Desarrollo de un videojuego Soulslike en Unreal Engine

Grado en Ingeniería Multimedia

Trabajo Fin de Grado

Autor:
Antonio José Fernández Belliure

Tutor/es:
Rosana Satorre Cuerda

Diciembre 2022

Versión del documento

2022.12.14

Licencia

Se permite la reproducción, distribución y comunicación pública de la obra, incluso con fines comerciales siempre y cuando reconozca y cite la obra de la forma especificada por el autor o el licenciante.

Resumen

Se plantea el desarrollo de un videojuego de género de rol de acción con estilo ***Soulslike*** en el motor ***Unreal Engine***. Para ello, el autor propone la programación de un personaje jugable con todas las mecánicas típicas de un ARPG estilo *Soulslike* así como la programación tres personajes con **Inteligencia Artificial**: un enemigo base (*Mob Enemy*), un Jefe Final (*Final Boss*) y un personaje que ayuda al personaje principal a derrotar a los enemigos. Además, el autor debe implementar el flujo de juego y controlar toda la casuística que envuelve el tener un prototipo mínimo completo jugable. Todo ello mediante el uso de las tecnologías ***Blueprints y C++*** disponibles en el motor *Unreal Engine*. Por último, se incluye un estudio de mercado para evaluar los costes y viabilidad del proyecto, así como un Análisis de casos de uso y Especificación de requisitos.

Motivación, justificación y objetivo general

El mundo de los videojuegos siempre ha estado presente en mi vida. Desde que encendí por primera vez mi *PlayStation 2* no ha habido día donde los videojuegos no hayan estado en mis pensamientos. Con ellos me he reído, me he enfadado, me he frustrado, he disfrutado, he conocido gente nueva y he descubierto que hay mil y una formas de expresar la creatividad que uno tiene dentro.

Es cuando descubro la saga ***Souls***, en concreto con el juego *Dark Souls: Prepare to Die Edition,* donde realmente me doy cuenta de lo abrumadora pero maravillosa que puede llegar a ser la imaginación. Sus escenarios, banda sonora, personajes, jefes finales, historias y mecánicas hicieron que me enamorase de los videojuegos y despertaron en mi la idea de querer hacerlos.

En un primer momento me llamó mucho la atención el poder llegar a crear personajes y escenarios tan espectaculares como los de ese juego, pero a lo largo de la carrera ha ido ganando fuerza la idea de dotar de vida y sentido a todos los elementos de un videojuego mediante la programación.

Es por ello que con este proyecto pretendo plasmar los conocimientos obtenidos a lo largo del Grado en Ingeniería Multimedia mediante el desarrollo de un videojuego **ARPG** estilo *Dark Souls* a través del diseño y la programación de las mecánicas clásicas de este tipo de juegos, así como la Inteligencia Artificial de los enemigos base, de un jefe final y de un personaje de ayuda.

Este reto me servirá para ampliar los conocimientos y habilidades obtenidos en el ABP (Aprendizaje Basado en Proyectos) sobre programación y diseño de Inteligencia Artificial y *Gameplay*. Además, me ayudará a dominar las diferentes tecnologías que utiliza el motor ***Unreal Engine*** para la programación de videojuegos y donde pondré a prueba mis habilidades de gestión de proyectos.

Es un proyecto viable ya que se desarrollará un producto reducido, pero con un buen acabado y jugable. En él, quiero transmitir al usuario, al menos, una mínima parte de la magia y de las sensaciones que me produce a mi esta saga de videojuegos.

Agradecimientos

Siempre me ha costado mucho mostrar mi agradecimiento y en general mis sentimientos por escrito, pero espero que en esta ocasión sepa transmitir todo aquello que siento a todos a los que dedico este trabajo.

En primer lugar, quiero dar las gracias a mi equipo, **Ocacho Games**. Habéis sido mi principal motivación para aprender y mejorar cada día. No solo en lo académico, sino también en lo personal. Todo lo vivido durante la carrera no se me va a olvidar en la vida, sobre todo este último año. Estemos donde estemos siempre vamos a estar juntos. Sois personas maravillosas y estoy eternamente agradecido de haberme encontrado con vosotros. Os quiero.

En segundo lugar, mi fan incondicional, mi mayor apoyo, mi novia Alba. Gracias por decirme tantas veces lo orgullosa que estás de mí. Gracias por saber cómo levantarme el ánimo en los momentos más difíciles. Gracias por hacerme ver las cosas más claras cuando todo se torna borroso. Gracias por motivarme a seguir mejorando cada día. Gracias por crecer conmigo. Te quiero.

En tercer lugar, mis amigos de toda la vida. Me siento afortunado de tener un grupo como vosotros, un grupo con tantos años como los que tenemos nosotros. Aunque a veces tardemos en vernos siempre estáis ahí. Siempre me habéis apoyado y sé que lo seguiréis haciendo. Gracias.

En cuarto lugar, mi familia. Gracias a mis padres, me lo habéis dado todo. Siempre atentos, siempre apoyándome y siempre defendiendo a muerte mis decisiones. Gracias a vosotros he podido hacer lo que he querido y ser quien soy. Gracias a mi hermana, por traer ternura a mi vida y hacerme sentir el hermano más afortunado del mundo. Sigue siendo como eres y llegarás muy lejos. Os quiero.

Por último, a los profesores y profesoras que se han preocupado en ofrecerme una enseñanza de calidad y formarme en el ámbito profesional y, en ocasiones, en el personal. Gracias a Sabina por introducirme en este mundo. Gracias a Fran, por todo lo que me has enseñado este último año. No ha habido profesor que me haya inspirado tanto como tú. Y en especial a mi tutora, Rosana, gracias por haberme apoyado a lo largo de todo el proyecto.

Citas

La vida es un lienzo en blanco, y debes lanzar sobre él toda la pintura que puedas.

Danny Kaye

Índice de contenidos

Resumen ... 3

Motivación, justificación y objetivo general ... 4

Agradecimientos ... 5

Citas ... 6

Índice de figuras ... 13

Índice de tablas ... 16

1. Introducción ... 20

1.1. Estructura del documento ... 21

2. Estudio de viabilidad ... 23

2.1. Análisis DAFO ... 23

2.1.1. Estrategias de ataque ... 24

2.1.2. Estrategias de supervivencia ... 24

2.1.3. Estrategias defensivas ... 25

2.1.4. Estrategias de reorientación o adaptativas ... 25

2.2. Análisis de riesgos ... 26

2.2.1. Enfermedades físicas y mentales ... 26

2.2.2. Pandemia de la COVID – 19 ... 26

2.2.3. Carga de trabajo durante el curso académico ... 27

2.2.4. Reuniones y compromisos ... 27

2.2.5. Desconocimiento de las tecnologías a utilizar ... 28

2.2.6. Depuración de errores ... 28

2.3. Estudio de mercado ... 28

2.3.1. Estudio de mercado teórico ... 29

2.3.2. Estudio de mercado real ... 33

3. Planificación ... 34

3.1. Diagrama de Gantt ... 34

3.2. Cumplimiento de plazos 36

4. Situación actual 37

4.1. ¿Qué es un videojuego? 37

4.1.1. Clasificación de los videojuegos por género 38

4.2. Videojuegos RPG 44

4.2.1. JRPG 44

4.2.2. WRPG 46

4.2.3. MMORPG 47

4.2.4. ARPG 49

4.3. La saga *Souls* como inspiración 50

4.3.1. Contexto 50

4.3.2. Género *Soulslike* 52

4.4. Juegos referentes 56

4.4.1. Dark Souls y Dark Souls III 56

4.4.2. Bloodborne 58

4.4.3. Sekiro: Shadows Die Twice 59

4.5. *Elden Ring*, donde todo converge 60

5. Objetivos 62

6. Metodología 64

6.1. Metodología de desarrollo de software 64

6.2. Gestión del proyecto 64

6.3. Repositorio y control de versiones 67

7. Análisis y especificación 69

7.1. Casos de uso 69

7.2. Especificación de requisitos 70

8. Elección de herramientas 71

8.1. Motor de juego 71

8.1.1. Unreal Engine 71

8.2. Recursos 3D externos 72
8.2.1. Bazar de Unreal Engine 72
8.2.2. Mixamo 73
8.3. Entornos de desarrollo integrado (IDEs) 73
8.3.1. Visual Studio Community 2022 73
8.4. Software de sonido 74
8.4.1. Librerías de sonidos 74
8.4.2. Audacity 74
9. Implementación 75
9.1. Introducción a *Unreal Engine* 75
9.1.2. Blueprints 77
9.1.3. Animation Blueprints 79
9.2. Proceso de *Retarget* 82
9.2.1. IK Rig Asset 82
9.2.2. IK Retargeter 83
9.3. Mecánicas de movimiento 85
9.3.1. Tipos de animaciones del personaje 85
9.3.2. Creación del *Blueprint* del jugador 86
9.3.3. Creación del *Animation Blueprint* del jugador 88
9.3.4. *Blend Space* de movimiento 89
9.3.5. Movimiento libre 90
9.3.6. Movimiento radial o fijo (*Lock-on*) 91
9.3.7. Rodar y esquivar (*Roll and Dodge*) 92
9.3.8. Salto 93
9.4. Mecánicas de combate 94
9.4.1. Mecánicas de ataque 94
9.4.2. Mecánicas de defensa 97
9.4.3. Refactorización y *Combat Component* 98

9.5. Actores interactivos 99

9.5.1. Actores ‘pickeables’ 99

9.5.2. Actores equipables 99

9.6. Componente de colisión 101

9.6.1. ANS_HitCollisionDetection 101

9.7. Mecánicas de daño, vida, resistencia y maná 103

9.7.1. Estadísticas de las armas y escudos 103

9.7.2. Stats Component 103

9.7.3. Daño y muerte del personaje 105

9.7.4. Representación gráfica de las estadísticas del personaje 106

9.7.5. Curación, recuperación de maná y regeneración de estamina 108

9.8. Componente de inventario 109

9.8.1. Representación en el HUD de los objetos del inventario y armas 109

9.9. State Manager Component 110

9.10. Inteligencia Artificial 111

9.10.1. Inteligencia artificial en los videojuegos 111

9.10.2. Inteligencia artificial en Unreal Engine 112

9.11. Implementación del actor *BP_MasterAI* 118

9.11.1. Componentes y armas 118

9.11.2. Principales diferencias con la clase *BP_PlayerCharacter* 118

9.11.3. *Animation blueprints* para los enemigos 119

9.12. AIController 119

9.13. Implementación de tareas y comportamientos específicos 120

9.13.1. Steering Behaviours 120

9.13.2. Custom Tasks 121

9.13.3. Custom Services 122

9.13.4. UtilityAI 122

9.14. Enemigo simple (*Mob Enemy*) 124

9.14.1. Barra de vida del enemigo simple 125

9.14.2. *AIPerception* del enemigo base 125

9.14.3. Árbol de comportamiento y *blackboard* 126

9.15. Jefe Final 128

9.15.1. Barra de vida del jefe final 128

9.15.2. AIPerception del jefe final 129

9.15.3. Árbol de comportamiento y *blackboard* 130

9.15.4. Combate dividido en dos fases 130

9.16. Personaje invocado 131

9.16.1. *BP_PlayerAI* 132

9.16.2. *BP_ToughSword* 132

9.16.3. Árbol de comportamiento y *blackboard* 133

9.16.4. *AIPerception* del *BP_PlayerAI* 133

9.16.5. *BP_PlayerHelper* 134

9.17. Flujo de juego 134

9.17.1. Menú principal 134

9.17.2. Menú de pausa 135

9.17.3. Pantalla de controles 135

9.17.4. Reinicio del nivel cuando el jugador muere 135

9.17.5. Comportamiento cuando el Jefe Final muere 136

9.17.6. Mensaje de muerte o victoria 137

9.18. Descansar en la hoguera 137

9.18.1. Interacción con el actor *BP_Fireplace* 137

9.18.2. Mensaje al encender la hoguera 138

9.19. Atravesar la niebla 138

9.20. Sistema de recompensa: almas 139

9.20.1. *BPC_SoulsPointsComponent* 139

9.20.2. *BP_SoulsRestorer* 140

9.20.3. WB_SoulsPoints 140
9.21. Efectos visuales y sonoros 140
9.21.1. Anim Notify AN_PlayAuroraFX 141
9.21.2. Sonido ambiente del nivel y de la pelea contra el jefe final 142
10. Pruebas y validación 143
11. Resultados 145
11.1. Resultados de las pruebas de validación 145
11.2. Revisión del análisis de costes 146
12. Conclusiones y trabajo futuro 148
Referencias 149
Anexos 159
A. Especificación de requisitos detallada 159
a. Requisitos funcionales 160
b. Requisitos no funcionales 214
B. Créditos y artefactos externos 219
a. Modelados de los personajes 219
b. Modelados del entorno 219
c. Animaciones 219
d. VFX 219
e. SFX 220
f. GUI y HUD 220
C. Repositorio y descarga del proyecto 221

Índice de figuras

Figura 1: Final del Mundial de League of Legends (2015) 20

Figura 2: Videojuego Arcade: PAC-MAN 38

Figura 3: Microsoft Flight Simulator 2020 38

Figura 4: Combate en Final Fantasy I 45

Figura 5: Combate Final Fantasy VII Remake Integrade 45

Figura 6: Combate en Pokémon Espada y Escudo 45

Figura 7: Combate en Pokémon Rojo 45

Figura 8: Personalización del personaje en Ultima 47

Figura 9: Personalización del personaje en The Witcher 3: Wild Hunt 47

Figura 10: Raid en World Of Warcraft 48

Figura 11: Combate contra el Caballero Artorias Caminante del Abismo en Dark Souls 49

Figura 12: Hidetaka Miyazaki 50

Figura 13: Zona de Anor Londo en Dark Souls y Dark Souls III 51

Figura 14: Mapa completo de Dark Souls con la zona central marcada 52

Figura 15: Jugador descansando en la hoguera del Santuario de Enlace de Fuego, Dark Souls 53

Figura 16: Fotograma de gameplay cooperativo en Dark Souls III 55

Figura 17: Pelea contra Gwyn, Señor de la Ceniza 56

Figura 18: Tronos de los Antiguos Señores de la Ceniza 57

Figura 19: Imagen ingame de Bloodborne 58

Figura 20: Lobo desplazándose con el gancho 59

Figura 21: Fotograma del Trailer-Gameplay de Elden Ring en el SumerFest 2021 60

Figura 22: Tablero de tareas de Gitlab 66

Figura 23: Tablero de tareas de Toggle 66

Figura 24: Tablero de registro de tareas de Clockify 67

Figura 25: Diagrama de casos de uso 70

Figura 26: Imagotipo de Unreal Engine 72

Figura 27: Imagotipo de Mixamo 73

Figura 28: Jerarquía de clases básica de entidades de juego en Unreal Engine. 76

Figura 29: EventGraph del Blueprint del personaje principal. 80

Figura 30: AnimGraph del Blueprint del personaje principal. 81

Figura 31: Máquina de estados prinpcipal del AnimGraph 81

Figura 32: IK Rig asset del presonaje principal 82
Figura 33: Relación utomática de cadenas de huesos entre dos IK Rig assets 83
Figura 34: Previsualización en tiempo real de una animación transferida a otro modelo. 84
Figura 35: Malla base y malla objetivo en la pose base adecuada. 84
Figura 36: Blueprint BP_PlayerCharacter previsualizado en el viewport del editor de Unreal Engine.. 87
Figura 37: Asignando el AnimBP al componente de malla esquelética del BP del personaje . 88
Figura 38: Blend space de movimiento del personaje principal 89
Figura 39: Blend space BS_Greystone empleado en el estado Idle/Walk de la máquina de estados 90
Figura 40: Personaje principal orientado hacia el enemigo fijado 91
Figura 41: Animation Montage con todas las animaciones de rodar 93
Figura 42: Uso de Anim Montages seguido del cambio de socket para modificar la posición de la espada. 95
Figura 43: Animation Montage dividido en secciones y con Anim Notify States.................... 97
Figura 44: Uso del blend nodes en el Animaton Blueprint del personaje principal................ 98
Figura 45: ANS marcados en rojo para activar y desactivar la colisión del arma.................. 101
Figura 46: Cálculo de colisiones en la animación de ataque 102
Figura 47: Barras de stats dentro del HUD de juego 107
Figura 48: HUD actualizado mostrando por pantalla las estadísticas del personaje, el arma equipada, el escudo equipado y el consumible activo... 110
Figura 49: Modelo general de IA en los videojuegos 112
Figura 50: Parámetros de configuración de la vista y percepción de daño........................... 113
Figura 51: Grafo en forma de Árbol 115
Figura 52: Parte de un Árbol de decisiones en Unreal Engine. 117
Figura 53: Espada del enemigo simple y sus datos para el daño, costes y utilidad 124
Figura 54: Malla esquelética del MobEnemy obtenida del personaje Kwang del juego Paragon 124
Figura 55: Barra de vida del enemigo simple 125
Figura 56: Árbol de comportamiento del enemigo simple. 127
Figura 57: Aurora, Glacial Empress y su espada de hielo 128
Figura 58: Barra de vida del jefe final 129
Figura 59: Árbol de comportamiento del jefe final. 130
Figura 60: Aurora, Glacial Empress en la fase evolucionada... 131

Figura 61: Malla esquelética del personaje Greystone con la sking Tough del juego Paragon. 132
Figura 62: Espada de Greystone Tough, usada para el personaje invocado. 133
Figura 63: Actor BP_PlayerHelper en el mundo de juego. 134

Índice de tablas

Tabla 1: Análisis DAFO del proyecto ... 23
Tabla 2: Estudio de mercado teórico ... 32
Tabla 3: Estudio de mercado real ... 33
Tabla 4: Planificación del desarrollo del proyecto mediante Diagrama de Gantt ... 35
Tabla 5: Leyenda de símbolos del diagrama UML de Casos de uso ... 69
Tabla 6: Pros y contras de usar Blueprints o C++ ... 78
Tabla 7: Información sobre la clasificación de casos de uso ... 159
Tabla 8: Información sobre la clasificación de los requisitos ... 159
Tabla 9: Descripción CU_01 ... 160
Tabla 10: Descripción RF_B_01 ... 161
Tabla 11: Descripción CU_02 ... 161
Tabla 12: Descripción RF_B_02 ... 162
Tabla 13: Descripción CU_03 ... 162
Tabla 14: Descripción RF_B_03 ... 163
Tabla 15: Descripción CU_04 ... 163
Tabla 16: Descripción RF_A_01 ... 164
Tabla 17: Descripción CU_05 ... 165
Tabla 18: Descripción RF_A_02 ... 165
Tabla 19: Descripción CU_06 ... 166
Tabla 20: Descripción RF_A_03 ... 166
Tabla 21: Descripción CU_07 ... 167
Tabla 22: Descripción RF_A_04 ... 168
Tabla 23: Descripción CU_08 ... 168
Tabla 24: Descripción RF_A_05 ... 169
Tabla 25: Descripción CU_09 ... 170
Tabla 26: Descripción RF_B_04 ... 170
Tabla 27: Descripción CU_10 ... 171
Tabla 28: Descripción RF_B_05 ... 172
Tabla 29: Descripción RF_B_06 ... 172
Tabla 30: Descripción RF_B_07 ... 173
Tabla 31: Descripción CU_11 ... 174
Tabla 32: Descripción RF_B_08 ... 174

Tabla 33: Descripción CU_12 175
Tabla 34: Descripción RF_B_09 176
Tabla 35: Descripción CU_13 176
Tabla 36: Descripción RF_B_10 177
Tabla 37: Descripción CU_14 178
Tabla 38: Descripción RF_B_11 178
Tabla 39: Descripción RF_B_12 179
Tabla 40: Descripción CU_15 179
Tabla 41: Descripción RF_B_13 180
Tabla 42: Descripción CU_16 181
Tabla 43: Descripción RF_B_14 181
Tabla 44: Descripción CU_17 182
Tabla 45: Descripción RF_B_15 183
Tabla 46: Descripción CU_18 183
Tabla 47: Descripción RF_B_16 184
Tabla 48: Descripción CU_19 185
Tabla 49: Descripción RF_B_17 186
Tabla 50: Descripción CU_20 186
Tabla 51: Descripción RF_B_18 187
Tabla 52: Descripción CU_21 188
Tabla 53: Descripción RF_B_19 189
Tabla 54: Descripción CU_22 189
Tabla 55: Descripción RF_B_20 190
Tabla 56: Descripción CU_23 190
Tabla 57: Descripción RF_B_21 191
Tabla 58: Descripción CU_24 192
Tabla 59: Descripción RF_B_22 192
Tabla 60: Descripción CU_25 193
Tabla 61: Descripción RF_B_23 193
Tabla 62: Descripción CU_26 194
Tabla 63: Descripción RF_B_24 195
Tabla 64: Descripción CU_27 196
Tabla 65: Descripción RF_B_25 196
Tabla 66: Descripción CU_28 197
Tabla 67: Descripción RF_B_26 197

Tabla 68: Descripción CU_29 198
Tabla 69: Descripción RF_B_27 199
Tabla 70: Descripción CU_30 199
Tabla 71: Descripción RF_A_06 200
Tabla 72: Descripción CU_31 200
Tabla 73: Descripción RF_A_07 201
Tabla 74: Descripción CU_32 202
Tabla 75: Descripción RF_A_08 203
Tabla 76: Descripción CU_33 203
Tabla 77: Descripción RF_B_28 204
Tabla 78: Descripción CU_34 204
Tabla 79: Descripción RF_B_29 205
Tabla 80: Descripción CU_35 205
Tabla 81: Descripción RF_B_30 206
Tabla 82: Descripción CU_36 207
Tabla 83: Descripción RF_B_31 208
Tabla 84: Descripción CU_37 208
Tabla 85: Descripción RF_B_32 209
Tabla 86: Descripción CU_38 209
Tabla 87: Descripción RF_B_33 210
Tabla 88: Descripción CU_39 210
Tabla 89: Descripción RF_B_34 211
Tabla 90: Descripción CU_40 211
Tabla 91: Descripción RF_B_35 212
Tabla 92: Descripción CU_41 213
Tabla 93: Descripción RF_B_36 214
Tabla 94: Descripción RNF_B_01 216
Tabla 95: Descripción RNF_B_02 216
Tabla 96: Descripción RNF_A_01 216
Tabla 97: Descripción RNF_B_03 216
Tabla 98: Descripción RNF_B_04 217
Tabla 99: Descripción RNF_B_05 217
Tabla 100: Descripción RNF_B_06 218
Tabla 101: Descripción RNF_A_02 218

1. Introducción

Los videojuegos han evolucionado de una forma increíble en las últimas décadas. Este sector ha ido creciendo no solo en el ámbito técnico sino también en el ámbito social y económico, llegando a ser, hoy en día, el principal medio de entretenimiento digital y una de las industrias artísticas que más dinero genera a nivel global.

Tanto es el impacto de los videojuegos hoy en día que a partir de ellos han aparecido diferentes formas de entretenimiento digital que se nutren directamente de los videojuegos. Por una parte, han surgido los denominados ***eSports***: competiciones profesionales de videojuegos [1]. Si bien empezaron siendo eventos reducidos donde apenas asistían un centenar de personas, hoy en día se han convertido en eventos masivos como se observa en la **Figura 1**.

Figura 1: Final del Mundial de League of Legends (2015)
(Fuente: https://www.desconsolados.com/2015/11/02/cronica-de-la-final-del-mundial-de-league-of-legends/)

Por otro lado, con la aparición de plataformas como *YouTube* o *Twitch*, surge una nueva definición para el concepto de ***gameplay***: vídeo o retransmisión en directo donde se muestra una partida de un videojuego [2]. Al principio se esperaba que no tuviera mucho éxito este formato de entretenimiento ya que se suponía más aburrido el hecho de ver a alguien jugar a un videojuego en vez de hacerlo uno mismo, pero en realidad es una forma de ocio digital consumida por millones de personas.

Pero ¿a qué se debe este aumento del número de personas interesadas en los videojuegos? A medida que el hardware ha ido evolucionando así lo han hecho los videojuegos, explotando al máximo sus características.

También es cierto que, a medida que los videojuegos han ido mejorando tanto a nivel gráfico, a nivel de jugabilidad como a nivel de rendimiento, los jugadores y jugadoras se han ido haciendo

más exigentes. Cada vez se exige mayor calidad gráfica, mayor fotorrealismo, más horas de juego y más innovación a la hora de diseñar mecánicas o estilos de juego nuevos.

Este aumento de la demanda y la constante necesidad de mejora de los videojuegos ha hecho que se amplíe la comunidad de desarrolladores y desarrolladoras de videojuegos, llegando a ser uno de los sectores que más empleo genera [3].

Es por ello que, con este proyecto el autor, aparte de demostrar los conocimientos obtenidos durante el paso por el Grado en Ingeniería Multimedia, quiere dominar las tecnologías más demandadas en la industria de los videojuegos actualmente.

Para cumplir con dicho propósito desarrolla un videojuego ***ARPG*** estilo ***Dark Souls,*** una de las sagas más influyentes de los últimos años y la cual ha servido como cuna de otros títulos como *Bloodborne* o el reciente *Elden Ring*. Más adelante, en el apartado 4. Situación actual, se profundiza en explicar los orígenes de este género y de la saga *Souls*, así como el estado actual de títulos similares y cómo estos videojuegos aprovechan las tecnologías actuales.

1.1. Estructura del documento

A lo largo de este documento se presentan las diferentes partes que componen el desarrollo del proyecto. Tras una breve introducción, se procede a detallar el apartado 2. Estudio de viabilidad, donde se estudia si los objetivos del proyecto son viables y se analizan los diferentes riesgos que pueden hacer variar el resultado. También se realiza un estudio de mercado donde se estiman los costes y beneficios del producto a desarrollar.

Seguidamente, en la sección 3. Planificación, se realiza una planificación de las diferentes etapas y tareas que se necesiten llevar a cabo. Esta estimación de tareas facilita la organización de forma más estricta y permite llevar un ritmo constante de desarrollo.

A continuación, en el apartado 4. Situación actual, se estudia el pasado, presente y futuro de los géneros y tecnologías que envuelven a los videojuegos y en concreto a los relacionados con la saga *Souls*. En este apartado se analiza en profundidad el contexto del problema, los recursos disponibles y el impacto que este proyecto puede llegar a tener.

En el apartado 5. Objetivos, se definen y explican de forma detallada los objetivos que se pretenden llevar a cabo. Conseguir estos objetivos significa que se han superado las dificultades o problemas planteados inicialmente.

Una vez claros los objetivos, se procede a detallar la metodología a seguir para el desarrollo del proyecto. En esta sección, 6. Metodología, se describe cómo se va a trabajar, definiendo la o las metodologías de trabajo y las herramientas de gestión utilizadas. Además, es útil para aprender a estimar el tiempo que se necesita para cada tarea en futuros proyectos, analizando si se han subestimado algunos apartados o sobreestimado otros.

En cuanto al análisis y especificación de requisitos, estos se encuentran detallados en el apartado 7. Análisis y especificación. En este apartado se detallan los diferentes casos de uso que forman parte del videojuego y los diferentes requisitos funcionales y no funcionales que conforman la solución en sí. Asimismo, es en el anexo A. Especificación de requisitos detallada, donde se muestran todos los casos de uso y sus requisitos en tablas.

Antes de proponer un diseño para la solución del proyecto, en el punto 8. Elección de herramientas, se hace un estudio de las diferentes herramientas de desarrollo de videojuegos. En él se analizan motores de juego comerciales, algunos Entornos de Desarrollo Integrados (IDE) y diferentes fuentes para obtener tanto recursos visuales como sonoros. Todo ello con el fin de contrastar unas herramientas con otras y así decidir cuáles utilizar en el desarrollo del proyecto.

Llegados a este punto nos encontramos con el núcleo del documento, el apartado 9. Implementación. En él se especifica el diseño y la implementación para desarrollar el videojuego. Además, se muestra la estructura del proyecto y, si es necesario, se detallan aquellas partes más importantes de la solución mediante algún fragmento del código explicado.

Los últimos apartados corresponden al 10. Pruebas y validación, 11. Resultados y 12. Conclusiones y trabajo futuro. Estos puntos suponen el broche final de este documento, el cual ha sido desarrollado con el objetivo de que cualquier lector entienda lo que se está exponiendo y con qué motivación.

2. Estudio de viabilidad

Antes de arrancar con el resto del proyecto el autor procede a realizar un estudio de viabilidad del proyecto. En él se puede analizar si el proyecto en sí mismo, sus objetivos y sus pretensiones son viables. Además, es interesante conocer qué riesgos pueden afectar al desarrollo del proyecto y qué planes de contingencia se adoptan.

Por último, se hace un estudio de mercado donde se explica y estima cuál sería el coste de desarrollo del proyecto y los beneficios que se deberían obtener para que el proyecto fuese rentable.

2.1. Análisis DAFO

El análisis DAFO es una metodología de estudio de la situación de un proyecto que analiza sus **características internas** (Debilidades y Fortalezas) y su **situación externa** (Amenazas y Oportunidades) en una matriz como se observa en la Tabla 1.

Tabla 1: Análisis DAFO del proyecto

ÁNALISIS DAFO		
	DEBILIDADES	**FORTALEZAS**
ORIGEN INTERNO	• Proyecto independiente (sin respaldo de un gran estudio o *publisher*) • Proyecto individual • Poca experiencia en desarrollo de videojuegos • Dominio incompleto de las herramientas	• Experiencia adquirida en el ABP • Formación técnica y académica adecuadas • Multidisciplinar • Constancia y perseverancia • *Know how* de pipelines de desarrollo de videojuegos
	AMENAZAS	**OPORTUNIDADES**
ORIGEN EXTERNO	• Escasez de recursos desarrollo personajes • Elevado precio herramientas • Mercado competitivo • Soluciones alternativas con más recursos económicos, publicitarios y de desarrollo	• Auge de los videojuegos • Difusión gratuita en RRSS • Proyectos autodidactas altamente valorados • Alta demanda juegos de rol de acción • Nuevas herramientas más potentes

Con la información recaudada en la matriz DAFO se pueden adaptar diferentes estrategias para dar solución a las necesidades del proyecto.

2.1.1. Estrategias de ataque

Estas estrategias se obtienen relacionando las **Fortalezas** con las **Oportunidades**. La primera estrategia de ataque consiste en explotar el auge de los videojuegos con el desarrollo de este proyecto haciendo uso de la experiencia adquirida en el ABP en el itinerario de Creación y Entretenimiento Digital del grado en Ingeniería Multimedia de la Universidad de Alicante.

Otra estrategia es aprender cosas de forma autodidacta que involucren conocimientos que no se desarrollan en profundidad durante el grado para así ampliar la formación de la que se parte.

Por otro lado, si se conocen los flujos de trabajo actuales para desarrollar un videojuego, de forma general, no supone un gran esfuerzo adaptar esos flujos a las nuevas herramientas que surjan.

Por último, tener conocimientos multidisciplinares puede ayudar a desarrollar las diferentes tareas que se requieren para llevar a cabo un juego de rol de acción, aprovechando así la alta demanda de este género para incluir en el mercado el proyecto y al desarrollador.

2.1.2. Estrategias de supervivencia

En este subapartado se relacionan las **Amenazas** y las **Debilidades**. Con ello se crean estrategias para eliminar aspectos negativos que amenazan al proyecto.

En primer lugar, al ser un proyecto independiente, sin el respaldo de un estudio con fondo económico o que cuenta con un *Publisher*[1], los recursos económicos son muy escasos haciendo así que muchas herramientas útiles que son de pago queden descartadas. Para subsanar este problema existen herramientas gratuitas alternativas, en ocasiones igual de potentes, que pueden ser útiles para el proyecto.

Además, al ser un proyecto individual no es posible abarcar el cien por cien de los apartados que componen un videojuego, como por ejemplo el modelado, texturizado, *rigging*[2] y animaciones de los personajes para el nivel de calidad y detalle que se exige en el proyecto. Es por ello por lo que la escasez de recursos para integrar personajes en el videojuego puede suponer un retraso considerable o incluso un problema paralizante. Para sobrevivir a este problema se puede llegar

[1] *Publisher*: Persona o grupo de personas dedicadas a la publicación y marketing de videojuegos. En ocasiones el *Publisher* puede ofrecer una pequeña inversión económica para el desarrollo del producto.
[2] *Rigging:* Dotar de esqueleto y demás elementos a una escultura digital tridimensional para poder deformar la malla y otorgarle movimiento y expresividad.

a valorar el reducir la complejidad de los personajes para que sea un producto jugable, pero con menor nivel de detalle.

La breve experiencia en el desarrollo de videojuegos sumado a la competitividad del mercado puede ser un factor altamente negativo para el proyecto en caso de salir a la venta. Para evitar dicho problema se crea este proyecto no con el objetivo de competir con productos similares con más recursos sino con el objetivo de dar a conocer al desarrollador y demostrar sus capacidades sin tener en mente el beneficio económico.

2.1.3. Estrategias defensivas

En cuanto a las estrategias defensivas, son aquellas que pretenden mejorar la situación actual del proyecto y evitar agravamientos. Se crean analizando las **Fortalezas** y las **Amenazas**.

Para subsanar el problema de la escasez de recursos para el desarrollo de personajes se recurre a la implementación propia de aquello imprescindible para el desarrollo de personajes que no se pueda obtener mediante recursos ya existentes. Esto es posible gracias al perfil multidisciplinar del autor.

Por otro lado, gracias a la constancia y perseverancia en el desarrollo se puede realizar una planificación con el objetivo de aprovechar al máximo el uso de herramientas de pago antes de que finalice su periodo de prueba o el tiempo que ofrecen sus licencias gratuitas para estudiantes, por ejemplo.

2.1.4. Estrategias de reorientación o adaptativas

Las estrategias de reorientación o adaptativas buscan eliminar las debilidades y crear nuevas fortalezas. Para ello se analizan las **Debilidades** y las **Oportunidades**.

Gracias al alcance que se puede conseguir en redes sociales se puede llegar a contactar con personas o entidades interesadas en proyectos independientes que quieran ayudar de una forma u otra a que el proyecto se lleve a cabo.

Por otro lado, los proyectos autodidactas en la actualidad son altamente valorados, siendo esta una de las principales cartas de presentación a la hora de solicitar un trabajo en la industria de los videojuegos. De este modo, que el proyecto sea individual se puede convertir en un punto de atracción de cara al mercado laboral.

2.2. Análisis de riesgos

En este apartado se analizan riesgos que pueden afectar al proyecto, pero no desde la perspectiva de una organización con un producto a desarrollar, sino desde un punto más personal. También, se proponen soluciones a dichos riesgos para conducir al proyecto al éxito.

2.2.1. Enfermedades físicas y mentales

Una de las principales causas de retraso en el proyecto es el estado de salud del autor. Tanto las enfermedades físicas como mentales son grandes amenazas ya que dependiendo de la gravedad en la que se padezcan pueden llegar a retrasar el proyecto entre unos pocos días hasta semanas.

Para evitar lesiones físicas, sobre todo relacionadas con el cuello y la espalda debidas a la postura de trabajo, se plantea un horario para realizar deporte de forma regular y mantener el cuerpo en buen estado de forma. Por otro lado, practicar deporte ayuda a despejar la mente y reducir el estrés, que bien puede ser también causante de lesiones físicas.

Por otro lado, enfermedades como la gripe, faringitis o gastroenteritis, por ejemplo, son fáciles de padecer y que bien pueden suponer un retraso de varios días en función de la gravedad. Para evitar dichas enfermedades se debe mantener una dieta sana y equilibrada, evitar la exposición a cambios bruscos de temperatura y cumplir con las medidas higiénicas habituales.

En cuanto a evitar enfermedades mentales, se plantea un ritmo de trabajo constante y bien distribuido para evitar periodos con sobrecarga. Con ello se pretende reducir la cantidad de estrés que puede acumular el autor y no saturar la mente con un mismo tema durante muchas horas, ya que eso puede derivar en el uso ineficiente de horas de desarrollo que no lleven a ninguna solución.

2.2.2. Pandemia de la COVID – 19

Desde el 13 de marzo de 2020 el mundo se enfrenta a una pandemia a nivel global ocasionada por la COVID – 19. El principal riesgo de la pandemia consiste en el contagio y los efectos de este virus, derivando en ocasiones al ingreso en la unidad de cuidados intensivos. Para evitar el contagio de este virus se deben respetar todas las medidas sanitarias e higiénicas necesarias.

Además, esta pandemia trae consigo diferentes consecuencias negativas para la salud tanto físicas como mentales. El trabajo remoto y el confinamiento ha supuesto la adaptación de un ritmo de vida sedentario el cual puede ocasionar problemas cardíacos, musculares y óseos. Para evitar dicho ritmo de vida sedentario se plantea un horario para realizar deporte de forma regular, ya sea en el exterior de la vivienda o dentro.

También, el estar en casa más horas de las habituales ha derivado en aprovechar el tiempo de desplazamiento del lugar de trabajo a casa y viceversa para incluirlo como tiempo de trabajo, alargando las jornadas de trabajo de forma desmedida. Como consecuencia los niveles de ansiedad y estrés aumentan ya que la mente no tiene tiempo de descansar. Todo lo expuesto junto con otros factores nocivos para la salud mental puede derivar en depresión.

Para evitar padecer dichas consecuencias el autor plantea el salir a dar un paseo diariamente si es posible o, en caso de volver a unas medidas de restricción de movilidad estrictas tales como el confinamiento, salir a la terraza o balcón de su domicilio para despejar la mente y desconectar de todo aquello que le puede sugerir ideas relacionadas con el proyecto. Otra medida que ayuda a evitar un deterioro del estado mental es la interacción social, siempre respetando las medidas necesarias para no contagiarse del COVID – 19.

2.2.3. Carga de trabajo durante el curso académico

Este proyecto se desarrolla en paralelo con el cuarto curso del autor en el grado de Ingeniería Multimedia, donde la evaluación de este se realiza mediante **ABP** (Aprendizaje Basado en Proyectos), concretamente siguiendo el itinerario de Creación y Entretenimiento Digital. Es un año académico duro y con una carga de trabajo mayúscula, de modo que la organización para distribuir de forma adecuada la carga de trabajo es crucial.

El proyecto se comienza en julio de 2021 con el objetivo de seguir un ritmo de trabajo constante hasta la fecha de entrega de la convocatoria ordinaria de mayo-junio de 2022. En caso de no poder avanzar al ritmo deseado se plantea la posibilidad de solicitar la convocatoria extraordinaria de julio-septiembre. Como tercera opción se contempla solicitar la convocatoria extraordinaria C1 del curso académico 2022-2023.

Dado que durante el curso académico 2021-2022 el ritmo de desarrollo del proyecto se va a ver afectado por la carga de trabajo del ABP se pretende aprovechar los días que el autor no tenga carga lectiva durante la semana para avanzar con este trabajo. También, se propone la consecución de hitos de este proyecto en fechas donde la carga del ABP sea menor.

2.2.4. Reuniones y compromisos

Actos como tutorías, reuniones con la tutora del trabajo de fin de grado, reuniones de asociaciones de estudiantes e incluso compromisos con compañeros y profesores pueden llegar a retrasar el desarrollo del proyecto unos cuantos días si son actos que se repiten a lo largo del tiempo, en horas que serían utilizadas para el desarrollo y programados de forma imprevisible.

Para evitar estos retrasos se procura organizar dichas reuniones fuera del horario estipulado para el avance del proyecto si es posible, aprovechando horas libres o tiempo reservado para reuniones y compromisos. En caso de tener que utilizar horas de desarrollo para establecer compromisos, se redistribuirá la carga de trabajo para recuperar ese tiempo.

2.2.5. Desconocimiento de las tecnologías a utilizar

Dadas las características y objetivos del proyecto, el autor considera que no domina lo suficiente las tecnologías a emplear y que no debe afrontar el proyecto sin cubrir esas deficiencias. Entiende que ir aprendiendo a medida que se va implementando el trabajo, puede derivar en un proyecto poco eficiente, con errores y fases completas que deban ser desarrolladas de nuevo. Este motivo le lleva a invertir un tiempo previo al desarrollo en conocer las tecnologías a utilizar y dominarlas antes de aplicarlas al proyecto. En caso de tardar más de lo esperado en dominar la tecnología puede provocar un retraso en el proyecto, pero menor en comparación al método comentado en el párrafo anterior ya que, de este modo, el desarrollo será más fluido y con menos errores.

2.2.6. Depuración de errores

Al tratarse del desarrollo de un videojuego los errores son un factor intrínseco del proyecto. Puede haber errores de compilación, de enlazado, de ejecución o simplemente que no se conozca qué hace lo que se está usando. Es por ello que se deben tener instaladas las herramientas de depuración y visualización de datos adecuadas para detectar dónde están los errores cuanto antes. Además, se debe tener a mano la documentación del lenguaje de programación empleado, así como la del motor de juego que se haya seleccionado.

Otra medida para reducir los errores de programación es analizar bien el problema que se quiere resolver, estudiar qué elementos son necesarios y, con lápiz y papel, definir cuál es el flujo adecuado y óptimo para resolver el problema antes de escribir una línea de código.

2.3. Estudio de mercado

Un apartado interesante dentro de desarrollo de un videojuego es el estudio de mercado, donde se analizan cuáles son los costes de producción y cuáles deben ser teóricamente los beneficios a obtener para que el producto sea rentable.

Este apartado se divide en dos estudios de mercado diferentes ya que se puede hacer un estudio aproximado antes de iniciar el proyecto para estimar costes, pero durante el desarrollo pueden

surgir costes adicionales relacionados con las herramientas a utilizar, aumento de las horas de trabajo, etc.

2.3.1. Estudio de mercado teórico

- **Mano de obra**

El primer factor a considerar para realizar el estudio es la mano de obra. Dado que es un proyecto individual se supone que únicamente se debe pagar un sueldo durante el tiempo de desarrollo, pese a que dentro de un estudio hay diferentes roles y diferentes sueldos. Para el estudio se analiza el rol de **programador/a de videojuegos**.

Según estudios de portales de trabajo un/a junior cobra alrededor de **11,20€/hora** [4]. Con lo que al mes suma 1791,66€ con una jornada de 8 horas diarias 5 días a la semana, contando como que el mes tiene 4 semanas aproximadamente. En total cobra en un año 21.500€, aproximadamente.

Si el desarrollo de este proyecto se contabiliza con un total de **300 horas** (el equivalente teórico a los 12 créditos que corresponden al Trabajo de Fin de Grado), el coste es de: 300*11,20=**3.360€.**

- **Hardware necesario y sistema operativo**

Para llevar a cabo el desarrollo de un videojuego se necesita tener un ordenador adecuado para que el trabajo sea más ameno y eficiente, evitando largos tiempos de carga y procesado para cada cambio que sufra el proyecto.

  - **Torre**

Se podría emplear tanto una torre como un portátil para el desarrollo del proyecto si tienen componentes similares, sin embargo, las torres cuentan con un mejor sistema de ventilación, factor que con las altas temperaturas que alcanzan el procesador y la tarjeta gráfica es muy importante.

Una torre de trabajo para el desarrollo de videojuegos de características óptimas asciende a un total de **2.344,36€** con los siguientes componentes [5]:

    - **Torre**: *Cooler Master MasterCase H500 ARGB Cristal Templado USB 3.0 Gris Metalizado*
    - **Fuente de alimentación**: *Gigabyte P750GM 750W 80 Plus Gold Full Modular*

- **Procesador**: *AMD Ryzen 7 5800X 3.8 GHz Tray*
- **Ventilador**: *Corsair ICUE H100i RGB PRO XT*
- **Placa Base**: *Gigabyte X570 Aorus Elite*
- **Disco duro**
  - *HDD 2TB SATA3*
  - *Gigabyte AORUS NVMe Gen4 SSD 1TB M2 3D TLC (Velocidad de lectura: 5000 MB/s y Velocidad de escritura: 4400 MB/s)*
- **Memoria**: *32GBDDR4 3600MHz 2x16GB CL18*
- **Gráfica**: *Gigabyte GeForce RTX 3070 Ti GAMING OC 8GB GDDR6X*
- **Sistema operativo:** *Windows 10 Professional 64 bits* (**11,90€**) [6]

- **Monitor**

Debido a la gran cantidad de horas que el autor va a estar delante de la pantalla lo primordial es el cuidado de la vista. Para evitar daños en la vista, se escoge un monitor que sea poco nocivo para los ojos. Además, cuanta más resolución, más ventanas puede tener abiertas y monitorizar un mayor número de tareas a la vez.

Un monitor con tecnología de cuidado para la vista, 27 pulgadas de tamaño y resolución 2K (25560 x 1440 píxeles) tiene un precio de **269€** [7]**.**

- **Teclado y ratón**

El objetivo es que el autor se encuentre cómodo trabajando, por ello lo ideal es tener un teclado [8], de forma ideal mecánico, con reposamuñecas y un ratón ergonómico [9]. La suma de ambos componentes da un total de 29,98€ + 4,99€ = **34,97€.**

De esta forma el **total** a invertir en el hardware necesario para el desarrollo es de: **2.660,23€**.

- **Coste del motor de juego**

Actualmente dos de los motores de juego más potentes son *Unreal Engine* [10] *y Unity* [11]. La característica de interés en este apartado es que ambos ofrecen su uso de forma **gratuita** bajo ciertas condiciones.

- **Unreal Engine**

Utilizar el motor es **gratis.** Si se usa la licencia de *Creator* no se puede poner a la venta nada que se haya desarrollado bajo esa licencia de motor. Por otro lado, si se usa la licencia de *Publishing* (Como empresa distribuidora) también es **gratis** si no haces un beneficio mayor a 1.000.000$ (842.403,20€) [12]. En el momento que se supere esa cantidad de beneficios

hay que pagar un 5% de los beneficios que obtengas cada cuatrimestre de año fiscal. En caso de publicar el juego en la ***Epic Games Store*** el 5% ya va incluido en los beneficios que obtiene *Epic* en las ventas del juego [13].

- **Unity**

La licencia *Personal* es **gratis** y se puede poner a la venta lo que se desarrolle con ella. En el momento que se superen los 100.000$ (84.240,32€) de beneficios con ese producto, se debe actualizar la licencia a *Unity Plus*, con un precio de 399$ al año por puesto en la empresa (336,12€). En caso de superar los 200.000$ (168.480,64€) se debe solicitar el plan *Unity Pro*, que cuesta 1.800$/ al año por puesto en la empresa (1.516,33€) [14] [15].

- **Coste de herramientas para programar**

También son necesarios entornos y editores para el desarrollo del código y la implementación. Herramientas como *Visual Studio Community* y el propio entorno de *Unreal Engine* son **gratuitos**.

- **Coste de recursos ya existentes para integrar en el proyecto**

Es necesario incluir recursos ya existentes (***assets***) al proyecto para alcanzar el nivel de detalle y calidad exigidos para la parte de modelos 3D, texturas, animaciones, efectos visuales y sonoros. Se pueden obtener muchos *assets* de forma gratuita, pero se reservan alrededor de 200€ para invertir en caso de ser necesario.

- **Coste de publicar un juego en un portal**

A la hora de poner a la venta un producto en alguna plataforma de venta digital de videojuegos puede haber costes asociados. En este caso se analizan dos portales, *Steam y Epic Games Store.*

- **Pago inicial para publicar un producto**

Para subir un producto de software a *Steam* es necesario realizar un pago inicial de 100$ **(84,19€)**. Esta es una decisión de *Steam* para hacer que el desarrollador/a se comprometa a subir un producto de calidad y que de verdad quiera publicarlo, para así evitar gastar recursos en proyectos de calidad inferior. En el caso de *Epic Games Store* no se atribuye ningún pago inicial para subir el producto a su plataforma.

Sin embargo, el pagar o no una tasa fija inicial no significa que el producto vaya a ser publicado, ya que posteriormente las plataformas realizan un estudio del mismo para definir si es un producto de calidad y que reúne los requisitos para estar en su portal de ventas.

- **Porcentaje de beneficios que obtienen los portales**

En el caso de *Epic Games Store*, este portal obtiene un 12% del valor de la venta y el 88% restante es para el vendedor [16]. Por el contrario, *Steam* se agencia un 30% del valor de la venta y el 70% restante es para el vendedor [17] [18].

- **Precio de venta en diferentes portales**

A modo de resumen de toda la información expuesta se presenta una tabla de costes donde se han introducido los elementos que, de forma teórica antes de empezar con el proyecto, se suponen necesarios para el desarrollo del producto y su lanzamiento al mercado.

Con esta tabla se plantea una inversión total de forma teórica, desde la perspectiva de un estudio independiente que no puede utilizar licencias gratuitas de estudiante, y diferentes escenarios de venta que se podrían dar para al menos recuperar la inversión del proyecto.

Tabla 2: Estudio de mercado teórico

<table>
<tr><th colspan="4">ESTUDIO DE MERCADO TEÓRICO</th></tr>
<tr><th>HERRAMIENTA</th><th colspan="2">NOMBRE</th><th>PRECIO</th></tr>
<tr><td>Motor</td><td colspan="2">Unreal Engine</td><td>Gratis</td></tr>
<tr><td rowspan="2">Software</td><td colspan="2">Visual Studio Community 2022</td><td>Gratis</td></tr>
<tr><td colspan="2">Assets necesarios</td><td>200€</td></tr>
<tr><td>Sueldo</td><td colspan="2">Ingeniero Junior (300h de trabajo)</td><td>3.360€</td></tr>
<tr><td>Hardware</td><td colspan="2">Ordenador + Periféricos + Sistema Operativo</td><td>2.660,23€</td></tr>
<tr><td colspan="3" align="right">COSTE TOTAL:</td><td>6.620,23€</td></tr>
<tr><th colspan="4">VENTAS PARA RECUPERAR LA INVERSIÓN</th></tr>
<tr><td>Precio de venta</td><td>0,99€</td><td>1,49€</td><td>1,99€</td></tr>
<tr><td>Copias vendidas en Epic Games Store</td><td>7.600</td><td>5.000</td><td>5.000</td></tr>
<tr><td>Copias vendidas en Steam</td><td>9.500</td><td>6.400</td><td>4.800</td></tr>
</table>

Estos precios de venta representan un producto de calidad con contenido reducido, es decir, un producto que no llega a la complejidad de desarrollo de un videojuego de un estudio con mayores recursos, más horas de juego y mejor calidad. Los objetivos y el alcance del proyecto,

así como las funcionalidades que incluye se redactan en los apartados 5. Objetivos, 7. Análisis y especificación Y 9. Implementación.

2.3.2. Estudio de mercado real

Este apartado detalla cuál ha sido el coste real del desarrollo del proyecto. Si bien el punto anterior ha sido escrito antes de empezar con el trabajo, este se redacta una vez se saben las herramientas que se utilizan, las horas que se han invertido y el equipo informático utilizado para el desarrollo. Todos los detalles sobre los elementos que han hecho posible este trabajo se estudian y justifican más adelante en esta memoria.

Dado que la información necesaria para este estudio es la misma que en el apartado anterior se procede a mostrar directamente la tabla resumen de cuál es el coste real del proyecto.

Tabla 3: Estudio de mercado real

<table>
<tr><th colspan="4">ESTUDIO DE MERCADO REAL</th></tr>
<tr><th>HERRAMIENTA</th><th colspan="2">NOMBRE</th><th>PRECIO</th></tr>
<tr><td>Motor</td><td colspan="2">Unreal Engine</td><td>Gratis</td></tr>
<tr><td rowspan="4">Software</td><td colspan="2">Blender</td><td>Gratis</td></tr>
<tr><td colspan="2">Visual Studio Community 2022</td><td>Gratis</td></tr>
<tr><td colspan="2">Modelos 3D del Bazar de Unreal</td><td>Gratis</td></tr>
<tr><td colspan="2">Assets visules y sonoros</td><td>Gratis</td></tr>
<tr><td>Sueldo</td><td colspan="2">Ingeniero Junior (570h de trabajo)</td><td>6.384€</td></tr>
<tr><td>Hardware</td><td colspan="2">Ordenador + Periféricos + Sistema Operativo</td><td>2.545€</td></tr>
<tr><td colspan="3" align="right">COSTE TOTAL:</td><td>8.929€</td></tr>
<tr><th colspan="4">VENTAS PARA RECUPERAR LA INVERSIÓN</th></tr>
<tr><td>Precio de venta</td><td>0,99€</td><td>1,49€</td><td>1,99€</td></tr>
<tr><td>Copias vendidas en Epic Games Store</td><td>10.250</td><td>6.800</td><td>5.100</td></tr>
<tr><td>Copias vendidas en Steam</td><td>13.000</td><td>8.600</td><td>6.400</td></tr>
</table>

3. Planificación

Un factor decisivo en el desarrollo de un proyecto es la planificación. Hacer un primer planteamiento teórico de cómo distribuir el trabajo, estimar cuánto tiempo va a llevar cada apartado y fijar fechas límite ayuda a llevar un ritmo de desarrollo constante.

