Universidad Nacional de Educación a Distancia

Escuela Técnica Superior de Ingeniería Informática

Proyecto de fin de Grado en Ingeniería Informática

Simulación de ópticas: Renderizador de trayectorias de luz

Alberto Viedma Ortiz-Cañavate

Dirigido por: Alfonso Urquía Moraleda

Codirigido por: Carla Martín Villalba

Curso: 2023/24

Simulación de ópticas: Renderizador de trayectorias de luz

Proyecto de Fin de Grado en Ingeniería Informática
de modalidad específica

Realizado por: Alberto Viedma Ortiz-Cañavate

Dirigido por: Alfonso Urquía Moraleda

Codirigido por: Carla Martín Villalba

Fecha de lectura y defensa: octubre de 2024

Resumen

El trabajo trata de la implementación de un *Backward Path Tracer*, una aplicación software de renderización de *gráficos 3D offline* que simula la iluminación de escenas virtuales trazando rayos lumínicos desde la cámara hasta las fuentes de luz, calculando los rebotes en los diferentes objetos en su camino.

El proyecto se ha desarrollado siguiendo un enfoque iterativo en tres fases principales:

1. Construcción de un *ray caster* básico.
2. Implementación de mejoras para convertirlo en un *Whitted Ray Tracer*.
3. Desarrollo de un sistema de iluminación global para alcanzar la funcionalidad completa de un *Backward Path Tracer*.

El proyecto comenzó con la creación de la lógica para representar escenas con figuras geométricas simples, calculando intersecciones rayo-primitiva mediante el álgebra lineal y la geometría. Se diseñaron clases y estructuras de datos correctas para el trazado de rayos, permitiendo escalar el programa con nuevas funcionalidades. Se investigó cómo representar la luz en pantalla, implementando técnicas simples como Phong y Blinn-Phong, y luego avanzando a cálculos de radiancia con técnicas basadas en físicas (PBR), considerando luces directas y las propiedades de los materiales.

Dado el alto costo computacional, se usaron estructuras de aceleración como *Bounding Boxes*, o la *Grid de Fujimoto* y se paralelizaron cálculos con OpenMP. Finalmente, se implementó un sistema de iluminación global con integración por Montecarlo, que calcula rebotes de luz y su impacto en la escena. Se emplearon

algoritmos avanzados como *Next Event Estimation y Multiple Importance Sampling*, optimizando la convergencia con técnicas como *Russian Roulette*. El resultado es un *Backward Path Tracer* que produce imágenes de gran realismo, aportando un valioso estudio en computación gráfica, relevante en industrias como cine, videojuegos y arquitectura.

Abstract

The project focuses on the implementation of a Backward Path Tracer, an offline 3D graphics rendering software application that simulates scene illumination by tracing light rays from the camera to the light sources, calculating reflections off various objects along the way.

The project was developed following an iterative approach across three main phases:

1. Construction of a basic ray caster.
2. Implementation of enhancements to transform it into a Whitted Ray Tracer.
3. Development of a global illumination system to achieve the full functionality of a Backward Path Tracer.

The project began by creating the logic to represent scenes with simple geometric figures, calculating ray-primitive intersections using linear algebra and geometry. Appropriate classes and data structures were designed for ray tracing, allowing the program to scale with new functionalities. Research was conducted on how to represent light on the screen, starting with simple techniques like Phong and Blinn-Phong Shading, and progressing to radiance calculations with Physically Based Rendering (PBR) techniques, considering direct lighting and material properties.

Given the high computational cost, acceleration structures such as Bounding Boxes and the Fujimoto Grid were used, and calculations were parallelized using OpenMP. Finally, a global illumination system was implemented using Monte Carlo integration, which calculates light bounces and their impact on the scene. Advanced algorithms

like Next Event Estimation and Multiple Importance Sampling were employed, optimizing convergence with techniques such as Russian Roulette. The result is a Backward Path Tracer that produces highly realistic images, contributing valuable research in computer graphics, relevant to industries such as film, video games, and architecture.

Palabras Clave

Ray caster

Ray Tracer

Ray Tracing

Path Tracing

Backward Path Tracer

Trazador de Rayos

PBR

Método de Montecarlo

Key words

Ray caster

Ray Tracer

Ray Tracing

Path Tracing

Backward Path Tracer

PBR

Monte Carlo Method

Índice

Índice de figuras

Capítulo 1

Introducción, objetivos y estructura

1.1 Introducción

El *Ray Tracing* ha revolucionado la creación de imágenes digitales, redefiniendo los estándares visuales en múltiples industrias, desde el cine hasta los videojuegos y la arquitectura. Esta técnica, que simula con precisión el comportamiento de la luz, ha permitido a los creadores generar imágenes de un realismo sin precedentes, haciendo posibles escenas visuales que antes parecían inalcanzables.

El *Ray Tracing* tradicionalmente ha estado relacionado con el campo del *Renderizado Offline*, en contraposición al *Renderizado* en *Real Time*. Se trata de imágenes de gran calidad generadas en tiempos normalmente largos no aptas para su consumo inmediato. Pero los avances en hardware han hecho posible el uso de *Ray Tracing* en tiempo real en los últimos años, con sus consiguientes limitaciones con respecto al *Ray Tracing Offline*.

El desarrollo del *Ray Tracing* tiene sus raíces en los avances científicos y técnicos de varias figuras clave. En la década de 1980, *Turner Whitted* introdujo la técnica fundamental del *ray tracing*, que permitió simular reflexiones, sombras y transparencias de manera precisa, marcando el inicio de una nueva era en el renderizado gráfico. Su trabajo fue complementado por las contribuciones de *Bui Tuong Phong* y *James F. Blinn*, quienes desarrollaron y perfeccionaron técnicas de sombreado que mejoraron la representación de superficies brillantes y la reflexión

especular, componentes esenciales para aumentar el realismo en las imágenes generadas por computadora.

Kajiya Kay, James T. Kajiya, añadió un avance clave con la formulación de la ecuación del renderizado, que proporcionó la base matemática necesaria para modelar cómo la luz interactúa con superficies tridimensionales, un elemento central en la técnica de *ray tracing.*

Además de estas técnicas fundamentales, otros científicos han contribuido significativamente al campo del renderizado avanzado. Henrik Wann Jensen introdujo la técnica de *Subsurface Scattering*, que permite simular cómo la luz penetra y se dispersa en materiales translúcidos, como la piel humana, elevando el nivel de realismo en personajes y criaturas digitales. Esta técnica ha sido utilizada en películas como *Avatar* y *The Lord of the Rings* para crear efectos visuales sorprendentes. *Marc Levoy* también aportó al desarrollo del ray tracing con sus investigaciones en gráficos por computadora y su trabajo en la captura de luz y la fotografía computacional, que han ampliado las aplicaciones de esta tecnología.

El *Physically-Based Rendering (PBR),* cuyo desarrollo tuvo un boom en los 2000 de la mano de científicos destacados como *Brent Burley* de Disney, es otra técnica que ha transformado el *ray tracing* al modelar cómo la luz interactúa con los materiales de manera física y realista. Películas como *Big Hero 6, Wall-E o Ratatouille* han empleado PBR para lograr texturas y efectos visuales que imitan la realidad de manera precisa. *Eugene d'Eon y Jacopo Pantaleoni* también han contribuido a mejorar esta técnica, especialmente en la simulación de la dispersión subsuperficial y otros fenómenos ópticos complejos.

Kajiya también es conocido por introducir el concepto de *Path Tracing* (Trazado de caminos), una técnica de *renderizado* que sigue trayectorias de rayos individuales desde la cámara a través de la escena para simular la iluminación global. Esta técnica es una implementación práctica de la Rendering Equation y ha sido clave en el desarrollo de métodos avanzados de renderización, como el *Backward Path Tracing (Trazado de caminos inverso).*

Por otro lado, la técnica de *Next Event Estimation (NEE),* utilizada en la integración Montecarlo, ha sido esencial para mejorar la eficiencia del *ray tracing. Eric Veach* es

uno de los científicos que desarrolló esta técnica, que reduce el ruido en las imágenes y optimiza el cálculo de la interacción de la luz, permitiendo el uso del *ray tracing* en *tiempo real*, una innovación fundamental para su aplicación en videojuegos modernos.

El impacto de estas innovaciones no se limita a los aspectos técnicos; también ha transformado la industria del entretenimiento visual. En el cine, motores de renderizado como *Pixar's RenderMan* han sido esenciales para crear imágenes y efectos visuales complejos en películas como *Toy Story*, *Finding Nemo*, y *Monsters, Inc.*, todas ellas pioneras en el uso del *Ray Tracing* para lograr un nivel de detalle visual sin precedentes. *Arnold*, un renderer español, otro motor clave, ha permitido a los cineastas desarrollar mundos visualmente impresionantes en películas como *Gravity* y *Blade Runner 2049*, manteniendo un alto nivel de realismo en cada escena.

En la industria de los videojuegos, la adopción del *Ray Tracing* en tiempo real ha transformado la experiencia de juego. Juegos como *Cyberpunk 2077* y *Control* han utilizado esta tecnología para crear entornos urbanos y de ciencia ficción con una iluminación y reflejos increíblemente realistas, elevando la inmersión del jugador a nuevas alturas. La capacidad de ofrecer estas experiencias visuales avanzadas ha sido posible gracias a la evolución de las GPUs, como la serie RTX de NVIDIA, que ha permitido a los desarrolladores integrar el *Ray Tracing* directamente en sus títulos.

En conjunto, el *Ray Tracing* y los avances asociados, impulsados por la labor de científicos como *Whitted, Phong, Blinn, Kajiya, Jensen, Burley, Veach y Levoy*, han transformado no solo la calidad visual de las imágenes digitales, sino también la economía de las industrias que dependen de estos gráficos. Motores como *RenderMan* y *Arnold* han sido fundamentales en el cine, mientras que la tecnología de GPUs avanzadas ha permitido que el *Ray Tracing* se convierta en una parte integral de los videojuegos modernos. Este conjunto de innovaciones sigue ampliando los límites de lo que es posible en la creación de gráficos digitales, con un impacto duradero en el futuro del entretenimiento visual y más allá.

Sobre las técnicas de renderizado, existen dos técnicas principales de renderizado 3D: el *Ray Tracing* y la *rasterización*. La *Rasterización* es una técnica ingeniosa, muy eficiente en términos de recursos computacionales, lo que la hace ideal para

aplicaciones en tiempo real, como los videojuegos. Aunque es menos precisa que el *Ray Tracing*, permite aproximaciones suficientemente buenas de las interacciones de luz para la mayoría de los usos prácticos.

El *Ray Tracing*, por otro lado, es una técnica muy intuitiva, mucho más precisa y costosa computacionalmente, que simula de manera más realista las interacciones de la luz. Dentro de esta categoría se encuentran: el *Ray Casting*, que renderiza imágenes calculando intersecciones; el *Ray Tracing*, que amplía sus capacidades y calcula reflexiones y refracciones; y, por último, el *Path Tracing*, que incorpora modelos de simulación de iluminación global mediante el método de Montecarlo. Aunque su nombre especifico es *Monte Carlo Ray Tracing* y el *Path Tracing* es una configuración del mismo, es extendido el uso del nombre *Path Tracing.*

Mientras que la *Rasterización* consiste en proyectar las figuras en la pantalla, triángulos normalmente, (transformándolas desde el espacio tridimensional al plano 2D de la pantalla) almacenando su valor de profundidad en un búfer para determinar qué píxeles son ocluidos por otros, el *Ray Tracing* se basa en trazar el camino que sigue la luz al interactuar con los objetos. Esto incluye calcular las intersecciones de los rayos con las superficies de los objetos para simular efectos ópticos como reflexiones, refracciones y sombras de manera mucho más precisa.

Grosso modo, la Rasterización es por tanto una proyección de la geometría sobre una pantalla y "pixel a pixel", se decide qué triangulo se renderiza, a esta operación se la conoce como *raster*. El Ray Casting, por el contrario, lanza un rayo "por pixel" y calcula la intersección de la primitiva con el rayo analíticamente, mediante ecuaciones.

Existe otra técnica llamada *Ray Marching*, en la que iterativamente se calcula la intersección de manera numérica. Una técnica en la que España cuenta con una eminencia a nivel mundial llamada *Iñigo Quilez*. El *Ray Marching* se utiliza para el cálculo de iluminación en medios participativos: niebla, nubes, atmósfera…

Todas son técnicas que pueden ser usadas a la vez, con diferentes propósitos. En videojuegos se usa la Rasterización pues las tarjetas gráficas aceleran su cálculo mediane un pipeline especializado. Pero los videojuegos actuales también usan *Ray Marching* o *Ray Tracing* si la tarjeta gráfica es lo suficientemente avanzada para ello.

En *Renderizado Offline* se suele usar *Path Tracing*, que calcula todas las interacciones posibles de la luz e innumerables rebotes. Calcula los caminos de la luz. El *Path Tracing* es prohibitivo para Tiempo Real y los sistemas más avanzados de iluminación global como *Lumen* de Unreal lo simulan mediante heurísticas de muy baja calidad, teniendo que filtrar sus resultados.

Hay distintos tipos de *Path Tracers*:

En el *Backward Path Tracing*, los rayos se emiten desde la cámara (o el ojo del observador) hacia la escena, y luego se calculan sus interacciones con los objetos en el camino hasta llegar a una fuente de luz, lo que se conoce como *Backward Path Tracing* o *Path Tracing inverso*.

Este método es más eficiente que trazar rayos desde las fuentes de luz hacia la cámara (lo que sería *Forward Path Tracing*), porque la mayoría de los rayos emitidos desde una fuente de luz no alcanzan la cámara, lo que resultaría en un desperdicio significativo de recursos computacionales.

Pero trazar rayos desde la fuente de luz tiene sus ventajas, especialmente en luces que viajan por pequeñas rendijas en una escena. Por ello existen no obstante *Path Tracers* conocidos como *Bidirectional Ray Tracers* que suman ambos tipos haciéndolos más versátiles en innumerables ocasiones.

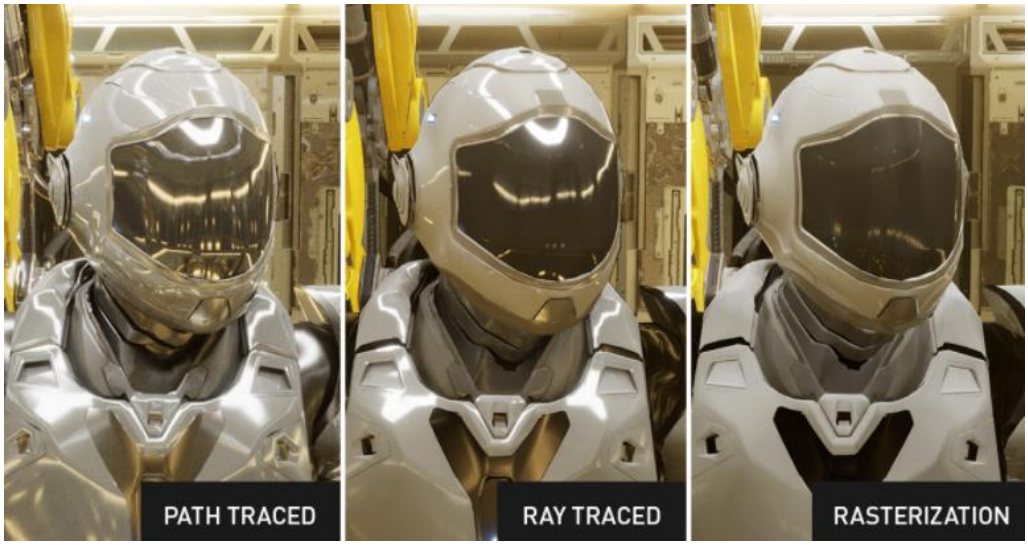

Figura 1.1 Diferencias entre path tracing, ray tracing y rasterización **[7]**.

Hay otros tipos de técnicas que no dependen de la cámara y que también se pueden sumar al *Path Tracing*, como el *Photon Mapping*, con la que se mapean paquetes de fotones en la escena emitidos por las luces que se guardan en estructuras de datos especiales. El *Photon Mapping* es especialmente interesante para la representación de refracciones, sombras suaves y cáusticas.

También hay técnicas especiales para hacer a los *Path Tracers* converger más rápido como *Metropolis Light Transport* que se basa en una matemática muy compleja que muy pocos renderers profesionales implementan basada en investigación operativa como las *cadenas de Markov*.

La motivación principal de realizar este proyecto radica en la investigación del renderizado 3D, explorando las técnicas de renderización offline, sus ventajas, sus limitaciones y su implementación. Analizando los motores de renderizados profesionales y las técnicas que emplean. Este trabajo no supone ningún avance en el campo de la computación gráfica, se trata de una recopilación y aplicación de técnicas existentes. Aunque se propone el uso del valor absoluto en lugar de la función máximo en algunas de las ecuaciones del sombreado y una interpolación rápida y original para el *Multiple Importance Sampling.*

1.2 Objetivos

El presente trabajo tratará de la implementación de un *Backward Path Tracer*, una aplicación de software para la renderización de gráficos 3D offline que simula la iluminación de escenas virtuales mediante el trazado de rayos lumínicos desde la cámara hacia las fuentes de luz, calculando los rebotes que estos realizan en los diferentes objetos de la escena. Se plantearán los siguientes objetivos generales:

- Investigación de técnicas de renderización offline existentes, analizando sus ventajas y contrapartidas.
- Estudio de los renderers profesionales en el campo, así como de las técnicas que implementan.

- Investigación y desarrollo de técnicas de simulación de iluminación, desde modelos altamente eficientes a modelos basados en la física.
- Adquisición e implementación de conocimientos matemáticos y estadísticos aplicados al cálculo de iluminación global.
- Desarrollo e implementación de algoritmos y estructuras de datos utilizados en este tipo de programas.
- Desarrollo integral de un *Backward Path Tracer,* asegurando su funcionalidad y eficiencia en la simulación de la iluminación.

El proyecto se desarrollará siguiendo un enfoque iterativo en tres fases principales:

1. Desarrollo de un *Ray Caster* básico.
2. Incorporación de mejoras para evolucionarlo a un *Whitted Ray Tracer*.
3. Creación de un sistema de iluminación global para lograr la funcionalidad completa de un *Backward Path Tracer*.

El primer paso consistirá en crear la lógica para representar escenas compuestas de figuras geométricas simples (como esferas, planos, triángulos, etc.), calculando las intersecciones rayo-primitiva mediante técnicas matemáticas, utilizando álgebra lineal y geometría.

Será imperativo diseñar correctamente clases y estructuras de datos que permitirán el trazado de rayos y que serán lo suficientemente robustas para escalar el programa añadiendo funcionalidades. A su vez, se estudiará la representación de la luz en pantalla y cómo traducir el flujo luminoso en unidades comprensibles para la lente. Se comenzará implementando técnicas de iluminación simple como Phong o Blinn-Phong, y en iteraciones posteriores, se calculará la radiancia de las reflexiones con técnicas más sofisticadas basadas en física (PBR).

Inicialmente, se implementarán luces directas: puntuales, de foco y direccionales. Todo esto tomará en cuenta las propiedades de los materiales de los objetos, usando texturas y definiendo materiales que describirán las funciones de reflectancia y color de las superficies.

Estos cálculos serán intensivos y no viables en un tiempo razonable para el renderizado de escenas complejas, por lo que se investigarán formas de optimizar la geometría utilizando estructuras de aceleración, como Bounding Boxes o el grid de Fujimoto. Además, se explorará la paralelización de cálculos en CPU y GPU mediante la librería OpenMP para optimizar el rendimiento.

Este proyecto estará vinculado a la óptica y la radiometría, lo que requerirá una comprensión de conceptos de cálculo y física, como la Ley de Snell, el coseno de Lambert, el efecto Fresnel, la radiancia, la irradiancia, y la función bidireccional de reflectancia (BRDF), junto con el uso del método de Montecarlo.

Finalmente, se implementará un sistema de iluminación global mediante la integración por Montecarlo, que calculará tanto la luz directa como la indirecta, considerando todos los rebotes de luz. Para acelerar la convergencia y el ruido de la imagen, se estudiarán algoritmos avanzados como Next Event Estimation y Multiple Importance Sampling, junto con técnicas como Russian Roulette.

Por último, se realizará una pequeña interfaz gráfica para el usuario, GUI, para que se pueda usar el Path Tracer.

El resultado será un Backward Path Tracer capaz de producir imágenes altamente realistas, aunque a un alto costo computacional. Este desarrollo representará un estudio fundamental en el campo de la computación gráfica, un área que abarca tanto el renderizado offline como en tiempo real, la simulación física y la representación geométrica.

Es un área del que depende muchísima industria, como el cine, los videojuegos, la arquitectura, el diseño en general, la visualización de datos y un largo etcétera. Cabe destacar que España cuenta con investigadores y empresas líderes en este campo, como Next Limit o Solid Angle.

1.3 Estructura

A continuación, se describe cómo se estructura la memoria:

El Capítulo 1, “Introducción, objetivos y estructura”, presenta una breve introducción histórica y técnica del Ray Tracing y el interés que tiene realizar un proyecto de esta naturaleza. Se define el objetivo del proyecto. Y se indica como se estructuran los capítulos de la memoria.

En el Capítulo 2, “Marco conceptual”, se presentan conceptos relacionados con el mundo 3D y las técnicas de renderizado, proporcionando una base teórica para comprender los principios fundamentales y metodologías empleadas en la creación y simulación de imágenes tridimensionales.

En el Capítulo 3, “Análisis y metodología”, se describe la metodología utilizada y se muestran los requisitos del sistema mediante historias de usuario en un Backlog.

En el Capítulo 4, “Diseño”, se desarrolla la explicación del diseño proyecto.

En el Capítulo 5, “Implementación”, se desarrolla la explicación del diseño y los algoritmos del proyecto. Se muestran imágenes que son las pruebas de validación de los criterios de aceptación de las historias de usuario descritas en el Capítulo 3 durante las tres iteraciones, por último, se explica cómo se ha generado la interfaz de usuario”.

En el Capítulo 6, “Pruebas” se explica el tipo de pruebas realizadas y se muestra una suerte de pruebas de validación de las historias de usuario del Backlog.

En el Capítulo 7, “Conclusiones y trabajos futuros”, contiene las conclusiones de la memoria del proyecto, y un posible proyecto futuro.

El Anexo A, “Breve historia del renderizado y de la computación gráfica”, narra brevemente la historia del renderizado y de la computación grafica desde sus albores.

El Anexo B, “Manual de usuario”, describe la forma de uso de la aplicación.

El Anexo C, “Código fuente”, recoge el código del programa.

Organización del soporte (CD) donde se ha entregado la memoria:

En el directorio raíz se incluye la memoria en PDF, en el directorio RayTracer se encuentra el código y el proyecto del RayTracer. En el directorio Interfaz se encuentra

el código y el proyecto de la interfaz del usuario. En el directorio ejecutable, se encuentra el programa portable ejecutable junto con algunos ejemplos de prueba.

Capítulo 2

Marco conceptual

2.1 Introducción

En esta sección se procurará unificar, en la medida de lo posible, los conceptos, nociones y conocimientos más relevantes relacionados con el Renderizado Offline. Este es un campo vasto y complejo, por lo que es probable que algunas técnicas y aspectos queden sin abordarse. No obstante, se intentará ofrecer una visión general que permita comprender de manera integral el tema en cuestión.

2.2 Conceptos básicos

Las principales técnicas de renderizado 3D comprenden la rasterización, el ray casting, el ray tracing, el path tracing y el ray marching. Cada una de estas metodologías se caracteriza por su funcionamiento básico, sus aplicaciones específicas y las variaciones en términos de velocidad y realismo.

2.2.1 Rasterización

La rasterización es un proceso rápido y ampliamente utilizado en gráficos por computadora, especialmente en tiempo real. Su principal objetivo es convertir representaciones 3D de objetos (generalmente en forma de polígonos o triángulos) en una imagen 2D. Los vértices de los polígonos se proyectan en la pantalla y, pixel a pixel, se comprueba si éste (el pixel) pertenece al polígono, queda dentro de sus

límites. Normalmente se rasterizan polígonos convexos que conllevan un bajo coste computacional de comprobación, una serie de productos escalares y vectoriales. Las figuras se proyectan mediante una serie de transformaciones algebraicas lineales.

Se trata de una técnica extremadamente rápida y eficiente, el pipeline de las GPUs está preparado para realizar todas estas operaciones en paralelo. Pero, tiene sus limitaciones a la hora de hacer cálculos de iluminación, teniendo que servirse de ingeniosas técnicas para crear iluminaciones realistas, por ejemplo, el cálculo de sombras requiere técnicas como el Shadow Mapping [8].

2.2.2 Ray casting

El ray casting es una técnica analítica. Es la base del ray tracing o el path tracing. Básicamente trata de calcular la intersección de un objeto con un rayo, mediante ecuaciones geométricas. Se trata de una matemática muy similar a la que se aprende en el instituto en cuanto a intersecciones de figuras geométricas. Normalmente se asocia el ray casting a un único rayo generado por píxel de la cámara [9].

2.2.3 Ray tracing

El ray tracing (trazado de rayos) se basa en el ray casting desde la cámara a través de los píxeles de la imagen para determinar el color del píxel dependiendo en las interacciones con las diferentes superficies.

Grosso modo, se lanzan rayos desde la cámara, que interactúan con los objetos en la escena. Estos rayos pueden rebotar, reflejarse, refractarse y dispersarse, lo que permite simular de manera precisa efectos como sombras, reflexiones y refracciones.

El casteo de rayos es computacionalmente muy caro en comparación con otras técnicas, como el rasterizado. Hay múltiples formas de mejorar su rendimiento, como las estructuras de aceleración en escenas complejas o el paralelismo en CPU y GPU gracias a las nuevas innovaciones en el hardware,

Se conoce como Whitted Ray Tracing a aquel que implementa reflexiones y refracciones simples con iluminación directa y Monte Carlo Ray Tracing al ray tracing

que integra la iluminación global de la escena usando el método de Montecarlo siguiendo la Ecuación del Renderizado de Kajiya.

La integración de la iluminación global por Montecarlo supone el lanzamiento de innumerables rayos alrededor del hemisferio de una esfera que rodea al punto a ser iluminado. Es una técnica estadística. Aproximación de una integral que introduce varianza (ruido) debido a su naturaleza [10].

2.2.4 Path tracing

El path tracing es un ray tracing de Montecarlo con unas condiciones muy específicas, aunque a veces se usan ambos conceptos indiferentemente. La condición del path tracer es que los rayos lanzados no han de tener forma de árbol para el cálculo de la integración. Los caminos o paths han de producir un hijo y no más. Es en el mismo pixel donde se muestrean múltiples paths. En cambio, el ray tracing por Montecarlo generaría arboles de rayos por cada rebote [11].

2.2.5 Ray marching

El ray marching es una técnica utilizada principalmente para renderizar volúmenes y superficies implícitas, como los Campos de Distancia con Signo o Signed Distance Fields (SDF, por sus siglas en inglés). Es una técnica numérica y no analítica. En lugar de lanzar un rayo y calcular su intersección directa con una superficie geométrica, un punto "marcha" (avanza en pequeños pasos) a través del rayo. Cada paso evalúa una función que indica la distancia al objeto más cercano. El punto avanza hasta que se aproxima lo suficiente a una superficie o pasa un umbral de distancia. El caso más sencillo sería la intersección de una esfera: en este caso se calcularía la distancia del punto al centro de la esfera y, si es menor o igual que el radio, entonces se tiene constancia de una intersección.

Es ideal para representar geometría implícita, efectos volumétricos, como la niebla, o los medios participativos y superficies complejas que serían difíciles de modelar con técnicas tradicionales de polígonos.

El español Iñigo Quilez es una eminencia en el campo [12].

2.3 Modelos de reflexión

Los modelos de reflexión son necesarios para simular cómo la luz interactúa con las superficies en el renderizado 3D. Desde los enfoques iniciales como el modelo phong más simples hasta métodos avanzados y la BRDF Cook-Torrance, estos modelos permiten representar con precisión las características y el comportamiento de las superficies. Capturan cómo la luz se refleja, refracta y dispersa.

2.3.1 Primeros modelos

Los primeros rasterizadores y ray casters renderizaban imágenes en blanco y negro y más tarde con colores planos, sin tener en cuenta la interacción de la luz con los objetos. La comunidad científica comenzó un arduo trabajo para generar escenas con una iluminación que sombreara como en pintura los objetos que se representaban en pantalla, con un cierto grado de realismo.

2.3.2 Phong

Uno de los primeros avances significativos en los modelos de iluminación fue el desarrollo del modelo de reflexión de Phong, una revolución en realidad, una propuesta empírica destinada a representar la iluminación local en puntos específicos de una superficie. Este modelo fue creado por Bui Tuong Phong, un destacado investigador en el área de la computación gráfica.

El modelo de Phong fue desarrollado en la Universidad de Utah y presentado en la disertación doctoral de Phong en 1975. Su publicación incluyó, además del modelo de reflexión, una técnica para interpolar los cálculos de iluminación en cada píxel derivado de una superficie poligonal rasterizada. Esta técnica, conocida como sombreado de Phong, se emplea incluso en combinación con modelos de reflexión distintos al propuesto por Phong. Las técnicas de Phong son consideradas innovadoras y disruptivas, sentando las bases de técnicas posteriores como el Renderizado Físicamente Correcto o PBR. Es una técnica de alta eficiencia y bajo coste computacional.

En cuanto a su funcionamiento, el modelo de reflexión de Phong describe la forma en que una superficie refleja la luz combinando dos tipos de reflexión: la difusa, propia de superficies rugosas, y la especular, característica de superficies brillantes. Phong observó que las superficies brillantes presentan reflejos especulares pequeños e intensos, mientras que las superficies rugosas muestran reflejos más amplios que disminuyen gradualmente en intensidad. Además, el modelo incorpora un término ambiental que simula la luz difusa que se esparce por toda la escena, aportando un nivel adicional de realismo.

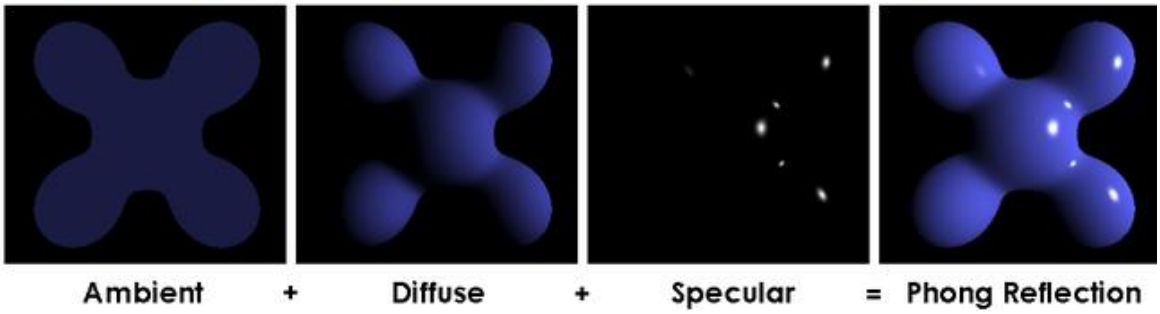

Figura 2.1 Representación visual del modelo Phong **[13].**

Realmente la parte difusa que Phong expresó es la que todos los modelos basados en física usan para su parte "lambertiana". Johann Heinrich Lambert fue un matemático, físico y astrónomo suizo del siglo XVIII. Nació en 1728 y murió en 1777. Lambert es más famoso en el campo de la óptica por formular la Ley del Coseno de Lambert, que describe cómo se atenúa la intensidad de la luz al incidir sobre una superficie en función del ángulo de incidencia. Según esta ley, la intensidad luminosa observada en una superficie es proporcional al coseno del ángulo entre el rayo de luz incidente y la normal a la superficie.

Las ecuaciones del modelo Phong son las siguientes:

Para la parte ambiental:

$$I_{ambiente} = k_a \cdot I_a \tag{2.1}$$

Donde:

- $I_{ambiente}$: Intensidad de luz ambiente.
- k_a: Coeficiente de reflexión ambiente del material (define cuánta luz ambiente refleja la superficie).
- I_a: Intensidad de la luz ambiente en la escena.

Para la parte difusa:

$$I_{difusa} = k_d \cdot I_l \cdot \max(0, .\vec{N} \cdot \vec{L}) \quad (2.2)$$

Donde:

- I_{difusa}: Intensidad de la luz difusa.
- k_d: Coeficiente de reflexión difusa del material.
- I_l: Intensidad de la luz emitida por la fuente.
- $\vec{N}$: Vector normal de la superficie en el punto.
- $\vec{L}$: Vector de dirección de la luz.

Y $\vec{N} \cdot \vec{L}$ es el coseno de Lambert.

Y para la parte especular:

$$I_{especular} = k_s \cdot I_l \cdot max(0, (\vec{R} \cdot \vec{V})^n) \quad (2.3)$$

Donde:

- $I_{especular}$: Intensidad de la luz especular.
- k_s: Coeficiente de reflexión especular del material.
- I_l: Intensidad de la luz emitida por la fuente.
- $\vec{R}$: Vector de la dirección de reflexión de la luz.
- $\vec{V}$: Vector de dirección del observador.
- n: Exponente de brillo, que controla la nitidez del reflejo (superficies más pulidas tienen valores de n más altos).

La forma general es:

$$I = I_{ambiente} + I_{difusa} + I_{especular} \quad (2.4)$$

Gracias a su equilibrio entre precisión visual y eficiencia computacional, el modelo de Phong ha sido ampliamente adoptado en diversas aplicaciones de gráficos por computadora y sigue vigente hoy en día, por ejemplo, para mejorar el rendimiento con el renderizado rápido de objetos lejanos.

El consiguiente cálculo de la luz ambiental, será posteriormente objeto de estudio. Un cálculo correcto de ésta es considerado como la implementación de Iluminación Global

Este método de iluminación comenzó siendo usado con iluminación directa, con luces directas, es decir, fuentes de luz puntuales, direccionales y de foco que son irreales pero útiles. Estas fuentes producen luz que viaja en líneas rectas desde el emisor hasta las superficies que iluminan, sin dispersión o rebote intermedios (a diferencia de la iluminación indirecta) [13].

2.3.3 Luz puntual

Una luz puntual emite luz de manera equitativa en todas las direcciones desde un solo punto en el espacio, similar a una bombilla. Tiene posición en el espacio, intensidad y color. Se utiliza para simular emisiones cercanas en el espacio. Suele tener también una propiedad de atenuación, cuanto más alejado se esté de ella más se atenúa la luz.

2.3.4 Luz direccional

Este tipo de luz simula la luz proveniente de una fuente distante, como el sol. Emite rayos paralelos de luz sobre una escena. Son una dirección, que es la que se toma en cuenta en la ecuación de Phong.

2.3.5 Luz focal (Spotlight)

Una luz focal emite luz en forma de cono. Es en realidad una luz puntual con una dirección y una apertura. La luz se atenúa con un producto escalar dependiente del vector de dirección de la luz y el formado con la posición de la luz y el punto de la superficie.

También se le puede dar a un objeto una componente emisiva, en principio un color con el que se ilumina el objeto sin importar la luz incidente en el mismo. Al principio estos objetos no afectaron en lo absoluto a la iluminación del resto, como hemos dicho. Pero después, tras la implementación de la iluminación global, serían conocidos como luces de área, luces cuya aportación es puramente indirecta.

2.3.6 Blinn-Phong

El modelo de sombreado Blinn-Phong fue creado por James F. Blinn en 1977 como una mejora del modelo de reflexión Phong.

La principal contribución de Blinn fue el uso del vector intermedio (halfway vector) entre la fuente de luz y la dirección del observador para calcular el brillo especular. Esta modificación hizo que el cálculo del sombreado fuera más eficiente y produjera reflejos especulares más suaves, especialmente en situaciones donde las fuentes de luz están en ángulos pronunciados con respecto al observador.

Las ecuaciones de Blinn-Phong son iguales salvo por la parte especular:

$$I_{especular} = k_s \cdot I_l \cdot max(0, (\vec{H} \cdot \vec{N})^n) \tag{2.5}$$

Donde $\vec{H}$ es el Halfway vector.

$$\vec{H} = \frac{\vec{V} + \vec{L}}{|\vec{V} + \vec{L}|} \tag{2.6}$$

Estos modelos de iluminación simples fueron realzados añadiendo texturas a los modelos, para crear materiales vistosos y dar mayor riqueza a las escenas [14].

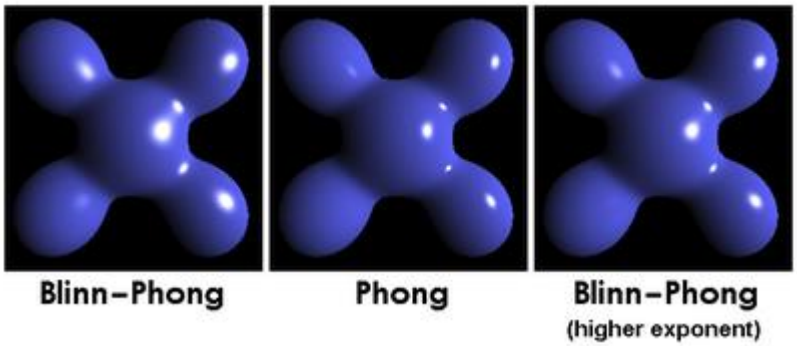

Figura 2.2 Blinn Phong vs Phong **[15].**

2.3.7 Mapas de texturas

Edwin Catmull, científico informático y cofundador de Pixar, es ampliamente reconocido por desarrollar el concepto de texture mapping como parte de su tesis doctoral en 1974. Catmull introdujo la idea de proyectar una imagen 2D (o "textura") a un modelo 3D, lo que supuso un gran avance al permitir la creación de superficies mucho más detalladas y realistas en comparación con los modelos de colores planos. Este concepto se convirtió en una técnica fundamental, transformando el realismo y detalle de los objetos 3D.

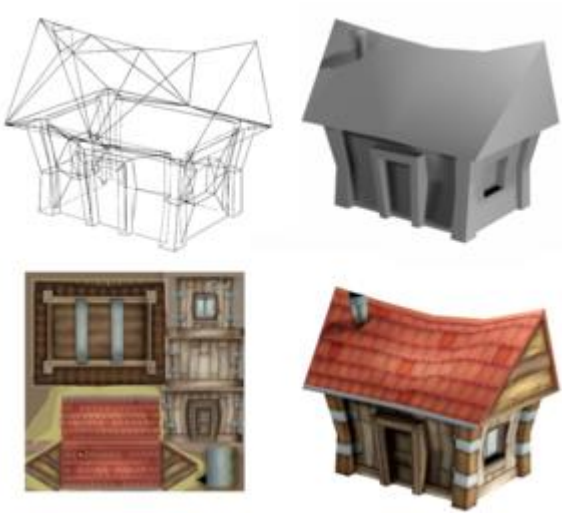

Figura 2.3 Aplicación de un mapa de texturas sobre un objeto **[16]**.

Con el uso de las texturas y los mapas de textura, se comenzó a dar mayor riqueza visual a los objetos, proyectándolas sobre las superficies de diferentes formas y generando efectos visuales sobre ellas. Los mapas de textura se pueden proyectar de múltiples formas sobre los triángulos. Los vértices de los modelos normalmente tienen una coordenada de la textura correspondiente, conocida como UV en (contraposición con XY). Luego, mediante interpolación de coordenadas baricéntricas, se obtiene el texel a representar (el pixel de la textura) en el pixel a renderizar en la pantalla. Las formas más comunes de proyección son la proyección planar, la proyección cúbica, la esférica, la proyección triplanar y el mapeo de Uvs. Las proyecciones de figuras proyectan las coordenadas de las primitivas en la figura a mapear. El mapeo de Uvs suele consistir en un trabajo artesanal conocido como Unwraping, aunque existen técnicas más o menos avanzadas que son automáticas.

Al principio los mapas de textura sólo modificaban el color. Pero pronto se implementaron mapas con múltiples propósitos, entre los más comunes los que modificaban los valores relacionados con el cálculo de la reflexión de la luz. Mapas de texturas tradicionales son:

El mapa difuso o de color que cambia el color del objeto según el pixel

- El mapa especular que controla los coeficientes especulares para representar superficies más pulidas o rugosas según el punto renderizado e incluso el color del brillo
- Los mapas de normales y bump maps que transforman las normales de los objetos en el pixel renderizado
- El mapa de oclusión ambiental que representa zonas donde la luz indirecta difícilmente se cuela, como grietas o rendijas.

Y muchos otros más. Curiosamente fue James F. Blinn quién introdujo los bump maps en 1978.

La representación de imágenes y texturas no estaba exenta de problemas como ruidos o errores en el muestreo. Uno de los temas más comunes en este campo es el aliasing [17].

2.3.8 Aliasing

El aliasing es un fenómeno que ocurre cuando una señal continua (como una imagen o un sonido) es muestreada de forma insuficiente o incorrecta al convertirla a una señal digital. El estudio del aliasing y su mitigación, a través de técnicas como el anti-aliasing, proviene del campo del procesamiento de señales y su formalización se remonta a trabajos en teoría de muestreo por parte de científicos como Harry Nyquist y Claude Shannon en los años 1920 y 1940.

En computación gráfica, las técnicas de anti-aliasing comenzaron a desarrollarse para reducir estos artefactos visuales y suavizar las imágenes, lo que ha sido una parte integral del desarrollo de gráficos digitales desde los años 70 y 80.

El aliasing en computación gráfica se presenta de diferentes formas, desde líneas en forma de sierra, ruido en texturas, hasta efectos temporales en caso de una sucesión

de imágenes como cuando una rueda parece girar en la dirección opuesta a la que debería en un video.

- Spatial aliasing (aliasing espacial): Ocurre en computación gráfica cuando se intenta representar detalles pequeños o texturas finas en una imagen con baja resolución. Los píxeles no son lo suficientemente densos para representar con precisión los detalles, lo que genera bordes irregulares y escalonados.
- Temporal aliasing (aliasing temporal): Este tipo de aliasing se refiere a artefactos visuales que ocurren cuando la tasa de cuadros por segundo (FPS) es insuficiente para captar el movimiento suave de un objeto. Se puede observar en gráficos animados o videojuegos cuando el movimiento parece entrecortado o tembloroso o incluso contrario al que debería.
- Spectral aliasing (aliasing espectral): Ocurre cuando las texturas con patrones regulares (como rayas o líneas paralelas) se muestrean incorrectamente y producen patrones de interferencia, llamados moiré. Esto es común en texturas con frecuencia espacial alta.
- Pixel aliasing (aliasing de píxeles): Ocurre cuando la resolución de píxeles es demasiado baja para representar con precisión los detalles, lo que da como resultado artefactos visuales visibles en bordes, especialmente en objetos en ángulo.

Por consiguiente, se empezaron a usar innumerables técnicas para solucionar estos artefactos entre las más importantes se cuentan:

- El sobremuestreo (supersampling) es una técnica de anti-aliasing desarrollada en los años 80 y 90 para mitigar el aliasing de píxel, espectral y espacial. Consiste en renderizar una escena a una resolución más alta que la mostrada y luego reducirla, lo que suaviza los bordes y minimiza los artefactos visuales. Aunque mejora significativamente la calidad de la imagen, tiene un alto costo computacional, ya que requiere procesar más píxeles de los que se ven en pantalla.
- El mipmapping es una técnica que crea versiones reducidas de una textura (mipmap levels) para adaptarse a diferentes distancias en la escena. Cuando una textura se ve de lejos, el sistema selecciona la versión adecuada para evitar

patrones de aliasing espectrales. Esto reduce significativamente el riesgo de interferencias.

- El filtro anisotrópico mejora la calidad de las texturas cuando se ven en ángulos oblicuos o distantes. Mientras que el mipmapping ayuda a combatir el aliasing a distancias lejanas, el anisotropic filtering trabaja en mejorar la claridad de las texturas sin introducir artefactos, especialmente cuando las superficies están inclinadas con respecto a la cámara. Se diferencia de filtros isotrópicos como el filtro bilinear o trilinear en que tiene en cuenta la inclinación de la textura. Los filtros isotrópicos son muy usados como prefiltros para mejorar la continuidad de los texeles (píxeles de la textura), haciéndolos más suaves o para reescalarlos.

Existen muchas otras técnicas, incluso temporales, que tienen en cuenta múltiples factores e incluso gradientes de velocidad entre frames renderizados [18].

2.3.9 Primeros modelos físicos

Al mismo tiempo que Phong demostraba cómo se podía representar la luz de forma simple, el renderizado comenzó a incorporar conceptos físicos para la correcta y realista representación de luces y materiales. La función de reflexión de Phong, por ejemplo, es una de muchas Funciones Bidireccionales de Reflexión o BRDF y la medida radiométrica que representa es la radiancia. A continuación, se hablará de los principales conceptos técnicos y físicos que son manejados en el renderizado.

2.3.10 Magnitudes físicas

La radiometría es la ciencia que mide la radiación electromagnética, incluyendo la luz visible, en términos de energía física. En el contexto del renderizado, es necesario entender algunos conceptos fundamentales para simular adecuadamente cómo la luz interactúa con los objetos y cómo se percibe por el ojo humano. A continuación, se describen algunos conceptos clave:

- Flujo Radiante (Φ o Radiant Flux): El flujo radiante, o potencia radiante, mide la cantidad total de energía emitida, reflejada, transmitida o recibida por una

superficie en todas las direcciones, en una unidad de tiempo. Se mide en vatios [W], y es una medida de la potencia total de la radiación electromagnética.

- En el renderizado, el flujo radiante es importante porque describe la cantidad de energía luminosa que emite una fuente de luz y que será capturada por la cámara o los receptores en la escena.
- BRDF: La función de distribución de reflectancia bidireccional (BRDF, por sus siglas en inglés: Bidirectional Reflectance Distribution Function) es una función matemática que describe cómo la luz se refleja en una superficie. Fue introducida por Fred Nicodemus alrededor de 1965. La BRDF, $f_r(\vec{p}, \overrightarrow{\omega_i}, \overrightarrow{\omega_r})$, se define como la relación entre la radiancia reflejada en una dirección $\overrightarrow{\omega_r}$ y la irradiancia incidente desde una dirección $\overrightarrow{\omega_i}$, En términos matemáticos, se expresa de la siguiente manera:

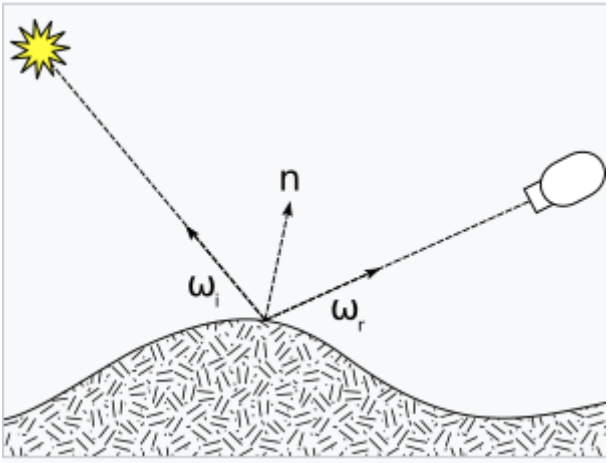

Figura 2.4 Diagrama mostrando los vectores utilizados para definir el BRDF **[19]**.

$$f_r(\vec{p}, \overrightarrow{\omega_i}, \overrightarrow{\omega_r}) = \frac{dL_r(p, \overrightarrow{\omega_i})}{dE_i(\vec{p}, \overrightarrow{\omega_r})} \tag{2.7}$$

Donde:

- $\overrightarrow{\omega_i}$ es la dirección desde la cual la luz incide en la superficie.
- $\overrightarrow{\omega_r}$ es la dirección en la que la luz es reflejada.
- $E_i(\vec{p}, \overrightarrow{\omega_r})$ es la irradiancia reflejada en la dirección $\overrightarrow{\omega_r}$.
- $L_i(\vec{p}, \overrightarrow{\omega_i})$ es la radiancia incidente desde la dirección $\overrightarrow{\omega_i}$.

Irradiancia (E): La irradiancia es la densidad del flujo radiante que llega a una superficie. Es decir, mide cuánta energía radiante impacta por unidad de área en una

superficie dada. Se expresa en vatios por metro cuadrado [W/m^2]. Este concepto es importante para calcular cómo se ilumina una superficie y cómo influirá en su apariencia en la escena.

Radiancia (L): La radiancia mide la cantidad de energía radiante que se emite, refleja o transmite desde una superficie en una dirección específica por unidad de área proyectada y por unidad de ángulo sólido. El ángulo sólido será explicado posteriormente, cuando se formule la ecuación del renderizado, pero podría entenderse como una superficie infinitesimal proyectada en una esfera o semiesfera que representa al rayo incidente y el flujo energético que este rayo tiene por esta unidad de superficie. Se mide en vatios por metro cuadrado por estereorradián [$W/m^2 \cdot sr$].

En renderizado, la radiancia es clave porque determina cómo la luz se propaga en el espacio y cómo se distribuye desde una fuente luminosa en diferentes direcciones. Es la unidad que se transporta normalmente en los rayos de los ray tracers.

Las BRDF que son físicamente correctas cumplen una serie de requisitos:

- La BRDF debe ser simétrica, lo que significa que el valor de la reflectancia en una dirección particular es el mismo si se invierten las direcciones de incidencia y reflexión: $f_r(\vec{p}, \overrightarrow{\omega_i}, \overrightarrow{\omega_r}) = f_r(\vec{p}, \overrightarrow{\omega_r}, \overrightarrow{\omega_i})$, es la conocida como Reprocidad Helmholtz.
- Una BRDF debe cumplir con la ley de conservación de la energía, lo que implica que la cantidad de energía reflejada no puede superar la cantidad de energía incidente.
- La BRDF nunca puede ser negativa, ya que la radiancia reflejada y la irradiancia incidente son cantidades físicas no negativas. Phong no respeta estos principios, por ejemplo.
- Sensibilidad de los conos del ojo humano: El ojo humano tiene tres tipos de células cono en la retina, que son sensibles a diferentes longitudes de onda de la luz, lo que nos permite percibir el color. Estos conos responden a:
    - Conos L: Sensibles a longitudes de onda largas.
    - Conos M: Sensibles a longitudes de onda medias.
    - Conos S: Sensibles a longitudes de onda cortas.

Aunque la radiometría mide la luz en términos físicos absolutos, el ojo humano no responde igualmente a todas las longitudes de onda. La percepción de la luz depende de la fotometría, que ajusta las medidas radiométricas para reflejar cómo los humanos perciben diferentes intensidades de luz.

Cabe destacar que tenemos muchos más conos medios y largos que bajos. Por lo que percibimos más unos colores que otros. Los conos L, M, V perciben un espectro amplio de luces, siendo los L y M los que más espectro perciben. Los conos S están especializados en luces azules y violetas. Las intersecciones del espectro que los conos abarcan se perciben como más brillantes. Es por eso que percibimos el color verde-amarillo como más brillante y vivo que otros colores, pese a que tengan la misma intensidad o energía.

De Radiometría a Fotometría: De Vatios a Lúmenes: La fotometría es la ciencia que mide la luz en función de su capacidad para producir una respuesta visual humana. Mientras que la radiometría mide la energía física en vatios, la fotometría mide la luz en términos de su percepción, usando lúmenes (lm).

Para convertir entre radiometría y fotometría, se utiliza la función de eficiencia luminosa espectral del ojo humano, representada por una curva que muestra la sensibilidad de los conos a diferentes longitudes de onda. La mayor sensibilidad está en la longitud de onda de alrededor de 555 nm (luz verde-amarilla).

La relación entre la radiancia y la luminancia está dada por la siguiente fórmula:

$$L = K_m \cdot V_\lambda \cdot L_e \qquad (2.8)$$

donde:

K_m(555nm)=683 lm/W es el valor máximo de la eficiencia luminosa de la sensibilidad humana a la luz verde-amarilla.

$V_\lambda$es la curva de eficiencia luminosa que depende de la longitud de onda y va de 0 a 1.

L [lm]: La luminancia. Los lúmenes son una unidad fotométrica que mide el flujo luminoso, es decir, la cantidad de luz visible que una fuente emite, ajustada según la sensibilidad del ojo humano. Por ejemplo, una fuente de luz que emite más luz en las longitudes de onda verde será percibida como más brillante (más lúmenes) que una

fuente que emite la misma cantidad de energía en el espectro rojo o azul, debido a la mayor sensibilidad del ojo humano al verde. La luminancia expresa la luminosidad visual que genera la radiancia.

L_e es la radiancia [20].

2.3.11 Funciones de reflectancia

Existen muchos tipos de funciones de reflectancia. Las físicamente correctas suelen ser modelos basados en microfacetas. Las microfacetas son las caras microscópicas de los materiales, de las superficies. Según sea su disposición, los fotones rebotan de distintas maneras. Superficies rugosas presentan microfacetas de normales desordenadas produciendo reflexiones difusas, las superficies lisas tienen microfacetas apuntando a una dirección parecida, produciendo reflejos nítidos y brillantes.

Figura 2.5 Luz interactuando con microfacetas **[21]**.

También hay modelos que tienen en cuenta el tipo de material en el que rebota la luz: los materiales dieléctricos reaccionan a la luz de manera distinta a los metálicos, pues los metálicos suelen reflectar la luz en longitudes de onda específicas, además de que tienen propiedades de reflexión y refracción completamente diferentes.

Las BRDFs pueden medirse directamente a partir de objetos reales de los distintos materiales utilizando cámaras y fuentes de luz calibradas; sin embargo, se han propuesto muchos modelos fenomenológicos y analíticos.

Algunos ejemplos:

- Modelo Lambertiano: representa superficies perfectamente difusas (mate) mediante una BRDF constante.
- Modelo Lommel–Seeliger: reflexión lunar y marciana.

- Modelo de dispersión de Hapke: aproximación físicamente motivada de la solución de transferencia radiativa para una superficie porosa, irregular y particulada. A menudo se utiliza en astronomía para simulaciones de reflexión en superficies de planetas o cuerpos pequeños. Existen múltiples versiones y modificaciones.
- Modelo de reflectancia de Phong: un modelo fenomenológico similar a la especularidad plástica.
- Modelo de Blinn–Phong: similar a Phong, pero permite que ciertas cantidades sean interpoladas, reduciendo el costo computacional.
- Modelo de Torrance–Sparrow: un modelo general que representa superficies como distribuciones de microfacetos perfectamente especulares.
- Modelo de Cook–Torrance: un modelo de microfacetos especular (Torrance–Sparrow) que tiene en cuenta la longitud de onda y, por tanto, el cambio de color.
- Modelo de Ward: un modelo de microfacetos especular con una función de distribución elíptica-gaussiana dependiente de la orientación de la tangente de la superficie (además de la normal de la superficie).
- Modelo de Oren–Nayar: un modelo de microfacetos "difuso dirigido", con microfacetos perfectamente difusos (en lugar de especulares).
- Modelo de Ashikhmin–Shirley: permite la reflectancia anisotrópica, junto con un sustrato difuso bajo una superficie especular.
- Modelo HTSG (He, Torrance, Sillion, Greenberg): un modelo comprensivo basado en principios físicos.
- Modelo ajustado de Lafortune: una generalización de Phong con múltiples lóbulos especulares, destinado a ajustes paramétricos de datos medidos.
- Modelo de Lebedev: aproximación de BRDF analítica mediante mallas.
- Modelo ABg
- Modelo de correlación K (ABC)

Una de las funciones más importantes de reflexión es la BRDF de Cook-Torrance que se analizará a continuación [22].

2.3.12 BRDF Cook-Torrance

La BRDF de Cook-Torrance fue desarrollada en 1982 por Robert Cook y Kenneth Torrance, quienes presentaron su modelo en un artículo titulado *"A Reflectance Model for Computer Graphics*". Este trabajo revolucionó la representación de la reflectancia especular al introducir un modelo físicamente basado que toma en cuenta la microestructura de las superficies, lo que permite reflejos más realistas.

El modelo de Cook-Torrance se basa en la teoría de las microfacetas. Este enfoque mejoró significativamente la forma en que las superficies rugosas y especulares podían ser representadas en gráficos por computadora, en comparación con modelos más simples como el de Phong.

La BRDF de Cook-Torrance se divide en dos partes, una parte difusa y una parte especular:

$$f_r = k_d f_{ambert} + k_s f_{cook-torrance} \tag{2.9}$$

La reflectancia difusa en PBR suele modelarse con un enfoque lambertiano, donde se asume que la luz se refleja igualmente en todas las direcciones desde una superficie rugosa o mate. El modelo lambertiano sigue la ley de Lambert, que establece que la cantidad de luz reflejada es proporcional al coseno del ángulo entre la luz incidente y la normal de la superficie. Esto se expresa de manera simple como:

Donde:

$$f_{ambert} = \frac{c}{\pi} \tag{2.10}$$

- c es albedo es el color base o la reflectancia inherente del material.

La reflectancia especular se modela mediante la BRDF de Cook-Torrance, que es un modelo más complejo y físicamente preciso que captura las propiedades especulares de los materiales. Este modelo se basa en tres componentes clave:

$$f_{cook-torrance} = \frac{F(\theta_r, \theta_h, F_0) \cdot G(\overrightarrow{\omega_i}, \overrightarrow{\omega_r}) \cdot D(\theta_h)}{4 \cdot \cos(\theta_i) \cdot \cos(\theta_r)} \tag{2.11}$$

Fresnel (F): Describe cómo la cantidad de luz reflejada varía según el ángulo de incidencia de la luz. A medida que el ángulo de incidencia se acerca a los 90 grados (rasante a la superficie), más luz se refleja en lugar de refractarse dentro del material. La aproximación de Schlick es una versión simplificada y computacionalmente eficiente de la ecuación de Fresnel completa:

$$F\left(\overrightarrow{\omega_r}, \vec{H}, F_0\right) = F_0 + (1 - F_0) \cdot (1 - (\vec{H} \cdot \overrightarrow{\omega_r}))^5 \qquad (2.12)$$

Donde F_0 es la reflectancia especular del material en un ángulo normal (incidencia directa).

Existen otras aproximaciones:

Normal Distribution Function (D): Describe la distribución de la normal de de microfacetas en una superficie. Las microfacetas son pequeñas variaciones en la superficie que afectan la reflexión especular. La D comúnmente utilizada en PBR es la de Trowbridge-Reitz (GGX), que modela mejor cómo las microfacetas afectan la dispersión de la luz especular en materiales rugosos:

$$D_{GGXTR}(\theta_h) = \frac{\alpha^2}{\pi \cdot \left(\left(\vec{N} \cdot \vec{H}\right)^2 \cdot (\alpha^2 - 1) + 1\right)^2} \qquad (2.13)$$

Donde:

α es la rugosidad del material.

$\vec{N}$ es la normal de la superficie.

$\vec{H}$ es el vector mitad (entre la dirección de la luz y la vista).

Función de Sombreado o Geometría (G): Esta función describe el efecto de auto-sombreamiento de las microfacetas de una superficie y cómo afectan la cantidad de luz reflejada. Específicamente, describe cómo las microfacetas bloquean la luz, tanto

la luz incidente (sombreamiento) como la luz reflejada (ocultación). Se utilizan modelos como Smith para calcular este efecto:

$$G_{SchlickGGX}(\vec{\omega}) = \frac{\vec{N} \cdot \vec{\omega}}{(\vec{N} \cdot \vec{\omega}) \cdot (1 - k) + k} \tag{2.14}$$

Para que se cumpla la conservación de la energía, es necesario hacer una clara distinción entre la luz difusa y la luz especular. En el momento en que un rayo de luz incide sobre una superficie, se divide en dos partes: una de refracción y otra de reflexión. La parte de reflexión corresponde a la luz que se refleja directamente sin penetrar en la superficie; esto es lo que se conoce como iluminación especular. La parte de refracción es la luz restante que entra en la superficie y es absorbida; esto es lo que se denomina iluminación difusa.

Cabe destacar ciertos matices, ya que la luz refractada no se absorbe de manera inmediata al entrar en contacto con la superficie. Desde la física, se sabe que la luz puede modelarse como un haz de energía que sigue avanzando hasta que pierde toda su energía; la forma en que un haz de luz pierde energía es a través de colisiones. Cada material está compuesto por diminutas partículas que pueden colisionar con el rayo de luz, como se ilustra en la imagen siguiente. Las partículas absorben parte o la totalidad de la energía de la luz en cada colisión, que se transforma en calor.

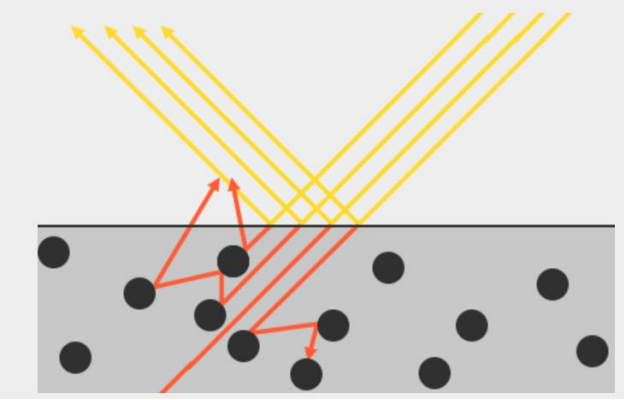

Figura 2.6 Scatering dentro del material **[23]**.

En el modelo de Cook-Torrance, los materiales dieléctricos y metálicos se modelan de manera diferente debido a sus propiedades físicas distintivas en relación con la

reflexión y la refracción de la luz. A continuación, se explican las diferencias clave y cómo el modelo Cook-Torrance aborda estos materiales:

Materiales dieléctricos:

Los dieléctricos son materiales no conductores, como el vidrio, el plástico, el agua, la madera o la piel. Tienen ciertas propiedades que afectan cómo reflejan y refractan la luz.

Propiedades de los dieléctricos:

- Albedo: Los materiales dieléctricos tienen una parte difusa (componente lambertiano) que refleja la luz de manera uniforme en todas las direcciones. Esto es lo que da lugar a su color base o albedo.
- Índice de refracción (IOR): Los dieléctricos tienen un índice de refracción entre 1 y 2, lo que significa que la luz se refleja y refracta parcialmente al entrar en el material. Esto afecta los cálculos de Fresnel.

Materiales metálicos:

Los metales (conductores) como el oro, la plata, el aluminio o el cobre tienen características muy diferentes a los dieléctricos en términos de interacción con la luz.

Propiedades de los metales:

- Sin reflexión difusa: A diferencia de los dieléctricos, los metales no tienen una componente difusa significativa. La luz que incide en un metal es reflejada principalmente de manera especular.
- Reflectancia especular alta: Los metales tienen una reflectancia especular alta en todos los ángulos de incidencia. Esto se debe a que casi toda la luz que incide sobre un metal es reflejada de manera especular. No hay transmisión de luz (o refracción) dentro del material, ya que los metales no permiten que la luz penetre en ellos.
- Color metálico: Los metales tienen un color especular que es característico del material. A diferencia de los dieléctricos, donde el especular tiende a ser de color blanco, los metales reflejan la luz con un tinte que depende del metal específico (por ejemplo, el oro tiene un reflejo dorado, y el cobre, uno rojizo).

Cómo se modela en Cook-Torrance: principalmente mediante el Fresnel, que también decide los coeficientes difusos y especulares de la ecuación (Kd y Ks respectivamente).

Cook-Torrance es la BRDF más extendida en la actualidad, debido a su versatilidad y su bajo coste computacional. En la década de los 2010 sustituyó a Phong en casi toda la industria del tiempo real introduciendo nuevos mapas de texturas para controlar la función en las superficies: el mapa de albedo o color, el mapa de normales, el mapa de metalicidad, el mapa de rugosidad y el mapa de oclusión ambiental [23].

2.3.13 Coseno de Lambert

El coseno de Lambert está presente en todas las ecuaciones de reflexión, entenderlo es sencillo, pero importante:

El coseno de Lambert es consecuencia de la ley de Lambert sobre la iluminancia. Dependiendo del ángulo de visión el ángulo solido permanece constante, pero su elemento de área diferencial se hace mayor al proyectarse sobre la superficie, mayor cuanto más oblicuo, cubriendo más superficie que el elemento diferencial de área existente. Por ello hay que reducir el ángulo sólido de visión dependiendo del coseno con la normal, para que cubra la misma área diferencial. En consecuencia, hay atenuación.

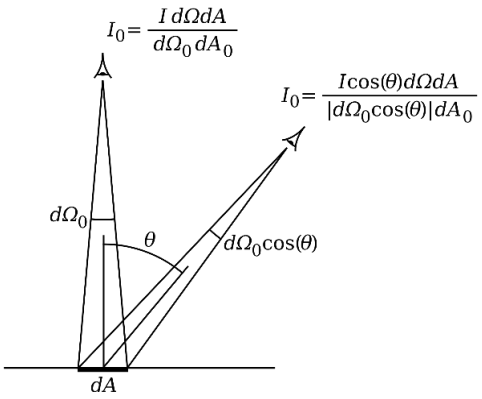

Figura 2.7 Intensidad observada (fotones/ (s·cm^2·sr)) para un observador normal y oblicuo; dA0 es el área de la abertura de observación y dΩ es el ángulo sólido subtendido por la abertura desde el punto de vista del elemento de área de emisión **[24]**.

Entendiendo el coseno de Lambert, se puede hallar algunas constantes de la función de distribución de la radiancia en un material difuso.

2.3.14 Función de distribución difusa

En el sombreado lambertiano o difuso, la función de distribución distribuye uniformemente por una semiesfera. Luego la función de distribución es una densidad constante para una semiesférica en coordenadas esféricas:

$$f_r(\vec{p}, \overrightarrow{\omega_i}, \overrightarrow{\omega_o}) = \rho \tag{2.15}$$

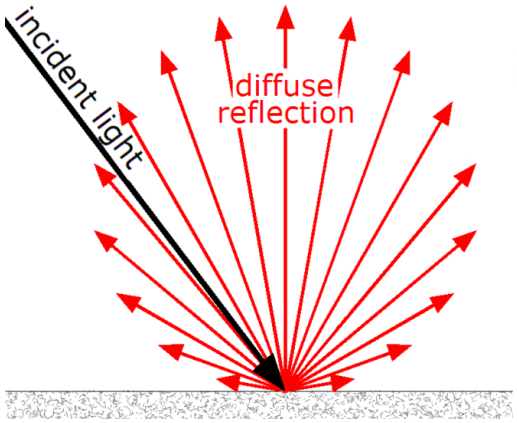

Figura 2.8 Distribución lambertiana de la radiancia de la reflexión en un punto **[25]**.

$$L_o(\vec{p}, \overrightarrow{\omega_o}) = \int_S f_r(\vec{p}, \overrightarrow{\omega_i}, \overrightarrow{\omega_o}) L_i(\vec{p}, \overrightarrow{\omega_i})(\overrightarrow{\omega_i} \cdot \vec{n}) d\omega \tag{2.16}$$

Siendo, para que haya conservación de la energía:

$$L_o(\vec{p}, \overrightarrow{\omega_o}) \leq L_i(\vec{p}, \overrightarrow{\omega_i}) \tag{2.17}$$

Luego se tiene:

$$f_r(\vec{p}, \overrightarrow{\omega_i}, \overrightarrow{\omega_o}) = \rho \tag{2.18}$$

$$L_o(\vec{p}, \overrightarrow{\omega_o}) = \int_S \rho \cdot L_i(\vec{p}, \vec{\omega}_i)(\overrightarrow{\omega_{i_i}} \cdot \vec{n}) d\omega \tag{2.19}$$

$$L_o(\vec{p}, \overrightarrow{\omega_o}) = \rho \int_0^{2\pi} \int_0^{\pi/2} L_i \cdot cos\theta \cdot sen\theta \cdot d\theta \cdot d\varphi \tag{2.20}$$

$$L_o(\vec{p}, \overrightarrow{\omega_o}) = 2\pi \cdot \rho \cdot L_i \cdot \left[\frac{-cos^2\theta}{2}\right]_0^{\pi/2} \tag{2.21}$$

$$L_o = L_i \cdot \pi \cdot \rho \tag{2.22}$$

$$L_i \cdot \pi \cdot \rho \leq L_i \tag{2.23}$$

$$\rho \leq \frac{1}{\pi} \tag{2.24}$$

Esta constante ρ, está presente en las ecuaciones de distribución que se han visto.

