Trabajo de Fin de Máster

Máster Universitario en Desarrollo de Videojuegos

Recreación de ambientes subacuáticos para un videojuego de supervivencia

Re-creating underwater environments for a survival videogame

Christian Jesús Pérez Hernández

La Laguna, 4 de julio de 2024

D. **Jose Ignacio Estévez Damas**, con N.I.F. 43.786.097-P profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

C E R T I F I C A

Que la presente memoria titulada:

”*Recreación de ambientes subacuáticos para un videojuego de supervivencia*”

ha sido realizada bajo su dirección por D. **Christian Jesús Pérez Hernández**, con N.I.F. 51.153.935-A.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 4 de julio de 2024

Agradecimientos

En primer lugar, me gustaría agradecer a mi tutor Jose Ignacio Estévez Damas por su apoyo, dedicación, tutoría y por todo el conocimiento que he podido adquirir.

Me gustaría agradecer también al profesorado del Máster por su apoyo en la formación durante el Máster.

Y en especial me gustaría agradecer a mis padres y a mi novia, Yurena, ya que sin ellos no habría sido posible.

Licencia

Resumen

El objetivo de este trabajo ha sido abordar la creación de un videojuego de supervivencia en un ambiente submarino. Para ello será necesario abordar y explorar la capacidad de un motor de videojuegos moderno para utilizar las técnicas requeridas para recrear ambientes subacuáticos. En este trabajo se trata especialmente la creación de materiales, la iluminación y el movimiento de los personajes. Todo ello, en el contexto de un videojuego de supervivencia en tercera persona que transcurre en este entorno.

El motor elegido ha sido Unreal Engine en su versión 5.3, y mediante su sistema de desarrollo se ha creado un nivel extenso, donde el terreno se ha modelado utilizando el modo paisaje de dicho sistema, incluyendo una gran variedad de relieves que fomentan la exploración del jugador.

Los videojuegos del género de supervivencia se caracterizan por tener una o más mecánicas que obligan al jugador a estar pendientes de los recursos disponibles, en este trabajo se ha desarrollado la necesidad de salir a la superficie para recuperar el oxígeno.

Además, se ha implementado un sistema de combate mediante el cuál el jugador pueda defenderse de un atacante, en este caso un tiburón, que actúa como enemigo final. Para dicho tiburón, se ha empleado un árbol de decisiones que lo dota de comportamientos como deambular, perseguir o atacar al jugador.

Palabras clave: *videojuegos, entornos subacuáticos, árboles de decisiones, videojuego de supervivencia*

Abstract

The objective of this work has been to address the creation of a survival video game in an underwater environment. To achieve this, it's necessary to approach and explore the capability of a modern video game engine to utilize the required techniques to recreate undewater environments. This work particularly focuses on the creation of materials, lighting, and character movement. All of this is within the context of a third-person survival video game that takes place in this environment.

The chosen engine is Unreal Engine 5.3, and through its development system, an extensive level has been created. The terrain has been modeled using the landscape mode of this system, including a wide variety of reliefs that encourage player exploration.

Survival genre video games are characterized by having one or more mechanics that require the player to be mindful of the available resources. In this work, the need to go out to the water surface to replenish oxygen has been developed.

Furthermore, a combat system has been implemented which allows the player to attack and defend against an enemy, a shark which serves as the final boss. This shark's behavior is controlled by a behavior tree, endowing it with behaviors such as wandering, chasing or attacking the player.

Keywords: video games, underwater environments, behavior trees, survival video game.

Índice general

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

La relación entre los niveles acuáticos en los videojuegos y los jugadores es difícil desde sus comienzos. Los primeros niveles acuáticos tenían unos controles lentos y pesados que frustraban al jugador, pues los niveles se hacían muy largos y tediosos. Las mecánicas de combate en esta clase de niveles tampoco fueron fáciles de diseñar. Por ejemplo, esquivar los ataques de los enemigos requería un mayor grado de planificación dada la lentitud en las maniobras de escape. Además del movimiento, hay también otros aspectos diferenciadores tanto en la jugabilidad como en el arte, como por ejemplo, la distribución y tipología de los enemigos en el nivel, la estructura de este (la dimensión vertical cobra un mayor protagonismo), las condiciones para mantener al personaje con vida (recuperación de oxígeno), los sonidos del ambiente y la música o el movimiento de la cámara. Estos aspectos tienen que estar debidamente diseñados para hacer de un mundo subacuático una experiencia que pueda ser disfrutada por el usuario.

Por lo tanto, los juegos en ambientes subacuáticos son un desafío en cuanto al diseño y aprovechamiento de la tecnología para crear un ambiente con las características adecuadas. Pero también, son un contexto en el que se puede aprender mucho acerca de la creación de videojuegos, así como de las herramientas que lo facilitan. Esta combinación de desafío y oportunidad de aprendizaje es la principal motivación del proyecto.

A continuación, en este capítulo se describen algunos antecedentes de esta clase de juegos. Luego se concretan los objetivos que han conducido al desarrollo del videojuego presentado. El capítulo segundo es el documento de diseño del juego, donde se esboza el producto que se desarrollará. El capítulo tercero se centra en describir las herramientas elegidas para el desarrollo, en concreto algunas de las características y editores de assets del motor Unreal Engine 5.3. capítulo cuarto profundiza en la implementación del juego, destacando aquellos aspectos especialmente relacionados con las particularidades de la ambientación submarina. El capítulo quinto analiza las observaciones de un conjunto de jugadores que han podido acceder a la demo terminada. El capítulo sexto es una recapitulación final donde se extraen unas conclusiones y se plantean mejoras, líneas de desarrollo y alternativas tras el resultado conseguido. Además, Se ha creado un entrada en el portafolio con las instrucciones de instalación de la demo [26].

1.1. Antecedentes

Se podrían considerar los videojuegos “Super Mario Bros” [24], “Las Tortugas Ninja” [20] y “Sonic the Hedgehog” [29] precursores representativos de este tipo de niveles. Es

por ello que se presenta a continuación un análisis de algunos aspectos en los que estos videojuegos destacaron (para bien o para mal) al enfrentarse a este nuevo medio.

En los niveles acuáticos de “Super Mario Bros”, la flotabilidad de Mario es muy baja, por lo que se hunde rápidamente. El jugador deberá pulsar el botón “A” repetidamente para ascender, aunque lo hará a menor velocidad. Este tipo de movimiento propicia esquivar a los enemigos por debajo ya que es más rápido descender. Pero, hay que tener en cuenta los obstáculos que pueden impedirlo y a los enemigos, que son inmortales, excepto que se posea el poder de “la flor de fuego”. Si además, añadimos a esto que el movimiento de los enemigos “calamares” es impredecible, se trata de un movimiento en diagonal a intervalos irregulares, podemos concluir que los niveles acuáticos son considerablemente más complicados que otros en este videojuego.

Sin embargo, con el salto del 2D al 3D en “Super Mario 64” [23] observamos en el nivel *Dire Dire Docks* nuevas carácterísticas en lo referente a la gestión del oxígeno y la música[6] que pueden considerarse buenas prácticas. A lo largo del nivel están dispuestos varios cofres y monedas que regeneran el oxígeno al jugador, hay además zonas exteriores donde se puede respirar, evitando o al menos reduciendo el agobio y frustración al jugador. En cuanto a la música, se presenta una melodía dinámica a la que se le sumarán instrumentos según la situación del jugador.

“Las Tortugas Ninja” y “Sonic the Hedgehog” comparten una elevada dificultad, en gran medida provocada por un exceso de amenazas o enemigos en los niveles acuáticos, hasta el punto que en ocasiones da la sensación de que no se han tenido en cuenta los problemas asociados a las diferencias respecto a la movilidad de los personajes en este nuevo medio.

Figura 1.1: “Las Tortugas Ninja”, pasillo angosto repleto de algas que representan una amenaza

En “Las Tortugas Ninja”, el nivel que transcurre en una presa 1.1 es recordado por una gran cantidad de aficionados como uno de los momentos más odiados y frustrantes de la entrega. Aquí nos enfrentaremos a angostos pasillos, interminables y repletos de algas que nos quitarán vida si las tocamos, añadiendo dificultad por los numerosos giros que hay que realizar.

El caso de “Sonic the Hedgehog” es similar 1.2. Para respirar bajo el agua, Sonic utiliza unas burbujas que puede recoger por el nivel. Estas duran 60 segundos, pero se romperán si algún enemigo tiene éxito en el ataque, teniendo que buscar otra en 5 segundos o se perderá la partida. Durante estos 5 segundos se reproduce una música estridente, que interrumpe la melodía del nivel. La distribución de burbujas por la escena en algunos puntos es insuficiente, provocando que el jugador no tenga margen de error. Además, no existe un indicador del tiempo restante de oxígeno, salvo que se esté en los 5 segundos restantes, esto hace que el jugador no pueda planificar la ruta o estrategia para superar el nivel, teniendo que repetir muchas veces este para terminarlo.

Por último, si tenemos en cuenta que una de las características principales de Sonic es la velocidad, los niveles subacuáticos hacen que se pierda en parte la identidad del mismo.

Figura 1.2: “Sonic The Hedgehog 2”: algunos de los obstáculos en el nivel

Resulta también interesante, explorar aquellas sagas como “Grand Theft Auto” [27], “Assasins Creed” [30] o “The Elder Scrolls” [2], sagas tan extensas que han podido pasar por las distintas etapas de los niveles subacuáticos. En “*GTA*” y “Assasins Creed” entrar en el agua suponía la muerte del personaje, en “Assasins Creed II” se añade la posibilidad de nadar en la superficie, y no es hasta “Assasins Creed Black Flag” que la exploración submarina recibe un mayor peso y el jugador puede bucear para encontrar tesoros hundidos. De manera similar ocurre con “*GTA*” y “The Elder Scrolls”, sin embargo, este último ya contaba con mecánicas, aunque toscas de natación y buceo.

1.1.1. “Subnautica”

Poniendo el foco en videojuegos más recientes, estos problemas se han ido solucionando y se comienzan a desarrollar juegos donde el agua recibe un papel principal, como es el caso de “Subnautica” [31].

“Subnautica”, es un videojuego de supervivencia de mundo abierto cuya aventura transcurre principalmente en el océano 1.3. Tras un naufragio, el jugador despierta en una cápsula de salvamento y deberá sobrevivir y explorar el mundo para descubrir los

motivos del naufragio. Para ello, se pueden recolectar materiales del fondo marino y utilizarlos para construir objetos, como crear vehículos para aumentar la movilidad y capacidad de almacenamiento, o tanques de oxígeno para aguantar más tiempo bajo el agua.

Como se menciona en [15], el aspecto fundamental y característico de “Subnautica” es la historia, que se desarrolla en conjunción con el progreso del jugador. A medida que se consiga mejor equipamiento para explorar los distintos biomas, se irán descubriendo partes de la historia, como otras cápsulas de salvamento, el motivo del accidente y otros aspectos de la trama.

En cuanto al apartado estético, “Subnautica” logra crear una buena inmersión mediante técnicas de iluminación como las cáusticas, los rayos de luz que atraviesan la superficie del océano o la absorción de luz por parte del agua. Además, la ilusión de inmersión en un mundo extraterrestre se refuerza por la utilización de personajes como monstruos marinos, plantas insólitas y multitud de seres bioluminiscentes.

La música, también juega un papel en la inmersión del jugador. Se crearon para “Subnautica” 52 obras, que se reproducen en función del estado del juego, pudiendo diferenciarse 3 géneros principales, música calmada o intrigante para acompañar la exploración y música de terror en los enfrentamientos con los monstruos.

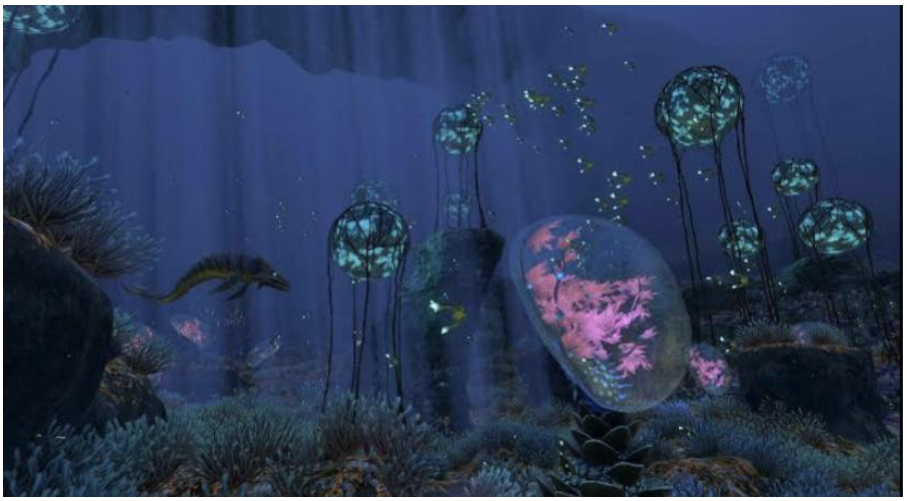


Figura 1.3: Apartado visual de "Subnautica"

1.2. Objetivos generales

En este trabajo, se utilizarán las herramientas de desarrollo que ofrece un motor de videojuegos en el estado del arte, para conseguir una estética acorde a un ambiente submarino, tanto en la iluminación, creación del terreno, materiales y mallas 3D. Además, se desarrollará un videojuego de supervivencia en tercera persona, donde el personaje deberá acabar con un tiburón que custodia unas ruinas para terminar el juego. Mientras, deberá sobrevivir teniendo en cuenta el nivel de oxígeno para no morir ahogado.

1.3. Objetivos específicos

Se establecieron los siguientes objetivos específicos:

- Creación de un terreno subacuático, esto incluye tanto el *landscape*, como el agua, inclusión de mallas 3D adecuadas para la decoración y creación de materiales para el terreno.
- Creación de una iluminación adecuada para el entorno.
- Creación de un personaje que pueda nadar y bucear para explorar el nivel.
- Creación de una mecánica de supervivencia como el hambre o el oxígeno.
- Creación de un sistema de combate para el personaje y el enemigo.
- Creación de un enemigo subacuático dotado de cierto nivel de inteligencia.

Capítulo 2

Documento de diseño del videojuego

Mediante un documento de diseño de videojuegos se recopila toda la información acerca del diseño. En él se pueden incluir textos, imágenes, contenido multimedia... para reflejar con mayor exactitud el diseño deseado. Este documento es utilizado por los desarrolladores para guiar el proceso de desarrollo. A continuación, se muestra el documento de diseño de videojuegos para el videojuego desarrollado en este TFM.

2.1. Concepto

Género: Supervivencia.
Plataforma: Windows.
Tecnología: Unreal Engine 5.3.
Público: Dirigido a adolescentes y adultos que buscan una experiencia supervivencia al estilo Subnautica.
Sinopsis de jugabilidad: El jugador deberá sobrevivir a un naufragio en el océano, mientras busca una manera de pedir ayuda.

2.2. Mecánica del juego

Es un juego en 3D con una cámara en tercera persona. El jugador podrá desplazarse utilizando las teclas de movimiento WASD y el ratón (inclinación, rotación), para explorar el entorno que le rodea, mientras, deberá controlar su nivel de oxígeno saliendo a la superficie para no morir ahogado. Podrá defenderse de un tiburón utilizando el click derecho del ratón. Pulsando la tecla E podrá recoger objetos por el nivel.