Teóricamente, este trabajo está planteado para invertir en él alrededor de unas 300h de trabajo a lo largo de un curso académico (entre 9 y 10 meses). Sin embargo, dependiendo de la envergadura del proyecto, las capacidades del autor y otros factores vistos en el apartado anterior, estas cantidades pueden variar.

Es por lo que a la hora de hacer una planificación inicial tenemos que tomarla como una aproximación de lo que se quiere cumplir en un caso ideal, pero a su vez teniendo en cuenta que la realidad provocará previsiblemente modificaciones en los tiempos previstos. Siempre se debe tener presente la posibilidad de hacer reajustes, ya sea por retrasos o porque se han alcanzado metas antes de lo previsto.

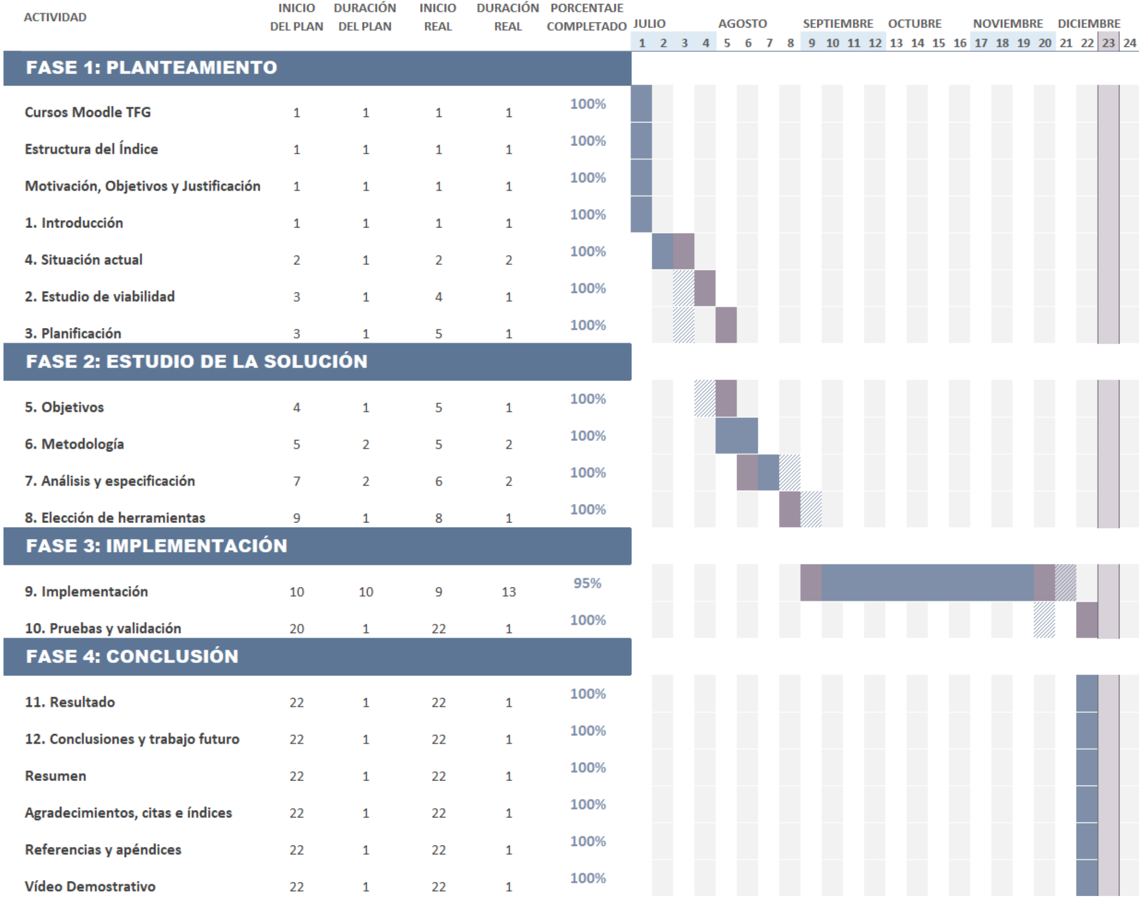
3.1. Diagrama de Gantt

Para la planificación de este proyecto se ha hecho uso de un **diagrama de *Gantt***. El diagrama de *Gantt* consiste en un diagrama sencillo y que muestra de un solo vistazo el trabajo realizado hasta la fecha, los plazos a cumplir y el trabajo restante. Además, también refleja la cantidad de trabajo que se ha realizado dentro y fuera de los plazos previstos.

Toda esta información es representada mediante una lista de actividades a las que se les asigna un periodo de inicio teórico, una duración teórica, un periodo de inicio real y una duración real. En cuanto a la organización cronológica, los periodos se corresponden con semanas, donde cada periodo está identificado con un número y cada mes cuenta con 4 periodos de trabajo. Así pues, las actividades comienzan en una semana concreta de trabajo y tienen asignada una duración teórica.

Este tipo de diagrama permite al autor hacer una planificación inicial, tomando como objetivo teórico el finalizar el proyecto para la convocatoria ordinaria de Mayo – Junio del curso 2021/2022 pero ofreciéndole la flexibilidad de reajustar la planificación si decide presentarse a la convocatoria extraordinaria de Julio – Septiembre del curso 2021/2022 o a la convocatoria C1 del curso 2022/2023. Pudiendo así mantener reflejado el progreso obtenido a la vez que se vuelven a estimar tareas que aún no se han completado.

Tabla 4: Planificación del desarrollo del proyecto mediante Diagrama de Gantt



ACTIVIDAD	INICIO DEL PLAN	DURACIÓN DEL PLAN	INICIO REAL	DURACIÓN REAL	PORCENTAJE COMPLETADO
FASE 1: PLANTEAMIENTO					
Cursos Moodle TFG	1	1	1	1	100%
Estructura del Índice	1	1	1	1	100%
Motivación, Objetivos y Justificación	1	1	1	1	100%
1. Introducción	1	1	1	1	100%
4. Situación actual	2	1	2	2	100%
2. Estudio de viabilidad	3	1	4	1	100%
3. Planificación	3	1	5	1	100%
FASE 2: ESTUDIO DE LA SOLUCIÓN					
5. Objetivos	4	1	5	1	100%
6. Metodología	5	2	5	2	100%
7. Análisis y especificación	7	2	6	2	100%
8. Elección de herramientas	9	1	8	1	100%
FASE 3: IMPLEMENTACIÓN					
9. Implementación	10	10	9	13	95%
10. Pruebas y validación	20	1	22	1	100%
FASE 4: CONCLUSIÓN					
11. Resultado	22	1	22	1	100%
12. Conclusiones y trabajo futuro	22	1	22	1	100%
Resumen	22	1	22	1	100%
Agradecimientos, citas e índices	22	1	22	1	100%
Referencias y apéndices	22	1	22	1	100%
Vídeo Demostrativo	22	1	22	1	100%

El autor inicia el desarrollo del proyecto en el verano de 2021 con la redacción de las dos primeras fases de la memoria y con la idea de presentar el trabajo de fin de grado para la convocatoria de mayo-junio de 2021. Sin embargo, debido a la exigencia del ABP el autor debe pausar el desarrollo del proyecto y retomarlo en septiembre de 2022, pasando a ser la fecha de entrega la convocatoria C1 del curso académico 2022-23. De este modo, las Fases 1 y 2 fueron desarrolladas en los meses de julio y agosto de 2021 y el resto del proyecto entre septiembre y diciembre de 2022.

3.2. Cumplimiento de plazos

En este apartado se muestra de forma directa y resumida el cumplimiento de plazos y fechas previstos inicialmente y mostrados en el apartado 3.1. Diagrama de Gantt

ACTIVIDAD	FECHA LÍMITE TEÓRICA	FECHA DE FINALIZACIÓN	RETRASO GLOBAL
FASE 1: PLANTEAMIENTO			
Cursos Moodle TFG	9 de julio de 2021	5 de julio de 2021	0 semanas
Estructura del Índice	9 de julio de 2021	9 de julio de 2021	0 semanas
Motivación, Objetivos y Justificación	9 de julio de 2021	6 de julio de 2021	0 semanas
1. Introducción	9 de julio de 2021	9 de julio de 2021	0 semanas
4. Situación actual	16 de julio de 2021	23 de julio de 2021	1 semana
2. Estudio de viabilidad	23 de julio de 2021	30 de julio de 2021	1 semana
3. Planificación	23 de julio de 2021	2 de agosto 2021	2 semanas
FASE 2: ESTUDIO DE LA SOLUCIÓN			
5. Objetivos	30 de julio de 2021	3 de agosto de 2021	1 semana
6. Metodología	13 de agosto de 2021	9 de agosto de 2021	0 semanas
7. Análisis y especificación	27 de agosto de 2021	20 de agosto de 2021	0 semanas
8. Elección de herramientas	10 de septiembre de 2021	29 de agosto de 2021	0 semanas
FASE 3: IMPLEMENTACIÓN			
9. Implementación	25 de noviembre de 2022	11 de diciembre de 2022	2 semanas
10. Pruebas y validación	2 de diciembre de 2022	14 de diciembre de 2022	2 semanas
FASE 4: CONCLUSIÓN			
11. Resultados	9 de diciembre de 2022	14 de diciembre de 2022	1 semana
12. Conclusiones y trabajo futuro	15 de diciembre de 2022	14 de diciembre de 2022	0 semanas
Resumen	15 de diciembre de 2022	12 de diciembre de 2022	0 semanas
Agradecimientos, citas e índices	15 de diciembre de 2022	12 de diciembre de 2022	0 semanas
Referencias y apéndices	15 de diciembre de 2022	13 de diciembre de 2022	0 semanas
Vídeo	15 de diciembre de 2022	15 de diciembre de 2022	0 semanas

4. Situación actual

En este apartado se describe el contexto que envuelve al proyecto. Para un correcto desarrollo es necesario conocer el campo sobre el que se va a trabajar, analizar las bases que han inspirado la temática de este trabajo, estudiar las referencias más influyentes y tener en cuenta hacia dónde se dirige esta problemática.

4.1. ¿Qué es un videojuego?

Los videojuegos están tan integrados hoy en día en la vida de las personas que cualquiera sabe lo que es un videojuego. Pese a ello, no todo el mundo tiene una definición clara de este concepto.

Depende de dónde se busque la información o a quién se le pregunte, la respuesta es diferente y en todos los casos adecuada. Según la Real Academia Española un **videojuego** es un *"Juego electrónico que se visualiza en una pantalla"* [19].

Otra de las definiciones de videojuego es que *"Un videojuego es una aplicación interactiva orientada al entretenimiento que, a través de ciertos mandos o controles, permite simular experiencias en la pantalla de un televisor, una computadora u otro dispositivo electrónico"* [20].

Esta segunda definición es un poco más extensa que la primera y un poco más atrevida, afirmando que un videojuego está orientado al **entretenimiento**, pero ¿únicamente están orientados a entretener?

Si se le pregunta a un docente o a un terapeuta posiblemente responda que un videojuego sirve también para **aprender** y su definición de videojuego sería distinta a las anteriores. Por otro lado, si esta misma pregunta se le formula a un desarrollador de software o videojuegos podría concluir que los videojuegos son un campo de **investigación** sobre los gráficos por computador, por ejemplo.

Sin embargo, también hay personas que consideran un videojuego como una **experiencia** más de su vida, ya sea por la historia que narra, por los elementos visuales que lo forman o por el vínculo que el jugador o jugadora puede crear con alguno de los personajes [21].

Teniendo en cuenta estos detalles, se podría concretar la definición de videojuego como: *"Un videojuego es un programa informático creado con el fin básico de entretener, donde una o más personas interactúan con un dispositivo electrónico que reproduce el juego. En él, los jugadores*

controlan diversos elementos gráficos para conseguir un objetivo concreto mediante experiencias que, en ocasiones, no podrían vivir en la vida real."

4.1.1. Clasificación de los videojuegos por género

Como se ha visto en la sección anterior, es muy complejo definir cuál es el objetivo concreto de un videojuego más allá del entretenimiento. Es por ello, que dentro de los videojuegos existen varias clasificaciones en función del objetivo que persiguen, su estilo artístico o la forma de interactuar con la persona que juega.

En este subapartado se clasifican los videojuegos según los principales géneros conocidos, empezando por los videojuegos arcade, **Figura 2**, hasta acabar con los simuladores fotorrealistas, como el Microsoft Flight Simulator [22], **Figura 3**.

- **Arcade:**

Figura 2: Videojuego Arcade: PAC-MAN
(Fuente: https://www.lainformacion.com/tecnologia/los-diez-mejores-juegos-arcade/6273814/)

Figura 3: Microsoft Flight Simulator 2020
(Fuente: https://www.muycomputer.com/2020/08/24/microsoft-flight-simulator-crysis/)

Un videojuego *arcade* es aquel que por su estética o sencillez de uso recuerda a los videojuegos de las máquinas recreativas. Un juego es arcade cuando es muy sencillo de jugar y/o cuando sus físicas o controles no responden de manera realista a las leyes de la física, con el fin de facilitar su manejo [23].

Algunos ejemplos de videojuegos arcade son: *PAC-MAN (1980)*, *Donkey Kong (1981)* o *Arkanoid (1986)*.

- **Plataformas**

Un videojuego de plataformas se caracteriza por controlar un personaje en un mundo 2D o 3D donde el jugador/a utiliza sus habilidades para realizar desplazamientos laterales a izquierda o derecha, correr, escalar o saltar (a veces de forma acrobática) una serie de plataformas u obstáculos evitando a los enemigos y recolectando una serie de objetos para avanzar en el juego. En algunas ocasiones, estos movimientos se ven potenciados por algunos elementos del propio entorno u objetos especiales que ha recogido el jugador/a [24] [25].

Dentro de este género se encuentran títulos tan conocidos como *Super Mario Bros., Donkey Kong* o *Spyro*.

- **Acción**

Los videojuegos de acción son aquellos donde el usuario/a debe hacer uso de su velocidad, destreza y tiempo de reacción para avanzar. Dado que es un género muy amplio, dentro de este se disponen varios subgéneros como son los videojuegos de lucha o de disparos, entre otros [26].

  - **Lucha**

  Este subgénero se caracteriza por que la persona que juega se enfrenta en combates **uno contra uno** contra otro personaje controlado por la máquina u otro ser humano. Dentro de este género, cada personaje cuenta con una gran cantidad de movimientos diferentes, que se encadenan unos con otros mediante combos [27].

  Otra característica es que los escenarios cuentan con elementos interactivos que los jugadores/as pueden utilizar para hacer ataques especiales, esquivar o activar transiciones entre mapas.

  Existen infinidad de videojuegos de este subgénero, por ejemplo, *Mortal Kombat, Street Fighter* o *Tekken*.

  - **Disparos** (*Shooter*)

  El principal objetivo de estos videojuegos es disparar y matar enemigos, generalmente con armas de fuego. Dentro de este subgénero encontramos otras clasificaciones más específicas, diferenciadas principalmente por el tipo de cámara o perspectiva utilizada [28].

    - FPS (*First Person Shooter*)

    La perspectiva seleccionada es en primera persona, simulando de forma realista cómo sería disparar un arma por uno mismo, pero a través de los ojos del personaje.

    - TPS (Third Person Shooter)

    En este caso el jugador/a observa al personaje que controla desde una perspectiva en tercera persona desde la parte posterior de este, normalmente a la altura del brazo para facilitar el apuntado.

Algunos ejemplos son las sagas *Call of Duty, Battlefield* o *Counter Strike.*

- **Aventura**

Estos videojuegos están diseñados de forma que, para que el/la protagonista avance en la trama, se debe interactuar con otros personajes del juego, interactuar con objetos, tomar decisiones, investigar el escenario o resolver puzles [29].

Otra característica a destacar de este tipo de juegos es que al final de cada fase o nivel hay un jefe de nivel, el cual debe ser derrotado para poder avanzar en la historia y que motivará al jugador/a a seguir hacia el final.

Dentro de este género se encuentran juegos como *BioShock Infinte, Metal Gear Solid, Tomb Rider* o *Uncharted.*

- **Estrategia**

Para clasificar un videojuego dentro de este género es necesario que exista una manipulación de un numeroso grupo de personajes, objetos o datos para lograr los diferentes objetivos propuestos.

En función de la temática que representen pueden ser juegos de gestión económica, social o de tipo bélico. Esta última rama es la tendencia actual dentro del género. A su vez, dependiendo de la mecánica podemos encontrar juegos de estrategia por turnos o en tiempo real.

Como referencia se pueden nombrar títulos como *StarCraft, Civilization* o *Age of Empires*.

- **Survival Horror**

Este género se encuentra inspirado por las películas de terror. Se caracterizan por su ambientación de terror y porque la persona que juega tiene en todo momento la sensación de que es inferior al enemigo [30].

Normalmente, el personaje principal carece de los medios necesarios para acabar con los enemigos, de modo que este se ve obligado a evadir o escapar de los enemigos. A parte, el jugador/a debe resolver una gran cantidad de puzles y acertijos para poder avanzar en el videojuego.

Silent Hill, Resident Evil, The Evil Within o *Outlast* son fieles representantes de este género.

- **Rol**

En los videojuegos de rol el/la protagonista interpreta un papel y ha de mejorar sus habilidades o estadísticas, tales como su nivel de vida, resistencia, aguante, fuerza, destreza, inteligencia, etc., para poder definir su identidad y su estilo de lucha [31].

Por otro lado, las peleas son muy comunes en los videojuegos de rol. Dichas peleas se pueden llevar a cabo en tiempo real o por turnos y pueden ser combates locales pequeños uno contra uno o combates masivos multijugador online.

Además, el/la protagonista debe interactuar con el entorno y con otros personajes para indagar sobre la historia, comprar o vender objetos y tomar decisiones. Este último acto, normalmente, suele guiar al protagonista a decantar su destino, dando lugar a diferentes hechos y finales en función de las decisiones tomadas a lo largo del juego.

En el apartado 4.2. Videojuegos RPG se comentan algunos de los subgéneros de los juegos de rol más relevantes para este proyecto.

Algunos juegos de rol son *Darkest Dungeon, Final Fantasy* o *Dark Souls* entre muchos otros.

- **Carreras**

Los videojuegos de carreras son aquellos en los que se pilotan diferentes vehículos, reales o ficticios, con el fin de ganar carreras. Originalmente, los juegos de carreras se centraban en carreras de coches donde el objetivo era llegar primero/a a la meta, pero este género ha ido evolucionando y añadiendo distintas alternativas de juego. Hoy en día es posible conducir o pilotar casi cualquier tipo de vehículo (coches, motos, barcos, aviones o naves espaciales) y no solo con el fin de llegar primero/a a la meta, sino conseguir más puntos haciendo derrapes, huyendo de la policía o batiendo récords.

Una gran influencia para este género ha sido Internet, dando lugar a partidas online donde jugadores y jugadoras de todo el mundo pueden medir sus habilidades y poner a prueba las diferentes mejoras, tanto estéticas como mecánicas, de sus vehículos.

La principal referencia de este género es la saga *Need For Speed*, aunque en los últimos años los títulos de *Forza* han ganado terreno gracias a su fotorrealismo extremo. Pese a ello, el favorito para jugar entre amigos o en familia a día de hoy es *Mario Kart 8 Deluxe*.

- **Deportivos**

Los videojuegos deportivos son aquellos que recrean eventos de las diferentes disciplinas deportivas (fútbol, baloncesto, tenis, golf, fútbol americano, béisbol, skate, etc.) tanto individuales como colectivas y donde el objetivo es siempre ganar. Normalmente el jugador/a controla a todo el equipo cambiando constantemente entre los diferentes jugadores/as que lo componen, siendo el modo de juego más directo y demandado, pero existen otras variantes donde se controla al entrenador/a y este debe saber gestionar los recursos de su equipo para ganar.

Los títulos de este género suelen tener revisiones anules y nuevos lanzamientos con cada temporada, actualizando plantillas y en ocasiones añadiendo nuevas mejoras en cuanto a gráficos y jugabilidad.

El año de salida o de la temporada suele formar parte del título del videojuego, así se encuentran como tendencias actuales el *FIFA 22* para el fútbol, *NBA2K 22* para baloncesto y *Mario Golf: Super Rush*.

- **Musicales**

Los videojuegos musicales se caracterizan por centrarse en el uso de la música para el desarrollo de la jugabilidad [32].Dentro de este género se encuentran varios subgéneros:

  - **Baile**

  Subgénero caracterizado por la representación más o menos fiel de una coreografía ejecutada por el jugador/a y captada por un sistema de control de movimiento [33] como el sistema Kinect de Xbox [34].

  - **Karaoke**

  Los videojuegos de karaoke son aquellos que, mediante el uso de un micrófono conectado a la plataforma donde se está ejecutando el juego, permiten al jugador/a conseguir puntos a medida que canta con mayor o menor fidelidad la canción seleccionada.

  - **Ritmo**

  Dentro de este subgénero la jugabilidad va ligada al ritmo de la canción, teniendo que pulsar un botón o realizar un movimiento en el momento exacto de acuerdo con el ritmo [35].

Como ejemplos de videojuegos musicales se puede hacer referencia a títulos como los *Guitar Hero*, *SingStar* o *BeatSaber*, siendo este último una gran tendencia de los juegos musicales gracias a su implementación con realidad virtual.

- **Educativos**

En la actualidad, los videojuegos son considerados también como una buena herramienta educativa. Si se encuentra el equilibrio entre el entretenimiento y el aprendizaje, el resultado puede ser muy rico para el usuario que está jugando a un videojuego de este tipo.

Es por ello por lo que dentro de este género encontramos videojuegos orientados al aprendizaje de idiomas (*Influent*), recetas de cocina (*Cook, Serve, Delicious!)* o juegos para mantener o mejorar la agilidad mental (*Brain Training, del Dr. Kawashima*).

- **Simuladores**

Aquellos juegos que intentan representar de la forma más fiel la realidad, evitando la simplicidad de controles e imitando de la forma más correcta las leyes de la física. Asimismo, otro factor que caracteriza a algunos de estos videojuegos es el grado de detalle y fotorrealismo que alcanzan [36].

Dentro de este género se encuentran simuladores de vuelo, de conducción de camiones o incluso de gestión de parques temáticos o zoos. Algunos ejemplos son el *Microsoft Flight Simulator* o *Euro Truck Simulator* [37].

Por otro lado, también hay videojuegos de simulación sociales donde, el detalle de las interacciones sociales entre los personajes y las propias carencias del cuerpo humano están muy logradas, permitiendo al jugador/a experimentar cualquier acción que podría hacer en la vida real dentro del videojuego. El ejemplo más claro de este tipo de videojuegos es *Los Sims* [38].

4.2. Videojuegos RPG

Una vez definido el concepto de videojuego y hecha una primera clasificación de estos en los principales géneros, se describe en detalle el tipo de videojuegos de rol, conocidos como juegos **RPG** (*Role Playing Game*).

La razón de este análisis más detallado en este género en particular se debe a que es el género al que pertenecen los videojuegos más influyentes en este proyecto, de modo que para alcanzar un producto que se pueda clasificar dentro de este género es necesario prestarle un tiempo y una atención especial en estudiarlo.

Así pues, también es interesante estudiar algunos de los subgéneros de este, ya que el género de juegos de rol es muy amplio y dentro del propio género existen videojuegos que son muy diferentes a otros en función del subgénero al que pertenecen.

En primer lugar, se estudian los videojuegos **JRPG** y **WRPG**, hechos en la región de Japón o zonas limítrofes y los creados en zonas de Europa o América, respectivamente. Posteriormente, se analiza el subgénero **MMORPG**, donde se indaga sobre la influencia de Internet en los videojuegos de rol. Por último, se detallan las características del subgénero **ARPG**, donde se clasifica la saga *Souls*.

4.2.1. JRPG

Los videojuegos JRPG (*Japanese Role Playing Game*) se crearon en **Japón** y en los países limítrofes. Estos juegos cuentan con un estilo artístico peculiar, definido por personajes muy jóvenes y andróginos, donde la historia es lineal y el combate se realiza por turnos donde el jugador/a tiene un tiempo, normalmente ilimitado, para decidir qué movimiento hacer [39].

Dentro de la partida, el jugador/a tiene **numerosos enfrentamientos** contra enemigos mientras avanza por el mapa, siendo a veces necesario que se quede en determinadas zonas peleando y ganando experiencia para poder derrotar a enemigos más poderosos.

Estos juegos no buscan acercarse a la realidad, sino que intentan recrear ambientes inspirados en la mitología japonesa o china con estética *Manga*[3] o *Anime*[4], dando lugar a videojuegos muy extravagantes, coloridos y muy llamativos.

---

[3] *Manga*: son el equivalente japonés de los cómics norteamericanos, y al igual que ellos, poseen una gran popularidad que ha traspasado fronteras, haciendo que la mayoría sean adaptados en series animadas de caricatura manga conocidas como *anime*.

[4] *Anime*: Adaptación animada de los cómics japoneses, denominados Manga.

Los abanderados de este subgénero y que lo dieron a conocer más allá de las costas niponas son los títulos de ***Final Fantasy y Pokémon***.

Por un lado, los *Final Fantasy* dejaron de lado la estética *pixel art 2D* (**Figura 4**) y el combate por turnos para dar el salto a las consolas de *PlayStation*, creciendo hasta llegar a títulos como el *Final Fantasy VII: Remake Integrade* (**Figura 5**) para *PlayStation 5*, donde la calidad de los entornos y personajes 3D remarcan las características artísticas que definen a este subgénero.

Figura 4: Combate en Final Fantasy I
(Fuente: https://retrogameman.com/2016/09/01/nes-gba-review-final-fantasy-dawn-of-souls/)

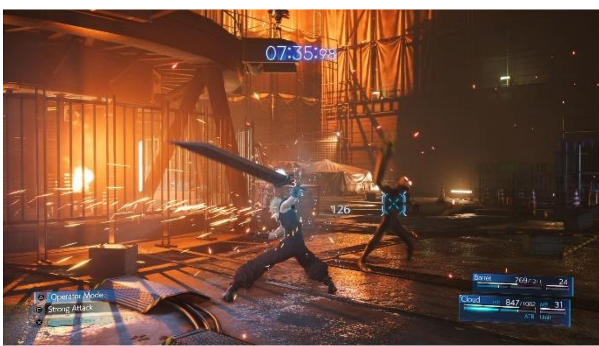


Figura 5: Combate Final Fantasy VII Remake Integrade
(Fuente: https://store.eu.square-enix-games.com/es_FR/product/630687/final-fantasy-vii-remake-intergrade-ps5)

Por otro lado, la franquicia Pokémon experimenta su apogeo con los primeros juegos, *Pokémon Rojo* y *Azul* (**Figura 7**), los cuales, hoy en día, siguen siendo los más venidos. A partir de ahí, simplemente han mantenido su estilo aprovechando la tecnología del momento y dando como resultado juegos como *Pokémon Espada* y *Escudo* (**Figura 6**).

Figura 7: Combate en Pokémon Rojo
(Fuente: https://locosxlosjuegos.com/las-10-batallas-pokemon-que-nunca-olvidaras/)

Figura 6: Combate en Pokémon Espada y Escudo
(Fuente: https://www.enter.co/evaluaciones/evaluacion-pokemon-espada-y-escudo/)

4.2.2. WRPG

El subgénero WRPG (*Western Role Playing Game*) hace referencia a los juegos de rol desarrollados en **Europa y América**. Antes del auge de los juegos JRPG, cualquier juego de rol que se desarrollaba era considerado un WRPG, pero con el lanzamiento de *Dragon Quest* desde el país nipón es necesario hacer la distinción entre juegos de rol desarrollados en Japón y juegos de rol desarrollados en Europa y América [40].

Como elementos diferenciadores de los WRPG se pueden enumerar varios, tales como la apariencia visual de los personajes, el estilo de combate y las diferentes formas en las que se puede desarrollar el juego.

El diseño de personajes intenta ser más realista y similar al mundo real en contraposición a los rasgos *Anime*, con muchos más detalles faciales y más opciones de personalización. En cuanto a las habilidades iniciales, son configurables, dando más flexibilidad al jugador/a para elegir el rol que quiere cumplir desde el inicio.

Por otro lado, el estilo de combate en los inicios también es por turnos, pero la tendencia actual está en los **combates en tiempo real con pausa**. En ellos, el jugador/a lucha contra los enemigos en tiempo real pero el combate se puede pausar para hacer un cambio de arma, consumir un objeto o cambiar el set de habilidades. Esto ayuda al jugador/a a adaptar su forma de combate en función de cómo se está desarrollando la pelea.

Por último, los WRPG barajan varios enfoques sobre cómo avanzar en el juego. Uno de ellos es el *Dungeon Crowler,* donde el jugador/a debe **explorar cuevas y mazmorras**, **matar enemigos** y **saquear cofres**. Otro enfoque es el *Sandbox RPG*, subgénero de los WRPG donde, adicionalmente a **seguir** la **historia principal**, se pueden hacer **multitud** de **misiones secundarias** en el **orden** que se **desee**, con historias muy ricas y muchas más formas de interacción que solo el combate. La tendencia más reciente para desarrollar un WRPG es la Historia Narrativa, centrándose en crear tramas mucho más complejas y con personajes muy interesantes.

Como títulos representativos de este subgénero se encuentran la saga *Ultima* (**Figura 8**)*, Diablo, Fallout* o como predominante de la historia narrativa, la saga *The Witcher* (**Figura 9**).

Figura 8: Personalización del personaje en Ultima
(Fuente: https://www.ign.com/lists/top-100-rpgs/24)

Figura 9: Personalización del personaje en The Witcher 3: Wild Hunt
(Fuente: https://forums.cdprojektred.com/index.php?threads/character-builds-ability-points-how-to-kick-ass-thread.39089/page-28)

4.2.3. MMORPG

El subgénero MMORPG (_Massive Multiplayer Online Role Playing Game_) engloba aquellos juegos de rol que únicamente se pueden jugar online, donde **multitud de jugadores y jugadoras** (de centenares a miles) se **conectan al** mismo **mundo virtual** para luchar entre ellos o cooperar para conseguir objetivos mayores. Normalmente, se producen eventos comunes a todos los jugadores/as, pero también, cada persona tiene su partida guardada en su perfil y rara vez las acciones de los demás jugadores/as en el mundo influyen en su partida [41].

El referente clásico de este subgénero es el archiconocido _World Of Warcraft,_ donde el jugador/a cuenta con infinidad de opciones de personalización, desde selección de bando (_Horda_ o _Alianza_), raza, género y clase hasta el tipo de montura que desea tener, armas y armadura. Todo ello con el fin de subir el nivel del personaje completando los eventos repartidos por todo su mundo [42].

Este videojuego sirve para conocer palabras como *Lore o Raid* por su importancia en este subgénero. En los juegos de rol comentados anteriormente la historia se consolida a través de las diferentes entregas que se lanzan al mercado, pero en juegos de rol donde el objetivo principal es hacer misiones y mejorar a tu personaje, la historia durante el juego queda relegada a un papel secundario, dejando ese peso narrativo en el *Lore*.

De modo que el término *Lore* hace referencia al conjunto de historias, datos, personajes y representaciones que conforman el universo representado en el videojuego y le dan coherencia [43].

En cuanto al término *Raid*, o Incursión, hace referencia a misiones donde un gran número de jugadores y jugadoras se enfrenta a una misión o un jefe enemigo de gran dificultad [44]. Un ejemplo de este término lo encontramos en la **Figura 10**.

Figura 10: Raid en World Of Warcraft
(Fuente: https://imgur.com/bjDsBsx)

En este tipo de misiones, es muy importante **que cada jugador/a cumpla con el rol** que ha decidido **desarrollar** a lo largo de la partida, pudiendo diferenciar así jugadores/as encargados: de curar, *Sanador;* de recibir daño, *Tanque*; aplicar daño, *Berserker* (Guerrero) entre otros.

4.2.4. ARPG

El subgénero ARPG (*Action Role Playing Game*) sigue la progresión típica de los juegos de rol, pero se caracteriza porque sus **combates se realizan en tiempo real**, donde además, tienen una **gran relevancia las estadísticas del personaje** y las **habilidades de la propia persona**[45]. Estos combates en ocasiones son parecidos a los ofrecidos en juegos del género *Hack and Slash* (“cortar y rajar”).

Otra característica de este subgénero es que suelen ser juegos **de uno contra muchos**, es decir, el personaje principal solo contra todos los enemigos, mientras que en los juegos de rol clásicos hay un equipo que pelea en conjunto.

En función de cómo el jugador/a distribuya sus niveles de experiencia entre las diferentes habilidades que puede desarrollar, los combates tendrán una tónica u otra. Si incrementa mucho su nivel de fuerza y destreza los combates se desarrollarán cuerpo a cuerpo, mientras que, si el jugador/a decide potenciar la inteligencia y la fe, por ejemplo, los combates serán a distancia y el daño se infringirá mediante hechizos, piromancias y milagros.

En este subgénero se encuentran videojuegos nombrados anteriormente como *The Witcher* o *Diablo, The Elder Scrolls V: Skyrim* y la saga *Souls* (**Figura 11**), la cual se estudia en detalle en el apartado 4.3. La saga *Souls* como inspiración.

Figura 11: Combate contra el Caballero Artorias Caminante del Abismo en Dark Souls
(Fuente: https://www.fandom.com/video/Mcrms7nA/dark-souls-lore-knight-artorias-the-abysswalker)

4.3. La saga *Souls* como inspiración

En este apartado se analiza la saga de videojuegos *Souls*, desde sus orígenes hasta la actualidad. También, se realiza un estudio del subgénero *Soulslike*, concebido a partir de esta saga y que ha servido de inspiración a títulos posteriores.

Este análisis es necesario porque dichos videojuegos son la principal influencia de este trabajo, de modo que es importante comprender cómo funcionan sus mecánicas, qué elementos los componen, cómo se avanza en el juego y qué experiencias pueden sentir los usuarios/as al jugarlos para poder plasmar todo ello en el proyecto.

4.3.1. Contexto

La saga *Souls* tiene sus orígenes en el año 2009 con la salida del videojuego ***Demon's Souls*** para *PlayStation 3* [46], título creado y desarrollado por la empresa ***FromSoftware*** [47]. El creador de la saga es **Hidetaka Miyazaki** [48], quien ha estado presente en el desarrollo del resto de títulos que componen la serie.

El sucesor de este primer título es *Dark Souls*, lanzado en 2011 y al que le siguieron *Dark Souls II* en 2014 y Dark Souls III en 2016. Todos ellos mantienen una conexión directa entre sí excepto *Dark Souls II*. Esta entrega no guarda una relación tan clara con el resto de la saga, pese a mantener el título y demás rasgos característicos. Además, Hidetaka Miyazaki (**Figura 12**) no fue el director del proyecto, ejerciendo únicamente como supervisor.

Figura 12: Hidetaka Miyazaki
(Fuente: http://nosologamer.com/miyazaki-da-por-concluida-la-saga-dark-souls-de-momento/)

Dentro de esta saga se clasifican también las entregas con contenido descargable adicional: *Dark Souls: Prepare to Die Edition*, *Dark Souls II: Scholar of the First Sin* y *Dark Souls III: The Fire Fades Edition*; aparte del *remaster*[5] de *Dark Souls* (2018) y el *remake*[6] de *Demon's Souls* (2020) para *PlayStation 5*.

Algunas de las influencias de esta serie son los juegos de la saga *King's Field*, los primeros *Dragon Quest*, y los manga *Berserk* o *Saint Seya* [49] [50]. Por otro lado, también está presente el valor estético de la cultura clásica japonesa a través del concepto ***mono no aware*** que significa: "sentir cierta melancolía o tristeza ante lo efímero" [51],.

La saga se caracteriza por controlar a un **personaje en tercera persona** y centrar el combate en la **pelea en tiempo real con armas y magia**. Dichas peleas suelen ser contra jefes finales, representados por seres fantásticos, normalmente, con anatomía humana, aunque en ocasiones el jugador/a debe enfrentarse a criaturas como dragones, lobos o demonios.

Suelen desarrollarse en un **ambiente de fantasía medieval** (**Figura 13**) con diferentes zonas interconectadas y donde la trama trata de mostrar un antiguo reino glorioso, pero actualmente en decadencia. El objetivo que persigue el personaje es diferente en cada entrega, pero suele estar encaminado a prolongar esa época de prosperidad a cambio de un sacrificio.

Otro factor importante es la exploración e **interacción con el entorno**, con los objetos y con los diferentes personajes no jugables. Muchas de estas interacciones ayudan a la persona que juega a comprender la historia que se narra en el juego y describir el *lore* que precede al juego.

Figura 13: Zona de Anor Londo en Dark Souls y Dark Souls III
(Fuente: https://www.reddit.com/r/darksouls3/comments/4h5lgl/anor_londo_then_and_now/)

[5]*Remaster*: Mejorar el sonido o la imagen de una obra sonora o audiovisual mediante procedimientos digitales.
[6] *Remake*: Rehacer el videojuego de nuevo conservando los elementos centrales del juego original y añadiendo nuevos. Normalmente con el fin de ser más atractivos y adaptarse a las nuevas tecnologías.

4.3.2. Género *Soulslike*

A continuación, se detallan en profundidad las características de los *Souls* y cómo gracias a estos juegos surge el género *Soulslike*.

Al analizar las diferentes entregas que componen la serie, se encuentran elementos centrales que son comunes a todos los *Souls*:

- **Dificultad**

Una de las principales características de estos juegos es la dificultad. Son videojuegos que normalmente hacen que el jugador/a se enfade y se frustre en numerosas ocasiones. Pese a ello, se ha convertido en uno de los principales atractivos ya que los jugadores y jugadoras sienten una satisfacción desmesurada al completar los retos.

Cabe destacar que la dificultad queda justificada ya que forma parte del diseño de estos juegos, así como la curva de aprendizaje que, pese a ser grande, conforme el jugador/a avanza, consigue experiencia y se acostumbra a la forma de combate de los enemigos, siente que domina el juego.

- **Zona central**

Otra de las características de los *Soulslike* es la existencia de una zona central del mapa. Dentro de lo complejos que suelen ser los mundos de esta serie, siempre hay un **nexo** (llamado así en *Demon's Souls*) que sirve como enlace con el resto de las zonas del mapa.

Esta zona es un **lugar seguro** para el personaje principal, sin enemigos, con una zona de recuperación y donde se encuentran personajes no jugables con los que puede interactuar, ya sea para charlar con ellos, hacer pactos o comprar y vender objetos.

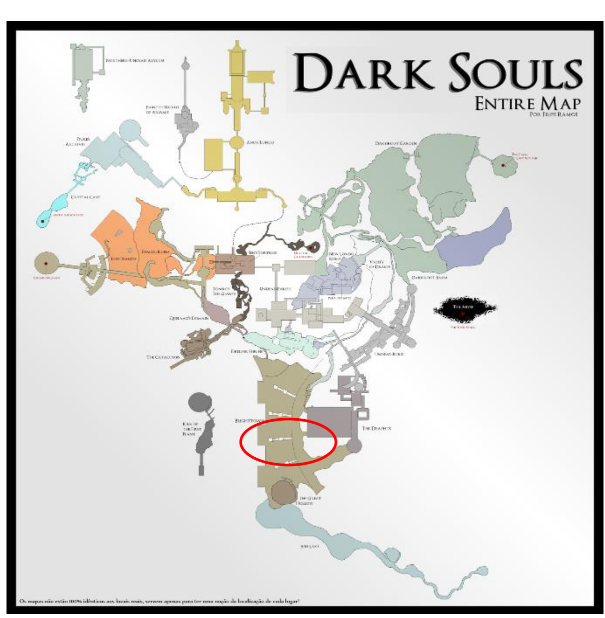


Figura 14: Mapa completo de Dark Souls con la zona central marcada
(Fuente: https://darksouls.wiki.fextralife.com/file/Dark-Souls/dark%20souls%20entire%20map%20bosses.png?v=1518927256403)

Por ejemplo, la **Figura 14** corresponde al mapa completo de *Dark Souls*, y en ella se ha marcado con una elipse roja la zona central, llamada "Santuario de enlace de Fuego". Desde aquí el

jugador/a puede seguir el camino que quiera, pero en función de su nivel debe averiguar cuál es el correcto para cada momento.

- **Puntos de control que reinician el mundo**

Este elemento está presente a partir de *Dark Souls* y sirve para que el personaje principal restaure su vitalidad (nivel de vida), guarde ese punto como lugar de reaparición y pueda realizar diferentes tareas (**Figura 15**).

Dependiendo del diseño del juego el jugador/a podrá realizar unas funciones u otras, pero algunas comunes a todas las entregas *Dark Souls* son: usar el viaje rápido, aprender/equipar nuevos hechizos y aumentar el uso de los frascos de *Estus* (pociones de vida) [52].

En los *Dark Souls* estos puntos de control se denominan **Hogueras** (*Bonfire*). Están formadas por cenizas, huesos y una espada en espiral [53].

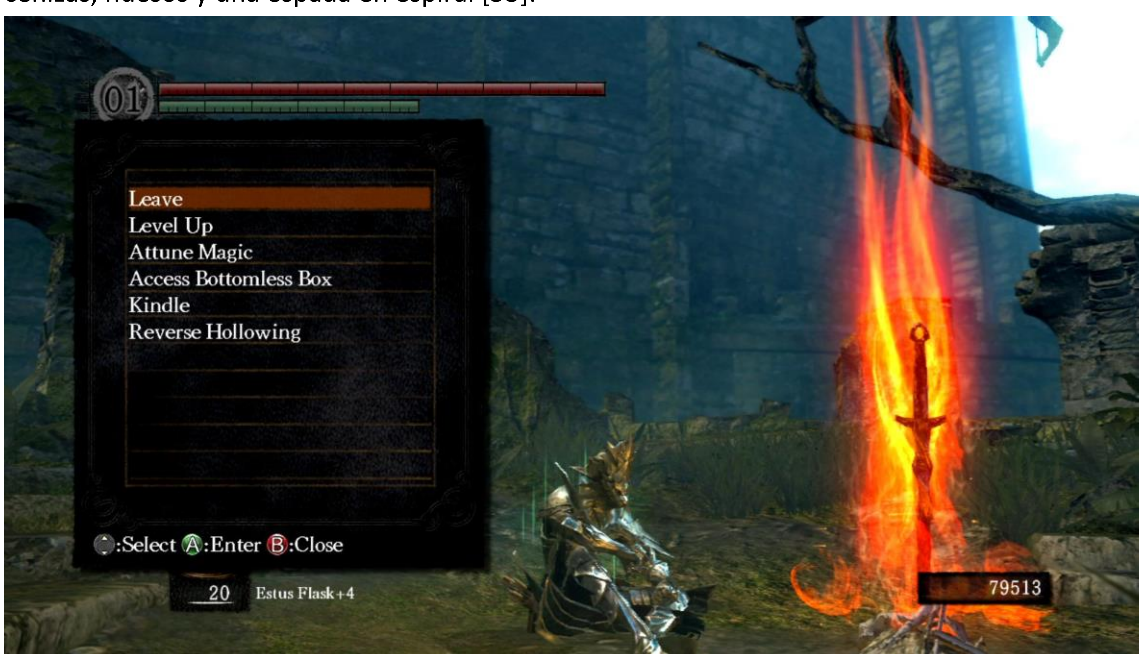


Figura 15: Jugador descansando en la hoguera del Santuario de Enlace de Fuego, Dark Souls
(Fuente: https://wallpapersafari.com/w/sjCaDk)

- **Almas (u otra forma de pago)**

Dentro del juego el jugador/a debe subir de nivel para mejorar las estadísticas de su personaje. Para ello se necesita pagar un precio por cada nivel de habilidad que se quiera subir. Aparte, si el jugador/a quiere comprar armas, armaduras, objetos o nuevos hechizos también necesita una forma de pago. En la saga *Souls*, esta moneda son las **almas**. Cuando el jugador/a mata a un enemigo gana cierto número de almas, cuanto más poderoso el enemigo, mayor el número de almas obtenidas.

También, existen almas especiales necesarias para poder avanzar en la historia o forjar armas especiales y son las almas de los jefes finales. Al acabar con cada uno de ellos se le otorga al personaje principal una gran cantidad de almas normales a parte del alma especial.

- **Castigo por muerte**

La dificultad de los *Souls* no reside solo en lo complicados que sean los jefes finales o en la cantidad injusta de enemigos que pueden atacar al jugador/a, sino también en las consecuencias de morir.

Cuando el jugador/a muere reaparece en la última hoguera en la que ha descansado, sin almas, sin las bonificaciones que tuviera activas antes de morir y además todos los enemigos se restauran (esto último también ocurre cuando se descansa en una hoguera).

Pero no es ahí donde reside la verdadera penalización, ya que si la persona es capaz de volver al lugar donde ha muerto puede recuperar las almas que tenía antes de morir, pero en caso de morir antes de recuperarse sí que pierde las almas de forma definitiva.

- **Cantidad de jefes finales**

Esta serie y los juegos que se engloban en este subgénero son conocidos por albergar una gran cantidad de jefes finales. Lo habitual es que haya como mínimo un jefe por cada zona.

Dentro del total de jefes finales que el jugador/a puede enfrentar es necesario aclarar que algunos están clasificados como opcionales, es decir, que no es necesario pasar por ellos para avanzar en el juego. Pero no por eso la cantidad de jefes finales obligatorios es pequeña, siendo algo más de una docena de media solo con los obligatorios [54].

- **Combate metódico**

Entre las características que conforman al personaje figura el aguante o *estamina* que hace referencia al cansancio del personaje. Se mide mediante una barra similar a la de vida, de modo que cuando el personaje corre, rueda, se protege o ataca se resta cierta cantidad. Cuando la barra se consume el jugador/a no puede realizar acciones que consuman resistencia hasta que se haya restaurado la cantidad necesaria para hacer esa acción.

Este es un factor muy importante en los combates ya que el jugador/a necesita administrar bien la cantidad de resistencia que dispone para poder acercarse al jefe, atacarle y esquivar sus ataques, además de intentar protegerse.

- **Multijugador**

Una parte del diseño de los *Souls* es el sistema multijugador, el cual cuenta con 2 planteamientos: **modo cooperativo** o **modo *PvP*** (Jugador contra jugador). En el modo cooperativo (**Figura 16**) el jugador/a puede solicitar la ayuda de un personaje no jugable o de otro jugador/a en línea, pero únicamente para pelear contra el jefe final. Al acabar la pelea el ayudante se desvanece de la partida.

En el modo *PvP* el jugador/a puede invadir el mundo de otro jugador/a en línea o este puede ser invadido por otro jugador/a. El combate finaliza cuando uno de los jugadores muere, haciendo que el vencedor se quede con las almas, en el caso de *Dark Souls*, que el perdedor tenía acumuladas en ese momento.

Figura 16: Fotograma de gameplay cooperativo en Dark Souls III
(Fuente: https://www.youtube.com/watch?v=m8vtomwB7jg&t=306s&ab_channel=IGN)

En conclusión, se define el subgénero ***Soulslike*** como aquel cuyos juegos se caracterizan por integrar las características centrales de la serie *Souls*, adaptadas a la estética y coherentes con el diseño del videojuego en el que se quieren introducir.

4.4. Juegos referentes

Tras repasar el concepto de videojuego, los diferentes géneros que existen y sabiendo identificar un ***Soulslike*** es momento de analizar de forma breve los videojuegos de referencia que se han tomado para inspirar la idea y el desarrollo de este proyecto.

4.4.1. Dark Souls y Dark Souls III

Concretamente *Dark Souls: Prepare to Die Edition* y *Dark Souls III: The Fire Fades Edition* son las principales referencias del proyecto.

- ***Dark Souls: Prepare to Die Edition***

La historia de *Dark Souls* es compleja ya que, aparte de las conexiones con *Demon's Souls*, este universo tiene un *lore* muy extenso. A modo de resumen, el protagonista despierta en una celda donde encierran a los "**Huecos**" o no muertos. Pronto llega a sus oídos la profecía de que un elegido de los no muertos será el encargado de tocar las Campanas del Despertar y así descubrir cuál es el destino de los no muertos. El elemento que otorga luz y poder al mundo es la Primera Llama, pero esta se está apagando y con ello se acerca el fin de la Edad de Fuego.

Tras tocar las campanas, el protagonista comprende que su destino es suceder a *Gwyn, Señor de la ceniza* (**Figura 17**), arrojándose a la **Primera Llama** para prolongar la Edad de Fuego o matar a los dioses antiguos y apagar la llama para siempre, convirtiéndose en un Señor Oscuro que gobierne sobre la Edad de la Humanidad.

Figura 17: Pelea contra Gwyn, Señor de la Ceniza
(Fuente: https://darksouls.fandom.com/wiki/Gwyn,_Lord_of_Cinder)

Esta entrega sirve de referencia para el **uso de armas y escudos** tanto a nivel artístico como a nivel de mecánicas. Además, también se toman como inspiración las **zonas de descanso** (hogueras), la **interfaz** y el **diseño de niveles**.

- ***Dark Souls III: The Fire Fades Edition***

Aquí el protagonista no es un Hueco, sino un **Latente**, una criatura conectada al poder de la Primera Llama a través de las hogueras. Su objetivo en el reino de *Lothric* es volver a encender la Primera Llama, reducida a ascuas, y así evitar el comienzo de la Era de la Oscuridad. Para ello, el protagonista debe devolver a los **Antiguos Señores de la Ceniza**, héroes que se sacrificaron para mantener viva a la Primera Llama, a sus **tronos** (**Figura 18**), desde donde protegían a la Primera Llama.

Figura 18: Tronos de los Antiguos Señores de la Ceniza
(Fuente: https://as.com/meristation/2016/04/09/guia_pagina/1460192402_154628.html)

De aquí se toman como referencia las **diferentes fases** de **pelea** contra los **jefes** finales. Además, de este videojuego se toman como modelo el **sistema de vida, resistencia y maná**, así como algunos **patrones** de **comportamiento** de los **jefes finales**.

4.4.2. Bloodborne

Bloodborne es un videojuego de rol de acción en tercera persona desarrollado por *FromSoftware y JapanStudio* y dirigido por Hidetaka Miyazaki para *PlayStation 4* [55]. Este videojuego también es clasificado como un *soulslike* ya que reúne todo lo expuesto en el apartado 4.3.2. Género *Soulslike* pero adaptado a la historia que cuenta este título [56].

Además, *Bloodborne* aporta otros elementos al juego que le hacen tener un carácter propio, como el uso de armas de fuego para el combate o el sistema de "riesgo-versus-recompensa". Esta característica se enfatiza a través del sistema de recuperación de vitalidad, donde el personaje puede recuperar vida, dentro de un corto periodo de tiempo, realizando contraataques.

El videojuego se desarrolla en una ciudad de estilo victoriano donde sus habitantes han sido infectados con una enfermedad de transmisión sanguínea anormal (**Figura 19**). El protagonista despierta tras recibir un tratamiento conocido como la "sangre milagrosa", de modo que su objetivo es encontrar algo conocido como "sangre pálida" para poner fin a esa pesadilla.

Figura 19: Imagen ingame de Bloodborne
(Fuente: https://www.muycomputer.com/2020/08/12/bloodborne-remastered-ps5-pc/)

De este videojuego se toma como referencia el **comportamiento** de los **enemigos simples**.

4.4.3. Sekiro: Shadows Die Twice

Sekiro es un videojuego de rol y aventura en tercera persona, desarrollado por *FromSoftware*, distribuido por *Activision* y dirigido por Hidetaka Miyazaki [57]. El videojuego sigue la historia de Lobo, un *shinobi* del Japón Feudal que intenta vengarse de un clan de samuráis que atacó y secuestró a su maestro.