2.4 Iluminación global

La iluminación global, como se ha venido diciendo, es la iluminación de una escena teniendo en cuenta todas las interacciones de los rayos de luz y sus colisiones con el entorno. Antes de los años 80 esas interacciones se simulaban con artificios que no eran físicamente correctos. En los 80 aparecieron los métodos más importantes para calcular la iluminación indirecta y por tanto las primeras implementaciones de iluminación global.

2.4.1 Radiosity

La iluminación directa fue una revolución en el campo. Pero, la investigación continuó para poder llevar a cabo un cálculo correcto de la iluminación indirecta, en lugar de tener una constante ambiental. Una de las primeras técnicas creadas para ellos fue la Radiosidad o Radiosity.

Radiosity es un método de iluminación global utilizado en la computación gráfica que simula el intercambio de luz entre superficies de una escena. A diferencia de los enfoques de iluminación directa que solo consideran los rayos de luz que van directamente de la fuente a los objetos, la radiosidad busca modelar el comportamiento de la luz de manera más realista, teniendo en cuenta sólo la reflexión difusa entre superficies. Este método es útil para lograr imágenes fotorrealistas, especialmente en escenas donde las superficies difusas juegan un papel importante, como interiores con luz indirecta

La técnica de radiosidad fue desarrollada principalmente en el ámbito de la arquitectura y la simulación de luz en los años 1950 y 1960, como una extensión de la teoría de transferencia de calor radiante. Sin embargo, su adaptación al campo de la computación gráfica ocurrió más tarde, en los años 1980, a medida que los investigadores buscaban formas más precisas de representar la luz en las imágenes generadas por ordenador.

Uno de los primeros trabajos importantes en este campo fue realizado por Goral, Torrance, Greenberg y Battaile en un artículo de 1984 titulado *Modeling the Interaction of Light Between Diffuse Surfaces.*

El método de radiosidad se basa en la idea de que la luz se refleja de manera difusa entre las superficies. Cada superficie en una escena es tratada como un emisor y receptor de luz. El proceso involucra los siguientes pasos:

1. Descomposición de la escena en pequeñas superficies o "patches": Se divide el entorno en pequeñas áreas llamadas "patches", que son lo suficientemente pequeñas como para aproximar el intercambio de luz de manera precisa.
2. Cálculo de la energía emitida por cada superficie: Se calcula cuánta luz es emitida por una superficie hacia otras superficies de la escena. Esto se expresa mediante una ecuación de radiosidad que tiene en cuenta:
    - La radiosidad (cantidad de luz reflejada por la superficie).
    - La emitancia (cantidad de luz emitida por la superficie).
    - El factor de forma entre superficies, que representa cómo una superficie "ve" a otra, dependiendo de su orientación y distancia.
3. Iteración para distribuir la luz: Se realiza un cálculo iterativo que distribuye la energía de una superficie a otras. Este proceso se repite hasta que la cantidad de luz transferida entre superficies se estabiliza.
4. Visualización: Después de calcular la cantidad de luz reflejada por cada superficie, los resultados se utilizan para iluminar la escena de manera más precisa y detallada.

Entre las ventajas de la radiosidad se tienen:

Ventajas:

- La radiosidad captura los efectos de luz indirecta de manera precisa, lo que genera imágenes con un alto nivel de realismo en escenas con superficies difusas.
- Los cálculos realizados son independientes de la vista, lo que significa que una vez calculada la radiosidad, la información de iluminación puede ser utilizada desde cualquier ángulo o perspectiva.

Desventajas:

- Es un proceso computacionalmente intensivo debido a la gran cantidad de cálculos necesarios para modelar el intercambio de luz entre todas las superficies.
- El método no maneja bien las superficies especulares (reflejos directos) ni la transmisión de luz a través de objetos transparentes [1] [26].

2.4.2 La ecuación de renderizado

Dos años después de que se introdujera la Radiosidad en el campo de la computación gráfica, James Kajiya introdujo la Ecuación de Renderizado en su artículo titulado "The Rendering Equation" en 1986, un trabajo fundamental que lo cambió todo y sigue siendo hasta la fecha el trabajo más revolucionario en renderizado.

Esta ecuación describe cómo la luz interactúa con las superficies en un entorno, lo que permite generar simular la iluminación tanto la difusa como la especular de forma eficiente. El objetivo principal de esta ecuación es calcular la radiancia que se refleja o emite desde un punto hacia el ojo o la cámara en una escena y el transporte de la radiancia en los múltiples choques que sufre un rayo de luz.

La ecuación de renderizado se puede expresar matemáticamente como:

$$L_o(\vec{p}, \overrightarrow{\omega_o}) = L_e(\vec{p}, \overrightarrow{\omega_o}) + \int_\Omega f_r(\vec{p}, \overrightarrow{\omega_i}, \overrightarrow{\omega_o}) L_i(\vec{p}, \overrightarrow{\omega_i})(\overrightarrow{\omega_i} \cdot \vec{n}) d\omega \qquad (2.13)$$

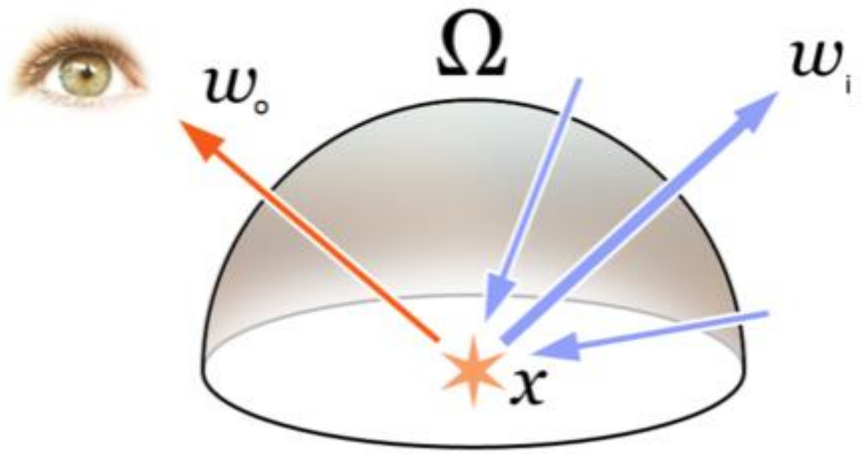

Figura 2.9 Ecuación del renderizado **[27]**.

Esta ecuación tiene varias componentes importantes, cada una de las cuales contribuye a describir cómo se distribuye la luz. A continuación, se desglosan cada uno de los términos, la mayoría introducidos en capítulos anteriores:

1. $L_o(\vec{p}, \overrightarrow{\omega_o})$ - Radiancia saliente: Este es el término que se desea calcular. Representa la radiancia (cantidad de luz por unidad de área y en una dirección específica) que sale de un punto $\vec{p}$ en una dirección $\overrightarrow{\omega_o}$ hacia la cámara o el observador.
2. $L_e(\vec{p}, \overrightarrow{\omega_o})$- Radiancia emitida: Este término describe cualquier luz que sea emitida directamente por el punto $\vec{p}$ en la dirección $\overrightarrow{\omega_o}$. Por ejemplo, si el punto está en una superficie emisora de luz (como una lámpara o una pantalla), esta sería la luz que emite dicha fuente en esa dirección.
3. $\int_\Omega d\omega$ - Integral sobre el hemisferio: Este es el término clave de la ecuación. Representa la luz que llega al punto $\vec{p}$ desde todas las direcciones dentro del hemisferio superior de dicho punto. Esta luz se refleja y contribuye a la radiancia saliente Lo. La integral tiene en cuenta todas las direcciones incidentes $\overrightarrow{\omega_i}$ en el hemisferio Ω.
4. $f_r(\vec{p}, \overrightarrow{\omega_i}, \overrightarrow{\omega_o})$– BRDF: La función de distribución bidireccional de reflectancia que modela cómo la luz que llega desde una dirección $\overrightarrow{\omega_i}$ se refleja en la superficie hacia otra dirección $\overrightarrow{\omega_o}$. Es una función que depende tanto de la dirección de la luz incidente como de la dirección de la luz reflejada. Las características físicas de la superficie (rugosidad, color, reflectancia, etc.) influyen en esta función.

5. $L_i(\vec{p}, \overrightarrow{\omega_i})$ - Radiancia incidente: Es la radiancia que llega al punto $\vec{p}$ desde una dirección $\overrightarrow{\omega_i}$. Esta luz puede venir directamente desde una fuente luminosa o ser luz reflejada por otras superficies de la escena.
6. $(\overrightarrow{\omega_i} \cdot \vec{n})$ - Factor de coseno: El producto punto $\overrightarrow{\omega_i}$·n es el coseno del ángulo entre la dirección de la luz incidente $\overrightarrow{\omega_i}$y la normal de la superficie en el punto $\vec{p}$. Este factor es transcendente porque describe cómo la luz que llega en ángulos oblicuos aporta menos radiancia que la luz que llega perpendicularmente. Este efecto se conoce como la ley del coseno de Lambert ya mencionada tantas veces: cuando la luz incide en un ángulo más inclinado, se dispersa más y menos energía llega a la superficie por unidad de área. Es el producto escalar que introdujo Phong para la parte difusa.
7. dω - Diferencial del ángulo sólido: Este es un elemento de área en el espacio direccional, que describe un pequeño ángulo en el hemisferio sobre el punto p. La integral sumatoria tiene que considerar todas las direcciones en ese hemisferio, por lo que este diferencial permite sumar las contribuciones de la luz desde cada dirección.

La ecuación del renderizado integra la radiancia incidente en un punto mediante un hemisferio. La radiancia se transporta en rayos hasta llegar a la cámara, teniendo en cuenta todos los rebotes. Es una ecuación recursiva puesto que la radiancia incidente es la radiancia saliente de otros rebotes. Para calcular tal integral se usan métodos numéricos por ordenador, en concreto el método de Montecarlo.

Algunos de los conceptos clave se mencionaron en secciones anteriores, como los conceptos radiométricos. A continuación, se clarifican estos otros:

1. Hemisferio de integración: La integral se realiza sobre un hemisferio (Ω) centrado en el punto p, porque el punto solo recibe luz desde la mitad del espacio (es decir, el hemisferio visible). La superficie del hemisferio sobre $\vec{p}$ representa todas las direcciones posibles desde las cuales puede llegar luz.
2. Trazado de caminos y Montecarlo: Como se mencionó antes, la ecuación de renderizado es imposible de resolver de manera analítica debido a su complejidad (particularmente por la integral). Aquí es donde el método de Montecarlo y el trazado de rayos juegan un papel clave. Al muestrear

aleatoriamente diferentes direcciones $\overrightarrow{\omega_i}$ y promediar los resultados, el método de Montecarlo permite aproximar la solución de la ecuación de renderizado.

3. Interacción entre fuentes y reflejos: La ecuación de renderizado considera tanto la luz directa de una fuente como la luz que ha sido reflejada varias veces en una escena antes de llegar al punto de interés. Esto es lo que permite calcular de manera realista fenómenos complejos como sombras suaves, reflexión difusa global y causticas.
4. Se introducen luces de área puesto que se tiene en consideración la radiancia emitida de los objetos en la ecuación.
5. Varianza: la varianza es el ruido que puede presentar una imagen final. Un número infinito de muestras ofrecerían una solución perfecta. Existen técnicas para acelerar la convergencia a la solución final y por tanto reducir la varianza [28].

El ángulo sólido: es una medida tridimensional del "área angular" que ocupa un objeto en una esfera centrada en un punto dado. Este ángulo se mide en esteradianes (sr) y se define como la proyección de un área en una esfera unitaria. La fórmula general para calcular el ángulo sólido, Ω, en esteradianes es:

$$\Omega = \frac{A}{r^2} \tag{2.25}$$

Donde:

- Ω es el ángulo sólido en esteradianes.
- A es el área proyectada del objeto sobre la superficie de una esfera.
- r es el radio de la esfera.

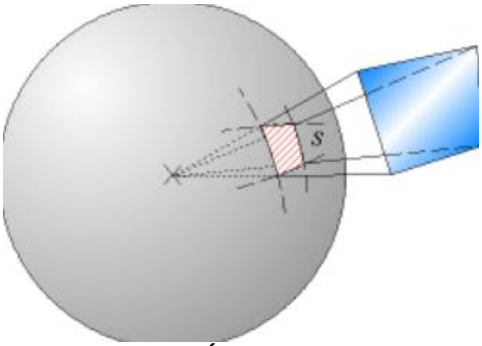

Figura 2.10 Ángulo sólido **[29]**.

2.5 El método de Montecarlo

El método de Montecarlo es una técnica numérica basada en el uso de muestras aleatorias para aproximar soluciones a problemas matemáticos que pueden ser difíciles o imposibles de resolver de manera analítica. Este método se utiliza en una amplia gama de disciplinas, incluyendo física, matemáticas, economía...

El método de Montecarlo fue desarrollado en los años 1940, durante la segunda guerra mundial, por científicos que trabajaban en el proyecto Manhattan. Uno de ellos, Stanislaw Ulam, lo nombró "Montecarlo" en referencia al famoso casino en Mónaco, debido a la naturaleza aleatoria del juego y a las similitudes con el proceso de muestreo aleatorio.

En el contexto del renderizado y la computación gráfica, Montecarlo se usa principalmente para resolver integrales complejas que modelan la luz, como la ecuación de renderizado. Los problemas de renderizado generalmente involucran la integración sobre muchas dimensiones (como las múltiples direcciones en las que la luz puede incidir y reflejarse en una superficie) el método de Montecarlo se convierte en una herramienta poderosa para abordar esta complejidad.

Algunas características del método de Montecarlo son:

1. Aleatoriedad y muestreo: El método de Montecarlo depende de la generación de muestras aleatorias para estimar una solución. La idea básica es que, en lugar de intentar calcular una solución exacta (como una integral) de forma determinista, podemos muestrear valores de la función a resolver de manera aleatoria, promediar esos valores y así obtener una estimación aproximada del resultado.

En el caso de la ecuación de renderizado, estas muestras aleatorias podrían ser las direcciones desde las que la luz incide en una superficie. En lugar de evaluar la luz en todas las direcciones posibles (lo que sería computacionalmente prohibitivo), se seleccionan algunas direcciones de manera aleatoria y se promedia la contribución de la luz que llega desde estas direcciones.

2. Ley de los grandes números: El fundamento teórico detrás del método de Montecarlo es la ley de los grandes números, que establece que, a medida que el número de muestras aleatorias aumenta, la media de estas muestras converge a la media real del conjunto completo. Esto significa que, aunque una estimación con pocas muestras puede ser inexacta, a medida que se toman más muestras, la estimación se vuelve más precisa.

En términos de renderizado, esto implica que cuantos más rayos de luz simulamos en nuestra escena, más precisa será la imagen final. Sin embargo, también es cierto que el incremento en precisión se ralentiza conforme aumenta el número de muestras.

3. Función de densidad de probabilidad (PDF): En Montecarlo, la función de densidad de probabilidad es necesaria para realizar muestreo eficiente. Esta función define cómo se distribuyen las muestras en el espacio. Por ejemplo, en el renderizado, podría tener más sentido muestrear más direcciones donde hay una aportación de luz más significativa dependiendo de las propiedades de la superficie, lo que reduce la varianza.

Las técnicas de muestreo son esenciales en el método de Montecarlo para generar muestras aleatorias de una distribución específica. Estas técnicas se utilizan en una variedad de contextos, desde la simulación de fenómenos físicos hasta la optimización en la inteligencia artificial y el renderizado. Algunas de las técnicas de muestreo más comunes son; el método de rechazo, el método de inversión y el método de transformación de distribuciones.

1. El método de rechazo es una técnica simple pero poderosa para generar muestras de una distribución objetivo complicada, utilizando una distribución más sencilla de la cual es fácil generar muestras. La idea central es generar muestras de una distribución conocida y rechazar aquellas que no se ajustan bien a la distribución objetivo.
2. El método de inversión es una técnica que se utiliza cuando se conoce la función de distribución acumulativa (CDF) de la distribución objetivo. Este método es especialmente útil cuando la CDF tiene una forma inversa conocida, permitiendo generar muestras directamente a partir de una distribución uniforme.

3. El método de transformación de distribuciones es una generalización del método de inversión. Se utiliza para transformar una muestra de una distribución más simple (por lo general, la normal estándar o la uniforme) en una muestra de una distribución más compleja a través de una transformación matemática [1] [30].

En el proyecto se ahondará más en el método de inversión.

2.6 Trazadores de caminos

Con la formulación de la ecuación del renderizado de Kajiya se crearon los primeros trazadores de caminos (Path Tracers) en los años 80. Hay diferentes variantes de trazadores de caminos, cada una con sus propias características. A continuación, se detallan sus características, lo que implica cada uno y sus fortalezas y debilidades [31]:

2.6.1 Trazador de caminos inverso

El trazador de caminos inverso (backward path tracer) es una técnica de trazado de caminos que comienza en la cámara (o el observador) y sigue los caminos de la luz hacia las fuentes de luz en la escena. Este enfoque se utiliza comúnmente debido a su eficiencia en encontrar los rayos que realmente contribuyen a la imagen final.

El trazador de caminos inverso es una solución buena, pero tiene limitaciones para obtener buenos resultados con luces pequeñas o luces que se cuelan por pequeñas rendijas o huecos e incluso cáusticas (refracciones de la luz que convergen en un punto, como un efecto de lupa).

2.6.2 Trazador de caminos directo

El trazador de caminos directo (forward path tracer) comienza en las fuentes de luz y traza los rayos hacia la escena, intentando detectar si estos alcanzan la cámara. Aunque conceptualmente sencillo, no es comúnmente utilizado por su ineficiencia en encontrar los rayos que impactan en la cámara. Este enfoque es anterior al trazado inverso, pero rápidamente fue sustituido por el método inverso debido a su falta de eficiencia.

El trazado de caminos directo es mejor que el inverso con caminos de luz complejos, como en rendijas y cáusticas, pero en términos generales presenta una mayor varianza que el trazado de caminos inverso.

2.6.3 Trazador de caminos bidireccional

El trazador de caminos bidireccional (bidirectional path tracer) combina tanto el trazador de caminos directo como el inverso. Los rayos se trazan simultáneamente desde la cámara y desde las luces, y luego se conectan en el espacio de la escena para calcular la contribución total de la iluminación.

Esta técnica fue desarrollada en los años 90 como una mejora para las situaciones en las que ni el trazado directo ni el inverso son suficientes para capturar correctamente la complejidad de la iluminación global.

El trazador de caminos bidireccional es la solución ideal, pero supone implementaciones mucho más complejas y costosas [32].

2.6.4 Comparativa general

Método	Fortalezas	Debilidades
Backward Path Tracer	Eficiente para iluminación directa; manejo estándar de escenas	Ineficiente para caminos de luz complejos
Forward Path Tracer	Simulación precisa de caminos de luz complejos	Extremadamente ineficiente, muchos rayos no alcanzan la cámara
Bidirectional Path Tracer	Eficiente para luz directa e indirecta; ideal para escenas complejas	

Tabla 2-1 Comparativa de caminados.

2.7 Next Event Estimation (NEE)

Next Event Estimation (NEE) es una técnica utilizada principalmente para mejorar la eficiencia en la simulación de la iluminación indirecta, particularmente en el trazado de rayos. Esta técnica forma parte de las soluciones para resolver la ecuación de renderizado.

La estimación del siguiente evento (NEE) es un método que se usa para mejorar la convergencia de los cálculos de iluminación en renderización, lo que significa que ayuda a reducir el ruido o varianza en las imágenes generadas mediante trazado de rayos, especialmente cuando se trata de fuentes de luz de área.

En el ray tracer de Montecarlo, un gran desafío es calcular cómo la luz de una fuente llega a un punto específico en una escena después de reflejarse o refractarse en varias superficies. Sin NEE, el trazador de rayos dependería de la pura suerte para encontrar un rayo que golpee una fuente de luz de área, lo que podría requerir muchos intentos para obtener una estimación precisa, resultando en imágenes ruidosas y en un rendimiento ineficiente.

NEE reduce la varianza en los cálculos de iluminación mediante la técnica de muestreo de la fuente de luz. En lugar de esperar a que un rayo reflejado al azar golpee una luz de área, NEE introduce una técnica en la que se selecciona directamente una fuente de luz cercana (o la siguiente fuente de luz más relevante), y se lanza un rayo hacia ella para evaluar si contribuye a la iluminación en un punto determinado.

Este enfoque se puede dividir en los siguientes pasos:

1. Muestreo de luz: Se elige una luz de la escena, a menudo utilizando una distribución de probabilidad para seleccionar luces más relevantes.
2. Cálculo de visibilidad: Se lanza un rayo desde el punto en la escena hacia la luz seleccionada para determinar si está visible o si está bloqueada por otros objetos.
3. Cálculo de iluminación: Si la luz es visible, se calcula la cantidad de luz que llega al punto y cómo se refleja o refracta en la escena [33].

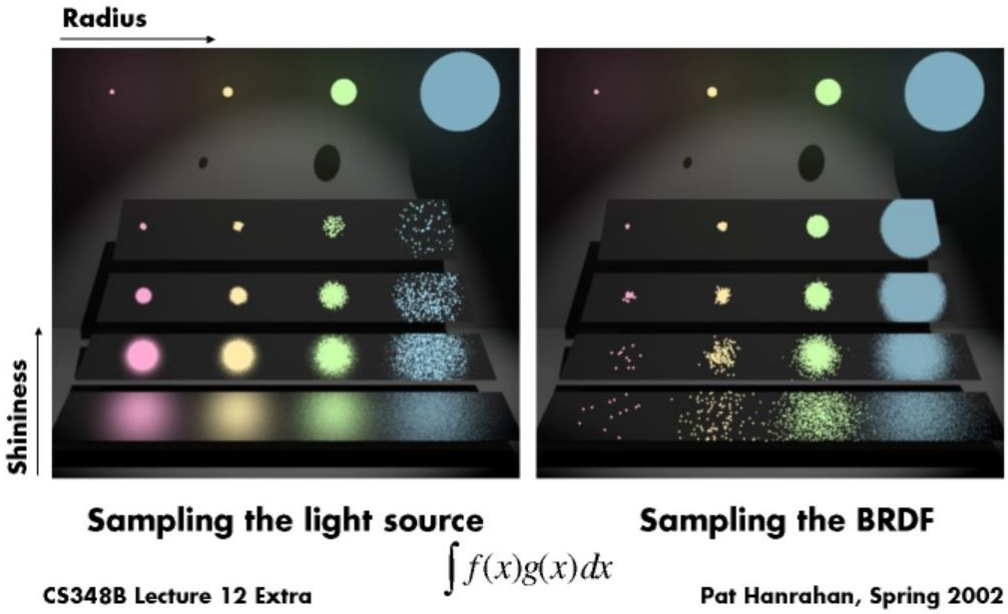

Figura 2.11 BRDF vs NEE [34].

2.8 Russian roulette

La russian roulette es una técnica propia del método de Montecarlo usada para optimizar el cálculo de iluminación global y reducir el costo computacional de simular reflejos y refracciones en escenas complejas. Esta técnica no está relacionada con el famoso juego de azar, sino que comparte su nombre debido a su uso de la probabilidad en la toma de decisiones para optimizar cálculos.

El russian roulette, en el contexto de gráficos por computadora, fue introducido como parte del algoritmo de Montecarlo para simulaciones de trayectorias de rayos, en el artículo de James T. Kajiya "The Rendering Equation".

En el trazado de rayos, cada rayo puede interactuar con varias superficies antes de llegar a la cámara, reflejándose o refractándose, lo que genera una cadena potencialmente infinita de cálculos. Para evitar tener que seguir cada rayo infinitamente, la técnica de russian roulette decide de manera probabilística si un rayo debe seguirse o terminarse en un punto determinado, ahorrando tiempo de cómputo. Si un rayo se termina, se le asigna un peso (probabilidad) que ajusta su contribución a la imagen final.

Esta técnica es útil para manejar situaciones donde hay muchos rayos secundarios que contribuyen muy poco a la iluminación final, como en áreas de sombras o reflejos oscuros. En lugar de calcular cada uno de estos rayos, russian roulette permite terminarlos de forma anticipada según una probabilidad determinada [35].

2.9 Más allá de la BRDF

Además de la BRDF (Bidirectional Reflectance Distribution Function), existen otros conceptos clave relacionados con la reflexión y la refracción en el ámbito de la simulación de materiales y el renderizado. A continuación, se mencionan algunos de los más importantes:

2.9.1 Bidirectional Transmittance Distribution Function (BTDF)

La BTDF es la contraparte de la BRDF para la transmisión de luz. Mientras que la BRDF describe cómo la luz es reflejada por una superficie, la BTDF describe cómo la luz atraviesa un material y emerge en otra dirección. Este concepto sirve para simular materiales transparentes o translúcidos, como el vidrio, el agua o el plástico.

2.9.2 Bidirectional Scattering Distribution Function (BSDF)

La BSDF es una función general que engloba tanto la BRDF como la BTDF. Se utiliza para describir cómo la luz se dispersa en todas las direcciones debido a una superficie, considerando tanto la reflexión como la transmisión. En pocas palabras, la BSDF combina la reflectancia y la transmitancia de la luz.

2.9.3 Bidirectional Surface Scattering Reflectance Distribution Function (BSSRDF)

La BSSRDF extiende el concepto de BRDF al considerar no solo la luz que se refleja directamente en el punto de incidencia, sino también la luz que penetra en el material y luego emerge en otro punto. Este tipo de dispersión es importante para materiales translúcidos como la piel humana, la cera o el mármol, donde la luz entra en el material, se dispersa internamente y sale por otra parte de la superficie.

Este fenómeno de dispersión interna se conoce como Subsurface Scattering (SSS), y es esencial para representar materiales que tienen una apariencia suave o translúcida [36].

2.10 Medios participativos o ecuación de transferencia radiativa

Además, existen técnicas para simular cómo la luz viaja a través de distintos medios como la atmósfera, el humo o el agua. A estas técnicas se las conocen como medios participativos (Participating Media). Los medios participativos son aquellos que no solo afectan la luz directamente, sino que también interactúan con ella de manera más compleja, dispersando, absorbiendo y emitiendo luz a lo largo de un volumen de espacio. Esta interacción se suele implementar mediante ray marching, calculando paso a paso cómo el rayo se modifica teniendo en cuenta:

- La Dispersión de la luz, se refiere a cómo la luz es redirigida cuando interactúa con partículas dentro de un medio participativo. Se clasifican en dos tipos principales:
    - Dispersión isotrópica: La luz se dispersa de manera uniforme en todas las direcciones.
    - Dispersión anisotrópica: La luz tiene una preferencia direccional al dispersarse, siendo más común en materiales como el humo o la niebla, donde la luz tiende a dispersarse hacia adelante o hacia atrás.
- La absorción se refiere a la cantidad de luz que un medio absorbe a medida que pasa a través de él. En medios altamente absorbentes, como el agua oscura o ciertos gases densos, la luz se atenúa significativamente. Este fenómeno está controlado por el coeficiente de absorción, que varía en función del material y su densidad.
- La emisión: Algunos medios participativos pueden emitir luz, lo que se llama auto emisión. Un ejemplo común sería una nube de gas caliente o plasma que emite luz por sí misma o debido a la fluorescencia. La luz emitida por el medio puede interactuar con otros elementos de la escena, y esto puede ser complejo de simular para obtener resultados realistas.

- La transmitancia: describe cómo la luz viaja a través de un medio y cuánto de ella se mantiene sin ser absorbida o dispersada. Este valor es esencial para calcular la visibilidad a través de medios como niebla densa o atmósferas complejas.
- La función de fase describe la probabilidad de que la luz se disperse en una dirección particular cuando interactúa con una partícula dentro de un medio. Para medios participativos, las funciones de fase son críticas para determinar el comportamiento de la luz dispersada. La función de fase puede ser isotrópica o anisotrópica, dependiendo de la naturaleza del medio [37].

2.11 Photon mapping

Photon mapping es una técnica de simulación de iluminación global sesgada. Photon mapping fue desarrollado por el investigador danés Henrik Wann Jensen en el año 1995. La técnica fue publicada por primera vez en su tesis doctoral en 1996. El objetivo principal de Jensen con este método era abordar las limitaciones de las técnicas de iluminación global más tradicionales, como el radiosity y el ray tracing básico, mejorando la eficiencia en el cálculo de la iluminación indirecta y los efectos ópticos avanzados.

Photon mapping se basa en dos fases principales:

1. Emisión y almacenamiento de fotones: En esta fase, se lanzan fotones desde las fuentes de luz en la escena, simulando cómo estos interactúan con los objetos (reflejan, refractan o se absorben). Cada interacción de los fotones se almacena en un "mapa de fotones" (un espacio de datos especializado) que captura la energía y la dirección de los fotones en varios puntos de la escena.
2. Renderizado: Durante esta fase, se utiliza el mapa de fotones almacenado para estimar la cantidad de luz que llega a cualquier punto de la escena. Cuando se trazan rayos desde la cámara para calcular el color de los píxeles, el algoritmo consulta el mapa de fotones para determinar cuánta luz indirecta afecta el punto de impacto del rayo. Esto permite reproducir con precisión efectos como

la caustica (la luz concentrada en superficies por la refracción o reflexión) y la iluminación suave de la luz indirecta.

Photon mapping es especialmente valioso en la simulación de iluminación indirecta, donde la luz rebota de un objeto a otro, y para generar causticas, que son los patrones de luz concentrada que se forman cuando la luz atraviesa o se refleja en superficies transparentes o brillantes. Debido a su capacidad para manejar estos efectos de manera eficiente, ha sido utilizado en aplicaciones que requieren alta calidad visual, como el cine, la arquitectura y los videojuegos.

Aunque photon mapping es más eficiente que otras técnicas en algunos casos, sigue siendo intensivo en términos de cálculo y puede requerir grandes cantidades de memoria y tiempo de procesamiento, especialmente para escenas complejas.

Photon mapping es una técnica de renderizado que introduce sesgo (bias) debido a las aproximaciones que utiliza para hacer más manejable el cálculo de la luz. Esto incluye la búsqueda de fotones cercanos y el suavizado de la iluminación, lo que puede afectar la precisión, pero mejora la eficiencia.

En comparación, photon mapping es más rápido que el path tracing (unbiased) pero introduce sesgo, mientras que path tracing es más preciso, pero computacionalmente más costoso. A veces se usan combinadamente, para paliar las limitaciones del backward ray tracing [38].

2.12 Multiple Importance Sampling (MIS)

Multiple Importance Sampling (MIS) es una técnica de muestreo numérico introducida por primera vez por Éric Veach en su tesis doctoral titulada "Robust Montecarlo Methods for Light Transport Simulation" en 1997. Esta técnica se desarrolló para mejorar la convergencia de los métodos Montecarlo.

El problema surge cuando se tiene más de una función de densidad de probabilidad (PDF) potencialmente útil para muestrear la misma integral. Si se elige mal la PDF, se puede aumentar la varianza, llevando a imágenes con ruido o artefactos. El objetivo

de multiple importance sampling es combinar varias estrategias de muestreo (cada una con su propia PDF) de manera que se minimice la varianza en la estimación final.

El concepto central de MIS es combinar los resultados de varios métodos de muestreo utilizando pesos de importancia para reducir la varianza. En lugar de usar solo una PDF, MIS permite usar varias, asignando pesos a cada uno de los métodos de muestreo en función de su relevancia para diferentes partes de la función que se está integrando.

Los pesos se calculan de tal manera que se reduzca el impacto de las muestras ineficaces (las que contribuyen menos a la estimación precisa) y se potencie el impacto de las muestras más eficaces. El resultado es una estimación más precisa de la integral con menos ruido en el resultado final.

Ventajas de MIS:

1. Reducción de la varianza: Al combinar diferentes estrategias de muestreo, MIS puede reducir significativamente la varianza en la estimación, lo que lleva a imágenes más limpias y menos ruido en simulaciones.
2. Robustez: No depende exclusivamente de una estrategia de muestreo, lo que le permite adaptarse mejor a diferentes escenarios, ya que en algunos casos un método de muestreo puede ser más eficiente que otro.
3. Versatilidad: Puede aplicarse a una amplia variedad de problemas que involucran integración, no solo en gráficos por computadora, sino también en otros campos como la física computacional y el procesamiento de señales [39].

2.13 Metropolis Light Transport (MLT)

Metropolis Light Transport (MLT) es un algoritmo avanzado para el cálculo de la iluminación global en gráficos por computadora. Metropolis Light Transport fue un artículo escrito por Eric Veach y Leonidas J. Guibas en el año 1997. El trabajo de Veach introdujo varias técnicas innovadoras para la simulación de la propagación de la luz, con un enfoque en la eficiencia de los algoritmos de Montecarlo en escenas complejas.

MLT es una variación del algoritmo de path tracing. Metropolis light transport usa un enfoque basado en la teoría de Markov Chain Monte Carlo (MCMC). Esto permite que el algoritmo explore más eficientemente las trayectorias de luz en una escena, particularmente en aquellas con condiciones difíciles, como luces indirectas complejas, superficies especulares o causticas [40].

El algoritmo funciona de la siguiente manera:

1. Exploración de caminos de luz: En lugar de trazar rayos desde la cámara de forma aleatoria y esperar que intersequen con la fuente de luz, el algoritmo de MLT busca caminos de luz más eficientes mediante una cadena de Markov. Esto significa que se genera un camino inicial y, a partir de este, se crean variaciones pequeñas para explorar otras posibles rutas.
2. Mutaciones de caminos: Una vez que se encuentra un camino de luz válido, el algoritmo muta ligeramente ese camino para crear variantes, que también se evalúan en función de su contribución a la imagen final. Las mutaciones permiten que el algoritmo explore eficientemente las áreas difíciles de la escena (por ejemplo, luz reflejada en múltiples superficies).
3. Mejora de la eficiencia: Gracias a la naturaleza de la cadena de Markov, MLT puede enfocarse en las partes de la escena donde la luz es más compleja y difícil de calcular, mejorando drásticamente la eficiencia en comparación con otros métodos de Montecarlo tradicionales.

Ventajas y aplicaciones

- Eficiencia en escenas difíciles: MLT es especialmente eficiente en situaciones donde las técnicas de Montecarlo convencionales fallan o requieren un número extremadamente grande de muestras, como las causticas (efectos de luz concentrada, como los destellos de luz a través de un vaso).
- Escenas con iluminación compleja: Permite renderizar escenas con múltiples tipos de interacciones de luz (reflejos, refracciones, etc.) de manera más realista [41].

2.14 Estructuras de aceleración

En el renderizado offline, las estructuras de aceleración son estructuras de datos que mejoran la eficiencia de los cálculos de intersección de rayos. Estas estructuras organizan los objetos de una escena de tal manera que se puedan descartar grandes grupos de objetos rápidamente, evitando verificar las intersecciones con todos los objetos de la escena. Esto es decisivo para reducir el número de cálculos y hacer manejables los tiempos de renderizado en escenas complejas, especialmente cuando hay muchos polígonos.

2.14.1 Bounding box

Una bounding box es una caja rectangular tridimensional que encierra un objeto o conjunto de objetos. Su función es delimitar el volumen ocupado por estos, lo que permite descartar de forma rápida las intersecciones de rayos cuando no atraviesan esta caja. Hay dos tipos principales:

- Axis-Aligned Bounding Boxes (AABB): están alineadas con los ejes de coordenadas.
- Oriented Bounding Boxes (OBB): pueden rotarse para ajustarse mejor al objeto.

Las bounding boxes son eficientes al reducir los cálculos necesarios, pero en escenas muy complejas, su uso individual se vuelve ineficiente. Por ello, se combinan en estructuras más avanzadas, como las Bounding Volume Hierarchies (BVH).

2.14.2 Bounding Volume Hierarchies (BVH)

- Descripción: Una BVH organiza bounding boxes en una jerarquía de volúmenes. Cada nodo en la jerarquía contiene una bounding box que engloba varios objetos o subnodos. Los rayos sólo necesitan verificar los volúmenes donde podría haber intersecciones, descartando ramas completas de la jerarquía si no hay intersección con la caja en un nivel superior.

- Ventaja: Reduce significativamente los cálculos necesarios al descartar grandes grupos de objetos de una sola vez. Es altamente eficiente en escenas dinámicas, ya que permite actualizaciones parciales.
- La estructura mejora si en lugar de usar bounding boxes se ajustan los objetos a estructuras como bounding volumes intersecados que reducen los espacios vacíos que producen otras estructuras sin componer.

2.14.3 KD-Trees (K-Dimensional Trees)

- Descripción: Un KD-Tree divide el espacio tridimensional de manera recursiva en dos mitades en cada nodo, usando planos ortogonales. Cada objeto se asigna a uno de los subespacios resultantes.
- Ventaja: Muy eficiente en escenas estáticas, ya que los rayos solo deben verificar los objetos en el subespacio que atraviesan.
- Desventaja: No es adecuado para escenas dinámicas, ya que las actualizaciones del árbol son costosas.

2.14.4 Grids (Uniform grids)

1. Descripción: Divide el espacio de la escena en una cuadrícula regular donde cada celda contiene un subconjunto de objetos. Los rayos interactúan solo con los objetos en las celdas que atraviesan.
2. Ventaja: Sencillas de implementar y rápidas en el acceso a las celdas.
3. Desventaja: Ineficientes si los objetos no están distribuidos de manera uniforme, lo que provoca que algunas celdas contengan demasiados objetos y otras muy pocos.

2.14.5 Octrees

1. Descripción: Un octree subdivide el espacio en ocho partes o "octantes", y a medida que un rayo viaja por la escena, solo verifica los objetos en el octante correspondiente.
2. Ventaja: Más eficientes que las cuadrículas en escenas no uniformemente distribuidas, ya que ajustan el tamaño de los octantes según la densidad de objetos.

3. Desventaja: Su implementación es más compleja y su eficiencia depende de cómo se subdivida la escena [42] [43].

2.15 El Presente y futuro del renderizado offline

El presente y el futuro del renderizado offline sigue centrado en lograr una fidelidad visual extrema, con énfasis en la simulación física precisa y aumentar la velocidad de cómputo de los renderizados disminuyendo la varianza. El campo está relativamente estancado en comparación con momentos de la historia anteriores, pero ha habido avances significativos a la hora de mejorar los tiempos de cómputo al hacer uso de procesamiento en paralelo por GPU y CPU, SIMD y cálculos en la nube. Además, se han introducido nuevas técnicas que cachean la irradiancia para reducir los tiempos de cómputo. Se ha hecho hincapié en mejorar las interacciones de la luz teniendo en cuenta los espectros reflejados. Además, recientemente se están implementando técnicas híbridas basadas en redes neuronales para reducir la varianza.

Algunas de las investigaciones recientes más importantes son:

2.15.1 Stochastic Progressive Photon Mapping (SPPM) (2008)

Stochastic Progressive Photon Mapping fue introducido en 2008 por Hachisuka, Jensen y otros. Es una mejora sobre el photon mapping tradicional, que se adapta bien a escenas con causticas, reflejos especulares y refracciones. SPPM utiliza un proceso progresivo para refinar el cálculo de la iluminación global y reducir el error de manera continua, lo que permite obtener imágenes más precisas con cada iteración, al tiempo que mejora la eficiencia en comparación con el photon mapping estándar [44].

2.15.2 V-Ray's irradiance cache (2002)

Irradiance caching ha sido una técnica decisiva en la reducción de costos computacionales en iluminación global, y su implementación en V-Ray a principios de la década de 2000 ha sido una de las innovaciones más influyentes. Aunque no es nueva en su concepto, las versiones avanzadas y optimizadas de esta técnica, desarrolladas y aplicadas en motores de render como V-Ray, han permitido obtener

renderizados mucho más eficientes en producciones comerciales, manteniendo alta calidad visual [45].

2.15.3 Lightcuts (2005)

Presentado por Walter, Fernand y Bala en 2005, Lightcuts es una técnica para gestionar escenas con una gran cantidad de fuentes de luz, un problema común en el renderizado de producciones cinematográficas y de efectos visuales. Lightcuts crea un árbol jerárquico que agrupa fuentes de luz similares y ajusta dinámicamente la cantidad de luces que se evalúan en cada píxel, reduciendo así los costos computacionales sin comprometer la calidad de la iluminación [46].

2.15.4 Denoising techniques (Post 2010)

A partir de 2010, las técnicas de denoising comenzaron a ganar importancia, especialmente con el auge de los métodos de path tracing. Estas técnicas permiten reducir el número de muestras por píxel necesarias al aplicar algoritmos que eliminan el ruido residual de imágenes generadas con pocas muestras. Algunas implementaciones notables incluyen el uso de filtros inteligentes y, más recientemente, el uso de redes neuronales para denoising, lo que ha permitido acelerar significativamente el proceso de renderizado sin pérdida significativa de calidad visual [47].

2.15.5 Machine learning y neural rendering (Post 2015)

Neural rendering es un campo emergente que ha comenzado a ganar tracción en los últimos años. Estas técnicas utilizan redes neuronales para predecir ciertos aspectos de la escena, como iluminación indirecta o sombras, basándose en patrones aprendidos de datos previos. Algunas implementaciones de machine learning y redes neuronales profundas también se están utilizando para mejorar procesos como el denoising, super-resolución y la predicción de iluminación. Aunque aún está en desarrollo y no es tan ubicua como otras técnicas, tiene el potencial de transformar la eficiencia del renderizado offline en los próximos años [48].

2.15.6 Path guiding (2017)

Path guiding es una técnica más reciente introducida alrededor de 2017 que optimiza el proceso de path tracing mediante la recolección de información sobre cómo se comportan los rayos en una escena para guiar el trazado de futuros rayos de manera más eficiente. A diferencia de los métodos tradicionales de Montecarlo, que trazan rayos de manera uniforme, path guiding utiliza información acumulada para dirigir los rayos hacia las áreas más relevantes de la escena, reduciendo significativamente el ruido en escenas con iluminación compleja o con superficies reflectivas y refractivas. Esta técnica está ganando popularidad en los sistemas de renderizado offline por su capacidad para mejorar la eficiencia sin sacrificar el realismo [49].

2.15.7 Spectral rendering (2010s)

El Spectral rendering ha experimentado un resurgimiento en la última década debido a la demanda de un realismo físico extremo en la simulación de la luz. El espectral rendering tiene en cuenta la interacción de los materiales con las diferentes longitudes de onda de la luz, proporcionando una mayor precisión en fenómenos como la dispersión de sub-superficie, causticas complejas y el comportamiento de materiales como el vidrio y el metal. Aunque es computacionalmente costoso, ha sido adoptado en aplicaciones científicas y en producciones de cine de alto presupuesto que requieren un realismo absoluto [50].

2.16 Escenas y objetos de prueba

A lo largo de la historia del renderizado computacional, ha habido varias escenas de prueba que han sido fundamentales para estudiar y demostrar los avances en técnicas de renderizado. La más famosa de estas escenas es la Cornell Box, pero hay otras igualmente importantes. Aquí se mencionan algunas de las más relevantes: [51]

2.16.1 Cornell box (1984)

Historia: La Cornell box fue creada por investigadores de la Universidad de Cornell en 1984 como una escena estándar para estudiar el fenómeno de la iluminación global

y validar algoritmos de renderizado, especialmente los que involucraban el cálculo de la reflexión y la refracción de la luz. La escena original consistía en una caja con paredes de colores (roja, verde y blanca) y dos objetos simples, típicamente un cubo y una esfera.

Figura 2.12 Cornell box **[52]**.

Importancia: Sirve como una referencia estándar para medir la exactitud de los algoritmos de iluminación global, ya que los materiales, las luces y las propiedades geométricas están perfectamente definidas.

2.16.2 Teapot de Utah (1975)

Historia: Este es uno de los objetos 3D más emblemáticos en la historia de la computación gráfica. Fue modelado por Martin Newell en la Universidad de Utah y se ha utilizado extensamente en la investigación del renderizado. La tetera es un objeto complejo con curvas suaves y variaciones en su superficie, lo que lo convierte en un desafío interesante para la representación visual precisa. Importancia: Aunque no es una escena completa como la Cornell Box, la Tetera de Utah se convirtió en un modelo estándar para probar algoritmos de iluminación, sombreado y texturizado, gracias a su forma compleja y reconocible.

Figura 2.13 Utah teapot **[53]**.

2.16.3Sponza atrium (2002)

Historia: El Sponza atrium es una escena creada por Marko Dabrovic como parte de la conferencia de gráficos computacionales Eurographics. Representa una estructura renacentista con arcos, columnas y texturas detalladas. Se ha convertido en una escena estándar debido a su complejidad y riqueza geométrica, así como sus propiedades físicas y de iluminación complejas.

Importancia: Es muy utilizada para probar efectos de iluminación global y simulación de ray tracing, ya que tiene múltiples fuentes de luz indirecta y geometría variada, lo que hace que el comportamiento de la luz sea un desafío realista.

Figura 2.14 Sponza atrium **[54]**.

2.16.4San Miguel (2013)

Historia: Esta escena representa una réplica detallada de un mercado al aire libre con texturas complejas, geometría realista y una rica interacción de luz. Fue creada por Morgan McGuire y fue utilizada inicialmente para probar sistemas de renderizado de path tracing.

Importancia: La escena es significativa por su alta densidad de detalles y texturas, proporcionando un campo de pruebas excelente para algoritmos de renderizados modernos como el path tracing, que requiere una simulación precisa de luz, materiales y sombras.

Figura 2.15 Escena de San Miguel **[55]**.

2.16.5Barcelona pavilion (2008)

Historia: Modelada a partir del pabellón diseñado por Ludwig Mies van der Rohe para la Exposición Internacional de Barcelona de 1929, esta escena es utilizada por la comunidad gráfica debido a su arquitectura moderna y minimalista. Contiene superficies reflectantes como mármol, vidrio y agua, lo que la convierte en una escena excelente para probar efectos de reflexión, refracción y causticas.

Importancia: Es útil para probar algoritmos de iluminación indirecta, ya que presenta tanto grandes áreas planas con luz difusa como superficies reflectantes complejas que exigen una simulación realista.

Figura 2.16 Barcelona pavilion **[56]**.

2.16.6 Bistro scene (2017)

Historia: Esta escena fue presentada por NVIDIA como parte de sus pruebas para el renderizado de ray tracing en tiempo real, específicamente para mostrar las capacidades de su arquitectura RTX. Es una representación rica en detalles de un bistró parisino, con materiales variados, luces volumétricas y reflejos complejos.

Importancia: Sirve como referencia moderna para la evaluación de la calidad y velocidad del ray tracing, especialmente en el contexto de videojuegos y simulaciones en tiempo real.

Estas escenas y modelos se han utilizado para evaluar la fidelidad visual de distintos algoritmos de renderizado, ayudando a definir estándares en la industria de gráficos computacionales. [57]

Figura 2.17 Escena de Bistro **[58]**.

Hay además varios modelos estándar icónicos que se utilizan para probar, comparar y analizar técnicas de renderizado. Algunos de los más populares incluyen:

2.16.7Stanford bunny

El Stanford bunny es un modelo 3D icónico utilizado en gráficos por computadora desde 1994, derivado de un escaneo 3D de una pequeña estatua de conejo. Es utilizado principalmente para probar técnicas de renderizado y procesamiento geométrico debido a su forma compacta y su complejidad moderada.

Figura 2.18 Standford bunny **[59]**.

2.16.8Suzanne

Un modelo de cabeza de mono, que es la mascota de Blender, utilizado comúnmente en esa comunidad para pruebas rápidas de renderizado. Su simplicidad lo hace ideal para evaluaciones preliminares.

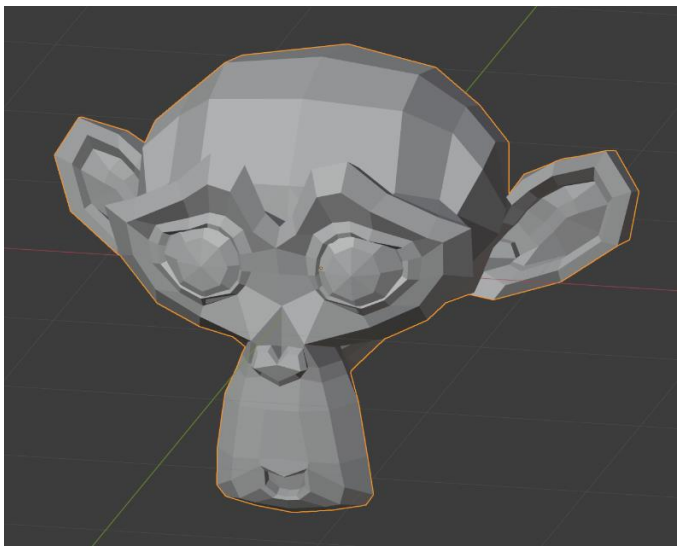

Figura 2.19 Suzanne **[60]**.

2.16.9Dragon de Stanford

Otro modelo estándar derivado de un escaneo 3D, con una geometría más compleja que el conejo de Stanford. Se utiliza para probar técnicas avanzadas de renderizado y procesamiento de geometría.

Figura 2.20 Dragón de Standford **[61]**.

2.16.10 Armadillo de Stanford

Similar al dragón y el conejo de Stanford, es un modelo detallado con alta cantidad de polígonos, ideal para pruebas de renderizado de alta resolución.

Figura 2.21 Armadillo de Standford **[62]**.

2.17 Renderers comerciales

El renderizado offline es una técnica utilizada principalmente para producir imágenes de alta calidad en las que el tiempo no es un factor crítico, lo que lo hace ideal para cine, efectos visuales (VFX) y visualización de alto nivel. A lo largo de las décadas, han surgido varios renderizadores offline comerciales, cada uno con características, ventajas e innovaciones únicas. A continuación, se presenta una visión general de los renderizadores offline más importantes, su origen, creadores, historia y características definitorias. Nótese que España y en especial Madrid es una potencia en renderizado offline:

2.17.1 RenderMan de Pixar

1. **País de origen:** Estados Unidos
2. **Creadores:** Pixar Animation Studios (fundada por Ed Catmull, Alvy Ray Smith, Steve Jobs)

3. **Historia:** RenderMan fue desarrollado por Pixar a finales de los años 80, diseñado para ser utilizado tanto por Pixar como por otras empresas para efectos visuales de alta calidad en largometrajes. RenderMan se ha convertido en un estándar en la industria del cine, utilizado en películas como Toy Story, Jurassic Park y Avatar. Está basado en el algoritmo REYES (Renders Everything You Ever Saw) y luego incorporó renderizado basado en la física (PBR).
4. **Características:**
   - Tipo: Híbrido (REYES, Ray tracing, PBR)
   - Lenguaje de sombreado: RenderMan Shading Language (RSL)
   - Fortalezas: Motion blur de alta calidad, dispersión subsuperficial, mapas de sombras profundas, soporte para escenas masivas, y algoritmos sofisticados como ray tracing.
5. **Usos notables:** Películas animadas de Pixar, VFX en Hollywoodç

2.17.2 Arnold

1. **País de origen**: España (desarrollado inicialmente en España, luego trasladado a Estados Unidos)
2. **Creadores:** Marcos Fajardo (desarrollador inicial), Solid Angle (empresa responsable), actualmente propiedad de Autodesk
3. **Historia:** Arnold comenzó a desarrollarse a finales de los 90 y ganó reconocimiento a principios de los 2000. Fue diseñado como un renderizador avanzado de ray tracing centrado en el PBR. Arnold se convirtió en un estándar en Hollywood debido a su eficiencia en el manejo de escenas complejas y la calidad fotorrealista. Películas como Gravity, Guardianes de la Galaxia y Avengers utilizaron Arnold para sus efectos visuales.
4. **Características:**
   - Tipo: Trazado de caminos de Montecarlo
   - Lenguaje de sombreado: Soporte para Open Shading Language (OSL), sombreadores personalizados
   - Fortalezas: Físicamente preciso, escalable para producciones de alto nivel, ray tracing avanzado, sin necesidad de precomputación para la

iluminación indirecta, fácil integración con las principales herramientas DCC (Maya, Houdini, etc.).

5. **Usos notables:** Largometrajes, VFX de alto nivel

2.17.3 Vray

1. **País de origen:** Bulgaria
2. **Creadores**: Chaos Group, fundada por Vladimir Koylazov y Peter Mitev
3. **Historia:** V-Ray fue desarrollado a finales de los 90, inicialmente para la visualización arquitectónica, pero luego evolucionó para su uso en diversas industrias, incluyendo el cine y los videojuegos. Ganó popularidad por su versatilidad y equilibrio entre calidad y velocidad de renderizado. Actualmente, V-Ray se utiliza ampliamente en VFX, arquitectura y diseño.
4. **Características:**
    - Tipo: Híbrido (Photon Mapping, Ray Tracing, Brute Force Path Tracing)
    - Lenguaje de sombreado: SDK de V-Ray, soporte para OSL
    - Fortalezas: Versátil, buen equilibrio entre velocidad y calidad, robusto para renderizado en CPU y GPU, fuertes opciones de iluminación global (GI), soporte para renderizado distribuido.
5. **Usos notables:** Visualización arquitectónica, VFX, comerciales

2.17.4 Mental Ray

1. **País de origen:** Alemania
2. **Creadores:** Mental Images (fundada por Rolf Herken)
3. **Historia:** Mental Ray fue desarrollado a finales de los 80 y se popularizó en los 90 y 2000. Era conocido por su integración con aplicaciones 3D como Autodesk Maya y 3ds Max. Mental Ray fue uno de los primeros renderizadores en ofrecer ray tracing e iluminación global, desempeñando un papel clave en la adopción de técnicas avanzadas de iluminación en CGI. Fue discontinuado en 2017.
4. **Características:**
    - Tipo: Híbrido (Ray Tracing, Scanline, Photon Mapping)

- Lenguaje de sombreado: Mental Ray Shader Language, soporte para MetaSL y OSL
- Fortalezas: Iluminación global precisa, ray tracing, soporte para escenas grandes, sistema de sombreado altamente personalizable.

5. **Usos notables:** Largometrajes, visualización arquitectónica (históricamente)

2.17.5 Maxwell

1. **País de origen:** España
2. **Creadores:** Next Limit Technologies (fundada por Víctor González e Ignacio Vargas)
3. **Historia:** Maxwell Render fue introducido en 2004 y es conocido por ser uno de los primeros renderizadores comerciales basados completamente en PBR. Tiene reputación por producir imágenes altamente realistas con una configuración mínima, utilizando un motor de renderizado espectral para simular el comportamiento real de la luz.
4. **Características:**
    - Tipo: Trazado de caminos espectral
    - Lenguaje de sombreado: Sistema propietario
    - Fortalezas: Renderizado físicamente preciso, simulación de transporte de luz, configuración intuitiva de materiales e iluminación, ideal para visualización arquitectónica y de productos.
5. **Usos notables:** Visualización arquitectónica, diseño de productos, renderizados fotorrealistas

2.17.6 Mantra

1. **País de origen**: Canadá
2. **Creadores**: SideFX (fundada por Kim Davidson y Greg Hermanovic)
3. **Historia**: Mantra es el renderizador integrado en Houdini, desarrollado por SideFX. Se popularizó por su flujo de trabajo procedural y su integración profunda en Houdini, lo que lo convierte en una solución preferida para la

animación procedural y los efectos visuales. Soporta tanto el renderizado de micropolígonos (similar a REYES) como el PBR.

1. **Características**:
    - Tipo: Híbrido (Micropolygon Rendering, Ray Tracing, Path Tracing)
    - Lenguaje de sombreado: VEX (Vector Expression Language)
    - Fortalezas: Fuerte integración procedural, sistema de sombreado flexible, altamente escalable para flujos de trabajo de VFX grandes, robusto para simulaciones complejas.
2. **Usos notables**: Efectos visuales procedurales, simulaciones, producción de VFX de alto nivel

2.17.7 Octane

1. **País de origen**: Nueva Zelanda
2. **Creadores**: OTOY Inc. (fundada por Jules Urbach)
3. **Historia**: Octane Render es un renderizador acelerado por GPU, introducido en 2009, que trajo mejoras significativas en velocidad al aprovechar las GPU para ray tracing y path tracing. Fue uno de los primeros renderizadores no sesgados disponibles comercialmente en utilizar completamente la aceleración por GPU.
4. **Características**:
    - Tipo: Trazado de caminos no sesgado.
    - Lenguaje de sombreado: Soporte para OSL, sombreadores personalizados
    - Fortalezas: Capacidades de renderizado en tiempo real, aceleración completa por GPU, fotorrealismo preciso, integración con varias herramientas DCC.
5. **Usos notables**: Visualización arquitectónica, aplicaciones en tiempo real, VFX

2.17.8 Corona

1. **País de origen:** República Checa
2. **Creadores:** Ondřej Karlík y un equipo de Chaos Czech
3. **Historia:** Corona Renderer comenzó como un proyecto universitario en 2009 y rápidamente se convirtió en una opción popular en las industrias de arquitectura y VFX. Se enfoca en la simplicidad y facilidad de uso, haciendo que el renderizado físicamente preciso de alta calidad sea accesible sin configuraciones extensas.
4. **Características:**
    - Tipo: Path Tracing
    - Lenguaje de sombreado: Soporte para OSL, plugins de sombreadores personalizados
    - Fortalezas: Interfaz fácil de usar, altamente eficiente para resultados fotorrealistas, tiempos de renderizados rápidos, configuración mínima para obtener resultados de calidad, fuerte renderizado en CPU.
5. **Usos notables:** Visualización arquitectónica, diseño de productos, VFX

Cada renderizador tiene ventajas distintas según el caso de uso específico, el flujo de trabajo y las necesidades de la industria. RenderMan y Arnold se utilizan ampliamente en VFX de alto nivel, mientras que V-Ray y Corona dominan la visualización arquitectónica. Por otro lado, Octane y Maxwell son preferidos por quienes buscan velocidad y precisión mediante la aceleración por GPU o el renderizado espectral [63] [64] [65].

Renderizador	País de origen	Tipo	Lenguaje de sombreado	Fortalezas	Usos notables
RenderMan	EE. UU.	Híbrido (REYES, Ray tracing, PBR)	RenderMan Sombreado Language (RSL)	Motion blur, dispersión subsuperficial, sombras profundas	Películas de Pixar, VFX en Hollywood
Arnold	España/EE. UU.	Trazado de caminos de Montecarlo	Soporte para OSL	Precisión física, escalable, fácil integración	Largometrajes, VFX de alto nivel
V-Ray	Bulgaria	Híbrido (Photon Mapping, Ray Tracing)	SDK de V-Ray, soporte para OSL	Versátil, soporte para CPU y GPU, renderizado distribuido	Visualización arquitectónica, VFX, comerciales
Mental Ray	Alemania	Híbrido (Ray tracing, Scanline)	Mental Ray Shader Language	Iluminación global precisa, sombreado personalizable	Largometrajes (histórico), arquitectura
Maxwell	España	Trazado de caminos espectral	Sistema propietario	Renderizado físicamente preciso, realismo sin esfuerzo	Visualización arquitectónica, diseño de productos
Mantra	Canadá	Híbrido (Micropolígonos, Ray tracing)	VEX (Vector Expression Language)	Procedural, escalable, integración con Houdini	Efectos visuales procedurales, simulaciones
Octane Render	Nueva Zelanda	Trazado de caminos imparcial	Soporte para OSL	Renderizado en tiempo real, aceleración completa por GPU	Aplicaciones en tiempo real, VFX
Corona	República Checa	Path Tracing	Soporte para OSL, plugins personalizados	Fácil de usar, resultados rápidos, renderizado eficiente	Visualización arquitectónica, diseño de productos, VFX

Tabla 2-2 Resumen de características.

2.18 Conclusiones

El capítulo dos establece el marco conceptual del proyecto, tratando diversas técnicas esenciales en el ámbito del renderizado 3D que serán fundamentales para su desarrollo. Se explican métodos como la rasterización, el ray tracing, el path tracing y el ray marching. Además, se analizan diferentes modelos de reflexión, desde el modelo Phong hasta la BRDF Cook-Torrance.

Asimismo, el capítulo trata técnicas avanzadas de iluminación global, como radiosity y la ecuación de renderizado, que permiten la iluminación indirecta, rebotes y sangrados de luz. Se discute el uso del Método de Montecarlo para, método que introdujo Kayiya para poder lograr la iluminación global. Además, se presentan optimizaciones específicas como la next event estimation y la russian roulette, junto con técnicas avanzadas como el photon mapping y el multiple importance sampling, que aumentan la eficiencia del proceso de renderizado.

Finalmente, se revisan las estructuras de aceleración, como BVH, KD-Trees y grids, otras técnicas aparte de las preexistentes y se evalúan renderizadores comerciales como RenderMan o Arnorld.

Muchos de los conceptos aquí expuestos son los objetivos a implementar en el producto final del proyecto. Incluso más adelante se podrá observar el uso de la tetera de Utah o la utilización de la caja de Cornell.

Capítulo 3

Análisis y metodología

3.1 Introducción

El análisis de software es el proceso de estudiar y entender los requisitos, funcionalidades y limitaciones de un sistema para diseñar una solución adecuada que cumpla con las necesidades del usuario. Involucra identificar problemas, definir las características del sistema y especificar los recursos necesarios para su desarrollo [65].

La metodología de desarrollo de software se refiere al conjunto de prácticas y enfoques organizados que guían el proceso de diseño, codificación, pruebas e implementación del software, tales como metodologías ágiles, en cascada o iterativas, cada una con diferentes enfoques para gestionar y completar proyectos de software. En este capítulo se define el análisis de requisitos y la metodología usada [67] [67].

3.2 Metodología

Se decidió la utilización de una metodología ágil para aprovechar la flexibilidad y adaptabilidad que este enfoque brinda ante posibles cambios durante el desarrollo. Aunque se trata de un proyecto individual, se adopta un enfoque poco ortodoxo de Scrum, combinándolo con Kanban como herramienta de gestión. Esta combinación permite mantener una visión clara y continua del progreso, optimizando la priorización de tareas, mejorando la eficiencia en la gestión del tiempo y facilitando

la detección temprana de posibles bloqueos. Kanban, al promover un flujo de trabajo constante y organizado, complementa la estructura ágil del proyecto, proporcionando un sistema eficaz para gestionar el desarrollo de manera visual y ajustable.

3.3 Análisis

Un backlog es una lista priorizada de tareas, historias de usuario o funcionalidades pendientes de implementar en un proyecto, que sirve como una hoja de ruta para el equipo de desarrollo. En metodologías ágiles, como Scrum, el backlog incluye tanto los requisitos funcionales como las mejoras y correcciones, y está en constante evolución. Las tareas se priorizan según su valor, urgencia o complejidad, permitiendo al equipo centrarse en lo más importante durante cada ciclo de trabajo o sprint.

En este apartado se definen los requisitos en el product backLog desglosados en los sprint backLog.

3.3.1 Sprint backlog: Implementación de un trazador de rayos básico

Historia de usuario: Como desarrollador, quiero implementar una forma de generar imágenes.

Criterios de aceptación:

- Se deben poder definir la imagen pixel a pixel en RGB.

Historia de usuario: Como desarrollador, quiero implementar el cálculo de intersecciones de rayos con primitivas (esferas y triángulos) para poder definir las geometrías básicas.

Criterios de aceptación:

- Se deben detectar correctamente las intersecciones con triángulos.
- Se deben detectar correctamente las intersecciones con esferas.

Historia de usuario: Como desarrollador, quiero importar mallas de polígonos y calcular intersecciones usando álgebra aplicada a gráficos para representar modelos más complejos.

Criterios de aceptación:

- Se debe poder cargar mallas poligonales en formato estándar.
- El cálculo de intersecciones debe ser preciso para modelos con múltiples polígonos.
- Las figuras deben ser bien representadas y estar bien ubicadas.

Historia de usuario: Como desarrollador, quiero implementar los modelos de sombreado Phong y Blinn-Phong con luces puntuales, direccionales y de foco para simular efectos de iluminación realista.

Criterios de aceptación:

- Implementación correcta del modelo Phong.
- Implementación del modelo Blinn-Phong.
- Las luces puntuales, direccionales y de foco deben afectar correctamente a la escena.
- Las sombras deben calcularse correctamente en función de la posición de las fuentes de luz.

3.3.2 Sprint backlog: Transformación del ray caster en Whitted ray tracer

Historia de usuario: Como desarrollador, quiero implementar sobremuestreo para mejorar la calidad visual del renderizado, reduciendo el aliasing y posteriormente el ruido en el Backward Path Tracer.

Criterios de aceptación:

- El sobremuestreo debe reducir los artefactos visuales.

Historia de usuario: Como desarrollador, quiero implementar texture mapping para aplicar texturas a las superficies y mejorar el realismo.

Criterios de aceptación:

- Se debe poder cargar y aplicar texturas de diferentes formatos.

El mapeo de texturas debe ser preciso y sin artefactos.

Historia de usuario: Como desarrollador, quiero implementar la ley de Snell y las ecuaciones de Fresnel para simular correctamente la refracción y reflexión en materiales transparentes.

Criterios de aceptación:

- La refracción y reflexión deben seguir la física de la óptica basada en la ley de Snell.
- Las ecuaciones de Fresnel deben implementarse correctamente para el cálculo de la reflectancia.

Historia de usuario: Como desarrollador, quiero implementar el Renderizado Basado en Físicas (PBR) aplicando técnicas avanzadas de radiometría para un realismo más profundo.

Criterios de aceptación:

- Se debe implementar las ecuaciones de Cook-Torrance.
- El modelo PBR debe simular aproximadamente materiales como metales y dieléctricos.

Historia de usuario: Como desarrollador, quiero desarrollar estructuras de datos optimizadas y algoritmos de aceleración para mejorar el rendimiento del renderizado.

Criterios de aceptación:

- Se deben calcular las intersecciones con bounding boxes.
- Se deben implementar estructuras de aceleración como las bounding boxes o la red de Fujimoto.

Historia de usuario: Como desarrollador, quiero implementar el procesado en paralelo.

Criterios de aceptación:

- Se debe paralelizar las tareas independientes.

3.3.3 Sprint backlog: Implementación de un backward path tracer

Historia de usuario: Como desarrollador, quiero implementar la iluminación global.

Criterios de aceptación:

- Se debe implementar el método de Montecarlo para la emisión de rayos secundarios.
- Las sombras deben tener bordes suaves y graduales.

Historia de usuario: Como desarrollador, quiero implementar luces de área para simular fuentes de luz más realistas y volumétricas.

Criterios de aceptación:

- Las luces de área deben iluminar correctamente las superficies y proyectar sombras difusas.

Historia de usuario: Como desarrollador, quiero implementar Importance Sampling para mejorar la eficiencia del muestreo y reducir el ruido en la imagen.

Criterios de aceptación:

- La técnica debe reducir el ruido general de la escena.
- El rendimiento no debe verse comprometido.

Historia de usuario: Como desarrollador, quiero implementar la técnica de Next Event Estimation para mejorar el cálculo de la iluminación indirecta.

Criterios de aceptación:

- El cálculo de iluminación indirecta debe mejorar en el ruido sin incrementar significativamente el tiempo de renderizado o introducir un sesgo exagerado.

Historia de usuario: Como desarrollador, quiero implementar la técnica de aceleración Russian Roulette para optimizar la terminación de trayectorias sin afectar la calidad visual.

Criterios de aceptación:

- La técnica debe reducir el número de rayos calculados sin comprometer la calidad visual al introducir demasiado ruido.

Historia de usuario: Como desarrollador, quiero desarrollar una interfaz de usuario para interactuar con el trazador de rayos de forma eficiente y ajustable.

Criterios de aceptación:

- La interfaz debe ser intuitiva y permitir el ajuste de parámetros como la escena a cargar, la resolución, el sobremuestreo o los rayos secundarios emitidos.

Este backlog puede ser priorizado en función de la importancia de cada funcionalidad y del flujo de trabajo que se para optimizar el proceso de desarrollo.

El desarrollo se realizó en tres iteraciones o sprints principales, cada uno con una duración de 4 semanas, y con objetivos específicos y medibles en el Backlog, lo que permite dividir el trabajo en etapas manejables y asegurar la entrega progresiva de funcionalidades clave.