2.3. Interfaces

El juego debe comenzar con un menú de inicio, donde podrá iniciar o salir del juego 2.1

Figura 2.1: Aspecto del menú principal

Una vez dentro del juego, se utilizará una interfaz 2.2 para que el jugador conozca en todo momento los valores de oxígeno y salud

Figura 2.2: Estadísticas del jugador

2.4. Progreso del juego

El jugador comienza en medio del océano y deberá sumergirse para explorar los alrededores y encontrar alguna manera de pedir ayuda. En la exploración, se encontrará con unas ruinas oceánicas custodiadas por un tiburón, allí deberá buscar una bengala, mientras se defiende del enemigo. Una vez conseguida la bengala, deberá subir a la superficie y utilizarla para pedir ayuda y acabar la partida.

Capítulo 3

Herramientas de desarrollo

Para el desarrollo del videojuego se utilizó el sistema de desarrollo y características del motor de videojuegos 3D Unreal Engine en su versión 5.3, propiedad de la empresa Epic Games. Este motor se caracteriza por el número de herramientas disponibles en su sistema de desarrollo y por mantenerse en el estado del arte del renderizado de gráficos en tiempo real. Además, en el momento de la redacción de esta memoria, la empresa Epic Games promueve su utilización gratuita (incluido todo el sistema de desarrollo) para educación, y para individuos y pequeñas empresas con beneficios inferiores a 1 millón de dolares anuales y no cobra royalties sobre juegos que vendan menos de ese millón de dolares. Dadas estas características y puesto que el objetivo de este proyecto es desarrollar un juego supervivencia en un entorno subacuático, a la vez que conocer las capacidades de los motores modernos en cuanto a la recreación adecuada del ambiente, con realismo y sin sacrificar la jugabilidad, el motor Unreal Engine fue elegido como herramienta de desarrollo del juego.

3.1. Características generales del motor Unreal Enigne

Unreal Engine fue desarrollado por Tim Sweeny para Epic Games en 1998 y desde un comienzo sus desarrolladores tuvieron en mente crear un motor orientado a juegos multijugador en red y que pudiera ser extendido y mejorado por otras personas. En él, se pueden desarrollar videojuegos para multitud de plataformas: sobremesa, realidad virtual, consolas o móvil, así como multitud de géneros. Está muy orientado hacia juegos 3D, y pese a que es posible también desarrollar juegos 2D, no es tan habitual. Aunque su objetivo inicial era la creación de videojuegos, dada la calidad de sus sistema de renderizado de gráficos también se han ganado un hueco en otros sectores, como la visualización de proyectos de arquitectura o la industria cinematográfica y la televisión, por ejemplo, ha sido utilizado en la producción de películas como “The Mandalorian”. Es un motor ampliamente utilizado en la industria del videojuego, siendo la herramienta de elección no solo de los títulos de Epic Games. Por ejemplo, la versión 4 de Unreal Engine está detrás de casi 500 videojuegos comerciales en el periodo entre 2017 y 2023

Unreal frente a otros motores suele incorporar rápido el estado del arte en cuanto a la renderización de gráficos en tiempo real y especialmente en la renderización basada en física. Además, para emular ambientes, como entornos subacuáticos, Unreal incorpora *shaders* para la creación de materiales, que junto con los componentes ya desarrollados como el componente de movimiento aceleran el desarrollo.

Actualmente, Unreal permite dos tipos de programación: a través de un lenguaje visual denominado *blueprints* por una parte, y Unreal C++ Scripting por otra. Mientras que programar con Blueprints aporta rapidez e inmediatez, permitiendo que desarrolladores con no tanta experiencia programando construyan videojuegos completos, Unreal C++ Scripting permite a desarrolladores experimentados utilizar toda la flexibilidad de C++ en combinación con la infraestructura ya existente de las librerías del motor. Estas dos aproximaciones se combinan para balancear el tipo de desarrollo en función de las necesidades: para tareas menos críticas o que no necesiten ejecutarse en tiempo real se puede utilizar blueprints, mientras que C++ puede emplearse para tareas que necesiten de la eficiencia que aporta el lenguaje compilado. Además,la orientación a objetos del sistema de programación de Unreal, permite construir clases base en C++ con funcionalidades centrales que luego pueden ser ampliadas por el mecanismo de herencia en otras clases C++ o Blueprint o por la adición de componentes.

Por otra parte, Unreal Engine ofrece también un cómodo entorno de desarrollo centralizado con múltiples editores, como por ejemplo editores de animaciones, materiales, árboles de comportamiento, sonido, efectos de partículas, etcétera. Entre ellos siempre destacó el editor de materiales y las herramientas de animación. El primero permite construir visualmente “shaders” para el renderizado a partir de bloques (de forma similar a los blueprints), poniendo al alcance de un mayor número de profesionales el desarrollo detallado de los materiales de las escenas. Por otra parte, las animaciones son una parte cada vez más relevante del sistema de desarrollo, incluye un sistema de importación de animaciones desde otros programas, editores para la creación de nuevas animaciones a partir de otras existentes (montajes, mezclas) y un sistema de programación de las animaciones basado en la combinación de máquinas de estado de animación con programación mediante blueprints.

Además, en la versión 5 del motor se incluyen nuevas tecnologías importantes para el renderizado de escenas realistas. Nanite es un sistema que permite trabajar en niveles con mallas compuestas por millones de polígonos sin afectar enormemente al rendimiento del juego, logrando escenas mucho más detalladas. De cara a este proyecto, sin embargo, es más relevante Lumen, que consiste en un sistema de iluminación global que funciona dinámicamente en tiempo real, de modo que la iluminación de la escena incluye reflexiones secundarias en los objetos de la escena afectando al color y las sombras. Finalmente, este motor puede aprovechar la tecnología de trazado de rayos tanto por software como por hardware, lo que se aprovecha en el tratamiento dinámico de la iluminación, para crear sombreas más suaves en la iluminación dinámica o para generar reflexiones más realistas.

Finalmente, Unreal Engine tiene desde el editor, una conexión directa para acceder a los *assets* de Quixel Bridge, assets de gran calidad que pueden ser usados en los proyectos de Unreal. Además, en este proyecto se han utilizado las plataformas de Sketchfab [28] e Itch.io [18] para la obtención de mallas se los animales y para iconos y fuentes para el menú respectivamente.

3.2. Sistema de clases de Unreal y componentes

Unreal posee un robusto sistema para el manejo de objetos en un videojuego. La clase base para cualquier objeto es *UObject*, y de esta parten todos los demás. Estas clases

tiene ciertos beneficios al estar controladas por el sistema de Unreal: como el recolector de basura, sistema de reflexión, serialización, replicación de red, entre otros beneficios que agilizan y facilitan el desarrollo de nuevas funcionalidades. El motor, ya cuenta con un conjunto de clases programadas que están destinadas a conformar parte del *Game Framework*, como es el caso de la clase Actor, que representa cualquier objeto que puede ser colocado en el nivel. En cierta medida, los actores pueden ser entendidos como contenedores de otro tipo de objeto, llamados Componentes, estos son los encargados de añadir principalmente la funcionalidad al actor, definiendo como se mueve, como se renderiza, como reacciona a estímulos externos, etcétera.

3.3. El componente de movimiento

El componente de movimiento es un componente del actor que define formas de movimiento para el actor o el personaje. Existen distintos tipos: el componente de movimiento de tipo proyectil, para simular proyectiles, el componente de movimiento de rotación que ejecuta una rotación continuada a un ritmo definido y el componente de movimiento de personaje.

Este último, establece distintos tipos de movimiento ideados para personajes humanoides, como caminar, nadar, volar o caer. Además, las características de cada movimiento se pueden controlar a través de parámetros, como por ejemplo: la fricción, la flotabilidad o la velocidad.

Este componente es realmente complejo, ya que además, implementa un sistema robusto para la sincronización del movimiento en los videojuegos multijugador. Mediante dicho sistema, el movimiento en local se almacena y se encola para ser enviado al servidor, el servidor también realiza el movimiento del personaje y comprueba que haya sido válido. Si la posición final en el servidor es distinta a la recibida, se envía una corrección al cliente para que ajuste la posición final. Si la posición final es correcta, el movimiento se marca como válido y se envía a los demás *proxies*, que actualizarán la posición del personaje.

3.4. El editor de materiales

Unreal posee un potente editor de materiales 3.1 con el cual crear *shaders* o materiales para el proyecto. Mediante estos materiales, se puede definir el aspecto o las propiedades de la superficie de los objetos del nivel, ya sean mallas estáticas, terrenos, interfaces, etcétera.

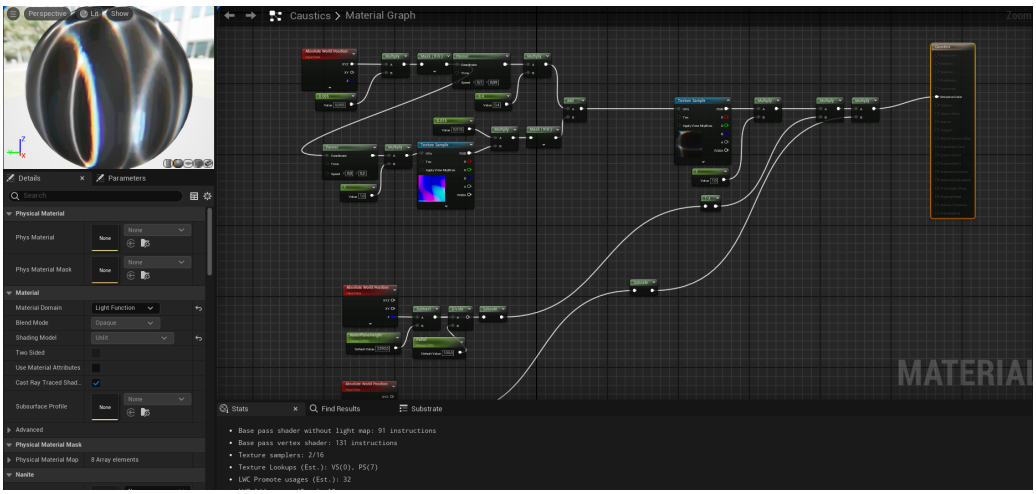


Figura 3.1: Editor de materiales para la creación de las cáusticas

El editor incluye diferentes dominios, los cuales establecen familias de *shaders* que definirán cómo será tratado el material, por ejemplo: opaco, rebotando la luz en su superficie, translúcido, atravesando la luz la superficie, sin iluminación, para que la luz no sea considerada, etcétera.

La creación de un material puede llegar a ser muy compleja, requiriendo grafos muy extensos, esto complica la depuración y el desarrollo de los materiales. Además, es común que ciertas partes de la creación de un material sean reutilizadas en otros. Por ello, el editor permite crear funciones de material para reutilizar el código en otras partes, y además, que esté mejor estructurado. También, permite crear instancias de un material, para modificar la parametrización de una instancia sin la necesidad de crear nuevos materiales.

El editor también contiene una paleta de nodos mediante los que poder realizar operaciones matemáticas o muestreo de texturas de manera rápida y sencilla, agilizando la creación de materiales.

3.5. Editores de animación

El editor de animaciones 3.2 implementa múltiples funcionalidades para tratar con los *assets* de animación para una malla de huesos. En el editor de animaciones se pueden reproducir diferentes *assets* de animación, como las secuencias, espacios de combinación o montajes.

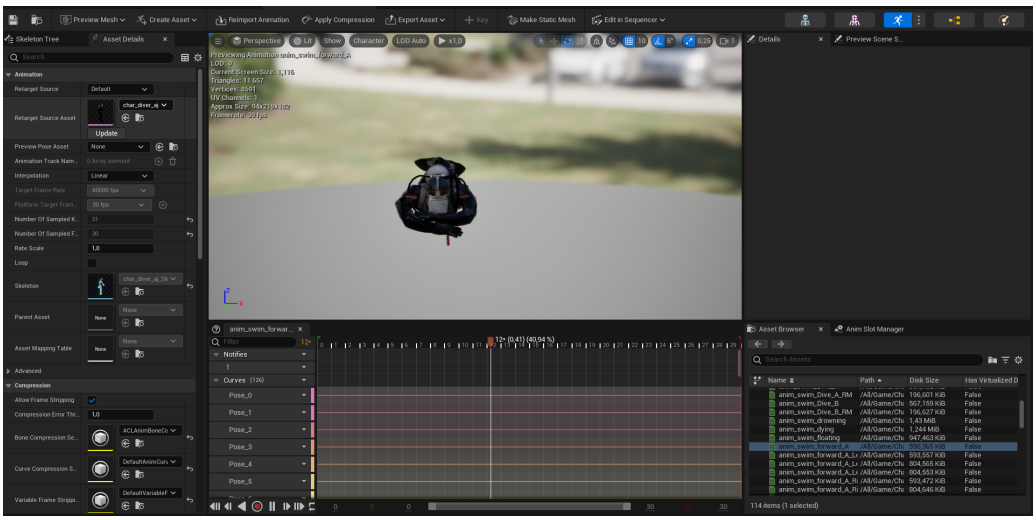


Figura 3.2: Editor de los *assets* de animación

Además, mediante el editor del *blueprint* de animación, se pueden controlar las animaciones de una *Skeletal Mesh*. En este editor, se puede añadir lógica para controlar la animación final que será ejecutada, o mezclar animaciones en función de parámetros, como se puede ver en la figura 3.3.

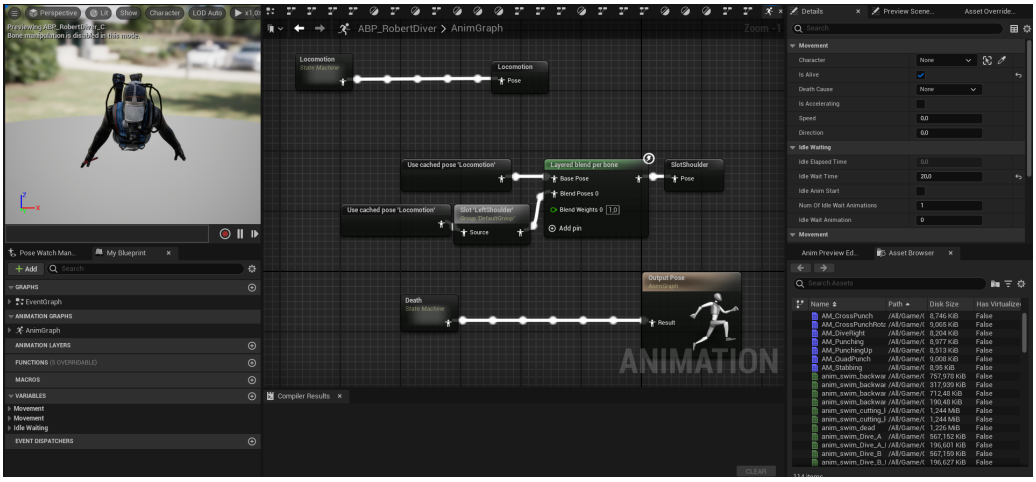


Figura 3.3: *Blueprint* de animación del personaje

Este editor también permite editar o crear animaciones en base a otras, o añadir notificaciones, eventos que se serán disparados cuando la animación reproduzca los *frames* seleccionados, útiles para reproducir sonidos en determinados momentos de la animación entre otras funcionalidades.

