Implementación sobre *hardware* gráfico de la dispersión de la luz en la atmósfera

Gregorio Iniesta Ovejero
Javier Pérez Martínez

Grado en Ingeniería Informática
Facultad de Informática
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

Trabajo del fin de grado del Grado en Ingeniería Informática

Madrid, 19 de junio 2015

Director: Pedro Jesús Martín de la Calle

Índice

Resumen

La atmósfera terrestre está compuesta por partículas que interactúan con la luz solar provocando un comportamiento conocido como dispersión atmosférica de la luz. Es esta dispersión de la luz la causante de que, por ejemplo, los cuerpos lejanos sufran variaciones de color, o de que el cielo tenga diferentes tonalidades según la distancia que la luz recorre a través de la atmósfera.

Desde hace unos años se busca obtener una simulación cada vez más real del comportamiento de la luz en programas utilizados por la industria del entretenimiento, con el fin de crear en el usuario una mejor experiencia.

Gracias a la potencia de cálculo que ofrece actualmente el *hardware* especializado, es posible procesar las fórmulas proporcionadas por los modelos físicos que estudian estos fenómenos.

El propósito del trabajo descrito en esta memoria es implementar, de manera realista, el comportamiento de la luz en su paso por la atmósfera en un escenario tridimensional utilizando *hardware* gráfico. Para obtener una simulación en tiempo real, se han empleado las tecnologías actuales de programación gráfica. La finalidad es imitar el comportamiento de la luz en cualquier momento del día, para cualquier posición de la cámara, incluso cuando se encuentra fuera de la atmósfera. Es decir, un programa que permita que la sensación del usuario en cualquier escenario, sin importar la posición del sol ni del punto de vista, sea lo más realista posible en lo que concierne a la luz.

La simulación del efecto provocado por la dispersión atmosférica no proporciona por sí sola un resultado suficientemente realista. Pero si lo acompañamos de otras técnicas de uso común en la iluminación de entornos tridimensionales, como son la proyección de sombras o el *Phong shading*, obtenemos un resultado notablemente superior.

La forma de lograr un rendimiento aceptable de la simulación pasa por el uso de estructuras de programación que agilicen el cálculo como, por ejemplo, tablas de consulta, mapas de sombras y optimización del código orientada a la arquitectura del *hardware* gráfico moderno.

Abstract

The Earth's atmosphere is composed of particles that interact with sunlight causing a behavior known as atmospheric scattering. It is this scattered light which causing, for example, distant bodies undergo color variations, or the sky to have different shades depending on the distance light travels through the atmosphere.

For some years it is sought an ever more real behavior of light in software used by the entertainment industry, in order to create a better user experience simulation.

With computing power currently provides specialized hardware, it is possible to process the equations provided by the physical models studying these phenomena.

The purpose of the work described herein is implement, realistically, the behavior of light as it passes through the atmosphere in a three-dimensional scene using graphics hardware. For a real-time simulation, we have used modern technologies in graphic programming. The purpose is to mimic the behavior of light at any time of day, for any camera position, even when the camera is out of the atmosphere. That is, a program allowing the user to feel at any stage, regardless of the sun's position and the point of view, as realistic as possible with respect to light.

The simulation of effect caused by atmospheric dispersion does not provide a sufficient realistic result by itself. But if we apply other techniques commonly used in three-dimensional environments lighting, such as casting shadows or Phong shading, we get a significantly better result.

The way to achieve acceptable performance of the simulation through the use of programming structures to speed up the calculation, for example, lookup tables, shadow maps and code optimization oriented to modern graphics hardware architectures.

1. Introducción

1.1. Antecedentes

El estudio sobre la dispersión de la luz en la atmósfera no es algo novedoso, lo relativamente novedoso es conseguir que la simulación del comportamiento de la luz en una imagen sea de buena calidad y sea capaz de generar imágenes en tiempo real (60 fotogramas por segundo).

En 1987 Klassen sienta las bases físicas de la dispersión lumínica e implementa un modelo simplificado en CPU. Pero este modelo da por hecho que tanto la tierra como su atmósfera son planas, y que la densidad del aire a lo largo de la atmósfera así como la intensidad de la luz son constantes.

Nishita (entre 1993 y 1996) realiza un notable avance al aplicar un modelo terrestre esférico e introducir variaciones en la densidad del aire en la atmósfera en función de la altura. También introduce en sus ecuaciones el cálculo de la dispersión por las moléculas del aire (dispersión de Rayleigh) y la provocada por los aerosoles (dispersión de Mie). Estas ecuaciones, aunque válidas, son demasiado complejas para realizar su cálculo en tiempo real, porque hacen uso de integrales.

En 1999, Preetham mejora la precisión de las ecuaciones de los fenómenos de dispersión Rayleigh y Mie a lo largo de la atmósfera.

Hoffman, en 2002, realiza la primera aproximación implementada en GPU (*Graphics Processor Unit*, Unidad de Procesamiento Gráfico) del algoritmo de la dispersión de la luz en la atmósfera. Para obtener los cálculos en tiempo real, utiliza una atmósfera plana y con densidad constante, logrando así resultados aceptables cuando la cámara se mantiene cerca de la superficie.

Entre 2004 y 2005 Sean O'Neil en [1], siguiendo el modelo de Nishita, implementa las ecuaciones completas computables en tiempo de ejecución, manteniendo una alta tasa de imágenes por segundo. Mejora el algoritmo generalizando una parte del cálculo integral en una tabla precomputada.

Más recientemente, en 2012, Schüler en [2] mejora la precisión del cálculo integral con la función de Chapman y usando *ray marching*.

Pero no es hasta 2014 cuando se alcanza una reducción significativa de la cantidad de cálculos. Para conseguirlo, Yusov en [3] utiliza "*epipolar sampling*" con árboles "minimax" de una dimensión y mejora el modelo físico eliminando del cálculo integral las partículas que no están siendo iluminadas.

1.2. Objetivo

Este proyecto ha sido elaborado con el objetivo de crear una simulación en tiempo real del fenómeno de la dispersión de la luz provocado por la atmósfera en un entorno tridimensional, utilizando herramientas que aprovechen el *hardware* gráfico, y con vistas a su aplicación en escenarios realistas.

1.3. Plan de trabajo

Teniendo en cuenta el objetivo principal, estos son los objetivos parciales que nos hemos propuesto:

- Aprender a programar la GPU con fines gráficos.
    - Utilizando una versión de librería gráfica moderna.
    - Conociendo el funcionamiento de la tubería gráfica.
    - Investigando el desarrollo de *shaders*.
    - Profundizando en la optimización del código con fines gráficos.
- Explorar la carga de mallas tridimensionales desde archivo.
- Trasladar un modelo realista de la dispersión de la luz a OpenGL.
- Estudiar diferentes sistemas de sombras dinámicas en entornos a gran escala.

1.3.1. Organigrama temporal de objetivos

Octubre	Decidir la temática y definir el objetivo principal.
Noviembre	Buscar información acerca de los trabajos previos en la materia.
Diciembre	Estudiar la programación moderna en *hardware* gráfico.
Enero	Aprender a utilizar la librería de gráficos OpenGL y su lenguaje anexo GLSL (OpenGL *Shading Language*).
Febrero	Montaje de la arquitectura del proyecto. Investigar e implementar la carga de mallas de tipo "Wavefront .obj".
Marzo	Primera aproximación del proyecto en la que se realiza una implementación inicial del modelo realista de la dispersión de la luz.
Abril	Segunda aproximación en la que se realizan optimizaciones y se mejoran los resultados.
Mayo	Simulación final añadiendo mapas de sombras, efectos como *light shaft* y ajuste de parámetros. Realización de la memoria.

1.3.2. Metodologías usadas en el desarrollo

Se ha utilizado una metodología de programación extrema. Había una funcionalidad implementada cada una o dos semanas y cada mes, aproximadamente, había una versión totalmente estable del proyecto.

Se ha elegido esta metodología porque se tenía claro el objetivo principal, pero existían muchas tareas opcionales que mejorarían el resultado final. Durante el desarrollo se contemplaban diferentes alternativas y al final de cada iteración se estimaba cuál iba a ser la siguiente, en función de su importancia y de la dificultad estimada para implementarla.

Además, se puso en práctica la técnica de "programación en pareja", o *pair programming*, que consiste en que dos programadores participan simultáneamente en un mismo sitio de trabajo. Mientras uno escribe código, el otro se encarga de revisar el trabajo y de verlo de manera más global y, cada cierto tiempo, se hace un intercambio de roles.

1. Introduction

1.1. Preview

The study about atmospheric scattering is not new. The newest is to simulate the behavior of the light getting a great quality image that can be rendered in a real time (60 frames per second).

Klassen in 1987 provides the physical basis of light scatter and implements a simplified model running on CPU. But this model assumes that both the earth and its atmosphere are flat, and air density along the atmosphere and the intensity of light are constant.

Nishita (1993 to 1996) takes a significant step forward by applying a spherical earth model and introduce variations in the density of air in the atmosphere according to height. He also added in his equations to calculate scattering by air molecules (Rayleigh scattering) and caused by aerosols (Mie scattering). These equations, though valid, are too complex for real-time calculation, because they make use of integrals.

In 1999, Preetham improves the accuracy of the equations relating to the phenomena of Rayleigh and Mie scattering along the atmosphere.

Hoffman, 2002, makes the first approach algorithm of light scattering in the atmosphere implemented in GPU (Graphics Processor Unit). He uses a flat and constant density atmosphere to obtain real-time calculations and achieved acceptable results when the camera is held near the surface.

Between 2004 and 2005 Sean O'Neil at [1] and taking Nishita's model, implements the complete equations that can be computed on runtime, maintaining a high frame rate per second. He improved algorithm generalizing a part of integral calculus in a precomputed table.

More recently, in 2012, Schüler at [2] improves the accuracy of integral calculus with the Chapman function and using ray marching.

But it was not until 2014 when a significant reduction in the amount of computation is reached. To achieve this, Yusov at [3] use "epipolar sampling" with "minimax" 1D trees

and he improves physical model of integral calculus removing particles that are not being lit.

1.2. Objective

This project has been developed with the objective of creating a real-time simulation of the phenomenon of light scattering caused by the atmosphere in a three-dimensional environment, using tools that take advantage of graphics hardware, and with the intention of applying it in realistic scenarios.

1.3. Work Project

Given the main objective, we have set ourselves these partial objectives:

- Learn to program the GPU with graphic purposes.
    - Using a modern graphic library version.
    - Knowing the graphic pipeline functionalities.
    - Searching about the shader development.
    - Studying about code optimization with graphic purposes.
- Explore the three-dimensional mesh load from file.
- Port a realistic light scattering model to OpenGL.
- Study about different dynamic shadow systems in large-scale environments.

1.3.1. Temporal objective organization chart

October	Decide the subject And define the main objective.
November	Look for information about previous work on the subject.
December	Study modern graphics hardware programming.
January	Learn to use the OpenGL graphics library and GLSL. (OpenGL Shading Language).
February	Set up the project architecture. Research and implement the “Wavefront .obj” mesh load.
March	First project approach in an initial implementation of realistic light scattering model takes place.
April	Second approach in which optimizations are performed and the results are improved.
May	Final simulation by adding shadow maps, effects such as light shaft and parameters adjustment. Perform memory.

1.3.2. Software development methodology

An extreme programming methodology was used. There was a functionality implemented every one to two weeks and every month or so, there was a completely stable version of the project.

This methodology was chosen because the main objective was clear, but there were many optional tasks that improve the bottom line. During development alternatives they were contemplated. At the end of each iteration was estimated what would be the next task, depending on their importance and estimated difficulty to implement.

In addition, the "pair programming" technique was taken in practice, which consists of two programmers involved simultaneously in the same workplace. As one write code, the other is responsible for reviewing the work and sees it more globally. From time to time, an exchange of roles is made.

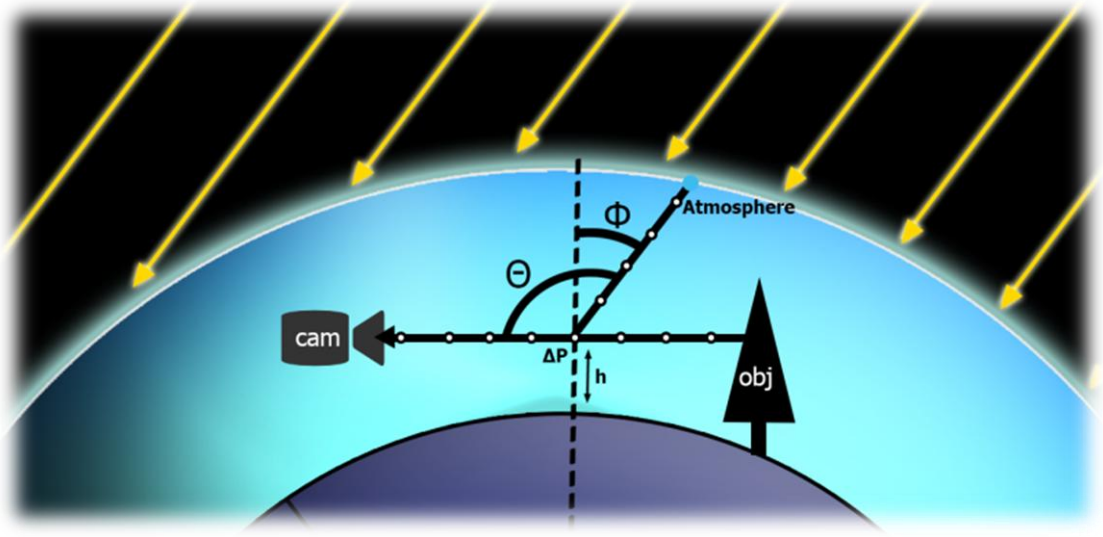
2. Fundamentos teóricos

2.1. Modelo físico del aire

En el vacío la luz viaja en línea recta si no hay nada que la perturbe. Al penetrar en la atmósfera, la luz del sol interactúa con las partículas del aire, ya sea con un grano de polvo (o cualquier otra sustancia en suspensión) o con una molécula. Según el tipo de interacción se producen dos fenómenos distintos, conocidos como dispersión de Rayleigh y dispersión de Mie.

En el caso de la primera, las partículas en suspensión provocan que la luz incidente se refleje en todas las direcciones sin modificar su color. Esto ocurre porque las partículas son de mayor tamaño que la longitud de onda de la luz. En el de la segunda, cuando una onda luminosa impacta contra una molécula, esta absorbe la luz para posteriormente emitirla en otra dirección, siendo mucho más eficiente a la hora de dispersar las longitudes de onda corta que las de onda larga.

Los rayos de luz dispersados por las partículas pueden atravesar el punto desde el cual se visualiza la escena (habitualmente llamado "cámara"), produciéndose lo que se denomina "dispersión incidente" o *in-scattering*. En caso contrario, la luz se pierde en lo que se denomina "profundidad óptica", también conocida como *out-scattering*. Para determinar la cantidad de luz dispersada individualmente por cada partícula se utiliza la llamada función fase.