3.4 Conclusiones

En este capítulo se ha definido la metodología utilizada y se han establecido los objetivos del proyecto a través de historias de usuario. En los siguientes capítulos primero se mostrará un ligero diseño de cómo es la aplicación y luego se documentarán las iteraciones del desarrollo en forma de memoria, detallando los avances realizados. Además, se describirá la algoritmia y el diseño necesario para implementar un Backward Path Tracer y se incluirán imágenes como evidencia de validación de los objetivos alcanzados.

Capítulo 4

Diseño

4.1 Introducción

En este capítulo se presenta el diseño del ray tracer de Montecarlo. Se describen los componentes fundamentales del programa y se explica cómo se han materializado los requisitos y las historias de usuario identificadas durante la fase de análisis.

Adoptando una metodología ágil, el diseño se ha centrado en ser flexible y adaptable, permitiendo iteraciones rápidas y facilitando la incorporación de mejoras continuas. Se ha evitado la complejidad innecesaria asociada con metodologías más pesadas como el proceso unificado, optando por utilizar patrones de diseño que aportan simplicidad y eficacia al desarrollo.

4.2 Visión general de la arquitectura

El proyecto trata de la implementación de un software de ray tracing de Montecarlo, complementado con una interfaz gráfica de usuario desarrollada en Windows Forms. La arquitectura del sistema es sencilla y eficaz, basada en la interacción entre estos dos componentes principales de manera desacoplada.

La interfaz gráfica ofrece al usuario una forma intuitiva de configurar los parámetros de renderizado. Permite seleccionar el archivo de la escena a renderizar, definir el nombre del archivo de salida y ajustar diversos parámetros de renderizado, como la resolución, el número de muestras y otros ajustes específicos.

Una vez configurados los parámetros, la interfaz gráfica ejecuta el programa del ray tracer como un proceso independiente. La comunicación entre la interfaz y el ray tracer se realiza mediante el envío de argumentos en la línea de comandos. Específicamente, se le proporcionan:

- Nombre del archivo de la escena a renderizar: Indica al ray tracer qué escena debe procesar.
- Nombre del archivo de salida: Especifica dónde se almacenará la imagen renderizada.
- Parámetros de renderizado: Incluye opciones como la resolución, número de muestras por píxel y otros ajustes que afectan al proceso de renderizado.

Esta forma simple de interacción es eficaz, ya que mantiene ambos componentes desacoplados. La interfaz gráfica no necesita conocer los detalles internos del ray tracer y éste no depende de la interfaz para funcionar, facilitando el mantenimiento y la posibilidad de mejorar o reemplazar cada componente de forma independiente sin afectar al otro.

La arquitectura propuesta cumple con los objetivos del proyecto al proporcionar una experiencia de usuario amigable y un sistema modular. Se logra un diseño flexible y escalable, preparado para futuras extensiones o mejoras, al separar la lógica de presentación (interfaz gráfica) de la lógica de procesamiento (ray tracer),

4.3 Componentes principales y responsabilidades

El proyecto se descompone en tres módulos principales: Render, Geometry y Shading.

Módulo Render: Contiene el ciclo principal para la generación de la imagen final. Aquí se instancian todos los objetos y estructuras de datos necesarias para el renderizado.

Módulo Geometry: Incluye funciones algebraicas que permiten el cálculo de intersecciones entre geometrías. Este módulo se usa para determinar cómo los rayos interactúan con los objetos de la escena.

Módulo Shading: Alberga todas las operaciones relacionadas con el color y la iluminación de la escena. Dentro de este módulo se encuentra el **submódulo PBR** (Physically Based Rendering), que implementa las ecuaciones de Cook-Torrance.

Clases principales

Ray: Representa un rayo en la escena. Contiene información sobre el color, dirección, origen y punto de impacto del rayo. Cuenta con métodos para la creación de rayos de reflexión, refracción y aleatorios en la hemiesfera.

Triangle: Describe un triángulo y almacena datos como sus vértices, área, bounding box, normales por vértice y material asociado. Esta clase hereda de una clase abstracta llamada Figure y es compartida mediante shared_ptr tanto por la malla a la que pertenece como por la Fujimoto Grid.

Hit: Estructura que almacena información sobre el punto de impacto de un rayo. Diseñada para residir en la pila, lo que permite un acceso rápido y eficiente. Contiene datos como el color en el punto de impacto, metalicidad, rugosidad y otros parámetros utilizados en el proceso de sombreado. Esto permite que la clase Ray sea más ligera al no tener que almacenar información adicional.

TriangleMesh: Representa una malla u objeto compuesto por múltiples triángulos. Almacena triángulos, vértices, normales y materiales. Incluye métodos para aplicar transformaciones como rotaciones y para definir los triángulos que la componen.

Material: Estructura que contiene las propiedades físicas del material que compone una malla o triángulo, como el color, rugosidad y metalicidad. Puede contener punteros a hasta ocho texturas. Los materiales se asocian a los triángulos y mallas mediante shared_ptr.

Texture: Maneja los mapas de texturas. Utiliza la librería STB para cargar imágenes en una cadena char*. Proporciona métodos para obtener la información de un texel dada una coordenada de textura, usada para el proceso de sombreado. Las texturas se encuentran en los materiales y en la escena como punteros.

Camera: Representa la cámara o punto de vista del observador en la escena. Contiene información sobre su posición, orientación, ángulo de visión y otros parámetros

necesarios para dirigir los rayos en la escena y determinar la perspectiva de la imagen renderizada.

Light: Clase abstracta que define una fuente de luz. Cuenta con cuatro subclases que representan las principales fuentes de luz utilizadas en la industria:

- **PointLight:** Luz puntual que emite desde una posición específica en todas las direcciones.
- **SpotLight:** Luz focalizada que emite dentro de un cono desde una posición determinada.
- **DistantLight:** Luz distante que simula una fuente de luz situada en el infinito, como el sol.
- **AreaLight:** Luz de área que emite desde una superficie.

Clases estructurales

Scene: Responsable de crear e instanciar todas las mallas, materiales y texturas. También define e instancia las luces y la cámara. Utiliza la librería ASSIMP para la importación de escenas y modelos 3D. Incluye métodos para recorrer nodos y establecer posiciones en la escena.

Fujimoto Grid: Implementa una estructura de aceleración espacial basada en una rejilla, conocida como "Fujimoto Grid". Contiene celdas que almacenan shared_ptr a objetos Triangle. Esta estructura permite calcular de forma eficiente las intersecciones rayo-triángulo, usando funciones del módulo Geometry. Actúa como interfaz entre la escena y el módulo Render durante el cálculo de intersecciones.

NEE (Next Event Estimation): Estructura que almacena triángulos con materiales o texturas emisivas, es decir, que emiten luz. Se utiliza para integrar triángulos emisivos con luces de área al implementar la técnica de Next Event Estimation.

Herramientas complementarias

Generador de puntos de Poisson: Se emplea un generador de puntos de Poisson para la estratificación en el sobremuestreo, creado por Sergey Kosarevsky.

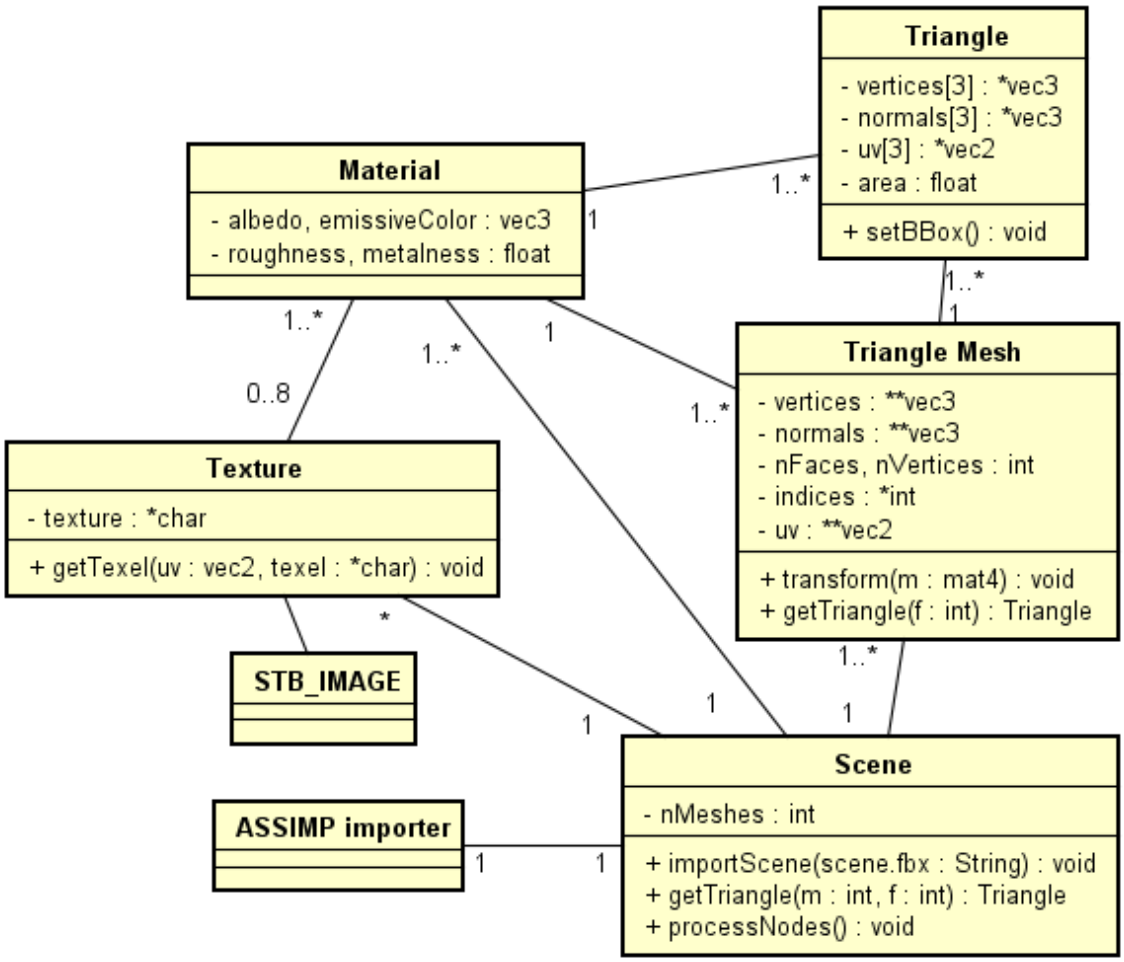

Figura 4.1 Diagrama de Clases Simplificado.

4.4 Patrones de diseño aplicados

En el diseño del pathtracer, se han implementado unos pocos patrones de diseño para lograr un sistema modular, flexible y escalable. Se ha puesto especial énfasis en lograr máxima cohesión y mínimo acoplamiento entre los componentes, facilitando así el mantenimiento y la extensibilidad del software. A continuación, se describen algunos de los patrones de diseño utilizados.

Patrón composite (Compuesto):

El patrón composite se ha utilizado en la implementación de la clase TriangleMesh, que representa una malla compuesta por múltiples triángulos (Triangle). Este patrón permite tratar objetos individuales y compuestos de manera uniforme.

- Figure: Es una clase abstracta que define la interfaz común para todas las figuras geométricas.
- Triangle: Hereda de Figure y representa un triángulo individual.
- Esphere: Hereda de Figure y representa una esfera.
- TriangleMesh: También hereda de Figure y agrupa múltiples Triangle objetos, permitiendo operar sobre la malla completa como si fuera un solo objeto.

Patrón cadena de responsabilidad (Chain of responsibility):

El Patrón cadena de responsabilidad es un patrón de diseño de comportamiento que permite pasar una solicitud a lo largo de una cadena de manejadores hasta que alguno de ellos la procesa. Cada manejador decide si procesa la solicitud o la pasa al siguiente en la cadena.

- Al pedir triángulos a la escena, la escena los solicita a la malla de triángulos o también sucede los mismo al usar la Fujimoto Grid.

Como nota adicional, el uso de algunos patrones del diseño como visitor podría generar una pequeña sobrecarga de cálculos, pese a que promueven la modularidad y la escalabilidad del código por lo que si se hubieran utilizado habría sido en detrimento del rendimiento del programa.

4.5 Decisiones en el diseño

En las decisiones de diseño, se ha priorizado la velocidad de ejecución por encima de todo, implementando optimizaciones que agilizan el procesamiento y mejoran el rendimiento general del programa.

Para aumentar la velocidad del programa, se ha decidido tratar las clases Ray y Triangle como estructuras (structs), lo que significa que sus atributos son públicos. Al hacer esto, se elimina la necesidad de utilizar métodos getter para acceder a los atributos, evitando así la sobrecarga de llamadas a funciones adicionales. El acceso directo a los datos internos permite operaciones más rápidas, ya que se reduce el tiempo de ejecución asociado con la encapsulación tradicional. Por lo tanto, se

considera que esta optimización hará que el programa sea más eficiente al acelerar el acceso y manipulación de los atributos de estas clases.

Se ha intentado que se reprodujera el mínimo de cálculos posibles, en algunas ocasiones aumentando un poco el uso de memoria al almacenar variables precalculadas.

En las ecuaciones de sombreado, se ha optado por utilizar el valor absoluto del producto escalar con cualquier vector normal, en lugar de calcular el máximo entre cero y dicho producto escalar. De esta manera, la orientación de la cara del triángulo intersecado se vuelve irrelevante, y al mismo tiempo, al ser una operación más rápida, se ha logrado aumentar la velocidad de cómputo. Aunque en rasterización sería una temeridad, ya que se iluminarían caras debido a luces que están detrás, en ray tracing, al trazarse caminos e intersecciones, es viable, reduciendo el número de operaciones para calcular y aplicar la inversión de la normal y reduciendo la carga en las operaciones del sombreado al cambiar una función max por una función abs, considerando que en el proceso estas operaciones se realizan infinidad de veces.

4.6 Conclusiones

Debido a la metodología ágil adoptada y al pequeño tamaño de la aplicación, este capítulo se trata de manera concisa, con pocos diagramas y un enfoque práctico.

Se ha descrito la estructura general de la pequeña arquitectura que compone la aplicación, junto con su interfaz. Además, se han analizado las principales clases y módulos, explicando su función en el correcto funcionamiento del programa.

También se han mencionado los patrones de diseño utilizados, priorizando la máxima cohesión y el mínimo acoplamiento entre los componentes. Por último, se han discutido algunas de las decisiones clave tomadas para acelerar el proceso de renderizado.

Capítulo 5

Implementación

5.1 Introducción

Este capítulo se divide en cuatro subcapítulos: primera iteración, segunda iteración, tercera iteración y desarrollo de la interfaz gráfica (GUI). Su propósito principal es detallar la implementación de los algoritmos investigados durante el desarrollo del *Backward Path Tracer*.

Aunque los algoritmos utilizados son creaciones de reconocidos expertos en el campo, su inclusión en este capítulo es justificada porque aquí se explica cómo se han integrado específicamente en el proyecto. Presentar estos algoritmos en el capítulo de implementación permite mostrar de manera práctica cómo funcionan dentro del sistema desarrollado, pudiendo destacar aspectos como la estructura del código, la interacción entre diferentes componentes o el rendimiento obtenido.

Esta aproximación práctica complementa la revisión teórica realizada en el marco conceptual, proporcionando una visión completa trata tanto el conocimiento existente como su aplicación concreta en la construcción del *Backward Path Tracer*.

Su implementación y su desarrollo en este proyecto es original e inspirado en la bibliografía investigada, y en algunos casos se han realizado pequeñas modificaciones para adaptarlos o mejorar las necesidades específicas del desarrollo.

Este capítulo es un compendio muy completo que recoge todas las técnicas aplicadas.

5.2 Primera iteración

5.1.1 Introducción de la primera iteración

En este subcapítulo se describen los objetivos cumplimentados del backlog en el primer sprint de la metodología ágil. Algunas de las imágenes son pruebas de validación de los resultados obtenidos en las 4 semanas.

5.1.2 Generación de imágenes

El proyecto comenzó con la configuración del entorno de desarrollo en Visual Studio, en el lenguaje C++. El primer desafío fue generar una imagen. Para superarlo, se utilizó la biblioteca estándar (STD) para implementar la funcionalidad necesaria que permitiera crear un archivo en blanco, en el cual se pudiera escribir la información requerida para convertirlo en una imagen.

Se eligió el formato PPM como estándar de la generación de imágenes. Para crear un archivo PPM (Portable Pixmap), se comienza con un archivo genérico en blanco. Primero, se especifica el formato "P3", seguido de la resolución de la imagen y el valor máximo de color, que para P3 es 255. Después de esto, se describe el contenido de cada píxel, escribiendo las componentes de color (rojo, verde y azul) línea por línea. Por ejemplo:

```
P3

850 480
255
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
...
```

Para implementar la generación del archivo PPM, se construyó un bucle doble donde los pixeles ordenados se escribían en el archivo fila a fila.

Las primeras imágenes obtenidas muestran los colores en pantalla obtenidos; una en la que solo se definió el color rojo, (255,0,0) y en la otra se hizo una sencilla interpolación de colores según la posición del pixel como se muestra en la Figura 5.1.

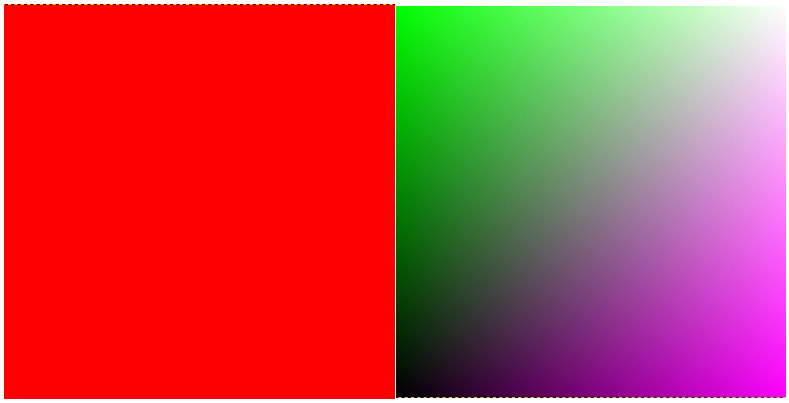

Figura 5.1 Visualización del archivo ppm.

También se incorporó la librería GML para el cálculo matemático en coma flotante. Y se preparó un esquema Kanban con los objetivos marcados.

A partir de aquí, cumplido el objetivo marcado para la primera semana, se tuvo una base con la que empezar a plasmar en pantalla los resultados durante las distintas iteraciones.

5.1.3 Ray casting

El segundo subobjetivo marcado era la intersección de los rayos con las figuras. Para ello primero se generaron los rayos. Para la generación de rayos se creó una clase, llamada Ray. La clase Ray contaba con la información inicial de su origen, su color y su dirección.

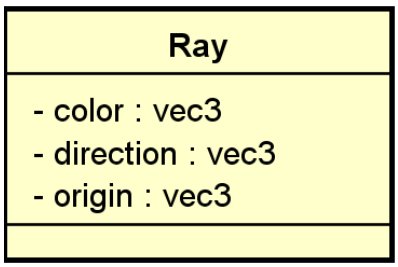

Figura 5.2 Clase Ray.

Además, se generó un buffer de rayos del mismo tamaño que la resolución de la pantalla ya que en el ray casting se emite un rayo por pixel.

El ray casting consiste en la emisión de rayos, donde a cada píxel se le asigna un rayo específico. En coordenadas cartesianas (x, y, z), los rayos se emiten desde el ojo o la cámara hacia la posición del píxel en la pantalla o lienzo (canvas). Los rayos atraviesan el centro de cada píxel.

En una primera aproximación se consideró que la cámara estaba en una posición (0,0,-1) y el lienzo en (0,0,0). Se consideró que la pantalla era cuadrada, con un tamaño que iba de (-1, 0) a (1, 0) en el eje x y de (0, -1) a (0, 1) en el eje y, formando un área de 2x2 unidades.

Para emitir y definir un rayo se usa el siguiente algoritmo:

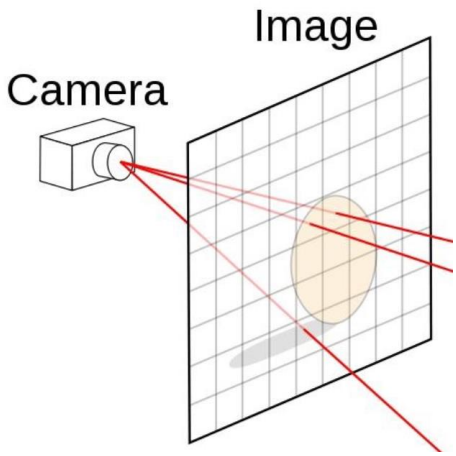

Figura 5.3 Representación del ray casting **[69]**.

El vector $\vec{r}$ representa un rayo cualquiera:

$$\vec{r} = \vec{o} + \lambda \cdot \vec{d} \tag{5.1}$$

El origen, $\vec{o}$, es el lugar del ojo o cámara:

$$\vec{o} = (0,0,-1) \tag{5.2}$$

y el vector $\vec{d}$, la dirección, es la resta de la posición del origen y la posición del pixel $\vec{p}$.

$$\vec{d} = \frac{\vec{p} - \vec{o}}{|\vec{p} - \vec{o}|} \tag{5.3}$$

$$\vec{p} = (\mathrm{x}_i, \mathrm{y}_j, 0) \tag{5.4}$$

$$0 \leq i < anchura \tag{5.5}$$

$$0 \leq j < altura \tag{5.6}$$

$$x_i = \frac{1 - 2(i + 0.5)}{anchura} \tag{5.7}$$

$$y_j = \frac{1 - 2(j + 0.5)}{altura} \tag{5.8}$$

Donde anchura y altura corresponden con la resolución de la pantalla.

En el ray casting el color del píxel en pantalla corresponde al color que adquiere el rayo vinculado. En este caso, los colores, siguiendo el estándar de la industria, se representan en valores de coma flotante que van de 0 a 1. Estos rayos se denominan rayos primarios.

Por el momento, los rayos solo podían ser negros, con color = (0,0,0), o rojos, con color = (1,0,0).

Generar todos los rayos primarios tiene un coste O(n) donde n es el número de pixeles de la resolución de la pantalla, aunque es un proceso que se puede paralelizar.

Una vez que se pudo generar rayos, se pasó al cálculo algebraico de intersecciones de rayos con otros objetos.

5.1.4 Intersección con un triángulo

El primer paso y el más importante para cualquier render es mostrar un triángulo en pantalla. Posteriormente se fue complicando el programa, integrando nuevas funcionalidades.

Por cada pixel se emitió un rayo, hasta el infinito. El triángulo se definió por los vértices (-1,-1,5), (1,-1,5) y (0,1,5).

Para cada rayo definido por su punto de origen y su vector de dirección, $\vec{r} = \vec{o} + \lambda \cdot \vec{d}$, se comprobó si intersecaba con el triángulo.

Hay muchas formas de medir la intersección, por ejemplo, una se basa en encontrar la normal del triángulo y comprobar a qué distancia, λ=t, corta la línea con el plano del triángulo. Luego, usando coordenadas baricéntricas, se halla si el punto de corte se encuentra dentro del triángulo.

Las coordenadas baricéntricas (u,v,w) son una transformación que sitúa un punto en el plano normal del triángulo dependiendo del peso de cada vértice del triángulo. u+v+w=1 siempre dentro de los límites del triángulo.

Por así decirlo, si se tiene un triángulo ABC y un punto $\vec{P}$ de corte en el plano del triángulo, se calculan las áreas de tres subtriángulos con un vértice en P; ABP, ACP, BCP, si las tres subáreas suman más que el triángulo base es que el punto esta fuera. O visto de otra forma, si las coordenadas baricéntricas suman más de 1. u+v+w>1;

Para calcular el punto de intersección tenemos la línea y su ecuación paramétrica:

$$\vec{r} = \vec{o} + \lambda \cdot \vec{d} \tag{5.9}$$

Los vértices del triángulo:

$$\overrightarrow{v_o}, \overrightarrow{v_1}, \overrightarrow{v_2} \tag{5.10}$$

Y el plano del triángulo y su ecuación implícita:

$$\mathrm{Ax} + \mathrm{By} + \mathrm{Cz} + \mathrm{D} = 0 \tag{5.11}$$

donde (A, B, C) son las coordenadas cartesianas de la normal del triángulo que se pueden hallar con el producto vectorial normalizado de $(\overrightarrow{v_1} - \overrightarrow{v_o})$ y $(\overrightarrow{v_2} - \overrightarrow{v_o})$:

$$\vec{N} = \frac{(\overrightarrow{v_1} - \overrightarrow{v_o}) \times (\overrightarrow{v_2} - \overrightarrow{v_o})}{|(\overrightarrow{v_1} - \overrightarrow{v_o}) \times (\overrightarrow{v_2} - \overrightarrow{v_o})|} = (A, B, C) \tag{5.12}$$

Y D:

$$\mathrm{A} \cdot v_{0x} + \mathrm{B} \cdot v_{0y} + \mathrm{C} \cdot v_{0z} + \mathrm{D} = 0 \tag{5.13}$$

$$\vec{N} \cdot \overrightarrow{v_o} + \mathrm{D} = 0 \tag{5.14}$$

ó

$$\vec{N} \cdot \overrightarrow{v_1} + \mathrm{D} = 0 \tag{5.15}$$

ó

$$\vec{N} \cdot \overrightarrow{v_2} + \mathrm{D} = 0 \tag{5.16}$$

Resolviendo tenemos:

$$\mathrm{D} = -\vec{N} \cdot \overrightarrow{v_o} \tag{5.17}$$

El punto de intersección $\vec{P}$ es:

$$\vec{\mathrm{P}} = \vec{o} + t \cdot \vec{d} \tag{5.18}$$

Donde t es la distancia al plano normal del triángulo con el origen del rayo y es un escalar.

$$\vec{N} \cdot \vec{P} + \mathrm{D} = 0 \tag{5.19}$$

$$\vec{N}\cdot(\vec{o}+t\cdot\vec{d})+\mathrm{D}=0 \tag{5.20}$$

$$\vec{N}\cdot\vec{o}+\vec{N}\cdot(t\cdot\vec{d})+\mathrm{D}=0 \tag{5.21}$$

$$\vec{N}\cdot(t\cdot\vec{d})=-(\mathrm{D}+\vec{N}\cdot\vec{o}) \tag{5.22}$$

$$t\cdot\vec{N}\cdot\vec{d}=-(\mathrm{D}+\vec{N}\cdot\vec{o}) \tag{5.23}$$

$$\mathrm{t}=\frac{-(\mathrm{D}+\vec{N}\cdot\vec{o})}{\vec{N}\cdot\vec{d}} \tag{5.24}$$

$$\mathrm{t}=\frac{\vec{N}\cdot\overrightarrow{v_o}-\vec{N}\cdot\vec{o}}{\vec{N}\cdot\vec{d}} \tag{5.25}$$

$$\vec{\mathrm{P}}=\vec{o}+t\cdot\vec{d} \tag{5.26}$$

Luego el área de un triángulo se puede calcular hallando la longitud del producto vectorial de sus vectores partido entre dos.

$$\mathrm{Area}_{ABC}=\frac{|\overline{AB}\times\overline{AC}|}{2} \tag{5.27}$$

Y las subáreas:

$$\mathrm{Area}_{ABP}=\frac{|\overline{AB}\times\overline{AP}|}{2} \tag{5.28}$$

$$\text{Area}_{ACP} = \frac{|\overline{AC} \times \overline{AP}|}{2} \tag{5.29}$$

$$\text{Area}_{BP} = \frac{|\overline{BC} \times \overline{CP}|}{2} \tag{5.30}$$

Y las coordenadas baricéntricas:

$$u = \frac{\text{Area}_{ABC}}{\text{Area}_{ABP}};\ v = \frac{\text{Area}_{ABC}}{\text{Area}_{ACP}}; w = \frac{\text{Area}_{ABC}}{\text{Area}_{BCP}} \tag{5.31}$$

Otro algoritmo podría calcular la posición del punto de intersección con respecto a los bordes del triángulo y hallar la intersección de esta forma.

La implementación de las ecuaciones anteriores se puede optimizar en gran medida. El algoritmo de Möller-Trumbore es el método más eficiente para calcular la intersección entre una línea (o rayo) y un triángulo usando estas ecuaciones. Fue desarrollado por *Tomas Möller y Ben Trumbore*, y publicado en 1997 en el artículo *"Fast, Minimum Storage Ray-Triangle Intersection".* [69]

Se implementó y aplicado el algoritmo de Möller-Trumbore satisfactoriamente, se obtuvieron los siguientes resultados mostrados en la Figura 5.4 y la Figura 5.5:

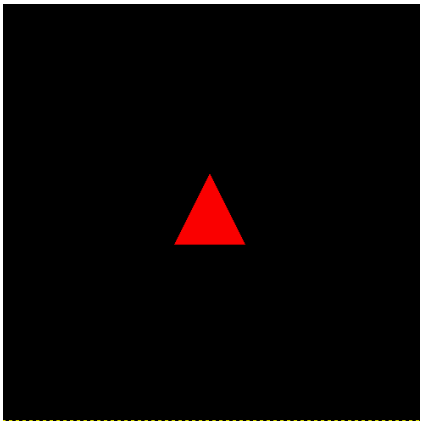

Figura 5.4 Primer triángulo.

Para obtener la imagen se estableció que al golpear al triangulo el rayo se volvía rojo y si fallaba se quedaba negro.

Al variar el tamaño del marco o la resolución, la imagen se deformaba, por lo que fue necesario ajustar y crear una interfaz, una cámara a la que se le pudieran aplicar transformaciones, efectos, giros y movimientos, manteniendo la representación de los objetos de forma consistente.

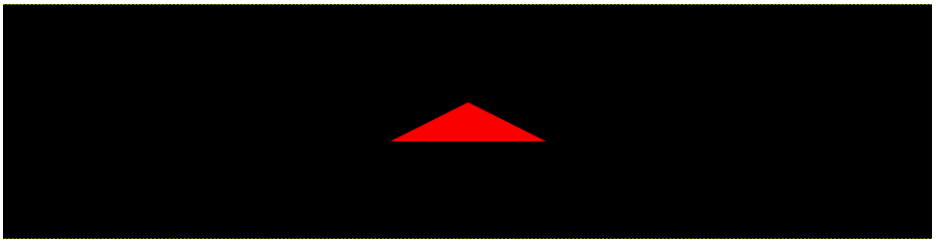

Figura 5.5 Primer triángulo con otra resolución.

5.1.5 Intersección con una esfera

Una vez se estableció el choque triángulo-rayo se exploraron otras formas de choque. El choque de una figura tridimensional es fácil de calcular si se cuenta con su ecuación implícita, en el caso de la esfera el algoritmo de intersección es el siguiente.

Una esfera puede definirse con un punto y un radio. Una de las formas de calcular la intersección con la esfera es de la forma analítica, partiendo de la ecuación implícita:

$$x^2 + y^2 + z^2 = R^2 \tag{5.32}$$

O:

$$|\vec{P}|^2 = R^2 \tag{5.33}$$

$$|\vec{o} + t \cdot \vec{d}|^2 = R^2 \tag{5.34}$$

Donde t es un escalar cuya magnitud es la distancia hasta la esfera:

Si la esfera no está centrada en el origen hay que resolver la siguiente ecuación implícita:

$$|\vec{P} - \vec{C}|^2 = R^2 \tag{5.35}$$

Donde $\vec{C}$ es el punto centro u origen de la esfera. Resolviendo nos queda una ecuación de segundo grado cuyos coeficientes a, b y c de la solución general son:

$$a \cdot t^2 + b \cdot t + c = 0 \tag{5.36}$$

$$a = |\vec{d}|^2 = 1 \tag{5.37}$$

$$b = 2\vec{d} \cdot (\vec{o} - \vec{C}) \tag{5.38}$$

$$c = \left|\vec{o} - \vec{C}\right|^2 - R^2 \tag{5.39}$$

cuya solución nos dará dos t, una más lejana y otra más lejana dependiendo de su signo, pero siempre el de menor valor absoluto, si la solución es compleja no hay intersección. Resolver la posición en la esfera del punto se puede hacer fácilmente usando coordenadas esféricas y calculando los ángulos en la esfera. Y la normal es el vector normalizado $\vec{P} - \vec{C}$.

Aplicando el algoritmo el resultado que se obtuvo es el mostrado en la Figura 5.6.

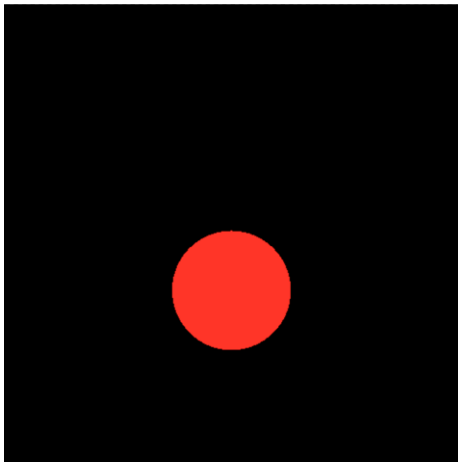

Figura 5.6 Esfera.

5.1.6 Intersección con un box

El cálculo de la intersección con un box fue posterior, se aplicó al final de la segunda iteración, pero se plasma en este apartado para mantener una consistencia de los temas que trata de la intersección con los rayos.

Los boxes son polígonos rectangulares con sus caras paralelas al sistema de referencia, coordenadas cartesianas. Un box está delimitado por dos vértices, uno con los valores mínimos y otro con los valores máximos de los cuatro vértices que la conformarían. Se usan para crear hitboxes o bounding boxes que se usaron más adelante.

La forma de resolver este problema de intersección, donde se tiene dos puntos $\overrightarrow{B1}$ y $\overrightarrow{B0}$, es descomponer el problema para cada una de las coordenadas cartesianas. Calculando la distancia de intersección para cada recta que lo compone.

Para calcular el punto de intersección por cada coordenada se hace uso de la ecuación explicita de la recta.

$$y = m \cdot x + b \tag{5.40}$$

Para la línea constante que representa un punto tenemos.

$$y = \mathrm{B0}_x \tag{5.41}$$

Para la línea que representa el rayo en la coordenada x:

$$y = \mathrm{t} \cdot \mathrm{d}_x + o_x \tag{5.42}$$

Luego la distancia es:

$$\mathrm{B0}_x \quad = \mathrm{t} \cdot \mathrm{d}_x + \mathrm{o}_x \tag{5.43}$$

$$t_{x0} = \frac{\mathrm{B0}_x \ - o_x}{d_x} \tag{5.44}$$

Y los cálculos para el resto de casos es similar dando lugar a:

$$t_{x0} = \frac{\mathrm{B0}_x - o_x}{d_x} \tag{5.45}$$

$$t_{x1} = \frac{\mathrm{B1}_x - o_x}{d_x} \tag{5.46}$$

$$t_{y0} = \frac{\mathrm{B0}_y - \mathrm{o}_y}{d_y} \tag{5.47}$$

$$t_{y1} = \frac{\mathrm{B1}_y - o_y}{d_y} \tag{5.48}$$

$$t_{z0} = \frac{\mathrm{B0}_z - o_z}{\mathrm{d}_z} \tag{5.49}$$

$$t_{z1} = \frac{\mathrm{B1}_z - o_z}{d_z} \tag{5.50}$$

Por componentes, si alguna distancia mínima de una coordenada es mayor que otra máxima de otra, no se producirá el choque. Es decir. Si min (t_{x0}, t_{x1})> max(t_{y0}, t_{y1}) o min(t_{y0}, t_{y1}) >max (t_{x0}, t_{x1}), etc, no se produce un choque, en otro caso sí, como se muestra en la Imagen 5.7:

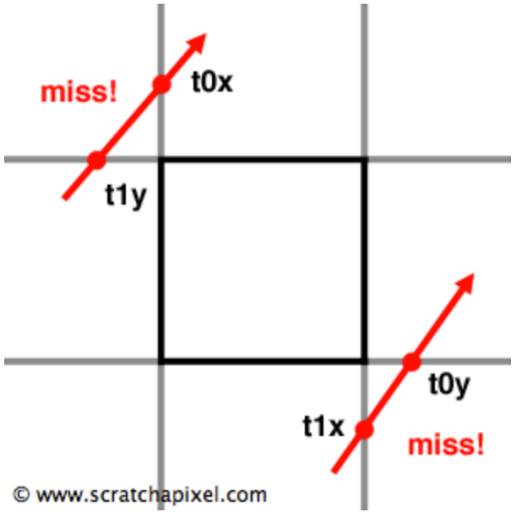

Figura 5.7 Dos rayos fallando un box **[71]**.

En el programa se usó el algoritmo publicado en el artículo *"An Efficient and Robust Ray–Box Intersection Algorithm"de Amy Williams, Steve Barrus, R. Keith Morley y Peter Shirley University of Utah*. Para el cálculo de intersecciones con el box de forma eficiente [71].

5.1.7 Generación de una cámara

Para solventar el problema de escalados con respecto al tamaño del lienzo de la cámara y poder aplicar movimientos de cámara fue necesario definir la cámara.

Una cámara se define como un punto donde se sitúa el observador y una pantalla o lienzo donde se proyecta la imagen.

La cámara se define principalmente con 3 vectores y un vértice. Los vectores son:

- Forward, vector que apunta a la dirección a la que apunta la cámara.
- Up, vector que señala el eje vertical de la cámara.
- Near, vector o la distancia del canvas a la cámara.

El vértice es el origen o posición de la cámara.

Por motivos geométricos, en el programa, al aplicar las transformaciones, giros o movimientos, el vértice origen marca la posición del canvas y el near apunta a la posición de la cámara.

La cámara se muestra comúnmente en NCD una forma normalizada, estándar para la mayor parte de las APIs. Donde el eje más corto mide un intervalo de [-1,1] y el otro más largo es proporcional según la aspect ratio.

Respetando estás proporciones se aplican el FOV, que tiene sus componentes vertical y horizontal según el aspecto ratio también.

El FOV (Field of View o Campo de Visión) se refiere al ángulo de visión que la cámara puede capturar.

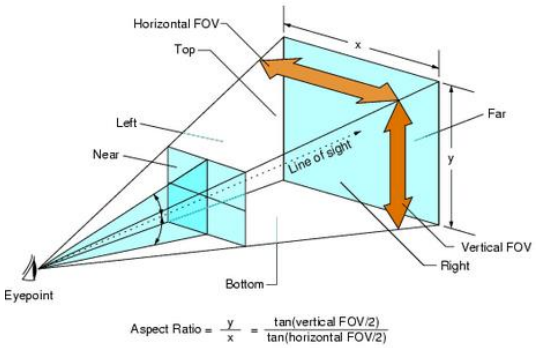

Figura 5.8 Esquema cámara y FOV **[73]**.

Para el cálculo de emisión de rayos (ray casting) se obtiene un nuevo algoritmo, donde la posición del pixel viene definida por:

$$0 \leq i < ancho \tag{5.51}$$

$$0 \leq j < alto \tag{5.52}$$

$$x_i = \frac{1 - 2(i + 0.5)}{\text{ancho}} \tag{5.53}$$

$$y_j = \frac{1 - 2(j + 0.5)}{alto} \tag{5.54}$$

al aplicar el FOV obtenemos una nueva escala:

$$\text{escala}_y = \tan(\frac{\text{FOV}_y}{2}) \tag{5.55}$$

luego:

$$y_j = \frac{1 - 2(j + 0.5)}{\text{alto}} \cdot \text{escala}_y \tag{5.56}$$

Por otro lado:

$$\text{escala}_x = \tan\left(\frac{\text{FOV}_x}{2}\right) = \frac{anchura}{altura} \cdot \tan(\frac{\text{FOV}_y}{2}) \tag{5.57}$$

Luego:

$$x_i = \frac{1 - 2(i + 0.5)}{\text{ancho}} \cdot \text{escala}_x = \frac{1 - 2(i + 0.5)}{\text{ancho}} \cdot \text{escala}_y \cdot aspect\ ratio \tag{5.58}$$

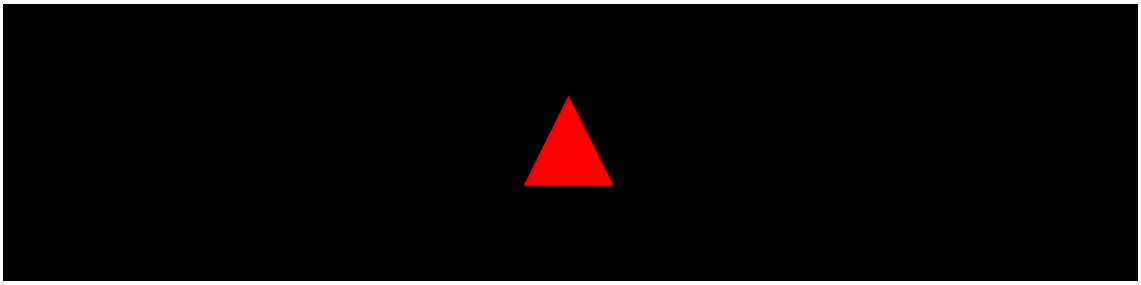

Figura 5.9 Renderizado de un triángulo sin alterar con una resolución muy ancha.

Para pasar la cámara del espacio de cámara NCD al espacio del mundo, se requiere multiplicar las íes por el vector up y las exis por el producto vectorial $\overrightarrow{right}=\overrightarrow{up}x\overrightarrow{\text{forward}}$. Si se requiriere aplicar movimientos o giros, se trata de aplicar las matrices a estos vectores y vértices y luego multiplicarlos a las coordenadas de cámaras.

$$(\text{x},\text{y},\text{z})_{camara\ mundo} = ((\overrightarrow{right}),(\overrightarrow{up}),(\overrightarrow{\text{forward}}))_{3x3} \cdot (\text{x}_i,\text{y}_j,\text{z})_{camara\ NCD} \qquad (5.59)$$

Los rayos se constituyen con la siguiente ecuación, para un rayo i cualquiera se tiene:

$$\vec{r}_i = \vec{o} + \lambda\vec{d} \qquad (5.60)$$

$$\vec{r}_i = \overrightarrow{\text{cam}} + \lambda \cdot ((\text{x}_i,\text{y}_j,\text{z})_{camara\ mundo} - \overrightarrow{\text{cam}}) \qquad (5.61)$$

Donde $\overrightarrow{\text{cam}}$ es el origen de la cámara.

Se creó una clase llamada Camera, que almacenaba información sobre sí misma, permitiendo aplicar movimientos o giros, y era capaz de proporcionar una dirección y un origen para los rayos según el píxel (i, j) dado. La Figura 5.9 muestra el resultado que se obtuvo.

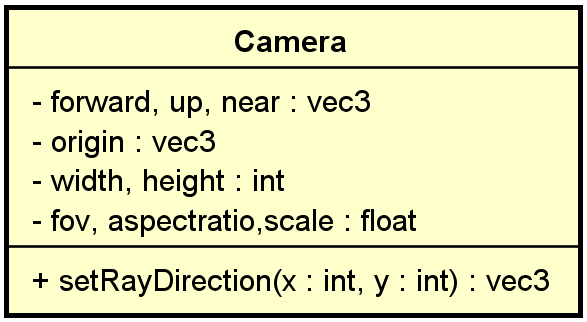

Figura 5.10 Clase Camera.

5.1.8 Mallas poligonales

Una vez establecidos los choques con distintas figuras y la cámara se dio paso al segundo subobjetivo; el estudio y la inclusión de las mallas poligonales.

Las mallas poligonales, también conocidas como superficies poligonales u objetos 3D, están definidas por sus facetas, que a su vez se definen por vértices. Por ejemplo, un

cubo está compuesto por 6 facetas cuadradas, y cada una de estas facetas está definida por cuatro vértices. En total, un cubo tiene 8 vértices únicos, pero si consideramos los vértices de cada cara individualmente, se podría pensar que tiene 24 vértices (6 caras x 4 vértices), aunque en realidad muchos de estos vértices son compartidos entre las caras.

En computación gráfica, se definen los puntos que forman el objeto. Luego, se define el número de caras, y para cada cara se especifica el número de vértices que la componen, numerados de manera consistente, normalmente en sentido antihorario, para asegurar una orientación coherente. Esto se hace utilizando referencias a los vértices del objeto. Por ejemplo, un cubo cuyos vértices son: {0,0,0}, {1,0,0}, {1,1,0}, {0,1,0}, {0,0,1}, {1,0,1}, {1,1,1}, {0,1,1}, tiene 6 caras compuestas por 4 vértices cada una. Las caras se pueden definir como una lista de índices que hacen referencia a los vértices: {0,1,2,3}, {4,5,6,7}, {0,1,5,4}, {2,3,7,6}, {0,3,7,4}, {1,2,6,5}.

Además de la geometría, cada vértice de la malla puede tener asociada información adicional, como un vector normal.

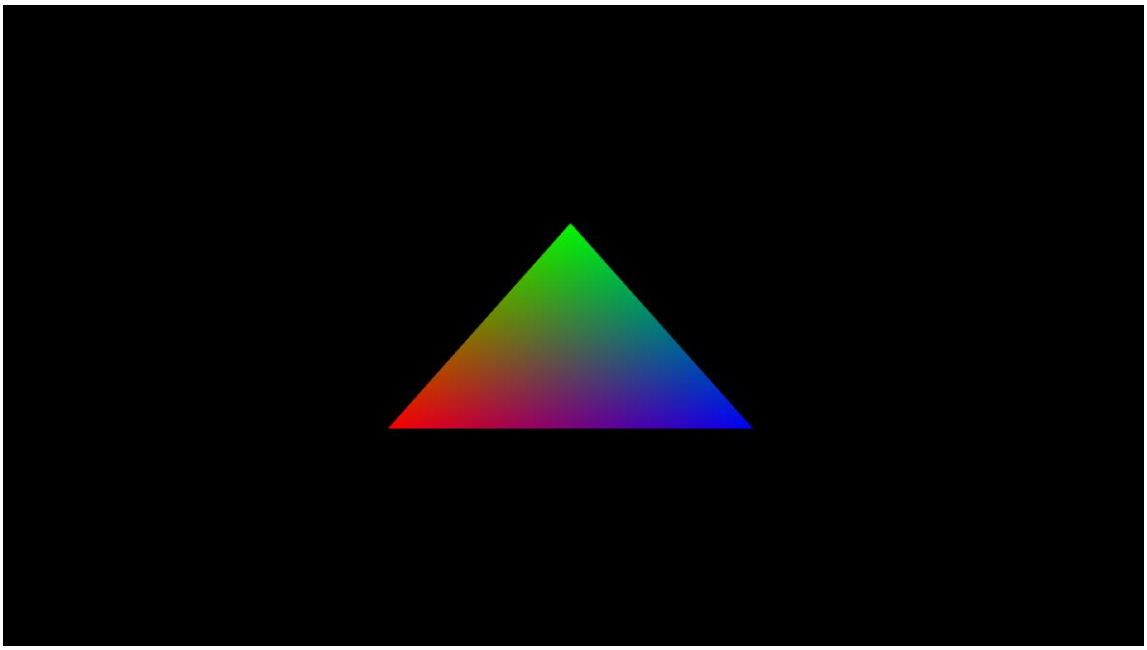

Figura 5.11 triangulo coloreado según sus coordenadas.

En cuanto a las unidades de medida, un estándar común adoptado por Pixar y otras compañías es utilizar centímetros (cm) como unidad base.

Para poder distinguir las caras, se hizo que el color del triángulo anterior dependiera de las coordenadas baricéntricas del mismo, u, v y w. El color del rayo, (r,g,b), se convirtió en (u,v,w).

Para calcular la intersección con una malla poligonal, primero se descomponían los polígonos de la malla en triángulos. Aunque la malla pudiera tener otro tipo de caras, el render sabía calcular las intersecciones con triángulos. Para determinar con qué triángulo se intersecaba cada rayo, se calculaba la colisión utilizando el algoritmo *Möller-Trumbore* para todos los triángulos, y el de menor distancia era el que prevalecía.

Calcular todas las intersecciones tiene coste $O(n^2)$, recorrer todos los rayos y recorrer todos los triángulos. Aunque todos esos cálculos son independientes y paralelizables.

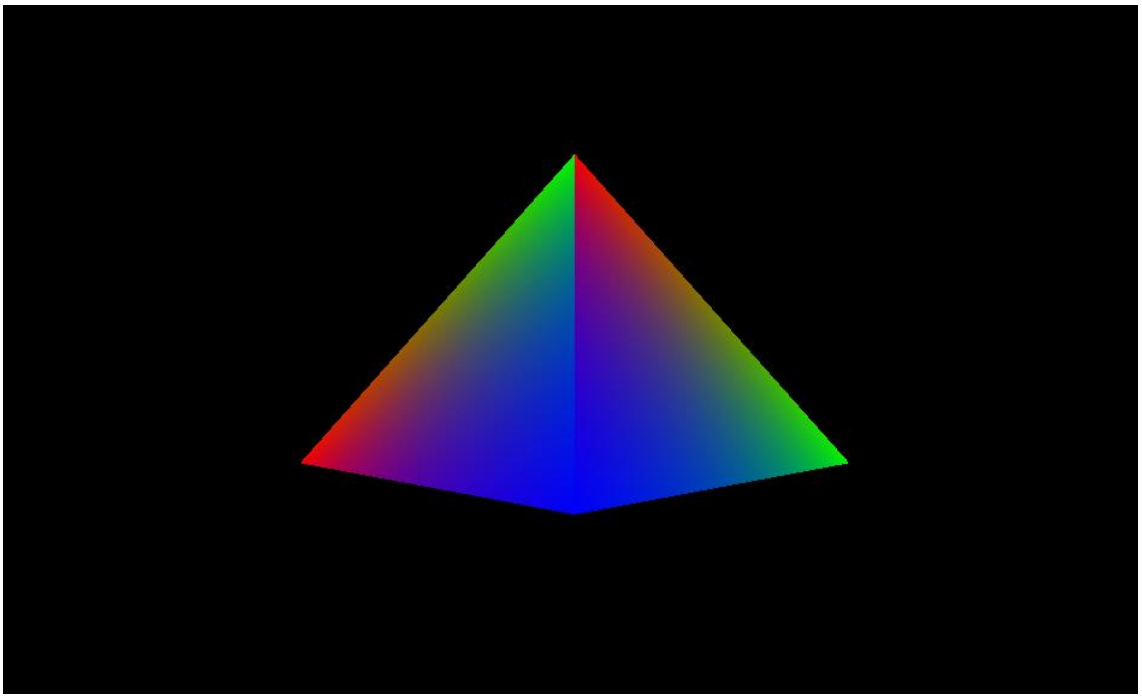

Figura 5.12 Tetraedro coloreado según las coordenadas de sus triángulos.

La estructura de datos que puede definir este tetraedro es:

Numero de caras: 4
Numero de vértices por cara: 3 3 3 3
Índices a los vértices: 0 3 1 0 2 3 0 1 2 1 2 3

Vértices: -0.5 0.0 -1.0 -1.0 5.0 1.0 -1.0 5.0 0.0 1.0 5.0 (se sumó 5cm en el eje z para obtener la imagen)

Al introducir un tetraedro, se obtuvo la visualización del tetraedro, Figura 5.12. En lugar de profundizar en la creación de un lector de mallas, se utilizó la librería ASSIMP, que está optimizada para leer todo tipo de archivos. Con ASSIMP, fue fácil convertir toda una escena en triángulos y obtener la información de los triángulos que componen los objetos o las escenas, incluyendo sus vértices, normales y texturas. Aquí, las normales se entienden como normales simuladas, lo cual se explica más adelante. Se utilizó ASSIMP para leer escenas en formato FBX y verterlas en las estructuras de datos propias.

5.1.9 ASSIMP

Assimp fue desarrollada inicialmente por *Marcus "acgessler" Geelnard* en 2006, con el objetivo de crear una herramienta simple y robusta para importar modelos 3D en diferentes formatos. Con el tiempo, la biblioteca ha evolucionado para soportar más de 40 formatos de archivos 3D, incluyendo populares como OBJ, FBX, Collada (DAE), STL, 3DS, entre otros.

La comunidad de código abierto ha contribuido significativamente a su desarrollo, y la biblioteca está licenciada bajo la licencia BSD. Esto la convierte en una opción atractiva para muchos desarrolladores y proyectos, tanto comerciales como de código abierto.

La estructura de escena de Assimp es el núcleo de su funcionalidad, donde organiza toda la información contenida en un archivo 3D importado. Esta estructura se divide en varios componentes clave:

- **Escena**: El contenedor que guarda todo lo que hay en el archivo 3D, como los objetos, las luces, las cámaras, las mallas, y más.
- **Mallas (Meshes)**: Las mallas son las formas que componen los objetos en 3D. Contiene una estructura similar al tetraedro anterior y aparte añade vectores para las normales, uvs para texturas, y otras, todas ellas relacionadas con los

vértices. Cada vértice tiene asociada una normal y una coordenada de textura uv.

- **Materiales**: Los materiales son las propiedades que le dan color a los objetos.
- **Nodos**: Los nodos es una estructura jerárquica en árbol que guarda punteros a las mallas para aplicar movimientos, giros y otras trasformaciones de forma iterativa recorriendo el árbol.

Se creó un conjunto de estructuras de datos similares: una clase Triángulo, una clase Malla y una clase Escena. La escena almacenaba mallas y materiales; las mallas almacenaban triángulos, vértices, índices de los vértices y un puntero al material; y los triángulos almacenaban punteros a los vértices y al material. El material contenía información sobre el color del objeto, entre otros aspectos.

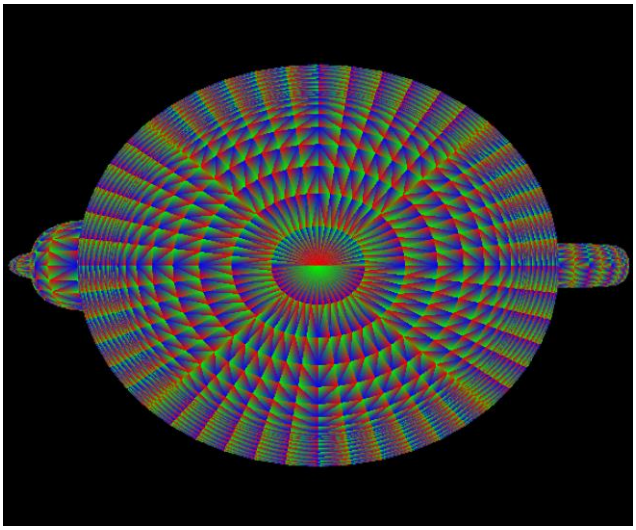

Figura 5.13 Una tetera desde abajo.

En el momento de renderizar la tetera [73] no se habían introducido los materiales y otros elementos que se fueron añadiendo consecutivamente. El color que se muestra depende de las coordenadas baricéntricas de cada triangulo.

Movimiento de mallas

Teniendo una tetera desubicada surgió la necesidad de añadir movimientos, por lo que se indagó sobre cómo se obtienen y cómo es el estándar en la industria del 3D:

La tetera, triángulos, tetraedros, o cualquier malla en general, inicialmente se encuentra en el espacio de objeto, donde son el centro del universo. A estas figuras se les aplican movimientos, giros, escalados u otras transformaciones y pasan a estar en

el espacio de mundo. En las imágenes mostradas hasta ahora, a los objetos se les aplicó un movimiento en el eje z para que fueran visibles por la cámara.

En una escena normalmente se guardan los objetos en sus espacios de objeto y cuando se procesa la escena se colocan estratégicamente pasando a estar en el espacio mundo. Lo hacen mediante una estructura de datos en árbol, los nodos, que se va recorriendo desde el nodo padre pasando por los nodos hijos. Cada nodo guarda una matriz de transformación y mallas vinculadas. A lo largo del recorrido se aplica la transformación a las mallas vinculadas y se acumula mediante un producto matricial la transformación del nodo padre a la del nodo hijo. Luego, las mallas del nodo hijo sufren la transformación del nodo padre además de la de su nodo.

Una transformación, un movimiento, un giro, un escalado es una matriz cuadrada 4x4. Para aplicarla solamente hay que multiplicar el vector o el vértice por ella. En el caso de vectores sus componentes (x,y,z) deben ir acompañadas por un 0, (x,y,z,0) en el caso de un vértice por un 1 (x,y,z,1). Hay vectores como las normales que pueden ser modificados si se aplica escalado y deben ser multiplicados por la traspuesta de la matriz inversa de la transformación en lugar de la matriz de la transformación debido a sus propiedades ortonormales y para no producir una aberración.

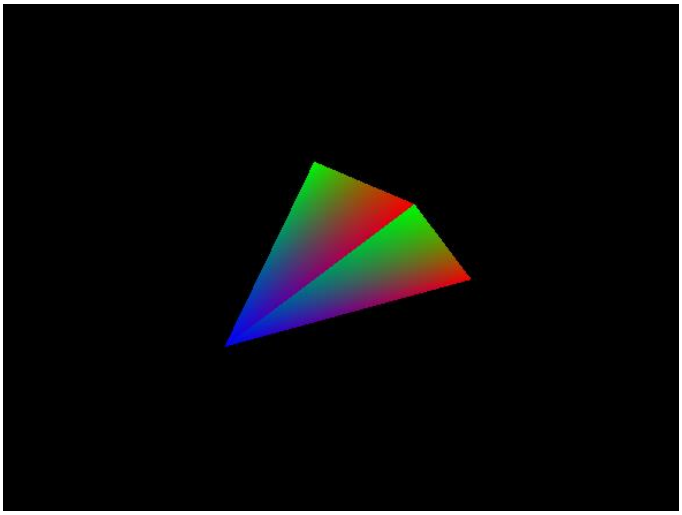

Figura 5.14 Giro de tetraedro.

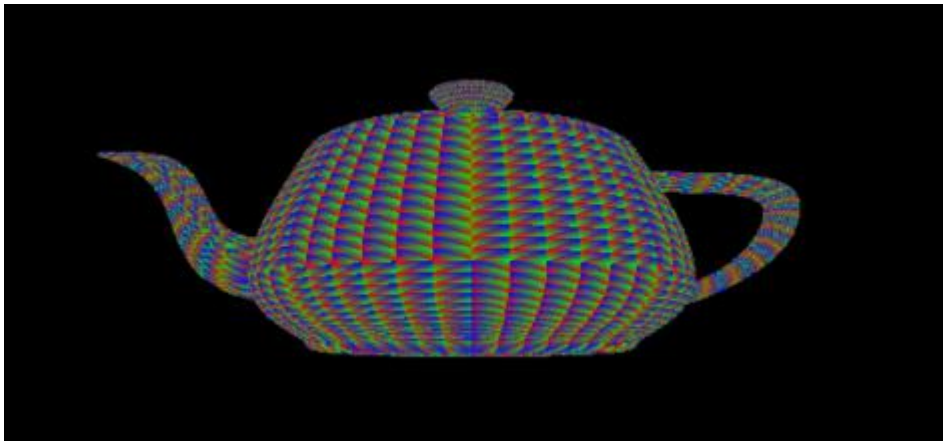

Figura 5.15 Tetera enderezada.

Los giros, movimientos y demás transformaciones de los objetos se establecieron mediante un método en la clase TriangleMesh. El recorrido de nodos se encuentra en un método en la clase Scene, mediante el recorrido de los nodos se aplican transformaciones sobre las instancias de Mesh. Una vez añadido a la aplicación se pudo obtener las Figuras 5.14 y 5.15 respectivamente.

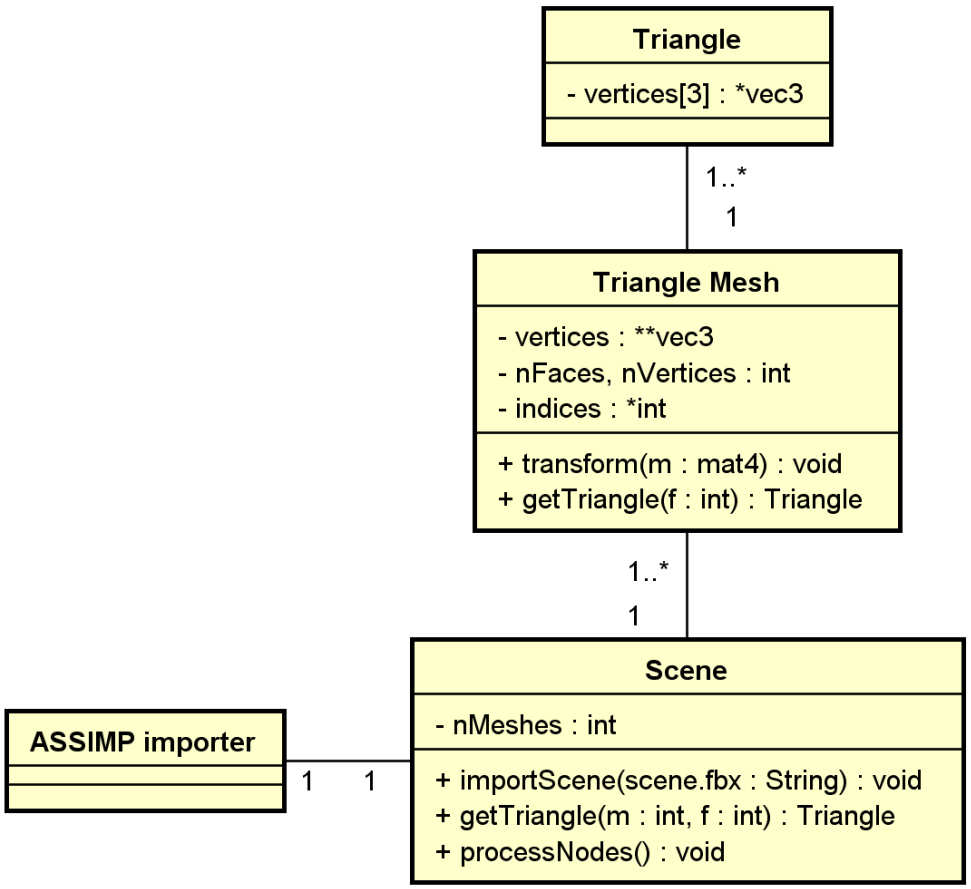

Figura 5.16 Simplificación del diseño en diagrama de clases.

Faltaba algo, color en los objetos, fue el momento de cambiar de subobjetivo y de aplicar el sombreado de Phong.

5.1.10 Implementación del sombreado (shading)

Se comenzó implementando el modelo de Phong, que calcula la luz total en tres componentes: ambiental, difusa y especular. cómo se vio en el apartado 2.4.

$$I = I_{ambiente} + I_{difusa} + I_{especular} \tag{5.62}$$

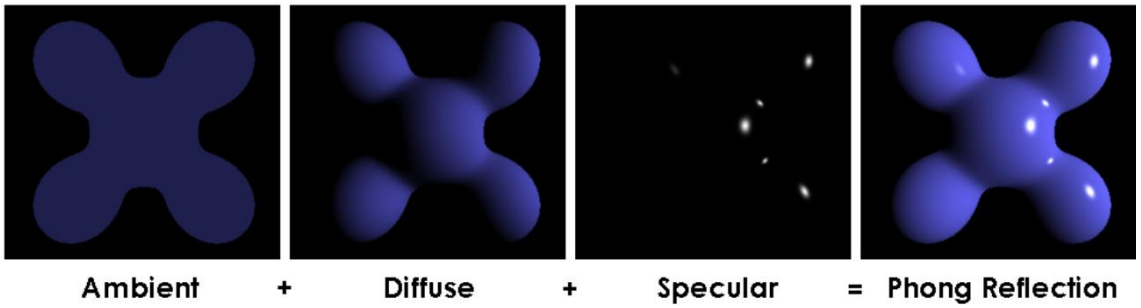

Figura 5.17 Ilustración visual de la ecuación de Phong **[75]**.

Nótese que el modelo de Phong usa intensidad luminosa en lugar de radiancia.

Implementación del sombreado difuso

El primer paso fue implementar el sombreado difuso, que calcula la cantidad de luz que una superficie refleja en función del ángulo entre la normal de la superficie y la dirección de la luz. Este cálculo se basó en la siguiente fórmula:

$$I_{difusa} = k_d \cdot I_l \cdot \max(0, .\vec{N} \cdot \vec{L}) \tag{5.63}$$

Como decisión personal, para traducirla al código, se hizo que el coseno de Lambert estuviera en valor absoluto, porque es más rápido y mira ambas caras de un plano. La dirección del vector normal debería cambiar dependiendo de la cara en la que se choca.

$$Color_{rayo} += Color_{albedo} \cdot k_d \cdot I_i(f) \cdot abs\,(\vec{N} \cdot \vec{L}) \tag{5.64}$$

Se utilizó el producto escalar de la dirección de luz incidente y la normal en cada punto de la superficie para determinar la cantidad de luz difusa reflejada.

Interpolación de normales

Se añadió un vector normal ligado a cada vértice de la malla. En un inició todas las informaciones e interpolaciones referentes al choque del rayo se guardaban en el propio rayo y luego se creó la estructura Hit.

Para mejorar la suavidad de la iluminación en las superficies, se implementó la interpolación de normales. Se interpolaron las normales en cada punto de choque utilizando las coordenadas baricéntricas (u, v, w) de los triángulos, con la fórmula:

$$\vec{N} = u \cdot \vec{N}[0] + v \cdot \vec{N}[1] + w \cdot \vec{N}[2] \qquad (5.65)$$

Esto permitió suavizar la transición de la iluminación entre píxeles, evitando bordes bruscos en los objetos y mejorando la calidad visual del sombreado.

Figura 5.18 Tetera de Utah con sombreado de Phong (sólo difuso).

En otros modelos como Goraurd no hay interpolación de normales.

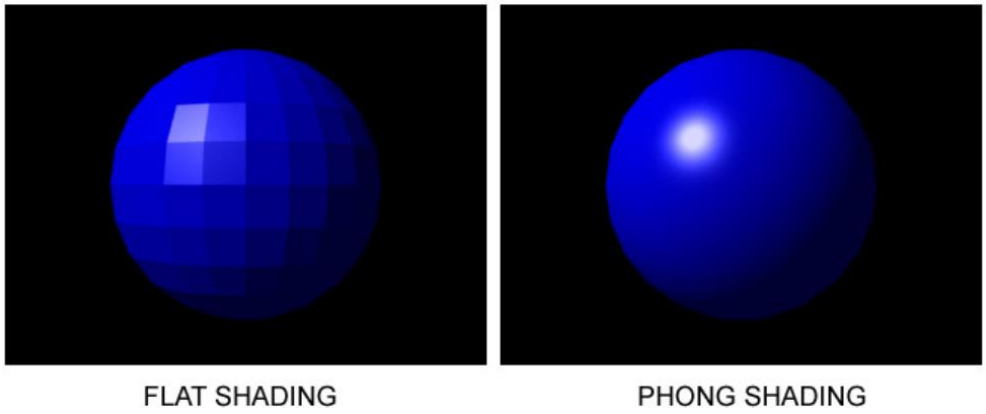

Figura 5.19 Sombreado Gouraud vs Phong **[76]**.

Implementación de struct material.

En el proyecto, se añadió una estructura (Struct) llamada Material ligada a una malla. En ella se almacenaba información como el color del material, la constante k_D, la constante k_S, el exponente especular n y el color especular del modelo Phong visto en el apartado 2.4.

