UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Centre de la Imatge i la Tecnologia Multimèdia

Desarrollo de un renderer 3D con path tracing

Lucas Pérez García

Director: Torres Jiménez, Marc

Grau: Diseño y desarrollo de videojuegos

Curs: 2022-23

Universitat: CITM-UPC

Índice

Resumen 5
Palabras clave 6
Enlaces 6
Índice de tablas 7
Índice de figuras 8
Glosario 12
1. Introducción 13
1.1 Motivación 13
1.2 Formulación del problema 13
1.3 Objetivos generales del TFG 14
1.4 Objetivos específicos del TFG 14
1.5 Alcance del proyecto 15
2. Estado del arte/Marco Teórico/Contextualización/Estudio de Mercado 16
2.1 Deferred rendering 16
2.1.1 Cómo funciona la técnica 17
2.1.2 Ventajas y desventajas 18
2.2 Physically based rendering 18
2.3 Iluminación global 19
2.2.1 Ray tracing 20
2.2.2 Path tracing 25
3. Gestión del proyecto 30
3.1. DAFO 30
3.2. Riesgos y plan de contingencias 30
3.2.1. Motivación 31
3.2.2. Problemas de hardware 31
3.2.3. Proyecto muy complejo 31
3.2.4. Documentación escasa 32
3.3. Análisis inicial de costos 32
4. Metodología 34
4.1. Herramientas de monitorización del proyecto 35
4.1.1. Trello 35
4.1.2. GitHub 36
4.2. Herramientas de validación 37
4.3. Modificaciones 38
5. Desarrollo del proyecto 40

5.1. Introducción al desarrollo 40
5.2. Librerías externas utilizadas 42
5.3. Base del motor 42
5.3.1. Aplicación 42
5.3.2. Ventana, eventos e inputs 43
5.3.3. Logging 44
5.3.4. Sistema de layers 44
5.3.5. Renderer 45
5.4. Renderer 3D con path tracing 49
5.4.1. Gestión de los datos que se envían a la GPU 50
5.4.2. Visualización del resultado 54
5.4.3. Desarrollo del path tracing 56
5.4.4. Funcionalidades del editor 69
6. Validación del proyecto 73
7. Conclusiones 74
7.1 Líneas de futuro 74
8. Bibliografía 76

Resumen

Actualmente es muy difícil obtener resultados gráficos realistas de manera fidedigna a la hora de renderizar escenas 3D. Muchas veces se utilizan muchos otros conceptos para intentar simular de la mejor manera posible, pero la realidad es que esos resultados, pese a ser muy buenos, no llegan a un nivel de hiper realismo suficiente. El objetivo principal de este proyecto es crear un renderer 3D que implemente la técnica del path tracing para conseguir renderizar una escena 3D simulando la iluminación de la manera más realista posible. Otros de los objetivos de este proyecto son aumentar los conocimientos en esta materia con la intención de que sirva para poder trabajar sobre ella en el futuro.

Este proyecto va dirigido a todo aquel que tenga curiosidad acerca de la implementación de esta técnica y que le pueda servir de ayuda, no solo a nivel informativo, sino también en caso de querer implementar de forma propia esta técnica o alguna similar como puede ser el ray tracing. También está dirigido este trabajo a todo aquel que quiera realizar un renderizado de una escena 3D creada por él mismo.

Para realizar este trabajo se utiliza la metodología iterativa e incremental que permite dividir el proyecto en pequeñas tareas las cuales se van añadiendo y mejorando poco a poco. Esta metodología permite llevar un mayor control sobre lo que ocurre en el proyecto y permite ir iterando el proyecto hasta llegar al resultado que se desea.

El desarrollo de este proyecto ha sido llevado a cabo correctamente y se ha conseguido crear un editor que renderice escenas 3D utilizando la técnica del path tracing y permitiendo exportar este resultado en formato png. Al ser un editor, también se permite modificar ciertas características de los materiales para que el usuario pueda crear sus propias escenas.

Finalmente, este proyecto ha sido realmente complejo de realizar, debido en parte a que pese a que hay mucha teoría sobre él y la mayoría de fórmulas para resolver el problema de la iluminación fueron descritas hace muchísimos años, la información sobre la implementación de esta técnica y cómo hacerlo correctamente es algo más escasa. Pese a esta falta de información a nivel práctico y que la planificación del proyecto no fue la adecuada, el resultado ha sido el que se describió en los objetivos al iniciar el proyecto.

Palabras clave

Global Illumination, OpenGL, shader, path tracing, ray tracing, PBR, deferred rendering, reflexión, refracción.

Enlaces

Link al repositorio de GitHub: https://github.com/LucasPG14/Lux

Link a la release del proyecto: https://github.com/LucasPG14/Lux/releases/tag/V1.0

Link al video-teaser del proyecto: https://youtu.be/JFEBzc2ctzo

Índice de tablas

Tabla 1: Análisis DAFO Pág. 30

Tabla 2: Riesgos del proyecto y posibles soluciones Pág. 31

Índice de figuras

Figura 1.1: Imagen comparativa entre path tracing (izquierda), ray tracing (medio) y rasterización (derecha). Pág. 14

Figura 2.1: Imagen del render de una escena 3D con 1847 luces. Pág. 16

Figura 2.2: Imagen de la información que se guarda en cada una de las texturas del g-buffer. Pág. 17

Figura 2.3: Imagen de una pared de ladrillos renderizada usando PBR. Pág. 19

Figura 2.4: Escena con iluminación directa (izquierda) y utilizando la técnica de radiosity (derecha). Pág. 20

Figura 2.5: Imagen de escena 3D en Unreal sin ray tracing (izquierda) y con ray tracing (derecha). Pág. 21

Figura 2.6: Imagen de cómo se lanza un rayo desde la cámara hasta la pantalla. Pág. 22

Figura 2.7: Ejemplo de cómo funciona la técnica del ray tracing. Pág. 23

Figura 2.8: Imagen que muestra el camino que siguen los rayos originados desde la cámara. Pág. 24

Figura 2.9: Imagen renderizada con ray tracing. Pág. 25

Figura 2.10: Imagen de un renderizado con el programa Arnold. Pág. 26

Figura 2.11: Imagen renderizada en Hyperion. Pág. 26

Figura 2.12: Imagen del programa de Pixar, RenderMan. Pág. 27

Figura 2.13: Imagen renderizada con path tracing. Pág. 28

Figura 2.14: Imagen que muestra cómo se reduce el ruido a más muestras. Pág. 29

Figura 3.1: Desglose de los gastos del proyecto. Pág. 33

Figura 4.1: Ejemplo del proceso de la metodología iterativa e incremental. Pág. 34

Figura 4.2: Imagen del proceso de desarrollo del proyecto en Trello. Pág. 36

Figura 4.3: Imagen de los commits del proyecto en GitHub. Pág. 37

Figura 4.4: Desglose de los gastos totales del proyecto debido al prolongamiento del mismo. Pág. 38

Figura 5.1: Imagen de la planificación de las tareas. Pág. 41

Figura 5.2: Imagen del bucle de la aplicación. Pág. 43

Figura 5.3: Imagen del callback del scroll del ratón. Pág. 43

Figura 5.4: Imagen de las funciones para escribir en la consola. Pág. 44

Figura 5.5: Imagen de la clase base Layer y sus funciones. Pág. 45

Figura 5.6: Imagen de la clase Renderer y algunas funciones. Pág. 45

Figura 5.7: Imagen de la detección de errores en el shader. Pág. 46

Figura 5.8: Imagen de la struct necesaria para crear un framebuffer. Pág. 47

Figura 5.9: Imagen del código para crear un SSBO. Pág. 47

Figura 5.10: Imagen de cómo se cambia solo parte de la información del SSBO. Pág. 48

Figura 5.11: Imagen de cómo se crea un texture array. Pág. 48

Figura 5.12: Imagen del resultado final del libro Ray Tracing in One Weekend. Pág. 49

Figura 5.13: Imagen de cómo se juntan los vértices de todos los modelos. Pág. 50

Figura 5.14: Imagen de cómo se juntan los índices de todos los modelos. Pág. 51

Figura 5.15: Imagen de cómo se guarda la información de cada malla. Pág. 51

Figura 5.16: Imagen de cómo se extraen las normales de cada modelo. Pág. 52

Figura 5.17: Imagen de cómo se extraen las coordenadas de textura. Pág. 52

Figura 5.18: Imagen de cómo se extraen las matrices de transformación de cada modelo. Pág. 52

Figura 5.19: Imagen de cómo se extrae la textura en el shader. Pág. 53

Figura 5.20: Imagen de cómo se lee la textura de la GPU para guardar la información de esta. Pág. 53

Figura 5.21: Imagen de cómo se guarda la información de cada material. Pág. 54

Figura 5.22: Imagen de la información que se almacena sobre el objeto. Pág. 54

Figura 5.23: Imagen del ray tracing con únicamente dos muestras. Pág. 55

Figura 5.24: Imagen del ray tracing con cien muestras. Pág. 55

Figura 5.25: Imagen de cómo se envia al shader la textura con el frame anterior. Pág. 56

Figura 5.26: Imagen de la mezcla entre el frame anterior y el actual. Pág. 56

Figura 5.27: Imagen de cómo se iteran los objetos para encontrar el más cercano. Pág. 58

Figura 5.28: Imagen de cómo se recoge toda la información cuando se colisiona con un triángulo. Pág. 59

Figura 5.29: Imagen de cómo se extrae la información del material. Pág. 59

Figura 5.30: Imagen de cómo se crea la microfacet normal. Pág. 60

Figura 5.31: Imagen de cómo se obtienen los dos valores del Fresnel. Pág. 60

Figura 5.32: Imagen de cómo se calculan los pesos en función del tipo de material. Pág. 60

Figura 5.33: Imagen del proceso a seguir si el material es difuso. Pág. 61

Figura 5.34: Imagen de la fórmula de Disney para calcular el color difuso. Pág. 62

Figura 5.35: Imagen de cómo se obtiene el color difuso. Pág. 62

Figura 5.36: Imagen de un renderizado de 2000 sampleos de objetos difusos. Pág. 63

Figura 5.37: Imagen de cómo obtener la información para procesar el color de un objeto reflectivo. Pág. 63

Figura 5.38: Imagen del proceso a seguir para conseguir el color en la reflexión. Pág. 64

Figura 5.39: Imagen de un renderizado de 2000 sampleos de algunos objetos reflectivos. Pág. 64

Figura 5.40: Imagen del proceso a seguir para los objetos refractivos. Pág. 65

Figura 5.41: Imagen de la fórmula para calcular el color de un objeto refractivo. Pág. 65

Figura 5.42: Imagen del proceso a seguir para calcular el color de un objeto refractivo. Pág. 66

Figura 5.43: Imagen de un renderizado de 2000 sampleos con algunos objetos refractivos. Pág. 66

Figura 5.44: Imagen de cómo se añade el color en caso de que el objeto sea emisivo. Pág. 67

Figura 5.45: Imagen de cómo se modifica el acumulado con el color del rayo actual. Pág. 67

Figura 5.46: Imagen de cómo se crea el nuevo rayo. Pág. 68

Figura 5.47: Imagen de un renderizado de 2000 sampleos con objetos difusos, reflectivos y refractivos. Pág. 69

Figura 5.48: Imagen de un renderizado de 2000 sampleos con objetos difusos, reflectivos, refractivos y emisivos. Pág. 69

Figura 5.49: Imagen del editor con todas sus funcionalidades visibles. Pág. 70

Figura 5.50: Imagen del panel de opciones del renderizado. Pág. 71

Figura 5.51: Imagen del panel de recursos. Pág. 71

Figura 5.52: Imagen del editor con todas sus funcionalidades visibles. Pág. 72

Glosario

API: Una API es un software que permite comunicarse con otro software y compartir información entre ellos. Por ejemplo, OpenGL es una API gráfica que permite comunicarse para mandar a la tarjeta gráfica para poder dibujar en pantalla.

Global Illumination: Es un conjunto de técnicas que se encargan de simular como afecta la luz indirecta tanto en entornos 2D como 3D.

Shader: Es un programa que se ejecuta en la tarjeta gráfica.

Material: Un material es un conjunto de valores que definen cómo es el objeto que tiene el material. Es decir, define el color del objeto, cómo de rugoso es, si es metálico, etc.

Renderer: Es un programa que permite crear imágenes de escenas por ordenador. Unreal o Unity, que son motores de videojuegos, son también a su vez renderers.

Deferred rendering: Es una técnica que consiste en aplazar el cálculo de la iluminación de la escena, guardando los datos de cada objeto en diferentes texturas. De esta manera, la iluminación se calcula por cada píxel de la pantalla y no por cada vértice, lo cual disminuye considerablemente el cálculo de la iluminación, evitando que sea tan costoso.

Geometry Buffer: Utilizado en el deferred rendering. Consiste en tener 3 texturas, para almacenar la información de los objetos a renderizar. Se guardan las posiciones, las normales y el color del objeto, cada valor en una textura.

Depth Buffer: Textura que tiene codificada la profundidad de los elementos visibles en pantalla. Se representa en una escala de color unidimensional.

Pdf: Hace referencia a *Probability Density Function*, es una función que se utiliza en el path tracing para saber como afecta el color calculado debido a que los rayos que se lanzan son aleatorios y, por tanto, de esta manera se hace un cálculo de probabilidad del nivel de incidencia del color.

Shader Storage Buffer Object: Es un bloque de memoria en la GPU, similar a los uniform buffers con la diferencia de que, en este, se pueden agregar arrays de tamaño limitado. Destacar que solo se puede añadir un array de tamaño ilimitado por SSBO.

1. Introducción

1.1 Motivación

La principal motivación para realizar este trabajo es la pasión por la programación gráfica. Pese a tener algunos conocimientos en la materia, tanto de OpenGL como de cómo funcionan los shaders, la intención de este trabajo es ampliar todavía más esos conocimientos aprendiendo e intentando exprimir al máximo todo lo que se puede hacer en el apartado gráfico.

El panorama de programación gráfica ha cambiado mucho con el paso de los años, ya que es un campo que está constantemente en evolución, siempre hay algo nuevo en cuanto a optimización de técnicas ya existentes, o incluso nuevas técnicas que redefinen las utilizadas hasta ahora. Todo esto es una motivación extra para realizar este trabajo, y poder adquirir estos conocimientos para poder formarme en el campo la programación gráfica de cara a poder dedicarme a ello en el futuro.

1.2 Formulación del problema

La iluminación en las escenas 3D ya sea para realizar un renderizado o en los videojuegos, que estos son a tiempo real, es muy difícil de replicar de manera fiel a la realidad. Existen muchas técnicas para resolver este problema, pero necesitan de un hardware muy potente para poder llevarlas a cabo. En el caso de los videojuegos, como se comentaba anteriormente, al suceder a tiempo real, la dificultad para implementar estas técnicas aumenta drásticamente.

De hecho, muchas de estas técnicas de iluminación indirecta llevan muchísimos años definidas, y algunas de ellas, como el ray tracing, han empezado a ser viables para los videojuegos desde hace muy poco tiempo.

En este trabajo se trabajará la técnica del path tracing para realizar un renderer 3D. Esta técnica apenas se puede encontrar implementada de manera fiel a tiempo real a día de hoy, lo que sí se puede encontrar es esta técnica con un menor número de muestras, y utilizando otras técnicas para reducir el ruido generado en la imagen.

De esta manera, sí podemos encontrar path tracing a tiempo real, pero no utilizando esta técnica al 100% sino que se usan otros métodos para hacerla posible con el hardware actual.

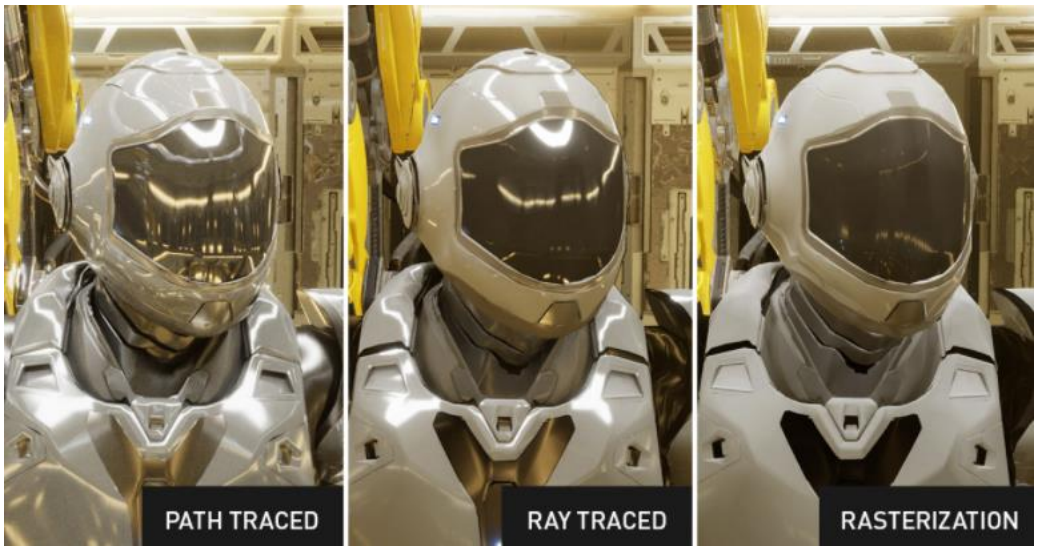

Figura 1.1
Imagen[1] comparativa entre path tracing (izquierda), ray tracing (medio) y rasterización (derecha).

En este trabajo no encontraremos una implementación a tiempo real, ya que cómo se ha comentado anteriormente es prácticamente inviable, pero sí se encontrará una implementación de esta técnica lo más fiel posible en un renderer 3D.

1.3 Objetivos generales del TFG

El objetivo general de este TFG es crear un renderer en 3D que implemente la técnica de path tracing. Con este trabajo se quiere conseguir una aplicación en la que se pueda observar esta técnica y dejar al usuario poder modificar algunas opciones del renderer para que pueda ver cómo cambia la imagen renderizada en función de esos parámetros.

Los objetivos generales de este TFG son los siguientes:

- Crear un programa de renderizado 3D usando path tracing.
- Profundizar los conocimientos en OpenGL y en shaders.
- Demostrar una gran capacidad autodidacta, ya que es un tema con poca información disponible.
- Demostrar todo lo aprendido en la carrera durante todos estos años.

1.4 Objetivos específicos del TFG

Juntamente con lo comentado en el apartado anterior, el principal objetivo específico de este trabajo es realizar un programa que permita al usuario que lo utilice renderizar una escena 3D con la mayor calidad posible, y para ello, este programa hará uso de la técnica path tracing para conseguir un resultado lo más fiel a la realidad.

[1] Imagen extraída de la siguiente página web.

Los objetivos específicos de este trabajo son los siguientes:

- Crear un programa de renderizado 3D que utilice la técnica del path tracing.
- Aprender las matemáticas que hay detrás de esta técnica y cómo aplicarlas.
- Permitir cargar modelos 3D de manera que el usuario pueda crear su propia escena 3D dentro del programa.
- Permitir al usuario modificar ciertas opciones del renderizado como, por ejemplo, el número total de muestras que quiere para su imagen.
- Permitir que el usuario pueda exportar la imagen renderizada (ya sea en formato .png o .jpg).

1.5 Alcance del proyecto

Este trabajo va dirigido a un sector muy específico. Va dirigido a todo aquél que tenga interés en la iluminación indirecta en entornos 3D, y más concretamente en el path tracing, de cómo funciona la técnica y que pueda observar cómo se va realizando el renderizado de la imagen, y que a cada frame se va reduciendo el ruido de la imagen, obteniendo así una imagen más fiel a la realidad.

Podrá utilizarlo cualquier persona que tenga interés en indagar cómo está aplicada la técnica y cómo se realiza todo el proceso, es decir, que podrá ser utilizado de manera didáctica e informativa para cualquier persona interesada en este tema. Por otra parte, también podrá ser utilizado por cualquier persona que quiera realizar un renderizado de una escena 3D con la mayor calidad posible.

Por último, de este trabajo se podrá beneficiar cualquier persona que quiera utilizar el programa para poder realizar sus propios renderizados de escenas 3D creadas por él, y también se podrá beneficiar todo aquel que quiera información sobre esta técnica, ya sea simplemente a nivel informativo, o con la intención de querer realizar este mismo proyecto o uno similar de manera propia.

2. Estado del arte/Marco Teórico/Contextualización/Estudio de Mercado

En este apartado se encuentran las técnicas que se utilizan en el proyecto, cómo funcionan, y si son lo más avanzado actualmente o hay otras alternativas a día de hoy que pueden ser más eficaces.

Actualmente, es muy difícil encontrar una implementación de path tracing a tiempo real completamente fiel a la técnica, es decir, que haya otras técnicas entre medio para combinarlas con el path tracing y así ganar tiempo con ello. Esta técnica será utilizada para crear un renderer 3D que renderice imágenes con un gran resultado gráfico. No encontraremos en este proyecto la creación de un renderer 3D con un grandísimo editor como el de algunos estudios de animación. Será un proyecto mucho más simple en ese sentido, a pesar de que evidentemente contará con un editor para que el usuario pueda crear sus propias escenas en el programa.

2.1 Deferred rendering

El deferred rendering es una técnica de renderizado que permite agilizar el cálculo de la iluminación de una escena 3D. Esto permite tener escenas 3D con una grandísima cantidad de luces, y manteniendo un tiempo de ejecución bajo.

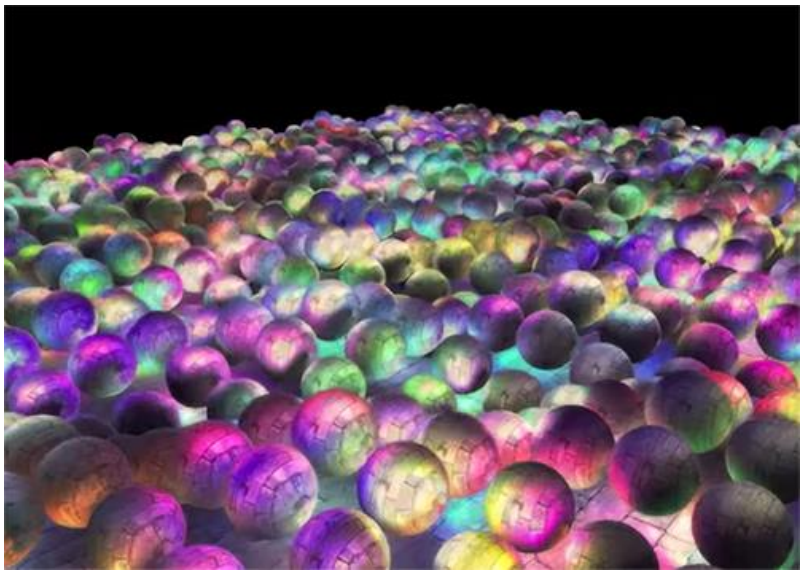

Figura 2.1
Imagen[2] del render de una escena 3D con 1847 luces.

