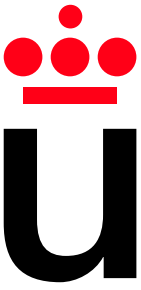

Universidad
Rey Juan Carlos

Escuela Técnica Superior
de Ingeniería Informática

Grado en Diseño y Desarrollo de Videojuegos

Curso 2022-2023

Trabajo Fin de Grado

SIMULACIÓN DE EFECTOS NATURALES PARA ESCENARIOS MARÍTIMOS EN VIDEOJUEGOS

Autor: Javier Raja Huertas
Tutor: Dr. Jorge Félix López Moreno

Agradecimientos

La primera linea de mis agradecimientos no puede ir para otra persona que para mi madre, D.ª Ascensión Huertas María Dolores. Quien ha sido uno de mis mayores apoyos a lo largo de mi vida, y si siquiera este documento ha podido siquiera llegar a realizarse es gracias a su inestimable apoyo. Haciéndome creer en mí y en mis ideas y otorgándome la mayor de las comprensión en los momentos mas duros.

Quería agradecer también su inestimable apoyo a D. Jesús Triviño Miranda, por su incondicional compañía y firme punto de vista a lo largo de todo el camino hasta aquí.

Y por supuesto a toda mi familia y amigos, que han sido partícipes en mayor o en menor medida del camino hasta aquí. No me voy a cansar de dar las gracias; por escuchar mis *parrafadas* sobre como funciona esto y aquello, por hacerme caso cuando decía "*No prestéis atención a este bug*", por darme ideas inocentes que resultan ser clave, y por todas las palabras y gestos bonitos que habéis tenido conmigo a lo largo de este tiempo.

También quería agradecer a mi Tutor D. Jorge Félix López Moreno acceder a formar parte de este proyecto y dar tanta rienda suelta a mi creatividad. Creo que crecido mucho como profesional de su mano, y estoy seguro de que no habría sido posible sin tanta confianza depositada mi.

Por último dedicado a D. Antonio Huertas Pérez. Allá donde estés, gracias por ser la fuente de inspiración de este proyecto.

Resumen

A lo largo de este documento se presenta como y mediante que técnicas es posible la implementación de algunos efectos naturales en los videojuegos. Consintiendo finalmente en la creación de una demostración técnica que aglutine y compacte estos efectos para que finalmente generen un entorno marítimo lo mas realista posible.

A lo largo de este trabajo se ha buscado ofrecer el mayor realismo posible, y se han tratado de representar las técnicas mas novedosas, a fin de familiarizarse con ellas. Así es como finalmente se ha optado por implementar efectos para representar: la Flotabilidad de un objeto en el mar, Nubes Volumétricas y el Ciclo Día-Noche. Además de una interfaz que ofrezca información y control sobre el estado de la escena.

Además se ha optado por investigar acerca de posibles modificaciones o ampliaciones sobre los sistemas ya implementados, a fin de aumentar la interactividad entre estos. De esta forma se han buscado soluciones para tratar de representar el rastro de un barco cuando navega, además de tratar de aumentar el realismo en su iluminación mediante el uso de Sondas de Reflexión.

El resultado final es el de una escena en la cual el *jugador* toma el control de un barco en mitad del océano. Dicho jugador puede, además de controlar el barco, controlar los distintos estados del clima y hora del día mediante una interfaz clara y sencilla. De esta forma el jugador tiene libertad total para interactuar con los distintos efectos en escena y comprobar de que son capaces estas técnicas.

Palabras clave:

- Shader
- Cauce gráfico
- Efectos visuales
- Nubes Volumétricas
- Viento
- Flotabilidad
- Interactividad

Índice de contenidos

Índice de tablas **X**

Índice de figuras **XIII**

Índice de códigos **XV**

1. Introducción **1**
1.1. Descripción del problema 1
1.2. Objetivo principal 1
1.3. Requisitos 2

2. Contexto del proyecto, trabajos previos **4**
2.1. Efectos visuales en los videojuegos 4
2.2. La superficie del mar en los videojuegos 5
2.2.1. Técnicas de aproximación 6
2.2.2. Olas de Gerstner 8
2.2.3. Trasformada rápida de Fourier (FFT) 11
2.2.4. Aproximación Multi-Cascada 15
2.3. Generación y visualización de nubes en videojuegos 16
2.3.1. Textura de 'Skybox' Animado 16
2.3.2. Modelado a través ruido 17
2.4. Ciclo día-noche 19

3. Efectos desarrollados **21**
3.1. Introducción 21
3.2. Detalles de la implementación del Océano 22
3.2.1. Cambios y añadidos 23
3.2.2. Flotabilidad 24
3.2.3. Navegabilidad 26
3.2.4. Velas y bandera 27
3.3. Detalles de la implementación de las Nubes 28
3.3.1. Modelado 29
3.3.2. Renderizado 31

3.3.3. Iluminación . . . 33
3.4. Ciclo día noche . . . 35

4. Resultados finales **37**
4.1. Interfaz de usuario y controles . . . 37
4.2. Océano . . . 39
4.2.1. Flotabilidad . . . 39
4.2.2. Animación de Velas y Banderas . . . 41
4.2.3. Investigaciones y pruebas . . . 43
4.3. Nubes . . . 44
4.3.1. Noise Simulator . . . 44
4.3.2. Gradient Simulator . . . 45
4.3.3. Cloud Simulator . . . 46
4.3.4. Efecto resultante . . . 47
4.4. Skybox . . . 48

5. Conclusiones y trabajos futuros **53**

Bibliografía **56**

Apéndices **58**

A. Código Nubes Volumétricas **60**

B. Código SkyBox Procedimental **69**

Índice de tablas

Índice de figuras

2.1. Fotogramas de los títulos: *Sly Cooper 3: Honor entre ladrones* (2005), desarrollado por Sucker Punch Productions, y *Sea of Thieves* (2018), desarrollado por Rare. 6
2.2. Representación esquemática de la Normal $\vec{N}$, la Tangente $\vec{T}$ y la Binormal $\vec{B}$ de un punto P, perteneciente a la esfera. 7
2.3. Representación gráfica de una Ola de Gerstner genérica. 10
2.4. Representación gráfica de los espectros de energía $E(\omega)$ y de dispersión direccional $D(\theta, \omega)$. 13
2.5. Representación gráfica de un Espectro Oceanográfico dividido en cascadas. 15
2.6. Muestras de unas texturas de 1024x1024 píxeles de ruido de Worley (izquierda) y Perlin (derecha). 19

3.1. Obtención del vector normal $\boldsymbol{N} = A \times B$ a partir de 3 puntos. . . 25
3.2. Corrección de la posición a la hora de sondear la textura de desplazamiento en busca de la altura. 26
3.3. Diagrama UML del sistema de navegación y flotabilidad. [REVISAR] 27
3.4. Diagrama resumen de la implementación de las nubes. 29
3.5. Representación gráfica de las texturas de Ruido de Volumen, a la izquierda, y de Detalle, a la derecha, utilizados en la demostración final. 30
3.6. Representación del camino que realiza un rayo a través de la escena. 31
3.7. Representación de el sondeo de cada uno de los puntos sobre los que se ha marchado. 32
3.8. Representación de la Ley de Beer, el efecto polvo (*powder effect*), y la convolución de ambas. 33
3.9. Comparativa de resultados finales utilizando las distintas relaciones. 34
3.10. Diagrama resumen de la implementación del SkyBox. 36

4.1. Interfaz de información y control de la nave. 38
4.2. Interfaz de control del clima y el tiempo. 38
4.3. Diagrama de los controles. 38
4.4. Vista desde el inspector, de el componente *FloatingObject.cs*. . . . 40
4.5. Visualización desde el inspector de el componente *SailObject.cs*. . 41

4.6. Algunas configuraciones posibles de el Shader de telas. 42
4.7. Resultado aplicación de un desplazamiento sobre la malla resultante. 43
4.8. Aplicación de un "*decal*" con forma triangular sobre la malla. . . 44
4.9. Captura de pantalla de el Resultado Final de las Nubes. 47
4.10. Comparativa de la escena con el uso o no de la Sonda de Reflexión. 48
4.11. Paleta de colores según la hora del día. 49
4.12. Captura de pantalla diurna de la demo final. 51
4.13. Captura de pantalla nocturna de la demo final. 51

Índice de códigos

A.1. Primera parte del shader de Nubes 61
A.2. Segmento de variables del shader de Nubes 62
A.3. Métodos auxiliares del shader de Nubes 63
A.4. Medida de distancias dentro del contenedor en el shader de Nubes y primera parte del muestreo de densidad. 64
A.5. Muestreo de la densidad dada una posición 65
A.6. Método encargado de calcular la luz correspondiente a una posición dada. 66
A.7. Primera parte del shader de fragmentos 67
A.8. Bucle principal asociado a la técnica RayMarching 68
B.1. Propiedades y declaraciones previas 70
B.2. Métodos auxiliares 71
B.3. Shader de Vértices 72
B.4. Shader de Fragmentos 73

1 Introducción

1.1. Descripción del problema

A lo largo de este documento se aborda la creación de una demostración técnica interactiva. La cual presenta un océano que utiliza la trasformada rápida de fourier (FFT), nubes diseñadas a partir de ruido procedimental y un shader en espacio de pantalla, además de un shader que genera una transición entre el día y la noche. En esta demostración, el usuario se convierte en el capitán de un barco que navega por este mar de vela, incluyendo todas las mecánicas que esto conlleva.

1.2. Objetivo principal

El propósito primordial de este trabajo radica en el auto-desarrollo como creador de efectos visuales (VFX), con el objetivo de comprender las técnicas involucradas en LA creación de dichos efectos para eventualmente aplicarlas en la producción de trabajos propios. El deseo de aportar una perspectiva creativa y cohesiva a estos efectos fue una motivación adicional.

El entorno digital está experimentando una rápida evolución, impulsando la innovación y la creatividad en el campo de los gráficos y los efectos visuales. El modelo de interacción y visualización se está volviendo cada vez más sofisticado y dinámico. Esta transformación está permitiendo que los artistas y desarrolladores de efectos se conviertan en verdaderos 'alquimistas digitales', experimentando con

una variedad de técnicas y herramientas para crear experiencias interactivas lo suficientemente creíbles.

En este documento, se proporcionará un análisis detallado de cada uno de los siguientes efectos:

- **Efecto Flotabilidad**. Se ha implementado una extensión sobre un sistema preexistente que permite animar los objetos en escena como si se encontrasen flotando en el mar.
- Nubes Volumétricas. Se ha implementado un sistema de renderizado de nubes volumétricas.
- **SkyBox Procedimental**. Se ha optado por implementar un Shader que genera el efecto día noche. Rotando las distintas luces de la escena y calculando el color de la atmósfera.
- **Deformación de las Velas.** Se ha creado un Shader encargado de aproximar el movimiento las velas por el viento.
- **Interfaz de control.** A fin de conseguir la mayor interactividad con el sistema, se ha optado por implementar una interfaz que permita controlar fácilmente

La intención final no es tanto llegar a conclusiones definitivas como explorar y descubrir técnicas en la creación de efectos visuales y experimentar con ellas.

1.3. Requisitos

Los requisitos mínimos que debe tener el proyecto es que sea capaz de ejecutarse de una manera fluida (60 cuadros por segundo) y estable. Además este debe plasmar con el mayor realismo posible los distintos efectos que forman parte de la escena.

El resto de requisitos del proyecto son los siguientes:

1. **Unity URP**, o Universal Render Pipeline, es una solución de renderización diseñada por Unity Technologies que proporciona un mayor control sobre la organización del cauce gráfico. Usando esta solución los desarrolladores son capaces de ajustar y optimizar la calidad visual y el rendimiento según las necesidades específicas de su proyecto. Además esta solución es muy próxima al ámbito profesional, donde normalmente los motores corporativos requieren un mayor conocimiento de cómo, dónde y porqué se hacen las cosas.

2. **Interfaz de usuario amigable**, pese a tener un contenido jugable escaso, es necesario que el jugador pueda interactuar con las distintas entidades en escena desde una interfaz sencilla y fácilmente entendible. De esta manera se requiere que el jugador sea capaz de poner.

3. **Interactividad entre efectos**, a fin de que la escena genere la mayor capacidad de inmersión posible. Se precisa que los efectos implementados cuenten con la mayor interactividad posible entre ellos. Esto es, por ejemplo, que el océano pueda reflejar el color del cielo, que las nubes que se encuentren justo encima del jugador sean visibles si éste se sumerge unos metros o que los objetos floten de manera suficientemente realista. Además de que, las velas se inflen, las olas viajen, y las nubes se desplacen, en la misma dirección que el viento.

2

Contexto del proyecto, trabajos previos

El universo de los gráficos por computadora siempre ha estado en constante evolución, empujando los límites de lo que percibimos como realista. El surgimiento de nuevas técnicas sofisticadas, tanto como el aumento de la capacidad de los dispositivos, ha revolucionado tanto el sector del cine como el de los videojuegos, permitiendo la creación de experiencias audiovisuales verdaderamente inmersivas y cautivadoras.

Los efectos visuales, o mas comúnmente conocidos como VFX (*Visual effects*), cuentan con multitud de aplicaciones dentro de la industria del entretenimiento. Así pues, en la industria del cine o televisión estos efectos son los encargados de generar las imágenes que no pueden ser captadas en vivo durante la filmación, ya sea porque son demasiado peligrosas, impracticables o simplemente muy costosas de llevar a cabo con técnicas de grabación tradicionales. En los videojuegos, los VFX también son esenciales para mejorar la inmersión y la interactividad, dando vida a elementos como explosiones, fuego, agua, y una variedad de otros fenómenos y efectos ambientales. En este capítulo, desentrañaremos la complejidad de los efectos visuales y las técnicas utilizadas en la demostración técnica final.

2.1. Efectos visuales en los videojuegos

Dentro del sector de la producción de videojuegos, el empleo de efectos visuales una práctica cada vez más prevalente y esencial. Para entender el proceso subyacente de la creación de estos efectos, es crucial abordar el cauce gráfico,

o *pipeline* gráfico, y las etapas de renderizado. Las dos principales etapas programables del renderizado de gráficos son la etapa de `vértices` y la etapa de `fragmentos`. En la etapa de vértices, cada vértice de los modelos es procesado individualmente, son transformados y encuadrados para proyectarlos en el espacio de la cámara. En la etapa de fragmentos, estos vértices transformados se utilizan para generar el color de cada uno de los píxeles que se proyectan en la pantalla.

En este contexto, un shader es un fragmento de código que se ejecuta dentro de la tarjeta gráfica de un ordenador. Se utiliza para configurar y controlar estas operaciones de renderizado a fin de generar el efecto deseado sobre un objeto, la escena, la cámara o crear una textura necesaria para el procesamiento de otro shader posterior. El diseño y la implementación de estos shaders implican la combinación de arte y ciencia. Es por esto que los desarrolladores deben utilizar sus habilidades técnicas y creativas para crear efectos visuales realistas. A menudo es necesario gran cantidad de conocimientos matemáticos, de arquitectura de los motores gráficos y de técnicas propias del sector de los gráficos por ordenador. Existen varios lenguajes de shading, tales como `GLSL` (OpenGL Shading Language), `HLSL` (High-Level Shading Language), y `CG` (C for Graphics). En el caso de Universal Render Pipeline (`URP`), se emplean principalmente `HLSL` y ShaderLab. A este último se le podría describir como un tipo de lenguaje de marcado. Este es empleado para determinar la forma en que el motor de juegos debe realizar la renderización del shader bajo distintas circunstancias. El código Cg/HLSL, por su parte, se utiliza para detallar las consideraciones matemáticas relativas a la operación del shader.

Por ejemplo, se podría emplear ShaderLab para especificar que ciertas partes del shader deben ser transparentes, o que el shader debe usar un cierto tipo de iluminación. Para posteriormente, utilizar Cg/HLSL para describir exactamente cómo debería calcularse dicha iluminación.

Por último, es importante mencionar que muchas de estas técnicas y metodologías empleadas en los videojuegos no son exclusivas de este género, y que pueden ser utilizadas en otros sectores, como el cine. En particular, Houdini, una herramienta líder en la industria del cine para la creación de efectos visuales, se beneficia en gran medida de estas técnicas durante la etapa de post-producción. Houdini utiliza estos procesos y algoritmos avanzados para generar efectos espectaculares que mejoran la estética y el realismo de las producciones.

2.2. La superficie del mar en los videojuegos

El mar, un espacio fascinante y omnipresente, constituye un elemento clave en nuestro entorno natural. Su vastedad y belleza capturan la imaginación y evocan un sentido de asombro y misterio. Normalmente se asocia a menudo con conceptos tan variados como la libertad, el peligro, la aventura y el descubrimiento, todos

Figura 2.1: Fotogramas de los títulos: *Sly Cooper 3: Honor entre ladrones* (2005), desarrollado por Sucker Punch Productions, y *Sea of Thieves* (2018), desarrollado por Rare.

ellos elementos que se prestan con facilidad a la narrativa y al drama.

Por todas estas razones, el mar ha sido una fuente constante de inspiración en diversas formas de arte y medios de comunicación, y los videojuegos no son una excepción. Sin embargo, su incorporación en este medio digital presenta retos particulares. La representación del mar en los videojuegos no solo debe capturar su apariencia visual, sino también transmitir sus cualidades dinámicas a través de la interactividad y la simulación. Esto quiere decir; que dependiendo la naturaleza del título a desarrollar, un mar poco detallado o poco interactivo puede sacar al jugador por completo de la experiencia.

Realizar una aproximación precisa y detallada de la superficie del mar, con sus cambiantes características físicas, supone una considerable dificultad técnica y creativa. Al igual que el resto de los efectos visuales, el desarrollo de los mismos se encuentra supeditado a la capacidad de cálculo de los dispositivos disponibles. Este hecho da lugar a apreciar la diferencia de trece años entre los títulos presentados en la figura 2.1, donde se emplean soluciones totalmente distintas para alcanzar una composición similar.

En el siguiente apartado se profundizará en esta cuestión, explorando algunas de las técnicas exploradas para realizar el efecto de la superficie del mar en la demostración técnica, sus desafíos e implicaciones.