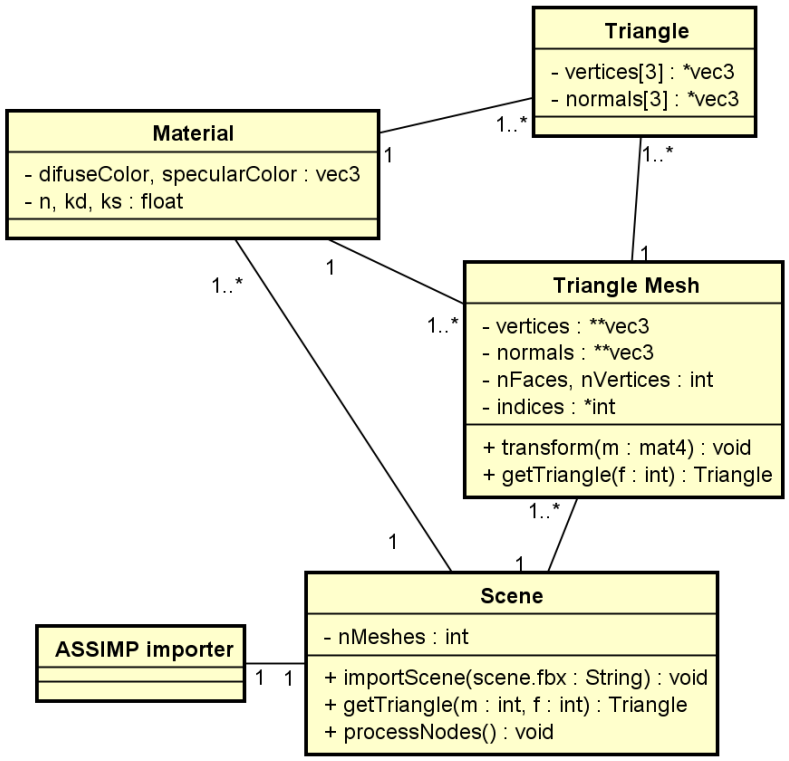

Figura 5.20 Simplificación del diseño en diagrama de clases.

Implementación del sombreado especular

El siguiente paso fue implementar el sombreado especular, que añade el brillo visible en superficies brillantes cuando la luz se refleja directamente hacia el observador. El brillo especular se calculó utilizando el vector de reflexión $\vec{R}$ y la dirección de la cámara $\vec{V}$:

$$I_{especular} = k_s \cdot I_l \cdot max(0, (\vec{R} \cdot \vec{V})^n) \quad (5.66)$$

En este caso no se calculó el valor absoluto del producto vectorial, no se puede o debe porque no depende del vector normal, quedando así la ecuación usada:

$$Color_{rayo} += Color_{especular} \cdot k_d \cdot I_i(f) \cdot max(0, (\vec{R} \cdot \vec{V})^n) \quad (5.67)$$

El vector de reflexión $\vec{R}$ se calculó a partir de la normal y la dirección de la luz $\vec{L}$.

$$\vec{R} = 2 \cdot (\vec{N} \cdot \vec{L}) \cdot \vec{N} - \vec{L} \quad (5.68)$$

Este cálculo permitió ajustar la intensidad y distribución del brillo en función del material y la posición del observador.

En este caso, la estructura Material almacenaba los parámetros del brillo especular, como ks y el exponente n, que fueron ajustados para controlar la intensidad y el tamaño del reflejo en la superficie.

Los resultados fueron los siguientes (Figura 5.21):

Figura 5.21 Render de una planta y su componente difusa y especular.

Ampliación a Blinn-Phong

Finalmente, se optimizó el cálculo del brillo especular implementando el modelo Blinn-Phong, que utiliza el vector halfway $\vec{H}$ en lugar del vector de reflexión. Esto simplificó el cálculo de la componente especular:

$$I_{especular} = k_s \cdot I_l \cdot max(0, (\vec{H} \cdot \vec{N})^n) \tag{5.69}$$

Al estar presente el vector normal se usó el valor absoluto del producto escalar:

$$Color_{rayo} += Color_{especular} \cdot k_s \cdot I_i(f) \cdot \text{abs}\ (\vec{N} \cdot \vec{H})^n \tag{5.70}$$

Donde $\vec{H}$, el vector halfway es:

$$\vec{H} = \frac{\vec{V} + \vec{L}}{|\vec{V} + \vec{L}|} \tag{5.71}$$

Blinn-Phong resultó más eficiente computacionalmente porque evita el cálculo del vector de reflexión $\vec{R}$. El brillo especular fue generalmente más suave y se distribuyó de manera más natural en la superficie. Se considera más adecuado para superficies rugosas o menos perfectas.

Al aplicar Blinn-Phong se obtuvieron imágenes con brillos especulares más anchos, como se muestra en las Figuras 5.22 y 5.23 de una planta [76].

Figura 5.22 Diferentes brillos especulares Blinn-Phong y Phong.

Figura 5.23 Planta Blinn-Phong.

5.1.11 Iluminación directa

Hasta este punto, el ray caster contaba con la emisión de rayos, el cálculo de intersecciones, un sistema de escenografía y un modelo de sombreado. El siguiente paso en la implementación fue añadir luces directas y un sistema de sombras.

Para implementar la iluminación directa, se añadió un proceso que calcula la luz que llega directamente desde una fuente de luz a la superficie impactada por un rayo. Se lanzó un rayo adicional desde el punto de impacto hacia la fuente de luz, conocido como shadow ray o rayo sombra. Este rayo se utiliza para verificar si existe una línea de visión directa entre el punto y la luz.

Si el shadow ray no encuentra ninguna obstrucción (es decir, no se interseca con ningún objeto en el camino), se considera que el punto está iluminado directamente por la fuente de luz. Este cálculo se integró con el sistema de sombreado existente para ajustar la intensidad y el color de la luz que afecta al punto.

El shadow ray se definió como un rayo que tiene origen en el punto de choque y se dirige hacia la luz.

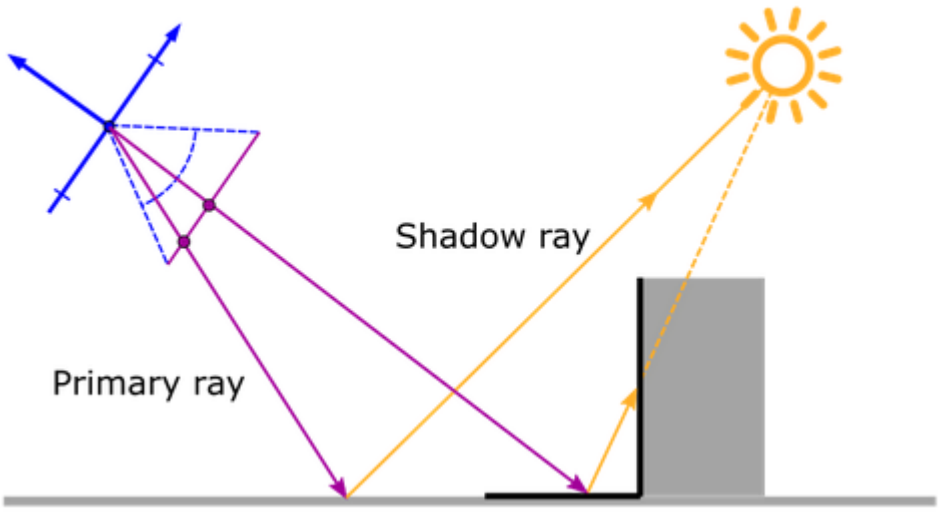

Figura 5.24 Generación de sombras **[78]**.

Al crear rayos de sombra, se variaba mínimamente su origen desde el punto de choque del rayo primario para evitar que se volviera a chocar con la superficie desde donde comenzaba. Luego, se calculaba con el algoritmo Möller-Trumbore si se producía el choque y a qué distancia.

5.1.12 Luces

Para implementar el modelo de Phong se crearon las luces. Con estas se calcula el vector $\vec{L}$, el color de la luz y la intensidad lumínica que se muestran en las ecuaciones de Phong. En el programa se definieron de la siguiente forma.

Luz Puntual (Point Light)

Una luz puntual emite luz en todas las direcciones desde un solo punto en el espacio. No tiene un tamaño físico, un volumen, por lo que se considera una fuente de luz infinitamente pequeña.

Características: Genera sombras duras y bien definidas. La intensidad de la luz disminuye con el cuadrado de la distancia, debido a la atenuación y de forma cuadrática inversa, al distribuirse la luz por una superficie esférica y con radio igual a la distancia, $A = 4 \cdot \pi \cdot r^2$.

Aplicaciones: Simula pequeñas fuentes de luz como bombillas, faros de coches, o cualquier otro punto luminoso.

Implementación: La luz se define con un punto, un color y una intensidad. Si la distancia del shadow ray a la luz es mayor que la distancia a otro objeto se produce una sombra. También se produce atenuación con la distancia.

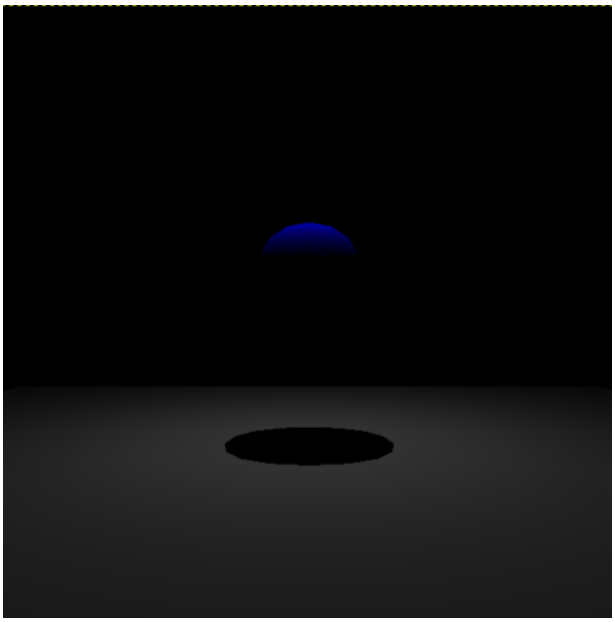

Figura 5.25 Luz puntual.

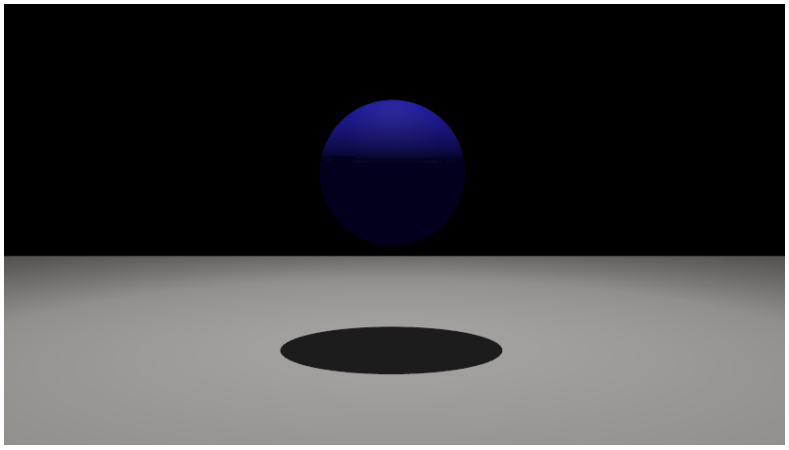

Figura 5.26 Renderizada luz de punto con luz ambiental.

Luz Direccional (Directional Light)

Una luz direccional emite rayos de luz paralelos en una dirección específica. Se asume que la fuente de luz está infinitamente lejos, como el sol, por lo que la luz llega en líneas rectas paralelas.

Características: No tiene atenuación con la distancia y produce sombras muy consistentes y uniformes, ideales para simular la luz solar.

Aplicaciones: Utilizada para simular la luz del sol o cualquier fuente de luz que esté muy lejos del escenario.

La luz se define con un vector dirección, un color y una intensidad.

Implementación: Si el shadow ray con dirección -$\vec{L}$, que se dirige hacia a la luz, choca con otro objeto, entonces se produce una sombra.

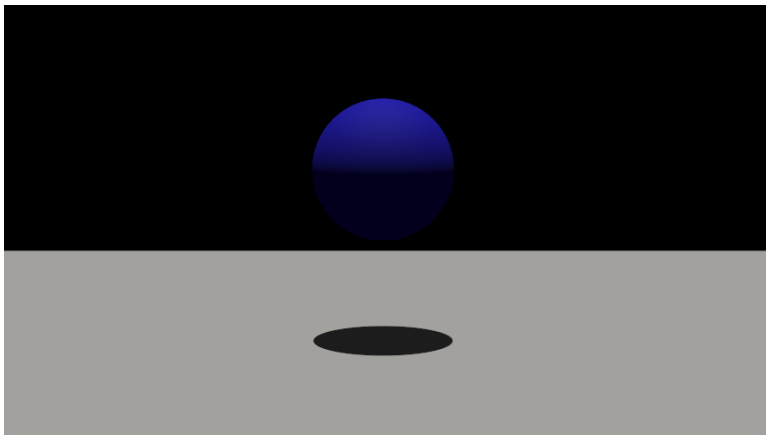

Figura 5.27 Renderizada una luz direccional con luz ambiental.

Luz de foco (Spot Light)

Una Spot Light o luz de foco es un tipo de fuente de luz que emite luz desde un punto en una dirección específica formando un cono de luz.

Características: La luz de foco emite luz en un cono de luz, con un ángulo interno, spot angle, donde la luz permanece con la misma intensidad y un ángulo externo, Falloff, donde se produce una atenuación gradual simulando un halo de scattering. Las sombras generadas por una Spot Light son generalmente nítidas en los bordes del cono de luz y pueden volverse más suaves hacia los bordes exteriores del cono dependiendo del ángulo de penumbra.

Aplicaciones: Utilizada para simular una linterna, un foco, luces de coche, etc.

Implementación: La luz de foco se define con un punto, un vector dirección y dos ángulos. El shadow ray se dirige al punto y se produce atenuación si se encuentra en el falloff, si choca contra otro objeto o está fuera del ángulo externo se produce una sombra.

$$Atenuación_{fallof} = \frac{cos(angulo\ rayo\ sombra) - cos(angulo\ interno)}{cos(angulo\ interno) - cos(angulo\ externo)} \qquad (5.72)$$

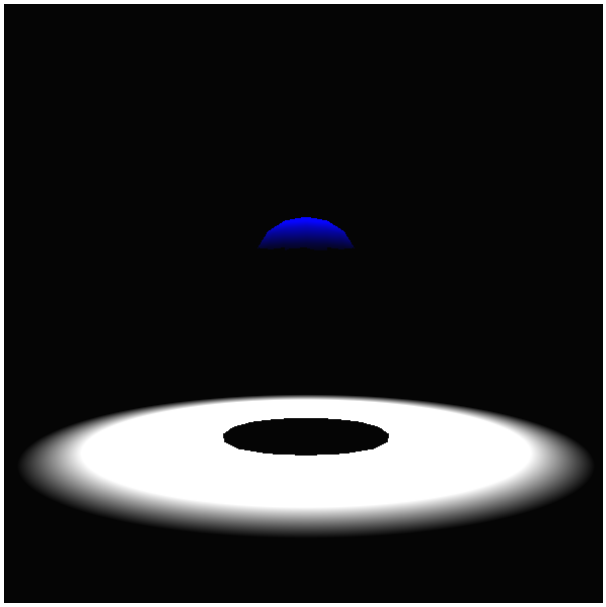

Figura 5.28 Luz de foco.

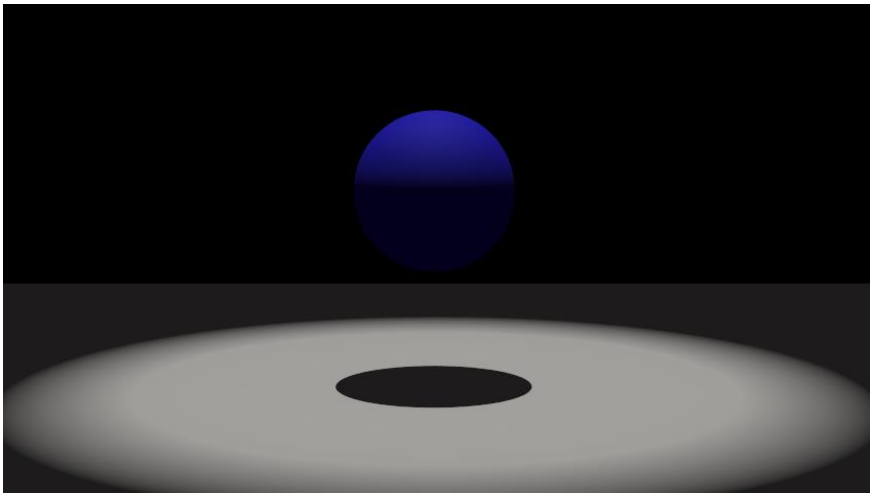

Figura 5.29 Luz de foco con luz ambiental.

Se creó una clase llamada Light y cuatro clases que heredaban de ella: DistantLight, SpotLight, PointLight y AmbientLight. Se implementaron luces ambientales para evitar que la imagen fuera completamente negra en las sombras duras, aunque las sombras suaves reales se abordaron en la última parte del proyecto. Con esto, se finalizó la generación de un ray caster simple.

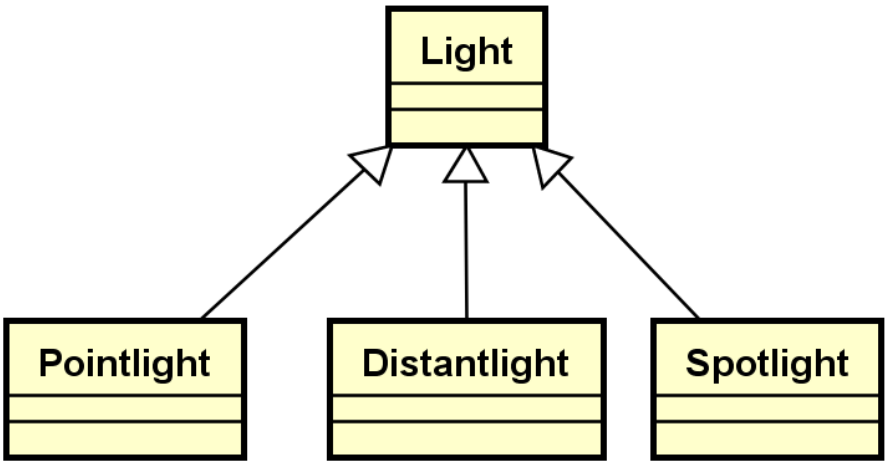

Figura 5.30 Simplificación del diseño en diagrama de clases.

5.1.13 Conclusiones primera iteración

En este sprint se estudiaron e implementaron los fundamentos y técnicas clave del ray casting, comenzando con un enfoque básico para el cálculo de intersecciones de rayos con primitivas geométricas como triángulos, esferas y bounding boxes. Se desarrolló una cámara virtual y se integró la librería ASSIMP para la importación de

modelos poligonales, lo que permitió gestionar y manipular mallas poligonales de manera eficiente.

En cuanto a sombreado, se incluyeron modelos de Phong y Blinn-Phong. Además, se implementaron los conceptos esenciales de iluminación directa, incluyendo la creación de sombras y la incorporación de diferentes tipos de luces: puntuales, direccionales y de foco. Con estos elementos, se cumplimentaron los requisitos del primer sprint del proyecto para la generación de escenas tridimensionales.

5.3 Segunda iteración

5.3.1 Introducción de la segunda iteración

En una segunda iteración, o sprint de 4 semanas, al ray caster se le añadieron nuevas funcionalidades, como el sobremuestreo, la lectura de mapas de texturas, técnicas de sombreado basadas en la física (PBR), reflexión y refracción de rayos, y técnicas de aceleración del software como el cálculo paralelo, la malla de Fujimoto y las bounding boxes. Una vez se ampliaron estas funcionalidades, el programa pudo considerarse un Whitted Ray Tracer.

Este sprint de 4 semanas sufrió un retraso de 2 semanas a la hora de la aplicación del requisito funcional de las texturas, se explicará más adelante.

5.3.2 Sobremuestreo

El primer problema en abordarse fue el del sobremuestreo (supersampling):

Como se vio en el marco teórico, el sobremuestreo es una técnica de anti-aliasing que mejora la calidad de las imágenes al renderizarlas a una resolución más alta de la necesaria y luego reducirlas al tamaño final, promediando los píxeles para suavizar bordes y reducir el aliasing y el ruido.

Para implementar sobremuestreo se emitió más de un rayo por pixel en la pantalla y se promediaron sus colores en el pixel. Por ahora, solo se observaba el efecto de anti-aliasing de los dientes de sierra, pero en la iluminación global con Montecarlo, las imágenes perdían ruido al disminuir la varianza.

La implementación del sobremuestreo es relativamente sencilla, se introdujo un anidamiento más a la hora de generar rayos, reflejando un mínimo movimiento de la muestra en ese anidamiento. Para cada rayo en el mismo pixel (i, j) se tiene:

$$x_i = \frac{1 - 2(i + aleatorio1)}{\text{ancho}} \tag{5.73}$$

$$y_j = \frac{1 - 2(j + aleatorio2)}{alto} \tag{5.74}$$

Donde aleatorio 1 y 2 son números a aleatorios menores que 1 y mayores que 0.

Figura 5.31 Renderizado con sobremuestreo 1.

A su vez, se añadió otro anidamiento a la hora de escribir el archivo PPM y se promediaban los colores de los rayos que iban al mismo pixel. Los resultados se muestran en la Figura 5.31, Figura 5.32 y Figura 5.33.

Para mejorar el rendimiento es normal el uso de un jitter, es decir, que los puntos en el pixel sean aleatorios o pseudoaleatorios pero que no estén nunca muy juntos entre ellos estratificando las muestras.

Se usó el generador de Poisson disc de Sergey Kosarevsky [78] que se puede encontrar en github. El resultado es similar sólo que se necesitan menos sobremuestreo para una imagen con igual anti-aliasing.

Figura 5.32 Renderizado con sobremuestreo 10.

Figura 5.33 Renderizado con sobremuestreo 100.

Poisson Disc es un algoritmo utilizado para generar un conjunto de puntos distribuidos uniformemente en un espacio, de tal forma que no haya puntos demasiado cercanos entre sí ni demasiado separados. A diferencia de la distribución completamente aleatoria, que puede resultar en agrupamientos de puntos, el método Poisson Disc asegura que cada punto está a una distancia mínima de los demás, creando una distribución más uniforme.

5.3.3 Texturas

Una vez cumplido el subobjetivo se pasó al siguiente y se implementaron los mapas de texturas.

Como ya se vio en el marco conceptual, las texturas son imágenes o patrones que se aplican a la superficie de un modelo para darle detalles visuales, como colores, rugosidad o relieve. El proceso de aplicar estas texturas a un modelo 3D se llama texturización o texture mapping, donde las coordenadas 2D de la textura se asignan a la geometría 3D del objeto. Esto permite que la superficie del objeto tenga una apariencia más realista sin necesidad de aumentar la complejidad del modelo, simulando características como materiales, sombras, y reflejos mediante diferentes tipos de mapas como el de color difuso, normales, rugosidad, entre otros.

Los mapas de texturas son un papel de envoltura 2D que posee la malla, más allá de los materiales, la textura puede contener información detallada como brillos especulares, normales de la superficie simuladas, dando una riqueza espacial a la malla. Una textura puede ser un dibujo, archivo jpg, png, ppt… o incluso ser generada de forma procedural con un algoritmo.

Para implementar los mapas de texturas hay que calcular la posición de la textura o texel que se golpea hay que hallar sus coordenadas uvs que se encuentran en el espacio de la textura.

Cada vértice tiene asociado un punto 2D, UV, donde u y v van de 0 a 1. Coordenadas que representan la posición en la imagen 2D de arriba abajo e izquierda a derecha. Para calcular la coordenada uv del triángulo, se pondera las 3 coordenadas de textura ligadas a cada vértice del triángulo con las tres coordenadas baricéntricas del mismo:

$$\overrightarrow{UV} = u \cdot \overrightarrow{UV}_{\boldsymbol{triangulo}}[0] + v \cdot \overrightarrow{UV}_{\boldsymbol{triangulo}}[1] + w \cdot \overrightarrow{UV}_{\boldsymbol{triangulo}}[2] \qquad (5.75)$$

UV es la coordenada de textura, en el espacio de la imagen, mientras que u, v y w son las coordenadas en el triángulo según el peso de cada vértice del triángulo.

Para la implementación de texturas, se utilizó la librería STB. Se creó una clase Texture que se instanciaba para cada textura.

Las texturas se almacenaban en la escena, y cada material instanciado poseía punteros a las texturas correspondientes. Algunas texturas podían ser compartidas entre varios materiales. La librería STB_IMAGE almacenaba el archivo de la textura en una sola cadena de caracteres, permitiendo guardarlas en formato RGB o RGBA, donde A representa la transparencia.

Cada cuatro o tres caracteres estaban ligados a un píxel de la imagen. Hay mapas de texturas que solo usan un color como la de oclusión ambiental, otras son predominantemente de un color como la de mapa de normales que son muy azules ya que el azul representa el vector normal, mientras el rojo y el verde el tangente y el bitangente.

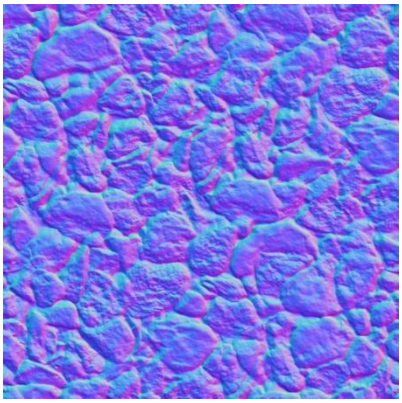

Figura 5.34 Mapa de texturas normales de una superficie [79].

Se implementaron texturas difusas y especulares que proporcionaban color difuso y especular a la malla. También se implementaron texturas ligadas al PBR, como las de normales, que añadían realismo a la superficie de la malla; las emisivas, que permitían que la malla emitiera luz (luz de área); las de oclusión ambiental, que simulaban la absorción de la luz en diferentes zonas; las de opacidad, que mostraban la transparencia; y otras dos texturas relacionadas con la metalicidad y rugosidad del material.

En la figura 5.36 se puede observar ligeramente la aplicación de textura de normales al aumentar los relieves del cubo [80].

Este objetivo sufrió un gran retraso de 2 semanas debido a que las coordenadas de las texturas se salían del rango [1,0], después de numerosas pruebas se comprobó que si una textura se sale de rango se supone que está dando una vuelta a la textura. Aplicando el siguiente código se solucionó el problema, listado de código 5-1.

Listado de código 5-1 Pseudocódigo

Coordenada de textura = uv[2]
if (uv.x > 1) uv.x = uv.x - (int)uv.x;
else if (uv.x < 0) uv.x = -uv.x + (int)uv.x;
if (uv.y > 1) uv.y = uv.y - (int)uv.y;
else if (uv.y < 0) uv.y = -uv.y + (int)uv.y;

Listado de código 5-2 Solución a las coordenadas de texturas fuera de rango.

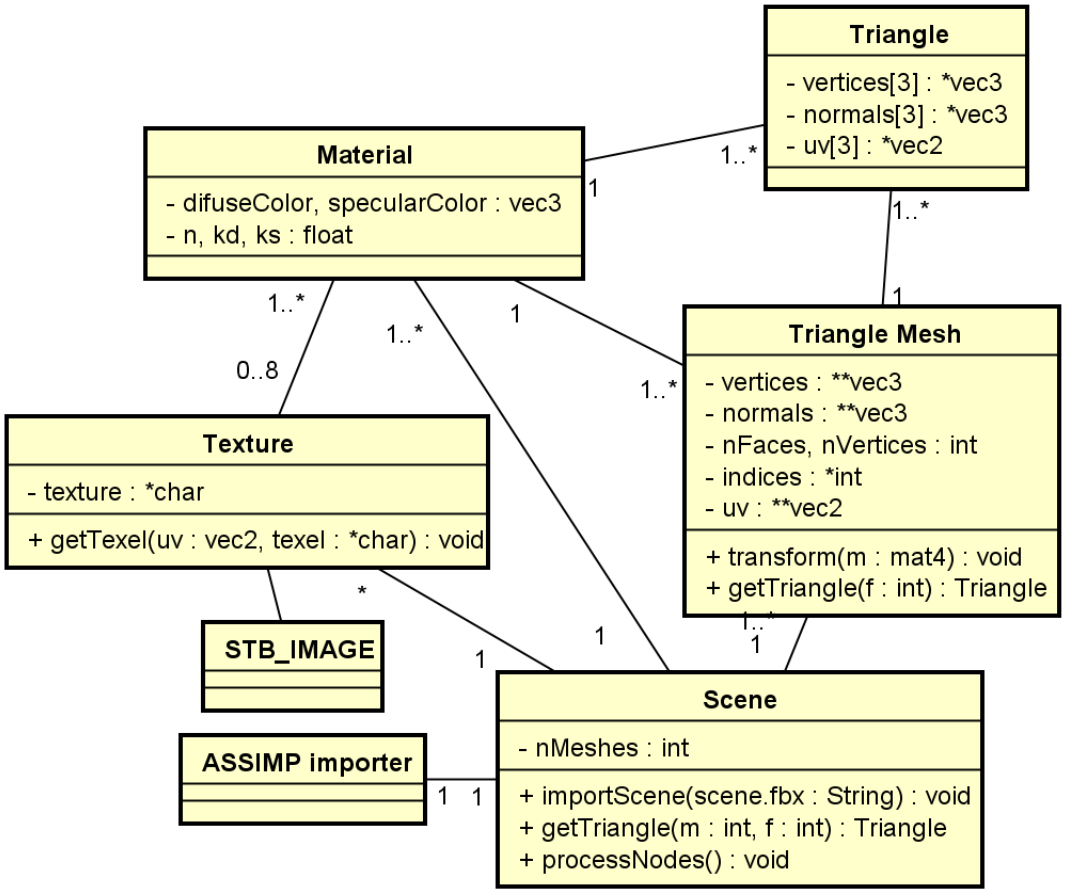

Figura 5.35 Simplificación del diseño en diagrama de clases.

Figura 5.36 Renderizado de dos cubos, uno con mapa de normales(derecha).

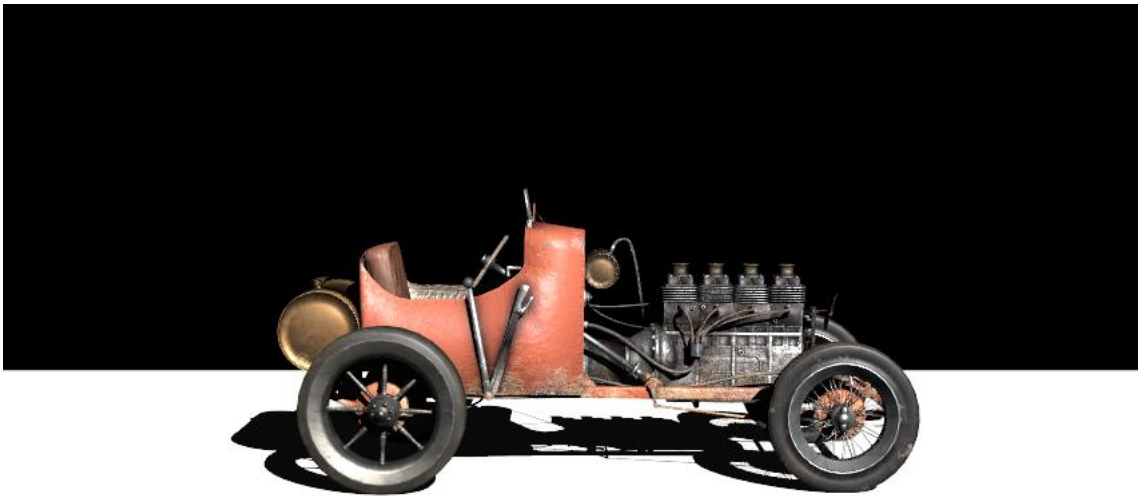

Figura 5.37 Renderizado coche de carreras vintage (con PBR) [81].

5.3.4 Reflexión y refracción

En un Whitted ray tracer, la simulación de reflexión y refracción es obligatoria para poder considerarse como tal. Estos procesos se usan para lograr una representación realista de la interacción de la luz con las superficies, permitiendo la creación de efectos como reflejos y distorsiones refractivas en materiales transparentes.

Para implementar la refracción y reflexión, se utilizó la ley de Snell, junto con las ecuaciones de Fresnel y el ángulo límite para la reflexión interna total. En una primera aproximación, se consideró que todos los materiales transparentes tenían un índice de refracción de 1.4 y que no existía dependencia de la frecuencia de la luz o la polaridad.

Usando la ley de Snell se tiene que la dirección del rayo reflejado es:

$$\vec{R} = 2 \cdot (\vec{N} \cdot \vec{I}) \cdot \vec{N} - \vec{I} \tag{5.76}$$

Donde $\vec{R}$ es la dirección del rayo reflejado, $\vec{I}$ es el vector del rayo incidente y $\vec{N}$el vector normal. Para el rayo refractado:

$$\vec{T} = \frac{n_1}{n_2} \cdot \vec{I} + \left(\frac{n_1}{n_2} \cdot cos(\theta_i) - cos(\theta_r)\right) \cdot \vec{N} \tag{5.77}$$

$$cos(\theta_r) = \sqrt{1 - \left(\frac{n_1}{n_2}\right)^2 \left(1 - \left(\vec{I} \cdot \vec{N}\right)^2\right)} \tag{5.78}$$

Donde $\vec{T}$es la dirección del rayo transmitido, θ_r es el ángulo de refracción y $\theta_i$es el ángulo de incidencia.

Para la ecuación de Fresnel, la parte perpendicular y paralela se promedian como si la luz no estuviera polarizada:

$$R(\theta) = \frac{1}{2}\left[\left(\frac{n_1 \cdot cos(\theta_r) - n_2 \cdot cos(\theta_i)}{n_2 \cdot cos(\theta_i) + n_1 \cdot cos(\theta_r)}\right)_{||}^2 + \left(\frac{n_1 \cdot cos(\theta_i) - n_2 \cdot cos(\theta_r)}{n_1 \cdot cos(\theta_i) + n_2 \cdot cos(\theta_r)}\right)_{\perp}^2\right] \tag{5.79}$$

$R(\theta)$ es la reflectancia y la transmitancia sería: $T(\theta) = 1 - R(\theta)$

Además, se optó por usar la aproximación Schlick para el Fresnel ya que requiere muchos menos cálculos y es una buena aproximación:

$$R(\theta) = R_0 + (1 - R_0) \cdot (1 - cos(\theta))^5 \tag{5.80}$$

Donde R_0 es la reflectancia cuando el ángulo de incidencia es 0.

$$R_0 = \left(\frac{n1 - n2}{n1 + n2}\right)^2 \tag{5.81}$$

La ecuación condicional del ángulo límite para la reflexión interna total cuando el rayo pasa de un material con un índice mayor a otro con un índice de refracción menor.

$$sen(\theta) \cdot \left(\frac{n_1}{n_2}\right) > 1 \tag{5.82}$$

Si esto sucede no hay rayo transmitido y solo lo hay reflejado.

Si un rayo primario chocaba con un objeto transparente, se emitían dos rayos secundarios: uno de reflexión y otro de transmisión. Para evitar que volvieran a chocar con el mismo objeto, se desviaba su origen mínimamente hacia las direcciones de ambos rayos. El proceso de emisión de rayos secundarios se realizó de forma iterativa en una primera instancia, y luego se convirtió en una especie de llamadas recursivas al método render. Los rayos secundarios pasaban por el mismo proceso de color que los rayos primarios y posteriormente proporcionaban color a su rayo padre según la reflectancia.

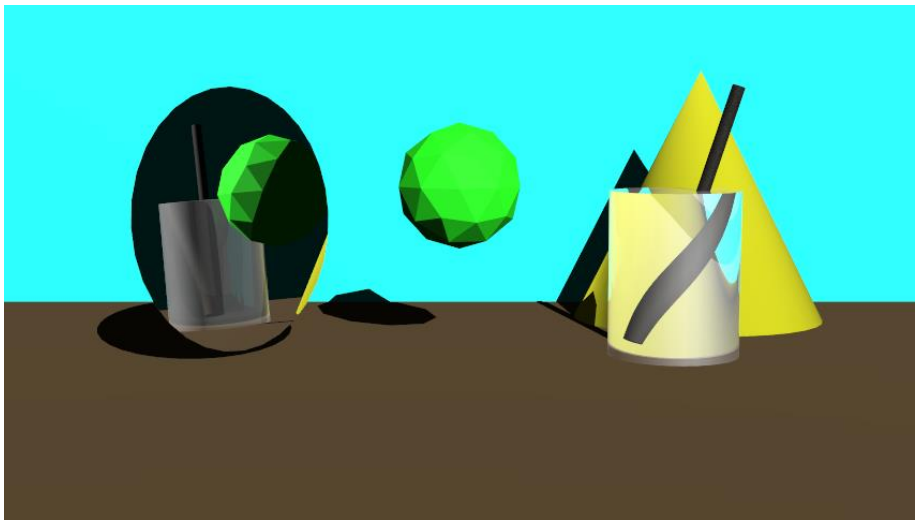

Figura 5.38 Renderizados reflexiones y refracciones.

Para evitar el desbordamiento de memoria con los reflejos, se procuró implementar una magnitud, profundidad máxima, que limitaba el número de procesos recursivos o iterativos generados. Además, a la clase Ray se le añadieron métodos y funciones para poder crear y emitir los rayos de reflexión y refracción.

Se añadieron 3 métodos para crear, calcular la posición y dirección de los rayos secundarios y la transmitancia según los algoritmos anteriores.

createTransmissionRay, createReflectionRay y fresnel.

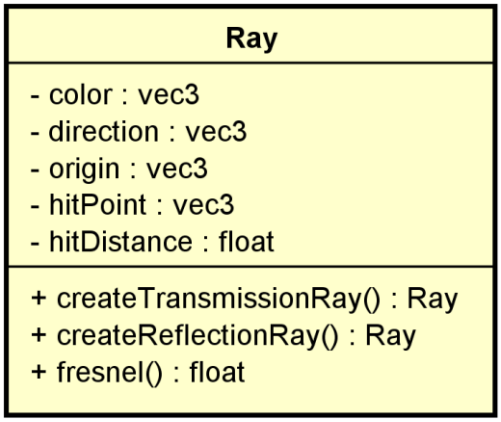

Figura 5.39 Simplificación del diseño en diagrama de clases

5.3.5 Implementación de PBR Cook-Torrance

Siguiendo con la iteración, se abandonó el sombreado de Blinn-Phong y se implementó técnicas de reflexión PBR de Cook-Torrance:

Recuérdese que la BRDF de Cook-Torrance tiene este aspecto:

$$f_r = k_d f_{ambert} + k_s f_{cook-torrance} \tag{5.83}$$

Donde la parte lambertiana es:

$$f_{ambert} = \frac{c}{\pi} \tag{5.84}$$

Donde c es el albedo o el color difuso del material y está divida entre pi por razones de conservación de la energía como se vio en el apartado 2.4.14 La parte especular tiene esta forma:

$$f_{cook-torrance} = \frac{F(\theta_r, \theta_h, F_0) \cdot G(\overrightarrow{\omega_i}, \overrightarrow{\omega_r}) \cdot D(\theta_h)}{4 \cdot \cos(\theta_i) \cdot \cos(\theta_r)} \tag{5.85}$$

Donde $\overrightarrow{\omega_i}$ es la dirección incidente, $\overrightarrow{\omega_r}$ dirección de reflejado, θ_h se refiere al ángulo que forma el vector halfway con el vector normal. También respeta la conservación de la energía.

Desglosándolo se puede dar el siguiente significado físico a las partes de la ecuación:

- La componente $F(\theta_r, \theta_h, F_0)$ es el término Fresnel y permite que ciertos efectos o propiedades varíen según la longitud de onda de la luz.
- $G(\omega_i, \omega_r)$ es la atenuación geométrica, el resultado del renderizado disminuye dependiendo de la cantidad de sombra o enmascaramiento que se produce en una superficie.
- $D(\theta_h)$ es la función de distribución y determina qué porcentaje de microfacetas están orientadas para reflejar en la dirección de la cámara.
- $\cos(\theta_i)$ cuánta de la superficie macroscópica es visible para la fuente de luz.
- $\cos(\theta_r)$ cuánta de la superficie macroscópica es visible para la cámara.
- $\overrightarrow{\omega_i}, \overrightarrow{\omega_r}$ son la dirección de incidencia y de salida. Y los ángulos mostrados son los ángulos que conforma cada dirección con la normal de la superficie.
- F_0 representa la reflectancia base de la superficie, cuando la normal y la dirección son paralelas, que se calcula utilizando los índices de refracción y se intenta que sea dependiente de la frecuencia mediante una interpolación sencilla. Haciendo que sea más dependiente del color del material cuánto más metálico es el material.

Para implementar el modelo hay que tener en cuenta lo siguiente:

La constante difusa, k_d, y la constante especular, k_s ahora son dependientes de la frecuencia de la luz. Para respetar la conservación de la energía.

$$1 = k_d + k_s \tag{5.86}$$

Y las constantes son:

$$k_s = F(\theta_r, \theta_h, F_0) \tag{5.87}$$

$$F_0 = (1 - metalness) \cdot color_{blanco} \cdot 0.4 + metalness \cdot color_{albedo} \tag{5.88}$$

Y k_d disminuye cuánto más metálico es el material, porque los materiales metálicos absorben en mayor medida la luz.

$$k_d *= (1 - metalness) \tag{5.89}$$

Donde metalness es la metalicidad del material, un escalar que va de 0 a 1. 1, completamente metálico y 0, completamente dieléctrico.

Se implementaron las mismas funciones utilizadas por el motor Unreal Engine 5 de Epic Games, que son Trowbridge-Reitz GGX para $D(\theta_h)$, la aproximación Fresnel-Schlick para $F(\theta_i)$ y Smith's Schlick-GGX para $G(\omega_i, \omega_r)$.

$$D_{GGXTR}(\theta_h) = \frac{\alpha^2}{\pi \cdot \left(\left(\vec{N} \cdot \vec{H}\right)^2 \cdot (\alpha^2 - 1) + 1\right)^2} \tag{5.90}$$

Donde α es la rugosidad de material y es otro escalar que va de 0 a 1, siendo 0 un material imposible completamente liso y 1 un material muy rugoso.

$$G_{SchlickGGX}(\vec{\omega}) = \frac{\vec{N} \cdot \vec{\omega}}{\left(\vec{N} \cdot \vec{\omega}\right) \cdot (1 - k) + k} \tag{5.91}$$

Donde k es un remapeo de α dependiente del tipo de luz:

$$k = \frac{(\alpha + 1)^2}{8} \tag{5.92}$$

Para aproximar de manera efectiva la geometría, es necesario tener en cuenta tanto la dirección de la vista (obstrucción de la geometría) como el vector de dirección de la luz (sombreado de la geometría). Ambos factores se pueden considerar utilizando el método de Smith.

$$G(\omega_i, \omega_r) = G_{SchlickGGX}(\omega_i) \cdot G_{SchlickGGX}(\omega_r) \tag{5.93}$$

Y el Fresnel:

$$F\left(\overrightarrow{\omega_r}, \vec{H}, F_0\right) = F_0 + (1 - F_0) \cdot (1 - (\vec{H} \cdot \overrightarrow{\omega_r}))^5 \tag{5.94}$$

Para luz ambiental se inventó un algoritmo considerando que esa luz no iba a ser reflejada en metales similar a la parte difusa, que era:

$$color \mathrel{+}= \; c \cdot ambientlight * (1 - metalness); \qquad (5.95)$$

Para incluir las ecuaciones anteriores se creó un módulo llamado PBR. En él se incluyeron las ecuaciones, creando sus métodos y funciones y, como el último modelo era el cubo, se puso a renderizar. La diferencia se pudo ver en la barra de metal y en un menor brillo especular en la madera que se puede apreciar en la Figura 5.40

También se añadieron los atributos metalicidad y rugosidad al struct material y la posibilidad de leer mapas de texturas relacionados con el PBR.

Cabe destacar que normalmente, cuando se codifica, los productos escalares están bajo la función max(0,producto escalar). Como sucedió con el modelo Phong, todos los productos escalares que tuvieran el vector $\vec{N}$ en él, se sustituyeron las funciones max por funciones abs, que requieren menor cómputo y permite que el choque sea independiente de la dirección del vector normal sin tener que invertir la normal.

Además de la Figura 5.40, se renderizaron una suerte de bolas, bolas PBR, que varían sus propiedades según se encuentran más a la izquierda o más hacia abajo. Nótese que las ecuaciones de Cook-Torrance contemplan la interacción en frecuencias de la luz con las superficies, siendo los brillos dieléctricos blancos y los brillos metálicos del color del metal. También se puede observar que cuánto más metálico menos irradiancia difusa se emite absorbiéndose casi toda la irradiancia incidente. Figura 5.41

Figura 5.40 Renderizados cubos, con PBR izquierda, sin Blinn-Phong derecha.

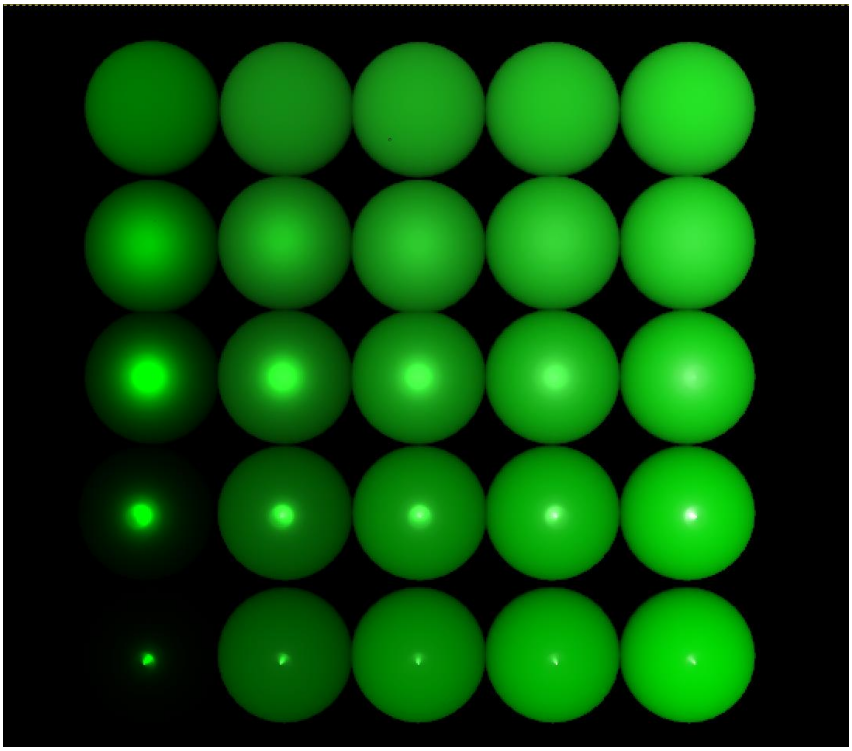

Figura 5.41 Renderizado de bolas PBR; más metálicas, izquierda, dieléctricas, derecha y más rugosas hacía arriba.

Corrección gamma

Además de PBR, se introdujo una sencilla ecuación de corrección gamma:

La Corrección gamma, (Gamma correction), es un ajuste que se realiza en imágenes para alinear cómo las pantallas muestran la luz con la forma en que el ojo humano percibe la luminosidad. Los ojos humanos no perciben el brillo de manera lineal; somos más sensibles a los cambios en las sombras que en las luces. La Corrección Gamma 2.2 corrige esta diferencia aplicando una curva que ajusta la luminosidad de la imagen, de modo que la luz se muestra de una manera que parece más natural y realista al ojo humano, compensando así la no linealidad tanto de las pantallas como de nuestra percepción visual.

Para aplicar la corrección gamma 2.2 en una imagen, cada valor de color se ajustó elevándolo a la potencia de 1/2.2. Es decir, si se tenía un valor de intensidad de un píxel, se corregía aplicando la fórmula: valor corregido = valor original$^{1/2.2}$. Esto reduce la intensidad de los píxeles en las áreas brillantes y realza los detalles en las sombras, haciendo que la imagen final se vea más natural y fiel a cómo el ojo humano percibiría la luz en la vida real. Los colores procedentes de mapas de texturas deben excluirse del proceso por cuestiones empíricas.

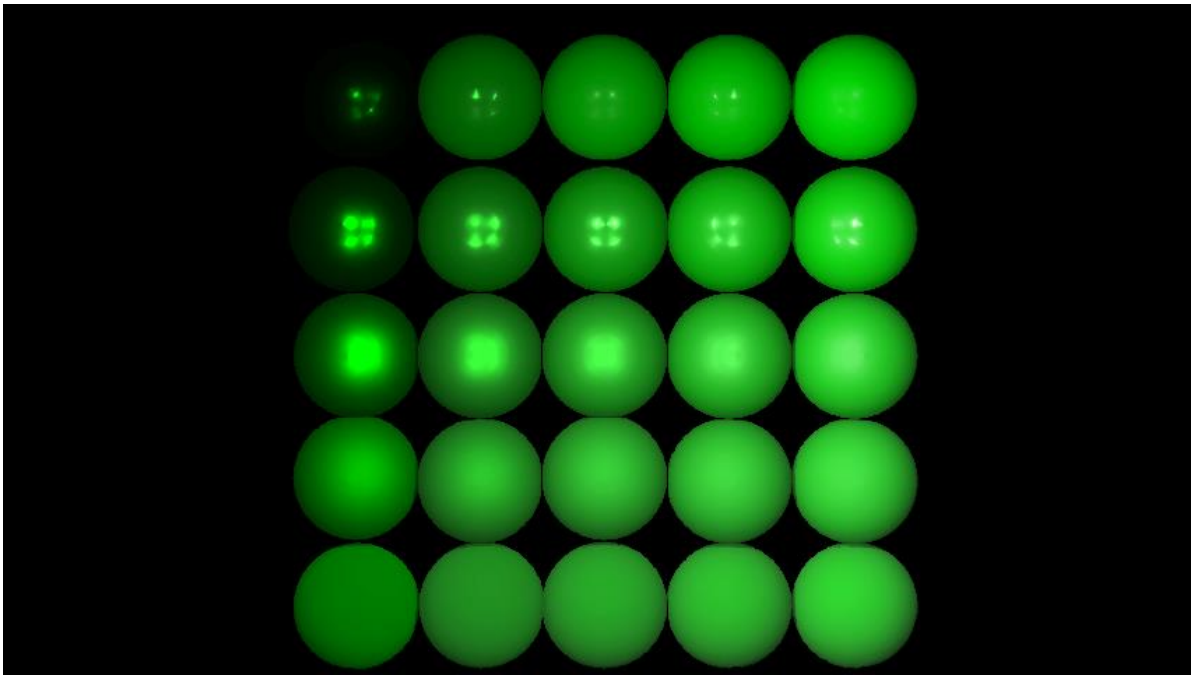

Figura 5.42 Renderizado de bolas PBR; más metálicas, izquierda, dieléctricas, derecha y más rugosas hacía abajo, con PBR y 4 puntos de luz.

Nótese que el modelo de las bolas de Figura 5.42 no es completamente esférico y por eso parece que tiene un pico en la parte más próxima a la cámara. En la Figura 5.43 se muestran esferas perfectas.

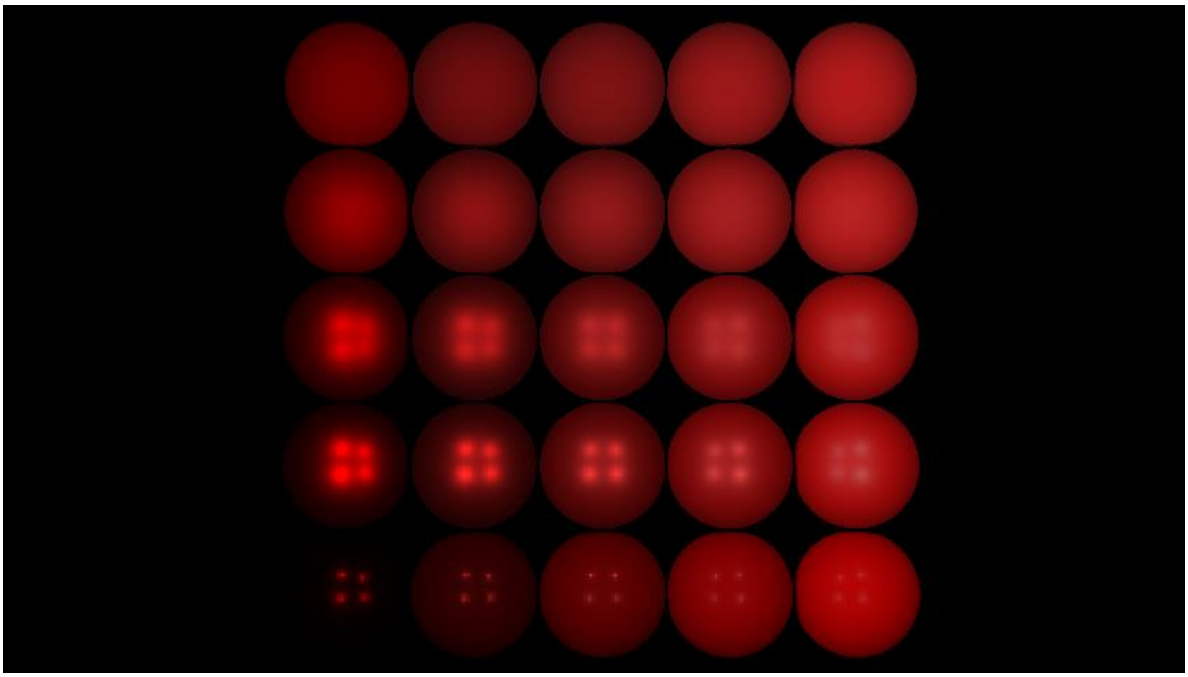

Figura 5.43 Renderizado de bolas PBR; más metálicas izquierda, más rugosas hacía arriba, con PBR y 4 puntos de luz.

5.3.6 Estructuras de aceleración

Debido a la gran ineficiencia de los cálculos realizados hasta el momento, donde se calculaban todos los choques de todos los rayos con todos los triángulos en $O(n^2)$, se requirió encontrar una forma de abordar este problema antes de pasar a la siguiente iteración, en la cual se emitirían aún más rayos.

Las estructuras de aceleración como las bounding boxes o hitboxes, o las hitboxes jerárquicas en árboles de hitboxes (BVH) o la grid de Fujimoto, existen para optimizar el proceso de ray tracing al reducir drásticamente la cantidad de cálculos necesarios para determinar las intersecciones de rayos con los objetos en una escena. Estas estructuras organizan el espacio y los objetos en volúmenes más simples, permitiendo descartar rápidamente grandes áreas donde no ocurren intersecciones, y concentrar el poder de cómputo solo en las áreas relevantes. Esto mejora la eficiencia notablemente.

Las hitboxes son volúmenes simplificados que rodean una malla. En lugar de calcular las colisiones con todos los triángulos de la malla, primero se calcula el choque con la hitbox y si este sucede se da paso a el cálculo de los choques de los triángulos de la malla.

Generar una hitbox es sencillo, hallando el mínimo y máximo de las coordenadas de todos los vértices de la malla. La intersección rayo-hitbox se explicó en el apartado 4.6 del capítulo anterior.

Se añadieron hitBoxes a las clases Triangulo, TriangleMesh y Scene y métodos para calcular las intersecciones.

Para el renderizado del cubo de madera que consta de tres o cuatro mallas se redujo el tiempo en 1/3 aproximadamente Figura 5.45.

Las hitboxes se podrían complicar usando un sistema jerárquico en forma de árbol de bounding boxes BVH, etc, como vimos en el capítulo 2. En el programa, sin embargo, se implementó otro método, la Fujimoto Grid.

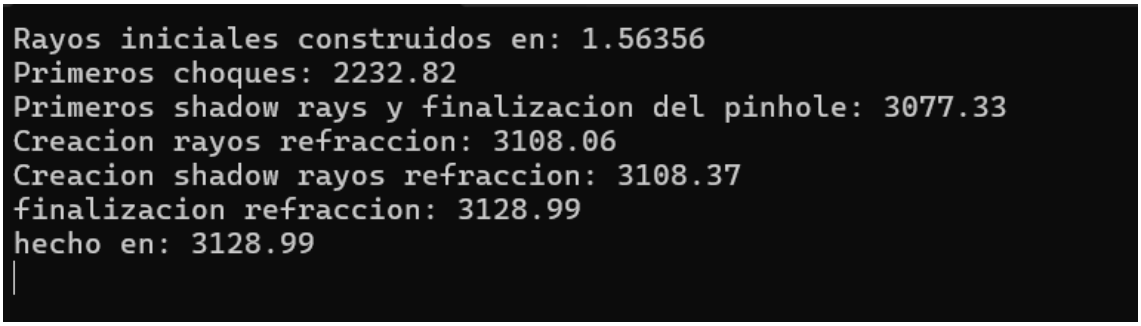

Figura 5.44 Imagen consola 1: Con una resolución 1600x1600 y sobremuestreo 30 y sin la estructura de aceleración inicial.

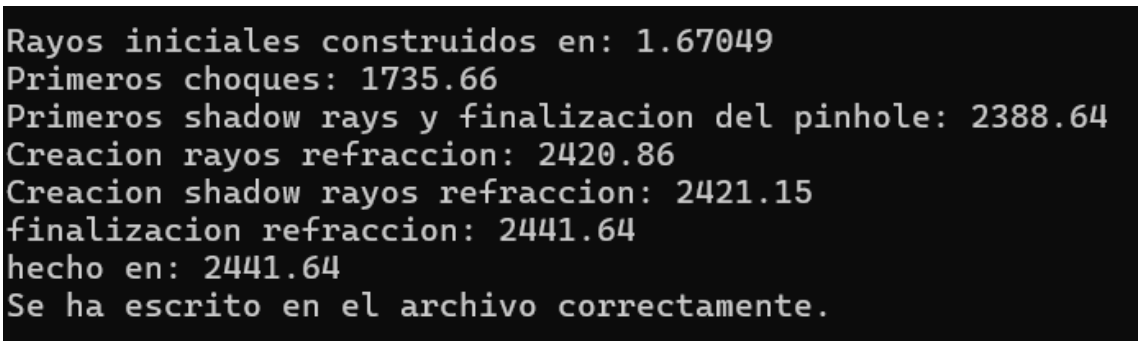

Figura 5.45 Imagen consola 2: Con una resolución 1600x1600 y sobremuestreo 30 y con la estructura de aceleración inicial.

La grid de Fujimoto es un tipo de grid que divide el espacio tridimensional en una cuadrícula regular de celdas, cubos o paralelepípedos, donde cada celda contiene referencias a los objetos que intersecan con ella, triángulos, esferas, etc. Cuando un rayo atraviesa la escena, solo se verifican las intersecciones con los objetos dentro de las celdas por las que pasa el rayo, lo que reduce significativamente la cantidad de cálculos. Este método fue propuesto por *Akira Fujimoto* en el artículo titulado *"Accelerated Ray Tracing*", publicado en 1986 en el *Proceedings of the IEEE Computer Society's Symposium on Computer Graphics and Image Processing.*

Se implementó el algoritmo de la Fujimoto Grid en una clase llamada Grid, comenzando con la creación de la cuadrícula para luego llenarla con triángulos.

El algoritmo determina en qué celdas de la cuadrícula se encuentra cada triángulo. Durante el cálculo de intersecciones de los rayos, se emplea otro algoritmo que recorre las celdas de la cuadrícula, 3D-Digital Differential Analyser. Primero, se identifica la celda en la que el rayo interseca, ya sea externa o interna, y luego se verifica si hay colisiones con los triángulos dentro de esa celda. Si no se detectan colisiones, se calcula hacia qué celda se moverá el rayo desde la celda actual. Este proceso continúa hasta que el rayo interseca un triángulo o sale de la cuadrícula.

La Clase Grid se convirtió en una interfaz intermediaria entre el render y la escena. Ya no se le pedían los triángulos a la escena, sino que el algoritmo 3D-Digital Differential Analyser calculaba la intersección con los triángulos haciendo uso del algoritmo Möller-Trumbore.

```
Fujimoto grid hecho en: 0.0108707
Rayos iniciales construidos en: 1.57407
Primeros choques: 172.444
Primeros shadow rays y finalizacion del pinhole: 208.025
Creacion rayos refraccion: 238.266
Creacion shadow rayos refraccion: 238.426
finalizacion refraccion: 258.259
hecho en: 258.26
Se ha escrito en el archivo correctamente.
```

Figura 5.46 Imagen consola 2: Con una resolución 1600x1600 y sobremuestreo 30 y con la malla de fujimoto.

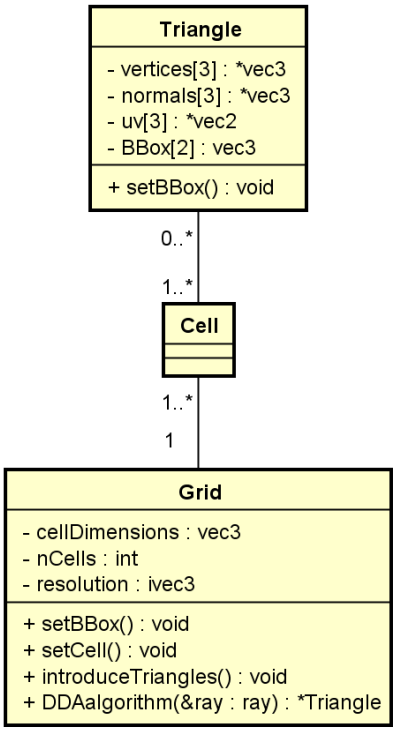

Figura 5.47 Simplificación del diseño en diagrama de clases.

Akira Fujimoto Grid redujo en 10 el tiempo de computación del cubo. El problema de la red de Fujimoto se encuentra cuando un paralelepípedo contiene un objeto muy pequeño con muchas caras; en ese caso, la técnica deja de ser efectiva. Sin embargo, se pueden combinar varias técnicas a la vez o incluso utilizar subredes de Fujimoto u

octrees. Esta obra de arte podría ser análoga de cómo se almacenan objetos dentro de la red, Figura 5.48.

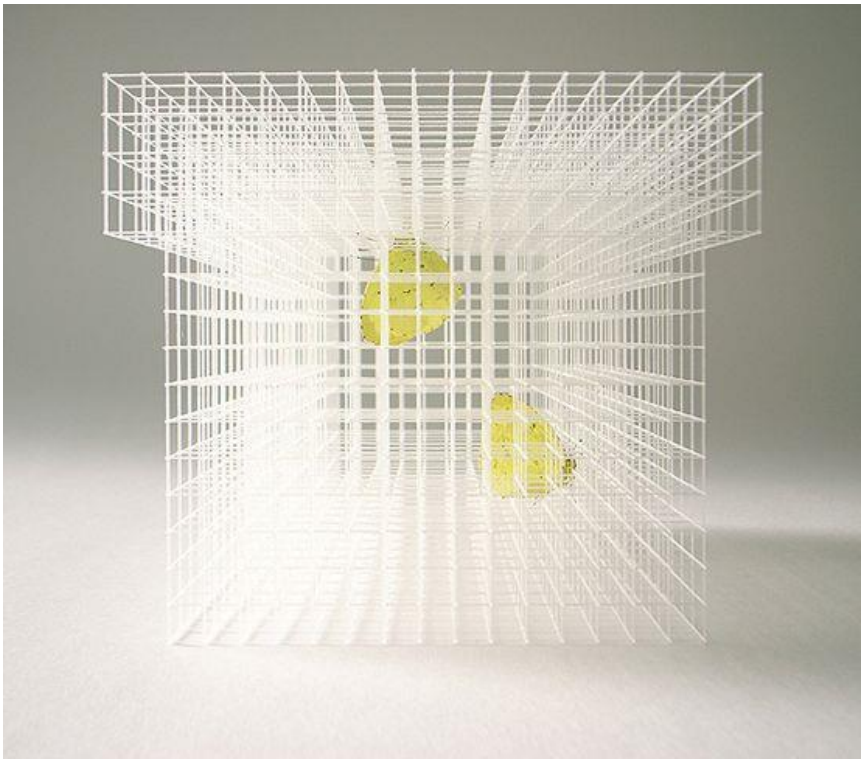

Figura 5.48 'Blossom' por Ryuji Nakamura [82].

Además de integrar la Grid de Fujimoto, se aprovechó el cálculo en paralelo mediante OpenMP. Esta historia de usuario se priorizó y se implementó en el primer sprint, dado que la independencia entre rayos, figuras, y sus intersecciones permitía una ejecución en paralelo.

OpenMP (Open Multi-Processing) es una API que permite la programación paralela en sistemas multiprocesador de memoria compartida.

OpenMP ofrece directivas sencillas que se integran en lenguajes como C, C++ y Fortran, facilitando a los desarrolladores la paralelización de secciones de código, como bucles, sin necesidad de reescribir grandes partes del programa. Es ampliamente utilizado en aplicaciones científicas, simulaciones y gráficos por computadora para mejorar el rendimiento y la eficiencia de las operaciones complejas.

Un tutorial conciso es el siguiente: https://openmp.paralelo.dev/

5.3.7 Conclusiones de la segunda iteración

Con esto, se concluyó la segunda iteración, culminando en la creación de un Whitted Ray Tracer plenamente funcional. Mencionar que el proyecto sufrió más retrasos aparte del relacionado con las texturas, no cumpliendo con el calendario pre establecido.

En este sprint, se construyó un Whitted Ray Tracer. Se comenzó explorando técnicas de sobremuestreo para mejorar la calidad de la imagen al reducir el aliasing, y luego se avanzó hacia la aplicación de texturas que añadieron detalle y realismo a las superficies.

También se cubrió la algoritmia de la reflexión y la refracción, fundamentales para simular cómo la luz interactúa con materiales refractivos. Además, se introdujo el modelo de Renderizado Físicamente Basado (PBR) y el Cook-Torrance BRDF, una técnica avanzada que simula de manera más precisa cómo la luz se refleja en diferentes tipos de superficies.

Asimismo, se implementó la corrección gamma, que simula la adaptabilidad humana a las sombras y luces. Finalmente, la iteración concluyó con un estudio y aplicación de las estructuras de aceleración, como la rejilla de Fujimoto, las hitboxes y el uso de OpenMP, que optimizan el rendimiento del ray tracing al reducir el tiempo de cálculo necesario para determinar la intersección de rayos con objetos en la escena.

5.4 Tercera iteración

5.4.1 Introducción de la tercera iteración

En este tercero y último sprint de 4 semanas se creó un *Backward Path Tracer*:

Un backward path tracer es un algoritmo de trazado de rayos que simula de manera realista la iluminación global. Utiliza el método de Montecarlo para estimar la contribución de la luz indirecta y las sombras suaves.

La iluminación indirecta es la luz que no llega directamente de una fuente luminosa a una superficie, sino que ha sido reflejada por otras superficies antes de alcanzarla. En un entorno realista, la luz no sólo proviene directamente de fuentes de luz como el sol o una lámpara (iluminación directa), sino que también rebota en paredes, techos, muebles, y otros objetos, produciendo sangrados de luz y contribuyendo a la luz global en la escena.

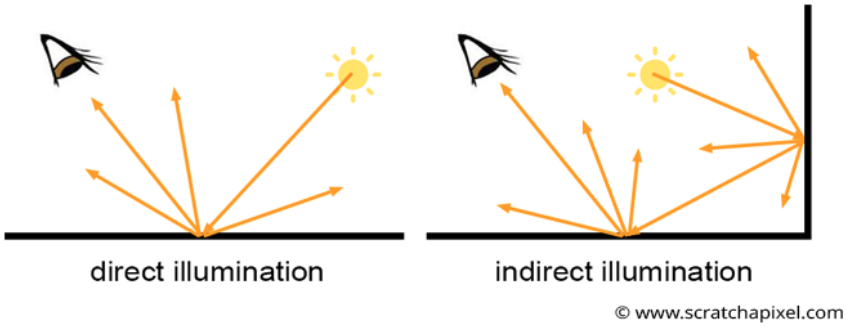

Figura 5.49 Iluminación directa vs indirecta [83].

Para simular la luz indirecta, por cada choque se emiten nuevamente numerosos rayos aleatorios en todas direcciones a través de una semiesfera sobre el plano. A su vez, en los puntos donde estos rayos intersecan, se lanzan más rayos en todas direcciones, continuando este proceso hasta alcanzar una profundidad de trazado máxima o una fuente de luz de área. Las contribuciones de estos rayos secundarios a la irradiancia de sus rayos padres se discuten a continuación.

