Universidad Zaragoza

Proyecto Fin de Carrera

Generación de paisajes procedurales con Direct3D y GPU

Autor

D. Juan Gallego Molina

Director

Dr. D. Francisco José Serón Arbeloa

Escuela de Ingeniería y Arquitectura

2013/2014

Gracias a todos los compañeros que he tenido durante estos años, tanto en Zaragoza como en mis aventuras en Cork y Donosti, donde me he sentido como en casa.

A los profesores que me han ayudado durante mi formación, sobre todo a Paco por sus consejos.

Y en especial a mi familia, sobre todo a mi hermano, que ya me avisó de los peligros de querer ser ingeniero.

Generación de paisajes procedurales con Direct3D y GPU

RESUMEN

Las modernas unidades de procesamiento gráfico ofrecen capacidades operacionales de cálculo general. Este ámbito está cobrando importancia ya que muchos cálculos pueden derivarse a la GPU, donde la existencia de numerosos núcleos trabajando en paralelo permite obtener una mayor rapidez en la resolución de algunos problemas, tanto de índole científico como para los videojuegos.

En este ámbito es donde surge este Proyecto de Fin de Carrera. En concreto se ha creado un paisaje procedural formado por varios elementos como montañas, playa, agua, un planeta y nubes. Todos ellos se han creado utilizando el ruido de Perlin para generar formas fractales. Dicho algoritmo está especialmente aconsejado para la generación de objetos naturales y además es adecuado para aprovechar la capacidad de cálculo en paralelo de las unidades de procesamiento gráfico, ya que cada vértice puede calcularse de manera individual.

Además de los aspectos relacionados con la geometría de estos elementos se han tenido en cuenta otros componentes de una pipeline gráfica como son el modelo de iluminación y sombras, el texturizado de los objetos, el audio, el movimiento de la cámara y la animación de algunos elementos.

La aplicación se ha desarrollado utilizando la API Direct3D 11 de Microsoft, ya que es una de las más usadas en el mundo de los gráficos y nos permite aprovechar las características de la GPU. Además este API contiene varias herramientas que sirven para controlar el estado de actividad de la unidad gráfica, incluyendo el tiempo que está trabajando, lo que nos ha permitido poner de manifiesto el beneficio de una ejecución en paralelo.

Finalmente, se ha integrado el algoritmo de ruido de Perlin en el entorno de desarrollo de videojuegos UDK, Unreal Development Kit, donde se ha usado para crear un planeta fractal. De esta manera se demuestra la capacidad técnica para adaptar y aplicar los conocimientos adquiridos en aplicaciones comerciales ya existentes.

ÍNDICE DE CONTENIDOS

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1. INTRODUCCIÓN

El presente documento tiene como objetivo presentar toda la información relacionada con la realización de este Proyecto de Fin de Carrera (PFC) titulado "Generación de paisajes procedurales con Direct3D y GPU". En este apartado titulado "Introducción" se describe el contexto tecnológico en el que se inserta este proyecto así como sus objetivos y la estructura general de la memoria.

1. 1. Contexto tecnológico

1. 1. 1. Unidades de procesamiento gráfico

Las unidades de procesamiento gráfico (GPU) han evolucionado enormemente a lo largo de los años, pasando de ofrecer una pipeline fija a otra mucho más flexible que puede ejecutar programas, denominados "shaders", realizados por cualquier desarrollador usando alguno de los lenguajes existentes: HLSL, de Microsoft; CG, de nVidia; o GLSL, libre, aunque realmente hay pocas diferencias conceptuales además de las sintácticas.

Es aquí donde se basa el concepto de cálculo general en la unidad gráfica, o GPGPU por su nombre en inglés, "General Purpose computing on Graphics Processing Units". Debido a que las GPU no son más que coprocesadores especializados en operaciones con números reales es teóricamente posible utilizar su gran número de núcleos trabajando en paralelo para cualquier tipo de problema paralelizable del tipo SIMD, "Single Instruction Multiple Data".

Para la programación de las unidades gráficas se usan distintas API, siendo Direct3D, propiedad de Microsoft, y OpenGL, libre, las más utilizadas para el apartado gráfico. Por otra parte, DirectCompute, también de Microsoft; CUDA, de nVidia, y OpenCL son las usadas para la parte computacional. Todas ellas se pueden integrar fácilmente en cualquier aplicación.

1. 1. 2. Videojuegos

Entre las aplicaciones comunes que más utilizan la GPU están los videojuegos. Por ello se ha querido enmarcar este proyecto en este ámbito.

El desarrollo de un videojuego es un proceso complejo que no se limita tan solo a la programación de la parte visual. Recordemos algunas de las fases por las que se pasa durante su creación:

- **Concepción:** definir la idea, el género del juego, cómo y dónde va a ser jugado, la historia, los personajes o la ambientación.

- **Diseño:** definir las ideas obtenidas en el paso anterior, refinando la historia a contar, la jugabilidad y comenzar con la parte artística, realizando bocetos del mundo y sus componentes. También puede ser el momento de elegir las herramientas y metodologías a usar durante el resto del desarrollo.

- **Planificación:** dividir el trabajo a realizar en sus distintas etapas, así como marcar fechas límites y los entregables de cada fase. También habrá que tener en cuenta el presupuesto que se necesitará para acometer el trabajo y el personal requerido.

- **Desarrollo:** es el momento de crear todos los componentes del juego, que podríamos dividir en varios apartados:
    - Arte 2D – 3D: los modelos 3D o los elementos 2D, como sprites o texturas. Se pueden crear usando herramientas propias pero lo habitual es usar algunas herramientas externas que están muy extendidas por su calidad, como Zbrush o Maya para los modelos en 3D y Photoshop o Gimp para los 2D.
    - Motor del juego: el motor es el encargado de proporcionar la mayoría de funcionalidades necesarias como el renderizado de los gráficos, la animación, detección de colisiones, IA, simulación de físicas, networking, gameplay o audio. No obstante, programar un motor entero desde cero es muy costoso, por lo que es habitual usar alguno ya existente, como Unity o Unreal Engine, y modificar algunas partes para adaptarlo a nuestras necesidades. Hay que tener en cuenta que las licencias para usar un motor pueden ser muy caras y las gratuitas no suelen ofrecer acceso a todo el código de éste, por lo que la capacidad de modificarlo es limitada.
    - Audio: hay que crear, bien desde cero o utilizando algunos elementos ya grabados, todos los sonidos del juego, tanto los producidos por los objetos como las voces y la música. También suelen usarse herramientas externas como Wwise y FMOD.
    - Niveles: Una vez contamos con un motor y todos los elementos artísticos es el momento de unirlos y crear los niveles jugables, lo que implica colocar los objetos en sus sitios y programar los distintos eventos del juego.

- **Pruebas:** Como en cualquier proyecto hay que comprobar la calidad del producto. Por esto se requiere un conjunto de tests completo que comprendan todos los apartados del juego y un plan de soluciones para llevar a cabo en caso de encontrar errores.

Es en la parte del motor gráfico donde se ubica este proyecto, evidentemente no en su creación completa, pero sí en la posibilidad de aplicar los conocimientos adquiridos para la programación de alguna de sus partes y su posterior integración con el resto del motor, en especial aquellos apartados relacionados con las unidades gráficas. Para ello se analizaron 3 opciones distintas donde poder integrar los resultados de este trabajo, todas ellas para PC con dos requisitos, que fueran potentes y de uso gratuito:

- **Unity:** es uno de los motores gratuitos más usados por desarrolladores indies. Soporta las plataformas más importantes: PC, Linux, PS3, Xbox360, WiiU, iOS y Android. Se caracteriza por su facilidad de uso, con la posibilidad de crear prototipos de manera rápida, sencilla y flexible, ya que existen muchos componentes creados previamente por la comunidad existente. Además nos permite programar nuestros propios shaders, ya sea usando Cg, GLSL o HLSL.

La versión gratis es, no obstante, incompleta y solo la de pago ofrece todas las características [1].

- **CryEngine 3:** es uno de los motores más famosos en la industria ya que ofrece gráficos realistas y simulación de físicas para PC, PS3 y Xbox360. Permite usar un conjunto de herramientas completas de manera gratuita, siendo el editor de terrenos uno de sus puntos más fuertes. Aunque el código del motor no es accesible directamente, se puede personalizar y programar en C++ gran parte de los subsistemas de éste, que llamarán al código fuente. No obstante, el apoyo de la comunidad y la existencia de tutoriales es escaso, requiriendo conexión a Internet para identificarse y usar las herramientas. Además el sistema de shaders es cerrado, por lo que no es factible programar los deseados [2].

- **Unreal Engine 3 / Unreal Development Kit:** UDK es la versión gratuita del motor Unreal Engine 3, uno de los motores más potentes y usados en la industria. Ofrece un completo conjunto de herramientas que permiten crear juegos para PC y iOS, aunque la versión de pago soporta también MAC, Xbox 360, PS3, WiiU, PSVita y Android. No es posible acceder directamente al código fuente, pero si usar el lenguaje de programación UnrealScript para retocar y acceder a cualquier parte del motor a alto nivel. La comunidad existente es muy amplia, con gran número de tutoriales, y además es posible crear nuestros propios shaders usando HLSL [3].

Puede verse una comparación en detalle de las características que ofrece cada motor en el ANEXO B: COMPARACIÓN DE MOTORES.

En este trabajo cabe mencionar que CryEngine deja de ser una opción puesto que no permite la programación de shaders. En cuanto a Unity y UDK, es este último el que ofrece más funcionalidades y por eso es el motor elegido.

El motor Unreal en PC utiliza Direct3D como API gráfica, por lo que es obligatorio elegir HLSL como lenguaje de programación de shaders..

El motor puede usar tanto Direct3D 9 como 11, que son las dos últimas versiones más extendidas de la API. La versión 10 es muy poco utilizada, ya que se creó para funcionar en el sistema operativo Windows Vista, cuya penetración en el mercado fue escasa en comparación con XP, que usa la versión 9. Direct3D 11 se desarrolló para funcionar con Windows 7 y es la elegida puesto que es la más moderna y la que más funcionalidades incorpora, incluyendo el último modelo de shaders que nos permite usar compute shaders y utilizar la GPU para cómputo general [4].

1. 2. Objetivos

La idea de este proyecto es utilizar las unidades de procesamiento gráfico y mostrar sus capacidades en el mundo de los videojuegos.

Los objetivos concretos a alcanzar son:

- Generación de una escena formada por elementos naturales.
- Comprender el funcionamiento de la API Direct3D, la pipeline gráfica y computacional.
- Implementar la creación de agua, montañas, nubes y planetas basados en algoritmos procedurales fractales y ruido de Perlin.
- Comparar el rendimiento de la implementación entre CPU y GPU
- Integrar parte de las funcionalidades implementadas dentro de un motor gráfico para videojuegos ya existente.

1. 3. Estructura del documento

La memoria del PFC se divide en los siguientes capítulos:

I. Trabajo realizado. Se describen los algoritmos fractales que se han implementado. Posteriormente se explica la estructura conceptual de Direct3D, que es la API elegida para desarrollar el trabajo. Después se pasa a describir el paisaje modelado y el algoritmo en que se basa su apariencia. Se continúa comparando el rendimiento obtenido usando una CPU y una GPU y se finaliza comentando la integración de parte de las funcionalidades desarrolladas en un motor gráfico para videojuegos ya existente.

II. Conclusiones. Se comentan los resultados obtenidos, las conclusiones alcanzadas y posibles maneras de continuar en el futuro con el trabajo.

III. Bibliografía. Referencias de artículos y libros utilizados.

IV. Anexos. En ellos se profundiza en distintos aspectos del proyecto que, por brevedad, no pueden comentarse en el cuerpo principal de la memoria.

2. TRABAJO REALIZADO

En esta sección se detalla el trabajo realizado a lo largo del PFC. Se comienza describiendo los fractales, sección 2.1, y se sigue con el estudio de Direct3D, API utilizada para crear nuestra aplicación, en la sección 2.2. Después se explica la creación de la escena, sección 2.3. A continuación se compara el rendimiento obtenido usando una CPU y una GPU, sección 2.4. Por último, se integra parte del trabajo realizado en el motor Unreal Engine, sección 2.5.

2. 1. Fractales y ruido

2. 1. 1. Fractales

Un fractal es un patrón que se repite a diferentes escalas para describir la forma de un objeto irregular que no se podría obtener con los métodos de la geometría clásica. Éstos nos permiten, a través de algoritmos, reproducir la complejidad de elementos típicos de la naturaleza como son las montañas, las nubes o incluso planetas enteros.

Aunque las ideas que existen tras estos métodos se remontan al siglo 17 en esa época los matemáticos estaban limitados a las imágenes que podían reproducir ellos mismos. Es por eso que empiezan a cobrar importancia a partir de los años 70, cuando aparecen los primeros gráficos por computador. Es en el año 1979 cuando se crea la primera animación con elementos fractales, un corto de 2 minutos llamado “Vol Libre” de Loren Carpenter [5].

No obstante, el primer trabajo formal en el que se define el concepto de los fractales tal y como los conocemos ahora se debe a Benoit Mandelbrot en su libro “The Fractal Geometry of Nature”, publicado en 1982 [6]. En ese año también se crea el primer planeta fractal mostrado en el cine en la película Star Trek II, también por Carpenter y su equipo de Pixar [7].

Desde entonces las imágenes creadas usando fractales han evolucionado junto con la tecnología, que cada vez permite alcanzar mayor realismo. Algunas de estas imágenes pueden verse en el ANEXO F: RUIDO Y FRACTALES.

Es importante mencionar la aparición en 1985 del Ruido de Perlin, ideado por Ken Perlin [8]. Con él comenzaron los métodos procedurales de generación de fractales que aún hoy se siguen utilizando. En el mismo anexo citado se presenta una descripción más detallada de las funciones usadas en estos métodos.

Finalmente, comentar la existencia de la primera edición del libro “Texturing & Modelling: A Procedural Approach” en 1994 [9]. Es uno de los libros de referencia sobre este tema y el que ha servido, en su tercera edición, como guía e inspiración para el trabajo realizado en este PFC.

En ese mismo libro, además de tratar los problemas tradicionales encontrados al trabajar con fractales, también se presentan las técnicas más actuales y los nuevos retos aparecidos al usar unidades gráficas como herramienta de cálculo.

2. 1. 2. Ruido de Perlin

El ruido de Perlin es una función matemática que usa interpolación entre varios gradientes precalculados para obtener un valor entre -1 y 1 que varía pseudo-aleatoriamente en el espacio o en el tiempo. Además tiene la particularidad de ser coherente, es decir, el ruido cambia suavemente y no existen discontinuidades [10].

La función puede definirse en cualquier espacio, sea cual sea su número de dimensiones.

El espacio de *n* dimensiones que elijamos se divide en una malla definida para cada punto con coordenadas enteras y a cada uno de ellos se le asigna uno de los gradientes precalculados. Dichos gradientes se generaban inicialmente de manera aleatoria con componentes entre -1.0 y 1.0, descartando aquellos cuya longitud sea superior a 1 y normalizando el resto para evitar desviaciones alineadas con alguno de los ejes del espacio.

En una revisión posterior Perlin decidió dejar de generar gradientes aleatorios ya que la manera de asignarlos, comentada a continuación, otorga suficiente aleatoriedad. Desde entonces se utilizan gradientes que van desde el centro de un objeto de referencia de dimensiones unidad, como un cuadrado en 2 dimensiones o un cubo en un espacio 3D, a las aristas de dicho objeto y se normalizan, como vemos en la figura 1.

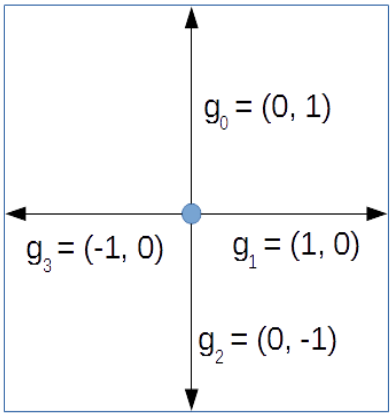


Figura 1: Gradientes generados en 2 dimensiones

La asignación de estos gradientes a cada punto de una malla de dimensión [*t, t*] se realiza utilizando una permutación aleatoria de números enteros de tamaño *t* y las coordenadas del punto, como mostramos a continuación para un espacio de dos dimensiones:

Grad es una tabla de dimensión [4, 2] con los gradientes g_0, g_1, g_2 y g_3.
Perm es una tabla de dimensión *t* con una permutación de enteros de tamaño *t*.
P es un punto de la malla con coordenadas (x,y).
El gradiente asignado a dicho punto es:

Grad[modulo(4, Perm[modulo(t, y+Perm[x])])].

Para obtener el valor del ruido en cualquier punto de entrada *p* a la función debemos usar los 2^n puntos de la malla que lo rodean, como vemos en la figura 2.

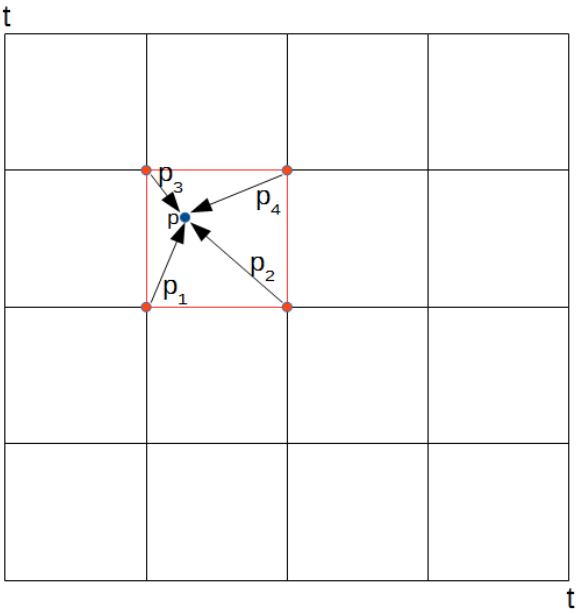


Figura 2: Malla en 2 dimensiones, n = 2

Para cada uno de ellos tendremos un gradiente asignado, en este caso $g_1 - g_4$, y un vector que va desde cada punto de la malla que rodea a *p*, tras lo cual obtendremos el producto escalar de cada pareja de vectores asociados al mismo punto de la malla, como se ve en el ejemplo de 2 dimensiones:

d_1 = producto escalar (p_1, g_1)
d_2 = producto escalar (p_2, g_2)
d_2 = producto escalar (p_3, g_3)
d_2 = producto escalar (p_4, g_4)

Finalmente interpolamos los 2^n productos escalares usando la función $6p_1^5 - 15p_1^4 + 10p_1^3$ para obtener el valor del ruido. En el ejemplo dado primero interpolamos en x entre d_1 y d_2 y entre d_3 y d_4. A continuación interpolamos los 2 resultados obtenidos en y para obtener el ruido final *r*. Elegimos esta función por ser una curva de forma "S" que crece suavemente de 0 a 1 exagerando la cercanía a ambos extremos. Además tanto su primera como su segunda derivada son 0 si las evaluamos en 0 o en 1, lo que evita que se vean discontinuidades cuando tenemos una superficie desplazada usando el ruido de Perlin y debemos calcular la normal.

Este procedimiento permite generar una función de ruido con frecuencia 1 y amplitud 1.

Para definir la función con frecuencia *f* y amplitud *a* debemos cambiar algunas cosas.

Lo primero que modificamos es que el espacio se dividirá en una malla de dimensión [*f* * *t*, *f* * *t*] y la asignación de los gradientes será como sigue:

Grad es una tabla de dimensión [4, 2] con los gradientes g_0, g_1, g_2 y g_3.
Perm es una tabla de dimensión *t* con una permutación de enteros de tamaño *t*.
P es un punto de la malla con coordenadas (x,y).
El gradiente asignado a dicho punto es

Grad[modulo(4, Perm[modulo(t, y+Perm[modulo(t, x)])])].

Además usaremos como punto de entrada a la función *f* * *p* y el valor que se devolverá finalmente *a* * *r*.

La función de ruido es utilizada posteriormente para generar un fractal. Para ello se suman distintas frecuencias de ruido a diferentes escalas, llamadas octavas si cada frecuencia es múltiplo de 2, como se explica con más detalle en el ANEXO F: RUIDO Y FRACTALES y se muestra en la figura 3.

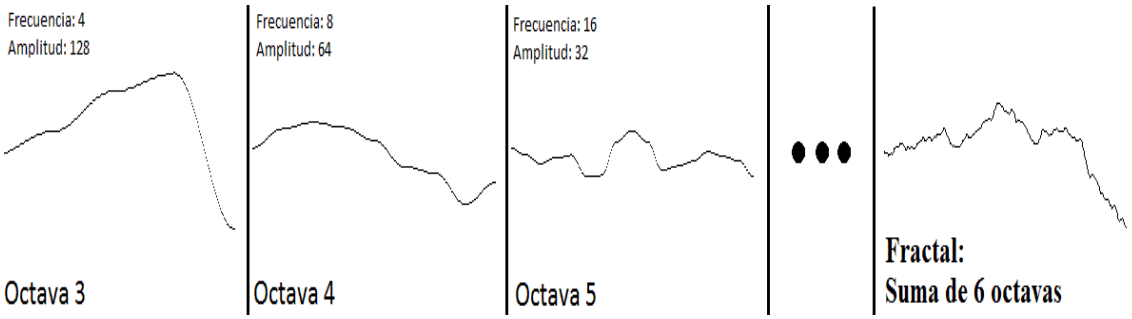


Figura 3: Octavas de ruido

Además del ruido de Perlin hay otros tipos de ruido, cuya información se puede consultar en el mismo anexo que acabamos de citar [11].

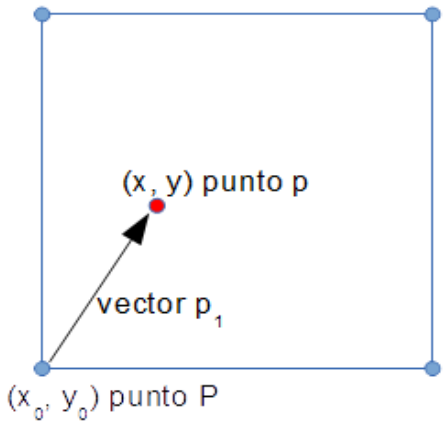
2. 1. 3. Ruido de Perlin en la GPU

A continuación se explica cómo se ha implementado el ruido de Perlin en la GPU, utilizando algunas figuras explicativas y pseudocódigo [12].

Aunque se han implementado versiones en 2 y 3 dimensiones aquí solo se explica la primera ya que es sencillo añadir una o varias dimensiones adicionales.

Inicialmente creamos una permutación de números enteros de tamaño 1024 y se generan los 4 gradientes que van desde el centro de un cuadrado a sus aristas. Ambos datos se guardarán en texturas, que además de permitirnos hacer accesos rápidos se encargan de hacer la operación módulo automáticamente.

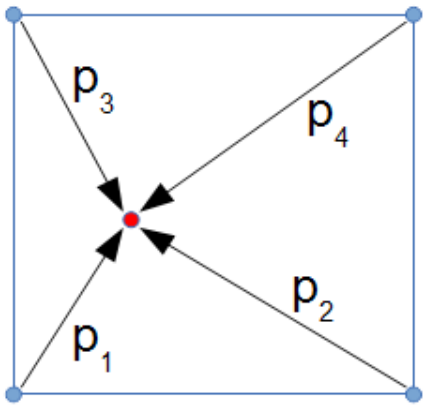
Cada punto p con coordenadas reales (x,y) cae dentro de una celda de la malla, la cual podemos localizar a partir de la esquina (x_0, y_0). De esta misma manera se puede obtener el vector que va desde dicha esquina al punto p, como se observa en la figura 4.



p es el punto en el que queremos calcular el ruido de Perlin;
P = floor (p);
$p_1 = p - P$;

Figura 4: Ruido de Perlin, paso 1

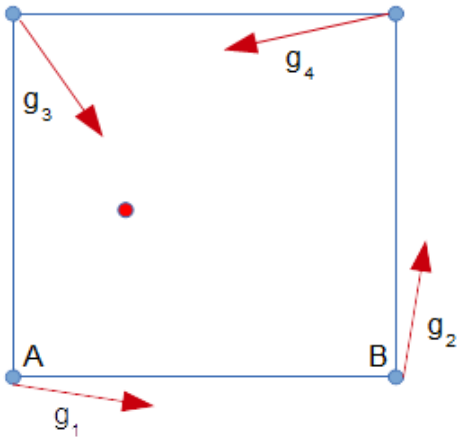
Usando este vector p_1 obtenemos el resto de vectores que van desde las esquinas de la celda al punto p, como se muestra en la figura 5.



$p_2 = p_1 + (-1, 0);$
$p_3 = p_1 + (0, -1);$
$p_4 = p_1 + (-1, -1);$

Figura 5: Ruido de Perlin, paso 2

Ahora que tenemos los 4 vectores debemos obtener los gradientes asignados a cada punto de la malla para obtener los productos escalares. Para ello necesitamos saber los índices correspondientes para indexar la tabla de gradientes, los cuales se obtienen usando la permutación de números enteros, como se ve en la figura 6.



A = permutación(P.x) + P.y;
B = permutación(P.x + 1) + P.y;

g_1 = gradiente(permutación(A));
g_2 = gradiente(permutación(B));
g_3 = gradiente(permutación(A+1));
g_4 = gradiente(permutación(B+1));

Figura 6: Ruido de Perlin, paso 3

Finalmente, se obtiene el producto escalar de cada vector con el gradiente correspondiente y se interpolan los resultados obtenidos, usando la curva $6t^5 - 15t^4 + 10t^3$ evaluada en p_1, para hallar el valor final del ruido.

