GENERACIÓN DE TERRENOS FRACTALES PARA ESCENAS 3D, EN OPENGL Y VULKAN

FRACTAL TERRAINS GENERATION FOR 3D SCENES, WITH OPENGL AND VULKAN

TRABAJO FIN DE GRADO
CURSO 2019-2020

AUTORES
DIEGO BARATTO VALDIVIA
JORGE RODRÍGUEZ GARCÍA
GONZALO SANZ LASTRA

DIRECTORA
ANA GIL LUEZAS

GRADO EN DESARROLLO DE VIDEOJUEGOS
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

GENERACIÓN DE TERRENOS FRACTALES PARA ESCENAS 3D, EN OPENGL Y VULKAN

FRACTAL TERRAINS GENERATION FOR 3D SCENES, WITH OPENGL AND VULKAN

TRABAJO DE FIN DE GRADO DE DESARROLLO DE VIDEOJUEGOS
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

AUTORES
JORGE RODRÍGUEZ GARCÍA
GONZALO SANZ LASTRA
DIEGO BARATTO VALDIVIA

DIRECTORA
ANA GIL LUEZAS

CONVOCATORIA: JUNIO 2020

GRADO EN DESARROLLO DE VIDEOJUEGOS
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

26 DE JUNIO DE 2020

RESUMEN

Generación de terrenos fractales para escenas 3D, en OpenGL y Vulkan

Se presenta el desarrollo e implementación de una aplicación interactiva para renderizar terrenos fractales, utilizando el algoritmo *Ray Marching*, y modelos de mallas de triángulos, basado en la tubería clásica de renderizado. Se incorpora también un movimiento físico del modelo sobre las superficies generadas, mediante el cálculo de puntos de colisión entre ellos. Además, la aplicación permite utilizar como API de programación gráfica OpenGL y Vulkan, con GLSL como *shading language*.

Palabras clave

OpenGL, Vulkan, Fractal, Terrenos, *Shader*, *Ray Marching*.

ABSTRACT

Fractal terrains generation for 3D scenes, with OpenGL and Vulkan

It has been developed an interactive application in order to render fractal terrains through the Ray Marching algorithm and vertex mesh models using the classic rendering pipeline. Collision points between them can be calculated, allowing a simple, physical movement of the model upon the created surfaces. Furthermore, the application lets OpenGL *and* Vulkan *to be used, with* GLSL *as shading language.*

Keywords

OpenGL, Vulkan, *Fractal*, *Terrain*, *Shader*, *Ray Marching*.

Índice de contenidos

ÍNDICE DE FIGURAS

1. Introducción

La generación de contenido gráfico 3D interactivo, como, por ejemplo, en los videojuegos, se realiza habitualmente utilizando métodos basados en la tubería de renderizado, que actúa sobre modelos de mallas de vértices, utilizando algunas de la *APIs* para la programación gráfica en GPU, como OpenGL, Direct3D o Vulkan.

En este proyecto, sin embargo, para el renderizado de fractales se utilizan algoritmos como *Ray Marching*, que actúa sobre modelos basados en *Signed Distance Functions* (SDFs) o fórmulas de distancia obtenidas a partir de descripciones matemáticas de sus geometrías. De esta forma se pueden producir diferentes entornos solamente atendiendo a dichas fórmulas matemáticas.

Además, la aplicación permite utilizar como *API* de programación gráfica OpenGL y Vulkan, ambas multiplataforma, con GLSL (y SPIR-V como lenguaje compilado en Vulkan) como lenguaje de *shaders*.

1.1. Motivación

Se ha tenido como principal inspiración el videojuego *Marble Marcher - A game of fractals based on physics* [6], basado en terrenos dinámicos generados mediante fractales en tres dimensiones . En esta aplicación, se utilizan físicas sobre este tipo de terrenos y el juego consiste en recorrerlos en el menor tiempo posible.

Ésto generó el atractivo principal del proyecto, teniendo la potencialidad o posibilidad de, mediante técnicas y fórmulas matemáticas sencillas, generar terrenos y formas muy complejas y dinámicas aplicables a videojuegos.

1.2. Objetivos

El principal objetivo es desarrollar una aplicación interactiva que integre la generación de terrenos mediante fractales y objetos basados en mallas de triángulos, de forma que puedan interactuar entre ellos mediante un movimiento físico simple de dichos objetos sobre las

superficies generadas mediante fractales. Utilizando para eso las últimas técnicas de las *APIs* de programación gráficas OpenGL y Vulkan.

Para lograrlo, se plantean como objetivos básicos iniciales la creación de terrenos mediante fractales tridimensionales (*Ray Marching*) y su implementación mediante dos *APIs* diferentes, OpenGL y Vulkan, profundizando en todas sus posibilidades y últimas técnicas. Además, se han extendido estos objetivos básicos añadiendo comunicación entre la tarjeta gráfica y la CPU con el objeto de añadir físicas a los terrenos, así como incorporar el renderizado de un modelo de malla de triángulos, mezclando ambas técnicas de renderizado diferentes mediante el *buffer* de profundidad.

1.3. Plan de trabajo

Una vez planteados los objetivos, se establecen la siguiente secuencia de etapas, que aunque no se realizaron en estricto orden, sí que son todas las tareas realizadas.

1.3.1 *Ray Marching* y OpenGL

El primer objetivo es el estudio del algoritmo *Ray Marching* y el desarrollo de una aplicación, utilizando OpenGL, para dibujar cualquier forma geométrica dada por la fórmula de distancia (SDF) que requiere *Ray Marching*.

En un primer lugar, se desarrolla una aplicación con la posibilidad de incluir y utilizar *shaders* para que los resultados de su ejecución sean visibles en una ventana. Estos *shaders* implementan el algoritmo de *Ray Marching* y constan de geometrías básicas como primeros ejemplos de prueba del correcto funcionamiento del mismo.

A continuación, se introducen en *shaders* las fórmulas de distancia de distintos fractales tridimensionales, sobre los que se aplicará el algoritmo *Ray Marching*, generando así las formas complejas dinámicas que tiene como finalidad el proyecto. Con esto, se puede pintar cualquier forma geométrica dada su fórmula de distancia en dicha *API*, consiguiendo a su vez el renderizado de fractales tridimensionales.

1.3.2 Estudiar la *API* gráfica de Vulkan

Teniendo el renderizado de fractales en OpenGL, se añade otra implementación de la aplicación utilizando la *API* de Vulkan, permitiendo ejecutar la misma funcionalidad en cualquiera de las dos *APIs*, agregando la implementación específica de las interfaces de cada una de las *APIs*.

Tras el estudio de Vulkan, se decide utilizar GLSL como lenguaje de programación de los *shaders* para su reutilización.

1.3.3 Física de colisiones en ambas *APIs*

Una vez conseguido el funcionamiento correcto en ambas *APIs*, el siguiente paso es conseguir una comunicación precisa y adecuada entre la tarjeta gráfica y la CPU, y con ello incorporar a la aplicación las físicas de colisión con los fractales. Para ello, se ha hecho uso de los *shader* de cómputo, los cuales permiten a la GPU realizar los cálculos especificados, liberando a la CPU de una gran carga de trabajo.

Conseguida la comunicación entre GPU y CPU, se implementa una simulación física newtoniana utilizando el algoritmo de *Ray Marching* de nuevo para calcular las distancias de cualquier objeto al fractal en cuestión y así poder colisionar con él.

1.3.4 Modelos de mallas y depth testing

Por último, se posibilita la carga de modelos de mallas en ambas *APIs*, con el objetivo de conseguir integrar el renderizado de estos modelos con los fractales, utilizando el test de profundidad o *depth test*, y delegar la tarea de decidir qué píxel se visualiza a la *API* gráfica utilizada. De esta forma, el fractal se renderiza con los *shaders* que implementan *Ray Marching*, y al jugador con los *shaders* que actúan sobre la tubería de renderizado clásico, permitiendo así interactuar ambas técnicas.

1.3.5 Interfaz de usuario

Como añadido, se ofrece un pequeño menú en el que se despliegan todas las opciones disponibles, para poder observar y explorar todos los terrenos y fractales en los que se ha trabajado.

Se ha utilizado GitHub como repositorio de trabajo para el proyecto, el código de la aplicación puede encontrarse en el siguiente enlace: https://github.com/DiegoBV/TFG-Repo. Para compilar su contenido es necesario descargar sus dependencias, siguiendo las instrucciones del archivo README.md, situado en el directorio raíz del repositorio.

2. Ray Marching

La parte inicial del proyecto se centra en los *shaders,* o conjunto de instrucciones *software* usados para calcular efectos de renderizado en *hardware* gráfico, y el algoritmo de *Ray Marching*. Dicho algoritmo se caracteriza por ser una implementación de *Ray Tracing*. Se sitúa una cámara en una posición determinada y se establece una matriz o cuadrícula de puntos en frente de ella, correspondiendo cada uno de ellos con un píxel de la imagen final. Posteriormente, se lanza un rayo desde la cámara hasta cada uno de los puntos.

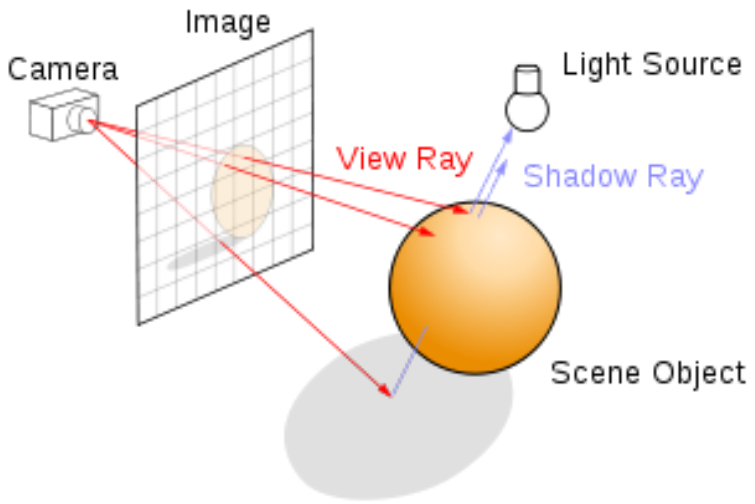

Figura 2-1: Descripción gráfica del algoritmo *Ray Tracing*.

Fuente: Jamie Wong, *Ray Marching and Signed Distance Functions* [54]

La diferencia radica en la forma en la que se define la escena, determinada por funciones de distancia (*Signed Distance Function*, SDF) y la forma de detectar la colisión entre el rayo y el objeto. Esto se realiza avanzando una cantidad determinada a lo largo del rayo hasta que, según la fórmula de distancia, ocurra la colisión. Mientras que en Ray Tracing (o Ray Casting) se calcula la colisión exacta entre el rayo y los objetos, que pueden ser mallas de triángulos componiendo formas irregulares, en *Ray Marching* se calcula una colisión aproximada, dada por una fórmula de distancia, por lo que los objetos deben poder ser descritos mediante dichas fórmulas matemáticas.

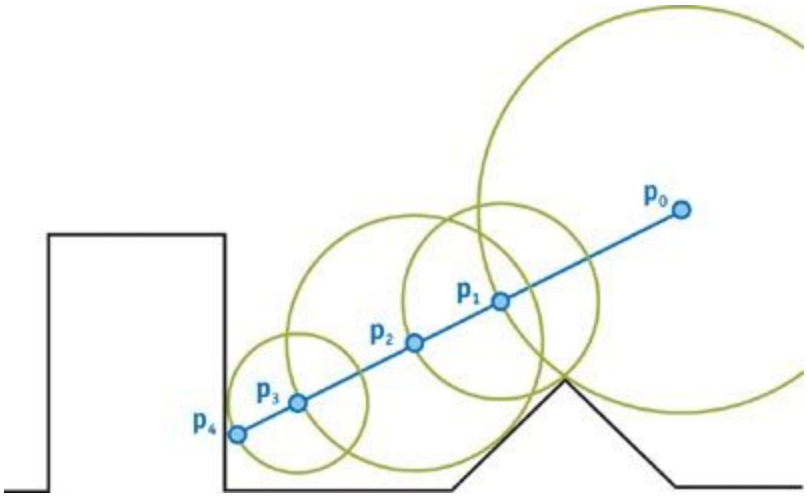

Figura 2-2: Descripción gráfica del algoritmo *Ray Marching*.

Fuente: Jamie Wong, Ray Marching and Signed Distance Functions [54]

Para implementarlo, se utilizan dos tipos de shader: *vertex* y *fragment*. El *vertex shader* se encarga de transformar la posición tridimensional de cada vértice en las coordenadas bidimensionales del píxel. Por otro lado, el *fragment shader* se encarga de dar color a cada uno de los píxeles correspondientes al interior de las primitivas generadas a partir del output del *vertex shader*. Dadas las necesidades de este proyecto, todas las funcionalidades de Ray Marching están implementadas en los *fragment shaders* (en estos casos también denominados *pixel shaders*).

2.1. Shaders

Los objetivos inicialmente planteados se basaron en la página web *Ray Marching and Signed Distance Functions* [54], complementados con el vídeo del canal *Ray Marching for Dummies!* [50], para aprender a utilizar *shaders* en OpenGL y el algoritmo de *Ray Marching*.

El primer objetivo fue, por tanto, conseguir dibujar una esfera utilizando fórmulas de distancia y *Ray Marching*. La implementación del algoritmo se realiza en el *fragment shader*, configurando el *vertex shader* para que, al renderizar un rectángulo, desencadene la ejecución del

fragment shader para cada uno de los píxeles de la ventana (como para un postprocesado). En este caso, se utiliza un rectángulo que cubre todo el puerto de vista.

Durante este proceso ocurrieron una serie de problemas, donde la esfera sólo se dibujaba cuando la distancia a la cámara y su radio eran muy parecidos. Además, ésta quedaba distorsionada con un efecto "ojo de pez". En un principio se intentó utilizar el depurador que ofrece Visual Studio para depurar el código de los *shaders*, pero debido a la dificultad de enlazar el *shader* con el depurador se abandonó esta idea.

Al final se consiguió solucionar cerrando el FOV o *field of view* de la cámara, como se detalla posteriormente en el apartado referente a la cámara, y se logró pintar una primera esfera utilizando *Ray Marching*.

Pseudocódigo del algoritmo *Ray Marching*:

```
float rayMarch(posCamera, direccionRayos, puntoInicio, puntoFinal){
      distanciaRecorrida = puntoInicio; // distancia recorrida por el rayo
      for (i = 0, i < pasosRaymarching, i++){
        distancia = distanciaEsfera(posCamera +
        distanciaRecorrida * direccionRayos);
        if (distancia < distanciaMinima) // si choca
            return distanciaRecorrida; // devuelve el punto de colisión
        distanciaRecorrida += distancia ; // actualiza distancia recorrida
        if (distanciaRecorrida >= puntoFinal)// si se pasa del punto final
            return puntoFinal; // devuelve el punto final
      }
      return puntoFinal; // si no ha chocado, devuelve el punto final
}
```

Pseudocódigo de la fórmula de distancia a la esfera:

```
float funcionDistanciaEsfera(píxel) {
    distanciaEsfera = longitud(píxel - centroEsfera) - radioEsfera;
    return distanciaEsfera;
}
```

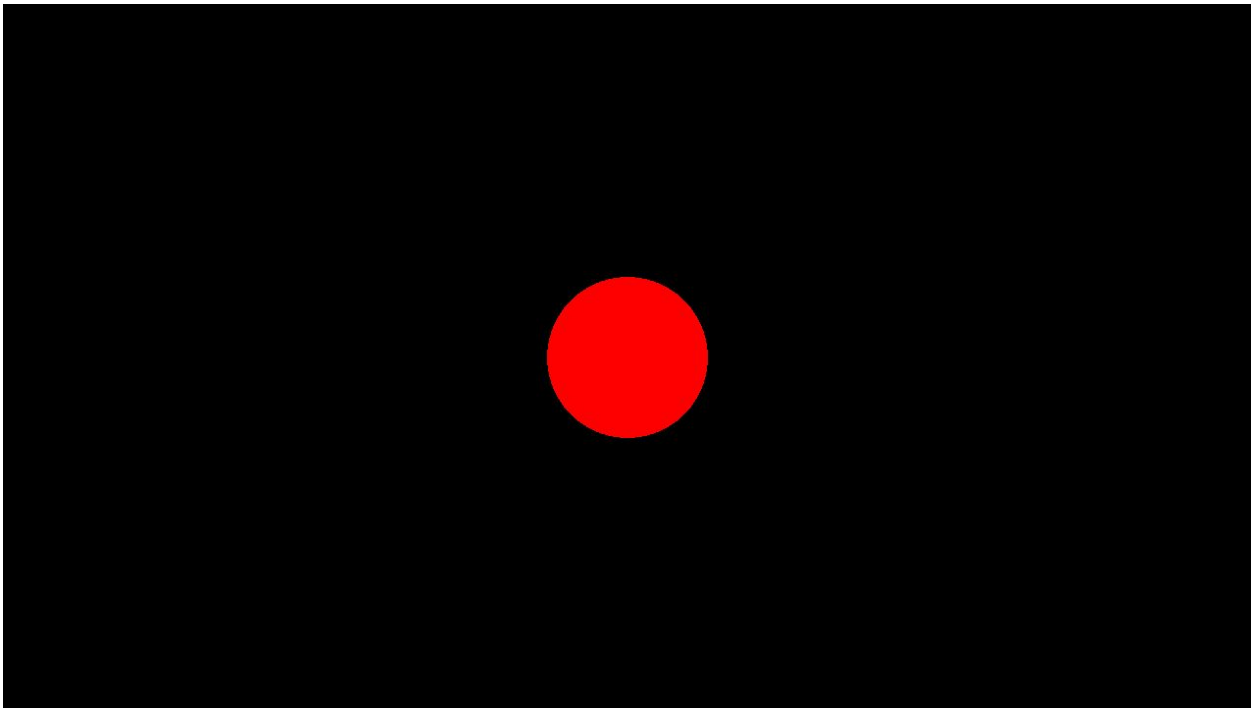

Figura 2.1-1: Esfera dibujada utilizando fórmulas de distancia y *Ray Marching*.

2.2. Iluminación

Una vez conseguido pintar la esfera sin problemas, se consiguió iluminar la misma siguiendo las fuentes de información ya mencionadas. Para ello también se utiliza *Ray Marching*, lanzando los rayos esta vez desde la fuente de luz y en la dirección en la que ésta esté supuestamente orientada. Los rayos que colisionen con la esfera le aplicarán un cambio de color, a la vez que dejarán el suelo sin colorear (negro), simulando así la sombra de ésta.

Pseudocódigo iluminación:

```
   float calcularLuz(píxel){
    coeficienteDifuso = multiplicar(normal, luz);
    distancia = rayMarching(píxel + (normal * distanciaMinima* 2.0),
    luz);
    if(distancia < longitud(posLuz - píxel)) coeficienteDifuso *= 0.1;
    return coeficienteDifuso;
}
```

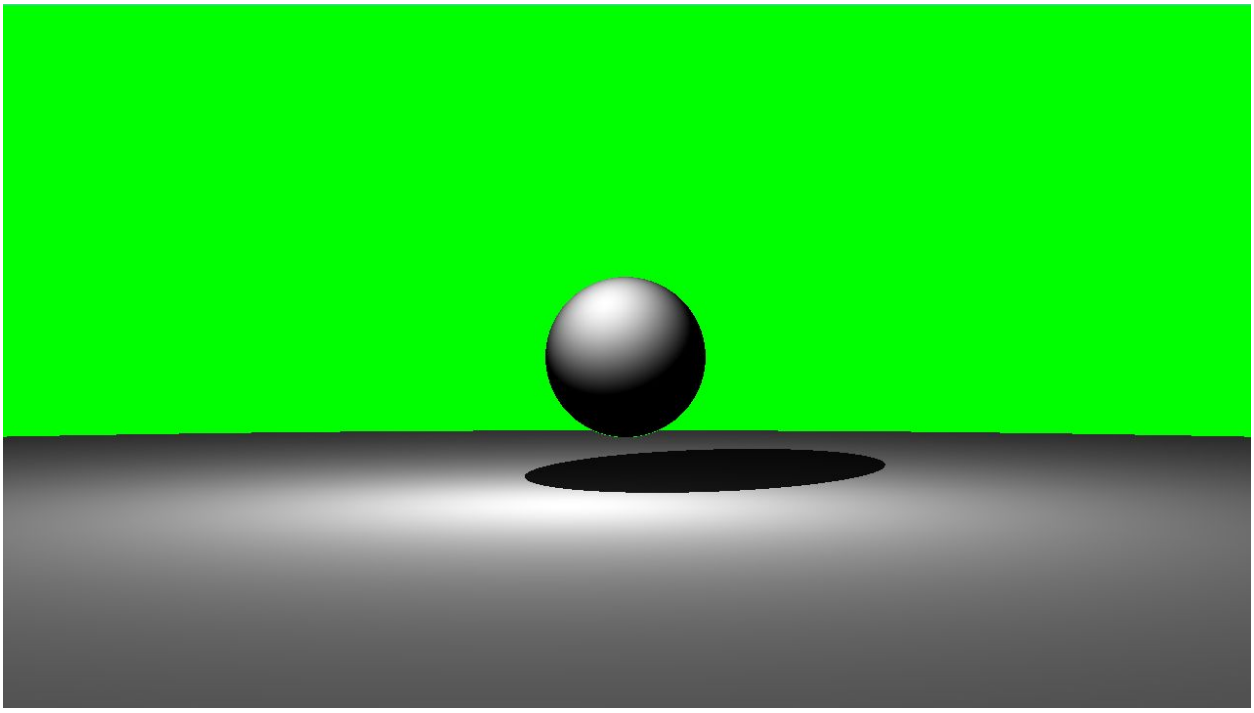

Figura 2.2-1: Esfera iluminada.

2.3. Primitivas básicas, rotaciones y objetos compuestos

Se alcanzó el objetivo de renderizar primitivas básicas, tales como cilindros, planos y cubos, aparte de la esfera. Intentando hacer el código algo más limpio y genérico, las funciones de estas primitivas se implementaron en un archivo diferente. En una primera instancia, se pretendía hacer clases para las geometrías básicas que heredasen de una clase genérica para ahorrar repetición de código, pero debido a que *glsl* está basado en C, el cual carece de estructura de clases y herencia, se acabaron haciendo *structs* con parámetros básicos y una función de distancia (SDF: *Signed Distance Function*) para cada uno de estos *structs*.

Pseudocódigo cubo:

```
struct Cubo{
    dimensiones;
    centro;
    matrizRotacion;
};
```

```
SDF(cubo, puntoOrigen) {
    puntoOrigen *= cubo.matrizRotacion;
    dimensiones = (abs(puntoOrigen - cubo.centro)) - cubo.dimensiones;
    return longitud(max(q,0.0)) + min(max(q.x,max(q.y,q.z)),0.0);
}
```

También se añadieron funciones para poder transformar las primitivas: trasladar, escalar y rotar, en cualquiera de los tres ejes (x, y, z).

Pseudocódigo rotaciones:

```
mat3 rotateX(angle){
    matrizRotacion = (
        1, 0, 0,
        0, coseno(angle), -seno(angle),
        0, seno(angle), coseno(angle)
    );
    return matrizRotacion;
}
```

Y a su vez, para construir objetos combinando las ya mencionadas primitivas mediante operaciones booleanas: intersección, unión y diferencia.