Dado que es de la misma desarrolladora y director que la saga *Souls* y *Bloodborne*, se aprecian algunos detalles que clasificarían a *Sekiro: Shadows Die Twice* como un *soulslike*, entre ellos la curva de aprendizaje y dificultad, la cantidad de jefes finales o el sistema de combate.

Sin embargo, es de los títulos que más elementos propios añade, como por ejemplo el utilizar un gancho para desplazarse (**Figura 20**) o que el arma principal sea una *katana*, en vez de tener libertad para equipar el arma deseada. Otra diferencia es que *Sekiro* no incluye sistema multijugador, pero gracias a su comunidad de jugadores y jugadoras se ha desarrollado un *mod* (modificación para el juego) que lo incorpora al juego.

Figura 20: Lobo desplazándose con el gancho
(Fuente: https://www.redbull.com/se-en/9-combat-tips-for-sekiro-shadows-die-twice)

De *Sekiro* se toman como referencias las **fases de pelea contra los jefes**, se **comparan** las **dificultades** de las peleas con la de los otros juegos de referencia y se tiene en cuenta la **sensación** que produce el **movimiento** del **personaje**.

4.5. *Elden Ring*, donde todo converge

Después de estudiar algunas de las obras más influyentes para este proyecto, para qué plataformas están disponibles y observar la evolución tanto de jugabilidad como gráfica hasta la actualidad, es momento de analizar cómo ha convergido todo en una obra única.

Elden Ring es el nuevo juego de rol de acción (ARPG) desarrollado por *FromSoftware* y distribuido por *Bandai Namco* para *PlayStation* 4, *PlayStation* 5, XBOX SERIES X|S, XBOX ONE y PC [58] [59]. Este título está dirigido por Hidetaka Miyazaki y cuenta con la colaboración de George R. R. Martin, conocido por ser el autor de "Canción de Hielo y Fuego" [60].

Si bien es cierto que R. R. Martin se ha encargado de crear una nueva mitología para esta entrega sigue habiendo conexiones con el mundo de los *Souls*, ya que Miyazaki aprovecha esta ocasión para llevar la fórmula de los *Dark Souls* al **mundo abierto**.

El juego se desarrolla en un mundo de fantasía oscura, "Las Tierras Intermedias", con grandes llanuras (**Figura 21**) y oscuras mazmorras conectadas de forma fluida y sin interrupciones. Esto contrasta con la serie *Souls* ya que, pese a contar con mapas de considerable tamaño, trata zonas muy concretas y cerradas.

Figura 21: Fotograma del Trailer-Gameplay de Elden Ring en el SumerFest 2021
(Fuente: https://www.youtube.com/watch?v=E3Huy2cdih0&ab_channel=BANDAINAMCOEntertainmentEurope)

En relación con el nuevo mundo abierto, ahora el jugador/a puede recorrer este mundo con una **montura** y **combatir** contra los enemigos **mientras cabalga**. Además, hay muchas más

posibilidades para plantear la forma de combate y la elección de equipamiento en comparación con los títulos anteriores: espectros que se pueden invocar, nuevos ataques especiales en las armas, mayor número de hechizos y armas, etc.

El autor, tras haberse puesto en la piel del jugador, afirma que todos los elementos principales y que hacen destacables a cada uno de los títulos anteriores han sido combinados a la perfección. El movimiento del personaje es muy fluido y satisfactorio; el desplazamiento por el mapa es sencillo, rápido e intuitivo; los enemigos están situados de forma adecuada a lo largo y ancho de todo el mapa; los jefes finales tienen un diseño muy cuidado y que supone un reto atractivo para el usuario.

Además, la dificultad del juego (factor de mucha polémica) se ve adaptada a las facilidades que ofrece el juego: mayor número de invocaciones que ayudan al jugador/a en zonas complejas; mayor número de hogueras (llamadas Gracias en este caso); señalización constante para seguir el flujo principal de la historia.

En conclusión, *Elden Ring* tiene todo lo que el autor quiere plasmar en el proyecto, tanto el apartado de diseño, mecánicas e IA como en las sensaciones que quiere transmitir a los jugadores y jugadoras.

5. Objetivos

El objetivo general del proyecto es el desarrollo de un videojuego **ARPG** estilo *Dark Souls* en el motor de juego ***Unreal Engine*** en su versión 5.0.3 a través de la programación de las mecánicas clásicas de este tipo de juegos y el diseño y desarrollo de tres tipos de Inteligencia Artificial: un enemigo básico, un jefe final y un personaje que ayude a la persona que juega.

A continuación, se detalla la lista de objetivos concretos a conseguir con el proyecto por parte del autor:

- Integrar elementos 3D externos al motor de juego para crear el mapa del juego.
- Integrar ***assets*** tanto de **modelado** como de **animaciones** para dar un nivel de acabado profesional a los personajes.
    - **Aprender a utilizar la herramienta de *retarget***[7] de este motor para aprovechar animaciones entre modelos con diferente forma o esqueleto.
- **Implementar las mecánicas que componen el movimiento y jugabilidad** de un personaje ARPG estilo *Dark Souls***.**
- **Implementar sistema de daño, vida y resistencia** típico de un *Soulslike.*
- **Diseñar y programar la inteligencia artificial** de:
    - Un enemigo simple.
    - De un jefe final con dos fases de combate.
    - De un personaje que ayuda al jugador.
- **Emplear efectos visuales y sonoros** que añadan inmersividad al videojuego.
- **Implementar una GUI y un menú** estilo *Soulslike* para mostrar la información necesaria al usuario.
- **Implementar el flujo de juego completo** para obtener un prototipo mínimo completo jugable.

Además, se pretende proyectar todo el proceso de desarrollo de *software* desde la planificación de tareas, análisis de herramientas y estudio de los casos de uso. Asimismo, se plantea expandir el conocimiento adquirido en el ABP sobre diseño y desarrollo de videojuegos, así como dominar las tecnologías que engloban el desarrollo con el motor *Unreal Engine*.

---

[7] *Animation Retargeting:* Es una herramienta que permite reutilizar animaciones entre personajes que usan el mismo esqueleto, pero su modelo tiene diferentes proporciones [103]. También es posible reutilizar animaciones entre esqueletos, pero deben compartir una jerarquía de huesos similar.

6. Metodología

Un paso previo al comienzo del desarrollo es definir la metodología de desarrollo a utilizar. Esta es una decisión crucial ya que marca la forma de trabajar, el ritmo de desarrollo y la evolución del producto. Además, también se deben especificar las herramientas relacionadas con la gestión de tareas, así como las empleadas para el manejo del repositorio donde se almacena el proyecto y el control de versiones.

6.1. Metodología de desarrollo de software

Tras analizar diferentes metodologías de desarrollo ágiles la empleada en este proyecto es ***Kanban*** [61]. Se emplea esta metodología ya que se basa en ideas muy simples que ayudan a visualizar el flujo de trabajo de forma sencilla, limitar el trabajo en curso (***WIP:*** *Work In Progress*) y modificar las actividades en curso según las necesidades del cliente.

A diferencia de otras metodologías como SCRUM [62], las iteraciones de tiempo definido no son obligatorias, haciendo así que el autor pueda ajustar el tiempo empleado para completar tareas que aporten valor al producto en función de la carga de trabajo externa que pueda tener. No obstante, sí que se plantean reuniones mensuales (cada 4 semas aproximadamente) con la tutora del trabajo para actualizar la situación del proyecto, hablar del trabajo realizado hasta la fecha y establecer unas tareas a completar (de forma ideal) antes de la próxima reunión y que aporten valor al producto. Asimismo, de forma paralela al desarrollo del producto se toman notas y apuntes que son necesarios para la redacción de la memoria conforme se avanza.

6.2. Gestión del proyecto

El factor clave que ha hecho que el autor se decante por utilizar *Kanban* como metodología es la flexibilidad y la forma de organización de tareas que ofrece su tablero. Dentro de él se pude limitar la carga de trabajo en progreso, el número de tareas que pueden estar a la espera de ser realizadas y el número de tareas que pueden estar atascadas.

También, gracias a estas características se pueden observar con facilidad cuellos de botella, que pueden suponer: un atasco en el desarrollo de alguna tarea, que la carga de trabajo es muy grande en ese momento o que hay tareas que están estancadas o que no han sido revisadas y que bloquean el avance del desarrollo.

A continuación, se presenta la organización de las columnas que componen el tablero Kanban de este proyecto:

- ***Product Backlog***: En esta columna se sitúan todas las tareas que se deben implementar a lo largo del desarrollo. Estas tareas vienen derivadas del análisis realizado en el apartado 7. Análisis y especificación.
- ***Buffer***: Aquí se colocan las tareas que se deben realizar antes de la próxima reunión con la tutora.
- ***In Progress***: Las tareas que se encuentran en este espacio son las tareas que se están realizando en el momento.
- ***On Hold***: Todas las tareas que estaban en desarrollo que no se pueden completar por algún motivo pero que no sean incapacitantes para el progreso del proyecto se deben mover a esta columna hasta que se encuentre la solución. En ese momento la tarea pasa de nuevo a la columna de *In Progress* para que sea finalizada.
- ***To Be Reviewed***: Toda tarea que haya sido finalizada debe pasar por este estado antes de considerarla como acabada. Este es el momento en el que se prueba que la funcionalidad asociada a la tarea funciona correctamente y puede ser integrada al proyecto.
- ***Done***: En esta columna se encuentran todas aquellas tareas que han sido finalizadas, comprobadas e integradas correctamente en el proyecto.
- ***Closed***: Columna donde se mueven las tareas para que sean cerradas y no aparezcan como pendientes en el proyecto.

Cada columna cuenta con un límite de tareas que puede tener en un mismo momento. Esto sirve para no saturar la carga de trabajo, evitar tareas que se suponen en progreso pero que están ociosas y no acumular muchas tareas en ***On Hold*** para así priorizar la corrección de estas.

Esta organización de tablero ayuda al autor a identificar las tareas que más valor aportan al producto y detectar de forma visual donde se están produciendo cuellos de botella, pudiendo reajustar el número de tareas que se permiten en cada columna para deshacer ese atasco.

Para la gestión de tareas se emplea el tablero de tareas de ***GitLab*** [63]. Se hace uso de esta herramienta ya que cuenta con todas las funciones que se necesitan para el control de tareas y, además, al ser un servicio más de *Gitlab*, la creación de ramas a partir de las tareas (***Issues***) es muy cómoda. También cuenta con características que ayudan al monitoreo de tiempo empleado, estimaciones, etiquetas y pesos para cada tarea. La **Figura 22** muestra el tablero empleado en el desarrollo de este proyecto.

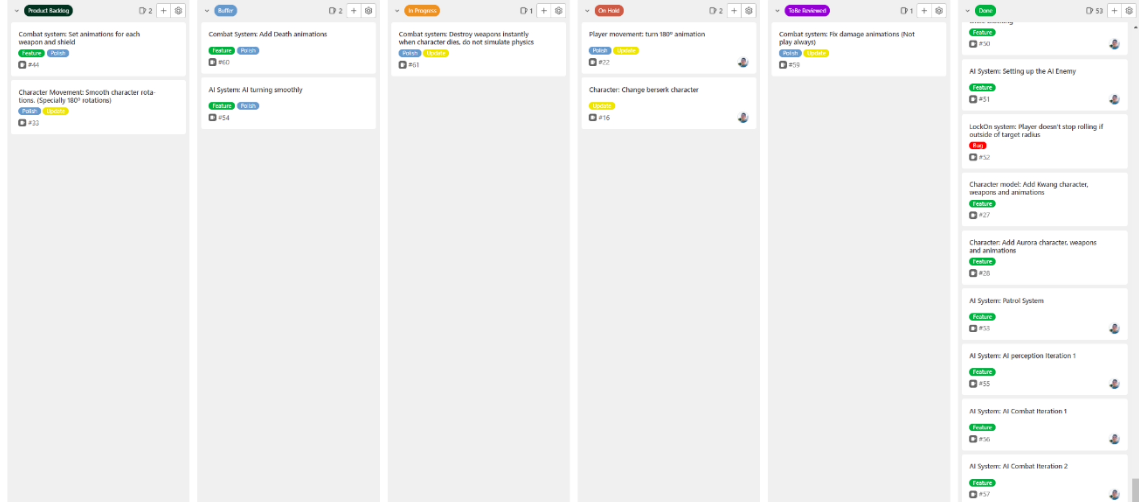


Figura 22: Tablero de tareas de Gitlab
(Fuente propia)

En cuanto al registro de horas empleadas en el proyecto se hace uso, inicialmente, de ***Toggle*** [64]. Con esta aplicación se crea una lista de *logs* donde se indica el proyecto al que pertenece la tarea, una etiqueta para indicar a qué fase del desarrollo pertenece y el tiempo empleado en esta (**Figura 23**). Además, sirve para visualizar el rendimiento de cada semana y también se puede hacer la estimación de tareas desde aquí.

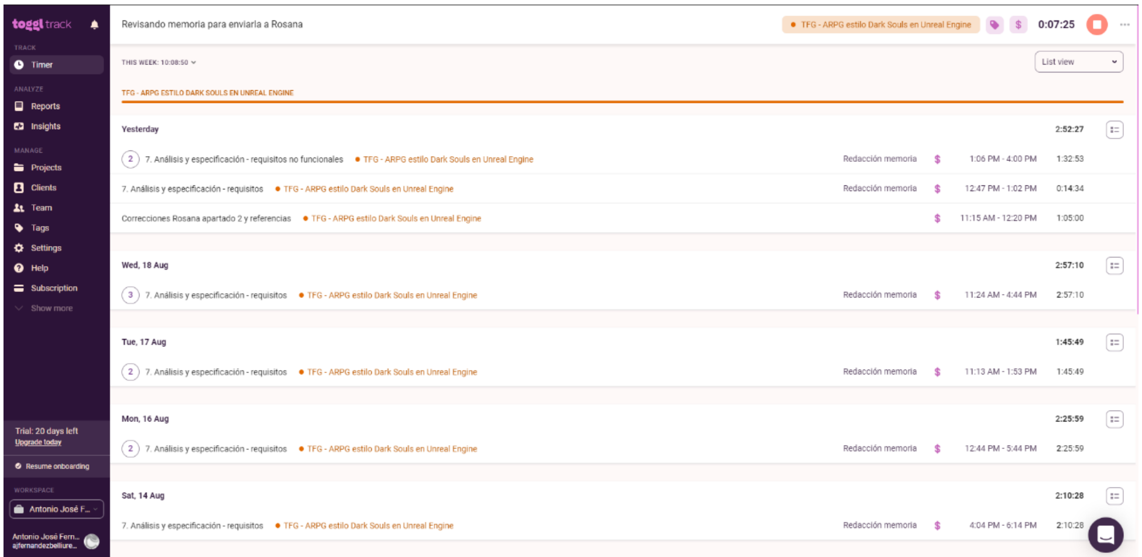


Figura 23: Tablero de tareas de Toggle
(Fuente propia)

Sin embargo, tras realizar el ABP, el autor decide adoptar ***Clockify*** como herramienta de registro de trabajo [65]. *Clockify* incluye todo lo expuesto en el párrafo anterior y se puede utilizar como extensión del navegador, con lo que tiene una gran compatibilidad con diferentes servicios

relacionados con el desarrollo de software y el autor se encuentra más cómodo con ella. La **Figura 24** muestra cómo se representan los registros de tiempos en esta plataforma.

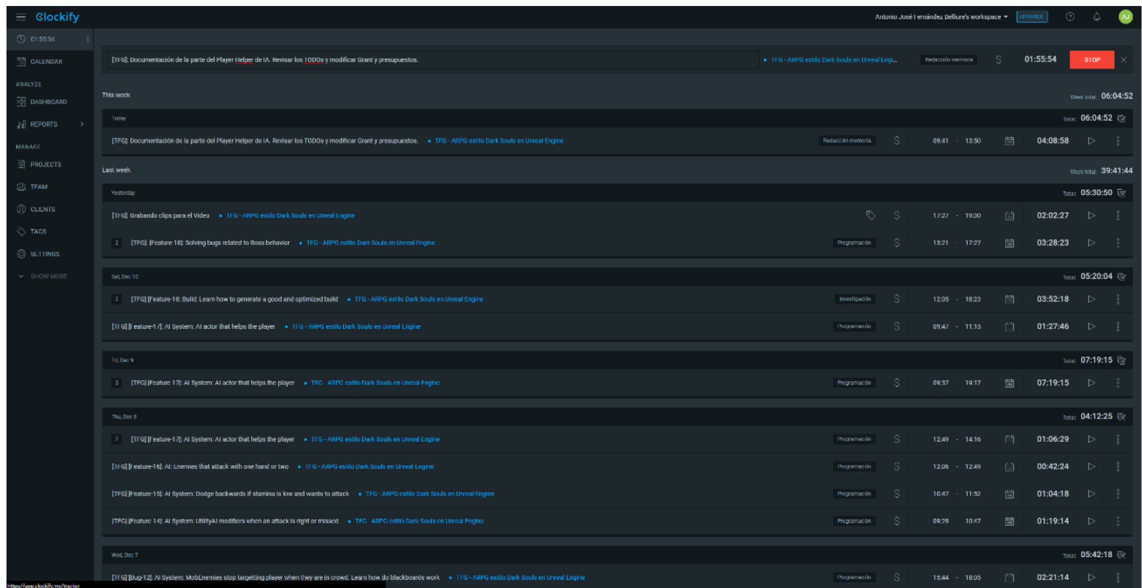


Figura 24: Tablero de registro de tareas de Clockify
(Fuente propia)

6.3. Repositorio y control de versiones

En lo referente al almacenamiento, respaldo y control de versiones del proyecto se emplea ***GitLab*** [66]. Esta herramienta de repositorios ***git*** cuenta con almacenamiento de hasta 10GB por repositorio, público o privado, y no tiene límite de tamaño de archivos. Estas dos características son importantes ya que los proyectos realizados con un motor como *Unreal Engine* suelen tener un tamaño considerable.

La importancia de utilizar una herramienta de control de versiones reside en, aparte de tener un respaldo del proyecto en un repositorio, que se almacenan versiones anteriores de este. Esto quiere decir que, si el proyecto sufre algún error que no se encuentra a simple vista, se pueden analizar las versiones anteriores para averiguar dónde está el error y solucionarlo en la versión más reciente.

Además, *GitLab*, cuenta con otras herramientas integradas como un tablero propio para gestionar las tareas, creación automática de ramas a partir de las tareas que se incluyan en el proyecto y cierre automático de las ramas mediante ***Merge Requests***. Esta última actividad, la de cierre automático, la realiza una herramienta que revisa nuevo código implementado en otra rama antes de juntarlo al grueso del proyecto, para así evitar conflictos entre versiones y evitar integrar tareas que puedan producir errores.

En cuanto al almacenamiento de los modelos 3D, texturas y sonidos se emplea Google Drive. La razón de almacenar copias de seguridad y demás recursos multimedia fuera del repositorio donde se encuentra el código del videojuego se debe a que estos archivos ocupan mucho espacio, además de que esta forma el repositorio queda limpio de archivos que pueden no ser compatibles con *git*.

7. Análisis y especificación

En este apartado se hace un estudio para detallar los requisitos que debe cumplir el proyecto y que deben estar implementados para que se desarrolle un producto jugable y de calidad. Para ello primero se realiza un análisis de casos de uso, los cuales son desglosados posteriormente en requisitos, tanto funcionales como no funcionales y que sirven para que el autor tenga claro todo lo que se tiene que desarrollar para que el producto sea funcional. También, este análisis sirve para delimitar el alcance del producto, así como detectar aquellas características que son posibles o imposibles de implementar.

7.1. Casos de uso

Los casos de uso sirven para realizar el modelado de un sistema desde el punto de vista del usuario/a. El objetivo de estos es explicar la manera en la que se usará el sistema mediante diagramas que describen la interacción entre un agente externo (actor) y el sistema.

Los diagramas UML *(Unified Modelling Language)* de casos de uso permiten definir los límites del sistema, expresar el comportamiento deseado del sistema y establecer las necesidades que quieren satisfacer los usuarios [67]. Con el fin de que el diagrama sea comprensible se muestra la **Tabla 5**, que explica los símbolos empleados y el significado que tiene cada uno dentro de este:

Tabla 5: Leyenda de símbolos del diagrama UML de Casos de uso

FIGURA	ELEMENTO	DESCRIPCIÓN
Actor	Actor	Alguien o algo que interactúa sobre el sistema que se está desarrollando.
	Caso de uso	Comportamiento del sistema que estamos desarrollando que produce un resultado de interés para algún actor. No describe solo una funcionalidad sino también una interacción entre un actor y el sistema.
<<Include>>	Asociación de caso de uso de tipo <<Incluir>>	Un caso de uso base incorpora explícitamente el comportamiento de otro caso de uso.
<<Extend>>	Asociación de caso de uso de tipo <<Extender>>	El caso base puede necesitar o no la funcionalidad del caso de uso extendido.

A continuación, se presenta el diagrama de casos de uso para este proyecto (**Figura 25**). En él se detallan las acciones que un actor, en este caso un jugador o jugadora, quiere hacer dentro del videojuego. A partir de estos casos de uso, se detallan los requisitos que se deben implementar para satisfacer las necesidades declaradas por el usuario en dicho diagrama.

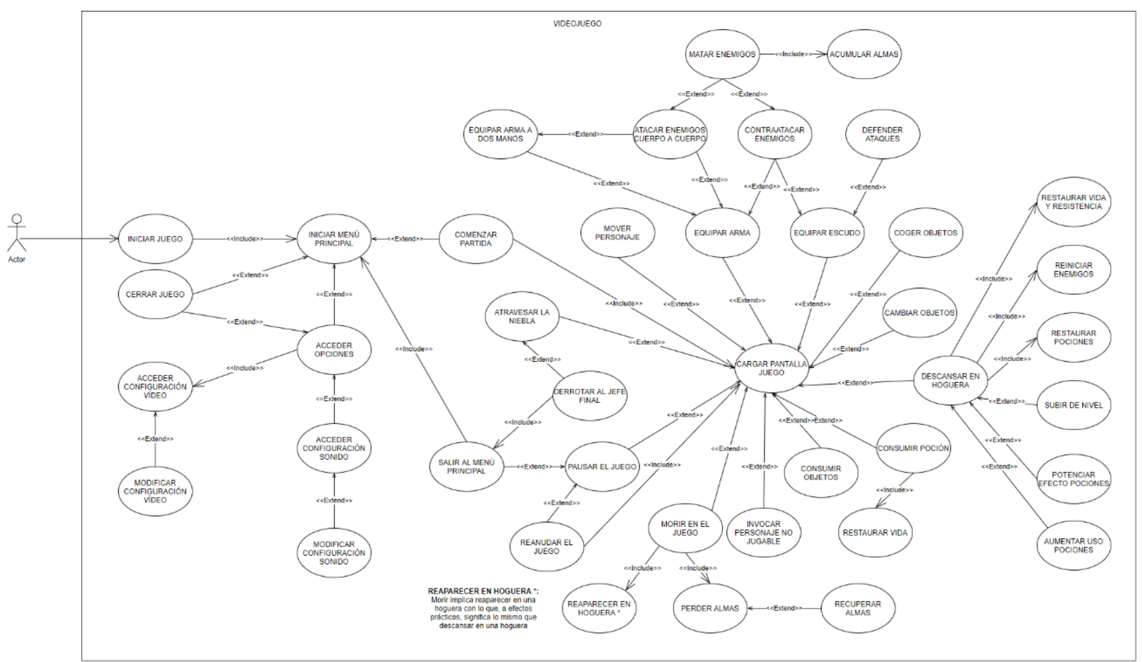


Figura 25: Diagrama de casos de uso
(Fuente propia)

7.2. Especificación de requisitos

Un requisito es una característica que el sistema debe tener para ser aceptado o algo que el sistema debe hacer para el usuario. Los requisitos surgen de las necesidades del cliente, es por ello que un requisito describe una utilidad para el usuario/a.

Una vez estudiados los casos de uso se procede a listar los requisitos que debe cumplir el proyecto. Cada caso de uso está identificado por un identificador único y a partir de cada caso de uso se derivan uno o más requisitos. Estos también están identificados por un identificador único, a la vez que se distribuyen entre requisitos funcionales y no funcionales.

Dada la extensa cantidad de casos de uso y requisitos del proyecto, el autor adjunta en el anexo A. Especificación de requisitos detallada, la especificación en tablas de los requisitos con el fin de no romper el flujo de lectura del documento.

8. Elección de herramientas

Una vez se conocen las necesidades que tiene que cubrir el proyecto es el momento de analizar las diferentes herramientas a disposición del autor para desarrollar la solución adecuada a los problemas que presenta el proyecto.

8.1. Motor de juego

Es el elemento principal necesario para llevar a cabo la solución. Es el núcleo del desarrollo y sobre el cuál se va a formar el videojuego ya que es el software que tiene las rutinas de programación que permiten el diseño, la creación y el funcionamiento de este. En la actualidad existen varias alternativas para desarrollar videojuegos sin necesidad de crear un motor de juego propio. Por ello en este subapartado se exponen los motivos que decantan al autor a elegir ***Unreal Engine*** como motor de juego para este proyecto.

8.1.1. Unreal Engine

Unreal Engine es un motor de juego multiplataforma creado por *Epic Games* que permite el desarrollo de videojuegos 2D y 3D en tiempo real [10] (**Figura 26**). Dado que está desarrollado con C++ ofrece un alto grado de portabilidad y por ello es una de las herramientas más utilizadas hoy en día para el desarrollo de videojuegos.

Otro de los motivos por el que es un motor ampliamente utilizado es su motor gráfico, el cual cuenta con una calidad gráfica superior al de *Unity*. Características como el sistema de iluminación dinámica o de partículas ayudan al desarrollador a lograr un nivel de profesionalidad y acabado del producto elevados.

En el momento en que se redacta este documento, la última versión estable del motor es la 4.27 [68], lanzada el 19 de agosto de 2021. Sin embargo, ya está disponible el acceso temprano (*early access*) de ***Unreal Engine 5*** [69]**.** La quinta generación del motor cuenta con un rediseño de su interfaz, así como mejoras a nivel de rendimiento, iluminación y animación.

Además, *Unreal Engine* ofrece un sistema de programación mediante *scripting* llamado ***Blueprint Visual Scripting*** [70]**.** Este sistema permite al desarrollador programar mediante nodos y crear las entidades del juego desde el propio editor del motor. También, es posible integrar el desarrollo con C++ y *Blueprints*, combinación de herramientas más potente y flexible que el lenguaje de script en C# y *SahderLab* [71] utilizados en *Unity.*

Por todo lo expuesto y porque es de uso **gratuito** (restringido a las condiciones expuestas en el apartado 2.3. Estudio de mercado) es el motor de juego elegido para este proyecto.

Figura 26: Imagotipo de Unreal Engine
(Fuente: https://es.wikipedia.org/wiki/Unreal_Engine)

8.2. Recursos 3D externos

8.2.1. Bazar de Unreal Engine

El bazar de *Unreal Engine* es una tienda donde los desarrolladores ponen a la venta recursos creados por ellos para que puedan ser integrados en los proyectos implementados en este motor. Se pueden encontrar infinidad de recursos como: artefactos 3D simples para ambientar un escenario, niveles y escenarios completos, modelados de armas, conjuntos de animaciones, modelados 3D de personajes, materiales, VFX, SFX, etc.

Este bazar es una fuente muy rica de recursos y que puede dar el nivel de acabado deseado a un proyecto incluso si el desarrollador necesita algún elemento que no es capaz de crear por sí mismo. Un ejemplo importante de los recursos disponibles en este bazar son los personajes utilizados en el videojuego ***Paragon.*** Se encuentran disponibles de forma **gratuita** y contienen los modelos 3D con los esqueletos integrados, animaciones y diferentes aspectos.

El autor utiliza el bazar de Unreal Engine para obtener recursos 3D, SFX y VFX necesarios para la creación del mapa, armas, personajes y acabado profesional del proyecto.

8.2.2. Mixamo

Mixamo es una compañía de tecnología de gráficos 3D [72] (**Figura 27**). La compañía se centra en desarrollar y proporcionar servicios web relacionados con la animación, el *rigging* y el modelado 3D de personajes. Para este proyecto es interesante el hecho de que ofrece modelos de personajes 3D de forma gratuita y también se pueden obtener animaciones compatibles con dichos modelos. Este servicio se utiliza en el proyecto en conjunto con el *Bazar de Unreal Engine* para obtener modelos 3D de personajes y sus animaciones.

Figura 27: Imagotipo de Mixamo
(Fuente: https://adobe.fandom.com/wiki/Mixamo)

8.3. Entornos de desarrollo integrado (IDEs)

Dado que el proyecto está orientado a la parte de programación de un videojuego, es necesario seleccionar las herramientas que hagan más eficiente y cómoda esta parte del desarrollo. De ello se encargan los Entornos de desarrollo integrado, que contienen todas las herramientas necesarias para agilizar el proceso de programación, depuración y lanzamiento de la aplicación que se está desarrollando.

8.3.1. Visual Studio Community 2022

Visual Studio Community es el IDE elegido por el autor para el desarrollo del proyecto. Es un IDE extensamente utilizado no solo en la industria de los videojuegos sino en el desarrollo de aplicaciones en general. Tiene soporte para C++ y es compatible con *Unreal Engine*.

El *debugger* funciona bastante bien ya que pertenece a los mismos desarrolladores del compilador de C++ de Microsoft Visual Studio. Además, ha sido utilizado muchos años por ***Epic Games***, con lo que estos ponen a disposición ayudas como la *UnrealVS Extension* [73].

Sin embargo, actualmente existe ***Rider***, desarrollado por *JetBrains* [74].Este nuevo IDE está ganando fuerza ya que cuenta con una mayor compatibilidad con *Unreal Engine* y C++, entre otras cosas. Es más ligero que VS, ofrece mejor detección de errores y autocompletado y soporta sistemas de control de versiones punteros. Además, detecta las clases que son *BluePrints* y de qué clases hereda, crea vínculos entre esas clases y permite abrirlas de forma sencilla en el editor

de *Unreal Engine*. Sin embargo, no es el elegido por el autor ya que es de pago y el depurador carece de algunas funciones avanzadas de depuración.

8.4. Software de sonido

Los sonidos en los videojuegos aportan ese toque de inmersión que hace que el jugador/a sienta que está dentro del videojuego. Es extraño, e incluso a veces irritante, ir caminando y no escuchar los pasos, golpear a un enemigo y que no suene un sonido de espada o empezar una pelea contra un jefe final y que no comience una épica canción que acompañe la batalla.

8.4.1. Librerías de sonidos

FreeSound [75] conforma una gran base de datos colaborativa de sonidos regidos bajo la licencia de *Creative Commons* [76]. Esta iniciativa sostenida por la Universidad Pompeu Fabra de Barcelona aporta todo tipo de sonidos, onomatopeyas y fragmentos musicales listos para ser usados. Este servicio se utiliza en el proyecto para obtener aquellos sonidos necesarios para los efectos especiales de sonido del videojuego.

En combinación con *FreeSound* y el Bazar de Unreal, el autor hace uso de la sección de ***GameAudioGDC*** disponible en *Soniss* [77]. Dicho apartado contiene una gran cantidad de recursos sonoros empleados en videojuegos y que han sido puestos a disposición de los desarrolladores de forma gratuita y bajo licencia de libre uso.

8.4.2. Audacity

Audacity es un programa de grabación y edición de audio gratuito de código abierto. Es una herramienta útil para editar los sonidos obtenidos mediante las herramientas anteriormente mencionadas, ya que en ocasiones pueden contener ruido o bien almacenar varios sonidos en una sola pista de audio, siendo necesaria su edición y retoque para adaptarlos al proyecto.

9. Implementación

Una vez seleccionadas las herramientas de trabajo y analizadas las necesidades a resolver, llega el momento de empezar la implementación del proyecto. En este apartado se documenta el desarrollo del videojuego explicando qué soluciones ha encontrado el autor a los requisitos planteados anteriormente y qué tecnologías he empleado para ello.

Antes de profundizar en los detalles de la implementación del videojuego, se ofrece una vista más detallada del motor *Unreal Engine* y de las principales tecnologías que el autor ha utilizado. Esto ayuda al lector/a ya que introduce conceptos que se emplean con frecuencia en los siguientes puntos del apartado.

Posteriormente, la implementación se encuentra dividida en tres grandes apartados, facilitando el entendimiento y seguimiento del desarrollo al lector. El primero de ellos engloba el desarrollo completo del personaje controlado por el jugador/a, base primordial del videojuego. En segundo lugar, se encuentra la implementación de la Inteligencia Artificial, otro pilar fundamental ya que sin ella la jugabilidad no existiría. En tercer lugar, el control de flujo de juego y manejo del nivel, necesario para que haya un producto mínimo completo y jugable.

9.1. Introducción a *Unreal Engine*

En este punto se amplía la información sobre el motor *Unreal Engine*, brevemente introducido en el apartado 8.1. Motor de juego, profundizando más en el motor y en las tecnologías empleadas para la implementación del proyecto.

Unreal Engine es un motor que lleva alrededor de 24 años de desarrollo, durante los cuales ha lanzado varias versiones con diferentes modificaciones en su implementación. Este apartado se centra en la versión 5.0.3, lanzada en julio de 2022 [78].

Actualmente, *Unreal Engine* cuenta con una estructura basada en componentes, pero todavía mantiene una estrecha relación con la programación orientada a objetos. El hecho de que aún se mantenga este paradigma es el resultado de la larga trayectoria del motor y el alto precio que supondría hacer un cambio tan grande como cambiar de paradigma por completo. Como resultado de este paradigma, la jerarquía de clases básica de entidades de juego en *Unreal Engine* está definida de la siguiente manera.

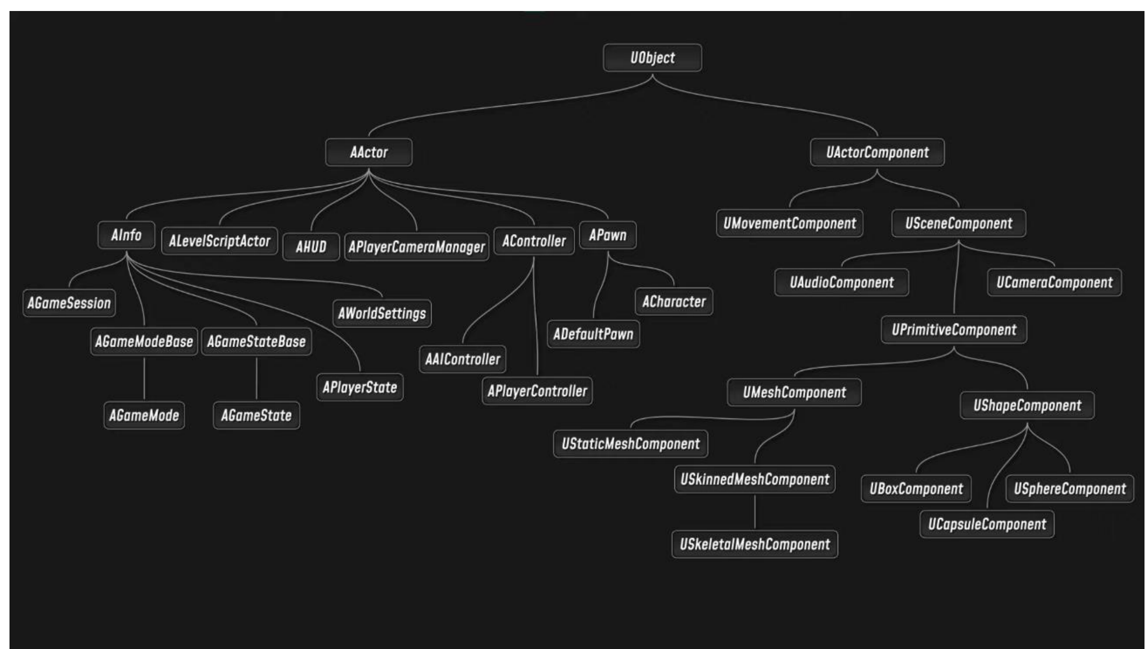


Figura 28: Jerarquía de clases básica de entidades de juego en Unreal Engine.
(Fuente: Alex Forsythe, https://www.youtube.com/watch?v=IaU2Hue-ApI)

9.1.1.1. Actores en Unreal Engine

El primero de los dos grandes grupos ilustrado en la **Figura 28: Jerarquía de clases básica de entidades de juego en Unreal Engine.** es el de los Actores. La clase ***AActor*** es la base de la cual heredan todos los objetos que se sitúan en el mundo del juego. En el siguiente nivel, las clases más relevantes para este proyecto son ***APawn*** y ***AController***.

La clase *APawn* es la encargada de poseer a las entidades dirigidas por el jugador/a o por la IA. Son la representación física en el juego de la persona que juega o de criaturas dentro del nivel. De ella heredan las clases ***ADefaultPawn*** y ***ACharacter***. Los actores de tipo *ACharacter* ya tienen por defecto asignados una malla, un componente de colisión y la lógica básica para el movimiento humanoide.

Por otro lado, la clase *AController* son actores sin propiedades físicas que pueden poseer a un *APawn* para controlar sus acciones. Estos controladores pueden ser de tipo ***APlayerController*** (actor controlado por un ser humano) o ***AAIController*** (actor controlado por inteligencia artificial).

9.1.1.2. Componentes en Unreal Engine

Dentro de *Unreal Engine,* un componente es funcionalidad que se le asigna a una entidad. Estos componentes son independientes de la entidad a la que se le asigna, de modo que varias entidades pueden contener el mismo componente para llevar a cabo la misma funcionalidad [79].

Asimismo, los componentes son independientes unos de otros, con lo que se pueden asignar componentes diferentes a una entidad para modelar su comportamiento. Con esta independencia se consigue un sistema con mayor grado de escalabilidad y reusabilidad. Por otro lado, las entidades en sí pueden implementar comportamientos generales de la entidad, los cuales serán compartidos por todas las entidades del mismo tipo.

Algunos de los componentes principales son aquellos relacionados con el movimiento de entidades, propiedades físicas para el cálculo de colisiones, cámara y la malla que representa al personaje.

9.1.1.3. Interfaces en Unreal Engine

Las interfaces en *Unreal Engine* son empleadas para asegurar que un conjunto de entidades no relacionadas implementa una serie de funciones comunes, pero con comportamientos diferentes [80]. Pueden ser útiles, por ejemplo, para definir el comportamiento como consecuencia de la colisión entre un *trigger volume* (colisión que dispara un evento) y otra entidad (la del jugador/a, por ejemplo). Dependiendo de si es una trampa, una zona de alerta de enemigos o un objeto que otorga puntos la reacción será distinta, pero todos ellos compartirían un evento llamado "ReaccionAlTrigger" con implementaciones diferentes.

9.1.2. Blueprints

Los *Blueprints* son un lenguaje de scripting visual propio de *Unreal Engine*. Está basado en la unión de nodos para crear la lógica de los elementos del juego desde el propio editor de *Unreal Engine* [81].

Los *Blueprints* están basados en clases implementadas en C++, con lo que son una tecnología altamente empleada en el mundo de los videojuegos ya que permite a los diseñadores y diseñadoras extender los comportamientos base implementados por los programadores/as en C++, así como crear nuevos comportamientos de forma sencilla.

Por otro lado, hace que el prototipado de mecánicas sea más rápido ya que el hecho de unir nodos es más intuitivo que programar las mecánicas en un nivel más bajo con C++. Asimismo, permite que el desarrollo de un proyecto sea más organizado y ágil ya que mientras que los programadores/as se centran en tareas críticas de rendimiento, desarrollo de nuevas tecnologías o desarrollo de herramientas con C++, el apartado de diseño puede ir prototipando o incluso desarrollando las mecánicas del juego sin depender del equipo de programación.

Finalmente, los *Blueprints* son compilados por una máquina virtual que traduce su código visual y la lógica implementada a código de C++. Además, para compilar los *Blueprints* no es necesario cerrar editor de *Unreal Engine* ni la aplicación, con lo que es bastante ágil.

9.1.2.1. C++ vs. Blueprints

Para implementar lógica y funcionalidad en *Unreal Engine* se puede emplear tanto programación con C++ como programación visual con *Blueprints*. ¿Cuál se debería emplear para hacer un videojuego? Como todas las decisiones dentro del desarrollo de software tienen sus ventajas e inconvenientes. A continuación, la **Tabla 6** muestra algunas ventajas e inconvenientes de los *Blueprints* y C++:

Tabla 6: Pros y contras de usar Blueprints o C++

	Blueprints	C++
Tiempo de ejecución	Llamadas a nodos, funciones o macros con muchos nodos y mucha lógica puede ser costoso.	Llamadas a líneas de código normalmente rápidas.
Tiempo de compilación	Más rápido que C++ sin necesidad de reiniciar ni el juego ni el motor.	Más lento que *Blueprints*. En la mayoría de los casos es necesario reiniciar el motor y el juego.
Diseño	Posible tendencia a crear grafos gigantes casi imposibles de seguir la traza.	Control preciso sobre qué exponer al usuario y cómo diseñar tu API.
Acceso externo	Menos funcionalidades del motor dedicadas a compartir información entre sistemas.	Variables y funciones expuestas correctamente fácilmente accesibles por todos los sistemas.
Multijugador en línea	Sistema de réplica diseñado para juegos pequeños y con poca complejidad.	Mayor control del consumo de ancho de banda y tiempo de respuesta.
Matemáticas	Rendimiento lento y difícil de diseñar si las operaciones matemáticas son complejas.	Mejor rendimiento y diseño de operaciones matemáticas complejas.
Control de versiones	Puede dar conflicto fácilmente.	Al almacenarse como texto plano, varias personas

		pueden trabajar sobre el mismo fichero sin problemas.
Creación / Prototipado	Más rápido añadir variables y funciones para prototipar.	Creación lenta de nuevas clases, datos y funciones.
Iteración	Modificar variables y lógica sin necesidad de recompilar el juego o el motor.	Necesidad de recompilar el proyecto para que los cambios tengan efecto.
Flujo	Si el grafo no es complejo es más sencillo seguir las llamadas a los nodos durante la ejecución.	Complejo seguimiento de llamadas a funciones durante la ejecución.
Edición	Diseñadores/as y artistas pueden modificar el comportamiento de *assets* que usen *Blueprints* sin necesidad de saber C++.	Lento proceso para añadir *assets* con datos y lógica definidos desde C++ directamente para hacer pruebas.
Almacenar datos	Más sencillo y seguro almacenar datos en un *Blueprint*.	Si no se tiene cuidado, puede ser desastroso un mal control de los datos.

Si bien es cierto que hay que vigilar el rendimiento del proyecto muy de cerca, una vez se conocen las ventajas e inconvenientes de cada lenguaje, se debe sopesar cuál casa mejor con las necesidades del proyecto. Por ello, si el proyecto se está desarrollando con *Blueprints* y este crece considerablemente en tamaño, entra en juego la posibilidad de refactorizar partes del proyecto a C++ para mantener un buen rendimiento. Todos los detalles sobre qué lenguaje usar y cómo combinar C++ con *Blueprints* en un proyecto se encuentra en la documentación oficial del motor [82].

9.1.3. Animation Blueprints

Los *Blueprints* forman parte intrínseca del motor *Unreal Engine* y extienden su funcionalidad más allá del almacenamiento de datos o del diseño de lógica para las entidades del juego. En este caso, los *Animation Blueprint* se encargan de las tareas relacionadas con las animaciones y el proceso de animación, haciéndolo más sencillo e intuitivo para el usuario [83].

Al igual que los *Blueprints* normales, estos también pueden ser programados en C++, pero únicamente se puede programar uno de los dos grafos que componen a este tipo de *Blueprint*: el ***EventGraph***. El otro grafo principal es el ***AnimGraph***. Ambos grafos se detallan en los siguientes puntos.

9.1.3.1. EventGraph

Es un grafo genérico como el que se puede encontrar en cualquier *Blueprint* y que dispara una serie de eventos relacionados con las animaciones. Es el encargado de almacenar los estados en los que puede estar el personaje: corriendo, andando, rodando, protegiéndose, atacando, etc.

Normalmente los datos que determinan los estados son recogidos desde el *Blueprint* o clase de C++ que contiene la lógica de comportamiento de la entidad que va a ser animada. Posteriormente, estos datos son transmitidos al *AnimGraph*, el cual los emplea para determinar las transiciones y el flujo de las animaciones.

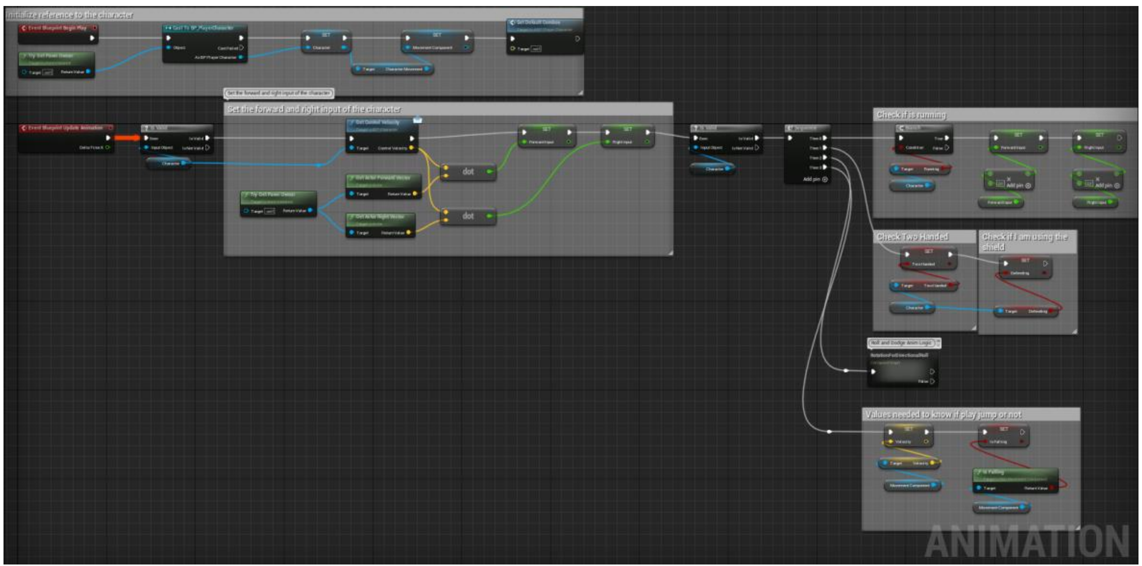


Figura 29: EventGraph del Blueprint del personaje principal.
(Fuente propia)

La **Figura 29** muestra cómo es el *EventGraph* del personaje principal y cómo se recogen los datos necesarios para el flujo de las animaciones.

9.1.3.2. AnimGraph

Este grafo es el encargado de contener toda la lógica necesaria para el flujo de las animaciones, manejo de transiciones de un estado a otro y el comportamiento entre ellas.

En este grafo el foco principal lo tienen las **máquinas de estados** y el nodo ***Output Pose*** (nodo marrón de la **Figura 30**), el cual reproduce la postura final que tiene que mostrar el personaje. Las máquinas de estados están formadas generalmente por **Estados** y **Transiciones** (**Figura 31**).

Los estados suelen contener una secuencia de animación que es reproducida cuando el personaje se encuentra en ese estado. Las transiciones contienen las reglas lógicas para poder cambiar de un estado a otro además de diferentes parámetros que definen cómo va a ser la transición entre estados (más o menos brusca, más o menos rápida, en determinado momento de la animación que se está reproduciendo, etc.).

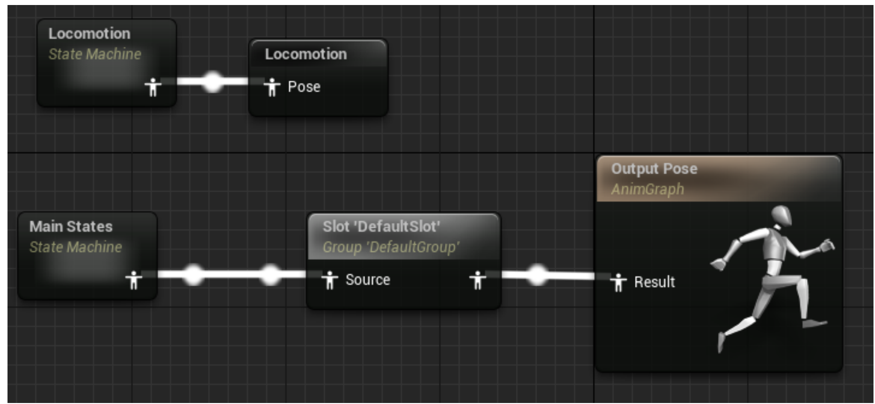


Figura 30: AnimGraph del Blueprint del personaje principal.
(Fuente propia)

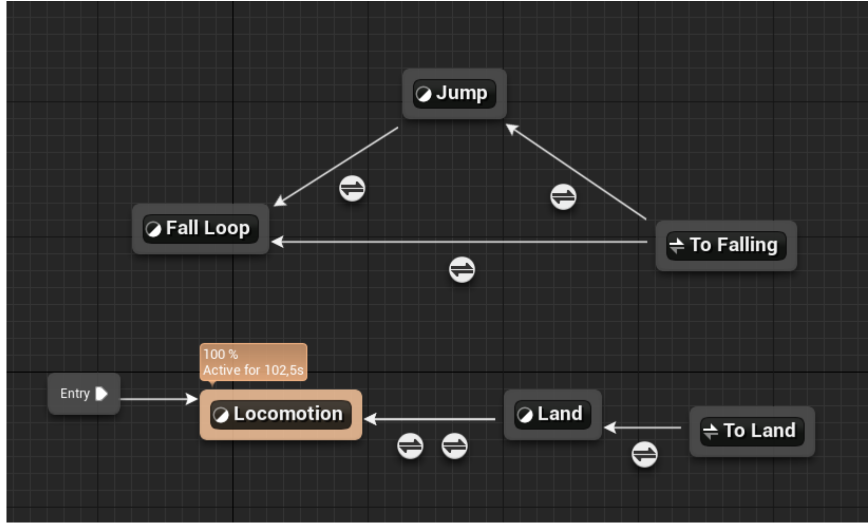


Figura 31: Máquina de estados prinpcipal del AnimGraph
(Fuente propia)

Más adelante, en la implementación, se muestran algunas funcionalidades que se pueden llevar a cabo dentro de las máquinas de estados, del *AnimGraph* y del *EventGraph*, pero si el lector/a desea averiguar más sobre estos términos puede consultar la documentación oficial de *Unreal Engine* [84].

9.2. Proceso de *Retarget*

Una vez introducido el motor y algunas de sus herramientas al lector/a, el autor procede a documentar la implementación del videojuego. El primer paso es el proceso de *retarget* para la obtención de las animaciones empleadas en el movimiento de los personajes.

Este proceso permite al usuario compartir información de las animaciones entre diferentes mallas esqueléticas (diferentes mallas con esqueletos diferentes) sin necesidad de clonar dichas animaciones fuera del motor con otras herramientas. Para este proceso se emplean los *assets* ***IK Rig*** e ***IK Retargeter***.

9.2.1. IK Rig Asset

Este artefacto del motor permite al usuario definir cadenas de huesos dada una malla esquelética [85]. Para definir estas cadenas de huesos el usuario/a debe asignarles un **nombre**, un **hueso de inicio** y un **hueso de fin**. Si el usuario/a desea transferir una animación de una malla esquelética a otra, debe crear **un *IK Rig* para cada una**. Además, para que el proceso de *retarget* sea automático, los nombres de las cadenas de huesos deben ser el mismo en ambos *assets*. Asimismo, se debe elegir qué hueso va a ser el hueso raíz en el que se va a basar el proceso de *retarget*. En este caso el autor selecciona la **pelvis** como hueso base ya que todos los demás huesos dependen de este.

La **Figura 32** muestra el resultado de generar diferentes cadenas de huesos para el proceso de *retarget* sobre la malla esquelética del personaje principal.

