Universitat de Girona
Escola Politècnica Superior

Treball final de grau

Estudi: Grau en Disseny i Desenvolupament de Videojocs

Títol: Desarrollo del remake de un videojuego de plataformas con desplazamiento lateral

Document: Memoria

Alumne: David García Prieto

Tutor: Gustavo Patow
Departament: IMAE
Àrea: LSI

Convocatòria (mes/any) Septiembre 2020

Índice

1. Introducción, objetivos y tipología del proyecto 15
1.1 Introducción 15
1.2 Objetivos 16
1.2.1 Objetivo principal del Trabajo de Fin de Grado 16
1.2.2 Desglose de Objetivos 16
1.3 Tipología del proyecto 16
2. Estudio de viabilidad 18
2.1 Estudio de mercado 18
2.1.1 Búsquedas realizadas 18
2.1.2 Resultados obtenidos 18
2.1.3 Conclusiones 21
2.2 Viabilidad tecnológica 22
2.3 Viabilidad económica 22
2.4 Viabilidad legal 22
2.5 Conclusión 23
3. Planificación 24
3.1 Paquetes de trabajo 24
3.2 Descripción de paquetes 25
3.2.1 Documentación 25
3.2.2 Análisis de requisitos 26
3.2.3 Estética 26
3.2.4 Mecánicas 27
3.2.5 Desarrollo tecnológico y testing 27
3.2 Cronograma 28
3.3 Metodología de trabajo 29
4. Marco de trabajo y conceptos previos 30
4.1 Referencia 30
4.2 Plataformas 31
4.3 PEGI 31
5. Herramientas de trabajo 33
5.1 Motor de juegos 33
5.1.1 Godot 33

5.1.2 Source 2 34
5.1.3 Unity 5 34
5.1.4 Unreal Engine 4 35
5.1.4 Conclusión 35
5.2 Programa de Modelado 3D 36
5.2.1 Blender 36
5.2.2 3ds Max 36
5.2.3 Maya 37
5.3.3 Conclusión 37
5.3 Programa de animación 37
5.4 Programa de diseño gráfico y vectorial 38
5.4.1 Adobe Photoshop 38
5.4.2 Adobe Illustrator 39
5.4.3 GIMP 39
5.4.4 Inkscape 40
5.4.5 Conclusión 40
6. Diseño del videojuego 41
6.1 Narrativa 41
6.1.1 Argumento 41
6.1.2 Localizaciones 41
6.2 Estética 42
6.2.1 Espacio 42
6.2.2 Localizaciones 42
6.2.3 Elementos que forman el escenario 46
6.2.2 Personajes 47
6.3 Objetos 49
6.4 Interfaz de Usuario 51
6.4.1 HUD 51
6.4.2 Menús 52
6.5 Gameplay 56
6.5.1 Definición de retos 56
6.5.2 Jerarquía de retos 57
6.4 Mecánicas 58
6.4.1 Recursos 59
6.4.2 Entidades 59

6.4.3 Acciones del jugador 60
6.4.4 Mecánicas e interacción con enemigos 61
6.4.5 Mecánicas e interacción con objetos 62
6.5 Flowchart 63
6.6 Level design 64
6.6.1 Economía interna 64
7. Desarrollo tecnológico, implementación y pruebas 65
7.1 Creación del proyecto 65
7.2 Importación de Personajes y Animaciones 67
7.2.1 Obtención de los modelos de los personajes 67
7.2.2 Rigging y animación con Mixamo 69
7.2.3 Cambio en los modelos de los enemigos 71
7.3 Cámara 73
7.4 Movimiento y acciones del personaje 76
7.4.1 Input 76
7.4.2 Blueprint de movimiento y acciones del personaje 77
7.5 Desarrollo de nivel 82
7.5.1 Pipeline de creación e importación de assets 82
7.5.2 Creación del terreno del nivel 85
7.5.3 Iluminación 86
7.6 Sonorización 87
7.6.1 Música y sonido ambiental 87
7.6.2 Efectos de sonido 88
7.7 Implementación de mecánicas de juego 89
7.7.1 Implementación de enemigos 89
7.7.2 Implementación de objetos 99
7.7.3 Otros 109
7.8 Implementación del Level blueprint y Game State blueprint 111
7.8.1 Level blueprint 111
7.8.2 Game State blueprint 112
7.9 Implementación del Animation Blueprint del personaje 113
7.10. Implementación de la Interfaz de usuario 117
7.10.1 HUD 118
7.10.2 Menú Principal 120
7.10.3 Menú de Opciones 123

7.10.4 Menús de Nivel Completado 127
7.10.5 Menú de Final de Partida 130
8. Resultados 131
9. Conclusiones personales 133
10. Trabajo futuro 134
11. Bibliografía 136

Índice de Figuras

Figura 1 Captura ingame de Donkey Kong Country Remix 15
Figura 2 Captura del juego Toki 19
Figura 3 Captura del juego Monster Boy y el Reino Maldito 19
Figura 4 Captura del juego Super Mario Bros U Deluxe 20
Figura 5 Captura del juego Potata: Fairy Flower 21
Figura 6 Planificación 24
Figura 7 Cronograma de trabajo 28
Figura 8 Captura del juego Donkey Kong Country2: Diddy's Kong Quest 30
Figura 9 Logo PEGI 31
Figura 10 Etiquetas PEGI 31
Figura 11 Logo Godot 33
Figura 12 Logo Source 2 34
Figura 13 Logo Unity 34
Figura 14 Logo Unreal Engine 4 35
Figura 15 Logo Blender 36
Figura 16 Logo 3ds Max 36
Figura 17 Logo Maya 37
Figura 18 Logo Adobe Photoshop 38
Figura 19 Logo Adobe Illustrator 39
Figura 20 Logo GIMP 39
Figura 21 Logo Inkscape 40
Figura 22 Captura de la parte 1 del Mapa Gangplank Galley del juego Donkey Kong Country 2: Diddy's Kong Quest 42
Figura 23 Captura de la parte 2 del Mapa Gangplank Galley del juego Donkeyo Kong Country 2: Diddy's Kong Quest 43
Figura 24 Captura de la parte 3 del Mapa Gangplank Galley del juego Donkeyo Kong Country 2: Diddy's Kong Quest 43
Figura 25 Captura de la parte 4 del Mapa Gangplank Galley del juego Donkeyo Kong Country 2: Diddy's Kong Quest 43
Figura 26 Captura dentro del Level Editor Viewport de Unreal Engine 4 de la primera parte del nivel Entrada en la isla 44
Figura 27 Captura dentro del Level Editor Viewport de Unreal Engine 4 de la segunda parte del nivel Entrada en la isla 44
Figura 28 Captura dentro del Level Editor Viewport de Unreal Engine 4 de la tercera parte del nivel Entrada en la isla 44
Figura 29 Captura dentro del Level Editor Viewport de Unreal Engine 4 de la cuarta parte del nivel Entrada en la isla 45
Figura 30 Captura dentro del Level Editor Viewport de Unreal Engine 4 de la quinta parte del nivel Entrada en la isla 45
Figura 31 Captura dentro del Level Editor Viewport de Unreal Engine 4 de la sexta parte del nivel Entrada en la isla 45
Figura 32 Captura dentro del Level Editor Viewport de Unreal Engine 4 de la septima parte del nivel Entrada en la isla 46
Figura 33 Assets provenientes del Platformer Starter Pack del usuario Platfunner en el bazar de Unreal 46

Figura 34 Captura del agua utilizada en los escenarios 46
Figura 35 Captura del modelo de un barril 47
Figura 36 Modelo de Diddy Kong 47
Figura 37 Modelo del orco 48
Figura 38 Modelo de la caracola 48
Figura 39 Modelo del dragón 49
Figura 40 Captura desde el editor de Unreal de diferentes objetos del juego 49
Figura 41 Captura desde el editor de Unreal del modelo del trampolín 50
Figura 42 Captura desde el editor de Unreal del efecto de partícula del lanzador 50
Figura 43 Sprites del juego Donkey Kong Country 2: Diddy's Kong Quest 51
Figura 44 Captura de los elementos de los elementos del HUD desde el Unreal Editor 51
Figura 45 Captura del menú principal del juego final 52
Figura 46 Captura del menú de opciones del juego final 53
Figura 47 Captura del menú de controles del juego final 53
Figura 48 Captura del menú de resolución del juego final 54
Figura 49 Captura del menú de sonido del juego final 54
Figura 50 Captura del menú de nivel completado 1 del juego final 55
Figura 51 Captura del menú de nivel completado 2 del juego final 55
Figura 52 Captura del menú de final de partida del juego final 56
Figura 53 Jerarquía de retos 57
Figura 54 Jerarquía del reto 'Encuentro con enemigo' 58
Figura 55 Flowchart de Donkey Kong Country Remix 63
Figura 56 Captura de lanzamiento del juego desde Epic Games Launcher 65
Figura 57 Captura de opciones de nuevo proyecto del motor Unreal Engine 4 66
Figura 58 Captura del editor Unreal Engine 4 con el nuevo proyecto haciendo uso de la plantilla SideScroller 66
Figura 59 Captura del Editor de Unreal Engine 4 mostrando el mesh de Diddy Kong 68
Figura 60 Captura del Editor de Unreal Engine 4 mostrando el mesh de un cocodrilo enemigo 68
Figura 61 Captura del Editor de Unreal Engine 4 mostrando el mesh de una araña enemiga 69
Figura 62 Captura del Editor de Unreal Engine 4 mostrando el mesh de la Rata enemiga 69
Figura 63 Captura de Imagen indicativa de Mixamo 70
Figura 64 Captura de la aplicación Mixamo con el modelo de Diddy Kong y el esqueleto generado 70
Figura 65 Captura de la aplicación Mixamo con el modelo de Diddy Kong y el esqueleto generado 71
Figura 66 Captura del Editor de Unreal Engine 4 mostrando el mesh del Dragón 72
Figura 67 Captura del Editor de Unreal Engine 4 mostrando el mesh del Orco 72
Figura 68 Captura del Editor de Unreal Engine 4 mostrando el mesh del Caparazón 73
Figura 69 Componentes del blueprint del personaje principal Diddy Kong 73
Figura 70 Detalles del componente SpringArm 74
Figura 71 Ajustes del componente SpringArm 74
Figura 72 Ajustes del componente SideViewCamera 75
Figura 73 Captura de la configuración de Inputs 76
Figura 74 Captura de la lógica del Movimiento básico 77
Figura 75 Captura de la lógica del salto 78
Figura 76 Captura de la lógica del dash parte 1 78
Figura 77 Captura de la lógica del dash parte 2 79
Figura 78 Captura de la lógica del dash parte 3 79

Figura 79 Captura de la lógica del dash parte 4 80
Figura 80 Captura de la lógica de colgarse parte 1 80
Figura 81 Captura de la lógica del colgarse parte 2 81
Figura 82 Captura de la lógica de bailar 81
Figura 83 Captura del Unreal Editor con la vista del nivel 82
Figura 84 Captura de la exportación de una malla desde 3ds Max 83
Figura 85 Captura de la ventana de importación de un archivo .fbx en Unreal Engine 4 83
Figura 86 Captura del Unreal Editor en la ventana del mesh de un modelo 84
Figura 87 Captura del Epic Games Launcher 84
Figura 88 Captura desde el editor de Unreal de una parte del terreno del nivel 85
Figura 89 Captura del barril desde el editor de Unreal 86
Figura 90 Parámetros del Light Source 86
Figura 91 Blueprint del Sound cue de la música de fondo 87
Figura 92 Blueprint del Sound cue de la música ambiental 87
Figura 93 Blueprint del nivel con la inicialización de ambas pistas de sonido 88
Figura 94 Nodo Play Sound 2D del blueprint Trampolín 88
Figura 95 Captura de la configuración del componente InterpToMovement 89
Figura 96 Captura del blueprint de asignación de los puntos A y B cómo Control Points en la Interpolación del orco 90
Figura 97 Captura del blueprint de la lógica del comportamiento del orco 90
Figura 98 Captura del apartado viewport del blueprint del orco 91
Figura 99 Captura del blueprint de la lógica de colisión de la caja 91
Figura 100 Captura del blueprint de la lógica de colisión de la cápsula parte 1 92
Figura 101 Captura del blueprint de la lógica de colisión de la cápsula parte 2 93
Figura 102 Captura del blueprint de la lógica de colisión de la cápsula parte 3 93
Figura 103 Captura del blueprint de la lógica de colisión de la cápsula parte 4 94
Figura 104 Captura del blueprint de la lógica de colisión de la cápsula parte 5 94
Figura 105 Captura de la composición de la animación de la caracola en estado reposo 95
Figura 106 Captura de la composición de la animación de la caracola en estado defensivo 96
Figura 107 Captura del viewport del blueprint de la caracola 96
Figura 108 Blueprint con la lógica que vincula las cápsulas de colisión a partes del esqueleto 97
Figura 109 Parte del blueprint del segundo tipo de caracola con la lógica que se ejecuta después de ser derrotada 97
Figura 110 Captura del viewport del blueprint del dragón 98
Figura 111 Captura del viewport del blueprint del dragón 98
Figura 112 Captura del blueprint del dragón de la lógica de colisión de la cápsula 99
Figura 113 Captura del viewport del blueprint de la banana 100
Figura 114 Captura del blueprint de la lógica de la banana 100
Figura 115 Captura del viewport del blueprint de las monedas KONG 101
Figura 116 Captura del blueprint de la lógica de la moneda K 102
Figura 117 Captura del viewport del blueprint de la moneda de final de nivel 102
Figura 118 Captura del blueprint de la lógica de la moneda de final de nivel parte 1 103
Figura 119 Captura del blueprint de la lógica de la moneda final de nivel parte 2 103
Figura 120 Captura del viewport del blueprint del gancho 104
Figura 121 Captura del blueprint de la lógica del gancho parte 1 104
Figura 122 Captura del blueprint de la lógica del gancho parte 2 105
Figura 123 Captura del personaje agarrado en la ubicación final de la interacción 106
Figura 124 Captura del viewport del blueprint del trampolín 106

Figura 125 Captura del blueprint de la lógica del trampolín 107
Figura 126 Captura del viewport del blueprint del lanzador 107
Figura 127 Captura del blueprint de la lógica del lanzador 108
Figura 128 Captura del blueprint de la lógica añadida en el lanzador con trigger 108
Figura 129 Captura dentro del Level Editor Viewport de Unreal donde se encuentra el lanzador con trigger 109
Figura 130 Captura del blueprint de la lógica del checkpoint 109
Figura 131 Captura del blueprint de la lógica de la zona de muerte parte 1 110
Figura 132 Captura del blueprint de la lógica de la zona de muerte parte 2 110
Figura 133 Lógica del Level blueprint del Level 1 parte 1 111
Figura 134 Lógica del Level blueprint del Level 1 parte 2 111
Figura 135 Lógica del Level blueprint del Level 1 parte 3 112
Figura 136 Lógica del Game State blueprint parte 1 112
Figura 137 Lógica del Game State blueprint parte 2 113
Figura 138 Lógica del Game State blueprint parte 3 113
Figura 139 Lógica principal del Animation Blueprint 1 114
Figura 140 Lógica principal del Animation Blueprint 2 114
Figura 141 Máquina de estados del Animation Blueprint 115
Figura 142 Lógica del estado de animación Hanging 115
Figura 143 Lógica del estado de animación Walk/Run 115
Figura 144 Viewport de la configuración del nodo Blendspace Player del editor de Unreal 116
Figura 145 Lógica de la regla de transición del estado JumpStart a Hanging 116
Figura 146 Lógica de la regla de transición del estado Hanging a JumpStart 117
Figura 147 Lógica de la regla de transición del estado JumpStart a JumpLoop 117
Figura 148 Pestaña Designer del Widget Blueprint del HUD principal 118
Figura 149 Función para el contador de bananas 118
Figura 150 Función para el contador de vidas 119
Figura 151 Pestaña Designer del Widget Blueprint del HUD secundario 119
Figura 152 Función para la moneda K 120
Figura 153 Viewport del editor de Unreal del nivel MainMenu 120
Figura 154 Lógica principal del Level Blueprint del MainMenu 121
Figura 155 Pestaña Designer del Widget Blueprint del MainMenu 121
Figura 156 Lógica de los botones del Widget Blueprint del MainMenu 122
Figura 157 Menú principal del juego en ejecución 122
Figura 158 Pestaña Designer del Widget Blueprint del menú Options 123
Figura 159 Lógica de los botones del Widget Blueprint de Options parte 1 123
Figura 160 Lógica de los botones del Widget Blueprint de Options parte 2 124
Figura 161 Pestaña Designer del Widget Blueprint del menú Controls 125
Figura 162 Pestaña Designer del Widget Blueprint del menú Resolution 125
Figura 163 Lógica de los botones del Widget Blueprint de Resolution 126
Figura 164 Pestaña Designer del Widget Blueprint del menú Sound 126
Figura 165 Lógica de las sliders del Widget Blueprint de Sound 127
Figura 166 Pestaña Designer del Widget Blueprint del menú LevelCompleted 128
Figura 167 Función para el tiempo total 128
Figura 168 Función para las bananas totales 128
Figura 169 Pestaña Designer del Widget Blueprint del menú LevelCompleted2 129
Figura 170 Pestaña Designer del Widget Blueprint del menú GameOver 130

Índice de Tablas

Tabla 1 Cuadro de autoevaluación 17
Tabla 2 Modelo de tabla de los paquetes de trabajo 25
Tabla 3 Documentación 25
Tabla 4 Análisis de requisitos 26
Tabla 5 Estética 26
Tabla 6 Mecánicas 27
Tabla 7 Desarrollo tecnológico y testing 27
Tabla 8 Enemigos 60
Tabla 9 Objetos 60

1. Introducción, objetivos y tipología del proyecto

1.1 Introducción

En esta memoria se explica todo el proceso tanto de análisis, diseño y desarrollo del prototipo del videojuego ***Donkey Kong Country Remix***.

En este documento se explican todos los pasos que se han llevado a cabo para poder realizar el proyecto de manera detallada, además de mencionar las dificultades que han ido surgiendo durante el desarrollo, las soluciones aplicadas y las tomas de decisiones realizadas en diferentes puntos del proyecto.

Antes de entrar en más detalles, se explicará de manera general que concepto hay detrás de este videojuego. ***Donkey Kong Country Remix*** es un prototipo de un posible remake que podría tener el juego clásico *Donkey Kong Country 2: Diddy's Kong Quest (1995).* Tanto el juego desarrollado cómo el clásico pertenecen al género de plataformas con desplazamiento lateral, dónde la jugabilidad es el pilar más importante. Por ese motivo, una de las motivaciones principales del proyecto ha sido traer de vuelta uno de los grandes títulos del género utilizando las nuevas tecnologías actuales para lograr cosas que antes no eran posibles de hacer.

Figura 1 Captura ingame de Donkey Kong Country Remix

1.2 Objetivos

1.2.1 Objetivo principal del Trabajo de Fin de Grado

El objetivo del proyecto se basa en la creación del prototipo de un remake del videojuego Donkey Kong Country 2 (1995) utilizando el motor de videojuegos Unreal Engine 4, y centrando todo su desarrollo en el sistema de scripting visual Blueprints que ofrece el motor. Con el objetivo mencionado se pretende seguir las bases del juego, junto a un rediseño y mejora tanto jugable como visual gracias a las características tecnológicas que nos ofrezca el motor. Para ello, se necesitará analizar las particularidades que ofrece el motor, y aprender tanto su funcionamiento de desarrollo por Blueprints, como el pipeline necesario para la inclusión de assets.

1.2.2 Desglose de Objetivos

En este apartado se especifican los objetivos del TFG en tareas más específicas:

1. Realizar un análisis de diseño y requisitos del videojuego.
2. Cubrir los aspectos de pipeline necesario para la importación de assets en el motor desde software externo a Unreal Engine 4.
3. Diseño y creación de niveles utilizando las herramientas que ofrece el editor de Unreal.
4. Implementar la lógica del juego utilizando el sistema de scripting visual Blueprints.
5. Diseño y creación del sistema de Interfaz de usuario del juego.
6. Implementación de sonidos y efectos visuales del juego.
7. Realizar y analizar pruebas de testing para balancear dificultad, mecánicas, entendimiento, etc.

1.3 Tipología del proyecto

Las tareas que se deben realizar para cumplir el objetivo de este proyecto siguen la distribución que aparece en la Tabla 1.

1. **Estética**: Se seguirán los cánones de la estética del juego el cual tomamos referencia y se adaptarán al diseño de escenarios, objetos y selección de los modelos de diferentes personajes con los que contará el proyecto
2. **Narrativa**: Se mantendrá la narrativa del juego original *Donkey Kong Country 2: Diddy's Kong Quest* adaptando el guion en los cambios de personaje que se realizarán durante el desarrollo del prototipo del juego.
3. **Mecánicas**: Diseñar, implementar y balancear tanto las diferentes mecánicas y acciones que dispondrá el jugador cómo las interacciones con los diferentes enemigos y objetos que aparecerán en el mundo. Es una de las partes importantes del proyecto.
4. **Desarrollo tecnológico**: El desarrollo tecnológico e implementación del proyecto se realizará con el motor de juegos Unreal Engine 4 y su sistema basado en scripting visual Blueprints. El prototipo ira dirigido para la plataforma de Pc. No se descarta en un futuro

llevarlo a plataformas de consola cómo Nintendo Switch, PlayStation 4, etc. Es la parte más importante del proyecto.

Estética	10
Narrativa	0
Mecánicas	30
Desarrollo Tecnológico	60

Tabla 1 Cuadro de autoevaluación

2. Estudio de viabilidad

Para analizar la viabilidad de nuestro proyecto, se ha hecho un estudio de mercado enfocado a los juegos de plataformas publicados este último año (2020) y el anterior (2019). De esta forma podremos ver y entender la tendencia que está siguiendo este género de juegos en la actualidad, y si hay un hueco en el mercado para nuestro proyecto. Además de esto, se han hecho análisis desde el punto de vista tecnológico, económico y legal.

2.1 Estudio de mercado

Para llevar a cabo el estudio de mercado, se ha hecho una búsqueda en diferentes plataformas con las que cuentan tienda digital en PC y en la plataforma de venta de juegos online de Nintendo (empresa creadora del juego original *Donkey Kong Country 2: Diddy's Kong Quest*).

A partir de esta búsqueda se ha hecho un análisis de los resultados obtenidos para poder sacar una conclusión en base a ello.

2.1.1 Búsquedas realizadas

Como se menciona al inicio del apartado, la búsqueda se ha centrado en juegos del mismo género al de nuestro proyecto y que cuenten con ciertas características presentes en nuestro proyecto:

- Desplazamiento lateral.
- Estética cartoon (se excluye el pixel art).
- Apto para una gran parte de público.

Para realizar la búsqueda de coincidencias se han utilizado las palabras clave y las etiquetas de manera individual y combinadas entre ellas:

Plataformas, desplazamiento lateral, aventura.

La búsqueda de coincidencias se ha llevado a cabo en las tiendas de las plataformas de Steam, Epic Games, Origin y Nintendo eShop.

2.1.2 Resultados obtenidos

Tras hacer la búsqueda en las diferentes plataformas de venta y analizar los diferentes juegos encontrados a través de las palabras clave y etiquetas mencionadas, se han encontrado los siguientes títulos que contienen las características requeridas anteriormente:

- **Toki**

Figura 2 Captura del juego Toki

Toki es un juego que se publicó en 1989 para máquinas recreativas. Tras 30 años se implementó una versión totalmente nueva, adaptada a la generación actual.

Es un juego lanzado inicialmente para Nintendo Switch en 2018 y posteriormente en Steam el 7 de junio de 2019. Tiene una clasificación PEGI 7 lo que lo hace apto para un gran público a pesar de su dificultad, que también encontramos en diferentes juegos de los años 80 y 90.

- **Monster Boy y el Reino Maldito**

Cómo en el caso anterior, este título también fue lanzado para Nintendo Switch en 2018, pero hasta el 25 de julio de 2019 no estuvo disponible en Steam.

Figura 3 Captura del juego Monster Boy y el Reino Maldito

Este título toma lo mejor de varios juegos clásicos y los adapta a nuestra época con mejoras tanto en jugabilidad como en el apartado gráfico.

Tiene una clasificación PEGI 7, de género de acción y aventuras en gráficos 2D con desplazamiento lateral.

- **New Super Mario Bros U Deluxe**

  Exclusivo de Nintendo Switch, este juego fue lanzado el 11 de enero de 2019 con una catalogación PEGI 3, accesible para todas las edades.

Figura 4 Captura del juego Super Mario Bros U Deluxe

De la mítica saga *Super Mario Bros*, este título de Super Mario mantiene su estilo clásico con una gran cantidad de distintos niveles en 2D de desplazamiento lateral.

- **Potata: Fairy Flower**

  A diferencia de los dos primeros títulos presentados, este juego salió el 17 de diciembre de 2019 en la plataforma de Steam. Su aparición a la plataforma Ninentdo Switch se ha realizado recientemente en la fecha 6 de junio de 2020.

Figura 5 Captura del juego Potata: Fairy Flower

  Potata se trata de un juego de puzles, plataformas y aventuras en el que el jugador se adentrará en un largo viaje por tal ayudar a salvar su aldea. Con clasificación PEGI 3 lo hace apto para todo tipo de público.

Además de los juegos presentados, se han encontrado algunos más que se asemejan a las características detalladas de la búsqueda, cómo pueden ser *Ori and the Bind Forest: Definitive Edition o Megabyte Punch*. Sin embargo, ambos títulos encontrados en la plataforma de Nintendo eShop, ya fueron lanzamos varios años antes para PC, por ese motivo no se han incluido en los lanzamientos recientes.

Hay que destacar algo muy importante durante la realización de la búsqueda hecha en las diferentes plataformas, y es que en todas ellas se ha observado una fuerte tendencia a que la gran mayoría de títulos encontrado con las palabras clave y etiquetas descritas en los últimos dos años, hacen uso de gráficos con estilo *Pixel Art*. Este hecho es tan claro y contundente, que tanto en las plataformas de Epic Games cómo Origin, ha resultado imposible encontrar un juego con las características marcadas que no hiciera uso de esta técnica gráfica.

2.1.3 Conclusiones

En los dos últimos años, los juegos lanzados que comparten el género y las características del proyecto son en gran mayoría con gráficos *Pixel Art*, aspecto que marca una gran distinción tanto

en el aspecto grafico como el jugable. Por ese motivo, y la clara aceptación que han tenido los juegos analizados en el apartado anterior que comparten cierta similitud al nuestro, nos damos cuenta que la propuesta a desarrollar tiene un espacio en el mercado.

2.2 Viabilidad tecnológica

Para poder realizar una evaluación de la viabilidad tecnológica que requiere el desarrollo del proyecto, se ha hecho una lista de las necesidades tecnológicas necesarias para poder llevarlo a cabo:

- Un motor de juegos.
- Acceso a diferentes assets compatibles con el motor de juegos a utilizar.
- Un programa de modelado 3D.
- Un programa de rigging y animación.
- Un programa de diseño gráfico.
- Un computador que nos permita hacer uso del software mencionado y que nos permita hacer las pruebas de testeo.

La universidad dispone de licencias estudiantiles para el uso del tipo de software que se necesita. Por la parte de hardware, disponemos de maquinaria con los requisitos suficientes para poder llevar a cabo tanto el uso del software para desarrollo e implementación, cómo para realizar las pruebas de testeo. El único requerimiento que se puedan tener más problemas es el acceso a los assets necesarios. Por lo tanto, desde un punto de vista tecnológico, el proyecto a llevar a cabo es viable.

2.3 Viabilidad económica

En cuanto a la viabilidad económica requerida a las necesidades del proyecto son las que van asociadas al software y hardware mencionado en el apartado anterior. Dado a que se disponen de licencias proporcionadas por la universidad o porque el software es gratuito, no tenemos ningún problema a nivel económico. Por la parte de hardware, se dispone del equipo necesario para llevar a cabo el proyecto. El único gasto que pueda producirse, vendrá dado por el acceso a ciertos assets requeridos durante el desarrollo (se estima un gasto en torno a 10-40 €).

2.4 Viabilidad legal

En cuanto a la viabilidad legal del proyecto, no se trata con información del usuario por lo que no tenemos un tratamiento con datos personales ni confidenciales. Por lo tanto, no es necesario aplicar el reglamento general de protección de datos ni tomar medidas especiales en este aspecto.

Por parte de los derechos de la propiedad intelectual del material tanto visual como auditivos utilizados en el proyecto, la mayoría de son de creación propia, comprados o con licencia de uso

gratuito. Sin embargo, como se utiliza cierto material que no, hay que puntualizar que la finalidad de esta versión del proyecto es de ámbito académico y en ningún momento para uso comercial ni para fines económicos.

2.5 Conclusión

Después de haber analizado la viabilidad desde el punto de vista tecnológico, económico y legal podemos decir que el desarrollo del proyecto tiene sentido y es viable.

3. Planificación

En este apartado se define la planificación del proyecto indicando como se ha estructurado su desarrollo en diferentes paquetes de trabajo que indican las tareas principales que se deben realizar y su temporalización.

3.1 Paquetes de trabajo

Para el desarrollo del proyecto, se ha ido siguiendo la planificación que se muestra en la Figura 6 donde se muestran los diferentes paquetes de trabajo principales con las diferentes tareas asociadas a cada uno de ellos. La estructura del proyecto en cuanto a paquetes de trabajo está formada por cinco paquetes principales que son: documentación, análisis de requisitos, estética, mecánicas y desarrollo tecnológico y testing. Cada uno de estos paquetes cuenta con unas tareas específicas asociadas.

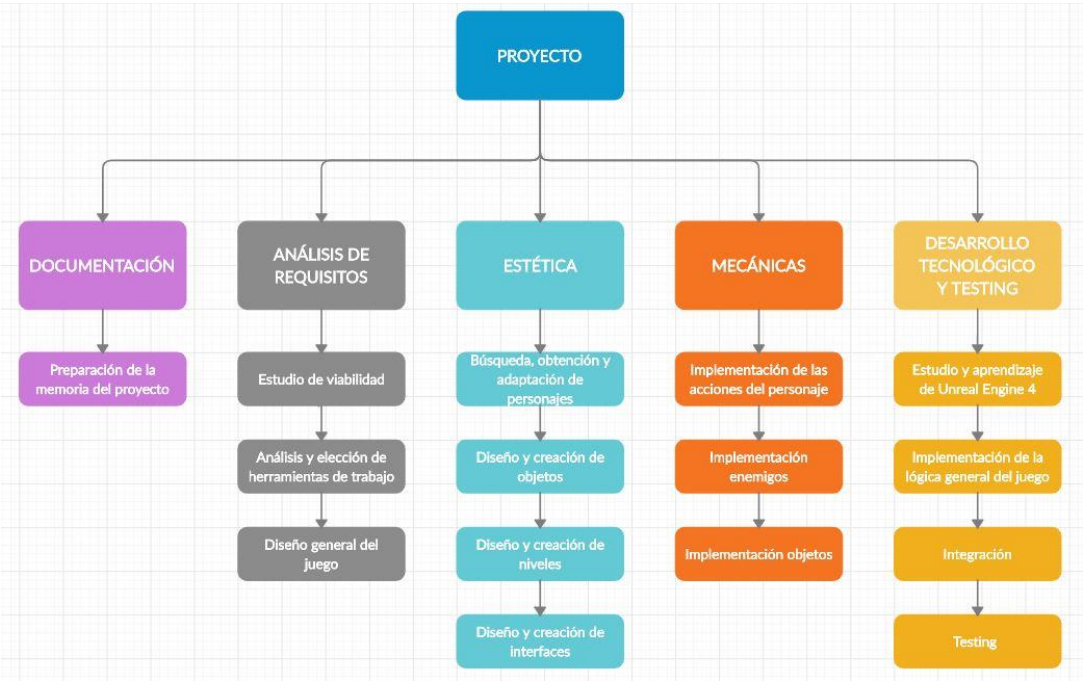

Figura 6 Planificación

3.2 Descripción de paquetes

A continuación, se presentará para cada uno de los paquetes principales de trabajo su descripción, las tareas a realizar y el tiempo previsto para la ejecución de cada paquete.

Está información se presentará siguiendo la tabla que se muestra en la Tabla 2.

NOMBRE DEL PAQUETE	
DESCRIPCIÓN	
TAREAS A REALIZAR	
TEMPORALIZACIÓN	

Tabla 2 Modelo de tabla de los paquetes de trabajo

3.2.1 Documentación

NOMBRE DEL PAQUETE	DOCUMENTACIÓN
DESCRIPCIÓN	Recolección de toda la información y acciones hechas durante el desarrollo del proyecto
TAREAS A REALIZAR	Memoria del proyecto
TEMPORALIZACIÓN	3 meses

Tabla 3 Documentación

3.2.2 Análisis de requisitos

NOMBRE DEL PAQUETE	ANÁLISIS DE REQUISITOS
DESCRIPCIÓN	Análisis de los requisitos necesarios para llevar a cabo el desarrollo del juego
TAREAS A REALIZAR	Estudio de viabilidad Análisis y elección de las herramientas de trabajo Diseño general del juego
TEMPORALIZACIÓN	1 semana

Tabla 4 Análisis de requisitos

3.2.3 Estética

NOMBRE DEL PAQUETE	ÉSTETICA
DESCRIPCIÓN	Diseño y creación de la estética global del juego
TAREAS A REALIZAR	Búsqueda, obtención y adaptación de personajes Diseño y creación de objetos Diseño y creación de niveles Diseño y creación de interfaces
TEMPORALIZACIÓN	2 meses

Tabla 5 Estética

3.2.4 Mecánicas

NOMBRE DEL PAQUETE	MECÁNICAS
DESCRIPCIÓN	Implementación de la lógica de los elementos del juego
TAREAS A REALIZAR	Implementación de las acciones del personaje Implementación de enemigos Implementación de objetos
TEMPORALIZACIÓN	2 meses

Tabla 6 Mecánicas

3.2.5 Desarrollo tecnológico y testing

NOMBRE DEL PAQUETE	DESARROLLO TECNOLÓGICO Y TESTING
DESCRIPCIÓN	Desarrollo del juego llevado a cabo en el motor Unreal Engine 4 junto a la integración de todos los elementos.
TAREAS A REALIZAR	Estudio y aprendizaje de Unreal Engine 4 Implementación de la lógica general del juego Integración Testing
TEMPORALIZACIÓN	3 meses

Tabla 7 Desarrollo tecnológico y testing

3.2 Cronograma

En este punto se ha hecho un cronograma donde se puede ver una visión global de la temporalización de paquetes de trabajo asociada al proyecto (ver Figura 7).

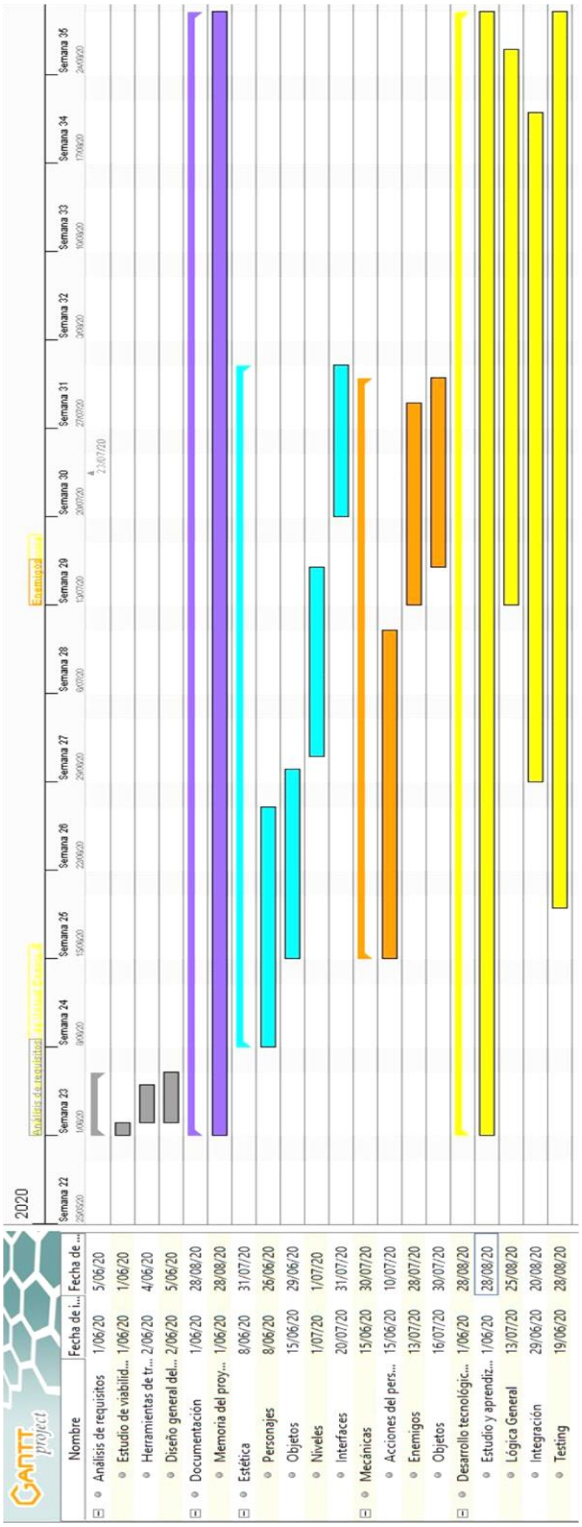

Figura 7 Cronograma de trabajo

3.3 Metodología de trabajo

Por ciertas circunstancias que se explicarán en el apartado 9. Conclusiones, el tiempo que se ha tenido para el desarrollo del proyecto ha sido muy limitado (3 meses). Por ese motivo, una vez hecho el análisis de requisitos, se ha aplicado una metodología de trabajo en que se iban realizando las diferentes tareas de los diferentes paquetes de trabajo en paralelo siempre que las tareas no tuvieran ciertas dependencias entre paquetes. Este paralelismo se puede observar en el cronograma anterior que contiene la Figura 7.

Por otra banda, se ha ido documentando el proceso de realización de las tareas a medida que se iban completando, anotando tanto las dificultades cómo las soluciones llevadas a cabo en cada una de ellas.

Todo este proceso se ha ido supervisando con reuniones semanales junto al tutor del proyecto para tratar de aplicar las correcciones necesarias lo antes posible y así optimizar el tiempo de desarrollo lo máximo posible.

4. Marco de trabajo y conceptos previos

4.1 Referencia

La idea de nuestro videojuego surge debido a la reciente atención por parte del público sobre traer de vuelta juegos clásicos de los años 90 a las nuevas generaciones con sus respectivas mejoras tanto, gráficas cómo técnicas y jugables. Tal cómo hemos visto anteriormente en el apartado 2, hay un hueco en el mercado para el desarrollo de un remake del videojuego de plataformas de desplazamiento lateral *Donkey Kong Country 2: Diddy's Kong Quest* (ver Figura 8).

El objetivo de rehacer este juego clásico publicado en 1995, consiste en traer de vuelta el género de plataformas y desplazamiento lateral olvidado estos últimos años a la nueva generación, ofreciendo mejoras en el apartado gráfico, como el efecto de luces, partículas o modelados en 3D, mejoras en el apartado técnico y jugable con la mejora de las mecánicas ya existentes en el título y la creación de nuevas, para así mantener la esencia del título que tantas horas de diversión nos ha dado, y llevarlo al siguiente nivel.

Figura 8 Captura del juego Donkey Kong Country2: Diddy's Kong Quest

El objetivo del juego trata de ir superando diferentes niveles de forma lineal hasta llegar al final. Cada nivel cuenta con un escenario concreto el cual está repleto tanto de enemigos que deberemos superar para avanzar, como de objetos que tendremos que recolectar para obtener diferentes recompensas. Para poder superar el nivel, deberemos llegar desde el punto de inicio hasta la línea de meta sin morir más veces de las vidas que dispongamos.

Nuestro personaje cuenta con un set de habilidades muy característico del género de plataformas que nos permite superar de manera ágil los diferentes enemigos que habitan en el nivel y a desplazarnos por él de manera rápida y cómoda.