5.4.2 Iluminación global

Según Kayiya, si se desea aproximar la ecuación de la irradiancia se usa el método de Montecarlo. Usando el método Montecarlo y el método de inversión se tiene la siguiente función de irradiancia, por pasos:

Para una función de probabilidad que se distribuye uniformemente por la semiesfera p=C para Ω y p=0 fuera de Ω:

$$1 = \int_{\mathbf{0}}^{\mathbf{2\pi}} \int_{\mathbf{0}}^{\mathbf{\pi/2}} C \cdot sen\varphi \cdot d\varphi \cdot d\theta \tag{5.96}$$

$$1 = 2 \cdot \pi \cdot C \tag{5.97}$$

$$C = \frac{1}{2\pi} \tag{5.98}$$

Luego tenemos una pdf, función de probabilidad:

$$pdf = \frac{1}{2\pi} \tag{5.99}$$

Por lo tanto:

$$Irradiancia(\vec{p}, \overrightarrow{\omega_o}) \approx \frac{2\pi}{N} \sum_{i=0}^{N-1} Radiancia(\vec{p}, \overrightarrow{\omega_o}, \overrightarrow{\omega_i}) \tag{5.100}$$

Donde $\overrightarrow{\omega_i}$es una dirección incidente aleatoria en el hemisferio superior del punto, $\vec{p}$, y $\overrightarrow{\omega_o}$ la dirección saliente. N es el número de rayos secundarios generados y $\vec{p}$ el punto de choque. A continuación, se discute el algoritmo para generar rayos secundarios.

Para generar un rayo secundario aleatorio apuntando a la superficie semiesférica se usan coordenadas esféricas. Donde:

$$x = sen(\varphi) \cdot \cos(\theta) \tag{5.101}$$

$$y = sen(\varphi) \cdot \text{sen}(\theta) \tag{5.102}$$

$$z = cos(\varphi) \tag{5.103}$$

Luego se generan dos números aleatorios en un intervalo de valores de 0 a 1; aleatorio1, aleatorio2. Para los ángulos theta y phi y la probabilidad constante en el hemisferio superior d la esfera se tiene:

$$p(\theta, \varphi) \cdot d\varphi \cdot d\theta = p(\omega) \cdot d\omega \tag{5.104}$$

$$p(\theta, \varphi) \cdot d\varphi \cdot d\theta = p(\omega) \cdot \sin(\varphi') \cdot d\varphi \cdot d\theta \tag{5.105}$$

$$p(\theta, \varphi) = \frac{\sin(\varphi)}{2\pi} \tag{5.106}$$

$$p(\varphi) = \int_0^{2\pi} p(\theta, \varphi) \cdot d\theta = \sin(\varphi) \tag{5.107}$$

Según Bayes al ser independientes las probabilidades:

$$p(\theta) = \frac{p(\theta, \varphi)}{p(\varphi)} = \frac{1}{2\pi} \tag{5.108}$$

$$p(\varphi) = \int_0^{\varphi} p(\varphi') \cdot d\varphi' = \int_0^{\varphi} \sin(\varphi') \cdot d\varphi' = 1 - \cos(\varphi) = aleatorio_2 \tag{5.109}$$

$$\cos(\varphi) = 1 - aleatorio_2 \tag{5.110}$$

Si el coseno va de [0,1] el ángulo va de [0, $\frac{\pi}{2}$]. Aunque se podría seguir desarrollando z=cos(φ)=$1 - aleatorio_2$, pero se va a dejar aquí $\varphi = {}^{\pi}/_{2} \cdot aleatorio_2$.

$$p(\theta) = \int_0^{\theta} p(\theta') \cdot d\theta' = \int_0^{\theta} \frac{1}{2\pi} \cdot d\theta' = \frac{\theta}{2\pi} = aleatorio_1 \tag{5.111}$$

$$\theta = 2\pi \cdot aleatorio_1 \tag{5.112}$$

$$\varphi = {}^{\pi}/_{2} \cdot aleatorio_2 \tag{5.113}$$

Y para pasar del espacio rayo al espacio mundo la dirección $\vec{d}$:

$$\vec{d} = (\overrightarrow{bitangente}, \overrightarrow{tangente}, \overrightarrow{normal})_{3x3} \cdot (x, y, z) \tag{5.114}$$

Donde la tangente, bitangente y normal son propias de la superficie del punto p.

Para la emisión de rayos secundarios, se añadió un método a la clase Ray llamado randomRay con este algoritmo dentro. Los rayos emitidos pasaban por la función de renderizado de forma recursiva, adquiriendo color y convirtiéndose en rayos de luz con radiancia propia, que transmitían a los rayos progenitores a través de la superficie compartida, utilizando la función BRDF.

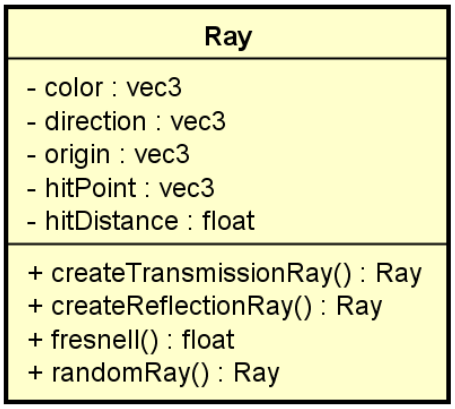

Figura 5.50 Simplificación del diseño en diagrama de clases.

Luces de área

Las luces de área son objetos con luz propia, objetos emisivos de luz. Si un rayo choca con un objeto luminoso se supone que es un rayo emitido por el objeto luminoso. El método de Montecarlo trata de forma estadística la atenuación de las luces de área.

Para definir una luz de área, bastó con agregar un color emisivo al material de su malla o hacer lo mismo utilizando un mapa de textura que contemplaba la emisión de luz. Figura 5.51.

Considerando la emisión de 50 rayos secundarios por choque, una profundidad máxima de 3, y sobremuestreo de 3 rayos por pixel, se pudieron obtener imágenes como esta, Figura 5.52.

Las sombras suaves son consecuencia del método de Montecarlo y el gradiente de la iluminación.

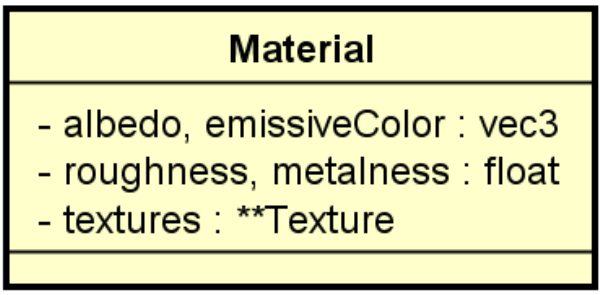

Figura 5.51 Simplificación del diseño en diagrama de clases

Figura 5.52 Renderizado de una habitación [84].

Cornell box

Hasta ahora se han mostrado imágenes de una forma poco rigurosa, en computación grafica se usa la Cornell Box, vista en el apartado 2.17, para poder cotejar los resultados con la realidad:

Se desarrolló una Cornell Box propia creando una escena 3D y, con 50 rebotes secundarios, sin sobremuestreo y una profundidad máxima de 2, se obtenía en menos de un minuto la imagen mostrada en la Figura 5.53.

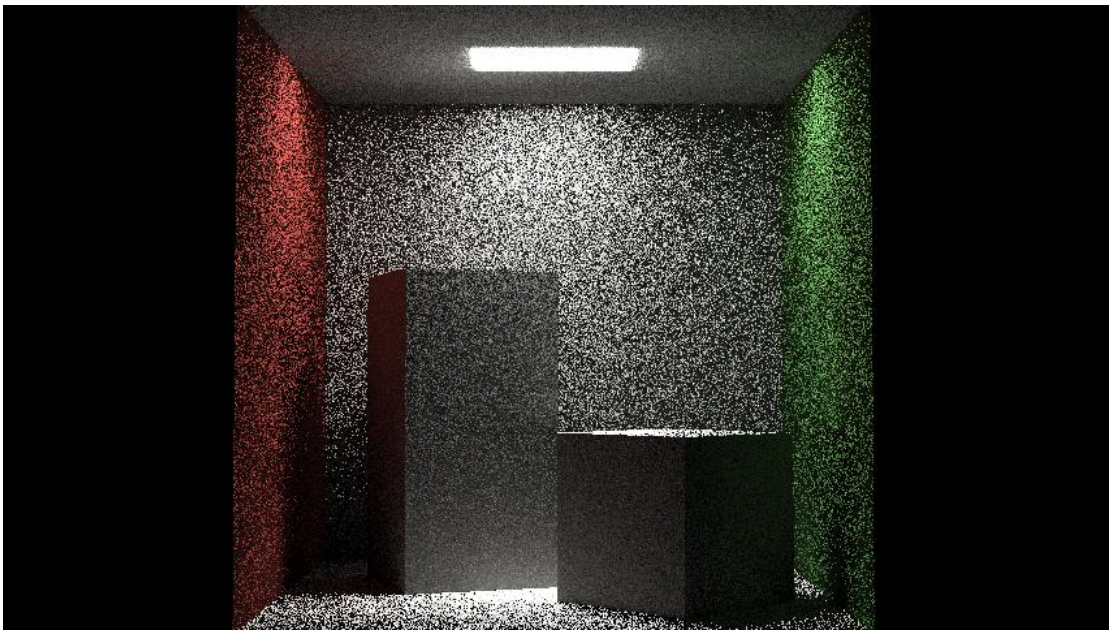

Figura 5.53 Cornell box.

En la Figura 5.53 el ruido es consecuencia de la varianza en el método de Montecarlo, se eligió sobremuestreo 1 para que el ruido fuera patente.

Y con esto se consiguió la implementación de un *Backward Path Tracer* funcional, a partir de aquí se añadieron funcionalidades para eliminar la varianza y reducir el tiempo de renderizado como las técnicas que se recogen en el *Multiple Importance Sampling* (MIS) y el *Russian Roulette Path Terminator*.

5.4.3 Multiple importance sampling

El *Multiple Importance Sampling* (MIS) es un conjunto de técnicas utilizado para reducir el ruido y la varianza. Combina múltiples métodos de muestreo, ponderando sus contribuciones de manera adaptativa. Al equilibrar las contribuciones de los diferentes muestreos, MIS minimiza la probabilidad de generar muestras irrelevantes o ineficientes, lo que resulta en una reducción significativa del ruido y una imagen más limpia y precisa con menos muestras. En este apartado se aúnan técnicas como la estratificación de muestras, el *importance sampling* y el *next estimation event*.

Estratificación de muestras

La estratificación de muestras es una técnica que busca generar un conjunto de puntos pseudoaleatorios distribuidos uniformemente en un espacio, de tal forma que no haya puntos demasiado cercanos entre sí ni demasiado separados. De esta forma se reduce la varianza:

$$N = \sum_{i=1}^{m} N_i \tag{5.115}$$

$$F_N = \frac{1}{N} \sum_{i=1}^{m} N_i \cdot F_i \tag{5.116}$$

$$(F_N) = \frac{1}{N^2} \sum_{i=1}^{m} N_i \cdot V(F_i) \tag{5.117}$$

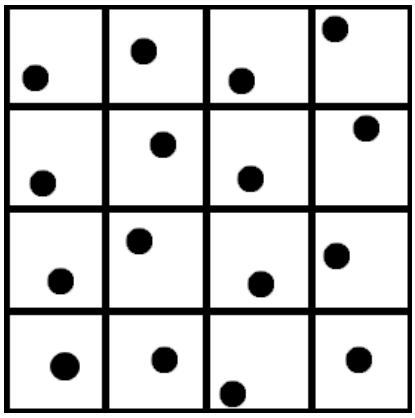

Figura 5.54 Conjunto de muestras estratificadas.

Se probaron numerosas formas de estratificar muestras aleatorias, como el uso de puntos Poisson, la creación de un jitter sencillo o la incorporación de un generador de la secuencia de Halton. En algunos casos, se producían sesgos indeseados debido a la falta de suficiente aleatoriedad. En otros casos, como con Halton o el jitter propio estratificando los puntos, similar a cómo se emitían los rayos primarios, no se observaron mejoras apreciables en la varianza, pero sí un aumento significativo en el cálculo computacional, lo que retrasaba la formación de imágenes. Por estos dos motivos, a pesar de estar estadísticamente probado, al no apreciarse ninguna mejora aparente, se optó por no implementar, por ahora, la estratificación de muestras al generar rayos secundarios.

Importance Sampling (IS)

Tras probar la estratificación de muestras se implementó el *importance sampling*:

El *Importance Sampling* (IS) es una técnica para reducir la varianza, enfocando las muestras en las direcciones que más contribuyen a la BRDF. En el caso de materiales difusos, se emplea la función de probabilidad del coseno del ángulo del rayo incidente para muestrear los rayos secundarios, ya que la irradiancia de una superficie difusa es mayor en ángulos cercanos a la normal debido al coseno de Lambert. Para materiales especulares, se usa el vector de reflexión, convirtiéndolo en la nueva normal de los rayos secundarios para orientar los rayos hacia el lóbulo especular, maximizando la eficiencia del muestreo en las direcciones donde la reflectancia especular es más significativa. Como se puede observar en las imágenes el ruido es reducido significativamente.

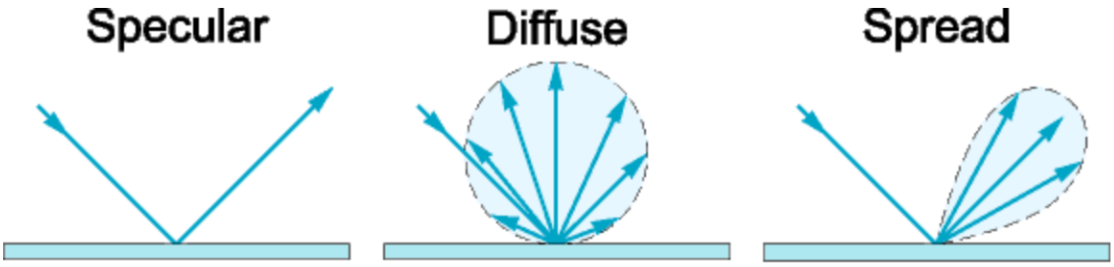

Figura 5.55 Distribuciones especulares y difusas **[86]**.

Importance sampling difuso

Primero se implementó el importance sampling difuso y la algoritmia para el cálculo de la irradiancia y el muestreo de los rayos secundarios:

Para una función de probabilidad p que se distribuye según el coseno por la semiesfera, p=C·cosφ.

$$1 = \int_0^{2\pi} \int_0^{\pi/2} C \cdot \cos\varphi \cdot sen\varphi \cdot d\varphi d\theta \tag{5.118}$$

$$1 = C \cdot 2\pi \cdot \left[\frac{-cos^2\varphi}{2}\right]_0^{\pi/2} \tag{5.119}$$

$$1 = C \cdot \pi \tag{5.120}$$

$$C = \frac{1}{\pi} \tag{5.121}$$

Luego la pdf es:

$$pdf = \frac{\cos\varphi}{\pi} \tag{5.122}$$

Montecarlo quedaría:

$$Irradiancia(\vec{p}, \overrightarrow{\omega_o}) \approx \frac{\pi}{N} \sum_{i=0}^{N-1} \frac{Radiancia(\vec{p}, \overrightarrow{\omega_o}, \overrightarrow{\omega_i})}{\cos\varphi_{i,N}} \tag{5.123}$$

Donde el coseno de cosφ es el coseno de Lambert y tanto él como pi se pueden anular con sus homólogos en la ecuación de la radiancia. A continuación, se discute el algoritmo para generar rayos secundarios.

Para generar un rayo secundario aleatorio apuntando a la superficie semiesférica se usan coordenadas esféricas. Donde según el método de la inversión:

Para que haya distribución del coseno:

$$aleatorio_2 = \int_{\mathbf{0}}^{\mathbf{2\pi}} \int_{\mathbf{0}}^{\boldsymbol{\varphi}} \frac{cos\varphi'}{\pi} \cdot sen\varphi' \cdot d\varphi' \cdot d\theta = 1 - cos^2\varphi \tag{5.124}$$

Se despeja:

$$cos\ \varphi = \sqrt{1 - aleatorio_2} \tag{5.125}$$

$$z = cos\ \varphi = \sqrt{1 - aleatorio_2} \tag{5.126}$$

$$x = \sqrt{1 - z^2} \cdot \cos(\theta) = \sqrt{aleatorio_2} \cdot \cos(\theta) \tag{5.127}$$

$$y = \sqrt{1 - z^2} \cdot \text{sen}(\theta) = \sqrt{aleatorio_2} \cdot \text{sen}(\theta) \tag{5.128}$$

En resumidas cuentas:

$$x = \sqrt{aleatorio_2} \cdot \cos(\theta) \tag{5.129}$$

$$y = \sqrt{aleatorio_2} \cdot \text{sen}(\theta) \tag{5.130}$$

$$z = \sqrt{1 - aleatorio_2} \tag{5.131}$$

Para el ángulo theta se sigue teniendo:

$$aleatorio_1 = \int_0^{\theta} \int_0^{\pi/2} \frac{cos\varphi}{\pi} \cdot sen\varphi \cdot d\varphi \cdot d\theta' = \left[\frac{-cos^2\varphi}{2\pi}\right]_0^{\pi/2} = \frac{\theta}{2\pi} \tag{5.132}$$

$$\theta = 2\pi \cdot aleatorio_1 \tag{5.133}$$

Y para pasar del espacio rayo al espacio mundo la dirección $\vec{d}$:

$$\vec{d} = (\overrightarrow{bitangente}, \overrightarrow{tangente}, \overrightarrow{normal})_{3x3} \cdot (x, y, z) \tag{5.134}$$

Donde la tangente, bitangente y normal son propias de la superficie del punto $\vec{p}$. Se aplicó el algoritmo en el método randomRay. Al crear los rayos secundarios, se desvió el origen del rayo mínimamente para evitar que volviera a chocar con el punto de partida. Se pudo observar una mejoría notable al añadir la función de probabilidad coseno a la generación de rayos secundarios.

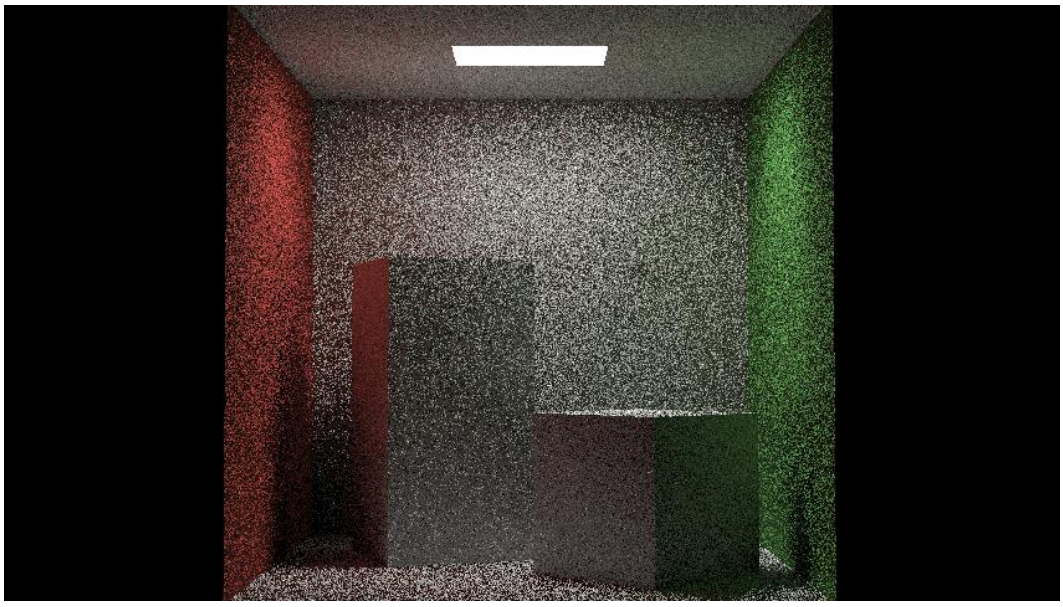

Figura 5.56 Importance sampling coseno.

Se puede notar que la pdf es similar a la BRDF para la parte difusa y que será similar si se amplía al caso especular.

Importance sampling especular

Tras implementar el importance sampling difuso se integró el imporance sampling *especular*:

En el caso especular se usa una pdf similar a la BDFR de Cook-Torrance, pero apuntando al lóbulo especular, al vector de *Reflexión*.

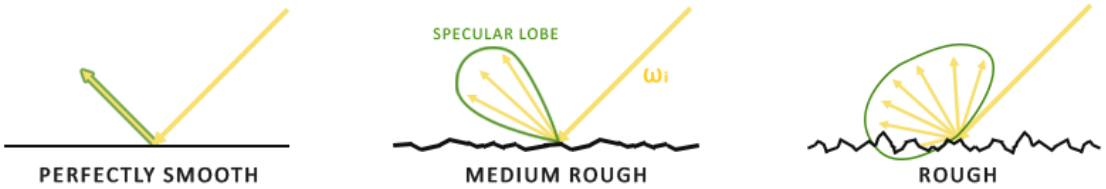

Figura 5.57 Distribución especular [14]

$$pdf = \frac{\cos(\theta_r)}{\pi} + \frac{\cos(\theta_h) \cdot D(\theta_h)}{4 \cdot \cos(\theta_r)} \tag{5.135}$$

Donde los cosenos son los cosenos de las direcciones de reflexión y del vector halfway con la normal. Además, la pdf se interpola según sea la suavidad, t, del material con un mínimo de 0.25, para poder simular brillos especulares en materiales rugosos.

$$pdf = (t-1) \cdot \frac{\cos(\theta_r)}{\pi} + t \cdot \frac{\cos(\theta_h) \cdot D(\theta_h)}{4 \cdot \cos(\theta_r)} \tag{5.136}$$

Para la irradiancia del rayo padre se tiene la siguiente función de irradiancia:

$$Irradiancia(\vec{p}, \overrightarrow{\omega_o}) \approx \frac{1}{N} \sum_{i=0}^{N-1} \frac{Radiancia(\vec{p}, \overrightarrow{\omega_o}, \overrightarrow{\omega_i})}{pdf(t, \theta_r, \theta_h)}$$

Para completar el algoritmo se decide si el material es especular o difuso de forma aleatoria con una probabilidad p<t para ser especular p>t para que el material sea difuso. Donde p es un número coma flotante aleatorio de 0 a 1. Usándose el *Importance Sampling* especular o difuso dependiendo del resultado.

Para generar un rayo secundario aleatorio apuntando al vector $\vec{R}$, de reflexión, se usan coordenadas esféricas. Donde:

$$z = cos(\varphi) \tag{5.137}$$

$$y = sen(\varphi) \cdot \text{sen}(\theta) \tag{5.138}$$

$$x = sen(\varphi) \cdot \cos(\theta) \tag{5.139}$$

Luego se generan dos números aleatorios en un intervalo de valores de 0 a 1; aleatorio1, aleatorio2. Para los ángulos theta y phi se tiene:

$$\theta = 2\pi \cdot aleatorio_1 \tag{5.140}$$

$$\varphi = \tan^{-1}\left[t \cdot \frac{\sqrt{aleatorio_2}}{\sqrt{1 - aleatorio_2}}\right] \tag{5.141}$$

Y para pasar del espacio rayo al espacio mundo la dirección $\vec{d}$:

$$\vec{d} = (\overrightarrow{bitangente_R}, \overrightarrow{tangente_R}, \vec{R})_{3x3} \cdot (x, y, z) \tag{5.142}$$

Donde la tangente de $\vec{R}$, bitangente de $\vec{R}$ son perpendiculares a $\vec{R}$ y entre sí.

Tanto para el caso difuso como para el especular, se amplió el método *randomRay*, incorporando esta matemática. Los resultados mostraron una mejora en la reducción de ruido, sin incrementar el tiempo de renderizado. En el importance sampling especular, la calidad de los brillos especulares y la reducción del ruido en ellos se

pueden apreciar al comparar el uso exclusivo de IS difuso con el IS especular en una bola metálica. Estas diferencias se observan tanto entre las Figura 5.58 y 5.59 como entre las renderizaciones Figura 5.60 y Figura 5.61.

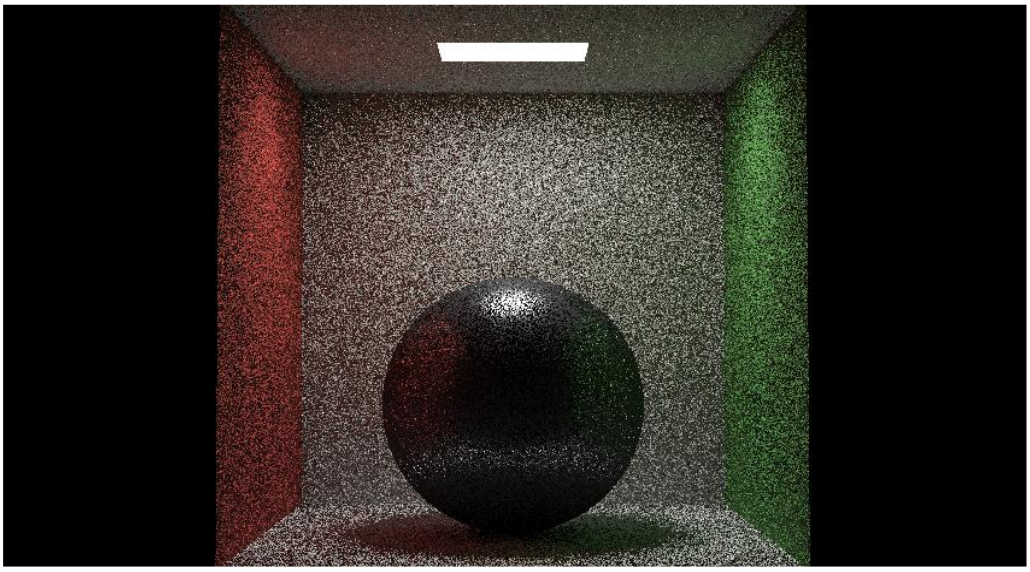

Figura 5.58 Bola metálica con importance sampling coseno.

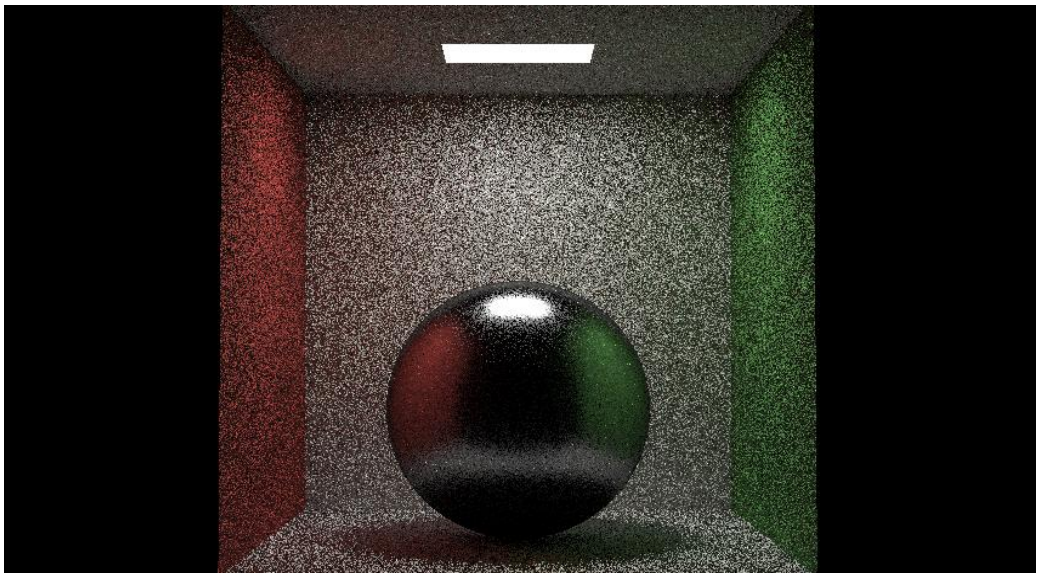

Figura 5.59 Bola metálica con importance sampling especular.

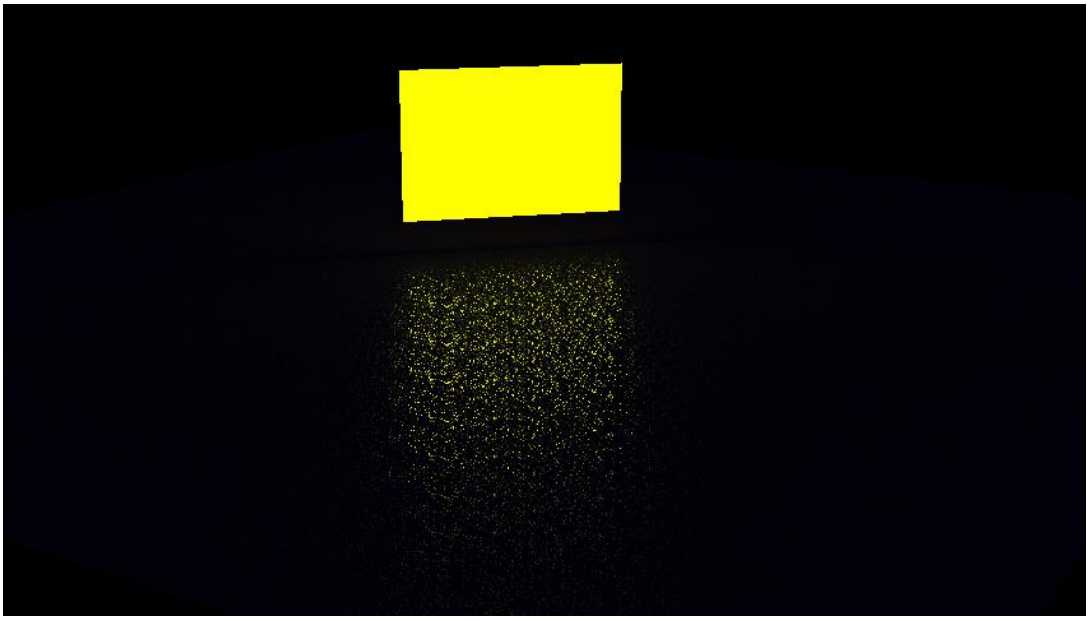

Figura 5.60 Reflejo en superficie especular metálica con importance sampling difuso.

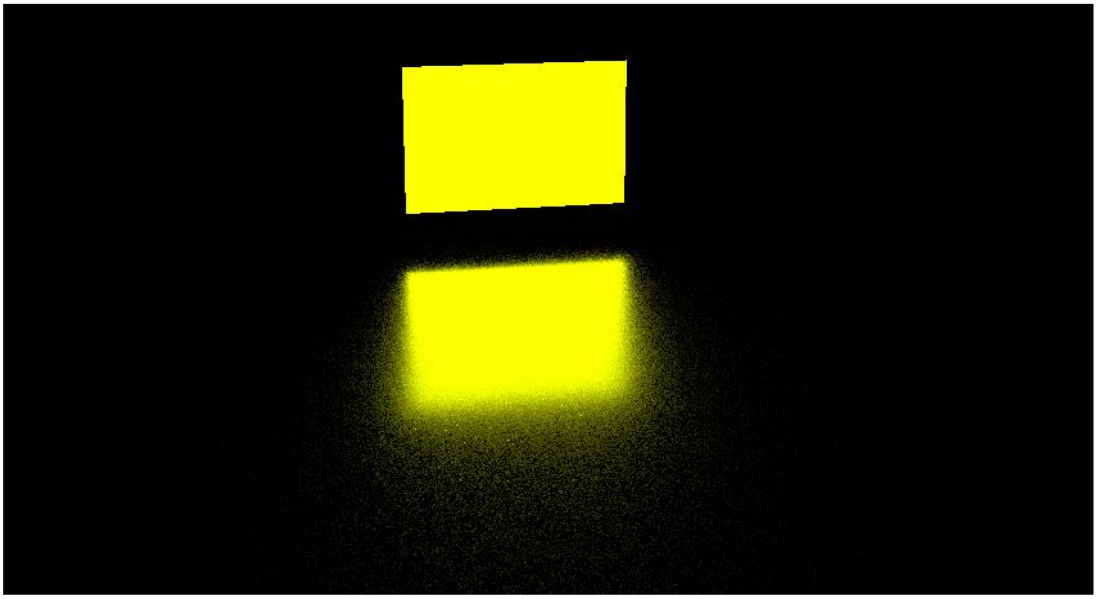

Figura 5.61 Reflejo en superficie especular metálica con importance sampling especular.

5.4.4 Next event estimation

Como objetivo optativo del trabajo, se incluyó la next event estimation:

La *Next Event Estimation* (NEE), introducida por *Eric Veach* en su tesis doctoral, "*Robust Montecarlo Methods for Light Transport Simulation.*" en 1997, [86] es una técnica utilizada para reducir el ruido y el número de cálculos. Consiste en medir la

contribución directa de las fuentes de luz de área hacia un punto de choque, p, en la escena, trazando varios rayos aleatorios directamente hacia una o varias fuentes.

La iluminación directa secundaria resultante se compagina con los métodos de iluminación indirecta. Para implementarla, se han almacenado los objetos emisivos aparte en una estructura de datos que se ha reflejado en la clase NEE.

Figura 5.62 Rayos secundarios iluminación indirecta y rayos secundarios iluminación directa **[88]**.

Paralelamente a la emisión de rayos secundarios de iluminación indirecta, se emiten rayos secundarios aleatorios hacia las fuentes de la luz. La pdf del método es proporcional a la probabilidad de acertar al objeto:

$$p(\omega)d\omega = p_p(q)dA \tag{5.143}$$

La probabilidad de que el rayo pase por una pequeña área de la semiesfera en concreto, es la misma de que el rayo acierte en una pequeña área en el objeto.

Donde, si se proyecta $d\omega$, se tiene.

$$d\omega = \frac{\mathrm{dA} \cdot \cos(\theta)}{(distancia(\vec{p},\vec{q}))^2} \tag{5.144}$$

Luego:

$$p(\omega)\frac{\mathrm{dA} \cdot \cos(\theta)}{(distancia(\vec{p},\vec{q}))^2} = p_p(q)dA \tag{5.145}$$

Es trivial que para las muestras que se reparten en un área se tiene que su pdf es:

$$p_p(q) = \frac{1}{A} \tag{5.146}$$

Se obtiene:

$$p(\omega)\frac{\mathrm{dA}\cdot\cos(\theta)}{(distancia(\vec{p},\vec{q}))^2} = \frac{\mathrm{dA}}{A} \tag{5.147}$$

$$pdf(\omega) = \frac{(distancia(\vec{p},\vec{q}))^2}{\mathrm{A}\cdot\cos(\theta)} \tag{5.148}$$

Con respecto a las pdfs, hay que tener cuidado por las divisiones por números muy pequeños o 0.

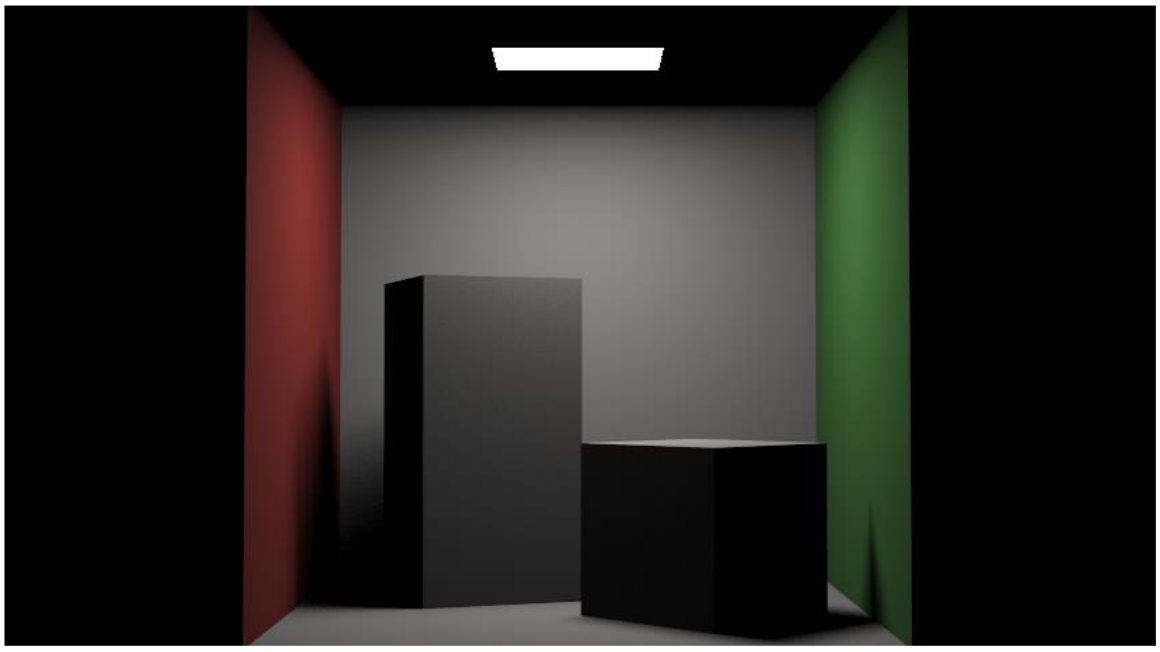

Figura 5.63 NEE sin casi ruido para una profundidad 1 y sobremuestreo 1.

Para implementar NEE (Next Event Estimation), se definió una nueva clase llamada NEE, que almacenaba los objetos emisivos en un vector. Esta clase también almacenaba las áreas superficiales de los objetos para determinar cuántos rayos debían emitirse desde la superficie hacia la luz. Si se producían choques con la luz, se incrementaba la irradiancia debido a la iluminación directa. Además, se estableció que, cuanto más lejos estuviera la fuente de luz, menos rayos se emitirían hacia ella,

lo que se permitía aproximar la luz a un punto cuando se encontraba a gran distancia. El uso de NEE retrasa en menor medida la velocidad de renderizado, en 1/3 en este render.

La técnica indirecta, BRDF, es más ruidosa, pero maneja los reflejos especulares mejor. NEE es mucho mejor para todo excepto para estos reflejos (Figura 5.63). Hay que combinar ambas técnicas de forma no sesgada, dejando casi todo el peso sobre NEE y los reflejos para BRDF.

Para combinar ambas técnicas, NEE y BRDF, *Veach y Guibas* propusieron las heurísticas de poder [88].

$$Irradiancia(\vec{p}, \overrightarrow{\omega_i}) \approx \sum_{i=1}^{N} \frac{1}{N_i} \sum_{j=1}^{N} \omega(\overrightarrow{\omega_i}) \cdot \frac{Irradiancia(\vec{p}, \overrightarrow{\omega_j}, \overrightarrow{\omega_i})}{pdf_i(\overrightarrow{\omega_i}, \overrightarrow{\omega_j})} \tag{5.149}$$

$$\omega(\overrightarrow{\omega_i}) = \frac{pdf_i^{\beta}(\overrightarrow{\omega_i})}{\sum_{K=1}^{N} pdf_k^{\beta}(\overrightarrow{\omega_i})} \tag{5.150}$$

Con β = 2.

Dado que era muy costoso computacionalmente almacenar, tratar, normalizar y multiplicar las PDFs para una infinidad de rayos, opté por utilizar mi propia interpolación para asignar pesos a las contribuciones de radiancia de NEE y BRDF (INEE, IBRDF). Esta interpolación se basa en la metalicidad del objeto y la irradiancia NEE, debido a que los metales reflejan menos irradiancia difusa. El factor de interpolación que se usó es el siguiente:

$$Interpolación = \sqrt{Metalicidad \cdot |Irradiancia_{NEE}|} \tag{5.151}$$

Entendiendo que la irradiancia máxima es la de la sobrexposición y dando un valor máximo a la interpolación de 0.8.

$$Irradiancia(p, \overrightarrow{\omega_i}) \approx \text{I}nterpolación \cdot INEE + (1 - Interpolación) \cdot IBRDF \tag{5.152}$$

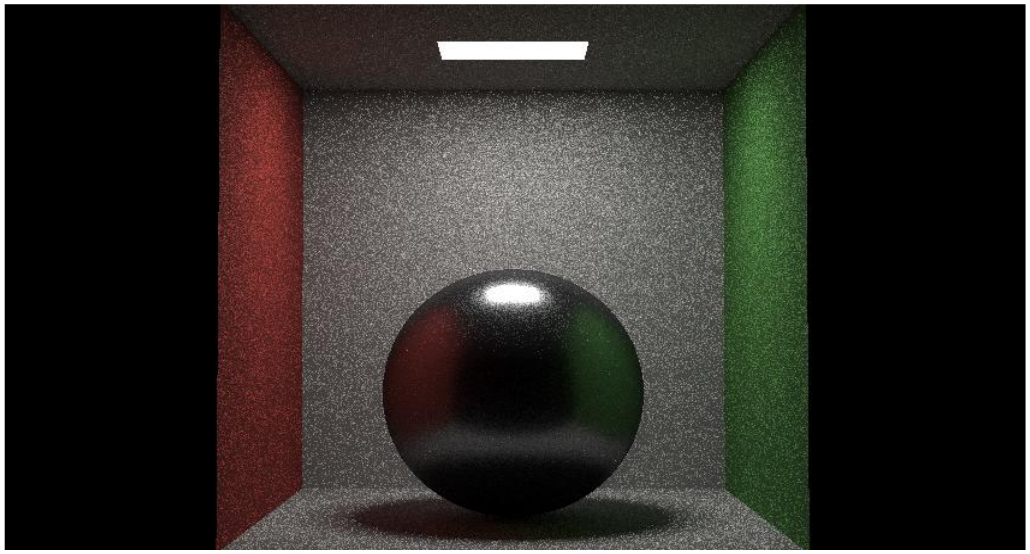

Figura 5.64 MIS con interpolación ½·INEE+½·IBRDF.

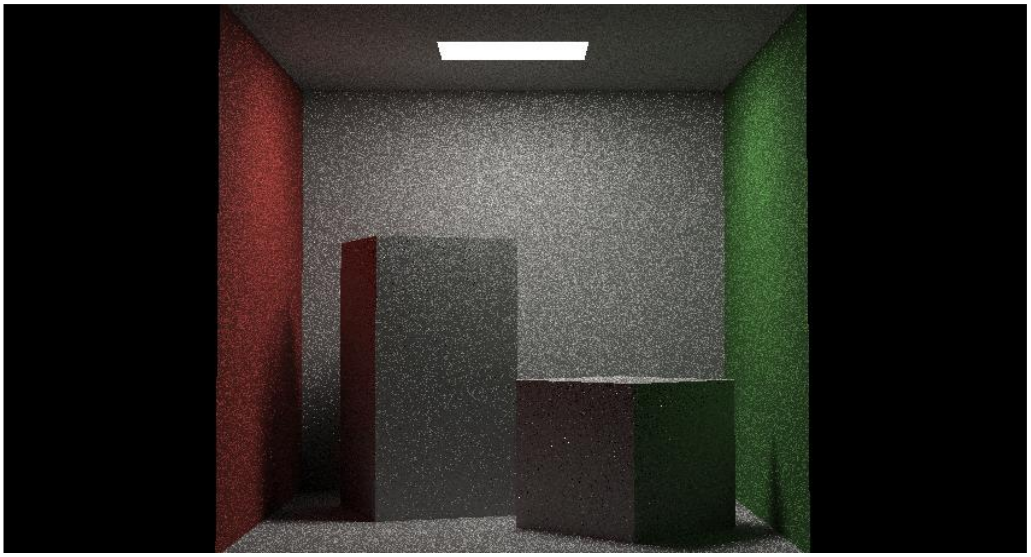

Figura 5.65 MIS con interpolación ½·INEE+½·IBRDF.

Para una interpolación ½ se consiguieron buenos resultados, aunque sesgados. Fíjese en la luz del techo en ambas Figuras 5.65 y 5.64 y Compárese con los métodos no sesgados de IS.

Para la interpolación casera obtenemos el siguiente resultado, muy acertado, Figura 5.66:

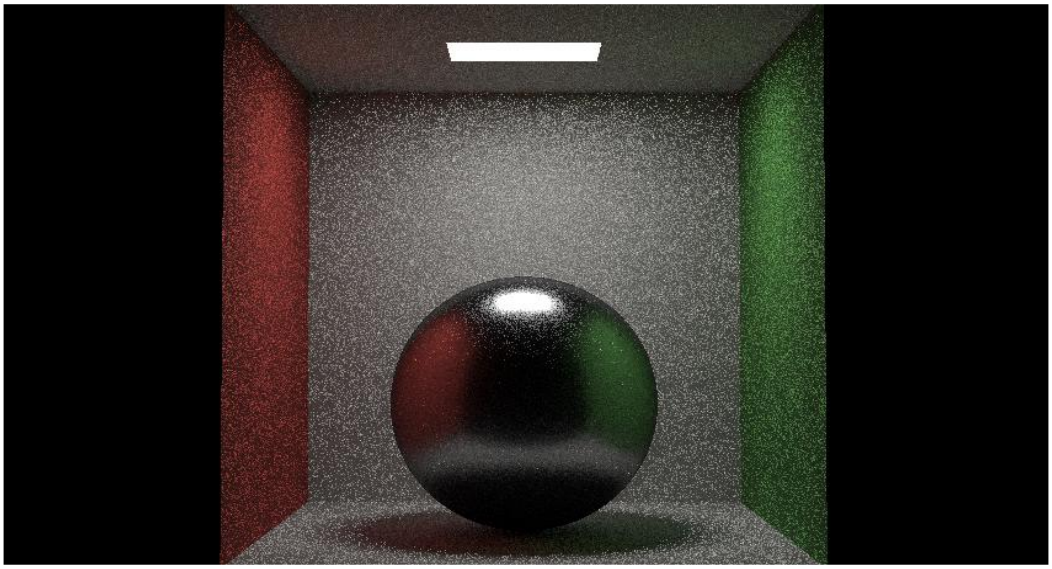

Figura 5.66 MIS con interpolación propia.

5.4.5 Russian roulette path terminator

Por último, se añadió el russian roulette path terminator:

Recordando, el término russian roulette path terminator se refiere a una técnica probabilística utilizada para decidir si un proceso iterativo o recursivo debe continuar o detenerse. Se introduce un factor de azar donde, a cada paso, se calcula la probabilidad de continuar o terminar. Si se decide continuar, el proceso sigue; si se decide detener, el proceso finaliza en ese punto. Esta técnica es eficiente porque evita realizar cálculos innecesarios cuando la contribución adicional de los rayos sería mínima, ahorrando recursos y tiempo de cómputo sin comprometer significativamente la calidad del resultado final [89].

La decisión de continuar o no se basa en el rendimiento o throughput de la iteración actual. La probabilidad de terminación es:

$$q = 1 - \min\left(max\left(T_r, T_g, T_b\right), 1\right) \tag{5.153}$$

Donde T es el throughput para color. Se construye un numero aleatorio y si es menor que q, la recursividad termina.

Por razones estadísticas, para no producir sesgos, si el rayo sobrevive, su parte indirecta y solamente la indirecta, debe ser incrementada. El boost que se la da es:

$$boost = \frac{1}{(1-q)} \tag{5.154}$$

Para calcular T, se puede usar un algoritmo en función del coseno de Lambert o en función de la BRDF, etc. Se aplicó la BRDF a la hora de calcular el Throughput, se observó la creación de ruido si se usa en profundidad baja, se decidió que se activara a partir de la 3 iteración. El algoritmo es capaz de reducir el tiempo de renderizado abruptamente.

Las siguientes figuras, Figura 6.67 y Figura 6.68 son pruebas de validación del uso del algoritmo de *Russian Roulette Path Terminator*.

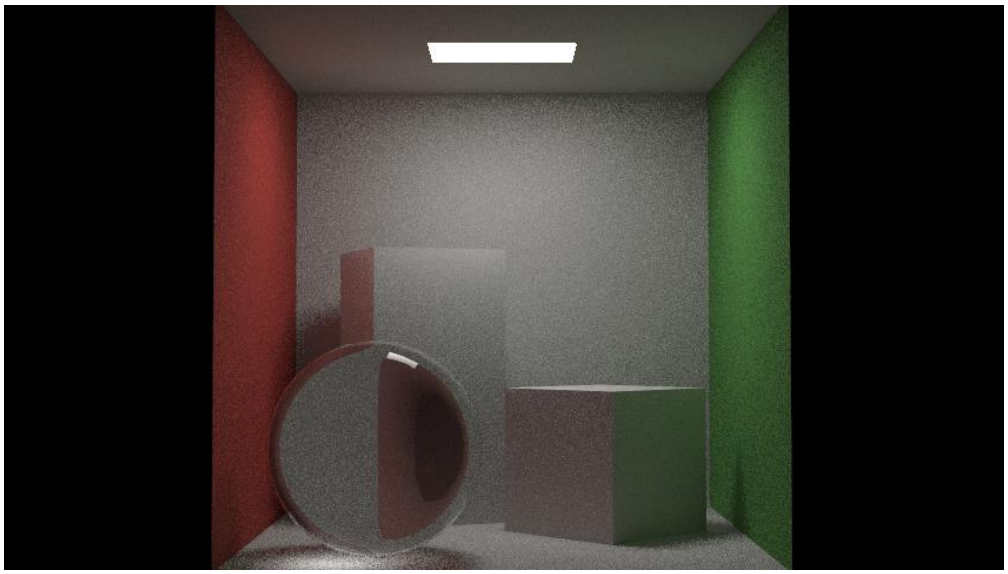

Figura 5.67 Cornell box con sobremuestreo 10.

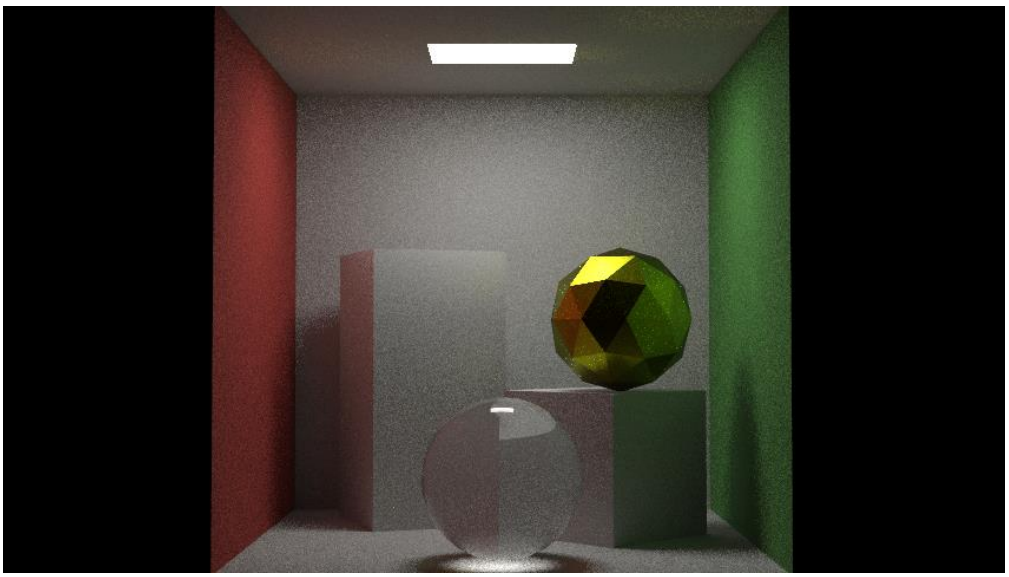

Figura 5.68 Cornell box con sobremuestreo 10.

5.4.6 Conclusiones tercera iteración

En este sprint, se construyó el *Backward Path Tracer*, una técnica avanzada de trazado de rayos que utiliza el Método de Montecarlo para simular con precisión la interacción de la luz en una escena. Se han definido las luces de área y su implementación en el renderizado de una habitación, así como la utilización de la Cornell Box, un estándar en la evaluación de algoritmos de renderizado.

Se ha introducido el concepto de *Multiple Importance Sampling* (MIS); se probó sin éxito estratificación de muestras. Se implementó el *Importance Sampling* (IS), tanto difuso como especular, que proporcionaron una reducción de ruido sin impactar al rendimiento de la aplicación. Además, se ha discutido e incluido la técnica de Next *Event Estimation* (NEE), que mejora enormemente la calidad en el ruido al enfocar los cálculos en las fuentes de luz más relevantes de forma sesgada y suponiendo un aumento leve del tiempo de renderizado.

Finalmente, se ha usado el terminador de caminos russian roulette, una estrategia que optimiza los recursos computacionales al decidir cuándo terminar un rayo cuando su contribución a la irradiancia se estima insignificante.

5.5 Desarrollo de una interfaz de usuario (GUI)

5.5.1 Introducción del desarrollo de la GUI

El proyecto experimentó algunos retrasos, lo que llevó a que la historia de usuario relacionada con la interfaz gráfica de usuario se pospusiera para un sprint final. Durante este último sprint, se completó la implementación de una interfaz gráfica destinada al usuario final, proporcionando una solución interactiva para la experiencia del usuario.

5.5.2 Implementación

Para lograr este objetivo y nueva funcionalidad se hizo el uso de Windows Forms usando esta vez el lenguaje C#, aunque inicialmente se planteó usar la librería de ImGUI por motivos de tiempo se descartó.

Windows Forms (WinForms) es una tecnología de .NET Framework utilizada para crear aplicaciones de escritorio con interfaces gráficas en Windows. Ofrece un conjunto de controles visuales (como botones, cuadros de texto y menús) que se pueden arrastrar y soltar en un entorno de desarrollo visual como Visual Studio, facilitando el diseño de la interfaz. Funciona con un modelo basado en eventos, donde los controles generan eventos que el desarrollador puede manejar mediante código. Aunque es una tecnología más antigua, es fácil de usar y sigue siendo relevante para la creación rápida de aplicaciones de escritorio simples o medianamente complejas.

Se establecieron varias etiquetas, Labels, y varios cuadros interactivos, text boxes. Además, se establecieron reglas en los eventos de las text boxes para no poder incluir datos incorrectos. Se añadió un botón para iniciar la renderización o cancelarla a mitad de proceso. También se añadió una imagen.

Los datos a manejar por el usuario son, el nombre del archivo de la escena en FBX, el nombre de la imagen resultante, la resolución de la imagen, el sobremuestreo, el número de rayos secundarios, la profundidad máxima y la exposición.

Como resultado se obtuvo está sencilla pero eficaz interfaz de usuario:

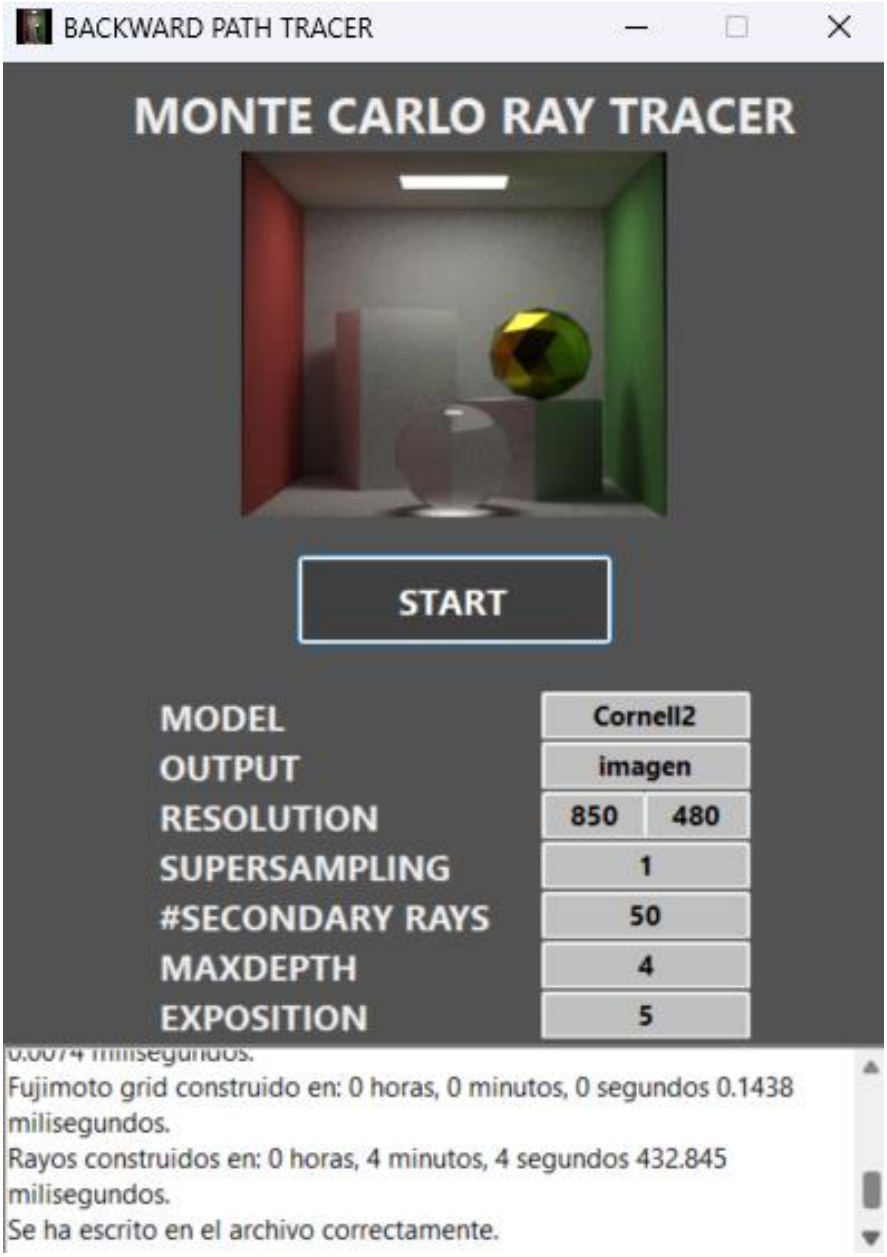

Figura 5.69 Interfaz de usuario.

5.5.3 Conclusiones del desarrollo de una interfaz de usuario

En esta sección se describe la forma en la que se implementó la GUI, mediante WINFORMS, una solución que facilita mucho la creación de interfaces de usuario. El modelo es sencillo, pero efectivo.

5.6 Conclusiones

Este capítulo ofrece una descripción detallada de la implementación de los algoritmos utilizados en el desarrollo del Backward Path Tracer, dividido en cuatro subcapítulos que comprenden desde las primeras iteraciones hasta el desarrollo de la interfaz gráfica de usuario (GUI).

A lo largo de las tres iteraciones del proyecto, se desarrolló un trazador de rayos. En la primera iteración, se implementaron los fundamentos del ray casting, incluyendo el cálculo de intersecciones con primitivas geométricas, una cámara virtual y modelos de sombreado Phong y Blinn-Phong, además de iluminación directa con sombras y diferentes tipos de luces. La segunda iteración resultó en la creación de un Whitted ray tracer, incorporando técnicas de sobremuestreo para reducir el aliasing, aplicación de texturas, algoritmos de reflexión y refracción, el modelo de PBR con el BRDF de Cook-Torrance, corrección gamma y estructuras de aceleración como la rejilla de Fujimoto, hitboxes y OpenMP, aunque hubo retrasos en el cronograma. En la tercera iteración, se construyó el Backward Path Tracer utilizando el método de Montecarlo para simular las interacciones de la luz, implementando luces de área, el Importance Sampling (IS) difuso y especular para reducir el ruido, la técnica de Next Event Estimation (NEE) y el terminador de caminos russian roulette, terminándose la construcción integra de un *Backward Path Tracer* con optimizaciones.

Capítulo 6

Pruebas

6.1 Introducción

La implementación iterativa y la entrega incremental supusieron un sistema de prueba y error de retroalimentación continua. Cuando se identificaba un error se ajustaba el desarrollo en función de los resultados obtenidos.

Durante el desarrollo del Backward Path Tracer, se aplicaron diversos tipos de pruebas para asegurar la calidad y funcionalidad del software. Se realizaron pruebas unitarias para verificar componentes individuales como el cálculo de intersecciones y los modelos de sombreado. Las pruebas de integración aseguraron que los diferentes módulos, como la carga de mallas y la aplicación de texturas, funcionaran correctamente en conjunto. Se llevaron a cabo pruebas de sistema para evaluar el rendimiento del sistema completo. Además, se implementaron pruebas de regresión para garantizar que las nuevas funcionalidades no afectaran las existentes y pruebas de rendimiento para optimizar tiempos de renderizado y uso de recursos.

6.2 Pruebas de validación

Para asegurar la correcta implementación y funcionamiento del Backward Path Tracer, se han diseñado una serie de pruebas de validación alineadas con los sprint backlogs y las historias de usuario definidas. A continuación, se detallan las pruebas

correspondientes a cada sprint, acompañadas de imágenes que sirven como evidencia visual de los resultados obtenidos.

6.3 Iteración 1: Implementación de un trazador de rayos básico

Historia de usuario 1: *Como desarrollador, quiero implementar una forma de generar imágenes.*

- **Prueba de validación:**
    - Verificar que se puedan definir imágenes píxel a píxel en RGB.
- **Evidencia visual:**

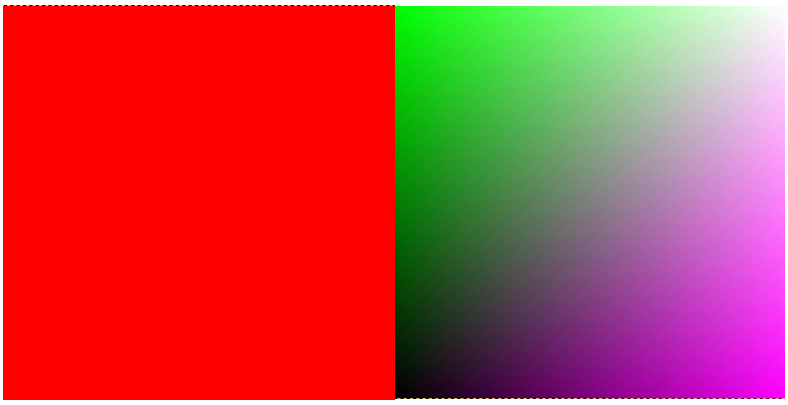

Figura 6.1 Imagen generada donde cada píxel está definido con valores RGB específicos, demostrando la capacidad de generar imágenes básicas.

Historia de usuario 2: *Como desarrollador, quiero implementar el cálculo de intersecciones de rayos con primitivas (esferas y triángulos) para poder definir las geometrías básicas.*

- **Prueba de validación:**
    - Comprobar que las intersecciones con triángulos y esferas se detectan correctamente.
- **Evidencia visual:**

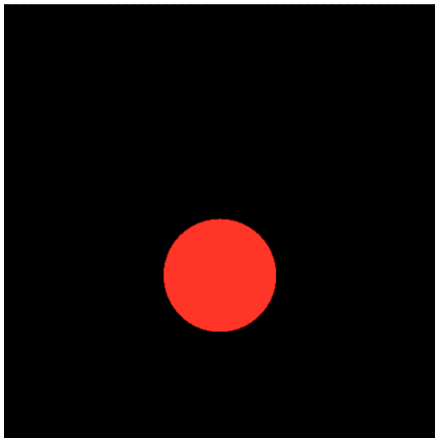

Figura 6.2 Demostración visual de rayos intersecando esferas, mostrando los puntos de intersección correctamente calculados.

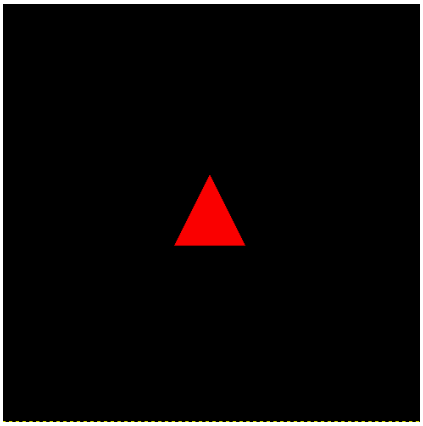

Figura 6.3 Demostración visual de rayos intersecando triángulos, verificando la precisión en el cálculo de intersecciones.

Historia de usuario 3: *Como desarrollador, quiero importar mallas de polígonos y calcular intersecciones usando álgebra aplicada a gráficos para representar modelos más complejos.*

- **Prueba de validación:**
    - Verificar la carga correcta de mallas poligonales y la precisión en el cálculo de intersecciones para modelos complejos.
- **Evidencia visual:**

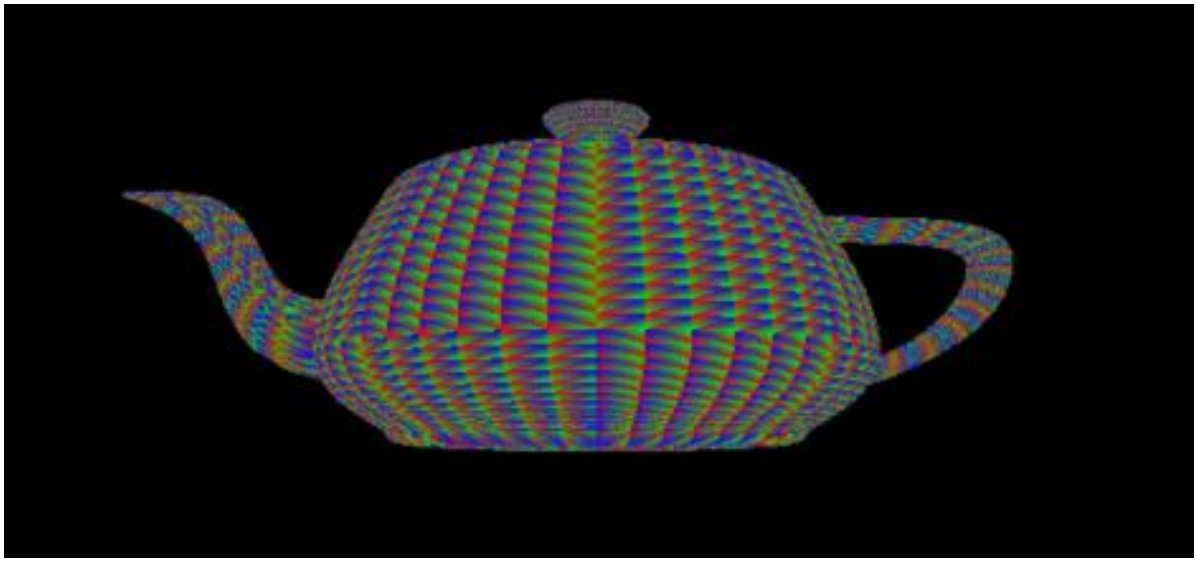

Figura 6.4 Visualización de una malla poligonal cargada, mostrando la correcta representación y ubicación de múltiples polígonos.

Historia de usuario 4: *Como desarrollador, quiero implementar los modelos de sombreado Phong y Blinn-Phong con luces puntuales, direccionales y de foco para simular efectos de iluminación realista.*

- **Prueba de validación:**
    - Verificar la correcta implementación de los modelos Phong y Blinn-Phong, así como el efecto de diferentes tipos de luces en la escena.
- **Evidencia visual:**

Figura 6.5 Comparativa de la iluminación utilizando el modelo Phong.

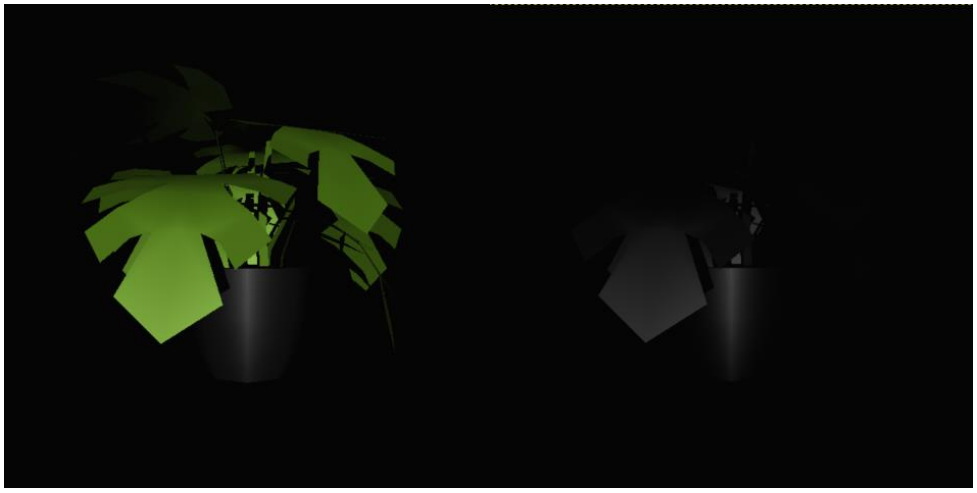

Figura 6.6 Comparativa de la iluminación utilizando el modelo Blinn-Phong.