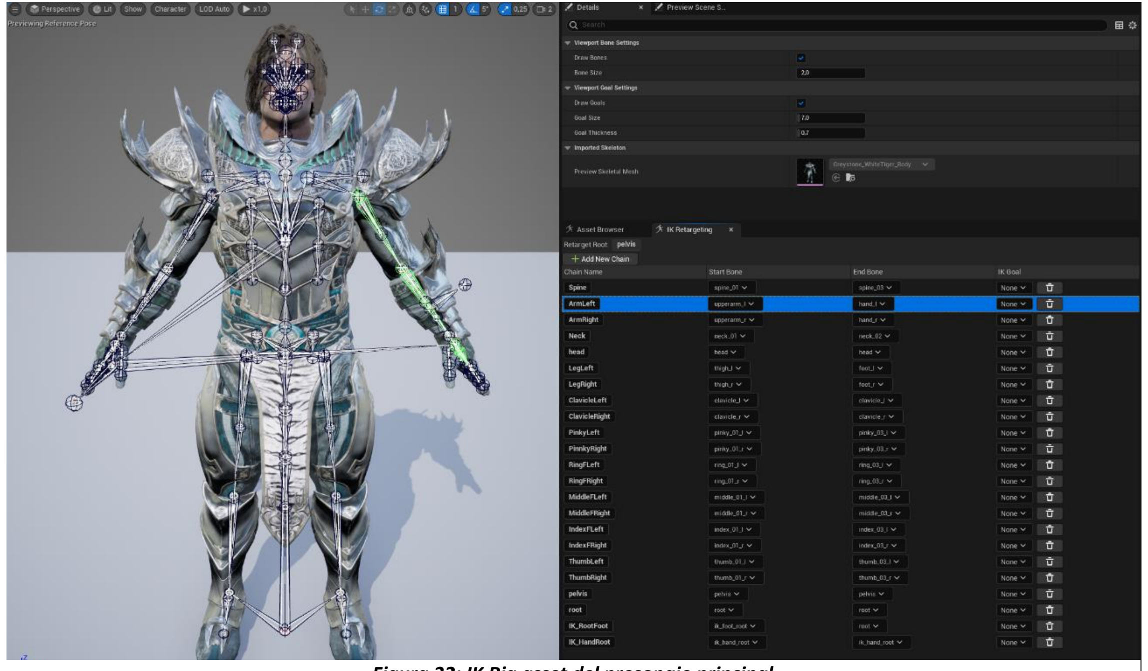


Figura 32: IK Rig asset del presonaje principal
(Fuente propia)

9.2.2. IK Retargeter

El *IK Retargeter* es el *asset* encargado de **transferir** la **información** de las **animaciones** asociadas a una malla esquelética a otra haciendo uso de los *IK Rig assets* definidos previamente [86]. Este *asset* se crea a partir de un *IK Rig*, el cual se toma como base para la transferencia de animaciones. Posteriormente, se selecciona el *IK Rig* objetivo (*Target*) para el cual se van a exportar las nuevas animaciones. Mediante un algoritmo de búsqueda de palabras en función de caracteres y longitud, el *IK Retargeter* relaciona automáticamente las cadenas de huesos de cada *IK Rig* (si tienen el mismo nombre) como muestra la **Figura 33**.

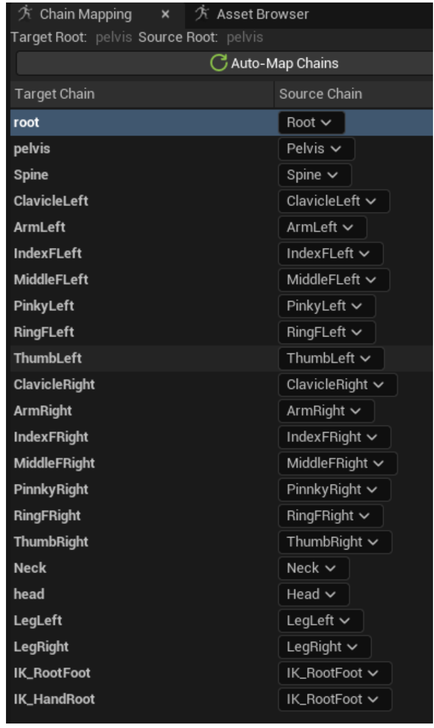


Figura 33: Relación utomática de cadenas de huesos entre dos IK Rig assets
(Fuente propia)

Una vez relacionadas correctamente las cadenas de huesos, el usuario/a debe **ajustar** la **pose inicial** del **modelo objetivo** ya que tiene que estar de la forma más similar al modelo base para que la transferencia de datos sea adecuada, como muestra la **Figura 35**. Si no se hace este paso puede dar lugar a transformaciones incorrectas y deformaciones inesperadas en la malla objetivo.

Figura 35: Malla base y malla objetivo en la pose base adecuada.
(Fuente propia)

El último paso es seleccionar las animaciones que se quieren transferir de la malla base a la objetivo, previsualizarlas en el editor para comprobar que no haya ningún error o deformación (**Figura 34**) y pulsar el botón de exportar.

Figura 34: Previsualización en tiempo real de una animación transferida a otro modelo.
(Fuente propia)

Las animaciones han sido obtenidas del portal *Mixamo* y de diferentes packs del Bazar de *Unreal Engine*: ***Bossy Enemy Animation Pack*** y ***Rolls and Dodges Animation Set***. Todas ellas fueron importadas al proyecto con el esqueleto base del maniquí de Unreal Engine 4 y "*retargeteadas*" al personaje *Greystone*, modelo del juego *Paragon*, disponible en el Bazar.

9.3. Mecánicas de movimiento

Una vez obtenidas todas las animaciones necesarias para el movimiento del personaje el autor procede a darle vida mediante la programación del movimiento.

9.3.1. Tipos de animaciones del personaje

El primer paso es definir cómo se va a mover el personaje y qué tipo de animaciones se van a emplear. Existen dos posibilidades: mediante animaciones estáticas o animaciones con desplazamiento.

Para decidir qué tipo es el adecuado, es necesario saber que todos los esqueletos suelen tener un hueso raíz o ***root*** del cual dependen el resto de los huesos y que tiene como origen el punto (0,0,0) del eje de coordenadas local del personaje.

9.3.1.1. Animaciones InPlace

Normalmente, la animación de personajes en videojuegos se realiza con animaciones estáticas, denominadas *InPlace*. En ellas, aunque el esqueleto se mueva reproduciendo la animación de caminar, correr o saltar el hueso raíz se mantiene estático en el origen, con lo cual el personaje se mantiene fijo en el sitio (como si caminara en una cinta de correr).

Esta aproximación no parece lógica si la comparamos con el mundo real, pero en el mundo de los videojuegos permite a los programadores/as dirigir el movimiento de los personajes mediante sus propiedades físicas: velocidad, aceleración, fuerza de salto, etc. Esto permite a los desarrolladores/as definir cómo de rápido o lento se va a desplazar un personaje y en función de ello decidir cómo se van a reproducir las animaciones consiguiendo así la ilusión de movimiento.

Este tipo de animaciones son útiles en videojuegos frenéticos donde se necesita una rápida respuesta a los controles que el usuario/a está pulsando y donde el movimiento del personaje debe ser definido por los programadores/as mediante sus propiedades físicas.

Esta técnica, pese a ser muy extendida en la industria del videojuego no es la elegida para el proyecto ya que el autor busca acercarse más a la realidad y el nivel de frenetismo no es el mismo que se puede encontrar en juegos como *League of Legends* o *Apex Legends*, ambos videojuegos multijugador online.

9.3.1.2. Animaciones Root Motion

Las animaciones con ***Root Motion*** son aquellas que llevan el desplazamiento del personaje intrínseco en la animación [87]. En este caso, el hueso raíz es desplazado durante la animación,

con lo que el esqueleto también se desplaza y a su vez la malla con él. De este modo, el movimiento del *root* se debe aplicar a la cápsula de colisión que envuelve al personaje y quitar esa responsabilidad del personaje. Es decir, el personaje se desplaza conforme lo hace la animación y no conforme a sus propiedades físicas.

Este tipo de animaciones aportan más realismo al juego y son de utilidad para el autor ya que, para los ataques y demás mecánicas, las propias animaciones desplazan al personaje tal cual lo ha querido el animador, ofreciendo un movimiento realista de forma simple y directa. Es por ello por lo que todas las animaciones que se emplean en el juego son con *Root Motion*. Esta decisión influye de manera directa en cómo se debe programar el movimiento del personaje.

9.3.2. Creación del *Blueprint* del personaje principal

Las características del proyecto y la restricción de desarrollar el trabajo en un tiempo limitado llevan al autor a iniciar la implementación utilizando *Blueprints* y dejando C++ para más adelante. El primer *blueprint* que crea es el del personaje controlado por el jugador/a: ***BP_PlayerCharacter*** (representado en la **Figura 36**) el cual hereda de la clase *PlayerCharacter* implementada en C++.

Este *blueprint* ya es un actor que se puede situar en el mundo de juego y que contiene una serie de componentes heredados de la clase *PlayerCharacter* y sus padres:

- **Cápsula de colisión**: Componente con forma de píldora que envuelve a todo el personaje y es empleado para los cálculos de colisiones. Es el componente raíz o ***root***.
- **Cámara de seguimiento**: Componente de renderizado situado detrás del personaje. Esta cámara sirve como campo de visión para el jugador/a y seguirá al personaje cuando se mueva.
- **Malla esquelética**: Representación gráfica del personaje dentro del mundo. El modelo con el esqueleto necesario para reproducir las animaciones y con la malla para que sea renderizado por pantalla.
- **Flecha de dirección**: Componente meramente informativo para el desarrollador/a para saber la orientación del personaje en el editor. No se renderiza en el juego.
- **Componente de movimiento**: Componente con variables y métodos relacionados con el movimiento de la entidad. El movimiento se basa en actualizar la posición del componente raíz en función de los datos y la lógica de este componente.

Además, este *blueprint* también hereda una serie de eventos relacionados con el movimiento y la actualización de datos dentro del juego:

- ***Event Tick:*** Evento llamado en cada fotograma de juego. Todo lo que se quiera actualizar cada fotograma debería llamarse desde aquí. Sin embargo, para evitar problemas de rendimiento debería evitarse hacer muchas llamadas a otros métodos desde aquí y bajar el *framerate* (cantidad de veces que se llama por fotograma) para ajustarlo a las necesidades estrictas del juego.
- ***Event Begin Play:*** Método llamado cuando el juego empieza o el personaje es creado en el mundo del juego. Empleado normalmente para inicializar valores de variables o referencias a otros objetos.
- ***MoveForward:*** Método que añade movimiento hacia delante o atrás al personaje en función de valor de *input* recibido desde el teclado/mando.
- ***MoveRight:*** Método que añade movimiento hacia derecha o izquierda al personaje en función de valor de *input* recibido desde el teclado/mando.
- ***TurnAtRate:*** Método que rota la cámara a izquierda o derecha en función de valor de *input* recibido desde el ratón/mando.
- ***LookUpRate:*** Método que rota la cámara hacia arriba o abajo en función de valor de *input* recibido desde el ratón/mando.
- ***SetupPlayerInput:*** Método encargado de relacionar las teclas/botones del mando con los métodos a los que tiene que llamar cuando se pulsen.
    - Esta forma de relacionar *inputs* con métodos no es la recomendada ya que no es sencilla de acceder desde otros sistemas y no es personalizable por el usuario. La forma recomendada es modificando los ajustes globales del proyecto y asignando la relación de teclas y eventos desde ahí.

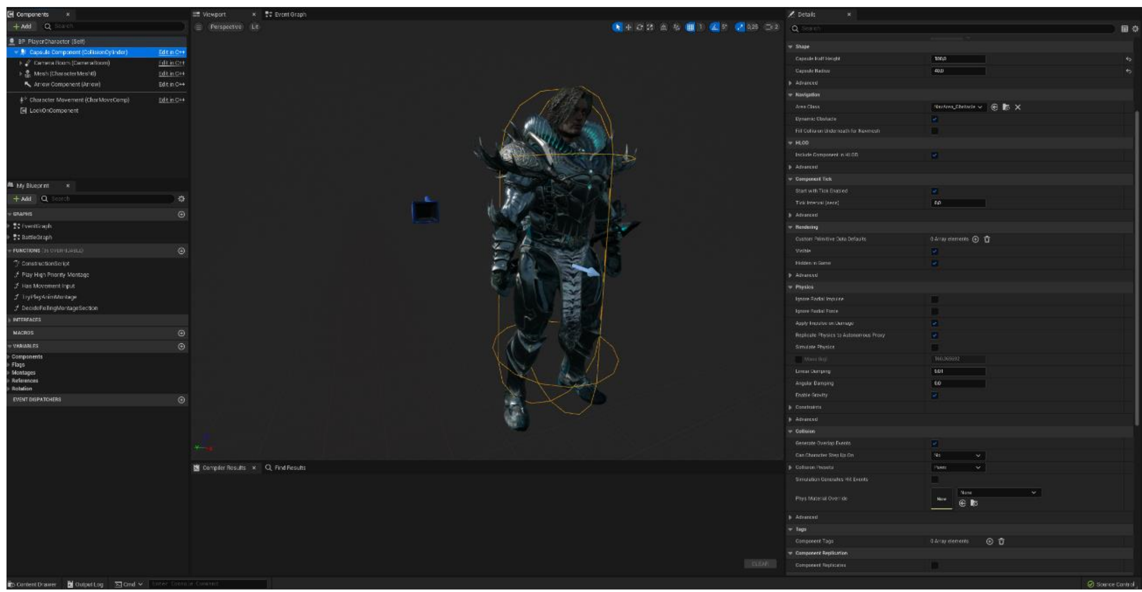


Figura 36: Blueprint BP_PlayerCharacter previsualizado en el viewport del editor de Unreal Engine
(Fuente propia)

9.3.2.1. Selección de la malla del personaje principal

El autor considera interesante incluir en este apartado la reflexión que hace para seleccionar la malla para el personaje principal. El autor se decanta por emplear el personaje *Greystone* ya que es el modelo que mejor se adapta a las características del proyecto, tanto en lo relacionado a armas y escudos disponibles como por la estética. Sin embargo, considera que si hubiera un personaje femenino disponible con el mismo rango de armas y escudos y estética adecuadas para el proyecto podría ser perfectamente el modelo seleccionado como personaje principal.

9.3.3. Creación del *Animation Blueprint* del personaje principal

Si ahora se situara el personaje recién creado en el mundo del juego, se podría mover y saltar sin problemas, pero lo haría sin reproducir ninguna animación y en pose T (pose tradicionalmente empleada cuando un modelo no tiene un esqueleto animado asignado).

Es por ello que se necesita crear un ***Animation Blueprint*** que gestione todo lo relacionado con las animaciones y que sea asignado a la malla esquelética del personaje. Este *animation blueprint* es el ***ABP_PlayerCharacter*** y hereda de la clase *AnimInstance* implementada en C++.

Dado que las animaciones empleadas por el autor en el proyecto son con *Root Motion*, hay que indicárselo al *Anim Blueprint*, activando la opción ***'Enable Root Motion from Everything'*** para que cuando este *blueprint* trabaje con las animaciones correspondientes, sepa cómo interpretar el movimiento de la malla esquelética correctamente.

El siguiente paso es asignar una máquina de estados a la *Output Pose* del *AnimGraph* como se ha mostrado en la **Figura 30** en la explicación de este tipo de *blueprint*. Por último, se debe asignar este *Anim Blueprint* a la malla esquelética en el *blueprint* del personaje como ilustra la **Figura 37**.

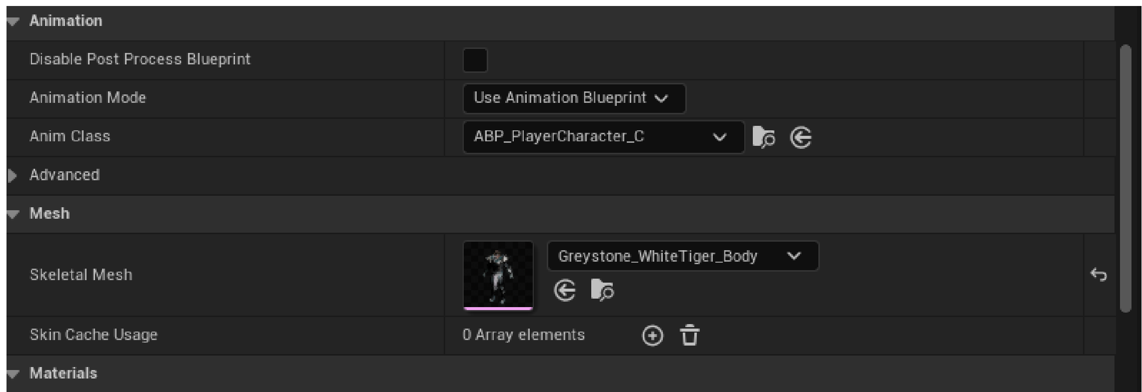


Figura 37: Asignando el AnimBP al componente de malla esquelética del BP del personaje
(Fuente propia)

9.3.4. *Blend Space* de movimiento

El movimiento del personaje del principal se ha diseñado en base a los videojuegos mencionados en el apartado **4.4. Juegos referentes**, de modo que este tiene dos tipos de movimiento: uno libre y otro radial cuando se fija a un enemigo.

Para ello, el primer paso es crear un ***Blend Space*** de animaciones. Un *Blend Space* es un *asset* de *Unreal Engine* el cual consiste en un grafo con dos ejes en el que se colocan diferentes secuencias de animaciones que mediante la técnica de ***Blending***[8] y 2 valores de entrada permite transicionar entre ellas. En este caso los dos valores de entrada son la cantidad de *input* recibida para caminar hacia delante (***ForwardInput***) y hacia la derecha (***RightInput***).

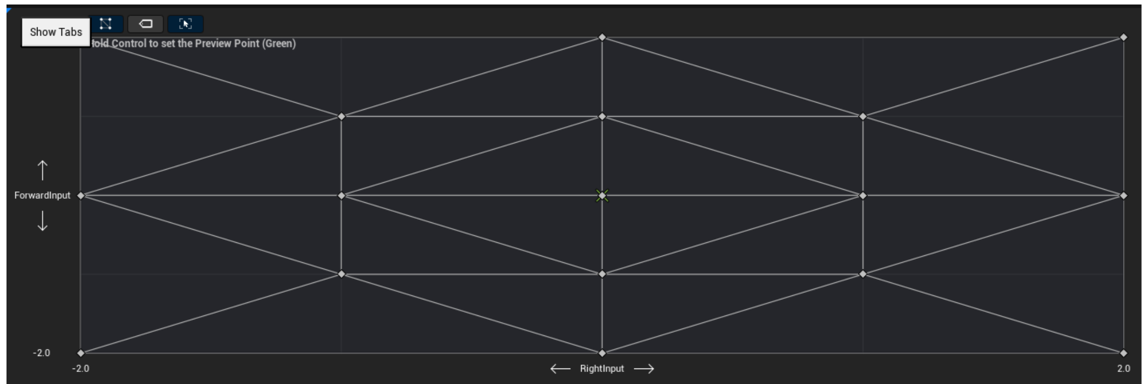


Figura 38: Blend space de movimiento del personaje principal
(Fuente propia)

Cada punto del grafo de la **Figura 38** es una secuencia de animación. El punto central (0,0) es la animación estática *idle*, pose donde el personaje está quieto. Las animaciones de la parte superior del grafo (X, 1 ó 2) son aquellas correspondientes al movimiento hacia delante. Las que están el parte inferior (X, -1 ó -2) corresponden al movimiento hacia atrás. De la misma manera, las situadas a la derecha corresponden al movimiento a la derecha (1 ó 2, Y) y las de la izquierda al movimiento hacia izquierda (-1 ó 2, Y).

De esta manera, cuando el jugador/a mueve el personaje hacia adelante, por ejemplo, se recogen los valores (0,1), realizando una transición desde la animación base en la que se encuentre hacia la de caminar para adelante.

Este *blend space* contiene las animaciones tanto para el movimiento libre como para el radial, ya que en función de la lógica de movimiento que esté activa en ese momento, se recogerán unos valores u otros de *input* para cambiar de una animación a otra y, por consiguiente, mover

[8] ***Blending***: Técnica de animación basada en la interpolación, lineal o cúbica, entre dos animaciones a lo largo de un periodo de tiempo. Para cada articulación relevante del esqueleto en ambas animaciones se interpolará la traslación de estas desde la animación base hasta la objetivo.

al personaje de una manera u otra. Dicha lógica de movimiento se explica en los siguientes apartados.

Para que estas animaciones se reproduzcan durante el juego, el *blend space* debe estar dentro de un estado de la máquina de estados conectada a la *OutputPose* del *AnimGraph* en el *blueprint ABP_PlayerCharacter*. La **Figura 39** muestra cómo se emplea un *blend space* dentro de un estado.

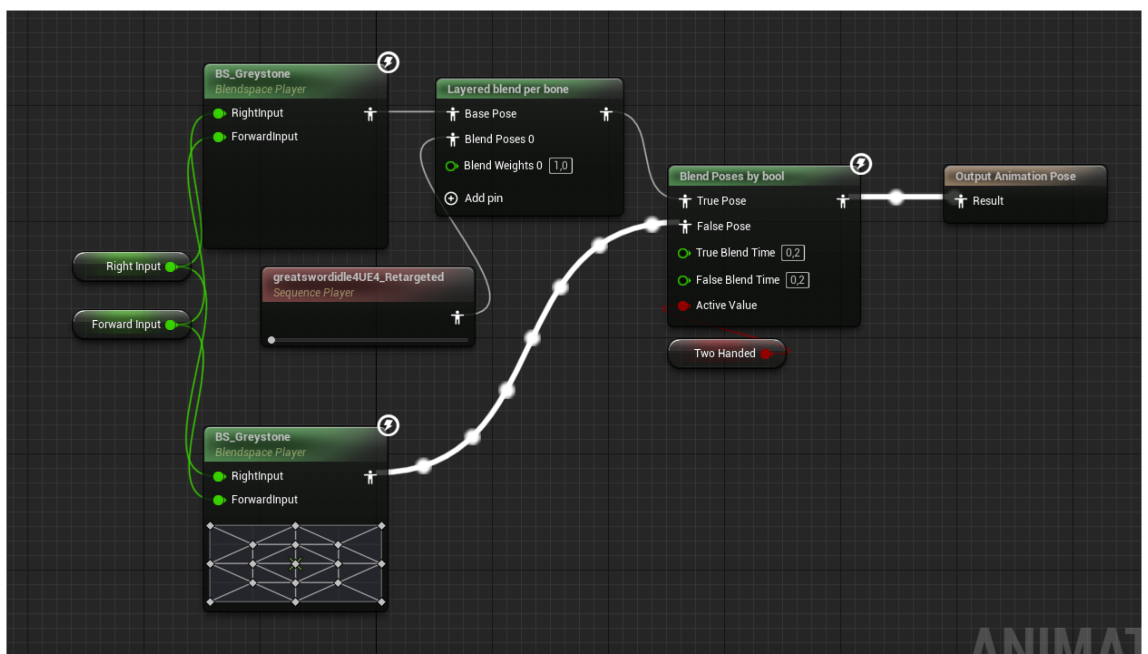


Figura 39: Blend space BS_Greystone empleado en el estado Idle/Walk de la máquina de estados
(Fuente propia)

9.3.5. Movimiento libre

Lo primero a implementar del personaje principal es el movimiento libre. Para ello el autor crea la lógica de movimiento en el Event Graph del ***BP_PlayerCharacter***. En él se calcula el vector director normalizado del ***Controller*** que posee al personaje en función de los valores de input que se reciben y la rotación del *controller* en ese momento. Una vez conocido hacia dónde apunta dicho vector después de recibir el input, se debe interpolar la rotación del personaje para que apunte en la misma dirección.

Con estos cálculos, si la cámara no se mueve el personaje se moverá en todas direcciones y para girar rotará sobre sí mismo pero la cámara no rotará con él. Por otro lado, dado que el *controller* y la cámara están asociados, en el momento en el que la cámara rota, lo hace también el *controller*. Esto es, por ejemplo, si la cámara se gira hacia la derecha o la izquierda y el personaje sigue caminando hacia delante, éste se irá girando tal y como lo está haciendo la cámara. Con ello se consigue dirigir la dirección de movimiento a partir de la cámara también.

Por otro lado, en el *AnimBP* se deben calcular los valores ***ForwardInput*** y ***RightInput*** necesarios para el *blend space* de movimiento. Para ello se debe recoger el valor del vector director del *controller* y los vectores *Forward* y *Right* del personaje relativos al espacio del mundo de juego (*World space*). Una vez obtenidos esos valores se debe hacer el **producto escalar** entre el **vector director** con el ***ForwardVector*** y el ***RightVector***.

En este caso, el resultado siempre va a ser que el *ForwardInput* es 0 (*Idle*) 1(*Walking*) o 2 (*Running*) y el *RightInput* es 0, de manera que en este modo de movimiento la animación siempre va a ser la de caminar hacia delante.

9.3.6. Movimiento radial o fijo (*Lock-on*)

La siguiente mecánica es el ***Lock-on enemy***, el jugador/a puede fijar a un enemigo para que lo tenga como objetivo y que la cámara apunte a él mientras está fijado. En ese momento el personaje comenzará a desplazarse de forma radial en torno al enemigo y mirando hacia éste.

En este caso, cuando el jugador/a pulsa el botón de fijar al enemigo se traza un rayo en la dirección del *Forward Vector* del *controller* (dirección en la que mira la cámara) con un radio determinado. Si dentro de ese rango se detecta a un enemigo, el *controller* se rota automáticamente para quedar enfocado hacia el enemigo. A continuación, el enemigo queda marcado como *target* y el personaje se orienta hacia él interpolando entre su propia rotación y la que tiene el *controller*, quedando también el personaje orientado hacia el enemigo. A la hora de desplazarse, se va actualizando la rotación del *controller* respecto al enemigo en función de la posición del personaje principal. Todo este proceso se muestra resumido en la **Figura 40**.

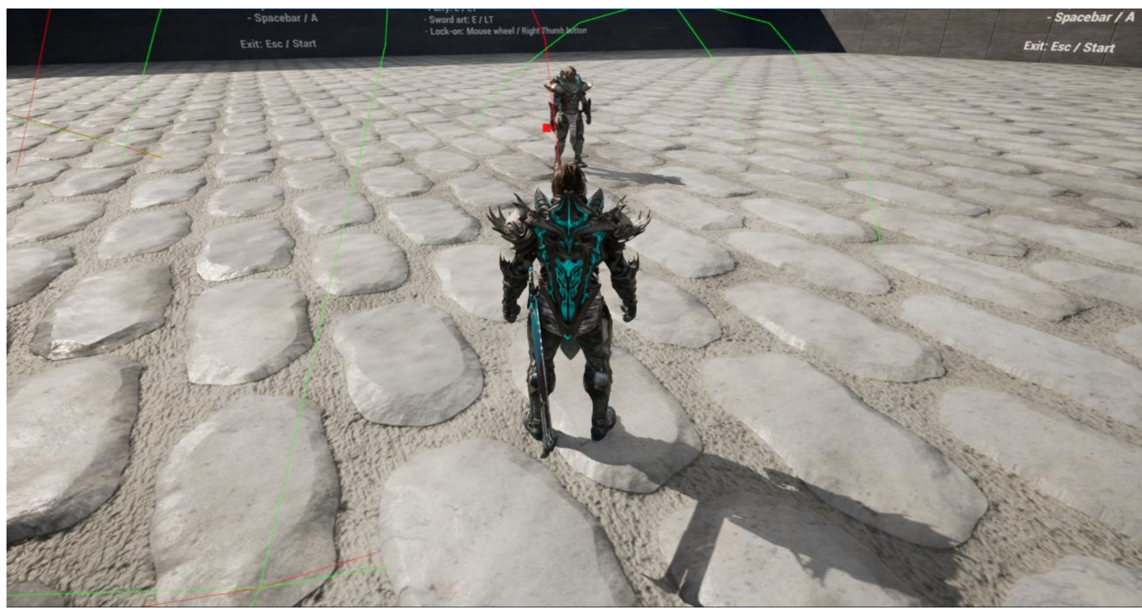


Figura 40: Personaje principal orientado hacia el enemigo fijado
(Fuente propia)

En esta mecánica el autor introduce por primera vez un componente propio con el ***BPC_LockOnComponent***, el cual contiene toda la lógica descrita anteriormente. Si se quitase dicho componente el personaje dejaría de poder fijar a los enemigos.

En este caso, dada la naturaleza de los cálculos de rotación, el resultado de la operación para determinar el *ForwardInput* y el *RightInput* (parámetros del *blend space* de movimiento) varía. Ahora se pueden dar como resultado todas las posibles animaciones que componen el *blend space*, haciendo así que el personaje se desplace reproduciendo las animaciones de desplazamiento frontal, hacia atrás, lateral y en diagonal.

Además, esta orientación se hace de forma automática y se actualiza en el ***Tick Event*** del componente, con lo que mientras que el jugador/a tenga fijado a un enemigo siempre se va a actualizar dicha rotación, incluso cuando quiere golpearle o se está reproduciendo algún combo de ataque.

Sin embargo, simulando el movimiento de los juegos de referencia, cuando el jugador/a decide correr teniendo al enemigo fijado, la cámara se mantiene enfocada en el enemigo y rota respecto a él, pero el personaje principal vuelve a poder moverse de forma libre.

9.3.7. Rodar y esquivar (*Roll and Dodge*)

Otra de las mecánicas referentes al movimiento del personaje es esquivar los ataques del enemigo. Para ello, el jugador/a puede esquivarlos dando un paso hacia atrás o rodando. Si no se recibe ninguna dirección de movimiento el personaje reproduce la animación de esquiva hacia atrás, mientras que si se detecta alguna dirección de movimiento se reproduce la animación de rodar.

Mientras el movimiento del personaje es libre, cuando se rueda, siempre se hace con la animación de rodar hacia delante, pero rotando al personaje en la dirección correspondiente. Sin embargo, cuando el movimiento es radial, se debe determinar qué animación de rodar es la adecuada. Una posible implementación es haciendo uso de *blend spaces* y con otro estado dentro de la máquina de estados. Pero esta vez el autor decide emplear otros artefactos disponibles en *Unreal Engine*: los ***Animation Montages***.

9.3.7.1. *Animation Montages*

Un *Animation Montage* es un *asset* el cual ofrece una forma de controlar las animaciones directamente desde *blueprints* o código en C++. En un *Animation Montage* se pueden combinar diferentes secuencias de animaciones (varias animaciones de rodar, una para cada dirección, por ejemplo) y tratarlo como un único *asset*. Además, se pueden partir en secciones para que

se reproduzcan como animaciones individuales o combinarlas. Por otro lado, también pueden disparar eventos como sonidos o efectos especiales en determinado momento de una animación.

Dada la flexibilidad de esta herramienta, el autor decide crear un *Animation Montage* que contiene todas las animaciones de rodar y una función para determinar en qué dirección está rodando. Esto permite al autor reproducir la animación correcta en cada caso desde el *blueprint*, sin necesidad de crear nuevos estados en el *AnimGraph* y sin complicar la lógica de animaciones.

De forma adicional, desde el *Animation Montage* se pueden editar las secuencias de animaciones individualmente, así como ajustar la función y tiempo de *blending* entre animaciones.

La **Figura 41** muestra cómo es un *Animation Montage* con el fin de ayudar al lector a visualizar todos los conceptos descritos en los párrafos anteriores.

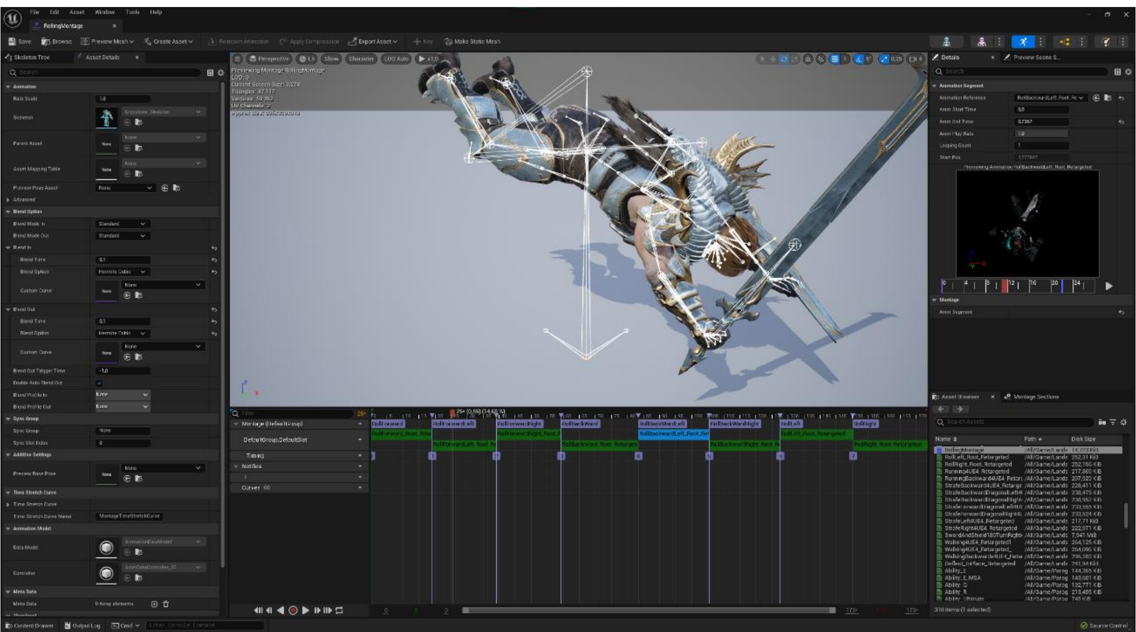


Figura 41: Animation Montage con todas las animaciones de rodar
(Fuente propia)

9.3.8. Salto

La última mecánica de movimiento del personaje principal es el salto. Con ella el personaje puede saltar en estático o en movimiento. Para este caso, el autor hace una excepción y emplea animaciones ***InPlace*** en combinación con máquinas de estado. Esta decisión es debida a que manejar el personaje en el aire aporta dinamismo al movimiento y al combate, factor que se puede echar en falta en alguno de los títulos de referencia.

Para ello, se crea una máquina de estados nueva únicamente para controlar los estados cuando el personaje esté en el aire. Así, el autor puede controlar las transiciones entre la máquina de estados cuando el personaje está en tierra y cuando está saltando, haciendo uso de ***Cached Poses*** y ***State Alias*** [88] [89].

La **Figura 31** ilustra dicha máquina de estados y cómo se comunica con el estado ***Locomotion*** (máquina de estados en tierra).

9.4. Mecánicas de combate

Una vez el personaje es capaz de moverse por la escena es momento de implementar las mecánicas de combate. Dadas las especificaciones del proyecto, el jugador/a debe poder atacar y defenderse.

9.4.1. Mecánicas de ataque

En este caso, el autor decide implementar primero las mecánicas de ataque ya que sin ellas el jugador/a no podría cumplir con el objetivo del juego: derrotar al jefe final. Sin ellas el juego no sería más que un simulador de caminar en un mundo fantástico.

9.4.1.1. Envainar y desenvainar el arma

El arma equipada del personaje se encuentra asociada a un ***Socket*** del esqueleto del personaje [90], concretamente en la cintura, simulando que el arma está envainada. Para poder atacar, el jugador/a debe desenvainar el arma. Para ello, el autor emplea un *Animation Montage* para reproducir la animación de desenvainar y a la vez modifica el ***socket*** al cual está asociada el arma, siendo ahora un ***socket*** situado en la mano derecha.

El hecho de utilizar *Animation Montages* ofrece al autor ciertas ventajas para este caso:

- **Conocer si se está reproduciendo ya una animación**

  Es necesario controlar que el usuario/a no pueda interrumpir la reproducción de una animación y volver a empezarla indefinidamente pulsando repetidas veces el mismo botón. Por ello, si se conoce si se está reproduciendo una animación, el autor puede dejar sin efecto el evento asociado a pulsar ese botón hasta que finalice la animación.

- **Disparar un evento en un momento determinado de la animación**

  Para cambiar la posición del arma (pasar de la cintura a la mano) se debe hacer en un momento de la animación en el que quede natural hacerlo. Puede ser que en el inicio o

en el final no quede realista y se deba disparar en mitad de la animación mediante un ***Anim Notify*** [91].

- **Controlar el flujo de la lógica de juego**

El nodo del *blueprint* que contiene el *Animation Montage* se ejecuta instantáneamente, con lo que termina mucho antes de que la animación se complete. En ocasiones, el autor desea que el flujo de ejecución no prosiga hasta que dicha animación haya terminado o que prosiga en determinado momento de la animación (como ilustra la **Figura 42**) con el fin de evitar situaciones ilógicas en el juego. Un ejemplo sería que el jugador/a pudiera atacar cuando el arma aún no está en su mano.

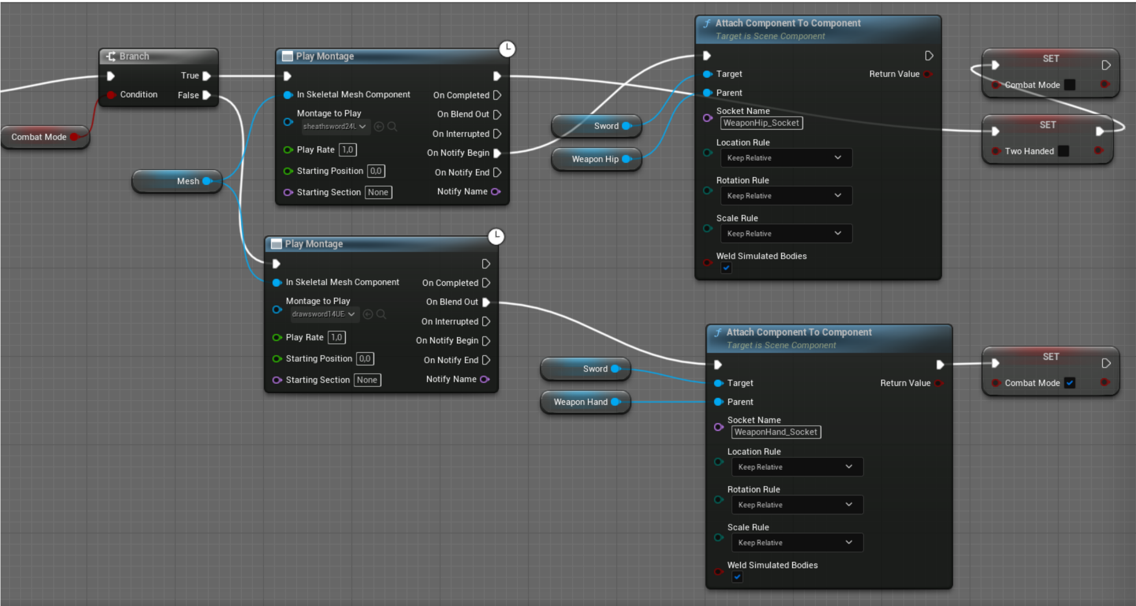


Figura 42: Uso de Anim Montages seguido del cambio de socket para modificar la posición de la espada.
(Fuente propia)

Asimismo, como es lógico, el mismo botón que se emplea para desenvainar el arma se emplea para envainarla. Mediante el uso de un booleano a modo de estado se controla si se debe reproducir una animación u otra.

9.4.1.2. Ataque ligero y fuerte (a una y dos manos)

Una vez el arma está operativa, el personaje debe poder atacar. Dadas las especificaciones del proyecto, puede hacerlo con un ataque ligero o con uno fuerte. El ataque ligero consume menos energía del personaje y ejerce menos daño. Por otro lado, el ataque fuerte ejerce más daño a costa de un mayor consumo de energía.

Además, puede atacar con el arma a una mano o a dos manos. Dado que el sistema empleado y la lógica es la misma se procede a explicar la implementación global de todos los tipos de ataque en este apartado.

El sistema de ataque se implementa haciendo uso de:

- ***Blueprint BP_PlayerCharacter***: para recoger los eventos de *input* del jugador/a e implementar la lógica asociada al daño y la resistencia.
- ***Animation Blueprint ABP_PlayerCharacter*** para implementar la lógica y flujo de reproducción de las animaciones.
- ***Animation Montages:*** para contener todas las animaciones en un solo *asset* y facilitar la implementación del sistema de ataques y combinaciones.
- ***Interfaz BPI_Character:*** para poder hacer desde el BP_PlayerCharacter uso de un evento implementado en ABP_PlayerCharacter (*Play Combo Attack Montage*).
- ***Enum E_AttackTypes:*** para saber si se deben reproducir animaciones de ataque ligero o fuerte.

Desde el *blueprint* del personaje del jugador/a se recibe el input de ataque. En función de cuál haya sido se llama al evento *Play Combo Attack Montage* pasando como parámetro uno de los elementos del *enum E_AttackTypes* (*Light o Heavy*). El *animation blueprint* que tiene la implementación del evento recibe esa llamada y reproduce la animación adecuada siempre que no se esté reproduciendo ya una animación.

9.4.1.3. Combinaciones de ataques

El sistema descrito, presenta un grado de escalabilidad y de personalización bastante altos y permite al autor implementar combinaciones de ataques con relativa sencillez. Para ello, es necesario añadir un elemento más: el ***Anim Notify State ANS_AttackCombo*** y las variables ***NextLightCombo*** y ***NextHeavyCombo.***

El *Anim Notify State* permite al autor introducir un *Anim Notify* personalizado dentro del *Animation Montage* para indicar qué segmento del *Animation Montage* se debe reproducir después del actual si se pulsa de nuevo el botón de atacar mientras se está reproduciendo la animación de ataque. Mediante la concatenación de animaciones individuales el autor consigue crear una combinación de ataques coherente y personalizada.

Para llevar esto a cabo, se divide el *Animation Montage* en diferentes secciones y se asigna un nombre único a cada una. Dentro de cada sección se añade una instancia del *Anim Notify State* indicando qué sección de ataque es la siguiente, creando así la concatenación de animaciones. Posteriormente, en el *Animation Blueprint* se recogen los nombres de las secciones que se deben reproducir y se pasan como parámetro al *Animation Montage*.

Figura 43: Animation Montage dividido en secciones y con Anim Notify States
(Fuente propia)

La **Figura 43** muestra al lector cómo se divide el *Animation Montage* en diferentes secciones (una para cada ataque) y cómo se utilizan los *Anim Notify States* descritos (representados por las etiquetas lila de la fila inferior).

9.4.1.4. Ataque en el aire y ataque especial

Una de las mecánicas que aporta cierto dinamismo al juego es el ataque en salto. En este caso el autor hace uso de un sistema similar al descrito anteriormente pero donde no se concatenan animaciones para hacer combinaciones, sino que simplemente se selecciona qué *Animation Montage* reproducir en función del elemento del *Enum **E_AttackTypes*** que reciba el evento ***Play Special Attack*** como parámetro (*Jump, Special o Parry*).

Finalmente, la última mecánica de ataque es el ataque especial *o **weapon art**.* Este ataque suele ejercer más daño y consumir maná además de energía. Este ataque se puede ejecutar solo cuando el arma está a dos manos o cuando el arma está a una mano y el personaje no lleva el escudo en el brazo.

9.4.2. Mecánicas de defensa

Con tal de evitar recibir daño, el jugador/a puede cubrirse haciendo uso del escudo. Para ello, igual que ocurre con el arma, el escudo debe ser colocado en el brazo para su uso en el juego. Dado que la implementación es la misma no se explica de nuevo en este apartado. Asimismo, se puede cubrir también con el arma cuando la tiene a dos manos.

9.4.2.1. Defensa con escudo

Una vez el personaje tiene el escudo asociado al *socket* del brazo, el jugador/a puede cubrirse pulsando el botón correspondiente. En este caso, si se mantiene pulsado el botón, el estado del personaje se mantiene. Aquí, el autor emplea una herramienta disponible en los *Animation Blueprints*: ***Layered blend per bone*** [92]. Este nodo permite al autor mezclar dos animaciones sobre un mismo modelo y concretar a qué huesos afecta específicamente la mezcla.

Se tiene una animación base (caminar normal) y una animación que mezclar (*idle* de cubrir con escudo). El autor conecta ambas animaciones al nodo, especificando un porcentaje de mezcla e indicando qué hueso es el afectado. En este caso, interesa mantener el tren inferior intacto para seguir reproduciendo las animaciones de caminar correctamente con lo que el hueso afectado debe ser la espina dorsal. Dado que el resto de los huesos del tren superior dependen de éste, todos ellos se verán afectados, consiguiendo la pose de defensa con escudo al mismo tiempo que el personaje se desplaza de forma natural.

Otra herramienta relacionada con la mezcla de animaciones es el nodo *Blend poses by bool*. Este nodo permite cambiar entre animaciones haciendo uso de un booleano como *flag.* Una de las aplicaciones es cambiar entre la animación empleada para llevar el arma a una mano o dos. Otra de ellas es cambiar la animación de defensa entre la defensa con escudo o defensa con la espada, como muestra la **Figura 44**.

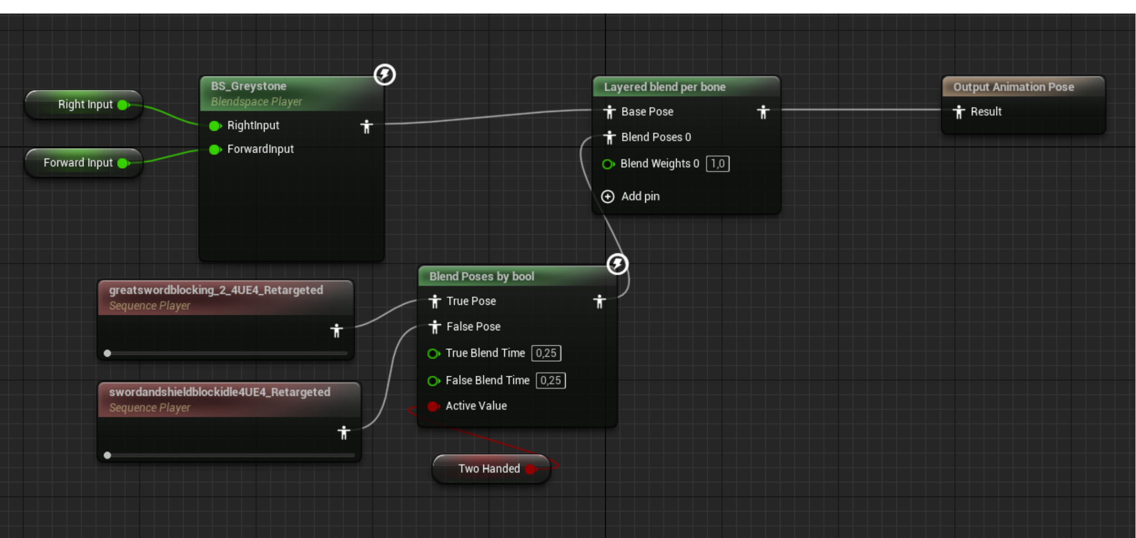


Figura 44: Uso del blend nodes en el Animaton Blueprint del personaje principal
(Fuente propia)

9.4.2.2. Defensa especial (Parry)

Una de las mecánicas clásicas de los *Soulslike* es el ***Parry***. Consiste en repeler el ataque del enemigo dejándolo indefenso. Para ello, el jugador/a debe pulsar el botón de ataque especial cuando tiene el escudo en el brazo y en el momento justo en el que se va a recibir el impacto del arma del enemigo. Esta mecánica está implementada con el mismo sistema que el ataque en salto y el ataque especial.

9.4.3. Refactorización y *Combat Component*

Conforme el autor continúa con el desarrollo del proyecto se siente más cómodo con el motor y comprende mejor cómo está estructurado. Es por ello por lo que decide refactorizar el código implementado hasta el momento y sacar más provecho a los componentes de *Unreal Engine*.

Conforme la lógica de combate crece en tamaño y complejidad a lo largo del desarrollo, no es la mejor decisión de diseño dejar toda esa lógica en el *blueprint* del personaje principal. Para corregir esta decisión inicial el autor crea el ***Combat Component***. Este componente contiene toda la lógica de combate descrita anteriormente en funciones que pueden ser invocadas desde el *blueprint* del personaje principal.

Asimismo, separar la lógica en componentes permite que cuando se implemente la Inteligencia Artificial el autor haga uso de dichos componentes y aproveche la lógica ya implementada para el personaje principal.

9.5. Actores interactivos

Una de las partes esenciales en los videojuegos ARPG, y especialmente en los *soulslike,* son los objetos interactivos: pociones, almas, armas, escudos, armaduras, recompensas, etc. Cada tipo de actor interactivo tiene un comportamiento diferente que se dispara en el momento en el que jugador/a interactúa con él. Esta situación es la óptima para crear una interfaz (***BPI_Interactable***) con el evento ***Interact*** donde cada tipo de actor le dé una implementación diferente.

9.5.1. Actores 'pickeables'

El primer tipo de objeto interactivo a desarrollar son aquellos que el jugador/a puede recoger. Dentro de este grupo se encuentran las pociones, almas, armas, escudos y partes de armadura.

Este tipo de actores se encuentran distribuidos por el mapa y solo se puede interactuar con ellos una vez (en cuanto el personaje los recoge, estos son destruidos). Para ello el autor implementa el *blueprint* ***BP_PickupActor*** compuesto por una malla estática y una esfera de colisión. Además, este *blueprint* implementa la interfaz *BPI_Interactable* para implementar un comportamiento propio del evento *Interact*.

Por otro lado, el autor debe implementar en el *blueprint BP_PlayerCharacter* el evento de interacción del personaje. Dicho evento consiste en buscar objetos de tipo ***Interactable*** en un radio alrededor del personaje. En el momento que la esfera de búsqueda del personaje colisiona con la esfera de colisión del ***BP_PickupActor*** se dispara el evento *Interact* de este último.

9.5.2. Actores equipables

Estos tipos de actores son aquellos que se adjuntan como actores hijo de otro actor. Estos actores no heredan del *blueprint BP_PickupActor* sino que son un tipo de actores independientes aunque conceptualmente están relacionados.

Cada *BP_PickupActor* contiene una referencia de clase de los diferentes tipos de objetos equipables, con lo que en el momento que se recoge un objeto *'pickeable'* este se destruye, pero se crea un nuevo actor de la clase equipable especificada en el objeto *'pickeable'*.

De este modo, el autor implementa la clase padre ***BP_BaseEquipable***, la cual implementa los eventos y funciones básicas necesarias para hacer a un objeto equipable. Las más relevantes son: ***OnEquip, OnUnequip, AttachActor*** y ***DettachActor***.

Además, contiene una malla esquelética y una malla estática. Contiene ambos tipos de malla ya que puede darse el caso de que un objeto equipado requiera moverse conforme el esqueleto del persoanje (un fragmento de armadura, por ejemplo).

Por otro lado, este tipo de actores se debe asociar a un ***socket*** del esqueleto del actor padre. Por defecto únicamente se guarda un nombre de *socket* en este objeto, el socket al que se va a asociar en el momento de su creación, pero conforme se baja en el nivel de jerarquía, cada actor almacena más de un nombre para diferentes *sockets* en función de sus necesidades.

9.5.2.1. Escudos y Armas

El siguiente nivel de la herencia lo implementan las clases ***BP_BaseWeapon*** y ***BP_BaseShield***. Estos *blueprints* son el puente entre un objeto equipable genérico y un arma o escudo específicos. Este nivel intermedio permite al autor guardar una referencia de una espada y escudo bases en el ***Combat Component***, facilitando así el cambio del arma o escudo específico en cualquier momento del desarrollo o del transcurso del juego sin necesidad de cambiar el tipo de referencia ni la lógica implementada.

Es en este nivel de la jerarquía donde se sobrescribe la lógica del método *OnEquip* de la clase padre. En él se implementa la lógica para asociar el arma o escudo al actor padre y se establece la referencia en el componente de combate.

El último nivel de herencia lo implementan las espadas y escudos específicos, entre ellos los *blueprints* ***BP_WhiteTigerSword*** and ***BP_WhiteTigerShield***.

