecer una base, y en el ámbito civil, incluyendo el software y equipo empleado durante la ejecución, así como una introducción a la programación con el programa *Unreal Engine* (UE).
3. **Desarrollo:** Se describe de manera detallada todo el proceso de ejecución del proyecto, estructurándolo en las distintas partes que lo comprenden.
4. **Resultados:** Se exponen los productos obtenidos de la ejecución y el desarrollo del proyecto, fruto del cual obtenemos la IA de los enemigos, para su correcta aplicación a cualquier escenario de combate.
5. **Conclusiones y líneas futuras:** Se realiza un análisis de los dos puntos anteriores, incluyendo posibles aplicaciones futuras del proyecto y en concreto el simulador.

2 ESTADO DEL ARTE

2.1 Descripción del apartado

En este apartado se busca establecer una base de conocimientos que facilite la comprensión del resto del proyecto y lo contextualice, permitiendo entender cómo se ha llegado al desarrollo de este Trabajo fin de grado. Se comenzará con un apartado en el que se describan aspectos generales de la RV, posteriormente se abordará un apartado que tratará las características generales y los conceptos básicos del combate en población, así como las reglas de enfrentamiento. Finalmente, se profundizará en la programación con *Blueprints* (BP) y se describirán los equipos y el software utilizados a lo largo de la realización del trabajo.

2.2 Conceptos generales de la RV

En un mundo donde la tecnología 4.0 avanza de manera exponencial, la manera en que interactuamos con entornos digitales ha evolucionado significativamente. En este contexto, la RV, aumentada y mixta han surgido como herramientas clave en diversos ámbitos, transformando desde el entretenimiento hasta la educación y el entrenamiento profesional. Sin embargo, para comprender su impacto y aplicaciones, es fundamental diferenciar estos conceptos y analizar sus características, especialmente en sectores donde su implementación representa un cambio estratégico, como en el ámbito militar

2.2.1 Realidad virtual, aumentada y mixta

La realidad virtual, mixta y aumentada son tres términos complicados y fácilmente confundibles, pero su diferencia radica en sus distintas aplicaciones e impacto en sus respectivas áreas de trabajo. La RV crea entornos totalmente digitales, lo que puede limitar la interacción social. La realidad aumentada añade elementos digitales al mundo real, facilitando usos prácticos en educación, diseño y medicina. La realidad mixta combina ambos, permitiendo la coexistencia e interacción de objetos virtuales y reales en tiempo real. Cada tecnología tiene sus ventajas y desafíos, y la disputa reside en cuál será la más dominante y beneficiosa en el futuro.

Este trabajo se enmarca en el ámbito de la RV, cuya definición según la RAE es la “representación de escenas o imágenes de objetos producida por un sistema informático, que da la sensación de su existencia real”. La RV, vista desde una perspectiva conceptual es una tecnología que utiliza dispositivos como gafas especiales para sumergir a los usuarios en entornos completamente generados por ordenadores. A través de estos dispositivos, las personas pueden experimentar y explorar mundos digitales en 3D, que pueden ser simulaciones de lugares reales o escenarios completamente ficticios. Se utiliza en una variedad de aplicaciones, desde videojuegos y entretenimiento hasta

entrenamiento militar, educación y terapia médica. La clave de esta es proporcionar una experiencia inmersiva que haga que los usuarios sientan que están "dentro" del mundo digital, interactuando con él de manera natural y envolvente [5].

Cabe resaltar que este trabajo de fin de grado tiene como objeto continuar con la integración de estas tecnologías en las FAS para su desarrollo y evolución proporcionando un entorno de entrenamiento seguro y eficiente mediante simulaciones. Además, permite recrear escenarios complejos, optimizar la coordinación entre unidades y ofrecer evaluaciones con retroalimentación en tiempo real. Asimismo, los programas de entrenamiento pueden adaptarse a necesidades específicas, lo que garantiza una preparación más efectiva de los soldados para afrontar los desafíos en el escenario táctico. En el ámbito de la Armada, ya se hacen uso de simuladores para recrear el estudio de estrellas o el ataque y navegación costera de un barco en RV, para simular situaciones de riesgo donde el personal pueda entrenarse sin costes materiales y personales y poder desempeñar de una manera más eficiente sus funciones.

2.2.2 Tipos y características de realidad virtual

Ya ha quedado definido el concepto y definición de la RV, pero es de importancia entender que características posee y que tipos según su inmersión y funcionalidad existen. Aunque a lo largo de lo anteriormente mencionado ya se han resaltado palabras como inmersiva, retroalimentación o interactividad es de interés entrar en estas, para establecer de una forma más clara las bases de la RV y el porqué de su integración en la sociedad actual:

- **Interactividad:** La RV permite al usuario interactuar con su entorno virtual en distintos niveles, dependiendo del tipo y la tecnología utilizada. Esta interacción puede ser activa, como manipular objetos virtuales, moverse libremente por el espacio simulado, o incluso colaborar con otros usuarios dentro del entorno. En el caso del trabajo se abordarán estos tres aspectos haciendo uso del manejo y carga de armamento, interacción resolutive frente a enemigos y el movimiento libre del personaje principal por el escenario.
- **Inmersión:** Hace referencia a la sensación de estar completamente inmerso en un entorno simulado. Esta inmersión puede ser total o parcial, y depende de factores como la calidad visual, la precisión del audio espacial y la capacidad del sistema para rastrear los movimientos del usuario en tiempo real. Cuanto mayor es el realismo y la interacción del entorno virtual, más se potencia la inmersión, haciendo que el usuario perciba el entorno simulado como una extensión de la realidad física.
- **Retroalimentación:** Los sistemas avanzados de RV integran diversos tipos de retroalimentación para enriquecer la experiencia del usuario. Esto incluye retroalimentación háptica, que recrea la sensación del tacto; audio 3D, que mejora el realismo con sonidos direccionales; y efectos visuales dinámicos que reaccionan a las acciones del usuario en tiempo real.
- **Adaptabilidad:** Los entornos de RV ofrecen un alto grado de personalización y pueden ser configurados para alcanzar metas específicas. Esto hace posible su utilización en numerosos ámbitos, tales como la educación, la salud, la industria y el entretenimiento, adaptándose tanto a las necesidades y preferencias de los usuarios como a las exigencias del proyecto [6].

Habiéndose destacado las anteriores características, cabe reflejar los distintos tipos de RV actuales, los cuales ofrecen al usuario diferentes experiencias en función de su finalidad:

- **No inmersiva:** La RV no inmersiva no sumerge por completo al usuario en el entorno virtual, ya que la interacción se realiza a través de pantallas tradicionales como ordenadores, tabletas o periféricos como teclados y ratones. Ofrece menor interactividad y presencia en comparación con la realidad virtual inmersiva, pero es útil en aplicaciones como simulaciones en 3D para entrenamiento técnico, diseño de productos, planificación industrial y juego de videojuegos como se muestra en la Figura 2-1.



Figura 2-1 Ejemplos de realidad virtual no inmersiva [7].

- **Semi-inmersiva:** Combina elementos de inmersión con interacción limitada. Suele emplear grandes pantallas, simuladores y sistemas de proyección. Ofrece una experiencia visual y sensorial envolvente, pero no alcanza el nivel de interactividad o sensación de presencia de la realidad virtual completamente inmersiva. Tiene como aplicaciones el entrenamiento de vuelo con simuladores, aprendizaje interactivo y recreaciones históricas como se expone en la Figura 2-2.



Figura 2-2 Ejemplos de realidad virtual semi-inmersiva [8].

- **Inmersiva:** La RV inmersiva envuelve totalmente al usuario en un ambiente creado por ordenador a través de cascos y gafas de realidad virtual, controladores de movimiento y sistemas de monitorización. Ofrece una experiencia sensorial sofisticada, que puede abarcar audio 3D, *feedback* háptico e incluso olores. Está concebida para potenciar el sentimiento de presencia, provocando que el usuario sienta que realmente se encuentra en el ambiente virtual. Sus usos incluyen prácticas militares, simulaciones de medicina, videojuegos, educación interactiva y diseño de arquitectura (Figura 2-3) [9].



Figura 2-3 Ejemplo de realidad virtual inmersiva [10].

2.2.3 Realidad virtual aplicada al ámbito militar

La historia de la simulación militar se remonta a los inicios de la humanidad, cuando los primeros conflictos llevaron a desarrollar estrategias para minimizar la incertidumbre del combate. Desde los juegos de guerra como el Wei Hai de Sun Tzu (~3000 a.C.) y el *Chaturanga*, precursor del ajedrez en la India (~700 a.C.), hasta los modelos en cajones de arena utilizados por los romanos, la simulación siempre ha sido una herramienta clave para la preparación militar. En la Edad Moderna, el *Kriegsspiel* (Figura 2-4) prusiano de 1811 revolucionó el entrenamiento táctico, mientras que en el siglo XX el desarrollo de modelos matemáticos y el uso de computadoras permitieron simulaciones más complejas. Con la llegada de la era digital, los simuladores avanzaron exponencialmente, integrando IA, RV y redes interconectadas, consolidándose como un pilar fundamental del entrenamiento militar en la actualidad.



Figura 2-4 *Kriegsspiel*, el juego de mesa que entrenó a los prusianos [11].

Hoy en día países como Estados Unidos, China, Rusia o Israel han integrado la RV al ámbito militar de una manera extraordinaria, desarrollando simuladores y videojuegos que sumerge a sus soldados a situaciones complejas donde la toma de decisiones es un factor imprescindible para el logro de la misión. El ejército español de forma progresiva está integrando métodos inmersivos para poder amortizar sus capacidades. Es por ello, que es imprescindible mantener el continuo desarrollo de estas tecnologías en el ámbito militar para aumentar la operatividad de medios y personal de una manera eficaz, poco costosa y más accesible. A pesar de esta necesidad de desarrollo, las FAS actualmente ya disponen de simuladores y equipos que implementan las tres variantes de RV [12]. A continuación, se expndern algunos ejemplos particulares de cada ejército:

- **Ejército de Tierra:** Cuenta actualmente con una amplia variedad de simuladores avanzados para el entrenamiento de sus tropas, abarcando desde la instrucción básica hasta ejercicios tácticos complejos. Entre los principales se encuentra el Simulador de Artillería de Campaña (SIMACA), desarrollado por Tecnobit, que permite el entrenamiento de unidades de artillería en condiciones realistas. El Simulador de Puntería y Tiro (SPT) Leopard reproduce las estaciones de combate del carro de combate Leopard, permitiendo prácticas de tiro y coordinación. El Simulador *Víctrix*, diseñado por Indra, mejora la precisión en el entrenamiento de tiro a través de entornos virtuales configurables. También destaca el Simulador del Vehículo de Combate de Infantería Pizarro, que permite a las tripulaciones entrenar de forma conjunta conectando cabinas en red. Además, como se observa en la Figura 2-5 el Simulador Multi-Vehículo (MVATS), desarrollado por Simumak, proporciona un

entorno táctico avanzado para operadores de vehículos blindados, ofreciendo realismo e interactividad [13] [14] [15].

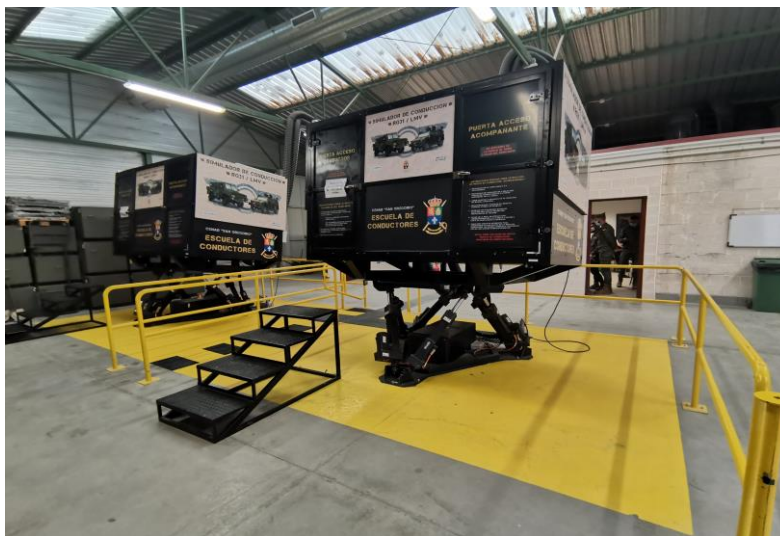


Figura 2-5 Simulador Multi-Vehículo desarrollado por Simumak [15].

- **Ejército del aire:** Dispone actualmente de una variedad de simuladores avanzados para el entrenamiento de sus pilotos. Entre ellos se encuentran los Full Mission Simulator (FMS) para aeronaves como el F-18 y el Eurofighter, que permiten a los pilotos practicar tácticas y procedimientos de emergencia en un entorno controlado. Además, se han incorporado simuladores para el avión de entrenamiento Pilatus PC-21 (Figura 2-6), utilizados en la Academia General del Aire (AGA) para la formación de nuevos pilotos. En cuanto a helicópteros, destaca el simulador del NH90, desarrollado por Indra, que reduce significativamente las horas de vuelo real necesarias para el adiestramiento. Estos sistemas de simulación no solo mejoran la eficacia del entrenamiento, sino que también incrementan la seguridad y reducen los costos operativos [16] [17].



Figura 2-6 Simulador PC-21 de la AGA [18].

- **Armada:** La Armada Española dispone de una amplia variedad de simuladores diseñados para la instrucción de sus efectivos en distintos ámbitos operativos. En la Escuela Naval Militar, el simulador de navegación permite a los alumnos maniobrar virtualmente variedad de unidades de la Flota, como las F-100 enfrentándose a situaciones como entradas en puerto, fondeo de precisión y navegación en condiciones adversas. Junto a este, el simulador táctico

Galeón replica el Centro de Información y Combate (CIC) de una fragata, equipándolo con sensores, armas y sistemas de comunicación para el adiestramiento en escenarios de conflicto. La Armada también emplea el simulador de tiro *Víctrix*, que sumerge a los infantes de marina en entornos urbanos virtuales mediante realidad aumentada. En Cartagena, el simulador de submarinos y cazaminas permite entrenar tripulaciones bajo condiciones realistas, mientras que en Ferrol se encuentra el centro de instrucción de seguridad interior (CISI), donde marineros y oficiales se adiestran en temario de seguridad interior, apagando fuegos, dando teóricas de control de averías y contraincendios. En Rota, se encuentra otro centro de adiestramiento que realiza también prácticas de seguridad interior, en el hay dos infraestructuras que simulan un barco, un helicóptero y un Harrier (AV8B+), como se muestra en la Figura 2-7. Recientemente, la Armada ha proyectado la construcción de un nuevo edificio de simulación en Cartagena, que integrará simuladores de tiro, tácticos y técnicos, reforzando el adiestramiento de la Fuerza de Infantería de Marina.



Figura 2-7 Centro de Adiestramiento de Seguridad Interior en Rota [19].

Los simuladores militares permiten un entrenamiento realista y seguro, optimizando la preparación operativa a través de la reducción de costos, la repetición ilimitada de ejercicios y la recreación de escenarios tácticos complejos en condiciones controladas. Es por ello, que el Ejército español incluye este tipo de preocupaciones de forma más habitual en su I+D+I con el objeto de amplificar sus aplicaciones actuales y extrapolarlas a unas que aporten más detalle y desarrollo en:

- **Entrenamiento y simulación en combate:** La RV permite recrear escenarios de combate realistas para entrenar a los soldados en tácticas, estrategias y toma de decisiones bajo presión. Esto incluye simulaciones de enfrentamientos urbanos, rescates y operaciones especiales, como se ha mencionado con anterioridad y a su vez se muestra en la Figura 2-8.
- **Entrenamiento de vehículos y aeronaves:** Los simuladores de RV se emplean para entrenar a pilotos de aviones, helicópteros y drones, así como a conductores de tanques y otros vehículos militares. Estas simulaciones permiten practicar maniobras complejas sin los riesgos ni los costes asociados al uso de equipos reales. Indra ha implementado en el ejército español el sistema SIMCUI, un simulador portátil que acelera la formación de pilotos civiles y militares, reduciendo el tiempo de entrenamiento a la mitad. Utilizan gafas de RV e IA para evaluar rendimiento y compatibilidad con 18 tipos de aeronaves permitiendo entrenar en simulaciones realistas, incluso de alto riesgo. Además, incluye opciones para

entrenamiento remoto y modelos personalizados, destacando el modelo del Targus, una aeronave remotamente pilotada desarrollada en España [20].



Figura 2-8 Simulador de armas pesada de la Guardia Nacional de Nueva Jersey [21].

- **Desactivación de explosivos (EOD):** La RV se utiliza así mismo, para entrenar a especialistas en desactivación de explosivos, permitiéndoles practicar técnicas de manejo de bombas en un entorno seguro antes de enfrentarse a situaciones reales.
- **Reconocimiento y planificación de misiones:** Esta tecnología permite realizar simulaciones de reconocimiento para planificar operaciones en terrenos desconocidos, simulando mapas o escenarios desde una perspectiva espacial detallada. Con mapas 3D y entornos virtuales, los equipos pueden familiarizarse con la zona y prever posibles desafíos antes de desplegarse, así como estrategias o planeamiento de misiones sin la necesidad de mapas.
- **Tratamiento de estrés postraumático (TEPT):** En el campo de la medicina, la se emplea para tratar a soldados con estrés postraumático a través de terapias de exposición controlada, permitiéndoles enfrentar y superar recuerdos traumáticos en un entorno seguro [22].

2.3 Combate en población

2.3.1 Características y retos del MOUT

El primer concepto, y de hecho el más relevante, es entender qué es el combate en población y por qué se está integrando en una simulación de RV con UE. Por ello, se abordarán una serie de características y principios básicos, así como procedimientos y reglas, para exponer de forma más clara y comprensible este amplio ámbito.

Las operaciones en entornos urbanos son cada vez más comunes en diversos escenarios y requieren un planeamiento detallado y un alto nivel de adiestramiento del personal. Estas operaciones se distinguen por su descentralización, lo que implica que compañías, secciones y pelotones a menudo ejecuten sus tareas de manera independiente del resto de los elementos de su unidad superior.

En España, los principales campos de entrenamiento para combate urbano incluyen el Polígono de Combate en Zona Urbanizada de Casas Altas, en el CENAD de San Gregorio (Zaragoza), el más completo del país; el futuro Centro Avanzado de Combate en Zona Urbana en la base 'El Empecinado' (Valladolid), que contará con edificios, túneles y bosques para entrenamiento en diferentes niveles; y el campo de adiestramiento en combate subterráneo de la Brigada de La Legión "Rey Alfonso XIII", pionero en su tipo. Estas instalaciones permiten un entrenamiento realista y efectivo en escenarios

urbanos complejos, permitiendo a la Infantería de Marina Española, recrear las situaciones que en un futuro puedan darse en zona de operaciones [23].



Figura 2-9 Infraestructuras del campo de prácticas de combate en población de Figueirido [24].

La Escuela Naval Militar tiene entre sus objetivos impartir a los alumnos las bases y conocimientos del procedimiento MOUT (Military Operations in Urban Terrain). Para su entrenamiento, los alumnos acuden a la Base Militar General Morillo (Figura 2-9), situada cerca de Figueirido, donde se encuentra un campo de combate poblacional, al que en la simulación se le dará el nombre de PICOZU (Polígono de Instrucción de Combate en Zona Urbana). En este lugar realizan una variedad de ejercicios prácticos, como la captura de rehenes, la neutralización de enemigos de relevancia y el manejo de la fuerza en situaciones donde hay presencia de civiles, adaptándose a las complejidades del entorno urbano.

Las operaciones en entornos urbanos requieren la adaptación de los fundamentos de la guerra a un contexto complejo y dinámico. Aunque las tácticas básicas del pelotón de fusiles se mantienen, el adiestramiento debe ajustarse a las particularidades del combate en ciudades, donde la presencia de obstáculos y la cobertura favorecen al enemigo.

El crecimiento de las ciudades y la urbanización han generado diferentes tipos de entornos urbanos, desde megaciudades densamente pobladas hasta pueblos rurales con infraestructuras limitadas. Esto influye en la movilidad, la seguridad y la estrategia de combate. Es por ello, por lo que la Infantería de Marina realiza simulaciones de estos pueblos o ciudades (Figura 2-10) para poder entrenar a sus soldados en estos movimientos y tácticas para futuras incursiones en zona de operaciones.

Uno de los principales desafíos es la geometría tridimensional del campo de batalla, donde los ataques pueden provenir de múltiples niveles, incluyendo edificios altos y pasajes subterráneos. Además, la concentración de combate en espacios reducidos dificulta la dispersión y aumenta la vulnerabilidad de las fuerzas.

Debido a la presencia de civiles, los jefes de pelotón deben gestionar simultáneamente misiones humanitarias, operaciones de seguridad y combate activo en distintas áreas. Esto exige un alto grado de disciplina y planificación estratégica para ajustar la intensidad del conflicto según la situación.

La iniciativa táctica es clave en el combate urbano, ya que la cadena de mando tradicional puede ser ineficaz en un entorno cambiante. Para ello, cada infante de marina debe estar capacitado para tomar decisiones rápidas basadas en seis principios fundamentales:

- **Seguridad:** Mantener alta alerta y cobertura efectiva.
- **Sorpresa:** Utilizar el sigilo y la velocidad operativa.

- **Tiempo:** Ejecutar acciones de manera fluida y eficaz.
- **Precisión:** Garantizar un conocimiento situacional compartido.
- **Organización *ad hoc*:** Adaptar la estructura de la unidad según la situación.
- **Violencia de la acción:** Mantener la iniciativa con fuego preciso y maniobras agresivas.

Asimismo, el combate urbano exige adherirse a tres reglas básicas:

- Identificar y cubrir zonas peligrosas hasta que sean aseguradas.
- Enfrentar amenazas con precisión, diferenciando civiles de combatientes.
- Protegerse mutuamente, manteniendo la comunicación y la seguridad de la unidad [25].



Figura 2-10 Zona destinada a combate urbano en San Gregorio [23].

2.3.2 Control del territorio y defensa en áreas urbanas

2.3.2.1 Movimiento exterior

En operaciones urbanas, la dispersión, la cobertura, el apoyo mutuo y la vigilancia continúan siendo de vital importancia. Los jefes de pelotón deben ajustar sus movimientos según la amenaza y el ritmo táctico, empleando rutas alternativas como pasajes subterráneos, atravesando estructuras o utilizando escaleras para superar obstáculos. Elegir itinerarios estratégicos con cabeza y experiencia es crucial para mantener la ventaja y desbaratar las tácticas enemigas. Se van a destacar a continuación los procedimientos de relevancia en el exterior:

- **Movimiento por exteriores:** N1 a vanguardia, N2 lado opuesto, N3 lado por el que se progresa y N4 retaguardia. En caso de ser más los sectores se cubrirán por binomios en los sectores que prosigan, dejando a N7 y N8 realizando la misma función que N1 y N2, y a N5 y N6 desempeñando el papel de N3 y N4, pero en el sector contrario, como se muestra en la Figura 2-11 [25].

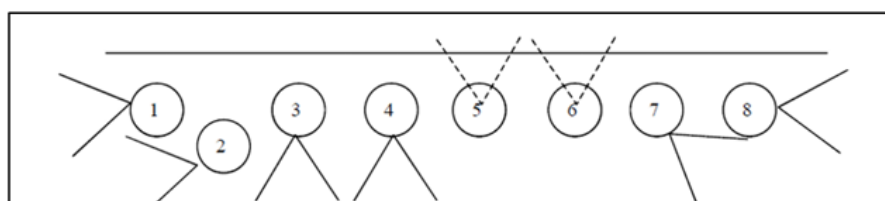


Figura 2-11 Movimiento en exterior por pelotón [25].

- **Zonas peligrosas:** La cobertura en el cruce de zonas peligrosas puede realizarse mediante dos técnicas principales: cobertura sucesiva y cobertura continua. En la cobertura sucesiva, cada integrante del pelotón avanza de manera escalonada, proporcionando seguridad al siguiente mientras cruza. En la cobertura continua, un infante mantiene la vigilancia constante mientras el resto de la unidad cruza, ocupando luego la última posición para proteger la retaguardia (Figura 2-12). Además, el movimiento en doble columna mejora la cobertura y el apoyo mutuo al dividir al pelotón en ambos lados de la calle, permitiendo cubrir múltiples direcciones y reducir la vulnerabilidad ante amenazas en altura [25].

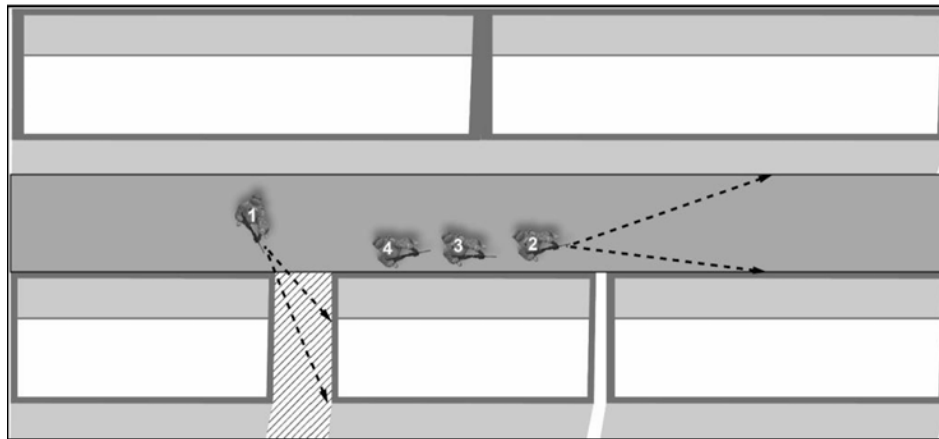


Figura 2-12 Ejemplo cobertura continua en exteriores [25].

- **Acordonamiento y búsqueda:** El acordonamiento y búsqueda es una táctica utilizada para aislar un objetivo, evitar fugas y realizar un registro minucioso, adaptándose a distintos niveles de amenaza. Se estructura en tres elementos: el cordón exterior, que impide interferencias y controla accesos; el cordón interior (Figura 2-13), encargado de la seguridad en la zona objetivo; y el elemento de búsqueda, que inspecciona, clasifica información y, si es necesario, asume un rol ofensivo. A diferencia de un simple perímetro de 360°, el cordón se organiza estratégicamente para optimizar cobertura y vigilancia. Además, el uso de escaleras portátiles facilita la movilidad vertical, permitiendo acceder a posiciones elevadas o superar obstáculos urbanos, lo que mejora el despliegue y la seguridad en espacios compartimentados. La correcta planificación y sincronización de estas acciones es clave para minimizar riesgos y garantizar el éxito operativo [25].

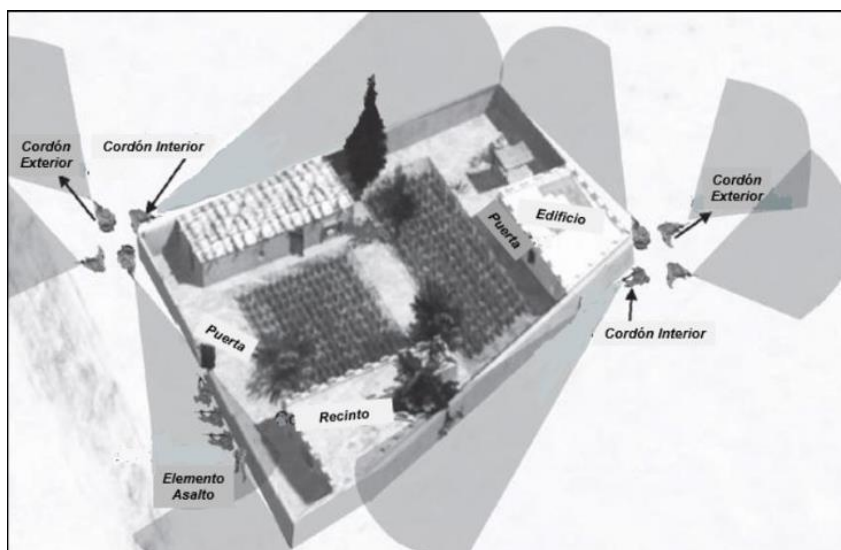


Figura 2-13 Cordón exterior e interior [25].

2.3.2.2 El ciclo de ataque urbano

El ciclo de ataque urbano es un proceso estructurado para transitar del movimiento exterior a la entrada en una zona hostil, compuesto por cinco fases: reconocimiento, aislamiento, establecimiento de la posición de apoyo, limpieza del objetivo y consolidación y reorganización. Primero, se recopila información mediante drones, fotografías o estimación visual para identificar los puntos de entrada. Luego, se aísla el área con un cordón de seguridad para evitar interferencias. La entrada se ejecuta de manera agresiva y decisiva, superando cualquier obstáculo defensivo y estableciendo una posición de apoyo. Posteriormente, se realiza una limpieza metódica del objetivo para eliminar amenazas. Finalmente, se consolida y reorganiza la unidad, redistribuyendo recursos, evacuando bajas y estableciendo defensas para resistir posibles contraataques y garantizar la continuidad de las operaciones [25].

2.3.2.3 Movimiento interior

La limpieza de edificios en combate urbano es una tarea peligrosa que debe estudiarse antes de ejecutarse. Si una edificación no posee un valor táctico y puede tener artefactos explosivos, es preferible evitarla y mantener vigilancia sobre ella. Si es un objetivo hostil con baja probabilidad de daños colaterales, se recomienda el uso de fuego de apoyo para despejarla antes de exponer al personal. En casos donde el combate cercano es inevitable, se deben emplear granadas de fragmentación o aturdidoras para minimizar riesgos. La limpieza se realiza siguiendo los siguientes principios clave: maximizar la visibilidad desde el exterior, evitar entradas directas si existen alternativas, moverse agresivamente a través de zonas vulnerables, asegurar puntos de control con cobertura, eliminar amenazas de manera sistemática y registrar meticulosamente la habitación. Es esencial marcar los espacios despejados y, si no se mantiene vigilancia constante, repetir la limpieza antes de reentrar. La estrategia debe considerar la tridimensionalidad del compartimento en cuestión, verificando peligros en todas direcciones y manteniendo siempre el control táctico de la operación [25]. Una mala gestión de estos aspectos puede llegar a causar consecuencias fatales para el pelotón, por lo que el seguimiento estricto de las maniobras debe tenerse en cuenta (Figura 2-14).

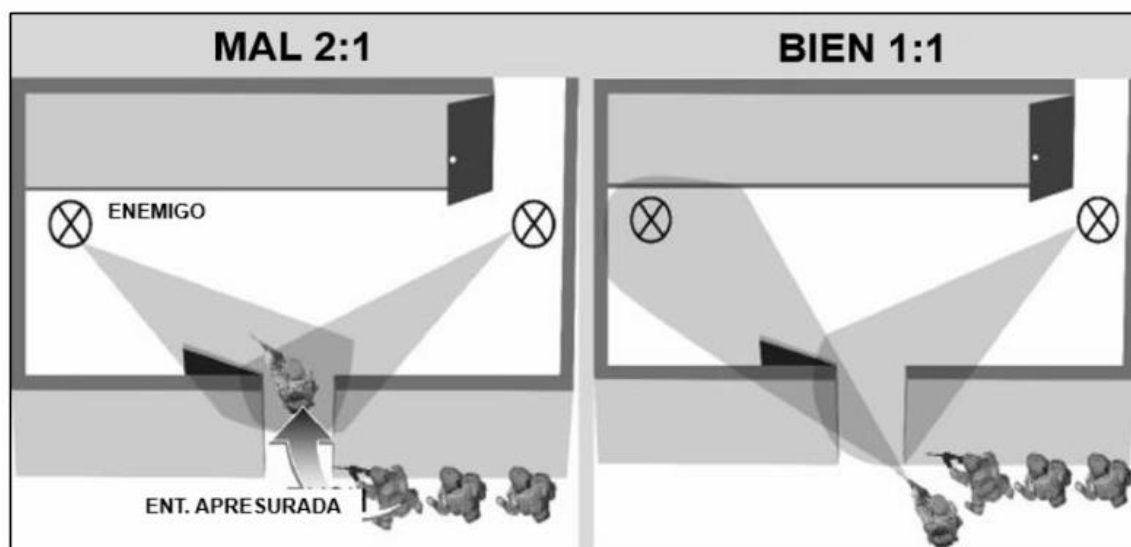


Figura 2-14 Ejemplo de cómo hacer y como no hacer entrada a una habitación [25].

La forma actualmente correcta ya se encuentra estipulada y consiste en que el primer infante de marina que la identifica se aparta de la puerta mientras inspecciona el interior desde el exterior tanto como sea posible. Una vez que ha evaluado la situación, entra rápidamente y se dirige a la esquina con menor visibilidad para asegurarla. El segundo soldado le sigue, avanzando en la dirección opuesta y cubriendo el centro de la habitación o cualquier punto ciego restante. Los demás miembros del equipo

entran progresivamente, asegurando sus sectores de tiro hacia el centro de la habitación para garantizar el control total del espacio. Una vez asegurada, la unidad se reagrupa para continuar la limpieza o abandonar la zona. Antes de salir, el jefe del equipo verifica el estado del pelotón y confirma que todo está en orden antes de seguir [25]. Se procede a mostrar un ejemplo donde se observa la entrada de cuatro soldados en una habitación como se muestra en la Figura 2-15:

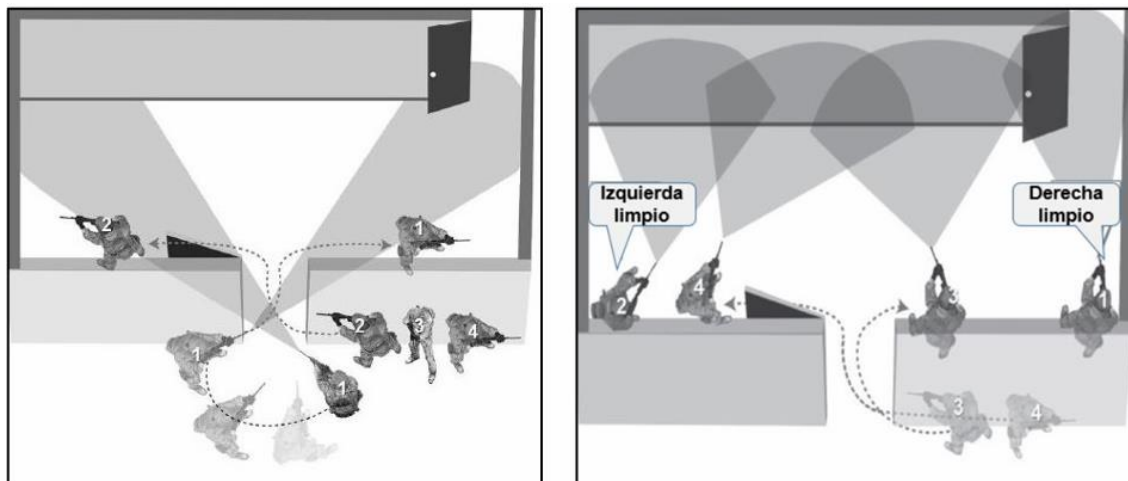


Figura 2-15 Forma correcta y estipulada de limpiar una habitación una vez se entra [25].

A continuación, dado que las maniobras en el interior son muy extensas, se van a citar los métodos y procedimientos de las situaciones más probables a la hora de trabajar en compartimentos. Cabe resaltar que estos procedimientos son aplicables al exterior:

- **Esquinas:** Para doblar una esquina de manera segura, se debe evitar exponer el arma antes del cuerpo. La mejor técnica consiste en ampliar el ángulo de observación sin comprometer la posición, moviéndose con control y en silencio, como puede observar en la Figura 2-16. Cuanto más larga sea el arma, mayor deberá ser la apertura al girar la esquina para evitar sorpresas. Esta técnica también es útil para inspeccionar habitaciones sin necesidad de ingresar directamente [25].

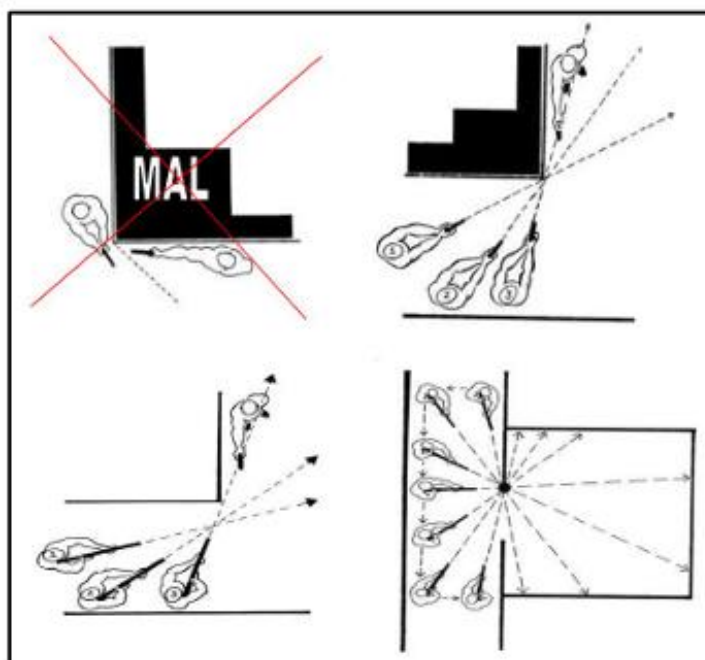


Figura 2-16 Forma correcta de actuar frente a una esquina [25].

- **Intersección de pasillos en forma de “L”:** Al enfrentar un pasillo en forma de "L", se emplean técnicas similares a la limpieza de habitaciones. El primer infante de marina identifica el pasillo, detiene el avance de la unidad antes de la curva y evita que su cuerpo o el cañón del fusil sobresalgan. Mientras mantiene la seguridad en vanguardia, el segundo infante avanza hasta la esquina sin exponerse (Figura 2-17). Cuando está listo, ambos se mueven simultáneamente: el primero se mantiene cerca de la esquina como cobertura, mientras el segundo avanza agresivamente para asegurar el control del pasillo [25].

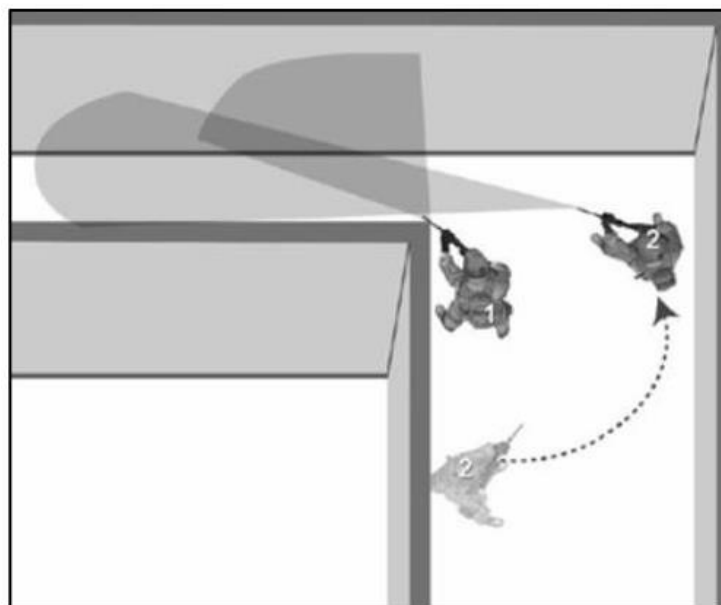


Figura 2-17 Manera de acceder a un pasillo en forma de “L” [25].

- **Intersección de pasillos en forma de “T” y escaleras:** El procedimiento en pasillos “T” es igual al del pasillo con forma de “L”, lo único que varía es que es necesario la existencia de un par de binomios para que cada uno cubra sus sectores. Respecto a la actuación frente a escaleras, cabe resaltar que estas representan uno de los espacios más complejos en combate urbano, por lo que conocer su estructura ayuda a determinar la mejor táctica de limpieza. Existen dos tipos principales: continuas y discontinuas. Las continuas, ubicadas generalmente en los extremos o esquinas de los edificios, requieren avanzar por segmentos asegurando cada tramo antes de continuar, con un equipo tomando la iniciativa en cada nivel. Las discontinuas, más abiertas y con rellanos, suelen encontrarse en el centro del edificio y requieren un movimiento constante para evitar ser blanco fácil. En este caso, el primer infante de marina avanza cubriendo los niveles superiores, mientras el segundo protege el frente. Si la entrada al edificio se ha realizado desde un helicóptero, la limpieza sigue el mismo procedimiento, pero descendiendo, asegurando las salidas para evitar la fuga del enemigo y verificando la estabilidad de las azoteas antes del desembarco [25].

2.3.3 Reglas de enfrentamiento y ámbito de aplicación

Las ROE son directrices operativas que regulan el uso de la fuerza en operaciones militares, estableciendo las condiciones, circunstancias y límites en los que puede aplicarse. Su objetivo es garantizar que las acciones militares sean necesarias, proporcionales y conformes con el Derecho Internacional Humanitario (DIH) y el *Law of Armed Conflict* (LOAC). A nivel OTAN, las ROE son imprescindibles para coordinar las fuerzas multinacionales y asegurar una respuesta coherente en misiones conjuntas. Estas reglas no son leyes en sí mismas, pero deben respetar el marco normativo internacional y nacional, funcionando como órdenes militares legítimas cuyo incumplimiento puede

acarrear sanciones disciplinarias o penales. Al igual que existen las reglas de enfrentamiento, también se encuentran las *Rules of Engagement Nullified* (REN), que al igual que las ROE son pedidas por mandos en baja posición dentro de la jerarquía (Figura 2-18) a mandos autoritarios. En la Armada, por ejemplo, el comandante de un barco debe solicitar las ROE al CTG (*Commander Task Group*) o CTF (*Commander Task Force*) en función de las acciones que pueda tomar con la aplicación de las ROE [26].

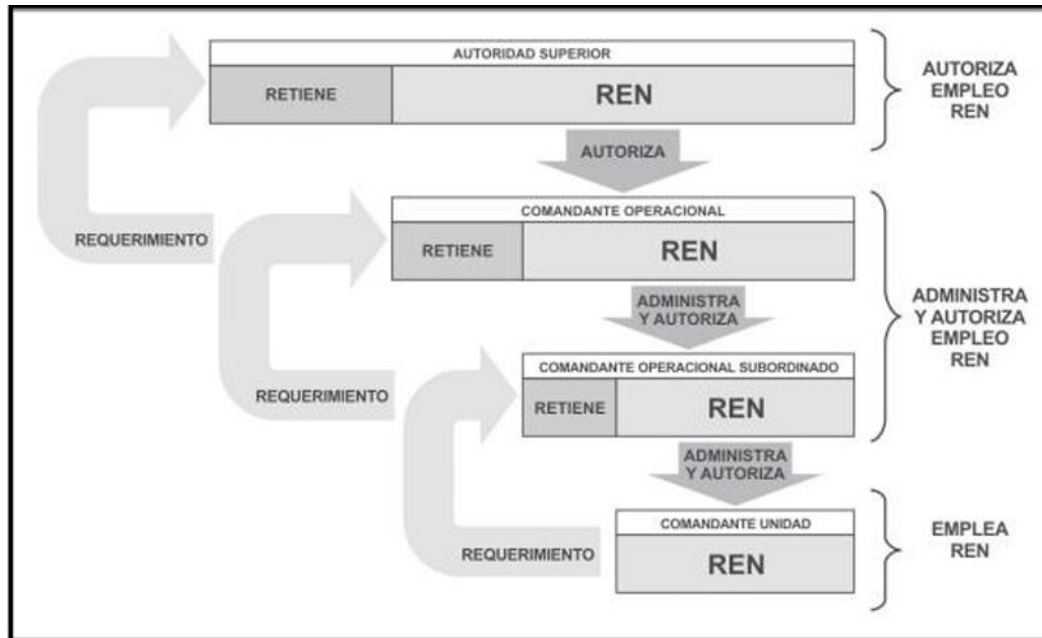


Figura 2-18 Jerarquía del requerimiento de las REN a nivel general [26].

Las operaciones militares están establecidas en base a el derecho internacional, incluyendo la LOAC como se ha mencionado con anterioridad, la Ley de Ginebra y la Ley de La Haya, así como el derecho internacional de los derechos humanos. Cada país tiene la obligación de concienciar a sus fuerzas del cumplimiento de estas normativas y de adaptar las ROE para garantizar el uso de la legítima defensa de la fuerza. Además, las leyes nacionales pueden imponer restricciones adicionales al empleo de la fuerza, lo que debe ser comunicado a los aliados en operaciones multinacionales para evitar conflictos operativos a causa de desconocimiento. Las políticas nacionales también influyen en la planificación y ejecución de misiones, ya que algunos países pueden establecer límites más estrictos sobre el daño colateral o restringir el papel de sus fuerzas en la aplicación de la ley. En operaciones multinacionales, estas diferencias legales y políticas pueden generar fricciones en la aplicación de las ROE, por lo que se deben negociar para lograr una mayor coherencia entre los miembros de la coalición, evitando que las restricciones de un solo país comprometan la efectividad operativa del grupo [26].

Es necesario, antes de seguir con el desarrollo de estas reglas, dejar claros dos conceptos: legítima defensa, que hace referencia al derecho de una unidad, pelotón, brigada, etc., para defenderse de un ataque o de un ataque inminente. Algunos países permiten que los comandantes limiten la autodefensa de un individuo de la misma forma que la autodefensa de una unidad. Y por otro lado la defensa extendida, donde los comandantes o capitanes de compañías tienen el derecho de defender unidades de su país frente a un ataque o ataque inminente [26].

A la hora de pedir las ROE en un enfrentamiento se ha de seguir un procedimiento, para que estas sean aprobadas y por tanto utilizadas. Las ROE son aprobadas por las autoridades nacionales o por organismos internacionales, como la ONU, según sus procedimientos y acuerdos. Se desarrollan en paralelo con el plan de operaciones (OPLAN) o la orden de operaciones (OPORDER) y pueden incluirse en estos documentos o publicarse por separado. Además, definen el área geográfica de aplicación dentro de la misión. Las ROE deben ser revisadas continuamente para garantizar su claridad, legalidad y adecuación a los requerimientos operacionales, asegurando que los comandantes cuenten con la

autoridad necesaria para responder a las amenazas. Para ello, existen formatos específicos para solicitar (ROEREQ), autorizar (ROEAUTH) e implementar (ROEIMP) las ROE. En cuanto a su seguridad, mientras que las ROE para entrenamiento suelen ser no clasificadas, las utilizadas en operaciones reales generalmente tienen el mismo nivel de clasificación que el OPLAN o la OPORDER [26].

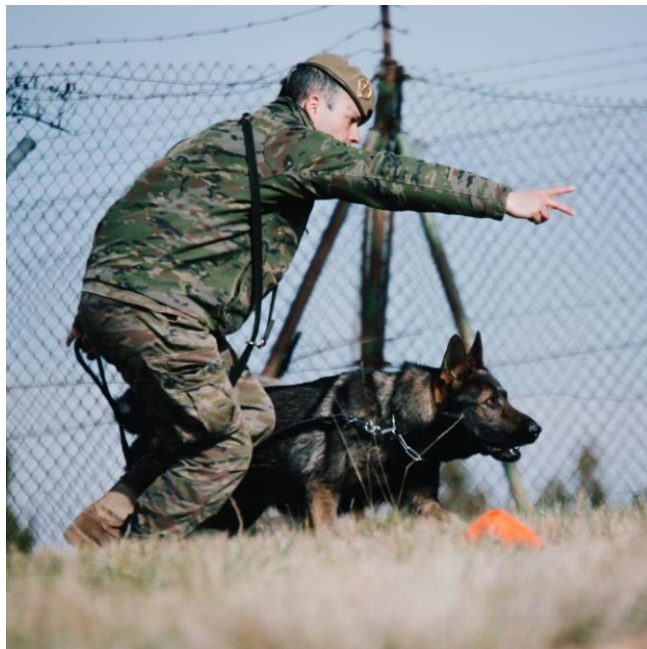


Figura 2-19 Soldado del Ejército de Tierra dando órdenes a un perro de raza Pastor Belga [27].

En el ámbito terrestre abarcan una amplia gama de situaciones operativas para regular el uso de la fuerza y garantizar el cumplimiento de la misión dentro del marco legal. Entre ellas, se incluyen la protección de la libertad de movimiento, el uso de disparos de advertencia, y la búsqueda y detención de personas. También se contemplan reglas para la protección y destrucción de propiedad, posicionamiento geográfico y relativo de las fuerzas y la realización de ejercicios en presencia de un adversario, entre muchas otras.

A su vez, su ámbito de aplicación se basa en el escalamiento de la fuerza que regula el uso progresivo de la fuerza en autodefensa, priorizando siempre opciones menos dañinas antes de recurrir a medidas letales. Inicia con presencia disuasoria, advertencias y control físico, avanzando solo si es necesario hacia armas no letales (Figura 2-19) y, en última instancia, letales. Su objetivo es evitar el uso innecesario de la fuerza letal y debe aplicarse según cada situación. Aunque las ROE pueden restringir ciertas opciones, nunca limitan el derecho a la autodefensa. Para una correcta aplicación, el entrenamiento en escenarios realistas es clave [26].

La adecuada coordinación entre los diferentes actores involucrados en la misión permite minimizar riesgos y optimizar la toma de decisiones en situaciones críticas. Asimismo, la planificación detallada y el análisis de posibles contingencias favorecen la capacidad de respuesta ante escenarios imprevistos. En este sentido, la implementación de protocolos claros y el entrenamiento continuo de las fuerzas resultan esenciales para garantizar que cada acción tomada se ajuste a los principios establecidos en las ROE y al marco normativo vigente.

Cada mando es responsable de definir un planteamiento estratégico para la misión que se va a llevar a cabo, considerando todos los posibles escenarios que puedan presentarse. Esto implica evaluar el nivel de escalada en el que se encuentra la fuerza (crisis, tiempos de paz, guerra, etc.) y, con base en este análisis, establecer el *Course of Action* (COA) y ROE necesarias para cada situación o acción específica. Una forma simplificada y visual de representar este proceso se muestra en el esquema de la Figura 2-20 a continuación:

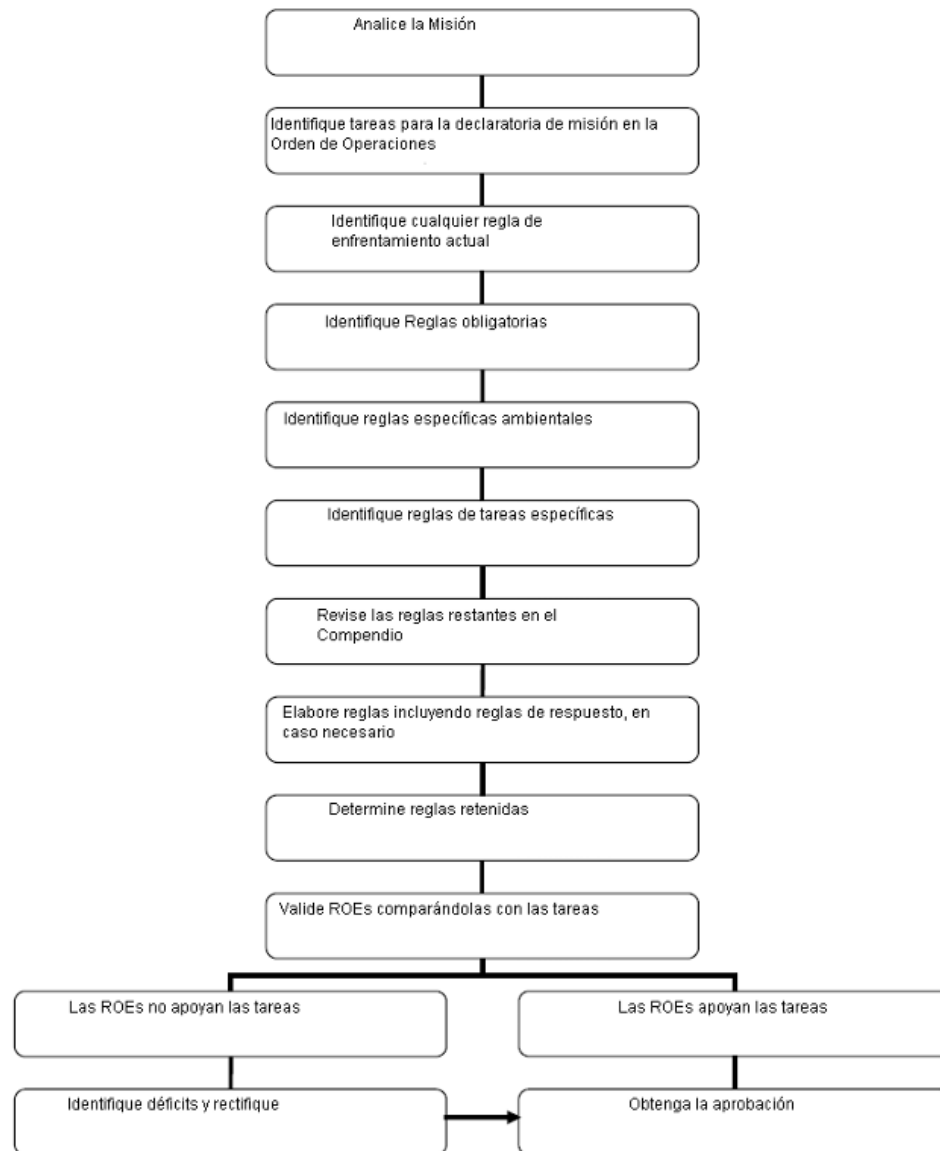


Figura 2-20 Planteamiento estructurada de la aplicación de las ROE [26].

2.4 Programación

2.4.1 Concepto general de la programación en UE

El lanzamiento de UE ha abierto nuevas posibilidades para los desarrolladores en la creación de videojuegos, ofreciendo herramientas líderes en la industria para diseñar experiencias visuales muy realistas. Este motor de juego permite programar interacciones complejas, IA y efectos visuales mediante dos opciones principales: BP, un lenguaje visual que facilita la creación de lógica conectando nodos (una especie de cajas asociadas a una función, variable, interfaz...), y C++, que ofrece mayor control y flexibilidad [28].

Los BP están compuestos por nodos, donde cada uno representa una acción específica, como el desplazamiento de un personaje o la reproducción de sonidos. Estos nodos se conectan a través de líneas que establecen el flujo de ejecución del programa. Las variables cumplen la función de almacenar y manipular datos dentro del juego, permitiendo gestionar valores como la posición del personaje mediante una variable de tipo *vector* o el estado de salud del jugador.

Los eventos actúan como desencadenantes de sucesos en respuesta a acciones del usuario o del sistema, como la recogida de un objeto o la carga de un nivel. El nodo *On Component Begin Overlap*,

mostrado en la Figura 2-21, es un ejemplo de evento que permite iniciar la secuencia de acciones especificada. Por otro lado, las condiciones permiten la toma de decisiones dentro del flujo lógico del juego, estableciendo verificaciones que determinan la ejecución de diferentes acciones según se cumplan ciertos criterios.

Las funciones encapsulan conjuntos de nodos reutilizables, lo que facilita la organización, el mantenimiento y la legibilidad del código. En este contexto, el encapsulamiento se refiere a la agrupación de datos y funciones dentro de una estructura, ocultando su implementación interna y permitiendo el acceso únicamente a través de métodos específicos, con el fin de mejorar la modularidad y la seguridad del código [29].

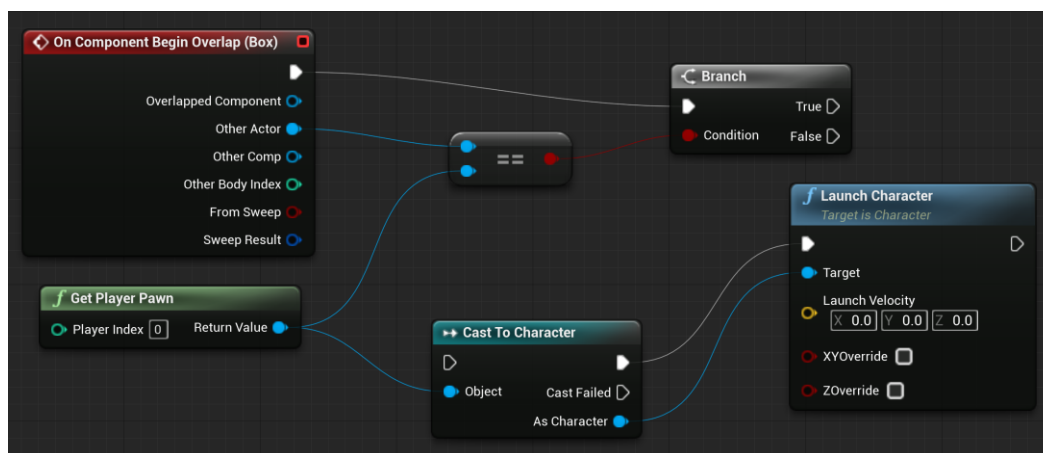


Figura 2-21 Ejemplo de programación con BP en UE [30].

2.4.2 Tipos y características

Los BP son una de las herramientas más potentes y versátiles de UE5 como se ha citado anteriormente, diseñadas para permitir a los desarrolladores implementar lógica, interacciones y funcionalidades en sus proyectos sin necesidad de escribir código. Dicho funcionamiento, hace más accesible la herramienta a programadores que no tengan excesiva experiencia. A continuación, se describen los principales tipos de BP, junto con información adicional sobre sus características y aplicaciones:

- **Level Blueprint:** Este BP, actúa como un gráfico de eventos global que permite manejar la programación específica de un nivel del juego. Es ideal para gestionar interacciones y eventos únicos en el nivel actual, como activadores de puertas, sistemas de iluminación dinámica o secuencias de cinemáticas. Cada nivel tiene su propio *Level Blueprint*, lo que facilita el diseño de comportamientos independientes entre distintos escenarios.
- **Blueprint Class:** Este BP, extiende la funcionalidad de las clases base programa, permitiendo agregar contenido personalizado sobre estas. Por ejemplo, un desarrollador puede tomar una clase existente como *Actor* o *Character* y añadirle comportamientos específicos, como habilidades especiales, propiedades físicas únicas o interacciones con otros objetos. Esto hace que *Blueprint Class* sea una herramienta esencial para crear personajes, armas y objetos interactivos. Exactamente, en el programa de UE se ofrecen los siguientes *Blueprint class* [31]: Los de tipo *Actor* representan objetos que pueden colocarse en el mundo, como cubos, árboles, puertas, edificios, armas, proyectiles, entre otros. Por otro lado, el *Pawn* o NPC (Non-Player Character) sirve como base para entidades controlables, ya sea por el jugador o la IA. El *Character* extiende el *Pawn* con funcionalidades específicas para personajes humanoides, como movimiento y animaciones. Este tipo de clase se utiliza para los enemigos colocados en el futuro escenario PICOZU, los cuales serán controlados por una IA. La clase *GameMode* define las reglas y la lógica principal del juego, mientras

que el *PlayerController* maneja la entrada del jugador y su interacción con el mundo. Por otro lado, el *AIController* se encarga de controlar personajes no jugables mediante IA. Prosiguiendo está el *Scene Component* que es un *Actor Component* con una posición en el mundo definida por un *FTransform* (ubicación, rotación y escala). Pueden formar jerarquías mediante adjuntos, donde un *Actor* tiene un único *Scene Component* raíz que define su transformación. Solo los *Scene Components* pueden adjuntarse entre sí, sin permitir ciclos, usando *SetupAttachment* o *AttachToComponent*. Además, los *Primitive Components* (*UPrimitiveComponent*) son *Scene Components* con geometría para renderizado o colisión, destacando los *Box*, *Capsule*, *Static Mesh* y *Skeletal Mesh Components*[31]. En la Figura 2-22 se muestra la interfaz de estas clases de BP.

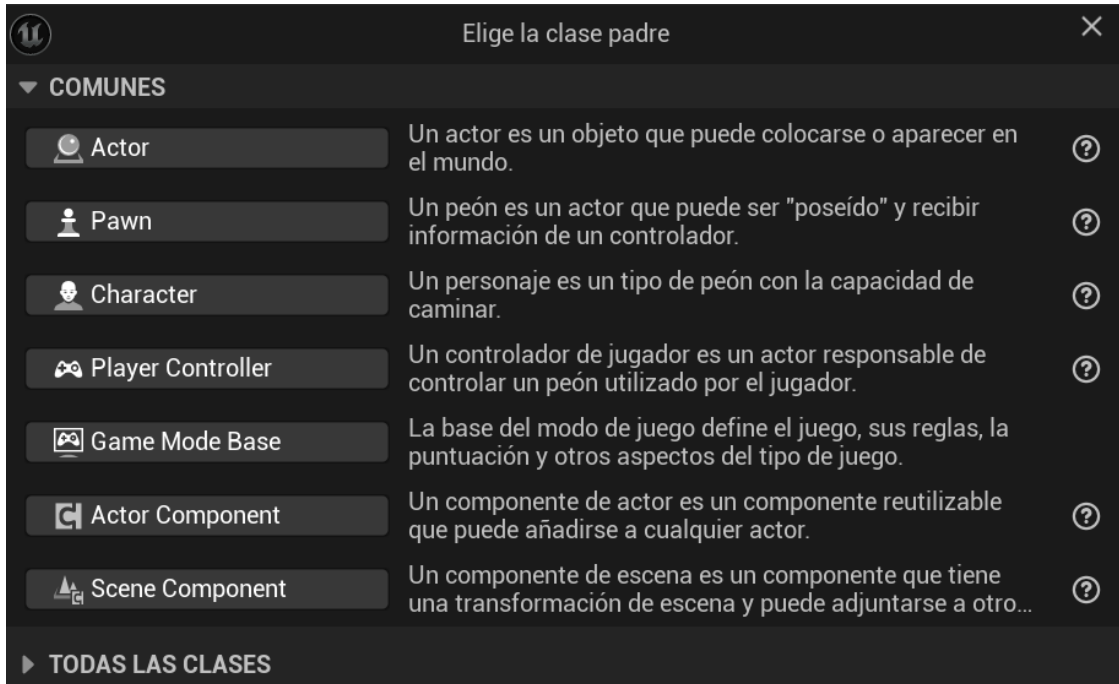


Figura 2-22 Interfaz de UE al seleccionar nuevo BP (Autoría propia).

- **Blueprint de Interfaz:** Implementa un conjunto de funciones que pueden ser implementadas por diversas clases, facilitando una comunicación uniforme entre ellas.
- **Blueprint de Macro:** Reúne un grupo de nodos que pueden ser reutilizados en otros BP, simplificando así la lógica repetitiva. A diferencia de los BP de función, las funciones no admiten retrasos y resultan más eficaces al operar únicamente cuando se llaman.
- **Blueprint de Función:** Este BP facilita la encapsulación de lógica reutilizable con el fin de optimizar la organización, eficacia y la lectura de los BP. Estas funciones admiten "inputs" a partir de los cuales retornan valores de salida y preservan un contexto local para prevenir conflictos con variables a nivel mundial. Son perfectos para realizar cálculos, tareas reiterativas y segmentar lógica compleja en secciones fácilmente manejables [31].
- **Input Mapping Context:** Aunque no son como tal BP, los *Input Action* y el *Input Mapping Context* (IMC) son dos conceptos de especial relevancia a la hora de programar el escenario. Son eventos modificables que permiten manejar los movimientos en el escenario del jugador de manera flexible y estructurada. En lugar de enlazar directamente teclas o botones, se definen acciones como "Saltar" o "Disparar" dentro de un IMC, el cual está conformado por diversos *Input Actions* que llevan asignados controles específicos (teclado, botones, ratón, etc.). Este sistema, tiene su base en el *Enhanced Input*, que permite modificar controles en tiempo real, adaptarse a diferentes dispositivos y gestionar entradas con mayor precisión, como la presión en un gatillo de un mando.

Estos BP mencionados son los más usados, pero hay una infinidad de tipos y subtipos específicos utilizados para dar más realismo, y de hecho alguno de ellos se usará para el trabajo.

2.4.3 Otras funcionalidades y servicios

UE5 a nivel programación es capaz de intensificar el escenario con muchos otros servicios, tanto propios como externos, para que se vea lo más realista posible. Desde iluminación hasta sonido, cada detalle cuenta para lograr esa sensación de inmersión total.

La luz es una de las cosas más importantes para hacer que un entorno se sienta creíble. Con *Lumen*, UE permite tener iluminación global dinámica, lo que significa que la luz se comporta de manera realista, rebotando en superficies y adaptándose en tiempo real sin necesidad de generar *lightmaps* como se muestra en la Figura 2-23. Si se tiene una tarjeta gráfica potente, se puede llevar esto más allá activando *Ray Tracing*, lo que mejora sombras, reflejos y oclusión ambiental para un nivel de realismo casi cinematográfico. Para añadir más profundidad se dispone del *Volumetric Fog*, una neblina volumétrica que interactúa con la luz, perfecta para ambientes húmedos, misteriosos o de ciencia ficción. [32]



Figura 2-23 Escenario generado con Lumen [32].

Otro factor que rompe la inmersión es una textura borrosa o un modelo con pocos detalles. Aquí es donde *Nanite* entra, ya que permite usar modelos con millones de polígonos sin afectar el rendimiento, lo que significa que puedes importar objetos escaneados de la vida real sin preocuparse por la optimización. Para mejorar aún más los materiales, se puede hacer uso de *Substance*, que da control total sobre detalles como rugosidad, reflejos y mapas normales, logrando superficies súper realistas. Además, con *Virtual Texturing*, se pueden llegar a manejar texturas enormes (8K o más) sin que el rendimiento sufra, asegurando que cada detalle de tu escenario se vea nítido [32].

Un cielo realista hace una diferencia notoria en la ambientación. Con *Sky Atmosphere*, se puede simular la dispersión de la luz en la atmósfera de manera precisa, logrando amaneceres y atardeceres impresionantes que cambian según la posición del sol. Para que las nubes que se sientan más naturales y no solo una textura en el cielo, *Volumetric Clouds* es la herramienta, ya que reacciona a la luz y puede animarse para crear un clima dinámico. Para darle a la atmósfera vida con lluvia, nieve o tormentas, el sistema de partículas *Niagara* permite crear efectos climáticos hiperrealistas que afectan el entorno y los personajes en tiempo real. En mayor profundidad, *Niagara* es un sistema de efectos visuales (VFX) avanzado de UE5, diseñado para crear partículas realistas como fuego, humo, lluvia, explosiones entre otras [32].

El mundo real no es estático, y en un juego realista todo debe reaccionar de forma natural. Para esto, *Chaos Physics* permite simular colisiones, destrucción y físicas avanzadas sin necesidad de *plugins* externos. Los *MetaHuman* (Figura 2-24), modelos hiperrealistas con animaciones faciales detalladas, ideales para cinemáticas o juegos narrativos también colaboran a ambientar el escenario y dotarlo de

realismo. Y para agregar más vida en el entorno, se puede usar *Wind System* para hacer que el viento mueva árboles, pasto y ropa, logrando un escenario que se sienta dinámico y lleno de energía [32].

El sonido es otro factor clave para la inmersión, y en *Unreal* se puede hacer que cada efecto tenga una ubicación y dirección precisa con el sistema de audio espacial 3D. Si se quiere que el sonido se adapte al entorno, *Convolution Reverb* es la herramienta adecuada, ya que simula cómo el eco y la reverberación cambian dependiendo del espacio en el que estés (por ejemplo, una cueva tendrá un eco más largo que una habitación pequeña). Esto hace que los pasos, los disparos o incluso el viento se sientan mucho más realistas y envolventes [32].

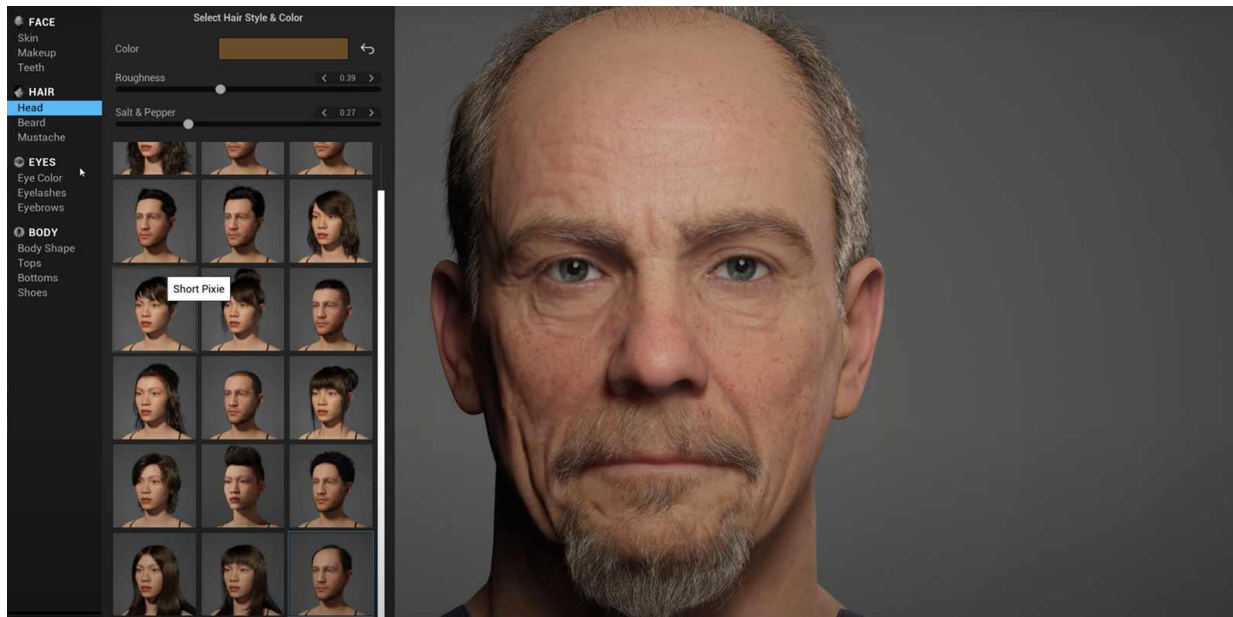


Figura 2-24 Ejemplo de MetaHuman [33].

Como último elemento está *Quixel Megascans* (Figura 2-25), que es una enorme biblioteca de recursos gráficos, escaneados del mundo real con calidad cinematográfica. En caso de querer modelar un escenario, como puede ser la casa personal de un individuo se puede usar *RealityCapture* que permite tomar fotos de un lugar y convertirlas en modelos 3D detallados, llevándolos directamente al juego [32].