Pseudocódigo intersección:

```
    maximo(distanciaA, distanciaB);
```

Pseudocódigo unión:

```
    minimo(distanciaA, distanciaB);
```

Pseudocódigo diferencia:

```
    maximo(distanciaA, -distanciaB);
```

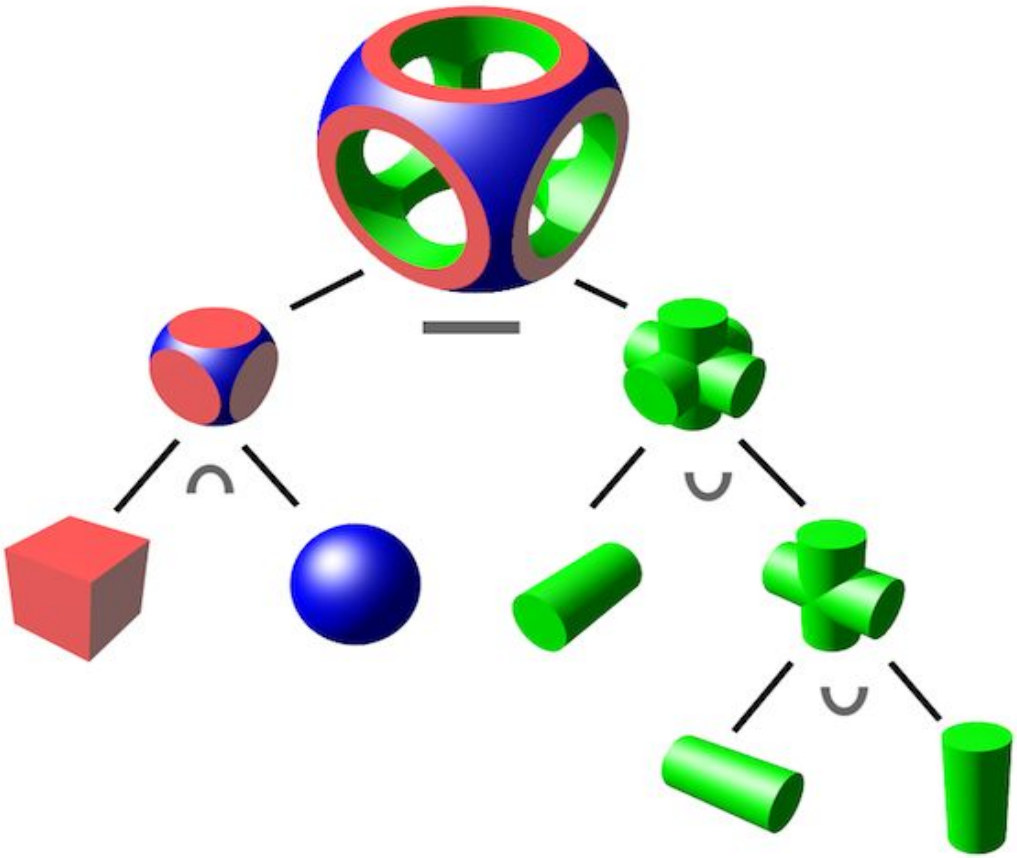

Figura 2.3-1: Ejemplos de intersección, unión y diferencia.

Fuente: Jamie Wong, Ray Marching and Signed Distance Functions (2016)

Para terminar, se dividió el *fragment shader* en diferentes archivos con el propósito de limpiar y organizar el código de los *shaders*. Cumplidos todos estos objetivos, se decidió aplicarlos en una misma escena dando lugar al siguiente resultado:

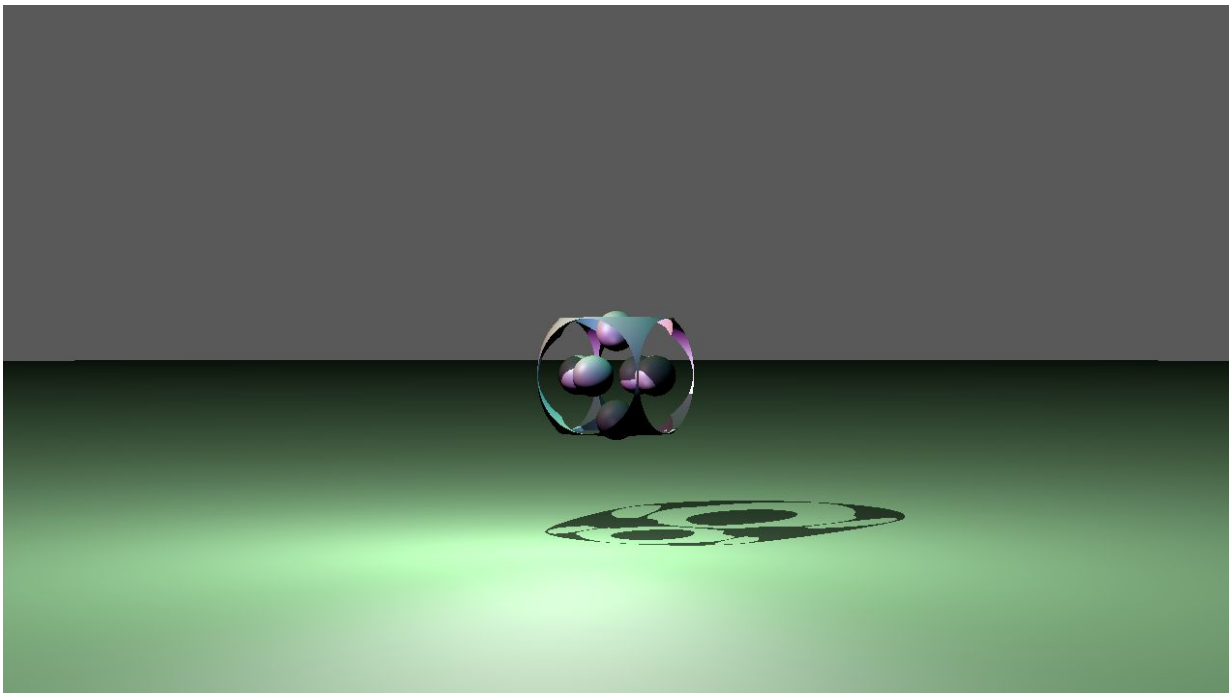

Figura 2.3-2: Escena final.

3. Aplicación

El desarrollo de la aplicación se ha realizado con el objetivo de que Vulkan y OpenGL coexistan en el mismo proyecto. Por ello, se pueden distinguir dos partes: código común a ambas *APIs*, como puede ser la cámara y la ventana; código específico respectivo a cada interfaz de programación, entre lo que se encuentra las llamadas a los *shaders* y la implementación de una comunicación entre GPU y CPU.

3.1. Ventana

La creación de la ventana se realiza a través de la librería GLFW. Su creación es dependiente a las dos configuraciones disponibles (Vulkan y OpenGL) debido a que dichas plataformas requieren una serie de parámetros iniciales distintos entre sí, como se puede observar a continuación:

```
#if defined (GL_DEBUG) || defined (GL_RELEASE)          // OpenGL
     glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
     glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 5);
     glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
#elif defined(VULKAN_DEBUG) || defined(VULKAN_RELEASE) // Vulkan
     glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
     glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);
#endif
```

Sin embargo, su uso es independiente de estas dos configuraciones, ya que éste resulta igual en ambas. Todo lo relacionado con la entrada de usuario o *user input* se maneja haciendo uso de las retrollamadas o *callbacks* ofrecidos por la librería, entre ellos el movimiento a través de pulsaciones de teclado, el giro del ratón o salir de la aplicación con la tecla *escape*.

En relación a la parte gráfica, la ventana tiene coordenadas normalizadas, y para pintar en ella, los shaders deberán hacer uso de vértices en los extremos de la misma, formando así dos triángulos haciendo las veces de lienzo de fondo, como se muestra en la siguiente imagen.

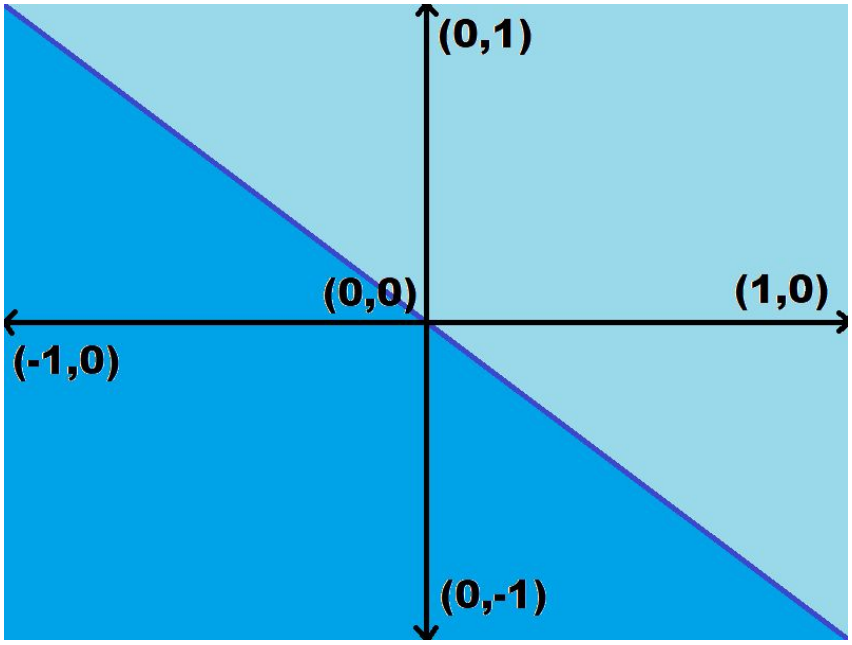

Figura 3.1-1: Disposición de la ventana de la aplicación

3.2. Cámara

Inicialmente se implementó una cámara con funcionalidad básica, basándose en el diseño propuesto en *LearnOpenGL - Camera* [17], ofreciendo libertad para observar el entorno creado en la aplicación. Se puede mover dicha cámara con las teclas W, A, S, y D para desplazarla hacia adelante, izquierda, atrás y derecha, respectivamente. Asimismo, el ratón es usado para controlar la rotación, imitando el movimiento del *mouse* en la aplicación.

Para respetar los cálculos realizados por los *shaders* y el entorno se visualice de la manera esperada, la matriz de vista calculada por la función gl::lookAt en cada *frame* es transmitida a los shaders como variable *uniform* para calcular la dirección correspondiente de los rayos del *Ray Marching*.

Con el fin de evitar el fenómeno “ojo de pez” o visión distorsionada debido a la dispersión de los rayos expuesto en un capítulo anterior, se ha implementado el cálculo de las direcciones del rayo atendiendo al campo de visión o *field of view* (FOV), lanzando dichos rayos en un cono con origen en la cámara y con una apertura máxima del campo de visión especificado. Utilizando la siguiente fórmula, se halla la posición del *near plane* y, por tanto, la inclinación o apertura que deben tener los rayos:

```
vec3 rayDirection(campoDeVision, tamañoVentana, coordenadaFragmento) {
        uv = (coordenadaFragmento.x - (tamañoVentana.x/2.0),
        coordenadaFragmento.y - (tamañoVentana.y/2.0));
        planoCercano= tamañoVentana.y / tan(radians(campoDeVision) / 2.0);
        return normalizar(vec3(uv, -planoCercano)); // dirección del rayo
}
```

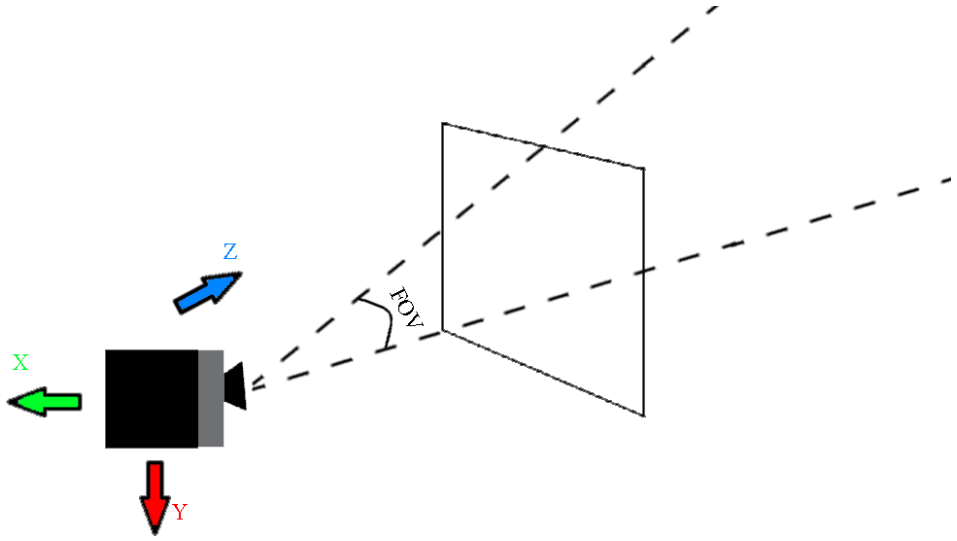

Figura 3.2-1: Visualización del campo de visión de una cámara.

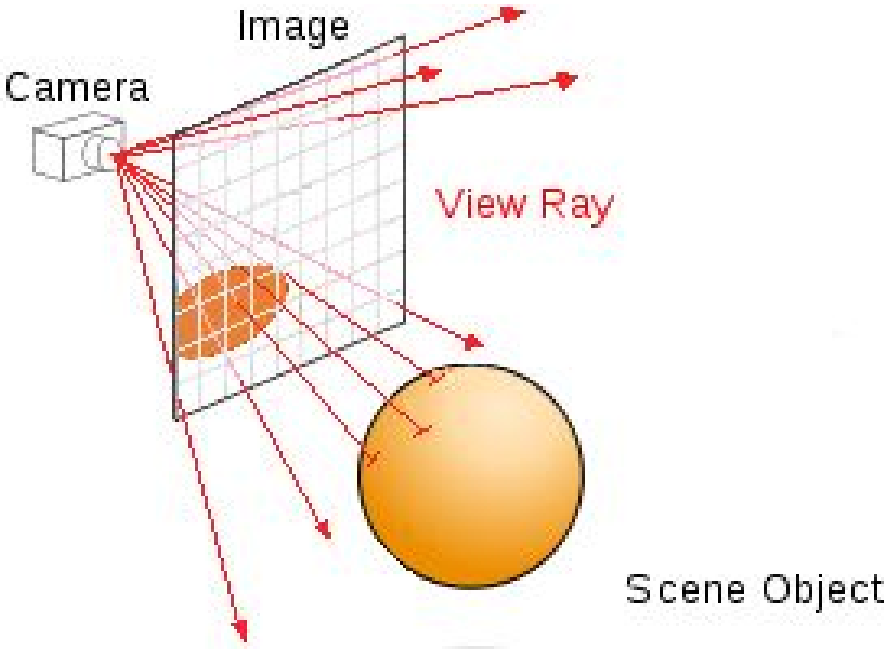

Figura 3.2-2: Distorsión de la imagen por excesiva dispersión de los rayos.

Modificación personal de una imagen de Jamie Wong, Ray Marching and Signed Distance Functions (2016)

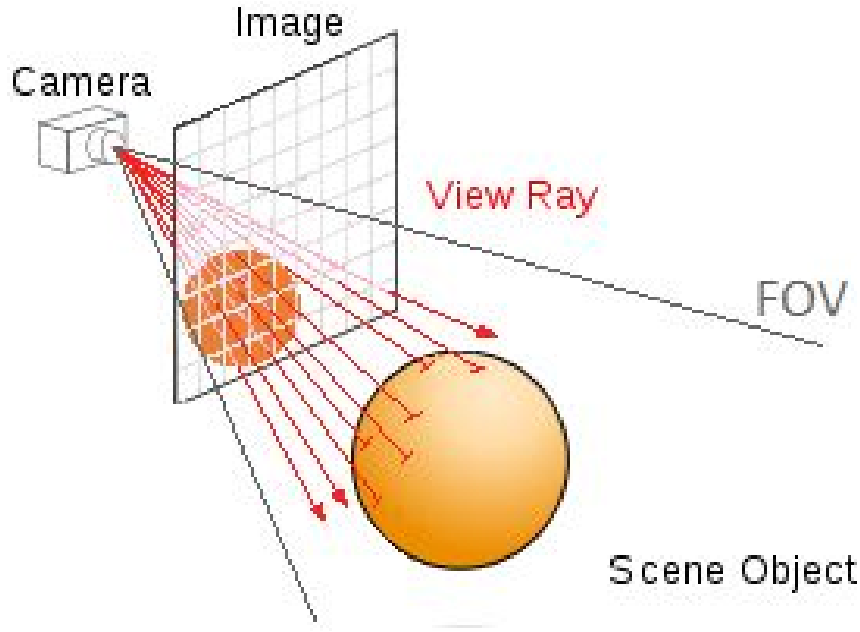

Figura 3.2-3: Corrección de distorsión reduciendo la dispersión mediante el FOV.

Modificación personal de una imagen de Jamie Wong, Ray Marching and Signed Distance Functions (2016)

Una vez obtenido este tipo de cámara "libre", se le añadió la funcionalidad de seguir un punto objetivo y pivotar en torno a él, pensando en un futuro jugador que recorriera los terrenos mediante físicas, como se explica en apartados más adelante.

El objetivo fue disponer de una cámara tipo "*shooter* en tercera persona", la cual está situada siempre detrás del jugador y está anclada a él, de forma que le sigue a todas partes. Al girar la cámara, ésta rota alrededor del jugador en esa dirección y siempre a la misma distancia de él (*offset*), describiendo así una circunferencia en la que el centro es siempre el jugador. Además, al rotarla se modifica también la orientación del jugador, que siempre apuntará en la misma dirección de la cámara. Ahora las teclas W, A, S, D moverán al jugador en vez de la cámara, y ésta seguirá al mismo a su distancia *offset* como se ha explicado anteriormente.

Para implementar esta funcionalidad, en un principio se calculó qué posición debía tener la cámara en cada momento mediante trigonometría. De esta forma, la cámara estaría situada siempre a una distancia *offset* del jugador detrás de él, calculando este punto exacto mediante el desplazamiento que hubiese producido la cámara al rotar; es decir, si la cámara desde el reposo (apuntando hacia delante y situado justo detrás del jugador, ángulo 0), rotaba 20º hacia la izquierda, ésta tendría que situarse 20º a la derecha respecto al jugador y a la misma distancia *offset* de él.

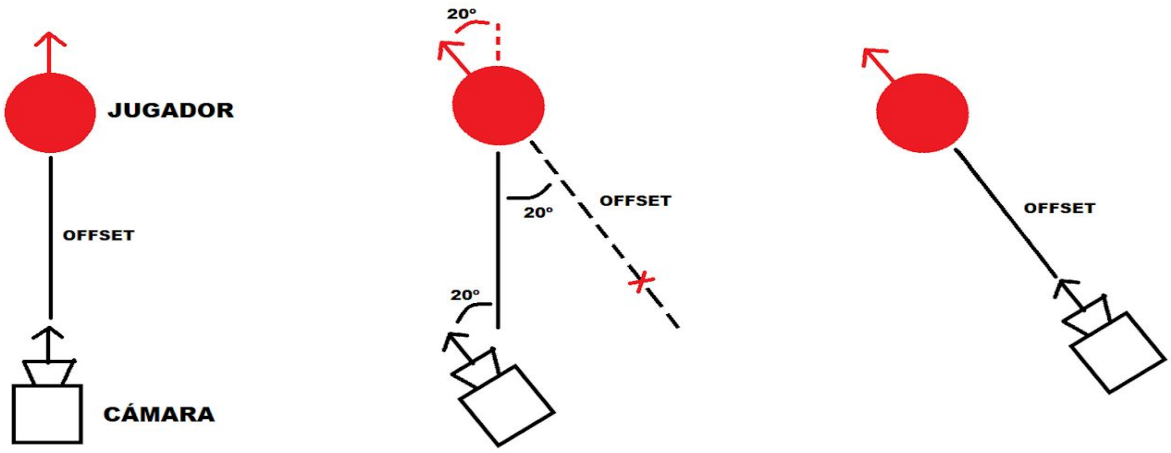

Figura 3.2-4: Primer método para hacer pivotar la cámara.

Este cálculo resultó ser completamente funcional y correcto; sin embargo, se sustituyó por uno más sencillo y rápido, y es que al estar orientado el jugador siempre en la dirección de la cámara, basta con situar a la cámara a *offset* unidades de distancia del jugador en la -dirección de apuntado de éste.

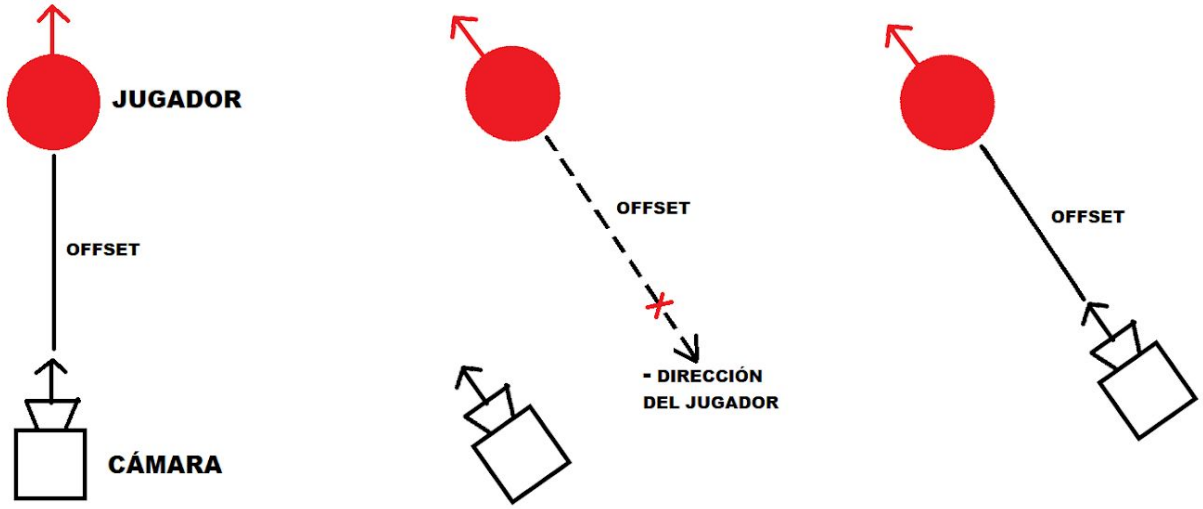

Figura 3.2-5: Segundo método para hacer pivotar la cámara

Cabe decir que, debido al seguimiento continuo de la cámara al jugador, se producía una especie de vibración de la cámara, ya que el jugador estaba constantemente colisionando con el suelo moviéndose levemente de arriba a abajo. Esto se solucionó poniendo un rango de movimiento que el jugador tiene que superar en cualquiera de los ejes para que la cámara se ajuste a éste, además de aplicar la fuerza normal del suelo sobre el jugador..

Una vez terminado, se añadió la posibilidad de habilitar o deshabilitar el modo de cámara en "tercera persona" mediante la pulsación de la tecla F, permitiendo así cambiar entre la cámara libre para poder explorar el terreno o volver a quedar ligado al jugador y moverlo a través de él.

3.3. Aplicación en OpenGL

Se ha generado la aplicación en OpenGL usando las últimas funcionalidades ofrecidas por la *API*. Debido a las distintas versiones de los *drivers* de OpenGL, no se sabe la localización de muchas de sus funciones en tiempo de compilación, por lo que deben ser almacenadas por el programador en punteros a funciones en tiempo de ejecución. Ya que este proceso es complejo y específico de cada sistema operativo, se ha utilizado también la librería GLAD, que gestiona esto por el desarrollador.

Toda la funcionalidad relacionada con la aplicación en OpenGL ha sido delegada en un gestor llamado "GLManager". Este gestor inicializa OpenGL, GLAD y pinta los objetos de vértices (*vertex buffer objects* o VBOs) que estén suscritos a OpenGL, siguiendo los pasos indicados en los diferentes subapartados del conocido tutorial *Learn OpenGL, extensive tutorial resource for learning Modern OpenGL* [9, 10, 11, 12, 13, 14]. La aplicación también cuenta con los *shaders* que se utilizarán y métodos para ejecutarlos y pintarlos en pantalla, cuya clase se creó basándose en *LearnOpenGL - Shaders* [15].

3.3.1. Shaders

Para implementar el algoritmo *Ray Marching* se hizo uso de *vertex shaders* y *fragment shaders* como se ha explicado en apartados anteriores. Para facilitar la tarea y poder reutilizar código en los *shaders*, se implementó desde código C++ una clase que unificara todos estos archivos mediante un sencillo sistema de inclusión similar al presente en la mayoría de lenguajes de programación, basándose en el repositorio *tntmeijs/GLSL-Shader-Includes: A utility class which adds a way to include external files in a shader file* [37]. De esta forma, en los *shaders* se pueden incluir otros haciendo uso de la palabra reservada *#include* seguido del nombre del archivo que se quiere incluir, y se hará una búsqueda y copia recursiva hasta realizar todas las inclusiones de archivos necesarias, volcando todo el código final en uno solo.

Posteriormente se añadió un tercer tipo de *shader*, el *compute shader*, cuya función es delegar en la GPU una serie de cálculos, liberando a la CPU de esa carga de trabajo. El problema principal que surgió a la hora de añadir estos *shaders* al proyecto fue recoger la información

calculada en ellos. Se llegó a una solución acertada gracias a la existencia de los *ShaderStorageBufferObject* o SSBOs y de la posibilidad de enlazarlos a una región de memoria compartida entre CPU y GPU.