Esquema de la geometría de la dispersión de la luz en la atmósfera

2.1.1. La función fase

La función fase determina la cantidad de luz dispersada por una partícula en la dirección de la cámara. El resultado depende de una constante G y del ángulo Θ formado por la dirección de incidencia de la luz solar y el vector determinado por la cámara y el objeto sobre el que se aplica la dispersión (ver imagen anterior). La constante G se utiliza para modificar la simetría de la dispersión entre los ángulos positivos y negativos.

Existen diferentes alternativas para implementar la función fase. Por ejemplo, en [1] se utiliza una adaptación de Henyey-Greenstein. En nuestro caso, nos hemos decantado por usar la función que aparece en [3], comúnmente conocida como la función de Cornette-Shanks.

$$phase(\theta) = \frac{1}{4\pi}\,\frac{3(1-G^2)}{2(2+G^2)}\,\frac{1+\cos^2(\theta)}{(1+G^2-2G\cos(\theta))^{\frac{3}{2}}}$$

2.1.1.1. La función de Rayleigh

La dispersión producida por las moléculas del aire sobre la luz está descrita de manera precisa por la teoría de Rayleigh y por ello es conocida como “dispersión de Rayleigh”. Este fenómeno provoca que el cielo sea azul y que el sol en ocasiones se vea amarillo.

La distribución angular de la luz en la dispersión de Rayleigh se define como simétrica. Teniendo en cuenta esta característica, los cálculos se simplifican considerablemente en la función fase.

$$phase_{Ray}(\theta) = \frac{3}{16\pi}(1+\cos^2(\theta))$$

2.1.1.2. La función de Mie

La dispersión ocasionada por los aerosoles es descrita por la teoría de Mie. Al contrario de la teoría de Rayleigh, la distribución angular es asimétrica. Esta característica es la causante de que aparezca un halo de luz blanca alrededor del sol y esa tonalidad grisácea que podemos observar en el cielo.

$$phase_{Mie}(\theta) = phase(\theta)$$

Los valores recomendados para la constante G de la función fase están comprendidos entre 0.75 y 0.999, y no se deben utilizar los valores 1 o -1 debido a que pueden anular el divisor cuando G y cos(Θ) valen lo mismo. Los valores menores aumentan el halo del sol, mientras que los valores cercanos a 1 definen mejor su silueta.

2.1.2. La ecuación de dispersión

El cálculo de la dispersión lumínica depende esencialmente de dos factores complementarios. Por un lado del *out-scattering* y por otro del *in-scattering*. Ambos dependen de la profundidad óptica y de la función fase.

La profundidad óptica mide la densidad del aire entre dos puntos. Para ello se realiza el cálculo integral de la densidad de partículas a lo largo de la trayectoria del segmento definido por los dos puntos. Es importante tener en cuenta que la densidad de la atmósfera (ρ) varía exponencialmente en función de la altura (h). El cálculo de la densidad se escala con la constante ρ_0 que determina la densidad a nivel del mar.

$$\rho = \rho_0 e^{-h/H}$$

La constante H se denomina "altura de escala de partículas" y se refiere a la altura que tendría la atmósfera en el caso de que la densidad fuera uniforme. En el caso de los aerosoles (Mie), al ser partículas más densas, se concentran en la zona baja de la atmósfera y por eso su constante H es menor respecto a la de Rayleigh. Los valores utilizados para Rayleigh y Mie son los siguientes.

$$H_{Ray} = 7994m, H_{Mie} = 1200m$$

La ecuación de la profundidad óptica (τ) se calcula a partir del coeficiente de extinción del color a nivel del mar (β^{ext}) y de la densidad entre los dos puntos para los que se calcula.

$$\tau(\overline{AB}) = \beta^{ext} \int_A^B e^{-h(t)/H}\, dt$$

Para poder entender la ecuación de profundidad es necesario conocer el coeficiente de extinción (β^{ext}), que se obtiene a partir del coeficiente de dispersión y de absorción. El coeficiente de extinción sigue esta fórmula:

$$\beta^{ext} = \beta^{scatt} + \beta^{abs}$$

El coeficiente de dispersión (β^{scatt}) determina la cantidad de energía dispersada por unidad de longitud. La cantidad de energía perdida por unidad de longitud (en este caso, por metro) a lo largo de la atmósfera, causada por la absorción y la dispersión de las partículas, se determina con los coeficientes de absorción (β^{abs}) y extinción (β^{ext}) respectivamente. Tanto Rayleigh como Mie tienen sus propios coeficientes de dispersión y absorción.

Como en [3] y en otros trabajos previos (como [1], [4] y [5]), el coeficiente de dispersión del color al nivel del mar para la función de Rayleigh es el siguiente:

$$\beta_{Ray}^{scatt} = (5.8, 13.5, 33.1)10^{-6}/m$$

El coeficiente de dispersión del color al nivel del mar para Mie también ha sido obtenido de [3]:

$$\beta_{Mie}^{scatt} = (2.0, 2.0, 2.0)10^{-5}/m$$

Mientras que el coeficiente de absorción de Rayleigh se considera despreciable, el de Mie es proporcional al de dispersión:

$$\beta_{Mie}^{abs} = 0.1\beta_{Mie}^{scatt}$$

Finalmente, la ecuación que calcula la profundidad óptica en la atmósfera se define como la suma de la profundidad óptica de ambos tipos de partículas entre dos puntos (A y B) que delimitan el segmento en el que se calcula dicha profundidad.

$$\tau(\overline{AB}) = \beta_{Ray}^{ext} \int_A^B e^{-h(t)/H_{Ray}}\, dt + \beta_{Mie}^{ext} \int_A^B e^{-h(t)/H_{Mie}}\, dt$$

2.1.2.1. Out-scattering

La cantidad de luz dispersada que no es percibida por la cámara se denomina *out-scattering* y produce la extinción del color, es decir, al aumentar la dispersión de luz disminuye el contraste de la imagen. El nuevo color tras la extinción se calcula mediante la evaluación de la profundidad óptica desde la cámara, en la posición C, al punto O, perteneciente al objeto visible, y se multiplica por el color original del punto en cuestión.

$$Color_{out}(O) = Color_{original}(O) * \tau(\overline{CO})$$

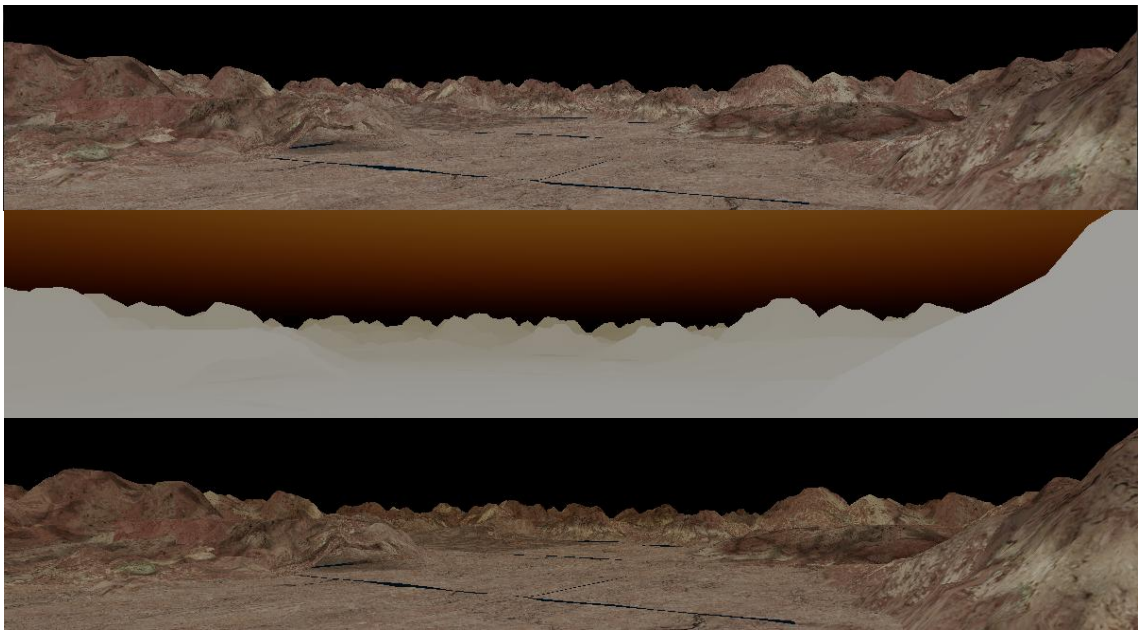


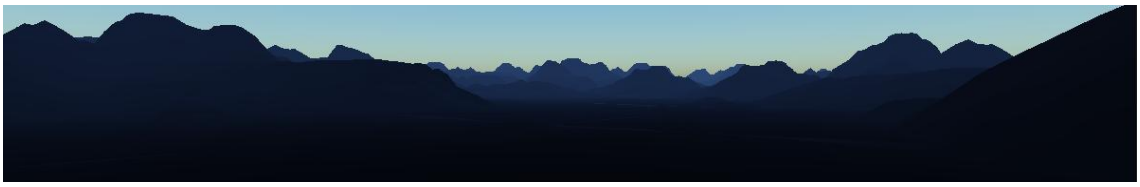
Imagen sin out-scattering *(arriba), cálculo de* out-scattering *sobre blanco (centro) y resultado final (abajo)*

2.1.2.2. In-scattering

La ecuación de *in-scattering* define la cantidad de luz que es añadida a la trayectoria punto-cámara. Para ello, por cada punto intermedio (P), se tiene en cuenta la cantidad de luz que llega desde A, que corresponde al punto de entrada de la luz en la atmósfera. Si el punto P se encuentra en sombra, no se añade luz. La forma de calcular el color tras añadir la luz percibida en la trayectoria es resolviendo la integral entre la posición C de la cámara y el punto O perteneciente al objeto visible.

$$Color_{in}(P) = \int_{C}^{O} Color_{Sun} * e^{-\tau\left(\overline{A(s)P(s)}\right)} * e^{-\tau\left(\overline{P(s)C}\right)} * V\big(P(s)\big) * \Big(\beta_{Ray}^{scatt}\, e^{-h(s)/H_{Ray}}\, phase_{Ray}(\theta) + \beta_{Mie}^{scatt}\, e^{-h(s)/H_{Mie}}\, phase_{Mie}(\theta)\Big)\, ds$$

Donde la función V es 0 si el punto proporcionado está en sombra y 1 si está iluminado.



Cálculo de in-scattering *sobre negro*

2.1.2.3. Ecuación final

Una vez resueltas las ecuaciones de *in-scattering* y *out-scattering*, existen varias alternativas para obtener el color final del punto. Se ha optado por la ecuación propuesta por Sean O'Neil en [1]. Es una ecuación sencilla que, además, cuenta con un factor de exposición constante para simular la apertura de una pupila o la del diafragma de una cámara. En este trabajo, dicho factor pasa a ser variable y, aprovechando el ángulo γ formado entre la cámara y la dirección de la luz, disminuye la apertura cuando el ángulo es pequeño. El principal objetivo de esto es que, cuando se mire directamente al sol, la imagen reaccione igual que reaccionarían nuestros ojos, cerrando la pupila, y como resultado la imagen percibida sea más oscura. La función *clamp* mantiene el resultado del primer parámetro, limitado entre el segundo y el tercero; en este caso, no puede ser superior a 1 ni inferior a 0.

$$pupil = clamp(1 - \frac{\cos(\gamma)}{4}, 0, 1)$$

$$Color_{final}(O) = 1 - e^{-pupil*(Color_{out}(O) + Color_{in}(O))}$$

2.1.3. Resolviendo la ecuación

La ecuación que determina el color final de un punto va a ser resuelta por aproximaciones, ya que las integrales que se manejan no admiten soluciones analíticas. El proceso que vamos a seguir calcula una aproximación que mejora en función del tiempo de cómputo que se invierta.

Las integrales de este proyecto se calculan sobre el segmento formado entre dos puntos, el proceso consiste en aplicar la integral de Riemann. Esto divide al segmento en un número de subintervalos para los que se computa su contribución al resultado de la integral. Cuanto más alto sea el número de subintervalos mayor será la precisión y el tiempo de cómputo. Esto en *software* se lleva a cabo con un bucle que computa valores en los puntos intermedios.

Para calcular el color final lo primero es evaluar la cantidad de color que se extingue, para lo cual hay que calcular la profundidad óptica entre el observador y el objeto observado. Esto implica el cálculo de una integral, como se ha mencionado en el párrafo anterior.

La cantidad de color que se le añade a la imagen debido al *in-scattering* es la segunda operación a resolver. Es bastante compleja porque hay que calcular dos integrales por cada punto intermedio de la integral principal. La primera ($e^{-\tau(\overline{A(s)P(s)})}$) obtiene la profundidad óptica desde el punto P del subintervalo (s) hasta el punto A de corte con la atmósfera. El punto A se obtiene mediante el vector que indica la dirección de la luz y el punto P del subintervalo, con esto construimos una recta y calculamos su intersección con la atmosfera obteniendo así el punto A. Todo ello se puede simplificar de tal manera que se realicen las cuentas en dos dimensiones, utilizando un círculo para la atmósfera en lugar de una esfera. Esta integral se resuelve aplicando de nuevo el planteamiento de Riemann.

La segunda integral del *in-scattering* ($e^{-\tau(\overline{P(s)C})}$) se calcula, en realidad, mediante una acumulación de todos los puntos calculados en iteraciones anteriores, de tal manera que no hay que recurrir a un nuevo bucle para resolverlo.

La función V(P(s)) indica si el punto P del intervalo (s) se encuentra en sombra o está siendo iluminado. El apartado "3.4.2. Cálculo de sombras" está dedicado a cómo resolver el cálculo de la visibilidad.

La única parte que podría resultar compleja y necesitar explicación es $e^{-h(s)/H_{xxx}}$. La función h se refiere a la altura del punto en el subintervalo (s) y queda dividida por la H de Mie o de Rayleigh. La definición de H_{xxx} ya se vio en el apartado "2.1.2. La ecuación de dispersión". El resto de la ecuación del *in-scattering* es trivial ya que únicamente consiste en realizar sumas y multiplicaciones.

Habiendo resuelto $Color_{out}$ y $Color_{in}$ solo nos falta calcular la apertura de la pupila y resolver el resto de la ecuación. Dentro de la ecuación de la pupila hay que calcular el coseno del ángulo entre la dirección en la que mira la cámara y la dirección de la luz. La manera más sencilla de calcularlo es con el producto escalar:

$$\cos(\gamma) = \overrightarrow{Cam_{forward}} \cdot -\overrightarrow{light_{dir}}$$

Así se obtienen todos los elementos necesarios para resolver la ecuación que calcula el color final de un punto.

3. Implementación

El principal objetivo del trabajo es que la implementación de la dispersión de la luz en la atmósfera funcione en tiempo real en un ordenador personal. Una gran parte del algoritmo tiene que ver con el cálculo de integrales, lo que implica un importante coste de procesamiento. Para poder afrontar este tipo de algoritmos en tiempo real es necesario utilizar herramientas de aceleración *hardware*. Además, resulta interesante el uso de ciertas estructuras de programación que agilicen el proceso, como las tablas precomputadas.