$d_1 = dot(p_1, g_1);$ $d_2 = dot(p_2, g_2);$ $d_3 = dot(p_3, g_3);$ $d_4 = dot(p_4, g_4);$
ix_1 = interpolación en x de d_1 y d_2;
ix_2 = interpolación en x de d_3 y d_4;

ruido final = interpolación en y de ix_1 y ix_2;

2. 2. Direct3D

La API utilizada para interaccionar con la GPU en nuestra aplicación es Direct3D 11, la cual está diseñada para poder ser accedida directamente mediante código en C/C++ [4] [13] [14] [15].

2. 2. 1. Arquitectura y pipeline

La arquitectura gráfica seguida en Windows puede verse en la figura 7. La aplicación está en el nivel más alto y es la que controla la escena: objetos 2D y 3D, animaciones, texturas, etc. Ésta interactúa principalmente con Direct3D, que convierte los datos de alto nivel a un formato que pueda ser utilizado por el user mode driver, un driver de alto nivel que no puede acceder directamente al kernel de la GPU para simplificar su funcionamiento y reducir el número de errores.

Este driver se encarga de la traducción a instrucciones que puedan ser ejecutadas por la unidad de procesamiento gráfico y sus resultados se pasan al siguiente nivel, DXGI, que maneja los recursos de la GPU y se comunica directamente con la tarjeta gráfica.

Este nivel está preparado para funcionar con próximas iteraciones de Direct3D, por lo que también puede accederse directamente desde la aplicación si necesitamos realizar alguna operación que aún no está implementada en la API Direct3D 11.

Finalmente, el último nivel es el propio hardware.

Ya que la aplicación se comunica principalmente con Direct3D el programador puede olvidarse del tipo de dispositivo gráfico que esté instalado, abstrayéndose de la complejidad del hardware disponible.

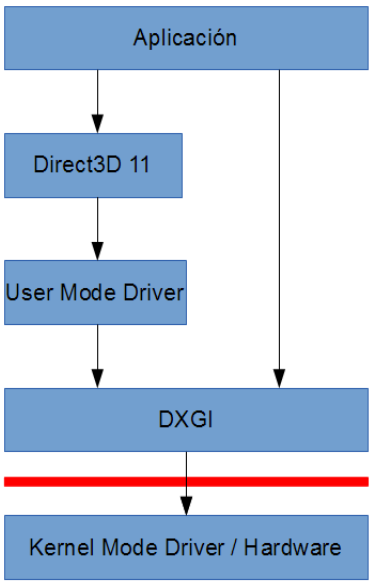


Figura 7: Arquitectura gráfica

Además es importante conocer la configuración de la pipeline de Direct3D, cómo fluyen los datos y se transforman en cada etapa.

La principal pipa es la gráfica, mostrada en la figura 8, que se encarga de transformar una escena 3D en una imagen en 2 dimensiones. Entre las etapas de la pipeline destacan 2, el "vertex shader", que realiza operaciones sobre vértices, y el "píxel shader", que hace operaciones sobre píxeles.

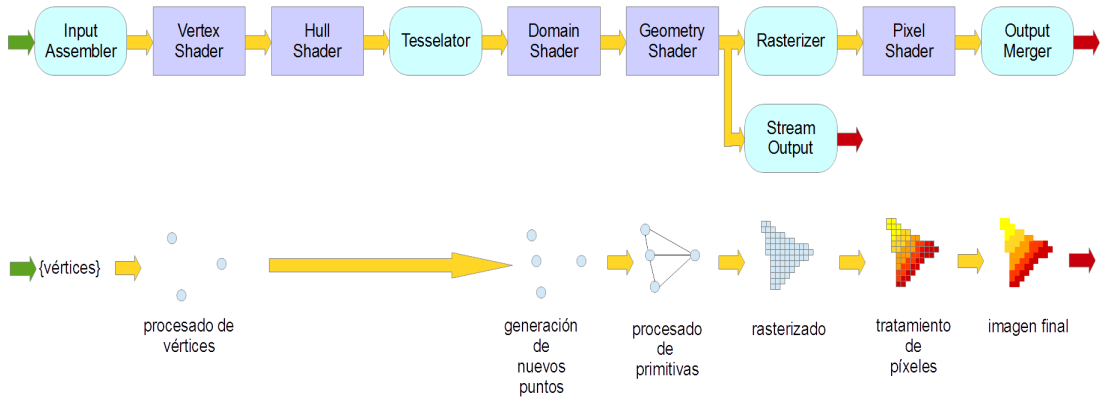


Figura 8: Pipeline gráfica

Además existe una pipeline de cómputo para otras aplicaciones, completamente independiente de la pipa anterior y compuesta por una sola etapa llamada "compute shader", como se ve en la figura 9. En esta pipeline el hardware disponible permite ejecutar operaciones con números enteros y reales, vectores de hasta 4 componentes o matrices de hasta 4x4, ya que disponemos de operadores para hacer sumas, restas, multiplicaciones y divisiones escalar x escalar, escalar x vector, escalar x matriz, vector x vector, vector x matriz y matriz x matriz. Además se pueden hacer operaciones lógicas, logaritmos en base 2 y raíces cuadradas.

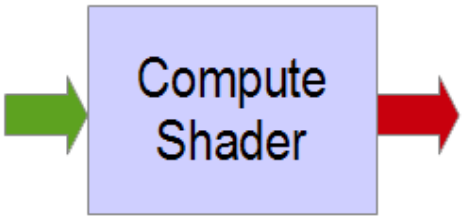


Figura 9: Pipeline computacional

La pipeline computacional nos permite considerar la GPU como un conjunto de procesadores trabajando en paralelo, cada uno de ellos ejecutando un hilo de procesamiento o thread. La disposición de estos hilos, que se unen formando grupos, puede apreciarse en la figura 10, siendo los elementos rojos los grupos de threads y los azules threads concretos.

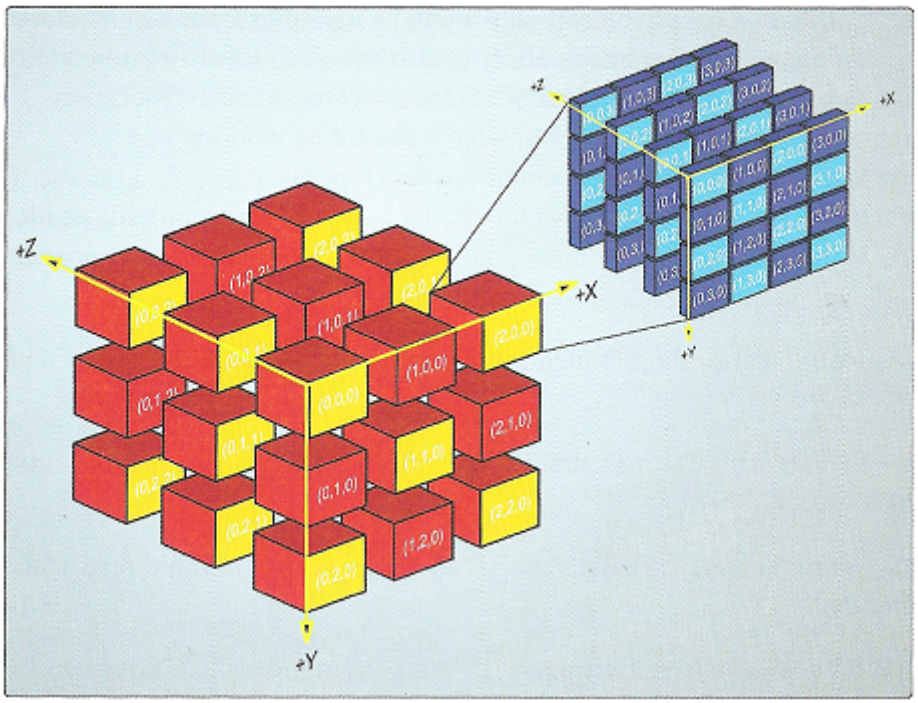


Figura 10: Threads en la GPU

Ambas pipelines se explican con más detalle en el ANEXO C: LA PIPELINE GRÁFICA y el ANEXO D: LA PIPELINE DE CÁLCULO.

2. 2. 2. Interacción con Direct 3D

Los datos en memoria que utiliza la GPU, ya sea en su comunicación con la CPU o en el intercambio de información interna entre etapas, se llaman recursos y Direct3D los cataloga en 2 tipos: buffers y texturas.

Los buffers son conjuntos unidimensionales de datos y se usan para guardar información sobre la geometría u otros datos necesarios en los shaders. Hay 3 tipos de buffers:

- **Vertex Buffers**: Contienen los vértices que definen la geometría de la escena. El formato de cada vértice deber ser especificado por el desarrollador.
- **Index Buffers**: Contienen punteros a los vértices para no repetir información en caso de que algún vértice se utilice en varios polígonos.
- **Constant Buffers**: Contienen cualquier otro tipo de datos. Los formatos de estos datos pueden ser muy variados, existiendo tipos predefinidos, como reales o enteros, y la posibilidad de especificar alguna estructura propia.

Las texturas representan conjuntos de datos estructurados en 1, 2 o 3 dimensiones. Cada elemento de una textura se conoce como texel y contiene información en uno de los muchos formatos predefinidos, siendo los que contienen información de colores los más habituales, como el formato RGBA de 128 bits. La mayoría de unidades de procesamiento gráfico contienen hardware especializado para trabajar con texturas, por lo que si nuestros datos encajan en alguno de los formatos disponibles deberíamos utilizarlas ya que su manejo será mas eficiente que si usáramos un buffer.

Con el uso de texturas también surge la idea de mip-map, distintas versiones de la misma imagen usando diferentes resoluciones. Éstas se pueden intercambiar en función del detalle que necesitemos en cada momento, usando más o menos espacio en la memoria de la GPU. Este concepto se muestra en la figura 11.

Figura 11: Mip-mapping

Además las texturas van asociadas con un "sampler state", un objeto que define como se realiza el acceso a la textura desde un shader.

Con los tipos de recursos explicados debemos introducir el concepto de vista. Una vista representa una instancia de un recurso, es decir, la asociación de un recurso a una etapa concreta de la pipeline. La idea surge porque un mismo recurso puede ser accedido desde distintas partes de la pipa, las texturas, por ejemplo, siempre deben asociarse a la pipeline mediante una vista. Éstas pueden ser de solo lectura o de lectura y escritura.

En las figuras 12 y 13 pueden verse los lugares en ambos tipos de pipeline donde pueden asociarse los recursos, con fondo rojo.

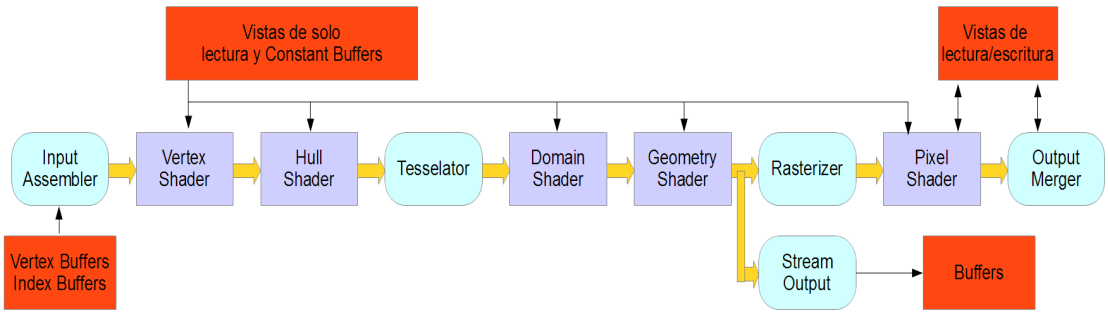


Figura 12: Recursos en la pipeline gráfica

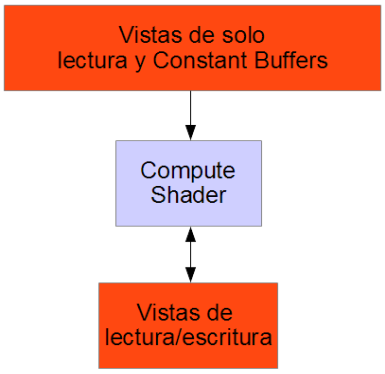


Figura 13: Recursos en la pipeline computacional

Todos los recursos, así como las vistas, se crean mediante la interfaz "Device" de Direct3D 11. Esta misma interfaz también se utiliza para cargar y compilar los shaders y para interrogar al hardware sobre la disponibilidad de ciertas características, ya que estas varían en función de la versión de Direct3D implementada.

La asociación de cada recurso a un lugar concreto de la pipeline se hace usando la interfaz "Device Context", que también es la encargada de configurar las etapas de la pipeline y ordenar su ejecución, con el comando "draw" para ejecutar la pipa gráfica y el comando "dispatch" para la computacional.

Al estado de la pipeline en un momento dado, es decir, su configuración y recursos asociados, se le llama "contexto" y puede ser de dos tipos: inmediato o diferido. El inmediato es único y es el que se usa durante la ejecución de la pipeline mientras que el diferido se guarda para una ejecución posterior, momento en el que se pasará a ser el inmediato. Puede haber varios contextos diferidos y son útiles cuando una CPU tiene varios hilos de procesamiento distintos, de manera que podemos ahorrar tiempo creando varios a la vez en paralelo.

2. 2. 3. Ejemplo de estudio

Para comprender el funcionamiento de Direct3D se ha desarrollado una simulación de agua que nos ha permitido observar las distintas características comentadas. Una descripción detallada de ésta se encuentra en el ANEXO E: APLICACIÓN ESTUDIADA.

2. 3. Creación de la escena

Se ha creado un paisaje con varios elementos generados de manera procedural: unas montañas, una playa, un mar y un planeta. Además en la escena se presentan otros objetos no procedurales como el cielo, unos pájaros y varias palmeras.

Los objetos procedurales se crean usando varias octavas de ruido de Perlin que se suman para dar lugar a un fractal que calcule cierto valor: la altura de un vértice para la playa y la montaña, la posición completa de un vértice en el caso del agua y el color del planeta. En todos ellos los datos de la función fractal se han ajustado para conseguir las

distintas formas naturales por ensayo y error, pues es la mejor manera para conseguirlos, ya que con solo las funciones matemáticas y sus factores no es fácil prever el resultado.

Los 3 primeros elementos parten originalmente de 3 planos distintos y a partir de ellos se calculan los 3 fractales, que se superponen para crear el paisaje, como se ve en las figuras 14, 15 y 16. Aunque es la manera más costosa, es la única forma de conseguir intersecciones realistas entre las distintas partes.

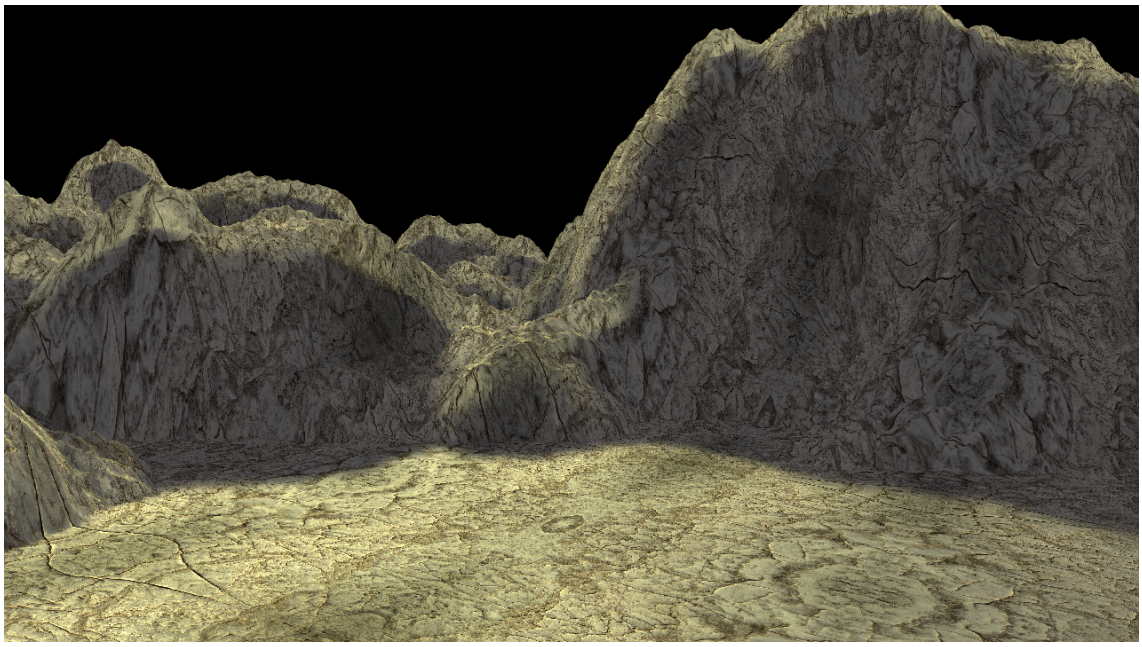


Figura 14: Escena con un solo fractal

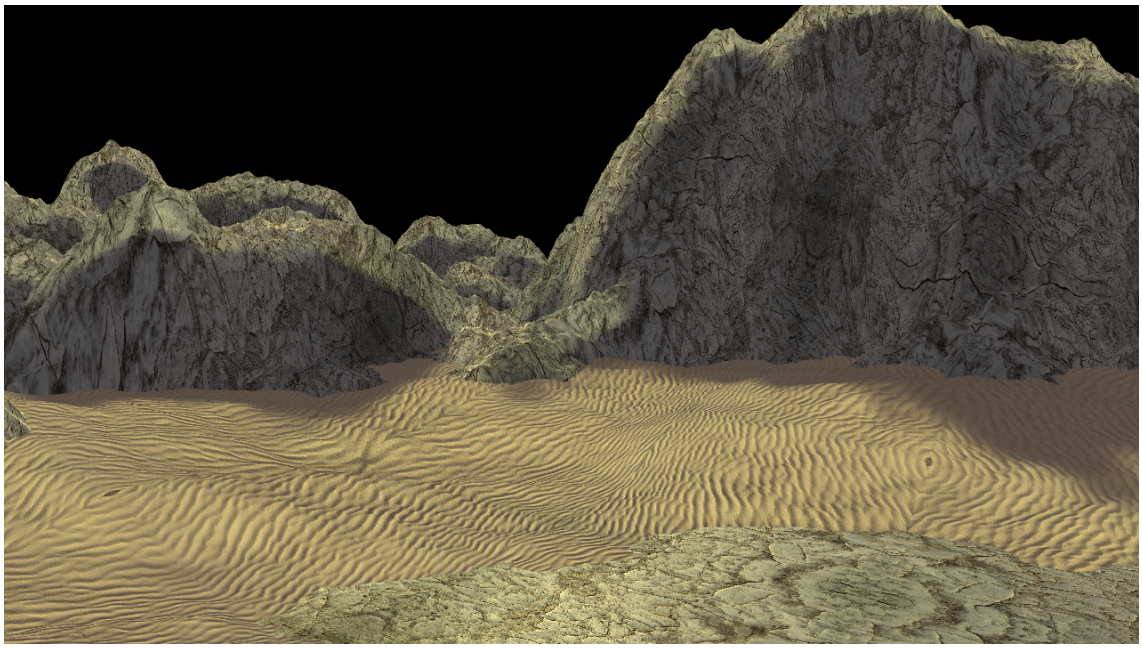


Figura 15: Escena con 2 fractales

Figura 16: Escena con los 3 fractales

A continuación se describen todos los elementos que componen la escena final, que puede verse en la figura 17:

Figura 17: Escena completa

2. 3. 1. Montañas y playa

Las montañas se calculan inicialmente a partir de un plano dividido en triángulos. Un fractal básico con 10 octavas de ruido de Perlin en 2 dimensiones se utiliza para calcular la altura de cada vértice, la cual se escala para obtener el tamaño deseado [16]. Tras haber obtenido las nuevas posiciones de cada vértice debemos calcular las normales en cada uno de ellos para que luego la visualización sea posible.

La playa se calcula de manera similar, con 8 octavas de ruido de Perlin que nos dan la altura del vértice.

Al hacer los cálculos de la altura guardamos información de todos aquellos vértices que superen una altura límite. Estos datos se utilizan durante el cálculo del agua para reducir operaciones, ya que en esos puntos el nivel del agua siempre estará por debajo de los otros elementos y no se verá, por lo que no necesitamos hacer la simulación.

Estos pasos se realizan en la GPU en dos compute shaders, uno para el cálculo de la altura y otro para el de las normales. Se ejecutan al inicio de la aplicación ya que nos basta con hacerlos una vez. Como ambos objetos son planos divididos en vértices aprovechamos esta disposición y organizamos los threads de la GPU en 3 dimensiones como dos planos de igual tamaño, mostrados en la figura 18. Cada grupo se compone de un único thread ya que no necesitamos una organización más compleja y a cada hilo se le asigna un vértice.

Gracias a esta disposición podemos tener todos los núcleos de la unidad gráfica trabajando en paralelo, cada uno con un vértice distinto, algo que en una CPU debería hacerse de manera secuencial y tardaría más.

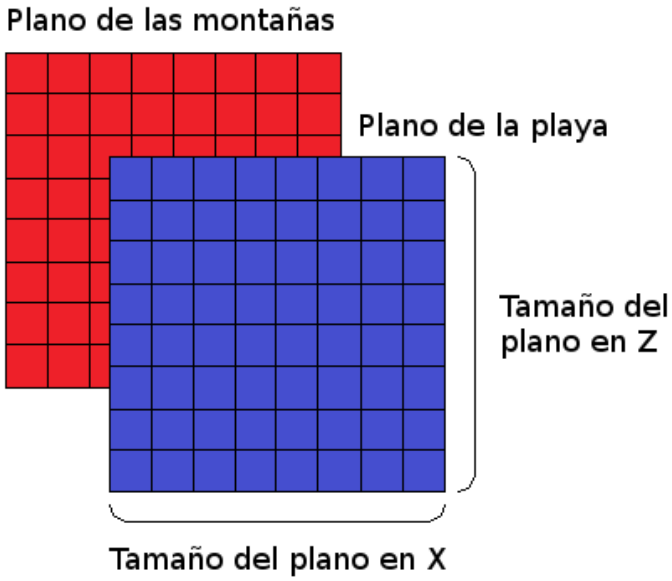


Figura 18: Cálculo del fractal de la playa y las montañas

2. 3. 2. Agua

De nuevo partimos de un plano de triángulos cuyos vértices se van moviendo con el tiempo. Ya que debemos actualizarla en cada momento el agua es el componente que más influye en el rendimiento de la aplicación, por eso es importante no calcular zonas que no se ven, para lo que utilizamos los datos guardados en el paso anterior, o no añadir el máximo detalle en zonas muy alejadas de la cámara.

Para comenzar se utiliza una función sinusoidal para modificar la altura de cada vértice, lo que nos da un oleaje base. A continuación, en función de la distancia a la cámara, se utilizan una o dos octavas de ruido de Perlin en 3 dimensiones que modifican la posición del vértice. Se añade una dimensión respecto a los casos anteriores para utilizar

el tiempo como factor dinámico. Además en este caso no se utiliza directamente el valor del ruido, sino que se usa la inversa de su valor absoluto para acentuar los bordes.

Para añadir algo más de detalle se suma otro fractal que simula la influencia del viento. En su cálculo se utilizan 2 octavas de ruido de Perlin distorsionado, es decir, desplazamos el punto inicial una distancia aleatoria en la dirección (1,1,1) antes de calcular el valor del ruido. Como obtener esto cada instante incrementaría el tiempo de procesamiento lo calculamos una vez al inicio de la aplicación y lo guardamos en una textura. Esto nos obliga a tener un viento fijo en cada posición, pero los cambios que el fractal del agua sufre con el tiempo ya otorgan suficiente dinamismo a la superficie.

Tras haber actualizado la posición de los vértices debemos calcular la normal igual que hacemos con la playa y las montañas.

Cuando hacemos los cálculos de la posición también guardamos 2 texturas de información. Una de ellas sirve para marcar las zonas de la playa que están, o han estado alguna vez, por debajo del nivel de agua y así poder tratarlas como si estuvieran mojadas. La otra sirve para guardar la distancia que hay entre el nivel de playa y el agua, que se utilizará para la espuma.

Todos los cálculos se hacen en compute shaders en la GPU, cuyos threads se organizan de manera similar a la comentada en el apartado anterior. En este caso hay un compute shader inicial para el viento y dos que se ejecutan cada frame para la posición y las normales del agua.

2. 3. 3. Planeta

El planeta está representado por una esfera de triángulos que va rotando sobre sí misma y su modelado consta de dos partes, el cálculo de la superficie, que es estática, y el de las nubes, que son dinámicas.

Para obtener la superficie utilizaremos una tabla de colores que represente cada tipo de terreno: agua, desiertos, praderas, montañas, etc. Se empieza por utilizar un fractal con 8 octavas de ruido de Perlin en 3 dimensiones que nos da la altura, la cual nos sirve para separar los océanos de los continentes. A continuación se usa otro fractal de 6 octavas que, junto con la altura previamente calculada y la latitud del punto, nos sirve para obtener el color que representa nuestro tipo de terreno en función del clima. Finalmente, para evitar que las zonas sean demasiado uniformes y queden poco realistas, utilizamos un fractal de 5 octavas para motear y distorsionar ligeramente el color. El resultado puede verse en la figura 19.

Esta parte se hace al inicio de la aplicación, ya que no varía, en un compute shader. Como la esfera está organizada en paralelos y meridianos, existiendo vértices en los puntos en los que se cortan, disponemos los threads de manera similar, usando la latitud y la longitud para que cada vértice sea tratado en un hilo distinto.