3.3.2. Comunicación entre GPU y CPU

Con el objetivo de implementar un sistema físico sencillo que será descrito en apartados posteriores, se ha requerido establecer un canal de comunicación bidireccional entre la tarjeta gráfica y la CPU.

La solución alcanzada en OpenGL es sencilla. Se ejecuta un *shader* de cómputo o *compute shader*, se enlaza con una estructura de datos denominada *StorageBufferObject*, definida respetando el convenio de nombres descrito en *LearnOpenGL - OpenGL* [11]. Tras ello, se espera a la finalización del *shader* y se recuperan los valores calculados en el mismo para su posterior uso en la CPU.

El *shader* de cómputo puede recibir parámetros a través de esta estructura, además de poder guardar en la misma información calculada en el *shader* para su uso en la CPU. En este contexto se necesita, entre otros, un vector que almacene la posición que se quiere calcular dentro del *shader*.

Figura 3.3.2-1: Esquema básico de la funcionalidad del *compute shader*.

3.4. Aplicación en Vulkan

Se ha generado la aplicación en Vulkan usando las últimas funcionalidades ofrecidas por la *API*. Para ello se han seguido los pasos indicados en la documentación de Vulkan, con apoyo de *Vulkan CookBook* [30], descritos a continuación.

Con el objetivo de inicializar Vulkan y realizar los ajustes necesarios para poder interactuar con el hardware gráfico a través de comandos, se requiere seguir una serie de indicaciones iniciales. Se necesita elegir el dispositivo físico o GPU adecuada para lo deseado, enlazar dicha GPU con un dispositivo virtual, administrar las capas de validación o *validation layers*, permitiendo así visualizar los mensajes de control y error transmitidos por Vulkan, crear las diferentes estructuras necesarias para crear y usar la cadena de intercambio de *framebuffers* utilizados por la tarjeta gráfica o la *Swap Chain*, establecer la tubería gráfica, indicando los *shaders* que serán usados posteriormente en el renderizado y el orden en el que serán ejecutados, el control de acceso a recursos compartidos entre diferentes hebras lanzadas por Vulkan y la gestión de memoria compartida para comunicar los *shaders*, ejecutados en la tarjeta gráfica, con el código ejecutado en CPU. Con todo ello, inicialmente se consiguió renderizar un triángulo básico como viene indicado en *Drawing a triangle - Vulkan Tutorial* [30] y, posteriormente, los terrenos fractales.

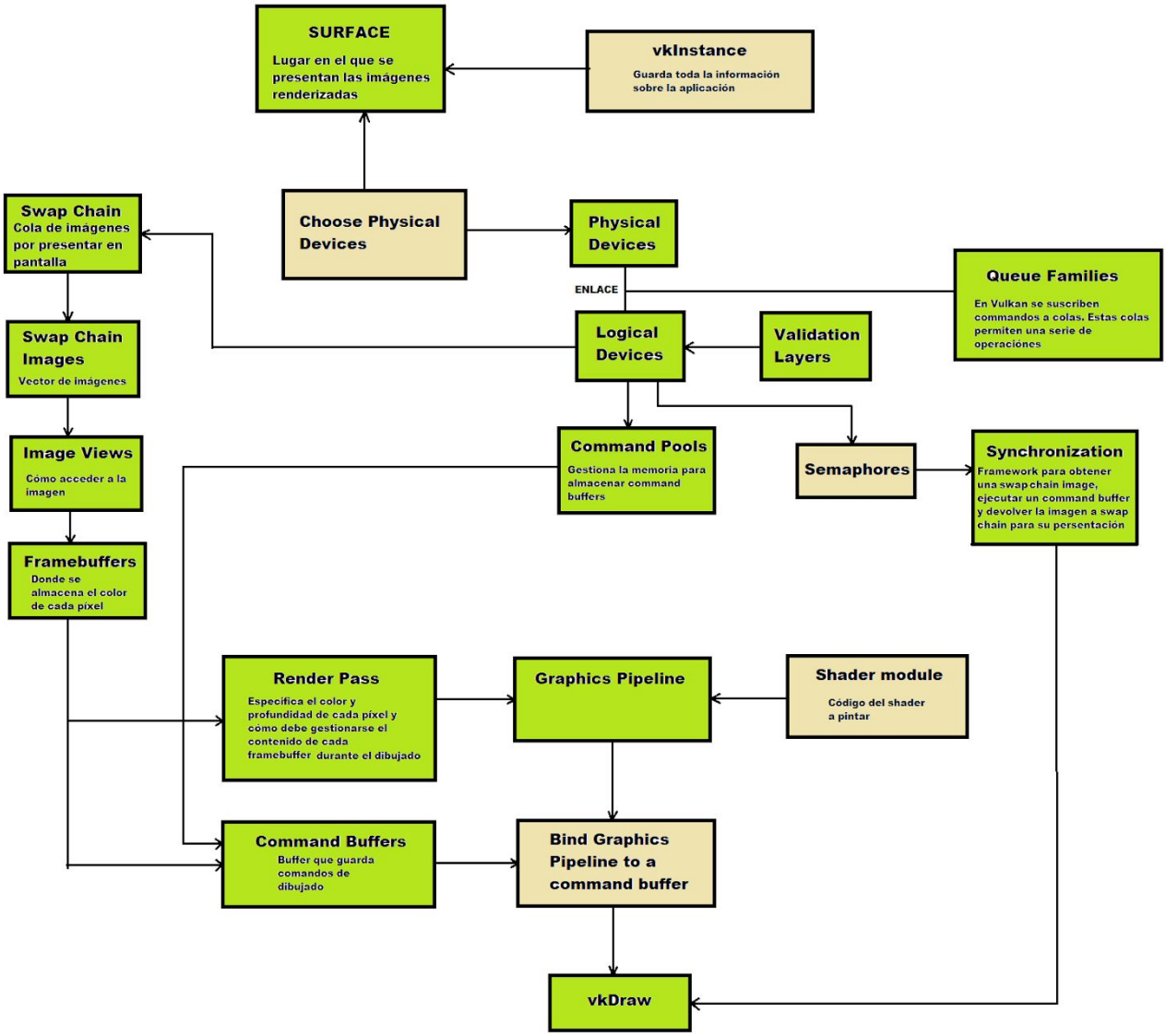

Figura 3.4-1: Esquema básico de inicialización y dibujado según los pasos del tutorial de Vulkan.

Toda la funcionalidad relacionada con la aplicación en Vulkan ha sido delegada en un gestor llamado "VulkanManager". Este gestor inicializa Vulkan, almacena los vértices e índices necesarios al igual que en OpenGL basándose en *Vertex buffers - Vulkan Tutorial* [32], crea una tubería por cada conjunto de *shaders* recibidos, permitiendo la ejecución simultánea de *shaders*, actualiza y recibe la información de los *shaders* y dibuja el siguiente *frame*.

3.4.1. Shaders

El objetivo principal fue no sólo introducir *shaders* en la *API* de Vulkan, sino que además éstos fuesen capaces de funcionar correctamente tanto en OpenGL como en Vulkan con las mínimas modificaciones; es decir, dados los mismos *vertex*, *fragment* y *compute shader*, que éstos funcionasen igual en ambas *APIs*.

El primer obstáculo que se encontró es el sistema de coordenadas utilizado por cada *API*. Vulkan utiliza un sistema de coordenadas de mano derecha, mientras que OpenGL hace uso de un sistema de mano izquierda. Este problema resultaba en que el origen de la ventana en Vulkan se encuentra en la esquina superior izquierda de la misma y su eje Y apunta hacia abajo, y en OpenGL el origen de la ventana está situado en la esquina inferior izquierda y su eje Y apunta hacia arriba, dando lugar a que todas las imágenes mostradas en pantalla usando Vulkan estén invertidas en el eje Y con respecto a las mostradas por OpenGL, como se expone en el blog "Vulkan's coordinate system" [5].

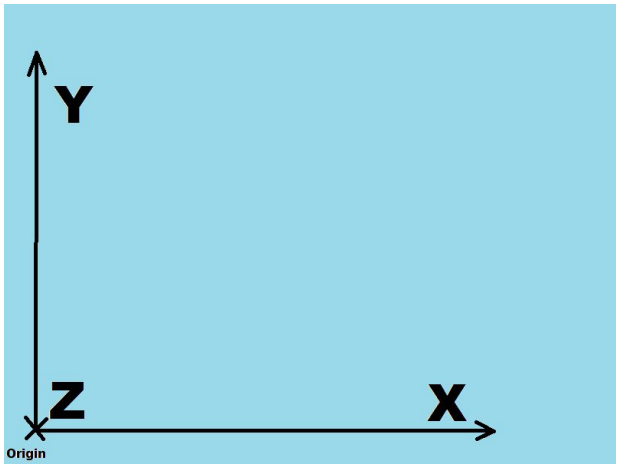

Figura 3.4.1-1: Sistema de coordenadas de OpenGL

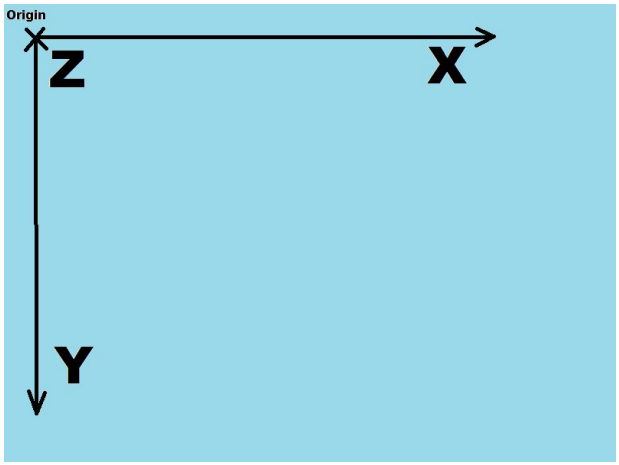

Figura 3.4.1-2: Sistema de coordenadas de Vulkan

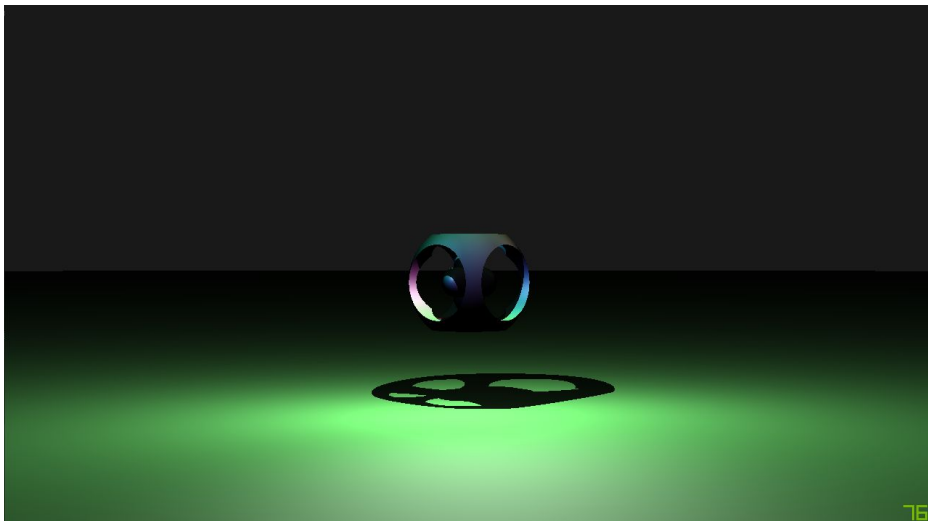

Figura 3.4.1-3: Shader scene0 mostrado con OpenGL, coordenadas de mano izquierda

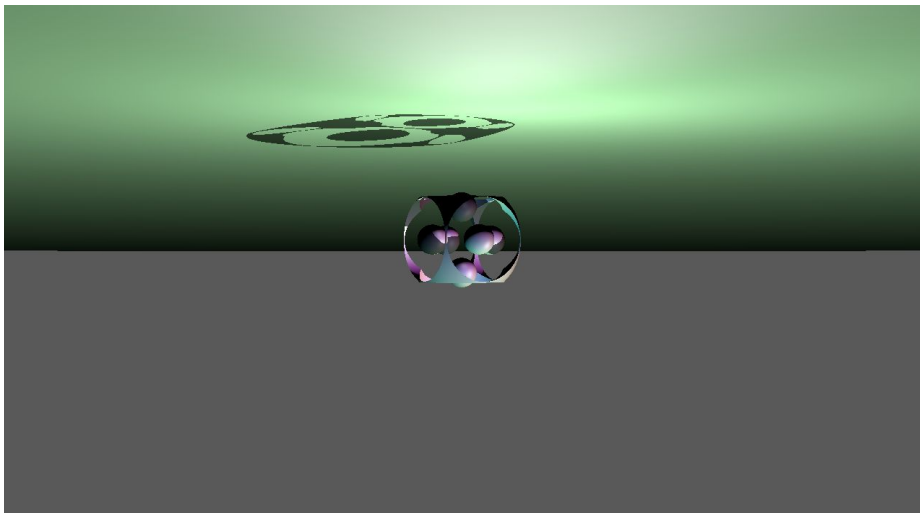

Figura 3.4.1-4: Shader scene0 mostrado con Vulkan, coordenadas de mano derecha

La solución a este problema, atendiendo al artículo *Flipping the Vulkan viewport*, Sascha Willems [53] y con apoyo de su implementación en *glsl - Flipping the viewport in Vulkan - Stack Overflow* [43], fue utilizar la extensión de Vulkan *VK_KHR_Maintenance1*, añadida a partir de la versión 1.0, que permite pasarle al puerto de vista o *viewport* de la aplicación alturas negativas. De esta forma, se estableció que el origen de la ventana se encontrase en 0 en el eje X (sin modificar) y *WindowHeight* en el eje Y, situándose así, al igual que en OpenGL, en la esquina inferior izquierda. A esto se le suma que el alto del *viewport* se estableció como

-*WindowHeight* para que, al igual que en OpenGL, el eje Y apunte hacia arriba. Además, la componente Y de las coordenadas *uv* de los *shaders* es multiplicada por -1 en el caso de encontrarse en Vulkan, para invertir su eje y coincidir así con lo mostrado mediante OpenGL. Para ello se consultó "*Fix" for DirectX Flipping Vertical Screen Coordinates in Image Effects not Working - Unity Forum* [51], y con el fin de elegir la cara visible (o *front*) de los triángulos (depende del orden de sus vértices), se analizó *VkFrontFace(3)* [29].

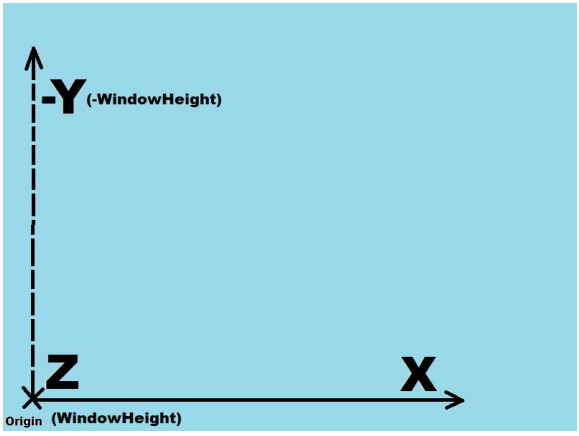

Figura 3.4.1-5: Sistema de coordenadas de Vulkan adaptado

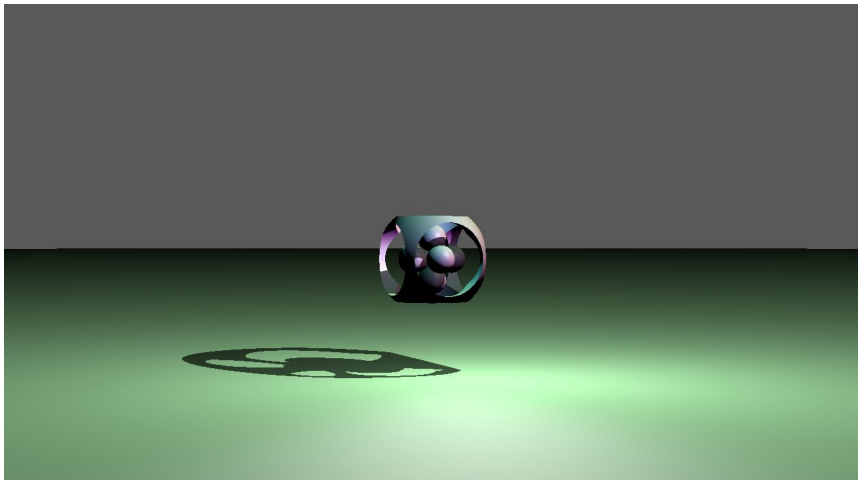

Figura 3.4.1-6: Shader scene0 mostrado con Vulkan, con las coordenadas adaptadas

Otro de los problemas encontrados al adaptar los *shaders* a ambas *APIs* fue la imposibilidad de tener funciones dentro de los structs, característica permitida por GLSL, pero

no soportada por SPIR-V, por lo que hubo que adaptar los *structs* de las geometrías primitivas para tener sus funciones de distancia fuera de ellas, como se puede observar en el apartado 2.1.3. Problemas similares relacionados con la sintaxis de los *shaders* entre ambas *APIs* fueron solucionados sin mayor problema.

Posteriormente, se implementó en OpenGL la posibilidad de recibir las variables *uniform* mediante un *struct*, ya que en Vulkan son recibidas de esta manera y no en variables por separado como se tenía en un principio en OpenGL, como se indica en *Uniform buffers - Vulkan Tutorial* [33].

Después, puesto que Vulkan necesita interpretar los *shaders* en el lenguaje SPIR-V, se hizo uso de código condicional para, en caso de encontrarse en dicha *API*, compilar el código del *shader* a SPIR-V a través de un archivo .bat, en vez de ser utilizado directamente como en el caso de OpenGL.

Una vez realizados estos ajustes y teniendo una única versión de los *shaders*, éstos funcionaron correctamente independientemente de la *API* que los esté ejecutando, facilitando así el desarrollo de futuros *shaders*. También fue introducida la posibilidad de crear múltiples tuberías de renderizado en la aplicación para poder tener diferentes conjuntos de *shaders*.

Por último, la implementación de los *shaders* de cómputo se basó en los mismos principios de creación de los *shaders* anteriores. Se debe crear una tubería de cómputo, así como un *buffer* de comandos específico para el *shader*. Una vez logrado, el problema surgió, al igual que en OpenGL, en el acceso a la información calculada. Se consiguió solucionar vinculando manualmente un *buffer* a la ejecución del *shader* para posteriormente acceder a dicha región de memoria y realizar la copia a un objeto almacenado en la CPU.

3.4.2. Comunicación entre GPU y CPU

La realización de esta tarea no fue tan simple como la descrita anteriormente en el apartado de OpenGL. Sin embargo, el concepto principal es el mismo. Como se describe en

sheredom/VkComputeSample [42], se necesita de una estructura que sea rellenada en la GPU para poder usar esa información en la CPU. Para ello se vuelve a hacer uso del mismo *compute shader* y de la estructura *StorageBufferObject*. Con el objetivo de poder hacer uso del *compute shader* se requiere, al igual que para utilizar un *vertex shader* y un *fragment shader*, de una nueva *compute pipeline* o tubería de cómputo. Se ha de crear dicha tubería y todos los parámetros relacionados con la misma, entre ellos el conjunto de descriptores o el *buffer* de comandos, necesario para la ejecución de los *shaders*.

Tras haberse realizado el proceso de creación, el siguiente paso es ejecutar el propio *shader* y obtener los valores calculados en el mismo. Para ello, la ejecución de la tubería gráfica jugó un papel importante, ya que Vulkan trata de ser consistente a la hora de realizar tareas parecidas, es decir, se realizan los mismos pasos con una serie de pequeñas modificaciones en los parámetros recibidos por los métodos de Vulkan. A alto nivel, el proceso consiste en, una vez cada *frame*, ejecutar el *buffer* de comandos, establecer un control de acceso a memoria (*memory barrier*) con el objeto de proteger los cálculos realizados en el *shader*, vincular y comenzar la tubería de cómputo, acceder a la memoria actualizada en la tarjeta gráfica y obtener los datos calculados en la misma, estableciendo así una comunicación exitosa entre GPU y CPU.

4. Fractales

Una vez aprendido el funcionamiento de los *shaders* en ambas *APIs* junto con el algoritmo de *Ray Marching*, se implementaron varios fractales básicos, al ser uno de los objetivos principales del proyecto. Como inspiración de con qué fractales empezar, sirvieron las siguientes páginas *3D fractal trip - Far away…* [3] y *7 Free 3D Fractal Generation Software - CadNav* [4].

Para poder dibujar cualquier fractal con *Ray Marching*, se necesitan sus fórmulas de distancia, como viene indicado en *fractals, computer graphics, mathematics, shaders, demoscene and more* [44] y demostrado en los capítulos "*Procedural fractal terrains*" [23] y "Mojoworld" [24] del libro *Texturing and modeling: a procedural approach*. Al ser el fractal *Mandelbrot* (fractal bidimensional) uno de los fractales más conocidos, se intentó implementar la versión tridimensional de éste: el fractal *Mandelbulb*. En un principio se utilizó como inspiración el fractal dado en *Shader - Shadertoy BETA - Mandelbulb-derivative* [46], aunque por motivos de eficiencia se decidió descartar. Finalmente, se obtuvo tanto la fórmula de distancia como la fórmula para colorear el mismo de *Shader - Shadertoy BETA - mandelbulb raymarched* [40], obteniendo el siguiente resultado:

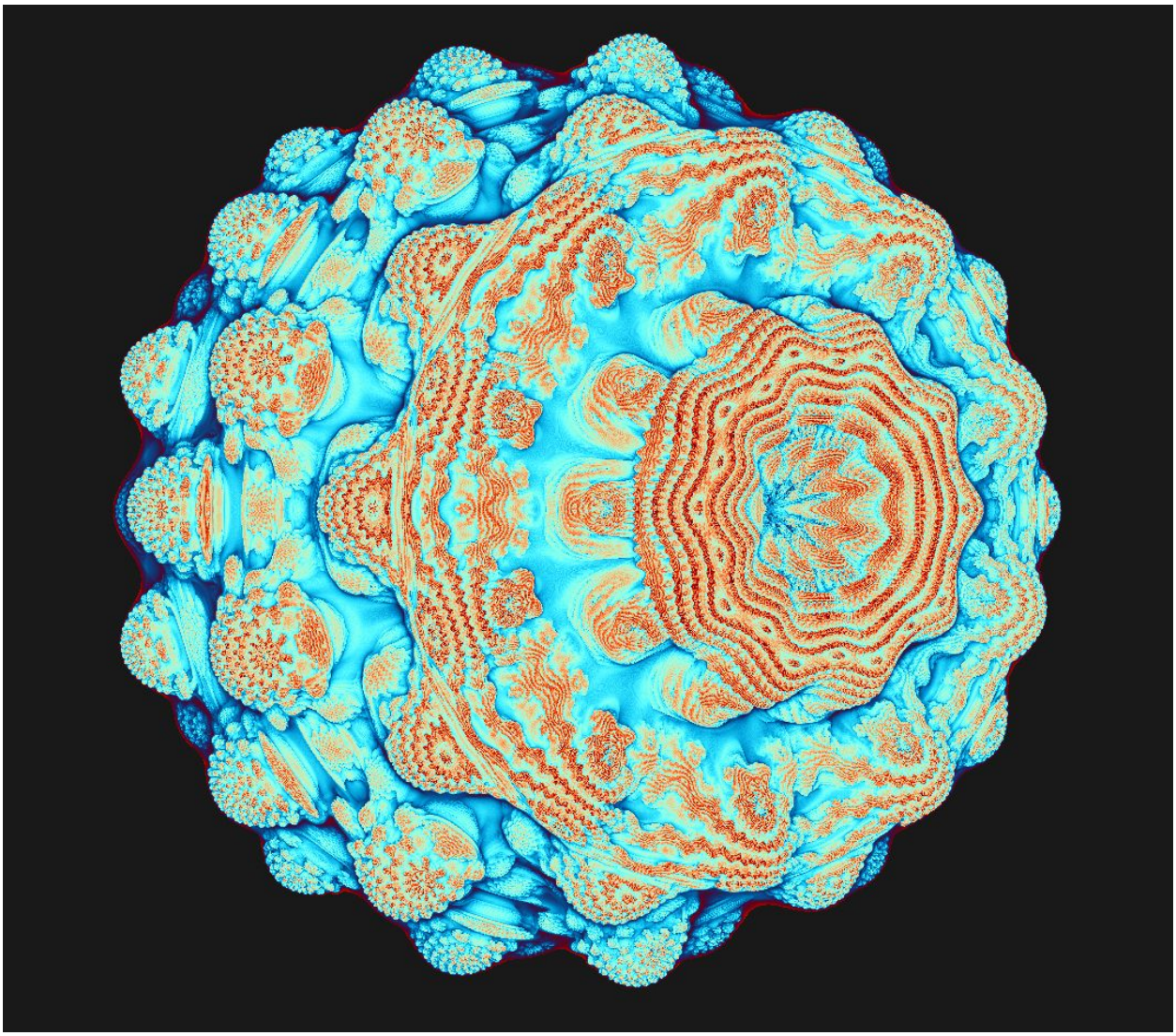

Figura 4-1: Fractal Mandelbulb

Una vez conseguido pintar el primer fractal, se investigó la forma de incluir un segundo, el fractal *mandelbox*. Como documentación, se visitaron diversos sitios web con información sobre este tipo de fractal, utilizados como apoyo para la correcta implementación de éste: Fractal en 3D… Un viaje por el interior de un Mandelbox [1], *Shadertoy*[8][26], "*Distance Estimated 3D Fractals (VI): The Mandelbox*" [27] y *A Mandelbox distance estimate formula* [28].