Las tecnologías elegidas para el desarrollo son las siguientes:

- OpenGL y GLSL para desarrollo sobre GPU, por ser multiplataforma y *open source*. También porque ya se poseía cierto conocimiento previo.
- C++ como lenguaje de programación, por ser uno de los lenguajes de alto nivel más eficientes. Además, OpenGL está desarrollado en C, lo que facilita la integración en el proyecto.
- SDL2 como sistema de ventanas, por ser libre y de uso profesional.
- Visual Studio como entorno de desarrollo con el *plugin* Nvidia Nsight, para realizar la depuración en GPU y obtener perfiles de ejecución con el fin de detectar cuellos de botella.

3.1. OpenGL, *shaders* y GPU

El elemento principal de procesamiento gráfico en GPU es la tubería de "renderizado". Su función es recibir la representación de una escena tridimensional como entrada y generar una imagen bidimensional como salida. La tubería de "renderizado" puede simplificarse en tres partes:

- La primera parte se denomina *vertex shader* y es la encargada de recibir información sobre los vértices y operar sobre ellos. Habitualmente en esta fase se aplica la matriz Modelo-Vista-Proyección (en adelante MVP). Además, se pueden realizar otras operaciones a nivel de vértices individuales, como la asignación de las coordenadas de textura, color e incidencia de la luz. Es importante por ser una de las partes programables en GLSL.

- La segunda parte de la tubería es estática y por lo tanto no se puede programar. Su principal función es la "rasterización" y el descarte de fragmentos que se encuentran fuera del área visible de la escena. La salida está compuesta por los fragmentos generados tras el proceso (uno por píxel).
- La tercera parte se denomina *fragment shader* en GLSL y *pixel shader* en HLSL (*High Level Shading Language*, usado por la librería Direct3D), y se ubica en la parte final de la tubería gráfica. Al igual que el *vertex shader*, es programable y en este caso recibe fragmentos que corresponden a la información necesaria para computar un píxel en el puerto de vista. Se utiliza para determinar el color final de cada fragmento de forma individual, aplicando modificadores como texturas, luces y sombras. Es aquí donde se encuentra la mayor parte de la carga de procesamiento para el cálculo de la dispersión de la luz.

Existen otras partes programables no mencionadas (ver [6] y [7]), como el *tesselation shader* y el *geometry shader*, que realizan operaciones sobre el conjunto de vértices y se encuentran inmediatamente después del *vertex shader*. A diferencia del *fragment* y el *vertex shader*, son opcionales y su uso no ha sido necesario para este trabajo.

3.1.1. OpenGL

OpenGL es una interfaz de programación de aplicaciones que permite acceder a funcionalidades del *hardware* gráfico. Gracias a la portabilidad, últimamente se ha popularizado su uso a través de variantes, como OpenGL ES para sistemas empotrados. Ha demostrado tener el rendimiento necesario para ser una alternativa real frente a otras librerías privativas. Otra de las razones por las que se ha elegido es el gran soporte que ofrece su comunidad.

La capacidad de programar la tubería de la GPU a alto nivel se incluye desde la versión 2.0 de OpenGL, siendo 4.5 la versión actual. Las tarjetas gráficas pueden ser programadas con diferentes niveles de abstracción, siendo OpenGL ARB (*Arquitecture Review Board*) el más bajo. La alternativa de alto nivel más extendida en OpenGL es GLSL. Se han añadido al proyecto las librerías GL3W (OpenGL3 *Wrangler*) y GLEW (OpenGL *Extension Wrangler*), que unifican y facilitan el uso de las extensiones de OpenGL.

3.1.2. GLSL – *OpenGL shading language*

Este lenguaje fue creado por OpenGL ARB basándose en la sintaxis de C. Se creó desde un principio para facilitar el uso de las nuevas herramientas que proporcionaba OpenGL para programar la tubería gráfica. La especificación del lenguaje facilita el cálculo vectorial y soporta de forma nativa estructuras típicas de procesamiento gráfico como matrices, *buffers* o texturas. También incluye funciones matemáticas sobrecargadas para aceptar los diferentes tipos de datos.

3.2. Primera aproximación

El primer objetivo es obtener la imagen de una escena en un entorno abierto y aplicar sobre ella las funciones de dispersión simplificadas y sin optimizaciones. A partir de este punto, se podrá trabajar en las mejoras visuales y de rendimiento. En una aproximación previa, se utilizó una atmósfera plana, pero después se pasó al modelo esférico aquí descrito.

La proyección utilizada para la visualización es la perspectiva cónica. Es un sistema de representación gráfico que se basa en la proyección de un objeto tridimensional sobre un plano mediante la proyección de rectas que atraviesan un punto. En informática gráfica, se utilizan matrices de transformación para aplicar este tipo de proyecciones y se suelen manejar en el *vertex shader*.

En este proyecto concreto, la matriz de transformación MVP se aplica en el *vertex shader* porque en la fase de descarte de fragmentos se filtran los píxeles que, después de aplicar la matriz, están dentro de los límites del volumen del puerto de vista. El resto de operaciones se realizan en el *fragment shader*, ya que así la iluminación es más realista a nivel de píxel y se pueden aplicar efectos como los que se consiguen con los mapas de normales.

3.2.1. Pseudocódigo del *vertex shader*

Aplicamos la matriz MVP en dos fases, ya que nos interesa la posición del objeto después de aplicar la matriz de modelado (`model_matrix`), pero antes de aplicarle la proyección (`projection_matrix`). Este valor será el utilizado para realizar los cálculos dependientes de la posición en el *fragment shader*. La variable `gl_Position`

es interna a GLSL y es la que se utilizará para realizar el filtrado en la fase de descarte de fragmentos. La variable de salida `vertex_tex` guarda las coordenadas de la textura.

Las variables de entrada son cuatro: `vPos`, que guarda la posición del vértice; `vTex`, que almacena las coordenadas de la textura; `model_matrix`, que contiene la matriz del modelo; y `proyection_matrix`, que aúna la matriz de vista y la de proyección.

Las variables de salida son las siguientes: `gl_Position`, que corresponde al vértice multiplicado por la matriz MVP; `obj`, que es el vértice multiplicado únicamente por la matriz de modelado; y `vertex_tex`, que guarda las coordenadas de la textura para ese vértice.

```
vec4 pos = (model_matrix * vec4(vPos, 1));
vertex_tex = vec4(vTex, 0, 1);

gl_Position = projection_matrix * pos;
obj = pos.xyz;
```

3.2.2. Pseudocódigo del *fragment shader*

El *fragment shader* se encarga de realizar todas las cuentas que tienen que ver con el cálculo de la dispersión atmosférica. Otros autores, como [8], realizan este cálculo en el *vertex shader* por ser mucho menor la carga de trabajo, pero el resultado es peor y se producen efectos no deseados.

Tiene muchos parámetros de entrada debido a la cantidad de datos que son necesarios para configurar la dispersión. Recibe las variables de salida del *vertex shader* (`gl_Position`, `vertex_tex` y `obj`). Las constantes de la dispersión son las que siguen:

- `H_R` y `H_M` corresponden a H_{Ray} y H_{Mie} fueron vistas en los fundamentos teóricos.
- `WORLD_RADIUS` es el radio de la Tierra.
- `C_EARTH` es el centro de la Tierra. Se utiliza en las otras aproximaciones.
- `ATM_TOP_HEIGHT` se utiliza en "`calculateAtmospherePoint`" e indica la altura de la atmósfera con respecto a la superficie terrestre.

- `ATM_RADIUS` es el radio de la atmósfera desde el centro de la Tierra y se utiliza en las otras aproximaciones.
- `M_PI`, `_3_16PI` y `_3_8PI` corresponden a π, 3/16π y 3/8π respectivamente, así se evita la repetición de cuentas en cada píxel.
- `G` y `G2` corresponden a las constantes G y G^2 de la función fase, también se proporcionan ambas para disminuir la cantidad de cálculos.
- `P0`, cuyo valor base es 1 y, si se modifica, escala la densidad de Mie y de Rayleigh al nivel del mar.
- `betaER`, `betaEM`, `betaSR`, `betaSM` corresponden a β_{Ray}^{ext}, β_{Mie}^{ext}, β_{Ray}^{scatt}, β_{Mie}^{scatt} respectivamente del apartado "2.1.2. La ecuación de dispersión".
- `lightDir` y `lightSun` son la dirección y la intensidad de la luz del sol respectivamente.

El uso de todas estas constantes está descrito y explicado más adelante, en la sección "6.3. Constantes utilizadas". A continuación se procede a explicar la primera versión del código. Este extracto de código es un fragmento del utilizado para la primera versión y pretende ser meramente ilustrativo, no funcional.

Se calcula la distancia al punto sobre el que se integrará y se divide entre el número de pasos que se vayan a dar para afinar la integral. Después, definimos las variables que van a ser incrementadas en la integración.

```
vec3 dP = (obj - cam) / N_STEPS;
float ds = length(dP);
vec2 dPC = vec2(0, 0);
vec3 rayLeigh = vec3(0, 0, 0);
vec3 mieScattering = vec3(0, 0, 0);
```

El bucle de la integral se repetirá en función del nivel de detalle. Esto, a su vez, repercutirá en el rendimiento.

```
vec3 cEarth = vec3(0.0f, -WORLD_RADIUS, 0.0f);
for (float s = 0.5f; s < N_STEPS; s += 1.0f){
```

Se obtiene un punto intermedio y la información necesaria sobre él para calcular el paso de la integral.

```
vec3 point = cam + dP * s;
float h = length(point - cEarth) - WORLD_RADIUS;
vec3 normalEarth = point - cEarth / length(point - cEarth);
```

Se aplica la fórmula de la densidad para Rayleigh y para Mie según la altura.

```
vec2 pRM = P0 * exp(-h / vec2(H_R, H_M));

float cosPhi = dot(normalEarth, -normLightDir);
```

Para calcular la densidad desde el punto hasta la atmósfera se realiza otra integral. En la siguiente aproximación, el código mejora extrayendo los resultados de una tabla de datos calculada previamente, sección “3.3.1. Función de Chapman”.

```
vec2 dAP = vec2(0.0, 0.0);
vec3 A = calculateAtmospherePoint(cosPhi, point, cEarth);

vec3 delta_A = (point - A) / N_STEPS;
float diferential_h = length(delta_A);

for (float step = 0.5f; step < N_STEPS; step += 1.0f) {
  float hPoint = length(A + delta_A * step - cEarth)- WORLD_RADIUS;
  dAP += exp(-vec2(hPoint, hPoint) / vec2(H_R, H_M))* diferential_h;
}
```

Se incrementa la acumulación de densidad hasta el punto y se calcula la extinción del color.

```
dPC += pRM * ds;
vec2 dAPC = dAP + dPC;

vec3 tR = dAPC.x * betaER;
vec3 tM = dAPC.y * betaEM;
vec3 tRM = tR + tM;

vec3 extRM = exp(-(dAPC.x * betaER + dAPC.y * betaEM));

vec3 difLR = pRM.x * betaSR * extRM * ds;
vec3 difLM = pRM.y * betaSM * extRM * ds;
```

Se suman las densidades parciales al total y queda preparada la variable para el cómputo de visibilidad, aunque su uso no tendrá sentido hasta la última aproximación, sección “3.4.2. Cálculo de visibilidad”.

```
float visi = 1;

rayLeigh += difLR * visi;
mieScattering += difLM * visi;

} // FIN FOR
```

Una vez terminado el bucle, las integrales quedan computadas y se aplican las funciones fase de Mie y de Rayleigh, para ello se necesita el coseno de Θ.

```
float cosTheta = dot(normalize(cam-obj), normLightDir);
float cos2ThetaP1 = 1 + (cosTheta * cosTheta);

float phase_rayLeigh = 3/16pi * cos2ThetaP1;
float phase_mieScattering = 3/8pi * (((1.0f - G2) * cos2ThetaP1) /
((2.0f + G2) * pow(1.0f + G2 - 2.0f * G*cosTheta, 1.5f)));
```

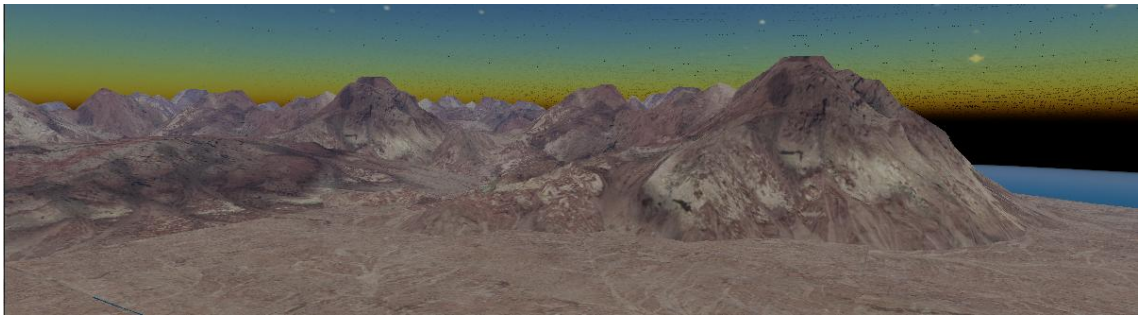
Solo queda aplicar las ecuaciones de *in-scattering* y *out-scattering* y obtener el color final.

```
vec3 inScattering = (rayLeigh * phase_rayLeigh + mieScattering *
phase_mieScattering) * lightSun;
vec3 extintion = exp(-(dPC.x * betaER + dPC.y * betaEM));

vec3 L0_Ext = texture(texture_diffuse, vs_fs_color.xy) * extintion;
color = vec4(1.0f - exp(-1.0f * (L0_Ext + inScattering) ), 1);
```

3.2.3. Resultados

Con el código del *fragment shader* definido en el apartado anterior, el rendimiento del programa es demasiado bajo (a una resolución de 1280x720 píxels obtenemos 26 *frames per second,* en adelante FPS) con una aproximación integral muy pobre, además el resultado no es el esperado. Se puede apreciar ruido en la imagen y la línea naranja del horizonte se encuentra desplazada hacia arriba y, como consecuencia, se aprecia una franja negra.



Primera aproximación con el sol en su cénit. N_STEPS = 30 (26 FPS)

La parte positiva que encontramos en esta aproximación es que en los puntos a media distancia, como son las montañas más alejadas, se puede apreciar un blanco azulado, siendo este uno de los resultados deseados. Además, el cielo empieza a mostrar su azul característico y el mar, en el horizonte, aparece con ligeros toques blanquecinos.

3.3. Segunda aproximación

El objetivo prioritario en esta fase es externalizar la integral que calcula la profundidad óptica desde la entrada de la luz en la atmósfera hasta los puntos intermedios, haciendo uso de la función de Chapman. La principal ventaja de esta función es que solo depende de dos parámetros y por lo tanto es generalizable.

3.3.1. Función de Chapman

La función de Chapman depende únicamente de la altura a la que se encuentra el punto y del ángulo Φ formado por el sol y la normal de la Tierra que pasa por dicho punto (ver esquema de la sección "2.1. Modelo físico del aire").