En resumen, nuestro proyecto seguirá los pilares básicos del juego original, pero tomando libertad tanto en el diseño como en la creación, dando como resultado no sólo una versión mejorada del juego, sino un nuevo juego que trae como columna vertebral el estilo clásico del género de plataformas.

4.2 Plataformas

El juego va dirigido para la plataforma de PC, ya que por desgracia no tenemos el acceso a consolas de desarrollo para poder exportar y probar el prototipo en diferentes plataformas como Nintendo Switch, Xbox One o PlayStation 4.

4.3 PEGI

En este apartado se hablará sobre la regularización de nuestro videojuego a partir del sistema PEGI (Pan European Game Information).

Figura 9 Logo PEGI

El sistema PEGI, se trata de un sistema de autorregulación donde la propia empresa desarrolladora del título decide su calificación para que el consumidor sepa qué tipo de contenido puede aparecer y cuál es la edad recomendable para jugarlo. Una vez la empresa lo cataloga, debe pasar por un proceso de comprobación por el cual se entrega una licencia que autoriza el uso de etiquetas PEGI. Estas etiquetas se clasifican en dos tipos:

- Edad recomendada
- Contenido del juego

De esta manera, antes que el consumidor compre un juego, puede ver qué tipo de contenido se encuentra en el título y cuál es la edad mínima recomendable (ver Figura 10). En ningún momento estas etiquetas prohíben la adquisición del producto si no cumples con la edad recomendada, ya que como bien se especifica, es sólo una recomendación.

Figura 10 Etiquetas PEGI

En el caso de nuestro juego, debido a su temática y diseño, podemos utilizar con seguridad un **PEGI 3** y el **PEGI OK**, ya que no contiene ningún elemento de los que podemos ver en la Figura 10.

5. Herramientas de trabajo

Para poder realizar apropiadamente el desarrollo del proyecto se necesita el conjunto de herramientas que veremos a continuación:

5.1 Motor de juegos

Las opciones disponibles para la elección del motor son amplias, ya que existen diferentes motores gráficos con distintas ventajas e inconvenientes según el tipo de juego que se quiere desarrollar. Además, también se ha tenido en cuenta la accesibilidad al uso del motor, ya que no todos ellos son de uso gratuito. Por ese motivo se han listado algunos de ellos que si cumplen esa condición:

- Godot
- Source 2
- Unity 5
- Unreal Engine 4

5.1.1 Godot

Figura 11 Logo Godot

Godot es un motor de videojuegos 2D y 3D multiplataforma, libre y de código abierto, publicado bajo la Licencia MIT (Licencia de software utilizada regularmente en el software libre). También cuenta con un avanzado, independiente y completo motor 2D que le permite entre muchas otras cosas, la posibilidad de mezclar 2D con 3D o a la inversa. Su última versión cuenta con la característica de desarrollar juegos 2.5D.

El motor utiliza un sistema de codificación en el lenguaje de programación GDScript, su lenguaje nativo. También cuenta con soporte para otros lenguajes como C#, C++ y VisualScript.

5.1.2 Source 2

Figura 12 Logo Source 2

El motor de videojuegos Source 2 es el último motor desarrollado por Valve, sucesor del motor original Source.

Esta nueva versión tiene la capacidad de renderizar escenas muy complejas y bien detalladas sufriendo caídas mínimas de cuadros, gracias a su nueva arquitectura y soporte de **Vulkan** (API multiplataforma para desarrollo de aplicaciones con gráficos 3D), permitiendo así un mejor renderizado multinúcleo y 3D más eficiente.

Este motor está disponible de forma gratuita para todos los desarrolladores de contenido.

5.1.3 Unity 5

Figura 13 Logo Unity

El motor de videojuegos Unity tiene la particularidad de poseer una interfaz sencilla de entender y cuenta con un sistema de integración multiplataforma que permite exportar juegos para casi todo tipo de plataforma, lo que lo hace el mejor motor para desarrollo 3D destinados a la plataforma Android. Cuenta con herramientas de compresión que permiten que los videojuegos no sean tan pesados y evita un consumo de recursos excesivo.

Unity es compatible con diferentes softwares de modelado y animación 3D, como pueden ser 3ds Max, Maya, Blender y otros, ofreciendo un soporte de lectura de archivos exportados muy bueno.

Unity cuenta con una versión gratuita bastante completa, aunque su versión pro es de pago. Sin embargo, existen licencias de estudiante para su uso no comercial.

5.1.4 Unreal Engine 4

Figura 14 Logo Unreal Engine 4

Unreal Engine 4 cuenta con unas grandes capacidades gráficas, cómo pueden ser sus funcionalidades avanzadas en iluminación dinámica o su sistema de partículas que es capaz de manejar un número muy grande de ellas a la vez en una sola escena.

Además de contar con un sistema de scripting por C++, cuenta con un sistema de scripting gráfico por Blueprints que te da la opción de realizar juegos completos sin necesidad de programar en C++.

Su licencia en base a su uso para la producción comercial de videojuegos es del 5% de las ganancias por título a partir de los primeros 300 dólares americanos por cada cuatrimestre.

5.1.4 Conclusión

De los motores presentados, algunos los hemos utilizado durante el grado como es el caso de Godot, Unity 5 y Unreal Engine 4. Sin embargo, desde el inicio del proyecto se ha tenido claro que el motor a utilizar iba a ser Unreal Engine 4 por las siguientes razones:

- Es un motor que hemos visto, estudiado y trabajado en una de las asignaturas del grado. No obstante, durante las prácticas de Unreal Engine 4, fui el encargado del modelado, texturización, rigging y animación de los personajes. Por ese motivo, se tiene especial interés en aprovechar este proyecto en el aprendizaje del motor y la obtención de un producto final sólido.
- Es gratuito y no tiene limitaciones de software para el desarrollo no comercial.
- Tiene gran compatibilidad con software externo de modelado 3D, animación, etc.
- Consta con una gran cantidad de documentación y tutoriales para su uso, además de una comunidad muy activa y colaborativa.

5.2 Programa de Modelado 3D

Por lo que hace en programas de modelado 3D, su peso en el desarrollo será ínfimo ya que el proyecto está centrado en el uso del motor como tecnología, el desarrollo de mecánicas y el diseño de niveles. A pesar de ello, es posible que la falta de assets específicos haga que se deba hacer uso de este tipo de software para modelar ciertos objetos del escenario o ítems del juego.

Durante el grado nos han presentado y enseñado a utilizar diferentes programas de modelaje 3D:

5.2.1 Blender

Figura 15 Logo Blender

Blender es un programa multi plataforma dedicado al modelado, iluminación, renderizado, animación y creación de gráficos en 3D. Cuenta con una gran variedad de primitivas geométricas, un sistema de partículas estáticas enfocado a la simulación de cabellos y pelajes y características interactivas para jugos como detección de colisiones, entre otras muchas cosas.

Gracias a ser un software libre y gratuito, ha gozado de la aceptación de muchas animadores independientes y pequeños estudios.

5.2.2 3ds Max

Figura 16 Logo 3ds Max

3ds Max es un programa de creación de gráficos y animación 3D desarrollado por Autodesk.

Gracias a su arquitectura basada en plugin, este programa es uno de los más utilizados en la creación de videojuegos, anuncios de televisión, arquitectura o en películas. También cuenta con una interfaz de usuario muy intuitiva ampliable y personalizable, un elevado control del movimiento humano verosímil en la creación de animaciones 3D y gran capacidad de renderización en 3D entre otros aspectos.

A pesar que este software no es gratuito, cuenta con licencias para uso estudiantil.

5.2.3 Maya

Figura 17 Logo Maya

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

Este software se caracteriza por su potencia y las posibilidades que ofrece en la expansión y personalización de su interfaz y herramientas, dando acceso a la creación de scripts y personalización del paquete. También posee una diversa colección de herramientas para modelado, animación, renderización, simulación de ropa i cabello, simulación de fluidos, etc.

Como en el caso anterior, el software no es gratuito, pero existen licencias para su uso académico.

5.3.3 Conclusión

De estas tres opciones, hemos aprendido a modelar en 3D tanto con Blender como con 3ds Max. Entre tanto, Maya se ha utilizado para hacer rigging y animación, todo y que también exista la posibilidad de modelar.

Por ese motivo, la elección final ha sido utilizar 3ds Max, ya que es el programa con el que más experiencia y comodidad he tenido. A pesar de no ser gratuito, la empresa Autodesk tiene licencias para estudiantes y universidades, con la cual la Universitat de Girona cuenta.

5.3 Programa de animación

Tal como se ha explicado en el apartado anterior, sólo se hará uso de este tipo de programas en situaciones oportunas. En una instancia inicial, los programas candidatos a utilizar tanto para hacer el rigging y animar eran:

- Maya
- 3D Max
- Blender

Las tres alternativas tienen la capacidad de hacer rig y animar, por lo tanto, se puede utilizar uno para todo.

Sin embargo, debido al enfoque que presenta este proyecto en las partes de mecánica y tecnología, se ha encontrado una alternativa para poder agilizar el trabajo en esta parte del desarrollo. La alternativa en cuestión es Mixamo, de la empresa Adobe.

Mixamo es un software gratuito que permite, entre otras cosas, poder importar un modelo 3D, generarle un esqueleto de manera automática, y a partir de ese esqueleto poder exportar diferentes animaciones que proporciona la plataforma.

La limitación está en que la generación automática de esqueleto está basada en modelos humanoides, lo que dará problemas al animar personajes con cola, como ocurre en este proyecto.

5.4 Programa de diseño gráfico y vectorial

Durante el grado nos han presentado diversos programas tanto de diseño gráfico como vectorial, como pueden ser:

5.4.1 Adobe Photoshop

Este programa es uno de los editores de imágenes más populares del mercado actual. Cuenta con una ilimitada de posibilidad en la edición de imágenes, haciéndolo sobresaliente ante los demás editores gráficos competidores. Cuenta con características de edición cómo selección, el uso de capas, fusión de imágenes, cambio de color selectivo, etc.

Este software es utilizado tanto en diseño como en fotografía.

Figura 18 Logo Adobe Photoshop

5.4.2 Adobe Illustrator

Adobe Illustrator es un editor de gráficos vectoriales en forma de taller de arte, que trabaja sobre un tablero de dibujo, destinado a la creación artística de dibujo y pintura para ilustración, creación y diseño de imágenes aplicado a la ilustración técnica o diseño gráfico.

Figura 19 Logo Adobe Illustrator

5.4.3 GIMP

GIMP es un programa de edición de imágenes digitales en forma de mapa de bits, tanto para dibujos como fotografías. Es un programa de software libre y gratuito.

Este software permite el tratado de imágenes por capas, además cuenta con muchas herramientas como pueden ser las de selección, tijeras inteligentes, herramientas de pintado, clonado, entre muchas otras. Además, cuenta con una gran extensión de plugins tanto oficiales cómo creados por otros usuarios que posteriormente se integran al software de manera permanente si pasan ciertas pruebas.

Figura 20 Logo GIMP

5.4.4 Inkscape

Inkscape es un editor de gráficos vectoriales libre y de código abierto, capaz de crear y editar diagramas, líneas, gráficos, logotipos e ilustraciones complejas. Aunque este software está desarrollado principalmente para el sistema operativo GNU/Linux, es una herramienta multiplataforma que posee versiones tanto para Windows como MAC OS X.

Figura 21 Logo Inkscape

5.4.5 Conclusión

Varios de estos programas los hemos practicado durante el transcurso del grado, siendo Adobe Photoshop el programa que hemos escogido para su uso en diversos trabajos, y por lo tanto el más utilizado de todos.

Por ese motivo, se utilizará Adobe Photoshop para la creación de toda la parte de la interfaz de usuario del juego, ya que resultará más sencillo por el simple motivo que tengo conocimientos previos del software, además de la libertad de creación y desarrollo que ofrece.

Está herramienta de adobe no es gratuita, no obstante, existen licencias en la universidad para poder utilizar este programa.

6. Diseño del videojuego

6.1 Narrativa

Tal como se indica en las ponderaciones de peso en los diferentes apartados del proyecto, la narrativa que se trabajará en este proyecto es prácticamente inexistente. Es cierto que existe un contexto narrativo previo, ya que el proyecto se basa en el prototipo de un remake de un juego antiguo existente. Pero no se trabajará durante el desarrollo del juego.

6.1.1 Argumento

El argumento del juego original *Donkey Kong Country 2: Diddy's Kong Quest* trata sobre el secuestro que sufre Donkey Kong por parte de su archienemigo y la aventura que emprende su amigo Diddy Kong para salvarlo.

Donkey Kong es secuestrado por su archienemigo mientras disfrutaba del sol de la playa, dejando éste una nota de rescate para su amigo. Cuando Diddy Kong llega a la playa y encuentra la nota en la que se le pide como rescate entregar todas las bananas conseguidas por él y Donkey Kong, se niega profundamente a aceptar dicho trato y decide emprender una aventura hacia la isla dónde se encuentra preso Donkey Kong.

6.1.2 Localizaciones

La idea del desarrollo completo del juego es seguir un circuito lineal de diferentes localizaciones dentro de la isla donde se encuentra Donkey Kong e ir avanzando dichos lugares hasta llegar a liberarlo. En este proyecto las localizaciones **no** son las mismas que las del juego original, aunque sí que recogen su esencia.

Estas localizaciones seguirían el siguiente orden:

- **Entrada a la isla**: Esta localización hace referencia a la llegada a la isla y el inicio de su aventura hasta rescatar a su amigo. Es una zona con agradable, rodeada de mar y con ciertos aspectos relacionados con la visita regular de embarcaciones.
- **Bosque embrujado**: Un oscuro y tenebroso bosque habitado por esqueletos revividos, fantasmas y otras criaturas horripilantes.
- **Volcán abandonado**: Tras el bosque nos adentramos en este volcán dónde anteriormente se habían construido varias minas para la extracción de carbón y metales preciosos.
- **Pantano putrefacto**: Localización formada por un enorme pantano repleto de residuos contaminantes dónde habitan ciertas criaturas anormales y peligrosas.
- **Guarida del enemigo**: La guarida del enemigo está formada por la construcción de una mazmorra sobre suelo ártico que se tendrá que atravesar hasta llegar a la sala principal, lugar dónde se encuentra su amigo encerrado junto al archienemigo.

Para el desarrollo del prototipo en este proyecto se implementará el primer nivel de la **Entrada en la isla**.

6.2 Estética

Tanto la parte estética cómo la narrativa, van cogidas de la mano, ya que una depende de la otra y viceversa. Por ese motivo, nuestro juego al tratarse de una historia fantástica dónde los protagonistas son dos monos que se dedican a recoger bananas, y sus enemigos se basan en tanto seres fantásticos cómo animales con capacidades sobrehumanas, seguirá una estética cartoon.

A diferencia del juego original hecho en 2D con el uso de sprites, el prototipo que se realizará contará con profundidad. Tanto la localización cómo los personajes y enemigos serán compuestos por modelos en 3D.

6.2.1 Espacio

El espacio del juego estará formado por un mundo general separado en diferentes niveles, los cuales se deberán ir completando en orden para poder acceder al siguiente, siguiendo una estructura lineal. Cada nivel cuenta con un circuito por el cual nos desplazaremos de manera lateral, formado por una serie de plataformas por donde podremos desplazarnos, enemigos los cuales nos deberemos enfrentar y una serie de objetos con los cuales podremos interactuar para obtener diferentes finalidades.

6.2.2 Localizaciones

Como se ha explicado en el apartado 7.2 Localizaciones, se enseñará las diferentes zonas del escenario **Entrada a la isla**, pero antes de ello se mostrará el nivel del juego original en el que se inspiró en las Figuras 22, 23, 24 y 25:

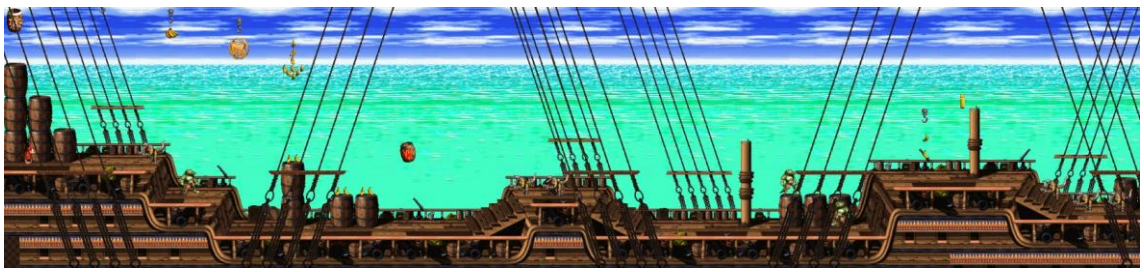

Figura 22 Captura de la parte 1 del Mapa Gangplank Galley del juego Donkey Kong Country 2: Diddy's Kong Quest

Figura 23 Captura de la parte 2 del Mapa Gangplank Galley del juego Donkeyo Kong Country 2: Diddy's Kong Quest

Figura 24 Captura de la parte 3 del Mapa Gangplank Galley del juego Donkeyo Kong Country 2: Diddy's Kong Quest

Figura 25 Captura de la parte 4 del Mapa Gangplank Galley del juego Donkeyo Kong Country 2: Diddy's Kong Quest

A continuación, en las Figuras 26, 27, 28 y 29, veremos el escenario de **Entrada en la isla**:

Figura 26 Captura dentro del Level Editor Viewport de Unreal Engine 4 de la primera parte del nivel Entrada en la isla

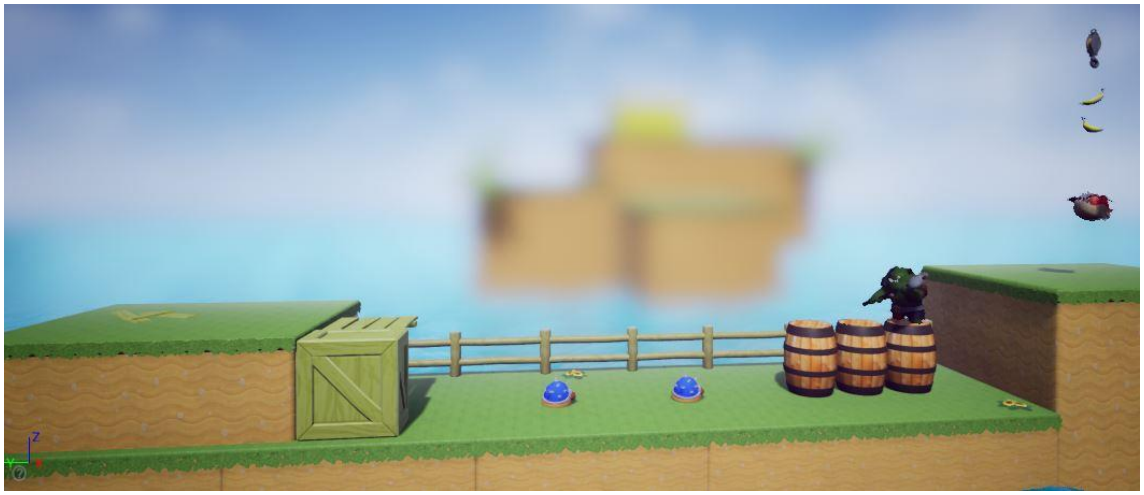

Figura 27 Captura dentro del Level Editor Viewport de Unreal Engine 4 de la segunda parte del nivel Entrada en la isla

Figura 28 Captura dentro del Level Editor Viewport de Unreal Engine 4 de la tercera parte del nivel Entrada en la isla

Figura 29 Captura dentro del Level Editor Viewport de Unreal Engine 4 de la cuarta parte del nivel Entrada en la isla

Figura 30 Captura dentro del Level Editor Viewport de Unreal Engine 4 de la quinta parte del nivel Entrada en la isla

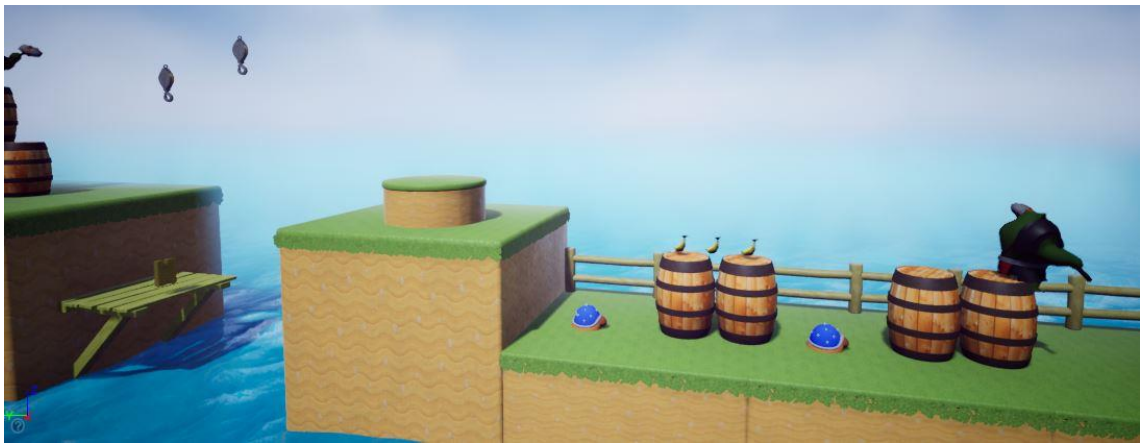

Figura 31 Captura dentro del Level Editor Viewport de Unreal Engine 4 de la sexta parte del nivel Entrada en la isla

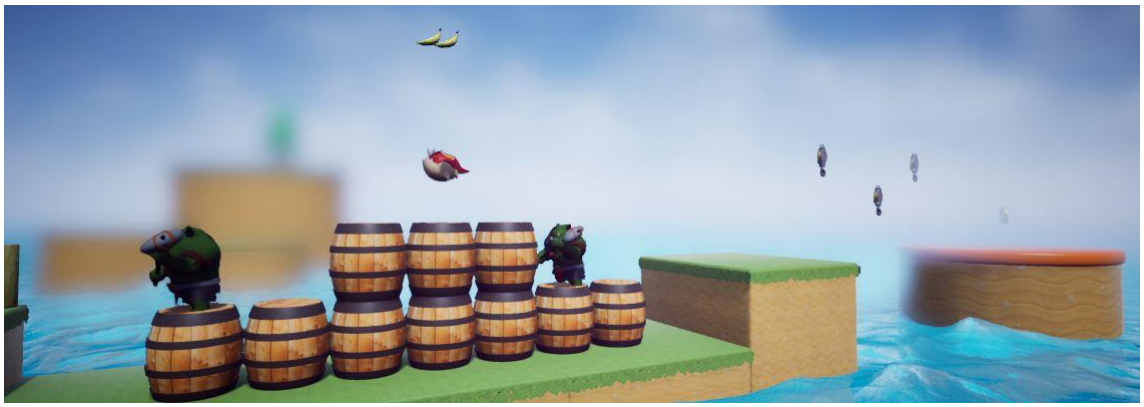

Figura 32 Captura dentro del Level Editor Viewport de Unreal Engine 4 de la septima parte del nivel Entrada en la isla

6.2.3 Elementos que forman el escenario

En este punto se muestran los diferentes assets utilizados en la creación del terreno y decoración. Estos elementos sólo intervienen en el gameplay como terreno jugable y decorativo, son estáticos y no sufren cambios de estado.

Figura 33 Assets provenientes del Platformer Starter Pack del usuario Platfunner en el bazar de Unreal

Figura 34 Captura del agua utilizada en los escenarios

Para poder crear los escenarios del juego, se han utilizado gran parte de los assets que dispone el pack de la Figura 33.

Figura 35 Captura del modelo de un barril

Como no podía faltar el elemento clásico por excelencia en el juego, se ha optado por crear el barril e incluirlo cómo componente del terreno del nivel (ver Figura 35).

6.2.2 Personajes

En este apartado se presentarán el personaje principal y los enemigos que habitan en la localización **Entrada en la isla,** y veremos cómo lucen físicamente dentro de la estética cartoon que comparten.

- **Diddy Kong**: Nuestro personaje principal es un mono de la raza chimpancé proveniente de la selva tropical, de pelaje marrón y ojos negros. Se caracteriza por llevar siempre puesta una gorra de color rojo y una camiseta del mismo color a juego con unas estrellas de color amarillo estampadas en ella (ver Figura 36).

Figura 36 Modelo de Diddy Kong

- **Orco:** El orco es una criatura que habita en la isla y hace guardia en la **Entrada a la isla.** Este ser es originario de la isla tiene una musculatura muy fuerte, piel verdosa y áspera, ojos amarillos, dientes grandes y puntiagudos, orejas en punta y lleva consigo una vestimenta con partes metálicas que le sirven como armadura (ver Figura 37).

Figura 37 Modelo del orco

- **Caracola**: La caracola proviene del mar que rodea la isla. Tiene una coraza de color azul con unas púas que la protegen cuando se siente amenazada. Tiene un enorme ojo de color azulado que siempre se encuentra medio cerrado debido a su somnolencia (ver Figura 38).

Figura 38 Modelo de la caracola

- **Dragón**: El dragón es de procedencia desconocida. Tiene un aspecto parecido al de un lagarto, pero con forma esférica. La parte superior de su cuerpo está cubierta de escamas rojas, y la inferior por una piel rugosa de color crema. Cuenta con unos dientes en forma de sierra, unos pinchos puntiagudos ubicados entre sus ojos amarillos y sus enormes cejas negras, y unas alas de color naranja un tanto pequeñas en comparación con el volumen de su cuerpo (ver Figura 39).

Figura 39 Modelo del dragón

6.3 Objetos

Los objetos se encuentras repartidos a lo largo de los diferentes escenarios. En este punto se enseñarán los modelos de los objetos existentes en el juego.

Figura 40 Captura desde el editor de Unreal de diferentes objetos del juego

Figura 41 Captura desde el editor de Unreal del modelo del trampolín

Figura 42 Captura desde el editor de Unreal del efecto de partícula del lanzador

En la Figura 40 nos encontramos con las diferentes monedas del juego y el gancho, y en las Figuras 40 y 41, con el trampolín y el lanzador. Siguiendo el orden de izquierda a derecha y orden de figuras tenemos:

- **Banana**: Elemento de recolección y bonificación que nos encontraremos ubicado a lo largo de los escenarios.
- **Monedas KONG**: Las forman las 4 monedas con forma de las diferentes letras que forman la palabra KONG, únicas por cada nivel del juego.
- **Moneda de final de nivel**: Moneda con forma de estrella ubicada al final de cada nivel.
- **Gancho**: El gancho tiene la forma habitual de un gancho perteneciente a las obras con una polea encima.
- **Trampolín**: El trampolín se asemeja a una cama elástica que resalta por su color y su material blando y esponjoso.
- **Lanzador**: El lanzador tiene la particularidad que el modelo utilizado no es un mesh, sino un efecto de partícula que simula un aro de propulsión.

6.4 Interfaz de Usuario

En este apartado se presentarán las interfaces de usuario del juego, formadas por el HUD, el menú principal, menú de opciones, etc.

6.4.1 HUD

El HUD (heads-up display), se basa en un conjunto de información visual en 2D, repartida sobre la pantalla del juego, que aporta información útil y necesaria al jugador.

Figura 43 Sprites del juego Donkey Kong Country 2: Diddy's Kong Quest

Figura 44 Captura de los elementos de los elementos del HUD desde el Unreal Editor

Para nuestro juego hemos utilizado la fuente ***GillSansStd Ultra BoldCond*** tanto para las letras, signos especiales y números.

El HUD nos muestra de forma permanente en el transcurso del nivel la información siguiente:

- El número de bananas recogidas
- Las vidas restantes del jugador
- Las monedas Kong recogidas (estas monedas irán apareciendo en nuestro HUD a medida que vayamos recogiendo cada una).

6.4.2 Menús

Los menús del juego están formados por el menú principal, el menú de opciones junto a sus submenús, el menú de pausa y los menús de nivel completado y de final de partida. Todos los menús comparten la misma estética cartoon, haciendo uso de componentes cómo botones, sprites y sliders.

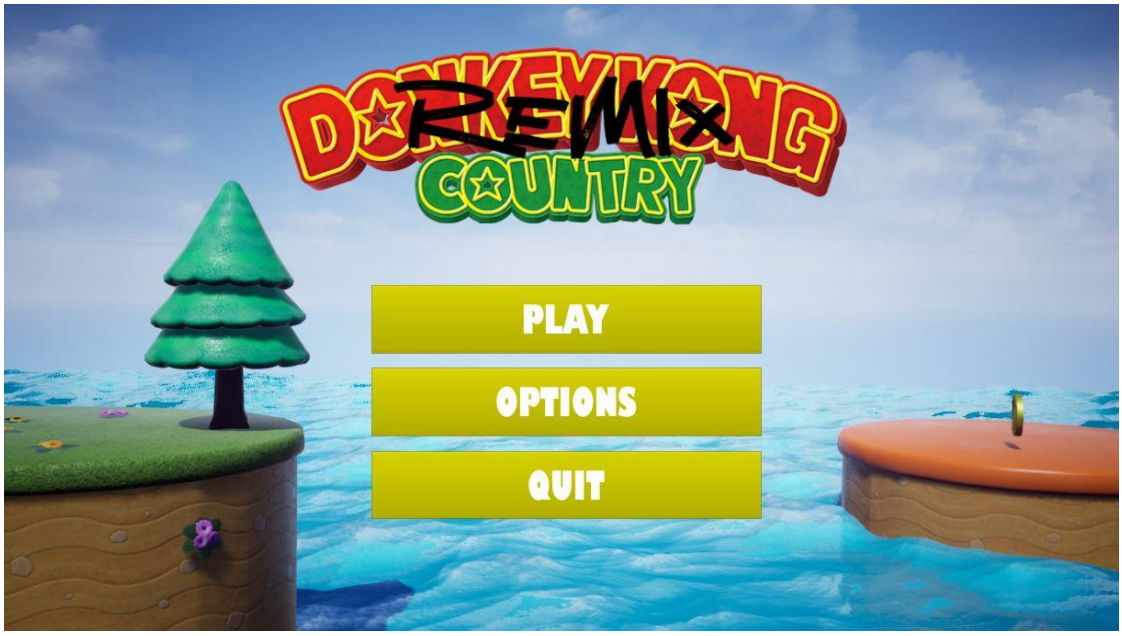

Figura 45 Captura del menú principal del juego final

El menú principal cuenta con un background animado que sigue la estética del escenario dónde parte el juego. Para el logo del juego, se ha utilizado el logo original de uno de los títulos de la saga Donkey Kong Country, modificando y añadiendo ciertos elementos para obtener el resultado que vemos en la Figura 45.

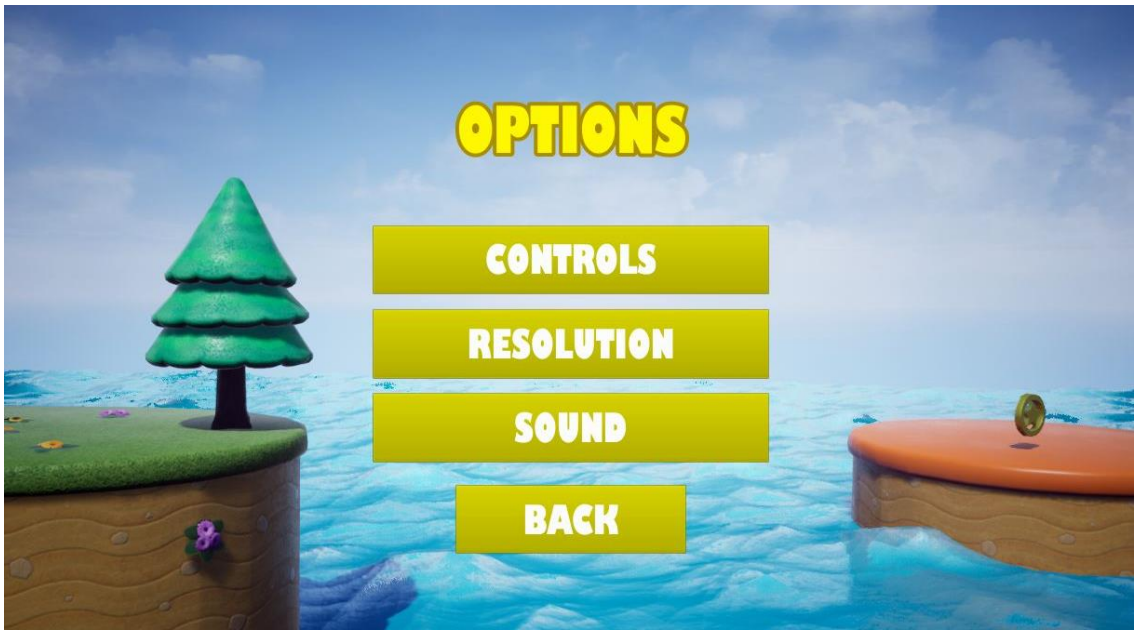

Figura 46 Captura del menú de opciones del juego final

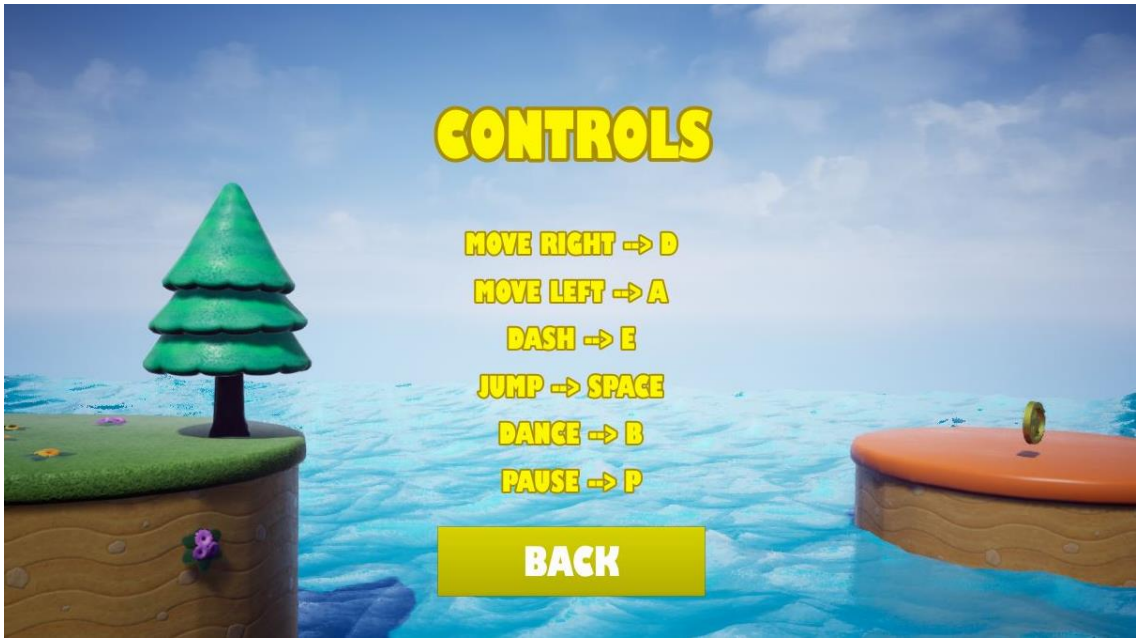

Figura 47 Captura del menú de controles del juego final

Figura 48 Captura del menú de resolución del juego final

Figura 49 Captura del menú de sonido del juego final

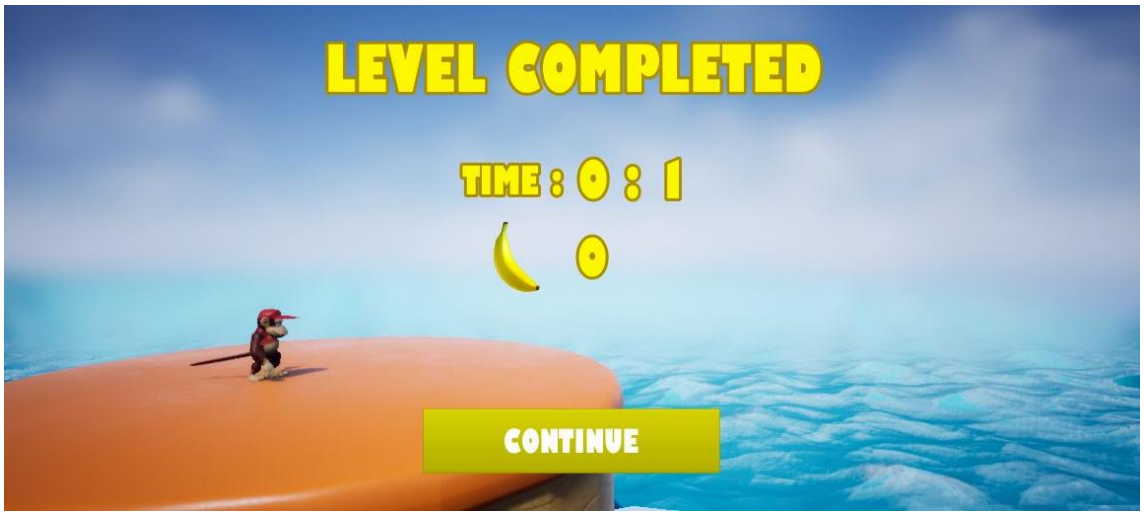

Figura 50 Captura del menú de nivel completado 1 del juego final

Figura 51 Captura del menú de nivel completado 2 del juego final

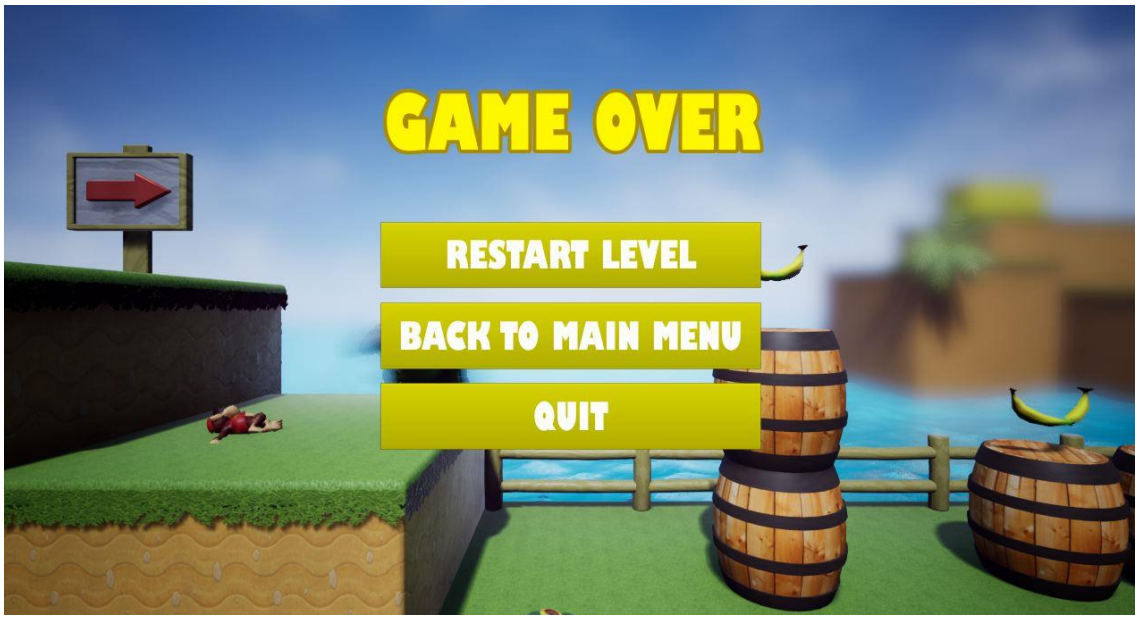

Figura 52 Captura del menú de final de partida del juego final

6.5 Gameplay

En este apartado se definirán todos los elementos que componen el gameplay, la parte que más importancia tiene a la hora de diseñar un videojuego.

El gameplay está formado por una serie de componentes que se encargan de definir los diferentes retos que el jugador deberá enfrentar y las acciones que el jugador puede realizar para poder superarlos.

6.5.1 Definición de retos

El reto principal de nuestro juego es el de llegar al final de la isla y salvar a nuestro amigo Donkey Kong. A partir de esto, aparecen diferentes subconjuntos de retos como el de llegar al final de cada nivel para poder ir avanzando al siguiente y así poder superar el objetivo principal. Estos retos se pueden ir simplificando hasta llegar a un nivel atómico, en el cual no se puedan simplificar más. Esta simplificación la veremos a continuación gracias a la jerarquía de retos, explicándolo de manera gráfica.