Es una técnica que es muy utilizada en la actualidad en caso de querer renderizar escenas muy grandes y con una gran cantidad de luces, ya que con la técnica del forward rendering se perdería muchísimo tiempo.

[2] Imagen extraída de la siguiente página web.

2.1.1 Cómo funciona la técnica

Para poder hacer uso de esta técnica es necesario crear tres texturas para guardar la información de todo aquello que se va a renderizar. En una de las texturas, guardaremos las posiciones de los objetos correspondientes a ese píxel, en otra de ellas, guardaremos las normales correspondientes a ese píxel y en la última textura guardaremos el color del objeto en ese píxel. Esto es conocido como g-buffer.

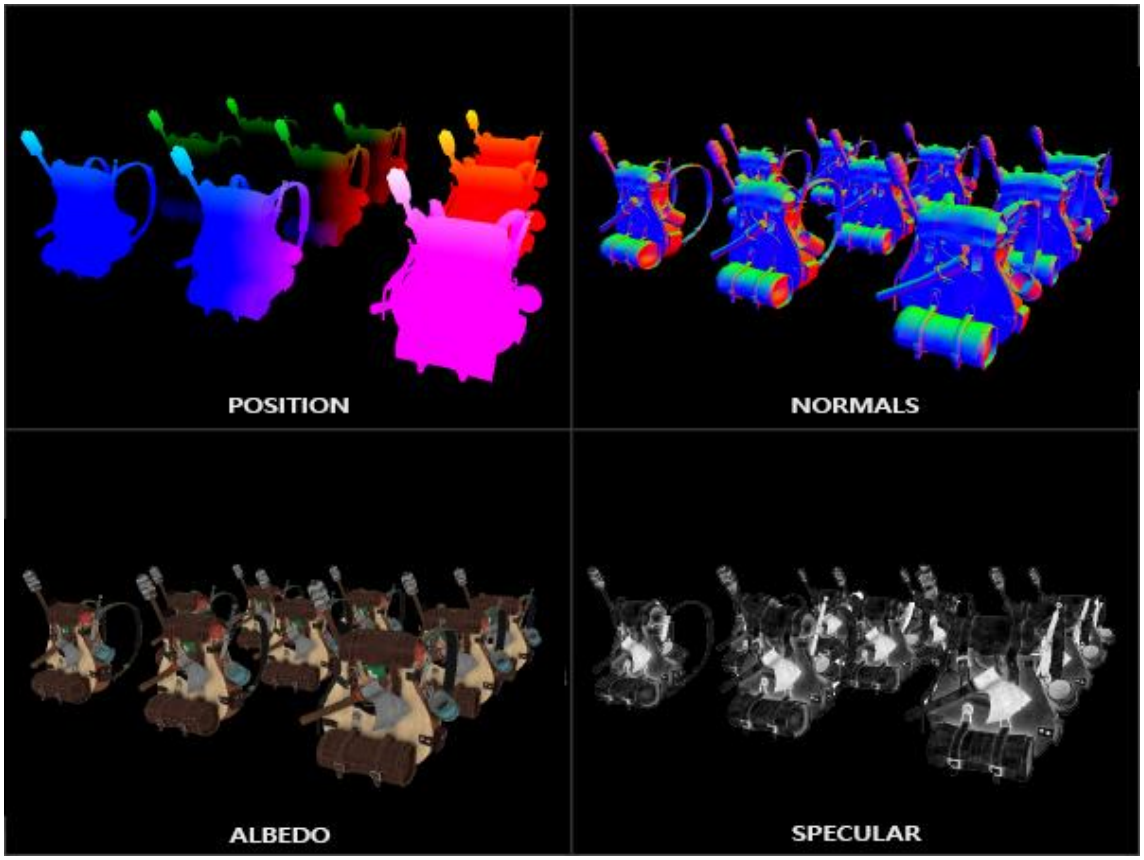

Figura 2.2
Imagen[3] de la información que se guarda en cada una de las texturas del g-buffer.

En la *Figura 2.1,* se observan cuatro texturas distintas en las que se muestran las posiciones, las normales, el color y el especular. A pesar de que en la figura vemos 4 texturas, realmente el g-buffer sólo está formado por 3 texturas. Como cada textura es en formato RGBA, el especular es un solo valor y, por tanto, se aprovecha la textura del color para almacenar el valor del especular en el componente alfa de la textura del color.

Una vez se tiene esta información, se realiza un siguiente paso de dibujado en el cual se activan estas tres texturas para poder utilizarlas en este paso. En este paso, se hace un render to texture, que es básicamente dibujar un cuadrado del tamaño de la pantalla, como si realmente estuviésemos dibujando en una textura. Con las tres texturas activadas, lo único necesario será extraer la información de las texturas de posiciones,

[3] Imagen extraída de la siguiente página web.

normales y color, y utilizar esos valores para calcular la iluminación. La iluminación se calcula de la misma manera en que se haría si se hiciera uso del forward rendering.

2.1.2 Ventajas y desventajas

Las ventajas que ofrece esta técnica es que consigue que sólo se calculen las luces por cada píxel de la pantalla, a diferencia del forward rendering. Además, todo ello lo hace con un tiempo de ejecución muy bajo, lo cual permite añadir más luces a la escena 3D.

Pese a todo, tiene ciertas desventajas. La más evidente, es que al hacer uso del z-buffer, los objetos transparentes no son soportados, a no ser que se modifique el orden de renderizado de los objetos, dejando los objetos transparentes para el final. También se podría utilizar el depth peeling para conseguir que no tenga importancia el orden de renderizado de los objetos, pero esto implica un mayor tamaño del g-buffer y del uso de batches adicionales.

2.2 Physically based rendering

El PBR es un método de gráficos por ordenador que intenta renderizar imágenes de tal manera que se modele el flujo de la luz en el mundo real. Dicho de otra manera, el PBR consigue que los materiales de una escena 3D se vean afectados por la luz de una manera realista.

Además, con el PBR se consigue este efecto sin un grandísimo coste computacional debido a las aproximaciones que hace de la función de distribución bidireccional de la reflectancia y también de la ecuación de renderizado.

En otros métodos, se intenta diferenciar entre objetos reflectantes y no reflectantes. En el caso del PBR, este asume que todos los objetos brillan, ya sea en mayor o en menor medida, pero todos los objetos brillan. Habrá materiales más planos y mates que tendrán un grado de reflexión de la luz más pequeño, y otros materiales más metalizados, que tendrán un grado de reflexión más alto.

Figura 2.3
Imagen[4] de una pared de ladrillos renderizada usando PBR.

2.3 Iluminación global

La iluminación global (en inglés Global Illumination), o también conocido como iluminación indirecta, es el nombre que se le da a un conjunto de algoritmos o técnicas que se encargan de hacer más realistas las escenas 3D. Para ello, aparte de simular la iluminación directa, es decir, la iluminación desde una fuente de luz, también simulan la iluminación de los rayos de luz que provienen de las reflexiones de estos rayos con los objetos de la escena 3D.

A diferencia de las imágenes renderizadas únicamente con iluminación directa, las imágenes renderizadas con iluminación indirecta tienen un mayor grado de realismo debido a que son algoritmos que resuelven de manera correcta y fiel el comportamiento de los rayos de luz cuando indicen en un objeto. El principal problema de estos algoritmos es que son muy costosos a nivel computacional y, por tanto, generar una imagen con una de estas técnicas requiere de mucho tiempo.

[4] Imagen extraída de la siguiente página web.

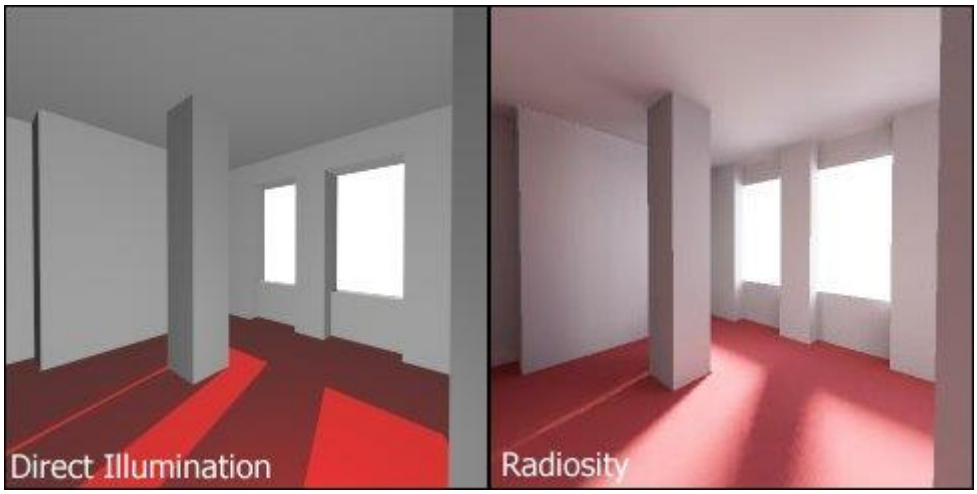

Figura 2.4
Escena[5] con iluminación directa(izquierda) y utilizando la técnica de radiosity(derecha).

En los gráficos en tiempo real, se utiliza a menudo un término llamado luz ambiental para aproximar el resultado de las inter reflexiones difusas. Pese a que no es iluminación global, sí que es cierto que a nivel computacional es sencillo y poco costoso, y en caso de querer realizar gráficos a tiempo real o en caso de tener un equipo poco potente, es una gran solución para conseguir un buen resultado gráfico.

A continuación, se enumeran varios de los algoritmos de iluminación global que existen:

- Ray tracing
- Path tracing
- Bidirectional path tracing
- Photon mapping
- Metropolis light transport
- Radiosity

A día de hoy, con el gran avance que ha habido en el hardware, podemos encontrar implementaciones a tiempo real de alguna de estas técnicas como, por ejemplo, el ray tracing, que actualmente puede verse en varios videojuegos. Esto es gracias al hardware, pero también gracias a APIs gráficas como DirectX o Vulkan, que permiten a los desarrolladores sacar el máximo del equipo. Estas APIs permiten la aceleración por hardware, lo cual es algo muy beneficioso a la hora de implementar técnicas de iluminación indirecta como las mencionadas anteriormente.

2.2.1 Ray tracing

2.2.1.1. Historia

El concepto de ray tracing fue por primera vez utilizado por Albrecht Dürer en el siglo XVI, a quién se le atribuye la invención del mismo. Dürer determinó varias técnicas para proyectar escenas 3D en una imagen.

[5] Imagen extraída de la siguiente página web.

Varios siglos después, en 1968, se utilizó por primera vez el ray tracing en ordenador para generar imágenes sombreadas. El encargado de ello fue Arthur Appel, que utilizó el ray tracing para la parte visual principal, y después trazó rayos secundarios hacia las fuentes de luz para determinar si ese punto estaba en sombra o no.

Tres años después, en 1971, Goldstein y Nagel publicaron *3D Visual Simulation,* donde se utilizaba el ray tracing para crear imágenes simulando el proceso fotográfico a la inversa. Lanzaban un rayo a cada píxel de la pantalla que iba destinado a la escena 3D, y aquel objeto con el que intersecara primero, sería el visible. Esta es una técnica no recursiva, que a día de hoy se conoce como *ray casting.*

Turner Whitted fue el primero que mostró un ray tracing recursivo para la reflexión en un espejo y para la refracción en objetos translúcidos con un ángulo determinado por un índice de refracción. También utilizó el ray tracing para resolver el antialiasing. En el año 1979, realizó una película llamada *The Compleat Angler,* en la que utilizaba ray tracing recursivo. Esta muestra de ray tracing de Whitted supuso un gran cambio en la visión que había del renderizado en aquel entonces, ya que cambió el concepto de resolución del problema, el problema ya no era determinar la visibilidad de la superficie del objeto, sino del transporte de la luz.

Durante muchísimos años, se ha falseado la iluminación global en la mayoría de películas hechas por ordenador haciendo uso de luces adicionales. Gracias al ray tracing, esta situación ha ido cambiando con el paso de los años, ya que esta técnica permite simular el transporte de la luz basado en la física.

Figura 2.5
Imagen[6] de escena 3D en Unreal sin ray tracing (izquierda) y con ray tracing (derecha).

[6] Imagen extraída de Unreal Engine.

En la *Figura 2.2*, se puede observar la diferencia entre una misma escena renderizada con y sin ray tracing. El efecto es muy notable a primera vista, ya que se puede comprobar como las reflexiones son muchísimo más realistas cuando se renderiza con ray tracing.

2.2.1.2. Cómo funciona la técnica

A continuación, se explicará en que consiste la técnica del ray tracing, explicando detalladamente cada paso.

Primero de todo, se lanza un rayo hacia cada píxel de la pantalla. Para generar un rayo, es necesario un origen y una dirección. En este caso, el origen del rayo es la posición de la cámara, y la dirección será aleatoria para cada píxel.

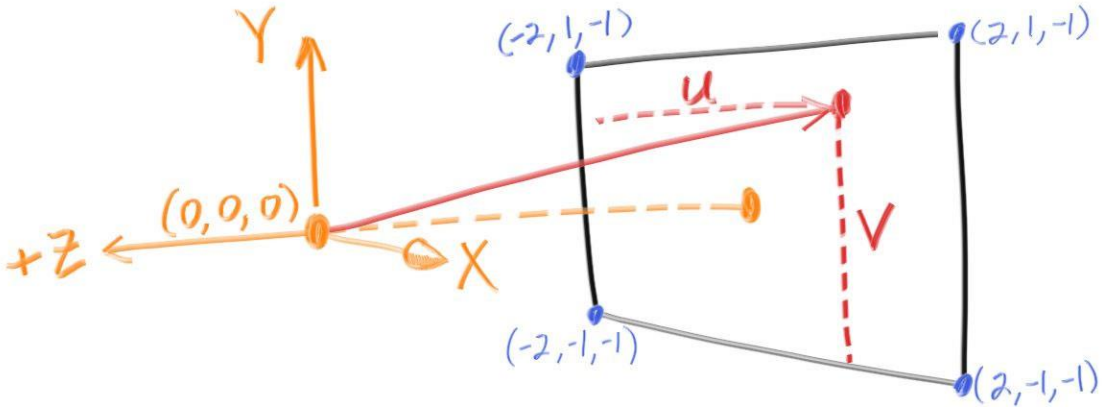

Figura 2.6
Imagen[7] *de cómo se lanza un rayo desde la cámara hasta la pantalla.*

Una vez se tiene generado el rayo con el origen y la dirección, se comprueba este rayo con todos los objetos que tiene la escena 3D para comprobar con cuál de ellos ha intersecado y quedarse con el más cercano de todos.

[7] Imagen extraída de la siguiente página web.

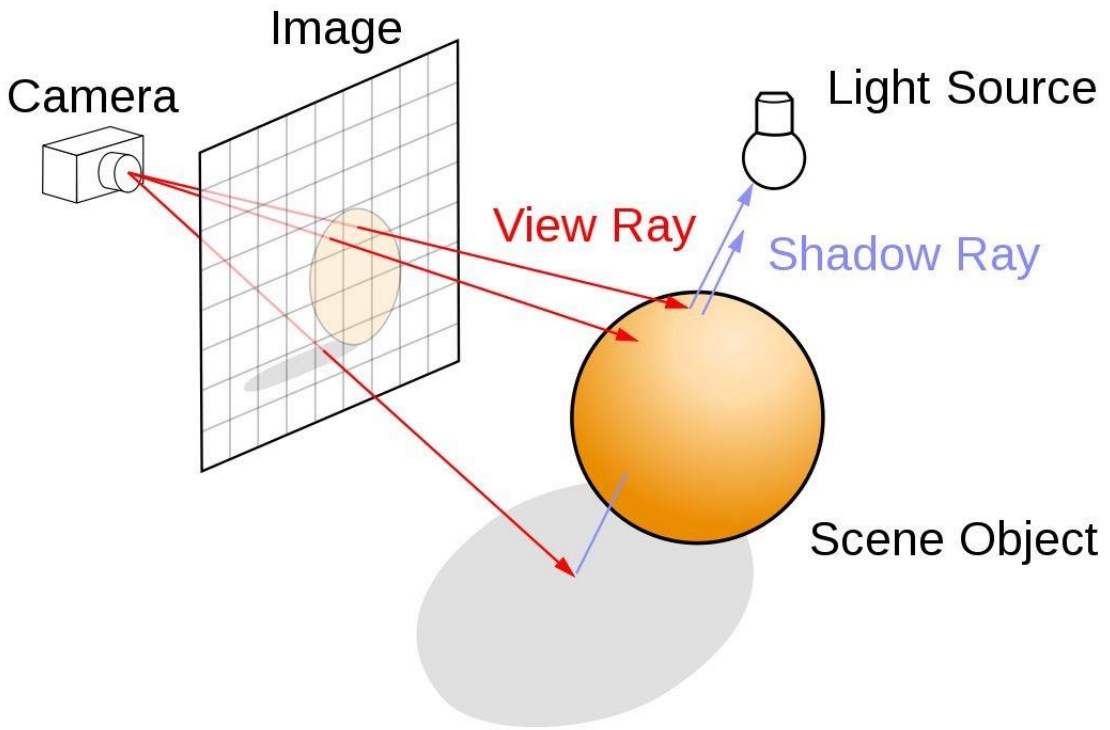

Figura 2.7
Ejemplo[8] de cómo funciona la técnica del ray tracing.

Una vez tenemos el objeto más cercano con el que ha intersecado el rayo, tenemos que generar otro rayo que proviene del rebote del rayo original con el objeto. Para generar este rayo, se necesita lo mismo que para el anterior, es decir, un origen y una dirección. En este caso, el origen del rayo no será la posición de la cámara, sino que será el punto en el espacio donde el rayo original ha intersecado con el objeto. Para conseguir este punto es necesario utilizar la siguiente ecuación: $o + t * d$.

Con esta ecuación se puede conseguir cualquier punto en el espacio que haya intersecado con el rayo sabiendo la distancia a la que está el objeto en cuestión. En este caso, como es necesario saber la distancia para quedarse con el objeto más cercano, ya se tiene toda la información necesaria para resolver esa ecuación, donde o es el origen del rayo, t es la distancia que hay desde el rayo hasta el objeto, y d es la dirección del rayo. Una vez se tiene este punto, ya se tiene el origen del rayo rebotado, y sólo falta saber la dirección en la que sale rebotado el rayo.

Para ello son necesarias varias variables, empezando por conocer la normal en el punto donde el rayo y el objeto han intersecado. Junto con la normal y una dirección aleatoriamente generada, se puede obtener la dirección en la que va a ir el rayo rebotado, pero para ello también es necesario saber el tipo de material del objeto, ya que no funciona de la misma manera un material reflexivo que uno refractivo. También es necesario obtener el color del objeto en el punto de intersección entre el rayo y el objeto para poder calcular el color final del píxel.

[8] Imagen extraída de la siguiente página web.

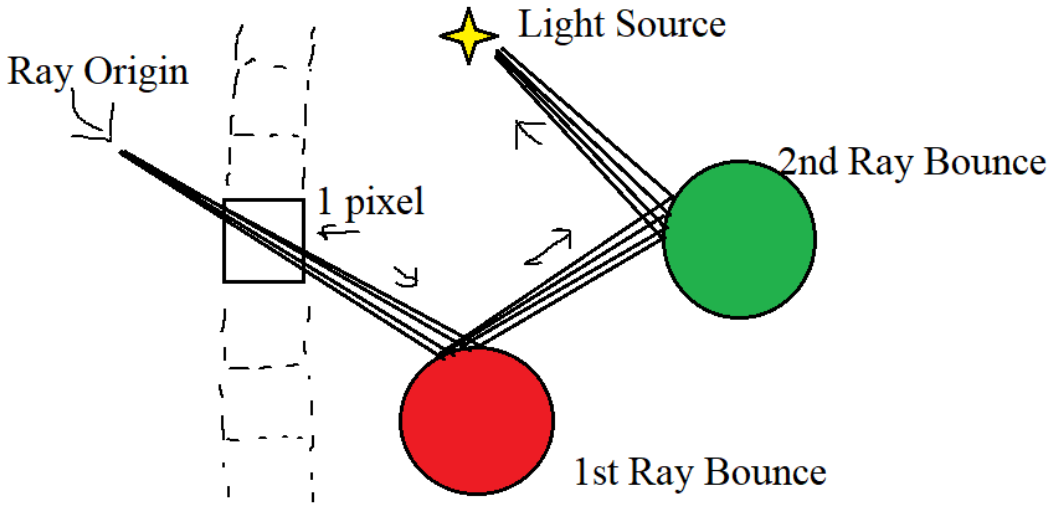

Figura 2.8
Imagen[9] que muestra el camino que siguen los rayos originados desde la cámara.

Una vez se tiene toda esa información, y se tiene la dirección del rayo rebotado en función del material del objeto, sólo queda repetir el proceso de forma cíclica volviendo a comprobar todos los objetos de la escena con el nuevo rayo generado por la intersección anterior. Cada vez que se genera una intersección se va acumulando el color de los objetos hasta que el proceso llega a su fin, y el color resultante será el color final de ese píxel.

Para finalizar el proceso pueden ocurrir varias cosas, la primera es que el rayo no interseque con ningún objeto de la escena y, por tanto, el proceso acabará ahí. Otra posibilidad es que el rayo rebote infinitamente hasta que el color resultante del píxel sea negro, lo cual querrá decir que la energía se ha absorbido totalmente.

Por último, también existe la opción de fijar un límite de rebotes para evitar que el proceso de computación se demore mucho, y a pesar de que añadir un límite de forma manual por el usuario implica no realizar la técnica de manera fiel, la diferencia es prácticamente imperceptible y, por tanto, en caso de estar realizando ray tracing en tiempo real, esta puede ser una buena posibilidad para acelerar el proceso.

[9] Imagen extraída de la siguiente página web.

Figura 2.9
Imagen[10] renderizada con ray tracing.

2.2.2 Path tracing

2.2.2.1. Historia

En 1986, James Kajiya presentó la ecuación de renderizado y su uso en los gráficos por ordenador. El path tracing se introdujo para dar una solución numérica a la integral de la ecuación de renderizado. Varios años más tarde, Lafortune propuso varias mejoras a esta técnica como, por ejemplo, el bidirectional path tracing.

