Escuela Politécnica Superior

Desarrollo de un Videojuego con Unreal Engine 4

Grado en Ingeniería Multimedia

Trabajo Fin de Grado

Autor:
José María Egea Canales

Tutor/es:
Miguel Ángel Lozano Ortega

Universitat d'Alacant
Universidad de Alicante

Septiembre 2015

Justificación y Objetivos

Dadas las capacidades aprendidas durante el Grado en Ingeniería Multimedia que nos permiten abarcar diversos aspectos en cuanto a contenidos multimedia, se pretende desarrollar un videojuego 3D que ponga a prueba estas capacidades.

Para la construcción del videojuego, se pretende utilizar el motor Unreal Engine 4. Además, el proyecto incluye tanto el Documento de Diseño del Videojuego (GDD), como la elaboración del mismo.

El videojuego estará realizado utilizando el sistema de scripting visual Blueprints, de Unreal Engine 4. Además, será compatible con el periférico Oculus Rift DK1.

Para la construcción de los escenarios de juego y menús, se abarcarán temas de editor de terrenos, de efectos de partículas, iluminación, sonorización, entre otros.

Índice de contenidos

Índice de contenidos 1
1. Introducción 9
2. Marco teórico o Estado del arte 10
2.1. Concepto de Videojuego 10
2.2. Motores para producir videojuegos 10
2.2.1. Motores en su contexto 10
2.2.2. Comparativa de motores para videojuegos en el mercado 11
2.2.2.1. Unreal Engine 4. 13
2.2.2.2. Unity. 13
2.2.2.3 CryEngine. 14
2.2.3. Elección del motor 15
2.2.4. Conceptos de la Arquitectura de Unreal Engine 4 y su sistema de programación de scripting Blueprints 15
2.2.4.1. UObjects y Actores 16
2.2.4.2. Gameplay y Clases del Framework 16
2.2.4.3. Representando jugadores, amigos, y enemigos en el mundo 16
2.2.4.3.1. Pawn 16
2.2.4.3.2. Character 17
2.2.4.4. Controlar Pawns con la entrada de jugador o con Inteligencia Artificial. 17
2.2.4.4.1 Controller 17
2.2.4.4.2. PlayerController 17
2.2.4.4.3 AIController 17
2.2.4.5. Mostrando información de los jugadores 17
2.2.4.5.1. HUD 17
2.2.4.5.2. Camera 17
2.2.4.6. Estableciendo el seguimiento y las reglas de juego. 17
2.2.4.6.1. GameMode 17
2.2.4.6.2. GameState 18
2.2.4.6.3. PlayerState 18
2.2.4.7. Relaciones de las clases del framework 18

2.2.4.8. Blueprints Visual Scripting 19
2.2.4.8.1. Blueprints en términos generales 19
2.2.4.8.2. Tipos de Blueprints 20
2.2.4.8.2.1. Clase Blueprint 20
2.2.4.8.2.2. LevelBlueprint 20
2.2.4.8.2.3. Blueprint Interface 20
2.2.4.8.3. Anatomía de un Blueprint 21
2.2.4.8.3.1. Ventana de Componentes (Components Window) 21
2.2.4.8.3.2. Construction Script 21
2.2.4.8.3.3. Event Graph 21
2.2.4.8.3.4. Functions 22
2.2.4.8.3.5. Variables 22
3. Objetivos 23
3.1. Objetivo principal del Trabajo de Fin de Grado. 23
3.2. Desglose de Objetivos. 23
4. Metodología 24
4.1. Metodología de desarrollo 24
4.2. Gestión del proyecto 25
4.3. Control de versiones y repositorio 25
5. Cuerpo del trabajo 26
5.1. Documento de Diseño del Videojuego (GDD) 26
5.1.1. El juego en términos generales 26
5.1.1.1. Resumen del argumento 27
5.1.1.2. Conjunto de características 27
5.1.1.3. Género 27
5.1.1.4. Audiencia 28
5.1.1.5. Resumen del flujo de juego 28
5.1.1.6. Apariencia del juego 29
5.1.1.7. Ámbito 30
5.1.2. Jugabilidad y mecánicas 31
5.1.2.1. Jugabilidad 31
5.1.2.1.1. Objetivos del juego 31
5.1.2.1.2. Progresión 31
5.1.2.1.3. Misiones y estructura de retos 32
5.1.2.1.4. Acciones del personaje 38
5.1.2.1.5. Controles de juego 39

5.1.2.2. Mecánicas 40
5.1.2.3. Opciones de juego 48
5.1.2.4. Rejugar y salvar........ 49
5.1.3. Historia, características, y personajes 54
5.1.3.1. Historia........ 54
5.1.3.2. Mundo de juego........ 55
5.1.3.2.1. Isla de inicio. 56
5.1.3.2.2. Isla de Tótem 1. 56
5.1.3.2.3. Isla de Tótem 2. 57
5.1.3.2.4. Isla de Tótem 3. 58
5.1.3.2.5. Isla de Tótem 4. 58
5.1.3.2.6. Isla de Tótem 5. 59
5.1.3.2.7. Isla de Tótem 6. 59
5.1.3.2.8. Isla de Tótem 7. 60
5.1.3.2.9. Isla de Tótem 8. 60
5.1.3.2.10. Isla del poblado. 61
5.1.3.2.11. Isla del bosque. 62
5.1.3.2.12. Isla final. 62
5.1.3.3. Personaje Principal. Arbennig. 63
5.1.3.3.1. Personaje NPC 1. Árbol de la vida. 65
5.1.3.3.2. Personaje NPC 2. Tótems. 65
5.1.3.3.3. Personaje NPC 3. Seres con aspecto de ancianos. 66
5.1.4. Nivel de juego – Delfrydoland. 68
5.1.4.1. Resumen 68
5.1.4.2. Material de introducción 68
5.1.4.3. Objetivos........ 69
5.1.4.4. Mapa 69
5.1.4.5. Caminos........ 69
5.1.4.6. Encuentros 70
5.1.4.7. Guía del nivel 70
5.1.4.8. Material de cierre........ 71
5.1.5. Interfaz 71
5.1.5.1. HUD........ 71
5.1.5.2. Menús 73
5.1.5.2.1. Menú Principal. 74
5.1.5.2.2. Menú de opciones. 75

5.1.5.2.3. Menú de Pausa. 76
5.1.5.2.4. Menú de confirmación de nueva partida 76
5.1.5.2.5. Menú de partida terminada (cuando jugador muere) 77
5.1.5.3. Cámara 80
5.1.6. Sonido 80
5.1.7. Sistema de ayuda 81
5.1.8. Inteligencia Artificial 82
5.1.8.1. IA de NPCs (Árbol de la vida, aldeanos con aspecto de anciano y tótem de la isla final) 82
5.1.8.2. IA de los tótems guardianes de las runas. 83
5.1.8.3. IA del Golem. 84
5.1.9. Guía Técnica[25] 85
5.1.9.1. Hardware 85
5.1.9.2. Software 85
5.1.9.3. Arquitectura del juego 85
5.2. Desarrollo e implementación. 86
5.2.1. Creación del proyecto 86
5.2.2. Implementar las acciones del personaje. 92
5.2.1.1. Rotación de la cámara. 94
5.2.1.2. Movimiento del personaje. 97
5.2.1.3. Ascender/Descender 101
5.2.1.4. Saltar 101
5.2.1.5. Frenada en seco 102
5.2.1.6. Sprint corriendo 102
5.2.1.7. Turbo en Racing 103
5.2.1.8. Gestión de cambios de estado. 104
5.2.1.8.1. Cambios de estado manuales. 104
5.2.1.8.1.1. Cambios de estado manuales de Falling 104
5.2.1.8.1.2. Cambios de estado manuales de vuelo 104
5.2.1.8.1.3. Cambios de estado manuales de Racing 105
5.2.1.8.2. Gestión de cambios de estado automáticos 106
5.2.1.8.2.1. Gestión de cambio de estado automático entre Walking y Falling 107
5.2.1.8.2.2. Interpolaciones de cambio de estado 108
5.2.1.8.2.3. Interpolación automática de flying a falling 109
5.2.1.8.2.4. Interpolación automática de falling a flying 111

5.2.1.8.2.5. Interpolación automática de Racing a Flying ... 113
5.2.1.8.2.6. Activar interpolaciones ... 115
5.2.1.9. Compatibilizar controles con Oculus Rift DK1. ... 116
5.2.3. Desarrollo del nivel de juego. ... 117
5.2.3.1. Pipeline de creación e importación de Assets. ... 117
5.2.3.1.1. Importación de Assets desde 3ds max ... 117
5.2.3.1.2. Importación de assets desde otro proyecto de UE4 ... 119
5.2.3.2. Creación de terrenos (landscape) del proyecto. ... 119
5.2.3.3. Crear un efecto de partículas de chispas de fuego. ... 121
5.2.3.3.1. Anatomía de un sistema de partículas. ... 122
5.2.3.3.2. Implementando el sistema de chispas. ... 123
5.2.3.4. Añadir la iluminación. ... 131
5.2.4. Sonorización de juego. ... 131
5.2.5. Implementación de la Interfaz de Usuario ... 132
5.2.5.1. Pipeline de la creación de interfaces de usuario con Widget Blueprints ... 133
5.2.5.2. Uso en el juego ... 134
5.2.5.3. Menú principal ... 134
5.2.5.4. Menú de opciones ... 135
5.2.5.4.1. Modificar resolución de pantalla. ... 135
5.2.5.4.2. Modificar el volumen general de juego. ... 136
5.2.5.5. Menú de Pausa ... 137
5.2.5.6. HUD ... 138
5.2.5.6.1. Inventario. ... 138
5.2.5.6.2. Temporizador para pruebas de aros ... 141
5.2.5.6.3. Pantalla de Prólogo, Epílogo, y Créditos. ... 142
5.2.5.6.4. Pantallas de Desafío superado y no superado, partida guardada, y no puedes usar el objeto. ... 143
5.2.6. Implementación de las mecánicas de juego. ... 143
5.2.6.1. Teletransportador ... 144
5.2.6.2. Aros ... 146
5.2.6.3. Diálogos de los NPC y tótems ... 150
5.2.6.4. Puerta del final del juego. ... 154
5.2.7. Sistema de guardado/cargado de partida. ... 157
5.2.7.1. Guardar partida. ... 158
5.2.7.2. Cargar partida. ... 159

5.2.7.3. Borrar partida. 160
5.2.8. Implementación de la IA. 160
5.2.9. Implementación del flujo de juego general de juego en el Level Blueprint 163
6. Conclusiones 167
6.1. Propuestas de mejora y trabajo futuro. 168
7. Bibliografía y referencias 170
7.1. Introducción. 170
7.2. Marco Teórico o Estado del Arte. 170
7.3. Metodología 171
7.4. Documento de Diseño del Videojuego. 172
7.5. Desarrollo e implementación. 172

1. Introducción

Conforme avanzan los años, las personas desarrollan nuevas formas de entretenimiento, porque la necesidad de ocio forma parte de la misma naturaleza humana. Entre estas formas encontramos los videojuegos, que desde el comienzo de su historia por allá por la década de los 40[1], han ido creciendo de forma sustancial tanto en adeptos, como en géneros, formas de jugarlos, cantidad de equipos de desarrollo, y en tecnología, llegando a facturar desde hace varios años más que la música y el cine juntos en España, y cada año aumentando[2].

Actualmente además, dada la gran cantidad de tecnologías para desarrollar videojuegos, la cantidad de plataformas, y la cada vez mayor asimilación de la tecnología por parte de todas las personas por su costo cada vez menor, es posible encontrar en el mercado de los videojuegos desde proyectos con costes multimillonarios, como el caso de Destiny, cuyo costo alcanzó los 380 millones de euros[3], con equipos de desarrollo expertos en muchas disciplinas y trabajando duro durante varios años, hasta proyectos con coste 0 que puede hacer cualquier persona en su casa y que pueden acumular millones, como el caso de Flappy Bird para Android, que lo creó una persona en su casa en varios días, y que cada día hacía ganar al creador, Dong Nguyen, unos 90000 dólares estadounidenses[4].

Con todo esto, se puede apreciar que el mundo de los videojuegos puede resultar una apuesta interesante para cualquier desarrollador multidisciplinario, o pequeño equipo de desarrollo, y hablamos de multidisciplinario porque un proyecto de videojuegos incluye trabajo de programación, de sonorización, de diseño, y también artístico 2D y cada vez más, 3D. Es entonces, aquí donde se entra en cuestión. Como ingenieros multimedia, debemos ser capaces de manejar el desarrollo de un proyecto de estar características, en especial, si hemos realizado la especialización de creación y entretenimiento digital, donde incluso se tratan aspectos de Realidad Virtual[5].

Para este Trabajo de Fin de Grado, vamos a comprobar si es posible abarcar un proyecto multidisciplinario como es el de un videojuego 3D, y utilizaremos para ello el motor de videojuegos Unreal Engine 4. Además, queremos que este proyecto, dado el itinerario que se ha realizado, ofrezca compatibilidad con el periférico Oculus Rift DK1, las gafas de realidad virtual con las que actualmente cuenta la universidad a disposición del alumnado. Crearemos además el Game Design Document (GDD) del videojuego, para describir todo el diseño del mismo a realizar posteriormente.

2. Marco teórico o Estado del arte

Con este bloque se pretende hacer un análisis de los motores de videojuegos en su contexto, realizar una comparación entre los motores más populares que existen ahora mismo, y también, destacar las utilidades y características que ofrece Unreal Engine 4, las cuales han llevado a la elección de esta herramienta para la creación del proyecto que vamos a documentar.

Es necesario destacar que es con el motor de videojuegos con lo que vamos a construir el juego, mientras que se utilizará herramientas de modelado y animación, programas de edición de audio, y otros de edición de gráficos, estos solo nos sirven para la producción de Assets (recursos), que a través del motor daremos uso. Es por esta razón que en el marco teórico se ha decidido abarcar como tema los motores de videojuegos, y el de videojuego en sí.

2.1. Concepto de Videojuego

Vamos a tratar establecer primeramente una definición de videojuego, de forma que podamos entender mejor qué estamos tratando con este trabajo.

Los videojuegos, aunque complejos, se basan en algo simple. Bernard Suits escribió que “jugar a un videojuego es un esfuerzo voluntario de superar obstáculos innecesarios”[6]. A su vez, Wikipedia nos da una definición, diciendo que un videojuego “es un juego electrónico en el que una o más personas interactúan, por medio de un controlador, con un dispositivo dotado de imágenes de vídeo”[7].

Teniendo en cuenta que un juego consiste en una actividad que:

- Requiere de al menos un jugador.
- Tiene reglas.
- Tiene una condición de victoria.

Lo más acertado probablemente con la definición de videojuego es “un juego que es jugado a través una pantalla de vídeo”.

2.2. Motores para producir videojuegos

2.2.1. Motores en su contexto

Antes de comenzar, cabe definir lo siguiente, ¿qué es un motor de videojuegos?

El término motor de videojuegos nace a mediados de los años noventa, en referencia a los primeros juegos en primera persona de disparos (FPS), especialmente el conocido Doom, de id Software. Doom fue diseñado con una arquitectura muy bien definida, contando con una excelente separación entre los componentes del núcleo del motor (como el sistema de renderizado

tridimensional, la detección de colisiones, o el sistema de audio), los recursos artísticos del juego, los mundos de juego, y las reglas para jugarlos.

El valor de esta separación se vuelve evidente cuando los desarrolladores empezaron a crear juegos que partían de esa estructura básica, pero con distinto arte, mundos, armas, vehículos, y otros assets (recursos), además de las reglas de juego, todo esto solo haciendo mínimos cambios al "motor" ya existente. Esto marcó el nacimiento de la comunidad de "mods", un grupo de jugadores individuales y pequeños estudios independientes que construían sus videojuegos modificando juegos ya existentes, usando set de herramientas (toolkits) gratuitos proporcionados por los desarrolladores originales. Y es que gracias a los motores de videojuegos, los diseñadores ya no dependen de los diseñadores, y además es posible añadir o cambiar partes del juego rápidamente, lo cual antes de su invención, podía resultar costoso[8].

A partir del motor de id software, otras empresas decidieron construirse su propio motor de videojuegos, tal como decidió hacer Epic Games con Unreal Engine en 1998[9], o Valve con su motor Source en 2004[10], y otras empresas posteriores.

2.2.2. Comparativa de motores para videojuegos en el mercado

Actualmente existe una lista muy amplia de motores para producir videojuegos, tanto 2D, como 3D, de pago y gratuitos, y que ofrecen mayor o menor compatibilidad con plataformas tanto para desarrollar como para las que producir videojuegos.

Nombrarlos todos sería complicado, dada la cantidad, Wikipedia hace una lista con una gran cantidad de ellos clasificados por licencia:

V·T·E		Game engines (list) [hide]
		Source port · First-person shooter engine (list) · Tile engine · Game engine recreation (list) · Game creation system
Free software / open source	2D	Adventure Game Studio · Beats of Rage · Box2D · Chipmunk · Cocos2d · Digital Novel Markup Language · Flixel · Exult · Game-Maker · Gosu · Jogre · KiriKiri · Moai SDK · ORX · Pygame · Ren'Py · StepMania · Stratagus · Thousand Parsec · VASSAL · Xconq
	2.5D	Aleph One · Build · Flexible Isometric Free · Id Tech 1 · Wolfenstein 3D
	3D	Away3D · Axiom · Blender Game · Cafe · Crystal Space · Cube · Cube 2 · Delta3D · Dim3 · Genesis3D · GLScene · Horde3D · HPL 1 · Irrlicht · Id Tech 2 · id Tech 3 (Ioquake3) · id Tech 4 · JMonkey · Luxinia · OGRE · Ogre4j · Open Wonderland · Panda3D · Papervision3D · Platinum Arts Sandbox Free 3D Game Maker · PlayCanvas · PLIB · Python-Ogre · Quake · Nebula Device · RealmForge · Retribution · Torque 3D
	Mix	Allegro · Construct Classic · Godot · Lightweight Java Game Library · Spring · Visualization Library
Proprietary	2D	Clickteam Fusion · Coldstone · Construct 2 · Corona · CRX · Fighter Maker · Filmation · GameMaker · GameMaker: Studio · Garry Kitchen's GameMaker · Generic Tile · Gold Box · MADE · Mscape · M.U.G.E.N · NScripter · RPG Maker · Shoot the Bullet · Sim RPG Maker · Sound Novel Tsukūru · Southpaw · Stencyl · Vicious · Virtual Theatre · V-Play · Z-machine · Zillions of Games · ZZT
	2.5D	Genie · INSANE · Infinity · Jedi · Pie in the Sky · Super Scaler · UbiArt Framework
	3D	4A · Advance Guard Game · Anvil/Scimitar · Arsys · Beelzebub · Bork3D · BRender · C4 · Chrome · Creation · CryEngine · Crystal Tools · Dagor · Diesel · Digital Molecular Matter · Disrupt · Dunia · EAGL · EGO · Electron · Elflight · Enforce · Enigma · Essence · Flare3D · Fox · Freescape · Frostbite · Geo-Mod · GoldSrc · HeroEngine · HydroEngine · HPL 2 · id Tech 5 · id Tech 6 · Ignite · Iron · IW · Jade · Kinetica · LS3D · Leadwerks · LithTech · Luminous Studio · LyN · Marmalade · Mizuchi · MT Framework · NanoFX GE · Odyssey · Orochi · Outerra · Panta Rhei · Phoenix Engine (Relic) · Phoenix Engine (Wolfire) · PhyreEngine · Q · Real Virtuality · REDengine · Refractor · RenderWare · Revolution3D · Riot · RAGE · SAGE · Serious · Shark 3D · Silent Storm · Sith · Source · SunBurn XNA · Titan · TOSHI · Truevision3D · Unigine · Unity · Unreal · Vengeance · Visual3D · Voxel Space · XnGine · X-Ray · Yebis · YETI · Zero
	Mix	CPAGE · Dark · Gamebryo · Hybrid Graphics · Kaneva Game Platform · Metismo
Proprietary Game middleware (list)		AiLive · Euphoria · Gameware · GameWorks · Havok · iMUSE · Kynapse · Quazal · SpeedTree · Xaitment

Figura 1. Lista de motores

Fuente: Wikipedia[11]

Nosotros vamos a destacar y comparar los 3 motores más famosos para la producción de videojuegos actualmente. Estos son Unreal Engine 4 de Epic Games, Unity de Unity Technologies, y CryEngine de Crytek[15].

2.2.2.1. Unreal Engine 4.

Figura 2. Logo Unreal Engine

Fuente: Wikimedia[12]

Unreal Engine 4, abreviado UE4, tiene unas excelentes capacidades gráficas, incluyendo, entre otros aspectos, funcionalidades avanzadas de iluminación dinámica y un sistema de partículas que puede manejar hasta un millón de partículas en una escena a la vez.

A pesar de que Unreal Engine 4 es el sucesor del UDK, el cambio de versión entre UDK y UE4 es realmente notorio. Cambios realizados con el objetivo de mejorar la facilidad de producción de videojuegos.

Entre los cambios a destacar, uno de los principales es el lenguaje de scripting para la UE4, que en UDK era el lenguaje UnrealScript, y ahora ha sido completamente sustituido por C ++ en UE4. Además, UE4 utiliza ahora Blueprints para scripting gráfico, una versión avanzada de Kismet con la que se puede llegar a realizar por completo un videojuego sin necesidad de programar en C++.

Unreal Engine 4 tiene una licencia de cobrar el 5% de las ganancias de un videojuego a partir de los primeros 3000 dólares americanos por cada cuatrimestre.

2.2.2.2. Unity.

Figura 3. Logo Unity

Fuente: Wikimedia[13]

El motor Unity ofrece una amplia gama de características y posee una interfaz sencilla de entender. Además, posee un sistema de integración multiplataforma que permite exportar juegos para casi todas las existentes, siendo actualmente la mejor opción para desarrollo 3D en plataformas de Android, por sus herramientas de compresión que permiten que los videojuegos no sean especialmente pesados, y no consuman excesivos recursos.

El motor de juego es compatible con las principales aplicaciones de modelado y animación 3D, como 3ds Max, Maya, Softimage, Cinema 4D, y Blender, entre otros, lo cual se traduce en que soporta la lectura de archivos exportados con estos programas sin ofrecer ningún problema.

Como contra, Unity es un motor de pago, donde a pesar de desbloquear desde su versión 5.0 casi todas las características que poseía antes solo la versión pro, en este aspecto, Unreal Engine 4 sale aventajado al ser gratuito y ofrecer a cualquier persona que lo descargue, el 100% de todas las funcionalidades del motor.

2.2.2.3 CryEngine.

Figura 4. Logo CryEngine

Fuente: Wikimedia[14]

CryENGINE es un motor considerablemente potente diseñado por la empresa Crytek, que se introdujo al mundo de los videojuegos con FarCry.

Este motor ha sido diseñado para ser utilizado en consolas de sobremesa y PC, incluyendo la actual generación de Sony y Microsoft, es decir, PlayStation 4 y Xbox One.

Las capacidades gráficas de CryENGINE se pueden equiparar a las de Unreal Engine 4, con excelente iluminación, física realista, sistemas avanzados de animación, etc. Además, de igual modo que UE4, el motor posee unas características de editor y diseño de niveles muy potentes.

Por otro lado, a pesar de que CryEngine es un motor realmente potente, su curva de aprendizaje es un poco complicada para empezar a utilizar el motor de juego de manera productiva. Si no se pretende producir un juego de características visuales a nivel de triple A (producciones normalmente multimillonarias), no se suele recomendar su utilización.

CryEngine tiene una licencia de suscripción con coste 9.90€$ al mes. Para proyectos de gran envergadura es necesario ponerse en contacto con ellos para obtener una licencia que permita acceso al 100% del código del motor y a asistencia directa de Crytek.

2.2.3. Elección del motor

Si nos hacemos la pregunta a nivel de desarrollador de qué motor elegir, nos encontramos ante una decisión a considerar, ya que cuenta con una curva de aprendizaje importante y además es necesario familiarizarse con numerosas herramientas. Es por esto que lo recomendable es elegir aquel que se adapte mejor a nuestras necesidades, sin embargo, es conveniente tener en cuenta una serie de detalles[16]:

- Que ofrezca una buena documentación.
- Que tenga una buena comunidad de usuarios que además no se haya abandonado.
- Tener en cuenta si queremos modificar el motor o no.

Para este proyecto hemos optado por utilizar el motor Unreal Engine 4 que, además de ofrecer el 100% de sus características de forma gratuita, permitirnos obtener fácilmente unos resultados considerablemente buenos, y tener una comunidad actual de usuarios a pesar de haber salido en 2014 que permite resolver actualmente prácticamente cualquier duda que se plantee, incorpora la tecnología Blueprints, la cual nos permite desarrollar teóricamente un videojuego completo al 100% sin necesidad de programar, y en la que vamos a basar al completo el desarrollo.

2.2.4. Conceptos de la Arquitectura de Unreal Engine 4 y su sistema de programación de scripting Blueprints

A continuación vamos a hacer una breve introducción a las características del núcleo del motor, y a introducir algunos conceptos de la arquitectura[17]. Posteriormente, realizará una introducción al sistema de scripting gráfico de Unreal Engine 4, Blueprints[19].

2.2.4.1. UObjects y Actores

Los actores son instancias de clases que derivan desde la clase AActor; la clase base de todos los objetos que pueden ser posicionados en el mundo de juego.

Los objetos son instancias de clases que heredan desde la clase UObject; la clase base para todos los objetos en Unreal Engine, incluyendo los actores. Así que, en realidad, todas las instancias en Unreal Engine son Objetos; a pesar de esto, el término Actor es usado comúnmente para referirnos a instancias de clases que derivan desde AActor en su jerarquía, mientras que el término Objeto es usado para referirnos a instancias de clases que no heredan desde la clase AActor. La mayoría de las clases que creemos heredarán de AActor en algún punto de su jerarquía.

En general, los actores pueden ser interpretados como ítems en sí o como entidades, mientras que los Objetos son partes más especializadas. Los actores a menudo hacen uso de Componentes, que son Objetos especializados, para definir ciertos aspectos de su funcionalidad o guardar valores de una colección de propiedades. Tomando como ejemplo un coche, el coche es el Actor en sí, mientras que las partes del coche, como las ruedas y puertas, probablemente sean componentes de ese Actor.

2.2.4.2. Gameplay y Clases del Framework

Las clases de gameplay básicas incluyen funcionalidad para representar jugadores, aliados, y enemigos, así como el control de estos avatares con la entrada del jugador o con una inteligencia artificial. Hay también clases para la creación de HUDs (heads-up displays) y cámaras para jugadores. Finalmente, las clases de gameplay como GameMode, GameState, y PlayerState establecen las reglas del juego, y permiten hacer un seguimiento cómo el juego y los jugadores progresan.

Todas estas clases son tipos de Actores, las cuales es posible posicionar en el nivel a través del editor, o hacer spawn (hacerlas aparecer) de ellas cuando sea necesario.

2.2.4.3. Representando jugadores, amigos, y enemigos en el mundo

2.2.4.3.1. Pawn

Un Pawn es un Actor que puede ser un “agente” dentro del mundo. Los Pawn pueden ser poseídos por un Controller, ya que están creados para aceptar input de forma fácil, y pueden realizar acciones propias de cada uno. Hay que tener en cuenta que un Pawn no tiene por qué ser humanoide.

2.2.4.3.2. Character

Un Character es un Pawn de estilo humanoide. Viene con un CapsuleComponent para colisión, y un CharacterMovementComponent por defecto. Puede realizar acciones básicas humanas como moverse, posee funcionalidad para su uso en red, y tiene algo de funcionalidad de animación.

2.2.4.4. Controlar Pawns con la entrada de jugador o con Inteligencia Artificial.

2.2.4.4.1 Controller

Un Controller es un Actor que puede ser responsable de dirigir a un Pawn. Típicamente viene de dos formas, AIController y PlayerController. Un Controller puede "poseer" un Pawn y tener el control sobre él.

2.2.4.4.2. PlayerController

Un PlayerController es una interfaz entre el Pawn y el jugador humano controlándolo. El PlayerController esencialmente realiza básicamente lo que el jugador le ordena hacer.

2.2.4.4.3 AIController

Una AIController es una inteligencia artificial que puede controlar a un Pawn.

2.2.4.5. Mostrando información de los jugadores

2.2.4.5.1. HUD

Un HUD es un "heads-up display", o una pantalla 2D que es común en muchos juegos. En ella se puede representar en pantalla salud, munición, etc. Cada PlayerController tiene normalmente un HUD.

2.2.4.5.2. Camera

La PlayerCameraManager son los ojos y gestiona cómo se comporta. Cada PlayerController tiene típicamente una de ellas.

2.2.4.6. Estableciendo el seguimiento y las reglas de juego.

2.2.4.6.1. GameMode

El concepto de "juego" está dividido en dos clases. La clase GameMode es la definición del juego, que incluye aspectos como sus reglas y condiciones de victoria. Solo existe en el servidor. No debería tener muchos datos que cambien durante el juego.

2.2.4.6.2. GameState

La clase GameState contiene el estado del juego, que podría incluir aspectos como la lista de jugadores conectados, la puntuación. Es lo que las piezas son en el ajedrez, o la lista de misiones que se deben completar en un juego de mundo abierto. GameState existe en el servidor y en todos los clientes, y puede replicar información libremente para mantener todas las máquinas actualizadas.

2.2.4.6.3. PlayerState

PlayerState es el estado de un participante en un juego, como un jugador humano o un bot que simula a un jugador. La IA que existe como parte del juego no debería tener PlayerState. Como datos de ejemplo que serían apropiados en un PlayerState incluyen el nombre del jugador, su puntuación, o si está llevando una bandera en un juego de captura la bandera. Hay una clase PlayerState para cada uno de los jugadores que existen en cada una de las máquinas, y pueden replicarse en la red libremente para mantener todo sincronizado.

2.2.4.7. Relaciones de las clases del framework

El siguiente diagrama ilustra cómo todas estas clases del núcleo de juego se relacionan entre ellas. Un juego está hecho de un GameMode y un GameState. Los jugadores humanos que se unen al juego están asociados con PlayerControllers. Estos PlayerController permiten a los jugadores poseer Pawns en el juego, y así poder tener representación física en el nivel. Los PlayerControllers también proporcionan a los jugadores la entrada de controles, un HUD, y una PlayerCameraManager para manejar las vistas de cámara.

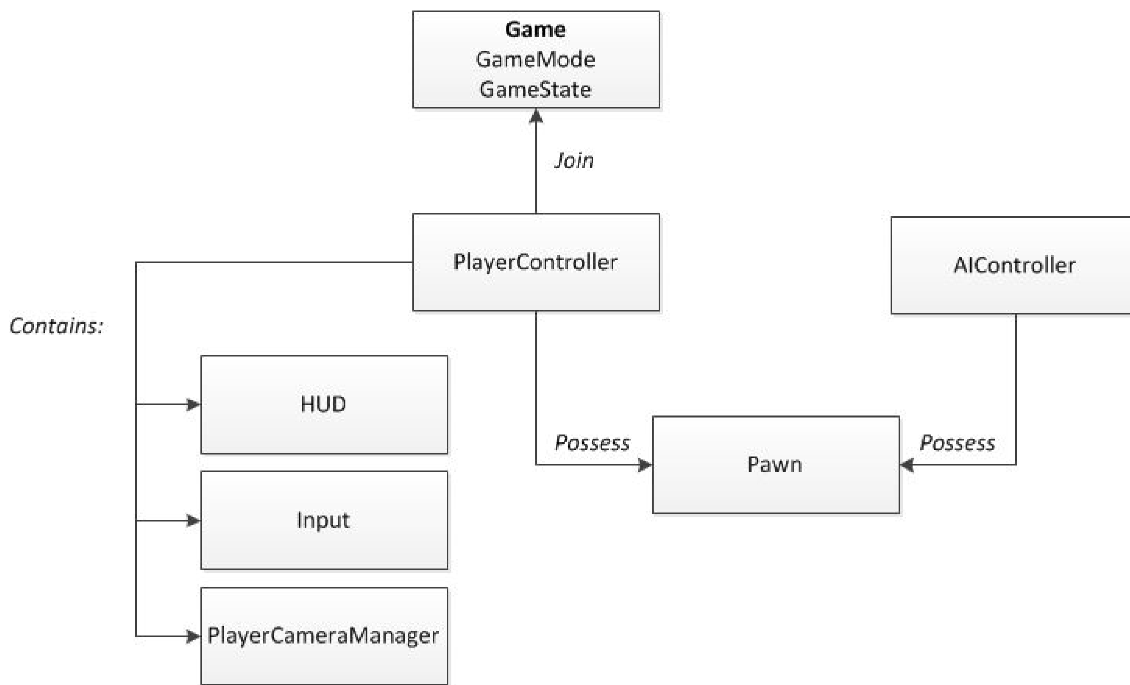


Figura 5. Diagrama de relación de clases en Unreal Engine.

Fuente. Web de Unreal Engine[18]

2.2.4.8. Blueprints Visual Scripting

El Scripting Visual de Blueprints en Unreal Engine es un sistema completo de gameplay implementado con el concepto de una interfaz basada en nodos para crear elementos de gameplay a través del Editor de Unreal (Unreal Editor). Este sistema es muy flexible y potente dado que proporciona la habilidad a los diseñadores de usar virtualmente el completo rango de conceptos y herramientas generalmente antes solo disponibles a programadores.

A través del uso de Blueprints, los diseñadores pueden crear prototipos, implementar, o modificar virtualmente cualquier elemento de gameplay, como:

- Juegos. Las reglas de juego, condiciones de victoria, etc.
- Jugadores. Modificar y crear mallas y materiales o personalizar personajes.
- Cámaras. Crear prototipos de perspectivas de cámaras o cambiar la cámara dinámicamente durante el juego.
- Entrada (Inputs). Cambiar los controles del jugador o permitir a los jugadores pasar sus controles a ítems.
- Ítems. Armas, hechizos, triggers, etc.
- Ambientes. Crear assets aleatorios o generar ítems de forma procedural.

2.2.4.8.1. Blueprints en términos generales

Los Blueprints son assets especiales que proporcionan una interfaz basada en nodos intuitiva que puede ser usada para crear nuevos tipos de actores y realizar scripting de eventos de nivel, otorgando a los diseñadores y programadores de gameplay las herramientas para crear e iterar gameplay rápidamente a través del editor de Unreal sin la necesidad de escribir código.

2.2.4.8.2. Tipos de Blueprints

Los Blueprints pueden ser de varios tipos, poseyendo cada tipo su propia funcionalidad, vamos a definir los más importantes que pretendemos utilizar en el proyecto.

2.2.4.8.2.1. Clase Blueprint

Una clase Blueprint, normalmente abreviado como Blueprint, es un asset que permite a los creadores de contenido añadir rápidamente funcionalidad a las clases del juego existentes. Los Blueprints son creados dentro del Editor de Unreal de forma visual, en lugar de escribiendo código, y son guardados como assets en un paquete de contenidos. Estos definen básicamente nuevas clases o tipos de actores que pueden entonces ser posicionados en mapas como instancias que se comportan como cualquier otro tipo de Actor.

2.2.4.8.2.2. LevelBlueprint

El LevelBlueprint es un tipo especializado de Blueprint que actúa como el grafo de nodos global del nivel. Cada nivel en un proyecto tiene su propio Level Blueprint creado por defecto que puede ser editado a través del editor de unreal. No es posible crear nuevos Level Blueprints a través del editor de interfaces.

Los eventos pertenecientes al nivel en sí, o instancias específicas de Actores a través del nivel, son usados para activar secuencias de acciones en forma de llamadas a funciones u operaciones de flujo de control. Es básicamente el mismo concepto para el que servía Kismet en UDK.

El LevelBlueprint también proporciona un mecanismo de control para el streaming y Matinee así como para comunicarse con eventos de los actores situados dentro del nivel.

2.2.4.8.2.3. Blueprint Interface

Una Blueprint Interface es una colección de una o más funciones que puede ser añadida a otros Blueprints. Cualquier Blueprint que tenga la interfaz debe tener esas funciones.

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

de objetos compartir y acceder a variables de una interfaz común. Para simplificar, una Blueprint Interface permite a diferentes Blueprints compartir y enviar datos a otros.

Las Blueprint Interfaces pueden ser hechas por creadores de contenido a través del editor de una forma similar a otros Blueprints, pero vienen con algunas limitaciones que no es posible realizar en ellas:

- Añadir nuevas variables.
- Editar grafos.
- Añadir componentes.

2.2.4.8.3. Anatomía de un Blueprint

La funcionalidad de los Blueprints está definida por varios elementos, algunos de los cuales son presentados por defecto, mientras que otros pueden ser añadidos en función de nuestras necesidades. Esto nos otorga la habilidad de definir componentes, mejorar la inicialización y operaciones de inicialización, responder a eventos, organizar y modularizar operaciones, definir propiedades, etc.

2.2.4.8.3.1. Ventana de Componentes (Components Window)

Con el entendimiento de qué son los Componentes, la ventana de componentes dentro del Blueprint Editor permite añadir nuevos a nuestro Blueprint. Esto nos proporciona formas de añadir geometría de colisión como CapsuleComponents, BoxComponents, o SphereComponents, añadir geometría en forma de StaticMeshComponent o SkeletalMeshComponent, controlar el movimiento usando MovementComponents, etc. Los componentes añadidos en la lista de componentes pueden también ser asignados a variables de instancia proporcionando el acceso a ellas en su grafo de Blueprint o en otros Blueprints.

2.2.4.8.3.2. Construction Script

El Construction Script entra en funcionamiento siguiendo a la lista de componentes cuando una instancia de una clase Blueprint es creada. Contiene un nodo de grafo que es ejecutado permitiendo a la instancia de la clase Blueprint ejecutar operaciones de inicialización. Esta característica resulta útil para acciones de modificar propiedades de mallas y materiales, o pintar trazos en el mundo. Por ejemplo, un Blueprint con una luz podría determinar qué tipo de terreno hay situado y elegir la malla correcta para usar desde un conjunto de mallas.

2.2.4.8.3.3. Event Graph

El EventGraph de un Blueprint contiene un grafo de nodos que usa eventos y llamadas a funciones para realizar acciones en respuesta a eventos de gameplay asociados con el Blueprint. El EventGraph es usado para añadir funcionalidad que es común a todas las instancias del Blueprint. Además, es donde la interactividad y las respuestas dinámicas se establecen. Por ejemplo, una luz Blueprint podría responder a un evento de daño, apagándose, y este comportamiento lo tendrían todas las luces que hayamos creado de ese tipo de blueprint, de modo que conforme vaya saltando su correspondiente evento de daño, se van apagando.

2.2.4.8.3.4. Functions

Las funciones son nodos de los grafos que poseen a un Blueprint particular, y pueden ser ejecutadas o llamadas desde otro grafo a través del Blueprint. Las funciones tienen un único punto designado por un nodo con el mismo nombre de la función conteniendo un único pin de ejecución de salida. Cuando una función es llamada desde otro grafo, el pin de ejecución de salida es activado, lo que hace que podamos proceder con la lógica del scripting.

Para explicarlo más claramente, si tenemos una función, tendremos un único pin de entrada y de salida, el pin de entrada es la señal para que se active la función, mientras que el de salida es una señal para que se ejecute otro componente, de esta forma, podemos crear cadenas de ejecución fácilmente.

2.2.4.8.3.5. Variables

Las variables son las propiedades que guardan valores o referencias a Objects o Actores en el mundo. Podremos acceder a estas propiedades de forma interna a través del Blueprint que las contenga, o pueden ser publicadas para poder acceder a ellas y modificar sus valores desde otros Blueprints.

3. Objetivos

3.1. Objetivo principal del Trabajo de Fin de Grado.

El objetivo de este proyecto es la realización de un videojuego 3D para PC utilizando el motor de videojuegos Unreal Engine 4, y centrando su desarrollo completo en el sistema de Scripting Visual Blueprints.

Con este objetivo se pretende realizar un Diseño previo del videojuego, analizar las características que ofrece el motor, y aprender tanto su funcionamiento, como pipeline necesario para la inclusión de Assets en él.

3.2. Desglose de Objetivos.

A continuación se presentan los objetivos del TFG en forma de tareas más específicas:

1. Realizar el GDD (Game Design Document) de un videojuego.
2. Cubrir los aspectos del pipeline necesario para la importación de Assets en el motor desde Autodesk 3ds Max.
3. Creación de un sistema de Menús y HUD.
4. Crear un mundo de juego utilizando las herramientas que ofrece Unreal Editor.
5. Implementar la lógica de juego al completo utilizando Blueprints.
6. Compatibilizar el videojuego con el dispositivo de realidad virtual Oculus Rift DK1.
7. Sonorizar el videojuego.

4. Metodología

En este apartado se abarcan los aspectos de Gestión de Proyectos y Metodología de Desarrollo. Como extra, se añade un ligero subapartado tratando aspectos de control de versiones y repositorio.

4.1. Metodología de desarrollo

La metodología de desarrollo elegida para el desarrollo de este proyecto ha sido Kanban[20], la metodología ágil basada en tableros con etiquetas.

El motivo de esta decisión viene marcado por la escasez de imposiciones sobre el desarrollo que ofrece la metodología, la orientación al número de personas al que va dirigida (normalmente a equipos pequeños), y la facilidad para controlar la carga de trabajo de forma adecuada. Además, como se trata de una metodología ágil, permite un desarrollo menos marcado en fases específicas y que soporta cambios con mayor facilidad que otro tipo de metodologías más cerradas o con unas especificaciones de diseño demasiado estrictas.

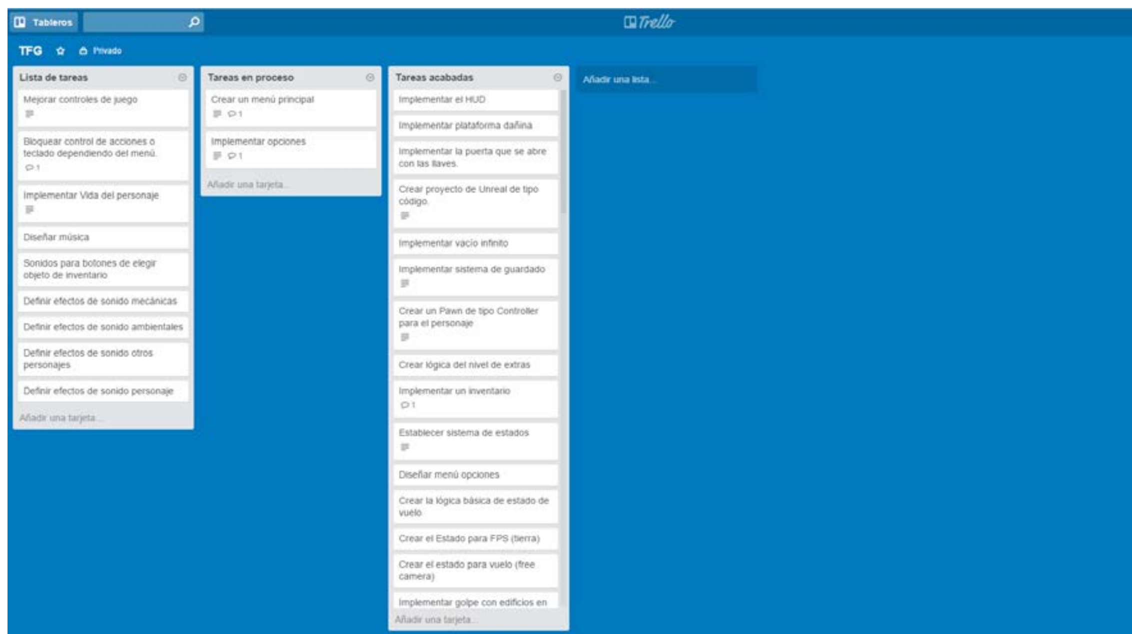


Figura 6. Captura de la organización Kanban del proyecto

Fuente: Elaboración propia.

El flujo de trabajo en este proyecto ha consistido en añadir nuevas tareas pendientes al tablero. Cuando se pretendía realizar alguna o varias tareas, se colocaban en una lista de tareas en proceso, y, cuando acababan, se colocaban por último en una lista de tareas terminadas.

4.2. Gestión del proyecto

Para la gestión de tareas del proyecto, se ha decidido utilizar la aplicación Trello, que permite la creación de un tablero y la organización de tareas por etiquetas. Las tareas en esta aplicación soportan una variedad de opciones que resultan cómodas en el momento de definir tareas o añadir modificaciones o notas a estas (en forma de comentarios en nuestro caso particular, al solo ser una persona). Como es multiplataforma, permite la gestión rápida de estas desde smartphones o tablets, para hacer posibles cambios o añadir nuevas tareas en cualquier momento.

4.3. Control de versiones y repositorio

Este trabajo se basa en la realización de un Videojuego en Unreal Engine 4, y teniendo además en cuenta que el número de participantes en la creación del mismo es únicamente el propio autor de esta memoria, no se ha establecido un control de versiones, dado que un proyecto en Unreal Engine 4 suele ser bastante pesado y realizar subidas periódicas de archivos de esta índole puede requerir un tiempo considerable, dado también que al no haber más integrantes en el desarrollo, solo una persona necesita trabajar sobre el proyecto, y por último, teniendo en cuenta que el mismo motor UE4 realiza autoguardados periódicos programables para evitar pérdidas de información, se ha optado por limitar a un par de copias del proyecto por seguridad de forma manual.

5. Cuerpo del trabajo

El cuerpo del trabajo de desarrollo queda dividido en dos bloques principales.

1. Un primer bloque dedicado al **Documento de Diseño del Videojuego (GDD)**, donde se cubren todos los aspectos del diseño del videojuego desarrollado.
2. Y un segundo bloque, de **Desarrollo e implementación**, correspondiente a los aspectos técnicos más relevantes de la producción del proyecto de estudio.

5.1. Documento de Diseño del Videojuego (GDD)

Figura 7. Logo del Videojuego.

Fuente: Elaboración propia

En este bloque se describe con detalle todos los aspectos de diseño necesarios para la correcta implementación posterior del videojuego desarrollado. Se trata tanto historia, experiencia de juego, y diseño de niveles, como interfaz de usuario y detalles técnicos.

Este documento está basado de acuerdo a la plantilla de GDD ofrecida en la asignatura de Fundamentos de los Videojuegos[21], obligatoria de tercer curso en Ingeniería Multimedia.

El nombre que se ha decidido dar al proyecto es **Arbennig: Mundo Equivocado**.

5.1.1. El juego en términos generales

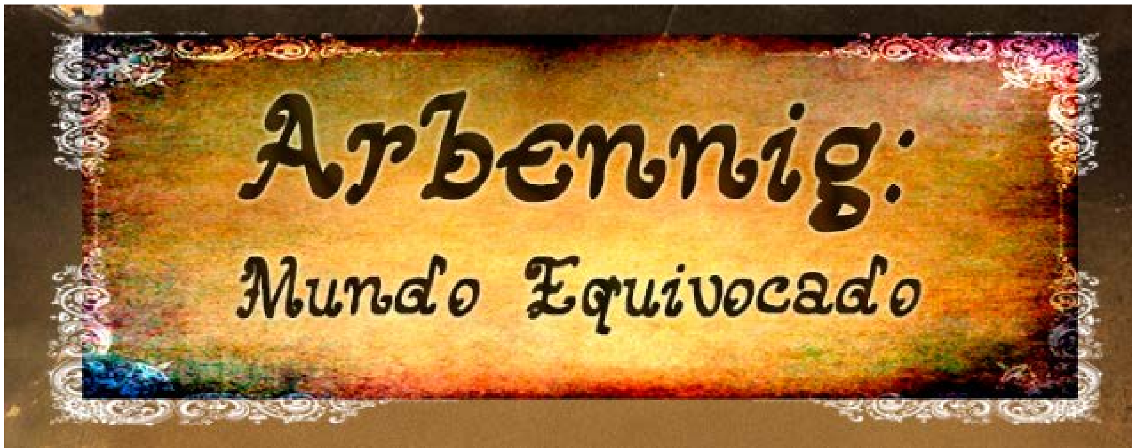


Figura 8. Título del Videojuego.

Fuente: Elaboración propia.

5.1.1.1. Resumen del argumento

Nos trasladamos a un mundo aislado y fantástico, una aislada tierra de espíritus encarnados en tótems y otros artefactos, y seres de alguna próspera civilización extinta y otra nueva. Hemos nacido de un árbol como un miembro de esta antigua civilización, y nuestra especie ya no pertenece a esta tierra.

Sin embargo, no está todo perdido, ya que la antigua civilización viajó a otro mundo y es posible reunirse con ellos. Para ello, deberá acceder a la sala de viaje interdimensional, la cual está sellada y solo es posible entrar en ella obteniendo 8 runas, que están bajo la custodia de 8 tótems repartidos en esta tierra.

Los tótems están poseídos por espíritus, y nos otorgarán la runa que guardan a cambio de que tratemos con ellos y sus necesidades.

Una vez conseguidas todas las runas, el sello de la sala final se romperá, y podremos viajar con nuestra civilización.

5.1.1.2. Conjunto de características

Las principales características atractivas del videojuego son las siguientes:

- **Atractivo** apartado visual perfectamente explorable.
- Compatible con el periférico **Oculus Rift**.
- Desarrollado con el reciente motor **Unreal Engine 4**.

5.1.1.3. Género

El género de Arbennig es el de un videojuego en primera persona de mundo abierto, con componente de puzle y habilidad. Un claro ejemplo de este género de este género corresponde a juegos como The Elder Scrolls: Skyrim.

5.1.1.4. Audiencia

Dadas las características del diseño, el juego se encuentra dentro de una clasificación por edades de tipo PEGI12, de acuerdo a la descripción ofrecida por la web oficial de PEGI sobre PEGI12[23]:

Figura 9. Logo Pegi12.

Fuente: Wikimedia[22]

> *"En esta categoría pueden incluirse los videojuegos que muestren violencia de una naturaleza algo más gráfica hacia personajes de fantasía y/o violencia no gráfica hacia personajes de aspecto humano o hacia animales reconocibles, Así como los videojuegos que muestren desnudos de naturaleza algo más gráfica. El lenguaje soez debe ser suave y no debe contener palabrotas sexuales."*

A pesar de esto, el videojuego es apto para personas de cualquier edad posterior.

5.1.1.5. Resumen del flujo de juego

A continuación se muestra un diagrama del flujo de juego básico. En él podemos apreciar cada una de las partes de las que cuenta el juego, incluyendo menús, de juego y el juego en sí mismo.