9.5.2.2. Anim Notify AN_AttachWeapon/ShieldActor

Anteriormente, el autor asociaba una malla estática al personaje principal después de reproducir las animaciones de envainar y desenvainar. Eso era simplemente una implementación para mostrar de forma más directa cómo quedaban las animaciones de las mecánicas de combate. Con la creación de los nuevos actores equipables y del componente de combate, el autor aprovecha para utilizar los ya introducidos ***Animation Notifies***. En este caso, dichos *Anim Notifies* implementan la lógica de asociar/desasociar el actor arma/escudo al personaje en el

momento del *Animation Montage* que el autor quiere sin necesidad de llamar posteriormente a otro método.

9.6. Componente de colisión

Una vez creados los actores de espadas y escudos es momento de otorgarles funcionalidad. Su tarea principal es colisionar con la malla del enemigo para poder disparar los eventos relacionados al daño, la vida, la resistencia y el maná (descritos más adelante). Para tal fin, el autor implementa el ***BP_CollisionComponent***. Este componente se encarga de calcular si hay colisión entre el actor que contiene el componente y la malla del enemigo mientras se reproduce la animación de ataque.

Para facilitar la explicación del funcionamiento del componente y hacer más sencilla la compresión del apartado, se procede a utilizar sólo el arma como objeto de estudio. El primer paso es guardar una referencia de la malla del objeto que contiene el componente. Para ello, en cuanto el jugador/a recoge el arma se pasa una referencia de la malla al componente de colisión para que la guarde como variable miembro. Asimismo, la malla contiene dos *sockets* que sirven para delimitar el principio y fin de la cápsula de colisión que envuelve al arma. Esta cápsula no está activa continuamente, sino que se activa y desactiva en momentos concretos de la animación de ataque. Esto es posible gracias al *asset* ***ANS_HitCollisionDetection***.

9.6.1. ANS_HitCollisionDetection

Este *asset* es un *Anim Notify State* empleado para activar y desactivar la colisión del actor que contiene al componente. Este se coloca dentro del *Animation Montage* de ataque y se expande el número de fotogramas exactos en función de la duración del ataque. Esta forma de activar y desactivar el cálculo de colisiones es más limpia y configurable que haciéndolo desde el *blueprint* del personaje o desde el componente de combate, por ejemplo. La **Figura 45** muestra su uso en la animación de ataque especial, donde se concatenan tres ataques consecutivos.

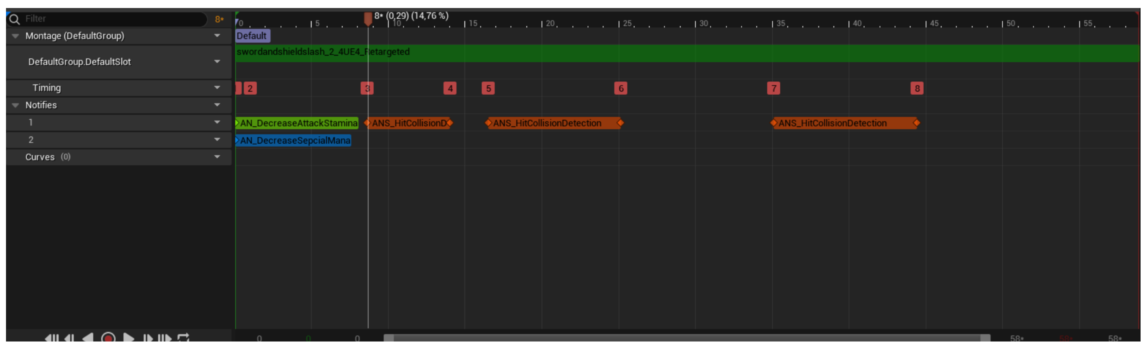


Figura 45: ANS marcados en rojo para activar y desactivar la colisión del arma
(Fuente propia)

Una vez activada la colisión es momento de realizar el cálculo en sí. Para ello se emplea el nodo ***Multi Sphere Trace For Objects*** que comprueba si hay colisión entre el objeto que invoca el método (el arma) y los tipos de objeto especificados como parámetro. Este método es ejecutado por el ***Event Tick*** del componente cada fotograma mientras la colisión está activa. Es por ello por lo que también se debe pasar como parámetro un array con referencias a los actores que ya han sido golpeados en ese ataque para que solo sean golpeados una vez y no repetidas veces. Este array se vacía en cuanto la colisión se desactiva (cuando acaba el ataque).

Para mejor comprensión, la **Figura 46** muestra las cápsulas de colisión y su resultado (en verde si colisiona, en rojo en caso contrario) en el momento en el que el personaje del jugador/a ataca.

Figura 46: Cálculo de colisiones en la animación de ataque
(Fuente propia)

Finalmente, para cada resultado satisfactorio del cálculo de colisiones se llama al evento ***OnHit*** implementado en el arma para que aplique el daño a los enemigos golpeados.

9.7. Mecánicas de daño, vida, resistencia y maná

Hasta ahora, el autor ha descrito las mecánicas de combate y movimiento, los diferentes objetos que se emplean y sus componentes, así como el cálculo de colisiones en los ataques. El siguiente paso lógico es dar sentido a dichas mecánicas con los atributos de daño, vida, resistencia y maná. Este apartado junto con los anteriores sienta las bases del juego y, una vez implementados sus puntos, da pie a comenzar con el desarrollo de la inteligencia artificial.

9.7.1. Estadísticas de las armas y escudos

Una de las características principales de los *soulslike* es que cada arma tiene unos atributos y estadísticas diferentes. Estas propiedades ofrecen un amplio abanico de armas y escudos que el jugador/a puede elegir para configurar su estilo de juego en la partida. Los principales atributos son:

- **Daño base del arma**
  - Es la cantidad de vida que se va a restar al enemigo si se le golpea.
- **Potenciadores de daño en función del ataque**
  - Para cada tipo de ataque hay un multiplicador que aumenta, o no, la cantidad de daño que ejerce el arma. Para ello el autor emplea los mapas como estructuras de datos donde cada tipo de ataque tiene asociado un potenciador.
- **Coste de resistencia (*stamina*) por ataque**
  - Para cada tipo de ataque se tiene un coste determinado de la resistencia del personaje. De la misma manera que los potenciadores de daño, se emplea un mapa para guardar un coste para cada tipo de ataque.
- **Coste de maná de los ataques especiales**
  - Cantidad fija de maná que se resta al personaje al ejecutar un ataque especial.
- **Cantidad de daño bloqueado**
  - Cantidad de daño que se resta al daño recibido si el bloqueo es satisfactorio.
- **Cantidad de resistencia consumida por bloquear**
  - Cantidad de resistencia que se resta al personaje si bloquea un ataque**.**

9.7.2. *Stats Component*

El componente ***BPC_StatsComponent*** es el encargado de almacenar y gestionar los datos de las estadísticas y atributos del personaje: **vida, resistencia y maná**. Dentro del componente se hace la distinción entre estadísticas base del personaje y estadísticas actuales. Las primeras sirven para inicializar a los personajes en el momento que son instanciados en el mapa y las segundas

son las que se modifican y actualizan durante la partida. Para almacenar las estadísticas iniciales se hace uso de un mapa donde la clave es el nombre del *stat* (vida, resistencia, maná) y los valores son el valor base y el valor máximo. De forma similar, para las estadísticas actuales se hace uso de un mapa con los mismos valores para las claves, pero con un único *float* para el valor de ese *stat*.

Se ha visto que todas estas estadísticas se decrementan, pero también es esencial que se regeneren para poder continuar jugando. Por un lado, la regeneración de la resistencia se hace de forma automática y progresivamente. Después de cada acción que consume resistencia se llama al método encargado de incrementar la estamina repetidamente hasta que el valor actual alcanza el máximo. Por otro lado, la vida y el maná se recuperan consumiendo pociones que incrementan una cantidad fija el valor actual.

Asimismo, estos datos son privados y solo son modificables a través de los métodos del componente. De este modo el autor se asegura que no se hacen modificaciones en partes del código que no se debe y tiene control en todo momento de estos datos.

9.7.2.1. Uso de stats en el Componente de combate

En los apartados anteriores ya se ha mencionado al ***Combat Component***. Es el encargado de almacenar la lógica de las acciones de combate y contener las referencias del arma y escudo principales del personaje. Hasta el momento dichas acciones no tenían en cuenta las estadísticas o el estado del personaje, pero gracias al *BPC_StatsComponent* el autor puede emplear esa información para implementar correctamente la lógica de juego. El uso principal de componente de *stats* dentro del componente de combate está relacionado con el uso de la **estamina**. Cada acción de combate consume estamina, con lo que para poder ejecutar una acción previamente se debe comprobar si el personaje tiene suficiente resistencia para poder ejecutar la acción, en caso contrario debe esperar hasta tener la cantidad suficiente. Lo mismo ocurre con la cantidad de maná para los ataques especiales.

Asimismo, el componente de combate cuenta con una función para decrementar la resistencia (***DecreaseStamina***) y otra para el maná (***DecreaseMana***) en función del ataque que se esté reproduciendo y del arma empleada. Esta función pasa como parámetro la cantidad calculada de estamina que se debe decrementar a la función ***ModifyCurrentStatValue*** del *StatsComponent*.

9.7.2.2. *Uso de Anim Notifies para decrementar stats*

Una de las problemáticas que acomete el autor es el decremento de la resistencia y del maná. Estas únicamente se deben decrementar si se reproduce la animación pertinente y no cada vez que se pulse el botón de atacar por ejemplo o rodar.

Como se ha explicado en apartados anteriores, el hecho de pulsar un botón y llamar al evento que reproduce una animación es casi instantáneo pero la reproducción de una animación no. Esto supone que una acción llamada a continuación del evento de reproducir la animación se ejecute, se reproduzca o no la animación.

Es por ello por lo que, para abordar esta problemática, el autor emplea ***Anim Notifies*** para acceder al *CombatComponent* y hacer uso de la función *DecreaseStamina* o *DecreaseMana.* De esta forma el autor se asegura de que se ejecuten esas funciones únicamente si se llega a reproducir la animación y en ningún otro caso.

9.7.3. Daño y muerte del personaje

Como se detalla al final del apartado 9.6. Componente de colisión, cada vez que un arma colisiona satisfactoriamente contra un enemigo, se llama el evento *OnHit* implementado en el *BP_Base_Weapon*. Este evento recibe la información de la colisión y la pasa como parámetro a la función ***Apply Damage***. En dicha función se comprueba si el actor golpeado puede recibir daño (no está muerto). En caso de cumplir la condición satisfactoriamente se invoca el evento ***Apply Point Damage*** disponible en *Unreal Engine*. A dicho nodo se le debe pasar como parámetros: el actor que ejerce el daño (el propietario del arma), la dirección de golpeo, la información de la colisión, quién ejerce el daño (el arma) y la cantidad total de daño. Este último parámetro se calcula en función de las características propias del arma y del tipo de ataque que se esté reproduciendo.

La llamada de este último evento se recoge en el *blueprint* del personaje afectado. Así pues, gracias a toda la información recibida se puede llamar a la función ***Take Damage*** implementada en el *BP_PlayerCharacter*. Después de comprobar que efectivamente el actor puede recibir daño se procede a ejecutar los eventos correspondientes a la reacción al golpe. Estos incluyen:

- **Reproducir la animación de recibir daño**
    - Haciendo uso del **producto escalar** entre el personaje que golpea y el que recibe el daño se determina si el golpe viene desde adelante o atrás, reproduciendo así una animación de reacción u otra.
- **Reproducir sonidos de golpeo**

  - En caso de que el golpeo sea efectivo y no se esté cubriendo el enemigo, se reproduce un sonido de daño. En caso contrario se reproduce un sonido de choque metálico.
  - Asimismo, se puede utilizar una *Sound Cue* compuesta por varias pistas de sonido para que cada vez que se golpee se reproduzca una pista aleatoria y no siempre la misma.
- **Reproducir efecto visual de golpe**
  - Al igual que con el sonido, si el golpeo es efectivo se reproduce un efecto visual (VFX) de sangrado. En caso contrario se emiten unas chispas de golpe metálico.

Después de efectuar estos eventos se llama a la función ***TakeDamage*** del ***BPC_StatsComponent*** pasando como parámetro el daño a ejercer. Esta función únicamente modifica el *stat* de vida del personaje afectado restando la cantidad recibida. El procedimiento para recibir o no daño en área es el mismo, pero invocando el evento ***ApplyRadialDamage.***

Asimismo, en el *blueprint* del personaje principal hay implementado un evento que es disparado cada vez que el *stat* de vida cambia su valor. En caso de que el valor llegue a 0 se procede a ejecutar el método ***PerformDeath***. Este en primer lugar, cambia el estado del personaje a **muerto,** incapacitando los movimientos y acciones. A continuación, deshabilita el componente de combate y el de *LockOn* para que el personaje no pueda ejecutar estas acciones después de muerto. En siguiente lugar, se desactiva la colisión de la cápsula que envuelve al personaje muerto. Una vez desactivado todo lo anterior se reproduce la animación de muerte. En caso de los enemigos simples la entidad queda inmóvil en el suelo en la posición final de la animación, sin embargo, la entidad del jefe final es destruida. Esta diferencia de comportamiento se justifica en el punto **9.17. Flujo de juego**. Asimismo, en ese punto también se detalla el comportamiento cuando el personaje principal es derrotado.

9.7.4. Representación gráfica de las estadísticas del personaje

Una parte primordial de los videojuegos es ofrecer una respuesta visual a la persona que juega de los eventos que están pasando dentro del juego. Por ello es necesario representar a través de la pantalla el estado de las estadísticas del personaje para que el jugador/a no se tenga que preguntar por qué su personaje ha dejado de atacar o se ha desplomado en el suelo de repente, por ejemplo.

Para representar la vida, la resistencia y el maná normalmente se suelen emplear barras de progreso. Estas barras están totalmente coloreadas cuando las estadísticas están al máximo y

van vaciándose a medida que desciende el valor de la estadística. De la misma manera representan también cuando el personaje se cura o recupera maná y resistencia.

Para implementar dicha funcionalidad en *Unreal Engine* es necesario emplear ***WidgetBlueprints*** [93]. Estos son *bluepritns* cuyo diseño está pensado para facilitar la creación de la interfaz de usuario y el **HUD** (*Head-Up Display*) del juego. El HUD habitualmente se refiere a la información que se muestra de manera visual durante toda la partida en forma de iconos, texto, número y, como en este caso, barras de progreso. El primer *WidgetBlueprint* que implementa el autor es el ***WB_MainHUD***. Este está compuesto por un ***canvas*** que a su vez contiene todos los elementos que se dibujan en la pantalla y que forman parte del HUD. En este punto del desarrollo los únicos elementos que se dibujan son las barras de progreso relacionadas con la vida, la resistencia y el maná. Para ello el autor implementa otro *WidgetBlueprint*, ***WB_StatBar,*** que únicamente contiene una barra de progreso.

Al tratarse de un *blueprint* se le puede asignar lógica y comportamiento como a cualquier otro actor del juego y le permite comunicarse con el resto de las entidades de este. Para poder realizar esta última tarea, el *WB_MainHUD* que contiene las instancias de los *WB_StatBar* debe relacionarse con el ***Controller*** que dirige al personaje del jugador/a. Una vez relacionado el HUD con el persoanje, se puede acceder al componente de *stats*, facilitando así la obtención de los valores de estos para poder ser representados en las barras de progreso. Asimismo, en el *blueprint WB_StatBar* se implementa un evento que es llamado cada vez que se actualiza el valor de un *stat* del personaje principal, haciendo así que la barra de progreso se actualice a la par que el valor que representa. La **Figura 47** muestra cómo se ven dichas barras de progreso en la pantalla de juego. En rojo se muestra la barra de vida, en azul la barra de maná y en verde la de resistencia.

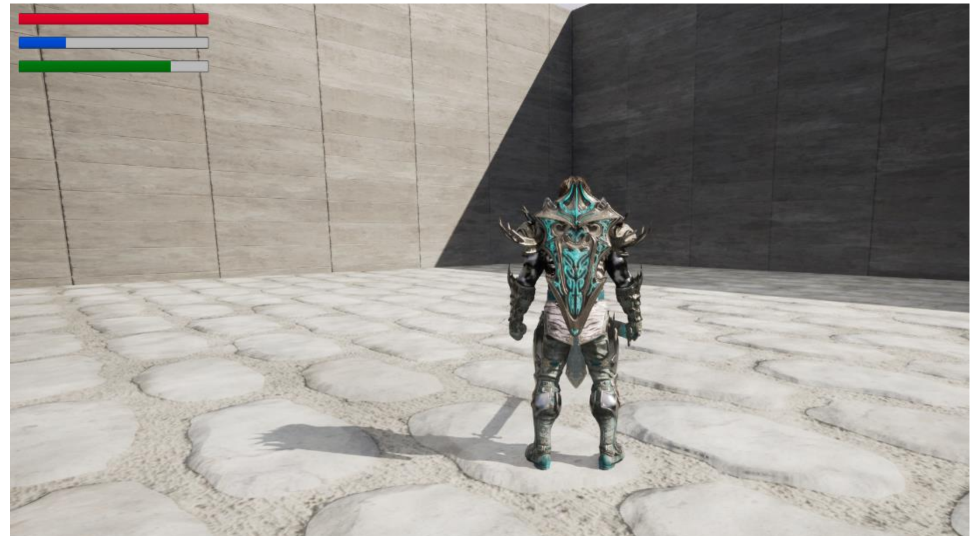


Figura 47: Barras de stats dentro del HUD de juego
(Fuente propia)

9.7.5. Curación, recuperación de maná y regeneración de estamina

Del mismo modo que el personaje principal recibe daño, se cansa o gasta el maná también debe ser capaz de regenerar estos atributos para poder seguir avanzando el juego. En el caso de la vida y el maná se recuperan de la misma forma, bebiendo una poción (siendo esto un símil a tomarse una medicina en la vida real). Por otro lado, la estamina se regenera automáticamente después de descansar un breve periodo (también como lo hacemos los humanos).

9.7.5.1. Objetos consumibles

Los objetos consumibles son aquellos que el jugador/a puede usar a lo largo de la partida pero que tienen un número limitado de usos. Dentro de este grupo se encuentran las pociones de curación y de maná. Al consumir estos objetos se suma una cantidad fija de vida o maná al personaje, pero nunca sobrepasando el valor máximo de estos.

Para ello, el autor implementa el actor ***'BP_BaseConsumable'*** que deriva del actor *BP_BaseEquipable.* Esta clase es sencilla ya que únicamente se encarga de actualizar el número de usos restantes del objeto consumible en el momento en el que se llama al método ***UseItem*** y de almacenar el sistema de partículas que se debe reproducir cuando se consume dicho objeto.

Asimismo, el autor crea dos clases que heredan de *BP_BaseConsumable*: ***BP_HealthPotion*** y ***BP_ManaPotion***. Ambas clases tienen el mismo comportamiento, sin embargo, cada una afecta a una estadística distinta, tiene su propio sistema de partículas y su propia imagen para mostrar en el HUD. En ambas clases la función *UseItem* es sobrescrita para que, después de llamar a la función padre, puedan afectar a la estadística pertinente (si quedan usos disponibles).

Por otro lado, esta mecánica debe tener una representación visual para el jugador/a más allá de que la barra de vida o maná recupera parte de su valor. Para ello, en el momento en el que el jugador/a pulsa el botón de consumir una poción, por ejemplo, se debe reproducir la animación de beber. Con el fin de evitar que el personaje se cure sin reproducir la animación o que abuse de pulsar el botón repetidas veces para curarse rápidamente, el autor adopta el sistema descrito en el apartado *9.7.2.2 Uso de Anim Notifies para decrementar stats*. En este caso crea los *Anim Notify* ***AN_ConsumeItem*** y ***AN_PlayItemParticleFX***. El primero de ellos se encarga de consumir el objeto activo y el segundo de reproducir el efecto de partículas correspondiente al objeto que se va a consumir. Pero ¿cómo se sabe qué objeto está activo? Gracias al Componente de Inventario descrito a continuación.

9.8. Componente de inventario

Tal y como se describe en los Casos de uso y Requisitos del proyecto, debe de haber un inventario que almacene los objetos que puedan ser consumidos por el personaje principal. Para crear el inventario el autor implementa el componente de inventario ***'BPC_InventoryComponent'***. Este componente está formado por un Array de referencias a objetos *BP_BaseConsumable* y un entero que indica qué posición del array (que objeto) es el activo. En cuanto a la lógica, simplemente se encarga de actualizar el índice que marca al objeto activo, añadir objetos al array y devolver el objeto activo. Es por ello que, gracias a este último método, el autor es capaz de emplear una única animación e implementación de *Anim Notifies* para usar cualquier tipo de objeto consumible, haciendo que cada uno ejecute su comportamiento y reproduzca sus efectos visuales.

9.8.1. Representación en el HUD de los objetos del inventario y armas

Una vez implementado el inventario, el jugador/a debe saber qué objeto está activo, cuántas consumiciones tiene disponibles, qué arma tiene equipada y qué escudo está usando. Es por ello por lo que el autor debe implementar una representación visual de estos datos para informar al usuario. Para ello crea el *Widget* ***WB_Inventory*** que muestra tres iconos: a la izquierda el escudo activo, a la derecha el arma equipada y abajo los objetos consumibles.

En el caso del arma y el escudo son recogidos del componente de combate, que es el encargado de almacenar dichos objetos. Por otro lado, el consumible activo es recogido del componente de inventario. Todos estos objetos heredan del actor *BP_BaseEquipable*, que contiene un atributo ***Texture2D*** que almacena la imagen de **icono** que se debe mostrar en este *Widget*.

La forma de actualizar los iconos a mostrar es asociando un evento propio del objeto a consumir o arma a equipar y un evento del *WB_Inventory*. Esto se consigue mediante ***Event Dispatchers*** [94], una utilidad de *Unreal Engine* que permite disparar uno o más eventos cuando se llama desde otra función u otro evento. Para los objetos consumibles se deben asociar dos eventos: uno para cuando se cambia de objeto activo y otro para cuando se consume el objeto activo. En el primer caso se llama al *Event Dispatcher* cuando el usuario/a cambia de objeto activo para actualizar la imagen a mostrar en el HUD y también, actualizar el número de usos del objeto para mostrar los del nuevo objeto activo. En el segundo caso, el *Event Dispatcher* es llamado cuando el personaje consume el objeto activo para actualizar el número de consumiciones del objeto activo, pero no su imagen.

Asimismo, el *WB_Inventory* debe ser incluido dentro del *WB_MainHUD*, explicado anteriormente, para que se muestre por pantalla igual que las barras de estado del personaje principal. El resultado final de lo explicado hasta el momento en relación con el HUD se muestra en la **Figura 48**.

Figura 48: HUD actualizado mostrando por pantalla las estadísticas del personaje, el arma equipada, el escudo equipado y el consumible activo.
(Fuente propia)

9.9. State Manager Component

Conforme avanza el desarrollo del proyecto el autor considera necesario hacer refactorización del sistema de *flags* empleado para indicar estados del personaje con el fin de que el manejo de estados, animaciones y acciones sea más limpio y extensible que empleando booleanos. Para ello crea el ***BPC_StateManagerComponent***, encargado de gestionar el estado de los personajes del juego, definidos en la lista ***E_CharacterStatesEnum***. Es un componente sencillo el cual tiene funciones básicas de *set* y *get* para saber qué estado tiene actualmente la entidad, establecer su estado actual y comprobar que el estado actual es o no alguno de los que el autor desea comprobar pasando una lista de estados.

El componente es empleado, por ejemplo, para marcar a una entidad como muerta y que no pueda ejercer ninguna acción salvo morir. Este también es empleado en el control de reproducción de animaciones, ya que, haciendo uso del *Anim Notify* ***AN_SetCurrentStatus*** podemos modificar el estado de un personaje para indicar que está en cierto estado mientras se reproduce una animación para que pueda o no reproducir otras animaciones. Un ejemplo de ello es cuando el personaje está rodando. Quedaría extraño que, en mitad de la voltereta, si el personaje recibe daño reproduzca la animación de daño. Lo lógico es que siga rodando, pero sí

que reciba el daño. Otro ejemplo se muestra cuando un personaje está reproduciendo la animación de recibir daño y quiere atacar. Sería ilógico que pudiera evadirse de ese estado de recibir daño para atacar. Solo cuando ha dejado de recibir daño puede volver a moverse con libertad y huir.

9.10. Inteligencia Artificial

Este punto de la implementación conforma uno de los puntos más importantes del proyecto. En él se detalla y explica la implementación de las entidades con Inteligencia Artificial del juego: los enemigos humanoides simples (*Mob Enemy* o enemigo de mazmorra), el Jefe Final y el personaje de ayuda. En primer lugar, se introduce al lector/a en las técnicas y tecnologías empleadas para la creación de Inteligencia Artificial en videojuegos, concretamente con el motor *Unreal Engine*. En segundo lugar, se explica la implementación y diseño de un enemigo base del cual heredan los diferentes tipos de enemigos del juego. A continuación, se detalla el diseño de comportamiento de los diferentes enemigos. Por último, se detalla la implementación del personaje de ayuda.

9.10.1. Inteligencia artificial en los videojuegos

Se han planteado varias definiciones de Inteligencia Artificial desde que Alan Turing publicó su estudio *"Computer Machinery and Intelligence"* en 1950 [95]. Una de las más reconocidas actualmente es la propuesta por John McCarthy en 2004: *"Es la ciencia y la ingeniería de hacer máquinas inteligentes, especialmente programas informáticos inteligentes. Está relacionado con la tarea similar de usar computadoras para comprender la inteligencia humana, pero la IA no tiene que limitarse a métodos que son biológicamente observables".*

Asimismo, existen diversos tipos de Inteligencia Artificial, que componen dos grandes grupos: *Weak/Narrow AI* (IA débil) y *Strong AI* (IA fuerte). El primero de ellos engloba toda IA entrenada para cumplir una tarea determinada. El segundo en cambio está compuesto por *Artificial General Intelligence (AGI)* que cuenta con el mismo nivel de inteligencia que los humanos y la *Artificial Super Intelligence* que superaría los límites de la inteligencia y habilidades humanas. Si bien es cierto que este grupo aún está en un estado teórico. Por otro lado, se encuentran los subcampos de *Machine Learning*, *Deep Learning* y las Redes Neuronales, los cuales escapan al ámbito de este proyecto [96].

Pero ¿qué definición se debe dar a la Inteligencia Artificial en los videojuegos? ¿En qué grupo se engloba? El autor considera adecuada la definición proporcionada por ARM: *"La IA en videojuegos se refiere a* ***experiencias sensibles y adaptativas dentro del juego****. Estas*

*experiencias interactivas propulsadas por la IA son generadas normalmente a través de personajes no jugables (**Non-Playable Characters**, NPCs), que actúan de forma **inteligente** o creativa, como si fueran controlados por un jugador humano. La IA es el motor que determina el comportamiento de un NPC en el mundo del juego"* [97].

9.10.2. Inteligencia artificial en Unreal Engine

Generalmente en los videojuegos, la IA (representada por NPCs) tiene como base de su diseño tres secciones: movimiento, toma de decisión y estrategia. Las dos primeras contienen algoritmos y técnicas que actúan en cada agente por individual mientras que la última tiene efecto sobre equipos o grupos de varios agentes. Este modelo se representa en la **Figura 49** pero, no todos los videojuegos toman este modelo como referencia. Un ejemplo es este proyecto, donde los enemigos no colaboran estratégicamente para conseguir un objetivo común, sino que simplemente se mueven y toman decisiones para cumplir un objetivo individual.

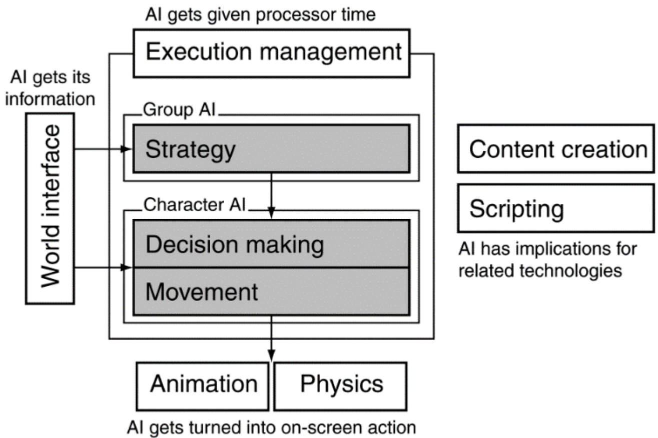


Figura 49: Modelo general de IA en los videojuegos
(Fuente: AI for Games, Third Edition, Ian Millington, 2019)

A continuación, se presentan las diversas técnicas y algoritmos de inteligencia artificial disponibles en *Unreal Engine* que dotan de inteligencia a los NPC y que les permiten llevar a cabo la percepción del entorno, el movimiento y la toma de decisión.

9.10.2.1. Perception System

Un factor que ayuda a un NPC a tomar decisiones de forma inteligente es ser consciente del entorno que le rodea y los estímulos que percibe de este. Estos estímulos son recibidos por la IA gracias al sistema ***AIPerceptionSystem*** [98]. Este sistema dota al NPC con capacidades como: saber de dónde proviene un sonido, saber si ha visto a otro personaje o si ha recibido daño. Asimismo, para poder recibir esta información, el NPC debe contener el

AIPerceptionComponent, que actúa como perceptor de los estímulos. Este componente tiene una serie de eventos asociados que son disparados en el momento que un estímulo es percibido.

Para los enemigos de este videojuego el autor únicamente considera necesarios el sentido de la vista y el estímulo relacionado con recibir daño. La **Figura 50** muestra los diferentes parámetros de configuración de dichos sentidos.

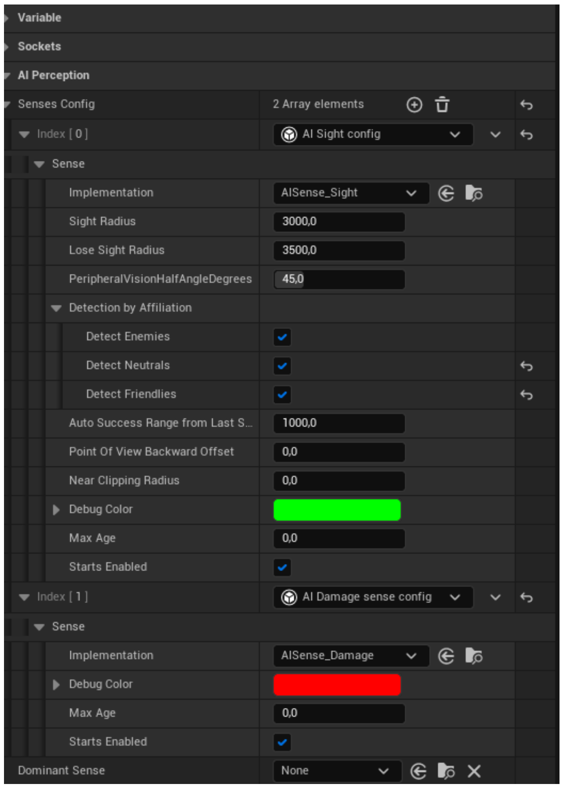


Figura 50: Parámetros de configuración de la vista y percepción de daño
(Fuente propia)

Los parámetros más relevantes para modelar diferentes tipos de enemigos son el radio de visión, el ángulo de visión y el tiempo que recuerda haber recibido ese estímulo.

9.10.2.2. Pathfollowing Component

Otro de los componentes implementados en *Unreal Engine* es el ***Pathfollowing Component*** [99]. Este componente es el encargado de calcular el camino que debe seguir un NPC hasta una posición objetivo. Incluye todos los métodos necesarios para el cálculo de puntos a seguir hasta llegar a la posición final, así como todas las variables de configuración necesarias para otorgar diferentes comportamientos a cada entidad. Algunas de estas variables son: el radio de llegada, indicar si la entidad debe frenarse conforme se acerca al objetivo o, en caso de tener que alcanzar la posición de otro actor, una referencia a ese actor para ir actualizando el camino conforme se mueve el actor objetivo.

Para que este componente se active y pueda obtener información del mundo de juego es necesario añadir una ***NavMeshBoundsVolume***. Este elemento se añade directamente sobre el nivel del juego y determina automáticamente el área que tiene disponible para hacer los cálculos el *Pathfollowing Component*. Resumidamente, la *NavMesh* divide el escenario de juego en polígonos que luego son subdivididos en triángulos por cuestiones de eficiencia. Estos **triángulos** son los **nodos** de un **grafo** gigante donde si dos triángulos son **adyacentes** significa que están **conectados**. De esta manera, el componente de *Pathfollowing* tiene como base de su implementación algoritmos de búsqueda de caminos como el A*. Habitualmente, se toma como heurística el camino más corto o más rápido, pero dependiendo del juego puede haber otros factores que deban tenerse en cuenta como por ejemplo si hay enemigos disparando por ese camino o si el personaje puede nadar o subir una montaña. Dependiendo del caso puede ser que el mejor camino sea uno más largo donde haya cobertura de fuego amigo, tenga que bordear el lago si no sabe nadar o bordear la montaña si no hay ningún túnel o no sabe escalar. En ese caso se debe modificar y extender la implementación base de este sistema. En cuanto a este proyecto no es necesario modificar la implementación ya que la implementación genérica se adecua a las necesidades del juego.

9.10.2.3. Steering Behaviors

Los ***Steering Behaviors*** son comportamientos dirigidos, es decir, tareas que dirigen el movimiento del personaje con IA. Algunos de los comportamientos dirigidos implementados en los videojuegos son:

- *Pathfollowing*: Dado un camino, calcular la posición inicial de la entidad y el punto final y seguir el camino (*Path*) indicado.
- *Seek and Flee*: Moverse hacia un punto determinado que se está buscando o alejarse de un punto o entidad determinada (huir).
- *Wander*: Esperar o vigilar sin movimiento.
- *Collision Avoidance***:** Evitar obstáculos u otras entidades que se encuentren en el camino.
- *Patrol*: Dados unos puntos de patrulla, desplazarse de uno a otro de forma cíclica.

Cada uno de estos comportamientos tiene su propia implementación en cada juego y no existe una única solución. Más adelante el autor explica más detalladamente los *Steering Behaviors* que son necesarios para este proyecto y cómo son implementados. Sin embargo, *Unreal Engine* tiene implementados algunos comportamientos dirigidos como el *MoveTo*, que mueve a la entidad de un punto a otro.

9.10.2.4. Behavior Trees and Blackboards

Los *Behavior Trees* o ***Árboles de comportamiento*** son una de las técnicas más utilizadas en el desarrollo de IA en los videojuegos. Son la herramienta encargada de la toma de decisión de las entidades con IA, son el cerebro.

Son denominados Árboles de comportamiento porque su implementación se basa en la estructura matemática y de teoría de grafos de un Árbol. Un árbol describe una relación entre nodos de forma similar a la que lo hace un árbol genealógico. En este caso se parte de un **Nodo Raíz** (*Root*), que no tiene nodo padre. A partir de este, puede haber uno o más nodos hijo, pero cada nodo hijo únicamente puede tener un nodo padre. Los nodos que no tienen ningún hijo son llamados **Nodos Hoja** (*Leaf*). La **Figura 51** muestra un ejemplo de un árbol y los nombres de los diferentes nodos que lo componen.

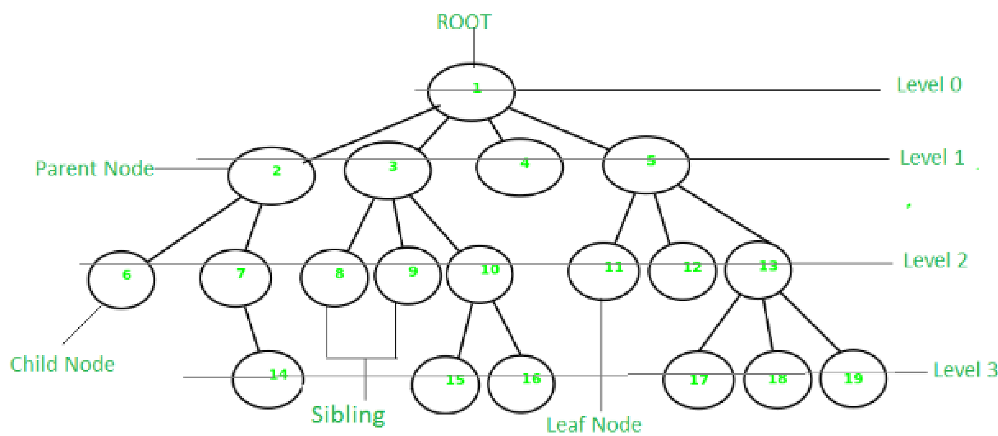


Figura 51: Grafo en forma de Árbol
(Fuente: https://www.geeksforgeeks.org/introduction-to-tree-data-structure-and-algorithm-tutorials/)

En el contexto de *Unreal Engine*, la forma de **leer** este grafo es de **arriba hacia abajo** y los nodos son **evaluados de izquierda a derecha**. En otros contextos es posible que sea de forma contraria, leyendo los nodos de izquierda a derecha y evaluados de arriba abajo.

Mientras que el árbol se encarga de ejecutar la lógica (pensar), la ***Blackboard*** (Pizarra) se encarga de almacenar la información que fundamenta los *'pensamientos'* y decisiones tomadas por el NPC. Esta estructura de datos guarda claves de diferentes tipos: desde tipos básicos como enteros o booleanos a referencias de otras entidades del mundo de juego. Siguiendo el símil con un ser humano, la pizarra sería la **memoria** o el conocimiento. Recibe este nombre ya que, de forma similar a la pizarra en un aula, las entidades pueden compartir información en una única pizarra (como lo harían un grupo de alumnos y alumnas) y en función de esa información cada una actúa de una forma u otra dependiendo del razonamiento que haga en base a esa información.

A continuación, el autor introduce brevemente los tipos de nodos que forman un Árbol de comportamiento en *Unreal Engine*:

- ***Root***: Es el nodo donde empieza la ejecución del árbol. La raíz únicamente puede tener un hijo y debe ser del tipo *Composite.*
- ***Tasks***: Nodos hoja que encapsulan acciones finales, no toman decisiones, sino que ejecutan un comportamiento. Estos nodos reportan éxito o fallo dependiendo de si su ejecución ha sido exitosa o no. Estos nodos no terminan su ejecución hasta que se reporta un final u otro. Los encargados de gestionar el resultado son los nodos *Composite*.
- ***Composite Nodes***: Son los únicos nodos que no son hojas y definen el flujo de cómo se ejecutará la subrama de la que son raíz. Engloban a los nodos de tipo:
  - ***Selector***: Un nodo *Selector* ejecuta el nodo hijo más a su izquierda (puede ser un nodo hoja o una rama entera). En caso de que ese primer hijo reporte éxito, el nodo *Selector* no ejecuta sus siguientes hijos y reporta éxito a su nodo padre. En caso de que el hijo más a la izquierda falle, continúa con el siguiente más a la izquierda. Y así sucesivamente hasta que uno de ellos reporte éxito. En caso de que todos los hijos del nodo Selector reporten fallo, este reportará fallo a su padre.
  - ***Sequence***: El nodo secuencia funciona al contrario que el nodo selector. Este nodo ejecuta su hijo más a la izquierda, si este reporta éxito continúa con la ejecución de su siguiente hijo más a la izquierda. Si todos los hijos reportan éxito, el nodo secuencia reporta éxito a su nodo padre, pero en el momento que un hijo del nodo secuencia falla se detiene la ejecución del resto de nodos hijos y se reporta fallo, con lo cual toda la subrama del nodo secuencia ha fallado.
  - ***Simple parallel***: Es un tipo de nodo que únicamente puede tener dos hijos y ejecuta ambos en paralelo. El hijo más a la izquierda debe ser una *Task* y el otro puede ser otra *Task* u otro *Composite* (dando lugar a un subárbol). El nodo principal es la *Task* más a la izquierda y el nodo padre tendrá éxito o fallo según reporte el hijo principal. Además, el nodo padre puede parar la ejecución en el momento que su hijo reporta un estado o esperar a que el segundo hijo acabe su ejecución. Normalmente, este comportamiento se puede conseguir con los otros tipos de nodos *Composite*.
- ***Decorator Nodes***: Son condiciones que se pueden añadir a los nodos *Composite* o *Task* para decidir si un nodo se ejecutará o no. Son la lógica de ejecución del árbol. Estos

nodos pueden reportar un fallo preventivo de una tarea o una rama entera del árbol si no se cumplen las condiciones que él establece. También, puede mantener en ejecución una subrama del árbol hasta que se cumpla cierta condición, esperar cierto tiempo para poder volver a ejecutar la tarea o subrama o incluso cambiar el resultado que se le reporta.

- ***Service Nodes***: Los nodos servicio también se añaden a los nodos *Task* o *Composite* y estarán en ejecución mientras cualquier hijo del subárbol del nodo en el que están esté en ejecución. Están en ejecución continuamente y son los encargados de actualizar los valores de la *Blackboard* en tiempo real. Este tipo de nodos son tan específicos de la aplicación que *Unreal Engine* no cuenta con un gran abanico de ellos, sino que es más común que sean implementados por los desarrolladores del juego.

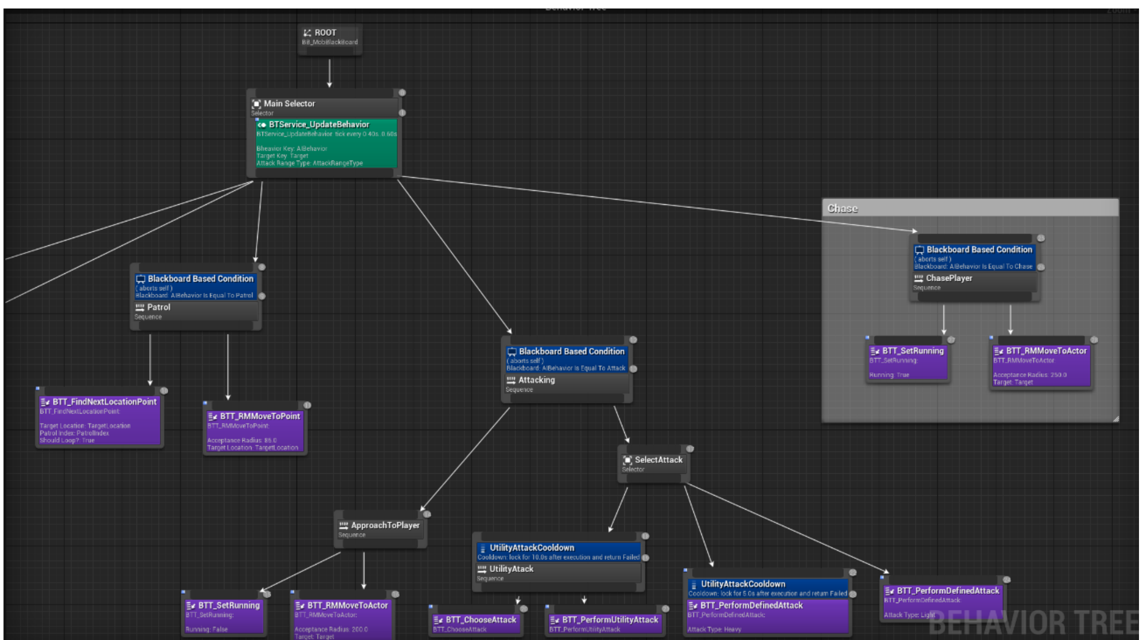


Figura 52: Parte de un Árbol de decisiones en Unreal Engine.
(Fuente propia)

La **Figura 52** muestra una parte de un árbol de comportamiento en *Unreal Engine*. Los nodos de color lila son los nodos *Task* (hojas del árbol). Aquellos en color gris son los *Composite*. Los nodos azules son los *Decorators*, los cuales se encuentran dentro de los nodos *Composite*. El nodo de color verde es el *Service* que actualiza las claves de la *Blackboard*. Por último, el nodo *Root* (raíz), de donde parte la ejecución del árbol.

En cuanto al funcionamiento de los árboles de comportamiento en *Unreal Engine*, puede diferir de otras implementaciones o usos de esta técnica en otros ámbitos. En *Unreal Engine*, los *Behavior Trees* son conducidos por eventos. Esto quiere decir que esperan pasivamente escuchando eventos que disparen algún evento que modifique los datos o la ejecución del árbol

en vez de comprobar cada fotograma si ha cambiado el estado interno del árbol. Esto supone un beneficio en términos de eficiencia y depuración de la aplicación. Por otro lado, en otras implementaciones normalmente los nodos condicionales son nodos hoja que no hacen nada más que reportar éxito o fallo. En *Unreal Engine* se usan los *Decorators* para esa tarea ya que así se mantiene más limpio de forma visual el árbol y se sabe exactamente qué acciones relevantes se puede o se están ejecutando en el árbol. Por último, gracias a los *Simple Parallel nodes* y las propiedades de los *Decorators*, el desarrollador puede conseguir el mismo resultado para comportamientos concurrentes que haciendo uso de nodos de ejecución paralelos complejos que pueden ejecutar varias subramas del árbol al mismo tiempo, haciendo más compleja la tarea de depuración.

9.11. Implementación del actor *BP_MasterAI*

Dadas las características del proyecto, los enemigos comparten la mayoría de las funcionalidades del personaje principal, es por ello que la clase base para los enemigos se crea a partir de la clase *BP_PlayerCharacter*. La clase padre de los enemigos en este proyecto es la clase ***BP_MasterAI***. En ella se implementan todos los métodos y se almacenan todas las variables comunes a todos los tipos de enemigo.

9.11.1. Componentes y armas

En cuanto a componentes, los enemigos cuentan con los mismos componentes que el personaje principal, excepto el de inventario, ya que los enemigos no tienen la capacidad de recoger elementos del entorno ni consumir objetos. Esta reutilización de componentes ofrece una rápida implementación de un enemigo base sin necesidad de reescribir código o duplicarlo. Por otro lado, las armas, en vez de ser recogidas del entorno son creadas en el evento *BeginPlay* del *blueprint* de cada tipo de enemigo ya que no todos tienen el mismo tipo de arma y no todos tienen el mismo comportamiento al iniciar la partida.

9.11.2. Principales diferencias con la clase *BP_PlayerCharacter*

Pese a compartir la mayoría de la implementación con la clase *BP_PlayerCharacter*, los enemigos implementan otras funcionalidades y almacenan datos adicionales o distintos. El primer ejemplo son los *assets* visuales que se muestran cuando un enemigo es fijado por el jugador/a y la barra de vida un enemigo. El primero de ellos es un *widget* visual con forma de punto que se dibuja sobre el enemigo fijado mientras está vivo y el jugador/a lo tiene marcado como objetivo. En cuanto a la barra de vida del enemigo, esta es una barra de vida similar a la del personaje principal pero que se muestra encima de la cabeza del enemigo mientras está vivo y está

marcado como objetivo. Asimismo, simulando la implementación de los *soulslike*, si el enemigo no está fijado pero recibe daño, esta barra de vida también aparece en pantalla por unos instantes y luego desaparece si el enemigo no vuelve a recibir daño.

La siguiente diferencia es que los enemigos necesitan almacenar un array con puntos de patrulla (*TargetPoints*). Estos puntos de patrulla son utilizados para indicar a dónde se tiene que dirigir el enemigo en el caso de que esté ejecutando el comportamiento de patrulla (más adelante se explica en detalle la implementación de los comportamientos de los enemigos). Asimismo, necesitan almacenar otras variables empleadas en el cálculo de distancia con el jugador/a (uno de los posibles factores que determinen el tipo de ataque), puntos a seguir por el mapa hasta llegar al jugador/a para atacarle o datos que estén implicados en el cálculo de utilidad de una acción.

9.11.3. *Animation blueprints* para los enemigos

Por otro lado, el *animation blueprint* del personaje principal también sirve como base para los *animation blueprints* de los enemigos, solo que en este caso se crea directamente uno para cada enemigo ya que al ser esqueletos diferentes no pueden compartir una misma base de la que heredar.

9.12. AIController

De la misma forma que el personaje principal es controlado por un controlador de tipo *Player Controller*, las entidades con IA deben ser controladas por un controlador de tipo ***AIController***. Mientras que con *Player Controller* se centra en recibir input por parte de la persona que está jugando, el *AIController* se centra en recibir estímulos del entorno y del mundo de juego. A su vez, es el encargado de tomar decisiones, ejecutar acciones y reaccionar al entorno sin la necesidad del input de un humano.

En este caso, el autor implementa el controlador ***BP_AIController,*** que hereda de la clase *AIController* y el cual contiene los componentes:

- ***Actions Component***: Encargado de implementar los métodos de utilidad para obtener al personaje que controla, el propio controlador y ejecutar acciones entre otras funcionalidades.
- ***AIPerception***: Se ha descrito anteriormente. Es el encargado de recibir los estímulos del entorno.

- ***Path Following Component***: Se ha descrito anteriormente. Es el encargado de calcular el camino que debe seguir un NPC hasta una posición objetivo.

Este controlador, aparte de los ya conocidos eventos ***BeginPlay*** y ***Event Tick***, también implementa el evento ***OnPossess***, el cual es invocado cuando empieza la posesión de una entidad que utiliza este controlador. En este momento el autor almacena una referencia de la entidad que posee y asigna el árbol de comportamiento que debe ejecutar el controlador. Dentro de cada entidad con IA se encuentra una referencia al árbol de comportamiento que tiene que ejecutar, haciendo así fácil e intuitivo el hecho de asignar un tipo de árbol u otro al controlador.

Por otro lado, el componente de percepción implementa el evento ***OnPerceptionUpdated***, el cual es disparado en el momento que la percepción de la entidad con IA recibe un estímulo del mundo de juego. En este caso, el autor implementa la función ***UpdatePerception***, encargada de actualizar la entidad objetivo del enemigo, es decir, actualiza la referencia al jugador/a dependiendo de si lo ve o no y si ha recibido daño o no.

9.13. Implementación de tareas y comportamientos específicos

A continuación, el autor procede a explicar algunas de las tareas y comportamientos que ha tenido que implementar específicos para el proyecto, ya sea porque no están implementados en el motor o porque la implementación que ofrece el motor no se adecua a las necesidades del proyecto.

9.13.1. Steering Behaviours

Uno de los problemas que se encuentra el autor a la hora de utilizar los comportamientos dirigidos implementados en *Unreal Engine* es que estos hacen uso del componente de movimiento, el cual dirige el movimiento de las entidades por propiedades físicas (velocidad y aceleración). Esto es útil en la mayoría de los videojuegos donde el movimiento es dirigido por físicas y las animaciones se emplean para dar sensación de movimiento conforme se mueve la cápsula del personaje. Pero como se ha explicado anteriormente, el autor emplea animaciones con ***Root Motion***, lo que quiere decir que son las animaciones las que mueven la cápsula del personaje. Es por ello por lo que el autor ha implementado sus propios métodos para moverse de punto a punto (cuando el personaje patrulla, por ejemplo) o cuando debe moverse hacia la posición de otro personaje (cuando persigue al personaje principal).

9.13.1.1. MoveToActor and MoveTo

Ambos métodos han requerido la **modificación** del método ***Get Control Velocity***, que calcula el vector director hacia donde se debe mover el personaje. Cuando un personaje con IA sigue un camino calculado por el componente de *Path Following*, este conoce cuáles son los puntos que ha de seguir hasta llegar al objetivo. Con ello, el autor debe calcular el vector director como el vector unitario desde la posición actual de la entidad hasta el siguiente punto al que debe ir. Una vez conocido el vector, el personaje se orienta hacia él de la misma manera que lo hace el personaje principal.

En cuanto a la implementación específica del evento ***MoveToActor***, el autor hace uso de la función ***Find Path to Actor Synchronously***. Esta función calcula los puntos de camino que debe seguir la entidad hacia un actor. Además, si la posición del actor cambia, estos puntos son recalculados para que el movimiento del NPC se redirija. Para poder ejecutar esta función, el enemigo debe tener una referencia válida del personaje al que seguir. Por otro lado, el evento *MoveTo*, hace uso de la función ***Find Path to Location Synchronously***, donde se calculan los puntos a seguir hasta una posición objetivo fijo. Ambas funciones implementan los cálculos necesarios para la búsqueda de camino y esquiva de obstáculos que les impidan llegar al punto objetivo.