Desde hace varios años, el hardware ha avanzado mucho y eso ha permitido que el path tracing genere más interés al dejarse de ver como algo inalcanzable a nivel de hardware, debido al enorme coste computacional que tiene.

En la industria del cine, el path tracing ha tenido una gran importancia. Monster House fue el primer largometraje realizado totalmente con path tracing, utilizando el renderer Arnold[11]. Empresas como Walt Disney Animation Studios o Pixar Animation Studios tienen sus propios renderers, llamados Hyperion y RenderMan, respectivamente.

Arnold es un renderer desarrollado por principalmente Marcos Fajardo, y a pesar de que ya tenía varias versiones del renderer, no es hasta 2009, que funda la empresa Solid

[10] Imagen extraída de la siguiente página web.
[11] Página web de Arnold.

Angle SL. Varios años después, en el año 2016, Autodesk compra la empresa y pasa a ser el propietario de Arnold.

Figura 2.10
Imagen[12] de un renderizado con el programa Arnold.

En el caso de Hyperion, el renderer de Walt Disney, llevan utilizándolo desde que hicieron la película Big Hero 6, en el año 2014.

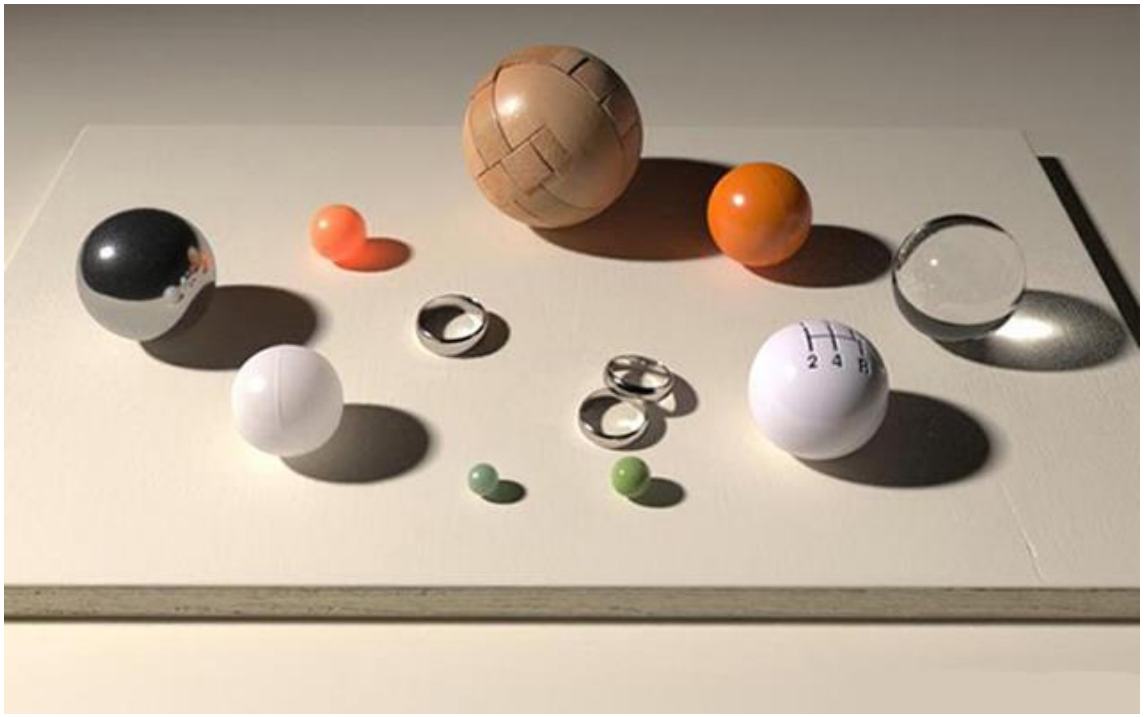

Figura 2.11
Imagen[13] renderizada en Hyperion.

[12] Imagen extraída de la siguiente página web.
[13] Imagen extraída de la siguiente página web.

Por otra parte, RenderMan es un programa de uso comercial, desarrollado por Pixar. RenderMan ha sido utilizado en películas como Toy Story, Jurassic Park o Avatar, entre muchas otras.

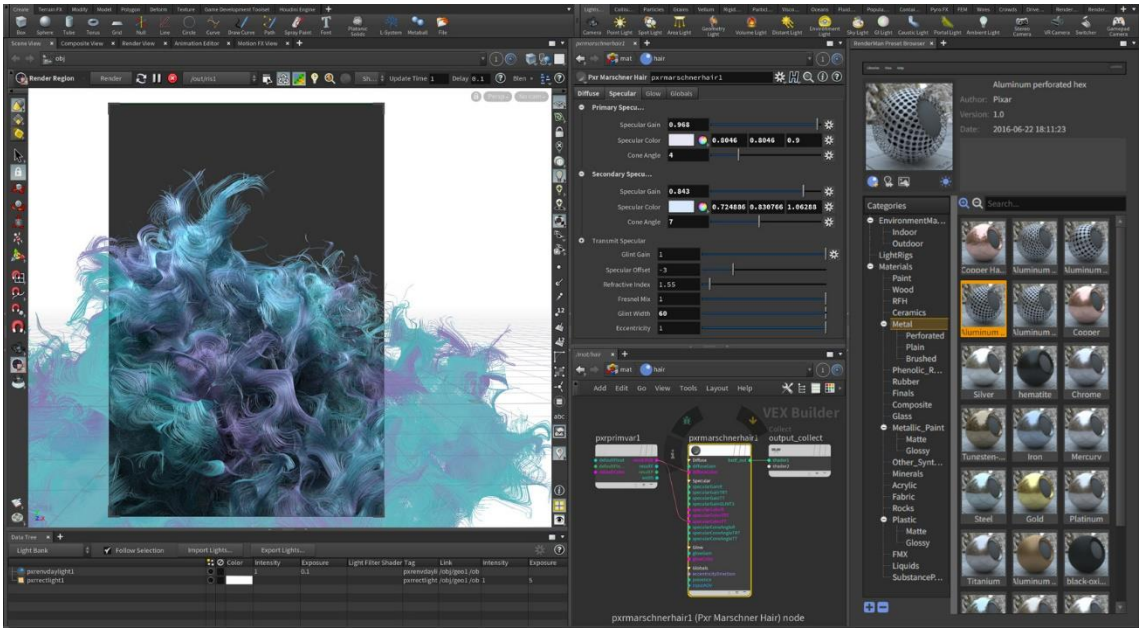

Figura 2.12
Imagen[14] del programa de Pixar, RenderMan.

2.2.2.2. Cómo funciona la técnica

La técnica del path tracing no difiere mucho del ray tracing, de hecho, son bastante similares. En el path tracing también se trazan rayos desde la posición de la cámara hacia cada píxel de la pantalla, y se va comprobando con los objetos de la escena 3D para quedarse con el objeto más cercano. Y una vez hecho ese proceso, se repite nuevamente de forma cíclica hasta finalizar el proceso.

La lógica detrás del path tracing es la misma que en el ray tracing, pero en el path tracing se realizan cientos de muestras de un solo píxel, para luego juntarlo haciendo una media de todos. Además, también cambia la forma en la que se calcula el color resultante.

Para calcular correctamente el color en el path tracing hay que aplicar la siguiente fórmula: *Emittance + (BRDF * Incoming * CosTheta / P).* Cada variable proviene de las siguientes fórmulas:

- Emittance: Es el color del material del objeto en el punto en el que ha intersecado con el rayo.
- BRDF: Para calcularlo es necesario dividir el color reflectante del material del objeto entre el número pi.
- Incoming: Al ser recursivo, incoming es el valor final del color obtenido con el rayo rebotado.

[14] Página web de RenderMan.

- CosTheta: Es el ángulo que se obtiene haciendo el producto escalar entre la dirección del nuevo rayo rebotado y la normal de la superficie del objeto en el punto donde han intersecado.
- P: Es la probabilidad del nuevo rayo. Para obtener este valor sólo hay que dividir 1 entre 2 por el número pi.

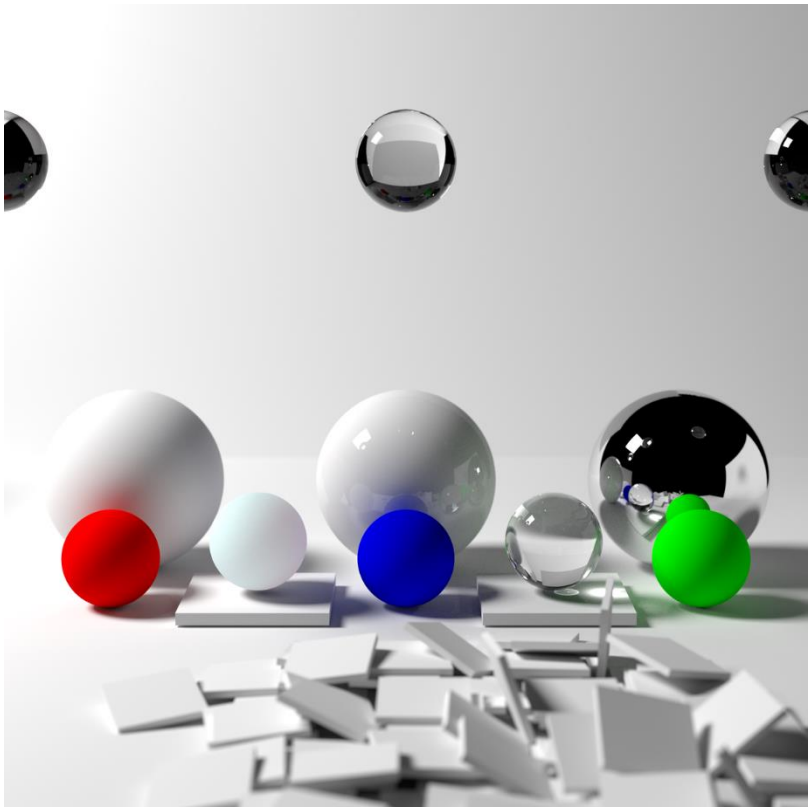

Figura 2.13
Imagen[15] renderizada con path tracing.

Como se puede observar en la *Figura 2.13,* las imágenes producidas con path tracing ofrecen un resultado gráfico impresionante, y con una simulación de la iluminación lo más parecida posible a la realidad. El gran problema de esta técnica es el altísimo coste computacional que tiene, ya que se necesitan muchísimas muestras por píxel para poder obtener un resultado como el que se puede observar en esa imagen.

El problema principal por el cual esta técnica necesita muchas muestras, es porque genera mucho ruido y, por tanto, necesita muchas muestras para poder generar una media y obtener el color correspondiente al píxel en cuestión. Para que una imagen se aprecie bien, y no se note el ruido, se necesitan fácilmente más de 5000 muestras, lo cual es muchísimo aun con el hardware que se tiene a día de hoy.

[15] Imagen extraída de la siguiente página web.

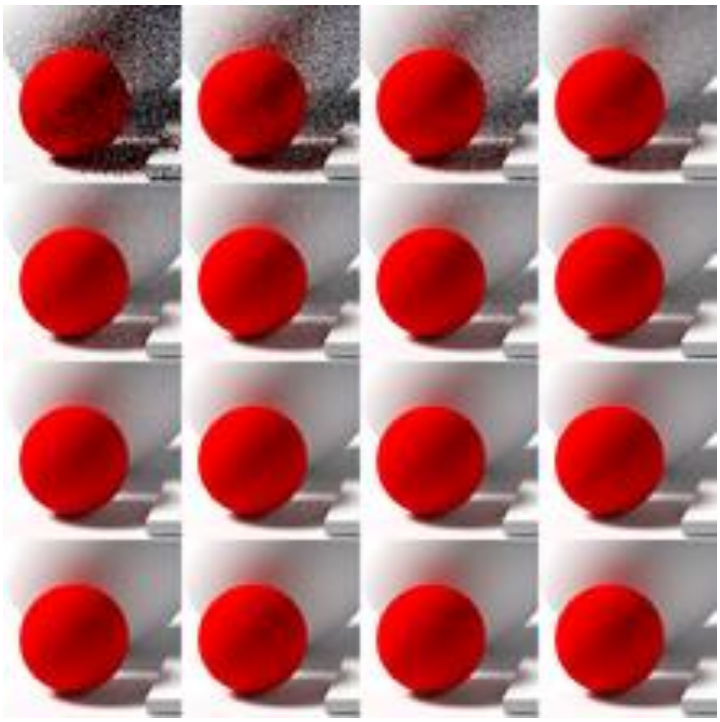

Figura 2.14
Imagen[16] que muestra cómo se reduce el ruido a más muestras.

En la *Figura 2.14,* observamos la repetición de la misma imagen, cada vez con menos ruido. Empieza con una sola muestra, y va doblando la cantidad de muestras de izquierda a derecha. En la última imagen, abajo a la derecha, hay un total de 32768 muestras. Esta imagen demuestra el alto coste de esta técnica, ya que, por ejemplo, la primera foto de la tercera fila, tiene un total de 256 muestras, y aun y así se percibe bastante el ruido en la imagen.

[16] Imagen extraída de la siguiente página web.

3. Gestión del proyecto

3.1. DAFO

Este proyecto tiene grandes ventajas y desventajas, ya que es un tema muy interesante y de gran importancia actualmente. Por ello, se mostrará a continuación, una tabla en la que se analizan las fortalezas, debilitades, oportunidades y amenazas del proyecto. Esto es conocido como análisis DAFO.

	Positivos	**Negativos**
Origen Interno	**Fortalezas** • Conocimientos previos en OpenGL y shaders. • Acostumbrado a trabajar de forma autodidacta. • Realización de un motor con C++ y OpenGL anteriormente.	**Debilidades** • Falta de experiencia en ray tracing y sus elementos principales. • Pérdida de motivación fácilmente cuando las cosas no van correctamente. • Conocimiento escaso en la parte más avanzada de OpenGL y de todo lo que se puede realizar.
Origen Externo	**Oportunidades** • Implementación de una técnica importante de cara al futuro. • Aprender las matemáticas que hay detrás de esta técnica. • Al haber pocas implementaciones, podría convertirse en una herramienta para futuras personas interesadas en este tema.	**Amenazas** • Falta de información a nivel práctico acerca del tema. • Falta de tiempo a causa de otras asignaturas y prácticas. • Sólo una persona se encargará de este proyecto.

Tabla 1
Análisis DAFO

3.2. Riesgos y plan de contingencias

Como todo gran proyecto, puede haber situaciones en las que se tambalee todo y resulte difícil llevarlo a cabo, pero es necesario mantener la calma y analizar los posibles problemas que pueden llegar a ocurrir.

A continuación, se mostrará una tabla en la que se han analizado los riesgos que pueden suceder durante el proyecto, y se ofrecen soluciones para ellos. Más tarde, se explican más detalladamente todos y cada uno de los riesgos analizados junto a sus soluciones también explicadas de forma más detallada.

Riesgo	Solución
Pérdida de motivación cuando las cosas no van bien.	Tratar de desconectar para luego poder avanzar con la cabeza más centrada.
El ordenador deja de funcionar y/o se estropea.	Tener siempre una copia de seguridad del documento ya que, de la parte práctica, al estar en GitHub no hay problema.
Proyecto muy complejo.	Organizar bien las tareas y desglosarlas correctamente en tareas más pequeñas para ver de forma más clara los avances que se van realizando en el proyecto.
Documentación escasa.	Resolver dudas preguntando en diferentes foros y chats de internet.

Tabla 2

Riesgos del proyecto y posibles soluciones

3.2.1. Motivación

La pérdida de motivación cuando las cosas no van bien, o te sientes encallado son bastante habituales y afectan bastante al desarrollo de un proyecto. Por tanto, habrá que buscar alguna manera de minimizar ese efecto, ya sea saliendo a la calle a desconectar, practicar deporte o cualquier otro ocio que permita desconectar y recargar las pilas para poder afrontar los problemas con mayor energía y sin llegar a entrar en la desesperación, ya que llegar a ese punto, supondría un golpe muy negativo para el desarrollo del proyecto.

3.2.2. Problemas de hardware

Otro problema que nos podríamos encontrar en este trabajo es que se estropeasen los periféricos con los que se realizan este trabajo. En ese caso, habría que tener una copia de seguridad del trabajo escrito para evitar perderlo, ya que de la parte práctica no habría problema ya que se utiliza GitHub y, por tanto, está en la nube. A raíz de este problema, no se podría trabajar y habría que buscar una alternativa como, por ejemplo, ir a una biblioteca o al aula de estudio de la universidad para poder seguir desarrollando el proyecto mientras se repara el equipo principal con el que se realiza el trabajo.

3.2.3. Proyecto muy complejo

Este proyecto requiere de muchísimo tiempo para realizarse ya que es un tema muy complejo y también requiere de una cierta experiencia tanto en la programación gráfica como con la API gráfica de OpenGL. A diferencia de otras APIs gráficas como pueden ser DirectX o Vulkan en las cuales es algo más sencillo tener toda la información de la escena, en OpenGL es necesario explorar otras alternativas bastante más complejas para poder tener toda la información de la escena 3D, ya sea mallas, materiales o texturas, para poder llevar a cabo esta técnica.

Para resolver este problema, se hará un gran trabajo de organización de todo lo que se necesita para realizar este proyecto, dividiendo el trabajo en tareas más específicas e ir avanzando poco a poco ya que si se trabaja en grandes bloques y grandes tareas, el avance del trabajo será muchísimo más lento.

3.2.4. Documentación escasa

Este es uno de los grandes inconvenientes de este trabajo, y es que la documentación que hay acerca del tema es bastante escasa. Si bien es cierto que de la parte teórica sí que hay información, de la parte práctica, que sería la implementación de la técnica, apenas encontramos ejemplos en internet y algunos de los ejemplos que se encuentran en internet son con otra API gráfica, como puede ser Vulkan, que tiene un pipeline en específico para realizar esta técnica y que facilita la tarea en ese aspecto.

En este caso no hay una solución al 100% y, por tanto, se dependerá mucho de la poca información que se pueda conseguir preguntando en diferentes foros de internet como puede ser Discord[17] o Reddit[18].

3.3. Análisis inicial de costos

Los costes de este trabajo se pueden resumir en 4 grandes bloques: costes de obra, software, hardware y administrativo.

En el primer bloque, que sería el de costes de obra, se encuentran los gastos del equipo que realizará este trabajo. En este caso, el trabajo será realizado únicamente por una persona y, por tanto, sólo se tendrá que pagar el sueldo de un programador, que cobrará 1.800€ mensualmente.

En el segundo bloque se encuentra el gasto en los diferentes programas que son necesarios para llevar a cabo este trabajo. El gasto en este apartado será de 0€ ya que sólo se utilizarán dos programas y los dos son gratuitos. El primer programa que se utilizará es Visual Studio, que es un IDE con el que se realizará todo el proyecto. El segundo y último programa que se utilizará es GitHub, que será donde estará guardado el proyecto. GitHub es una plataforma en la nube para almacenar principalmente código fuente de programas de ordenador.

En el tercer bloque se encuentra el hardware necesario para la realización de este proyecto. Será necesario un ordenador potente para poder trabajar correctamente ya que la técnica que se implementará requiere de un gran equipo, por tanto, serán necesarios alrededor de 1.500€ para tener un buen equipo con el que realizar el proyecto. Por otra parte, también será necesario el mantenimiento mensual de este equipo, que se valora en unos 100€ mensuales.

Por último, en el cuarto bloque se encuentran los gastos administrativos. Estos gastos son el alquiler, que se cifra en unos 750€ mensuales y también será necesario pagar agua, luz y gas, por un valor de 50€ mensuales.

Todos estos gastos se prolongarán por un período de seis meses, que es el tiempo que se espera que dure este proyecto. En total, serán necesarios 15.000€ para poder llevar a cabo este proyecto.

[17] Página web de Discord.
[18] Página web de Reddit.

	Mes 1	Mes 2	Mes 3	Mes 4	Mes 5
Programador	1.800€	1.800€	1.800€	1.800€	1.800€
Costes de obra	1.800€	1.800€	1.800€	1.800€	1.800€
Visual Studio	0€	0€	0€	0€	0€
GitHub	0€	0€	0€	0€	0€
Software	0€	0€	0€	0€	0€
1 ordenador	1.500€	0€	0€	0€	0€
Mantenimiento del equipo	100€	100€	100€	100€	100€
Hardware	1.600€	100€	100€	100€	100€
Alquiler	750€	750€	750€	750€	750€
Agua, luz, gas	50€	50€	50€	50€	50€
Costes totales	800€	800€	800€	800€	800€
Total	4.200€	2.700€	2.700€	2.700€	2.700€
				Total	15.000€

Figura 3.1
Desglose de los gastos del proyecto.

En la *Figura 3.1*, se puede observar una tabla con los diferentes gastos desglosados en meses, en los que se observa el coste individual de cada recurso, el coste total de cada bloque, y el coste total del proyecto.

4. Metodología

Este proyecto se divide en dos grandes bloques, la parte práctica, que es el desarrollo de un renderer 3D con path tracing y, por otra parte, está la parte teórica que es la búsqueda de información y documentación de la parte práctica y, por ende, el desarrollo de este documento. Debido a que la parte más importante y de mayor peso en este trabajo es la parte práctica, la metodología escogida es la más acorde para el desarrollo de la misma, que es la metodología iterativa e incremental.

La metodología iterativa e incremental consiste en planificar el trabajo en tareas más pequeñas, creando de esta manera pequeños proyectos que se van entregando y poco a poco aportan un resultado al producto final. Estos pequeños proyectos se llaman iteraciones. Por tanto, con esta metodología se tienen varias iteraciones, las cuales se realizan en un período de tiempo determinado, que normalmente suele ser corto, que lo que hacen es ir aumentando y mejorando el proyecto. Se realizan pequeñas entregas de cada una de las iteraciones y cada una de estas, parte de base de las iteraciones realizadas anteriormente.

Con esta metodología se consigue un mayor control de las tareas más importantes del proyecto, ya que al dividirlas en tareas más pequeñas e ir mejorando a cada iteración, todas las partes importantes del proyecto están implementadas desde el principio y se van mejorando poco a poco, lo cual resulta en que es más improbable que una de estas tareas pueda dar muchos problemas que si se añadiese desde cero con el proyecto muy avanzado, con lo cual esta metodología es clave para la realización de este proyecto.