La mayoría de retos en nuestro juego son explícitos, ya que el jugador sabe desde un principio que debe llegar a la meta del nivel para poder acceder al siguiente o que los enemigos pueden matarte si no los esquivas o derrotas. Sin embargo, también existen algunos de implícitos, como pueden ser la obtención de las diferentes monedas del juego. Son retos opcionales, que no influyen en el avance hacía el objetivo principal, pero sí que ofrecen diferentes recompensas.

Los tipos de retos que se encuentran en el juego pueden ser de tipo **coordinación física** o de **exploración**. El primer tipo hace referencia a todos los retos que tienen que ver con alcanzar el reto principal del juego, ya que, para superar el nivel, se pondrán a prueba la velocidad y tiempo

de reacción del jugador, la precisión y la combinación de movimientos. El segundo tipo hace referencia a los retos que no van asociados directamente con alcanzar el reto principal y que tienen que ver con la recolección de las diferentes monedas y coleccionables distribuidos por los escenarios. En este caso, se pondrá a prueba la capacidad de exploración del jugador.

6.5.2 Jerarquía de retos

Como se ha mencionado anteriormente, se describirá el funcionamiento haciendo uso de una jerarquía de retos representada gráficamente en forma de diagrama.

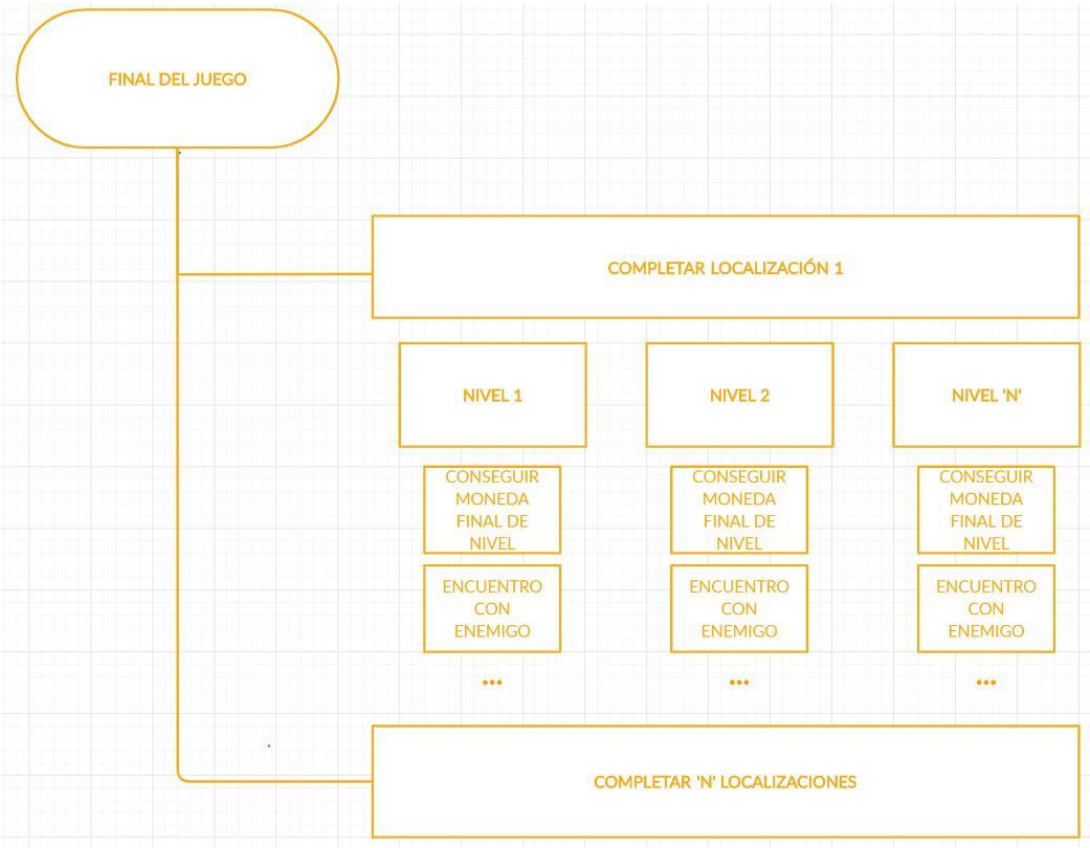

Figura 53 Jerarquía de retos

En la Figura 54 se puede ver los diferentes tipos de acciones que se pueden llevar a cabo durante el encuentro de un enemigo dentro de un nivel cualquiera del juego.

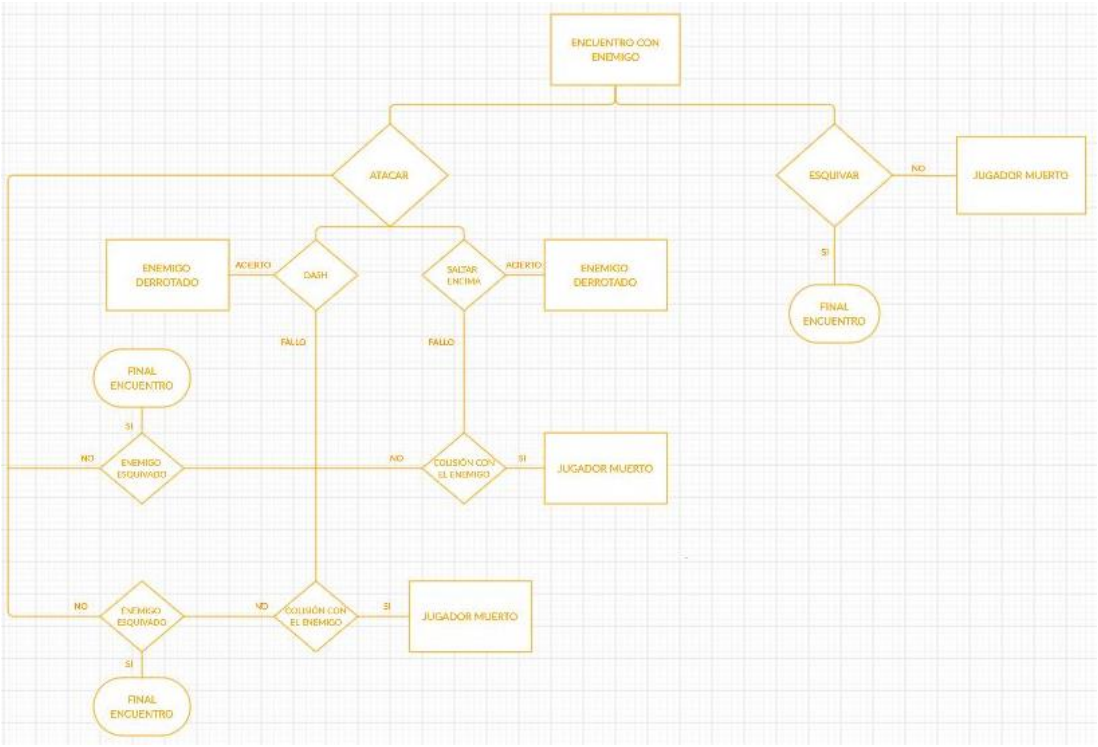

Figura 54 Jerarquía del reto 'Encuentro con enemigo'

6.4 Mecánicas

Las mecánicas del juego son las encargadas en definir las reglas del juego y cómo el jugador puede interactuar con el mundo virtual. Se podría decir que es el elemento que empieza a generar la jugabilidad, ya que ésta surge de las acciones disponibles del jugador en relación con los desafíos que se van presentando en el transcurso del juego.

Existen muchísimas mecánicas generadas tanto por las acciones del jugador como por la interacción de este con los elementos que forman el juego. Por ejemplo, el movimiento del personaje es una mecánica, y el recoger una moneda gracias al movimiento da lugar a una nueva mecánica, creada por la interacción del movimiento del jugador con el espacio que ocupa la moneda en el escenario.

A continuación, para la explicarlas mecánicas en profundidad, se dividirá el apartado en los subapartados siguientes:

6.4.1 Recursos

Los recursos de un videojuego son entendidos cómo el tipo de objetos o materiales que el juego puede gestionar, ya sea moviéndolos por el mundo del juego o cambiando los valores de sus atributos. Hay que aclarar que un recurso no se trata de un objeto específico, sino de un tipo de objeto.

Por ejemplo, las monedas en un juego pueden ser un recurso si el jugador puede recogerlas, venderlas o cambiarlas. Sin embargo, una sola moneda no es un recurso, sino las monedas en general. Es decir, las monedas son un recurso, pero las monedas que ha recogido el jugador durante el nivel son el objeto de un recurso.

6.4.2 Entidades

Una entidad puede ser tanto un objeto particular de un recurso como el estado de algún elemento en el mundo del juego. Un ejemplo de esto puede ser el de un personaje específico como entidad, cómo lo puede ser también el estado de un semáforo (verde, amarillo, rojo). Sin embargo, lo que diferencia las entidades de los recursos es que los recursos son un tipo concreto de objeto, mientras que las entidades son el objeto en sí.

Existen 3 tipos de entidades:

- **Entidades simples**: Son aquellas entidades que sólo tienen un tipo de estado. Este estado puede ser tanto un valor numérico como un estado definido por un valor simbólico.
- **Entidades compuestas**: Cuando es necesario más de un estado para definir un elemento del juego, estas entidades son los atributos de una entidad principal. Un ejemplo puede ser la entidad “Dragón” como entidad principal y las entidades “Vida”, “Velocidad” y “Daño” como atributos.
- **Entidad compuesta que contiene otra entidad compuesta como atributo**: En este caso, tomando el ejemplo anterior, si la entidad “Daño” tuviera los atributos “Fuego”, “Agua” y “Físico”, la entidad “Dragón” sería de este tipo.

A continuación, podemos ver en las Tablas 8 y 9, los recursos principales del juego:

Recurso	Entidad	Atributo	Estado
Enemigo	Orco	Vida	1 hit
		Velocidad	Media
		Daño	1 hit
		Dirección	Izquierda/Derecha
	Caracola	Vida	1 hit
		Daño	1 hit
	Dragón	Vida	1 hit
		Velocidad	Nula
		Daño	1 hit

Tabla 8 Enemigos

Recurso	Entidad	Atributos	Estado
Objeto	Banana	Bonificación	Si /No obtenido
	Moneda KONG	Bonificación	Si /No obtenido
	Moneda final de nivel	-	Si/No obtenido
	Gancho	Agarre	Si/No en uso
	Trampolín	Impulso	Si/No en uso
	Lanzador	Impulso	Si/No en uso
Interfaz principal	Contador de bananas	-	
	Contador de vidas	-	
	Indicador de letras conseguidas (KONG)	-	

Tabla 9 Objetos

6.4.3 Acciones del jugador

Las acciones del jugador también son conocidas como los “verbos” de las mecánicas del juego, ya que dicen en qué condiciones se pueden llevar a cabo las tareas para poder progresar. Además, también describen qué efectos tienen al cambiar el estado del juego.

Las **acciones principales** con las que contará el jugador son las siguientes:

- Desplazamiento lateral
- Saltar
- Dash (embestida)
- Dash golpeando
- Saltar + golpear y rebotar
- Colgarse
- Bailar

6.4.4 Mecánicas e interacción con enemigos

Los enemigos cuentan con unas mecánicas básicas que comparten entre ellos, como pueden ser el movimiento (puede ser estático o dinámico) o la mecánica de colisión, que viene dada por la interacción de colisión de nuestro personaje con ciertas partes del cuerpo del enemigo. A continuación, se explicarán las diferentes interacciones que se pueden ocasionar con los diferentes enemigos:

- **Interacción con orco**: El orco cuenta con un desplazamiento continúo definido por dos puntos ubicados en el mapa. Mientras se mueve desde un punto a otro, el jugador puede interactuar, ya sea para esquivarlo, matarlo o morir. El esquivarlo no significa otra cosa que intentar saltar por encima de él sin que le toque. En cambio, para matar al enemigo existen dos opciones, hacer un dash contra él o saltar encima. Si saltamos encima y lo matamos, se producirá la mecánica de rebote que nos impulsará nuevamente hacia arriba. En cambio, si lo matamos con el dash, el enemigo morirá sin aplicar ningún efecto sobre nosotros. Cuando el enemigo colisiona con nosotros sin haber podido alcanzar su cabeza, moriremos.

- **Interacción con caracola**: En el caso de la caracola, mantiene la misma posición sin desplazarse. Cuenta con dos estados, en reposo y en defensa. Cuando la caracola está en estado de reposo estará escondida en su caparazón, pudiendo saltar encima de ella para matarla. En cambio, sí se encuentra en estado defensivo, sacará de su caparazón unas púas que harán que, si se salta sobre ella, el personaje muera. Además de estas interacciones, contamos con las mismas que el orco, como la de golpearla con el dash, efectivo contra todos sus estados.

- **Interacción con dragón**: El dragón, cómo el caso de la caracola, se mantiene en la misma posición con la diferencia que éste se encuentra en el aire. Las interacciones son las mismas que con los dos enemigos anteriores con unas pequeñas diferencias:
    - No se le puede matar con un dash por el hecho que se encuentra en el aire
    - Al golpearlo en la cabeza, el impulso que se recibe es mayor que el que se recibe de los dos enemigos anteriores.

6.4.5 Mecánicas e interacción con objetos

A lo largo del gameplay nos encontraremos diferentes tipos de objetos con los que podremos interactuar de diferentes formas:

- **Banana**: Al recoger una banana nos sumará 1 el contador total de bananas.
- **Moneda KONG**: Existen 4 monedas KONG, una para cada letra de la palabra KONG. Existe una moneda de cada letra por escenario, que cuando se recogen, aparece en el HUD la letra procedente de la moneda.
- **Moneda de final de nivel**: Este objeto se encuentra al final de cada nivel, cuando se recoge, da por terminado el nivel y permite avanzar al siguiente.
- **Gancho**: Para interactuar con el gancho, el jugador debe saltar a la posición donde se encuentre un gancho. Si logra alcanzarlo, este se quedará sostenido. Para soltarse del gancho se debe volver a saltar, lo cual generará un nuevo salto desde la posición del gancho, liberándose de él.
- **Trampolín**: Este objeto lo podemos encontrar por el terreno del escenario. Si saltamos sobre él, rebotaremos hacía arriba llegando a una altura mucho mayor que la que proporciona el salto del personaje.
- **Lanzador**: El lanzador interacciona de la misma forma que el trampolín. La diferencia está en que lo podemos encontrar en el aire en vez de sobre el terreno.

6.5 Flowchart

El flowchart es una representación del flujo de trabajo en forma de diagrama. En este apartado se representará el flujo de trabajo de nuestro juego a partir de este método, enseñando las diferentes pantallas que contiene, y el flujo que sigue para acceder a cada una de ellas.

Figura 55 Flowchart de Donkey Kong Country Remix

6.6 Level design

En cuanto al diseño de niveles, hemos introducido las diferentes localizaciones tanto en el punto 6.1.2 cómo en el punto 6.2.2 hablando desde los puntos de vista narrativo y estético. En este apartado, se trata el diseño de estos escenarios y la importancia de ello debido a que condiciona el tipo de retos que nos podemos encontrar según el tipo de escenario que nos encontremos. Un ejemplo de esto, puede ser que, dependiendo del diseño del nivel, incremente o decremente en nivel de dificultad. No es lo mismo encontrarse con un nivel repleto de obstáculos y enemigos en un espacio reducido que otro mucho más grande. Por ese motivo se debe adecuar el diseño de niveles a la progresión del jugador.

Además de esto, también se deben adecuar tanto los enemigos como los elementos que interactúan en el nivel, manteniendo una concordancia entre ellos y el escenario en cuestión.

6.6.1 Economía interna

La economía de un juego es el sistema en el que los recursos y entidades son producidas, consumidas e transformadas en cantidades cuantificables. En el caso de nuestro título, la economía interna es bastante clara debido al género el cual pertenece el juego, haciendo de ello una ventaja a la hora de poder clasificarlo y posteriormente balancear el juego y su dificultad relacionada con este factor.

A continuación, se explicarán los diferentes elementos que controlan la economía en nuestro juego:

- **Vida**: La economía de la vida del personaje se basa en el número de vidas que tiene en el nivel que se encuentra. Cuando el personaje es “golpeado” por un enemigo o cae fuera del terreno del nivel, muere directamente, perdiendo una vida del total que tiene, y apareciendo en el último checkpoint del nivel. Si el jugador se queda sin vidas durante el nivel, perderá todo lo obtenido del nivel y deberá empezarlo nuevamente.
- **Daño**: En el caso del daño, la economía es muy clara, un hit equivale a morir. Tanto si se golpea un enemigo como si se es golpeado por éste, indicará la muerte del jugador. En un futuro se puede llegar a modificar esto con la inclusión de nuevos enemigos.
- **Monedas**: Existen dos tipos de monedas adquiribles durante el transcurso del nivel, las bananas y las monedas KONG. Las bananas se acumulan todas juntas en un contador global mientras que las monedas KONG son únicas entre ellas y se contabilizan de manera individual, necesitando conseguir las 4 monedas escondidas por el nivel para obtener la palabra completa. La adquisición de todas las bananas y las monedas KONG dan lugar a obtener el 100% del nivel.

En cuanto al balanceo del juego, es un punto que se irá trabajando y haciendo pruebas durante la implementación del juego ya que influyen muchos factores que a vista previa no pueden identificados como tales.

7. Desarrollo tecnológico, implementación y pruebas

En este punto se explicará todo el proceso de implementación y realización del prototipo final, junto a las diferentes pruebas de funcionamiento que se fueran realizando a medida de su desarrollo, haciendo una mayor o menor profundización en los diferentes aspectos de la implementación según su grado de importancia y relevancia en el producto final.

7.1 Creación del proyecto

El primer paso se da a cabo con la creación del proyecto y su correspondiente puesta en marcha.

Desde la aplicación Epic Games Launcher podemos acceder al motor de Unreal Engine 4. Este proyecto se realizará con la versión del motor 4.22.3 (ver Figura 6.1).

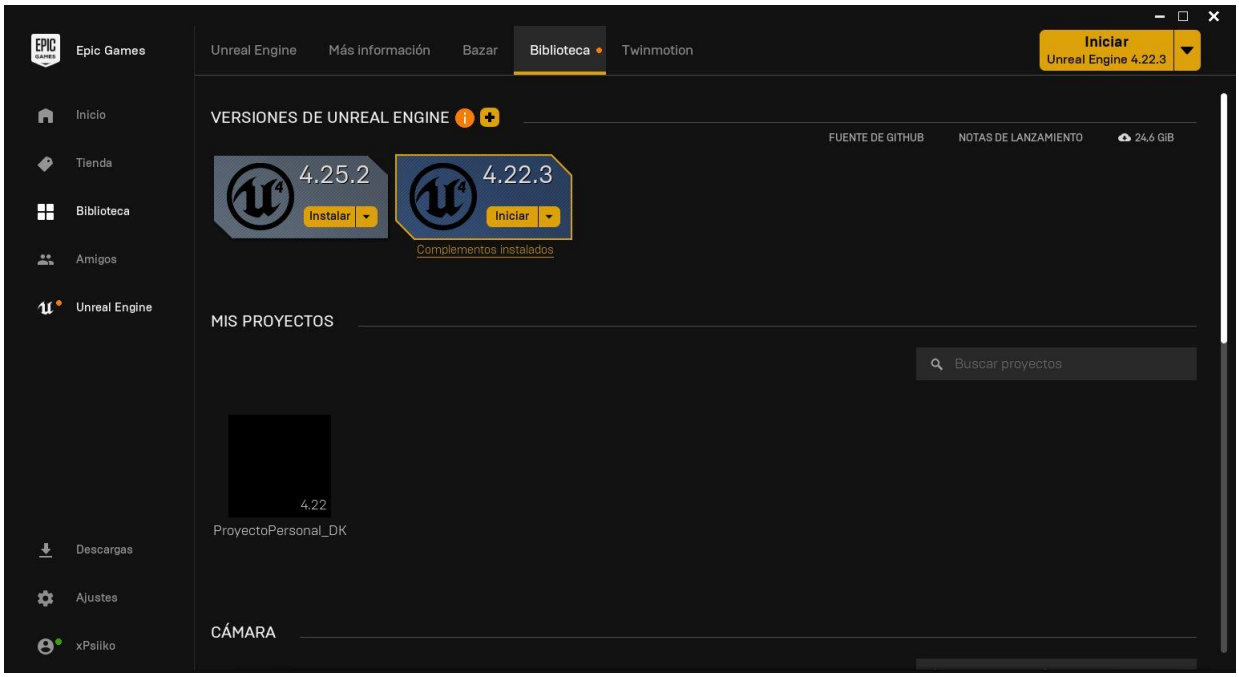

Figura 56 Captura de lanzamiento del juego desde Epic Games Launcher

Al iniciar el motor, es necesario configurar el proyecto inicial. En este caso, como se muestra en la Figura 57**,** como el juego es de desplazamiento lateral, se utilizará la plantilla de Blueprints Side Scroller.

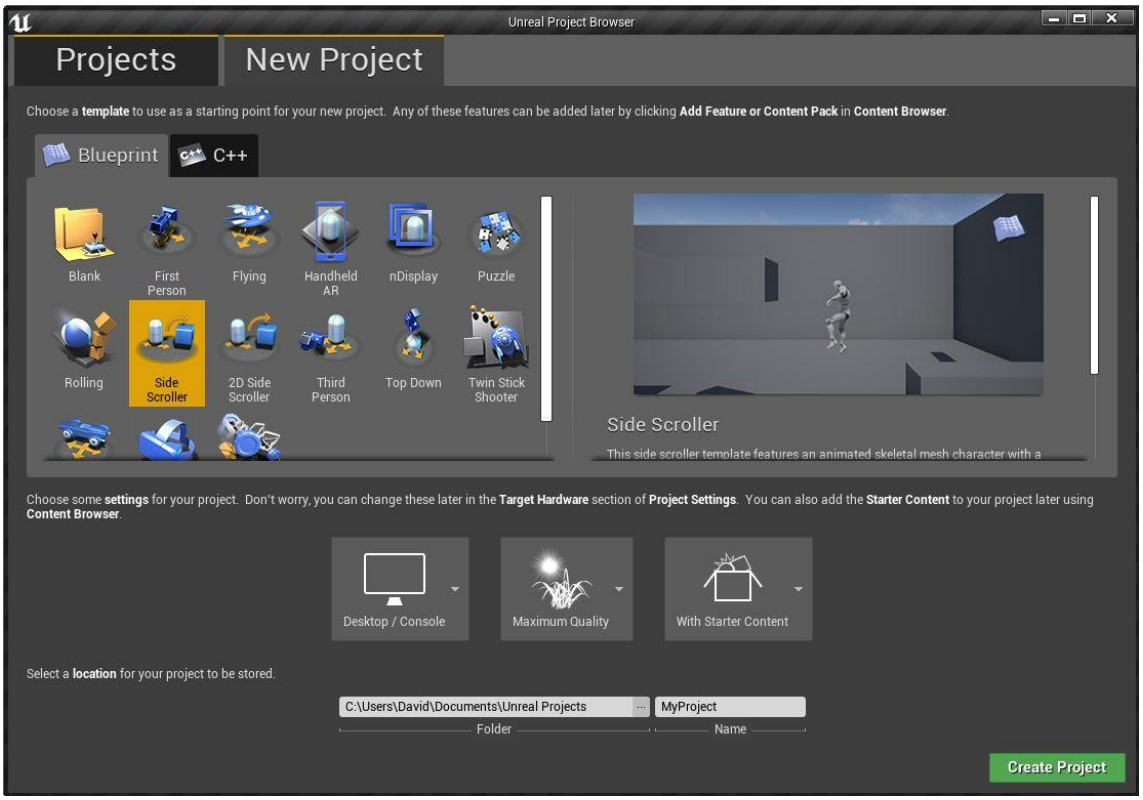

Figura 57 Captura de opciones de nuevo proyecto del motor Unreal Engine 4

Una vez creado, podemos ver que el proyecto cuenta con una configuración base donde el blueprint de nuestro personaje cuenta con ciertos componentes que nos facilitarán posteriormente la implementación, tanto de la cámara como el movimiento del personaje.

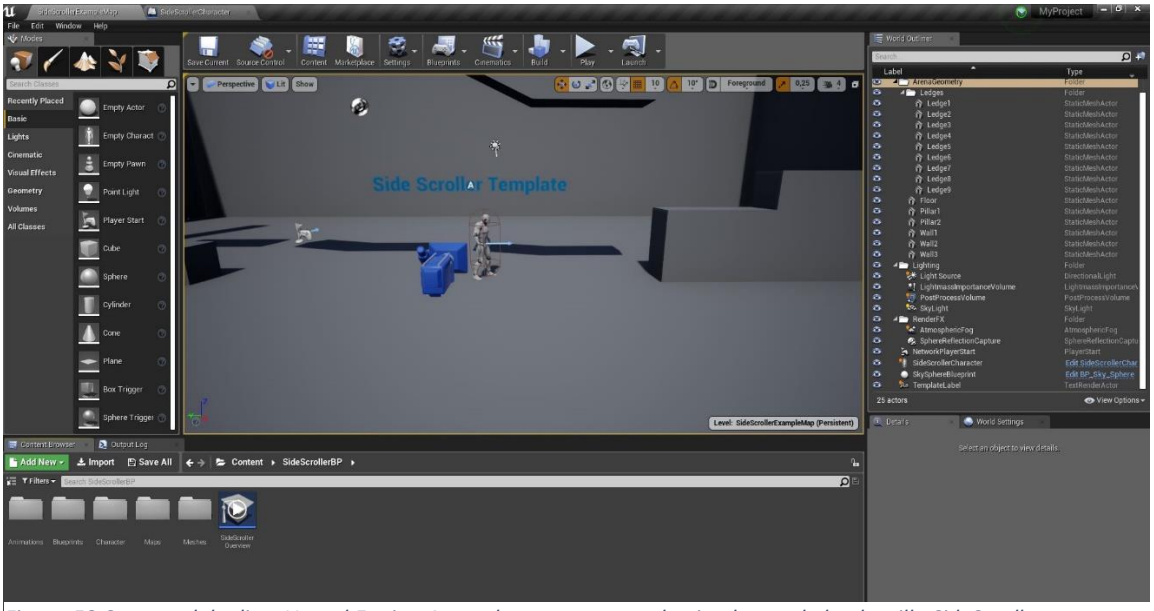

Figura 58 Captura del editor Unreal Engine 4 con el nuevo proyecto haciendo uso de la plantilla SideScroller

Una vez creada la base del proyecto, no existe un orden establecido por el cual se deba empezar a implementar, así que se explicará la implementación en función al orden que ha seguido el proyecto.

7.2 Importación de Personajes y Animaciones

En este apartado se explicará el procedimiento que se ha llevado a cabo para poder obtener los diferentes modelos de los personajes que aparecen en el juego junto a sus animaciones, tanto el personaje principal como los enemigos.

7.2.1 Obtención de los modelos de los personajes

Los modelos utilizados en el juego son provenientes de diferentes lugares, ya que no es fácil encontrar assets gratuitos o con un precio accesible que cuenten con una estética que concuerde con la similitud del juego. A pesar de ello, se ha conseguido encontrar el modelo del personaje principal Diddy Kong en 3D y con un formato óptimo para poder generarle posteriormente un esqueleto y así animarlo (ver Figura 59)**.**

Por otra parte, encontrar los enemigos originales del juego *Donkey Kong Country 2: Diddy's Kong Quest* fue imposible. Por ese motivo se intentó encontrar modelos que se asemejaran a los originales o que por lo menos pudieran cumplir la función en cuanto a concordancia con el comportamiento, función a desarrollar y estética del nivel del juego.

Cabe destacar que, además de lo mencionado anteriormente, los diferentes modelos deben estar en formato obj o fbx, formatos compatibles para poder trabajar con la aplicación Mixamo.

Figura 59 Captura del Editor de Unreal Engine 4 mostrando el mesh de Diddy Kong

Figura 60 Captura del Editor de Unreal Engine 4 mostrando el mesh de un cocodrilo enemigo

Figura 61 Captura del Editor de Unreal Engine 4 mostrando el mesh de una araña enemiga

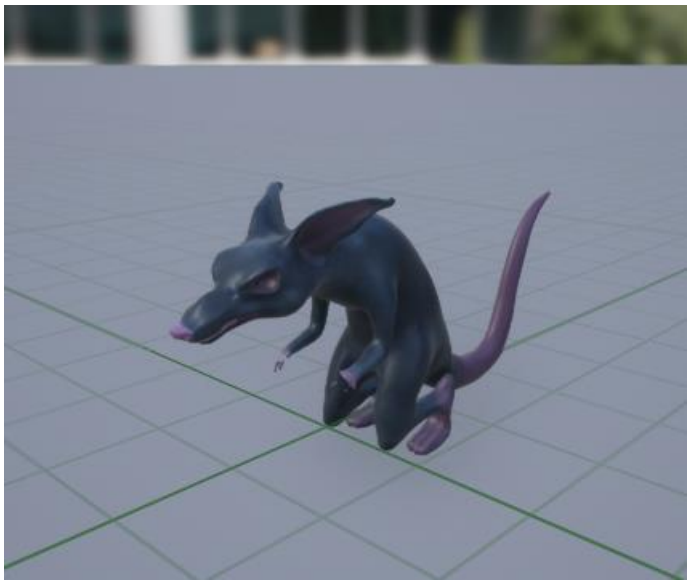

Figura 62 Captura del Editor de Unreal Engine 4 mostrando el mesh de la Rata enemiga

7.2.2 Rigging y animación con Mixamo

En este punto, ya hemos comprobado que los modelos se importan y cargan correctamente en el motor de videojuegos. Para hacer el rigging y las animaciones se ha utilizado Mixamo, ya que permite generar el esqueleto de manera automática y da acceso a una gran variedad de animaciones a partir del esqueleto creado. Hay que tener en cuenta que, al ser un programa pensado para modelos humanoides, puede haber ciertos problemas al intentar generar el esqueleto de un animal u otro ser no bípedo. Por ese motivo, el primer modelo a probar fue el personaje principal.

Para cargar el personaje en la aplicación sólo hay que pulsar el botón **Upload Character** y seleccionar el personaje en cuestión que se quiere riggear o animar. Una vez el programa empiece a procesar el modelo, dependiendo la topología del personaje, quizás se deba ajustar una serie de marcas que indican donde se ubican ciertas partes del esqueleto (ver Figura 63):

- Barbilla
- Muñecas
- Codos
- Rodillas
- Ingle

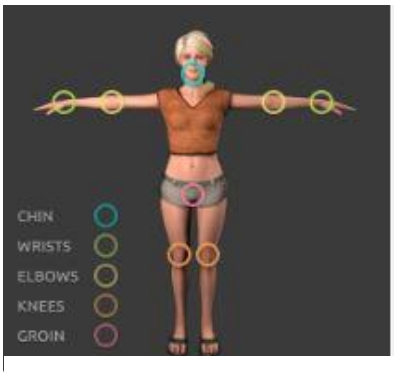

Figura 63 Captura de Imagen indicativa de Mixamo

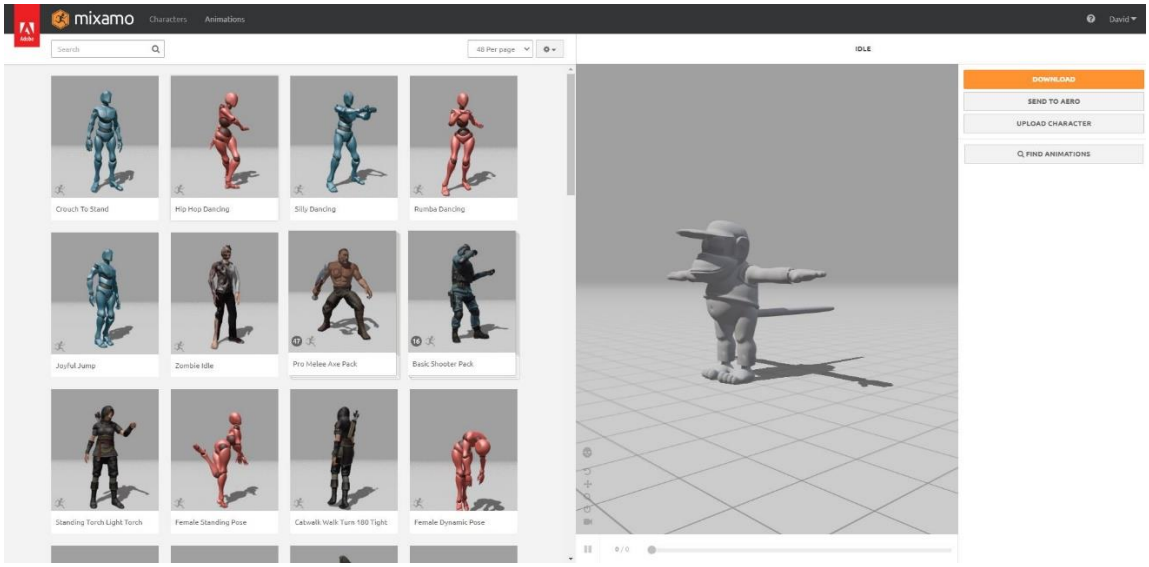

Figura 64 Captura de la aplicación Mixamo con el modelo de Diddy Kong y el esqueleto generado

La generación del esqueleto en Diddy Kong fue completamente automática, con un resultado excelente. Acto seguido, se han buscado entre las opciones que ofrece Mixamo, las diferentes animaciones que el modelo necesitaría más adelante, durante la implementación de las mecánicas del personaje (ver Figura 64). Es necesario hacer un pequeño inciso en esta parte: es importante aclarar que las animaciones proporcionadas por Mixamo no cuentan con que el modelo tenga una cola, por ese motivo en las animaciones de Diddy Kong aparecerá la cola rígida.

A continuación, se repitió el proceso con el cocodrilo, para el cual se obtuvo un resultado óptimo, pero no tan bueno como en el de Diddy Kong debido a que la cola del modelo es

bastante grande y sufre una pequeña deformación en la parte inferior en ciertas animaciones. Aun así, se ha conseguido un resultado bastante bueno.

En el caso de la araña no ha sido necesario utilizar Mixamo, ya que contaba con animaciones y un esqueleto previo que acabamos de ajustar con el software 3ds Max.

Sin embargo, en el caso de la rata sí que es necesario generarle el esqueleto. El resultado con ella no es el deseado debido a su tipología (ver Figura 65).

Figura 65 Captura de la aplicación Mixamo con el modelo de Diddy Kong y el esqueleto generado

A pesar de ello, seguimos adelante con la implementación del proyecto teniendo en cuenta que posteriormente se necesitará algún modelo más.

7.2.3 Cambio en los modelos de los enemigos

Hay que aclarar que el cambio en los modelos de los personajes enemigos se ha hecho durante el transcurso del proyecto y no desde el principio. En este punto se mostrarán los modelos finales de los enemigos utilizados en el prototipo final del juego.

A mediados del desarrollo del proyecto y contando con los modelos de los enemigos anteriores ya implementados dentro del juego y con el asset listo, se ha decidido cambiarlos por otros totalmente diferentes. La razón principal por la cual se ha llevado a cabo este cambio se debe a la dificultad de encontrar modelos independientes que compartan una apariencia y estética que se asemeje a la que el proyecto necesita. Por este motivo, y aprovechando la tienda de assets con la que cuenta el Epic Games Launcher, se decidió hacer la compra de un pack de personajes hechos por el artista Dungeon Mason que se adecuan bastante bien a la estética del proyecto.

Los modelos del pack de modelos utilizados en el proyecto son los que podemos ver en las Figuras 66, 67 y 68:

Figura 66 Captura del Editor de Unreal Engine 4 mostrando el mesh del Dragón

Figura 67 Captura del Editor de Unreal Engine 4 mostrando el mesh del Orco

Figura 68 Captura del Editor de Unreal Engine 4 mostrando el mesh del Caparazón

Todos los modelos del pack vienen con un esqueleto ya generado junto a un set de animaciones bastante completo. Gracias a ello y la facilidad que ofrece la tienda de Epic Games, se pueden importar el pack completo de forma directa al proyecto, sin complicaciones añadidas.

7.3 Cámara

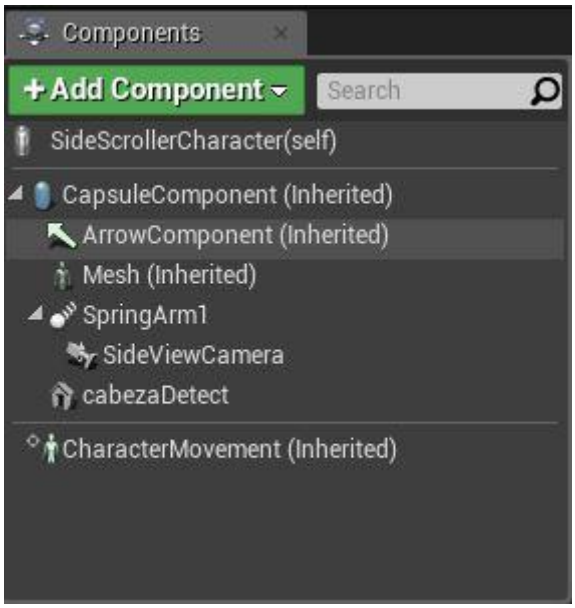

Figura 69 Componentes del blueprint del personaje principal Diddy Kong

Para el movimiento de la cámara aprovecharemos la configuración previa con la que cuenta la plantilla **SideScroller** vinculada al blueprint de nuestro personaje principal. En este proyecto sólo

trabajaremos con una única cámara ya que no necesitamos generar diferentes vistas en el transcurso del gameplay.

En la Figura 69 podemos ver los componentes **SpringArm** y **SideViewCamera**, que son los que se encargarán de todo el control de cámara.

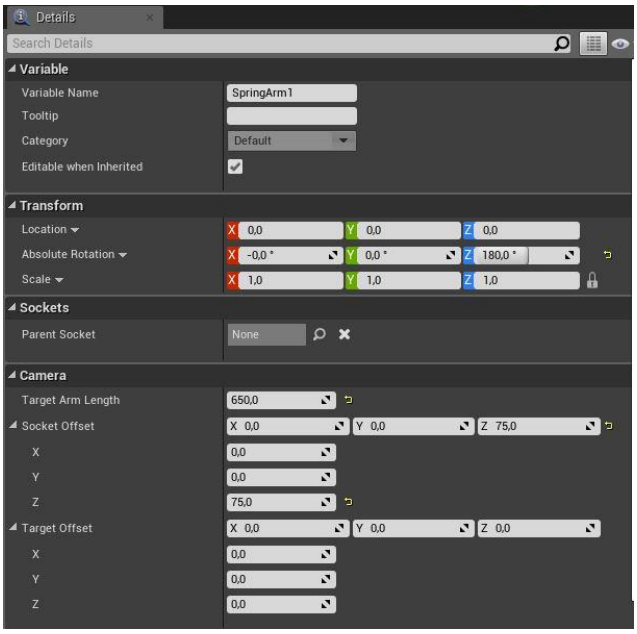

Figura 70 Detalles del componente SpringArm

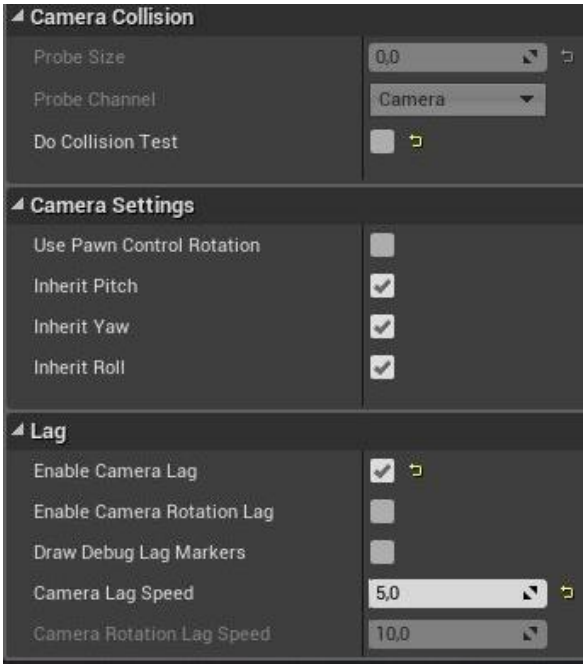

Figura 71 Ajustes del componente SpringArm

La función de este componente se asemeja al que haría un brazo estabilizador para una cámara en la vida real, pero con muchas más funcionalidades. Gracias a esto, podemos controlar automáticamente muchos aspectos y situaciones en las que podríamos tener problemas, por ejemplo, la obstrucción de la cámara por culpa de la geometría del nivel, penetración de objetos, etc. En nuestro caso, vamos a utilizarla para poder ajustar la longitud del brazo (**Target Arm Lenght**) y el retraso de la cámara en función del movimiento del personaje (**Camera Lag**). Ajustando la longitud del brazo conseguimos que la cámara se ubique a una mayor o menor distancia del personaje, de manera que podamos ajustar a nuestra conveniencia que tan lejos o cerca queremos ver el nivel. El valor de éste se irá ajustando a lo largo del proyecto ya que es necesario hacer pruebas jugables para poder escoger un valor que se adecúe al gameplay deseado.