2.2.1. Técnicas de aproximación

En las fases mas tempranas del proyecto se exploraron las siguientes técnicas a fin de familiarizarse con las soluciones preexistentes:

1. **Shader de Oceano**. Si el mar que se precisa implementar no cuenta con mucho protagonismo en el título, pero se necesita representar un oleaje. Es común representar este efecto mediante un shader que se encarga de generar

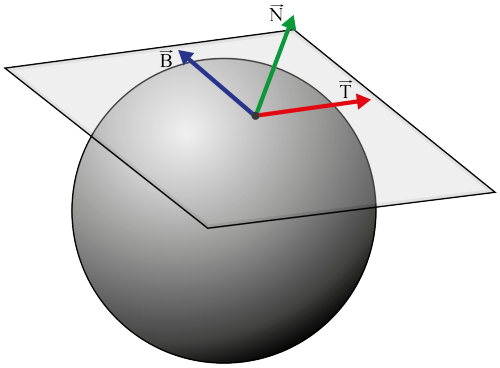

Figura 2.2: Representación esquemática de la Normal $\vec{N}$, la Tangente $\vec{T}$ y la Binormal $\vec{B}$ de un punto P, perteneciente a la esfera.

una deformación de tipo sinusoidal dando a la malla una apariencia de olas del mar y para posteriormente otorgarle propiedades reflexivas (Destellos en la superficie cuando la cámara mira muy paralela a la superficie de este) y refractante (Distorsión en de las partes de un objeto que se encuentren sumergidas total o parcialmente en el agua. Esta distorsión se produce tanto si miramos desde arriba como si miramos desde dentro del mar).

De esta forma se define la posición de cada vértice como $P = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$. Donde P que es la posición del final del vértice en coordenadas locales, viene dada por:

$$P = \begin{bmatrix} x \\ A\sin(kx + \omega t) \\ z \end{bmatrix} \tag{2.1}$$

El parámetro A representa la amplitud de la onda, la k hace referencia al número de onda, ω hace referencia a la frecuencia de la misma y t hace referencia al tiempo en ejecución de la aplicación.

En lugar de tratar de obtener las normales al plano usando la función del motor"*Mesh.RecalculateNormals()*", para obtener las normales de cada punto se usará la siguiente definición:

$$\vec{N} = \begin{bmatrix} -kA\cos(kx + \omega t) \\ 1 \\ 0 \end{bmatrix} \tag{2.2}$$

Esta definición es obtenida de la siguiente forma: primero obtengamos la función que define el vector tangente $\vec{T}$ de cada punto en la dimension x

aplicando la derivada parcial.

$$\vec{T} = \frac{\partial P}{\partial x} = \begin{bmatrix} \frac{\partial x}{\partial x} \\ \frac{\partial A \sin(kx+\omega t)}{\partial x} \\ \frac{\partial z}{\partial x} \end{bmatrix} = \begin{bmatrix} 1 \\ kA\cos(kx+\omega t) \\ 0 \end{bmatrix} \tag{2.3}$$

Por otro lado, el vector normal es el producto vectorial del vector tangente y binormal (2.4). Como estas olas han sido definidas constantes en z, la bitangente es siempre el vector unitario $\vec{k}$, tal que $\vec{B} = \vec{k}$. Finalmente, (2.2) se obtiene después de haber realizado este cálculo.

$$\vec{N} = T \times B \tag{2.4}$$

A este shader se le podrían añadir mas capas, así como, algún tipo de ruido procedimental para tratar de mitigar el efecto patrón, proyectar en el fondo una textura de cáusticas, o incluso algún emisor de particulas que reaccionara al contacto. Dando lugar a un shader mucho mas completo y creíble. Por ultimo uno de los mayores puntos fuertes de esta técnica es que es barata en términos computacionales, si la comparamos con otras técnicas mas complejas. Dejando mucho espacio de ejecución a otros procesos dentro del bucle principal.

Los puntos en contra de esta tecnología es que a menudo presenta patrones repetitivos, puesto que la función que se utiliza para generar la geometría de las olas (2.1) es periódica. Además de que es posible de que haya algunos valores de amplitudes para los que nuestro mar acabe generando artefactos visuales, en los cuales nuestros vértices pueden llegar a superponerse creando bucles, o 'rizos', en las crestas de las olas.

2. **Textura animada**. Otra manera realmente económica de realizar este efecto es la de animar una textura de desplazamiento. De esta forma toda la parte asociada al calculo de posiciones de los vértices queda pre-grabada dejando aún mas hueco dentro del bucle de ejecución del videojuego. Esto la hace la opción mas viable para desarrollos que no necesitan mayor protagonismo por parte de la masa de agua a representar.

   El principal problema de estas animaciones radica su escasa interactividad, y su exagerada monotonía uniformidad. La cual proviene de su condición pre-grabada. Aunque de nuevo, estas desventajas pueden ser mitigadas con emisores de partículas o superposición de ruidos procedimentales.

2.2.2. Olas de Gerstner

Las ondas sinusoidales vistas anteriormente, aunque simples, no capturan fielmente la forma de las olas reales. . En cambio, las grandes olas provocadas por

el viento se modelan de manera más exacta a través de la función de onda de Stokes, cuya formulación resulta bastante compleja. Sin embargo, las ondas propuestas por Franz Josef Gerstner en 1802 son significativamente mas sencillas de implementar en una animación a tiempo real, dado el buen resultado que ofrece.

Al recurrir a este método y aprovechar la interferencia entre ondas, es posible concatenar la suma de varias, incluso miles, de ondas de Gerstner. De este modo, se genera una representación mucho más realista y visualmente rica de la superficie del mar. De esta forma, una onda de Gerstner se define por los siguientes parámetros:

La dirección de la ola se define como:

$$\vec{D} = \begin{bmatrix} D_x \\ D_z \end{bmatrix} t.q. \left|\vec{D}\right| = 1$$

Para controlar fácilmente la longitud de onda, existe el parámetro λ el cual calcula el numero de onda k aprovechándose de que el máximo de la función sin se encuentra en $2\pi \approx 6{,}28$.

$$k = \frac{2\pi}{\lambda} \;\; t.q. \;\; \lambda \neq 0$$

Para evitar la generación de bucles dentro de nuestro ola podemos ajustar la pendiente de la ola $\boldsymbol{s}$.

$$A = \frac{s}{k} \;\; t.q. \;\; 0 \leq s \leq 1$$

Las olas simuladas pueden moverse a cualquier velocidad que se desee, pero en la naturaleza estas suelen hacerlo de forma relacionada al número de onda.

$$c = \sqrt{\frac{g\lambda}{2\pi}} = \sqrt{\frac{g}{k}}$$

Con todas estas variables definidas se define la función:

$$f(x,z) = k\left(\vec{D} \cdot \begin{bmatrix} x \\ z \end{bmatrix} - ct\right)$$

Finalmente la posición de los puntos de la malla en coordenadas locales viene dada por:

$$P = \begin{bmatrix} x + D_x A \cos f(x,z) \\ A \sin f(x,z) \\ z + D_z A \cos f(x,z) \end{bmatrix} \tag{2.5}$$

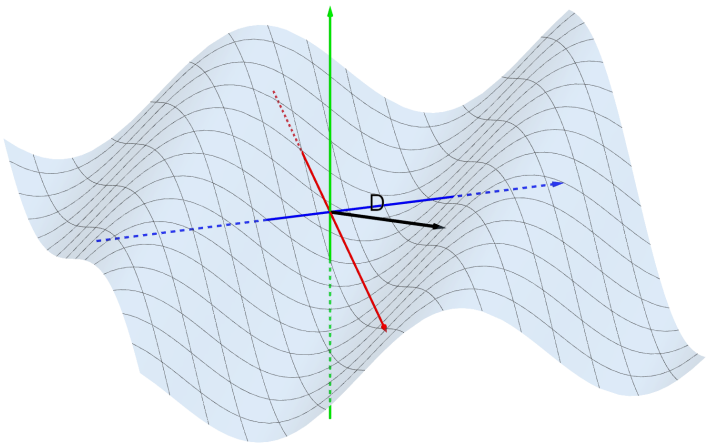

Figura 2.3: Representación gráfica de una Ola de Gerstner genérica.

Esta transformación (2.5) utiliza el desplazamiento vertical para calcular un desplazamiento en los ejes x y z. Si en cambio quisiéramos fijar estas coordenadas podríamos adoptar esta otra variante de la misma.

$$P = \begin{bmatrix} x \\ A \sin f(x,z) \\ z \end{bmatrix} \tag{2.6}$$

El proceso, al igual que en el apartado (2.3), vamos a obtener la tangente aplicando la derivada parcial en la dimension x a la trasformacion definida en (2.5).

$$\mathbf{T} = \frac{\partial P}{\partial x} = \begin{bmatrix} 1 - D_x^2 s \sin f(x,z) \\ D_x s \cos f(x,z) \\ -D_x D_z s \sin f(x,z) \end{bmatrix} \tag{2.7}$$

Para calcular la binomial se realizará un proceso similar pero en este caso con la derivada parcial en la dimensión z.

$$\mathbf{B} = \frac{\partial P}{\partial z} = \begin{bmatrix} -D_x D_z s \sin f(x,z) \\ D_z s \cos f(x,z) \\ 1 - D_z^2 s \sin f(x,z) \end{bmatrix} \tag{2.8}$$

Por ultimo, como hemos establecido anteriormente (2.4), para obtener la normal de cada punto ya solo faltaría realizar el producto vectorial de ambas tangentes.

2.2.3. Trasformada rápida de Fourier (FFT)

Esta es la mas compleja de todas las técnicas que serán expuestas, y la que finalmente ha sido escogida para su utilización en la demostración técnica. La cual esta basada en los artículos de Jerry Tessendorf–"*Simulating ocean water*"[1], y el de Chistopher J. Horvath–"*Empirical directional wave spectra for computer graphics*"[2]. Esta solución se ha convertido en un estándar de la industria a la hora de renderizar, de la forma mas realista posible, la superficie del océano. Esta solucion cuenta con dos componentes clave; la Trasformada Rápida de Fourier y el Espectro Oceanográfico.

Utilizando lo aprendido en el apartado 2.2.1 y 2.2.2 para generar un mar lo mas realista posible se podrían sumar del orden de 10^3 olas. Utilizando la tecnología anteriormente comentada la complejidad de computo sería $\mathcal{O}(nm)$. Siendo n el numero de vértices de la malla y m el numero de olas a representar. Una posible solución para poder realizar esta operación sería utilizar la Trasformada Rápida de Fourier.

Esta técnica matemática permite transformar una función o señal del dominio del tiempo al dominio de la frecuencia, y viceversa. O en términos más sencillos, descompone una señal o función en una suma de ondas sinusoidales de diferentes frecuencias. Cada sinusoidal representa una frecuencia presente en la señal original y su amplitud corresponde a la intensidad de dicha frecuencia. Esto es útil para analizar las características frecuenciales de una señal, como en la música o las señales de radio, o para aplicaciones de procesamiento de imágenes. Aunque en este caso será utilizada para generar el desplazamiento final de los vértices de una malla que se encuentra siendo modificada afectada por gran cantidad de Olas de Gerstner.

La Transformada Rápida de Fourier (FFT, *Fast Fourier Transform*) es un algoritmo que calcula la Transformada Discreta de Fourier (DFT, *Discrete Fourier Transform*). DFT es una Trasformada de Fourier que se define sobre un dominio discreto, y no continuo, como puede ser un mallado uniforme de puntos.

De esta forma el algoritmo FFT, creado por James William Cooley y John Wilder Tukey en 1965, reduce el número de cálculos necesarios para obtener la DFT de n puntos de $\mathcal{O}(n^2)$ a $\mathcal{O}(n \log n)$, lo que es mucho más eficiente.

Para poder aplicar esta técnica, se debe actualizar la matemática de los apar-

tados anteriores usando la formula de Euler (2.9).

$$e^{ix} = \cos x + i \sin x \tag{2.9}$$

$$A \sin(kx + \omega t) \rightarrow h e^{i(kx+\omega t)}$$

Utilizando esta notación, el desplazamiento de la superficie de la suma de ondas se puede calcular atendiendo a la siguiente ecuación.

$$\eta(x,t) = \sum_{k} h_0(k) e^{i(kx+\omega t)}$$

Asumiendo que: el numero de puntos de la malla va a ser igual al numero de ondas, y que las coordenadas delos puntos y los números de onda se encuentran en cuadriculas homogéneas. De esta forma, y operando sobre el anterior sumatorio, podríamos calcular el desplazamiento de N puntos para N ondas.

$$\eta_n(t) = \sum_{m=-\frac{N}{2}}^{\frac{N}{2}} h_m(t) e^{2\pi i \frac{m}{N} n}$$

Ahora se debe definir los parámetros, k, ω y h_0.

El parámetro k hace referencia al número de onda y se encuentra discreto, o definido dentro de un mallado uniforme. El número de onda entonces, para L representando al factor de longitud de onda, se define como:

$$k = \frac{2\pi n}{L}$$

Y se expresan el resto de parámetros como función de él;

$$\omega(k), h_0(k)$$

De manera abstracta es posible modelar una onda con cualquier combinación de número de onda y frecuencia. Pero la naturaleza, estos parámetros están conectados a través de una función, denominada relación de dispersión. Es así como las ondas electromagnéticas en el vacío tienen una frecuencia proporcional al número de onda. Pero en cambio, en el caso de las olas del mar estas están mayoritariamente compuestas por ondas gravitacionales creadas por el viento. Lo que quiere decir que la fuerza perturbadora que actúa sobre el agua es el viento, mientras que la fuerza restauradora es la gravedad. Para estas ondas la relación queda definida de de la siguiente manera.

$$\omega(k) = \sqrt{gk \tan(kd)}$$

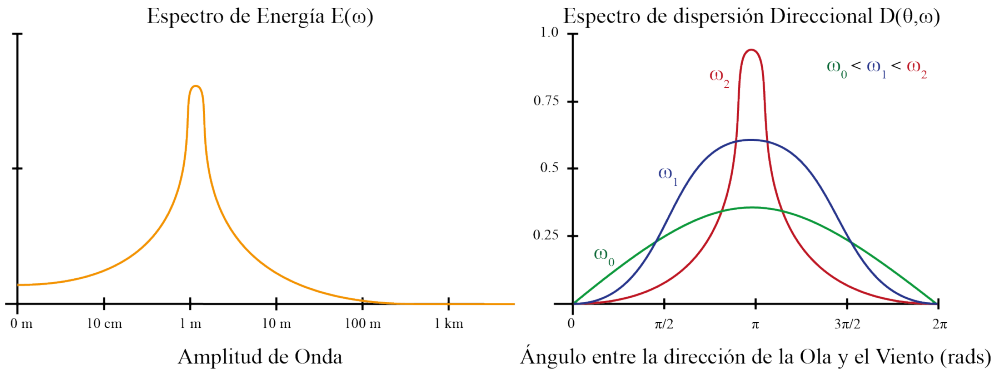

Figura 2.4: Representación gráfica de los espectros de energía $E(\omega)$ y de dispersión direccional $D(\theta, \omega)$.

Con g igual a la aceleración de la gravedad, y d a la profundidad el agua.

El último parámetro, h_0, representa la amplitud compleja de las ondas. El proceso de creación de olas es en gran medida aleatorio. Sin embargo, mientras que la amplitud de cualquier onda en particular es aleatoria, normalmente estas obedecen algunas normas. Por ejemplo, las olas mas grandes suelen tener también mayor amplitud, o las olas que viajan de manera perpendicular al viento son mucho mas pequeñas que las que lo acompañan. Dos tipos de tales relaciones son el espectro de energía $E(\omega)$ y la dispersión direccional $D(\theta, \omega)$

El espectro de energía describe que tan grandes son las olas con una frecuencia determinada, y de manera independiente a la dirección de la onda. Hay distintos modelos de espectro creados por oceanógrafos de todo el mundo, como el modelo Pierson-Moskowitz, el JONSWAP o el TMA. Pero todos suelen contar con varias características comunes. Como se puede apreciar en la muestra de la figura 2.5, todos ellos tienen un corte en la zona de altas amplitudes, o bajas frecuencias, debido a que la distribución de olas posible tiene una longitud de onda máxima posible. Además también tienen una longitud de onda para la que las olas son predominantes y una zona de ondas pequeñas, cortas y muy frecuentes. Algunos de los parámetros usados para modelar este espectro son, la velocidad del viento, la aceleración de la gravedad o la profundidad.

La dispersión direccional representa si las olas tienden a seguir la dirección del viento, o a seguir su propio curso. Esta dispersión depende de el ángulo entre la onda y la dirección del viento, y de la longitud de onda de la misma. De esta forma las olas mas grandes tienden a tener la misma dirección del viento mientras que las mas pequeñas pueden incluso a llegar a ir en contra de el.