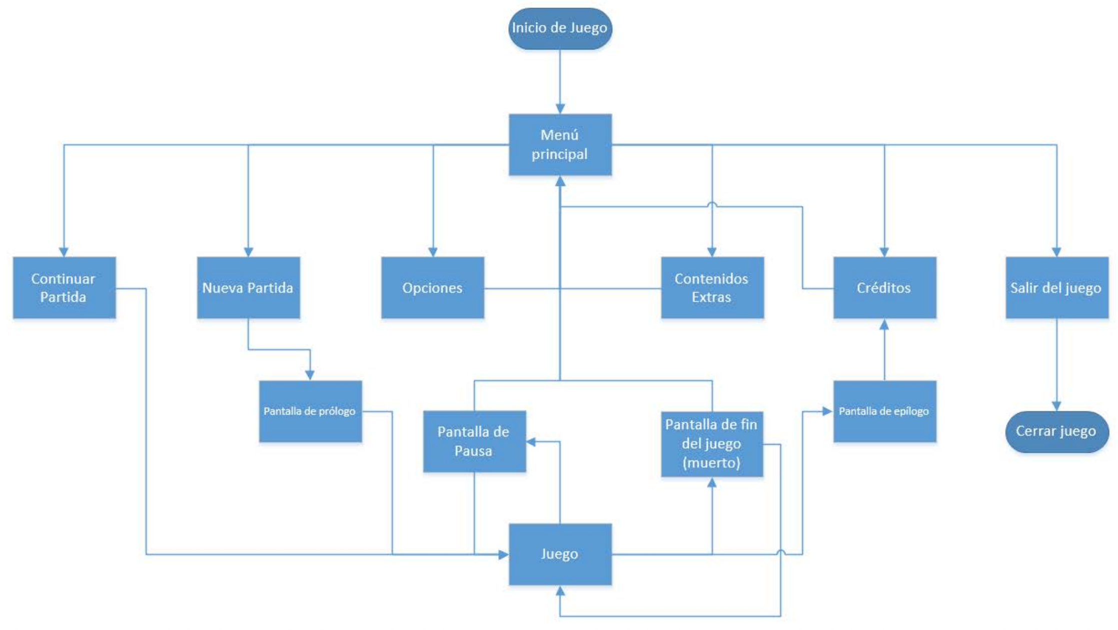


Figura 10. Flujo de pantallas del juego.

Fuente: Elaboración propia.

Este flujo de juego consiste básicamente en:

1. Iniciar aplicación de juego.
2. Se abre un menú principal con distintas opciones, entre las que podemos continuar partida (inhabilitado si todavía no tenemos ninguna), o empezar una nueva, modificar algunos parámetros de configuración, acceder a contenidos extra de la historia, a los créditos de desarrollo, y, a la opción de salir del juego, que nos sacaría de la aplicación. Seguiremos por comenzar una partida para seguir con el flujo de juego.
3. Comenzamos una nueva partida, en la que podemos superar el juego al 100%, el cual nos llevaría finalmente a una pantalla de créditos de juego a través de la sala final. O, por otro lado, podemos acceder al menú de pausa y regresar al menú principal.
4. Si por el contrario, el jugador muere, aparece un menú de fin de partida que da la opción de volver a jugar desde el último punto de guardado, o de regresar al menú principal.
5. Si superamos el juego al 100%, tras acceder a los créditos, estos nos llevan, al finalizar, hasta el menú principal.

5.1.1.6. Apariencia del juego

En cuanto a la experiencia de juego, hay dos aspectos que se pretende destacar principalmente, la libertad y la fantasía.

Por un lado, dar la apariencia de libertad al jugador a la hora de descubrir el entorno que lo rodea. La habilidad de volar sobre un mundo plenamente abierto y perfectamente explorable es el principal punto para esto. Además, el hecho de ser un videojuego en primera persona provoca una mayor inmersión y mayor sensación de libertad.

Por otro lado, la fantasía, el juego llama a un mundo fantástico, idílico, un mundo fantástico, un espacio en el que desde el comienzo del juego podrá apreciar que va a formar parte de una aventura.

La banda sonora del juego, tanto con la música, como los sonidos de ambiente, y los efectos de sonido, trata de potenciar estas sensaciones descritas anteriormente.

5.1.1.7. Ámbito

Al encontrarnos ante un videojuego de mundo abierto, dadas las características específicas que este tiene, contamos con un único gran nivel general, un espacio común sobre el que se desarrolla la aventura y los distintos desafíos. Este único mundo corresponde a la tierra de Delfrydoland.

Figura 11. Mapa de Delfrydoland desde el editor.

Fuente: Elaboración propia.

Por otro lado, respecto a los personajes que vamos a encontrar, solo habrá NPC aparte de nuestro jugador. Los NPC que encontraremos corresponden a 3 tipos:

1. **NPC de ayuda (árbol padre)**, que nos asistirán sobre controles y sobre qué objetivos tiene el jugador.
2. **NPC de aventura (tótems)**, para los que habrá que realizar alguna prueba y nos recompensen.
3. **NPC genéricos (habitantes nuevos del lugar)**, para habitar mejor la tierra de Delfrydoland, no hablan o dan algún mensaje genérico sobre el lugar.

5.1.2. Jugabilidad y mecánicas

En este apartado se describe tanto los aspectos de jugabilidad, donde se incluye progresión del jugador, misiones del juego, controles de juego, y otros, tanto como los aspectos de las mecánicas de juego, sistema de guardado y un flujo de juego en mayor detalle, entre otros.

5.1.2.1. Jugabilidad

Cabe recalcar, antes de adentrarnos en temas de jugabilidad, saber lo que ésta representa. Una clara definición es la que proporciona Wikipedia[24]:

> *"El grado en el que jugadores alcanzan metas específicas del videojuego con efectividad, eficiencia, flexibilidad seguridad y especialmente satisfacción en un contexto jugable de uso"*

Lo cual podríamos resumir en lo cómodo que es jugar a un videojuego.

5.1.2.1.1. Objetivos del juego

El jugador tiene como objetivo en el juego superar todos los desafíos que se le presentan, ya descritos anteriormente.

Una vez superados todos ellos, el juego queda completado. Dada la dinámica del juego, donde el jugador puede superar en el orden que desee los distintos desafíos que se le presentan, tanto en la escuela de vuelo, como los de los personajes, el juego no lanza un final una vez se supera el último de los desafíos, pero se desbloqueará el acceso a través de una sala especial que lleva a los créditos de juego cuando todos ellos han sido completados, y cuyos créditos simbolizan el final del juego. Esta sala especial es accesible tras conseguir todas las llaves especiales de juego.

5.1.2.1.2. Progresión

La progresión del jugador en el videojuego viene marcada a través de la obtención de las runas de los tótems.

Conforme el jugador vaya superando las misiones o retos, irá adquiriendo mayor habilidad en el videojuego, además de conocer más acerca del mundo que lo rodea y de la antigua civilización.

Además, tras lograr obtener todas las runas, podrá utilizarlas para acceder a la sala final que le permita terminar el juego.

5.1.2.1.3. Misiones y estructura de retos

En cuanto a los desafíos del juego, todos tienen en común que son planteados por los 8 tótems que hay distribuidos por el escenario.

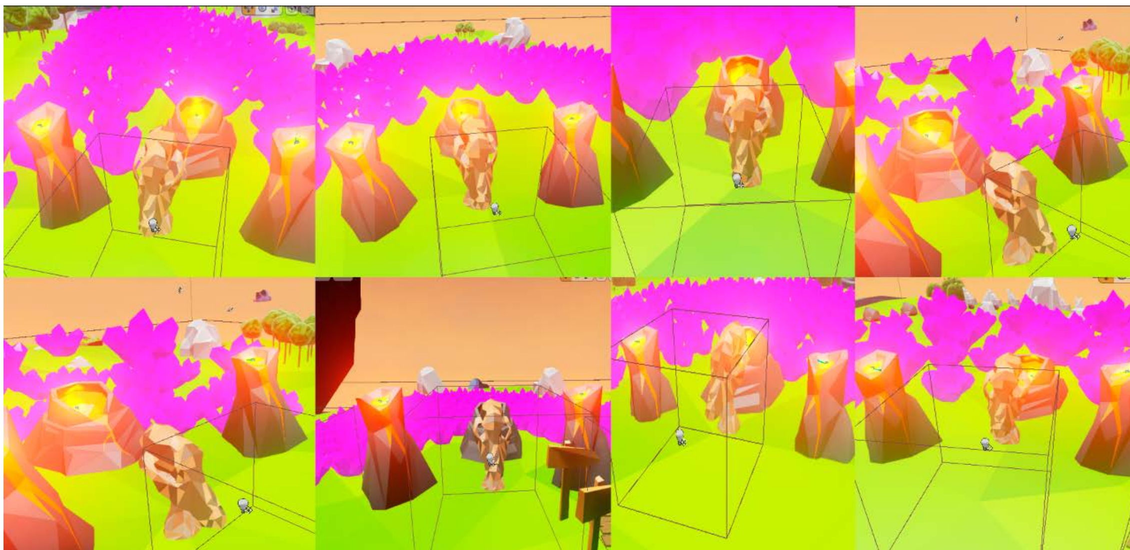


Figura 12. Capturas de los 8 tótems desde el editor.

Fuente: Elaboración propia.

Además, no es posible realizar más que un solo desafío a la vez, es decir, cuando se comienza una prueba de tótem, no es posible realizar más pruebas hasta que esta termine, o se hable con el tótem con la intención de cancelar la prueba.

El juego cuenta con los siguientes desafíos en función de cada tótem.

1. **Tótem 1. Desafío de escucha**. En este desafío solo es necesario encontrar este tótem, dado que es el primero, y hablar con él hasta que termine de contar todo lo que tiene que decir acerca del mundo de juego.

Figura 13. Captura del desafío del primer tótem.

Fuente: Elaboración propia.

2. **Tótem 2. Difusión en el pueblo.** El tótem nos ordena difundir un mensaje a 9 NPC de la isla poblada del juego. Para ello nos proporciona 9 cartas de mensaje que debemos entregarles y, tras esto, regresar al tótem para avisarlo de que hemos completado el desafío.

Figura 14. Captura del desafío del segundo tótem.

Fuente: Elaboración propia

3. **Tótem 3. Limpieza.** En este desafío, el tótem indica que su isla es la más alejada de todo el mundo, y que han aparecido malas hierbas (10 matorrales) en la isla que quiere que retiremos. Las malas hierbas las debemos colocar en un trozo de madera que nos indica, y una vez retiradas todas, se supera el desafío.

Figura 15. Captura del desafío del tercer tótem.

Fuente: Elaboración Propia.

4. **Tótem 4. Desafío de aros 1.** Se habilita en el escenario una serie de aros por los que el jugador debe pasar (por su interior) antes de que se acabe el tiempo marcado. Una vez pasado por todos los aros, el jugador supera el reto. En caso de no lograr el desafío en el tiempo indicado, el jugador deberá repetir el reto volviendo a hablar con el tótem.

Figura 16. Captura del desafío del tótem 4.

Fuente: Elaboración propia.

5. **Tótem 5. Laberinto.** En esta isla el jugador debe introducirse por un laberinto subterráneo, en el que al final del mismo se encuentra el tótem. Este desafío consiste en superar el laberinto.

Figura 17. Captura del desafío del tótem 5.

Fuente: Elaboración propia.

6. **Tótem 6. Persecución.** El tótem nos ordena correr hasta la otra punta de la isla en la que se encuentra, recoger una bandera, y regresar y hablar con él. En esta carrera, el jugador será perseguido por varios NPC de tipo Golem perseguidor, que no nos debe alcanzar si queremos superar el desafío. Además, no está permitido volar, solo correr, y en caso de volar, el jugador pierde el desafío y es teletransportado a la entrada.

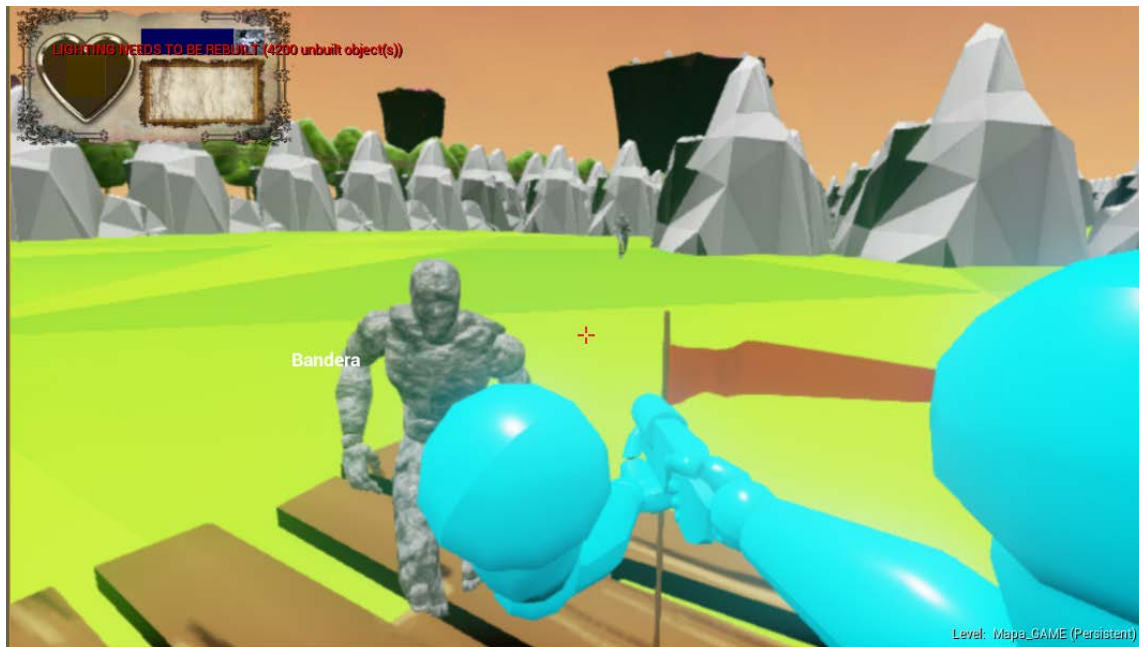


Figura 18. Captura del desafío del tótem 6.

Fuente: Elaboración propia.

7. **Tótem 7. Desafío de aros 2.** Sigue el mismo funcionamiento que el desafío de la isla del tótem 4 pero cambia los aros por otro tipo.

Figura 19. Captura del desafío del tótem 7.

Fuente: Elaboración propia.

8. **Tótem 8. Búsqueda.** El jugador debe adentrarse en el bosque que hay al sur del mundo, y encontrar una bandera ahí colocada. Debe descubrir cómo atravesar el bosque (pasando

a través del camino sin vegetación sin volar alto), porque si no pasa por el camino adecuado, es teletransportado a la entrada del bosque.

Figura 20. Captura del desafío del tótem 8.

Fuente: Elaboración propia

Respecto al último aspecto de misiones del juego, es decir, el de acceso a la sala final para completarlo, el jugador puede explorar el mundo y acceder a un terreno que contiene una edificación con 8 cerraduras. Conforme vaya consiguiendo llaves, estas irán apareciendo en su HUD, y al obtenerlas todas, podrá interactuar con las 8 cerraduras, que pasarán a tener las runas, y la puerta que bloquea el paso a la sala se abrirá.

Figura 21. Captura de la puerta final del juego abierta al colocar las runas.

Fuente: Elaboración propia.

5.1.2.1.4. Acciones del personaje

El personaje puede realizar acciones tanto en tierra como en aire. Cuenta con tres estados claramente diferenciados, entre los cuales puede alternar libremente:

Estado de tierra.

En este estado el personaje no puede volar, y puede realizar las distintas acciones:

a. Moverse en todas las direcciones que desee (pero no hacia arriba o hacia abajo).
    i. Hacia delante.
    ii. Hacia atrás.
    iii. Hacia la derecha.
    iv. Hacia la izquierda.
b. Girarse
    i. ±360º en el eje Y
    ii. ±90 en el eje X
c. Saltar.
d. Correr.
e. Interactuar con el entorno.
    i. Hablar con otros personajes.
    ii. Coger/Usar/Interactuar con elementos no personajes del escenario.
f. Activar estado de vuelo

Estado de vuelo.

En este estado, muy parecido al estado de tierra, el jugador puede realizar las siguientes acciones:

g. Moverse en todas las direcciones que desee.
    i. Hacia delante.
    ii. Hacia atrás.
    iii. Hacia la derecha.
    iv. Hacia la izquierda.
    v. Hacia arriba.
    vi. Hacia abajo.
h. Girarse

i. ±360º en el eje Y.
ii. ±90 en el eje X.

i. Interactuar con el entorno.
   i. Hablar con otros personajes.
   ii. Coger/Usar/Interactuar con elementos no personajes del escenario.

j. Activar el estado de tierra.
k. Activar el estado de vuelo rápido o Racing.
l. Frenar en seco.
m. Caer en picado.

2. **Estado de vuelo rápido o Racing.**

   En este estado, el jugador tiene una velocidad que aumenta progresivamente hasta alcanzar un máximo, que, alcanzado, se mantiene constante. Las acciones que puede realizar el jugador son las siguientes:

   a. Moverse (no puede moverse hacia atrás porque el estado Racing hace desplazarse hacia adelante al jugador con velocidad en aumento, además, dado que automáticamente se mueve hacia delante, el personaje no tiene el control del movimiento hacia delante):
      i. Hacia arriba.
      ii. Hacia abajo.
      iii. Hacia la derecha.
      iv. Hacia la izquierda.
   b. Girarse.
      i. ±90º en el eje X
      ii. ±90º en el eje Y
   c. Pasar al estado de vuelo.
   d. Activar turbo extra. Este turbo consume barra de energía del jugador, que se regenera cuando no está activo el turbo y al pasar por un aro.

5.1.2.1.5. Controles de juego

Es posible jugar utilizando teclado y ratón, y teniendo oculus rift o sin él.

El sistema detecta automáticamente al iniciar el juego si tenemos o no conectado el dispositivo oculus rift, en caso afirmativo, la cámara puede ser controlada mediante este, independientemente si estamos usando el ratón para ello.

Por otro lado, los controles de juego para teclado y ratón, incluyendo los controles opcionales del periférico Oculus Rift son:

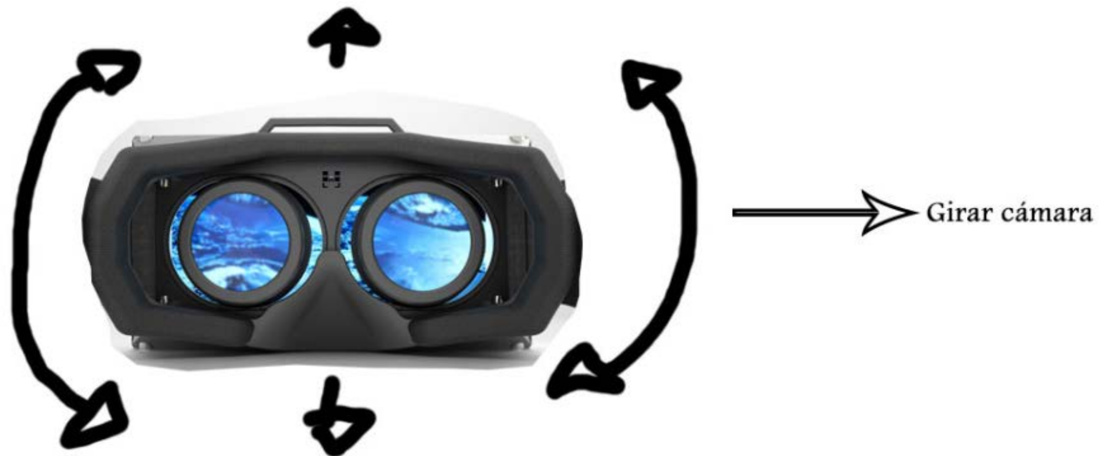


Figura 22. Uso de Oculus en el juego.

Fuente: Elaboración propia.

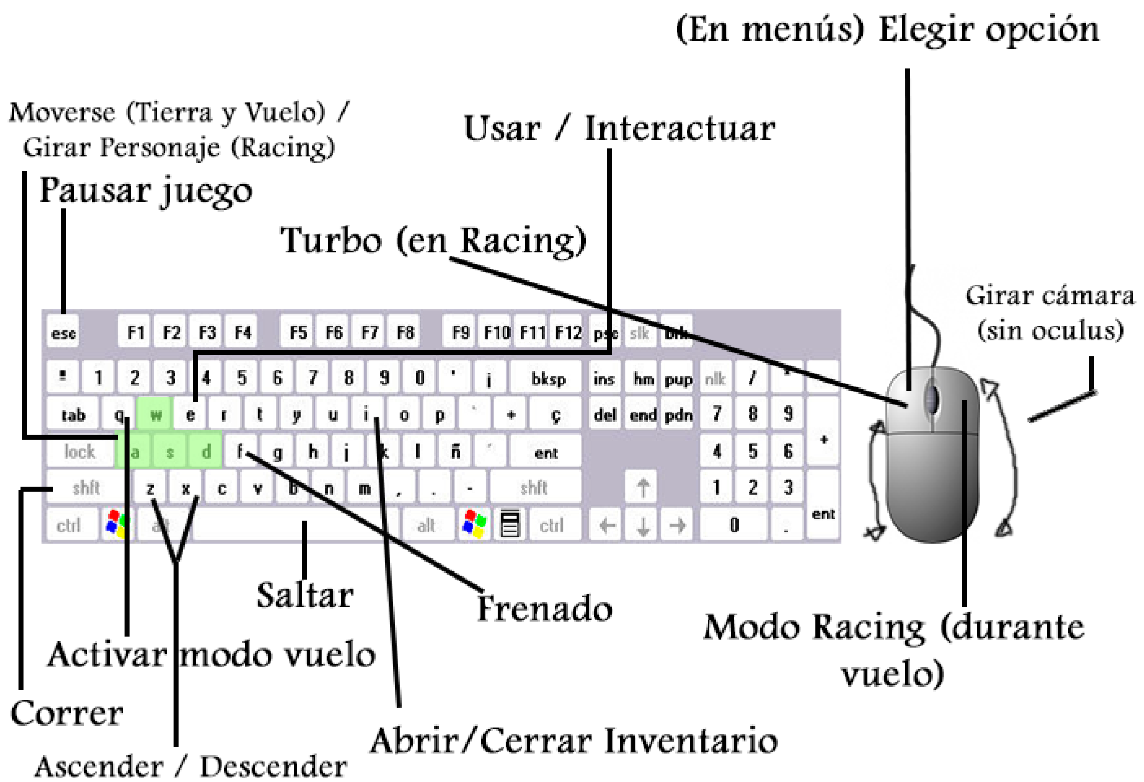


Figura 23. Controles de juego con teclado y ratón.

Fuente: Elaboración propia.

5.1.2.2. Mecánicas

Por mecánicas de juego entendemos algo con lo que el jugador interactúa con el fin de crear o contribuir en la jugabilidad, como por ejemplo, plataformas que se mueven, cuerdas para balancearse, o hielo resbaladizo. En este apartado, además de mecánicas de juego, incluiremos peligros (mecánica que puede herir o matar al jugador, pero que no posee inteligencia, tales como plataformas dañinas), power-ups (un ítem que es recogido por el jugador para ayudarlo con su jugabilidad, como un turbo), y coleccionables (elementos que son recogidos por el jugador, pero no tienen un impacto inmediato en la jugabilidad, como pueden ser las llaves del juego).

Dicho esto, en Arbennig, las mecánicas que encontramos son las siguientes:

Mecánicas

Teletransportador.

Figura 24. Teletransportador y destino de teletransporte pintados con líneas de debug, respectivamente, de izquierda a derecha.

Si el jugador entra en contacto con ese componente, es teletransportado a otra zona del mapa.

Puerta que se abre.

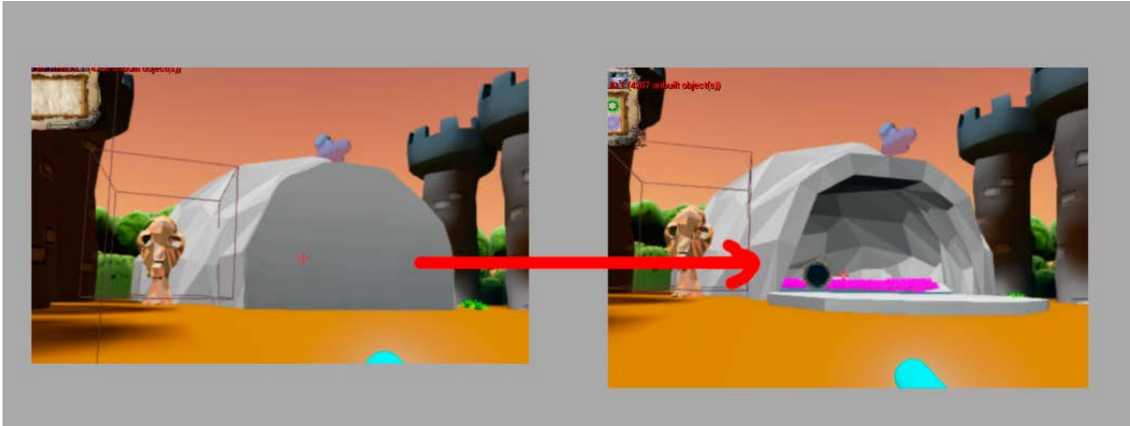


Figura 25. Capturas de puerta cerrada y abierta.

Fuente: Elaboración propia.

Puertas que al cumplir con ciertas condiciones o simplemente interactuar con ellas, se abren.

<u>Puerta del final del juego.</u>

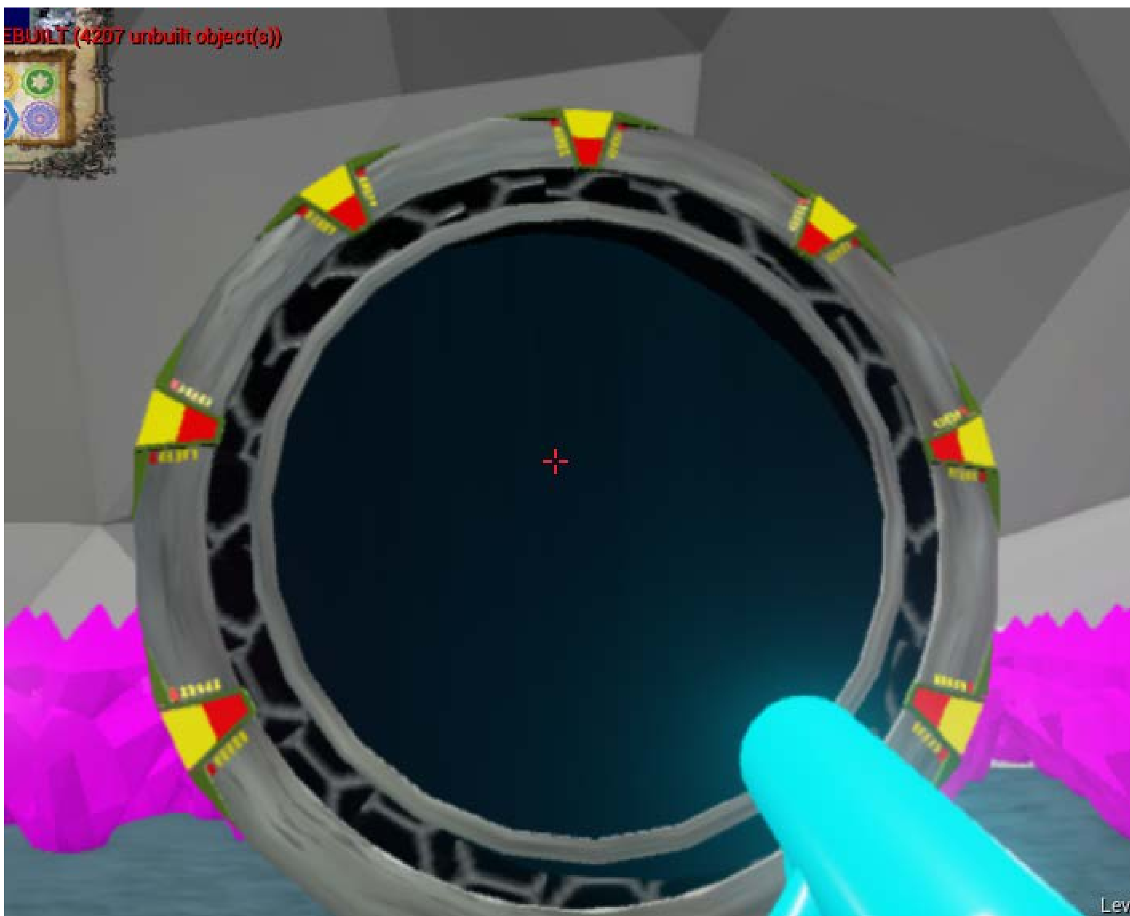


Figura 26. Captura de Puerta Final.

Fuente: Elaboración propia.

Esta puerta nos permite terminar el juego una vez interactuemos con ella o la atravesemos.

Peligros

Plataformas/Sustancia dañina.

Figura 27. Captura de varias plataformas dañinas (morado)

Fuente: Elaboración propia.

Componentes que si el jugador los toca, pierde vida. Es posible encontrarlos en los retos de aros.
Plataforma de muerte instantánea.

Figura 28. Captura del mar del juego (plataforma de muerte instantánea)

Fuente: Elaboración propia.

El jugador muere al entrar en contacto con ella.

Power Ups

<u>Aro.</u>

Figura 29. Captura de los dos tipos de aros en el juego.

Fuente: Elaboración propia.

Proporciona al jugador una recarga instantánea de su energía para turbos.

<u>Punto de guardado.</u>

Figura 30. Captura de un punto de guardado.

Fuente: Elaboración propia.

Proporciona al jugador una recarga de su salud y de la energía para turbos.

Coleccionables

Runas.

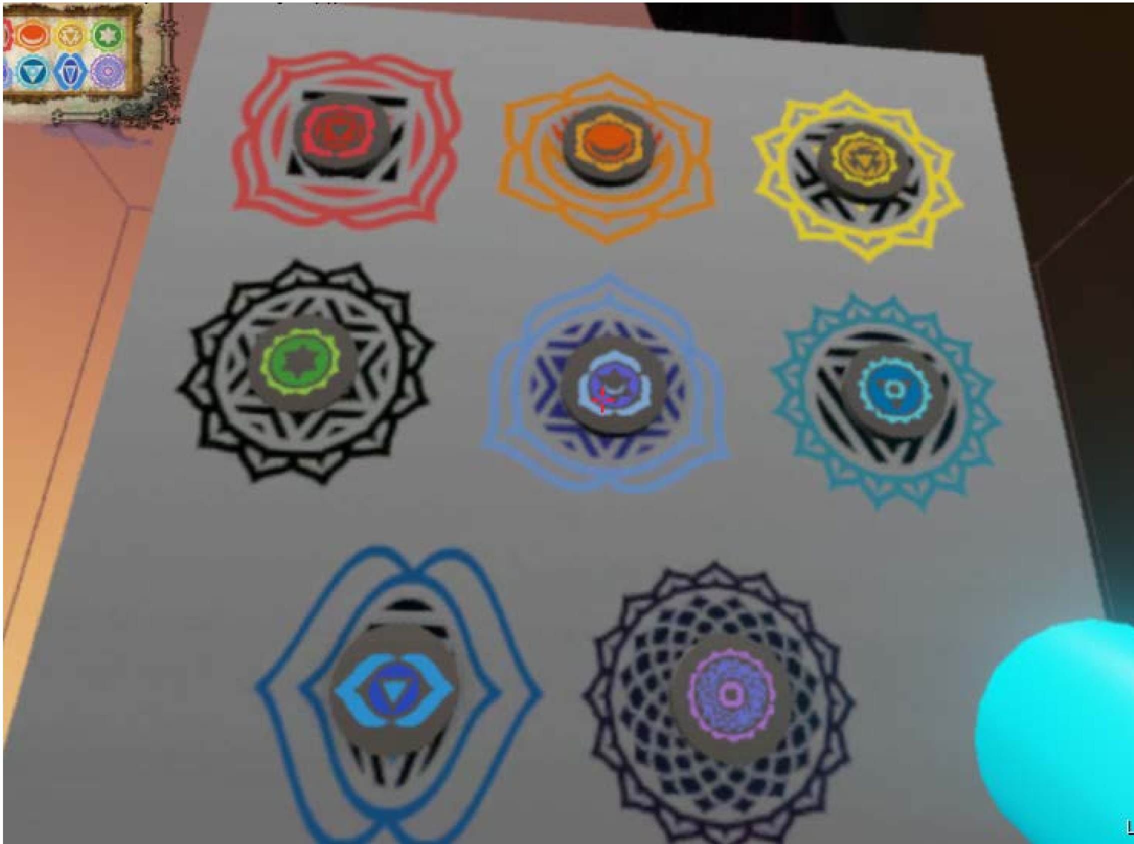


Figura 31. Captura de las 8 runas colocadas en el muro final.

Fuente: Elaboración propia.

Cuando el jugador supera las pruebas y desafíos que se le imponen, consigue unas llaves especiales que le permiten acceder a una sala final para terminar el juego. Puede encontrar también en el mundo más llaves.

Elementos de tótem de isla 2. Folletos.

Figura 32. Captura de folleto informativo.

Fuente: Elaboración propia.

El jugador los puede recoger para entregarlos posteriormente a través de su inventario.

<u>Elementos de tótem de isla 3. Matorrales.</u>

Figura 33. Captura de matorral.

Fuente: Elaboración propia.

El jugador los puede recoger para dejarlos en otra parte a través de su inventario.

5.1.2.3. Opciones de juego

Figura 34. Captura de las opciones de juego.

Fuente: Elaboración propia.

Las opciones de juego, solo accesibles a través del menú principal, permiten modificar parámetros de vídeo y audio.

Estos parámetros son:

- Modificar la resolución de juego, dando a elegir entre varias opciones de formato 4:3 y formato 16:9.
- Modificar el volumen general de juego.

5.1.2.4. Rejugar y salvar

Para guardar la partida, es posible acceder a distintos puntos de guardado, distribuidos por el escenario.

Figura 35. Captura de un punto de guardado.

Fuente: Elaboración propia.

Estos puntos de guardado almacenan los progresos del jugador (tótems superados), y deja al jugador en la posición donde se encuentra el punto de guardado.

Cuando se inicia el juego desde el escritorio, se carga automáticamente la partida de juego si la hay, mostrando bloqueada o no la opción de continuar partida en el menú principal (en función de si existe o no una partida guardada).

Figura 36. Captura del menú principal con la opción de continuar bloqueada (no hay partida guardada).

Fuente: Elaboración propia.

Además de esto, al cargar el nivel de extras del juego, contiene mayor o menor cantidad de extras en función del progreso obtenido por el jugador.

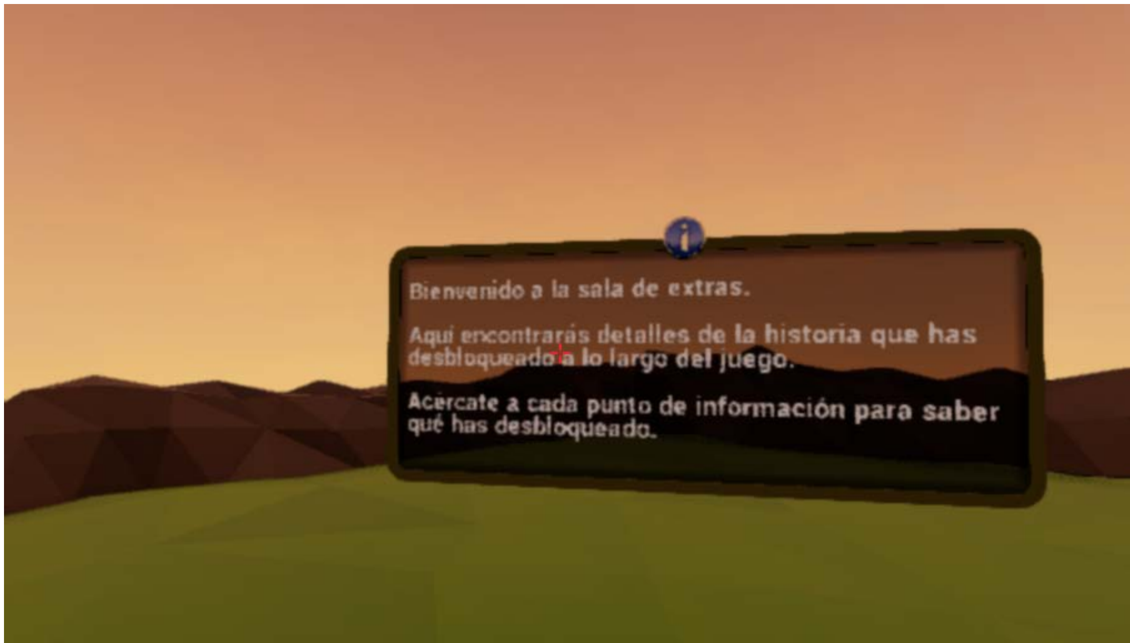


Figura 37. Captura del nivel de extras.

Fuente: Elaboración propia.

En caso de seleccionar la opción de empezar nueva partida existiendo una partida guardada ya, la partida guardada se perdería en cuanto el jugador guardara en la nueva partida, sobrescribiendo la anterior. Se avisa al jugador de este paso, por si ha hecho clic involuntariamente.

Figura 38. Captura del menú de confirmación de nueva partida.

Fuente: Elaboración propia.

Por otro lado, si el jugador dentro del juego, muere, es enviado a una pantalla de fin del juego, que le da la opción de volver a jugar desde el último punto guardado. Perdiendo todos los progresos conseguidos antes de guardar la partida.

Figura 39. Captura del menú de fin de partida.

Fuente: Elaboración propia.

5.1.3. Historia, características, y personajes

5.1.3.1. Historia

La tierra de Delfrydoland es un lugar misterioso. Aislada del tiempo y del espacio con el mundo exterior, vive una sociedad que no mucho conoce de su pasado, presente, o futuro. Una sociedad de seres creados por un árbol de la vida que mantiene un equilibro natural, dando origen a través de sus poderes a criaturas.

Esta sociedad de seres con aspecto de anciano, tiene toda el mismo aspecto, además, no comen, ni duermen, no realizan actividad alguna, no envejecen, y ni siquiera tienen nombre u ocupan su tiempo libre. Suelen vivir en un poblado que se encuentra en una isla donde simplemente están esperando a que ocurra algún suceso, que probablemente no será debido a ellas. Es más, el poblado ni siquiera ha sido construido por ellas, aunque eso no les importa.

En este ambiente, donde nadie hacía muchas preguntas acerca de las cosas, el árbol de la vida crea a un miembro de la civilización anterior a la actual de ancianos en Delfrydoland, crea a Arbennig.

Arbennig pertenece a la civilización de los Barod que decidió marcharse del mundo de Delfrydoland para buscar nuevas aventuras en otros mundos. Los Barod fueron los creadores del árbol de la vida, y programaron la creación de un miembro de su especie siglos después de su marcha para que pudiera informarlos acerca del mundo que dejaron atrás, pero para poder informarlos, Arbennig debería demostrar que es capaz de viajar a otros mundos a vivir aventuras con ellos. Esto es, que para poder cambiar de mundo, dejaron un teletransportador de mundos en el interior de una sala de piedra sellada por 8 runas mágicas, que estarían bajo la custodia de 8 tótems que la entregarían a Arbennig a cambio de realizar algún desafío.

Gracias a los tótems y algunas hadas, Arbennig logra averiguar que los Barod, una especie muy próspera, inteligente, y con un gran afán por descubrir nuevos mundos, acabaron cansados del maravilloso pero limitado mundo de Delfrydoland cuando ya no había nada, a su parecer, que pudieran descubrir o lograr en él, y crearon una tecnología especial que les permitió viajar a otro mundo mayor para explorar.

Y finalmente, cuando Arbennig logra reunir todas las llaves y accede a esta máquina transportadora, toma la decisión de regresar con su especie y vivir muchas nuevas aventuras.

5.1.3.2. Mundo de juego

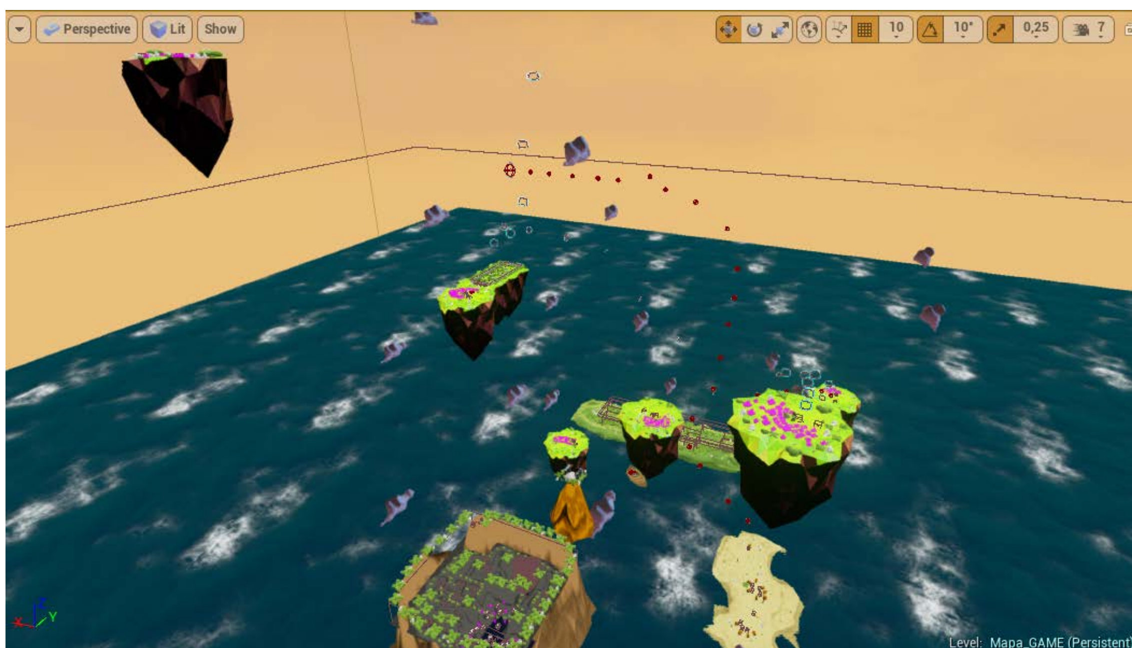


Figura 40. Captura del mundo de juego desde el editor.

Fuente: Elaboración propia.

El mundo de juego es un lugar variado en mitad de un océano que presenta tanto islas en este, como islotes flotantes en el cielo. En total encontramos 12 formaciones, 5 islas en el mar, y 7 islas flotantes.

5.1.3.2.1. Isla de inicio.

Figura 41. Captura de la isla de inicio desde el editor.

Fuente: Elaboración propia.

En esta isla es donde nace el jugador, aquí encontramos el árbol de la vida y una serie ítems que nos enseñan cómo jugar.

5.1.3.2.2. Isla de Tótem 1.

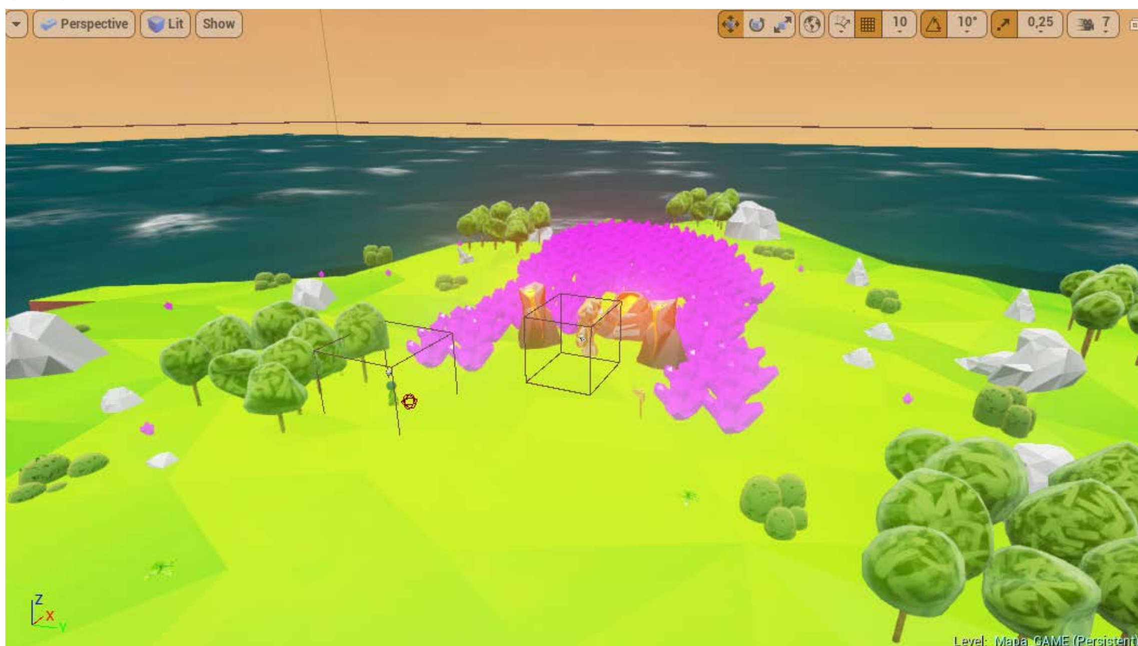


Figura 42. Captura de la isla de tótem 1 desde el editor.

Fuente: Elaboración propia.

La isla del tótem 1 es la más próxima a la isla de inicio, se trata de una isla flotante.

5.1.3.2.3. Isla de Tótem 2.

Figura 43. Captura de la isla de tótem 2 desde el editor.

Fuente: Elaboración propia.

Al igual que la isla del tótem 2, nos encontramos ante otra isla flotante, donde encontramos el tótem del desafío que quiere que avisemos que va a desaparecer cuando nos entregue la runa que protege.

5.1.3.2.4. Isla de Tótem 3.

Figura 44. Captura de la isla de tótem 3 desde el editor.

Fuente: Elaboración propia.

La isla del tótem 3 es la más alejada de todo el juego, que se encuentra en un extremo y a mucha altitud.

5.1.3.2.5. Isla de Tótem 4.

Figura 45. Captura de la isla de tótem 4 desde el editor.

Fuente: Elaboración propia.

En esta isla flotante encontramos al tótem que nos envía a realizar el primer desafío de vuelo.

5.1.3.2.6. Isla de Tótem 5.

Figura 46. Captura de la isla de tótem 5 desde el editor.

Fuente: Elaboración propia.

La isla de tótem 5 es la que forma un laberinto creado por los Barod, y está llena de teletransportadores que nos llevan a la salida, mientras que el tótem se encuentra al final del teletransportador.

5.1.3.2.7. Isla de Tótem 6.

Figura 47. Captura de la isla de tótem 6 desde el editor.

Fuente: Elaboración propia.

La isla del tótem 6 es la más grande de los islotes flotantes, ya que son 3 islotes juntos, donde se encuentra una prueba de velocidad y capacidad de esquivar a unos personajes de piedra del desafío.

5.1.3.2.8. Isla de Tótem 7.

Figura 48. Captura de la isla de tótem 7 desde el editor.

Fuente: Elaboración propia.

La isla 7 contiene otro desafío de aros, más complejo que el de la isla 4.

5.1.3.2.9. Isla de Tótem 8.

Figura 49. Captura de la isla de tótem 8 desde el editor.

Fuente: Elaboración propia.

La isla 8 es donde se encuentra el tótem que nos envía a buscar una bandera en la isla del bosque.

5.1.3.2.10. Isla del poblado.

Figura 50. Captura de la isla del poblado desde el editor.

Fuente: Elaboración propia.

En esta isla encontramos a la nueva civilización creada por el árbol. Las casas son casas de Barod, no tienen puertas porque estos entraban utilizando teletransportadores.

5.1.3.2.11. Isla del bosque.

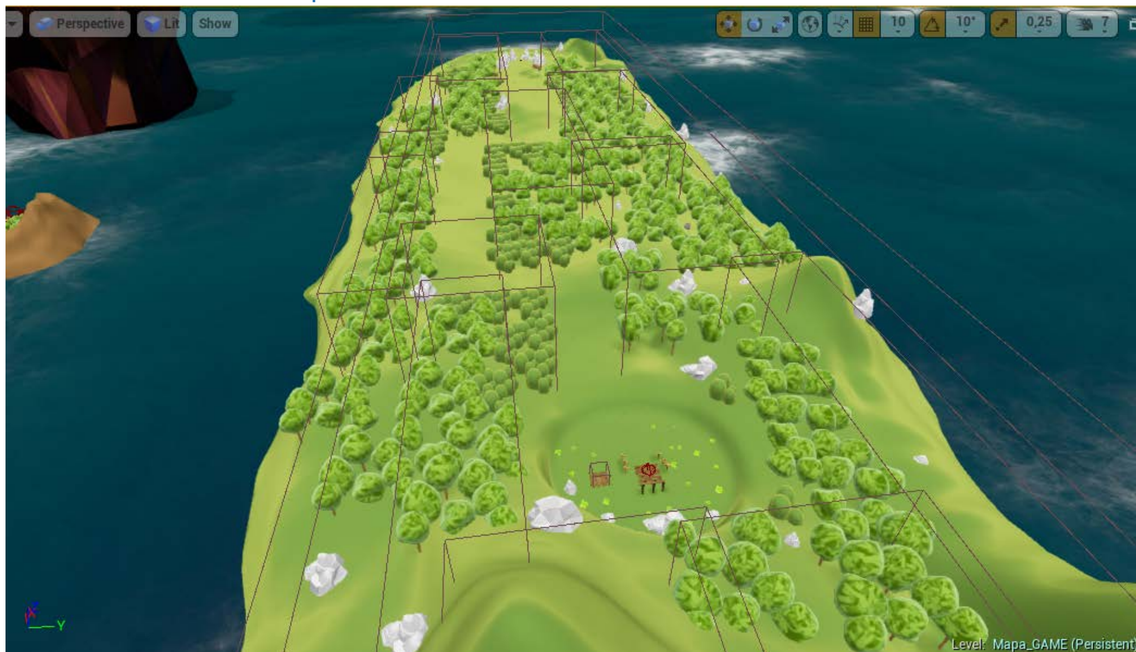


Figura 51. Captura de la isla del bosque desde el editor.

Fuente: Elaboración propia.

La isla del bosque es un lugar lleno de teletransportadores, que para atravesar es necesario seguir un camino marcado en el suelo del mismo, y que deberemos seguir si queremos superar el desafío del tótem de la isla 8.

5.1.3.2.12. Isla final.

Figura 52. Captura de la isla final desde el editor.

Fuente: Elaboración propia.