Como inspiración final se utilizó el fractal generado en *Shader - Shadertoy BETA - Mandelbox rotating* [7], y dado que es de la misma familia que el anterior, fue adaptado para que la forma de pintarse fuera la misma y la fórmula de distancia fuese muy parecida. Debido a su similitud, se decidió usar la misma estructura de código que para el anterior, cada uno en un archivo diferente definiendo una función *SDF* y los parámetros necesarios de esta misma, para que en el archivo en donde se ejecute el algoritmo de *Ray Marching* solo hay que cambiar qué archivo se incluye (*Mandelbulb* o *Mandelbox*) para renderizar el fractal deseado. Se reorganizaron los archivos para tener funciones genéricas para todos los fractales que no se repitiesen, aunque éstos utilizan también métodos antiguos como los de rotación de primitivas.

Resultado del fractal *Mandelbox*:

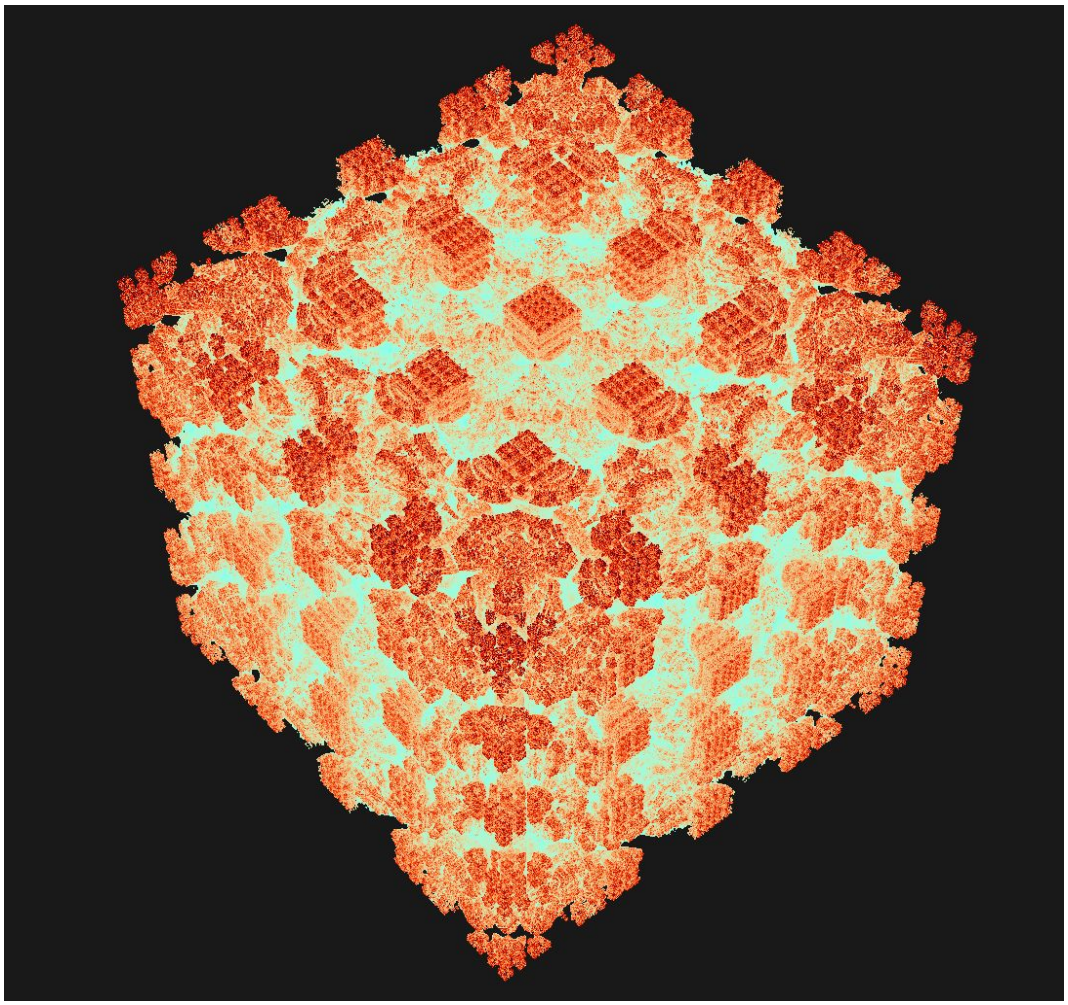

Figura 4-2: Fractal Mandelbox

Después, se añadió el mismo fractal *mandelbox* pero versión túnel, permitiendo al usuario introducirse fácilmente en él, basado en *Shader - Shadertoy BETA - Mandelbox Tunnel* [22]:

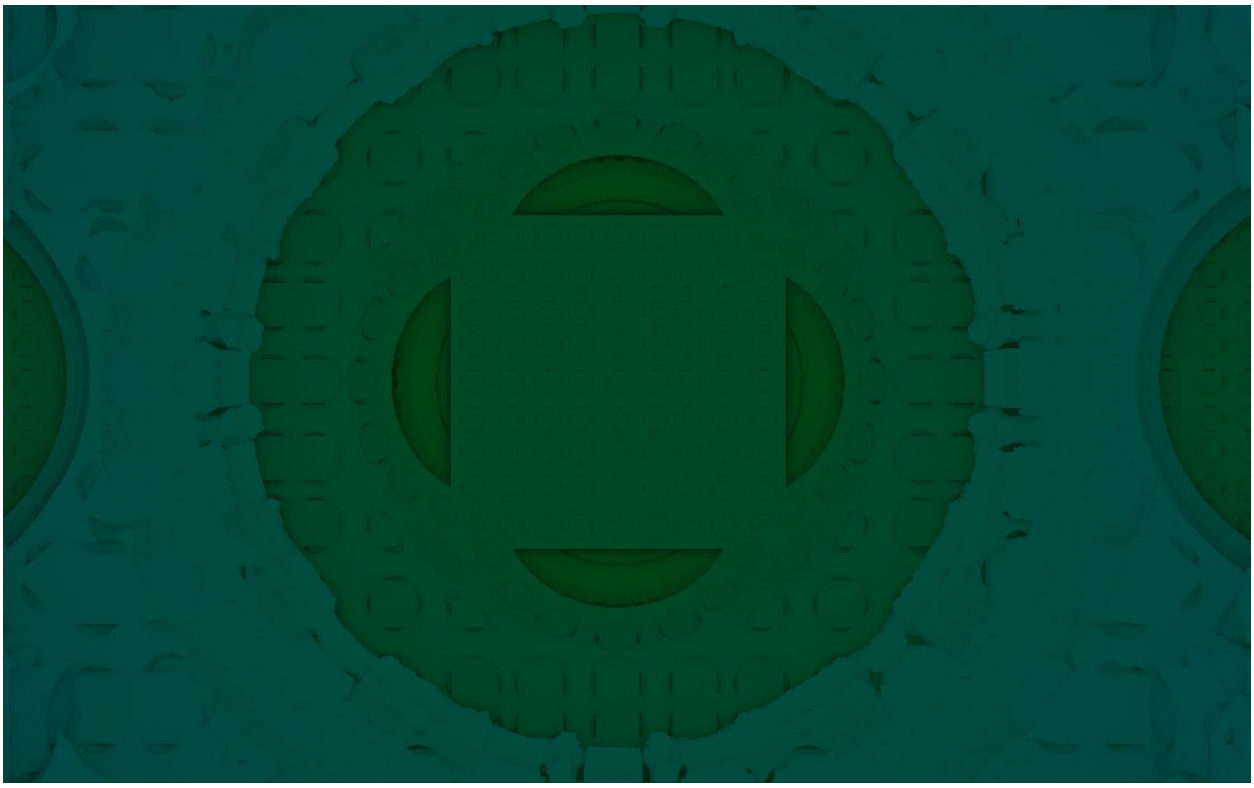

Figura 4-3: Fractal Mandelbox túnel

4.1. Fractales de terreno

Posteriormente, se implementó un fractal de terreno, para en un futuro poder simular físicas en él. Se visitaron numerosos ejemplos del uso de fractales en la generación de terrenos, como *Shader - Shadertoy BETA - Fractal terrain generator* [38] o *Shader - Shadertoy BETA - Elevated* [45]. Finalmente, se introdujo un terreno basándose en el ejemplo *Shader - Shadertoy BETA - Polar Night* [49] y con ayuda de distintas fuentes como el artículo *Fractales 3D* [39]. Dado que este fractal era bastante diferente a los anteriores, se tuvo que reorganizar el código de los *shaders* para que de nuevo sólo hubiera que cambiar el archivo del fractal que se incluía,

moviendo diversas funciones que antes tenían los otros dos fractales en común a un archivo en concreto para ellos, dando como resultado un archivo con funciones para los terrenos fractales utilizado por el nuevo terreno, y otro con funciones de fractales puros, que comparten los dos anteriores.

El resultado final se puede observar en la siguiente imagen:

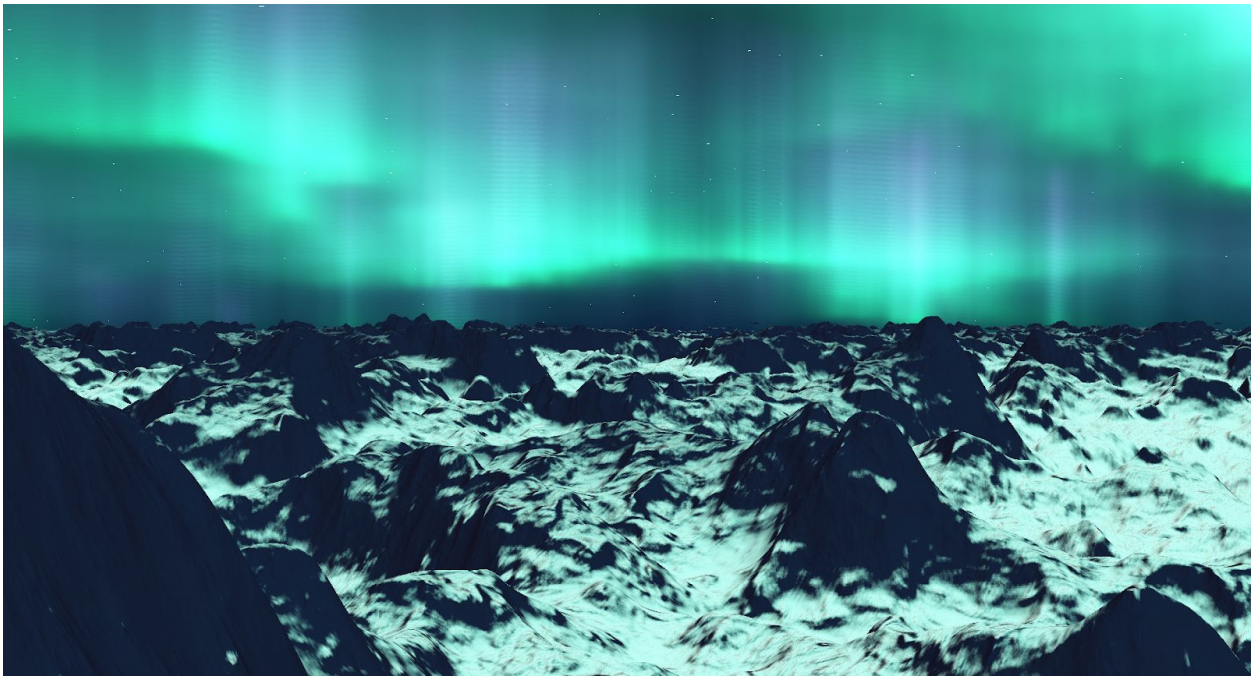

Figura 4.1-1: Fractal de terreno (snow terrain)

Otros fractales de terreno implementados:

Fractal simulando el océano mediante técnicas similares a *snow terrain*, implementado basándose en *Shader - Shadertoy BETA - Oceanic* [25]:

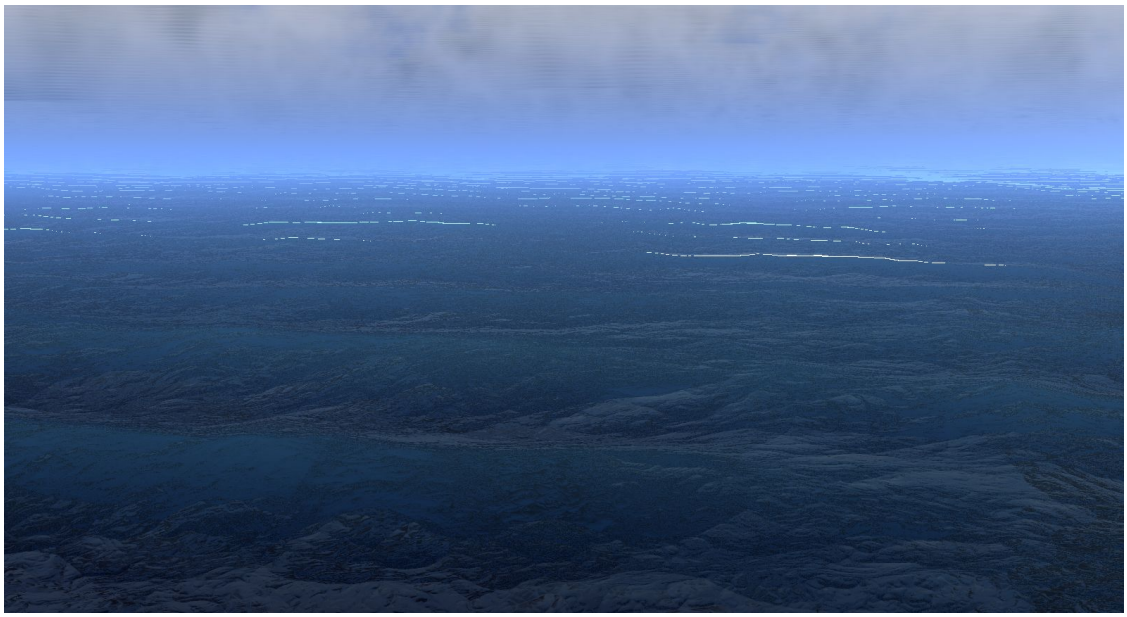

Figura 4.1-2: Fractal de terreno (ocean terrain)

Fractal con forma de planeta realizado mostrando las físicas tipo “planeta”. Implementado gracias a las indicaciones en *Shader - Shadertoy BETA - Planet Fall* [47]:

Figura 4.1-3: Fractal de terreno (autumn terrain)

5. Físicas

Teniendo ya fractales de terreno, el siguiente objetivo fue que interactuasen físicamente con otros objetos no basados en *RayMarching* mediante colisiones.

5.1. Físicas en los fractales de terreno

Lo primero fue conseguir una *hitbox* (forma invisible empleada para detectar colisiones entre objetos) esférica para que el objeto interactuase con el fractal. La primera idea para esta *hitbox* fue encontrar *n* vectores directores equidistantes desde el centro de la *hitbox*, y buscando diferentes formas, se eligió el siguiente algoritmo: la esfera de Fibonacci, descrito en la publicación *Evenly distributing n points on a sphere - Stack Overflow* [2].

Lo ideal de este algoritmo es el poder definir con mayor o menor precisión la *hitbox* simplemente cambiando el número de muestras que recibe, generando así los puntos en la superficie de la esfera y, por tanto, el número de puntos de colisión. El resultado es el siguiente:

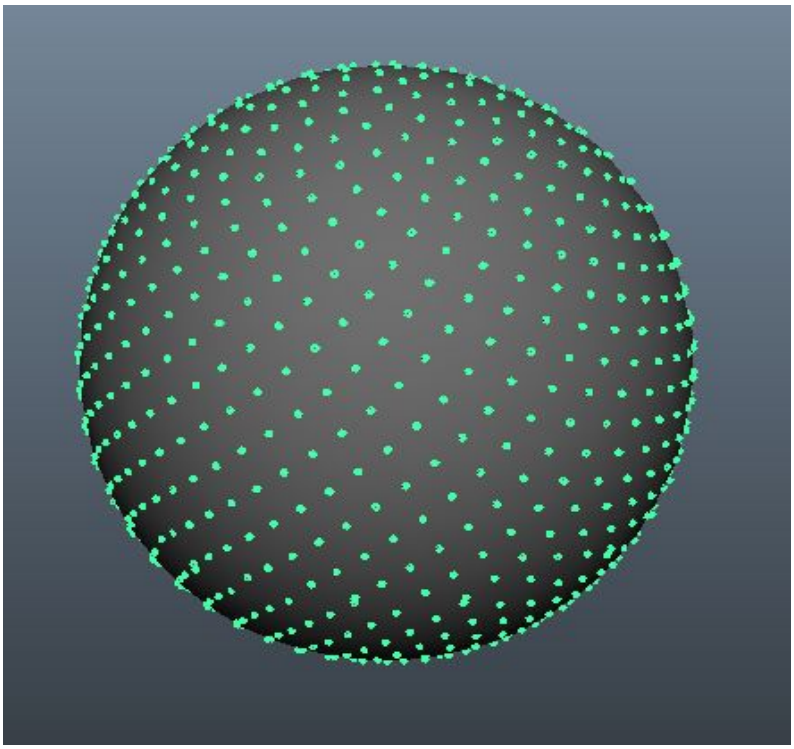

Figura 5.1-1: Esfera de Fibonacci con 1000 muestras.

Una vez conseguida la *hitbox* de la esfera, se volvería a utilizar el algoritmo de *Ray Marching* para calcular si hay colisión o no, lanzando rayos desde el centro de la misma, en cada una de las direcciones de los puntos calculados anteriormente. Teniendo los puntos equidistantes en la esfera, las direcciones de colisión (vectores directores equidistantes) se obtienen uniendo cada uno de esos puntos con el centro de la esfera. En el *compute shader* se realizará el siguiente cálculo: por cada vector director de colisión, se ejecuta el algoritmo Ray March con origen en el centro de la *hitbox*, devolviendo así la distancia de choque más cercana en esa dirección. Si esa distancia es menor que el radio de la *hitbox*, significará que ésta está colisionando con un objeto, y es recolocada.

```
for(i = 0; i < COLLISION_SAMPLES; i++){
      dist = rayMarch(centro, fibonacci_dirs[i], MIN_DIST, MAX_DIST);
      if(dist < radius){
      //si la distancia desde el centro al fractal es menor que el radio,
      se produce colisión y la esfera se recoloca
           position += abs(dist - radius) * -(dirs[i]);
      }
}
```

El siguiente paso después de la colisión entre esfera y fractal fue crear un modelo de físicas newtoniano, incluyendo aceleraciones (fuerzas) y velocidades. Se añadieron fuerzas de la gravedad, fuerzas para que la esfera se pudiera mover por el fractal (utilizando teclas W, A, S, D) y fuerza de rozamiento con el mismo. Cada una de estas fuerzas es añadida a una fuerza total que recibe el *compute shader* mediante el *struct ShaderStorageBufferObject*, dando lugar al siguiente código:

```
posicion += velocidad * deltaTime;           // se aplica la velocidad
vec3 aceleracionTotal = fuerzaTotal / masa;  // se calcula la aceleración
velocidad += aceleracionTotal* deltaTime;    // se aplica la aceleración
velocidad *= power(rozamiento, deltaTime);   // se aplica el rozamiento
```

Para dar un comportamiento más realista, más tarde se añadió un rozamiento en el aire distinto al del suelo, además de impedir que la esfera se pueda mover cuando no esté colisionando con el fractal.

Resultado final:

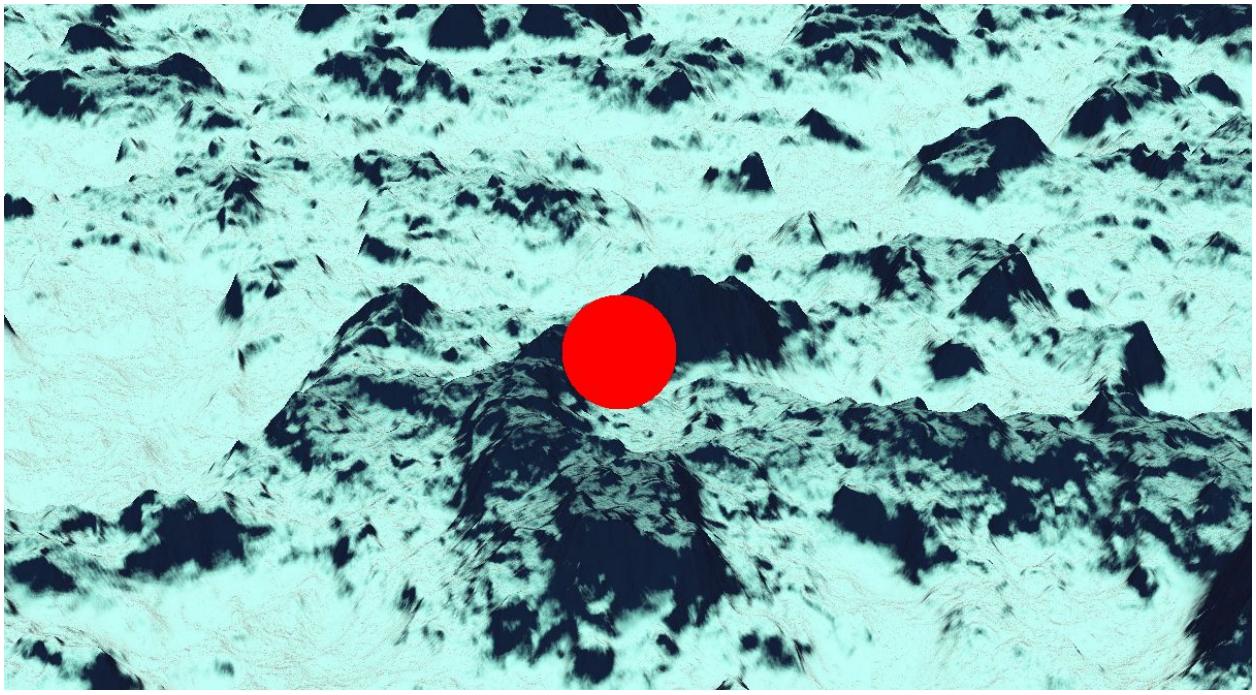

Figura 5.1-2: Esfera colisionando con el fractal de terreno.

5.2. Físicas en el resto de fractales

Una vez conseguidas las físicas de colisión en el terreno fractal, éstas se ampliaron para adaptarse al resto de fractales del trabajo. Mientras que este primer terreno está situado horizontalmente y la gravedad ejerce su fuerza hacia abajo, pegando el objeto al suelo, en fractales como Mandelbulb o Mandelbox, se necesita que la gravedad ejerza su fuerza hacia el centro del fractal, atrayendo los objetos hacia sí para que estos puedan caminar por su superficie, como si de un planeta se tratase.