3.6. Editores de comportamientos y blackboard para controladores IA

Los árboles de comportamiento permiten establecer tareas que serán ejecutadas por los personajes controlados por la inteligencia artificial, por ejemplo, creando tareas para la persecución de un objetivo o para atacarle.

Para la creación de árboles de comportamiento, se utilizan, aparte de las tareas, nodos de utilidad como el selector, la secuencia, decoradores y servicios. El selector ejecuta de manera secuencial las tareas y para en el momento en el que una de ellas tiene éxito, la secuencia ejecuta de manera secuencial, pero para cuando alguna falla. Los decoradores se pueden acoplar a cualquier nodo y establecen si el nodo puede ser ejecutado en base a condiciones. Por último, los servicios pueden acoplarse tanto a nodos, como al selector o la secuencia y funcionan de manera similar al método *Tick*, ejecutándose repetidamente a cierta frecuencia, mientras que la rama en la que se encuentra esté activa.

Los árboles pueden hacer uso de un componente *blackboard* o pizarra en la que establecen variables que pueden ser luego utilizadas tanto por las tareas, como por los decoradores y servicios, para controlar el flujo de ejecución.

En la siguiente figura 3.4, se utilizan los diferentes nodos y la pizarra para definir un comportamiento base para los animales hostiles.

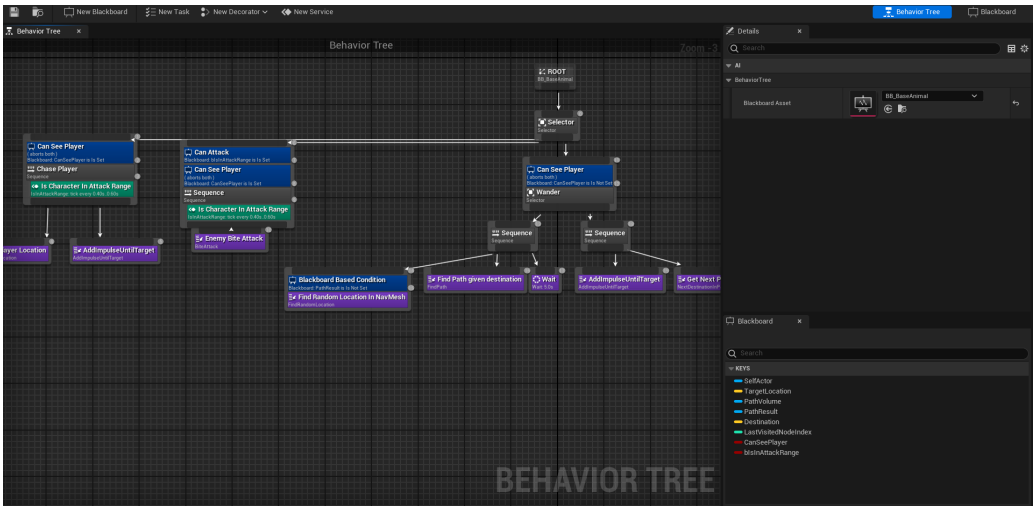


Figura 3.4: Árbol de comportamiento base para los animales hostiles

3.7. Otros elementos: actores de iluminación y volumen de postprocesamiento

El motor de Unreal Engine también aporta actores de iluminación ya creados con los que poder simular entornos. Se resumen a continuación, los actores más importantes en cuanto a la iluminación se refiere.

La luz direccional permite simular una fuente de luz que emite desde una distancia muy lejana, por lo que se puede considerar que los rayos procedentes de la misma son paralelos. Por sus características, este tipo de luz está ideada para simular la luz solar. Además, se pueden modificar sus propiedades para establecer un color de luz, intensidad, generación de sombras, etcétera.

El *Sky atmosphere* permite simular la atmósfera planetaria, para ello, realiza las siguientes técnicas:

- Control de la dispersión de la luz a través de la atmósfera.

- Simulación de la curvatura del mundo al transicionar a una vista aérea a gran distancia del planeta.
- Variación del color del cielo dependiendo de la altitud del Sol.

La Niebla de altura exponencial crea un volumen de niebla, que modifica su densidad en función de la altura, a más profundidad, más densa será la niebla. El crecimiento de la densidad de la niebla es configurable, adaptándose a las necesidades del proyecto. Además, se puede establecer una segunda niebla para añadir otra capa de espesor.

El volumen de post-procesamiento permite definir el aspecto final que tendrá la escena a través de la configuración de sus múltiples propiedades, estas permiten modificar el color del nivel, el mapeado de tonos, la iluminación y otros aspectos. Al volumen se le puede establecer un rango para que afecte a una determinada área, o que no tenga límites y que afecte a todo el nivel.

Por último, el *SkyLight* captura la luz de la escena o de un *cubemap*, un método para mapear entornos a un textura, con la finalidad de utilizar la información de iluminación para iluminar la escena. Este componente también permite ajustar el número de rebotes de la luz, cuantos más rebotes más iluminada se verá la escena.

3.8. Conclusión

En conclusión, Unreal ofrece un desarrollo ágil frente a otros motores, en parte gracias a la multitud de componentes ya desarrollados, que nos permite crear prototipos rápidamente. Además, suele ser de los primeros en incluir el estado del arte en cuando a renderizado de gráficos e incluye varios editores en un entorno centralizado que facilitan la creación de *assets* sofisticados, que van desde materiales, animaciones o sistemas de partículas.

Capítulo 4

Desarrollo del juego

En esta sección, se tratarán diferentes aspectos de la implementación del videojuego. Se comenzó creando un diseño para el terreno con la intención de tener una buena experiencia de juego y progresión para el jugador.

4.1. Terreno

El terreno 4.1 se ha dividido en distintas alturas, dado que a mayor profundidad mayores amenazas tendría que enfrentar el jugador y es por eso, que el mapa tiene una meseta central donde aparece el personaje, a poca profundidad, donde no existen amenazas y hay comida abundante. Esta se encuentra rodeada por llanuras abisales, zonas más profundas con enemigos y con menor sustento. A lo lejos, se pueden observar otras mesetas que el jugador podría visitar desde un inicio sin descender.

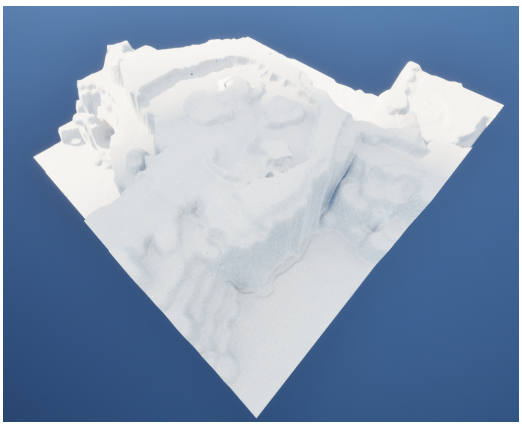


Figura 4.1: Diseño del terreno

Finalmente, tras las llanuras abisales se encuentra una zona mucho más profunda que llega hasta el final del terreno. Esta zona por tanto estaría vacía de animales o vegetación, pues se pretende que el usuario no llegue a explorarla y además, se ha configurado la absorción de luz de tal manera que prácticamente no tenga visibilidad a estas profundidades.

En algún punto de las llanuras abisales se han añadido unas ruinas, que serán el punto al que debe dirigirse el jugador para progresar en la partida.

Para la creación del terreno se utilizó el modo paisaje de Unreal Engine [33], con las herramientas de esculpir, borrar, suavizar y muchas otras especializadas en el modelado de terreno. Inicialmente, se creó un boceto rápido con la herramienta de esculpir, esta herramienta permite elevar el terreno con el click del ratón o hundirlo con la combinación Mayús + click del ratón. Se puede modificar la fuerza con la que trabaja, decrementándola para crear pequeños relieves en las zonas de meseta o incrementándola para generar grandes depresiones en las zonas más profundas. Incluso, permite cambiar la forma de la brocha o añadir la nuestra propia, para crear relieves distintos.

Una vez realizado el boceto, se le añadieron detalles usando las herramientas de erosión. Estas herramientas simulan los efectos de erosión hídrica y erosión por desprendimientos del suelo. Para alisar las mesetas se utilizó la herramienta de aplanado y posteriormente la de esculpir para un resultado más natural.

Por último, para pulir los detalles y evitar problemas con la iluminación como los que se aprecian en la figura 4.3 o relieves dentados como se aprecia en la figura 4.2, se utilizó la herramienta de suavizado con un fuerza baja para retocar el terreno.

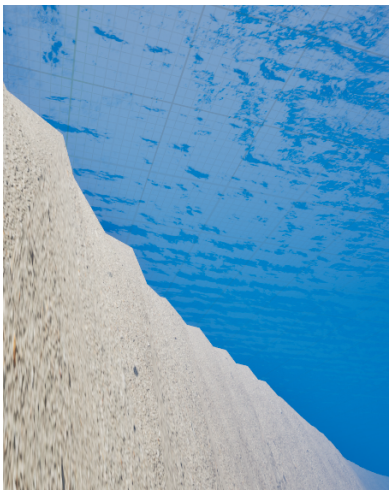


Figura 4.2: Bordes dentados en el terreno

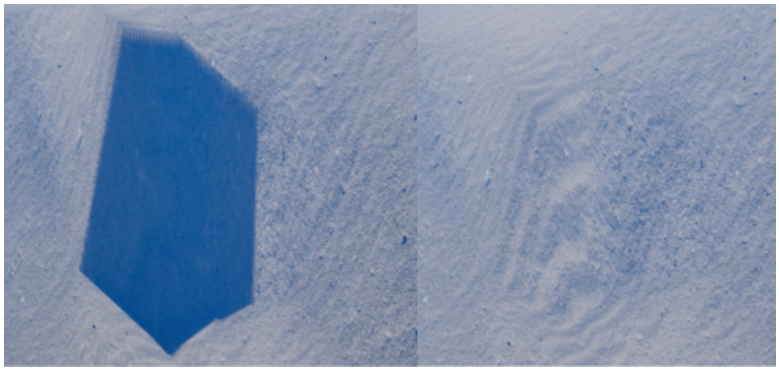


Figura 4.3: Problemas de iluminación

El siguiente paso ha sido añadir materiales al terreno. Para ello, se han creado varios tipos de arenas para el suelo y un material rocoso para las paredes. Estos materiales se combinan en otro material usando la función *Landscape Layer Blend* [21], que luego puede ser usado por la herramienta de pintado en el modo paisaje.

Una problemática muy común al pintar texturas, es que se puede apreciar la repetición de esta en el terreno, siendo esto poco realista. Existen varias técnicas para ocultar este efecto, en este caso, se está mezclando con una textura de ruido para que la repetición sea menos evidente, así como una mezcla de texturas dependiendo de la distancia, cuanto más lejos se encuentra la textura, mayor es su tamaño, como se puede apreciar en las siguientes figuras 4.4 4.5.

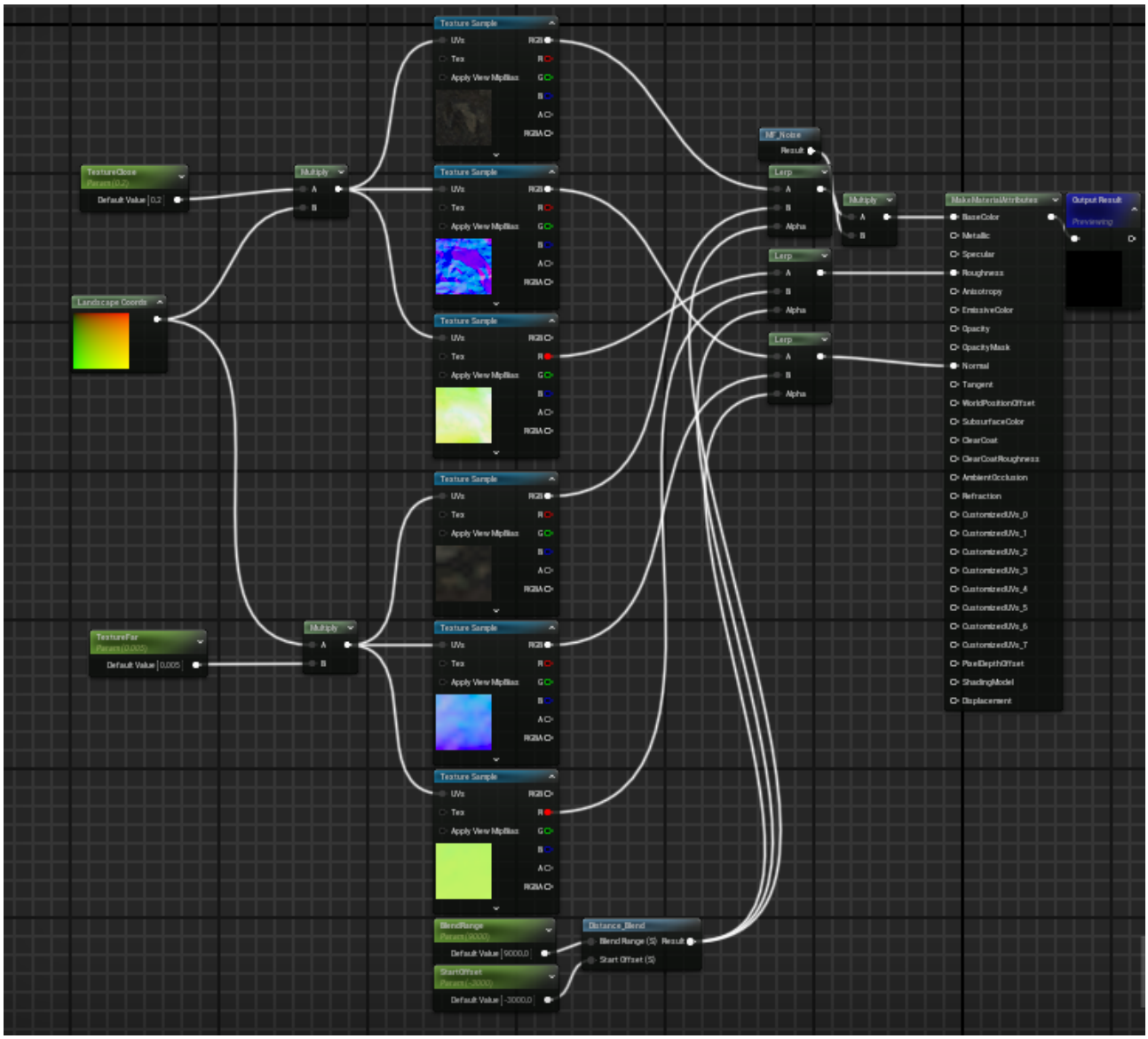


Figura 4.4: Material con técnicas para evitar la repetición de textura

Figura 4.5: Mezcla por distancia del material. En la pared, la textura tiene una mayor escala que en el suelo

4.2. Iluminación y estética

En este videojuego hay que tener en cuenta que existen principalmente dos tipos de iluminaciones, cuando el jugador está bajo el agua y cuando está en la superficie o fuera del agua. Dependiendo de la posición del jugador se activarán o desactivarán configuraciones para simular los dos entornos.