Por otra parte, la sección de **Lag** nos permite generar un efecto muy característico con el que cuenta el juego original. Al activar el **Enable Camera Lag** podemos ajustar una serie de parámetros para crear un retraso en el seguimiento que hace la cámara al personaje. Ajustando el parámetro **Camera Lag Speed** determinamos si queremos más o menos retraso en el seguimiento de la cámara en función al desplazamiento del personaje. Este parámetro se deberá ir ajustando y probando a lo largo del proyecto, igual que el anterior, para encontrar el valor ideal.

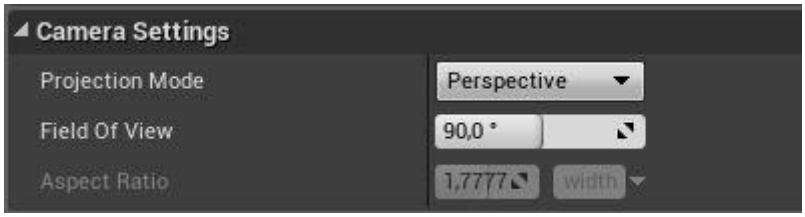

Figura 72 Ajustes del componente SideViewCamera

Como hijo del componente **SpringArm** tenemos su respectiva cámara.

En los ajustes de la cámara nos centramos en el modo de proyección y el ángulo del campo de visión. Trabajaremos con una proyección en perspectiva y un ángulo de campo de 90º, consiguiendo el aspecto 2,5D que se desea.

7.4 Movimiento y acciones del personaje

El movimiento del personaje consiste en el desplazamiento en dos dimensiones. Al ser un juego de desplazamiento lateral el jugador tendrá la posibilidad de moverse en horizontal y verticalmente a lo largo del nivel.

7.4.1 Input

En el apartado de **Project Settings** del proyecto, encontramos la sección de **Input**, dónde se han añadido los elementos identificativos para los controles del juego. En otras palabras, una lista de definiciones para los controles en las que podremos acceder desde el **Graph Editor** de blueprints por el nombre que definamos aquí (ver Figura 73).

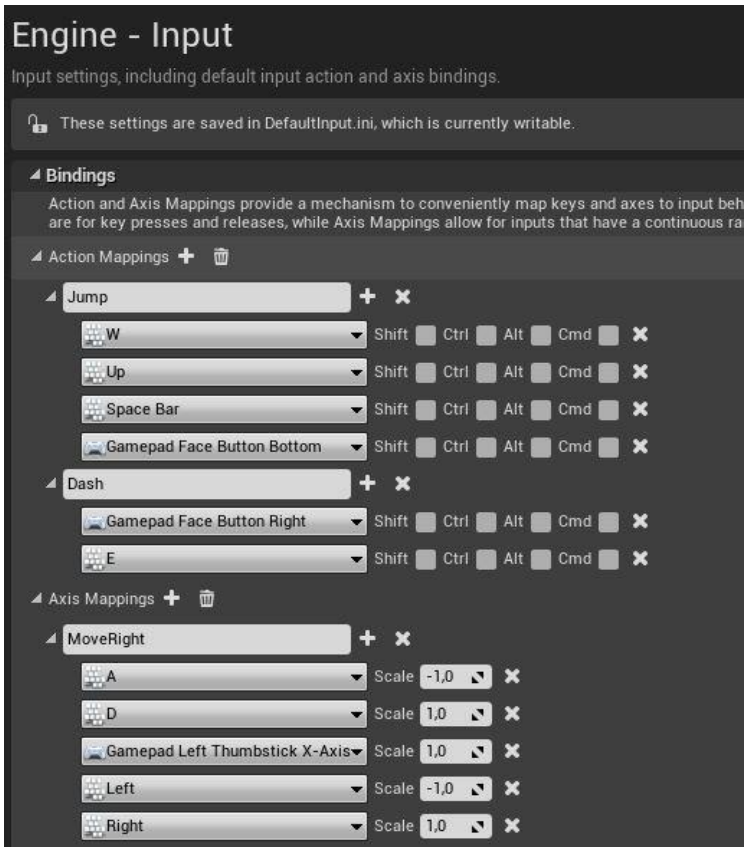

Figura 73 Captura de la configuración de Inputs

Si nos fijamos en el apartado de input, contamos con valores de **Action Mappings** y valores de **Axis Mappings**. La diferencia entre estos dos está en que los **Axis Mappings,** al ser pensados para movimiento, guardan una variable de tipo **float** que aumenta o decrementa dependiendo del tiempo que el jugador pulse la tecla, facilitando la implementación progresiva del movimiento.

Para los valores del **Axis Mapping** no es conveniente definir uno para cada dirección de movimiento, sino que podemos utilizar la misma definición asignando a cada tecla un valor

específico. Por ejemplo, en nuestro juego, al pulsar A el personaje se mueve hacia la izquierda en el eje horizontal, por lo que tiene asignado un valor de escala -1.0. Por lo contrario, si pulsa D, el valor que tomará la variable será de la escala 1.0 y se desplazará hacia la derecha en el eje horizontal.

7.4.2 Blueprint de movimiento y acciones del personaje

A continuación, explicaremos el blueprint de control de movimiento del personaje con todas sus acciones y cambios de estado (andando, saltando, agarrado, etc.).

El blueprint del personaje se basa en el que nos da la plantilla utilizada, por ese motivo heredamos el movimiento básico de la clase **CharacterMovement**, que consiste en un desplazamiento básico y un salto. A partir de esto, modificaremos este blueprint para adaptarlo a nuestro juego.

7.4.2.1 Desplazamiento Lateral

El desplazamiento lateral es el movimiento básico del personaje. Como podemos ver, el input que recibimos del jugador antes definido será el que tomará la función de **Add Movement Input**, encargada de aplicar el movimiento recibido al personaje, solamente si el jugador no está agarrado (**Is Hanging**). La acción de agarrarse la explicaremos más adelante es su apartado respectivo.

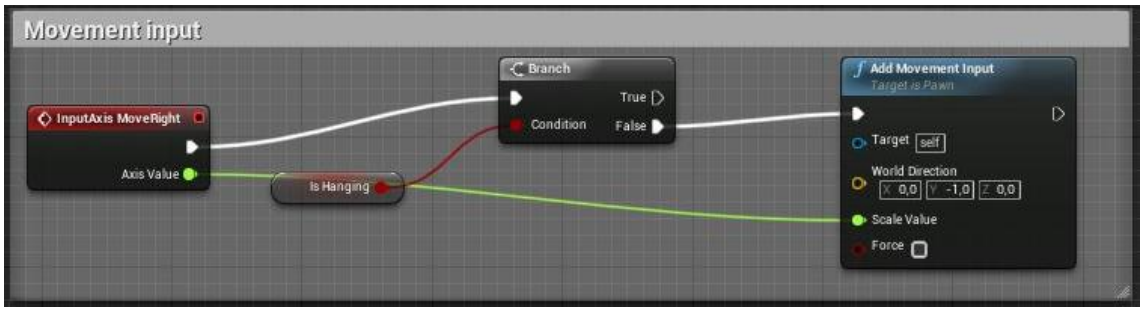

Figura 74 Captura de la lógica del Movimiento básico

7.4.2.2 Salto

La lógica del salto aplicada a nuestro personaje también necesita saber si el estado de nuestro personaje se encuentra agarrado, de manera que, si el personaje se encuentra en dicho estado, se llama al evento **ExitLedge** (se explicará más adelante su funcionamiento) antes de poder saltar. El resultado es el mismo tanto si el personaje se encuentra agarrado como si no, saltará.

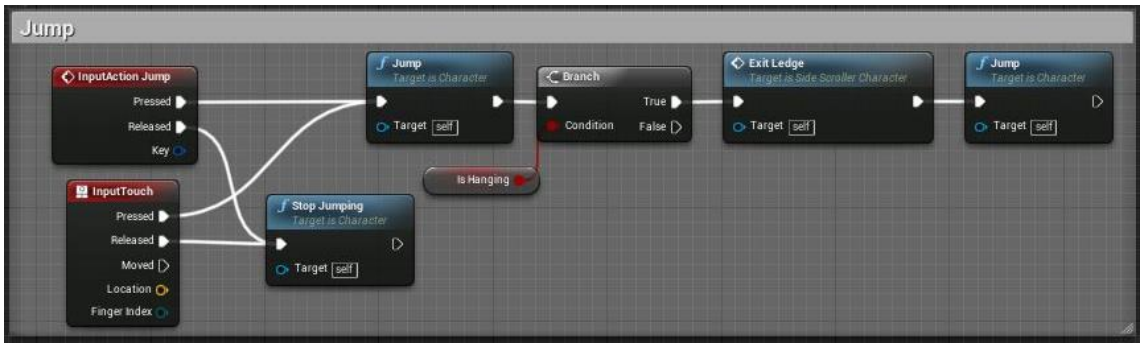

Figura 75 Captura de la lógica del salto

7.4.2.3 Dash

Para explicar el blueprint del dash lo dividiremos en varias partes, ya que su lógica es más compleja que las dos anteriores.

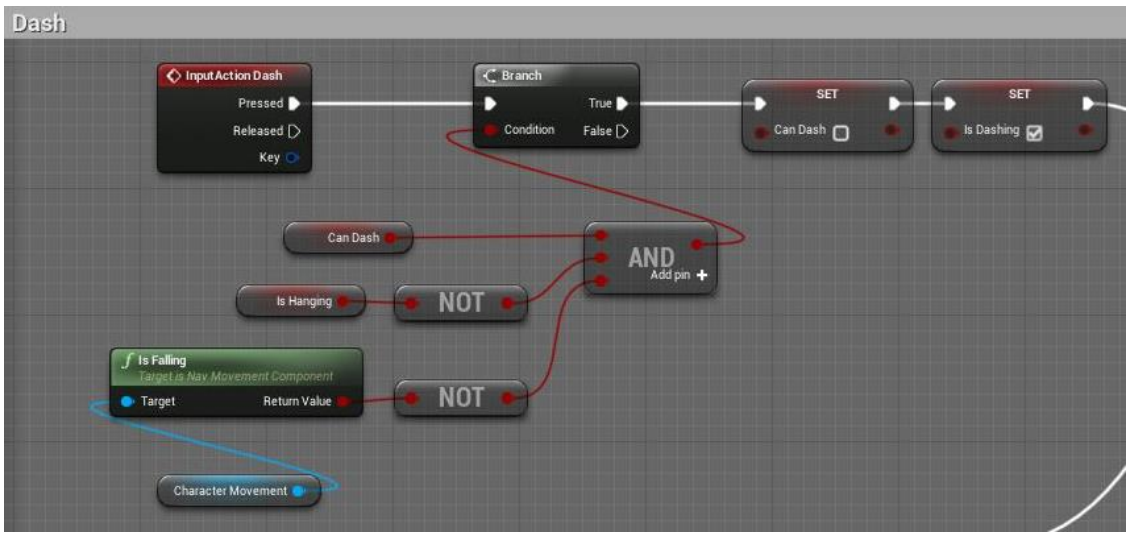

Figura 76 Captura de la lógica del dash parte 1

Para la implementación del dash (embestida) lo primero que hacemos es comprobar si el personaje se encuentra en un estado en el que puede embestir, si no está agarrado y si no está cayendo. Estos estados los obtenemos a partir de unas variables de tipo booleano que se han creado en la clase personaje y a las que se les modifica su valor dependiendo de los inputs y acciones que se reciben por parte del jugador.

Acto seguido, si todas las condiciones son ciertas, cambiamos el valor de **CanDash** a falso para evitar un loop infinito al pulsar repetidamente el input, y modificamos el valor a verdadero de la variable **isDashing** para saber en qué estado se encuentra el personaje.

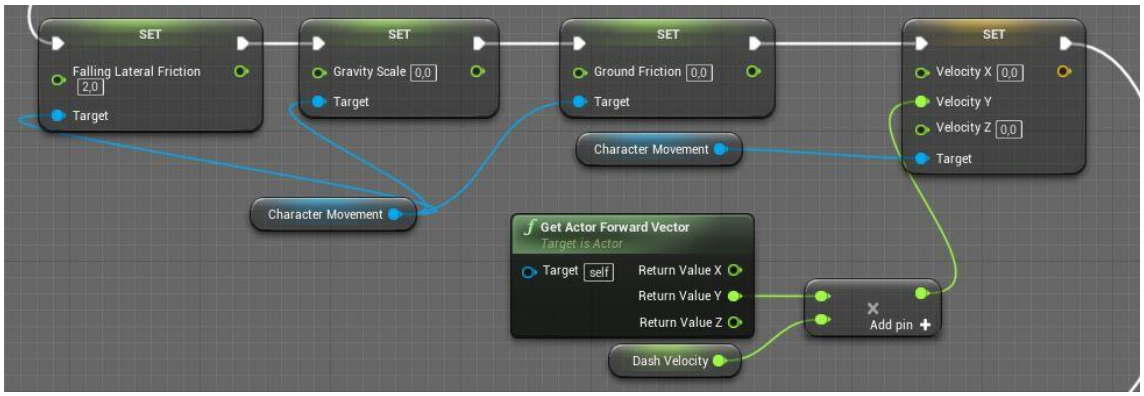

Figura 77 Captura de la lógica del dash parte 2

A continuación, en la Figura 77, podemos ver cómo se modifican ciertos parámetros del **Character Movement** y además se multiplica la velocidad de desplazamiento del personaje por una velocidad concreta providente de la variable **Dash Velocity**.

Los parámetros los cuales establecemos un nuevo valor son:

- **Falling Lateral Friction**: Este parámetro se encarga de establecer la fricción que el aire aplica sobre el personaje cuando está cayendo. Al principio de la implementación se pasó por alto este detalle, pero en la fase de prueba se detectó cómo al embestir y acabar en el aire debido a un cambio de altura, el personaje no realizaba correctamente el desplazamiento del dash debido a que esta fricción se lo impedía. Por ese motivo, durante la realización de la acción de dash se redujo su valor a de 5 a 2.
- **Gravity Scale**: En el caso de la escala de gravedad, se cambió su valor a 0 para evitar alterar el desplazamiento durante la acción si hay un cambio de altura en el nivel.
- **Ground Friction**: Se redujo a 0 para que el dash sea un deslizamiento.

  En el ajuste de la velocidad del personaje, sólo aplicamos el incremento en el eje horizontal por un valor fijo.

Tanto los parámetros cómo los valores que vemos representados se han ido ajustando y probando durante todo el desarrollo hasta conseguir el resultado mostrado.

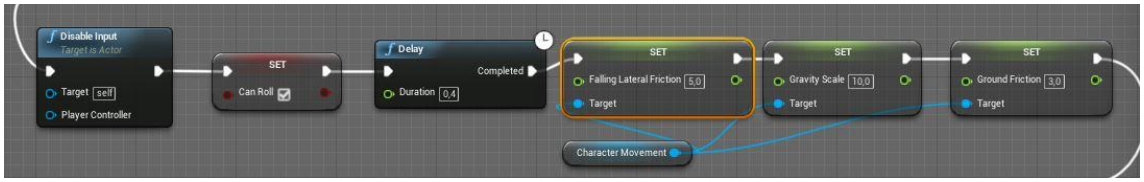

Figura 78 Captura de la lógica del dash parte 3

A continuación, deshabilitamos el control del personaje al jugador para que no pueda intervenir en la ejecución del dash y cambiamos de valor variable **CanRoll** a cierto (la variable **CanRoll** se utiliza en el blueprint de animaciones del personaje). Todo seguido, establecemos un delay para que se realice el desplazamiento al embestir y reestablecemos los valores anteriores de los parámetros del **Character Movement** modificados.

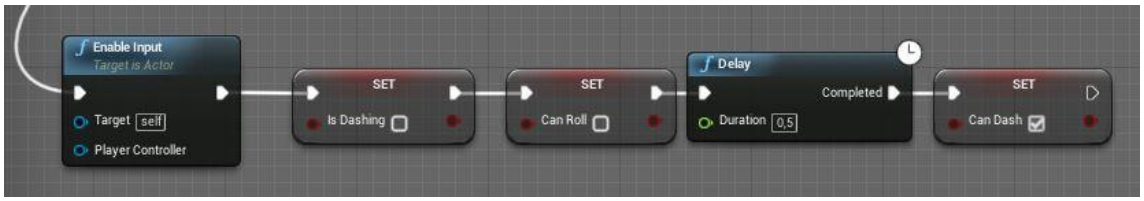

Figura 79 Captura de la lógica del dash parte 4

Por último, habilitamos el input del control del personaje al jugador y modificamos el valor de las variables **IsDashing** y **CanRoll** a falso nuevamente. En el caso de la variable **CanDash**, aplicamos un delay de medio segundo antes de establecer el valor a verdadero para así evitar la posibilidad que el jugador pueda hacer un uso abusivo consiguiendo ser invulnerable al daño enemigo de forma ilimitada.

7.4.2.4 Colgarse

La acción de colgarse viene dada con la interacción del personaje con el objeto gancho.
Cuando el gancho detecta colisión con el personaje (la implementación de la interacción viene explicada en el apartado del objeto), esté llama al evento **GrabLedge** creado en el blueprint del personaje y que se explicará a continuación (ver Figuras 80 y 81).

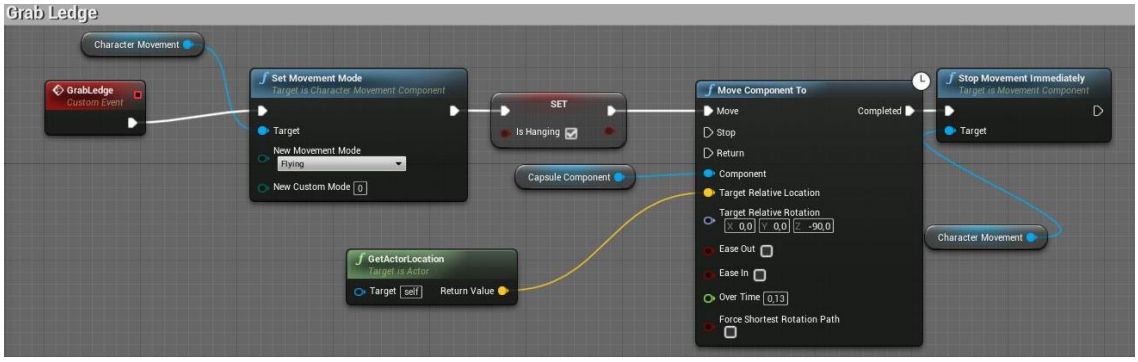

Figura 80 Captura de la lógica de colgarse parte 1

En el evento **GrabLedge** llamamos al nodo **Set Movement Mode** dónde cambiamos el estado del movimiento del personaje al modo **Flying**, y cambiamos el estado de la variable de control de estado **Is Hanging** a cierto. A continuación, llamamos al nodo **Move Component To** conectando al pin **Component** la cápsula del personaje y en el pin **Target Relative Location** la ubicación de nuesto actor (el personaje) dando un tiempo de interpolación de 0,13 segundos. Por último, paramos el movimiento del personaje llamando al nodo **Stop Movement Immediately** para que el personaje quede quieto sobre el gancho.

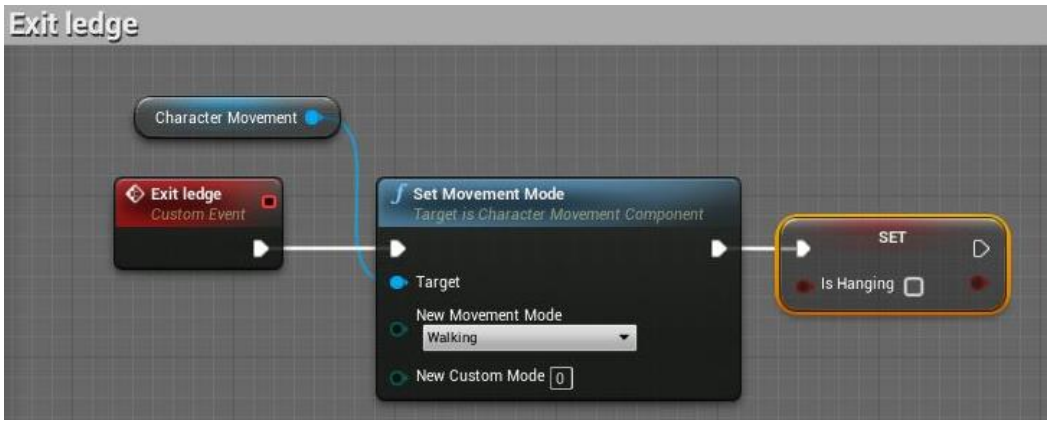

Figura 81 Captura de la lógica del colgarse parte 2

Para descolgarse del gancho, sólo hay que volver a presionar el input de salto, y este llamará al evento **ExitLedge** si el jugador está en estado colgado. En el evento **ExitEdge** llamamos al nodo Set **Movement Mode** cambiando el estado de movimiento del personaje a **Walking** y modificando el valor de la variable **Is Hanging** a falso, para que el personaje pueda abandonar el estado de agarre del gancho saltando.

7.4.2.5 Bailar

Esta acción se ha utilizado en un principio como una forma de probar las diferentes animaciones y finalmente se ha dejado su input funcional como un gesto divertido en el juego.

Se trata de una animación de baile que se puede realizar manteniendo pulsado la tecla B. Para dejar de bailar simplemente se ha de deja pulsar.

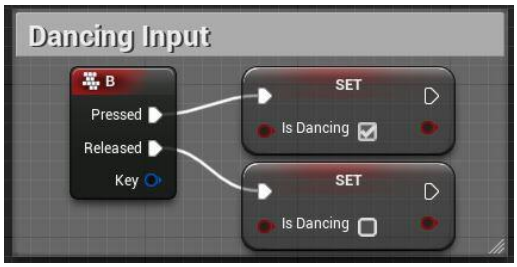

Figura 82 Captura de la lógica de bailar

7.5 Desarrollo de nivel

En este apartado se explicará la implementación de todos los componentes del juego que no influyen una lógica de implementación que interfiera en el flujo del juego. En otras palabras, se presentarán los assets utilizados, su uso para la creación del nivel y diferentes configuraciones tanto en iluminación como en efectos de renderizado.

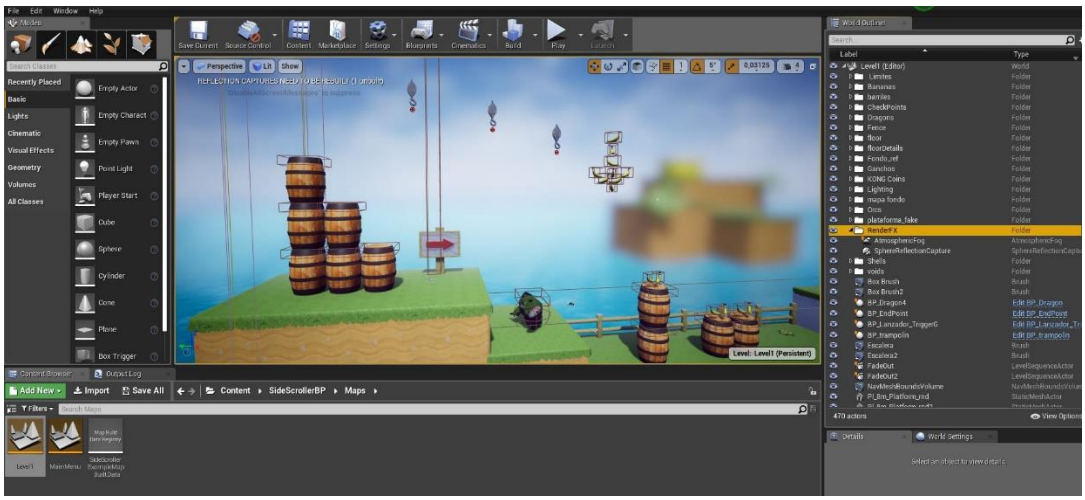

Figura 83 Captura del Unreal Editor con la vista del nivel

7.5.1 Pipeline de creación e importación de assets

En la creación del nivel se utilizan tanto assets obtenidos de manera gratuita en el mismo bazar de la tienda de Epic Games Launcher, como assets propios u obtenidos a partir de una base previa (descarga de modelos con permisos de uso), adecuándolos al proyecto con el programa de modelado 3ds Max.

7.5.1.1 Importación de assets desde 3ds Max

Para la importación y exportación de mallas desde 3ds Max a Unrel Engine 4, se ha utilizado el formato de exportación .fbx en la versión que aparece por defecto (FBX 2014/2015). Una vez exportamos el archivo, podemos añadirlo a nuestro proyecto de Unreal haciendo clic en el botón **Import** ubicado en el panel del Content Brower del Unreal Editor.

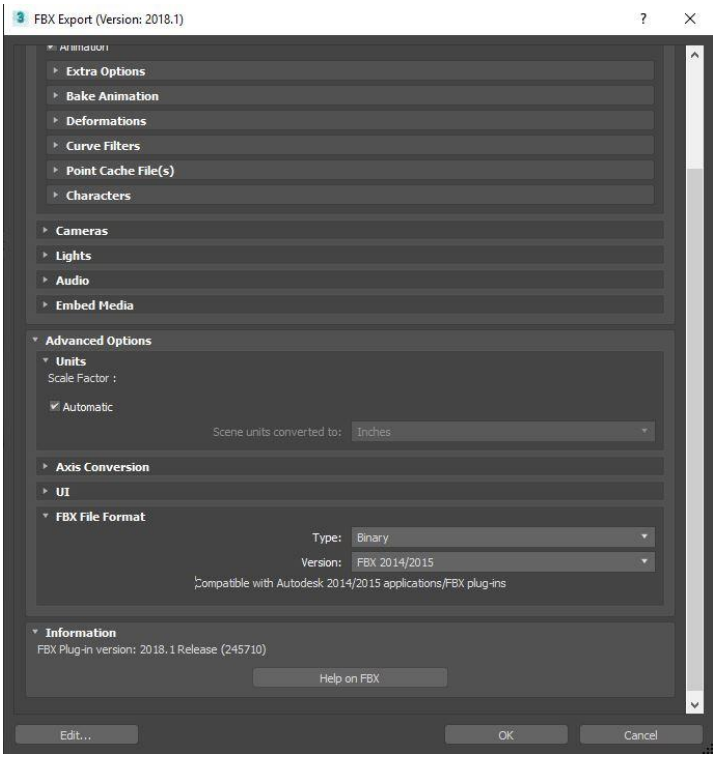

Figura 84 Captura de la exportación de una malla desde 3ds Max

Tras seleccionar el archivo que queremos importar, el motor lo reconocerá y nos mostrará una ventana de importación con diferentes opciones de importación. No es necesario modificar ningún parámetro en este caso, simplemente haremos clic en el botón de **Import**. Dependiendo del uso que queramos dar a las mallas importadas, luego podemos modificar su colisión accediendo al static mesh del modelo importado y eliminando su colisión.

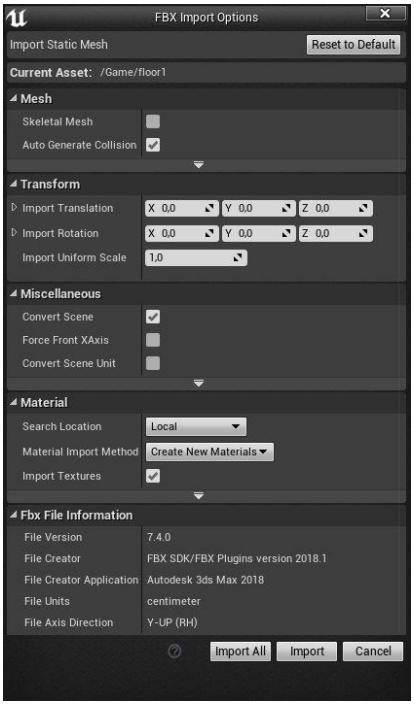

Figura 85 Captura de la ventana de importación de un archivo .fbx en Unreal Engine 4

Una vez importado el .fbx, el motor asigna automáticamente los materiales que tenía la malla en el proyecto de 3ds Max, aunque en ciertas ocasiones puede no hacerlo, por lo cual hay que reasignarlos de forma manual, accediendo al editor del mesh y seleccionando cada material en su sección correspondiente.

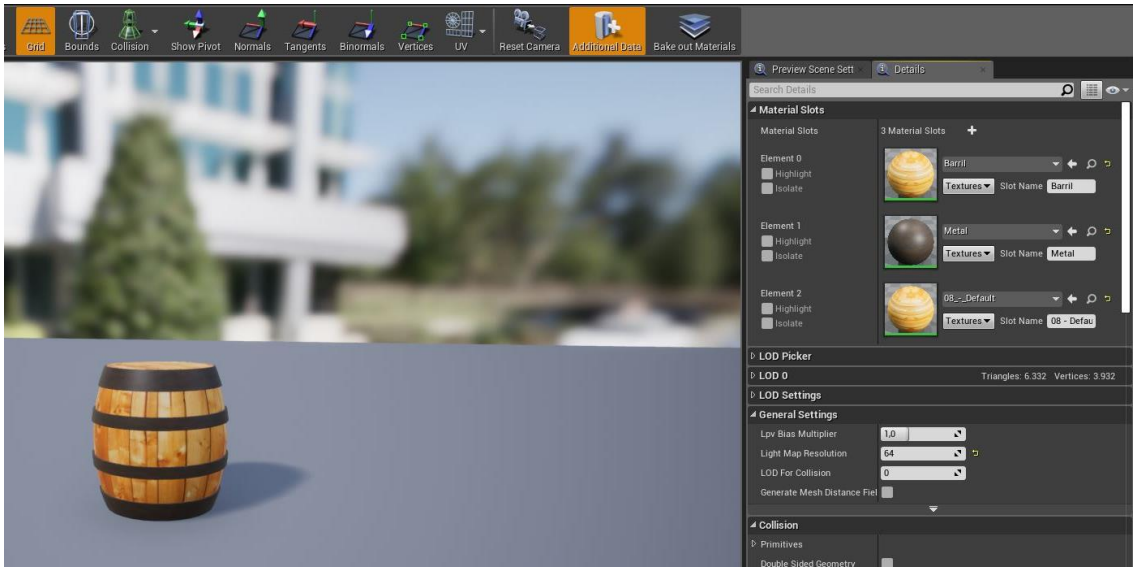

Figura 86 Captura del Unreal Editor en la ventana del mesh de un modelo

7.5.1.2 Importación de assets desde el bazar de Epic Games Launcher

Para la importación de assets desde el bazar, el proceso es mucho más sencillo al que se ha explicado anteriormente. En este caso, una vez obtengamos el asset en cuestión del bazar, nos aparecerá justo debajo de nuestros proyectos de Unreal en la pestaña Biblioteca del Epic Store Launcher.

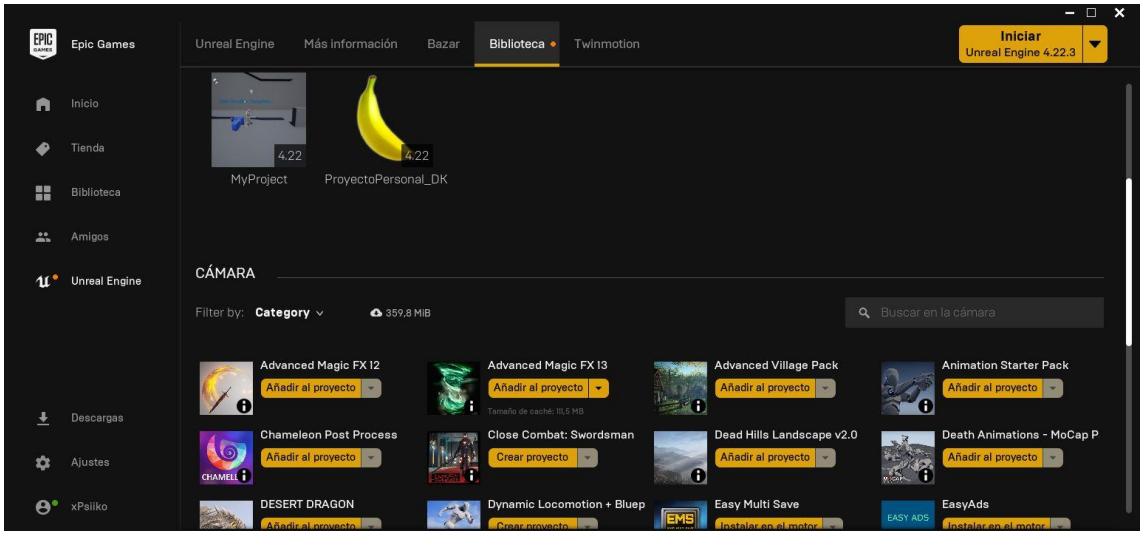

Figura 87 Captura del Epic Games Launcher

En el apartado Cámara veremos todos los assets que hayamos obtenido desde el bazar. Si no lo hemos instalado previamente, en el botón asociado al asset nos aparecerá dicha opción. Una vez instalado el mismo botón nos dará la opción de añadirlo al proyecto que deseemos.

7.5.2 Creación del terreno del nivel

Para la creación del terreno del nivel, se han utilizado los assets presentados en el apartado 6.2.2, que forman el escenario. La creación del escenario se lleva a cabo haciendo uso de los assets importados en el motor y las herramientas que ofrece el editor para poder mover, escalar, rotar, duplicar, etc. los diferentes elementos que añadamos en el nivel (ver Figura 88).

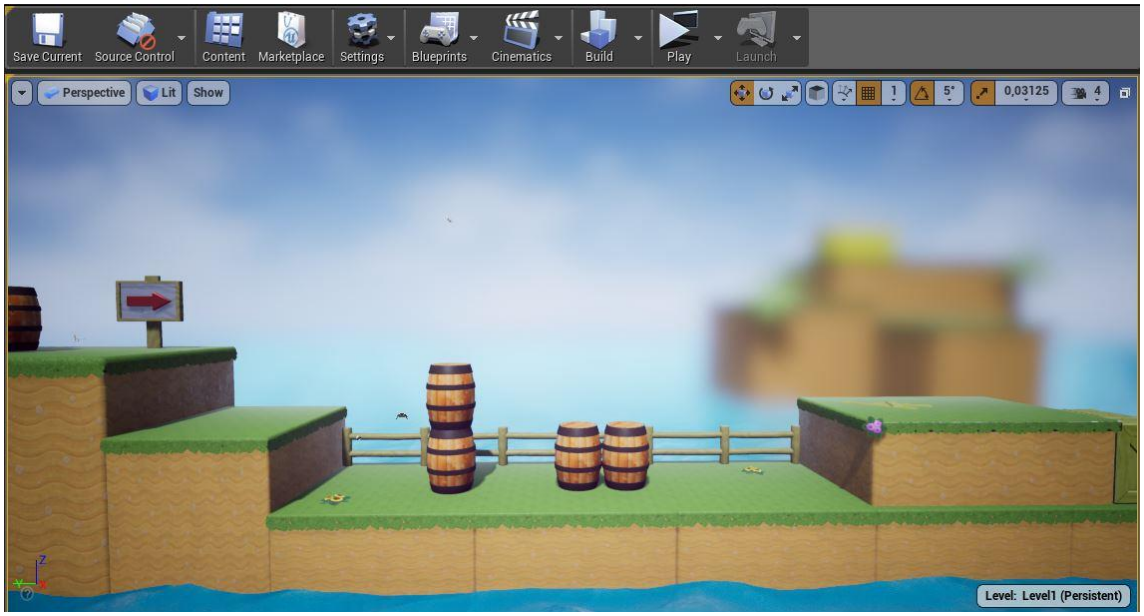

Figura 88 Captura desde el editor de Unreal de una parte del terreno del nivel

Hay que tener en cuenta que lo assets que forman el terreno cuentan con un collision shape, el cual se encargará de todas las colisiones con otros elementos. Esto es importante porque nuestro personaje se desplazará por encima de estos elementos. Por ese motivo debemos tener en cuenta la forma de estos collision shapes para evitar interacciones no deseadas.

Figura 89 Captura del barril desde el editor de Unreal

En el caso del barril, se ha añadido una caja de colisión en la parte superior del mesh para evitar el problema mencionado anteriormente. El mesh del barril ya cuenta con un collision shape. Sin embargo, debido a su forma, al agrupar más de un barril de manera continua, no permite que el jugador pueda desplazarse correctamente por encima (ver Figura 89).

7.5.3 Iluminación

En cuanto a la iluminación del juego, se utilizan dos tipos de iluminación ofrecidas por el motor, **Light Source** y **Sky Light**. El Light Source se encarga de la luz global del nivel, manteniendo todo el escenario iluminado de la forma en que nosotros ajustemos sus parámetros.

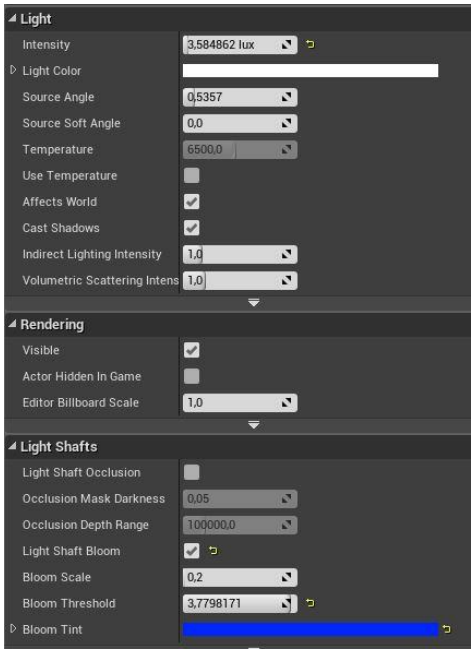

Figura 90 Parámetros del Light Source

Por otra banda, el **Sky Light** captura las partes distantes del nivel y las aplica como una luz. Se ha configurado en la opción de movilidad estacionario, que hace que sólo se capture la iluminación, sombras y rebotes de luz a partir de la geometría estática de los modelos. Todas las demás luces serán dinámicas.

Para realizar los cálculos de luz de forma correcta, se ha utilizado el componente **Lightmass Importance Volume**, encargado de calcular toda la iluminación dentro de un entorno indicado.

La forma de añadir estos componentes es como todos los actores (elementos) del proyecto, se añaden al escenario arrastrando el componente deseado con el ratón o añadiéndolo como componente de algún blueprint.

7.6 Sonorización

El proceso de sonorización de nuestro juego se compone por la música de fondo del nivel, el sonido ambiental de nivel y menú, y los efectos de sonido producidos por ciertas interacciones del jugador durante el gameplay.

Para poder hacer uso de un sonido, primero hay que importarlo al proyecto en formato .wav, único formato de audio soportado. Una vez importado, el archivo se convierte en formato **Sound Wave** (formato de audio que utiliza Unreal).

7.6.1 Música y sonido ambiental

En el caso de la música de fondo y la ambiental, hacemos clic derecho sobre el Sound Wave y seleccionamos la opción **Create cue** (ver Figuras 91 y 92).