Para ello, se añadió una vez por iteración el cálculo de la dirección de la gravedad como el vector comprendido entre la posición del objeto a colisionar y el centro del fractal. En el caso de tener terrenos horizontales como el visto anteriormente, se mantuvo el vector director de la gravedad como (0, -1, 0), es decir, ejerciendo su fuerza hacia abajo.

Después, se creó un *compute shader* para cada fractal, en los que se calculan los puntos de colisión y la recolocación del objeto de la misma forma que se explicó anteriormente, pero cambiando la función SDF de *RayMarch* por la del fractal correspondiente.

Llegado este punto, se obtuvo una *hitbox* capaz de caminar correctamente sobre la superficie de todos los tipos de fractales realizados.

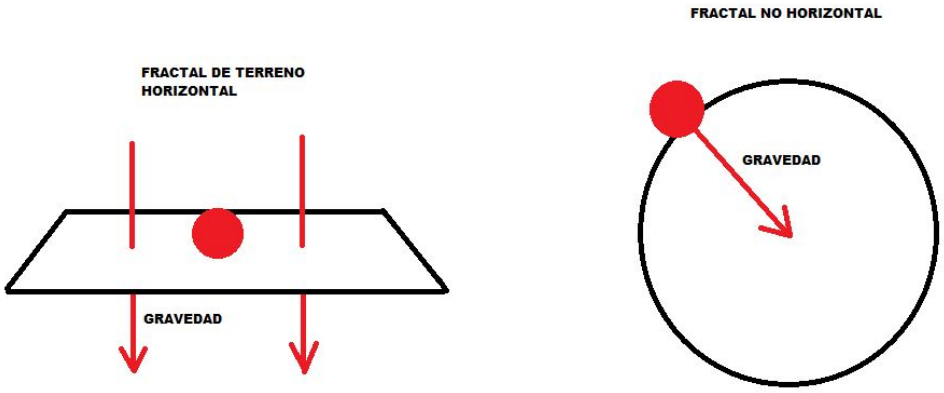

Figura 5.2-1: Diferencia en la actuación de la gravedad entre fractales

Sin embargo, en los fractales de tipo "planeta" (no horizontales), como Mandelbulb o Mandelbox, también se tendría que orientar al futuro modelo asociado para que éste apareciese situado en perpendicular a la superficie sobre la que se encuentre, para dar la sensación de estar caminando sobre ella. Además, también habría que modificar la orientación de la cámara para que siguiese al objeto correctamente a lo largo de todo el fractal.

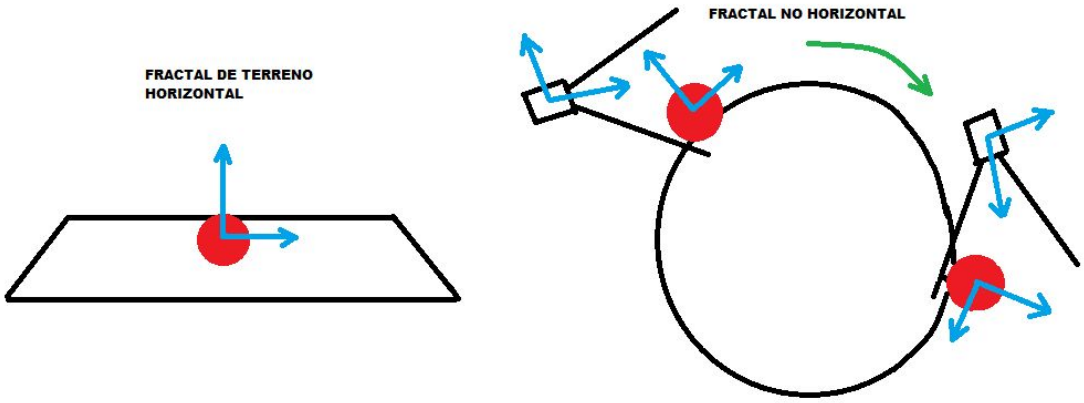

Figura 5.2-2: Necesidad de ajustar la orientación de la cámara y objetos en los fractales no horizontales

Para ello, inicialmente se calculó el nuevo sistema de coordenadas local que tendría el objeto, igualando su eje *y* a la inversa de la dirección de la gravedad, es decir, perpendicular a la superficie del planeta. El resto de ejes, *x* y *z*, se calcularon a partir del eje *y* conforme a lo indicado en *How do you build a rotation matrix from a normalized vector?* [41], de la siguiente forma:

```
glm::vec3 nuevaY = -direccionGravedad;
glm::vec3 nuevaX = normalize(productoCruzado(vec3(0, 0, 1), nuevaY));
glm::vec3 nuevaZ = normalize(productoCruzado(nuevaY, nuevaX));
```

Una vez calculado este nuevo sistema de coordenadas local, se sustituyó por el sistema anterior en la matriz de modelado, permitiendo así que el objeto quedase siempre perpendicular a la superficie. Este cálculo y actualización del sistema local se realizaba una vez por iteración.

Lo último por adaptar para los fractales de tipo “planeta” fue la cámara. El objetivo era que ésta siguiese al objeto a lo largo de todo el fractal orientándose de la misma forma que él, es decir, viendo siempre la superficie del fractal debajo y al objeto sobre ella. Para conseguirlo, se barajaron varias posibilidades, entre ellas igualar el nuevo sistema de coordenadas local del objeto al de la cámara, o calcular cuántos grados se había desplazado el objeto en *z* (cabeceo hacia delante o hacia atrás) o *x* (cabeceo hacia izquierda o derecha), y simularlos en la cámara

mediante desplazamientos en su *pitch* o su *roll*, respectivamente, siguiendo las explicaciones de *Find the angle between two vectors from an arbitrary origin - Stack Overflow* [47].

Aparte de pequeños problemas de imprecisión en estos métodos, se generó un grave problema, y es que la cámara invertía su perspectiva al dar una vuelta completa sobre sí misma en el eje *z* o *x*. Es decir, la cámara invertía su visión al posicionarse "boca a bajo", y se empezaba a ver el modelo desde delante, invirtiendo todos los controles.

Al no encontrar ninguna solución aparentemente sencilla, además de los problemas que ya producía esta solución, se optó por adoptar otra estrategia: mantener la orientación del objeto colisionable y la cámara como si de un fractal horizontal se tratase (como se tenía en un principio, mirando siempre hacia delante y sin inclinaciones en x o z), y mover al fractal en vez de al objeto. De esta forma, la sensación resultante sería la de que el objeto esté avanzando siempre por la superficie del terreno de la forma que se pretendía en un principio.

Para ello, se amplió la recogida de *input* de usuario de forma que, en caso de encontrarse en un terreno horizontal, seguir manteniendo las direcciones del objeto dadas por W, A, S, D para posteriormente aplicarles las físicas y mover el objeto; pero en caso de encontrarse en un fractal no horizontal, como podrían ser Mandelbulb o Mandelbox, utilizar esas direcciones para girar dichos fractales desde los *shaders* mediante funciones de rotación que ya estaban implementadas anteriormente.

Haciendo esta diferenciación entre los tipos de fractal, se pudo ampliar las físicas iniciales para adaptarse a cualquiera de ellos y dar la sensación de recorrer su superficie.

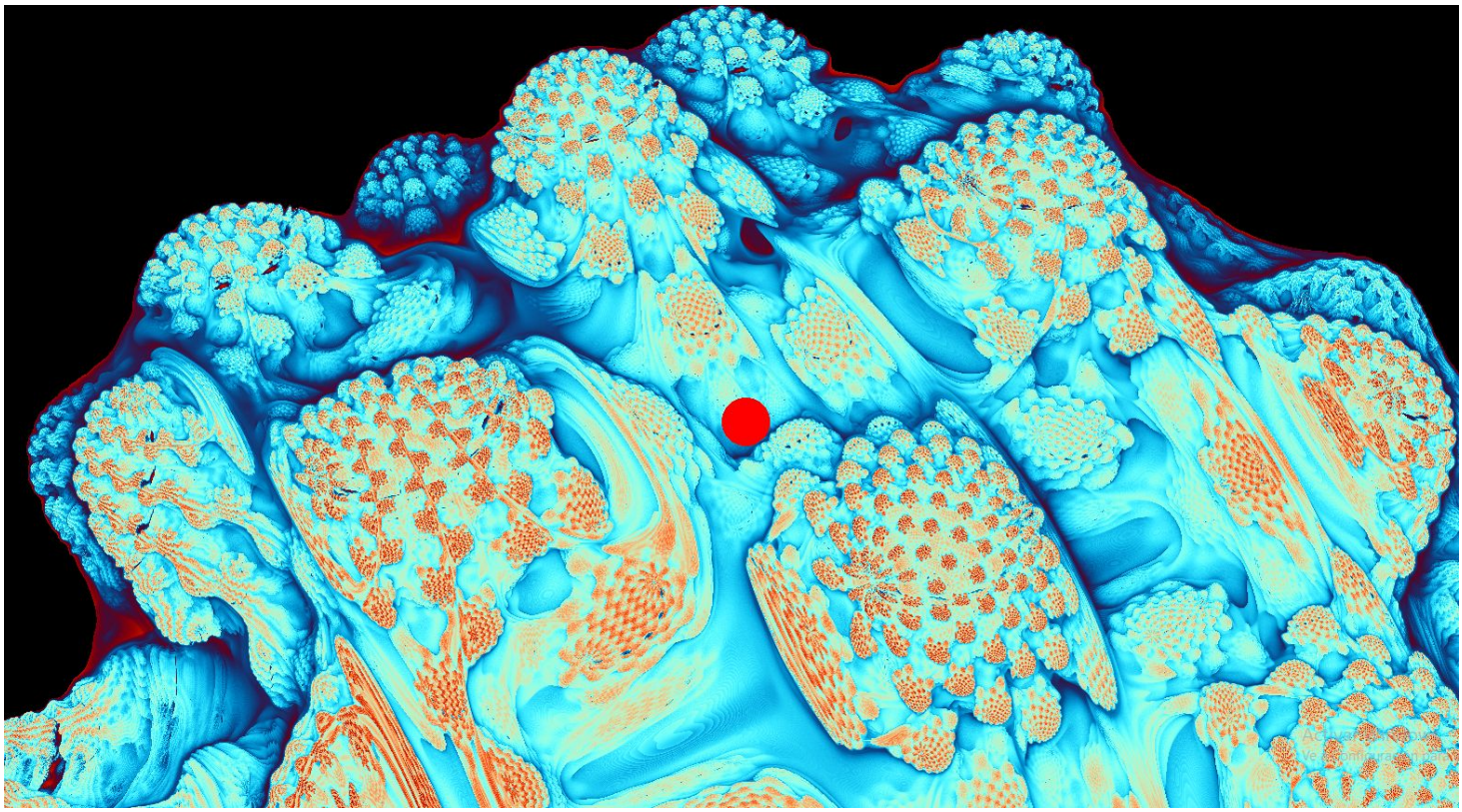

Figura 5.2-3: Resultado final: esfera colisionando con cualquier tipo de fractal

6. Modelos

El objetivo de este apartado es añadir a la escena modelos de mallas de triángulos que se rendericen de forma habitual. Para esto será necesario tener una secuencia de shaders distinta que se encargue de pintar un modelo, conservar la anterior secuencia de shaders que pinta el fractal, y mediante la técnica de depth testing, combinar ambos de forma que se dibujen conjuntamente.

Para ello, ambos procesos deberán escribir en el *buffer* de profundidad, que después será utilizado para visualizar una combinación de ambos en el *frame buffer* final, dependiendo de la profundidad a la que se encuentre cada píxel. De esta forma, el modelo aparecerá integrado sobre el terreno del fractal, pintándose sobre él en los momentos en los que se encuentre por delante (más cerca de la cámara), y detrás de él cuando deba quedar oculto por protuberancias del terreno, como montañas.

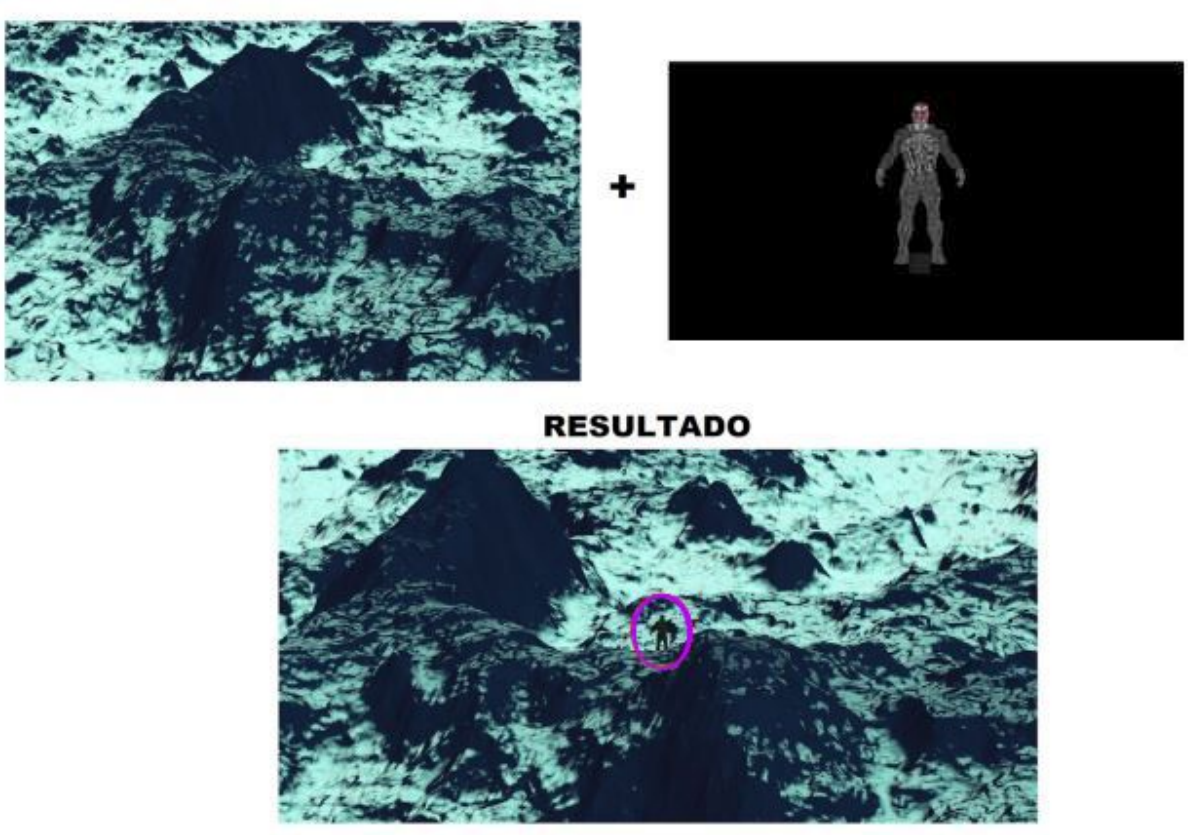

Figura 6-1: Resultado de la combinación del shader de terreno y el shader de modelo

Sin embargo, para obtener esta funcionalidad en *OpenGL* y *Vulkan*, la implementación es ligeramente distinta para cada una de las dos *APIs*.

6.1. Modelo en OpenGL

En OpenGl, se siguió el tutorial *Learn OpenGL,* en concreto los subapartados *Assimp* [18]*, Mesh* [19]*, Model* [20]*, Depth testing* [21]. Para la carga de modelos, se utilizó la librería *Assimp* como bien se explica en el tutorial ya mencionado, creando una clase modelo que se encarga de crear y cargar las mallas, además de cargar las imágenes utilizando la librería *stb_image* y aplicarlas correctamente a las mallas en cuestión, también partiendo de *LearnOpenGL - Textures* [16].

Debido a problemas de compatibilidad con Vulkan, se cambió de localización las normales y las coordenadas de textura, haciendo que sean iguales tanto el Vulkan como en OpenGL facilitando su uso en los shaders. Esto dio lugar a varios problemas iniciales, debido principalmente a que la matriz de vista no se transponía, siendo el resultado algo como esto:

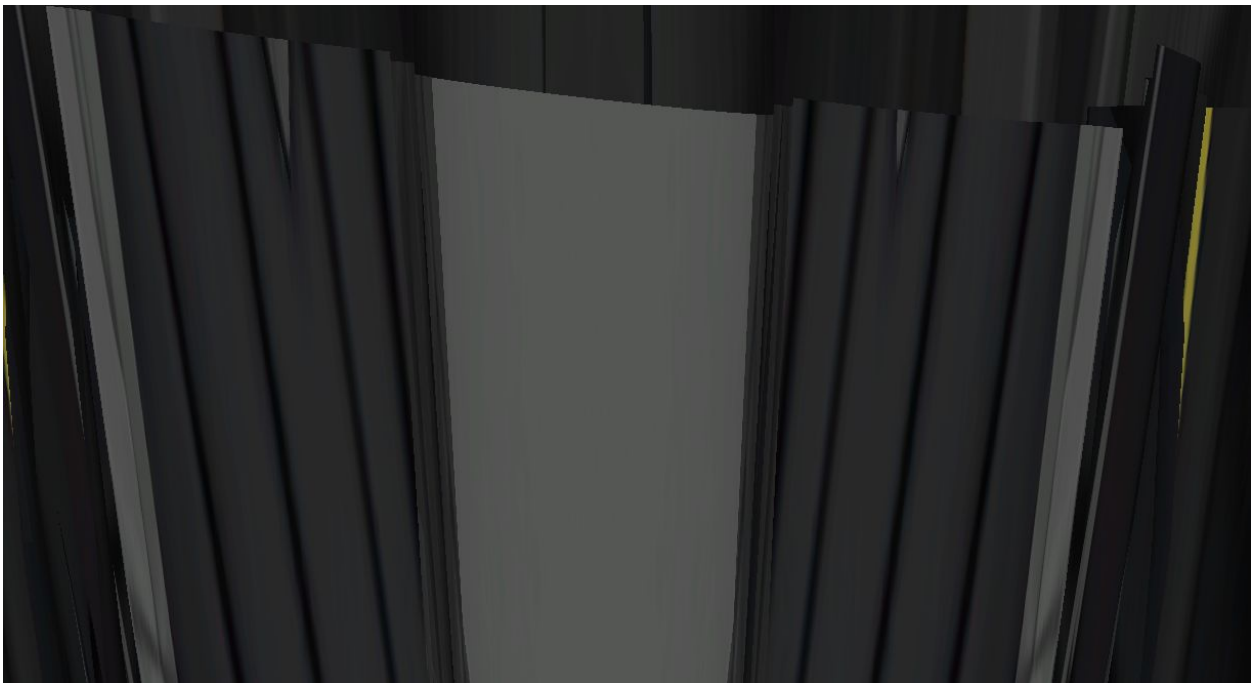

Figura 6.1-1: Visualización con errores de un modelo en OpenGL

Solucionando estos errores de visualización, surgidos a partir de la adaptación de los *shaders*, se consiguió por fin el resultado deseado mostrado a continuación:

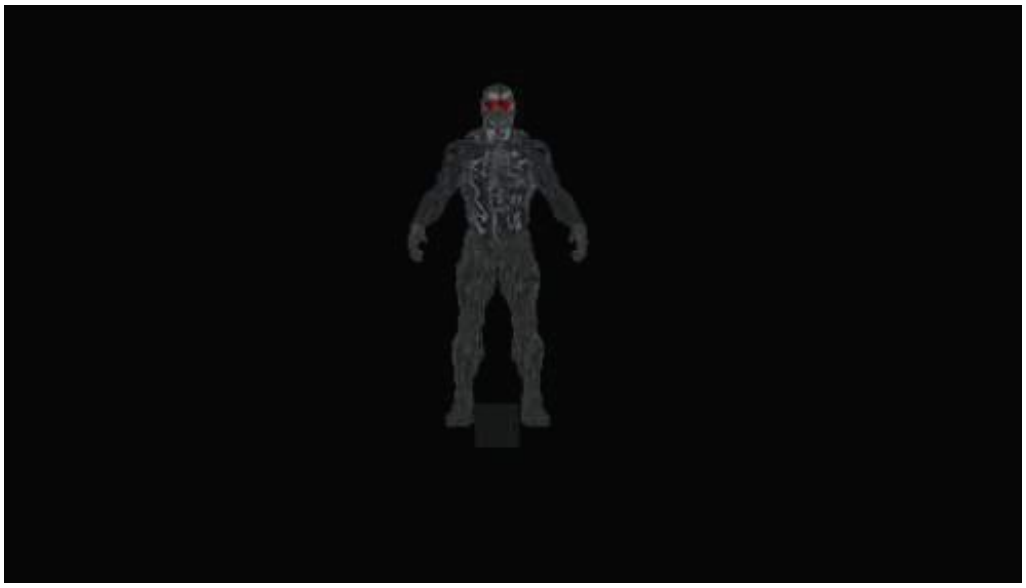

Figura 6.1-2: Visualización correcta de un modelo en OpenGL

El siguiente paso era mezclar el modelo junto con el fractal. Como viene explicado en la parte del tutorial *Depth testing*, primero se activó la opción de OpenGL glEnable(GL_DEPTH_TEST), pudiendo así modificar el *buffer* de profundidad en los *shaders* con total libertad.

Los valores del *buffer* de profundidad de OpenGL están comprendidos en el rango [-1, 1], aunque es rellenado por parte del *shader* del modelo con valores en el rango [0, 1], como se indica en el tutorial. Para ello, se utiliza la función de profundidad proporcionada por el mismo dividida entre *far plane* para adoptar dicho rango. Además, los valores que se usan para calcular la profundidad lineal, *gl_FragCoord.z*, están también comprendidos en el mismo intervalo [0, 1].

En el caso del fractal utilizamos las distancias obtenidas del algoritmo de *Ray Marching* divididas entre la máxima distancia que éstas pueden alcanzar (*far plane*), consiguiendo así de nuevo el intervalo [0, 1].

Función lineal de profundidad:

```
float LinealizarProfundidad(profundidad)
{
    profundidad = profundidad * 2.0 - 1.0;
    return (2.0 * planoCercano * planoLejano) / (planoLejano +
    planoCercano - profundidad  * (planoLejano - planoCercano ));
}
```

Como resultado final, se obtiene una imagen que combina correctamente el fractal y el modelo:

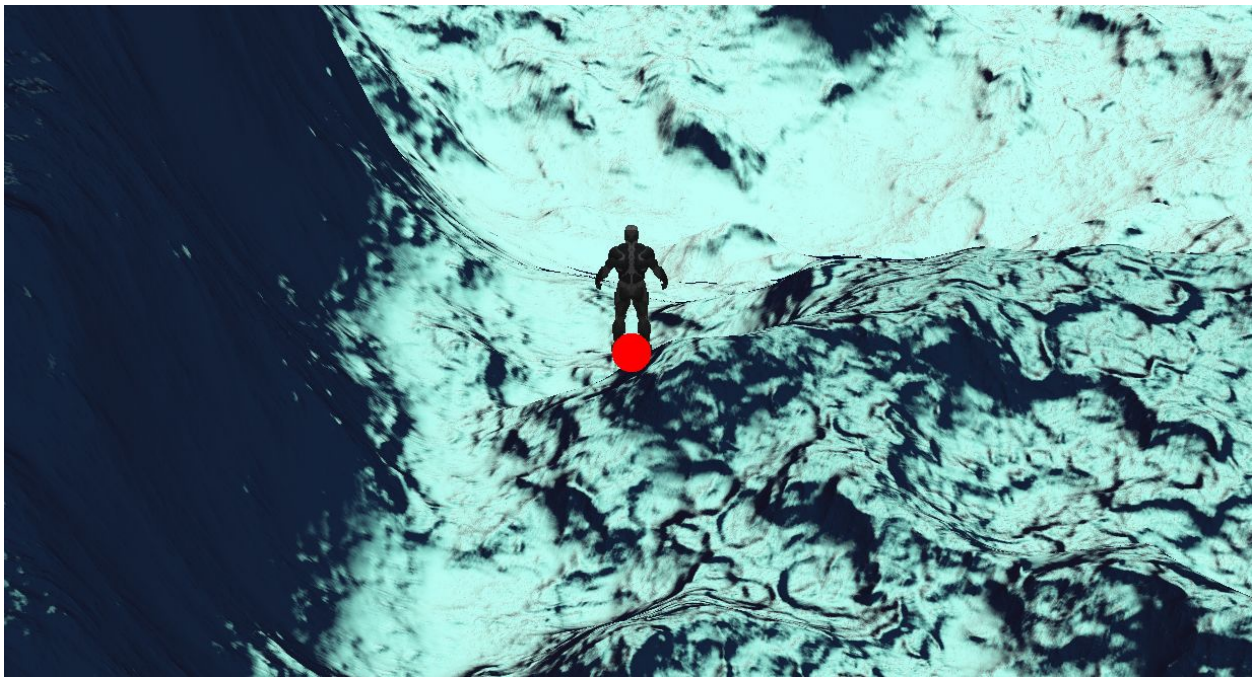

Figura 6.1-3: Modelo y fractal combinados en OpenGL correctamente

6.2. Modelo en Vulkan

Para conseguir visualizar el modelo de la misma forma utilizando la *API* de *Vulkan*, se siguieron las indicaciones descritas en los apartados *Texture mapping* [34], *Depth buffering* [35] y *Loading models* [36] de la documentación de Vulkan. Los pasos fueron los siguientes: visualización una imagen cargada de archivo, superposición de dos texturas con distintas profundidades y aplicación de los dos pasos anteriores para cargar un modelo de archivo.