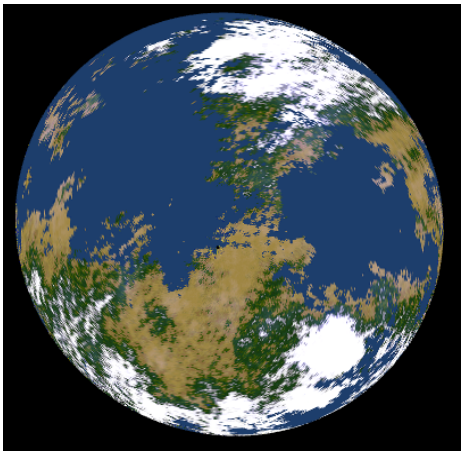


Figura 19: Superficie del planeta fractal

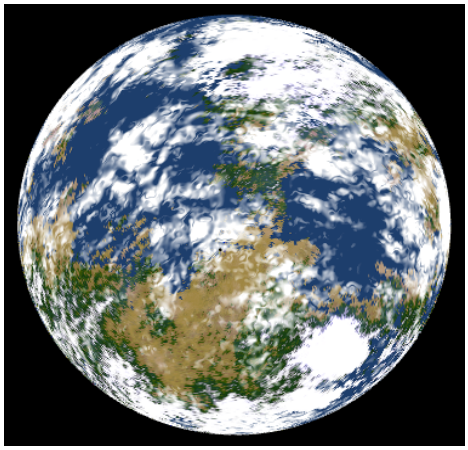


Figura 20: Planeta con nubes

El cálculo de las nubes se hace cada instante puesto que cambia con el tiempo. Se usa un fractal de 6 octavas con ruido de Perlin en 3 dimensiones distorsionado, igual que el viento que empuja el agua. Este valor se usa para interpolar entre el blanco de las nubes y el color ya calculado en el paso anterior, obteniendo así cielos despejados o zonas cubiertas, como se observa en la figura 20.

A diferencia de todos los cálculos comentados hasta ahora este último se hace directamente en un pixel shader por dos motivos. El primero es que la esfera está muy alejada, podríamos decir que fuera del planeta que contiene nuestro paisaje, y por eso no se ilumina de la misma forma, por lo que el color final del pixel es directamente el calculado. La segunda razón es que en el pixel shader solo es visible una porción del planeta, por lo que el número total de operaciones para obtener el fractal es menor.

2. 3. 4. Objetos no procedurales

Para recrear el cielo de nuestra escena se usa una esfera, siempre centrada alrededor de la cámara, a la cual se le aplica una textura que representa un fondo de cielo algo nublado con una cordillera de montañas rodeando el paisaje [17]. Podemos verlo en la figura 21. Este objeto tampoco interacciona con la luz de la manera habitual, por lo que no necesitamos calcular las normales.

También se han añadido varias palmeras para tener una referencia del tamaño de la escena [18]. Tanto los modelos como las texturas aplicadas han sido obtenidos de una página externa[1]. Las texturas que se aplican a las hojas tienen partes que deben ser transparentes, por lo que ha sido necesario configurar la pipeline para usar el canal alpha [19].

1 http://www.loopix-project.com

Figura 21: Cielo

Además hay unos pocos pájaros, diez, para completar el fondo. Son objetos muy sencillos, apenas 16 vértices en el mismo plano, y con posiciones distintas de las alas que se van intercambiando en el tiempo para crear la animación de vuelo, como se muestra en la figura 22.

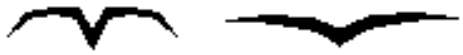


Figura 22: Animación de los pájaros

Cada pájaro tiene información de 3 posiciones, incluida la inicial, obtenidas de manera aleatoria y que marcan su camino. Sabiendo la posición actual y la próxima podemos hacer que se muevan por el paisaje en una linea recta que cambia cuando el pájaro alcanza su meta. Debido a su sencillez tampoco es necesario calcular sus normales.

2. 3. 5. Resto de los elementos de la escena

Además de los objetos que se ven hay otros elementos que componen la escena y que pasamos a comentar en este apartado.

- **Cámara:** por supuesto es necesaria una cámara para observar la escena. Es posible usar el teclado para girarla y desplazarla. También se han precalculado dos trayectorias para observar la escena completa.

- **Iluminación:** la escena está iluminada por una luz direccional que simula la procedente de un sol lejano, además de existir una luz ambiental para evitar zonas de total oscuridad. De la misma manera que con la cámara es posible cambiar la dirección de la luz utilizando el teclado. Además en el pixel shader se usa el modelo de iluminación de Phong, que usa las 3 componentes típicas para calcular la interacción de la luz con un objeto: luz ambiental, luz difusa y luz especular.

- **Materiales y texturas:** las montañas, la playa, diferenciada en función de las zonas secas y las mojadas, y el agua, con zonas de espuma, están formadas por materiales distintos que sirven para definir el color de cada objeto y algunos factores que indican como interaccionan con la luz. Además las montañas y la playa tienen texturas, descargadas de páginas gratuitas[2] [3] [4], que sirven para definir el color de cada píxel y para aplicar bump mapping, modificando el valor de la normal, y obtener mayor realismo [20].

- **Sombras:** el sombreado de la escena se hace con un sencillo mapa de sombras que se calcula cada vez que movemos la luz [21]. Posteriormente, cuando estamos renderizando la escena y queremos ver si un pixel está iluminado o no solo hay que acceder a dicho mapa. Para evitar que queden sombras muy marcadas también se utiliza el valor en los píxeles vecinos, interpolando y suavizando así los bordes [22].

- **Sonido:** además de los componentes visuales se han añadido un par de sonidos que representan el mar y las gaviotas. Para esto se ha utilizado la API XAudio2 de Microsoft, usada para cargar los sonidos .wav, obtenidos de internet[5], y reproducirlos en un bucle infinito mientras dure la aplicación [23].

2. 3. 6. Detalles de la geometría

Salvo los pájaros todos los objetos mostrados por pantalla se definen usando listas de triángulos. Estas constan de dos partes, una lista de vértices, que se guarda en un "vertex buffer", y una lista de punteros a los vértices, guardados en un "index buffer". De esta manera cada 3 índices tenemos definido un nuevo triángulo y no es necesario repetir los vértices que formen parte de varios polígonos a la vez.

El caso de los pájaros es especial porque son objetos muy sencillos. En este caso solo definimos los vértices, guardados en un "vertex buffer". Los 3 primeros vértices sirven para definir el primer triángulo y a partir de ese momento cada nuevo vértice suma otro triángulo, formado por este nuevo vértice y los 2 inmediatamente anteriores.

A continuación, en la tabla 1, vamos a detallar el número de vértices, índices y triángulos de cada uno de los objetos que acabamos de mencionar.

2 http://www.texturex.com
3 http://www.blendswap.com
4 http://www.colorburned.com
5 http://www.soundjax.com

Elemento	Vértices	Índices	Triángulos
Montañas	324.900	1.942.566	647.522
Playa	324.900	1.942.566	647.522
Agua	324.900	1.942.566	647.522
Pájaro (x10)	16	0	14
Planeta	152.202	955.200	318.400
Cielo	82	480	160
Palmera 1	1.008	1.008	336
Palmera 2	1.008	1.008	336
Palmera 3	1.008	1.008	336
Palmera 4	1.008	1.008	336
Palmera 5	276	276	92
Total:	**1.131.452**	**6.787.686**	**2.262.702**

Tabla 1: Geometría de la escena

En esta tabla podemos observar que la mayoría de polígonos se encuentran en los planos que se utilizan para crear los fractales. Esto es así porque necesitamos suficientes triángulos para añadir el detalle, ya que no subdividimos la geometría como se hace en otras aproximaciones para generar objetos procedurales [24]. Como en este proyecto se pretende simplemente usar la GPU y comprobar sus capacidades el elevado número de polígonos no importa, pero en caso de que necesitáramos obtener un framerate estable suficientemente alto, como en un videojuego, podría ser importante reducir el número de triángulos [25].

Creando un fichero .obj desde la aplicación y usando un software externo podemos reducir el número de vértices y comprobar los resultados visualmente, algo que se muestra en las figuras 23 - 26 con los planos de la montaña y la playa.

En estas figuras se aprecia que es posible reducir el número de vértices y polígonos sin que se aprecien diferencias a primera vista. Solo cuando bajamos del 25% de datos originales se empiezan a hacer visibles los triángulos de la malla, aunque en función del detalle que queramos y mediante la aplicación de texturas y materiales, algo que en el ejemplo se ha evitado por sencillez, podríamos reducir más el número de polígonos y seguir teniendo un buen resultado visual.

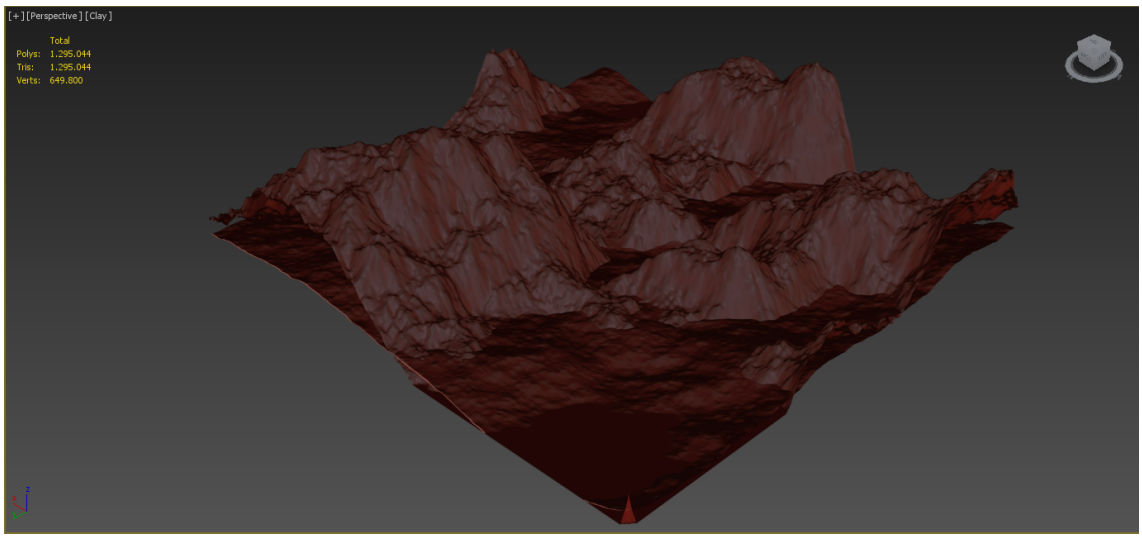


Figura 23: Paisaje 100% - 649.800 vértices

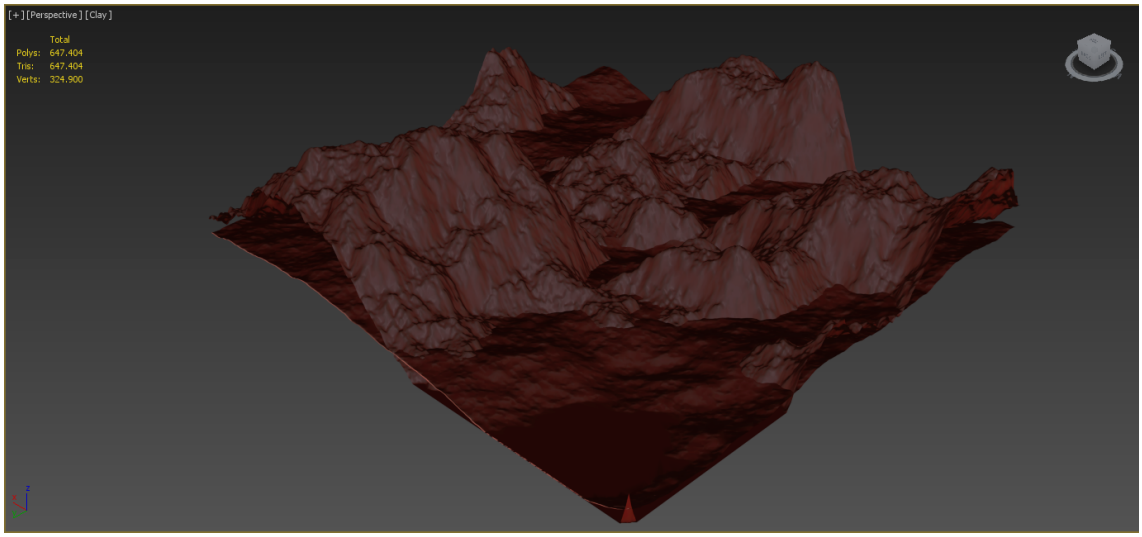


Figura 24: Paisaje 50% - 324.900 vértices

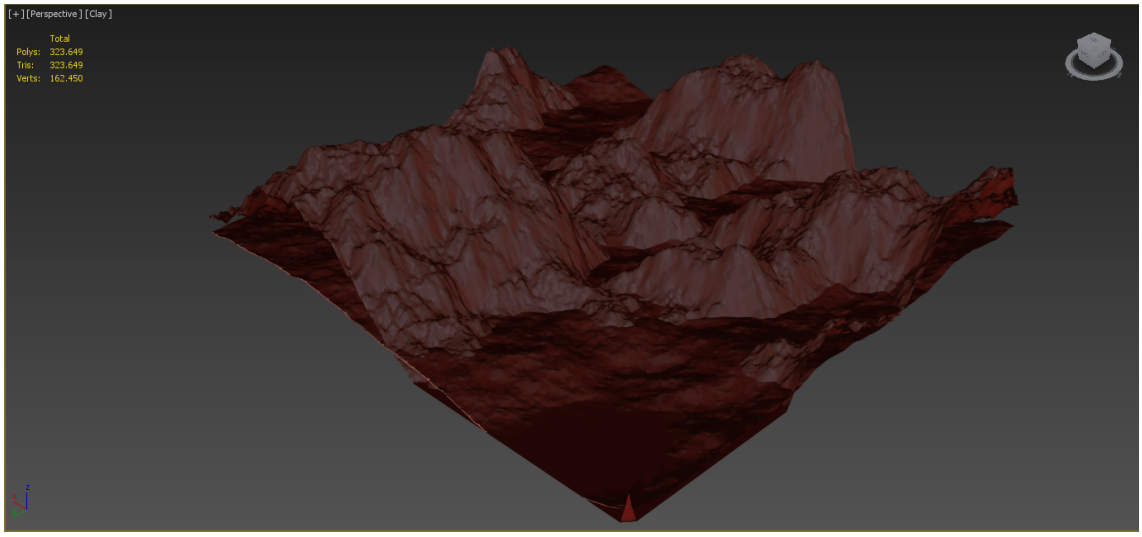


Figura 25: Paisaje 25% - 162.450 vértices

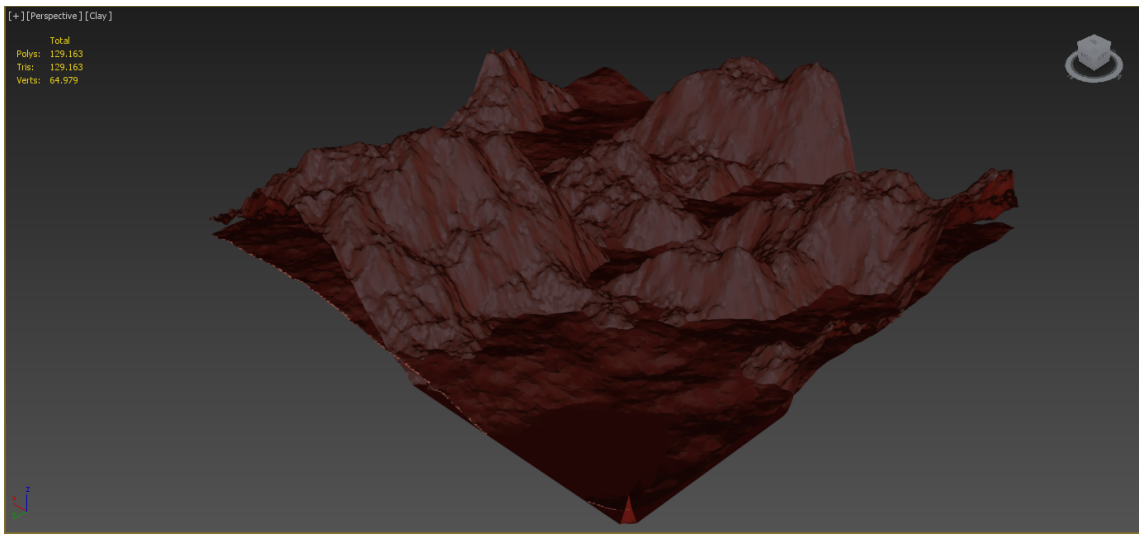


Figura 26: Paisaje 10% - 64.979 vértices

2. 4. Análisis de rendimiento

Una vez finalizada la aplicación se van a tomar varias medidas de tiempos que nos permitan ver el comportamiento de la CPU y la GPU en varios equipos distintos.

2. 4. 1. Definición de la escena

La escena está compuesta por los elementos mencionados en el apartado anterior y, como a la hora de renderizar la geometría se usan varias llamadas "draw" distintas para ejecutar la pipeline gráfica, vamos a mostrar la siguiente información agrupada de esta manera. Para cada llamada se indicarán los polígonos dibujados y el espacio en memoria de la GPU, en bytes, que ocupan todos los datos de entrada necesarios: buffers de vértices e índices, programas shaders, resto de buffers y texturas.

Las tablas 2 - 5 muestran cada llamada en el orden en que se ejecutan.

- Cielo: tabla 2

Triángulos dibujados	160
Tamaño del Vertex Buffer	5.576 B
Tamaño del Index Buffer	1.920 B
Tamaño del Vertex Shader	15.256 B
Tamaño del Pixel Shader	14.512 B
Tamaño de los buffers constantes	192 B
Tamaño de las texturas	66.354.240 B
Total de memoria de video usada	**66.391.696 B**

Tabla 2: Llamada draw 1

- Planeta: tabla 3

Triángulos dibujados	318.400
Tamaño del Vertex Buffer	10.825.736 B
Tamaño del Index Buffer	3.820.800 B
Tamaño del Vertex Shader	19.660 B
Tamaño del Pixel Shader	46.536 B
Tamaño de los buffers constantes	208 B
Tamaño de las texturas	4.288 B
Total de memoria de video usada	**14.717.228 B**

Tabla 3: Llamada draw 2

- Pájaro: tabla 4

Triángulos dibujados	14
Tamaño del Vertex Buffer	1088 B
Tamaño del Index Buffer	0 B
Tamaño del Vertex Shader	15.108 B
Tamaño del Pixel Shader	14.336 B
Tamaño de los buffers constantes	192 B
Tamaño de las texturas	0 B
Total de memoria de video usada	**30.724 B**

Tabla 4: Llamadas draw 3 -12

- Montañas, playa, agua y palmeras: tabla 5

Triángulos dibujados	1.944.002
Tamaño del Vertex Buffer	66.572.544 B
Tamaño del Index Buffer	23.328.024 B
Tamaño del Vertex Shader	23.720 B
Tamaño del Pixel Shader	33.992 B
Tamaño de los buffers constantes	464 B
Tamaño de las texturas	335.229.192 B
Total de memoria de video usada	**425.187.936 B**

Tabla 5: Llamada draw 13

Podemos ver que la última llamada es la que más memoria consume y, por tanto, en la que nos deberíamos fijar si queremos disminuir el espacio a ocupar, no solo en polígonos, como ya hemos mencionado anteriormente, sino también en el número de texturas y su resolución. En este caso los datos de entrada ocupan unos 400 MB, por lo que en GPUs con poca memoria podría aparecer la necesidad de mover datos entre la memoria gráfica y la principal del sistema, con el consecuente tiempo adicional.

Otra posible opción sería dividir la última en varias llamadas distintas, cada una más pequeña que la original, puesto que hay objetos distintos. La pega de esta solución es que añade más operaciones en la CPU, ya que hay cambiar más veces de contexto, y esto puede ser el cuello de botella de la aplicación.

2. 4. 2. Equipos

Se han utilizado 4 equipos distintos para realizar las pruebas, los tres primeros de sobremesa y uno último portátil, que vamos a definir en función de su CPU, GPU y memoria RAM, ya que son los componentes que más pueden influir en los resultados.

En la tabla 6 tenemos las especificaciones de todos ellos.

	Equipo 1	**Equipo 2**	**Equipo 3**	**Equipo 4**
CPU	Intel Core i7-4770	Intel Core i7-2600	AMD Phenom II X4 955	Intel Core i5-430M
Núcleos	4	4	4	2
Threads	8	8	4	4
Frecuencia	3,40 GHz	3,40 GHz	3,20 GHz	2,27 GHz
Caché L1	4 x 64 KB	4 x 64 KB	4 x 128 KB	2 x 64 KB
Caché L2	4x 256 KB	4x 256 KB	4 x 512 KB	2 x 256 KB
Caché L3	8 MB	8 MB	6 MB	3 MB
GPU	NVIDIA GeForce GTX 660	NVIDIA GeForce GTX 570	AMD Radeon HD 6850	ATI Mobility Radeon HD 5470
Núcleos	960	480	960	80
Frecuencia	980 MHz	732-1464 MHz gráficos-shaders	775 MHz	750 MHz
Tamaño VRAM	2 GB DDR5	1,25 GB DDR5	1 GB DDR5	512 MB DDR5
Ancho de banda	144,2 GB/s	152 GB/s	128 GB/s	24,6 GB/s
RAM	8 GB DDR3	6 GB DDR3	8 GB DDR3	4 GB DDR3
Ancho de banda	12.800 MB/s	12.800 MB/s	12.800 MB/s	8.500 MB/s

Tabla 6: Equipos de prueba

Debemos comentar la particularidad de la GPU del equipo 2, ya que en ella los procesadores de shaders funcionan a una frecuencia distinta que el resto de la unidad, en este caso el doble.

2. 4. 3. Pruebas

Para realizar las pruebas van a recolectarse datos sobre el tiempo de inicio de la aplicación, con operaciones ejecutadas en la CPU, y de varios tiempos obtenidos de dos vistas distintas de la escena, calculadas en la GPU. Para obtener los datos de la unidad gráfica usamos la interfaz "Query" de Direct3D 11, que permite obtener información sobre el estado de la ejecución de la tarjeta gráfica [26].

Estos son los datos recogidos:

- Tiempo de inicio de la aplicación en segundos (CPU).
- Tiempo total de un frame en milisegundos (GPU).
- Tiempo de cálculo del agua de un frame milisegundos (GPU).
- Tiempo de renderizado de un frame en milisegundos (GPU).

En todos los equipos se harán pruebas usando la CPU y la GPU en conjunto así como con la CPU en solitario, ya que Direct3D permite que un procesador normal simule el funcionamiento de una unidad gráfica. Aunque es posible implementar nuestro propio software de virtualización Microsoft ofrece uno en el propio SDK de DirectX, conocido como Windows Advanced Rasterization Platform (WARP), que es el que usaremos [27].

En cuanto a las imágenes escogidas, se ha elegido una vista aérea del paisaje, que podría encajar en una escena de una película o un videojuego, y otra más alejada donde se ven todos los elementos pero donde también se observan los límites del propio paisaje.

Cada vista está definida por los siguientes factores, que también se obtienen usando la interfaz "Query" :

- Vértices de entrada a la pipeline
- Triángulos de entrada a la pipeline
- Invocaciones de Vertex Shaders
- Invocaciones de Píxel Shaders
- Triángulos dibujados

A continuación, en las figuras 27 y 28, se muestran y especifican ambas escenas:

Figura 27: Escena 1

Vértices de entrada: 6.787.846
Triángulos de entrada: 2.262.702
Ejecuciones de Vertex Shaders: 2.269.388
Ejecuciones de Pixel Shaders: 3.245.677
Triángulos dibujados: 1.886.400

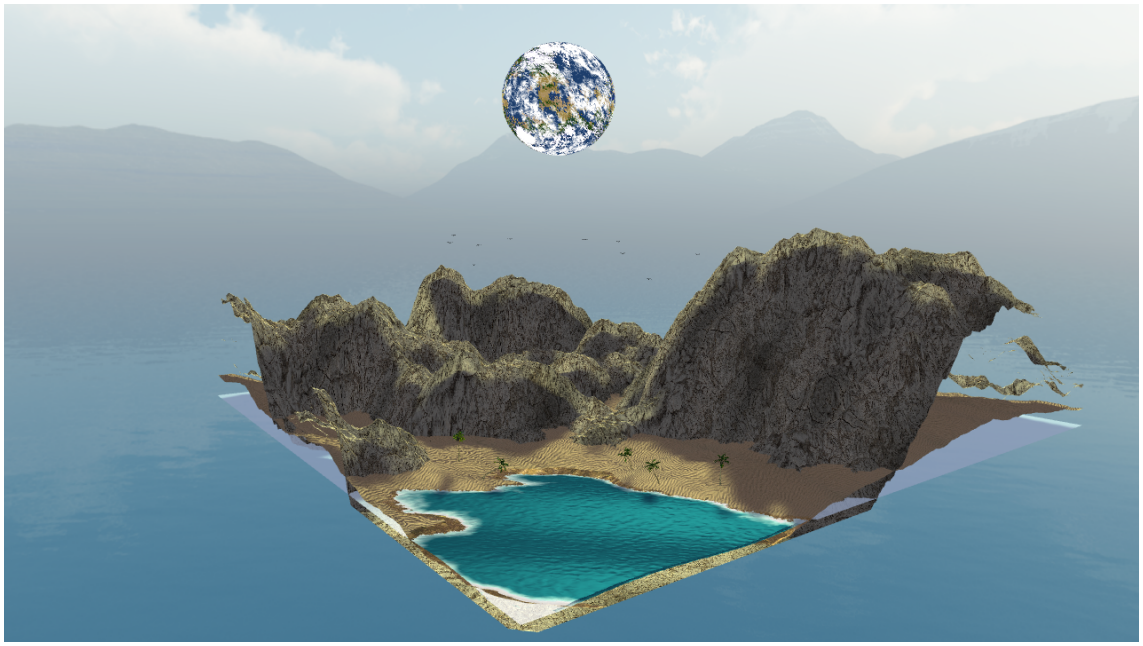


Figura 28: Escena 2

Vértices de entrada: 6.787.846
Triángulos de entrada: 2.262.702
Ejecuciones de Vertex Shaders: 2.269.388
Ejecuciones de Pixel Shaders: 1.915.028
Triángulos dibujados: 2.262.568

El primer factor puede llevar a confusión ya que al detallar la geometría de la escena hemos dicho que ésta se hacía con 1.131.452 vértices, mientras que aquí tenemos 6 veces esa cantidad. Esto se debe a que la pipeline cuenta cada vértice que entra, aunque estén repetidos. De hecho el número de vértices coincide exactamente con el número de vértices de los pájaros más el número de índices del resto de objetos, ya que cada uno de ellos hace referencia a un vértice.