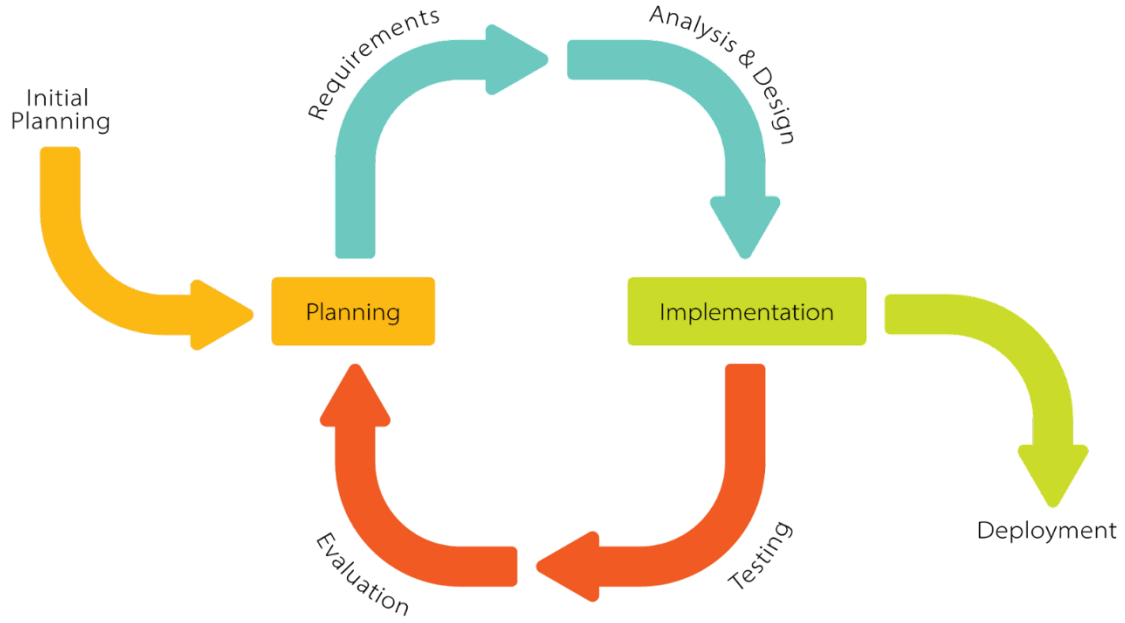

Figura 4.1
Ejemplo[19] del proceso de la metodología iterativa e incremental.

[19] Imagen extraída de la siguiente página web.

Como se puede observar en la *Figura 4.1*, cada iteración consta de cuatro fases básicas, que se realizan de forma cíclica. Estas fases son las siguientes:

- Planificación: En esta fase se detalla todo lo necesario para esta iteración, que es lo que se va a realizar y cómo.
- Implementación: Es la fase en la cual se desarrolla todo lo planeado en la fase de planificación.
- Testeo: Esta fase sirve para probar las tareas implementadas en esta iteración y comprobar que todas ellas funcionan correctamente tanto de manera individual como de manera colectiva con el resto de tareas ya implementadas en otras iteraciones.
- Evaluación: Se evalúa el resultado de las tareas para decidir si es el correcto y esperado y también se revisa que la fase de testeo haya ido bien y no haya problemas en las tareas implementadas.

Una vez finalizado todo este proceso, se pasaría a la fase de lanzamiento, en la cual se da por finalizada esta iteración. A esta fase sólo se llegará en caso de que las tareas cumplan con el objetivo marcado en la planificación de la iteración y no genera ningún problema ni tampoco tiene errores. En caso contrario, se volvería a iterar hasta que se solucionen los problemas y errores de la tarea.

4.1. Herramientas de monitorización del proyecto

Para poder gestionar este proyecto de manera correcta y llevar a cabo la metodología explicada en el apartado anterior, se utilizarán diferentes softwares que se explicarán a continuación.

4.1.1. Trello

La primera herramienta que se utiliza para monitorizar las tareas a realizar es Trello[20]. Trello permite crear todas las listas que el usuario quiera y asignar cartas en cada una de ellas. Estas cartas tienen varias funciones para hacer más fácil la gestión de los proyectos. A cada tarea se le puede asignar una fecha de finalización de la misma y también se le pueden añadir tags. Un tag es una manera de marcar las tareas a aquello que quiera el usuario, es decir, en caso de tener un proyecto grande con muchos campos distintos, se puede crear un tag para cada campo y asignar cada uno a la tarea correspondiente. Además, cada carta admite más de un tag, así que en caso de tener un campo muy amplio que se pueda dividir en subcampos, a una carta se le puede añadir un tag que haga referencia al campo principal, y a su vez un tag que haga referencia al subcampo.

[20] Página web de Trello.

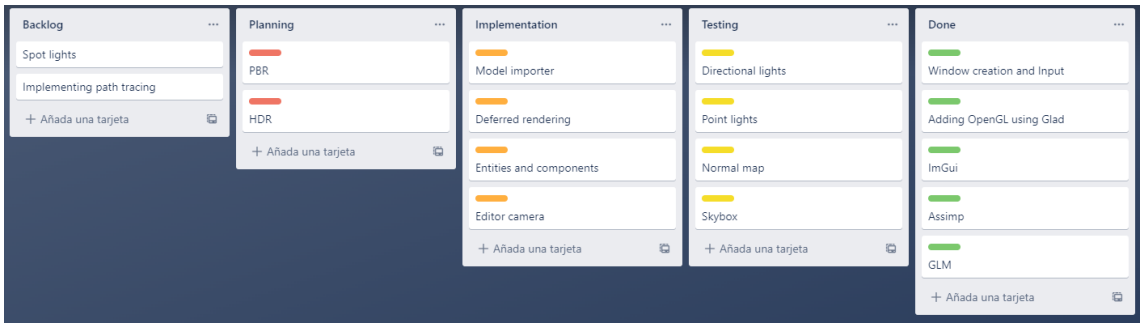

Figura 4.2
Imagen del proceso de desarrollo del proyecto en Trello.

Hay otras herramientas como HacknPlan[21] que también funcionan de manera similar a Trello y que ofrecen un resultado parecido o prácticamente idéntico a ella. La razón principal por la que se ha escogido Trello y no HacknPlan, a pesar de haber trabajado anteriormente con las dos, es por una mayor comodidad con la interfaz de Trello respecto a la de HacknPlan, y también porque aun habiendo trabajado con las dos herramientas, se ha trabajado en mayor medida con Trello que con HacknPlan y, por tanto, se domina mejor Trello que no HacknPlan.

4.1.2. GitHub

GitHub[22] es otra de las herramientas que se utiliza en este proyecto. GitHub es una plataforma que se utiliza en su mayoría para crear y alojar código fuente de programas de ordenador mediante repositorios.

GitHub cuenta con un sistema de commits para poder monitorizar todos los cambios que se realizan en el proyecto. Un commit es cada vez que se publica un cambio en el repositorio. Los commits necesariamente han de tener un título, y de manera opcional, pueden tener también una descripción de los cambios que se han realizado. En caso de que hubiera algún fallo en algún commit, se puede revertir hasta una versión estable del proyecto. Esto es de gran ayuda para el proyecto, ya que siempre estará disponible el historial con todos los commits realizados, y si se añade una descripción en cada uno de ellos acerca de los cambios realizados en ese commit, facilitará muchísimo la tarea en caso de que se tenga revertir hacia algún commit más antiguo si sucede algún problema.

[21] Página web de HacknPlan.
[22] Página web de GitHub.

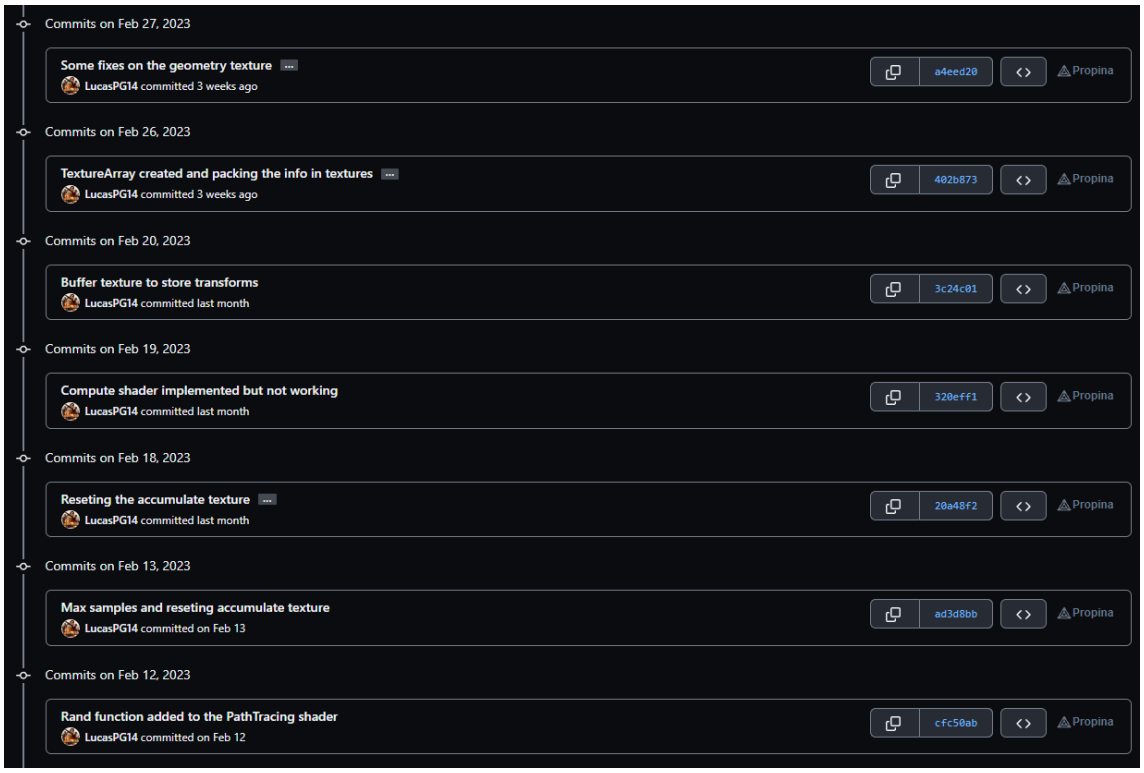

Figura 4.3
Imagen de los commits del proyecto en GitHub.

Hay varias maneras de poder publicar estos commits en GitHub, pero en este proyecto se utiliza la herramienta de GitHub Desktop para poder realizar todos estos cambios y trabajar cómodamente con el entorno de GitHub.

4.2. Herramientas de validación

Para validar las tareas realizadas, en Trello hay una columna de *Testing*, como se puede observar en la *Figura 4.2,* en la cual estarán todas las tareas que ya han sido implementadas, y se ha de comprobar exhaustivamente que cumplan con todos los requisitos.

Una vez la tarea haya sido probada y se haya llegado a la conclusión de que no genera ningún problema en el programa ni tampoco con otras tareas, se dará por finalizada.

En caso de que se encuentren problemas en esa tarea, o algo no funcione como se desea, la tarea se moverá de la columna de testing y volverá a la de desarrollo, donde se quedará hasta que se arregle el problema. Una vez se solucione el problema, la tarea se volverá a añadir a la columna de testing y se volverá a probar que todo funcione correctamente. De esta manera creamos un sistema cíclico en el que una tarea no finaliza hasta que está 100% completada y no genera problemas.

De esta manera se pueden validar las tareas correctamente y permite llevar un gran control sobre el proyecto, y también permite que el proceso sea sencillo, ya que, si se encuentra un problema, no se genera una nueva tarea, sino que sólo hay que desplazar la tarea en la que se ha encontrado el problema a la fase anterior.

Por otra parte, en GitHub también se puede llevar un control de versiones, de tal manera que siempre habrá una versión estable del programa. Además, se pueden crear varias ramas para cada campo a desarrollar. De esta manera, se trabaja en una rama exclusiva para la tarea que se quiere implementar, y cuando se tiene conocimiento de que esta tarea funciona al 100%, entonces se puede hacer un merge a la rama principal. Esto permite tener un grandísimo control del estado del proyecto, y también en caso de ocurrir un problema, permite saber fácilmente de donde viene ese problema.

4.3. Modificaciones

Respecto a la metodología a seguir, no ha habido ningún cambio, se sigue utilizando Trello como herramienta para organizar las tareas a realizar y tampoco ha habido ningún cambio acerca de cómo se gestionan las tareas en caso de completarse o de que sucediera algún fallo.

En cuanto a la planificación, aquí sí que ha habido cambios que han provocado que el proyecto se haya demorado más de lo que se tenía planeado. En concreto se ha demorado 3 meses, lo que cual influye en el coste total del proyecto.

Principalmente el problema ha sido que la planificación no ha sido la adecuada a la hora de llevar a cabo este proyecto. También ha habido determinadas cosas que se han querido hacer de una manera y posteriormente se ha tenido que cambiar de método, bien sea porque el resultado que se conseguía no era el que se esperaba para el proyecto, o bien porque el rendimiento no era óptimo. Por ejemplo, al principio de todo se quiso enviar toda la información mediante uniforms, lo cual no era óptimo ni tampoco se obtenía el resultado esperado. Seguidamente, esto se cambió por texturas, es decir, se enviaba toda la información en texturas, con este método sí se conseguía el resultado esperado, pero era poco óptimo ya que se perdía mucho tiempo extrayendo la información de las texturas. Finalmente se decidió utilizar SSBOs para enviar toda la información de la escena a la GPU. Con este método se conseguía el resultado que se esperaba, y a la vez era mucho más óptimo que los probados anteriormente.

	Mes 1	Mes 2	Mes 3	Mes 4	Mes 5	Mes 6	Mes 7	Mes 8
Programador	1.800€	1.800€	1.800€	1.800€	1.800€	1.800€	1.800€	1.800€
Costes de obra	1.800€	1.800€	1.800€	1.800€	1.800€	1.800€	1.800€	1.800€
Visual Studio	0€	0€	0€	0€	0€	0€	0€	0€
GitHub	0€	0€	0€	0€	0€	0€	0€	0€
Software	0€	0€	0€	0€	0€	0€	0€	0€
1 ordenador	1.500€	0€	0€	0€	0€	0€	0€	0€
Mantenimiento del equipo	100€	100€	100€	100€	100€	100€	100€	100€
Hardware	1.600€	100€	100€	100€	100€	100€	100€	100€
Alquiler	750€	750€	750€	750€	750€	750€	750€	750€
Agua, luz, gas	50€	50€	50€	50€	50€	50€	50€	50€
Costes totales	800€	800€	800€	800€	800€	800€	800€	800€
Total	4.200€	2.700€	2.700€	2.700€	2.700€	2.700€	2.700€	2.700€
							Total	23.100€

Figura 4.4
Desglose de los gastos totales del proyecto debido al prolongamiento del mismo.

En la *Figura 4.3* encontramos el desglose total del coste del proyecto en base a los meses que se ha prolongado el desarrollo del mismo. Podemos observar que el coste total del proyecto es de 23100 euros, que son 8100 euros más que lo que se contempló en una primera instancia, que eran 15000 euros.

5. Desarrollo del proyecto

5.1. Introducción al desarrollo

Este proyecto podría dividirse en dos partes. La primera es el desarrollo de la base del motor y la segunda es la utilización de este motor para realizar el renderer 3D con path tracing.

Para realizar la base del motor, se ha seguido una playlist[23] del youtuber TheCherno, en la cual realiza un motor de videojuegos. Cabe destacar que la playlist no se ha seguido al 100% debido a que hay ciertas cosas que para este proyecto no son necesarias y, por tanto, se ha decidido obviarlas. Por ejemplo, en la playlist se implementa un sistema de scripting o un motor de físicas, y para este proyecto no son necesarios.

La razón por la que se ha decidido escoger esta base del motor, es principalmente por la estructura del mismo. Este motor está bien estructurado, y está todo abstraído correctamente. Es importante empezar el proyecto con una buena base para que el desarrollo del proyecto luego sea más sencillo, y esta playlist aporta justamente eso.

La segunda parte del proyecto, que es la que corresponde al desarrollo de la aplicación que se quiere realizar, sí que es totalmente propia y no se sigue ningún tutorial para la consecución de la misma.

5.1.1. Planificación

Primero de todo se hizo un planteamiento de las tareas que se tenían que realizar excluyendo la parte de la base del motor, es decir, las tareas se centraban solo en la parte del desarrollo del path tracing y todo lo necesario para conseguirlo.

[23] Playlist del tutorial.

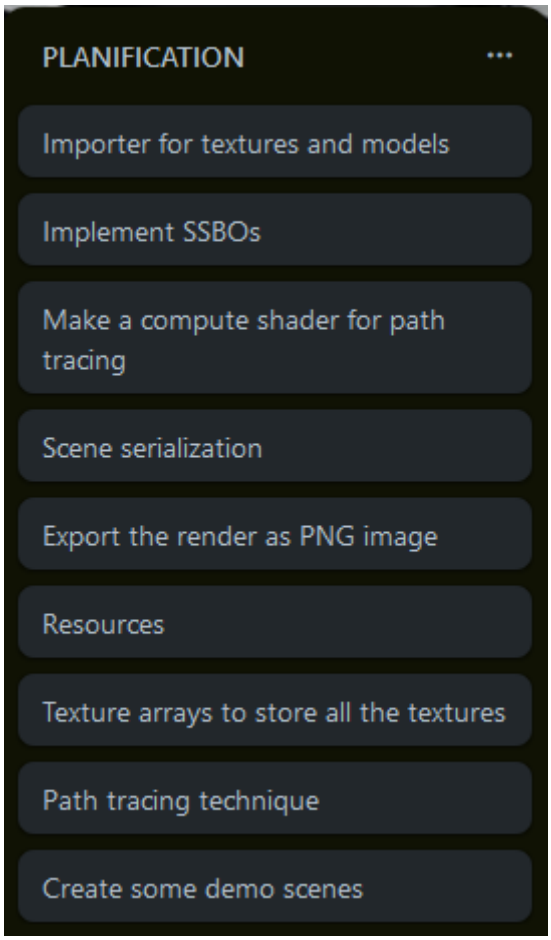

Figura 5.1
Imagen de la planificación de las tareas.

En la Figura 5.1, se puede ver la planificación de las tareas para llegar a desarrollar el path tracing. Primero de todo se realiza el importer de modelos y texturas porque para crear los SSBOs, primero se tiene que tener toda la información disponible. Posteriormente de crear los SSBOs para toda la información de la escena, se realizó una pequeña escena de muestra con un par de objetos para poder probar la técnica del path tracing.

Después se añadió la exportación del framebuffer a formato png, ya que era una tarea individual a la técnica del path tracing, es decir, que no afectaba el resultado del path tracing a la implementación de la exportación, así que se decidió realizar esta tarea para que no se acumulase la faena ya que además era una tarea sencilla. Posteriormente se realizó el sistema de recursos y también la serialización de las escenas. Finalmente se realizó la técnica del path tracing, y una vez hecha, se creó el texture array para pasar las texturas a la GPU. La razón por la cual el texture array fue lo último es porque realmente para poder realizar la técnica no eran completamente necesarias las texturas, ya que servía con usar el color difuso del material, y una vez la técnica funcionaba correctamente, se añadieron las texturas.

En la imagen anterior se puede ver una tarea que era para hacer el path tracing en un compute shader, esta tarea se llegó a realizar pero el resultado es que era más lento que en un shader normal, y por tanto, se decidió descartarla y utilizar un shader normal.

5.2. Librerías externas utilizadas

Para realizar toda la base del motor, se han utilizado librerías externas, es decir, desarrolladas por otras personas. A continuación, se enumeran todas las librerías externas que se utilizan en este proyecto y en qué consiste cada una.

- **Assimp:** Esta librería sirve para importar modelos en casi todos los formatos existentes.
- **GLFW:** Esta librería se encarga de generar la ventana de la aplicación, así como de los inputs del usuario.
- **Glad:** Esta librería se encarga de ofrecer todas las funciones de OpenGL. Es una librería clave junto con GLFW.
- **GLM:** Esta es una librería de matemáticas basada en cómo funciona OpenGL. Permite trabajar de forma mucho más cómoda y además contiene todas las funciones matemáticas necesarias para trabajar con entornos 3D.
- **ImGUI:** Esta es una librería que nos permite crear UI de forma muy sencilla para la aplicación. Con ella, la realización del editor del motor será muy fàcil.
- **Spdlog:** Esta es una librería que permite escribir en la consola de manera rápida y eficiente. Es muy útil para reportar errores y para sacar información de debug, como por ejemplo, si los modelos se han cargado correctamente, o si hay algún fallo en el shader.
- **STB:** Librería para cargar texturas desde cualquier formato, ya sea png, jpg o cualquier otro.

Una vez desglosadas las librerías externas cabe destacar, que pese a tener más experiencia con SDL y MathGeoLib, se ha decidido escoger GLFW y GLM respectivamente para reemplazarlas. La razón principal es porque, como se ha comentado anteriormente, esta base se ha hecho siguiendo un tutorial, y en este se usan estas librerías. Pese a ello, en caso de que no se siguiese este tutorial para realizar la base del motor, también se escogerían estas dos librerías frente a las otras. La razón es que son algo más sencillas de utilizar y más intuitivas, y en el caso de GLM, esta está mucho mejor optimizada que MathGeoLib, sobre todo cuando se trabaja en debug.

5.3. Base del motor

5.3.1. Aplicación

Es la clase base del motor, y la que se encarga de gestionar el loop principal. Contiene la ventana y se encarga de distribuir los eventos generados en la ventana hacia todas las layers. También contiene el sistema de capas y se encarga de actualizarlas.

```
void Application::Run()
{
    while (running)
    {
        float time = (float)glfwGetTime();
        Timer timer = time - lastFrameTime;
        lastFrameTime = time;

        for (Layer* layer : layerStack)
            layer->Update(timer);

        imguiLayer->Begin();

        for (Layer* layer : layerStack)
            layer->RenderImGui();

        imguiLayer->End();

        window->Update();
    }
}
```

Figura 5.2
Imagen del bucle de la aplicación.

En la *Figura 5.2* se observa el bucle de la aplicación, se puede ver cómo se actualizan todas las capas que hay en la lista y también se renderizan. Al final de todo se encuentra la actualización de la ventana, en la cual se hace el cambio de buffers para mostrar el frame que se ha dibujado y se limpia el anterior para poder comenzar a dibujar el siguiente frame.

5.3.2. Ventana, eventos e inputs

Esta clase se encarga de inicializar la librería de GLFW y de crear la ventana. En esta clase también se generan los eventos que luego son pasados a la aplicación para que ésta los distribuya a las diferentes capas. En esta clase también se realiza el cambio de los buffers al final de cada frame.

```
// Setting Mouse Scroll Callback
glfwSetScrollCallback(window, [](GLFWwindow* window, double offsetX, double offsetY)
{
    WindowData& data = *(WindowData*)glfwGetWindowUserPointer(window);

    MouseScrolledEvent event((float)offsetX, (float)offsetY);
    data.eventCallback(event);
});
```

Figura 5.3
Imagen del callback del scroll del ratón.

