Proyecto Final

Instituto Tecnológico de Buenos Aires - ITBA

Ray Tracing: herramienta de edición y renderización basada en física

Proyecto Final de Ingeniería Informática

Autores:

- Tomás E. Ferrer (Leg. Nº 57207)
- Johnathan Katan (Leg. Nº 56653)
- Marcos Lund (Leg. Nº 57159)

Tutor: Gonzalo Dolagaratz

Fecha: Julio 2021

Resumen

El presente informe describe el diseño y la implementación de una aplicación que genera imágenes fotorrealistas en base a escenas tridimensionales. Mediante una interfaz de usuario simple permite configurar interactivamente características del espacio representado en la imagen, como sus propiedades lumínicas o los materiales de objetos que conforman la escena. A través de técnicas de *ray tracing* fundamentadas en la física, y aprovechando el poder de cómputo de las tarjetas de video modernas, la aplicación intenta obtener resultados fieles a la realidad de manera veloz. Frente a la creciente popularidad de estas tecnologías, se busca ofrecer una opción cuya curva de aprendizaje permita brindar una experiencia amigable al usuario.

Índice

1. Introducción

El primer registro histórico de *ray tracing* (trazado de rayos) se remonta a los experimentos del artista Albrecht Dürer en el siglo XVI. A lo largo de su vida escribió múltiples libros en los que exploró diversos conceptos de geometría y sus aplicaciones, y documentó técnicas para alcanzar el realismo en producciones artísticas. En el cuarto libro de su serie *Los Cuatro Libros de la Medida*[1] describe un experimento donde se representa un rayo de luz reflejado por un objeto desde un punto de vista determinado mediante la manipulación de un hilo que pasa dentro un marco hueco, con el objeto a analizar en un extremo y una pared en el otro.

Fuente: metmuseum.org

Figura 1: Experimento de trazado de rayos de luz de Albrecht Dürer.

Siglos más tarde, este principio de trazado de rayos de luz fue incorporado al mundo de la computación gráfica para generar gráficos tridimensionales utilizando una computadora, reemplazando el hilo y el marco hueco por la ecuación de una recta y la ecuación de un plano, respectivamente. En los años 70 se generaron las primeras imágenes con *ray tracing*[2]. El poder de cómputo de los microprocesadores ha crecido desde entonces a un ritmo exponencial siguiendo las tendencias enunciadas por la ley de Moore[3], mientras que su costo se ha reducido a un ritmo similar, posibilitando el acceso a tecnología avanzada a una mayor cantidad de personas. Esto ha permitido que recién a partir de los años 80 comenzaran a popularizarse métodos de generación de imágenes basados en física. A partir de esta década comenzaron

a verse aplicaciones de estas técnicas fuera del ámbito académico. Los primeros ejemplos más representativos tuvieron lugar en la industria del cine, en películas y series de televisión como *Tron (1982)* o *Star Trek II: The Wrath of Khan (1982)*. En la actualidad, empresas como Pixar continúan haciendo uso de estas técnicas en sus producciones y llevan a cabo investigaciones propias[4] para conseguir resultados cada vez más fotorrealistas. La película *Monsters University*, lanzada en 2013, fue la primera película que Pixar produjo enteramente con *ray tracing*, dado que en los filmes anteriores dependían de sombras ubicadas manualmente y otras técnicas que no eran lo suficientemente escalables debido a la creciente complejidad de las escenas en este tipo de películas animadas. A la vez, otras industrias, como la de los videojuegos, están comenzando a implementar *ray tracing* en sus producciones.

Fuente: pixar.com

Figura 2: Iluminación realista de *Monsters University*.

Este trabajo se propone desarrollar una solución que integre estas técnicas de generación de imágenes fotorrealistas con una interfaz gráfica sencilla y fácil de usar que permita la configuración previa de escenas tridimensionales, y asimismo sea performante mediante el uso de hardware especialmente eficiente para este tipo de operaciones. Esto último resulta de suma importancia al tratarse de procesos que se caracterizan por ser computacionalmente intensivos y, en consecuencia, requieren grandes cantidades de tiempo de procesamiento, más aún si se desean obtener resultados que se asemejen lo más posible a la realidad.

En resumen, la aplicación desarrollada busca llegar a usuarios sin conocimientos de computación gráfica ni de *ray tracing* pertenecientes a otras disciplinas/industrias

que puedan sacar provecho de dichas técnicas o, al menos, descubrir el potencial del uso de esta tecnología.

En las siguientes secciones se ahondará en el funcionamiento del algoritmo de *ray tracing*, para luego describir los componentes de la aplicación desarrollada, tanto a nivel funcional como técnico. Consiguientemente se comentarán los detalles del proceso de desarrollo y se analizarán los resultados obtenidos. Finalmente, se incluirá un manual de uso, se explorarán posibles ampliaciones para este proyecto y se presentarán algunas conclusiones sobre el trabajo realizado, los desafíos afrontados y las expectativas iniciales en comparación con los resultados obtenidos.

2. Estado del arte

En primer lugar, se presentará una explicación detallada acerca del algoritmo de *ray tracing* desde un punto de vista teórico. En segundo lugar, teniendo en cuenta los conceptos presentados, se buscará describir soluciones existentes en la industria de la computación gráfica que implementen dichos conceptos.

2.1. Presentación conceptual de *Ray Tracing*

Como se comentó en la sección anterior, la técnica de *ray tracing* busca generar imágenes tridimensionales fotorrealistas a través de la simulación de la luz y su interacción con los materiales pertenecientes a los objetos que se encuentran en la escena. En particular, para la implementación de esta técnica, se busca conceptualizar la luz bajo la teoría de la óptica geométrica, en donde se la considera como rayos originados a partir de determinadas fuentes, sin tener en cuenta su naturaleza ondulatoria. Estos rayos viajan por el espacio y terminan impactando en el ojo de un observador, o siendo absorbidos, reflejados, desviados, o refractados por algún material perteneciente a un objeto. Cuando estos rayos impactan en el ojo de un observador, la fuente del rayo se hace visible para el mismo. A continuación, se presenta un esquema que describe este proceso:

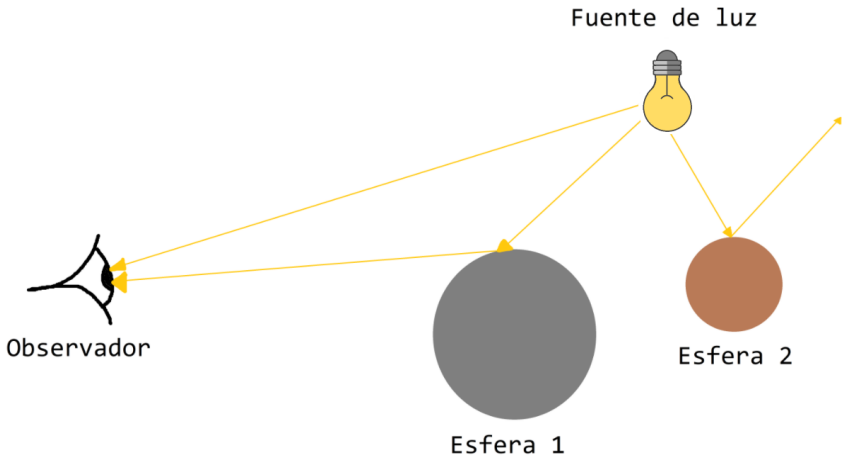


Figura 3: Trayectoria de rayos de luz en la óptica geométrica.

En la figura 3, se observan dos rayos que terminan impactando en el observador. Debido a esta situación, tanto la fuente de luz como la Esfera 1 son visibles para el observador. Por otro lado, se tiene un rayo que se desvía al impactar contra la Esfera 2, y como este rayo no termina impactando en un observador, esta esfera no será visible en la escena.

En la técnica de *ray tracing* se busca simular este comportamiento, pero como lo importante es obtener en la imagen final solamente los objetos visibles para un observador, se simulan los rayos de luz en la dirección inversa, es decir, originándose en el ojo del observador, y finalizando en una fuente de luz, como se muestra a continuación en el siguiente esquema:

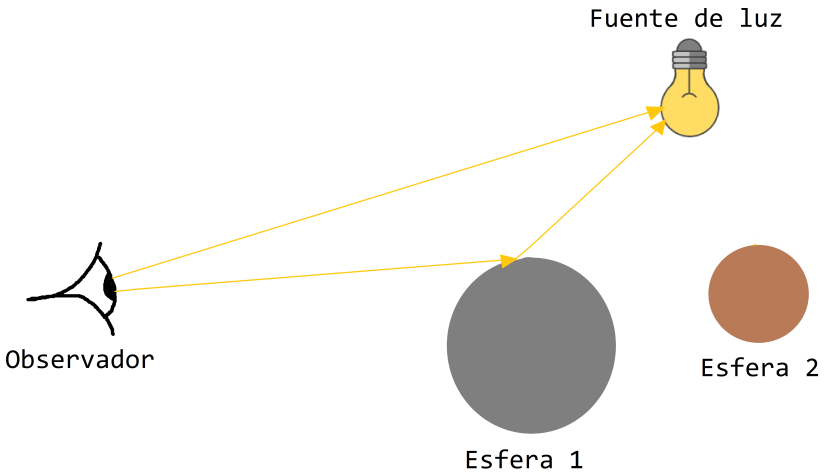


Figura 4: Trayectoria de rayos de luz en un *ray tracer*.

Como se observa en la figura 4, tanto la Esfera 1 como la fuente de luz son visibles para el observador, ya que todos los rayos que se originan a partir del mismo terminan impactando en una fuente de luz. De la misma forma, como ningún rayo originado en el ojo del observador termina impactando contra la Esfera 2 y posteriormente sobre una fuente de luz, entonces esta no será visible en la escena.

Ahora, teniendo estos conceptos en cuenta, para terminar de cubrir el aspecto teórico general del funcionamiento de un *ray tracer*, queda por explicar cómo se determina la cantidad, origen, y dirección de los rayos de luz a simular, cómo se simula la interacción de un rayo con un objeto y cómo se representan los objetos en la escena.

Respecto a la primera cuestión, la cantidad de rayos a simular dependerá de la resolución deseada en la imagen final, es decir, la cantidad de píxeles que componen la misma. Luego, el origen de los rayos dependerá del punto desde el cual se quiera visualizar la escena, es decir, el punto del observador. Y finalmente, para definir la dirección de los rayos a simular, se define un plano de proyección a una determinada distancia del punto del observador. Es sobre este plano de proyección donde se calculará cada uno de los píxeles en la imagen final, de forma similar al experimento descrito por Albrecht Dürer en el siglo XVI referenciado en la sección *Introducción*. La diferencia es que en este caso se utiliza la ecuación de una recta en vez un hilo,

y se define matemáticamente el plano de proyección, en lugar de utilizar un marco hueco. A continuación, se presenta un esquema que describe cómo se originan los rayos desde un observador, y su trayectoria a través del plano de proyección:

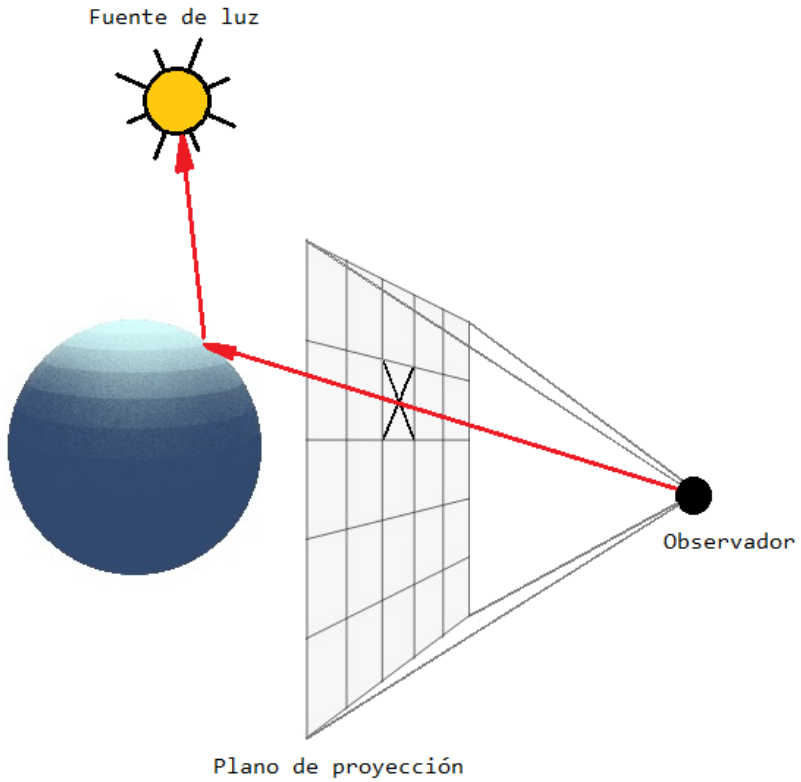


Figura 5: Trayectoria de rayos a través del plano de proyección.