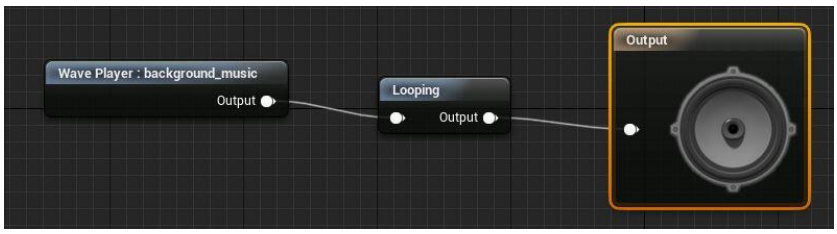

Figura 91 Blueprint del Sound cue de la música de fondo

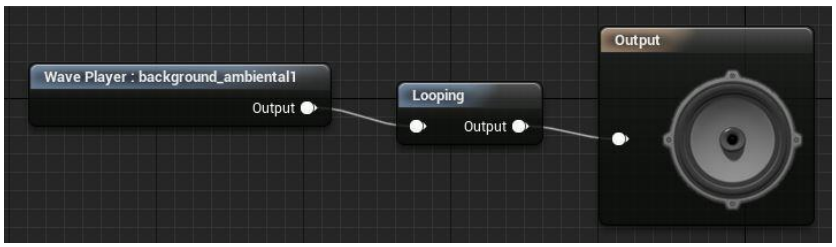

Figura 92 Blueprint del Sound cue de la música ambiental

Una vez creas los **Sound cue**, añadimos entre el nodo **Wave Player** y el **Output** un Looping indicando un loop indefinido para tener una reproducción constante de las pistas durante el transcurso del nivel.

También ajustaremos el volumen del output de ambas blueprints, dándole un mayor volumen de manera predefinida a la música de fondo que a la ambiental.

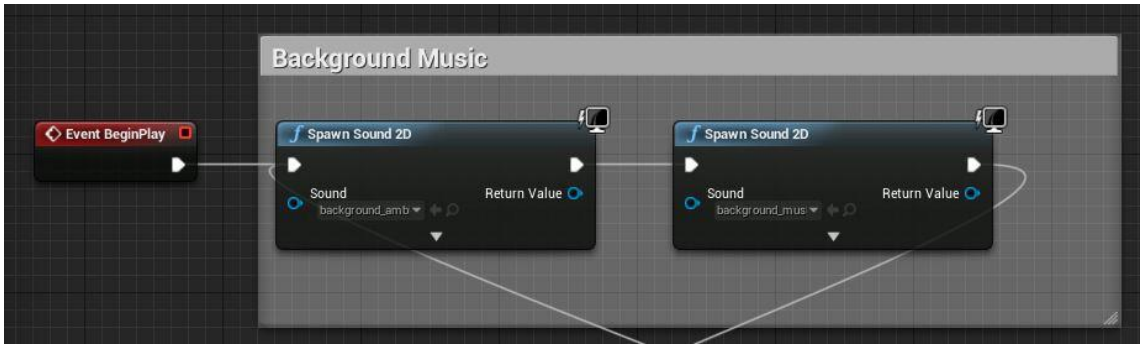

Figura 93 Blueprint del nivel con la inicialización de ambas pistas de sonido

Con ambos Sound cue configurados, sólo debemos crearlos en el blueprint del nivel. Una vez este inicialice haremos un **Spawn Sound 2D** asignando cada pista a reproducir (ver Figura 93).

7.6.2 Efectos de sonido

Para los efectos de sonido el procedimiento es más sencillo, sólo se necesita llamar al nodo **Play Sound 2D** en el blueprint donde queramos reproducir el sonido (ver Figura 94).

Figura 94 Nodo Play Sound 2D del blueprint Trampolín

7.7 Implementación de mecánicas de juego

7.7.1 Implementación de enemigos

En este apartado se explicará la implementación del comportamiento de los diferentes enemigos y las mecánicas que surgen al interactuar con el jugador. Hemos de tener en cuenta que cada tipo de enemigo cuenta con un blueprint distinto, aunque en ciertos aspectos tengan una gran similitud entre tipos.

A continuación, veremos de manera más detallada y específica, como se lleva a cabo el comportamiento e interacciones de los tipos de enemigos que el juego presenta.

7.7.1.1 Orco

Encontramos los orcos a lo largo del nivel, ubicados encima del terreno del escenario. Su comportamiento consiste en desplazarse de un punto A a un punto B y viceversa, indefinidamente. Para implementar este comportamiento añadimos en el blueprint del orco el componente **InterpToMovement**, que se encargara de interpolar el movimiento del orco entre los puntos de control que le asignemos (ver Figura 95).

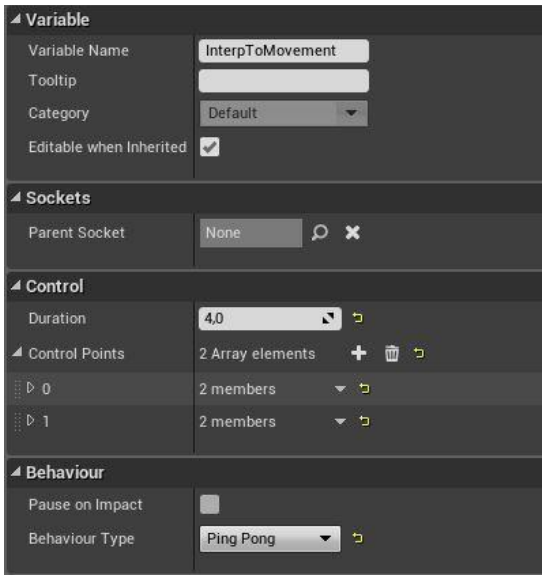

Figura 95 Captura de la configuración del componente InterpToMovement

En la configuración del componente, añadimos dos Control Points que representan el punto A y B, y ajustamos la duración en la que queremos que tarde el desplazamiento de un punto al otro. También ajustamos el comportamiento en **Behaviour Type** como Ping Pong, haciendo que vaya de un punto a otro constantemente.

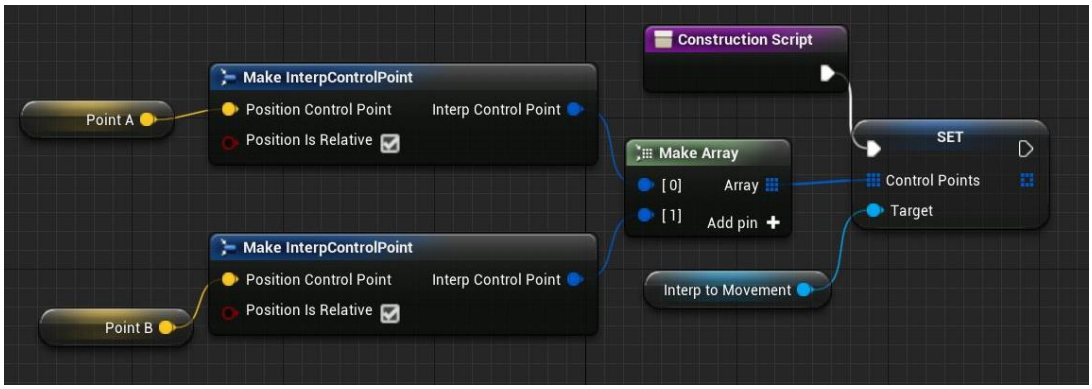

Figura 96 Captura del blueprint de asignación de los puntos A y B cómo Control Points en la Interpolación del orco

A continuación, creamos dos variables de tipo vector y las hacemos visibles en el editor para poder desplazarlas a nuestro gusto y asignar los puntos de interpolación del ogro de manera sencilla. Una vez hemos creado las dos variables, hay que asignarlas como Control Points del componente **InerpToMovement**. Para ello, nos vamos a la ventana de **Contruction Script** del blueprint del orco, y creamos el flujo de nodos de la Figura 96.

1. Cogemos la referencia de los puntos A y B.
2. Conectamos los puntos al nodo **Make InterpControlPoint** para convertirlos en Control Points y con ellos creamos un array.
3. El array obtenido lo establecemos como array de Control Points del componente **InterpToMovement**.

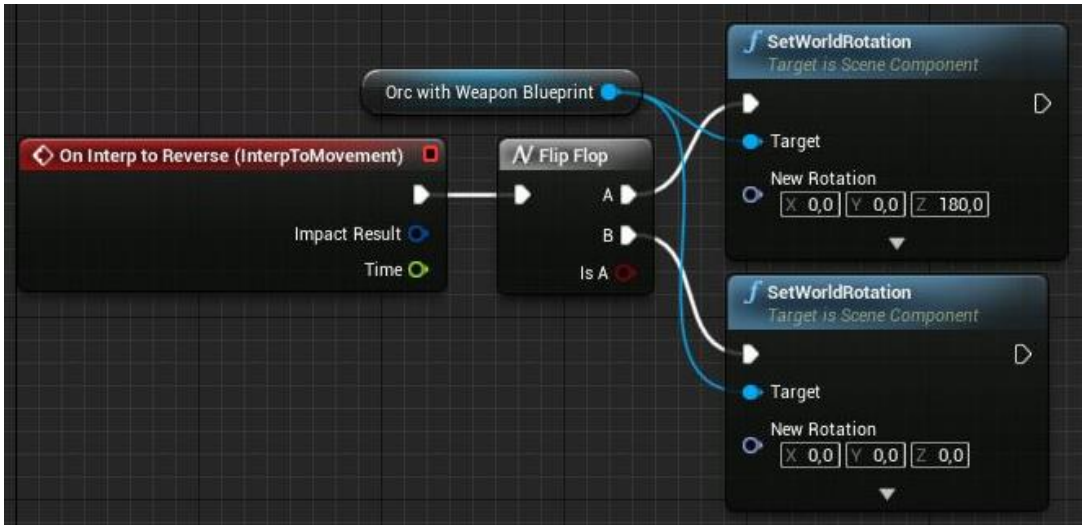

Figura 97 Captura del blueprint de la lógica del comportamiento del orco

Por último, establecemos la lógica principal llamando al evento **On Interp to Reverse** que se ejecutará de forma automática ya que seleccionaremos la casilla AutoActivate del componente **InterpToMovement**. En este blueprint (ver Figura 97), haremos un Flip Flop de los puntos A y B para ir alternando la dirección del orco e iremos rotando el modelo del orco 180 grados positivo y negativo en cada punto para mantener la posición del cuerpo del modelo acorde con la dirección en la que se desplaza. Mientras se desplaza, el orco ejecuta la animación de caminar asociada al esqueleto del componente. Esta animación es constante, sin cambios de estado, se va reproduciendo continuamente.

Una vez tenemos el comportamiento del orco acabado, pasamos a implementar las interacciones con el jugador.

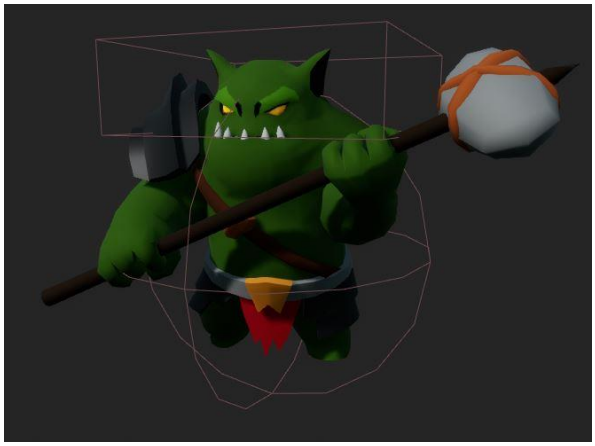

Figura 98 Captura del apartado viewport del blueprint del orco

La mecánica de interacción entre el orco y el jugador se activan mediante triggers proporcionados por los diferentes componentes de colisión creados en el orco.

El orco cuenta con dos componentes de colisión, una caja de colisión en la parte de la cabeza y una cápsula de colisión para el resto del cuerpo. Cuando el jugador colisione con la caja, el orco será derrotado y destruido durante el transcurso de todo el nivel, proporcionando un impulso de rebote al jugador hacía arriba. En cambio, si lo hace con la cápsula, será el propio jugador quien pierda una vida.

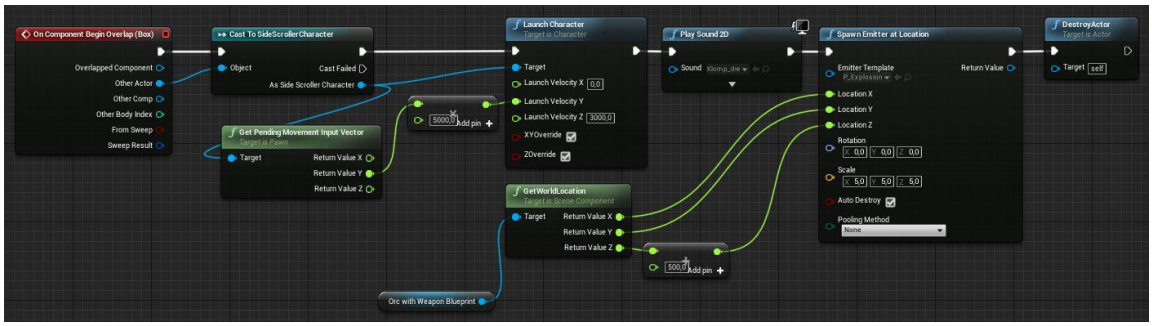

Figura 99 Captura del blueprint de la lógica de colisión de la caja

La implementación de la lógica de la colisión con la caja se detecta con el evento **On Component Begin Overlap**, que se ejecuta cuando detecta que otro componente de la escena entra en contacto. Cuando el actor entra en contacto, hacemos un cast (función que permite acceder a todas las propiedades) para obtener nuestro personaje y su vector de movimiento. A continuación, procedemos a hacer lo siguiente:

1. Llamar al nodo **Launch Character** para simular el rebote en la cabeza del orco, pasándole cómo parámetros al personaje y la coordenada Y del vector de movimiento multiplicada

por 5000. Sólo se aplica velocidad de lanzamiento en los ejes Y (dependiente del vector de movimiento del personaje) y Z (valor fijo en 3000).

2. Una vez lanzado el personaje, llamamos a **Play Sound 2D** para reprooducir el sonido del orco recibiendo el golpe.
3. Acto seguido, obtenemos la posición del orco en el nivel llamando al nodo **GetWorld Location** y creamos un efecto de partícula ajustando un poco la posición obtenida para simular la destrucción del orco con **Spawn Emitter at Location**.
4. Por último, destruimos al orco llamando al nodo **Destroy Actor**.

Para la lógica de colisión de la cápsula, tenemos que tener en cuenta 2 posibles interacciones; una por la colisión con el personaje en estado **Dashing** y la otra sin él.

Cuando ocurre la primera, el personaje colisiona con la cápsula del orco mientras hace la acción de dash, proporcionándole la capacidad de poder derrotar al orco. En cambio, si colisiona con la cápsula en cualquier otro estado, el jugador será derrotado perdiendo una vida.

Para explicar la lógica de colisión de la cápsula, se ha dividido el blueprint en varias partes que veremos a continuación:

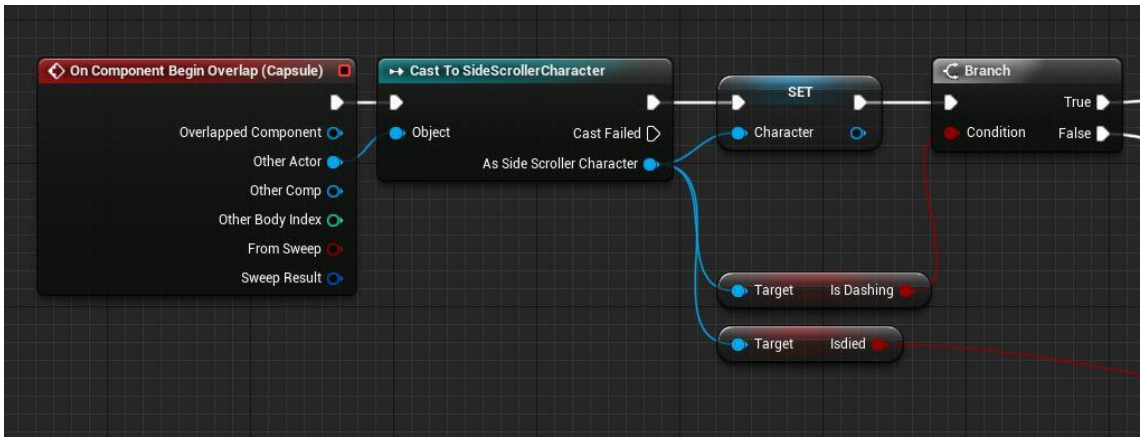

Figura 100 Captura del blueprint de la lógica de colisión de la cápsula parte 1

Como en el blueprint anterior, la colisión de la cápsula se detecta con el evento **On Component Begin Overlap**. Cuando el actor entra en contacto, obtenemos el personaje y sus variables boleanas **IsDashing** y **Isdied** proporcionadas por el cast. A partir de aquí creamos un Branch condicionado por la variable **IsDashing** que dependiendo si el jugador está haciendo un dash o no, el blueprint seguirá un flujo de ejecución u otro (ver Figura 100).

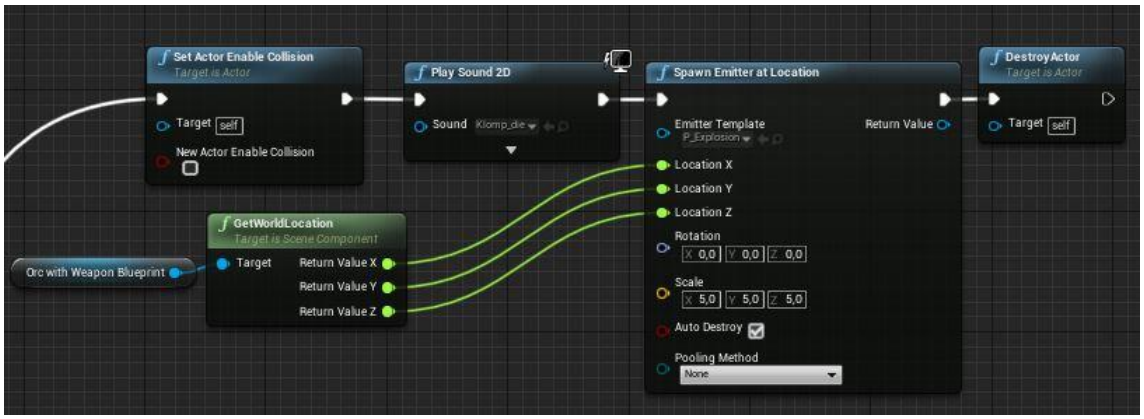

Figura 101 Captura del blueprint de la lógica de colisión de la cápsula parte 2

En caso que sea cierto, el flujo de ejecución será el siguiente (ver Figura 101):

1. Desactivamos la colisión del orco con **Set Actor Enable Colision** para evitar que el evento se vuelva a activar y detecte otra colisión distinta.
2. Llamamos a **Play Sound 2D** para reproducir el sonido del orco recibiendo el golpe.
3. Obtenemos la posición del orco en el nivel llamando al nodo **GetWorld Location** y creamos un efecto de partícula ajustando un poco la posición obtenida para simular la destrucción del orco con **Spawn Emitter at Location**.
4. Destruimos al orco llamando al nodo **Destroy Actor**.

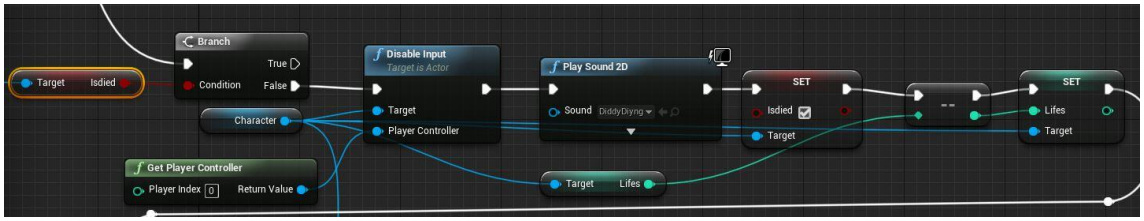

Figura 102 Captura del blueprint de la lógica de colisión de la cápsula parte 3

En caso que la condición sea falso, el flujo de ejecución será éste (ver Figura 102):

1. Comprobamos si el jugador está muerto haciendo uso de un **Branch** con la condición dada por la variable del personaje **Isdied** (éste se ha incluido posteriormente a la implementación inicial para corregir ciertos fallos en la interacción del personaje simultáneamente con ciertos enemigos).
2. Si la condición es falsa, deshabilitamos el Input del jugador para que no pueda controlar el personaje que acaba de ser derrotado, y llamamos a **Play Sound 2D** para reproducir el sonido de muerte del personaje.
3. Modificamos el estado de la variable **Isdied** a true y restamos 1 la variable de vidas del personaje obtenida por el cast hecho anteriormente.

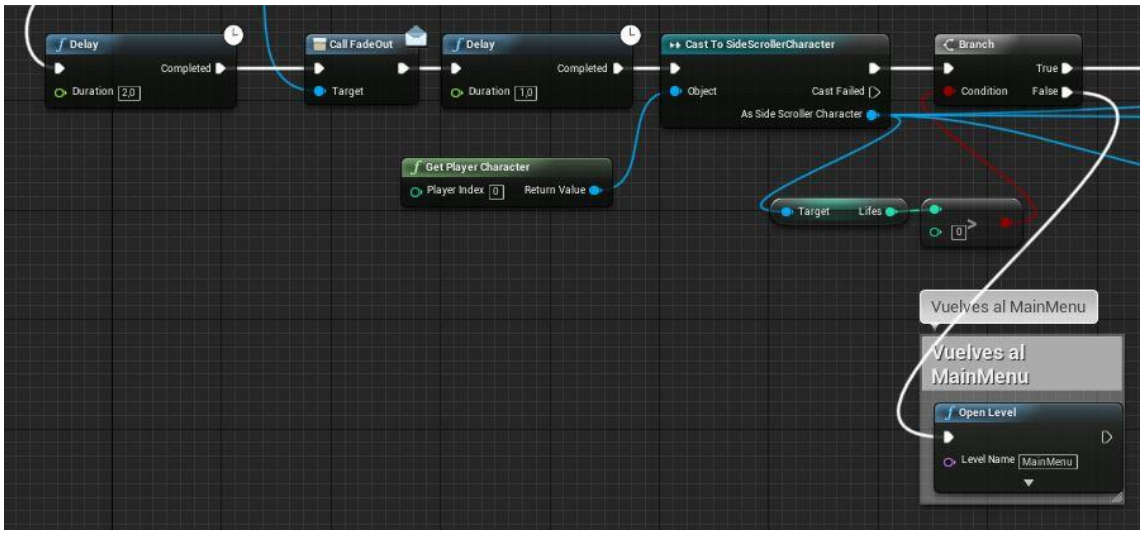

Figura 103 Captura del blueprint de la lógica de colisión de la cápsula parte 4

4. A continuación, como se ve en la Figura 103, aplicamos un delay de 2 segundos para dar tiempo a que se complete la animación de muerte del personaje (la implementación de la lógica de las animaciones se explicará posteriormente en su apartado respectivo) y llamamos al **Level Sequence FadeOut** (secuencia creada que aplica un efecto de transición sobre el gameplay).
5. Una vez llamado el **FadeOut**, aplicamos otro delay de 1 segundo para dar tiempo a la ejecución de la secuencia y volvemos a hacer un cast para obtener el personaje y su variable de vida.
6. Con la variable de vida, comprobamos si es mayor que 0 para utilizar su resultado cómo condición para el **Branch** creado a continuación. Si al personaje no lo quedan vidas, la condición del **Branch** será falsa y se llamará al nodo **Open Level**, dando por perdida la partida y devolviendo al jugador al menú principal.

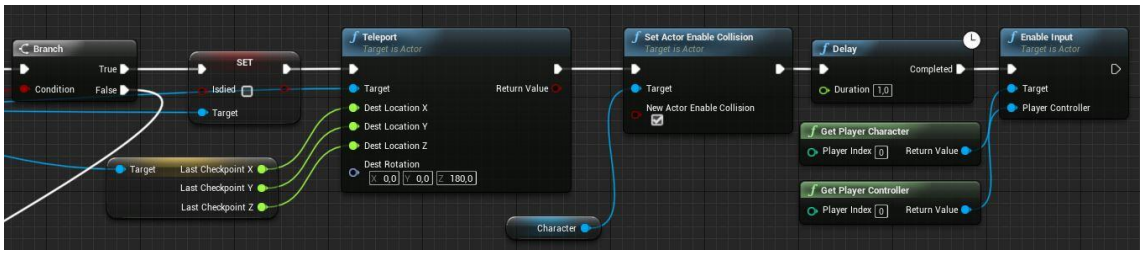

Figura 104 Captura del blueprint de la lógica de colisión de la cápsula parte 5

7. En el caso que al jugador le quede 1 o más vidas, la condición del **Branch** será cierta y seguirá el flujo que vemos en la Figura 104.
8. Modificamos el estado de la variable **Isdied** a false y llamamos al nodo **Teleport** pasándole la referencia del personaje y la posición del último checkpoint alcanzado (la lógica y funcionamiento del checkpoint se explicará posteriormente en su respectivo apartado).
9. Por último, habilitaremos nuevamente la colisión del personaje y aplicaremos un delay de 1 segundo para dar tiempo a que se complete el teleport. Acto seguido habilitaremos de nuevo el input del jugador permitiendo tener el control del personaje.

7.7.1.2 Caracola

Las caracolas también nos las encontraremos ubicadas sobre diferentes partes del terreno que forman el nivel. Su comportamiento es diferente al del orco, ya que esta se mantiene siempre en la misma ubicación sin desplazarse. Sin embargo, cuenta con dos estados definidos por una composición de animaciones (ver Figuras 105 y 106).

- **Reposo**: Cuando la caracola está en reposo, tanto su cabeza como sus púas están escondidas bajo el caparazón. En este estado, el jugador puede matarla tanto saltando encima suya como golpeándole con el dash.

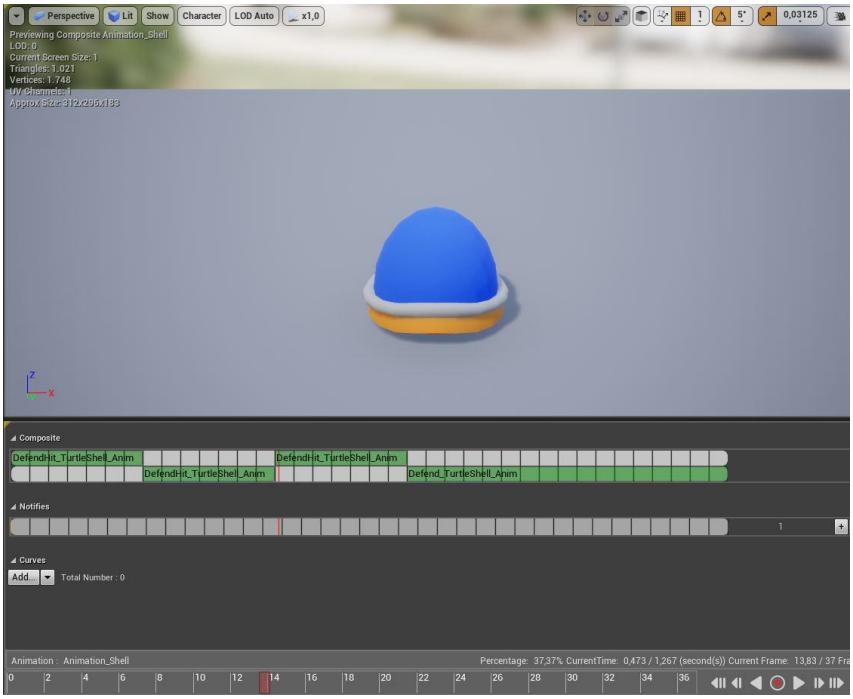

Figura 105 Captura de la composición de la animación de la caracola en estado reposo

- **Defensivo**: Si se encuentra en estado defensivo, el jugador no podrá matarla saltándole encima ya que te matará por la colisión con las púas. La única forma de derrotarla en este estado es golpeándole con el dash.

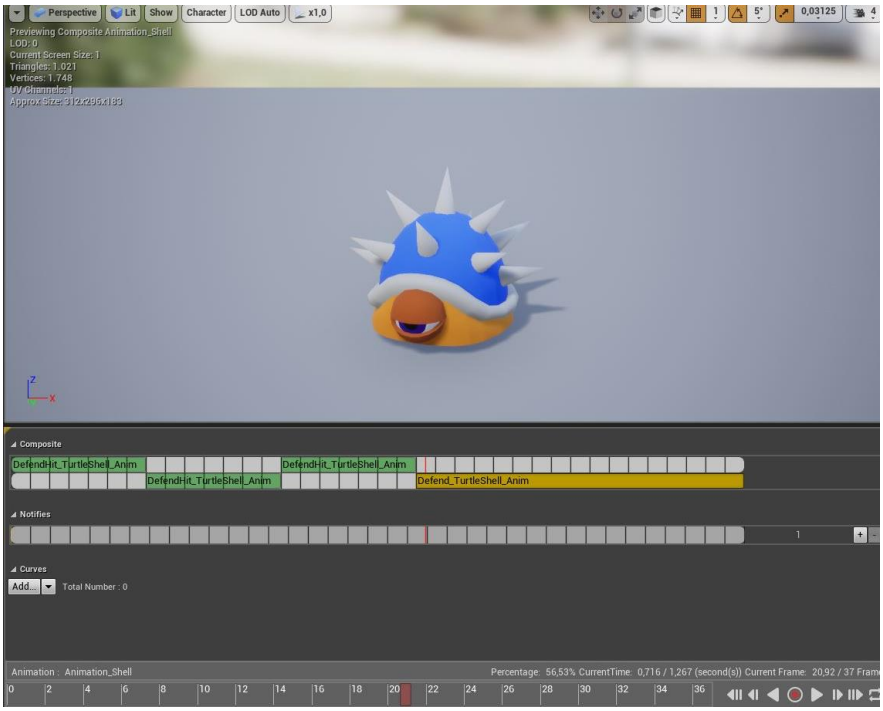

Figura 106 Captura de la composición de la animación de la caracola en estado defensivo

Como en el caso anterior, la caracola también cuenta con dos componentes de colisión que utiliza como trigger para detectar la interacción con otros actores (ver Figura 107). Estos componentes son dos cápsulas que van vinculadas a diferentes partes del esqueleto de la caracola, para así poder cambiar su ubicación según en qué estado de animación se encuentre, es decir, las cápsulas de colisión se mueven de sitio acorde con el movimiento que haga el esqueleto de la caracola.

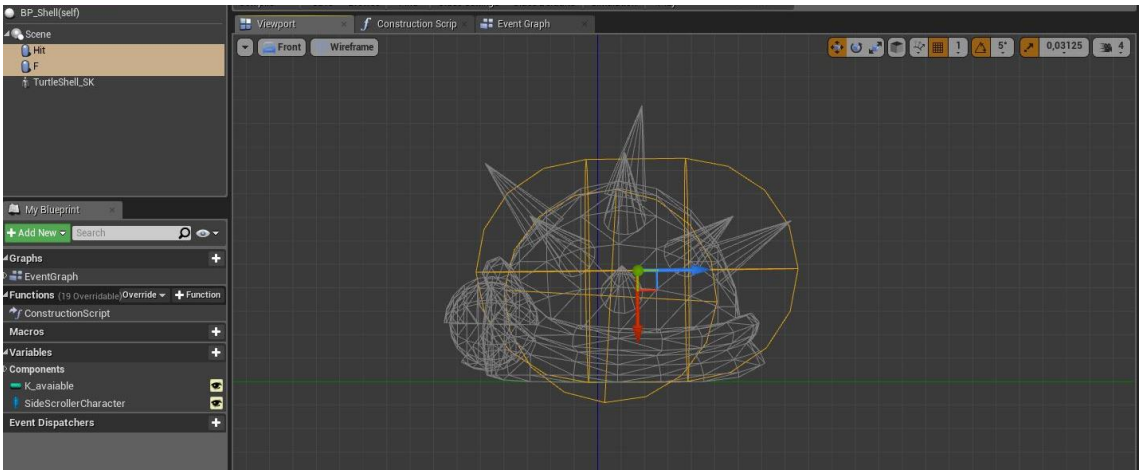

Figura 107 Captura del viewport del blueprint de la caracola

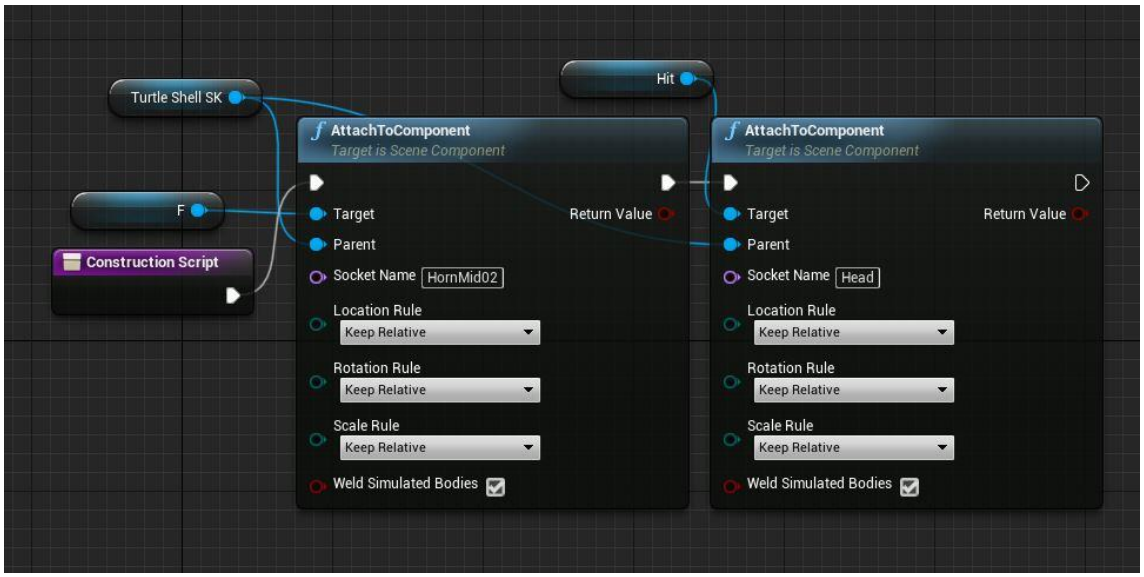

Figura 108 Blueprint con la lógica que vincula las cápsulas de colisión a partes del esqueleto

En la Figura 108, podemos ver como llamamos al nodo **AttachToComponent** y vinculamos la primera cápsula de colisión al socket HornMid02, hueso ubicado en el centro de las púas. En el segundo nodo **AttachToComponent** asociamos la segunda cápsula de colisión al socket Head, hueso ubicado en el cráneo de la caracola. Ambos AttachToComponent llevan conectados como parent al modelo de la caracola para poder acceder a su esqueleto.

Una vez tenemos vinculadas las cápsulas de colisión, pasamos a la implementación de su lógica de interacción.

Esta parte de la implementación es la misma que la que se ha visto en el orco, sólo cambian las referencias de los modelos, las cápsulas de colisión, los sonidos y el orden de algún nodo. Por ese motivo, sólo se explicará la parte de variación que existe en el blueprint de un segundo tipo de caracola.

Para ponernos en contexto, existe un segundo tipo de caracola completamente idéntico al anterior excepto en que cuando esta caracola muere, modifica el valor de una variable del blueprint **GameState** (blueprint del estado del juego que sólo contiene variables de control del nivel).

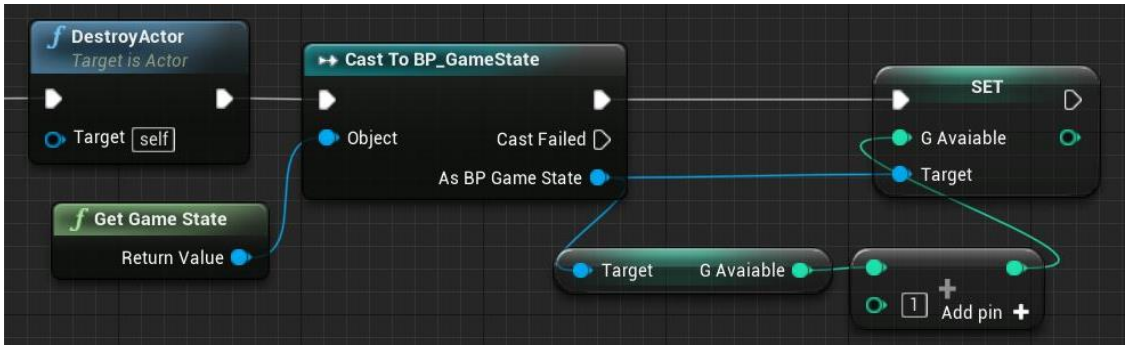

Figura 109 Parte del blueprint del segundo tipo de caracola con la lógica que se ejecuta después de ser derrotada

Esta modificación actúa como trigger que dará como resultado un cambio en el estado del juego que explicaremos más adelante.

7.7.1.3 Dragón

Los dragones los veremos ubicados en ciertas zonas del espacio aéreo del escenario. Su comportamiento en cuanto al movimiento es estático, permanecerá siempre en el mismo lugar mientras itera de forma permanente su animación de vuelo en reposo.

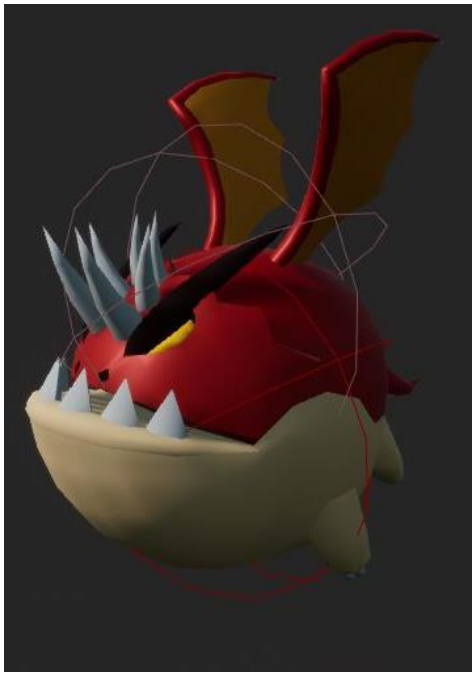

Figura 110 Captura del viewport del blueprint del dragón

Sus componentes de colisión están formados por una cápsula y una esfera de colisión encargadas de detectar las interacciones que se produzcan con otros actores.

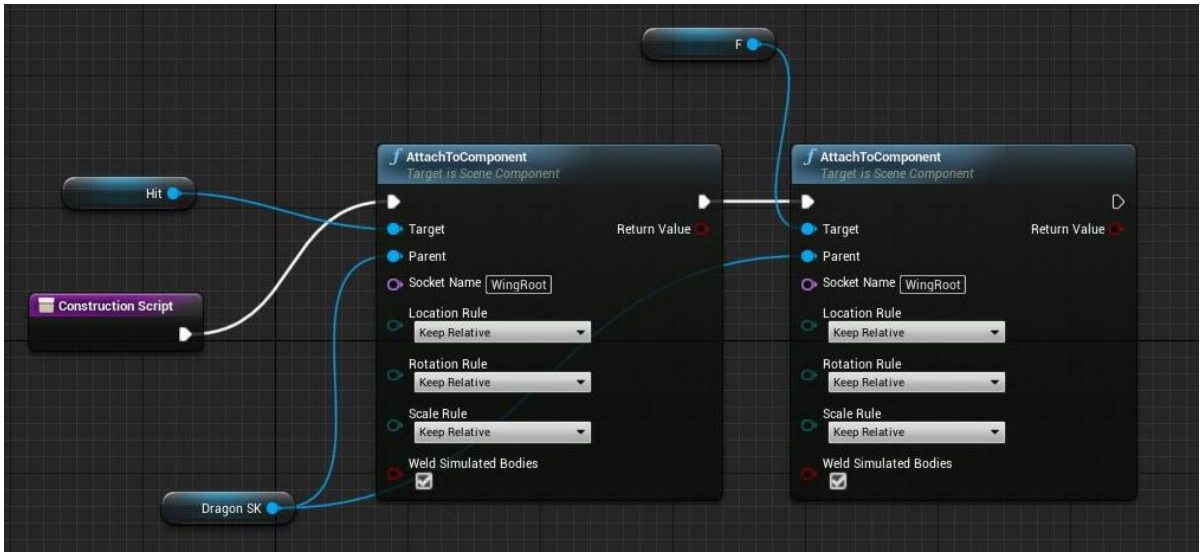

Figura 111 Captura del viewport del blueprint del dragón

Ambos componentes de colisión van asociados a la misma parte del esqueleto del dragón. El socket WingRoot se encuentra en la zona central superior del esqueleto. Hemos asociado tanto la cápsula como la esfera en el mismo lugar, para que cuando el dragón ejecute su animación de vuelo, ambas colisiones se desplacen junto al esqueleto evitando que la zona de colisión pueda variar.