Ambas imágenes tienen una resolución de 1280 x 720, por lo que en total hay 921.600 píxeles. No obstante en las dos vistas se hacen más invocaciones a pixel shaders de las estrictamente necesarias ya que distintos objetos pueden situarse sobre el mismo pixel, aunque finalmente solo uno se muestre.

La segunda escena contiene más triángulos pero estos ocupan menos píxeles de la imagen ya que los objetos se ven más pequeños, por lo que serán necesarias menos invocaciones de píxel shaders. Por eso esperamos que los tiempos obtenidos sean menores, poniendo de manifiesto el tiempo que consume el cálculo del color de un píxel.

2. 4. 4. Tiempos de inicio

En la figura 29 puede verse el tiempo en segundos necesario para iniciar la aplicación, o lo que es lo mismo, crear todos los objetos necesarios durante la ejecución de la pipeline. Se presentan los resultados cuando el dispositivo a ejecutar los comandos Direct3D es la GPU y cuando es la CPU.

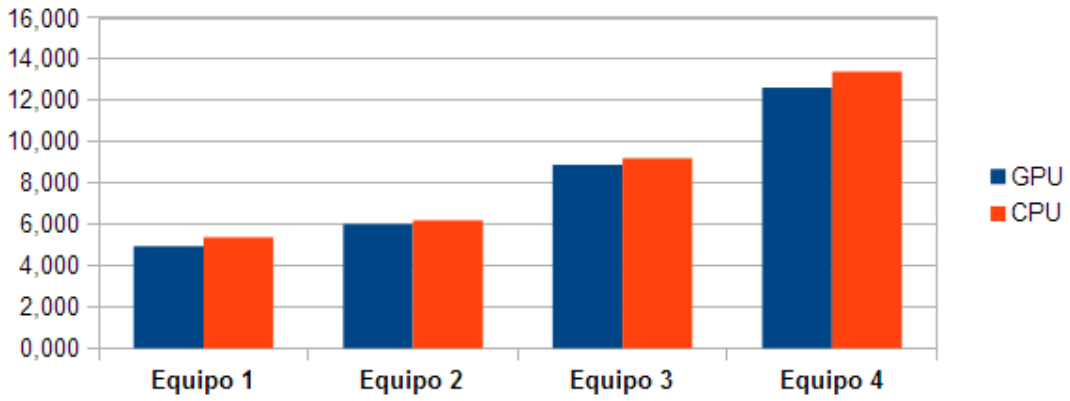


Figura 29: Tiempo de inicio (s)

Observamos que iniciar el dispositivo cuando éste es el procesador principal cuesta algo más por el hecho de tener que cargar el software de virtualización WARP, que ya hemos mencionado.

2. 4. 5. Cálculo y renderizado en la GPU

Podemos ver en la figura 30 los tiempos obtenidos en ambas escenas usando la GPU. Se observa como el cálculo del agua se lleva la mayor parte del tiempo del cálculo del frame, siendo algo menos en la segunda escena gracias a las medidas comentadas en el apartado 2.3.2: "no es necesario añadir el máximo detalle en zonas muy alejadas de la

cámara". También podemos comprobar como el menor número de ejecuciones de pixel shaders reduce el tiempo de renderizado.

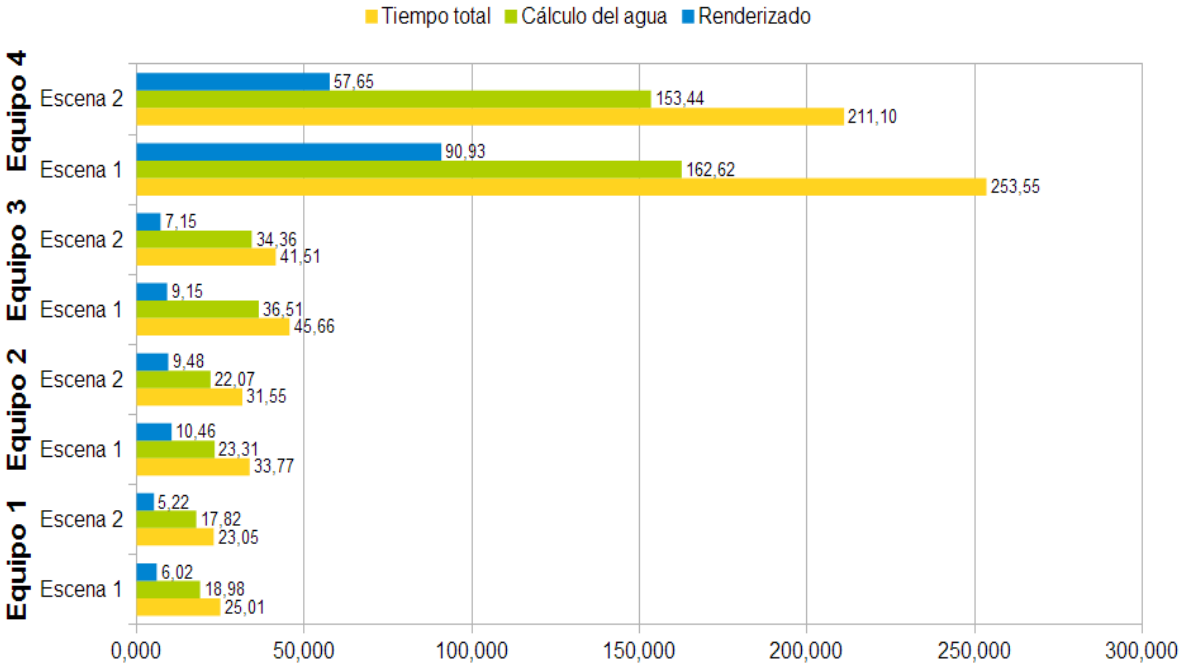


Figura 30: Tiempos en la GPU (ms)

2. 4. 6. Cálculo y renderizado en la CPU

En la figura 31 se presentan los tiempos necesitados por la CPU para realizar las mismas operaciones. Ahora es la etapa de renderizado, con gran diferencia, la que más tiempo consume. Esto es así porque en ella se realizan los cálculos más especializados mientras que el cálculo del agua, de tipo más general, encaja mejor con un procesador corriente.

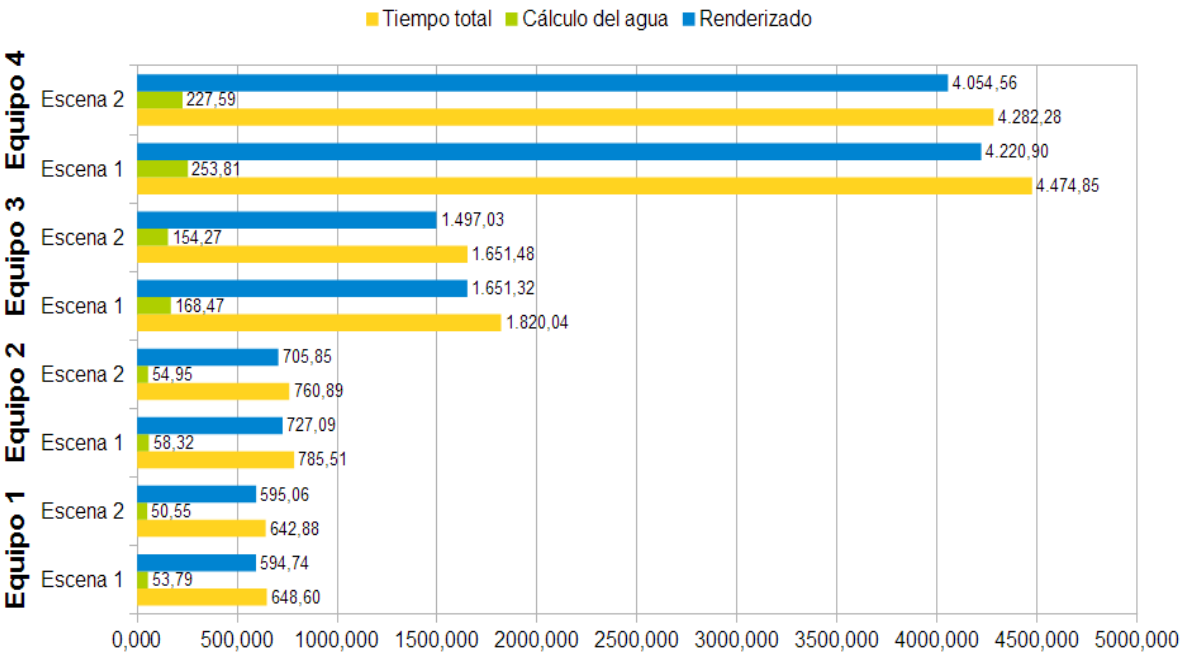


Figura 31: Tiempos en la CPU (ms)

2. 4. 7. Cálculo y renderizado GPU vs CPU

La figura 32 muestra una comparación de los tiempos obtenidos para la escena 1 en la GPU y la CPU. Podemos observar como los tiempos son mucho menores cuando usamos la unidad gráfica gracias a que todos sus núcleos pueden trabajar en paralelo.

No obstante, y como ya hemos mencionado, la mejora es menos patente en los shaders de cálculo, 1,5 – 3 veces más rápido, puesto que no es el objetivo con el que las tarjetas gráficas se diseñan, aunque también es algo que empieza a considerarse por los fabricantes. Es en la pipeline gráfica donde se observa un mayor impacto, consiguiendo un ratio de mejora comprendido entre 50 y 200.

La GPU nos permite obtener un framerate adecuado en los dos primeros equipos, 40 en el más potente y 30 en el segundo, aunque el equipo 3 se queda en solo 20 y el cuarto, el portátil, apenas llega a 4 fps.

Sin embargo la CPU no permite en ningún caso alcanzar un ratio de refresco suficientemente alto como para que no sea evidente al ojo.

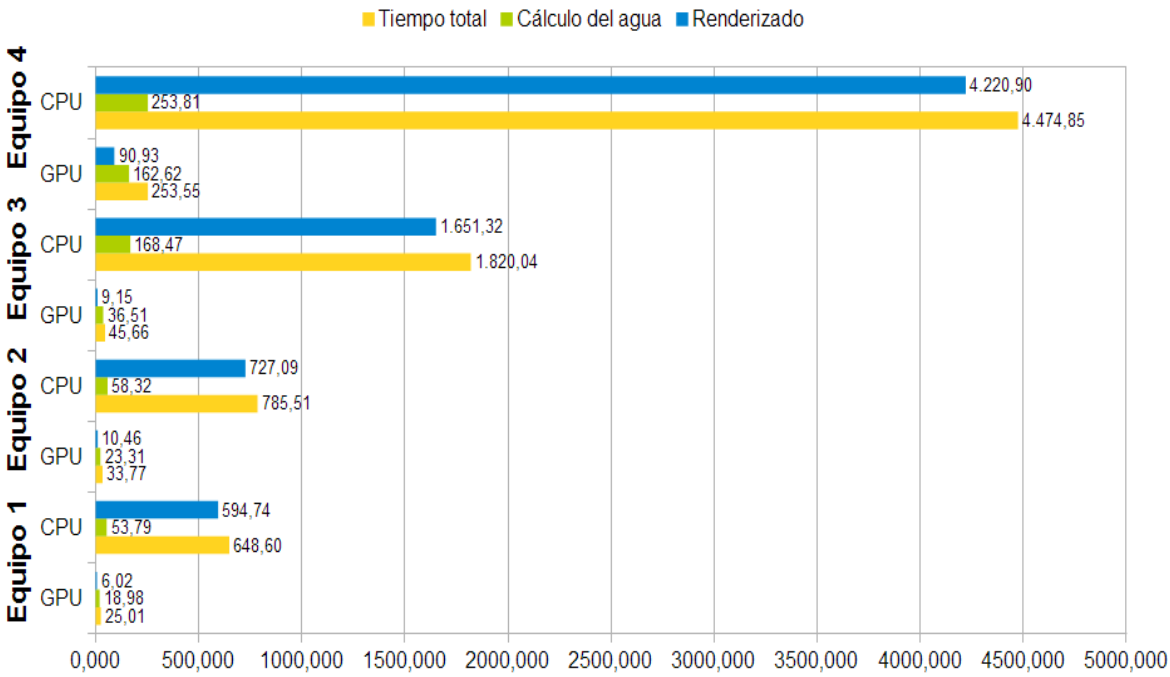


Figura 32: Tiempos escena 1 GPU vs CPU (ms)

Si comparamos entre equipos se ve como las tarjetas gráficas se comportan según lo esperado, siendo la GTX 660 la más potente y la que mejores resultados obtiene. Cabe mencionar que no podemos medir la potencia solo por el número de núcleos y su frecuencia, dados en este mismo apartado, ya que el diseño de cada núcleo varía entre fabricantes y arquitecturas. Es por eso que la GTX 570, con la mitad de núcleos que la HD 6850, aunque al doble de frecuencia, se comporta significativamente mejor en la etapa de cálculo. La tarjeta gráfica del portátil es la que peores resultados da debido a su bajo número de núcleos y su menor frecuencia. Incluso el tamaño de la memoria influye, ya que sus escasos 512 megas obligan a que aumente el flujo de datos entre CPU y GPU.

En cuanto a la CPU, podríamos hacer una lectura similar de los resultados.

2. 5. Integración en UDK

Con objeto de demostrar que el uso de shaders es de inmediata aplicación al mundo de los videojuegos, parte del trabajo realizado se ha integrado en una escena creada con el entorno de desarrollo del motor Unreal.

2. 5. 1. Ruido y materiales

Para usar nuestro ruido de Perlin debemos utilizar el editor de materiales, que usa una red de nodos para crear cada material [28]. Este editor ofrece varios tipos de nodos, cada uno con un código HLSL asociado y que se van uniendo hasta formar el código final de los shaders [29]. Entre estos tipos de nodos existe uno que nos permite escribir nuestras propias instrucciones en HLSL y que se combina con el resto de la misma manera. Hay que tener en cuenta que este editor tiene sus particularidades y no es tan

simple como copiar y pegar desde nuestra aplicación. Para empezar solo pueden construirse vertex y pixel shaders, no es posible utilizar compute shaders. Esto nos impide precalcular y guardar algunos datos, de manera que los cálculos se repiten cada pasada. Además cada nodo personalizado solo puede contener una función, por lo que las llamadas a otras funciones deben sustituirse por el código entero o hacerlas en otros nodos anteriores y propagar los resultados hacia delante. También debemos mencionar que el editor solo acepta texturas con tamaños que sean potencia de dos y que sigan el formato clásico RGBA, por lo que cualquier otro tipo de textura debe ser adaptado.

Tras estas consideraciones pasamos a mostrar nuestra implementación del ruido de Perlin en 3 dimensiones. Aprovechando que el editor ofrece un recuento de las instrucciones de cada material se ha hecho una versión que tiene los datos necesarios para el cálculo, permutación y gradientes, como variables dentro del propio código y otra con ellos guardados en texturas. En la figura 33 se observan los resultados obtenidos.

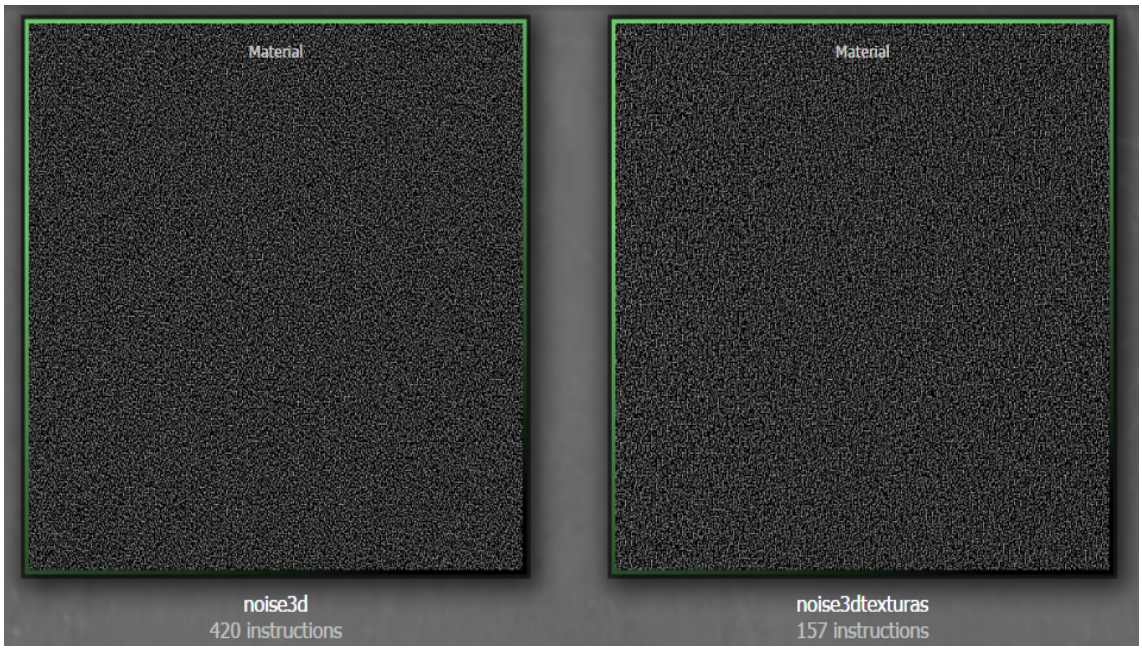


Figura 33: Ruido de Perlin en UDK

Podemos comprobar como el uso de texturas agiliza la ejecución de nuestro código, en este caso se reduce en 263 instrucciones, menos de la mitad que en el original. Aunque también habría que tener en cuenta el tiempo necesario para procesar cada una, es un buen indicador de la complejidad y el tiempo total que se necesitará. Esta mejora se consigue gracias al hardware especializado para el manejo de texturas que las GPU actuales tienen debido a que se usan muy a menudo.

2. 5. 2. Escena

Tras la implementación, para obtener cualquier fractal que queramos integrar en nuestra escena solo hay que crear un material que sume varias octavas de ruido y aplicarlo a un objeto. Como ejemplo se ha decidido usar el fractal del planeta aplicado a una bola del mundo que va rotando y cuyo modelo se ha conseguido en internet[6], que se ha integrado

6 http://www.3delicious.net/

en un mapa ya existente y que viene con el motor: Epic Citadel, usado por Epic como demo técnica. El resultado final puede verse en la figura 34.

Figura 34: Escena UDK

3. CONCLUSIONES

En este último apartado de la memoria se analizan los resultados obtenidos y si se han cumplido los objetivos, se indican posibles líneas de trabajo a seguir en el futuro y se realiza una valoración personal del PFC.

3. 1. Resultados obtenidos

Los objetivos marcados al inicio del proyecto se han conseguido. Hemos podido crear un paisaje completo usando algoritmos fractales que utilizan ruido de Perlin. Esto nos ha permitido comprender como se implementan y utilizan distintos tipos de ruido y el potencial de algoritmos fractales para la generación de elementos naturales.

Además se han estudiado las características de Direct3D, tanto a nivel de gráficos como de cálculo general, viendo así como se organiza la API, su flexibilidad y su relación con los elementos del hardware.

Los resultados obtenidos en el análisis de rendimiento nos han ofrecido una visión de la potencia de las unidades gráficas y nos han permitido comprobar de primera mano por qué su uso para cálculo general ha cobrado fuerza y así lo sigue haciendo cada día que pasa.

También hemos conseguido implementar nuestro algoritmo de ruido dentro del entorno de desarrollo del motor Unreal, demostrando que los conocimientos adquiridos pueden adaptarse y aprovecharse en casi cualquier otro tipo de aplicación, en este caso del mundo de los videojuegos.

3. 2. Líneas futuras

Tras finalizar el proyecto surgen ideas sobre como se podría continuar trabajando en el mismo ámbito en el futuro.

Una de estas ideas es el uso de la pipeline de teselado, característica nueva en Direct3D 11, en la generación del paisaje. Ya que las tarjetas gráficas modernas implementan la división de polígonos por hardware podría ser interesante reducir la resolución de la malla a la hora de calcular los fractales, disminuyendo así el tiempo necesario, algo que se notaría especialmente en los objetos dinámicos como el agua, que deben calcularse constantemente. Con la malla subdividida podríamos añadir más detalle usando algún mapa de desplazamiento, que variaría en función de la distancia y así poder ahorrar espacio en la memoria de la GPU.

También sería posible mejorar el rendimiento de los compute shader. En el caso del cálculo de las posiciones cada vértice se trata de manera aislada y no hay posibilidad de mejorar la comunicación con los vecinos, ya que es inexistente, pero cuando calculamos las normales podríamos utilizar grupos de threads, que comparten un espacio en memoria. Esto nos permitiría calcular las normales de un triángulo y guardar la

información en la memoria compartida, reduciendo el número de cálculos repetidos y el tiempo de acceso a los datos necesarios.

Además sería interesante traducir la implementación de nuestro ruido y los fractales al sistema de materiales de UDK, usando siempre nodos predefinidos en vez de escribir nuestro propio código. Aunque la red de nodos final sería más compleja el editor está desarrollado para trabajar con estos y genera código HLSL optimizado, a diferencia del que indiquemos directamente nosotros, que se traduce tal cual a los shaders finales.

3. 3. Valoración personal

La realización de este proyecto me ha permitido aplicar muchos de los conocimientos adquiridos a lo largo de estos 5 años de carrera, suponiendo la culminación de un largo e importante periodo de mi vida, el cual me marcará durante los años que están por llegar.

Además me ha otorgado la oportunidad de estudiar, comprender y trabajar con herramientas que se usan en el mundo actual, concretamente con tecnologías que se usan en el mundo de los videojuegos, donde espero poder trabajar. De especial utilidad creo que van a ser los conocimientos adquiridos sobre Direct3D y, sobretodo, las pipelines gráfica y computacional, ya que es algo más reciente y que cada vez se usa más, por lo que acabar la carrera habiendo usado éstas puede suponer una diferenciación positiva de cara al futuro.

No obstante el hecho de que sean tecnologías propietarias tiene sus pegas. Aunque sean las dominantes en la situación actual del mercado y por eso es importante conocerlas y saber usarlas, la comunidad de software libre, y en concreto la de OpenGL, suele ser mucho más participativa, por lo que es más fácil encontrar documentación y ayuda ante los problemas. La parte positiva de esto es que hemos aprendido a manejarnos con la documentación de Microsoft, que, aunque sea más escueta y en ocasiones parezca escasa, es la oficial y por tanto la que se debería usar como referencia.

Finalmente, ha sido interesante ver el funcionamiento de un motor de videojuegos y como sus componentes trabajan en conjunto para crear el producto final. Es una pena que los kits de desarrollo de motores conocidos y potentes que se ofrecen de manera gratuita no permitan acceder al código del motor. Aunque es algo comprensible ya que las compañías ponen muchos recursos en su desarrollo me hubiera gustado estudiar sus partes internas con más detalle.

4. BIBLIOGRAFÍA

[1] Unity License Comparisons
http://unity3d.com/unity/licenses

[2] CryEngine Overview
http://mycryengine.com/?conid=2

[3] Unreal Engine Features
http://www.unrealengine.com/en/features/

[4]: Jason Zink, Matt Petineo, Jack Hoxley (2011). *Practical Rendering & Computation with Direct3D 11*. CRC Press

[5] The World's First Fractal Movie - "Vol Libre" 1979 - Loren Carpenter
http://www.liveleak.com/view?i=b75_1371314878

[6]: Benoit B. Mandelbrot (1982). *The Fractal Geometry of Nature*. W. H. Freeman and Company

[7] The First Completely Computer-Generated (CGI) Cinematic Image Sequence in a Feature Film
http://www.historyofinformation.com/expanded.php?id=3584

[8] Making Noise
http://www.noisemachine.com/talk1/

[9]: David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley (2002). *Texturing & Modelling: A Procedural Approach. Third edition*. Morgan Kaufman Publishers

[10] The Perlin noise math FAQ
http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html

[11] Procedural Noise Categories
http://physbam.stanford.edu/cs448x/old/Procedural_Noise%282f%29Categories.html

[12]: Matt Pharr, Randima Fernando, Tim Sweeney (2005). *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional

[13] MSDN: Direct3D 11 Graphics
http://msdn.microsoft.com/en-us/library/windows/desktop/ff476080(v=vs.85).aspx

[14] MDSN: HLSL
http://msdn.microsoft.com/en-us/library/windows/desktop/bb509561(v=vs.85).aspx

[15] Dark Secrets of Shader Development or What Your Mother Never Told You About Shaders
http://developer.amd.com/wordpress/media/2012/10/Dark_Secrets_of_shader_Dev-Mojo.pdf

[16] Scape 2: Procedural Basics
http://www.decarpentier.nl/scape-procedural-basics

[17] DirectX 11 Skybox
http://www.braynzarsoft.net/index.php?p=D3D11CUBEMAP

[18] Direct3D 11: Loading Static 3D models
http://www.braynzarsoft.net/index.php?p=D3D11OBJMODEL

[19] DirectX 10: Blending and Alpha Blending
http://takinginitiative.wordpress.com/2010/04/09/directx-10-tutorial-6-transparency-and-alpha-blending/

[20] Direct3D 11 Normal (Bump) Maps
http://www.braynzarsoft.net/index.php?p=D3D11NORMMAP

[21] DirectX10: Shadow Mapping
http://takinginitiative.wordpress.com/2011/05/15/directx10-tutorial-10-shadow-mapping/

[22]: Randima Fernando, David Kirk (2004). *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Addison-Wesley Professional

[23] MSDN: XAudio2 APIs
http://msdn.microsoft.com/en-us/library/hh405049(v=vs.85).aspx

[24] Procedural Shape Synthesis on Subdivision Surfaces
http://lvelho.impa.br/spd/spd.pdf

[25] Beautiful, Yet Friendly Part 1: Stop Hitting the Bottleneck
http://www.ericchadwick.com/examples/provost/byf1.html

[26] MSDN: ID3DQuery interface
http://msdn.microsoft.com/en-us/library/windows/desktop/ff476578(v=vs.85).aspx

[27] MSDN: Windows Advanced Rasterization Platform (WARP) Guide
http://msdn.microsoft.com/en-us/library/windows/desktop/gg615082(v=vs.85).aspx

[28] Material Editor User Guide
http://udn.epicgames.com/Three/MaterialEditorUserGuide.html

[29] Materials Compendium
http://udn.epicgames.com/Three/MaterialsCompendium.html

II ANEXOS

ANEXO A: GESTIÓN DEL PROYECTO

En este anexo se explican la metodología y planificación seguidas para el desarrollo del proyecto y las herramientas utilizadas.