Como se observa en la figura 5, se puede pensar al plano de proyección como una cuadrícula, con cada uno de sus recuadros representando un píxel. Entonces, por cada uno de estos recuadros, se simula un rayo que se origina desde el observador, y su dirección es tal que atraviesa a la posición de dicho píxel en el plano de proyección. Una vez que el rayo atraviesa el plano, se simula su interacción con los objetos de la escena. En el caso de la figura 5, el rayo rebota contra la esfera azul, y luego termina impactando en una fuente de luz. Por lo tanto, el color final que se asignará al píxel atravesado por dicho rayo será azul. A continuación, se presenta un esquema que representa un plano de proyección coloreado con los píxeles resultantes en la imagen final, después de haber simulado todos los rayos necesarios:

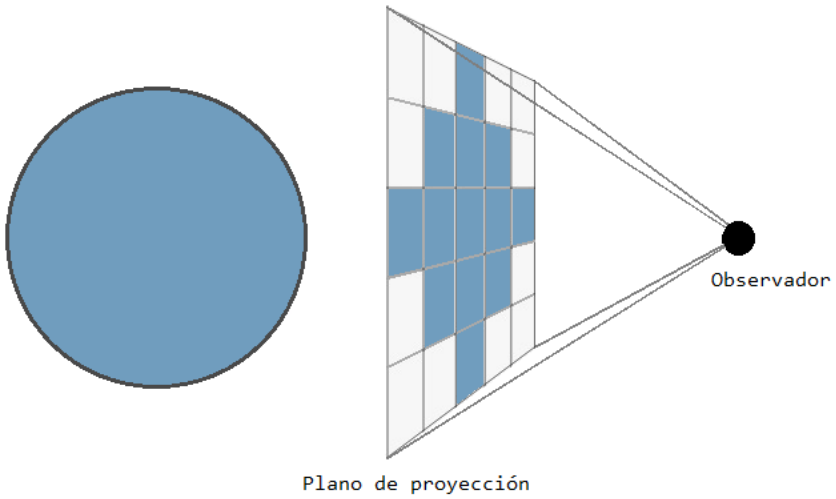


Figura 6: Píxeles resultantes en el plano de proyección.

Los píxeles que se observan coloreados en la figura 6, son los que finalmente se terminan escribiendo en un archivo que representa la imagen final renderizada. Es notable que los bordes de la esfera representada en el plano de proyección no son suaves como los bordes de la esfera real que se quiere representar. Este problema se llama *aliasing*, y se genera debido a que los píxeles tienen un determinado ancho y alto, por lo que los bordes curvos o diagonales de un objeto pueden verse distorsionados. Existen técnicas de *antialising* que buscan evitar este problema y obtener una mejor calidad de imagen. A continuación se muestra una esfera renderizada sin *antialiasing* y otra con *antialiasing*:

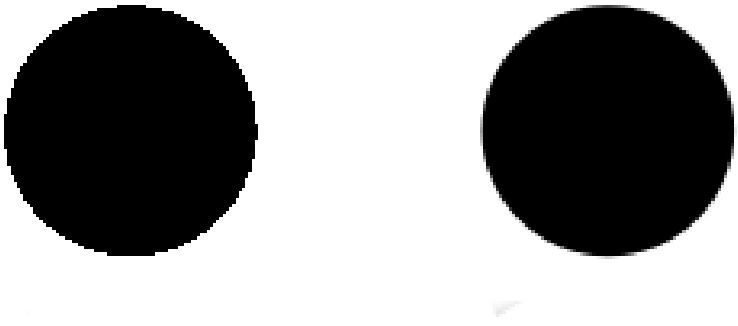


Figura 7: **Izquierda**: esfera renderizada sin técnicas de *antialiasing*; **derecha**: esfera renderizada con técnicas de *antialiasing*.

Respecto a la interacción de la luz con un objeto, esta dependerá de las propiedades de la superficie de dicho objeto. Dependiendo del material que tiene el objeto, la luz podrá ser desviada de determinada manera, absorbida, reflejada o refractada. En las secciones de *Extensión y alcance del proyecto* y *Producto desarrollado* se explica en detalle los materiales implementados y la interacción de la luz con los mismos. El rayo a simular que se origina desde la posición del observador atraviesa el plano de proyección en una posición determinada en función del píxel que se está simulando y, en caso de colisionar contra un objeto, se simulará un determinado comportamiento para el rayo resultante en función del material del objeto. Luego, si el rayo fuera reflejado por el material del objeto, se simularía el nuevo rayo reflejado y la interacción con otros objetos de la escena. En general, en los *ray tracer*, se suele definir un parámetro el cual permite limitar la cantidad máxima de reflexiones, ya que puede suceder que un rayo rebote infinitamente entre dos superficies, generando una sombra sobre una de esas dos, como se muestra a continuación en el siguiente esquema:

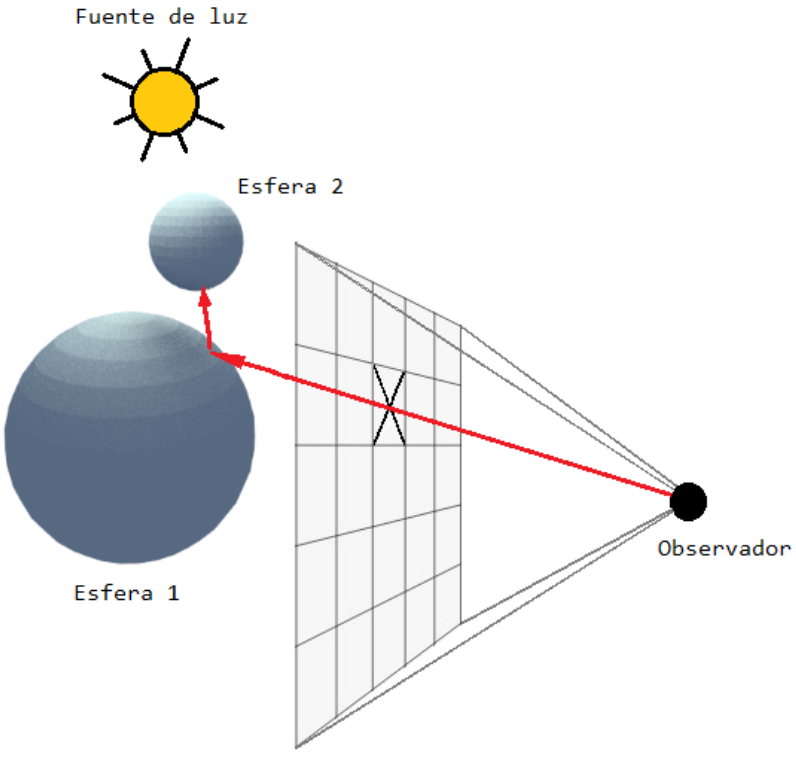


Figura 8: Sombra generada entre dos esferas.

En el caso de la figura 8, el punto atravesado en el plano de proyección, finalmente representará un píxel de color oscuro, ya que el rayo reflejado sobre la Esfera 1

colisionará con la Esfera 2 sin alcanzar una fuente de luz.

En lo que respecta a cómo se representan los objetos en la escena, hasta ahora los ejemplos exhibidos involucraron esferas en todos los casos. Éstas se representan en el espacio a través de la siguiente ecuación:

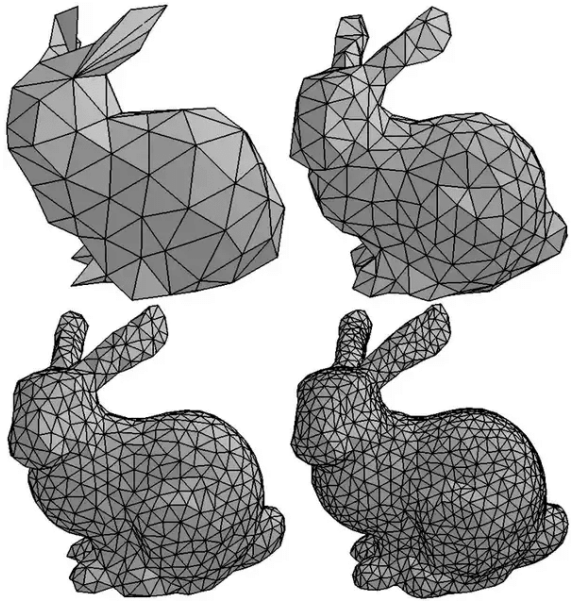
$$(x-a)^2 + (y-b)^2 + (z-c)^2 = R^2$$

donde (a, b, c) son las coordenadas del centro de la esfera, y R es el radio de la misma. Luego, el rayo de luz se puede representar a través de la ecuación de la recta:

$$P(t) = A + t \cdot b$$

donde A es un vector que representa el origen del rayo, b un vector que representa la dirección del rayo, y t un parámetro que indica la distancia de un punto sobre la recta medida desde el origen del rayo. Entonces, para determinar si el rayo colisiona contra la esfera, basta con calcular un valor de t tal que $P(t)$ satisfaga la ecuación de la esfera. En caso de no encontrar este valor, entonces el rayo no colisiona contra la misma.

Si bien se menciona la explicación de la interacción del rayo con la esfera a modo de ejemplo, en un *ray tracer* se intenta renderizar objetos con geometrías más complejas y genéricas. Inicialmente, podría considerarse la posibilidad de representar formas complejas usando como geometría primitiva a una esfera, y componiendo múltiples esferas en el espacio hasta lograr la geometría deseada. Si bien este es un principio que se sigue en los *ray tracer*, se usan triángulos en lugar de esferas ya que los triángulos resultan más útiles que las esferas para representar geometrías compuestas. Por ejemplo, es más fácil representar un plano utilizando dos triángulos que hacerlo utilizando infinitas esferas. Por otro lado, los triángulos se pueden componer generando polígonos de cierta cantidad genérica de vértices, y a su vez, los polígonos se pueden componer para generar geometrías más complejas en el espacio, como un auto, una casa, o una persona.



Fuente: The University of Texas at Austin

Figura 9: Distintas descomposiciones en triángulos de un mismo objeto.

Es por esto que el cálculo de la colisión de una recta con un triángulo es uno de los componentes fundamentales de un *ray tracer*, y por lo tanto deberá ser performante, pues será la operación que más veces se repita. En la subsección *Motor de renderización* dentro de la sección *Producto desarrollado*, se detalla cómo se resolvió este componente, al igual que cómo se representan las geometrías complejas compuestas de polígonos.

2.2. Soluciones existentes

A continuación, se presentan algunas soluciones existentes en el ámbito académico y profesional que implementan técnicas de *ray tracing* para la generación de imágenes fotorrealistas.

La primera solución existente que se destaca, es la implementada por por Matt Pharr, Wenzel Jakob, y Greg Humphreys en su libro *Physically Based Rendering: From Theory To Implementation* [5], por el cual en 2014 ganaron un premio otorgado por la Academia de Artes y Ciencias Cinematográficas debido al impacto que tuvo el libro en la producción de películas en lo que respecta a los sistemas de iluminación y sombreado. El libro describe tanto la teoría como la implementación de un renderizador basado en física, explicando también la resolución de problemas de optimización e iluminación en escenas con determinadas características. Las diferencias entre la implementación presentada en *Physically Based Rendering* y la

presentada en este proyecto son las siguientes: en primer lugar, además de implementar un renderizador también se implementó una interfaz gráfica para facilitar la configuración de sus parámetros. Por el contrario, el libro *Physically Based Rendering* no incluye una interfaz gráfica, aunque el apéndice B contiene una descripción de una interfaz de programación para configurar los parámetros del renderizador. Como este proyecto busca acercar a usuarios no expertos en el área de la computación gráfica, fue fundamental el desarrollo de una interfaz gráfica que no requiera conocimientos específicos del funcionamiento del motor de renderización. La segunda diferencia que se destaca, es el uso de hardware específico para la optimización de operaciones computacionalmente costosas durante la renderización de la imagen. En el proyecto propuesto, se puso foco en implementar la solución sobre tarjetas de video (GPUs). El libro *Physically Based Rendering* omite el uso de hardware específico, y su implementación se limita a la CPU.

Otra solución destacada que se usa ampliamente en la industria del cine y videojuegos es *Blender*, un programa de modelado y animación 3D que tiene un renderizador integrado el cual permite el uso de GPUs. Esto permite generar imágenes fotorrealistas a partir de los modelos y animaciones creadas. Al ser una herramienta diseñada para artistas especializados en el área de la computación gráfica, cuenta con una interfaz gráfica que permite configurar una gran variedad de parámetros, además de permitir el modelado y animación de objetos en una escena virtual. La diferencia que presenta el proyecto implementado en este trabajo, es la propuesta de una interfaz más amigable para usuarios no expertos en el área de la computación gráfica, pero que aún así puedan configurar la escena y generar imágenes fotorrealistas.