La forma de llevarlo a cabo es generar una textura para cada tipo de dispersión y utilizarlas como tablas de consulta en las que las filas y las columnas se corresponden con la altura y el coseno de Φ respectivamente. A continuación, rellenamos cada píxel de las texturas con la profundidad óptica correspondiente al punto determinado por esas dos variables. Finalmente, se dispone de dos tablas precalculadas con la información de la densidad entre el borde exterior de la atmósfera y cualquier otro punto dentro de ella, para los dos tipos de dispersión.

OpenGL nos da la gran ventaja de poder guardar estos cálculos en una textura. Uno de los múltiples modificadores que permite asociar OpenGL a una textura es el modificador GL_LINEAR. Este, en concreto, hace uso de un filtro bilineal cuando se accede a la zona comprendida entre dos píxeles. Aprovechando esta herramienta que el entorno de OpenGL facilita, logramos mejorar de manera notable la precisión de los valores, pudiendo así prescindir del uso de tablas de consulta que son más extensas y costosas de calcular.

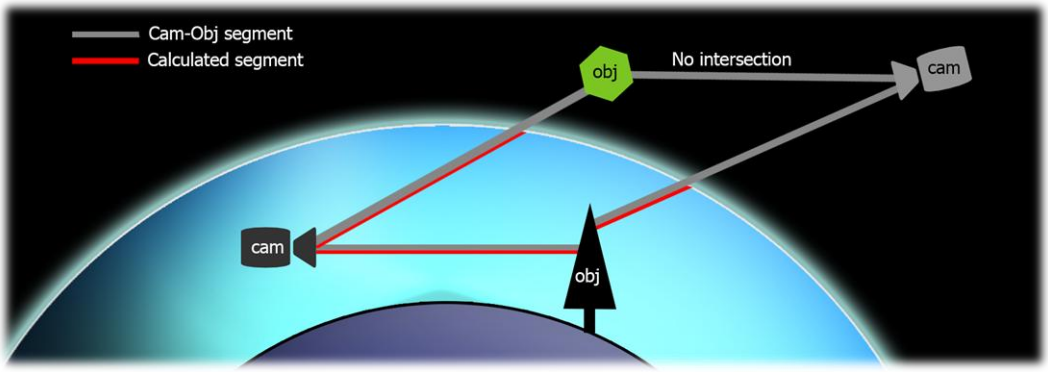
3.3.2. Cálculo de intersección con la atmósfera

Hasta ahora, la integral se calculaba hasta un punto visible. En muchos casos, este punto correspondía a la esfera que contiene al resto de la escena y que representa el universo, lo que daba como resultado algo poco eficiente, ya que una parte de los puntos intermedios del cálculo de la integral se encontraban fuera de la atmósfera, es

decir, se obtenían valores nulos por carecer estos de densidad (ver esquema siguiente).

El segundo objetivo ha sido, por lo tanto, crear una función que calcule el segmento que transcurre dentro de la atmósfera y medir la profundidad óptica de dicho segmento. Esta función se ha de aplicar tanto dentro del *shader* como en el código que generan las tablas precomputadas.

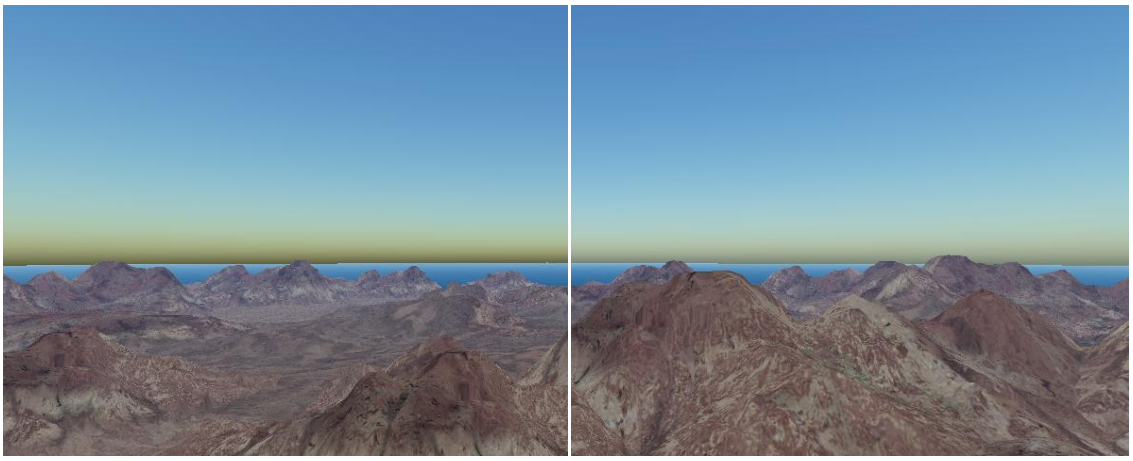
El cálculo de la intersección a partir de dos puntos, la cámara y el punto visualizado, indica si se produce una intersección con la atmósfera y genera dos puntos que delimitan el segmento sobre el que se calcula la profundidad óptica. Por ejemplo, cuando la cámara se encuentra dentro de la atmósfera, si mira hacia el universo se devolverá el punto de intersección con la atmósfera y el de la propia cámara, en cambio, si mira hacia un objeto en la tierra, se devolverán los puntos originales.



Esquema del segmento calculado dentro de la atmósfera

3.3.3. Resultados

Haciendo estos dos grandes cambios y ajustando algunos valores constantes, el resultado que hemos obtenido ha mejorado y ya es aceptable. El rendimiento mejora hasta un 700% (182 FPS) con respecto a la solución anterior, gracias a las tablas precomputadas. Como dato a tener en cuenta, ahora, al hacer el cálculo durante el inicio de la ejecución, obtenmos una mayor precisión tan solo aumentando el número de iteraciones y sin aumentar el coste durante el resto de la ejecución. Si bajamos el número de iteraciones de la primera aproximación a 20, la diferencia de rendimiento solo mejora hasta un 280%.



Segunda aproximación con el sol en su cénit

N_STEPS = 20 (240 FPS) a la izquierda, N_STEPS = 30 (182 FPS) a la derecha

3.4. Aproximación final

En esta última fase se pretende conseguir el efecto conocido como *light shaft*, que dibuja líneas de luz alrededor de los objetos. Este efecto, en realidad, es producido porque la dispersión no afecta a las partículas que se encuentran en sombra, por lo tanto hay que encontrar las zonas que están en sombra.

Después de investigar acerca de las diferentes maneras que existen de calcular sombras, se ha procedido a usar una de las formas más comunes por sencillez y rendimiento.

3.4.1. Mapeo de sombras

El método consiste en generar una imagen de la escena desde el punto de vista del sol sobre la zona en la que se vayan a calcular las sombras y, extrayendo únicamente la profundidad de los objetos, enviando la información obtenida al *scattering shader* que lo utiliza para conocer la visibilidad de los puntos.

El tipo de proyección utilizada debe ir asociada al tipo de luz que se emite. En este caso, al ser una luz global, los rayos de luz son paralelos y por lo tanto la proyección empleada es ortogonal.

Si simplemente se procesara la imagen desde el sol, se dibujaría sobre el *buffer* por defecto, es decir, en la ventana. Para evitar esto, hay que asociar a la tubería gráfica un nuevo *buffer* de “renderizado” que consiste en una textura en la tarjeta gráfica. Esta

textura se asocia, como Sampler2D, al *scattering shader* para que pueda acceder directamente a la información que contiene.

3.4.2. Cálculo de visibilidad

La manera de calcular si un punto se encuentra en sombra es aplicarle la matriz de transformación que se ha utilizado para crear la proyección ortogonal a la hora de generar la textura. Este proceso devuelve las coordenadas X e Y correspondientes a su lugar en la textura y una tercera coordenada Z, también llamada de profundidad, que se utilizará para comprobar si se encuentra en sombra. Por ser nuestra textura un *buffer* de profundidad, al acceder a las coordenadas (X, Y) proporcionadas por la transformación, nos devuelve la profundidad almacenada en ese píxel. Una vez obtenidas las dos profundidades, comprobamos si el punto se encuentra a más profundidad que el almacenado en la textura y, si se cumple, significa que dicho punto está en sombra; en caso contrario, está iluminado.

La comprobación de la visibilidad afecta al cálculo de la profundidad óptica y provoca un efecto llamado *light shaft* o *God rays* (rayos de Dios). Este efecto se aprecia cuando se interpone un objeto en la trayectoria del sol, es entonces cuando la sombra provoca que se resalten unos haces de luz.

Tres ejemplos en los que se aprecian los "rayos de Dios"

3.4.3. Proyección de sombras

Uno de los principales problemas de la proyección de sombras es la resolución del mapa de sombras utilizado y las limitaciones de las tarjetas gráficas a la hora de crear texturas de gran resolución. Cuando proyectamos una sombra sobre el escenario, usando el método del cálculo de visibilidad que acabamos de describir, si el espacio abarcado supera notablemente el tamaño de la textura, la sombra se dibuja

escalonada, debido a que un píxel de la textura corresponde a muchos en el puerto de vista.

Existen diferentes alternativas para mejorar la proyección de las sombras en grandes escenarios de exterior (ver [9] [10] [11] [12] [13] [14]). Estas alternativas se pueden dividir en dos grupos: por un lado, las que mejoran la captación del mapa de sombras, en ocasiones utilizando múltiples texturas de manera simultánea; y por otro, las que mejoran el procesamiento de la información contenida en el mapa de sombras.

3.4.3.1. Mejora en la captación del mapa de sombras

La primera alternativa no aumenta el tamaño del mapa de sombras, sino que captura solo la parte que se encuentra delante de la cámara. Las principales ventajas son que el tamaño del mapa de sombras no aumenta y que basta con una única textura para toda la escena. Las desventajas son que es necesario calcular la textura en cada fotograma, con el coste añadido que eso conlleva, y que las sombras vibran con el movimiento de la cámara; aun así, se mantiene un pequeño escalonado.

La segunda alternativa es aplicar un desenfoque “gaussiano” a la textura resultante, lo que requiere un post-procesado de la imagen.

La tercera consiste en utilizar múltiples mapas de sombras sobre la escena; se podría considerar equivalente a crear una única gran textura. La utilidad de este método reside en salvar la limitación del tamaño de textura. El principal inconveniente es el número de veces que se “renderiza” la escena y la memoria ocupada en la tarjeta gráfica, aunque se puede mejorar procesando solo los mapas cuando se necesite.

La cuarta y última alternativa contemplada (obtenida de [10]) es utilizar múltiples texturas que se ajusten al campo de visión y ordenarlas por proximidad, de tal manera que las texturas cercanas tengan mejor definición que las lejanas. Este tipo de mapas son los más utilizados en juegos con visión en primera o tercera persona. Su principal virtud es que se consigue definición de sombras donde importa, es decir, en los objetos cercanos, dejando la lejanía con menor o ninguna definición de sombras. Además, basta con pocos mapas, lo que mejora la alternativa anterior. El principal defecto es que el salto de una textura a la siguiente genera una irregularidad en la

sombra, pero esto se puede disimular superponiendo las texturas y mejorando el procesado de la proyección.

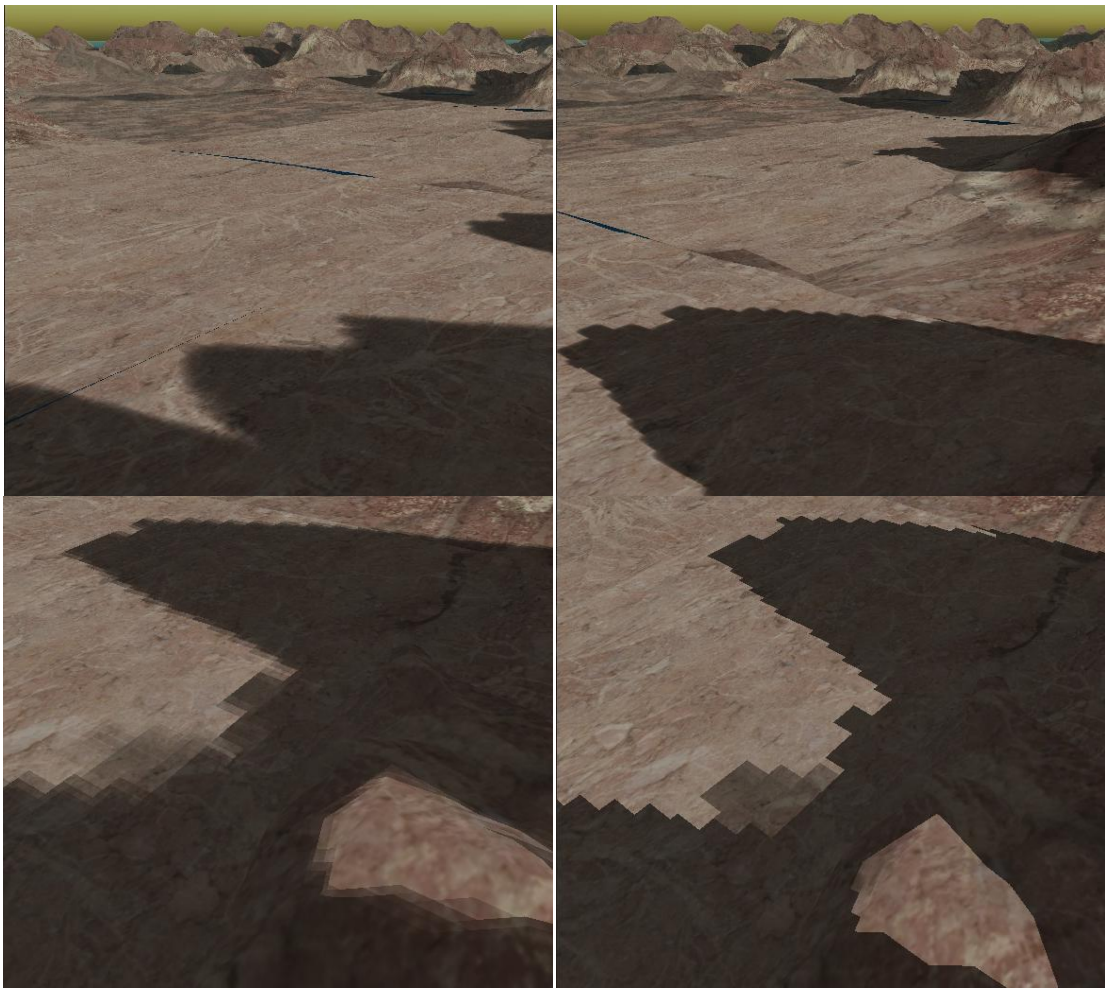
Las primeras pruebas de sombras se realizaron utilizando la primera alternativa, después de ver los resultados, se decidió probar otras opciones. Se implementó el desenfoque "gaussiano" aunque no proporcionó los resultados esperados, probablemente por el tamaño del escenario. Finalmente, se tuvieron que descartar las alternativas aquí presentadas y se decidió mantener una textura estática que solo se actualizara al cambiar la dirección de la luz y, para obtener mejores sombras, recurrir a un método de mejora del procesado de la proyección.