Esta isla final es donde encontramos la sala de piedra que debe abrirse una vez contamos con todas las runas.

Figura 53. Captura del interior de la sala final desde el editor.

Fuente: Elaboración propia.

En el interior de la sala se encuentra el teletransportador.

5.1.3.3. Personaje Principal. Arbennig.

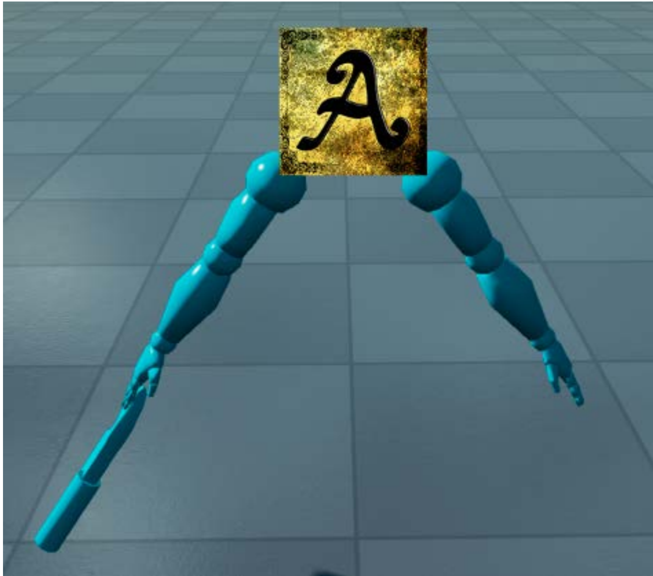


Figura 54. Personaje Arbennig en primera persona.

Fuente: Elaboración propia.

Arbennig es el Barod creado por el árbol de la vida. Debe tratar con los 8 tótems del mundo para poder obtener las runas que le permitirán viajar con su civilización.

Su nacimiento estaba programado por los Barod para siglos después de haberse marchado, de forma que pudiera informar acerca del mundo de Delfrydoland.

Es Arbennig el personaje que puede realizar las acciones de personaje (5.2.1.4), y también quien posee vida y energía para turbos volando.

5.1.3.3.1. Personaje NPC 1. Árbol de la vida.

Figura 55. Captura del árbol de la vida.

Fuente: Elaboración propia.

El árbol de la vida es una de las mayores invenciones de los Barod. Puede crear seres vivientes y comunicarse telepáticamente con cualquier ser del mundo de Delfrydoland. Es el creador de Arbennig, y también de la nueva civilización con aspecto de anciano.

5.1.3.3.2. Personaje NPC 2. Tótems.

Figura 56. Captura de un tótem desde el juego en modo debug.

Fuente: Elaboración propia.

Han sido creados por los Barod para guardar las runas que permitirán a Arbennig acceder a la sala final de teletransporte a otro mundo. Son todos iguales, aunque cada personaje tiene su propia personalidad.

Cuando entregan al runa a Arbennig, su alma deja de estar ligada al tótem, de modo que dejan de ser capaces de comunicarse con el mundo.

Existe a pesar de esto, un tótem que nunca abandona su cuerpo ni tiene una runa que entregar, es el tótem que se encuentra en la sala final, situado ahí para explicar a cualquiera que se acerque cómo puede entrar en la sala sellada.

5.1.3.3.3. Personaje NPC 3. Seres con aspecto de ancianos.

Figura 57. Captura de los seres con aspecto de anciano.

Fuente: Elaboración propia (malla gratuita del pack mixamo de unreal).

Es la nueva civilización que habita Delfrydoland, no tienen mucho que decir más que un par oraciones a aquel que quiera hablar con ellos, ni tampoco tienen mucho que hacer. Además todos cuentan con el mismo aspecto.

5.1.4. Nivel de juego – Delfrydoland.

Figura 58. Caotura del nivel juego.

Fuente: Elaboración propia.

5.1.4.1. Resumen

Este mundo es el único del videojuego. Se trata de un mundo abierto que el jugador puede explorar y también realizar 8 misiones que le permitirán acceder a la sala final donde termina el juego. Aquí es donde encontramos todos los personajes y lugares descritos anteriormente, y todas las mecánicas y misiones.

5.1.4.2. Material de introducción

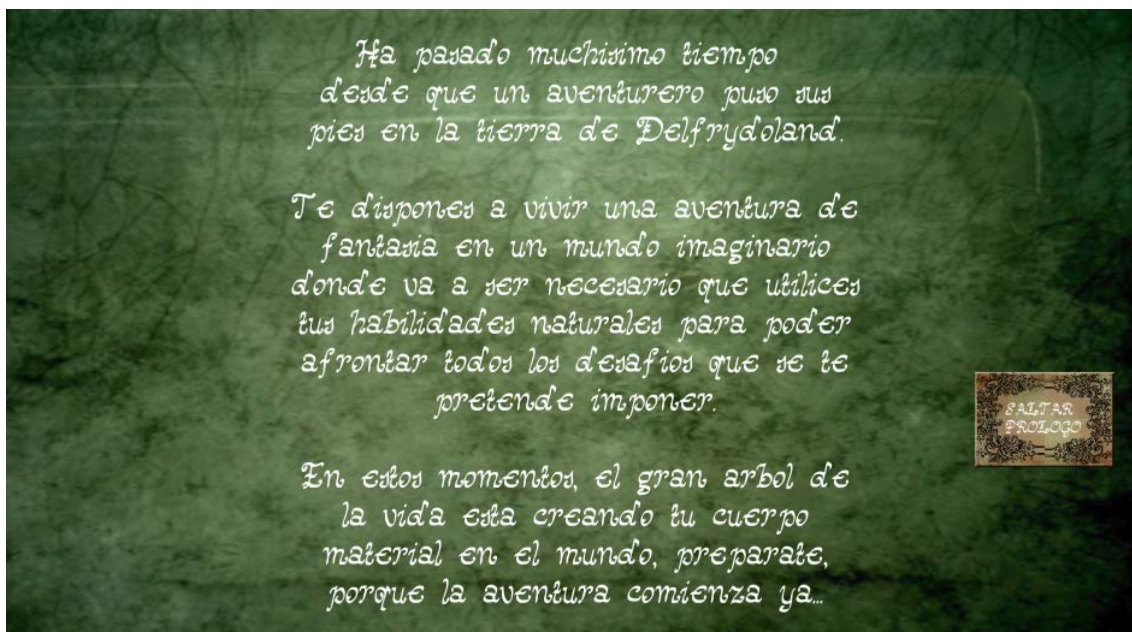


Figura 59. Captura prólogo.

Fuente: Elaboración propia.

Antes de acceder al nivel como tal, aparece una pantalla de prólogo que nos permite conocer un poco de la historia del mismo. Este prólogo es posible saltarlo con un botón.

Además de esto, una vez se comienza la partida, el jugador se encuentra en una isla inicial llena de mensajes de ayuda que le permitirán conocer los controles de juego, y más detalles sobre el mundo.

5.1.4.3. Objetivos

Los objetivos a realizar en el mundo de Delfrydoland son todos los descritos en el apartado de misiones y estructura de retos (5.1.2.1.3).

5.1.4.4. Mapa

Figura 60. Mapa del juego.

Fuente: Elaboración propia

5.1.4.5. Caminos

El jugador, dado que puede realizar acciones de vuelo, y dado que se encuentra en un mundo abierto, puede realizar cualquiera de las misiones del juego en el orden que desee. Es decir, desde

la isla de inicio donde aparece, puede dirigirse libremente siguiendo el camino que desee hasta cualquier isla con tótem para realizar el desafío que este le indique.

5.1.4.6. Encuentros

Explorando el mundo, es posible encontrarse con los distintos personajes descritos en el apartado de personajes (5.3.3). Es decir, podrá encontrarse con los 8 tótems de desafío o el tótem de la isla final, con el árbol de la vida, y con los seres con aspecto de anciano que se encuentran casi todos en el poblado, a excepción de uno, que está en la isla del laberinto. Cada personaje mencionado tiene algo que decir al jugador, proporcionándole información.

5.1.4.7. Guía del nivel

Como tal, no existe una guía del nivel, como se ha mencionado en el apartado de Caminos (5.4.5), el jugador puede realizar cualquiera de los desafíos en el orden deseado, cuya lógica encontramos descrita en el apartado de misiones y estructura de retos (5.2.1.3).

Una vez que el jugador realice cada uno de los desafíos, puede acceder a la sala final, interactuando con un pilar donde se colocan las runas, que abrirá la puerta de acceso a la sala final del juego, con un teletransportador que al atravesarlo o interactuar con él, hace que el juego termine.

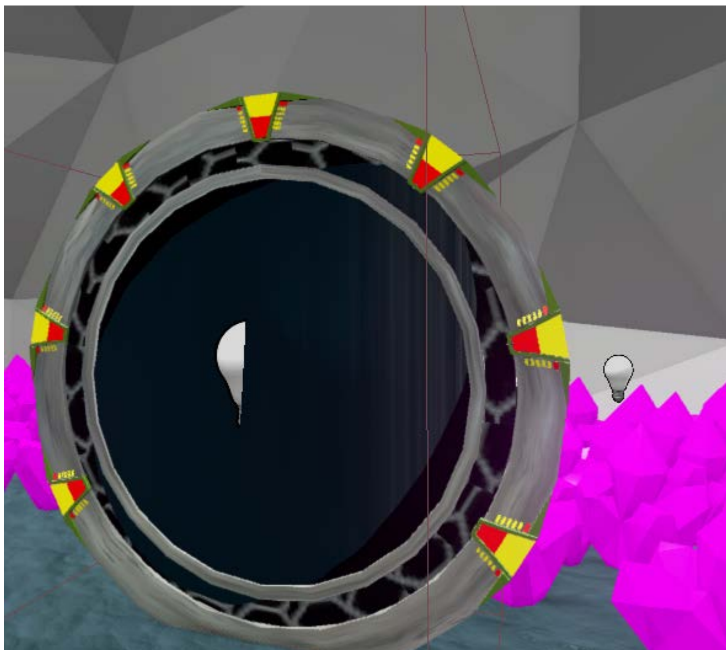


Figura 61. Captura de la puerta de final del juego desde el editor.

Fuente: Elaboración propia.

5.1.4.8. Material de cierre

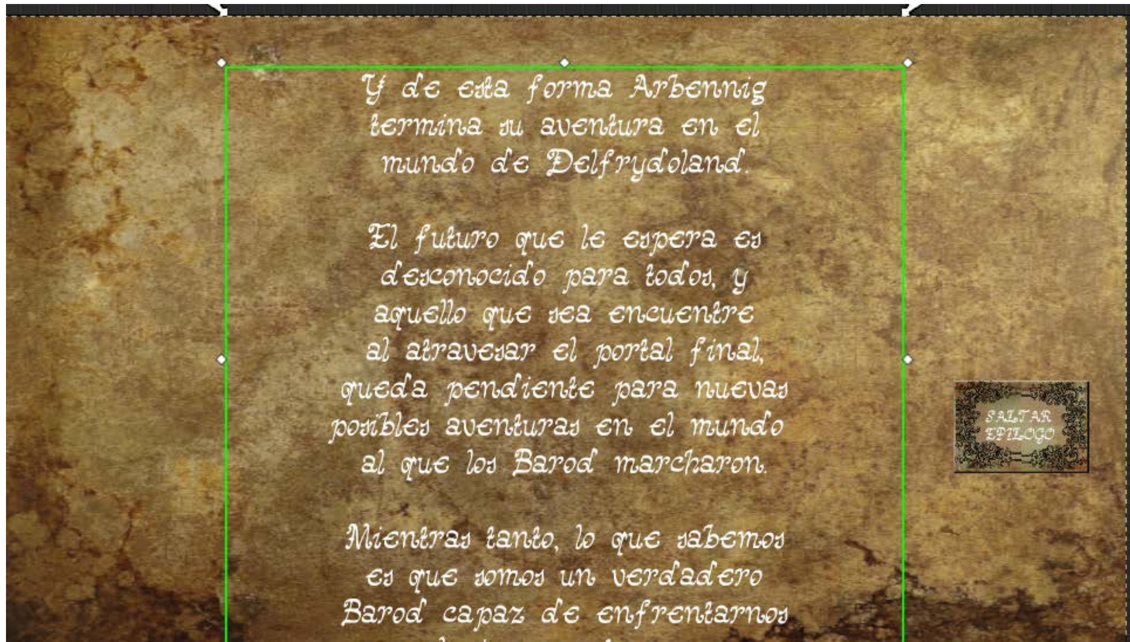


Figura 62. Captura del epílogo desde el editor.

Fuente: Elaboración propia.

Cuando el jugador toca o interactúa con el teletransportador, se carga una pantalla con un texto animado, que contiene en epílogo del juego. El epílogo cuenta al jugador que ha superado con éxito el nivel y que es aventurarse en nuevos mundos.

Cuando finaliza el epílogo, que es posible saltar si no se quiere ver, se pasa a la pantalla de créditos, que sigue el mismo funcionamiento del epílogo y la introducción. Y, finalmente, al terminar los créditos o saltarlos, se accede al menú principal.

5.1.5. Interfaz

5.1.5.1. HUD

El HUD (heads-up display), recordemos como se ha mencionado brevemente en el apartado de estado del arte, es información 2D que aparece dibujada sobre la pantalla de juego, mostrando información útil al jugador.

Figura 63. Captura del HUD permanente en el nivel de juego desde el editor.

Fuente: Elaboración propia.

En Arbennig, mostramos en el HUD de forma permanente una vez se empieza la partida:

- La vida del personaje.
- La energía del personaje.
- Las runas conseguidas y no conseguidas.

Por otra parte, en el HUD se muestra el inventario de juego cuando el jugador lo activa, que a su vez, posee un submenú de acción que aparece cuando seleccionamos algún objeto del inventario.

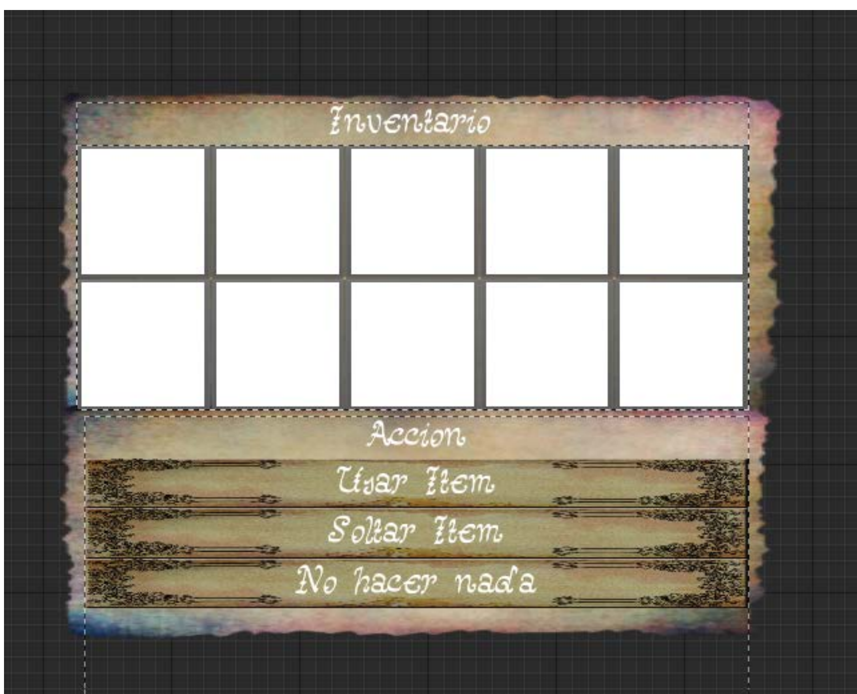


Figura 64. Captura del Inventario desde el editor.

Fuente: Elaboración propia.

También mostramos información sobre los aros recogidos y tiempo restante cuando realiza las pruebas de desafío de aros.

Figura 65. Captura del temporizador y contador de aros desde el editor.

Fuente: Elaboración propia.

Y haremos aparecer la notificación de si el jugador ha superado o no algún desafío de tótems o ha guardado una partida, o si puede usar algún objeto o no a través del HUD.

Figura 66. Otras notificaciones desde el HUD.

Fuente: Elaboración propia.

5.1.5.2. Menús

Los menús nos permitirán acceder al nivel de juego, modificar aspectos del juego (resolución y volumen general), acceder a niveles de extras, salir del juego, pausarlo, o acceder a los créditos, entre otras cosas.

A continuación vamos a explicar cada uno de estos menús y pantallas, para los cuales nos vamos a apoyar en la figura de flujo de juego del apartado de Resumen del flujo de juego (5.1.1.5).

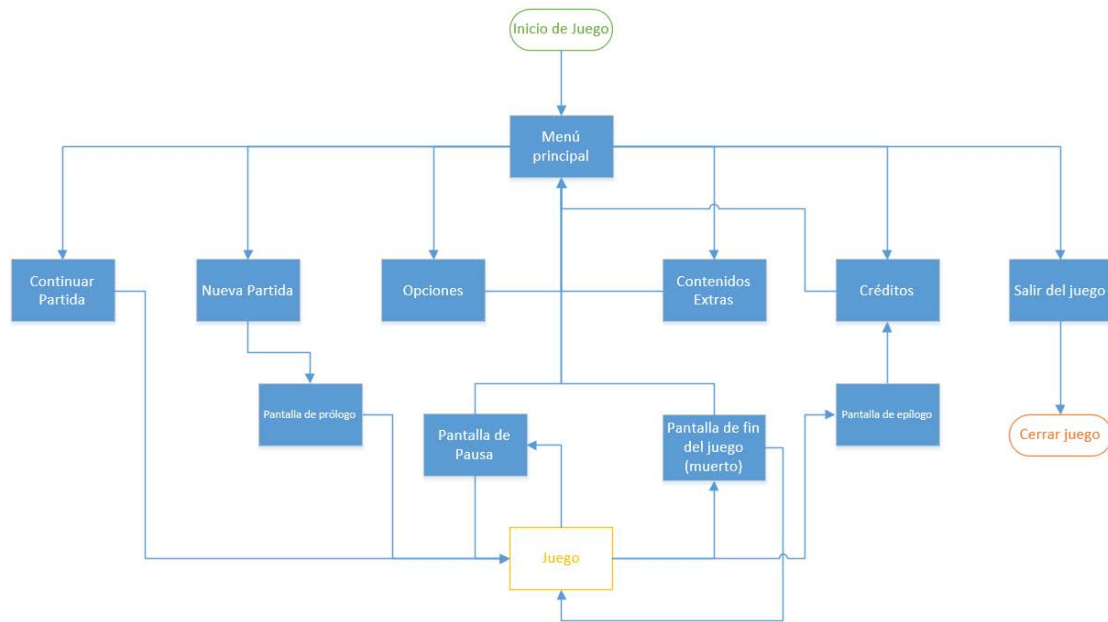


Figura 67. Diagrama de menús del juego.

Fuente: Elaboración propia.

5.1.5.2.1. Menú Principal.

Figura 68. Captura del menú principal desde el editor.

Fuente: Elaboración propia

Este menú es el que se abre al iniciar el juego. En él encontramos la opción de empezar o cargar (si tenemos datos guardados) la partida, como también el acceso al menú de opciones, al nivel de créditos, y al de extras. Es a través de este menú que podemos salir del juego, desde la opción de salir.

5.1.5.2.2. Menú de opciones.

Figura 69. Captura del menú de opciones desde el editor.

Fuente: Elaboración propia

Nos permite modificar la resolución de pantalla, dando a elegir entre tres formatos 4:3, que son 640x480, 800x600, y 1024x768, y otros tres formatos en 16:9, que son 1280x720 (la opción que aparece por defecto), 1366x768, y 1920x1080. Cuando una resolución está establecida, el botón para seleccionarla se deshabilita.

Este menú también permite modificar el volumen general del juego a través de un slider, y por último, regresar al menú principal.

5.1.5.2.3. Menú de Pausa.

Figura 70. Captura del menú de pausa desde el editor.

Fuente: Elaboración propia

El menú de pausa, que aparece cuando el jugador pausa el juego (obvio), nos da la opción de volver a la partida o de regresar al menú principal.

5.1.5.2.4. Menú de confirmación de nueva partida

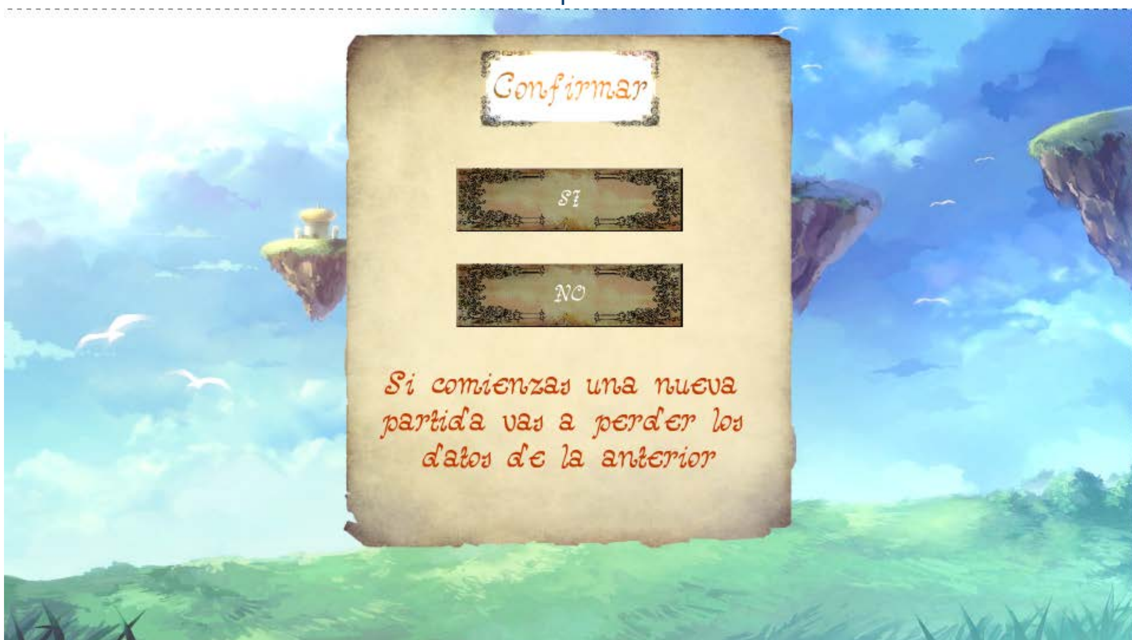


Figura 71. Captura del menú de confirmación de nueva partida desde el editor.

Fuente: Elaboración propia

Este menú salta cuando hacemos clic en la opción del menú principal de nueva partida, existiendo una partida existente guardada ya. Desde este menú podemos volver al menú principal, o confirmar que queremos que se borre la partida y empezar una nueva.

5.1.5.2.5. Menú de partida terminada (cuando jugador muere)

Figura 72. Menú de fin de partida tras morir el jugador.

Fuente: Elaboración propia

Cuando el jugador toque la plataforma de muerte (el agua del océano en el juego), o por chocar con varias plataformas dañinas se quede sin salud, se carga este menú.

Aquí se nos da la opción de empezar en el último punto de guardado, o de volver al menú principal.

Pantalla de prólogo.

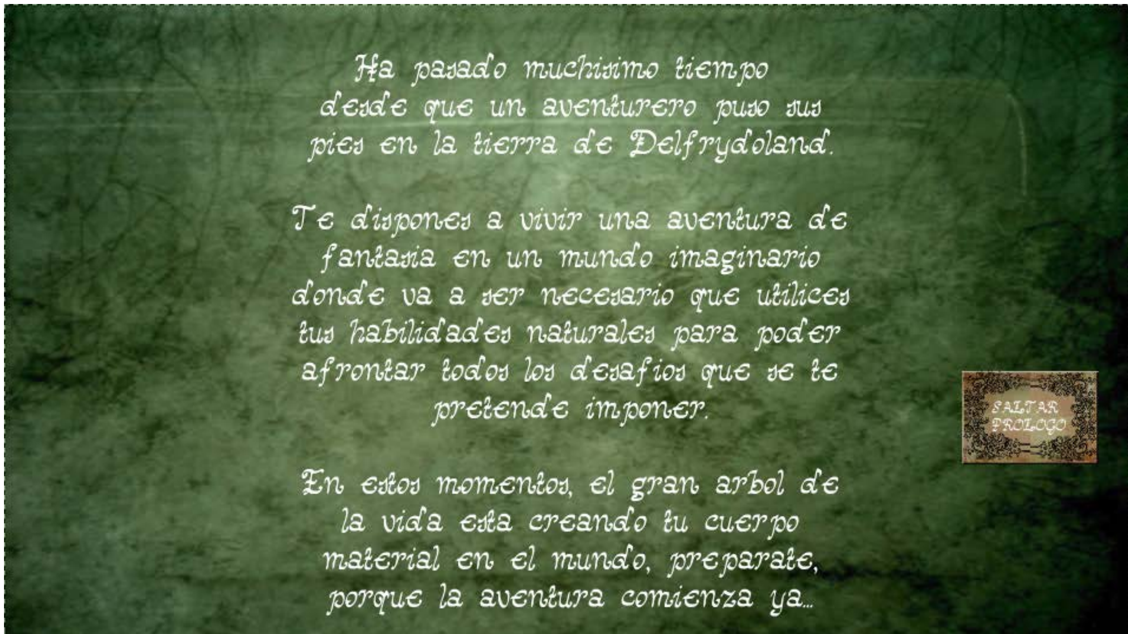


Figura 73. Captura de la pantalla de prólogo desde el editor.

Fuente: Elaboración propia

Esta pantalla funciona también como un menú en sí misma. Se trata de una pantalla temporal con un texto animado subiendo hacia arriba, cuando el texto termina de subir, se carga la partida de juego. Ofrece la posibilidad también de saltar el prólogo con un botón.

Pantalla de epílogo.

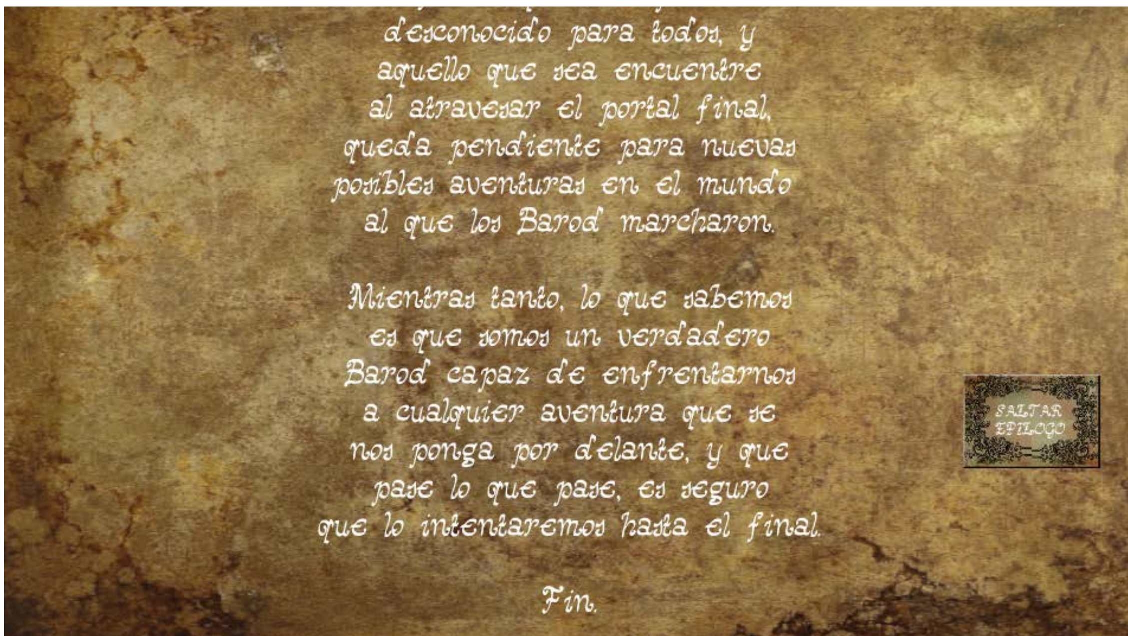


Figura 74. Captura de la pantalla de prólogo desde el editor.

Fuente: Elaboración propia

Su funcionamiento es el mismo que con la pantalla de prólogo, pero con la diferencia de que a través de esta pantalla no accedemos a la partida, sino a los créditos de juego.

Pantalla de créditos.

Figura 75. Captura de la pantalla de prólogo desde el editor.

Fuente: Elaboración propia

Al igual que con el prólogo y el epílogo, los créditos tienen el mismo funcionamiento, pero estos apuntan al menú principal.

5.1.5.3. Cámara

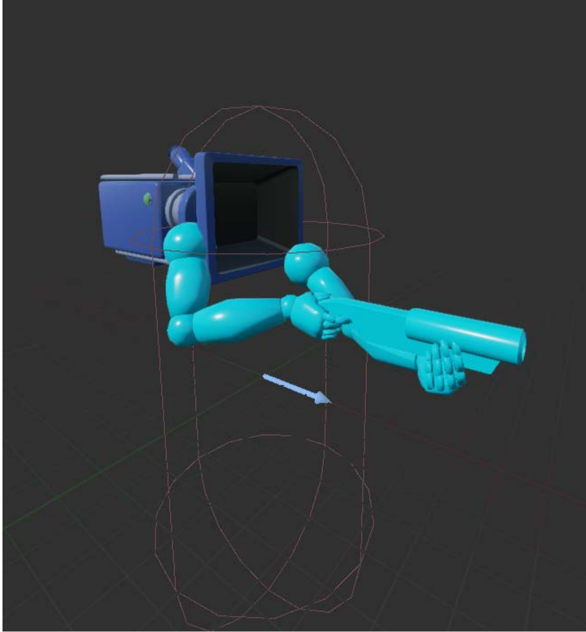


Figura 76. Captura del personaje desde el editor apreciándose sus componentes.

Fuente: Elaboración propia.

La cámara de Arbennig es una cámara únicamente en primera persona. Corresponde a la visión del personaje dentro del juego, representando el mundo desde su punto de vista.

Además, es la única cámara con la que contamos en proyecto, dado que el resto de componentes son menús y cinemáticas que se muestran en 2D sobre la pantalla.

5.1.6. Sonido

El audio en el videojuego pretende acompañar tanto menús como cinemáticas y niveles de juego. Lo dividiremos por comodidad en música, sfx, y UI (User Interface), de modo que con música nos referimos a audio que más bien son canciones de fondo que suenan constantemente, y SFX y UI son efectos de sonido, aunque los SFX podrán ser acciones del jugador (suenan como los de UI al no tener una posición en el mundo) o no (suenan en alguna posición en el mundo, y cuanto más nos acercamos a ellos, más suenan).

Respecto a la música, dependiendo de si nos encontramos en un menú, una cinemática, el nivel de juego, o el de extras, sonará una canción u otra de fondo en bucle.

Los SFX y UI, como se ha descrito, acompañarán a los distintos componentes del mundo, al personaje, y también a las acciones del jugador a través del menú (cuando pulse botones en ellos o pase el cursor por encima).

5.1.7. Sistema de ayuda

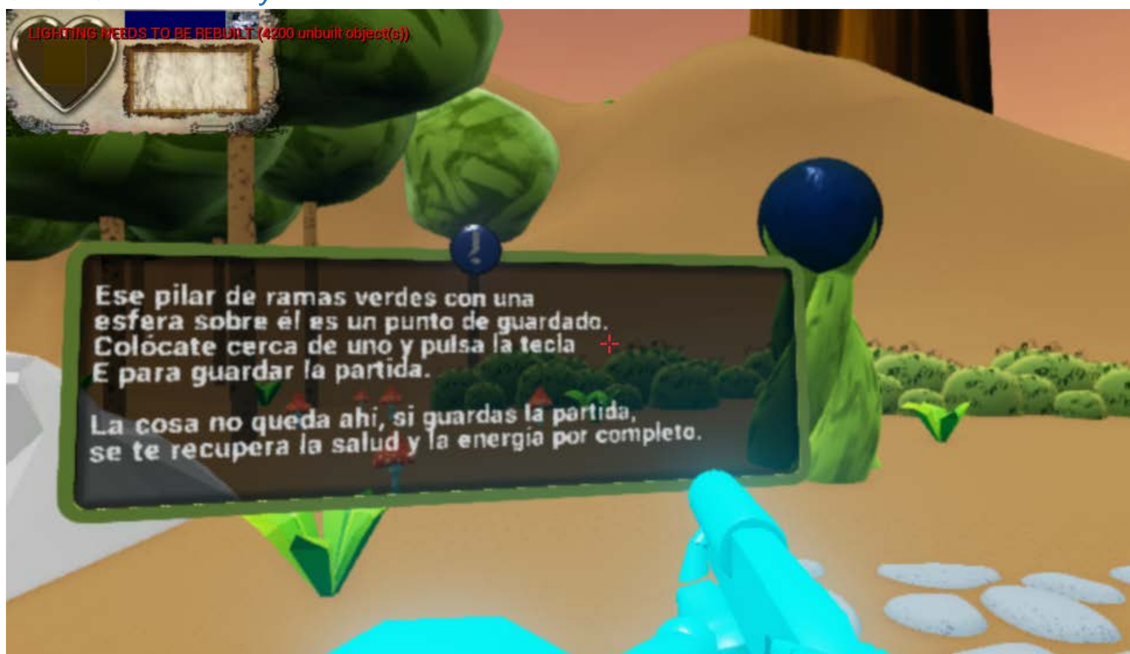


Figura 77. Captura de punto de información activado.

Fuente: Elaboración propia.

Con el fin de asistir al jugador cuando comienza la partida, la isla de inicio contiene una serie de mensajes informativos de ayuda, que aparecen cerrados, y se despliegan cuando el jugador se acerca a ellos lo suficiente. Estos mensajes proporcionan al jugador información relevante sobre los controles de juego, y algunos consejos para jugar.

Figura 78. Captura de punto de información sin activar.

Fuente: Elaboración propia.

5.1.8. Inteligencia Artificial

5.1.8.1. IA de NPCs (Árbol de la vida, aldeanos con aspecto de anciano y tótem de la isla final)

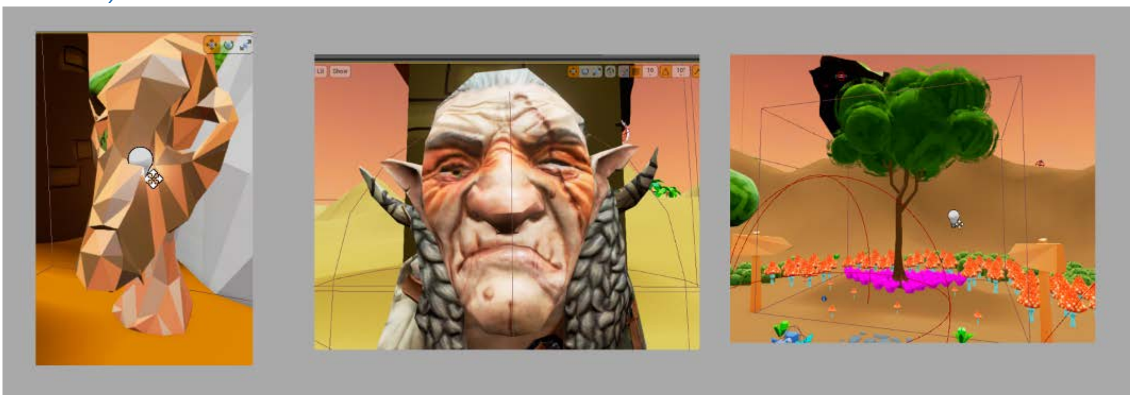


Figura 79. Tótem, aldeano con aspecto de anciano, y árbol de la vida desde el editor.

Fuente: Elaboración propia.

Respecto al árbol de la vida, aldeanos con aspecto de anciano, y el tótem de la isla final, su única lógica de juego consiste en la posibilidad de poder entablar una conversación con ellos.

Cuando el jugador interactúa cerca de ellos, se abre una ventana de diálogo que muestra el mensaje del NPC y una o varias respuestas del jugador.

Estos personajes no realizan ningún tipo de acción más, siempre aparecen quietos y no patrullan.

5.1.8.2. IA de los tótems guardianes de las runas.

Figura 80. Captura de tótem desde el editor.

Fuente: Elaboración propia.

Los tótems que guardan las runas tienen casi exactamente la misma funcionalidad que los NPCs normales, con el añadido que conversando con ellos, es posible aceptar el desafío que tienen preparado (si lo tuvieren), y una vez terminado, nos entregan la runa que guardan. Entonces, luego de esto, dejamos de poder conversar con ellos, puesto que su alma abandona su cuerpo al haber entregado la runa.

5.1.8.3. IA del Golem.

Figura 81. Captura del golem desde el editor.

Fuente: Elaboración propia.

El Golem de piedra, que aparece en el desafío de la sexta runa, funciona de la siguiente forma:

1. Permanece quieto mientras no puede ver al jugador.
2. Cuando ve al jugador, lo persigue por el mapa.
3. Cuando lo alcanza y colisiona con él, vuelve a estar quieto de nuevo.

Su funcionamiento es así dado que las condiciones del desafío de la runa 6 consisten en que el jugador debe alcanzar corriendo una bandera que se encuentra al final de la isla, y luego de esto,

llevarla hasta un lugar que el tótem le indica. El camino está lleno de golems, que al ver al jugador, lo perseguirán, y si lo tocan, el jugador pierde el desafío y tiene que volver a intentarlo.

5.1.9. Guía Técnica[25]

A continuación se describen algunos detalles técnicos acerca de los requerimientos de hardware, software, y la arquitectura del juego.

5.1.9.1. Hardware

Respecto al hardware necesario para jugar, se recomienda tener el siguiente mínimo:

Memoria RAM: 8 GB

Procesador: Intel o AMD de tipo Quad-Core

Tarjeta Gráfica: Compatible con DirectX11.

5.1.9.2. Software

Se recomienda Windows 7 u 8 con arquitectura de 64 bits.

5.1.9.3. Arquitectura del juego

La arquitectura que sigue el videojuego se corresponde a la ya explicada en el apartado de Marco teórico o Estado del arte (2.2.4). Es decir, la arquitectura en la que se basa el motor Unreal Engine 4.

5.2. Desarrollo e implementación.

Nos disponemos, a explicar el proceso de realización del videojuego que hemos diseñado previamente en el GDD, donde procedemos a explicar cada uno de los aspectos desarrollados, haciendo mayor o menor hincapié en ellos dependiendo de los aspectos más relevantes que estos contengan.

5.2.1. Creación del proyecto

El primero de todos los pasos es la creación del proyecto y su correspondiente set up.

En nuestro caso, dada la compatibilidad que ofrece con Oculus Rift DK1, creamos a través del Epic Games Launcher un proyecto en blanco de Blueprints con la versión 4.6.1, ya que en versiones posteriores se ha comprobado personalmente que no funciona correctamente esta versión del DK de Oculus. Una vez creado, ya podremos acceder a él rápidamente desde el launcher.

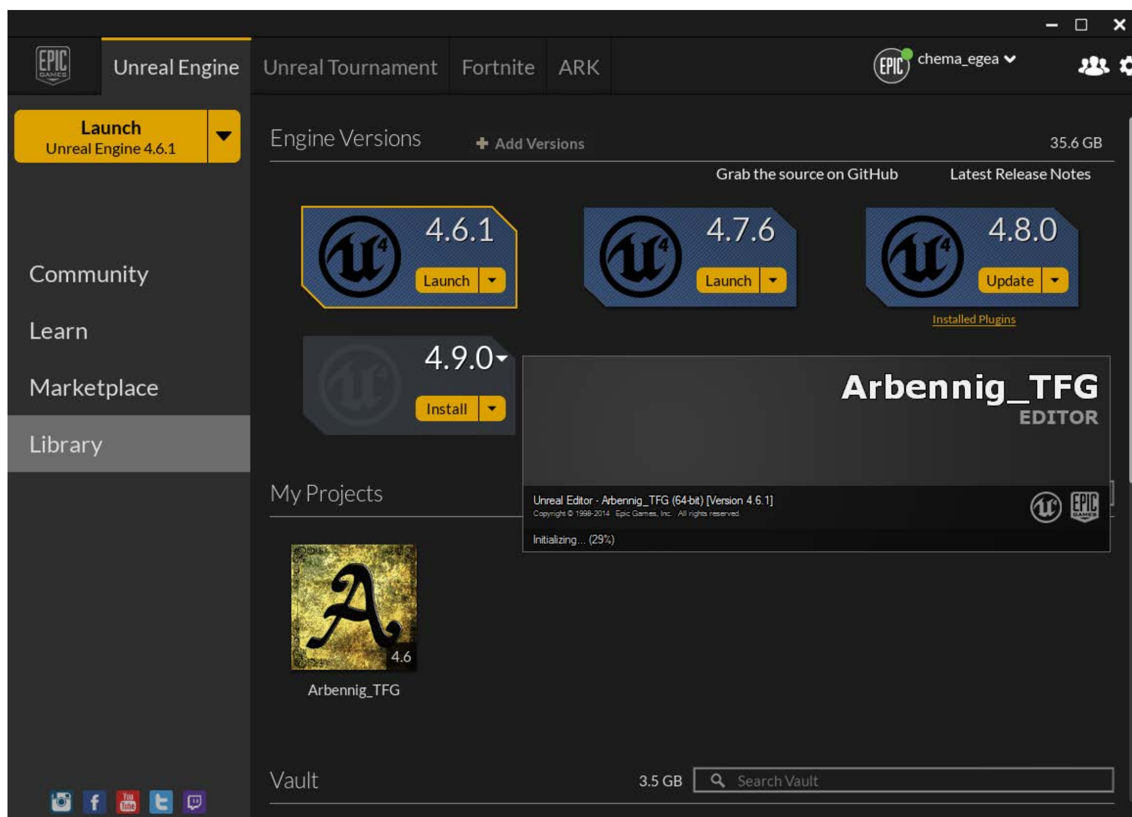


Figura 82. Captura de lanzamiento del juego desde el Epic Games launcher.

Fuente: Elaboración propia.

Una vez creado el proyecto y haber accedido a él, es necesario configurarlo, creamos desde el editor las clases que nos harán falta para poder hacer el setup del proyecto posteriormente.

Las clases básicas que vamos a crear son:

- Una clase GameMode, donde estableceremos las reglas del juego.
- Una clase de tipo HUD (que es distinta al HUD que tendremos visual, ya que esta clase HUD funciona con código, y es donde representaremos el target).
- Una clase PlayerController que nos servirá para que el jugador posea al personaje principal.
- Una clase Character, que poseerá el PlayerController (aunque no por defecto, para que solo sea durante el juego cuando controlamos al personaje, y mientras, tengamos una free view que no reaccione con eventos del jugador, solo a los que indiquemos de ratón para los menús). Creamos Character y no un Pawn básico porque con Character ya poseemos funcionalidades de personaje humanoide que nos facilitarán el desarrollo.
- Una clase PlayerCameraManager, que asignaremos al PlayerController para tener el control de una cámara creada por nosotros.

Una vez creadas estas clases accedemos a las Project Settings del editor, y modificamos los valores que consideremos necesarios. En nuestro caso, hemos modificado (debemos tener en cuenta que ya tenemos niveles y personajes creados, que en esta guía de implementación se define su implementación en los puntos posteriores):

Description, añadiendo la información pertinente acerca del creador, datos del proyecto.

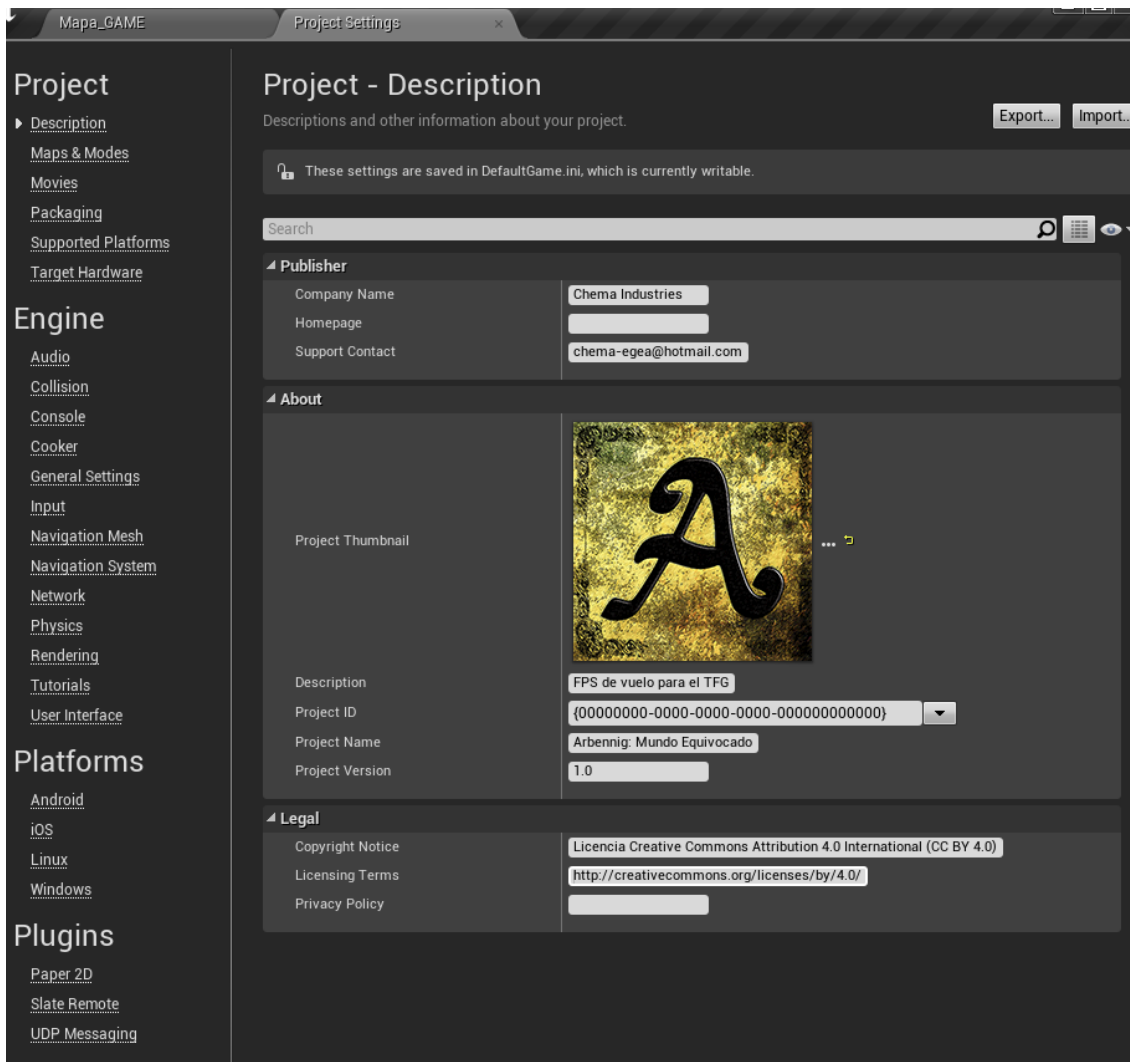


Figura 83. Captura de las modificaciones añadidas en el apartado descripción para Arbennig.

Fuente: Elaboración propia

Maps & Modes, seleccionando el mapa que se carga por defecto al cargar el juego (mapa de juego) y el editor (mapa de menú principal), y los modos de juego, descritos en el apartado de marco teórico o estado del arte (2.2.4), en los que indicamos nuestra clase GameMode, si tenemos alguna clase Pawn por defecto (en nuestro caso no tenemos asignada, e ingame poseemos un Character), el HUD que hayamos creado, la clase PlayerController, y por último, la clase GameState.

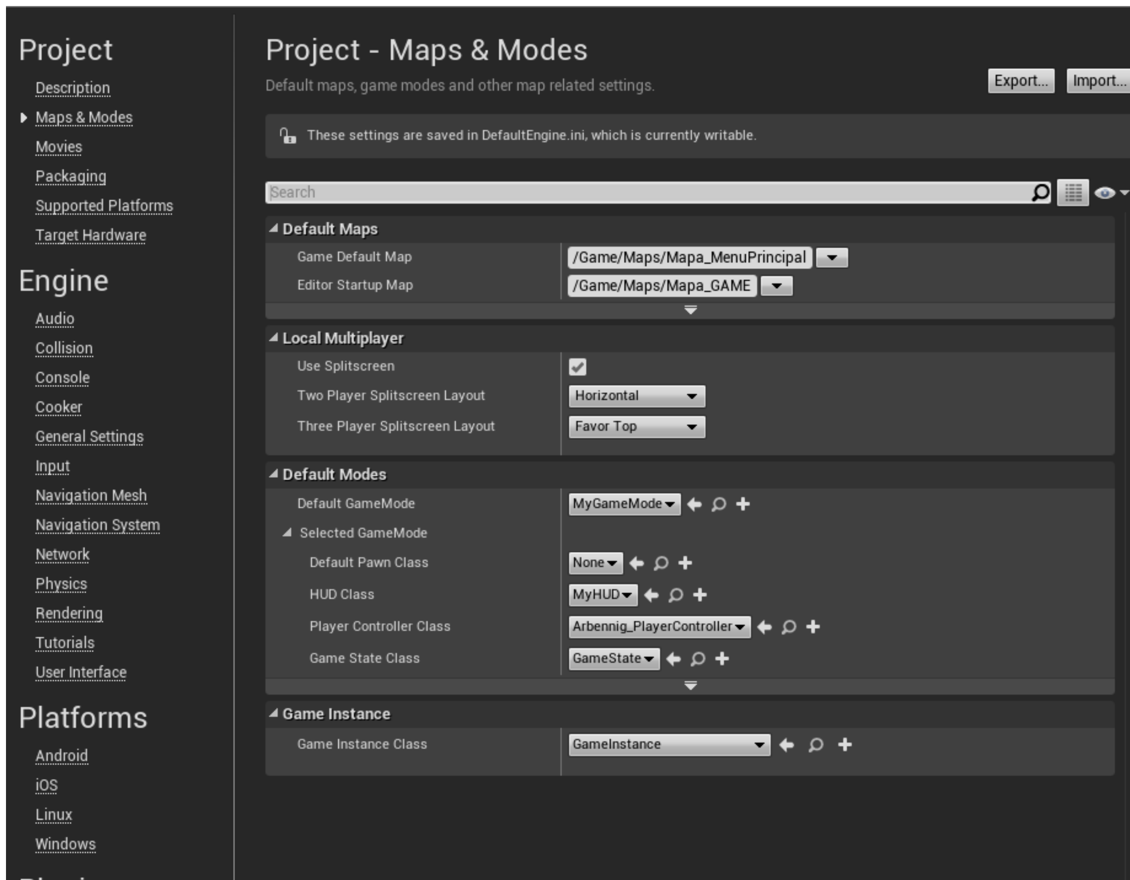


Figura 84. Captura de la configuración de Maps & Nodes para el proyecto Arbennig.