A. 1. Planificación

El desarrollo se ha dividido en 5 fases distintas. La primera ha sido el estudio de las herramientas y técnicas que íbamos a usar.

El siguiente paso ha sido implementar la creación del paisaje fractal usando C++ y el entorno de desarrollo Visual Studio 2013. Se ha seguido un desarrollo iterativo e incremental que ha permitido partir de una idea básica de lo que queríamos hacer, el paisaje, para ir refinando el objetivo conforme avanzábamos y añadíamos características. Además de esta manera podemos modificar los elementos ya existentes conforme agregamos nuevos. El límite de los detalles adicionales implementados viene marcado por el tiempo disponible, ya que también debíamos finalizar el resto del proyecto antes de la fecha límite marcada y podríamos haber seguido añadiendo más elementos a la escena.

Posteriormente se ha estudiado el desempeño de la aplicación en la CPU y la GPU, analizando el rendimiento en ambos casos.

A continuación se ha implementado el ruido de Perlin en el entorno UDK para poder crear un fractal. También hemos seguido un desarrollo iterativo e incremental, aunque con pocas iteraciones y más cortas que en la creación de la escena principal.

Por último se ha escrito la memoria y preparado la presentación del proyecto.

En la tabla 7 pueden verse las distintas tareas planificadas, su duración y el tiempo en horas dedicado para su finalización.

Descripción	Fecha inicio	Fecha fin	Dedicación (h)
Estudio de técnicas y herramientas	1/09/2013	8/10/2013	100
Creación de la escena	9/10/2013	16/02/2014	195
Análisis de rendimiento	3/02/2014	10/02/2014	35
Integración en UDK	24/01/2014	5/02/2014	30
Escribir memoria y anexos	1/10/2013	19/02/2014	90
Preparar presentación	21/02/2014	10/03/2014	30
Total	**1/09/2013**	**10/03/2014**	**480**

Tabla 7: Dedicación

A. 2. Herramientas utilizadas

A continuación, en la tabla 8, se indican las herramientas utilizadas en el desarrollo del proyecto junto con una breve descripción.

Herramienta	Descripción
Sistema Operativo	
Windows 7	Sistema operativo utilizado en 3 de los equipos de prueba.
Windows 8	Sistema operativo de un equipo de prueba.
Desarrollo	
Visual Studio 2013	IDE para escribir el código de la aplicación principal, los shaders y su depuración.
Unreal Development Kit (Julio 2013)	Entorno de desarrollo del motor Unreal para integrar el ruido en una escena.
DirectX SDK (Junio 2010)	Bibliotecas para trabajar con DirectX.
Gráficos	
CrazyBump 1.2	Herramienta para generar normal maps a partir de una imagen.
3DS Max 2014	Software de creación de gráficos, usado para leer la malla del paisaje y modificar su resolución.
Vue xStream 9.5	Aplicación para generar las texturas del cubo del cielo.
Microsoft DirectX Texture Tool	Herramienta para agrupar las texturas del cielo en un solo archivo.
Paint	Edición de imágenes.
Gestión	
Google Drive	Copias de seguridad, control de cambios y sincronización entre ordenadores.
Documentación	
LibreOffice 4.2.0 Writer	Memoria del proyecto y anexos.
LibreOffice 4.2.0 Calc	Análisis de rendimiento.
LibreOffice 4.2.0 Impress	Presentación del PFC.
LibreOffice 4.2.0 Draw	Creación de imágenes
DoxyGen 1.8.6	Documentación de la aplicación.
Otros	
FRAPS 3.5.99	Captura de imágenes y video. Control de framerate.

Tabla 8: Herramientas utilizadas

ANEXO B: COMPARACIÓN DE MOTORES

En este anexo se realiza una comparación entre varios motores gráficos de videojuegos, potentes y disponibles para PC de manera gratuita. Aunque existen muchas opciones diferentes solo nos vamos a centrar en 3 de los motores más famosos y utilizados: Unreal Development Kit, CryEngine 3 y la versión gratuita de Unity.

B. 1. Herramientas disponibles

En este apartado se van a comparar las distintas herramientas que ofrecen los motores y sus diferentes características. Para ello éstas serán listadas y se presentará una tabla que permite hacer la comparación de manera rápida y sencilla.

B. 1. 1. Animación

1. Animación basada en esqueletos.
2. Huesos que pueden influenciar cada vértice.
3. Asociación de objetos externos con alguna parte del esqueleto.
4. Transición entre animaciones distintas.
5. Cinemática inversa.
6. Animaciones grupales.
7. Animación morph-target.
8. Animación facial.

Característica	UDK	CryEngine 3	Unity
1	✔	✔	✔
2	4	4	4
3	✔	✔	✔
4	✔	✔	✔
5	✔	✔	X
6	✔	✔	X
7	✔	✔	X
8	✔	✔	X

Tabla 9: Motores. Animación

B. 1 . 2. Audio

1. Control de volumen, atenuación o tono.
2. Algoritmos de compresión y descompresión.
3. Audio 3D.
4. Multi-canal.
5. Control de uso de recursos.
6. Filtros.

Característica	UDK	CryEngine 3	Unity
1	✔	✔	✔
2	✔	✔	✔
3	✔	✔	✔
4	Hasta 5.1	Hasta 7.1	Hasta 7.1
5	✔	✔	✔
6	✔	✔	X

Tabla 10: Motores. Audio

B. 1. 3. Cinemáticas

1. Key-frames para actores y objetos.
2. Manejo de cámaras y otros objetos.
3. Control de bucles, velocidad, audio y partículas.
4. Efectos de post-procesado.

Característica	UDK	CryEngine 3	Unity
1	✔	✔	✔
2	✔	✔	✔
3	✔	✔	✔
4	✔	✔	X

Tabla 11: Motores. Cinemáticas

B. 1 . 4. Networking

1. Soporte para conexión LAN e Internet.
2. Modelo cliente-servidor.
3. Descarga y caché de contenidos automáticas.

Característica	UDK	CryEngine 3	Unity
1	✔	✔	✔
2	✔	✔	✔
3	✔	X	X

Tabla 12: Motores. Networking

B. 1. 5. Editor

1. Integración del resto de herramientas.
2. Gestor de contenidos.
3. Renderizado completo para ver el aspecto final del juego.
4. Posibilidad de jugar desde el editor para comprobaciones rápidas.

Característica	UDK	CryEngine 3	Unity
1	✔	✔	✔
2	✔	✔	✔
3	✔	✔	✔
4	✔	✔	✔

Tabla 13: Motores. Editor

B. 1. 6. Físicas

1. Rigid bodies, soft bodies y ragdoll.
2. Combinación de físicas con animaciones.
3. Simulación de multitudes.
4. Destrucción de escenarios y objetos.
5. Simulación de ropas.
6. Simulación de fluidos.
7. Campos de fuerza.
8. Sistema de partículas.

Característica	UDK	CryEngine 3	Unity
1	✔	✔	✔
2	✔	✔	✔
3	✔	✔	X
4	✔	✔	X
5	✔	✔	✔
6	✔	✔	✔
7	✔	✔	✔
8	✔	✔	✔

Tabla 14: Motores. Físicas

B. 1. 7. Iluminación

1. Iluminación global.
2. Ambient occlusion.
3. Iluminación per-pixel.
4. Luces dinámicas.
5. Luces precalculadas.
6. Reflejos en tiempo real.
7. Sombras dinámicas.
8. Sombras precalculadas.
9. Deferred Lighting.

Característica	UDK	CryEngine 3	Unity
1	✔	✔	X
2	✔	✔	✔
3	✔	✔	✔
4	✔	✔	✔
5	✔	X	✔
6	✔	✔	X
7	✔	✔	X
8	✔	X	✔
9	✔	✔	X

Tabla 15: Motores. Iluminación

B. 1. 8. Inteligencia artificial

1. Grafos de navegación.
2. Generación automática de caminos.
3. Búsqueda de caminos.
4. Obstáculos y restricciones.

Característica	UDK	CryEngine 3	Unity
1	✔	✔	X
2	✔	✔	X
3	✔	✔	X
4	✔	✔	X

Tabla 16: Motores. Inteligencia Artificial

B. 1. 9. Programación

1. Lenguaje de programación usado.
2. Paradigma.
3. Acceso al código del motor.
4. Atributos específicos de las clases para juegos.
5. Sensible a mayúsculas.
6. Herencia.
7. Manejo de excepciones.
8. Sobrecarga de funciones.
9. Sobrescritura de funciones.
10. Sobrecarga de operadores.
11. Sobrescritura de operadores.
12. Recolector de basura y creación/destrucción de objetos de manera automática
13. Comprobación de tipos.

Característica	UDK	CryEngine 3	Unity
1	UnrealScript	C++ (y Lua)	C # (Javascript y Boo)
2	Orientado a objetos	Orientado a objetos	Orientado a objetos
3	Indirecto	Indirecto	Indirecto
4	Estados, tiempo y red	X	X
5	X	✔	✔
6	Simple	Simple y múltiple	Simple
7	X	✔	✔
8	X	✔	✔
9	✔	✔	✔
10	✔	✔	✔
11	X	✔	X
12	✔	X	✔
13	✔	✔	✔

Tabla 17: Motores. Programación

B. 1. 10. Renderizado

1. Direct3D.
2. OpenGL.
3. HDR de 64-bit.
4. Modelo de iluminación.
5. Teselado.
6. Corrección gamma.
7. Normal mapping.
8. Depth of field.
9. Filtros anisotrópicos.
10. Desplazamientos.
11. Atenuación.
12. Niebla.
13. Texturas.
14. Efectos de post-procesado.
15. Distintos niveles de detalle.

Característica	UDK	CryEngine 3	Unity
1	9 y 11	9, 10 y 11	9 y 11
2	Solo la versión IOs	X	✔
3	✔	✔	X
4	Phong	Phong	Phong / Gouraud
5	✔	✔	✔
6	✔	✔	X
7	✔	✔	✔
8	✔	✔	X
9	✔	✔	✔
10	✔	✔	✔
11	✔	✔	✔
12	✔	✔	✔
13	✔	✔	✔
14	✔	✔	X
15	✔	✔	X

Tabla 18: Motores. Renderizado

B. 1. 11. Shaders y materiales

1. Editor de materiales.
2. Factores básicos.
3. Funciones para combinar shaders.
4. Texturas.
5. Subsurface Scattering.
6. Creación automática de shaders en función de la plataforma.
7. Custom shaders.
8. Anti-aliasing.

Característica	UDK	CryEngine 3	Unity
1	Gráfico (nodos)	Gráfico (árbol)	Código
2	Difuso, especular, emisor y opacidad	Difuso, especular, brillo, emisor y opacidad	Difuso, especular, emisor, brillo y transparencia
3	✔	Suma	✔
4	✔	✔	✔
5	✔	✔	X
6	✔	✔	✔
7	HLSL	X	HLSL, GLSL y CG
8	✔	✔	✔

Tabla 19: Motores. Shaders y materiales

B. 1. 12. Terreno

1. Herramientas para creación de terrenos exteriores.
2. Creación de vegetación.
3. Simulación de erosión y efectos climáticos.
4. Creación de ríos.
5. Creación de carreteras y caminos.
6. Distintos niveles de detalle.

Característica	UDK	CryEngine 3	Unity
1	✔	✔	✔
2	✔	✔	✔
3	✔	✔	X
4	X	✔	X
5	X	✔	X
6	✔	✔	✔

Tabla 20: Motores. Terreno

B. 1. 13. Otros

1. Distribución de tareas entre múltiples núcleos.
2. Visual scripting.
3. Creación de menús e interfaces.
4. Documentación disponible.
5. Plataformas soportadas.
6. Editor de vehículos.
7. 3D estereoscópico.

Característica	UDK	CryEngine 3	Unity
1	✔	✔	✔
2	✔	✔	X
3	✔	X	✔
4	✔✔	✔	✔✔✔
5	Windows y IOs	Windows, Xbox 360 y PS3	Windows, Linux, Mac, Web player, Xbox 360, PS3, WiiU, IOs y Android
6	X	✔	X
7	✔	✔	X

Tabla 21: Motores. Otros

B. 2. Imágenes comparativas

A continuación, en las figuras 35 - 46, se presentan algunas imágenes de juegos comerciales o demostraciones técnicas creadas con los motores comparados para así poder comprobar la calidad gráfica que se puede obtener con ellos y ver en funcionamiento las características comentadas.

B. 2. 1. Unreal Development Kit

Figura 35: Muestra UDK 1

Figura 36: Muestra UDK 2

Figura 37: Muestra UDK 3

Figura 38: Muestra UDK 4

B. 2. 2. CryEngine 3

Figura 39: Muestra CryEngine 1

Figura 40: Muestra CryEngine 2

Figura 41: Muestra Cryengine 3


Figura 42: Muestra CryEngine 4

B. 2. 3. Unity

Figura 43: Muestra Unity 1

Figura 44: Muestra Unity 2

Figura 45: Muestra Unity 3


Figura 46: Muestra Unity 4

ANEXO C: LA PIPELINE GRÁFICA

En este apartado se explica el funcionamiento de la pipeline gráfica de Directc3D 11 y las características del lenguaje que se usa para la programación de sus etapas: HLSL.

C. 1. Etapas

En la figura 47 podemos observar la pipeline gráfica de Direct3D 11, siendo las etapas enmarcadas en rectángulos las programables y donde, por tanto, podemos ejecutar nuestro propio código. Las otras, aunque más o menos configurables, harán siempre las mismas operaciones y por ello ofrecen una flexibilidad más limitada.

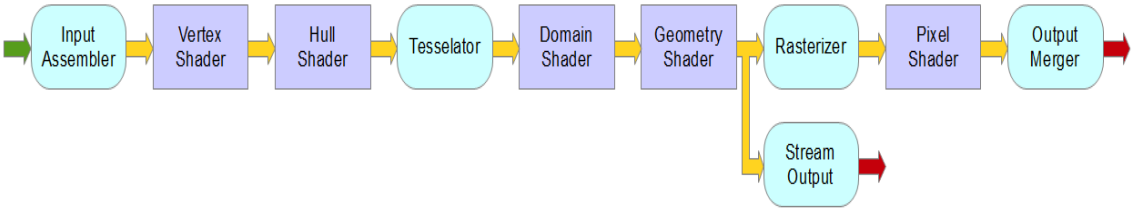


Figura 47: Pipeline gráfica

Además hay que considerar que las etapas de teselado, hull shader, tesselator y domain shader, siempre las 3 juntas, son opcionales y pueden utilizarse según nos convenga en la aplicación. Lo mismo ocurre con el geometry shader y el stream output.

A continuación se realiza un pequeño resumen de las etapas de la pipeline, haciendo hincapié en las programables, mostrando sobre qué trabajan y algunos ejemplos de para qué pueden utilizarse.

C. 1. 1. Input Assembler

La primera etapa es la encargada de construir los vértices que se utilizarán en el resto de la pipeline, así como de las uniones entre ellos.

C. 1. 2. Vertex Shader (programable)

Se ejecuta una vez por cada vértice de la escena y trabaja sobre cada uno de ellos de manera aislada. Entre las operaciones más típicas que realiza está el transformar el sistema de coordenadas del vértice o calcular alguna iluminación sencilla.

C. 1. 3. Hull Shader (programable)

Ésta es la primera de las etapas de teselado que a partir de los vértices iniciales, considerados puntos de control, consigue generar otros nuevos. Se divide en dos funciones distintas, la primera es la encargada de configurar la siguiente etapa y la otra se encarga de hacer cualquier modificación que queramos a dichos puntos.

C. 1. 4. Tesselator
Crea una serie de coordenadas en función de la configuración que hayamos indicado en el paso previo, independientemente de la posición o cualquier otro atributo de los puntos de control.

C. 1. 5. Domain Shader (programable)
Utilizando los puntos de control y las coordenadas generadas en la anterior etapa y con el algoritmo que el programador implemente se crearán los nuevos vértices para ser pasados al resto de la pipeline.

C. 1. 6. Geometry Shader (programable)
En este shader podemos modificar la geometría, vértices y sus conexiones, cambiando sus atributos o incluso añadiendo nuevos puntos. Además tiene otros usos como seleccionar qué parte de la escena no queremos que continúe a las siguientes etapas y así reducir el número de operaciones.

C. 1. 7. Stream Output
El único objetivo de esta etapa es guardar el resultado obtenido hasta ahora en un buffer externo para su futuro uso o inspección. Realmente la etapa no realiza ningún tipo de operación puesto que la copia se hace durante la anterior, el geometry shader o el vertex shader si el primero está desactivado.

C. 1. 8. Rasterizer
En esta etapa la geometría se convierte a un formato adecuado para poder ser presentado en pantalla, es decir, los triángulos se convierten en píxeles. Aquí también se elimina la parte de la escena que no debe ser mostrada en pantalla en función del tamaño de la cámara, su posición o qué parte de los objetos no se verán por no mirar a ésta.

C. 1. 9. Pixel Shader (programable)
El último shader programable es invocado una vez por cada pixel y se encarga de calcular su color final. Por ello es en esta etapa donde se realizan operaciones como aplicar materiales o texturas. También guarda información de la profundidad de los píxeles para poder comprobar a continuación cuales se corresponden con los objetos que están ocultos y cuales deben ser presentados.

C. 1. 10. Output Merger
Finalmente se combina la información de color y profundidad y se calcula el color final que tendrá cada píxel. Estos colores serán los que se guarden y los que se obtengan como salida final de la pipeline.

C. 2. HLSL

High Level Shading Language, HLSL, es un lenguaje derivado de C++ pero con una serie de características particulares para la programación de tarjetas gráficas, como el soporte para vectores o matrices y la inclusión de operaciones típicas con ellos, aunque carece de otras como el uso de punteros o la reserva dinámica de memoria.

Es utilizado en todas las etapas programables de la pipeline, permitiendo la abstracción del hardware que haya debajo. El código escrito, que implementará el algoritmo deseado, será compilado y asociado a la parte de la pipeline correspondiente.

Un shader puede utilizar variables de tipo entero, real y booleano, que también pueden ser usados en vectores de hasta 4 variables, matrices de hasta 4x4 y estructuras definidas por el usuario. Además es capaz de acceder a variables de sistema, muchas generadas automáticamente, como la posición de un píxel, y que pueden ser usadas para compartir información entre etapas o simplemente para calcular algo en nuestro algoritmo.

También es posible acceder a distintos buffers que contienen información intercambiada con la CPU o con otras partes de la pipeline, como podría ser una textura o un mapa de alturas.

ANEXO D: LA PIPELINE DE CÁLCULO

La potencia de cálculo que ofrecen las tarjetas gráficas actuales puede ser aprovechada para realizar operaciones de cálculo general, en especial aquellas que se beneficien de una ejecución en paralelo. En este anexo se explica como Direct3D 11, que incorpora la API DirectCompute para GPGPU, permite explotar estas características a través de la pipeline de cálculo.

D. 1. Etapas

La pipeline de cálculo consta de una sola etapa, el Compute Shader, que se ejecuta de manera totalmente separada de la pipeline gráfica tradicional. Este shader debe escribirse en HLSL, como cualquier otro, y será el encargado de ejecutar el algoritmo que queramos implementar en la GPU.

D. 2. Características

Para comprender como podemos utilizar la GPU para la ejecución de un algoritmo de tipo general y aprovechar su potencia es necesario comentar algunas de sus particularidades.

D. 2. 1. Modelo de threading

El trabajo a ejecutar en la GPU se divide en grupos de threads. Éstos se distribuyen en un array de 3 dimensiones comprendidas entre 1 y 64K, teniendo cada uno un índice para ser identificado. A su vez cada grupo está formado por varios threads, con su propio identificador, distribuidos en otro array de 3 dimensiones comprendidas entre 1 y 64, siendo 1024 el número máximo de threads a ejecutar en cada grupo. Esta disposición se puede observar en la figura 48.

Ya que cada thread ejecutará el mismo shader es necesario que cada uno actúe sobre unos datos distintos. También en la figura 48 es fácil ver la importancia de organizar nuestros threads para aprovechar las características del algoritmo y como se van a dividir los datos.

Para ello disponemos de los índices mencionados anteriormente, tanto el de grupo como el de thread dentro de grupo, así como la posición global del thread y la posibilidad de identificar a qué grupo pertenece. Estos índices contendrán la posición del grupo o el thread en los 3 ejes, así que con toda esta información podemos acceder sin problemas a la porción de los datos que necesitemos.

Por supuesto el número de shaders que se ejecutarán realmente en paralelo dependerá del hardware empleado, pero siempre hay que tratar de ajustar al máximo las propiedades del problema a la organización de los threads.

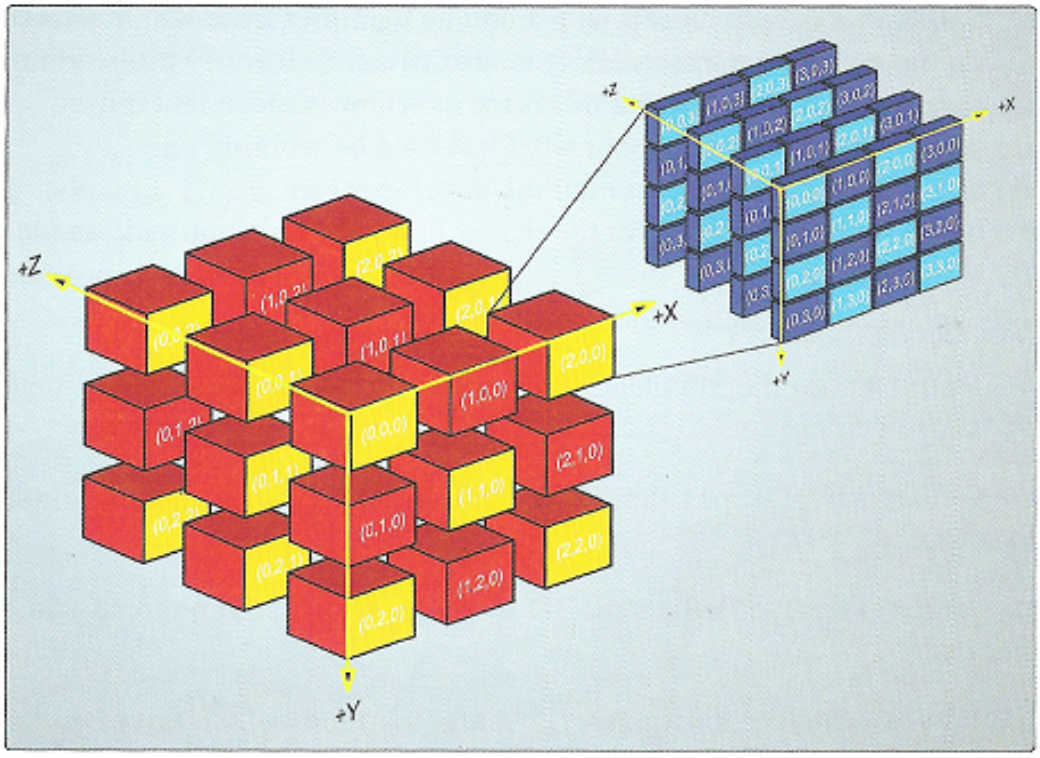


Figura 48: Organización de los threads de un compute shader

D. 2. 2. Memoria

Como en cualquier otro shader es posible acceder a los recursos guardados en distintos buffers y texturas, que estarán dentro la memoria de la GPU, si es posible, o en memoria externa si no caben. Todos los threads tendrán acceso a los mismos recursos, ya sean lecturas o escrituras, por ello habrá que sincronizarlos, como se comenta en el siguiente apartado.

Además todos los threads del mismo grupo pueden compartir hasta 32 KB de memoria dentro de la disponible en la GPU. Esta memoria, llamada "Group Shared Memory" (GSM), también debe ser sincronizada para evitar conflictos y es la manera más rápida de compartir información entre threads.

D. 2. 3. Sincronización

Podemos evitar los problemas de sincronización en ambos tipos de memoria de varias maneras, siendo la más sencilla la que ya hemos comentado, que cada thread trabaje con una parte distinta de los datos.

También podemos utilizar barreras para sincronizar los accesos a memoria de un mismo grupo. Es posible sincronizar la memoria de grupo, la externa o ambas a la vez. Además en los 3 casos se pueden utilizar 2 tipos de barreras distintos. El primero espera a que todas las escrituras pendientes en ese momento acaben antes de seguir, puesto que hay un lapso de tiempo desde que la instrucción se ejecuta en el shader y éste es realmente realizado a nivel hardware. El segundo tipo, además de garantizar lo anterior, espera a que todos los threads del grupo lleguen a la llamada a la función de barrera.

Finalmente, es posible utilizar varias funciones atómicas disponibles que garantizan que su ejecución se completa antes de que cualquier otra operación se vaya a realizar sobre los mismos datos.