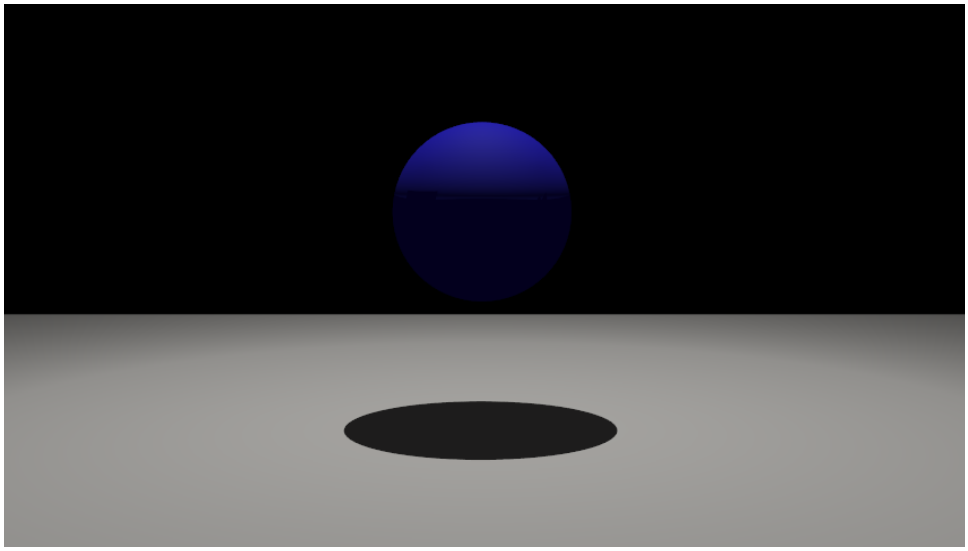

Figura 6.7 Escena iluminada con luz puntual.

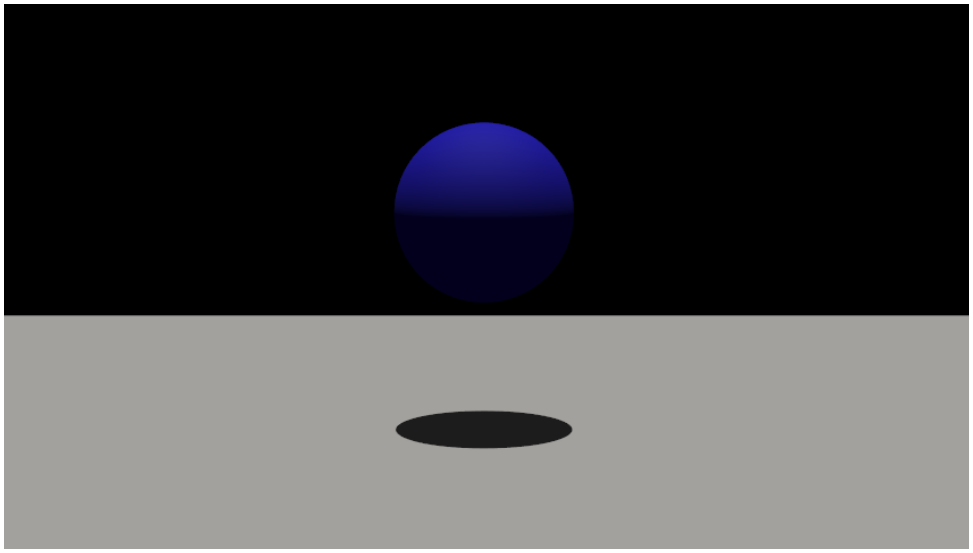

Figura 6.8 Escena iluminada con luz direccional.

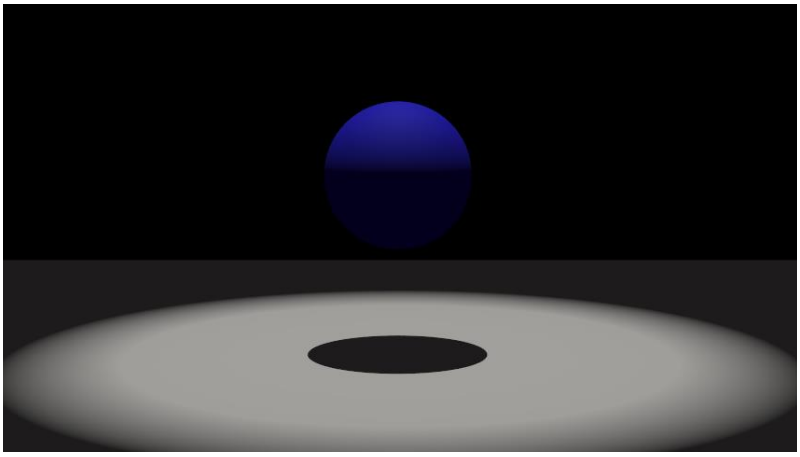

Figura 6.9 Escena iluminada con luz de foco.

6.4 Iteración 2: Transformación del ray caster en Whitted ray tracer

Historia de usuario 1: *Como desarrollador, quiero implementar sobremuestreo para mejorar la calidad visual del renderizado, reduciendo el aliasing y posteriormente el ruido en el Backward Path Tracer.*

- **Prueba de validación:**
    - Verificar que el sobremuestreo reduce los artefactos visuales como el aliasing.
- **Evidencia visual:**

Figura 6.10 Comparación de imágenes renderizadas con y sin sobremuestreo, evidenciando la reducción de aliasing.

Historia de usuario 2: *Como desarrollador, quiero implementar texture mapping para aplicar texturas a las superficies y mejorar el realismo.*

- **Prueba de validación:**

- Verificar la correcta carga y aplicación de texturas en diferentes formatos sin artefactos.

- **Evidencia visual:**

Figura 6.11 Superficies con texturas aplicadas, mostrando la precisión y ausencia de artefactos en el mapeo.

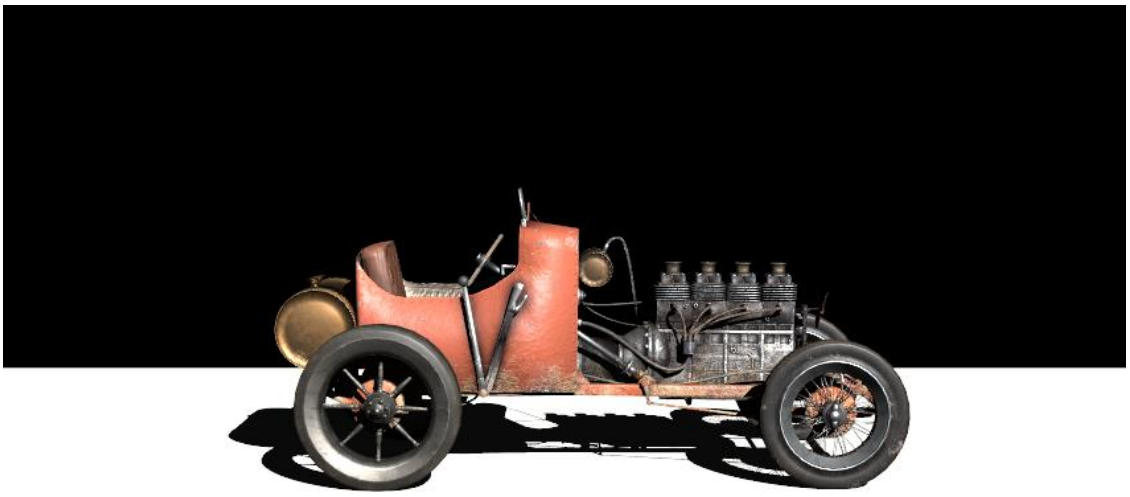

Figura 6.12 Superficies con texturas aplicadas, mostrando la precisión y ausencia de artefactos en el mapeo.

Historia de usuario 3: *Como desarrollador, quiero implementar la ley de Snell y las ecuaciones de Fresnel para simular correctamente la refracción y reflexión en materiales transparentes.*

- **Prueba de validación:**

- Comprobar que la refracción y reflexión siguen la ley de Snell y las ecuaciones de Fresnel.

- **Evidencia visual:**

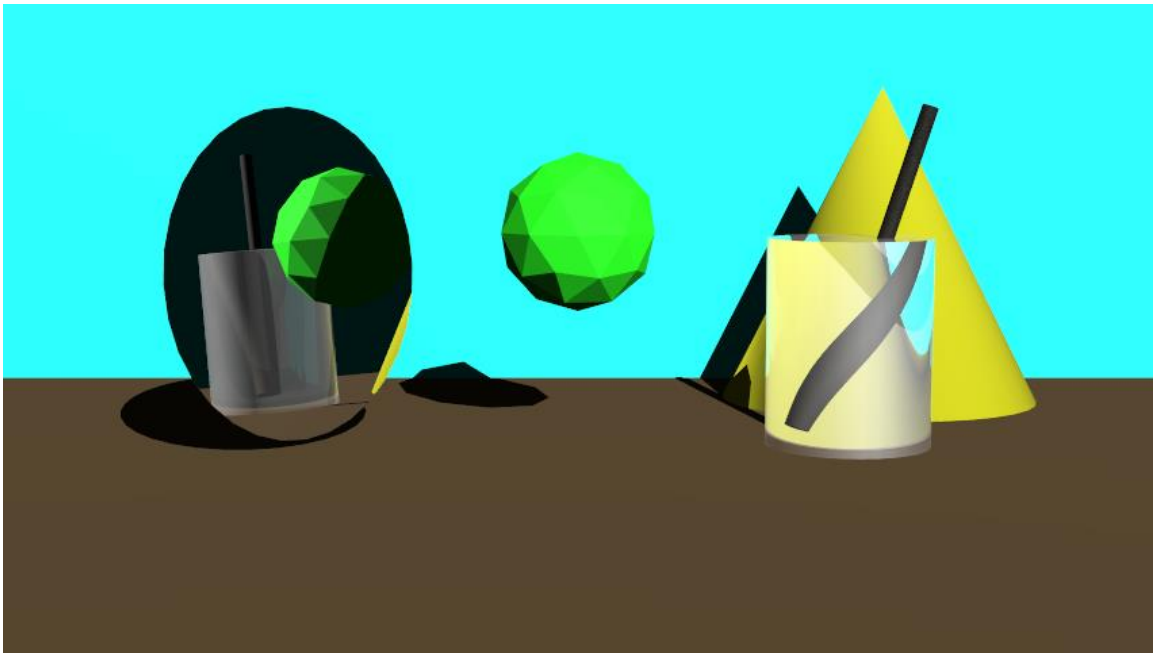

Figura 6.13 Escena que muestra rayos refractados y reflejados en materiales transparentes, verificando la correcta implementación de las leyes físicas.

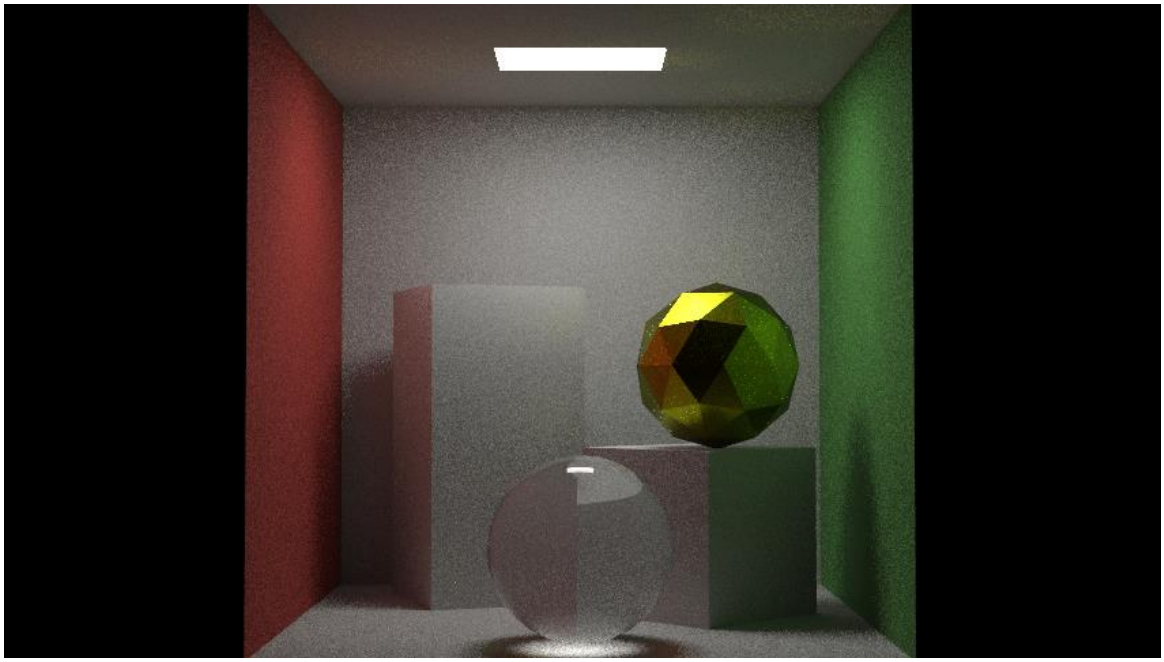

Figura 6.14 Escena que muestra rayos refractados y reflejados en materiales transparentes, verificando la correcta implementación de las leyes físicas.

Historia de usuario 4: *Como desarrollador, quiero implementar el Renderizado Basado en Físicas (PBR) aplicando técnicas avanzadas de radiometría para un realismo más profundo.*

- **Prueba de validación:**
    - Verificar la implementación de las ecuaciones de Cook-Torrance y la simulación de materiales metálicos y dieléctricos.
- **Evidencia visual:**

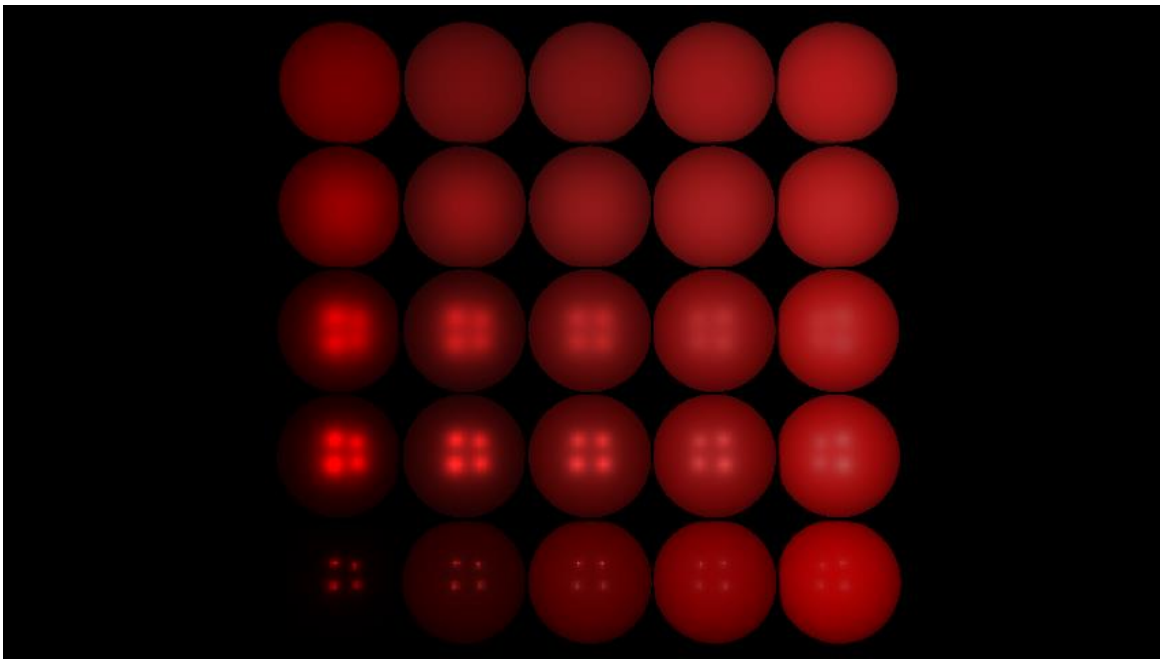

Figura 6.15 Materiales metálicos y dieléctricos renderizados con PBR, mostrando realismo.

Historia de usuario 5: *Como desarrollador, quiero desarrollar estructuras de datos optimizadas y algoritmos de aceleración para mejorar el rendimiento del renderizado.*

- **Prueba de validación:**
    - Verificar el cálculo de intersecciones con bounding boxes y la implementación de estructuras de aceleración como las bounding boxes o la red de Fujimoto.
- **Evidencia visual:**

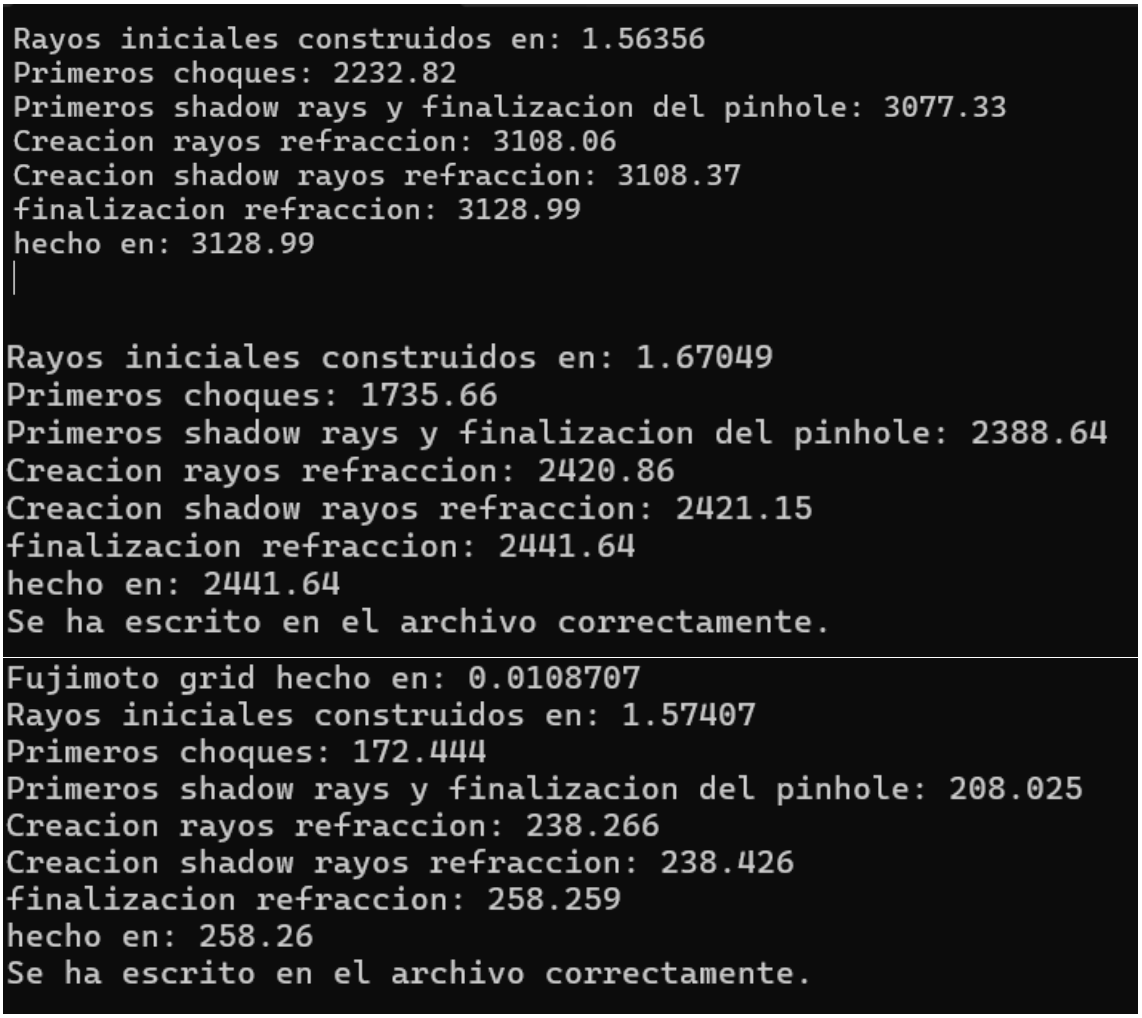

Figura 6.16 Visualización del tiempo de aceleración de las estructuras de datos implementadas y su impacto en el rendimiento del renderizado en ms.

6.5 Iteración 3: Implementación de un backward path tracer

Historia de usuario 1: *Como desarrollador, quiero implementar la iluminación global.*

- **Prueba de validación:**
    - Verificar la implementación del método de Montecarlo para la emisión de rayos secundarios y la generación de sombras con bordes suaves.
- **Evidencia visual:**

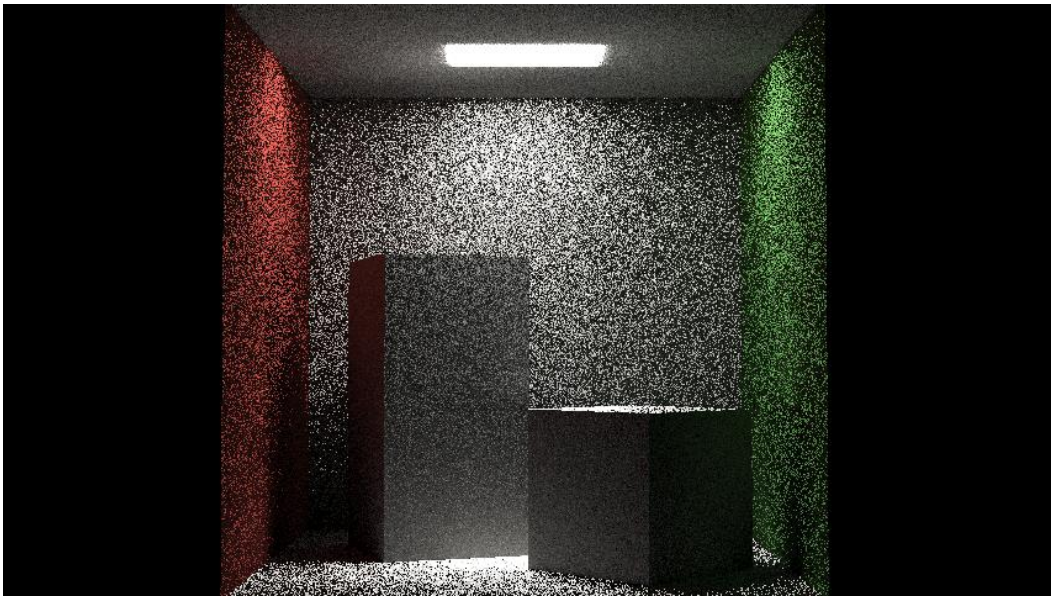

Figura 6.17 Escena renderizada con iluminación global, mostrando sombras suaves y graduales.

Historia de usuario 2: *Como desarrollador, quiero implementar luces de área para simular fuentes de luz más realistas y volumétricas.*

- **Prueba de validación:**
    - Comprobar que las luces de área iluminan correctamente las superficies y proyectan sombras difusas.
- **Evidencia visual:**

Figura 6.18 Escena con luces de área, evidenciando la iluminación realista y las sombras difusas proyectadas.

Historia de usuario 3: *Como desarrollador, quiero implementar Importance Sampling para mejorar la eficiencia del muestreo y reducir el ruido en la imagen.*

- **Prueba de validación:**
    - Verificar que la técnica de Importance Sampling reduce el ruido general sin comprometer el rendimiento.
- **Evidencia visual:**

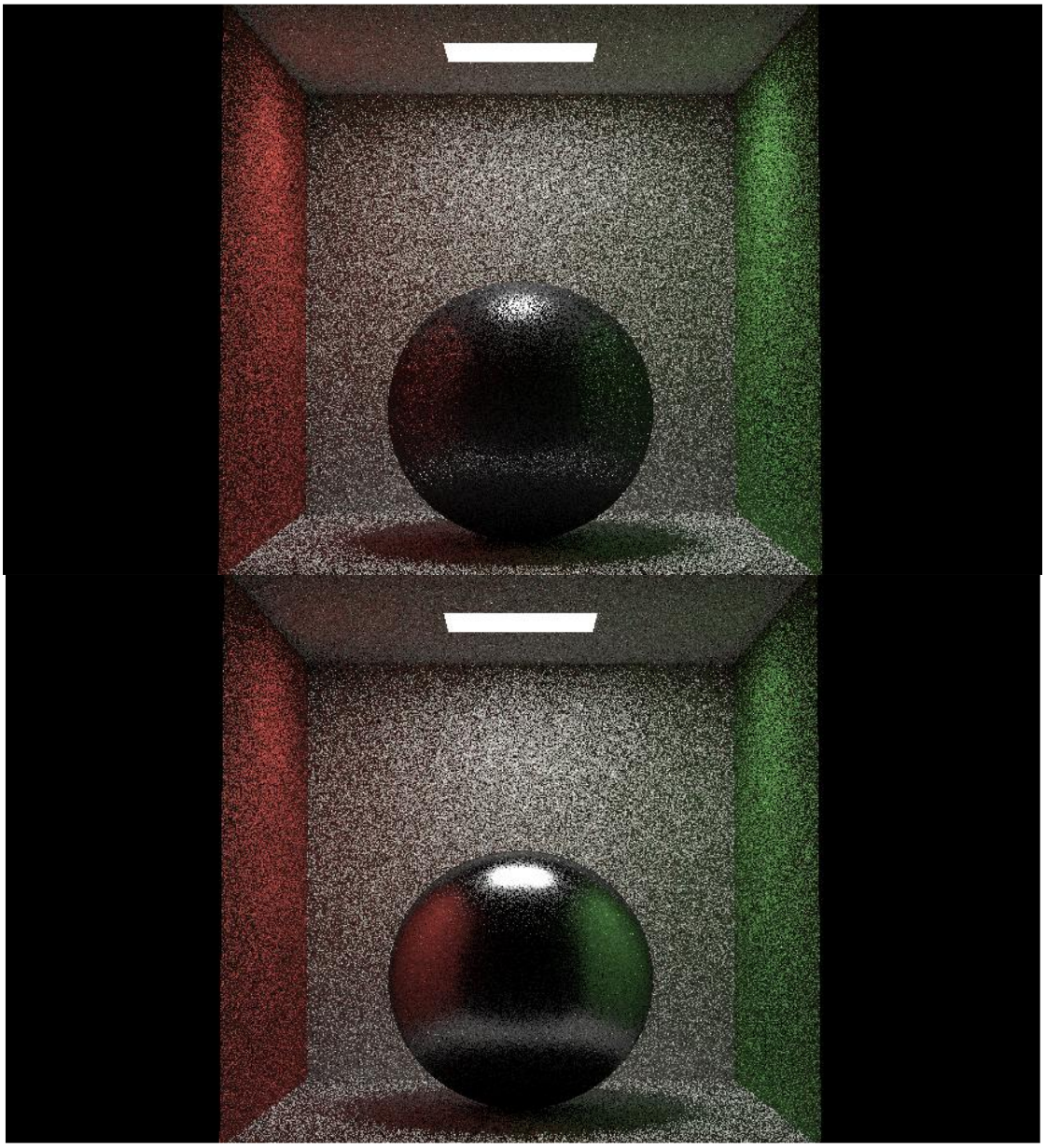

Figura 6.19 Comparativa de imágenes con y sin Importance Sampling, demostrando la reducción del ruido.

Historia de usuario 4: *Como desarrollador, quiero implementar la técnica de Next Event Estimation para mejorar el cálculo de la iluminación indirecta.*

- **Prueba de validación:**
    - Comprobar que el cálculo de iluminación indirecta mejora en términos de ruido sin incrementar significativamente el tiempo de renderizado.
- **Evidencia visual:**

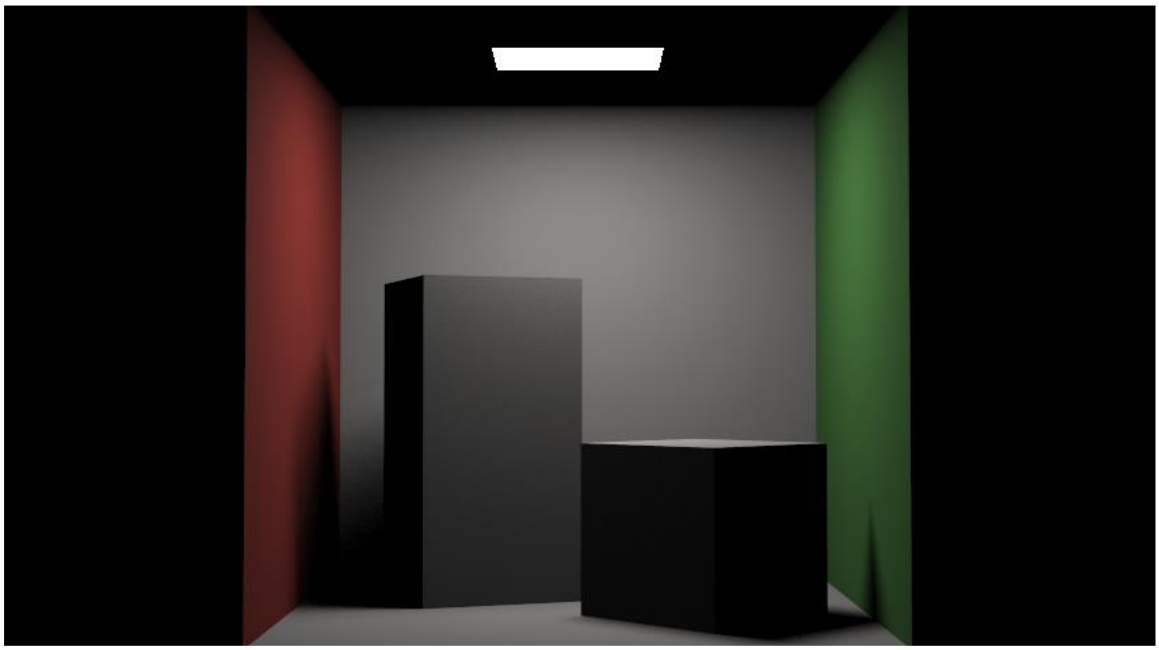

Figura 6.20 Escena renderizada con Next Event Estimation, mostrando una iluminación indirecta más precisa y con menor ruido.

6.6 Conclusiones

En este capítulo se ha hecho una breve descripción del tipo de pruebas realizadas, tanto de caja blanca como de caja negra durante las iteraciones. También se han mostrado alguna de las pruebas de validación que acreditan el correcto funcionamiento del *Ray Tracer* por iteración.

Capítulo 7

Conclusiones y trabajos futuros

7.1 Introducción

El proyecto presentado finaliza con la creación satisfactoria de un Backward Path Tracer. El desarrollo se estructuró en tres fases iterativas: la construcción de un ray caster básico, su evolución a un Whitted Ray Tracer, y finalmente, la implementación de un sistema completo de iluminación global.

7.2 Conclusiones

Este proyecto ha supuesto un exhaustivo estudio e implementación de diversas técnicas de renderizado 3D, con un enfoque especialmente dirigido a las relacionadas con el Ray Tracing.

Se han aplicado exitosamente algoritmos optimizados como el de Möller-Trumbore y el de intersección con cajas, que se tradujeron en la visualización de imágenes mediante archivos PPM. También se implementó la carga de mallas poligonales gracias a la librería ASSIMP y STB, permitiendo la creación de escenas con múltiples objetos, materiales y texturas, con la posibilidad de rotarlos, moverlos o transformarlos a través de una estructura en árbol basada en nodos.

En una primera iteración, se construyó el sombreado de Phong y su variante Blinn-Phong, algoritmos empíricos y ligeros que ofrecieron buenos resultados. Sin embargo, al no estar basados en principios físicos, requieren numerosos ajustes para lograr

efectos de iluminación consistentes. En una segunda iteración, estos algoritmos fueron reemplazados por las ecuaciones de Cook-Torrance, una de las soluciones más rápidas y sencillas dentro del PBR (Physical Based Renderizado). Estas ecuaciones, basadas en la radiometría, buscan aproximar las leyes de Maxwell, con el objetivo de generar imágenes lo suficientemente realistas como para engañar al ojo humano, una ciencia que está en plena expansión. Este enfoque permite que el sombreado sea más consistente, independientemente de la disposición de la escena o la iluminación.

Además, se implementaron técnicas para reproducir reflejos y refracciones en materiales transparentes o translúcidos, logrando un árbol de rayos binarios que al renderizarse resultan en un Whitted Ray Tracer.

Por último, se construyó un sistema de iluminación global, que genera iluminación indirecta mediante los reflejos de la luz sobre los objetos, creando gradientes de luz y sombra que simulan a las sombras reales. Para conseguirlo, se implementó con éxito el método de Montecarlo.

La convergencia y la reducción de ruido se optimizaron utilizando técnicas de Importance Sampling, aplicando probabilidades en el muestreo tanto para reflexiones especulares como difusas. Como objetivo adicional, se incorporó el Next Event Estimation, un método de reducción de ruido que, combinado con el Importance Sampling, forma el Multiple Importance Sampling, una técnica que ha demostrado resultados sorprendentes en la reducción de ruido. Finalmente, se utilizó el Russian Roulette Path Terminator, una técnica inherente del método de Montecarlo que permite una convergencia más rápida, aunque introduce algo de ruido, lo que redujo notablemente los cálculos necesarios para generar imágenes.

En definitiva, los objetivos se han cumplido uno a uno, siguiendo una metodología ágil. Se ha construido un Backward Path Tracer, cuyos resultados y pruebas de validación son claramente visibles en las imágenes generadas y presentadas a lo largo de la memoria.

7.3 Trabajos futuros

En un próximo proyecto, con los conocimientos adquiridos, me gustaría profundizar en el estudio de APIs como Vulkan u OpenGL para implementar un path tracer avanzado que combine el procesamiento en GPU y CPU. Mi objetivo es desarrollar tanto un modelo en tiempo real (online) como uno offline. Además, planeo explorar técnicas mencionadas, pero no investigadas en profundidad, como Ray Marching, Forward Path Tracing, Bidirectional Ray Tracing, Photon Mapping y Metropolis Light Transport.

Bibliografía

[1] P. B. K. B. Philip Dutré, Advanced Global Illumination, 2ª edición ed., A K Peters/CRC Press, 2006.

[2] P. S. Steve Marschner, Fundamentals of Computer Graphics, 4ª edición ed., CRC Press, 2018.

[3] P. Shirley, Ray Tracing in One Weekend, 2016.

[4] E. a. Jean-Colas Prunier, «https://www.scratchapixel.com,» septiembre 2024. [En línea].

[5] J. d. Vries, «https://learnopengl.com,» septiembre 2024. [En línea].

[6] P. Shirley, «https://raytracing.github.io/,» septiembre 2024. [En línea].

[7] Imagen, «https://la.blogs.nvidia.com/blog/que-es-el-path-tracing/,» Septiembre 2024. [En línea].

[8] K.-T. Chang, Programming ArcObjects with VBA: A Task-Oriented Approach, Second Edition ed., CRC Press, 2007.

[9] S. D. Roth, "Ray Casting for Modeling Solids", Computer Graphics and Image Processing, February 1982.

[10] P. Shirley, Realistic Ray Tracing., 2nd edition ed., A K Peters/CRC Press, 2003.

[11] J. T. Kajiya, «The rendering equation,» de *Proceedings of the 13th annual conference on Computer graphics and interactive techniques.*, 1986.

[12] I. Quilez, «https://iquilezles.org/articles/raymarchingdf/,» septiembre 2024. [En línea].

[13] I. P. Model, «https://en.wikipedia.org/wiki/Phong_reflection_model,» Septiembre 2024. [En línea].

[14] B. T. Phong, «Illumination for computer generated pictures,» *Communications of ACM 18 ,* 1975.

[15] I. Phong, «https://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_reflection_model, » septiembre 2024. [En línea].

[16] Imagen, «https://en.wikipedia.org/wiki/Texture_mapping,» septiembre 2024. [En línea].

[17] H. Wang, «"Texture Mapping",» 2013.

[18] A. Reshetov, «Morphological antialiasing,» de *HPG'09: Proceedings of the Conference on High Performance Graphics*, New York, 2009.

[19] Imagen, «https://es.wikipedia.org/wiki/Funci%C3%B3n_de_distribuci%C3%B3n_de_reflectancia_bidireccional,» septiembre 2024. [En línea].

[20] S. l. espectral, «https://docta.ucm.es/rest/api/core/bitstreams/eefb2757-9fc9-423e-97a6-3950d19c3440/content,» septiembre 2024. [En línea].

[21] LearnOpenGL, «https://learnopengl.com/PBR/Theory,» septiembre 2024. [En línea].

[22] B. Duvenhage, «Numerical verification of bidirectional reflectance distribution functions for physical plausibility,» de *Proceedings of the South*

African Institute for Computer Scientists and Information Technologists Conference, 2013.

[23] LearnOpenGL, «https://learnopengl.com/PBR/Theory,» septiembre 2024. [En línea].

[24] Imagen, «https://es.wikipedia.org/wiki/Ley_de_Lambert],» Septiembre 2024. [En línea].

[25] Imagen, «https://es.wikipedia.org/wiki/Reflexi%C3%B3n_difusa,» Septiembre 2024. [En línea].

[26] G. B. Goral Torrance, «Modeling the Interaction of Light Diffuse Surfaces,» *ACM SIGGRAPH Computer Graphics,* 1984.

[27] Imagen, «https://en.wikipedia.org/wiki/Rendering_equation,» septiembre 2024. [En línea].

[28] J. T. Kajiya, «THE RENDERING EQUATION,» *California Institute of Technology,* 1986.

[29] Imagen, «https://es.wikipedia.org/wiki/%C3%81ngulo_s%C3%B3lido,» septiembre 2024. [En línea].

[30] https://en.wikipedia.org/wiki/Monte_Carlo_method, Monte Carlo Methods, Wiley-VCH, 2008.

[31] P. tracing, «https://en.wikipedia.org/wiki/Path_tracing,» septiembre 2024. [En línea].

[32] standford.edu, «https://cs.stanford.edu/people/eroberts/courses/soco/projects/1997-98/ray-tracing/types.html#Forward%20Ray%20Tracing,» septiembre 2024. [En línea].

[33] B. Kerbl, «Rendering: Next Event Estimation,» TU Wien, Austria.

[34] P. Hanrahan, «CS348B Lecture 12 extra,» 2002.

[35] P. book, «https://pbr-book.org/3ed-2018/Monte_Carlo_Integration/Russian_Roulette_and_Splitting,» septiembre 2024. [En línea].

[36] B. surfaceoptics, «https://surfaceoptics.com/bsdf-brdf-btdf-review-of-measurement-approaches/,» septiembre 2024. [En línea].

[37] C. &. F. M. Kulla, «Importance sampling techniques for path tracing in participating media.,» *ComputerGraphicsForum,* vol. 31, nº 4, pp. pp. 1519-1528, 2021.

[38] H. Jensen, «Global Illumination using Photon Maps,» *Rendering Techniques (Proceedings of the Seventh Eurographics Workshop on Rendering),* p. 21–30, 1996.

[39] pbr-book, «https://www.pbr-book.org/3ed-2018/Monte_Carlo_Integration/Importance_Sampling,» septiembre 2024. [En línea].

[40] M. L. Transport, «http://graphics.stanford.edu/papers/metro/,» septiembre 2024. [En línea].

[41] pbr-book, «https://www.pbr-book.org/3ed-2018/Light_Transport_III_Bidirectional_Methods/Metropolis_Light_Transport,» septiembre 2024. [En línea].

[42] E. H. N. H. Tomas Akenine-Möller, Real-Time Rendering.

[43] I. scratchapixel, «https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-acceleration-structure/introduction.html,» septiembre 2024. [En línea].

[44] T. y. J. H. W. Hachisuka, «Stochastic Progressive Photon Mapping,» *ACM SIGGRAPH,* 2009.

[45] I. Map, «https://docs.chaos.com/display/VRAY3SOFTIMAGE/Irradiance+Map,» septiembre 2024. [En línea].

[46] S. F. A. A. K. B. M. D. D. P. G. Bruce Walter, «Lightcuts: A Scalable Approach to Illumination,» *SIGGRAPH,* 2005.

[47] M. e. a. Kettunen, «Path Space Regularization for Holistic and Robust Light Transport,» *ACM Transactions on Graphics,* 2015.

[48] N. Research, «Neural Radiance Caching for Path Tracing,» *SIGGRAPH,* 2021.

[49] T. N. Müller, «Practical Path Guiding for Efficient Light-Transport Simulation,» *ACM Transactions on Graphics,* 2017.

[50] A. e. a. Wilkie, «Spectral Ray Differentials,» *Eurographics,* 2014.

[51] C. 3. Models, «https://en.wikipedia.org/wiki/List_of_common_3D_test_models,» septiembre 2024. [En línea].

[52] I. C. Box, «https://en.wikipedia.org/wiki/Cornell_Box,» septiembre 2024. [En línea].

[53] Imagen, «https://en.wikipedia.org/wiki/Utah_teapot,» septiembre 2024. [En línea].

[54] S. Atrium, «https://embed-3dwarehouse-classic.sketchup.com/model/u28505339-7bf4-46d7-81c2-531d83b5c1c2/Sponza-Atrium,» septiembre 2024. [En línea].

[55] S. M. Scene, «https://3dworldgen.blogspot.com/2017/01/san-miguel-scene.html,» septiembre 2024. [En línea].

[56] Modelo, «https://media.sketchfab.com/models/087054490fc44681aad746a4e9d4d

cf9/thumbnails/9157b8314f384b6ca3d0d5888792fcec/1024x576.jpeg,» septiembre 2024. [En línea].

[57] S. 3. Repository, «http://graphics.stanford.edu/data/3Dscanrep/,» septiembre 2024. [En línea].

[58] Imagen, «https://www.blender3darchitect.com/wp-content/uploads/2018/12/BistroExterior_700px_op.jpg,» septiembre 2024. [En línea].

[59] Imagen, «http://graphics.stanford.edu/data/3Dscanrep/stanford-bunny-cebal-ssh.jpg,» septiembre 2024. [En línea].

[60] Imagen, «https://s3.amazonaws.com/cgcookie-rails/uploads/1609866706467-01+-+Suzanne.png,» septiembre 2024. [En línea].

[61] Modelo, «https://media.sketchfab.com/models/d6b85e8dc4b54269b3df6c7e1e5541ba/thumbnails/c8b1e55a6a3e47a09a958b68f603cde1/1024x576.jpeg,» septiembre 2024. [En línea].

[62] Imagen, «https://www.wolframcloud.com/obj/resourcesystem/images/03c/03c07bce-5d63-4fab-acc2-ab9614745588/72386679ad987c81.png,» septiembre 2024. [En línea].

[63] «The State of Rendering,» *FXGuide,* 2021.

[64] Thepixellab, «https://www.thepixellab.net/best-external-render-engine,» septiembre 2024. [En línea].

[65] L. d. renderizadores, «https://en.wikipedia.org/wiki/List_of_3D_rendering_software,» septiembre 2024. [En línea].

[66] M. Miñan, «https://ejemplosverdes.com/analisis-de-software-5-ejemplos-segun-autor-definicion/,» septiembre 2024. [En línea].

[67] Valtx, «https://www.valtx.pe/blog/metodologias-para-el-desarrollo-de-software-que-son-y-para-que-sirven,» septiembre 2024. [En línea].

[68] L. E. P. Pazos, Metodologías Ágiles en el Desarrollo de Software, Marcombo, 2017..

[69] Nvidia, «https://developer.nvidia.com/discover/ray-tracing,» Septiembre 2024. [En línea].

[70] B. T. Tomas Möller, «"Fast, Minimum Storage Ray-Triangle Intersection".,» *Journal of Graphics Tools,* 1997 .

[71] I. scratchapixel, «https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-box-intersection.html,» Septiembre 2024. [En línea].

[72] S. B. R. K. M. ,. P. S. Amy Williams, «“An Efficient and Robust Ray–Box Intersection Algorithm”,» *University of Utah,* 2005.

[73] Imagen, «https://steamcommunity.com/sharedfiles/filedetails/?l=spanish&id=157462564,» Septiembre 2024. [En línea].

[74] J. Blinn, Artist, *Utah teapot.* [Art].

[75] Imagen, «https://es.wikipedia.org/wiki/Sombreado_de_Phong,» Septiembre 2024. [En línea].

[76] Imagen, «https://es.wikipedia.org/wiki/Sombreado_de_Phong,» septiembre 2024. [En línea].

[77] Turbosquid, Artist, *Houseplant_7.* [Art].

[78] gsn-lib, «https://www.gsn-lib.org/docs/nodes/raytracing.php,» Septiembre 2024. [En línea].

[79] S. Kosarevsky, «https://github.com/corporateshark/poisson-disk-generator,» Septiembre 2024. [En línea].

[80] «opengameart.org,» Septiembre 2024. [En línea].

[81] *Medieval Bucket.* [Art]. Profenix Studio.

[82] jeandiz, Artist, *Vintage Racing Car.* [Art]. Turbosquid.

[83] R. Nakamura, Artist, *Blossom.* [Art]. Ryuji Nakamura and Associates.

[84] J.-C. Prunier, «https://www.scratchapixel.com/lessons/3d-basic-rendering/global-illumination-path-tracing/introduction-global-illumination-path-tracing.html,» [En línea]. [Último acceso: Septiembre 2024].

[85] Bescec, Artist, *Isometric Room.* [Art]. Turbosquid.

[86] Imagen, «https://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/RYER/ch03.html,» 2024. [En línea]. [Último acceso: septiembre].

[87] E. Veach, «Robust Monte Carlo Methods for Light Transport Simulation.,» *Ph.D. dissertation, Stanford University,* December 1997.

[88] Tuwien, «https://www.cg.tuwien.ac.at/sites/default/files/course/4411/attachments/08_next%20event%20estimation.pdf,» septiembre 2024. [En línea].

[89] R. Ramamoorthi, «Lectures,» Septiembre 2024. [En línea].

[90] FXguide, «https://www.fxguide.com/fxfeatured/the-state-of-rendering/,» Septiembre 2024. [En línea].

[91] A. v. D. S. K. F. y. J. F. H. James D. Foley, Computer Graphics: Principles and Practice, 3ª edición ed., Addison-Wesley, 2014.

[92] Historia, «https://en.wikipedia.org/wiki/Computer_graphics,» Septiembre 2024. [En línea].

[93] B. T. Phong, «"Illumination for Computer Generated Pictures".,» *Communications of the ACM, Volume 18, Issue 6,* 1973.

[94] nvidia, «https://la.blogs.nvidia.com/blog/que-es-el-path-tracing/,» septiembre 2024. [En línea].

Glosario

A: Area.

AA: Anti-Aliasing.

ACM: Association for Computing Machinery.

Ambient Reflection: Reflexión Ambiente.

API: Application Programming Interface (Interfaz de Programación de Aplicaciones).

ASSIMP: Open Asset Import Library (Biblioteca Abierta de Importación de Activos).

BB: Bounding Box (Caja Envolvente).

Blinn-Phong Shading Model: Modelo de Sombreado Blinn-Phong.

BRDF: Bidirectional Reflectance Distribution Function (Función de Distribución Bidireccional de Reflectancia).

BSDF: Bidirectional Scattering Distribution Function (Función de Distribución Bidireccional de Dispersión).

BSSRDF: Bidirectional Surface Scattering Reflectance Distribution Function (Función de Distribución Bidireccional de Reflectancia y Dispersión de Superficie).

BTDF: Bidirectional Transmittance Distribution Function (Función de Distribución Bidireccional de Transmitancia).

BVH: Bounding Volume Hierarchy (Jerarquía de Volúmenes Envolventes).

c: Albedo (Coeficiente de reflexión difusa).

C: Usado para función de distribución acumulada o como vector de un vértice que señala el centro de un objeto.

CDF: Cumulative Distribution Function (Función de Distribución Acumulada).

Cook-Torrance Model: Modelo de Cook-Torrance.

CPU: Central Processing Unit (Unidad Central de Procesamiento).

d: Vector dirección de un rayo.

D: Distribution Function in Microfacet Reflection Models (Función de Distribución en modelos de reflexión de microfacetas).

Diffuse Reflection: Reflexión Difusa.

E: Irradiancia.

EM: Electromagnético.

F: Fresnel Term (Término de Fresnel).

F_0: Fresnel Reflectance at Normal Incidence (Reflectancia Fresnel en Incidencia Normal).

FBX: Filmbox (Formato de archivo para intercambio de datos 3D).

FOV: Field of View (Campo de Visión).

G: Geometry Function in Reflection Models (Función de Geometría en modelos de reflexión).

GGX Distribution: Distribución GGX.

GPU: Graphics Processing Unit (Unidad de Procesamiento Gráfico).

GPGPU: General-Purpose computing on Graphics Processing Units (Computación de Propósito General en GPU).

GUI: Graphical User Interface (Interfaz Gráfica de Usuario).

H: Halfway Vector (Vector Medio entre la luz y la vista en Blinn-Phong).

I: Intensidad luminosa.

IOR: Index of Refraction (Índice de Refracción).

IS: Importance Sampling (Muestreo por Importancia).

JPEG: Joint Photographic Experts Group (Formato de Compresión de Imágenes).

KD-Tree: K-Dimensional Tree (Árbol K-Dimensional).

L: Radiancia o vector de dirección de la luz en modelos de iluminación.

MIS: Multiple Importance Sampling (Muestreo de Importancia Múltiple).

MLT: Metropolis Light Transport (Transporte de Luz Metrópolis).

n: Índice de refracción y también utilizado para el exponente especular de Phong.

N: Vector Normal a la superficie o número de rayos secundarios en Montecarlo.

NEE: Next Event Estimation (Estimación del Próximo Evento).

o: Vector que señala el vértice del origen de un rayo.

OBJ: Formato de archivo de objeto 3D.

Octree: Estructura de datos que subdivide el espacio en ocho octantes.

OpenGL: Open Graphics Library (Biblioteca Gráfica Abierta).

p: Función de probabilidad o vector que señala un vértice.

PBR: Physically Based Rendering (Renderizado Basado en Física).

PDF: Probability Density Function (Función de Densidad de Probabilidad).

PNG: Portable Network Graphics (Gráficos Portátiles de Red).

PPM: Progressive Photon Mapping (Mapeo Progresivo de Fotones).

R: Vector transmitido o también utilizado para la reflectancia.

RTX: Ray Tracing Technology (Tecnología de Trazado de Rayos de NVIDIA).

SDK: Software Development Kit (Kit de Desarrollo de Software).

SIGGRAPH: Special Interest Group on Computer Graphics and Interactive Techniques.

SIMD: Single Instruction, Multiple Data (Una Instrucción, Múltiples Datos).

Snell: Ley de Snell (Relación entre ángulos de incidencia y refracción).

SPPM: Stochastic Progressive Photon Mapping (Mapeo Progresivo Estocástico de Fotones).

SSS: Subsurface Scattering (Dispersión Subsuperficial).

STB: Biblioteca simple para cargar imágenes (stb_image.h).

t: Distancia a la que se produce un choque.

T: Throughput (Eficiencia en Russian Roulette), vector transmitido o también utilizado para la transmitancia.

UV Mapping: Mapeo de texturas 2D en superficies 3D.

V: Vector de Visión o dirección del observador)

Vulkan: API de gráficos y cómputo de bajo nivel.

Φ: Flujo luminoso.

ω (Omega): Ángulo sólido o dirección de un rayo.

α (Alpha): Rugosidad (Parámetro de rugosidad en modelos de microfacetas).

θ: Ángulo (generalmente utilizado para ángulos de incidencia, reflexión y refracción).

θ_i: Ángulo de incidencia.

θ_r: Ángulo de reflexión.

θ_t: Ángulo de transmisión o refracción.

σ_a: Coeficiente de absorción.

σ_s: Coeficiente de dispersión.

σ_t: Coeficiente de extinción ($\sigma_t = \sigma_a + \sigma_s$).

Anexos

Anexo A. Breve historia del renderizado y de la computación gráfica

Historia Previa

Desde los albores de la humanidad, el ser humano ha representado imágenes en pinturas. Las pinturas rupestres de Altamira, las pinturas del templo de Cnosos o los frescos de la basílica de San Clemente en Roma dan cuenta de ello. A la vez, grandes genios como Tales de Mileto, Pitágoras, Euclides, Arquímedes, y muchos otros, analizaron las formas matemáticamente. El arte y la técnica convergieron en innumerables ocasiones, siendo una de las más sonadas la introducción de la perspectiva por Brunelleschi, allá por 1425. Los grandes artistas eran grandes estudiosos del color, de los efectos atmosféricos, de la luz, de la sombra, de los materiales, de la forma, de la proporción que proyectaban con maestría en sus obras de forma técnica.

Mientras tanto la geometría, el álgebra, la óptica, la colorimetría, el cálculo, la estadística... florecían con grandes genios. Las aportaciones Descartes, Newton, Huyges, Gauss, Maxwell, Galois, Hilbert, Lagrange, Hamilton, Fourier, Rienmann, Laplace, Fresnel y un sin fin de nombres… fueron fundamentales para lo que vendría poco después: la computación gráfica.

Fue un artista y químico francés, Louis Daguerre, quien inventó la fotografía, el gran paso de la tecnología en la generación de imágenes. A finales del siglo XIX, los hermanos Lumiere inventaron el cine, de la misma manera que los efectos especiales.

En 1897, se inventó el tubo de rayos catódicos, el tubo Braun, permitiendo la creación del osciloscopio y de los paneles de control militares. Estos son los precursores directos de la computación gráfica, pues fueron pantallas electrónicas de dos dimensiones que respondían a la interacción de un programa con un usuario.

Inicios de la computación gráfica: los 50

En 1950 la computación gráfica nace como una disciplina de investigación en la universidad y en el laboratorio, especialmente en Estados Unidos, donde tuvo una aplicación militar directa en el desarrollo de tecnologías para el radar o el sonar, entre muchas otras. Todo tipo de nuevas pantallas fueron desarrolladas enriqueciendo su capacidad para ofrecer información de todo tipo, lo que conllevó el desarrollo de la computación gráfica.

La computación gráfica tradicionalmente se divide en dos campos: el campo del tiempo real y el del renderizado offline, utilizado principalmente para producir imágenes de alta calidad en las que el tiempo no es un factor crítico. Podría decirse que la computación gráfica nace en el del real time o tiempo real.

Los primeros proyectos, como Whirlwind y SAGE, establecieron el uso del tubo de rayos catódicos como una interfaz, además de introducir el lápiz óptico como un nuevo dispositivo de entrada. En este contexto, Douglas T. Ross, mientras trabajaba en el sistema Whirlwind SAGE, realizó un experimento personal donde desarrolló un pequeño programa capaz de capturar el movimiento de su dedo y trazar su trayectoria en una pantalla, formando su nombre.

En 1958, uno de los primeros videojuegos con gráficos interactivos, Tennis for Two, fue creado por William Higinbotham en el Brookhaven National Laboratory para entretener a los visitantes. Este juego, diseñado con un osciloscopio, simulaba un partido de tenis.

Al año siguiente, en 1959, mientras trabajaba en el MIT en la conversión de expresiones matemáticas en vectores tridimensionales para herramientas de mecanizado, Douglas T. Ross también generó en una pantalla la imagen de un personaje animado de Disney.

California pronto se convirtió en el principal centro tecnológico del mundo debido a la Universidad de Berkeley y su relación con el mundo empresarial. Una transformación, que duró décadas, comenzó en el área sur de la Bahía de San Francisco, convirtiéndose en lo que hoy conocemos como Sillicon Valley. El campo de la computación gráfica se desarrolló en paralelo a la aparición del hardware especializado en gráficos.

En 1959, la computadora TX-2 del MIT integró nuevas interfaces hombre-máquina, incluyendo un lápiz óptico que, junto con el software Sketchpad de Ivan Sutherland, permitía dibujar y manipular formas en la pantalla de manera interactiva. Sketchpad introdujo conceptos clave en gráficos por computadora, como la creación de objetos modelados, que podían modificarse sin distorsionar otras partes, y la automatización de formas perfectas, como cuadrados, a partir de simples especificaciones de tamaño y ubicación. Esto sentó las bases para muchos de los estándares actuales en interfaces gráficas.

Los 60: el principio del ray casting

En los años sesenta se comenzó a hablar de computación gráfica, término acuñado por William Fetter, un diseñador gráfico de Boeing.

En 1961, Steve Russell, un estudiante del MIT, creó uno de los juegos más influyentes en la historia de los videojuegos: Spacewar!, diseñado para la computadora DEC PDP-1. El juego tuvo un éxito inmediato.

En los años 60, Elizabeth Waldram en la Universidad de Cambridge desarrolló código para mostrar mapas radio-astronómicos en una pantalla de tubo de rayos catódicos. Al mismo tiempo, en Bell Telephone Laboratory, E. E. Zajac creó una película generada por computadora que mostraba cómo cambiar la trayectoria de un satélite en órbita, y Ken Knowlton, Frank Sinden y otros comenzaron a trabajar en gráficos por computadora para ilustrar leyes físicas de Newton. En Renault, Pierre Bézier utilizó las curvas de Casteljau para desarrollar técnicas de modelado 3D en el diseño de carrocerías de automóviles, sentando las bases para el trabajo futuro en modelado de curvas.

En los años 60, empresas como IBM y Magnavox impulsaron el desarrollo de gráficos por computadora, con IBM lanzando el primer terminal gráfico comercial y Ralph Baer creando el videojuego doméstico Odyssey en 1966.

Ese mismo año, Ivan Sutherland, innovador del MIT, creó el primer casco de realidad virtual (HMD) que permitía ver imágenes estereoscópicas en 3D.

David C. Evans y Ivan Sutherland fueron figuras clave en la Universidad de Utah, convirtiendo su programa en un centro líder de investigación en gráficos. En 1968, ambos fundaron Evans & Sutherland, la primera empresa de hardware de gráficos por computadora.

En 1969 también llegó el hombre a la Luna. Ese mismo año, nace el Renderizado Offline: Arthur Appel describió el primer algoritmo de Ray Casting, precursor de los algoritmos de ray tracing usados para lograr el fotorrealismo en computación gráfica.

En 1969, la ACM creó el grupo de interés SIGGRAPH, que organizó su primera conferencia en 1973 y se convirtió en un foro fundamental para la evolución de los gráficos por computadora.

Los años 70: el principio del sombreado

En la década de 1970, se produjeron varios avances importantes en el campo de los gráficos por computadora en la Universidad de Utah, que había contratado a Ivan Sutherland. Junto con David C. Evans, impartió una clase avanzada de gráficos por computadora, que aportó una gran cantidad de investigaciones fundamentales al campo y formó a estudiantes que más tarde fundarían algunas de las empresas más influyentes de la industria, como Pixar, Silicon Graphics y Adobe Systems. Además, Tom Stockham lideró el grupo de procesamiento de imágenes en la universidad, que colaboraba estrechamente con el laboratorio de gráficos por computadora.

Uno de los estudiantes destacados de la Universidad de Utah fue Edwin Catmull. Proveniente de The Boeing Company y con estudios en física, Catmull creció admirando la animación de Disney, aunque pronto descubrió que no tenía talento para el dibujo. Sin embargo, vio en las computadoras el futuro de la animación y quiso ser parte de esa revolución. La primera animación por computadora que Catmull vio

fue la suya propia: una mano abriéndose y cerrándose. En 1974, también fue pionero en el mapeo de texturas, una técnica fundamental en el modelado 3D. Uno de sus objetivos se convirtió en producir un largometraje completamente animado por computadora, un sueño que alcanzaría dos décadas después al cofundar Pixar. En la misma clase, Fred Parke creó una animación del rostro de su esposa, y ambas animaciones se incluyeron en la película Futureworld de 1976.

El laboratorio de gráficos por computadora de la Universidad de Utah atrajo a pioneros como John Warnock, fundador de Adobe Systems, quien revolucionó el mundo editorial con el lenguaje PostScript y desarrolló software como Photoshop y After Effects. También estuvo James Clark, quien fundó Silicon Graphics, una empresa líder en sistemas de renderizado avanzados que dominó los gráficos de alta gama hasta los años 90.

Un avance importante en gráficos 3D fue desarrollado en la Universidad de Utah por estos pioneros: la determinación de superficies ocultas. Esta técnica permite a la computadora identificar qué superficies de un objeto 3D están "detrás" desde la perspectiva del espectador y deben ocultarse al renderizar la imagen (culling). Además, el 3D Core Graphics System fue el primer estándar gráfico desarrollado, establecido por un grupo de expertos de SIGGRAPH en 1977, que sentó las bases para futuros avances en gráficos.

En los años 70, Henri Gouraud, Jim Blinn y Bui Tuong Phong hicieron contribuciones fundamentales al sombreado en gráficos por computadora. Crearon los modelos de sombreado de Gouraud y Blinn–Phong, que dieron a los gráficos mayor realismo al representar la profundidad. En 1978, Blinn también introdujo el bump mapping, una técnica para simular superficies irregulares, precursora de técnicas más avanzadas de mapeo usadas hoy en día.

En la década de 1970, nacieron los videojuegos arcade modernos con gráficos de sprites 2D en tiempo real. Pong (1972) fue uno de los primeros grandes éxitos en este formato. Speed Race (1974) introdujo sprites que se movían por una carretera en desplazamiento vertical, y Gun Fight (1975) presentó personajes animados con apariencia humana. Space Invaders (1978), con numerosos personajes animados en pantalla, utilizó un circuito especializado (barrel shifter) para que el microprocesador

Intel 8080 pudiera gestionar eficientemente los gráficos, marcando un hito en la evolución de los arcades.

Las shadow maps (mapas de sombras) fueron introducidas por primera vez en 1978 por Lance Williams, el mismo investigador que posteriormente presentó las sombras volumétricas. Williams publicó la técnica de shadow mapping en su artículo titulado "Casting Shadows in Computer Generated Images", presentado en la conferencia SIGGRAPH 1978.

Los años 80: nace la ecuación del renderizado y la iluminación Global

En los años 80, la comercialización de los gráficos por computadora se aceleró con la expansión de las computadoras personales. La tecnología de integración a gran escala (VLSI) permitió el desarrollo de los primeros procesadores gráficos (GPU), revolucionando los gráficos de alta resolución. NEC lanzó el primera GPU, el µPD7220, que permitió resoluciones de hasta 1024x1024, marcando el inicio del mercado de gráficos para PC. Con la disminución del costo de la memoria MOS, se hizo posible el uso de framebuffers asequibles, como la VRAM introducida por Texas Instruments. En 1984, Hitachi lanzó el primer GPU CMOS y, en 1986, TI presentó el primer procesador gráfico totalmente programable.

Durante esta década, los terminales gráficos se volvieron más autónomos, migrando el procesamiento gráfico hacia estaciones de trabajo independientes, como la Orca 3000, utilizada en la ingeniería asistida por computadora. Las computadoras personales, especialmente la Amiga y la Macintosh, se volvieron herramientas importantes para artistas y diseñadores gráficos, que valoraban su precisión y eficiencia. La interfaz gráfica de usuario (GUI), que emplea símbolos e íconos en lugar de texto, se convirtió en una característica clave en la tecnología multimedia.

El modelo de microfacetas fue desarrollado en 1982 por Robert Cook y Kenneth Torrance, en un artículo titulado "A Reflectance Model for Computer Graphics" hito en el Renderizado Físicamente Correcto (PBR).

En Japón, la Universidad de Osaka desarrolló el sistema LINKS-1 en 1982, un superordenador diseñado para renderizar gráficos 3D realistas mediante ray tracing, usando hasta 257 microprocesadores Zilog Z8001. Este sistema fue uno de los más potentes del mundo en 1984, logrando una renderización rápida de imágenes altamente realistas.

Radiosity fue realizado por investigadores como Goral, Torrance, Greenberg y Batty, quienes publicaron en 1984 el artículo titulado "Modeling the Interaction of Light Between Diffuse Surfaces" en la conferencia SIGGRAPH. Este trabajo fue pionero en aplicar la técnica para calcular la interacción de la luz entre superficies difusas, lo que permite una simulación precisa de la iluminación indirecta, logrando efectos como el "color bleeding" (el derrame de color entre superficies cercanas).

El primer path tracer fue desarrollado por James Kajiya en su influyente trabajo titulado "The Rendering Equation", presentado en 1986 en la conferencia SIGGRAPH. En este artículo, Kajiya introdujo la Ecuación de Renderizado (Rendering Equation), que formaliza el proceso de transporte de luz en gráficos por computadora, y propuso el path tracing como un método para resolverla, usando el método de Montecarlo. Nace así la Iluminación Global.

En los años 80, la popularidad de Star Wars y otras franquicias de ciencia ficción impulsó el uso de gráficos por computadora (CGI) en el cine, con Lucasfilm e Industrial Light & Magic destacándose como líderes en el sector. Se realizaron avances importantes en la técnica de chroma key (pantalla azul o verde) para las películas posteriores de la trilogía original de Star Wars. Dos videos icónicos de esa época también dejaron una huella duradera: el video casi completamente generado por CGI de "Money for Nothing" de Dire Straits en 1985, que popularizó el CGI entre los fanáticos de la música, y una escena de Young Sherlock Holmes ese mismo año, con el primer personaje completamente creado por CGI en una película (un caballero animado de vidrieras).

Las sombras volumétricas fueron introducidas por primera vez en los gráficos por computadora en 1986, cuando Lance Williams publicó su artículo titulado "Casting Curved Shadows on Curved Surfaces" en la conferencia SIGGRAPH.

En 1988, Pixar, que ya se había separado de Industrial Light & Magic, desarrolló los primeros shaders, pequeños programas diseñados específicamente para sombreado, aunque sus resultados no se verían hasta la década siguiente. A finales de los 80, las computadoras Silicon Graphics (SGI) se utilizaron para crear algunos de los primeros cortometrajes totalmente generados por computadora en Pixar y las máquinas SGI se consideraban el estándar más alto en gráficos por computadora de la época.

Los años 80 se conocen como la "era dorada de los videojuegos". Sistemas como los de Atari, Nintendo y Sega, junto con computadoras personales como MS-DOS, Apple II, Macintosh y Amiga, introdujeron por primera vez a una audiencia joven al mundo de la computación gráfica, permitiendo incluso a los usuarios programar sus propios juegos si tenían las habilidades necesarias. En los arcades, en 1988 se introdujeron las primeras placas dedicadas a gráficos 3D en tiempo real, como el Namco System 21 y el Taito Air System.

En el ámbito profesional, empresas como Evans & Sutherland y Silicon Graphics (SGI) desarrollaron hardware de gráficos 3D rasterizados, lo que influyó en el posterior desarrollo de la unidad de procesamiento gráfico (GPU), que optimiza los gráficos trabajando en paralelo con el CPU. Durante esta década, los gráficos por computadora también se aplicaron en mercados profesionales como el entretenimiento basado en ubicaciones, la educación, el diseño de vehículos, la simulación de vehículos y la química.

Años 90

En la década de 1990, el modelado 3D se masificó y la calidad del CGI mejoró significativamente. Las computadoras domésticas comenzaron a asumir tareas de renderizado que antes estaban limitadas a estaciones de trabajo costosas, lo que impulsó el ascenso de software como 3D Studio en sistemas Windows y Macintosh, mientras disminuía la popularidad de las estaciones de Silicon Graphics. A finales de la década, las GPU comenzaron a ganar protagonismo.

En 1991, Maxwell Hanrahan y Pat Hanrahan introdujeron un método para calcular la dispersión de la luz en volúmenes en su artículo titulado "Ray Tracing Volume

Densities". Este trabajo fue uno de los primeros en proporcionar una técnica práctica para renderizar medios como niebla o humo utilizando ray tracing, simulando de manera eficiente la interacción de la luz con estos medios volumétricos.