En la *Figura 5.3* se ve un ejemplo de un callback de la librería de GLFW. En este caso es el callback para el scroll del ratón. Cuando se detecta el input en la rueda del ratón, se genera un evento para distribuir esta información a través de toda la aplicación y sus capas.

5.3.3. Logging

Este fue el siguiente apartado después de crear la ventana, los eventos y el input. El logging es una herramienta muy útil para este proyecto ya que nos permite reportar cualquier tipo de información a la consola. Para ello se utiliza la librería de spdlog, la cual se encarga de escribir en la consola aquello que se le indica. Hay diferentes funciones para reportar esta información, y la propia librería escribe esa información en colores diferentes. En caso de querer reportar información meramente informativa, las funciones que se utilizan escriben en color verde o blanco en la consola, pero si queremos reportar algún error o fallo, este mensaje aparecerá en color rojo en la consola.

Esto es especialmente útil a la hora de programar ya que facilita mucho el encontrar errores si se usa correctamente. Esta es una herramienta que realmente no aporta ningún valor como tal al proyecto que se quiere desarrollar, pero sí que ayuda mucho al programador.

```
// Core log macros
#define LUX_CORE_TRACE(...)    ::Lux::Log::GetCoreLogger()->trace(__VA_ARGS__)
#define LUX_CORE_INFO(...)     ::Lux::Log::GetCoreLogger()->info(__VA_ARGS__)
#define LUX_CORE_WARN(...)     ::Lux::Log::GetCoreLogger()->warn(__VA_ARGS__)
#define LUX_CORE_ERROR(...)    ::Lux::Log::GetCoreLogger()->error(__VA_ARGS__)
#define LUX_CORE_FATAL(...)    ::Lux::Log::GetCoreLogger()->fatal(__VA_ARGS__)
```

Figura 5.4
Imagen de las funciones para escribir en la consola.

En la *Figura 5.4* se puede observar lo comentado anteriormente. Cada función escribe de un color distinto en la consola, haciendo más visual la información que se muestra y que sea más fácil de detectar si ha habido algún error o problema.

5.3.4. Sistema de layers

Las capas son las que definen la aplicación y que se ejecutan por encima de la base del motor. El sistema funciona como si fuese una lista, y se pueden añadir tantas como se requieran. Es realmente útil para poder diferenciar todas las funcionalidades que son básicas del motor, y aquello que es simplemente específico de la aplicación que se está desarrollando.

Las capas se añaden a una lista, y esta se actualiza y se renderiza a cada frame, iterando sobre todas las capas que existen en esta lista. De esta manera, se consigue diferenciar fácilmente el motor de la aplicación.

A pesar de que en este proyecto solo se va a hacer una aplicación, puede parecer que no es algo realmente importante, pero sí que es cierto que es la arquitectura correcta a la hora de desarrollar un motor. También cabe destacar que en caso de querer hacer simples pruebas de cosas que se estén desarrollando, pero sin tener toda la interfaz de usuario que hay en la aplicación, sería tan sencillo como crear una nueva capa para aquello que se quiera probar, y añadir esta capa a la lista y no en la que se está trabajando.

```
Layer(const std::string& name = "Layer");
virtual ~Layer();

virtual void OnCreate() {}
virtual void OnDestroy() {}

virtual void Update(Timer timer) {}
virtual void RenderImGui() {}

virtual void OnEvent(Event& e) {}
```

Figura 5.5
Imagen de la clase base Layer y sus funciones.

En la *Figura 5.5* se observa la clase Layer y las funciones que contiene. La capa utilizada para crear la aplicación hereda de esta clase base. Es decir, las funciones de la clase Layer se definen propiamente en la capa que crea el programador. Es por esto mismo que la información de toda la aplicación que se está creando, está en la capa y no en la base del motor.

5.3.5. Renderer

El renderer es la parte más amplia de la base del motor ya que contiene muchísimas clases y mucha información. Además, teniendo en cuenta que este proyecto se centra en el apartado gráfico, es extremadamente importante que la arquitectura del renderer sea correcta y ordenada para facilitar el dibujado de la aplicación.

El renderer es una clase estática, es decir, que se puede utilizar desde cualquier lugar sin necesidad de crear una instancia. Esta clase contiene las funciones básicas para dibujar por pantalla, bien sean modelos en 3D, o bien un cuadrado que ocupe toda la pantalla.

```
class Renderer
{
public:
    static void Init();

    static void BeginScene(const PerspectiveCamera& camera);
    static void EndScene();

    static void Submit(const std::shared_ptr<Shader>& shader, const std::shared_ptr<Material>& material,

    static void DrawFullscreenQuad();
```

Figura 5.6
Imagen de la clase Renderer y algunas funciones.

En la *Figura 5.6* se ven algunas de las funciones que tiene la clase Renderer. La función Submit por ejemplo, sirve para dibujar por pantalla un modelo 3D. También se puede ver la función BeginScene, en la cual se guarda la información de la cámara para luego pasársela al shader y poder realizar los cálculos que sean necesarios.

5.3.5.1. Shaders

Los shaders son uno de los aspectos más importantes de este proyecto. Un shader es un programa que se ejecuta en la tarjeta gráfica y que específica aquello que se tiene que dibujar. Al ser algo tan importante y que puede llegar a requerir muchísima información, se ha decidido abstraer para facilitar su uso.

Se han creado funciones específicas para subir información a la GPU y no tener que hacerlo de forma manual todo el rato. También cabe destacar que se ha añadido la detección de errores de sintaxis en el shader. Esto es realmente útil, ya que normalmente, en caso de haber un error en el shader, simplemente aparece la pantalla de color negro, como si no se hubiese dibujado nada.

```
if (isCompiled == GL_FALSE)
{
    GLint maxLength = 0;
    glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &maxLength);

    std::vector<GLchar> infoLog(maxLength);
    glGetShaderInfoLog(shader, maxLength, &maxLength, &infoLog[0]);
    glDeleteShader(shader);

    LUX_CORE_ERROR("{0}", infoLog.data());
    LUX_CORE_ASSERT(false, "Shader Compilation Failed!");
    break;
}
```

Figura 5.7
Imagen de la detección de errores en el shader.

En la *Figura 5.7* se observa la prevención de errores comentada anteriormente. Tener esta prevención de errores facilita mucho el trabajo ya que evita muchísimas horas buscando el error en el shader, ya que, si el shader no compila, la aplicación no se podrá ejecutar y en la consola aparecerá el error que hay en el shader, así que la detección de errores de sintaxis será inmediata.

5.3.5.2. Framebuffer

El framebuffer también es una parte importante de este proyecto. Se ha decidido abstraer ya que facilita mucho la tarea para limpiarlo, modificar el tamaño, pero sobretodo porque en este proyecto es necesario utilizar más de uno, así tenerlo abstraído es de gran ayuda.

Se ha creado una pequeña struct la cual es necesaria para la creación del framebuffer, en esta struct se indica la información acerca de cómo se quiere que sea el framebuffer, es decir, se especifica cuántas texturas de color debe tener, si tiene una textura de depth o no, si el formato de la textura de color es de 8 bits o 16, etc.

Esto automatiza mucho el proceso de crear un framebuffer, ya que, si se tuviese que hacer de forma manual cada vez que se quiere crear uno, el código sería mucho menos legible, y también se perdería demasiado tiempo.

```
FramebufferSpecification spec;
spec.width = 1280;
spec.height = 720;
spec.swapChainTarget = true;

spec.attachments.attachments =
{
    FramebufferTextureFormat::RGBA16,
    FramebufferTextureFormat::RGBA16,
    FramebufferTextureFormat::RGBA16,
    FramebufferTextureFormat::DEPTH24_STENCIL8
};

sceneFramebuffer = CreateSharedPtr<Framebuffer>(spec);
```

Figura 5.8
Imagen de la struct necesaria para crear un framebuffer.

5.3.5.3. Shader Storage Buffer Object

Esta clase ha sido creada siguiendo la misma línea que el resto de clases mostradas anteriormente, pero no ha sido creada mediante el tutorial del engine, ya que no se implementa el SSBO en ese tutorial.

El shader storage buffer es un elemento muy importante y clave de este proyecto ya que es el que nos permite subir toda la información de la escena a la GPU para poder realizar el path tracing. Ha sido abstraído para facilitar su uso, ya que se iba a utilizar un SSBO para los vértices, índices, normales, mallas, materiales, matrices de transformación, AABBs y objetos, y como cada uno tiene un formato diferente, era mejor abstraerlo para facilitar la tarea.

```
glGenBuffers(1, &ssbo);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo);
glBufferData(GL_SHADER_STORAGE_BUFFER, size, data, GL_DYNAMIC_COPY);
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, binding, ssbo);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, 0);
```

Figura 5.9
Imagen del código para crear un SSBO.

También se ha creado una función para cambiar solo parte de la información del SSBO. Esto es realmente útil a la hora de cambiar partes pequeñas como, por ejemplo, cuando se cambia el color del material de un objeto, se cambia únicamente ese material y así se evita tener que volver a generar el SSBO de los materiales desde 0.

```
void ShaderStorageBuffer::ChangeData(void* data, uint32_t offset, uint32_t size)
{
    glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo);
    glBufferSubData(GL_SHADER_STORAGE_BUFFER, offset, size, data);
    glBindBufferBase(GL_SHADER_STORAGE_BUFFER, bindingIndex, ssbo);
    glBindBuffer(GL_SHADER_STORAGE_BUFFER, 0);
}
```

Figura 5.10
Imagen de cómo se cambia solo parte de la información del SSBO.

5.3.5.4. Texture array

De la misma manera que el SSBO, esta clase también ha sido creada siguiendo la estructura del resto de clases, pero no ha sido creada siguiendo el tutorial que sí se ha usado para el resto de la base del motor.

Esta clase es realmente útil para poder subir todas las texturas que se están usando en la escena, para luego posteriormente extraerlas en el shader. A pesar de que, a diferencia de los SSBO, en este proyecto solo hay una texture array, también se ha decidido abstraer esta clase para facilitar el trabajo y la legibilidad del código.

```
glGenTextures(1, &textureID);
glBindTexture(GL_TEXTURE_2D_ARRAY, textureID);
glTexParameteri(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexImage3D(GL_TEXTURE_2D_ARRAY, 0, GL_RGBA8, 1024, 1024, size, 0, GL_RGBA, GL_UNSIGNED_BYTE, data);
glBindTexture(GL_TEXTURE_2D_ARRAY, 0);
```

Figura 5.11
Imagen de cómo se crea un texture array.

5.4. Renderer 3D con path tracing

Este es el objetivo de este proyecto, realizar un renderer 3D con path tracing. Para llevar a cabo este trabajo, antes se ha realizado un proyecto siguiendo el tutorial del libro Ray Tracing in One Weekend.

La razón de esta decisión es porque es una buena base para entender la técnica del ray tracing, entender los conceptos básicos, y de esta manera después será más sencillo realizar el paso de ray tracing a path tracing.

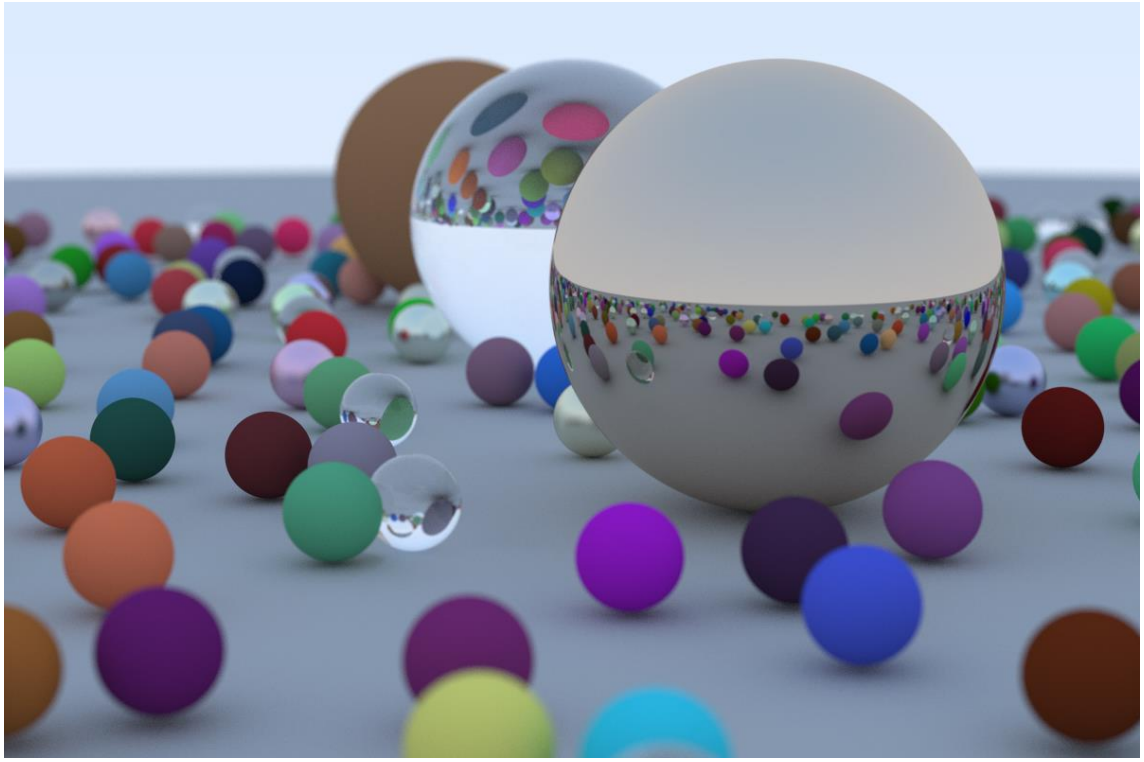

Figura 5.12
Imagen[24] del resultado final del libro Ray Tracing in One Weekend.

En la *Figura 5.12* se ve el resultado final que se obtiene al realizar este tutorial. Uno de los principales inconvenientes de este tutorial es que se realiza en la CPU y el renderer que se quiere desarrollar en este proyecto se realiza en la GPU, pero a pesar de ello, es una grandísima base a nivel conceptual para entender el proceso de esta técnica, así que este tutorial se utiliza de manera didáctica para poder replicar el proceso del tutorial en la GPU con los conocimientos aprendidos.

Para realizar este tutorial se creó un proyecto nuevo en la solución de Visual Studio. Además, este fue un caso en el que se aplicó lo explicado en el apartado del sistema de capas. Simplemente se cambió la capa que se estaba usando para el desarrollo del renderer con path tracing por una nueva capa en la que se desarrollaba este tutorial. Este proyecto se puede encontrar en el código fuente del trabajo, pese a ello, hay que tener en cuenta que es un renderer en la CPU y, por tanto, el programa tarda mucho en

[24] Imagen extraída de la siguiente página web.

realizar la imagen final ya que se está mandando un rayo por cada píxel del framebuffer, y a su vez este va rebotando, lo cual aumenta los cálculos necesarios.

Esto es un problema que en el renderer que se hace por GPU también podemos encontrar, pero la incidencia de este problema es menor en el caso de la GPU, ya que cada píxel es independiente del otro y por tanto se calculan múltiples píxeles a la vez (esto es debido a la cantidad de threads que tiene la tarjeta gráfica, que permite realizar varias tareas simultáneamente).

5.4.1. Gestión de los datos que se envían a la GPU

Para realizar el path tracing, es necesario tener toda la información relativa a todos los objetos que existen en la escena. Pasar toda esta información a la GPU es bastante costoso y, además, bastante difícil ya que son muchos datos y mucha memoria.

5.4.1.1. Vértices

En una primera instancia se decidió realizar bounding boxes (las bounding boxes son cubos que encajan a la perfección con la malla, para crearlas se utiliza el punto mínimo y máximo de esa malla) por cada triángulo que hubiese en esa malla. El resultado obtenido no fue el esperado ya que, a pesar de ser cubos muy pequeños, al realizar las colisiones con cada una de las bounding boxes, el resultado final que daba era la forma de la bounding box y no del modelo como tal, es decir, si el objeto era una esfera, lo que se veía renderizado al fin y al cabo era un cubo, del mismo tamaño que la esfera, pero evidentemente no tenía la forma correcta.

Aparte de no obtener el resultado esperado, otro inconveniente de las bounding boxes es que no se podían subir muchas a la GPU, ya que, a partir de cierto tamaño, no se pueden declarar arrays en el shader, porque la memoria de la GPU tiene un límite. Se valoró la posibilidad de realizar esto en un compute shader, ya que en este tipo de shader, se pueden declarar arrays sin especificar el tamaño que tendrá, pero como se sabía que el principal inconveniente de las bounding boxes era que el resultado final no era el correcto, se decidió descartar esta vía también.

Después se decidió empaquetar los vértices de cada modelo 3D en una sola textura, y extraerlos de esa textura en el shader, y con esos vértices, se podían crear los triángulos y comprobar la colisión del rayo con el triángulo. Esto funcionó correctamente, pero tenía un gran inconveniente, y es que el proceso era muy lento.

Finalmente, la opción final fue crear un SSBO para añadir todos los vértices y subirlos a la GPU. El SSBO está formado por un array de vec4 de tamaño indefinido. Sólo es posible crear un array de tamaño indefinido por cada SSBO, por eso se crean varios SSBO para guardar el resto de la información.

```
float offsetVertices = positions.size();
positions.resize(offsetVertices + comp->GetPositions().size());
std::copy(comp->GetPositions().begin(), comp->GetPositions().end(), &positions[offsetVertices]);
```

Figura 5.13
Imagen de cómo se juntan los vértices de todos los modelos.

Aparte de los vértices de cada modelo, se sube a la GPU, la información de los objetos de la escena. Es decir, de cada modelo se declara cual es el offset y el tamaño de los vértices, para luego poder sacar del SSBO los vértices correctos de esa malla. Esto es debido a que en la escena hay más de un objeto y, por tanto, se necesita saber de alguna manera donde empiezan y donde acaban los vértices de cada modelo.

5.4.1.2. Índices

Para poder recrear los triángulos de la malla en el shader, son necesarios también los índices. Para acceder a ellos se realiza de la misma manera que con los vértices. Los índices se compactan todos en una misma estructura de datos, y se crea un SSBO con la información de esta estructura.

```
float offsetIndices = indices.size();
indices.resize(offsetIndices + comp->GetIndices().size());
std::copy(comp->GetIndices().begin(), comp->GetIndices().end(), &indices[offsetIndices]);
```

Figura 5.14
Imagen de cómo se juntan los índices de todos los modelos.

En la *Figura 5.14* se ve cómo se empaquetan los índices en la estructura de datos. Esto, al igual que con los vértices, se hace iterando sobre todos los objetos de la escena, y una vez se finaliza el proceso, se crea la textura con toda esta información.

```
meshesInfo.push_back(glm::vec4(offsetVertices, sizeVertices, offsetIndices, sizeIndices));
aabbs.push_back(mesh->GetAABB());
```

Figura 5.15
Imagen de cómo se guarda la información de cada malla.

En la *Figura 5.15* se puede observar cómo se guardan la información del número de vértices e índices que tiene esa malla y también del offset de vértices e índices para poder extraerlos correctamente en el shader. También cabe remarcar que cada malla se añade solo una vez, es decir, a cada malla se le asigna un identificador cuando es añadida a esta lista y, por tanto, si hay algún otro objeto en la escena con esta misma malla, en la estructura de datos de información del objeto (que se verá posteriormente) se le asignará este identificador, optimizando así el espacio de las mallas en el SSBO.

5.4.1.3. Normales

Cuando un rayo colisiona con un objeto es necesario saber la normal del objeto en el punto donde han colisionado él y el rayo, para poder saber hacia dónde sale rebotado el rayo. Esta información también se necesita en la GPU, y para ello se sigue el mismo proceso que con los vértices y los índices. En este caso, como hay una normal por vértice, no hace falta que guardar el offset y el tamaño, ya que será el mismo que el de los vértices.

```
// Extracting normals from the SSBO
vec4 normal1 = normalsBO[int(indices[i].x + meshInfo.x)];
vec4 normal2 = normalsBO[int(indices[i].y + meshInfo.x)];
vec4 normal3 = normalsBO[int(indices[i].z + meshInfo.x)];
```

Figura 5.16
Imagen de cómo se extraen las normales de cada modelo.

5.4.1.4. Coordenadas de textura

Para las coordenadas de textura, se realiza una pequeña optimización para evitar tener que crear un SSBO específico para ellas. Esta optimización consiste en guardar las coordenadas de textura en el cuarto elemento de los vec4 de las arrays de vértices y normales. Las coordenadas de textura son dos valores, y como un triángulo tiene tres vértices, se necesitan almacenar un total de seis valores. Cuando se guardan los vértices y las normales, quedan los huecos libres del componente w de cada vector, al haber tres vértices y tres normales, quedan en total seis huecos libres que son los justos para almacenar todas las coordenadas de textura.

```
vec2 tC1 = vec2(v1.w, normal1.w);
vec2 tC2 = vec2(v2.w, normal2.w);
vec2 tC3 = vec2(v3.w, normal3.w);
```

Figura 5.17
Imagen de cómo se extraen las coordenadas de textura.

Como se observa en la *Figura 5.17*, los componentes x de las coordenadas de textura se guardan en el componente w del vec4 de cada vértice mientras que los componentes y se guardan en el componente w del vec4 de cada normal.

5.4.1.5. Matrices de transformación

Cada objeto en la escena está situado en una posición en concreto, que está definida por una matriz de transformación. Los vértices están en posiciones locales y, por tanto, también se necesita subir esa información a la GPU, y se hace de la misma manera que con los vértices, agrupando todas las matrices de transformación y creando un SSBO con todos esos datos. Para poder diferenciar cada matriz de transformación, se añade en la estructura de información del objeto un identificador, que es la posición en el array de matrices del SSBO, y de esa manera, se puede extraer la matriz de transformación de ese objeto.

```
mat4 modelMatrix = transforms[int(object.x)];
```

Figura 5.18
Imagen de cómo se extraen las matrices de transformación de cada modelo.

5.4.1.6. Texturas

Para poder acceder a las texturas, se utiliza un texture array. Es decir, se almacenan todas las texturas en un vector como información, y luego posteriormente se crea el

texture array con toda esa información. El texture array ha de tener un tamaño fijo, es decir, las texturas han de ser del mismo tamaño, por eso mismo, cuando se añade una textura al motor, se comprueba si esta es de 1024x1024, que es el tamaño escogido para el texture array. Si la textura no es de 1024x1024, se procederá a reescalar la imagen para que este en ese tamaño. Además, todas han de tener el mismo número de canales, es por eso que también al reescalarlas, se crean como texturas RGBA, ya que el texture array se crea con formato RGBA.

```
texture(texturesTex, vec3(hit.texCoords, hit.material.textureIDs.x)).xyz;
```

Figura 5.19
Imagen de cómo se extrae la textura en el shader.