Fuente: Elaboración propia.

Supported Platforms, seleccionando que queremos producir el videojuego solo para Windows.

Audio, donde elegimos la clase de sonido por defecto que vamos a utilizar, en nuestro caso hemos creado la clase BP_Master_Sound (5.2.4).

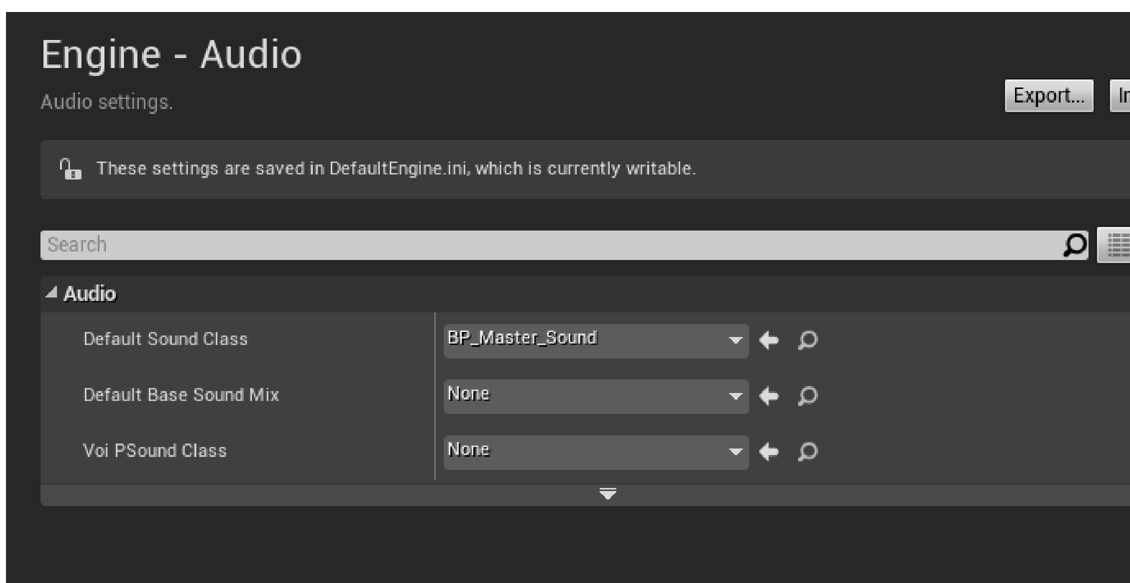


Figura 85. Captura que indica dónde seleccionar la clase de sonido del juego.

Fuente: Elaboración propia.

Input, donde se han añadido eventos identificativos para los controles de juego, es decir, se ha añadido una lista de definiciones para los controles de modo que podamos acceder luego a ellas desde el Graph Editor de blueprints por el nombre que indiquemos. Se ha decidido incluir en este apartado los controles para un control de xbox, con previsión a posibles mejoras futuras que permitan su uso.

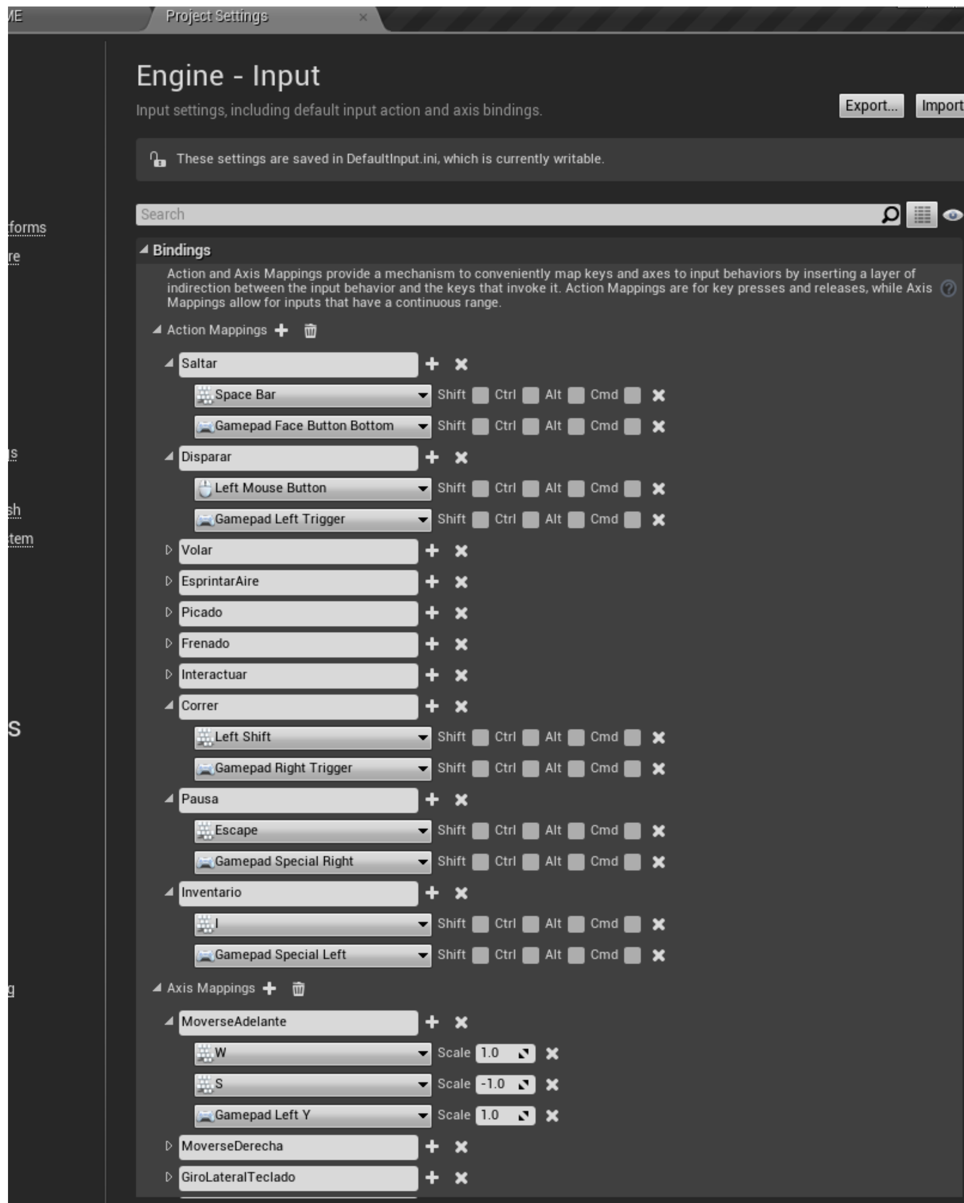


Figura 86. Captura de la configuración de Inputs del proyecto.

Fuente: elaboración propia.

Cabe mencionar para el apartado de input, que contamos con valores de Action Mappings, y valores de Axis Mappings. La diferencia entre estos es que con los Axis Mappings se guarda también un valor de tipo float, ya que son para movimiento, lo que facilita ahorra que si queremos que mientras el jugador pulse una tecla, esté, por ejemplo, ascendiendo, simplemente vamos añadiendo el float a la posición del jugador.

También, cabe por último destacar que para los valores de axis mappings, no es conveniente realizar una definición para cada uno de los movimientos del personaje, sino que podemos simplemente poner su float en negativo. Por ejemplo, si el personaje se mueve hacia delante con W, y esto tiene asignado un float de escala 1.0, añadimos a esta acción de moverse hacia delante, que si pulsa la tecla S, su float va a ser de -1.0, y con esto, ya tenemos definido el control hacia delante y hacia atrás.

Figura 87. Captura de definición de control de Movimiento hacia delante y hacia atrás.

Fuente: Elaboración propia.

Es necesario también asignar a nuestro PlayerController una clase PlayerCameraManager que nos permita tener control sobre la cámara del juego.

Simplemente en las definiciones de PlayerController, elegimos la clase creada.

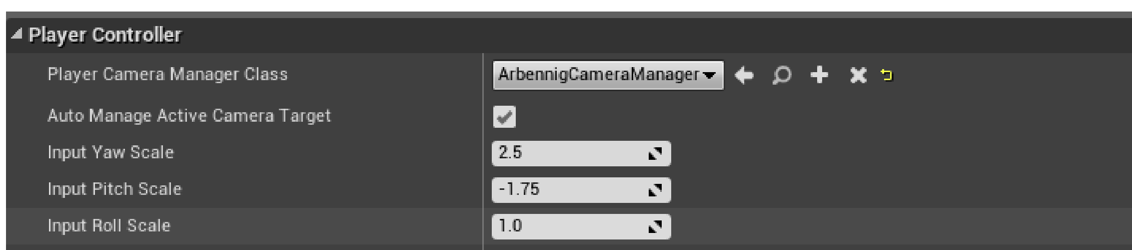
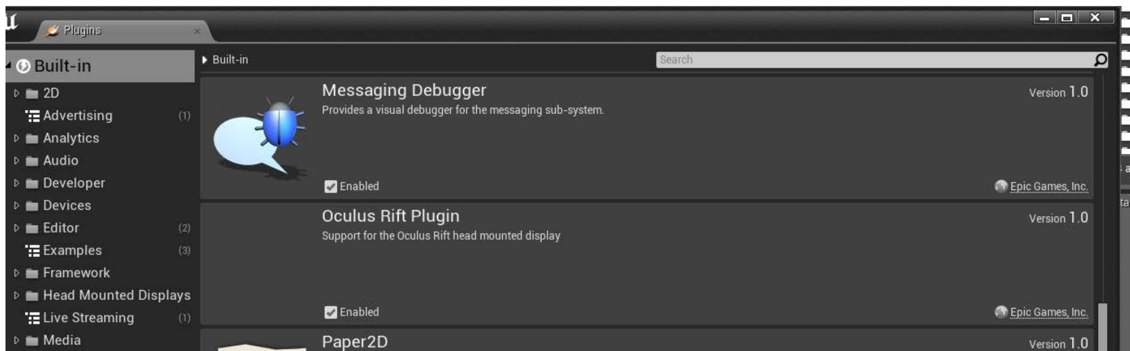


Figura 88. Captura de la clase PlayerCameraManager creada asignada al PlayerController.

Fuente: Elaboración propia.

Por último, en la ventana de plugins que encontramos en Window, activamos el plugin de oculus rift, que está incluido por defecto en el motor.



5.2.2. Implementar las acciones del personaje.

A continuación se procede a explicar el blueprint de creación del control de movimiento del personaje en función de sus cambios de estado (tierra, vuelo, y Racing). El blueprint de control se ha optado por realizarlo en la clase Character, en nuestro caso, Arbennig_Character, y es esta clase probablemente la más extensa de todo el juego junto con el Level Blueprint, es tan extensa que realizando un zoom out desde el Graph Editor, no podemos ver más que una pequeña parte de este blueprint, de modo que se ha optado por realizar una explicación de los pasos más relevantes en cuanto a su creación, pero no en pleno detalle, puesto que la extensión de esta memoria casi se duplicaría.

Dicho esto, debemos saber que dado que la clase Character proporciona un enumerado con distintos estados, entre los que se incluye Walking (el estado que usaremos para tierra), Flying (el estado que usaremos para vuelo), y Falling (lo usaremos para cuando el jugador decida hacer la caída en picado). Pero para ayudarnos con el desarrollo, vamos a crearnos también un par de enumerados, uno para estados de tierra, que contendrá Walking y Falling, y otro para estados de aire, que contendrá a Flying, y Racing (y añadimos dichos enumerados como variables al personaje para poder utilizarlos). El flujo de cambio de estos estados es el siguiente:

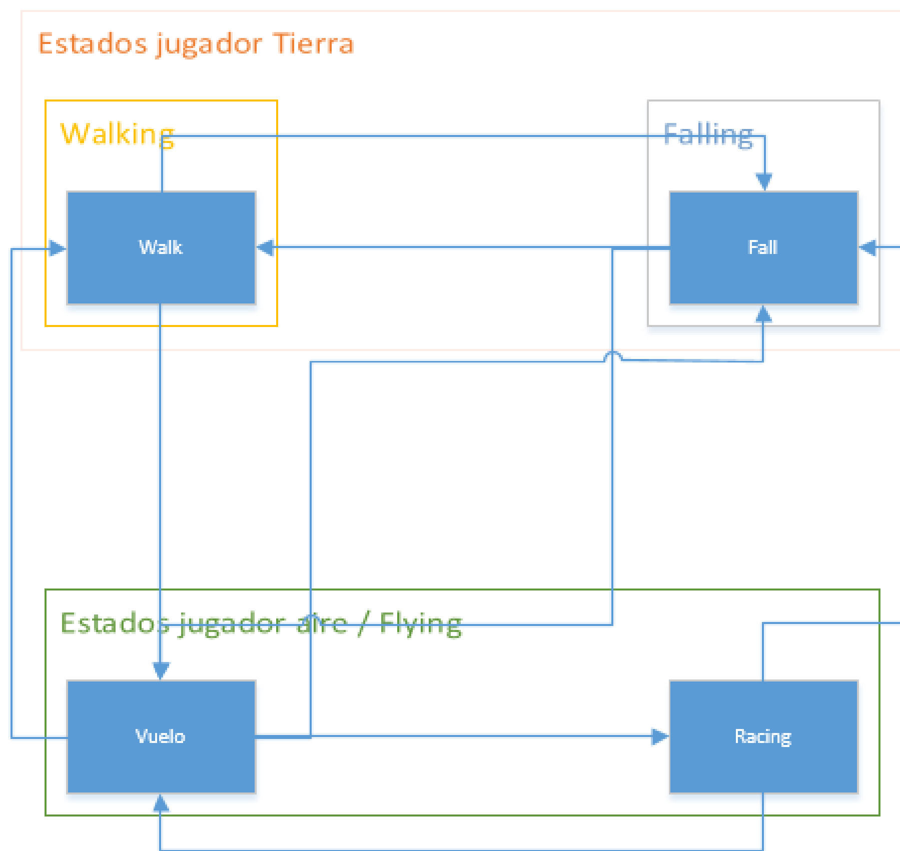


Figura 89. Diagrama de cambio de estados.

Fuente: Elaboración propia.

Como se puede apreciar:

- Desde Walk podemos pasar a los estados Vuelo y Fall.
- Desde Vuelo podemos pasar a los estados Walk, Fall, y Racing.
- Desde Racing podemos pasar a Fall y Vuelo.
- Desde Fall podemos pasar al estado Walk y al estado de Vuelo.

Es preciso saber que para que podamos tener funcionalidad de los distintos estados de la clase Character, lo que debemos hacer es en sus parámetros de inicialización, marcar las casillas de Can Fly, Can Jump, y Can Walk, en su Movement Component.

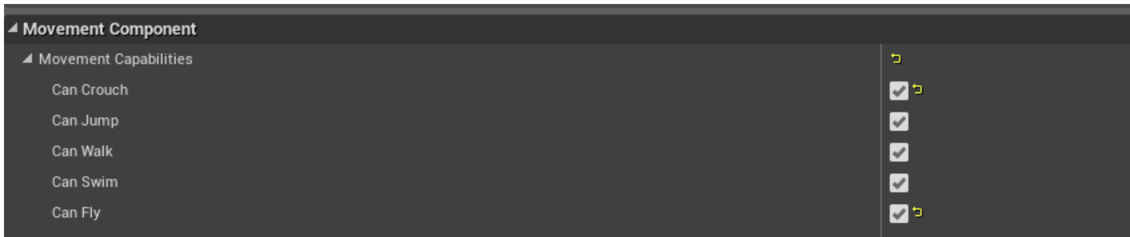


Figura 90. Captura de las variables que permiten al jugador cambiar entre los estados de movimiento por defecto.

Fuente: Elaboración propia.

5.2.1.1. Rotación de la cámara.

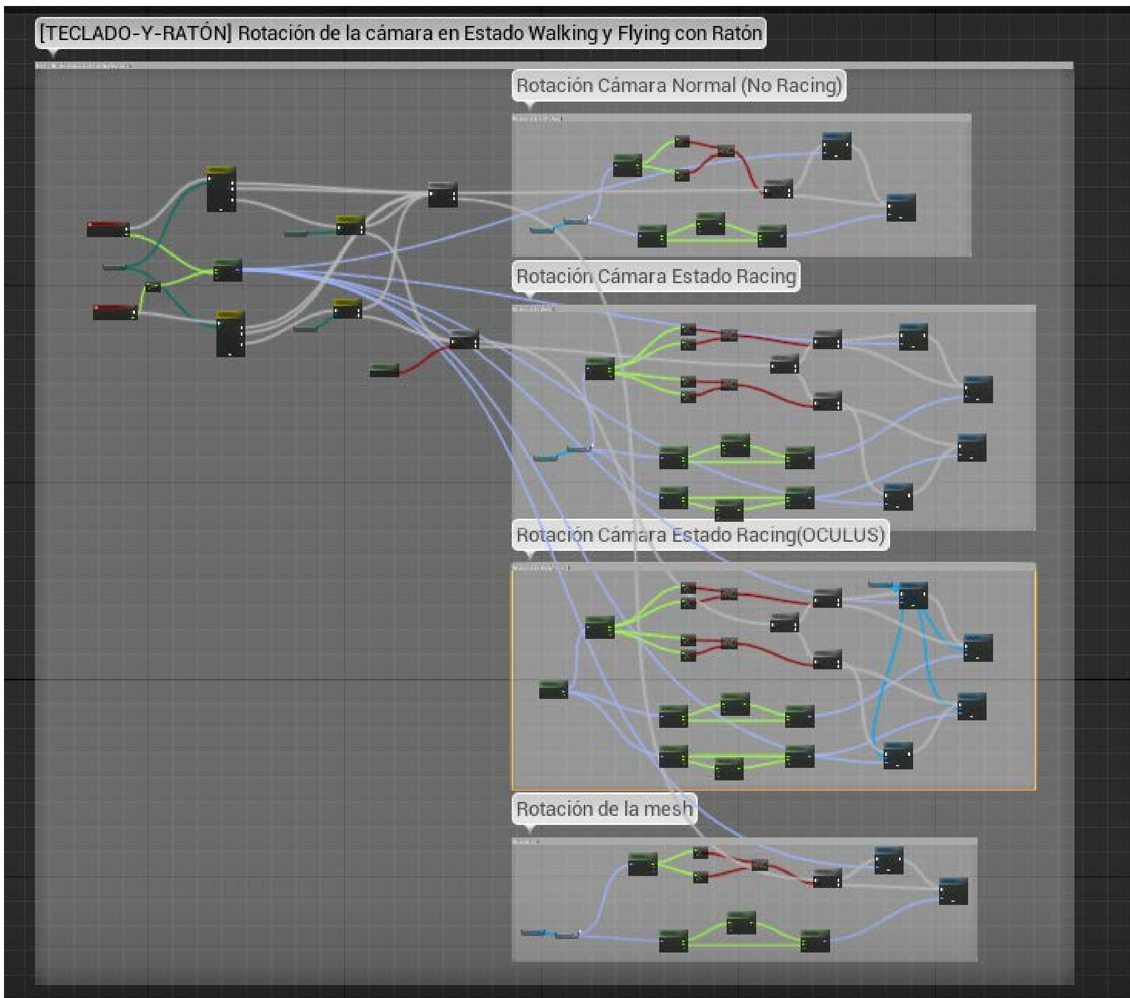


Figura 91. Visión general de la porción del grafo de rotación de cámara.

Fuenta: Elaboración propia.

La rotación del jugador se ha optado por dividirla en dos partes, por una lado, la cámara, y por otro, la malla que tiene. Esto se ha decidido por el control de Racing, donde se puede mover libremente con respecto al cuerpo.

Para la implementación del giro de cámara, primero hacemos una serie de comprobaciones iniciales para determinar el estado de movimiento del jugador, y también, si está o no oculus conectado. Además, creamos un rotator (vector de rotaciones) con los inputs de entrada del ratón, de modo que podamos sumarlos luego a la rotación de cámara y malla.

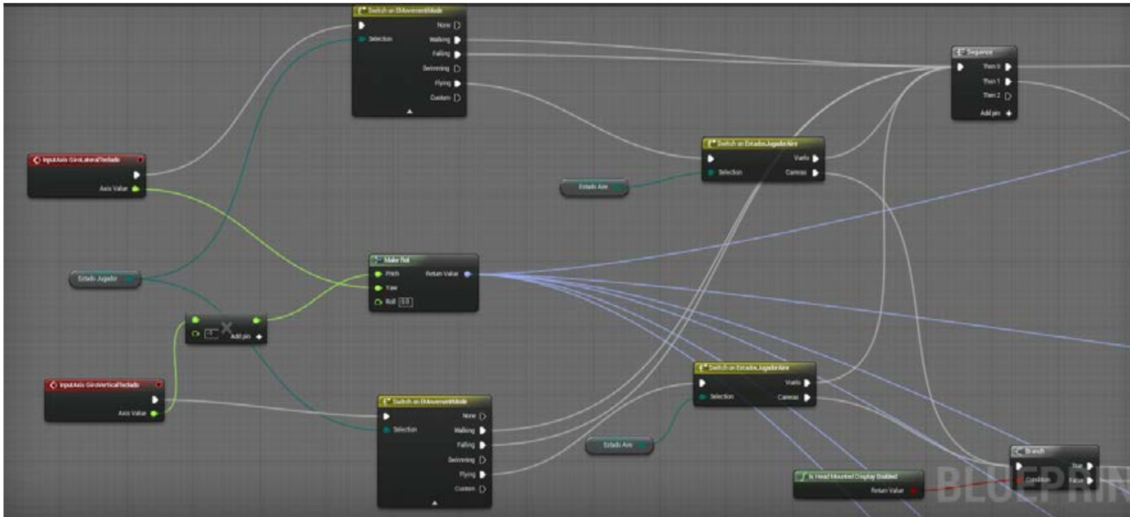


Figura 92. Captura de la porción de grafo de comprobaciones iniciales antes de proceder a girar.

Fuente: Elaboración propia.

Luego de esto, si no estamos en modo Racing, el giro de la cámara se aplica considerando un clamp del ángulo en que nos encontramos para no pase de cierta cantidad y surjan problemas de rotaciones al intentar pasar los límites de los 90º. Y por último, se añade la rotación si no estamos superando ningún límite, y luego de esto, se establece la rotación de nuevo para evitar un problema de bloqueo de cámara que surge en Unreal debido a que intenta aplicar las rotaciones de pitch y yaw a la vez, de esta forma lo arreglamos.

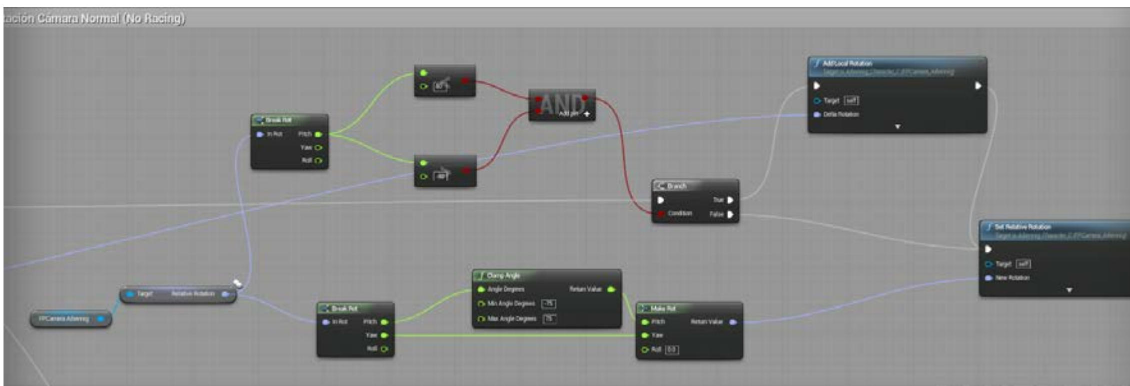


Figura 93. Captura de rotación de cámara cuando no estamos en modo Racing.

Fuente: Elaboración propia

En cuanto a la rotación de la cámara en el estado Racing, es un tanto particular, ya que está diseñada para que la compatiblidad con oculus sea correcta. Cuando entramos a modo Racing, la cámara queda desprendida del cuerpo del jugador, estando libre para que podamos visualizar el entorno mientras volamos en una dirección concreta. A pesar de esto, la diferencia con la cámara que no es Racing, es que aquí limitamos tanto el ángulo del pitch (arriba/abajo) como el del yaw (hacia los lados), porque la cabeza del personaje no puede girarse hacia atrás.

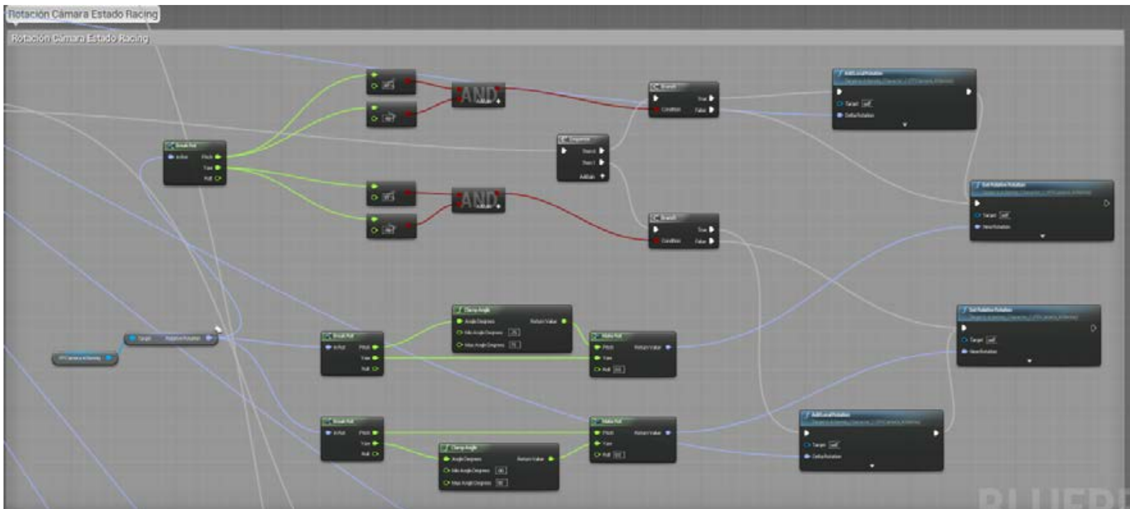


Figura 94. Captura de rotación de cámara cuando estamos en modo racing

Fuente: Elaboración propia

Para la rotación de la cámara con oculus en racing, la diferencia principal es que obtenemos la orientación desde el periférico en lugar de directamente desde la cámara que tenemos, el resto de pasos son los mismos.

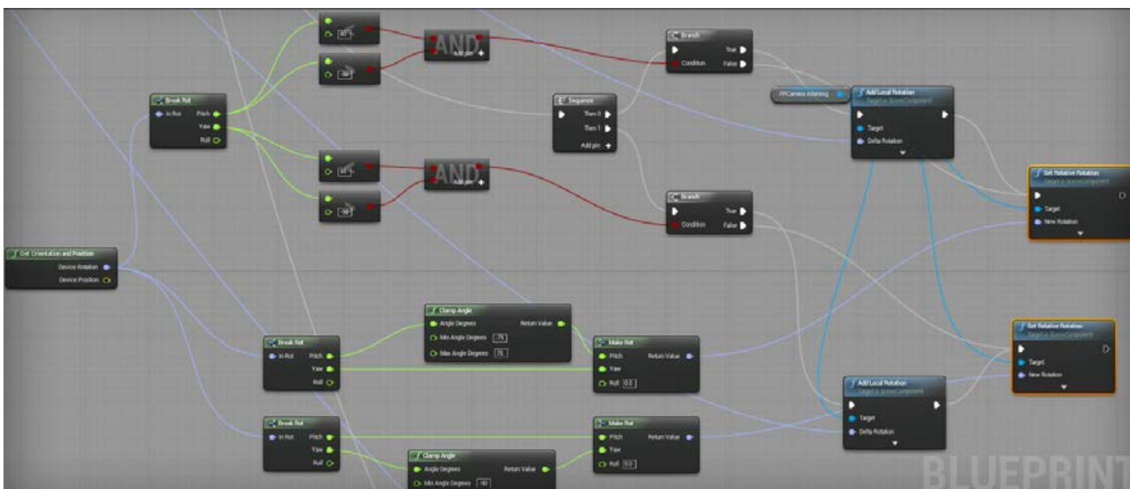


Figura 95. Captura de rotación de cámara cuando estamos en modo Racing y oculus activado.

Fuente: Elaboración propia

Para acabar, en cuanto a la rotación de la malla del personaje, va a rotar igual que rota la cámara cuando estemos en un estado que no sea Racing. Ya que en Racing girará con personaje en sí (recordemos que el personaje es un conjunto de una malla y una cámara, así que tenemos una entidad de personaje, que puede girar y mover todos sus componentes, y por otro lado, podemos acceder a cada uno de sus elementos por separado y realizarles las modificaciones que necesitemos).

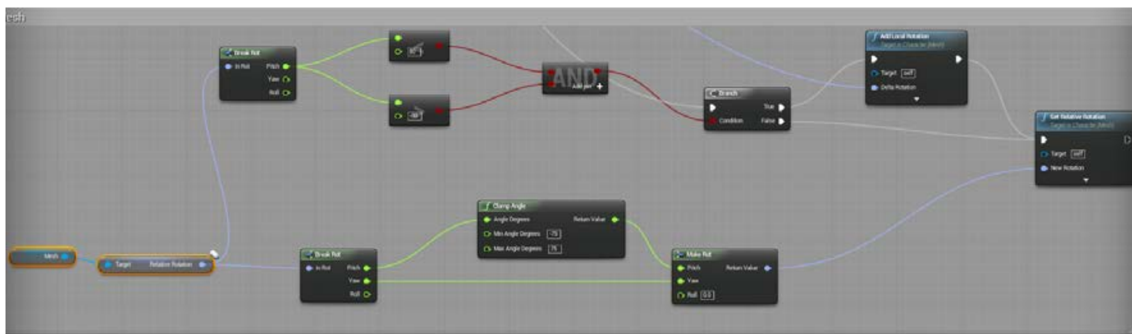


Figura 96. Captura de porción de código de rotación de la malla.

Fuente: Elaboración propia

5.2.1.2. Movimiento del personaje.

El movimiento del personaje viene determinado dependiendo del estado en que nos encontremos, dado que la lógica de funcionamiento del estado Racing es considerablemente distinta a la del resto de estados, utilizando en el resto las teclas WASD para desplazarnos, mientras que en Racing hay una fuera que nos empuja hacia delante y con WASD lo que hacemos es girar al personaje.

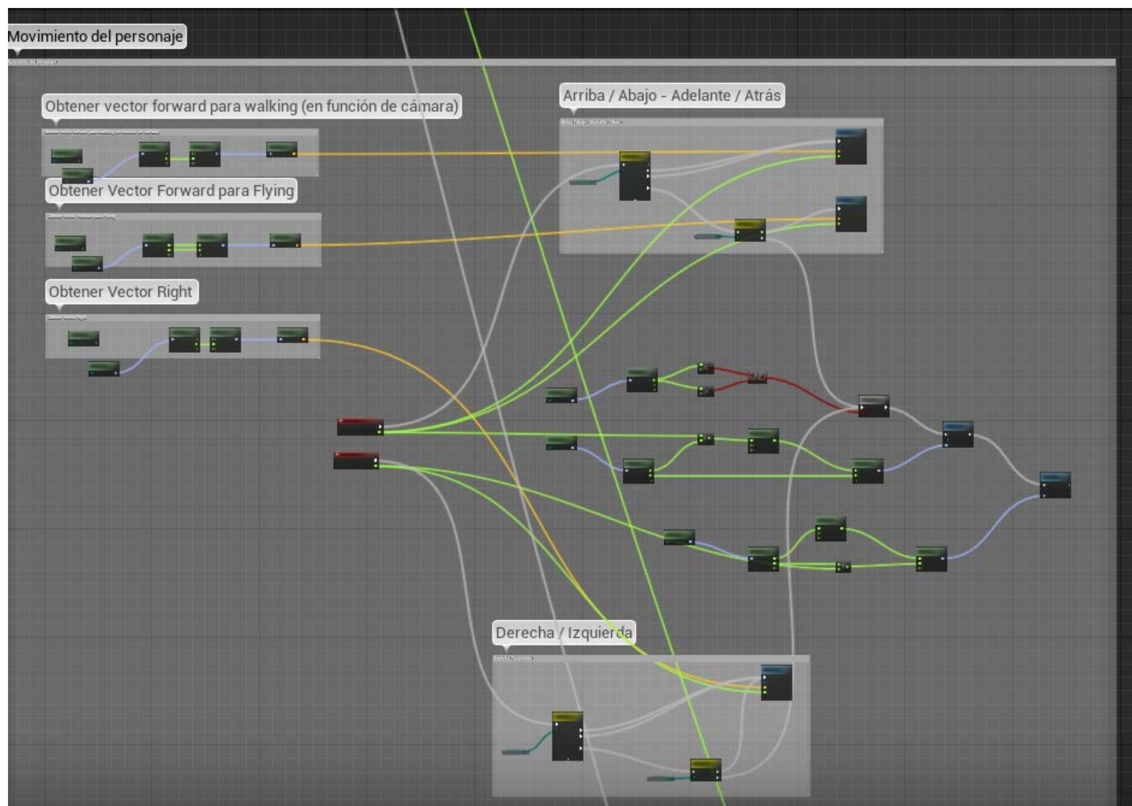


Figura 97. Captura general del movimiento del personaje en todos sus estados.

Fuente: Elaboración propia

Lo primero que hacemos para poder establecer el movimiento siempre y cuando no estemos en modo Racing, es obtener sus vectores Forward (para movimiento hacia delante y hacia atrás) y Right (para movimiento lateral). Estos vectores los podemos obtener en función de la rotación del personaje en el mundo de forma automática, ya que el motor nos proporciona una función para ello a la que solamente hay que pasarle un rotator.

Dado que en walking solo nos podemos mover digamos por tierra, es decir, en un espacio “bidimensional”, mientras que en vuelo podemos movernos libremente también hacia arriba y hacia abajo, calculamos dos vectores forward distintos.

Para el caso del vector Right no nos hace falta esta distinción, ya que es el mismo.

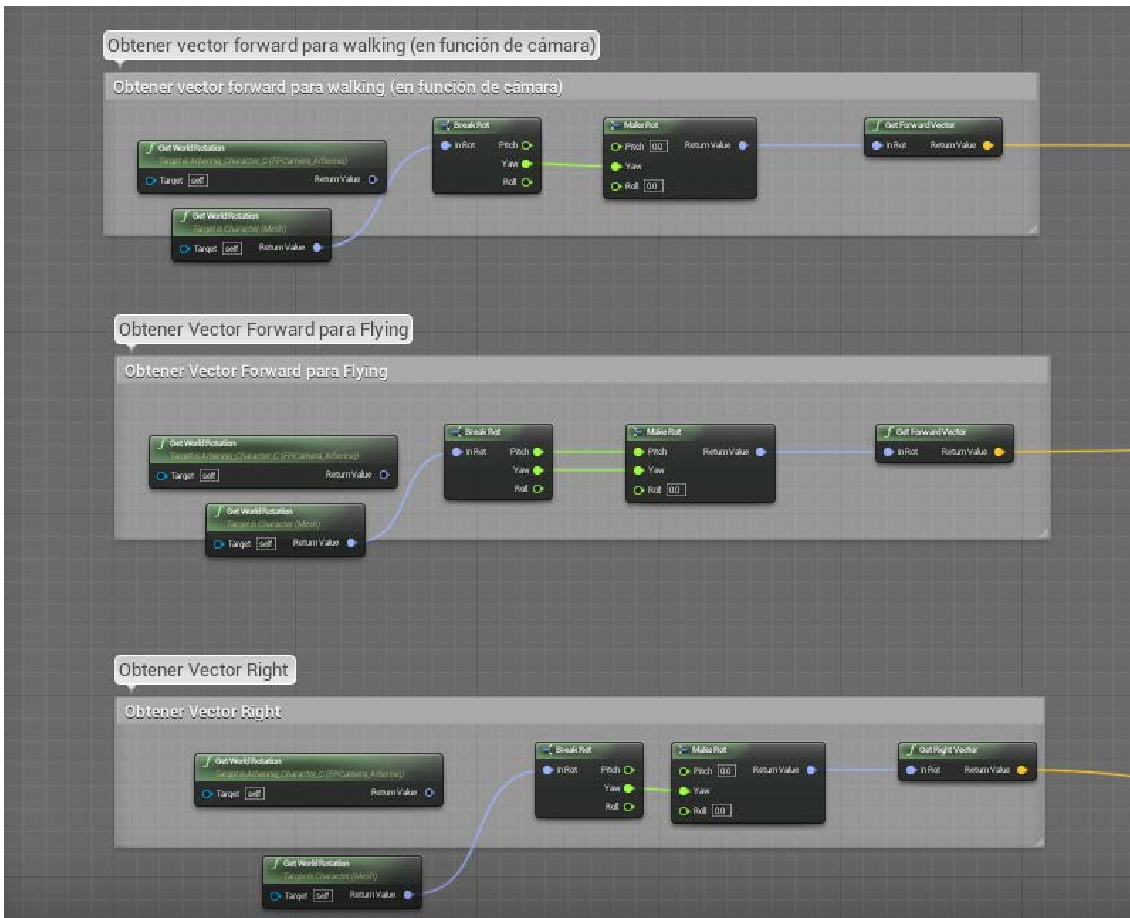


Figura 98. Captura de la obtención de los vectores que apuntan hacia delante y hacia la derecha para determinar el movimiento.

Fuente: Elaboración propia

Una vez obtenidos los vectores, solo es necesario añadir a nuestro Pawn el movimiento en la dirección de dichos vectores, tanto para movimiento hacia delante/hacia atrás, como para movimientos laterales.

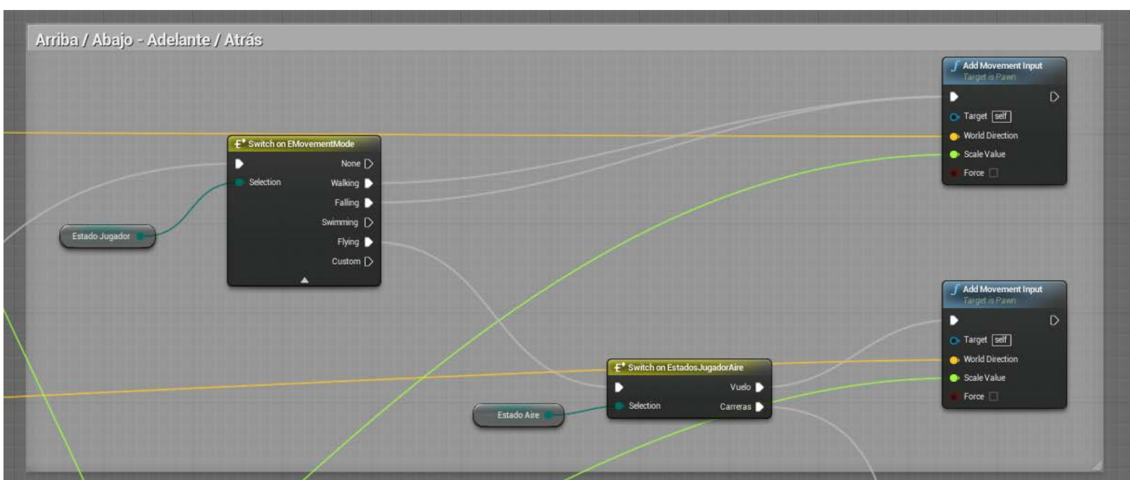


Figura 99. Captura de la lógica de movimiento hacia delante y hacia atrás en Walking, Falling y Flying(vuelo).

Fuente: Elaboración propia

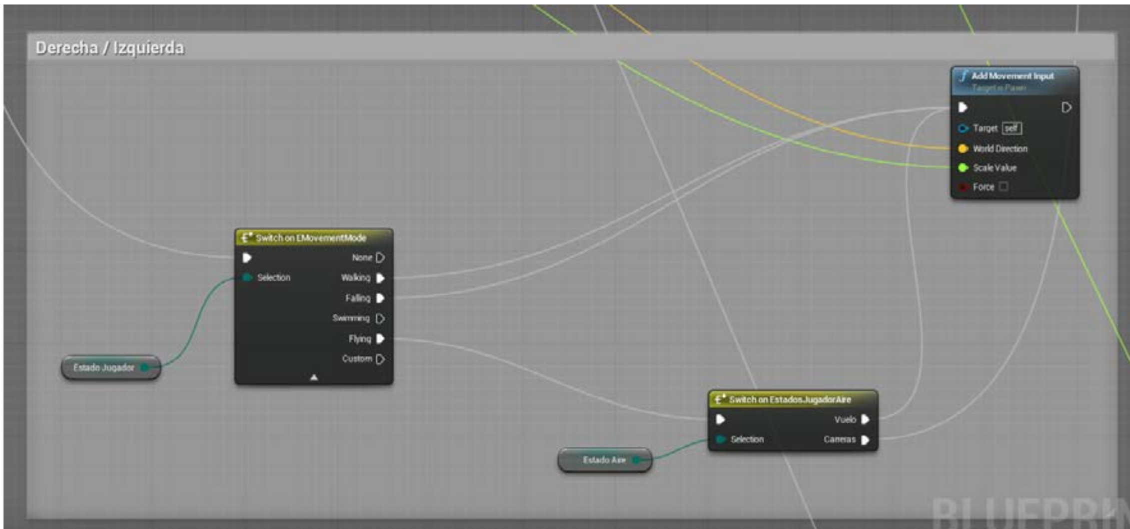


Figura 100. Captura de la lógica de movimiento para moverse hacia la derecha e izquieda en Walking, falling, y flying (vuelo).

Fuente: Elaboración propia

Por otra parte, para el movimiento en estado Racing, como se ha descrito anteriormente, ya está aplicado de forma constante al entrar en dicho estado, de modo que cuando el jugador pulsa alguna de las teclas WASD, lo que hace es girar el Actor (estableciendo, eso sí, al igual que con los giros descritos anteriormente, unos controles de ángulos para que no pasen de ciertas cantidades que den problemas).

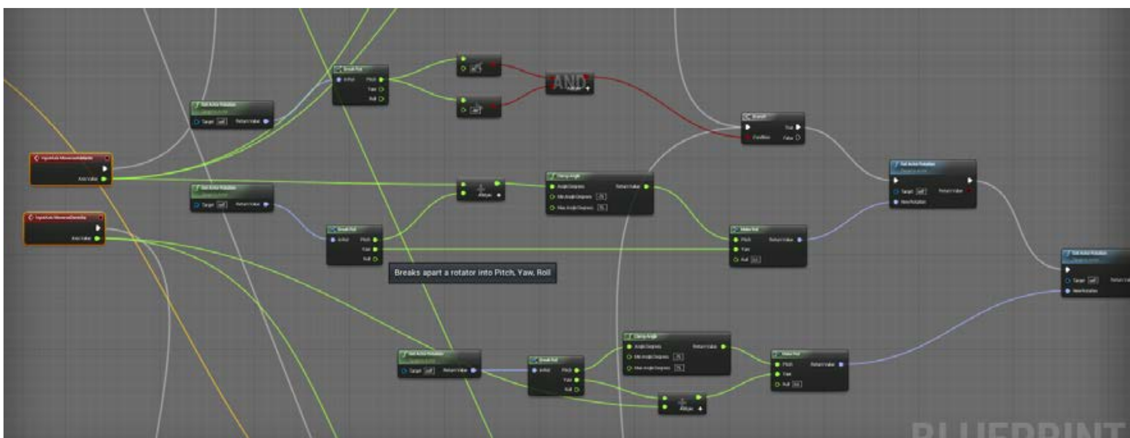


Figura 101. Captura de pantalla para el movimiento en Racing.

Fuente: Elaboración propia

5.2.1.3. Ascender/Descender

El ascenso y descenso vertical del personaje, que se da solo en el estado de vuelo, se aplica simplemente añadiendo movimiento en el eje Z a nuestro personaje usando su evento de Ascenso.

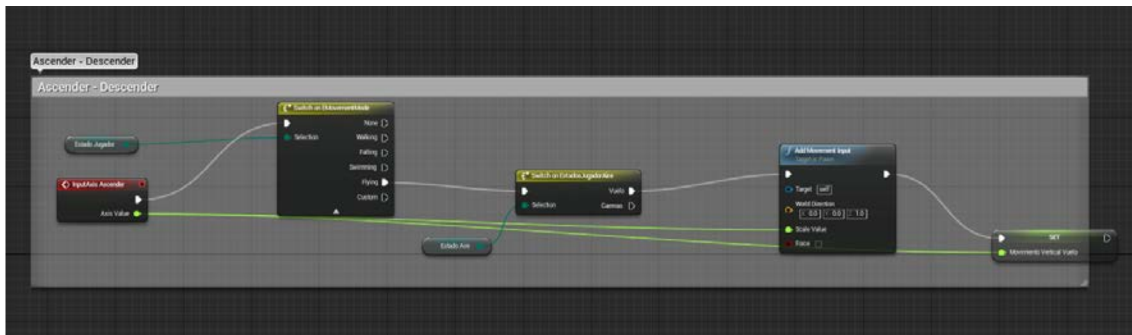


Figura 102. Captura de la lógica para ascender y descender en Vuelo (no Racing).

Fuente: Elaboración propia

5.2.1.4. Saltar

La lógica del salto la ofrece ya implementada la clase character, pero para aplicarla a nuestro personaje, controlamos que el estado en que se encuentra sea Walking.

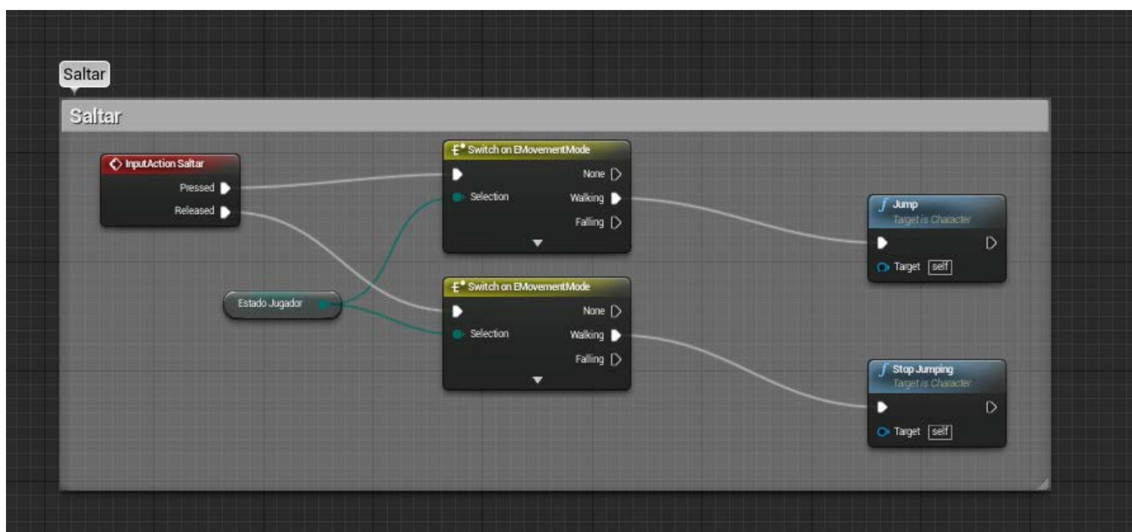


Figura 103. Captura de pantalla para realizar saltos (solo en Walking)

Fuente: Elaboración propia

5.2.1.5. Frenada en seco

La frenada en seco, que solo puede ser aplicada en el estado de vuelo, la realizamos accediendo a las componente de Character Movement del personaje, que contiene propiedades y funciones relacionadas con detalles de movimiento del pawn, y accedemos a su velocidad a 0,0,0.

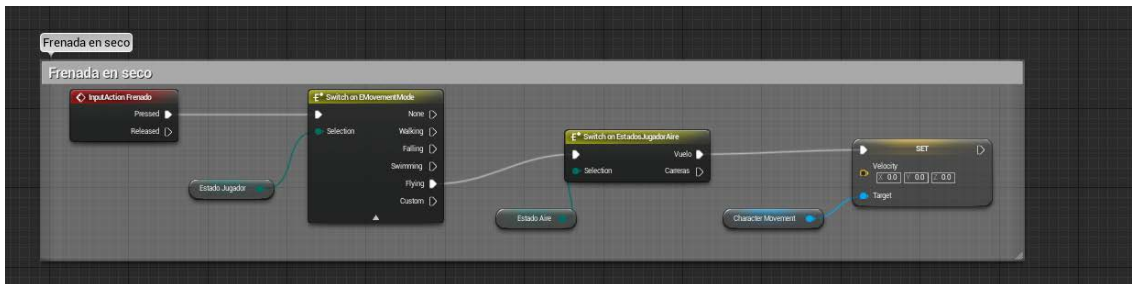


Figura 104. Captura con la lógica de realizar un frenado en seco en estado de vuelo.

Fuente: Elaboración propia

5.2.1.6. Sprint corriendo

Para que el jugador pueda correr, accedemos a su componente character movement y modificamos su velocidad máxima de movimiento a un poco más, de 600 a 900.

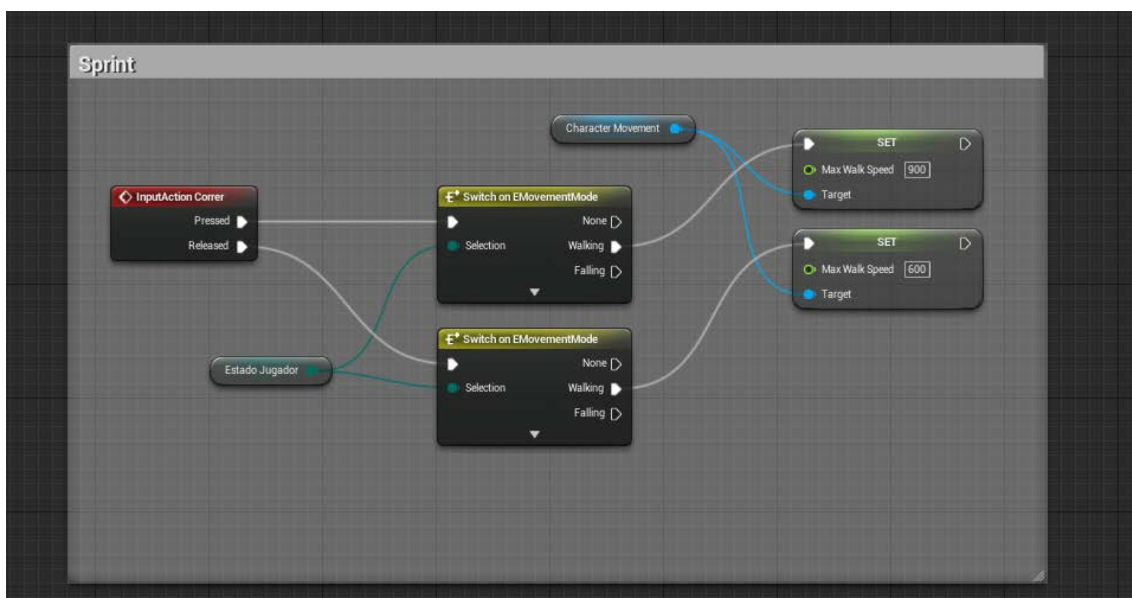


Figura 105. Captura de pantalla con la lógica para correr en Walking.