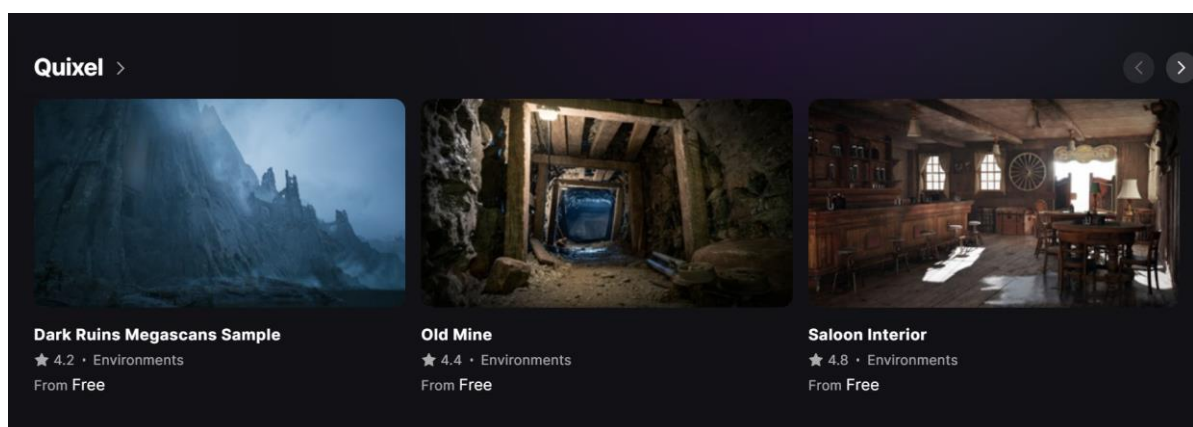


Figura 2-25 Interfaz de la página Quixel MegaScans [34].

2.4.4 Inteligencia artificial

Una vez establecidas las bases de BP en UE y considerando el trabajo que se está realizando, es importante resaltar y profundizar en el aspecto de la IA. La IA es un sistema que permite crear movimientos dinámicos y realistas en personajes NPC (Figura 2-26) o de tipo *Character*, proporcionándoles un "cerebro" para que actúen de forma inteligente y puedan tomar decisiones en

función de la situación, comunicándose entre ellos. Este sistema ha estado presente en UE desde su tercera versión y ha evolucionado a lo largo de sus diferentes actualizaciones [35].



Figura 2-26 NPC [36].

A la hora de programar un enemigo de tipo *Character*, existen dos posibilidades o vertientes: la programación con BP, utilizando *Input Actions* y *Axis Mappings* configurados en el *Character Movement Component*; o, por otro lado, el uso de IA, que emplea *Behavior Trees* y *Blackboards* para generar un movimiento más realista e inteligente del personaje [35].

La IA en UE es un concepto que se extiende considerablemente, abarcando diversas ramas, que a continuación se van a dividir para entender con mayor profundidad los ámbitos que abarca este aspecto:

- **Behavior Trees (BT):** Mientras que el BT ejecuta las ramas de lógica, este depende de un activo llamado Pizarra o *Blackboard* (BB), que actúa como su "cerebro" y almacena información clave como si fuese una lista. El Blackboard contiene las llamadas *claves* definidas por el usuario, que son en resumen las diferentes variables a las que llama cada una de las acciones programadas en el jugador, como por ejemplo una variable tipo *bool* "Luz_Encendida". En caso de estar la luz encendida, una cucaracha podría huir, y si está apagada, podría moverse aleatoriamente. Los BT pueden ser simples, como en el caso anteriormente mencionados, o tan complejos como simular un jugador humano en un juego multijugador, buscando cobertura, disparando y recolectando objetos [37]. Cuando se mencionan los BT, se debe hacer énfasis en identificar sus diferentes "nodos", similares a los utilizados en BP, para estructurar y secuenciar el comportamiento de la IA. Existen tres principales, el tipo *Selector* que ejecuta ramas de izquierda a derecha y detiene su búsqueda cuando encuentra una tarea que se ejecuta con éxito, permaneciendo en esa rama hasta que finaliza. En cambio, los de tipo *Sequence* también ejecutan las ramas en orden, pero siguen hasta que un nodo no termina o falla, deteniendo el flujo. Por último, el tipo *Simple Parallel* que permite ejecutar una tarea principal y una secundaria en simultáneo, finalizando, dependiendo de si la tarea principal concluye primero o si debe esperar la ejecución de la secundaria. Algo parecido ocurre con los *Decorators*, conocidos como "if", ya que imponen condiciones a las diferentes tareas que ejecuta el sistema. En la Figura 2-27, se observan estos tres tipos de nodos en un BT ejemplo [37].
- **Sistema de Navegación y pathfinding:** Permite que la IA se desplace por el nivel mediante *pathfinding* (cuya función es darle infinitud de caminos al NPC para desplazarse) a través de un *NavMesh* a partir de la geometría de colisión. Este *NavMesh* se divide en *tiles*, que son fragmentos del *NavMesh* y luego en polígonos, los cuales tienen un peso asignado para calcular la ruta óptima. Se puede modificar el *NavMesh* para así afectar el movimiento de los personajes y conectar áreas no contiguas, como puentes y plataformas a través de *NavLinks*. Existen tres modos de generación del *NavMesh*: estático, dinámico y solo modificadores dinámicos, que determinan cómo se actualiza en el proyecto. Además, el sistema incluye dos métodos para evitar

obstáculos: *Reciprocal Velocity Obstacles* (RVO) y *Detour Crowd Manager*, permitiendo que los agentes esquiven objetos y otros NPCs en tiempo real [38].

- **Sistema de percepción (AI Perception):** El sistema *AI Perception* en *Unreal* es una herramienta fundamental en el desarrollo de la IA, para que los personajes detecten y reaccionen a estímulos del entorno como ruido o daño. A través de componentes (*AI Perception Component*), los *Characters* pueden procesar información a través de sensores utilizando distintos módulos como *AI Sense_Sight* para detección visual, *AI Sense_Hearing* para eventos sonoros, *AI Sense_Damage* para registrar impactos y *AI Sense_Touch* para contacto físico con objetos o personajes. Estos sensores pueden configurarse en el *AI Controller* y ser integrados con la BB y BT para mejorar la toma de decisiones de la IA, a la hora de actuar o pensar. Este sistema es realmente útil en la creación de enemigos que patrullan, detectan jugadores o reaccionan a cambios en el entorno de forma temporal, haciendo del escenario un entorno más realista e inmersivo [39].

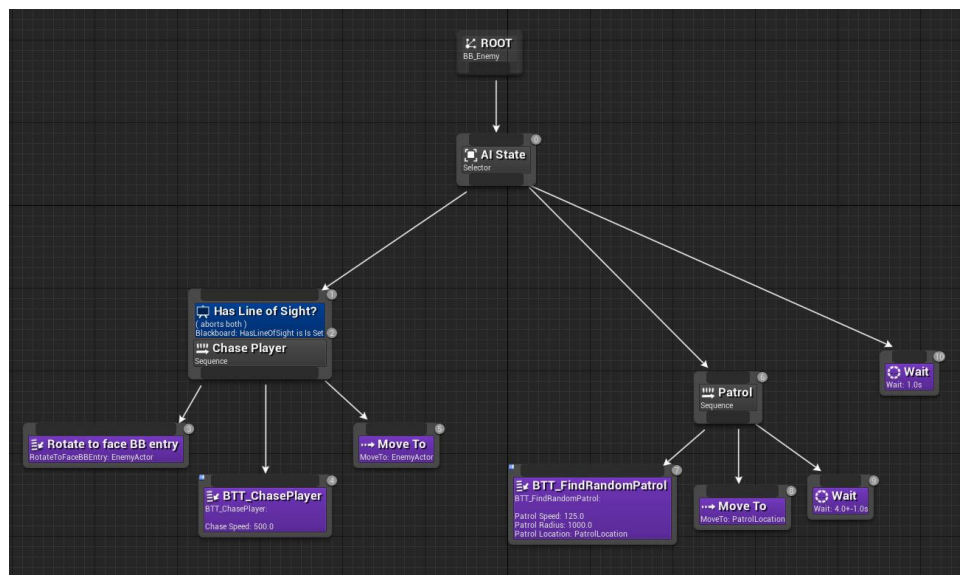


Figura 2-27 Ejemplo de un BT en UE [40]

2.5 Software y equipos

Este apartado tiene como propósito presentar de manera clara los diferentes programas y aplicaciones, aparte del motor de juego UE, que fueron empleados a lo largo del desarrollo del proyecto. De esta manera, se establecen las bases y conceptos necesarios para comprender su funcionalidad e integración en el proceso de trabajo.

2.5.1 Software utilizado

- **Epic Games:** Fundada en 1991 por Tim Sweeney, *Epic Games* es una de las empresas más importantes en el desarrollo de videojuegos y motores gráficos. Originalmente, la compañía era conocida como *Potomac Computer Systems* y, posteriormente, *Epic MegaGames*. Inicialmente, la empresa desarrollaba juegos *shareware*, destacando títulos como la versión de *ZZT* y *Jazz Jackrabbit*. Su estrellato llegó en 1998, cuando se lanzó *Unreal*, un *shooter* en primera persona que, además de convertirse en una franquicia de éxito, introdujo UE, el motor gráfico más importante de la compañía. En el transcurso de los años, *Epic Games* ha desarrollado sagas enormes, como *Gears of War* y *Fortnite* (Figura 2-28), el juego que ha cambiado las reglas de actuación en la facturación a través de micro transacciones y el juego como servicio. La compañía también es la propietaria de series como *Psyonix* y *Quixel*. *Epic Games Store* apareció en 2018 para competir con *Steam*, ofreciendo mejores condiciones a los desarrolladores. Desde el punto de vista financiero, también han recibido inversiones de *Tencent Holdings* con el fin de poder expandir su ecosistema. A día de hoy, continúan innovando con UE5, utilizado en el sector de

los videojuegos, y se han convertido en uno de los pilares fundamentales para la evolución del entretenimiento digital [41].



Figura 2-28 Escenario de *Fortnite* y logotipo de *Epic Games* [41] [42].

- **Steam:** Es otra plataforma de distribución digital de videojuegos que tuvo su origen en 2003, siendo su versión inicial, una edición destinada a gestionar las actualizaciones de juegos. Hoy en día, se ha añadido la compra y la distribución de juegos de terceros, permitiendo comprar y gestionar videojuegos, protegiéndose así de la piratería. Ofrece emparejamiento multijugador, descargas en la nube, chat de voz, retransmitir videojuegos a través de *streaming* y servicios sociales. A lo largo de su historia ha evolucionado con ciertas herramientas como *Steam Workshop*, cuya función es crear y distribuir cierto contenido generado por los usuarios, *Steam Greenlight* y *Steam Direct*, que permite a los desarrolladores independientes publicar los juegos creados por ellos, y *Steam Cloud*, encargado de la sincronización de datos de videojuegos. Ha desarrollado también, hardware y aplicaciones como *Steam Link*, que permite jugar en los dispositivos de forma remota [43]. De todas formas, la subvertiente importante para el trabajo es la de *Steam VR* (Figura 2-29), dedicada a la conexión de visores con el ordenador. *Steam VR* es una plataforma de RV que permite a los usuarios acceder y jugar títulos compatibles con RV en *Steam* [44].



Figura 2-29 Interfaz de la sala general de control de *Steam VR Home* [45].

- **Vive Hub:** Es un software de *HTC Vive*, una línea de visores de RV y mixta desarrollada por *HTC Corporation*, que incluye dispositivos para PC y modelos independientes como *Vive Focus*, *Vive Flow* y *Vive Elite XR*. Permite gestionar y configurar los dispositivos de RV, además de servir como un centro de control donde los usuarios pueden ajustar los parámetros del visor, calibrar los controladores y acceder a contenido optimizado para la plataforma. También facilita la integración con *Steam VR*, garantizando la compatibilidad con juegos y aplicaciones desarrolladas para esta tecnología [46].

2.5.2 Equipos y material

El desarrollo de este trabajo se ha llevado a cabo en el Laboratorio de RV del Centro Universitario de la Defensa (CUD), ubicado en la Escuela Naval Militar. Este entorno ha proporcionado las herramientas necesarias para diseñar, programar y probar el comportamiento de los personajes dentro del mundo virtual creado en UE5.5, permitiendo la generación de una experiencias inmersiva y realista para la simulación del escenario.

Para lograr un rendimiento adecuado, se ha hecho uso de un ordenador de alta potencia, equipado con un procesador avanzado capaz de manejar gráficos detallados y simulaciones complejas (Figura 2-30). Combinado con elementos estándar como el teclado, ratón y una pantalla de alta resolución. La pieza clave para el procesamiento visual es la tarjeta gráfica EVGA GeForce RTX 3080 que posee el equipo, que ha permitido que los entornos virtuales se mostraran con gran fluidez y nitidez.

Para sumergirse en el mundo virtual, se ha hecho uso del visor HTC Vive Focus Vision, un dispositivo que funciona sin necesidad de cables ni sensores externos, gracias a su sistema de seguimiento integrado. Su conexión con UE5.5 se realizó mediante *VIVE Hub* y *Steam VR*, lo que permitió aprovechar sus pantallas de alta resolución y reducir el efecto *screen door*, brindando imágenes más limpias. Además, sus controles hápticos mejoraron la sensación de interacción con los objetos dentro del entorno.

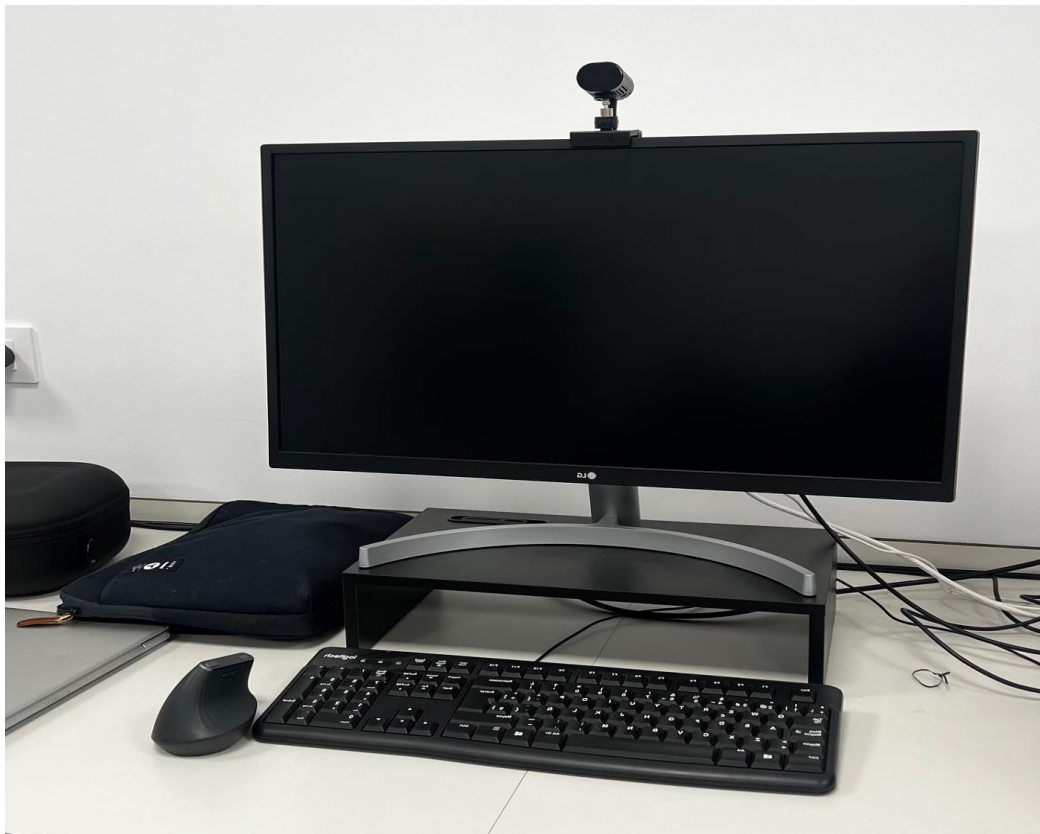


Figura 2-30 Pantalla de ordenador, ratón y teclado del equipo de trabajo (Autoría propia).

2.5.2.1 HTC Vive Focus Vision

El HTC Vive Focus Vision (Figura 2-31) es un visor de RV y realidad mixta creado por HTC, que puede operar de forma independiente o estar conectado a un PC. Presenta una resolución de 2448x2448 píxeles por ojo, un campo de visión de 120°, una tasa de refresco de 90 Hz, seguimiento ocular y *passthrough* a color gracias a sus cámaras de 16 MP. Viene con 12 GB de RAM, 128 GB de almacenamiento ampliable y un seguimiento de manos muy preciso gracias a las cámaras infrarrojas [47].



Figura 2-31 Visor HTC Focus Vision y mandos de control [48].

Los mandos del visor HTC Vive Focus Vision cuentan con seis grados de libertad, botones y gatillos ergonómicos, además de sensores táctiles para un control preciso e intuitivo. Son compatibles con otros visores de la línea Vive, como el Vive XR Elite y el Vive Focus 3, permitiendo una experiencia unificada. Además, pueden complementarse con el *VIVE Ultimate Tracker* para mejorar el seguimiento y la inmersión en entornos de RV más exigentes [49].

2.5.2.2 Tarjeta Gráfica NVIDIA GeForce RTX 3080

La EVGA GeForce RTX 3080 (Figura 2-32) es una tarjeta gráfica diseñada para dar un alto rendimiento. Cuenta con la arquitectura *NVIDIA Ampere* soportando tecnologías como *Ray Tracing* y *DLSS* para ofrecer gráficos ultrarrealistas y ofrecer una experiencia más inmersiva en juegos de alta exigencia. La EVGA GeForce RTX 3080 tiene un sistema de refrigeración *iCX3* con triple ventilador para mantenerse limpio. Con un diseño sólido y componentes de alta calidad para estabilidad y durabilidad, cuenta también con iluminación RGB ajustable y de herramientas como *Precision X1* para poder ajustar el rendimiento y la estética al gusto, adecuada para resoluciones de hasta 8K y conexiones modernas como HDMI 2.1. Esta gráfica es óptima para la creación de videojuegos y escenarios que requieran alta resolución gráfica, como es el caso de este trabajo de fin de grado [51].



Figura 2-32 Tarjeta Gráfica NVIDIA GeForce RTX (Autoría propia)

3 DESARROLLO

3.1 Descripción del apartado

Este apartado tiene como finalidad exponer de forma organizada el desarrollo de la programación del trabajo, abarcando aspectos como la integración de la IA de los enemigos, la IA de los soldados de la compañía, la creación de un escenario apto para que estos se desenvuelvan, así como el contexto del ejercicio del trabajo. Para ello se ha hecho uso del programa UE, la versión 5.5, que ha permitido crear un ejercicio de RV que permita al jugador tomar decisiones, y aplicar las ROE para su entrenamiento personal.

Para conseguir esto, se ha seleccionado una plantilla en tercera persona (*Third Person*), que facilita visualmente el movimiento y acciones de los enemigos. A continuación, en la Figura 3-1, se muestra de forma esquematizada la secuencia del proyecto desde el inicio hasta su finalización.

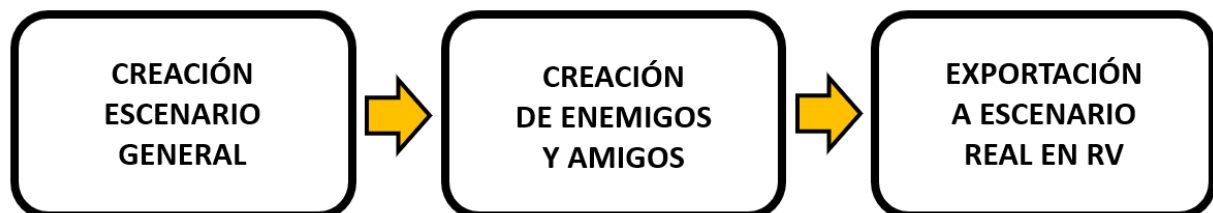


Figura 3-1 Esquema general de la consecución del trabajo (Autoría propia).

Cada etapa ha sido la base de la siguiente, para así poder llegar a una correcta resolución del ejercicio final, creando de esta forma una serie de eventos entrelazados e indispensables. A continuación, se explica de forma breve cada uno de los apartados, los cuales serán desarrollados más adelante:

1. Creación del escenario: Para poder dar una mayor visión a la programación de los enemigos, se ha creado un escenario simple y visual que permita centrar su objetivo en las acciones de los personajes.
2. Creación de los enemigos y soldados de la compañía: Se expone la programación en profundidad de los enemigos a través de los BP, así como la programación del equipo del personaje principal.
3. Exportación a un escenario real de combate en RV: Para formalizar el uso de los enemigos, estos se han exportado al escenario de combate PICOZU realizado por Diego Garabito de la Fuente, adaptándolo para que los personajes cumplan las funciones que deben realizar. Asimismo, se ha exportado el trabajo a un *template* de RV con las *meshes* y armamento real usado en España.

3.2 Escenario

Como se ha comentado anteriormente, es imprescindible buscar un escenario que contenga los elementos necesarios para crear un ambiente donde los personajes puedan realizar todo tipo de movimientos y acciones, aunque el escenario final es el de PICOZU, para visualizar de manera más clara como los personajes actúan o reaccionan al entorno y al encuentro con otros personajes, se ha escogido una plantilla *Third Person* en UE. Esta plantilla o *template* permite ver a los personajes desde un punto de vista espacial, permitiendo una visión más clara del entorno.

3.2.1 Concepto del escenario

El marco legal en el que se desarrolla el ejercicio es un factor clave, especialmente cuando las fuerzas armadas operan en apoyo o ausencia de las autoridades civiles. La normativa aplicable es determinada por asesores legales a nivel estratégico, y aunque el comandante táctico debe estar informado, no le corresponde tomar estas decisiones. Las operaciones de seguridad militar están reguladas por el derecho civil y el Derecho Internacional de los Derechos Humanos (DIDH), a diferencia de las operaciones de combate, que se rigen por la LOAC. En estos escenarios, los militares actúan como fuerzas de seguridad, priorizando la protección de la población y la reducción del uso de la fuerza letal, conforme a los Principios Básicos de las Naciones Unidas sobre el Uso de la Fuerza y Armas de Fuego [52].

El marco legal también influye en el cambio entre estados de conflicto y tiempos de paz, requiriendo que los comandantes ajusten sus tácticas y para garantizar el cumplimiento de la normativa vigente en función de la situación. En cada país, la legislación establece si las FAS pueden intervenir en apoyo de las autoridades civiles y en qué condiciones, siendo esta una decisión de las autoridades políticas o judiciales. Durante la operación, es fundamental equilibrar la importancia militar con las implicaciones legales y humanitarias, asegurando una ejecución conforme al derecho.

La preparación de inteligencia del entorno operativo se ha convertido en una herramienta clave para las operaciones de seguridad, proporcionando información sobre el terreno y facilitando la toma de decisiones estratégicas. El análisis del entorno considera tanto factores físicos, como rutas y ubicaciones clave, como aspectos humanos, incluyendo patrones de comportamiento y riesgos para la población civil. La cooperación con autoridades civiles y líderes comunitarios fortalece la gestión de crisis y permite ajustar estrategias de forma efectiva [52].

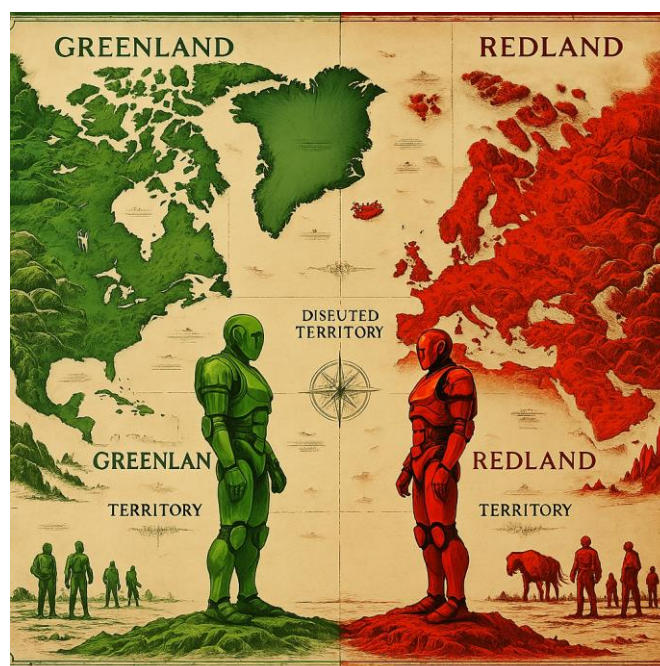


Figura 3-2 Situación del conflicto entre Redland y Greenland (generado por IA).

Este trabajo recrea un ejercicio de combate en población basado en procedimientos de Infantería de Marina dentro de un entorno de RV. La simulación introduce dos países ficticios: Redland y Greenland (Figura 3-2). Redland, con un historial de conflictos armados e incursiones militares, mantiene una relación tensa con Greenland. En este contexto, el usuario, equipado con gafas de RV, lidera un equipo de operaciones especiales compuesto por tres soldados con la misión de recuperar un territorio que Redland arrebató a Greenland años atrás.

El ejercicio de RV ofrece una experiencia táctica inmersiva donde el jugador debe tomar decisiones estratégicas en tiempo real, siguiendo las ROE para evitar una escalada del conflicto. Dado que el estado entre Greenland y Redland es de *status quo*, el uso de la fuerza debe ser proporcional y necesaria, priorizando la neutralización de amenazas sin violaciones a las reglas operativas.

El objetivo principal del equipo es capturar a un líder enemigo de Redland (*BP_TargetEnemy*), refugiado en una casa dentro del área de operaciones. La misión requiere de precisión en la ejecución, por lo que el usuario debe coordinar cada movimiento y dar órdenes a su equipo para ingresar al objetivo de manera organizada y eficiente. Antes de la incursión, es esencial identificar posibles amenazas y establecer posiciones estratégicas para minimizar riesgos.

Durante la operación, el jugador se enfrentará a distintos tipos de enemigos con comportamientos programados específicos. Entre ellos, un francotirador, que representa una amenaza de largo alcance y debe ser neutralizado antes de avanzar; tiradores hostiles patrullando el área, que reaccionan agresivamente si detectan al equipo; un enemigo armado con un machete, que intentará atacar en combate cercano; enemigos en patrulla, que recorren zonas clave y requieren una aproximación sigilosa; un enemigo que intenta escapar, cuya huida podría alertar a otros hostiles; y un enemigo que se rinde, cuya captura sin recurrir a la violencia.

3.2.2 Creación del escenario desde UE

Para iniciar el desarrollo del escenario, lo primero que se ha hecho es abrir la aplicación de UE y crear un nuevo proyecto. Dentro del menú de selección de plantillas, en la sección “Games” de la interfaz de entrada, se ha escogido la *Third Person Template*, como se puede observar en la Figura 3-3. Este *template* proporciona una base inicial idónea, ya que cuenta con una configuración estructurada que incluye elementos y personajes base, ofreciendo un punto de partida óptimo para el desarrollo del entorno. Al seleccionar *Starter Content*, UE proporciona automáticamente una serie de elementos prediseñados que resultan útiles para la fase inicial del diseño. Estos objetos permiten simular estructuras como edificios, puentes y calles, facilitando la visualización y planificación del escenario antes de exportarlo a un escenario real.

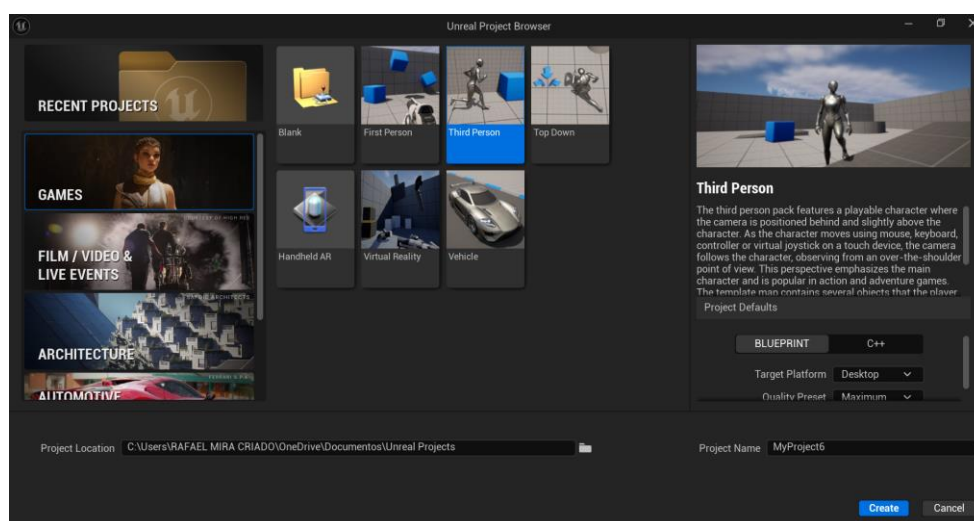


Figura 3-3 Interfaz de entrada de UE y sus distintos *templates* (Autoría propia).

La iluminación dentro del *template*, está compuesta por una *Directional Light*, que simula la luz del sol y genera sombras realistas en el escenario. También se incluye una *Sky Light*, la cual aporta iluminación ambiental y suaviza los contrastes en las áreas de sombra. Además, la escena cuenta con un *Sky Sphere*, que define el cielo y permite simular diferentes condiciones atmosféricas. La plantilla a su vez proporciona un conjunto básico de texturas y materiales que pueden aplicarse a los objetos del escenario. Estos materiales incluyen opciones para superficies metálicas, de madera y de piedra, permitiendo la personalización de los elementos sin necesidad de crearlos desde cero.

Los objetos y estructuras presentes en el *template* consisten mayoritariamente en plataformas, rampas y bloques de diferentes tamaños, los cuales están diseñados para probar las capacidades de movimiento del personaje. Estos elementos están contruidos con *Static Meshes*, asegurando que mantengan su forma y posición dentro del nivel. También se incluyen BP preconfigurados que permiten interactuar con ciertos objetos, como puertas o plataformas móviles, lo que sirve como referencia para añadir nuevas mecánicas en el desarrollo del proyecto.

El sistema de físicas del *template* está integrado en el personaje y en los objetos del entorno. Gracias a la implementación de un *Collision System*, el personaje detecta superficies y reacciona de manera realista al contacto con ellas, subiendo las rampas o chocando con objetos provocando su movimiento. Además, el *template* incorpora *Physics Materials*, que permiten definir el comportamiento de los objetos en función de su material, como superficies resbaladizas o de alta fricción.

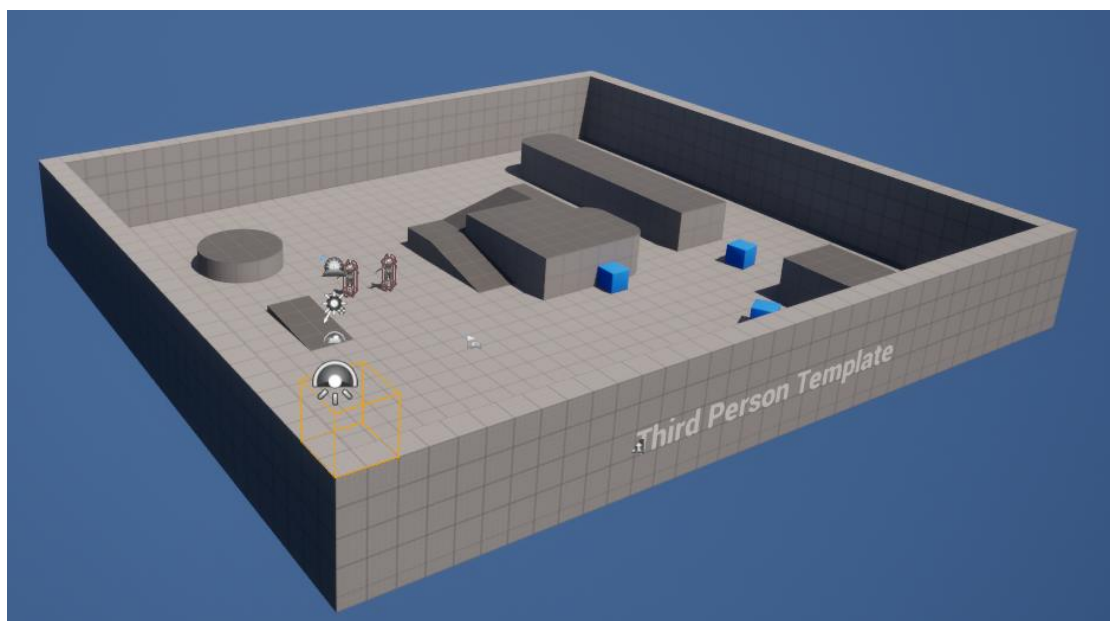


Figura 3-4 *Third Person Template* por defecto de UE (Autoría propia).

3.2.3 Modelado y creación del escenario final

El escenario se ha estructurado dividiéndolo en cinco zonas principales, cada una destinada a representar diferentes situaciones de combate y facilitar la prueba de distintos comportamientos de la IA en los enemigos y aliados. Estas zonas incluyen entornos variados como espacios abiertos, áreas cerradas, zonas de cobertura, puntos de emboscada y sectores de patrullaje, lo que permite simular un amplio abanico de condiciones tácticas. Asimismo, permite que los personajes puedan interactuar con el entorno de manera realista, inteligente y ajustada a sus roles dentro de la simulación, favoreciendo así un desarrollo más preciso de rutinas de percepción, decisión y acción que reflejen comportamientos adaptativos y coherentes en tiempo real.

En el centro del mapa se ha ubicado al *BP_EnemyMelee*, un enemigo armado con un machete cuya IA está programada para detectar al jugador por la vista, el sonido y el daño recibido. Esta zona está

rodeada de obstáculos estratégicos que permiten evaluar cómo el enemigo reacciona cuando pierde al jugador de vista. Su comportamiento incluye buscar en áreas cercanas, realizar ataques con diferentes animaciones, saltar sobre estructuras utilizando *NavLinks* y simular un combate cuerpo a cuerpo dinámico y adaptativo.

En el sector norte (N) se ha diseñado un espacio con plataformas y estructuras a diferentes alturas, simulando un entorno urbano con edificios. Esta zona está destinada al *BP_PatrolEnemy*, un enemigo que sigue una ruta predefinida patrullando el área. La programación de este enemigo permite evaluar su capacidad de detección del jugador en movimiento, así como su respuesta ante situaciones en las que se interrumpe su rutina, ya sea por la presencia del jugador o por estímulos externos.

En el sector este (E) se ha introducido una casa simulada, en la que se encuentra el *BP_EnemyRanged*, un enemigo armado con un rifle que tiene la capacidad de atacar a distancia determinada. Esta zona está diseñada para analizar el comportamiento de este enemigo en situaciones de combate prolongado, observando cómo busca cobertura cuando su salud baja a un nivel crítico. Además, su IA le permite decidir entre mantenerse en combate o retirarse estratégicamente para curarse o reubicarse, haciendo que los enfrentamientos sean más realistas.

El sector oeste (W) está destinado a la presencia del *BP_ShooterLongRangedEnemy* y los tres aliados que forman parte del equipo del jugador (*BP_FriendTeam*). Se han ubicado estructuras elevadas para el francotirador, lo que le permite obtener una visión clara de la zona y simular un enfrentamiento desde larga distancia. La inclusión de los aliados en esta zona permite evaluar cómo interactúan con el jugador y cómo se integran en el combate en función de las órdenes recibidas.

En el sector sur (S) se han incorporado varias áreas pequeñas especializadas en el comportamiento de enemigos con reacciones específicas. En la parte izquierda se encuentran dos subzonas donde se han colocado el *BP_SurrenderEnemy*, un enemigo que al detectar al jugador decide rendirse en lugar de atacar, y el *BP_RunningAwayEnemy*, un enemigo que huye inmediatamente al detectar al jugador, buscando refugio en un área lejana del mapa.

Además, dentro del mismo sector sur, se han posicionado dos enemigos de distintos equipos en combate entre sí. Esto permite simular situaciones de fuego cruzado y daño colateral, mostrando cómo los enemigos pueden interactuar entre sí y cómo el jugador puede encontrarse con escenarios en los que otras facciones ya están enfrentadas.

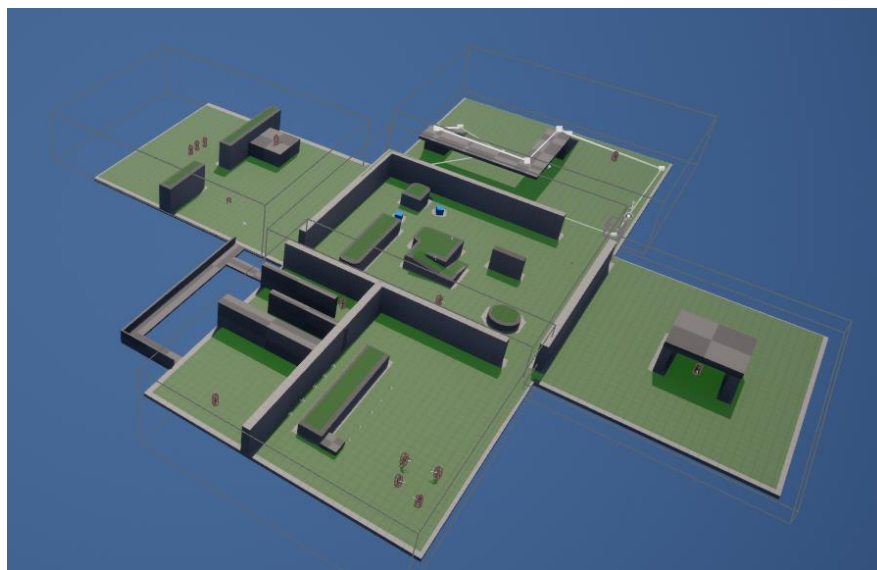


Figura 3-5 Plantilla final del escenario (Autoría propia).

Cada elemento del escenario ha sido colocado con un propósito específico, permitiendo que cada tipo de enemigo tenga un entorno optimizado para sus acciones y asegurando que el jugador pueda

experimentar distintas dinámicas de combate en un mismo espacio. A lo largo del desarrollo del proyecto, se detallará en profundidad la programación de cada enemigo y su integración con el entorno para garantizar una experiencia de entrenamiento en RV eficaz y funcional.

3.3 Creación de enemigos y soldados del equipo del personaje

Una vez establecidas las bases del escenario, en este apartado se procederá a explicar detalladamente la programación desde cero de los enemigos y de los miembros del equipo del *Player Character* (personaje principal del juego). Se abordará el proceso desde la creación inicial de BT básicos y funcionales hasta el desarrollo avanzado de los comportamientos individuales de cada enemigo.

3.3.1 Creación inicial de *BP_Player* y *BP_EnemyBase*

Es importante mencionar que, a lo largo del desarrollo del proyecto, todos los elementos han sido nombrados de manera estructurada y conforme a su tipo. Por ejemplo, “BP_” para *Blueprints*, “E_” para *Enums*, “M_” para *Montages*, “A_” para *Anims* y “BTT_” para *Blueprint Tasks*, entre otros. Este sistema de nomenclatura facilita la organización del código y permite identificar rápidamente la función de cada elemento dentro del proyecto. Lo primero que se ha realizado es la creación del *Player Character* (*BP_Player*) y del *BP_EnemyBase* (BP de tipo *Character* que sirve como base para todos los enemigos). Para comenzar, se ha duplicado el *Third Person Character* que viene por defecto en el *template*. De esta manera, se ha obtenido el *BP_Player*, conservando la estructura base del original. Una de las principales ventajas de duplicar el *Third Person Character* es que su sistema de nodos de programación para el movimiento ya está preconfigurado. Esto incluye acciones como correr, saltar, las físicas del personaje, la detección de colisiones mediante *Box Collision* y su representación en el entorno a través de *Static Meshes*, entre otros aspectos esenciales.

El *BP_EnemyBase* en cambio, se ha creado desde cero partiendo de un BP de tipo *Character*. Esto significa que su sistema de movimiento y acciones no depende de las configuraciones preestablecidas en el personaje por defecto del *template*, sino que ha sido desarrollado de manera independiente. Cada comportamiento ha sido programado específicamente, permitiendo un mayor control sobre la IA.

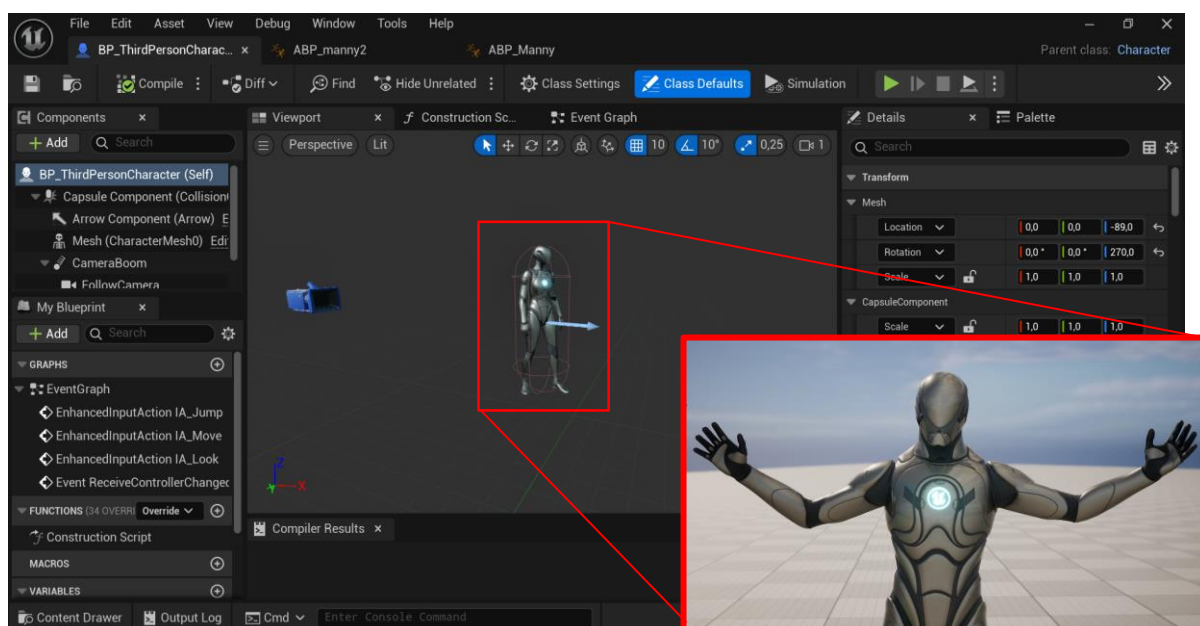


Figura 3-6 Apariencia del *Static Mesh* por defecto del *ThirdPersonCharacter* (Autoría propia).

Como se muestra en la Figura 3-6, en la parte superior central hay tres pestañas. La primera es el *Viewport*, que permite visualizar al enemigo y cómo se verá en el escenario. El *Construction Script*, que se ejecuta únicamente cuando el objeto se coloca o actualiza en el nivel, antes de que comience el juego.

Su función es modificar propiedades estáticas como la apariencia, la escala o la salud de los enemigos, asegurando que esté correctamente configurado sin afectar el rendimiento durante la ejecución. El *Event Graph* maneja eventos en tiempo real, como el movimiento, la interacción y el comportamiento de la IA. Es el encargado de ejecutar acciones continuamente según las condiciones del juego, permitiendo que los personajes o elementos reaccionen a lo que ocurre en la escena.

Por otro lado, para comprender cómo un personaje puede moverse en el mapa, es fundamental entender el concepto de las animaciones. Estas, a través de código basado en nodos, son las encargadas de asignar al personaje las animaciones adecuadas que ejecutar para cada situación. En la parte derecha de la Figura 3.6, se puede observar el panel de *Details*, donde, una vez seleccionado el *BP_Player*, es posible definir el tipo de animación que se quiere utilizar en el *Character*. En el caso de los enemigos, se empleará el *ABP_Manny* (ABP hace referencia a *Animation Blueprint*), una animación predefinida de UE que incluye acciones básicas como saltar, correr y caminar, proporcionando una base sólida para el comportamiento de los personajes dentro del juego.

El *ABP_Manny* tiene un funcionamiento base que da sentido a la lógica de movimientos del personaje. El sistema está compuesto por un *State Machine* que, en UE, es esencial para que un NPC pueda moverse de manera fluida y coherente. Básicamente, funciona como un conjunto de reglas que determinan qué animación debe reproducirse en cada momento según el estado del personaje. El *State Machine* y *Locomotion* (Figura 3-7), controlan la transición entre animaciones clave como *Idle* (reposo), *Walk/Run* (caminar/correr), *Jump* (saltar), *Fall Loop* (caída) y *Land* (aterrizaje). Esto significa que el NPC no se mueve de manera brusca o descoordinada, sino que cambia de animación de forma progresiva según su estado actual.

Por ejemplo, si el NPC está quieto, su animación estará en *Idle*. Cuando comienza a moverse, el *State Machine* detecta el cambio de velocidad y activa la animación de *Walk/Run*, asegurando una transición similar a una acción humana. Si el personaje salta, la animación cambia a *Jump*, y si sigue en el aire, entra en la animación de *Fall Loop*. Una vez que toca el suelo, se reproduce la animación de *Land*, permitiendo un aterrizaje natural antes de volver a *Idle* o *Walk/Run* según sea necesario.

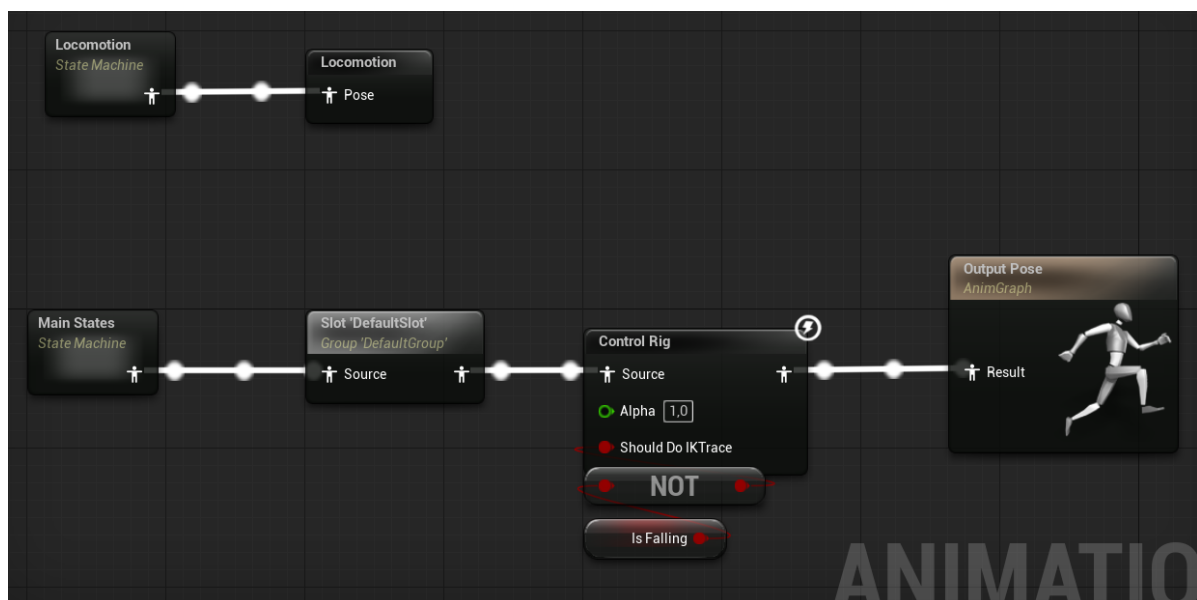


Figura 3-7 Lógica de *ABP_Manny* que usan los enemigos de la simulación (Autoría propia).

Una vez establecido cómo el personaje puede moverse y ejecutar animaciones en función de su desplazamiento, es fundamental comprender el funcionamiento de la interconexión de nodos en UE y cómo, a través de los BP, el jugador es capaz de realizar ciertas acciones. Existen diversas opciones para programar esta lógica, ya sea utilizando C++, los nodos *AI Move To* o mediante el uso de BT y BB. Se ha comenzado implantando estas acciones utilizando los nodos *AI Move To*, los cuales definen la lógica

de acción del *BP_EnergyBase* directamente desde el *Event Graph* de su BP. En concreto, se han empleado los nodos mostrados en la Figura 3-8, donde, de manera simplificada, se ha insertado un evento personalizado (*Custom Event*) llamado *Chase Player*. Este evento llama al *Pawn Self* (haciendo referencia al *BP_EnergyBase* como actor que actúa de *Pawn*) y le asigna como *Target Actor* al *Player Character* (es decir, el usuario que interactúa con gafas de RV, teclado u otro dispositivo de control). Posteriormente, se ha introducido un *Delay* de 2 segundos para que, una vez completada la acción, el *BP_EnergyBase* vuelva a buscar al jugador llamando nuevamente a la misma función, creando así un sistema de persecución continuo.

Cabe destacar que, aunque El *BP_EnergyBase* tenga esta lógica en su *Event Graph*, para que realmente pueda moverse es necesario agregar un *Nav Mesh Volume* al mundo que cubra el área transitable. Sin este volumen, el personaje no podrá moverse correctamente.

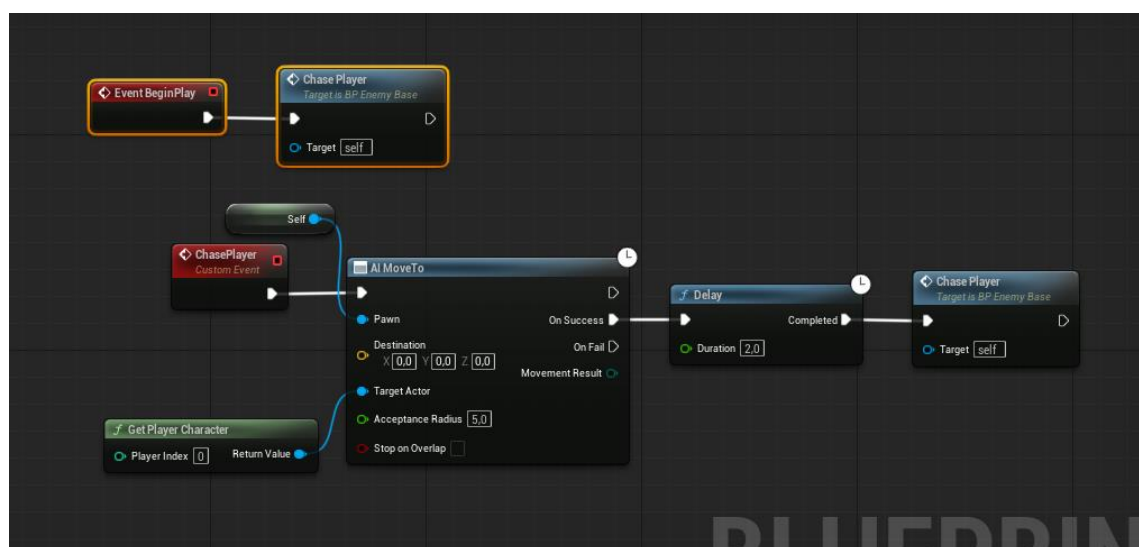


Figura 3-8 Función de persecución en el *Event Graph* del *BP_EnergyBase* (Autoría propia).

3.3.2 Introducción a los BT y BB

A pesar de que la programación puede realizarse mayoritariamente en el *Event Graph* y el *Construction Script* del personaje, concentrar toda la lógica en estas secciones haría que la comprensión del funcionamiento de la IA y del enemigo resultara tediosa. Para estructurar y organizar mejor cómo los personajes piensan y actúan, se recurre a los BT y las BB previamente explicados en el apartado 2.4.4. La sustitución de la lógica representada en la Figura 3-8 por BT y las BB es un proceso sencillo. De manera general, los pasos seguidos han consistido en la creación de un BT llamado *BT_EnergyBase* y una BB llamada *BB_EnergyBase*.

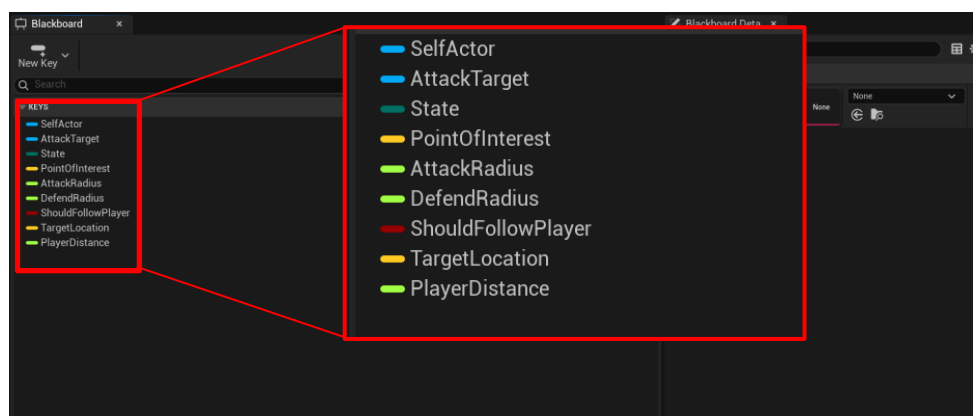


Figura 3-9 *BB_EnergyBase* con las variables de toda la programación (Autoría propia).

La *BB_EnergyBase*, como se muestra en la Figura 3-9, funciona como un almacén de variables que el programa actualiza en tiempo real para su uso dentro del BT. En este caso, se ha introducido una variable de tipo *Object* llamada *AttackTarget* (entidad a atacar), que almacenará la referencia del *Player Controller* y, así en el futuro, la de cualquier otro personaje que deba ser atacado por la IA controlada por el BT. Es de hecho, la variable más importante de la simulación puesto que almacena como el enemigo ve al jugador, para así saber que hacer frente a su presencia, en función de lo que este lleve programado. Al final del juego la variable *AttackTarget* adaptará la variable al jugador con las gafas de RV.

Sin embargo, el simple hecho de agregar una variable en la BB no implica que esta reconozca automáticamente quién es el *AttackTarget*. Para que la BB identifique correctamente este objetivo, se ha creado un BP de tipo *AIController*, el cual controla la lógica de IA guiando el comportamiento de los personajes en la simulación. Este *AIController* es la base de todo el comportamiento de los personajes en la simulación, evitando programar la lógica de comportamiento de manera individual en cada uno de los enemigos. Dentro del BP, cuya interfaz y funciones son iguales a cualquier otro BP, se ha configurado a la IA para que utilice el BT correspondiente.

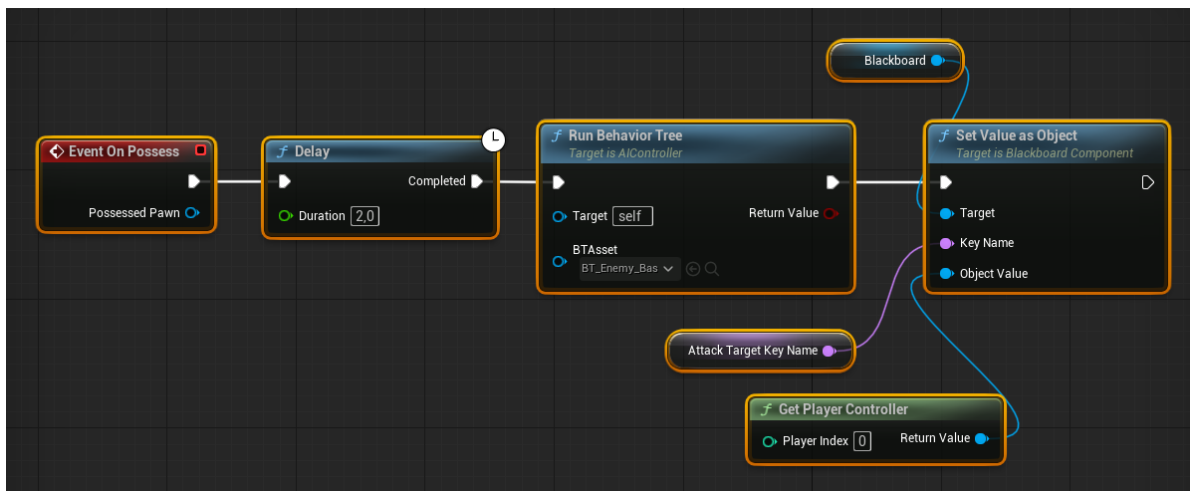


Figura 3-10 Event Graph del AIController (Autoría propia).

Como se observa en la Figura 3-10, al poseer un actor, el *AIController* ejecuta el *BT_EnergyBase* y, posteriormente, asigna la variable *AttackTarget* al *Player Controller*, asignándole el valor que se busca. Es importante que en el *Key Name* del nodo *Set Value As Object*, el valor de la variable *Attack Target Key Name* sea exactamente el nombre de la variable puesta en *BB_EnergyBase*, para que la llame correctamente. En adición se ha añadido un *delay* para dar tiempo al *AIController* a poseer al actor sin que se produzcan *debugs*.

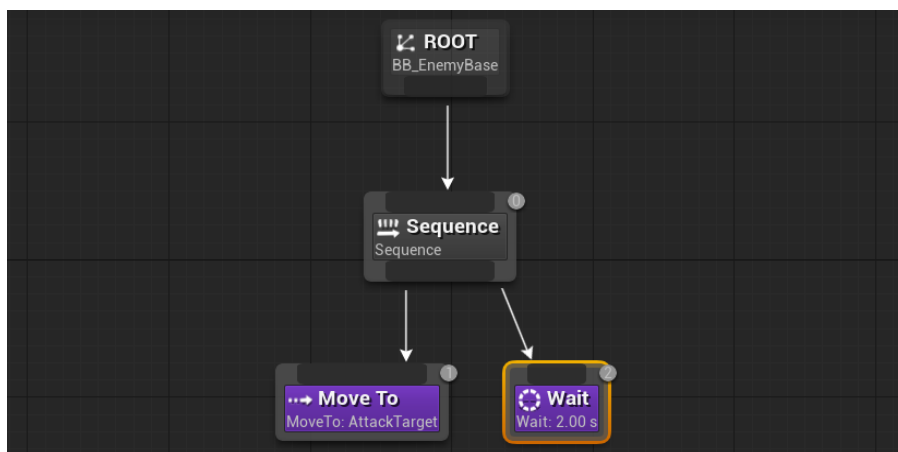


Figura 3-11 BT básico de persecución (Autoría propia).

Una vez que la variable ha sido almacenada correctamente en la BB, se procede a su implementación en el BT, estructurando su acción de persecución. Lo que se muestra en la Figura 3-11, es un BT donde el enemigo va a la posición del *BP_Player* a través del nodo *Move To* (con la *key* establecida en *AttackTarget*), espera dos segundos y vuelve a ejecutarlo en secuencia.

Lo siguiente que se ha realizado ha sido dotar al enemigo de un ataque para que, al llegar a la posición del *BP_Player*, pueda atacarlo. Para ello, ha sido necesario crear un *BTTask*. Actualmente, los nodos *Move To* y *Wait*, mostrados en la Figura 3-11, vienen por defecto en UE y forman parte de los *BBtasks*. Del mismo modo, es posible crear tareas personalizadas para que el enemigo ejecute las funciones específicas que el usuario desee. Por lo que para el ataque se ha creado una *Task* personalizada, llamada *BTT_DefaultAttack*. Esta tarea posee una lógica que identifica al enemigo con *Cast To BP_EnergyBase* y obtiene el objetivo desde la BB con *Get Blackboard Value as Actor*. Luego, activa la función *Attack*, indicando a quién atacar, y vincula el evento *On Attack End* para detectar cuándo termina la acción. Finalmente, *Finish Execute* marca la finalización de la tarea, permitiendo que la IA continúe con la siguiente acción programada (Figura 3-12). De esta manera al implementar la tarea en el BT, el enemigo va al *BP_Player*, le ataca, espera y repite.

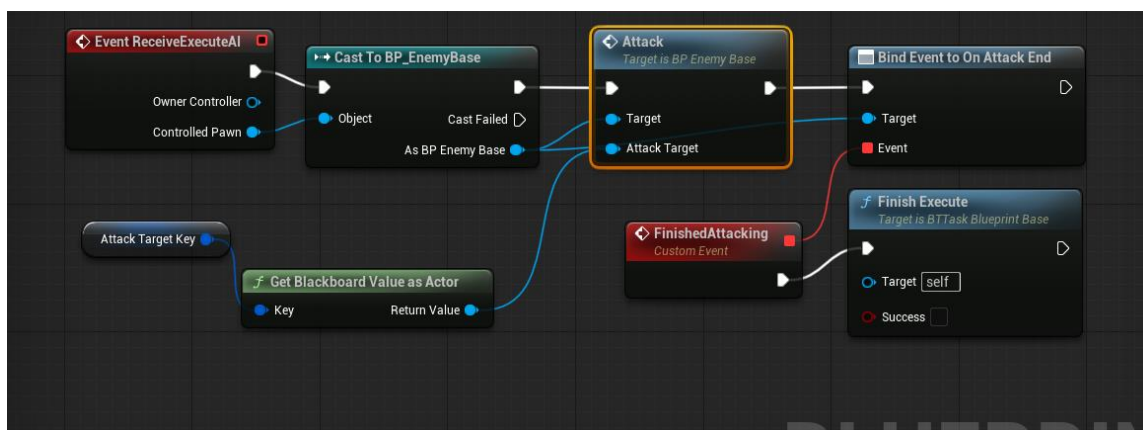


Figura 3-12 *BTT_DefaultAttack* (Autoría propia).

Para que el movimiento de ataque se asemeje a un ataque realista, se ha descargado desde *FAB*, plataforma de *Epic Games Launcher* que permite obtener elementos, animaciones y otros recursos, un *Montage* que simula un ataque con machete. Esta animación será utilizada posteriormente por el *BP_EnergyMelee*, mientras que al *BP_RangedEnemy* se le asignará su propio *Montage* adecuado. Asimismo, se ha descargado el paquete *AnimStarterPack*, el cual contiene diversas animaciones útiles, con un *mesh* de tipo *Character* con un esqueleto rosa (Figura 3-13), para los enemigos, permitiendo seleccionar la más adecuada para cada caso.

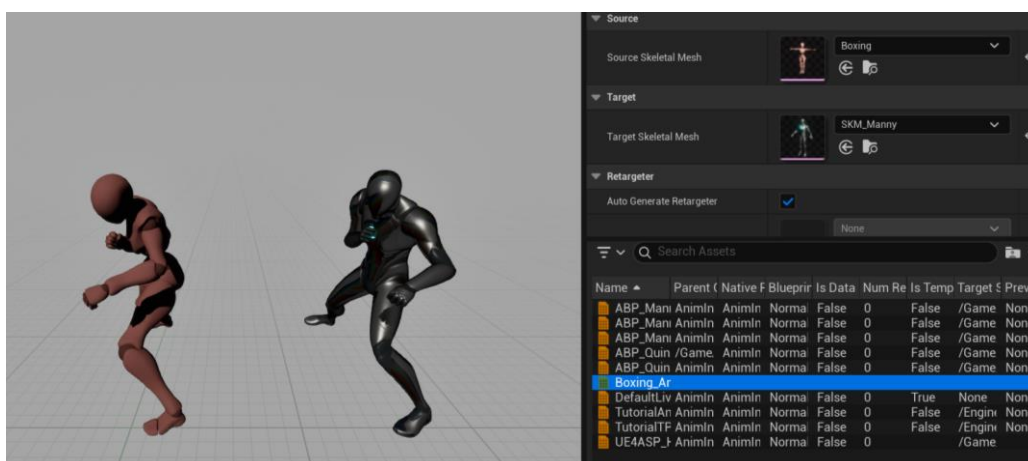


Figura 3-13 *Retarget* de la animación de ataque del enemigo (Autoría propia).

Una vez descargadas estas animaciones, se ha de hacer con ellas *retargeting*, ya que los huesos del esqueleto del personaje por defecto no son los mismos que los del *BP_Player*, el cual utiliza el esqueleto de *Manny* (personaje con *Static Mesh* y *Skeleton* predeterminados en UE). Para llevar a cabo el *retargeting*, solo es necesario seleccionar la animación, asignarle el nuevo esqueleto y exportar la nueva animación. En la Figura 3-13 se puede observar cómo ambos esqueletos realizan la animación de atacar en el proceso de *retargeting*.

Para implementar este *Montage* se ha accedido al *Event Graph* del *BP_EnemyBase*, agregando el nodo de *Attack* por defecto de UE, que es utilizado por la *Task* del BT mostrada en la Figura 3-12, y luego llamar al nodo *Play Montage*. De esta manera, se establece que, al ejecutar un ataque, el enemigo reproduzca la animación correspondiente al *Montage*.

La lógica hasta el momento es que el enemigo ataca, pero si *BP_Player* se mueve, el enemigo no lo sigue con la mirada y continúa atacando al aire, lo cual resta realismo a la acción. Para solucionar esto, se ha creado una *Task* llamada *BTT_FocusTarget*, la cual, de manera simplificada, utiliza el nodo *Set Focus* por defecto y establece como objetivo de enfoque la variable *Attack Target Key*, de tipo *Blackboard Key Selector* (ya mencionada en la Figura 3-12). Como consecuencia de esta nueva *Task*, el enemigo, con su actualizado BT que incluye *BTT_FocusTarget*, se dirige hacia la posición del *BP_Player*, sigue la secuencia establecida para encararlo, lo ataca y posteriormente espera.

Si el *BP_Player*, que tiene la capacidad de saltar implementada en su lógica, se encuentra sobre una superficie elevada a la que solo se puede acceder mediante una rampa, el enemigo no podrá saltar directamente. En su lugar, seguirá el camino más accesible utilizando la rampa, con el problema de estar fijando su mirada al *BP_Player* mientras realiza la acción. Para ello se ha creado otra *Task*, llamada *BTT_ClearFocus* que lo que hace es contrarrestar el nodo de *Set Focus* permitiendo al *BP_EnemyBase* ir al objetivo mirando por donde va y no encarando a *BP_Player*.

En este punto, el enemigo es capaz de atacar y buscar a *BP_Player*, pero actualmente no posee un arma real, sino que simplemente ejecuta el movimiento de la animación. Por ello, se ha descargado un machete desde *FAB*, como se muestra en la Figura 3-14, para equiparlo en sus *Montages* y animaciones, asegurando así un mayor realismo visual en el ataque. Una vez importados el *Static Mesh*, el esqueleto y las físicas del machete, se crea un BP de tipo *Actor* en el que se coloca el machete como *Static Mesh* del BP. Como segundo paso, se ha creado un *socket* en la mano, lugar donde se ubicará el machete. Finalmente, la lógica se ha implementado en el *Event Graph* del *BP_EnemyBase*. Para hacerlo, se ha creado un *Custom Event*, llamado *WieldKnife*, el cual utiliza el nodo *SpawnActor* (que hace referencia al BP del machete) y lo conecta con el nodo *Attach Actor to Component*, encargado de unir el machete con el personaje, asignando el nombre exacto del *socket* creado en la mano.

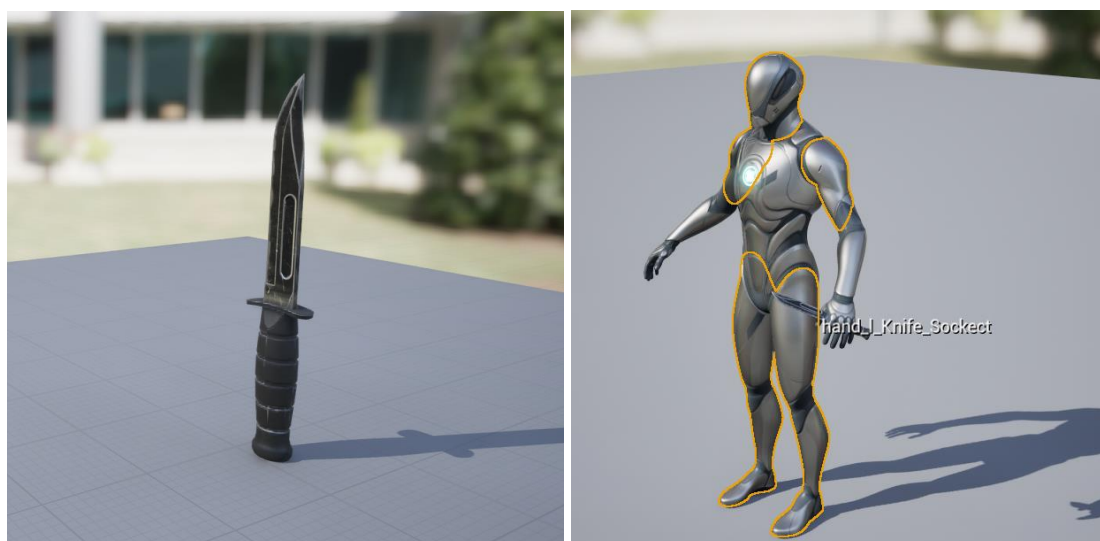


Figura 3-14 Machete que lleva equipado el enemigo en su *socket* (Autoría propia).

Una vez que el machete ha sido unido al *BP_EnemyBase*, se ha creado una *Task* llamada *BTT_EquipWeapon*, encargada de asignar el arma al enemigo desde el BT. Dentro de esta, se ha añadido un *Decorator*, llamado *BTD_IsWildingKife*. En este caso, el *Decorator* verifica si el enemigo ya tiene el arma equipada y, en caso de no tenerla, permite que la *Task* se ejecute para equiparlo automáticamente. El BT, con todas las funciones previamente implementadas, quedaría estructurado como se muestra en la Figura 3-15:

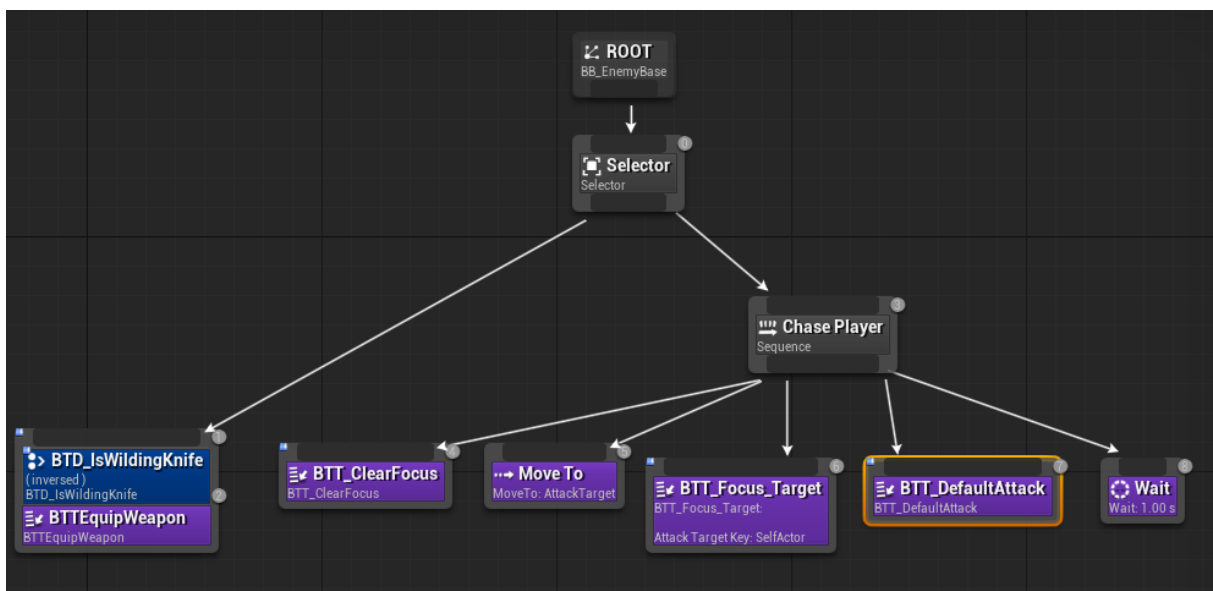


Figura 3-15 BT del *BP_EnemyBase* añadiendo nuevos nodos (Autoría propia).