Como se observa en la *Figura 5.19,* para acceder a la textura se utilizan las coordenadas de textura, como es normal a la hora de extraer el color de la textura en esa coordenada, pero en este caso, además de las coordenadas también es necesario añadir el identificador de la textura, que está en la información del material. En caso de que un objeto no tenga alguna de las texturas posibles, ya sea diffuse, normal map, metallic o roughness map, el identificador es igual a -1. Esto se comprueba en el shader para evitar que al multiplicar con el color del objeto o algún otro valor, este de como resultado 0 si el objeto no tiene textura. Con este identificador evitamos principalmente que haya texturas repetidas, al igual que con las mallas y, por tanto, si dos objetos tienen la misma textura, tendrán el mismo identificador en la estructura de datos de sus respectivos materiales. De esta manera, optimizamos la memoria y el espacio en el texture array.

Para crear el texture array, se necesita la información de la textura, es decir, todos los píxeles. Puesto que las texturas ya se cargan en memoria en la GPU, para evitar tener una variable que guarde toda la información de la textura, lo que se hace es que, al rellenar la información de las texturas, se bindea la textura, se leen los píxeles de esa textura y se añaden al texture array. Esto se puede observar en la *Figura 5.20*.

```
int size = 1024 * 1024 * 4;
std::vector<char> dataImage(1024 * 1024 * 4);
glBindTexture(GL_TEXTURE_2D, textureID);

glGetTexImage(GL_TEXTURE_2D, 0, GL_RGBA, GL_UNSIGNED_BYTE, dataImage.data());

textures.reserve(size + textures.size());
textures.insert(textures.end(), dataImage.begin(), dataImage.end());

glBindTexture(GL_TEXTURE_2D, 0);
```

Figura 5.20
Imagen de cómo se lee la textura de la GPU para guardar la información de esta.

5.4.1.7. Materiales

Para los materiales, se crea una estructura de datos donde hay 4 vec4. El primero es el de los identificadores de las texturas, del diffuse, normal, metallic y roughness map, en ese concreto orden. Después encontramos un vec4 que es el color del objeto. El tercer

vec4 son las propiedades del objeto, donde se guarda el metallic, roughness, índice de refracción y la transmisión del objeto. Finalmente hay un vec4 donde se guarda el color emisivo de ese objeto, el color se guarda en los tres primeros componentes, y en el último se guarda el valor de la booleana de emisividad, que se puede activar y desactivar desde el editor.

```
MaterialInfo matInfo;
matInfo.textureIDs.x = material->GetDiffuse() ? material->GetDiffuse()->GetImageID() : -1.0;
matInfo.textureIDs.y = material->GetNormalMap() ? material->GetNormalMap()->GetImageID() : -1.0;
matInfo.textureIDs.z = material->GetMetallicMap() ? material->GetMetallicMap()->GetImageID() : -1.0;
matInfo.textureIDs.w = material->GetRoughnessMap() ? material->GetRoughnessMap()->GetImageID() : -1.0;
matInfo.color = material->GetColor();
matInfo.properties.x = material->GetMetallic();
matInfo.properties.y = material->GetRoughness();
matInfo.properties.z = material->GetRefractionIndex();
matInfo.properties.w = material->GetTransmission();
matInfo.emissive = glm::vec4(material->GetEmissive(), material->GetEmission());
```

Figura 5.21
Imagen de cómo se guarda la información de cada material.

De la misma forma que con el resto de estructuras de datos, los materiales también se suben a la GPU mediante un SSBO.

5.4.1.8. Objetos

Para los objetos también se crea una textura para poder extraer toda la información explicada anteriormente. En este caso, para los objetos se necesita subir el índice donde se encuentra la matriz de transformación de este objeto. Seguidamente es necesario subir el identificador de la malla que utiliza ese objeto. Por último, también es necesario guardar el índice del material del objeto, para poder extraer el material del SSBO de materiales.

```
ObjectInfo info;
info.info.x = transforms.size() - 1;
info.info.y = mesh->GetID();
info.info.z = materialsInfo.size();
objectsInfo.push_back(info);
```

Figura 5.22
Imagen de la información que se almacena sobre el objeto.

5.4.2. Visualización del resultado

Para la visualización del resultado del path tracing, se ha decidido que sea progresivo, es decir, que a cada frame se vaya realizando una muestra y se vaya acumulando con el resultado del frame anterior. La razón por la cual se ha decidido que esto sea así es porque al ser una técnica tan costosa, si intentáramos hacer la imagen de un solo paso, el usuario vería la pantalla en negro hasta que la tarjeta gráfica acabase de generar la textura.

Esto se considera algo a evitar ya que, principalmente el usuario no va a poder realizar nada hasta que se acabe de dibujar la imagen, por tanto, el programa se quedará

esperando, algo francamente negativo. Por otra parte, y esta ya es a nivel más estético de la aplicación, se considera que es mucho mejor que el usuario pueda ir viendo el proceso de cómo su imagen va obteniendo mayor nitidez y calidad a medida que van habiendo más muestras.

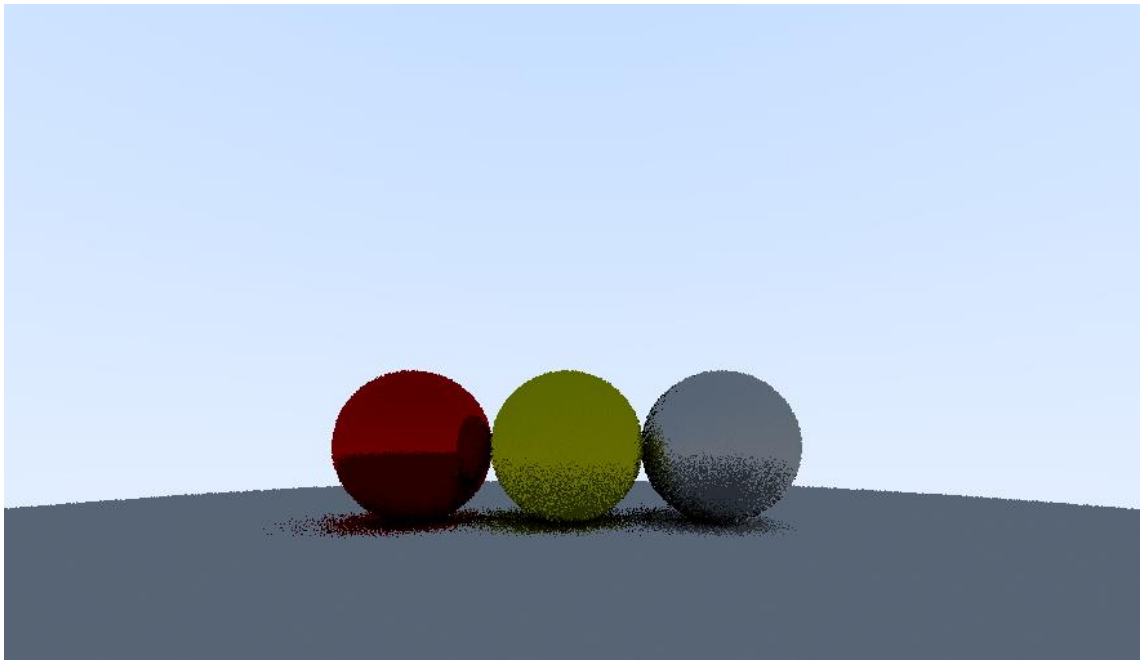

Figura 5.23
Imagen del ray tracing con únicamente dos muestras.

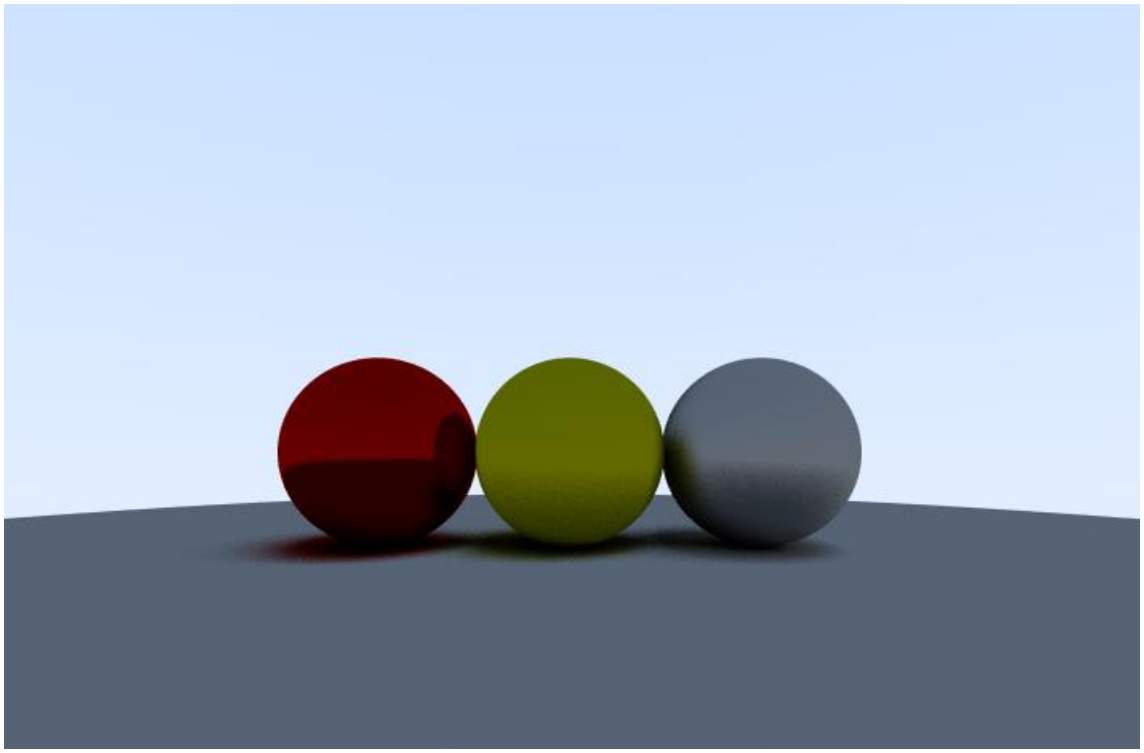

Figura 5.24
Imagen del ray tracing con cien muestras.

En las *Figuras 5.23 y 5.24*, se puede ver la diferencia entre tener pocas muestras o muchas. En este caso contamos con dos muestras en la *Figura 5.23* y con cien en la *Figura 5.24*. A medida que se aumentan las muestras, va disminuyendo el ruido de la imagen generada.

Para conseguir esto, cuando se realiza un frame, este se guarda en otro framebuffer, y cuando se va a dibujar el siguiente frame, se sube la textura de este framebuffer para reaprovecharla en el pase de dibujado. Se realizan los cálculos de la misma manera y se añade el color del anterior frame y se hace una mezcla entre ambos. Y una vez se tiene el resultado para el frame actual, se vuelve a guardar en el otro framebuffer para poder reaprovecharlo en el siguiente frame, y así sucesivamente.

```
accumulateFramebuffer->BindTextures(3);
lightingPass->SetUniformInt("accumulateTexture", 3);
lightingPass->SetUniformInt("samples", samples++);
```

Figura 5.25
Imagen de cómo se envia al shader la textura con el frame anterior.

En la *Figura 5.25* se observa parte de lo comentado anteriormente. Se ve cómo se activa la textura en la cual se acumulan los frames y se sube al shader para ser utilizada. También se sube el número de frames que van dibujados para poder hacer la mezcla correctamente, y que no salga demasiado claro ni demasiado oscuro.

```
color = (float(samples) * prev + color) / float(samples + 1);

fragColor = vec4(color, 1.0);
```

Figura 5.26
Imagen de la mezcla entre el frame anterior y el actual.

Para obtener el color en el frame actual, se realiza la fórmula que se observa en la *Figura 5.26*. Se multiplica el color del píxel del frame anterior por el número de muestras que había en el último frame y se le suma el color del frame actual. Esto dará un valor muy alto, y por eso se divide entre el número de muestras actual. Todo esto está encapsulado sobre la condición de que el color del frame anterior no puede ser negro, esto es debido a que queremos evitar que realice este cálculo en el primer frame, ya que en el primer frame no habrá nada en la textura acumulada, y esto condicionaría el color del proceso.

5.4.3. Desarrollo del path tracing

Para realizar el path tracing se ha decidido usar un shader normal y no un compute ya que en un principio se implementó el compute shader pero era más lento que en un shader normal y por eso mismo se decidió utilizar un shader normal. Por otra parte, para los cálculos del BRDF se ha decidido utilizar el mismo de Disney, en este caso se han implementado el diffuse, el reflection y refraction. Se decidió usar el modelo de Disney

para ello debido a su buen resultado visual y también a que, al ser una función utilizada por Disney, habría ciertas optimizaciones respecto a la ecuación de renderizado que son útiles para conseguir que el cálculo del color del píxel sea más rápido.

Primero de todo, se lanza el rayo y se comprueba con todos los objetos de la escena, y para ello se hace uso del SSBO de objetos. Se itera sobre el SSBO de objetos y se va extrayendo la información necesaria. Una vez se extrae la información de ese objeto del SSBO de objetos, se procede a sacar la matriz de transformación del objeto, accediendo a ella con el identificador guardado en la información del objeto. Esta matriz de transformación se invierte, y la posición y dirección del rayo se multiplican por esta matriz para que el rayo esté en el mismo sistema de coordenadas que los vértices. También se podrían multiplicar los vértices por la matriz de transformación sin necesidad de invertirla, pero esto, evidentemente es mucho más costoso ya que se deberían multiplicar todos los vértices, y aquí en cambio sólo se realizan dos multiplicaciones por objeto.

Una vez se han multiplicado la posición y la dirección del rayo por la matriz de transformación del objeto, se procede a comprobar con la AABB de la malla. En caso de colisionar con la AABB, se continua con el proceso y se pasará a comprobar cada uno de los triángulos de la malla. En caso contrario, se procederá a comprobar el siguiente objeto. Si el rayo colisiona con la AABB, entonces se extrae la información de la malla, es decir, el offset y tamaño de vértices e índices. Con esta información extraída se procede a iterar sobre los triángulos de esa malla, y se comprueba la colisión del rayo que se ha lanzado con el triángulo de la malla. Si el rayo no colisiona con el triángulo, se pasa al siguiente triángulo.

```
for (int j = 0; j < numObjects; ++j)
{
    vec4 object = objects[j];

    mat4 modelMatrix = transforms[int(object.x)];

    vec3 origin = vec3(inverse(modelMatrix) * vec4(ray.origin, 1.0));
    vec3 direction = vec3(inverse(modelMatrix) * vec4(ray.direction, 0.0));

    float aabbT = HitAABB(origin, direction, int(object.y), closest);
    if (aabbT == -1.0 || aabbT > closest)
        continue;

    vec4 meshInfo = meshes[int(object.y)];
    int meshIndices = int(meshInfo.z + meshInfo.w);

    for (int i = int(meshInfo.z); i < meshIndices; ++i)
    {
        vec4 v1 = vertices[int(indices[i].x + meshInfo.x)];
        vec4 v2 = vertices[int(indices[i].y + meshInfo.x)];
        vec4 v3 = vertices[int(indices[i].z + meshInfo.x)];

        vec3 v1v2 = v2.xyz - v1.xyz;
        vec3 v1v3 = v3.xyz - v1.xyz;

        // OPTIMIZED VERSION
        vec3 point = cross(direction, v1v3);
        float det = dot(v1v2, point);

        if (det < 0.0003)
            continue;

        float invDet = 1.0 / det;

        vec3 tVector = origin - v1.xyz;
        float u = dot(tVector, point) * invDet;
        if (u < 0.0 || u > 1.0)
            continue;

        vec3 qVector = cross(tVector, v1v2);
        float v = dot(direction, qVector) * invDet;
        if (v < 0.0 || u + v > 1.0)
            continue;

        float t = dot(v1v3, qVector) * invDet;
```

Figura 5.27
Imagen de cómo se iteran los objetos para encontrar el más cercano.

En caso de que el rayo colisione con el triángulo, se recoge toda la información necesaria de ese triángulo (punto de intersección, coordenadas de textura en ese punto, normal en ese punto, etc.) pero también se prosigue con el resto de triángulos porque se necesita saber cuál es el más cercano, y para ello es necesario iterar sobre todos ellos.

```
if (t > 0.0 && minT < t && t < closest)
{
    closest = t;
    somethingHit = true;
    hit.t = t;

    hit.point = GetRayAt(ray, t);

    // Extracting normals from the SSBO
    vec4 normal1 = normalsBO[int(indices[i].x + meshInfo.x)];
    vec4 normal2 = normalsBO[int(indices[i].y + meshInfo.x)];
    vec4 normal3 = normalsBO[int(indices[i].z + meshInfo.x)];

    vec2 tC1 = vec2(v1.w, normal1.w);
    vec2 tC2 = vec2(v2.w, normal2.w);
    vec2 tC3 = vec2(v3.w, normal3.w);

    hit.texCoords = tC1 * (1.0 - u - v) + tC2 * u + tC3 * v;

    hit.material = materials[int(object.z)];

    hit.normal = normalize(normal1.xyz * (1.0 - u - v) + normal2.xyz * u + normal3.xyz * v);
    hit.normal = normalize(transpose(inverse(mat3(modelMatrix))) * hit.normal);

    hit.frontFace = dot(hit.normal, ray.direction) <= 0.0;
    hit.normal = hit.frontFace ? hit.normal : -hit.normal;
    hit.transformID = int(object.x);
}
```

Figura 5.28
Imagen de cómo se recoge toda la información cuando se colisiona con un triángulo.

Una vez se ha acabado con el proceso de comprobar todos los objetos de la escena, se pasa a realizar el cálculo de color de ese píxel. Primero de todo, se extrae toda la información del material de ese objeto, tanto texturas como las variables de roughness, metallic y demás.

```
// Material properties
vec3 albedo = hit.material.color.xyz;
if (hit.material.textureIDs.x != -1) albedo *= texture(texturesTex, vec3(hit.texCoords, hit.material.textureIDs.x)).xyz;
float metallic = hit.material.properties.x;
if (hit.material.textureIDs.z != -1) metallic *= texture(texturesTex, vec3(hit.texCoords, hit.material.textureIDs.z)).x;
float roughness = hit.material.properties.y;
if (hit.material.textureIDs.w != -1) roughness *= texture(texturesTex, vec3(hit.texCoords, hit.material.textureIDs.w)).x;
float refractionIdx = hit.material.properties.z;
float transmission = hit.material.properties.w;
```

Figura 5.29
Imagen de cómo se extrae la información del material.

Posteriormente se procede a calcular la tangente y bitangente que serán necesarias para recrear la microfacet normal. Una vez calculadas estas dos variables, se pasa la dirección del rayo a coordenadas locales para poder realizar el cálculo de la normal del microfacet model correctamente. Una vez se consigue este valor, se pasa esta normal a coordenadas de mundo, ya que el resto de cálculos que se harán posteriormente, están en coordenadas de mundo.

```
// Calculating Microfacet normal
vec3 t, b;
TangentAndBitangent(hit.normal, t, b);
vec3 v = ToLocal(t, b, hit.normal, -direction);
vec3 h = SampleGGXVNDF(v, roughness2, frand(), frand());
if (h.z < 0.0)
    h = -h;
h = ToWorld(t, b, hit.normal, h);
```

Figura 5.30
Imagen de cómo se crea la microfacet normal.

Después de obtener la microfacet normal, se calcula el valor del Fresnel. En este caso, ya que se sigue el ejemplo de Disney, se calcularán dos Fresnel, el común, y el Fresnel dieléctrico. Esto se hace para poder obtener de manera más realista si un objeto está reflejando o refractando.

```
// Fresnel
float VdotH = dot(-direction, h);
vec3 f0 = mix(vec3(0.04), albedo, metallic);
vec3 fresnelValue = FSchlick(f0, VdotH);
float dielectricFresnel = Fresnel(hit.frontFace ? 1.0 : refractionIdx, refractionIdx, abs(VdotH), 0.0, 1.0);
```

Figura 5.31
Imagen de cómo se obtienen los dos valores del Fresnel.

Una vez se tienen los valores de los dos Fresnel, se procede a calcular el valor de los pesos, es decir, el porcentaje equivalente a diffuse, metallic y refractive de ese objeto. Para obtener el valor de estos pesos, se utilizan las variables de metallic y tranmission extraídas anteriormente del material. Si el valor metallic y transmission de un objeto es equivalente a 0, significa que ese objeto es completamente diffuse. Para evitar que el porcentaje se pueda pasar de 1, una vez se calculan los tres pesos, se divide 1 entre la suma de los tres pesos, y con ese valor obtenido, se multiplican los tres pesos.

```
// Weight probability
float diffuseWeight = (1.0 - metallic) * (1.0 - transmission);
float reflectWeight = dot(fresnelValue, vec3(0.299, 0.587, 0.114));
float refractWeight = (1.0 - metallic) * (transmission) * (1.0 - dielectricFresnel);
float invW = 1.0 / (diffuseWeight + reflectWeight + refractWeight);

diffuseWeight *= invW;
reflectWeight *= invW;
refractWeight *= invW;
```

Figura 5.32
Imagen de cómo se calculan los pesos en función del tipo de material.

Acto seguido se realiza el cálculo de un número aleatorio entre 0 y 1, y se procede a seleccionar si ese objeto reflejará, refractará o será completamente difuso. Se podría hacer una suma de los 3 tipos y luego aplicar cada color obtenido en función de los pesos, pero debido a que es un número aleatorio, en cada frame entrará en una branch

distinta y, por tanto, al ir acumulando los frames obtendremos el resultado deseado. Es decir, si un objeto es mitad difuso y mitad reflectivo, mediante este número aleatorio, en algunos frames se realizará el cálculo como si el objeto fuese difuso, y en otros como si el objeto fuese reflectivo.

```
float rnd = frand();
if (rnd < diffuseWeight)
{
    L = CosineSampleHemisphere(hit.normal);
    h = L + (-direction);

    float NdotL = dot(hit.normal, L);
    float NdotV = dot(hit.normal, -direction);
    if (NdotL <= 0.0 || NdotV <= 0.0)
    {
        return;
    }

    float LdotH = dot(L, h);
    float pdf = NdotL / PI;

    brdf.rgb = DisneyDiffuseBRDF(albedo, NdotL, NdotV, LdotH, roughness2) * (1.0 - fresnelValue);
    brdf.a = diffuseWeight * pdf;
}
```

Figura 5.33
Imagen del proceso a seguir si el material es difuso.