En el primer paso, se hizo uso de la librería *stb_image* para la carga de imágenes y se utilizaron *Samplers*, que permiten utilizar imágenes y acceder a sus colores desde un *shader* cuando éstas son usadas como texturas. Sumado a esto, los vértices utilizados por los *vertex shaders* fueron ampliados para incluir coordenadas de textura, además de su anterior posición.

Seguidamente (segundo paso), se modificó el *Render pass* para aceptar *depth buffering* y el *Frame buffer* para añadir la *depth image* a la lista de elementos adjuntos (antes sólo contaba con las *swapchain images*). Por último, se habilitó el *depth testing* en la *graphicsPipeline* y se logró así visualizar dos texturas, una delante de otra, de forma que la más cercana se viera por encima de la lejana.

Para el tercer paso, se utilizó la librería *tinyobjloader*, que permite cargar modelos con una textura asociada. Los vértices del modelo son leídos utilizando dicha librería y almacenados en los *buffer* de vértices que usan los *vertex shader*, junto con sus índices, normales y coordenadas de textura.

A la hora de intentar visualizar modelos en formato .obj en Vulkan, se descubrió que las coordenadas de textura se leían de manera inversa en el eje *y*, teniendo que invertirlas al utilizarlas en el *shader* del modelo obteniendo así el correcto funcionamiento de las texturas.

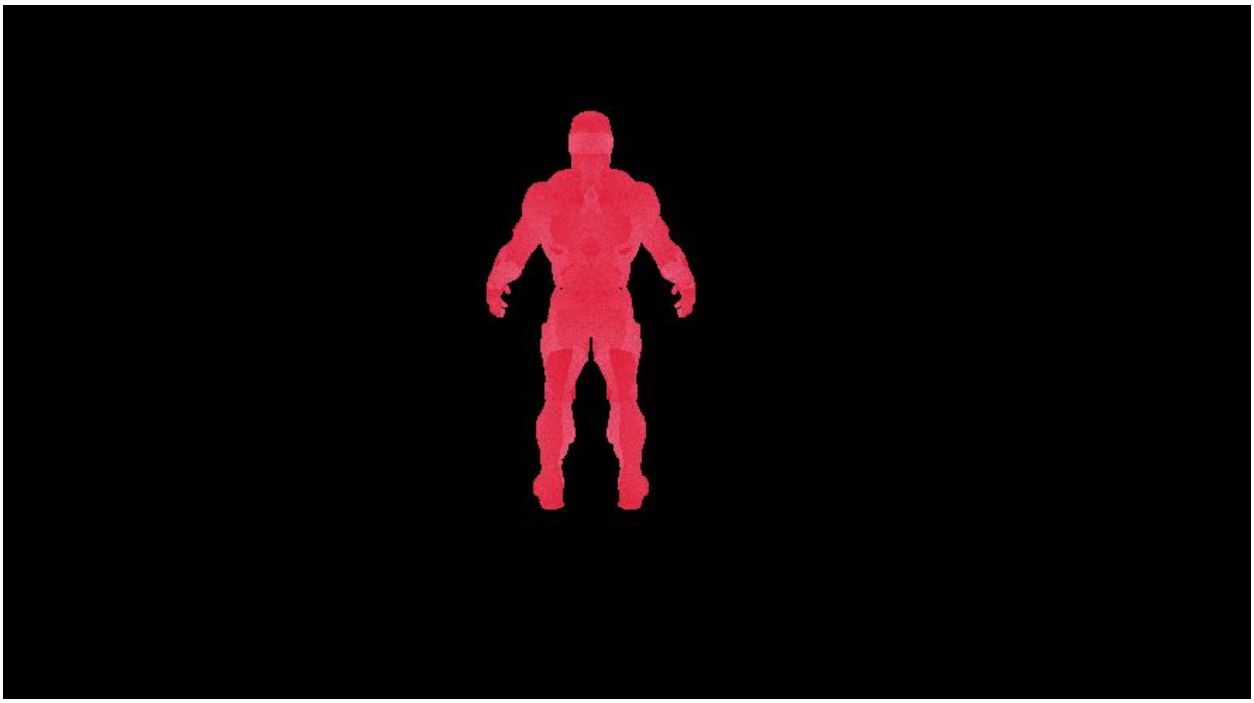

Figura 6.2-1: Visualización de un modelo en Vulkan

Conseguido esto, se pudo visualizar de forma correcta un modelo en Vulkan. Sin embargo, la combinación de los *shaders* encargados de pintar el modelo y los encargados de pintar el fractal no fue la esperada, y sólo podía verse el resultado del que primero que se añadiese al gestor de Vulkan.

Para conseguir la combinación deseada, se creó una nueva *graphics Pipeline*, ya que antes se utilizaba una *graphics pipeline* compartida, y se hizo uso de dos *subpasses* distintos, atendiendo al artículo *Vulkan input attachments and sub passes*, Sascha Willems [52]. Los *subpasses* son utilizados en el *render pass* para hacer combinaciones de imágenes o *frame buffers*, antes de generar el *frame buffer* final que será volcado en pantalla. De esta forma, uno de los *subpasses* escribe en el *frame buffer* la información del fractal, y el otro lee el anterior *subpass* y escribe sobre él la información del modelo si fuera necesario; es decir, sólo si ese píxel en concreto debe sobreescribir la información existente, atendiendo al *depth buffer* (si está más cerca de la cámara que el píxel escrito anteriormente en esa posición). Esa combinación de los *subpasses* es leída y mostrada finalmente por el *render pass* en el *frame buffer* final. En este

punto, se obtuvo una imagen en la que ya se podían visualizar el modelo y el fractal conjuntamente.

Sin embargo, el modelo no se mostraba de la forma correcta, ya que parecían faltarle triángulos a su malla, y en varias partes aparecían triángulos que no deberían (véase las líneas que cruzan de extremidad a extremidad en el modelo siguiente).

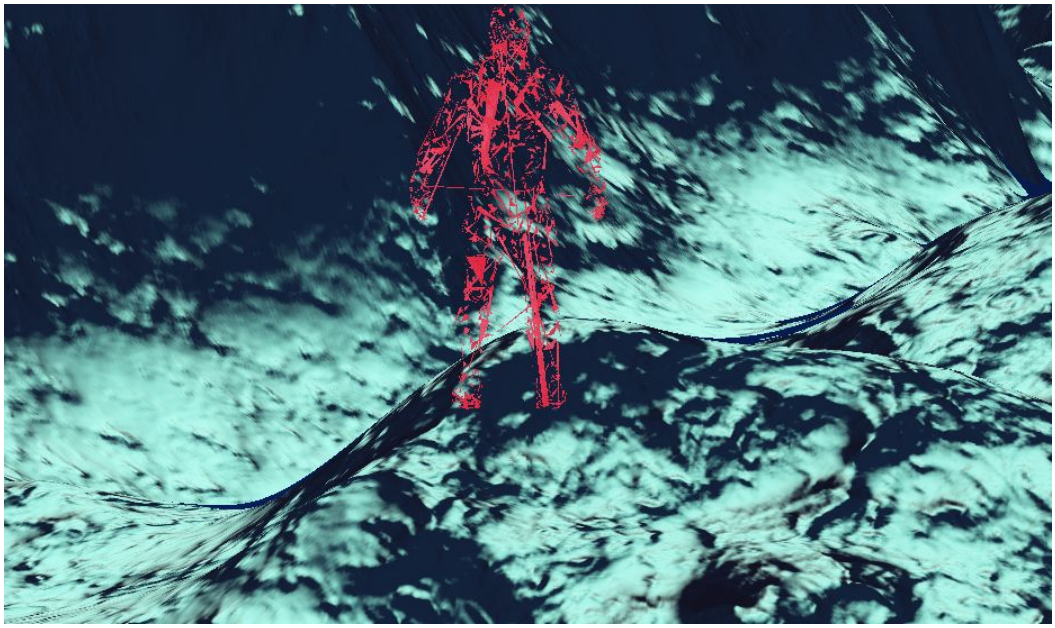

Figura 6.2-2: Modelo y fractal combinados en Vulkan, con error de triángulos en el modelo

Para solucionarlo, se separaron los vértices e índices utilizados por los *vertex shaders*, uno para el *vertex shader* del modelo y otro para el del fractal, ya que hasta ahora se tenían unidos en los mismos vectores. De esta manera, se creó un *vertex buffer* con los vértices del rectángulo para el *vertex shader* del fractal (los que se tenían inicialmente), y otro *vertex buffer* para los nuevos vértices correspondientes al modelo (lo mismo se hizo para los índices del fractal y los índices del modelo). Estos vértices e índices se enlazaron en su *subpass* correspondiente, quedando cada *shader* con su propia información en vez de una compartida, lo que finalmente hizo que el modelo se visualizase de la forma correcta.

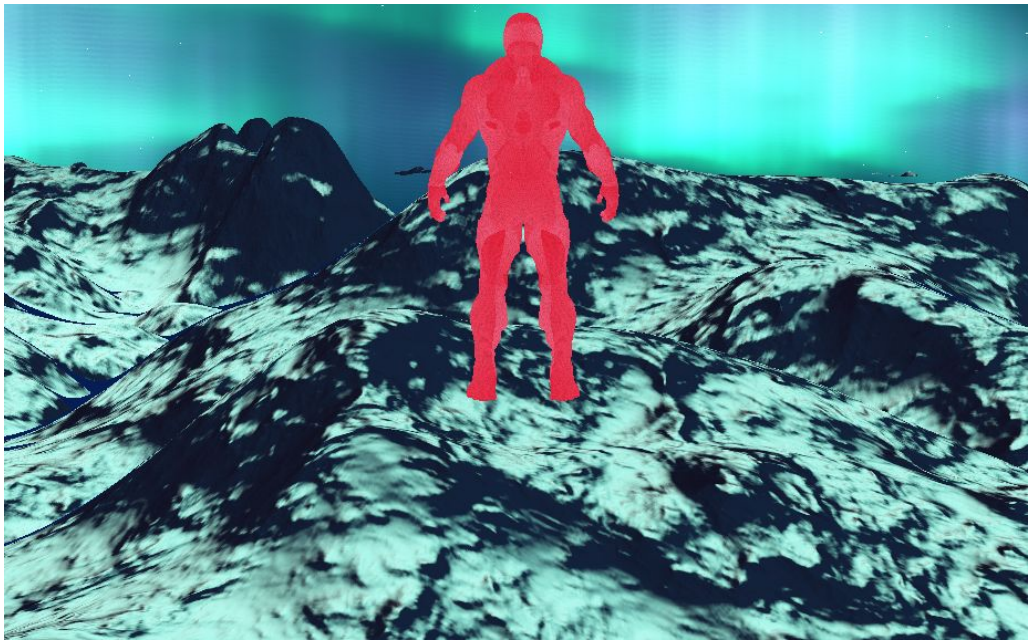

Figura 6.2-3: Modelo y fractal combinados en Vulkan, solucionado error de triángulos en el modelo

El último problema que se afrontó en la implementación en *Vulkan* estuvo relacionado con la forma de escribir en el *depth buffer* desde el *fragment shader* del modelo. Se descubrió que en *Vulkan* la variable *gl_FragCoord.z* viene dada en el intervalo [-1, 1], mientras que en *OpenGL* su intervalo es de [0, 1]. Por tanto, al utilizar esta misma variable sin adaptar su valor al rango [0, 1] en ambas *APIs* para rellenar el *depth buffer*, se obtenían resultados distintos, y en *Vulkan* el modelo no se ocultaba tras las protuberancias del terreno de la forma esperada.

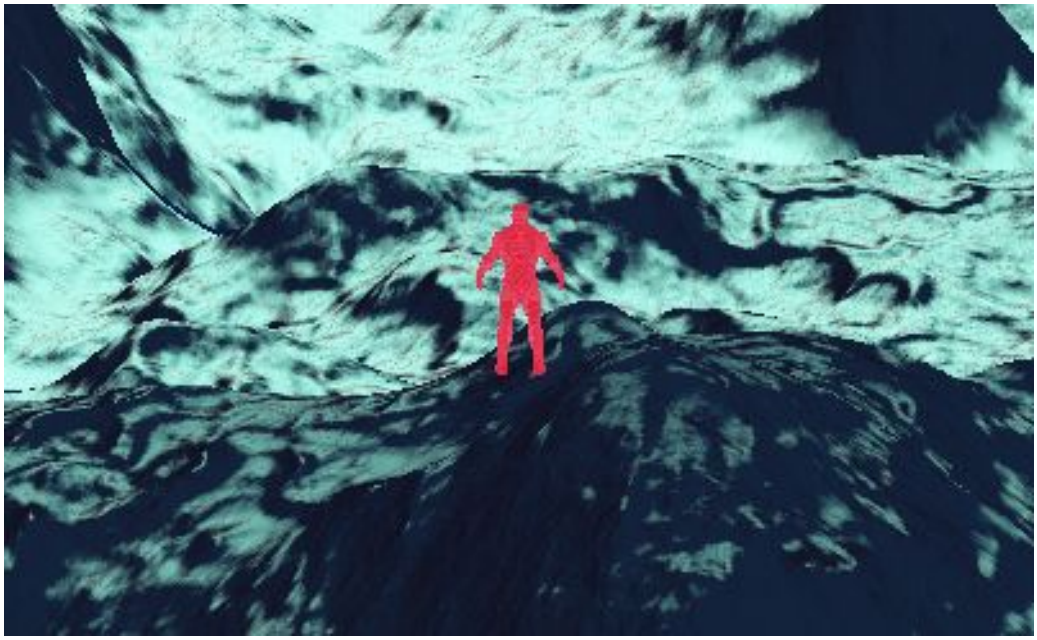

Figura 6.2-4: Modelo y fractal combinados en Vulkan, con error en la profundidad del modelo

La solución consistió en utilizar en cada *shader* una variable de tipo *#define* indicando la *API* que se estuviese usando en cada momento, *OpenGL* o *Vulkan*. Estas variables (#define VULKAN y #define OPENGL) se definen en el momento de leer los *shaders*, y dependiendo de la configuración que estemos ejecutando la aplicación se escribe en el *shader* a compilar una u otra. En el *shader*, a la hora de escribir en el *depth buffer*, se hizo uso de código condicional, tratando la variable *gl_FragCoord.z* en el caso de encontrarse en *Vulkan*, para igualar su intervalo al de *OpenGL* y así obtener el mismo resultado en ambas *APIs*.

```
#ifdef VULKAN
   gl_FragDepth = LinearizeDepth((gl_FragCoord.z + 1)/2)/far;
// tratamiento de la variable para obtener intervalo [0, 1]
#else
   gl_FragDepth = LinearizeDepth(gl_FragCoord.z)/far;
// sin tratamiento de la variable, intervalo [0, 1] por defecto
#endif
```

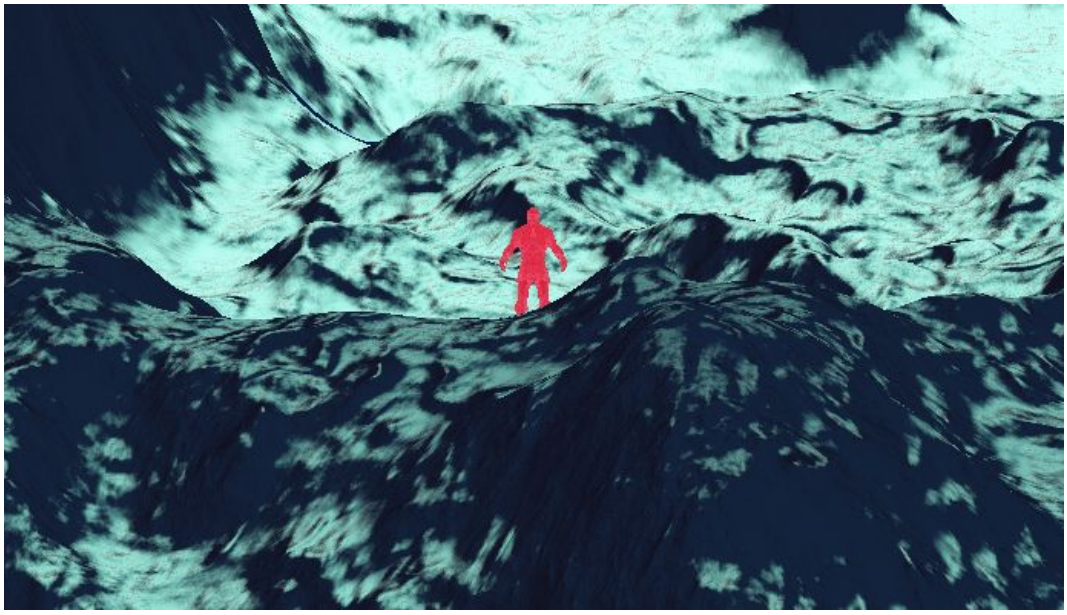

Figura 6.2-5: Modelo y fractal combinados en Vulkan correctamente

Tras conseguir visualizar modelos correctamente en ambas plataformas, el modelo de pruebas fue sustituido por un modelo del personaje Pacman con una sola textura, que gira sobre sí mismo al avanzar hacia delante, atrás o hacia los lados, dando la sensación de que el jugador pueda rodar por el suelo.

7. Estructura de la aplicación

En este apartado se describe el resultado final de la aplicación, incluyendo la organización de carpetas que se ha seguido, qué estructura de clases de C++ y librerías se han utilizado y cómo se ha gestionado el funcionamiento de ambas *APIs* (OpenGL y Vulkan) como parte del mismo proyecto.

Dicho trabajo ha sido desarrollado en el lenguaje C++ con la herramienta Visual Studio 2019, y todas las librerías incorporadas al proyecto son multiplataforma.

Como ya se ha mencionado, como repositorio de trabajo para el proyecto se ha utilizado GitHub. El código de la aplicación puede encontrarse en el siguiente enlace: https://github.com/DiegoBV/TFG-Repo. Para compilar su contenido es necesario descargar sus dependencias, siguiendo las instrucciones del archivo README.md, situado en el directorio raíz del repositorio.

7.1. Estructura de carpetas

En el directorio raíz del trabajo se encuentran las siguientes carpetas:

- Dependencies: contiene todas las dependencias externas necesarias para el correcto funcionamiento del proyecto. En concreto, contiene los archivos .lib y el código fuente de la librería *Assimp* para la carga de modelos en OpenGL, *tinyobjloader* para la carga de modelos en Vulkan, *stb_image* para la carga de texturas, GLAD para gestionar los punteros a funciones de OpenGL en tiempo de ejecución, GLFW para la gestión de la ventana, GLM para cálculos matemáticos y la librería del propio Vulkan. Todas estas librerías fueron descargadas de los sitios oficiales (LunarG) y compiladas posteriormente para Windows x64 utilizando el programa CMake.

- exes: contiene una carpeta Assets con todos los *assets* utilizados por el trabajo, en este caso únicamente el modelo y la textura del jugador, y una carpeta por cada *API*, OpenGL

y Vulkan. En la carpeta de OpenGL se generan los ejecutables (ya sean Debug o Release, diferenciándose éstos por el nombre) generados al compilar el proyecto con esa *API*, junto con los archivos .dll que necesite OpenGL. De la misma forma, la carpeta de Vulkan contiene los ejecutables generados al compilar el proyecto con Vulkan y los archivos .dll que éste necesite.

- Inter: contiene los archivos generados de la compilación intermedia, separados en carpetas según tipo de *API* y modo de compilación.

- Projects: contiene los archivos de todos los proyectos de la solución. En un principio se pensó tener dos proyectos en forma de librerías estáticas, uno para cada *API*, junto con el proyecto principal, que enlazaría el proyecto correspondiente y lanzaría la aplicación, aunque posteriormente se decidió tener ambas en el proyecto principal Main.

- Shaders: contiene los archivos de todos los *shaders* utilizados por la aplicación. Entre éstos se cuenta con un *fragment shader*, un *compute shader* y un archivo con funciones específicas por cada fractal, un *vertex shader* y un *fragment shader* para el modelo, un *vertex shader* compartido por todos los fractales, un archivo de funciones comunes para los fractales de tipo terreno, un archivo de funciones comunes para los fractales de tipo "planeta", un archivo con geometrías básicas y sus funciones de distancia y un archivo con funciones de iluminación.

- src: contiene el código fuente de la aplicación.

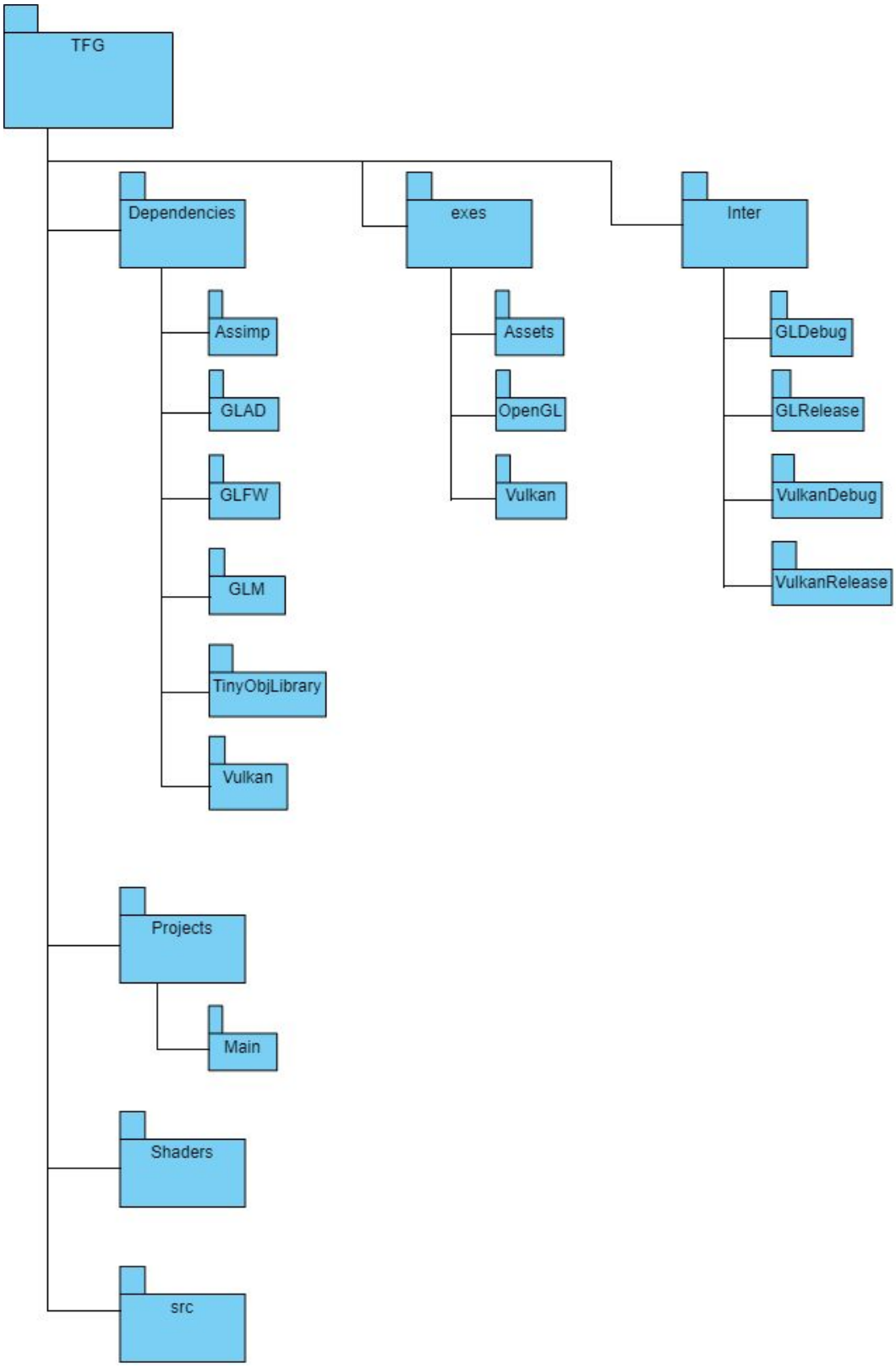

Figura 7.1-1: Diagrama de carpetas utilizadas en el trabajo

7.2. Estructura de clases

Se ha dividido el código fuente en varios grupos o filtros: cámara y ventana, objetos controlables por el usuario, utilidades genéricas accesibles desde el resto del código, implementación en OpenGL, implementación en Vulkan y archivos que sirven de intermediarios para decidir cuál de estas dos *APIs* se usa para compilar.