3.4.3.2. Mejora en el procesado del mapa de sombras

La primera opción, explicadas también en [11], [9] y [14], consiste en almacenar la profundidad de cada píxel y su cuadrado, para luego pasar por un filtro "gaussiano" a ambas texturas y, finalmente, elevar al cuadrado el primer valor. La ventaja es que se obtiene un resultado óptimo en tiempo real; el inconveniente es la necesidad de tener que hacer un post-procesado.

La segunda opción, encontrada en [12], consiste en aplicar un filtro que determina si los puntos adyacentes quedan en sombra o no. Cabe la posibilidad de utilizar infinidad de matrices de filtrado diferentes. Su mayor virtud es que producen un resultado realista. El principal defecto de este método es el coste del cálculo del filtro por cada píxel.

Finalmente se ha optado por esta segunda opción aplicando el filtro "gaussiano", por ser este el que mejores resultados da. Habiendo probado los dos tipos de acceso al mapa de sombras (el píxel más cercano y la interpolación bilineal), se aprecia que se han obtenido los mejores resultados al usar el acceso bilineal.



Arriba interpolación bilineal, abajo acceso al más cercano

Izquierda aplicando desenfoque gaussiano, derecha sin filtro

3.4.4. Sombreado mediante *Phong shading* y mapa de normales

Phong shading es una técnica utilizada para simular la iluminación en objetos tridimensionales (fuentes: [15], [16] y [17]). El color final de un píxel viene determinado, además de por su propio color, por varios factores como son la reflexión ambiente y las componentes difusa y especular de las luces y materiales.

El vector normal se utiliza para calcular el ángulo de incidencia de la luz sobre el vértice; si el vector normal no es unitario, el vértice estará sobreiluminado o infrailuminado. Es importante que, para cada vértice, se defina la normal, ya que esta no se asigna de manera automática por OpenGL.

3.4.4.1. Reflexión ambiente

La reflexión ambiente es la forma en que el material refleja la luz, independientemente de la dirección por la que esta venga. Con esta técnica se simula la luz que recibe un objeto procedente de la luz reflejada por el resto de objetos. Para ello, se añade una pequeña cantidad de luz a la hora de calcular el color de un píxel, a fin de evitar que aparezca completamente negro y otorgando así un toque realista.

La reflexión ambiente clásica añade una cantidad de color al modelo, habitualmente se usan colores pálidos, como amarillos pastel o azul cielo, o simplemente blanco. El problema de este tipo de iluminación es que, si se aplica mucha iluminación ambiental, el escenario parece tener una niebla constante. En este trabajo se ha cambiado la forma de calcular la reflexión ambiente de manera que se multiplica el factor ambiente por el color de la textura, por lo que la iluminación está basada en el color real y en el de la luz ambiente.

3.4.4.2. Reflexión difusa

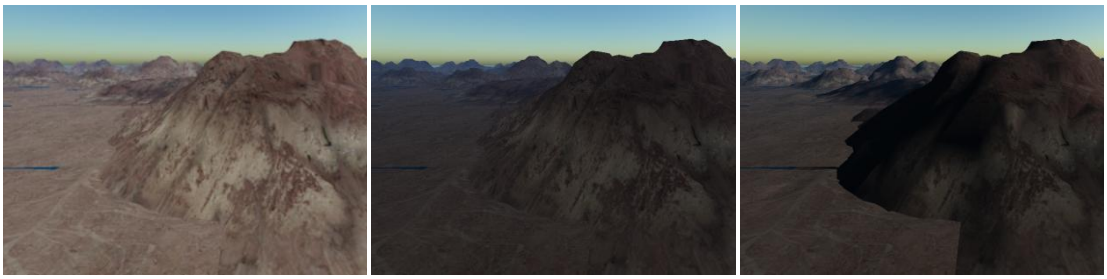
La reflexión difusa determina la cantidad de luz que refleja un píxel según la dirección de incidencia de la luz. Por lo tanto, el ángulo formado por la luz y la normal de un píxel influyen en lo iluminado que está. Así, si la luz incide perpendicularmente en la superficie, el píxel estará más iluminado que si lo hace de manera oblicua.

3.4.4.3. Reflexión especular

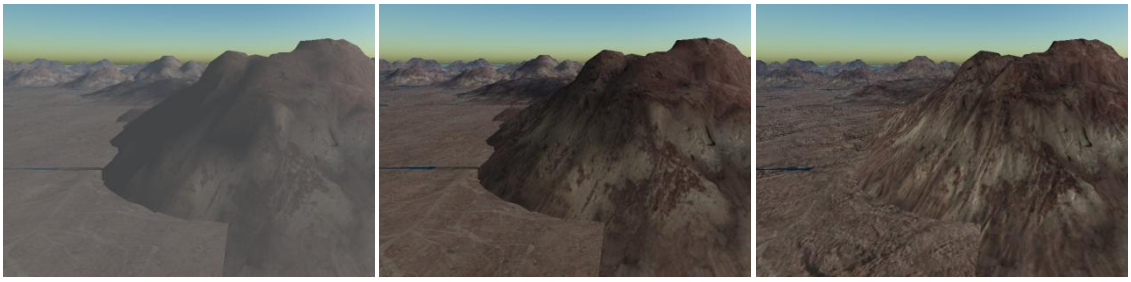
La reflexión especular define la luz emitida en una dirección que rebota según la normal del fragmento y provoca un efecto brillo en el objeto. Con este efecto se simula la luz especular que captamos cuando esta se refleja en nuestra dirección. Para ello, se calcula el ángulo entre la dirección de la luz reflejada y la dirección de la cámara. Cuando el ángulo formado es muy pequeño, se añade cierta cantidad de luz al píxel.

3.4.4.4. Mapa de normales

Hay dos formas de asignar la normal a un vértice: calculando su normal a partir de los vértices que forman la cara, u obteniéndola de un mapa de normales, como si de la textura de un material se tratase. En este trabajo se ha utilizado la segunda opción, por tener acceso a dicho mapa de normales.



Izquierda: color de textura. Centro: reflexión ambiente. Derecha: reflexión difusa



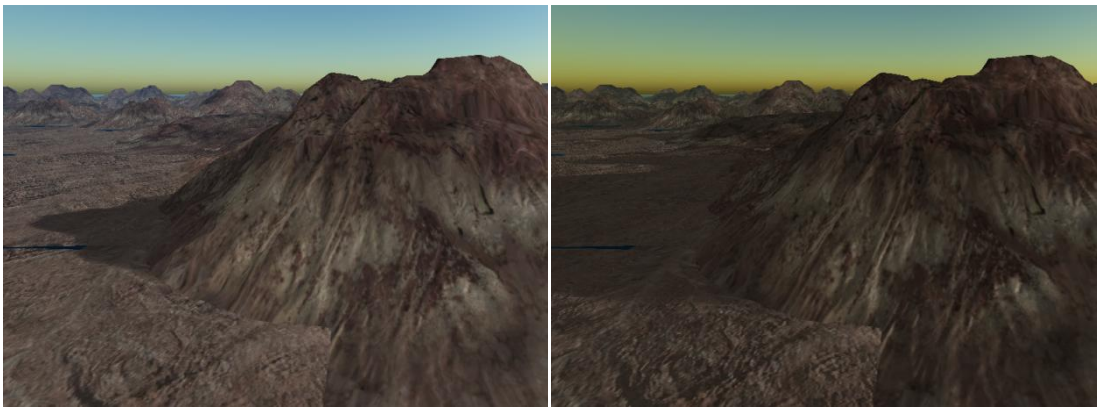
Izquierda: reflexión ambiente clásica más reflexión difusa

Centro: reflexión ambiente por textura más reflexión difusa

Derecha: reflexión completa con mapa de normales

3.4.5. Integración con la proyección de sombras

A la hora de proyectar las sombras en el suelo, se pueden elegir diferentes intensidades, desde un negro sólido hasta un gris muy tenue. Con la intención de que las sombras proyectadas se integren visualmente con el *Phong shading*, la iluminación difusa se calcula eligiendo el mínimo, es decir, el más oscuro entre el cálculo de la sombra y el valor difuso original.



Phong shading *y proyección de sombras con dos inclinaciones diferentes de la luz*

4. Arquitectura *software*

Dentro de la arquitectura del programa, se van a contemplar dos aspectos: la estructura general de ejecución y la jerarquía de clases.

4.1. Secuencia de la ejecución en CPU

La estructura general de ejecución está basada en un bucle de juego simple:

1. Inicialización de recursos.
2. GameLoop o bucle de juego que se divide en tres partes:
    a. Control de Eventos.
    b. Procesamiento del modelo.
    c. "Renderización".
3. Liberación de recursos.
4. Finalización del programa.

El bucle de juego se repite hasta que la lógica lo interrumpa, bien sea por un error o por petición del usuario. Aunque existen otros bucles de juego que permiten lógicas más complejas, usarlos complicaría innecesariamente el código debido a la simplicidad de la lógica utilizada.

4.1.1. Inicialización de recursos

En la inicialización se realiza la configuración de SDL y OpenGL que, entre otras cosas, invoca la ventana en la que se va a dibujar. Además, aquí es donde se decide si se va a activar la sincronización vertical de la GPU, que se encarga de limitar la cantidad de imágenes por segundo que se van a producir. En este trabajo se ha desactivado para poder apreciar las mejoras de rendimiento entre diferentes modelos.

Después, se cargan los objetos que se encuentran en formato "Wavefront .obj". La carga del objeto también ha sido implementada, siguiendo el formato que aparece en [18], y se ha pensado de manera que sea fácilmente escalable si se desean introducir nuevos datos en el archivo. La estructura generada puede ser utilizada para el almacenamiento y el procesamiento de manera muy sencilla, ya que agrupa los componentes en elementos típicos del "renderizado", como son los *buffer* de vértices o los materiales.

En este momento, también se cargan y compilan los *shaders* y se genera la información extra necesaria para su correcto funcionamiento. El elemento más importante que se produce aquí es la textura con las densidades, que va a servir como tabla precomputada.

El tratamiento de imágenes no está soportado nativamente por las principales librerías de OpenGL, por lo tanto se utiliza una librería complementaria de SDL llamada SDL_Image que facilita la carga y la modificación de las imágenes desde diferentes formatos, como jpg, bmp o png.

4.1.2. Control de eventos

El control de eventos se ocupa de interpretar y delegar los controles del programa. El controlador de la cámara obtiene la entrada del ratón y del teclado, encargándose tanto de desplazar como de girar la cámara. Otras funciones también controladas son la salida manual del programa, la posición del sol o la captura del ratón.

4.1.3. Procesamiento del modelo

El procesamiento del modelo realiza los cambios pertinentes en el modelo según lo requiera la lógica. Por ejemplo, al presionar la tecla “w” el control de eventos establece el estado “desplazándose hacia delante”, pero es en el procesamiento donde, en función del tiempo que ha pasado, se modifica el valor de la posición.

4.1.4. “Renderizado”

El “renderizado”, habitualmente, recorre la escena y dibuja cada modelo utilizando su *shader* asociado. En este trabajo, además, se realiza un “renderizado” previo para generar el mapa de sombras, cuando se considere necesario, ignorando la esfera que representa el universo. La fase de “renderizado” es donde reside la mayor carga de la aplicación y es la que limita el rendimiento de la simulación.

4.1.5. Liberación de recursos y finalización del programa

La liberación de recursos se encarga de vaciar toda la memoria, tanto de GPU como de CPU, que ha sido ocupada en la fase de inicialización. Por último, durante la finalización del programa, se cierra la ventana invocada y se permite al programa terminar la ejecución correctamente.

4.2. Secuencia de la ejecución en GPU

En la GPU se configuran dos tuberías de ejecución utilizando un *vertex shader* y un *fragment shader* para cada una de ellas.

4.2.1. Tubería del mapa de sombras

De las dos tuberías, la tubería del mapa de sombras es la primera en ejecutarse y solamente se ejecuta al lanzar el programa y cuando se modifica la posición del sol. Se encarga de generar el mapa de sombras, como ya se ha explicado en el punto "3.4.1. Mapeo de sombras".

Tanto el *vertex* como el *fragment shader* son muy simples. El *vertex shader* recibe la posición del vértice que se va a procesar y la matriz MVP, que sitúa la cámara mirando en la misma dirección que la luz y devuelve la posición del vértice transformada, como ya se ha explicado anteriormente. El *fragment shader* recibe la posición del fragmento y devuelve la profundidad en Z de este, que será utilizada para el cómputo de las sombras.

4.2.2. Tubería de la dispersión atmosférica

La tubería de la dispersión atmosférica se ejecuta una vez por iteración del bucle, como se explicó en el punto "4.1 Secuencia de la ejecución en CPU". Si las dos tuberías se ejecutan en la misma iteración, la tubería del mapa de sombras se ejecuta primero debido a que prepara información para la tubería de dispersión. La función de esta tubería es la más importante de todo el programa ya que contiene el *fragment shader*, que aplica todas las ecuaciones del modelo físico de la atmósfera y calcula el color para cada fragmento.

El *vertex shader* es bastante sencillo, simplemente se encarga de aplicar la matriz MVP y devolver la posición transformada y sin transformar, la normal de los vértices y la posición de las texturas para que el *fragment shader* pueda operar correctamente.

El *fragment shader* aplica la dispersión y es el último paso que da cada iteración de dibujado. Recibe todas las constantes necesarias para calcular las ecuaciones de dispersión y los *sampler* de texturas. Además, toma del *vertex shader* las coordenadas de texturas, las normales de los vértices y las posiciones.

4.3. Jerarquía de clases

La clase `Engine` es donde empieza la ejecución. Es la clase que se encarga de inicializar la escena y los objetos necesarios para la ejecución, además de mantener el bucle de juego. Contiene un `Info_Manager`, una `CameraFPS` y una `ScatteringScene`.

`CameraFPS` es la clase que se encarga del manejo de la cámara. Los controles están inspirados en la mayoría de juegos en primera persona que sitúan la cámara en la cabeza del personaje manejado.

El `Info_Manager` lleva información sobre los *frames* por segundo y puede obtener información sobre la tarjeta gráfica.

`Scene` es una clase que está pensada para heredar de ella. Contiene una lista de mallas y unas funciones que facilitan realizar ciertas operaciones, como las referidas a la limpieza, la inicialización y el dibujado, llamando a esa misma función en todas las mallas. Además, tiene un puntero a un *shader* activo que utiliza a la hora de dibujar y antes de llamar a las diferentes mallas con su matriz de modelado.

`ScatteringScene` hereda de `Scene` y genera todas las mallas que van a formar parte del escenario de ejecución durante la inicialización. Para ello, utiliza las clases auxiliares `ObjLoader`, que se encarga de interpretar el archivo .obj junto con los archivos .mtl que se requieran, y `ObjToMesh`, que reestructura los datos para facilitar su uso durante el "renderizado". Contiene todos los *shaders* que se van a utilizar durante la ejecución de esa escena, de tal manera que asocia al *shader* activo aquel que se vaya a utilizar en el siguiente "renderizado". Aquí es donde se define la información de las constantes que van a ser utilizadas en el "renderizado", como el radio de la Tierra, la dirección y luminosidad del sol o las β utilizadas en la función fase.