La cápsula de colisión ubicada en la parte superior del dragón se encarga de detectar la colisión que hará que el jugador pueda derrotar al dragón. La esfera de colisión que se encuentra en la parte inferior es la encargada de detectar si el jugador entra en contacto con el dragón sin haber saltado por encima (haber colisionado con la cápsula), de manera que el jugador será derrotado perdiendo una vida.

La lógica de colisión e interacción implementada en el dragón es la misma que se ha enseñado en el orco, con la diferencia que en el blueprint de la esfera de colisión ubicada en la cabeza del dragón, el rebote que se produce utilizando el nodo **LaunchCharacter**, que recibe solamente el eje Z de velocidad, dándole una velocidad superior provienente de la variable **Jump Z Velocity** obtenida del cast del movimiento del personaje. Esto hace que el rebote con el dragón permita llegar más alto que con el resto de enemigos (ver Figura 112).

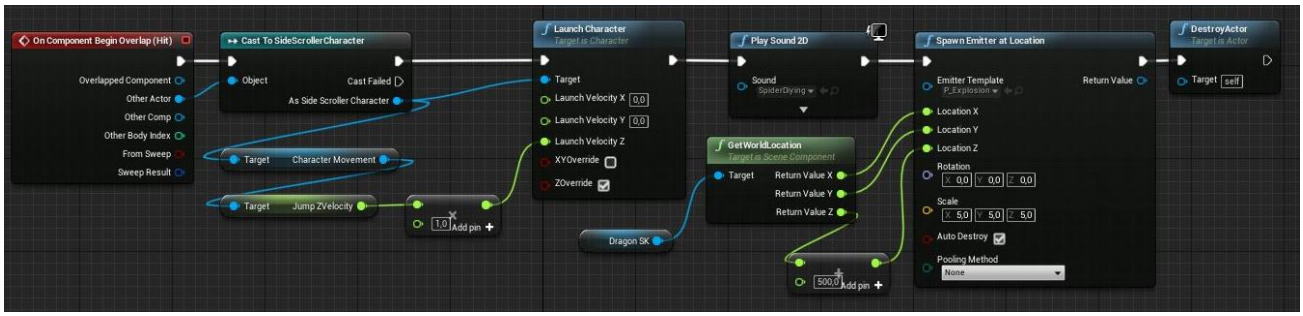

Figura 112 Captura del blueprint del dragón de la lógica de colisión de la cápsula

Además de esto, también se gestiona la muerte del dragón de manera diferente a la de los demás enemigos, ya que es necesario poder hacer reaparecer los diferentes dragones en el nivel si el jugador muere y reaparece, por el motivo que la interacción con ciertos dragones es imprescindible para poder avanzar en ciertas partes del nivel. Para ello, cuando el jugador elimina a un dragón, en vez de llamar al nodo **Destroy Actor** para eliminarlo de la escena, se llamarán a los nodos **Set Visibility** y **Set Actor Enable Collision** para desactivar tanto su visibilidad como colisión cuando sean eliminados. Cuando el jugador muera, se llamará a la función **Respawn Dragones** dónde se volverán a llamar estos dos nodos activando nuevamente su visibilidad y colisión (durante el testing se ha decidido gestionar todas las muertes de los enemigos de la misma forma).

7.7.2 Implementación de objetos

La implementación de los objetos no es más que la aplicación de un conjunto de mecánicas dónde el trigger pasa a ser el objeto en cuestión. Gracias al modelo del objeto, el jugador puede distinguir de manera anticipada la posible funcionalidad de cada uno de ellos antes de interactuar con él. Todos los triggers funcionan a través de la colisión con ellos. Sin embargo, cada tipo de objeto cuenta con una mecánica distinta y, por lo tanto, un blueprint distinto.

A continuación, se mostrarán las implementaciones de los objetos existentes en el juego:

7.7.2.1 Banana

Las bananas son el tipo de objeto más habitual que podemos encontrar y recolectar durante el recorrido del nivel. Pueden estar ubicadas en lugares dónde las veamos a simple vista, o "escondidas" en diferentes zonas del nivel, teniendo siempre la posibilidad de poder cogerlas. También tienen la peculiaridad que dan vueltas constantemente sobre su eje vertical, haciendo que llamen la atención del jugador.

El trigger de la banana viene dada por la colisión que se produce con ella, mediante su caja de colisión.

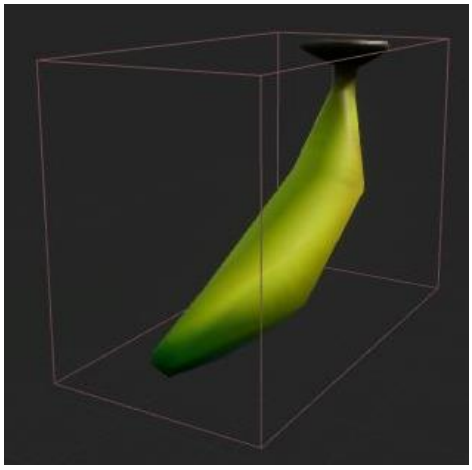

Figura 113 Captura del viewport del blueprint de la banana

Cuando el personaje colisiona con la caja, la banana desaparece y se suma en el contador de bananas que posee el HUD del jugador. A continuación, se explicará por pasos la implementación de la lógica del blueprint que vemos en la Figura 114.

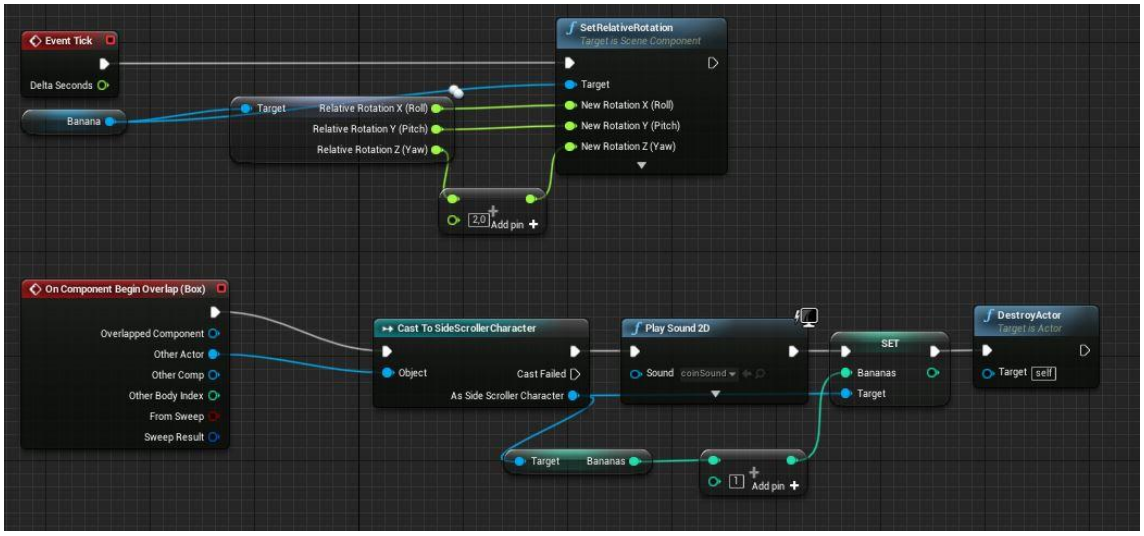

Figura 114 Captura del blueprint de la lógica de la banana

1. Utilizando el **Event Tick** que es llamado en cada fotograma, añadimos el nodo **SetRelativeRotation** para poder modificar la rotación de la banana.
2. En el target del este nodo le pasamos el modelo de la banana y en los nuevos ejes de rotación conectaremos los ejes relativos de rotación de la banana, sumando 2 unidades en el eje Z. Esta suma en el eje Z hará que, en cada frame, el valor vaya aumentando y la banana gire sobre su eje Z.
3. En el evento **On Component Begin Overlap** de la caja de colisión, haremos un cast para obtener el personaje y su variable **Bananas** que se encarga de llevar la cuenta de las bananas totales recogidas.
4. A continuación, llamamos al nodo **Play Sound 2D** y reproducimos el sonido escogido para la recolección de una banana.
5. Haciendo uso de la variable **Bananas** del personaje, modificamos su valor sumándole 1 al contador global.
6. Por último, llamamos al nodo **DestroyActor** para eliminar la banana recogida.

7.7.2.2 Monedas KONG

Las monedas KONG están formadas por 4 monedas coleccionables únicas por cada nivel. Cada una de ellas representa una letra de la palabra KONG. Estos objetos coleccionables están escondidos a lo largo del nivel, haciendo que el jugador deba explorar cada rincón del escenario para poder hacerse con todas ellas.

Al igual que las bananas dan vueltas sobre su eje vertical, y, para accionar su trigger hay que colisionar con su caja de colisión.

Cada moneda KONG cuenta con un blueprint independiente, dando la posibilidad en un futuro de aplicar comportamientos y mecánicas más diferenciadas dependiendo la moneda.

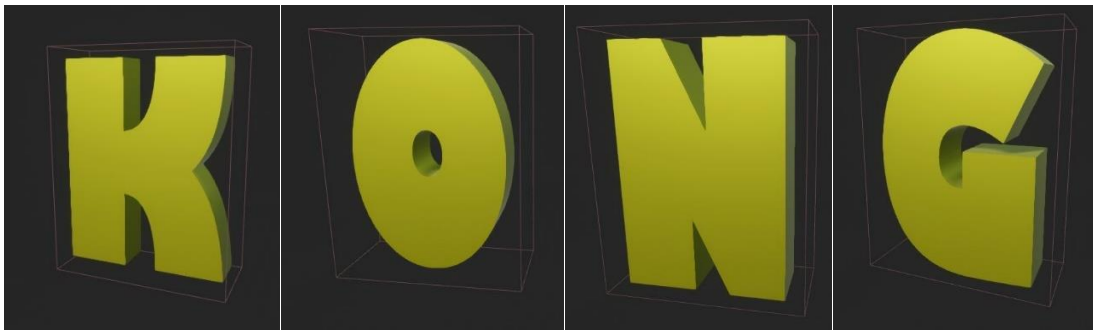

Figura 115 Captura del viewport del blueprint de las monedas KONG

Cuando el personaje colisiona con la caja de colisión de la moneda, ésta desaparece del nivel, apareciendo un Sprite en el HUD del jugador con la letra de la moneda recogida. La implementación de la lógica es la misma en las 4 monedas, por ese motivo explicaremos solamente la implementación del blueprint de una de ellas.

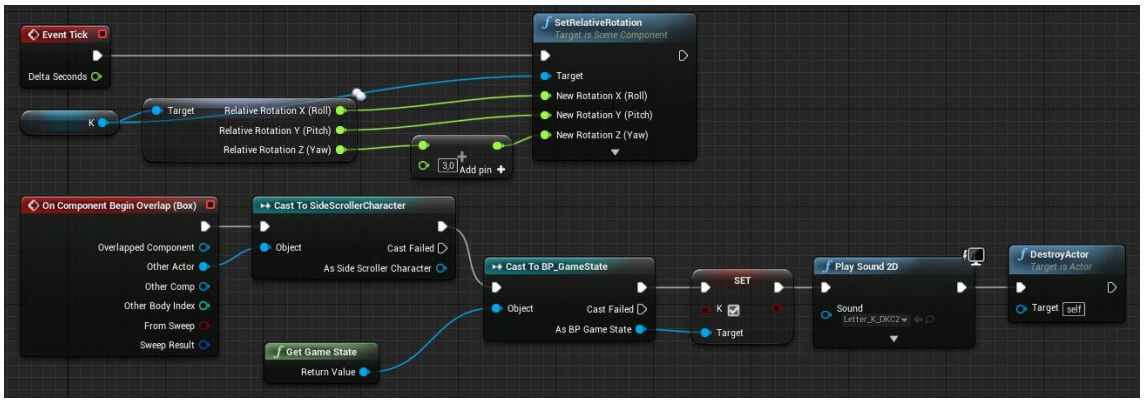

Figura 116 Captura del blueprint de la lógica de la moneda K

En la Figura 116 podemos ver el blueprint con la lógica que se explica en los pasos siguientes:

1. En el nodo **Event Tick** seguimos el mismo procedimiento explicado anteriormente en la banana cambiando el modelo y los ejes de rotación de la banana por los de la moneda.
2. En el evento **On Component Overlap** de la caja de colisión haremos un cast del Game State del nivel para poder modificar el estado de la variable de control K a cierto. Esta variable también hace de trigger en otros eventos de otras blueprint, como puede ser el HUD.
3. Luego, llamamos al nodo **Play Sound 2D** para reproducir el sonido escogido para la recolección de la letra en cuestión.
4. Por último, llamamos al nodo **Destroy Actor** y eliminamos la moneda en cuestión.

7.7.2.3 Moneda de final de nivel

La moneda de final de nivel se encuentra ubicada en la parte final del escenario y como bien indica su nombre, la mecánica que desempeña es finalizar el nivel, dándolo por completado.

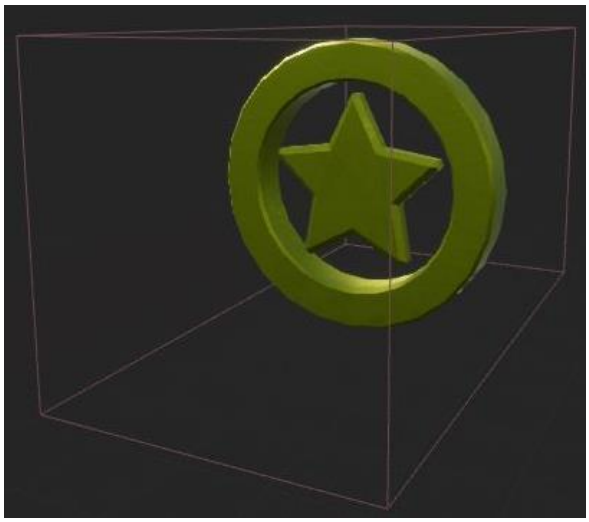

Figura 117 Captura del viewport del blueprint de la moneda de final de nivel

Al producirse la colisión del personaje con la caja de colisión de la moneda, ésta desaparece y da lugar a la reproducción de un sonido de celebración junto a la acción del baile del personaje.

El blueprint de la lógica de la moneda final de nivel viene dado por los dos eventos que utilizan las lógicas de los dos objetos anteriores: el **Event Tick** y el **On Component Begin Overlap** de la caja de colisión. En el caso del primer evento, la lógica es la misma que las dos anteriores, como podemos ver en la Figura 118.

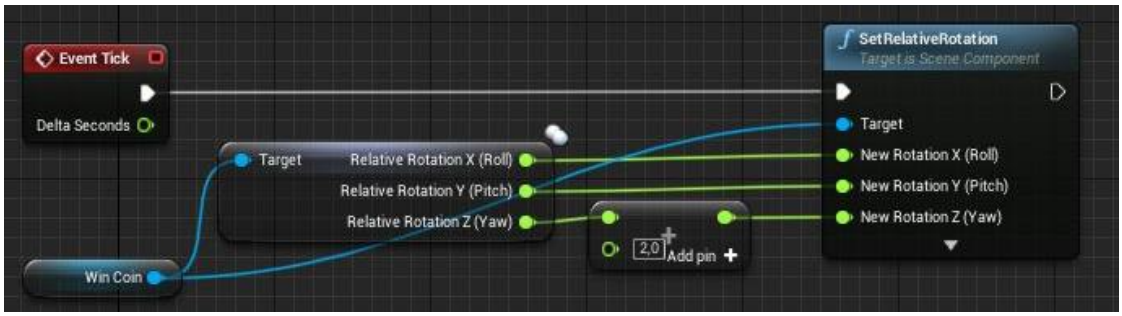

Figura 118 Captura del blueprint de la lógica de la moneda de final de nivel parte 1

Por otra parte, la lógica que aparace en la Figura 119 sigue los pasos de implementación que se explican a continuación.

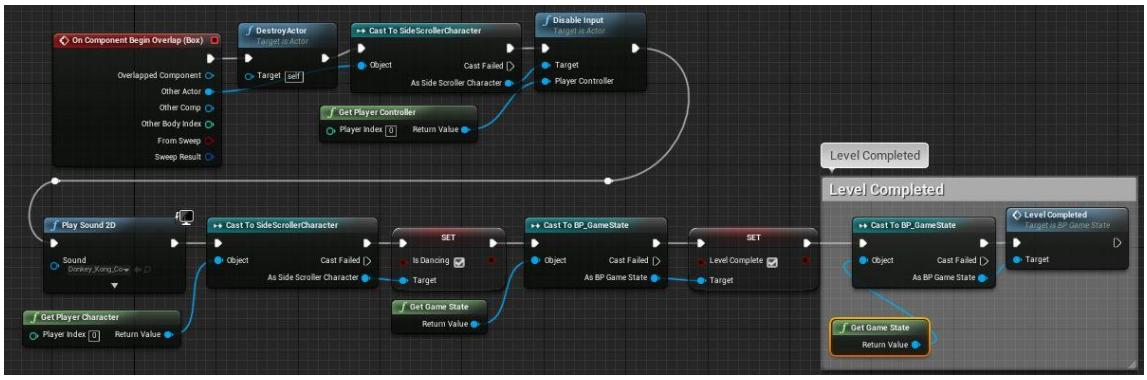

Figura 119 Captura del blueprint de la lógica de la moneda final de nivel parte 2

En el evento **On Component Overlap** de la caja de colisión llamamos al nodo **Destroy Actor** para eliminar la moneda de la escena. Acto seguido, hacemos un cast para obtener el personaje y llamamos al nodo **Disable Input** para poder deshabilitar el control del personaje al jugador. Una vez el jugador ya no tiene el control del personaje, llamamos al nodo **Play Sound 2D** para reproducir el sonido de victoria y modificamos el valor de la variable **isDancing** (variable utilizada para el control de estados en las animaciones) del jugador a cierto. A continuación, hacemos un cast al Game State para poder modificar la variable de control **Level Complete** a cierto conforme hemos completado el nivel, y llamamos al evento **Level Complete** creado en el blueprint GameState (se explicará su funcionamiento en la implementación del blueprint GameState).

7.7.2.4 Gancho

Los ganchos se encuentran repartidos por el espacio del nivel y a diferentes alturas. La lógica de su mecánica consiste en agarrar al personaje bajo suyo, una vez éste colisiona con la parte inferior del gancho.

Figura 120 Captura del viewport del blueprint del gancho

Para lograr la interacción con el personaje, el gancho cuenta con 2 componentes necesarios para poder llevar a cabo esta mecánica. Justo debajo del mesh del gancho hay ubicada una esfera de colisión para detectar cuando el personaje hace contacto con ella. El segundo componente se trata de un attach point que se encuentra ubicado dentro del modelo en la parte inferior del gancho. Su función consiste únicamente en proporcionar su ubicación, que se utilizará en la lógica del blueprint que se explica a continuación (ver Figuras 121 y 122).

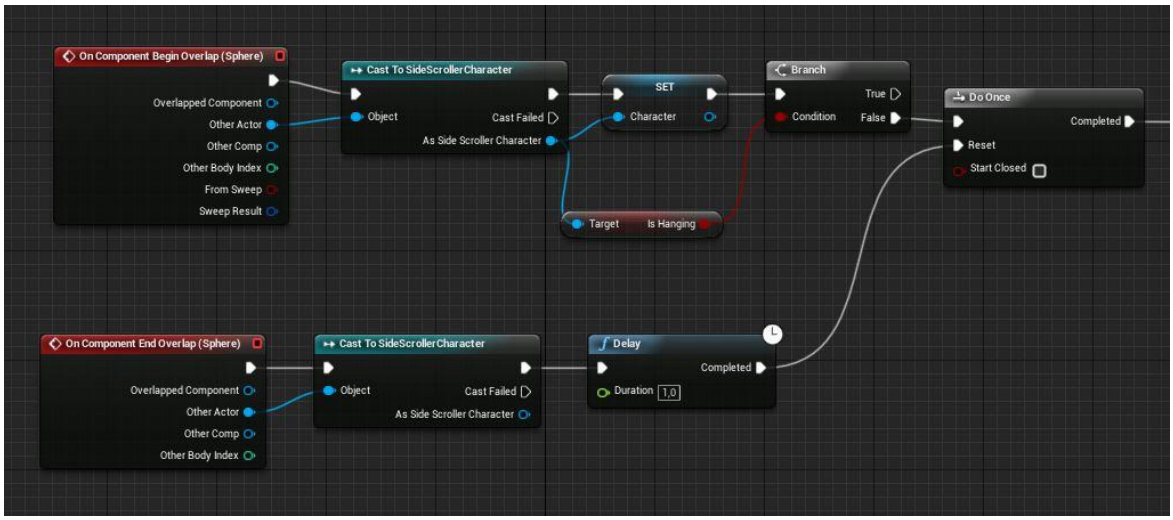

Figura 121 Captura del blueprint de la lógica del gancho parte 1

La lógica del gancho utiliza los eventos **On Component Begin Overlap** y **On Component End Overlap** de la esfera de colisión.

En el primer evento que es ejecutado cuando el personaje entra en contacto con la esfera de colisión, hacemos un cast del personaje y obtenemos tanto el personaje como su variable de estado **IsHanging**, que nos indica si el personaje se encuentra colgado o no. A continuación, creamos un **Branch** que tenga como condición el estado de la variable **IsHanging**. Si la condición es falsa (el personaje no se encuentra agarrado), llamamos al nodo **Do Once** que solamente disparará una vez su pulso para que siga la ejecución del blueprint. La única forma para que el

nodo **Do Once** permita disparar el flujo de nuevo, es que este nodo reciba un flujo por su pin **Reset.**

En el evento **On Component End Overlap** que se ejecuta cuando se termina de traslapar la esfera de colisión con el personaje. En este caso hacemos otro cast del personaje y aplicamos un delay de 1 segundo conectando su pin en el pin de **Reset** del nodo **Do Once**. El delay aplicado sirve para que, cuando el personaje se suelte del gancho, exista una franja de tiempo que permita que el jugador salga de la zona de colisión con la esfera antes de reiniciar el nodo **Do Once**, y permita volver a colgarse del gancho. Sin este delay, el jugador no podría descolgarse nunca.

Ambos eventos dirigen su pulso al nodo **Do Once**.

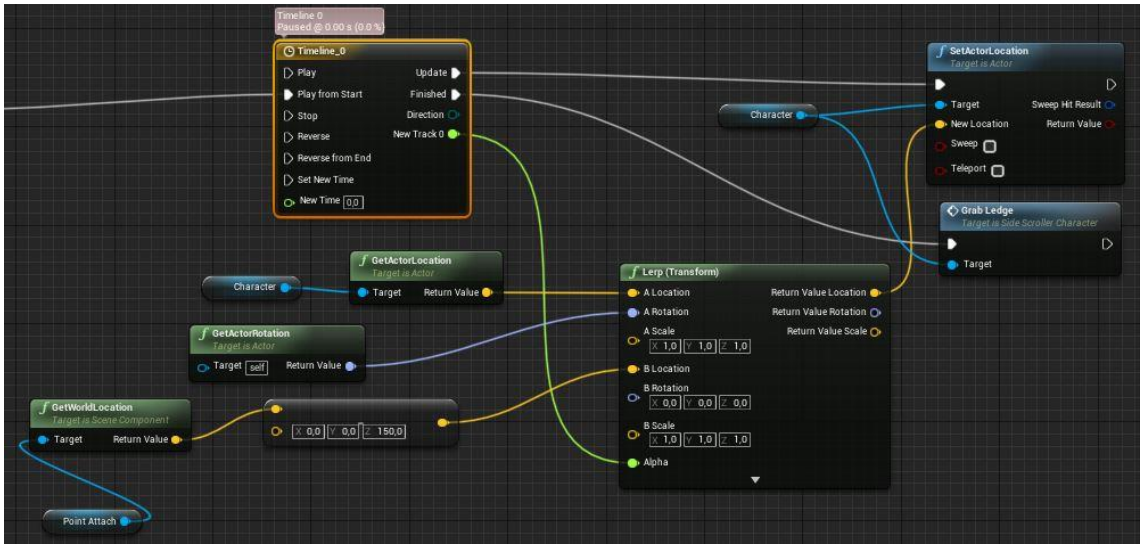

Figura 122 Captura del blueprint de la lógica del gancho parte 2

Después del nodo **Do Once** creamos un **Timeline** que nos permitirá diseñar y reproducir rápidamente una animación simple basada en el tiempo, en función de los eventos del juego. A ésta le conectamos el flujo en el pin de **Play from Start** para que reproduzca la animación que crearemos desde el principio cada vez que el **Do Once** se ejecute. La animación la crearemos utilizando el nodo **Lerp** que interpolará la ubicación del personaje con la del componente attach point del gancho. El resultado de la ubicación la utilizamos en el nodo que llamamos después del **Update** del **Timeline**, creando un nodo **SetActorLocation** para cambiar la ubicación del personaje a la ubicación resultante del nodo **Lerp**. En el pin **Finished**, llamamos al evento personalizado **GrabLedge**, proveniente del blueprint del jugador y que realiza la acción de agarrarse.

Figura 123 Captura del personaje agarrado en la ubicación final de la interacción

La animación realizada trata de suavizar el cambio de ubicación del personaje al desplazarlo desde el punto donde colisiona con la esfera de colisión hasta el componente point attach, ubicado en el punto donde debe quedar agarrado.

7.7.2.5 Trampolín

El trampolín es un objeto que podemos encontrar sobre el terreno del juego. Su funcionalidad consiste en que, cuando el jugador salta sobre la caja de colisión del trampolín, esté aplica un lanzamiento del personaje hacia arriba (ver Figura 124).

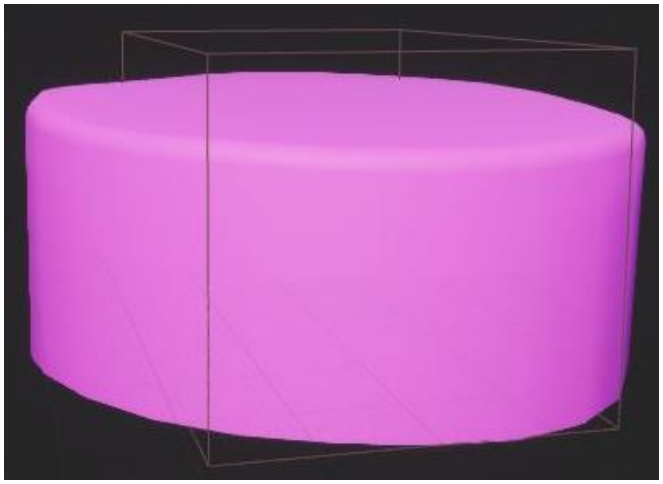

Figura 124 Captura del viewport del blueprint del trampolín

En cuanto a la implementación de la lógica, se basa en evento **On Component Begin Overlap** de la caja de colisión (ver Figura 125).

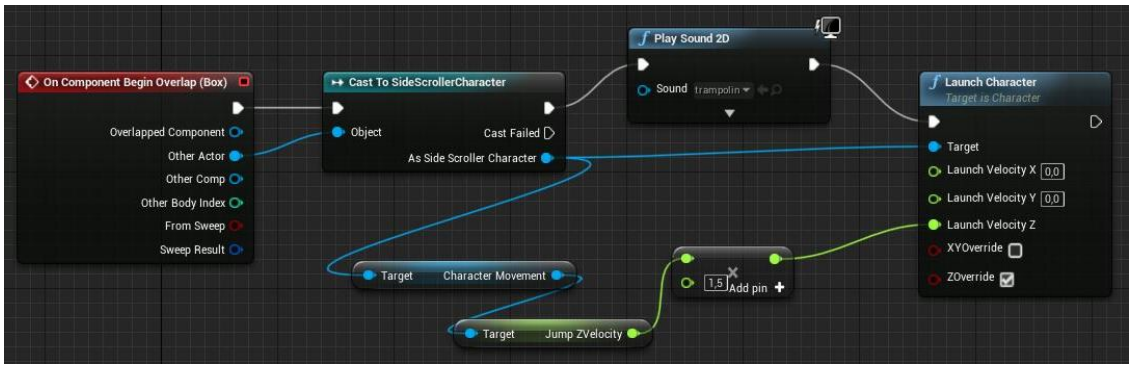

Figura 125 Captura del blueprint de la lógica del trampolín

En el evento **On Component Begin Overlap** hacemos un cast del personaje para obtener su velocidad de salto en el eje Z. Seguido, llamamos al nodo **Play Sound 2D** para reproducir el efecto de sonido del rebote del trampolín y llamamos al nodo **Launch Character** para aplicar rebote del personaje pasando en el pin de **Launch Velocity Z** la velocidad de salto del personaje en ese eje, multiplicada por 1,5.

7.7.2.6 Lanzador

El lanzador tiene un funcionamiento casi identico al del trampolín, con la diferencia que en este objeto la interacción para recibir el impulso no tiene por qué ser sobre él sino que también podemos ser impulsados colisionando con el sobre su parte inferior. Esto es debido a que el lanzador no se encuentra ubicado en el terreno, sinó en el aire.

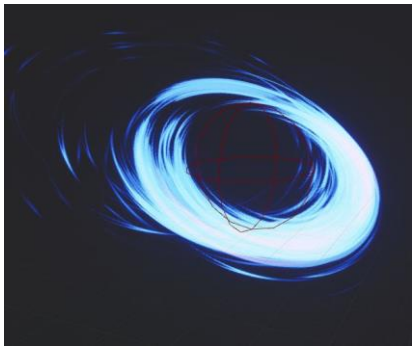

Figura 126 Captura del viewport del blueprint del lanzador

Como se puede ver en la Figura 126, el lanzador cuenta con una esfera de colisión, con la que el jugador hará contacto para activar el evento correspondiente (ver Figura 127).

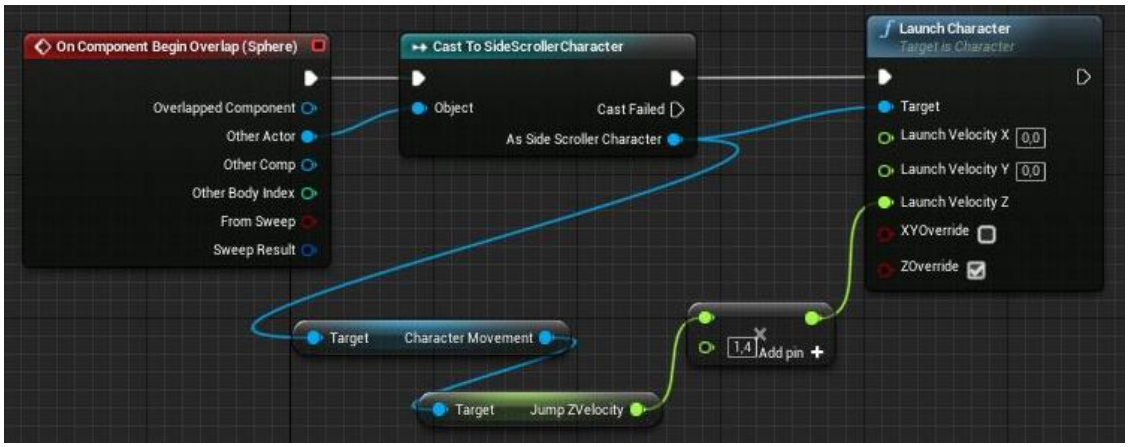

Figura 127 Captura del blueprint de la lógica del lanzador

La implementación de la lógica es idéntica a la explicada en el trampolín, con la diferencia que, en vez de multiplicar su velocidad de salto en 1,5 unidades, lo hacemos por 1,4.

7.7.2.6.1 Lanzador con trigger

Existe una variación del lanzador base que se utiliza en el nivel del juego. Esta variación se diferencia por la implementación de un trigger que hace que el lanzador aparezca en el nivel cuando se cumplen ciertos requisitos (ver Figura 128).

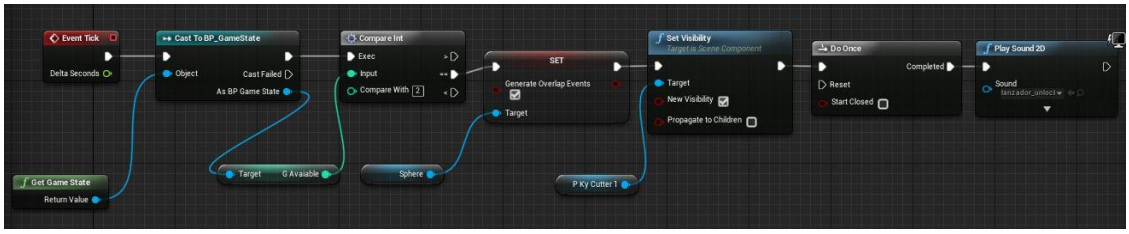

Figura 128 Captura del blueprint de la lógica añadida en el lanzador con trigger

Para la lógica añadida, utilizamos el evento **Event Tick** dónde haremos un cast al blueprint **GameState** para obtener la variable de control **G Avaiable**. Esta variable al inicio del nivel tiene un valor de 0, y sólo cambiará de valor si una o las dos **caracolas con trigger** (variación del enemigo caracola explicado en el apartado 7.7.1.2) existentes en el nivel muere, sumando una unidad por cada una de ellas.

Por ese motivo, comprobamos a continuación el valor de la variable con el número 2. Si el valor es igual, modificamos el parámetro **Generate Overlap Events** de la esfera de colisión del lanzador a cierto para que detecte la colisión con el personaje, y hacemos visible el efecto de partículas del lanzador llamando al nodo **Set Visibility**.

Por último, llamamos al nodo **Do Once** para ejecutar el **Play Sound 2D** con el sonido de aparición del lanzador una sola vez, ya que al estar en el flujo del **Event Tick**, se reproduciría en cada frame del tiempo de ejecución si no hiciéramos uso del **Do Once**.

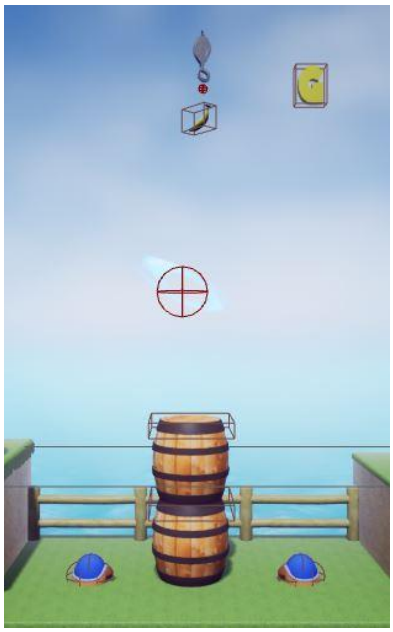

Figura 129 Captura dentro del Level Editor Viewport de Unreal donde se encuentra el lanzador con trigger

7.7.3 Otros

Además de los enemigos y los objetos, existen otros elementos en el juego que aportan otras mecánicas no menos importantes. Estos elementos se presentarán y explicarán a continuación en este apartado.

7.7.3.1 Checkpoint

Los checkpoints son los elementos encargados de actualizar los diferentes puntos de control por los que va alcanzando el jugador. Existen diferentes puntos de control repartidos a lo largo del nivel. Cuando el personaje colisiona con la caja de colisión del elemento, se actualiza la variable que almacena su último punto de control, reemplazándolo por el nuevo. Esta variable luego se utiliza para la reaparición del personaje cuando muere, permitiendo localizar la posición del último punto de control alcanzado.

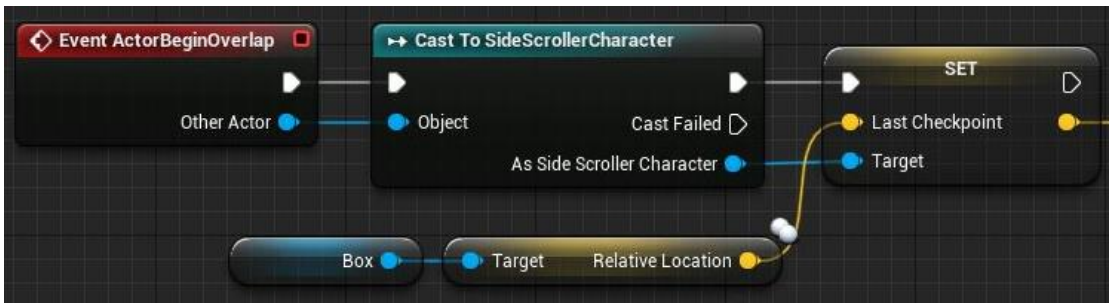

Figura 130 Captura del blueprint de la lógica del checkpoint

Para la implementación de la lógica del checkpoint, hacemos uso del evento **ActorBeginOverlap** que se ejecutará cuando detecte colisión con otro actor del juego. En el evento hacemos un cast

del personaje para obtener la variable **Last Checkpoint** y modificarla por la posición relativa de la caja de colisión del nuevo checkpoint.

7.7.3.2 Zona de muerte

A lo largo del escenario existen ciertas zonas que carecen de terreno sólido y dónde solo hay agua. Cuando el jugador cae en estos lugares, muere. Para lograr esta mecánica, se crea una zona de muerte a la altura del agujero, para poder detectar la colisión del personaje cuando cae. La zona de muerte está compuesta por una caja de colisión que permite detectar cuando el personaje entra en contacto con ella. De esta manera podemos implementar la lógica al recibir esta interacción que veremos en las Figuras 131 y 132.

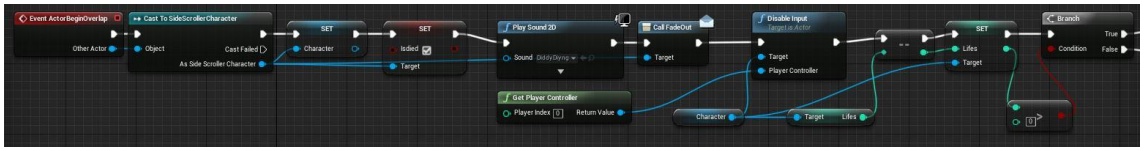

Figura 131 Captura del blueprint de la lógica de la zona de muerte parte 1

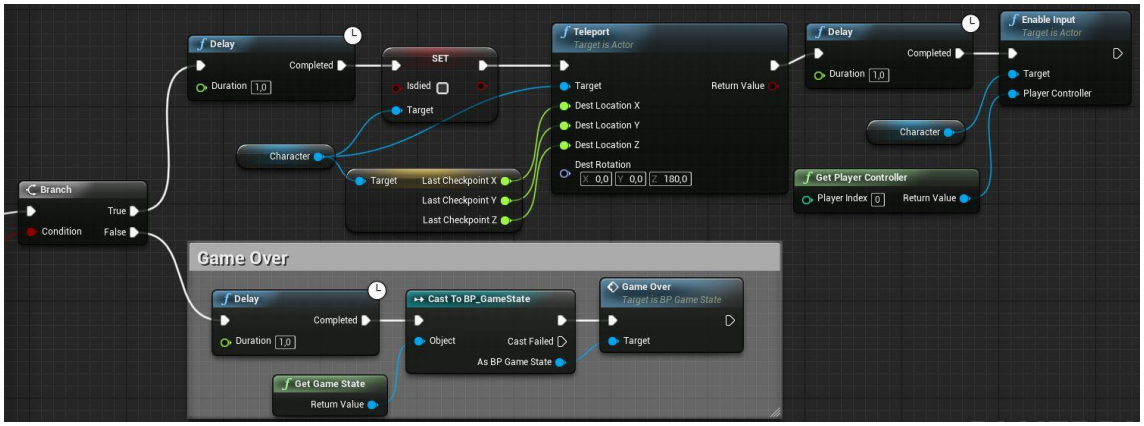

Figura 132 Captura del blueprint de la lógica de la zona de muerte parte 2

La lógica viene dada por el evento **ActorBeginOverlap** y con una lógica casi idéntica a la que se produce en la muerte del jugador contra los diferentes enemigos. Al producirse el evento cambiamos las variables de control de personaje referentes a su muerte y reproducimos el sonido de muerte, junto a la llamada del efecto de secuencia para la transición del personaje. Si al jugador le quedan vidas, lo teletransportamos al último punto de control y le devolvemos el control del personaje. En caso contrario hacemos un cast del blueprint GameState para llamar al evento **GameOver** que se encargará de la lógica del menú de final de partida.