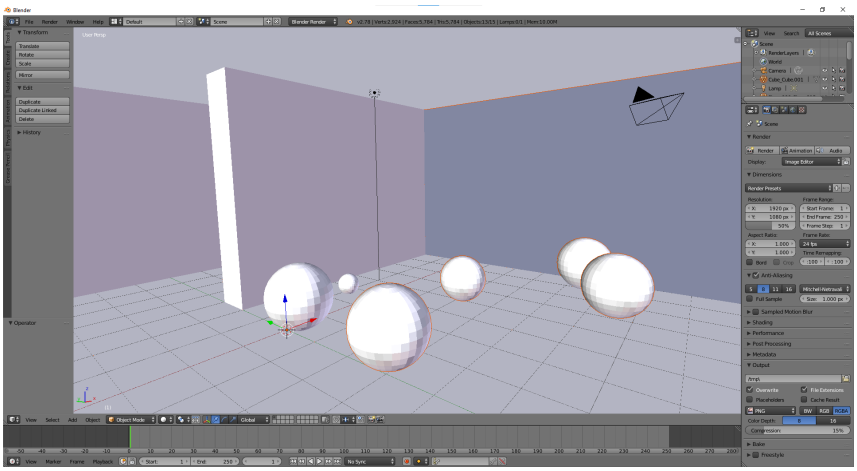


Figura 10: Captura de pantalla de *Blender* con una escena cargada.

Por último, también es importante mencionar al renderizador de uso profesional desarrollado por *Pixar* llamado *Renderman*, uno de los más usados en la industria y con una gran trayectoria, habiendo producido una gran cantidad de películas animadas como *El rey león (2019)* o *Ready Player One (2018)*. Esta implementación, que también cuenta con una interfaz gráfica y soporte para GPUs, está destinada para artistas profesionales que se dedican a la producción de gráficos fotorrealistas. Al igual que con *Blender*, la implementación propuesta en este proyecto final intenta diferenciarse por el hecho de ser más accesible para usuarios no expertos en el área, con una curva de aprendizaje menor.

3. Extensión y alcance del proyecto

Teniendo en cuenta los conceptos introducidos en la sección anterior, a continuación se describe el alcance del *ray tracer* implementado en este proyecto.

La solución implementada cubre los aspectos fundamentales para permitir la renderización de objetos con geometrías complejas. Los mismos se representan como una malla de polígonos, cada uno compuesto por un conjunto de triángulos. Los modelos generados mediante programas de modelado tridimensional son exportados generalmente como archivos de texto plano. En éstos se escriben las coordenadas en el espacio correspondientes a los vértices de los triángulos que forman polígonos. A su vez, los polígonos terminan formando geometrías más complejas, dándole forma al modelo tridimensional final. Como la aplicación desarrollada representa internamente a los objetos como un conjunto de triángulos, permite renderizar modelos tridimensionales a partir de archivos generados por programas externos como *Blender*, *3D Studio Max* o *Maya*. En particular, el tipo de archivo soportado es *Wavefront*[6], usado ampliamente en el campo de la computación gráfica.

También, como se tratará con mayor detalle en la próxima sección del informe, se implementaron tres tipos distintos de materiales, los cuales presentan una interacción diferente con la luz y pueden asignarse a cualquier geometría. El material metálico refleja el rayo de luz incidente sobre la superficie, mientras que en el material dieléctrico, una parte del rayo incidente se refleja, y otra se refracta, dependiendo de su índice de refracción. Este tipo de materiales es útil para representar superficies como vidrio o agua. Finalmente, el lambertiano se utiliza para representar superficies difusas como una bola de billar o una pared.

A su vez, el *ray tracer* implementado soporta dos tipos de iluminación distintos. Por un lado, se tiene la iluminación ambiente, que no se origina de una fuente de luz en particular sino que, como el nombre indica, está presente en toda la escena. Por otro lado, se soportan materiales que emiten luz, a veces nombrados luces en este informe pero no equivalentes a las luces puntuales comúnmente implementadas en otros *ray tracers*. La ubicación en el espacio de estas luces puede ser configurada por el usuario, así como su radio de iluminación.

Como se mencionó en secciones anteriores, se permite la configuración de los materiales mencionados, la posición e intensidad de las luces, y la posición y orientación de la cámara a través de una interfaz gráfica que busca ser amigable para un usuario no experto.

Por último, se implementaron dos optimizaciones en el cálculo de los colores de los píxeles de la imagen resultante de tal manera que pudieran ejecutarse varias pruebas en el menor tiempo posible. La primera de ellas involucra el uso de una tarjeta gráfica para optimizar el procesamiento de cómputos intensivos. La otra consiste en una *estructura de aceleración* que disminuye la cantidad de chequeos de colisiones entre rayos y triángulos, evitando cálculos innecesarios.

4. Producto desarrollado

La aplicación desarrollada cuenta con una GUI (*Graphical User Interface*) que permite cargar y previsualizar una escena tridimensional a partir de un archivo Wavefront (.obj). Esta le permite al usuario ajustar en tiempo real múltiples parámetros de la escena. Una vez obtenida la configuración deseada, la aplicación permite renderizar la escena con *ray tracing*. Al finalizar este proceso se muestra la imagen resultante, permitiendo guardar la misma en formato PPM (*Portable PixMap*)[7].

4.1. Requerimientos

Previo al comienzo de la etapa de desarrollo de este proyecto se establecieron determinados lineamientos y criterios que la aplicación debía cumplir. A continuación se enuncian los mismos, distinguiendo entre aquellos que pueden ser considerados requisitos funcionales y aquellos que pueden ser considerados requisitos no funcionales.

4.1.1. Requisitos funcionales

La aplicación debe permitir al usuario cargar una escena tridimensional generada previamente, y dicha escena debe poder ser previsualizada en la interfaz gráfica. Asimismo debe ser posible realizar alteraciones a la escena, permitiendo que las mismas sean reflejadas en tiempo real. Las alteraciones contempladas son las siguientes:

- Cambiar el material (y las propiedades del mismo) de cada uno de los objetos de la escena por separado
- Usar una textura como material de los objetos
- Cambiar la posición y la rotación de la cámara
- Cambiar el color de la luz ambiente
- Agregar y borrar geometrías con materiales lumínicos
- Cambiar la ubicación y la escala de las geometrías con materiales lumínicos
- Cambiar el radio de iluminación de las geometrías con materiales lumínicos
- Eliminar objetos de la escena

Para poder renderizar la misma escena en distintas sesiones, el programa debe ofrecer algún método de preservación del estado instantáneo de la escena con todos sus parámetros, y también debe permitir recuperar el mismo fácilmente en una sesión posterior.

La calidad del resultado obtenido debe ser alta y, en la medida en que fuera posible, debe aproximarse a la realidad.

Finalmente, la aplicación debe permitir al usuario iniciar el proceso de renderización soportando distintas resoluciones de imagen, cantidades de muestras por píxel y límites para la cantidad de veces que un rayo de luz puede ser reflejado.

4.1.2. Requisitos no funcionales

La aplicación debe contar con una interfaz de usuario que cumpla con las convenciones de diseño y usabilidad de la plataforma donde será ejecutada, permitiendo que la misma se encuentre alineada con el modelo mental del usuario y sea clara y fácil de usar.

El proceso de renderización debe ser performante sin impactar negativamente en la calidad del resultado final. La imagen obtenida debe tener un aspecto realista y no debe incorporar defectos o artefactos visuales indeseados más allá de los esperados (como el *aliasing* o ruido visual), de acuerdo a los parámetros ingresados por el usuario. Para lograr un resultado lo más realista posible es requisito que el *ray tracer* haga uso de alguna técnica de *antialiasing*.

Por último, el programa debe ser lo suficientemente estable como para poder renderizar distintas imágenes en una misma ejecución. Cada escena cargada debe ser independiente de la anterior, pudiéndose renderizar distintas escenas en una misma ejecución.

4.2. Arquitectura

4.2.1. Aspectos técnicos

Esta aplicación busca distinguirse de otros renderizadores por brindar una interfaz gráfica amena al usuario. Dicha interfaz está programada en el lenguaje de programación *C++* y utiliza la librería *Qt*[8]. Esta librería se caracteriza por su facilidad de uso y, además, incluye módulos complementarios que permiten hacer uso de la librería gráfica *OpenGL*[9]. *OpenGL* es una de las librerías más utilizadas mundialmente para representar escenas tridimensionales, y posee algunas ventajas frente a otras: cuenta con una gran trayectoria, habiéndose lanzado su primera versión hace aproximadamente treinta años; es un proyecto abierto soportado por diversos sistemas operativos y por prácticamente todos los fabricantes de hardware. Este proyecto utiliza la versión 4.3 de *OpenGL* (lanzada en 2012) para previsualizar en tiempo real la escena tridimensional a renderizar. Los cambios de configuración efectuados por el usuario a través de la interfaz se aplican en forma automática e instantánea.

Inicialmente se contempló la posibilidad de soportar los tres principales sistemas operativos de computadoras de escritorio: Windows, macOS y Linux, así como las marcas de tarjetas de video con mayor cuota de mercado: *NVIDIA*, *Intel* y *AMD*.

La librería *Qt* está disponible en estos sistemas operativos y es independiente del hardware de video utilizado. *OpenGL* también funciona en los tres sistemas operativos, pero a medida que el proyecto requirió funciones más avanzadas fue necesario subir la versión mínima requerida a 4.3.

El sistema operativo macOS de Apple, no soporta versiones de *OpenGL* superiores a 4.1 por lo que fue necesario descartar la compatibilidad con el mismo. Es importante destacar que algunos drivers de video antiguos tampoco implementan esta versión de *OpenGL*, como los más recientes para la GPU integrada Intel HD 2000 para Windows. No obstante, la mayor parte de los drivers posteriores al año 2012 sí la soportan.

Las primeras versiones del *ray tracer* desarrollado funcionaban sobre la CPU, utilizando un único hilo para computar los píxeles de la imagen resultante. Sin embargo, esta solución demostró ser poco eficiente. Como la performance del sistema fue uno de los atributos de calidad considerados durante el desarrollo, fue necesario investigar otras soluciones. Como los procesadores gráficos (o GPU) son más eficientes para realizar ciertos cálculos que los propios procesadores centrales (CPU) de las computadoras, se optó por delegar el proceso de cómputo mencionado anteriormente a la GPU. Las dos tecnologías más populares para realizar operaciones de cómputo en la GPU son *CUDA* y *OpenCL*, ambas compatibles con Windows y Linux. *CUDA* es una tecnología propietaria de *NVIDIA* que requiere exclusivamente una GPU de este fabricante. Su primer revisión fue lanzada en el año 2006 y en algunas disciplinas dentro de la ciencia computacional, como el aprendizaje automático, es el estándar *de facto*[10].

Por otra parte, *OpenCL* es un *framework* abierto y mantenido por el grupo Khronos (el mismo grupo detrás de *OpenGL*). Al igual que sucede con *OpenGL*, existen distintas implementaciones de esta API, por lo que el soporte de las diferentes versiones de la misma varía en función del fabricante. Mientras que la versión 2.0 de *OpenCL* fue lanzada en 2013, *NVIDIA* no incorporó versiones mayores a 1.2 en sus drivers hasta comienzos del 2021, cuando decidió incorporar la versión 3.0.[11]. Finalmente se optó por implementar el *ray tracer* en *CUDA* considerando que todos los miembros del equipo cuentan con hardware compatible con la versión 11.2, la cual era la más reciente al momento de comenzar el desarrollo. El soporte de *OpenCL* en estos dispositivos era parcial hasta comienzos de este año. Además, *CUDA* cuenta con documentación clara y extensa, y con muchos ejemplos online.

Para simplificar la configuración del entorno de desarrollo se trabajó sobre Windows, debido a que la IDE de Microsoft *Visual Studio* se integra fácilmente con el *toolkit* de *CUDA*. Además se puede configurar el proyecto con la IDE propia de *Qt* llamada *Qt Creator*, aunque la integración de esta IDE con *CUDA* es bastante laboriosa.

La aplicación desarrollada tiene una arquitectura monolítica y está comprendida por un único proyecto de *Visual Studio*. No obstante, puede establecerse una clara

división conceptual entre dos partes de la misma: la interfaz gráfica y el motor de renderización. Ambas partes realizan tareas diferentes y prácticamente funcionan en forma disjunta. En las siguientes dos secciones se detalla el funcionamiento de estas dos subdivisiones conceptuales de la aplicación.

4.2.2. Modelo de clases

A continuación se presentan las distintas clases construidas, basadas en los lineamientos de la programación orientada a objetos e implementadas utilizando las herramientas mencionadas en la sección anterior. Cada diagrama está estandarizado según el lenguaje unificado de modelado (UML) y muestra las propiedades y los métodos de cada clase, junto con sus niveles de acceso, tipos y parámetros. En primer lugar se exponen todas las componentes que forman parte del motor de renderización, y luego aquellas pertenecientes a la interfaz gráfica y relacionadas con *Qt*. En las siguientes secciones se explica detalladamente el funcionamiento de ambos módulos.