`Mesh` es la clase que guarda toda la información sobre las mallas, como son los vértices, las normales, las caras y su material asociado. Un material consta de una textura, un mapa de normales y factores para el sombreado Phong. Tiene un atributo "`visible`" que evita el "renderizado" de la malla. En este trabajo, esto se utiliza para

evitar el dibujado de la esfera que representa el universo cuando se calcula el mapa de sombras.

`Shader` guarda y maneja toda la información relacionada con OpenGL para simplificar el uso de los *shaders*. Los métodos más importantes son "`use`", que activa el *shader*, y "`load`", que dados los nombres de unos archivos, los lee y compila. También tiene otras funciones interesantes, como "`preDraw`", que se llama antes del dibujado y permite preparar cierta información, o como las funciones clásicas para establecer la matriz del modelo, la matriz de proyección y la cámara.

`ScatteringShader` se encarga de mandar a los diferentes *shaders* la información sobre las variables y sobre las constantes que se utilizan en las funciones de dispersión. Y contiene un `ShadowMapShader`, porque necesita conocer la matriz de transformación con la que se ha creado el mapa de sombras.

`ShadowMapShader` crea la matriz de transformación que permite situar la escena desde el punto de vista del sol, con proyección ortogonal.

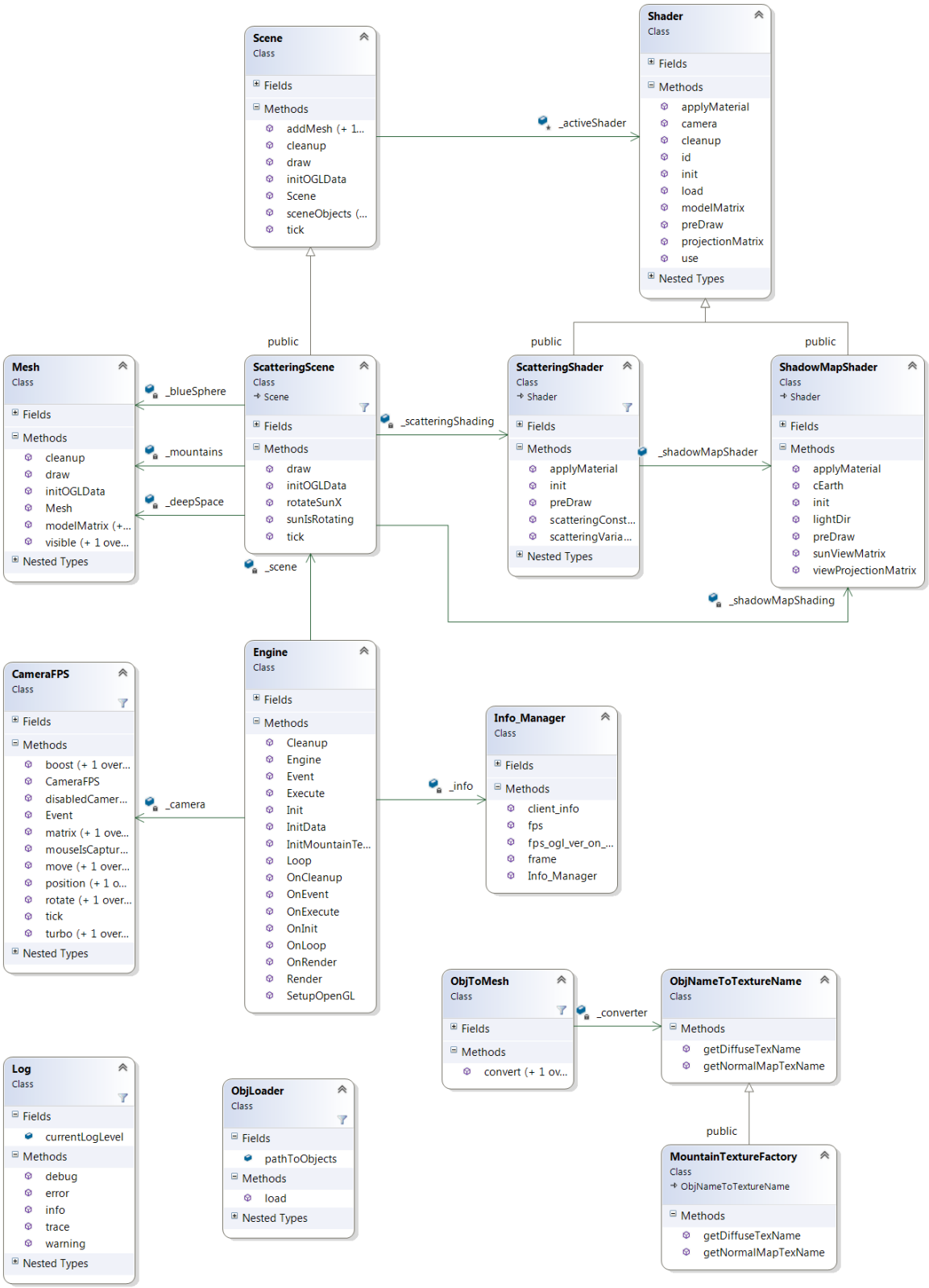


Diagrama de clases obtenido desde Visual Studio

5. Incidencias y soluciones durante el desarrollo

A lo largo del desarrollo del proyecto han ido surgiendo una serie de problemas para los cuales hemos tenido que buscar soluciones para superarlos. Algunos tenían un grado de dificultad casi trivial, pero en otros hemos llegado a perder bastante tiempo. En este apartado nos centraremos en los que más nos han costado.

En un proyecto, habitualmente, se empieza a trabajar directamente sobre la implementación de la idea sin necesidad de crear configuraciones complejas. El hecho de poner a funcionar el OpenGL moderno desde cero es una dificultad en sí misma. Se necesitan varias librerías específicas y una cantidad considerable de líneas de código para construir un esqueleto del proyecto que nos permita empezar a desarrollarlo.

Aunque el uso de archivos de mallas está bastante extendido en los grandes motores, existen ciertas dificultades para implementar su uso. Tuvimos problemas a la hora de utilizar librerías que leyeran este tipo de archivos y, después de varios intentos fallidos, decidimos implementar la carga de la malla basándonos en un esquema incompleto encontrado por la red [19].

Este proyecto, a su vez, tiene una gran parte de investigación previa, ya que requiere entender la evolución de la ecuación de dispersión y las diferentes maneras de calcularla. Esto nos llevó bastante tiempo, más de un mes, teniendo que volver a leerlo varias veces durante el desarrollo del proyecto, debido a la dificultad de las ecuaciones y del proceso. En una primera iteración, se intentó plasmar la fórmula en el código pensando que no sería necesaria su completa comprensión para implementarla. Esto nos creó problemas y no conseguíamos hacer que funcionara correctamente, por lo tanto, decidimos dedicarle más tiempo al estudio de las ecuaciones y su significado.

Otro de los problemas que encontramos fue tener que aplicar trigonometría en entornos tridimensionales, ya que habitualmente en las clases de matemáticas se suelen hacer ejemplos en dos dimensiones. Además, contábamos con la dificultad añadida de la depuración en GPU, de la cual hablaremos más adelante. La solución no fue otra que estudiar, investigar por Internet y simplificar a dos dimensiones todo lo que fuera posible.

Debido a la dificultad de implementar el bucle (ver "2.1.3. Resolviendo la ecuación"), intentamos realizar una primera aproximación simplificando las cuentas y haciendo que el bucle interno tuviera un valor constante por defecto. La intención era comprobar que las diferentes partes de la ecuación funcionaban correctamente por separado. No conseguimos hacer estas comprobaciones porque reducir el bucle a una constante provoca resultados incorrectos.

Al generar las texturas de precálculo, no sabíamos cómo comprobar si lo estábamos haciendo correctamente o si nos estábamos equivocando en el planteamiento de la ecuación. No sabíamos cuáles eran los valores aproximados que debían dar como resultado estas ecuaciones y tuvimos que estimar si los valores eran correctos. Una vez generada la textura y habiendo comprobado manualmente unos pocos valores mediante depuración, solo disponíamos de los colores de la textura (*floats* codificados en RGBA) para comprobar que los valores de los píxeles, como conjunto, mantenían cierta lógica. Después de considerar si tenían sentido, decidimos comprobarlo mediante su integración en el programa.

Existen varias codificaciones para texturas y, aunque la más extendida es RGBA, muchas imágenes utilizan otras como BGRA. Esto supuso un problema porque la librería utilizada para leer las texturas no proporcionaba la información necesaria para saber la codificación de cada imagen. Finalmente optamos por unificar la codificación de todas las imágenes.

Habitualmente depurar un programa en lenguajes como C++ o Java resulta una tarea tediosa, pero existen herramientas que facilitan el proceso y lo hacen más agradable. En OpenGL existen dos grandes problemas a la hora de depurar:

- El primero es que la librería OpenGL no lanza excepciones cuando una llamada es incoherente y la única información es una ventana en negro. La manera de obtener información de los errores es mediante la función "`glGetError`", que devuelve el código de error. El problema, nuevamente, es que el código suele tener varias llamadas a la función y se desconoce su línea. Para solucionarlo aprovechamos la potencia de C++ para crear una función llamada desde un "`#define`" que informaba de la línea y del fichero donde se había

producido el error e interpretaba el error escribiendo un mensaje más comprensible.

- El segundo de los inconvenientes es depurar el código GLSL que no está soportado nativamente por ninguna de las plataformas de programación más habituales. Después de investigar, encontramos el *plugin* de Visual Studio llamado Nvidia Nsight que permite ver el estado de la memoria de la tarjeta gráfica y con ello conocer las texturas, buffers y otros elementos almacenados. Otra de sus posibilidades es depurar la ejecución de un píxel, pero son necesarias dos tarjetas gráficas Nvidia en el mismo ordenador o dos ordenadores con tarjeta gráfica Nvidia conectados de manera remota. Además, la última generación de tarjetas gráficas no está soportada por el *plugin* y la depuración en remoto nos dio problemas y solo pudimos realizar unos segundos de depuración por ejecución.

Pese a todos los problemas que hubo depurando, gracias al *plugin* conseguimos encontrar algunos fallos y pudimos solventarlos. Cuando no podíamos acceder a dos ordenadores la depuración la hacíamos mediante códigos de color; por ejemplo, en el cálculo de la intersección del segmento vista-objeto con la atmósfera, mostrábamos negro o blanco según se intersecaba o no.

Uno de los problemas que solucionamos mediante la depuración con Nsight fue el de las constantes de GLSL. Los "`#define`" de C++ son constantes y la precisión es bastante alta. Al ser GLSL un lenguaje basado en C++ supusimos un comportamiento similar y definimos las constantes con directivas de preprocesado "`#define`". Nos sorprendimos bastante durante la depuración al ver que las constantes así definidas tenían una precisión muy baja. Como solución decidimos externalizar la definición de estas constantes y proporcionárselo al *shader* como si de una variable de entrada se tratara.

Al principio de la implementación del algoritmo de generación de sombras, hay que construir la matriz MVP en proyección ortogonal para que sitúe la cámara en la dirección de la luz. Para ello se suelen utilizar librerías de operaciones matriciales y vectoriales que facilitan la construcción de la matriz mediante funciones como

“`lookAt`”. En este caso utilizábamos la librería recomendada por [6] llamada “`vmath`”. Esta librería no generaba las matrices de transformación correctas para “`lookAt`” y “`ortho`”, y el proyecto se encontraba demasiado avanzado como para utilizar otra librería diferente, como la que recomienda actualmente OpenGL (GLM, OpenGL *Mathematics*). Decidimos crear la matriz y realizar las operaciones manualmente para conseguir resolver el problema.

Como ya ha sido mencionado en la sección “3.4 Aproximación final”, encontramos muchos problemas a la hora de obtener unas sombras con la calidad deseada. Invertimos bastante tiempo en probar las diferentes implementaciones hasta que dimos con la configuración que nos pareció óptima para este trabajo.

Por último, analizando la ejecución sobre la tarjeta gráfica, vimos que la cantidad de llamadas que se producían a la misma era muy elevada, por lo que decidimos reducir el número de objetos a dibujar, optimizando el archivo “Wavefront .obj” de las montañas, lo que rebajó considerablemente el número de llamadas (a un cuarto, aproximadamente) y mejoró de manera sensible el rendimiento. Esa optimización la llevamos a cabo mediante un algoritmo propio.

6. Resultados

El modelo utilizado para la realización de las pruebas consta de una malla de montañas, una esfera para la Tierra y otra esfera para simular el espacio exterior. También se han realizado pruebas sustituyendo las montañas por la malla de una ciudad sin obtener una gran diferencia en la tasa de imágenes por segundo.

Malla	**Número de triángulos**
Montañas	97 650
Tierra	327 680
Espacio	968
Ciudad	11 820

6.1. Rendimiento

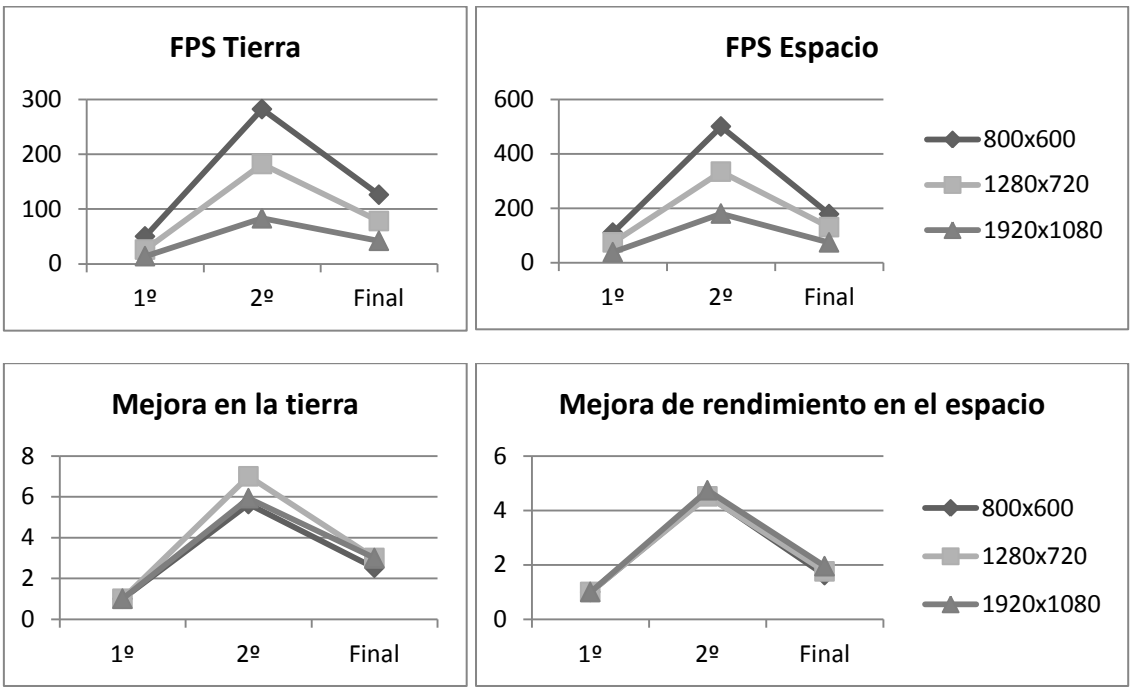
A continuación se muestra una tabla de rendimiento para las diferentes aproximaciones en tres resoluciones diferentes. El bucle para la aproximación integral realiza 30 iteraciones. El rendimiento se mide cerca de la superficie de la Tierra y desde el espacio exterior viendo solo una parte de ella. Se han utilizado dos configuraciones de ordenador y las montañas como escenario para realizar estas estadísticas.