Siguiendo las definiciones de los artículos de J. Tessendorf [1] y C. Hobarth [2]. Es posible calcular la amplitud compleja utilizando la siguiente fórmula.

$$h_0(k) = \frac{1}{\sqrt{2}}(\xi_1 + i\xi_2)\sqrt{2S(\omega)D(\theta,\omega)\frac{d\omega(k)}{dk}\frac{1}{k}\Delta k_x \Delta k_z} \tag{2.10}$$

De esta forma dentro de la ecuación 2.10; $(\boldsymbol{\xi_1 + i\xi_2})$ define dos valores aleatorios de una distribución normal con media igual a cero y desciación típica igual a uno, $\xi_n \sim N(0,1)$. Y la ultima parte $\frac{d\omega(k)}{dk}\frac{1}{k}\Delta k_x \Delta k_z$ viene de la conversión de un espectro de frecuencias a un numero de onda. Para evolucionar este espectro en el tiempo, de acuerdo con J.Tessendorf [1], es preciso utilizar la ecuación (2.11). La cual garantiza que el resultado de la Trasformada Rápida de Fourier sea un número perteneciente al conjunto de los números reales.

$$h(k,t) = h_0(k)e^{i\omega(k)t} + h_0(-k)e^{-i\omega(k)t} \tag{2.11}$$

Esta amplitud compleja es para calcular el desplazamiento vertical de la superficie.

$$\eta_y(x,t) = \sum_k h_0^y(k)e^{i(k\cdot x+\omega t)}$$

Pero como anteriormente ha sido comentado en el punto (2.5), es posible calcular las amplitudes del desplazamiento horizontal usando la amplitud vertical de la siguiente forma, Multiplicamos por $\boldsymbol{i}$ para combertir los senos en cosenos y dividimos por el vector normalizado de la onda para obtener la dirección del desplazamiento.

$$h_0^{(x)} = ik_x\frac{1}{k}h_0^{(y)}$$
$$h_0^{(z)} = ik_z\frac{1}{k}h_0^{(y)}$$

Para poder renderizar estas posiciones es necesario que sean calculadas las normales, y para esto han de ser calculadas las derivadas parciales en las dimensiones x y z.

$$s = \left(\frac{\frac{\partial \eta_y}{\partial x}}{1+\frac{\partial \eta_x}{\partial x}}, \frac{\frac{\partial \eta_y}{\partial z}}{1+\frac{\partial \eta_z}{\partial z}}\right)$$
$$n = \frac{(-s_x, 1, -s_z)}{\|(-s_x, 1, -s_z)\|}$$

Es posible estimar estas derivadas parciales a través del propio desplazamiento de manera numérica. Pero como la derivada parcial y la trasformada de Fourier son trasformaciones lineales, es posible intercambiar sus ordenes y calcularlas dentro

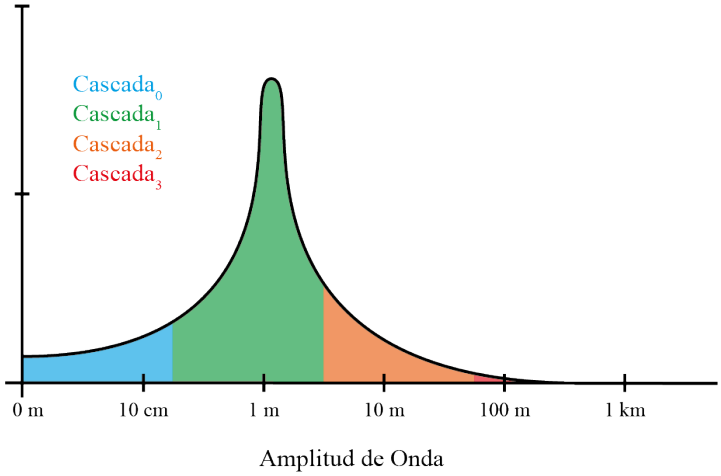

Figura 2.5: Representación gráfica de un Espectro Oceanográfico dividido en cascadas.

de la Trasformada Rápida de Fourier.

$$\frac{\partial}{\partial x}\eta(x,t) = \sum_k \frac{\partial}{\partial x} h_0(k) e^{i(k\cdot x+\omega t)}$$

$$\frac{\partial}{\partial x}\eta(x,t) = \sum_k ik_x h_0(k) e^{i(k\cdot x+\omega t)}$$

2.2.4. Aproximación Multi-Cascada

Esta técnica, cuando se trata con precisión, puede requerir una gran cantidad de cálculos de transformada rápida de Fourier (FFT). Sin embargo, en un intento de optimizar esta alta demanda computacional, se encuentra ya implementada una técnica de un enfoque multi-cascada.

En el contexto del análisis de Fourier y de señales, una"cascada" en FFT generalmente se refiere a una visualización tridimensional o plana coloreada del espectro de frecuencias de una señal a lo largo del tiempo. En este contexto concreto se refiere a una serie de Trasformadas que se realizan en 'tiles' o celdas de diferentes longitudes de onda, formando una cadena de cálculos FFT que se realizan en paralelo. De esta forma cada cascada representa un conjunto de longitudes de onda, donde el rango de números de onda de cada cascada se determina por su escala de longitud y resolución. Para ilustrar esto, consideremos una serie de cascadas, cada una con una longitud de onda y resolución diferentes, todas contribuyendo a formar el aspecto final del océano.

El método multi-cascada permite un aumento en la resolución espacial sin aumentar necesariamente la resolución de la FFT. Este enfoque mejora la eficien-

cia computacional ya que no necesita calcular trasformadas grandes para que las olas se vean bien. Sin embargo, se debe tener en cuenta que los bordes de estas cascadas son críticos para la calidad visual final. Las olas pequeñas que terminan en una cascada grande no tendrán suficiente detalle espacial, y las olas grandes en las cascadas pequeñas pueden hacer que el embaldosado sea notorio.

En resumen, la técnica de cascada múltiple se presenta como una estrategia muy efectiva para mejorar la eficiencia a la hora de aproximar los desplazamientos. Mediante el ajuste meticuloso de las longitudes de onda y la resolución de cada cascada, es posible alcanzar un equilibrio idóneo entre el rendimiento computacional y la calidad visual.

2.3. Generación y visualización de nubes en videojuegos

Para quien navega, el cielo y el mar se convierten en los únicos compañeros de viaje. Los mares pueden estar en calma o rugir con fuerza, pero el cielo con sus nubes siempre está allí, otorgando información al marinero sobre el cambiante clima, y personalidad a la escena.

Por lo tanto, al trasladar esta realidad a la demostración técnica, es crucial poder representar de manera fiel y creíble estos elementos. No solo son las nubes meros adornos visuales; son, de hecho, una parte integral de la atmósfera y del realismo en un videojuego. Las nubes pueden marcar la diferencia entre un juego que parece plano y uno que nos sumerge de lleno en un entorno tridimensional y dinámico.

La forma en que los videojuegos representan las nubes puede variar enormemente, desde simples texturas animadas en el cielo hasta simulaciones climáticas complejas y dinámicas. En esta sección se exploraran, las técnicas exploradas para su representación y final implementación dentro de la demostración técnica.

2.3.1. Textura de ‘Skybox’ Animado

Es una técnica ampliamente utilizada en el diseño de videojuegos, que permite dar vida al cielo virtual de nuestros mundos. Este método, relativamente simple pero efectivo, involucra el uso de texturas proyectadas en un çajón"que envuelve la totalidad del espacio de juego, creando la ilusión de un cielo vasto y lejano.

Esta técnica da un paso más allá al permitir que las texturas proyectadas cambien y se muevan, dando lugar a la aparición de nubes que flotan por el cielo, estrellas que parpadean y cambian con el tiempo, o el paso del día a la noche y

viceversa. Sin embargo no está exenta de limitaciones, su naturaleza estática y predefinida puede resultar en cielos y nubes que carecen de variedad y realismo a largo plazo. Además de que estás nubes se sitúan en un lugar inalcanzable para el jugador, por lo que sería imposible interactuar con estas nubes, así como 'volar' a través de ellas, o que estas reaccionen de manera creíble a los cambios en la dirección del viento, o los cambios en el clima. En pocas palabras todo lo que no se encuentre previamente animado queda excluido de las situaciones a poder representar.

2.3.2. Modelado a través ruido

Debido a la naturaleza de este proyecto era preciso una técnica que permitiera un mayor control sobre las nubes en escena. Es por eso que se ha decidido implementar la técnica presentada por Andrew Schneider en su conferencia "The real-time volumetric cloudscapes of Horizon: Zero Dawn", en la conferencia SIGGRAPH'15 [3].

A manera de resumen, esta técnica gráfica nos permite renderizar nubes volumétricas realistas de forma muy económica en términos de tiempo de computación. Para logralo, utiliza varias capas de ruido procedimental de distintos tipos para modelar la forma de las nubes a pintar en la escena, para posteriormente, utilizando técnicas de sondeo iluminarlas de la manera realista posible.

Antes de seguir ahondando en esta técnica es preciso que se explique como se obtienen los ruidos procedimentales, que a posteriori servirán para modelar la forma irregular de las nubes. Ya que los artistas de VFX necesitan entender cómo funcionan los diferentes tipos de ruido y cómo pueden ser controlados y modificados para crear los efectos deseados.

- **Ruido de Perlin**. El Dr. Ken Perlin, insatisfecho con la apariencia"artifial"de las imágenes generadas por ordenador en 1983, desarrolló el ruido Perlin. Esta técnica se detalló en su artículo para SIGGRAPH en 1985,"An Image Synthesizer"[4]. A modo de curiosidad, Perlin desarrolló esta técnica tras trabajar en la icónica película de ciencia ficción, Tron (1982). Para generar este ruido, es necesario realizar los siguientes pasos:

  1. **Se define una cuadrícula de puntos de un espacio n-dimensional**. En cada uno de estos puntos de la cuadricula se asigna un vector dirección aleatorio.
  2. **Para cada ubicación en el espacio que deseas conocer el ruido a partir de ahora p**. Se encuentra la cuadrícula que lo encierra, y se calcula un vector desde cada punto de la cuadrícula hasta p.
  3. **Calcular el producto escalar de cada vector de la cuadrícula**

con el vector calculado en el apartado anterior. Esto genera un número para cada punto de la cuadricula que rodea a p.

4. **Interpolar entre estos números para obtener el valor del ruido en el punto intermedio**. Esta interrelación puede ser o no lineal, depende de la apariencia final que deseemos para la textura de ruido.

- **Ruido de Worley**, también conocido como ruido celular o ruido de Voronoi, es un tipo de ruido procedimental ampliamente utilizado en gráficos por ordenador. Este método fue nombrado así por Steven Worley, quien describió el algoritmo en 1996 en su articulo 'An Image Synthesizer"[5] durante la conferencia SIGGRAPH de ese año.

  En esencia, el ruido de Worley se genera creando un conjunto de puntos en un espacio (a menudo se piensa en un espacio de dos dimensiones para simplificar, pero podría ser tres, cuatro, o sucesivas). Luego, para cada ubicación en ese espacio, se calcula la distancia al punto más cercano. Esto crea una serie de celdas, cada una de las cuales es el área más cercana a un punto en particular. Estas celdas forman lo que se conoce como un Diagrama de Voronoi. Cada celda tiene un punto "semilla.[en] su centro y todas las ubicaciones dentro de la celda están más cerca de su semilla que de cualquier otra.

  El ruido de Worley se puede usar para generar texturas que tienen un aspecto orgánico, ya que puede producir patrones que se parecen a las células bajo un microscopio, las manchas en un animal, o las grietas en la lava seca, entre otras posibles aplicaciones.

  Un aspecto interesante del ruido de Worley, es que puede ser manipulado de varias formas para producir diferentes tipos de patrones. Por ejemplo, en lugar de tomar la distancia al punto más cercano, se podría tomar la distancia al segundo punto más cercano, o la diferencia entre las distancias al punto más cercano y al segundo más cercano, y así infinidad de tratamientos. Estas y otras posibles variaciones pueden generar una variedad de efectos visuales.

El ruido procedimental es una técnica fundamental en la creación de efectos visuales e imágenes generadas por ordenador. Suele utilizarse para agregar detalles y realismo a las texturas, superficies, partículas y otras características de una escena.

El ruido procedimental se genera mediante algoritmos, en lugar de ser muestreado de un mapa de textura preexistente. Esto significa que puede ser creado y modificado dinámicamente en tiempo real, lo que lo hace altamente flexible y adaptable a diferentes necesidades. Además, como se genera matemáticamente, puede ser escalado a cualquier resolución sin perder detalle o calidad.

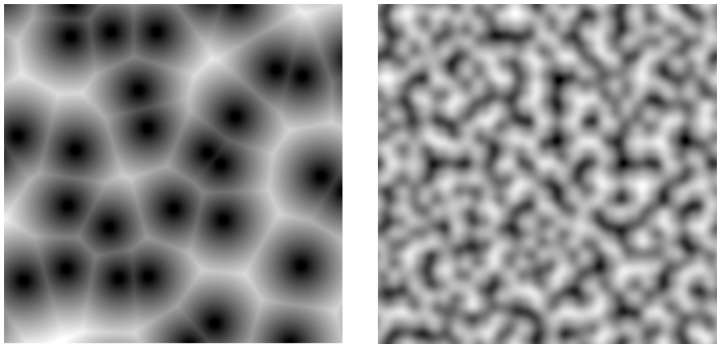

Figura 2.6: Muestras de unas texturas de 1024x1024 píxeles de ruido de Worley (izquierda) y Perlin (derecha).

2.4. Ciclo día-noche

Como se detalló en los requisitos (1.3) de la demostración técnica, es importante tratar de representar el mayor número de condiciones en las que se puede encontrar el medio natural. Para ello, se ha implementado un shader que genera una 'SkyBox' de forma procedimental según vaya avanzando la hora del día en la que nos encontremos. Este sistema tiene además que simular la progresión del tiempo, que incluye la transición del día a la noche, la aparición de estrellas en el cielo y fenómenos como el amanecer y el atardecer. Además de difuminar el sol como sucede en el medio natural.

Esto último se logra gracias a un fenómeno denominado Dispersión de Mie. El cual es un tipo de dispersión de la luz que ocurre cuando las partículas en un medio son del mismo orden de magnitud que la longitud de onda de la luz. Esto es común, por ejemplo, en la atmósfera de la Tierra, donde la dispersión de Mie es responsable de varios fenómenos relacionados con la apariencia del cielo y las nubes.

En gráficos por ordenador, a menudo se utiliza una versión parametrizada de la función de fase de Mie que incluye un factor g, que representa la asimetría de la dispersión. Un valor de g cerca de 0 produce una dispersión más isotrópica (en todas direcciones), mientras que valores cerca de 1 o -1 producen una dispersión más dirigida en la dirección de la luz incidente o en la dirección opuesta, respectivamente. La ecuación parametrizada se conoce como la función de fase de Henyey-Greenstein (3.1).

3

Efectos desarrollados

3.1. Introducción

A lo largo de este apartado se abordará como y mediante que tipo de herramientas han sido implementadas las técnicas gráficas introducidas en el capítulo anterior. Pero antes se debe comentar algunas características del motor utilizado, las cuales permiten que estas técnicas tengan el menor impacto posible dentro del bucle de ejecución de la demostración técnica.

- **Scriptable Renderer Feature**. Parte integral de Unity Universal Render Pipeline (URP), permite personalizar el proceso de renderizado para satisfacer unas necesidades específicas. Estos ficheros de código personalizados o 'scripts', pueden ser agregados a un *Renderer* para cambiar la forma en la que URP renderiza cada cuadro o 'frame'.

  Un '*RenderPass*', por tanto, es una unidad de trabajo que el Renderer ejecuta. De esta forma cada *RenderPass* realiza una tarea específica, como renderizar todos los objetos opacos, procesar la iluminación, aplicar el post-proceso, etc.

  El orden en que se ejecutan estos *RenderPasses* en URP es crucial para obtener los resultados visuales deseados y optimizar el rendimiento. Al igual que en cualquier pipeline de renderizado, cada paso está diseñado para preparar la escena, los objetos y la iluminación de una manera específica antes de pasar al siguiente paso.

  A pesar de la notable flexibilidad y el potencial de personalización que

ofrecen los *ScriptableRendererFeatures*, es importante considerar algunas desventajas potenciales. Por un lado, la creación de estas puede representar un desafío debido a su complejidad, y requiere un sólido entendimiento del proceso de renderizado. En términos de optimización, es vital tener precaución para no introducir ineficiencias en el pipeline de renderizado. Por ejemplo, una mala gestión podría llevar a renderizar los mismos objetos repetidamente, lo que podría afectar negativamente sobre el rendimiento. En relación con la compatibilidad, cabe mencionar que no todas las características y efectos de renderizado son compatibles entre sí. Esto significa que podrían surgir problemas si se intenta combinar ciertas características de formas no previstas.

- **Shaders de Computo**, o '*Compute Shaders*'. Son una característica de los modernos sistemas de gráficos que permiten utilizar la GPU para tareas de cómputo en general, no sólo para renderizado de gráficos. Esta capacidad para realizar cálculos en paralelo hace que los *Compute Shaders* sean increíblemente eficientes para ciertos tipos de tareas que pueden ser paralelizadas, como procesamiento de imágenes, simulaciones físicas y procesamiento de geometría.

  En Unity, los Compute Shaders se escriben en un lenguaje de alto nivel similar a HLSL/Cg, y se ejecutan en la GPU. Estos pueden manipular y acceder a buffers y texturas de manera más flexible que los shaders tradicionales, lo que los hace útiles para una amplia gama de tareas.

  Así como en generaciones pasadas de videojuegos se utilizaban Shaders de Geometría, o 'Geometry Shaders' para la generación y manipulación de geometría directamente en la GPU. Los Shaders de Computo nos pueden ayudar a realizar este tipo de tareas de forma mas optimizada y rápida.

3.2. Detalles de la implementación del Océano

Tras una extensa investigación sobre la implementación de la Transformada Rápida de Fourier (FFT) para la generación de un mar en Unity URP, se descubrió un repositorio en GitHub [6] desarrollado por Ivan Pensionerov (*@gasgiant*) que ya había abordado satisfactoriamente esta problemática. Este repositorio, con una licencia MIT que permite su libre utilización y modificación, presentó un excelente nivel de trabajo y fue motivo de inspiración. Se identificaron áreas potenciales de ampliación y, consecuentemente, se decidió descargar el repositorio y adaptarlo a la demo técnica, dando lugar a una versión ajustada a las necesidades del sistema.

Esta implementación se caracteriza por un extenso conjunto de componentes que facilitan todas las operaciones necesarias para la visualización del Océano. Se abarca desde el aproximado de la geometría hasta el pintado de la espuma del

mar, incluyendo implementaciones de varios modelos de espectro oceanográfico, así como un sistema propio de marcado para la visualizacion en el editor. Se optó, entonces, no por una implementación desde cero, sino por trabajar a partir de lo existente, como si se tratara de la incorporación de un artista a una plantilla con un marco de trabajo ya establecido.

En una perspectiva general, la composición del océano se distingue por dos elementos fundamentales: *OceanSimulation.cs*, que asume la tarea de actualizar el espectro y aproximar la geometría de la malla, y *OceanRenderer.cs*, encargado de la tarea de gestionar los distintos parametros de renderizado de esta geometría, supervisando todas las variables del shader del Océano.

Además, se dispone de su propia *ScriptableRendererFeature*, equipada con tres *ScriptableRenderPass*: *OceanGeometryPass*, *OceanUnderwaterEffectPass*, y *OceanSkyMapPass*. Cada uno de estos se encarga de ejecutar los búferes de comandos, o en inglés "*commands buffers*", asociados a su respectivo proceso, así como de generar los efectos de postprocesado a los que están vinculados.

Es importante destacar que un búfer de comandos se define como una lista de comandos de renderizado que se pueden grabar y posteriormente ejecutar en momentos específicos durante la cadena de renderizado.

3.2.1. Cambios y añadidos

Aunque la intención inicial era aprovechar ciertos elementos y mejorar aquellos aspectos donde se detectaban debilidades en el sistema, es importante destacar la alta calidad del trabajo original. Incluso cuando se realizaron algunas mejoras, la mayor parte del sistema inicial se mantuvo intacto debido a su sofisticación y efectividad.

Dentro de las modificaciones que se han llevado a cabo en el sistema, es necesario destacar dos aspectos principales: la optimización de la claridad del océano durante las inmersiones y la implementación de una serie de cambios orientados a mejorar la interactividad con la escena, a través de un tratamiento diferenciado de la iluminación principal que incide sobre el océano, dependiendo si es de día o de noche.