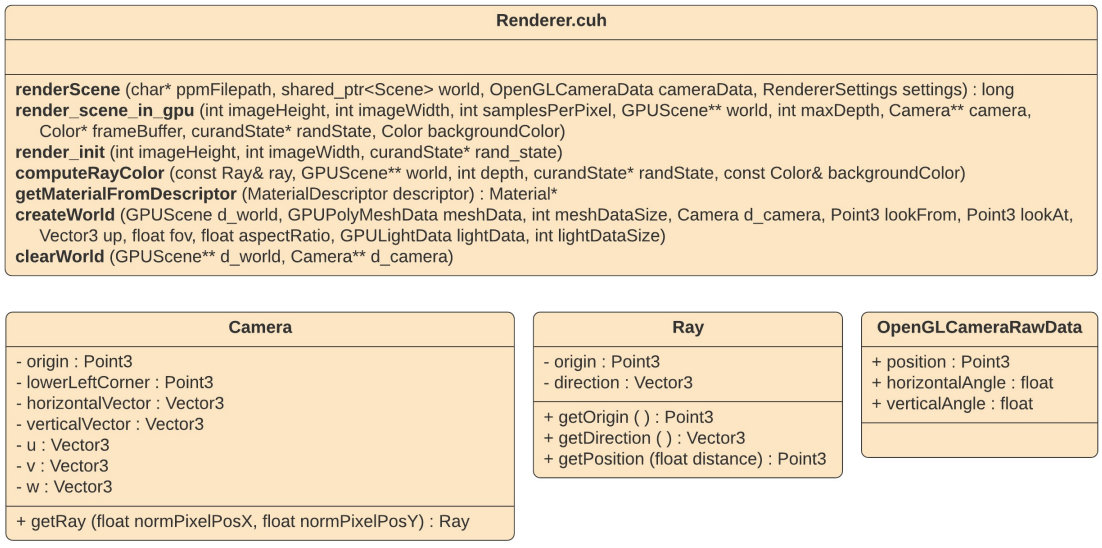


Figura 11: Diagrama UML de clases que representan el flujo básico del *ray tracer* y encabezado del motor de renderización.

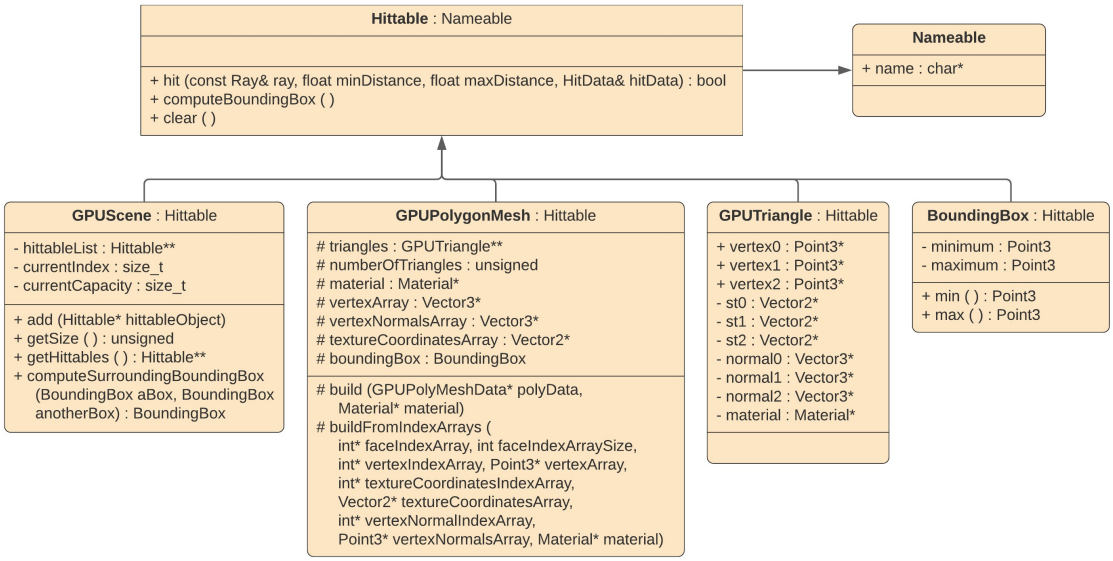


Figura 12: Diagrama UML de clases con las representaciones de las distintas entidades de la escena, almacenadas en la memoria de video.

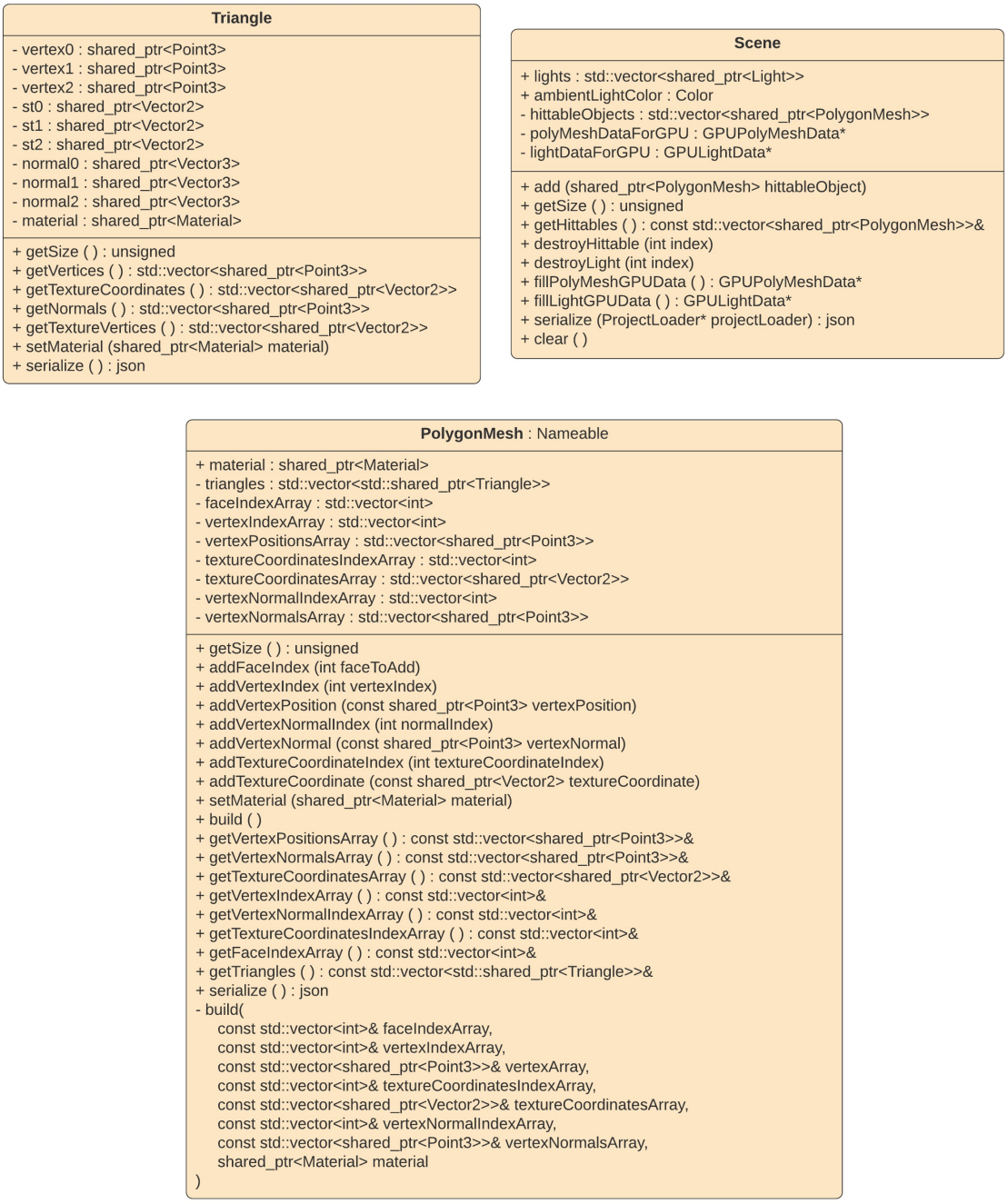


Figura 13: Diagrama UML de clases con las representaciones de las distintas entidades de la escena, utilizadas por la interfaz gráfica.

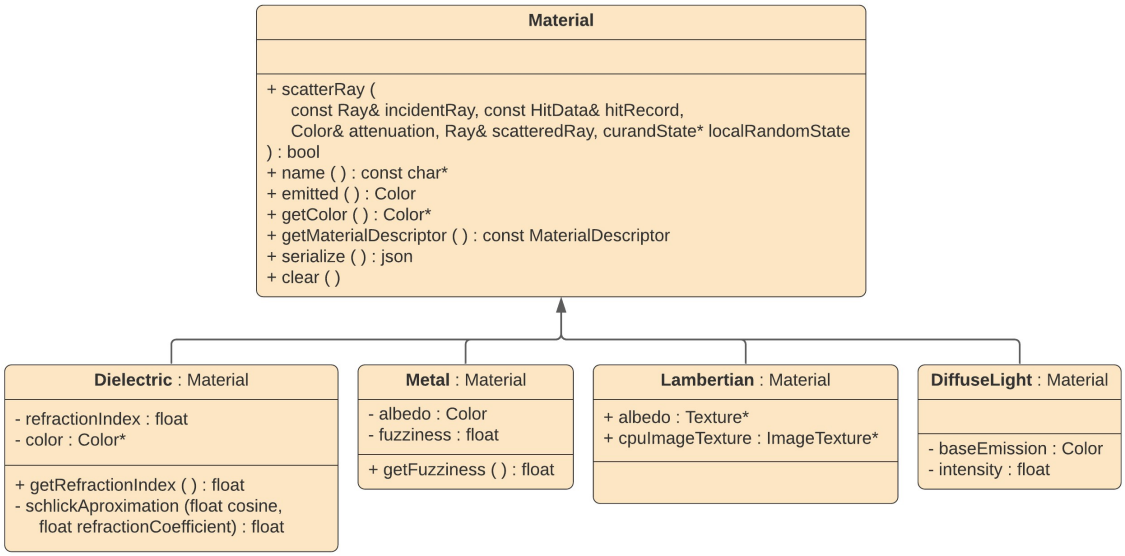


Figura 14: Diagrama UML de clases con los distintos materiales implementados.

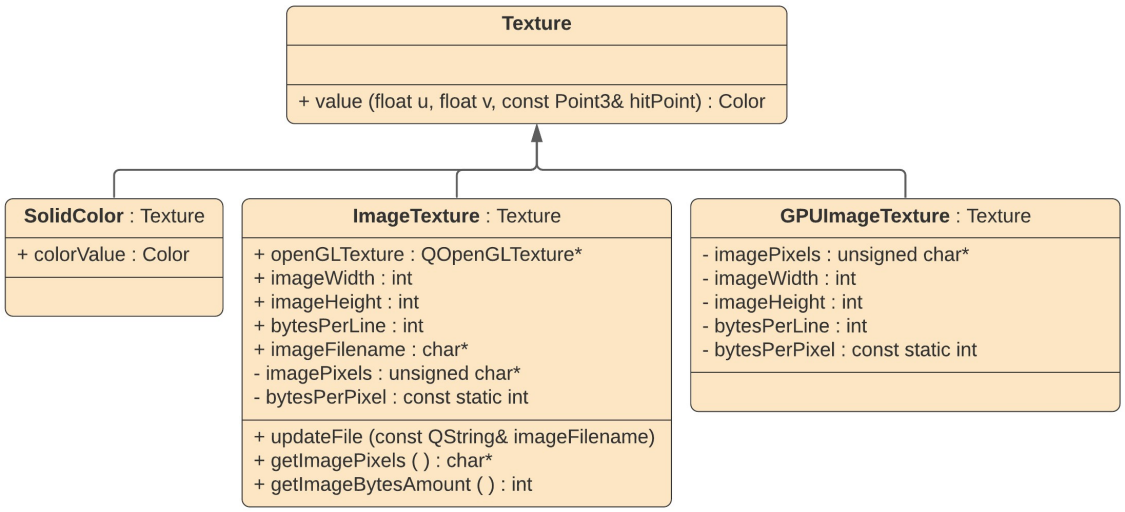


Figura 15: Diagrama UML de clases que representan las texturas de objetos.

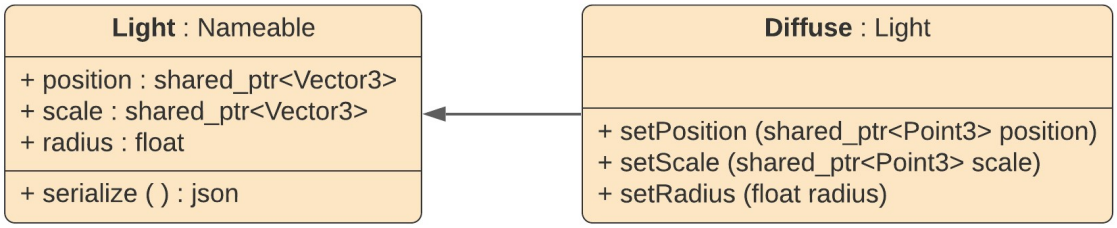


Figura 16: Diagrama UML de clases que representan luces, utilizadas por la interfaz gráfica.