Versión	**Resolución**	**FPS Configuración A**		**FPS Configuración B**	
		Tierra	**Espacio**	**Tierra**	**Espacio**
Primera aproximación	800x600	50	110	218	402
	1280x720	26	74	122	262
	1920x1080	14	38	60	128
Segunda aproximación	800x600	282	500	934	1349
	1280x720	182	334	542	862
	1920x1080	83	180	272	506
Aproximación final.	800x600	126	178	444	636
	1280x720	78	130	270	458
	1920x1080	42	74	136	246

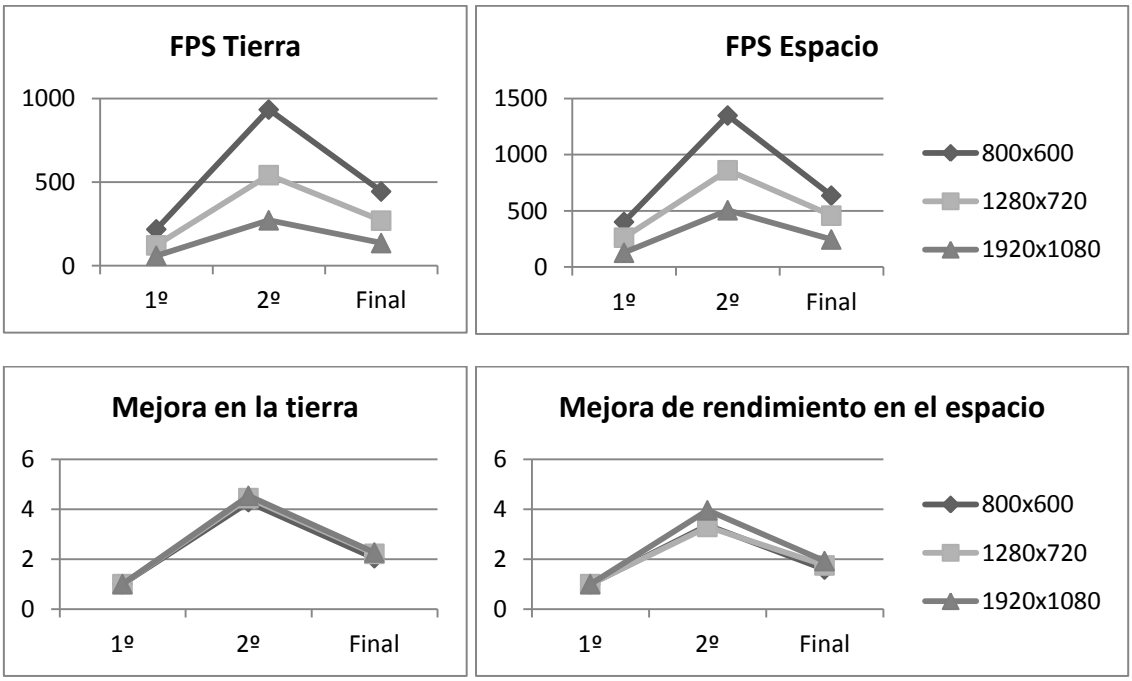
Las pruebas para la configuración A se han realizado con un equipo portátil que consta de una tarjeta gráfica Nvidia GTX850M, un procesador Intel i5-4210H y 8 GB de memoria RAM.

Las pruebas para la configuración B se han realizado con un equipo de sobremesa que consta de una tarjeta gráfica Nvidia GTX970, un procesador Intel i5-750 y 8 GB de memoria RAM. Ambas con Windows 7 Ultimate SP1 64bit como sistema operativo.

6.1.1. Gráficas de la configuración A



6.1.2. Gráficas de la configuración B



Las mejoras de rendimiento se calculan con respecto a la primera aproximación, de tal manera que se divide la aproximación elegida con la de referencia.

6.2. Análisis de resultados

La primera aproximación, sin la textura de la profundidad óptica precomputada, es claramente la más ineficiente de todas las configuraciones. Se puede apreciar en la segunda aproximación que el precómputo aumenta hasta siete veces el rendimiento de la ejecución. En la aproximación final hay una caída bastante fuerte de FPS respecto de la segunda pero, a cambio, la calidad de la imagen mejora notablemente cuando nos encontramos cerca de la superficie terrestre.

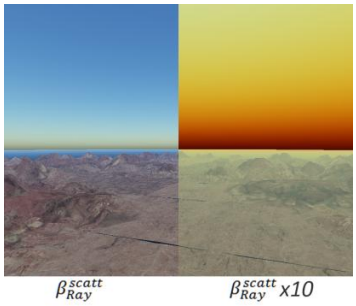
Ya que la aproximación final solo añade elementos que son interesantes en el interior de la atmósfera, si hubiera que usar este algoritmo únicamente para el "renderizado" de escenarios que se visualizan desde el espacio exterior, la mejor opción sería la segunda aproximación, que proporciona un rendimiento elevado con una calidad de imagen muy alta y no es necesario el sobre cálculo que provocan las sombras y los "rayos de Dios".

Podemos decir que los resultados para la aproximación final dan una tasa de imágenes por segundo por encima de los 30 FPS mínimos que se suelen exigir, dando en la mayoría de los casos más de 60 FPS. Además, después de todas las características añadidas para la versión final, se ha conseguido duplicar el rendimiento con respecto a la primera versión, dando como resultado una imagen fluida y de calidad.

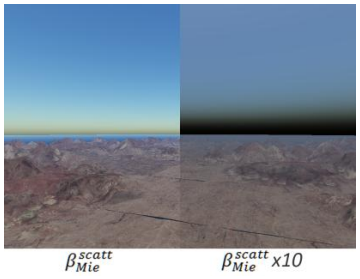
6.3. Constantes utilizadas

El valor de las constantes definidas para el cálculo de la dispersión se ha obtenido de trabajos previos (como [1], [3], [8] y [4]) y, en algunos casos, se han añadido ciertas modificaciones.

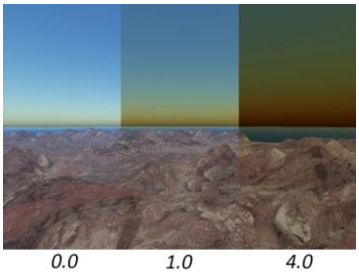
Constante:	β_{Ray}^{scatt}
Definición:	Coeficiente de Rayleigh que indica la cantidad de energía dispersada por unidad de longitud.
Uso:	Al aumentar los valores, el espectro de luz se desplaza hacia frecuencias más bajas.
Valor base:	R=5.8 * 10^{-6}, G= 13.5 * 10^{-6}, B=33.1 * 10^{-6}



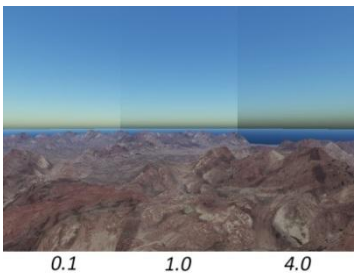
Constante:	β_{Mie}^{scatt}
Definición:	Coeficiente de Mie que indica la cantidad de energía dispersada por unidad de longitud.
Uso:	Al aumentar los valores, el cielo se vuelve hacia tonalidades grisáceas.
Valor base:	R=2.0 * 10^{-5} , G=2.0 * 10^{-5}, B=2.0 * 10^{-5}



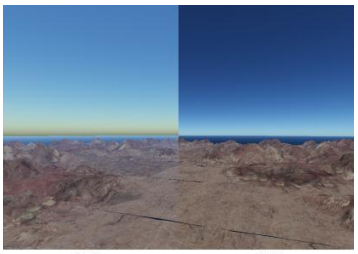
Constante:	β_{Ray}^{abs}
Definición:	Cantidad de energía perdida por la absorción de las moléculas.
Uso:	Al ser moléculas, absorben una cantidad de energía despreciable.
Valor base:	0.0



Constante:	β_{Mie}^{abs}
Definición:	Cantidad de energía perdida por la absorción de los aerosoles.
Uso:	Aumenta los efectos de la dispersión de Mie y guarda relación con ella.
Valor base:	$\beta_{Mie}^{scatt} * 0.1$

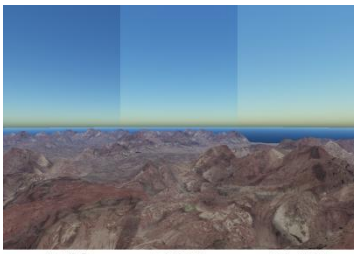


Constante:	P_0
Definición:	Factor de densidad al nivel del mar.
Uso:	Escala la densidad del aire y se refleja en Rayleigh y Mie.
Valor base:	[0.7, 1.3]



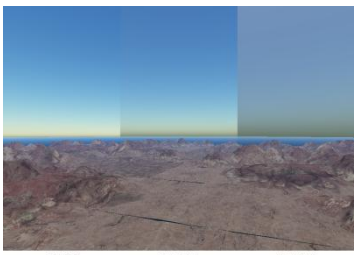
1.0 *0.3*

Constante:	H_R
Definición:	La altura de la atmosfera si la densidad para Rayleigh fuese constante.
Uso:	Modifica la densidad del aire para Rayleigh.
Valor base:	7994



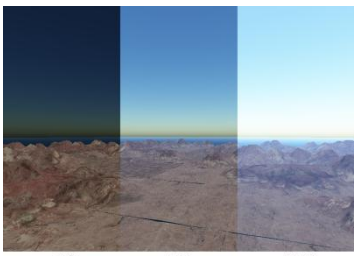
4 500 *7 994* *16 000*

Constante:	H_M
Definición:	La altura de la atmosfera si la densidad para Mie fuese constante.
Uso:	Modifica la densidad del aire para Mie.
Valor base:	1200



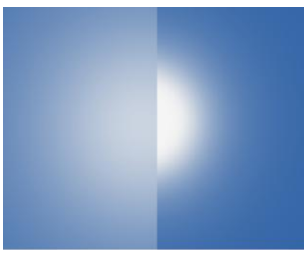
300 *1200* *7994*

Constante:	Intensidad del sol.
Definición:	Cantidad de luz emitida.
Uso:	Aumenta el brillo de los efectos de dispersión.
Valores:	[30, 100]



10 *50* *150*

Constante:	G
Definición:	Índice de simetría de la dispersión de Mie.
Uso:	Varía el halo de luz que se crea alrededor del sol.
Valores:	[0.70, 0.999]



0.75 *0.95*

7. Contribución Individual

7.1. Gregorio Iniesta Ovejero

El trabajo ha sido desarrollado utilizando la técnica de *pair programming,* como ya ha sido explicado en el apartado de "1.3.2 Metodologías usadas en el desarrollo". Debido a esto se hace difícil diferenciar cuál ha sido la aportación de cada uno al trabajo, por lo que a continuación voy a exponer las partes en las que más he estado implicado de manera individual y en las que hemos participado simultáneamente, siguiendo un orden cronológico.

Para empezar, es importante tener en cuenta los conocimientos que habían sido adquiridos antes del inicio del desarrollo del proyecto. Ambos habíamos cursado la asignatura de Informática Gráfica y gracias a eso el aprendizaje de OpenGL fue más rápido y, sobre todo, nos evitó tener que aprender a utilizar ciertos recursos matemáticos utilizados habitualmente en gráficos, como la matriz MVP. Personalmente, hace algún tiempo me dediqué a estudiar las bases de la programación moderna en OpenGL (versiones 4.X), lo que ha contribuido a agilizar el aprendizaje de técnicas avanzadas tanto para mi compañero como para mí.

La primera tarea consistió en encontrar documentos y trabajos previos que tratasen sobre la dispersión de la luz en la atmósfera. En esto hubo una participación equivalente, ya que cada uno se puso a buscar información tanto en Internet como en la biblioteca de la facultad. Al final de cada día nos juntábamos para mirar qué teníamos y seleccionar la información que nos era útil. Finalizada la fase de investigación, decidimos poner en orden todos los documentos y realizar un filtrado final antes de comenzar a trabajar.

El siguiente paso fue empezar a programar y para ello necesitábamos diseñar una arquitectura que nos permitiera hacer un proyecto basado en "renderizado" gráfico. Yo tenía cierta experiencia a la hora de desarrollar este tipo de aplicaciones, así que me encargué de construir la arquitectura a alto nivel. Desarrollé el bucle de juego, la medición de imágenes por segundo y creé la maquetación de los primeros enlaces con OpenGL y la inicialización de los *buffer* que iban a almacenar información en la GPU.

Después, me encargué de realizar varias refactorizaciones de tal manera que el código se mantuviera fragmentado y modular. Esto resultó especialmente útil en la ampliación que introdujo los mapas de sombras en el proyecto, debido a que solo hubo que modificar unas pocas líneas de código para poder acoplarlo.

Cuando estuvo montada la primera iteración de la arquitectura general, llegó el momento de aprender a utilizar los *shaders*. Cada uno fuimos encargándonos de aprender alguna funcionalidad concreta sobre los *shaders* para luego ponerla en común. Yo me encargué de aprender a texturizar las mallas y esto me obligó a conocer las diferentes maneras de pasar información a un *shader* y cómo se comparte la memoria entre ellos.

Una vez supimos utilizar los *shaders*, fue necesaria la carga de mallas de objeto desde archivo en tiempo de ejecución. Juntos estuvimos estudiando las mejores maneras de abordar esto y vimos que los tipos más recomendados para novatos con OpenGL eran FBX y "Wavefront .obj". Me encargué personalmente de realizar la carga de este tipo de archivos y, tras el fallido intento de carga de los ficheros FBX, me dediqué de lleno a los archivos "Wavefront .obj". No resultó difícil, pero sí bastante tedioso debido a la cantidad de formatos de línea que soporta y que, además, supone una carga adicional de los archivos de material con un formato similar.

El núcleo de nuestro trabajo es el *fragment shader* que se encarga de aplicar las principales ecuaciones sobre dispersión. Debido a la importancia que tiene este *shader*, durante su desarrollo estuvimos los dos integrantes del grupo alternándonos a la hora de escribir código, mientras el otro verificaba la validez del mismo. Esto promovió un código más limpio, con menos *bugs*, y agilizó la resolución de errores, lo cual es importante en un entorno que no contaba con grandes herramientas de depuración.