En la Figura 5.33, se puede observar el proceso a seguir en caso de que el objeto sea difuso. Primero de todo se obtiene la dirección aleatoria hacia donde irá el rayo, en función de la normal del punto donde ha intersecado el rayo con el objeto. Después se calcula el producto escalar de la normal con la dirección del rayo, y también el producto escalar de la normal con la dirección aleatoria calculada previamente. Si alguno de estos dos valores es igual o inferior a 0, se descarta y por tanto contará como que el color es negro completo, ya que implica que no es físicamente posible que ese rayo pueda afectar a la zona donde ha intersecado.

Después de esto, se calcula el producto escalar de la dirección aleatoria con la suma de la dirección aleatoria y la dirección del rayo actual. Esto, se divide entre el número pi para poder sacar el valor necesario para luego samplear el color. Después se calcula el color difuso en base a la fórmula de Disney, que es el método que se utiliza en este caso. La fórmula se puede observar en la *Figura 5.34*.

$$f_d = f_{Lambert}(1 - 0.5F_L)(1 - 0.5F_V) + f_{retro-reflection}$$

$$f_{retro-reflection} = \frac{baseColor}{\pi} R_R(F_L + F_V + F_L F_V(R_R - 1))$$

$$F_L = (1 - \cos\theta_l)^5$$

$$F_v = (1 - \cos\theta_v)^5$$

$$R_R = 2 * roughness * \cos^2(\theta_d)$$

Figura 5.34
Imagen[25] de la fórmula de Disney para calcular el color difuso.

Para calcular el color difuso seguimos los pasos que hay en la fórmula, calculamos los dos valores de Fresnel, y se devuelve el color del objeto multiplicado por los dos valores de fresnel entre el número pi. Con ello, obtendremos el color difuso, como podemos observar en la *Figura 5.35*.

```
vec3 DisneyDiffuseBRDF(vec3 color, float NdotL, float NdotV, float LdotH, float roughness)
{
    float FD90 = 0.5 + 2.0 * roughness * pow(LdotH, 2.0);
    float a = FSchlick(1.0, FD90, NdotL);
    float b = FSchlick(1.0, FD90, NdotV);

    return color * (a * b / PI);
}
```

Figura 5.35
Imagen de cómo se obtiene el color difuso.

[25] Imagen extraída de la siguiente página web.

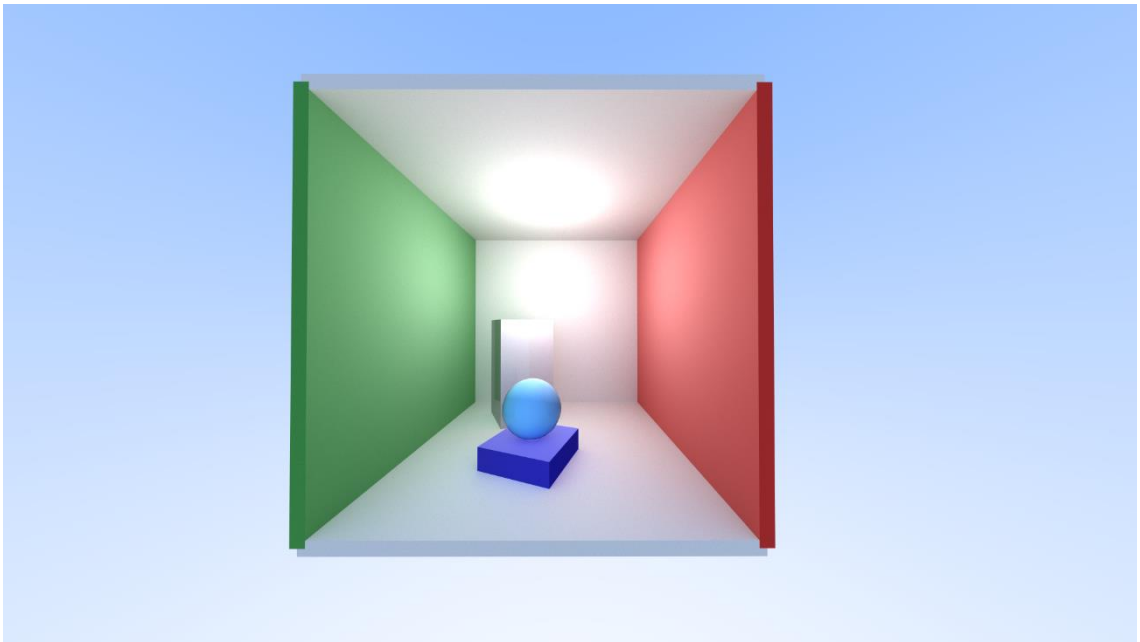

Figura 5.36
Imagen de un renderizado de 2000 sampleos de objetos difusos.

En caso de que el objeto no sea difuso y sea reflectivo se siguen los mismos pasos que en la técnica de Cook-Torrance microfacet BRDF.

```
else if (rnd < diffuseWeight + reflectWeight)
{
    L = reflect(direction, h);

    float NdotL = dot(hit.normal, L);
    float NdotV = dot(hit.normal, -direction);
    if (NdotL <= 0.0 || NdotV <= 0.0)
    {
        return;
    }

    float NdotH = min(0.99, dot(hit.normal, h));
    float pdf = GGXVNDFPdf(NdotH, NdotV, roughness2);

    brdf.rgb = DisneyReflectionBRDF(roughness, fresnelValue, NdotH, NdotV, NdotL);
    brdf.a = reflectWeight * pdf;
}
```

Figura 5.37
Imagen de cómo obtener la información para procesar el color de un objeto reflectivo.

Primero de todo se hace un reflect de la dirección con el vector intermedio calculado anteriormente en el proceso del microfacet model. De la misma manera que con los objetos difusos, se calcula el producto escalar de la normal del punto donde ha intersecado con la dirección hacia donde irá el nuevo rayo y con la dirección del rayo actual. En caso de que alguno de estos dos valores sea igual o inferior a 0 se descartará el color y, por tanto, será negro. En caso contrario, se sigue el proceso y se calcula el producto escalar de la normal con el vector intermedio calculado previamente. Después

se procede a calcular el pdf, que servirá para luego poder samplear correctamente el color obtenido en la reflexión.

```
vec3 DisneyReflectionBRDF(float matRoughness, vec3 F, float NdotH, float NdotV, float NdotL)
{
    float roughness = pow(matRoughness, 2.0);
    float D = DGTR(roughness, NdotH, 2.0);
    float G = GeometryTerm(NdotL, NdotV, pow(0.5 + matRoughness * 0.5, 2.0));

    vec3 spec = D * F * G / (4.0 * NdotL * NdotV);

    return spec;
}
```

Figura 5.38
Imagen del proceso a seguir para conseguir el color en la reflexión.

En la Figura 5.38 se puede observar cómo se obtiene el color en caso de que el objeto esté reflejando. Para ello se calcula el valor de la distribución de normales y el valor de la función geométrica. Estos valores se multiplican junto con el valor del Fresnel, y este resultado se divide entre 4 por el valor de los dos productos escalares que se han calculado previamente. Se puede observar que aquí no hay ningún cambio en la fórmula de Disney respecto a la fórmula clásica de Cook-Torrance microfacet BRDF, como si lo había para los materiales difusos.

Finalmente, se calcula el valor necesario para samplear posteriormente este color, que será el valor del peso de la reflexión que ha sido calculado previamente por el valor del pdf que también se ha calculado previamente y este valor se guarda en el componente alfa de la variable brdf.

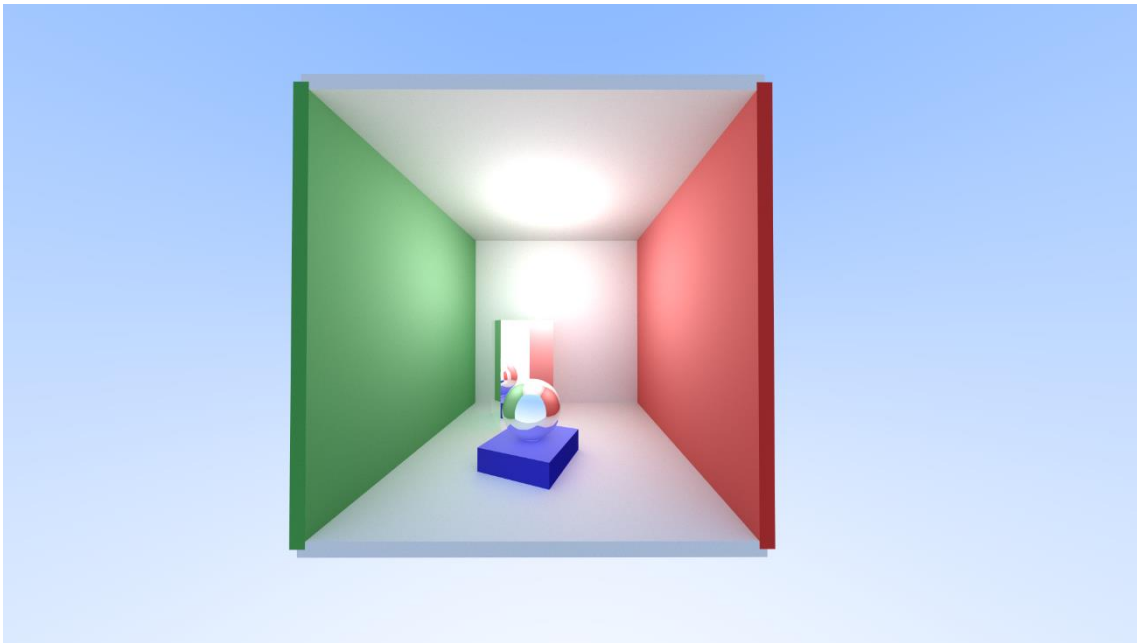

Figura 5.39
Imagen de un renderizado de 2000 sampleos de algunos objetos reflectivos.

Finalmente, para calcular como de transmisivo es un objeto, se sigue el siguiente proceso. El proceso es prácticamente idéntico al BRDF para los objetos reflectivos, simplemente tiene alguna pequeña modificación para tener en cuenta el índice de refracción del objeto con el que ha intersecado el rayo.

```
else
{
    refState.isRefracted = !refState.isRefracted;
    float refractionRatio = hit.frontFace ? 1.0 / refractionIdx : refractionIdx;
    L = refract(direction, h, refractionRatio);

    float NdotL = dot(hit.normal, L);
    if (NdotL <= 0.0)
    {
        return;
    }

    float NdotV = dot(hit.normal, -direction);
    float NdotH = min(0.99, dot(hit.normal, h));
    float LdotH = dot(L, h);

    brdf = DisneyRefractionBRDF(albedo, dielectricFresnel, NdotH, NdotV, NdotL, VdotH, LdotH, refractionRatio, roughness);
    brdf.a = refractWeight * brdf.a;
}
```

Figura 5.40
Imagen del proceso a seguir para los objetos refractivos.

$$f_t(i,o,n) = \frac{|i\cdot h_t|}{|i\cdot n|}\frac{|o\cdot h_t|}{|o\cdot n|}\frac{\eta^2}{(i\cdot h_t+\eta o\cdot h_t)^2}\frac{1}{\eta^2}(1-F(i,h_t))G(i,o,h_t)D(h_t)$$

with $h_t = -\frac{i+\eta o}{\|i+\eta o\|}$ and $\eta = \frac{\eta_i}{\eta_o}$ being the relative index of refraction.

Figura 5.41
Imagen[26] de la fórmula para calcular el color de un objeto refractivo.

Como se observa en la *Figura 5.40*, primero se cambia el valor de una booleana que sirve para mantener un control del estado de refracción de un objeto, ya que, si el objeto está refractando, pero ya lo ha hecho previamente, quiere decir que ese rayo ya está saliendo del objeto y, por tanto, no debe refractar más. Después se calcula el valor de refracción, que en caso de que la colisión del rayo con el objeto haya sido de cara, la ratio de refracción será de 1 entre el valor del índice de refracción del objeto. Es 1 ya que, al colisionar de cara, se asume que el rayo viene del aire, y el índice de refracción del aire es 1.

Una vez con la ratio de refracción calculada, realizamos un refract entre la dirección del rayo actual y el vector intermedio que ha sido calculado previamente. Una vez se tiene el valor del refract, se hace el producto escalar del resultado de este refract con la normal del objeto. De la misma manera que se ha visto en los casos de difuso y reflectivo, si el valor de este producto escalar es igual o inferior a 0, se descarta y por tanto el color devuelto será negro.

Si se sigue con el proceso, se calcula el producto escalar de la normal con la dirección del rayo actual y el vector intermedio, y también se calcula el producto escalar de la

[26] Imagen extraída de la siguiente página web.

dirección del nuevo rayo con el vector intermedio. Con toda esta información, se puede calcular el color de este objeto refractivo. En este caso, el pdf se calcula también dentro de la función donde se calcula el color del objeto refractivo.

```
vec4 DisneyRefractionBRDF(vec3 albedo, float F, float NdotH, float NdotV, float NdotL, float VdotH, float LdotH, float refractionRatio, float matRoughness)
{
    float roughness = pow(matRoughness, 2.0);
    float D = DGTR(roughness, NdotH, 2.0);
    float G = GeometryTerm(NdotL, NdotV, pow(0.5 + matRoughness * 0.5, 2.0));
    float denom = pow(LdotH + VdotH * refractionRatio, 2.0);

    float jacobian = abs(LdotH) / denom;
    float pdf = SmithG(abs(NdotL), roughness * roughness) * max(0.0, VdotH) * D * jacobian / NdotV;

    vec3 spec = pow(1.0 - albedo, vec3(0.5))  * D * (1.0 - F) * G * abs(VdotH) * jacobian * pow(refractionRatio, 2.0) / abs(NdotL * NdotV);
    return vec4(spec, pdf);
}
```

Figura 5.42
Imagen del proceso a seguir para calcular el color de un objeto refractivo.

Como se puede observar en la Figura 5.42, el proceso es bastante similar al BRDF de un objeto reflectivo, ya que también se tiene que calcular la distribución de normales y la función geométrica. La diferencia respecto al BRDF de un objeto reflectivo se encuentra en el denominador y en el resultado final, ya que se tiene en cuenta la ratio de refracción del objeto.

Una vez se ha realizado el cálculo del color, sea difuso, reflectivo o refractivo, se debe multiplicar este color por el producto escalar de la dirección del nuevo rayo con la normal del objeto. En la *Figura 5.43* se puede observar el resultado de un renderizado con objetos refractivos, que dependiendo del índice de refracción del objeto será de una manera u otra.

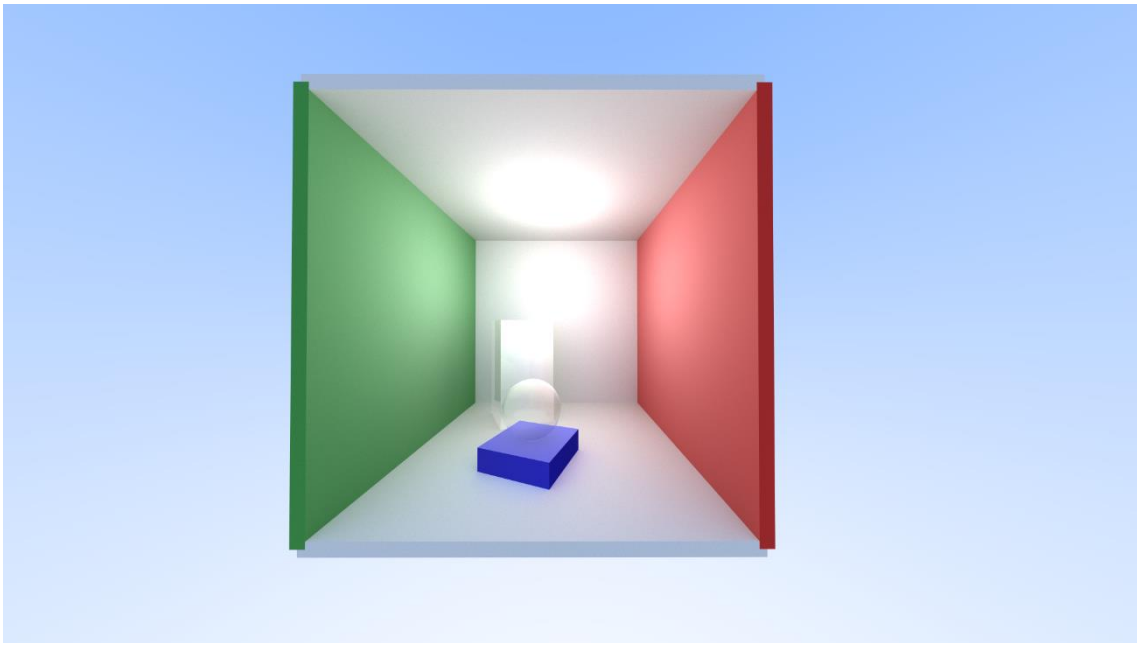

Figura 5.43
Imagen de un renderizado de 2000 sampleos con algunos objetos refractivos.

Una vez se ha obtenido el color del objeto, se procede a comprobar si el objeto es emisivo ya que, si es el caso, se debe añadir el color emisivo directamente en el color final del píxel. Para ello usamos el componente alfa del color emisivo del material, que

es donde se guarda el booleano que indica si el objeto es emisivo o no. Si es emisivo, se multiplica el color emisivo por el acumulado de los rayos que se han lanzado y se añade al color final del píxel. Esto se puede observar con más detalle en la *Figura 5.44*.

```
CalculateBRDF(hitRay.direction, hit, refState, newDir, brdf);

if (hit.material.emissive.w == 1.0)
    color += hit.material.emissive.xyz * throughput;
```

Figura 5.44
Imagen de cómo se añade el color en caso de que el objeto sea emisivo.

Después, se multiplica el color acumulado por el color que se ha calculado anteriormente siguiendo las fórmulas de Disney, y se divide entre el pdf, que ha sido guardado en el componente alfa del vector donde está guardado también el color. En caso de que el pdf sea 0, se descarta este proceso ya que en ese caso nos daría un valor de 0 y por tanto el acumulado pasaría a ser 0 también. En caso de que el objeto sea refractivo, se añade un cálculo para tener en cuenta la absorción del objeto refractivo y, ya que esta puede variar en función de cada material. La absorción de cada material se guarda en el componente alfa del color del objeto. En este caso, la absorción siempre será 0, lo que hará los objetos plenamente transparentes.

```
if (brdf.a > 0.0)
    throughput *= brdf.rgb / brdf.a;

if (refState.wasRefracted)
{
    throughput *= exp(-hit.t * ((vec3(1.0) - hit.material.color.xyz) * 0.0));
}

hitRay.origin = hit.point;
```

Figura 5.45
Imagen de cómo se modifica el acumulado con el color del rayo actual.

Después de esto, ya solo queda crear el nuevo rayo, que la posición del nuevo rayo será el punto donde ha intersecado el rayo con el objeto, y la dirección será la que se ha calculado previamente en el cálculo de color, en función de si el objeto era difuso, reflectivo o refractivo. A la posición del nuevo rayo se le añade un pequeño desfase en base a la normal del objeto para evitar que el nuevo rayo pueda colisionar con ese mismo objeto. Si el objeto es reflectivo o difuso, este desfase se añadirá en positivo, pero en caso de que el objeto sea refractivo, se añadirá en negativo. La razón es que, al ser un objeto refractivo, implica que el rayo pasará a través de ese objeto y, por tanto, la normal del objeto debe estar en negativo. Esto se puede observar en la *Figura 5.46*.

```
hitRay.origin = hit.point;

if (refState.isRefracted)
{
    hitRay.origin += -hit.normal * 0.01;
}
else if (refState.wasRefracted && !refState.isRefracted)
{
    hitRay.origin += -hit.normal * 0.01;
}
else
{
    hitRay.origin += hit.normal * 0.01;
}

hitRay.direction = newDir;

float q = max(throughput.r, max(throughput.g, throughput.b));
seed *= 1.456;
if (frand() > q)
    break;

throughput /= q;
```

Figura 5.46
Imagen de cómo se crea el nuevo rayo.

Finalmente se realiza una ruleta rusa para cortar el proceso de rebote de los rayos. Para ello se obtiene el valor máximo de los componentes rgb del color acumulativo. Una vez se tiene ese valor, se genera un número aleatorio y si este es más grande que el valor máximo obtenido previamente, se corta el proceso de rebote. En caso contrario, se divide el color acumulativo entre el valor máximo que se ha calculado. En la *Figura 5.47 y 5.48*, se puede observar un renderizado con objetos difusos, metálicos, refractivos y emisivos.

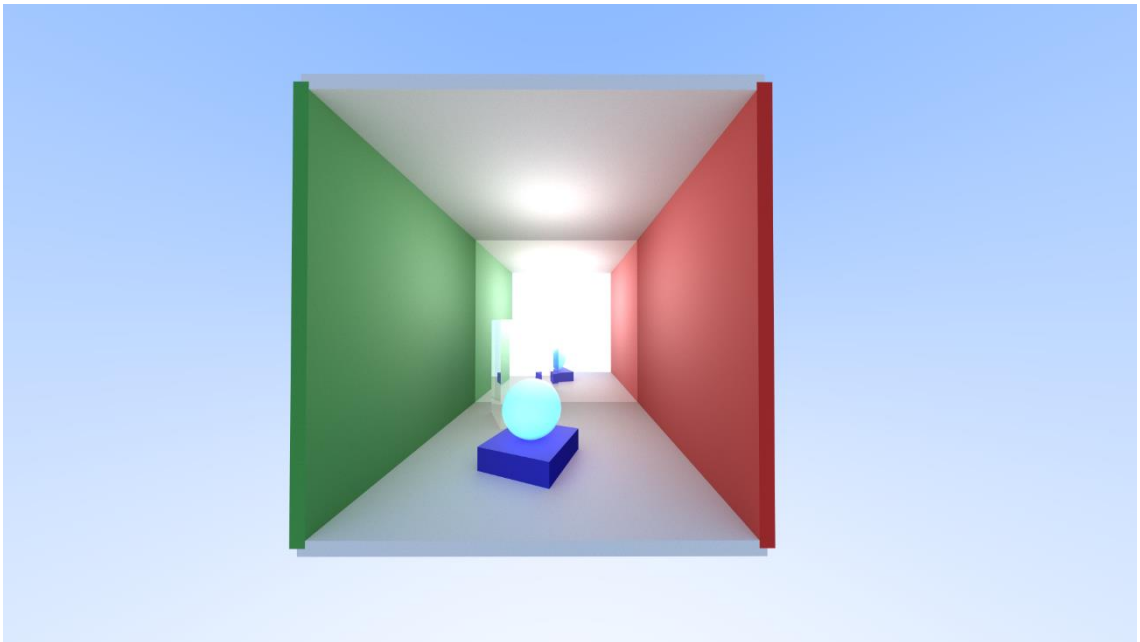

Figura 5.47
Imagen de un renderizado de 2000 sampleos con objetos difusos, reflectivos y refractivos.