9.13.2. Custom Tasks

Pese a que ***Unreal Engine*** tiene implementadas algunas tareas genéricas ha sido necesario implementar otras tareas personalizadas para llevar a cabo acciones concretas para los enemigos.

9.13.2.1. Custom movement tasks

- ***BTT_FindNextLocationPoint***: Encargada de actualizar punto de patrulla al que se debe dirigir el personaje si está patrullando.
- ***BTT_RMMoveToActor***: Encargada de ejecutar el movimiento del personaje hasta alcanzar al actor objetivo.
- ***BTT_RMMoveToPoint:*** Encargada de ejecutar el movimiento del personaje hasta alcanzar el punto de patrulla objetivo.
- ***BTT_SetRunning:*** Encargada de establecer a verdadero o falso si el personaje está corriendo. Normalmente el enemigo corre cuando está persiguiendo al jugador/a.

9.13.2.2. Custom attack tasks

- ***BTT_ChooseAttack***: Tarea encargada de calcular qué ataque es más útil para el personaje (el cálculo de utilidad se detalla más adelante).

- ***BTT_PerformBossAttack***: Tarea que recibe como parámetro qué tipo de ataque ejecutar el jefe final.
- ***BTT_PerformDefinedAttack***: Tarea que recibe como parámetro el ataque que debe ejecutar el enemigo. La tarea reporta éxito si se ha reproducido la animación y fallo en caso contrario.
- ***BTT_PerformUtilityAttack***: Tarea que se encarga de ejecutar el ataque calculado por la tarea *BTT_ChooseAttack*. Reporta éxito si el ataque se lleva a cabo y fallo en caso contrario.

9.13.3. Custom Services

Dada la especificidad de las características del proyecto, el autor no puede emplear los servicios para árboles de comportamiento empleados en el motor. De modo que para actualizar qué comportamiento debe ejecutar la entidad con IA crea el servicio ***BTS_UpdateMobEnemy.*** En este momento de desarrollo el servicio se encarga de cambiar el comportamiento a ejecutar entre: *Patrol*, *Chase* y *Attack*. En caso de que el enemigo no sepa dónde está el personaje del jugador/a, este seguirá su patrulla. En el momento que ve al personaje principal, el comportamiento pasa a ser *Chase* y lo persigue hasta estar suficientemente cerca (rango marcado por una variable interna). En el momento que entra dentro del rango de ataque cambia su estado a *Attack.* Lo que se actualiza en este servicio es la clave *Behavior* almacenada en la *Blackboard* asociada al árbol de comportamiento. De este modo, el autor puede hacer uso de *Decorators* para saber qué subrama del árbol ejecutar en función del estado que haya indicado en la pizarra.

9.13.4. UtilityAI

En esta parte de la implementación, el autor pone en práctica un punto especial aprendido en el ABP durante el desarrollo del motor de IA. Todo lo explicado anteriormente resulta siempre en el mismo comportamiento para todos los enemigos. Esto resulta en una IA monótona y contra la que el jugador/a no va a querer jugar. Para aportar variedad y salir un poco de la metodología *if-then* se integra ***UtilityAI*** en la toma de decisiones.

Esta estrategia consiste en calcular la utilidad que tiene acertar o fallar una acción, de modo que cada vez que la IA tiene que elegir una acción escoge la que más utilidad le va a aportar. La forma de modelar este comportamiento viene definida por una probabilidad de acierto y fallo así como una ponderación de valor por acertar o fallar cada acción. En este proyecto, las IAs almacenan estos valores en las variables: *confidenceSuccess*, *confidenceFail, riskyness*, *betSuccess* y *betFail*; todas ellas se encuentran definidas en la clase *BP_MasterAI*.

En el caso de ***confidenceSuccess*** y ***confidenceFail*** se da una probabilidad entre 0 y 1 de acertar o fallar en una acción. La probabilidad de fallo se calcula como 1-probabilidad de acierto. Cuanto más alto el valor de acierto, más confiada se siente la entidad y la probabilidad de coger un ataque más arriesgado pero más potente será mayor.

La variable ***riskyness*** es una probabilidad de riesgo. Este valor sirve para definir los dos valores *betSuccess* y *betFail*. Cuanto más arriesgada sea la entidad, más valor le dará al acierto y menos valor le dará al fallo y viceversa. Las variables ***betSuccess*** y ***betFail*** indican la relación entre el valor de acertar y fallar en una acción. La mayor relación es de 1 a 11, para ambos extremos.

La inicialización de estas variables se lleva a cabo en el método ***InitializeUtilityAI*** de la clase *BP_MasterAI* y el cálculo de utilidad se presenta principalmente a la hora de seleccionar ataque.

Una vez detalladas las variables de utilidad que dotan de personalidad a un enemigo, es momento de explicar cómo el autor hacer el cálculo de utilidad. En primer lugar, cada ataque tiene asociado un valor de utilidad de 0 a 1. De este modo, el método ***CalculateAttackUtility*** de la clase *BP_MasterAI* se encarga de recoger todos los tipos de ataque y sus valores de utilidad para iterar sobre ellos haciendo el siguiente cálculo:

- En primer lugar, se guarda en una variable local el valor de utilidad del ataque que se está analizando (***L_CurrentAttackUtility***).
- A continuación, se calcula el valor de acertar y el valor de fallar si se eligiese ese ataque:
  - ***L_SuccesValue*** *= L_CurrentAttackUtility * betSuccess*
  - ***L_FailValue*** *= - (L_CurrentAttackUtility * betFail + L_CurrentAttackUtility)*
- Ahora es momento de calcular la utilidad de acertar y fallar:
  - ***L_SuccessUtility*** *= L_SuccessValue * confidenceSuccess*
  - ***L_FailUtility*** *= L_FailValue * confindeceFail*
  - ***L_TotalUtility*** *= L_SuccessUtility + L_FailUtility*

Este último valor es la utilidad total de elegir este ataque. Si es mayor que el valor de utilidad total actual, se actualiza el índice del ataque a seleccionar al final del bucle. Una vez seleccionado el ataque con más utilidad se comprueba si tiene estamina suficiente para hacer dicho ataque. En caso de tener estamina suficiente se devuelve el tipo de ataque seleccionado, sino se devuelve el ataque ligero como ataque seleccionado.

Por último, para que el comportamiento del enemigo sea variable a lo largo del juego, se establece un porcentaje pequeño que añade o quita confianza al enemigo en función de si el ataque ha impactado en el personaje principal o ha sido bloqueado.

9.14. Enemigo simple (*Mob Enemy*)

En este apartado se detallan las principales características que dan forma al enemigo simple o ***MobEnemy***. El primer paso para implementarlo es crear el fichero ***BP_HumanoidEnemy*** que hereda de *BP_MasterAI*. La principal diferencia es que este *blueprint* sí que tiene asociada una malla esquelética y un *animation blueprint*. La **Figura 54** muestra el modelo empleado para el enemigo simple.

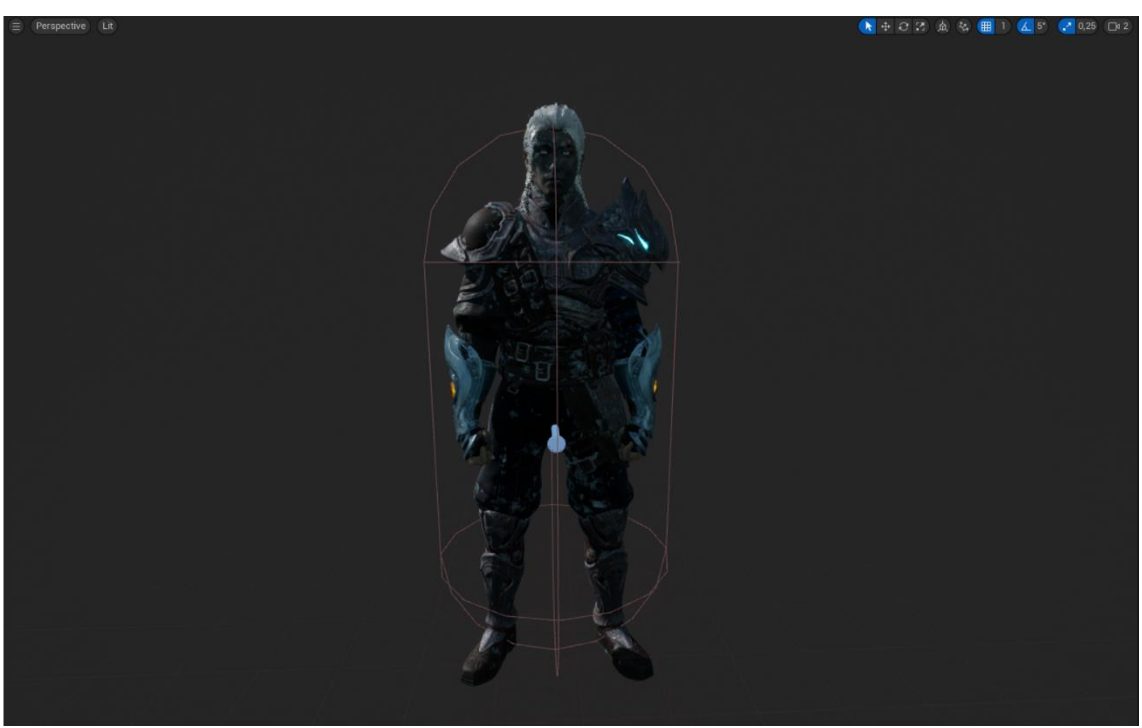


Figura 54: Malla esquelética del MobEnemy obtenida del personaje Kwang del juego Paragon
(Fuente propia)

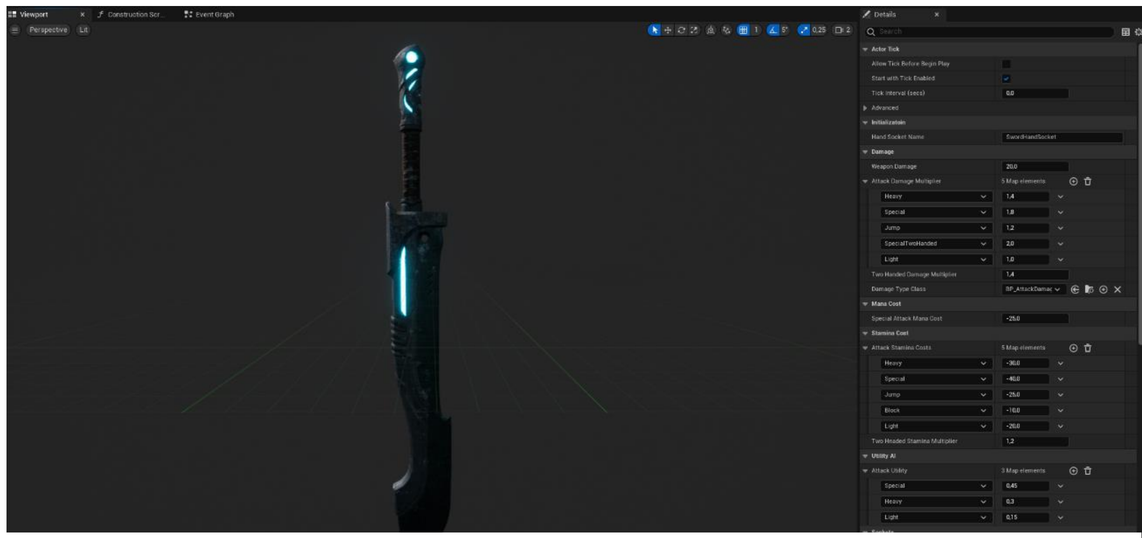


Figura 53: Espada del enemigo simple y sus datos para el daño, costes y utilidad
(Fuente propia)

En cuanto al arma del personaje, es necesario crear otro *blueprint* que herede de la clase *BP_BaseWeapon*. En este caso el autor crea la clase ***BP_KwangSword***, a la que le asigna la malla

de la espada de *Kwang* (personaje del juego *Paragon*) y define los datos personalizados de esta arma para el daño, coste de estamina, maná y utilidad de los ataques. Esto se ilustra en la **Figura 53**.

9.14.1. Barra de vida del enemigo simple

Como se ha comentado anteriormente, una de las principales diferencias entre el personaje principal y los enemigos es la barra de vida. Ya se ha descrito su comportamiento, de modo que a continuación, se explica la implementación de dicho *widget*. Para poder mostrar la vida del enemigo por pantalla el autor crea el *widget* ***WB_HealthBar***, un widget con un comportamiento similar al *WB_StatsBar* pero específico para mostrar la vida de los enemigos. Este *widget* se asigna como componente al *blueprint BP_MasterAI*, del mismo modo que tiene como componente el *widget* para mostrar el punto cuando está marcado como objetivo. Esto permite al autor mover la barra de vida para que se muestre encima de la cabeza del enemigo desde el editor del *blueprint*. Asimismo, cuando la vida del enemigo se actualiza lo hace también su barra de vida mediante eventos asociados. La **Figura 55** muestra el resultado de este *widget*.

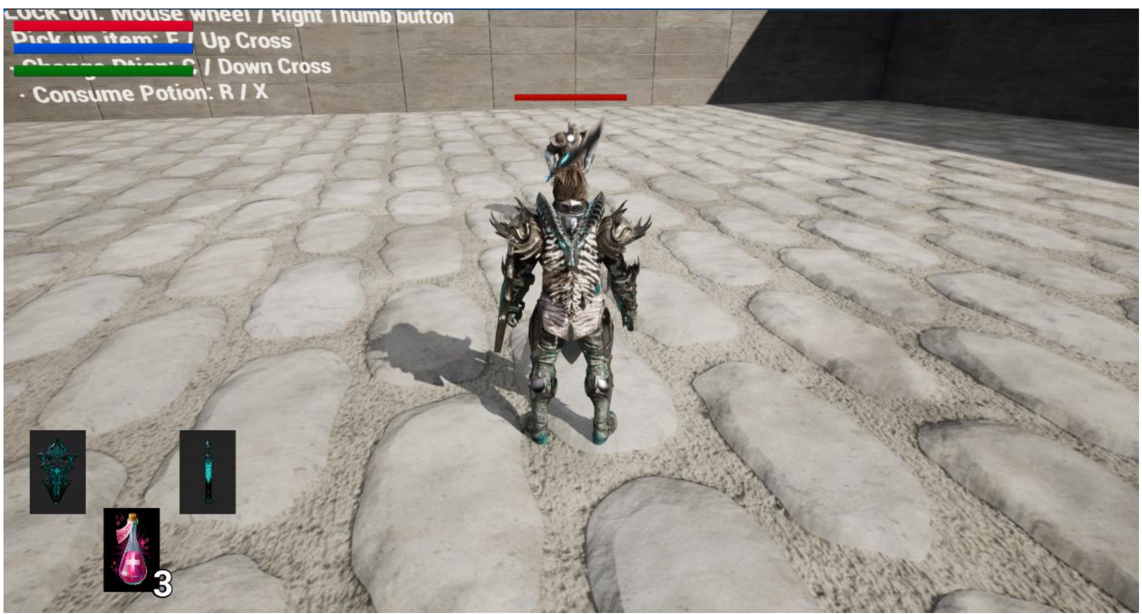


Figura 55: Barra de vida del enemigo simple
(Fuente propia)

9.14.2. *AIPerception* del enemigo base

En cuanto a la percepción del enemigo base, tiene un ángulo de visión de 180 grados con un radio de visión de 750 unidades y un radio exterior de pérdida de visión de 1000 unidades. Además, la posición del personaje que ha visto se actualiza cuando la diferencia desde la última posición percibida es de 500 unidades y tiene una memoria de 5 segundos. En cuanto al daño, únicamente se parametriza la memoria de haber recibido daño cada 5 segundos. En el vídeo demostrativo se muestra cómo se depura este sistema en *Unreal Engine*.

9.14.3. Árbol de comportamiento y *blackboard*

Por último, lo más importante de esta entidad es el árbol de comportamiento, ilustrado en la X. Sin él la entidad quedaría sin la capacidad de 'pensamiento'. Asimismo, el árbol de comportamiento necesita una pizarra para almacenar y modificar la información que conoce. El hijo de la raíz es un selector que contiene un ***Custom Service*** encargado de actualizar el comportamiento de la entidad cuando el valor ***Target*** de la pizarra tiene un valor válido (el enemigo sabe dónde está el personaje a atacar). Al ser un nodo selector este evaluará a sus nodos hijo de izquierda a derecha hasta que uno de ellos reporte éxito. Para ahorrar tiempo de proceso y evitar comportamientos inadecuados todos los hijos implementan uno o varios ***Decorators*** con precondiciones que se deben cumplir para continuar con la evaluación de sus hijos. Si alguna de estas precondiciones no se cumple se reporta fallo automáticamente y no se sigue evaluando esta rama del árbol.

Así pues, mientras la entidad no sabe dónde está su objetivo, sigue una patrulla formada por puntos objetivo colocados en el mundo de juego. Sin embargo, cuando la percepción de la IA se actualiza (bien por la vista o por el daño) la clave *Target* se actualiza y pasa a almacenar una referencia a la entidad que ha visto o que le ha atacado. En este momento, cuando se evalúa el *Decorator* de la secuencia que define la patrulla, reporta fallo, con lo que el selector pasa a evaluar sus siguientes hijos.

En este momento la entidad desenfunda su arma para habilitar el combate y a continuación puede actuar de tres formas posibles. La primera en ser evaluada en el árbol es si recular y ganar estamina por si no tiene suficiente energía para realizar el ataque más flojo. El siguiente comportamiento en ser evaluado es el de atacar. Si el objetivo está dentro del rango de ataque del enemigo, este último procede a acercarse un poco, calcular cuál es su ataque de utilidad y realizar dicho ataque. En caso de que el objetivo esté fuera del rango de ataque, el enemigo le persigue corriendo hasta que este está de nuevo dentro del rango de ataque. Si durante la persecución el objetivo escapa del rango de visión del enemigo, este deja de perseguirlo, enfunda su arma y vuelve a la patrulla.

La **Figura 56** muestra el árbol de comportamiento del enemigo simple.

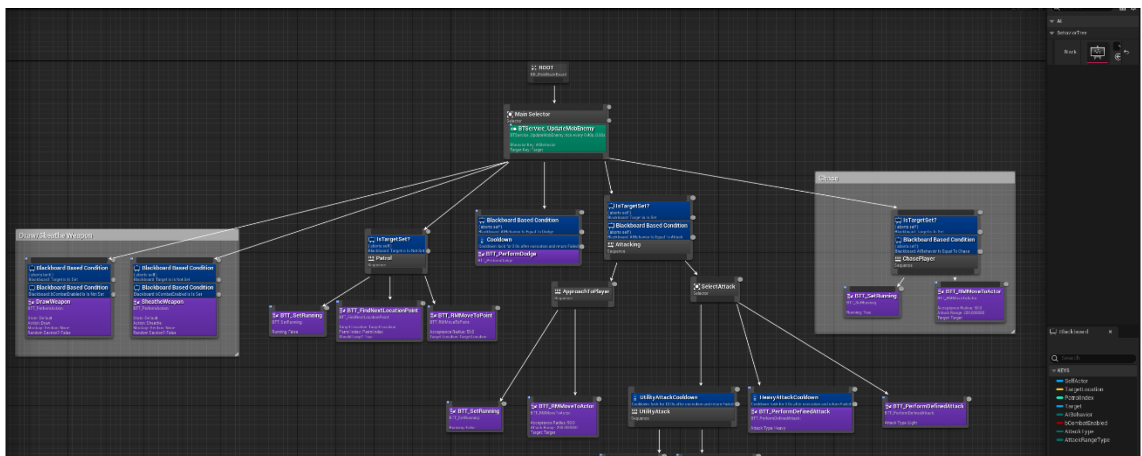


Figura 56: Árbol de comportamiento del enemigo simple.
(Fuente propia)

Asimismo, las claves que componen la pizarra son:

- ***SelfActor:*** Referencia a sí mismo.
- ***Target:*** Referencia a la entidad objetivo.
- ***TargetLocation:*** Coordenadas del punto de patrulla al que dirigirse.
- ***PatrolIndex:*** Índice del punto de patrulla actual.
- ***AIBehavior:*** Comportamiento que debe ejecutar si sabe dónde está su objetivo.
- ***bCombatEnabled:*** Booleano que indica si el combate está habilitado o no.
- ***AttackType:*** Tipo de ataque a ejecutar.
- ***AttackRangeType:*** Rango de ataque. En el enemigo simple siempre es de rango cercano.

9.15. Jefe Final

De forma similar al enemigo simple, el autor crea un nuevo *blueprint* para el jefe final (***BP_FinalBoss***) y otro para su arma (***BP_AuroraSword***). En esta ocasión el personaje elegido es *Aurora* con la skin de *Glacial Empress*, obtenida también de los *assets* gratuitos del juego *Paragon*. La **Figura 57** muestra como es el aspecto del jefe final y su arma.

Figura 57: Aurora, Glacial Empress y su espada de hielo
(Fuente propia)

Asimismo, siguiendo el mismo razonamiento que para seleccionar el modelo del personaje principal, se escoge un modelo femenino para para ejercer el rol de jefe final del juego. El modelo de *Aurora* es el que mejor encaja ya que su estética y armas son las idóneas para establecer la temática del juego entorno a ellas.

9.15.1. Barra de vida del jefe final

A diferencia del enemigo básico, la barra de vida del jefe final no se muestra cuando el jugador/a lo tiene como objetivo o cuando le ejerce daño. Además, tampoco se muestra encima de la cabeza del enemigo. Al tratarse del jefe final se debe diferenciar de la del resto de enemigos. En este caso, la barra de vida se muestra fija en la parte inferior de la pantalla, con un tamaño de aproximadamente tres cuartos del ancho de la pantalla e indicando el nombre del jefe final justo encima de ella. Esta barra de vida se muestra por pantalla en el momento que el jefe final percibe a su objetivo y no desaparece hasta que el jugador/a ha derrotado al jefe final.

Para llevar a cabo esta tarea el autor crea el *widget* ***WB_BossHealthBar***, el cual está compuesto por un *canvas* para contener un widget de tipo *WB_HealthBar* y un bloque de texto indicando el nombre del jefe final. Dado que la lógica de actualizar la barra de vida la tiene el widget *WB_HealthBar* no es necesario implementar esa lógica en el nuevo widget. De este modo, en el *blueprint BP_BossEnemy* se controla la visibilidad de la barra de vida del jefe en función de los eventos o métodos que sean ejecutados en esta clase. La **Figura 58** muestra el resultado de este *widget.*

Figura 58: Barra de vida del jefe final
(Fuente propia)

9.15.2. AIPerception del jefe final

La percepción del jefe final es una percepción sobrehumana ya que este cuenta con un ángulo de visión de 360 grados y un radio de visión de 4000 unidades. Además, para que la precisión de los ataques sea mayor, la posición del objetivo se actualiza constantemente, con lo que no hay una distancia de margen entre la posición actual y la percibida en la iteración anterior. El autor considera que esta configuración de la vista es necesaria para favorecer a la experiencia de juego ya que así evita que el jefe final pierda de vista a menudo su objetivo si se pone a su espalda o si se aleja mucho del jefe. Por otro lado, aunque el conocimiento de la posición del objetivo sea muy precisa, el autor ha diseñado los ataques y la orientación del jefe final para que la sensación de juego suponga un reto pero que no sea imposible esquivar los ataques.

9.15.3. Árbol de comportamiento y *blackboard*

La capacidad de 'pensamiento' del jefe final funciona de la misma forma que la del enemigo simple, con un árbol de comportamiento y una pizarra. Sin embargo, en este caso el primer nodo después de la raíz no ejecuta un *Custom Service*. El raciocinio y la evaluación de la información se lleva a cabo mediante *decorators* y nodos hoja ya que la única información relevante que se actualiza es la referencia al personaje principal y la distancia a la que están una entidad de la otra. Además, el jefe final ya se coloca en el nivel con el arma desenfundada y el modo de combate habilitado. Por otro lado, los ataques del jefe final se deciden por probabilidad. Sin embargo, esta es una de las características del proyecto que el autor se propone ampliar en el futuro. La **Figura 59** muestra el árbol de comportamiento del jefe final.

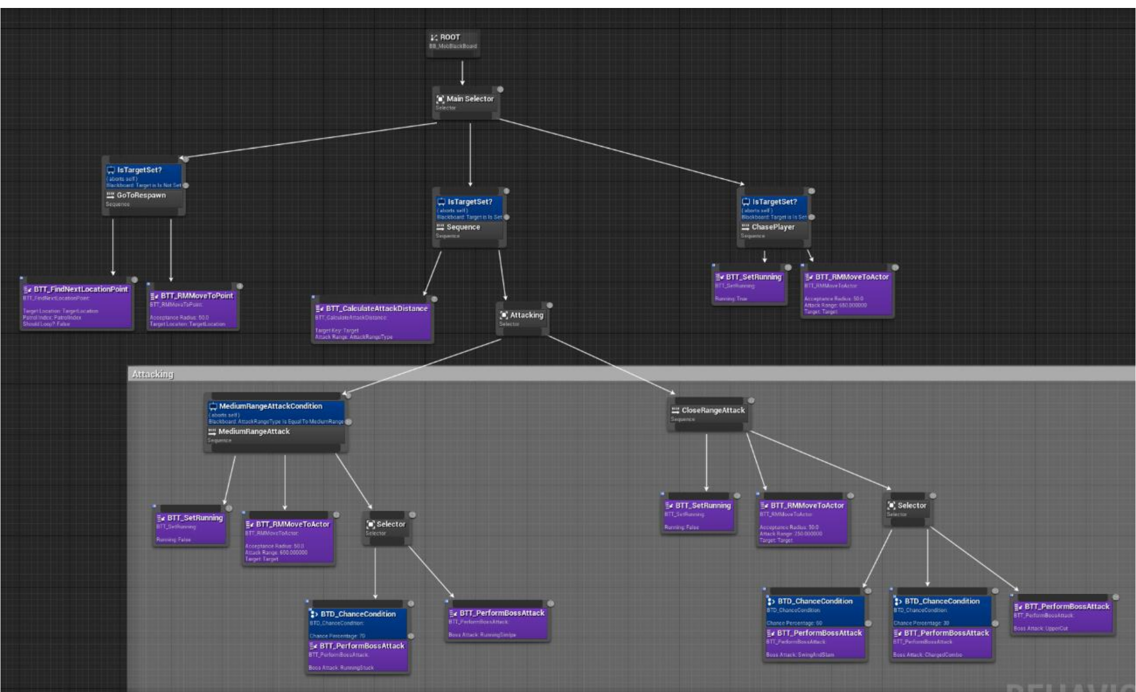


Figura 59: Árbol de comportamiento del jefe final.
(Fuente propia)

9.15.4. Combate dividido en dos fases

Uno de los objetivos de este proyecto es diseñar una pelea contra el jefe final divida en dos fases. En este punto se detalla cómo se ha implementado y las características que definen cada momento del combate.

9.15.4.1. Componente BPC_VariablePhase

Para manejar el cambio de fase, actualización de daños y modificación de efectos y mallas el autor ha implementado el componente ***BPC_VariablePhase***. Este componente encapsula los datos y la lógica para aplicar el cambio de fase en la pelea contra el jefe final. Con este

componente se libera de esa carga a la entidad *BP_BossEnemy*, manteniendo así una estructura ordenada y escalable del código del proyecto.

Este componente tiene como atributos las mallas, sonidos y efectos de partículas que se deben aplicar a la entidad que posee el componente. Contiene tanto los atributos de la fase inicial como los de la fase evolucionada. De esta forma, el jefe final empieza el combate con la malla que se ha mostrado anteriormente. Además, cuando el ataque del jefe final golpea el suelo se reproduce un sonido de golpeo contra piedra simple y el efecto de partículas es de impacto contra rocas.

Sin embargo, cuando la vida del jefe final pasa a ser menos de la mitad el combate evoluciona de fase y la malla pasa a ser la ilustrada en la **Figura 60**. Asimismo, el sonido al golpear el suelo se mezcla con un sonido reverberado de impacto de hielo y el efecto de partículas es el de un hechizo de hielo radial. Es por ello, que cuando el jefe final golpea el suelo en los ataques de la fase evolucionada, ejerce daño en área. Ambas fases del combate se muestran en el vídeo demostrativo.

Figura 60: Aurora, Glacial Empress en la fase evolucionada
(Fuente propia)

9.16. Personaje invocado

El último objetivo relacionado con la inteligencia artificial es la implementación de una entidad que ayuda al jugador/a a pelear contra los enemigos, tanto contra los básicos como contra el

jefe final. Este punto detalla la implementación de esta entidad y del elemento utilizado para su invocación.

9.16.1. BP_PlayerAI

El primer paso es crear el actor ***BP_PlayerAI*** que hereda del actor base *BP_MasterAI*. Este actor tiene la malla del personaje *Greystone* con la *skin Tough*. El autor decide emplear este personaje porque dado que el esqueleto es el mismo que el del personaje principal, no es necesario hacer otro *animation blueprint* y todas las animaciones son perfectamente útiles. Asimismo, su comportamiento es exactamente el mismo que el del enemigo simple, con lo cual no es necesario hacer grandes modificaciones, excepto que esta nueva entidad sí que puede recibir daño en área y que implementa un método para seguir al jugador/a cuando no está en combate. La **Figura 61** muestra la malla de esta nueva entidad.

Figura 61: Malla esquelética del personaje Greystone con la sking Tough del juego Paragon.
(Fuente propia)

9.16.2. BP_ToughSword

Asimismo, esta entidad debe tener un arma para poder atacar. Para ello el autor crea el actor ***BP_ToughSword***, el cual tiene la misma implementación que el resto de las armas descritas

anteriormente. La diferencia principal es su malla, ilustrada en la **Figura 62**, que es la correspondiente a la *skin* de este personaje.

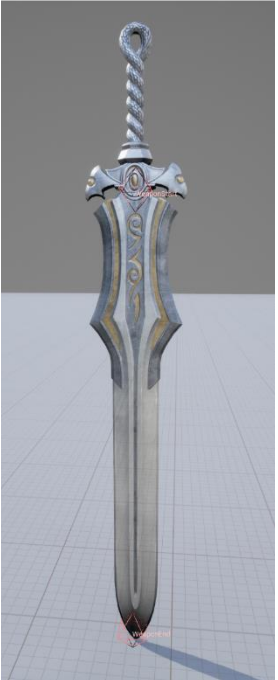


Figura 62: Espada de Greystone Tough, usada para el personaje invocado.
(Fuente propia)

9.16.3. Árbol de comportamiento y *blackboard*

El árbol de comportamiento de esta entidad es el mismo prácticamente que el del enemigo simple, pero cambiando el comportamiento de patrulla cuando no se sabe dónde está el enemigo, por el de seguir al jugador/a. Además, el autor implementa la *custom task* ***BTT_FollowPlayer***, encargada de dirigir el movimiento del personaje cuando sigue al jugador/a. Asimismo, la pizarra no tiene la clave de *TargetLocation* pero sí que almacena una referencia al personaje principal.

9.16.4. AIPerception del *BP_PlayerAI*

Del mismo modo que el resto de las características, la percepción de esta entidad tiene las mismas configuraciones que las del enemigo simple. Sin embargo, al añadir esta entidad en el proyecto el autor debe actualizar la forma de búsqueda de entidades enemigas en el ***BP_AIController***. Antes de incorporar esta entidad al juego, el método *UpdatePerception* únicamente actualizaba los sentidos si detectaba entidades con el tag ***Character.Player***. Sin embargo, este comportamiento ahora es erróneo ya que, si se mantiene así, el personaje de ayuda también va a detectar como enemigo al personaje principal. De modo que ahora cada entidad almacena una variable con el tag exacto por el cual debe buscar el método que actualiza la percepción, siendo *Character.Player* para los enemigos y ***Character.Enemy*** para la entidad que ayuda al jugador/a.

9.16.5. BP_PlayerHelper

El actor ***BP_PlayerHelper*** es un actor interactivo que sirve para crear al personaje de ayuda. Esta entidad tiene las mismas características que el resto de las entidades de este tipo, la única diferencia es su evento *Interact.* Este método se encarga de crear el personaje de ayuda y de establecer como invisible el sistema de partículas que le da visibilidad dentro del mundo de juego. Asimismo, la esfera de interacción se deshabilita para que el jugador/a no pueda volver a interactuar con ella y que se creen una cantidad indefinida de personajes de ayuda. La **Figura 63** muestra dicho objeto dentro del mundo de juego.

Figura 63: Actor BP_PlayerHelper en el mundo de juego.
(Fuente propia)

9.17. Flujo de juego

Una vez implementado todo lo necesario para poder jugar, es necesario convertir el proyecto en un producto jugable. Para ello se implementa el flujo de juego, el cual incluye el menú principal y de pausa para poder iniciar la partida, pausarla o salir de ella. Asimismo, se controla el reinicio del nivel cunado el personaje principal muere y el comportamiento cuando el jugador/a derrota al jefe final.

9.17.1. Menú principal

Para la implementación del menú principal el autor crea una nueva escena de *Unreal Engine* en blanco. Esta escena simplemente sirve para poder mostrar el menú principal por pantalla. A su vez, el menú se implementa en el *widget* ***WB_MainMenu***. Este widget está compuesto por un lienzo donde se añaden los siguientes elementos: texto con el título del juego, botón de inicio de partida, botón para salir del juego y botón para mostrar la pantalla de controles. En función del botón pulsado se ejecutará un comportamiento u otro relacionado con la carga y descarga de escenas y niveles.

9.17.2. Menú de pausa

Otro de los menús a implementar es el menú de pausa. Para ello el autor crea el *widget* ***WB_PauseMenu***. La estructura es similar a la del menú principal, sin embargo, este widget no necesita de un nivel en blanco para ser mostrado, sino que se muestra ya en el nivel de juego principal. Asimismo, la acción encargada de crear el menú se encuentra en la clase ***BP_PlayerController***. Al pulsar el botón de pausa, si el *widget* ya estaba creado, simplemente se hace visible y se establece el *input* del jugador/a como UI y juego para que pueda seguir moviendo al personaje, pero a la vez interactuar con el menú. En caso de que el widget no exista, se crea y se añade al *viewport* de juego y se procede de la misma forma que antes. En este caso el jugador/a puede interactuar con 3 botones: volver al juego, controles y quitar el juego. El botón de volver al juego simplemente esconderá el menú de pausa, el de controles mostrará la pantalla de controles y el último hará que se cierre el juego.

9.17.3. Pantalla de controles

El *widget* encargado de mostrar la pantalla de controles es el ***WB_ControlsScreen***. En este caso el widget cuenta con una imagen de fondo y dos botones: uno para volver hacia atrás y otro para cambiar la imagen de fondo. La imagen de fondo muestra los controles para jugar con mando o bien los controles para jugar con teclado y ratón. Este widget se muestra encima del resto de menús ya que solo es accesible desde ellos. Asimismo, cuando se decide volver a atrás se oculta, volviendo a quedar visible el menú de donde viene.

9.17.4. Reinicio del nivel cuando el personaje principal muere

Tal y como se especifica en la especificación de requisitos, cuando el personaje principal muere se debe reiniciar el nivel. Esta situación está formada por varios comportamientos, explicados a continuación.

9.17.4.1. Reaparecer en la última hoguera activa

Más adelante se explica en detalle la mecánica de interacción con las hogueras, pero por el momento solo es necesario saber que cuando el jugador/a interactúa con una de ellas, su posición de reaparición pasa a ser la posición donde se encuentra situada la hoguera. Asimismo, tras reproducir la animación de muerte, el personaje principal es movido a dicha posición y reproduce una animación de reaparición.

9.17.4.2. Reiniciar todas las estadísticas

Por otro lado, para que el jugador/a pueda seguir jugando es necesario que sus estadísticas y los objetos del inventario se reinicien. En este caso, la vida, el maná y la resistencia vuelven a su valor máximo, así como los objetos consumibles vuelven a contar con el número máximo de

usos. Los métodos encargados de estas tareas son ***InitializeStats*** del *StatsComponent* y ***ResetAllItemsUses*** del Componente de Inventario. También, se debe reiniciar el estado del personaje a Default y se debe volver a activar la cápsula de colisión y el componente de combate (ambos desactivados en el momento que el personaje muere).

9.17.4.3. Reiniciar a todos los enemigos

Además, para poder continuar jugando es necesario reiniciar a los enemigos. Estos reaparecen en la misma posición en la que fueron creados, con las estadísticas iniciales y con el comportamiento inicial. Además, la información de sus *blackboards* se reinicia a los valores iniciales ya que si no sabrían donde se encuentra el personaje principal e irían a por él nada más reaparecer.

Todos los comportamientos de reinicio explicados son invocados desde el método ***Reset*** que implementan todas las entidades. Para llamar a dicho método es necesario recoger a todas las entidades del nivel y llamar a su método *Reset*. La clase encargada de resetear todas las entidades del nivel es el *blueprint* ***BP_SoulslikeGameMode***, que a su vez hereda de la clase *GameModeBase*. Un ***GameMode*** es un mánager de juego, el cual tiene definidas algunas reglas básicas sobre el flujo de juego e inicialización de este [100]. El autor decide que esta clase encapsule los métodos de reseteo de entidades y flujo de juego ya que, aunque los métodos de acceso y manejo de entidades están disponibles de forma global, prefiere mantenerlo aislado del comportamiento de cada entidad. Dos de estos métodos son: ***GetActorOfClass*** y ***GetAllActorsOfClass***. Ambos métodos funcionan de forma similar, reciben por parámetro una referencia de clase para saber el tipo de actor o actores que debe devolver. Una vez obtenidos los actores de tipo *BP_PlayerCharacter* y *BP_MasterAI* se llama a su método *Reset.*

Asimismo, hay entidades como el *BP_PlayerHelper* que también deben ser reestablecidas para que el jugador/a pueda volver a hacer uso de ellas. Por último, el actor *BP_PlayerAI* es destruido cuando el personaje principal muere.

9.17.5. Comportamiento cuando el Jefe Final muere

La única entidad que no reaparece ni se reinicia si es derrotada es el jefe final. Esta entidad tras reproducir su animación de muerte es destruida y eliminada del nivel, con lo cual no puede ser reiniciada. Asimismo, antes de ser destruida, crea una hoguera en el sitio donde nace y crea un objeto interactivo. Este objeto interactivo es la espada del jefe final, que si el jugador/a quiere puede interactuar con ella y pasa a ser su espada activa. Además, la entidad *BP_PlayerAI* es destruida.

9.17.6. Mensaje de muerte o victoria

Como es característico en los juegos del estilo *Soulslike*, cuando el personaje principal muere aparece el siguiente mensaje en pantalla: ***"YOU DIED"*** (HAS MUERTO). En cambio, cuando el jugador/a logra derrotar a un jefe final también se le notifica por pantalla con un mensaje de victoria. Este mensaje ha ido variando a lo largo de las diferentes entregas, pero el elegido para este proyecto es ***"ENEMY FELLED"*** (ENEMIGO DERRIBADO). Dado que el comportamiento es el mismo, el autor únicamente crea un *widget*, ***WB_VictoryDefeatMessage***. Este widget es creado desde el *BP_PlayerController* en el momento que el jugador/a muere o derrota al jefe final y en función de la situación que se dé se mostrará como visible un mensaje u otro.

9.18. Descansar en la hoguera

Otra de las mecánicas características de un *Soulslike* es poder descansar en una hoguera. A continuación, se explica cómo se ha implementado la interacción con dicho objeto y las consecuencias que tiene esta acción sobre el juego.

9.18.1. Interacción con el actor *BP_Fireplace*

Para llevar a cabo esta mecánica el autor implementa el actor ***BP_Fireplace***, el cual tiene un comportamiento similar al *blueprint BP_PickupActor*. Este actor está formado por una esfera de colisión para poder detectar la interacción, una malla estática (el objeto hoguera en sí) y un sistema de partículas que simula el fuego de una hoguera. Además, este actor implementa el evento de interfaz ***EventInteract***. En función de si la hoguera está encendida o no el comportamiento que ejecuta sobre el personaje principal es uno u otro.

9.18.1.1. Encender la hoguera

Si la hoguera está apagada, cuando el jugador/a interactúa con ella se producen los siguientes comportamientos:

- El personaje principal reproduce la animación de encendido de hoguera (diferente a la de interacción con otro objeto).
- Se hace visible el sistema de partículas de fuego.
- Se establece como posición de reinicio del jugador/a la posición de la hoguera.
- Aparece en pantalla un mensaje indicando que la hoguera ha sido encendida.

9.18.1.2. Descansar en la hoguera

En cambio, cuando la hoguera ya está encendida se producen comportamientos diferentes:

- El personaje principal reproduce la animación de descansar en la hoguera, quedando en este estado hasta que se vuelve a interactuar con la hoguera.
    - Cuando el jugador/a está descansando en la hoguera no puede realizar ninguna acción de movimiento.
    - Si el jugador/a está descansando y vuelve a interactuar con la hoguera, el personaje reproduce la animación de levantarse.
- Cuando el personaje está descansando se regeneran sus estadísticas y el número de usos de los consumibles.
- Se reinician todos los enemigos, excepto el jefe final si ha sido derrotado.
- Se destruye la entidad *BP_PlayerAI* si ha sido invocada.
- Se reinician los actores *BP_PlayerHelper* si han sido utilizados.

9.18.2. Mensaje al encender la hoguera

De forma similar al mensaje de muerte o de victoria, el autor implementa el widget ***WB_BonfireLitMessage***. Este widget es creado cuando el jugador/a interactúa con una hoguera por primera vez después de reproducir la animación de encenderla. El mensaje que se muestra es ***"BONFIRE LIT"*** (HOGUERA ENCENDIDA) y este desaparece al cabo de 3 segundos de forma automática.

9.19. Atravesar la niebla

Otro de los rasgos característicos de los juegos estilo *Soulslike* es que la puerta de acceso a la pelea contra el jefe final suele estar bloqueada por una niebla. Para poder atravesarla el jugador/a debe interactuar con ella. Esta niebla únicamente se desvanece cuando el jugador/a consigue derrotar al jefe de esa zona.

Para llevar a cabo este comportamiento el autor implementa el actor ***BP_BossFog*.** Es un actor bastante simple y similar al de la hoguera. Está compuesto por una malla estática (un cubo en este caso) que bloquea el paso y otro cubo con las propiedades físicas necesarias para poder recibir el evento de interacción por parte del jugador/a. De este modo, también implementa el evento de interfaz ***Event Interact***, que en este caso hace que el personaje reproduzca la animación de interacción y deshabilite la colisión de la malla de forma temporal para que este pueda atravesar el muro de niebla. Asimismo, en este momento se destruye la entidad *BP_PlayerAI* en el caso de que haya sido invocada anteriormente ya que esta no puede atravesar la niebla. Sin embargo, hay otro objeto de invocación dentro de la zona de pelea contra el jefe final.

Por otro lado, para que el muro desaparezca cuando el jugador/a derrota al jefe final, el autor implementa un método en el *BP_SoulslikeGameMode* (mencionado anteriormente) que busca a todos los actores de tipo *BP_BossFog* y los elimina. En este proyecto, ya que solo va a haber un muro de niebla, esta implementación es suficiente, pero en caso de haber más muros de niebla el autor debería establecer una relación entre el jefe final y el muro de niebla de su zona para destruir únicamente ese muro y no todos.

9.20. Sistema de recompensa: almas

Todos los juegos *Soulslike* tienen especial cuidado con el diseño de recompensas para generar en el jugador/a una sensación de satisfacción cuando avanza. La mayor satisfacción es derrotar al jefe final de la zona, generando la sensación de haber superado un reto mayúsculo. El segundo lugar lo ocupan las hogueras. El jugador/a se siente aliviado cuando encuentra un sitio de resguardo y a la vez siente que avanza ya que encontrar una hoguera indica que el jugador/a llega a una nueva zona de exploración o que hay nuevos retos cerca. Por último, la recompensa más inmediata es recibir puntos (conocidos como almas en la saga *Souls*) y son empleados como moneda del juego. En este proyecto, el personaje obtiene puntos cuando derrota a un enemigo.

Asimismo, los juegos *Soulslike*, también castigan al jugador/a, de forma temporal o de forma definitiva. Uno de los ejemplos se encuentra en los puntos. Cuando el personaje muere pierde de forma temporal todos ellos, sin embargo, estos se quedan almacenados en un objeto interactivo que aparece en la posición en la que personaje ha muerto. Si el jugador/a consigue interactuar con el objeto antes de volver a morir puede recuperar todos los puntos que tenía, pero si muere antes de volver a conseguirlos los pierde definitivamente.

Para lograr este comportamiento, el autor implementa el componente ***BPC_SoulsPointsComponent*** y el actor ***BP_SoulsRestorer***.

9.20.1. BPC_SoulsPointsComponent

Este componente otorga a la entidad la capacidad de almacenar almas cuando mata a un enemigo o de perderlas en caso de morir. Tiene un entero como único atributo y sus métodos se centran en el manejo de esta variable (incrementar su cantidad, decrementarla o restaurarla). Asimismo, este componente es el encargado de crear el actor *BP_SoulsRestorer* cuando la entidad que contiene el componente muere. En este proyecto, únicamente el principal tiene este componente.

Por otro lado, es el propio enemigo derrotado el encargado de llamar a la función de incrementar las almas de este componente. Dado que el actor enemigo almacena una referencia a la entidad objetivo puede llamar al componente de esa entidad y pasarle como parámetro el número de almas que tiene como atributo el propio enemigo. En caso de que el actor que ha derrotado al enemigo sea el ayudante invocado, las almas también deben pasar al jugador/a.

9.20.2. BP_SoulsRestorer

Esta entidad es creada por el ***BPC_SoulsPointsComponent*** cuando el personaje principal muere. Es otro actor interactivo con comportamiento similar al resto anteriormente explicados. Esta entidad tiene como atributo un entero que indica el número de almas que va a sumarle al personaje del jugador/a cunado interactúe con ella. Asimismo, el momento que el personaje interactúa con la entidad, esta es destruida. Como se ha explicado anteriormente, si el jugador/a no llega a interactuar con el restaurador de puntos antes de volver a morir, pierde las almas de forma definitiva. Esto es debido a que el restaurador de almas guarda las almas que el personaje tiene en el momento que muere. Si el personaje muere una primera vez con 10.000 almas, por ejemplo, si de camino a recuperarlas ha conseguido unas 500 pero muere antes de recuperar las 10.000, el restaurador de 10.000 es destruido y se crea uno nuevo que almacena 500 únicamente.

9.20.3. WB_SoulsPoints

Por último, el jugador/a debe saber en todo momento cuántos puntos tiene en su posesión. Para ello el autor crea el *widget* ***WB_SoulsPoints***. Este widget está formado por una caja negra y un texto que indica el número de almas actuales del personaje. Este widget actualiza su texto en el momento que se actualiza el número de almas del componente *BPC_SoulsPointsComponent*, mediante un *Event Dispatcher* (explicados anteriormente). A su vez, este widget es añadido al widget del HUD principal de juego, junto con las barras de estado y los objetos equipados.

9.21. Efectos visuales y sonoros

Llegados a este punto de la implementación, también es importante introducir efectos visuales y sonoros para ofrecer un mayor nivel de acabado. Dado que el número de efectos incluidos en el proyecto es elevado el autor no entra en detalle en explicar todos y cada uno de ellos. En este caso explica el proceso general para reproducirlos y solamente destaca la implementación de aquellos más relevantes.

Muchos de los efectos reproducidos en el juego están relacionados con los ataques. En cuanto a los sonidos, los de movimiento de espada son invocados desde el propio *Anim Montage* mediante *Anim Notifies*. Por otro lado, para las reacciones a los golpes se reproducen desde los propios *blueprints* de las entidades con las funciones de *Unreal Engine Play Sound at Location* o *Play Sound 2D*, en función de si se quiere reproducir un sonido desde una localización específica o no. Asimismo, los sistemas de partículas de golpeo son reproducidos desde el *blueprint* de la entidad ya que dependiendo del resultado del golpeo debe reproducirse un efecto u otro.

Además, los efectos sonoros son reproducidos como *Sound Cues*. Este tipo de *asset de Unreal Engine* permite introducir diversos archivos ***.wav*** (ondas de sonido) y tratarlas directamente desde el editor del motor. En las *Sound Cues* el autor puede añadir un nodo ***Random*** para hacer que queda vez que se reproduzca una *Cue* se reproduzca un sonido diferente de ataque o daño, así como variar los parámetros de la atenuación para sonidos 3D, empleado en las hogueras, por ejemplo. También es posible modificar el volumen de esa *Cue* y la altura.

9.21.1. Anim Notify AN_PlayAuroraFX

A continuación, el autor quiere destacar el caso de los efectos visuales empleados para la pelea contra el jefe final. Para reproducirlos, el autor implementa el *Anim Notify* ***AN_PlayAuroraFX***.

Este *Anim Notify* es el encargado de reproducir los efectos sonoros y visuales de los ataques fuertes del jefe final. Este *Anim Notify*, en función de la fase en la que esté la pelea contra el jefe final (explicado en el apartado **9.15.4. Combate dividido en dos fases**), reproduce uno u otro. Si el jefe está en la primera fase reproduce un efecto visual de choque simple contra rocas y un sonido de golpeo no demasiado brusco. Sin embargo, cuando está en la segunda fase reproduce un efecto visual de explosión de hielo y un sonido más brusco y agresivo. Asimismo, este *Anim Notify* se coloca en el fotograma exacto de la animación en el que la espada del jefe final golpea el suelo.

Por otro lado, cuando la pelea está en la segunda fase, al golpear la espada contra el suelo también se ejerce daño en área. Para ello se emplea el evento ***ApplyRadialDamage***, el cual ejerce daño en área. Todas las entidades que estén dentro del radio de daño reciben la cantidad indicada de daño. Uno de los parámetros de esta función permite indicar si el daño recibido varía en función de la distancia al centro o se recibe el total del daño en todo el rango. En este caso, el autor decide ejercer daño en función a la distancia del centro.

Por último, para añadir un toque de inmersividad, este *Anim Notofy* también es el encargado de reproducir una vibración de la cámara, simulando la sensación del temblor de la tierra cuando una fuerza grande la golpea. Esto se consigue gracias a la función de *Unreal Engine* ***Play World***

Camera Shake. Esta función recibe por parámetro un *blueprint* con los datos que definen el tipo de vibración (desplazamiento, frecuencia, duración, etc.) y el radio de efecto. De este modo, todas las cámaras que se encuentren dentro del radio serán agitadas. En este caso el autor implementa el *blueprint* *BP_CameraShakeBossAttack*, que hereda de la clase *MatineeCameraShake* de *Unreal Engine* [101]. En él el autor define los parámetros que definen la vibración para este tipo de ataque.

9.21.2. Sonido ambiente del nivel y de la pelea contra el jefe final

Para reproducir el sonido ambiente el autor crea una *Cue Sound* formada por el nodo que contiene el archivo *.wav* con el sonido ambiente y le añade un nodo ***Loop***, el cual permite que esta cola se reproduzca en bucle todo el rato que el nivel esté activo. Asimismo, el autor modifica el alcance del sonido para que cuando se acerque a la zona de pelea del jefe final, este sonido no se escuche y no interfiera con la banda sonora de la pelea. Para reproducir esta *Cue*, el autor simplemente la añade como un actor más al nivel desde el editor para que cuando cargue el nivel se reproduzca de forma automática.

Por otro lado, para la banda sonora de la pelea contra el jefe final se implementa una *Cue* similar. Sin embargo, el encargado de almacenar y reproducir la propia *Cue* es el *blueprint* del jefe final. Esta acción se lleva a cabo cuando se establece el objetivo al que tiene que atacar. Del mismo modo, cuando el personaje principal o el jefe mueren, esta *Cue* deja de reproducirse con un efecto de desvanecimiento.

Por último, de forma similar al sonido ambiente, el autor introduce un actor a la pantalla en blanco del menú principal. Esta pista comienza a reproducirse de forma automática cuando el menú principal es cargado y se deja de reproducir cuando se carga el nivel de juego.