La iluminación del nivel es fundamental, ya que esta será la encargada de dar la sensación al jugador de que se encuentra dentro del agua. Para ello se utilizan diferentes técnicas:

- Afectando a todo el nivel, se ha añadido a este una luz direccional simulando el Sol.
- Afectando especialmente a las escenas donde el personaje está en la superficie, tenemos un *Sky atmosphere* para imitar la atmósfera terrestre y nubes volumétricas añadiendo nubes al nivel.
- Cambiar el color de la iluminación al sumergirse.
- Cuando se está por debajo del agua, añadir un niebla de altura exponencial [13] para reducir la visibilidad simulando la absorción por parte del agua de las ondas de luz
- Efecto de cáusticas en las superficies sumergidas no muy profundas afectadas por la luz.

A continuación analizaremos por separado estos elementos.

4.2.1. Luz Direccional

Se está utilizando la luz direccional simulando la luz solar. La configuración por defecto funciona bien cuando el jugador está fuera del agua, aunque la intensidad de la luz es muy alta cuando se está buceando, pero en lugar de reducirla directamente en sus propiedades, se utilizará un material de post procesado, que se explicará posteriormente

para cambiar el color de luz cuando se esté bajo el agua. Al cambiar el color de la luz da la sensación que la intensidad de la luz es menor.

Además, se hicieron pruebas utilizando el parámetro de temperatura para controlar el color de luz en la escena. Este valor puede variar desde 1700 Kelvin hasta los 12000 Kelvin, obteniendo tonos rojizos y anaranjados cuando el valor es próximo a 1700 Kelvin y tonos azules cuando el valor se acerca a 12000 Kelvin. Los cambios en este valor, aunque notables no producen el resultado deseado, pero puede ser útil para añadirle matices al color de la luz.

El color de la luz se establece en un azulado muy ligero y muy próximo al color blanco, pues valores muy altos de azul reducen demasiado la visibilidad del nivel en conjunción con la niebla, haciendo muy fácil desorientarse. Además, para aumentar el tamaño del Sol, se ha incrementado el valor del parámetro ángulo de la fuente de 0,5357 a 4, de esta manera se consigue que la reflexión de los rayos del Sol sea más evidente en el agua como se puede observar en la siguiente figura 4.6.

Figura 4.6: Reflexión de los rayos de la luz direccional en el agua

4.2.2. Niebla de altura

Mediante la Niebla de altura exponencial simularemos la absorción de luz del agua. El océano absorbe las longitudes de onda más largas, empezando por el rojo y naranja, pero no es capaz de absorber las longitudes de onda más cortas como el azul o el violeta, es por ello que el mar está compuesto en su mayoría por azules y violetas [5].

Normalmente, a partir de los 200 metros de profundidad no existe prácticamente luz [22]. Esta zona desde los 200 metros hasta los 1000 metros, que se conoce como Mesopelágica se caracteriza porque aunque penetra algo de luz solar, esta es insuficiente para que los organismos puedan realizar la fotosíntesis.

Utilizando la niebla de altura exponencial podemos simular este efecto, en este caso a partir de las llanuras que rodean las mesetas dejará de llegar la luz solar 4.7. Este componente, tiene ciertas propiedades que resultan útiles, como la densidad de la niebla

tanto global como secundaria, pues contiene la posibilidad de activar una segunda capa que aumente el grosor de esta o el factor de crecimiento de la niebla, que establece cuan rápido aumenta la densidad en función de la altura. Para que la reducción de visibilidad sea progresiva, no se reduzca demasiado rápido y permita ver a baja profundidad, se utiliza un factor de crecimiento relativamente bajo para las dos capas de niebla, 0,2 para la capa secundaria y 0,8 para la global. Ambas nieblas están a 0,05 de densidad.

Figura 4.7: Niebla de altura exponencial alrededor de las ruinas y niebla volumétrica

Finalmente, se está usando la opción de niebla volumétrica para generar rayos de luz visibles provenientes del Sol cuando se produce oclusión por parte de otros objetos 4.8. Se utiliza un albedo azul claro para el color de los rayos y se establece una distancia de comienzo de 200cm a partir de la cual se genera niebla volumétrica, de esta manera el jugador tiene mayor visibilidad en un pequeño rango a su alrededor, procurando reducir así la desorientación de este.

Figura 4.8: Oclusión de los rayos solares por un ballena

4.2.3. Cáusticas

Las cáusticas, es un fenómeno que usualmente se puede observar en la naturaleza y vida cotidiana. Surge cuando los rayos de la luz se reflejan o refractan desde una superficie curva centrándose únicamente en ciertas áreas de la superficie receptora. Normalmente, en el campo de la computación de gráficos los métodos desarrollados para replicar las cáusticas se centran en el aspecto estético, como por ejemplo [17], ya que desarrollar algoritmos que sean físicamente precisos sería muy costoso, porque habría que tener en cuenta los millones de fotones que están involucrados en la creación de estas.

Se ha basado la creación del material para las cáusticas en el siguiente vídeo [7]. La idea principal, es utilizar una textura que se desplace en el plano XY, a la que se le aplica una distorsión sumándole una textura con las normales del agua. Estas normales que también se desplazan le dan el realismo necesario para que no parezca un simple textura en movimiento, ya que le añade el movimiento de las olas como distorsión a la textura.

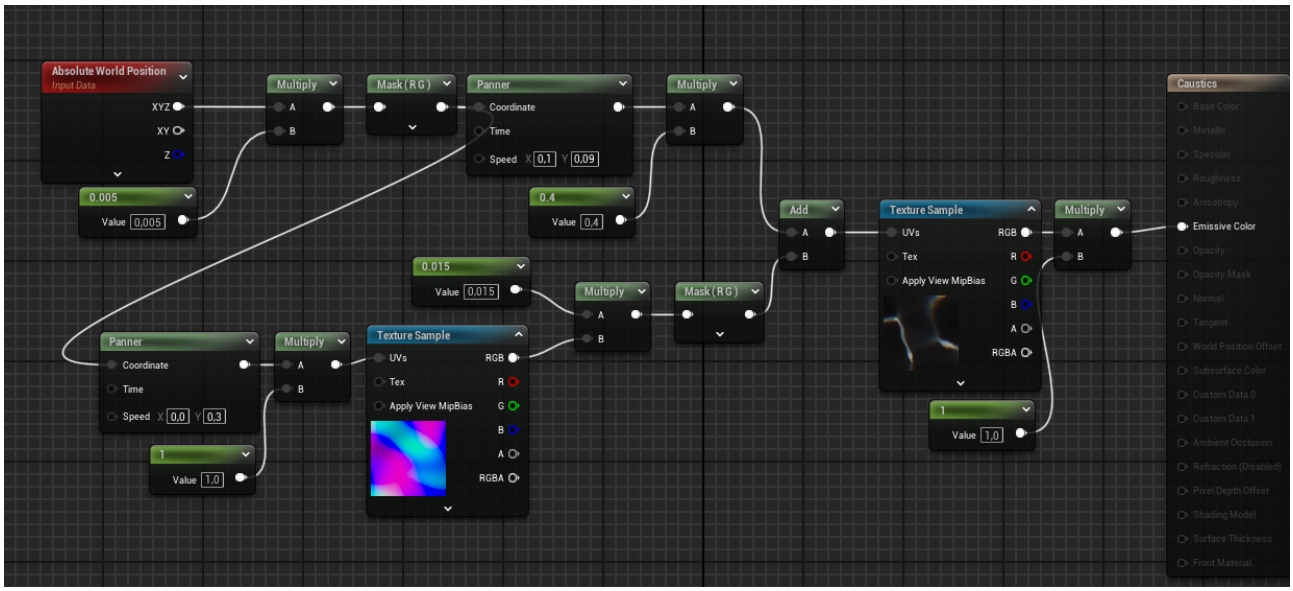


Figura 4.9: Grafo para la creación de las cáusticas

Como demuestra la figura 4.9 se obtiene la posición del mundo y se realiza un desplazamiento de las coordenadas en los ejes XY. Para añadirle la distorsión, se le suma el valor resultante del desplazamiento de la textura usada para las normales del océano. Este material se configura como una función de luz [34], de esta manera se puede aplicar a la luz direccional 4.11.

Esto tiene ciertas ventajas e inconvenientes, el principal problema es que al estar aplicado a la luz direccional, estas cáusticas se aplican a todo el nivel, tanto por fuera del agua como a gran profundidad, donde la luz ya no podría llegar. Para resolver esto se crea una máscara inferior y superior para que la función no se aplique por encima del océano ni por debajo de los 500m aproximadamente.

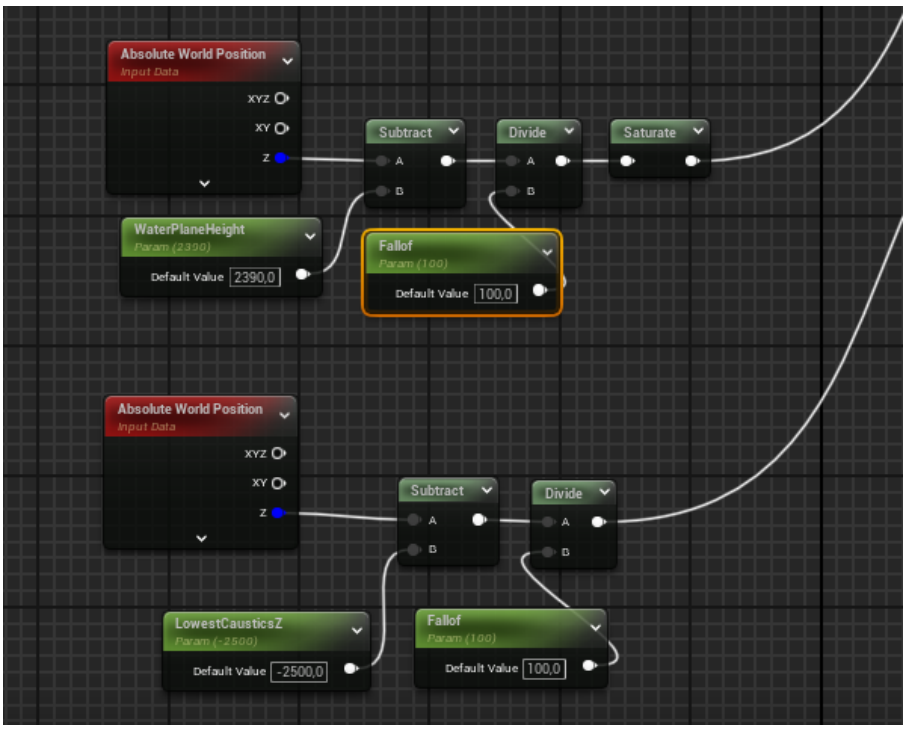


Figura 4.10: Máscaras inferior y superior para las cáusticas

Como se aprecia en la figura 4.10 para la máscara superior se resta la altura máxima, se divide para reducir en un factor la altura y por último se normaliza. Para el límite superior además es necesario obtener el valor complementario, de lo contrario se aplicaría como un límite inferior.

Figura 4.11: Resultado de las cáusticas

Inicialmente, se trató de crear las cáusticas como un *decal actor*, pero mediante esta aproximación, no se puede obtener información de la luz en el nivel, creando cáusticas en zonas sombrías, es por eso que esta opción fue descarta en favor de aplicarla como una función de luz.

4.2.4. Transición entre buceo y natación

Como ya se ha comentado anteriormente, es necesario que cuando el jugador comience a bucear, el color de la escena cambie para mostrar una iluminación adecuada. Lo mismo ocurriría cuando el jugador pase de bucear a nadar, necesitamos que el color de la escena vuelva a ser el anterior para que parezca que realmente ha salido del agua.

Adaptando el vídeo [19], se está utilizando un material de post procesado que realiza una interpolación lineal entre el color de la escena y un color azul en función de la posición de la cámara con respecto al plano del océano.

Un problema que no se ha podido solucionar, es que el cálculo de la altura del océano parece no ser correcta y esto provoca que si el jugador se sitúa en torno a la línea del océano se aprecien artefactos 4.12.

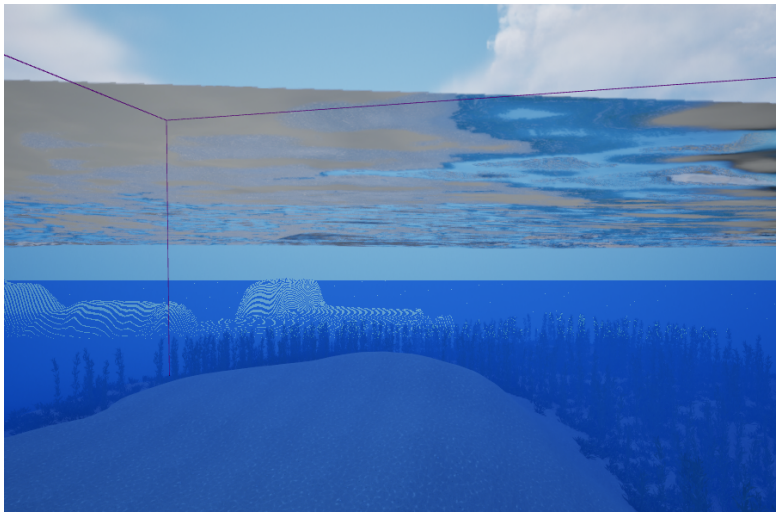


Figura 4.12: Artefactos cuando el jugador está en la línea del océano

4.3. Océano

El océano es otro aspecto fundamental y por ello se ha creado un material que simula el océano abierto. Se ha utilizado como referencia el documento de Nvidia [14] acerca de métodos para la generación de agua. Como comentan en él, uno de los primeros enfoques podría ser utilizar ondas sinusoidales para representar el agua, pero estas tienen un defecto ya que son demasiado redondeadas y aunque podría estar bien para masas de agua tranquilas con poco movimiento, como un estanque, no es lo más realista para representar el océano.

Las ondas de Gerstner, por el contrario, permiten controlar la pendiente de la ondas, generando resultados más realistas. Estas se rigen por la siguiente función:

$$P(x,y,t) = \begin{bmatrix} x + \sum(Q_i A_i \times D_i.x \times cos(w_i D_i \cdot (x,y) + \varphi_i t)) \\ x + \sum(Q_i A_i \times D_i.y \times cos(w_i D_i \cdot (x,y) + \varphi_i t)) \\ \sum(A_i sin(w_i D_i \cdot (x,y) + \varphi_i t)) \end{bmatrix}$$

Donde Q_i controla la pendiente de las olas, un valor de 0 funcionaría como una onda sinusoidal y un valor de $Q_i = \frac{1}{w_i A_i}$ resultaría en una cresta afilada. Sin embargo, hay que

tener cuidado con valores muy altos de Q_i ya que producen unos bucles en lo alto de las ondas. La amplitud A_i se define como la distancia desde el plano de agua hasta la cresta. La constante de fase φ_i representa la velocidad de desplazamiento de las ondas multiplicada por la frecuencia, $\varphi = Speed \times w$ y por último $w = \frac{2}{L}$, siendo L la longitud de onda que se define como la distancia de cresta a cresta entre las ondas.