El evento **GameOver** se ha implementado posteriormente en todas las blueprints dónde se puede producir la muerte del jugador.

7.8 Implementación del Level blueprint y Game State blueprint

Estos dos blueprints los hemos utilizado para implementar la creación e inicialización de ciertos elementos del nivel y para implementar la lógica de eventos que cambian el estado general del juego. En los siguientes subapartados se explicarán con profundidad ambos blueprints.

7.8.1 Level blueprint

Primero de todo, cuando se crea el nivel, se llama al evento **Begin Play**, el cual se encarga del proceso mostrado en las Figuras 133, 134 y 135.

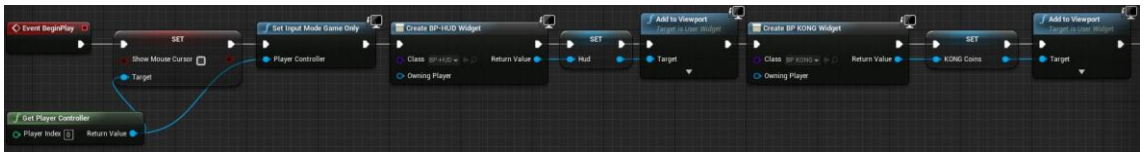

Figura 133 Lógica del Level blueprint del Level 1 parte 1

En el flujo del evento, hacemos un **Set** a la variable **Show Mouse Cursor** y modificamos su valor a falso para esconder el ícono del mouse que se ha utilizado en la pantalla del menú. Después, llamamos al nodo **Set Input Mode Game Only** para que el juego sólo reciba los inputs que están definidos para el personaje, pasándole el **Player Controler** del jugador. Creamos los widgets del HUD principal y del HUD de las monedas KONG añadiéndolos al viewport.

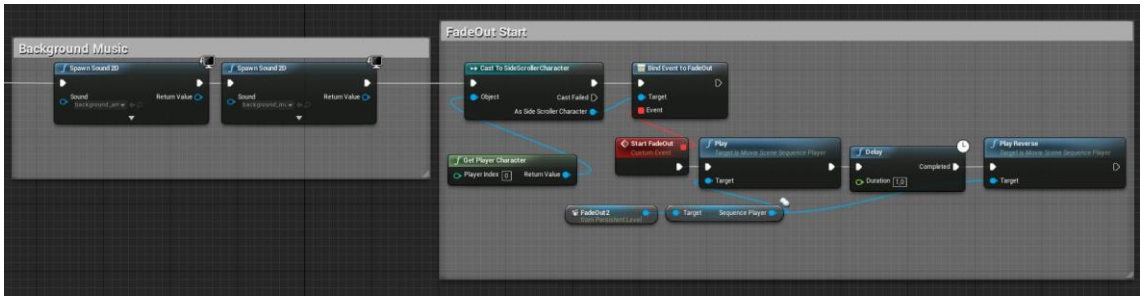

Figura 134 Lógica del Level blueprint del Level 1 parte 2

A continuación, añadimos la música de fondo del nivel y la de ambiente utilizando el nodo **Spawn Sound 2D** para cada una de las pistas de sonido.

Por último, pasamos a crear la lógica relacionada con la secuencia **FadeOut** que utilizamos en las transiciones de muerte del jugador. Para ello, hacemos un cast del jugador y llamamos al nodo **Bind Event to FadeOut**. Este **Bind** nos permitirá llamar a la función que crearemos a continuación desde cualquier blueprint del juego. En el pin del evento conectaremos la función que crearemos a continuación (**Start FadeOut**). En este evento personalizado llamamos al nodo **Play** y conectamos en su pin **Target** la secuencia **FadeOut2** creada como transición. A continuación, aplicamos un delay de 1 segundo para dar tiempo a que se ejecute el **FadeOut** y volvemos a reproducirlo llamado al nodo **Play Reverse**. De este modo conseguimos una transición con un efecto de oscurecimiento de la pantalla cuando el jugador muere, seguido del mismo efecto al revés, de pasar de tener la pantalla totalmente oscura a tenerla totalmente visible.

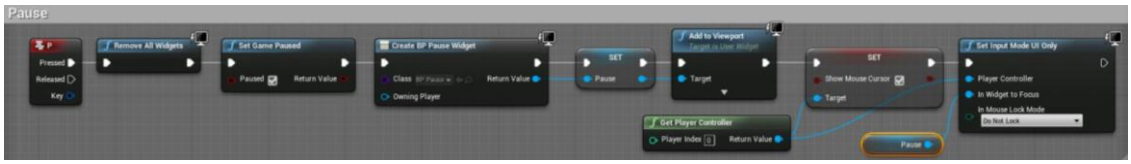

Figura 135 Lógica del Level blueprint del Level 1 parte 3

En el Level Blueprint del **Nivel 1** también se ha implementado el evento de **Pause**, el cual podemos utilizar durante el transcurso del nivel del juego.

Para implementar la lógica, llamamos el **input Key P**, la cual se ha asignado como tecla de pausa. En el evento, llamamos al nodo **Remove All Widgets** para eliminar todo el HUD antes de visualizar la interfaz de pausa. A continuación, llamamos al **Set Game Paused**, cambiando a cierto el valor de **Paused**, y llamamos al nodo **Create Widget** pasándole la clase del menú de pausa. Una vez creado, lo añadimos al viewport y modificamos la variable **Show Mouse Cursor** a cierto para que el jugador pueda utilizar el mouse para interactuar con los botones del menú de pausa. Por último, llamamos al nodo **Set Input Mode UI Only** para que el juego sólo haga caso a los inputs que se produzcan en el menú de pausa.

7.8.2 Game State blueprint

En cuanto al Game State, se ha utilizado esta blueprint para implementar ciertos eventos que dependen del estado del juego, además de crear ciertas variables de control en las que las diferentes clases pueden acceder a partir de un cast a este blueprint.

La lógica de los diferentes eventos implementados se explicará a continuación a partir de sus blueprints (ver Figuras 136, 137 y 138).

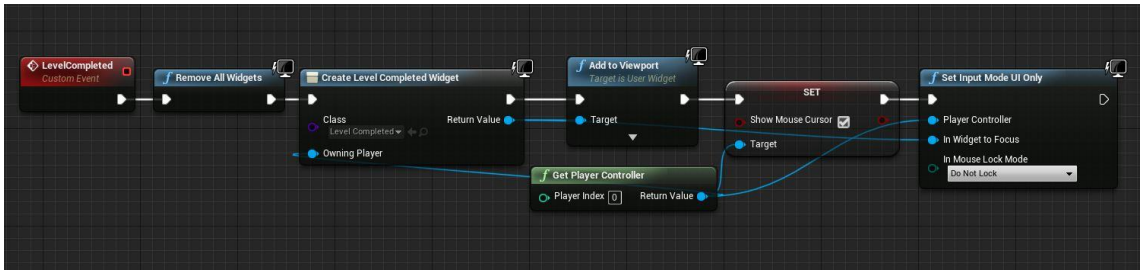

Figura 136 Lógica del Game State blueprint parte 1

En el evento **Level Completed** que hemos creado, implementaremos la lógica para finalizar el nivel y dar paso al menú del nivel completado. Para ello, empezamos el flujo llamando al nodo **Remove All Widgets** para eliminar todo el HUD de la pantalla. A continuación, creamos un nuevo widget pasando como clase el menú del nivel completado y lo añadimos al viewport. Para que el jugador pueda hacer uso del menú, hacemos un **Set** de la variable **Show Mouse Cursor** cambiando su valor a cierto y hacemos un **Set Input Mode UI Only**.

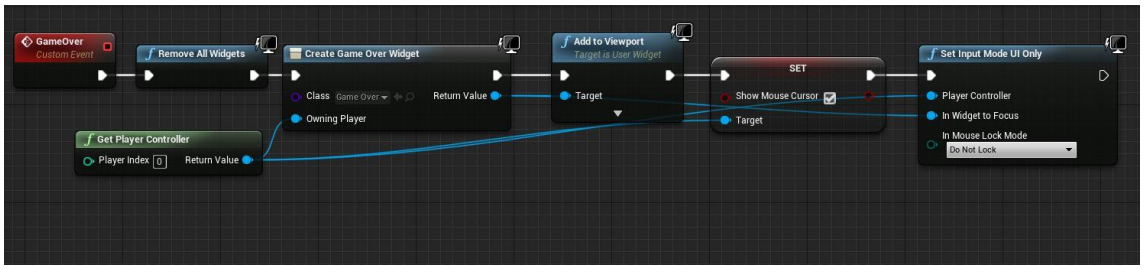

Figura 137 Lógica del Game State blueprint parte 2

En el evento personalizado **Game Over**, implementaremos la lógica para finalizar el nivel y dar paso al menú de **Game Over**. Para ello, eliminamos todos los widgets y creamos uno nuevo de la clase del menú **Game Over**, añadiéndolo al viewport. Una vez creado y añadido, hacemos un Set a cierto de la variable **Show Mouse Cursor** y llamamos al nodo **Set Input Mode UI Only**.

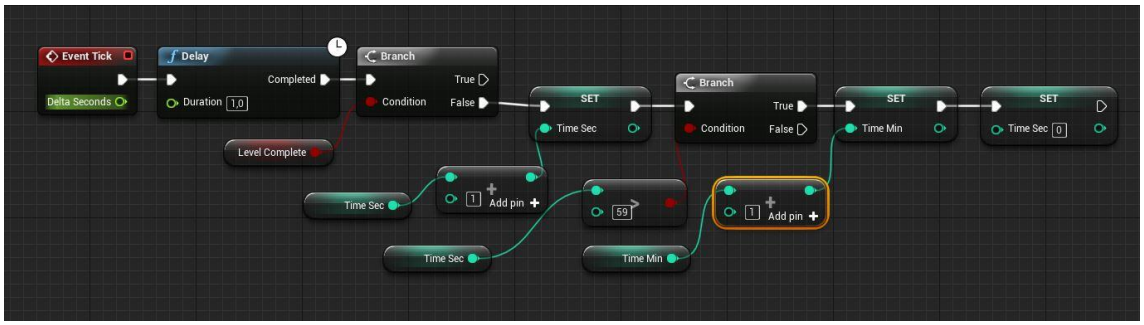

Figura 138 Lógica del Game State blueprint parte 3

En el **Event Tick**, crearemos un temporizador para contar el tiempo que tarda el jugador en completar el nivel. Para ello aplicaremos un delay de 1 segundo, seguido de un **Branch** que irá condicionado por la variable que indica si el nivel se ha completado. En caso falso, se sumará un segundo en la variable **Time Sec** que contiene los segundos totales. Para llevar la cuenta de los minutos totales, se creará un nuevo **Branch** con la condición que verifica si la variable **Time Sec** es mayor de 59. En el caso que sea cierto, sumaremos un minuto en la variable **Time Min** y modificaremos la variable **Time Sec** con valor 0.

7.9 Implementación del Animation Blueprint del personaje

Toda la lógica relacionada con el cambio de animación del personaje en función de las acciones que realiza o los cambios de estado que sufre, se producen gracias a la implementación realizada en el **Animation Blueprint** del personaje. Este blueprint contiene una lógica principal cómo las que hemos visto en elementos anteriores, y una máquina de estados dónde definimos qué animación se va a reproducir según una serie de condiciones, y qué transiciones entre animación son posibles. A continuación, se explicarán tanto la lógica principal como la máquina de estados implementada:

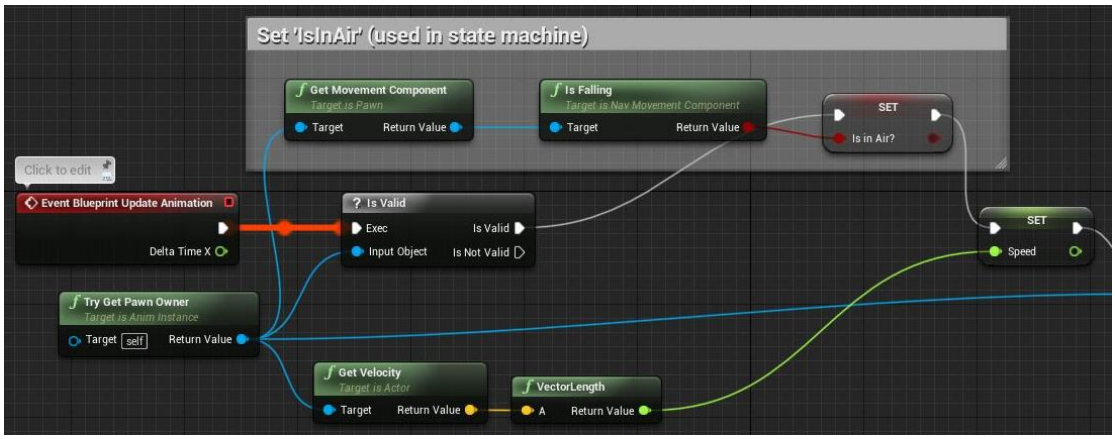

Figura 139 Lógica principal del Animation Blueprint 1

La lógica principal utiliza el evento **Blueprint Update Animation** para inicializar y actualizar el plano de animación. En el flujo de este evento, empezaremos llamando al nodo **is Valid** para asegurarnos que existe el movimiento del pawn que le pasamos al pin **Input Objet** mediante la función **Try Get Pawn Owner**. En caso que sea válido, hacemos un set a la variable creada **Is in Air** para comprobar si el pawn del personaje se encuentra en el aire. Esta variable recibe el valor que le proporciona la función **is Falling** del componente de movimiento del pawn, que devolverá cierto en caso que el jugador esté saltando o falso en caso contrario. A continuación, hacemos un set a la variable **Speed** creada, pasándole el valor de la velocidad del pawn del personaje.

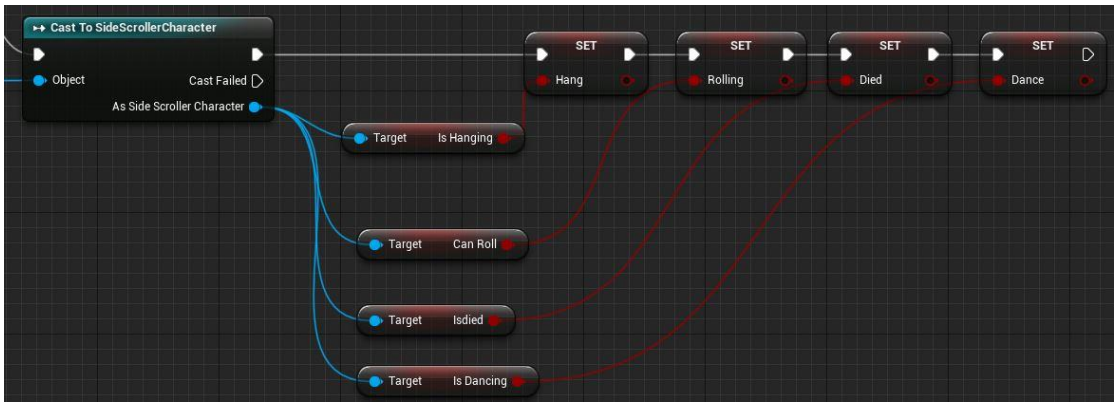

Figura 140 Lógica principal del Animation Blueprint 2

Seguimos el flujo haciendo un cast al personaje y obteniendo las variables de control de estado **Is Hanging**, **Can Roll**, **is died** y **is Dancing**. El valor de estas variables lo utilizamos para modificar el valor de las variables de control de animación **Hang**, **Rolling** (hace referencia a la acción de dash), **Died** y **Dance** creadas en el blueprint.

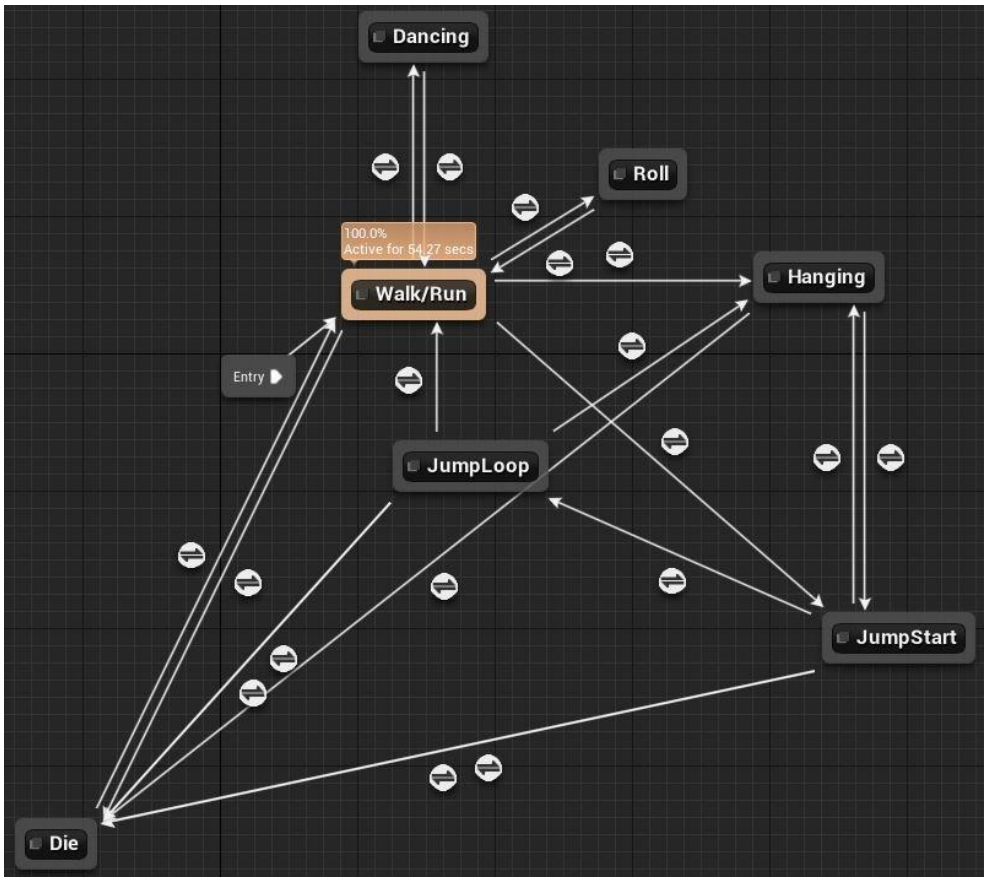

Figura 141 Máquina de estados del Animation Blueprint

En la máquina de estados, creamos todos los estados de animación posibles del personaje, añadiendo las reglas de transición posibles entre los diferentes estados. Todos los estados cuentan con una lógica de reproducción de la animación idéntica, excepto el estado **Walk/Run** (ver Figura 143).

Figura 142 Lógica del estado de animación Hanging

Figura 143 Lógica del estado de animación Walk/Run

En el caso del estado **Walk/Run,** existen tres animaciones posibles dependiendo de la velocidad del personaje. Para definir qué animación se reproduce según la velocidad, creamos un nodo

Blendspace Player que recibe la variable **Speed** y lo configuramos de la manera que se muestra en la Figura 144.

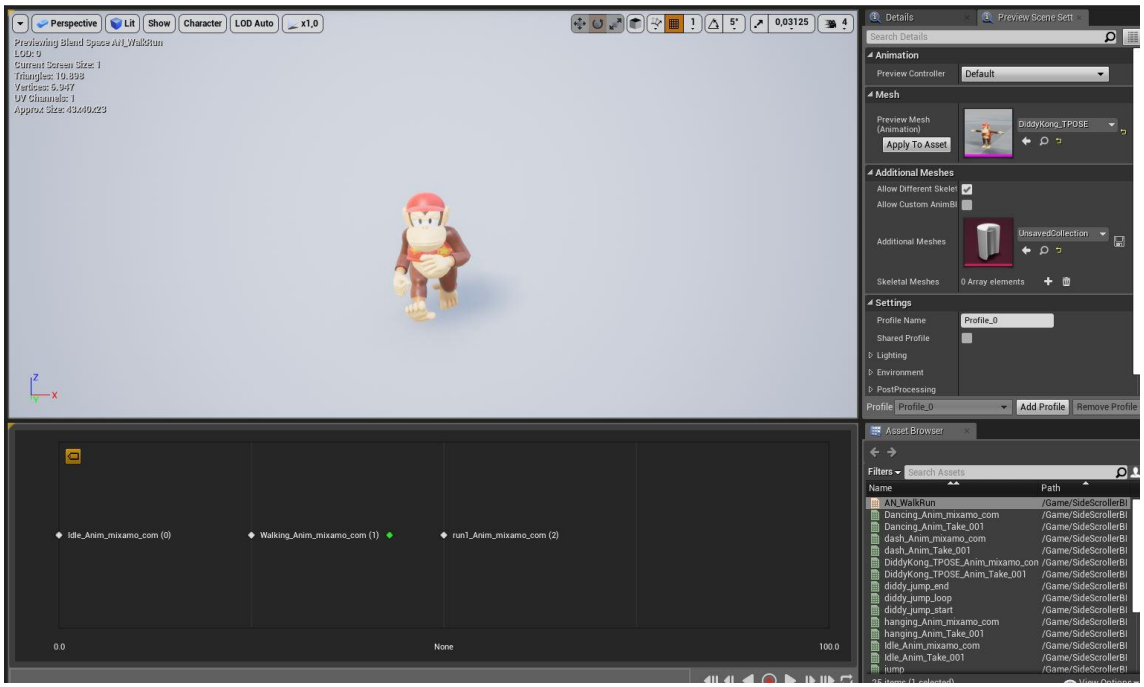

Figura 144 Viewport de la configuración del nodo Blendspace Player del editor de Unreal

Dentro del editor del **Blendspace Player** nos encontramos con una gráfica lineal definida por la variable **Speed**. En esta gráfica podemos configurar la transición de animaciones que se producen dependiendo de su valor, arrastrando la animación del pawn en ella y definiendo en qué valor la ubicaremos. De esta manera, ajustaremos la animación de Idle en el valor 0, la de **Walk** en el valor 25 y la de Run en el valor 50, consiguiendo esta transición de animaciones a partir de la velocidad que vaya alcanzando el jugador.

Volviendo al diagrama principal de la máquina de estados, nos fijamos en las diferentes flechas que marcan las posibles transiciones entre los diferentes estados. Cada transición contiene una regla que se encarga de cambiar de un estado a otro. Estas reglas por lo general siguen la misma lógica (ver Figuras 145 y 146).

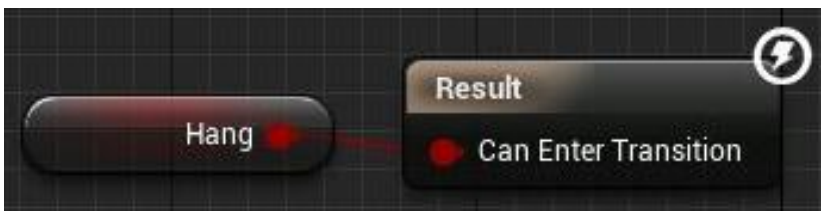

Figura 145 Lógica de la regla de transición del estado JumpStart a Hanging

En el caso de la regla de transición que vemos en la Figura 145, utilizamos la variable **Hang** (la cual va cambiando su valor en la lógica principal dependiendo de las variables de estado del personaje), cómo condición. Si su valor es cierto, se producirá el cambio de estado de la animación.

Figura 146 Lógica de la regla de transición del estado Hanging a JumpStart

Para el caso de la regla de transición inversa, la lógica es la misma que la anterior, pero con la variable **Hang** negada, de manera que si el valor es falso cambiará de estado.

Esta lógica es la misma para todas las demás reglas de transición, utilizando sus respectivas variables de condición definidas en la lógica principal, excepto en el caso de la regla que va desde el estado **JumpStart** a **JumpLoop** (ver Figura 147). En esta regla, la condición de cambio de estado viene dada por la relación de tiempo que le queda a la animación por terminar. Cuando este valor es menor que 0,1 se produce el cambio de estado.

Figura 147 Lógica de la regla de transición del estado JumpStart a JumpLoop

7.10 Implementación de la Interfaz de usuario

En este apartado se incluye tanto la implementación del HUD como la implementación de menús y el menú principal como otro nivel independiente.

Para la creación de interfaces de usuario en Unreal Engine, haremos uso de los Unreal Motion Graphics, cuyo núcleo son los **Widgets**. Estos Widgets contienen una serie de funciones predefinidas utilizadas en las interfaces (botones, sliders, cuadros de texto …) que nos facilitarán su implementación. Para hacer uso de ellos, existe un tipo de blueprint especial llamada **Widget Blueprint**, que nos permitirá crear nuestras interfaces en una capa visual haciendo uso de los Widgets además de poder implementar su lógica utilizando blueprints.

7.10.1 HUD

El HUD del juego está compuesta por dos Widget Blueprints: el principal y el secundario.

El HUD principal contiene los elementos relacionados con las vidas restantes y las bananas conseguidas por el jugador. En el lado izquierdo inferior de la Figura 148, podemos ver en la jerarquía los elementos que contiene el canvas. Tanto el contador de bananas como el de vidas cuentan con una imagen y un elemento de texto que informa del número total de cada elemento. Para poder cambiar el número de elementos en función del gameplay, es necesario crear una función para cada uno de ellos y hacer un **Bind** de la función con el elemento en cuestión para que la función actúe sobre él

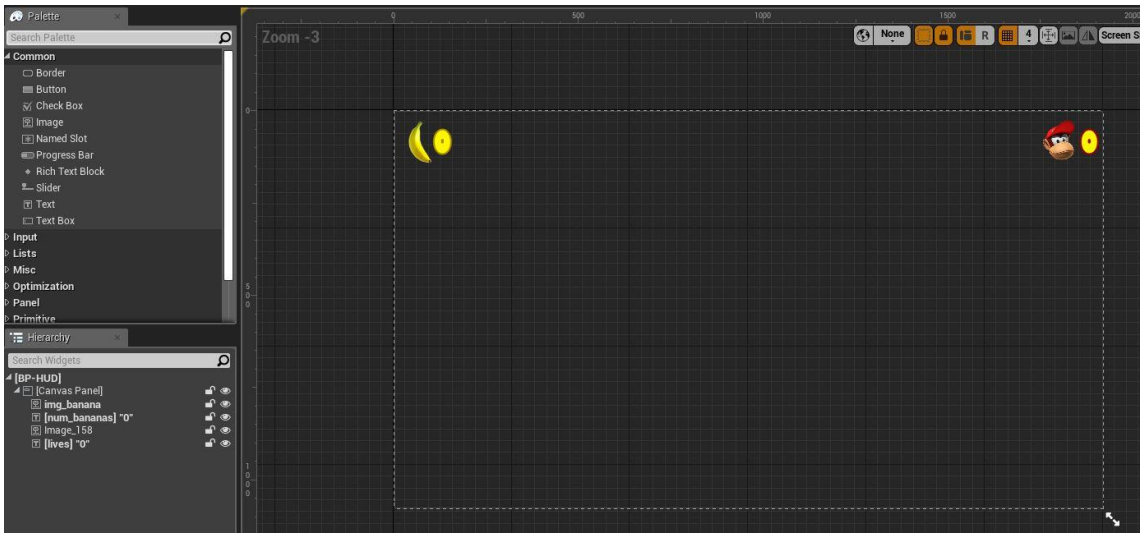

Figura 148 Pestaña Designer del Widget Blueprint del HUD principal

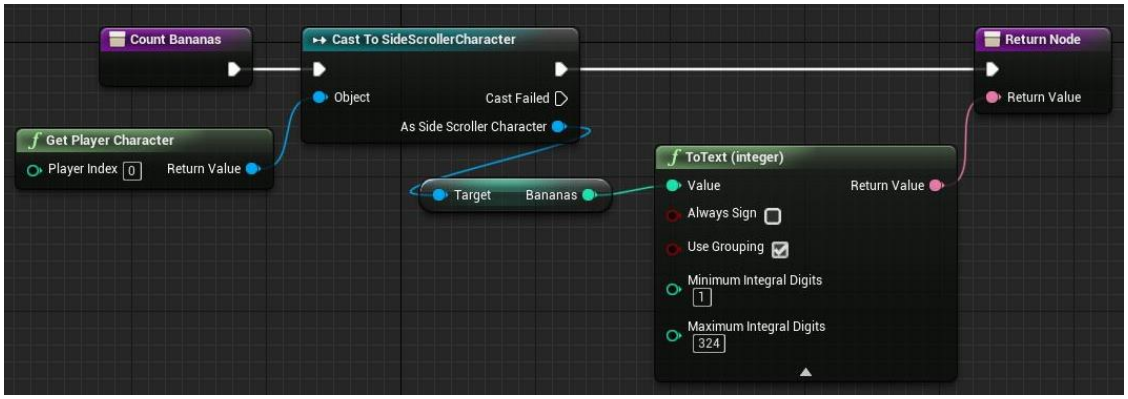

Figura 149 Función para el contador de bananas

La lógica de la función para el contador de bananas se basa en hacer un cast del personaje para obtener la variable de las bananas que lleva recolectadas el jugador. Una vez obtenemos la variable, la conectamos al pin de **Value** del nodo **ToText (integer)** para cambiar su formato de número entero a texto y poder devolverlo en el formato que utiliza el contador del canvas.

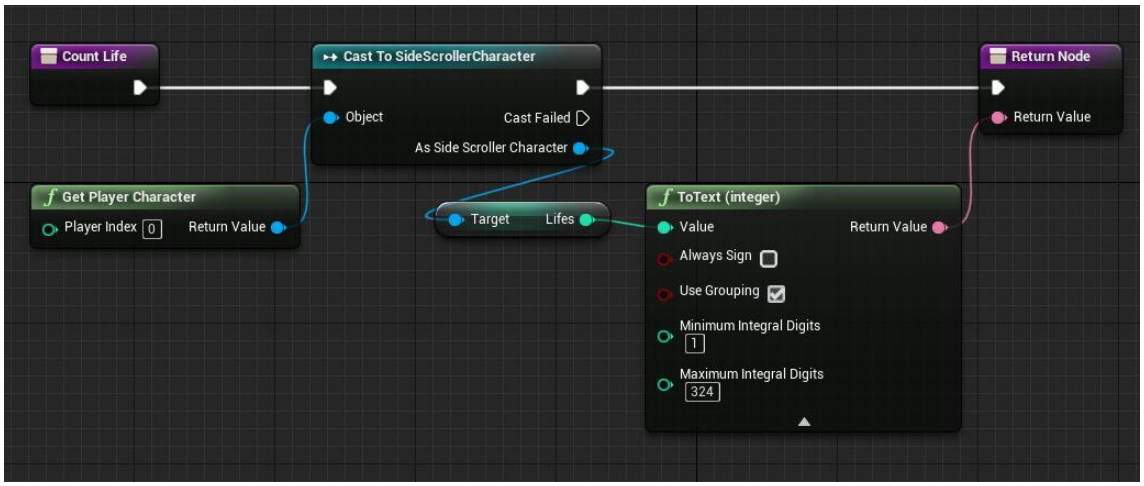

Figura 150 Función para el contador de vidas

En el caso del contador de vidas la lógica es la misma, cambiando la variable de las bananas obtenida del personaje por la de las vidas restantes.

En el caso del HUD secundario, tenemos los elementos que muestran las monedas KONG obtenidas en el transcurso del nivel.

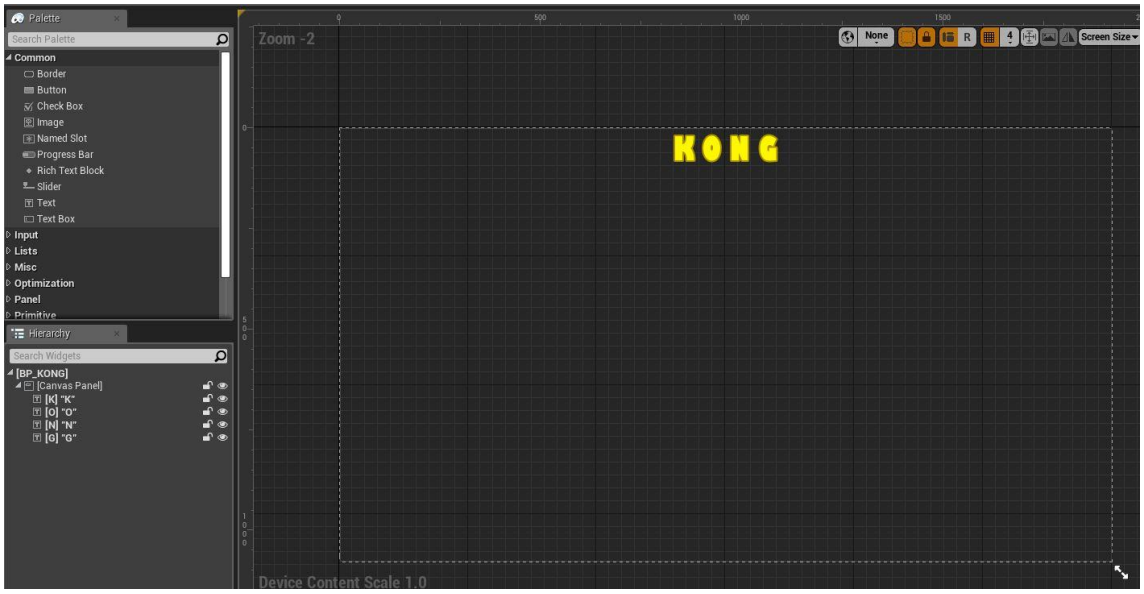

Figura 151 Pestaña Designer del Widget Blueprint del HUD secundario

Al empezar el nivel del juego, las letras KONG que se ven en la Figura 151 serán inicializadas en estado invisible. A medida que el jugador vaya recolectando cada una de ellas, irán cambiando su estado de invisible a visible.

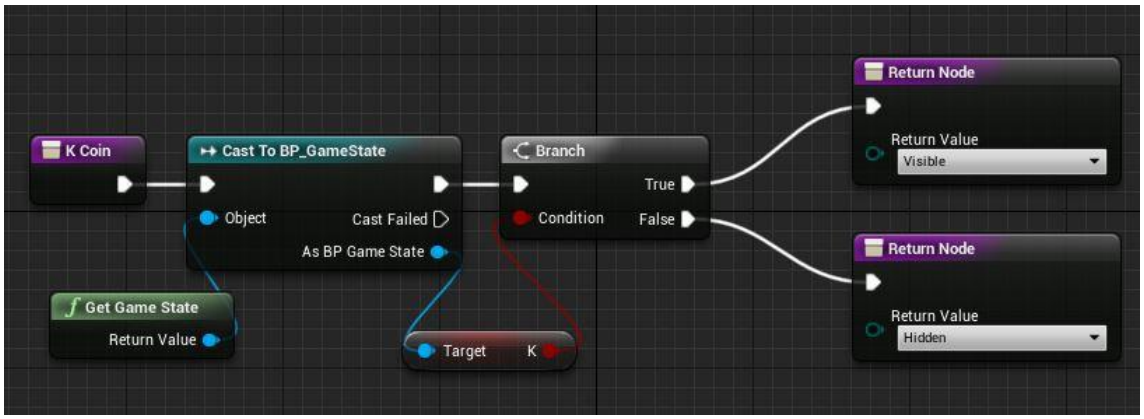

Figura 152 Función para la moneda K

Para cambiar a estado visible las diferentes monedas KONG, se ha implementado la lógica que vemos en la Figura 152, cambiando solamente el target por la moneda en cuestión. La lógica implementada se basa en hacer un cast al blueprint **GameState** para obtener la variable de control **K** que indica si la moneda ha sido recolectada. Esta variable sirve de condición en el **Branch** dónde haremos el elemento visible si es cierto, o invisible en caso contrario.

7.10.2 Menú Principal

El menú principal permite acceder al nivel del juego, aparte de contener el acceso a el menú de opciones o poder salir del juego. Este menú, además de contener la interfaz de usuario correspondiente, no se crea encima del nivel del juego, sino que forma parte de un nivel aparte llamado **MainMenu**.

Figura 153 Viewport del editor de Unreal del nivel MainMenu

Este nivel lo utilizamos exclusivamente para el menú, por ese motivo aprovechamos la lógica del **Level Blueprint** para fijar la cámara que hemos creado dentro del mapa como objetivo de vista

del menú. Además de esto, creamos el Widget del menú principal y lo añadimos al viewport, modificando el valor de la variable **Show Mouse Cursor** del **Player Controller** a cierto y cambiando el input a **Mode UI Only**.

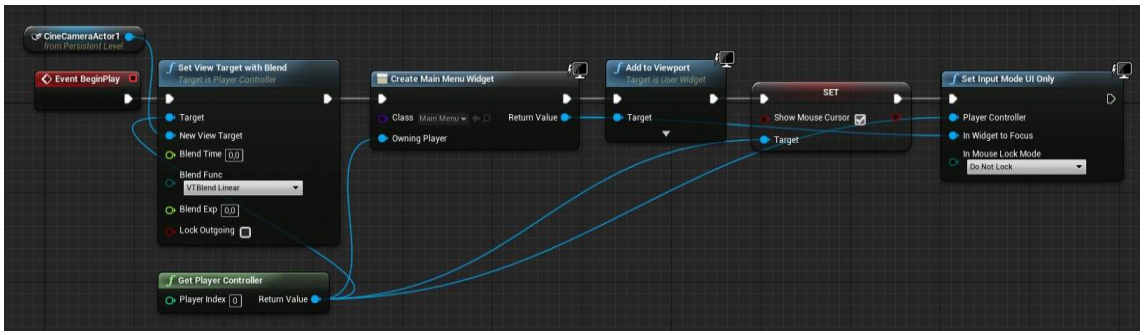

Figura 154 Lógica principal del Level Blueprint del MainMenu

Para la creación del Widget Blueprint, se ha utilizado el logo del juego y una serie de botones con las acciones de Play, Options y Quit.