En relación con la optimización de la claridad durante las inmersiones, se ha incorporado un parámetro adicional en el *"Ocean.Shader"*. Este nuevo parámetro tiene la función de regular el efecto de la neblina cuando la cámara se encuentra bajo el agua. Este efecto se logra por medio del *"ScriptableRenderer"*, más específicamente en la *ScriptableRenderFeature* llamada *"OceanRendererFeature"*. Este componente tiene la capacidad de detectar si la cámara está sumergida, sombreando la escena en consecuencia.

Con respecto a las mejoras en la interactividad con la escena, se han efectuado ajustes para solucionar los inconvenientes surgidos con la inclusión de la *"SkyBox"*, la cual generaba efectos no deseados,o artefactos, al adicionar dos fuentes de luz (la Luna y el Sol). La solución implementada ha consistido en la inclusión de un parámetro que representa la dirección de la luz principal, el cual es ajustado por los componentes que manejan el *"SkyBox"*.

3.2.2. Flotabilidad

Como se mencionó en la sección de requisitos al inicio del documento (1.3), los efectos introducidos en la escena deben ofrecer el máximo nivel de interactividad posible. En este contexto, uno de los comportamientos más interactivos que puede presentar un masa de agua es la flotabilidad. Para implementar este comportamiento se ha creado el componente "FloatingObject.cs", el cual recoge toda la funcionalidad asociada a este efecto.

Para hacer que un objeto genere la sensación de que flota es necesario conocer: el desplazamiento vertical η_y, o altura, y los desplazamientos horizontales η_x y η_z de cada vértice de la malla que constituye el océano. Dicha información se procesa en la tarjeta gráfica (GPU). Sin embargo, para la realización de estos cálculos, se requiere que dicha información resida en el procesador (CPU). Este hecho plantea la necesidad de una solución ingeniosa para trasladar la información de la GPU a la CPU sin generar un impacto significativo en el rendimiento del sistema. Para solucionar este problema fueron exploradas múltiples técnicas:

- **FFT en CPU**. Una posible solución sería la de tratar de procesar la cascada con las longitudes de onda mas grandes en CPU. Esto permitiría un acceso directo e instantáneo a la información a nivel de componente, evitando la latencia asociada al tráfico de datos entre estos dispositivos. Sin embargo, hay que tener en cuenta que la ejecución de la Transformada Rápida de Fourier (FFT) en la CPU puede ser computacionalmente muy costosa, con una relación coste-beneficio no muy favorable en este escenario particular. El objetivo principal es simular aproximadamente el movimiento de un objeto flotante, el cual es inherentemente variable y susceptible a múltiples influencias. Por lo tanto, la eficiencia y la precisión razonable podrían ser preferibles sobre la exactitud rigurosa que podría obtenerse con esta solución.
- **Lecturas asíncronas**. Una alternativa efectiva a la tradicional llamada de lectura a la tarjeta gráfica, que conlleva una inevitable interrupción de su actividad, sería emplear una modalidad que no induzca dicha parada. En el contexto de Unity URP, esta funcionalidad está encapsulada en la clase "AsyncGPUReadBack", la cual presenta un leve desfase, de unas pocas décimas de segundo, respecto a las operaciones realmente en curso. Dado

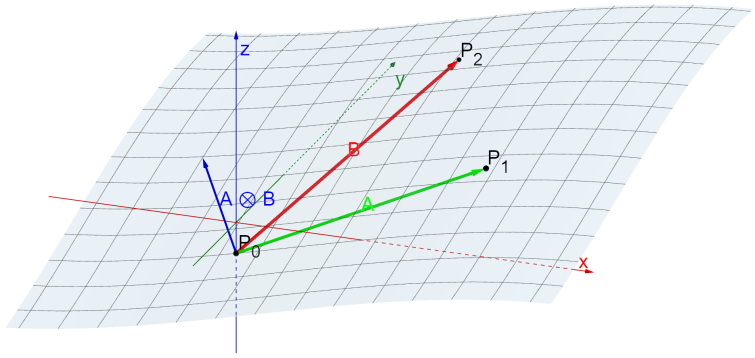

Figura 3.1: Obtención del vector normal $\boldsymbol{N} = A \times B$ a partir de 3 puntos.

el carácter del problema expuesto anteriormente, esta solución parece ajustarse mejor con las necesidades y requisitos de la demostración técnica. Al permitir la ejecución simultánea de otros procesos, se optimiza el uso de los recursos y se evita la paralización completa del sistema, lo que a su vez conduce a una mayor eficiencia en el rendimiento global.

Toda la funcionalidad asociada a estas lecturas se encuentra dentro de la clase "OceanCollision.cs". La cual es una clase estática que permite la lectura de los desplazamientos por parte de cualquier componente que lo necesite.

De esta forma el componente *"FloatingObject.cs"* solo necesitaría que se defina un número n de puntos p_n para consultar sus desplazamientos. Estos puntos son generados a través del editor de Unity y su posición puede ser modificada con facilidad. Por cada uno de estos puntos, se definen dos puntos adicionales a una unidad de distancia en los ejes x y z locales, y se calcula la normal en ese punto. Una vez calculadas las normales en estos puntos se calcula la media de los resultados, y así obtener el valor de la normal que se aplicará al modelo que se pretende hacer flotar, tal como se muestra en la figura 3.1.

$$\boldsymbol{N_i} = \left(p_i + \overrightarrow{i}\right) \times \left(p_i + \overrightarrow{k}\right)$$

$$Normal = \overline{\boldsymbol{N}} = \frac{\sum_{i=1}^{n} N_i}{n}$$

Además de aplicar al modelo esta normal, también es necesario que se calcule la altura del modelo h y sus desplazamientos horizontales d_x y d_z. Para obtener el desplazamiento final tan solo sería necesario obtener el desplazamiento en cada punto y hacer la media de los resultados. Estos desplazamientos, al igual que la

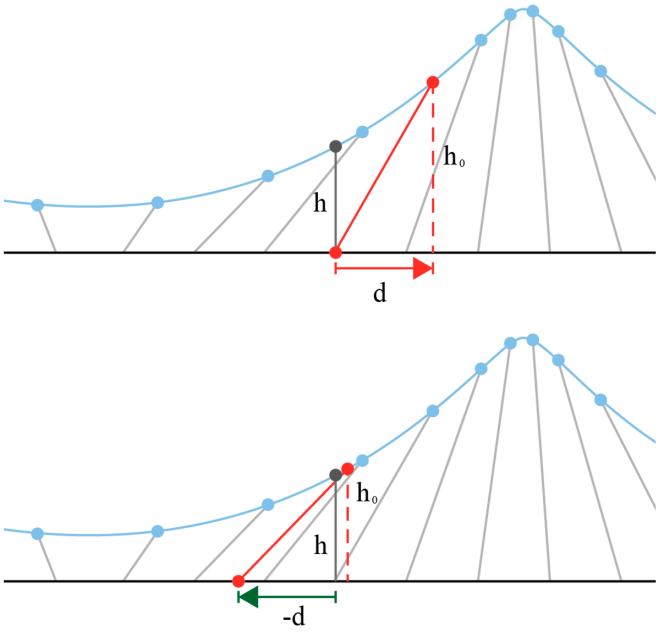

Figura 3.2: Corrección de la posición a la hora de sondear la textura de desplazamiento en busca de la altura.

altura, pueden ser consultados a través de los métodos *GetWaterDisplacement()* y *GetWaterHeight()*. Es necesario apuntar que para obtener la altura se itere varias veces aplicando el desplazamiento al punto sondeado, como se aprecia en la figura 3.2. Puesto que cuando se hace la lectura en un punto, devolverá la altura del punto desplazado. De esta forma el método anteriormente mencionado *GetWaterHeight()* consulta varias veces *GetWaterDisplacement()* en busca de obtener una altura suficientemente aproximada a la real de ese punto.

$$\boldsymbol{d_i} = d(p_i)$$

$$Desplazamiento = \overline{\boldsymbol{d}} = \frac{\sum_{i=1}^{n} d_i}{n}$$

3.2.3. Navegabilidad

Resulta esencial que, si se desea, un objeto pueda poseer la capacidad de desplazarse por esta superficie, motivo por el cual se ha implementado un componente para esta funcionalidad, "*NavigableObject.cs*". El cual es el componente encargado de otorgar el movimiento al objeto por la superficie. Adicionalmente, y por peculiaridades de la demostración técnica se precisaba alguna manera de navegar a vela. Por lo que adicionalmente se ha añadido un componente "*Wind.cs*"

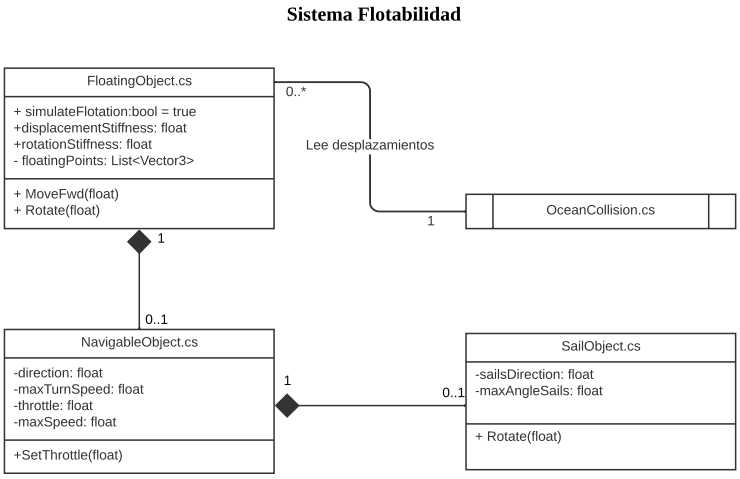

Figura 3.3: Diagrama UML del sistema de navegación y flotabilidad. [REVISAR]

encargado de gestionar el viento de manera global.

De esta forma el componente "*SailObject.cs*" tiene la responsabilidad de generar un empuje variable en función de la orientación de la vela, siendo mayor o menor según la perpendicularidad de esta respecto al sentido del viento. Esta orientación del viento se puede observar tanto en el HUD como en la vela situada en lo alto del mástil del barco. Además claro del sentido del oleaje y el movimiento de las nubes.

3.2.4. Velas y bandera

Tal como se abordó en el apartado previo, la nave presente en la escena es un velero; por ende, es necesario animar las velas para lograr una mayor fidelidad y sensación de realismo. De igual forma, con la bandera de la embarcación.

Con el objetivo de afrontar dicha problemática, se exploraron diversas tecnologías:

- **Animación Pregrabada:** Inicialmente, se contempló la posibilidad de representar las telas mediante un software de modelado 3D, tal como Blender o 3DS Max, y grabar una animación que emulase de manera realista el movimiento de la malla como resultado de la acción del viento. Entre las ventajas de esta técnica se encuentran un menor consumo de recursos, dado que la animación ya está precalculada. Y un resultado acotado, lo que significa que el sistema no puede generar ninguna geometría que no haya sido previamente almacenada en la animación, evitando así la aparición de

artefactos indeseados. No obstante, esta tecnología fue descartada para la demostración final debido a la notoria repetitividad de la animación y al elevado costo que supondría alcanzar un resultado satisfactorio, dada la falta de familiaridad con este tipo de programas.

- **Sistema Masa-Resorte:** En la industria de gráficos por computadora, cuando se trata de simular el comportamiento de telas y estructuras, es frecuente recurrir al sistema Masa-Muelle, o "*Mass-Spring*". Este enfoque implica tomar una malla de vértices, a los que se denominará nodos, y conectarlos mediante una serie de resortes con distintas constantes elásticas. Como resultado, se obtiene una simulación más dinámica y adaptable. Sin embargo, este método también presenta desafíos, tales como el alto costo computacional asociado al cálculo de posiciones y deformaciones, y la posibilidad de que surjan artefactos indeseados, nudos o malformaciones, en ciertas configuraciones y posiciones de la malla. Aunque este enfoque fue finalmente descartado debido a su rendimiento; tras realizar pruebas con ComputeShaders y animación en CPU con mallas más pequeñas, se determinó que el realismo adicional no justificaba el consumo de recursos.
- **Shader de Deformación:** Dado que uno de los problemas identificados con los enfoques anteriores era el consumo excesivo de recursos, se buscó una alternativa que permitiese realizar la deformación de manera eficiente y realista. Ya con conocimiento en técnicas de generación de ruidos procedimentales, se optó por desarrollar un shader que alterase las posiciones de los vértices mediante el muestreo de un ruido de Perlin variable en el tiempo. Aunque este método no proporciona el movimiento más realista posible, sí es considerablemente más eficiente en términos de rendimiento. Además, permite la utilización del mismo shader para aplicar texturas personalizadas u otros efectos.

3.3. Detalles de la implementación de las Nubes

Para poder llevar a cabo la técnica descrita en el apartado (2.3.2) se necesita, en primer lugar, implementar un componente para crear los ruidos de Worley y Perlin necesarios. Ya que para modelar las nubes se usarán dos texturas tridimensionales de ruido de Worley multi-canal y una textura unidimensional en escala de grises de ruido de Perlin. Además de una serie de parámetros artísticos los cuales influyen en las distintas propiedades del resultado final observable (densidad, absorcion de la luz, color, orientación de la luz...).

Para la generación de estas texturas de forma fácil y controlada, se ha optado por la creación de dos componentes diferenciados encargados de estas tareas. Denominados *NoiseSimulator*, encargado del ruido de Worley usado para modelar las nubes y *GradientSimulation*, encargado del ruido de Perlin y otras técnicas,

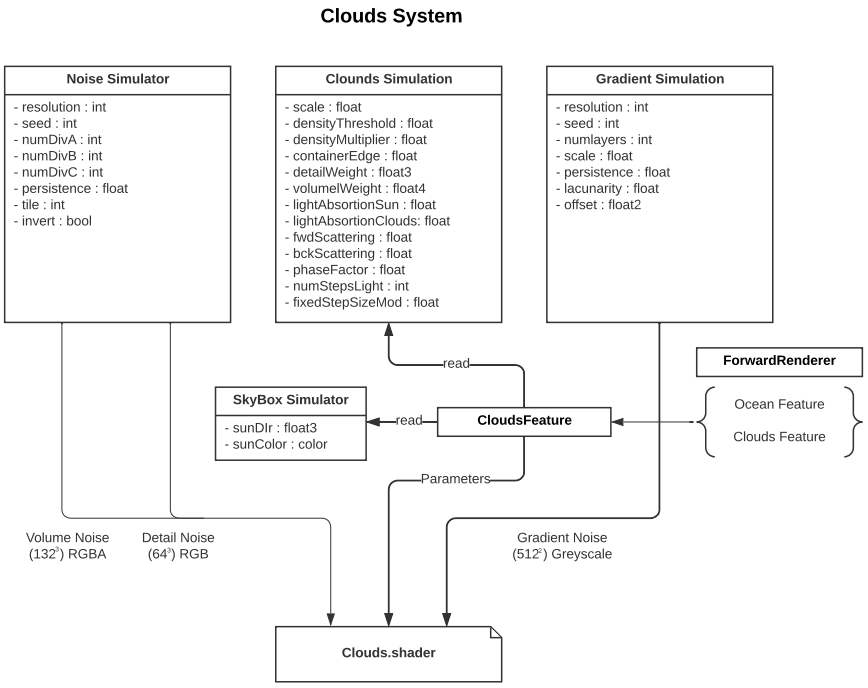

Figura 3.4: Diagrama resumen de la implementación de las nubes.

como la utilización de gradientes de altura, para transformar este modelado creando zonas con mayor y menor densidad, o la aparición de los distintos tipos de nubes.

De esta forma, y con el objetivo de controlar los distintos parámetros artísticos, se ha creado su propio Scriptable Renderer Feature (Clouds Feature). El cual es ejecutado dentro del FordwardRenederer de la aplicación final y se encargará de leer los datos correspondientes a cada estado de la escena, y propagarlos al shader en espacio de pantalla "*Clouds.shader*". La estructura final de la implementación se puede ver en el gráfico de resumen (3.4).

3.3.1. Modelado

El componente *NoiseSimulator* genera, escribe y lee los distintos ruidos multi-frecuencia de Worley, los cuales son indispensables para crear la forma de las nubes. Llegado este punto es necesario apuntar: que cuando se hace referencia a la característica de multi-canal o multi-frecuencia, se busca destacar la necesidad de una gran cantidad de ruidos de distintas frecuencias. Para satisfacer esta demanda de manera eficaz, se utiliza una única textura para alojar el mayor numero de estos ruidos, asignando a cada uno de los cuatro canales disponibles. Este enfoque permite maximizar el uso del espacio de almacenamiento y optimizar la eficiencia

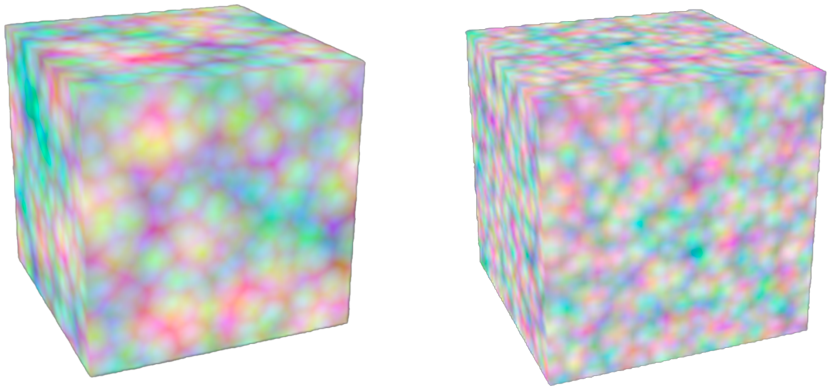

Figura 3.5: Representación gráfica de las texturas de Ruido de Volumen, a la izquierda, y de Detalle, a la derecha, utilizados en la demostración final.

del sistema.

Para la implementación del efecto requerido, es necesario contar con dos texturas tridimensionales de distinta resolución. La primera de ellas, o Ruido de Volumen, debe tener 4 canales (RGBA) y una resolución de 132^3 píxeles, y tendrá la función de modelar el aspecto final de las nubes. La segunda, o Ruido de Detalle, de 3 canales (RGB) con una resolución de 64^3 píxeles, será la encargada de conceder un mayor detalle a las nubes.