También debemos comentar que aunque todas estas maneras de sincronización son útiles solo la primera es deseable, ya que es la única que nos evita tener tiempos ociosos en la GPU y por tanto aprovecha al máximo sus capacidades.

ANEXO E: APLICACIÓN ESTUDIADA

En el presente anexo se trata la implementación de una aplicación para simular fluidos sacada del libro "Practical Rendering and Computation with Direct3D 11". Gracias a ella se han estudiado las características de Direct3D 11 y el uso y configuración de las distintas etapas de la pipeline, tanto de cálculo como de renderizado.

E. 1. Contexto

E. 1. 1. Contexto tecnológico

La simulación y renderizado en tiempo real de fluidos, en este caso agua, suele ser uno de los aspectos más costosos de una escena. Es debido a esta complejidad que puede ser beneficioso usar la potencia de una GPU para realizar los cálculos.

Hacer una representación físicamente perfecta es algo imposible para el hardware actual, pero no es necesaria para conseguir una aproximación que sea eficiente y visualmente capaz de dar la impresión adecuada. En nuestro caso el agua está formada por millones de moléculas pero solo nos interesa ver el comportamiento de la superficie para poder renderizarla. También podemos acotar el número de partículas reduciendo todo a una red cuadriculada de columnas de agua, cuya altura será la de la superficie.

E. 1. 2. Base matemática

Usando estas columnas es posible que el agua fluya de unas a otras si consideramos que las vecinas están conectadas por tuberías virtuales situadas al pie de cada columna. Para calcular este flujo necesitamos saber la presión *Pij* ejercida en el fondo de cada columna, la cual podemos hallar con la altura *Hij* de ésta, la densidad del agua *d*, la gravedad *g* y la presión atmosférica *p*.

$$Pij = Hij * d * g + p$$

La diferencia existente entre presiones de columnas vecinas será la que genere la aceleración *a* del flujo de cada tubería, que podemos calcular puesto que también conocemos el área *c* de ésta y la masa de agua *m* que ya hay en la columna.

$$a_{ij \to kl} = \frac{c * (H_{ij} - H_{kl})}{m}$$

Conociendo la aceleración y suponiendo que el flujo es constante durante un intervalo de tiempo Δt es posible calcular el flujo de agua *Q*.

$$Q_{ij \to kl}^{t+\Delta t} = Q_{ij \to kl}^{t} + \Delta t * (c * a_{ij \to kl})$$

Tras calcular los flujos a través de todas las tuberías virtuales es fácil calcular el cambio en el volumen de agua de una tubería ΔV.

$$\Delta V_{ij} = \Delta t * \sum_{kl \in n_{ij}} \left(\frac{Q^{t}_{ij \to kl} + Q^{t+\Delta t}_{ij \to kl}}{2} \right)$$

Finalmente, calcular el cambio de altura usando el área *A* de la columna de agua es trivial ya que

$$H_{ij} = \frac{V_{ij}}{A_{ij}}$$

E. 2. Diseño

E. 2. 1. Datos

Viendo las consideraciones anteriores llegamos a la conclusión de que para cada columna debemos mantener dos tipos de información: su altura y el flujo de las tuberías.

Como cada columna tiene 8 conexiones pero hay información compartida nos basta con utilizar 4 datos, la tubería de la derecha y las de abajo, como se muestra en la figura 49.

Figura 49: Conexión de columnas de agua

Tenemos en total 5 datos de tipo real por columna que serán guardados en un buffer constante estructurado y que contendrá el estado de la simulación. No obstante no nos basta con usar uno puesto que el estado actual debe ser mantenido hasta haber completado el cálculo del nuevo, así que usaremos 2 que irán intercambiando su papel.

E. 2. 2. Threading

Observando la figura 49 es fácil pensar que cada columna puede asociarse con un thread, que a su vez serán organizados en grupos cuadrados, en este caso de dimensión 16x16. Puesto que los threads vecinos comparten información es posible usar la memoria compartida de grupo para almacenar estos datos y mejorar el acceso a ellos. El problema aparece en los threads del borde del grupo, que deben comunicarse con otros que están fuera del grupo y, por tanto, fuera de la memoria compartida.

Para evitar tener que escribir y leer todos los datos en la memoria de la GPU en cada pasada añadimos una fila al perímetro del grupo, quedando 18x18, que se solapa con los grupos adyacentes. Estos threads cargarán las alturas correspondientes y realizarán los cálculos de flujo, que si podrán ser guardados en memoria de grupo, pero no actualizarán el estado de la simulación, ya que de eso se encargarán los threads correspondientes de los grupos vecinos. En resumen, para evitar tener que hacer más accesos a memoria lenta repetimos algunos cálculos en distintos grupos, que suponen una penalización menor.

E. 3. Aplicación

Con la teoría y el diseño en mente solo queda la implementación. Para empezar, analizaremos la aplicación. La original utilizada en el ejemplo del libro usa una serie de librerías creadas por los autores que no permiten ver las llamadas a Direct3D claramente, por ello se ha escrito una nueva donde se pueden apreciar.

```
// Incluimos los archivos necesarios
#include <windows.h>
#include <d3d11.h>
#include <d3dcompiler.h>
#include <directxmath.h>


// Incluimos las librerias necesarias
#pragma comment (lib, "d3d11.lib")
#pragma comment (lib, "d3dcompiler.lib")

using namespace DirectX;


// Variables globales
HWND hWnd;                                  // Controlador de la ventana
D3D11_VIEWPORT viewport;                    // Vista
IDXGISwapChain *swapchain = NULL;           // Swap chain
ID3D11Device *dev = NULL;                   // Direct3D device interface
ID3D11DeviceContext *devcon = NULL;         // Direct3D device context
ID3D11RenderTargetView *rtv = NULL;         // Render Target View
ID3D11VertexShader *pVS = NULL;             // Vertex Shader
ID3D11PixelShader *pPS = NULL;              // Pixel Shader
ID3D11ComputeShader *pCS = NULL;            // Compute Shader
ID3D11Buffer *pVBuffer = NULL;              // Buffer de los vértices
ID3D11Buffer *pIBuffer = NULL;              // Buffer de índices
ID3D11InputLayout *pLayout = NULL;          // Formato de entrada a la pipeline
ID3D11RasterizerState *pRState=NULL;        // Configuración de la etapa de
rasterizado
ID3D11Buffer *pCBufferMat = NULL;           // Constant buffer de las matrices de
transformacion
ID3D11Buffer *pCBufferDis = NULL;           // Buffer con la info de threads
utilizados
ID3D11Buffer *pCBufferDisTime = NULL;       // Buffer con la info de threads y de
tiempo
ID3D11Buffer *pWaterSim1 = NULL;            // Buffer para guardar el estado
actual de la simulación
ID3D11Buffer *pWaterSim2 = NULL;            // Buffer para guardar el próximo
estado de la simulación
ID3D11ShaderResourceView *pSRVWater= NULL; // Vista para leer la información de
la simulación
ID3D11UnorderedAccessView *pUAVWater= NULL;// Vista para leer/escribir la
información de la simulación
int numVertices = 0;                        // Número de vertices a dibujar
int numIndices = 0;                         // Número de indices
XMMATRIX myWorld;                           // Matriz del mundo
XMMATRIX myView;                            // Matriz de la vista
XMMATRIX myProjection;                      // Matriz de proyección
int ThreadGroupsX = 16;                     // Número de thread groups, x
int ThreadGroupsY = 16;                     // Número de thread groups, y
const int DispatchSizeX = 16;               // Número de threads por grupo, x
const int DispatchSizeZ = 16;               // Numeo de threads por grupo, Z
int SizeX= ThreadGroupsX*DispatchSizeX;     // Número de vértices en X
```

```
int SizeZ= ThreadGroupsY*DispatchSizeZ;    // Número de vértices en Z
LARGE_INTEGER tiempo;                      // Tiempo transcurrido desde la última
actualización de la simulacion
LARGE_INTEGER freq;                        // Frecuencia de la CPU, usada para
calcular el tiempo


// Estructuras a usar
// Vértice
struct VERTEX {
    float x, y, z; // Posición
};

// Información de un punto de la simulación
struct GridPoint
{
    float height;
    float flow[4];
};

// Buffer constante de las matrices de transformación
struct ConstantBufferMatrix
{
    XMMATRIX World;
    XMMATRIX View;
    XMMATRIX Projection;
};

// Buffer constante con la información de threads
struct ConstantBufferDis
{
    float DispatchSize[4];
};

// Buffer constante con al información de threads y tiempo
struct ConstantBufferDisTime
{
    float TimeFactor;
    float padding[3]; // Los buffers constantes deben estar alineados a 16bytes,
así que rellenamos
    float DispatchSize[4];
};


// Declaración de las funciones a usar
HRESULT InitWindow(HINSTANCE hInstance, int nCmdShow );
HRESULT InitDevice();
void InitPipeline();
void InitScene();
void InitSim();
void InitCamera();
LRESULT CALLBACK WindowProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
lParam);
void Calculate();
void Render();
void CleanD3D();


// Punto de entrada a la aplicación
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine,
int nCmdShow)
{
```

```
    // Creación de la ventana
    if( FAILED (InitWindow(hInstance,nCmdShow)) )
        return 0;

    // Creación del dispositivo y swap chain
    if( FAILED (InitDevice()) )
        return 0;

    // Inicializar la pipeline
    InitPipeline();

    // Inicializar la escena
    InitScene();

    // Iniciar la simulación
    InitSim();

    // Inicializar la cámara
    InitCamera();

    // Inicio de la información del timer
    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&tiempo);

    // Bucle principal
    MSG msg; // Mensaje de los eventos de Windows
    while(TRUE)
    {
        // Comprobar si hay mensajes esperando
        if(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
            // Traducir al formato de Windows
            TranslateMessage(&msg);

            // Mandar mensaje a WindowProc para procesarlo
            DispatchMessage(&msg);

            // Comprobar si hay que salir de la aplicación
            if(msg.message == WM_QUIT)
                break;
        }

        // Calcular un paso de la simulacion
        Calculate();

        // Renderizar la escena
        Render();
    }

    // Liberar recursos
    CleanD3D();

    // Devolver a Windows la siguiente info
    return msg.wParam;
}


// Función para iniciar la ventana
HRESULT InitWindow( HINSTANCE hInstance, int nCmdShow )
{
    WNDCLASSEX ventana; // Clase de la ventana
    ZeroMemory(&ventana, sizeof(WNDCLASSEX)); // Limpiamos la variable ventana
```

```
    // Llenamos la clase con la info correspondiente
    ventana.cbSize = sizeof(WNDCLASSEX);            // Tamaño de la clase
    ventana.style = CS_HREDRAW | CS_VREDRAW;        // Estilo
    ventana.lpfnWndProc = WindowProc;               // Puntero al controlador de
eventos
    ventana.hInstance = hInstance;                  // Instancia de la aplicación
con la ventana
    ventana.hCursor = LoadCursor(NULL, IDC_ARROW);  // Cursor
    ventana.hbrBackground = (HBRUSH)COLOR_WINDOW;   // Color del fondo
    ventana.lpszClassName = L"ClaseVentana";        // Nombre de la clase

    // Registramos la clase
    if (! RegisterClassEx(&ventana))
        return E_FAIL;

    // Creamos una ventana y obtenemos el controlador
    RECT wr = {0, 0, 800, 600}; // Indicamos el tamaño de la zona de la ventana
disponible para nuestra aplicación
    AdjustWindowRect(&wr, WS_OVERLAPPEDWINDOW, FALSE); // Obtenemos el tamaño
completo de la ventana

    hWnd = CreateWindowEx(NULL, // Estilo
                          L"ClaseVentana",     // Nombre de la clase ventana
                          L"Agua-Columnas",    // Titulo a mostrar
                          WS_OVERLAPPEDWINDOW, // Estilo
                          100,                 // Posicion x de la ventana
                          100,                 // Posicion y de la ventana
                          wr.right - wr.left,  // Ancho
                          wr.bottom- wr.top,   // Altura
                          NULL,                // Ventana padre, NULL
                          NULL,                // Uso de menus, NULL
                          hInstance,           // Instancia de la aplicación
con la ventana
                          NULL);               // Multiples ventanas, NULL

    if (!hWnd)
        return E_FAIL;

    // Mostrar ventana por pantalla
    ShowWindow(hWnd, nCmdShow);

    return S_OK;
}


// Función para crear el dispositivo y la swapchain
HRESULT InitDevice()
{
    // Información de la swap chain
    DXGI_SWAP_CHAIN_DESC scd;
    ZeroMemory(&scd, sizeof(DXGI_SWAP_CHAIN_DESC));

    scd.BufferCount = 1;                                // 1 back buffer
    scd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM; // Usar colores de 32 bit
    scd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;  // Modo de uso: salida de
la pipeline
    scd.OutputWindow = hWnd;                            // Ventana a usar
    scd.SampleDesc.Count = 1;                           // Multisamples
    scd.Windowed = TRUE;                                // Modo ventana/!pantalla
completa
```

```
    // Crear device, device context y swap chain
    HRESULT hr = D3D11CreateDeviceAndSwapChain(NULL,     // Adaptador gráfico a
usar (GPU). Se elige automáticamente
                               D3D_DRIVER_TYPE_HARDWARE,// Tipo de driver a usar
(software-hardware)
                               NULL,                    // Puntero al código del
driver si este es software
                               NULL,                    // Flags de
configuración
                               NULL,                    // Feature levels.
Distintas aracterísticas que pueden estar disponibles en el hardware
                               NULL,                    // Número de feature
levels
                               D3D11_SDK_VERSION,       // Versión de DirectX
con la que vamos a desarrollar la aplicación
                               &scd,                    // Info de la swap chain
                               &swapchain,              // Swap chain
                               &dev,                    // Device
                               NULL,                    // Máximo feature level
disponible en el equipo actual
                               &devcon);                // Device context
    if( FAILED(hr))
        return hr;

    // Crear render target view
    // Necesitamos la dirección del back buffer
    ID3D11Texture2D *pBackBuffer;
    swapchain->GetBuffer(0, __uuidof(ID3D11Texture2D), (LPVOID*)&pBackBuffer);

    // Usamos esta dirección para crear el render target
    dev->CreateRenderTargetView(pBackBuffer, NULL, &rtv);
    pBackBuffer->Release(); // Ya no necesitamos la textura

     // Configuramos la vista
    ZeroMemory(&viewport, sizeof(D3D11_VIEWPORT));
    viewport.TopLeftX = 0;
    viewport.TopLeftY = 0;
    viewport.Width = 800;
    viewport.Height = 600;

    return S_OK;
}


// Funcion para inicializar la pipeline
void InitPipeline()
{
    // Compilar y cargar los shaders
    ID3DBlob *VS, *PS, *CS;
    D3DCompileFromFile(L"Visualizacion.hlsl", NULL, NULL, "VSMAIN", "vs_5_0",
D3DCOMPILE_DEBUG, 0, &VS, NULL);
    D3DCompileFromFile(L"Visualizacion.hlsl", NULL, NULL, "PSMAIN", "ps_5_0",
D3DCOMPILE_DEBUG, 0, &PS, NULL);
    D3DCompileFromFile(L"AguaColumnas.hlsl", NULL, NULL, "CSMAIN", "cs_5_0",
D3DCOMPILE_DEBUG, 0, &CS, NULL);

    // Encapsular los objetos shaders
    dev->CreateVertexShader(VS->GetBufferPointer(), VS->GetBufferSize(), NULL,
&pVS);
    dev->CreatePixelShader(PS->GetBufferPointer(), PS->GetBufferSize(), NULL,
&pPS);
```

```
    dev->CreateComputeShader(CS->GetBufferPointer(), CS->GetBufferSize(), NULL,
&pCS);

    // Definir la estructura del input layout
    D3D11_INPUT_ELEMENT_DESC ied[] =
    {
        {"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
D3D11_INPUT_PER_VERTEX_DATA, 0},
    };

    // Crear el objeto input layout
    dev->CreateInputLayout(ied, ARRAYSIZE(ied), VS->GetBufferPointer(), VS-
>GetBufferSize(), &pLayout);

    // Configuramos el rasterizado
    D3D11_RASTERIZER_DESC rs;
    ZeroMemory(&rs, sizeof(rs));
    rs.FillMode =  D3D11_FILL_WIREFRAME; // Modo malla
    rs.CullMode = D3D11_CULL_BACK; // No mostrar las caras traseras
    dev->CreateRasterizerState( &rs, &pRState);
}


// Función para inicializar los objetos de la escena
void InitScene()
{
    numVertices=SizeX*SizeZ; // Número de vértices
    numIndices=6*(SizeX-1)*(SizeZ-1); // Número de índices

    // Creamos un plano de triángulos
    VERTEX *OurVertices;
    OurVertices= new VERTEX[numVertices];
    WORD *indices;
    indices = new WORD[numIndices];

    // Generar vértices
    for ( int y = 0; y < SizeZ; y++ )
    {
        for ( int x = 0; x < SizeZ; x++ )
        {
            OurVertices[y*SizeX+x].x = (float)x;
            OurVertices[y*SizeX+x].y = 0.0f;
            OurVertices[y*SizeX+x].z = (float)y;
        }
    }

    // Creamos el buffer de vértices
    D3D11_BUFFER_DESC bd;
    ZeroMemory(&bd, sizeof(bd));
    bd.Usage = D3D11_USAGE_DEFAULT; // Modo de uso. CPU -/- y GPU Read/Write
    bd.ByteWidth = sizeof(VERTEX) *numVertices;  // Tamaño
    bd.BindFlags = D3D11_BIND_VERTEX_BUFFER; // Donde puede asociarse en la
pipeline
    bd.CPUAccessFlags = 0;  // Permiso de escritura de la CPU

    D3D11_SUBRESOURCE_DATA InitData; // Datos iniciales
    ZeroMemory( &InitData, sizeof(InitData) );
    InitData.pSysMem = OurVertices;

    dev->CreateBuffer( &bd, &InitData, &pVBuffer); // Crear el buffer
```

```
    // Generar índices
    for ( int j = 0; j < SizeZ-1; j++ )
    {
        for ( int i = 0; i < SizeX-1; i++ )
        {
            indices[j*6*(SizeX-1)+i*6] = j*SizeX + i;
            indices[j*6*(SizeX-1)+i*6+1] = (j*SizeX + i) + SizeX;
            indices[j*6*(SizeX-1)+i*6+2] = (j*SizeX + i) + 1;

            indices[j*6*(SizeX-1)+i*6+3] = (j*SizeX + i) + 1;
            indices[j*6*(SizeX-1)+i*6+4] = (j*SizeX + i) + SizeX;
            indices[j*6*(SizeX-1)+i*6+5] = (j*SizeX + i) + SizeX + 1;
        }
    }

    // Crear el index buffer
    ZeroMemory(&bd, sizeof(bd));
    bd.Usage = D3D11_USAGE_DEFAULT;
    bd.ByteWidth = sizeof( WORD ) * numIndices; // Tamaño
    bd.BindFlags = D3D11_BIND_INDEX_BUFFER;
    bd.CPUAccessFlags = 0;

    InitData.pSysMem = indices; // Datos iniciales

    dev->CreateBuffer( &bd, &InitData, &pIBuffer ); // Crear el buffer
}


// Función para inicializar la simulación
void InitSim()
{
    // Creamos datos iniciales para la simulación
    GridPoint* pData;
    pData = new GridPoint[numVertices];

    for (int j = 0; j < SizeZ; j++)
    {
        for (int i = 0; i < SizeX; i++)
        {
            int x = i - 32;
            int y = j - 96;

            const float fFrequency = 0.1f;

            if (x*x + y*y != 0.0f)
            {
                pData[SizeX*j + i].height = 40.0f * sinf(sqrt((float)(x*x + y*y))
* fFrequency) / (sqrt((float)(x*x + y*y)) * fFrequency);
                pData[SizeX*j + i].flow[0] = 0.0f;
                pData[SizeX*j + i].flow[1] = 0.0f;
                pData[SizeX*j + i].flow[2] = 0.0f;
                pData[SizeX*j + i].flow[3] = 0.0f;
            }
            else
            {
                pData[SizeX*j + i].height = 40.0f;
                pData[SizeX*j + i].flow[0] = 0.0f;
                pData[SizeX*j + i].flow[1] = 0.0f;
                pData[SizeX*j + i].flow[2] = 0.0f;
                pData[SizeX*j + i].flow[3] = 0.0f;
            }
        }
```

```
    }

    // Datos iniciales
    D3D11_SUBRESOURCE_DATA InitialData;
    InitialData.pSysMem = pData;

    // Creamos los buffer que contendrán el estado de la simulación
    D3D11_BUFFER_DESC bd;
    ZeroMemory(&bd, sizeof(bd));
    bd.ByteWidth = numVertices*sizeof(GridPoint);
    bd.BindFlags = D3D11_BIND_SHADER_RESOURCE | D3D11_BIND_UNORDERED_ACCESS;
    bd.MiscFlags = D3D11_RESOURCE_MISC_BUFFER_STRUCTURED;
    bd.StructureByteStride = sizeof(GridPoint);
    bd.Usage = D3D11_USAGE_DEFAULT;
    bd.CPUAccessFlags = 0;

    dev->CreateBuffer(&bd, &InitialData, &pWaterSim1);
    dev->CreateBuffer(&bd, &InitialData, &pWaterSim2);

    // Ya no necesitamos la memoria de la CPU, pues la simulación es manejada por
la GPU
    delete[] pData;

    // Crear el constant buffer con la info de los threads usados
    ZeroMemory(&bd, sizeof(bd));
    bd.Usage = D3D11_USAGE_DEFAULT;
    bd.ByteWidth = sizeof(ConstantBufferDis);
    bd.BindFlags = D3D11_BIND_CONSTANT_BUFFER;
    bd.CPUAccessFlags = 0;
    dev->CreateBuffer(&bd, NULL, &pCBufferDis);

    // Actualizar los datos del buffer
    ConstantBufferDis cd;
    cd.DispatchSize[0] = float(DispatchSizeX);
    cd.DispatchSize[1] = float(DispatchSizeZ);
    cd.DispatchSize[2] = float(DispatchSizeX*ThreadGroupsX);
    cd.DispatchSize[3] = float(DispatchSizeZ*ThreadGroupsY);
    devcon->UpdateSubresource(pCBufferDis, 0, NULL, &cd, 0, 0);

    // Crear el constant buffer para el tiempo y la información de threads
    ZeroMemory(&bd, sizeof(bd));
    bd.Usage = D3D11_USAGE_DEFAULT;
    bd.ByteWidth = sizeof(ConstantBufferDisTime);
    bd.BindFlags = D3D11_BIND_CONSTANT_BUFFER;
    bd.CPUAccessFlags = 0;
    dev->CreateBuffer(&bd, NULL, &pCBufferDisTime);
}


// Función para iniciar la cámara
void InitCamera()
{
    // Inicializar la matriz del mundo
    myWorld = XMMatrixIdentity();

    // Inicialziar la vista
    XMVECTOR Eye = XMVectorSet(-10.0f, 30.0f, 3.0f, 0.0f);
    XMVECTOR At = XMVectorSet(30.0f, 2.0f, 30.0f, 0.0f);
    XMVECTOR Up = XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f);
    myView = XMMatrixLookAtLH(Eye, At, Up);

    // Inicializar la proyeccion
```

```
    myProjection = XMMatrixPerspectiveFovLH(XM_PIDIV4, 800.0f / 600.0f, 0.01f,
100.0f);

    // Crear el constant buffer para las matrices
    D3D11_BUFFER_DESC bd;
    ZeroMemory(&bd, sizeof(bd));
    bd.Usage = D3D11_USAGE_DEFAULT;
    bd.ByteWidth = sizeof(ConstantBufferMatrix);
    bd.BindFlags = D3D11_BIND_CONSTANT_BUFFER;
    bd.CPUAccessFlags = 0;
    dev->CreateBuffer(&bd, NULL, &pCBufferMat);

    // Actualizar buffer
    ConstantBufferMatrix cb;
    cb.World = XMMatrixTranspose(myWorld);
    cb.View = XMMatrixTranspose(myView);
    cb.Projection = XMMatrixTranspose(myProjection);
    devcon->UpdateSubresource(pCBufferMat, 0, NULL, &cb, 0, 0);
}


// Función para controlar eventos de windows
LRESULT CALLBACK WindowProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
    // Tratamos cada tipo de evento de una manera
    switch (message)
    {
    // Evento cerrar ventana
    case WM_DESTROY:
        // Cerrar la aplicación
        PostQuitMessage(0);
        return 0;
        break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
}


// Funcion para calcular un paso de la simulación
void Calculate()
{
    // Información del tiempo
    LARGE_INTEGER nuevoT;
    QueryPerformanceCounter(&nuevoT);
    float paso = float(nuevoT.QuadPart - tiempo.QuadPart);
    tiempo = nuevoT;

    // Limpiamos el estado de la pipeline
    devcon->ClearState();

    // Indicar los shaders a utilizar
    devcon->CSSetShader(pCS, 0, 0);

    // Buffers a usar
    devcon->CSSetConstantBuffers(0, 1, &pCBufferDisTime);

    // Creamos las vistas para acceder a la información. Como los buffers se
    // intercambian cada paso debemos volver a crear las vistas cada vez
    // Creamos la shader resource view para poder leer los gridpoints en la
pipeline
```

```
    D3D11_SHADER_RESOURCE_VIEW_DESC desc;
    desc.Format = DXGI_FORMAT_UNKNOWN;
    desc.ViewDimension = D3D11_SRV_DIMENSION_BUFFER;
    desc.Buffer.ElementOffset = 0;
    desc.Buffer.NumElements = numVertices;
    dev->CreateShaderResourceView(pWaterSim1, &desc, &pSRVWater);
    devcon->CSSetShaderResources(0, 1, &pSRVWater);

    // Creamos la unordered access view para poder escribir los gridpoints desde
la pipeline
    D3D11_UNORDERED_ACCESS_VIEW_DESC uesc;
    uesc.Format = DXGI_FORMAT_UNKNOWN;
    uesc.ViewDimension = D3D11_UAV_DIMENSION_BUFFER;
    uesc.Buffer.FirstElement = 0;
    uesc.Buffer.NumElements = numVertices;
    uesc.Buffer.Flags = D3D11_BUFFER_UAV_FLAG_COUNTER;
    dev->CreateUnorderedAccessView(pWaterSim2, &uesc, &pUAVWater);
    devcon->CSSetUnorderedAccessViews(0, 1, &pUAVWater, 0);

    // Actualizar variables
    ConstantBufferDisTime cb;
    cb.TimeFactor = paso / freq.QuadPart;
    cb.DispatchSize[0] = float(DispatchSizeX);
    cb.DispatchSize[1] = float(DispatchSizeZ);
    cb.DispatchSize[2] = float(SizeX);
    cb.DispatchSize[3] = float(SizeZ);

    devcon->UpdateSubresource(pCBufferDisTime, 0, NULL, &cb, 0, 0);

    // Simular en la GPU
    devcon->Dispatch(ThreadGroupsX, ThreadGroupsY, 1);

    // Intercambiar buffers
    ID3D11Buffer *Temp = pWaterSim1;
    pWaterSim1 = pWaterSim2;
    pWaterSim2 = Temp;

    Temp->Release();
}


// Función para renderizar un frame
void Render()
{
    // Limpiamos el estado de la pipeline
    devcon->ClearState();

    // Limpiar el backbuffer a un color gris
    float clearColor[4] = { 0.8f, 0.8f, 0.8f, 1.0f };
    devcon->ClearRenderTargetView(rtv, clearColor);

    // Elegir tipo de primitiva a usar
    devcon->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);

    // Indicamos el layout a utilizar
    devcon->IASetInputLayout(pLayout);

    // Indicar los shaders a utilizar
    devcon->VSSetShader(pVS, 0, 0);
    devcon->PSSetShader(pPS, 0, 0);

    // Indicar los buffers a usar
```

```
    UINT stride = sizeof(VERTEX);    // Tamaño del vértice
    UINT offset = 0;                // Offset inicial
    devcon->IASetVertexBuffers(0, 1, &pVBuffer, &stride, &offset);
    devcon->IASetIndexBuffer(pIBuffer, DXGI_FORMAT_R16_UINT, 0);
    devcon->VSSetConstantBuffers(0, 1, &pCBufferMat);
    devcon->VSSetConstantBuffers(1, 1, &pCBufferDis);

    // Indicar las vistas a usar
    devcon->VSSetShaderResources(0, 1, &pSRVWater);

    // Indicamos la configuración de rasterizado
    devcon->RSSetState(pRState);
    devcon->RSSetViewports(1, &viewport);

    // Indicamos cual es el render target a usar
    devcon->OMSetRenderTargets(1, &rtv, NULL);

    // Dibujar el vertex buffer al back buffer
    devcon->DrawIndexed(numIndices, 0, 0); // Nº vértices(indexados) empezando en
el 0

    // Intercambiar el backbuffer con el frontal
    swapchain->Present(0, 0);
}


// Funcion para cerrar y liberar los objetos usados
void CleanD3D()
{
    pVS->Release();
    pPS->Release();
    pCS->Release();
    swapchain->Release();
    rtv->Release();
    pVBuffer->Release();
    pLayout->Release();
    pCBufferMat->Release();
    pWaterSim1->Release();
    pWaterSim2->Release();
    pCBufferDis->Release();
    pCBufferDisTime->Release();
    pRState->Release();
    pSRVWater->Release();
    pUAVWater->Release();
    dev->Release();
    devcon->Release();
}
```