El siguiente paso, sería el cálculo del mapa de normales. La normal $N(x, y)$ viene dada por el producto vectorial de la binormal y la tangente. Estos vectores los podemos obtener de las derivadas parciales en las direcciones x e y respectivamente. Resultando en la siguiente ecuación:

$$N = \begin{bmatrix} -\sum(D_i.x \times w_i \times A \times cos(w_i \times D_i \cdot P + \varphi_i t)) \\ -\sum(D_i.y \times w_i \times A \times cos(w_i \times D_i \cdot P + \varphi_i t)) \\ 1 - \sum(Q_i \times w_i \times A \times sin(w_i \times D_i \cdot P + \varphi_i t)) \end{bmatrix}$$

Cono paso final, se combinan 8 ondas de Gerstner cada una con sus parámetros, como la dirección de movimiento, la longitud de onda... Expuestos en una instancia de material para poder realizar cambios rápidos a las ondas por separado. El control de los parámetros para generar un océano realista es un proceso detenido y cuidadoso ya que pequeños cambios pueden generar malos resultados por como se relacionan los parámetros.

Otro aspecto a tener en cuenta, son los valores máximos y mínimos que podemos utilizar para los parámetros. El plano que se está utilizando como malla estática ha sido creado en Blender [4] dividiendo un plano de 128cm de lado 7 veces, es decir formado por cuadrados de 1cm de lado. Teniendo estas dimensiones y según el teorema de Nyquist la frecuencia debe ser como mínimo mayor al doble del tamaño de las subdivisiones para que pueda ser representada. Aunque los valores que dan mejor resultado son aquellos que se encuentran en torno a 128.

Como se aprecia en la figura 4.13, el material por si solo no es suficiente para que de una sensación realista. Para mejorar el aspecto del océano se le añade un mapa de normales que aportan detalles al material como se aprecia en la figura 4.6

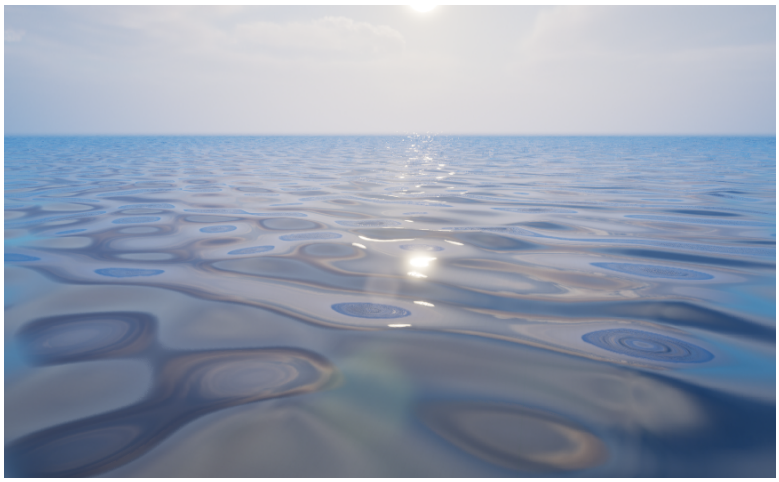


Figura 4.13: Material resultante del océano

Adaptando el vídeo [10] se crea una función de material que utiliza una textura 4.14 para formar el mapa de normales, el cuál es el resultado de la combinación 3 muestras de la textura con desplazamientos en distintas direcciones y escalas para generar un efecto de caos, dando como resultado el aspecto de la superficie marina que se muestra en 4.6.

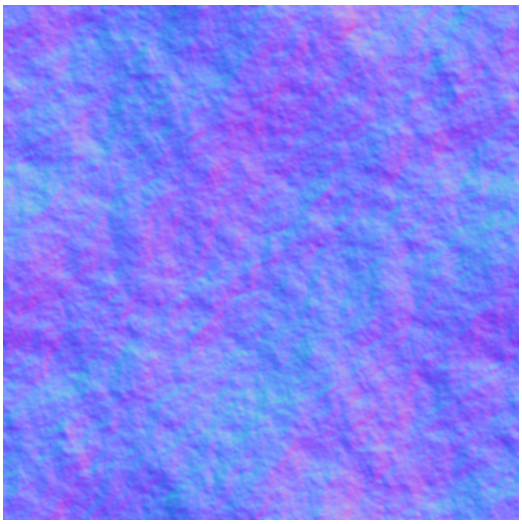


Figura 4.14: Mapa de normales de ondas en el agua

Por último en cuanto al océano, se puede añadir una máscara de profundidad, de tal manera que se controla la opacidad del material en función de la profundidad del fondo. Restando los nodos *Scene Depth y Pixel Depth* se puede obtener la distancia desde el plano de agua hasta el fondo 4.15. El valor resultante, lo dividimos por la profundidad máxima a partir de la cual el material será opaco 4.16.

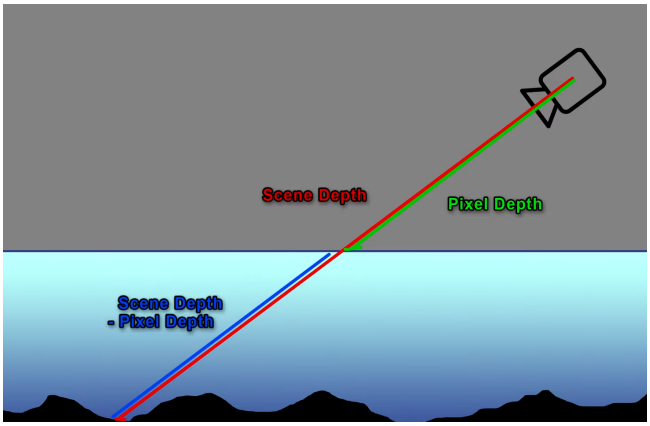


Figura 4.15: Cálculo de la profundidad usando Scene Depth y Pixel Depth [8]

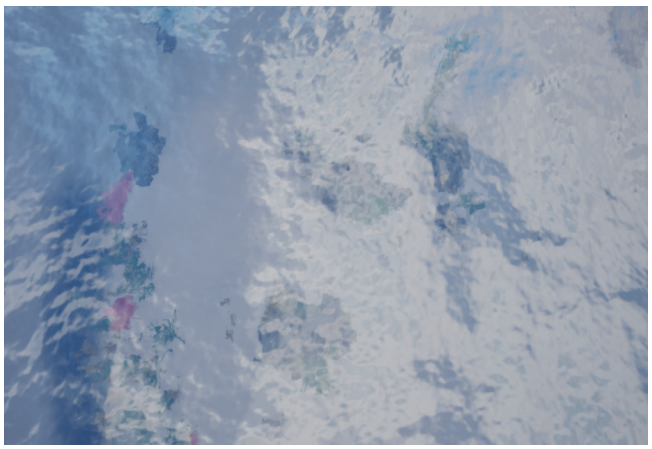


Figura 4.16: Opacidad en zona de baja profundidad

El mismo cálculo de profundidad también se puede utilizar para generar un efecto de espuma entorno a los objetos que estén en la superficie. Se ha basado la creación en el vídeo [9], para ello se utiliza una textura de espuma que emplea los distintos canales para especificar espumas de distintas densidades 4.17.

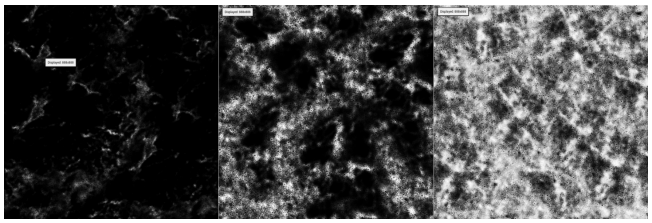


Figura 4.17: Textura de la espuma en sus distintos canales, de mayor densidad a menor

Junto con la textura se utiliza un gradiente que define en sus tres canales la opacidad que tendrá la textura de espuma, la espuma de alta densidad sería visible cerca del objeto, la de media densidad a una distancia cerca y media y la de baja densidad prácticamente en todas las zonas. Se utiliza la información de profundidad para establecer a partir de qué valor se activará la espuma y esta usa la función *Motion 4WayChaos* que genera un movimiento caótico en 4 direcciones con desplazamiento. El resultado se multiplica por el gradiente para obtener la opacidad deseada y por último se suman los canales. Este valor se añade al material del océano multiplicándose por el color base 4.18.

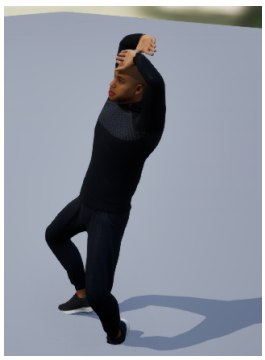
Figura 4.18: Espuma alrededor de la roca

4.4. Control del Jugador

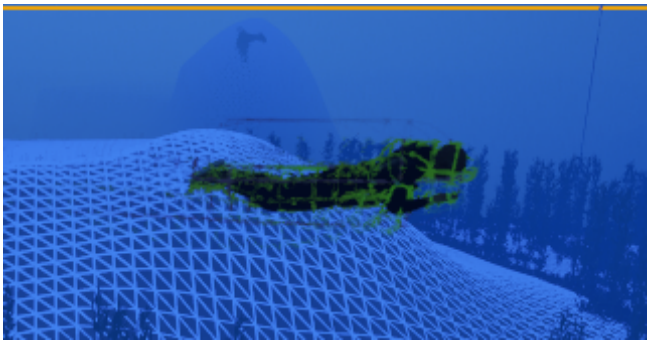
Una vez creado el entorno en el que se va a desarrollar el videojuego, el siguiente paso ha sido definir un controlador de personaje que permita bucear y nadar en la superficie.

Para ello, se han creado dos *input mapping context* que se cargan y se descargan en función de la posición del jugador. Cuando el personaje está en la superficie del océano se carga el contexto que le permite nadar, y en el caso de que esté sumergido, se carga el que le permite bucear. La acción de bucear, le permite un movimiento en todas las direcciones y la acción de nadar permite el mismo tipo de movimiento menos en el eje Z ascendente, de esta manera el jugador no pueda nadar fuera del agua. Además de dichas acciones, se establecen otras como mirar a su alrededor, que permite rotar la cámara entorno al personaje y la acción de atacar, que utiliza un sistema de habilidades para gestionar la lógica del ataque, que se comentará más adelante.

Inicialmente, surgieron problemas con las colisiones del personaje, pues el movimiento de este no rotaba la cápsula de colisiones al bucear o nadar, por lo que esta se mantenía en vertical. Inicialmente para arreglar este problema, se estableció un nuevo modo de movimiento extendiendo la clase *Character Movement Component* de tal manera que al desplazarse el personaje, se rotara el componente raíz para acompañar la dirección de movimiento. Además, hubo que reimportar las animaciones para compensar esta nueva rotación que se está aplicando. Como se aprecia en la siguiente figura 4.19a, la animación de nadar previamente en horizontal es rotada para que esté en vertical y así compensar la rotación que se aplica al moverse resultando en 4.19b.



(a) Compensación de la rotación en las animaciones



(b) Cápsula de colisiones correctamente ajustada a la posición y rotación del personaje

Figura 4.19: Rotación correcta del personaje

Este nuevo modo movimiento sobrescribía la funcionalidad del método *PhysicsRotation*, el cual se encarga de rotar al personaje hacia el movimiento cuando se tiene activado la opción de *Orient Rotation To Movement o Use Controller Desired Rotation*. De tal manera, que al cabeceo del personaje se le restan 90º para que la cápsula siempre estuviera orientada en horizontal. Pero esto no dio resultado ya que el personaje en determinados momentos, como estando estático rotaba en varios ejes, teniendo un comportamiento extraño.

Por ello, finalmente se decidió añadir una cápsula al personaje rotada 90º y utilizar dicha cápsula para las colisiones. La cápsula raíz que sería la que se usa en el movimiento se le reduce el tamaño, para que no cree problemas de colisión. Para evitar que esta nueva cápsula se pueda superponer con otros elementos que puedan a su vez lanzar eventos como es el caso del océano, se modifican sus colisiones para que ignore los objetos estáticos y dinámicos.

Para controlar el modo de movimiento del jugador se utiliza dos delegados en el océano, tal que si el jugador entra en contacto con este se establece el modo de natación y cuando deja de estar en contacto y su posición es superior al océano se activa el modo caminar.

Una vez creado el control del jugador se crean un conjunto de clases para representar a los distintos personajes que estarán presentes en el nivel.

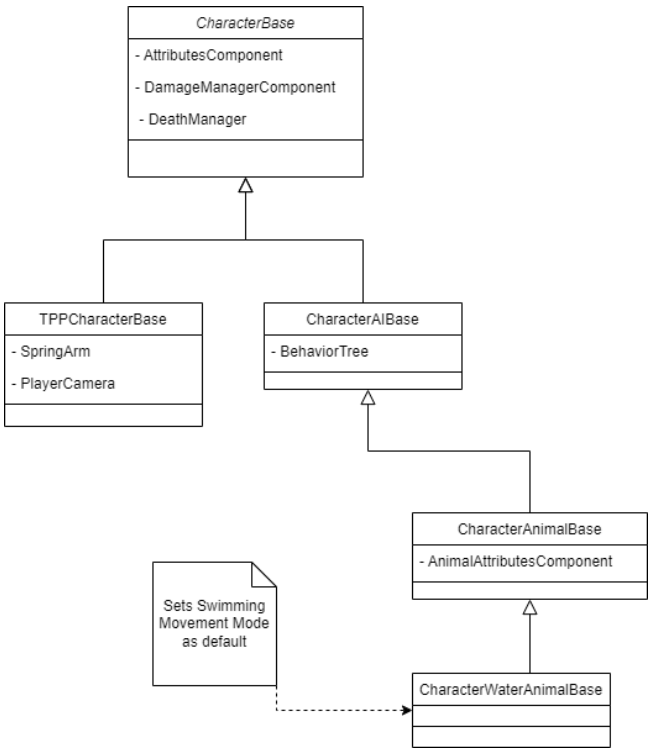


Figura 4.20: Jerarquía de las clases *Character*

Como se aprecia en la figura 4.20, se crea una clase *CharacterBase* la cual se encarga de gestionar aspectos presentes en todos los personajes. Es por ello, que crea múltiples componentes que serán comunes, como los atributos base donde se define la vida como atributo esencial que todo personaje tendrá, un sistema de habilidad mediante el cual se le podrá añadir comportamientos o estados, un componente encargado de manejar el daño para poder recibir ataques de otros enemigos o del entorno. Además, hereda de la clase *ACharacter* el componente de movimiento, este componente provee múltiples modos movimiento para personajes humanoides, como caminar, nadar, volar o caer, aunque también puede ser adaptado para otros tipos de personajes, como se ha hecho con los animales en este trabajo. Por último, declara una fuente de estímulos que podrá ser manejada por controladores de IA para establecer reacciones. Por ejemplo, dicha fuente se utiliza en el jugador para que los animales hostiles puedan detectarlo y perseguirlo.

La clase *TPPCharacterBase* representa la clase base utilizada para los personajes jugables, por ello contiene una cámara y un brazo de resorte. Esta clase además contiene un inventario para el jugador donde poder almacenar los objetos que se recojan en la escena. Por último, se encarga de mostrar al inicio información al jugador para que conozca el objetivo, así como de gestionar el final de la partida. Anteriormente, se ha comentado que el jugador debe recoger una bengala y activarla en la superficie para ser rescatado, por lo que en esta clase se comprueba si el jugador en algún momento posee en el inventario la bengala y si además está en la superficie, en dicho caso se le muestra a través de un *widget* que tiene que encender la bengala. Cuando lo haga, se lanza un temporizador que tras pasados unos segundos lleva al jugador al menú principal, acabando la partida.

De *CharacterBase* también hereda *CharacterAIBase* la clase base para todos los que recibirán algún control de IA, como los animales marinos. Para ello, añade un árbol de comportamientos donde se podrá definir tareas a ejecutar por la inteligencia artificial.