3.3.3 Patrullas y estados del enemigo

Para crear un enemigo que patrulle por una ruta determinada, lo primero es establecer la patrulla por la que se desplazará. Para ello, se ha creado un BP de tipo *Actor* llamado *BP_PatrolRoute*, al cual se le ha añadido un componente *Spline*, que define el recorrido por donde el enemigo se moverá. Dentro del *Event Graph* de *BP_PatrolRoute*, se ha creado un *Custom Event*, llamado *IncrementPatrolRoute*, cuya función es determinar la posición actual del enemigo, identificar el siguiente punto de la ruta y actualizar su movimiento. Para ello, se han añadido dos variables: *PatrolIndex* y *Direction*, ambas de tipo *Integer*. La variable *Direction* ha sido limitada a los valores 1 y -1, donde 1 indica que el enemigo avanzará en la patrulla y -1 indica que retrocederá.

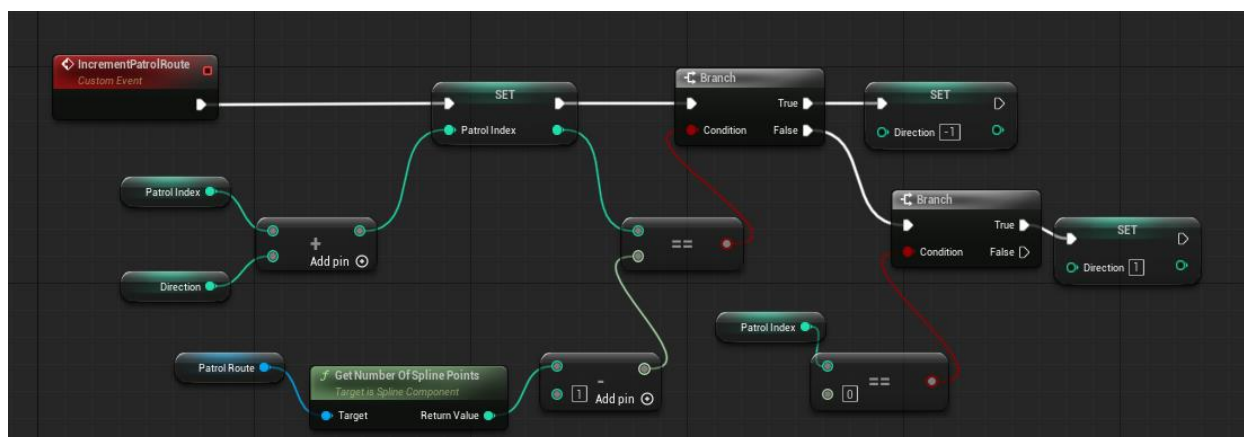


Figura 3-16 Lógica de nodos usados para que el enemigo patrulle en bucle (Autoría propia).

Para gestionar esta lógica (Figura 3-16), se ha implementado un nodo de suma que combina *PatrolIndex* y *Direction*, asegurando que el enemigo avance o retroceda en la patrulla. Posteriormente,

se ha verificado si *PatrolIndex* ha alcanzado el límite de puntos dentro del *Spline*. Para esto, se usa el nodo *Get Number of Spline Points*, que obtiene la cantidad total de puntos en la ruta. Si el *PatrolIndex* iguala este valor, se invierte la dirección asignando -1 a la variable *Direction*, haciendo que el enemigo regrese sobre la misma ruta. Por otro lado, si el índice llega a 0, se cambia la dirección nuevamente a 1, permitiendo que el enemigo reanude su camino en la dirección original.

Con esta lógica, se ha definido una patrulla con puntos y una dirección de movimiento para el enemigo. Sin embargo, aún no se tiene la ubicación exacta de estos puntos en el mundo. Para solucionarlo, se ha creado una función llamada *GetSplinePointsWorldLocation*. Esta utiliza el nodo *Get Location At Spline Point*, que viene por defecto en UE, tomando como referencia las variables *PatrolRoute* y *PatrolIndex* para obtener la posición en el espacio de cada punto de la patrulla. Para asignar la patrulla al enemigo desde el mundo, solo ha hecho falta crear la variable de la patrulla de tipo *BP_PatrolRoute* en el *BP_EnemyBase*, poner la variable como *Instance Editable* haciendo posible su implementación en el mundo. En la Figura 3-17 se puede observar la ruta y al enemigo que tiene estipulado usarla. Para actualizar el BT y hacer que el enemigo siga la ruta de patrulla, ha sido necesario crear una *Task* llamada *BTT_MoveAlongPatrolRoute*. Una vez llegado a este punto, se ha creado un BP de tipo *Interface* llamado *BPI_EnemyAI* (Figura 3-17).

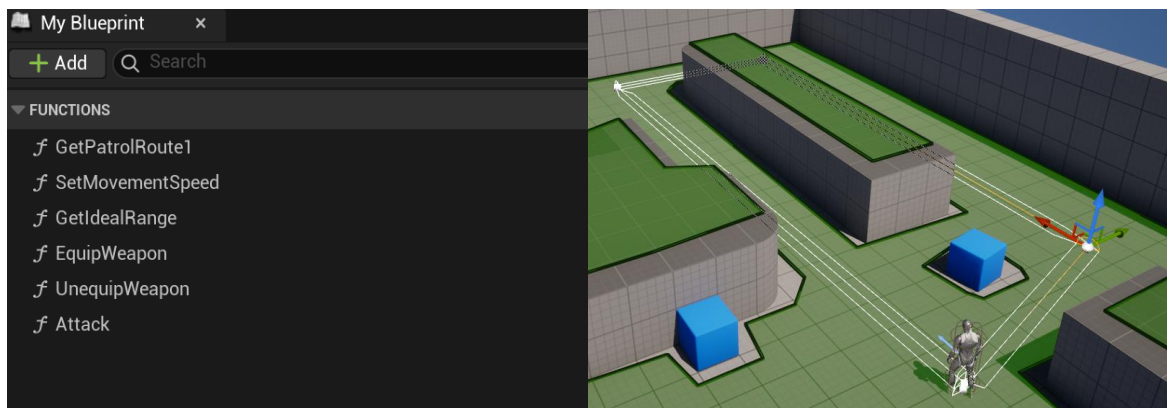


Figura 3-17 *BPI_EnemyAI* y representación visual de la *BP_PatrolRoute* (Autoría propia).

Desde *Class Settings*, se ha integrado esta interfaz en el *BP_EnemyBase*, permitiendo que pueda acceder a sus funciones. La primera función añadida ha sido *GetPatrolRoute*. La *task BTT_MoveAlongPatrolRoute*, primero verifica si la ruta de patrulla es válida. Si lo es, obtiene la ubicación del primer punto de la *spline* usando *GetSplinePointasWorldPosition*. Luego, el enemigo se mueve a esa ubicación mediante el nodo *AI Move To*. Una vez llega al punto, se ejecuta el evento *IncrementPatrolRoute* para avanzar al siguiente punto de la patrulla. Así se construye la lógica que permite al enemigo moverse al primer punto de la ruta y comenzar su patrullaje.

El enemigo, en este punto, se desplaza esprintando entre los puntos de patrulla, cuando lo ideal sería que caminara, haciendo que la patrulla fuese más realista. Para solucionar esto, se ha añadido una nueva función en el *BPI_EnemyAI* (Figura 3-17) llamada *SetMovementSpeed*, cuya finalidad es ajustar la velocidad del enemigo según sea necesario.

Dado que el enemigo puede adoptar diferentes velocidades y estados de movimiento, se ha creado un *Enum*, que es una enumeración de elementos que permite definir distintos valores preestablecidos. En este caso, se han añadido cuatro estados de movimiento: *Idle* (reposo, es decir, estático en el sitio), *Walking* (caminando), *Jogging* (trotando) y *Sprinting* (corriendo).

Una vez definida la función, es necesario establecer su lógica y determinar qué efecto tendrá en el personaje. En la Figura 3-18 se muestran los valores asignados a cada velocidad, de modo que, al crear una *Task* en el BT y llamar a una variable de tipo *Enum* que haga referencia al previamente creado, esta se configure como *Instance Editable*, permitiendo seleccionar la velocidad que se desea establecer desde el BT. Esta *Task*, llamada *BTT_SetMovementSpeed*, permite ajustar la velocidad del personaje antes de

cada movimiento, asegurando un desplazamiento natural y acorde con la situación. Para este escenario, en el que hay más de un enemigo y se busca que algunos patrullen y otros realicen otras acciones, se ha colocado un *decorator*, llamado *BTD_HasPatrolRoute*, que valida si tiene o no patrulla, para en caso de no tenerla no ejecutar la secuencia.

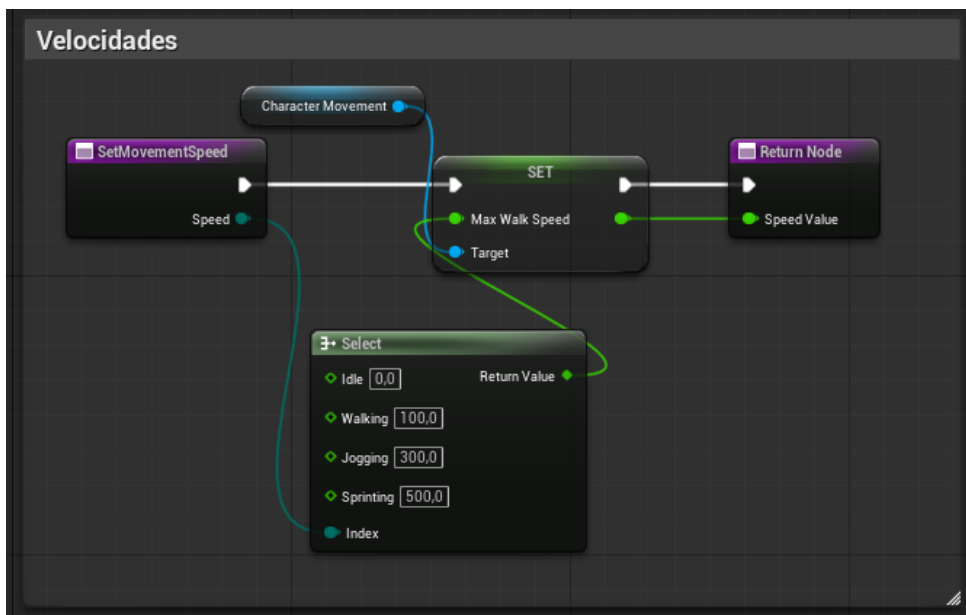


Figura 3-18 Establecimiento de valores de velocidad en *BP_EnergyBase* (Autoría propia).

Se han implementado ya varias funciones en *BP_EnergyBase*, pero actualmente el enemigo no tiene estados definidos, lo que significa que no distingue en función de su estado qué debe hacer. Para solucionar esto, al igual que con las velocidades, se ha creado un *Enum*, llamado *E_States*, para los diferentes estados del enemigo, permitiéndole reaccionar de manera adecuada según su estado. Este define distintos comportamientos que el enemigo puede adoptar en función del contexto.

Los estados asignados en *E_States* (Figura 3-19) son los siguientes: *Passive*, donde el enemigo no realiza ninguna acción; *Attacking*, cuando está en combate; *Frozen*, que se activará cuando reciba un golpe y deba ejecutar una animación de reacción, evitando problemas visuales debidos a incongruencias entre la velocidad y la animación; *Investigating*, cuando el enemigo escucha un sonido y va a la última posición en la que vio al *BP_Player*; *Dead*, cuando el enemigo muera; y *Seeking*, cuando pierda de vista al jugador y quiera buscar en las zonas más probables donde este podría estar.

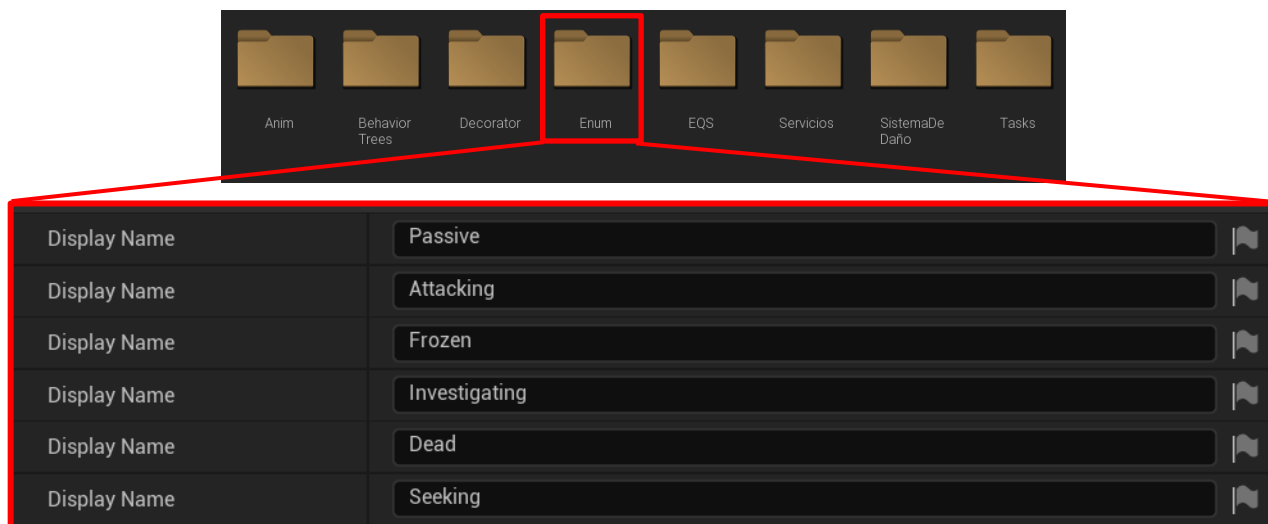


Figura 3-19 *Enum* que contiene los estados comunes de enemigos (Autoría propia).

Para que estos estados sean utilizables dentro del BT, es necesario crear una variable en *BB_EnergyBase* que permita identificar en qué estado se encuentra el enemigo en cada momento. En la Figura 3-9 se muestra esta variable, denominada *States*, de tipo *Enum*, la cual hace referencia a la lista de estados definidos en *E_States*.

Del mismo modo que se le ha asignado el objetivo a la variable *AttackTarget* en la Figura 3-10, es imprescindible que la BB registre los estados del enemigo y los interprete correctamente. Para ello, se ha seguido el mismo procedimiento. Para que el BT ejecute una acción u otra según el estado del enemigo, es suficiente con añadir *Decorators* de tipo *Blackboard* a los *Selectors* dentro del BT. Como se muestra en la Figura 3-20, estos *Decorators* permiten diferenciar cuándo el enemigo está en estado *Attacking* y cuándo está en estado *Passive*, asegurando que solo ejecute las acciones correspondientes a cada estado.

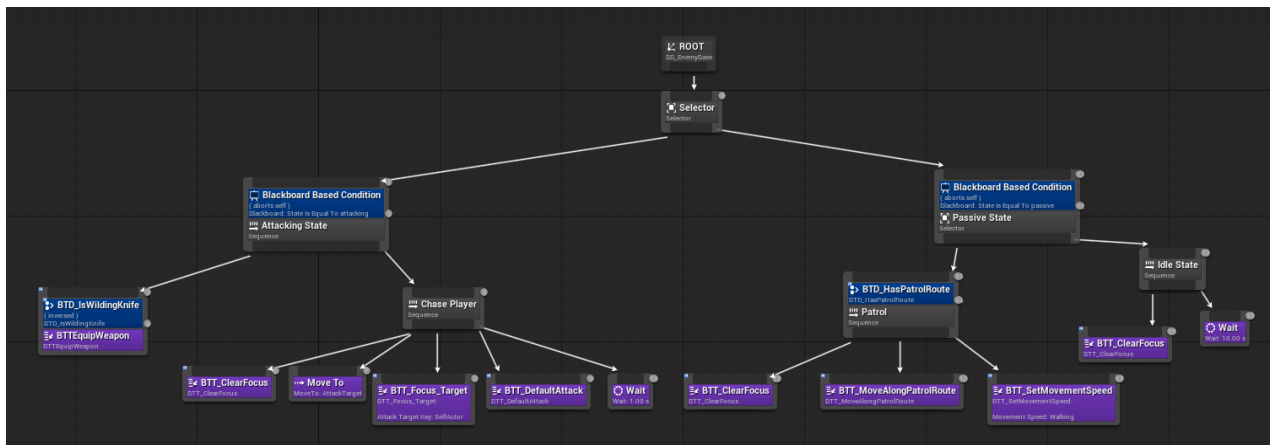


Figura 3-20 BT con *AttackingState* y *PassiveState* (Autoría propia).

La Figura 3-20 muestra lo que actualmente conforma el BT, compuesto por sus dos estados: *AttackingState* y *PassiveState*. Estos llevan sus respectivos *decorators*, y su lógica por debajo ya explicada a lo largo del desarrollo. Para permitir que se pueda hacer referencia a estos estados, se han creado funciones dentro de *AIC_EnergyBase*, de modo que cualquier enemigo que utilice este *AIController* pueda acceder a ellas. Para cada estado se ha implementado una función con el nombre *Set State As...* (por ejemplo, *SetStateAsPassive*), lo que facilita la asignación de estados específicos al enemigo en futura programación. La lógica detallada de estas funciones se muestra en la Figura 3-21.

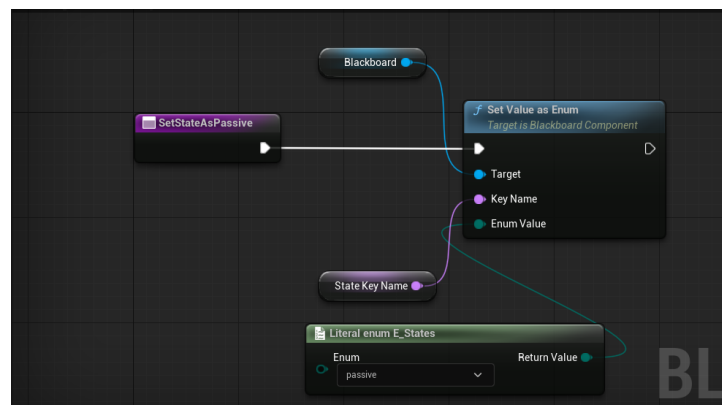


Figura 3-21 Implementación de la función *SetStateAsPassive* (Autoría propia).

3.3.4 Introducción a la percepción del entorno del enemigo

Es fundamental integrar un sistema de percepción en el enemigo para dotarlo de realismo, ya que no es lo mismo un NPC que no pueda ver, sentir o escuchar, que uno capaz de percibir su entorno a través de los sentidos. Este apartado se va a centrar en explicar el proceso seguido para incorporar el

AI Perception al enemigo, permitiéndole reaccionar a distintos estímulos y ejecutar acciones en función de ellos. Lo primero que se ha realizado es agregar un *AI Perception* como componente en *AIC_EnergyBase*. En la sección de *Details*, se han añadido tres *Array Elements*, uno para cada sentido: oír, ver y recibir daño. Posteriormente, se han configurado sus parámetros como en la Figura 3-22:

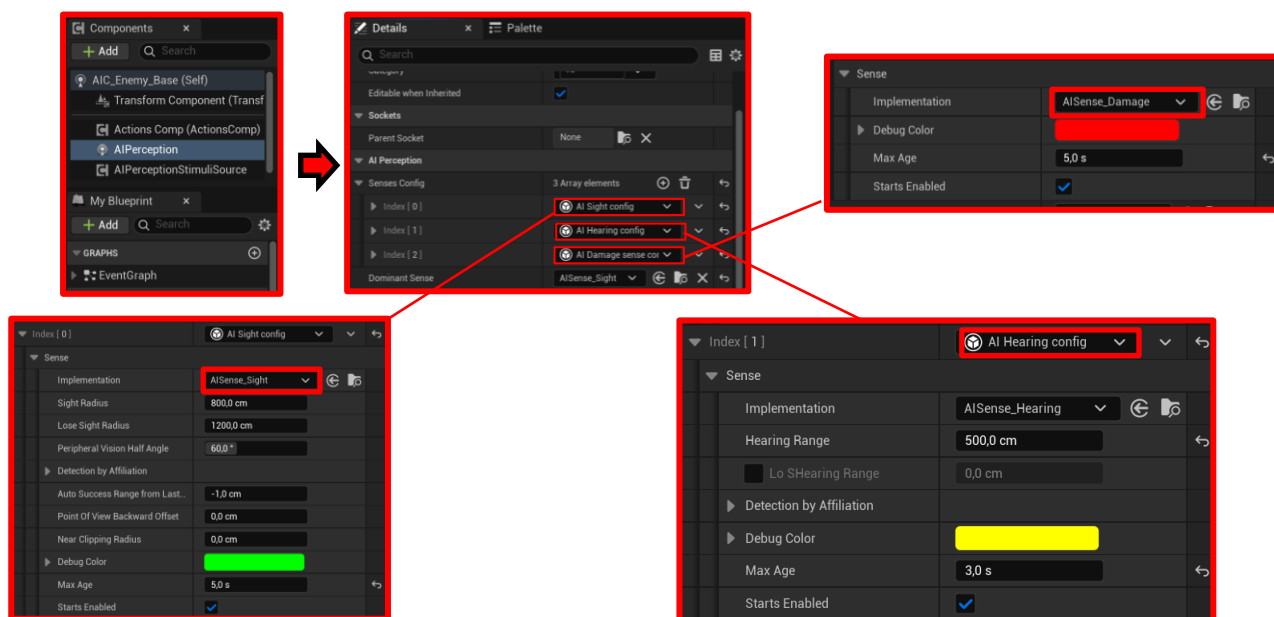


Figura 3-22 Implementación del *AI Perception* en el *AIC_EnergyBase* (Autoría propia)

La percepción visual permite al enemigo detectar objetos o personajes dentro de un rango de 8000 cm. Si un objetivo sale de este radio, aún será detectado hasta los 12000 cm, momento en el que el enemigo perderá su rastro. Además, el ángulo de visión periférica es de 60°, lo que significa que solo podrá ver lo que esté dentro de ese campo de visión. La percepción auditiva permite al enemigo escuchar sonidos en un radio de 500 cm. Esto significa que cualquier sonido generado dentro de esta distancia activará su detección y podrá reaccionar en consecuencia. Además, el sistema mantiene la información del sonido durante 3 segundos, permitiendo que el enemigo pueda actuar incluso si el sonido desaparece rápidamente. La percepción del daño hace que el enemigo detecte impactos o golpes recibidos. Este estado de alerta (*age*) se mantiene durante 5 segundos, tiempo en el que el enemigo seguirá reaccionando al daño antes de que la percepción de este evento desaparezca. Estos parámetros pueden ajustarse según el tamaño del escenario y los requerimientos específicos de la simulación, permitiendo adaptar la percepción del enemigo para que se alinee con el diseño y tenga sentido dentro del juego.

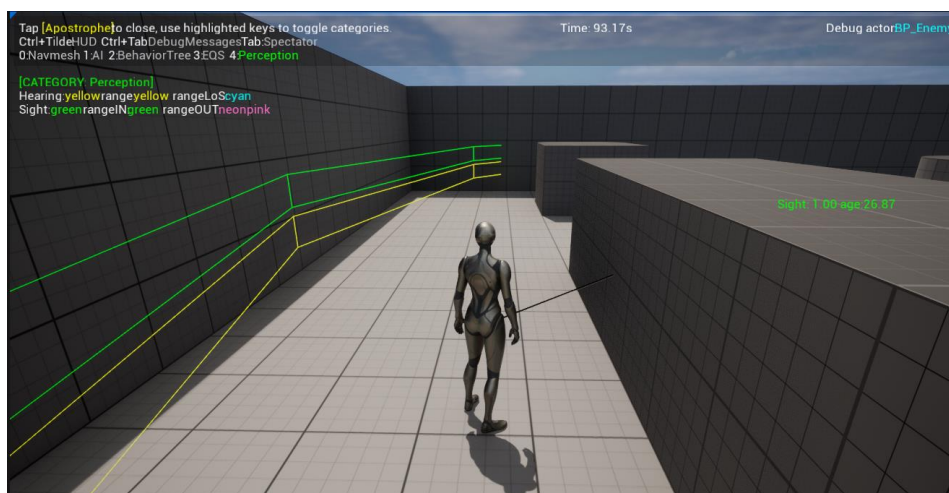


Figura 3-23 Visualización de los radios de los sentidos del enemigo [53].

En la Figura 3-23 se pueden observar líneas verdes que representan el rango de visión del NPC, indicando el área en la que puede detectar visualmente a otros actores. Las líneas amarillas corresponden al rango de audición, mostrando hasta dónde el enemigo puede escuchar sonidos generados en el entorno.

Una vez establecidos los parámetros de percepción en el enemigo, se ha implementado una lógica que haga que cuando el enemigo perciba alguno de los estímulos haga lo que se le programe, para darle realmente utilidad. En primer lugar, se ha creado la función *CanSenseActor* en el *AIC_EnemyBase*. A su vez, se ha creado otro *Enum*, llamado *E_Senses*, que plasma los posibles estímulos que el enemigo puede sentir (*Hearing*, *Sight*, *Damage*). La estructura de nodos de la función *CanSenseActor* se muestra en las Figura 3-24 y Figura 3-25, que procesan la percepción del enemigo, permitiéndole detectar los estímulos de otros actores en el entorno.

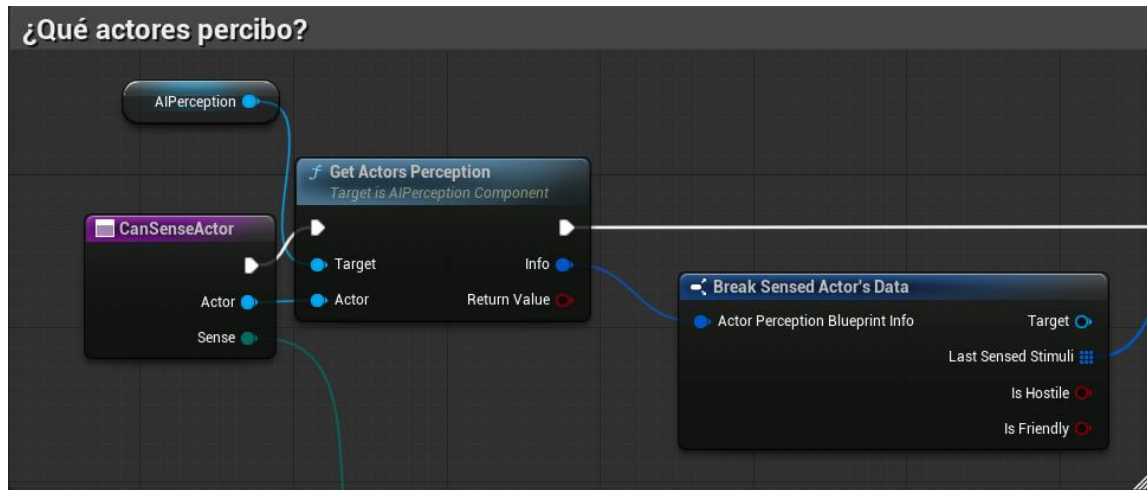


Figura 3-24 Primera parte de la lógica de la función *CanSenseActor* (Autoría propia).

La función recibe dos *inputs*: *Actor*, que detecta el actor que se quiere evaluar, y el tipo de percepción *Sense* (que se observan en el primer nodo de la Figura 3-24) que hace referencia a *E_Senses*. El proceso funciona de la siguiente manera: para cada actor del que se ha detectado un estímulo, se realiza un *Break* para identificar cuál ha sido el último estímulo recibido. Luego, con *Get Sense Class for Stimulus*, se ha obtenido la clase del estímulo y se ha comparado con los tres sentidos definidos en el *Enum*, utilizando un *Select* para determinar su categoría. Si la clase obtenida coincide con el sentido evaluado, el nodo *Branch* devuelve *True*, lo que permite que el *Return Node* confirme que el estímulo ha sido detectado con éxito, utilizando la información del *Successfully Sensed* del *Break AIStimulus*.

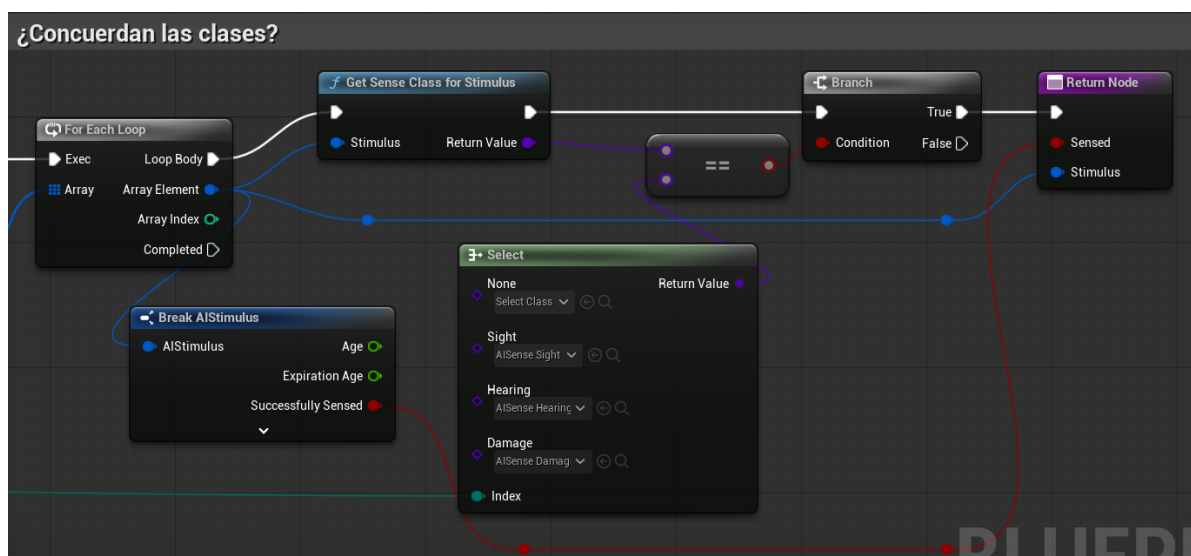


Figura 3-25 Segunda parte de la lógica de la función *CanSenseActor* (Autoría propia).

Dentro del *AIC_EnergyBase*, se ha programado la respuesta del enemigo cuando detecta con éxito alguno de los estímulos y determina qué debe hacer en cada caso. Para ello, se ha utilizado el nodo *On Perception Updated* de *AI Perception*, el cual se actualiza constantemente. Cuando este evento se activa, el *For Each Loop* revisa los actores detectados y el *Sequence* divide la ejecución en tres partes, analizando la visión, el sonido y el daño. Para cada sentido, la función *CanSenseActor* (Figura 3-26) verifica si el enemigo ha detectado algo y, si la condición se cumple, el nodo *Branch* que lo continua activa la función correspondiente. *HandleSenseSight* (Figura 3-27) gestiona la detección visual y la reacción del enemigo, *HandleSenseSound* (Figura 3-29) procesa la ubicación del sonido percibido con *Break AI Stimulus* y ajusta su comportamiento, mientras que *HandleSenseDamage* permite que el enemigo responda adecuadamente cuando recibe un ataque.

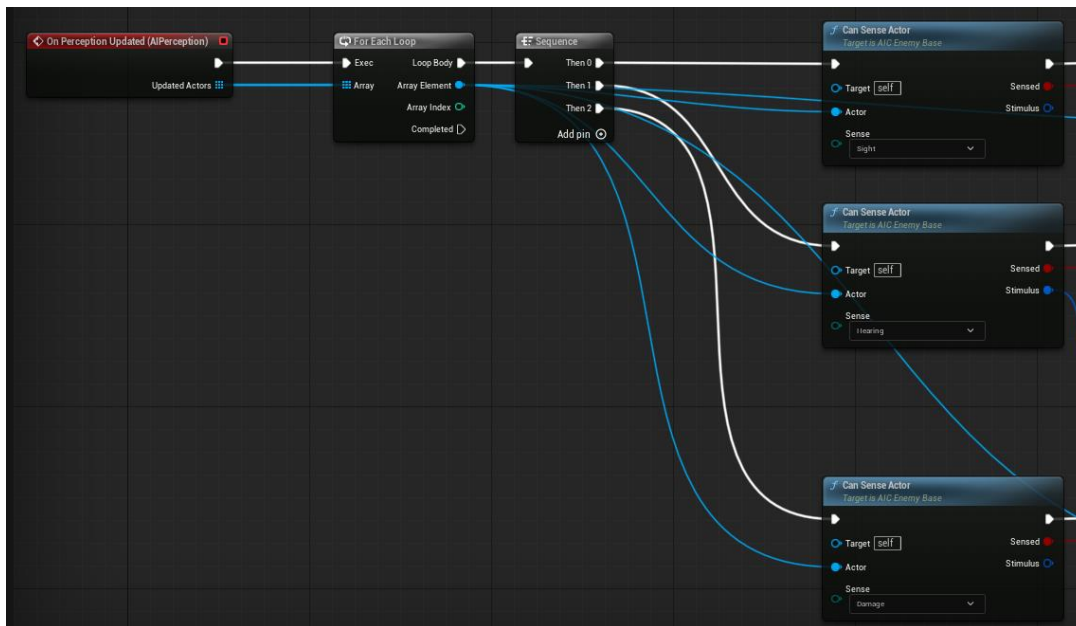


Figura 3-26 lógica inicial común para asignar funciones a los estímulos (Autoría propia).

En el caso de *HandleSenseSight*, como se puede observar en la Figura 3-27, si el enemigo percibe al *BP_Player*, primero obtiene su estado actual y luego utiliza un *Switch* en *E_States*. Si el enemigo se encuentra en estado *Investigating* o *Passive*, cambia automáticamente su estado a *Attacking* para iniciar la persecución.

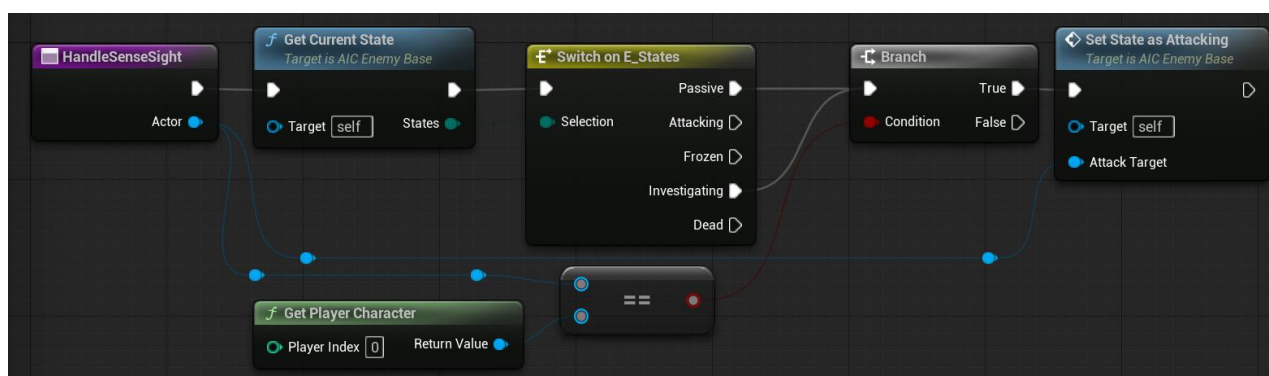


Figura 3-27 Función *HandleSenseSight* (Autoría propia).

La función *Get Current State* que se ve en la Figura 3-27, cumple la misma función que las de tipo “*Set State As*” mencionadas con anterioridad. La lógica de la función expuesta en la Figura 3-28 busca obtener el estado actual del enemigo desde la BB utilizando *Set Value As Enum* para acceder a *E_States*. Luego, almacena el valor de la variable *State* del *BB_EnergyBase* y lo entrega al *Return Node*,

permitiendo que otras funciones puedan conocer el estado en el que se encuentra el enemigo en cada momento.

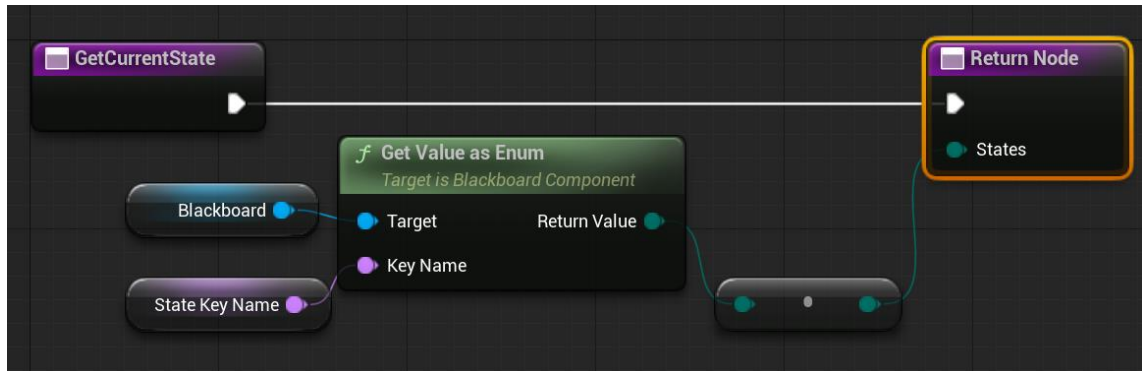


Figura 3-28 Función *GetCurrentState* (Autoría propia).

En la Figura 3-29, se muestra la función *HandleSenseSound*, que sigue el mismo proceso que *HandleSenseSight*. Desde el *Event Graph* de *AIC_EnemyBase*, la función *CanSenseActor* utiliza un *Branch* para comprobar si el enemigo ha detectado un sonido con éxito. A través de *Break AIStimulus*, se obtiene la localización del estímulo percibido y, como respuesta, el enemigo cambia su estado a *Investigating*, permitiéndole dirigirse al punto donde se originó el sonido.

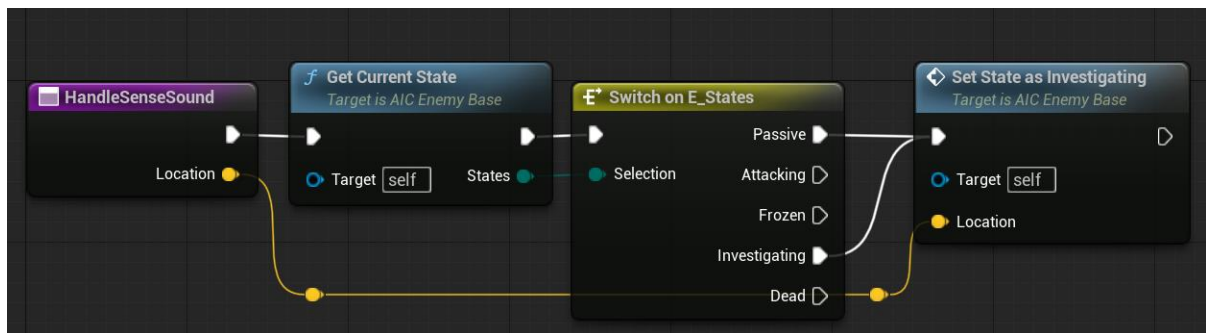


Figura 3-29 Función *HandleSenseSound* (Autoría propia).

Al igual que se ha hecho previamente con *Set StateAsPassive* y *SetStateAsAttacking*, es necesario crear *SetStateAsInvestigating* visible en la Figura 3-29. La lógica sigue el mismo principio que en la Figura 3-28, donde se asigna un valor a la variable del BB para que el enemigo actúe según lo esperado, en este caso, moviéndose a la ubicación del estímulo de sonido. Para ello, se ha accedido nuevamente a la BB y se ha creado la variable *PointOfInterest*, una variable clave que será utilizada frecuentemente. La función *SetStateAsInvestigating* (Figura 3-30) primero establece el nuevo estado *Investigating* dentro de *E_States* y luego asigna el valor correspondiente a *PointOfInterest*, permitiendo que se guarde la localización donde se detectó el sonido.

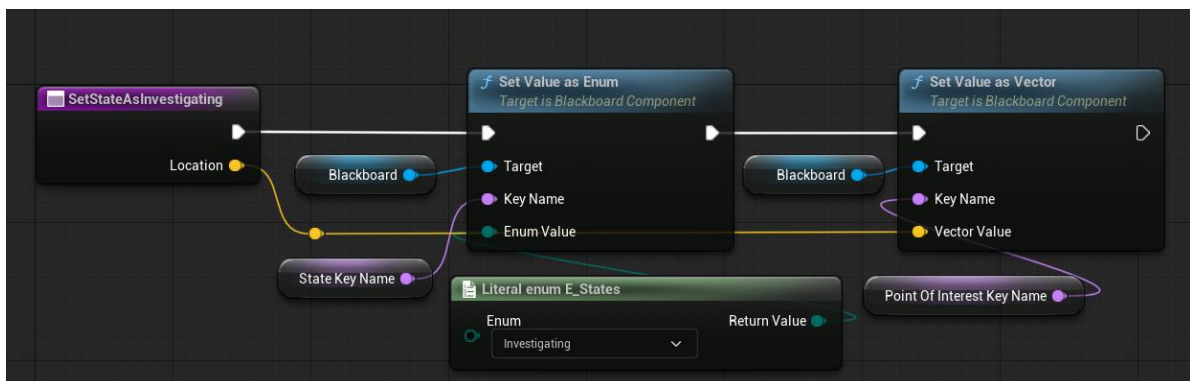


Figura 3-30 Función *SetStateAsInvestigating* (Autoría propia).

Para que el enemigo se dirija al *PointOfInterest*, ha sido necesario introducir esta funcionalidad dentro del *BT_EnemyBase*. La estructura de la ramificación sigue un enfoque similar al utilizado para los estados *Attacking* y *Passive*. Se ha establecido una secuencia con un *Decorator* de tipo BB. Dentro de esta secuencia, llamada *InvestigatingState*, el enemigo primero ajusta su velocidad a *Jogging* con el nodo *BTT_SetMovementSpeed*, luego ejecuta un *Move To* dirigido al *PointOfInterest* y, finalmente, espera un tiempo determinado. El objetivo es que el enemigo se desplace hasta la ubicación donde se detectó el estímulo y, si encuentra al jugador, proceda a atacarlo. Sin embargo, si al llegar no detecta al jugador, lo ideal es que vuelva a estado *Passive*, permaneciendo en su lugar en *Idle* o regresando a patrullar. Para lograr esta transición, se ha necesitado una *Task* llamada *BTT_SetStateAsPassive*, cuya lógica consiste en hacer un *Cast* al *AIC_EnemyBase* y llamar a la función *SetStateAsPassive*. Esta *Task* se ha colocado inmediatamente después del nodo *Wait*, garantizando que, si el enemigo no encuentra al jugador tras esperar, vuelva automáticamente a su estado *Passive* como se muestra en la Figura 3-31. Dentro de la secuencia se observa el comando *Abort Self*, que significa que, si el estado del enemigo cambia, por ejemplo, mientras se mueve o espera dentro de la secuencia *InvestigatingState*, la secuencia se aborta y el enemigo ejecuta las acciones del nuevo estado.

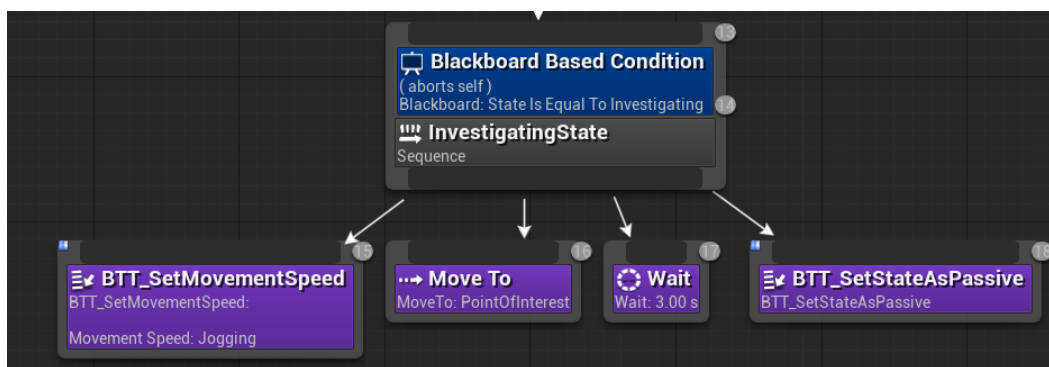


Figura 3-31 Lógica del estado *Investigating* implementada en *BT_EnemyBase* (Autoría propia).

La última función por implementar es *HandleSenseDamage*, cuya lógica es prácticamente idéntica a la de *HandleSenseSound*. La diferencia principal es que, en lugar de utilizar la ubicación del estímulo como *input* (*Location* en la Figura 3-30) utiliza *Actor* referenciado al *AttackTarget*. Esto permite que, si el enemigo recibe daño, cambie su estado y ataque de inmediato al *BP_Player*.

3.3.5 Introducción al Environment Query Sistem

El *Environment Query System* (EQS) es una herramienta dentro del sistema IA, que recopila datos del entorno y los procesa mediante diferentes *Tests* para determinar el mejor resultado según ciertas condiciones. Los *EQS* pueden integrarse en un BT para ayudar a la IA en la toma de decisiones, analizando ubicaciones o actores. Esto permite que los personajes controlados por IA encuentren las mejores posiciones para atacar, buscar cobertura o localizar recursos o actores.

En este apartado se abordará principalmente el uso de *EQS*, aunque también se implementarán otras mecánicas complementarias para mejorar el realismo del enemigo. Actualmente, el *BP_EnemyBase*, al atacar al *BP_Player*, se queda en su posición golpeándolo de manera constante. Para brindar mayor libertad de toma de decisiones al líder del equipo y permitirle evaluar mejor la situación antes de decidir su siguiente movimiento, se ha modificado el sistema de ataque. Ahora, el enemigo empleará una mecánica conocida como *strafing*, que consiste en moverse alrededor del *BP_Player*, desplazándose a puntos estratégicos en forma de círculo a una distancia determinada en cada uno de los ataques.

Para que el *BP_EnemyBase* bordeé al *BP_Player* de manera realista, es necesario un conjunto de animaciones adecuadas. Las animaciones por defecto de UE no permiten este tipo de movimientos, donde el enemigo se mueve agachado, con las rodillas flexionadas y los brazos en una posición más ofensiva. Para ello, se ha realizado *retargeting* de las animaciones necesarias utilizando el paquete

AnimStarterPack. En concreto, se ha creado un *Blend Space*, llamado *BS_Strafing*, que es una animación que integra varias animaciones dentro de una misma estructura.

Dentro del *BS_Strafing*, se han introducido las distintas animaciones necesarias. Como se puede observar en la Figura 3-32, hay una gráfica con dos ejes: *Speed* y *Direction*. La variable *Speed* varía entre 0 y 100, sin cambios intermedios de velocidad (aunque se podría diseñar un *Blend Space* que pasara de estar estático a caminar y luego a correr), por lo que solo tiene dos estados: quedarse en estado *Idle* o establecer la velocidad de la animación. Sin embargo, la variable *Direction* es especialmente relevante, ya que permite definir las animaciones de desplazamiento hacia adelante (*forward*), atrás (*backwards*), derecha (*right*) e izquierda (*left*). En la Figura 3-32 se muestra cómo se distribuyen estas animaciones dentro de la gráfica y qué sentido de movimiento cumple cada punto.

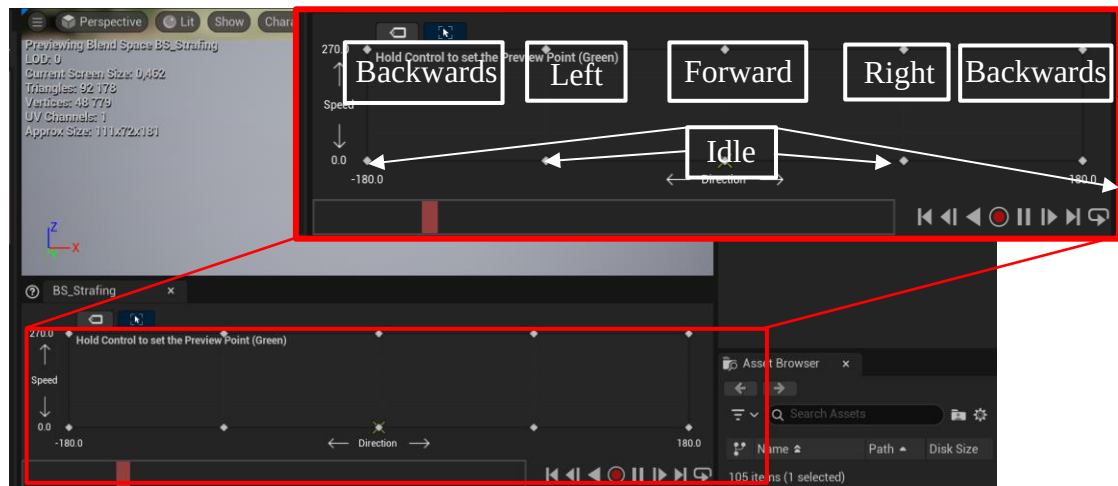


Figura 3-32 *BS_Strafing* (Autoría propia).

En *ABP_Manny*, específicamente en el *Event Graph*, se han realizado modificaciones para determinar la dirección del movimiento y así indicar a *BS_Strafing* qué animación debe utilizar en función del desplazamiento del enemigo. La dirección se obtiene a través de la lógica expuesta en la Figura 3-33, donde se emplea *Calculate Direction* para calcular la orientación del movimiento en función de la variable velocidad (*Velocity*) y la rotación del actor que utiliza el ABP.

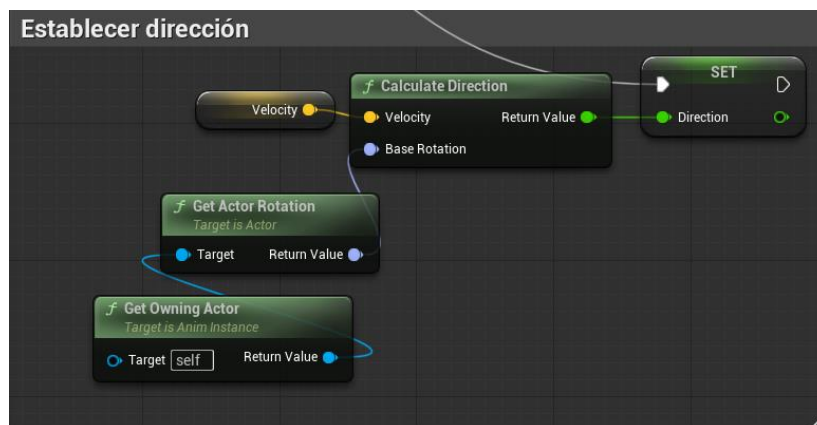


Figura 3-33 Establecimiento de la variable *Direction* (Autoría propia).

Lo que se busca es que la animación *Strafe* no se utilice en todo momento, sino que el enemigo use la animación predeterminada de *Walk/Run* de UE cuando se desplace corriendo hacia el *BP_Player* o este patrullando. Para lograr esto, es necesario alternar entre diferentes *Blend Spaces*, dependiendo de si el enemigo está mirando directamente al objetivo o simplemente patrullando por el escenario. Para ello, se ha creado la variable *Is Focusing Target*, la cual establece la lógica mostrada en la Figura 3-34. Esta lógica obtiene al actor que usa la animación, utiliza su *AIC_Controller* (siempre será *AIC_EnemyBase*)

y verifica si el enemigo tiene un objetivo a la vista válido. En caso afirmativo, la variable *Is Focusing Target* se activa.

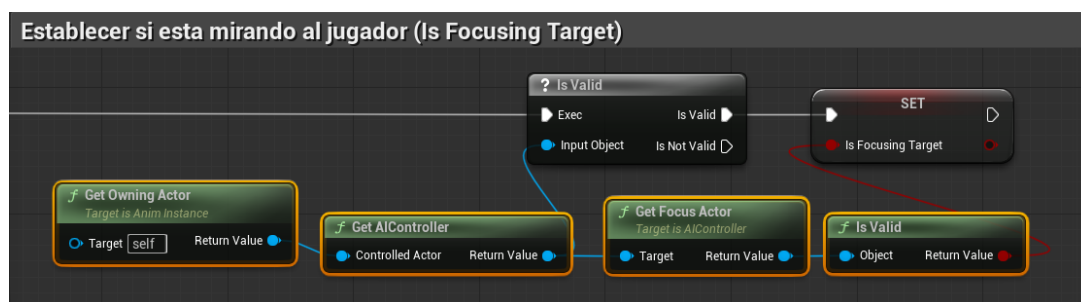


Figura 3-34 Establecimiento de la variable *Is Focusing Target* (Autoría propia).

Dentro del *AnimGraph* del ABP, específicamente en el nodo *Locomotion*, se han realizado modificaciones en la lógica del estado *Idle*. Anteriormente, este ejecutaba únicamente la animación predeterminada de *Idle* de UE, pero se ha sustituido por el nodo *Blend Poses By Bool* (Figura 3-35), el cual permite seleccionar entre dos animaciones en función de la condición *Is Focusing Target*, de manera que si es está activada, se ejecuta la animación de *Idle Strafing*, y en caso de no estarlo, se mantiene la animación predeterminada.

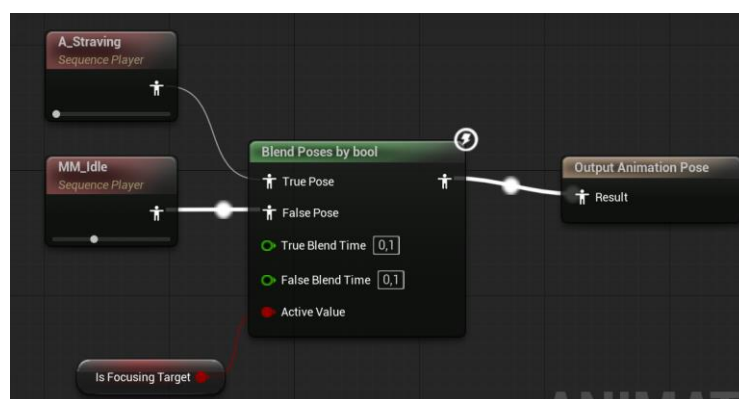


Figura 3-35 Nueva lógica en el estado *Idle* del *Locomotion* (Autoría propia).

Para el estado *Walk/Run*, se ha aplicado el mismo procedimiento, utilizando el nodo *Blend Poses By Bool* con la misma condición. La diferencia radica en que, al tratarse de BS, requieren dos variables de entrada en lugar de una. Como se observa en la Figura 3-36, la animación predeterminada solo necesita la variable *Ground Speed*, mientras que *BS-Strafing* requiere tanto *Direction* como *Ground Speed*.

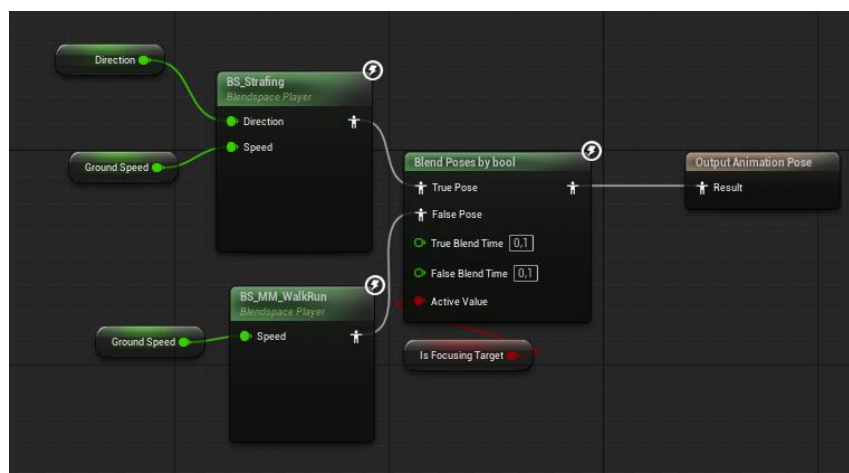


Figura 3-36 Nueva lógica en el estado *Walk/Run* del *Locomotion* (Autoría propia).

Una vez establecido el nuevo sistema de animaciones, se procede a integrar el *EQS*. Lo primero es crearlo, para lo cual se ha generado una carpeta específica donde se almacenarán los diferentes *EQS*. En este caso, se ha denominado *EQS_Strafe*. Al abrir su interfaz, se puede observar en la Figura 3-37 que es similar a la de un BT. Dentro del *Query Graph*, lo primero que se obtiene es un nodo *Root*, que funciona de manera similar al nodo principal en un BT. Apretando el botón izquierdo del ratón y sacado una flecha desde *Root*, se despliegan diversas opciones como *Generators*, los cuales permiten generar los puntos de referencia para la necesidad requerida. Existen varios tipos de *Generators*, como: cono, círculo o donut, entre otros. En este caso, se requiere generar puntos alrededor del *BP_Player*, por lo que se ha utilizado un *Generator* de tipo *Circle* (Figura 3-38). Se ha limitado su radio a 350 unidades y se ha definido un total de 5 puntos de referencia con el objetivo de que el *BP_EnergyBase* se mueva alrededor de estos usando el *BS_Strafing*.

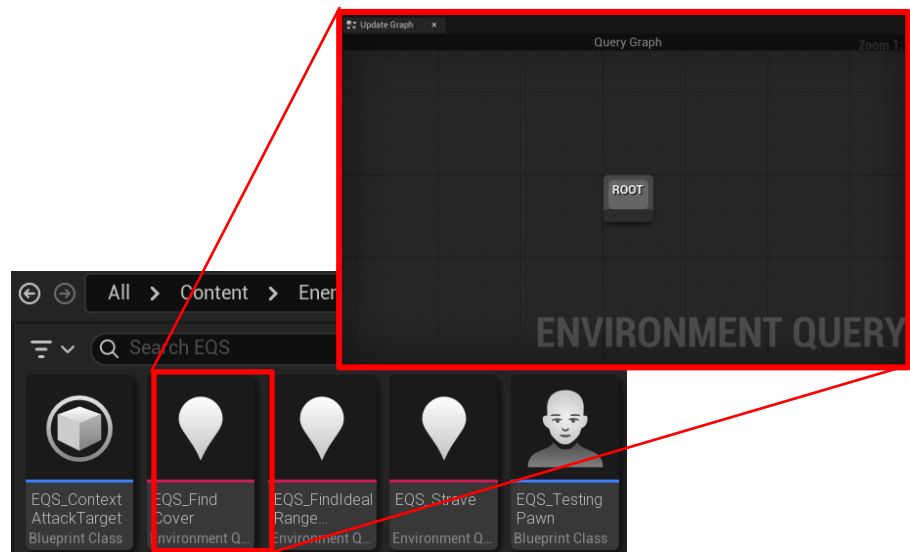


Figura 3-37 Carpeta de EQS y el Query Graph de EQS_Strafe (Autoría propia).

Una vez establecido el tipo de *Generator* necesario, el siguiente paso es centrarlo en el *BP_Player*. Para lograr esto, el enemigo necesita un BP de tipo *Context*, el cual le indicará en qué actor debe estar centrado el EQS. Por esta razón, se ha creado *EQS_Context_AttackTarget*. En la lógica mostrada en la Figura 3-39, se ha añadido una variable llamada *AttackTarget*, la cual proviene de la función *SetStateAsAttacking* dentro de *AIC_EnergyBase*, que se ha modificado añadiendo la variable *AttackTarget* y haciendo *Set AttackTarget* al final de la lógica para establecerla.

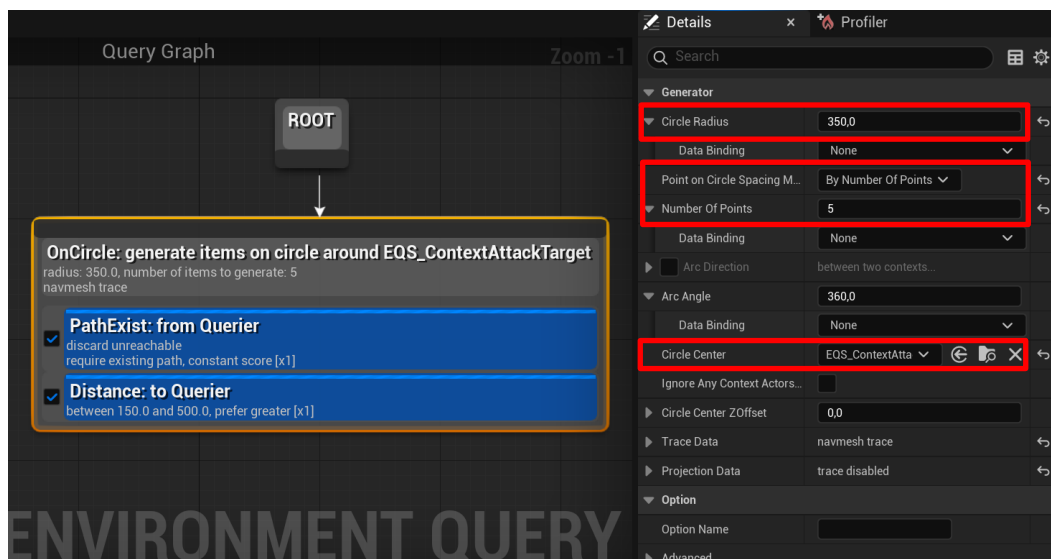


Figura 3-38 EQS_Strafe (Autoría propia).

Una vez definida la variable, la lógica en la Figura 3-39 consiste en obtener el *AIController*, hacer un *cast* a *AIC_EnemyBase* para llamarlo, establecer *AttackTarget* como la variable a utilizar y devolverla al *Return Node*. Finalmente, dentro del *Query Graph*, en la Figura 3-38, se encuentra un apartado denominado *Circle Center*. Aquí, basta con asignar el *Context* creado, lo que permitirá que el *EQS* se centre en el *AttackTarget*.

Como siguiente paso, se ha añadido el *Test Path Exist* para verificar si el punto al que debe desplazarse el enemigo es alcanzable. Esto evita situaciones en las que, por ejemplo, el enemigo intente moverse hacia un punto ubicado en la superficie de un bloque inaccesible, evitando que se quede bloqueado intentando alcanzarlo. Además, para finalizar, se incorpora el *Test Distance*, el cual mide la distancia con respecto al *Querier* (en este caso, el *AttackTarget*). Esto asegura que el enemigo solo se mueva dentro del perímetro del círculo sin atravesarlo ni acercarse demasiado, evitando que seleccione puntos demasiado lejanos afectando a la coherencia del movimiento.

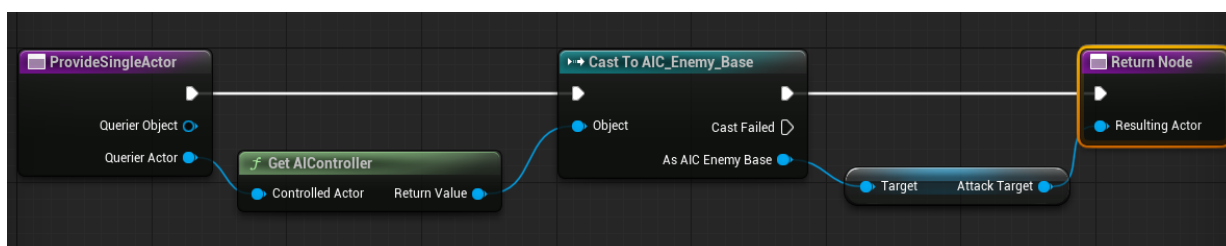


Figura 3-39 *EQS_Context_AttackTarget* (Autoría propia).

Como cualquier lógica, esta debe integrarse en el BT para que funcione (Figura 3-40). Se ha creado una secuencia llamada *Strafe*, donde el enemigo primero mira al objetivo, cambia su velocidad a *Walking*, ejecuta el *EQS_Strafe*, se mueve al punto calculado y luego espera un tiempo aleatorio para evitar movimientos predecibles. Inicialmente, el enemigo atacaba al detectar al jugador sin ejecutar la secuencia *Strafing*. Para solucionarlo, se ha añadido un *Cooldown* de 15 segundos tras cada ataque, forzando al enemigo a hacer *Strafing* antes de volver a atacar. El *Cooldown* está configurado para abortar acciones de menor prioridad (como patrullar), asegurando que el enemigo reaccione al jugador de inmediato. Además, si el jugador se aleja demasiado durante el *Strafing*, este se interrumpe y el enemigo ajusta su posición antes de retomar el movimiento lateral.

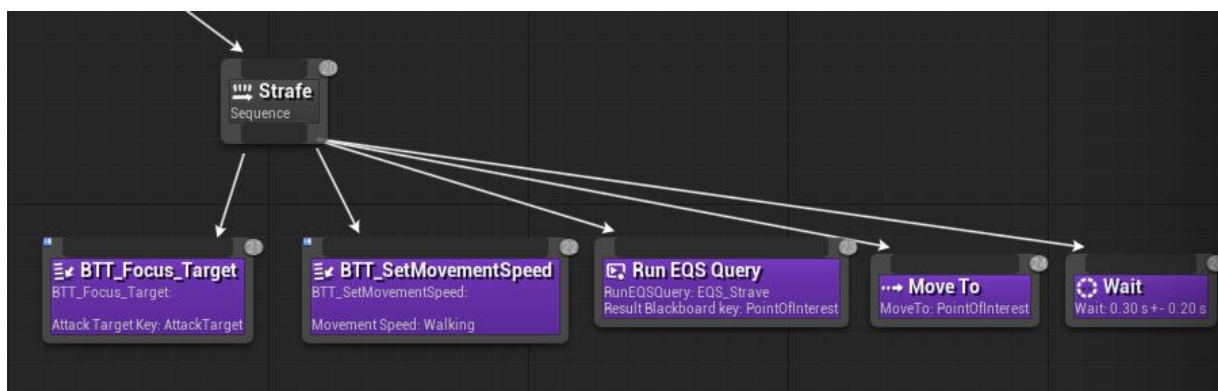


Figura 3-40 Ramificación del *Behavior Tree* de la secuencia *Strafe* (Autoría propia).

Cuando el enemigo se encuentra lejos del *BP_Player*, surgen discordancias entre las secuencias *Attack* y *Strafe*. Si detecta al jugador desde una distancia considerable, puede iniciar *Strafing* mientras aún está demasiado lejos, lo que genera un comportamiento poco realista. En lugar de acercarse primero en modo *Sprinting* o *Jogging*, lo hace en estado *Walking*. Para solucionar esto, se debe establecer una lógica que determine qué acción debe realizar el enemigo en función de la distancia a la que se encuentre. Si el enemigo está dentro de un rango cercano, puede ejecutar *Strafing*. Sin embargo, si se encuentra a mayor distancia, su prioridad debe ser acercarse primero al *BP_Player* en estado *Attack* antes de iniciar

Strafing. Esto evitará inconsistencias entre las secuencias y garantizará que el enemigo actúe de manera más natural y estratégica en el combate.

Lo primero es añadir dos nuevas variables a la BB (Figura 3-9): *AttackRadius* y *DefendRadius*. Posteriormente, se ha añadido una nueva función en el *BPI_EnergyAI*, llamada *GetIdealRange* (Figura 3-17) con dos *inputs*, llamados *AttackRadius* y *DefendRadius*, de tipo *float*. Desde ahí, abriendo la función se introducen los valores de las dos variables que, para el caso del mapa, son 100 para el *AttackRadius* y 350 para el *DefendRadius*. Como se hizo con *AttackTarget* en la Figura 3-10, desde el *AIC_EnergyBase* se asignan los valores a la BB de las variables, quedando la lógica como en la Figura 3-41 a continuación:

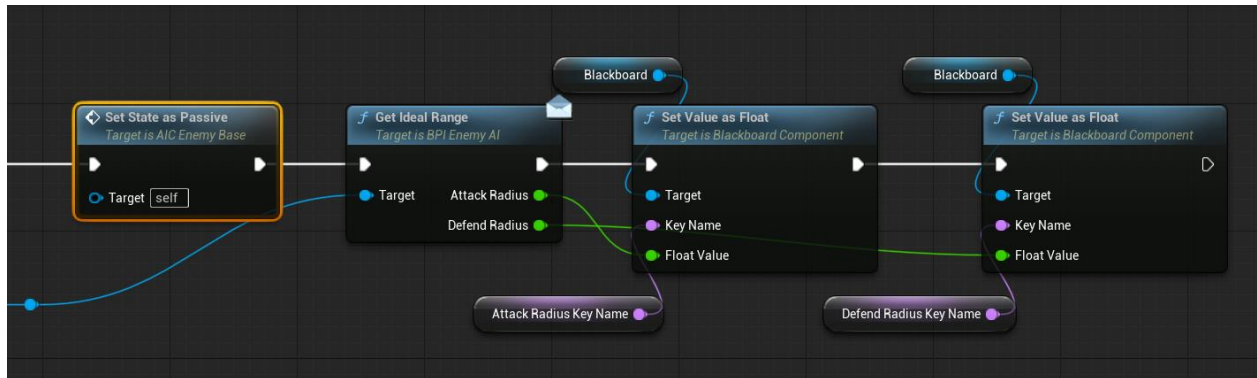


Figura 3-41 Asignación de *Attack Radius* y *Defend Radius* a la BB (Autoría propia).

Dentro del BT se ha creado un *Decorator*, denominado *BTD_IsAtPositiveDistance* (Figura 3-42). Su lógica consiste en obtener el valor del *AttackTarget* y calcular la distancia hasta él. Luego, compara esta distancia con el valor de *IdealRange* en la BB, aplicando un margen de error para evitar problemas innecesarios. Si la distancia es menor o igual al *IdealRange* más el margen de error, devuelve *true* al *Return Node*, indicando que la condición se cumple.