- Cámara/Ventana: contiene las clases Camera y Window, que gestionan las funcionalidades de éstas.

- Objetos controlables: contiene la clase padre de todos los objetos controlables que habrá en el juego, PlayableObject. Esta clase concede la funcionalidad de mover al objeto en cuestión y actualizar las fuerzas que intervienen en su movimiento, y deja como método abstracto a redefinir por las clases hijas el responsable de hallar los puntos de colisión del objeto, única característica que diferenciará unos objetos controlables de otros. También se cuenta con una clase hija llamada PlayableSphere que, como su nombre indica, redefine el método de los puntos de colisión para generar una *hitbox* esférica mediante la esfera de Fibonacci, ya mencionada con anterioridad en el apartado Físicas. En el proyecto se tiene únicamente un objeto controlable PlayableSphere para el jugador, de cuyo renderizado se encargan los *shaders* de modelo.

- Utils: contiene clases auxiliares disponibles desde cualquier parte del código, en su mayoría patrones *singleton* o clases estáticas. Entre ellas se encuentra FileHandler, encargada de abrir, leer, escribir y cerrar archivos. También se tiene una clase ShaderInclude, que se encarga de leer un shader recursivamente hasta realizar todas sus inclusiones apoyándose de la clase FileHandler (explicado con más detalle en el subapartado Shaders de Aplicación en OpenGL), y de devolver su código completo en forma de *string*. Siguiendo el patrón *singleton*, se encuentra una clase TimeManager que se apoya en otra llamada Timer, y que se encarga de devolver el tiempo desde el inicio de la aplicación, el tiempo entre *frames*, etc. Por último, esta sección contiene además un fichero ShaderUtils, con contenido compartido por todos los shaders, donde están

establecidos el *UniformBufferObject*, el *StorageBufferObject* y algunas variables globales.

- OpenGL: contiene todas las clases necesarias para el desarrollo de la aplicación en OpenGL. Estas clases son GLShader, como interfaz básica de todos los tipos de *shader* en OpenGL, y sus clases hijas, GLComputeShader, con la funcionalidad de los *compute shaders* para OpenGL, y GLRenderShader, con la funcionalidad de los *shaders* estándar para OpenGL. Además, se cuenta con las clases Mesh, como unidad mínima de un modelo, y GLModel, que gestiona la carga y la visualización de modelos en OpenGL mediante listas de objetos Mesh y texturas. Por último, se tiene la clase GLManager, encargada de gestionar la aplicación y ejecutar los *shaders*, como se indica en el apartado Aplicación en OpenGL.

- Vulkan: contiene toda las las clases necesarias para el desarrollo de la aplicación en Vulkan. Estas clases son VulkanShader, como interfaz básica de todos los tipos de *shader* en Vulkan, y sus clases hijas, VulkanComputeShader, con la funcionalidad de los *compute shaders* para Vulkan, y VulkanRenderShader, con la funcionalidad de los *shaders* estándar para Vulkan. Por último, se tiene la clase VulkanManager, encargada de gestionar la aplicación y ejecutar los *shaders*, como se indica en el apartado Aplicación en Vulkan. Esta clase gestiona también el cargado del modelo.

- Inter: contiene dos clases, Shader y ApplicationManager, que deciden qué clases, si las de OpenGL o las de Vulkan, deben ser utilizadas en el proyecto, dependiendo de la *API* elegida. Esta funcionalidad se explica con más detalle en el siguiente apartado.

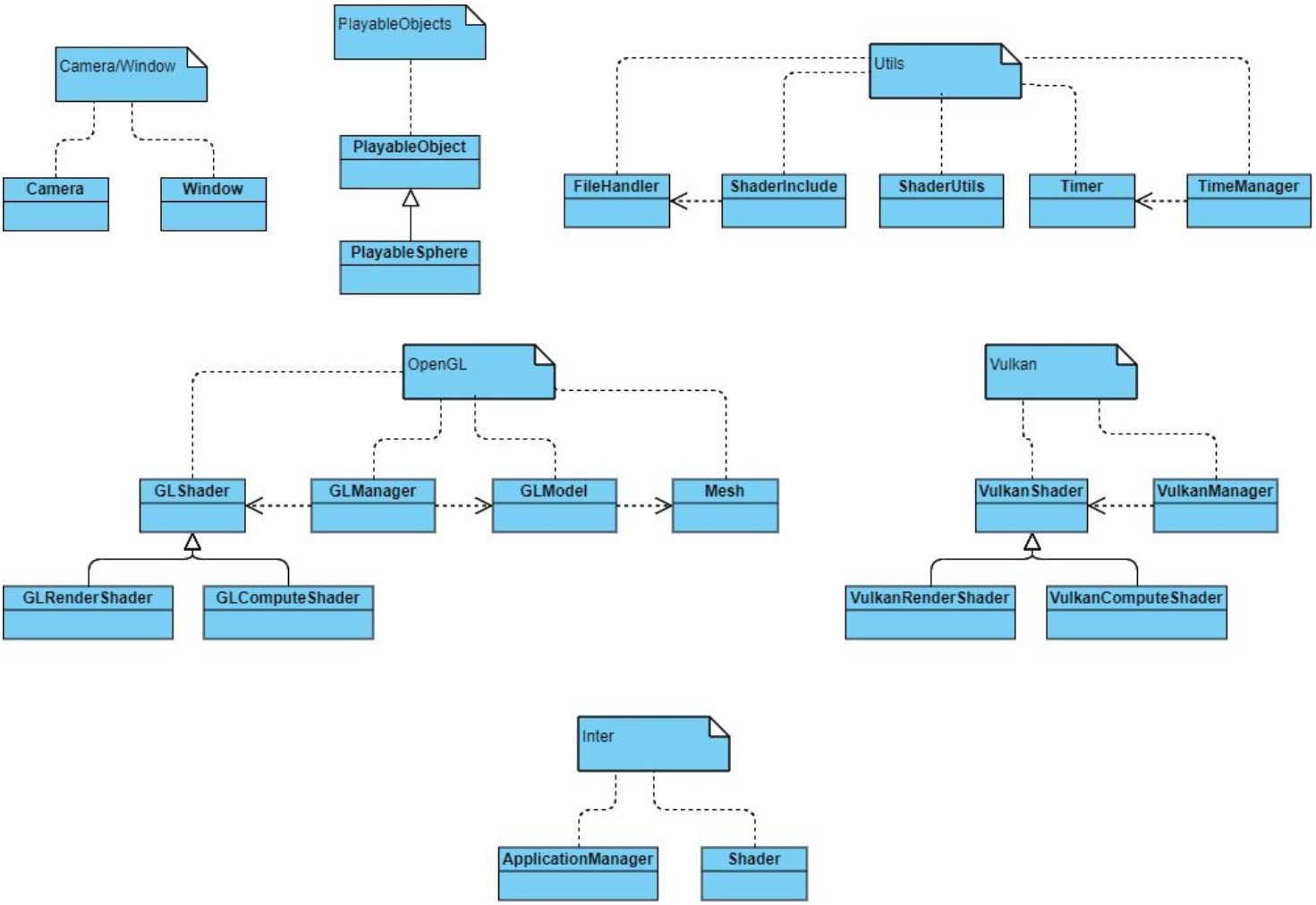

Figura 7.2-1: Diagrama de clases utilizadas en el trabajo

7.3. Funcionamiento en ambas *APIs*

Como se ha mencionado anteriormente, se desarrolló el trabajo con un único proyecto de Visual Studio, en el que coexisten OpenGL y Vulkan. Para ello, se crearon cuatro configuraciones distintas para Windows x64, GLDebug, GLRelease, VulkanDebug y VulkanRelease, permitiendo así la compilación con cada una de las *APIs* en modo Debug o en modo Release. A cada una de estas configuraciones se le dio un nombre para el preprocesador y así poder implementar código condicional sobre ellas.

De esta forma, el mecanismo que se siguió para compilar con una u otra *API* dependiendo de la configuración seleccionada se basó en el las directivas condicionales del precompilador. Cada archivo .cpp perteneciente a la aplicación de alguna de las *APIs* introduce todo su código entre directivas condicionales, de manera que sólo son compilados si la configuración de su *API* correspondiente, ya sea en modo *Debug* o modo *Release*, está seleccionada.

En el caso de OpenGL:

```
#if defined (GL_DEBUG) || defined (GL_RELEASE)
```

En el caso de Vulkan:

```
#if defined(VULKAN_DEBUG) || defined(VULKAN_RELEASE)
```

Sumado a esto, los archivos que antes denominábamos Inter, también cuentan con directivas condicionales, definiendo un objeto como su implementación en Vulkan o su implementación en OpenGL, dependiendo de la *API* seleccionada. Estos archivos sirven de intermediario entre la implementación de las dos *APIs* y el código que lanza la aplicación, que sólo se comunicará con los archivos Inter y utilizará los objetos devueltos por éstos, abstrayéndose de si se corresponde a la implementación de una *API* u otra. El funcionamiento de estos archivos es el siguiente:

```
#if defined(GL_DEBUG) || defined(GL_RELEASE)
#include "ClaseGL.h"
using Clase = ClaseGL;
#elif defined(VULKAN_DEBUG) || defined(VULKAN_RELEASE)
```

```
#include "ClaseVulkan.h"
using Clase = ClaseVulkan;
```

Atendiendo al pseudocódigo anterior, en las clases que se abstraigan de esta elección de *API*, simplemente se incluirá este fichero y se hará uso de la clase "Clase", cuyo código corresponderá con la *API* elegida.

Hay dos ficheros Inter, uno que elige el gestor de la aplicación a utilizar, GLManager o VulkanManager, y que establece su nombre como ApplicationManager, y otro que elige los shaders, entre GLRenderShader o VulkanRenderShader, al que llama RenderShader, y GLComputeShader o VulkanComputeShader, al que llama ComputeShader.

El resultado final es que desde main.cpp se incluyen los archivos Inter y se hace uso de las clases ApplicationManager, RenderShader y ComputeShader, usándolas indistintamente y sin preocuparse de qué *API* ha sido compilada y se está utilizando.

Cabe destacar que para que este mecanismo funcione correctamente, las clases implementadas en OpenGL deben tener los mismos métodos públicos que las implementadas en Vulkan, ya que éstos serán llamados desde main.cpp indistintamente de la *API*, como se ha explicado anteriormente.

Por último, para poder utilizar los mismos archivos de *shaders* en ambas *APIs*, se hicieron algunos ajustes. En un principio, estos archivos se hicieron pensando en *shaders* de OpenGL, por lo que su extensión era .vert para los *vertex shaders* y .frag para los *fragment shaders*, y se utilizaba la versión 430 de glsl, recibiendo las variables uniformes con la palabra reservada uniform. Al introducir los *shaders* en Vulkan, se produjeron algunos cambios para poder usar los mismos archivos en ambas *APIs* y no tenerlos por duplicado para cada una de ellas. Estos cambios consistieron en usar la versión 450 de glsl y cambiar ligeramente la sintaxis, ya que Vulkan pedía una bastante específica y OpenGL sin embargo se adaptaba correctamente a ligeros cambios. Las variables uniformes fueron sustituidas por structs *UBO* (Uniform Buffer Object), de nuevo para adaptarse a los requerimientos de Vulkan, y las extensiones de los

archivos cambiaron a .c. Además se ajustaron los parámetros de algunas funciones para generalizar su uso en los distintos tipos de *shader*.

Vulkan además necesita interpretar estos *shaders* en el lenguaje SPIR-V por lo que, tras leer todo el código de éstos, en caso de encontrarse en la configuración Vulkan (haciendo uso de código condicional), este código es compilado a SPIR-V a través de un archivo .bat, en vez de ser utilizado directamente como en el caso de OpenGL.

Gracias a esta estructura y ajustes, la aplicación es capaz de ejecutarse usando cualquiera de las dos *APIs*, tan sólo cambiando la configuración en la ventana de Visual Studio y teniendo una única versión de los *shaders*.

8. Conclusiones

Como se ha descrito, la aplicación ofrece la posibilidad de compilarse desde ambas *APIs* gráficas simplemente cambiando la plataforma desde el editor Visual Studio. Para ello, se ha hecho uso de conocimientos adquiridos previamente para realizar las interfaces y clases necesarias que cumplan con los objetivos impuestos. Sin embargo, se han encontrado también dificultades importantes al separar funcionalidad específica de cada *API*, en la gestión de clases y otros problemas mencionados anteriormente, tales como la diferencia en el *buffer* de profundidad o las operaciones del modelo.

Los resultados de la investigación realizada sobre ambas *APIs* demuestran que el trabajo realizado en Vulkan es más complejo. Esto ocurre debido a la libertad que ofrece Vulkan respecto a OpenGL, ya que permite configurar una gran cantidad de parámetros, entre los que se encuentran, por ejemplo, qué tarjeta gráfica se va a usar, la especialización o tarea de cada GPU según sus características o el control de memoria sobre diferentes partes del *hardware*. Se puede asumir que, debido a dicha libertad, Vulkan va a poder establecer un nuevo concepto de trabajo en las empresas dedicadas al desarrollo de aplicaciones gráficas.

Una de las partes más importantes del proyecto han sido el algoritmo *Ray Marching*, las fórmulas de distancia (SDF) y su implementación en los *shaders*. Con ellos, se ha conseguido renderizar todos los fractales descritos con anterioridad. Además, gracias al descubrimiento de los *compute shaders* se ha podido incluir un mecanismo sencillo de físicas utilizando las SDF. Los cálculos relacionados para la colisión entre el modelo controlado por el jugador y los terrenos se realizan enteramente en los *compute shaders* ejecutados en la tarjeta gráfica, comunicándose con la CPU con el resultado de dichos cálculos, permitiendo así el comportamiento correcto de las físicas.

Finalmente, se ha introducido otro tipo de renderizado relacionado con la inclusión de un modelo en la aplicación. Este renderizado es uno de los más conocidos y habituales, haciendo uso de los vértices que componen las mallas del modelo y el *testing* de profundidad de cada uno de los fragmentos o *pixels*. Por tanto, se ha logrado mezclar dos tipos de renderizado diferentes

de manera correcta respetando la profundidad específica de cada momento de ejecución (gracias al *test* de profundidad).

Con todo ello, se puede concluir que los objetivos pensados en un principio se han cumplido exitosamente, aplicando los conocimientos adquiridos durante toda la etapa académica. Se ha profundizado en los conceptos que rodean a las diferentes interfaces de programación utilizadas, así como en los diferentes métodos utilizados, desde las fórmulas de distancia hasta los *compute shaders*, para lograr el resultado final, adquiriendo así importantes competencias para el desarrollo de videojuegos.

8.1. Trabajo futuro

Teniendo en cuenta las metas alcanzadas durante el desarrollo del proyecto, se tienen en mente una serie de pasos claros que se deberían seguir en el futuro: primero, mejorar la calidad del código, especialmente aquel relacionado con Vulkan. Esto se debe a la gran cantidad de líneas que se necesitan para poder ejecutar Vulkan y requiere una atención especial. Segundo, la inclusión de nuevos terrenos y la mejora en la implementación de físicas a los mismos es otro paso muy importante a tener en cuenta para ampliar la aplicación. Finalmente, perfeccionar la interacción con el usuario, es decir, mejorar y añadir nueva interfaz de control para proporcionar libertad al usuario del programa, así como pulir los controles y el movimiento sobre los diferentes fractales disponibles. Con ello, se proporciona mayor sensación de acabado y se logra aportar una riqueza considerable al proyecto, lo cual es el principal objetivo del trabajo futuro.

Bibliografía

[1] Artacho, A. (2014): *Fractal en 3D… Un viaje por el interior de un Mandelbox* [En línea]. Disponible en: https://matematicascercanas.com/2014/05/28/fractal-en-3d-mandelbox-3d/ [Consulta: 25 de noviembre 2019].

[2] Befall (2012): *Evenly distributing n points on a sphere - Stack Overflow* [En línea]. Disponible en:

https://stackoverflow.com/questions/9600801/evenly-distributing-n-points-on-a-sphere
[Consulta: 20 de marzo 2020].

[3] Brunet, J. (n.d.): *3D fractal trip - Far away…* [En línea]. Disponible en: https://imaginary.org/film/3d-fractal-trip-far-away [Consulta: 25 de noviembre 2019].

[4] CadNav (2020): *7 Free 3D Fractal Generation Software - CadNav* [En línea]. Disponible en: http://www.cadnav.com/software/view-46041.html [Consulta: 23 de noviembre 2019].

[5] Christopoulos P. (2017): "Vulkan's coordinate system". *ANKI 3D ENGINE DEV BLOG*, 19 de julio [En línea]. Disponible en: http://anki3d.org/vulkan-coordinate-system/ [Consulta: 23 de febrero de 2020].

[6] CodeParade (2018): *Marble Marcher - A game of fractals based on physics* [En línea]. Disponible en: https://www.youtube.com/watch?v=9U0XVdvQwAI [Consulta 22 septiembre de 2019].

[7] darkeclipz (2019a): *Shader - Shadertoy BETA - Mandelbox rotating* [En línea]. Disponible en: https://www.shadertoy.com/view/wldGR4 [Consulta: 15 de febrero de 2020].

[8] darkeclipz (2019b): *Shader - Shadertoy BETA - Raymarch Mandelbox 2* [En línea]. Disponible en: https://www.shadertoy.com/view/tdXSRn [Consulta: 15 de febrero de 2020].

[9] De Vries, J. (2020a): *Learn OpenGL, extensive tutorial resource for learning Modern OpenGL* [En línea]. Disponible en: https://learnopengl.com/ [Consulta: 25 de octubre de 2019].

[10] De Vries, J. (2020b): *LearnOpenGL - Introduction* [En línea]. Disponible en: https://learnopengl.com/Introduction [Consulta: 25 de octubre de 2019].

[11] De Vries, J. (2020c): *LearnOpenGL - OpenGL* [En línea]. Disponible en: https://learnopengl.com/Getting-started/OpenGL [Consulta: 25 de octubre de 2019].

[12] De Vries, J. (2020d): *LearnOpenGL - Creating a window* [En línea]. Disponible en: https://learnopengl.com/Getting-started/Creating-a-window [Consulta: 25 de octubre de 2019].

[13] De Vries, J. (2020e): *LearnOpenGL - Hello Window* [En línea]. Disponible en: https://learnopengl.com/Getting-started/Hello-Window [Consulta: 25 de octubre de 2019].

[14] De Vries, J. (2020f): *LearnOpenGL - Hello Triangle* [En línea]. Disponible en: https://learnopengl.com/Getting-started/Hello-Triangle [Consulta: 25 de octubre de 2019].

[15] De Vries, J. (2020g): *LearnOpenGL - Shaders* [En línea]. Disponible en: https://learnopengl.com/Getting-started/Shaders [Consulta: 25 de octubre de 2019].

[16] De Vries, J. (2020h): *LearnOpenGL - Textures* [En línea]. Disponible en: https://learnopengl.com/Getting-started/Textures [Consulta: 5 de abril de 2019].

[17] De Vries, J. (2020i): *LearnOpenGL - Camera* [En línea]. Disponible en: https://learnopengl.com/Getting-started/Camera [Consulta: 6 de febrero de 2020].

[18] De Vries, J. (2020j): *LearnOpenGL - Assimp* [En línea]. Disponible en: https://learnopengl.com/Model-Loading/Assimp [Consulta: 5 de abril de 2020].

[19] De Vries, J. (2020k): *LearnOpenGL - Mesh* [En línea]. Disponible en: https://learnopengl.com/Model-Loading/Mesh [Consulta: 5 de abril de 2020].

[20] De Vries, J. (2020l): *LearnOpenGL - Model* [En línea]. Disponible en:

https://learnopengl.com/Model-Loading/Model [Consulta: 5 de abril de 2020].

[21] De Vries, J. (2020m): *LearnOpenGL - Depth testing* [En línea]. Disponible en: https://learnopengl.com/Advanced-OpenGL/Depth-testing [Consulta: 8 de abril de 2020].

[22] dr2 (2019): *Shader - Shadertoy BETA - Mandelbox Tunnel* [En línea]. Disponible en: https://www.shadertoy.com/view/MlfSWX [Consulta: 17 de febrero de 2020].

[23] Ebert, D. S., Kenton, F., Peachey, D., Perlin, K., Worley, S. (1994): "Procedural fractal terrains", en D. S. Ebert (ed.), *Texturing and modeling: a procedural approach*, Tercera Edición, Estados Unidos, Editorial Morgan Kaufmann Publishers, pp. 489-509.

[24] Ebert, D. S., Kenton, F., Peachey, D., Perlin, K., Worley, S. (1994): "Mojoworld", en D. S. Ebert (ed.), *Texturing and modeling: a procedural approach*, Edición, Lugar de publicación, Editorial, pp. 565-566.

[25] frankenburgh (2013): *Shader - Shadertoy BETA - Oceanic* [En línea]. Disponible en: https://www.shadertoy.com/view/4sXGRM [Consulta: 7 de mayo de 2020].

[26] Gijs (2019): *Shader - Shadertoy BETA - Fractal: Mandelbox* [En línea]. Disponible en: https://www.shadertoy.com/view/3dBXzd [Consulta: 17 de febrero de 2020].

[27] Hvidtfeldt M. (2011): "Distance Estimated 3D Fractals (VI): The Mandelbox". *Distance Estimated 3D Fractals (VI): The Mandelbox | Syntopia,* 11 de noviembre [En línea]. Disponible en: http://blog.hvidtfeldts.net/index.php/2011/11/distance-estimated-3d-fractals-vi-the-mandelbox/ [Consulta: 16 de febrero de 2020].

[28] JosLeys (2020): *A Mandelbox distance estimate formula* [En línea]. Disponible en: http://www.fractalforums.com/3d-fractal-generation/a-mandelbox-distance-estimate-formula/ [Consulta: 15 de febrero de 2020].

[29] Khronos (2020): *VkFrontFace(3).* [En línea]. Disponible en:

https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/VkFrontFace.html [Consulta: 26 de febrero 2019].

[30] Lapinski, P. L. (2017): *Vulkan Cookbook*, Primera edición, n.d., Editorial Packt.

[31] LunarG (2018a): *Drawing a triangle - Vulkan Tutorial.* [En línea]. Disponible en: https://vulkan-tutorial.com/Drawing_a_triangle/Setup/Base_code [Consulta: 18 de febrero de 2020].

[32] LunarG (2018b): *Vertex buffers - Vulkan Tutorial.* [En línea]. Disponible en: https://vulkan-tutorial.com/Vertex_buffers/Vertex_input_description [Consulta: 19 de febrero de 2020].

[33] LunarG (2018c): *Uniform buffers - Vulkan Tutorial.* [En línea]. Disponible en: https://vulkan-tutorial.com/Uniform_buffers/Descriptor_layout_and_buffer [Consulta: 20 de febrero de 2020].

[34] LunarG (2018d): *Texture mapping - Vulkan Tutorial.* [En línea]. Disponible en: https://vulkan-tutorial.com/Texture_mapping/Images [Consulta: 12 de abril de 2020].

[35] LunarG (2018e): *Depth buffering - Vulkan Tutorial.* [En línea]. Disponible en: https://vulkan-tutorial.com/Depth_buffering [Consulta: 18 de abril de 2020].

[36] LunarG (2018f): *Loading models - Vulkan Tutorial.* [En línea]. Disponible en: https://vulkan-tutorial.com/Loading_models [Consulta: 18 de abril de 2020].