Fuente: Elaboración propia

5.2.1.7. Turbo en Racing

La lógica del turbo la dividimos en dos partes, por un lado, la lógica cuando pulsa el botón de activar el turbo, que solo está disponible cuando nos encontramos en el estado Racing, y donde lo que hacemos es que cuando pulsamos la tecla de turbo y además nuestra energía no es 0, cambiamos la velocidad máxima de movimiento al doble de lo que permite el estado Racing, y activamos una booleana que nos permita gestionar el consumo de energía. Cuando el jugador suelte el botón de turbo, se restablece la velocidad máxima de movimiento de vuelo y la booleana.

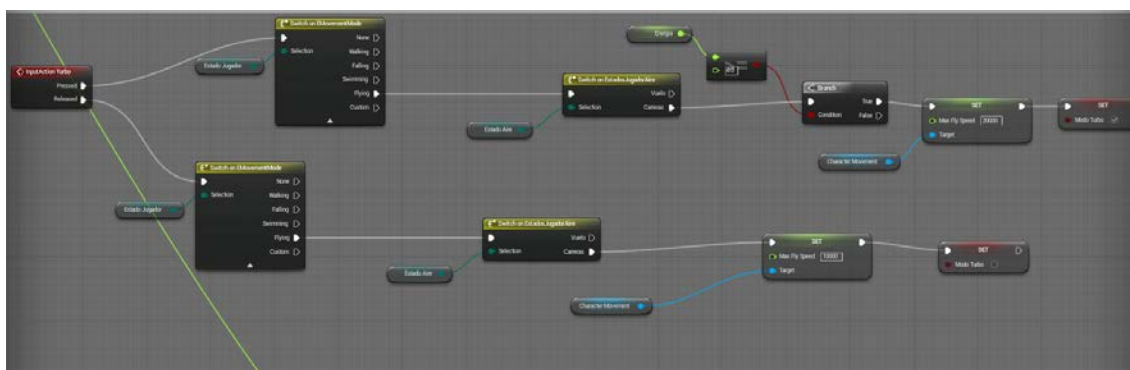


Figura 106. Captura de la lógica de realizar un turbo en Racing.

Fuente: Elaboración propia

En la segunda parte, del turbo, donde cada instante controlamos la gestión de la energía, comprobamos si está o no el turbo activado, y en función de esto, vamos a consumir energía o a recuperarla cada segundo poco a poco.

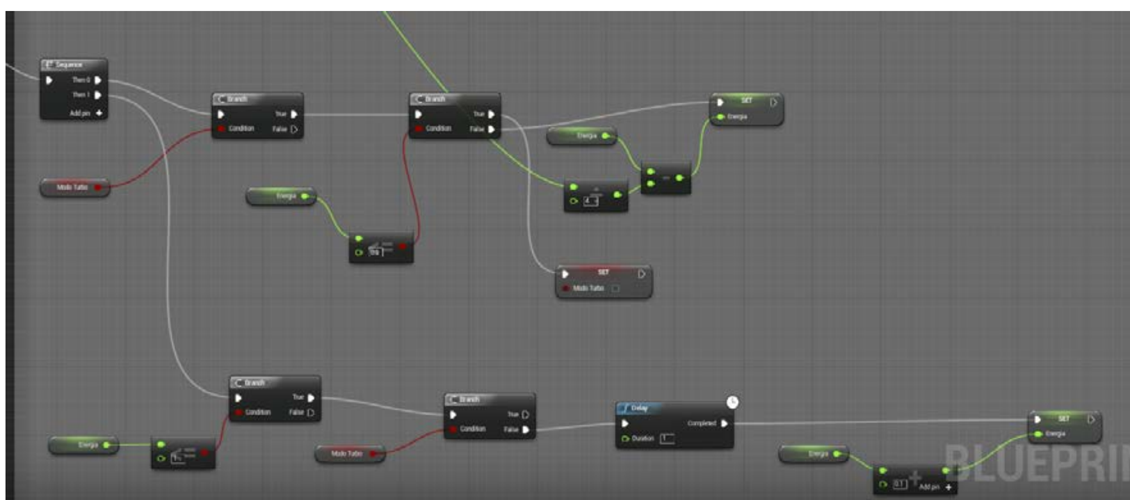


Figura 107. Captura de comprobaciones para disminuir/aumentar la energía del jugador en función de si está en uso o no.

Fuente: Elaboración propia

5.2.1.8. Gestión de cambios de estado.

Los cambios de estado que puede realizar el jugador, pueden ser manuales y automáticos, de modo que si está caminando y quiere volar, puede pasar a dicho estado pulsando una tecla, pero cuando está corriendo y de repente cae por ejemplo de alguna isla, pasa automáticamente del estado Walking a Falling.

5.2.1.8.1. Cambios de estado manuales.

Estos cambios, como hemos dicho, son los cambios que el jugador puede realizar pulsando los botones o teclas habilitados para ello.

5.2.1.8.1.1. Cambios de estado manuales de Falling

El paso de estado manual a Falling es posible realizarlo desde el estado de vuelo flying y también desde Racing.

Para ello simplemente detectamos que el jugador efectivamente está en uno de esos estados, y entonces realizamos el cambio de estado y colocamos la gravedad a la que está sometida el personaje a 1 (es un valor normalizado de 0 a 1). Cuando cambiamos al estado Falling estando en un estado de vuelo, podemos regresar a él una vez soltemos el botón de caída en picado.

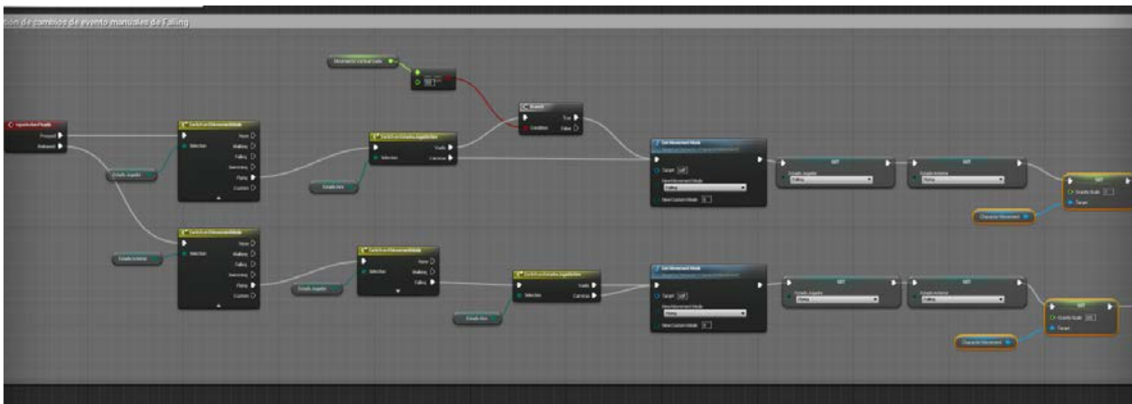


Figura 108. Captura de los cambios de estado manuales del estado Falling (caída en picado).

Fuente: Elaboración propia.

5.2.1.8.1.2. Cambios de estado manuales de vuelo

Para realizar el cambio a estado de vuelo, lo que hacemos primero de todo, igual que en el cambio de estado anterior, es comprobar si estamos en un estado que puede realizar el cambio a vuelo, y si es así, lo realizamos, y cambiamos la gravedad a la que está sometida el jugador a 0.

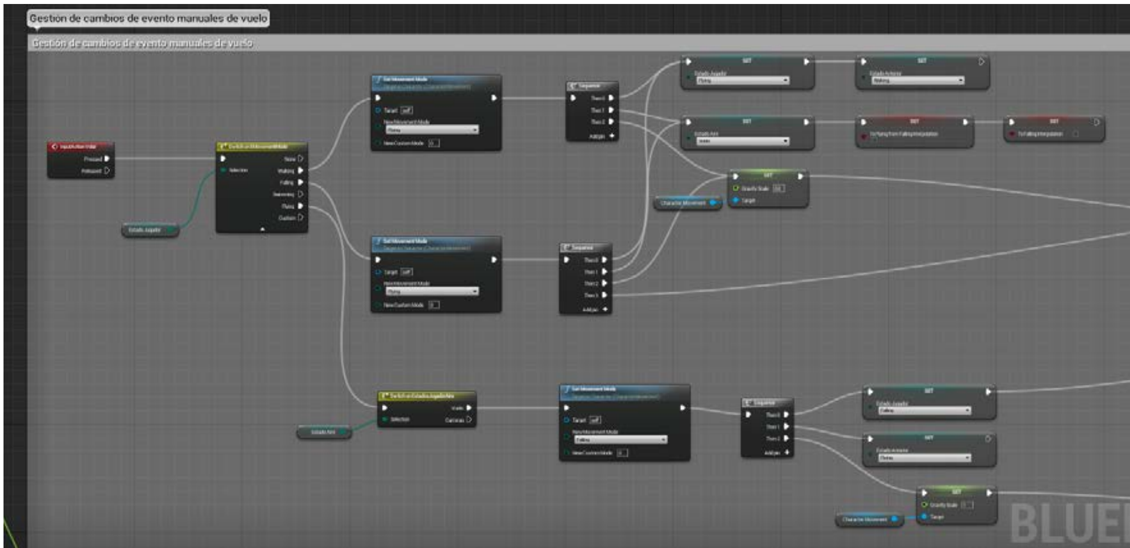


Figura 109. Captura de los cambios de estado manuales al estado de vuelo.

Fuente: Elaboración propia.

Cuando volvemos a pulsar la tecla de cambio a vuelo, si estamos ya volando, pasamos a estado de Falling, que automáticamente pasará a ser Walking si nos encontramos en el suelo, como ya veremos en el punto 5.2.1.8.2.

5.2.1.8.1.3. Cambios de estado manuales de Racing

Para realizar el cambio de estado a Racing, lo que hacemos primeramente, es, como en el resto de cambios de estado, comprobar el estado en que nos encontramos, y si en este caso es vuelo, podremos cambiar.

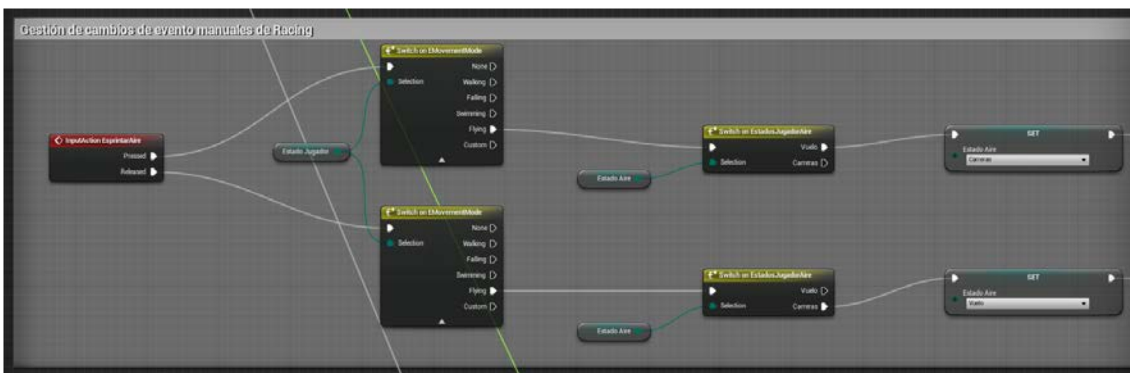


Figura 110. Captura de comprobaciones de cambios de evento manuales para determinar si se puede pasar al estado Racing.

Fuente: Elaboración propia.

Lo que hacemos para cambiar a este estado es:

1. Rotar el personaje (el actor entero) hacia donde está mirando la cámara.
2. Como habremos rotado el actor entero, la cámara también habrá rotado con él, así que debemos guardarnos la rotación que tenía antes de rotar el actor, y aplicarla una vez rotado.
3. Aplicarle un aumento de la velocidad de vuelo para que vaya más rápido al moverse.

Figura 111. Captura del cambio lógico al pasar al estado Racing.

Fuente: Elaboración propia.

4. Que se mueva constantemente hacia delante mientras estemos en este estado aplicándosele una velocidad con cierta fricción (si no le incluimos fricción, cuando dejemos de estar en este estado, va a seguir volando siempre con la misma velocidad impulsado y nunca se detendrá).

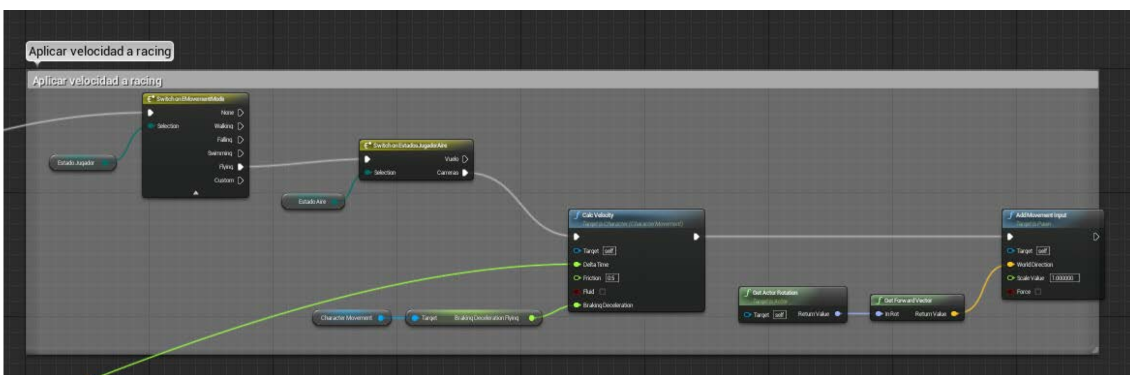


Figura 112. Captura de aplicar una velocidad constante al jugador cuando entra en Racing.

Fuente: Elaboración propia.

5.2.1.8.2. Gestión de cambios de estado automáticos

En cuanto a los cambios de estado automáticos, que como hemos dicho, son los que el jugador no puede controlar directamente con teclas de sus controles.

5.2.1.8.2.1. Gestión de cambio de estado automático entre Walking y Falling

Cuando el personaje salta por un precipicio, pasa al estado de falling, donde lo giramos 90 grados y pasa a no poder realizar acciones como saltar o correr. Para poder hacer esto, comprobamos si el personaje está cayendo midiendo su velocidad en el eje z, además, comprobamos que no esté ascendiendo de la misma forma (controlamos de esta forma que no esté saltando), y también por último, comprobamos el estado en que nos encontramos.

Si el personaje está en walking, añadimos un delay para determinar que simplemente no haya saltado, y volvemos a hacer la misma comprobación, que, si la cumple, pasará al estado falling.

Si el personaje está en el estado falling, comprobamos que ya no está cayendo a ninguna parte (si ha chocado con el suelo), y lo hacemos pasar al estado walking.

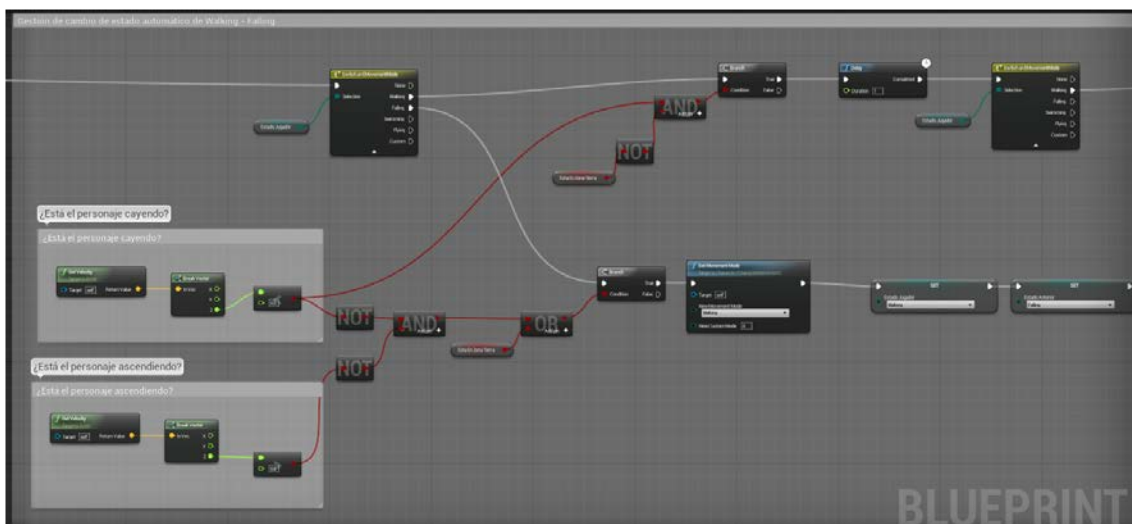


Figura 113. Captura de la gestión de cambio automático de estado entre Walking y Falling (parte 1).

Fuente: Elaboración propia.

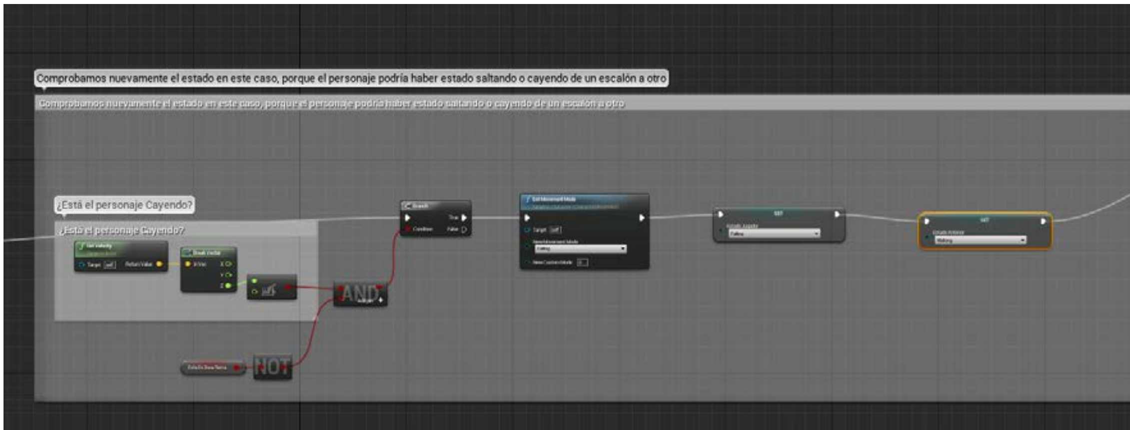


Figura 114. Captura de la gestión de cambio automático de estado entre Walking y Falling donde comprobamos nuevamente que el personaje está realmente cayendo.

Fuente: Elaboración propia.

5.2.1.8.2.2. Interpolaciones de cambio de estado

Las interpolaciones acompañan a cada cambio de estado, y nos permiten realizar una transición más suave entre cambios de estado, de forma que por ejemplo, si el jugador está cayendo y choca contra el suelo, se encuentra con su actor girado 90 grados (la cápsula está girada 90 grados) y tiene que restablecerse este cambio realizado (para que la cápsula vuelva a estar recta y así parezca que se levante), con una interpolación, poco a poco se establece la rotación que deseamos.

Tenemos interpolaciones principalmente ligadas a los cambios de evento, que son:

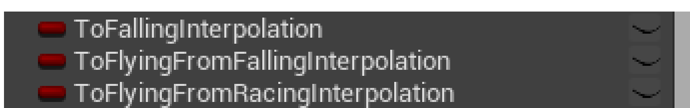


Figura 115. Captura de las variables booleanas que nos permiten pasar a las distintas interpolaciones de movimiento del personaje.

Fuente: Elaboración propia.

5.2.1.8.2.3. Interpolación automática de flying a falling

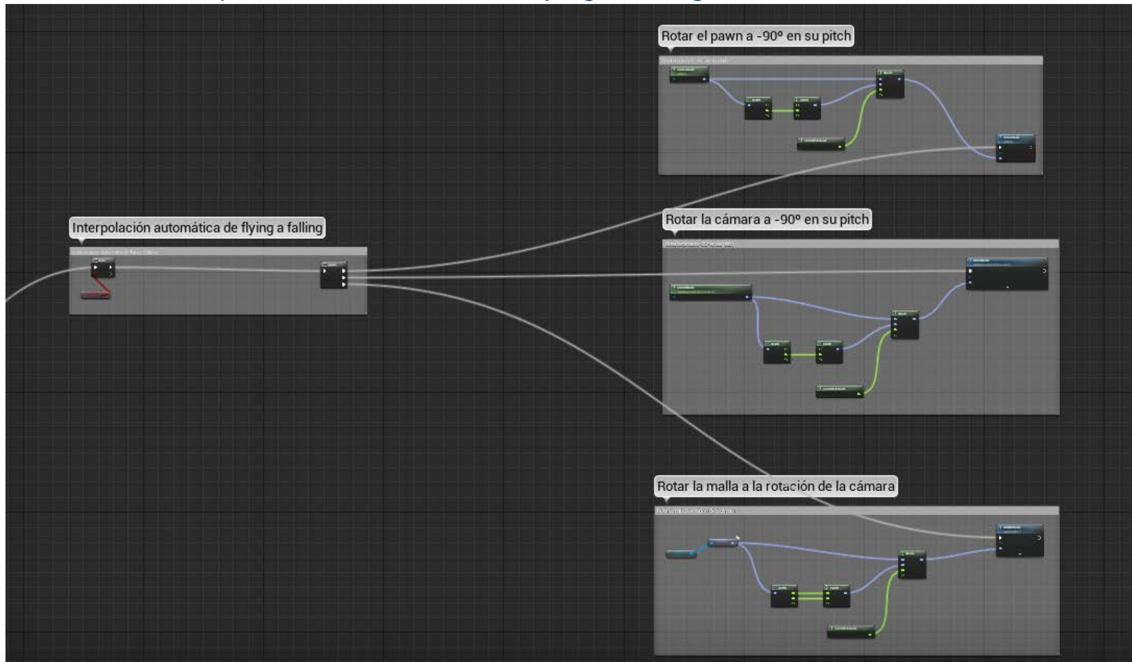


Figura 116. Captura de la lógica general de la interpolación de pasar del estado flying al estado falling.

Fuente: Elaboración propia.

La interpolación de flying a falling consiste básicamente en los siguientes pasos:

1. Comprobar que se debe realizar:

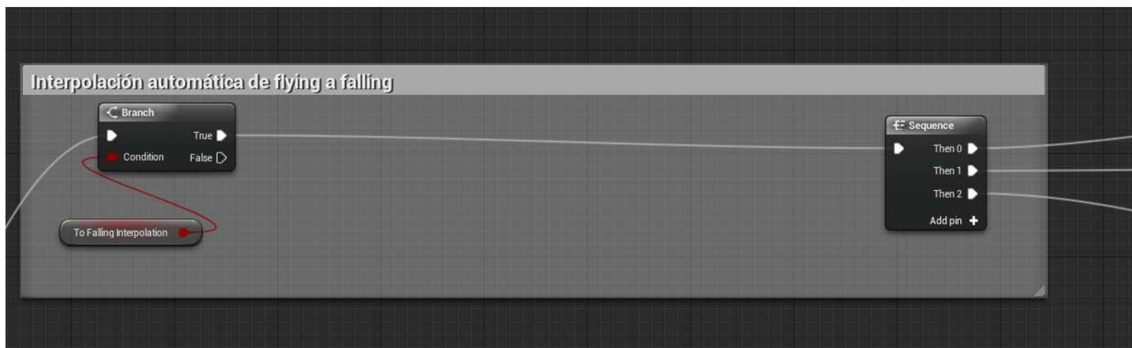


Figura 117. Captura de la comprobación inicial para determinar si se debe aplicar o no la interpolación de flying a falling.

Fuente: Elaboración propia.

2. Rotar el Character -90º en su pitch añadiendo interpolación.

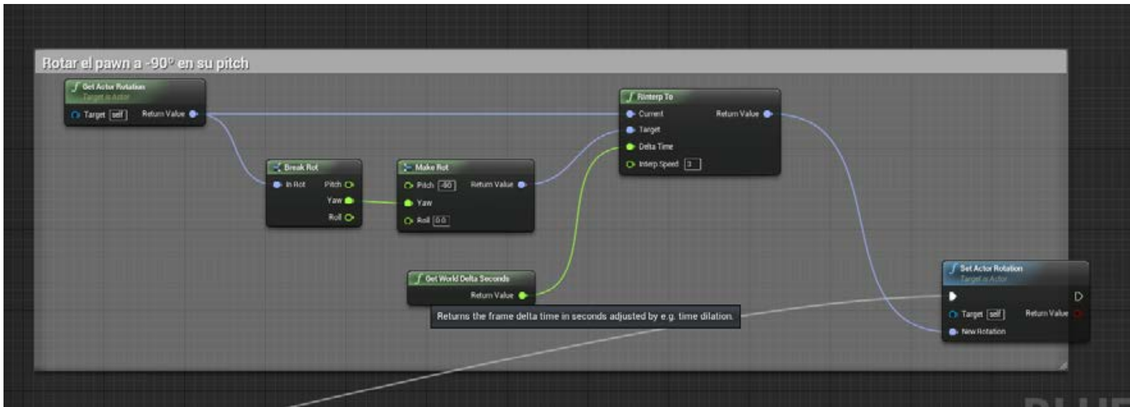


Figura 118. Captura del paso de rotación del pawn para simular que está cayendo hacia abajo.

Fuente: Elaboración propia.

3. Rotar la cámara -90º en su pitch añadiendo también interpolación (si el actor gira -90º, queremos que la cámara gire -90º su punto de visión, que aunque giremos el pawn, va a seguir la cámara enfocando al mismo sitio)

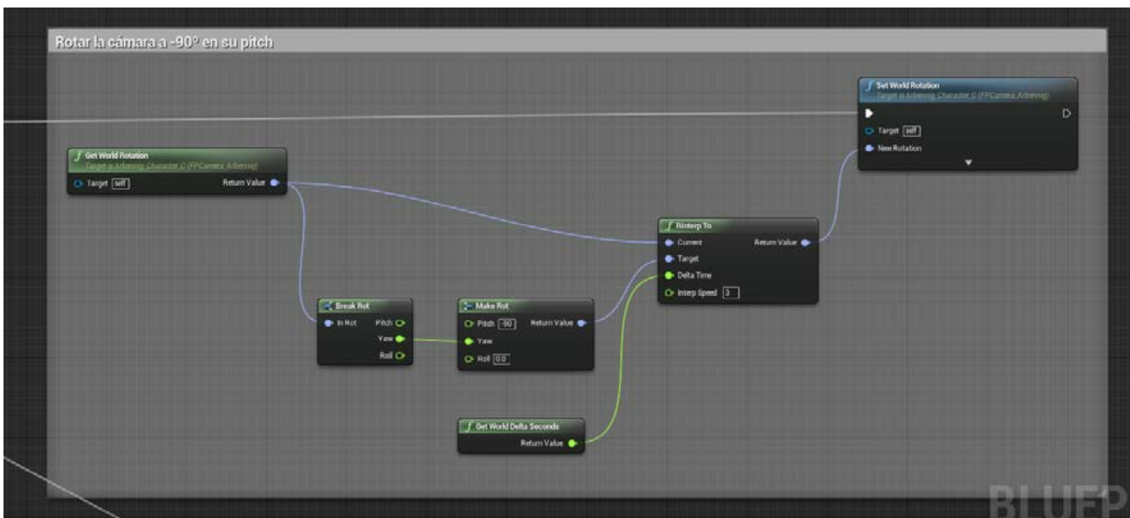


Figura 119. Captura de la rotación de la cámara para simular que el pawn está cayendo hacia abajo.

Fuente: Elaboración propia.

4. Rotar la malla a la rotación que tiene la cámara usando también una interpolación.

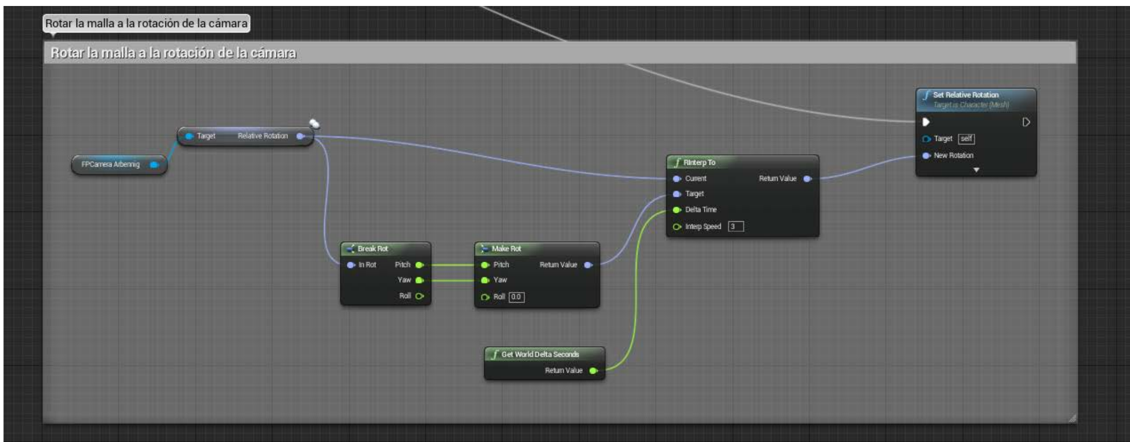


Figura 120. Captura de rotación de la porción de código de rotar la malla a la rotación de la malla.

Fuente: Elaboración propia.

5.2.1.8.2.4. Interpolación automática de falling a flying

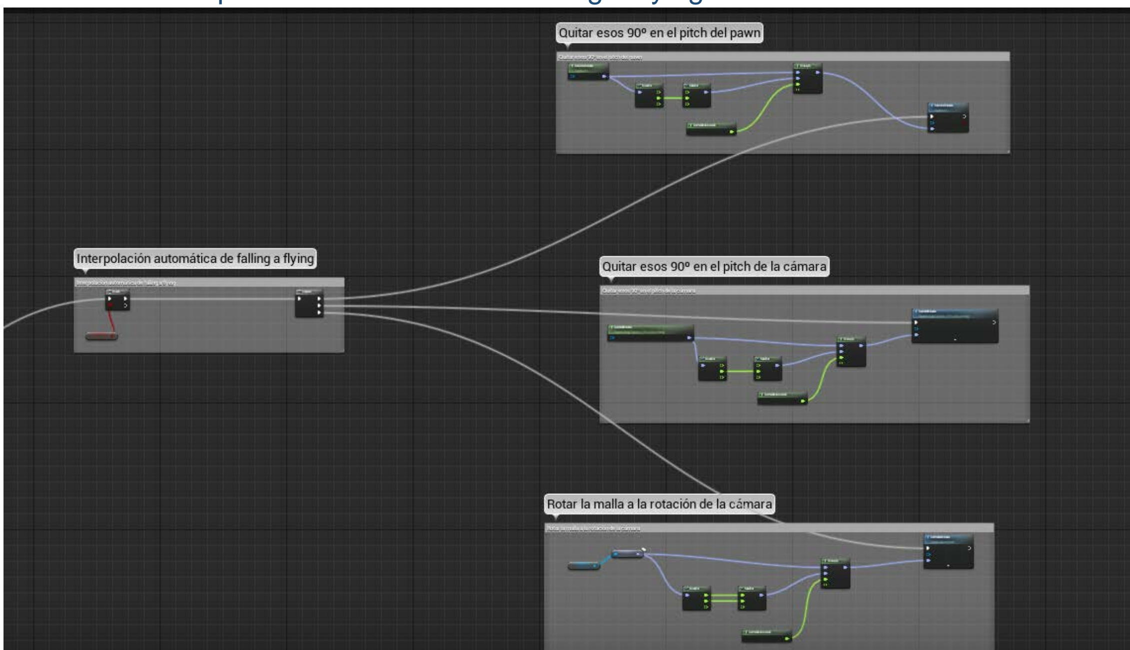


Figura 121. Captura de la lógica general de la interpolación del paso de falling a flying

Fuente: Elaboración propia.

Para la interpolación de falling a flying, los pasos son exactamente los mismos que de flying a falling, pero con la diferencia de que esta vez quitamos los -90º que habíamos añadido, dejándolos en 0.

Figura 122. Captura de comprobación si es necesario realizar la interpolación de flying a falling

Fuente: Elaboración propia.

Figura 123. Captura de restablecer la rotación previa al estado falling del character.

Fuente: Elaboración propia.

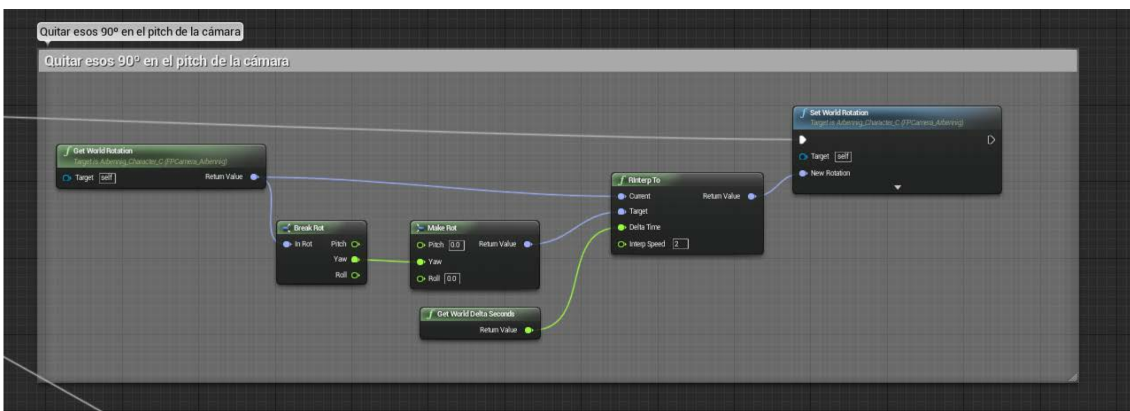


Figura 124. Captura de restablecer la rotación previa al estado falling de la cámara.

Fuente: Elaboración propia.

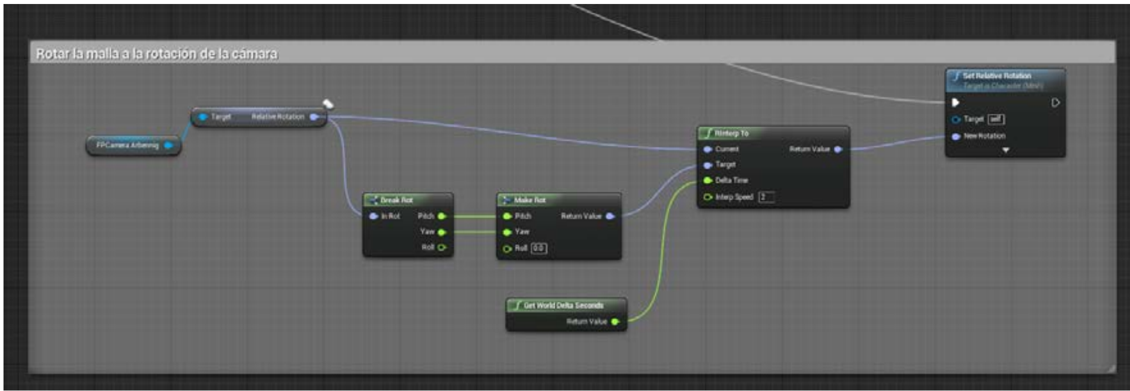


Figura 125. Captura de rotación de la malla a la rotación de la cámara

Fuente: Elaboración propia.

5.2.1.8.2.5. Interpolación automática de Racing a Flying

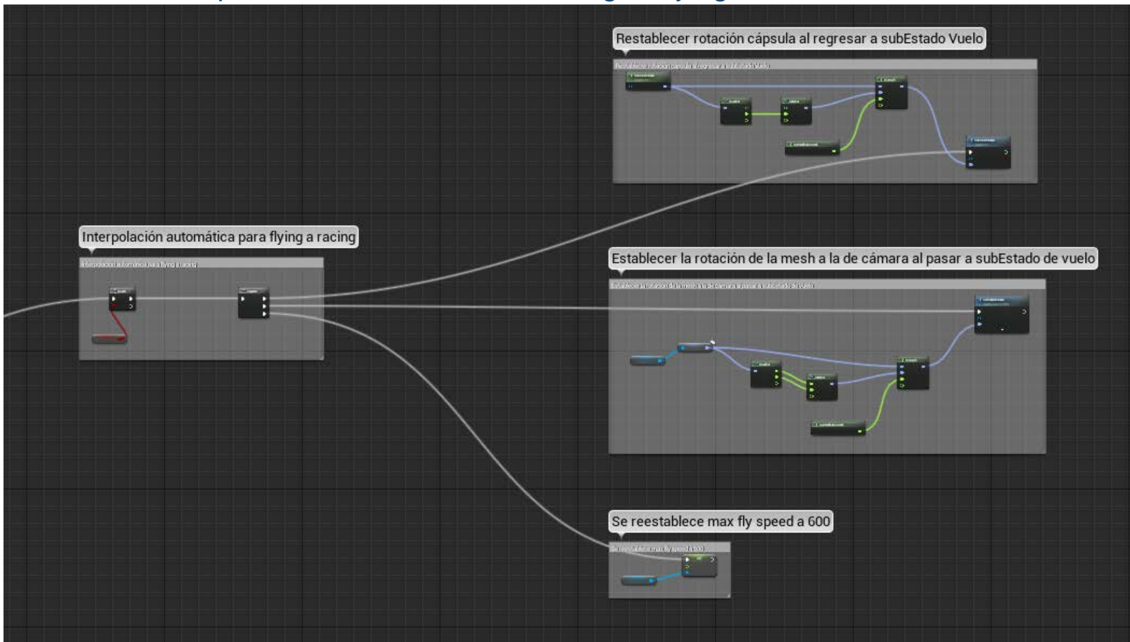


Figura 126. Captura de la lógica general de la interpolación automática de flying a Racing.

Fuente: Elaboración propia.

Para la interpolación de Flying a Racing, el proceso que realizamos es el de restablecer la rotación de la cápsula (el character) a 0 en su pitch y su roll, ya que en Racing estos valores cambian. Luego de esto, hacemos que la rotación de la mesh se asigne a la de la cámara (cuando estamos en Racing, la mesh gira con el actor, mientras que la cámara queda libre), y por último, bajamos la velocidad máxima de vuelo a 600.

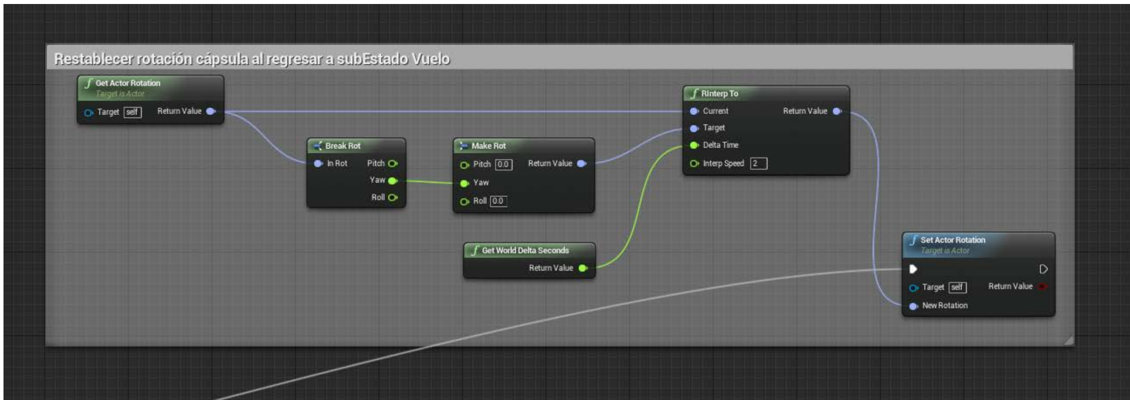


Figura 127. Captura para restablecer la rotación de la cápsula al regresar a vuelo.

Fuente: Elaboración propia.

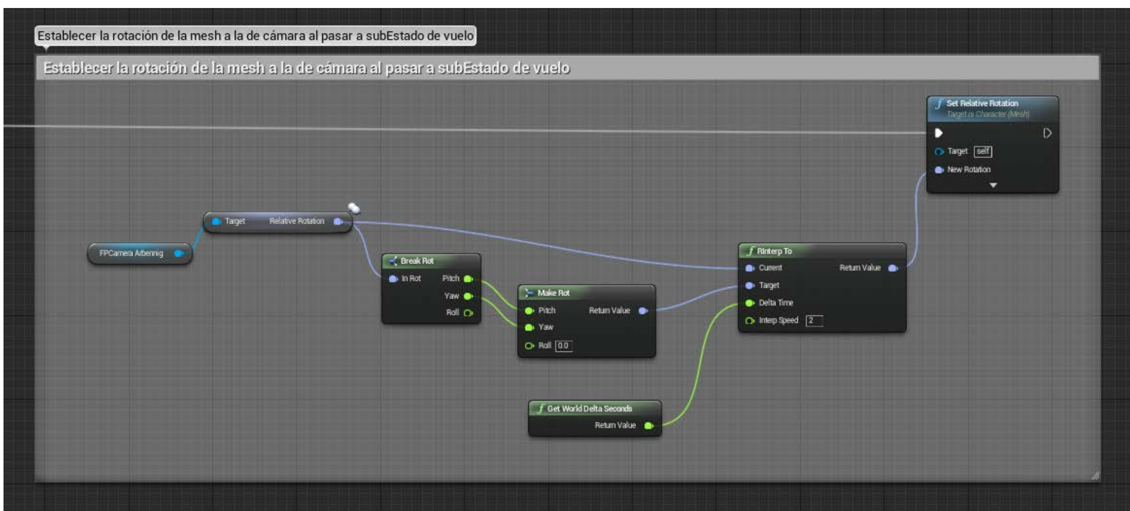


Figura 128. Captura de la lógica de establecer la rotación de la malla a la cámara al pasar a vuelo (no racing).

Fuente: Elaboración propia.

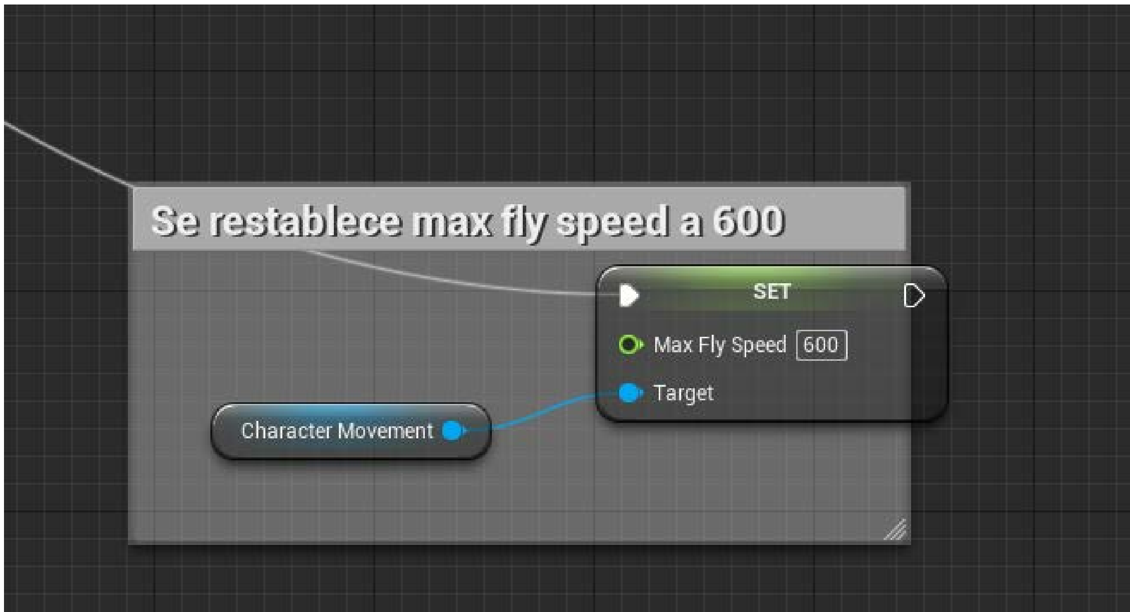


Figura 129. Captura de restablecer la velocidad máxima de movimiento a 600, en lugar de 10000 como tenemos en Racing.

Fuente: Elaboración propia.

5.2.1.8.2.6. Activar interpolaciones

Como último detalle a añadir en cuanto a las interpolaciones, las activamos para realizar los cambios anteriormente nombrados tanto en cambios de estado automáticos como en manuales, y lo hacemos colocando una de estas variables a true, y el resto a false.

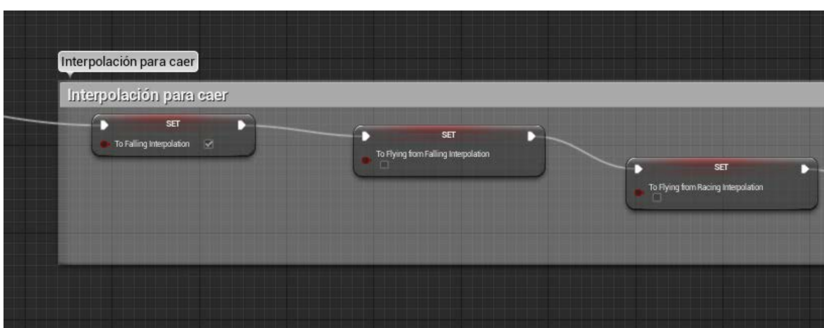


Figura 130. Captura de la activación de la interpolación de caer (pasando de walking a falling)

Fuente: Elaboración propia.

5.2.1.9. Compatibilizar controles con Oculus Rift DK1.

Para poder terminar de compatibilizar los controles de juego con la cámara de Oculus Rift (recordemos que para el estado Racing ya hicimos algunos cambios en el apartado anterior, y que hemos activado el plugin que nos permite usarlo), en nuestro caso particular, hemos decidido entrar en el player controller, y desde ahí, gestionar que los estados de nuestro character en Walking y Vuelo, de modo que la malla se mueva en función de la cámara del periférico, que estará asignado al PlayerCameraManager.

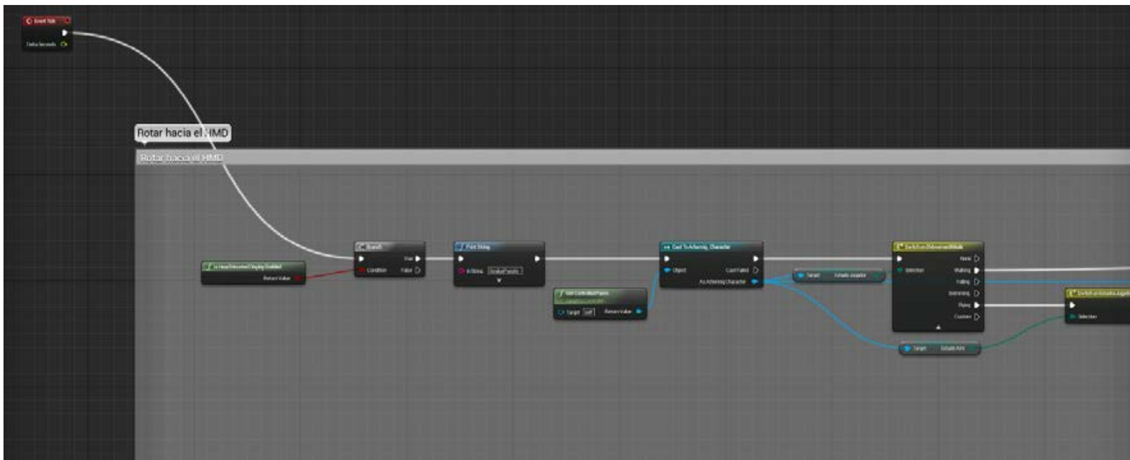


Figura 131. Captura de las comprobaciones necesarias en el PlayerController para que la cámara rote hacia donde gire el dispositivo de Oculus Rift.

Fuente: Elaboración propia.

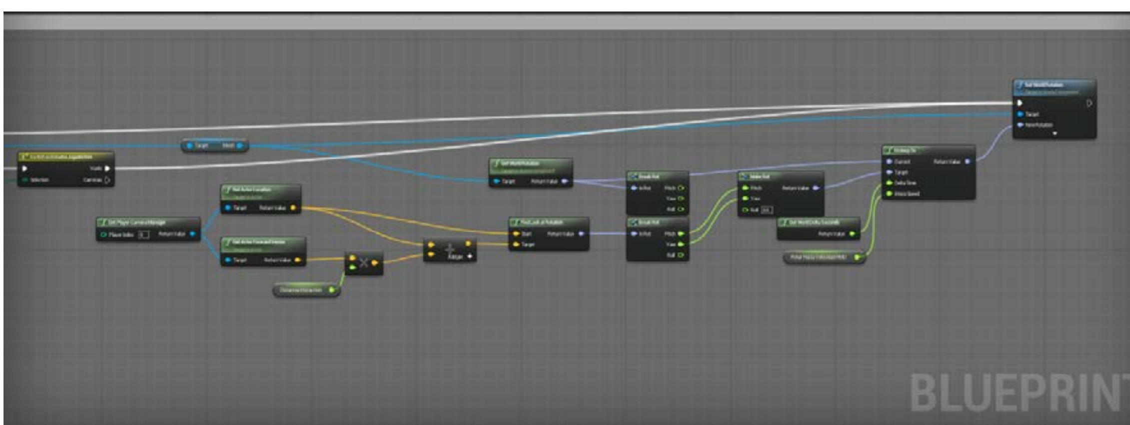


Figura 132. Captura de lógica para que la malla gire hacia la dirección que le indique Oculus en Walking y Vuelo.

Fuente: Elaboración propia.

5.2.3. Desarrollo del nivel de juego.

Figura 133. Captura de Unreal Editor con el proyecto de Arbennig abierto.