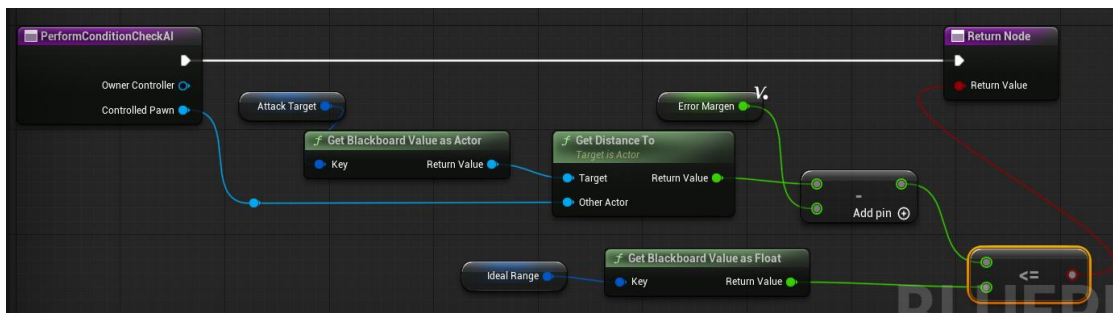


Figura 3-42 *BTD_IsAtPositiveDistance* (Autoría propia).

Este *Decorator* se aplicará en una nueva secuencia dentro de la ramificación de *Strafe* (Figura 3-44), y su cometido será comprobar si el enemigo está o no, a la distancia adecuada (*IdealRange*). Para ello, en los *details* del *BTD_IsAtPositiveDistance* se ajustará una lógica inversa a lo que debe hacer, asignándole como referencia la variable *DefendRadius*. Si el enemigo no se encuentra en la distancia del *DefendRadius*, se ejecutará un *Move To* para que vaya a esa distancia, como se muestra en la Figura 3-43.

Sin embargo, actualmente no existe un *Move To* específico que permita moverse al *DefendRadius*, ya que los *Move To* utilizados previamente empleaban un *Acceptable Radius* numérico para definir la distancia a la que debían posicionarse con respecto al *PointOfInterest* o *AttackTarget*. Para solucionar esto, se ha creado una nueva *Task* llamada *BTT_MoveToIdealRange*. Su lógica consiste en: introducir nuevamente las variables (*AttackTarget* e *IdealRange*) como en la Figura 3-42, llamarlas e introducirlas en el nodo *Move To* (*AttackTarget* como el *Target Actor*, e *IdealRange* como el *Acceptance Radius*) tal como se muestra en la Figura 3-43.

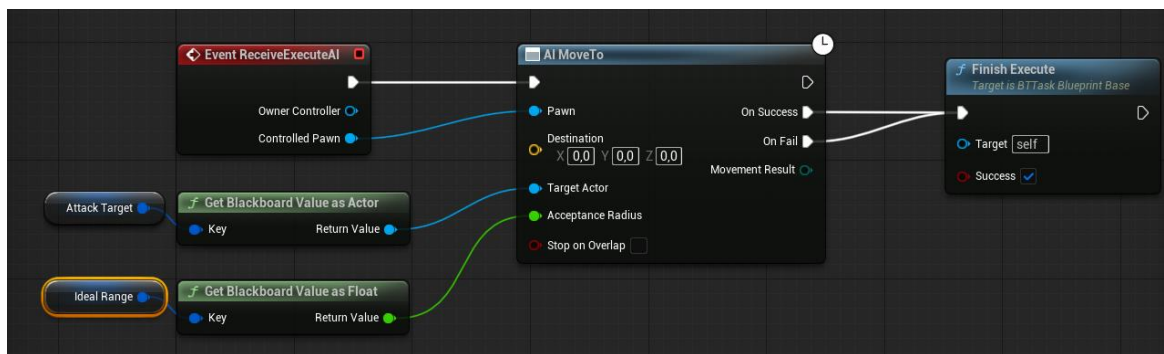


Figura 3-43 BTT_MoveToIdealRange (Autoría propia).

La ramificación *Strafe* quedaría entonces con la estructura mostrada en la Figura 3-44. Que ejecuta *Clear Focus* para eliminar cualquier objetivo previamente fijado, establece la velocidad del enemigo a *Sprinting* y luego ejecuta la *Task BTT_MoveToIdealRange*. De esta manera que si el enemigo no estaba dentro del rango adecuado mientras se encontraba en la secuencia *Strafe*, se moverá rápidamente hasta alcanzar la distancia ideal antes de proseguir.

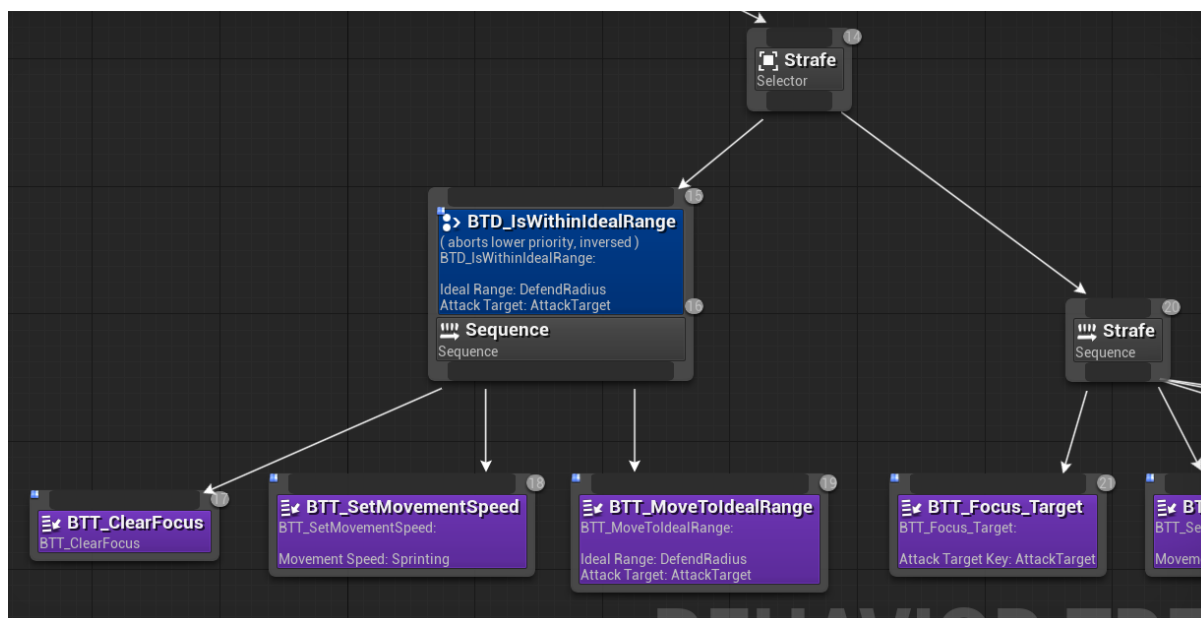


Figura 3-44 Actualización de la ramificación Strafe (Autoría propia).

Para finalizar con los EQS principales, se ha implementado una lógica que permite al enemigo esconderse detrás de una superficie, esperar un tiempo y luego atacar. Lo primero ha sido crear un nuevo EQS denominado *EQS_FindCover*, en el que se ha añadido un *Generator* de tipo *Simple Grid* (una cuadrícula de círculos). Como novedad, se ha agregado un *Test* de tipo *Trace*, cuya función es identificar puntos que no sean visibles. En el caso de un muro, si el enemigo se encuentra frente a él, los puntos que no estarían a la vista son los que más valdrían.

Hasta este punto, el enemigo seleccionaría cualquier punto que estuviera fuera de la línea de visión, pero al haber múltiples opciones, se busca que el enemigo elija la más cercana. Para lograrlo, se ha añadido un *Test Distance*, configurado *Score Only* y con la opción *Prefer Lesser*, lo que asigna puntuaciones más altas a los puntos más próximos. Como se observa en la Figura 3-45, el punto con la mejor puntuación (1.90) es el que está justo detrás del muro en la esquina. Los círculos azules en la Figura 3-45, son el resultado del *Path Finding*, el cual asigna una puntuación muy baja a las superficies inalcanzables para evitar que el enemigo considere moverse hacia ellas. Gracias a esta lógica, el sistema garantiza que el comportamiento del enemigo sea eficiente.

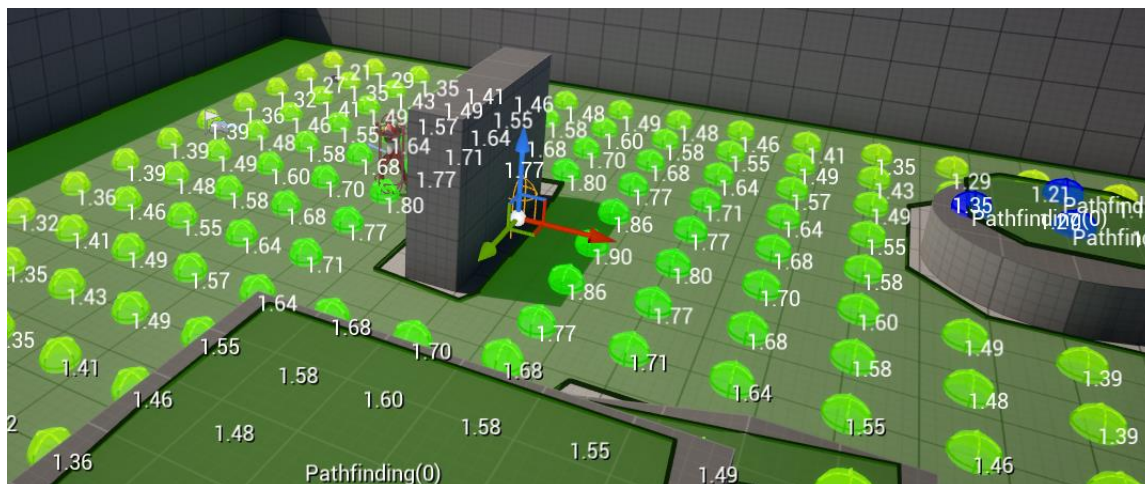


Figura 3-45 EQS_FindCover en el mundo gracias al Testing Pawn (Autoría propia).

En el BT se ha creado una nueva secuencia denominada *FindCover*, cuya función es: *Clear Focus*, establecer la velocidad a *Sprinting*, ejecutar el nuevo EQS, realizar un *Move To* hacia el *PointOfInterest*, esperar 3 segundos con un *Wait*, moverse al *IdealRange* y, finalmente, atacar. Esta secuencia se puede observar en la Figura 3-46 a continuación:

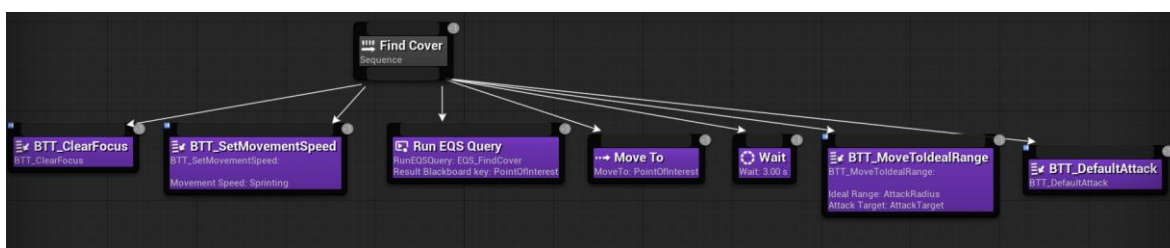


Figura 3-46 Lógica de Find Cover en el BT_EnemyBase (Autoría propia).

3.3.6 Sistema de salud

El escenario debe permitir que los enemigos inflijan daño, ya que, de lo contrario, sus ataques perderían realismo y esencia. Por ello, se ha desarrollado un sistema de salud general aplicable a cualquier enemigo, con el objetivo de que puedan recibir y causar daño, curarse e integrar una barra de salud visible sobre sus cabezas.

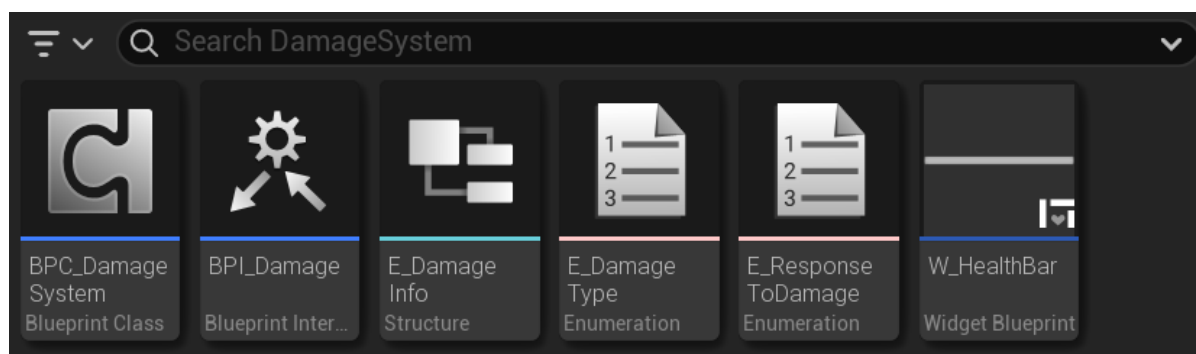


Figura 3-47 Carpeta con los elementos del sistema de daño (Autoría propia).

Como primer paso, se ha creado una carpeta que contendrá los distintos BP necesarios para la gestión del sistema de salud y daño, los cuales se pueden observar en la Figura 3-47. Entre ellos se encuentra *BPC_DamageSystem*, un *Blueprint Class* fundamental que regula cómo se aplican los impactos con la función *TakeDamage*, la recuperación de vida con la función *Heal* y las respuestas que se activan en función del tipo de daño recibido. Este sistema permite aplicar estas funciones y variables a los actores

para que reaccionen de manera coherente a los ataques, facilitando la implementación de mecánicas como curación, resistencias o efectos visuales asociados al daño.

BPI_Damage: Contiene las funciones usadas a lo largo del apartado, las cuales se exponen a continuación en la Figura 3-48:

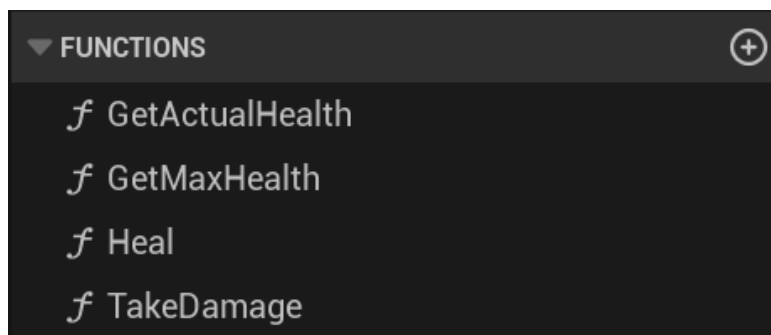


Figura 3-48 Funciones de *BPI_Damage* (Autoría propia).

E_DamageInfo: Es una estructura de datos (Figura 3-49) que almacena variables pudiendo referenciarlas como si fuesen una. Estas, contienen información del daño recibido y cómo actuar frente a este.



Figura 3-49 Variables de la estructura *E_DamageInfo* (Autoría propia).

Y, por último, *E_HitResponse* (Figura 3-50): define las posibles respuestas que un personaje o entidad puede tener al recibir daño. Así como, *E_TipoDeDaño* (Figura 3-50): clasifica los diferentes tipos de daño en el juego, como daño físico, del medio ambiente, de proyectil...



Figura 3-50 Enumeradores *E_HitResponse* y *E_TipoDeDaño* respectivamente (Autoría propia).

El siguiente paso ha sido acceder al *BPC_DamageSystem* y añadir las variables que serán utilizadas por las funciones. Se han incorporado las siguientes variables: *Health*, ya que los actores necesitan una cantidad de salud; *MaxHealth*, para definir el valor máximo que puede alcanzar la salud; y una llamada *IsDead*, que indicará si el personaje ha muerto. Una vez creadas las variables, se ha procedido a la implementación de las funciones *Heal* y *TakeDamage*, las cuales son esenciales para el funcionamiento del sistema.

La función *Heal* (Figura 3-51): primero, verifica si el personaje está muerto utilizando la variable *IsDead*. Si el personaje ha fallecido, la curación no se ejecuta. En caso contrario, se suma el valor de curación recibido a la variable *Health*. Para garantizar que la salud no supere su límite máximo, se emplea la función *Clamp (Float)*, que restringe el valor dentro de un rango entre 0 y *MaxHealth*. Finalmente, la nueva cantidad de salud se almacena en la variable *Health*.

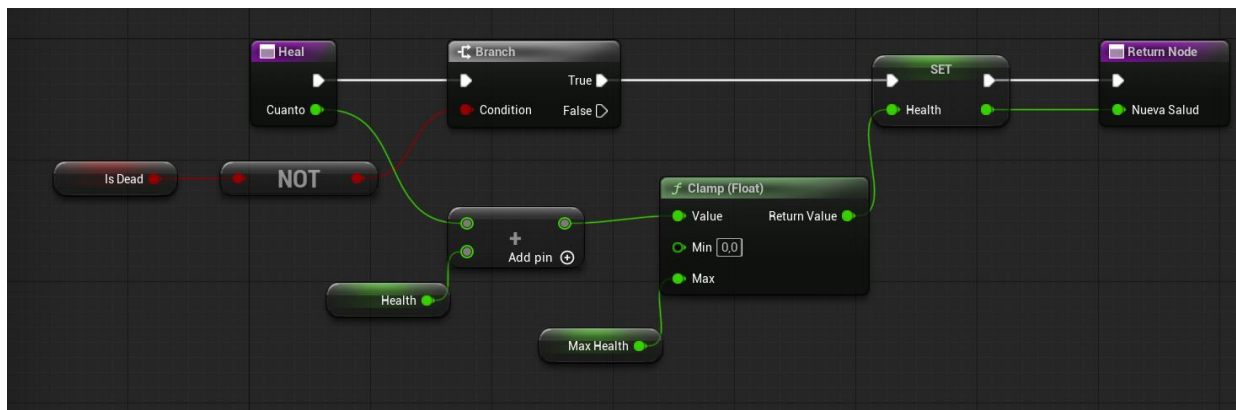


Figura 3-51 Función *Heal* (Autoría propia).

La función *TakeDamage* tiene una estructura más compleja, ya que debe gestionar diferentes situaciones en las que un actor recibe daño. Antes de entrar en detalle sobre su lógica, es importante destacar que se ha creado un *Macro* llamado *CanBeDamaged* (Figura 3-52), el cual permite devolver y recibir *Execution Pins* (los triángulos blancos al final de los nodos). Este *Macro* tiene como *inputs* la variable *CanBeBlocked* y como *outputs* las variables *DamageBlock*, *DoDamage* y *NoDamage*. Su lógica, mostrada en la Figura 3-52, consiste en verificar *IsDead*. Después, se revisa si el daño puede ser bloqueado (*CanBeBlocked*). En caso afirmativo, se comprueba si el actor está bloqueando (*IsBlocking*); si es verdadero, se activa *DamageBlock*. Si no está bloqueando, el daño se aplica activando *IveBeenDamaged*. Si el daño no se puede bloquear, se aplica directamente y se activa *IveBeenDamaged*. Si ninguna de estas condiciones se cumple, no recibe daño (*NoDamage*). La función posee tres *Event Dispatchers* que permiten ejecutar acciones en el *Event Graph* de los actores cuando son llamados a través del nodo *Assign on Event* (Crea una función asociada al *Event Dispatcher*, que se ejecuta cuando se llama a este). Estos *Event Dispatchers* son *WhenDead*, *WhenBlocked* y *HitResponse*, los cuales ejecutan funciones específicas dentro del *Event Graph* de los BP correspondientes.

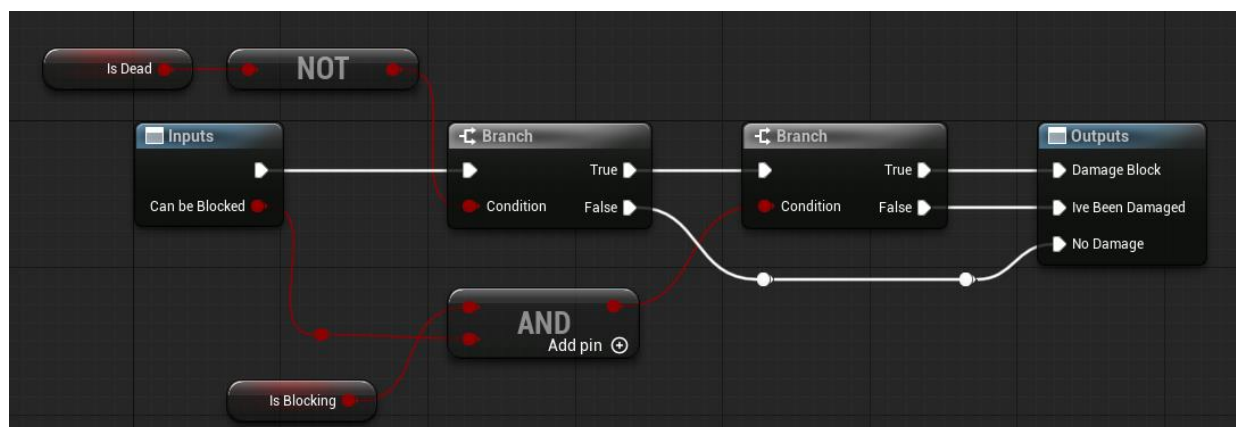


Figura 3-52 Macro *CanBeDamaged* (Autoría propia).

Con estos elementos definidos, se procede a la implementación de la lógica de *TakeDamage*. Cuando un actor recibe daño, se utiliza el *Macro CanBeDamaged* para determinar si debe bloquear, recibir daño o no hacer nada. Si el daño es bloqueado, se llama a la función *WhenBlocked* y el *Return Node* devuelve *WasDamaged* en *False*. Si el daño no es bloqueado, la función obtiene la cantidad de salud actual desde

el *Break E_DamageInfo*, resta el valor de daño recibido y actualiza la variable *Health*. Esta parte de la función se representa en la Figura 3-53.

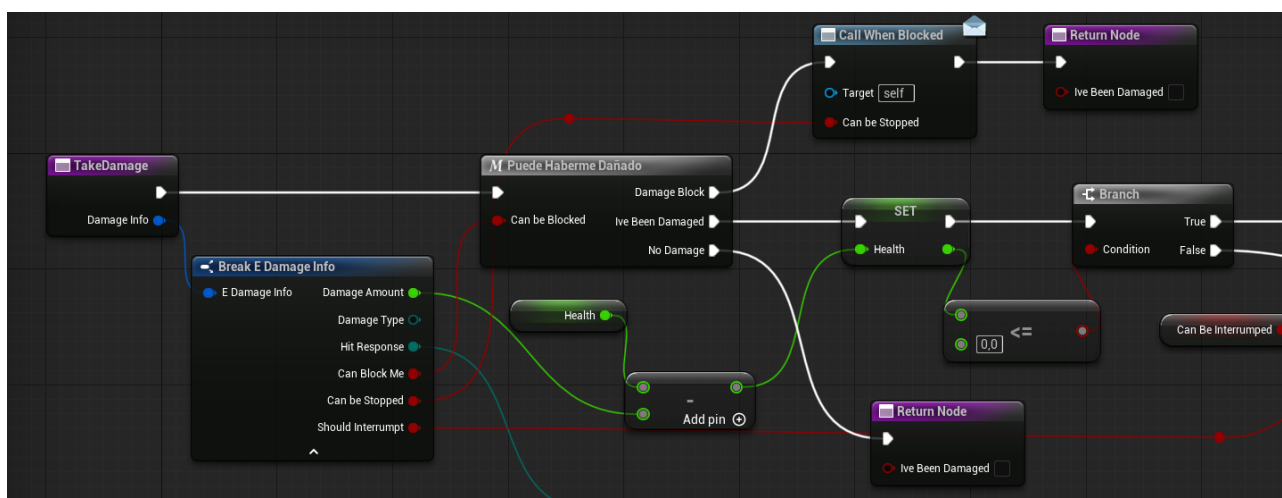


Figura 3-53 Inicio de la función *TakeDamage* (Autoría propia).

Luego, se verifica si la salud es menor o igual a 0; si lo es, la variable *IsDead* se establece en *True*, indicando que el actor ha muerto y llamando al *Event Dispatcher WhenDead* para gestionar la muerte. Si la salud no ha llegado a 0, se comprueba si el personaje puede interrumpir al ataque (*CanBeInterrupted*).

Si no puede ser interrumpido, la función devuelve *True* en *ShouldInterrupt*. Finalmente, si el actor ha recibido daño, se llama a la función *HitResponse* y el *Return Node* establece *IveBeenDamaged* en *True*, confirmando que el actor ha sufrido daño. El final de la lógica de la función expuesta en este párrafo queda reflejado en la Figura 3-54

Estas funciones (*GetMaxHealth*, *GetActualHealth*, *Heal* y *TakeDamage*) se han implementado tanto en *BP_Player* como en *BP_EnemyBase*, gracias a la integración previa de la interfaz en cada uno de ellos. En ambos casos, basta con llevar la variable del *BPC_SistemaDeDaño* a cada uno de estos BP y llamar a las funciones ya establecidas dentro del *BPC_SistemaDeDaño*. De esta manera, las funciones quedan completamente programadas y listas para ser utilizadas por estos actores dentro del juego.

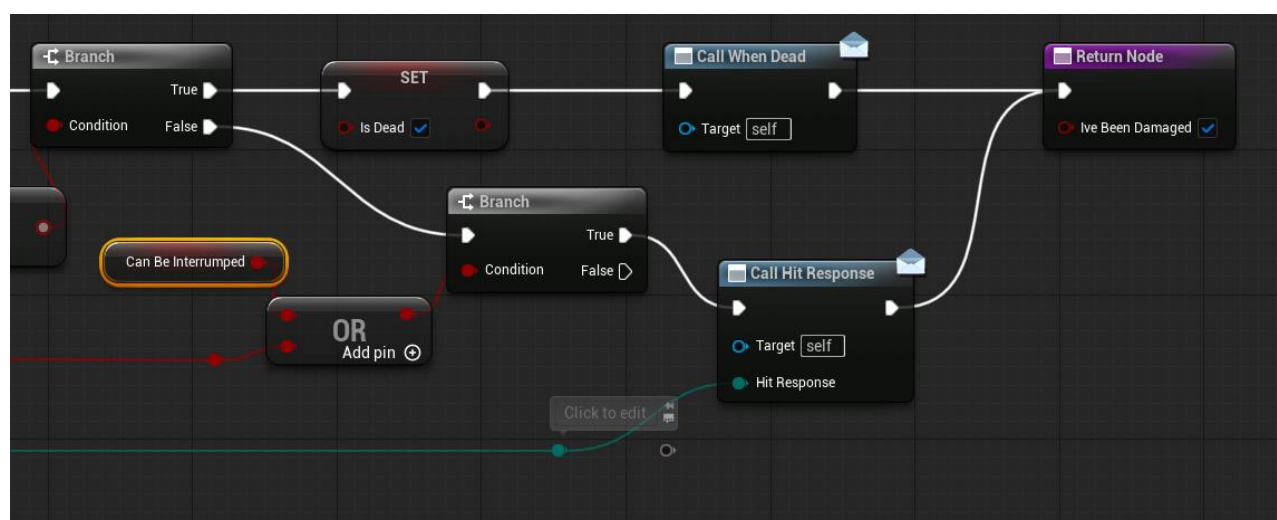


Figura 3-54 Continuación de la función *TakeDamage* (Autoría propia).

Junto con la creación del *Widget* de la *HealthBar*, esta implementación conforma la base del sistema de salud, permitiendo su integración en *BP_EnemyBase*. Para verificar su correcto funcionamiento,

dentro de *BP_Player* se ha implementado una función que permite infligir daño, cuya lógica se observa en la (Figura 3-55).

La función se activa al presionar la tecla F, lo que hace que el jugador inflija daño a un enemigo de clase *BP_EnemyBase*. Primero, obtiene una referencia a un actor de esta clase mediante *Get Actor of Class*. Luego, llama a *Report Damage Event* para registrar que se ha aplicado daño al enemigo, utilizando la ubicación del jugador obtenida con *Get Actor Location* como referencia. De manera simultánea, se genera un efecto visual con *Spawn Emitter at Location* en la misma posición, proporcionando una representación visual del impacto.

Posteriormente, la función *TakeDamage* es ejecutada en el enemigo, utilizando *E_DamageInfo* generado con *Make E Damage Info*, donde se establece un daño de 25 unidades y un tipo de daño *Default*. Finalmente, *TakeDamage* procesa el daño recibido y devuelve si ha sido aplicado correctamente. Más adelante, cuando se definan los diferentes tipos de enemigos, se establecerán tipos de daño específicos: *Melee Damage* para *BP_EnemyMelee* y *Projectile Damage* para *BP_RangedEnemy*.

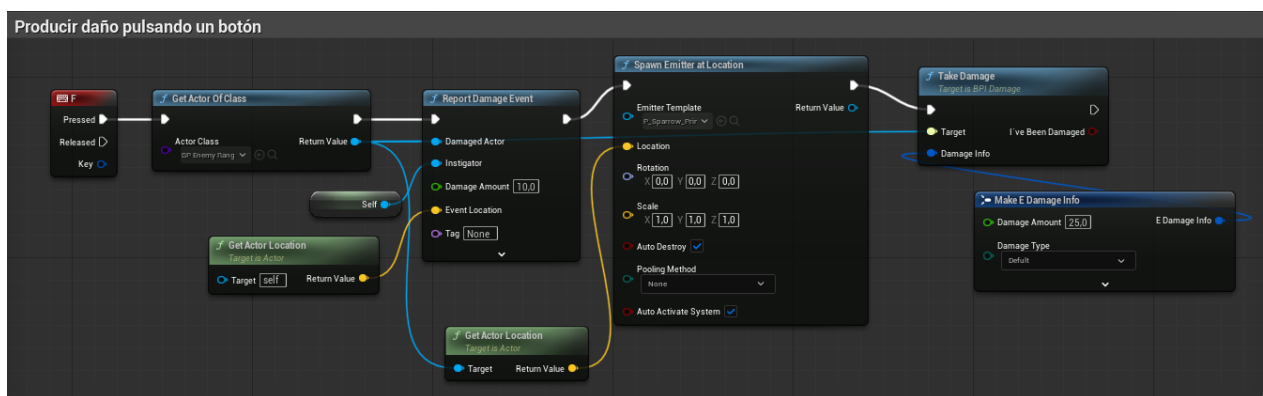


Figura 3-55 Función para hacer daño dentro de *BP_Player* (Autoría propia).

Por otro lado, se ha llevado a cabo la creación del *Widget* de la *Health Bar*, el cual, una vez diseñado, se añadió al *Viewport* de *BP_EnemyBase* para que se mostrara sobre su cabeza. El proceso ha consistido en agregar un *Canvas Panel* como contenedor para definir la estructura del rectángulo y una *Progress Bar* que representara visualmente la cantidad de vida restante. Como se observa en la (Figura 3-56), la *Progress Bar* se reduce gracias a la función *GetPercent* la cual calcula el porcentaje de vida restante dividiendo la salud actual entre la salud máxima del *Actor Dañable* (tipo *BPI_Damage*). El resultado se devuelve para actualizar visualmente la barra de vida en el HUD.

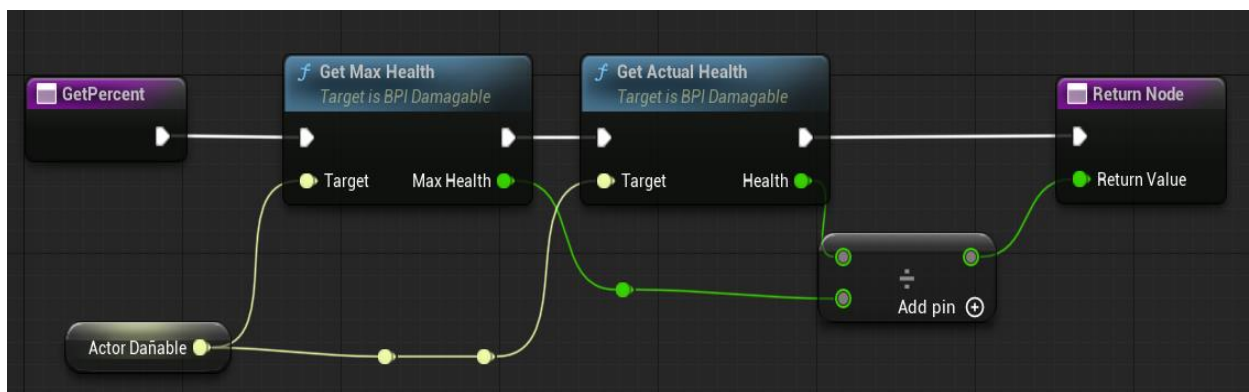


Figura 3-56 Función *GetPercent* (Autoría propia).

Bastaría con añadir después del *Event BeginPlay* la lógica que se muestra en la (Figura 3-57), asegurando que la barra de salud se represente correctamente encima del personaje al iniciar el juego. Hay que destacar que la *HealthBar*, es un *widget* común a todos los actores participantes en la simulación, por lo que es necesario implementar la lógica en cada uno individualmente.

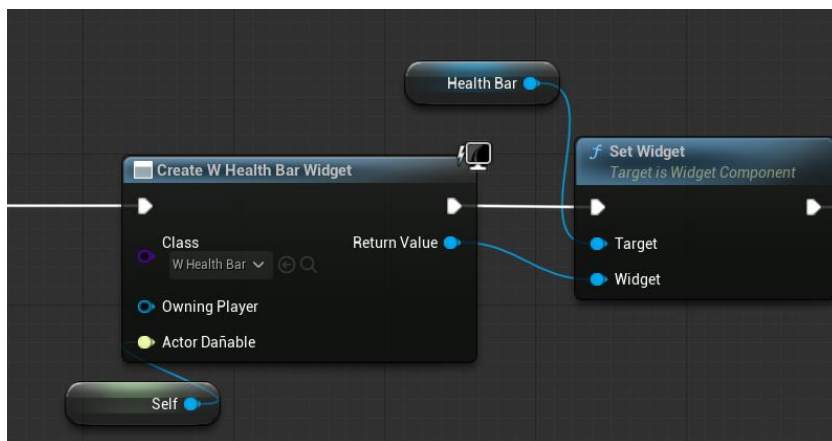


Figura 3-57 Lógica para establecer *Widget* en el mundo (Autoría propia).

Lo siguiente ha sido definir qué ocurre cuando el enemigo muere y su vida llega a 0. Para ello, en el *Event Graph*, se ha implementado la lógica que, como se ve en la (Figura 3-58), hace que el enemigo caiga al suelo y simule su muerte. La función *Die* se creó mediante un *Assign On Event* después de la lógica mostrada en la Figura 3-57, en el nodo *Event BeginPlay*.

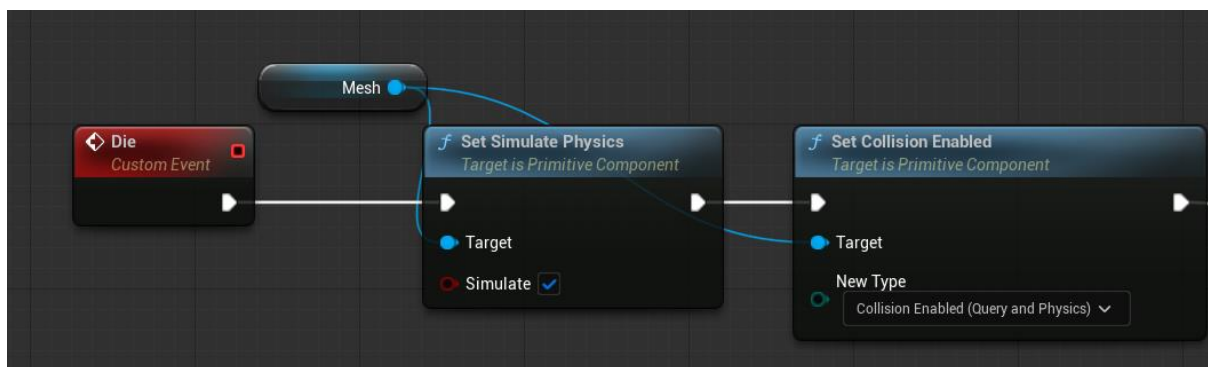


Figura 3-58 Comienzo de la función *Die* (Autoría propia).

Una vez establecida la lógica, se ha querido hacerla más profesional y estructurada. A continuación de la Figura 3-58 se ha creado una variable que referencia al *AIC_EnemyBase*, la cual ha sido utilizada en múltiples ocasiones a lo largo del desarrollo. Su implementación consiste en tomar como referencia *Self*, se obtiene el *AIController*, se realiza un *Cast* al *AIC_EnemyBase* y finalmente se almacena en la variable *AIC_EnemyBase*.

Una vez se ha definido esto, a continuación de la Figura 3-58 se han añadido nuevos nodos con el objetivo de detener la lógica del BT una vez muerto el enemigo (gracias al nodo *Stop Logic*). Esto se debe a que, en escenarios con múltiples enemigos que no solo atacan al *BP_Player*, sino también entre ellos, en el momento en el que uno muere, el que ha "ganado" continúa atacándolo. Para solucionar esto, se ha creado la función *SetStateAsDead*, cuya lógica es idéntica a la de la Figura 3-30 pero con el estado ajustado a *Dead*, colocándolo después de *Stop Logic*. Finalmente, se oculta la barra de salud cuando este muere.

Cuando el enemigo muere, el resto de los enemigos no saben que lo está. Para abordar esta situación, se ha diseñado un *service* que permita a los enemigos comprobar el estado de otros enemigos, permitiendo reaccionar de una manera adecuada. En el juego pueden darse conflictos que no involucran directamente al jugador principal, por lo que es fundamental que los NPCs sean capaces de evaluar estos cambios y actuar en consecuencia. Este servicio, llamado *BTS_StopAttackingIfTargetIsDead*, tiene como objetivo verificar si el *AttackTarget* ha muerto cuando el enemigo llega a la secuencia de ataque en el BT. Si el objetivo ha fallecido, el servicio interrumpe la acción de ataque y permite que el enemigo

cambie de estado en función de la lógica establecida, evitando que siga atacando a un enemigo ya derrotado.

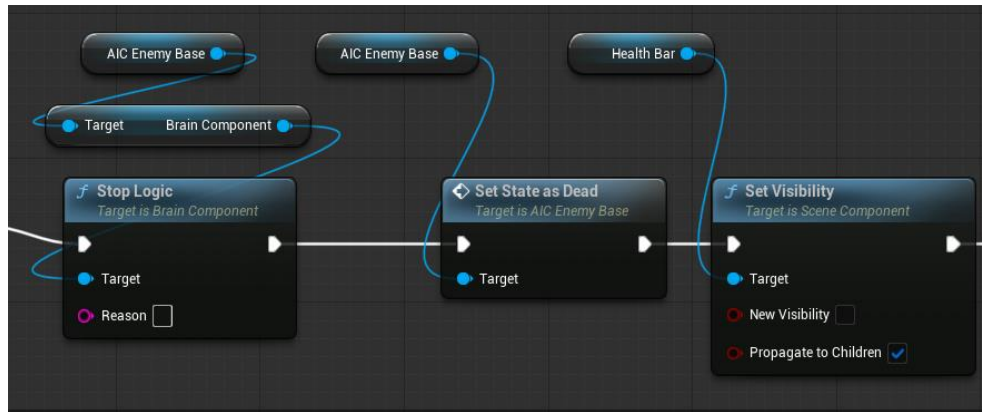


Figura 3-59 Continuación de la función *Die* (Autoría propia).

En el *BP_Player* se ha aplicado la misma lógica que en las Figura 3-58 y Figura 3-59, con la única diferencia de que se ha añadido el nodo *Disable Input*, permitiendo que, en caso de que el jugador muera, se deshabiliten sus controles, proporcionando una sensación más realista.

Una de las últimas mejoras implementadas en el sistema para dar mayor realismo a los personajes ha sido la creación de una acción que ejecute un *Montage* al recibir un golpe. Básicamente, se ha seguido un proceso similar al de la función *Die*, añadiendo un *Assign On Event* después *Assign On Event* de *Die*, creando el evento *DamageResponse*. Este evento comienza con un *Branch* para evitar que se ejecute en el francotirador. A continuación, detiene el movimiento del personaje y procede a ejecutar el *Montage*, el cual ha sido creado de la misma manera que los previamente mencionados. Una vez finalizada la animación, el personaje vuelve a su estado *Attacking*, asegurando así que pueda continuar combatiendo tras recibir daño.

Dentro del BT se ha implementado el estado *Frozen*, el cual sigue una lógica clara para garantizar que el enemigo detenga su acción temporalmente al recibir daño. Primero, ejecuta el nodo *BTT_ClearFocus*. Luego, se establece *Idle* mediante *BTT_SetMovementSpeed*. Finalmente, se utiliza el nodo *Wait* con una duración de 1 segundo antes de permitir que el enemigo continúe con su siguiente acción. Por último, se añade un *Decorator Blackboard* con la condición *State is equal to Frozen*, de la misma manera que en la Figura 3-31, asegurando que la secuencia solo se ejecute cuando el estado del enemigo sea *Frozen*.

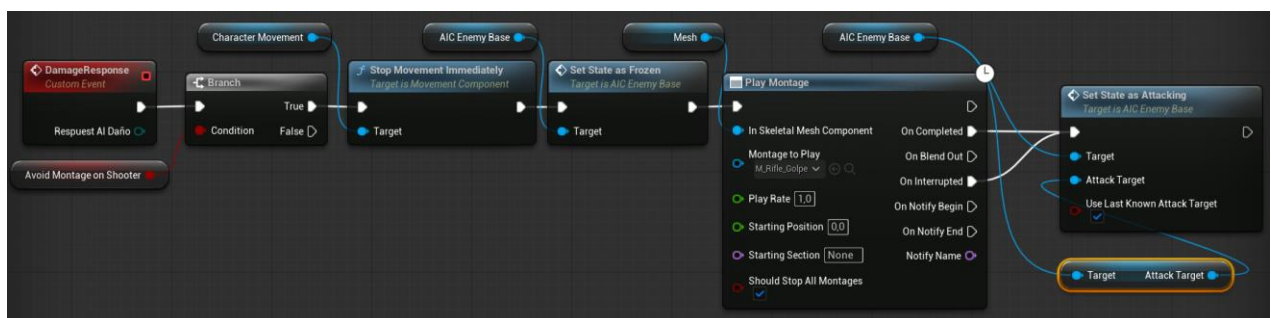


Figura 3-60 Lógica de *DamageResponse* (Autoría propia).

3.3.7 División de enemigos

Todo lo desarrollado hasta ahora ha establecido los fundamentos esenciales compartidos por todos los enemigos. Sin embargo, para ofrecer una mayor variedad de desafíos y situaciones al jugador, es necesario diversificar el comportamiento de los enemigos. De este modo, logrando que cada uno aporte una experiencia única al escenario, incentivando diferentes estrategias tácticas. En el apartado 3.2.3 se

ha mencionado qué enemigos coexisten en el mapa y su ubicación dentro del nivel. En este apartado, el objetivo es desglosarlos individualmente, detallando las acciones específicas que se han implementado en cada uno. Es por ello, que se le han puesto colores distintos que faciliten la diferenciación.

Lo último realizado a nivel general ha sido reestructurar el BT de *BP_EnergyBase*, quedando como se muestra en la Figura 3-61. El objetivo de esta reorganización es mantener únicamente las acciones comunes a todos los enemigos dentro del BT, asegurando una base sólida y modular. Sin embargo, muchas de las ramificaciones y lógicas ya implementadas en este BP serán reutilizadas y adaptadas en otros enemigos. Este BT ha quedado mucho más limpio y organizado, ya que la lógica de cada ramificación se ha agrupado en *SubTrees* específicos para cada estado. Esta optimización permite mejorar la legibilidad y facilitar futuras modificaciones.

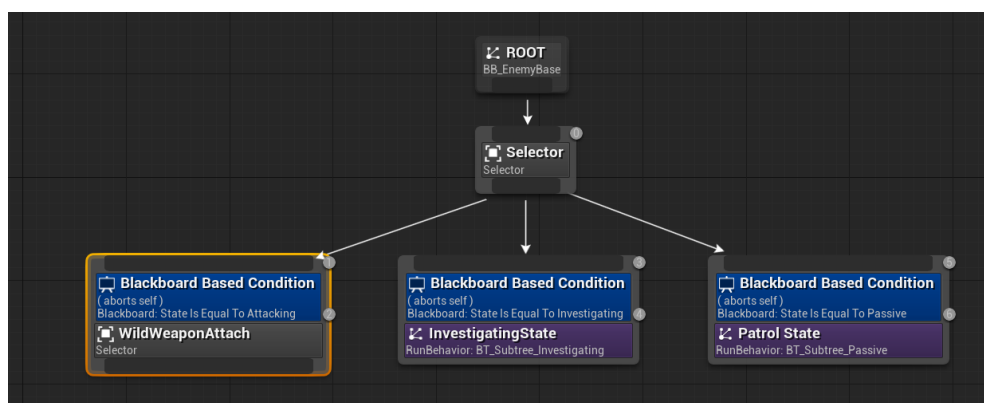


Figura 3-61 BT final del *BP_EnergyBase* (Autoría propia).

Para mejorar la movilidad e IA de los enemigos en el escenario, se han implementado *Smart Links*, elementos colocados estratégicamente que les permiten desplazarse de manera más fluida y reaccionar de forma más inteligente. Estos *NavLinks* facilitan el salto entre zonas con diferentes alturas, asegurando que los enemigos puedan seguir al jugador sin interrupciones. Para su integración, se ha añadido la función *JumpToDestination* dentro de *BPI_EnergyAI*, cuya lógica se encuentra implementada en *BP_EnergyBase*. Su funcionamiento consiste en obtener la ubicación actual del enemigo mediante *Get Actor Location* y establecer un destino con *JumpToDestination*. Luego, hace un *Break Vector* descomponiendo la posición en coordenadas y ajusta la altura del salto sumando unidades al eje Z. Lo continua *Suggest Projectile Velocity Custom Arc* que calcula la velocidad óptima para la trayectoria del salto, y *Launch Character* que aplica, permitiendo que el enemigo ejecute el movimiento de manera natural y eficiente.

3.3.7.1 *BP_EnergyMelee*



Figura 3-62 *BP_EnergyMelee* (Autoría propia).

Lo primero que se ha realizado al inicializar *BP_EnemyMelee* ha sido trasladar la lógica de *EquipWeapon* y la ejecución del *Montage* desde *BP_EnemyBase* a su propio *Event Graph*. Esto permite eliminar esa lógica del *BP_EnemyBase*, dejando en él únicamente las funciones realmente comunes a todos los enemigos. Dentro de *BPI_EnemyAI* se han añadido nuevas funciones que serán de utilidad a lo largo del desarrollo, aunque no todas estén directamente relacionadas con este enemigo en particular. Estas funciones pueden observarse en la Figura 3-63.

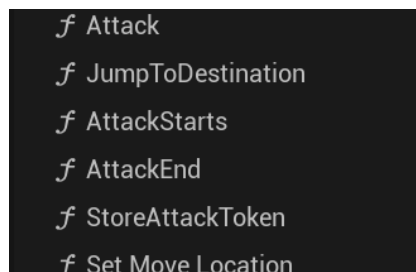


Figura 3-63 Continuación de *BPI_EnemyAI* (Autoría propia).

La función que resulta relevante tras trasladar la lógica mencionada en el párrafo anterior es la personalización del evento *Attack* para el *BP_EnemyMelee*. Se ha creado la función *Attack* dentro de *BPI_EnemyAI*, asegurando que tanto enemigos de combate cuerpo a cuerpo como a distancia puedan utilizar la misma estructura de ataque, pero con implementaciones personalizadas según su tipo de ataque.

En este punto del desarrollo, cabe mencionar que todos los enemigos utilizan *BT_EnemyBase* para actuar, lo cual no es óptimo, ya que cada uno tiene funciones específicas. Por ello, se ha creado una carpeta que contiene todos los BT de los enemigos que se van a usar (Figura 3-64).

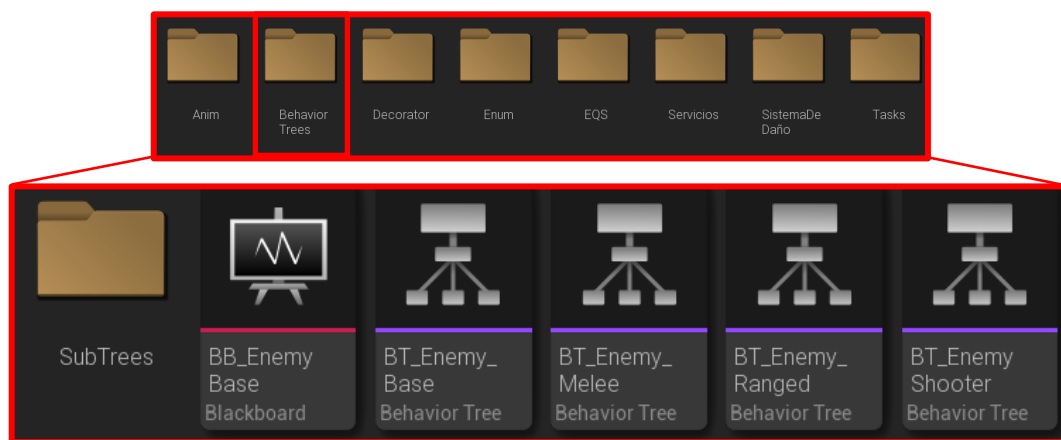


Figura 3-64 Carpeta de BT usados (Autoría propia).

El *AttackingState* ha eliminado de *BT_EnemyBase* (Figura 3-20) y trasladado a *BT_EnemyMelee*, ya que es una funcionalidad exclusiva de este enemigo. Esto implica que *BP_EnemyMelee* utilice las secuencias *Strafe*, *Attack* y *Is WieldingWeapon* dentro del *Attacking State*. Asimismo, también ha adoptado de *BT_EnemyBase*: *Passive State*, *Investigating State* y *Frozen State*.

Otra medida adoptada para asegurar que el programa ejecute el BT correctamente ha sido modificar la Figura 3-10 y reemplazarla por una versión más versátil (Figura 3-65). Para ello, se ha creado una variable llamada *BehaviorTree* dentro de *BP_EnemyBase*, la cual almacena el BT correspondiente en cada enemigo individualmente. Esta variable es utilizada en *AIC_EnemyBase* justo después del evento *On Possess*. En lugar de ejecutar *Run Behavior Tree* como en la figura anteriormente mencionada, primero se realiza un *Cast* a *BP_EnemyBase*, luego se obtiene la variable creada y, haciendo clic derecho, se convierte en *Validated Get*. Esto permite que, si la variable no es válida, se muestre un *Print String* con el mensaje "BT no válido", y si es válida, se ejecute el BT correspondiente.

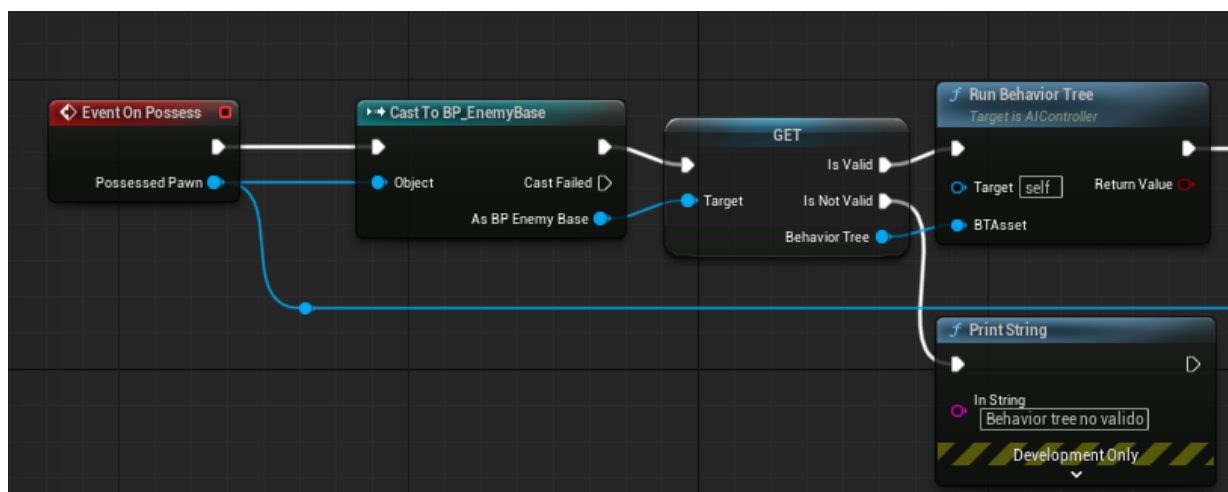


Figura 3-65 Uso de una variable para ejecutar todos los BT (Autoría propia).

Una vez establecida la dinámica del *BP_MeleeEnemy* y el comportamiento de su BT, es importante mencionar una funcionalidad adicional que mejora significativamente el realismo del comportamiento del enemigo. Esta consiste en que, cuando el enemigo pierda de vista a su objetivo después de un tiempo determinado (lo que en *AI Perception* se conoce como *Age*), comenzará a buscar en lugares probables donde este pueda haberse desplazado.

Para esta implementación, se han realizado varias acciones clave. Primero, se ha creado un array de tipo *Actor* para almacenar los personajes que el enemigo ha percibido. Este array se actualiza en la función *HandleSenseSight* utilizando el nodo *Add Unique*, asegurando que solo se añadan actores nuevos a la lista. Posteriormente, se ha creado un *Custom Event* llamado *CheckIfForgottenSeenActors*, encargado de gestionar la pérdida de visión de los objetivos y determinar la respuesta del enemigo.

Esta lógica en el *AIC_Energy_Base* comienza con el evento *CheckIfForgottenSeenActor* (Figura 3-66), que compara los actores percibidos actualmente con los que el enemigo había visto previamente. Para ello, obtiene la lista de actores visibles mediante *Get Known Perceived Actors* y la compara con *ActorsWeHaveSeen*. Si hay una diferencia entre ambas listas, significa que el enemigo ha perdido de vista a algún objetivo.

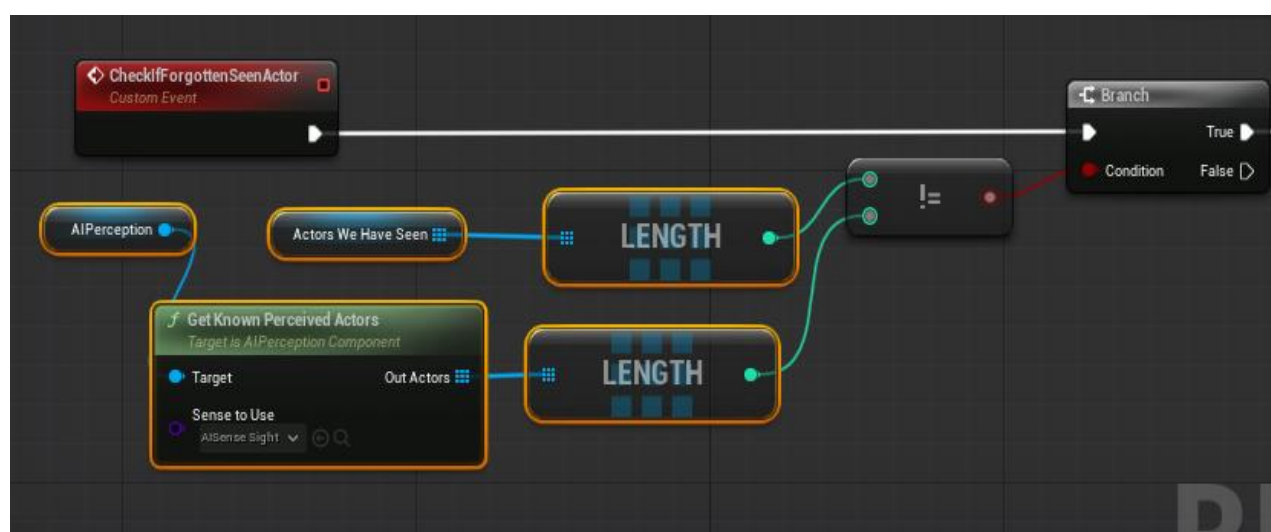


Figura 3-66 Primera parte de *CheckIfForgottenSeenActor* (Autoría propia).

Cuando esto ocurre, el sistema ejecuta un bucle *For Each Loop* que revisa los actores almacenados previamente y verifica si siguen en la nueva lista de percepción. Si un actor que estaba en la lista anterior

ya no se encuentra en la nueva, se considera olvidado y se ejecuta la función *HandleForgotActor* como se observa en la Figura 3-67.

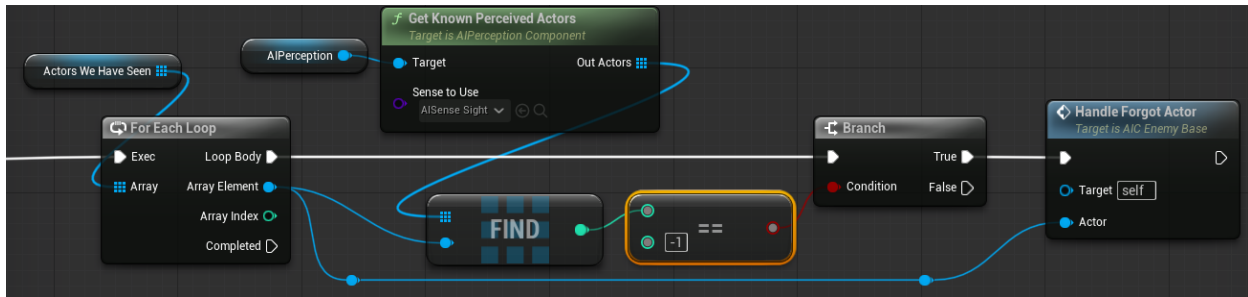


Figura 3-67 Segunda parte de *CheckIfForgottenSeenActor* (Autoría propia).

Dentro de *HandleForgotActor* (Figura 3-68), el actor se elimina del array *Actors We Have Seen* con el nodo *REMOVE*. Luego, verifica si el actor olvidado era el *AttackTarget*. En caso afirmativo, el *Branch* activa la función *SetStateAsPassive*, cambiando el estado del enemigo a *Passive* para que detenga su ataque y regrese a patrullar o buscar nuevos objetivos.

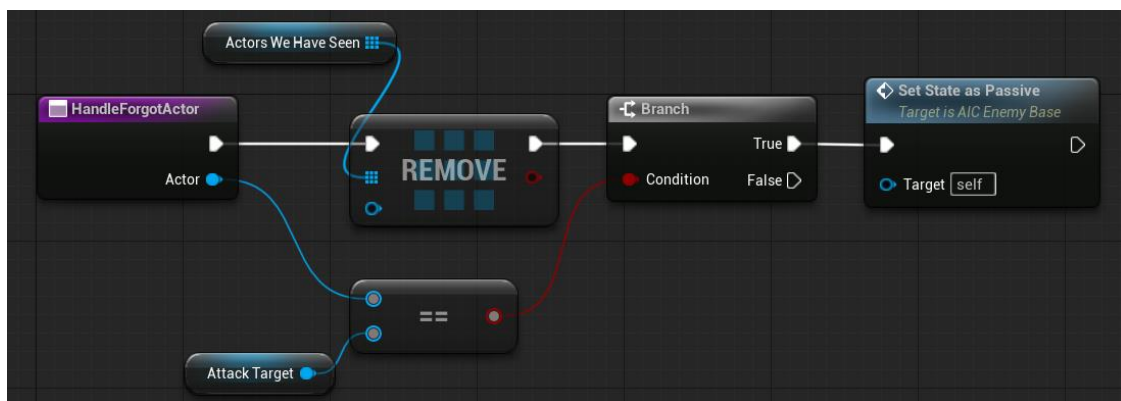


Figura 3-68 Función *HandleForgotActor* (Autoría propia).

Una vez creada la función, es necesario llamarla, por lo que dentro del evento *On Possess* se ha añadido un temporizador con *Set Timer by Event*, que ejecuta el evento *CheckIfForgottenSeenActor* cada 0.5 segundos. Esto permite que el enemigo revise constantemente si ha perdido de vista algún objetivo, asegurando que su percepción se mantenga actualizada. Además, en el evento *On UnPossess* (cuando el *AIController* pierde el control del enemigo), se ha utilizado *Clear and Invalidate Timer by Handle* para detener el temporizador y evitar que siga ejecutando la función cuando el enemigo ya no está activo.

Continuando la lógica de la función *CanSenseActor*, en la Figura 3-26, se encuentra un *Branch* que, si es *True*, activa *HandleSenseSight*, y si es *false*, no hace nada. Ahora, se ha añadido en el *false* una nueva función llamada *HandleLostSight* (Figura 3-69), cuyo propósito es gestionar el comportamiento del enemigo cuando pierde de vista a su objetivo.

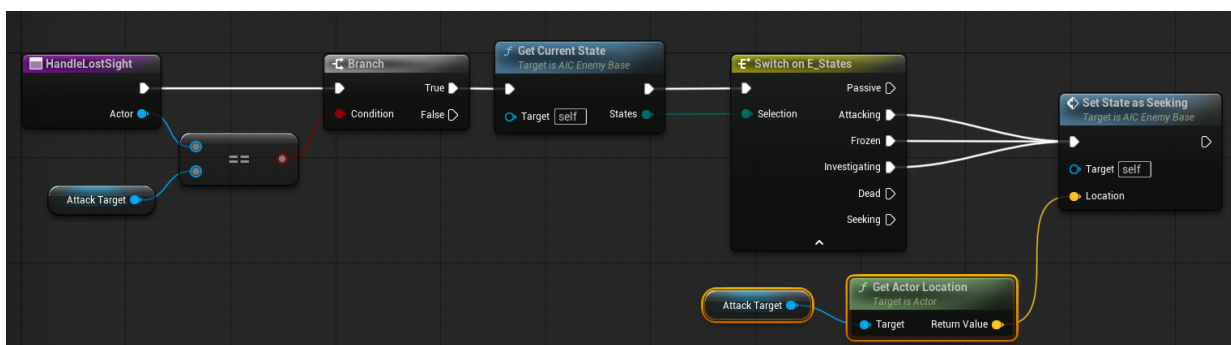


Figura 3-69 Función *HandleLostSight* (Autoría propia).

Esta función primero verifica si el actor que ha dejado de percibir es el *AttackTarget*. Si la condición se cumple, obtiene el estado actual del enemigo mediante *GetCurrentState* y lo evalúa con un *Switch on E_States*. Si el estado del enemigo es *Attacking*, *Investigating* o *Frozen*, se activa *SetStateAsSeeking* lo que cambia su estado a *Seeking*. Para determinar el punto de búsqueda, se obtiene la última ubicación conocida del *AttackTarget* con *Get Actor Location* y se establece como referencia, permitiendo que el enemigo continúe rastreando su objetivo de manera realista.

Se ha creado para el BT, un *Subtree* llamado *BT_SubTree_Seeking*, el cual sigue la misma estructura implementada en *Investigating* y *Passive*. Este *SubTree* ejecuta una secuencia que se divide en tres ramificaciones: *GoToLastKnownLocation*, *Seeking* y *Finish Seeking*, las cuales se ejecutan en orden de izquierda a derecha dentro de la secuencia.

La primera ramificación, *GoToLastKnownLocation* (Figura 3-70), tiene como objetivo dirigir al enemigo a la última posición donde vio a su objetivo antes de perderlo de vista. La secuencia inicia con *BTT_ClearFocus*, que elimina cualquier fijación previa para evitar que el enemigo continúe enfocado en un punto inexistente.

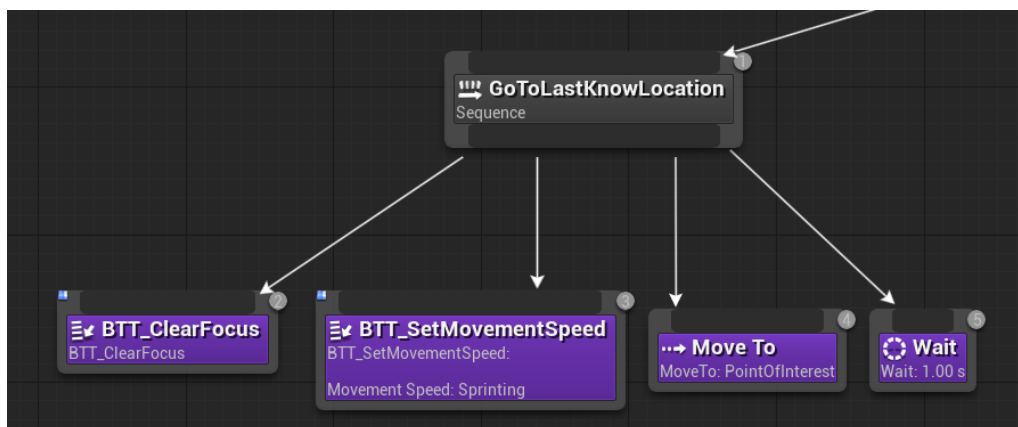


Figura 3-70 Ramificación *GoToLastKnownLocation* (Autoría propia).

Luego, *BTT_SetMovementSpeed* establece la velocidad en *Sprinting*, asegurando que el enemigo se desplace rápidamente hacia la ubicación en cuestión. A continuación, el nodo *Move To* dirige al enemigo hacia *PointOfInterest*, que es la posición almacenada cuando el estado se establece en *Seeking*. Finalmente, un nodo *Wait* introduce una pausa de 1 segundo, lo que le permite al enemigo evaluar la situación antes de decidir su próximo movimiento.

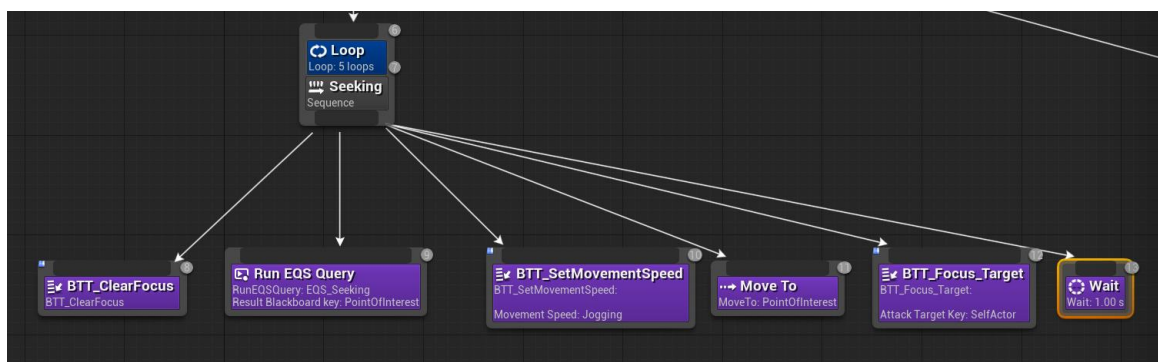


Figura 3-71 Ramificación *Seeking* (Autoría propia).

La segunda ramificación, *Seeking* (Figura 3-71), es responsable de que el enemigo busque activamente a su objetivo en caso de que no lo encuentre en la última ubicación conocida. Para esto, la secuencia se ejecuta dentro de un bucle de 5 iteraciones, permitiendo que el enemigo explore distintos puntos probables.

La lógica comienza con *BTT_ClearFocus*. Luego, un *Run EQS Query* ejecuta una consulta del sistema de percepción (*EQS_Seeking*), que genera ubicaciones estratégicas donde podría encontrarse el enemigo, almacenando el mejor resultado en la *Blackboard Key: PointOfInterest*. Posteriormente, *BTT_SetMovementSpeed* ajusta la velocidad a *Jogging* para que el movimiento sea más controlado durante la exploración. Luego, *Move To* dirige al enemigo hacia el *PointOfInterest* generado por el *EQS*. Si al llegar no encuentra nada, *BTT_FocusTarget* fija la mirada en el *PointOfInterest*, lo que le permite analizar mejor el entorno.

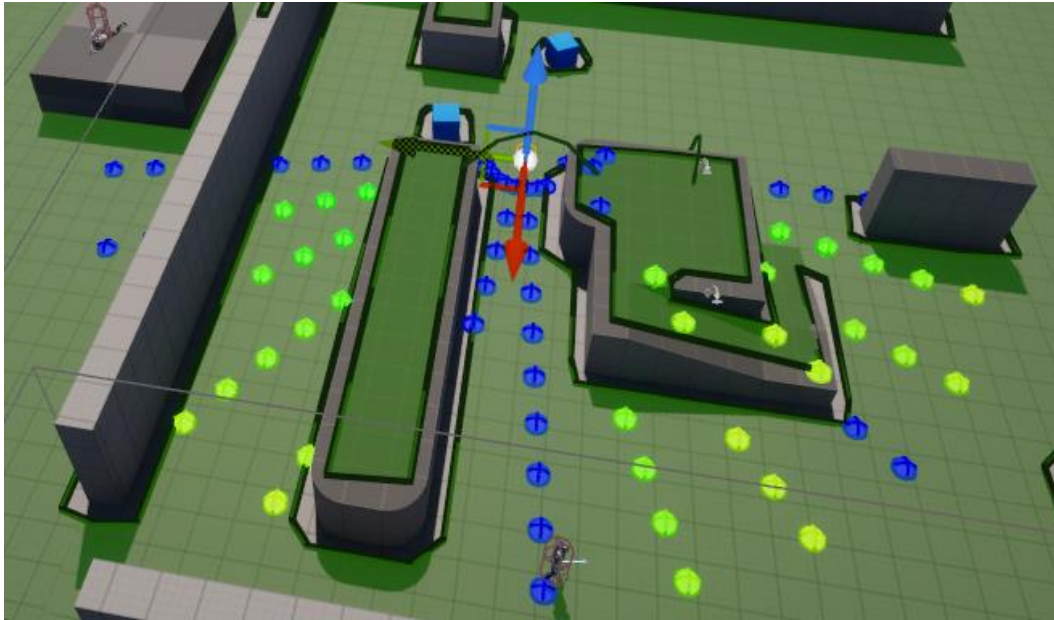


Figura 3-72 *EQS_Seeking* (Autoría propia).

Para implementar *EQS_Seeking*, se ha creado un *Query* que genera puntos en forma de cono de 180° frente al *Querier*, con pasos de 14°. Estos puntos se filtran con dos pruebas: *PathCost* prioriza ubicaciones accesibles y cercanas (entre 200 y 2000 unidades), y *Trace to Querier on Visibility* selecciona solo puntos donde la línea de visión esté bloqueada, permitiendo al enemigo buscar zonas donde podría estar oculto el objetivo. En la Figura 3-72 se muestran las bolas con sus costes, siendo las azules las de menos importancia y las verdes las de interés.