La clase base de los animales sobrescribe el componente de atributos. Inicialmente, se añadió el ataque base a los atributos de los animales para ser usado como el daño base del animal, pero al añadir el sistema de habilidades y las habilidades de ataque este ha perdido su propósito al ser redundante.

Por último, la clase de animales marinos establece el componente de movimiento en modo natación, para ello, activa la propiedad *bWaterVolume* del *PhysicsVolume* y cambia el modo de movimiento a natación, para permitir que estos animales naden por la escena.

4.5. Animaciones

Es necesario para un funcionamiento realista de los personajes controlar las animaciones. Para ello, se ha creado una clase que hereda de *UAnimInstance*, en esta clase se han implementado diversas funciones que permiten conocer si el jugador está en movimiento, si está vivo, la velocidad a la que se mueve o la dirección que lleva. La clase usa *NativeUpdateAnimation*, un método que es llamado en cada actualización para establecer información referente al jugador que luego servirá como variables en el *blueprint* de animación con las que decidir si cambiar a un estado u otro y en definitiva reproducir una animación u otra.

Además, para los ataques se han creado montajes, animaciones que pueden ser ejecutadas dinámicamente y que pueden ser combinadas con otras animaciones que esté reproduciendo el *blueprint* de animación. Por ejemplo, si el jugador está nadando y ejecuta la acción de golpear, la animación de nadar se mezcla con el montaje a partir de un hueso que hayamos seleccionado y se reproducirá dicha mezcla. En el caso del jugador se utiliza el hueso del hombro para mezclar una animación de puñetazo.

Esta mezcla, también se puede realizar basándose en otros parámetros. Cuando algún personaje muere, la variable *bIsAlive* se establece en falso y esto modifica el funcionamiento de un nodo que mezcla animaciones en base a un valor booleano, reproduciendo la animación de muerte.

Para las animaciones de locomoción de los personajes, se han creado espacios de combinación (*blendspaces*) [3]. Estos permiten mezclar animaciones en función de dos entradas, normalmente para la locomoción se utiliza la dirección en el eje horizontal y la velocidad en el eje vertical, de esta manera en función del movimiento del personaje se mezclarán las animaciones de nadar hacia delante, nadar hacia la izquierda, derecha o hacia atrás.

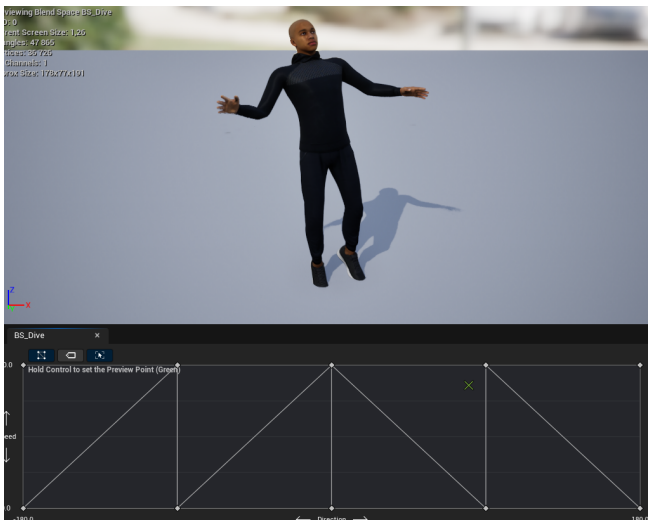


Figura 4.21: Editor de animaciones para el espacio de combinación

En la figura 4.21 se puede observar el gráfico con las distintas animaciones (representadas como puntos en el gráfico). Utilizando la tecla control y moviendo el ratón se puede previsualizar la combinación de las animaciones, por ejemplo en este caso se está mostrando la animación de nadar hacia la derecha ligeramente combinada con la de nadar hacia el frente.

4.5.1. Animaciones del jugador

El personaje cuenta con dos estados posibles, en la superficie del agua o buceando en ella, esto también se refleja en el *blueprint* de animaciones del jugador 4.22.

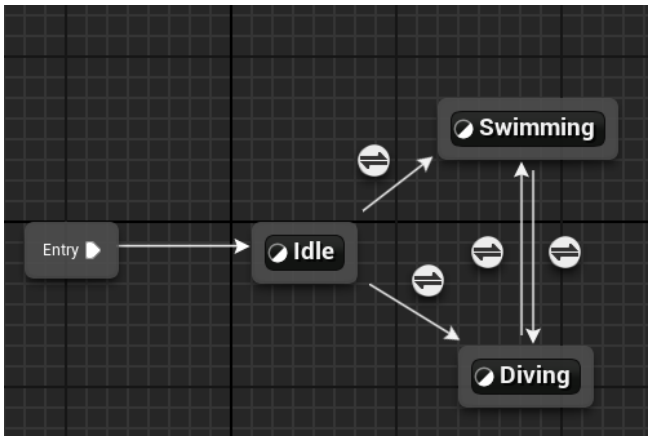


Figura 4.22: Locomoción del personaje jugable

En modo natación se ejecuta la siguiente máquina de estados 4.23. En ella, tenemos un estado inicial que ejecuta la animación de reposo. Además, si ha transcurrido un largo tiempo en este estado, se activa un booleano que provoca un cambio de animación a otra de reposo. El estado *SwimStart* se reproduce cuando el jugador pasa a estar en movimiento y ejecuta una animación que suaviza la transición de estático a en movimiento, una vez esta termina se ejecuta el estado *Swimming*, un espacio de combinación que en función de la dirección y la velocidad mezcla las animaciones de nado en la superficie. Por último, al dejar de moverse podría ejecutar otra animación que de la misma manera que

StartSwimming suavizara la parada del personaje, pero no se encontraron animaciones para ello. La locomoción para el buceo es similar, pero solo con los estados *Idle* y *Swimming*.

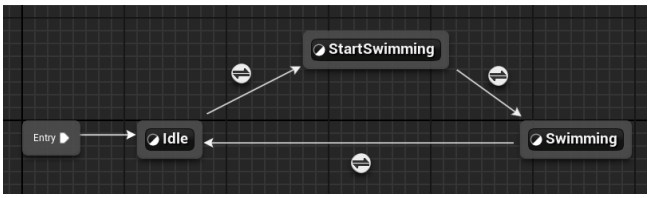


Figura 4.23: Locomoción para la natación

El siguiente paso en el *blueprint* es mezclar los montajes con la animación de locomoción. Como se aprecia en la figura 4.24, se obtiene la pose almacenada de la locomoción y se mezcla con los montajes, estos montajes están utilizando un *slot*, tal que todos aquellos marcados con el mismo se mezclaran este punto. El nodo *Layered blend per bone* permite indicar a partir de qué hueso queremos hacer la mezcla y cual será la profundidad de esta. Por ejemplo, con una profundidad 2 a partir del hombro se tomará el brazo y el antebrazo. En este caso, para que la animación de ataque se visualice correctamente se ha tomado el último hueso de la espina dorsal, ya que si se utilizaba el hombro derecho la dirección y forma del puñetazo era humanamente imposible.

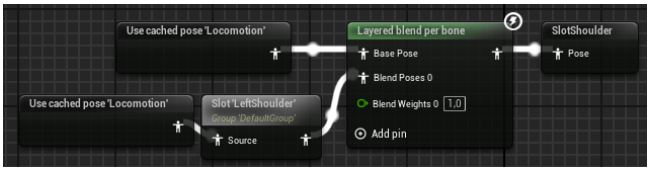


Figura 4.24: Mezcla de montajes con la locomoción

Por último, se añade el control de las animaciones de muerte. En este videojuego, el personaje puede morir por ataques de enemigos o por asfixia, es por eso que se ha creado un *Enum* que define estas dos posibilidades. El manejador de daño se encarga luego de detectar si se ha producido una asfixia o una muerte por ataque y establece el tipo de muerte que ha tenido el jugador, esta información se usa en las animaciones para reproducir los distintos tipos de muerte. Haciendo una mezcla basada en enteros se selecciona la animación de muerte y si la variable *bIsAlive* es falsa se reproduce, en caso contrario se reproduce la animación almacenada en el proceso anterior. La animación de asfixia está formada por varias animaciones, por ello se utiliza una máquina de estados para reproducirlas secuencialmente.

4.5.2. Animaciones de animales marinos

En el caso de los animales, se sigue una metodología similar, aquellos que no son hostiles, como la ballena tienen un comportamiento más sencillo, estableciendo una locomoción de dos estados, reposo y movimiento. Para el tiburón, se utiliza también el nodo *Layered blend per bone* para mezclar el montaje de ataque con su locomoción. Además, se controla la muerte de este de la misma forma que el jugador.

4.6. Sistema de Habilidades

Se ha utilizado un sistema de habilidades desarrollado en la asignatura de “Desarrollo de Videojuegos 3D” impartida en estos estudios de Máster, para establecer estados o acciones posibles a los personajes, por ejemplo para los ataques, sistema de oxígeno o el ataque del tiburón.

De forma análoga al sistema de habilidades de Unreal [16], este sistema permite definir habilidades las cuales ejecutarán su funcionalidad cuando se activen. Además, estas pueden interactuar con otras habilidades cancelándolas, requiriendo que otras estén activas o no pudiendo activarse si alguna de las habilidades establecidas como bloqueantes están en funcionamiento.

Esto se ha extendido para añadir otros tipos de habilidades, como habilidades con tiempo de enfriamiento, con retardo o en bucle. Se crea una clase para añadir el enfriamiento y el retardo mediante el uso de temporizadores. Para el retardo se establece un temporizador, tras el cual se llama a un método que deberá ser sobrescrito por las clases que hereden, donde se ejecuta la funcionalidad de la habilidad. Para implementar el tiempo de enfriamiento se sigue un procedimiento similar. Para controlar que la habilidad se pueda activar o no, en tiempo de enfriamiento se establece una etiqueta que prohíbe la activación de la misma, al terminarse el temporizador, esta etiqueta es eliminada pudiendo activarse la habilidad nuevamente.

Para las habilidades como el decremento de oxígeno, se ha implementado la posibilidad de reproducir en bucle una habilidad. Para ello, se hereda de la habilidad que añade retraso y tiempo de enfriamiento, y se sobrescribe la función que maneja el fin del temporizador del enfriamiento. En este caso, además de eliminar la etiqueta, también se vuelve a intentar activar la función, así indefinidamente o un determinado número de bucles. Definir un número de bucles para esta función podría ser útil para habilidades como la comida, por ejemplo, aumentando la salud del jugador paulatinamente. Las habilidades en bucle pueden ser desactivadas en cualquier momento, tanto por otras habilidades que las cancelen o de manera manual. Para cancelarlas, se sobrescribe el método *OnCancelAbility* utilizando la función *ClearTimer* para eliminar el manejador del temporizador y cancelando con ello la habilidad.

Las habilidades de decremento e incremento de oxígeno están creadas heredando de la habilidad de bucle. En ellas, en cada iteración se accede a los atributos del personaje y se incrementa o decrementa el oxígeno. Además, al decrementar se comprueba que este sea mayor que 0, en caso contrario se le aplica daño al jugador.

Para implementar los ataques del jugador o de los animales se hereda de la habilidad que implementa el retraso y el tiempo de enfriamiento. Esta clase añade métodos para hacer trazas esféricas o lineales que podrán utilizar luego los ataques para ejecutar el daño. Además, se sobrescribe el método *OnActivateAbility* llamado en la activación de la habilidad para que se ejecute un montaje de animación. De esta clase heredan las clases *BiteAttack*, que implementa el ataque del tiburón y *PunchAttack*, que implementa el puñetazo del jugador. Para el puñetazo, se define un *socket* de la malla que será utilizado como el origen de una traza esférica para detectar si algún actor se superpone con esta. En dicho caso, se le aplica daño al actor golpeado. El retraso en las habilidades de ataques

es muy importante, ya que nos permite sincronizar la animación con la realización de la traza.

Estás habilidades son añadidas a *CharacterBase* a través de un componente, este componente almacena un vector de habilidades, que representa las habilidades definidas para el personaje, y un contenedor de etiquetas para marcar las habilidades que están activas. Además, se añaden métodos que permiten añadir, eliminar, activar o cancelar habilidades.

4.7. Inteligencia Artificial

En el nivel, se encuentran varios animales como el tiburón o la ballena. Para definirles un comportamiento, se han utilizado árboles de comportamiento [1], estableciendo rutinas para deambular, perseguir y atacar.

Mediante estos, se pueden definir tareas que serán ejecutadas por los personajes controlados por IA. Estos árboles se combinan con la pizarra, componente que sirve para definir y establecer variables, que son realmente útiles para la toma de decisiones. Además, existen nodos de utilidad como el selector o la secuencia, el selector ejecuta de manera secuencial las tareas y para en el momento que una de ellas tiene éxito y la secuencia ejecuta de manera secuencial, pero para cuando alguna falla.

Para controlar los árboles de decisiones, también se puede hacer uso de decoradores y de servicios. Los decoradores se pueden acoplar a cualquier nodo y establecen si el nodo puede ser ejecutado en base a condiciones. Los servicios pueden acoplarse tanto a nodos, como al selector o la secuencia y funcionan de manera similar al método *Tick* ejecutándose repetidamente a cierta frecuencia, mientras que la rama en la que se encuentra esté activa.

Inicialmente, se ha intentado utilizar la funcionalidad propia de Unreal para el movimiento de los personajes, a través de un *Navigation Mesh* y de funciones como *GetRandomReachablePointInRadius* y *MoveTo*. Estas, están pensadas para el movimiento en superficies y en dos dimensiones, es por ello que no sirven para tipos de movimiento como nadar o volar.

Por este motivo, para implementar la natación de los animales se ha optado por usar el *plugin Customizable Pathfinding* [25], el cual permite encontrar caminos tanto en 3D como en 2D.

Los animales aparecen en el nivel gracias a la clase *Spawner*, en ella se pueden definir los animales que aparecerán y la cantidad de estos. Implementa un método virtual *Spawn* que permite definir la manera en la que los animales son añadidos a la escena, en este caso se comienzan a crear centrados en el actor y a los subsiguientes se les añade un desplazamiento, el cual editable en el editor. Esta clase almacena una referencia al *CPathVolume*, utilizado luego por los personajes para calcular los caminos. Para que estos puedan acceder al volumen en las tareas, en la creación se establece el *Spawner* como el objeto padre.

Los animales, tanto hostiles como neutrales deambularán por la escena, para ello se define la siguiente rama del árbol 4.25 y el conjunto de tareas.