Wolfenstein 3D (1992) desarrollado por id Software y dirigido por John Carmack, fue uno de los primeros videojuegos en popularizar los gráficos 3D en tiempo real. Aunque no era técnicamente el primer FPS (First Person Shooter), sí fue el que definió el género. Utilizó un motor gráfico innovador que simulaba un entorno tridimensional a través de la técnica de ray casting.

El CGI comenzó a producir gráficos que para el ojo no entrenado podían parecer fotorrealistas. En los videojuegos, títulos como Virtua Racing (1992) y la consola Sony PlayStation popularizaron los gráficos 3D. Juegos influyentes como Super Mario 64 y The Legend of Zelda: Ocarina of Time también marcaron este cambio más adelante.

En cine, Pixar alcanzó un éxito comercial con Toy Story en 1995, marcando un hito en la animación por computadora. En 1996, se inventó el normal mapping, una mejora del bump mapping. 3Dfx crea las primeras GPU para el mercado doméstico, 3Dfx es adquirida por Nvidia posteriormente. Al final de la década, los gráficos 3D se revolucionaron gracias a APIs gráficas como DirectX y OpenGL. Empresas como Nvidia y AMD se convirtieron en líderes en el desarrollo de hardware gráfico.

Jensen en su tesis doctoral titulada "Global Illumination using Photon Maps", presentada en 1996 en la Universidad de Washington habla por primera vez de Photon Mapping, un método eficiente para simular la iluminación global, incluyendo efectos como causticas, reflexiones indirectas y refracciones, que son difíciles de lograr con otras técnicas como el path tracing puro.

Multiple Importance Sampling (MIS) fue introducida por Eric Veach en su tesis doctoral titulada "Robust Montecarlo Methods for Light Transport Simulation", presentada en 1997 en la Universidad de Stanford. Esta técnica es una de las contribuciones clave en la mejora del rendimiento de los métodos de Montecarlo aplicados a la simulación de transporte de luz en gráficos por computadora.

Metropolis Light Transport (MLT) fue introducido por primera vez en gráficos por computadora en el artículo titulado "Metropolis Light Transport" presentado en la

conferencia SIGGRAPH en 1997. Este trabajo fue realizado por Eric Veach y Leonidas J. Guibas.

Los 2000

Durante esta época, el CGI se volvió omnipresente. A finales de los años 90 y durante los 2000, la computación gráfica se consolidó en videojuegos, cine y anuncios televisivos. La creciente sofisticación de las GPU fue clave, con la línea Nvidia GeForce dominando el mercado y la competencia de ATI (luego AMD). Hacia el final de la década, incluso las computadoras más económicas incluían GPU capaces de renderizar gráficos 3D.

El CGI en el cine y los videojuegos comenzó a alcanzar niveles de realismo tan altos que se produjo el fenómeno del “Uncanny Valley” o "valle inquietante": nuestros cerebros perciben gráficos próximos al realismo como falsos generando una incomodidad subconsciente. Películas animadas por CGI como Ice Age, Madagascar y Finding Nemo dominaron la taquilla. Final Fantasy: The Spirits Within (2001) fue la primera película en usar personajes CGI fotorrealistas y captura de movimiento, aunque no tuvo éxito en taquilla, en parte por el "Uncanny Valley". The Polar Express y la trilogía de precuelas de Star Wars también elevaron el estándar de los efectos CGI en el cine. [90]

El Subsurface Scatering fue formalizado y presentado de manera significativa en el ámbito de los gráficos por computadora en el artículo de Henrik Wann Jensen, Stephen R. Marschner, y otros en 2001 titulado "A Practical Model for Subsurface Light Transport".

En videojuegos, consolas como PlayStation 2 y 3, Xbox y GameCube mantuvieron una base sólida de seguidores, con títulos como Grand Theft Auto, Assassin's Creed y Final Fantasy mejorando la calidad visual. Los ingresos de la industria de videojuegos comenzaron a compararse con los del cine. Microsoft lanzó el programa XNA para apoyar a desarrolladores independientes, pero no fue exitoso, aunque DirectX siguió siendo popular, al igual que OpenGL.

Doom 3 usa por primera vez sombras volumétricas en 2004 y el shadow mapping se convierte en el estándar del renderizado en tiempo real.

En la computación científica, se desarrolló la técnica GPGPU, que permitió intercambiar grandes cantidades de datos entre la GPU y la CPU, acelerando experimentos en bioinformática y biología molecular, así como aplicaciones matemáticas, visión por computadora y renderizado.

Los 2010

En la década de 2010, el CGI se volvió casi omnipresente en medios visuales. Los gráficos pre-renderizados alcanzaron un gran nivel de fotorrealismo y los gráficos en tiempo real comienzan a implementar modelos PBR dada la mayor capacidad computacional de la tecnología existente.

Las GPUs cada vez ofrecieron más etapas programables en el pipeline: Los shaders se volvieron esenciales para el trabajo avanzado en gráficos.

En el cine, la mayoría de las películas animadas ahora son CGI, aunque pocas buscan el fotorrealismo debido al temor al "Uncanny Valley", optando por un estilo más caricaturesco en 3D.

En videojuegos, consolas como Xbox One, PlayStation 4 y Nintendo Switch dominaron el mercado, todas capaces de gráficos 3D avanzados, mientras que las PC con Windows seguían siendo una de las plataformas más activas para juegos.

Al final de la década los motores de videojuegos más potentes son capaces de renderizar iluminación global sirviéndose de múltiples técnicas, e imitan mediante heurísticas efectos antes solo aptos para renderizado offline. Es una iluminación global muy sesgada, El Ray Marching se populariza para representar niebla y nubes volumétricas. Así como las GPU empiezan a dar soporte a ray tracing por aceleración de hardware, siendo la primera la generación 2000 RTX de Nvidia en 2018, lo que mejora sobre todo los reflejos en los videojuegos, antes obtenidos mediante probes o Screen Space Reflections.

El renderizado offline exploró nuevas líneas de investigación: el renderizado espectral, Path Guiding, renderizado en la nube, la inteligencia artificial y las redes neuronales [91] [92].

Anexo B. Manual de usuario

Este anexo es el manual de usuario para facilitar el uso de la aplicación. Se trata de una aplicación sólo compatible con Windows.

Instalación

No es necesario instalar ninguna aplicación, es un programa portable para Windows. El programa se encuentra en una carpeta llamada Ejecutable y contiene un acceso directo llamado “PFG Alberto Viedma” para acceder a él.

Interfaz

La interfaz de usuario es muy intuitiva y fácil de usar. Para renderizar una imagen, se tiene que escribir el nombre de la escena que se desea usar sin reflejar la extensión del archivo, sólo se permite FBX. Por defecto se deja un modelo de prueba prestablecido Cornell_Box. Además, se puede proponer un nombre para la imagen de salida en la casilla output y elegir las variables de resolución, sobremuestreo, el número de rayos secundarios por choque, la profundidad máxima o la exposición de la imagen deseada.

Ubicación de las imágenes y las escenas

Las escenas e imágenes de salida deben estar en sus respectivas subcarpetas, Models y Output. Asimismo, si la escena contiene texturas, estas deben estar en la subcarpeta textures dentro de models. Se invita al usuario a probar los distintos modelos creados por defecto.

Advertencia de uso

El uso del programa es seguro. La ejecución del renderizado está dispuesta para hacer un uso intensivo de la CPU y es posible que si se está realizando otra tarea en el

ordenador se ralentice. El programa podría llegar a tardar días en renderizar una escena si se configura con una escena muy grande o valores de sobremuestreo y rayos secundarios altos. Los modelos Cornell tardan menos de un minuto en renderizarse en un ordenador de 6 núcleos y 12 hilos con los parámetros por defecto.

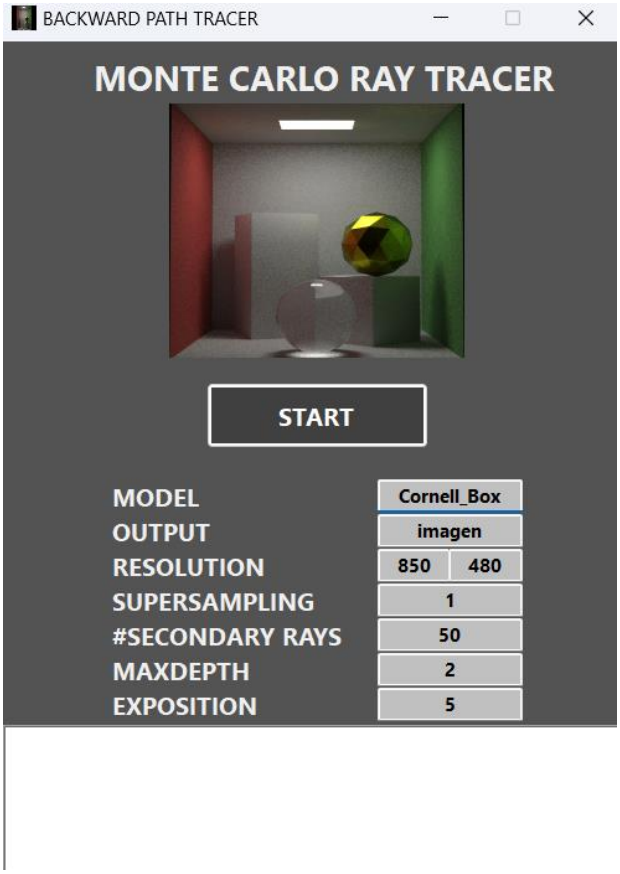

Figura B 1: Imagen de la interfaz.

Archivo de importación

El programa permite el uso de escenas FBX. Si se desea implementar luces de área, el nombre del material del objeto emisivo debe contener la cadena “emissive”. ASSIMP no permite la importación de materiales emisivos de luz por defecto. Los materiales que se desee que cuenten con un índice de refracción 1.4 deben contener la cadena “refractive” en el nombre del material. Para incluir una luz ambiental hay que renombra cual tipo de luz para que contenga la cadena “ambient”.

Ejemplo de uso

Este ejemplo muestra cómo se usa el programa de forma sencilla. Una vez introducido los datos y el modelo a renderizar se pulsa el botón START que cambia a botón CANCEL, puedes cancelar el proceso de renderizado cerrando el programa o pulsando cancel. Una vez terminado el proceso se mostrará en un cuadro cuánto ha tardado cada parte del proceso y se abrirá la imagen generada con el editor por defecto.

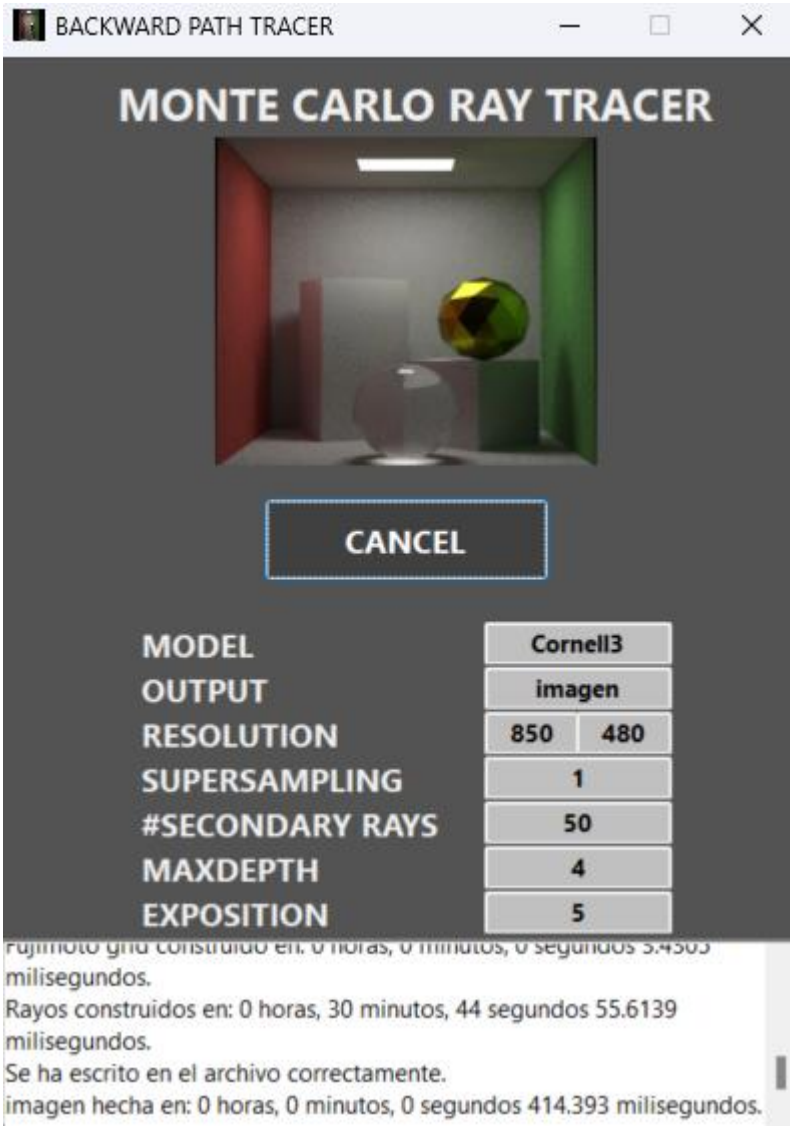

Figura B 2 Inicio de la operación.

La imagen obtenida en este caso es:

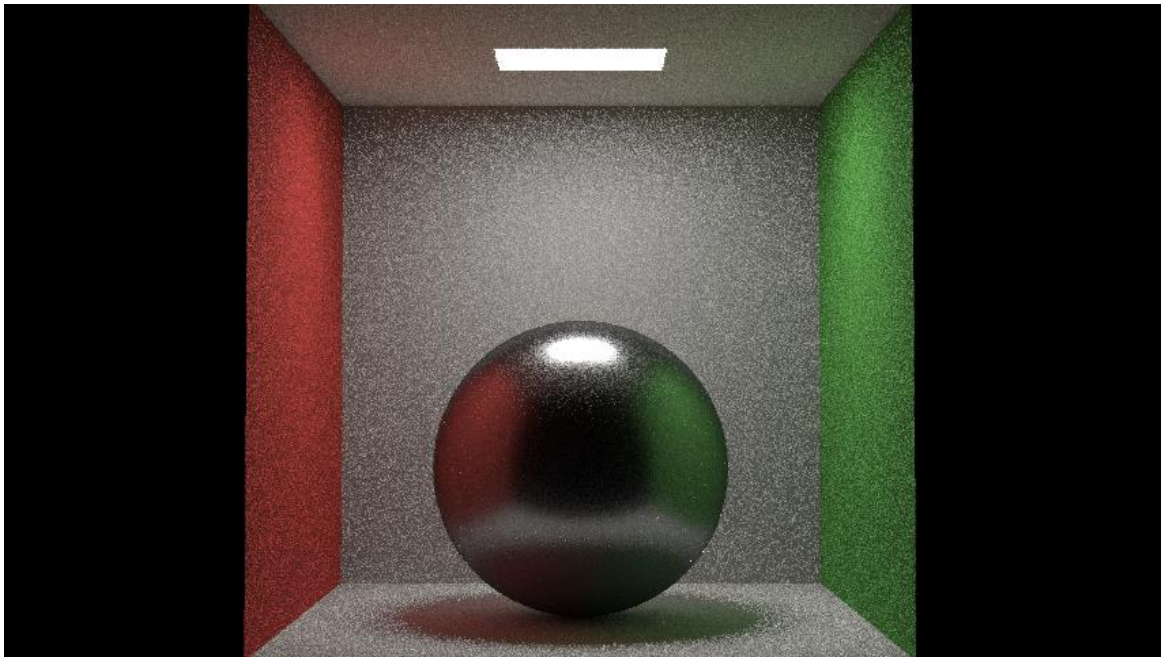

Figura B 3 Imagen generada.

Anexo C. Código fuente

C.1 GUI

```
//Codigo de la única ventana
namespace Interfaz
{
   partial class Ventana
   {
      /// <summary>
      ///  Required designer variable.
      /// </summary>
      private System.ComponentModel.IContainer components = null;

      /// <summary>
      ///  Clean up any resources being used.
      /// </summary>
      /// <param name="disposing">true if managed resources should be disposed;
otherwise, false.</param>
      protected override void Dispose(bool disposing)
      {
         if (disposing && (components != null))
         {
            components.Dispose();
         }
         base.Dispose(disposing);
      }

      #region Windows Form Designer generated code

      /// <summary>
      ///  Required method for Designer support - do not modify
      ///  the contents of this method with the code editor.
      /// </summary>
      private void InitializeComponent()
      {
```

```
        System.ComponentModel.ComponentResourceManager resources = new
System.ComponentModel.ComponentResourceManager(typeof(Ventana));
        label1 = new Label();
        pictureBox1 = new PictureBox();
        label2 = new Label();
        label3 = new Label();
        label4 = new Label();
        label5 = new Label();
        label6 = new Label();
        label7 = new Label();
        label8 = new Label();
        MODEL = new TextBox();
        OUTPUT = new TextBox();
        WIDTH = new TextBox();
        HEIGHT = new TextBox();
        SUPERSAMPLING = new TextBox();
        NSECOND = new TextBox();
        MAXDEPTH = new TextBox();
        EXPOSITION = new TextBox();
        botonstart = new Button();
        proceso = new System.Diagnostics.Process();
        richTextBox1 = new RichTextBox();
        ((System.ComponentModel.ISupportInitialize)pictureBox1).BeginInit();
        SuspendLayout();
        //
        // label1
        //
        label1.AutoSize = true;
        label1.Font = new Font("Segoe UI", 16.2F, FontStyle.Bold, GraphicsUnit.Point, 0);
        label1.ForeColor = SystemColors.ButtonFace;
        label1.Location = new Point(68, 9);
        label1.Name = "label1";
        label1.Size = new Size(366, 38);
        label1.TabIndex = 0;
        label1.Text = "MONTE CARLO RAY TRACER";
        //
        // pictureBox1
        //
        pictureBox1.Image = Properties.Resources.Portada;
        pictureBox1.Location = new Point(136, 50);
        pictureBox1.Name = "pictureBox1";
        pictureBox1.Size = new Size(238, 205);
        pictureBox1.SizeMode = PictureBoxSizeMode.StretchImage;
        pictureBox1.TabIndex = 1;
        pictureBox1.TabStop = false;
        //
        // label2
        //
        label2.AutoSize = true;
```

```
label2.Font = new Font("Segoe UI", 12F, FontStyle.Bold, GraphicsUnit.Point, 0);
label2.ForeColor = SystemColors.ButtonFace;
label2.Location = new Point(85, 352);
label2.Name = "label2";
label2.Size = new Size(82, 28);
label2.TabIndex = 2;
label2.Text = "MODEL";
//
// label3
//
label3.AutoSize = true;
label3.Font = new Font("Segoe UI", 12F, FontStyle.Bold, GraphicsUnit.Point, 0);
label3.ForeColor = SystemColors.ButtonFace;
label3.Location = new Point(85, 380);
label3.Name = "label3";
label3.Size = new Size(91, 28);
label3.TabIndex = 3;
label3.Text = "OUTPUT";
//
// label4
//
label4.AutoSize = true;
label4.Font = new Font("Segoe UI", 12F, FontStyle.Bold, GraphicsUnit.Point, 0);
label4.ForeColor = SystemColors.ButtonFace;
label4.Location = new Point(85, 408);
label4.Name = "label4";
label4.Size = new Size(135, 28);
label4.TabIndex = 4;
label4.Text = "RESOLUTION";
//
// label5
//
label5.AutoSize = true;
label5.Font = new Font("Segoe UI", 12F, FontStyle.Bold, GraphicsUnit.Point, 0);
label5.ForeColor = SystemColors.ButtonFace;
label5.Location = new Point(85, 436);
label5.Name = "label5";
label5.Size = new Size(175, 28);
label5.TabIndex = 5;
label5.Text = "SUPERSAMPLING";
//
// label6
//
label6.AutoSize = true;
label6.Font = new Font("Segoe UI", 12F, FontStyle.Bold, GraphicsUnit.Point, 0);
label6.ForeColor = SystemColors.ButtonFace;
label6.Location = new Point(85, 464);
label6.Name = "label6";
label6.Size = new Size(197, 28);
```

```
label6.TabIndex = 6;
label6.Text = "#SECONDARY RAYS";
//
// label7
//
label7.AutoSize = true;
label7.Font = new Font("Segoe UI", 12F, FontStyle.Bold, GraphicsUnit.Point, 0);
label7.ForeColor = SystemColors.ButtonFace;
label7.Location = new Point(85, 492);
label7.Name = "label7";
label7.Size = new Size(123, 28);
label7.TabIndex = 7;
label7.Text = "MAXDEPTH";
//
// label8
//
label8.AutoSize = true;
label8.Font = new Font("Segoe UI", 12F, FontStyle.Bold, GraphicsUnit.Point, 0);
label8.ForeColor = SystemColors.ButtonFace;
label8.Location = new Point(85, 520);
label8.Name = "label8";
label8.Size = new Size(129, 28);
label8.TabIndex = 8;
label8.Text = "EXPOSITION";
//
// MODEL
//
MODEL.BackColor = Color.Silver;
MODEL.Font = new Font("Segoe UI", 9F, FontStyle.Bold);
MODEL.Location = new Point(304, 353);
MODEL.Name = "MODEL";
MODEL.Size = new Size(117, 27);
MODEL.TabIndex = 9;
MODEL.Text = "Cornell_Box";
MODEL.TextAlign = HorizontalAlignment.Center;
//
// OUTPUT
//
OUTPUT.BackColor = Color.Silver;
OUTPUT.Font = new Font("Segoe UI", 9F, FontStyle.Bold);
OUTPUT.Location = new Point(304, 381);
OUTPUT.Name = "OUTPUT";
OUTPUT.Size = new Size(117, 27);
OUTPUT.TabIndex = 10;
OUTPUT.Text = "imagen";
OUTPUT.TextAlign = HorizontalAlignment.Center;
//
// WIDTH
//
```

```
WIDTH.BackColor = Color.Silver;
WIDTH.Font = new Font("Segoe UI", 9F, FontStyle.Bold);
WIDTH.Location = new Point(304, 409);
WIDTH.Name = "WIDTH";
WIDTH.Size = new Size(60, 27);
WIDTH.TabIndex = 11;
WIDTH.Text = "850";
WIDTH.TextAlign = HorizontalAlignment.Center;
WIDTH.TextChanged += WIDTH_TextChanged;
//
// HEIGHT
//
HEIGHT.BackColor = Color.Silver;
HEIGHT.Font = new Font("Segoe UI", 9F, FontStyle.Bold);
HEIGHT.Location = new Point(361, 409);
HEIGHT.Name = "HEIGHT";
HEIGHT.Size = new Size(60, 27);
HEIGHT.TabIndex = 12;
HEIGHT.Text = "480";
HEIGHT.TextAlign = HorizontalAlignment.Center;
HEIGHT.TextChanged += HEIGHT_TextChanged;
//
// SUPERSAMPLING
//
SUPERSAMPLING.BackColor = Color.Silver;
SUPERSAMPLING.Font = new Font("Segoe UI", 9F, FontStyle.Bold);
SUPERSAMPLING.Location = new Point(304, 437);
SUPERSAMPLING.Name = "SUPERSAMPLING";
SUPERSAMPLING.Size = new Size(117, 27);
SUPERSAMPLING.TabIndex = 13;
SUPERSAMPLING.Text = "1";
SUPERSAMPLING.TextAlign = HorizontalAlignment.Center;
SUPERSAMPLING.TextChanged += SUPERSAMPLING_TextChanged;
//
// NSECOND
//
NSECOND.BackColor = Color.Silver;
NSECOND.Font = new Font("Segoe UI", 9F, FontStyle.Bold);
NSECOND.Location = new Point(304, 465);
NSECOND.Name = "NSECOND";
NSECOND.Size = new Size(117, 27);
NSECOND.TabIndex = 14;
NSECOND.Text = "50";
NSECOND.TextAlign = HorizontalAlignment.Center;
NSECOND.TextChanged += NSECOND_TextChanged;
//
// MAXDEPTH
//
MAXDEPTH.BackColor = Color.Silver;
```

```
        MAXDEPTH.Font = new Font("Segoe UI", 9F, FontStyle.Bold);
        MAXDEPTH.Location = new Point(304, 493);
        MAXDEPTH.Name = "MAXDEPTH";
        MAXDEPTH.Size = new Size(117, 27);
        MAXDEPTH.TabIndex = 15;
        MAXDEPTH.Text = "2";
        MAXDEPTH.TextAlign = HorizontalAlignment.Center;
        MAXDEPTH.TextChanged += MAXDEPTH_TextChanged;
        //
        // EXPOSITION
        //
        EXPOSITION.BackColor = Color.Silver;
        EXPOSITION.Font = new Font("Segoe UI", 9F, FontStyle.Bold);
        EXPOSITION.Location = new Point(304, 521);
        EXPOSITION.Name = "EXPOSITION";
        EXPOSITION.Size = new Size(117, 27);
        EXPOSITION.TabIndex = 16;
        EXPOSITION.Text = "5";
        EXPOSITION.TextAlign = HorizontalAlignment.Center;
        EXPOSITION.TextChanged += EXPOSITION_TextChanged;
        //
        // botonstart
        //
        botonstart.BackColor = Color.FromArgb(64, 64, 64);
        botonstart.Font = new Font("Segoe UI", 12F, FontStyle.Bold, GraphicsUnit.Point,
0);
        botonstart.ForeColor = Color.White;
        botonstart.Location = new Point(166, 275);
        botonstart.Name = "botonstart";
        botonstart.Size = new Size(179, 53);
        botonstart.TabIndex = 17;
        botonstart.Text = "START";
        botonstart.UseVisualStyleBackColor = false;
        botonstart.Click += BOTÓNSTART_Click;
        //
        // proceso
        //
        proceso.StartInfo.Domain = "";
        proceso.StartInfo.LoadUserProfile = false;
        proceso.StartInfo.Password = null;
        proceso.StartInfo.StandardErrorEncoding = null;
        proceso.StartInfo.StandardInputEncoding = null;
        proceso.StartInfo.StandardOutputEncoding = null;
        proceso.StartInfo.UseCredentialsForNetworkingOnly = false;
        proceso.StartInfo.UserName = "";
        proceso.SynchronizingObject = this;
        proceso.Exited += process1_Exited;
        //
        // richTextBox1
```

```
    //
    richTextBox1.Location = new Point(2, 551);
    richTextBox1.Name = "richTextBox1";
    richTextBox1.Size = new Size(499, 121);
    richTextBox1.TabIndex = 18;
    richTextBox1.Text = "";
    //
    // Ventana
    //
    AutoScaleDimensions = new SizeF(8F, 20F);
    AutoScaleMode = AutoScaleMode.Font;
    BackColor = Color.FromArgb(83, 83, 83);
    ClientSize = new Size(502, 671);
    Controls.Add(richTextBox1);
    Controls.Add(botonstart);
    Controls.Add(EXPOSITION);
    Controls.Add(MAXDEPTH);
    Controls.Add(NSECOND);
    Controls.Add(SUPERSAMPLING);
    Controls.Add(HEIGHT);
    Controls.Add(WIDTH);
    Controls.Add(OUTPUT);
    Controls.Add(MODEL);
    Controls.Add(label8);
    Controls.Add(label7);
    Controls.Add(label6);
    Controls.Add(label5);
    Controls.Add(label4);
    Controls.Add(label3);
    Controls.Add(label2);
    Controls.Add(pictureBox1);
    Controls.Add(label1);
    FormBorderStyle = FormBorderStyle.FixedSingle;
    Icon = (Icon)resources.GetObject("$this.Icon");
    MaximizeBox = false;
    Name = "Ventana";
    StartPosition = FormStartPosition.CenterScreen;
    Text = "BACKWARD PATH TRACER";
    FormClosed += Form1_FormClosed;
    ((System.ComponentModel.ISupportInitialize)pictureBox1).EndInit();
    ResumeLayout(false);
    PerformLayout();
}

#endregion

private Label label1;
private PictureBox pictureBox1;
private Label label2;
```

```
        private Label label3;
        private Label label4;
        private Label label5;
        private Label label6;
        private Label label7;
        private Label label8;
        private TextBox MODEL;
        private TextBox OUTPUT;
        private TextBox WIDTH;
        private TextBox HEIGHT;
        private TextBox SUPERSAMPLING;
        private TextBox NSECOND;
        private TextBox MAXDEPTH;
        private TextBox EXPOSITION;
        public Button botonstart;
        private System.Diagnostics.Process proceso;
        private RichTextBox richTextBox1;
    }
}
/*****************************************************************************/

namespace Interfaz
{

    public partial class Ventana : Form
    {

        private bool iniciado;
        public Ventana()
        {
            proceso = new Process();
            InitializeComponent();
        }

        public async void BOTÓNSTART_Click(object sender, EventArgs e)
        {

            // Verifica si ya hay un proceso en ejecución
            if (!iniciado)
            {
                // Especifica la ruta de tu programa
                proceso.StartInfo.FileName = @"MonteCarloRayTracer.exe";


                // Definir 8 argumentos
                string arg1 = WIDTH.Text;
                string arg2 = HEIGHT.Text;
                string arg3 = EXPOSITION.Text;
                string arg4 = MAXDEPTH.Text;
```

```
        string arg5 = SUPERSAMPLING.Text;
        string arg6 = NSECOND.Text;
        string arg7 = "output/" + OUTPUT.Text + ".ppm";
        string arg8 = "models/" + MODEL.Text + ".fbx";

        // Pasar los 8 argumentos al proceso
        proceso.StartInfo.Arguments = $" {arg1} {arg2} {arg3} {arg4} {arg5} {arg6}
{arg7} {arg8}";

        proceso.EnableRaisingEvents = true;  // Habilitar el evento Exited

        // Suscribirse al evento Exited
        // Configurar el nombre del archivo del proceso (por ejemplo, "cmd.exe" o
cualquier ejecutable de consola)
        // Configurar el nombre del archivo del proceso (por ejemplo, "cmd.exe" o
cualquier ejecutable de consola)

        // Argumentos opcionales (en este caso, el comando dir)
        //proceso.StartInfo.Arguments = "/C dir";

        // Redirigir la salida estándar de la consola
        proceso.StartInfo.RedirectStandardOutput = true;
        proceso.StartInfo.UseShellExecute = false;
        proceso.StartInfo.CreateNoWindow = true;  // No mostrar ventana de consola


        string output = "Comenzando el trazado de rayos...";

        // Mostrar la salida en RichTextBox de forma progresiva
        richTextBox1.AppendText(output + Environment.NewLine);

        //Desplazar el scroll hacia abajo si quieres que siga la salida
        richTextBox1.ScrollToCaret();


        botonstart.Text = "CANCEL";
        iniciado = true;
        // Iniciar el proceso
        proceso.Start();

        // Cambiar el texto del botón para indicar que el programa está en ejecución


        // Leer la salida estándar de forma asíncrona línea por línea


        output = await proceso.StandardOutput.ReadToEndAsync();

        // Mostrar la salida en RichTextBox de forma progresiva
```

```
            richTextBox1.AppendText(output + Environment.NewLine);

            //Desplazar el scroll hacia abajo si quieres que siga la salida
            richTextBox1.ScrollToCaret();


            // Esperar a que el proceso termine
            await proceso.WaitForExitAsync();
        }
        else
        {
            // Si el proceso está en ejecución, lo terminamos
            proceso.Kill();
            proceso.WaitForExit();

            // Cambiar el texto del botón para indicar que el programa no está en
ejecución
            botonstart.Text = "START";
            iniciado = false;
        }
    }
    // Método que se llama cuando el proceso ha terminado

    private void process1_Exited(object sender, EventArgs e)
    {
        //MessageBox.Show("El proceso ha terminado.");
        BOTÓNSTART_Click(sender, e);
    }

    private void WIDTH_TextChanged(object sender, EventArgs e)
    {

        double numero;
        int entero;
        bool esNumero = double.TryParse(WIDTH.Text, out numero);
        if (!esNumero) WIDTH.Text = "850";
        entero = (int)numero;
        WIDTH.Text = entero.ToString(CultureInfo.InvariantCulture);
        if (entero < 1) WIDTH.Text = "850";
    }

    private void HEIGHT_TextChanged(object sender, EventArgs e)
    {
        double numero;
        int entero;
        bool esNumero = double.TryParse(HEIGHT.Text, out numero);
        if (!esNumero) HEIGHT.Text = "480";
        entero = (int)numero;
        HEIGHT.Text = entero.ToString(CultureInfo.InvariantCulture);
```

```
    if (entero < 1) HEIGHT.Text = "480";
}

private void SUPERSAMPLING_TextChanged(object sender, EventArgs e)
{
    double numero;
    int entero;
    bool esNumero = double.TryParse(SUPERSAMPLING.Text, out numero);
    if (!esNumero) SUPERSAMPLING.Text = "1";
    entero = (int)numero;
    SUPERSAMPLING.Text = entero.ToString(CultureInfo.InvariantCulture);
    if (entero < 1) SUPERSAMPLING.Text = "1";
}

private void NSECOND_TextChanged(object sender, EventArgs e)
{
    double numero;
    int entero;
    bool esNumero = double.TryParse(NSECOND.Text, out numero);
    if (!esNumero) NSECOND.Text = "50";
    entero = (int)numero;
    NSECOND.Text = entero.ToString(CultureInfo.InvariantCulture);
    if (entero < 1) NSECOND.Text = "1";
    if (entero == 1) label1.Text = "BACKWARD PATH TRACER";
    else label1.Text = "MONTE CARLO RAY TRACER";

}

private void MAXDEPTH_TextChanged(object sender, EventArgs e)
{
    double numero;
    int entero;
    bool esNumero = double.TryParse(MAXDEPTH.Text, out numero);
    if (!esNumero) MAXDEPTH.Text = "2";
    entero = (int)numero;
    MAXDEPTH.Text = entero.ToString(CultureInfo.InvariantCulture);
    if (entero < 0) MAXDEPTH.Text ="0";
}

private void EXPOSITION_TextChanged(object sender, EventArgs e)
{
    double numero;
    bool esNumero = double.TryParse(EXPOSITION.Text, out numero);
    if (!esNumero) EXPOSITION.Text = "5";
    if (numero < 0) EXPOSITION.Text = "5";
}
private void Form1_FormClosed(object sender, FormClosedEventArgs e)
{
    if (iniciado)
```

```
        {
          proceso.Kill();
          proceso.WaitForExit();
        }
      }
    }
}
/*****************************************************************************/
```