Otra parte decisiva del desarrollo fue la externalización explicada en el apartado "3.3.1. Función de Chapman". Aquí aporté mis conocimientos sobre la librería de lectura de imágenes que forma parte de SDL. Dado que ambos habíamos estado trabajando en el *fragment shader* simultáneamente, también hicimos juntos el trabajo de externalización. Esto fue especialmente útil ya que, debido a los pocos factores que

existían para verificar la validez del código, era necesario estar muy atento a los posibles errores que podían surgir en la traducción.

Por último, y para darle una mejor apariencia al escenario, decidimos añadir la iluminación clásica por píxel. Quise encargarme yo de esta tarea, que consistía en investigar la mejor manera de hacerla e implementarla. Finalmente me decidí por utilizar la técnica llamada Phong *shading*, debido a que ya conocía con anterioridad este tipo de iluminación, aunque no los detalles ni cómo implementarlo. Una vez realizada la primera iteración de este algoritmo, lo mejoré utilizando otra técnica llamada *Bump mapping*. Dado que me había encargado de la carga de mallas, me resultó sencillo modificar la carga de materiales para que admitiese la lectura de una textura adicional (el mapa de normales) e integrarla con el resto de la solución para que el *shader* pudiese utilizar la información de las normales.

7.2. Javier Pérez Martínez

La mayor parte del trabajo se ha realizado de manera conjunta, como se describe en el apartado “1.3.2. Metodologías usadas en el desarrollo”. Es por ello que resulta difícil la contribución al proyecto de manera individual sin repetir parte del contenido de mi compañero. Intentaré no extenderme en las partes hechas en común.

Antes de comenzar este trabajo de fin de grado, tenía ciertos conocimientos previos de OpenGL gracias a la asignatura de Informática Gráfica que cursamos durante el pasado año. Estos conocimientos fueron útiles porque ya conocíamos el funcionamiento y la aplicación de las matrices de transformación dentro de una escena. Otra parte de los conocimientos resultó de menor utilidad al haber sido adquiridos con una versión de OpenGL más antigua que la que decidimos utilizar para la realización del proyecto, con lo tuvimos que aprender a utilizar las nuevas técnicas de programación gráfica incluidas en la versión actual de OpenGL.

Una parte importante de este trabajo recae en la investigación del comportamiento de la luz en la atmósfera así como en la comprensión del funcionamiento de las fórmulas físicas que simulan este efecto. Para ello, recopilamos abundante material de carácter

científico que tuvimos que asimilar, a fin de lograr que el objetivo del proyecto fuese realista.

Para poder aplicar las fórmulas que describen el comportamiento de la luz y obtener buenos resultados, era necesario aprender a utilizar los *shaders.* Aunque la sintaxis de GLSL me es familiar, ya que es similar a la de C, tuve que estudiar las particularidades de este lenguaje, sobre todo lo referido a la recepción de variables y su salida. Además, la manera de enviar las variables al propio *shade*r, así como su carga, compilación y activación, son tareas que resultan complejas.

Dentro del uso de los *shaders* en OpenGL, yo me centré en cómo se pueden aplicar diferentes *shaders* a distintos objetos a lo largo de la ejecución y de una manera óptima. Para ello tuve que buscar información sobre cómo se activan y desactivan los *shaders*, previamente cargados en la unidad de *hardware* gráfico. Así mismo tuve que realizar diferentes pruebas de funcionamiento.

Tras el estudio previo, teníamos que plasmar y adaptar las fórmulas estudiadas en el código del *fragment shader*. Esto resultó ser una tarea costosa debido a la complejidad de las propias fórmulas y al uso de una librería gráfica con la que aún no estábamos familiarizados.

El cálculo de la profundidad óptica requiere de varias operaciones trigonométricas. Como fui capaz de solucionar sin demasiada dificultad el primer problema de esta índole que se nos presentó, a partir de ese momento me encargué de desarrollar las partes que requerían el uso de ecuaciones trigonométricas de cierta complejidad.

La función de Chapman permite generalizar el cálculo de la densidad parcial de los puntos y externalizar el bucle interno del cálculo integral. Para su implementación, tuvimos que recurrir al uso de texturas donde guardar el valor de la densidad según el tipo de partícula. En esta parte investigamos cómo se guardan valores de punto flotante en una textura para su posterior consulta en el *fragment shader*. La solución que encontramos fue multiplicar cada canal de color por un valor, en función del *byte* que ocupe.

Uno de los objetivos que nos marcamos desde el principio del proyecto fue el de obtener un cálculo de sombras realista. Yo me encargué de estudiar cómo se calculan las sombras y de las diferentes alternativas que hay para obtener un mejor resultado. En primer lugar, había que investigar una manera óptima de calcular las sombras y, tras comprobar que la forma más extendida de hacerlo era mediante el mapeo de sombras, pasé a investigar la manera de implementarlo en el proyecto. Primero tuve que crear un *shader* específico para la captación de sombras y guardar en el punto su posición en Z. Después había que integrar este *shader* con el resto del proyecto, ya que pintaba directamente sobre la salida de la ventana. Para ello recurrí al uso de un *frame buffer*, que es un tipo de *buffer* que se enlaza a la tubería para que redirija el pintado al *frame buffer*, en lugar de hacerlo sobre el *buffer* que corresponde a la ventana.

Una vez elegida la forma de captar las sombras, había que decidir entre las diferentes alternativas para mejorar su cálculo. Esta labor fue más complicada ya que hay muchas formas diferentes de hacerlo y, en principio, todas parecen obtener buenos resultados. Las alternativas que fui investigando se detallan en la sección "3.4.3. Proyección de sombras". No todas, pero sí gran parte de ellas, se llegaron a implementar para comprobar si los resultados eran lo bastante buenos, hasta que, finalmente, por rendimiento y calidad de las sombras, nos decantamos por utilizar un mapa de sombras sencillo y aplicar un filtro durante el cálculo de la visibilidad.

8. Conclusiones

La aplicación de modelos físicos en las simulaciones es imprescindible para proporcionar realismo a las imágenes generadas. Dentro de las simulaciones de modelos físicos, el de la dispersión de la luz en la atmósfera es esencial para ofrecer un entorno abierto o un modelo espacial que resulte realista.

Sin embargo, este tipo de modelos hacen uso de algoritmos que requieren cálculos que no son fáciles de procesar por los ordenadores y esto conlleva un coste demasiado alto, con su consecuente pérdida de rendimiento. Para aplicar este tipo de algoritmos en programas que requieran suficiente fluidez de imágenes, hay que recurrir a aproximaciones que ofrezcan resultados similares, sacrificando una parte de la calidad final.

Durante el desarrollo de los prototipos, encontramos una serie de problemas a los que tuvimos que dar solución. Por ejemplo, la carga de procesamiento resultó demasiado pesada y hubo que utilizar técnicas de optimización ya descritas.

El código tiene partes donde se encontraron problemas que habitualmente se resuelven depurándolos. La depuración sobre *hardware* gráfico es muy complicada de llevar a cabo. Intentamos resolverlos con un programa de terceros que no proporcionó todas las funcionalidades esperadas y, por ello, la corrección del código durante todo el desarrollo fue compleja.

Otro momento en el que tuvimos que diseñar soluciones asequibles fue durante la proyección de sombras. Después de explorar e implementar varias posibilidades, fue interesante comprobar que las soluciones más simples, bien utilizadas, a veces resultan ser las que dan mejores resultados.

Pese a todo, conseguimos realizar el objetivo planteado de obtener una simulación realista de los efectos de iluminación sobre la atmósfera, en tiempo real, para grandes escenarios en tres dimensiones, todo ello sobre un procesador gráfico de gama media. Además, esta solución integra sistemas comúnmente utilizados para obtener imágenes de calidad, por lo que podría ser utilizado por terceros en otras simulaciones.

Este trabajo ha requerido un gran componente de investigación que, por ello, creemos que no debería acabar aquí. Por ejemplo, se podría trabajar en la mejora del rendimiento y del realismo aplicando otras técnicas, como el "renderizado" en desfase o el cálculo dinámico de sombras.

Como resultado de este trabajo podemos decir que las tecnologías gráficas caminan en la dirección correcta para que los programadores puedan ofrecer cada día mejores experiencias a los usuarios que consumen este tipo de *software*. Existen otras áreas diferentes a las del entretenimiento en las que se podrían aprovechar estas tecnologías, como la física experimental o las ciencias medioambientales.

Todavía queda mucho por hacer para que este tipo de simulaciones no supongan una excesiva carga y para que, de esa manera, puedan ser incluidas en cualquier proyecto, sin preocuparse por el rendimiento del mismo. Quizá, con el tiempo, se podrán ofrecer soluciones que no tengan que recurrir a aproximaciones, sino que logren una simulación fluida que sea comparable con el mundo real.

8. Conclusions

The application of physical models in the simulations is essential to provide realism to the generated images. In simulations of physical models, the scattering of light in the atmosphere is essential to provide a realistic open environment or space model.

However, these models make use of algorithms that require calculations which are not easily processed by computers, and cause too high cost, with consequent loss of performance. To apply such algorithms in programs that require an adequate flow of images, we must turn to approximations that offer similar results by sacrificing some of the final quality.

During the development of the prototypes, we found a series of problems that had to be solved. For example, the processing load became too heavy and had to use optimization techniques already described.

Problems were found in the code that they are usually resolved by debugging. Graphic debugging on hardware is very difficult to carry out. We try to solve them with a third party program that did not provide all the expected features and therefore the correction of code throughout development was complex.

Another time we had to develop affordable solutions was during casting shadows. After exploring and implementing various possibilities, it was interesting that the simplest solutions, well used, sometimes turn out to be those that give the best results.

Nevertheless, we managed to achieve the our objective that was to obtain a realistic simulation of lighting effects on the atmosphere in real time, for large three-dimensional scenes, all on a midrange graphics processor. Moreover, this solution integrates commonly used systems to improve image quality, so it could be used by third parties developers in other simulations.

This work has required a great research; therefore, we believe it should not end here. For example, somebody could work on improving performance and realism using other techniques such as the deferred rendering or the dynamic calculation of shadows.

As a result of this work we can say that the graphics technologies walk in the right way for developers to offer every day better experiences to users who consume this kind of software. There are other different targets than the entertainment which could take advantage of these technologies, such as experimental physics and environmental sciences.

Much remains to be done before such simulations do not impose a disproportionate processing load, so that could be included in any project without worrying about performance. Perhaps, over time, they can offer solutions that do not have to use approximations, achieving a smooth simulation that is comparable to the real world.

Anexo

Controles de la demo

La configuración de controles para el manejo de la cámara es la siguiente:

- W, S, A y D para desplazamiento hacia delante, hacia atrás, a la izquierda y a la derecha, respectivamente.
- Espacio y Ctrl. para desplazamiento vertical hacia arriba y hacia abajo.
- Q y E para girar a la izquierda y a la derecha.
- R y F para cabecear arriba y abajo.
- La tecla O para capturar el ratón o dejar de hacerlo. Cuando el ratón está capturado se pueden realizar los giros que se producen con Q, E, R y F o directamente con el desplazamiento del ratón.
- Z para activar y desactivar el modo "turbo" con el que la cámara se desplaza por el escenario mucho más rápido.
- Más (+) y menos (–) para aumentar y disminuir la velocidad del modo "turbo". Aparece en la consola la velocidad actualizada.
- Asterisco (*) y barra (/) para aumentar y disminuir cien veces más la velocidad del modo "turbo". También sale reflejado el cambio en la consola.
- T desactiva todos los controles de la cámara o los vuelve a activar.

Otras opciones de configuración y controles son los siguientes:

- V muestra por consola la versión de OpenGL que se está utilizando.
- P muestra por consola la posición actual de la cámara.
- Las flechas de dirección de arriba y abajo cambian la dirección de la luz para simular el movimiento del sol.
- M activa el movimiento automático del sol.
- Esc. detiene la ejecución manualmente.

Bibliografía

[1] S. O'Neal, «Chapter 16. Accurate Atmospheric Scattering» de *GPU Gems 2*, Massachusetts, Pearson Addison Wesley Prof, 2005.

[2] C. Schüler, «An Approximation to the Chapman Grazing-Incidence Function for Atmospheric Scattering» de *GPU Pro 3 Advanced Rendering Techniques*, Florida, A K Peters/CRC Press, 2012.

[3] E. Yusov, de *GPU Pro 5 Advanced Rendering Techniques*, Florida, A K Peters/CRC Press, 2014.

[4] R. B. Pinheiro y A. R. Raposo, «Maxwell», 18 marzo 2014. [En línea]. Disponible: http://www.maxwell.vrac.puc-rio.br/23400/23400.PDF

[5] O. Elek, «Cescg», abril 2009. [En línea]. Disponible: http://www.cescg.org/CESCG-2009/papers/PragueCUNI-Elek-Oskar09.pdf

[6] The Khronos OpenGL ARB Working Group, OpenGL Programming Guide, Octava ed., Michigan: Addison-Wesley, 2013.

[7] J. Richard S. Wright, N. Haemel, G. Sellers y B. Lipchak, OpenGL Superbible, Quinta ed., Michigan: Addison-Wesley, 2010.

[8] RXminuS, «Atalassian Bitbucket», Noviembre 2014. [En línea]. Disponible: https://bitbucket.org/RXminuS/asteroidtrouble/src

[9] M. Stamminger y G. Drettakis, «Inria», 2002. [En línea]. Disponible: http://www-sop.inria.fr/reves/Basilic/2002/SD02/PerspectiveShadowMaps.pdf

[10] Microsoft, «MSDN», 16 noviembre 2013. [En línea]. Disponible: https://msdn.microsoft.com/en-us/library/ee416307(VS.85).aspx

[11] M. Fisher, «Matt's Webcorner», 2014. [En línea]. Disponible: https://graphics.stanford.edu/~mdfisher/Shadows.html

[12] «OpenGL Tutorial», 7 septiembre 2013. [En línea]. Disponible: http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/

[13] A. Lauritzen, «Chapter 8. Summed-Area Variance Shadow Maps» de *GPU Gems 3*, Indiana, Addison-Wesley, 2007.

[14] K. Myers, «Nvidia Developer», enero 2007. [En línea]. Disponible: http://developer.download.nvidia.com/SDK/10.5/direct3d/Source/VarianceShadowMapping/Doc/VarianceShadowMapping.pdf

[15] F. Rudolf, «NeHe», [En línea]. Disponible: http://nehe.gamedev.net/article/glsl_an_introduction/25007/

[16] E. Meiri, «Atspace», [En línea]. Disponible: http://ogldev.atspace.co.uk/

[17] Scratchapixel, [En línea]. Disponible: http://www.scratchapixel.com/index.php?nopage

[18] File Format, «File Format», 2014. [En línea]. Disponible: http://www.fileformat.info/

[19] Stackoverflow, «Stackoverflow», 14 enero 2014. [En línea]. Disponible: http://stackoverflow.com/questions/21120699/c-obj-file-parser

[20] P. Kmoch y O. Elek, «Oskee», 2010. [En línea]. Disponible: http://www.oskee.wz.cz/stranka/uploads/Real-Time_Spectral_Scattering_in_Large-Scale_Natural_Participating_Media.pdf