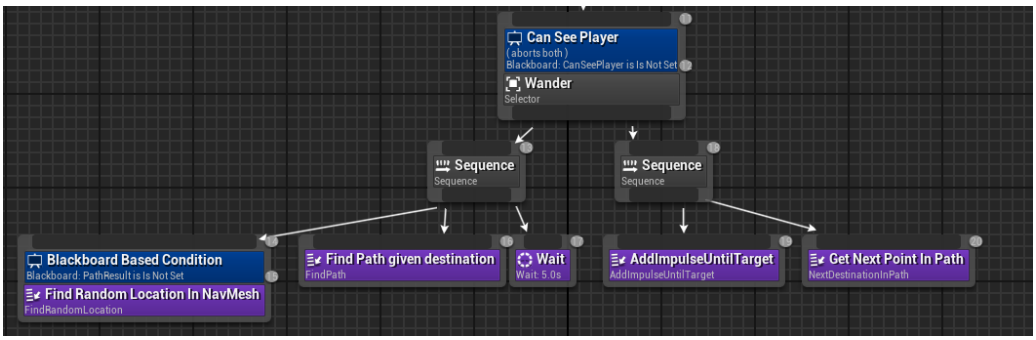


Figura 4.25: Comportamiento de deambular

La tarea *Find Random Location In NavMesh* devuelve un punto aleatorio dentro del perímetro del *CPathVolume*, actor que contiene el grafo utilizado para encontrar caminos. Esta búsqueda de un punto aleatorio no se ejecutará en el caso de que *PathResult*, variable que almacena el camino calculado tenga valor. *Find Path Given Destination* busca un camino de manera asíncrona. Para la búsqueda, es necesario indicar el inicio, que corresponde con la localización del agente y el destino, que se ha calculado en la tarea anterior y esta lo ha escrito en una variable en la pizarra. Además, se puede indicar a través de la variable *SmoothingPasses* cuan suavizado se quiere el camino encontrado. Para animales grandes como el tiburón, elegir un valor alto de suavizado da un mejor movimiento del animal, pues se reducen los giros que realiza. Esta búsqueda, puede fallar en caso de no encontrar ninguno, esto puede ocurrir si el animal o el destino se encuentran fuera del volumen entre otros casos. Si esto ocurre, se borra el resultado almacenado en la variable *PathResult* y se volvería a buscar otro punto aleatorio en una siguiente iteración. El motivo por el cual la búsqueda de un punto aleatorio no debería ejecutarse si *PathResult* tiene un valor, es porque en dicho caso habría que moverse hacia el destino. Como se ha comentado, el nodo Selector ejecuta las ramas hijas hasta que una tiene éxito. Por ello, cuando se establece *PathResult* la primera rama falla y se ejecuta la rama de la derecha que mueve al agente hacia el objetivo.

Esta rama comienza ejecutando *AddImpulseUntilTarget*, esta tarea accede a *PathResult*, para comenzar el movimiento hacia los nodos que conforman el camino. Es importante añadir, que dicha variable se añade a la clase *CharacterAIBase*, ya que las referencias a objetos de la pizarra son punteros *Weak* y por tanto eliminados por el recolector de basura, pudiendo generarse violaciones de acceso. Para el movimiento, se calcula la dirección de este restando al objetivo la localización del agente y esta se normaliza usando *GetSafeNormal*. El movimiento se aplica usando *AddMovementInput*. Además, si está lo suficientemente cerca del objetivo devuelve un éxito y termina la tarea, en caso contrario falla para seguir impulsando hacia el objetivo. Cuando la tarea anterior tiene éxito, *Get Next Point In Path*, actualiza al siguiente nodo de destino. Si ya se encuentra en el final, limpia los valores para que *Find Random Location In NavMesh* encuentre un nuevo punto.

Para que los personajes puedan detectar al jugador o a otros personajes se ha creado un controlador para la inteligencia artificial y se ha implementado un sistema de percepción [32]. Además, en la clase *CharacterBase* se configura una fuente de estímulo. Esta fuente de estímulo se puede registrar en el sistema de percepción, permitiendo que otros puedan detectarlos. El controlador de IA suscribe un método, *OnTargetDetected* al delegado *OnTargetPerceptionUpdated*, de tal manera que cuando este componente percibe un estímulo es llamado. *OnTargetDetected* establece una variable *CanSeePlayer*

de la pizarra, la cual permite comenzar a perseguir o atacar al objetivo. Además, en estas dos ramas se implementa un servicio para conocer cuando se está en rango de ataque y solo si las dos condiciones son verdaderas se ataca.

En el caso de que el personaje detecte a su objetivo, pero no esté lo suficientemente cerca como para realizar un ataque, se ejecuta una rama para perseguirlo 4.26. De manera análoga al comportamiento de deambular, este comportamiento establece como destino la ubicación del personaje y se mueve hacia esta hasta estar a menos de un límite.

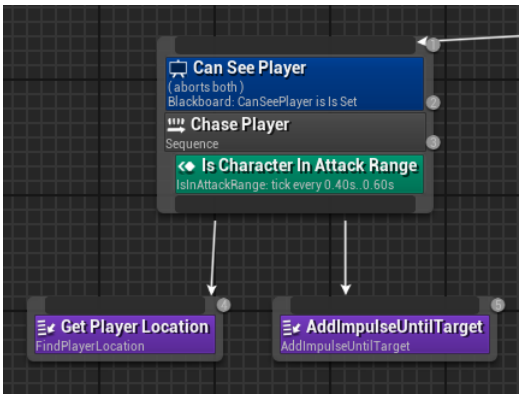


Figura 4.26: Persecución

Cuando se entra dentro del rango de ataque, se ejecuta la tarea BiteAttack 4.27, la cual hace uso del sistema de habilidades, pues accede al componente e intenta activar la habilidad de mordisco.

Figura 4.27: Comportamiento de atacar

4.8. Objetos

Para que el jugador pueda interactuar con la escena, se han añadido clases para representar a los objetos dentro del videojuego y un inventario básico, para conocer qué objetos ha recogido el jugador.

La clase base *Item*, define los componentes y funcionalidades básicas y comunes para todos los objetos. Por tanto, añade un componente de escena y una malla, que serán usados para representar gráficamente al objeto. Además, añade una variable *ItemName* que servirá como identificador.

Para tener objetos que puedan ser recolectados por el jugador, se hereda de la clase *Item* para crear *Collectable*, la cual añade un *SphereCollision*, que será usado para detectar cuando el jugador está dentro del rango de recolección. Se crean por tanto, dos métodos suscritos a los delegados *OnComponentBeginOverlap* y *OnComponentEndOverlap* de la esfera. Esta clase permite además definir un *Widget* que será mostrado al jugador cuando esté cerca del objeto 4.28, de tal manera que le muestre información útil como qué tecla debe de pulsar para recolectarlo. Para ello, se crea el *Widget* en el *BeginPlay* y se añade a la ventana. Al sobreponerse el jugador con la esfera, se usa el método *SetVisibility* para mostrar el *Widget* y al contrario al dejar de estar en contacto.

Figura 4.28: Se muestra el Widget al estar cerca del objeto

Por último, se comprueba continuamente si el jugador está en rango y además ha pulsado la tecla de recolección, en ese caso se llama al método *Collecting*, que introduce el objeto en el inventario del jugador, dicho inventario se define en la clase *TPPCharacterBase*, por lo que solo estaría disponible para el jugador. Además, añade la malla del objeto a la mano del jugador si la variable *bAttachToActor* es verdadera, utilizando el método *AttachToComponent* y el *Socket* de la mano derecha. De esta manera, se puede ver en el videojuego cuál ha sido el último objeto recolectado. Si es falsa, se destruye el objeto.

La clase *Collectable* se ha extendido, para permitir objetos que generen efectos especiales. Por ello, se le define un componente *NiagaraComponent* a la clase *FXCollectable*. Este componente se une a la malla como hijo, de esta manera podría ser luego activado por el jugador.

4.8.1. Bengala

Para el desarrollo de este videojuego, se ha añadido una bengala que tiene que recoger el jugador y utilizar en la superficie para terminar la partida. Esta bengala es una instancia de la clase *FXCollectable*, de esta manera puede emitir unas partículas cuando es utilizada.

Se ha seguido el vídeo [12] para la realización del efecto. Este sistema Niagara consta de varios nodos que conforman el encendido. Se parte de de la plantilla *Fountain* para crear unas chispas iniciales.

La intención de estas partículas iniciales es que se asemejen a la ignición de la bengala, como se muestra en [11], vídeo que ha sido utilizado para obtener referencias reales acerca del efecto, para encenderla se utiliza una piedra encendedora, de manera similar a las cerillas, lo que provoca una emisión de chispas. Para crear este efecto, se añade el módulo *Spawn Burst*, que modifica la aparición para emitir un chorro de partículas. Además, se establece un bucle de 3 iteraciones para este primer efecto, de tal manera que se emitirá inicialmente tres chorros de partículas.

Para que las partículas generadas no se dispersen demasiado, se le baja el tiempo de vida máximo a 0,3, de esta manera tras salir disparadas desaparecen y se establece un color rojo para ellas. Para que las partículas tengan un forma alargada en lugar de redondeada, se puede cambiar el tamaño del *sprite*, de tal manera que se alarga en el eje Y 4.29.

Figura 4.29: Partículas generadas en el inicio del encendido

Una vez generadas las chispas iniciales, se generarían unas chispas que se mantienen durante todo el efecto 4.30. Para estás partículas, se utiliza el módulo de *Spawn Rate* para la aparición, este módulo crea una aparición continuada de partículas. Para que comience su ejecución tras las chispas iniciales, se establece un retraso en la ejecución del bucle de 0,4 segundos. Además, se configura la velocidad, la gravedad y el movimiento para que se adapten al efecto deseado. Para añadir una mayor variabilidad al movimiento se añade el módulo *Curl Noise Force*, que añade un ruido a la fuerza de movimiento que se aplica.

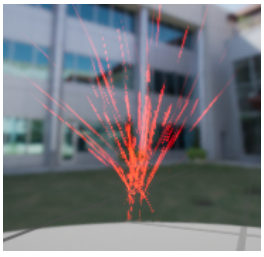


Figura 4.30: Partículas generadas durante el efecto

Para la creación del humo, se utiliza el material proveniente del contenido inicial de Unreal *SmokesubUV*, dicho material define una cuadrícula de *sprites* que representan los *frames* individuales que serán usados por las partículas 4.31. Para su uso, hay que configura el *Sprite Renderer* con el número de imágenes que posee el material, 64 en este caso. También se configura el tamaño, velocidad y tiempo de vida de las partículas

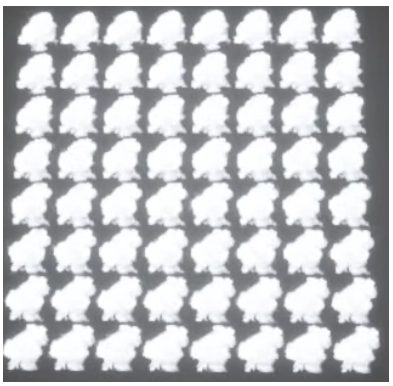


Figura 4.31: Material para replicar el movimiento del humo

Por último, se añade una pequeña llama que saldrá de la punta de la bengala. Para este efecto, las partículas deben de generarse todas desde un mismo punto, por lo que se elimina el módulo *Shape location*. Para la forma se utiliza el módulo de render *Ribbon Renderer*, que le dota de una forma en espiral. Además, para que la punta de la llama sea puntiaguda, se utiliza el módulo *Scale Ribbon Width*, que permite definir una curva de la escala en función del tiempo de vida de la partícula. El resultado final se representa en la figura 4.32

Figura 4.32: Efecto de la bengala de humo

4.8.2. Resaltar Objetos

Incluso en situaciones normales de visibilidad, puede resultar complicado encontrar objetos en la escena. Esto se ve acrecentado en niveles de baja visibilidad, como en los niveles subacuáticos. Es por ello, que para resaltar los objetos coleccionables y que el jugador pueda reconocerlos con facilidad, se ha añadido un material para destacar los objetos.

A través del nodo Fresnel, se crea un máscara para que el material se aplique exclusivamente a los bordes. Este nodo realiza un producto vectorial entre las normales del material y la dirección de la cámara, creando un efecto de desvanecimiento desde el exterior hacia el interior. 4.33

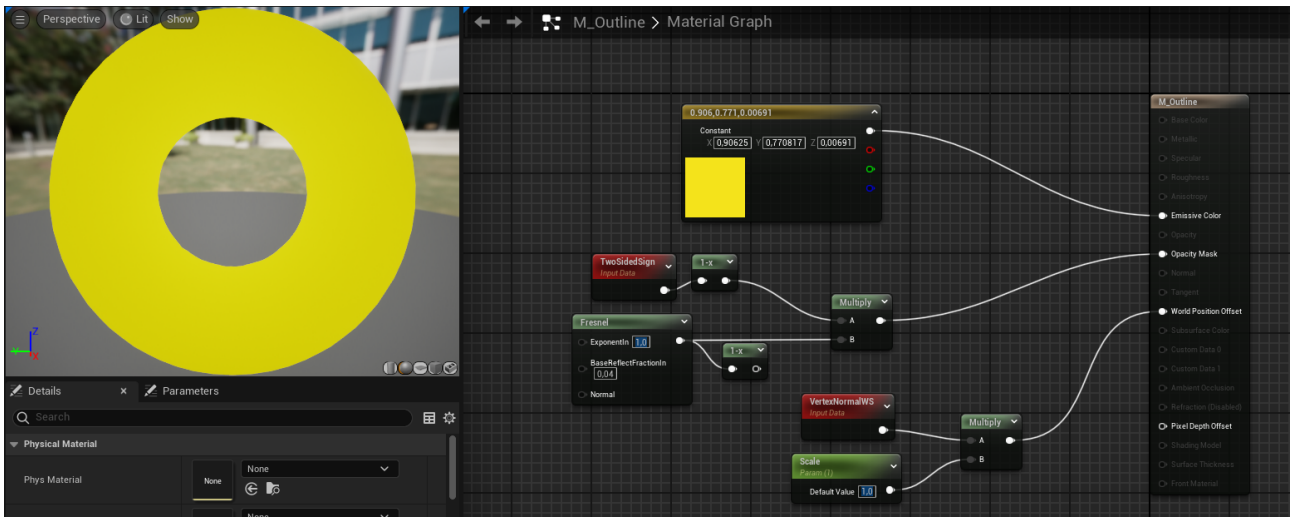


Figura 4.33: Grafo del material

Este material se está aplicando como un *overlay material* a la malla, mezclándose con esta 4.34. Un inconveniente de esta técnica es que la malla se dibujará dos veces, por lo que puede afectar al rendimiento.

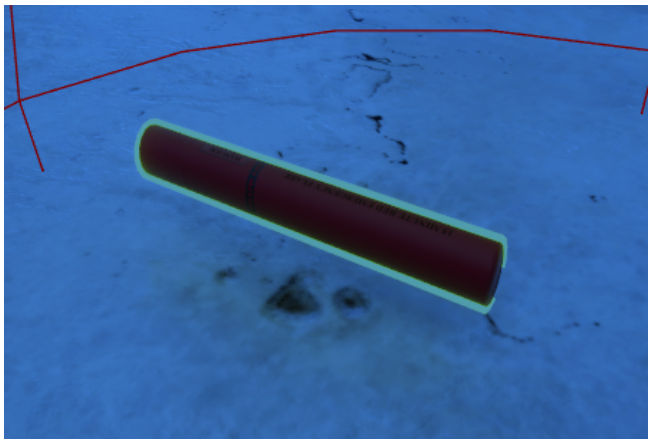


Figura 4.34: Bengala con el material para resaltar el borde

Capítulo 5

Análisis y evaluación del juego desarrollado

En este capítulo se valorará el prototipo realizado con la finalidad de comprender sus fortalezas y debilidades de cara a futuros desarrollos.

5.1. Valoraciones por parte de jugadores

En este apartado se exponen las valoraciones recibidas de otros jugadores que han podido probar la demo del videojuego. Para ello, se ha empaquetado el juego y se les ha enviado un copia.