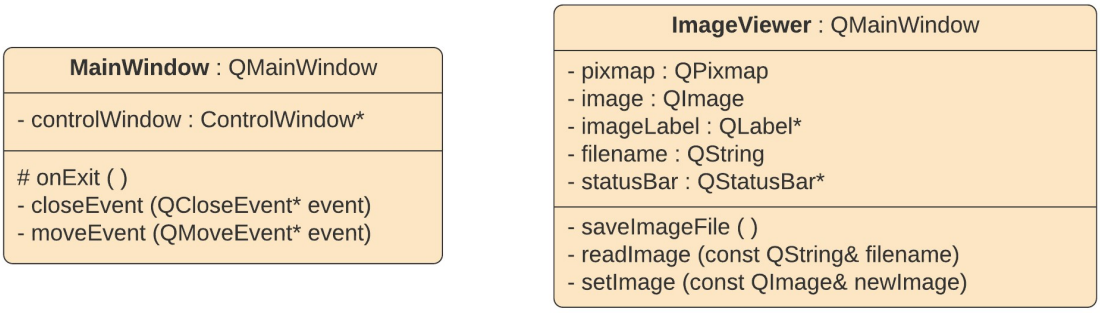


Figura 17: Diagrama UML de clases que representan ventanas independientes.

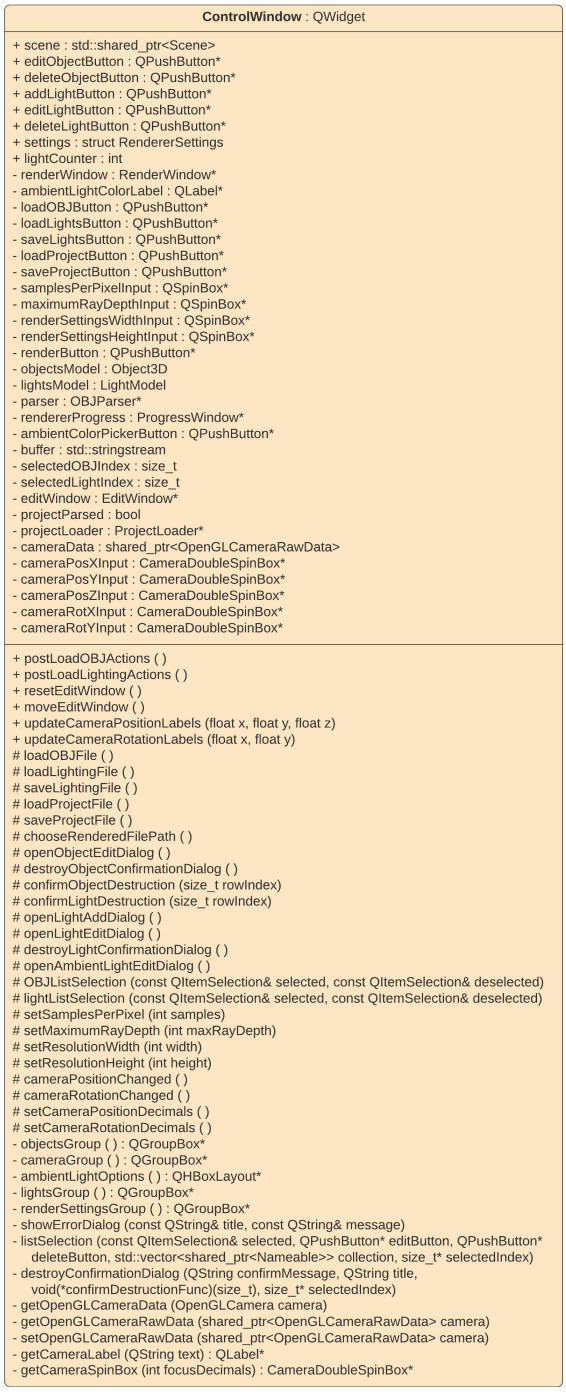


Figura 18: Diagrama UML de clase que representa la ventana principal de la aplicación.

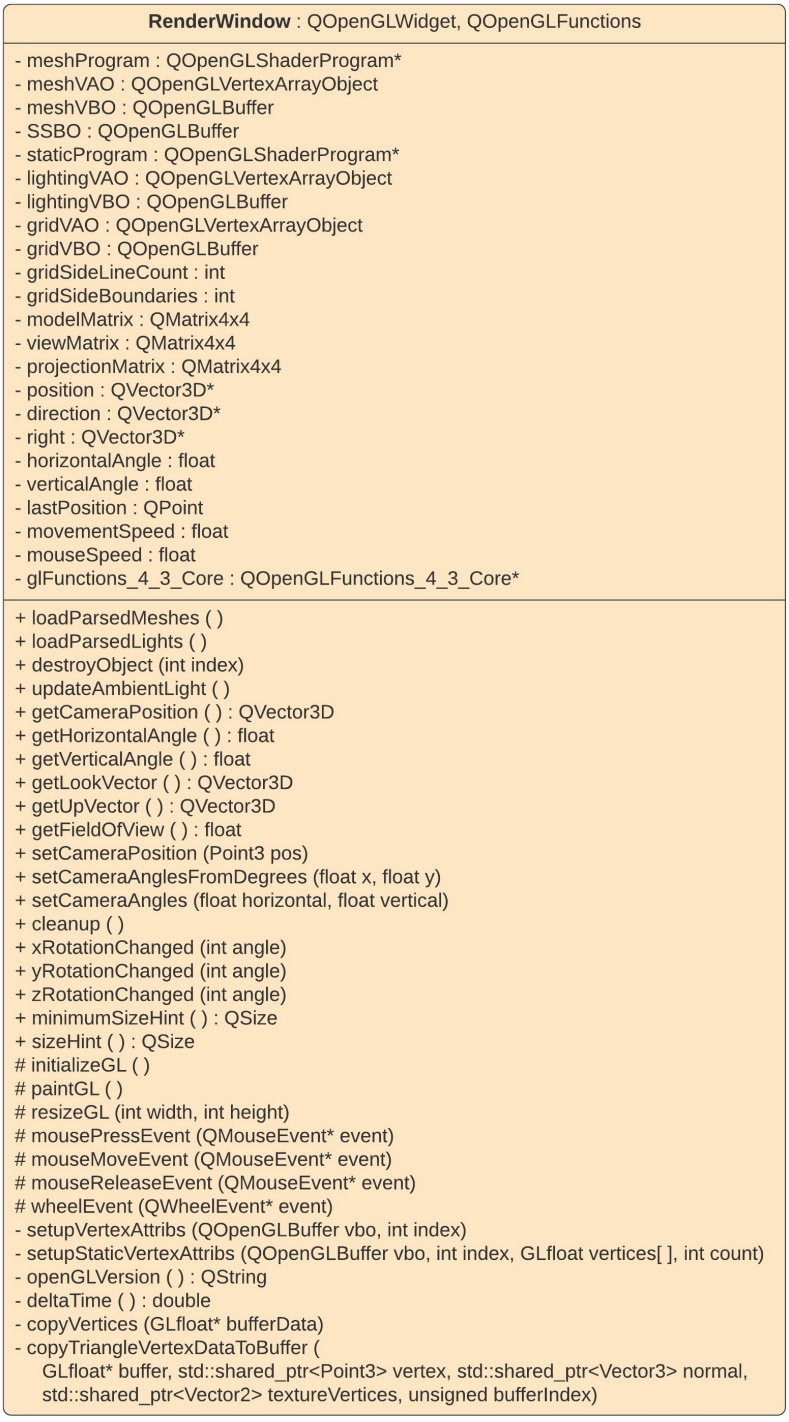


Figura 19: Diagrama UML de clase que representa la previsualización de una escena utilizando *OpenGL*.

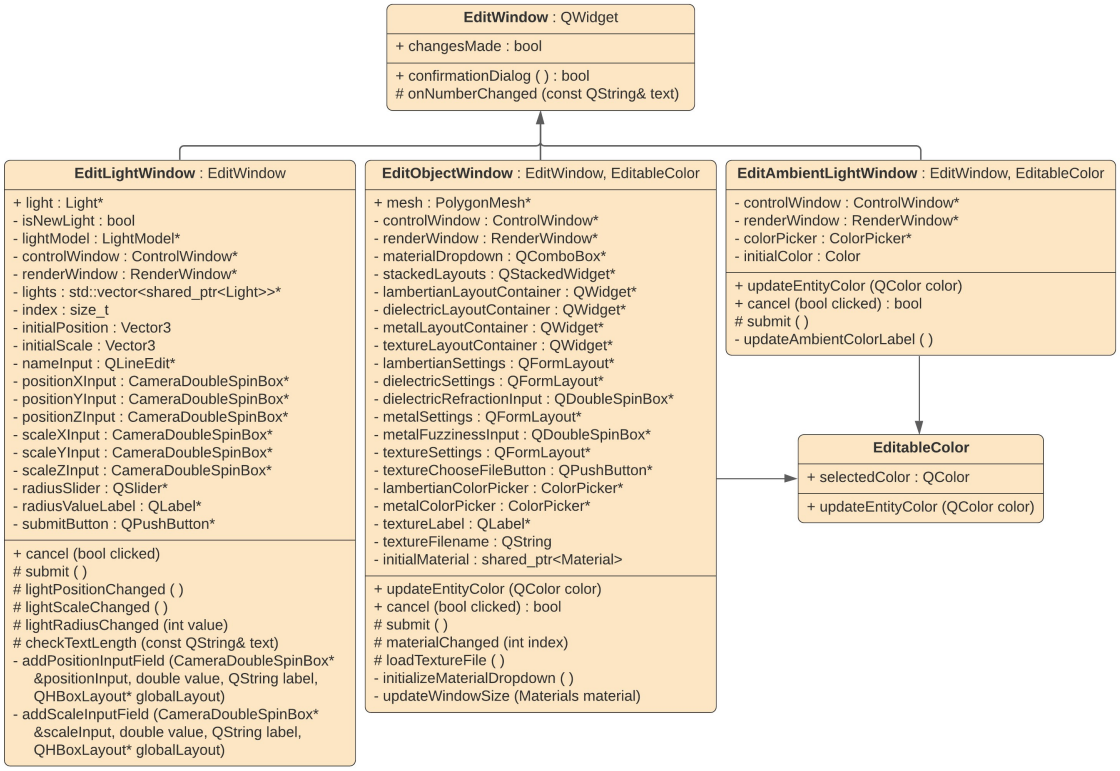


Figura 20: Diagrama UML de clases que representan las ventanas de edición de objetos y luces.

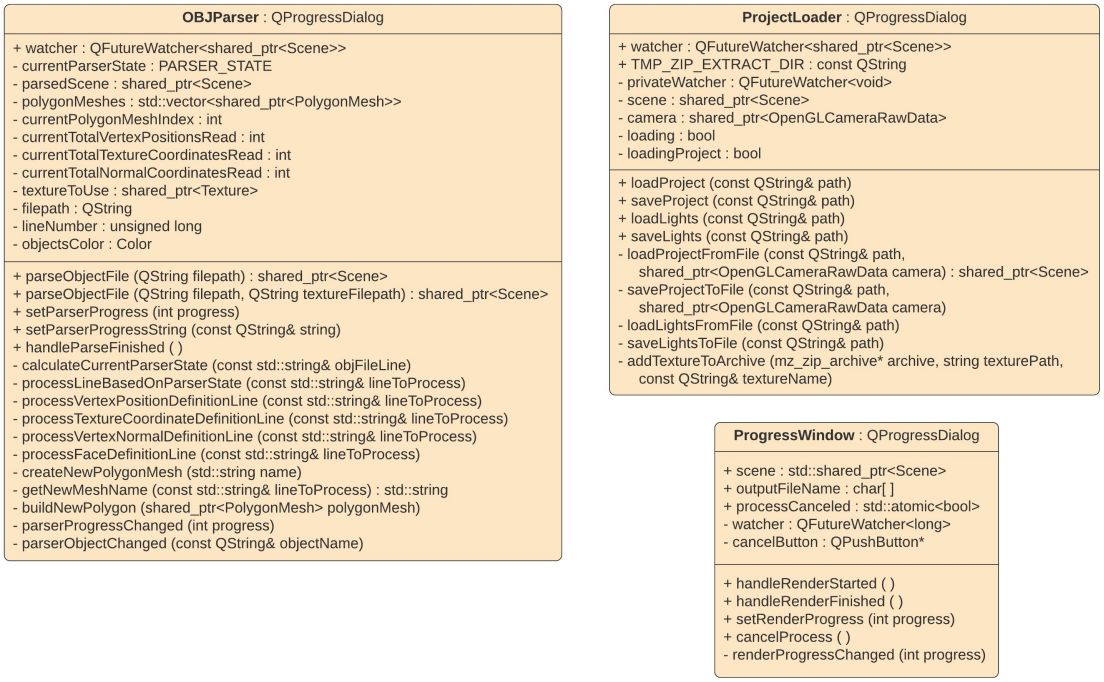


Figura 21: Diagrama UML de clases que representan barras de progreso y su lógica asociada.