Fuente: Elaboración propia.

En este apartado se pretende explicar la realización de todos los componentes del juego que no acarrean una lógica de implementación que interfiera en el flujo de juego. Esto es, importación y posicionamiento de assets, crear terrenos y efector de partículas, e iluminación.

5.2.3.1. Pipeline de creación e importación de Assets.

En este proyecto incluimos tanto assets traídos de otros templates y materiales ofrecidos por unreal, como assets propios que hayamos creado en algún programa de modelado 3D previamente.

A continuación se explica cómo se ha llevado a cabo ambas tareas.

5.2.3.1.1. Importación de Assets desde 3ds max

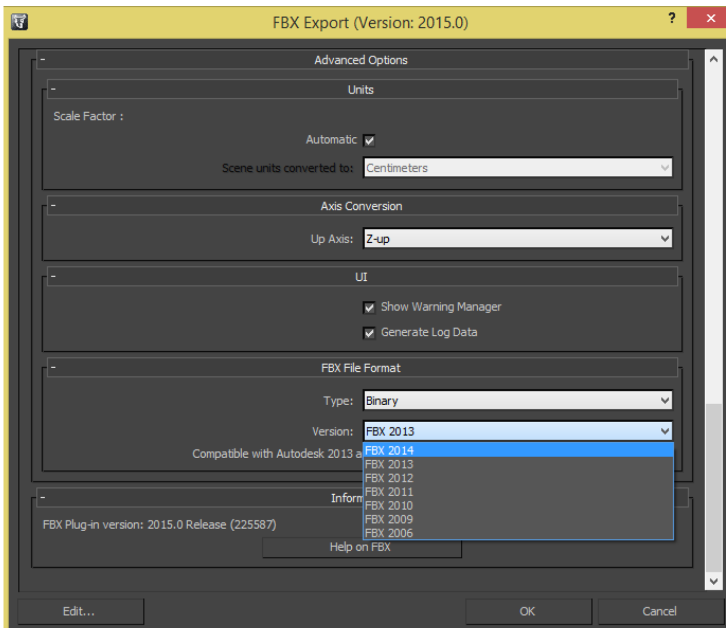


Figura 134. Captura de exportación desde 3ds max seleccionando FBX 2014.

Fuente: Elaboración propia.

Para importar mallas al motor, lo que debemos hacer es exportarlas al formato .fbx en su versión de 2014[26], y tras tener el archivo exportado, arrastrarlo al content browser del motor, o si se prefiere, hacer clic en el botón de import y seleccionar el archivo.

El motor lo reconocerá, y nos mostrará un mensaje de importación, donde simplemente hemos de hacer clic en Import, ya que por defecto es como nos va a interesar en un principio tener todas las mallas importadas.

Si luego, por ejemplo, no nos interesa que tenga una malla en especial colisión, podemos acceder a la static mesh ya importada y elegir remove collision.

Cuando importamos un fbx, se le asignan automáticamente los materiales que este tenía asignados en 3ds max, creándose automáticamente en el motor.

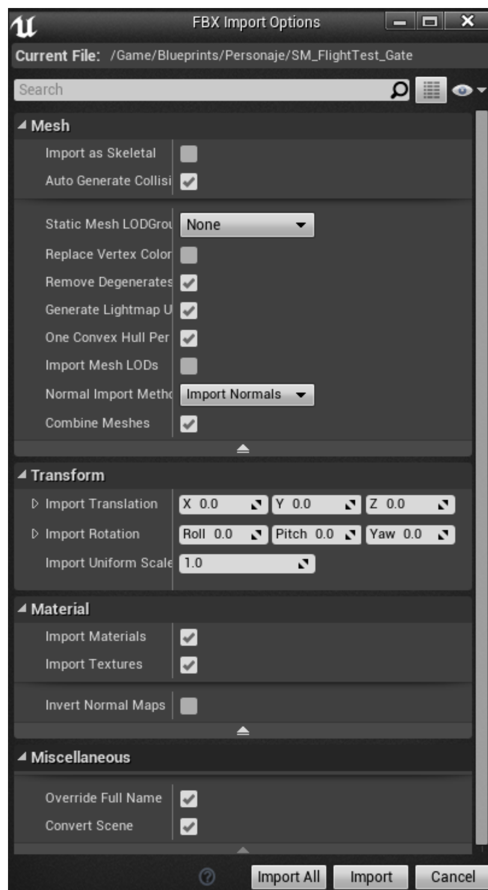


Figura 135. Captura del mensaje que aparece tras intentar importar una malla .fbx

Fuente: Elaboración propia.

Después de haber importado la malla, podemos colocarlas libremente por nuestro escenario arrastrándolas o haciendo spawn de ellas desde código.

5.2.3.1.2. Importación de assets desde otro proyecto de UE4

Si queremos importar assets de otros proyectos de Unreal Engine 4, por ejemplo de templates que se ofrecen para aprendizaje y muestra de características del motor, los cuales son gratuitos, podemos hacerlo haciendo clic sobre el asset, o conjunto de assets, y eligiendo entre sus opciones la opción Migrate, que nos pedirá un directorio donde migrarlos.

Para poder hacer esto es necesario que migremos assets de la misma versión del motor, ya que assets de versiones posteriores pueden no funcionar y dar problemas.

5.2.3.2. Creación de terrenos (landscape) del proyecto.

La creación de un terreno con el motor la podemos realizar utilizando la herramienta landscape que incluye el editor.

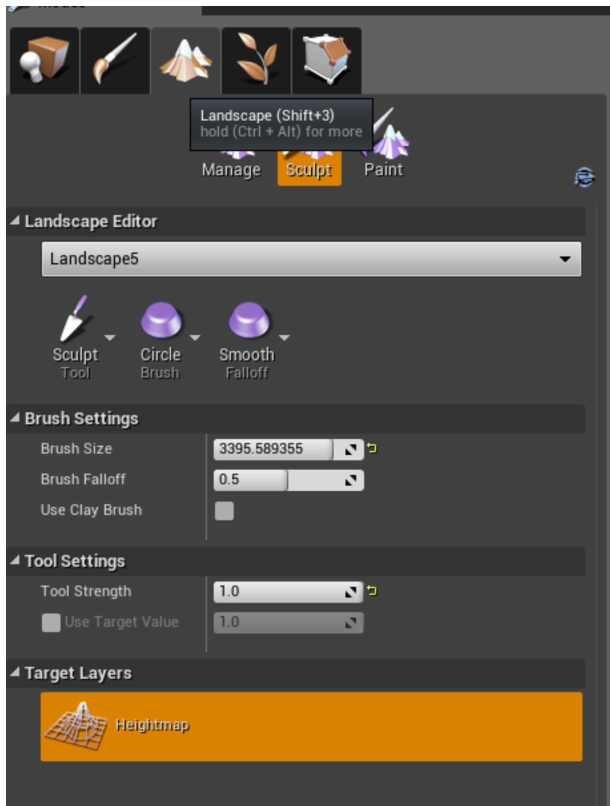


Figura 136. Captura de la herramienta landscape del motor.

Fuente: Elaboración propia.

Su funcionamiento es sencillo, utilizando la opción de manage, podemos crear nuevos landscape, donde nos aparecen distintas opciones de creación y selección del material a utilizar en este.

Mediante la herramienta Sculpt podemos "modelar" este landscape, que inicialmente es un plano, lo cual podremos hacer de forma libre con los brush que ofrece, o cargar un mapa de alturas y que haga la forma deseada automáticamente.

Por último, con Paint, podemos pintar el landscape con materiales.

En este proyecto, hemos utilizado la herramienta landscape para crear las distintas islas no flotantes que forman el mundo de Delfrydoland. Para el laberinto de la isla donde se encuentra el tótem 5, se ha creado un mapa de alturas básico y se ha retocado luego haberlo creado en el motor.

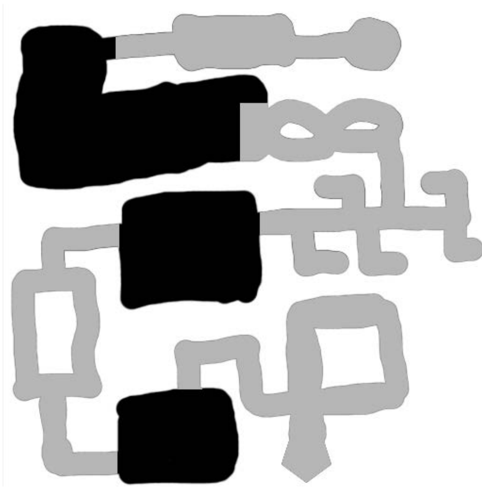


Figura 137. Mapa de alturas del laberinto de la isla 5.

Fuente: Elaboración propia.

En este mapa de alturas, que debe estar en escala de grises, el negro significa profundidad máxima, mientras que el blanco significa que no se desea aplicar nada de profundidad.

Figura 138. Landscape del laberinto una vez se ha creado con la herramienta del mapa de alturas de landscape.

Fuente: Elaboración propia.

El resto de islas han sido creadas de forma libre hasta acabar con una forma adecuada.

5.2.3.3. Crear un efecto de partículas de chispas de fuego.

Queremos utilizar un sistema de partículas de chispas de fuego como elemento decorativo en el nivel. Estas chispas dan una mayor sensación de realismo a los adornos con forma de volcán que suele aparecer junto a cada tótem.

5.2.3.3.1. Anatomía de un sistema de partículas.

Antes de ponernos a crear sistemas de partículas, es necesario que conozcamos las distintas partes de un sistema de partículas en Unreal Engine 4.

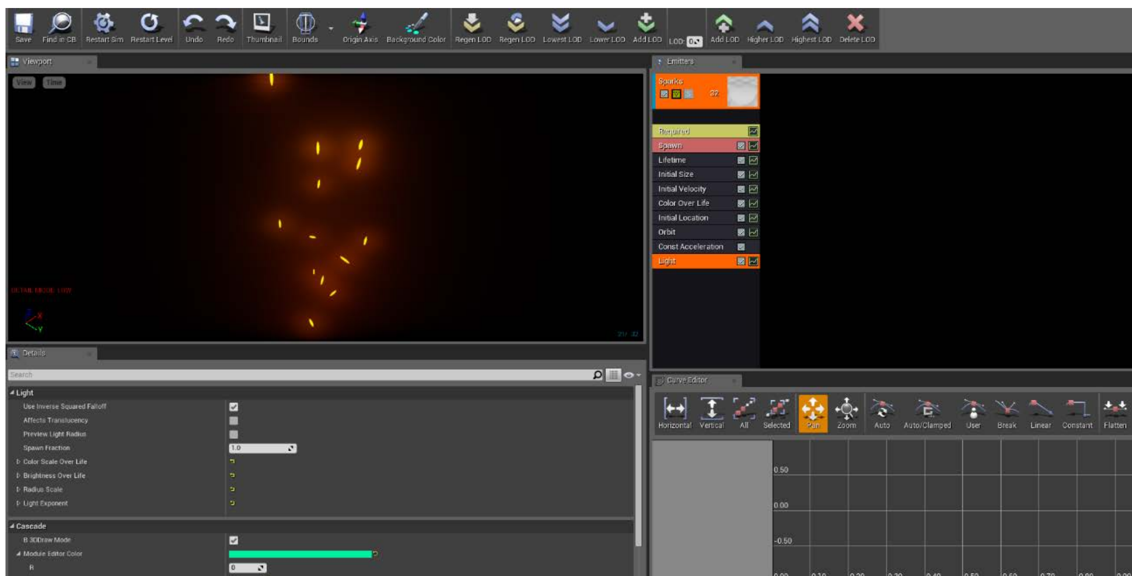


Figura 139. Captura del sistema de partículas que se ha creado.

Fuente: Elaboración propia.

Dicho esto, un sistema de partículas en este motor está compuesto por emisores (emitters), y un emisor está formado por un bloque de emisión, y una lista de módulos.

El bloque de emisión contiene distintas propiedades que nos ayudan a crear distintos componentes del sistema de partículas. Estas propiedades son la de visibilidad del bloque (para mostrar o no el bloque), la forma de emisión para ver en la ventana preview (puede ser la forma normal, solo con puntos, solo con ticks, como se vería en el escenario (lit), y no mostrar nada), y el botón "Solo" (cuando tenemos múltiples emisores, nos permite solo visualizar el que marquemos con Solo).

En cuanto a la lista de módulos, tenemos Required (nos va a permitir cambiar propiedades como el material de emisión, también se puede cambiar la dirección y rotación en la que se emite), Spawn (nos permite configurar la cantidad de partículas que se van a emitir (en el desplegable Rate->Distribution->Constant)), Lifetime, Initial Size, Initial Velocity, y Color Over Life, como módulos por defecto, pero es posible añadir o eliminar módulos haciendo clic derecho sobre la lista de módulos (todos los módulos, a excepción de Required, se pueden quitar del emisor). Los

módulos permiten hacer "expose" de cada uno de ellos, que nos permite cambiar sus propiedades a lo largo del tiempo en una curva de edición. El flujo de trabajo con respecto a cada módulo es primero acceder a las propiedades, y, luego de esto, si es necesario tocar la curva de edición, pasar a ella entonces.

Un sistema de partículas se puede componer de varios emisores. Para crear más emisores de partículas, se puede hacer con clic derecho sobre la ventana de emitters, y nos permite esa opción (create sprite emitter), para eliminarlo, se hace clic derecho sobre el emisor, y en la pestaña de emitters, nos permite distintas opciones, entre las que encontramos Delete emitter. Cada emisor se separa en columnas, que se llaman emitter columns.

Ahora que se ha explicado el funcionamiento de los sistemas de partículas en el motor, se procede a la creación del sistema de partículas que utilizaremos en el juego, vamos a crear unas chispas de adorno, que son del tipo sprite emitter.

5.2.3.3.2. Implementando el sistema de chispas.

Antes de empezar a crear el sistema de partículas, hay que tener en cuenta que realmente se compone de dos partes, por una parte tenemos lo visual (cómo vemos el efecto en sí), y por otra, su comportamiento (la lógica que tiene detrás, compuesta por lo descrito anteriormente de su anatomía).

El primer paso para la creación de un sistema de partículas, es crearlo en el content browser, para ello, se puede hacer tal y como se crean blueprints (clic derecho -> Particle System), y darle un nombre.

Lo siguiente, es establecer sus parámetros básicos, esto es, a través del módulo Required.

Las chispas, recordemos, necesitan un material base de emisión. Este material base lo creamos nosotros, en nuestro caso particular, será un material simple:

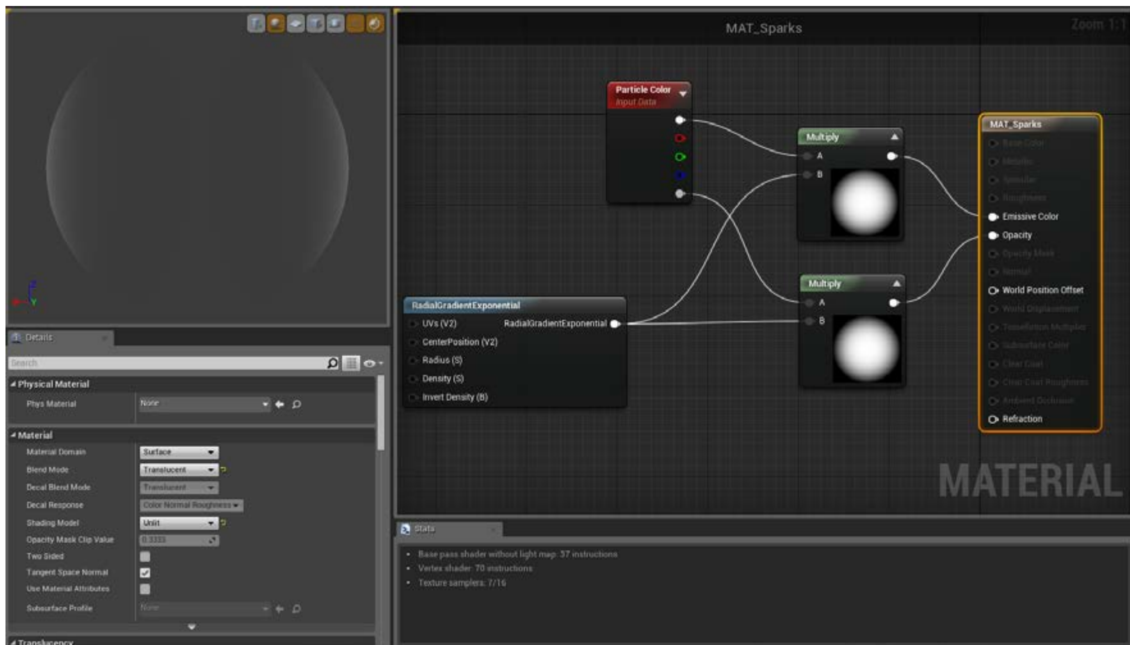


Figura 140. Captura del material que utiliza el sistema de partículas creado.

Fuente: Elaboración propia.

Lo particular de este material respecto a un material normal, es que recibe datos con un evento InputData de tipo ParticleColor. Por otro lado, para poder crear el material correctamente, necesitamos colocar en las propiedades del material, que su modo de blend es translucent (así se nos habilita la propiedad de opacidad), y el modo de shading lo colocamos en unlit, para que nuestro sistema de partículas sea lo menos agresivo posible con la CPU.

Una vez tenemos el material, volviendo a nuestro sistema de partículas de chispas, lo cambiamos en el módulo Required.

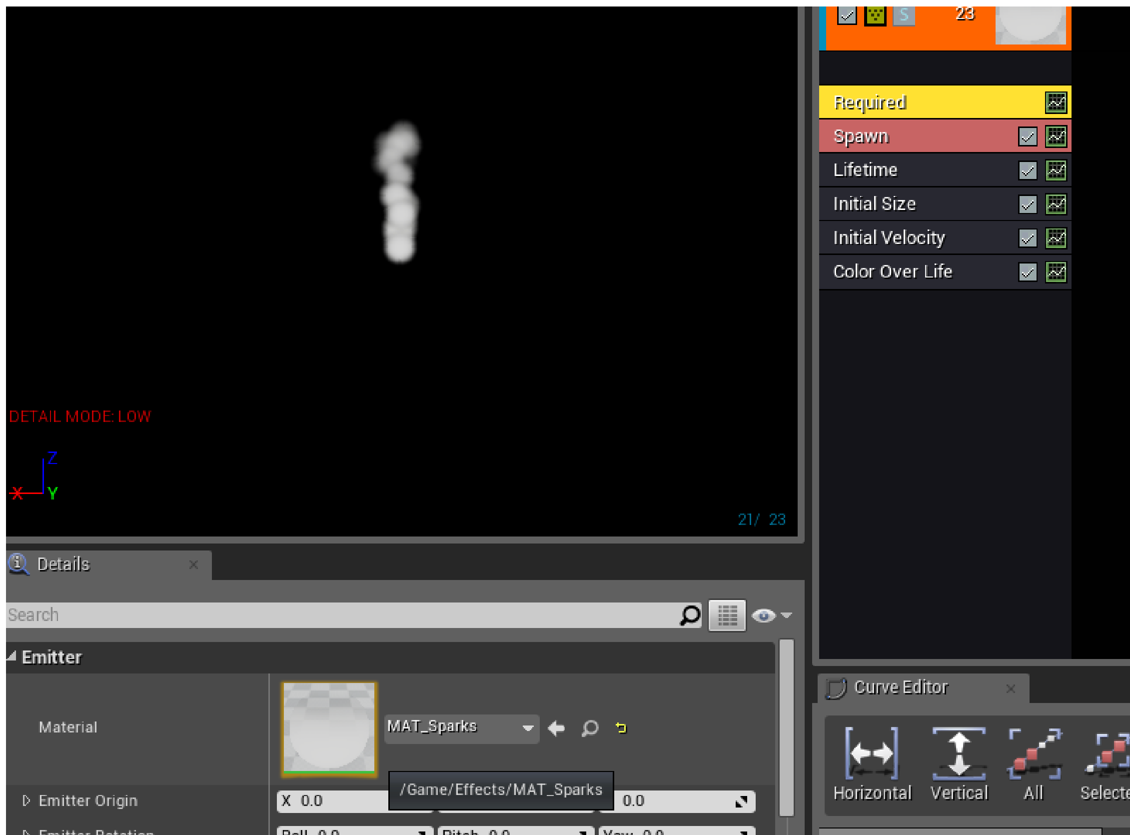


Figura 141. Captura del sistema de partículas tras aplicar el material creado.

Fuente: Elaboración propia.

En nuestro caso, queremos que la emisión esté orientada hacia arriba, así que no vamos a modificar la rotación del emisor, y tampoco queremos cambiar su origen.

Colocamos el Screen Alignment en PSA Velocity. El Screen Alignment sirve para cambiar las capacidades de cada partícula emitida. Por defecto está en PSA Square, que emite un único polígono cuadrado por cada partícula emitida, con PSA Velocity queremos que las partículas cambien, y que cambien con la velocidad si fuera necesario.

Figura 142. Captura de la opción de Screen Alignment

Fuente: Elaboración propia.

Pasamos al módulo Spawn, dado que no nos hace falta tener las partículas por defecto (20), sino que con menos basta, lo cambiamos a 10.

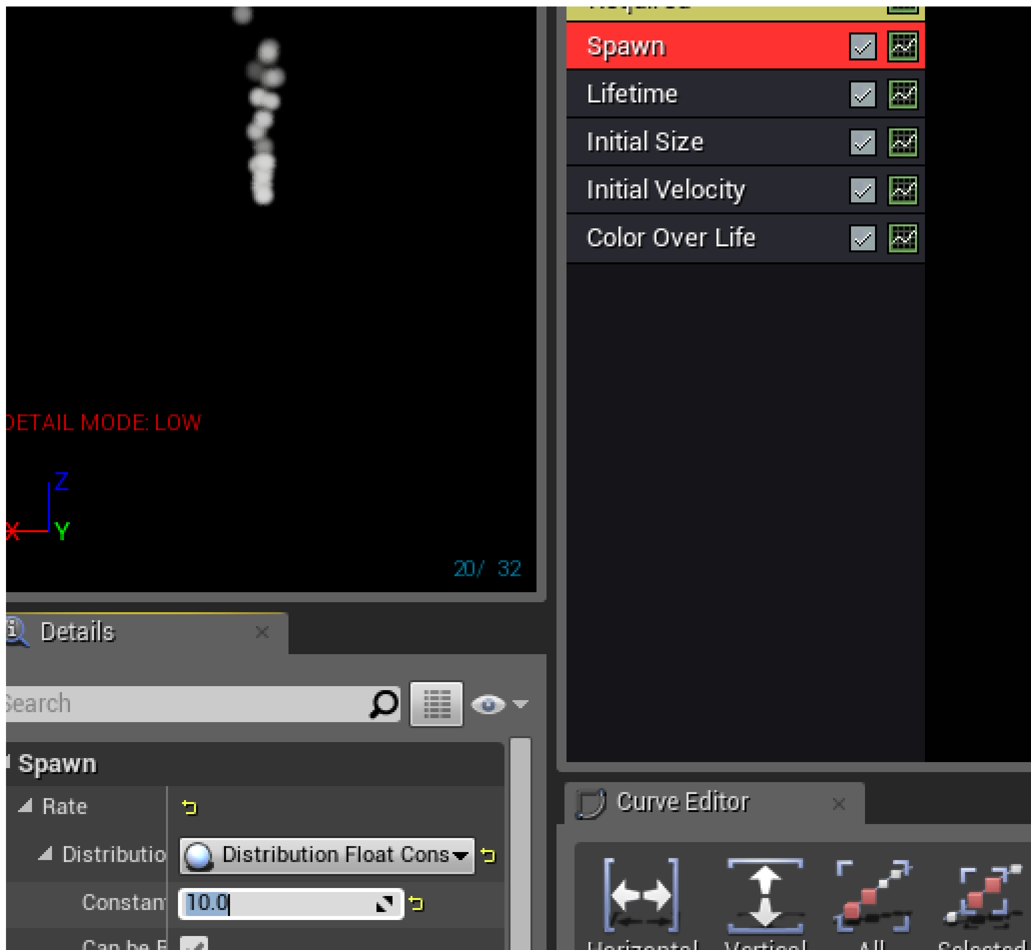


Figura 143. Captura del sistema de partículas tras reducir su cantidad a la mitad por defecto.

Fuente: Elaboración propia.

Seguimos al módulo Lifetime, en este módulo tenemos el tiempo mínimo y máximo de duración, por defecto nos aparecen ambos a 1. Cambiamos el tiempo máximo a 3, para que algunos duren más tiempo y otros menos, y no sea tan uniforme.

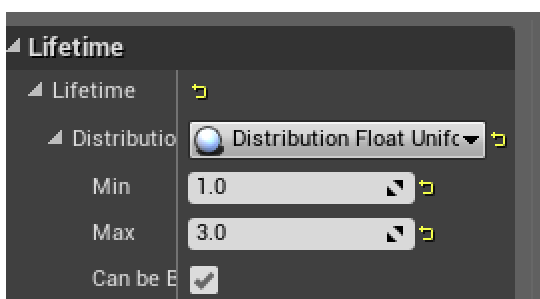


Figura 144. Captura del tiempo de vida mínimo y máximo que hemos decidido asignar.

Fuente: Elaboración propia.

En el módulo Initial Size, que nos sirve para cambiar el tamaño de las partículas, vamos a darles un tamaño mínimo y máximo igual, y de valores (2,10,2), que lo hace lucir más como unas chispas.

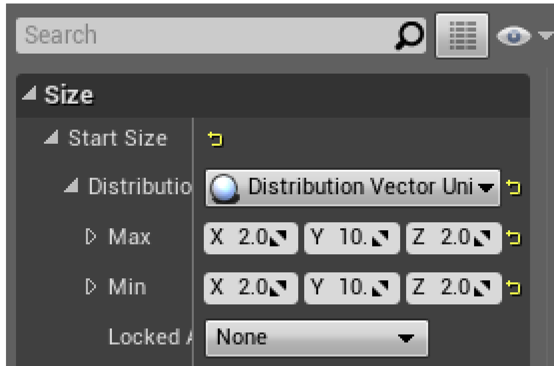


Figura 145. Captura del tamaño inicial de las partículas modificado a uno que más se adecúa al deseado.

Fuente: Elaboración propia.

Ahora pasamos a configurar el módulo Color Over Life, que sirve para modificar el color de las partículas a lo largo de su periodo de vida. Para hacer esto, tenemos un vector con dos puntos, 0 y 1. El valor inicial del color está en 0, y el final en 1.

Figura 146. Captura del color elegido que deben tener las partículas.

Fuente: Elaboración propia.

Colocamos como valor de salida (out val) en 0 lo siguiente (50,5,0), y en el punto 1 colocamos (50,10,0). Cuando pasamos de 1 en cada posición del vector RGB que nos aparece, empieza a crear un efecto Glow.

Figura 147. Captura del resultado tras sobrecargar el valor, se aprecia el efecto Glow.

Fuente: Elaboración propia.

El siguiente paso es añadir un módulo más al bloque de emisión, el módulo Initial Value (lo añadimos con clic derecho en la lista de módulos). Con este módulo vamos a hacer que el origen de las chispas aparezca “fuera”, de modo que las partículas empiezan a verse más distribuidas. Esto lo hacemos en nuestro caso cambiando la distribución máxima en (30,30,0) y la mínima en (-30,-30,0).

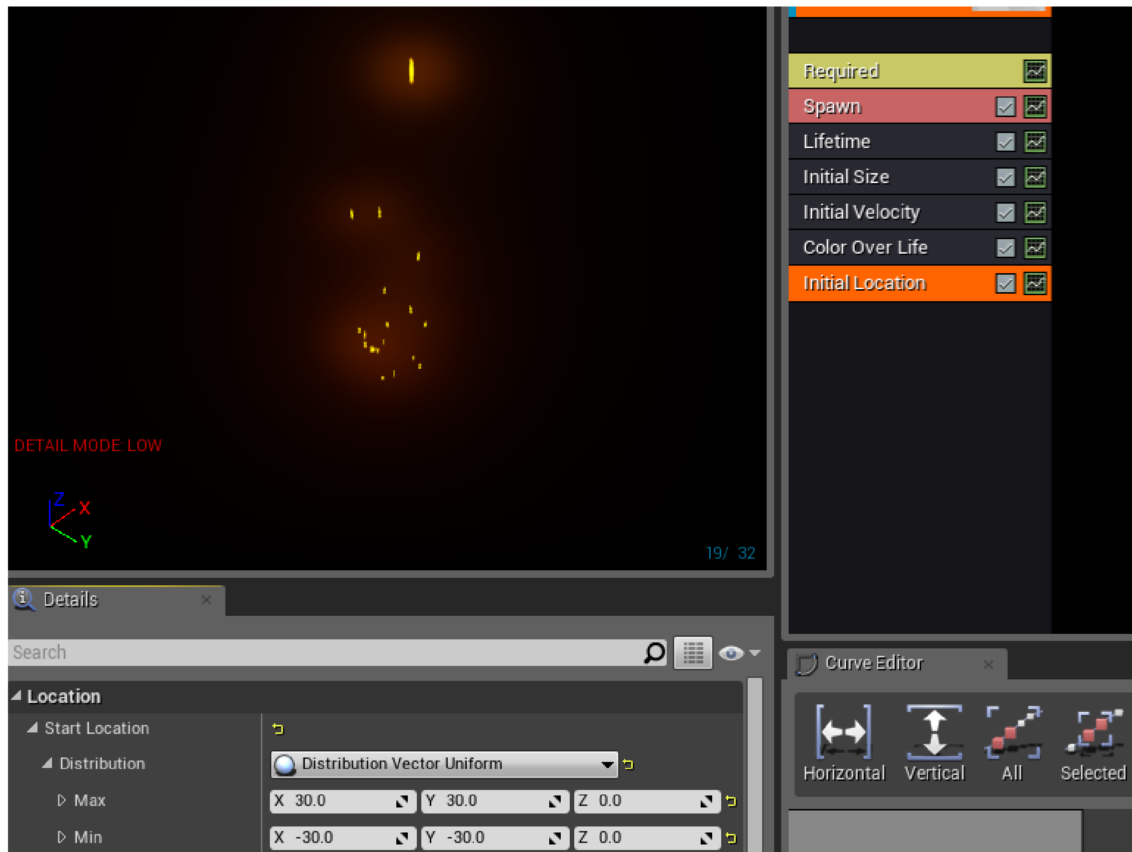


Figura 148. Captura del sistema de partículas tras modificar su localización inicial.

Fuente: Elaboración propia.

Luego de esto, el objetivo es que parezcan más unas chispas, con una distribución algo más caótica. Para esto, vamos a añadir el módulo orbit. Además, queremos que dé la sensación de que suben hacia arriba aumentando la velocidad hacia arriba, esto lo hacemos añadiendo el módulo const acceleration, e indicándole en la aceleración en Z un valor de 100, así, durante su tiempo de vida, las partículas irán aumentando constantemente su velocidad en Z hasta desaparecer.

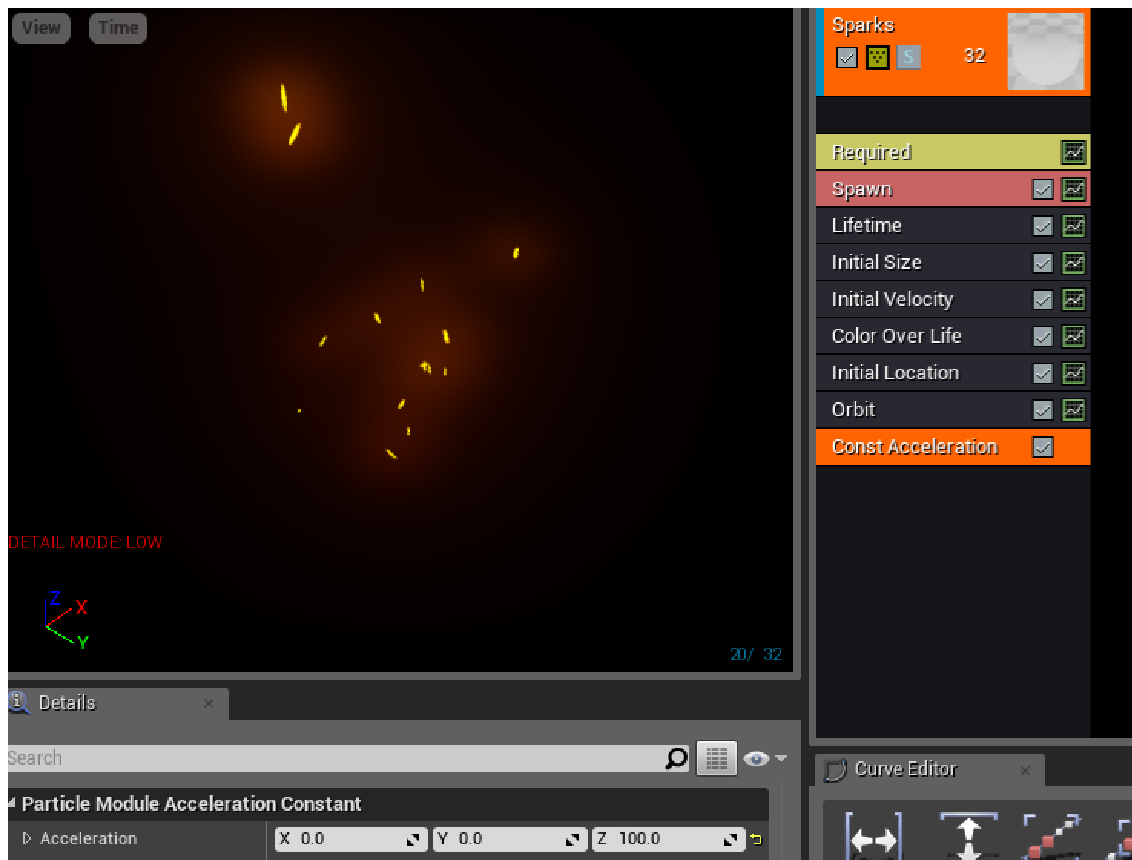


Figura 149. Captura del sistema de partículas tras aplicarle los módulos de Orbit y aceleración constante.

Fuente: Elaboración propia.

Y por último, vamos añadir un módulo Light, para que cada partícula tenga una fuente de luz. No es necesario modificarle parámetros. Y ya tenemos nuestro efecto de partículas. Ahora solo lo arrastramos al escenario para ver el resultado.

Figura 150. Captura del sistema de partículas dentro del juego.

Fuente: Elaboración propia.

5.2.3.4. Añadir la iluminación.

En el juego se ha decidido utilizar dos tipos de iluminación ofrecida por el motor, point lights, y Light Source.

Mientras que los point light son luces puntuales que se han colocado por el mapa para dar mayor iluminación a algunos assets, y para ayudar al jugador a entender que puede interactuar con algún NPC o trigger, el light source se ha utilizado como luz global del nivel, que mantiene todo el escenario iluminado con una intensidad de luz aceptable.

Para realizar los cálculos de luz de forma correcta, se ha utilizado un componente llamado lightmass volumen, que calcula todos los fotones dentro de un entorno cerrado indicado.

Las luces utilizadas no contienen modificación especial, en algunas se ha modificado la intensidad para que produzcan más luz, mientras que en otras, la intensidad por defecto se ha considerado la ideal.

Su forma de añadir es como todos los actores del juego, se ha añadido al escenario arrastrando con el ratón, o se ha añadido como componente de algunos blueprints de trigger o NPC.

5.2.4. Sonorización de juego.

La sonorización del juego en Unreal Engine 4 se trata de una forma considerablemente sencilla. Para poder reproducir un sonido en el juego, solo es necesario:

1. Importar el audio en el proyecto. El motor de Unreal solo soporta el formato .WAV para audio, el cual, al ser importado, convierte a Sound Wave. Para este proyecto, las conversiones de otros formatos a .WAV se han realizado con el programa Audacity.
2. Llamar a un nodo de reproducción de audio, que bien puede ser Play Sound at Location, Play Sound, Play (audio component), entre otros.

La forma de indicar que queremos que un audio se reproduzca en loop hasta que ordenemos que se detenga, viene definido en las propiedades del mismo audio importado, en un checkbox llamado loop que debemos llamar si queremos que se reproduzca constantemente.

5.2.5. Implementación de la Interfaz de Usuario

El apartado de Interfaz de usuario incluye tanto implementación de menús, como implementación del HUD y un Inventario.

Para la creación de interfaces de usuario en Unreal Engine, hacemos uso de los Unreal Motion Graphics (UMG), cuyo núcleo son los Widgets. Los Widgets son una serie de funciones predefinidas (como botones, sliders, barras de progreso, etc.) que pueden ser utilizadas para crear nuestras interfaces. Estos Widgets son editados en un tipo de Blueprint especial llamado Widget Blueprint, que contiene dos pestañas para la creación de interfaces:

La pestaña Designer (diseñador), que se trata de la capa visual de la interfaz.

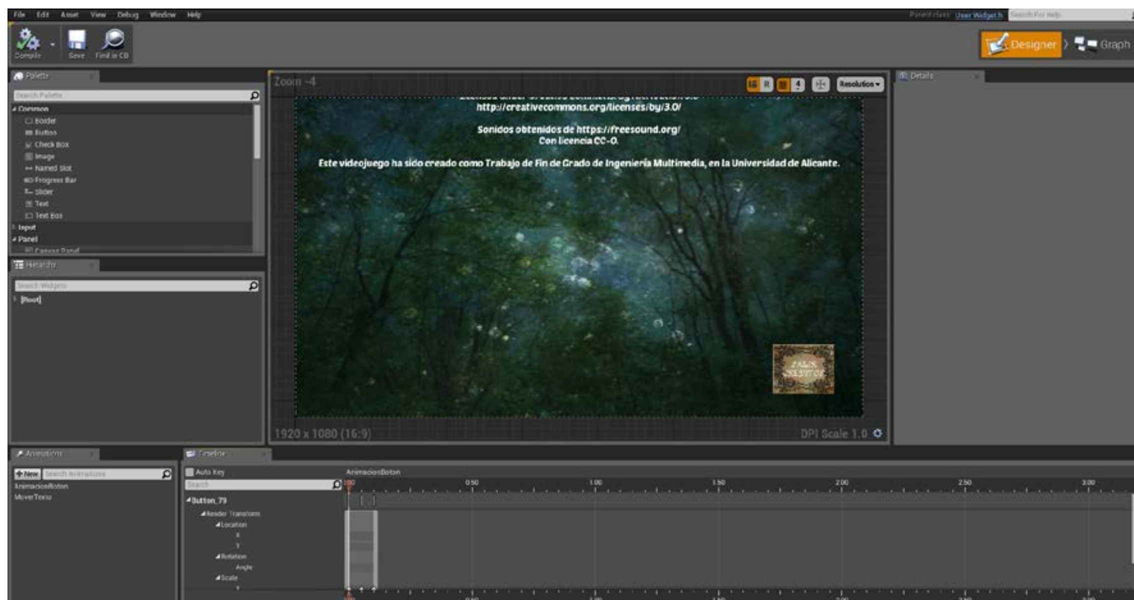


Figura 151. Captura de la pantalla Designer de un widget blueprint.

Fuente: Elaboración propia.

La pestaña Graph (grafo), que proporcionara la funcionalidad a los widgets utilizados.

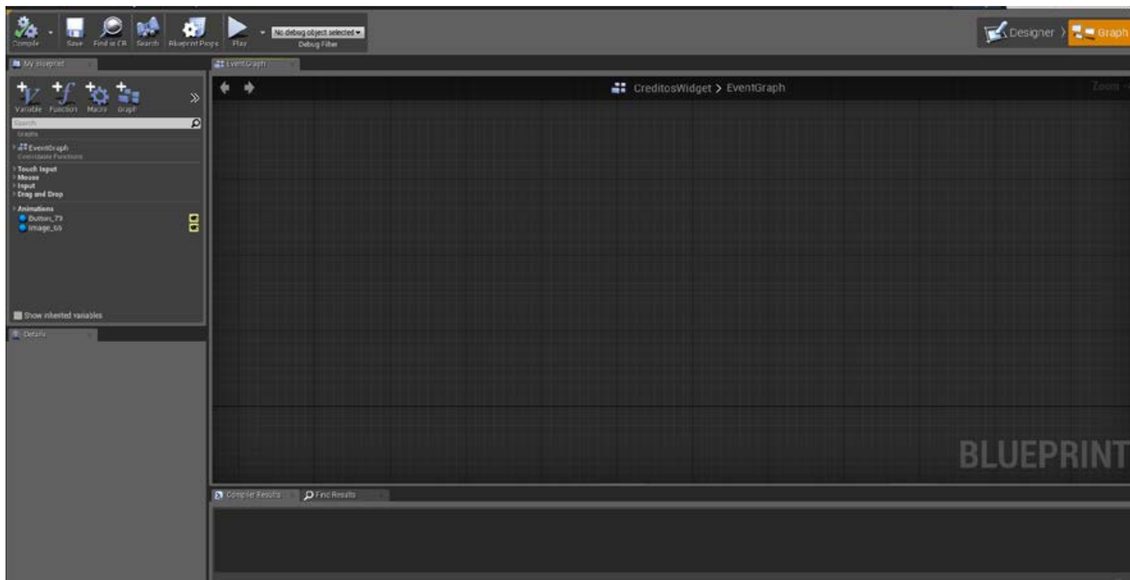


Figura 152. Captura de la pestaña Graph de un Widget Blueprint.

Fuente: Elaboración propia.

Es decir, con Designer añadimos elementos, creamos animaciones y personalizaciones de los elementos genéricos en un lienzo o canvas, y a través del Graph se les otorga de una lógica que, por ejemplo, permite cargar un nivel cuando se pulse un botón de menú de nueva partida.

Además de lo anterior, tal como funciona el posicionamiento de los elementos en un Widget Blueprint en otros motores de videojuegos como Unity, contamos con unos Anchors (anclajes), para poder situar nuestros elementos del menú como deseemos en función del tipo de pantalla. Dado que en nuestro caso particular, no nos vamos a encontrar con pantallas que no sean de de formato 16:9 o 4:3, se ha optado por realizar anclajes relativos en la pantalla. Esto es, que sea una pantalla de 800x600, o una pantalla de 1920x1080 de resolución, los elementos en pantalla van a ocupar siempre la misma posición dentro del marco de la pantalla, escalándose.

5.2.5.1. Pipeline de la creación de interfaces de usuario con Widget Blueprints

Independientemente de si se trata de un Menú, un HUD, o incluso un Inventario, el pipeline básico a la hora de utilizar cada IU es el mismo. El cual consiste básicamente en lo siguiente.

1. Creación de un Widget Blueprint.

2. En un blueprint ya existente, crear desde su Event Graph un Widget de la clase del Widget creado en el paso anterior.

3. Crear una variable para almacenar dicho Widget creado. La forma más rápida para esto, aunque no la única, es desde el Widget creado, que tiene como parámetro de salida el objeto creado, a través de este, se arrastra hasta que aparezca una línea conectora, y con Promote to variable, creamos una variable para el mismo. Guardamos la variable para

mayor comodidad, puesto que si no se guarda, cada vez que se quiera acceder a este objeto, se debe tirar una línea desde el nodo del Widget creado.

4. Cuando se quiera añadir el Widget a la pantalla de juego, se coloca como Get la variable que hemos creado en el paso 3, y a través de ella, se llama al nodo Add to Viewport, que mostrará el Widget de la Interfaz.

5. Por otro lado, cuando se quiera eliminar el Widget de la pantalla de juego, al igual que en el paso anterior, se coloca como Get la variable que hemos creado en el paso 3, y a través de ella, se llama al nodo Remove From Parent.

5.2.5.2. Uso en el juego

Como queda reflejado en el Documento de Diseño del videojuego, para Arbennig construimos varios menús de juego (menú principal, de opciones, de créditos, de confirmación, de pausa, de ayuda), y una interfaz de HUD, que incluye tanto barra de estado de personaje (vida, energía, y llaves conseguidas), como un inventario.

Cabe recalcar que los mapas donde encontramos menús son el mapa de menú principal, donde la interacción del jugador es plenamente con la IU, y el mapa de juego, donde el jugador cuenta con un HUD y el menú de pausa, además de otros menús 3D desplegados al interactuar con elementos del escenario.

Además, cada botón ha sido dotado de una animación de escalado, de modo que cuando son pulsados, el botón se escala 0.8 tanto en el eje de abscisas como el de ordenadas, y tras esto, vuelve a la normalidad (se escala de nuevo a 1). La animación se crea en el diseñador, y asigna a cada botón mediante el grafo, en el que al pulsar el botón, creamos un nodo Play Animation, al que le damos como parámetro de entrada la animación creada. A este nodo le sigue un nodo Delay que retrase cualquier otra funcionalidad después del botón, para que se pueda apreciar la animación.

Los botones tienen también un sonido para cuando son pulsados, y cuando el cursor se sitúa sobre ellos, aunque este aspecto lógico del juego es únicamente asunto del diseñador, ya que desde el grafo no es necesario añadir ninguna funcionalidad.

A continuación se procede a detallar puntos que se han considerado relevantes a la hora de la creación de las distintas interfaces de usuario.

5.2.5.3. Menú principal

El menú principal permite acceder al nivel de juego, además de moverse entre menús y salir del juego.

La funcionalidad a destacar corresponde al acceso al nivel de juego, y a cómo salir del juego.

Para acceder a un nivel de juego, es necesario llamar al nodo Open Level al pulsar el botón de nueva partida o continuar partida, al cual le indicamos el nombre que se le ha dado al nivel en el editor.

Por otro lado, para salir del juego, se logra ejecutando un comando de consola, para lo cual se debe llamar al nodo Execute Console Command, y en el comando a llamar, escribir quit.

5.2.5.4. Menú de opciones

En cuanto al menú de opciones, a través de él se puede configurar tanto la resolución a la que visualizar el juego (por defecto en 1280x720), y el volumen general del mismo.

5.2.5.4.1. Modificar resolución de pantalla.

Para la resolución, se ha establecido 6 configuraciones de pantalla distintas, 3 para el formato 4:3, y otras 3 para el formato 16:9.

Para modificar la resolución de pantalla, al igual que para salir del juego, se realiza mediante un comando de consola. En este caso se utiliza r.setRes seguido de la resolución deseada en AnchoxAlto (se incluye la x), por ejemplo, 800x600, y opcionalmente se añade una w al final para indicar si queremos la pantalla en modo ventana. Por temas de estética final, se ha optado por ejecutar el juego únicamente en pantalla completa.

Es preciso añadir que para la versión 4.6.1 de Unreal Engine 4, utilizada para este proyecto por la compatibilidad directa con el DK1 de Oculus Rift (se pierde en futuras versiones), existe un bug a la hora de realizar el cambio de resolución, que asigna a la vez todas las resoluciones de pantalla conforme se vayan eligiendo. Tras varias comprobaciones personales, se ha llegado a un truco para repararlo, que es estableciendo primero la resolución en modo ventana, y seguido de esto, la resolución en pantalla completa.

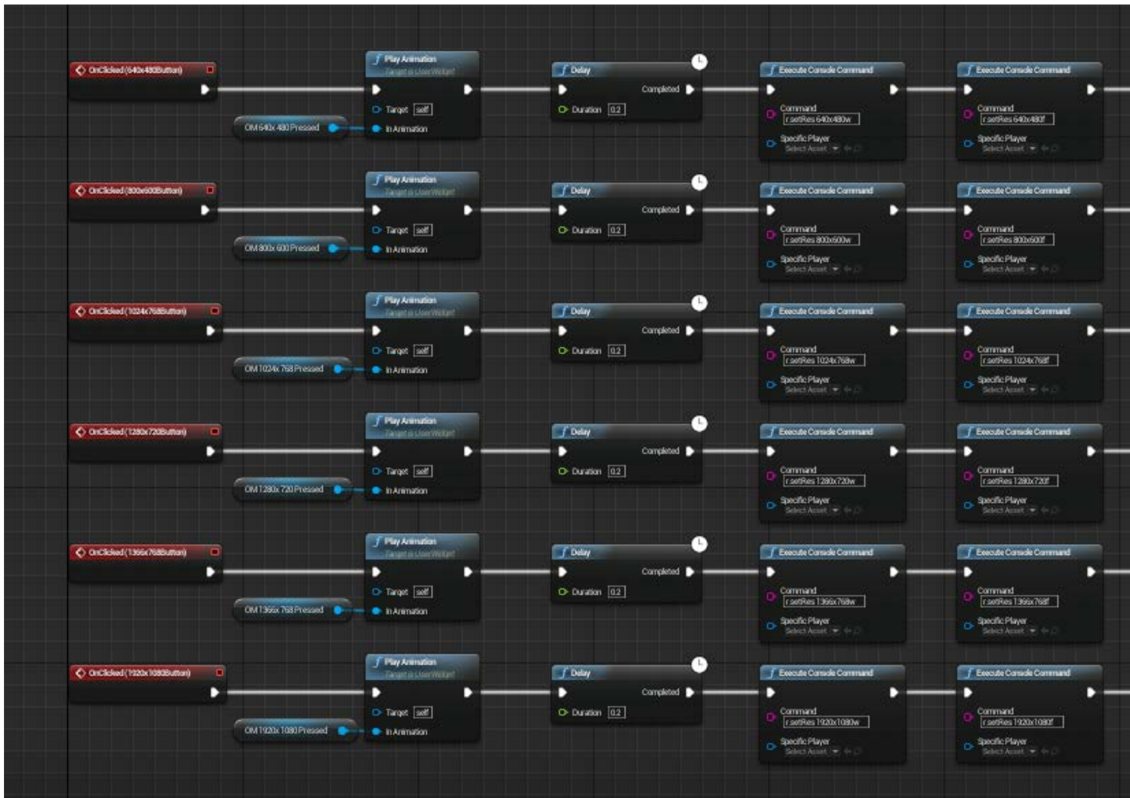


Figura 153. Captura de la lógica de cambiar la resolución de pantalla.

Fuente: Elaboración propia.

Por temas de usabilidad, una vez que está seleccionada una resolución, el botón para elegirla aparece desactivado. La desactivación de botones se realiza mediante el nodo Set Is Enabled, al que se le pasa como parámetro de entrada el botón a desactivar.

5.2.5.4.2. Modificar el volumen general de juego.

La modificación del volumen general del juego se realiza fácilmente mediante una jerarquización de audio. De modo que se ha creado una clase blueprints de audio genérica, llamada BP_Master_Sound, además de otras tres clases más, que son UI, SFX, y Music. Además de esto, se utiliza el plugin de Victory, que incorpora funciones extra al motor, y que descargamos desde la misma web de Unreal de forma gratuita.