La tercera y última ramificación, *FinishSeeking*, se ejecuta después de completar las 5 iteraciones de búsqueda. Si el enemigo no ha logrado encontrar a su objetivo tras recorrer los distintos puntos generados por el *EQS*, el estado del enemigo se establece nuevamente en *Passive*, permitiendo que retome su patrullaje o espere nuevas señales de presencia del jugador. De esta manera acabaría la acción de búsqueda.

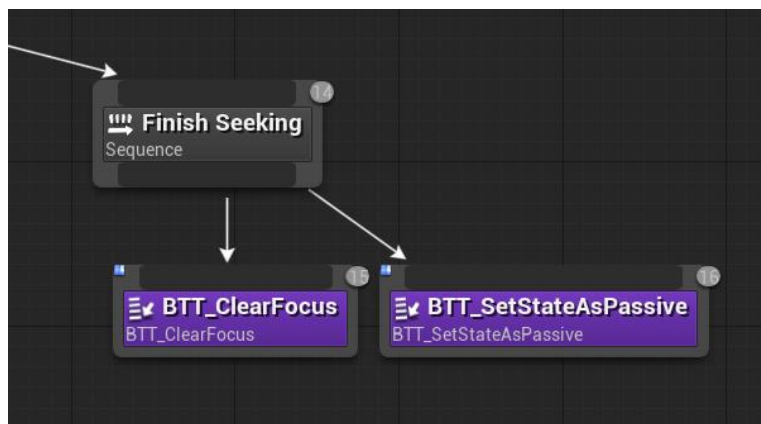


Figura 3-73 Ramificación *Finish Seeking* (Autoría propia).

Lo último que se ha hecho con este enemigo es mejorar su sistema de ataque mediante la implementación de varios *Montages*, creando dos tipos de ataques: *KnifeShortAttack* y *KnifeLongAttack*. La lógica del ataque de este enemigo ha evolucionado significativamente con respecto a la versión inicial programada en *BP_EnergyBase*, volviéndose más compleja y realista.

El primer paso ha sido diferenciar dos tipos de ataques según la distancia al objetivo: uno para cuando el enemigo está cerca y otro para cuando está lejos. El primero en desarrollarse ha sido *KnifeShortAttack*, que combina dos animaciones de ataque alternas. En la primera parte de su lógica, mostrada en la Figura 3-74, la ejecución comienza con el evento *KnifeShortAttack*, que recibe como parámetro el *AttackTarget*. Luego, llama a la función *Parent: Attack* para asegurar que la lógica general del ataque se ejecute antes de proceder con los *Montages* específicos del ataque corto con cuchillo.

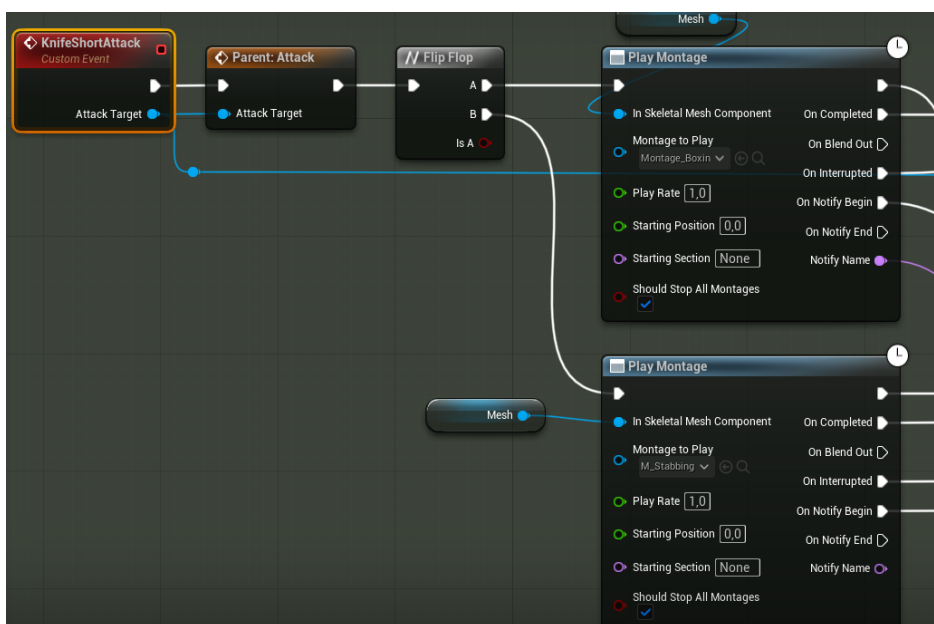


Figura 3-74 Primera parte función *KnifeShortAttack* (Autoría propia).

Para alternar entre diferentes animaciones, se utiliza un nodo *Flip Flop*, que intercala entre dos estados (A y B) cada vez que se ejecuta el evento. Dependiendo del estado en el que se encuentre, se activa una de las dos animaciones de ataque a través de los nodos *Play Montage*. Si el estado es A, se reproduce la animación *Montage_Boxing*, y si es B, se ejecuta *M_Stabbing*.

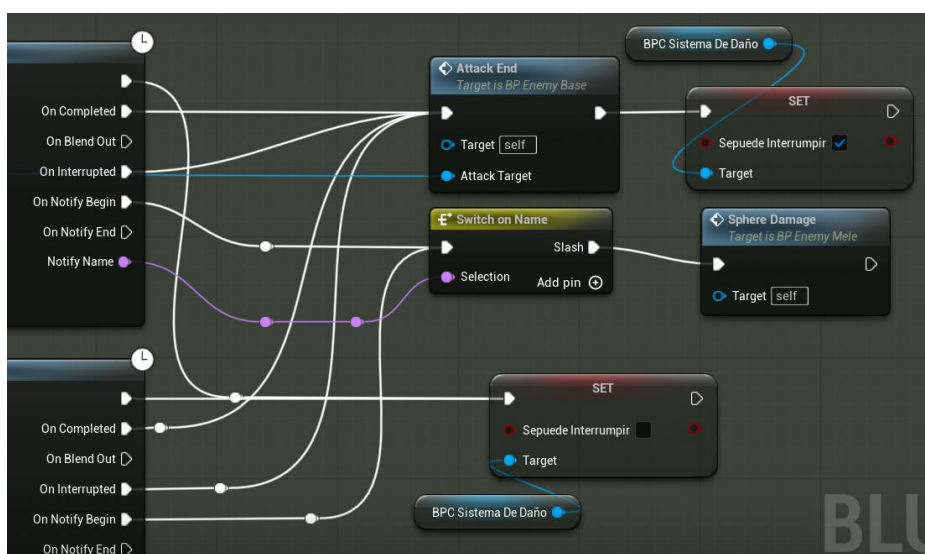


Figura 3-75 Segunda parte función *KnifeShortAttack* (Autoría propia).

Después de esto como se ve en la Figura 3-75, cuando se activa el *Notify*, el sistema evalúa su nombre usando *Switch on Name*. Si el nombre coincide con *Slash*, previamente establecido como *notify* en el propio *Montage*, se ejecuta la función *SphereDamage*. Simultáneamente, la variable *SePuedeInterrumpir* (*CanbeInterrumpid*) se gestiona en el *BPC_DamageSystem*. Al inicio del ataque, se establece en *true* para evitar interrupciones, y al finalizar, se devuelve a *false*, permitiendo que el enemigo pueda ser interrumpido nuevamente. Además, al finalizar el ataque, se llama a *AttackEnd* en *BP_EnemyBase* para restablecer el estado del enemigo tras la acción.

En lo que respecta a *KnifeLongAttack*, su lógica es muy similar a la de *KnifeShortAttack*, con la diferencia principal en el *Montage* y en el flujo posterior al nodo *Switch on Name* (Figura 3-76). En este caso, solo se emplea un único *Montage*, en lugar de alternar entre dos, asegurando que el ataque largo tenga una ejecución específica y diferenciada.

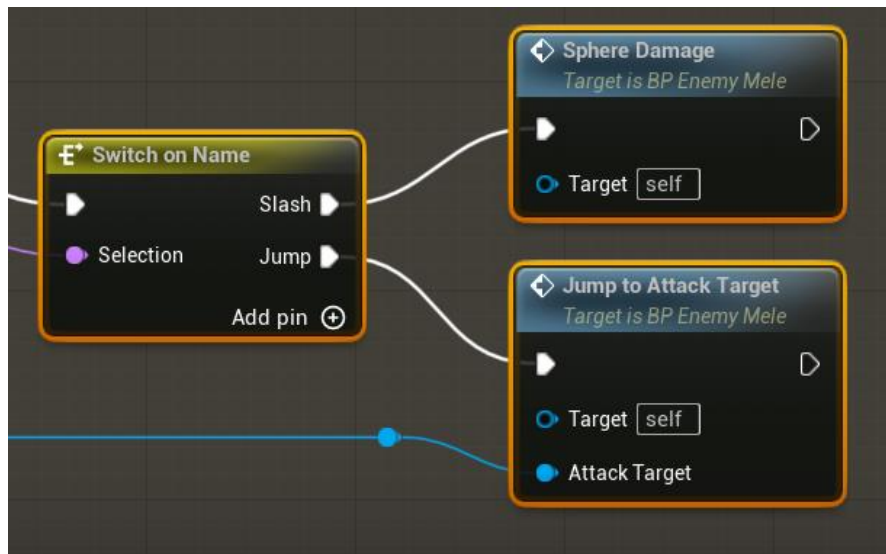


Figura 3-76 Diferencia de lógica del *KnifeLongAttack* (Autoría propia).

Antes de profundizar en la función *JumpToAttackTarget*, se ha implementado la función *SphereDamage*, como se observa en la última figura. Esta función permite visualizar si el enemigo ha impactado a su objetivo y aplicar daño de manera precisa.

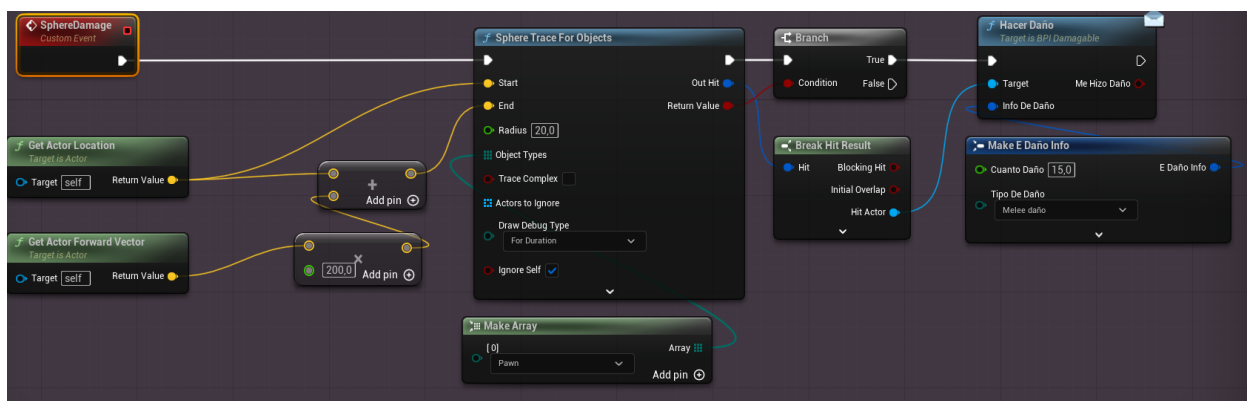


Figura 3-77 Función personalizada *SphereDamage* (Autoría propia).

La ejecución comienza con el evento *SphereDamage* (Figura 3-77), que obtiene la posición actual del enemigo mediante *Get Actor Location*. Luego, calcula una posición adelantada 200 unidades en la dirección en la que está mirando, utilizando *Get Actor Forward Vector*. A partir de esta información, *Sphere Trace for Objects* realiza una detección esférica con un radio de 20 unidades entre la posición inicial y el punto calculado, filtrando actores del tipo *Pawn* dentro de esa área. Si la detección identifica un objetivo, un *Branch* valida si hubo colisión. En caso afirmativo, *Break Hit Result* extrae el actor

impactado y se genera un *E_DañoInfo* con 15 unidades de daño de tipo *Melee daño*. Finalmente, *TakeDamage* aplica el daño al objetivo detectado, garantizando que el ataque tenga efecto en los enemigos dentro del rango establecido.

Como se observa en la Figura 3-76, el flujo de ejecución llama a la función *JumpToAttackTarget*, la cual establece la base del nuevo movimiento de ataque. En esta función, el enemigo realiza un salto desde una distancia considerable hacia su objetivo con la intención de atacarlo en el aterrizaje. Para hacer este ataque más preciso y realista, la lógica también calcula la posición futura del *AttackTarget* en caso de que este se esté moviendo. Esto permite que el enemigo ajuste su trayectoria de salto y aterrice en la ubicación donde se espera que el jugador esté en el momento del impacto, evitando ataques fallidos y haciendo que la mecánica de combate sea más dinámica y creíble.

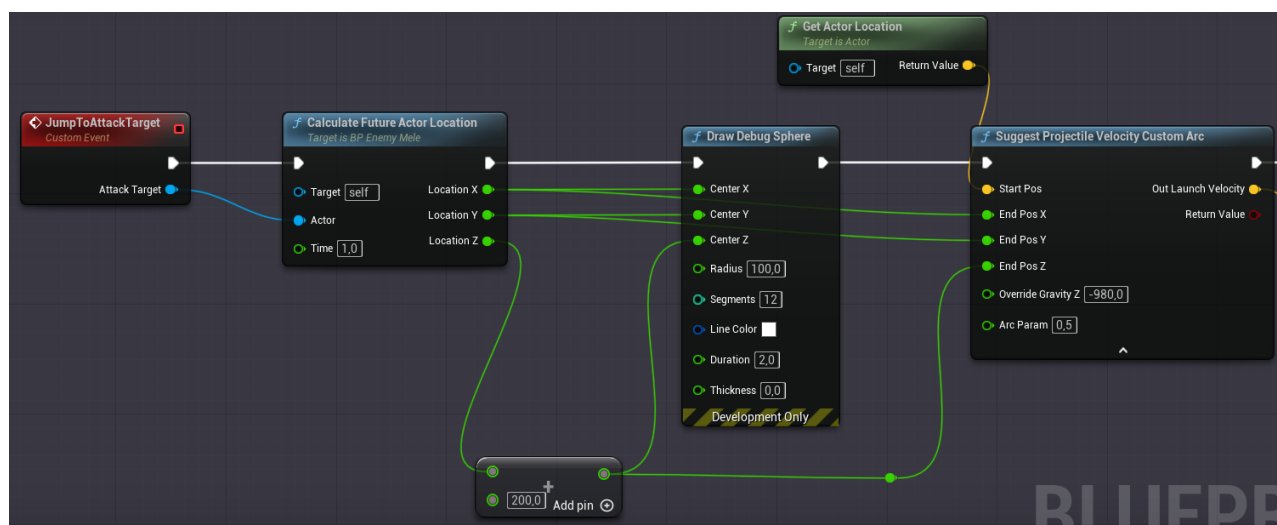


Figura 3-78 Primera parte función *JumpToAttackTarget* (Autoría propia).

Primero, como se observa en la Figura 3-78, se obtiene la posición actual del enemigo con *Get Actor Location* y, mediante la función *CalculateFutureActorLocation* (Figura 3-79), predice dónde estará el objetivo en un tiempo determinado. Esta predicción se basa en la velocidad actual del actor, obtenida con *Get Velocity*, la cual se multiplica por el tiempo de anticipación ingresado y se suma a la ubicación actual del objetivo, estimando así su futura posición.

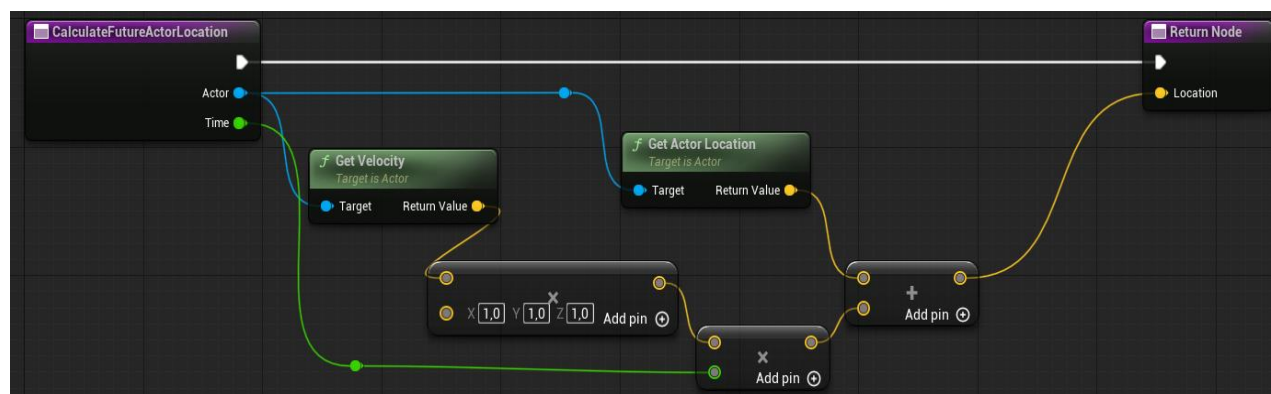


Figura 3-79 Función *CalculateFutureActorLocation* (Autoría propia).

Una vez calculado el punto de destino, se visualiza con *Draw Debug Sphere* para verificar su precisión en desarrollo. Luego, *Suggest Projectile Velocity Custom Arc* determina la trayectoria óptima para alcanzar el punto previsto y *Launch Character* aplica la velocidad calculada para ejecutar el salto. Durante el vuelo, el evento *Bind Event to Landed Delegate* detecta el aterrizaje, activando *OnLand* cuando el enemigo toca el suelo. Finalmente, tras aterrizar, se ejecuta *Stop Movement Immediately* para

detener cualquier desplazamiento adicional, asegurando una transición fluida a la siguiente acción. Esta última parte de la lógica se observa a continuación en la Figura 3-80:

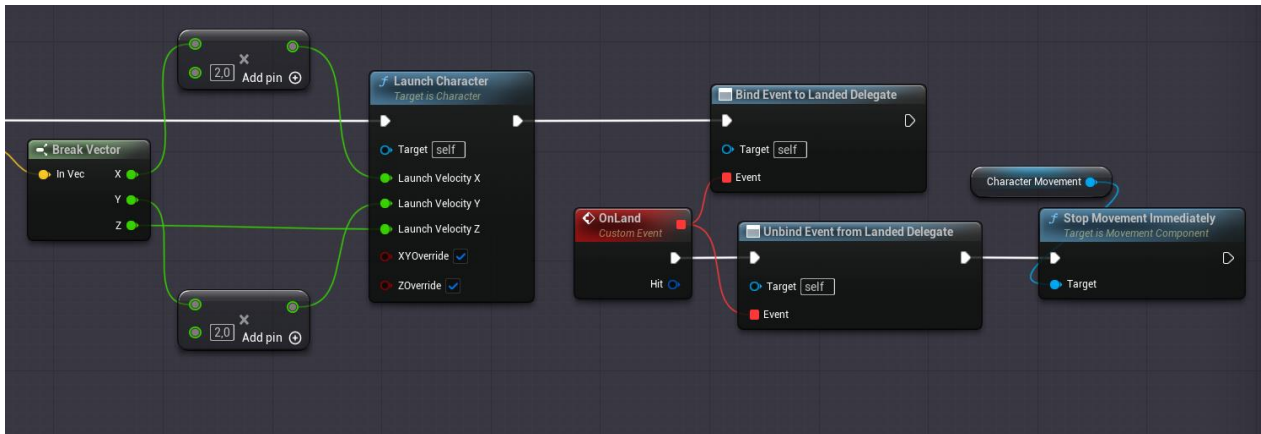


Figura 3-80 Segunda parte función *JumpToAttackTarget* (Autoría propia).

Hasta este punto, el BT del *Melee Enemy* contaba con un *AttackingState* que se dividía en tres acciones principales. Primero, *IsWieldingKnife*, que verifica si el enemigo porta un arma antes de proceder con el ataque. Luego, *Strafe*, encargado de hacer que el enemigo rodee a su objetivo utilizando los valores establecidos en *IdealRange*. Finalmente, la tercera acción es el evento *Attack*, cuya lógica específica se puede observar en la Figura 3-81.

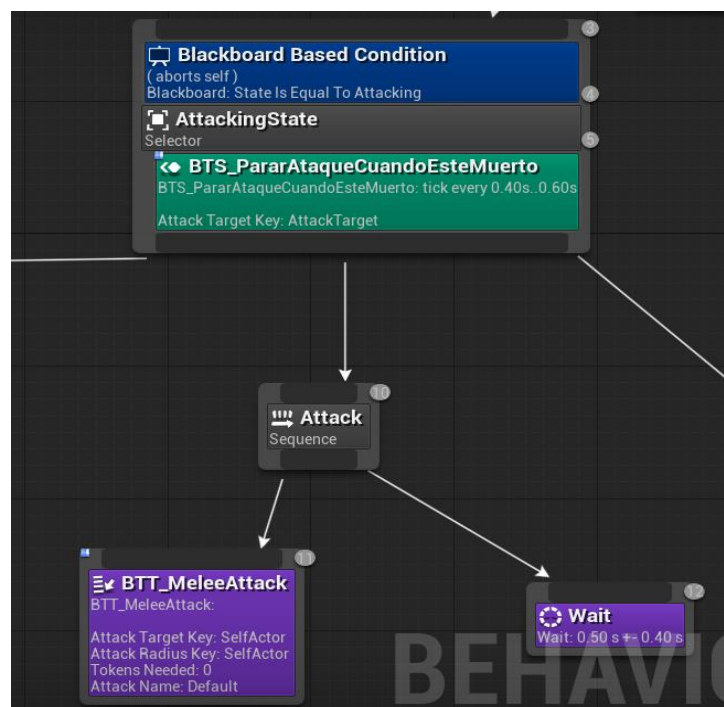


Figura 3-81 Ramificación *Attack* del *BP_EnergyMelee* (Autoría propia).

Lo primero ha sido añadir una nueva clave en la BB, llamada *DistanceToAttackTarget*, de tipo *float*, con el objetivo de actualizar constantemente este valor y monitorear la distancia en tiempo real. Para lograrlo, se ha creado un *service* llamado *BTS_UpdateDistanceToTarget*. Este sistema obtiene la ubicación del enemigo y la del *AttackTarget* en cada *tick*, calcula la distancia entre ambos y la almacena en la clave *DistanceToTargetKey* de la BB. Esto permite que la IA ajuste su comportamiento en función de la proximidad del objetivo. Para implementarlo, se ha creado un *Selector* llamado *CombatChoices* que parte del *AttackingState*, el cual contiene este *Decorator*. Desde ahí, se ramifican las acciones de *Strafe* y *Attack*.

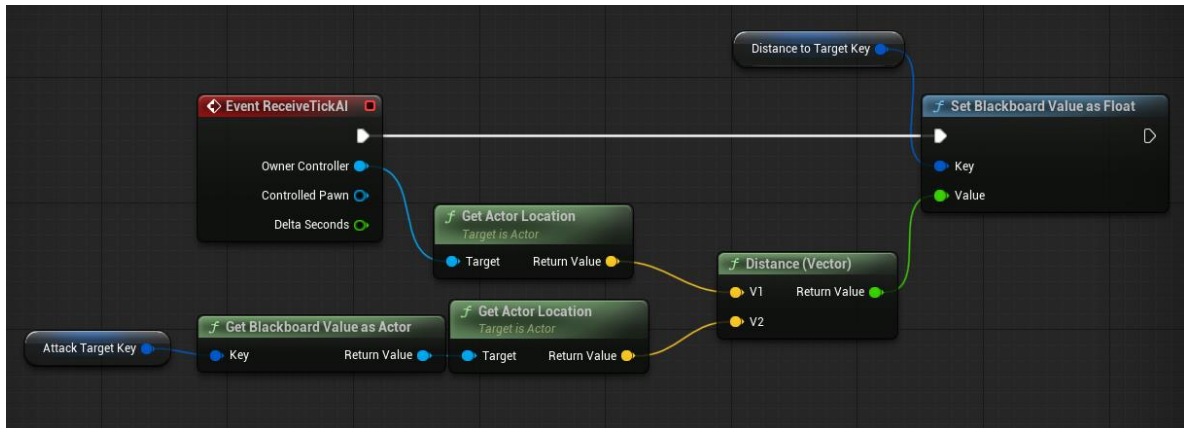


Figura 3-82 *BTS_UpdateDistanceToTarget* (Autoría propia).

Una vez establecida la distancia entre los actores, se ha implementado la lógica para los dos tipos de ataque dentro del BT. Para ello, desde el nodo *Attack*, se ha creado un *Selector* que determina cuál de los ataques ejecutar, además de incluir un *Wait*, como se muestra en la Figura 3-83. Esto permite que el enemigo elija de forma dinámica entre diferentes opciones de ataque en función de la situación.

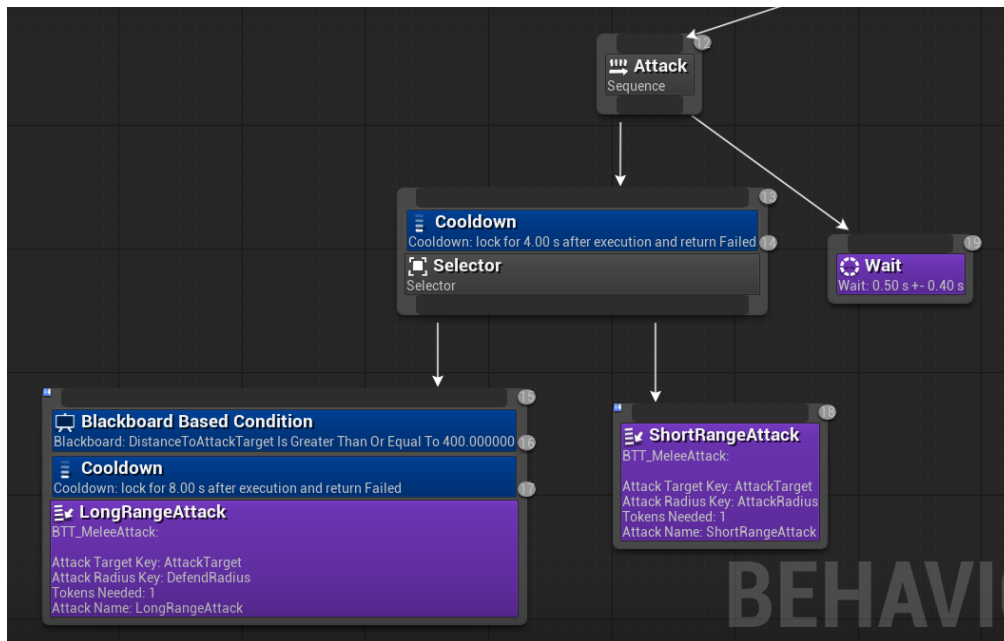


Figura 3-83 *Sequence Attack* final del *BP_EnergyMelee* (Autoría propia).

Como se observa en la Figura 3-83, se ha introducido un *Decorator* de tipo *Blackboard* que establece que, si la distancia al objetivo es mayor o igual a 400 unidades, el enemigo usará *LongRangeAttack*, mientras que, si es menor, ejecutará *ShortRangeAttack*. Para hacer el comportamiento más realista, se ha añadido un *Cooldown*, evitando que el enemigo ataque constantemente con *LongRangeAttack* si la distancia se mantiene por encima de 400 unidades.

La función del ataque gestiona este proceso dentro de la *BTT_MeleeAttack*. El proceso comienza, como se observa en la Figura 3-84, con la obtención del *Attack Target Key* desde el *Blackboard* mediante *Get Blackboard Value as Actor*, lo que permite identificar el actor que el enemigo debe atacar. Luego, se llama a *AttackStarts*, función ubicada en *BPI_Energy_AI*, que inicia el ataque y evalúa si el enemigo cumple con las condiciones necesarias para ejecutarlo. Si la condición es *True*, el ataque procede, en caso contrario, se finaliza con *Finish Execute* para evitar ataques en circunstancias no válidas. Antes de atacar, el enemigo ajusta su velocidad con *SetMovementSpeed*, configurándola en *Sprinting*, lo que le

permite desplazarse de manera agresiva hacia su objetivo. Además, se usa *Clear Focus* para eliminar el enfoque.

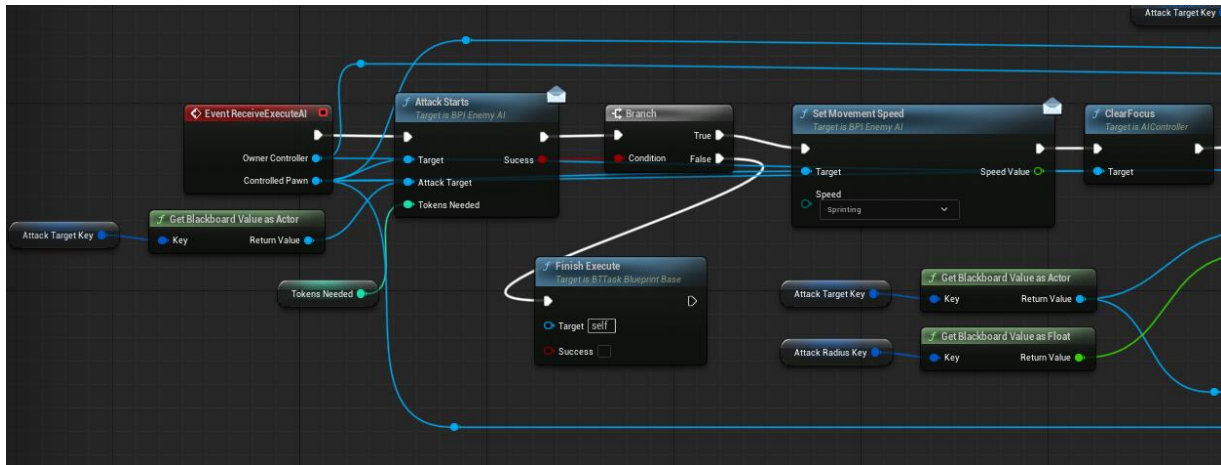


Figura 3-84 Primera parte de *BTT_MeleeAttack* en *BP_EnergyMelee* (Autoría propia).

Para acercarse al jugador, el enemigo utiliza *AI MoveTo*, el cual mueve al personaje hasta su *AttackTarget* con un *AttackRadius*. Si el movimiento se completa con éxito, el flujo avanza al siguiente paso; si falla, se cancela la ejecución del ataque. En esta parte, el sistema evalúa la distancia entre el enemigo y su objetivo. Lo siguiente es centrar la vista, con *Set focus*, al *AttackTarget*, y llamar al *BP_EnergyMelee* a través de un *cast*. Esta parte de la lógica se muestra a continuación en la Figura 3-85:

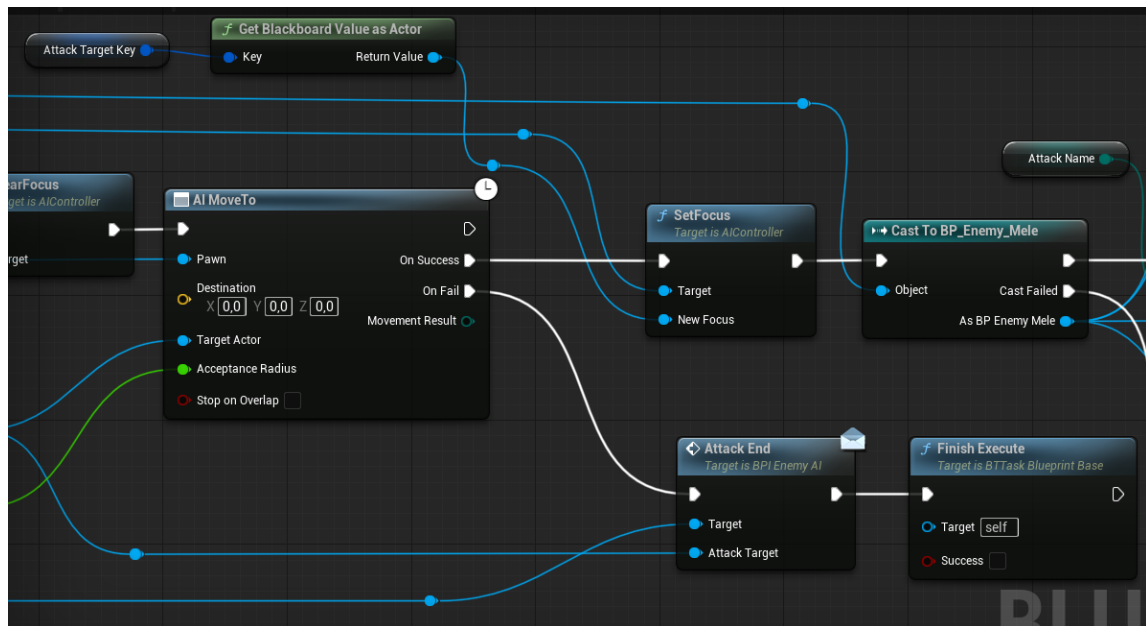


Figura 3-85 Segunda parte de *BTT_MeleeAttack* en *BP_EnergyMelee* (Autoría propia).

A través del nodo *Switch on E_Melee_Attack* Figura 3-86, se decide si se usa el *KnifeShortAttack* o el *KnifeLongAttack*. Este selector está basado en la clave *Distance to Target Key* en la BB, estableciendo que, si la distancia es mayor o igual a 400 unidades, se ejecuta *KnifeLongAttack*, mientras que, si es menor, se activa *KnifeShortAttack*. Tras ejecutar el ataque, el sistema maneja la finalización mediante *Bind Event to On Attack End*, el cual vincula la finalización de la animación de ataque a la función *FinishOnAttackEnd*, garantizando que el enemigo no quede atascado en la animación y pueda continuar con otras acciones dentro del BT. Con estas mejoras, el *BP_EnergyMelee* ahora elige su ataque según la distancia, persigue de manera más eficiente y reacciona al perder de vista al jugador. Su integración en el BT optimiza la toma de decisiones y evita patrones repetitivos, logrando el modo de combate esperado.

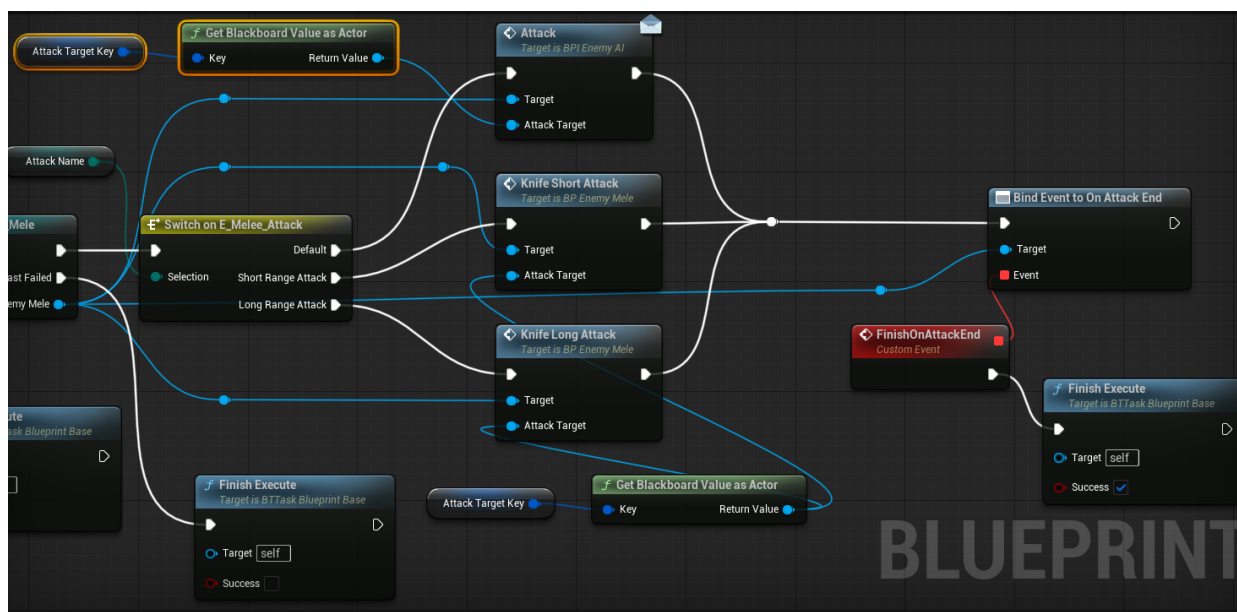


Figura 3-86 Tercera parte de BTT_MeleeAttack en BP_EnemyMelee (Autoría propia).

Con estas mejoras, el *BP_EnemyMelee* ahora elige su ataque según la distancia, persigue de manera más eficiente y reacciona al perder de vista al jugador. Su integración en el BT optimiza la toma de decisiones y evita patrones repetitivos, logrando el modo de combate esperado.

3.3.7.1 BP_EnemyRanged

Función general: El *BP_RangedEnemy* (Figura 3-87), en contraste con el enemigo cuerpo a cuerpo, busca mantenerse a una distancia considerable del jugador y atacarlo mediante disparos. Para mejorar su comportamiento y hacerlo más dinámico, contará con funciones adicionales que le permitirán buscar cobertura cuando sea necesario y curarse si su vida alcanza un nivel crítico, garantizando así una mayor supervivencia en combate.



Figura 3-87 BT final del BP_EnemyBase (Autoría propia).

Siguiendo el mismo procedimiento aplicado en la Figura 3-32, al introducir el *BP_RangedEnemy*, que representa un combatiente armado con un AK-47, ha sido necesario implementar un nuevo conjunto de *anim*s, ya que sus movimientos y posturas difieren del resto de los enemigos. Para ello, se ha creado *BS_Jog* (Figura 3-88), similar a los previamente desarrollados, con la diferencia de que la variable *Speed* se ha fragmentado en tres valores, permitiendo una transición más fluida entre *Idle*, *Walking* y *Sprinting*.

La variable *Direction* mantiene la misma lógica que la mostrada en la Figura 3-32, asegurando que el personaje reaccione correctamente a los cambios de dirección.

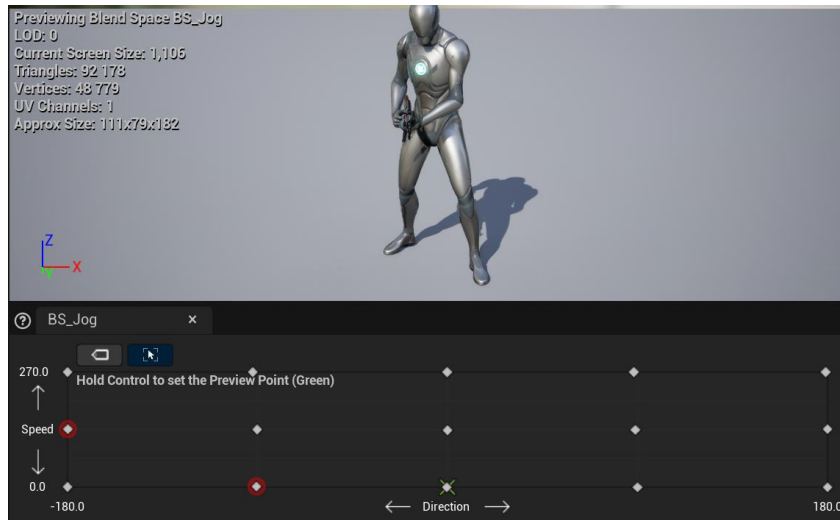


Figura 3-88 BS_Jog (Autoría propia).

El procedimiento para acoplar el arma al *BP_RangedEnemy* sigue la misma lógica que la utilizada en *BP_EnergyBase* con el machete. Se ha creado un *Socket* en la mano del personaje para posicionar correctamente el arma a través de *Spawn Actor*. Posteriormente, se ha usado *Attach Actor to Component* para vincularla al *Socket* correspondiente, asegurando que el arma se mantenga unida al personaje durante la animación. La lógica detallada de este proceso se muestra en la Figura 3-89, donde se observa cómo se gestiona la asignación del arma dentro del *Event Graph*.

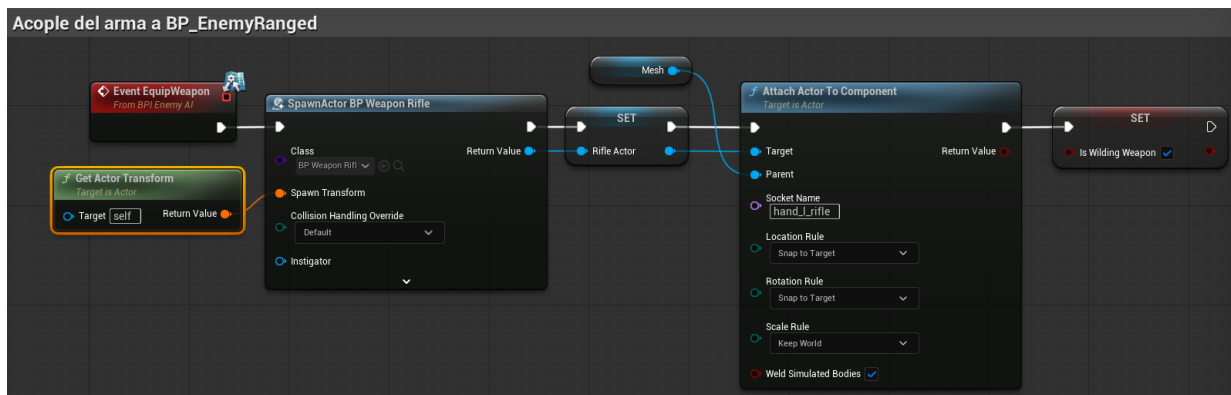


Figura 3-89 Acople del arma en *BP_EnergyRanged* (Autoría propia).

Como se observa en la Figura 3-89, la lógica de equipamiento del arma parte del nodo *EquipWeapon*, sustituyendo la función individual *WieldKnife* (propia del *BP_EnergyMelee*) que anteriormente se ha usado en *BP_EnergyBase*. Debido a que en este caso se pueden manejar diferentes armas y es necesario contar con una función adaptable, se ha creado *EquipWeapon*, que permite acoplar cualquier arma al *Socket* correspondiente. Dado que esta función es común para cualquier tipo de enemigo, se ha implementado dentro de *BPI_EnergyAI*, como se muestra en la Figura 3-17. Como se puede ver en la Figura 3-15, la función ya aparecía bajo el nombre *EquipWeapon*, aunque en su momento fue denominada *EquipKnife*. Además, en la Figura 3-89, se identifican dos nodos clave: uno que realiza un *Set* del arma y otro que, al final del flujo, activa un *bool* indicando que el arma ha sido equipada correctamente.

A diferencia de *BP_EnergyMelee* o *BP_EnergyBase*, donde era necesario verificar si el arma estaba equipada, en el caso de *BP_EnergyRanged* se ha decidido prescindir de esta comprobación. Para garantizar que el enemigo cuente con su arma desde el inicio, dentro del *Event Graph* de

BP_EnergyRanged, se llama directamente a la función *EquipWeapon* desde el *BeginPlay*, asegurando que el arma esté lista desde el primer momento de la simulación.

La creación del *Montage* para el *BP_RangedEnemy* (Figura 3-90) varía ligeramente respecto a los anteriores, ya que se han añadido *Notifies* específicos para mejorar la inmersión y la funcionalidad del disparo. En concreto, se han implementado tres *Notifies*: uno para generar un efecto visual al disparar (*P_SparrowPrimaryMuzzleFlash*) el cual se ha colocado en la bocacha del rifle, otro para reproducir el sonido del disparo (*sfx_Weapon_AutoRifle_AutoClick*) y un tercero que marca el momento exacto del disparo (*Fire*), permitiendo posteriormente aplicar daño si el proyectil impacta en un objetivo. La mayor parte de efectos se han sacado de *FAB*, del paquete *ParagonSparrow*.

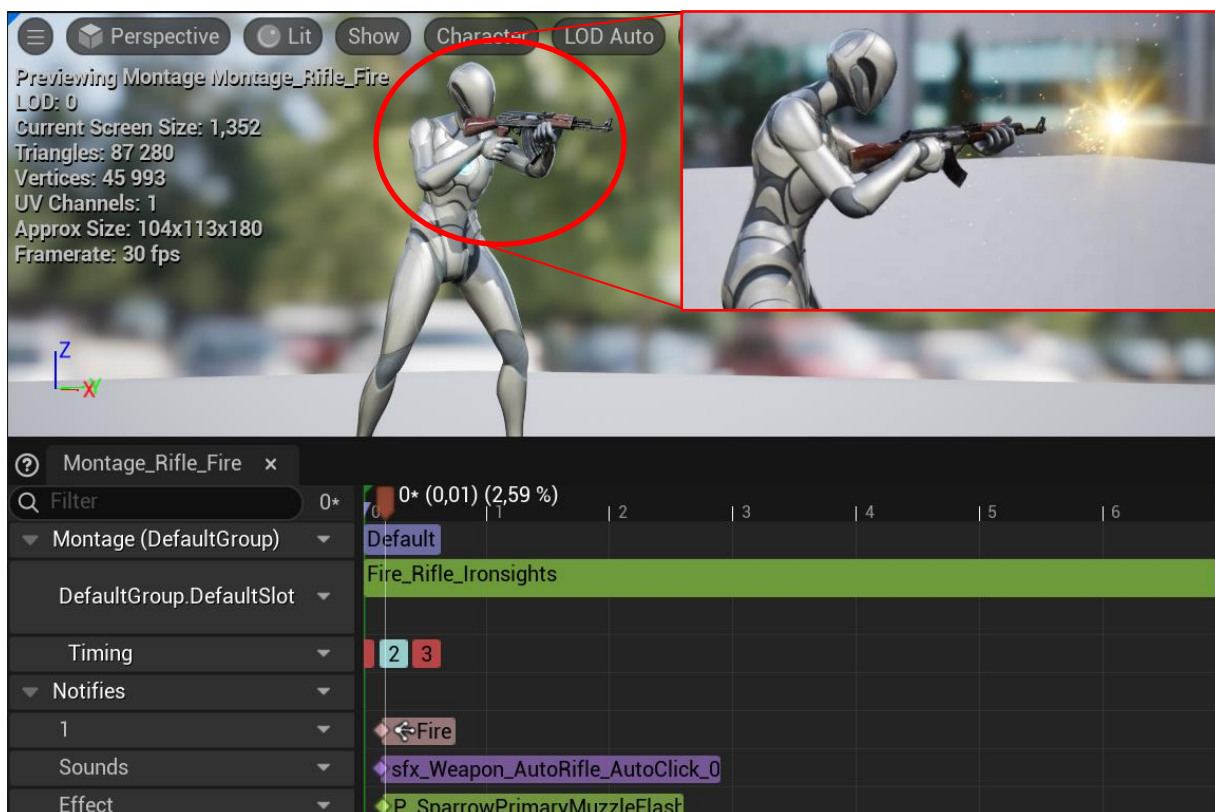


Figura 3-90 Acople del arma en *BP_EnergyRanged* (Autoría propia).

Dentro del *ABP_EnergyRanged*, se ha realizado una copia del *ABP_EnergyBase*, manteniendo la estructura, pero adaptándola al nuevo tipo de enemigo. En esta versión, los estados de *Idle* y *Walk/Run* han sido modificados con respecto a las animaciones utilizadas en las Figura 3-35 y Figura 3-36. Para el estado *Idle*, se ha asignado exclusivamente la animación *Rifle_Ironsight_Idle* del paquete *AnimStarterPack*, mientras que para *Walk/Run*, se ha reemplazado el sistema anterior por el BS creado específicamente para este personaje.

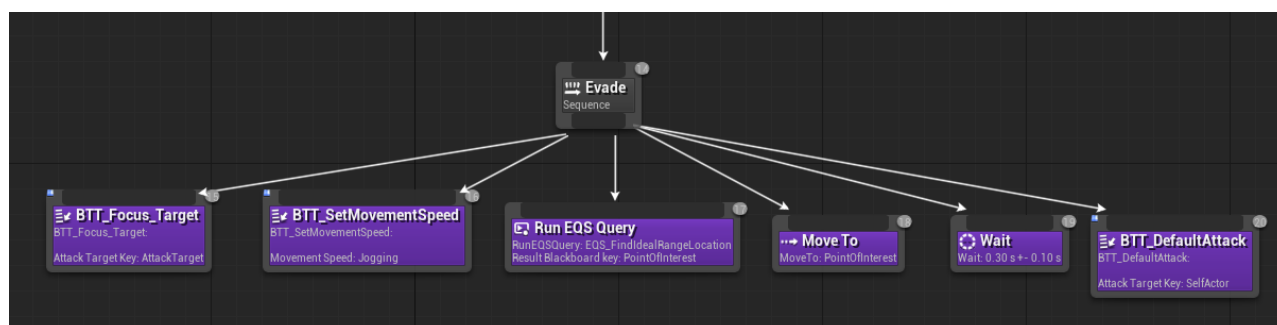


Figura 3-91 Secuencia *Evade* (Autoría propia).

Se ha procedido en este punto a la creación de su BT. Para ello, se ha copiado el *BT_EnemyBase* de la Figura 3-61 y se ha comenzado a editar su estado de ataque de manera individual. El primer paso ha sido definir claramente qué acciones debe realizar este personaje. Su principal objetivo es, al detectar al *BP_Player*, moverse hacia él, encontrar un radio adecuado (*IdealRange*) y seguro desde donde disparar. Para lograr esto, se ha creado una secuencia denominada *Evade*.

En lo que respecta a la parte *Evade* de la secuencia, su lógica (Figura 3-91) reutiliza nodos ya implementados en anteriores configuraciones. La única diferencia específica para este enemigo es que se han establecido los valores de *AttackRadius* en 500 y *DefendRadius* en 600, configurados desde la función *GetIdealRange* dentro de *BP_EnemyRanged*. Esto permite que el enemigo mantenga una distancia óptima para disparar mientras evita estar demasiado expuesto en el combate.

El EQS utilizado en *Run EQS Query* es diferente, ya que se ha creado uno nuevo específicamente para que el enemigo adopte posiciones aleatorias dentro de un rango ideal. Este EQS se ha denominado *EQS_FindIdealRange* (Figura 3-92) y su lógica consiste en generar una cuadrícula de puntos alrededor del *EQS_ContextAttackTarget* con el propósito de encontrar una ubicación óptima para que el enemigo se desplace de manera estratégica. Primero, descarta los puntos inalcanzables, garantizando que solo se consideren aquellos con un camino válido. Luego, evalúa la visibilidad del objetivo, priorizando las posiciones que mantienen una línea de visión clara hacia el *EQS_ContextAttackTarget*. Además, favorece las ubicaciones más alejadas del enemigo para evitar que se acerque demasiado. Finalmente, establece una distancia mínima para validar los puntos seleccionados, asegurando que el enemigo mantenga un rango estratégico de combate.

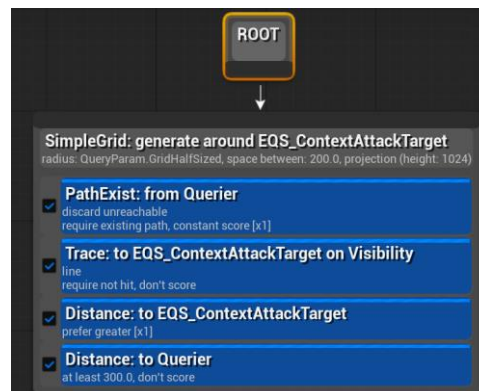


Figura 3-92 *EQS_FindIdealRange* (Autoría propia).

Una vez finalizada la parte *Evade* del *AttackingState*, ha sido necesario programar la mecánica de ataque. Para ello, se ha comenzado implementando el nodo *Attack* en el *Event Graph* del *BP_EnemyRanged*. La primera parte de la función, partiendo del evento *Attack*, ejecuta el *Montage*. Desde ahí, el nodo *Switch on Name* permite definir que el *Notify Fire* es el punto exacto donde debe activarse la lógica que realizará el disparo. Asimismo, como se muestra en la Figura 3-93, si el *Montage* se completa o se interrumpe, se llama a *Attack End* para indicar que la acción ha finalizado.

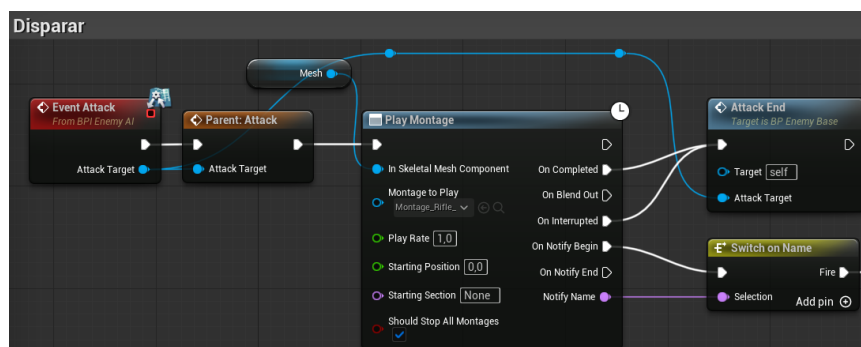


Figura 3-93 Primera parte de *Attack* en *BP_EnemyRanged* (Autoría propia).

Posteriormente como se ve en la Figura 3-93, se ejecuta un *Line Trace by Channel* para visualizar la trayectoria del disparo. Como punto de inicio, se utiliza la ubicación del rifle, y como punto de finalización, se obtiene la posición del *AttackTarget*. Para esto, se realiza un *Cast* dentro del *AIC_EnergyBase*, ya que es ahí donde la variable está almacenada. Para finalizar, se ejecuta la misma lógica presente al final de la Figura 3-55, donde se utiliza la función *TakeDamage* junto con *Make E_DamageInfo*. Si el *Branch* colocado antes de estas dos funciones devuelve *True*, indicando que el *Line Trace* ha impactado correctamente en un objetivo, realizando daño (*Projectile Damage*) al oponente.

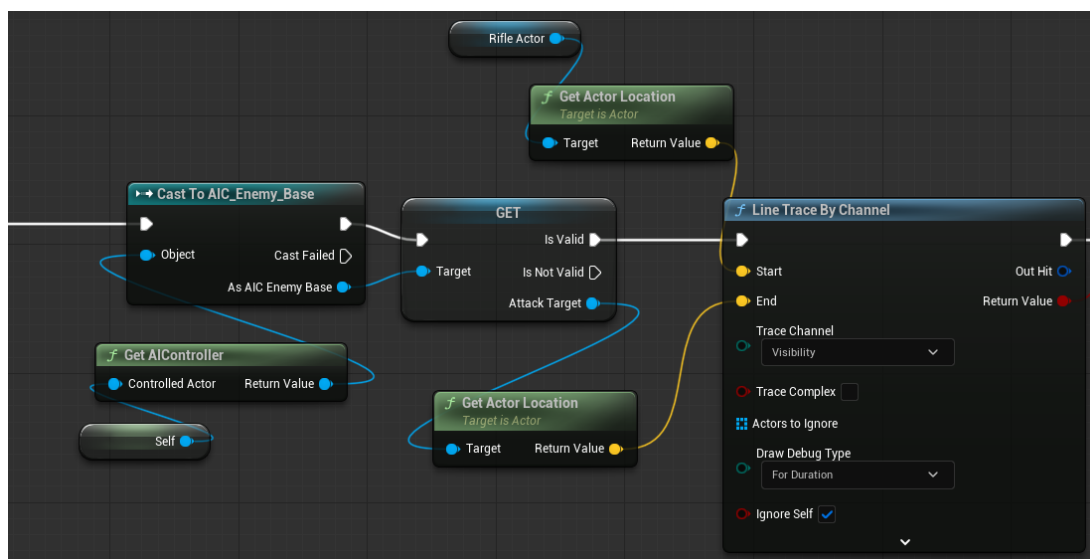


Figura 3-94 Segunda parte de *Attack* en *BP_EnergyRanged* (Autoría propia).

Una vez se ha finalizado esta acción, el enemigo ejecuta correctamente su comportamiento, pero incluso si el jugador se encuentra detrás de un muro, sigue disparando. Para solucionar esto, se ha creado un *Decorator* que verifica si el enemigo realmente tiene línea de visión con su objetivo. Este *Decorator*, llamado *BTD_CanSeeTarget*, tiene la lógica sencilla que muestra la Figura 3-95. Su función es comprobar si el enemigo puede ver al *AttackTarget* almacenándolo en la BB. Para ello, primero obtiene la ubicación tanto del enemigo como del objetivo y luego ejecuta un *Line Trace By Channel*, que lanza un rayo desde la posición del enemigo hasta la del objetivo. Si el *Line Trace* detecta un obstáculo en el camino, el resultado será *False*, lo que indica que el enemigo no tiene una línea de visión clara. En caso contrario, si no hay interrupciones, devuelve *True*, confirmando que el enemigo puede ver a su objetivo sin obstáculos.

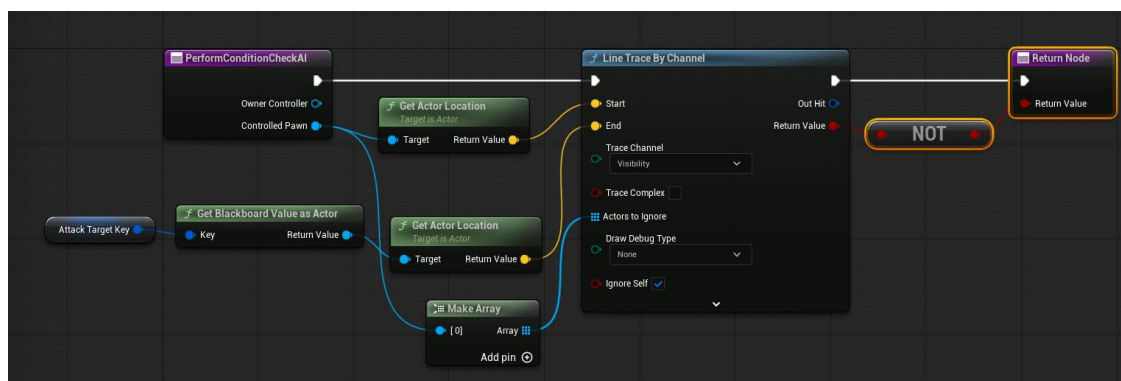


Figura 3-95 *BTB_CanSeeTarget* (Autoría propia).

Este *Decorator* se ha ubicado en la secuencia *Evade*, quedando el *BT_EnergyRanged* como se muestra en la Figura 3-91. Sin embargo, con esta lógica, si el jugador se mueve y se esconde, o si el

enemigo pierde de vista al *AttackTarget*, este quedaría inactivo sin una acción clara a seguir. Para abordar este escenario, se ha creado una nueva ramificación dentro del *AttackingState*, específicamente una secuencia llamada *MoveWhenTargetIsNotVisible*. Esta secuencia tiene como objetivo principal que el enemigo se desplace hacia un *PointOfInterest*, es decir, una ubicación donde tenga mayores probabilidades de recuperar la línea de visión con su objetivo. Para añadir más posibilidades al escenario, evitando que mueran demasiado rápido, se ha implementado una función que permite al enemigo, al llegar a cierto nivel de vida, esconderse en un sitio cercano, curarse y luego volver a atacar. Esta lógica se ha integrado en el *BT_EnemyRanged* a través de una *Sequence*. Antes de entrar en detalle sobre los nodos de esta ramificación, es necesario crear dos elementos fundamentales: en primer lugar, una función para curarse y, en segundo, un *Decorator* que verifique si la vida está por debajo de un nivel determinado.

Para la función de curación, se ha creado una *Task* llamada *BTT_Heal* (Figura 3-96), cuya lógica se muestra en la Figura 3-96, y consiste en lo siguiente: al ejecutarse, obtiene la *MaxHealth* del enemigo a través de la función *GetMaxHealth*. Luego, multiplica este valor por un porcentaje específico almacenado en *HealthPercentage*, calculando así la cantidad de vida que debe recuperar. Posteriormente, llama a la función *Heal*, que incrementa la salud del enemigo en función del valor calculado. Finalmente, ejecuta *Finish Execute* con el parámetro *Success* activado, indicando que la tarea ha finalizado correctamente. Es importante destacar que la variable *HealthPercentage* está marcada como *Instance Editable*, permitiendo ajustar la cantidad de curación directamente desde el BT.

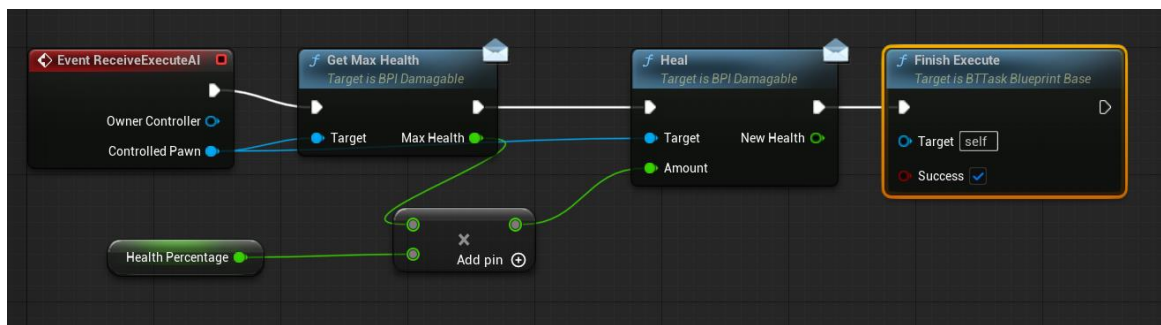


Figura 3-96 BTT_Heal (Autoría propia).

Una vez finalizada la *Task* de curación, se ha creado *BTD_IsHealthBelowWhatItShould* (Figura 3-97), la cual obtiene la *MaxHealth* del enemigo mediante *GetMaxHealth* y su salud actual con *GetCurrentHealth*. Luego, divide la salud actual entre la salud máxima para calcular el porcentaje de vida restante. Este valor se compara con un umbral definido en la variable *BelowWhatItShouldBe*. Si el porcentaje de vida es menor o igual a este umbral, la función devuelve *True* en el *Return Node*, permitiendo que el BT ejecute la secuencia de curación. En caso contrario, devuelve *False*, evitando que el comportamiento asociado se active.

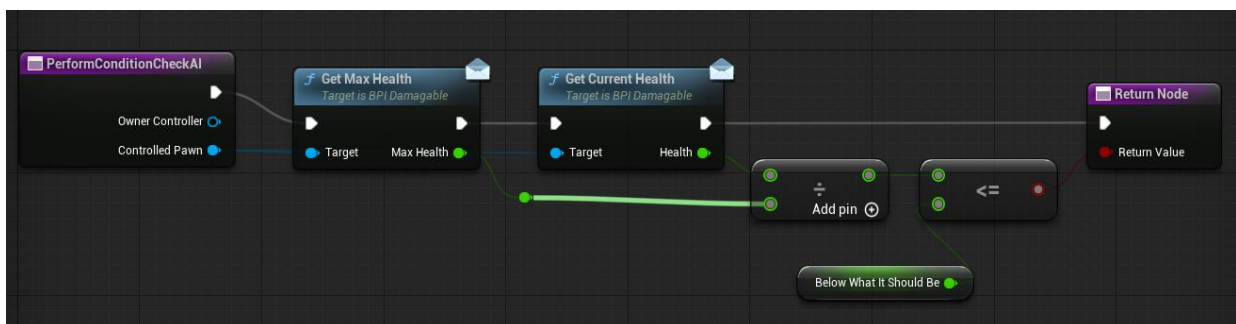


Figura 3-97 BTD_IsHealthBelowWhatItShould (Autoría propia).

De esta manera, el BT que implementa estas dos nuevas acciones queda estructurado como se muestra en la Figura 3-98. Esta nueva secuencia, al igual que *Evade* y *MoveWhenTargetIsNotVisible*,

parte del nodo *AttackingState*, el cual se activa cuando el estado del enemigo está en *SetStateAsAttacking*.

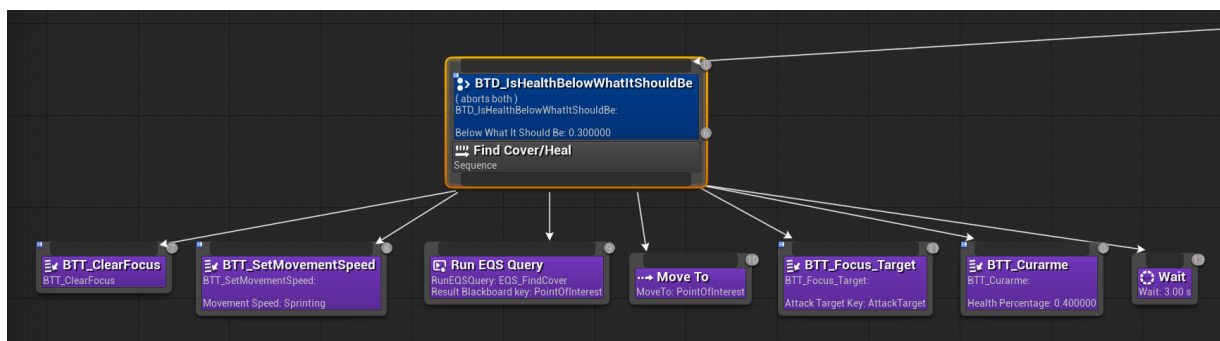


Figura 3-98 Ramificación de buscar refugio y curarse (Autoría propia).

Para *BP_EnergyRanged*, se ha creado un *attack* personalizado en su *Behavior Tree*. Su lógica en el *Event Graph* es idéntica a la de *BP_EnergyMelee*, pero solo ejecuta un *Montage*. En el BT, se ha implementado *BTT_RangedAttack*, que sigue la misma lógica que *BTT_MeleeAttack*, pero sin necesidad de seleccionar entre dos ataques.

Con estas mejoras, el *BP_RangedEnemy* no solo ajusta su posición para mantener un rango óptimo de ataque, sino que también busca cobertura, verifica su línea de visión antes de disparar y prioriza su supervivencia al curarse si su salud es baja.

3.3.7.2 BP_TeamFriend

Función general: *BP_TeamFriend* (Figura 3-99), representa la simulación de tres soldados de infantería de marina, diseñados para replicar el desplazamiento táctico en entornos urbanos. Su función principal es moverse en formación cubriendo distintos sectores, garantizando la seguridad del equipo mientras siguen al líder de la línea. Cada soldado mantiene su posición estratégica y ajusta su movimiento en función de la situación y las órdenes recibidas.

Cuando el líder ordena la entrada en una estructura o zona específica, los soldados ejecutan la maniobra siguiendo los procedimientos establecidos en combate poblacional, asegurando una coordinación precisa y adaptada a las condiciones del entorno.

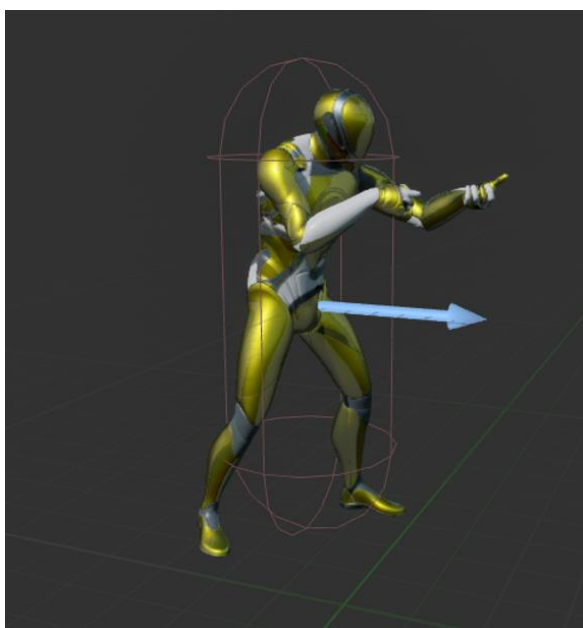


Figura 3-99 BT final del *BP_EnergyBase* (Autoría propia)

Para ello, se ha creado un nuevo *AIController* que establece la lógica del enemigo y guarda las variables necesarias usadas en la BB. El BT de este enemigo se muestra en la Figura 3-100.

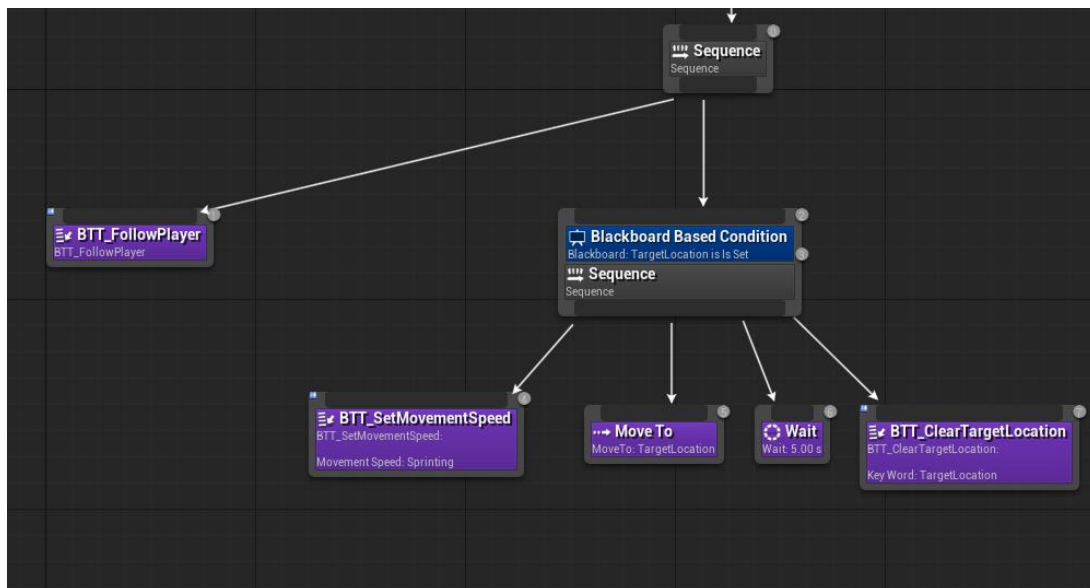


Figura 3-100 BT final del *BP_FriendTeam* (Autoría propia).

En la lógica de *BT_TeamFriend* se desarrolla una secuencia donde primero se ejecuta *BTT_FollowPlayer*, permitiendo que los soldados sigan al *BP_Player* de manera coordinada, tal como se representa en la Figura 2-11. Luego, si *Target Location Is Set*, los soldados se desplazan hacia la posición designada cuando el *BP_Player* pulsa el botón derecho del ratón en cualquier punto del mapa. Esto simula la entrada a una habitación y la toma de posiciones según los procedimientos tácticos establecidos en el trabajo.