E. 4. Shader de cálculo

El cálculo de la simulación se hace en un compute shader utilizando las ideas introducidas en los apartados de contexto y diseño. A partir de la diferencia de alturas entre columnas, que tenemos disponibles, se calcularán las diferencias de presión, aceleración y nuevo flujo, que es lo que se necesita para modificar la altura de cada columna.

```
// Estructuras usadas
// Estado de una columna de la simulacion
struct GridPoint
{
    float  Height;
    float4 Flow;
};


// Buffers usados
// Buffer constante con la informacion de tiempo de la simulacion y threads
usados
cbuffer TimeParameters
{
    float TimeFactor;
    float4 DispatchSize;
};

// Estados de la simulacion, el actual y el nuevo a calcular
RWStructuredBuffer<GridPoint> NewWaterState        : register( u0 );
StructuredBuffer<GridPoint>   CurrentWaterState : register( t0 );


// Datos a usar
// Tamaño de la simulacion
#define size_x 16
#define size_y 16

// Tamaño de grupo con perimetro extra
#define padded_x (1 + size_x + 1)
#define padded_y (1 + size_y + 1)

// Memoria de grupo para compartir informacion
groupshared GridPoint loadedpoints[padded_x * padded_y];

// Numero de threads por grupo
[numthreads(padded_x, padded_y, 1)]


// Compute shader
void CSMAIN( uint3 GroupID : SV_GroupID, uint3 DispatchThreadID :
SV_DispatchThreadID, uint3 GroupThreadID : SV_GroupThreadID, uint GroupIndex :
SV_GroupIndex )
{
    // Tamaño de una rejilla (vista)
    int gridsize_x = DispatchSize.x;
    int gridsize_y = DispatchSize.y;

    // Tamaño total del mapa de alturas
    int totalsize_x = DispatchSize.z;
    int totalsize_y = DispatchSize.w;

    // Hacemos todos los accesos a la memoria con el estado de la simulacion
actual para guardarlos en la memoria compartida
    // Datos iniciales a 0
    loadedpoints[GroupIndex].Height = 0.0f;
    loadedpoints[GroupIndex].Flow = float4( 0.0f, 0.0f, 0.0f, 0.0f );

    // Calcular la posicion dentro del buffer usando los IDs del thread
    int3 location = int3( 0, 0, 0 );
    location.x = GroupID.x * size_x + ( GroupThreadID.x - 1 );
    location.y = GroupID.y * size_y + ( GroupThreadID.y - 1 );
```

```
    int textureindex = location.x + location.y * totalsize_x;

    // Cargar la informacion a memoria compartida
    loadedpoints[GroupIndex] = CurrentWaterState[textureindex];

    // Sincronizar los threads del grupo para asegurar que todos han cargado los
datos
    GroupMemoryBarrierWithGroupSync();

    //---------------------------------------
    // Calcular y actualizar los flujos
    //
    //                  o--o x
    //                 /|\
    //                / | \
    //               o  o  o
    //              w   z   y

    // Cargar los flujos del estado actual de la simulacion
    float4 NewFlow = float4( 0.0f, 0.0f, 0.0f, 0.0f );

    // Comprobar que no estamos en el borde derecho del grupo
    if ( ( GroupThreadID.x < padded_x - 1 ) && ( location.x < totalsize_x - 1 ) )
    {
        NewFlow.x = ( loadedpoints[GroupIndex+1].Height -
loadedpoints[GroupIndex].Height );

        // Comprobar que no estamos en el borde inferior del grupo
        if ( ( GroupThreadID.y < padded_y - 1 ) && ( location.y < totalsize_y - 1
) )
        {
            NewFlow.y = ( loadedpoints[(GroupIndex+1) + padded_x].Height -
loadedpoints[GroupIndex].Height );
        }
    }

    // Comprobar que no estamos en el borde inferior del grupo
    if ( ( GroupThreadID.y < padded_y - 1 ) && ( location.y < totalsize_y - 1 ) )
    {
        NewFlow.z = ( loadedpoints[GroupIndex+padded_x].Height -
loadedpoints[GroupIndex].Height );

        // Comprobar que no estamos en el borde izquierdo del grupo
        if ( ( GroupThreadID.x > 0 ) && ( location.x > 0 ) )
        {
            NewFlow.w = ( loadedpoints[GroupIndex + padded_x - 1].Height -
loadedpoints[GroupIndex].Height );
        }
    }

    // Datos usados para calcular los nuevos flujos
    const float TIME_STEP = 0.005f;
    const float PIPE_AREA = 0.0001f;
    const float GRAVITATION = 10.0f;
    const float PIPE_LENGTH = 0.2f;
    const float FLUID_DENSITY = 1.0f;
    const float COLUMN_AREA = 0.05f;
    const float DAMPING_FACTOR = 1.0f; // Para que la inercia de la simulacion
vaya disminuyendo

    // Aceleracion del flujo
```

```
    float fAccelFactor = ( min( TimeFactor, TIME_STEP ) * PIPE_AREA * GRAVITATION
) / ( PIPE_LENGTH * COLUMN_AREA );

    // Calculo del nuevo flujo, sumando el valor anterior.
    NewFlow = ( NewFlow * fAccelFactor + loadedpoints[GroupIndex].Flow ) *
DAMPING_FACTOR;

    // Actualizar la memoria compartida con el nuevo flujo
    loadedpoints[GroupIndex].Flow = NewFlow;

    // Sincronizar todos los threads antes de seguir
    GroupMemoryBarrierWithGroupSync();

    //---------------------------------------

    // Actualizar la altura de cada columna
    // Flujo saliente
    loadedpoints[GroupIndex].Height = loadedpoints[GroupIndex].Height + NewFlow.x
+ NewFlow.y + NewFlow.z + NewFlow.w;

    // Flujo entrante de la izquierda
    loadedpoints[GroupIndex].Height = loadedpoints[GroupIndex].Height -
loadedpoints[GroupIndex-1].Flow.x;

    // Flujo entrante de la esquina superior izquierda
    loadedpoints[GroupIndex].Height = loadedpoints[GroupIndex].Height -
loadedpoints[GroupIndex-padded_x-1].Flow.y;

    // Flujo entrante de arriba
    loadedpoints[GroupIndex].Height = loadedpoints[GroupIndex].Height -
loadedpoints[GroupIndex-padded_x].Flow.z;

    // Flujo entrante de la esquina superior derecha
    loadedpoints[GroupIndex].Height = loadedpoints[GroupIndex].Height -
loadedpoints[GroupIndex-padded_x+1].Flow.w;

    // Actualizar el nuevo estado de la simulacion (ignorando el perimetro)
    if ( ( GroupThreadID.x > 0 ) && ( GroupThreadID.x < padded_x - 1 ) &&
( GroupThreadID.y > 0 ) && ( GroupThreadID.y < padded_y - 1 ) )
    {
        NewWaterState[textureindex] = loadedpoints[GroupIndex];
    }
}
```

E. 5. Shaders de renderizado

El renderizado de la superficie obtenida es sencillo y solo utiliza un vertex y un pixel shader. El vertex shader transforma la posición del vértice, cuya altura obtendrá de los cálculos previamente realizados, y elige un color en función del grupo de threads donde se ha realizado la simulación. El pixel shader simplemente usa este color como color definitivo.

```
// Estructuras usadas
// Entrada al vertex shader
struct VS_INPUT
{
    float3 position : POSITION; // Posicon del vertice
};
```

```
// Salida del vertex shader y entrada al pixel shader
struct VS_OUTPUT
{
    float4 position : SV_POSITION; // Posicion del pixel
    float4 color : COLOR; // Color
};

// Estado de una columna de la simulacion
struct GridPoint
{
    float Height;
    float4 Flow;
};


// Buffers usados
// Buffer constante con las matrices de transformacion
cbuffer Transforms
{
    matrix World;
    matrix View;
    matrix Projection;
}

// Buffer constante con informacion del numero de threads
cbuffer DispatchParams
{
    float4 DispatchSize;
};

// Buffer estructurado con el estado de la simulacion
StructuredBuffer<GridPoint>   CurrentWaterState  : register(t0);


// Vertex shader
VS_OUTPUT VSMAIN( in VS_INPUT v )
{
    VS_OUTPUT o = (VS_OUTPUT)0;

    // Determinar en que posicion del buffer con el estado de la simulacion
buscar la altura
    // Coordenadas en 2 dimensiones
    int2 coords = int2( floor(v.position.x), floor(v.position.z));
    // Pasamos a una sola dimension
    int textureindex = coords.x + coords.y * DispatchSize.z;

    // Obtener la altura de esa posicion
    float height = CurrentWaterState[textureindex].Height;

    // Transformar la posicion del vertice a clipspace
    float4 SampledPosition = float4( v.position.x, height, v.position.z, 1.0f );
    o.position = mul( SampledPosition, World);
    o.position = mul( o.position, View);
    o.position = mul( o.position, Projection);

    // Elegir color del punto en funcion del grupo de threads al que pertenece
    int2 GridPosition = (coords / 16.0f );
    float ColorIndex = frac( ( GridPosition.x + GridPosition.y ) / 2.0f );

    if ( ColorIndex < 0.5f )
        o.color = float4( 0.0f, 1.0f, 0.0f, 1.0f );
    else
```

```
        o.color = float4( 0.0f, 0.0f, 1.0f, 1.0f );

    return o;
}

// Pixel shader
float4 PSMAIN( in VS_OUTPUT input ) : SV_Target
{
    return input.color;
}
```

E. 6. Resultados

El resultado final puede verse en la figura 50

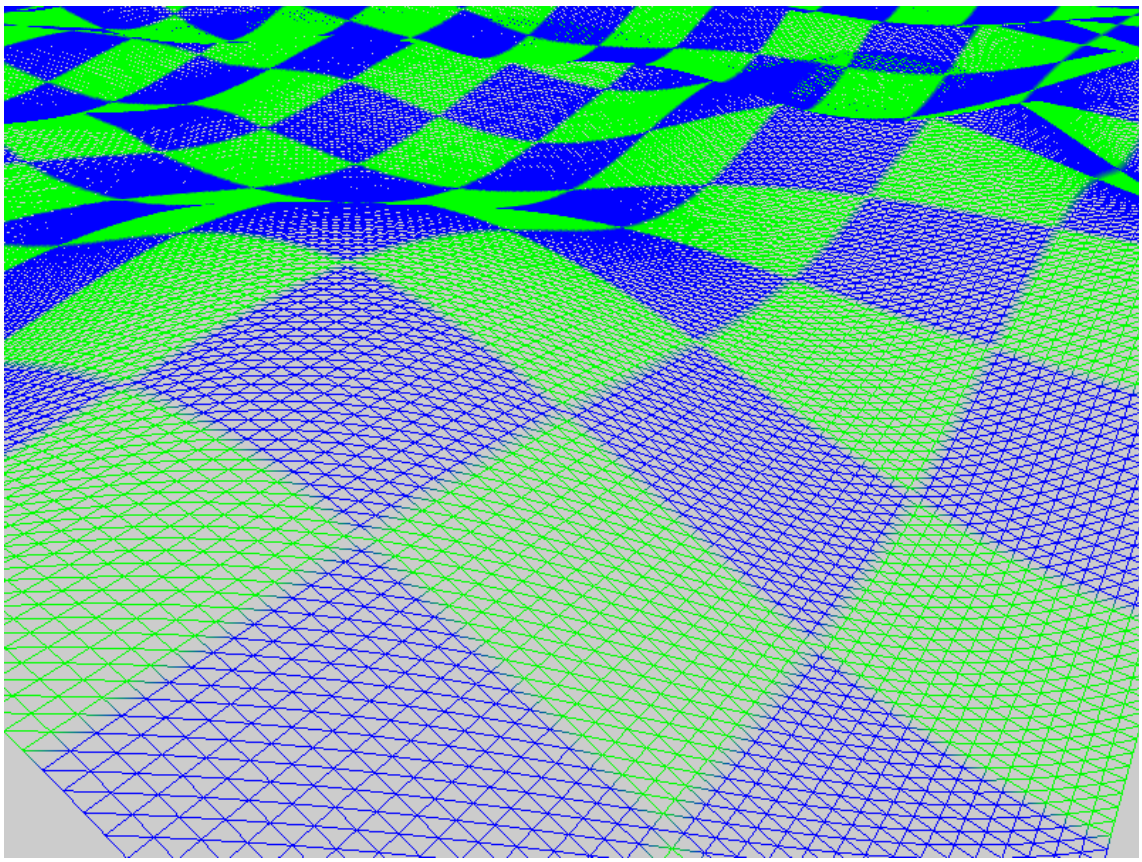


Figura 50: Simulación de agua con columnas

ANEXO F: RUIDO Y FRACTALES

En este anexo, titulado “Ruido y fractales”, se define qué es el ruido y se distingue entre varios tipos. Además se explica el concepto de fractal y como puede construirse uno usando alguna clase de ruido.

F. 1. Ruido

En nuestro ámbito el ruido es una función que varía de manera aparentemente aleatoria. Decimos que es aparente porque aunque no se observe ninguna pauta la función debe ser repetible. Esta aleatoriedad se usa para evitar la aparición de patrones que, aunque sí existen, no llegan a ser apreciables. Además el rango de salida de la función debe ser conocido y su ancho de banda limitado.

El ruido se usa habitualmente para generar objetos naturales como agua, montañas o nubes, ya sea para la construcción de su forma o para la de las texturas que se aplicarán a su superficie.

F. 2. Tipos de ruido

A continuación se hace una breve diferenciación de algunos de los tipos de ruido más usados.

F. 2. 1. Ruido de rejilla

Uno de los tipos de ruido más sencillo y eficiente. Se define una malla para cada punto de nuestro espacio cuyas coordenadas son enteras y se le asigna un número pseudoaleatorio, normalmente con una permutación aleatoria previamente calculada de números enteros en el rango deseado. Para obtener el valor en cualquier punto solo hay que interpolar entre los puntos de la rejilla que lo rodean. Dependiendo del método usado para interpolar y del número de vecinos utilizados se conseguirán unos resultados u otros.

F. 2. 2. Ruido de valores

Similar al anterior, en este caso el valor de cada punto de la malla es un número aleatorio entre -1 y 1.

F. 2. 3. Ruido de gradientes

En este caso tenemos también una permutación de números enteros, pero esta vez se usa para asignar a cada punto de la malla un gradiente pseudoaletorio. Estos gradientes son generados usando vectores unitarios distribuidos de manera aleatoria en la esfera unidad. Para calcular el ruido en un punto se usan los puntos de la celda en que se encuentra, obteniendo el producto escalar del gradiente por el vector que va desde cada punto de la celda de la rejilla al propio punto e interpolando.

F. 2. 4. Ruido de Perlin

Tipo particular del ruido anterior que usa gradientes fijos ya que la permutación de enteros ya otorga suficiente aleatoriedad. En cuanto a la función usada para interpolar, esta era originalmente $3t^2 – 2t^3$ pero la segunda derivada de ésta no es 0 ni en t=1 ni en t=0, por lo que el propio Perlin decidió más adelante cambiar y usar $6t^5 – 15t^4 + 10t^3$.

F. 2. 5. Ruido de valores-gradientes

El ruido de gradientes es siempre 0 para las coordenadas enteras, por lo que es posible que aparezca algún patrón. Para tratar de evitarlo el ruido de valores – gradientes hace una suma ponderada de un ruido de valores y un ruido de gradientes.

F. 2. 6. Ruido de rejilla compuesto

Como el primer tipo de ruido comentado pero usando algún kernel más complejo que simplemente utilizar los puntos de la celda donde nos encontremos para interpolar, como por ejemplo una esfera de cierto radio o un cono. De esta manera se evitan posibles errores alineados a los ejes.

F. 2. 7. Ruido de rejilla disperso

Este tipo de ruido utiliza puntos dispersos para realizar la interpolación: no se eligen los más cercanos, sino puntos aleatorios.

F. 2. 8. Ruidos explícitos

En los ruidos explícitos el valor del ruido se precalcula antes y se guarda, de manera que a la hora de utilizarlos solo hay que buscar el valor en una tabla, textura o cualquier lugar donde esté almacenado. Esto disminuye el tiempo necesario para obtener el ruido, pero limita la flexibilidad.

F. 2. 9. Síntesis espectral de Fourier

Se genera un espectro discreto de frecuencias de manera aleatoria y éste se filtra para que tenga el rango o forma deseados. Después se aplica una transformada inversa de Fourier para cambiar del domino de la frecuencia al del espacio, que será el ruido.

F. 3. Fractales

Un fractal es un patrón que se repite a diferentes escalas y que se utiliza para obtener la forma de un objeto irregular que no se podría conseguir con los métodos de la geometría clásica. Si a su construcción añadimos algún tipo de aleatoriedad tenemos un fractal aleatorio.

En nuestro caso el patrón viene marcado por una función de ruido sumada a distintas frecuencias y con diferentes escalas.

Una de las funciones fractales aleatorias más sencillas y utilizadas es la conocida como fBm, “fractal brownian motion”, definida por los siguientes parámetros:

Punto: lugar donde calculamos el valor del fractal.
H: incremento de la amplitud en cada frecuencia.
Lagunaridad: salto existente entre frecuencias sucesivas.
Octavas: número de frecuencias a sumar. Se usa el mismo nombre que en música porque normalmente el salto entre frecuencias es de 2.

A continuación podemos ver el código de dicha función:

```
float fbm(float2 punto, float H, float lagunaridad, float octavas)
{
	float value = 0.0f, sum = 0.0f, freq = 1.0f, amp = 1.0f;

	// Construcción del fractal
	for (float i = 0.0f; i < octavas; i++)
	{
		value = noise(punto * freq);
		sum += value * amp;
		freq *= lagunaridad;
		amp *= H;
	}
	return sum;
}
```

En este caso el patrón viene marcado directamente por el valor del ruido, pero podrían usarse funciones más complejas como senos o polinomios.

En las figuras 51, 52 y 53 se presentan imágenes de MojoWorlds, mundos enteramente construidos usando fractales y ruido, para comprobar la potencia de estas técnicas.

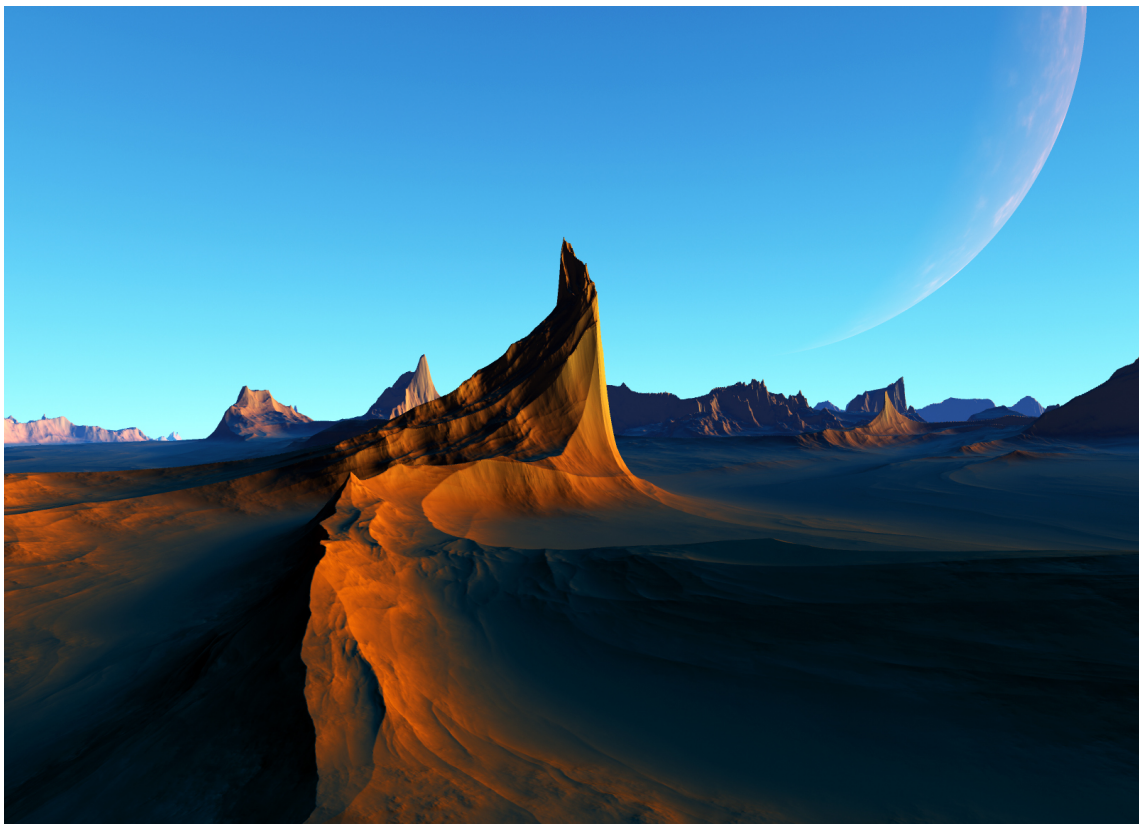


Figura 51: Montañas fractales

Figura 52: Lago fractal

Figura 53: Océano fractal

ANEXO G: DOCUMENTACIÓN DE LA APLICACIÓN

A continuación se presenta la documentación referente a la aplicación implementada para la generación del paisaje procedural. La documentación se ha generado de manera automática usando el programa DoxyGen.

G. 1. Lista de clases

AudioEngine (Motor de audio)
Camera (Una camara)
Camera::ConstantBufferCam (Buffer con la posicion de la camara)
Camera::ConstantBufferMatrix (Buffer con las matrices de la camara)
Cielo (Cielo de la escena)
D3D11Manager (Controlador de Direct3D 11)
Datos (Clase con varios datos necesarios)
Datos::ConstantBufferDatos (Buffer con datos para los shaders)
GPUProfiler (Profiler de la ejecucion de la GPU)
Light (Luz)
Light::ConstantBufferLight (Buffer con los datos de una luz)
ObjManager (Manejo de archivos .obj)
Paisaje (Paisaje de la escena)
Pajaros (Controlador de los pajaros)
Pajaros::InfoPajaro (Informacion de un pajaro)
Perlin (Datos del ruido de Perlin)
Planeta (Planeta fractal)
SamplerManager (Clase para el manejo de samplers de texturas
SceneManager (Controlador de la escena)
Shadow (Sombras)
Shadow::ConstantBufferMatrixShadow (Buffer con las matrices de la camara-luz)
TextureManager (Manejo de texturas externas)
VERTEX (Vertice)
WaveLoader (Cargador de archivos .wav)

G. 2. Documentación de clases

AudioEngine Class Reference

Motor de audio.