Es importante destacar que estas texturas presentan un ruido tridimensional, lo que implica que si se realiza un corte a este ruido, se encontraría una gran cantidad de información en su interior. En la figura (3.5), es posible observar el aspecto final de estos ruidos.

Estos ruidos de precisan además de una característica fundamental, y es que estos deben ser "tileables". O, en otras palabras, que si son colocados de forma contigua se genere la sensación de que forman parte de una misma textura. Esto se logra en el mismo momento del cálculo de la textura, ya que además de considerar la distancia a los puntos, se tiene en cuenta la proximidad a los puntos de todas las celdas vecinas. Nueve celdas en total si se trata de generar una textura en dos dimensiones, y veintisiete si se trata de tres dimensiones. Dado que se trata de la misma distribución de puntos desplazada, su cálculo es relativamente sencillo de calcular.

Es fundamental prestar atención a este aspecto, ya que un ruido inapropiado o uno sobre el cual no se pueden controlar suficientes parámetros puede generar una multitud de problemas y artefactos no deseados.

Por otra parte, el componente *GradientSimulator* es mucho mas sencillo de calcular. Sus labores son las de crear y superponer una serie de ruidos de Perlin los cuales posteriormente seran utilizados para modelar efectos clima de nuestra

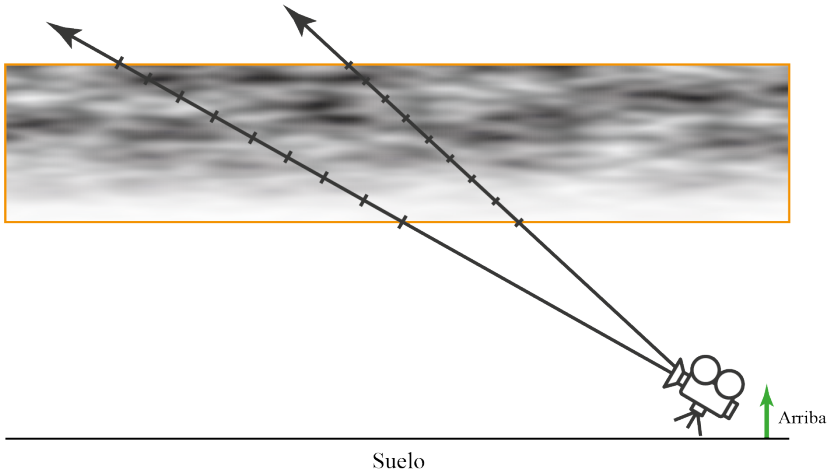

Figura 3.6: Representación del camino que realiza un rayo a través de la escena.

simulación. Esto es, Si lo deseamos podemos modelar zonas en las que no haya apenas nubes con otras en las que apenas veamos los rayos el sol de esta manera será posible. Este ruido no es necesario que sea tileable puesto que al ser un ruido procedimental el cual puede ser calculado para cualquier punto.

Los ruidos, tanto de Worley como de Perlin, se calculan mediante el uso de ComputeShaders, sin los cuales la realización de estos cálculos no sería factible. La capacidad de poder paralelizar esta tarea permite su ejecución en un tiempo casi imperceptible. Cualquier otra metodología para el cálculo de las texturas podría demandar un excesivo tiempo de ejecución. Es cierto que estas generaciones de ruido solo son producidas por el artista en el UnityEditor, por lo que la velocidad de creación no es un factor totalmente crítico. Pero también es cierto que durante la fase de personalización y parametrizado del VFX, resulta casi obligatorio que la creación y manipulación de estas texturas sea lo más rápida posible.

3.3.2. Renderizado

Una vez obtenidos los ruidos requeridos, se precisa implementar un método que modele y coloree las nubes usando las texturas generadas con anterioridad. Para cumplir este objetivo, se ha implementado una *ScriptableRendererFeature* específica para este efecto, encargada de representar las nubes en espacio de pantalla usando "*Raymarching*".

El *Ray Marching*, o en español 'marchado de rayos', es una técnica de renderizado ampliamente utilizada dentro del mundo de los gráficos por ordenador, a menudo usada cuando se necesita trabajar con objetos definidos matemáticamente, como fractales, o cuando se quieren crear efectos visuales que implican refracciones, reflejos y sombras.

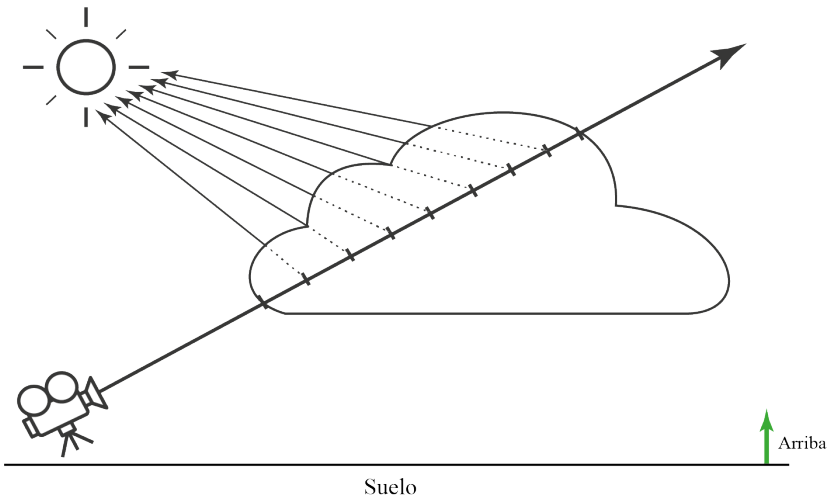

Figura 3.7: Representación de el sondeo de cada uno de los puntos sobre los que se ha marchado.

Este método utiliza una variante de la popular técnica "*Ray Tracing*", o en español 'trazado de rayos', pero en lugar de calcular las intersecciones exactas entre un rayo y un objeto, se utilizan técnicas de marcha para acercarse progresivamente a la intersección. El raymarching se basa en el concepto de campo de distancia firmada (SDF, por sus siglas en inglés *Signed Distance Function*), que es una función que, para cualquier punto en el espacio, devuelve la distancia más corta hasta la superficie de un objeto.

Mas o menos el funcionamiento de un shader de raymarching genérico contaría con los siguientes pasos:

1. **Emisión del Rayo**: Inicialmente, un rayo se lanza desde un punto origen, normalmente la cámara, en una dirección especifica.

2. **Medida de distancia**: Para cada rayo, se evalúa la función SDF en la posición actual del rayo para obtener la distancia mínima a la superficie del objeto más cercano.

3. **Marcha del rayo**: El rayo avanza una distancia igual al valor devuelto por la función SDF en el paso anterior. Dado que SDF proporciona la distancia mas corta para cualquier superficie, sabemos que podemos avanzar el rayo esta distancia sin pasar por alto ninguna superficie.

4. **Iteración**: Los pasos 2 y 3 se repiten hasta que el rayo esté lo suficientemente cerca de la superficie según algún umbral predefinido, o hasta que se alcance un número máximo de iteraciones.

5. **Iluminación y Sombreado**: Cuando un rayo golpea una superficie, se calcula la iluminación en ese punto utilizando algún modelo de iluminación

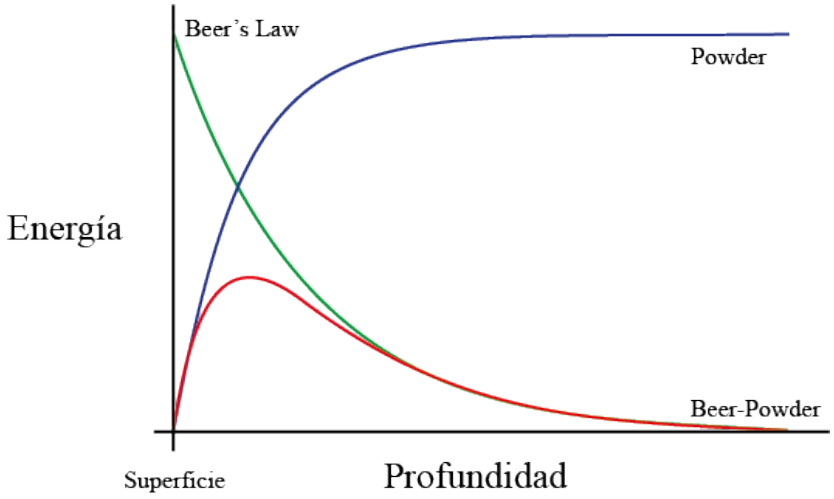

Figura 3.8: Representación de la Ley de Beer, el efecto polvo (*powder effect*), y la convolución de ambas.

específico previamente escogido.

Para esta implementación concreta (A.7 y A.8), se ha recurrido a una variación ligera de esta técnica. En vez de centrarse en la búsqueda de una intersección con una superficie sólida, el enfoque consiste en transitar a través de un volumen, recogiendo información relativa a la luz (A.6) y la densidad (A.4 y A.5) a lo largo del recorrido. Durante todo el trayecto del rayo, se acumula luz, que puede ser atenuada por la densidad del volumen, la absorción de luz en la nube, u otros parámetros artísticos (A.2). Esta progresión del rayo es detenida únicamente cuando la cantidad de luz acumulada resulta insignificante o una vez que se ha atravesado todo el volumen en cuestión.

3.3.3. Iluminación

Una vez determinado el volumen de la nube, es necesario especificar como se va a aproximar su iluminación. En la realidad, la luz que percibimos de una nube es el resultado de innumerables reflexiones y refracciones que ocurren dentro de la misma, donde las moléculas de agua interactúan con la luz. Este complejo proceso, si se intenta simular en su totalidad o incluso parcialmente, puede resultar en una tarea prácticamente inabordable debido a su complejidad.

Por tanto, se propone una técnica alternativa que consiste en trazar el camino inverso tomando las probabilidades de que se produzcan estos efectos. Este método involucra realizar otro 'ray marching', es decir, un muestreo de rayos, desde cada uno de los puntos que hemos sondeado dentro del volumen de la nube en dirección a la fuente principal de luz en la escena. De esta forma, pode-

Figura 3.9: Comparativa de resultados finales utilizando las distintas relaciones.

mos aproximar la iluminación que comúnmente atribuimos a estas formaciones atmosféricas.

Mas concretamente para aproximar esta luz es se usará la función de Henyey-Greenstein, definida en (3.1). Esta función es una forma de describir cómo se dispersa la luz en un medio, como un gas o una nube. En términos simples, la función de Henyey-Greenstein describe la probabilidad de que un fotón se disperse en un ángulo en particular cuando interactúa con una partícula en el medio. El resultado de la función es una distribución de probabilidad que depende de un parámetro llamado factor de asimetría g, tal que $-1 \leqslant g \leqslant 1$. Y de otro parámetro θ el cual representa el ángulo entre la dirección de la luz incidente y la dirección de la luz dispersada

$$HG(g,\theta) = \frac{1-g^2}{4\pi\sqrt{(1+g^2-2g\cos\theta)^3}} \tag{3.1}$$

En el fragmento de código (A.3), es posible observar una implementación similar de la fórmula mencionada en la demostración técnica. Cabe resaltar la facilidad de este enfoque, el cual permite introducir directamente el valor del coseno del ángulo ya calculado, aprovechando el hecho de que se están utilizando

vectores y que el producto vectorial es equivalente al coseno del ángulo formado por dos vectores.

Además del modelo de HenyeyGreenstein, es importante añadir la ley de Beer y efecto polvo. A fin de obtener un resultado mas natural y creíble para la iluminación final de las nubes.

$$Beer(d) = e^{-d} \tag{3.2}$$

$$Powder(d) = 1 - e^{-2d} \tag{3.3}$$

La Ley de Beer (3.2) modela que los rayos portan mas energía cuanto mas superficiales se encuentran (menor es su densidad). Mientras que el efecto polvo (3.3) relaciona lo contrario, como cuanto mayor es la densidad de polvo mayor es la probabilidad de que algún rayo que se encuentre rebotando en su interior acabe saliendo e impactando al espectador. En el gráfico (3.8) es posible apreciar ambas relaciones, y la convolución de ambas que finalmente ha sido implementada. Además, en la figura (3.9) se encuentra una comparativa de resultados visuales.

Desde una perspectiva visual, la nubes resultante posee una apariencia que puede asociarse al algodón de azúcar, con una base más amplia en comparación con su parte de arroba. Este efecto se logra mediante el uso de los gradientes de altura que son calculados en el *GradientSimulator*. Asimismo, la nube muestra una tonalidades más oscura en la zona opuesta a la fuente de luz, mientras que la parte orientada directamente hacia la fuente luminosa presenta un brillo intensificado.

3.4. Ciclo día noche

A la hora de implementar este efecto se ha optado por la siguiente estructura (3.10). En la cual podemos apreciar la presencia de dos scripts sobre los cuales recae la responsabilidad del dibujado del cielo “SkyboxController.cs”, sobre el cual cae la responsabilidad de actualizar los estados de las ditintas propiedades, y “Skybox.shader”, el cual es el encargado de realizar todos los cálculos de dibujado del cielo.

Además, debido a la falta de realismo en el cálculo de los colores del océano, fue necesario introducir un componente para calcular la “*Reflection Probe*”, o en español Sonda de Reflexión.

Las Sondas de Reflexión en Unity son un recurso muy eficaz para la generación de reflejos verosímiles en objetos tridimensionales. El procedimiento para

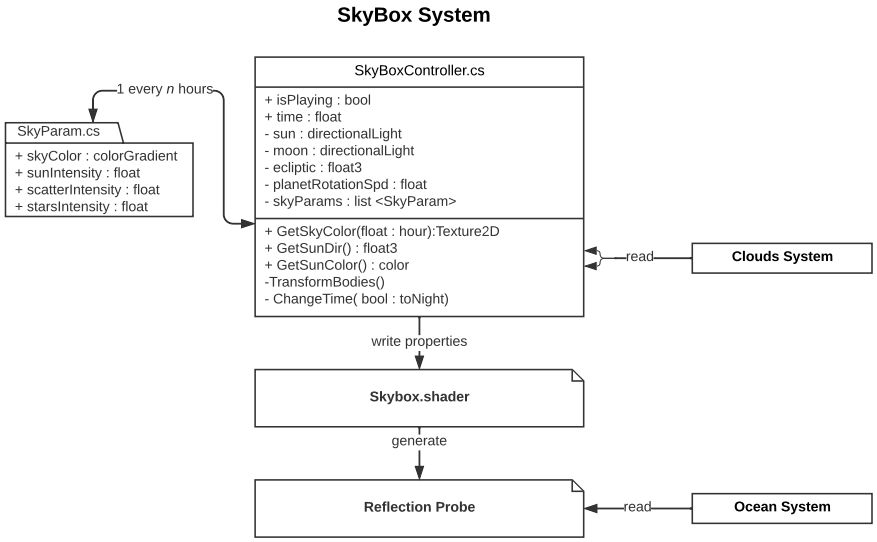

Figura 3.10: Diagrama resumen de la implementación del SkyBox.

su implementación se lleva a cabo mediante la creación de una representación gráfica esférica o cúbica del entorno escénico desde la perspectiva de la sonda. Posteriormente, esta imagen obtenida se aplica para el cálculo y la representación de los reflejos en la interfaz visual. En el caso de esta demostración técnica esta sonda es utilizada para calcular las reflexiones del océano y el barco.

Finalmente "SkyboxController.cs" realiza en CPU los cálculos necesarios para que el shader coloree el cielo con la gama de tonalidades correspondiente a la hora del día. Además de rotar las diferentes luces direccionales que representan a la luna y al sol. De esta forma estas propiedades se encuentran mas accesibles dando lugar a la capacidad de poder animarlas mas facilmente. Aunque podría moverse toda esta funcionalidad a su contraparte en la GPU "Skybox.shader".

Un aspecto interesante de este movimiento de los astros es la incorporación de la eclíptica, que es la línea que trazan los astros al cruzar el firmamento. La utilidad de esto no es mas que poder elegir el ángulo con el que inciden la luna y el sol sobre la escena.

Finalmente la implementación final de este shader esta disponible en el Apéndice (B) para consulta.

4

Resultados finales

4.1. Interfaz de usuario y controles

Finalmente y antes de analizar los resultados es importante que se introduzca la interfaz gráfica. El objetivo final de esta demostración técnica es la de mostrar una posible solución para representar estos efectos, por lo que se asume que el jugador final no va a usar mas de unos minutos para mirar alrededor de la escena. Aunque sea un tiempo muy escaso el jugador debe de ser capaz de poder cambiar las condiciones de esta y experimentar con los distintos cambios en la escena.

Por un lado el jugador debe de ser capaz de controlar el barco y obtener información de éste. Para lo cual se ha diseñado el siguiente panel (4.1). En el cual se sitúa en la esquina superior derecha de la aplicación, y puede controlar el timón y la orientación de la velas. A poder conocer rápidamente en que dirección incide el viento con respecto a la orientación del Barco.

Además, el jugador debe de ser capaz de controlar el paso del tiempo y el clima. Por lo que se ha decidido diseñar el panel (4.2) en el cual el jugador puede consultar la hora del día, y forzar distintos estados del clima.

Finalmente, además de poder configurar la escena de la manera en la que desee, el usuario debe de ser capaz de *navegar* por la escena libremente. Esto quiere decir, que se debe de diseñar un sistema híbrido en el cual se pueda; a ratos interactuar con los distintos elementos "*clicables*" de la interfaz, a ratos moverse con libertad por la escena. Para esto, tal como se muestra en la figura (4.3), cuando el jugador desee cambiar entre estos modos podrá hacerlo pulsando

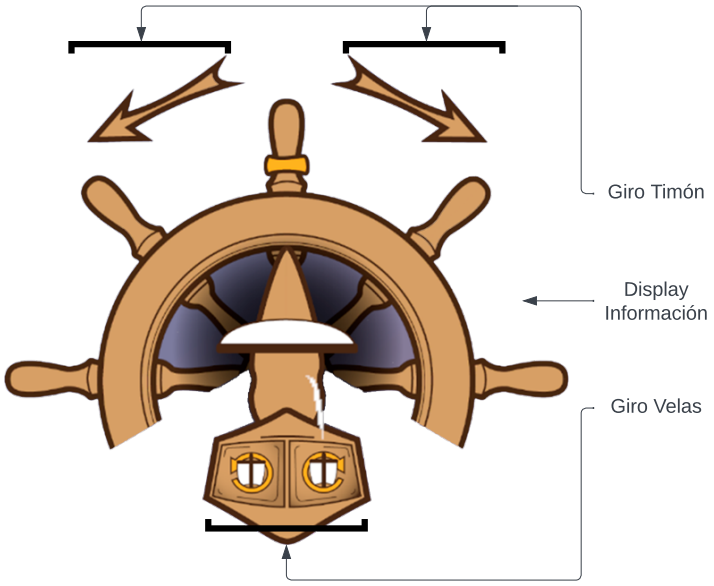

Figura 4.1: Interfaz de información y control de la nave.