A través de la clase BP_Master_Sound, se crean tres variables hijas, que son las tres nombradas en el párrafo anterior, y, en el grafo de esta clase padre, se enlazan nodos para indicar que son clases hijas.

Cuando importamos audio, lo que hacemos es indicar la clase sound a la que pertenece, estableciéndose por defecto BP_Master_Sound.

De este modo, si modificamos el volumen de la clase BP_Master_Sound, se modifica el volumen de las clases hijas. Además, esto permite que en futuros cambios sea posible añadir opciones más personalizadas de modificación de audio, como permitir que sea posible modificar independientemente el volumen de cada uno de los tipos, y también, esta estructura jerarquizada permite crear nuevas subclases hijas. Para este proyecto inicial, a pesar de estar jerarquizado, es

indiferente el tipo de clase a la que pertenece, pues el cambio de volumen se aplica a todas, pero en posibles ampliaciones, este detalle facilita cualquier adición de funcionalidad de audio.

Por último, para acabar con las opciones, a nivel lógico, utilizamos un slider con un valor de 0 a 1. En su evento OnChange, llamamos para actualizar el valor general de la clase de audio padre con “Victory Sound Volume Change”, y por otro lado, en cuanto al valor por defecto del slider, creamos una función bind para esta propiedad, que nos permitirá tener el valor inicializado al general, y a su vez, tenerlo controlado a nivel lógico.

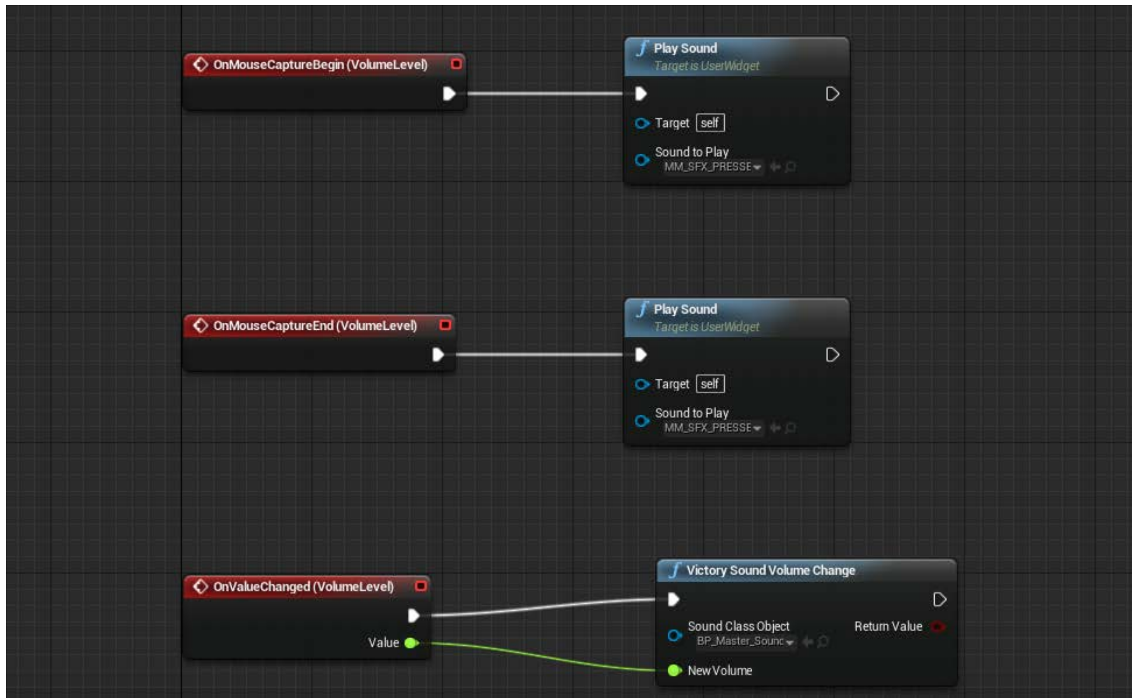


Figura 154. Captura de la lógica de modificar el volumen general de juego, y un ejemplo de reproducción de audio.

Fuente: Elaboración propia.

5.2.5.5. Menú de Pausa

El detalle funcional a tener en cuenta en el menú de pausa es, propiamente dicho, la posibilidad de pausar la partida.

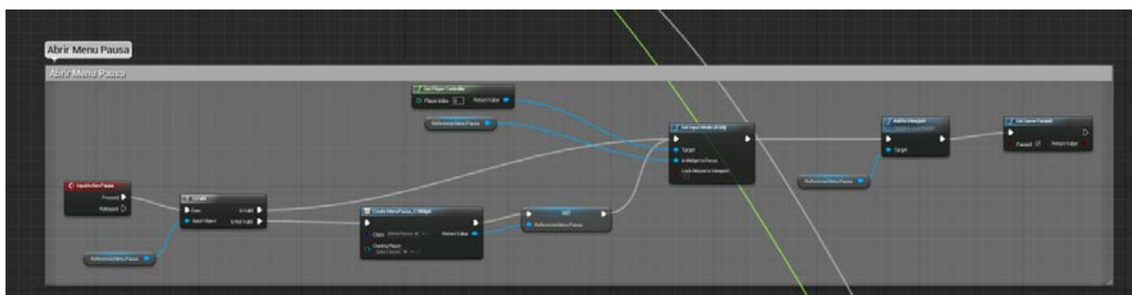


Figura 155. Captura de la lógica de la pausa.

Para pausar la partida en Unreal Engine 4, se cuenta con un nodo especial para ello, este nodo es Set Game Paused, que tiene una variable lógica que colocamos a true o false, dependiendo de si

queremos pausar o reanudar la partida. El motor de Unreal gestiona automáticamente la pausa del juego.

5.2.5.6. HUD

La funcionalidad a destacar con el HUD es que se comunica en todo momento con el blueprint de nuestro personaje, de modo que si este pierde vida, por ejemplo, el HUD debe reflejar esa pérdida en la barra de salud. O si el jugador recoge un objeto, este objeto debe aparecer en el inventario (cuando el jugador lo abra).

Para lograr esta comunicación directa entre ambos blueprints, se ha optado por crear el HUD desde el blueprint de nuestro personaje Arbennig, de modo que así se guarda una referencia al HUD en todo momento. Además, desde el HUD, se obtiene una referencia al personaje utilizando el nodo Get Player Character (con parámetro de entrada player index a 0, puesto que solo tenemos uno), y luego de esto, hacer un cast a la clase específica de nuestro personaje (Arbennig_Character), con el nodo Cast to Arbennig_Character. Finalmente, asignamos este cast a una variable que podamos cómodamente usar en cualquier momento, algo bastante rápido con promote to variable partiendo de la salida del objeto resultado del cast.

Una vez establecida la vía de comunicación entre blueprints, aquellos valores que se quieren modificar, son “bindeados”, para que nuestras variables estén siempre pendientes de cambios, y así nuestro HUD pueda reflejar cualquier cambio ocurrido al jugador, tal como perder vida o energía, o recoger una llave. Estas funciones de bind se asignan desde el Designer del Widget Blueprint, y se les da un comportamiento lógico desde el grafo.

Para las llaves, por ejemplo, se crean 8 variables de tipo ESlate Visibility (el tipo de estructura de datos que utiliza UE4 para realizar cambios de visibilidad en elementos) en el HUD, y 8 variables de tipo bool en el personaje Arbennig. Lo que hacemos en la función bind es comprobar con un branch si alguna llave pasa a estar en true, y si es así, su visibilidad pasará de hidden a visible. De este modo, si el jugador consigue alguna llave, el HUD escuchará ese cambio automáticamente, y procederá haciendo visible la llave obtenida.

Con energía y salud, que contamos con barras de progreso (progress bar), en lugar de hacer bind en la visibilidad, se hace bind en el valor del porcentaje de rellenado de la barra. Teniendo un valor normalizado de 0 a 1 en el jugador, simplemente se lee ese valor y se muestra el porcentaje en la barra.

5.2.5.6.1. Inventario

El funcionamiento extra del inventario es, en este caso, la IU que más funcionalidad extra contiene.

Su funcionamiento lógico nos permite que, estando en el juego, si abrimos el inventario, se muestre en la pantalla, y si lo cerramos desaparezca.

El inventario, además, se compone a su vez de dos partes, una parte de muestrario, donde se aprecian 10 casillas inicialmente vacías, en las que irán apareciendo objetos conforme se recojan, y, por otro lado, una parte de acción, que se hará visible si seleccionamos algún objeto recogido que aparezca en el inventario. Este submenú de acción permite realizar tres acciones, usar/entregar el objeto, soltar el objeto (lo deja caer al escenario), y cancelar acción (si no se desea hacer nada

con el objeto). Tanto usar/entregar (si es posible realizar esta acción con dicho objeto) como soltar, implican que el objeto desaparezca del inventario.

El inventario consiste en un array de elementos, de modo que, a diferencia del caso de las llaves, que ya tienen su posición y solo cambia que estén visibles o no, en este caso, cuando un objeto se recoge, se añade al array, cuando un objeto se usa o se tira, se quita del array, y los objetos que haya en el array, se colocan de forma óptima, no quedando así huecos intermedios, por ejemplo, una vez soltado un objeto.

Además, el inventario no debe interactuar solo con el jugador, también es preciso que los elementos que se puedan recoger, llamen al inventario para que este se actualice si se recogen.

Como extra, cuando nos acercamos a un objeto que es posible recoger, aparecerá sobre él un texto, indicando su nombre.

Dicho esto, para la creación del inventario, se ha utilizado, a parte de un hueco en el Widget Blueprint del HUD para visualizar el inventario, dos Widget Blueprints más:

1. Un Widget Blueprint para cada ranura o slot del inventario.

   En este Blueprint, que funciona como un botón con una imagen, lo que hacemos es principalmente dos cosas, hacer un bind de la imagen que tendrá la ranura o slot del Widget Blueprint, de modo que cuando el jugador recoja un objeto, esta imagen pase a ser la del objeto, y, por otro lado, que cuando el jugador haga clic sobre esta ranura, llame a un Event Dispatcher que envía como parámetro el slot sobre el que se ha hecho clic y activará un evento en el HUD indicándole que qué ranura ha sido seleccionada para tratar.

   Los botones, inicialmente se colocan como desactivados, hasta que se le asigna una imagen distinta de null (con el nodo Is Valid al que le pasamos la variable de imagen de nuestro Widget), lo que se traduce en que hasta que no recogemos un objeto, y se coloca en el slot, no vamos a poder interactuar con el mismo. Al tener un funcionamiento dinámico y deber estar a la escucha de cambios, hacemos bind al comportamiento del botón para su habilitación (Is Enabled?), que se pondrá a true cuando tenga una imagen no nula.

   A niveles de diseño, en el Designer, eliminamos el canvas que nos aparece en él, puesto que no lo necesitamos, y añadimos un botón, y, dentro de este, una imagen. A niveles lógicos, para hacer bind de la imagen que tendrá la ranura del inventario, como no podemos hacer bind de la propiedad de imagen, pero sí del brush (pincel) que la pinta, por código en nuestra función de bind, creamos una variable de tipo Texture2D (el tipo de variable en el que se almacenamos las imágenes representativas para los ítems del inventario), y esta variable la convertimos a Brush llamando al nodo Make Brush from Texture.

2. Un Widget Blueprint para mostrar el texto que aparece en los objetos a recoger.

   Este blueprint, al igual que el anterior, no requiere de un canvas, pues solo vamos a necesitar un texto. Aunque, para poder situarlo correctamente, creamos una horizontal box como contenedor, y dentro de esta, añadimos el texto. Este texto, que queremos que aparezca centrado, cuando creamos el widget, en su Event Construct, utilizamos el nodo

Set Alignment in Viewport, y lo colocamos en el centro, es decir, en 0.5 en x e y.

Aquí, la lógica que tenemos en el grafo nos permite mostrar o no el texto del Widget, además de posicionarlo en la Viewport (pantalla de juego), que lo haremos obteniendo la posición del actor sobre el que queremos mostrar el texto, le añadimos a esa posición, 100 unidades en Y para que no aparezca pegado al objeto, sino un poco más desplazado, y convertimos esa posición en coordenadas en pantalla en función del player controller (obtenemos el player controller con Get Player Controller con index 0), que para hacerlo, con el nodo de player controller, sacamos el nodo Convert World Location to Screen Location, al que le pasamos como input la posición sobre la que mostrar el texto y nos devuelve la posición en pantalla. Para determinar la visibilidad del texto, el nodo para convertir a coordenadas de pantalla, además de la posición en pantalla, nos devuelve una booleana que indica si se pudo o no realizar la conversión, esto es, que si por ejemplo el jugador está mirando hacia arriba, o está de espaldas, no se van a asignar unas coordenadas 2D, y en función de si esto pasa o no, creamos un select de booleanas, que en función de su valor, nos permita determinar la visibilidad del Widget (ESlate Visibility).

Y tres Blueprints de tipo distinto al Widget Blueprint:

1. Un Blueprint Interface, que nos permite comunicar el HUD con los elementos que podemos recoger. En este blueprint especial se crean dos funciones, la función use (usar), y la función drop (soltar). Para la función drop, en su grafo añadimos una variable actor, que corresponde al actor que vamos a soltar.

2. Un Blueprint de tipo Structure, que es básicamente una colección de diferentes variables, y lo vamos a utilizar como la estructura de cada slot del inventario. En este blueprint añadimos 4 variables, la primera, de tipo actor, que es el ítem que recogeremos, la segunda, de tipo Texture2D, que es la imagen que representará el ítem en el inventario, y las otras dos variables, de tipo Text, una para el texto que aparecerá flotante sobre el ítem cuando podamos recogerlo, y otra para el texto que aparecerá en el campo de usar, que bien puede ser entregar el ítem, si es de alguna misión de quest, o usarse, si es un elemento para curación. A este blueprint lo llamamos Inventory Structure.

3. Uno o más Blueprints de tipo Actor, que serán los distintos elementos a recoger del escenario. Este actor debe incluir una malla y un trigger de colisión para detectar al jugador, y como variables, una booleana para saber si el jugador está dentro del trigger (por tanto, se desplegará el mensaje de información del elemento), una variable de tipo Inventory Structure (el blueprint de tipo Structure creado anteriormente), y se rellenan en la pestaña de propiedades de esta variable, todos los componentes de la estructura, a excepción del actor, que lo indicamos por código, y, por último, a este blueprint, añadimos variables(que debemos colocar como públicas o editables para que puedan ser modificadas desde el editor) para aquello que hará el ítem al ser usado, en caso de necesitar algo, por ejemplo, si aumenta la salud del personaje, añadimos una variable float para ello.

Explicado lo anterior, dentro del HUD, añadimos en el Designer dos Vertical Box, una que contendrá el Inventario en sí, y otra que contendrá el submenú de acción que se mostrará al seleccionar algún ítem del inventario. En el Vertical Box del inventario, añadimos un Text Block (para indicarle al usuario que eso que ve es el inventario) y un Uniform Grid Panel, y lo que

hacemos es rellenarlo con 10 Inventory Slot (en la pestaña de User Created nos aparece, puesto que lo hemos creado nosotros). Mientras que en el Vertical Box del submenú de acción, añadimos un Text Block para indicar al jugador que puede realizar una de las acciones que se le muestran en este, y 3 botones, el botón de usar, el de soltar, y el de cancelar (no hacer nada).

Una vez creado todo lo anterior, lo que hacemos es:

1. En la clase del HUD, donde heredamos la interfaz que hemos creado, creamos un evento RefreshInventory, donde creamos un array con 10 variables de tipo slot de inventario. El array lo recorremos con un bucle foreach, comprobando si el inventario está lleno o no (en caso de estar lleno no hacemos nada más), que, en caso de no estarlo, vamos asignando Pickup images y luego de esto, asignamos con el nodo assign a cada slot del inventario un custom event para se haya hecho clic sobre algún elemento del inventario, y que lo que hará esto es inhabilitar el inventario para que aparezca el menú de acción.

   Además, en esta clase damos funcionalidad a los botones del menú de acción, haciendo que:
   a. Cuando se pulse el botón de cancelar acción, se reactive el inventario y se oculte el menú de acción.
   b. Cuando se pulse el botón de soltar objeto, enviemos un mensaje a través de la interfaz al blueprint que se desea soltar, y actualizar el inventario eliminando el objeto después de mandar ese mensaje del inventario y también llamamos a un evento personalizado de acción completada para que reactive el inventario.
   c. Cuando se pulsa el botón de usar objeto, se envía un mensaje a través de la blueprint interface al blueprint concreto que queremos usar, y luego de esto, se actualiza el inventario y se llama al evento de acción completada.
2. En la clase que usemos para crear un elemento colectable lo que hacemos es heredar, al igual que en el HUD, se debe heredar interfaz blueprint de acción que se ha creado, crear el widget que se muestra sobre el objeto con el que podemos interactuar, y asignarle el texto que debe mostrar, luego de esto, asignarle un custom Event de PickupItem, que lo que hará será comprobar si el jugador está dentro del rango para recoger el objeto, y si lo está, recogerlo, añadirlo al inventario, y actualizarlo.
   También, en esta clase se implementa la acción de usar, que dependiendo del objeto, puede tener una funcionalidad u otra.

3. Finalmente, en la clase del Character, que heredará también la interfaz de Action, lo que hacemos es controlar si el jugador pulsa la tecla de inventario para mostrarlo, y habilitar consigo los controles de ratón también en el juego.

   Por otro lado, si el jugador interactúa con algún ítem que es posible recoger, se llama al evento pickupItem que se encuentra en las clases de los objetos colectables recogidos.

   Y ya para acabar, con nuestro character debemos implementar el evento drop action, del que recibimos una variable con el objeto que vamos a soltar, y, luego de esto, reactivamos sus colisiones, lo volvemos a hacer visible, y lo cambiamos de lugar a una drop location.

5.2.5.6.2. Temporizador para pruebas de aros

Para la implementación de este temporizador y su correspondiente lógica de acuerdo a la prueba, lo que se ha hecho es tener en su función tick por un lado, la actualización del contador de tiempo, donde se ha optado por calcular el tiempo de forma se suma el valor de milésimas de cada instante, y cuando llega al valor de 1, se resta un segundo al tiempo restante.

Por otro lado, en la función tick, también comprobamos si el tiempo restante se ha acabado, o si los aros restantes y conseguidos (los conseguidos se van actualizando desde el LevelBlueprint), coinciden, para determinar así si el desafío ha acabado o no. Que llamaremos a un custom event en el grafo que se encargará de marcar que el desafío ha terminado (y si es con éxito o no), para que desde el level blueprint tengamos constancia de ello. Además, se teletransporta al jugador a la isla del desafío, y se hace desaparecer este widget blueprint de los aros y el tiempo.

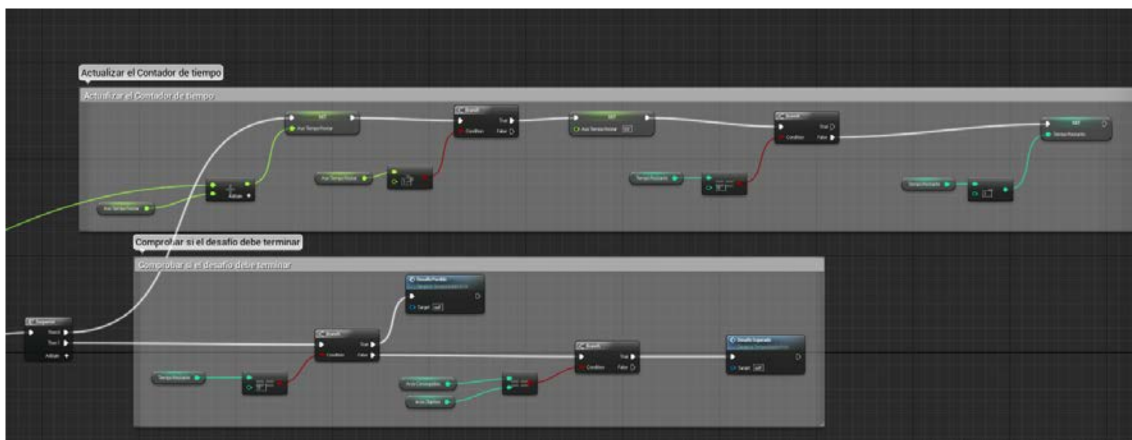


Figura 156. Captura de la lógica de juego del temporizador.

Fuente: Elaboración propia.

5.2.5.6.3. Pantalla de Prólogo, Epílogo, y Créditos.

La implementación de las pantallas de prólogo, epílogo, y créditos, contienen la misma lógica de juego, sin variaciones más que puntuales para decidir hacia dónde apuntan estas pantallas una vez ha terminado su funcionalidad.

Estas pantallas tienen en el diseñador una imagen de fondo, un texto animado que aparece debajo de la pantalla, y que se desplaza hacia arriba por completo, y un botón de saltar la "cinemática". Cada pantalla se encuentra en un nivel de juego distinto, y carga otros niveles cuando acaba de subir el texto hacia arriba o se pulsa el botón.

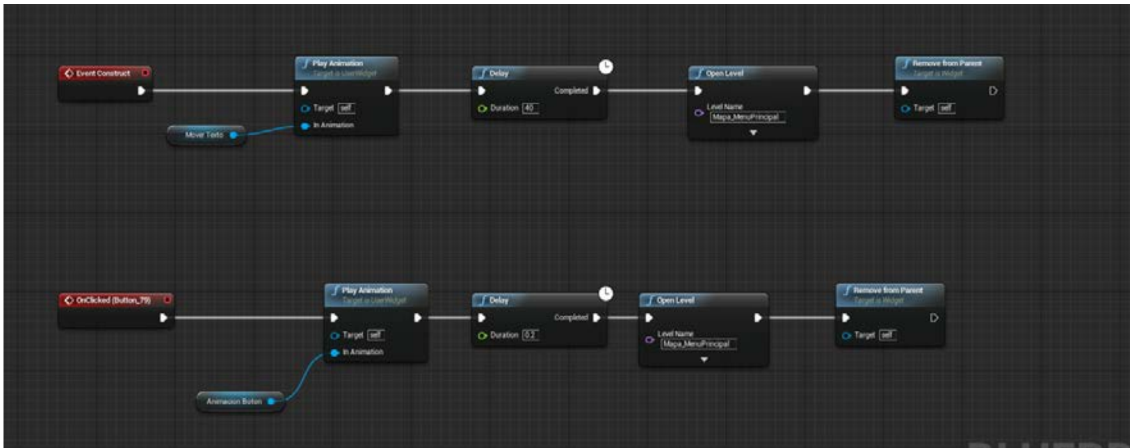


Figura 157. Captura de la lógica de la clase de créditos (igual para prólogo y epílogo).

Fuente: Elaboración propia.

La lógica que sigue para que cuando el texto suba del todo, se cambie de nivel es comprobar la duración en su timeline de animación, la duración que esta sea, la colocamos en un delay dentro del event construct, después de poner a reproducir la animación del texto, y, cuando acaba este delay, abre otro nivel.

Por otro lado, la lógica con el botón es que cuando lo pulse, se abra el nivel al que apunta.

5.2.5.6.4. Pantallas de Desafío superado y no superado, partida guardada, y no puedes usar el objeto.

Se trata de una serie de Widget Blueprints con un mensaje en su diseñador. Cuando se crean, tienen en su lógica interna un delay en su event construct, que cuando pasan un par de segundos, hace que se autodestruyan.

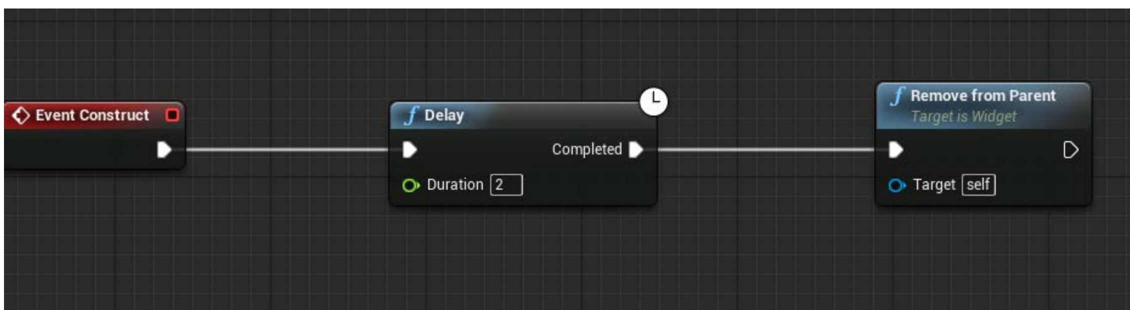


Figura 158. Captura de un Widget Blueprint con la funcionalidad indicada.

Fuente: Elaboración propia.

5.2.6. Implementación de las mecánicas de juego.

En el caso de este proyecto, todas las mecánicas de juego funcionan de forma parecida. Hemos de tener en cuenta que cada mecánica distinta, será un blueprint distinto, que podremos arrastrar al escenario (o hacer spawn de él de forma dinámica).

De cualquier modo, para cada una de las mecánicas, lo necesario esencialmente en cualquiera de ellas es aquello que desencadene la funcionalidad para la que está programada, y en este proyecto, todas las mecánicas activan su funcionalidad mediante triggers con los que el jugador colisiona. Como extra, es posible acompañar estos triggers con una malla que permita al jugador reconocer que se encuentra ante un elemento que no es un simple adorno.

Dicho esto, el funcionamiento más detallado y específico de cada mecánica incluida en el proyecto se presenta a continuación:

5.2.6.1. Teletransportador

Los teletransportadores los encontramos en el bosque de la isla que se encuentra más al sur en el mapa de juego, y en el laberinto al norte del mapa.

La lógica de esta mecánica consiste en mover el actor del personaje a otro punto en el mapa.

Se ha optado por implementar esta funcionalidad con la creación de dos blueprints. Un blueprint teletransportador, con el que si el jugador colisiona, se teletransportará, y un blueprint destino de teletransporte, en el cual aparecerá el jugador si colisiona con el blueprint anterior. A través del blueprint teletransportador, indicamos a qué actor blueprint en el mapa se debe teletransportar el jugador.

Descrita su funcionalidad, mediante blueprints, una vez creados ambos como blueprint de tipo actor, lo que debemos hacer, por un lado, en el blueprint teletransportador:

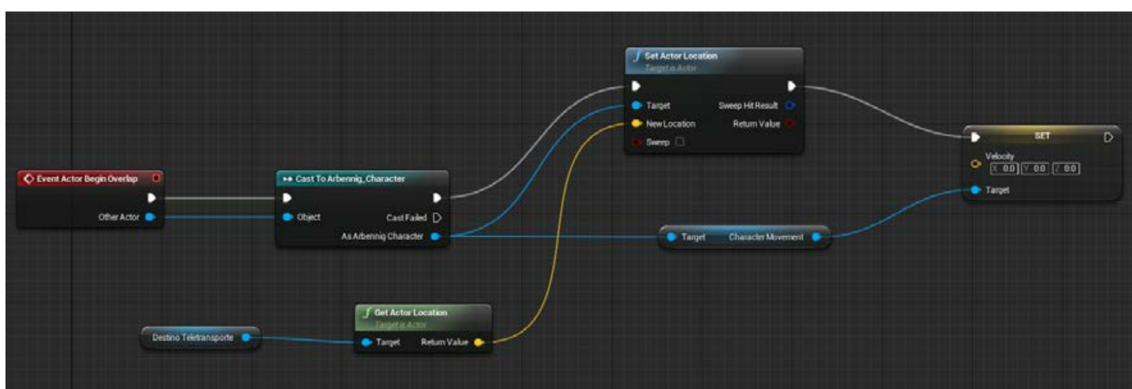


Figura 159. Captura del blueprint de la lógica de la clase del teletransporte.

Fuente: Elaboración propia.

1. Añadir un evento cuando el jugador interactúa con el trigger del teletransportador.
2. De este evento, obtenemos el actor que colisiona, y hacemos cast de él a Arbennig_Character, es decir, a nuestro personaje.
3. De este cast, del que obtenemos a nuestro personaje, hacemos clic en el pin de la variable y llamamos al nodo Set Actor Location, para establecer la nueva posición.
4. Para obtener la posición destino, creamos una variable del blueprint de la clase destino de teletransporte (el otro blueprint) a través de la pestaña de detalles de la variable, y la hacemos editable marcando el checkbox de editable (cuando hacemos una variable editable, es posible acceder a ella desde el editor de niveles) para poder mediante el editor inicializarla indicando a qué destino corresponde el teletransporte. Esta variable la arrastramos al grafo, indicando que la queremos obtener como Get, y de ella, arrastramos su pin para obtener un nodo Get Actor Location (que nos dará la posición de Destino), y por último, arrastramos el valor que proporciona este nodo, al input New Location del nodo Set Actor Location al que hemos llamado en el paso 3.
5. Arrastrando nuevamente desde la variable de nuestro personaje, obtenemos su Character Movement, y de este, llamamos a Set Velocity, para que si el jugador estaba en movimiento, su velocidad pase a ser 0.

En el blueprint de destino de teletransporte, por cuestiones de experiencia de juego:

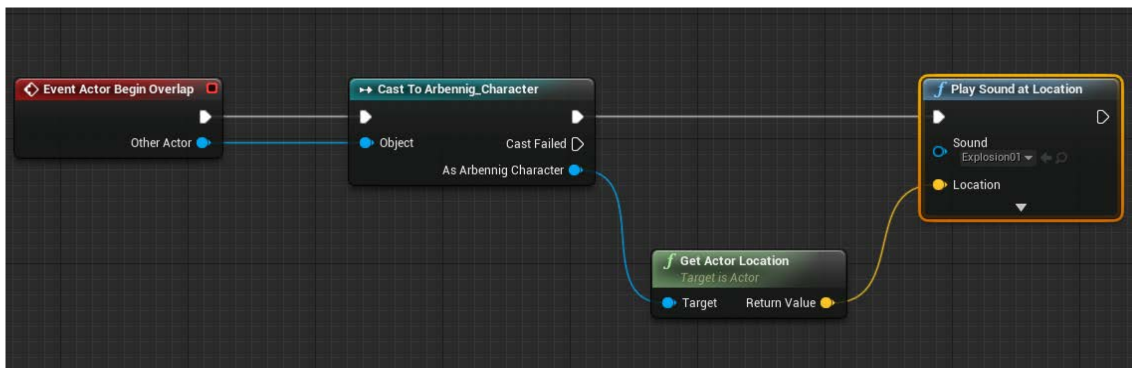


Figura 160. Captura del blueprint de Destino de Teletransporte.

Fuente: Elaboración propia.

1. Añadir un evento cuando el jugador interactúa con el trigger del teletransportador (cuando el jugador se haya teletransportado a esta posición, se activará).
2. De este evento, obtenemos el actor que colisiona, y hacemos cast de él a Arbennig_Character, es decir, a nuestro personaje.
3. Llamamos al nodo Play Sound at Location, y elegimos un sonido de nuestro content browser y colocamos la posición donde el jugador se encuentra usando un nodo de get actor location sacado del cast.

5.2.6.2. Aros

Los aros aparecen cuando el jugador accede a uno de los desafíos de vuelo en el juego tras hablar con el tótem del templo 4 o 7 y aceptar la misión.

El jugador ve aparecer un temporizador y un texto informativo de los aros conseguidos en la pantalla, se activan y vuelven visibles todos los aros, y se debe pasar por cada uno de los estos antes de que el tiempo se acabe. Cuando el jugador pasa por un aro, este realiza una ligera animación (gira más rápido) y, tras un par de segundos, desaparece. Además, si el tiempo llega a 0, los aros desaparecen, y deben restaurarse para un siguiente intento si el jugador lo desea reintentar. Además, cuando acaba el desafío de vuelo, el jugador es teletransportado al lugar donde el tótem correspondiente se encuentra.

La lógica de los aros hace uso de 4 blueprints principalmente:

1. **Blueprint de tipo Widget Blueprint** que vamos a crear para que se solape en la interfaz dado el momento, y así no tener el HUD muy recargado de lógica. Aquí nos encargamos de mantener actualizado el contador de tiempo restante y los aros recogidos respecto a los restantes. Además de eso, a través de este blueprint comprobaremos si el desafío termina o no, si ha sido con éxito o no, y, dependiendo de esto, indicarlo para que en el blueprint del nivel se proceda a entregar al jugador la runa o no. Por último, es a través de este blueprint donde se ha decidido teletransportar al jugador al lugar donde se encuentra el tótem.
2. **Blueprint del Aro**. Este blueprint, del que se ha establecido dos versiones, una más simple, y otra más compleja con efectos de partículas, se encuentra en el escenario girando a cierta velocidad. Cuando el jugador colisiona con el trigger de su interior, empieza a girar más rápido, y a los dos segundos desaparece. Además, restablece al máximo la energía del jugador, y registra que ha sido atravesado en una variable para que se pueda proceder en el blueprint del nivel.
3. **Blueprint del personaje**. Se accede a través de los distintos blueprints al personaje para modificar sus variables de runas, energía, posición, y estado.
4. **Blueprint de nivel de juego.** Con este blueprint vamos a iniciar los desafíos de aros, bloqueando así la realización de cualquier otro desafío, y reiniciar sus valores en caso de no ser superado. Además, enviaremos información al Widget Blueprint del temporizador, para indicarle el número actualizado de aros atravesados, y, por último, también estaremos comprobando si el desafío debe terminar, para otorgar al jugador la runa, y restablecer el juego de modo que puedan realizarse otros desafíos.

A continuación, se procede a explicar a nivel lógico los pasos más relevantes a la hora de haber realizado cada uno de los blueprints creados (por tanto, no del personaje, que ya ha sido creado con anterioridad, y no se le realiza ningún añadido de código).

Widget Blueprint del temporizador.

En este blueprint tenemos un widget de text para el temporizador, otro para los aros conseguidos respecto a los totales, y un par más para mostrar un mensaje de desafío completado o no. Estos textos incluyen una imagen de fondo para resaltar más.

Lo que hacemos es un bind del contenido del texto del temporizador y de los aros obtenidos. Este bind lo asociamos a unas variables de tipo integer, que castearemos primero a string para poder hacer append (nos permite unir strings), y así añadir "s" de segundos restantes, o "/" y el número total de aros. Por otro lado, hacemos bind de la visibilidad de los textos de desafío superado o no, asignándolos a una variable de tipo ESlate Visibility, de modo que tengamos el control de esto para cuando nos sea necesario mostrarlo.

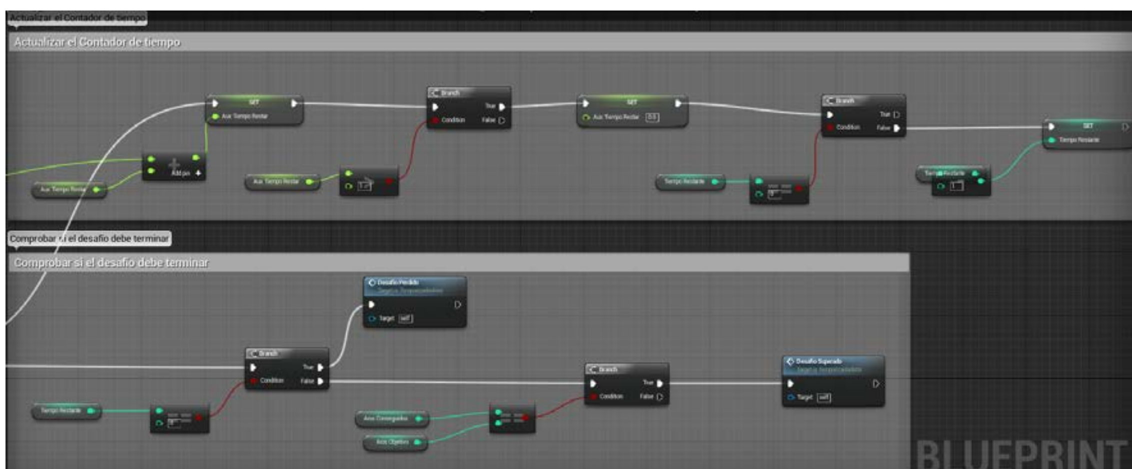


Figura 161. Captura de la lógica del grafo del Widget Blueprint

Fuente: Elaboración propia.

En cuanto al grafo principal de este blueprint, tenemos partiendo de una función tick que se irá llamando a una secuencia de dos pines mientras el desafío no esté terminado. En estos dos pines, el primero se encarga de actualizar el contador de tiempo (en función del delta time, va sumándose a una variable auxiliar, cuando el valor llega a uno, se resta una unidad al contador, y el auxiliar se reinicia a 0), y el segundo se encarga de comprobar si el tiempo restante es 0, para llamar a un Custom Event de desafío perdido, y comprobando en caso de no acabarse el tiempo, si el número de aros conseguidos es igual al de aros objetivo, que, en caso de ser así, llamaría a otro Custom Event, pero de desafío superado.

Los custom event de desafío perdido y superado, lo primero que hacen es cambiar a visible el texto de desafío superado, y su fondo de imagen, luego de esto, marcar a true la variable de desafío finalizado, y a true o false una variable que indica si el desafío ha terminado con éxito. A esto le sigue un delay de 3 segundos, para que el jugador pueda apreciar el texto de que ha superado o no el desafío. Una vez transcurrido el tiempo de delay, se cambia el estado del personaje, sea cual sea (seguramente vuelo), a falling, para que se establezca automático a walking (y parezca que nos han teletransportado y caemos), y, finalmente, llamamos a remove from parent para eliminar el widget blueprint.

Blueprint del Aro.

En este blueprint, que recordemos, tenemos dos (aunque la única diferencia es que uno tiene una mesh, y otro tiene un efecto de partículas alrededor del trigger), lo que hacemos es tener una función Tick, que está constantemente girando el aro mediante una rotación local, que se aplica multiplicando el delta time por un valor al gusto, creando un rotator con esto en un eje, y aplicándola a la mesh con Add Local Rotation. Incluimos antes de aplicar la rotación, una comprobación de si el jugador ha atravesado el trigger, para que si es el caso, el multiplicativo pase a ser otro número mayor.

Cuando el jugador atraviesa el trigger, salta el evento OnComponentBeginOverlap, en el que comprobamos si el trigger ya ha sido atravesado, activamos la variable bool para que el aro gire más rápido, y luego de esto, recargamos la energía del jugador (obteniendo el personaje con la función Get Player Character y haciendo cast a Arbennig Character), marcamos a true la variable que indica que el aro ha sido atravesado, y hacemos un delay de un par de segundos (para que el aro tenga 2 segundos de rotar rápido). Finalizado el delay, marcamos a false la variable que indica que el aro gire rápido, y ocultamos el actor con Set Actor Hidden in Game.

Blueprint del nivel de juego.

Para esta parte de los aros, lo hacemos es uso del evento begin play, y del evento tick en este blueprint de nivel, además de uso del evento de activar el desafío del tótem.

En begin play obtenemos referencias a los aros que hemos situado en el escenario, creamos un array con ellos, y ese array lo volvemos una variable a la que accederemos en la función tick varias veces. Los aros de cada desafío deben ir en distintos arrays.

Por otro lado, en la función tick, realizamos una serie de comprobaciones, que consisten en comprobar si el desafio de aros ha sido activado (lo activamos al aceptar el desafío por parte del tótem), y de si se ha inicializado ya o no los componentes del desafío. Dicho esto, inicializamos

primero los componentes del desafío (ya que no hemos inicializado aún ninguno), donde en cada aro del array cambiamos su visibilidad de oculto a visible, creamos el widget blueprint del temporizador y lo asignamos a una variable para tener referencias a él de forma cómoda, y lo añadimos a la viewport, y entonces accediendo a este widget, inicializamos a 0 los aros conseguidos, le indicamos los aros objetivo haciendo un length del array de aros, le asignamos el tiempo de desafío, y por último, el destino de teletransporte. Al acabar esta parte de inicialización, establecemos a true una variable que indica que hemos inicializado los componentes, y empezamos en la función tick como tal, donde vamos a actualizar los aros conseguidos y a comprobar si el desafío ha terminado y cómo.

Para actualizar los aros, se ha optado por hacer un loop foreach del array de aros, de modo que para cada elemento, comprobamos si su variable de que ha sido atravesado está en true o no, y las sumamos a una variable de aros conseguidos. Cuando el bucle termina, accedemos a la variable de aros conseguidos del widget blueprint, y le hacemos set del resultado, acabando por reiniciar a 0 nuestra variable de aros conseguidos, que al estar en la función tick, irá añadiendo valores constantemente.

En cuanto a la comprobación de si el desafío ha terminado o no, lo que hacemos es acceder al widget blueprint y comprobar si la variable bool que habíamos creado para indicar que el desafío está terminado está en true o false. En caso de ser true, comprobamos si su variable bool de desafío superado con éxito está también en true o false, e independientemente de esto, marcamos a false la variable que indica que nuestro desafío está activo.

Si el jugador ha terminado con éxito el desafío, accedemos a la variable de la runa 4 para colocarla a true, almacenada en el character de nuestro personaje (accedemos a ella con la función get player character, y de esta, haciendo un cast a Arbennig_Character), y por último, colocamos nuestra variable de enumerador de desafío de tótem en none, para que sea posible realizar más desafíos.

En caso de que el jugador no haya terminado con éxito el desafío, se restaura a false la variable de inicializar los componentes, para que sea posible reinicializarlos si se desea reintentar le desafío no superado. Además, se cambia la visibilidad de todos los aros a hidden (si hemos superado el desafío, ya estarían todos en hidden), y también, realizando un bucle foreach del array de aros, establecemos a false la variable que indica que los aros han sido atravesados. Por último, reestablecemos a 0 la variable de aros conseguidos, para que al iniciar el desafío de nuevo, el primer valor de aros atravesados no sea el de la anterior partida, y ya establecemos en none la variable enumerada que indica el desafío en que nos encontramos.

Respecto a la lógica que contiene el evento de activar el desafío del tótem, simplemente establecemos el enumerador de desafío al que vamos a realizar (4 o 7), y ponemos a true la variable de activar desafío de aros.

Para el desafío de aros con el otro tipo, los procedimientos son exactamente los mismos.

5.2.6.3. Diálogos de los NPC y tótems

Los diálogos que aparecen en el juego se muestran cuando el jugador interactúa con algún NPC (aldeanos del pueblo, árbol, y tótems). Consisten en un cuadro de diálogo, donde en la parte superior se encuentra el mensaje del NPC, y debajo de este se muestran las respuestas disponibles del jugador.

Con este sistema que el jugador y el NPC entablen una conversación de múltiples respuestas. En este proyecto, nuestro personaje puede hablar con los NPC de la aldea y el árbol de forma infinita, pero con los tótems, una vez se acepta el desafío y se supera, el tótem deja de poder hablar, porque "su alma ha sido liberada".

Para la implementación de este sistema de diálogos es necesaria la creación de dos widgets (un widget de opción, y otro de pantalla de diálogo), tres structs (árbol de diálogo, respuesta del NPC, y opciones del personaje), y un actor (sistema de diálogo). Además, es necesario hacer un par de llamadas al Blueprint de nuestro personaje.

El cómo encajan entre sí todos estas estructuras y clases es de la siguiente forma:

- El actor de sistema de diálogo contiene el widget de pantalla de diálogo (es solamente algo meramente visual, que servirá de contenedor para las distintas respuestas, tanto del NPC como del personaje), que, a su vez, contendrá dinámicamente al widget de respuestas al NPC (consiste en un botón con un texto sobre él).
- El struct de árbol de diálogo contiene al struct de respuesta del NPC, que, a su vez, contiene al struct de opciones del personaje.
- Cuando interactuemos con un NPC o tótem, desde el blueprint del personaje vamos a tener que habilitar controles sobre la interfaz de usuario, además de hacer spawn del actor que queramos mostrar.

Para habilitar los controles sobre la interfaz de usuario, creamos una función InitMouse, en la que creamos una referencia a nuestro player controller, y desde ella, establecemos el modo SetInputModeUIOnly, además de modificar la variable bool de mostrar o no la cruceta roja del hud, la de mostrar el cursor, y las de habilitar eventos de ratón.

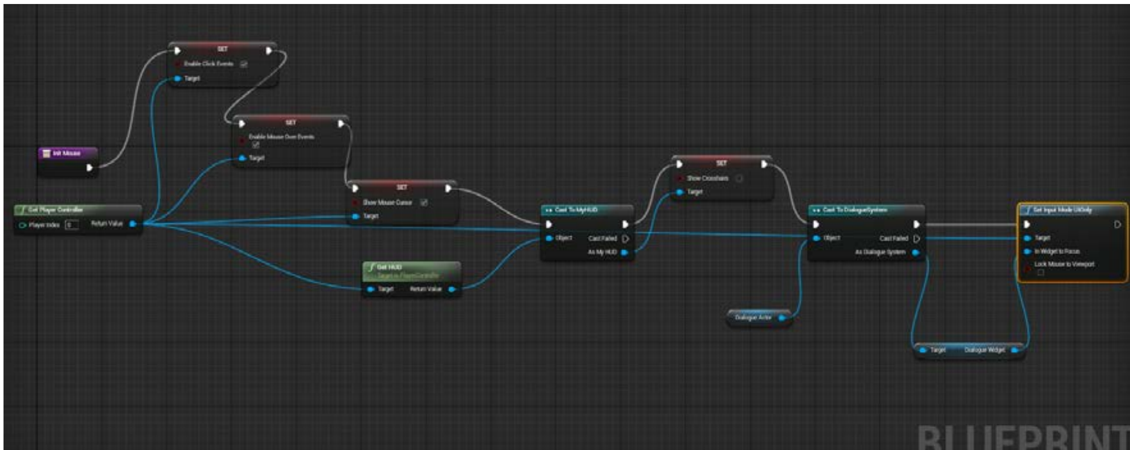


Figura 162. Captura de la función initMouse

Fuente: Elaboración Propia

Por otra parte, para hacer spawn del actor, nos creamos otra función, SpawnDialogue, y en ella simplemente llamamos al nodo SpawnActorFromClass. Como class, elegimos la clase Blueprint DialogueSystem (sistema de diálogos), y por último, nos guardamos la variable de este actor creado. La posición donde haga spawn del actor es indiferente, porque al crearse, el widget aparece en la viewport. Cuando sea que se hable con algún NPC, se podrá acceder así fácilmente al actor de diálogo a través de nuestro character.

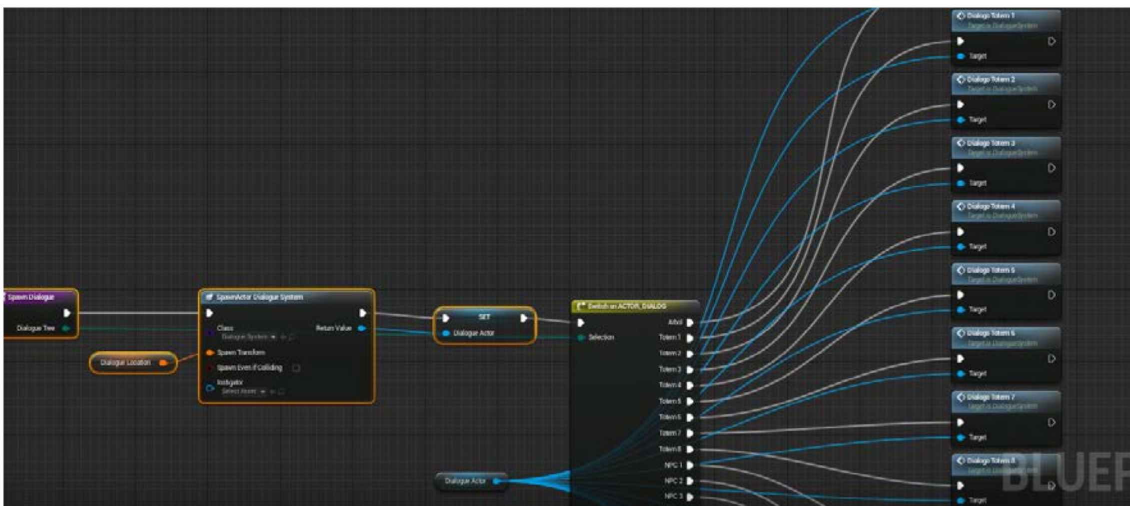


Figura 163. Captura de la función SpawnDialogue

Fuente: Elaboración propia.

En cuanto a los struct, contienen lo siguiente:

- Struct de Opciones de personaje (PC_Choices). Contiene Reply (respuesta), LinkToNode (link al nodo de respuestas al que apunta), y exitDialogue (bool para determinar si con esta opción, acaba la conversación).

- Struct de Respuesta del NPC (NPC_Reply). Contiene NPC_Reply (la respuesta del NPC), y un array de PC_Choices.
- Struct del árbol de diálogo (DialogueTree). Contiene una variable Conversation, que es un array de NPC_Reply.

En lo referente a los Widgets, el Widget de sistema de diálogo no tiene ningún tipo de lógica en su clase, simplemente tenemos aquí un contenedor de tipo border, que dentro incluye un texto (la respuesta del NPC), y otro border, que a su vez tiene dentro una Vertical Box (donde aparecerán las respuestas disponibles para el jugador que se añadirán de forma dinámica), pero es necesario indicar que tanto el texto como la vertical box son variables, y hacerlas públicas para poder editarlas desde fuera. En el Widget de respuestas debemos hacer variable pública el texto del botón, y, a nivel lógico, lo que hacemos es obtener el DialogueActor desde nuestro Character, y llamar a una función que crearemos en este actor, con nombre LoadDialogue, que recibe como parámetros LinkToNode (el nodo al que apunta la opción), y ExitConversation, y que lo que hará esta función es cargar/recargar el sistema de diálogos del juego cada vez que pulsamos una opción.

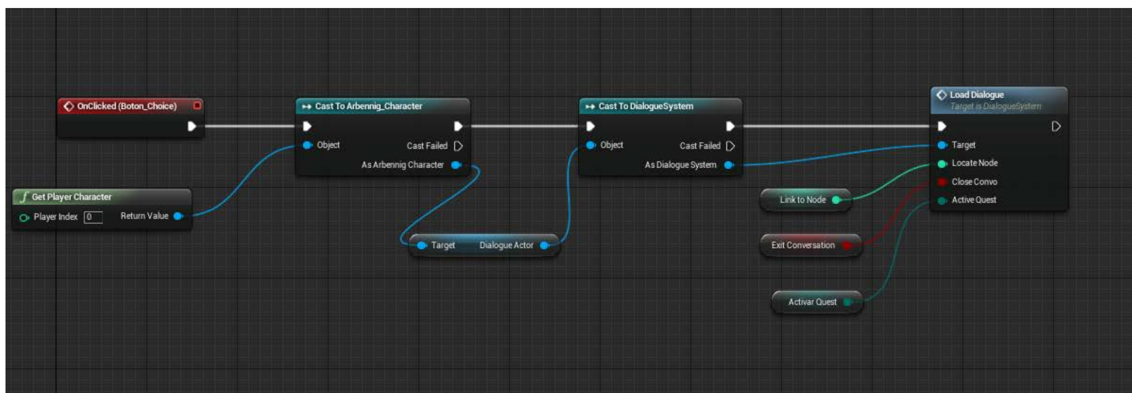


Figura 164. Captura de la lógica del Widget de opciones de diálogo.