C.2 Ray tracer

```
/*****************************************************************************/
/*********************************Modulo Render*******************************/
/*****************************************************************************/

int main(int argc, char* argv[]) {


   //variables de la aplicación
   int widthin = 850;
   int heightin = 480;
   float expositionin = 5;
   int maxDepthin = 2;
   int supersamplingin = 1;
   int numSecondaryRaysin = 50;
   std::string archiveNamein = "output\\imagen.ppm";
   std::string modelin = "models\\Cornell_Box.fbx";
   std::string filepath = argv[0];
   modelin = filepath.substr(0, filepath.find_last_of("/\\")) + "\\" + modelin;
   archiveNamein = filepath.substr(0, filepath.find_last_of("/\\")) + "\\" +
archiveNamein;
   if (argc == 9) {

      widthin = atoi(argv[1]);
      heightin = atoi(argv[2]);
      expositionin = std::stof(argv[3]);
      maxDepthin = atoi(argv[4]);
      supersamplingin = atoi(argv[5]);
      numSecondaryRaysin = atoi(argv[6]);
      archiveNamein = argv[7];
      modelin = argv[8];

   }
   // Imagen
   std::string archiveName = archiveNamein;
   //Pantalla
```

```
    int width = widthin;
    int height = heightin;
    exposition = expositionin; //exposicion de la escena
    maxDepth = maxDepthin; //PROFUNDIDAD MAXIMA de rayos
    int supersampling = supersamplingin; //rayos por pixel
    int numSecondaryRays = numSecondaryRaysin; //numero de rayos secundarios
    std::string modelName = modelin;



    std::cout << "Cargando: " << modelName << std::endl;

    //cronometro
    auto timeStart = std::chrono::high_resolution_clock::now();

    //poison points
    PoissonGenerator::DefaultPRNG PRNG; //generador de puntos poisson
    const auto poissonPoints =
PoissonGenerator::generatePoissonPoints(supersampling, PRNG);
    supersampling = poissonPoints.size();

    auto timeEnd = std::chrono::high_resolution_clock::now();
    auto passedTime = std::chrono::duration<double, std::milli>(timeEnd -
timeStart).count();
    std::cout << "Poisson grid construidos en: " << (int)(passedTime / 3600000) << "
horas, " << (int)(passedTime / 60000) % 60 << " minutos, " << (int)(passedTime /
1000) % 60 << " segundos " << (passedTime/1000-(int)(passedTime/1000))*1000 << "
milisegundos." << "\n";
    timeStart = std::chrono::high_resolution_clock::now();

    Camera camera = Camera(55.5f, width, height);

    //scene meshes
    Scene* scene = new Scene(modelName.c_str());
    //obtencion de la posicion de la camara de la escena
    scene->setCamera(camera);

    //luces
    std::vector<Light*> lights;
    //lights.push_back(new Distantlight(camera.forward));
    for (int l = 0; l < scene->getnLights(); l++) {
        Light* light = scene->getLight(l);
      if(light!=nullptr)lights.push_back(light);
    }

    //Búfer de rayos
    int size = width * height * supersampling;
    Ray* raybuffer = new Ray[size];
```

```
   timeEnd = std::chrono::high_resolution_clock::now();
   passedTime = std::chrono::duration<double, std::milli>(timeEnd - timeStart).count();
   std::cout << "Escena construida en: " << (int)(passedTime / 3600000) << " horas, "
<< (int)(passedTime / 60000) % 60 << " minutos, " << (int)(passedTime / 1000) % 60
<< " segundos " << (passedTime / 1000 - (int)(passedTime / 1000)) * 1000 << "
milisegundos." << "\n";
   timeStart = std::chrono::high_resolution_clock::now();

   //creación de la estructura de datos para el next estimation event
   NEE* nee = new NEE();

   for (int mesh = 0; mesh < scene->getNumMeshes(); mesh++) {
      for (int face = 0; face < scene->getMeshNumFaces(mesh); face++) {
         nee->addTriangle(scene->getTriangle(mesh,face));
      }
   }

   timeEnd = std::chrono::high_resolution_clock::now();
   passedTime = std::chrono::duration<double, std::milli>(timeEnd - timeStart).count();
   std::cout << "NextEventEstimation grid construido en: " << (int)(passedTime /
3600000) << " horas, " << (int)(passedTime / 60000) % 60 << " minutos, " <<
(int)(passedTime / 1000) % 60 << " segundos " << (passedTime / 1000 -
(int)(passedTime / 1000)) * 1000 << " milisegundos." << "\n";
   timeStart = std::chrono::high_resolution_clock::now();

   //Grid de fujimoto
   Grid* grid = new Grid(scene);

   timeEnd = std::chrono::high_resolution_clock::now();
   passedTime = std::chrono::duration<double, std::milli>(timeEnd - timeStart).count();
   std::cout << "Fujimoto grid construido en: " << (int)(passedTime / 3600000) << "
horas, " << (int)(passedTime / 60000) % 60 << " minutos, " << (int)(passedTime /
1000) % 60 << " segundos " << (passedTime / 1000 - (int)(passedTime / 1000)) * 1000
<< " milisegundos." << "\n";
   timeStart = std::chrono::high_resolution_clock::now();

      //definimos cada rayo y lo lanzamos
      #pragma omp parallel for collapse(3)
      for (int j = 0; j < height; j++) {
         for (int i = 0; i < width; i++) {
            for (unsigned int s = 0; s < supersampling; s++) {

               //espacio de camara de -1 a 1
               float x = (1 - 2 * (i + poissonPoints[s].x) / (float)width); //en el centro de
cada pixel, la pantalla está definida en (-1,-1,0)(-1,1,0)(1,-1,0)(1,1,0)
               float y = (1 - 2 * (j + poissonPoints[s].y) / (float)height);
               //mundo, se definen origen y direccion de los rayos de la camara
```

```
            raybuffer[j * width * supersampling + i * supersampling +
s].setDirection(camera.setRayDirection(x, y, raybuffer[j * width * supersampling + i *
supersampling + s].origin));
            //rayos
            render(raybuffer[j * width * supersampling + i * supersampling + s], *grid,
lights, *nee, maxDepth, numSecondaryRays);
          }
        }
      }

   timeEnd = std::chrono::high_resolution_clock::now();
   passedTime = std::chrono::duration<double, std::milli>(timeEnd - timeStart).count();
   std::cout << "Rayos construidos en: " << (int)(passedTime / 3600000) << " horas, "
<< (int)(passedTime / 60000) % 60 << " minutos, " << (int)(passedTime / 1000) % 60
<< " segundos " << (passedTime / 1000 - (int)(passedTime / 1000)) * 1000 << "
milisegundos." << "\n";
   timeStart = std::chrono::high_resolution_clock::now();

   //Escritura del archivo PPM
   //Se abre el archivo o se crea y se elimina el contenido
   std::ofstream archivo(archiveName, std::ios::trunc);

   // Verificar si el archivo se abrió correctamente
   if (archivo.is_open()) {

      // Escribir en el archivo
      archivo << "P3\n" << width << ' ' << height << "\n255\n";

      float ir = 0.0f, ig = 0.0f, ib = 0.0f;
      for (unsigned int j = 0; j < height; ++j) {
         for (unsigned int i = 0; i < width; ++i) {
            ir = 0.0f, ig = 0.0f, ib = 0.0f;

            for (int s = 0; s < supersampling; s++) {

               ir += raybuffer[s+i*supersampling+j*width*supersampling].color.x ;
               ig += raybuffer[s+i*supersampling+j*width*supersampling].color.y;
               ib += raybuffer[s+i*supersampling+j*width*supersampling].color.z;

            }
            //correccion gamma
            ir = glm::clamp(glm::pow(ir*exposition / (float)supersampling,
1.0f/2.2f),0.0f,1.0f)*255.0f ;
            ig = glm::clamp(glm::pow(ig*exposition / (float)supersampling,
1.0f/2.2f),0.0f,1.0f)*255.0f ;
            ib = glm::clamp(glm::pow(ib*exposition / (float)supersampling,
1.0f/2.2f),0.0f,1.0f)*255.0f ;
```

```
            archivo << static_cast<unsigned int>(ir) << ' ' << static_cast<unsigned int>(ig)
<< ' ' << static_cast<unsigned int>(ib) << '\n';
          }
        }
        //Cerrar el archivo cuando hayamos terminado de escribir en él
        archivo.close();
        std::cout << "Se ha escrito en el archivo correctamente.\n";
    }
    else {
        std::cerr << "No se pudo abrir el archivo.\n";
    }

    timeEnd = std::chrono::high_resolution_clock::now();
    passedTime = std::chrono::duration<double, std::milli>(timeEnd - timeStart).count();
    std::cout << "imagen hecha en: " << (int)(passedTime / 3600000) << " horas, " <<
(int)(passedTime / 60000) % 60 << " minutos, " << (int)(passedTime / 1000) % 60 << "
segundos " << (passedTime / 1000 - (int)(passedTime / 1000)) * 1000 << "
milisegundos." << "\n";
    timeStart = std::chrono::high_resolution_clock::now();


    //Borrado de instancias
    for (int i = 0; i < lights.size(); i++) delete lights[i];
    delete grid;
    delete nee;
    delete scene;
    delete[] raybuffer;

    // Comando para abrir la imagen con el programa predeterminado en Windows
    std::string comando = "start " + std::string(archiveName);
    // Ejecutar el comando para abrir la imagen
    system(comando.c_str());

    return 0;
}
/*****************************************************************************/

//Funcion recursiva rayos primarios
static void render(Ray& ray, Grid& grid, std::vector<Light*>& lights, NEE& nee, int
depth, int nsecond) {

    std::shared_ptr<Triangle> triangle = nullptr; //triangulo del choque
    glm::vec2 uv; //coordenadas baricentricas del choque

    triangle = grid.DDAalgorithm(ray, uv);//calculo del triangulo intersecado
    if (ray.hitDistance < 0.0f) return;

    Hit hit;
```

```
   getSurfaceProperties(ray, triangle.get(), hit, uv);//obtencion de las propiedades del
choque

   //choque de luz
   if (hit.emissive != glm::vec3(0)) {
      ray.color = hit.emissive;
      return;
   }

   #pragma omp parallel for
   for (int l = 0; l < lights.size(); l++) {
      lightHit(ray, lights[l], hit, grid);
   }

   //espejo
   if (hit.metalness == 1.0 && hit.roughness == 0.0) {
      Ray reflection = ray.createReflectionRay(hit.normal);
      renderSecond(reflection,ray, hit, grid, lights, nee, depth, nsecond);
      ray.color += reflection.color;
      return;
   }

   //refraccion snell + fresnel
   if (hit.opacity < 1.0f) {
      Ray transmission = ray.createTransmissionRay(hit.normal, hit.refractiveIndex);
      Ray reflection = ray.createReflectionRay(hit.normal);
      float fresnel = ray.fresnel(hit.normal, hit.refractiveIndex);
      #pragma omp parallel
      {
         #pragma omp single
         {
            #pragma omp taskwait
            {
               #pragma omp task
               {
                  if (fresnel < 1.0f) render(transmission, grid, lights, nee, depth, nsecond);
               }
               #pragma omp task
               {
                  if (fresnel > 0.001f) renderSecond(reflection, ray, hit, grid, lights, nee,
depth-1, nsecond);
               }
            }
         }
      }
      ray.color += (1.0f - hit.opacity) * ((1.0f - fresnel) * transmission.color +
(fresnel)*reflection.color);
      if (hit.opacity == 0.0f)return;
   }
```

```
   //direct montecarlo next even estimation
   //emitimos rayos a los triangulos emisivos si dan aplicamos la pdf

   if (glm::dot(ray.direction, glm::vec3(hit.normal)) > 0.0f) hit.normal = -hit.normal;
//ve las dos caras

   if (nsecond > 0 && depth > 0) {


      glm::vec3 directColor = glm::vec3(0.0f);

      glm::vec3* directColors = new glm::vec3[nee.size()];
      #pragma omp parallel for shared(directColor)
      for (int light = 0; light < nee.size(); light++) {
         directColors[light] = glm::vec3(0);
         glm::vec3& dcolor = directColors[light];
         int N = std::max(nee.areaLights[light].N /
std::max(std::pow(glm::length(nee.areaLights[light].center - ray.hitPoint) / 100.0f,
2.0f), 1.0f), 1.0f);//disminuyo el numero de puntos cuanto más lejos
         #pragma omp parallel for shared(dcolor)
         for (int point = 0; point <N; point++) {
            //generacion de numeros aleatorios para disparar un rayo al triangulo
iluminado
            float u = rand() % 1000 / 1000.0f, v = rand() % 1000 / 1000.0f, w;
            if (u + v > 1) {
               u = 1 - u;
               v = 1 - v;
            }
            w = 1 - u - v;
            glm::vec3 newpoint = (u * nee.areaLights[light].vertices[0] + v *
nee.areaLights[light].vertices[1] + w * nee.areaLights[light].vertices[2]);
            glm::vec3 direction = glm::normalize(newpoint - ray.hitPoint);
            glm::vec3 origin = ray.hitPoint + direction * 1e-2f;
            glm::vec2 uv;
            Ray ray2(direction, origin);
            std::shared_ptr<Triangle> triangle2 = grid.DDAalgorithm3(ray2, uv);
            //si hay hit se calcula su contribución de luz
            if (nee.emissivetriangles[light] == triangle2) {
               glm::vec3 N = *nee.emissivetriangles[light]->N[0] * (1.0f - uv.x - uv.y) +
*nee.emissivetriangles[light]->N[1] * uv.x + *nee.emissivetriangles[light]->N[2] * uv.y;
               float pdf = std::max(std::pow(ray2.hitDistance, 2.0f), 1.0f) /
(abs(glm::dot(ray2.direction, N)) * nee.emissivetriangles[light]->area);
               pdf = std::max(pdfmin, pdf);
               ray2.color = nee.emissivetriangles[light]->material->emissive;
               dcolor += indirectColorHit(ray, ray2, hit) / (pdf);
            }
         }
         directColor += directColors[light] / (float)(N);
```

```
    }
    delete[] directColors;

    depth -= 1;
    glm::vec3 indirectColor = glm::vec3(0.0);

    //indirect montecarlo rays renderizado
    Ray* indirectRays = new Ray[nsecond];
    glm::vec3* indirectColors = new glm::vec3[nsecond];

    #pragma omp parallel for
    for (int i = 0; i < nsecond; i++) {
      Triangle* triangle2 = nullptr;
      float pdf;
      indirectRays[i] = ray.randomRay(rand() % 10000 / 10000.0f, rand() % 10000 /
10000.0f, hit,pdf);
      renderSecond(indirectRays[i], ray, hit, grid, lights, nee, depth, nsecond);
      pdf = std::max(pdfmin, pdf);
      indirectColors[i] = indirectColorHit(ray, indirectRays[i], hit)/pdf;
    }
    delete[] indirectRays;
    for (int i = 0; i < nsecond; i++) {
      indirectColor += indirectColors[i];
    }
    delete[] indirectColors;

    float interpolation = std::min(std::sqrt((1.0f-
hit.metalness)*glm::length(directColor)*exposition),0.8f);
    ray.color += (directColor * (interpolation)) + indirectColor *(1.0f- interpolation) /
((float(nsecond)));
  }
  return;
}
/*****************************************************************************/
//funcion recursiva rayos secundarios
static void renderSecond(Ray& ray, const Ray& rayBefore, const Hit& hitBefore, Grid&
grid, std::vector<Light*>& lights, NEE& nee, int depth, int nsecond) {

  std::shared_ptr<Triangle> triangle = nullptr;//triangulo del choque
  glm::vec2 uv; //coordenadas baricentricas del choque

  triangle = grid.DDAalgorithm3(ray, uv);//obtención del triangulo del choque
  if (ray.hitDistance < 0.0f) return;

  Hit hit;
  getSurfaceProperties(ray, triangle.get(), hit, uv); //obtención de datos del hit

  //choque de luz
  if (hit.emissive != glm::vec3(0)) {
```

```
        ray.color = hit.emissive;
        return;
    }

    //Shadow rays e iluminación directa
    #pragma omp parallel for
    for (int l = 0; l < lights.size(); l++) {
        lightHit(ray, lights[l], hit, grid);
    }

    if (depth <= 0) return;

    //espejo
    if (hit.metalness == 1.0 && hit.roughness == 0.0) {
        Ray reflection = ray.createReflectionRay(hit.normal);
        renderSecond(reflection, ray, hit, grid, lights, nee, depth - 1, nsecond);
        ray.color += reflection.color;
    }

    //refraccion snell + fresnel
    if (hit.opacity < 1.0f) {
        Ray transmission = ray.createTransmissionRay(hit.normal, hit.refractiveIndex);
        Ray reflection = ray.createReflectionRay(hit.normal);
        float fresnel = ray.fresnel(hit.normal, hit.refractiveIndex);
        #pragma omp parallel
        {
            #pragma omp single
            {
                #pragma omp taskwait
                {
                    #pragma omp task
                    {
                        if (fresnel < 1.0f) renderSecond(transmission, ray, hit, grid, lights, nee,
depth-1, nsecond);
                    }
                    #pragma omp task
                    {
                        if (fresnel > 0.001f) renderSecond(reflection, ray, hit, grid, lights, nee,
depth - 1, nsecond);
                    }
                }
            }
        }
        ray.color += (1.0f - hit.opacity) * ((1.0f - fresnel) * transmission.color +
(fresnel)*reflection.color);
        if (hit.opacity == 0.0f)return;
    }

    //direct montecarlo next even estimation
```

```
   //emitimos rayos a los triangulos emisivos si dan aplicamos la pdf
   if (glm::dot(ray.direction, glm::vec3(hit.normal)) > 0.0f) hit.normal = -hit.normal;
//ve las dos caras
   if (nsecond>13)nsecond = std::min(nsecond * 0.5f, 13.0f);
   depth -= 1;

   if (nsecond > 0) {

      glm::vec3 directColor = glm::vec3(0.0f);
      glm::vec3* directColors = new glm::vec3[nee.size()];
      #pragma omp parallel for shared(directColor)
      for (int light = 0; light < nee.size(); light++) {
         directColors[light] = glm::vec3(0);
         int N = std::max(nee.areaLights[light].N /
std::max(std::pow(glm::length(nee.areaLights[light].center - ray.hitPoint) / 100.0f,
2.0f), 1.0f), 1.0f);//disminuyo el numero de puntos cuanto más lejos
         glm::vec3& dcolor = directColors[light];
         #pragma omp parallel for shared(dcolor)
         for (int point = 0; point < N; point++) {
            //generacion de numeros aleatorios para disparar un rayo al triangulo
iluminado
            float u = rand() % 1000 / 1000.0f, v = rand() % 1000 / 1000.0f, w;
            if (u + v > 1) {
               u = 1 - u;
               v = 1 - v;
            }
            w = 1 - u - v;
            glm::vec3 newpoint = (u * nee.areaLights[light].vertices[0] + v *
nee.areaLights[light].vertices[1] + w * nee.areaLights[light].vertices[2]);
            glm::vec3 direction = glm::normalize(newpoint - ray.hitPoint);
            glm::vec3 origin = ray.hitPoint + direction * 1e-2f;
            glm::vec2 uv;
            Ray ray2(direction, origin);
            std::shared_ptr<Triangle> triangle2 = grid.DDAalgorithm3(ray2, uv);
            //si hay hit se calcula su contribución de luz
            if (nee.emissivetriangles[light] == triangle2) {
               glm::vec3 N = *nee.emissivetriangles[light]->N[0] * (1.0f - uv.x - uv.y) +
*nee.emissivetriangles[light]->N[1] * uv.x + *nee.emissivetriangles[light]->N[2] * uv.y;
               float pdf = std::max(std::pow(ray2.hitDistance, 2.0f), 1.0f) /
(abs(glm::dot(ray2.direction, N)) * nee.emissivetriangles[light]->area);
               pdf = std::max(pdfmin, pdf);
               ray2.color = nee.emissivetriangles[light]->material->emissive;
               dcolor += indirectColorHit(ray, ray2, hit) / (pdf);
            }
         }
         directColor += directColors[light] / (float)(N);
      }
      delete[] directColors;
```

```
   ///Russian roulette, depende en el throughput BDRF
   Ray Throughput = ray;
   Throughput.color = glm::vec3(1.0f);
   glm::vec3 T = indirectColorHit(rayBefore, Throughput, hitBefore);
   float q;
   if ((maxDepth - depth) < 3) q = 0.0f;
   else q= 1.0f - std::min(std::max(std::max(T.x, T.y), T.z), 1.0f); //
min(std::max(ray.color.x, ray.color.y, ray.color.z), 1.0f);
   if ((rand() % 100 / 100.0f) < q) { //aqui podemos sesgar valores muy bajos
      ray.color += directColor;
      return;
   }
   else {

         glm::vec3 indirectColor = glm::vec3(0.0f);

         //indirect montecarlo rays
         Ray* indirectRays = new Ray[nsecond];
         glm::vec3* indirectColors = new glm::vec3[nsecond];

         #pragma omp parallel for
         for (int i = 0; i < nsecond; i++) {
            Triangle* triangle2 = nullptr;
            float pdf;
            indirectRays[i] = ray.randomRay(rand() % 10000 / 10000.0f, rand() % 10000
/ 10000.0f, hit, pdf);
            renderSecond(indirectRays[i], ray, hit, grid, lights, nee, depth, nsecond);
            pdf = std::max(pdfmin, pdf);
            indirectColors[i] = indirectColorHit(ray, indirectRays[i], hit)/ pdf;
         }
         delete[] indirectRays;
         for (int i = 0; i < nsecond; i++) {
            indirectColor += indirectColors[i];
         }
         //russian roulette boost
         //indirectColor = indirectColor * (1.0f / (1.0f - q));
         delete[] indirectColors;

         //interpolación propia e inventada
         float interpolation = std::min(std::sqrt((1.0f - hit.metalness) *
glm::length(directColor) * exposition), 0.8f);
         ray.color += (directColor * (interpolation)) + indirectColor * (1.0f -
interpolation) / ((float(nsecond)));
   }

   }
   return;
}
```

```
/*****************************************************************************/
/**************************Modulo Shading*************************************/
/*****************************************************************************/

//funcion para determinar el texel y el color del rayo
static inline void setTexel(const Triangle* triangle, Hit& hit, glm::vec2 uv) {

   Material* material = triangle->material;
   int* types = material->textureIndices;


   for (int i = 0; i < material->ntextures; i++) {

      if (types[i] != -1) { //no hay material de este tipo
         Texture* texture = material->textures[types[i]];

         //cuatro es el maximo
         unsigned char texel[4];

         texture->GetTexel(uv, texel);

         unsigned int r = (int)(texel[0]);
         unsigned int g = (int)(texel[1]);
         unsigned int b = (int)(texel[2]);
         unsigned int a;
         if (texture->nrComponents == 4) {
            a = (int)(texel[3]);
         }

         switch (texture->typen) {
         case 1:
            hit.color = glm::pow(glm::vec3({ r,g,b }) / 255.0f, glm::vec3(2.2f));
            if (texture->nrComponents == 4) hit.opacity = a/255.0f;
            break;
         case 2:
            hit.specularColor = glm::pow(glm::vec3({ r,g,b }) / 255.0f, glm::vec3(2.2f));
            break;
         case 4:
            hit.emissive = glm::pow(glm::vec3({ r,g,b }) / 255.0f, glm::vec3(2.2f));
            break;
         case 6:
            glm::vec3 normal = glm::normalize(glm::vec3(2.0f * (float)r / 255.0f - 1.0f, 2.0f
* (float)g / 255.0f - 1.0f, 2.0f * (float)b / 255.0f - 1.0f));
            hit.normal = glm::normalize(glm::mat3((hit.tangent), (hit.bitangent),
(hit.normal)) * (normal));
            hit.tangent = glm::normalize(hit.tangent - glm::dot(hit.tangent, hit.normal) *
hit.normal);
            hit.bitangent = glm::cross(hit.normal, hit.tangent);
            break;
```

```
            case 8:
                hit.opacity = (float)r / 255.0f;
                break;
            case 15:
                hit.metalness = (float)r / 255.0f;
                break;
            case 16:
                hit.roughness = (float)r / 255.0f;
                break;
            case 17:
                hit.oclussion *= (float)r / 255.0f;
                break;
            }

        }
    }
}

/*****************************************************************************/
//metodo para obtener la informacion del choque
static inline void getSurfaceProperties(Ray& ray, const Triangle* triangle, Hit& hit,
const glm::vec2& uv) {

    //coordenadas baricentricas u= uv.x, v = uv.y; w=1-u-v

    hit.normal = *triangle->N[0] * (1.0f - uv.x - uv.y) + *triangle->N[1] * uv.x + *triangle-
>N[2] * uv.y;
    hit.tangent = *triangle->T[0] * (1.0f - uv.x - uv.y) + *triangle->T[1] * uv.x + *triangle-
>T[2] * uv.y;
    hit.bitangent = glm::cross(hit.normal, hit.tangent);

    //para rayos secundarios
    if (glm::dot(ray.direction, hit.normal) > 0.0f){
        hit.R = ray.direction - 2 * glm::dot(ray.direction, -hit.normal) * -hit.normal;
    }
    else {
        hit.R = ray.direction - 2 * glm::dot(ray.direction, hit.normal) * hit.normal;
    }

    // Obtiene el color difuso del material
    hit.color = triangle->material->difuseColor;
    hit.specularColor = triangle->material->specularColor;
    // Obtiene el valor del albedo del material
    hit.opacity = triangle->material->opacity;
    hit.refractiveIndex = triangle->material->n;
    hit.metalness = triangle->material->metalness;
    hit.roughness = triangle->material->roughness;
    hit.emissive = triangle->material->emissive;
```

```
    if (triangle->hasTextures) {
        glm::vec2 uvs; //coordenadas para la textura
        uvs = *triangle->textureCoordinates[0] * (1.0f - uv.x - uv.y) + *triangle-
>textureCoordinates[1] * uv.x + *triangle->textureCoordinates[2] * uv.y;
        setTexel(triangle, hit, uvs);
    }

    hit.a = hit.roughness * hit.roughness;
    hit.a2 = hit.a * hit.a;
}
/*****************************************************************************/
//método que da color al rayo
static inline glm::vec3 color(glm::vec3 V, glm::vec3 L, glm::vec3 N, glm::vec3 F0, const
Hit& hit, glm::vec3 lightcolor) {

    glm::vec3 H = glm::normalize(V + L);
    glm::vec3 F = fresnelSchlick(std::max(glm::dot(H, V), 0.0f), F0);
    float NdotV = abs(glm::dot(N, V));
    float NdotL = abs(glm::dot(N, L));
    float NdotH = abs(glm::dot(N, H));

    glm::vec3 kS = F;
    glm::vec3 kD = glm::vec3(1.0) - kS;
    kD *= (1.0 - hit.metalness);

    glm::vec3 specular = cookTorranceSpecular(NdotV, NdotL, NdotH, hit.roughness, F);
    glm::vec3 color = (1 - hit.oclussion) * (kD * hit.color / glm::pi<float>() * hit.opacity +
specular) * lightcolor * NdotL;

    return color;
}
/*****************************************************************************/
//metodo para determinar iluminacion directa
static inline void lightHit(Ray& ray, Light* light, Hit& hit, Grid& grid) {

    glm::vec3 direction = glm::vec3(0.0f);
    glm::vec3 origin = ray.origin + ray.hitDistance * ray.direction - ray.direction *(5e-2f);
    float radiance = 1.0f;

    if (Distantlight* distantlight = dynamic_cast<Distantlight*>(light)) {

        direction = -distantlight->dir;
        //if (glm::dot(ray.direction, direction) > 0.0f) return;// la luz está detrás del plano
        Ray shadowRay;
        shadowRay.origin = origin;
        shadowRay.setDirection(direction);
        grid.DDAalgorithm2(shadowRay);

        if (shadowRay.hitDistance > 0)return;
```

```
        direction = -distantlight->dir;
        radiance *= light->intensity;
    }
    else if (Pointlight* pointlight = dynamic_cast<Pointlight*>(light)) {

        float distance;
        glm::vec3 vector = pointlight->pos - origin;
        direction = glm::normalize(vector);
        distance = glm::length(vector);

        //if (glm::dot(ray.direction, direction) > 0.0f) return;// la luz está detrás del plano

        Ray shadowRay;
        shadowRay.origin = origin;
        shadowRay.setDirection(direction);
        grid.DDAalgorithm2(shadowRay);

        if (shadowRay.hitDistance < distance && shadowRay.hitDistance>0)return;
        float denominator = std::max<float>((glm::pow(distance * 0.01f, 2.0f)), 1.0f);
        radiance *= light->intensity / denominator;//el flujo se divide en el plano esfera, la
esfera esta en centimetros 0.01 porque son centimetros
    }
    else if (Spotlight* spotlight = dynamic_cast<Spotlight*>(light)) {

        glm::vec3 vector = spotlight->pos - origin;
        direction = glm::normalize(vector);
        //if (glm::dot(ray.direction, direction) > 0.0f) return;// la luz está detrás del plano
        Ray shadowRay;
        shadowRay.origin = origin;
        shadowRay.setDirection(direction);
        grid.DDAalgorithm2(shadowRay);

        if (shadowRay.hitDistance < glm::length(vector) &&
shadowRay.hitDistance>0)return;
        float cosangle = glm::dot(spotlight->dir, -direction);
        float falloff = glm::clamp((cosangle - spotlight->cosInnerAngle) / (spotlight-
>cosInnerAngle - spotlight->cosOuterAngle), 0.f, 1.f); //atenuacion con respecto a la
posicion en el cono
        radiance *= light->intensity * falloff;
    }
    else if (Ambientlight* ambientlight = dynamic_cast<Ambientlight*>(light)) {
        ray.color += hit.color * ambientlight->color * (1.0f - hit.metalness);
        return;
    }
    else { return; }

    ///PBR kd y ks son ahora vectoriales

    glm::vec3 F0 = glm::mix(glm::vec3(0.04f), hit.color, hit.metalness); ///interpolacion
```

```
    glm::vec3 N = hit.normal;
    glm::vec3 L = direction;
    glm::vec3 V = -ray.direction;

    ray.color += color(V, L, N, F0, hit, light->color * radiance);
}
/*********************************************************************************/
//metodo para determinar iluminacion indirecta
static inline glm::vec3 indirectColorHit(const Ray& ray1, const Ray& ray2, const Hit&
hit) {

    if (ray2.color == glm::vec3(0.0f) || glm::isnan(ray2.color.x)) return glm::vec3(0.0f);

    glm::vec3 F0 = glm::mix(glm::vec3(0.04f), hit.color, hit.metalness);
    glm::vec3 N = hit.normal;
    glm::vec3 L = ray2.direction;
    glm::vec3 V = -ray1.direction;

    return color(V, L, N, F0, hit, ray2.color);
}
/*********************************************************************************/
/*********************************Módulo PBR**************************************/
/*********************************************************************************/
//PBR equations para el cook-torrance
glm::vec3 fresnelSchlick(const float& cosTheta, const glm::vec3& F0)
{
    return F0 + (glm::vec3(1.0) - F0) * std::pow(glm::clamp(1.0f - cosTheta, 0.0f, 1.0f),
5.0f);
}

inline float DistributionGGX(const float& NdotH, const float& roughness)
{
    float a = roughness * roughness;
    float a2 = a * a;
    float NdotH2 = NdotH * NdotH;

    float num = a2;
    float denom = (NdotH2 * (a2 - 1.0f) + 1.0f);
    denom = glm::pi<float>() * denom * denom;

    return num / denom;
}

inline float GeometrySchlickGGX(const float& NdotV, const float& roughness)
{
    float r = (roughness + 1.0f);
    float k = (r * r) / 8.0f;

    float num = NdotV;
```

```
    float denom = NdotV * (1.0f - k) + k;

    return num / denom;
}
inline float GeometrySmith(const float& NdotV, const float& NdotL, const float&
roughness)
{
    float ggx2 = GeometrySchlickGGX(NdotV, roughness);
    float ggx1 = GeometrySchlickGGX(NdotL, roughness);

    return ggx1 * ggx2;
}

inline glm::vec3 cookTorranceSpecular(const float& NdotV, const float& NdotL, const
float& NdotH, const float& roughness, const glm::vec3& F) {


    float NDF = DistributionGGX(NdotH, roughness);
    float G = GeometrySmith(NdotV, NdotL, roughness);

    glm::vec3 numerator = NDF * G * F;
    float denominator = 4.0f * NdotV * NdotL + 0.0001f;

    return numerator / denominator;
}
/*********************************************************************************/
/*******************************Modulo Geometry*******************************/
/*********************************************************************************/

//funcion que calcula si un punto esta dentro de una box
inline bool pointBoxOverlap(const glm::vec3 point, const glm::vec3* hitBox) {

    bool overlap = true; //si pongo directamente return el compilador o procesador usa
una optiomizacion que ahce que falle, algo he puesto mal
    for (int i = 0; i < 3; i++) {

        if (!(point[i] >= hitBox[0][i] && point[i] <= hitBox[1][i])) {
            overlap = false;
        }
    }
    return overlap;
}
/*********************************************************************************/
//calcula si una caja se superpone a otra
inline bool boxBoxOverlap(const glm::vec3* hitBox1, const glm::vec3* hitBox2) {

    bool overlap = true; //si pongo directamente return el compilador o procesador usa
una optiomizacion que ahce que falle, algo he puesto mal
    for (int i = 0; i < 3; i++) {
```

```
      if (!(hitBox1[0][i] <= hitBox2[1][i] && hitBox2[0][i] <= hitBox1[1][i])) {
         overlap = false;
      }
   }
   return overlap;
}
/*****************************************************************************/
//calcula el volumen de una BBox
inline glm::vec3 BBoxVolume(glm::vec3* BBox) {
   double width = std::abs(BBox[1].x - BBox[0].x);
   double height = std::abs(BBox[1].y - BBox[0].y);
   double length = std::abs(BBox[1].z - BBox[0].z);
   return glm::vec3(width, height, length);
}
/*****************************************************************************/
// Función para calcular el área de un triángulo dado sus tres vértices
float calculateTriangleArea(glm::vec3 p1, glm::vec3 p2, glm::vec3 p3) {
   glm::vec3 vector1 = glm::vec3({ p2.x - p1.x, p2.y - p1.y, p2.z - p1.z });
   glm::vec3 vector2 = glm::vec3({ p3.x - p1.x, p3.y - p1.y, p3.z - p1.z });
   glm::vec3 cross = glm::cross(vector1, vector2);
   return glm::length(cross / 2.0f);
}
/*****************************************************************************/
//An Efficient and Robust Ray–Box Intersection Algorithm
//funcion que calcula el choque de un vector con una caja rectangular, no es creada
por mi
inline bool boxHit(const Ray& ray, const glm::vec3* hitBox, float& thit) {
   if (pointBoxOverlap(ray.origin, hitBox)) return true;

   float tmin, tmax, txmin, txmax, tymin, tymax, tzmin, tzmax;

   txmin = (hitBox[ray.sign[0]].x - ray.origin.x) * ray.invDir.x; //para acelerar el
proceso se calculan las inversas de la direccion antes
   txmax = (hitBox[1 - ray.sign[0]].x - ray.origin.x) * ray.invDir.x;
   tymin = (hitBox[ray.sign[1]].y - ray.origin.y) * ray.invDir.y;
   tymax = (hitBox[1 - ray.sign[1]].y - ray.origin.y) * ray.invDir.y;
   tmin = txmin;
   tmax = txmax;
   if ((txmin > tymax) || (tymin > txmax)) return false;
   if (tymin > txmin) tmin = tymin;
   if (tymax < txmax) tmax = tymax;
   tzmin = (hitBox[ray.sign[2]].z - ray.origin.z) * ray.invDir.z;
   tzmax = (hitBox[1 - ray.sign[2]].z - ray.origin.z) * ray.invDir.z;
   if ((txmin > tzmax) || (tzmin > txmax)) return false;
   if (tzmin > txmin) tmin = tzmin;
   if (tzmax < txmax) tmax = tzmax;

   thit = tmin;
```

```
    return true;
}

//algoritmo moller trumbore
inline float triangleIntersection(Ray& ray, const glm::vec3* vertices) {
    //coordenadas baricentricas
    float v, u, hitDistance; //(u+v+w=1)

    // algoritmo Möller Trumbore
    glm::vec3 v0v1 = vertices[1] - vertices[0];
    glm::vec3 v0v2 = vertices[2] - vertices[0];
    glm::vec3 pvec = glm::cross(ray.direction, v0v2);

    //determinante y su inversa
    float det = glm::dot(pvec, v0v1);
    float invDet = 1 / det;

    if (fabs(det) < 1e-8) return -1.0;//no hit es paralelo al rayo

    glm::vec3 tvec = ray.origin - vertices[0];
    u = glm::dot(tvec, pvec) * invDet;
    if (u < 0 || u > 1) return -1.0;//no hit

    glm::vec3 qvec = glm::cross(tvec, v0v1);
    v = glm::dot(ray.direction, qvec) * invDet;
    if (v < 0 || u + v > 1) return -1.0;//no hit

    hitDistance = glm::dot(v0v2, qvec) * invDet;
    if (hitDistance < 1e-4)return -1.0;//no hit está detrás

    return hitDistance;
}
/*****************************************************************************/
// algoritmo Möller Trumbore
inline bool rayHit(Ray& ray, Triangle& triangle, glm::vec2& uv)
{
    //coordenadas baricentricas
    float v, u, hitDistance; //(u+v+w=1)

    // algoritmo Möller Trumbore
    glm::vec3 v0v1 = *triangle.vertices[1] - *triangle.vertices[0];
    glm::vec3 v0v2 = *triangle.vertices[2] - *triangle.vertices[0];
    glm::vec3 pvec = glm::cross(ray.direction, v0v2);

    //determinante y su inversa
    float det = glm::dot(pvec, v0v1);
    float invDet = 1.0f / det;
```

```
    if (fabs(det) < 1e-8) return false;//no hit es paralelo al rayo

    glm::vec3 tvec = ray.origin - *triangle.vertices[0];
    u = glm::dot(tvec, pvec) * invDet;
    if (u < 0 || u > 1) return false;//no hit

    glm::vec3 qvec = glm::cross(tvec, v0v1);
    v = glm::dot(ray.direction, qvec) * invDet;
    if (v < 0 || u + v > 1) return false;//no hit

    hitDistance = glm::dot(v0v2, qvec) * invDet;

    if (hitDistance < 1e-4)return false;//no hit está detrás
    if (ray.hitDistance > hitDistance || ray.hitDistance < 0.0f) {

        ray.hitDistance = hitDistance;
        //TODO mirar si hay algo más rapido que esto para hitpoint
        ray.hitPoint = hitDistance * ray.direction + ray.origin;
        uv = glm::vec2(u, v);
        return true;
    }
    return false; // hit
}
/*****************************************************************************/
// algoritmo Möller Trumbore 2 para shadow rays
inline void rayHit2(Ray& ray, const Triangle& triangle) {
    //coordenadas baricentricas
    float v, u, hitDistance; //(u+v+w=1)

    // algoritmo Möller Trumbore
    glm::vec3 v0v1 = *triangle.vertices[1] - *triangle.vertices[0];
    glm::vec3 v0v2 = *triangle.vertices[2] - *triangle.vertices[0];
    glm::vec3 pvec = glm::cross(ray.direction, v0v2);

    //determinante y su inversa
    float det = glm::dot(pvec, v0v1);
    float invDet = 1.0f / det;

    if (fabs(det) < 1e-8) return;//no hit es paralelo al rayo

    glm::vec3 tvec = ray.origin - *triangle.vertices[0];
    u = glm::dot(tvec, pvec) * invDet;
    if (u < 0 || u > 1) return;//no hit

    glm::vec3 qvec = glm::cross(tvec, v0v1);
    v = glm::dot(ray.direction, qvec) * invDet;
    if (v < 0 || u + v > 1) return;//no hit

    hitDistance = glm::dot(v0v2, qvec) * invDet;
```

```
   if (hitDistance < 1e-4)return;//no hit está detrás
   if (ray.hitDistance > hitDistance || ray.hitDistance < 0.0f) {
      ray.hitDistance = hitDistance;

      return;
   }
   return; // hit

   return;
}
/*********************************************************************************/
/*********************************Clase Triangle**********************************/
/*********************************************************************************/

class Figure {
public:
        Material* material;
        virtual ~Figure() {};
};
/*********************************************************************************/
class Esphere: public Figure {
public:
        glm::vec3 origin;
        float radius;
        Material* material=nullptr;
};
/*********************************************************************************/
class Triangle: public Figure {
public:
        glm::vec3* vertices[3], BBox[3]; //vertices del triangulo en sentido antihorario
y BBox(Bounding box)
        glm::vec3* N[3], *T[3], *B[3];     // normales, tangentes, bitangentes
        glm::vec2* textureCoordinates[3]; // coordenadas de las texturas
        float area = 0.0f;
        Material* material=nullptr;
        bool hasTextures = false; //si posee texturas

        Triangle();
        ~Triangle();
        void setBbox();
};

Triangle::Triangle() {

        }

Triangle::~Triangle() {
```

```
	}
void Triangle::setBbox() {

		BBox[0] = *vertices[0];
		BBox[1] = *vertices[0];

		for (unsigned int i = 0; i < 3; i++) {
			for (unsigned int j = 0; j < 3; j++) {
				BBox[0][j] = std::min((*vertices[i])[j], BBox[0][j]);
				BBox[1][j] = std::max((*vertices[i])[j], BBox[1][j]);
			}
		}
		area = 0.5f * glm::length(glm::cross(*vertices[1] - *vertices[0],
*vertices[2] - *vertices[0]));

	}

/*********************************************************************************/
*******************************Clase TriangleMesh*********************************/
/*********************************************************************************/

class TriangleMesh
{
private:
	unsigned int nFaces, nVertices; // numero de triangulos/caras indices y vertices
	unsigned int* indices;	// indices a los vertices
	glm::vec3** vertices;	// vertices de la malla
	glm::vec3 hitBox[2];	//hitbox de la malla para estructuras de aceleración
	glm::vec3** N, **T, **B;		// normales, tangentes, bitangentes
	glm::vec2** texCoordinates; // coordenadas de las texturas
	std::shared_ptr<Material> material; //material
	//crear matrices de objetomundo mundoobjeto
	glm::mat4 objectToWorld, worldtoObject;
	glm::mat3 objectToWorldNormals, worldtoObjectNormals;
	std::shared_ptr<Triangle>* triangles;
	bool hasTextures;

public:
	//Crear constructor
	TriangleMesh();
	//destructor
	~TriangleMesh();
	//setter de la mesh
	void setMesh(unsigned int f, unsigned int* ind, glm::vec3** vert, glm::vec3**
normals, glm::vec3** tangents, glm::vec2** ctextures, unsigned int numofVertices,
std::shared_ptr<Material> mat, bool hastextures);
	// getter numero de caras de la malla
	unsigned int getNumFaces();
	//getter de triangulo
```

```
        std::shared_ptr<Triangle> getTriangle(unsigned int f);
        //setter del triangulo Consideramos el triangulo un struct para acceder rapido
a él
        void setTriangle(unsigned int f);
        //Matriz de transformacion para giros por partes
        void transformObjectToWorld(glm::vec3 initialPosition, float scale, float
xDegrees, float yDegrees, float zDegrees);
        //matriz de transformacion para giros con la matriz directamente
        void transformObjectToWorld(glm::mat4 objectToWorld);
        //multiplicacion de vectores y vertices
        inline void matrixMultiplication(glm::mat4 matrix, glm::mat3 matrixNormals);
        // setter de la hitbox
        void setHitBox();
        //getter de la hitbox
        const glm::vec3* getHitBox();
};

        //Crear constructor
TriangleMesh::TriangleMesh() {

                nFaces = 0;
                nVertices = 0;
                indices = nullptr;
                vertices = nullptr;
                N = nullptr;
                T = nullptr;
                B = nullptr;
                texCoordinates = nullptr;
                material = nullptr;
                triangles = nullptr;
                objectToWorld = glm::mat4(1.0f);
                worldtoObject = glm::mat4(1.0f);
                objectToWorldNormals = glm::mat4(1.0f);
                worldtoObjectNormals = glm::mat4(1.0f);
                hasTextures = false;
                hitBox[0] = glm::vec3(0); //valores min de la hitbox
                hitBox[1] = glm::vec3(0); //valores max de la hitbox

        }

        //destructor
TriangleMesh::~TriangleMesh() {
                for (int v = 0; v < nVertices; v++) {
                        delete vertices[v];
                        delete N[v];
                        delete B[v];
                        delete T[v];
                        if (hasTextures)delete texCoordinates[v];
                }
```

```
		delete[] indices;	// indices a los vertices
		delete[] vertices;
		delete[] N;
		delete[] triangles;
		if (hasTextures)delete[] texCoordinates;
	}

	void TriangleMesh::setMesh(unsigned int f, unsigned int* ind, glm::vec3** vert,
glm::vec3** normals, glm::vec3** tangents, glm::vec2** ctextures, unsigned int
numofVertices, std::shared_ptr<Material> mat, bool hastextures) {

		nFaces = f;
		indices = ind;
		vertices = vert;
		N = normals;
		nVertices = numofVertices;
		texCoordinates = ctextures;
		material = mat;
		T = tangents;
		hasTextures = hastextures;
		B = new glm::vec3 * [nVertices];
		for (unsigned int m = 0; m < nVertices; m++) { B[m] = new
glm::vec3(glm::cross(*N[m], *T[m])); }//chequear si est<73> bien el orden

		triangles = new std::shared_ptr<Triangle>[nFaces];
		for (unsigned int face = 0; face < nFaces; face++) {
			triangles[face] = std::make_shared<Triangle>();
			setTriangle(face);
		}

	}
	unsigned int TriangleMesh::getNumFaces() { return nFaces; }

	std::shared_ptr<Triangle> TriangleMesh::getTriangle(unsigned int f) {
		return triangles[f];
	}

	void TriangleMesh::setTriangle(unsigned int f) {

		triangles[f]->material = material.get();
		triangles[f]->hasTextures = hasTextures;

		for (unsigned int i = 0; i < 3; i++) {
			triangles[f]->vertices[i] = vertices[indices[f * 3 + i]];
			triangles[f]->N[i] = N[indices[f * 3 + i]];
			triangles[f]->T[i] = T[indices[f * 3 + i]];
			triangles[f]->B[i] = B[indices[f * 3 + i]];

			if (hasTextures) {
```

```
                                triangles[f]->textureCoordinates[i] =
texCoordinates[indices[f * 3 + i]];
                        }
                        //triangles[f]->T[i] = glm::normalize(*triangles[f]->T[i] -
glm::dot(*triangles[f]->T[i], *triangles[f]->N[i]) * *triangles[f]->N[i]);
                        //triangles[f]->B[i] = glm::cross(*(triangles[f]->N[i]),
*(triangles[f]->T[i]));
                }
                triangles[f]->area = 0.5f * glm::length(glm::cross(*triangles[f]-
>vertices[1] - *triangles[f]->vertices[0], *triangles[f]->vertices[2] - *triangles[f]-
>vertices[0]));
                triangles[f]->setBbox();
        }

        //Matriz de transformaci<63>n
        void TriangleMesh::transformObjectToWorld(glm::vec3 initialPosition, float
scale, float xDegrees, float yDegrees, float zDegrees) {

                //if (worldtoObject != glm::mat4(1.0f)) {/*todo hacer vuelta atras
*/}TODO hacer para volver atras en las transformaciones

                glm::mat4 rotatex = glm::rotate(glm::mat4(1.0f),
glm::radians(xDegrees), glm::vec3(1.0f, 0.0f, 0.0f)); // Matriz de rotaci<63>n
                glm::mat4 rotatey = glm::rotate(glm::mat4(1.0f),
glm::radians(yDegrees), glm::vec3(0.0f, 1.0f, 0.0f)); // Matriz de rotaci<63>n
                glm::mat4 rotatez = glm::rotate(glm::mat4(1.0f),
glm::radians(zDegrees), glm::vec3(0.0f, 0.0f, 1.0f)); // Matriz de rotaci<63>n
                glm::mat4 scaleM = glm::scale(glm::mat4(1.0f), glm::vec3(scale, scale,
scale));
                glm::mat4 translate = glm::translate(glm::mat4(1.0f), initialPosition); //
Matriz de traslaci<63>n

                objectToWorld = translate * scaleM * rotatez * rotatey * rotatex;
                worldtoObject = glm::inverse(objectToWorld);
                objectToWorldNormals =
glm::mat3(glm::transpose(glm::inverse(scaleM)) * rotatez * rotatey * rotatex);//para
normales el escalado tiene que ser la matriz inversa traspuesta
                worldtoObjectNormals = glm::inverse(objectToWorldNormals);
                matrixMultiplication(objectToWorld, objectToWorldNormals);

                for (unsigned int face = 0; face < nFaces; face++)triangles[face]-
>setBbox();
        }

        void TriangleMesh::transformObjectToWorld(glm::mat4 objectToWorld) {

                //if (worldtoObject != glm::mat4(1.0f))

                this->objectToWorld = objectToWorld;
```

```
		worldtoObject = glm::inverse(objectToWorld);
		//los giros son ortonormales luego da igual invertirlos y trasponerlos junto al escalado
		objectToWorldNormals = glm::transpose(glm::inverse(objectToWorld));//para normales el escalado tiene que ser la matriz inversa traspuesta
		worldtoObjectNormals = glm::inverse(objectToWorldNormals);
		matrixMultiplication(this->objectToWorld, glm::mat3(objectToWorldNormals));

		for (unsigned int face = 0; face < nFaces; face++)triangles[face]->setBbox();
	}

	inline void TriangleMesh::matrixMultiplication(glm::mat4 matrix, glm::mat3 matrixNormals) {

		for (unsigned int i = 0; i < nVertices; i++) {
			*vertices[i] = glm::vec3(matrix * glm::vec4(*vertices[i], 1.0f));
			*N[i] = glm::normalize(matrixNormals * *N[i]);
			*T[i] = glm::normalize(matrixNormals * *T[i]);
			*T[i] = glm::normalize(*T[i] - glm::dot(*T[i], *N[i]) * *N[i]);
			*B[i] = glm::cross(*N[i], *T[i]);
		}
		setHitBox();
	}

	void TriangleMesh::setHitBox() {
		hitBox[0] = *vertices[0];
		hitBox[1] = *vertices[0];

		for (int v = 0; v < nVertices; v++) {  //vertices
			hitBox[0].x = std::min(vertices[v]->x, hitBox[0].x);
			hitBox[0].y = std::min(vertices[v]->y, hitBox[0].y);
			hitBox[0].z = std::min(vertices[v]->z, hitBox[0].z);

			hitBox[1].x = std::max(vertices[v]->x, hitBox[1].x);
			hitBox[1].y = std::max(vertices[v]->y, hitBox[1].y);
			hitBox[1].z = std::max(vertices[v]->z, hitBox[1].z);
		}
	}

	const glm::vec3* TriangleMesh::getHitBox() { return hitBox; }

/*****************************************************************************/
/**********************************Clase Mesh*********************************/
/*****************************************************************************/
class Scene {
```

```
private:
        const aiScene* scene;
        Assimp::Importer importer;
        Texture** textures;
        unsigned int textureCount; //numero de texturas en la escena
        TriangleMesh* meshes;
        unsigned int nMeshes;
        std::string filePath;

public:

        //carga de la escena y constructor
        Scene(const char* fileName);
        //Destructor
        ~Scene();
        //getter del numero de meshes que conforman la escena
        unsigned int getNumMeshes();
        //getter obtencion del numero de caras de una escena en particular
        unsigned int getMeshNumFaces(unsigned int m);
        //getter de un triangulo para una malla  dada y una cara dada
        std::shared_ptr<Triangle> getTriangle(unsigned int m, unsigned int f);
        //aplicar una transfomacion a una malla m en particular
        void transformObjectToWorld(glm::vec3 initialPosition, float scale, float
xDegrees, float yDegrees, float zDegrees, unsigned int m);

        //obtener la textura del importador
        const aiTexture* GetEmbeddedTexture(const char* x);

        //setter de texturas a la escenografia priopia
        void setTextures();
        int* getTextureIndices(unsigned int m, bool& hasTextures);

        //setter importacion de datos de los materiales
        std::shared_ptr<Material> setMaterial(unsigned int m);

        //getter de la hitbox de una malla m en particular
        const glm::vec3* getHitBox(int m);
        //getter de la hitbox de la escena completa
        glm::vec3* getSceneBbox();

        ///recorrido de nodos
        // Función para convertir aiMatrix4x4 a glm::mat4
        glm::mat4 aiMatrix4x4ToGlm(const aiMatrix4x4 & from);

        // Función recursiva para recorrer los nodos y obtener posiciones
        void processNode(aiNode * node, const aiScene * scene, glm::mat4
parentTransform);
        glm::mat4 getnodeMatrixTransformation(aiString name);
```

```
        //setters y getters de lights del importador a la escenografia propia
        aiLight* getSceneLight(int l);
        int getnLights();
        Light* getLight(int l);

        //setter de una cámara usando el importador ASSIMP
        void setCamera(Camera& camera);

};

Scene::Scene(const char* file){
                // Cargar el archivo de modelo de la escena
        filePath= getFileRelativePath(file);
        scene = importer.ReadFile(file, //tal vez haya puesto muchas opciones
                        //aiProcess_PreTransformVertices |
                        aiProcess_Triangulate |
                        aiProcess_GenSmoothNormals |
                        aiProcess_CalcTangentSpace |
                        aiProcess_FlipUVs |
                        aiProcess_TransformUVCoords
                );

                //escena u objeto
                if (!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE || !scene-
>mRootNode) {
                        std::cout << "ERROR::ASSIMP::" << importer.GetErrorString() <<
std::endl;
                        exit(0);
                }

                nMeshes = scene->mNumMeshes;
                meshes = new TriangleMesh[nMeshes];

                //Creacion de las meshes propias, entre otras cosas porque  assimp es
const y no se puede dejar en el heap
                //meshes = new TriangleMesh[nMeshes];
                unsigned int numofFaces;


                glm::vec3** vertices;
                glm::vec3** normals, ** tangents;
                glm::vec2** ctextures;
                unsigned int* indices;
                aiMesh* mesh;
                std::vector<std::shared_ptr<Material>> materials;
                textures = nullptr;
                textureCount = 0;
                setTextures();
```

```
		for (int mat = 0; mat < scene->mNumMaterials; mat++) {
			materials.push_back(setMaterial(mat));
		}


		for (unsigned int m = 0; m < nMeshes; m++) {
			mesh = scene->mMeshes[m];
			numofFaces = mesh->mNumFaces;
			indices = new unsigned int[numofFaces * 3]; //tres verices por
cara
			vertices = new glm::vec3 * [mesh->mNumVertices];
			normals = new glm::vec3 * [mesh->mNumVertices * 3];
			std::shared_ptr<Material>material = materials[mesh-
>mMaterialIndex];
			ctextures = new glm::vec2 * [mesh->mNumVertices];
			tangents = new glm::vec3 * [mesh->mNumVertices];

			bool hasTextures = mesh->HasTextureCoords(0);
			if (hasTextures) material->textureIndices = getTextureIndices(m,
hasTextures);

			//indices de las caras
			for (unsigned int f = 0; f < numofFaces; f++) {
				for (unsigned int i = 0; i < 3; i++) {
					indices[f * 3 + i] = mesh->mFaces[f].mIndices[i];
					//corrección local por si se pasa de rango, parecido
a la corrección de las texturas
					if (indices[f * 3 + i] > mesh->mNumVertices)
indices[f * 3 + i] = ((indices[f * 3 + i] / mesh->mNumVertices) - (int)(indices[f * 3 + i] /
mesh->mNumVertices)) * mesh->mNumVertices;
				}
			}

			for (unsigned nvertices = 0; nvertices < mesh->mNumVertices;
nvertices++) {
				vertices[nvertices] = new glm::vec3();
				normals[nvertices] = new glm::vec3();

				vertices[nvertices]->x = mesh->mVertices[nvertices].x;
				vertices[nvertices]->y = mesh->mVertices[nvertices].y;
				vertices[nvertices]->z = mesh->mVertices[nvertices].z;

				normals[nvertices]->x = mesh->mNormals[nvertices].x;
				normals[nvertices]->y = mesh->mNormals[nvertices].y;
				normals[nvertices]->z = mesh->mNormals[nvertices].z;

				if (hasTextures) {
					tangents[nvertices] = new glm::vec3(0);
```

```
                                        aiVector3D aiTexCoords = mesh-
>mTextureCoords[0][nvertices]; // Coordenadas de textura del v<>rtice i
                                        //las coordenadas uv se supone que van de [0,1]
pero si van mas all<6C> es porque dan vueltas a la textura, supongo y lo normalizo
                                        float u = aiTexCoords.x;
                                        float v = aiTexCoords.y;
                                        ctextures[nvertices] = new glm::vec2(u, v);
                                        tangents[nvertices]->x = mesh-
>mTangents[nvertices].x;
                                        tangents[nvertices]->y = mesh-
>mTangents[nvertices].y;
                                        tangents[nvertices]->z = mesh-
>mTangents[nvertices].z;
                                }
                                else {
                                        tangents[nvertices] = new
glm::vec3(glm::normalize(glm::cross(*normals[nvertices], glm::vec3(1.0f))));
                                        if (*tangents[nvertices] == glm::vec3(0.0f)) {
                                                tangents[nvertices] = new
glm::vec3(glm::normalize(glm::cross(*normals[nvertices], glm::vec3(0.0f, 0.0f, 1.0f))));
                                        }
                                }

                        }
                        meshes[m].setMesh(numofFaces, indices, vertices, normals,
tangents, ctextures, mesh->mNumVertices, material, hasTextures);
                        meshes[m].setHitBox();
                }
                // Procesar la raíz del nodo
                processNode(scene->mRootNode, scene, glm::mat4(1.0f));
        }
        unsigned int Scene::getNumMeshes() { return nMeshes; }
        unsigned int Scene::getMeshNumFaces(unsigned int m) { return
meshes[m].getNumFaces(); }
        std::shared_ptr<Triangle> Scene::getTriangle(unsigned int m, unsigned int f) {
return meshes[m].getTriangle(f); }
        void Scene::transformObjectToWorld(glm::vec3 initialPosition, float scale, float
xDegrees, float yDegrees, float zDegrees, unsigned int m) {
meshes[m].transformObjectToWorld(initialPosition, scale, xDegrees, yDegrees,
zDegrees); }
        Scene::~Scene() {
                for (unsigned int t = 0; t < textureCount; t++) {
                        delete textures[t];
                }
                delete[] textures;
        }
        const aiTexture* Scene::GetEmbeddedTexture(const char* x) {
                return scene->GetEmbeddedTexture(x);
        }
```

```
void Scene::setTextures() {

	aiMesh* mesh;
	aiMaterial* material;
	std::vector<Texture*> textureAux;

	for (unsigned int m = 0; m < nMeshes; m++) {
		mesh = scene->mMeshes[m];
		material = scene->mMaterials[mesh->mMaterialIndex];
		aiString str;

		//aiTextureType_DIFFUSE=1
		//aiTextureType_SPECULAR=2
		// aiTextureType_EMISSIVE=4
		//aiTextureType_NORMALS=6
		//aiTextureType_OPACITY=8

		//PBR
		/*aiTextureType_METALNESS=15
		aiTextureType_DIFFUSE_ROUGHNESS=16
		aiTextureType_AMBIENT_OCCLUSION=17*/

		int types[ntextureperMaterial] = { 1,2,4,6,8,15,16,17 };
		for (int i : types) {
			bool hadIt = false;
			if (material->GetTexture((aiTextureType)(i), 0, &str) ==
AI_SUCCESS) {
				auto it = textureAux.begin();
				//recorremos todas las texturas para ver si ya la
tenía
				while (it != textureAux.end()) {
					if (getFileName((*it)-
>path).compare(getFileName(str.C_Str()))) bool hadIt = true;
					++it;
				}

				if (!hadIt) {
					Texture* texture = new Texture;
					texture->id = textureCount;
					std::string texturePath = filePath + "/" +
std::string(str.C_Str());
					texture->setPath(texturePath.c_str());
					texture->setTextureType(i);
					textureCount++;
					textureAux.push_back(texture);
				}
			}
		}
```

```
		}

		//quiero tenerlo todo en un array porque es de mayor velocidad
		textures = new Texture * [textureCount];
		for (unsigned int t = 0; t < textureCount; t++) {
			textures[t] = textureAux[t];
		}
	}

	int* Scene::getTextureIndices(unsigned int m, bool& hasTextures) {

		aiMesh* mesh;
		aiMaterial* material;
		mesh = scene->mMeshes[m];
		material = scene->mMaterials[mesh->mMaterialIndex];
		int* indexes = new int[ntextureperMaterial];
		aiString str;
		int types[ntextureperMaterial] = { 1,2,4,6,8,15,16,17 };
		int count = 0;
		hasTextures = false;
		for (int i : types) {
			indexes[count] = -1;
			if (material->GetTexture((aiTextureType)(i), 0, &str) ==
AI_SUCCESS) {
				hasTextures = true;
				for (unsigned int t = 0; t < textureCount - 1; t++) {

					//se introduce la textura en el material

						if (getFileName(textures[t]-
>path).compare(getFileName(str.C_Str())) == 0) {
							indexes[count] = textures[t]->id;
							break;
						}
					}
				}
				count++;
			}
			return indexes;
		}

		std::shared_ptr<Material> Scene::setMaterial(unsigned int m) {

			std::shared_ptr<Material> material = std::make_shared<Material>();
			aiMesh* mesh = scene->mMeshes[m];
			aiMaterial* meshMaterial = scene->mMaterials[mesh-
>mMaterialIndex];

			//color y albedo
```

```
            aiColor3D color, specularColor;
            // Obtiene el color difuso del material
            meshMaterial->Get(AI_MATKEY_COLOR_DIFFUSE, color);
            meshMaterial->Get(AI_MATKEY_COLOR_SPECULAR, specularColor);
            // Obtiene el valor del albedo del material
            meshMaterial->Get(AI_MATKEY_SHININESS, material->specularN);
            meshMaterial->Get(AI_MATKEY_SHININESS_STRENGTH, material-
>specularK);
            material->difuseColor = { color.r, color.g, color.b };//hay que cogerlo de
la textura todo, el albedo viene por defecto
            material->specularColor = { specularColor.r, specularColor.g,
specularColor.b };
            material->textures = textures;
            material->ntextures = ntextureperMaterial;
            meshMaterial->Get(AI_MATKEY_OPACITY, material->opacity);
            meshMaterial->Get(AI_MATKEY_REFLECTIVITY, material->reflectivity);
            meshMaterial->Get(AI_MATKEY_METALLIC_FACTOR, material-
>metalness);
            meshMaterial->Get(AI_MATKEY_ROUGHNESS_FACTOR, material-
>roughness);
            if (material->metalness == 0)material->metalness = material-
>reflectivity;
            meshMaterial->Get(AI_MATKEY_COLOR_EMISSIVE, material->emissive);
            aiString name;
            meshMaterial->Get(AI_MATKEY_NAME, name);
            if (std::string(name.C_Str()).find("emissive") != std::string::npos) {
                  material->emissive = material->difuseColor*5.0f;
            }
            if (std::string(name.C_Str()).find("refractive") != std::string::npos) {
                  material->n = 1.4f;
            }
            return material;
      }
      const glm::vec3* Scene::getHitBox(int m) { return meshes[m].getHitBox(); }
      glm::vec3* Scene::getSceneBbox() {

            glm::vec3* bbox = new glm::vec3[2];
            bbox[0] = getHitBox(0)[0];
            bbox[1] = getHitBox(0)[1];

            for (unsigned int m = 0; m < nMeshes; m++) {
                  for (int i = 0; i < 3; i++) {
                        bbox[0][i] = std::min(getHitBox(m)[0][i], bbox[0][i]);
                        bbox[1][i] = std::max(getHitBox(m)[1][i], bbox[1][i]);
                  }
            }
            return bbox;
      }
```

```
void Scene::setCamera(Camera& camera) {

	if (scene->HasCameras()) {
		aiCamera* cam = scene->mCameras[0];
		aiNode* cameraNode = scene->mRootNode->FindNode(cam->mName);
		glm::mat4 nodeTransform;
		if (cameraNode != nullptr) nodeTransform = aiMatrix4x4ToGlm(cameraNode->mTransformation);
		else nodeTransform = aiMatrix4x4ToGlm(scene->mRootNode->mTransformation);

		camera.position = glm::vec3(nodeTransform * glm::vec4(cam->mPosition.x, cam->mPosition.y, cam->mPosition.z, 1.0f));
		camera.up = glm::normalize(glm::vec3(nodeTransform * glm::vec4(cam->mUp.x, cam->mUp.y, cam->mUp.z, 0.0f)));
		camera.forward = glm::normalize(glm::vec3(nodeTransform * glm::vec4(cam->mLookAt.x, cam->mLookAt.y, cam->mLookAt.z, 0.0f)));
		float verticalFOV = 2.0f * glm::atan(cam->mHorizontalFOV / ((float)camera.width / (float)camera.height));
		camera.near = camera.position - camera.forward;
		camera.scale = glm::tan(verticalFOV * 0.5f);

	}
	else {
		glm::vec3* bbox;
		bbox = getSceneBbox();
		camera.position = { 0.0,0.0,0.0 };
		camera.forward = glm::normalize((bbox[1] + bbox[0]) / 2.0f - camera.position);
		camera.near = camera.position - camera.forward;
		delete bbox;
	}
}

aiLight* Scene::getSceneLight(int l) {
	if (!scene->HasLights()) return 0;
	return scene->mLights[l];
}

int Scene::getnLights() {
	return scene->mNumLights;
}

///recorrido de nodos

  // Función para convertir aiMatrix4x4 a glm::mat4
glm::mat4 Scene::aiMatrix4x4ToGlm(const aiMatrix4x4& from) {
	glm::mat4 to;
```

```
		to[0][0] = from.a1; to[0][1] = from.b1; to[0][2] = from.c1; to[0][3] =
from.d1;
		to[1][0] = from.a2; to[1][1] = from.b2; to[1][2] = from.c2; to[1][3] =
from.d2;
		to[2][0] = from.a3; to[2][1] = from.b3; to[2][2] = from.c3; to[2][3] =
from.d3;
		to[3][0] = from.a4; to[3][1] = from.b4; to[3][2] = from.c4; to[3][3] =
from.d4;
		return to;
	}

	// Función recursiva para recorrer los nodos y obtener posiciones
	void Scene::processNode(aiNode* node, const aiScene* scene, glm::mat4
parentTransform) {
		// Convertir la transformación del nodo a glm::mat4
		glm::mat4 nodeTransform = aiMatrix4x4ToGlm(node-
>mTransformation);
		glm::mat4 globalTransform = parentTransform * nodeTransform;

		// Extraer la posición del nodo
		//glm::vec3 position = glm::vec3(globalTransform[3][0],
globalTransform[3][1], globalTransform[3][2]);
		//std::cout << "Node: " << node->mName.C_Str() << " Position: (" <<
position.x << ", " << position.y << ", " << position.z << ")\n";

		// Procesar las mallas del nodo
		for (unsigned int i = 0; i < node->mNumMeshes; i++) {
			//aiMesh* mesh = scene->mMeshes[node->mMeshes[i]];
			//std::cout << "Mesh " << i << " of node " << node-
>mName.C_Str() << " has " << mesh->mNumVertices << " vertices.\n";
			meshes[node-
>mMeshes[i]].transformObjectToWorld(globalTransform);
		}

		// Procesar hijos del nodo
		for (unsigned int i = 0; i < node->mNumChildren; i++) {
			processNode(node->mChildren[i], scene, globalTransform);
		}
	}
	glm::mat4 Scene::getnodeMatrixTransformation(aiString name) {

		aiNode* node = scene->mRootNode->FindNode(name);
		if (node != nullptr) return aiMatrix4x4ToGlm(node->mTransformation);
		else return glm::mat4(1);
	}

	Light* Scene::getLight(int l) {

		aiLight* light = getSceneLight(l);
```

```
		Light* lighto;
		glm::mat4 nodeTransform = getnodeMatrixTransformation(light->mName);
		aiString name = light->mName;
		if (std::string(name.C_Str()).find("ambient") != std::string::npos) {
			lighto = new Ambientlight();
		}
		else {
			// Tipo de luz
			switch (light->mType) {
			case aiLightSource_DIRECTIONAL:
				lighto = new Distantlight();
				break;
			case aiLightSource_POINT:
				lighto = new Pointlight();
				break;
			case aiLightSource_SPOT:
				lighto = new Spotlight();
				break;
			case aiLightSource_AMBIENT:
				lighto = new Ambientlight();
				break;
			default:
				std::cout << light->mName.C_Str() << ", no se ha definido una luz como esa" << std::endl;
				std::cout << light->mType << ", no se ha definido un tipo de luz como esa" << std::endl;

				return nullptr;
			}
		}
		// Color de la luz
		lighto->color.x = light->mColorDiffuse.r;
		lighto->color.y = light->mColorDiffuse.g;
		lighto->color.z = light->mColorDiffuse.b;

		// Posición y dirección (si aplica)
		if (Spotlight* spotlight = dynamic_cast<Spotlight*>(lighto)) {
			spotlight->pos.x = light->mPosition.x;
			spotlight->pos.y = light->mPosition.y;
			spotlight->pos.z = light->mPosition.z;
			spotlight->dir.x = light->mDirection.x;
			spotlight->dir.y = light->mDirection.y;
			spotlight->dir.z = light->mDirection.z;

			spotlight->innerAngle = glm::degrees(light->mAngleInnerCone);
			spotlight->outerAngle = glm::degrees(light->mAngleOuterCone);
			spotlight->setCos();
```

```
                        spotlight->pos = glm::vec3(nodeTransform * glm::vec4(spotlight->pos, 1));
                        spotlight->dir = glm::normalize(glm::vec3(glm::transpose(glm::inverse(nodeTransform)) * glm::vec4(spotlight->dir, 0)));


                }
                if (Distantlight* distantlight = dynamic_cast<Distantlight*>(lighto)) {
                        distantlight->dir.x = light->mDirection.x;
                        distantlight->dir.y = light->mDirection.y;
                        distantlight->dir.z = light->mDirection.z;

                        distantlight->dir = glm::normalize(glm::vec3(glm::transpose(glm::inverse(nodeTransform)) * glm::vec4(distantlight->dir, 0)));

                }
                if (Pointlight* pointlight = dynamic_cast<Pointlight*>(lighto)) {
                        pointlight->pos.x = light->mPosition.x;
                        pointlight->pos.y = light->mPosition.y;
                        pointlight->pos.z = light->mPosition.z;
                        pointlight->pos = glm::vec3(nodeTransform * glm::vec4(pointlight->pos, 1));
                }
                if (Ambientlight* ambientlight = dynamic_cast<Ambientlight*>(lighto)) {
                        ambientlight->color *= 0.0001;
                }
                return lighto;
        }

/*********************************************************************************************Clase Material*************************************/
/*****************************************************************************************/
//clase que define el material
struct Material
{
   glm::vec3 difuseColor, specularColor, emissive; //colores
   float n=1.0f, opacity=1.0f, reflectivity, roughness, metalness; //indice de refraccion y transparencia y reflexión
   float specularK, specularN;
   Texture** textures; //texturas
   int* textureIndices;
   int ntextures; //numero de texturas

   ~Material(){
      delete[] textureIndices;
      delete[] textures;
```

```
    }
};
/*********************************************************************************
**********************************Clase Texture**********************************/
/*********************************************************************************/
//clase que define y guarda las texturas
class Texture {
public:
        unsigned int id=0;              //id
        std::string type;             //tipo de textura
        int typen;
        std::string path;             // la ruta a la textura
        unsigned char* data=nullptr;          // la textura en sí
        int width=0, height=0, nrComponents=0; //datos sobre la imagen

    //métodos para la carga de la textura
    void setPath(const char* str);
    void setTextureType(int itype);
    void LoadTexture(const char* path, int& width, int& height, int& nrComponents);
    //metodo para obtener el textel
    void GetTexel(glm::vec2 uv, unsigned char* texel);
    ~Texture();

};

    void Texture::setPath(const char* str) {
        path = std::string(str);
        LoadTexture(str, width, height, nrComponents);
    }
    void Texture::setTextureType(int itype) {


        switch (itype) {
        case 1:
            type = "DIFFUSE";
            typen = 1;
            break;
        case 2:
            type = "SPECULAR";
            typen = 2;
            break;
        case 4:
            type = "EMISSIVE";
            typen = 4;
            break;
        case 6:
            type = "NORMALS";
            typen = 6;
            break;
```

```
    case 8:
      type = "OPACITY";
      typen = 8;
      break;
    case 15:
      type = "METALNESS";
      typen = 15;
      break;
    case 16:
      type = "ROUGHNESS";
      typen = 16;
      break;
    case 17:
      type = "AO";
      typen = 17;
      break;
    }

  }

  void Texture::LoadTexture(const char* path, int& width, int& height, int&
nrComponents) {

    data = stbi_load(path, &width, &height, &nrComponents, 0);
    if (!data) {
      std::cout << "Failed to load texture at path: " << path << std::endl;
      exit(0);
    }
  }
  void Texture::GetTexel(glm::vec2 uv, unsigned char* texel){

    // Convertir coordenadas UV a coordenadas de píxeles

    //si las coordenadas se salen de la textura vuelven dan vueltas a ella
    if (uv.x > 1) uv.x = uv.x - (int)uv.x;
    else if (uv.x < 0) uv.x = -uv.x + (int)uv.x;
    if (uv.y > 1) uv.y = uv.y - (int)uv.y;
    else if (uv.y < 0) uv.y = -uv.y + (int)uv.y;


    int x = static_cast<int>(std::floor((width - 1) * (uv.x))); // Coordenada x del texel
    int y = static_cast<int>(std::floor((height - 1) * (uv.y))); // Coordenada y del texel

    if (x < 0 || x >= width || y < 0 || y >= height) {
      std::cerr << "Texel coordinates out of bounds" << std::endl;
      std::cerr << x << " " << y << std::endl;
      std::cerr << width << " " << height << std::endl;
      exit(0);
    }
```

```
    int index = (y * (width)+x) * nrComponents;
    for (int i = 0; i < nrComponents; ++i) {
       texel[i] = data[index + i];
    }
  }
  Texture::~Texture() {
    //delete data;
    stbi_image_free(data);
  }

/*********************************************************************************
***********************************Clase Ray*************************************/
/*********************************************************************************/

//structura para mantener en pila los datos del choque y acelerar calculos
struct Hit {
        glm::vec3 normal = glm::vec3(0), tangent = glm::vec3(0), bitangent =
glm::vec3(0), color = glm::vec3(0), specularColor = glm::vec3(0), emissive =
glm::vec3(0), H = glm::vec3(0), R = glm::vec3(0),V = glm::vec3(0);
        float oclussion = 0.0, opacity = 1.0, roughness = 1.0, metalness = 1.0,
refractiveIndex = 1.0f;
        float a = 0.0f, a2 = 0.0f; //roughness ^2 ^4
};

/*********************************************************************************/
//los valores -1.0f son los inicializados
class Ray {

public:
        glm::vec3 direction, invDir; //dirección y la inversa de la dirección
        glm::ivec3 sign; //signo de cada direccion para el boxhit
        glm::vec3 origin, hitPoint, color;
        float hitDistance = -1.0f;

        Ray() {
                direction = glm::vec3(0.0f);
                color = { 0.0f,0.0f,0.0f };
                origin = { 0.0f,0.0f,0.0f };
                invDir = glm::vec3(0);
                sign = glm::ivec3(0);
                hitPoint = glm::vec3(0);
                hitDistance = -1.0f;
        }

        Ray(glm::vec3 dir, glm::vec3 orig) {
                setDirection(dir);
                color = { 0.0f,0.0f,0.0f };
                origin = orig;
                hitDistance = -1.0f;
```

```
            invDir = glm::vec3(1 / direction.x, 1 / direction.y, 1 / direction.z);
            hitPoint = glm::vec3(0);
        }

        ~Ray() {
        }

        Ray createReflectionRay(glm::vec3 hitNormal) {

            glm::vec3 origin2, direction2;

            /*if (hit.opacity = 1 && glm::dot(ray.direction, hit.normal) > 0) {
hit.normal = -hit.normal; }
            else if (hit.opacity < 1 && glm::dot(ray.direction, hit.normal) > 0) {
hit.refractiveIndex = 1; hit.hitrefractiveIndex = 1.5; hit.normal = -hit.normal;}
            else if (hit.opacity < 1 && glm::dot(ray.direction, hit.normal) < 0) {
hit.refractiveIndex = 1.5; hit.hitrefractiveIndex = 1;}*/
            if (glm::dot(direction, hitNormal) > 0) hitNormal = -hitNormal;
            direction2 = direction - 2 * glm::dot(direction, hitNormal) * hitNormal;
            origin2 = this->hitPoint + direction2 * 1e-2f;
            Ray ray2(direction2, origin2);
            return ray2;
        }

        Ray createTransmissionRay(glm::vec3 hitNormal, float refractiveIndex) {

            float n1, n2;
            n1 = 1;
            n2 = refractiveIndex;
            if (glm::dot(direction, hitNormal) > 0.0f) {
                hitNormal = -hitNormal;
                std::swap(n1, n2);
            }

            float n = n1 / n2;//indice fredfraccion material 1/indice de refraccion
material 2

            float c1, c2; //coeficiente 1 y 2;
            glm::vec3 origin2, direction2;

            c1 = glm::dot(direction, hitNormal);
            c2 = glm::sqrt(1.0f - glm::pow(n, 2.0f) * (1.0f - pow(c1, 2.0f)));
            direction2 = glm::normalize(n * direction + (n * c1 - c2) * hitNormal);

            origin2 = this->hitPoint + direction2 * 1e-2f;
            Ray ray2(direction2, origin2);
            return ray2;
        }
```

```
	float fresnel(glm::vec3 hitNormal, float refractiveIndex) {

		float cosi = glm::dot(direction, hitNormal);//coseno incidente
		float n1, n2; //coeficientes de refreacion de cada medio

		n1 = 1.0f;
		n2 = refractiveIndex;

		if (cosi > 0.0f) {//debajo o dentro
			std::swap(n1, n2);
			float sint = n1 / n2 * glm::sqrt(1.0f - glm::pow(cosi, 2.0f));
			if (sint > 0.98f) return 1.0f; //reflexion interna total
		}
		else cosi = glm::abs(cosi);


		return fresnelSchlick(cosi, n1, n2);
		/*
		float sint = n1 / n2 * glm::sqrt(1.0f - glm::pow(cosi, 2.0f));
		if (sint>1.0f) return 1.0f; //reflexion interna total
		float cost = glm::sqrt(1.0f - glm::pow(sint, 2.0f)); //coseno transmitido
		float Rs = ((n1 * cosi) - (n2 * cost)) / ((n1 * cosi) + (n2 * cost));//
	poralizacion perpenduclar polarizacion en s
		float Rp = ((n1 * cost) - (n2 * cosi)) / ((n2 * cosi) + (n1 *
cost));//polarizacion paralela polarizacion en p
		float result = (glm::pow(Rs, 2.0f) + glm::pow(Rs, 2.0f)) / 2.0f;
		return result;*/
	}

	static float fresnelSchlick(float cos, float n1, float n2) {
		//Schlick's approximation for reflectance fresnel.
		float r0 = (n1 - n2) / (n1 + n2);
		r0 = r0 * r0;
		return r0 + (1 - r0) * std::pow((1 - cos), 5);
	}

	void setDirection(glm::vec3 d) {
		direction = d;
		invDir = glm::vec3(1 / direction.x, 1 / direction.y, 1 / direction.z);
		sign[0] = (invDir.x < 0);
		sign[1] = (invDir.y < 0);
		sign[2] = (invDir.z < 0);
	}

	inline Ray randomRay(const float& random1, const float& random2, const Hit&
hit, float& pdf) {

		//random1 y 2 se dejan ahi por si se quiere usar estratificación.
```

```
            //en coordenadas esfericas la semiesfera
            float theta = random1 * 3.1416f * 2.0f; //los valores muy tangentes me
dan igual por eso 3.14 y no pi
            //float phi = random2 * 3.1416f/2.0f;

            //en coordenadas cartesianas con centro en el choque
            /*float x = glm::sin(phi) * glm::cos(theta);
            float y = glm::sin(phi) * glm::sin(theta);
            float z = glm::cos(phi);
            */
            float t = glm::clamp(1.0f - hit.roughness, 0.25f, 1.0f); //probabilidad de
cada caso
            float random3 = rand() % 10000 / 10000.0f;
            glm::vec3 direction2, origin2;
            //difuso importance sampling
            if (random3 > t) {

                  float x = std::sqrtf(random2) * glm::cos(theta);
                  float y = std::sqrtf(random2) * glm::sin(theta);
                  float z = std::sqrtf(1 - random2);

                  direction2 = glm::normalize(glm::mat3((hit.bitangent),
(hit.tangent), (hit.normal)) * glm::normalize(glm::vec3(x, y, z)));
                  origin2 = this->hitPoint + hit.normal * 5e-2f;

                  //PDF
                  pdf = glm::dot(hit.normal, direction2) / 3.1416f;
            }
            //specular importance sampling
            else {

                  glm::vec3 R = hit.R;

                  float phi = glm::atan(hit.roughness * std::sqrtf(random2) /
(std::sqrtf(1.0f - random2)));

                  //importance sampling coseno
                  float x = glm::sin(phi) * glm::cos(theta);
                  float y = glm::sin(phi) * glm::sin(theta);
                  float z = glm::cos(phi);

                  glm::vec3 T, B;

                  T = glm::cross(R, hit.tangent);
                  B = glm::cross(T, R);

                  direction2 = glm::normalize(glm::mat3(B, T, R) *
glm::normalize(glm::vec3(x, y, z)));
                  origin2 = this->hitPoint + hit.normal * 5e-2f;
```

```
                        //PDF
                        glm::vec3  H = glm::normalize(direction2-direction);
                        pdf = (1.0f - t) * glm::dot(hit.normal, direction2) / 3.1416f + t *
(glm::dot(hit.normal, H) * Distribution(hit.normal, H, hit)) / (4.0f *
glm::dot(direction2, H));
                }


                Ray ray2(direction2, origin2);
                return ray2;
        }
        float Distribution(glm::vec3 N, glm::vec3 H, const Hit& hit)
        {
                float NdotH = abs(glm::dot(N, H));
                float NdotH2 = NdotH * NdotH;

                float num = hit.a2;
                float denom = (NdotH2 * (hit.a2 - 1.0f) + 1.0f);
                denom = glm::pi<float>() * denom * denom;

                return num / denom;
        }
};
/*************************************************************************************
**********************************Clase Camera*************************************/
/*************************************************************************************/

class Camera
{
public:
        float fov, aspectRatio, aspectRatioy, scale;
        uint32_t width, height;
        glm::vec3 position, near, forward,up;// , vertices[4];//el vector near(el ojo) y los
vectores que definen la camara
public:
        Camera(float fov, uint32_t width, uint32_t height) {
                this->fov = fov;
                this->width = width;
                this->height = height;
                up = { 0,1,0 };
                forward = { 0,0,1 };
                position = { 0,0,0 };
                near = position-forward;
                scale = glm::tan(glm::radians(fov) * 0.5f);
                aspectRatio = ((float)width/(float)height);
                aspectRatioy = 1;

                if (aspectRatio < 1) {
```

```
                aspectRatioy = (float)height/(float)width;
                aspectRatio = 1;
            }
        }
        ~Camera(){}

        //con origin para que el origen este en el borde la camara.
        glm::vec3 setRayDirection(float x, float y,glm::vec3 &origin) {
            glm::vec3 right = glm::cross(up,forward);

            x *= scale * aspectRatio;
            y *= scale * aspectRatioy;

            glm::vec3 vector = glm::mat3((right), (up), (forward)) * glm::vec3(x, y,
1.0);
            origin = vector + near;
            return glm::normalize(vector);
        }
        glm::vec3 setRayDirection(float x, float y) {
            glm::vec3 right = glm::cross(up, forward);

            x *= scale * aspectRatio;
            y *= scale * aspectRatioy;

            return glm::normalize(glm::mat3((right), (up), (forward)) * glm::vec3(x,
y, 1.0));
        }
};
/*****************************************************************************
**********************************Clase Light**********************************/
/*****************************************************************************/
class Light {
public:
   glm::mat4 lightToWorld; //matriz de lugar de la luz
   glm::vec3 color = {0.0f,0.0f,0.0f};
   float intensity;
    Light() {
      lightToWorld = glm::mat4(1.0f);
      intensity = 1;
   }
    Light(glm::mat4 l2w) {
       lightToWorld = l2w;
       intensity = 1;
    }
   virtual ~Light() {}
};
/*****************************************************************************/

//Luz puntual
```

```
class Pointlight : public Light {
public:
   glm::vec3 pos; //postición
   Pointlight() {
      this->color = {1.0f,1.0f,1.0f};
      this->intensity = 1;
      pos = { 0.0,0.0,-1.0};
   }
};

/*****************************************************************************/
//Luz dinstante
class Distantlight : public Light {
public:

   glm::vec3 dir; //dirección de la luz


   Distantlight() {
      this->color = { 1.0f,1.0f,1.0f };
      this->intensity = 1;
      dir = {0,0,1};
      glm::normalize(dir);
   }

   Distantlight(glm::vec3 direction) {
      this->color = { 1.0f,1.0f,1.0f };
      this->intensity = 1;
      dir=direction;
      glm::normalize(dir);
   }

};

/*****************************************************************************/
class Spotlight : public Light {
public:

   glm::vec3 dir; //dirección de la luz
   glm::vec3 pos; //postición
   float innerAngle,cosInnerAngle, outerAngle, cosOuterAngle;

   Spotlight() {
      color = { 1.0f,1.0f,1.0f };
      intensity = 1;
      innerAngle = 30;
      outerAngle = 35;
      dir = {0.0f,0.0f,1.0f};
      pos = { 0.0, 0.0, 0.0 };
```

```
        setCos();

    }
    void setCos() {
        glm::normalize(dir);
        cosInnerAngle = glm::cos(glm::radians(innerAngle));
        cosOuterAngle = glm::cos(glm::radians(outerAngle));
    }

};

/*****************************************************************************/
class Ambientlight : public Light {
//no tiene informacion relevante
};

/*****************************************************************************/
//las luces de area se representan con objetos emisivos
class Arealight : public Light {
public:

    glm::vec3 vertices[3]; //postición
    glm::vec3 center; //center
    glm::vec3 *points; //puntos que conforman el Area light
    int N = 1; ///numero de particiones de la luz, para luces de area

    Arealight(glm::vec3 **verticesIN, glm::vec3 colorIN) {

        this->color = colorIN;
        for (int i = 0; i < 3; i++)vertices[i] = *verticesIN[i];

        float area = calculateTriangleArea(vertices[0], vertices[1], vertices[2]);
        N = (int)std::max(std::ceil(area * 10.0f/1000.0f),3.0f); //numero de
descomposición del triángulo, cuanto mas grande más puntos
        center = (vertices[0] + vertices[1] + vertices[2])/3.0f;
        points = new glm::vec3[N];
        color = { 1.0f,1.0f,1.0f };
        intensity = 1;

        areaPointDescomposition();
    }
    ~Arealight() { // los points no son suyos delete[] points;
    }

    void areaPointDescomposition() {

        //Primero calculammos las coordenadas baricentricas del triangulo para que los
puntos sean equiespaciados
        // y las pasamos a coordenadas carteasianas
```

```
    //poison points    PoissonGenerator::DefaultPRNG PRNG; //generador de puntos
poisson

    for (int i = 0; i < N; i++) {
      float random1 = rand() % 10000 * 1e-4f, random2 = rand() % 10000 * 1e-4f;
      points[i] = baricentricToCartesian(glm::vec2(random1, random2), vertices);
    }
  }

  inline float calculateTriangleArea(const glm::vec3 vert1, const glm::vec3 vert2, const
glm::vec3 vert3) {

    return 0.5f * glm::length(glm::cross(vert2 - vert1, vert3 - vert1));
  }

  //funcion que calcula las coordenadas cartesianas dadas las baricentricas
  glm::vec3 baricentricToCartesian(const glm::vec2 uv, const glm::vec3* vertices) {

    float u = uv.x, v = uv.y;

    float w = 1 - u - v;

    //corrección para baricentricas aleatorias [0,1][0,1]
    if (w < 0) {
      w = -w;
      v = 1 - v;
      u = 1 - u;
    }

    glm::vec3 cartesianCoordinates = glm::vec3(0);

    cartesianCoordinates.x = u * vertices[0].x + v * vertices[1].x + w * vertices[2].x;
    cartesianCoordinates.y = u * vertices[0].y + v * vertices[1].y + w * vertices[2].y;
    cartesianCoordinates.z = u * vertices[0].z + v * vertices[1].z + w * vertices[2].z;
    return cartesianCoordinates;
  }

};
/*********************************************************************************
*******************************Clase NEE*****************************************/
/*********************************************************************************/



//Next event estimation
//Estructura de datos que guarda triangulos emisivos

class NEE {
public:
```

```
   std::vector<std::shared_ptr<Triangle>> emissivetriangles;
   std::vector<Arealight> areaLights;
   int N = 0;

   void addTriangle(std::shared_ptr<Triangle> triangle) {


      if (triangle->material->emissive != glm::vec3(0.0f)) {
         emissivetriangles.push_back(triangle);
         areaLights.push_back(Arealight(triangle->vertices, triangle->material-
>emissive));
      }
      else if(triangle->hasTextures) {

         Material* material = triangle->material;
         int* types = material->textureIndices;

         //para materias con texturas

         #pragma omp parallel for
         for (int i = 0; i < material->ntextures; i++) {
            if (types[i] != -1) { //no hay material de este tipo emisivo
               Texture* texture = material->textures[types[i]];

               if (texture->typen == 4) {
                  unsigned char texel[3];
                  glm::vec2 uvs = *triangle->textureCoordinates[0];
                  texture->GetTexel(uvs, texel);

                  unsigned int r = (int)(texel[0]);
                  unsigned int g = (int)(texel[1]);
                  unsigned int b = (int)(texel[2]);
                  glm::vec3 light = glm::pow(glm::vec3({ r,g,b }) / 255.0f, glm::vec3(2.2f));
                  if (std::max(std::max(r,g),b) > 0) {

                     material->emissive = light;
                     emissivetriangles.push_back(triangle);
                     areaLights.push_back(Arealight(triangle->vertices, triangle->material-
>emissive));
                  }
               }
            }
         }
      }


      for (int light = 0; light < size(); light++) {
         N+= areaLights[light].N;
      }
```

```
    }
    int size() {
        return emissivetriangles.size();
    }
    int numberofPoints() {
        return N;
    }

};
/*************************************************************************************
*******************************Fujimoto Grid*****************************************/
/*************************************************************************************/

//cada celda del grid
class Cell {

    //vector de punteros a triangulos que posse
public:
    std::vector<std::shared_ptr<Triangle>> triangles;

    void addTriangle(std::shared_ptr<Triangle> triangle) {
        triangles.push_back(triangle);
    }
};
/*************************************************************************************/
//Akira Fujimoto Grid
class Grid
{
private:
    glm::vec3 BBox[2]; //Bound Box de la escena, para delimitar la grid
    Scene* scene;
    glm::vec3 cellDimension;//tamaño de las celdas y
    glm::ivec3 resolution; // numero de celdas por cada dimesion
    int nCells; //numero de celdas
    Cell* cells;//celdas

public:

    Grid(Scene* sceneIn) {

        scene = sceneIn;
        setBBox();
        setCell();
        cells = new Cell[nCells];
        introduceTriangles();
    }

    ~Grid() { delete[] cells; }
```

```
  //define los bordes de la grid
  void setBBox() {

    //inicializamos la bbox con la hitbox de la mesh de la escena
    glm::vec3 Box[2] = { scene->getHitBox(0)[0], scene->getHitBox(0)[1] };
    BBox[0] = Box[0];
    BBox[1] = Box[1];


    for (int mesh = 0; mesh < scene->getNumMeshes(); mesh++) {
      Box[0] = scene->getHitBox(mesh)[0];
      Box[1] = scene->getHitBox(mesh)[1];
      for (int i = 0; i < 3; i++) {
        BBox[0][i] = std::min(Box[0][i], BBox[0][i]);
        BBox[1][i] = std::max(Box[1][i], BBox[1][i]);
      }
    }
  }

  //define el tamaño de cada celda y cuántas hay por dimension
  void setCell() {
    int numtriangles = 0; //numero de triangulos
    int lambda = 7; // valor de corrección aleatorio
    for (int mesh = 0; mesh < scene->getNumMeshes(); mesh++) numtriangles +=
scene->getMeshNumFaces(mesh);

    glm::vec3 dim = BBoxVolume(BBox);
    float volume = dim.x * dim.y * dim.z;

    resolution = glm::ceil((std::cbrt((lambda * numtriangles) / volume)) * dim);
    cellDimension = dim / (glm::vec3)resolution;
    nCells = resolution.x * resolution.y * resolution.z;
  }
  //introduce los triangulos en cada celda del grid
  void introduceTriangles() {


    for (int mesh = 0; mesh < scene->getNumMeshes(); mesh++) {
      for (int triangle = 0; triangle < scene->getMeshNumFaces(mesh); triangle++) {
        //transformacion a las coordenadas de la grid                    resto -1
porque tiene que poder ser 0, he tardado en darme cuenta
        glm::ivec3 cellMin = glm::floor((scene->getTriangle(mesh, triangle)->BBox[0]
- BBox[0]) / (cellDimension)-glm::vec3(1));
        glm::ivec3 cellMax = glm::ceil((scene->getTriangle(mesh, triangle)->BBox[1] -
BBox[0]) / (cellDimension));


        //iba a crear una funcion de overlap pero esto es más sencillo
```

```
            for (int z = std::max<int>(cellMin.z, 0); z < std::min<int>(cellMax.z,
resolution.z); ++z) {
                for (int y = std::max<int>(cellMin.y, 0); y < std::min<int>(cellMax.y,
resolution.y); ++y) {
                    for (int x = std::max<int>(cellMin.x, 0); x < std::min<int>(cellMax.x,
resolution.x); ++x) {
                        cells[(x + resolution.x * y + z * resolution.x *
resolution.y)].addTriangle(scene->getTriangle(mesh, triangle));
                    }
                }
            }
        }
    }

    //algoritmo para recorrer la Grid (de akira fujimoto, no es mio propio, es adaptado)
    //devuelve un triangulo y coordenadas uv
    std::shared_ptr<Triangle> DDAalgorithm(Ray& ray, glm::vec2 &uv) {

        float tHit = 0.0f;
        if (!boxHit(ray, BBox, tHit)) return nullptr;
        glm::ivec3 exit, step, cell;
        glm::vec3 deltaT, nextCrossingT;


        // convertir el hit con la bbox a coordenadas de la grid
        glm::vec3 rayOrigCell = (ray.origin + ray.direction * tHit) - BBox[0];

        for (int i = 0; i < 3; ++i) {
            // convierte el choque ddel rayo en coordenadas de las celdas

            cell[i] = glm::clamp<int>((((float)rayOrigCell[i] / cellDimension[i]) - 1.0f), 0.0f,
(float)resolution[i] - 1.0f); //resto -1 porque tiene que poder ser 0, he tardado en
darme cuenta

            if (ray.direction[i] < 0) {

                deltaT[i] = -cellDimension[i] * ray.invDir[i];
                nextCrossingT[i] = tHit + (cell[i] * cellDimension[i] - rayOrigCell[i]) *
ray.invDir[i];
                exit[i] = -1;
                step[i] = -1;
            }
            else {

                deltaT[i] = cellDimension[i] * ray.invDir[i];
                nextCrossingT[i] = tHit + ((cell[i] + 1) * cellDimension[i] - rayOrigCell[i]) *
ray.invDir[i];
                exit[i] = resolution[i];
```

```
            step[i] = 1;
        }
    }

    //recorrido del rayo por el grid
    std::shared_ptr<Triangle> triangle = nullptr;
    while (1) {
        unsigned int o = cell.z * resolution.x * resolution.y + cell.y * resolution.x + cell.x;

        //boxHit(ray, cells[o].BBox, tHit);
        if (!cells[o].triangles.empty()) {

            for (int i = 0; i < cells[o].triangles.size(); ++i) {
                if(rayHit(ray, *(cells[o].triangles[i]), uv)) triangle = cells[o].triangles[i];
            }

        }
        unsigned int k =
            ((nextCrossingT[0] < nextCrossingT[1]) << 2) +
            ((nextCrossingT[0] < nextCrossingT[2]) << 1) +
            ((nextCrossingT[1] < nextCrossingT[2]));
        static const unsigned map[8] = { 2, 1, 2, 1, 2, 2, 0, 0 };
        unsigned int axis = map[k];

        if (ray.hitDistance < nextCrossingT[axis] && ray.hitDistance>0) break;
        cell[axis] += step[axis];
        if (cell[axis] == exit[axis]) break;
        nextCrossingT[axis] += deltaT[axis];
    }
    return triangle;
}


//para los shadow rays
void DDAalgorithm2(Ray& ray) {

    glm::ivec3 exit, step, cell;
    glm::vec3 deltaT, nextCrossingT;


    // convertir el hit con la bbox a coordenadas de la grid
    glm::vec3 rayOrigCell = ray.origin - BBox[0];

    for (int i = 0; i < 3; ++i) {
        // convierte el choque ddel rayo en coordenadas de las celdas

        cell[i] = glm::clamp<int>((((float)rayOrigCell[i] / cellDimension[i]) - 1.0f), 0.0f,
(float)resolution[i] - 1.0f); //resto -1 porque tiene que poder ser 0, he tardado en
darme cuenta
```

```
        if (ray.direction[i] < 0) {

          deltaT[i] = -cellDimension[i] * ray.invDir[i];
          nextCrossingT[i] =  (cell[i] * cellDimension[i] - rayOrigCell[i]) * ray.invDir[i];
          exit[i] = -1;
          step[i] = -1;
        }
        else {

          deltaT[i] = cellDimension[i] * ray.invDir[i];
          nextCrossingT[i] = + ((cell[i] + 1) * cellDimension[i] - rayOrigCell[i]) *
ray.invDir[i];
          exit[i] = resolution[i];
          step[i] = 1;
        }
      }

      //recorrido del rayo por el grid
      std::shared_ptr<Triangle> triangle = nullptr;
      while (1) {
        unsigned int o = cell.z * resolution.x * resolution.y + cell.y * resolution.x + cell.x;

        //boxHit(ray, cells[o].BBox, tHit);
        if (!cells[o].triangles.empty()) {


          for (int i = 0; i < cells[o].triangles.size(); ++i) {
                if (cells[o].triangles[i]->material->opacity !=0.0f) {
                rayHit2(ray, *(cells[o].triangles[i]));
              }
            }
        }
        unsigned int k =
          ((nextCrossingT[0] < nextCrossingT[1]) << 2) +
          ((nextCrossingT[0] < nextCrossingT[2]) << 1) +
          ((nextCrossingT[1] < nextCrossingT[2]));
        static const unsigned map[8] = { 2, 1, 2, 1, 2, 2, 0, 0 };
        unsigned int axis = map[k];

        if (ray.hitDistance < nextCrossingT[axis] && ray.hitDistance>0) break;
        cell[axis] += step[axis];
        if (cell[axis] == exit[axis]) break;
        nextCrossingT[axis] += deltaT[axis];
      }
    }

    //para los rayos dentro del grid
    std::shared_ptr<Triangle> DDAalgorithm3(Ray& ray, glm::vec2& uv) {
```

```
    glm::ivec3 exit, step, cell;
    glm::vec3 deltaT, nextCrossingT;

    // convertir el hit con la bbox a coordenadas de la grid
    glm::vec3 rayOrigCell = ray.origin - BBox[0];

    for (int i = 0; i < 3; ++i) {
      // convierte el choque ddel rayo en coordenadas de las celdas

      cell[i] = glm::clamp<int>((((float)rayOrigCell[i] / cellDimension[i]) - 1.0f), 0.0f,
(float)resolution[i] - 1.0f); //resto -1 porque tiene que poder ser 0, he tardado en
darme cuenta

      if (ray.direction[i] < 0) {

        deltaT[i] = -cellDimension[i] * ray.invDir[i];
        nextCrossingT[i] = (cell[i] * cellDimension[i] - rayOrigCell[i]) * ray.invDir[i];
        exit[i] = -1;
        step[i] = -1;
      }
      else {

        deltaT[i] = cellDimension[i] * ray.invDir[i];
        nextCrossingT[i] = ((cell[i] + 1) * cellDimension[i] - rayOrigCell[i]) *
ray.invDir[i];
        exit[i] = resolution[i];
        step[i] = 1;
      }
    }

    //recorrido del rayo por el grid
    std::shared_ptr<Triangle> triangle = nullptr;
    while (1) {
      unsigned int o = cell.z * resolution.x * resolution.y + cell.y * resolution.x + cell.x;

      //boxHit(ray, cells[o].BBox, tHit);
      if (!cells[o].triangles.empty()) {

        for (int i = 0; i < cells[o].triangles.size(); ++i) {
          if (rayHit(ray, *(cells[o].triangles[i]), uv)) triangle = cells[o].triangles[i];
        }

      }
      unsigned int k =
        ((nextCrossingT[0] < nextCrossingT[1]) << 2) +
        ((nextCrossingT[0] < nextCrossingT[2]) << 1) +
        ((nextCrossingT[1] < nextCrossingT[2]));
      static const unsigned map[8] = { 2, 1, 2, 1, 2, 2, 0, 0 };
```

```
        unsigned int axis = map[k];

        if (ray.hitDistance < nextCrossingT[axis] && ray.hitDistance>0) break;
        cell[axis] += step[axis];
        if (cell[axis] == exit[axis]) break;
        nextCrossingT[axis] += deltaT[axis];
      }
      return triangle;
    }
};
/*****************************************************************************
*****************************************************************************
*****************************************************************************/
```