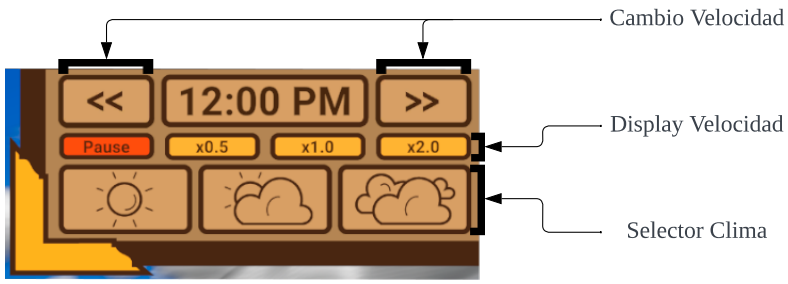

Figura 4.2: Interfaz de control del clima y el tiempo.

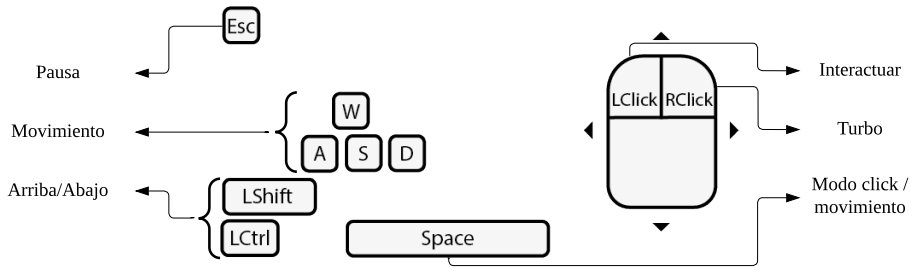

Figura 4.3: Diagrama de los controles.

la barra espaciadora. Por último, se ha incluido un simple menú pausa a través del cual salir de la aplicación a voluntad.

4.2. Océano

4.2.1. Flotabilidad

Para generar esta animación, se han desarrollado y ajustado los componentes presentados anteriormente (3.3). Los cuales, en conjunto, permiten la generación de un efecto de flotación en el mar. A continuación se expondrán los distintos parámetros que permiten ajustar los distintos comportamientos:

FloatingObject.cs

- **Simulate Flotation** (Booleano): El cual permite controlar cuando calcular la animación.
- **Flotating Points** (Lista de Vectores 3D): Puntos clave en los cuales se sondeará la textura de desplazamiento. A Mayor numero de puntos mejor la aproximación de la animación de flotado de los objetos grandes, pero esto también un incremento del tiempo de ejecución.

  Se ha observado una relación de aproximadamente 110 tics de reloj para el calculo de la normal de cada punto. En la demostración final se ha optado por implementar solo cuatro puntos, lo que hace que esta normal se calcule en un plazo de tiempo aproximado de $\sim$ 0,26 ms.
- **Displacement Stiffness** (Flotante): El cual permite ajustar la dureza con la que se requiere que se apliquen los desplazamientos en los ejes x y z.
- **Rotation Stiffness** (Flotante): El cual permite ajustar la dureza con la que se aplica la nueva orientacion del modelo.

NavigableObject.cs

Es el encargado de proporcionar movimiento a un objeto flotante en el mar. Al crear una escena y optar por incluir este componente en un objeto nuevo, se añade automáticamente un componente FloatingObject.cs, que es esencial para el adecuado funcionamiento. A continuación, se presentan los distintos parámetros que permiten ajustar el comportamiento de los objetos que incorporan este componente.

- **Direction** (Flotante): Toma valores entre -1 y 1. Describe la dirección del timón del barco.

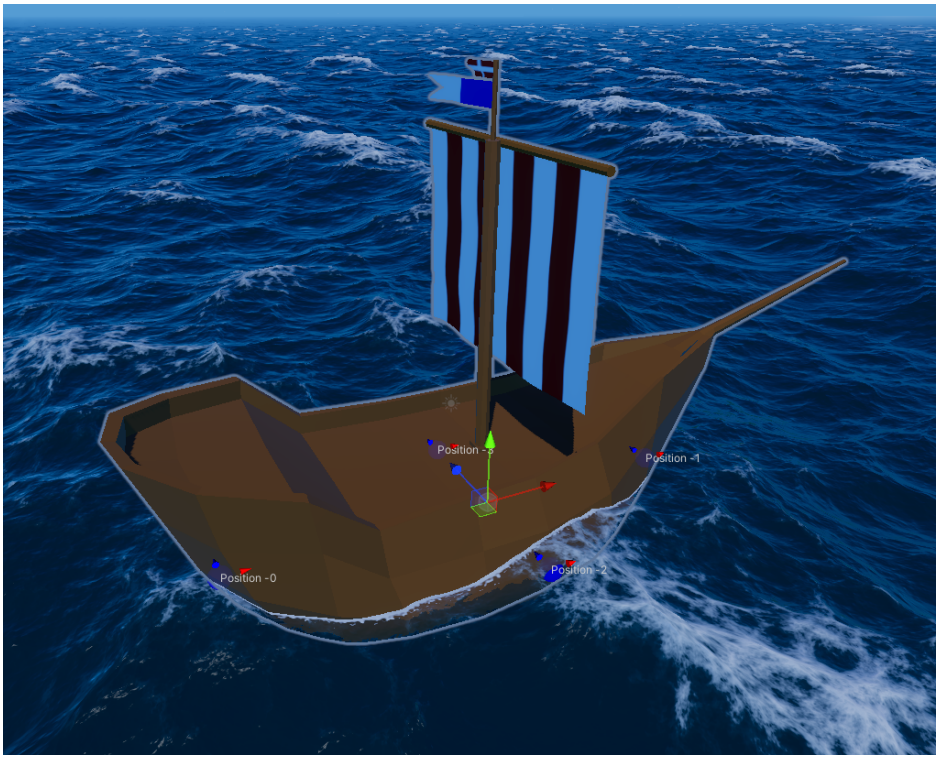

Figura 4.4: Vista desde el inspector, de el componente *FloatingObject.cs*.

- **Max. Turn Speed** (Flotante): Velocidad de giro máxima del barco.
- **Throttle** (Flotante): Toma valores entre -1 y 1. Describe la fuerza con la que el barco se mueve hacia delante.
- **Max. Speed** (Flotante): Velocidad máxima de la embarcación.

SailObject.cs

Por último, y debido a la naturaleza de la escena a representar, se ha implementado este componente. Con el objetivo de comunicarse con el componente *NavigableObject.cs* siguiendo la lógica que rige este tipo de embarcaciones. Este componente solo cuenta con dos parámetros:

- **Sails Direction** (Flotante): Toma valores entre -1 y 1. Describe la posición de las velas.
- **Max. Angle Sails** (Flotante): Configura el máximo ángulo de giro de las velas.

El resultado final del sistema es muy robusto y escalable. Permitiendo la animación y el comportamiento de casi cualquier tipo de embarcación un objeto que

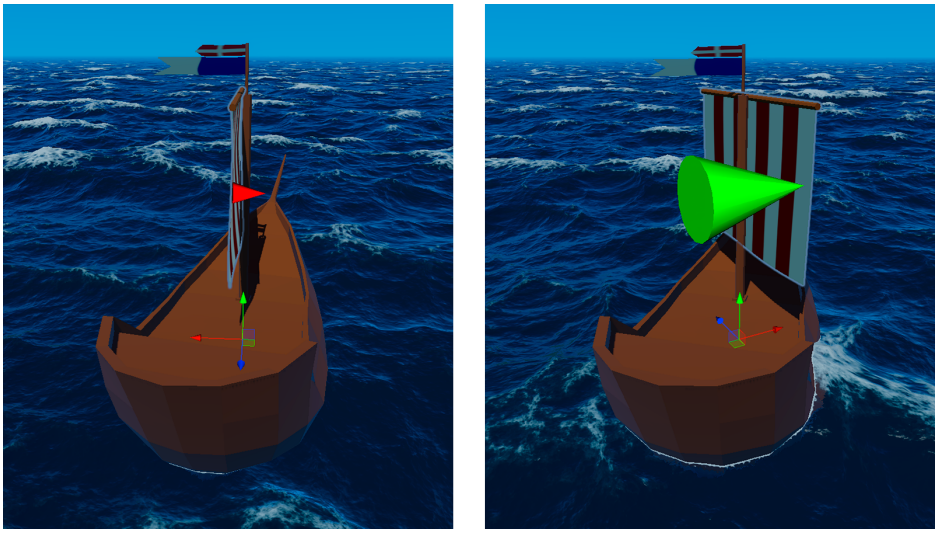

Figura 4.5: Visualización desde el inspector de el componente *SailObject.cs*.

se requiera que interactúe con el océano. De esta forma, por ejemplo, si se quisiera implementar una embarcación a motor, se podría usar la funcionalidad ya implementada y tratar de crear un componente que interactúe con *NavigableObject.cs* siguiendo la lógica que se precise.

4.2.2. Animación de Velas y Banderas

Con respecto a la simulación del movimiento de las distintas telas en escena, se ha optado finalmente por usar un Shader para generar este efecto. Además de esta manera también se aprovecha para dotar a estas de un texturizado personalizado. El Shader final cuenta con los siguientes parámetros:

- **TimeStep** (Flotante): Configura la velocidad de la animación.
- **TimeOffset** (Flotante): Es posible que si se decide usar varias de estas banderas de forma que estén muy cerca, se genere un efecto similar en todas ellas. Con este parametro podemos alterar esta similaridad, avanzando o retrasando en el tiempo algunas de ellas.
- **WaveHeight** (Flotante): Permite controlar la altura en la ondulación de la malla.
- **WaveSpeed** (Flotante): Controla la velocidad de esta ondulación.
- **WaveSize** (Flotante): Permite controlar en numero de ondulaciones presentes en la malla.

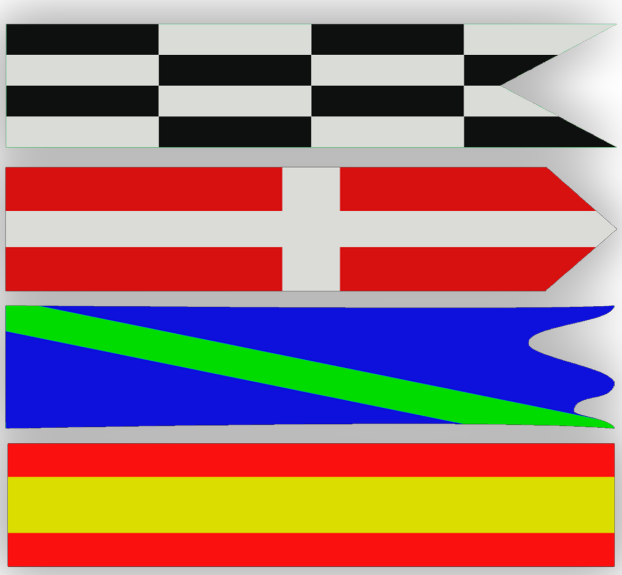

Figura 4.6: Algunas configuraciones posibles de el Shader de telas.

- **GradientWeight** (Vector 3D): Para generar la sensación de deformación de la tela, se ha añadido un ruido de gradientes procedimental que añade un aspecto mas aleatorio a la malla resultante. Este parámetro, permite controlar el peso de este ruido en los tres ejes locales. Permitiendo así utilizar el mismo shader en telas y banderas.

- **Alpha** (Textura): Normalmente las velas suelen ser rectangulares, aunque pueden presentar formas distintas. Es por esto que podemos seleccionar una textura en escala de grises para poder personalizar la forma de la bandera. Incluso pudiendo simular rotos o agujeros.

- **Pattern** (Textura): Las banderas pueden contar con multitud de patrones y diseños. Esta textura permitirá dibujar patrones de 2 colores sobre las telas a animar.

- **ColorA** (Color): Color principal de la vela.

- **ColorB** (Color): Color secundario de la vela.

El resultado final de este shader es bastante bueno. Pues cumple muy bien sus dos principales cometidos: Otorgar un movimiento aleatorio pero familiar para el movimiento de estos objetos en el medio natural, además de ser un efecto muy barato para el tan buen resultado que ofrece. Además de que este podría ser fácilmente ampliado para poder usar patrones con mas colores.

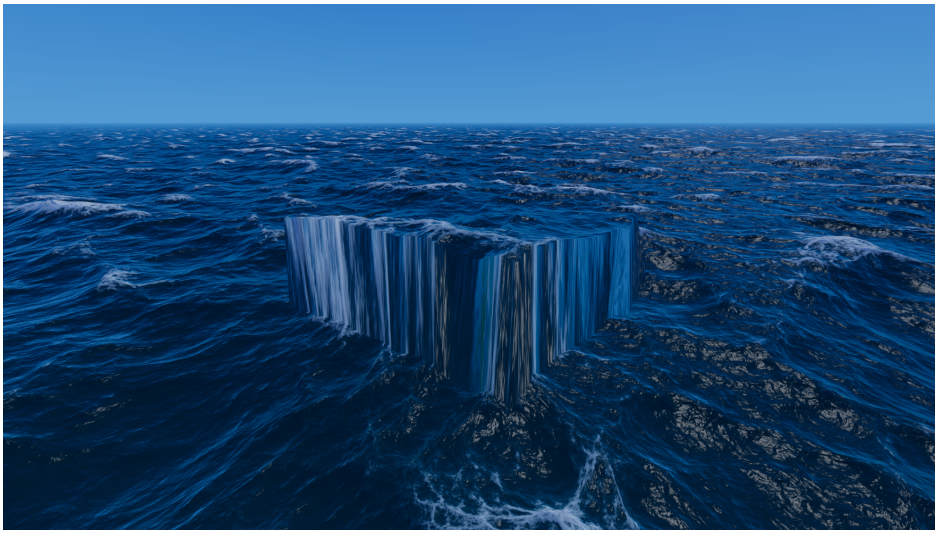

Figura 4.7: Resultado aplicación de un desplazamiento sobre la malla resultante.

4.2.3. Investigaciones y pruebas

Durante el desarrollo de la demostración técnica se ha tratado de poner mucho énfasis en la interactividad de los distintos efectos. Así es como en un determinado momento del desarrollo se optó por tratar de generar algún tipo de rastro cuando un barco surcara su superficie. Para ello, en primer lugar. se trató de aplicar una textura animada que desplazara la masa de agua al paso del barco.

Esta aproximación no es posible implementarla de forma inmediata debido a que como se vio en el apartado (2.2.3) las normales de la malla son calculadas de forma procedimental a través de FFT. Por lo que el resultado que ofrece aplicar directamente un desplazamiento sobre las malla resulta mas parecido algún tipo de artefacto visual (4.7), que el efecto que se quiere aproximar. Una posible solución sería la de volver a calcular las normales de la malla en los triángulos cuyos puntos se hayan visto desplazados.

Otra posible solución podría ser la de no desplazar la malla y solo añadir un "*decal*", o pegatina, la malla (4.8). La cual genere la espuma que suelen crear las embarcaciones al ir abriéndose paso por una masa de agua. Al igual que el acercamiento anterior esta opción fue desestimada debido a la complejidad técnica necesaria para implementar esta característica lo suficientemente bien.

En caso de tener que implementar esta característica por necesidades del proyecto u otros requerimientos; la opción que quizás se acerque mas al resultado esperado sea una conjunción de ambas técnicas. Pero requeriría mucha atención a la creación de ambas texturas animadas, ya que estas trasformaciones carecen de inercia propia y se moverían junto con el barco en giros y virajes bruscos producidos de el oleaje.

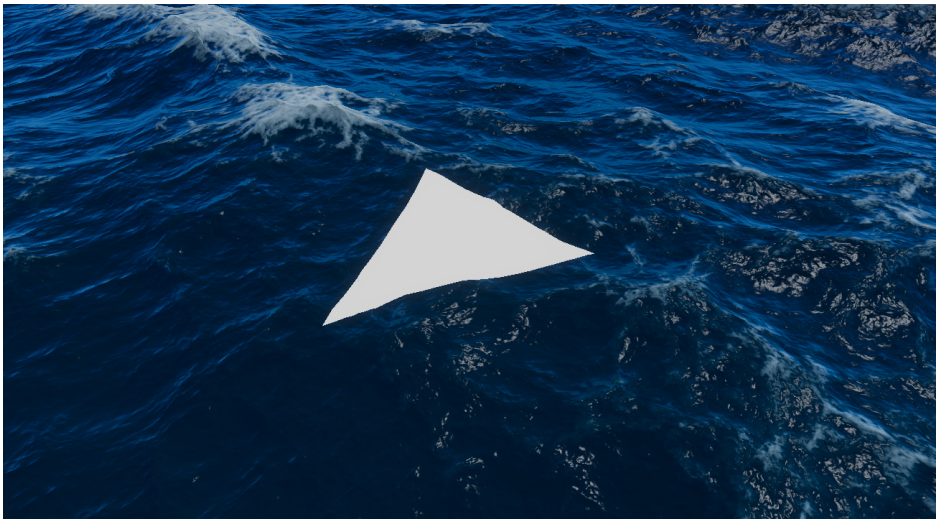

Figura 4.8: Aplicación de un "*decal*" con forma triangular sobre la malla.

4.3. Nubes

Finalmente para generar el efecto de las nubes volumétricas se ha optado por crear tres componentes diferentes, encargados cada uno de las distintas tareas necesarias para la creación de este efecto. Estos son:

4.3.1. Noise Simulator

Encargado de crear, leer y guardar las texturas 3D de los distintos ruidos de Worley necesarios para generar el efecto de las nubes volumétricas. Su principal utilidad se encuentra en ofrecer al artista la capacidad de generar un ruido que genere una nubosidad que se ajuste suficientemente bien a las necesidades en escena. Fuera de el editor, en la aplicacion final, este componente solo se limita a leer de memoria las texturas generadas con anterioridad y suministrarselas al shader. Este componente cuenta con gran numero de parámetros los cuales permiten controlar el aspecto final de estas texturas, algunos de ellos son:

- **Active Tex. Type** (Volume/Detail): Permite designar sobre que textura se quiere trabajar.
- **Active Channel** (R/G/B/A): Marca sobre que canal de la textura se quiere trabajar. Ya que en la textura de detalle solo se usan tres canales, el sistema detecta cuando se está queriendo escribir sobre ese canal y corre un Kernel el cual configura todo el canal con unos.
- **Volume Resolution** (Entero): Permite configurar la resolución del Ruido de Volumen.