Para acabar, la clase DialogueSystem. En esta clase tenemos lógica en el grafo principal, y en dos funciones que creamos, una es LoadDialogue, ya mencionada, y la otra es StartConversation, donde lo único que hacemos es llamar a la función LoadDialogue.

En el grafo principal de la clase nos creamos variables de estructuras de datos del tipo dialogueTree por cada conversación distinta y uno general al que asignar este valor. Aparte de esto, por una parte, tenemos una serie de custom events, que en función de NPC o tótem con el que hablemos, saltará uno u otro, asignando así un valor para un struct de árbol de conversación u otro. Por otra parte, mediante el evento begin play, que se llamará una vez se haga spawn de este actor, creamos un widget de DialogueScreen, nos almacenamos su valor en una variable, y lo añadimos a la viewport con su nodo add to viewport, finalmente, llamamos a la función StartConversation.

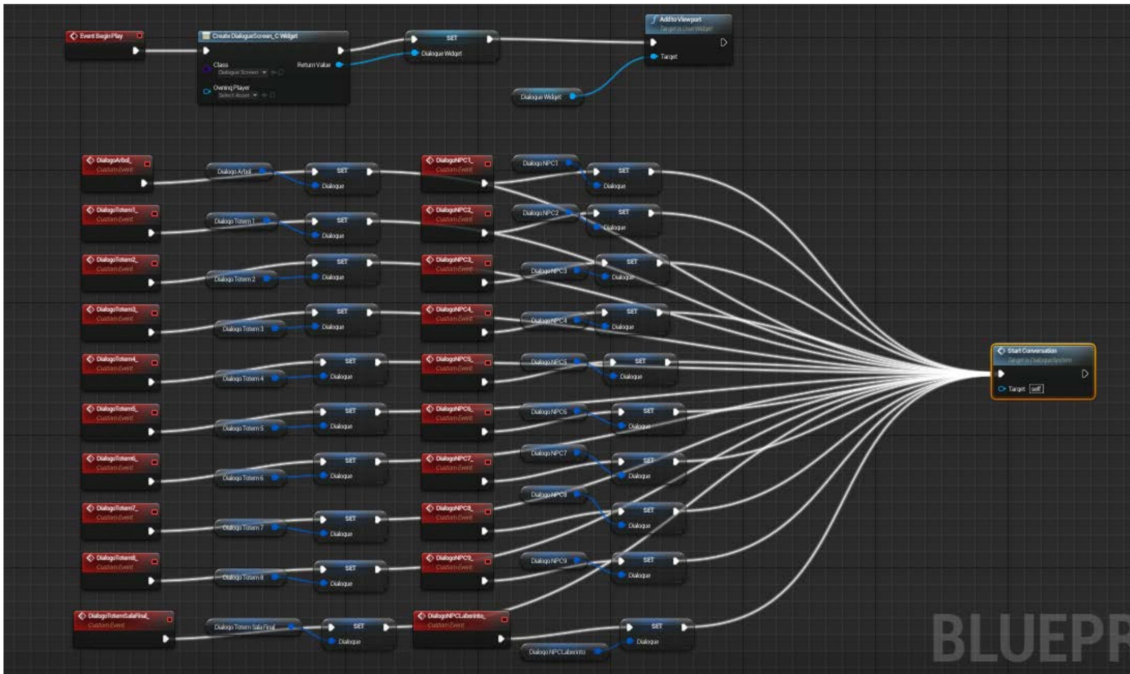


Figura 165. Graph node de la clase de DialogueSystem.

Fuente: Elaboración Propia

Para terminar, en la función loadDialogue, lo que hacemos es llamar a un branch con la condición de exit que parte como parámetro de entrada de la función. Si está en true, significa que la conversación ha terminado, y llamaremos a setVisibility del Widget de DialogueScreen, además de llamar al evento custom del personaje de reactivar los controles.

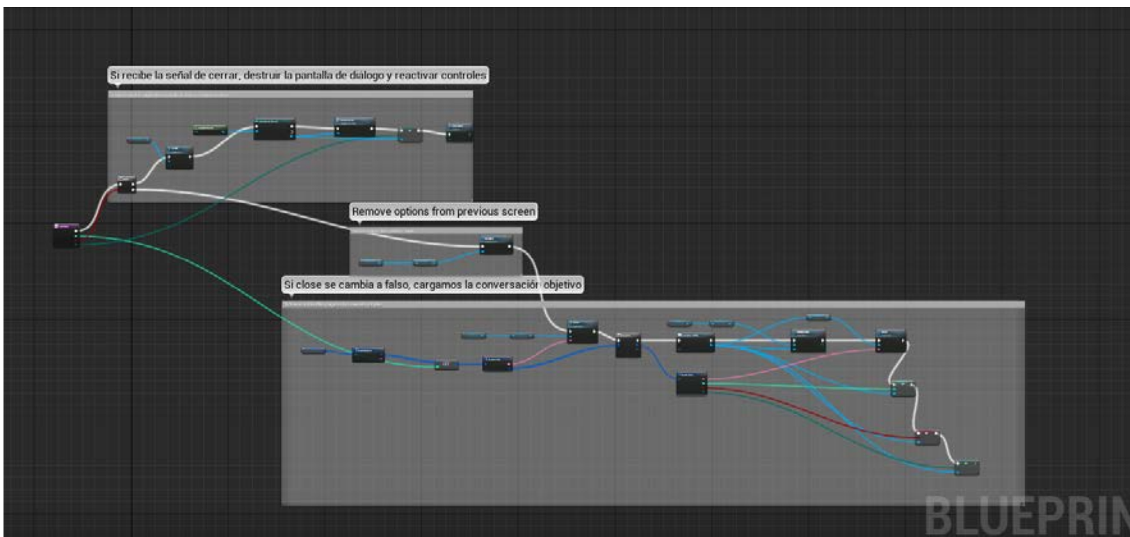


Figura 166. Función LoadDialogue completa.

Fuente: Elaboración propia.

Si el branch está en false, lo que hacemos es cargar todo en el widget de DialogueScreen que tenemos, pero primero, accedemos al contenedor vertical box de las opciones de conversación, y lo limpiamos con clear children. Para cargar la conversación adecuadamente, usamos el otro parámetro de entrada de la función, que corresponde al nodo de conversación que debe leer (con nodo queremos decir, por ejemplo, una respuesta de NPC, y las opciones para esta respuesta), y con esto, usando el struct de conversación que hemos asignado en el grafo principal en función del NPC con el que hablamos, obtenemos el nodo.de la conversación (llamamos a la función get del struct, y su índice es el valor del nodo que queremos). Seguido de esto, hacemos break de este nodo de conversación, que es un struct de NPC_Reply, y con él, asignamos el valor del texto de respuesta del NPC (usamos la función SetText para esto).

Por último, nos queda añadir las opciones de respuesta del jugador a la vertical box que tenemos preparada para ello, y esto lo hacemos llamando a un bucle foreach del array de PC_Choices que tenemos (cuando hemos hecho break del struct NPC_Reply, recordemos que contiene un NPC_Text y un array de PC_Choices), y por cada elemento de este, llamamos al nodo Add Child Vertical Box, y le asignamos de él el Texto correcto, el nodo de conversación al que apunta la respuesta, y el valor de la variable bool de ExitConversation.

5.2.6.4. Puerta del final del juego.

La mecánica de la puerta final del juego contiene su funcionalidad en dos partes, un trigger que mandará el aviso a la clase del nivel de que la puerta está lista para abrirse, y un trigger de la puerta, que contiene mediante una animación con timeline, la abre.

Empezando con la clase del trigger, que contiene una luz que se enciende y se apaga en función de si el jugador está sobre el trigger o no, lo que hacemos es detectar primero, cuando se colisione con el trigger, que es posible que el jugador abra la puerta (porque está dentro del radio de acción del trigger).

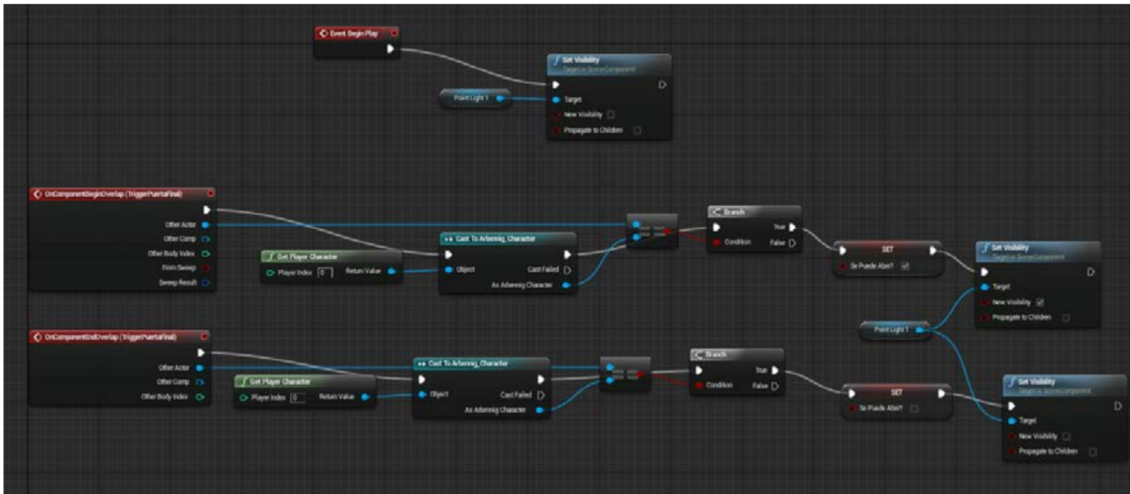


Figura 167. Captura de la inicialización y respuesta a los eventos cuando el personaje atraviesa y deja de atravesar el trigger.

Fuente: Elaboración propia.

Sabiendo si está dentro o no del trigger, dentro del evento tick lo comprobaremos, y también si el jugador está intentando interactuar con él, que, en caso afirmativo, comprobaremos si tiene todas las llaves, para enviar la señal al level blueprint de que la puerta se tiene que abrir.

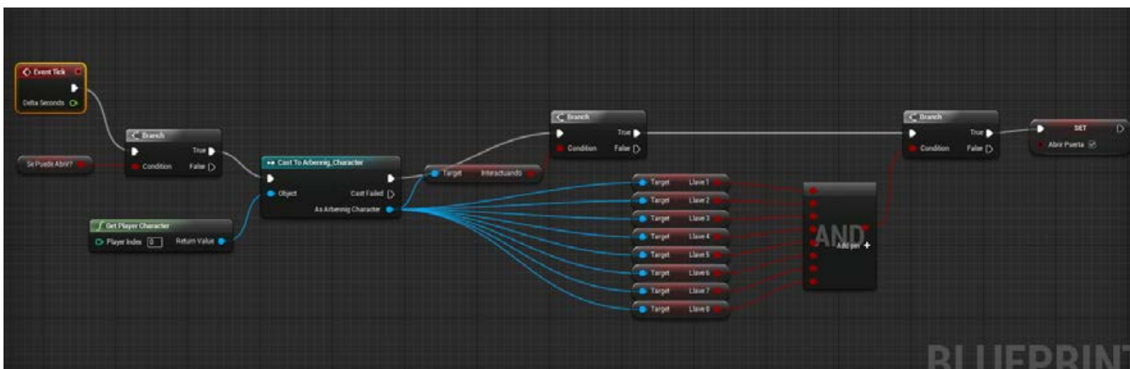


Figura 168. Captura de la lógica del trigger de abrir la puerta en la función tick.

Fuente: Elaboración propia.

Ahora, en el LevelBlueprint, lo que hacemos es tener en la función tick una comprobación de si hemos modificado o no la variable que hace de flag para que se abra la puerta y, en caso afirmativo, mandamos la señal a la puerta de que se abra usando otro flag.

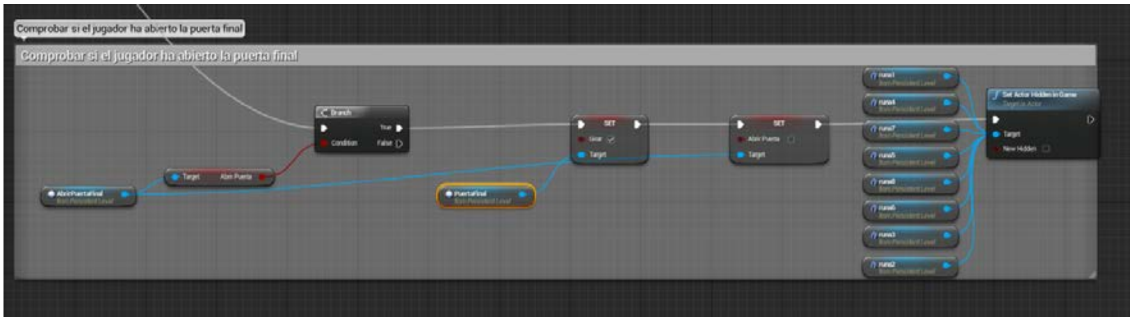


Figura 169. Captura de la lógica de abrir la puerta en el level blueprint.

Fuente: Elaboración propia.

Por último, dentro del blueprint de la puerta, lo primero que hacemos es en su evento de begin play, guardarnos la rotación inicial de esta, que nos hace falta para poder utilizar el timeline. Y finalmente, dentro de las comprobaciones con el evento tick, comprobamos si el flag de que se abra ha sido aactualizado a true, que, en caso afirmativo, usará un timeline, que es un componente de blueprints para realizar animaciones sencillas, y apoyándonos en una interpolación, haremos girar la puerta hasta que se abra.

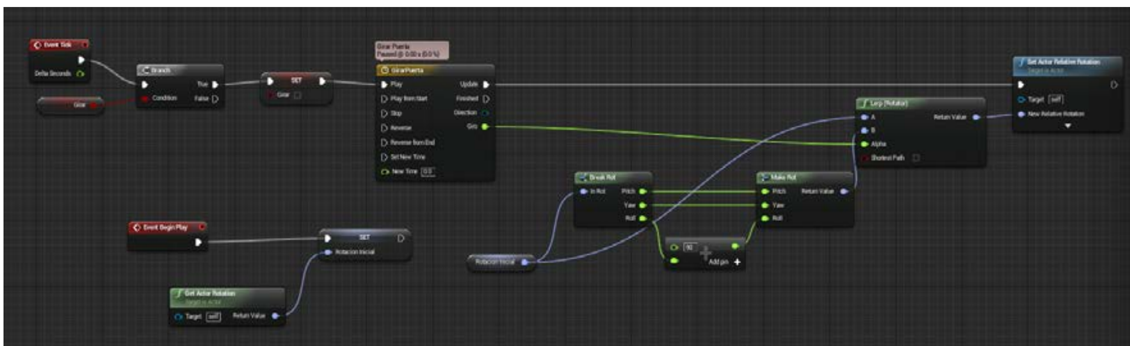


Figura 170. Captura de la lógica de la clase de la puerta final.

Fuente: Elaboración propia.

Como detalle extra acerca de los timeline, su funcionamiento consiste en una curva de animación que puede hacer variar el valor de una variable o un vector de acuerdo a esta. En nuestro caso, como solo nos es necesario que la puerta gire en un eje, con una variable float de parámetro de salida que modifique su valor nos es suficiente.

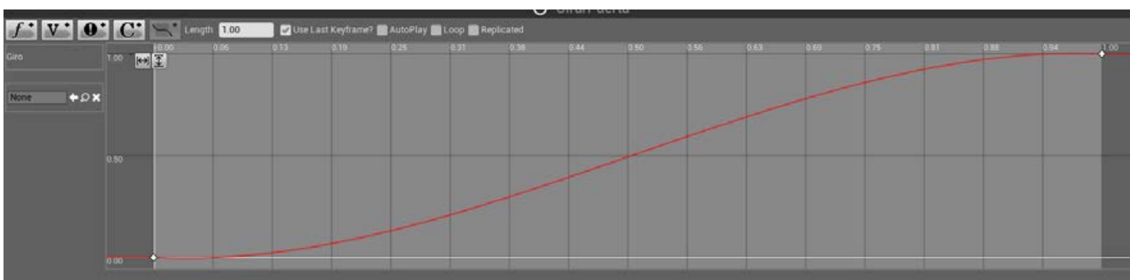


Figura 171. Captura del timeline de girar la puerta.

Fuente: Elaboración propia.

5.2.7. Sistema de guardado/cargado de partida.

El sistema de guardado en Unreal Engine 4 se realiza mediante un blueprint que hereda de la clase Save Game.

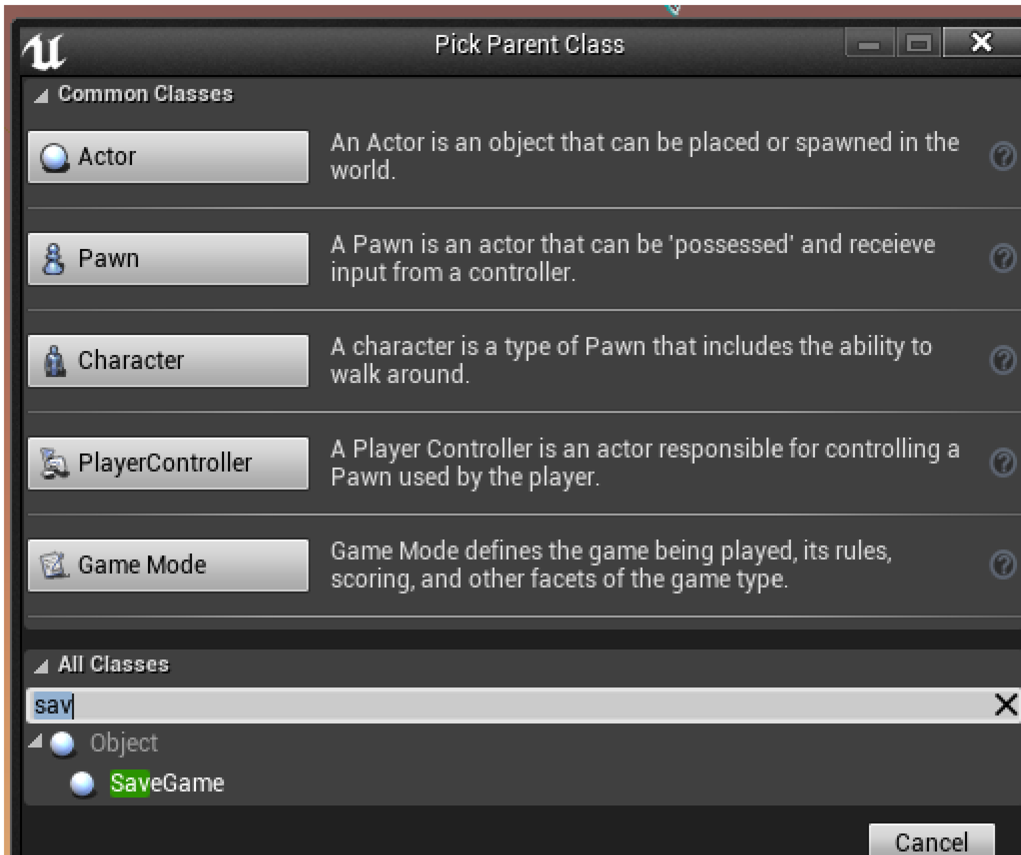


Figura 172. Captura de cómo encontrar la clase de SaveGame.

Fuente: Elaboración propia.

Con esta clase creada, lo que hacemos es almacenar en ella todos los datos que queramos guardar, creándonos variables para ello como si de un blueprint normal se tratara. En nuestro caso, nos interesa guardarnos 8 booleanas, una para cada llave, que estarán a true o false dependiendo de si el jugador las ha conseguido o no, también nos guardaremos un vector para almacenar la posición del jugador en la que guardó la partida, una booleana para determinar si hay o no una partida guardada, y por último, una variable string y otra integer que nos sirven para identificar el archivo de guardado que se crea. Todas estas variables las debemos dejar en visibles para el resto de clases, para que puedan acceder a sus valores.

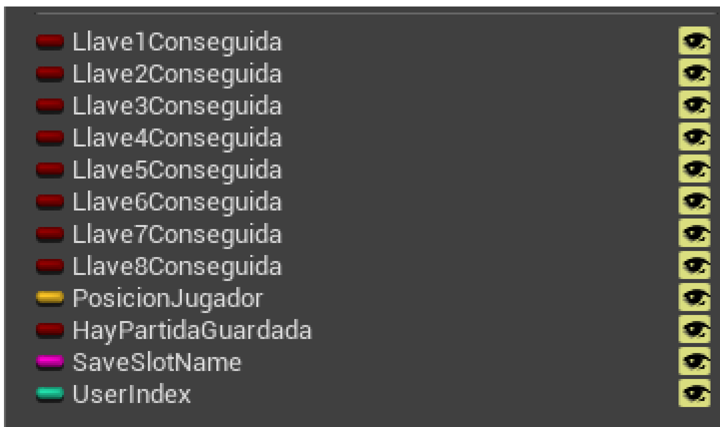


Figura 173. Captura de las variables almacenadas en el blueprint.

Fuente: Elaboración propia.

Con esta clase, ya podemos guardar la partida y cargarla.

5.2.7.1. Guardar partida.

Para guardar la partida, se ha creado un blueprint que consiste en un punto de guardado, el cual contiene una esfera de posición donde aparecerá el jugador. En él, cuando el jugador se acerca, pondrá a true un flag que indicará en su función tick que puede guardar la partida.

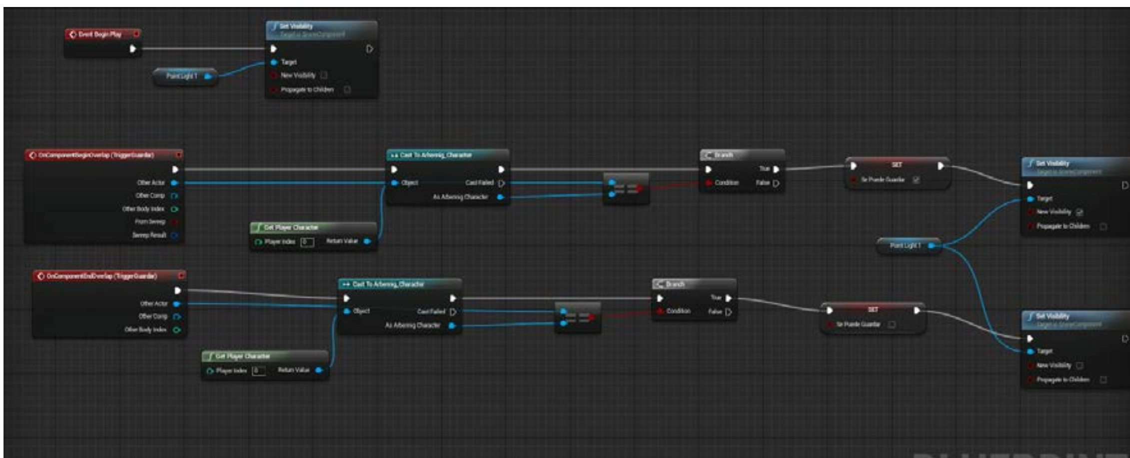


Figura 174. Lógica para detectar que el jugador está dentro del trigger de guardar la partida.

Fuente: Elaboración personal.

Hecho esto, y después de haber comprobado que el jugador está interactuando, la lógica del sistema de guardado consiste en llamar a la función Create Game Save Object a la que le indicaremos la clase blueprint que hemos creado, entonces, de este objeto de tipo game sabe, hacemos un cast a nuestro blueprint y nos guardaremos esa variable resultado del cast para ya acceder a las propiedades que contiene y modificarlas. Indicando que existe una partida guardada, accediendo nuestro character, con el que a su vez, accederemos a sus booleanas de llaves conseguidas, guardando en el vector de posición, el de la esfera de posición de reaparición que

hemos creado en el blueprint, y por último, indicando que vamos a guardar la partida en el índice 0, y su nombre va a ser 0.

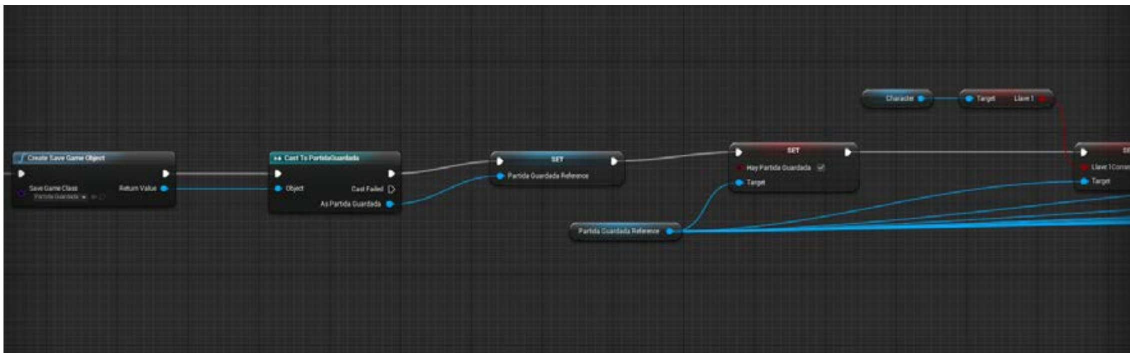


Figura 175. Captura de la creación del objeto save game, de su cast a partida guardada, y su comienzo a asignar valores a sus variables.

Fuente: Elaboración propia.

Cuando hemos terminado de asignar las variables, llamamos a la función save game to slot, y pasándole el objeto nuevo que tenemos de partida guardada, ya tenemos almacenada la partida.

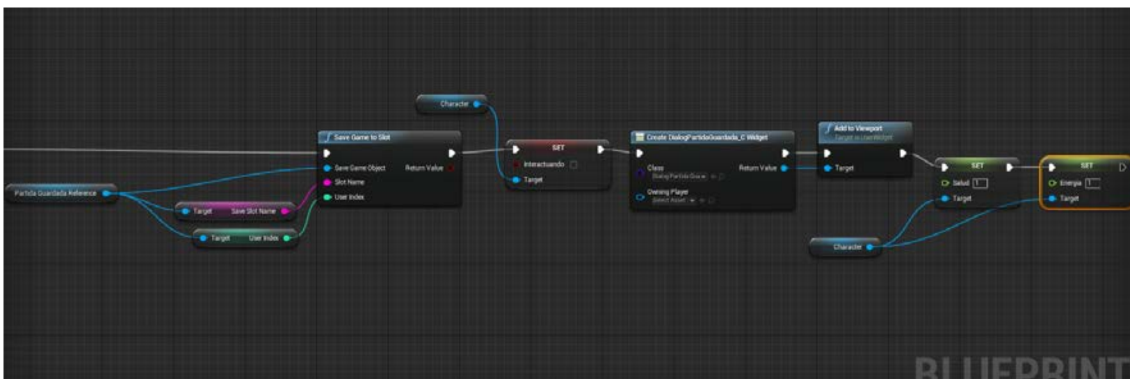


Figura 176. Captura de llamada a la función save game to slot.

Fuente: Elaboración propia.

Luego de esto, lo que hacemos en este blueprint de punto de guardado es llamar al WidgetBlueprint informativo de partida guardada, y restaurar al máximo la salud y la energía del personaje.

5.2.7.2. Cargar partida.

El proceso de carga de la partida sigue un proceso bastante similar al de guardado.

Lo que hacemos para cargar una partida es crear un save game object, castearlo a nuestro blueprint, y almacenar este cast (como en guardado). Una vez hecho esto, llamamos a la función load game from slot, donde indicaremos 0 en nombre y 0 en user index (como se ha decidido

guardar la partida), y el objeto que nos devolverá, lo casteamos al blueprint de nuestra clase save game, y lo asignamos a la referencia que ya teníamos.

Después de esto, ya podemos acceder a todas las variables que habíamos guardado.

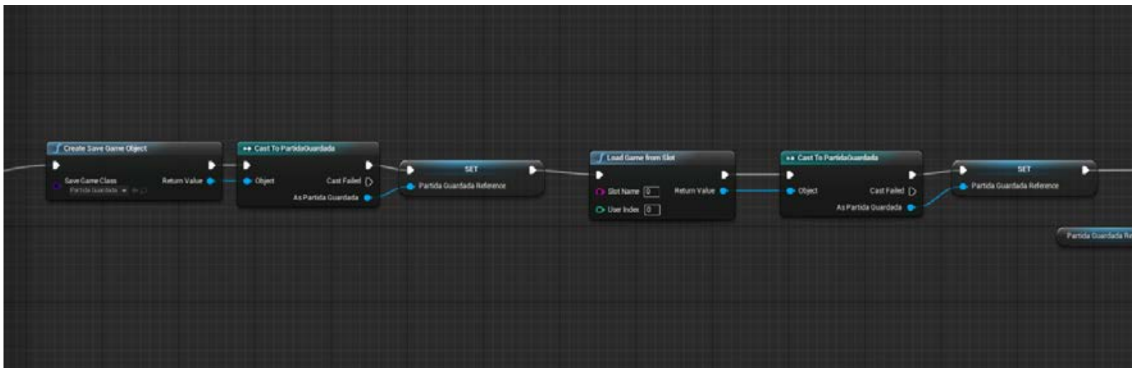


Figura 177. Cargado de una partida.

Fuente: Elaboración propia.

5.2.7.3. Borrar partida.

Para borrar la partida, solamente es necesario llamar a la función delete game in slot, indicándole el slot y el user index, que en nuestro caso es 0 y 0.

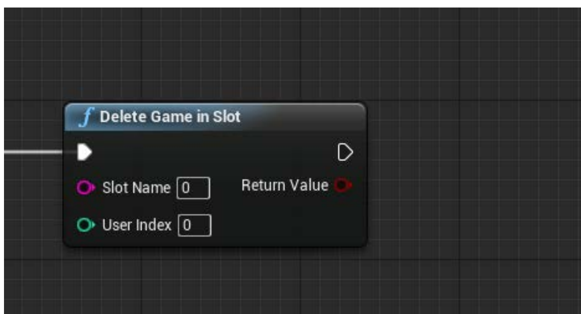


Figura 178. Función de eliminar la partida guardada.

Fuente: Elaboración propia.

5.2.8. Implementación de la IA.

La implementación de la IA definida en el GDD requiere la creación de una serie de componentes básicos en Unreal Engine 4, los cuales se procede a continuación a definir y explicar su set up básico, de modo que más adelante nos centremos en aquellas particularidades que distinguen cada una de las inteligencias artificiales descritas en el documento de diseño.

Los componentes básicos necesarios para la creación de una IA en este motor son 4, que pueden aumentar en función de la complejidad de la IA, son los siguientes:

1. Un Behaviour Tree, que nos permite establecer comportamientos en función de una serie de condiciones, y está preparado para la IA, de forma que permite una gestión de comportamientos más eficiente y gráfica. Este componente lo encontramos en el apartado de Miscellaneous.
2. Un Blackboard que nos sirve para almacenar variables que utilizamos con el Behaviour Tree. Al igual que el Behaviour Tree, este componente también lo encontramos en el apartado de Miscellaneous.
3. Un Blueprint de tipo Character. Este Actor corresponde al personaje que va a ser poseído por el Controller de la Inteligencia Artificial, siguiendo el mismo esquema de Unreal que utilizamos en Arbennig, donde nuestro Player Controller posee el Character Arbennig.
4. Un Blueprint de tipo AIController que poseerá al Blueprint de tipo Character (podría ser de tipo actor si es una IA muy específica, pero no es nuestro caso).

Creados estos componentes, comenzamos a añadirle su lógica de juego. En nuestro caso, la lógica de juego de la IA que vamos a utilizar, consiste en un personaje que permanece quieto hasta que ve al jugador, y entonces lo persigue hasta alcanzarlo.

Para el character de la IA, lo que vamos a hacer es añadirle un trigger que cubra al jugador. A este trigger le vamos a dar la funcionalidad de que si colisiona con el jugador, entonces el jugador cambia su posición a una indicada, que corresponde a la zona donde aparecerá por haber perdido la partida.

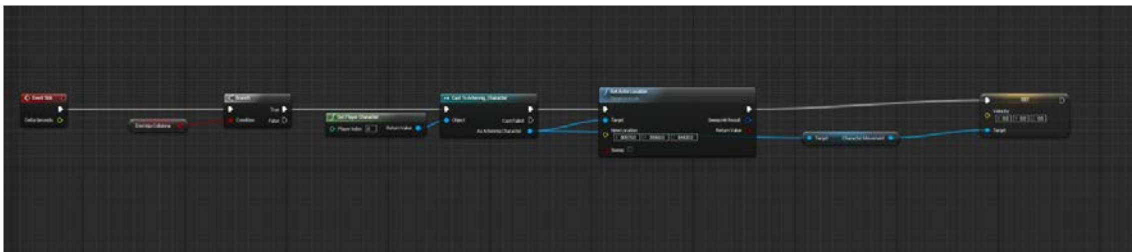


Figura 179. Captura de cuando el enemigo colisiona, lo cambiamos de posición a la indicada.

Fuente: Elaboración propia.

Por otra parte, en el blackboard, creamos 3 variables, una de tipo objeto que se corresponde al objetivo que va a perseguir, y dos vectores, uno para la posición actual, y otro para la posición objetivo a la que se moverá.

Cuando creamos estas variables, guardamos y asignamos el blackboard al Behaviour tree.

El behaviour tree es un grafo con un funcionamiento especial. En él, hay un nodo inicial llamado root, y de este nodo parten otros nodos hacia abajo, que se ejecutarán de izquierda a derecha. En este grafo es posible crear selectores y secuencias, que ejecutan los nodos hijos de izquierda a

derecha hasta que uno termine con éxito (selector) o falle (secuencia), que hacen que se suba en el árbol. Las condiciones y comprobaciones que se realizan en los behaviour tree reciben el nombre de decorators (condicionales) y services(para actualizar el balckboard y hacer checks), mientras que las tareas que queremos que se realicen cuando se cumplan ciertas características, se llaman tasks.

Nuestro Behaviour tree es el siguiente:

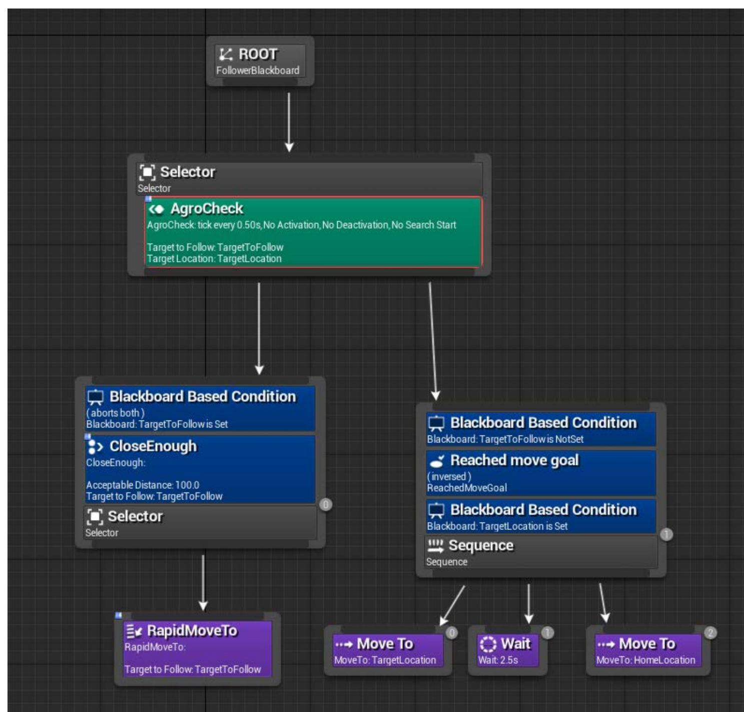


Figura 180. Captura de la lógica general del Behaviour Tree de nuestra IA.

Fuente: Elaboración propia.

Consiste en que, del nodo raíz, tenemos un selector que en cada instante comprobará si el Pawn que va a poseer la IA está asignado, y, si no es así, lo asignará. Además, también se encarga de trazar una serie de líneas para detectar a un objetivo (que se puede perder) para perseguir. De este parten dos blackboards, cada uno para comprobar si el jugador ha sido asignado o no. En caso de haber sido asignado, y de que esté lo suficientemente cerca (Decorator CloseEnough que comprueba distancias), entonces se lanza una tarea de perseguirlo. Por otro lado, si el jugador no está asignado, y no ha llegado todavía a su destino, pero tiene un destino al que moverse (caso en el que nos ha perdido de vista), lo que hace es dirigirse a la posición que tenía previsto ir, esperar 2.5 segundos, y entonces volver a la posición inicial donde se encontraba.

Para que todo esto funcione, desde el editor, lo que hacemos es crear un elemento nav mesh bound, que será la malla de navegación por la que el personaje correrá.

5.2.9. Implementación del flujo de juego general de juego en el Level Blueprint

El flujo de juego general en el level blueprint se divide en varias partes.

Primero de todo, cuando se crea el nivel, se llama al evento begin play, el cual se encarga de lo siguiente:

1. Que el PlayerController posea al Pawn y guardarlo en una variable.
2. Cargar la partida.
3. Crear un array de elementos para aquellos casos donde tengamos varios objetos de una misma clase (para los aros y para la IA), y ocultar todos los elementos del escenario que deban estarlo (aros de los desafíos 4 y 7, folletos del desafío 2, golems del desafío 6, bandera del desafío 8, y las runas del muro de runas).

Una vez hecho esto, la función tick comprobará si estamos dentro de algún desafío o no. Para esto nos hemos creado un enumerador con 9 valores, 8 para indicar los desafíos, y uno más, que se trata del valor none, para cuando no está en ninguno. Dado que la clase character la vamos a utilizar de puente por su fácil acceso desde cualquier blueprint del juego, creamos una variable tanto en el level blueprint como en el character de este tipo de enumerador.

En cada instante, la lógica que seguimos es la siguiente:

1. Comprobar el valor del enumerador, y en caso de ser distinto de none, significa que hay un desafío.
2. Comprobar si la variable del enumerador del character está ya actualizada al desafío actual, y en caso contrario, asignarle el valor al de nuestro level blueprint.
3. Inicializar el desafío correspondiente.
    a. Para los desafíos 1, 3, y 5, no es necesario inicializar nada.
    b. Para el desafío 2 se hacen visibles los folletos que vamos a entregar en el poblado.
    c. Para los desafíos 4 y 7 creamos el widget del temporizador de aros, lo almacenamos en una variable, y lo añadimos a la viewport, entonces, de este widget inicializamos los aros conseguidos a 0, le asignamos los aros objetivo a conseguir pasando la longitud del array de aros que hemos creado en begin play, indicamos el tiempo para superar el desafío, y por último, el destino de teletransporte para cuando el desafío termine.
    d. Para el desafío 6 hacemos visible la bandera que debe conseguir el jugador y los golems.
    e. Para el desafío 8, se hace visible la bandera que debe conseguir el jugador.

4. Comprobar si se cumplen las condiciones para terminar cada uno de los desafíos, y cuando así sea, llamar a un evento custom que se encarga de hacer el spawn de la runa correspondiente al desafío que se ha superado, y mostrar el Widget Blueprint de desafío superado.
    a. Los desafíos 1 y 5 llaman directamente al evento finalizar desafío, porque los desafíos se inician cuando acaba la conversación con el tótem.
    b. El desafío 2 llama a finalizar desafío cuando todos los folletos han sido entregados. Para determinar esto, tenemos un flag que se pone a true cuando un folleto ha sido entregado, y también una variable con la referencia al trigger donde está el jugador, cuando se entrega un folleto, lo añadimos a un array de booleanas en el level blueprint, y destruimos el trigger en el que se encuentra el personaje para que no pueda volver a entregar folletos en él, cuando la longitud del array alcanza la cantidad de folletos, entonces quiere decir que todos los folletos han sido entregados.
    c. El desafío 3 lo hace cuando se han colocado todos los matorrales donde se indica en la misión. Al terminar el desafío 3, porque el trigger donde se colocaban ya los contiene todos, se pone a true una variable creada en el character, que la leemos con el level blueprint.
    d. Los desafíos 4 y 7 llaman al evento de finalizar desafío cuando se han atravesado todos los aros en el tiempo indicado. La lógica de estos desafíos consiste en determinar cuándo un aro ha sido atravesado, el cual pondrá un flag a true, de modo que en cada instante, se recorre el array de aros, y suma a una variable integer +1 por cada flag de aro atravesado, y, al completar el recorrido, actualiza la variable del widget del temporizador y los aros restantes. Cuando el widget nos avisa de que el desafío ha terminado y cómo, si ha sido con éxito, llama al evento de finalizar desafío, mientras que si finaliza sin superarlo, se reinicia todo, estableciendo los enumeradores de desafío de tótem a none, estableciendo a false todas las variables de aros atravesados, y ocultando los aros en el juego.
    e. Para el desafío 6 llamamos a fianlizar desafio cuando una variable creada en nuestro character se establece a true, tras haber colocado donde nos indica el tótem la bandera que hay que recoger. También, se destruyen los actores del desafío si se ha superado. Por otro lado, en caso de que el desafío no se haya superado (lo cual se indica con otra variable desde el character), lo que hacemos es reiniciar el desafío, posicionando y ocultando de nuevo la bandera en su lugar, y lo mismo con los golems. Además también de cambiar los enumeradores a estado de none, y mostrar un widget de que el desafío no ha sido superado.

5. Comprobamos si el jugador muere, y si es así, abrir el nivel de fin de partida y remover el HUD.
6. Comprobar si una runa ha sido spawneada, y si es así comprobar si el jugador la ha recogido ya o no revisando si es un objeto válido. Dependiendo del desasfío que fuera el que realizábamos, destruimos el trigger que nos permitía hablar con el tótem de dicho desafío, y luego de esto, ponemos los enumeradores de desafío activo en none.
7. Por último, comprobamos si el jugador ha abierto la puerta final o no, esto es, comprobando si se ha puesto a true la booleana que tenemos en el trigger de colocar las runas, y si está en true, llamamos a girar la puerta y hacemos visibles las runas pegadas al muro final.

6. Conclusiones

Tras haber finalizado el proyecto, podemos destacar una serie de conclusiones sobre el mismo y también, realizar una evaluación sobre las herramientas utilizadas tras varios meses de uso.

Cabe destacar que se ha podido cumplir en mayor o menor medida con los objetivos postulados en el apartado 3 de la memoria, es decir, se ha creado el diseño completo de un videojuego y una vez creado, se ha implementado utilizando el motor de Unreal Engine 4, cubriendo los aspectos de creación del mundo de juego y su lógica completa, también se le ha añadido menús de juego, cinemáticas de prólogos y epílogos, se ha compatibilizado con el periférico oculus rift DK1, y se ha sonorizado el proyecto. Con todo esto, podemos determinar que el resultado obtenido ha sido correcto.

Aparte de esto, tras varios meses de desarrollo utilizando el sistema de scripting de blueprints, se puede afirmar que es un sistema extremadamente potente, y que nos permite realizar videojuegos completos al 100% sin necesidad de tocar nada de código. Aunque eso sí, el sistema de blueprints es scripting gráfico, es decir, lo que tenemos en el código, está en Blueprints representado con nodos para tenerlo todo de forma más visual. Y con esto quiero hacer hincapié en que una persona sin conocimientos de programación o simplemente conocimientos básicos, probablemente lo tenga muy complicado para desarrollar proyectos, de modo que los perfiles de diseñadores o artistas que quieran hacer uso de él, deben tener conocimientos de programación orientada a objetos y de la lógica de ejecución que siguen los programas.

Por otra parte, el motor tiene una curva de aprendizaje a la que resulta un poco complicado en algunos aspectos progresar. Si bien es cierto que con el motor de UDK ha mejorado indudablemente, la curva de aprendizaje que ofrecen motores como Unity es relativamente más sencilla, sobre todo para empezar a utilizarlo.

En cuanto a las posibilidades que ofrece el motor, queda evidente por qué se trata de uno de los motores más relevantes y famosos del mundo, con todas las herramientas que incluye, la interfaz actual, mucho más fácil de entender que en UDK, y la potencia y resultados visuales que nos permite mostrar son bastante notables.

Se ha de hacer mención a que, aunque previamente ha realizado proyectos con otros motores para videojuegos, incluyendo Unity y UDK, nunca hasta ahora había realizado un proyecto con UE4, y todas las mecánicas y apartados visuales, y todo en general, ha sido aprendido desde 0 en este motor.

Como conclusión negativa, he de indicar que durante el desarrollo, el motor ha sufrido constantes cierres forzosos, llegando en ocasiones al nivel de unos 10 diarios en los días que me dedicaba al proyecto por completo. Además de esto, normalmente cuando el proyecto se cierra inesperadamente, no es posible recuperar los avances realizados, así que es muy conveniente guardar por cada pocos cambios que se realicen en el proyecto.

Además de esto, se ha hecho de notar que aunque las capacidades de un ingeniero multimedia permiten realizar con éxito un videojuego por completo, las habilidades artísticas que posee están relativamente limitadas, a no ser que se lleve una práctica y afición por ellas, por lo que a la hora de desarrollar videojuegos con un corte más profesional, convendría realizar algún tipo de formación complementaria artística si se quiere poder desarrollar proyectos de una sola persona.

6.1. Propuestas de mejora y trabajo futuro.

Como propuestas de mejora y trabajo futuro, entre otros aspectos, el principal consistiría en realizar un modelado y animación del personaje en primera persona, cuya malla y animación son las que ofrece por defecto el motor para proyectos FPS.

Por otro lado, ofrecer la posibilidad al jugador de controlar el volumen por separado de la música y de los sonidos del juego, sustituir los assets gratuitos de unreal utilizados en el proyecto por assets propios creados con un programa de modelado como 3ds max, mejorar los controles de juego y añadir compatibilidad con el mando de Xbox también serían propuestas de mejora.

Además, para trabajos futuros, se intentará mejorar las habilidades donde se ha flaqueado, que son en el apartado artístico, o realizarlos junto con personas que tengan habilidades artísticas profesionales.

7. Bibliografía y referencias

7.1. Introducción.

[1] Historia de los videojuegos:

https://es.wikipedia.org/wiki/Historia_de_los_videojuegos#Balance.2C_presente_y_futuro_de_los_videojuegos

[2] Facturación de los Videojuegos:

http://elpais.com/diario/2008/04/09/radiotv/1207692005_850215.html

[3] Destiny videojuego más caro de la historia:

http://www.abc.es/tecnologia/videojuegos/20140909/abci-destiny-gameplay-videojuego-RPG-fPS-activision-201409091206.html

[4] Flappy Bird Éxito:

https://es.wikipedia.org/wiki/Flappy_Bird

[5] Capacidades Ingeniero Multimedia por la Universidad de Alicante:

http://cvnet.cpd.ua.es/webcvnet/planestudio/planestudiond.aspx?plan=C205

7.2. Marco Teórico o Estado del Arte.

[6] Definición de videojuego por Bernard Suits:

The Grasshopper: Games, Life and Utopia, Bernard Suits, 1978.

[7] Definición de videojuego por Wikipedia:

https://es.wikipedia.org/wiki/Videojuego

[8] Historia de los motores

Game Engine Architecture, Jason Gregory, 2015 (second edition)

[9] Unreal Engine se crea en 1998:

https://es.wikipedia.org/wiki/Unreal_Engine

[10] El motor Source se presenta en 2004:

https://es.wikipedia.org/wiki/Source

[11] Lista de motores de videojuegos:

https://en.wikipedia.org/wiki/List_of_game_engines

[12] Logo Unreal Engine:

https://upload.wikimedia.org/wikipedia/commons/e/ee/Unreal_Engine_logo_and_wordmark.png

[13] Logo Unity:

https://upload.wikimedia.org/wikipedia/commons/5/55/Unity3D_Logo.jpg

[14] Logo CryEngine:

https://upload.wikimedia.org/wikipedia/commons/8/8d/CryEngine_Nex-Gen(4th_Generation).png

[15] UE4 vs Unity vs CryEngine:

http://blog.digitaltutors.com/unity-udk-cryengine-game-engine-choose/

[16] Consejos elección de motor en diapositivas de la asignatura "Motores para videojuegos" de Videojuegos 2.

https://moodle2014-15.ua.es/moodle/pluginfile.php/18327/mod_resource/content/4/vii-08-motores.pdf

[17] Arquitectura básica del motor

https://docs.unrealengine.com/latest/INT/Programming/UnrealArchitecture/index.html

[18] Imagen del diagrama de clases del motor

https://docs.unrealengine.com/latest/images/Gameplay/Framework/QuickReference/GameFramework.jpg

[19] Información sobre Blueprints

https://docs.unrealengine.com/latest/INT/Engine/Blueprints/index.html

https://docs.unrealengine.com/latest/INT/Engine/Blueprints/Overview/index.html

7.3. Metodología

[20] Qué es kanban y cómo se utiliza:

http://www.javiergarzas.com/2011/11/kanban.html

7.4. Documento de Diseño del Videojuego.

[21] Plantilla GDD de Fundamentos de los Videojuegos, por Fidel Aznar.

[22] Logo PEGI:

https://upload.wikimedia.org/wikipedia/commons/thumb/4/44/PEGI_12.svg/426px-PEGI_12.svg.png

[23] Información PEGI12:

http://www.pegi.info/es/index/id/96/

[24] Definición de jugabilidad por Wikipedia:

https://es.wikipedia.org/wiki/Jugabilidad

[25] Las especificaciones mínimas de software y hardware son las descritas en el FAQ de la web de UE4:

https://www.unrealengine.com/faq

7.5. Desarrollo e implementación.

[26] Pipeline de importación de mallas estáticas en UE4:

https://docs.unrealengine.com/latest/INT/Engine/Content/FBX/StaticMeshes/index.html

7.5.1. Bibliografías frecuentemente consultadas

La web de documentación de Unreal Engine:

https://docs.unrealengine.com/latest/INT/index.html

El canal de youtube de Unreal Engine 4 oficial:

https://www.youtube.com/channel/UCBobmJyzsJ6Ll7UbfhI4iwQ

El foro de Unreal Engine 4:

https://forums.unrealengine.com/

UE4 AnswerHub:

https://answers.unrealengine.com/

Especificaciones Audio UE4:

https://docs.unrealengine.com/latest/INT/Engine/Audio/WAV/index.html