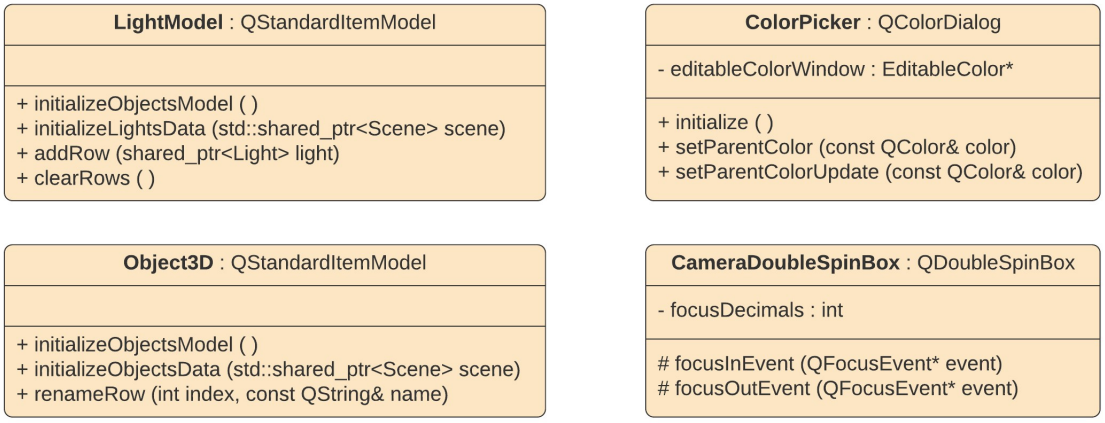


Figura 22: Diagrama UML de clases que proveen funcionalidad auxiliar.

4.3. Motor de renderización

El motor de renderización implementado toma la escena tridimensional y a partir de los parámetros configurados genera una imagen fotorrealista de la misma. En

las siguientes subsecciones se describe en detalle cada uno de los componentes que conforman el motor de renderización.

4.3.1. Algoritmo

Se presenta a continuación el pseudocódigo que describe la lógica general del *ray tracer* implementado:

```
1 List<Hittable> world = loadScene()
2 Camera camera = loadCamera()
3 for (int pixel_i=0; pixel_i < imageWidth; pixel_i++) {
4     for (int pixel_j=0; pixel_j < imageHeight; pixel_j++) {
5         Ray cameraRay = camera.getRay(pixel_i,pixel_j);
6         Color pixelColor = computeRayColor(cameraRay,world);
7         writeColorToOutputFile(pixel_i,pixel_j,pixelColor);
8     }
9 }
```

En la línea 1 del pseudocódigo se declara una variable `world` de tipo `List<Hittable>`, la cual se inicializa con el valor que `loadScene()` devuelve. `Hittable` es una clase abstracta que representa cualquier entidad que puede ser alcanzada por un rayo. Por lo tanto, las clases concretas que heredan de `Hittable` supondrían las entidades que se quieren renderizar, como los triángulos y polígonos. Entonces, la variable `world` representa la lista de objetos que forman parte de la escena a renderizar. La función `loadScene()` representa el nexo entre el motor de renderización y la interfaz gráfica, devolviendo los objetos que fueron configurados desde esta última. Luego, en la línea 2 se declara la variable `camera` de tipo `Camera`. Esta clase implementa un único método: `getRay()`, que toma la ubicación de un píxel de la imagen a renderizar y devuelve una variable de tipo `Ray` que representa la ecuación del rayo que atraviesa dicho píxel. La función `loadCamera()` toma la configuración de la cámara de la interfaz gráfica (lo cual incluye el ángulo y la posición en la escena) y crea una representación de la misma en el motor de renderización.

En las líneas 3 y 4 se observan dos ciclos anidados que iteran desde cero hasta los valores de `imageWidth` (ancho en píxeles de la imagen) e `imageHeight` (alto en píxeles de la imagen) donde `pixel_i` y `pixel_j` representan la posición de un píxel en el plano de proyección de la cámara. Después, en la línea 5 se llama al método `getRay()` de la cámara con la posición de un píxel como parámetro, y el método devuelve un rayo que atraviesa el plano de proyección en dicho píxel. Este rayo se guarda posteriormente en la variable `cameraRay` de tipo `Ray`. En la línea 6 se llama a la función `computeRayColor()`, pasando el rayo calculado y la lista de objetos a renderizar como parámetros. Esta función devuelve el color correspondiente para el rayo de luz calculado según la ubicación en el espacio, la iluminación y los materiales de los objetos con los que colisiona. Dicho color se guarda posteriormente

en la variable `pixelColor` de tipo `Color`. Por último, en la línea 7 se llama a la función `writeColorToOutputFile()`, pasando como parámetros la posición del píxel que está siendo procesado y su respectivo color (calculado previamente). Como el nombre de la función indica, este color es registrado en la posición correspondiente, por lo que luego de iterar por todos los píxeles se obtiene una imagen fotorrealista en formato PPM generada a partir de la escena previamente configurada.

A continuación se presenta el pseudocódigo que describe la lógica de la función `computeRayColor()` para explicar cómo se obtiene el color a partir de un rayo de luz en la escena:

```
1  Color computeRayColor(Ray ray, List<Hittable> world, int
       depth) {
2      if (depth >= MAX_DEPTH)
3          return Color(black);
4
5      HitData hitData;
6      if (world.hit(ray, hitData)) {
7          Ray scatteredRay;
8          Color attenuation;
9          Material hitMaterial = hitData.material;
10         if (hitMaterial.scatterRay(ray, hitData,
11                                    attenuation,
12                                    scatteredRay)) {
13             return attenuation * computeRayColor(
14                 scatteredRay, world, depth+1);
15         }
16
17         return Color(black);
18     }
19
20     return Color(ambientLight);
21 }
```

Se observa un nuevo parámetro que por simplicidad no se había mencionado en el bloque de pseudocódigo anterior: el parámetro `depth`. Este representa la profundidad del rayo que está siendo calculado, entendiendo como profundidad la cantidad de veces que el rayo fue reflejado por los materiales en la escena. En la línea 2 se verifica si esta profundidad supera el valor máximo permitido, y, en caso de superarlo, la función devuelve el color negro. Como se mencionó en la sección *Estado del arte*, se aplica este comportamiento para evitar que un rayo de luz se refleje infinitamente entre dos superficies.

En la línea 5 se declara la variable `hitData`, inicialmente vacía, de tipo `HitData`. `HitData` es una estructura que contiene la información de la colisión entre un rayo de luz y la superficie de un objeto. Esta información consiste en el punto de contacto, la normal al punto de contacto y el material de la superficie de contacto. En la línea 6 se llama al método `hit()` de la variable `world`, que recibe como parámetro

el rayo de luz, y la variable `hitData`. Este método se encarga de chequear si el rayo de luz colisiona con alguno de los objetos de la escena y, en caso de colisionar, devuelve `True` y completa la variable `hitData` con la información de la colisión. Más específicamente, itera por cada uno de los objetos de la lista de `Hittable` llamando al método `hit()` de cada instancia, y si alguno de estos devuelve `True`, significa que el rayó colisionó con un objeto de la escena. Si devuelve `False`, el rayo no colisiona con ningún objeto de la escena y el método retorna en la línea 20 el color de la luz ambiente.

En caso de haber una colisión entre el rayo y un objeto, se calcula un nuevo rayo resultante de la interacción del rayo incidente con el material impactado. Para esto, se toma en la línea 9 el material que fue impactado a partir de la variable `hitData`, asignándolo a la variable `hitMaterial` de tipo `Material`. `Material` es una clase abstracta que implementa el método `scatterRay()`, que recibe un rayo y la descripción de una colisión para dicho rayo, y calcula el rayo resultante dependiendo del material. En la línea 10 se llama a este método de la instancia del material impactado, pasando como parámetros las variables `ray` y `hitData`, y dos referencias a variables no inicializadas: `attenuation` y `scatteredRay`. La variable `attenuation` se declara en la línea 8 y el método `scatterRay()` le asigna el color resultante de la interacción entre el rayo y el material impactado. De la misma forma, se asigna a la variable `scatteredRay` (de tipo `Ray`) el rayo resultante de la interacción entre el rayo incidente y el material.

En caso de que el rayo resultante sea absorbido por el material, el método `scatterRay()` devuelve `False` y a posteriori la función `computeRayColor()` devuelve el color negro. Si el rayo es reflejado o refractado por el material, el método `scatterRay()` devuelve `True` y se vuelve a llamar a la función `computeRayColor()`, esta vez pasándole como parámetro el rayo resultante de la interacción, e incrementando en una unidad la profundidad de rayo en el parámetro `depth`. Luego, el valor de `attenuation` es multiplicado por el color que devuelve el nuevo llamado a la función `computeRayColor()`, de forma que `attenuation` representa la influencia de cada color devuelto en el paso recursivo tiene en el color final del píxel.

Nótese la recursión en la línea 13, donde se invoca a `computeRayColor()` múltiples veces hasta que se supera el límite de la profundidad de rayo o hasta que el mismo no colisiona contra ningún objeto. Se presentó el pseudocódigo de esta forma a modo de facilitar la exposición de la lógica del algoritmo, pero en la práctica fue implementado de forma iterativa. En las siguientes secciones se profundizan los conceptos introducidos en estos fragmentos de pseudocódigo.

4.3.2. Cámara

La cámara, similar al ojo de un observador, es el componente que define el origen y la dirección de los rayos de luz, al igual que el plano de proyección. Este componente

se representa con la clase `Camera`, cuyo constructor tiene la siguiente firma:

```
Camera(Point3 from, Point3 to, Vector3 up, float verticalFOV,
       float aspectRatio)
```

El tipo de dato `Point3` representa un punto en el espacio. El parámetro `from` representa el origen de la cámara, `to` su orientación, `up` representa la dirección vertical en el sistema de coordenadas elegido, y luego `verticalFOV` (campo de visión vertical) y `aspectRatio` (la relación de aspecto entre el ancho y el alto de la imagen) se utilizan para calcular las dimensiones del plano de proyección.

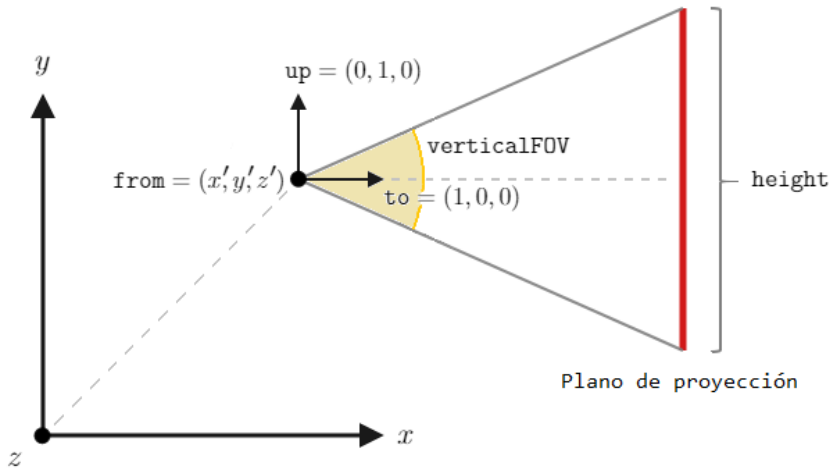


Figura 23: Ejemplo del plano de proyección y variables de la cámara.

Como se observa en la figura 23, la altura del plano de proyección se puede calcular a partir del ángulo representado por la variable `verticalFOV` mediante la siguiente expresión:

$$\texttt{height} = 2 \,.\, \tan \frac{\texttt{verticalFOV}}{2}$$

La variable `aspectRatio` representa la proporción del ancho del plano de proyección respecto de su altura. Entonces, el ancho de este plano se calcula de la siguiente forma:

$$\texttt{width} = \texttt{height} \,.\, \texttt{aspectRatio}$$

Luego, en función de los vectores `from`, `to`, y `up`, se define una base ortonormal descrita por los vectores unitarios $\vec{u}$, $\vec{v}$ y $\vec{w}$, que define la orientación de la cámara mediante las siguientes ecuaciones:

$$\vec{w} = \texttt{from} - \texttt{to}$$
$$\vec{u} = \texttt{up} \times \vec{w}$$
$$\vec{v} = \vec{w} \times \vec{u}$$

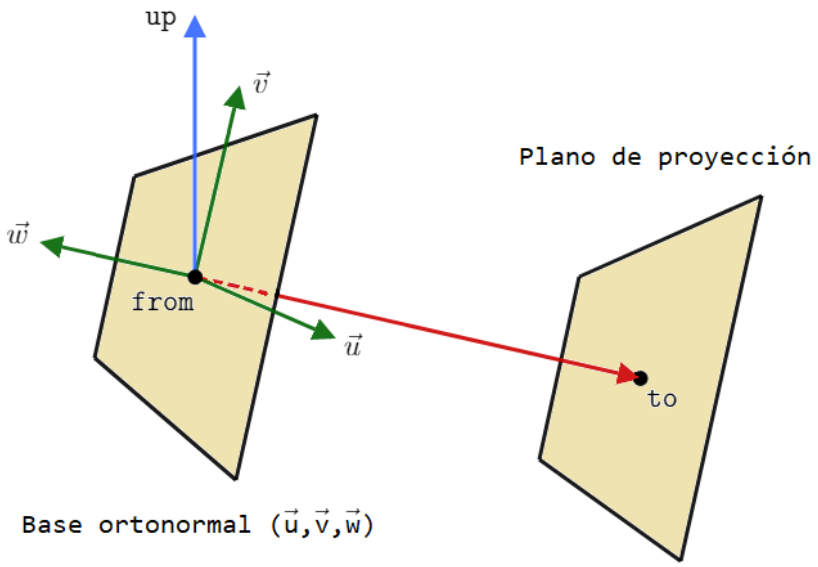


Figura 24: Plano compuesto por los vectores unitarios de la base ortonormal de la cámara, y plano de proyección.