- **Detail Resolution** (Entero): Permite configurar la resolución del Ruido de Detalle..
- **Volume/Detail Noise Params** (Lista de WorleyParams): Debe haber uno por cada uno de los canales a generar. Permiten guardar parámetros distintos para cada uno de los distintos canales de la textura.

El funcionamiento de este componente es, básicamente, correr a voluntad un Shader de Cómputo, o Kernel, el cual genere y guarde un ruido de Worley en el canal seleccionado de la textura seleccionada.

El rendimiento de este componente no es crítico para el bucle principal de juego, puesto que el grueso de su utilidad tiene lugar en el Editor. Pero aun así presenta una gran velocidad a la hora de generar y guardar estas texturas. Así es como para el ruido de Volumen, el cual cuenta con una resolución de 132^3 píxeles y ocupa una memoria total de $17,5$ MB, Tarda en generarse unos $\sim$ 11ms.

Para la generación del ruido de detalle, se observa que el tiempo requerido es aproximadamente el mismo que para el ruido anterior. Esto podría deberse a que, aunque el ruido de detalle tenga un tamaño significativamente menor (con 64^3 píxeles y 2 MB), el uso de la tarjeta gráfica para calcular estas texturas de forma paralela podría conllevar un tiempo de retardo adicional en el envío y recepción de la información. En este caso estaríamos muy próximos al tiempo que tarda la informacion en viajar de un punto al otro.

4.3.2. Gradient Simulator

Similar al componente anterior, este se encarga de generar el ruido de Gradientes, también conocido como ruido de Perlin, que es necesario para el renderizado. Dada la naturaleza de este componente, que busca modelar diversos climas sobre las nubes, la textura no se almacena para ser leída posteriormente, sino que puede ser creada "*on-the-fly*" según se desee dentro de la aplicación final, con el fin de simular, por ejemplo, el movimiento de un frente borrascoso o una zona con cielos despejados. Este componente permite la personalización y creación de la textura de ruido mediante los siguientes parámetros:

- **Resolution** (Entero): Permite configurar la resolución final.
- **Scale** (Flotante): Permite controlar el tamaño del ruido final.
- **Num Layers** (Entero): Numero de capas totales que compondrán la textura final.
- **Persistence** (Flotante): Controla cuánto contribuyen las frecuencias más altas (o detalle fino) al ruido total. Una persistencia alta significa que las

frecuencias más altas tendrán una gran influencia, lo que da como resultado un ruido más rugoso o accidentado.

- **Lacunarity** (Flotante): Controla cuán rápido aumentan las frecuencias de cada octava. Esto afecta la escala de los detalles en el ruido. Con una lacunaridad alta, los detalles serán más pequeños y más cercanos entre sí, mientras que con una "lacunaridad" baja, los detalles serán más grandes y más espaciados.

- **Offset** (Vector 2D): Permite el movimiento del ruido generado.

Aunque la intención sea la de ir calculando distintos ruidos a medida que avanza la aplicación. Finalmente en la demostración técnica solo tiene lugar un cálculo de esta textura al principio, debido a que se decidió no explorar esta posibilidad.

Es componente toma un tiempo aproximado de $\sim$ 15 ms para calcular una textura de 512^2 píxeles y que ocupa 2 MB de memoria.

4.3.3. Cloud Simulator

Finalmente, y para controlar todos los parámetros artísticos del Shader encargado de generar el efecto de las nubes, se ha implementado un componente encargado de los siguientes parámetros:

- **Cloud Scale** (Flotante): Controla el tamaño dentro de la simulación.

- **Density Threshold** (Flotante): Ajusta el umbral de densidad del ruido de volumen. Línea 15 del Anexo (A.4).

- **Density Multiplier** (Flotante): Ajusta la influencia del ruido de detalle en la densidad final de las nubes. Línea 26 del Anexo (A.4).

- **Container Edge Fade** (Flotante): Delimita cuanto de cerca al borde pueden formarse las nubes.

- **Volume/Detail Noise Weights** (Vector 4D/Vector 3D): Guardan el peso de cada uno de los canales de cada uno de los ruidos a la contribución final.

- **Num Steps Lightning** (Entero): Numero de pasos en los que se va a dividir el sondeo de iluminación.

- **Fixed Step Modeling** (Flotante): Tamaño de paso de sondeo de los ruidos que modelan el efecto.

Figura 4.9: Captura de pantalla de el Resultado Final de las Nubes.

- **Light Absortion Sun** (Flotante): Cuánta luz del sol es absorbida por las nubes. Entra en acción en la función *lightAtPos()*, línea 29 del Anexo (A.6).
- **Light Absortion Sun** (Flotante): Se usa para aproximar cuanta luz en general es absorbida mientras viaja a través de las nubes. Dentro del bucle principal de Raymarching, línea 19 del Anexo (A.8).
- **Darkness Threshold** (Flotante): Delimita el umbral de oscuridad a la hora de iluminar el lado oscuro de las nubes.
- **Fordward/Back Scattering** (Flotantes): Son usados como el parámetro g para la función Henyey-Greenstein (3.1), modela la dispersión hacia adelante y hacia atrás de la luz en las nubes.
- **Phase Factor** (Flotante): Peso aplicado al resultado combinado de las dos funciones Henyey-Greenstein.
- **Bare Brightness** (Flotante): Permite iluminar u oscurecer las nubes resultantes.

4.3.4. Efecto resultante

El resultado final se ajusta bastante a las necesidades de la escena. Puesto que es posible visualizarlas como un elemento mas de la escena el cual ocupa un volumen. Además debido a que los ruidos necesarios para ejecutar esta técnica son calculados con aterioridad se convierte en un efecto bastante barato para lo que ofrece.

El Shader en espacio de pantalla “Clouds.shader” toma del orden de $\sim$ 9,4ms en dibujar las nubes en la escena final, lo que supone un 70,7 % del tiempo

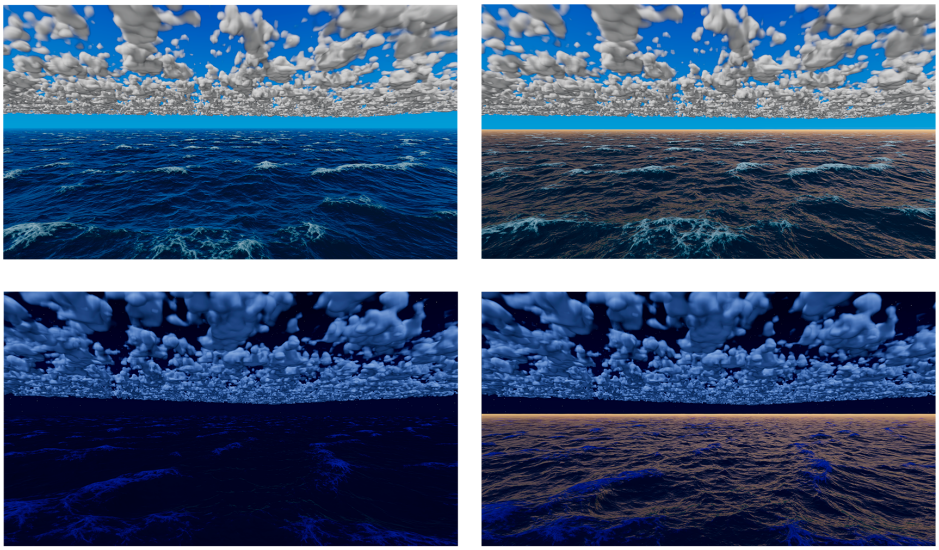

Figura 4.10: Comparativa de la escena con el uso o no de la Sonda de Reflexión.

de uso de la GPU. Aunque es opción muy rentable en términos de tiempo de computación, por supuesto cuenta con sus propias limitaciones y peculiaridades:

- Problemas con direcciones de luz principal muy perpendiculares. Debido a peculiaridades a la hora de medir la distancia recorrida dentro de la caja, este algoritmo presenta un punto débil muy importante en la linea 13 del Anexo (A.6). Ya que al pasar el inverso de la dirección de la luz es posible que para valores de coordenadas próximos al cero se generen extraños.

  En la aplicación final se ha optado por introducir un control sobre el valor de estas coordenadas (lineas 5-9, Anexo A.6). El cual minimiza, aunque no elimina, este efecto en la iluminación de las nubes.

- Problemas con el tamaño. Pata determinados tamaños de simulación es posible que este modelo no se ajuste todo lo bien que pudiera. Se ha observado que volúmenes pequeños pueden aguantar gran numero de pasos ofreciendo mucha calidad. En la aplicación final las nubes generadas son lo suficientemente buenas, pero no debe andar mucho mas lejos el limite de capacidad.

4.4. Skybox

Finalmente y para representar el efecto de paso del tiempo en la escena se han implementado los componentes anteriormente mencionados en el diagrama

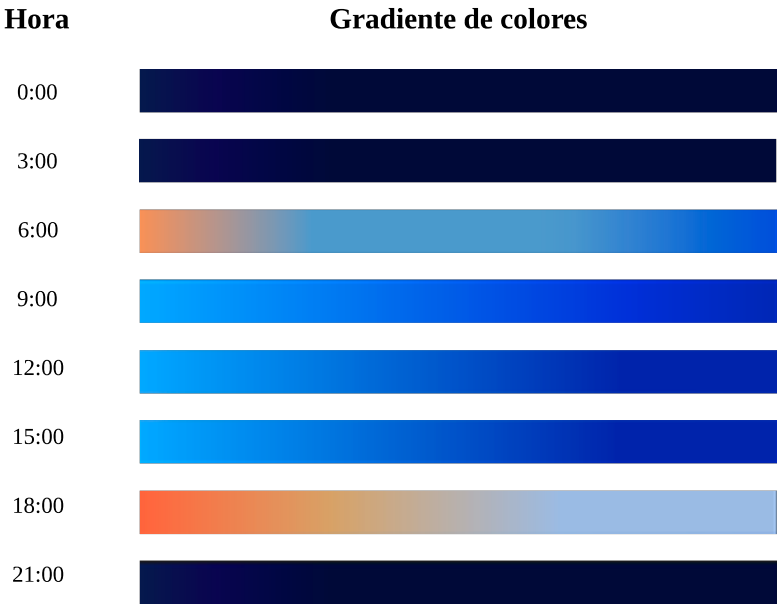

Figura 4.11: Paleta de colores según la hora del día.

(3.10). El objetivo de SkyboxController.cs, es el de controlar de manera accesible las distintas variables que configuran la SkyBox. Mientras que SkyBox.shader es el encargado de calcular dicha SkyBox aprovechándose de las propiedades de la gráfica para este tipo de procesamientos. Tras este calculo, una vez cada que se pinta un cuadro, el componente ReflectionProbe genera una Sonda de Reflexión para que sea utilizada posteriormente por los distintos elementos en escena. Es importante que sea actualizada a voluntad vía código, puesto que calcular dicha sonda es un proceso que puede ralentizar mucho la aplicación. Para que de esta forma cuando se encuentre pausado el tiempo en escena no es necesario que se siga calculando la sonda.

Este sistema es altamente adaptable a las necesidades del usuario y ofrece los siguientes parámetros de configuración:

- **Is Playing** (Booleano): Controla el paso o no del tiempo.
- **Time** (Flotante): Numero que representa la hora del día en Sistema decimal.
- **Sun / Moon** (Luz Direccional): Luces direccionales de la escena.
- **Ecliptic** (Vector 3D): Configura el eje de rotación de la tierra.
- **Planet Rotation Speed** (Float): Controla el tamaño de paso del paso del tiempo.

- **Sky Params** (Lista de SkyParams): Lista de ScriptableObjects, a partir de los cuales se calculará el estado del cielo en un instante determinado. Para esta implementación se ha optado por una división de un SkyParam cada tres horas (4.13).

El funcionamiento de este sistema es muy sencillo puesto que, básicamente:

1. SkyboxController avanza la hora conforme a un tamaño de paso fijado.
2. Comprueba que no se produzca desborde (Que no se alcance la hora 25).
3. Rota las distintas luces en escena atendiendo al eje fijado.
4. Calcula el gradiente de color y demás parámetros correspondiente a ese instante de día interpolando entre los SkyParams pertenecientes a las dos horas mas cercanas.
5. Coloca todas estas propiedades calculadas en el shader para calcular la Skybox.
6. Por ultimo, manda la orden al componente ReflectionProbe para calcular dicha sonda.

Todos estos pasos son ejecutados en la CPU en un tiempo aproximado de $\sim 2,1$ms, mientras que para calcular dicho Skybox solo $\sim 0,1$ms.Es posible que moviendo parte de la lógica de trasformación o interpolación a la gráfica se experimente una mejora significativa en este aspecto. Aunque sin duda la parte de este sistema que mas recuros precisa es la parte asociada a la creación a voluntad de la Sonda de Reflexión. Ya que aproximadamente toma $\sim$ 8ms en crearse, y actualizar toda la iluminación global (*DynamicGI.UpdateEnvironment()*).

Una posible solución para esto sería la de tratar de generar la textura CubeMap procedimentalmente, y evitar el uso del componente Refelction Probe.

Figura 4.12: Captura de pantalla diurna de la demo final.

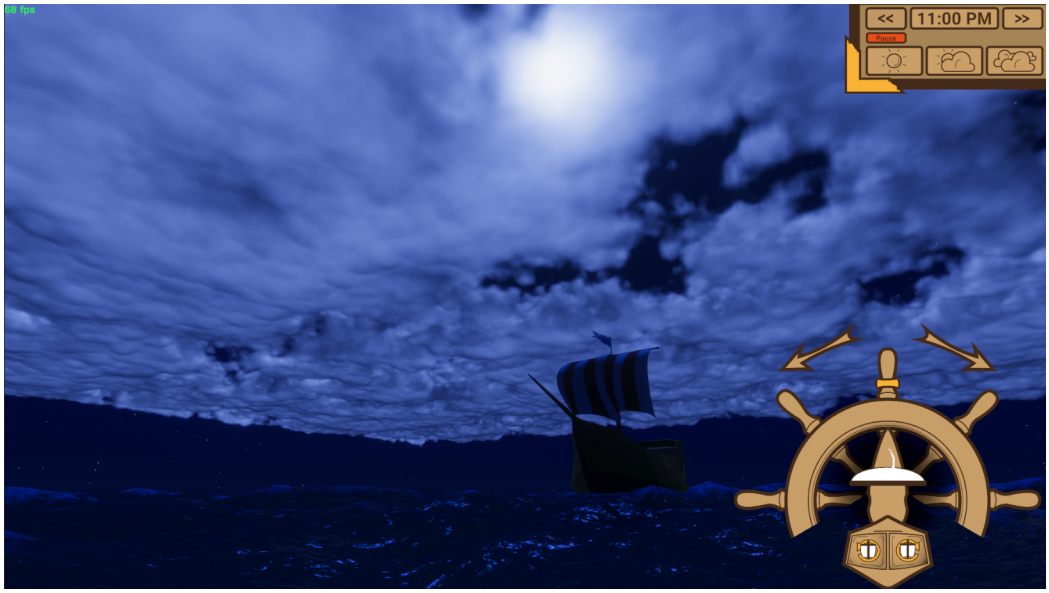

Figura 4.13: Captura de pantalla nocturna de la demo final.

5

Conclusiones y trabajos futuros

"*Las obras de arte no se terminan, se abandonan*", Leonardo da Vinci.

Bajo esta premisa, el presente estudio se asume como un principio más que un destino final. Las conclusiones alcanzadas aquí no deben entenderse como absolutas. Sino como progresos, hacia una comprensión más profunda de las técnicas implicadas en la creación de efectos visuales en los videojuegos.

Al igual que ocurre en todas las disciplinas artísticas, la práctica y la experimentación son indispensables en la búsqueda de la excelencia. La creación de efectos visuales para videojuegos, por tanto, no es una excepción. Ya que no solo se requiere un entendimiento profundo de las herramientas y técnicas, sino que también del modo en que estas se entrelazan para configurar una experiencia coherente y atractiva para el jugador.

Se requiere que estos artistas e ingenieros, como lo fue Da Vinci, posean las habilidades y conocimientos necesarios para producir obras bellas que sean capaces de cautivar al espectador. Este nivel de maestría únicamente se puede lograr mediante la constante práctica y estudio. Ya que constantemente se está innovando en búsqueda de métodos mas ingeniosos y realistas de realizar los mismos efectos.

De esta forma se aprecian las siguiente potenciales áreas de continuación:

1. Emisor de partículas. Aunque el sistema de Oceano contaba ya con un sistema muy complejo y elaborado de renderizado de la espuma del mar. Podría resultar muy beneficioso para la escena final la inclusión de algún tipo de

emisor de partículas en las crestas del mar mas cercanas al jugador. Este efecto se podría lograr incidiendo en el Shader que genera dicha espuma.

2. 'Stencil' del mar. A menudo, y sobretodo en mares muy bravos, es posible que las olas se rendericen a través del casco del barco generando un efecto extraño e indeseado. Para abordar esta mejora, no es posible usar el Stencil tal cual, ya que esto implica que siempre se va a renderizar el barco por encima del mar. Por lo que necesitamos, o bien marcar los vértices de la malla con aquellos que pertenecen a la parte de la cubierta del barco, o bien separamos la cubierta en otra malla ajena a el casco.

3. LLuvia/Tormentas. Es posible enriquecer la representación climática de la demostración técnica mediante la representación de lluvia en pantalla y rayos en escena. Para ello podría dotarse de mayor complejidad al GradientGenerator, para que determinados valores de este ruido correspondan a nubes en las que está lloviendo. O incluso sería posible tratar de cambiar este GradientGenerator a fin de que usara otro ruido procedimental sobre el que simular frentes anticiclónicos y borrascosos. Aunque esto genera retos derivados como la interactividad de las gotas de la lluvia con el mar y la bravura del mismo en esa zona.

4. Sonido. Un entorno no solo está compuesto por los elementos visibles. Puesto que para mantener la coherencia en la escena y poder llegar a hacer sentir la mayor sensación de inmersión, el sonido juega un papel fundamental. Esta demo-técnica no cuenta con audio ya que no es un tema en el que se haya puesto atención, pero sin duda un sistema encargado de cambiar el sonido ambiente cuando el jugador se sumerja o cuando vuela sin duda generaría una escena mas rica.