[37] Meijs, T. (2019): *tntmeijs/GLSL-Shader-Includes: A utility class which adds a way to include external files in a shader file.* [En línea]. Disponible en:

https://github.com/tntmeijs/GLSL-Shader-Includes [Consulta: 27 de noviembre 2019].

[38] michael0884 (2019): *Shader - Shadertoy BETA - Fractal terrain generator* [En línea]. Disponible en: https://www.shadertoy.com/view/3lcGRX [Consulta: 28 de febrero de 2020].

[39] Mullor, R. (2018): *Fractales 3D* [En línea]. Disponible en:

http://personales.upv.es/rmullor/Fractales/Fractales.htm [Consulta: 26 de noviembre 2019].

[40] Myro (2019): *Shader - Shadertoy BETA - mandelbulb raymarched* [En línea]. Disponible en: https://www.shadertoy.com/view/wdjGWR [Consulta: 18 de febrero de 2020].

[41] Nairou (2016): *How do you build a rotation matrix from a normalized vector?* [En línea]. Disponible en:

https://gamedev.stackexchange.com/questions/117262/how-do-you-build-a-rotation-matrix-from-a-normalized-vector [Consulta: 4 de marzo de 2020].

[42] Henning, N. (2016): *sheredom/VkComputeSample.* [En línea]. Disponible en: https://gist.github.com/sheredom/523f02bbad2ae397d7ed255f3f3b5a7f [Consulta: 11 de marzo de 2020].

[43] Nicol Bolas (2017): *glsl - Flipping the viewport in Vulkan - Stack Overflow.* [En línea]. Disponible en: https://stackoverflow.com/questions/45570326/flipping-the-viewport-in-vulkan [Consulta: 22 de febrero 2020].

[44] Quilez, I. (n.d.): *fractals, computer graphics, mathematics, shaders, demoscene and more* [En línea]. Disponible en: https://www.iquilezles.org/index.html [Consulta: 20 de noviembre 2019].

[45] Quilez, I. (2013a): *Shader - Shadertoy BETA - Elevated* [En línea]. Disponible en: https://www.shadertoy.com/view/MdX3Rr [Consulta: 28 de febrero de 2020].

[46] Quilez, I. (2013b): *Shader - Shadertoy BETA - Mandelbulb-derivative* [En línea]. Disponible en:

https://www.shadertoy.com/view/ltfSWn [Consulta: 18 de febrero de 2020].

[47] Quilez, I. (2018): *Shader - Shadertoy BETA - Planet Fall* [En línea]. Disponible en: https://www.shadertoy.com/view/lltBWB [Consulta: 11 de mayo 2020].

[48] Stacktest (2015): *Find the angle between two vectors from an arbitrary origin - Stack Overflow* [En línea]. Disponible en:

https://stackoverflow.com/questions/31064234/find-the-angle-between-two-vectors-from-an-arbitrary-origin [Consulta: 3 de marzo 2020].

[49] supervitas (2018): *Shader - Shadertoy BETA - Polar Night* [En línea]. Disponible en: https://www.shadertoy.com/view/3ss3R4 [Consulta: 29 de febrero de 2020].

[50] The Art Of Code (2018): The Art of Code, *Ray Marching for Dummies!* [En línea]. Disponible en https://www.youtube.com/watch?v=PGtv-dBi2wE [Consulta: 22 de septiembre de 2019].

[51] topsekret (2013): "*Fix" for DirectX Flipping Vertical Screen Coordinates in Image Effects not Working - Unity Forum.* [En línea]. Disponible en:

https://forum.unity.com/threads/fix-for-directx-flipping-vertical-screen-coordinates-in-image-effects-not-working.266455/ [Consulta: 27 de febrero 2020].

[52] Willems, S. (2018): "Vulkan input attachments and sub passes". *Sascha Willems*, 19 de julio [En línea]. Disponible en:

https://www.saschawillems.de/blog/2018/07/19/vulkan-input-attachments-and-sub-passes/ [Consulta: 19 de abril de 2020].

[53] Willems, S. (2019): "Flipping the Vulkan viewport". *Sascha Willems*, 28 de marzo [En línea]. Disponible en:

https://www.saschawillems.de/blog/2019/03/29/flipping-the-vulkan-viewport/ [Consulta: 22 de febrero de 2020].

[54] Wong, J. (2016): *Ray Marching and Signed Distance Functions* [En línea]. Disponible en: http://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/ [Consulta: 27 de octubre 2019].

APÉNDICES

Apéndice A - Contribución de cada participante

En esta sección se explicará la contribución en específico de cada participante del trabajo. El desarrollo del mismo puede verse en el historial de *commits* del repositorio facilitado, aunque éstos no implican necesariamente el trabajo exclusivo de su autor, ya que en la gran mayoría de casos los tres participantes trabajaron juntos en todos ellos. Por tanto, el número de *commits* por autor no es fiable para determinar el reparto de trabajo entre los participantes.

A.1. Diego Baratto Valdivia

Diego Baratto Valdivia colaboró en la elaboración del proyecto en Visual Studio, así como su enlazado con las librerías necesarias, la creación de la ventana, la aplicación en OpenGL, la introducción de los primeros *shaders* con el algoritmo *Ray Marching* y el dibujado de la primera esfera en pantalla mediante esta técnica.

Realizó el sistema de gestión de archivos incluido en la clase *FileHandler*, así como la introducción de la posibilidad de realizar inclusión de otros archivos en los *shaders*, similar a los lenguajes habituales de programación.

La introducción de Vulkan fue una de las principales tareas de las que formó parte, desde la inicialización de la propia *API* hasta la incorporación del modelo, así como la creación de la estructura del proyecto para lograr el correcto funcionamiento de la aplicación, tanto en OpenGL como en Vulkan. También trabajó en la correcta adaptación de los *shaders* de Vulkan para poder usar los mismos archivos que en OpenGL, así como en la automatización de su interpretación al lenguaje SPIR-V mediante el archivo BAT.

Por otra parte, formó parte en el trabajo realizado en la separación en clases del *manager* de Vulkan para poder crear diferentes *shaders* en dicha *API*, como se ha explicado en apartados anteriores, imitando la estructura seleccionada en OpenGL.

Otra de las tareas principales fue realizar la comunicación entre CPU y GPU para la introducción de físicas en el proyecto. Tras investigar las diferentes opciones disponibles, se llegó a la conclusión de que la forma más útil para el trabajo fue la inclusión de los *storage buffer objects* o ssbo que, junto a los *compute shaders* ejecutados en la gráfica, logran una correcta comunicación entre la tarjeta gráfica y la CPU. Tras lograr crear regiones de memoria accesibles desde ambos, introdujo la posibilidad de leer y escribir en dichas regiones desde la CPU, además de crear un prototipo de *compute shader* inicial para probar los *storage buffer objects*.

También trabajó en conjunto en la creación de los algoritmos que permiten las físicas, es decir, la inclusión de la esfera de Fibonacci para establecer una esfera de colisión alrededor del personaje y el uso de las SDF del fractal *snow terrain* para determinar los puntos de colisión con el fractal. Con el objetivo de realizar pruebas, también colaboró en que la esfera de colisión fuese visible junto al propio fractal, así como el seguimiento de la cámara respecto a la misma. También ayudó a mejorar las físicas, logrando que la aplicación de fuerzas al personaje mediante el *input* de usuario además de la gravedad.

Seguidamente, ayudó en la corrección de un problema visual con los colores de Vulkan, así como en la introducción de los modelos en ambas *APIs*, utilizando para ello el test de profundidad, trabajando sobre todo en la parte relacionada con Vulkan. También colaboró en la corrección de una serie de problemas relacionados con las *caches* de Vulkan y en la búsqueda e inclusión de nuevos modelos, entre ellos, Pacman.

Posteriormente realizó en conjunto el menú con las distintas opciones para que el usuario pueda acceder a los diferentes terrenos y escenas creadas en el proyecto de manera sencilla. También ayudó en la creación de nuevos *compute shaders* para aplicar físicas a los diversos

fractales que existen en la aplicación, así como en las físicas tipo “planeta” descritas en apartados anteriores.

Ayudó a solucionar problemas de texturas en los modelos de Vulkan y OpenGL. También introdujo la posibilidad de cambiar diferentes parámetros en cada escena para mejorar la calidad de la presentación, entre ellos, velocidad y posición inicial del jugador, dirección de la cámara y modo exploración.

Por último, colaboró para introducir dos nuevos fractales, el túnel de Mandelbox y *Autumn Terrain*.

En cuanto a la memoria, se encargó de introducir y explicar aquellas partes más relacionadas con su trabajo en específico, como la aplicación de Vulkan o la comunicación entre la GPU y la CPU, además de corregir y completar el documento.

A.2. Jorge Rodríguez García

Jorge Rodríguez García colaboró en la elaboración del proyecto en Visual Studio, así como su enlazado con las librerías necesarias. Más tarde, ayudó en la creación de la ventana, la aplicación en OpenGL, la introducción de los primeros *shaders* con el algoritmo *Ray Marching* y el dibujado de la primera esfera en pantalla mediante esta técnica.

Seguidamente, trabajó en la corrección del FOV de la cámara para la correcta visualización de la esfera y la posterior adición de iluminación sobre ella. Colaboró en la implementación de una cámara móvil y en la introducción de geometrías básicas en los *shaders*, así como sus principales funciones de distancia. Ayudó en la finalización de los objetivos propuestos en la página web de Jamie Wong, *Ray Marching and Signed Distance Functions* [52], dando lugar a la primera escena generando geometrías mediante *Ray Marching*. Introdujo una mejora en el cálculo de la matriz de vista de la cámara y colaboró en la ampliación del sistema de inclusión de archivos en los *shaders* para que ésta fuera recursiva, similar a los lenguajes de programación.

Más tarde, colaboró en la introducción de Vulkan al proyecto, así como en la estructura que sigue éste para gestionar ambas *APIs*, de la forma que se ha explicado en apartados anteriores. Después de esto, su trabajo se centró en la inclusión de fractales como el fractal Mandelbox, en mejorar la estructura de los archivos de los *shaders*, separando funciones comunes en archivos aparte para ser incluidos por otros posteriormente, etc. (como se indica en apartados anteriores de la memoria), y en la corrección de diversos problemas en el proyecto.

Seguidamente, trabajó en la correcta adaptación de los *shaders* de Vulkan para poder visualizar en esta nueva *API* las escenas ya conseguidas anteriormente en OpenGL. Para ello, invirtió el *viewport* de Vulkan y el eje *y* de los *shaders* en esta *API* (ya que en Vulkan el sistema de coordenadas es distinto que en OpenGL), y también retocó la estructura de los archivos de geometrías, modificando los *structs* ya presentes para que éstos funcionasen también correctamente en Vulkan. Más tarde, colaboró en la separación del código de VulkanManager, creando nuevas clases para los *shaders* en Vulkan y así imitar la estructura de clases seguida en

OpenGL. Trabajó en conjunto en el cambio de los archivos de los *shaders* de OpenGL para lograr hacer éstos multiplataforma (cambiando principalmente las estructuras uniformes recibidas y demás sintaxis), consiguiendo así obtener el funcionamiento correcto de todo el proyecto en ambas *APIs* y sin archivos de *shaders* repetidos.

Posteriormente, introdujo un nuevo fractal al proyecto, el *snow terrain*, estableciendo la separación entre los fractales de terreno y los fractales anteriormente introducidos mediante dos archivos con funciones para cada tipo de fractal, fractalFunctions.c y terrainFunctions.c. Como pequeña adición, rescató el antiguo *shader* de la primera escena conseguida y lo actualizó para, al igual que los nuevos fractales, poder ser visto en ambas *APIs* y así incluirlo entre los mapas finales del trabajo.

Colaboró en la introducción de los *storage buffer objects* y los *compute shaders* para lograr comunicación entre GPU y CPU en ambas *APIs*, necesaria para la introducción de las físicas. Para ello, incluyó métodos para leer y modificar estos *storage buffer objects* desde la CPU.

Trabajó en conjunto para la creación de una primera esfera de colisión visible que avanzaba por el *snow terrain*, junto con la creación de una clase de objeto controlable que en un futuro se correspondería con el jugador.

Más tarde, colaboró en la ampliación de la cámara para que ésta siguiese a la esfera de colisión como en un sistema *third person shooter*, y mejoró las físicas existentes añadiendo fuerzas ejercidas por el jugador mediante *input* y otras aplicadas por el entorno como el rozamiento.

Seguidamente, ayudó en la corrección de un error visual en los colores de Vulkan y colaboró en la introducción de modelos, tanto en OpenGL como en Vulkan, activando el *depth testing* en ambas *APIs* para la correcta combinación entre los *shaders* del fractal y del modelo, así como la orientación del mismo dependiendo de la orientación de la cámara, etc.

Colaboró en la corrección de un error con la *pipeline cache* de Vulkan en modo *release* y en la introducción de nuevos modelos como *pacman*.

Posteriormente, colaboró en la elaboración del menú por consola como selector de todos los mapas del proyecto y trabajó en la ampliación de las físicas para hacerlas presentes en el resto de mapas (físicas tipo "planeta" descritas en apartados anteriores).

Introdujo la posibilidad de cambiar entre cámara libre o en tercera persona con la pulsación de una tecla y ayudó a solucionar problemas con las texturas y los modelos en OpenGL.

Implementó giros en el modelo del jugador en función del *input* recibido, simulando así que éste ruede al avanzar por los fractales, y añadió la posibilidad de bloquear la cámara en tercera persona en los mapas para convertirlos en explorables mediante la cámara libre únicamente.

Por último, introdujo dos nuevos fractales, el túnel de Mandelbox y un nuevo fractal con forma de planeta *autumn terrain*, y mejoró la visibilidad y usabilidad de cada mapa, estableciendo parámetros de física específicos en cada fractal, como podrían ser la posición del jugador, el valor de su aceleración, la fuerza de la gravedad, modo exploración con cámara libre, etc.

En cuanto a la memoria, se encargó de escribir y reforzar mediante imágenes explicativas (generalmente de su autoría) todas las partes hechas por él en específico. Se encargó de la misma manera de la tercera parte del contenido trabajado en conjunto con el resto de participantes, que fue repartido equitativamente para su inclusión en la memoria.

A.3. Gonzalo Sanz Lastra

Gonzalo Sanz Lastra colaboró en la elaboración del proyecto en Visual Studio, así como su enlazado con las librerías necesarias. Además ayudó a la creación de la aplicación y ventana, implementación de los *shaders* iniciales basados en Ray Marching junto con el dibujado de primitivas básicas.

Siguió con la implementación de la escena 0, conformando esta diversos tipos de iluminación, la creación de una cámara en primera persona con el campo de visión corregido permitiendo así la correcta imagen de la escena. Una vez se obtuvo la visualización de una esfera con iluminación básica, procedió a introducir dentro de los *shaders* otro tipos de geometrías simples (cilindros, cubos) además de fórmulas de intersección, unión y diferencia entre las mismas. Mezclando todas estas funcionalidades realizó la escena que sirvió como primer objetivo, mostrada en la web Jamie Wong, *Ray Marching and Signed Distance Functions* [52], incluyendo además una animación de la misma. Colaboró en la reorganización de los diversos archivos *shaders* para que futuras ampliaciones y adiciones resultaran más cómodas de implementar.

Ayudó en menor medida en la introducción de Vulkan al proyecto, así como en la estructura que sigue éste para gestionar ambas *APIs*, de la forma que se ha explicado en apartados anteriores.

El siguiente paso que siguió fue la introducción de fractales al proyecto, empezando con la implementación del fractal mandelbulb, y colaboró en la reestructura de archivos *shaders* uniendo partes comunes como por ejemplo las fórmulas de distancia de diversos fractales.

Seguidamente, trabajó en la correcta adaptación de los *shaders* de Vulkan para poder visualizar en esta nueva *API* las escenas ya conseguidas anteriormente en OpenGL. Para ello, creó un fichero .bat que realizaba automáticamente la compilación de los *shaders* al lenguaje SPIR-V. Contribuyó a los cambios realizados en los *shaders* para hacer que funcionaran tanto para OpenGL como en Vulkan, cambiando la versión del compilador y las variables recibidas de

forma que ambas *APIs* puedan leerlas, evitando así la repetición de código para cada una de las plataformas.

Respecto a los *storage buffer objects* o *ssbo*, ayudó a su implementación tanto en OpenGL como en Vulkan, logrando así la comunicación entre CPU y GPU pudiendo crear entonces el llamado *compute shader*, necesario para la introducción de las físicas. Esto se obtuvo añadiendo el struct ssbo en ambas *APIs* con el que la CPU se comunicaría con la GPU.

Colaboró con la creación de la primera esfera que se movía y colisionaba con el fractal de terreno *snow terrain*, además de la clase que controlaba a dicha esfera haciendo la función básica de un *player* o jugador. Contribuyó a la creación de una cámara en tercera persona que seguía y rotaba al rededor de dicho jugador a cierta distancia.

Ayudó a crear y mejorar la física de colisiones basada en fuerzas añadiendo así fuerzas como la de rozamiento en diversos modos diferenciando así el rozamiento del aire con el rozamiento del suelo o fuerzas aplicadas mediante *input* para posibilitar el movimiento del jugador.

A continuación ayudó a introducir modelos de mallas en ambas *APIs*, teniendo que ampliar las clases ya existentes para renderizar varios *shaders* al mismo tiempo, teniendo que añadir una nueva *pipeline* de creación de shaders por cada uno que se creara. Además, colaboró a activar el llamado *depth testing* (en ambas *APIs*) con el objetivo de poder ver simultáneamente al modelo tridimensional con los fractales antes implementados.

Solucionó diversos problemas como la corrección del color en Vulkan o el error de compilación en Vulkan *release*. Otra cuestión que solucionó fue la vibración excesiva de la cámara al seguir continuamente al jugador que estaba colisionando con el suelo, haciendo que la cámara solo le siga cuando el jugador se ha desplazado una distancia mínima en el eje Y.

Añadió un modelo de pacman además de dibujar el mismo su textura permitiendo así la correcta visualización del modelo.

Introdujo la posibilidad de ver la *hitbox* del jugador mediante *input*, pudiendo activarla a placer y ayudó a solucionar problemas de texturas en los modelos de mallas tanto en OpenGL como en Vulkan.

Por último, introdujo dos nuevos fractales, el *ocean terrain* y un nuevo fractal con forma de planeta *autumn terrain*.

En cuanto a la memoria, introdujo fragmentos del propio código del proyecto definiendo también el formato del mismo. Se encargó de la tercera parte del contenido trabajado en conjunto con el resto de participantes, que fue repartido equitativamente para su inclusión en la memoria.

Apéndice B - *Introduction*

The generation of interactive 3D graphic content, such as in video games, is usually carried out using the model based on the rendering pipeline, which acts on vertex mesh models, using some of the APIs for graphical programming in GPUs, such as OpenGL, Direct3D or Vulkan.

In this project, however, for rendering fractals, algorithms such as Ray Marching are used which is based on the distance function formulas explained in later sections, being able to produce different environments only based on these mathematical formulas.

In addition, the application allows the use of OpenGL and Vulkan, both cross-platform, as graphical programming APIs, with GLSL (and SPIR-V) as the main shader language.

B.1. Motivation

The main inspiration has been the video game Marble Marcher - A game of fractals based on physics [6], *based on dynamic terrains generated by fractals in three dimensions. In this application, laws of physics are used on this type of terrain and the objective of the game is go through them in the shortest time possible.*

This generated the main attraction of the project, having the potential or possibility of, using simple mathematical formulas and techniques, generating very complex and dynamic terrains and forms applicable to video games.

B.2. Objectives

The main objective is to develop an interactive application that integrates the generation of terrains using fractals and objects based on triangle meshes. So that they can interact with each other by means of a simple physical movement of the objects on the surfaces generated by fractals, using is the latest techniques of the OpenGL and Vulkan APIs.

To achieve this, the first basic objectives are the creation of terrains using three-dimensional fractals (Ray Marching) and their implementation using two different APIs, OpenGL and Vulkan, exploring all their possibilities and latest techniques. In addition, these basic objectives have been extended by adding communication between the graphics card and the CPU in order to add physics to the terrains, as well as adding the rendering of a model (graphics pipeline), mixing both different rendering techniques.

B.3. Work plan

Once the objectives have been established, the following sequence of stages are created, although they were not followed out in strict order, but they are all the tasks performed.

B.3.1 Ray Marching and OpenGL

The first objective is the study of the Ray Marching algorithm and the development of an application, using OpenGL, to draw any geometric shape given by the signed distance formula (SDF) that Ray Marching requires.

First of all, the possibility of including and using shaders is added so that they are visible in a window. These shaders implement Ray Marching's algorithm and consist of basic geometries as first examples of proof of its correct operation.

The distance formulas of different three-dimensional fractals are introduced into shaders, on which the Ray Marching algorithm will be applied, thus generating the complex dynamic shapes that the project aims to achieve. With this, any geometric shape can be painted given its distance formula in said API, achieving in turn the rendering of three-dimensional fractals.

B.3.2 Study Vulkan graphics API

Having the rendering of fractals in OpenGL, another implementation of the application is implemented using the Vulkan API, allowing the same shaders to be executed in either of the two APIs, adding the specific implementation of the interfaces of each of the APIs.

B.3.3 Collision physics in both APIs

Once the correct operation has been achieved in both APIs, the next step is to achieve precise and adequate communication between the graphics card and the CPU, and thereby incorporate to the application the physical collision with the fractals. To do this, computational shaders have been used, which allow the graphics card to perform the specified calculations, freeing the CPU from a large workload.

Once communication between GPU and CPU is achieved, a Newtonian physics simulator is implemented using again the Ray Marching algorithm to calculate the distances between any object and the fractal and thus be able to calculate the collision with it.

B.3.4 Mesh models and depth testing

Finally, the loading of mesh models in both APIs is made possible, with the aim of integrating the rendering of these models with the fractals, using depth testing and delegating the task of deciding which pixel is displayed to the graphical API used. In this way, the fractal is rendered with the shaders that Ray Marching implements, and the player with the shaders, completely different, classic vertex transformation and triangle fill, thus allowing both different rendering techniques to interact.

B.3.5 User interface

Furthermore, it is offered a simple menu that displays every available option in order to observe and explore every terrain and fractal implemented.

GitHub has been used as a working repository for the project, and the application code can be found at the following link: https://github.com/DiegoBV/TFG-Repo. In order to compile its content, it is needed to download its dependencies, following the instructions given in the README.md file, located in the root directory of the repository.

Apéndice C - *Conclusions*

As previously described, the user can execute the application from both APIs by changing the platform from the Visual Studio editor. Interfaces and necessary classes were made in order to accomplish the objectives. However, the limitations found in the development of the application are also important: split specific functionality of both APIs, class management and other previous problems such as the differences in the depth buffer or the model's operations.

Results obtained in the research on both APIs show that Vulkan's work is far more complex. This happens due to the freedom that Vulkan offers regarding OpenGL. It allows the user to configure a large number of parameters (for instance, what graphic card is going to be used, specialization and task of every GPU or the memory management of different parts of the hardware). It is safe to say that Vulkan is going to be able to set on a new work concept in the companies dedicated to graphic application development.

Ray Marching algorithm, Signed Distance Functions (SDF) and their shader implementation are considered the project's main core. They have made possible to render every fractal previously described. Moreover, thanks to the discovery of compute shaders it has been able to include a simple, physic mechanism using the SDF. Collision between the user controlled model and the terrains are made entirely in the compute shaders executed in the GPU, sending the results to the CPU, allowing the physics' correct behaviour.

Finally, another rendering type has been introduced related to the addition of a model in the application. This is one of the most common types, using the vertices that compose the model's meshes and the depth testing of every fragment or pixel. The mix of both different rendering types has been achieved in a correct way, attending to the specific depth of each moment of execution.

It can be concluded that the original objectives have been accomplished correctly thanks to the acquired knowledge throughout college. It has been delved into the concepts that surround the different application programming interfaces, as well as the different methods, from the

Signed Distance Functions to compute shaders, to make possible the final result, thus acquiring important skills in the video game industry.

C.1. Future work

Taking into account all the goals reached during the development of the project, the next steps are clear: first, improve the code's quality, especially the one related to Vulkan. Second, the addition of new terrains and physics will add new surfaces to work on and upgrade the application. Finally, the user interface needs to improve, add new features that gives the user the feeling of freedom and polish the existing movement and controls. To sum up, giving a considerable variety to the project is the main goal of the future work.