Con el objetivo de conocer la opinión de los jugadores, se les ha enviado un cuestionario sobre el desempeño de la demo. A continuación, se exponen los resultados del cuestionario, cada pregunta tiene una puntuación del 1 al 5 y por último, se hará una evaluación del videojuego.

Cuestionario					
Cuestión	Jugador 1	Jugador 2	Jugador 3	Jugador 4	Promedio
Grado de avance en el juego	5	5	5	1	4
Grado de dificultad	1	2	2	3	2
Grado de inmersión en el contexto de un juego de supervivencia	3	4	3	4	3,5
Valoración general del juego	4	4	4	3	3,75
Valoración general de las mecánicas de la demo	4	3	5	4	4
Valorar la posibilidad de ampliar la demo a un juego completo con el estilo general presentado	5	5	4	4	4,5
Valorar comprensión del funcionamiento del juego	4	5	4	4	4,25

Valorar interfaz de usuario	3	3	3	3	3
Valorar control del personaje del jugador	4	4	5	4	4,25
Valorar el funcionamiento de los personajes controlados por IA	3	2	3	3	2,75
Valoración de gráficos, atendiendo al tipo de ambiente submarino que se pretende desarrollar	5	5	4	5	4,75

Tabla 5.1: Resultados del cuestionario completado por los jugadores. Escala de valoración del 1 (peor) al 5 (mejor).

En general, algunos jugadores han encontrado dificultades para encontrar las ruinas, han comentado que la extensión del mapa era bastante grande y esto ha provocado que un jugador no las encontrara, necesitando ayuda para proseguir con la demo.

Por otro lado, destacan el apartado visual, tanto el material del océano, como la creación de las cáusticas, niebla o los efectos especiales de la bengala, calificados como bastante realistas.

Acerca de la movilidad del personaje hay comentarios variados, algunos comentan la brusquedad de los movimientos al hacer giros, pero en general mencionan de manera positiva la transición de animaciones entre nadar en la superficie y bucear.

También se ha valorado positivamente la zona de las ruinas oceánicas, pero el combate con el tiburón ha resultado algo confuso a la mayoría, el ataque del buzo al ser hacia abajo ha resultado algo difícil a un jugador para apuntar contra el tiburón.

Por último, en cuanto a las interfaces gráficas, se ha comentado que las estadísticas del jugador resultaban algo pequeñas y ajustadas en la pantalla, provocando a un jugador una dificultad inicial para distinguir a qué estadística correspondía cada barra.

5.2. Análisis y valoración del desarrollador

Bajo mi punto de vista, la demo se encuentra en una fase de desarrollo muy temprana, en la que se han incluido las bases que conforman un videojuego de supervivencia, pero que necesita de otras mecánicas típicamente incluidas en videojuegos de este genero, como los sistemas de construcción de objetos o de edificios. En general, necesita más trabajo en cada una de sus mecánicas para añadir mayor profundidad, como es entendible, ya que un videojuego de este genero suele ser un proyecto ambicioso, que requiere bastante tiempo.

Dicho esto, si valoramos el estado actual del videojuego, considero que en cuanto al apartado gráfico se ha hecho un buen trabajo simulando los distintos efectos de la luz en entornos subacuáticos. En cuanto al movimiento y control del personaje, es cierto que el movimiento en determinadas direcciones puede resultar algo brusco, ya que no se han encontrado animaciones de transición suficientes que permitan suavizar el cambio de animaciones.

Aunque el movimiento de los animales se ha conseguido mejorar frente a sus etapas tempranas, en muchas ocasiones la tarea encargada de encontrar caminos falla, provocando que estos se queden estáticos durante largos periodos. Esto no solo le resta realismo al nivel, sino que además provoca que el jugador deje de estar inmerso en el juego.

En conclusión, mi valoración es positiva, aunque es verdad que hay varios componentes que necesitan trabajo, como la ya mencionada inteligencia artificial de los animales, considero que la demo puede ser tomada como una plantilla para crear videojuegos de supervivencia subacuáticos y que con un diseño mayor para desarrollar en él una buena trama, se podría crear un buen videojuego.

Capítulo 6

Conclusiones y líneas futuras

El propósito general del proyecto era generar un entorno acuático coherente y un videojuego de supervivencia en tercera persona, lo cual se ha cumplido, pues se ha trabajado en la estética de la escena para poder conseguir una buena iluminación que se adapte cuando el jugador esté bajo el agua, reduciendo la visibilidad con la niebla de altura exponencial, añadiendo una generación de las cáusticas y que estas estén limitadas a una altura.

Una vez logrado el primer objetivo, se ha creado un conjunto de clases para personajes humanoides o animales altamente extensible y configurable. Se ha añadido un controlador para el jugador que le permita nadar, bucear, interactuar con el entorno o defenderse. Por último, utilizando el paquete disponible en la tienda de Unreal Engine *Customizable Pathfinding* para generar trayectorias navegables en las tres dimensiones y considerando los obstáculos, se han podido creado tareas para dotar a los animales de movimiento deambulando por la escena, persiguiendo al jugador o atacándole, las cuales se podrían extender para animales voladores pues permiten cualquier tipo de movimiento 3D.

En el prototipo actual hay aspectos técnicos que deben acometerse de forma más prioritaria. Principalmente, habría que trabajar en el apartado sonoro, añadiendo sonidos ambiente, como las burbujas que genera el buzo al respirar, sonidos de los animales o añadiendo una música que acompañe la aventura dentro del videojuego. Además, habría que mejorar la transición al emerger y sumergirse, que requiere un cálculo correcto de la altura del océano para que no se generen artefactos. También en las animaciones de ataque habría que conseguir una mejor fluidez en su ejecución mientras se bucea. Finalmente, el material para el océano podría hacer uso de nodos personalizables HLSL, mejorando la parametrización y escalabilidad del material.

Como líneas futuras, sería interesante añadir más vida marina al entorno, utilizando nuevas tareas para definir distintos modos de movimiento que se adapten a cada uno de los animales, incluso, añadir nuevos ataques de los animales como veneno o la electricidad de las medusas. En cuanto al personaje se podrían desarrollar nuevos y mejores métodos de movimiento como vehículos o propulsores que faciliten la exploración, así como armas a distancia para defenderse de forma más cómoda.

Como se ha comentado, Unreal demuestra ser un motor completamente capaz para la recreación de ambientes subacuáticos, pudiendo adaptarse la iluminación o la absorción de luz bajo el agua, así como la generación de caminos en tres dimensiones para simular el nado de los animales y la adaptación de un controlador del jugador para el movimiento de este en el nivel.

Capítulo 7

Conclusions and future work

The main purpose of this project was to recreate a realistic underwater environment and also develop a third-person survival video game, which can be considered accomplished. The scene lighting has been configured to adapt its light color based on the player position using a post process material, reducing the visibility with the exponential height fog, adding the caustics generation and limiting its height based on a maximum and minimum threshold...

For the second objective, a set of highly extensible and configurable character classes has been developed to represent humanoids or animals. A controller was added for the player, allowing him to swim, dive, interact with its environment and defend himself. Finally, artificial intelligence tasks have been created using the Customizable Pathfinding plugin, which generates 3D navigable paths while considering obstacles, providing animals with the ability to wander around the scene, chase the player or attack him. These tasks could also be extended to flying animals as they allow for any type of 3D movement.

Nevertheless some technical aspects still need more work. Primarily, work should be done on the sound aspect, adding ambient sounds such as the bubbles generated by the diver when breathing, the sounds of animals, or adding music to accompany the adventure within the video game. In addition, the underwater-surface transition needs to be improved, by properly calculating the water height to remove the artifacts. The player attack montages could be better adapted to the underwater environment. The ocean material could use the customizable HLSL node to improve the parametrization and scalability of the material.

As future lines, it could be interesting to add more oceanic wildlife to the level, using new tasks to define different movement modes that adapt to each animal specifically. Even, adding new animal attack types such as poison or electricity from jellyfishes. For the character movement, new and improved movement methods could be added, for example vehicles or thrusters that ease the exploration, along with ranged weapons for more convenient defense.

As previously mentioned, Unreal Engine has proven capable of recreating underwater environments, by simulating the light absorption of water and adapting the lighting to an underwater scene. Additionally, it offers plugins to generate 3D paths to simulate the swimming movement of animals and the creation of a player controller to swim and dive in this video game.

Capítulo 8

Presupuesto

8.1. Costes de infraestructura

Para el desarrollo del trabajo, se ha utilizado un ordenador de sobremesa de alta gama con el que poder trabajar cómodamente con el motor de Unreal Engine, valorado en torno a los 2200€

8.2. Costes de desarrollo

El presupuesto estimado para el desarrollo de este proyecto suponiendo un coste por hora de 12€ sería de 4812€, más los 30€ destinados a la compra de *assets* para las animaciones del personaje, resultando en 4852€

Tarea	**Tiempo empleado**
Búsqueda de información y referencias	60 horas
Diseño del videojuego	40 horas
Diseño de materiales	120 horas
Programación	100 horas
Pruebas del videojuego	80 horas
*C*ompra de *assets*	1 hora

Tabla 8.1: Desglose de de tareas y tiempo empleado en cada una de ellas

Capítulo 9

Referencias

Bibliografía

[1] *Behavior Trees in Unreal Engine | Unreal Engine 5.4 Documentation | Epic Developer Community*. n.d. url: https://dev.epicgames.com/documentation/en-us/unreal-engine/behavior-trees-in-unreal-engine?application_version=5.0.

[2] Bethesda. *The Elder Scrolls*. MS-DOS, Windows. Maryland, Estados Unidos, 1994.

[3] *Blend Spaces*. n.d. url: https://docs.unrealengine.com/4.27/en-US/AnimatingObjects/SkeletalMeshAnimation/Blendspaces/Overview/.

[4] Blender Foundation. *blender.org - Home of the Blender project - Free and Open 3D Creation Software*. n.d. url: https://www.blender.org/.

[5] Charles L. Braun y Sergei N Smirnov. "Why Is Water Blue". En: *Journal of Chemical Education* (n.d.). url: https://web.archive.org/web/20191201000418/http://inside.mines.edu/fs_home/dwu/classes/CH353/study/Why%20is%20Water%20Blue.pdf.

[6] Golden Cartridge. "Side Quest — A Deeper Dive into Super Mario 64's "Dire Dire Docks"." En: (jul. de 2023). url: https://medium.com/@TheGoldenCartridge/a-deeper-dive-into-super-mario-64s-dire-dire-docks-c969c5a434f4.

[7] Ben Cloward. *UnderWater Caustics - Advanced Materials*. Youtube. 2024. url: https://www.youtube.com/watch?v=QhOGsNg7Ly4&ab_channel=BenCloward.

[8] Ben Cloward. *Water Depth Shader - UE4 Materials 101 - Episode 24*. 2020. url: https://www.youtube.com/watch?v=TObymSnTwV0&t=622s&ab_channel=BenCloward.

[9] Ben Cloward. *Water Foam - UE4 Materials 101 - Episode 28*. Mayo de 2020. url: https://www.youtube.com/watch?v=oI_q5g3580I.

[10] Ben Cloward. *Water Ripples Shader - UE4 Materials 101 - Episode 23*. Youtube. 2020. url: https://www.youtube.com/watch?v=r68DnTMeFFQ&list=PL78XDi0TS4lGXKflD2Z5aY2sLsD&index=2&ab_channel=BenCloward.

[11] EG Grenade Co. *How to ignite a smoke bomb*. Youtube, 2023. url: https://www.youtube.com/watch?v=e0SgkFcusvI&ab_channel=EGGrenadeCo..

[12] Motion Dreams. *Smoke Flare Effect in Unreal Engine - Niagara System*. Youtube. 2023. url: https://www.youtube.com/watch?v=RUn7nXjkyks&ab_channel=MotionDreams.

[13] *Exponential Height Fog User Guide*. n.d. url: https://docs.unrealengine.com/4.27/en-US/BuildingWorlds/FogEffects/HeightFog/.

[14] Mark Finch y Cyan Worlds. *Chapter 1. Effective Water Simulation from Physical Models*. 2007. url: https://developer.nvidia.com/gpugems/gpugems/part-i-natural-effects/chapter-1-effective-water-simulation-physical-models.

[15] Carlos G. Gurpegui. *Subnautica - Análisis*. IGN. 2018.

[16] *Gameplay Ability*. n.d. url: https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/GameplayAbilitySystem/GameplayAbility/.

[17] Juan Guardado y Daniel Sánchez-Crespo. *Chapter 2. Rendering Water Caustics*. 2007. url: https://developer.nvidia.com/gpugems/gpugems/part-i-natural-effects/chapter-2-rendering-water-caustics.

[18] Itch.io. *Itch.io*. n.d. url: https://itch.io/.

[19] Jourverse. *Underwater Effect with Post Processing Materials in UE5*. Youtube. 2024. url: https://www.youtube.com/watch?v=ALeBZEr--Ts&ab_channel=jourverse.

[20] Konami. *Teenage Mutant Hero Turtles*. Nintendo Entertainment System. Tokio, Japón, 1989.

[21] *Landscape Material Layer Blending in Unreal Engine*. n.d. url: https://dev.epicgames.com/documentation/en-us/unreal-engine/landscape-material-layer-blending-in-unreal-engine.

[22] National Oceanic and Atmospheric Administration. *How far does light travel in the ocean?* 2024. url: https://oceanservice.noaa.gov/facts/light_travel.html.

[23] Nintendo. *Super Mario 64*. Nintendo Entertainment System, IQue Player. Kioto, Japón, 1996.

[24] Nintendo. *Super Mario Bros*. Nintendo Entertainment System. Kioto, Japón, 1985.

[25] NonStatic. *Customizable Pathfinding in Code Plugins - UE Marketplace*. 2022. url: https://www.unrealengine.com/marketplace/en-US/product/customizable-pathfinding?sessionInvalidated=true.

[26] Christian Pérez Hernández. *Portafolio - Recreación de ambientes subacuáticos para un videojuego de supervivencia*. 2024. url: https://www.behance.net/gallery/202518573/TFM-Christian-Jesus-Prez-Hernandez?.

[27] Rockstar Games. *Grand Theft Auto*. PC, PlayStation y GameBoy Color. Nueva York, Estados Unidos, 1997.

[28] Sketchfab. *Sketchfab*. n.d. url: https://sketchfab.com/feed.

[29] *Sonic the Hedgehog*. Mega Drive/Genesis. Shinagawa, Japón: Sega, 1991.

[30] Ubisoft. *Assassins Creed*. PC, Play Station 3, Xbox 360. Montreuil-Sous-Bois, Francia, 2007.

[31] Unknown Worlds Entertainment. *Subnautica*. Pc, Mac, Xbox One, PS4. San Francisco, California, 2018.

[32] Unreal Engine. *AI Perception*. n.d. url: https://dev.epicgames.com/documentation/en-us/unreal-engine/ai-perception-in-unreal-engine?application_version=5.0.

[33] *Unreal Landscape Mode. Unreal sculpt tools to create landscapes*. n.d. url: https://dev.epicgames.com/documentation/en-us/unreal-engine/landscape-sculpt-mode-in-unreal-engine?application_version=5.0.

[34] *Using Light Functions in Unreal Engine | Unreal Engine 5.4 Documentation | Epic Developer Community*. n.d. url: https://dev.epicgames.com/documentation/en-us/unreal-engine/using-light-functions-in-unreal-engine?application_version=5.0.