Bibliografía

[1] J. Tessendorf, "Simulating ocean water," *Simulating nature: realistic and interactive techniques. SIGGRAPH*, vol. 1, no. 2, pp. 8–13, 2001.

[2] C. J. Horvath, "Empirical directional wave spectra for computer graphics," in *Proceedings of the 2015 Symposium on Digital Production*, ser. DigiPro '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 29–39. [Online]. Available: https://doi.org/10.1145/2791261.2791267

[3] A. Schneider, "The real-time volumetric cloudscapes of horizon: Zero dawn," in *SIGGRAPH'15*, 2015. [Online]. Available: https://www.schneidervfx.com/

[4] K. Perlin, "An image synthesizer," in *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '85. New York, NY, USA: Association for Computing Machinery, 1985, p. 287–296. [Online]. Available: https://doi.org/10.1145/325334.325247

[5] S. Worley, "A cellular texture basis function," in *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 291–294. [Online]. Available: https://doi.org/10.1145/237170.237267

[6] I. Pensionerov, "Fft-ocean," 2022. [Online]. Available: https://github.com/gasgiant/FFT-Ocean

Apéndices

A

Código Nubes Volumétricas

Código A.1: Primera parte del shader de Nubes

```
Shader "Clouds"
{
    Properties
    {
    }
    SubShader
    {
        Cull Off ZWrite Off ZTest Always
        Pass
        {


            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            #include "UnityCG.cginc"
            #include "Lighting.cginc"

            struct appdata
            {
                float4 vertex : POSITION;
                float2 uv : TEXCOORD0;
            };

            struct v2f
            {
                float4 pos : SV_POSITION;
                float2 uv : TEXCOORD0;
                float3 viewVector : TEXCOORD1;
            };

            v2f vert (appdata v)
            {
                v2f output;
                output.pos = UnityObjectToClipPos(v.vertex);
                output.uv = v.uv;
                float3 viewVector = mul(
                   unity_CameraInvProjection, float4(v.uv * 2 -
                    1, 0, -1));
                output.viewVector = mul(unity_CameraToWorld,
                   float4(viewVector, 0));
                return output;
            }
```

Código A.2: Segmento de variables del shader de Nubes

```
    sampler2D _MainTex;
    sampler2D _CameraDepthTexture;

    //Propiedades de la luz principal
    float3 _SunDir;
    float4 _SunColor;

    //Ruido de Volumen (Voronoi)
    Texture3D < float4 > VolumeNoise;
    SamplerState samplerVolumeNoise;

    //Ruido de Detalle (Voronoi)
    Texture3D < float4 > DetailNoise;
    SamplerState samplerDetailNoise;

    //Ruido de Clima (Perlin)
    Texture2D < float4 > GradientNoise;
    SamplerState samplerGradientNoise;

    //Propiedades del contenedor
    float3 _BoundsMin;
    float3 _BoundsMax;

    //Parametros artisticos de las nubes
    float4 _VolumeNoiseWeight;
    float3 _DetailNoiseWeight;}
    float3 _CloudOffset;
    float _CloudScale;
    float _DensityThreshold;
    float _DensityMultiplier;
    float _DetailNoiseMultiplier;
    float _ContainerEdgeFadeDst;
    float _FixedStepSize;
    float _LightAbsortionSun;
    float _LightAbsortionCloud;
    float _DarknessThreshold;
    float4 _PhaseParams;
    int _NumStepsLight;

    //Constantes
    #define UNITY_E 2.71828182845904523536028747135266249
```

Código A.3: Métodos auxiliares del shader de Nubes

```
    // Henyey - Greenstein
    float hg(float cosTheta, float g)
    {
        float g2 = g * g;
        float denom = 1 + g2 - 2 * g * cosTheta;
        return (1 - g2) / (4 * UNITY_PI * pow(denom, 1.5f));
    }

    float powder( float d)
    {
        return 1.0f - exp(-d * 2);
    }

    float phase(float alpha)
    {
        const float blendingFactor = .5f;
        float a = 1 - blendingFactor;
        float b = blendingFactor;
        float hg1 = hg(alpha, _PhaseParams.x);
        float hg2 = hg(alpha, -_PhaseParams.y);
        float blendedHenyeyGreenstein = (hg1 * a) + (hg2 * b);
        return _PhaseParams.z + (blendedHenyeyGreenstein *
            _PhaseParams.w);
    }

    float remap(float v, float minOld, float maxOld, float
        minNew, float maxNew)
    {
        return minNew + (v - minOld) * (maxNew - minNew) / (
            maxOld - minOld);
    }

    float2 squareUV(float2 uv)
    {
        float2 screenParams = _ScreenParams / 1000.0;
        return float2(uv.x * screenParams.x, uv.y *
            screenParams.y);
    }
```

Código A.4: Medida de distancias dentro del contenedor en el shader de Nubes y primera parte del muestreo de densidad.

```
1     float2 rayBoxDistance(float3 rayOrigin, float3 rayDir,
          float3 boxMin, float3 boxMax)
2     {
3         float3 tmin = (boxMin - rayOrigin) / rayDir;
4         float3 tmax = (boxMax - rayOrigin) / rayDir;
5         float3 t1 = min(tmin, tmax);
6         float3 t2 = max(tmin, tmax);
7
8         float dstA = max(max(t1.x, t1.y), t1.z);
9         float dstB = min(t2.x, min(t2.y, t2.z));
10
11        float dstToBox = max(0, dstA);
12        float dstInsideBox = max(0, dstB - dstToBox);
13
14        return float2(dstToBox, dstInsideBox);
15    }
16
17     float sampleDensity(float3 position)
18    {
19        // Constantes
20        const float3 size = _BoundsMax - _BoundsMin;
21        const float edgeFadeDivisor = 1.0f /
              _ContainerEdgeFadeDst;
22        const float maxDimension = max(size.x, size.z);
23        const float3 halfSize = size * 0.5f;
24
25        // Calcular la posicion de la muestra
26        float3 samplePosition = (halfSize + position) *
              _CloudScale * 0.001f;
27
28        // Calcular la distancia a los bordes
29        float distanceFromEdgeX = min(_ContainerEdgeFadeDst,
              min(position.x - _BoundsMin.x, _BoundsMax.x -
              position.x));
30        float distanceFromEdgeZ = min(_ContainerEdgeFadeDst,
              min(position.z - _BoundsMin.z, _BoundsMax.z -
              position.z));
31
32        // Ponderacion del borde
33        float edgeWeight = min(distanceFromEdgeZ,
              distanceFromEdgeX) * edgeFadeDivisor;
```

Código A.5: Muestreo de la densidad dada una posición

```
1
2         // Calcular el gradiente
3         float2 gradientUV = (halfSize.xy + (position.xz - (
              _BoundsMin + _BoundsMax * 0.5f))) / maxDimension;
4         float gradient = GradientNoise.SampleLevel(
              samplerGradientNoise, gradientUV, 0).x;
5         float gradientMin = remap(gradient, 0, 1, 0.1f, 0.5f);
6         float gradientMax = remap(gradient, 0, 1, gradientMin,
              0.9f);
7         float heightPercent = (position.y - _BoundsMin.y) /
              size.y;
8         float heightGradient = saturate(remap(heightPercent,
              0.0f, gradientMin, 0, 1)) * saturate(remap(
              heightPercent, 1, gradientMax, 0, 1));
9         heightGradient *= edgeWeight;
10
11        // Muestrear el ruido de volumen
12        float4 volumeNoise = VolumeNoise.SampleLevel(
              samplerVolumeNoise, samplePosition + _CloudOffset *
              0.1f, 0);
13        float4 volumeWeight = _VolumeNoiseWeight / dot(
              _VolumeNoiseWeight, 1);
14        float volumeFBM = dot(volumeNoise, volumeWeight) *
              heightGradient;
15        float volumeDensity = volumeFBM + _DensityThreshold *
              0.1f;
16
17        // Muestrear el ruido de detalle y calcular la densidad
              final
18        if (volumeDensity > 0)
19        {
20            float4 detailNoise = DetailNoise.SampleLevel(
                  samplerDetailNoise, samplePosition +
                  _CloudOffset * 0.2f, 0);
21            float3 detailWeight = _DetailNoiseWeight / dot(
                  _DetailNoiseWeight, 1);
22            float detailFBM = dot(detailNoise, detailWeight);
23            float detailErode = (1 - volumeFBM) * (1 -
                  volumeFBM) * (1 - volumeFBM);
24            float detailDensity = (1 - detailFBM) * detailErode
                   * _DetailNoiseMultiplier;
25
26            return volumeDensity - detailDensity *
                  _DensityMultiplier;
27        }
28
29        return 0;
30    }
```

Código A.6: Método encargado de calcular la luz correspondiente a una posición dada.

```
    float lightAtPos(float3 pos)
    {
        // Direccion a la luz, invertimos la direccion del sol
        float3 dirToLight = -_SunDir;
        const float epsilon = 0.085;  // Una constante cercana
           a cero

        // Evita dividir por un numero muy cercano a cero mas
           adelante
        if(abs(dirToLight.z) < epsilon)
            dirToLight.z = sign(dirToLight.z) * epsilon;


        // Calcula la distancia dentro de la caja
        float distanceInsideBox = rayBoxDistance(pos, 1 /
           dirToLight, _BoundsMin, _BoundsMax).y;

        // Calcula la longitud de paso (Numero de pasos fijado
           en el inspector)
        float stepSize = distanceInsideBox / _NumStepsLight;

        // Avanza medio paso antes de entrar al bucle
        pos += dirToLight * stepSize * 0.5f;

        // Bucle para calcular la densidad total
        float totalDensity = 0;
        for (int i = 0; i < _NumStepsLight; i++)
        {
            totalDensity += max(0, sampleDensity(pos) *
               stepSize);
            pos += dirToLight * stepSize;
        }

        float transmittance = exp(-totalDensity *
           _LightAbsortionSun);

        return _DarknessThreshold + transmittance * (1 -
           _DarknessThreshold);
    }
```

Código A.7: Primera parte del shader de fragmentos

```
1     fixed4 frag(v2f i) : SV_Target
2     {
3         // Calcular el vector de vista
4         float3 rayDir = normalize(i.viewVector);
5
6         // Obtener la profundidad del pixel
7         float nonLinearDepth = SAMPLE_DEPTH_TEXTURE(
              _CameraDepthTexture, i.uv);
8         float depth = LinearEyeDepth(nonLinearDepth) * length(i
              .viewVector);
9
10        // Calcular distancia al cuboide delimitador y dentro
              de este
11        float2 rayBoxInfo = rayBoxDistance(_WorldSpaceCameraPos
              , rayDir, _BoundsMin, _BoundsMax);
12        float dstToBox = rayBoxInfo.x;
13        float dstInsideBox = rayBoxInfo.y;
14
15        // Calcular el punto de entrada del rayo en el volumen
16        float3 entryPoint = _WorldSpaceCameraPos + rayDir *
              dstToBox;
17
18        // Inicializar variables raymarching
19        float dstTravelled = 0;
20        float stepSize = _FixedStepSize;
21        float transmittance = 1;
22        float lightEnergy = 0;
23
24        // Calcular coseno del angulo entre el rayo y la
              direccion del sol
25        float cosAlpha = dot(rayDir, -_SunDir);
26
27        // Calculamos Henyey - Greenstein Phase Function
28        float phaseResult = phase(cosAlpha);
29
30        // Limite de distancia a recorrer
31        float dstLimit = min(depth - dstToBox, dstInsideBox);
```

Código A.8: Bucle principal asociado a la técnica RayMarching

```
    // Bucle de ray marching
    while (dstTravelled < dstLimit)
    {
        // Calcular la posicion actual a lo largo del rayo
        float3 rayPosition = entryPoint + rayDir * dstTravelled
            ;

        // Obtener densidad en la posicion actual
        float density = sampleDensity(rayPosition);

        if (density > 0)
        {
            // Obtener transmitancia de luz en la posicion
                actual
            float lightTransmittance = lightAtPos(rayPosition);

            // Acumular energia de luz (HG y Powder)
            lightEnergy += density * stepSize * transmittance *
                 lightTransmittance * phaseResult * powder(
                density);

            // Actualizar transmitancia usando la Ley de Beer
            transmittance *= exp(-density * stepSize *
                _LightAbsortionCloud);

            // salida si la transmitancia es muy baja
            if (transmittance < 0.01)
                break;
        }
        // Avanzar a lo largo del rayo
        dstTravelled += stepSize;
    }

    // Obtener el color de fondo
    float3 backgroundColor = tex2D(_MainTex, i.uv);

    // Calcular el color de las nubes
    float3 cloudColor = lightEnergy * _SunColor;

    // Combinar el color de las nubes con el color de fondo
    return fixed4(backgroundColor * transmittance + cloudColor,
        1.0);
}
```

B

Código SkyBox Procedimental

Código B.1: Propiedades y declaraciones previas

```
Shader "ProceduralSkybox"
{
    Properties
    {
        _SunSize ("Sun Size", Range(0, 1)) = 0.05
        _SunIntensity ("Sun Intensity", float) = 2
        _SunCol ("Sun Colour", Color) = (1, 1, 1, 1)
        _SunDirectionWS("Sun DirectionWS", Vector) = (1, 1, 1,
           1)
        _ScatteringIntensity("Scattering Intensity", float) = 1
        _StarTex ("Star Texture", 2D) = "white" { }
        _MoonCol ("Moon Color", Color) = (1, 1, 1, 1)
        _MoonIntensity("Moon Intensity", Range(1, 3)) = 1.2
        _MoonDirectionWS("Moon DirectionWS", Vector) = (1, 1,
           1, 1)
        _StarIntensity("Star Intensity", float) = 1
    }
    SubShader
    {
        Tags { "Queue" = "Background" "RenderType" = "
           Background" "RenderPipeline" = "
           UniversalRenderPipeline" }

        HLSLINCLUDE
        #include "Packages/com.unity.render-pipelines.universal
           /ShaderLibrary/Core.hlsl"

        CBUFFER_START(UnityPerMaterial)

        float4 _SunCol;
        float4 _SunDirectionWS;
        float4 _MoonDirectionWS;
        float _SunSize;
        float _SunIntensity;
        float _ScatteringIntensity;

        float4x4 _MoonWorld2Obj;
        float4 _MoonCol;
        float _MoonIntensity;
        float _StarIntensity;

        CBUFFER_END
        ENDHLSL
```

Código B.2: Métodos auxiliares

```
Pass
    {
        Name "Skybox"
        HLSLPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        #define PI 3.1415926535
        #define MIE_G (-0.990)
        #define MIE_G2 0.9801
        static const float PI2 = PI * 2;
        static const float halfPI = PI * 0.5;

        struct a2v
        {
            float4 positionOS: POSITION;
        };
        struct v2f
        {
            float4 positionCS: SV_POSITION;
            float3 positionWS: TEXCOORD1;
            float3 moonPos: TEXCOORD2;
            float3 positionOS: TEXCOORD3;
        };

        TEXTURE2D(_SkyGradientTex);
        SAMPLER(sampler_SkyGradientTex);
        TEXTURE2D(_StarTex);
        SAMPLER(sampler_StarTex);
        TEXTURE2D(_MoonTex);
        SAMPLER(sampler_MoonTex);

        half getMiePhase(half eyeCos, half eyeCos2)
        {
            half temp = 1.0 + MIE_G2 - 2.0 * MIE_G * eyeCos;
            temp = pow(temp, pow(_SunSize, 0.65) * 10);
            temp = max(temp, 1.0e-4); // Previene /0
            temp = 1.5 * ((1.0 - MIE_G2) / (2.0 + MIE_G2)) *
                (1.0 + eyeCos2) / temp;
            return temp;
        }
        half calcSunAttenuation(half3 lightPos, half3 ray)
        {
            half focusedEyeCos = pow(saturate(dot(lightPos, ray
                )), 5);
            return getMiePhase(-focusedEyeCos, focusedEyeCos *
                focusedEyeCos);
        }
```

Código B.3: Shader de Vértices

```
v2f vert(a2v v)
    {
        v2f o;

        VertexPositionInputs positionInputs =
           GetVertexPositionInputs(v.positionOS.xyz);

        o.positionCS = positionInputs.positionCS;
        o.positionWS = positionInputs.positionWS;
        o.positionOS = v.positionOS.xyz;

        o.moonPos = mul((float3x3)_MoonWorld2Obj, v.positionOS.
           xyz) * 6;
        o.moonPos.x *= -1;

        return o;
    }
```

Código B.4: Shader de Fragmentos

```
 half4 frag(v2f i): SV_Target
{
    // Conversion de las coordenadas
    float3 positionOSNormalized = normalize(i.positionOS);
    float2 sphereUV = float2(atan2(positionOSNormalized.x,
       positionOSNormalized.z) / PI2, asin(positionOSNormalized
       .y) / halfPI);

    // Contribucion del Sol
    half4 sun = calcSunAttenuation(positionOSNormalized, -
       _SunDirectionWS) * _SunIntensity * _SunCol;
    half scatteringIntensity = max(0.15, smoothstep(0.6, 0.0, -
       _SunDirectionWS.y));
    half4 scattering = smoothstep(0.5, 1.5, dot(
       positionOSNormalized, -_SunDirectionWS.xyz)) * _SunCol *
        _ScatteringIntensity * scatteringIntensity;
    sun += scattering;

    // Muestreo del color del cielo
    half4 skyColor = SAMPLE_TEXTURE2D(_SkyGradientTex,
       sampler_SkyGradientTex, float2(sphereUV.y, 0.5));

    // Muestreo y ajuste de la intensidad de las estrellas
    float star = SAMPLE_TEXTURE2D(_StarTex, sampler_StarTex,
       sphereUV  * 2).r;
    star = saturate(star * star * star * 3) * _StarIntensity;

    // Contribucion de la Luna
    half4 moon = SAMPLE_TEXTURE2D(_MoonTex, sampler_MoonTex, (i
       .moonPos.xy + 0.5)) * step(0.5, dot(positionOSNormalized
       , -_MoonDirectionWS.xyz));
    half4 moonScattering = smoothstep(0.97, 1.3, dot(
       positionOSNormalized, -_MoonDirectionWS.xyz));
    moon = (moon * _MoonIntensity + moonScattering * 0.8) *
       _MoonCol;

    // Suma todas las contribuciones y devuelve el resultado
    return skyColor + sun + star + moon;
}
ENDHLSL
```