Si se quisiera que los soldados atacaran al detectar un enemigo, bastaría con implementar la lógica mostrada en la Figura 3-101, pero dado que el líder es quien toma las decisiones de combate, se ha optado por mantener su estado *Passive*, permitiendo que solo el *BP_Player* realice los disparos cuando sea necesario.

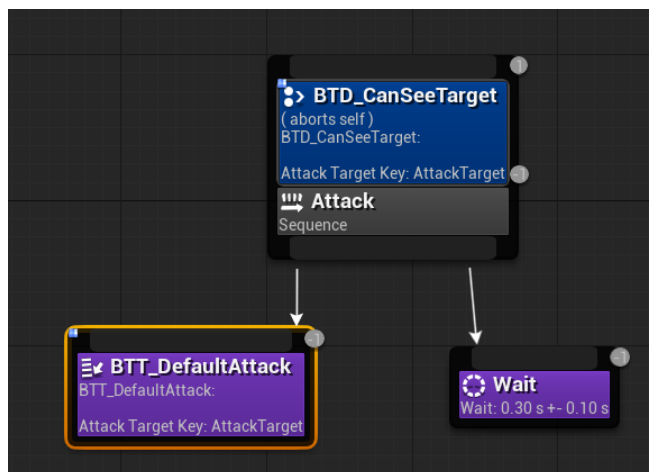


Figura 3-101 Secuencia *Attack* para *BP_FriendTeam* (Autoría propia).

3.3.7.3 *BP_PatrolEnemy*

Función general: *BP_PatrolEnemy* (Figura 3-102) es un enemigo diseñado para recorrer rutas predefinidas dentro del nivel, asegurando vigilancia en áreas específicas. Su comportamiento principal es seguir una secuencia de puntos de patrulla asignados desde el *World Outliner*, lo que le permite

moverse automáticamente por el entorno. Para su correcta ejecución, es necesario asignar en sus detalles la ruta de patrulla deseada, lo que le indicará los puntos a seguir durante su recorrido. Para ello se ha programado *BP_PatrolRoute* (realizado con anterioridad) como base para la lógica de patrullaje, pero es posible crear variaciones y nuevas rutas adaptadas al diseño del mapa. Esto permite asegurar que todos los sectores estén cubiertos estratégicamente, ajustando las rutas según la distribución del entorno y las necesidades del nivel.

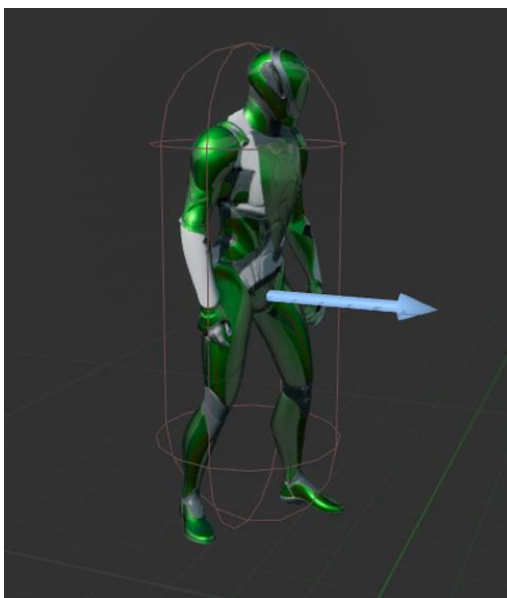


Figura 3-102 *BP_PatrolEnemy* (Autoría propia).

3.3.7.4 *BP_ShooterLongRangedEnemy*

Función general: El *BP_ShooterLongRangedEnemy* (Figura 3-103) está diseñado para atacar al *BP_Player* desde cotas más altas y distancias mayores, aprovechando su arma de alta letalidad para infligir más daño que otros enemigos. Su rol es esencial en escenarios de combate en entornos urbanos, ya que puede posicionarse estratégicamente en puntos elevados y sorprender al jugador con ataques precisos. Esta mecánica obliga al jugador a estar atento a posibles francotiradores y a buscar cobertura constantemente, aumentando la tensión y el realismo del enfrentamiento.

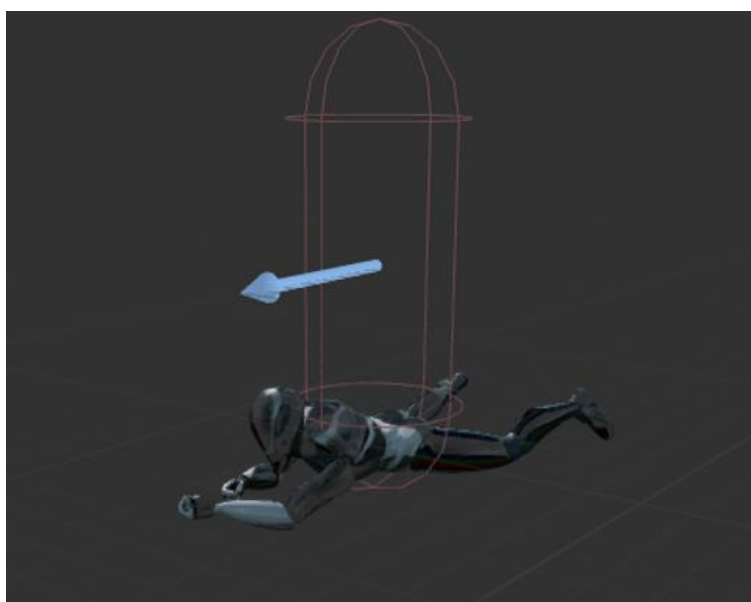


Figura 3-103 *BP_ShooterLongRangedEnemy* (Autoría propia).

El *BP_ShooterLongRangedEnemy* cuenta con un ABP distinto, ya que sus movimientos están limitados a una posición tumbada en el suelo. Debido a esta restricción, se ha creado una nueva animación adaptada a esta postura, asegurando que sus transiciones y comportamientos reflejen correctamente su rol en combate. Siguiendo el mismo procedimiento aplicado en anteriores modificaciones de ABP, se ha ajustado la estructura para que el enemigo se mantenga inmóvil salvo cuando cambia de posición o apunta a su objetivo.

El *BT_ShooterLongRangedEnemy* ha sido simplificado en comparación con el *BT_EnergyRanged*, eliminando ciertas mecánicas de movilidad y evasión para enfocarse en su rol de ataque desde posiciones fijas. Su lógica, representada en la Figura 3-104, se basa en detectar al jugador, ajustar su mira y disparar desde una posición estratégica sin necesidad de desplazarse constantemente.

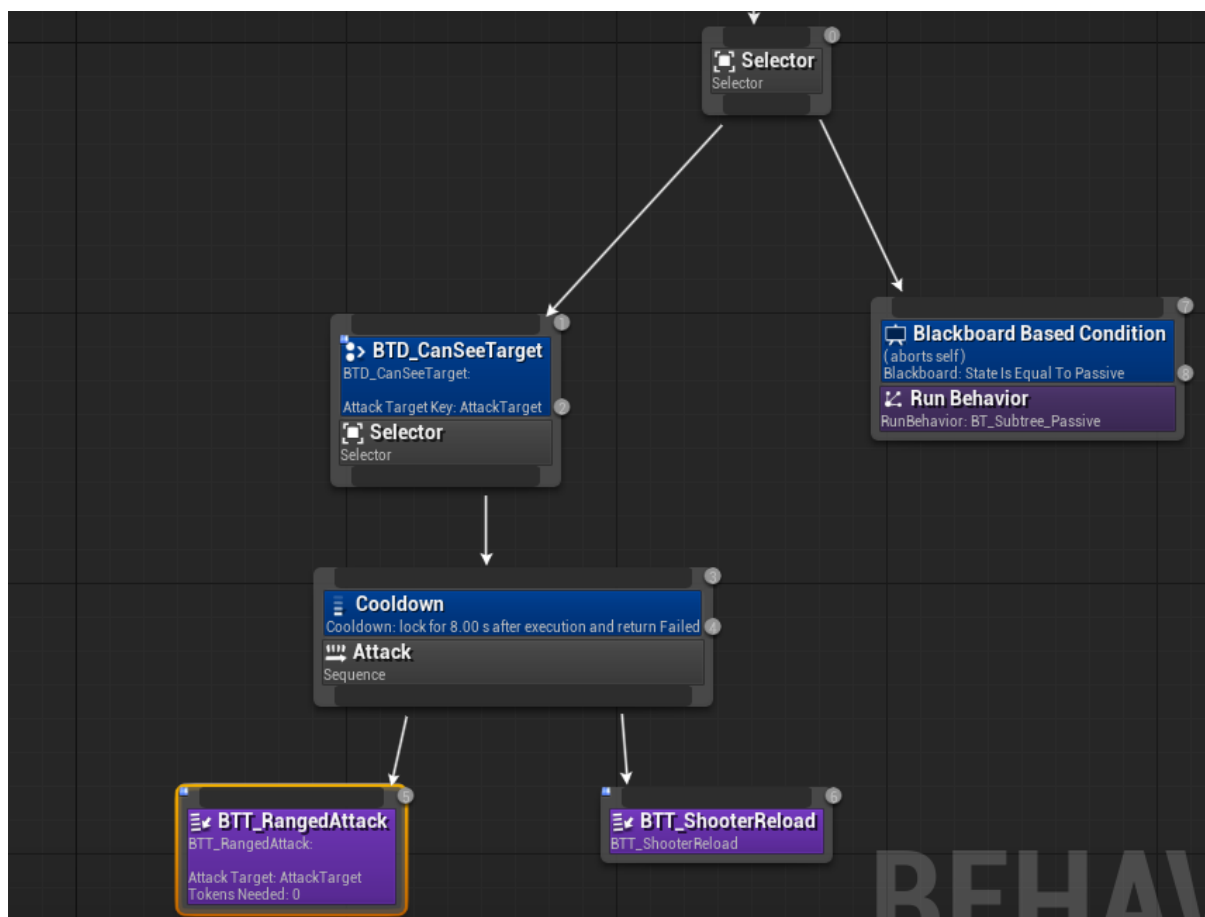


Figura 3-104 *BT_ShooterLongRangeEnemy* (Autoría propia).

Inicia con un *Selector* que primero verifica si el enemigo está en estado *Passive* mediante una *Blackboard Based Condition*, en cuyo caso ejecuta el *BT_Subtree_Passive*. Si no está en modo *Passive*, pasa a evaluar si tiene línea de visión con el *AttackTarget* usando el *BTD_CanSeeTarget*. Si el enemigo puede ver a su objetivo, ejecuta un *Cooldown* de 8 segundos entre disparos para simular el tiempo de espera entre cada ataque, evitando que dispare de manera continua. Dentro de la secuencia de ataque, se ejecuta *BTT_RangedAttack*, que maneja la lógica del disparo, seguido de *BTT_ShooterReload*, encargada de la recarga del arma antes de la siguiente acción a través de un *Montage*.

3.3.7.5 *BP_SurrenderEnemy* y *BP_RunAwayEnemy*

Función general: Cada escenario es un mundo, y las posibilidades y acciones por parte de los enemigos son muy variadas. Por ello, se han programado dos enemigos (Figura 3-105) con comportamientos distintos a los vistos anteriormente: uno que, al detectar al *Player Character*, huye a una ubicación aleatoria del mapa, y otro que se rinde ejecutando un *Montage*.

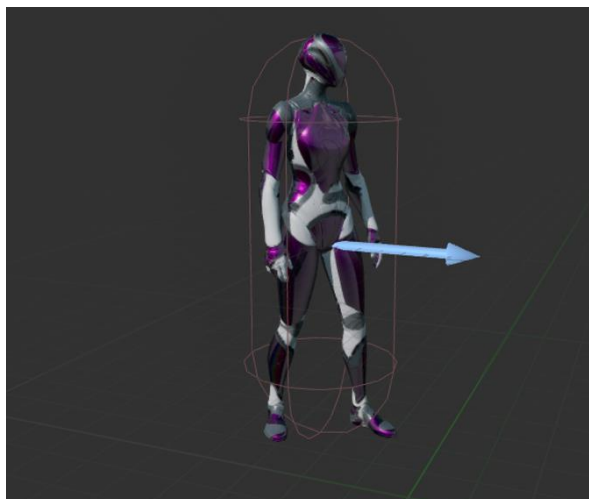


Figura 3-105 Static Mesh de BP_SurrenderEnemy y BP_RunAwayEnemy (Autoría propia).

En el caso de *BP_SurrenderEnemy*, su escape se basa en la lógica expuesta en las Figura 3-106 y Figura 3-107. La ejecución comienza con el evento *On Target Perception Updated*, que detecta cambios en la percepción del enemigo. A continuación, *Break AIStimulus* extrae información del estímulo recibido, como su antigüedad, intensidad y ubicación. Si el estímulo ha sido detectado con éxito, un *Branch* valida la condición de huida.

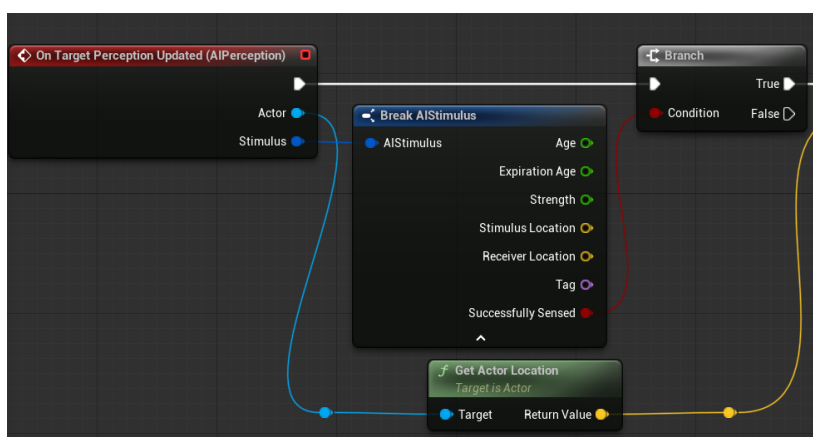


Figura 3-106 Primera parte de la lógica de escapar (Autoría propia).

La figura Figura 3-107, muestra que, en caso de ser *True*, se obtiene la posición actual del enemigo con *Get Actor Location* y se genera una nueva ubicación aleatoria en un radio de 5000 unidades mediante *Get Random Location in Navigable Radius*. Finalmente, *AI MoveTo* dirige al enemigo a la posición generada, asegurando su huida. Como indicador de depuración, se imprime el mensaje “Estoy huyendo, yo no he hecho nada” confirmando que el enemigo está escapando.

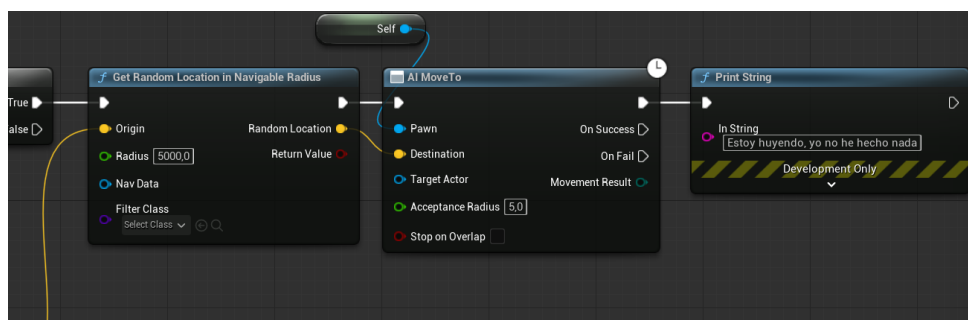


Figura 3-107 Segunda parte de la lógica de escapar (Autoría propia).

Por otro lado, en el caso de *BP_SurrenderEnemy*, su lógica es prácticamente idéntica, con la única diferencia de que, tras el *Branch*, en lugar de generar una ubicación aleatoria con *Get Random Location in Navigable Radius*, ejecuta un *Montage*, reproduciendo así una animación de rendición en lugar de huir.

El enemigo de interés al que se pretende capturar, denominado *BP_TargetEnemy*, ejecuta una secuencia en la que, al detectar a *BP_Player*, se rinde. A través de un BT, sigue al jugador durante un tiempo determinado, simulando su captura. Luego, permanece en la posición de *BP_Player*, y finalmente, el juego se detiene mostrando un widget con el mensaje "Objetivo Capturado".

3.3.8 Combate y comunicación del enemigo

Para hacer más realista el comportamiento de los enemigos en combate, se ha implementado un sistema de comunicación que coordina sus ataques. Sin este sistema, múltiples enemigos atacarían simultáneamente, lo que podría generar colisiones entre ellos y situaciones poco creíbles. Con esta nueva lógica, el número de enemigos que atacan a la vez se ajusta según las preferencias del *BP_Player*, permitiendo, por ejemplo, que solo uno o dos enemigos ataquen simultáneamente. El sistema se basa en el uso de *tokens*. El *BP_Player* tiene un número asignado de estos *tokens*, y los enemigos los solicitan antes de atacar. De esta manera, si hay *tokens* disponibles, el enemigo podrá iniciar su ataque; de lo contrario, deberá esperar su turno. Para gestionar este sistema, se han implementado dos funciones dentro de *BPC_DamageSystem*: *ReserveAttackToken* y *ReturnAttackToken*.

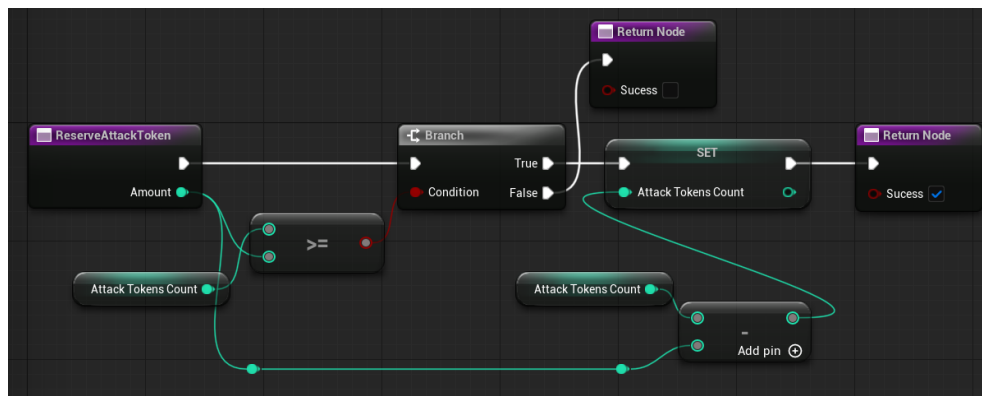


Figura 3-108 *ReserveAttackToken* (Autoría propia).

La función *ReserveAttackToken* (Figura 3-108) regula cuántos enemigos pueden atacar simultáneamente. Recibe un valor (*Amount*) y verifica si hay suficientes *tokens* disponibles comparando *AttackTokensCount* con la cantidad solicitada. Si hay suficientes, resta los *tokens* utilizados y devuelve *Success*; en caso contrario, finaliza sin cambios y retorna *Failure*. Esto evita que todos los enemigos ataquen al mismo tiempo y mantiene un combate más estructurado. Por otro lado, *ReturnAttackToken* (Figura 3-109) restaura los *tokens* una vez que un enemigo finaliza su ataque. Toma la cantidad a devolver, la suma al contador de *AttackTokensCount* y actualiza su valor. De esta manera, otros enemigos pueden acceder a los *tokens* liberados y atacar cuando sea su turno, asegurando un combate equilibrado.

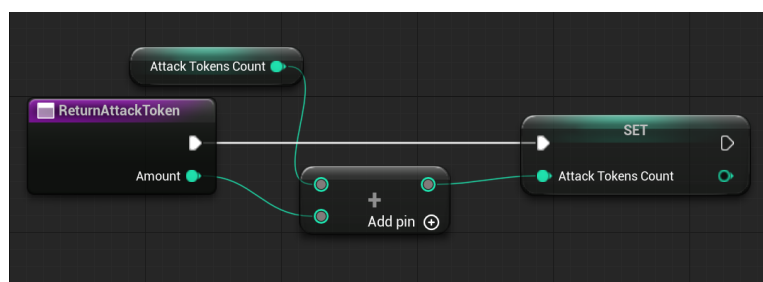


Figura 3-109 *ReturnAttackToken* (Autoría propia).

Desde el *Construction Script* de *BP_Player*, se inicializa la variable *AttackTokensCount* con un valor de 1, lo que implica que, por defecto, solo un enemigo podrá atacarlo a la vez. Este sistema se integra con las funciones *AttackStarts*, *AttackEnd* y *StoreAttackToken*, las cuales fueron diseñadas precisamente para gestionar la reserva y devolución de tokens en función del inicio y fin de cada ataque

Cada vez que un enemigo inicia un ataque, como se observa en la Figura 3-84, el nodo *AttackStarts* maneja la variable *TokensNeeded*, que almacena la cantidad de tokens requeridos por el ataque individual del enemigo. La lógica de almacenamiento y gestión de *tokens* ha sido implementada en *BP_EnergyBase*, desde donde es heredada por los demás enemigos.

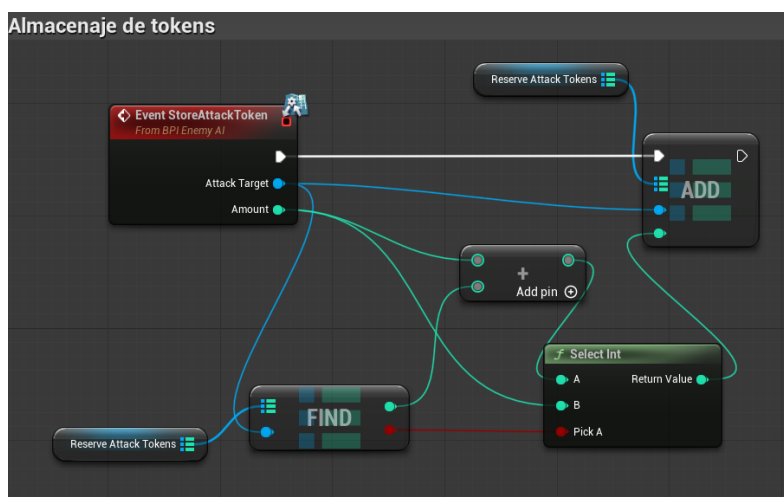


Figura 3-110 Función *StoreAttackToken* (Autoría propia).

Dentro del *Event Graph*, hay tres funciones que llevan la lógica general: la función *StoreAttackToken* (Figura 3-110) que evita que múltiples enemigos sobrecarguen un mismo objetivo. Si el *AttackTarget* ya tiene tokens asignados, los suma; de lo contrario, crea una nueva entrada en la lista *ReserveAttackTokens*. A continuación, la función *AttackStarts* solicita los tokens necesarios y *ReserveAttackToken* verifica si hay suficientes disponibles. Si la reserva es exitosa, el ataque continúa; de lo contrario, el flujo se interrumpe con un *Return Node* en falso. En caso de éxito, *StoreAttackToken* almacena la cantidad asignada y se actualiza *TokensUsedInCurrentAttack*, asegurando que solo los enemigos con *tokens* disponibles puedan atacar.

Cuando un ataque finaliza, el evento *AttackEnd* (Figura 3-111) gestiona la devolución de los *tokens*. Primero, *ReturnAttackToken* devuelve la cantidad almacenada al sistema de combate, permitiendo que otros enemigos puedan utilizarlos. Luego, *StoreAttackToken* actualiza el valor dentro del enemigo, reflejando que el ataque ha concluido. Finalmente, la variable *IsAttacking* se desactiva (Esta se activa en *Attack*) y se ejecuta *Call On Attack End*, indicando que la acción ha finalizado.



Figura 3-111 Función *AttackEnd* (Autoría propia).

Además, el evento *ReturnAttackToken*, integrado en el *BPC_DamageSystem*, centraliza la devolución de *tokens*, garantizando que el sistema de combate recupere los tokens utilizados una vez que el ataque termina.

3.3.9 Creación de equipos

El objetivo de este apartado es la creación de equipos, una funcionalidad clave que permite generar una amplia variedad de situaciones dentro del escenario. Esta mecánica ofrece la posibilidad de asignar diferentes equipos a civiles y enemigos, permitiendo que estos últimos los ataquen de manera autónoma. Del mismo modo, los enemigos pueden pertenecer al mismo equipo y enfrentarse al *BP_Player* en momentos inesperados.

Este sistema posibilita la creación de escenarios dinámicos, como conflictos callejeros donde varios enemigos se enfrentan entre sí, añadiendo una capa de imprevisibilidad a la simulación. La flexibilidad de esta función permite que los personajes, con sus respectivas mecánicas, sean combinados de diversas maneras, generando situaciones complejas que desafían la capacidad del jugador para evaluar y e implementar las ROE ante los eventos en tiempo real.

Lo primero que se ha realizado para estructurar el sistema ha sido crear la función *GetTeamNumber* dentro de *BPI_Damage*. Dado que todos los enemigos son *Child Blueprint Class* de *BP_EnemyBase* y heredan funciones clave como *HitResponse*, *Die Event*, *Tokens* y *NavLinks*, se ha programado esta función tanto en *BP_EnemyBase* como en *BP_Player*. Para permitir la configuración manual de equipos, la variable *TeamNumber* (de tipo *Integer* y creada en *BP_enemyBase*) se ha establecido como *Instance Editable*, permitiendo modificarla desde el *World Outliner* en cada enemigo.

Para asignar y verificar equipos, dentro de *AIC_EnemyBase* se ha creado la función *OnSameTeam* (Figura 3-112). Esta función toma un actor como referencia (*Other Actor*) y obtiene su número de equipo mediante *GetTeamNumber*, que está implementada en *BPI_Damage*. Simultáneamente, recupera el *Pawn* controlado por el *AIController* y extrae su número de equipo utilizando la misma función. Luego, compara ambos valores y devuelve un resultado booleano en *Return Node*: *true* si los actores están en el mismo equipo y *false* en caso contrario. Esta lógica es esencial para evitar ataques entre aliados y permitir interacciones diferenciadas según la afiliación de cada personaje.

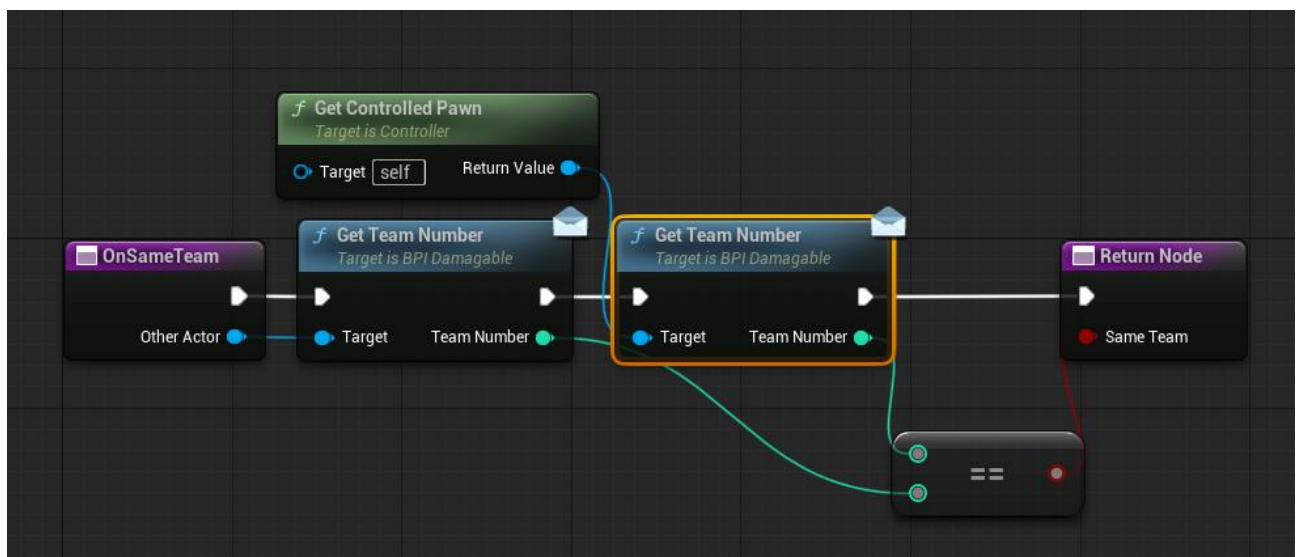


Figura 3-112 Función *OnSameTeam* (Autoría propia).

Dentro de *HandleSenseSight*, se han realizado varias modificaciones respecto a la Figura 3-27. Ahora se incorpora el nodo *OnSameTeam*, que verifica si el actor detectado pertenece al mismo equipo. Si los actores no están en el mismo equipo, el flujo de ejecución continúa normalmente, permitiendo que el enemigo reaccione al objetivo detectado. En caso contrario, si ambos personajes pertenecen al mismo

equipo, la lógica se detiene y no se realiza ninguna acción, evitando así la detección y el ataque entre aliados.

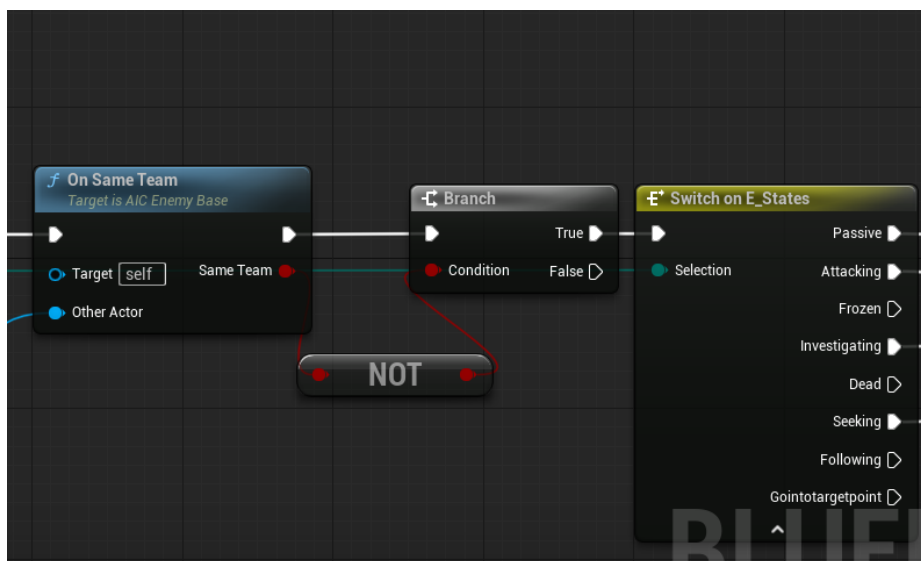


Figura 3-113 Lógica implementada en *HandleSenseSight* (Autoría propia).

En *HandleSenseDamage* también se ha implementado el sistema de equipos, aunque en principio dos personajes del mismo equipo no pueden dañarse entre sí. Sin embargo, si se programase una explosión en el suelo, esta podría afectar a todos los actores dentro del área, incluyendo aliados, lo que haría que el daño fuera negativo. Por otro lado, en *HandleSenseSound* no se ha realizado ninguna modificación, ya que se quiere mantener la lógica de que el enemigo investigue cualquier sonido detectado sin importar su origen. Aun así, si el ruido proviene de un aliado, el enemigo simplemente irá a revisar sin atacarlo, respetando la restricción de equipo establecida en *HandleSenseSight*.

3.4 Integración en un escenario para combate poblacional en VR

Finalmente, una vez terminada la programación de los enemigos (Figura 3-114), se ha procedido a exportar toda la lógica a un escenario de combate poblacional ya diseñado, el cual representa PICOZU, una réplica del entorno real en Figueirido, mostrado en la Figura 2-9.



Figura 3-114 Escenario integrado (Autoría propia).

Los enemigos, civiles y soldados de *BP_Player* se han exportado desde *FAB*, utilizando paquetes que incluyen *meshes* predeterminadas listas para su implementación. Respecto a los civiles, se han implementado dos tipos. El primero de ellos se muestra en la Figura 3-115 a continuación:



Figura 3-115 Civil 1 (Autoría propia).

El otro civil exportado es el que se observa en la Figura 3-116. Ambos civiles pueden adoptar comportamientos pasivos o actuar como combatientes encubiertos dentro del escenario. En el caso del “civil 1”, se utiliza un *child* de *BP_PatrolEnemy*, que sigue su ruta sin tener implementado el *AttackingState*. En el segundo caso (civil 2), se ha empleado el BT de *BP_EnemyRanged*, pero bajo la apariencia de un civil, generando la posibilidad de recrear situaciones impredecibles y complejas para el jugador, quien deberá evaluar con cautela cada encuentro.



Figura 3-116 Civil 2 (Autoría propia).

La *Static Mesh* de *BP_Player* (Figura 3-117) y los soldados de operaciones especiales que lo acompañan (*BP_FriendTeam*) proviene del paquete *ModularSoldierNato*, el cual simula uniformes y equipamientos utilizados por la OTAN, aportando mayor realismo a la simulación.



Figura 3-117 *Mesh* de *BP_Player* y soldados de operaciones especiales (Autoría propia).

Por último, se ha establecido la apariencia de los enemigos expuestos en el apartado 3.3.7, representando combatientes no militares. Esto se debe a que la organización de Redland, que ha tomado el control de este pueblo, está conformada por civiles armados en lugar de un ejército formal. Su objetivo es mantener la vigilancia en los territorios de Greenland ocupados por Redland, impidiendo su recuperación por parte del país afectado.



Figura 3-118 *Mesh* de *BP_EnemyBase* (Autoría propia).

Respecto al armamento utilizado en el juego, se ha implementado para los enemigos el uso de machetes o AK-47. Sin embargo, para aportar un mayor realismo al escenario, se ha implementado el armamento utilizado en España para los soldados de Greenland: el HK G36E y la FN-9. Estas armas han

sido importadas y se muestran a continuación en la Figura 3-119. El armamento ya viene programado con la lógica necesaria para su funcionamiento, incluyendo la mecánica de recogerlas, insertar el cargador, quitar el seguro, gestionar el decrecimiento de munición con cada disparo y ejecutar el proceso de recarga.



Figura 3-119 HK G36E y FN-9 (Autoría propia).

Posteriormente, tanto el escenario como los personajes han sido trasladados a un *template* de RV, adaptando sus mecánicas para una experiencia inmersiva en RV. Una vez migrados los archivos, ha sido necesario realizar algunos ajustes para asegurar el correcto funcionamiento del sistema dentro del nuevo entorno. Hay que destacar que, aunque los BT, BB y *AIControllers* de los enemigos son comunes a cualquier *template*, la forma en la que estos interactúan con el jugador varía. La *AI Perception* ha sido modificada, dado que en RV el *VRPawn* se teletransporta y no genera sonidos al moverse, lo que ha requerido la implementación de nuevas mecánicas de sonido. Respecto al *SenseSight*, en el *ThirdPerson tempalte*, el *AttackTarget* era un elemento claramente visible, representado por el *Static Mesh* de Manny; sin embargo, en RV, los enemigos solo pueden ver las manos del jugador, ya que es lo único que tiene visible el BP de *VRPawn* al usar las gafas de RV. Para solucionar esto, se ha añadido un *Box Collision* que permite a los enemigos detectar con mayor precisión la presencia del jugador.

Además, se ha agregado al arma (*BP_Pistol*) la mecánica de hacer daño expuesta en la Figura 3-55, ya que en este contexto no es necesario pulsar un botón específico para atacar si no apretar el gatillo. La lógica en RV, no se gestiona mediante botones, ratón o teclado, sino que se gestiona mediante *Input Actions* e *Input Mappings*, adaptando los controles a los mandos utilizados en la simulación. Todo lo que estaba programado en *BP_Player*, como la gestión de *tokens*, la salud máxima, la salud actual y la mecánica de muerte, se han trasladado al *BP_VRPawn*, integrándole las interfaces, anteriormente implementadas en *BP_Player*, necesarias para su correcto funcionamiento.

La implementación en RV permite una interacción mucho más inmersiva para el *BP_Player*, ahora convertido en *VRPawn*. Gracias a esta adaptación, el jugador puede agacharse, teletransportarse y moverse libremente, lo que dificulta significativamente que los enemigos logren detectarlo y derribarlo con facilidad. Además, la mecánica de combate se vuelve más realista, ya que el jugador debe coger el arma manualmente, presionar el gatillo y apuntar físicamente para impactar a los enemigos. Para garantizar que los disparos sean efectivos, los enemigos cuentan con una *Box Collision*, que actúa como área de detección del impacto. Esto añade un nivel extra de habilidad al combate, haciendo que la

precisión del jugador sea un factor clave en la simulación. Asimismo, los *widgets* que anteriormente se proyectaban sobre el *HUD*, como el expuesto en la Figura 3-120, las *HealthBar* o el *Widget* final han sido reemplazados por *widgets 3D*, los cuales son compatibles con la RV. De esta manera, se ha permitido la colocación sobre la cabeza de los enemigos de las *HealthBar*, mejorando la visibilidad dentro del entorno de RV.

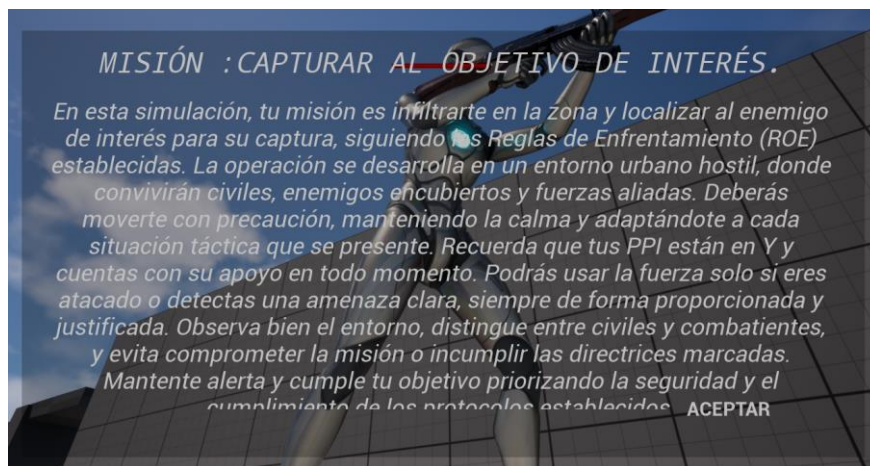


Figura 3-120 *Widget* de inicio (Autoría propia).

A continuación, en la **¡Error! No se encuentra el origen de la referencia.**, se muestra al jugador (*VRPawn*) con las gafas y controles. La imagen de la derecha muestra como *VRPawn* ve al *BP_EnergyRanged*, con su correspondiente *Widget* yendo a dispararle, gracias a las gafas.



Figura 3-121 Uso de gafas Focus Vision y escenario dentro de la simulación (Autoría propia).

4 RESULTADOS

4.1 Descripción del apartado

En el presente apartado se mostrarán los resultados obtenidos tras el desarrollo. Se listarán uno a uno cada uno de los elementos que han sido diseñados y programados de tal manera que tras la lectura del apartado el lector tenga claro qué objetivos se han alcanzado satisfactoriamente y cuál ha sido el desenlace del proyecto. Al tratarse de un proyecto marcado por lo visual, la mayoría de los resultados serán mostrados mediante imágenes de autoría propia.

En este apartado, los resultados se han organizado siguiendo el orden del desarrollo, pero también tomando como referencia el escenario descrito en el punto 3.2.3. Esto permite, de manera más visual, comprender y diferenciar a los distintos enemigos, así como sus funciones individuales.

4.2 Escenario

Aunque el escenario mostrado en la Figura 3-5 ha permitido crear la base de la acción de los enemigos y separar de manera exitosa sus funciones, este escenario no ha sido el definitivo. Una vez completada toda la programación, se ha importado el escenario PICOZU, creado por Diego Garabito De la Fuente, al *template* de tercera persona. Este entorno ha servido para generar la simulación final del ejercicio, donde se encuentran todos los enemigos, siendo su funcionalidad y propósito detallado más adelante. Este escenario se puede visualizar en la Figura 4-1.



Figura 4-1 Escenario para combate en población con enemigos y civiles (Autoría propia).

4.3 Funciones de enemigos

En la zona N, visible en la Figura 3-5, se encuentra el resultado del apartado 3.3.3, donde el enemigo asignado es *BP_PatrolEnemy*. Este enemigo sigue una ruta establecida de manera obligatoria, salvo que perciba a un objetivo. En ese caso, su comportamiento cambia al de *BP_EnemyMelee*, pasando a atacar al enemigo detectado. Su presencia en el escenario es fundamental, ya que simula a combatientes de Redland, quienes tienen órdenes estrictas de patrullar rutas específicas para no dejar sectores sin vigilancia y mantenerse en estado de alerta. Estos enemigos están programados para atacar a cualquier objetivo que no pertenezca a su país, convirtiéndose en una amenaza significativa para militares y civiles que no sean de agrado. Su combate es cuerpo a cuerpo, lo que indica que no portan armas de fuego y únicamente se defienden con un machete. En la Figura 4-2, se presenta una serie de imágenes que ilustran los aspectos generales de los enemigos “patrulla”. La imagen superior muestra en tiempo real cómo el enemigo se encuentra en el estado *Passive*, ejecutando la tarea *BTT_MoveAlongPatrolRoute*, lo que le permite recorrer con éxito los puntos establecidos en su ruta de patrullaje. Las dos imágenes inferiores muestran, por un lado, la parte del escenario donde se desarrolla la patrulla, incluyendo el camino a seguir, que simula un edificio con escaleras. Por otro lado, la otra imagen muestra al *BP_Player* observando el desplazamiento del *BP_PatrolEnemy* a lo largo de la ruta definida.



Figura 4-2 Patrullaje (Autoría propia).

Posteriormente, siguiendo con el desarrollo, se programó el *AI Perception* de los enemigos, logrando implementarlo con éxito. Como se observa en la

, los enemigos pueden detectar estímulos y reaccionar según lo programado. Esta figura, muestra en la imagen superior izquierda las esferas correspondientes a *SenseSight*, *SenseSound* y *SenseDamage*, representadas en verde, amarillo y rojo respectivamente. Gracias al *Gameplay Debugger* de UE, se ha verificado que el sistema funciona correctamente.

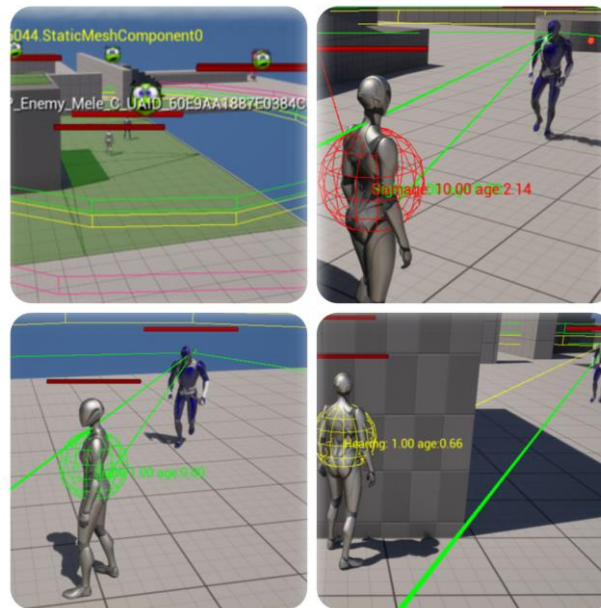


Figura 4-3 AI Perception del BP_EnemyBase (Autoría propia).

La imagen superior derecha muestra una esfera roja, indicadora de que el enemigo ha recibido un estímulo de daño, cambiando consecuentemente su estado a *Attacking*. En las imágenes inferiores, la izquierda ilustra el momento en que el enemigo detecta visualmente a su objetivo y procede a atacarlo, mientras que la imagen de la derecha muestra el caso en el que solo percibe un sonido, lo que lo lleva a investigar en lugar de atacar.

En el centro del mapa se encuentra el *BP_EnemyMelee*, el enemigo con más funcionalidades implementadas. La primera mecánica desarrollada para este enemigo fue la ejecución del *EQS_Strafe*, que le permite rodear al *BP_Player* y, tras un cierto tiempo, atacar. Esta acción se ha llevado a cabo con éxito y se puede observar en la Figura 4-4. La imagen superior muestra el punto ganador (*Winner*) donde el enemigo debe desplazarse, mientras que la imagen inferior izquierda muestra el siguiente punto ganador, ubicado justo al lado del anterior. Finalmente, la última imagen ilustra cómo, después del tiempo determinado por el *cooldown*, el enemigo ataca al jugador (el cilindro con colores es la *Sphere Trace* de la Figura 3-77). En las imágenes también puede observarse cómo el enemigo ejecuta correctamente su *Locomotion* individual, agachándose mientras rodea al *BP_Player*.

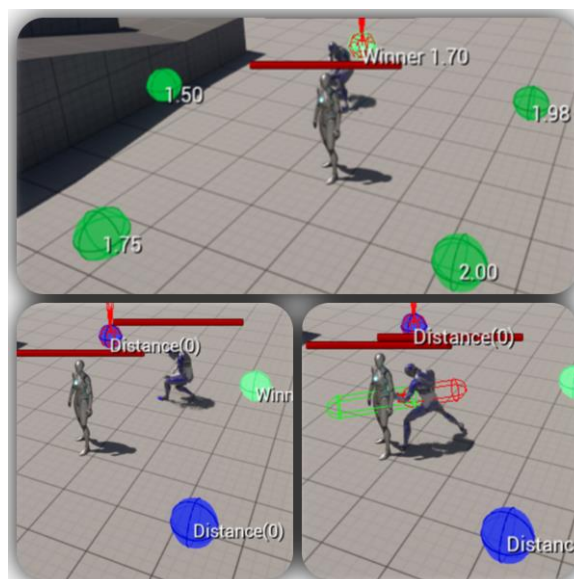


Figura 4-4 Strafing del BP_EnemyMelee (Autoría propia).

Cabe destacar que el sistema de salud se ha implementado en todos los personajes de la simulación, como se ve en la Figura 4-4. Esto permite que, al ejecutar los *Montages*, los personajes puedan infligir daño, curarse al bajar de vida y visualizar un *widget* de vida, el cual disminuye progresivamente hasta que el personaje muere.

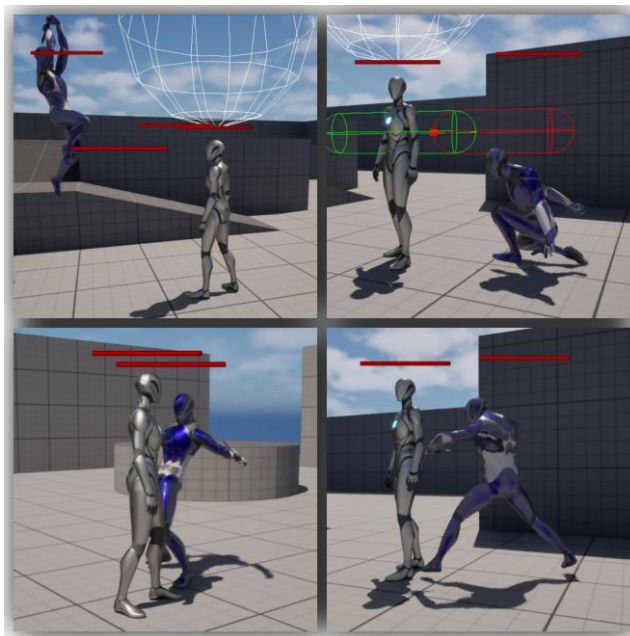


Figura 4-5 Ataques *BP_EnergyMelee* (Autoría propia).

Durante la programación de este enemigo se establecieron tres posibles ataques, los cuales se ejecutan alternándose o según la distancia al *BP_Player*. En caso de combate cercano el enemigo realiza *KnifeShortAttack*. Mientras que si está lejos ejecuta *KnifeLongAttack*, siendo este último un ataque especial en el que el enemigo salta hacia la futura posición del *BP_Player*, anticipando su movimiento. En la Figura 4-5 se observa la implementación exitosa de estos ataques. Las imágenes superiores muestran el inicio y la finalización del ataque a distancia, donde la esfera blanca representa la *Debug Sphere* de *JumpToLocation*, indicando dónde el enemigo predice que estará el *BP_Player*. En las imágenes inferiores se aprecian los ataques alternativos cuando el enemigo se encuentra dentro del *AttackRadius*, ejecutando animaciones de combate cuerpo a cuerpo.

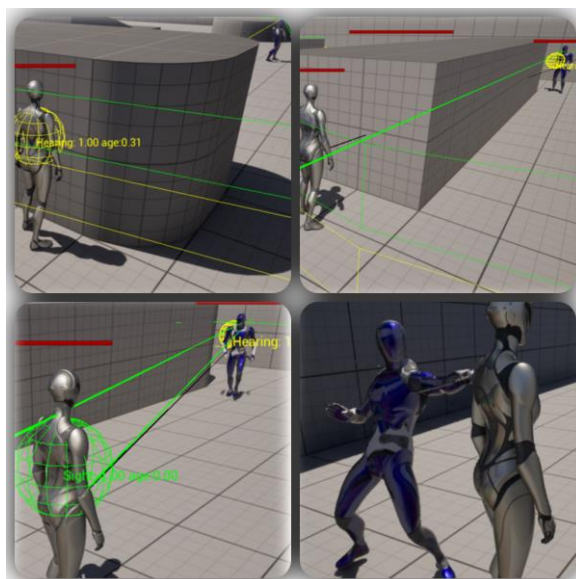


Figura 4-6 Acción Investigating (Autoría propia).

Como acción complementaria de este enemigo, se implementó el estado *Investigating*, permitiendo que, al percibir un sonido, el enemigo proceda a investigar la ubicación de su origen. En la

, se observa de manera secuencial cómo el enemigo reacciona al sonido percibido. En la imagen superior izquierda, el enemigo detecta el sonido que *BP_Player* ha producido, representado por la esfera amarilla y procede a ir hacia la ubicación del estímulo. Una vez llega a este (imagen superior derecha) pueden ocurrir dos situaciones: si no detecta al *BP_Player*, su estado cambia al estado *Passive* (imagen superior derecha); en caso contrario, como se aprecia en la imagen inferior izquierda y derecha, si logra verlo, procede a atacarlo de inmediato.

La última acción implementada para este enemigo es el estado *Seeking*. En este estado, el enemigo persigue al *BP_Player* hasta que lo pierde de vista durante 5 segundos. En ese momento, se desplaza automáticamente a la última ubicación donde lo vio y, desde allí, inicia un ciclo de búsqueda en cinco puntos estratégicos, utilizando el *EQS_Seeking*. Este sistema permite que el enemigo explore zonas alejadas y fácilmente accesibles (que son los lugares donde probablemente haya ido *BP_Player*), priorizando aquellas que no estén dentro de su línea de visión, incrementando así la posibilidad de encontrar al objetivo oculto. En la

Figura 4-7, se muestra claramente el comportamiento del enemigo durante el estado *Seeking*. En la imagen superior izquierda, se observa cómo detecta al *BP_Player* (esfera verde) y comienza a perseguirlo. Posteriormente, en la imagen central izquierda, al perder de vista al jugador, se dirige automáticamente a la última posición donde lo vio. La imagen superior derecha evidencia la ejecución de la rama correspondiente en el BT, vinculada a la Figura 3-70. Desde este punto, el enemigo activa el *EQS_Seeking*, desplazándose a cinco ubicaciones distintas en un intento de encontrar nuevamente al jugador, como se muestra en la imagen inferior izquierda. Finalmente, en la imagen inferior derecha, se ejecuta la rama del BT asociada a la Figura 3-71, demostrando el correcto funcionamiento de la búsqueda.

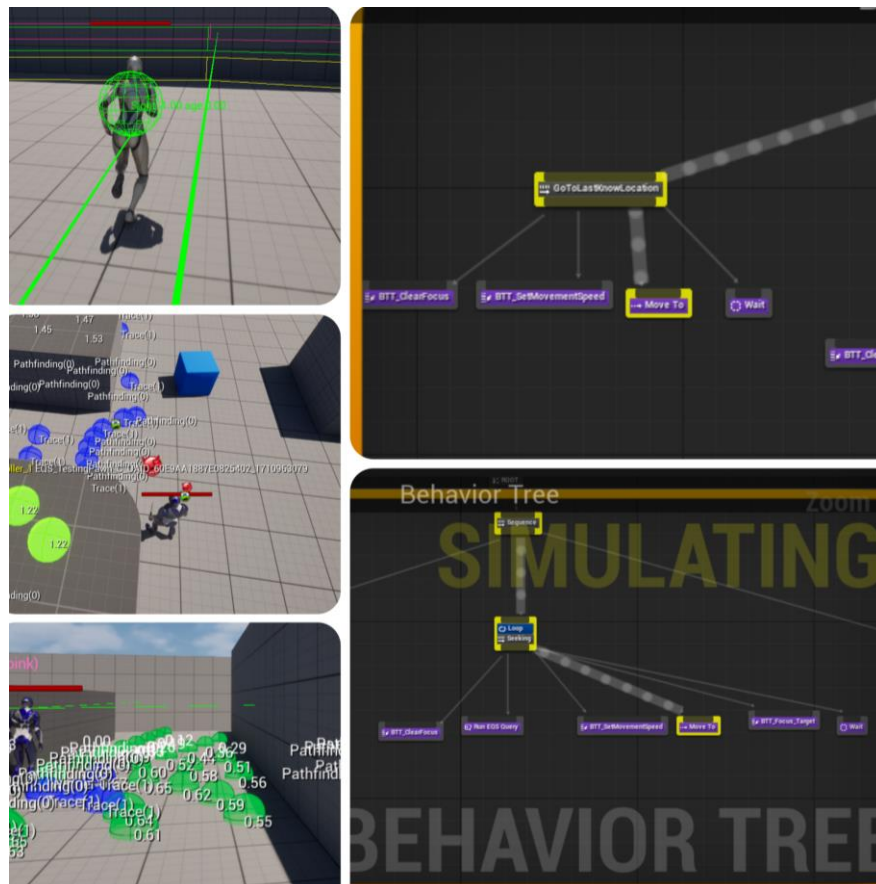


Figura 4-7 *Seeking* (Autoría propia).

Como inciso, y aplicable tanto a este personaje como al resto en la simulación, los *NavLinks* fueron implementados correctamente, permitiendo que los enemigos puedan saltar y moverse a distintas zonas del escenario, simulando un entorno más realista. Esto garantiza que los enemigos no estén restringidos solo a caminos predeterminados, sino que puedan acceder a las mismas áreas que el *BP_Player*. En la Figura 4-8 se puede observar cómo el enemigo salta a través del *NavLink*, desplazándose desde una plataforma elevada hasta el suelo.

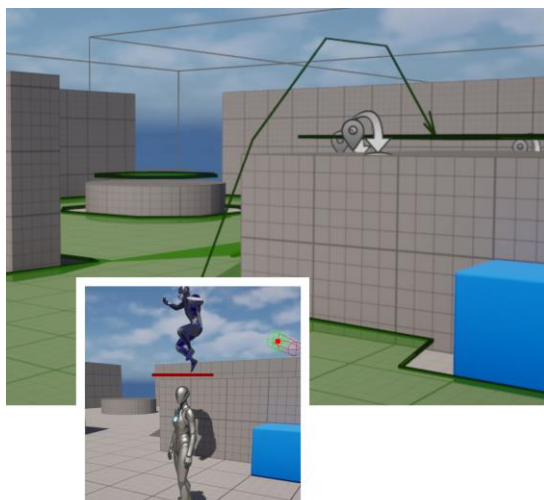


Figura 4-8 Uso de *NavLinks* (Autoría propia).

En la zona E del mapa general, se encuentra el *BP_EnergyRanged*, ubicado dentro de lo que simula ser una casa. Este enemigo, como se ha mencionado previamente, es capaz de disparar, ejecutando animaciones, efectos especiales y sonido para hacer la acción más inmersiva. Además, cuenta con la capacidad de buscar cobertura y curarse cuando su salud es baja, implementando las funciones y los EQS asociados a *FindCover* y *Heal*. Estas mecánicas permiten que *BP_EnergyRanged* adopte un comportamiento estratégico, evitando ser un blanco fácil y prolongando su tiempo en combate dentro de la simulación.

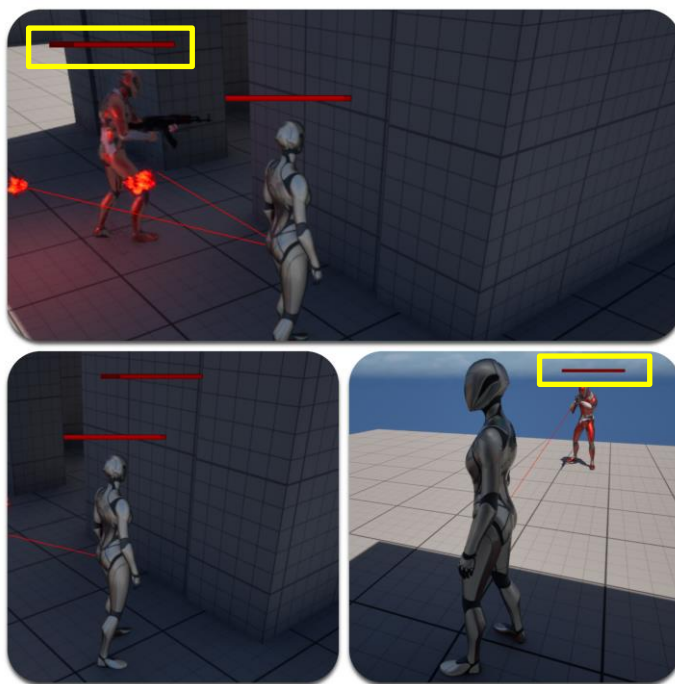


Figura 4-9 Función *FindCover* y *Heal* (Autoría propia).

El primer resultado, correspondiente a los apartados 3.3.5 y 3.3.6, es la implementación de la mecánica en la que el enemigo busca refugio y se cura cuando su vida alcanza un porcentaje específico. No se ha programado para que recupere una gran cantidad de salud, ya que resultaría poco realista, sino una porción pequeña, simulando a un combatiente que trata sus heridas para continuar en combate. Esta funcionalidad no solo refuerza la inmersión del *BP_Player*, sino que también introduce situaciones en las que el enemigo puede desaparecer momentáneamente del campo de visión, generando incertidumbre y obligando al jugador a estar atento a su reaparición.

En la Figura 4-9, se muestra cómo se ha logrado implementar correctamente las reacciones del enemigo al recibir daño y su acción de disparo. En la imagen superior, *BP_Player* ataca al enemigo, lo que provoca que se active la animación de impacto y que su barra de vida disminuya, mientras que la esfera roja indica que ha sido alcanzado. En la imagen inferior, el enemigo ejecuta su disparo, mostrando el *Line Trace* que define la trayectoria del proyectil, así como el efecto visual del disparo, reflejando el realismo buscado en el desarrollo.



Figura 4-10 Recibir y hacer daño (*TakeDamage*) (Autoría propia).

Además, para mejorar la inmersión visual del *BP_Player*, se le ha implementado un arma (Figura 4-11), la cual en futuras versiones será un *HK G36E*, permitiéndole disparar de manera similar al resto de los personajes de la simulación.



Figura 4-11 *BP_Player* con arma (Autoría propia).

La última funcionalidad implementada en el *BP_EnergyRanged* es *FindIdealRange*. Esta acción utiliza un EQS que genera una serie de puntos estratégicos en forma de rectángulo alrededor del enemigo, permitiéndole moverse a posiciones más favorables en combate. En la Figura 4-12, se observa cómo el enemigo selecciona el punto *winner* y se desplaza hacia él, asegurando que mantenga una distancia adecuada mientras busca la mejor cobertura posible.

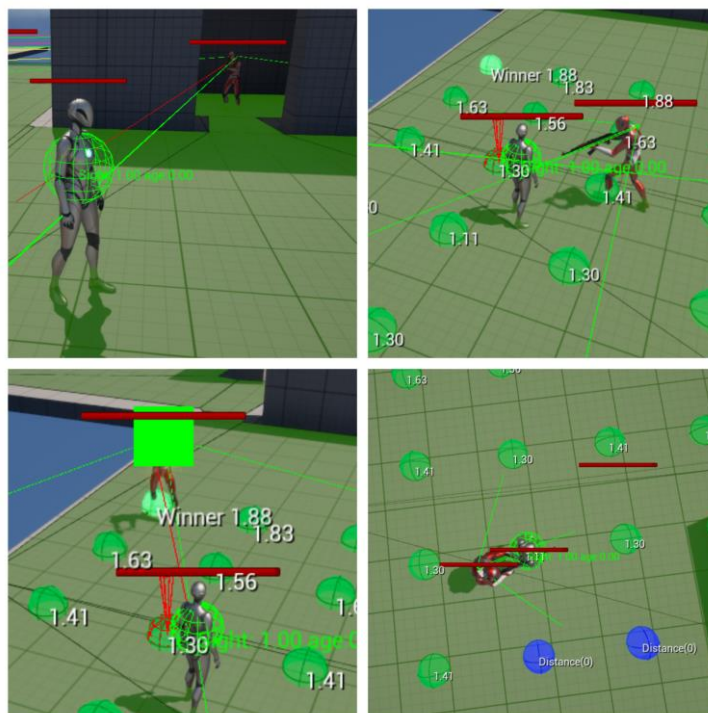


Figura 4-12 *EQS_FindIdealRange* (Autoría propia).

En la zona W del mapa, se encuentran dos enemigos: *BP_ShooterLongRange* y *BP_FriendTeam*. Ambos personajes han sido implementados correctamente en el escenario, ejecutando sus funciones de manera efectiva según lo establecido en el desarrollo.

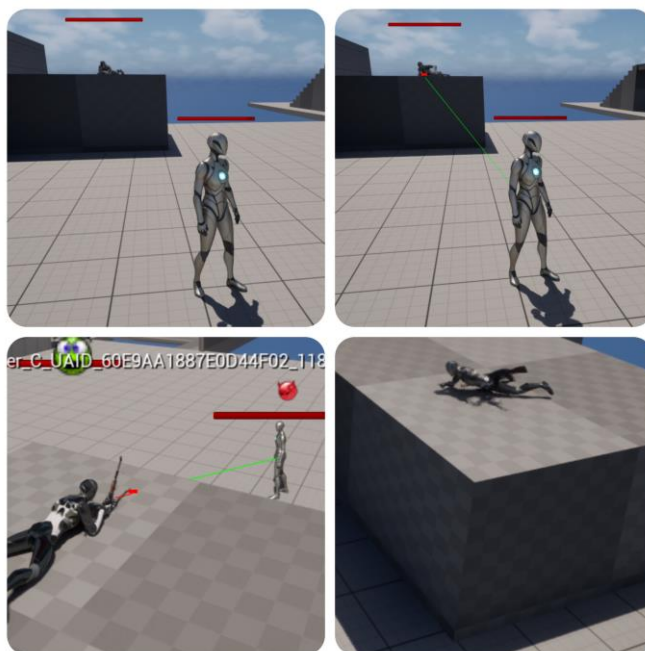


Figura 4-13 *BP_ShooterLongRange* (Autoría propia).

Respecto al *BP_ShooterLongRange*, como se observa en la Figura 4-13, este enemigo está diseñado para ubicarse en zonas elevadas. Su *AI Perception* es más relajada, permitiéndole detectar objetivos a mayor distancia y altura. En la imagen superior izquierda, el enemigo ha detectado al *BP_Player*; como consecuencia, en la imagen superior derecha, procede a dispararle. En las dos imágenes inferiores, se aprecia el proceso de recarga que realiza tras ejecutar sus ataques.

Por último, dentro de este sector del escenario se encuentran los soldados aliados del *BP_Player*. En la imagen se puede observar cómo, al inicio de la simulación, toman posición detrás del *BP_Player*, cubriendo diferentes sectores sin cambiar su ubicación (Aunque *BP_Player* se mueva). Sin embargo, cuando el *BP_Player* decide que deben moverse para cubrir una posición específica (como una entrada, una escalera, una esquina o un pasillo), los soldados se desplazan hasta el punto indicado y permanecen allí por un tiempo, como se ve en la imagen inferior derecha. Además, en la imagen inferior izquierda, se observa el *Line Trace* que lanza el *BP_Player* para designar la posición donde deben dirigirse.

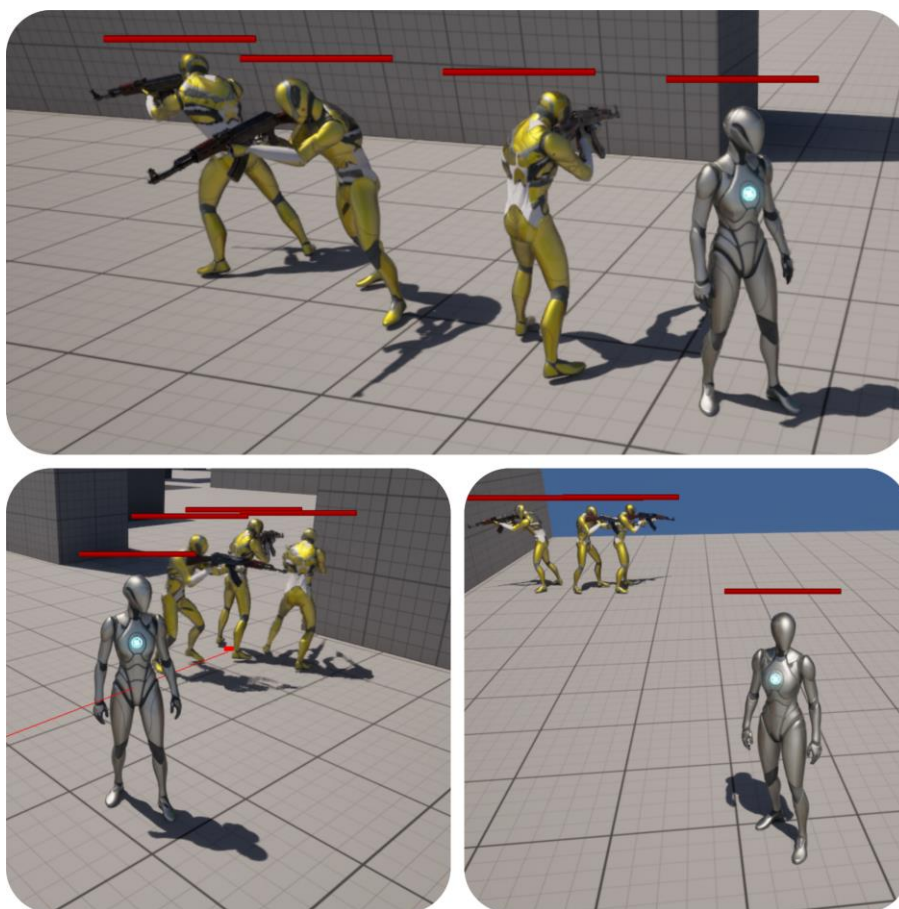


Figura 4-14 *BP_FriendTeam* (Autoría propia).

Finalizado el sector W, se da paso al sector S, el último del escenario. Este sector se subdivide en tres zonas diferenciadas: la zona del *BP_SurrenderEnemy*, la del *BP_RunawayEnemy* y la parte más amplia, donde se han implementado las lógicas descritas en los apartados 3.3.8 y 3.3.9.

La imagen del *BP_RunawayEnemy* muestra cómo el enemigo comienza en posición *Passive* al no percibir ninguna amenaza. Sin embargo, en el momento en que detecta la presencia de otro personaje, activa su comportamiento de huida mientras emite la frase: "Estoy huyendo, yo no he hecho nada". Las dos imágenes inferiores ilustran su escape progresivo hasta desaparecer del campo de visión. Este tipo de enemigo aporta variedad a la simulación, ya que se pueden generar *Child Blueprints* a partir de él, asignándoles un *mesh* de civil, permitiendo que personajes no combatientes reaccionen huyendo al detectar enemigos o individuos de equipos contrarios, proporcionando pistas al *BP_Player* sobre la posible presencia de amenazas. Además, en el caso de que el *BP_RunawayEnemy* sea un objetivo de

interés, su comportamiento de fuga puede generar situaciones de tensión y comprometer la estrategia del jugador.

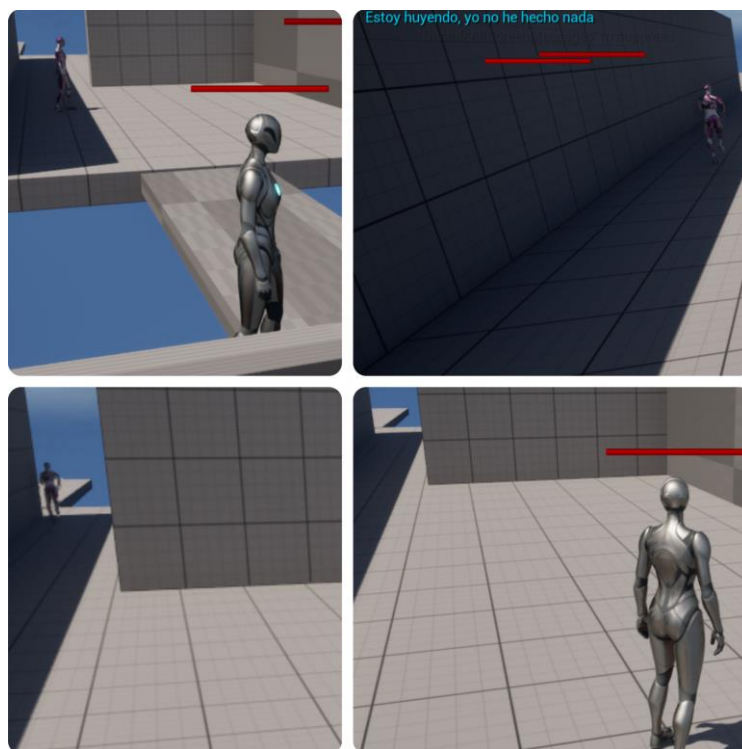


Figura 4-15 BP_RunAwayEnemy (Autoría propia).

El otro enemigo presente en este sector es el *BP_SurrenderEnemy*, cuya funcionalidad ha sido implementada con éxito. Cuando detecta la presencia de un enemigo, automáticamente ejecuta una animación de rendición mientras pronuncia la frase: "Me rindo". Este comportamiento añade profundidad a la simulación, permitiendo la posibilidad de enfrentar situaciones en las que el jugador deba decidir cómo proceder ante enemigos que no representan una amenaza inmediata.