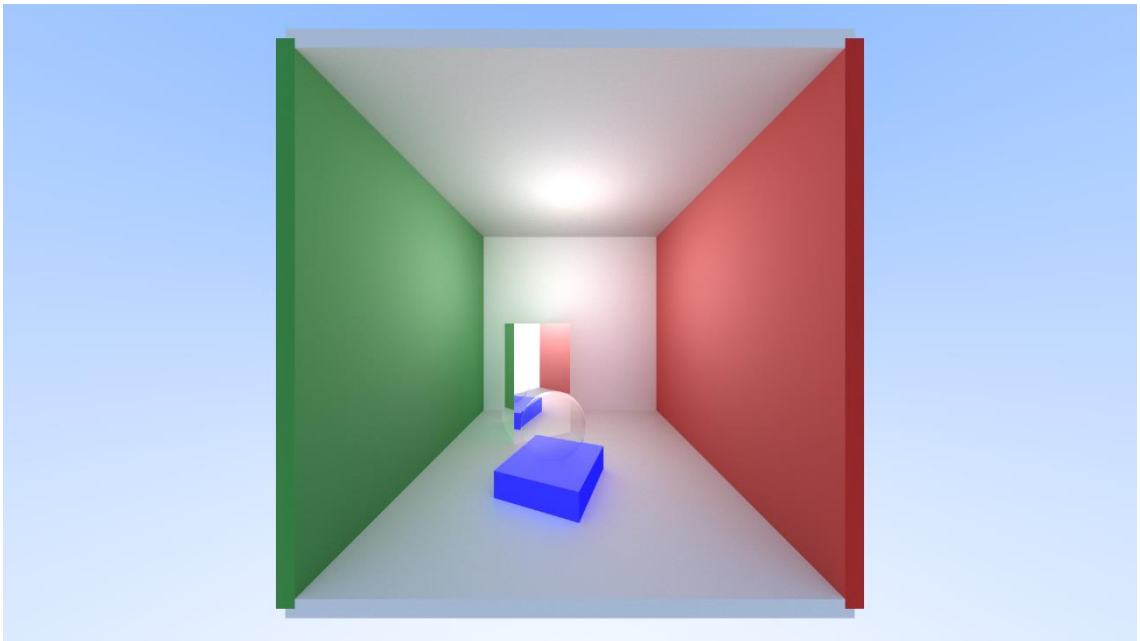

Figura 5.48
Imagen de un renderizado de 2000 sampleos con objetos difusos, reflectivos, refractivos y emisivos.

5.4.4. Funcionalidades del editor

En la *Figura 5.49*, se observa una imagen completa del editor. Cuenta con un total de cuatro paneles, dos de ellos situados a la izquierda, uno a la derecha, mientras que el último está situado abajo en el centro. En la izquierda encontramos los paneles de opciones del renderizado y de los objetos de la escena. Abajo en el centro encontramos el panel con los recursos del proyecto, y en el panel de la derecha se encuentran los componentes del objeto seleccionado, donde se puede modificar los valores del

material del objeto y la posición del mismo. Arriba se encuentra una barra donde se pueden con dos menús, uno de ellos permite crear point lights, y el otro nos permite guardar la escena actual, abrir una o empezar otra desde cero.

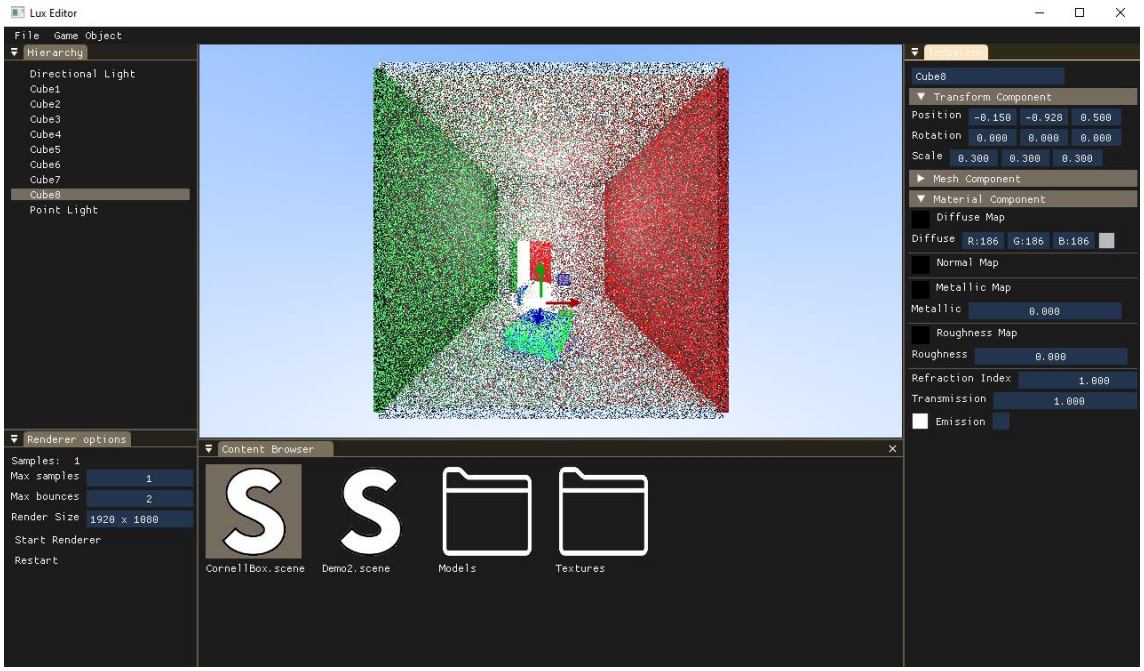

Figura 5.49
Imagen del editor con todas sus funcionalidades visibles.

En el panel de opciones de renderizado, se permite modificar el máximo de sampleos que se van a utilizar para el renderizado. También se permite modificar el número máximo de rebotes, que está limitado a un total de 8 rebotes porque permitir más podría ser peligroso para la GPU. Se permite escoger también la resolución a la que se quiere sacar el renderizado, pudiendo elegir entre 1920x1080 y 1080x720. Debajo de todas estas funciones encontramos un botón de iniciar el renderizado y otro de reiniciar. Cuando el renderizado llega al máximo de sampleos, aparece un botón para guardar el renderizado. Al pulsarlo se abrirá un menú para elegir el lugar donde se quiere guardar el renderizado en formato png. Todo esto se puede observar en la *Figura 5.50*.

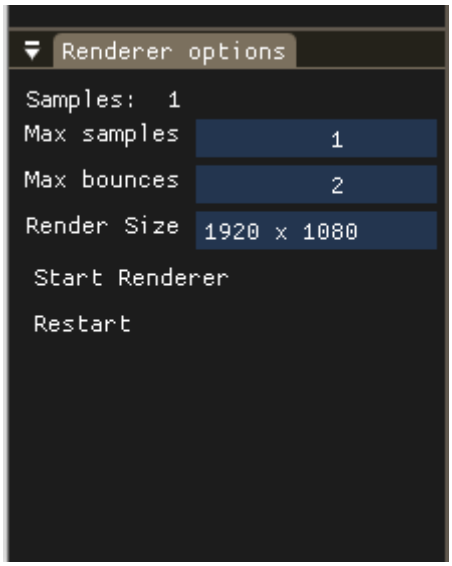

Figura 5.50
Imagen del panel de opciones del renderizado.

Después se encuentra el panel de recursos del proyecto, donde se encuentran todas las texturas, modelos y demás archivos que se han ido importando al proyecto. En el caso de las texturas, se pueden arrastrar hasta el menú del material del objeto para asignar esa textura al objeto seleccionado. Esto se puede ver en la *Figura 5.51*.

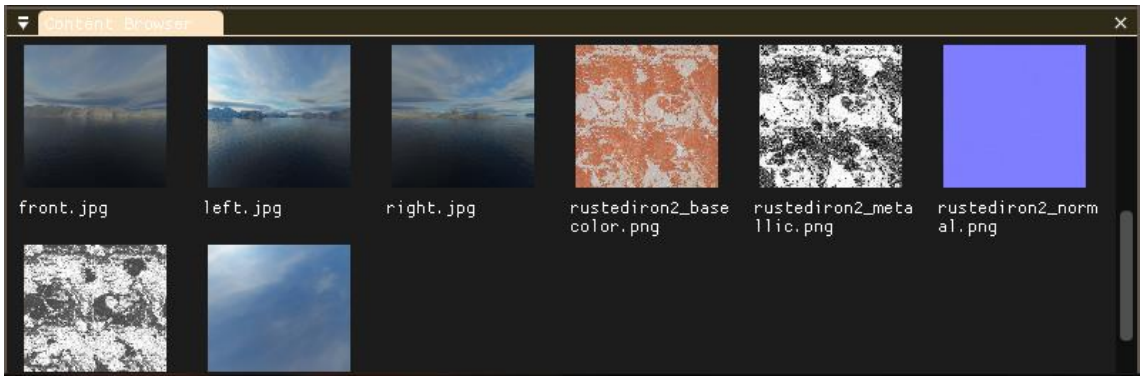

Figura 5.51
Imagen del panel de recursos.

Finalmente, en la Figura 5.52, se puede observar el menú de modificación del objeto seleccionado. En él se puede modificar la posición del objeto y también todos los valores del material que se utilizan para el renderizado de la escena, así como también se permite añadir texturas. Aquí se puede modificar la transmisión del objeto, como de metálico es, su índice de refracción y demás valores. También se puede modificar el color emisivo del objeto, y para que funcione, se debe activar mediante una checkbox.

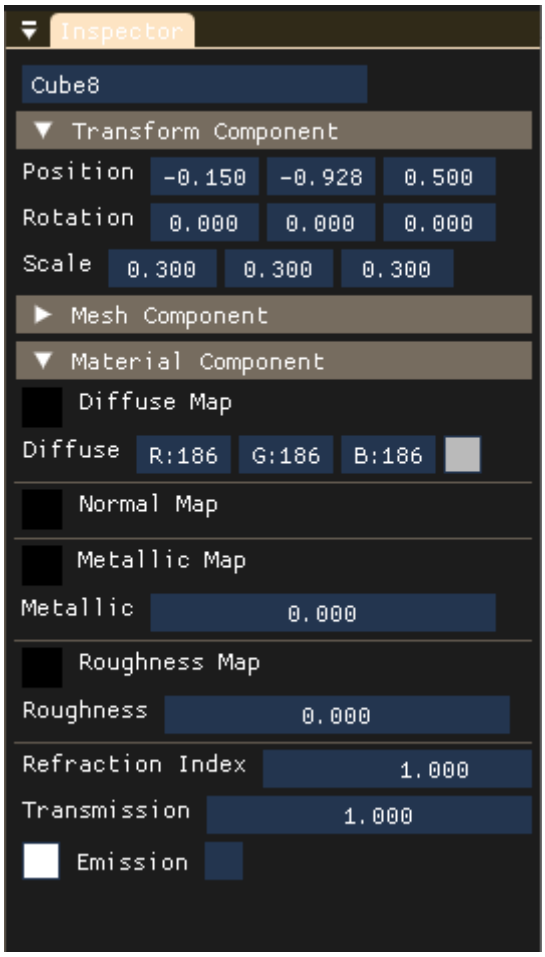

Figura 5.52
Imagen del editor con todas sus funcionalidades visibles.

Por defecto, en la escena hay siempre una directional light, pero también se pueden crear point lights. Tanto para una como para la otra, al seleccionar el objeto, aparece un menú en la derecha donde se permite cambiar el color de esta luz y también su intensidad. De la misma manera que con el resto de objetos, también podemos modificar su posición. En el caso de la directional light, si se modifica la posición no ocurrirá nada, pero si se modifica la dirección sí.

6. Validación del proyecto

Para poder validar este proyecto se ha decidido distribuir la release de este proyecto a diferentes personas del entorno de los videojuegos para que puedan probar la aplicación y posteriormente, realizar un test con una serie de preguntas acerca de la aplicación, tanto del renderizado como del editor, así como si hay alguna funcionalidad que echen en falta. El formulario utilizado para validar el proyecto ha sido el siguiente.

Por lo general, la mayoría de usuarios comentan que la interfaz de usuario se entiende y es fácil de usar ya que se asemeja más o menos a cualquier otro editor, como puede ser Unity o Unreal. Pese a entenderla correctamente, hay varias respuestas acerca de cómo mejorar la interfaz de usuario. Algunos señalan que podría ser algo positivo para la aplicación que se pudieran modificar los colores de los menús, es decir, dar libertad al usuario de crear una plantilla o bien tener un par de plantillas por defecto de diferentes colores y que el usuario escoja la que más desee.

En cuanto a los materiales, todos los usuarios preguntados afirman que los materiales son fácilmente modificables y una gran mayoría de usuarios preferirían añadir el subsurface scattering antes que cualquier otro efecto, seguido del sheen.

En general todos los usuarios creen que la modificación de los materiales está bien y que en algunos aspectos es prácticamente idéntica a otros editores como puede ser Unity, esto en el caso de propiedades como el metallic o el roughness y la emisión, que son valores que también se encuentran en Unity y que como funcionan de la misma manera, pues los usuarios entienden correctamente su función y cómo modificarlos. Luego si que algunos usuarios comentan acerca de algunas propiedades que no son tan usuales en Unity, como la transmisión del objeto, la absorción o el índice de refracción. De estas propiedades comentan que, pese a no estar familiarizados con ellas, se comprenden fácilmente ya que realmente funcionan de la misma manera que otras propiedades, a modo de slider.

Finalmente, acerca del renderizado final, los usuarios creen que el renderizado se ve muy bien y que los resultados son muy realistas y que creen que pueden ser imágenes de gran utilidad por si alguien quiere hacer un renderizado a máxima calidad de una escena propia. Algún usuario sí que ha comentado que podría estar bien añadir la opción de tener un skybox para que el fondo puede cambiar, y así se podrían tener renderizados con más luminosidad en caso de que el skybox sea más de día, o más oscuras si el skybox es más de noche. Argumentan que, en el caso de los skybox de noche, se vería aún más claro como afectan las point lights a la escena y creen que las imágenes que podrían resultar de esa combinación podrían ser increíbles.

7. Conclusiones

Este trabajo ha sido un completo desafío y echando la vista atrás a los objetivos marcados para él, se han conseguido realizar todos y cada uno de ellos. Se ha conseguido crear un editor utilizando la técnica del path tracing para renderizar modelos 3D y posteriormente exportar el resultado de ese renderizado en formato png. El editor permite al usuario modificar y crear escenas, añadiendo modelos propios y modificando los materiales del mismo para poder crear materiales difusos, metálicos o completamente refractivos.

Dicho esto, sí que cabe destacar que debido a todos los cálculos que necesita realizar la GPU, el renderizado es bastante lento, pero es algo que ya se tenía en cuenta al iniciar este proyecto ya que no se hizo con la idea de que fuese un renderer en tiempo real. Pero el hecho de que sea tan lento, dificulta un poco la funcionalidad del mismo, pese a que el resultado que se obtiene es el esperado. También es cierto que este editor funciona con escenas más bien pequeñas, y si se intentan añadir modelos con mucha geometría puede dar muchos problemas.

También cabe destacar, fuera de lo que ha sido el desarrollo del renderer en sí, que la planificación al principio no fue la más adecuada, y es por esa razón por la cual este proyecto ha sido más complejo de llevar a cabo. Una vez se realizó una planificación correcta y con unos plazos de tiempo correctos, facilitó bastante más la tarea general, es decir, la creación del renderer, pero también facilitó bastante la realización de las tareas más específicas ya que fue mucho más sencillo avanzar en el proyecto a base de pequeñas tareas bien marcadas y con unos plazos de tiempo acordes a ellas.

En cuanto al aspecto más personal, estoy satisfecho con este proyecto ya que se han cumplido los objetivos marcados al inicio del mismo, pero sobre todo también por todo lo que he aprendido con él tanto a nivel de programación de C++ como sobre todo de shaders y de la programación gráfica en general. Pese a ya tener conocimientos en la materia, este proyecto ha permitido profundizar aquellos conocimientos que ya tenía, pero a su vez he aprendido cosas nuevas que me han aportado mucho y que sé que también me aportarán mucho en el futuro.

Finalmente, y por resumir un poco todo lo que ha sido este proyecto, ha sido realmente duro la elaboración del mismo, y en muchas ocasiones ha sido difícil continuar ya que el resultado no estaba siendo el esperado pero una vez llegado al final, merece la pena toda la andadura que ha supuesto este trabajo ya que visualmente impacta mucho lo bien que se ven los renderizados. De hecho, creo que este es un tema muy actual y que la industria de los videojuegos se empieza a asomar la cabeza en este terreno y por tanto creo que es necesario seguir investigando este tema e ir aumentando los conocimientos en él.

7.1 Líneas de futuro

En cuanto a los aspectos que se pueden mejorar o en base a diferentes vías que se pueden explorar para mejorar el resultado de la aplicación, creo que es conveniente

dividirlas en dos partes, por un lado, las mejoras que tienen que ver con el editor de la aplicación, y por otra parte las que tienen que ver con el renderizado y con la técnica del path tracing.

En cuanto al editor, creo que podría ser mejorado sobre todo en usabilidad, es decir, podría ser más intuitivo y estar algo más pulido de cara a intentar que sea un producto usable. También, por otra parte, creo que se podría añadir un verdadero resource manager, ya que facilitaría bastante la carga de los modelos al cargar una escena, y al estar en un formato propio en binario, reduciría bastante los tiempos de carga de estos archivos, ya que son bastante pesados.

En cuanto al renderizado y a la técnica del path tracing, creo que la aplicación podría tener mucho más juego añadiendo algunas propiedades más a los materiales y, por tanto, dar más posibilidad de modificar un material para hacerlo lo más realista posible. También sería interesante aplicar alguna técnica más, ya que actualmente solo se pueden realizar objetos difusos, reflectivos o refractivos. Añadir el subsurface scattering podría ser muy interesante debido a que visualmente es un efecto muy atractivo. También se podría añadir el skybox para poder crear ambientes más realistas y permitir al usuario importar los skybox que desee.

Otro punto que se podría mejorar del renderizado es realizar un bounding volume algo más complejo, ya que de esta manera se acortarían los tiempos de renderizado, evitando que se comprueben objetos con los que el rayo no va a colisionar, ya que actualmente solo se comprueba con la AABB de cada objeto, lo cual deja de ser eficiente cuando se tiene una escena muy grande.

También, con más tiempo se podría mirar de optimizar alguna de las fórmulas que se utilizan para calcular el color final de cada píxel en base a cada material. Es posible que alguna de las fórmulas utilizadas se pueda simplificar para así evitar cálculos innecesarios y optimizar el tiempo de renderizado.

8. Bibliografía

- Learn OpenGL PBR Theory. *https://learnopengl.com/PBR/Theory.* Consultado. 14 Dic, 2022
- Learn OpenGL Deferred Rendering. *https://learnopengl.com/Advanced-Lighting/Deferred-Shading.* Consultado. 10 Mar, 2023
- Computer Graphics and Computer Animation: An Overview. *https://ohiostate.pressbooks.pub/graphicshistory/chapter/19-5-global-illumination/#:~:text=Radiosity%2C%20ray%20tracing%2C%20cone%20tracing,of%20light%20between%20diffuse%20surfaces.* Consultado. 12 Mar, 2023
- Nvidia Developer Ray Tracing. *https://developer.nvidia.com/discover/ray-tracing.* Consultado. 8 Mar, 2023
- Ray Tracing in One Weekend. *https://raytracing.github.io/books/RayTracingInOneWeekend.html.* Consultado. 17 Feb, 2023.
- Forward Rendering vs Deferred Rendering. *https://gamedevelopment.tutsplus.com/articles/forward-rendering-vs-deferred-rendering--gamedev-12342.* Consultado. 15 Mar, 2023
- Nvidia. *https://la.blogs.nvidia.com/2022/05/10/que-es-el-path-tracing/.* Consultado. 12 Mar, 2023
- Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2018. *Physically Based Rendering: From theory to implementation.* https://www.pbr-book.org/
- Wikipedia. *https://en.wikipedia.org/wiki/Path_tracing.* Consultado. 12 Mar, 2023
- Wikipedia. *https://en.wikipedia.org/wiki/Ray_tracing_(graphics).* Consultado. 8 Mar, 2023
- Cem Yuksel. https://www.youtube.com/watch?v=gGKup9tUSrU. Consultado. 16 Abr, 2023
- The Cherno. https://www.youtube.com/playlist?list=PLlrATfBNZ98dC-V-N3m0Go4deliWHPFwT. Consultado. 2 Feb, 2023
- Justin Solomon. https://www.youtube.com/watch?v=odXCvJTNn6s. Consultado. 30 Mar, 2023
- BoyBaykiller. https://github.com/BoyBaykiller/OpenTK-PathTracer. Consultado. 12 Abr, 2023
- StackOverflow. https://stackoverflow.com/questions/72187948/bounding-volume-hierarchy-traversal-in-glsl. Consultado. 28 Mar, 2023
- ShaderToy. https://www.shadertoy.com/view/4lfGWr. Consultado. 18 Abr, 2023
- Antongerdelan. https://antongerdelan.net/opengl/compute.html. Consultado. 4 Abr, 2023
- Martin Christen. 2005. Ray Tracing on GPU. https://www.researchgate.net/publication/228770849_Ray_tracing_on_GPU. Consultado. 26 Abr, 2023.

- Ray Tracing: The Rest of Your Life. https://raytracing.github.io/books/RayTracingTheRestOfYourLife.html. Consultado. 8 Abr, 2023.
- Disney Principled BSDF. *https://cseweb.ucsd.edu/~tzli/cse272/wi2023/homework1.pdf.* Consultado. 20 Ago, 2023.
- Implementing the Disney BSDF. https://schuttejoe.github.io/post/disneybsdf/. Consultado. 21 Ago, 2023.
- Learn OpenGL PBR Lighting. *https://learnopengl.com/PBR/Lighting.* Consultado. 27 Jul, 2023.
- The Cherno. *https://www.youtube.com/watch?v=AbVfW4X01a0*. Consultado. 28 Ago, 2023.