La clase `Camera` también cuenta con un método llamado `getRay()`, cuya firma es la siguiente:

```
Ray getRay(float x, float y) const
```

Los parámetros `x` e `y` representan la posición (x, y) de un píxel en un plano, y el método devuelve la ecuación de un rayo que atraviesa el plano de proyección en la posición correspondiente al píxel. La ecuación del rayo se representa a través de la clase `Ray`, la cual cuenta con dos variables de instancia `origin` y `direction`, esto

es, el origen y la dirección del rayo, respectivamente. Entonces, el método `getRay()` realiza los siguientes cálculos:

$$\texttt{rayOrigin} = \texttt{from}$$

$$\texttt{lowerLeftCorner} = \texttt{rayOrigin} - \frac{\texttt{width} \cdot \vec{u}}{2} - \frac{\texttt{height} \cdot \vec{v}}{2} - w$$

$$\texttt{rayDirection} = \texttt{lowerLeftCorner} + x \cdot \texttt{width} \cdot \vec{u} + y \cdot \texttt{height} \cdot \vec{v} - \texttt{rayOrigin}$$

A partir de la última ecuación se obtiene que la posición del píxel $(0, 0)$ corresponde a la esquina inferior izquierda del plano de proyección, mientras que la posición del píxel $(\texttt{imageWidth} - 1, \texttt{imageHeight} - 1)$ corresponde a la esquina superior derecha. Por lo tanto, a medida que se ejecuta el algoritmo del *ray tracer* presentado en la subsección anterior, se recorren los píxeles por el plano de proyección de abajo hacia arriba y de izquierda a derecha.

4.3.3. Escena

La escena se representa mediante la clase `GPUScene`, que a su vez hereda de la clase `Hittable`, por lo que implementa el método `hit()` que permite verificar si un rayo colisiona contra la escena. `GPUScene` también contiene una lista de instancias de `Hittable` que representa la lista de objetos en la escena. Cuando se llama al método `hit()` de una instancia de `GPUScene`, se itera por dicha lista y se llama a `hit()` para cada uno de los objetos. Si existe una colisión con alguno, retorna `True` e información sobre la colisión. Esta información, como se mencionó previamente, se representa con la clase `HitData`, y contiene el punto de contacto, la normal al mismo y el material de la superficie.

Modelar los objetos en la escena como elementos que heredan de una misma clase abstracta `Hittable` brinda extensibilidad, es decir, no es necesario ajustarse a un tipo particular de representación de objetos, sino que cualquier clase concreta que herede de `Hittable` podrá ser un objeto renderizable en la escena.

4.3.4. Polygon Mesh

Como se mencionó en la sección *Estado del Arte*, se pueden representar geometrías complejas en forma de conjuntos de triángulos. Para representar un conjunto de triángulos que componen la geometría de un objeto, se implementó la clase `GPUPolygonMesh` que hereda de `Hittable`. Esta clase contiene una lista de instancias de `GPUTriangle`, una clase que también hereda de `Hittable`. Esta lista supone un conjunto de triángulos que componen al objeto. Por lo tanto, el método `hit()` de `GPUPolygonMesh` tiene una implementación similar a la de la clase `GPUScene`. Es

decir, se itera por cada uno de los triángulos de la lista, llamando a su vez al método `hit()` de cada triángulo. Luego se devuelve la información del triángulo colisionado más cercano al origen del rayo, en caso de que el rayo haya colisionado con algún triángulo.

Anteriormente se explicó que un triángulo se representa mediante la clase `GPUTriangle`, que hereda de `Hittable`. Como el cálculo de la colisión de un rayo de luz con un triángulo es la operación que se repite más frecuentemente en el algoritmo del ray tracer, se consideraron distintas alternativas para implementar el método `hit()` del triángulo en búsqueda de la más óptima. Inicialmente, se propuso calcular el plano en el que se encuentra cierto triángulo y calcular la intersección del rayo de luz con dicho plano, verificando que el punto de intersección esté dentro de los límites de los vértices del triángulo. Pero los investigadores Tomas Möller y Ben Trumbore idearon en 1997 un algoritmo[12] que se encuentra vigente en la actualidad y que no requiere calcular el plano del triángulo, lo cual resulta más eficiente. Por lo tanto, se optó implementar este algoritmo.

4.3.5. Materiales

El material de un objeto de la escena se representa mediante la clase abstracta `Material`. Esta clase cuenta con un método `scatterRay()` que tiene la siguiente firma:

```
bool scatterRay(Ray& incidentRay, HitData& hitData,
                Color& attenuation, Ray& scatteredRay)
```

La función devuelve `True` si el rayo es reflejado o refractado, o bien `False` si el rayo es absorbido. El parámetro de entrada `incidentRay` representa el rayo incidente. El parámetro `hitData` también es de entrada y es de tipo `HitData`. Por último, los parámetros `attenuation` y `scatteredRay` son de salida. `attenuation` representa el color resultante de la superficie en el punto de contacto entre el material y el rayo mientras que `scatteredRay` representa el rayo resultante de la interacción del rayo incidente con el material. Las instancias de clases que heredan de `Hittable`, como `GPUPolygonMesh` y `GPUTriangle`, cuentan con una variable llamada `material`, cuyo tipo es una referencia a `Material`. Dado el carácter abstracto de la clase `Material`, es posible asignar cualquier tipo de material a cualquier objeto de la escena.

Se han implementado tres materiales: lambertiano, metal y dieléctrico. El material lambertiano, que se asocia a superficies de color difuso, se implementó a través de la clase `Lambertian`. Esta clase, además de implementar el método `scatterRay()`, cuenta con una variable de instancia `albedo`, que representa la fracción de luz reflejada por el material (o su reflectancia). Los autores Tinku Acharya y Ajoy K.

Ray describen a este tipo de materiales como *"superficies desde las cuales la luz es reflejada en todas las direcciones"*[13]. Este tipo de superficies presentan un color uniforme ya que la luz es reflejada uniformemente en todas las direcciones a través de toda la superficie. Entonces, se puede interpretar que todos los rayos incidentes tienen la misma probabilidad de reflejarse en una determinada dirección. En otras palabras, se puede calcular el rayo reflejado partiendo de una dirección aleatoria. Es por esto que, en el siguiente bloque de pseudocódigo, se calcula el rayo reflejado sumando un vector unitario aleatorio al vector normal del punto de contacto:

```
1 bool scatterRay(Ray incidentRay, HitData hitData)
2 {
3     Vector3 direction = hitData.normal + randomDirection();
4     scatteredRay = Ray(hitData.hitPoint, direction);
5
6     return true, scatteredRay, albedo.value(hitData.hitPoint
      );
7 }
```

En la línea 4 se crea un nuevo rayo, cuyo origen es el punto de contacto, y cuya dirección es la dirección aleatoria calculada en la línea anterior. Finalmente, se devuelve dicho rayo y el color del material en el punto de contacto. Este último de obtiene invocando el método `value()` de la variable `albedo` el cual recibe las coordenadas del punto de contacto sobre la superficie del objeto.

Respecto a los metales, la clase `Metal` también tiene la variable `albedo`, que representa el color del metal. Adicionalmente cuenta con una variable `fuzziness`, que representa cuan difuminado es el metal. Un metal con un valor de `fuzziness` nulo refleja perfectamente la luz incidente. Cuando su valor es positivo, la luz no se refleja completamente y por lo tanto el color en la superficie en dicho punto se verá ligeramente difuminado. La reflexión de la luz sobre este material, si su difuminación es nula, se describe en el siguiente esquema:

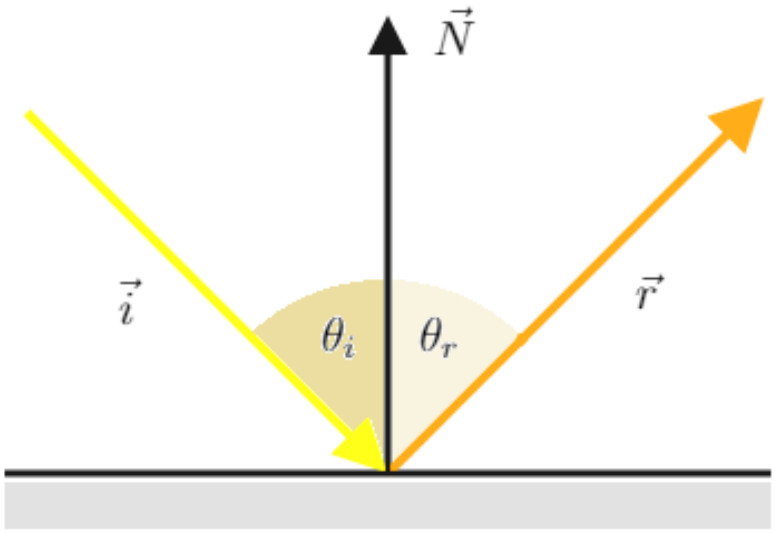


Figura 25: Reflección de un rayo incidente sobre un metal.

El vector $\vec{i}$ representa al rayo incidente y θ_i su ángulo respecto de la normal del punto de contacto. Análogamente, $\vec{r}$ refiere al rayo reflejado y θ_r al ángulo de reflexión. Se busca que el ángulo de reflexión sea igual al ángulo de incidencia, por lo que se puede plantear al rayo reflejado de la siguiente forma:

$$\vec{r} = \vec{i} - 2\,(\,\vec{i} \cdot \vec{N}\,)\,\vec{N}$$

Entonces, el método `scatterRay()` de los metales se describe mediante el siguiente bloque de pseudocódigo:

```
1  bool scatterRay(Ray incidentRay, HitData hitData)
2  {
3      Vector3 direction = reflect(incidentRay.direction(),
4          hitData.normal);
5
6      origin = hitData.hitPoint;
7      newDirection = direction + fuzziness * randomDirection()
8      scatteredRay = Ray(origin, newDirection);
9
10     return true, scatteredRay, albedo;
11 }
```

En el mismo se puede apreciar el uso de la variable `fuzziness` en la línea 6, que se multiplica por una dirección aleatoria y se suma al rayo reflejado. Cuanto mayor es su valor, mayor es la aleatoriedad en la dirección del rayo reflejado, generando el efecto difuminado en la superficie del material.

El tercer material implementado fue el dieléctrico, representado por la clase `Dielectric`. Este tipo de materiales refleja una parte de la luz y refracta la otra según su índice de refracción. En la implementación del método `scatterRay()` de este material, en lugar de retornar dos rayos (uno reflejado y otro refractado), se devuelve un solo rayo que será reflejado o refractado en base a una determinada probabilidad. Christophe Schlick propone un algoritmo[14] para calcular dicha probabilidad en función del ángulo de incidencia, el cual fue implementado en este trabajo. Para el cálculo del rayo refractado, se utilizó la ley de Snell[15].