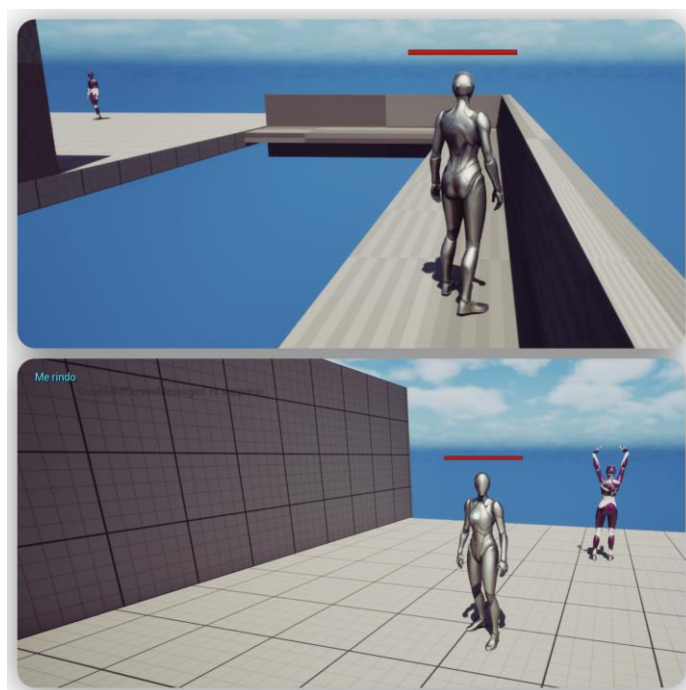


Figura 4-16 BP_SurrenderEnemy (Autoría propia).

Por último, en esta zona se ha implementado la programación de los puntos 3.3.8 y 3.3.9, logrando que los enemigos coordinen sus ataques al *BP_Player* en lugar de atacar todos simultáneamente. Esto se ha conseguido mediante el sistema de *tokens*, donde los enemigos solicitan y devuelven un *token* antes de atacar, asegurando que el combate sea más estructurado y estratégico.

En la Figura 4-17, se puede observar claramente este proceso. En la imagen superior izquierda, el *BP_Player* está siendo atacado por dos *BP_EnemyMelee*, lo que se evidencia con sus respectivas *Sphere Trace* generadas al realizar un ataque, además del *Line Trace* del disparo ejecutado por el *BP_EnemyRanged*. Sin embargo, en la imagen superior derecha, y siguiendo con la secuencia en la imagen inferior izquierda y finalizando en la inferior derecha, se aprecia la ejecución ordenada del combate: primero ataca el *BP_EnemyPatrol* (marcado en verde), realizando su *Sphere Trace* de daño, luego el *BP_EnemyMelee* y, por último, el *BP_EnemyRanged* efectúa su disparo.

Gracias a este sistema, la simulación adquiere una mayor fluidez e inteligencia, evitando situaciones donde el jugador se vea abrumado por múltiples ataques simultáneos y permitiendo un combate más equilibrado y realista.

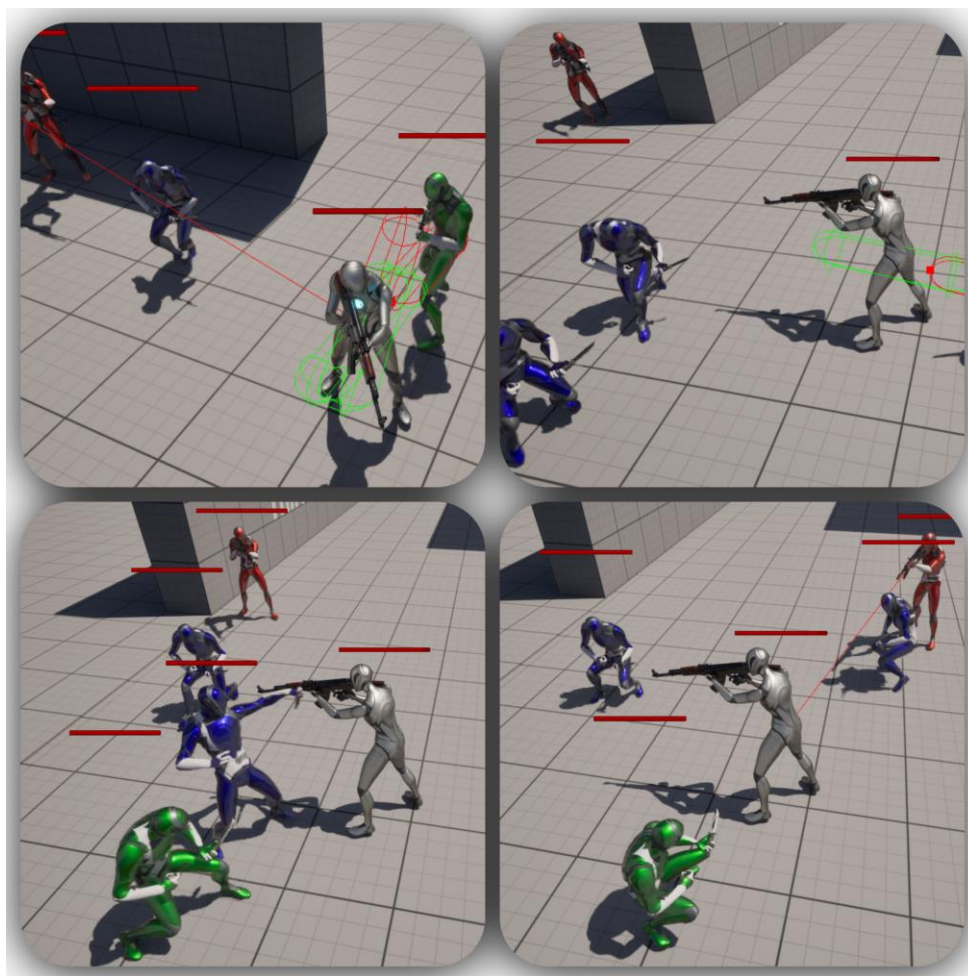


Figura 4-17 Combate conjunto a través de sistema de comunicación (Autoría propia).

Lo penúltimo en desarrollarse, y probablemente el aspecto más relevante, es la integración de equipos. Esta implementación ha permitido establecer una base sólida para generar una variedad de situaciones dentro de la simulación. Gracias a este sistema, es posible crear escenarios donde haya civiles encubiertos que puedan atacar repentinamente, enemigos que terminan aliándose con el *BP_Player* para alcanzar un objetivo en común, o incluso facciones en conflicto entre sí, agregando profundidad y realismo al entorno de combate. En la Figura 4-18, se muestra claramente el escenario inicial (imagen superior), donde dos *BP_EnemyRanged* y dos *BP_EnemyMelee* se enfrentan, al pertenecer a equipos

distintos. Hay que destacar, en esta imagen el logro de que *BP_EnergyRanged* quede en *passive* cuando muere el otro enemigo, y no siga disparando.

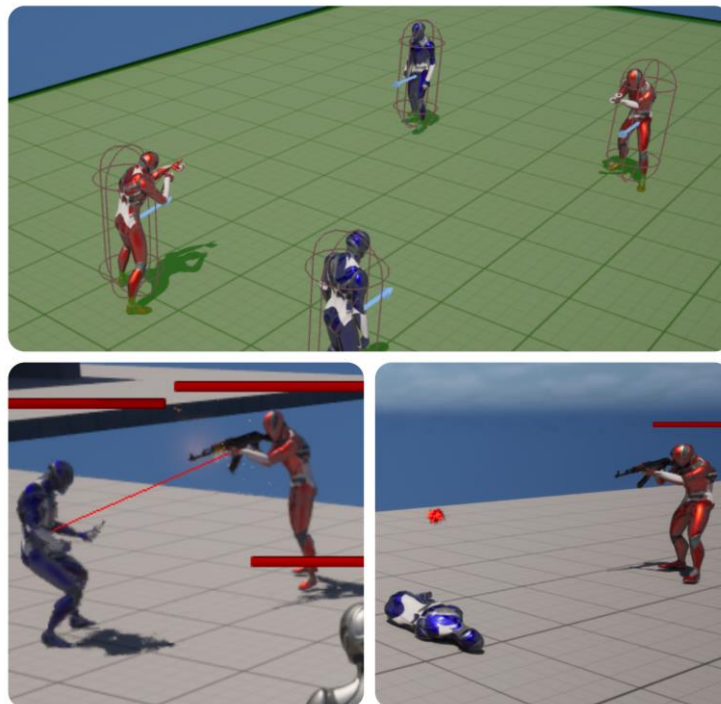


Figura 4-18 Combate entre enemigos de diferentes equipos (Autoría propia).

Por último, se ha conseguido que *BP_TargetEnemy* realice sus funciones de forma correcta, marcando el fin del juego al capturarlo. En la Figura 4-19, se puede observar visualmente cada paso del proceso: desde el momento en que el enemigo detecta a *BP_Player*, se rinde y dice: “me rindo, habéis logrado capturarosme”, hasta su desplazamiento hacia *BP_Player* y la aparición del *Widget* de finalización.



Figura 4-19 Captura del objetivo de interés (Autoría propia).

A través del siguiente código QR, se puede visualizar un vídeo explicativo donde se muestran las principales acciones y comportamientos de los enemigos. En él se detalla cómo reaccionan según el tipo de enemigo, su programación individual y colectiva, así como la aplicación del sistema de comunicación y las dinámicas derivadas de la integración de equipos. El enlace al vídeo es el siguiente: <https://youtu.be/DRjXn8evibo?si=JGLTPzLO0nfb4jnu>.



4.4 Integración en un escenario para aplicar procedimiento MOUT

En este apartado se expondrán figuras que representan el escenario final integrado en RV, ambientado en el escenario de PICOZU. Asimismo, se detallará la localización estratégica de enemigos y civiles dentro del escenario, diseñada para cumplir con los objetivos de la simulación.

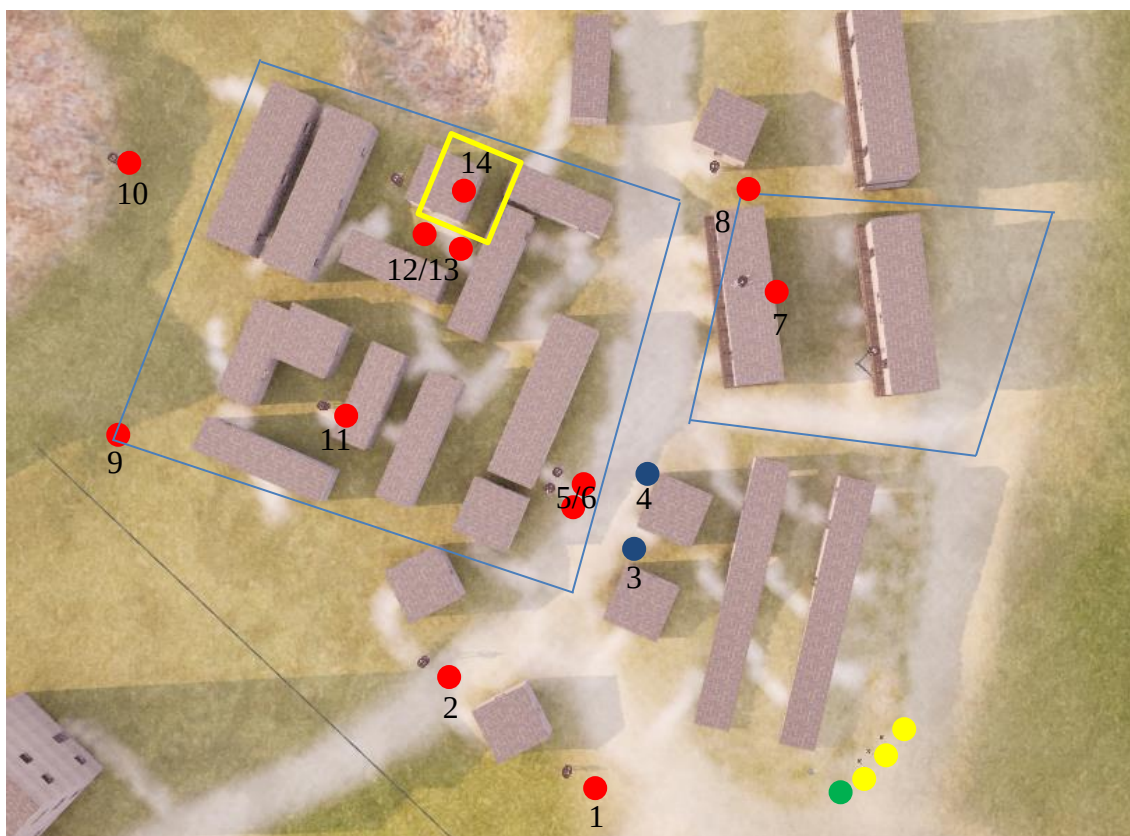


Figura 4-20 Mapa táctico de un ejercicio (Autoría propia).

Cabe destacar, respecto a la Figura 4-20, que este escenario ha sido creado como una prueba para que el jugador pueda observar todas las actividades programadas. No obstante, el entorno es totalmente adaptable y puede configurarse según las necesidades del ejercicio, ya sea diseñando escenarios con menor nivel de amenaza, utilizando enemigos más pasivos, o bien creando un entorno hostil donde la mayoría de los enemigos sean agresivos y se evalúe la capacidad de reacción y precisión del jugador.

En este escenario se han integrado múltiples situaciones diseñadas para sumergir al jugador en entornos complejos. El círculo verde representa al *VRPawn*, acompañado por tres puntos amarillos que simbolizan a los tres soldados (Figura 4-21) que lo escoltan durante la misión.



Figura 4-21 Soldados de operaciones especiales (Autoría propia).

En las inmediaciones, los puntos rojos 1 y 2 representan a un enemigo que huye al detectar al jugador y otro que se rinde de inmediato respectivamente. Los puntos rojos 5 y 6, junto con los puntos azules 3 y 4, corresponden a *child blueprints* de las clases base a los que se les ha incrementado la vida, permitiendo al jugador poder observar con detalle lo que sucede en el centro del escenario, sin antes haberse matado entre ellos. Los puntos azules representan a dos civiles: uno que adopta la lógica de *BP_EnemyBase*, manteniéndose pasivo, y otro que utiliza la lógica de *BP_EnemyMelee*, generando un conflicto armado con los enemigos ubicados en los puntos 5 y 6, quienes emplean las lógicas de *BP_EnemyRanged* y *BP_EnemyMelee*, respectivamente. Esta situación genera un dilema para el jugador, al presenciar tanto la agresión a civiles desarmados como el enfrentamiento con otros que sí responden con violencia, como se ve en la Figura 4-22.



Figura 4-22 Civil encubierto contra enemigo de Redland (Autoría propia).

Adicionalmente, los puntos 8 y 9 señalan a enemigos patrullando, mientras que el punto 7 posiciona a un francotirador (Figura 4-23) en una zona elevada. En el punto 11, se sitúa un civil encubierto que aparenta estar desarmado, pero que atacará al *VRPawn* si lo detecta, actuando como un enemigo más.



Figura 4-23 Francotirador (Autoría propia).

Finalmente, los puntos 12 y 13 corresponden a dos *BP_EnergyRanged* armados con AK-47. Estos son los encargados de proteger la zona donde se encuentra el *BP_TargetEnemy*. En la imagen superior de la Figura 4-24 se encuentra el *BP_TargetEnemy*, mientras que en la foto de abajo los dos *BP_EnergyRanged* protegiéndolo.



Figura 4-24 Infraestructura del *BP_TargetEnemy* (Autoría propia).

Cabe mencionar también la presencia de civiles patrullando o simplemente localizados en el mapa, simulando transeúntes en un entorno urbano, los cuales pueden o no ser atacados, generando así situaciones completamente imprevisibles y aleatorias.



Figura 4-25 Civiles por el mapa (Autoría propia).

Este es solo un escenario entre un sinnúmero de posibilidades. La ventaja de contar con equipos y *meshes* diferenciados es que permite una amplia personalización de cada ejercicio, generando situaciones diversas que obligan al *VRPawn* a adaptar su comportamiento. Esta flexibilidad brinda la opción de crear escenarios que vayan aumentando de manera exponencial su peligro. De esta manera, los jugadores pueden comenzar con ejercicios básicos de poca interactividad, hasta llegar a escenarios muy complejos donde la rápida implementación de las ROE es la clave del éxito.

A continuación, se muestran varias imágenes tomadas tanto directamente desde las gafas Focus Vision, como desde capturas del ordenador (Figura 4-26), que ilustran la perspectiva del jugador dentro de la simulación en RV. En ellas se aprecian distintas situaciones, como la presencia de víctimas, civiles que actúan como combatientes encubiertos, así como el uso de la pistola FN9 por parte del *VRPawn*, entre otros elementos destacados de la simulación.

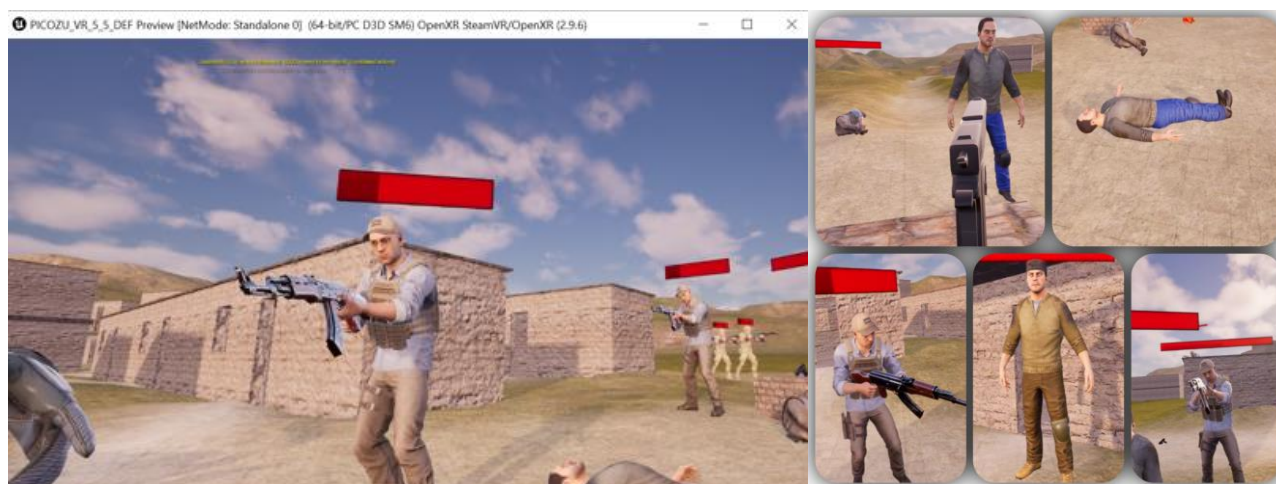


Figura 4-26 Captura de pantalla de lo que el *VRPawn* ve con las gafas (Autoría propia).

5 CONCLUSIONES Y LÍNEAS FUTURAS

5.1 Descripción del apartado

En este apartado se presentan las conclusiones derivadas de las distintas fases del desarrollo de este trabajo. Posteriormente, se realiza un resumen final del proyecto completo, evaluando si se lograron cumplir satisfactoriamente los objetivos planteados al inicio. Para concluir, se incluye una breve descripción de posibles mejoras o futuras modificaciones al proyecto, así como algunas alternativas de desarrollo que podrían ampliar el alcance y las funcionalidades de la simulación llevada a cabo.

5.2 Conclusiones

5.2.1 Diseño del escenario

Como primer punto a destacar, se puede afirmar que el diseño del escenario ha sido una parte esencial del proyecto, ya que ha servido de base para la posterior implementación de la lógica de los enemigos y del jugador. El escenario de prueba inicial permitió establecer un entorno controlado donde se validaron los sistemas básicos de IA y la lógica de combate. Sin embargo, ha sido la integración del escenario PICOZU la que ha dotado de sentido completo a la simulación, permitiendo reproducir un entorno urbano realista en el que poner a prueba todas las funcionalidades desarrolladas. Esta implementación, basada en un entorno con múltiples espacios abiertos y cerrados, ha posibilitado la creación de situaciones tácticas complejas que ponen en compromiso al jugador.

El diseño de este entorno ha facilitado el desarrollo de situaciones variadas, desde patrullas y enfrentamientos directos hasta el planteamiento de zonas de emboscadas o civiles encubiertos. Gracias al diseño del escenario, los jugadores que utilicen las gafas de realidad virtual se enfrentarán a un reto significativo a la hora de actuar correctamente, conociendo la situación táctica y el nivel de escalada. Factores clave como la agilidad mental, el conocimiento exhaustivo de las reglas de enfrentamiento, así como la correcta aplicación de los procedimientos de movimiento en exteriores e interiores, serán determinantes para alcanzar o no el éxito en la misión.

5.2.2 Programación e implementación de acciones

La programación mediante el sistema visual de BP de UE ha constituido, sin lugar a dudas, la piedra angular de este proyecto. El motor gráfico y su sistema de scripting han permitido una implementación modular y estructurada de las acciones, facilitando la creación de comportamientos tanto individuales como colectivos para cada uno de los personajes y elementos del entorno.

La lógica de cada enemigo ha sido diseñada cuidadosamente para garantizar que respondan de manera autónoma y diferenciada, respetando las características definidas durante la fase de desarrollo. Desde enemigos que patrullan, francotiradores en posiciones elevadas, hasta civiles encubiertos, todos

cuentan con sistemas de IA que les permiten tomar decisiones en función del contexto. De igual manera, se han desarrollado EQS personalizados para definir zonas seguras, rutas de patrulla y puntos estratégicos desde los cuales los enemigos pueden operar de manera efectiva.

Asimismo, se ha implementado la lógica completa del *BP_Player* y de los soldados aliados que le acompañan, quienes son capaces de realizar maniobras de cobertura y desplazamiento según las órdenes del jugador. Además, el sistema de salud, las animaciones de ataque, los efectos de disparo y la gestión sonora han sido integrados exitosamente, consiguiendo una simulación fluida y realista.

Cada una de estas acciones genera personajes inteligentes que crean situaciones impredecibles, sumergiendo al jugador en escenarios dinámicos y complejos donde incluso el instructor desconoce el resultado final. La base de IA desarrollada permite replicar procedimientos similares a los utilizados en la Escuela Naval Militar con el simulador Galeón, pasando de ejercicios iniciales de familiarización, con pocas ROE autorizadas, a situaciones de alta exigencia donde la inteligencia y la rapidez del jugador resultan determinantes.

5.2.3 Integración de la RV en el escenario

La integración de la RV ha supuesto un avance significativo en términos de inmersión y realismo para el trabajo. La migración de toda la lógica y programación al *template* de RV no ha sido una simple adaptación, sino una reconfiguración de ciertas mecánicas esenciales para su correcto funcionamiento en este nuevo entorno.

Esta integración permite que el instructor supervise en tiempo real las decisiones y acciones tomadas por el jugador a través de la pantalla del ordenador, mientras este interactúa con la simulación mediante las gafas de realidad virtual. La comunicación entre ambos puede establecerse mediante auriculares, lo que facilita al instructor la evaluación precisa de cada acción en el momento en que ocurre.

El uso de la realidad virtual como herramienta proporciona una experiencia mucho más realista que una simulación convencional en ordenador, donde resulta más difícil analizar de manera efectiva las decisiones del jugador. Además, la incorporación de *meshes* y la recreación de un entorno basado en un escenario real potencian la inmersión del usuario, acercando la simulación a una experiencia hiperrealista, comparable a la que se experimenta en un ejercicio tradicional.

5.2.4 Conclusión final

En definitiva, de esta manera el presente trabajo ha logrado la creación de una plataforma de simulación inmersiva que permite el adiestramiento táctico de combate urbano en un entorno controlado y altamente realista. A través de la implementación de pautas de control automatizadas, la programación modular mediante BP y la integración de las ROE, se ha conseguido ofrecer una herramienta formativa capaz de replicar situaciones operacionales reales y complejas. La transición del entorno de tercera persona a la Realidad Virtual ha reforzado la inmersión y la interactividad, dotando al usuario de una experiencia que mejora sus capacidades de toma de decisiones y ejecución táctica. Además, los objetivos planteados al inicio han sido alcanzados satisfactoriamente, e incluso se han superado con el desarrollo de elementos adicionales como la creación de escenarios realistas, la variedad de enemigos y la integración de armamento adaptado. En conjunto, el proyecto representa un paso adelante en la modernización de las metodologías de instrucción militar y pone de manifiesto el enorme potencial de la Realidad Virtual como herramienta de apoyo al adiestramiento en las FAS.

5.3 Líneas futuras

5.3.1 A corto plazo

Una mejora factible a corto plazo sería la implementación de un modo multijugador cooperativo que permita a varias unidades entrenar de forma conjunta dentro de la simulación, imitando de manera más realista las tácticas operativas de las Fuerzas Armadas. Esto facilitaría la coordinación estratégica entre

efectivos, permitiendo simular ejercicios más complejos, donde las directrices a los soldados no se programarían a través de los BP, sino que surgirían de la comunicación directa entre las distintas entidades. De este modo, el líder de la unidad podría posicionar a cada miembro según la doctrina, asignar tareas específicas y fomentar la transmisión de conocimientos entre los jugadores para mejorar la respuesta conjunta frente a la amenaza.

Por otro lado, sería interesante mejorar la interacción física del usuario con el entorno virtual mediante la incorporación de dispositivos de locomoción, como plataformas de movimiento o sensores integrados en el calzado, permitiendo caminar, correr o girar libremente en un espacio reducido, lo que incrementaría la inmersión y el realismo del adiestramiento. Además, convendría optimizar aspectos técnicos del simulador, como el código fuente, las animaciones del avatar y la complejidad de la inteligencia artificial enemiga, con el fin de crear un entorno más dinámico, desafiante y adaptativo a las decisiones del usuario.

5.3.2 A largo plazo

Con la vista puesta en el futuro, sería interesante evolucionar el simulador actual, que se limita a la aplicación de procedimientos de doctrina MOUT en entornos urbanos, hacia una plataforma que abarque operaciones conjuntas de los tres ejércitos. Se podrían desarrollar simulaciones multijugador en las que participen unidades de tierra, mar y aire, reproduciendo maniobras completas que incluyan fases de planeamiento e inteligencia, operaciones aéreas de apoyo cercano mediante pilotos en cabinas simuladas, y desembarcos anfibios llevados a cabo desde buques virtuales que permitan proyectar fuerzas terrestres en escenarios urbanos.

Asimismo, este enfoque permitiría crear entornos en los que la infantería terrestre desarrolle sus misiones en combinación con otros medios, siguiendo una estructura más parecida a las operaciones reales. Este planteamiento abriría la posibilidad de emplear la simulación para entrenar a personal de los tres ejércitos en misiones conjuntas, integrando desde el control de maquinaria naval hasta la coordinación con aeronaves que operan en apoyo a la maniobra terrestre. Además, la implementación de este tipo de simulaciones reduciría significativamente los costes asociados al adiestramiento tradicional, disminuyendo la necesidad de material físico, personal desplegado y facilitando el acceso a ejercicios complejos sin requerir grandes recursos logísticos.

6 BIBLIOGRAFÍA

- [1] R. Blanco, «LA INDUSTRIA 4.0: EL ESTADO DE LA CUESTIÓN».
- [2] «La cuarta revolución industrial, el “algoritmo de guerra” y su posible aplicación a la Defensa española», Real Instituto Elcano. Accedido: 27 de febrero de 2025. [En línea]. Disponible en: <https://www.realinstitutoelcano.org/analisis/la-cuarta-revolucion-industrial-el-algoritmo-de-guerra-y-su-posible-aplicacion-a-la-defensa-espanola/>
- [3] mkt@mexitrade.com.mx, «¿Qué es la Industria 4.0? Explicado fácilmente», Mexitrade. Accedido: 31 de enero de 2025. [En línea]. Disponible en: <https://mexitrade.com.mx/que-es-la-industria-4-0-explicado-facilmente/>
- [4] «1366x768 Starlink Laptop HD ,HD 4k Wallpapers,Images,Backgrounds,Photos and Pictures», hdqwalls. Accedido: 27 de febrero de 2025. [En línea]. Disponible en: <https://hdqwalls.com/wallpaper/1366x768/starlink>
- [5] «Diferencias entre realidad aumentada, realidad virtual y realidad mixta». Accedido: 2 de febrero de 2025. [En línea]. Disponible en: <https://www.xataka.com/basics/diferencias-entre-realidad-aumentada-realidad-virtual-y-realidad-mixta?>
- [6] K. Lemus, «Características de la Realidad Virtual: una Guía para Principiantes». Accedido: 2 de febrero de 2025. [En línea]. Disponible en: <https://blog.inmersys.com/caracteristicas-de-la-realidad-virtual>
- [7] «Non-immersive virtual reality isolated cartoon vector illustrations se Stock Vector by ©VisualGeneration 566361262». Accedido: 2 de febrero de 2025. [En línea]. Disponible en: <https://depositphotos.com/vector/non-immersive-virtual-reality-isolated-cartoon-vector-illustrations-se-566361262.html>
- [8] «Curso Realidad Virtual y Aumentada con Unity | Formación VR». Accedido: 2 de febrero de 2025. [En línea]. Disponible en: https://www.masterd.es/curso-realidad-virtual-y-aumentada?piloto=N84&utm_source=bing_o_c&utm_medium=Ads&utm_campaign=326428051&utm_content=&utm_term=b_estudia%20realidad%20virtual&adgrp=1254543287399499&adpos=&adpos=&extension={extensionid}&locphysical=164580&placement=&msclkid=651c180e27881fe2f5437f91f7a60487
- [9] luciaclemares, «¿Qué tres tipos de realidad virtual existen?», Telefónica. Accedido: 2 de febrero de 2025. [En línea]. Disponible en: <https://www.telefonica.com/es/sala-comunicacion/blog/que-tres-tipos-realidad-virtual-existen/>

- [10] A. I. USIL, «Realidad Virtual: otro mundo al alcance de tus ojos», FerVilela Digital Consulting. Accedido: 2 de febrero de 2025. [En línea]. Disponible en: <https://fervilela.com/realidad-virtual-otro-mundo-al-alcance-de-tus-ojos.html>
- [11] J. Álvarez, «Kriegsspiel, el juego de mesa que entrenó a los oficiales prusianos para ganar la guerra con Francia», La Brújula Verde. Accedido: 2 de febrero de 2025. [En línea]. Disponible en: <https://www.labrujulaverde.com/2018/05/kriegsspiel-el-juego-de-mesa-que-entreno-a-los-oficiales-prusianos-para-ganar-la-guerra-con-francia>
- [12] J. L. G. Benavides, «La simulación militar en España (I)», Ejércitos. Accedido: 2 de febrero de 2025. [En línea]. Disponible en: <https://www.revistaejercitos.com/articulos/la-simulacion-militar-en-espana-i-el-ejercito-de-tierra/>
- [13] «Indra entregará al ejército español siete nuevos simuladores para entrenar a soldados destinados a misiones de paz | indra». Accedido: 2 de febrero de 2025. [En línea]. Disponible en: <https://www.indracompany.com/pt-br/node/45459?>
- [14] «Ejército de Tierra recibe de Indra simulador vehículo combate Pizarro - Noticias Defensa España». Accedido: 2 de febrero de 2025. [En línea]. Disponible en: <https://www.defensa.com/espana/entregado-ejercito-tierra-primer-simulador-vehiculo-combate?>
- [15] simumak, «Simulador Multi-vehículo para el Ejército de Tierra», Simumak. Accedido: 2 de febrero de 2025. [En línea]. Disponible en: <https://simumak.com/blog/2024/02/12/simulador-vultivehiculo-ejercito-tierra/>
- [16] T. E. S. Pintado, «SIMULACIÓN EN EL EJÉRCITO DEL AIRE».
- [17] «El Ejército del Aire español compra otros 16 PC-21 y simuladores asociados. – Galaxia Militar». Accedido: 2 de febrero de 2025. [En línea]. Disponible en: <https://galaxiamilitar.es/el-ejercito-del-aire-espanol-compra-otros-16-pc-21-y-simuladores-asociados/?>
- [18] «f60162878258e98c13851a90cee42ade23ead117.jpg (1200×665)». Accedido: 2 de febrero de 2025. [En línea]. Disponible en: <https://simplaza.org/wp-content/uploads/2022/10/f60162878258e98c13851a90cee42ade23ead117.jpg>
- [19] Armada [@Armada_esp], «¡Siempre preparados para combatir cualquier incidencia a bordo! Los trozos de Seguridad Interior de la fragata #Canarias se adiestran en el Centro de Adiestramiento de Seguridad Interior de Rota. #SomosLaArmadaes  <https://t.co/LOVs3I8z2s>», Twitter. Accedido: 2 de febrero de 2025. [En línea]. Disponible en: https://x.com/Armada_esp/status/1624075972009164800
- [20] «Indra desarrolla un sistema de simulación basado en realidad virtual que reduce a la mitad el tiempo para formar pilotos | indra». Accedido: 3 de febrero de 2025. [En línea]. Disponible en: <https://www.indracompany.com/es/noticia/indra-desarrolla-simulacion-basado-realidad-virtual-reduce-mitad-tiempo-formar-pilotos?>
- [21] J. C. Benavidez, «La realidad virtual funciona, ayudando a entrenar al Ejército de EE. UU.», Zona Militar. Accedido: 3 de febrero de 2025. [En línea]. Disponible en: <https://www.zona-militar.com/2020/02/27/la-realidad-virtual-funciona-ayudando-a-entrenar-a-soldados-del-ejercito-de-ee-uu/>
- [22] «La Realidad Virtual y Aumentada en el Contexto Militar – CORPE | Naturaleza Militar». Accedido: 3 de febrero de 2025. [En línea]. Disponible en: <https://www.naturalezamilitar.es/la-realidad-virtual-y-aumentada-en-el-contexto-militar?>
- [23] «El Ejército cuenta con una nueva zona de combate urbano en San Gregorio». Accedido: 3 de febrero de 2025. [En línea]. Disponible en: <https://www.infodefensa.com/texto-diario/mostrar/3125820/ejercito-cuenta-nueva-zona-combate-urbano-san-gregorio?>

- [24] «Google Maps», Google Maps. Accedido: 3 de febrero de 2025. [En línea]. Disponible en: <https://www.google.es/maps>
- [25] «Rim PN Fusiles Experimental | PDF | Infantería | Ametralladora». Accedido: 3 de febrero de 2025. [En línea]. Disponible en: <https://es.scribd.com/document/663626672/Rim-Pn-Fusiles-Experimental>
- [26] E. M. Moreno, «Instituto Internacional de Derecho Humanitario».
- [27] Ejército de Tierra es [@EjercitoTierra], «#BuenosDías. En el empleo legítimo de la fuerza, el militar hará un uso gradual y proporcionado de la misma, de acuerdo con las reglas de enfrentamiento establecidas para las operaciones en las que participe. Art. 84 de las RR.OO. para las FA,s #SomostuEjército es <https://t.co/ot7lPqYtKu>», Twitter. Accedido: 4 de febrero de 2025. [En línea]. Disponible en: <https://x.com/EjercitoTierra/status/1656178012386099200>
- [28] somosxbox_Es, «¿Qué es Unreal Engine? Cómo funciona y por qué es tan famoso». Accedido: 4 de febrero de 2025. [En línea]. Disponible en: <https://www.somosxbox.com/que-es-unreal-engine-como-funciona-y-por-que-es-tan-famoso/>
- [29] iVisual, «[GUÍA] Programación en Unreal Engine 5 para principiantes». Accedido: 4 de febrero de 2025. [En línea]. Disponible en: <https://ivisualformacion.com/blog/tutoriales/programacion-en-unreal-engine>
- [30] «Blueprints Quick Start Guide | Unreal Engine 4.27 Documentation | Epic Developer Community», Epic Games Developer. Accedido: 4 de febrero de 2025. [En línea]. Disponible en: <https://dev.epicgames.com/documentation/en-us/unreal-engine/quick-start-guide-for-blueprints-visual-scripting-in-unreal-engine>
- [31] «Level Blueprint in Unreal Engine | Unreal Engine 5.5 Documentation | Epic Developer Community», Epic Games Developer. Accedido: 4 de febrero de 2025. [En línea]. Disponible en: <https://dev.epicgames.com/documentation/en-us/unreal-engine/level-blueprint-in-unreal-engine>
- [32] «Documentación de Unreal Engine 5.4 | Unreal Engine 5.5 Documentación | Epic Developer Community», Epic Games Developer. Accedido: 4 de febrero de 2025. [En línea]. Disponible en: <https://dev.epicgames.com/documentation/es-es/unreal-engine/unreal-engine-5-5-documentation>
- [33] «Unreal Engine's MetaHuman Creator, Analyzed and Explained». Accedido: 4 de febrero de 2025. [En línea]. Disponible en: <https://www.virtualhumans.org/article/unreal-engines-metahuman-creator-analyzed-and-explained>
- [34] «Fab», Fab.com. Accedido: 4 de febrero de 2025. [En línea]. Disponible en: <https://www.fab.com/channels/unreal-engine>
- [35] «Artificial Intelligence in Unreal Engine | Unreal Engine 5.5 Documentation | Epic Developer Community», Epic Games Developer. Accedido: 5 de febrero de 2025. [En línea]. Disponible en: <https://dev.epicgames.com/documentation/en-us/unreal-engine/artificial-intelligence-in-unreal-engine>
- [36] «Unreal Engine 4.26: Inteligência Artificial Avançada», Artificial Intelligence Application World. Accedido: 5 de febrero de 2025. [En línea]. Disponible en: <https://aiappworld.com/tutorial/unreal-engine-4-26-inteligencia-artificial-avancada/>
- [37] «Behavior Trees in Unreal Engine | Unreal Engine 5.5 Documentation | Epic Developer Community», Epic Games Developer. Accedido: 5 de febrero de 2025. [En línea]. Disponible en: <https://dev.epicgames.com/documentation/en-us/unreal-engine/behavior-trees-in-unreal-engine>
- [38] «Navigation System in Unreal Engine | Unreal Engine 5.5 Documentation | Epic Developer Community», Epic Games Developer. Accedido: 6 de febrero de 2025. [En línea]. Disponible en:

<https://dev.epicgames.com/documentation/en-us/unreal-engine/navigation-system-in-unreal-engine>

- [39] «AI Perception in Unreal Engine | Unreal Engine 5.5 Documentation | Epic Developer Community», Epic Games Developer. Accedido: 6 de febrero de 2025. [En línea]. Disponible en: <https://dev.epicgames.com/documentation/en-us/unreal-engine/ai-perception-in-unreal-engine>
- [40] «Árboles de Comportamiento para la Toma de Decisiones Dinámica de NPC e», peerdh.com. Accedido: 6 de febrero de 2025. [En línea]. Disponible en: <https://peerdh.com/es/blogs/programming-insights/behavior-trees-for-dynamic-npc-decision-making-in-unity>
- [41] «Epic Games», *Wikipedia, la enciclopedia libre*. 20 de enero de 2025. Accedido: 7 de febrero de 2025. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Epic_Games&oldid=164878202
- [42] «Fortnite OG: Drop Back into Battle Royale Chapter 1!», Epic Games' Fortnite. Accedido: 7 de febrero de 2025. [En línea]. Disponible en: <https://www.fortnite.com/news/fortnite-og-drop-back-into-battle-royale-chapter-1?lang=en-US>
- [43] «Steam», *Wikipedia, la enciclopedia libre*. 31 de enero de 2025. Accedido: 7 de febrero de 2025. [En línea]. Disponible en: <https://es.wikipedia.org/w/index.php?title=Steam&oldid=165106023>
- [44] «SteamVR en Steam». Accedido: 7 de febrero de 2025. [En línea]. Disponible en: <https://store.steampowered.com/app/250820/SteamVR/>
- [45] «SteamVR - Ya se encuentra disponible la versión beta de Inicio de SteamVR - Noticias de Steam». Accedido: 7 de febrero de 2025. [En línea]. Disponible en: <https://store.steampowered.com/news/app/250820/view/2898585530113880133>
- [46] «HTC Vive», *Wikipedia*. 25 de enero de 2025. Accedido: 7 de febrero de 2025. [En línea]. Disponible en: https://en.wikipedia.org/w/index.php?title=HTC_Vive&oldid=1271673043
- [47] «VIVE Focus Vision - Specs for the Standalone PC VR Headset | VIVE United States». Accedido: 7 de febrero de 2025. [En línea]. Disponible en: <https://www.vive.com/us/product/vive-focus-vision/specs/>
- [48] Redaccion, «HTC Vive Focus 3 el nuevo visor Standalone para empresas», Alehandoro VR. Accedido: 7 de febrero de 2025. [En línea]. Disponible en: <https://alehandorovr.com/htc-vive-focus-3-el-nuevo-visor-standalone-para-empresas/>
- [49] «Controller for VIVE XR/Focus Series (R)». Accedido: 10 de febrero de 2025. [En línea]. Disponible en: <https://www.vive.com/eu/accessory/vive-controller-right/>
- [50] «Controller for VIVE XR/Focus Series (R)». Accedido: 10 de febrero de 2025. [En línea]. Disponible en: <https://business.vive.com/hk/accessory/vive-controller-right/>
- [51] «EVGA - Artículos - Evga GeForce RTX™ 3080 12GB», EVGA. Accedido: 10 de febrero de 2025. [En línea]. Disponible en: <https://latam.evga.com/articles/01533/evga-geforce-rtx-3080-12gb/Default.asp?>
- [52] «Toma-de-decisiones-militares-operaciones-de-seguridad-1.pdf». Accedido: 28 de febrero de 2025. [En línea]. Disponible en: <https://www.jid.org/wp-content/uploads/2021/06/Toma-de-decisiones-militares-operaciones-de-seguridad-1.pdf>
- [53] «AI Debugging in Unreal Engine | Unreal Engine 5.5 Documentation | Epic Developer Community», Epic Games Developer. Accedido: 6 de marzo de 2025. [En línea]. Disponible en: <https://dev.epicgames.com/documentation/en-us/unreal-engine/ai-debugging-in-unreal-engine>

ANEXO I: IMPLICACIONES SOCIALES, Y/O ECONÓMICAS, Y/O AMBIENTALES

El proyecto desarrollado tiene implicaciones significativas en los ámbitos social, económico y medioambiental, particularmente dentro de la formación militar las escuelas y centros de adiestramiento e instrucción. Desde una perspectiva social, la implementación de las ROE en un entorno virtual inmersivo y realista, donde se simulan operaciones en escenarios urbanos bajo la aplicación de decisiones estratégicas, mejora la preparación de los alumnos de Infantería de Marina. Esta herramienta potencia la capacidad de identificación de amenazas y no combatientes, fomentando decisiones responsables y éticas bajo las premisas del Derecho Internacional Humanitario. La simulación de situaciones de estrés en un entorno seguro permite una mayor familiarización con la gestión de situaciones complejas que posteriormente pueden tener repercusión directa en operaciones reales, minimizando el riesgo de daños colaterales y mejorando la seguridad de la población civil en operaciones futuras.

Desde el punto de vista económico, la implementación de la realidad virtual en la formación ofrece una alternativa rentable al adiestramiento convencional. Tradicionalmente, los alumnos de la Escuela Naval Militar realizan su instrucción MOUT en PICOZU, situado a 8,3 km de la Escuela, lo que implica un gasto considerable en transporte, material, munición de fogeo y desgaste de equipo. El uso de la simulación permite reducir el número de desplazamientos y la cantidad de prácticas físicas, optimizando los recursos logísticos y financieros al disminuir la frecuencia de estas salidas sin eliminar su importancia dentro del plan de estudios.

A nivel medioambiental, la reducción del uso de autobuses y vehículos militares, así como la menor necesidad de consumir munición real o de fogeo y otros materiales desechables, contribuye a disminuir las emisiones de gases contaminantes y la producción de residuos. Esta iniciativa respalda la tendencia hacia un modelo de formación más sostenible y responsable, reduciendo la huella ecológica de las actividades de adiestramiento sin comprometer la calidad del aprendizaje.

En definitiva, la integración de este simulador en el itinerario formativo no solo mejora la eficiencia operativa y reduce costes, sino que también promueve una mayor conciencia medioambiental en la instrucción militar, contribuyendo a una formación más avanzada y comprometida con la sostenibilidad.

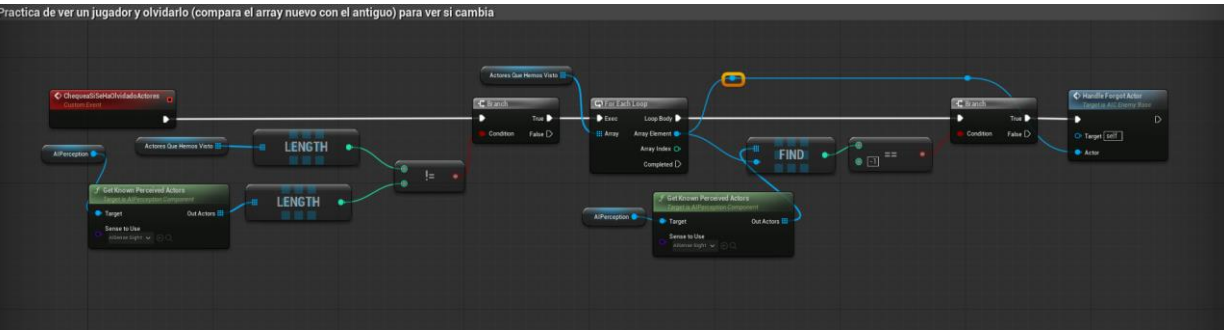
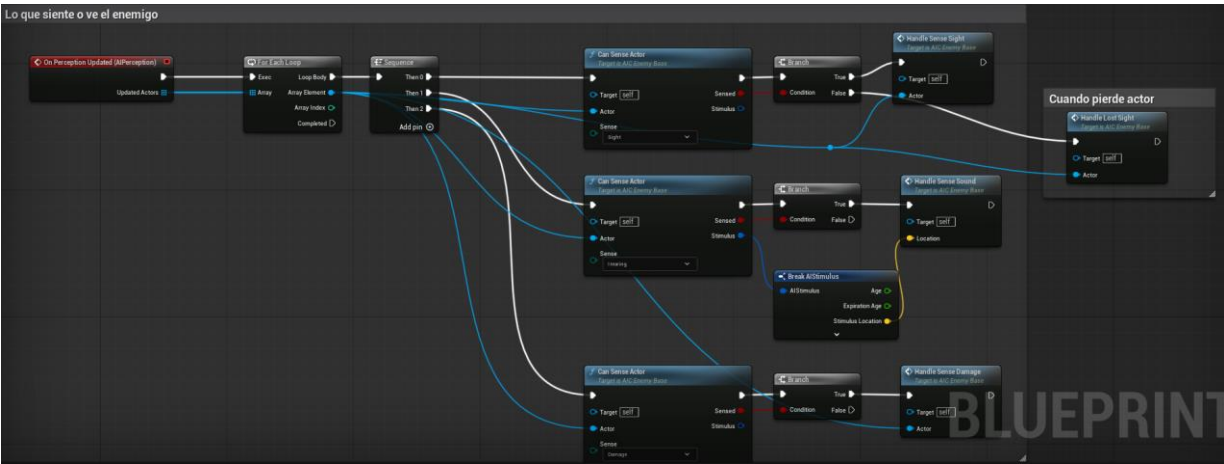
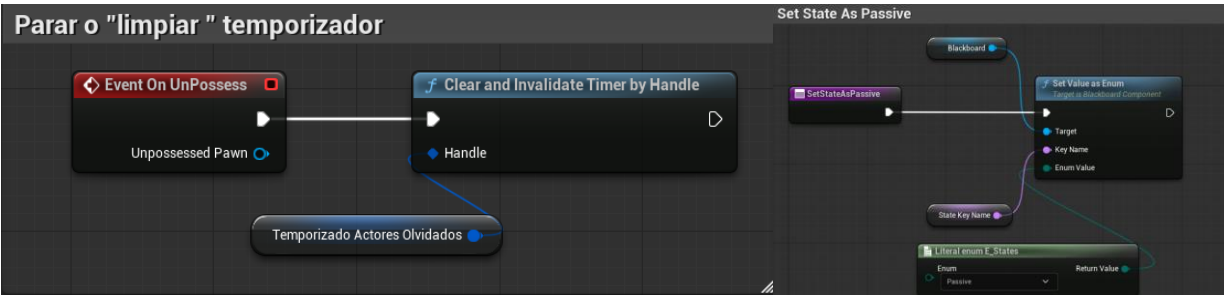
ANEXO II: REFLEXIONES ÉTICAS Y SOCIALES

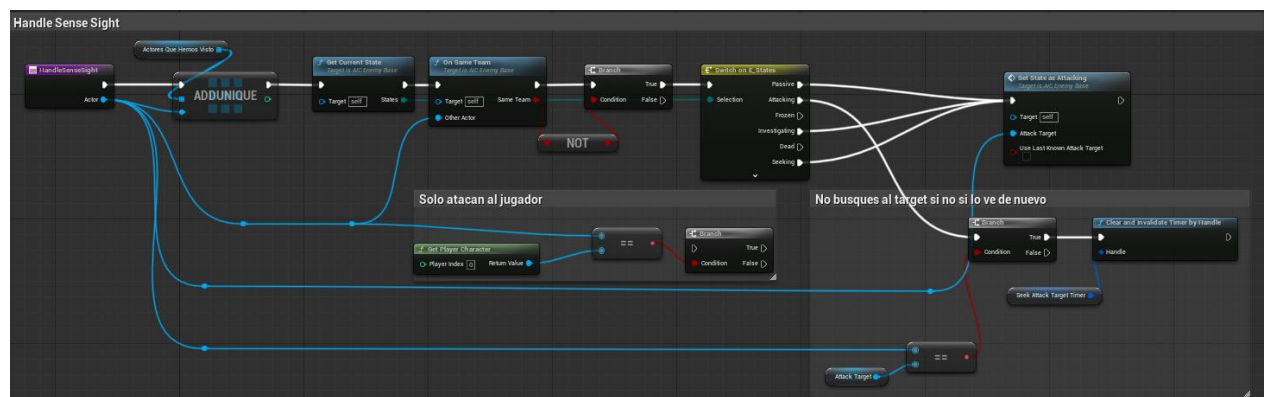
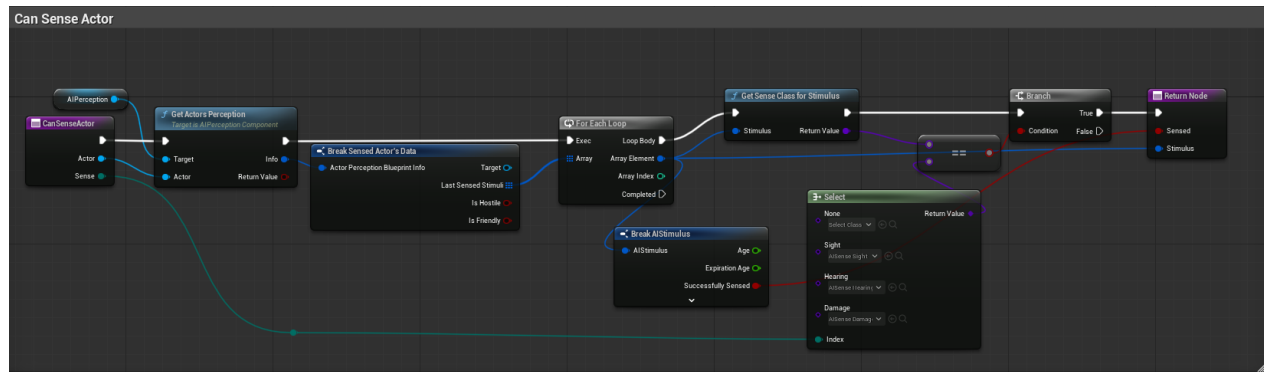
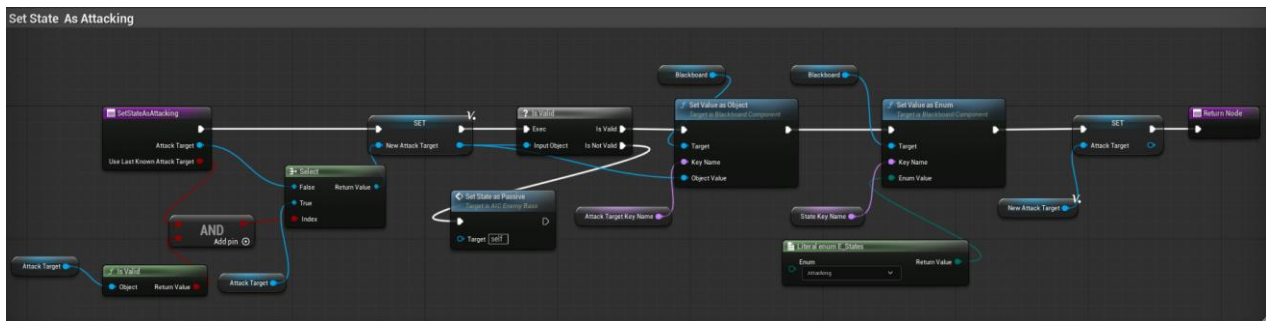
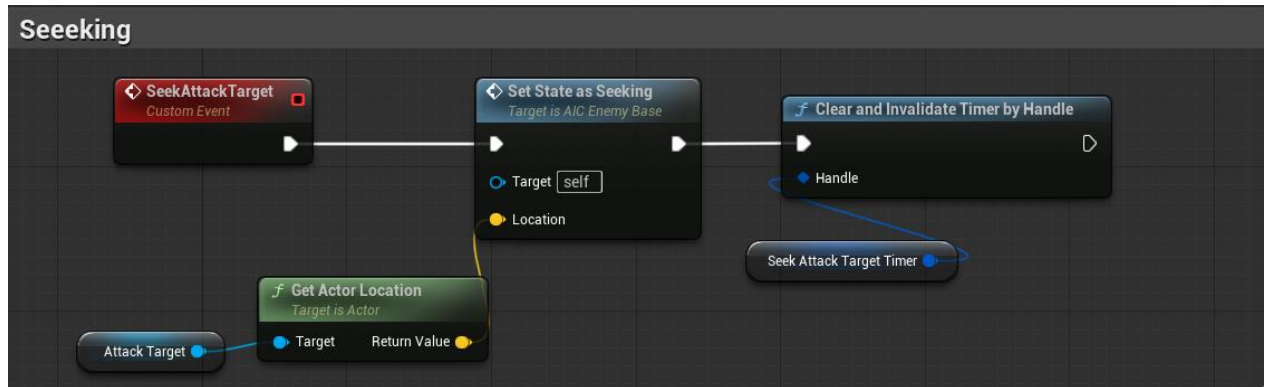
La implementación de un simulador de combate en población mediante RV, como el desarrollado en este trabajo, tiene implicaciones sociales directas para las FAS y su preparación en conflictos modernos. La posibilidad de recrear de manera realista entornos urbanos con presencia de civiles, combatientes encubiertos y escenarios tridimensionales complejos (calles estrechas, edificios y pasajes subterráneos), permite al personal militar entrenar para situaciones reales en las que la seguridad de la población es prioritaria. En el contexto de operaciones MOUT, donde la descentralización de la cadena de mando es común, esta simulación facilita la práctica de la toma de decisiones autónoma y responsable a nivel de pelotón o sección, reforzando la importancia de la coordinación táctica entre las unidades y la protección de los no combatientes.

Además, la simulación virtual contribuye a mejorar la conciencia situacional de los usuarios en entornos urbanos densos, donde la gestión simultánea de misiones humanitarias y de seguridad es crítica. Este entorno controlado permite al usuario aprender a gestionar el uso gradual y proporcionado de la fuerza, alineado con las ROE que priorizan la protección de la población civil. Asimismo, la flexibilidad de estos entornos virtuales permite replicar escenarios específicos de interés estratégico, contribuyendo a un adiestramiento más contextualizado y sensible a las realidades sociales de los conflictos contemporáneos.

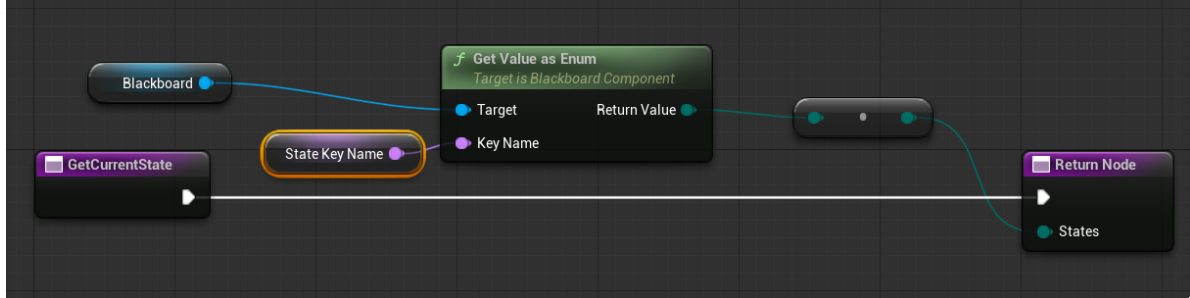
Desde un punto de vista ético, el simulador recrea dilemas que el personal militar debe afrontar durante operaciones en zonas urbanas, como la distinción entre combatientes y civiles en entornos donde las amenazas se camuflan entre la población. El hecho de entrenar la toma de decisiones bajo ROE y en cumplimiento del DIH permite preparar al usuario para minimizar los daños colaterales y garantizar la proporcionalidad en el uso de la fuerza. No obstante, el desarrollo de inteligencia artificial que simula comportamientos humanos (como enemigos que se rinden o civiles que huyen o interfieren en la operación) obliga a reflexionar sobre la responsabilidad del usuario en el entorno virtual y cómo este entrenamiento influye en la toma de decisiones en situaciones reales. Existe un riesgo potencial de desensibilización o automatización de respuestas si no se acompaña este proceso de una adecuada formación ética. Además, al tratarse de una simulación inmersiva, es necesario evaluar cómo afecta emocionalmente al usuario, tanto en su percepción del combate como en la gestión de la violencia en el contexto de ROE que restringen el uso indiscriminado de la fuerza

ANEXO III: BLOQUES DE CÓDIGO DEL AICONTROLLER

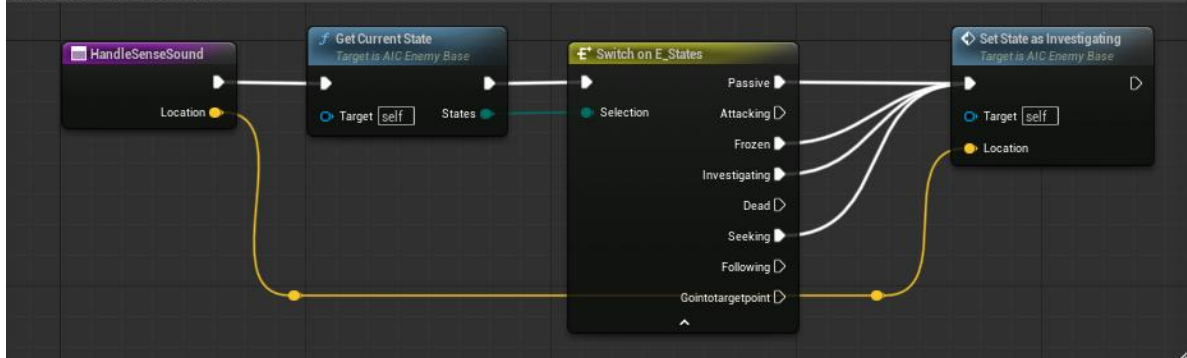




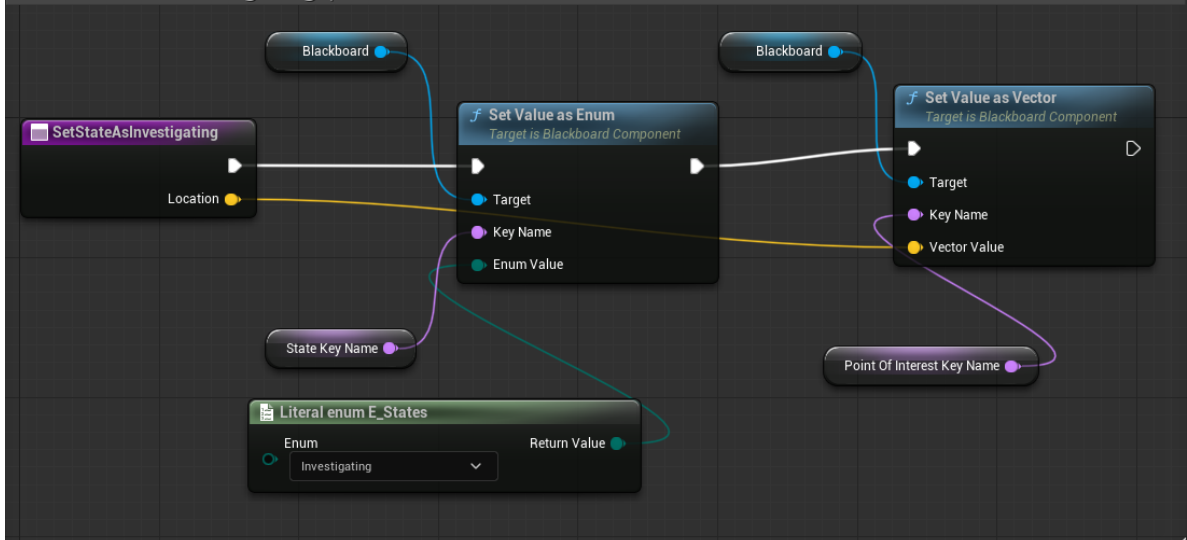
Adquirir estado (Get Current State)

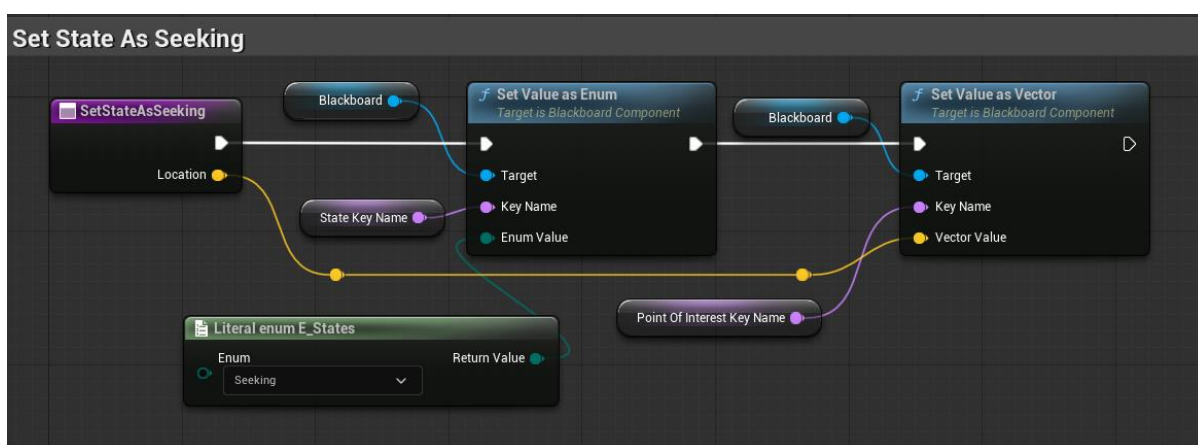
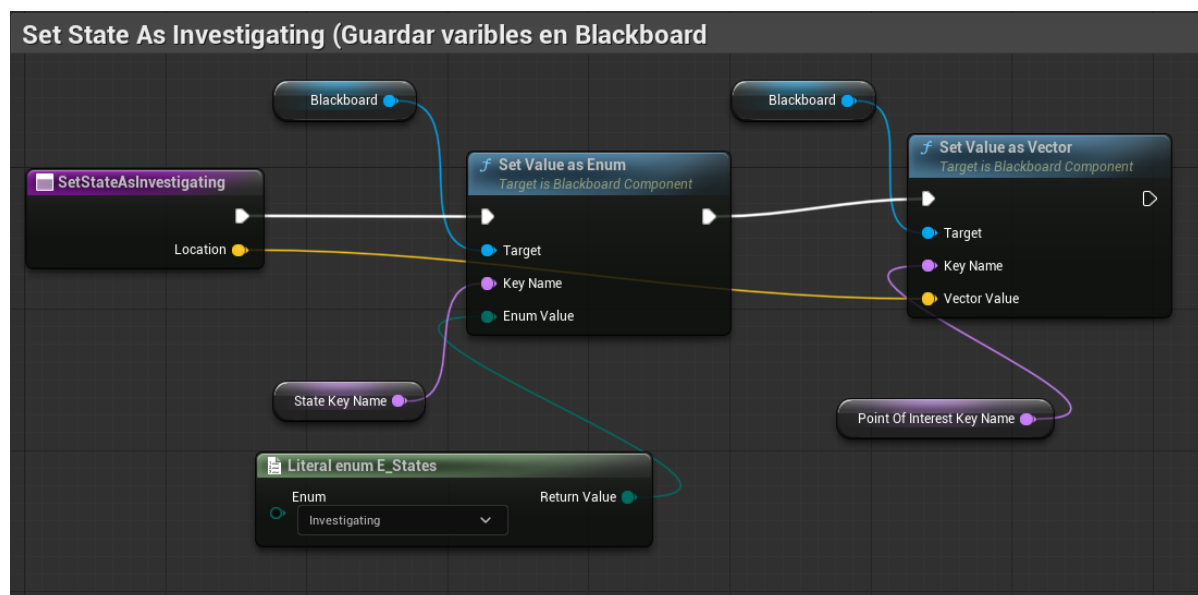
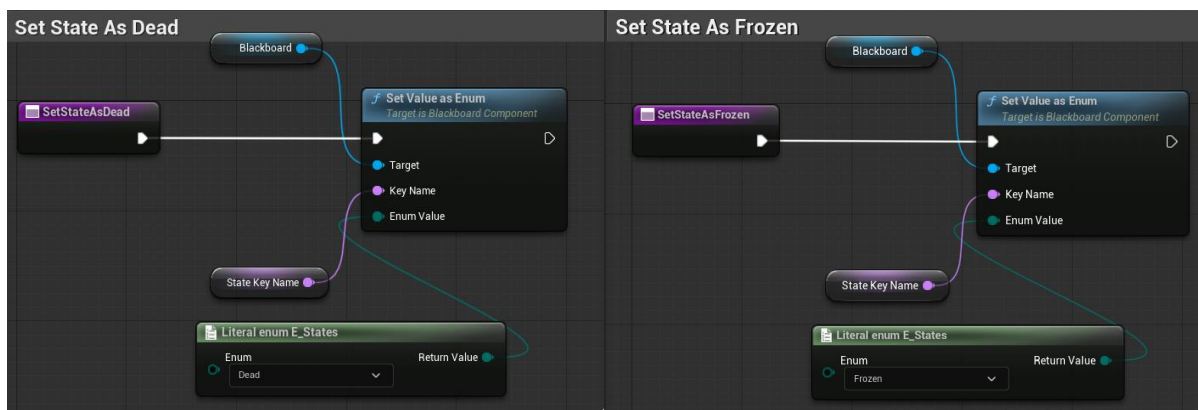


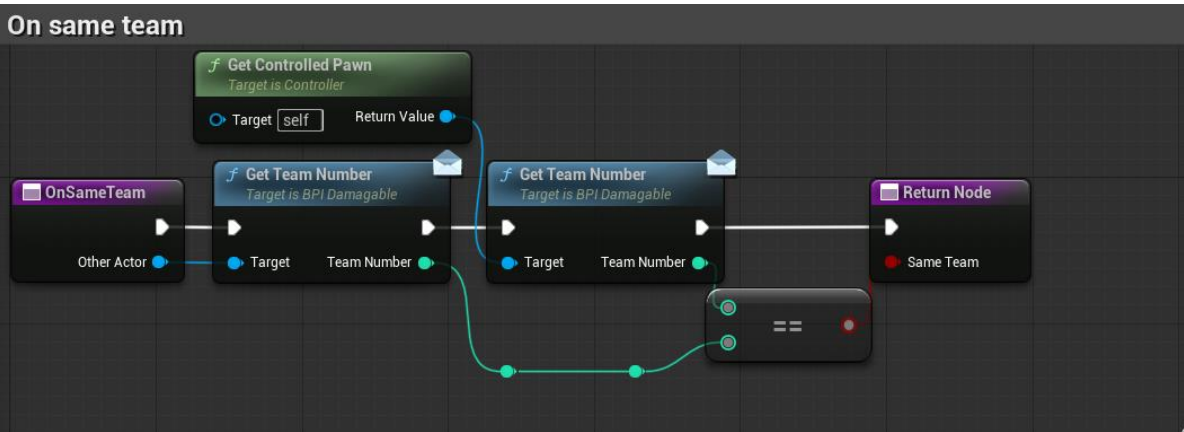
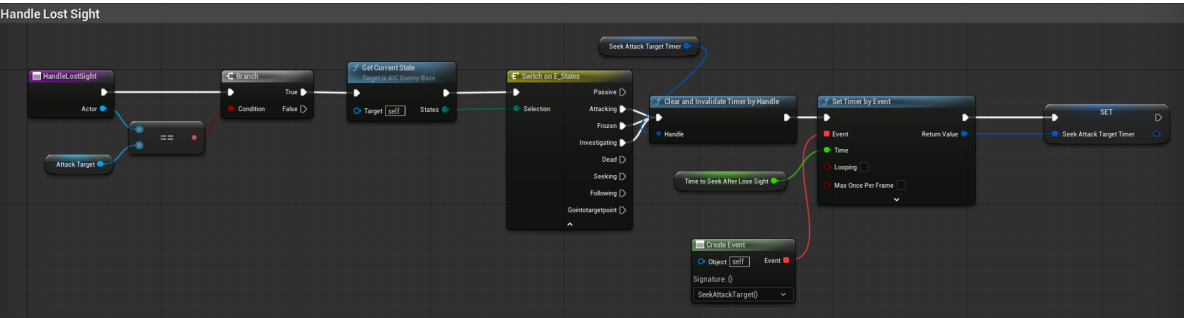
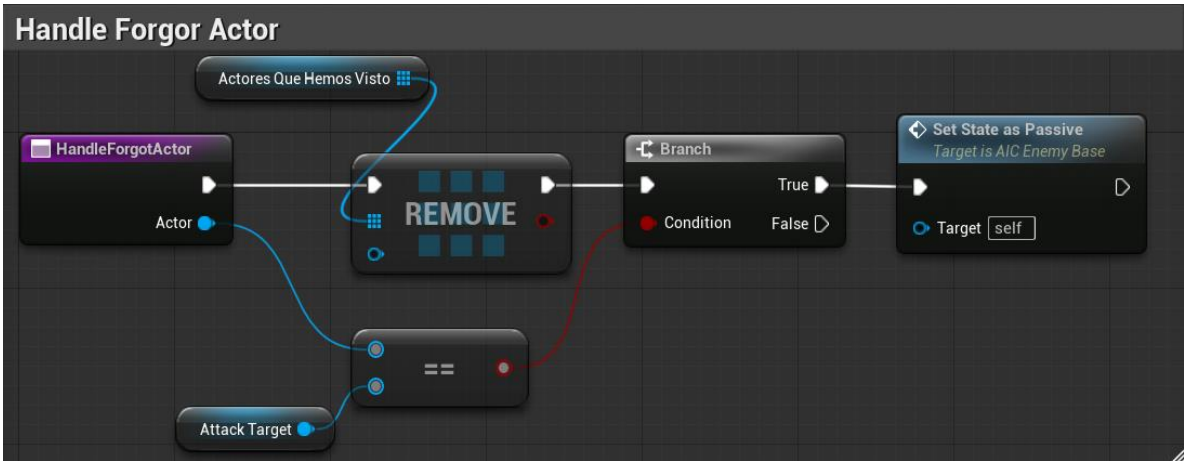
Handle Sense Sound