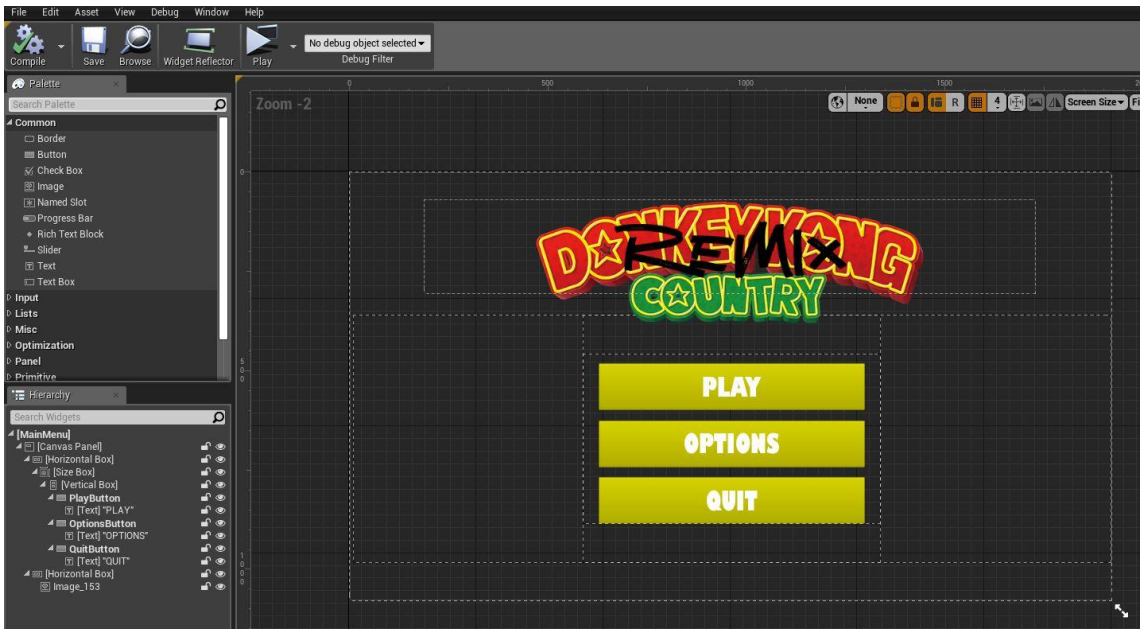

Figura 155 Pestaña Designer del Widget Blueprint del MainMenu

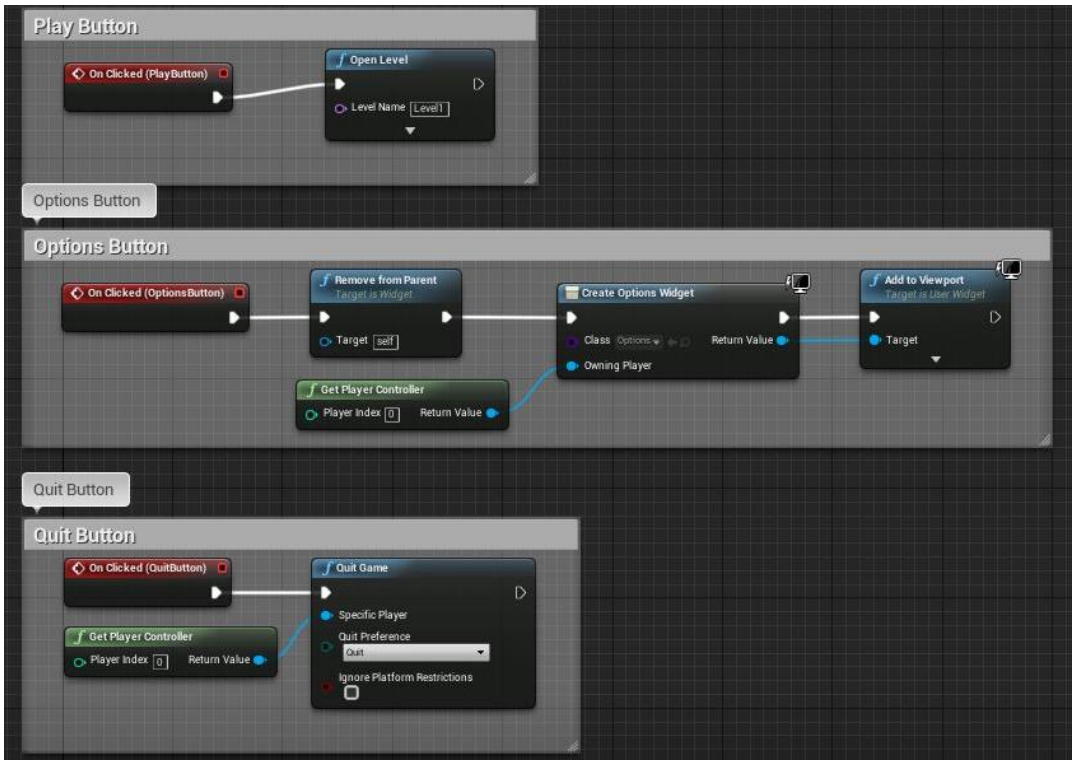

Figura 156 Lógica de los botones del Widget Blueprint del MainMenu

Para que los botones del menú desempeñen su función, debemos implementar la lógica que aparece en la Figura 156 y que se explica a continuación:

- **Botón Play**: En el evento **clic** del botón llamamos al nodo **Open Level**, indicando el nivel que queremos acceder. En este caso es el **Level1**.
- **Botón Options**: Cuando el evento clic de opciones se ejecuta, hacemos un **Remove From** Parent para eliminar el widget **MainMenu** y creamos el menú de opciones añadiéndolo al viewport.
- **Botón Quit**: En el caso del evento clic del Quit, llamaremos al nodo **Quit Game** para salir del juego.

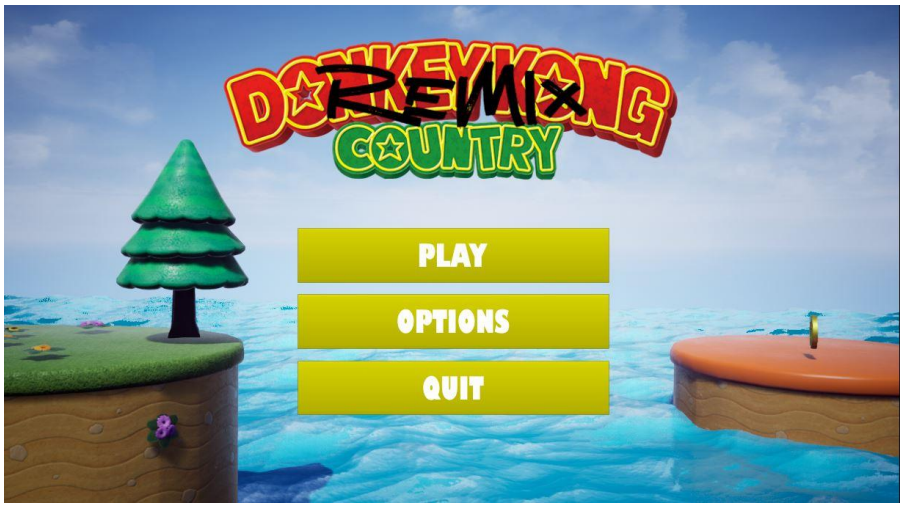

Figura 157 Menú principal del juego en ejecución

7.10.3 Menú de Opciones

El menú de opciones permite acceder al menú de controles del jugador, al de resolución y al sonido. Para poder volver al menú anterior, tenemos el botón Back (ver Figura 158).

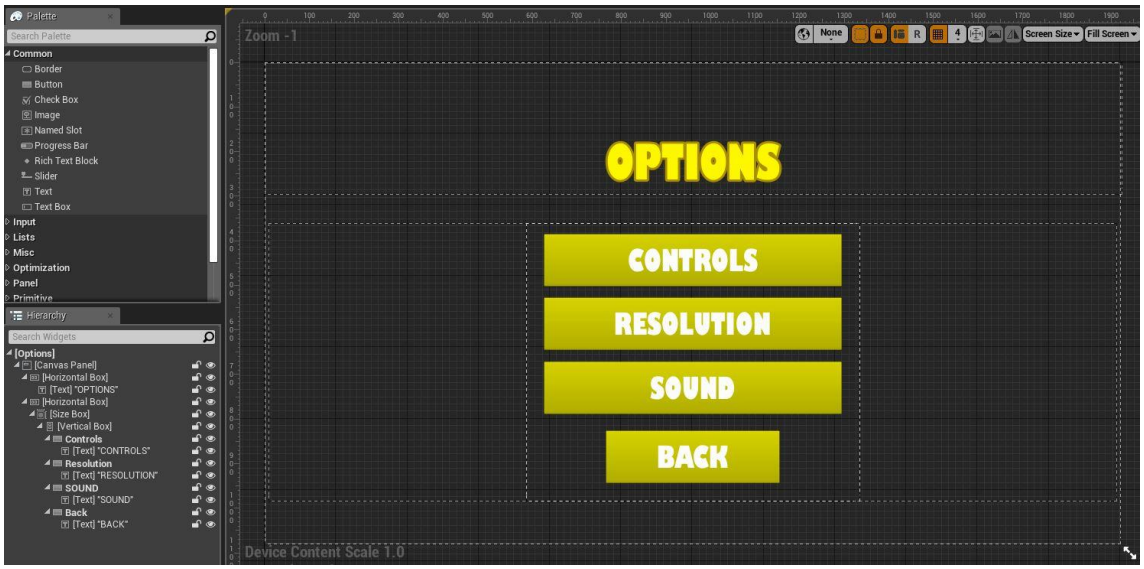

Figura 158 Pestaña Designer del Widget Blueprint del menú Options

La lógica que siguen los botones se implementa de la siguiente manera mostrada en la Figura 159.

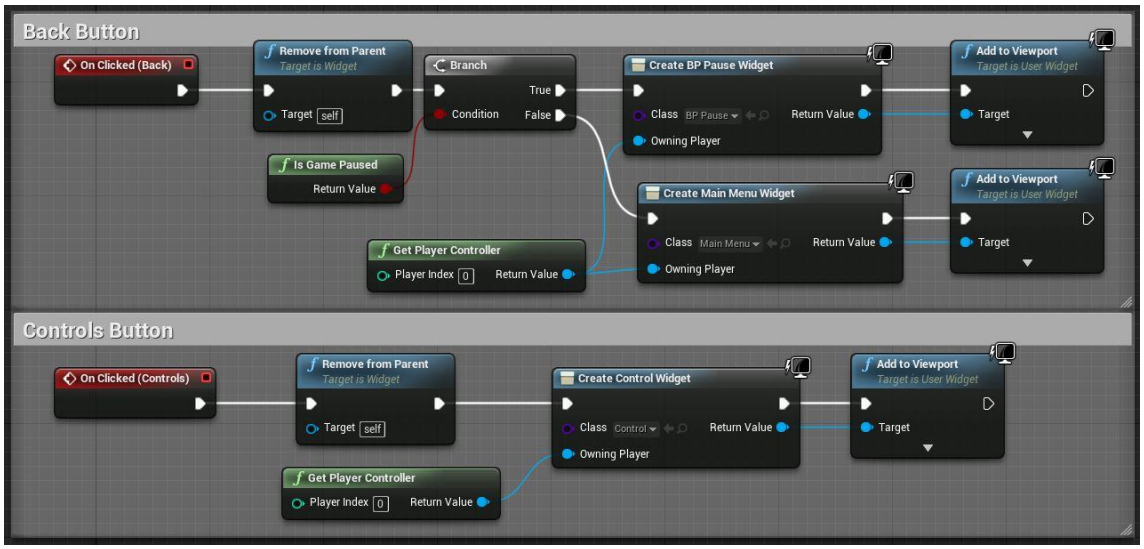

Figura 159 Lógica de los botones del Widget Blueprint de Options parte 1

- **Botón Back**: En el evento **clic** del botón **Back**, hacemos un **Remove From Parent** para eliminar el widget **Options** y creamos un **Branch** dando como condición si el juego está pausado. En caso cierto, creamos un widget con el menú de pausa añadiéndolo al viewport. En caso contrario creamos un widget con el **MainMenu**. Esta condición es debida a que, dependiendo de si el juego esta pausado o no, el menú anterior por el cual hemos accedido a las opciones puede ser distinto.

- **Botón Controls**: En el evento **clic** de este botón también eliminamos el widget en el que nos encontramos y creamos uno nuevo con la clase del menú **Controls**, añadiéndolo al viewport.

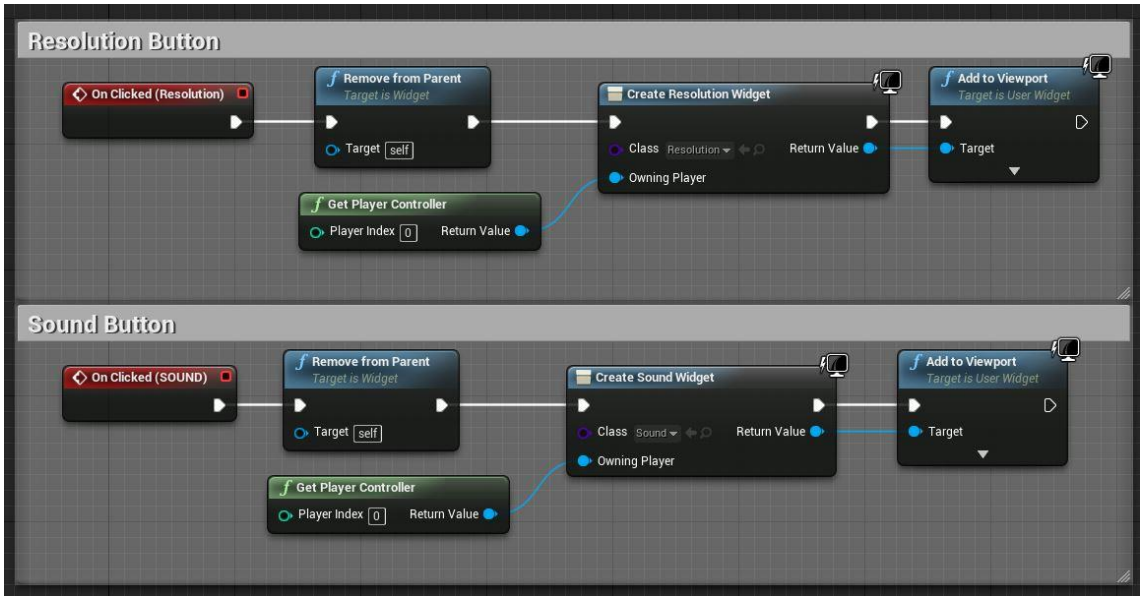

Figura 160 Lógica de los botones del Widget Blueprint de Options parte 2

- **Botón Resolución**: La lógica de este evento es la misma que la explicada anteriormente, eliminando el widget actual y creando en este caso el de **Resolutions** (ver Figura 160).
- **Botón Sound**: Con esta lógica ocurre lo mismo, eliminamos el widget actual y creamos el de **Sounds** (ver Figura 160).

7.10.3.1 Menú de Controles

El menú de controles se encarga de mostrar la información referente a los controles principales del juego. Este menú contiene la información descrita y un botón **Back**, para volver al menú anterior.

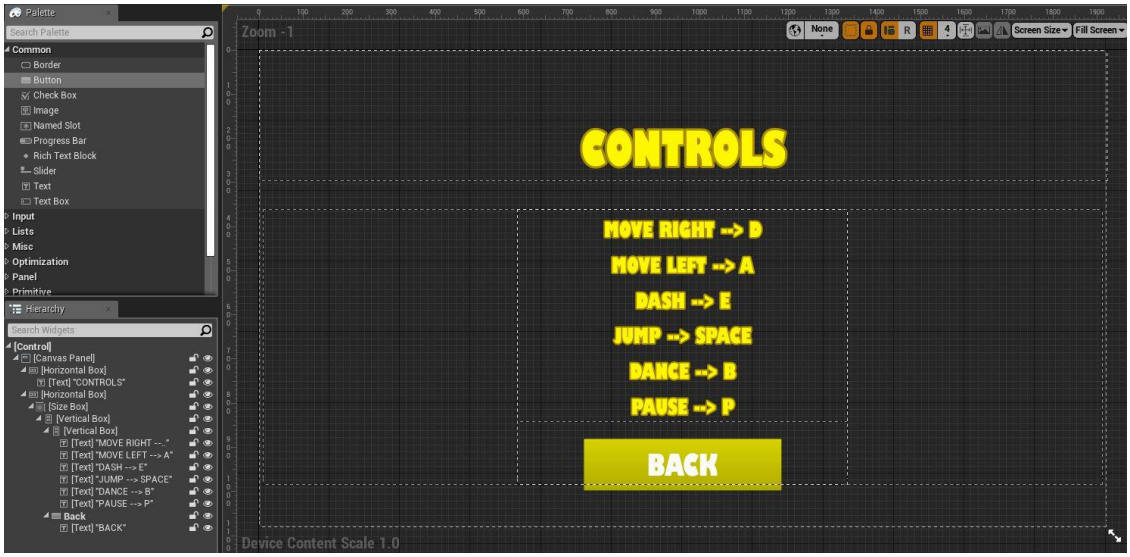

Figura 161 Pestaña Designer del Widget Blueprint del menú Controls

7.10.3.2 Menú de Resolución

El menú **Resolution** permite seleccionar y ajustar la resolución del juego, pudiendo escoger entre las resoluciones 640x480, 1280x1080, 1440x900 y 1920x1080. También contiene el botón **Back** para volver al menú anterior.

Figura 162 Pestaña Designer del Widget Blueprint del menú Resolution

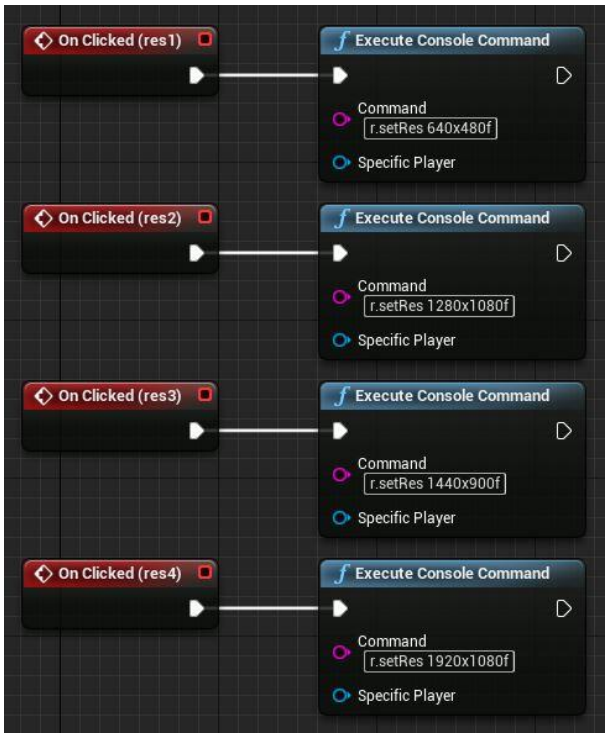

Figura 163 Lógica de los botones del Widget Blueprint de Resolution

La lógica implementada en los botones de las diferentes resoluciones viene dada por los eventos clic de cada una de ellas. Cada evento llama a un nodo **Execute Console Command**, dónde se ejecuta un comando específico para cambiar la resolución de la pantalla.

7.10.3.3 Menú de Sonido

El menú de sonido cuenta con dos sliders que permiten regular el sonido de la música del juego, y el sonido de los efectos especiales. Como en los menús anteriores, también tenemos el botón Back para retroceder al menú anterior.

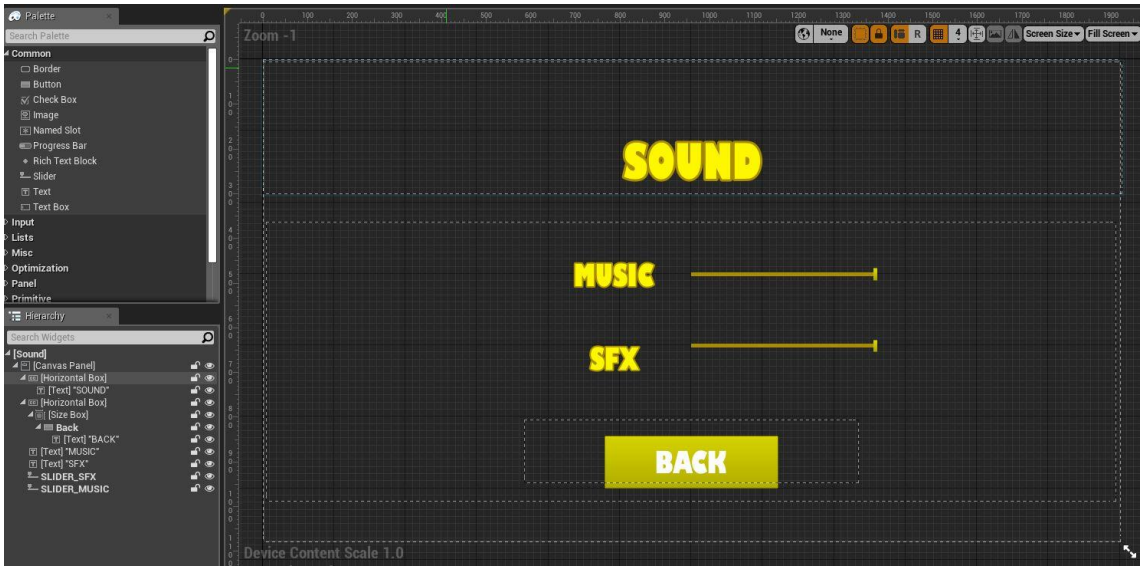

Figura 164 Pestaña Designer del Widget Blueprint del menú Sound

Para poder regular tanto el sonido de la musica como los efectos especiales, se han tenido que crear dos **Sound Class**, uno para las pistas de música y otro para las de efectos de sonido. Además, se ha cambiado el **Sound Class** determinado de todas las pistas de audio, por el **Sound Class** indicado para cada tipo de pista. Al tener dos de ellos, es posible regular el volumen de las pistas de sonido de forma independiente.

Figura 165 Lógica de las sliders del Widget Blueprint de Sound

Para la implementación de la lógica de las sliders, se utiliza el evento **On Value Changed** en cada uno de ellos. En este evento llamamos al nodo **Set Sound Mix Class Override**, donde conectamos en el pin de **Volume** el valor del slider y seleccionamos el Sound Class correspondiente en el pin para cada slider. Después de esto, hacemos un cast al **GameState** para modificar el valor de las variables de control **Music Vol** y **Sfx Vol**, pasándoles el valor de cada slider. Estas variables se utilizan para poder guardar el valor de los sliders una vez se sale del menú y se elimina el widget, de manera que, si se vuelve a acceder, podemos reestablecerlos a su último valor.

7.10.4 Menús de Nivel Completado

Los menús de nivel completado están formados por los Widget Blueprint **LevelCompleted** y **LevelCompleted2**. Cuando el jugador finaliza el nivel, aparece el primer menú dónde se muestran diferentes estadísticas que el jugador ha obtenido en la partida. Estas estadísticas están formadas por el tiempo total tardado en completar el nivel, las bananas totales recolectadas y las monedas KONG obtenidas. Para poder pasar al siguiente menú, éste cuenta con un botón **Continue**.

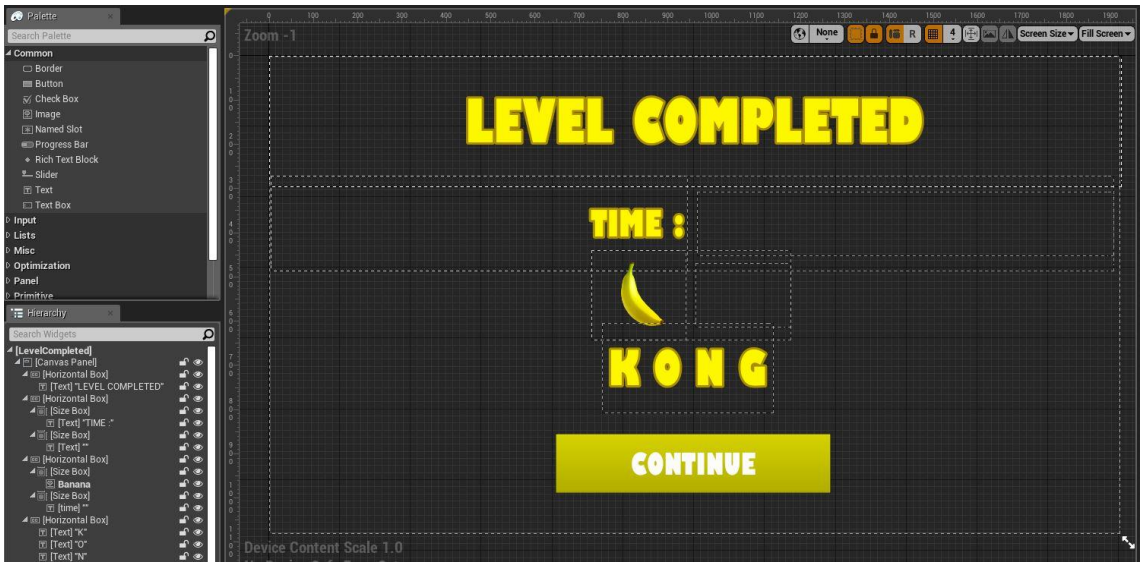

Figura 166 Pestaña Designer del Widget Blueprint del menú LevelCompleted

Para poder mostrar tanto el tiempo total, las bananas totales y las monedas KONG obtenidas, hay que hacer un **Bind** en cada uno de estos elementos del canvas, junto a una función que implemente su lógica.

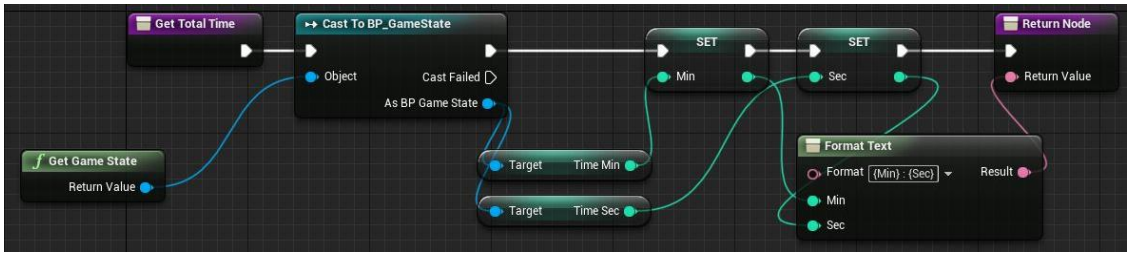

Figura 167 Función para el tiempo total

Para implementar la función del tiempo total, debemos empezar haciendo un cast al **GameState** para obtener las variables **Time Min** y **Time Sec**. Una vez obtenidas, hacemos un set a las variables propias del blueprint **Min** y **Sec** pasandoles los valores de las variables del **GameState**. Por último, cambiamos el formato de las variables con el nodo **Format Text** y devolvemos el valor final.

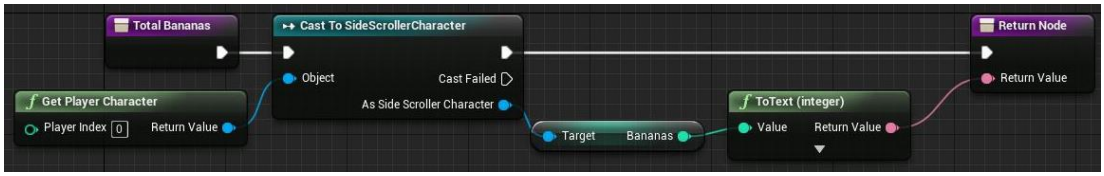

Figura 168 Función para las bananas totales

Para la función de las bananas totales hacemos un cast al personaje para obtener la variable **Bananas** (contiene el valor de las bananas totales recolectadas por el jugador), y llamamos al nodo **ToText (integer)** para cambiar el tipo de la variable a texto antes de devolverla.

En el caso de las funciones para las monedas KONG, utilizan la misma lógica que se ha explicado en el caso del HUD.

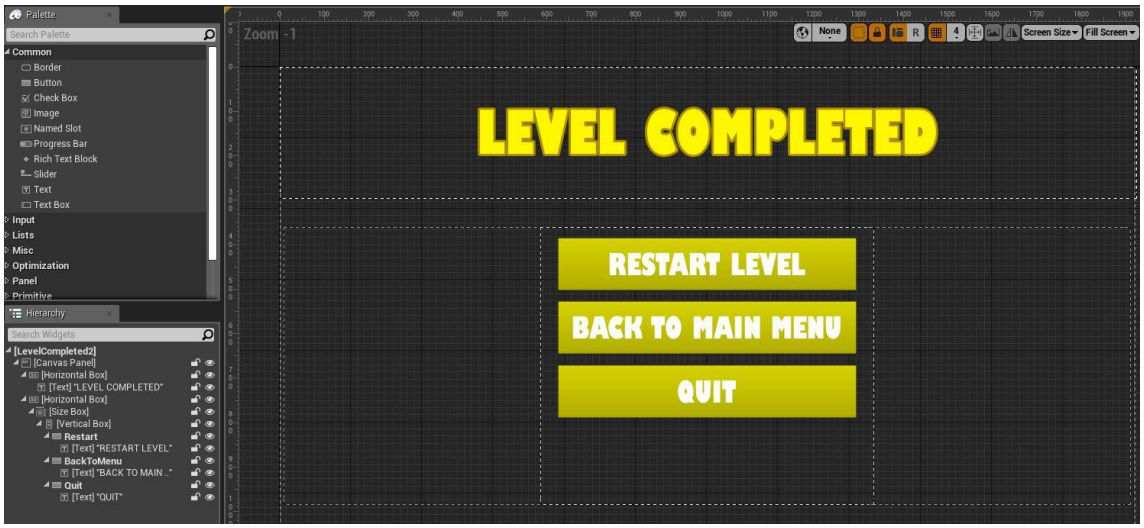

Figura 169 Pestaña Designer del Widget Blueprint del menú LevelCompleted2

Una vez hacemos clic en el botón **Continue** del primer menú, de elimina el widget y se crea uno nuevo con el segundo menú. Este menú contiene los botones para poder reiniciar el nivel, volver al menú principal o salir del juego.

La lógica implementada en estos botones es la misma vista anteriormente:

- El botón **Restart Level** hace un **Open Level** del mismo nivel.
- El botón **Back to Main Menu** hace un **Open Level** del nivel **MainMenu**.
- El botón **Quit** sale del juego.

7.10.5 Menú de Final de Partida

Para finalizar con los menús del juego, tenemos el menú de final de partida. El jugador accederá a este menú una vez muera y no le queden vidas para reaparecer. Este menú cuenta con los mismos botones que el menú de nivel completado, permitiendo reiniciar el nivel, volver al menú principal o salir del juego.

Figura 170 Pestaña Designer del Widget Blueprint del menú GameOver

8. Resultados

Para valorar el grado de cumplimiento en cuanto los resultados obtenidos, haremos uso de los objetivos que se marcaron en el inicio del proyecto.

El objetivo principal que se marcó en el proyecto fue la creación del prototipo de un remake del videojuego *Donkey Kong Country 2 (1995)* jugable, utilizando las tecnologías actuales

Para poder llevar a cabo esta tarea, se macaron una serie de objetivos específicos que ayudarían a cumplir el objetivo principal:

- **Realizar un análisis de diseño y requisitos del videojuego**: Antes de comenzar con la creación e implementación del videojuego, se hizo un estudio en cuanto los requisitos necesarios que se debían cumplir para poder realizar el juego. Estos requisitos venían dados tanto con la viabilidad del juego como en la disposición de las herramientas de trabajo necesarias para su desarrollo. Una vez vista la viabilidad del proyecto, se realizó un Game Document Design (apartado 6. Diseño del Videojuego).

- **Cubrir los aspectos de pipeline necesario para la importación de assets en el motor desde software externo a Unreal Engine 4**: Este objetivo hace referencia a todo el trabajo relacionado con la obtención y creación de modelos, animaciones, sprites, sonidos, etc. y posteriormente su importación al motor Unreal Engine para poder hacer uso de ello dentro del proyecto. Para el cumplimiento de este objetivo hubo ciertas dificultades en la obtención de todos los assets necesarios, ya que, por planificación de tiempo en el proyecto, no era viable crear los modelos, su esqueleto, y animarlos desde cero. Por ese motivo se estudiaron y utilizaron diferentes métodos, lo cual agilizó el trabajo explicado, tanto en la parte de diseño como implementación.

- **Diseño y creación de niveles utilizando las herramientas que ofrece el editor de Unreal**: Tanto el diseño como la creación de niveles viene explicado en detalle en los apartados de diseño e implementación. Se define previamente la referencia del diseño del nivel, y después se utilizan las herramientas del editor de Unreal junto a los assets importados para crear el escenario.

- **Implementar la lógica del juego utilizando el sistema de scripting visual Blueprints**: Este objetivo es entre los específicos, el que más importancia tiene y al que se le ha dedicado más tiempo y esfuerzo. Para implementar toda la lógica del juego, ha habido un estudio y aprendizaje del motor Unreal en paralelo al desarrollo e implementación del juego. Esto hace que tome mucha importancia, por el hecho que el conocimiento adquirido del motor durante el transcurso del proyecto, es incluso más valioso que el proyecto en sí. El grado de cumplimiento de este objetivo se ve reflejado en el apartado de implementación.

- **Diseño y creación del sistema de Interfaz de usuario del juego**: Tanto el diseño como la creación del sistema de Interfaz de Usuario se explica detalladamente en los apartados de diseño e implementación. A este objetivo se le ha dado bastante importancia, debido a que el diseño y creación de una buena UI es esencial para

cualquier tipo de videojuego. El jugador debe entender los diferentes menús y sentirse cómodo con la interfaz, ya que es la encargada de establecer la comunicación.

- **Implementación de sonidos y efectos visuales del juego**: La implementación de sonidos y efectos visuales del juego ha venido dada por dotar el juego con una música y sonido ambiental para el nivel jugable, efectos de sonido para las diferentes interacciones dadas en el nivel y en los menús, y el uso de efectos de partículas en situaciones como la desaparición al derrotar enemigos.

- **Realizar y analizar pruebas de testing para balancear dificultad, mecánicas, entendimiento, etc.**: Este objetivo se ha ido realizando durante toda la implementación del proyecto. Se han ido ajustando todos los aspectos relacionados con la dificultad y mecánicas dentro del gameplay del juego. De la misma manera, se ha diseñado tanto los menús como el HUD previamente, para asegurar un entendimiento de los mismos a la hora de que el jugador interactúe con estos elementos.

Además de todos los objetivos específicos que inicialmente se marcaron y han sido explicados, durante el desarrollo del proyecto han surgido muchos otros que se han ido llevando a cabo para poder cumplir con el objetivo principal.

9. Conclusiones personales

Gracias al desarrollo de este proyecto, he podido llevar a cabo el objetivo principal con el cual decidí cursar el grado de Diseño y Desarrollo de Videojuegos, desarrollar mi propio videojuego. En cursos anteriores ya había trabajado en el desarrollo de este tipo de proyectos, pero a mucha menor escala y con ciertas limitaciones a la hora de toma de decisiones.

El prototipo desarrollado llamado ***Donkey Kong Country Remix***, surge como consecuencia de un deseo personal que llevaba años rondando por mi cabeza. El juego clásico *Donkey Kong Country 2: Diddys Kong Quest* fue un título que marcó mi relación con los juegos de plataformas en la infancia, y viendo los pasos que la industria de los videojuegos estaba tomando con traer de vuelta títulos clásicos a la actualidad, me plantee por qué no traer de vuelta a uno de los pilares del género de plataformas.

Una de las principales dudas que tuve en escoger esta idea como proyecto de final de grado, fue la limitación de tiempo para su desarrollo. Este problema fue debido a que durante todo este año estuve en el programa de Movilidad Internacional para estudiantes, cursando mi último año de grado en una universidad de otro país. En un principio, este proyecto iba a ser llevado en la Universidad de Talca (Chile) y presentado en un doble tribunal de ambas universidades. Pero debido a todos los acontecimientos vividos en este último periodo de tiempo, se produjo una mala gestión de convalidaciones entre universidades que imposibilito cursar y dar comienzo al proyecto. Debido a esto, el tiempo que se le ha podido dedicar al proyecto ha sido desde que pude viajar de vuelta y la última convocatoria de presentación del año.

Estos meses de desarrollo han sido todo un reto, tanto en el estudio y aprendizaje del motor Unreal Engine 4 como en la escritura de la memoria de proyecto.

Sin embargo, pese a que en un principio se quería implementar una localización entera del juego, debido al tiempo con el que se contaba y la necesidad de estudio y autoaprendizaje previo que se ha requerido para poder hacer uso del motor, sólo se ha llevado a cabo la implementación del primer nivel de manera completa, junto a toda su interfaz de usuario. También hay que destacar la parte dedicada al ajuste y balanceo del juego, que se ha ido realizando de forma paralela junto a la creación del mismo. Es probable que el juego siga necesitando algún que otro ajuste, balanceo o corrección de errores. Por ese motivo todos los títulos de juegos actuales requieren de actualizaciones, ya que gracias al feedback de los jugadores se pueden localizar este tipo de incidencias que no han sido encontradas anteriormente.

Para concluir, aunque el proyecto sea un prototipo, se han podido cumplir todos los objeticos marcados al inicio del proyecto con un buen resultado desde la opinión personal. Gran parte del esfuerzo ha sido centrado en la jugabilidad, ya que, al tratarse de un juego del género de plataformas, es necesario que la jugabilidad sea lo más destacable del título. Otro factor a destacar es que, a pesar de sólo contar con un nivel, éste está implementado al 100%, enseñando todo lo que podría ofrecer el juego en un estado definitivo y dando pie a posibles trabajos futuros, tomándolo como base.

10. Trabajo futuro

En este apartado se detallan los diferentes puntos que quedan pendientes para el desarrollo de la versión final del juego, y otros aspectos los cuales se podrían mejorar o modificar.

General

- Crear todos los niveles de las diferentes localizaciones que se han especificado en el apartado de diseño.
- Gestión de datos de guardado y carga del juego.
- Compatibilizar los controles al 100% para controladores tanto de Xbox, PlayStation …
- Poder llevar el juego a otras plataformas cómo Nintendo Switch.

Personajes

- Añadir la dualidad en cuanto a personajes jugables dentro de los niveles. Es decir, en el título original se puede controlar a dos personajes diferentes. Mientras tienes el control de uno el otro te sigue, pudiendo cambiar el control del personaje durante el nivel.
- Crear nuevos enemigos con diferentes comportamientos e interacciones con el jugador y el escenario.
- Dentro de los enemigos, añadir jefes finales en cada localización.

Objetos

- Implementar un barril lanzador. Cuando colisionas con el barril, absorbe al jugador y lo dispara hacia una dirección. Este barril tiene ciertas variaciones como la posibilidad de que vaya girando y el jugador decida el momento de salir disparado, etc.
- Añadir la recolección de vidas durante el nivel.

Acciones

- Lanzar objetos
- Planear con el segundo personaje
- Poder manejar el control de ciertos aliados como los loros o elefantes, que se incluirían como “vehículos” manejables dentro de algunos puntos de ciertos niveles.

Por último, también está la idea de diseñar y desarrollar un modo multijugador local, que permita que dos jugadores puedan jugar simultáneamente el nivel controlado cada uno de ellos un personaje.

Llevando a cabo una revisión y mejora de la parte estética y mecánica, junto a la implementación de los puntos explicados, se conseguiría un producto final competitivo en el mercado actual del género de plataformas.

11. Bibliografía

1. App Creatly. app.creatly.com. sin fecha. https://app.creately.com/.
2. Autodesk. autodesk.com. sin fecha. https://www.autodesk.com/.
3. DKC Atlas. dkc-atlas.com. sin fecha. http://www.dkc-atlas.com/maps/dkc2/1-3.
4. Francisco Arias. youtube.com/FranciscoArias. sin fecha. https://www.youtube.com/watch?v=-Ds_LFLSjjI&list=PLIhhHOMPgei55ot0i_ks5wdOWAm_IfmLw&index=5&t=0s.
5. Freesound. Freesounds.org. sin fecha. https://freesound.org/people/Leszek_Szary/sounds/146718/.
6. GanntProject. Ganttproject.biz. sin fecha. https://www.ganttproject.biz/.
7. GIMP. gimp.com. sin fecha. https://www.gimp.org/.
8. Godot. godotengine.org. sin fecha. https://godotengine.org/.
9. Platformer Starter Pack. www.unrealengine.com. sin fecha. https://www.unrealengine.com/marketplace/en-US/product/platformer-starter-pack
10. Realtime 3D NowYoshi. youtube.com/Realtime3DNowYoshi. sin fecha https://www.youtube.com/watch?v=ZgL5HTz0L1M&list=PLIhhHOMPgei55ot0i_ks5wdOWAm_IfmLw&index=16&t=1459s.
11. Sprites DKC2. www.spriters-resource.com. sin fecha. https://www.spriters-resource.com/.
12. Strigifo. youtube.com/Strigifo. sin fecha. https://www.youtube.com/watch?v=tv-6fpB80mk&list=PLIhhHOMPgei55ot0i_ks5wdOWAm_IfmLw&index=10&t=0s.
13. Unity. unity.com. sin fecha. https://unity.com/.
14. Unreal Engine. unrealengine.com. sin fecha. https://www.unrealengine.com/.
15. Wikipedia. Wikipedia.org. sin fecha. https://es.wikipedia.org/wiki/ .