Public Member Functions

1. HRESULT **InitAudio** ()
   Iniciar el motor de audio.
2. void **LoadSounds** ()
   Cargar los sonidos.
3. void **Play** ()
   Iniciar reproduccion de sonidos.

4. void **Clean** ()
   Liberar recursos.

Public Attributes

1. IXAudio2 * **audioeng**
   Motor de audio.
2. IXAudio2MasteringVoice * **audiodev**
   Audio device.

Camera Class Reference

Una camara.

Classes

1. struct **ConstantBufferCam**
   Buffer con la posicion de la camara.

1. struct **ConstantBufferMatrix**
   Buffer con las matrices de la camara.

Public Member Functions

1. void **InitCamera** (**D3D11Manager** *D3D)
   Inicio de la camara.
2. void **IniciarPaseoAereo** ()
   Iniciar paseo Aereo.
3. void **IniciarPaseoPersona** ()
   Iniciar paseo Persona.
4. void **ZoomIn** ()
   Acercar la camara.
5. void **ZoomOut** ()
   Alejar la camara.
6. void **MoveUp** ()
   Mover la camara hacia arriba.
7. void **MoveDown** ()
   Mover la camara hacia abajo.
8. void **MoveRight** ()
   Mover la camara hacia la derecha.
9. void **MoveLeft** ()
   Mover la camara hacia la izquierda.
10. void **LookRight** ()
    Girar la camara hacia la derecha.
11. void **LookLeft** ()
    Girar la camara hacia la izquierda.
12. void **LookUp** ()
    Girar la camara hacia arriba.
13. void **LookDown** ()
    Girar la camara hacia abajo.
14. void **Home** ()
    Posicion inicial.
15. void **Aire** ()
    Posicion alejada.

16. void **Place** (XMVECTOR newPos, XMVECTOR newAt)
    Colocar la cámara.
17. void **UpdateMatrix** (XMMATRIX *newWorld)
    Actualizar matriz del mundo.
18. void **UpdateCamera** ()
    Actualizar posicion y datos de la camara durante los recorridos.
19. void **Clean** ()
    Liberar recursos.

Public Attributes

1. XMVECTOR **camPos**
   Posicion de la camara.
2. bool **camPaseoAereo**
   Variable que nos indica si estamos en medio del recorrido aereo.
3. bool **camPaseoPersona**
   Variable que nos indica si estamos en medio del recorrido de la persona.
4. ID3D11Buffer * **pCBufferMat**
   Constant buffer de las matrices de mundo, vista y proyeccion.
5. ID3D11Buffer * **pCBufferCam**
   Constant buffer con la posicion de la camara.

Member Function Documentation

void Camera::InitCamera (D3D11Manager * *D3D*)

Parameters:

D3D	IN: Controlador de Direct3D

void Camera::Place (XMVECTOR *newPos*, XMVECTOR *newAt*)

Parameters:

newPos	IN: Nueva posicion
newAt	IN: Nuevo punto de mira

void Camera::UpdateMatrix (XMMATRIX * *newWorld*)

Parameters:

newWorld	IN: Nueva matriz

Camera::ConstantBufferCam Struct Reference

Buffer con la posicion de la camara.

Public Attributes

1. XMVECTOR **CameraPos**
   Posicion de la camara.

Camera::ConstantBufferMatrix Struct Reference

Buffer con las matrices de la camara.

Public Attributes

1. XMMATRIX **World**
   Mundo.

2. XMMATRIX **View**
   Vista.
3. XMMATRIX **Projection**
   Proyeccion.

Cielo Class Reference

Cielo de la escena.

Public Member Functions

1. void **InitCielo** (**D3D11Manager** *D3D)
   Inicializar el cielo.
2. void **Render** (**Camera** *Cam, **TextureManager** *TextureMan, **SamplerManager** *SamplerMan)
   Renderizar el cielo.
3. void **UpdateCielo** (**Camera** *Cam)
   Actualizar posicion del cielo.
4. void **Clean** ()
   Liberar recursos.

Member Function Documentation

void Cielo::InitCielo (D3D11Manager * *D3D*)

Parameters:

D3D	IN: Controlador de Direct3D

void Cielo::Render (Camera * *Cam*, TextureManager * *TextureMan*, SamplerManager * *SamplerMan*)

Parameters:

Cam	IN: Camara de la escena
TextureMan	IN: Texturas
SamplerMan	IN: Samplers

void Cielo::UpdateCielo (Camera * *Cam*)

Parameters:

Cam	IN: Camara en torno a la que centramos el cielo

D3D11Manager Class Reference

Controlador de Direct3D 11.

Public Member Functions

1. void **InitD3D** (HWND hWnd, int windowWidth, int windowHeight)
   Inicio de Direct3D.
2. void **Clean** ()
   Liberar recursos.

Public Attributes

1. D3D11_VIEWPORT **viewport**
   Viewport.
2. IDXGISwapChain * **swapchain**
   Swap chain.

3. ID3D11Device * **dev** = NUL
   Direct3D device interface.
4. ID3D11DeviceContext * **devcon**
   Direct3D device context.
5. IDXGIFactory * **factory**
   Interfaz para crear objetos DXGI.
6. IDXGIAdapter * **adapter**
   Tarjeta grafica.
7. IDXGIOutput * **adapterOutput**
   Monitor.
8. ID3D11RenderTargetView * **rtv**
   Render Target View.
9. ID3D11Texture2D * **pDSB**
   Depth stencil buffer.
10. ID3D11DepthStencilView * **pDSV**
    Vista del depth buffer.
11. ID3D11InputLayout * **pLayout**
    Formato de entrada a la pipeline (vertices)
12. ID3D11RasterizerState * **pRStateCull**
    Configuracion de la etapa de rasterizado con culling.
13. ID3D11RasterizerState * **pRStateNoCull**
    Configuracion de la etapa de rasterizado sin culling.
14. ID3D11BlendState * **pBSTransparent**
    Blender state para usar transparencias.

Member Function Documentation

void D3D11Manager::InitD3D (HWND *hWnd*, int *windowWidth*, int *windowHeight*)

Parameters:

hWnd	IN: Ventana de la aplicación
windowWidth	IN: Anchura de la ventana
windowHeight	IN: Altura de la ventana

Datos Class Reference

Clase con varios datos necesarios.

Classes

1. struct **ConstantBufferDatos**
   Buffer con datos para los shaders.

Public Member Functions

1. void **InitData** (**D3D11Manager** *D3D)
   Inicio de los datos.
2. void **UpdateData** (float d0, float d1, float d2, float d3)
   Actualizar buffer de datos.
3. void **Clean** ()
   Liberar recursos.

Public Attributes

1. int **PaisajeX**
   Dimension X del paisaje.

2. int **PaisajeZ**
   Dimension Z del paisaje.
3. int **SizeX**
   Numero de puntos en X.
4. int **SizeZ**
   Numero de puntos en Z.
5. ID3D11Buffer * **pCBufferDat**
   Constant buffer con datos para los shaders.

Member Function Documentation

void Datos::InitData (D3D11Manager * *D3D*)

Parameters:

D3D	IN: Controlador de Direct3D

void Datos::UpdateData (float *d0*, float *d1*, float *d2*, float *d3*)

Parameters:

d0	IN: Primer dato del buffer
d1	IN: Segundo dato del buffer
d2	IN: Tercer dato del buffer
d3	IN: Cuarto dato del buffer

Datos::ConstantBufferDatos Struct Reference

Buffer con datos para los shaders.

Public Attributes

1. float **data** [4]
   Tiempo, Tamaño de la permutacion, SizeX/Latitud , SizeZ/Longitud.

GPUProfiler Class Reference

Profiler de la ejecucion de la GPU.

Public Member Functions

1. void **InitProfiler** (**D3D11Manager** *D3D)
   Inicializar el profiler.
2. void **SaveFrame1** ()
   Guardar info de la imagen 1.
3. void **SaveFrame2** ()
   Guardar info de la imagen 2.
4. void **Profile** ()
   Recoger y presentar los datos de un frame.
5. void **InitEncloser** ()
   Inicio del frame encloser.
6. void **EndEncloser** ()
   Fin del frame encloser.
7. void **InitFrame** ()
   Inicio del frame.
8. void **EndFrame** ()
   Final del frame.

9. void **InitCalc** ()
   Inicio del calculo del agua.
10. void **EndCalc** ()
    Fin del calculo del agua.
11. void **InitRender** ()
    Inicio del renderizado.
12. void **EndRender** ()
    Fin del renderizado.
13. void **InitPipeline** ()
    Inicio de la ejecucion de la pipeline grafica.
14. void **EndPipeline** ()
    Final de la ejecucion de la la pipeline grafica.
15. void **Clean** ()
    Liberar recursos.

Member Function Documentation

void GPUProfiler::InitProfiler (D3D11Manager * *D3D*)

Parameters:

D3D	IN: Controlador de Direct3D

Light Class Reference

Luz.

Classes

1. struct **ConstantBufferLight**
   Buffer con los datos de una luz.

Public Member Functions

1. void **InitLight** (**D3D11Manager** *D3D, **Datos** *Data)
   Inicializacion de la luz.
2. void **RotateRight** ()
   Girar luz hacia la derecha.
3. void **RotateLeft** ()
   Girar luz hacia la izquierda.
4. void **Home** ()
   Posicion inicial.
5. void **UpdateLight** ()
   Actualizar la luz.
6. void **Clean** ()
   Liberar recursos.

Public Attributes

1. XMFLOAT3 **lightDir**
   Direccion de la luz.
2. XMVECTOR **lightPos**
   Posicion de la luz.
3. XMVECTOR **lightAt**
   Lugar al que mira.
4. XMMATRIX **lightView**
   Matriz de vista de la luz.

5. XMMATRIX **lightProjection**
   Matriz de proyeccion de la luz.
6. ID3D11Buffer * **pCBufferLight**
   Buffer con los datos de la luz.
7. bool **luzActualizada**
   Acabamos de actualizar la luz.

Member Function Documentation

void Light::InitLight (D3D11Manager * *D3D*, Datos * *Data*)

Parameters:

D3D	IN: Controlador de Direct3D
Data	IN: Datos del paisaje

Light::ConstantBufferLight Struct Reference

Buffer con los datos de una luz.

Public Attributes

1. XMFLOAT3 **dirLight**
   Direccion de la luz.
2. float **pad**
   Alineacion del buffer.
3. XMFLOAT4 **colorLight**
   Color de la luz direccional.
4. XMFLOAT4 **ambientLight**
   Componente ambiental.

ObjManager Class Reference

Manejo de archivos .obj.

Public Member Functions

1. void **LoadObj** (std::vector< **VERTEX** > *vertices, std::vector< DWORD > *indices, std::wstring name, XMMATRIX *meshWorld, int *verticesMesh, int *indicesMesh, int tipo)
   Carga un archivo .obj.
2. void **SaveObj** (**VERTEX** *vertices, UINT SizeX, UINT SizeZ, const char *name)
   Crea un archivo .obj.

Member Function Documentation

void ObjManager::LoadObj (std::vector< VERTEX > * *vertices*, std::vector< DWORD > * *indices*, std::wstring *name*, XMMATRIX * *meshWorld*, int * *verticesMesh*, int * *indicesMesh*, int *tipo*)

Parameters:

vertices	OUT: Vector con los vertices del objeto
indices	OUT: Vector con los índices del objeto
name	IN: Fichero .obj
meshWorld	IN: Transformacion del objeto
verticesMesh	OUT: Numero de vertices del objeto
indicesMesh	OUT: Numero de indices
tipo	IN: Tipo de objeto

void ObjManager::SaveObj (VERTEX * *vertices*, UINT *SizeX*, UINT *SizeZ*, const char * *name*)

Parameters:

vertices	IN: Array con los vertices del objeto
SizeX	IN: Tamaño en X del array
SizeZ	IN: Tamaño en Z del array
name	IN: Nombre del fichero a crear

Paisaje Class Reference

Paisaje de la escena.

Public Member Functions

1. void **InitPaisaje** (**D3D11Manager** *D3D, **Datos** *Data)
   Inicializacion del paisaje.
2. void **CalcularPaisaje** (**Datos** *Data, **Perlin** *PerlinData, **SamplerManager** *SamplerMan)
   Calcular el paisaje inicial.
3. void **CalcularViento** (**Datos** *Data, **Perlin** *PerlinData, **SamplerManager** *SamplerMan)
   Calculo del viento.
4. void **UpdateAgua** (**Datos** *Data, **Perlin** *PerlinData, **SamplerManager** *SamplerMan, **Camera** *Cam, **GPUProfiler** *GPUProf)
   Actualizar agua.
5. void **Render** (**Datos** *Data, **SamplerManager** *SamplerMan, **TextureManager** *TextureMan, **Light** *Luz, **Shadow** *Shade, **Camera** *Cam)
   Renderizado del paisaje.
6. void **EscribirPaisaje** ()
   Guardar .obj del paisaje.
7. void **Clean** ()
   Liberar recursos.

Public Attributes

1. ID3D11Buffer * **pVBufferPaisaje**
   Buffer de vertices del paisaje.
2. ID3D11Buffer * **pIBufferPaisaje**
   Buffer de indices del paisaje.
3. XMMATRIX **paisajeWorld**
   Mundo del paisaje.
4. int **numIndices**
   Numero total de indices.

Member Function Documentation

void Paisaje::InitPaisaje (D3D11Manager * *D3D*, Datos * *Data*)

Parameters:

D3D	IN: Controlador de Direct3D
Data	IN: Datos del paisaje

void Paisaje::CalcularPaisaje (Datos * *Data*, Perlin * *PerlinData*, SamplerManager * *SamplerMan*)

Parameters:

Data	IN: Datos del paisaje
PerlinData	IN: Datos del ruido de Perlin
SamplerMan	IN: Samplers de texturas

void Paisaje::CalcularViento (Datos * *Data*, Perlin * *PerlinData*, SamplerManager * *SamplerMan*)

Parameters:

Data	IN: Datos del paisaje
PerlinData	IN: Datos del ruido de Perlin
SamplerMan	IN: Samplers de texturas

void Paisaje::UpdateAgua (Datos * *Data*, Perlin * *PerlinData*, SamplerManager * *SamplerMan*, Camera * *Cam*, GPUProfiler * *GPUProf*)

Parameters:

Data	IN: Datos del paisaje
PerlinData	IN: Datos del ruido de Perlin
SamplerMan	IN: Samplers de texturas
Cam	IN: Camara de la escena
GPUProf	IN: Profiler de la GPU

void Paisaje::Render (Datos * *Data*, SamplerManager * *SamplerMan*, TextureManager * *TextureMan*, Light * *Luz*, Shadow * *Shade*, Camera * *Cam*)

Parameters:

Data	IN: Datos del paisaje
SamplerMan	IN: Samplers de texturas
TextureMan	IN: Texturas externas
Luz	IN: Luces
Shade	IN: Sombras
Cam	IN: Camara

Pajaros Class Reference

Controlador de los pajaros

Classes

1. struct **InfoPajaro**
   Informacion de un pajaro.

Public Member Functions

1. void **InitPajaros** (**D3D11Manager** *D3D)
   Inicializacion de los pajaros.
2. void **UpdatePajaros** ()
   Actualizar posicion de los pajaros.
3. void **Render** (**Camera** *Cam)
   Renderizar los pajaros.
4. void **Clean** ()
   Liberar recursos.

Member Function Documentation

void Pajaros::InitPajaros (D3D11Manager * *D3D*)

Parameters:

D3D	IN: Controlador de Direct3D

void Pajaros::Render (Camera * *Cam*)

Parameters:

Cam	IN: Camara de la escena

Pajaros::InfoPajaro Struct Reference
Informacion de un pajaro.

Public Attributes
1. XMMATRIX **pajaroWorld**
   Mundo del pajaro.
2. XMMATRIX **pajaroRotation**
   Rotacion del pajaro.
3. XMMATRIX **pajaroTranslation**
   Translacion de los pajaros.
4. XMVECTOR **pajaroPos**
   Posicion del pajaro.
5. int **pajaroEstadoActua**l
   Estado actual de las alas del pajaro.
6. LARGE_INTEGER **pajaroTEstado**
   Tiempo para controlar el estado del pajaro.
7. LARGE_INTEGER **pajaroTPosicion**
   Tiempo para controlar la posicion de los pajaros.
8. XMVECTOR **pajaroGoals** [3]
   Puntos del camino que sigue el pajaro.
9. int **pajaroActualGoal**
   Actual objetivo del movimiento del pajaro.
10. XMVECTOR **pajaroDir**
    Direccion en la que se mueve el pajaro.

Perlin Class Reference
Datos del ruido de Perlin.

Public Member Functions
1. void **InitPerlin** (**D3D11Manager** *D3D)
   Crear datos relativos al ruido de Perlin.
2. void **Clean** ()
   Liberar recursos.

Public Attributes
1. ID3D11ShaderResourceView * **pSRVTexPerm**
   Vista con la textura de la permutacion de numeros enteros.
2. ID3D11ShaderResourceView * **pSRVTexGrad3**
   Vista con la textura de los gradientes 3D.
3. ID3D11ShaderResourceView * **pSRVTexGrad2**
   Vista con la textura de los gradientes 2D.
4. float **sizePerm** = 1024.0f
   Tamaño de la permutacion.

Member Function Documentation
void Perlin::InitPerlin (D3D11Manager * ***D3D*****)**

Parameters:

D3D	IN: Controlador de Direct3D

Planeta Class Reference
Planeta fractal.

Public Member Functions
1. void **InitPlaneta** (**D3D11Manager** *D3D)
   Inicio del planeta.
2. void **CalcularPlaneta** (**Perlin** *PerlinData, **SamplerManager** *SamplerMan, **Datos** *Data)
   Calcular fractal del planeta.

3. void **Render** (**Camera** *Cam, **Perlin** *PerlinData, **SamplerManager** *SamplerMan, **Datos** *Data)
   Renderizar el planeta.
4. void **UpdatePlaneta** ()
   Actualizar planeta.
5. void **Clean** ()
   Liberar recursos.

Member Function Documentation
void Planeta::InitPlaneta (D3D11Manager * *D3D*)

Parameters:

D3D	IN: Controlador de Direct3D

void Planeta::CalcularPlaneta (Perlin * *PerlinData*, SamplerManager * *SamplerMan*, Datos * *Data*)

Parameters:

PerlinData	IN: Datos del ruido de Perlin
SamplerMan	IN: Samplers de texturas
Data	IN: Datos del paisaje

void Planeta::Render (Camera * *Cam*, Perlin * *PerlinData*, SamplerManager * *SamplerMan*, Datos * *Data*)

Parameters:

Cam	IN: Camara
PerlinData	IN: Datos del ruido de Perlin
SamplerMan	IN: Samplers de las texturas
Data	IN: Datos del planeta

SamplerManager Class Reference
Clase para el manejo de samplers de texturas.

Public Member Functions
1. void **InitSamplers** (**D3D11Manager** *D3D)
   Inciar samplers de texturas.
2. void **Clean** ()
   Liberar recursos.

Public Attributes
1. ID3D11SamplerState * **pPointMirrorSampler**
   Sampler por puntos, mirror.
2. ID3D11SamplerState * **pLineMirrorSamplerComp**
   Sampler linear con comparacion, mirror.
3. ID3D11SamplerState * **pLineWrapSampler**
   Sampler lineal, wrap.

Member Function Documentation

void SamplerManager::InitSamplers (D3D11Manager * *D3D*)

Parameters:

D3D	IN: Controlador de Direct3D

SceneManager Class Reference

Controlador de la escena.

Public Member Functions

1. void **InitScene** (**D3D11Manager** *D3D)
   Inicializar la escena.
2. void **InitFractals** (**D3D11Manager** *D3D)
   Iniciar los fractales.
3. void **InitFrame** ()
   Marcar inicio de frame.
4. void **UpdateScene** (**D3D11Manager** *D3D)
   Actualizar escena.
5. void **RenderScene** (**D3D11Manager** *D3D)
   Renderizar escena.
6. void **EndFrame** ()
   Marcar final de frame.
7. void **ShowFrameInfo** ()
   Mostrar informacion del frame.
8. void **Modify** (WPARAM wParam)
   Modificar escena.
9. void **Clean** ()
   Liberar recursos.

Member Function Documentation

void SceneManager::InitScene (D3D11Manager * *D3D*)

Parameters:

D3D	IN: Controlador de Direct3D

void SceneManager::InitFractals (D3D11Manager * *D3D*)

Parameters:

D3D	IN: Controlador de Direct3D

void SceneManager::UpdateScene (D3D11Manager * *D3D*)

Parameters:

D3D	IN: Controlador de Direct3D

void SceneManager::RenderScene (D3D11Manager * *D3D*)

Parameters:

D3D	IN: Controlador de Direct3D

void SceneManager::Modify (WPARAM *wParam*)

Parameters:

wParam	IN: Tecla pulsada

Shadow Class Reference
Sombras.

Classes
1. struct **ConstantBufferMatrixShadow**
   Buffer con las matrices de la camara-luz.

Public Member Functions
1. void **InitShadow** (**D3D11Manager** *D3D, **Light** *Luz, XMMATRIX *world)
   Inicializacion de las sombras.
2. void **CalcularMapa** (ID3D11Buffer *pVBuffer, ID3D11Buffer *pIBuffer, int numIndices)
   Calcular mapa de sombras.
3. void **UpdateShadow** (**Light** *Luz, XMMATRIX *world)
   Actualizar sombras.
4. void **Clean** ()
   Librerar recursos.

Public Attributes
1. ID3D11Buffer * **pCBufferShadow**
   Constant buffer de las matrices de mundo, vista y proyeccion para las sombras.
2. ID3D11ShaderResourceView * **pSRVShadow**
   Vista para leer el mapa de sombras.

Member Function Documentation
void Shadow::InitShadow (D3D11Manager * *D3D*, Light * *Luz*, XMMATRIX * *world*)
Parameters:

D3D	IN: Controlador de Direct3D
Luz	IN: Luz que genera las sombras
world	IN: Mundo de la escena

void Shadow::CalcularMapa (ID3D11Buffer * *pVBuffer*, ID3D11Buffer * *pIBuffer*, int *numIndices*)
Parameters:

pVBuffer	IN: Vertices de la geometria que genera sombra
pIBuffer	IN: Indices de la geometria que genera sombra
numIndices	IN: Numero de indices de la geometria

void Shadow::UpdateShadow (Light * *Luz*, XMMATRIX * *world*)
Parameters:

Luz	IN: Luz que genera las sombras
world	IN: Mundo de la escena

Shadow::ConstantBufferMatrixShadow Struct Reference
Buffer con las matrices de la camara-luz.

Public Attributes
1. XMMATRIX **World**
   Mundo.
2. XMMATRIX **View**
   Vista.
3. XMMATRIX **Projection**
   Proyeccion.

TextureManager Class Reference
Manejo de texturas externas.

Public Member Functions
1. void **InitTextures** (**D3D11Manager** *D3D)
   Cargar texturas.
2. void **Clean** ()
   Liberar recursos.

Public Attributes
1. ID3D11ShaderResourceView * **pSRVTexMon**
   Vista de la textura de la montaña
2. ID3D11ShaderResourceView * **pSRVTexNormalMon**
   Vista de la textura con las normales de la montaña
3. ID3D11ShaderResourceView * **pSRVTexSand**
   Vista de la textura de la playa.
4. ID3D11ShaderResourceView * **pSRVTexNormalSand**
   Vista de la textura con las normales de la playa.
5. ID3D11ShaderResourceView * **pSRVTexWetSand**
   Vista de la textura de la playa, zona mojada.
6. ID3D11ShaderResourceView * **pSRVTexNormalWetSand**
   Vista de la textura con las normales de la playa, zona mojada.
7. ID3D11ShaderResourceView * **pSRVTexPalmera1**
   Vista de la textura de la palmera 1.
8. ID3D11ShaderResourceView * **pSRVTexPalmera2**
   Vista de la textura de la palmera 2.
9. ID3D11ShaderResourceView * **pSRVTexPalmera3**
   Vista de la textura de la palmera 3 y 4.
10. ID3D11ShaderResourceView * **pSRVTexPalmera5**
    Vista de la textura de la palmera 5.
11. ID3D11ShaderResourceView * **pSRVTexSky**
    Vista de la textura del cielo.

Member Function Documentation
void TextureManager::InitTextures (D3D11Manager * *D3D*)

Parameters:

D3D	IN: Controlador de Direct3D

VERTEX Struct Reference
Vertice.

Public Attributes
1. float **x**
   Posicion.x.
2. float **y**
   Posicion.y.
3. float **z**
   Posicion.z.
4. float **nx**
   Normal.x.

5. float **ny**
   Normal.y.
6. float **nz**
   Normal.z.
7. float **u**
   Coordenada u de la textura.
8. float **v**
   Coordenada v de la textura.
9. float **w**
   Coordenada w de la textura.
10. int **tipo**
    Tipo de objeto al que pertenece, agua, playas, montañas o palmeras.
11. int **numTriangulos**
    Numero de triangulos a los que pertenece.
12. int **triangulos** [6]
    Triangulos a los que pertenece.

WaveLoader Class Reference

Cargador de archivos .wav.

Public Member Functions

1. HRESULT **CargarSonido** (WAVEFORMATEXTENSIBLE *wfx, XAUDIO2_BUFFER *buffer, TCHAR *strFileName)
   Cargar fichero .wav.

Member Function Documentation

HRESULT WaveLoader::CargarSonido (WAVEFORMATEXTENSIBLE * *wfx*, XAUDIO2_BUFFER * *buffer*, TCHAR * *strFileName*)

Parameters:

wfx	Out: Formato de audio
buffer	Out: Buffer de audio
strFileName	In: Nombre del fichero