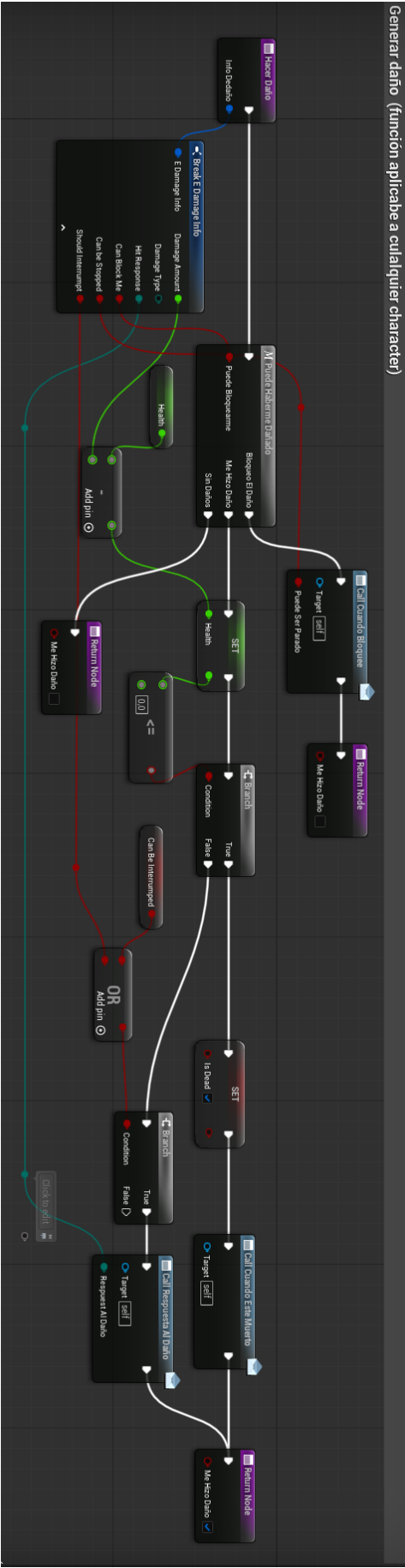
Set State As Investigating (Guardar variables en Blackboard)





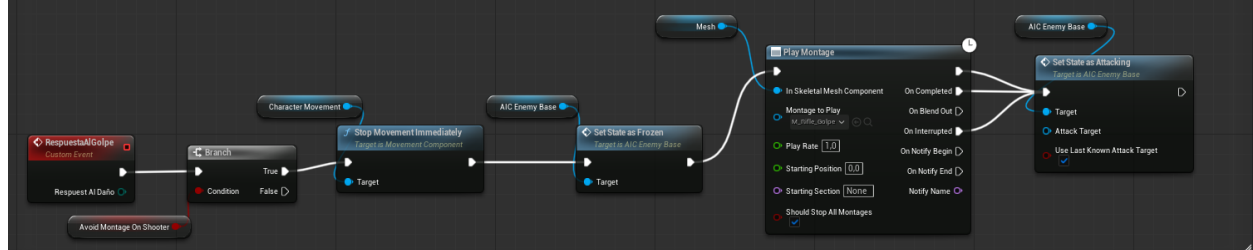


ANEXO IV: BLOQUES DE CÓDIGO DEL SISTEMA DE DAÑO

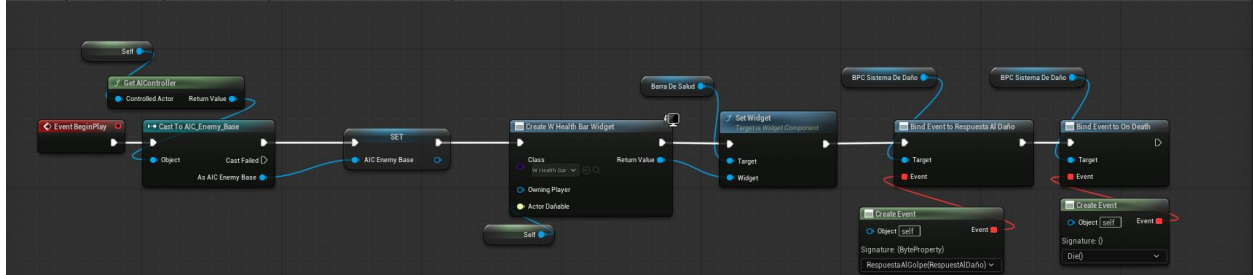


ANEXO IV: BLOQUES DE CÓDIGO DE ENEMIGOS

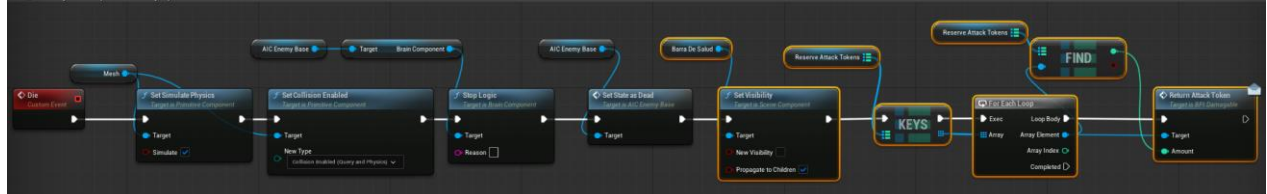
BP_EnergyBase (Event Graph)- Reacción al daño



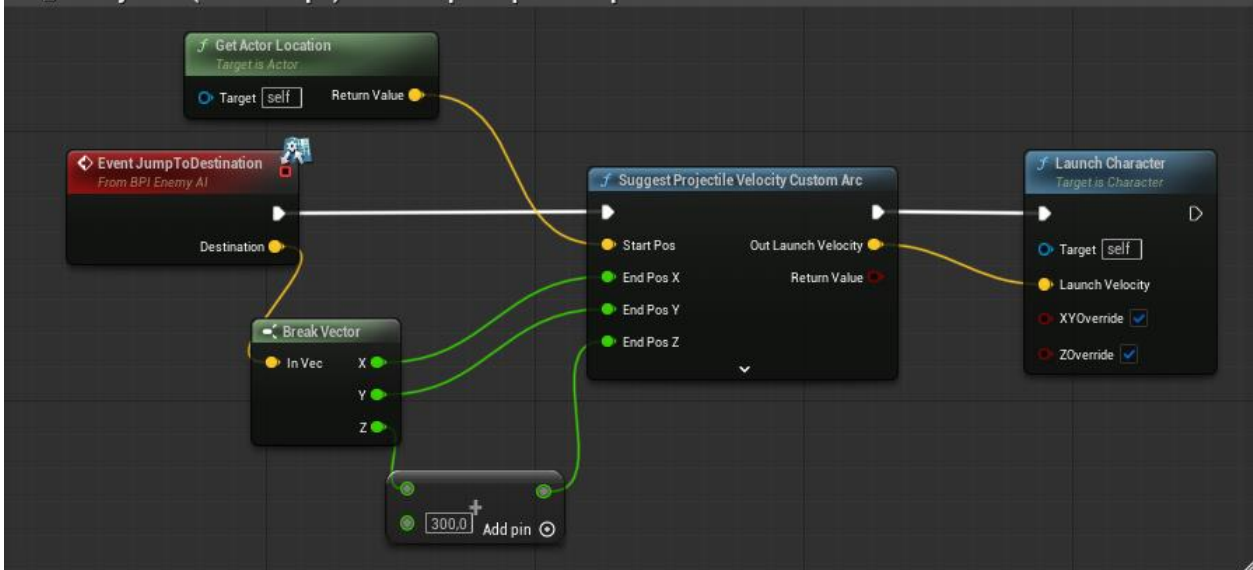
BP_EnergyBase (Event Graph)- Crear eventos y establecer la variable AIC Enemy Base

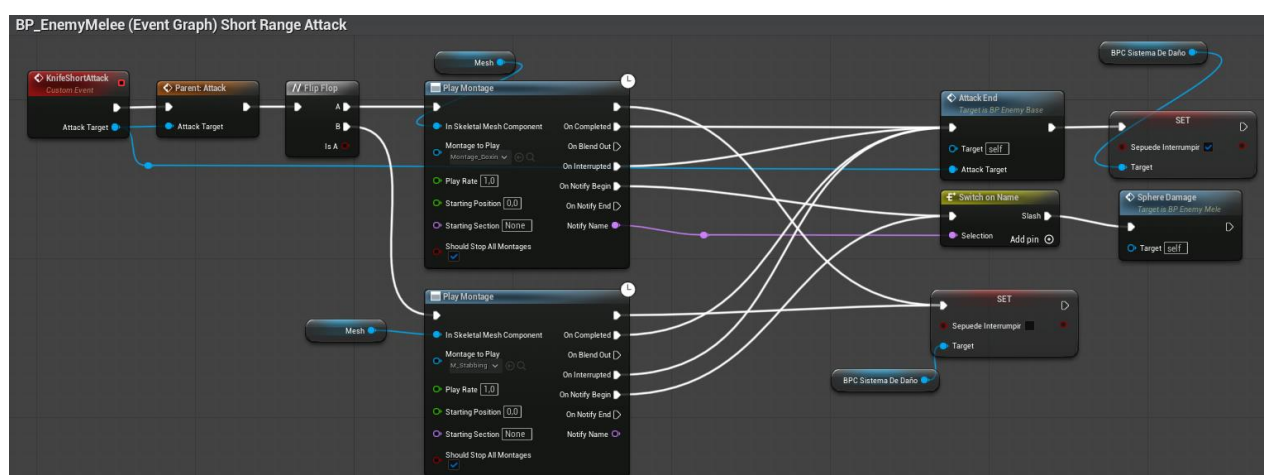
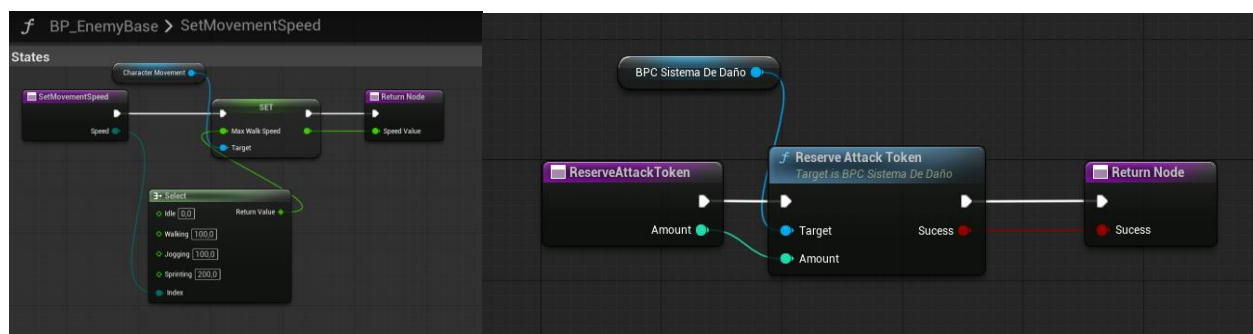
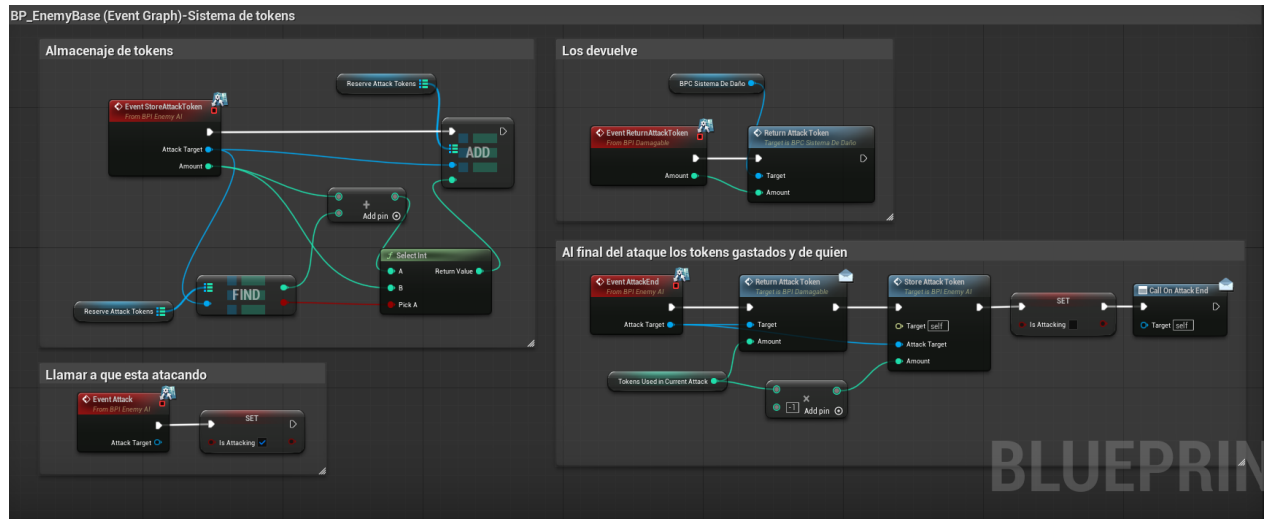


BP_EnergyBase (Event Graph)- Cuando muere

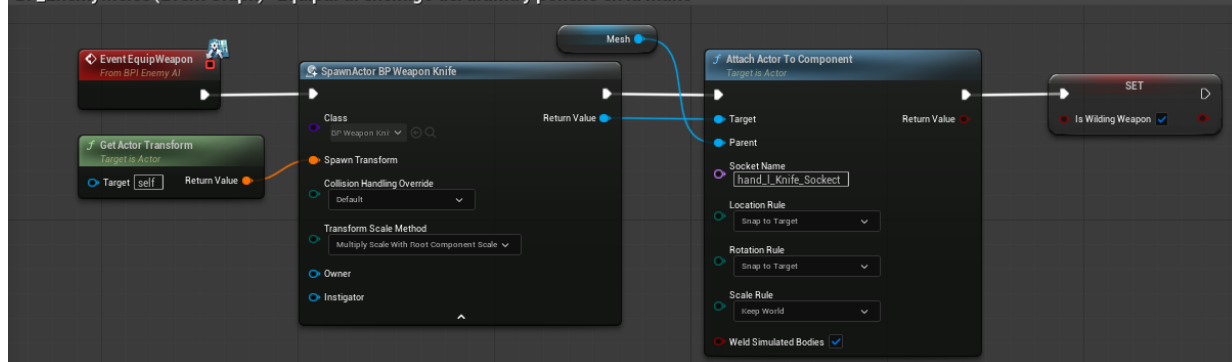


BP_EnergyBase (Event Graph)-Funcion para que salte por los navlinks

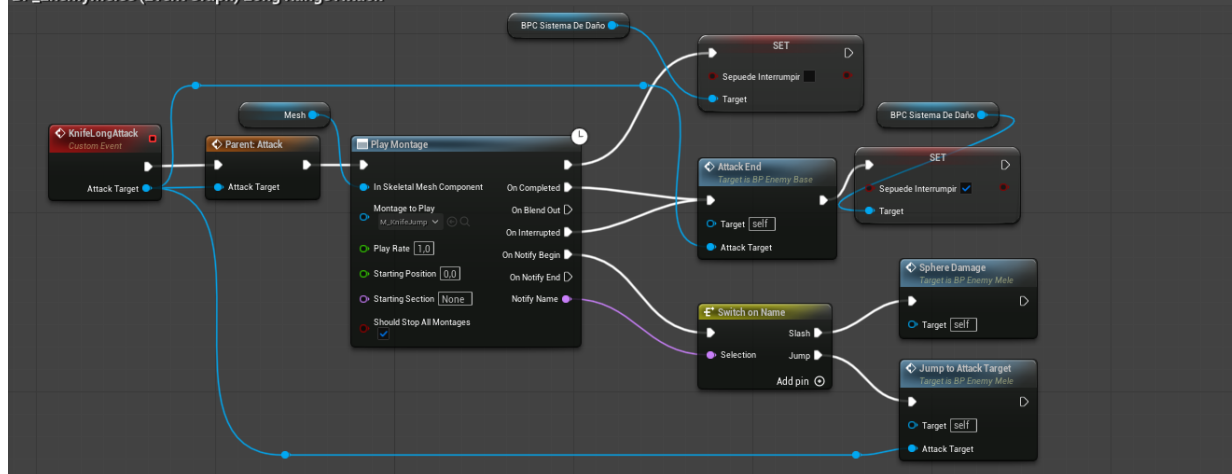




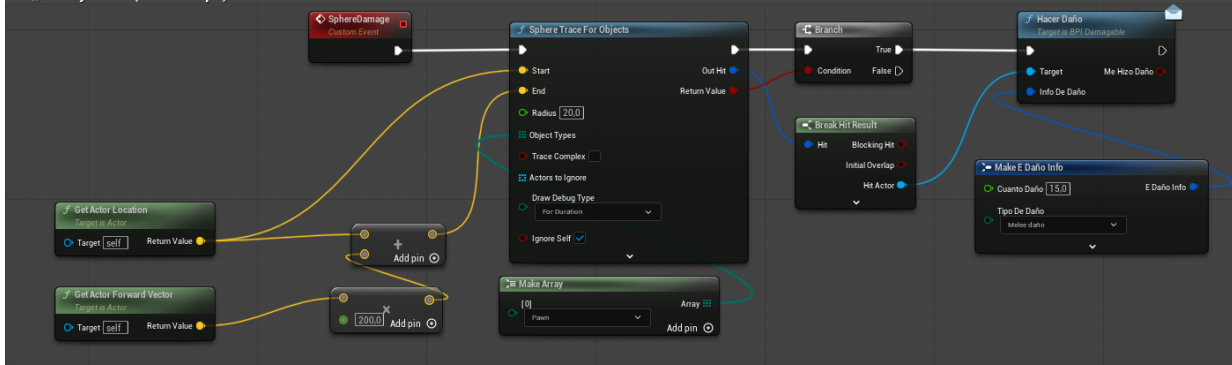
BP_EnemyMelee (Event Graph)- Equipar al enemigo del arma y ponerlo en la mano



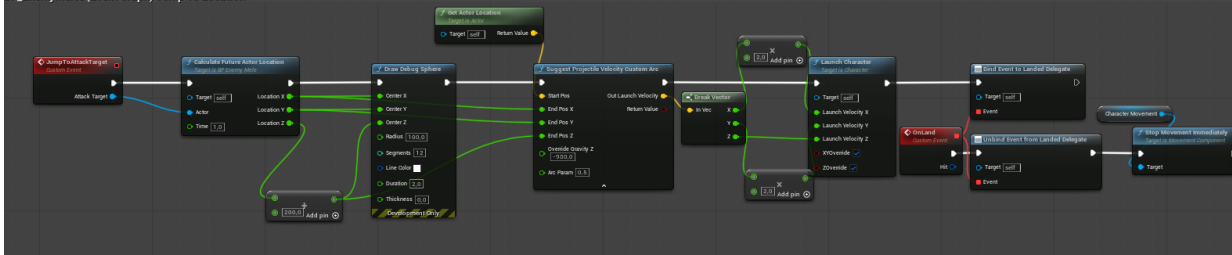
BP_EnemyMelee (Event Graph) Long Range Attack

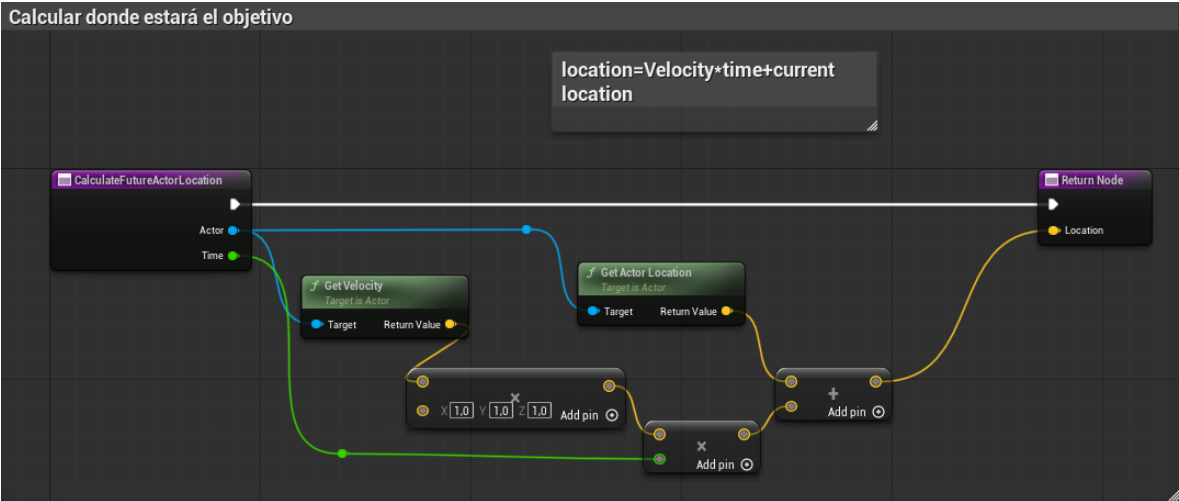


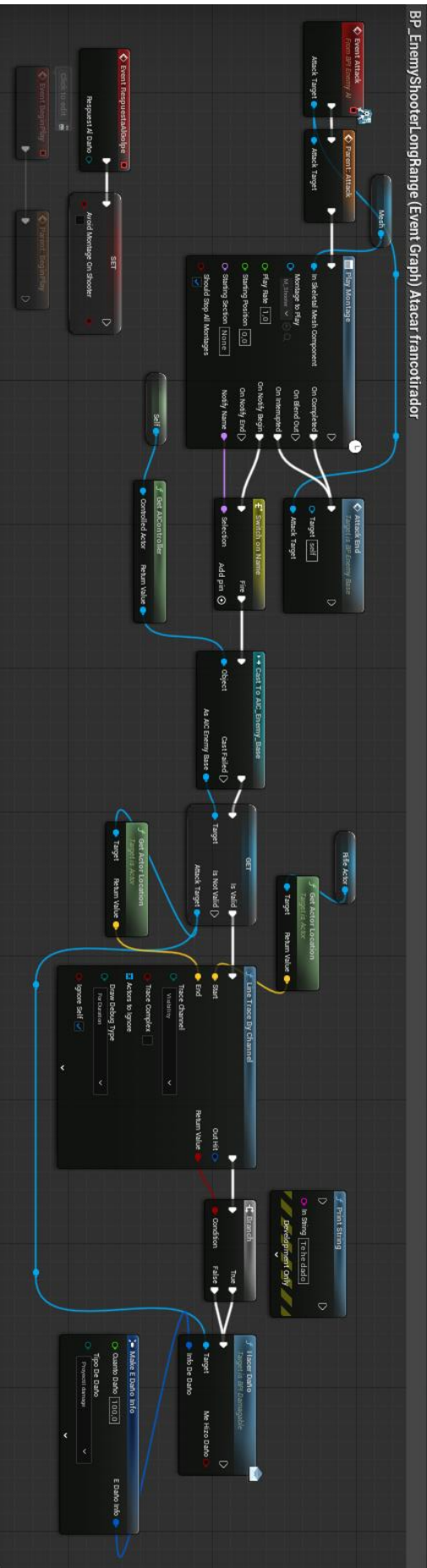
BP_EnemyMelee (Event Graph) Esfera de daño

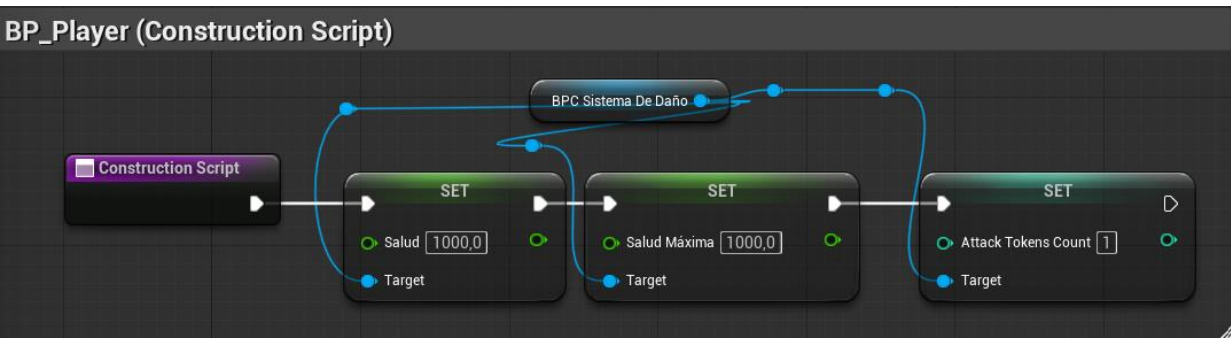
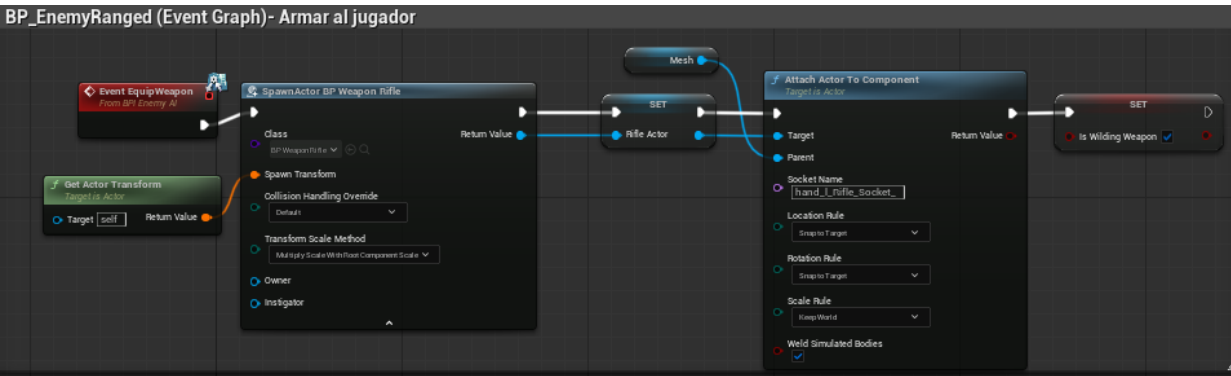
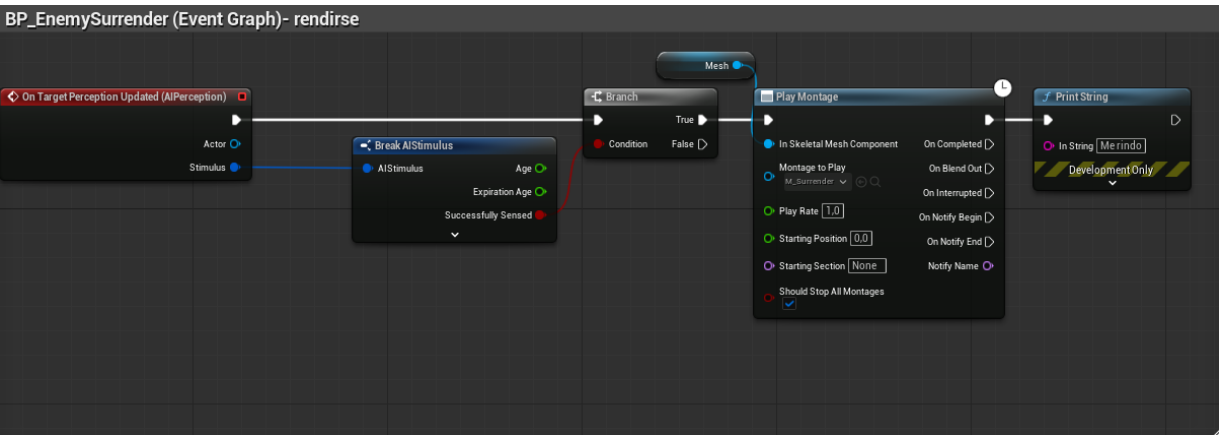
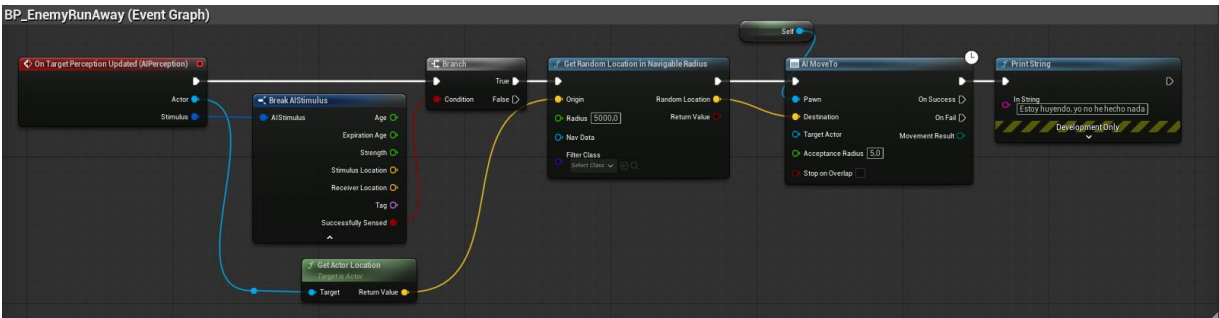


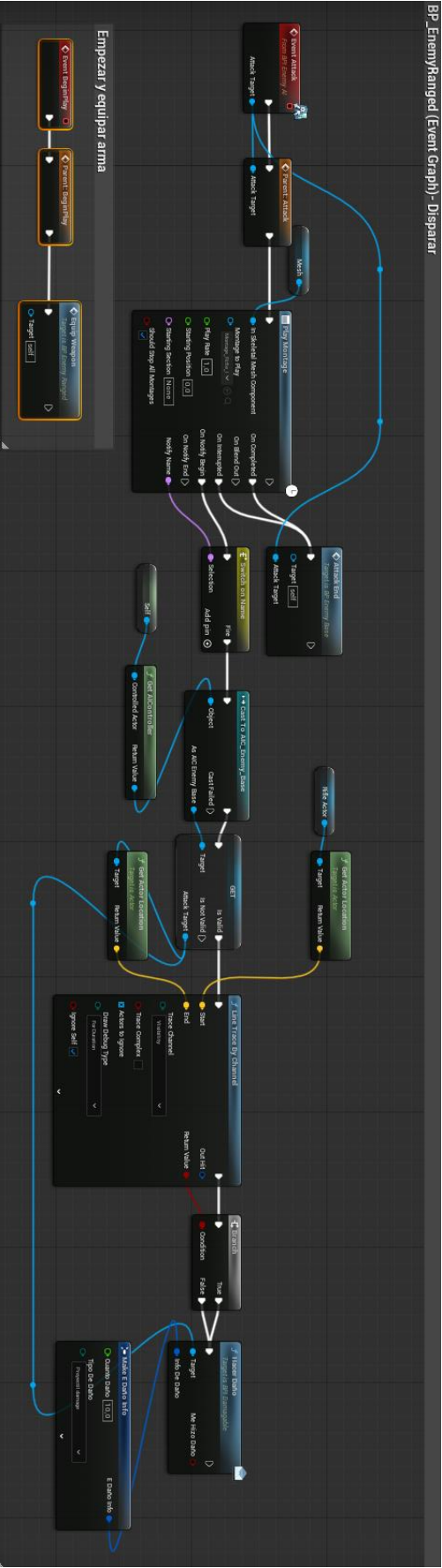
BP_EnemyMelee (Event Graph) Jump To Location



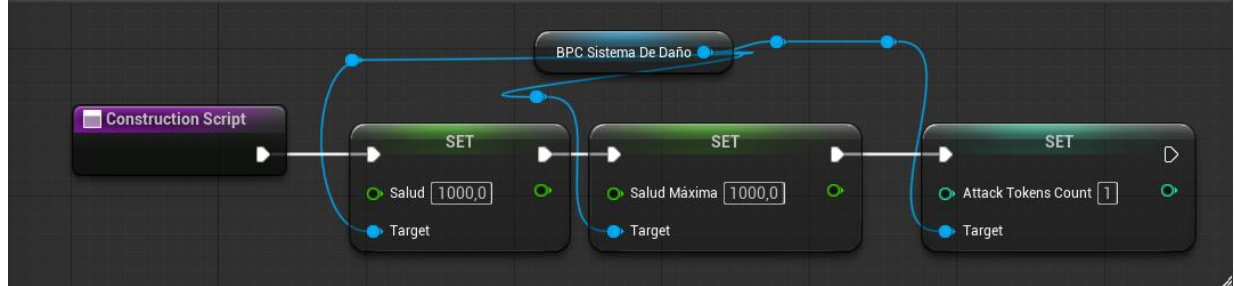




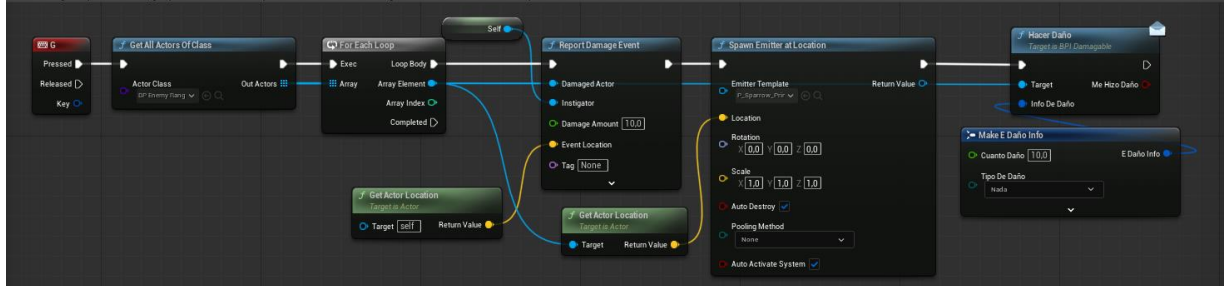




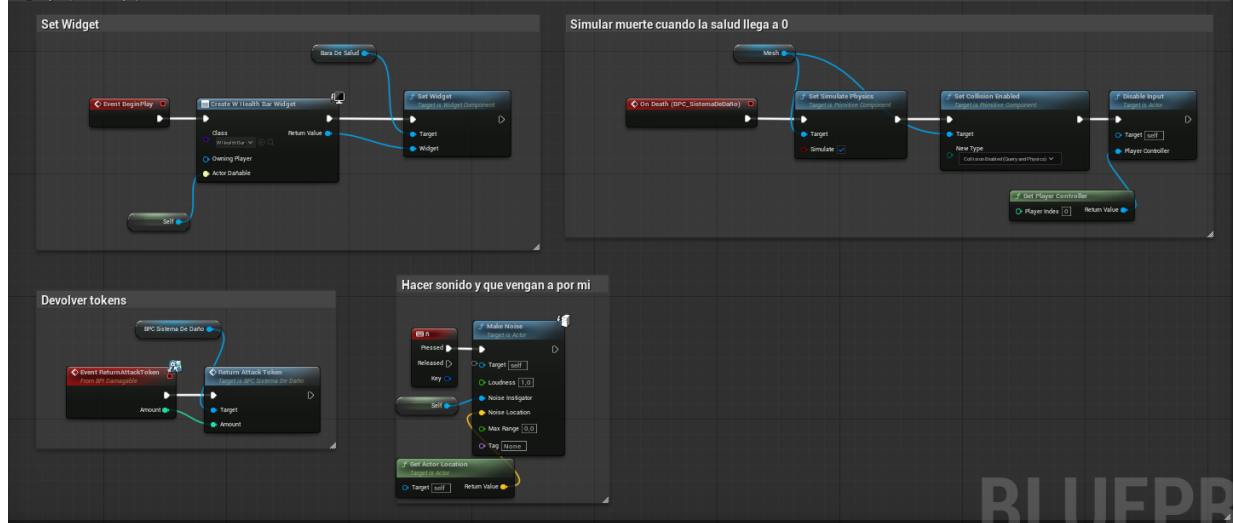
BP_Player (Construction Script)



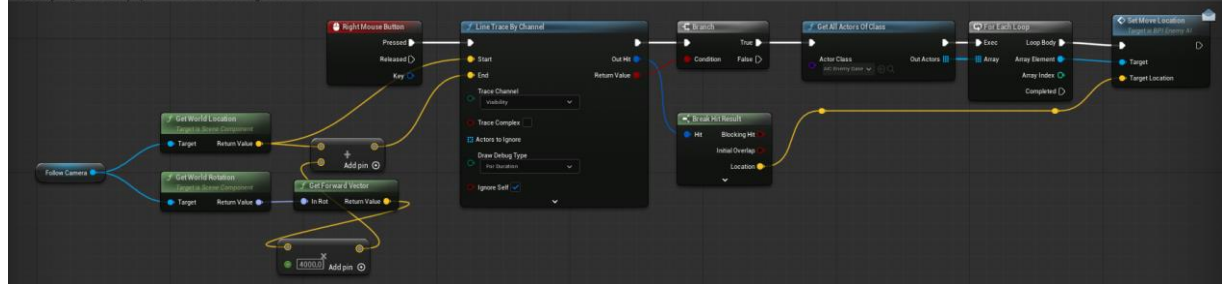
BP_Player (Event Graph)-Hacer daño (En realidad Virtual puesto a la BP_Pistol)



BP_Player (Event Graph)- Funcionalidades variadas

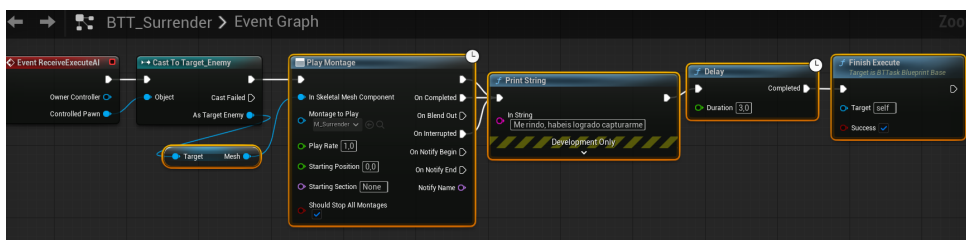
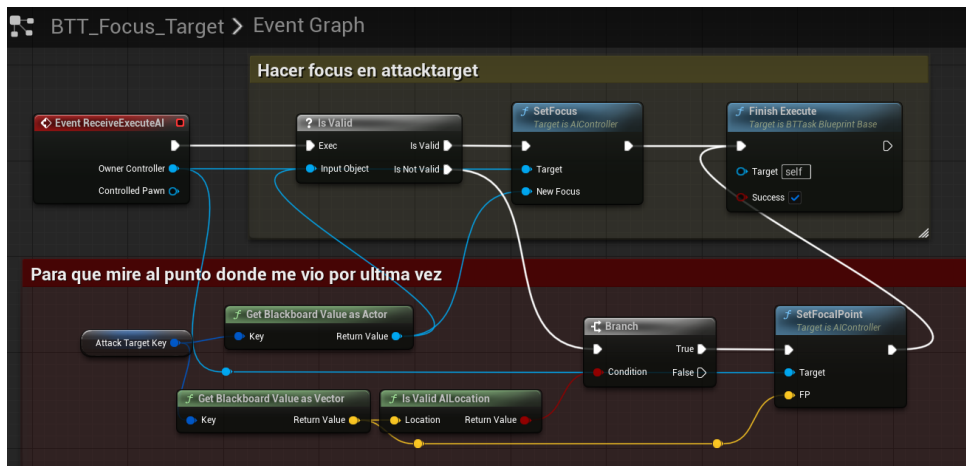
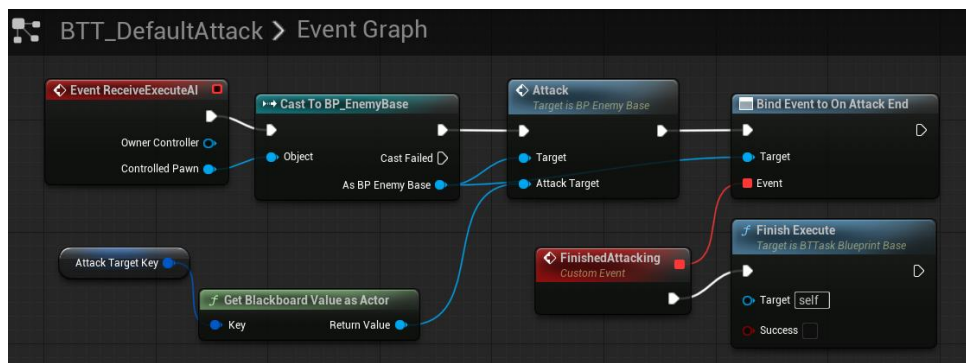
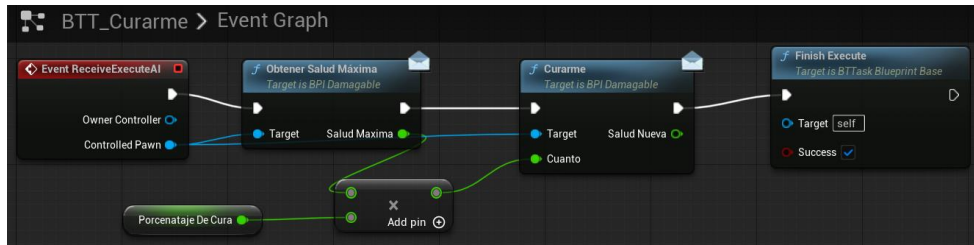


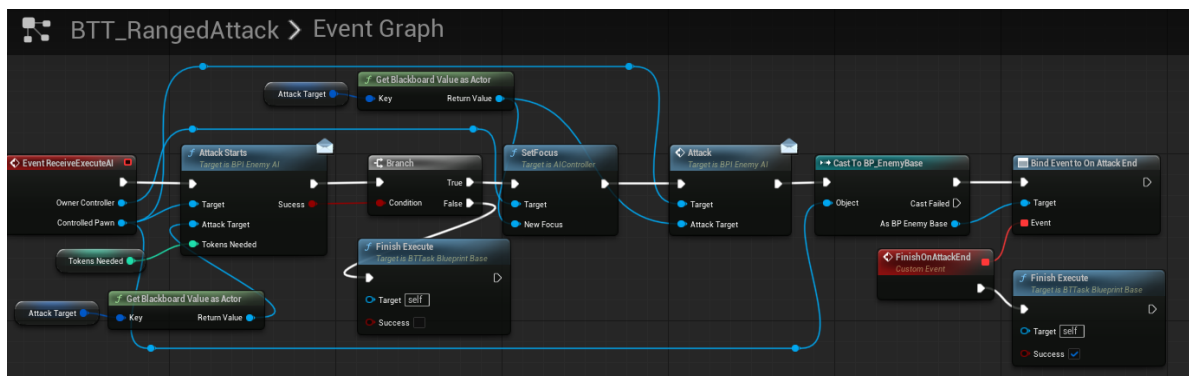
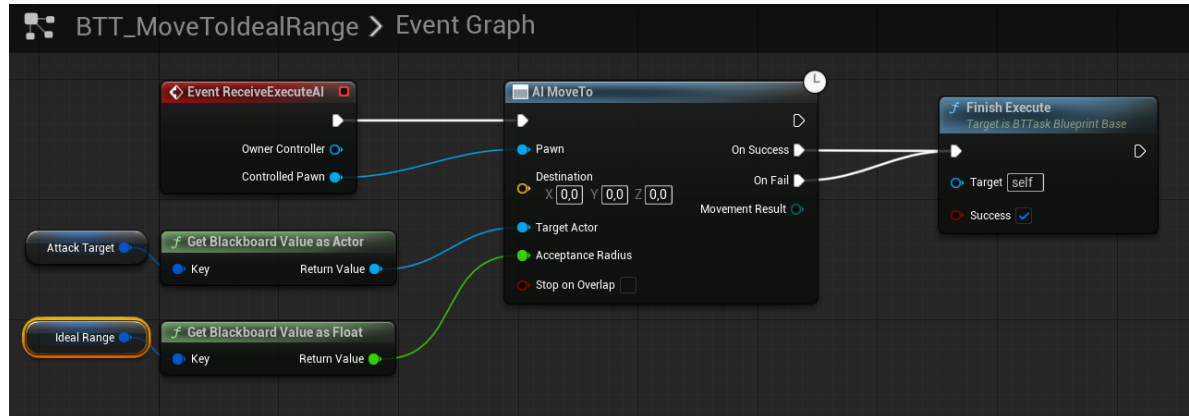
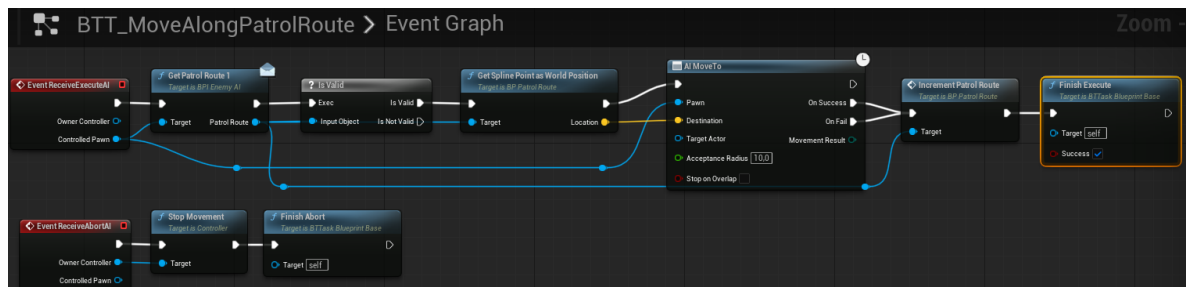
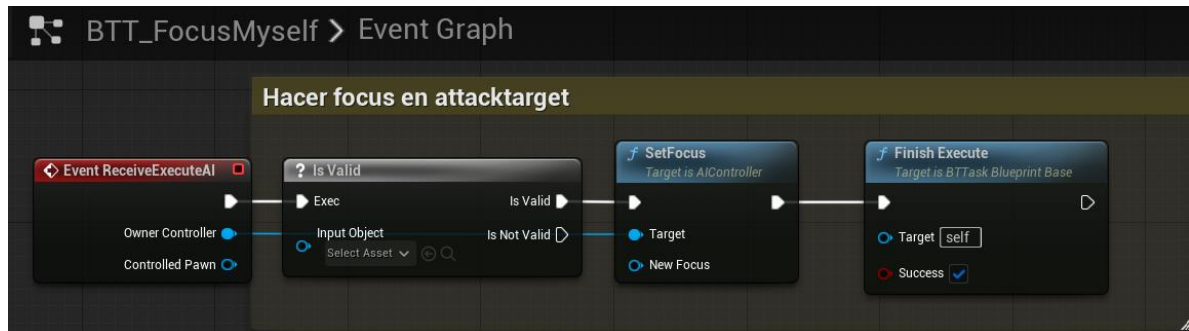
BP_Player (Event Graph)-Decirle a los enemigos dodne ir

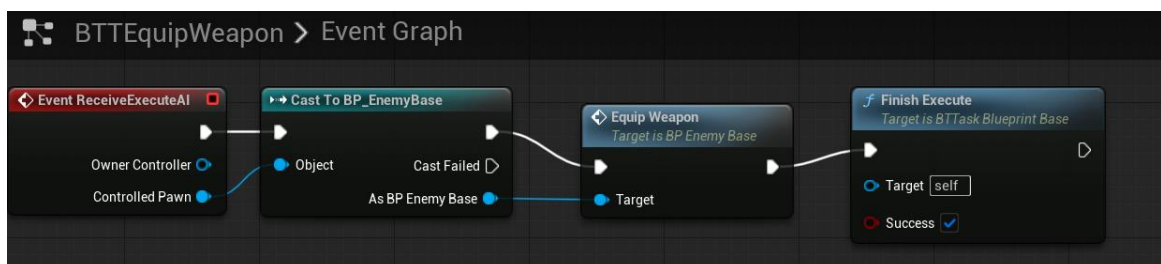
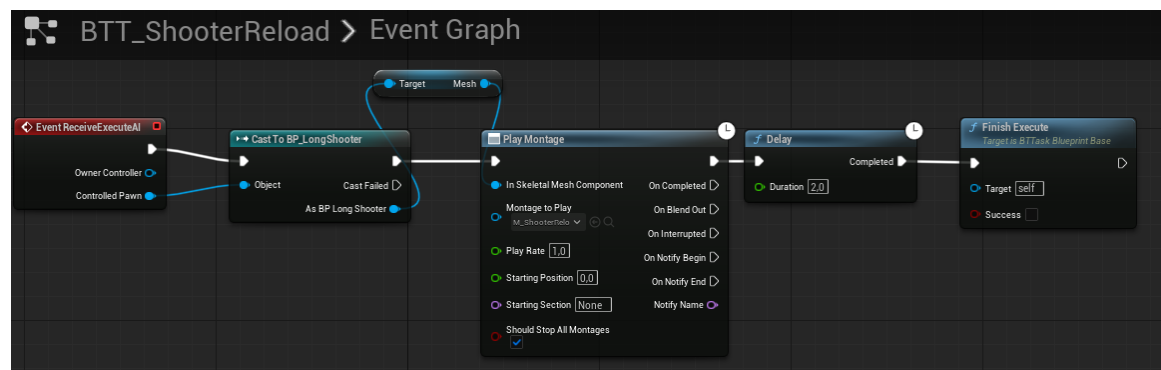
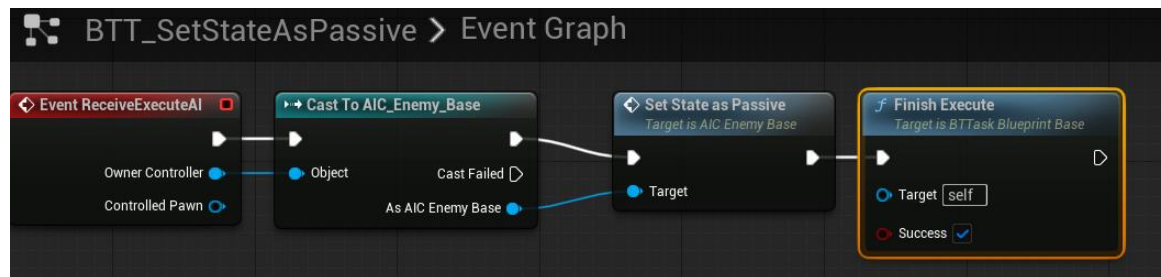
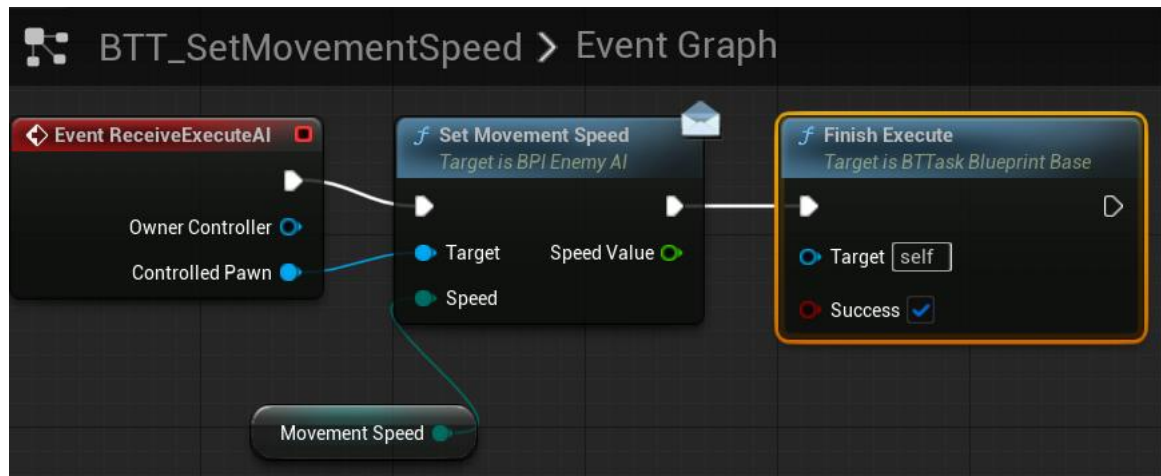


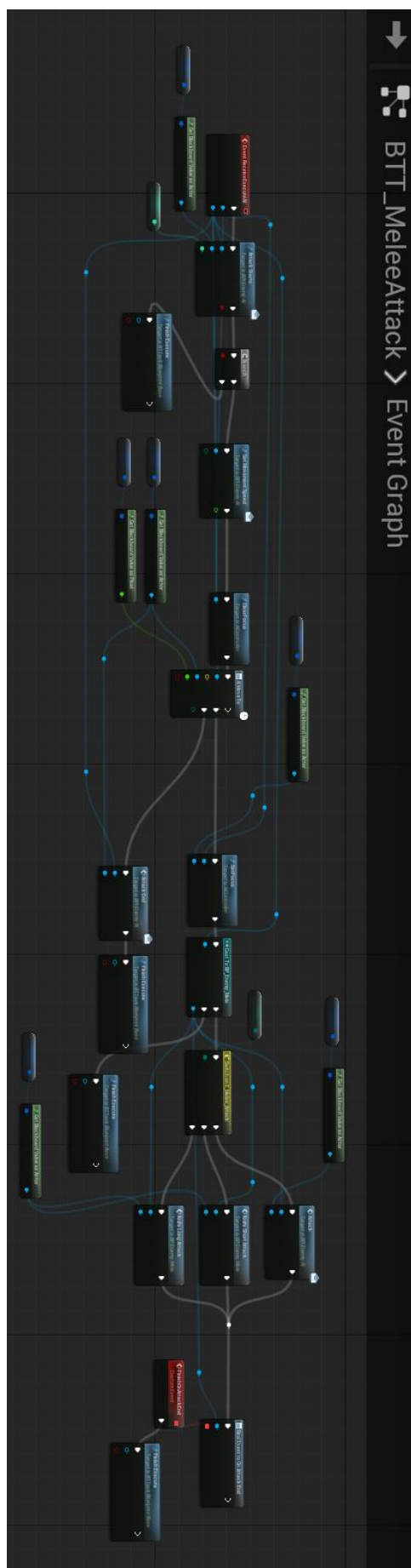
ANEXO V: BLOQUES DE CÓDIGO DE BEHAVIOR TREES

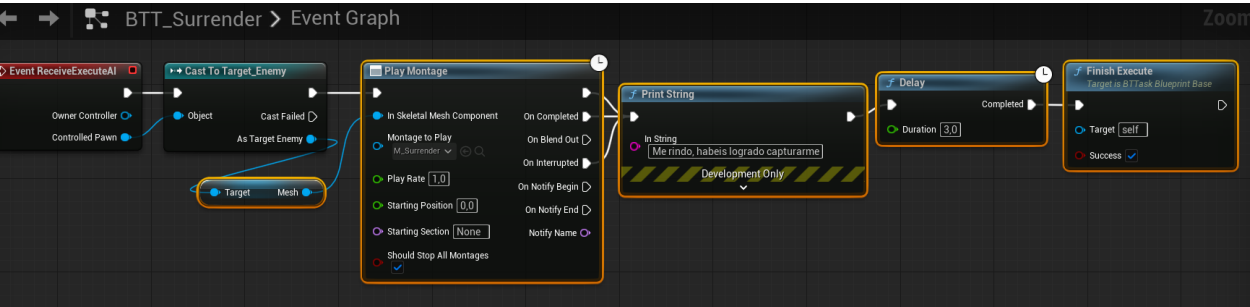
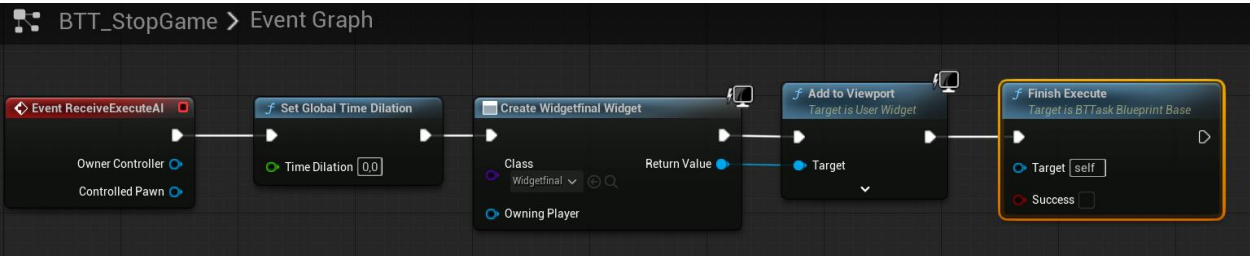
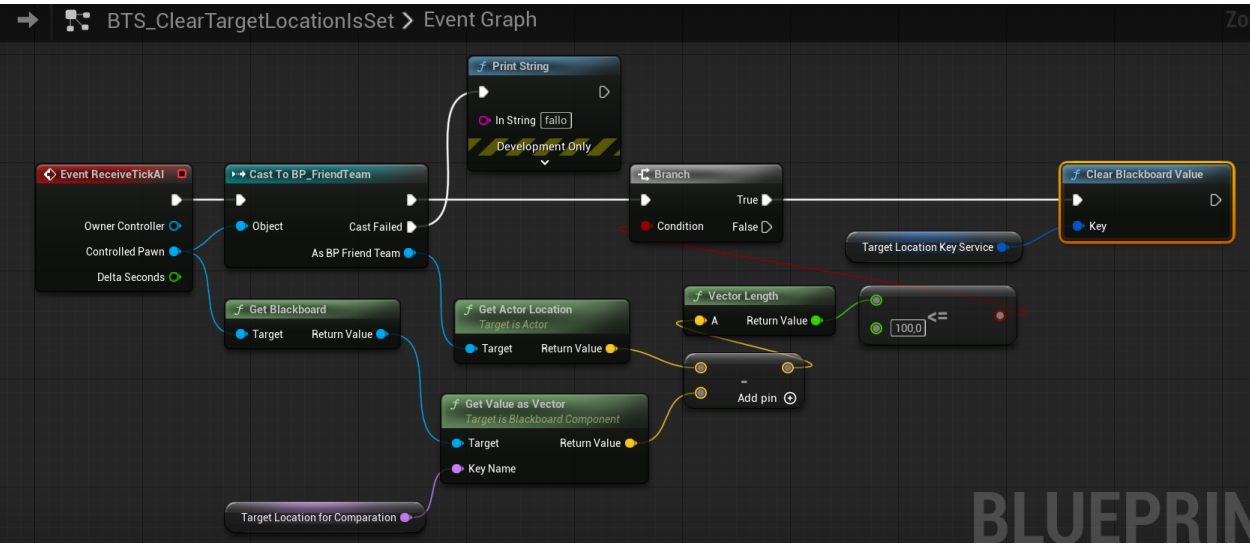
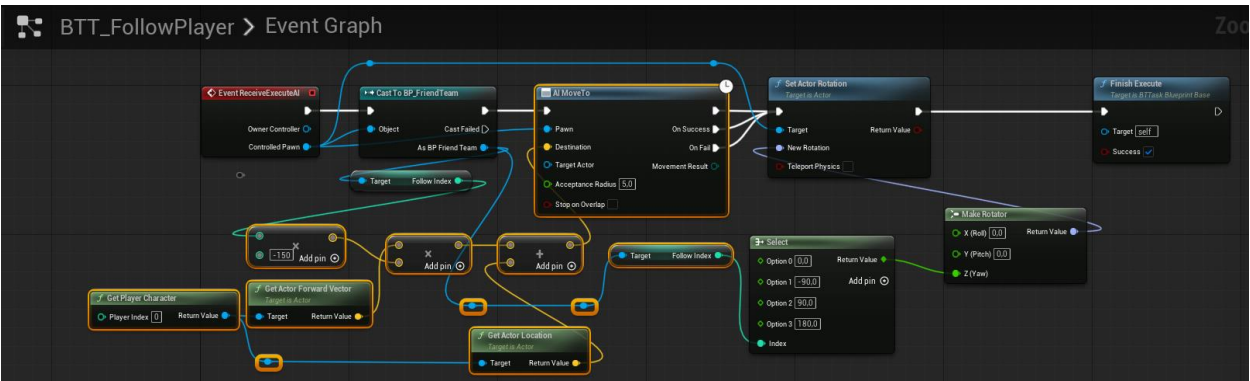
TASKS



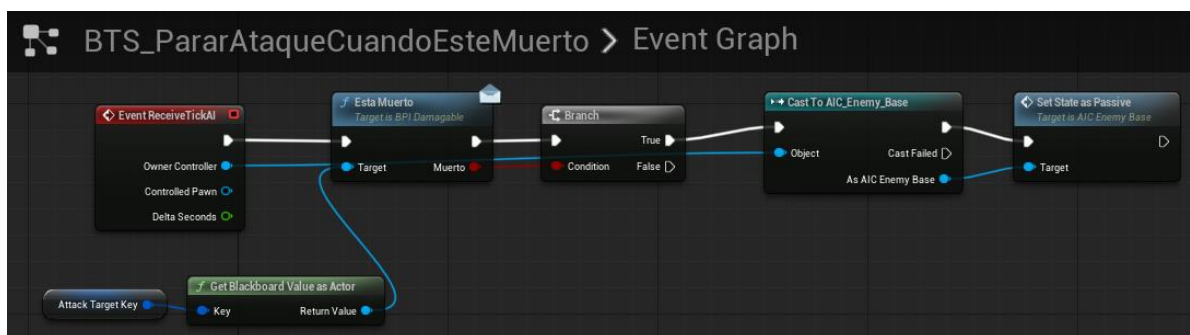
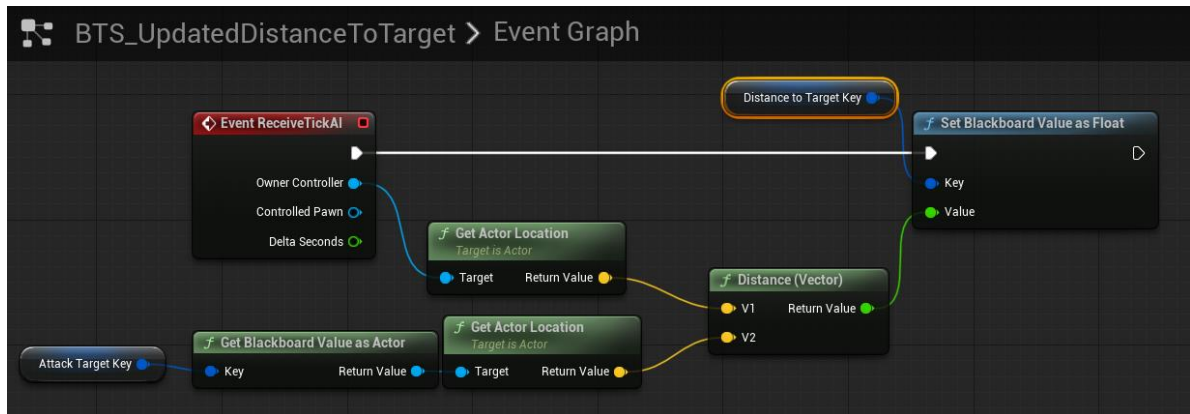




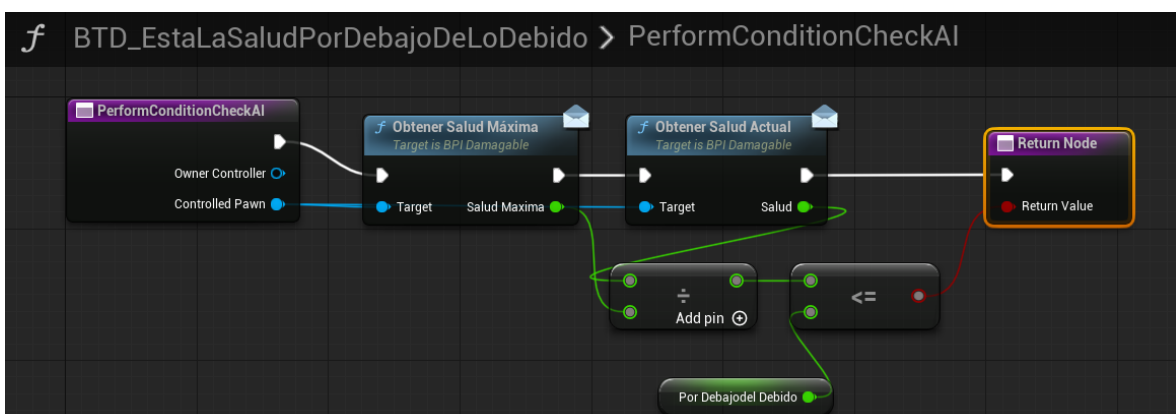
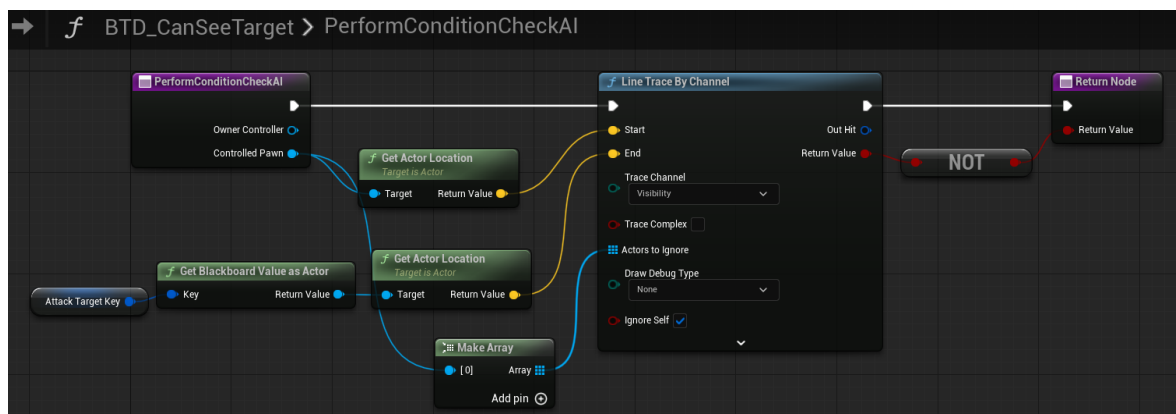


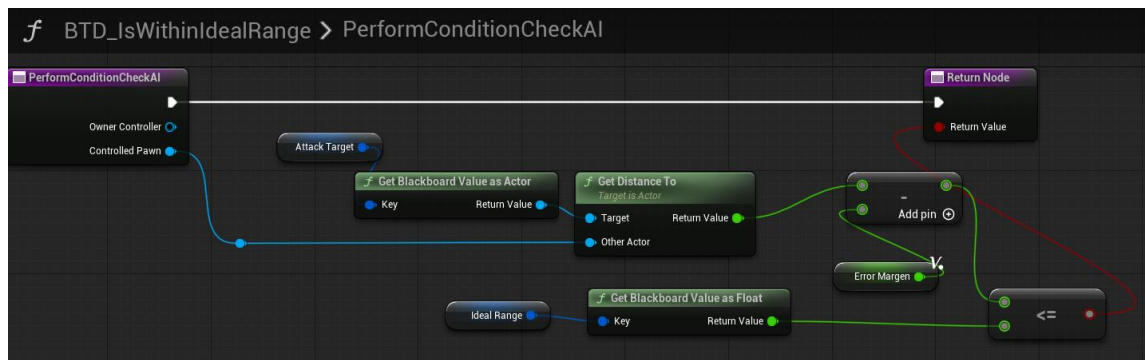
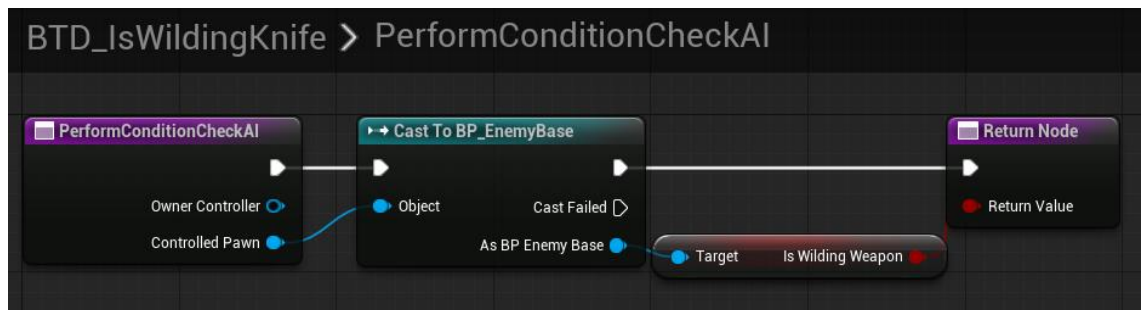
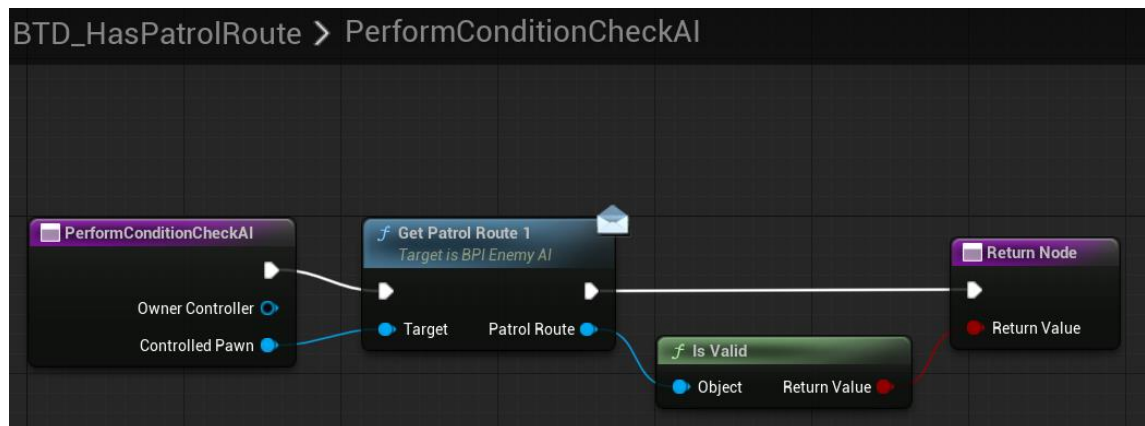


SERVICES



DECORATORS





BEHAVIOR TREES