10. Pruebas y validación

Una vez desarrollado una demo completa es momento de ponerla a prueba y analizar si el producto es robusto y fiable. Estas pruebas han de satisfacer las necesidades expuestas en el anexo **A.b. Requisitos no funcionales** así como el correcto comportamiento de la aplicación en lo que a flujo de juego, inicio y cierre se refiere. Para realizar estas validaciones el autor procede al empaquetado del proyecto y la generación de un ejecutable listo para ser ejecutado y distribuido.

El primer requisito no funcional está relacionado con la usabilidad del proyecto. Este requisito se valida viendo cómo diferentes personas interactúan con el ejecutable. Deben ser capaces de interactuar con los menús de forma correcta y demostrar que los controles son claros y se entienden para que puedan progresar en el juego.

En cuanto a la fiabilidad, el autor somete a la aplicación a un tiempo de juego prolongado para comprobar que puede jugar sin ninguna interrupción ni fallo inesperado de la aplicación. Asimismo, se comparte el ejecutable con un grupo reducido de usuarios y usuarias para que puedan validar este factor, así como comprobar que todos los elementos relacionados con el flujo de juego y correcto inicio y cierre de la aplicación funcionan de la forma esperada.

En lo referente al espacio en disco, el autor no expone ninguna prueba o test que deba superar el ejecutable más que su propio juicio y justificación del espacio que ocupa el ejecutable en función de las características de este.

Para analizar el rendimiento de la aplicación, el propio editor de *Unreal Engine* ofrece herramientas de análisis y datos de rendimiento. En este caso los indicadores de rendimiento principales son las veces que el bucle de juego se actualiza por segundo, medido en fotogramas por segundo (FPS) y el tiempo que tarda en actualizarse el bucle de juego, medido en milisegundos. Además, el monitor de rendimiento de *Unreal Engine* también permite analizar el tiempo de proceso que consume cada sistema, así como la cantidad de memoria ocupada y demás datos relevantes. Por otro lado, *Unreal Engine* permite la visualización del *viewport* de juego a través de diferentes modos, los cuales permiten al autor ver si hay demasiados polígonos en la escena o qué cantidad de procesamiento están consumiendo los diferentes sistemas de partículas y materiales entre otras funcionalidades.

Por último, en cuanto al nivel de fidelidad conseguido en este proyecto en comparación a los videojuegos desarrollados por *FromSoftware* es una cuestión subjetiva que debe evaluar cada persona que pruebe este videojuego.

11. Resultados

11.1. Resultados de las pruebas de validación

En cuanto a la usabilidad de la aplicación, dados los plazos de entrega y las características del proyecto el autor no ha podido validar este requisito de forma exhaustiva. Es por ello que, desde la fecha entrega del trabajo de fin de grado hasta la fecha defensa del mismo, el autor se propone recaudar información acerca de la experiencia de los usuarios y usuarias que prueben el videojuego con el fin de validar este requisito, así como el *'Game feel'* y la experiencia de juego en general.

La fiabilidad del ejecutable sí que se ha podido probar tanto en el equipo del autor como en otros equipos. El ejecutable es fiable ya que en todos los casos ha funcionado sin ninguna interrupción y no se ha producido ningún comportamiento inesperado. Sin embargo, hay casos puntuales en los que, debido del software instalado en la máquina, el ejecutable le indica al usuario que es necesario que instale la librería *XAudio 2.7* incluida con *DirectX*.

En lo referente al espacio en disco, 3.87GB, el autor considera que es adecuado para las características de los *assets* empleados en el videojuego. La calidad de los modelos, texturas y efectos es alta, de modo que es normal que el tamaño del ejecutable sea medianamente elevado para la cantidad de juego que se muestra. Una posible optimización sería reducir el tamaño de texturas, ofreciendo un nivel de acabado más bajo, pero a su vez ganando en espacio en disco al almacenar el proyecto y espacio en memoria al cargar el ejecutable.

En cuanto al rendimiento, la situación ideal para el autor es que el bucle de juego tarde aproximadamente 16ms para tener una tasa de refresco de 60FPS. En el hardware del equipo del autor, el bucle de juego tarda aproximadamente entre 40 y 44ms con lo que el juego se ejecuta con una frecuencia de 23FPS. Estos resultados son los obtenidos en la zona con más cantidad de información para procesar de todo el nivel. Sin embargo, en el espacio de combate contra el jefe final se alcanzan los 40FPS y una velocidad de actualización del bucle de 20 a 25ms. Los dos factores que más influyen en el tiempo de proceso del ejecutable son la proyección de sombras realistas y las iteraciones de los árboles de comportamiento de la IA. Tras este análisis, el autor decide hacer las siguientes optimizaciones:

- Mejorar el rendimiento de los árboles de comportamiento de la IA haciendo que únicamente se actualicen los árboles de los enemigos activos.

    - El jefe final únicamente se activa cuando el jugador/a entra en su zona de pelea.
    - Los enemigos simples dejan de actualizar el árbol de comportamiento en el momento que son derrotados.
- Establecer una distancia máxima de renderizado de las luces para que solo consuman tiempo de proceso e iluminen la escena cuando el jugador/a está lo suficientemente cerca de ellas.
- Eliminar las sombras proyectadas de las luces de las fogatas y hacer que únicamente la iluminación global (el sol de la escena) proyecte sombras, tanto dinámicas como estáticas.
- Reducir el número de partículas que emiten los sistemas de partículas de efecto nieve y fuego.
- Reducir la tasa de refresco de los sistemas de partículas.
- Establecer una distancia máxima de renderizado de los sistemas de partículas para que solo se muestren por pantalla cuando el personaje principal está lo suficientemente cerca.

11.2. Revisión del análisis de costes

Al igual que con el proyecto de ABP, este proyecto no es un producto final listo para su puesta a la venta en las plataformas estudiadas. Es un ejecutable de demostración completo con todas las mecánicas y que sirve como previsualización de lo que puede llegar a ser el juego final en un futuro. Es por ello por lo que el análisis de costes detallado en el apartado 2.3. Estudio de mercado no está del todo completo. Los costes estimados son los necesarios para recuperar la inversión del trabajo hecho hasta la fecha. Sin embargo, para poder sacar este producto al mercado es necesario:

- Tiempo de pulido de mecánicas, menús, interfaz y flujo de juego.
- Añadir más contenido para tener un juego de al menos dos horas y media de duración (Evitando así que el consumidor/a pueda devolver el producto una vez haya acabado de jugarlo).
- Establecer una campaña de marketing y publicación.

De este modo, el autor estima que solo con el tiempo necesario para el pulido y la creación de nuevo contenido, el coste de producción puede llegar incluso a triplicarse. Asimismo, la inversión de marketing suele ser propuesta por el *Publisher* que quiere llevar a cabo la publicación y distribución del videojuego.

12. Conclusiones y trabajo futuro

Para concluir, el autor expone las sensaciones que ha experimentado a lo largo del desarrollo de este proyecto, así como analizar los objetivos cumplidos y el trabajo pendiente.

En cuanto al resultado final del trabajo, el autor se muestra orgulloso del producto obtenido teniendo en cuenta el tiempo limitado de desarrollo y el desconocimiento inicial del motor. Obviamente no puede conseguir un videojuego de la calidad y el acabado de empresas multimillonarias con una numerosa plantilla y con años y años de experiencia, pero personalmente cree que ha conseguido plasmar todo aquello que considera imprescindible de los juegos *Soulslike.*

Asimismo, el autor se siente muy satisfecho con todo lo aprendido a lo largo del desarrollo del proyecto y del camino que ha recorrido. Durante el desarrollo el autor se ha enfrentado a una gran cantidad de retos y desafíos nuevos relacionados con la programación de videojuegos en el motor *Unreal Engine*. El hecho de trabajar para superar estos retos ha hecho que aprenda y descubra nuevos horizontes por explorar y en los que trabajar, sobre todo en el campo de la Inteligencia Artificial en videojuegos y el diseño de mecánicas de combate para videojuegos de este estilo.

Por otro lado, el autor ha cumplido todos los objetivos propuestos de forma satisfactoria. Sin embargo, como en todo proyecto, siempre queda algo en el tintero. Es por ello que el autor se propone como trabajo futuro implementar un menú de ajustes de vídeo y sonido, tal y como se detalla en la especificación de requisitos. Además, también queda pendiente el sistema de incremento de niveles y estadísticas del personaje principal, incremento de uso de pociones y un inventario completo de armas y objetos consumibles. En cuanto al diseño del comportamiento del jefe final, el autor se propone modificar el diseño de este para ampliar la capacidad de toma de decisiones para los ataques, estudiando las técnicas empleadas en los títulos que han inspirado este proyecto y que por falta de tiempo no se han podido integrar aquí.

En último lugar, este proyecto se planteó para ser programado en C++, sin embargo, el lenguaje de C++ de *Unreal Engine* es un lenguaje completamente nuevo y con una alta curva de aprendizaje. Debido al tiempo limitado del proyecto el autor decidió usar *blueprints*. Pero ahora, con el fin de aprender C++ con *Unreal Engine* el autor se propone pasar este proyecto a este lenguaje de programación y prescindir de los *blueprints* en todo aquello que sea posible.

Referencias

[1] P. GALIANA, «iebschool,» 27 Abril 2021. [En línea]. Available: https://www.iebschool.com/blog/que-es-esports-marketing-digital/.

[2] E-SPORT.tech, «E-SPORT.tech,» [En línea]. Available: https://e-sport.tech/glosario-gamer/que-significa-gameplay/.

[3] SoloEmpleo, «https://www.soloempleo.com/videojuegos-generador-de-empleo-cultura-y-riqueza,» 2018. [En línea]. Available: https://www.soloempleo.com/videojuegos-generador-de-empleo-cultura-y-riqueza. [Último acceso: 2021].

[4] Jobted, «Jobted,» [En línea]. Available: https://www.jobted.es/salario/programador-videojuegos . [Último acceso: 2021].

[5] PCComponentes, «PCComponentes,» [En línea]. Available: https://www.pccomponentes.com/pccom-gold-amd-ryzen-7-5800x-32gb-1tbssd-2tb-rtx3070ti. [Último acceso: 30 Julio 2021].

[6] codigi, «codigi,» [En línea]. Available: https://codigi.es/producto/windows-10-professional/. [Último acceso: 30 Julio 2021].

[7] PCComponentes, «PCComponentes,» [En línea]. Available: https://www.pccomponentes.com/msi-pro-mp271qp-27-led-ips-wqhd. [Último acceso: 30 Julio 2021].

[8] PCComponentes, «PCComponentes,» [En línea]. Available: https://www.pccomponentes.com/mars-gaming-mkxtkl-teclado-mecanico-gaming-rgb-switch-azul. [Último acceso: 30 Julio 2021].

[9] PCComponentes, «PCComponentes,» [En línea]. Available: https://www.pccomponentes.com/tempest-ms100-paladin-raton-gaming-1600dpi. [Último acceso: 30 Julio 2021].

[10] U. Engine, «Unreal Engine,» [En línea]. Available: https://www.unrealengine.com/. [Último acceso: 2021].

[11] Unity, «Unity,» [En línea]. Available: https://unity.com/es. [Último acceso: 2021].

[12] U. Engine, «Unreal Engine FAQ,» [En línea]. Available: https://www.unrealengine.com/ja/faq. [Último acceso: Julio 2021].

[13] U. Engine, «Unreal Engine Releasing Products,» [En línea]. Available: https://www.unrealengine.com/ja/faq?active=releasing-products. [Último acceso: Julio 2021].

[14] Unity, «Unity Software Additional Terms,» [En línea]. Available: https://unity3d.com/legal/terms-of-service/software. [Último acceso: Julio 2021].

[15] Unity, «Unity Compare Plans,» [En línea]. Available: https://store.unity.com/es/compare-plans. [Último acceso: Julio 2021].

[16] U. Engine, «Unreal Engine Publish,» [En línea]. Available: https://www.epicgames.com/store/en-US/publish. [Último acceso: Julio 2021].

[17] Steam, «Steam FAQ,» [En línea]. Available: https://partner.steamgames.com/doc/gettingstarted/faq. [Último acceso: Julio 2021].

[18] xsolla, «xsolla publish on Steam,» [En línea]. Available: https://xsolla.com/blog/self-publish-on-steam-the-ultimate-guide. [Último acceso: Julio 2021].

[19] R. A. Española, «Real Academia Española,» [En línea]. Available: https://dle.rae.es/videojuego?m=form. [Último acceso: 2021].

[20] Diccionario.De, «Diccionario.De,» [En línea]. Available: https://definicion.de/videojuego/. [Último acceso: 2021].

[21] D. Muriel, «El videojuego como experiencia,» *Caracteres. Estudios culturales y críticos de la esfera digital,* 2018.

[22] Microsoft, «Microsoft,» Xbox Game Studios, 2020. [En línea]. Available: https://www.microsoft.com/es-es/p/microsoft-flight-simulator-standard/9nxn8gf8n9ht?activetab=pivot:overviewtab. [Último acceso: 2021].

[23] J. T. García, «GamerDic,» [En línea]. Available: https://www.gamerdic.es/termino/arcade. [Último acceso: 2021].

[24] J. T. Garcia, «GamerDic,» [En línea]. Available: https://www.gamerdic.es/termino/plataformas/. [Último acceso: 2021].

[25] MeriStation, «MeriStation,» [En línea]. Available: https://as.com/meristation/juegos/top/videojuegos_plataformas/. [Último acceso: 2021].

[26] J. T. Garcia, «GamerDic,» [En línea]. Available: https://www.gamerdic.es/termino/accion/. [Último acceso: 2021].

[27] J. T. García, «GamerDic,» [En línea]. Available: https://www.gamerdic.es/termino/lucha/. [Último acceso: 2021].

[28] J. T. García, «GamerDic,» [En línea]. Available: https://www.gamerdic.es/termino/shooter. [Último acceso: 2021].

[29] Vandal, «Vandal,» [En línea]. Available: https://vandal.elespanol.com/rankings/pc/aventura. [Último acceso: 2021].

[30] J. T. García, «GamerDic,» [En línea]. Available: https://www.gamerdic.es/termino/survival-horror/. [Último acceso: 2021].

[31] EcuRed, «EcuRed,» [En línea]. Available: https://www.ecured.cu/Videojuego_de_rol. [Último acceso: 2021].

[32] J. T. García, «GamerDic,» [En línea]. Available: https://www.gamerdic.es/termino/musical/. [Último acceso: 2021].

[33] J. T. García, «GamerDic,» [En línea]. Available: https://www.gamerdic.es/termino/baile. [Último acceso: 2021].

[34] J. Penalva, «Xataka,» 2 Noviembre 2010. [En línea]. Available: https://www.xataka.com/videojuegos/kinect-para-xbox-360-jugar-sin-mando. [Último acceso: 2021].

[35] D. Moro, «Mersitation,» 19 Diciembre 2020. [En línea]. Available: https://as.com/meristation/2020/12/19/reportajes/1608372147_471449.html. [Último acceso: 2021].

[36] R. Márquez, «Xataka,» 31 Agosto 2019. [En línea]. Available: https://www.xataka.com/videojuegos/asi-hemos-pasado-ciencia-detras-simcity-a-campeonatos-tractores-virtuales. [Último acceso: 2021].

[37] S. Software, «Euro Truck Simulator 2,» SCS Software, 2012. [En línea]. Available: https://eurotrucksimulator2.com/. [Último acceso: 2021].

[38] E. Arts, «EA,» Electronic Arts, 2014. [En línea]. Available: https://www.ea.com/es-es/games/the-sims/the-sims-4/pc/store/mac-pc-download-base-game-standard-edition. [Último acceso: 2021].

[39] J. T. García, «GamerDic,» [En línea]. Available: https://www.gamerdic.es/termino/jrpg/. [Último acceso: 2021].

[40] tvtropes, «tvtropes,» 2021. [En línea]. Available: https://tvtropes.org/pmwiki/pmwiki.php/Main/WesternRPG. [Último acceso: 2021].

[41] J. T. García, «GamerDic,» [En línea]. Available: https://www.gamerdic.es/termino/mmorpg/. [Último acceso: 2021].

[42] Blizzard, «World Of Warcraft,» Blizzard, [En línea]. Available: https://worldofwarcraft.com/es-es/. [Último acceso: 2021].

[43] J. T. García, «GamerDic,» [En línea]. Available: https://www.gamerdic.es/termino/lore/. [Último acceso: 2021].

[44] J. T. García, «GamerDic,» [En línea]. Available: https://www.gamerdic.es/termino/raid. [Último acceso: 2021].

[45] J. T. García, «GamerDic,» [En línea]. Available: https://www.gamerdic.es/termino/rpg-de-accian/. [Último acceso: 2021].

[46] Ó. Bouzo, «Vidaextra,» 19 Octubre 2020. [En línea]. Available: https://www.vidaextra.com/analisis/demons-souls-analisis-review-video-experiencia-juego-para-ps3-playstation-3. [Último acceso: 2021].

[47] FromSoftware, «FromSoftware,» [En línea]. Available: https://www.fromsoftware.jp/ww/. [Último acceso: 2021].

[48] C. Carvajal, «rewisor,» 2 Octubre 2020. [En línea]. Available: rewisor.com/quien-es-hidetaka-miyazaki/. [Último acceso: 2021].

[49] D. Heaney, «Square Enix Games,» 18 Octubre 2019. [En línea]. Available: https://square-enix-games.com/es_ES/news/classic-dragon-quest-switch. [Último acceso: 2021].

[50] J. Tones, «Xataka,» 20 Mayo 2021. [En línea]. Available: https://www.xataka.com/literatura-comics-y-juegos/muere-kentaro-miura-autor-berserk-obra-clave-manga-ciencia-ficcion-que-queda-definitivamente-incompleta. [Último acceso: 2021].

[51] L. -. Japonismo, «Japonismo,» 25 Febrero 2014. [En línea]. Available: https://japonismo.com/blog/el-concepto-de-mono-no-aware. [Último acceso: 2021].

[52] W. D. Souls, «Wiki Dark Souls,» [En línea]. Available: https://darksouls.fandom.com/es/wiki/Frasco_Estus. [Último acceso: 2021].

[53] W. D. Souls, «Wiki Dark Souls,» [En línea]. Available: https://darksouls.fandom.com/es/wiki/Hoguera. [Último acceso: 2021].

[54] W. D. Souls, «Wiki Dark Souls,» [En línea]. Available: https://darksouls.fandom.com/es/wiki/Jefes_de_Dark_Souls. [Último acceso: 2021].

[55] PlayStation, «PlayStation,» [En línea]. Available: https://www.playstation.com/es-es/games/bloodborne/. [Último acceso: 2021].

[56] R. Ramsay, «gamespot,» 10 June 2014. [En línea]. Available: https://www.gamespot.com/articles/e3-2014-the-similarities-and-differences-between-bloodborne-and-dark-souls/1100-6420378/. [Último acceso: 2021].

[57] FromSoftware, «Sekirothegame,» [En línea]. Available: https://www.sekirothegame.com/es/home. [Último acceso: 2021].

[58] FromSoftware, «BandaiNamco,» 2021. [En línea]. Available: https://es.bandainamcoent.eu/elden-ring/elden-ring?gclid=Cj0KCQjw0emHBhC1ARIsAL1QGNcQz-

OAXrlJTtYf0NGYI__ZyQ4TipPNz9_xGYfNx9YZf3El9g5S-vIaAiN7EALw_wcB#news. [Último acceso: 2021].

[59] BandaiNamco, «BandaiNamco,» [En línea]. Available: https://es.bandainamcoent.eu/. [Último acceso: 2021].

[60] HieloYFuegoFandom, «hieloyfuego.fandom,» [En línea]. Available: https://hieloyfuego.fandom.com/wiki/Canci%C3%B3n_de_Hielo_y_Fuego. [Último acceso: 2021].

[61] Kanbanize, «Kanbanize,» [En línea]. Available: https://kanbanize.com/es/recursos-de-kanban/primeros-pasos/que-es-kanban. [Último acceso: 2021].

[62] C. DRUMOND, «Atlassian,» [En línea]. Available: https://www.atlassian.com/es/agile/scrum. [Último acceso: 2021].

[63] GitLab, «GitLab,» [En línea]. Available: https://about.gitlab.com/stages-devops-lifecycle/issueboard/. [Último acceso: 2021].

[64] Toggle, «Toggle,» [En línea]. Available: https://track.toggl.com/. [Último acceso: 2021].

[65] Clockify, «Clockify,» [En línea]. Available: https://app.clockify.me/tracker. [Último acceso: 2022].

[66] GitLab, «GitLab,» [En línea]. Available: https://about.gitlab.com/. [Último acceso: 2021].

[67] M. Fowler, UML gota a gota, 1999.

[68] U. Engine, «Unreal Engine 4.27 release,» [En línea]. Available: https://www.unrealengine.com/ja/blog/unreal-engine-4-27-released. [Último acceso: 2021].

[69] U. Engine, «Unreal Engine 5,» [En línea]. Available: https://www.unrealengine.com/ja/unreal-engine-5. [Último acceso: 2021].

[70] U. Engine, «Unreal Engine Blueprint,» [En línea]. Available: https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/. [Último acceso: 2021].

[71] Unity, «Unity Documentation,» [En línea]. Available: https://docs.unity3d.com/es/2018.4/Manual/SL-Shader.html. [Último acceso: 2021].

[72] Mixamo, «Mixamo,» [En línea]. Available: https://www.mixamo.com/#/. [Último acceso: 2021].

[73] E. Games, «UnrealVS Extension,» [En línea]. Available: https://docs.unrealengine.com/5.0/en-US/using-the-unrealvs-extension-for-unreal-engine-cplusplus-projects/. [Último acceso: 2022].

[74] JetBrains, «Rider for Unreal Engine,» [En línea]. Available: https://www.jetbrains.com/lp/rider-unreal/?source=google&medium=cpc&campaign=16134832534&term=unreal%20engine%205&gclid=Cj0KCQjw_viWBhD8ARIsAH1mCd6W6VIJLvs6zEmsr065HqO1VaECpAeQ49a82JB0pyappW_jZlmVNFwaApF_EALw_wcB. [Último acceso: 2022].

[75] Freesound, «Freesound,» [En línea]. Available: https://freesound.org/help/about/. [Último acceso: 2021].

[76] C. Commons, «Creative Commons,» [En línea]. Available: https://creativecommons.org/. [Último acceso: 2021].

[77] Soniss, «GameAudioGDC,» [En línea]. Available: https://sonniss.com/gameaudiogdc. [Último acceso: 2021].

[78] U. Engine, «Unreal Engine Forum,» [En línea]. Available: https://forums.unrealengine.com/t/design-paradigm-composition-vs-inheritance-vs-interface-in-unreal-with-example-need-help/96423/2. [Último acceso: Octubre 2022].

[79] U. Engine, «Componentes en Unreal Engine,» [En línea]. Available: https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/Actors/Components/. [Último acceso: Octubre 2022].

[80] U. Engine, «Interfaces en Unreal Engine,» [En línea]. Available: https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/GameplayArchitecture/Interfaces/ . [Último acceso: Octubre 2022].

[81] U. Engine, «Blueprints en Unreal Engine,» [En línea]. Available: https://docs.unrealengine.com/5.0/en-US/blueprints-visual-scripting-in-unreal-engine/. [Último acceso: Octubre 2022].

[82] U. Engine, «Balance entre Blueprints y C++,» [En línea]. Available: https://docs.unrealengine.com/4.27/en-US/Resources/SampleGames/ARPG/BalancingBlueprintAndCPP/. [Último acceso: Octubre 2022].

[83] U. Engine, «Animation Blueprints,» [En línea]. Available: https://docs.unrealengine.com/5.0/en-US/animation-blueprints-in-unreal-engine/. [Último acceso: Octubre 2022].

[84] U. Engine, «Grafos en los Animation Blueprints,» [En línea]. Available: https://docs.unrealengine.com/5.0/en-US/graphing-in-animation-blueprints-in-unreal-engine/. [Último acceso: Octubre 2022].

[85] U. Engine, «IK Rig en Unreal Engine,» [En línea]. Available: https://docs.unrealengine.com/5.0/en-US/ik-rig-in-unreal-engine/. [Último acceso: Octubre 2022].

[86] U. Engine, «Retarget de animaciones en Unreal Engine,» [En línea]. Available: https://docs.unrealengine.com/5.0/en-US/ik-rig-animation-retargeting-in-unreal-engine/. [Último acceso: Octubre 2022].

[87] U. Engine, «Root Motion,» [En línea]. Available: https://docs.unrealengine.com/4.27/en-US/AnimatingObjects/SkeletalMeshAnimation/RootMotion/#:~:text=of%20Root%20Motion.-,What%20Is%20Root%20Motion%3F,the%20character's%20root%20remains%20stationary.. [Último acceso: Octubre 2022].

[88] N. Zuo, «Use SaveCachedPose inside State Machine in UE4,» [En línea]. Available: https://arrowinmyknee.com/2020/07/30/compatibility-of-savecachedpose-inside-state-machine/. [Último acceso: Octubre 2022].

[89] U. Engine, «Satate Machines,» [En línea]. Available: https://docs.unrealengine.com/5.0/en-US/state-machines-in-unreal-engine/. [Último acceso: Octubre 2022].

[90] U. Engine, «Skeletal Mesh Sockets,» [En línea]. Available: https://docs.unrealengine.com/5.0/en-US/skeletal-mesh-sockets-in-unreal-engine/. [Último acceso: Octubre 2022].

[91] U. Engine, «Animation Notifies,» [En línea]. Available: https://docs.unrealengine.com/5.0/en-US/animation-notifies-in-unreal-engine/. [Último acceso: Octubre 2022].

[92] U. Engine, «Blend nodes,» [En línea]. Available: https://docs.unrealengine.com/4.27/en-US/AnimatingObjects/SkeletalMeshAnimation/NodeReference/Blend/. [Último acceso: Octubre 2022].

[93] U. Engine, «Widget Blueprints,» [En línea]. Available: https://docs.unrealengine.com/5.0/en-US/widget-blueprints-in-umg-for-unreal-engine/. [Último acceso: Octubre 2022].

[94] U. Engine, «Event Dispatchers,» [En línea]. Available: https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/UserGuide/EventDispatcher/. [Último acceso: Noviembre 2022].

[95] A. Touring, «Cmoputer Machinery and Intelligence,» 1950. [En línea]. Available: https://redirect.cs.umbc.edu/courses/471/papers/turing.pdf. [Último acceso: Noviembre 2022].

[96] IMB, «Artificial Intelligence (AI),» [En línea]. Available: https://www.ibm.com/cloud/learn/what-is-artificial-intelligence. [Último acceso: Noviembre 2022].

[97] ARM, «AI in Gaming,» [En línea]. Available: https://www.arm.com/glossary/ai-in-gaming#:~:text=AI%20in%20gaming%20refers%20to,behavior%20in%20the%20game%20world.. [Último acceso: Noviembre 2022].

[98] U. Engine, «AIPerception,» [En línea]. Available: https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/ArtificialIntelligence/AIPerception/. [Último acceso: Noviembre 2022].

[99] U. Engine, «PathfollowingComponent,» [En línea]. Available: https://docs.unrealengine.com/5.0/en-US/API/Runtime/AIModule/Navigation/UPathFollowingComponent/. [Último acceso: Noviembre 2022].

[100] U. Engine, «GameMode and GameState,» [En línea]. Available: https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Framework/GameMode/. [Último acceso: Noviembre 2022].

[101] U. Engine, «Camera Shakes,» [En línea]. Available: https://docs.unrealengine.com/4.26/en-US/BlueprintAPI/CameraShakes/. [Último acceso: Diciembre 2022].

[102] I. Sommerville, Ingenería del Software. Séptima edición, 2005.

[103] R. Blog, «Medium,» 20 Abril 2018. [En línea]. Available: https://medium.com/@requeridosblog/requerimientos-funcionales-y-no-funcionales-ejemplos-y-tips-aa31cb59b22a. [Último acceso: Agosto 2021].

[104] PEGI, «PEGI,» [En línea]. Available: https://pegi.info/es/node/19. [Último acceso: 2021].

[105] U. Engine, «Animation Retargeting,» [En línea]. Available: https://docs.unrealengine.com/4.26/en-US/AnimatingObjects/SkeletalMeshAnimation/AnimationRetargeting/. [Último acceso: 2021].

Anexos

A. Especificación de requisitos detallada

Un requisito es una característica que el sistema debe tener para ser aceptado o algo que el sistema debe hacer para el usuario. Los requisitos **surgen de las necesidades del cliente**, es por ello que un requisito describe una utilidad para el usuario. También, por este motivo, hay que pensar detalladamente en lo que se quiere hacer para hacerlo de forma correcta, de modo que se pueda producir beneficio.

Una vez estudiados los casos de uso se procede a listar los requisitos que debe cumplir el proyecto. Cada caso de uso está identificado por un identificador único y a partir de cada caso de uso se derivan uno o más requisitos. Estos también están identificados por un identificador único, a la vez que se distribuyen entre requisitos **funcionales** y **no funcionales**.

Antes de listar los requisitos se presentan la Tabla 7 y la Tabla 8 donde se muestra cómo se especifican tanto los casos de uso como los requisitos.

Tabla 7: Información sobre la clasificación de casos de uso

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	Identificador único por el cual se distingue el caso de uso. Su formato es: **CU_ID**.
Nombre del caso de uso	Nombre descriptivo del caso de uso.
Actor/actores	Actor o actores que intervienen en el caso de uso.
Descripción / Justificación	Breve descripción o justificación de las acciones que se realizan en el caso de uso.
Precondiciones	Condiciones previas que deben cumplirse para la ejecución del caso de uso.
Flujo normal	Curso normal por el que el caso de uso realiza sus acciones.
Flujo alternativo	Curso alternativo al normal que ocurre dentro del caso de uso porque no se puede ejecutar el flujo normal.
Postcondiciones	Condiciones que deben cumplirse para cerrar de forma correcta la ejecución del caso de uso.

Tabla 8: Información sobre la clasificación de los requisitos

PROPIEDAD	DESCRIPCIÓN

Identificador del requisito	Identificador único por el cual se distingue el requisito. Su formato es: **RF(NF)_B(A)_ID**. F o NF corresponden a Funcional o No Funcional y B o A corresponden a Básico o Avanzado.
Nombre del requisito	Nombre descriptivo del requisito.
Tipo	Identifica si es un requisito o una restricción.
Fuente del requisito	Evento que dispara la ejecución del requisito.
Prioridad del requisito	La prioridad de un requisito puede ser: Alta/Necesidad, Media/Deseado o Baja/Opcional.
Descripción del requisito	Representación del comportamiento que debe cumplir el requisito.
Datos de entrada	Datos que el requisito requiere para ser ejecutado.
Datos de salida	Datos que produce el requisito en su ejecución.
Requisito dependiente	Lista de identificadores de requisitos necesarios para que el requisito actual pueda ser ejecutado. En caso de que la lista sea muy extensa únicamente se muestran los requisitos de los que dependa directamente.

a. Requisitos funcionales

Los requisitos **funcionales** son aquellos que responden a una exigencia o estimulo del usuario, es decir, describen una interacción entre el sistema y su ambiente. Especifican una función que el sistema o componente de un sistema debe ser capaz de llevar a cabo [100].

- **Caso de uso nº 1.**
  - Caso de uso

Tabla 9: Descripción CU_01

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_01.**
Nombre del caso de uso	Iniciar juego
Actor/actores	Jugador/a
Descripción / Justificación	El jugador/a debe iniciar el programa que ejecuta el juego.
Precondiciones	Tener una versión ejecutable del juego.
Flujo normal	1. El jugador/a hace doble clic sobre el ejecutable. 2. El juego se ejecuta de forma correcta. 3. Se visualiza el menú principal.

Flujo alternativo	No existe flujo alternativo. En caso de fallo el jugador/a no puede acceder al juego.
Postcondiciones	Se visualiza el menú principal del juego.

- Requisitos

Tabla 10: Descripción RF_B_01

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_01**
Nombre del requisito	Iniciar juego
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	El juego deber poder ser compilado y ejecutado sin ningún problema para que se pueda iniciar el juego.
Datos de entrada	No se requiere ningún dato de entrada específico.
Datos de salida	Menú principal del juego.
Requisito dependiente	No depende de ningún requisito.

- **Caso de uso nº 2.**
    - Caso de uso

Tabla 11: Descripción CU_02

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_02**
Nombre del caso de uso	Iniciar menú principal
Actor/actores	Jugador/a
Descripción / Justificación	Mostrar en pantalla el menú principal y sus opciones.
Precondiciones	Haber iniciado el juego.
Flujo normal	1. El juego se ha iniciado correctamente. 2. Se muestran los apartados del menú principal. 3. El jugador/a puede seleccionar uno de los apartados. 4. El apartado seleccionado se ejecuta correctamente.
Flujo alternativo	No existe flujo alternativo. En caso de fallo el jugador/a no puede acceder al juego.

Postcondiciones	El menú se visualiza y sus opciones son seleccionables.

- Requisitos

Tabla 12: Descripción RF_B_02

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_02**
Nombre del requisito	Mostrar el menú principal
Tipo	Requisito
Fuente del requisito	RF_B_01
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	Se muestra la escena del menú principal con sus opciones después de haber iniciado el juego.
Datos de entrada	No se requiere ningún dato de entrada específico.
Datos de salida	Menú principal del juego.
Requisito dependiente	• RF_B_01

- **Caso de uso nº 3.**
  - Caso de uso

Tabla 13: Descripción CU_03

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_03**
Nombre del caso de uso	Cerrar juego
Actor/actores	Jugador/a
Descripción / Justificación	El jugador/a debe ser capaz de cerrar y dejar de ejecutar el juego.
Precondiciones	El jugador/a debe estar en el menú principal o en el menú de pausa.
Flujo normal	1. El jugador/a se encuentra en el menú principal o pausa. 2. Selecciona la opción de “Quit Game”. 3. Se detiene la ejecución del juego y este se cierra.

Flujo alternativo	1. Si no puede acceder al menú principal o no puede seleccionar la opción de "Quit Game" se puede cerrar directamente con el atajo de teclado "Alt+F4"
Postcondiciones	El juego se cierra y deja de ejecutarse correctamente.

- Requisitos

Tabla 14: Descripción RF_B_03

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_03**
Nombre del requisito	Cerrar el juego
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	Estando en la escena del menú principal o de pausa el jugador/a debe poder hacer clic sobre la opción de salir. Al hacer clic el juego debe dejar de ejecutarse y cerrarse correctamente.
Datos de entrada	Evento de clic izquierdo del ratón o el botón correspondiente del mando.
Datos de salida	Se deja de ejecutar el juego y se cierra.
Requisito dependiente	• RF_B_01 • RF_B_02

- **Caso de uso nº 4.**
  - Caso de uso

Tabla 15: Descripción CU_04

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_04**
Nombre del caso de uso	Acceder opciones
Actor/actores	Jugador/a
Descripción / Justificación	El jugador/a puede acceder a una pantalla de ajustes de vídeo y audio.
Precondiciones	El jugador/a debe estar en el menú principal.

Flujo normal	1. El jugador/a hace clic en el botón de “Options”. 2. El jugador/a decide entre el botón de “Video” o “Audio” para modificar. 3. Se carga la pantalla de propiedades seleccionada por el jugador/a.
Flujo alternativo	1. El jugador/a hace clic en el botón de “Options”. 2. En caso de que no se quiera modificar la opción seleccionada se debe pulsar el botón de “Back”. 3. Si no carga la pantalla de “Options” se debe reiniciar el juego y volver a acceder.
Postcondiciones	Se muestran las propiedades de “Video” y “Audio” para que sean modificadas.

- Requisitos

Tabla 16: Descripción RF_A_01

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_A_01**
Nombre del requisito	Mostrar menú de opciones de audio y vídeo
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Baja/Opcional
Descripción del requisito	Al hacer clic o seleccionar el elemento de “Options” con el mando se debe mostrar la pantalla con las opciones para acceder a los ajustes de “Video” y “Audio”.
Datos de entrada	Evento de clic izquierdo del ratón o el botón correspondiente del mando.
Datos de salida	Pantalla para seleccionar los ajustes de “Video” y “Audio”.
Requisito dependiente	• RF_B_01 • RF_B_02

- **Caso de uso nº 5.**
  - Caso de uso

Tabla 17: Descripción CU_05

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_05**
Nombre del caso de uso	Acceder configuración vídeo
Actor/actores	Jugador/a
Descripción / Justificación	El jugador/a debe poder ver los ajustes de vídeo actuales.
Precondiciones	Estar en la pantalla de "Options".
Flujo normal	1. El jugador/a accede a la pantalla de "Video" desde la pantalla "Options". 2. Se muestra la configuración actual de los ajustes de "Video".
Flujo alternativo	1. El jugador/a accede a la pantalla de "Video" desde la pantalla "Options". 2. Si no carga la pantalla de "Video" se debe reiniciar el juego y volver a acceder. 3. En caso de que no se quiera modificar la opción seleccionada se debe pulsar el botón de "Back".
Postcondiciones	Se muestran las propiedades actuales de vídeo para que el jugador/a pueda modificarlas.

- Requisitos

Tabla 18: Descripción RF_A_02

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_A_02**
Nombre del requisito	Mostrar configuración vídeo
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Baja/Opcional
Descripción del requisito	Al hacer clic en la opción de "Video" del menú de opciones se deben mostrar los valores actuales de la configuración de vídeo.
Datos de entrada	Evento de clic izquierdo del ratón o el botón correspondiente del mando.
Datos de salida	Propiedades actuales de la configuración de vídeo.

Requisito dependiente	• RF_B_01 • RF_B_02 • RF_A_01

- **Caso de uso nº 6.**
    - Caso de uso

Tabla 19: Descripción CU_06

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_06**
Nombre del caso de uso	Modificar configuración de vídeo
Actor/actores	Jugador/a
Descripción / Justificación	El jugador/a debe poder cambiar los ajustes de vídeo para que sean los más adecuados para su máquina.
Precondiciones	Estar en la pantalla de opciones de “Video”.
Flujo normal	1. El jugador/a accede a la pantalla de “Video” tras seleccionarla en la pantalla “Options”. 2. El jugador/a modifica las opciones que desee haciendo uso de los controles disponibles. 3. Se guardan las opciones modificadas. 4. Se pulsa el botón “Back” para volver a la pantalla anterior y que se guarden los nuevos valores.
Flujo alternativo	1. El jugador/a accede a la pantalla de “Video” tras seleccionarla en la pantalla “Options”. 2. Si no puede modificar ninguna opción se debe reiniciar el juego y volver a acceder. 3. En caso de que no se quieran modificar los ajustes de vídeo se debe pulsar el botón de “Back” sin cambiar ningún valor.
Postcondiciones	Los nuevos ajustes de vídeo quedan guardados.

- Requisitos

Tabla 20: Descripción RF_A_03

PROPIEDAD	DESCRIPCIÓN

Identificador del requisito	RF_A_03
Nombre del requisito	Modificar configuración de vídeo
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Baja/Opcional
Descripción del requisito	Se deben poder modificar los valores de la configuración de vídeo. Se disponen varias opciones seleccionables para cada propiedad de vídeo. En caso de que alguna sea modificada se debe guardar y aplicar en el momento que el usuario vuelva a la pantalla de "Options".
Datos de entrada	Los valores de los ajustes de vídeo ya sean modificados o no.
Datos de salida	Se muestran aplicados los nuevos valores de las propiedades de vídeo.
Requisito dependiente	• RF_A_01 • RF_A_02

- **Caso de uso nº 7.**
    - Caso de uso

Tabla 21: Descripción CU_07

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_07**
Nombre del caso de uso	Acceder configuración sonido
Actor/actores	Jugador/a
Descripción / Justificación	El jugador/a debe poder ver los ajustes de audio actuales.
Precondiciones	Estar en la pantalla de "Options".
Flujo normal	1. El jugador/a accede a la pantalla de "Audio" desde la pantalla "Options". 2. Se muestra la configuración actual de los ajustes de "Audio".
Flujo alternativo	1. El jugador/a accede a la pantalla de "Audio" desde la pantalla "Options". 2. Si no carga la pantalla de "Audio" se debe reiniciar el juego y volver a acceder.

	3. En caso de que no se quieran modificar los ajustes de audio se debe pulsar el botón de "Back" sin cambiar ningún valor.
Postcondiciones	Se muestran las propiedades actuales de audio para que el jugador/a pueda modificarlas.

- Requisitos

Tabla 22: Descripción RF_A_04

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_A_04**
Nombre del requisito	Mostrar configuración audio
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Baja/Opcional
Descripción del requisito	Al hacer clic en la opción de "Audio" del menú de opciones se deben mostrar los valores actuales de la configuración de audio.
Datos de entrada	Evento de clic izquierdo del ratón o el botón correspondiente del mando.
Datos de salida	Propiedades actuales de la configuración de audio.
Requisito dependiente	• RF_B_01 • RF_B_02 • RF_A_01

- **Caso de uso nº 8.**
  - Caso de uso

Tabla 23: Descripción CU_08

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_08**
Nombre del caso de uso	Modificar configuración audio
Actor/actores	Jugador/a
Descripción / Justificación	El jugador/a debe poder cambiar los ajustes de audio a su gusto.

Precondiciones	Estar en la pantalla de opciones de “Audio”.
Flujo normal	1. El jugador/a accede a la pantalla de “Audio” tras seleccionarla en la pantalla “Options”. 2. El jugador/a modifica las opciones que desee haciendo uso de los controles disponibles. 3. Se guardan las opciones modificadas. 4. Se pulsa el botón “Back” para volver a la pantalla anterior y que se guarden los nuevos valores.
Flujo alternativo	1. El jugador/a accede a la pantalla de “Audio” tras seleccionarla en la pantalla “Options”. 2. Si no puede modificar ninguna opción se debe reiniciar el juego y volver a acceder. 3. En caso de que no se quieran modificar los ajustes de audio se debe pulsar el botón de “Back” sin cambiar ningún valor.
Postcondiciones	Los nuevos ajustes de audio quedan guardados.

- Requisitos

Tabla 24: Descripción RF_A_05

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_A_05**
Nombre del requisito	Modificar configuración audio
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Baja/Opcional
Descripción del requisito	Se deben poder modificar los valores de la configuración de audio. Se disponen varias opciones seleccionables para cada propiedad de audio. En caso de que alguna sea modificada se debe guardar y aplicar en el momento que el usuario vuelva a la pantalla de “Options”.
Datos de entrada	Los valores de los ajustes de audio ya sean modificados o no.
Datos de salida	Se muestran aplicados los nuevos valores de las propiedades de audio.
Requisito dependiente	• RF_A_01

	• RF_A_04

- **Caso de uso nº 9.**
  - Caso de uso

Tabla 25: Descripción CU_09

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_09**
Nombre del caso de uso	Comenzar partida
Actor/actores	Jugador/a
Descripción / Justificación	El jugador/a decide iniciar la partida.
Precondiciones	El jugador/a debe estar en el menú principal.
Flujo normal	1. El jugador/a se encuentra en el menú principal. 2. Selecciona la opción "Start Game". 3. Se inicializan los elementos que componen el videojuego.
Flujo alternativo	1. El jugador/a se encuentra en el menú principal. 2. El jugador/a no quiere jugar y pulsa la opción "Quit Game" para cerrar el juego. 3. En caso de que no cargue la partida se debe reiniciar el juego y volver a acceder.
Postcondiciones	Todos los elementos del videojuego quedan inicializados y listos para ser mostrados en pantalla.

  - Requisitos

Tabla 26: Descripción RF_B_04

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_04**
Nombre del requisito	Iniciar partida
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Alta/Necesidad

Descripción del requisito	Al hacer clic en la opción "Start Game" del menú principal se deben iniciar todas las variables que componen a los elementos del juego para una correcta carga del nivel.
Datos de entrada	Valores para las variables que conformen a los diferentes actores del videojuego.
Datos de salida	No se produce ningún dato de salida.
Requisito dependiente	• RF_B_01 • RF_B_02

- **Caso de uso nº 10.**
    - Caso de uso

Tabla 27: Descripción CU_10

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_10**
Nombre del caso de uso	Cargar pantalla juego
Actor/actores	Jugador/a
Descripción / Justificación	Al hacer clic en la opción "Start Game" del menú principal se deben cargar todos los elementos que conforman el videojuego después de haber sido inicializados. En el nivel se debe mostrar el escenario completo con enemigos, objetos y el personaje principal. También, se carga la interfaz de juego.
Precondiciones	Que los elementos del juego se hayan inicializado de forma correcta.
Flujo normal	1. El jugador/a se encuentra en el menú principal. 2. Selecciona la opción "Start Game". 3. Se inicializan todos los elementos del videojuego. 4. Se carga la pantalla de juego.
Flujo alternativo	1. El jugador/a se encuentra en el menú principal. 2. El jugador/a no quiere jugar y pulsa la opción "Quit Game" para cerrar el juego. 3. En caso de que no cargue la partida se debe reiniciar el juego y volver a acceder.
Postcondiciones	Se muestran por pantalla todos los elementos que componen el videojuego, así como la interfaz de juego.

- Requisitos

Tabla 28: Descripción RF_B_05

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_05**
Nombre del requisito	Cargar pantalla de juego
Tipo	Requisito
Fuente del requisito	RF_B_04
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	Al hacer clic en la opción "Start Game" del menú principal se debe cargar la escena de juego principal. En esta escena se deben visualizar todos los elementos que componen el nivel completo, objetos, enemigos y el personaje principal.
Datos de entrada	Elementos que componen el videojuego inicializados.
Datos de salida	Todos los elementos que componen el nivel del videojuego se muestran por pantalla.
Requisito dependiente	• RF_B_01 • RF_B_02 • RF_B_04

Tabla 29: Descripción RF_B_06

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_06**
Nombre del requisito	Mostrar interfaz de juego
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	Al hacer clic en la opción "Start Game" del menú principal se debe cargar la interfaz con la información de vida, resistencia y objetos del personaje.
Datos de entrada	Datos de vida, resistencia y objetos que posee el personaje.
Datos de salida	Todos los elementos que componen la interfaz de juego se muestran por pantalla.

Requisito dependiente	• RF_B_01 • RF_B_02 • RF_B_04 • RF_B_05

Tabla 30: Descripción RF_B_07

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_07**
Nombre del requisito	Actualizar interfaz de juego
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	Durante la partida, los valores de vida, resistencia y objetos se ven modificados, de modo que esa información se debe ver actualizada en todo momento en la interfaz de juego.
Datos de entrada	Datos de vida, resistencia y objetos que posee el personaje.
Datos de salida	Todos los elementos que componen la interfaz de juego se muestran actualizados por pantalla.
Requisito dependiente	• RF_B_04 • RF_B_05 • RF_B_06 • RF_B_09 • RF_B_14 • RF_B_15 • RF_B_16 • RF_B_18 • RF_B_19 • RF_B_20 • RF_B_25 • RF_B_26 • RF_B_27 • RF_B_29 • RF_B_30

- **Caso de uso nº 11.**
  - Caso de uso

Tabla 31: Descripción CU_11

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_11**
Nombre del caso de uso	Mover personaje
Actor/actores	Jugador/a
Descripción / Justificación	El jugador/a debe poder mover al personaje principal por todo el escenario.
Precondiciones	• Estar en la pantalla de juego. • El jugador/a no se encuentra en el menú principal o de pausa.
Flujo normal	1. El jugador/a accede a la pantalla de juego. 2. Se cargan todos los elementos del juego. 3. El jugador/a puede mover al personaje haciendo uso del teclado y ratón o con un mando.
Flujo alternativo	1. El jugador/a no pulsa ninguna tecla que produce movimiento. 2. En caso de que no cargue la pantalla de juego se debe salir y reiniciar el juego. 3. En caso de que el personaje no se mueva se deben comprobar las conexiones de los periféricos. 4. Si el personaje sigue sin moverse se debe salir y reiniciar el juego.
Postcondiciones	El personaje se mueve a deseo del jugador/a.

  - Requisitos

Tabla 32: Descripción RF_B_08

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_08**
Nombre del requisito	Movimiento personaje principal
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Alta/Necesidad

Descripción del requisito	Se debe implementar el sistema de movimiento para el personaje principal. Este se moverá a partir de las entradas de teclado y ratón o del mando.
Datos de entrada	Eventos de entrada producidos por el teclado y ratón o por el mando.
Datos de salida	Se visualiza el correcto movimiento del personaje en función de las acciones del jugador/a.
Requisito dependiente	• RF_B_04 • RF_B_05

- **Caso de uso nº 12.**
    - Caso de uso

Tabla 33: Descripción CU_12

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_12**
Nombre del caso de uso	Equipar arma
Actor/actores	Jugador/a
Descripción / Justificación	El jugador/a debe poder coger un arma y equiparla para usarla en combate.
Precondiciones	• Estar en la pantalla de juego. • El jugador/a no se encuentra en el menú principal o de pausa.
Flujo normal	1. El jugador/a está en la pantalla de juego. 2. Que haya un arma disponible para ser equipada. 3. Pulsa el botón correspondiente para coger el arma estando cerca de ella. 4. El arma se equipa.
Flujo alternativo	1. El jugador/a está en la pantalla de juego 2. Si el personaje ya cuenta con un arma, la recogida se almacena para poder cambiada posteriormente. 3. En caso de que el personaje no pueda coger el arma se deben comprobar las conexiones de los periféricos.

	4. Si el personaje sigue poder coger el arma se debe salir y reiniciar el juego.
Postcondiciones	El arma queda equipada y el jugador/a puede usarla para atacar.

- Requisitos

Tabla 34: Descripción RF_B_09

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_09**
Nombre del requisito	Equipar arma
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	El jugador/a debe poder recoger un arma que esté cerca del personaje pulsando el botón correspondiente del teclado o del mando. En ese momento el arma se equipa y se adjunta al personaje como hijo del personaje. En caso de que ya haya un arma equipada, el arma recogida se almacena para poder ser cambiada posteriormente.
Datos de entrada	• Eventos de entrada producidos por el teclado y ratón o por el mando. • El arma que el jugador/a quiera recoger.
Datos de salida	No se produce ningún dato de salida.
Requisito dependiente	• RF_B_04 • RF_B_05

- **Caso de uso nº 13.**
  - Caso de uso

Tabla 35: Descripción CU_13

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_13**
Nombre del caso de uso	Equipar arma a dos manos
Actor/actores	Jugador/a

Descripción / Justificación	El jugador/a puede equipar el arma de ataque a dos manos para realizar ataques más potentes que con una mano. El escudo, en caso de poseerlo, queda a la espalda.
Precondiciones	• Estar en la pantalla de juego. • El jugador/a no se encuentra en el menú principal o de pausa. • Tener un arma equipada.
Flujo normal	1. El jugador/a se encuentra en la pantalla de juego. 2. El jugador/a tiene equipada un arma. 3. El jugador/a pulsa el botón correspondiente para equipar el arma a dos manos.
Flujo alternativo	1. El jugador/a se encuentra en la pantalla de juego. 2. El jugador/a no quiere equipar el arma a dos manos. 3. El jugador/a desea volver a tener el arma equipada con una mano solo.
Postcondiciones	El jugador/a puede realizar ataques a dos manos.

- Requisitos

Tabla 36: Descripción RF_B_10

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_10**
Nombre del requisito	Equipar arma dos manos
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Media/Deseado
Descripción del requisito	El jugador/a debe poder equipar el arma de ataque a dos manos pulsando el botón correspondiente. Se debe reproducir la animación de coger el arma a dos manos y el escudo, en caso de que lo posea, se debe colocar en la espalda.
Datos de entrada	Eventos de entrada producidos por el teclado y ratón o por el mando.
Datos de salida	No se produce ningún dato de salida.
Requisito dependiente	• RF_B_09

- **Caso de uso nº 14.**
    - Caso de uso

Tabla 37: Descripción CU_14

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_14**
Nombre del caso de uso	Atacar enemigos cuerpo a cuerpo
Actor/actores	Jugador/a
Descripción / Justificación	El jugador/a debe poder atacar a los enemigos cuerpo a cuerpo haciendo uso del arma que tenga equipada. Los ataques pueden ser a una o dos manos.
Precondiciones	• Estar en la pantalla de juego. • El jugador/a no se encuentra en el menú principal o de pausa. • El personaje debe tener equipada un arma.
Flujo normal	1. El jugador/a se encuentra en la pantalla de juego. 2. Hay un enemigo cerca. 3. El jugador/a pulsa el botón de ataque (normal o fuerte). 4. Se reproduce la animación de ataque. 5. Si se pulsa repetidas veces se realiza un ataque combinado. 6. El enemigo recibe daño.
Flujo alternativo	1. El jugador/a se encuentra en la pantalla de juego. 2. Hay un enemigo cerca. 3. El jugador/a decide no atacar. 4. En caso de que el personaje no pueda atacar se deben comprobar las conexiones de los periféricos. 5. Si el personaje sigue poder atacar se debe salir y reiniciar el juego.
Postcondiciones	El enemigo sufre daño.

- Requisitos

Tabla 38: Descripción RF_B_11

PROPIEDAD	DESCRIPCIÓN

Identificador del requisito	**RF_B_11**
Nombre del requisito	Realizar ataque
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	Al tener equipada un arma el jugador/a pulsa el botón de ataque normal o fuerte y se reproduce la animación correspondiente.
Datos de entrada	Eventos de entrada producidos por el teclado y ratón o por el mando.
Datos de salida	No se produce ningún dato de salida.
Requisito dependiente	• RF_B_09

Tabla 39: Descripción RF_B_12

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_12**
Nombre del requisito	Fijar enemigo
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	El jugador/a puede fijar la cámara en un enemigo para centrar sus ataques en él. Se dibuja una marca sobre el enemigo fijado.
Datos de entrada	Eventos de entrada producidos por el teclado y ratón o por el mando.
Datos de salida	Se dibuja el símbolo de fijado en el enemigo.
Requisito dependiente	• RF_B_09 • RF_B_11

- **Caso de uso nº 15.**
  - Caso de uso

Tabla 40: Descripción CU_15

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_15**

Nombre del caso de uso	Contraatacar enemigos
Actor/actores	Jugador/a
Descripción / Justificación	Cuando un enemigo ataca al jugador/a, este puede repeler el ataque del enemigo con el escudo y realizar un ataque especial.
Precondiciones	• Estar en la pantalla de juego. • El jugador/a no se encuentra en el menú principal o de pausa. • Tener equipada un arma y un escudo. • Que un enemigo ataque al jugador/a.
Flujo normal	1. El jugador/a es atacado por un enemigo. 2. El jugador/a pulsa el botón correspondiente para repeler el ataque. 3. El jugador/a pulsa el botón correspondiente para atacar mientras el enemigo está desprotegido.
Flujo alternativo	1. El jugador/a es atacado por un enemigo. 2. El jugador/a pulsa el botón para repeler el ataque tarde o pronto y recibe daño. 3. El jugador/a simplemente se protege del ataque con el escudo, pero no desvía el ataque.
Postcondiciones	El enemigo queda desprotegido y el jugador/a puede hacer un ataque especial.

- Requisitos

Tabla 41: Descripción RF_B_13

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_13**
Nombre del requisito	Contraatacar enemigo
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Media/Deseado
Descripción del requisito	Cuando el jugador/a recibe un ataque y este pulsa el botón correspondiente para evadirlo con el escudo equipado, puede

	efectuar un ataque especial pulsando el botón de ataque normal.
Datos de entrada	Eventos de entrada producidos por el teclado y ratón o por el mando.
Datos de salida	No se produce ningún dato de salida.
Requisito dependiente	• RF_B_09 • RF_B_11 • RF_B_16

- **Caso de uso nº 16.**
    - Caso de uso

Tabla 42: Descripción CU_16

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_16**
Nombre del caso de uso	Matar enemigos
Actor/actores	Jugador/a
Descripción / Justificación	El jugador/a mata a un enemigo tras ejercerle el daño equivalente a la vida del enemigo con los ataques.
Precondiciones	• Estar en la pantalla de juego. • El jugador/a no se encuentra en el menú principal o de pausa. • Tener equipada un arma.
Flujo normal	1. El jugador/a ataca al enemigo sucesivas veces. 2. El enemigo recibe daño. 3. La vida del enemigo llega a 0. 4. El enemigo queda reducido a un objeto sin comportamiento inteligente.
Flujo alternativo	No existe flujo alternativo.
Postcondiciones	El enemigo muere.

- Requisitos

Tabla 43: Descripción RF_B_14

PROPIEDAD	DESCRIPCIÓN

Identificador del requisito	**RF_B_14**
Nombre del requisito	Ejercer daño
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	Al entrar en contacto el arma del personaje y el enemigo cuando se ataca, el enemigo recibe la cantidad de daño que corresponda según el arma y las estadísticas del personaje.
Datos de entrada	• Eventos de entrada producidos por el teclado y ratón o por el mando. • Estadísticas del personaje y del arma. • Estadísticas del enemigo al que aplicar el daño.
Datos de salida	No se produce ningún dato de salida.
Requisito dependiente	• RF_B_09 • RF_B_11

- **Caso de uso nº 17.**
    - Caso de uso

Tabla 44: Descripción CU_17

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_17**
Nombre del caso de uso	Acumular almas
Actor/actores	Jugador/a
Descripción / Justificación	Cuando el jugador/a mata a un enemigo acumula almas, que son la moneda del juego.
Precondiciones	• Estar en la pantalla de juego. • El jugador/a no se encuentra en el menú principal o de pausa. • Tener un arma. • Matar a un enemigo.
Flujo normal	1. El jugador/a se encuentra en la pantalla de juego. 2. Mata a un enemigo.

	3. Acumula un número de almas determinado según el enemigo que haya matado.
Flujo alternativo	No existe flujo alternativo.
Postcondiciones	El contador de almas se incrementa.

- Requisitos

Tabla 45: Descripción RF_B_15

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_15**
Nombre del requisito	Acumular almas
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	Cuando el jugador/a mata a un enemigo el contador de almas se incrementa. En función del tipo de enemigo que haya matado se incrementa un número determinado de almas.
Datos de entrada	Número de almas a incrementar.
Datos de salida	Total de almas acumuladas.
Requisito dependiente	• RF_B_09 • RF_B_11 • RF_B_14

- **Caso de uso nº 18.**
  - Caso de uso

Tabla 46: Descripción CU_18

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_18**
Nombre del caso de uso	Equipar escudo
Actor/actores	Jugador/a
Descripción / Justificación	El jugador/a debe ser capaz de equiparse un escudo para defenderse de los ataques.
Precondiciones	• Estar en la pantalla de juego.

	• El jugador/a no se encuentra en el menú principal o de pausa.
Flujo normal	1. El jugador/a está en la pantalla de juego. 2. Que haya un escudo disponible para ser equipado. 3. Pulsa el botón correspondiente para coger el escudo estando cerca de él. 4. El escudo se equipa.
Flujo alternativo	1. El jugador/a está en la pantalla de juego. 2. Si el personaje ya cuenta con un escudo este se almacena para poder se cambiado por el actual. 3. En caso de que el personaje no pueda coger el escudo se deben comprobar las conexiones de los periféricos. 4. Si el personaje sigue sin poder coger el escudo se debe salir y reiniciar el juego.
Postcondiciones	El escudo queda equipado y el jugador/a puede usarlo para defenderse y desviar ataques.

- Requisitos

Tabla 47: Descripción RF_B_16

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_16**
Nombre del requisito	Equipar escudo
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	El jugador/a debe poder recoger un escudo que esté cerca del personaje pulsando el botón correspondiente del teclado o del mando. En ese momento el escudo se equipa y se adjunta al personaje como hijo del personaje. En caso de que ya haya un escudo equipado, el escudo recogido se almacena para poder ser cambiado posteriormente.
Datos de entrada	• Eventos de entrada producidos por el teclado y ratón o por el mando.

	• El arma que el jugador/a quiera recoger.
Datos de salida	No se produce ningún dato de salida.
Requisito dependiente	• RF_B_04 • RF_B_05

- **Caso de uso nº 19.**
    - Caso de uso

Tabla 48: Descripción CU_19

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_19**
Nombre del caso de uso	Defender ataques
Actor/actores	Jugador/a
Descripción / Justificación	El jugador/a debe poder defenderse de los ataques enemigos.
Precondiciones	• Estar en la pantalla de juego. • El jugador/a no se encuentra en el menú principal o de pausa. • Tener un escudo equipado.
Flujo normal	1. El jugador/a se encuentra en la pantalla de juego. 2. Hay al menos un enemigo atacando al jugador/a. 3. El jugador/a pulsa el botón correspondiente a defenderse. 4. Se resta cierta cantidad de resistencia. 5. No se resta vida.
Flujo alternativo	1. El jugador/a se encuentra en la pantalla de juego. 2. Hay al menos un enemigo atacando al jugador/a. 3. El jugador/a pulsa el botón correspondiente a defenderse. 4. En caso de no tener suficiente resistencia para protegerse bien del golpe se resta una determinada cantidad de vida. 5. Si se consume la resistencia restante de forma exacta el jugador/a queda desprotegido.

Postcondiciones	El jugador/a es capaz de protegerse contra los ataques enemigos o reducir el daño que recibe.

- Requisitos

Tabla 49: Descripción RF_B_17

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_17**
Nombre del requisito	Defender ataques
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	El jugador/a pulsa el botón de defenderse para evitar el daño del ataque enemigo. Se resta un porcentaje de resistencia en función del tipo de ataque y personaje enemigo. Mientras el jugador/a se protege la resistencia se recupera más despacio. En caso de no tener suficiente resistencia para protegerse completamente del golpe se resta una determinada cantidad de vida. Si se consume la resistencia restante de forma exacta el jugador/a queda desprotegido
Datos de entrada	• Resistencia actual del jugador/a. • Fuerza del ataque enemigo. • Total de defensa del escudo.
Datos de salida	Daño total y resistencia consumida.
Requisito dependiente	• RF_B_16

- **Caso de uso nº 20.**
  - Caso de uso

Tabla 50: Descripción CU_20

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_20**
Nombre del caso de uso	Coger objetos
Actor/actores	Jugador/a

Descripción / Justificación	El jugador/a debe poder coger objetos que no sean ni armas ni escudos que son útiles para la experiencia de juego.
Precondiciones	• Estar en la pantalla de juego. • El jugador/a no se encuentra en el menú principal o de pausa. • Que haya algún objeto que se pueda coger cerca del jugador/a.
Flujo normal	1. El jugador/a se encuentra en la pantalla de juego. 2. Hay al menos un objeto que se puede coger cerca del jugador/a. 3. EL jugador/a pulsa el botón correspondiente para coger el objeto. 4. El objeto se almacena para ser usado posteriormente.
Flujo alternativo	1. El jugador/a se encuentra en la pantalla de juego. 2. Hay al menos un objeto que se puede coger cerca del jugador/a. 3. El jugador/a decide no coger el objeto.
Postcondiciones	El objeto se almacena para ser usado posteriormente.

- Requisitos

Tabla 51: Descripción RF_B_18

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_18**
Nombre del requisito	Coger objeto
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	Cuando el jugador/a esté cerca de un objeto se mostrará un mensaje indicando el botón que se debe pulsar para coger el objeto. Cuando se coge el objeto se debe almacenar en los objetos que tiene el jugador/a para poder se usado posteriormente.
Datos de entrada	El objeto recogido por el jugador/a.

Datos de salida	Se presenta un mensaje indicando el objeto que se ha recogido.
Requisito dependiente	• RF_B_04 • RF_B_05

- **Caso de uso nº 21.**
  - Caso de uso

Tabla 52: Descripción CU_21

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_21**
Nombre del caso de uso	Cambiar objetos
Actor/actores	Jugador/a
Descripción / Justificación	El jugador/a debe poder cambiar de objeto para adaptar su forma de combate en todo momento.
Precondiciones	• Estar en la pantalla de juego. • El jugador/a no se encuentra en el menú principal o de pausa. • Que el jugador/a tenga al menos dos objetos.
Flujo normal	1. El jugador/a se encuentra en la pantalla de juego. 2. El jugador/a pulsa el botón correspondiente para cambiar entre objetos. 3. El objeto que tenía equipado pasa a estar oculto. 4. El nuevo objeto actual que ocupa la casilla de equipados que se muestra en la interfaz.
Flujo alternativo	1. El jugador/a se encuentra en la pantalla de juego. 2. El jugador/a pulsa el botón correspondiente para cambiar entre objetos. 3. El jugador/a no tiene objetos que cambiar con lo que no se aprecia ningún cambio.
Postcondiciones	El jugador/a /a puede cambiar el objeto que está equipado en función de sus necesidades.

  - Requisitos

Tabla 53: Descripción RF_B_19

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_19**
Nombre del requisito	Cambiar objeto
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Media/Deseado
Descripción del requisito	Cuando el jugador/a quiere cambiar los objetos que tiene equipados pulsa una de las flechas del mando o del teclado y cambia tanto los objetos consumibles como las armas y los escudos. El objeto que tiene equipado en el momento que pulsa la tecla o botón pasa a estar almacenado y el siguiente objeto que tuviera almacenado pasa a ser equipado.
Datos de entrada	Dirección de la flecha que corresponde a la casilla de objetos que quiere alternar.
Datos de salida	Se actualiza la interfaz con el objeto que pasa a estar equipado.
Requisito dependiente	• RF_B_18 • RF_B_16 • RF_B_09

- **Caso de uso nº 22.**
    - Caso de uso

Tabla 54: Descripción CU_22

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_22**
Nombre del caso de uso	Consumir objetos
Actor/actores	Jugador/a
Descripción / Justificación	El jugador/a debe poder consumir los objetos que se hayan integrado en el juego con dicho propósito.
Precondiciones	• Estar en la pantalla de juego. • El jugador/a no se encuentra en el menú principal o de pausa. • Que el jugador/a tenga al menos un objeto consumible.

Flujo normal	1. El jugador/a se encuentra en la pantalla de juego. 2. El jugador/a pulsa el botón correspondiente para consumir objetos. 3. Se modifican las estadísticas y datos correspondientes a consecuencia de haber consumido el objeto. 4. Se elimina el objeto de la lista de objetos consumibles.
Flujo alternativo	No existe flujo alternativo.
Postcondiciones	Los efectos del objeto se aplican al personaje del jugador/a y el objeto se elimina de la lista de consumibles.

- Requisitos

Tabla 55: Descripción RF_B_20

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_20**
Nombre del requisito	Consumir objeto
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	El jugador/a cuenta con una serie de objetos consumibles. Cuando pulsa el botón correspondiente para consumir el objeto este desaparece de la casilla de objetos consumibles a la vez que se aplican los efectos que este tiene asociados.
Datos de entrada	Objeto que se quiere consumir.
Datos de salida	No se producen datos de salida.
Requisito dependiente	• RF_B_18

- **Caso de uso nº 23.**
  - Caso de uso

Tabla 56: Descripción CU_23

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_23**

Nombre del caso de uso	Descansar en hoguera
Actor/actores	Jugador/a
Descripción / Justificación	El jugador/a debe poder descansar en la hoguera y tener un lugar donde no haya peligro.
Precondiciones	• Estar en la pantalla de juego. • El jugador/a no se encuentra en el menú principal o de pausa.
Flujo normal	1. El jugador/a está en la pantalla de juego. 2. El jugador/a se sitúa cerca de la hoguera. 3. El jugador/a pulsa el botón correspondiente para interactuar con la hoguera. 4. El personaje descansa en la hoguera.
Flujo alternativo	1. El jugador/a está en la pantalla de juego. 2. Hay uno o varios enemigos cerca de la hoguera. 3. El jugador/a no puede descansar en la hoguera porque hay enemigos cerca.
Postcondiciones	El jugador/a está fuera de peligro mientras descansa en la hoguera.

- Requisitos

Tabla 57: Descripción RF_B_21

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_21**
Nombre del requisito	Descansar en hoguera
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	Cuando el jugador/a se encuentre cerca de la hoguera y no haya enemigos cerca debe aparecer un mensaje indicando el botón a pulsar para interactuar con la hoguera. En el momento que lo pulse el jugador/a no podrá ser atacado y reproducirá la animación de sentarse en la hoguera.

Datos de entrada	• Eventos de entrada producidos por el teclado y ratón o por el mando.
Datos de salida	No se producen datos de salida.
Requisito dependiente	• RF_B_04 • RF_B_05

- **Caso de uso nº 24.**
  - Caso de uso

Tabla 58: Descripción CU_24

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_24**
Nombre del caso de uso	Reiniciar enemigos
Actor/actores	Jugador/a
Descripción / Justificación	Cuando el jugador/a descansa en la hoguera los enemigos se reinician.
Precondiciones	• Estar en la pantalla de juego. • El jugador/a no se encuentra en el menú principal o de pausa. • Haber descansado o reaparecido en la hoguera.
Flujo normal	1. El jugador/a está en la pantalla de juego. 2. El jugador/a descansa o reaparece en la hoguera. 3. Los enemigos desaparecen de la escena. 4. Los enemigos aparecen de nuevo en su lugar de origen con toda la vida y resistencia.
Flujo alternativo	No existe flujo alternativo.
Postcondiciones	Los enemigos quedan reiniciados al estado inicial.

  - Requisitos

Tabla 59: Descripción RF_B_22

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_22**
Nombre del requisito	Reiniciar enemigos
Tipo	Requisito

Fuente del requisito	RF_B_21
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	Cuando el jugador/a descansa o reaparece en la hoguera se eliminan todos los enemigos de la escena y vuelven a aparecer en sus posiciones originales con las estadísticas iniciales de vida y resistencia.
Datos de entrada	No hay datos de entrada.
Datos de salida	No se producen datos de salida.
Requisito dependiente	• RF_B_21

- **Caso de uso nº 25.**
  - Caso de uso

Tabla 60: Descripción CU_25

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_25**
Nombre del caso de uso	Restaurar pociones
Actor/actores	Jugador/a
Descripción / Justificación	Cuando el jugador/a descansa en la hoguera las pociones se restauran y consiguen su máximo de usos.
Precondiciones	• Estar en la pantalla de juego. • El jugador/a no se encuentra en el menú principal o de pausa. • Haber descansado o reaparecido en la hoguera.
Flujo normal	1. El jugador/a está en la pantalla de juego. 2. El jugador/a descansa o reaparece en la hoguera. 3. Las pociones vuelven a tener el máximo de usos.
Flujo alternativo	No existe flujo alternativo.
Postcondiciones	Las pociones vuelven a tener el máximo de usos.

  - Requisitos

Tabla 61: Descripción RF_B_23

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_23**

Nombre del requisito	Restaurar pociones
Tipo	Requisito
Fuente del requisito	RF_B_21
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	Cuando el jugador/a descansa o reaparece en la hoguera se restauran las pociones, volviendo a tener el máximo de usos.
Datos de entrada	No hay datos de entrada.
Datos de salida	No se producen datos de salida.
Requisito dependiente	RF_B_21

- **Caso de uso nº 26.**
    - Caso de uso

Tabla 62: Descripción CU_26

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_26**
Nombre del caso de uso	Subir de nivel
Actor/actores	Jugador/a
Descripción / Justificación	El jugador/a acumula almas al matar a los enemigos o al consumir objetos. Esas almas sirven para aumentar el nivel de las estadísticas del personaje. Subir de nivel una estadística tiene un coste determinado y que cada vez es mayor conforme más nivel tiene el personaje.
Precondiciones	• Estar descansando en la hoguera. • Tener almas suficientes para subir el nivel de alguna estadística.
Flujo normal	1. El jugador/a descansa en la hoguera. 2. El jugador/a aumenta el nivel de la estadística que quiera. 3. Acepta los cambios. 4. Se restan las almas que paga por subir de nivel. 5. Se aumenta la estadística. 6. Se aumenta el precio para subir de nivel la próxima vez.

Flujo alternativo	1. El jugador/a descansa en la hoguera. 2. El jugador/a aumenta el nivel varias estadísticas a la vez. 3. Cada estadística que quiere subir de nivel hace que la próxima que seleccione aun sin haber aceptado el cambio sea más cara. 4. En caso de que no quiera subir de nivel una estadística se abandona el menú de subir de nivel y no se gasta ningún alma.
Postcondiciones	El jugador/a ha subido de nivel la o las características deseadas y se han restado las almas que ha usado.

- Requisitos

Tabla 63: Descripción RF_B_24

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_24**
Nombre del requisito	Subir nivel de estadística
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Media/Deseado
Descripción del requisito	Cuando el jugador/a está descansando en la hoguera aparece un menú con la opción de subir de nivel. Al seleccionar esa opción aparece un menú con las estadísticas actuales del personaje. Para indicar que quiere subir de nivel una característica debe pulsar sobre la flecha que hay al lado de la característica. Para aplicar los cambios debe seleccionar la opción “Accept”.
Datos de entrada	• Estadística seleccionada • Coste total de almas
Datos de salida	Nuevo valor de la estadística.
Requisito dependiente	• RF_B_15 • RF_B_21

- **Caso de uso nº 27.**
  - Caso de uso

Tabla 64: Descripción CU_27

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_27**
Nombre del caso de uso	Restaurar vida, resistencia y maná
Actor/actores	Jugador/a
Descripción / Justificación	Cuando el jugador/a descansa en la hoguera, tanto la vida como la resistencia se restauran hasta el valor máximo.
Precondiciones	• Estar en la pantalla de juego. • El jugador/a no se encuentra en el menú principal o de pausa. • Haber descansado o reaparecido en la hoguera.
Flujo normal	1. El jugador/a está en la pantalla de juego. 2. El jugador/a descansa o reaparece en la hoguera. 3. La vida recupera su valor máximo. 4. La resistencia recupera su valor máximo. 5. El maná recupera su valor máximo.
Flujo alternativo	1. El jugador/a está en la pantalla de juego. 2. El jugador/a toma una poción para restaurar parte de la vida. 3. El jugador/a recupera resistencia de forma gradual si no ejerce ninguna acción que consuma resistencia.
Postcondiciones	La vida y la resistencia quedan restauradas a su valor máximo.

  - Requisitos

Tabla 65: Descripción RF_B_25

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_25**
Nombre del requisito	Restaurar vida, resistencia y maná
Tipo	Requisito
Fuente del requisito	RF_B_21
Prioridad del requisito	Alta/Necesidad

Descripción del requisito	Cuando el jugador/a descansa o reaparece en la hoguera la vida se restaura al completo instantáneamente. El mismo suceso se produce con la resistencia y el maná.
Datos de entrada	No se necesitan datos de entrada.
Datos de salida	No se producen datos de salida.
Requisito dependiente	RF_B_21

- **Caso de uso nº 28.**
  - Caso de uso

Tabla 66: Descripción CU_28

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_28**
Nombre del caso de uso	Consumir poción
Actor/actores	Jugador/a
Descripción / Justificación	El jugador/a decide tomar una poción para restaurar parte de su vida.
Precondiciones	• Estar en la pantalla de juego. • El jugador/a no se encuentra en el menú principal o de pausa. • Tener al menos una poción para consumir.
Flujo normal	1. El jugador/a está en la pantalla de juego. 2. El jugador/a tiene equipada una poción en la casilla de consumibles. 3. El jugador/a pulsa el botón correspondiente para consumir un objeto. 4. La poción se elimina de la lista de objetos consumibles.
Flujo alternativo	No existe flujo alternativo.
Postcondiciones	La poción no está disponible en la lista de consumibles.

  - Requisitos

Tabla 67: Descripción RF_B_26

PROPIEDAD	DESCRIPCIÓN

Identificador del requisito	**RF_B_26**
Nombre del requisito	Consumir poción
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	El jugador/a puede consumir una poción si se encuentra como activa en el espacio de objetos consumibles. Cuando el jugador pulsa el botón correspondiente para consumir la poción, esta desaparece de la lista de consumibles y aplica el efecto asociado, que en este caso es recuperar una porción de vida del personaje.
Datos de entrada	No se necesitan datos de entrada.
Datos de salida	No se producen datos de salida.
Requisito dependiente	RF_B_20

- **Caso de uso nº 29.**
  - Caso de uso

Tabla 68: Descripción CU_29

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_29**
Nombre del caso de uso	Restaurar vida
Actor/actores	Jugador/a
Descripción / Justificación	El jugador/a deber poder recuperar vida a lo largo de la partida sin necesidad de descansar en una hoguera.
Precondiciones	• Estar en la pantalla de juego. • El jugador/a no se encuentra en el menú principal o de pausa. • Tener al menos una poción para consumir.
Flujo normal	1. El jugador/a está en la pantalla de juego. 2. El jugador/a marca como consumible activo una poción. 3. El jugador/a consume la poción. 4. Se restaura un fragmento de vida instantáneamente.

Flujo alternativo	No existe flujo alternativo.
Postcondiciones	El jugador/a ha restaurado un fragmento de su barra de vida.

- Requisitos

Tabla 69: Descripción RF_B_27

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_27**
Nombre del requisito	Restaurar vida
Tipo	Requisito
Fuente del requisito	RF_B_26
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	Cuando el jugador/a consume una poción se restaura un fragmento de la barra de vida de forma casi instantánea.
Datos de entrada	Cantidad de vida a restaurar.
Datos de salida	Vida total actual.
Requisito dependiente	RF_B_26

- **Caso de uso nº 30.**
  - Caso de uso

Tabla 70: Descripción CU_30

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_30**
Nombre del caso de uso	Aumentar uso de pociones
Actor/actores	Jugador/a
Descripción / Justificación	El jugador/a puede aumentar el número de usos de las pociones de curación.
Precondiciones	• Estar descansando en la hoguera • Tener el objeto necesario para aumentar el uso de las pociones.
Flujo normal	1. El jugador/a está descansando en la hoguera. 2. El jugador/a ha conseguido un objeto para aumentar el número de usos de las pociones.

	3. El jugador/a selecciona la opción de aumentar número de pociones en el menú de la hoguera. 4. El jugador/a acepta los cambios. 5. Se elimina el objeto para aumentar el número de pociones. 6. El número de usos de las pociones se incrementa.
Flujo alternativo	No existe flujo alternativo.
Postcondiciones	El número de usos de las pociones se ha visto incrementado.

- Requisitos

Tabla 71: Descripción RF_A_06

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_A_06**
Nombre del requisito	Aumentar uso de pociones
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Baja/Opcional
Descripción del requisito	Cuando el jugador/a está descansando en la hoguera puede seleccionar la opción de aumentar el número de pociones que tiene. Para ello debe tener un objeto específico para ofrecerlo a la hoguera. Al ofrecer dicho objeto a la hoguera se aumenta el número de pociones que tiene el jugador/a.
Datos de entrada	• Objeto para aumentar uso de pociones. • Número máximo actual de pociones.
Datos de salida	Número máximo de pociones incrementado.
Requisito dependiente	RF_B_21

- **Caso de uso nº 31.**
  - Caso de uso

Tabla 72: Descripción CU_31

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_31**
Nombre del caso de uso	Potenciar efecto pociones

Actor/actores	Jugador/a
Descripción / Justificación	El jugador/a puede aumentar la cantidad de vida que restauran las pociones.
Precondiciones	• Estar descansando en la hoguera • Tener el objeto necesario para potenciar el efecto de las pociones.
Flujo normal	1. El jugador/a está descansando en la hoguera. 2. El jugador/a ha conseguido un objeto para potenciar el efecto de las pociones. 3. El jugador/a selecciona la opción de potenciar el efecto de las pociones en el menú de la hoguera. 4. El jugador/a acepta los cambios. 5. Se elimina el objeto para potenciar el efecto de las pociones. 6. El efecto de las pociones se incrementa.
Flujo alternativo	No existe flujo alternativo.
Postcondiciones	El efecto de las pociones se ha visto mejorado.

- Requisitos

Tabla 73: Descripción RF_A_07

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_A_07**
Nombre del requisito	Potenciar efecto pociones
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Baja/Opcional
Descripción del requisito	Cuando el jugador/a está descansando en la hoguera puede seleccionar la opción de potenciar el efecto de las pociones. Para ello debe tener un objeto específico para ofrecerlo a la hoguera. Al ofrecer dicho objeto a la hoguera se potencia el efecto de las pociones.
Datos de entrada	• Objeto para potenciar el efecto de las pociones. • Cantidad de vida que restaura la poción actualmente.

Datos de salida	No se producen datos de salida.
Requisito dependiente	RF_B_21

- **Caso de uso nº 32.**
    - Caso de uso

Tabla 74: Descripción CU_32

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_32**
Nombre del caso de uso	Invocar personaje no jugable
Actor/actores	Jugador/a
Descripción / Justificación	El jugador/a puede invocar a un personaje no jugable (NPC) para que le ayude en la pelea contra el jefe final.
Precondiciones	• Estar en la pantalla de juego. • El jugador/a no se encuentra en el menú principal o de pausa. • Estar cerca de una señal de invocación.
Flujo normal	1. El jugador/a está en la pantalla de juego. 2. El jugador/a se encuentra cerca de una señal de invocación. 3. El jugador/a pulsa el botón requerido para invocar al nuevo personaje. 4. El nuevo personaje se materializa en la partida. 5. El nuevo personaje se comporta de forma autónoma para atacar el jefe final.
Flujo alternativo	1. El jugador/a está en la pantalla de juego. 2. El jugador/a se encuentra cerca de una señal de invocación. 3. El jugador/a pulsa el botón requerido para invocar al nuevo personaje. 4. La invocación falla. 5. El nuevo personaje no puede ser materializado.
Postcondiciones	Se materializa un nuevo personaje para ayudar al jugador/a al derrotar al jefe.

- Requisitos

Tabla 75: Descripción RF_A_08

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_A_08**
Nombre del requisito	Invocar personaje no jugable
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Baja/Opcional
Descripción del requisito	El jugador/a se sitúa cerca de una señal de invocación. Al pulsar el botón correspondiente para activar la invocación aparece un personaje no jugable (NPC). Este personaje sigue al jugador/a hasta el jefe final y ayuda al jugador/a a derrotarlo. La implementación de este NPC es similar a la de los enemigos simples o el jefe final.
Datos de entrada	Eventos de entrada producidos por el teclado y ratón o por el mando.
Datos de salida	Se muestra en la pantalla de juego en nuevo personaje.
Requisito dependiente	• RF_B_04 • RF_B_05

- **Caso de uso nº 33.**
  - Caso de uso

Tabla 76: Descripción CU_33

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_33**
Nombre del caso de uso	Morir en el juego
Actor/actores	Jugador/a
Descripción / Justificación	Cuando la vida del personaje controlado por el jugador/a llega a 0, muere.
Precondiciones	• Estar en la pantalla de juego. • El jugador/a no se encuentra en el menú principal o de pausa. • Tener 0 puntos de vida.

Flujo normal	1. El jugador/a se encuentra en la pantalla de juego. 2. El jugador/a pierde toda la vida. 3. El jugador/a muere. 4. Se reproduce un mensaje de muerte en la pantalla.
Flujo alternativo	No existe flujo alternativo.
Postcondiciones	El jugador/a muere.

- Requisitos

Tabla 77: Descripción RF_B_28

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_28**
Nombre del requisito	Morir en el juego
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	Cuando la vida del jugador/a llega a 0, muere. De modo que se vuelve a cargar la pantalla de juego y el jugador/a reaparece.
Datos de entrada	No se necesitan datos de entrada.
Datos de salida	No se producen datos de salida.
Requisito dependiente	RF_B_04 RF_B_05 RF_B_14

- **Caso de uso nº 34.**
    - Caso de uso

Tabla 78: Descripción CU_34

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_34**
Nombre del caso de uso	Perder almas
Actor/actores	Jugador/a
Descripción / Justificación	Cuando el jugador/a muere pierde todas las almas acumuladas en la partida.
Precondiciones	• Estar en la pantalla de juego.

	• El jugador/a no se encuentra en el menú principal o de pausa. • Tener al menos un alma acumulada.
Flujo normal	1. El jugador/a está en la pantalla de juego. 2. El jugador/a pierde toda la vida. 3. El jugador/a muere. 4. El jugador/a pierde todas las almas.
Flujo alternativo	1. El jugador/a está en la pantalla de juego. 2. El jugador/a pierde toda la vida. 3. El jugador/a muere. 4. El jugado no tiene almas que perder. 5. El contador se mantiene a 0.
Postcondiciones	El jugador/a ha perdido todas las almas y el contador se establece a 0.

- Requisitos

Tabla 79: Descripción RF_B_29

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_29**
Nombre del requisito	Perder almas
Tipo	Requisito
Fuente del requisito	RF_B_28
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	Cuando el jugador/a muere el total de almas acumuladas por el jugador/a se establece a 0.
Datos de entrada	No se necesitan datos de entrada.
Datos de salida	No se producen datos de salida.
Requisito dependiente	RF_B_28

- **Caso de uso nº 35.**
  - Caso de uso

Tabla 80: Descripción CU_35

PROPIEDAD	DESCRIPCIÓN

Identificador del caso de uso	**CU_35**
Nombre del caso de uso	Recuperar almas
Actor/actores	Jugador/a
Descripción / Justificación	Cuando el jugador/a reaparece puede recuperar las almas que tenía acumuladas antes de morir si vuelve a la zona en la que ha muerto.
Precondiciones	• Estar en la pantalla de juego. • El jugador/a no se encuentra en el menú principal o de pausa. • Haber acumulado almas antes de morir.
Flujo normal	1. El jugador/a reaparece en la hoguera. 2. El jugador/a se desplaza por el escenario. 3. El jugador/a vuelve a la zona donde murió. 4. El jugador/a pulsa el botón correspondiente para recuperar las almas. 5. El jugador/a recupera las almas que tenía antes de morir.
Flujo alternativo	1. El jugador/a reaparece en la hoguera. 2. El jugador/a se desplaza por el escenario. 3. El jugador/a muere antes de llegar a la zona donde murió. 4. El jugador/a no recupera las almas. 5. El jugador/a reaparece. 6. El total de almas que puede recuperar es el actual o 0.
Postcondiciones	El jugador/a recupera las almas acumuladas antes de morir.

- Requisitos

Tabla 81: Descripción RF_B_30

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_30**
Nombre del requisito	Recuperar almas
Tipo	Requisito

Fuente del requisito	Jugador/a
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	Cuando el jugador/a muere se guarda la zona donde ha muerto y se genera una señal para indicarlo. Cuando el jugador/a reaparece puede volver al sitio donde ha muerto para recuperar las almas que tenía acumuladas antes de morir. Cuando llega a la zona pulsa el botón correspondiente para interactuar con la señal y recupera las almas. En caso de que muera antes de llegar a esa señal se elimina y se genera otra en el nuevo sitio donde ha muerto, perdiendo la opción de recuperar las almas que tenía antes de morir la primera vez.
Datos de entrada	Almas actuales y almas acumuladas antes de morir.
Datos de salida	Almas totales actuales.
Requisito dependiente	• RF_B_15 • RF_B_28 • RF_B_29

- **Caso de uso nº 36.**
    - Caso de uso

Tabla 82: Descripción CU_36

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_36**
Nombre del caso de uso	Reaparecer en hoguera
Actor/actores	Jugador/a
Descripción / Justificación	Cuando el personaje del jugador/a muere, reaparece en la última hoguera en la que había descansado antes de morir.
Precondiciones	• Haber muerto en el juego.
Flujo normal	1. El jugador/a está en la pantalla de juego. 2. El personaje del jugador/a muere. 3. El personaje reaparece en la última hoguera en la que descansó antes de morir.
Flujo alternativo	No existe flujo alternativo.
Postcondiciones	El jugador/a reaparece en la pantalla de juego.

- Requisitos

Tabla 83: Descripción RF_B_31

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_31**
Nombre del requisito	Reaparecer en hoguera
Tipo	Requisito
Fuente del requisito	RF_B_28
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	Cuando el personaje del jugador/a muere, la pantalla de juego vuelve a cargar de nuevo. El jugador/a reaparece en la última hoguera en la que descansó.
Datos de entrada	Identificador de la última hoguera en la que el jugador/a descansó antes de morir.
Datos de salida	Se vuelve a cargar la pantalla de juego.
Requisito dependiente	• RF_B_04 • RF_B_05 • RF_B_28

- **Caso de uso nº 37.**
  - Caso de uso

Tabla 84: Descripción CU_37

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_37**
Nombre del caso de uso	Pausar el juego
Actor/actores	Jugador/a
Descripción / Justificación	El jugador/a puede parar el juego en cualquier momento para simplemente pausar la partida o para salir al menú principal.
Precondiciones	Estar en la pantalla de juego.
Flujo normal	1. El jugador/a está en la pantalla de juego. 2. El jugador/a pulsa el botón correspondiente para pausar la partida. 3. Aparece el menú de pausa.
Flujo alternativo	No existe flujo alternativo.

Postcondiciones	Se muestra el menú de pausa en la pantalla y el juego se para.

- Requisitos

Tabla 85: Descripción RF_B_32

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_32**
Nombre del requisito	Pausar el juego
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Media/Deseado
Descripción del requisito	El jugador/a puede pausar el juego en cualquier momento para reanudarlo más tarde o salir al menú principal. Mientras el menú de pausa está activo el bucle del juego está pausado.
Datos de entrada	No se requieren datos de entrada.
Datos de salida	No se producen datos de salida.
Requisito dependiente	• RF_B_04 • RF_B_05

- **Caso de uso nº 38.**
  - Caso de uso

Tabla 86: Descripción CU_38

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_38**
Nombre del caso de uso	Reanudar el juego
Actor/actores	Jugador/a
Descripción / Justificación	Cuando el jugador/a se encuentra en el menú de pausa puede reanudar la partida para seguir jugando.
Precondiciones	• Estar en la pantalla de juego. • Estar en el menú de pausa.
Flujo normal	1. El jugador/a está en la pantalla de juego. 2. El jugador/a accede al menú de pausa. 3. El jugador/a selecciona la opción para volver al juego. 4. El jugador/a reanuda el juego.

Flujo alternativo	No existe flujo alternativo.
Postcondiciones	El juego se reanuda en el mismo estad en el que el jugador/a lo había pausado.

- Requisitos

Tabla 87: Descripción RF_B_33

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_33**
Nombre del requisito	Reanudar el juego
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Media/Deseado
Descripción del requisito	Cuando el jugador/a se encuentra en el menú de pausa y pulsa la opción de volver a la partida esta se reanuda en el mismo estado que estaba en el momento que se pausó.
Datos de entrada	Opción seleccionada del menú de pausa.
Datos de salida	No se producen datos de salida.
Requisito dependiente	• RF_B_04 • RF_B_05 • RF_B_32

- **Caso de uso nº 39.**
    - Caso de uso

Tabla 88: Descripción CU_39

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_39**
Nombre del caso de uso	Salir al menú principal
Actor/actores	Jugador/a
Descripción / Justificación	Cuando el jugador/a se encuentra en el menú de pausa puede salir al menú principal.
Precondiciones	• Estar en la pantalla de juego. • Estar en el menú de pausa.
Flujo normal	1. El jugador/a está en la pantalla de juego.

	2. El jugador/a accede al menú de pausa. 3. El jugador/a selecciona la opción para salir al menú principal. 4. Deja de reproducirse la pantalla de juego. 5. Se carga el menú principal.
Flujo alternativo	No existe flujo alternativo.
Postcondiciones	El jugador/a sale de la pantalla de juego y accede al menú principal.

- Requisitos

Tabla 89: Descripción RF_B_34

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_34**
Nombre del requisito	Salir al menú principal
Tipo	Reanudar el juego
Fuente del requisito	Requisito
Prioridad del requisito	Jugador/a
Descripción del requisito	Media/Deseado
Datos de entrada	Cuando el jugador/a se encuentra en el menú de pausa y pulsa la opción de salir al menú principal la pantalla de juego deja de reproducirse y se carga la pantalla del menú principal.
Datos de salida	Opción seleccionada del menú de pausa.
Requisito dependiente	No se producen datos de salida.
	• RF_B_02 • RF_B_32

- **Caso de uso nº 40.**
  - Caso de uso

Tabla 90: Descripción CU_40

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_40**
Nombre del caso de uso	Atravesar la niebla
Actor/actores	Jugador/a

Descripción / Justificación	Cuando el jugador/a se encuentra en la pantalla de juego y quiere acceder a la zona donde está el jefe final debe atravesar una pared de niebla.
Precondiciones	• Estar en la pantalla de juego. • El jugador/a no se encuentra en el menú principal o de pausa. • El jugador/a se encuentra cerca de la zona del jefe final.
Flujo normal	1. El jugador/a está en la pantalla de juego. 2. El jugador/a se acerca a la zona del jefe final. 3. El jugador/a pulsa el botón correspondiente para interactuar con la niebla. 4. El jugador/a atraviesa la niebla. 5. El jugador/a accede a la zona del jefe final.
Flujo alternativo	1. El jugador/a se encuentra en la zona del jefe final. 2. El jugador/a se acerca a la niebla que delimita la zona. 3. El jugador/a no puede atravesar de vuelta la niebla para escapar.
Postcondiciones	El jugador/a accede a la zona donde se encuentra el jefe final.

- Requisitos

Tabla 91: Descripción RF_B_35

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_35**
Nombre del requisito	Atravesar la niebla
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	Cuando el jugador/a se encuentra cerca de la zona del jefe final se presenta un muro de niebla en la entrada a dicha zona. El jugador/a debe acercarse a dicho muro para que se muestre un mensaje con el botón que debe pulsar para atravesar la niebla. Cuando el jugador/a pulse el botón debe reproducirse la

	animación correspondiente para atravesar el muro. Cuando el jugador/a accede a la zona del jefe final se activa el comportamiento de este. Los enemigos simples no pueden acceder a la zona del jefe.
Datos de entrada	No se necesitan datos de entrada.
Datos de salida	No se producen datos de salida.
Requisito dependiente	• RF_B_04 • RF_B_05

- **Caso de uso nº 41.**
    - Caso de uso

Tabla 92: Descripción CU_41

PROPIEDAD	DESCRIPCIÓN
Identificador del caso de uso	**CU_41**
Nombre del caso de uso	Derrotar al jefe final
Actor/actores	Jugador/a
Descripción / Justificación	El jugador/a debe poder matar al jefe final del juego para terminar la partida.
Precondiciones	• Estar en la pantalla de juego. • Estar en la zona del jefe final. • El jugador/a no se encuentra en el menú principal o de pausa.
Flujo normal	1. El jugador/a accede a la zona del jefe final. 2. Se activa el comportamiento del jefe final. 3. El jugador/a pelea contra el jefe final. 4. El jugador/a derrota al jefe final. 5. El jugador/a obtiene la recompensa.
Flujo alternativo	1. El jugador/a accede a la zona del jefe final. 2. Se activa el comportamiento del jefe final. 3. El jugador/a pelea contra el jefe final. 4. El jugador/a muere contra el jefe final. 5. Se reproduce un mensaje de muerte en la pantalla.

	6. El jugador/a reaparece en la última hoguera en la que ha descansado.
Postcondiciones	El jugador/a ha completado el videojuego.

- Requisitos

Tabla 93: Descripción RF_B_36

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RF_B_36**
Nombre del requisito	Derrotar al jefe final
Tipo	Requisito
Fuente del requisito	Jugador/a
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	Cuando el jugador/a accede a la zona del jefe final se activa el comportamiento de este. El jugador/a pelea contra el jefe final. En caso de que el jugador/a derrote al jefe final, este desaparece y el jugador/a recibe la recompensa. En caso de que el jugador/a muera en el combate, se reproduce un mensaje de muerte y reaparece en la última hoguera en la que ha descansado.
Datos de entrada	No se necesitan datos de entrada.
Datos de salida	No se producen datos de salida.
Requisito dependiente	• RF_B_02 • RF_B_08 • RF_B_11 • RF_B_13 • RF_B_14 • RF_B_17 • RF_B_35

b. Requisitos no funcionales

Los requisitos no funcionales son los que describen una restricción sobre el sistema que limita las elecciones en la construcción de una solución al problema. No están relacionados con las funcionalidades del sistema directamente sino con el comportamiento en general del sistema,

incluyendo así restricciones de tiempo, sobre el proceso de desarrollo, estándares, tolerancia a fallos y tiempos de ejecución, entre otros [101]. Dentro de los requisitos no funcionales se encuentran tres grandes categorías:

- **Requisitos de producto.** Aquellos requisitos que detallan el comportamiento del producto atendiendo a factores como el tiempo de respuesta, tiempo de ejecución, memoria requerida. Estos se clasifican en:
  - Requisitos de usabilidad.
  - Requisitos de portabilidad.
  - Requisitos de fiabilidad.
  - Requisitos de eficiencia.
    - Requisitos de espacio.
    - Requisitos de rendimiento.
- **Requisitos de organización.** Engloban políticas y procedimientos por parte de la organización del cliente y del desarrollador. Algunos ejemplos son requisitos relacionados con los lenguajes de programación, métodos de diseño y entregas. Los requisitos de organización se clasifican en:
  - Requisitos de entrega.
  - Requisitos de estándares.
  - Requisitos de implementación.
- **Requisitos externos.** Surgen a partir de factores externos al sistema y a su proceso de desarrollo. Incluyen requisitos de interoperabilidad entre otros sistemas de la organización, requisitos legales y éticos. Su clasificación es:
  - Requisitos de interoperabilidad.
  - Requisitos éticos.
  - Requisitos legislativos.
    - Requisitos de privacidad.
    - Requisitos de seguridad.

A continuación, se detallan los requisitos no funcionales incluidos en este proyecto, prestando especial interés en los requisitos de producto y de organización. En cuanto a los requisitos externos no se profundiza debido a que el carácter este proyecto es académico y el autor no tiene intención de comercializar el producto.

Los requisitos no funcionales utilizan el mismo formato de tabla que los requisitos funcionales para ser representados, pero omitiendo campos que no son necesarios en este caso.

Tabla 94: Descripción RNF_B_01

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RNF_B_01**
Nombre del requisito	Usabilidad
Tipo	Requisito
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	El jugador/a debe saber cómo interactuar con el sistema para poder progresar en el juego, de modo que el sistema debe ser usable y responder de la forma esperada por el jugador/a.

Tabla 95: Descripción RNF_B_02

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RNF_B_02**
Nombre del requisito	Fiabilidad
Tipo	Requisito
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	El jugador/a debe poder disfrutar de la experiencia de juego sin ninguna interrupción o fallo que pueda incomodarle.

Tabla 96: Descripción RNF_A_01

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RNF_A_01**
Nombre del requisito	Espacio en disco
Tipo	Restricción
Prioridad del requisito	Media/Deseado
Descripción del requisito	Los videojuegos desarrollados con el motor *Unreal Engine* suelen tener un peso considerable a la hora de ser almacenados. El autor pretende hacer un proyecto con el menor peso dentro de la medida de lo posible cuidando el tamaño de las texturas, elementos 3D, VFX y SFX.

Tabla 97: Descripción RNF_B_03

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RNF_B_03**
Nombre del requisito	Rendimiento

Tipo	Restricción
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	Una baja eficiencia en el uso de los recursos de la máquina sobre la que se ejecuta el juego puede repercutir sobre la experiencia de usuario y en el rendimiento. Asimismo, tratándose de un trabajo de fin de grado de Ingeniería Multimedia se toma como prioridad el funcionamiento óptimo y eficiente del proyecto para sacar el máximo provecho de la tecnología disponible en la máquina que ejecuta el juego.

Tabla 98: Descripción RNF_B_04

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RNF_B_04**
Nombre del requisito	Entrega
Tipo	Requisito
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	Dado que se trata de un proyecto académico deben cumplirse los plazos de entrega requeridos para que el trabajo pueda ser evaluado por el tribunal en la fecha acordada.

Tabla 99: Descripción RNF_B_05

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RNF_B_05**
Nombre del requisito	Implementación
Tipo	Requisito
Prioridad del requisito	Alta/Necesidad
Descripción del requisito	La implementación del proyecto es una necesidad ya que es una de las partes que muestra los conocimientos obtenidos a lo largo del grado aparte de este documento y la defensa ante el tribunal. La implementación se lleva a cabo según lo especificado en los diferentes apartados que componen este documento.

Tabla 100: Descripción RNF_B_06

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RNF_B_06**
Nombre del requisito	Fidelidad de experiencia
Tipo	Requisito
Prioridad del requisito	Media/Deseado
Descripción del requisito	El autor intenta mantener el estilo visual y de jugabilidad característico de los ***soulslike*** teniendo en cuenta los recursos de los que dispone. Con ello pretende que el jugador/a disfrute una experiencia lo más fiel posible en relación con dicho género.

Tabla 101: Descripción RNF_A_02

PROPIEDAD	DESCRIPCIÓN
Identificador del requisito	**RNF_A_02**
Nombre del requisito	Estándares
Tipo	Restricción
Prioridad del requisito	Baja/Opcional
Descripción del requisito	El producto resultante de este trabajo puede estar sujeto al estándar europeo **PEGI** (*Pan European Game Information*) [102]. Este dictamina la edad recomendada para jugar al videojuego y aporta información orientativa al consumidor.

B. Créditos y artefactos externos

a. Modelados de los personajes

- *Paragon Greystone*: https://www.unrealengine.com/marketplace/en-US/product/paragon-greystone
- *Paragon Aurora:* *https://www.unrealengine.com/marketplace/en-US/product/paragon-aurora*
- *Paragon Kwang:* *https://www.unrealengine.com/marketplace/en-US/product/paragon-kwang*

b. Modelados del entorno

- *Infinity Blades Ice Lands*: https://www.unrealengine.com/marketplace/en-US/product/infinity-blade-ice-lands
- *Bonfire*: "Dark Souls - Bonfire" (https://skfb.ly/6ztTx) by UselessViking is licensed under Creative Commons Attribution (http://creativecommons.org/licenses/by/4.0/).

c. Animaciones

- Mixamo Shield and Sword: https://www.mixamo.com/#/?page=1&query=sword+and+shield
- Bossy Enemy: https://www.unrealengine.com/marketplace/en-US/product/bossy-enemy-animation-pack
- Roll and Dodge: https://www.unrealengine.com/marketplace/en-US/product/rolls-and-dodges-animation-set

d. VFX

- Infinity Blades Effects: https://www.unrealengine.com/marketplace/en-US/product/infinity-blade-effects
- *Paragon Aurora Ice Skill*: *https://www.unrealengine.com/marketplace/en-US/product/paragon-aurora*
- *Paragon Greystone Novaborn:* https://www.unrealengine.com/marketplace/en-US/product/paragon-greystone

e. SFX

- *Infinity Blades Effects:* https://www.unrealengine.com/marketplace/en-US/product/infinity-blade-effects
- *Paragon Greystone Effort Sounds:* https://www.unrealengine.com/marketplace/en-US/product/paragon-greystone
- *Paragon Aurora Effort Sounds:* *https://www.unrealengine.com/marketplace/en-US/product/paragon-aurora*
- *Paragon Kwang Effort Sounds:* *https://www.unrealengine.com/marketplace/en-US/product/paragon-kwang*
- Música del menú principal: **"Private Reflection" Kevin MacLeod (incompetech.com) Licensed under Creative Commons: By Attribution 4.0 License http://creativecommons.org/licenses/by/4.0/**
- Música de batalla contra el jefe final: **"Gregorian Chant" Kevin MacLeod (incompetech.com) Licensed under Creative Commons: By Attribution 4.0 License http://creativecommons.org/licenses/by/4.0/**

f. GUI y HUD

- Tipografía: https://www.dafont.com/es/font-comment.php?file=diamond_gothic
- Icono del mando de XBOX: Image by Stephan https://pixabay.com/users/io-images-1096650/?utm_source=link-attribution&utm_medium=referral&utm_campaign=image&utm_content=1827840
- Icono del teclado: https://freesvg.org/img/1573723226Asus_K93SM_blank.png
- Icono del ratón: https://freesvg.org/img/DooFi-Mouse.png
- Icono de las pociones: https://www.freepik.es/vector-gratis/set-pociones-magicas_9464123.htm#query=magic%20potions%20set&position=4&from_view=search&track=sph en Freepik

C. Repositorio y descarga del proyecto

Repositorio de desarrollo: https://gitlab.com/ajfernandezbelliure/glacialsoul

Repositorio de *build*: https://gitlab.com/ajfernandezbelliure/glacial-soul-build

Página de Itch.io: https://antonio-jose-fernandez.itch.io/glacial-soul

Porfolio: http://antoniojosefernandezbelliure.atwebpages.com/index.html