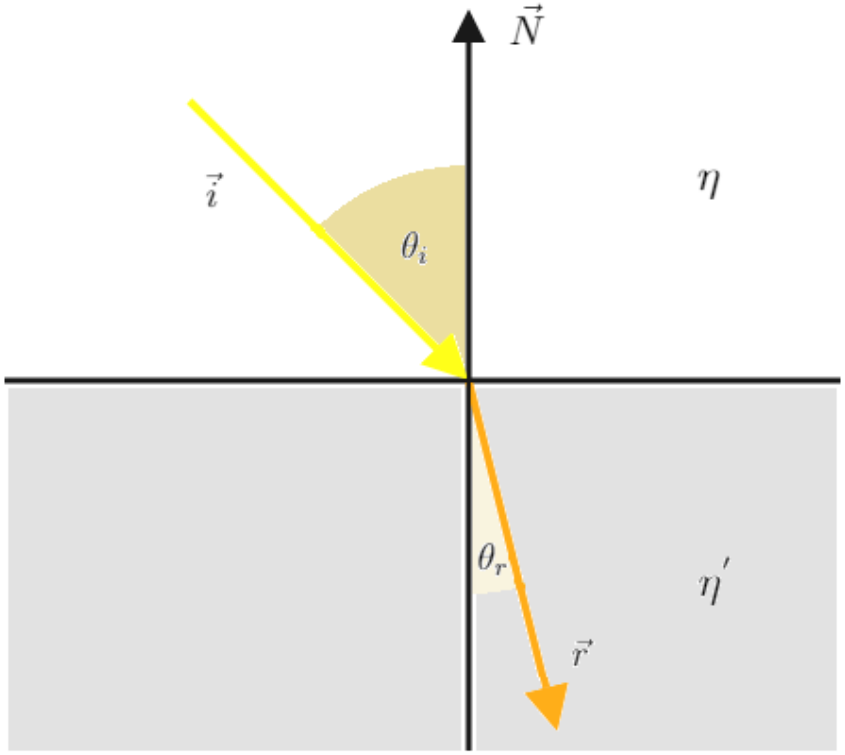


Figura 26: Refracción de un rayo de luz.

Teniendo en cuenta que $\vec{i}$ es el rayo incidente, $\vec{r}$ el rayo refractado, θ_i el ángulo de incidencia respecto de la normal a la superficie de contacto, θ_r el ángulo de refracción respecto de la misma normal, η el índice de refracción del medio del rayo incidente, y η' el índice de refracción del medio del rayo refractado, la ley de Snell establece lo siguiente:

$$\eta \cdot \sin(\theta_i) = \eta' \cdot \sin(\theta_r)$$

A partir de esta ecuación se puede despejar θ_r, y luego obtener la expresión de $\vec{r}$

de la misma forma que para un metal. Por lo tanto, el fragmento de pseudocódigo que representa la lógica implementada para este material se detalla a continuación:

```
1  bool scatterRay(Ray incidentRay, HitData hitData)
2  {
3      float shlickAproximation = schlickAproximation(
4          incidentRay);
5      bool refract = shouldRefract(schlickAproximation);
6      Vector3 direction;
7      if (refract)
8      {
9          direction = refract(incidentRay, hitData.normal);
10     }
11     else
12     {
13         direction = reflect(incidentRay, hitData.normal);
14     }
15
16     scatteredRay = Ray(hitData.hitPoint, direction);
17     return true, scatteredRay, Color(1,1,1);
18 }
```

En la línea 3 se determina si el rayo es reflejado o refractado, calculando la probabilidad de refracción usando la aproximación de Schlick. Luego, en función de este valor se refleja o se refracta el rayo incidente, y finalmente se retorna el rayo reflejado o refractado. En la línea 17 se puede observar que la atenuación es un color `Color(1,1,1)` . Considerando que la atenuación obtenida de los llamados a `scatterRay()` es un factor multiplicativo en el algoritmo principal del *ray tracer*, dicho valor representa una atenuación neutra, es decir, no afecta al color original.

4.3.6. Texturas

El uso de texturas permite envolver una imagen bidimensional alrededor de un objeto. Por ejemplo, si se quiere renderizar un piso de madera, se puede conseguir una imagen o tomar una foto de una superficie de madera y asignarla como textura al objeto que representa el piso.

Como todos los objetos a renderizar se representan como un conjunto de triángulos, se necesita determinar en primer lugar qué fragmento de la textura se debe proyectar sobre cada triángulo. Entonces, cada triángulo tiene asociado a cada uno de sus vértices un *texture coordinate* que representa la coordenada del mismo en el plano de la textura. El *texture coordinate* es un vector de dos dimensiones cuyas coordenadas están normalizadas en el rango $[0, 1]$.

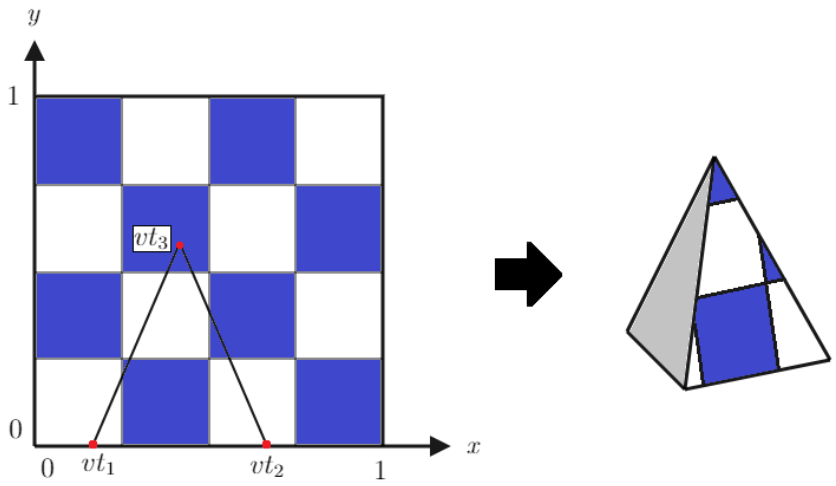


Figura 27: Triángulo proyectado sobre una imagen de cuadros, en función de sus *texture coordinates.*

En la figura 27 se observan las *texture coordinates* representadas por las coordenadas vt_1, vt_2 y vt_3. Si un triángulo de cierto objeto usa estas coordenadas, su superficie será cubierta por el fragmento de la imagen que se interseque con el mismo.

Por otro lado, resta definir el color que debe tener cierto punto sobre el triángulo dada una colisión con un rayo de luz en función de la textura asignada. Para esto se determinan las coordenadas (u, v) de dicho punto sobre el triángulo a lo largo de los ejes x e y respectivamente. Estas coordenadas (u, v) también se encuentran normalizadas en el rango $[0, 1]$. Por lo tanto, cuando un rayo colisiona con un triángulo basta con calcular las coordenadas (u, v) del punto de contacto y, utilizando las *texture coordinates* mencionadas previamente, se obtiene un píxel que representa el color a mostrar en el punto (u, v) del triángulo.

El cálculo de las coordenadas (u, v) se realiza en el método `hit()` de la clase `GPUTriangle`, a partir del algoritmo de colisión de rayos con triángulos de Möller y Trumbore. Estas coordenadas se almacenan en la estructura `HitData` (que contiene la información de la colisión) para luego poder ser utilizadas en el método `scatterRay()` del material. En particular, la clase `Lambertian` utiliza las coordenadas (u, v) para calcular el color a partir de texturas. Como se mencionó anteriormente, esta clase cuenta con una variable `albedo` de tipo `Texture`, la cual contiene un método `value()` que recibe las coordenadas (u, v). Si se desea mostrar un color sólido en lugar de una textura, es posible asignar una instancia de la clase `SolidColor` (que hereda de `Texture`) a la variable `albedo`. Esta clase contiene una variable `colorValue` que representa un color. En ese caso, el método `value()` ignora las coordenadas (u, v) y devuelve únicamente dicho color. En cambio, si se quiere utilizar una textura en lugar de un color uniforme, se debe hacer uso de la clase

`GPUTexture` (que también hereda de `Texture`) asignándose a la variable `albedo` del material. Esta clase tiene una variable `imagePixels` que contiene los píxeles de la imagen elegida. El método `value()` de `GPUTexture` utiliza entonces las coordenadas (u, v) recibidas como parámetro para indexar un píxel en la imagen y lo retorna. Finalmente, el color devuelto por el método `scatterRay()` del material resulta ser el píxel correspondiente a la textura en función de las coordenadas de colisión (u, v) del triángulo.

En resumen, la interacción de la luz con un objeto al cual se le ha asignado una textura resulta análoga a la interacción con superficies lambertianas, excepto porque se devuelve un píxel que pertenece a una imagen en lugar de un color uniforme.

4.3.7. Iluminación

Como se mencionó en secciones anteriores, el *ray tracer* cuenta con soporte para iluminación ambiente y materiales lumínicos. En la subsección *Algoritmo* se explicó que el método `computeRayColor()` se encarga de calcular el color a devolver para un rayo que se origina desde la cámara. En caso de que el rayo de luz no colisione con ningún objeto, se devuelve el color de la luz ambiente. La misma es configurable desde la interfaz gráfica y siempre se encuentra presente en la imagen final. Si se quiere simular una escena oscura es necesario asignar el color negro como color de la luz ambiente.

La luz emitida por los materiales lumínicos tiene un determinado rango de influencia configurable y es posible agregar tantas de estas luces como se deseen. Este material está representado por la clase `DiffuseLight` (que hereda de `Material`) y cuenta con dos variables de instancia: `baseEmission` e `intensity`. `baseEmission` representa el color blanco según sus componentes RGB mientras que `intensity` es de tipo `float` y funciona como un multiplicador para `baseEmission`, el cual amplifica o reduce el rango de influencia de la luz emitida en función de su intensidad. Respecto al método `scatterRay()` de este material, el mismo devuelve `False`: esto significa que el rayo no debe seguir reflejándose o refractándose. Anteriormente se mencionó que si un rayo es absorbido el método devuelve el color negro. Se explicó de esta forma a modo de simplificación, ya que todavía no se habían introducido los materiales lumínicos. En este caso, cuando la luz deja de ser reflejada o refractada por un material, en lugar de devolver el color negro se devuelve un color llamando al método `emitted()` de la clase `Material`. De esta forma, todos los materiales que no representan fuentes de luz y que absorben el rayo devuelven un valor de `emitted()` igual al color negro. Para los materiales de tipo `DiffuseLight`, en cambio, el valor devuelto por `emitted()` es `baseEmission` multiplicado por `intensity`. De esta forma es posible representar materiales que absorben luz y materiales que emiten luz.

4.3.8. CUDA

Inicialmente no formaba parte del alcance del proyecto desarrollar el motor de renderización para que se ejecute sobre la GPU, por lo que en una primera versión se desarrolló únicamente para que ejecute en la CPU. Luego de evaluar distintas técnicas de optimización, tanto de software como de hardware, se decidió modificar la implementación para que la aplicación hiciera uso de la GPU. La migración del motor de renderización a *CUDA* partiendo de la implementación para la CPU trajo consigo varios desafíos, tanto conceptuales como técnicos. El mayor desafío conceptual fue cómo transferir la información que se configura desde la interfaz gráfica, la cual se encuentra en la memoria principal del sistema accesible únicamente para la CPU, a la memoria interna de la GPU para poder renderizarla.

Para esto se implementaron dos métodos en la clase `Scene`: `fillPolyMeshGPUData()` y `fillLightGPUData()`. Los mismos se encargan de devolver una estructura que permita representar la escena en la GPU partiendo de la escena almacenada en la memoria principal. Las estructuras subyacentes de esta representación se denominan `GPUPolyMeshData` y `GPULightData`, y son generadas por la función `renderScene()` en el archivo `Renderer.cu`, el cual es el punto de entrada desde la interfaz gráfica al motor de renderización. Dentro de esta misma función, se llaman a otras funciones que ejecutan código dentro de la GPU, en particular `createWorld()`. La función `createWorld()` recibe la configuración de la cámara, de las luces y de los objetos en la escena a través de las estructuras `GPUPolyMeshData` y `GPULightData` y prepara la representación para la GPU.

Otro desafío fundamental surgido de la migración a *CUDA* fue la adaptación del código original para poder aprovechar la capacidad de ejecución de código paralelo en la GPU. Para esto se definió primero en `renderScene()` la cantidad de hilos de ejecución en base a la cantidad de píxeles a renderizar de tal forma que cada hilo se encargue de renderizar un solo píxel. Luego, se definió una variable llamada `frameBuffer`, que consiste en una lista inicialmente vacía de instancias de `Vector3` donde cada vector representa un color RGB. Luego, la función `renderSceneInGpu()` se encarga de ejecutar los hilos de forma paralela en la GPU. Ésta recibe como parámetros la escena cargada en la GPU, la configuración de la cámara y la lista `frameBuffer`, la cual representa un espacio de memoria compartido entre todos los hilos de ejecución. Cada hilo calcula el color de un píxel y guarda el color calculado en una determinada posición dentro de la lista `frameBuffer`. De esta forma, cuando todos los hilos de ejecución finalizan, la variable `frameBuffer` contiene una lista de píxeles que representan la imagen final renderizada. Por último, se escribe esta lista de píxeles que representan la imagen final en un archivo de formato PPM plano.

4.3.9. Multisampling

Como solución al problema de *antialiasing* se decidió implementar *multisampling*, o multimuestreo. Para esto se modificó la forma en la que se calculan los rayos que atraviesan el plano de proyección en la posición de un determinado píxel. Como un píxel no es puntual sino que tiene un ancho y un alto, el rayo que pasa por el plano de proyección atraviesa el centro de las dimensiones del píxel. A partir de la técnica de *multisampling* implementada, en lugar de calcular un sólo rayo que pasa por el centro del píxel, se calculan múltiples rayos tomando distintas muestras de la posición del píxel. Luego, todos los colores obtenidos para dichos rayos se promedian para obtener el color final del píxel.

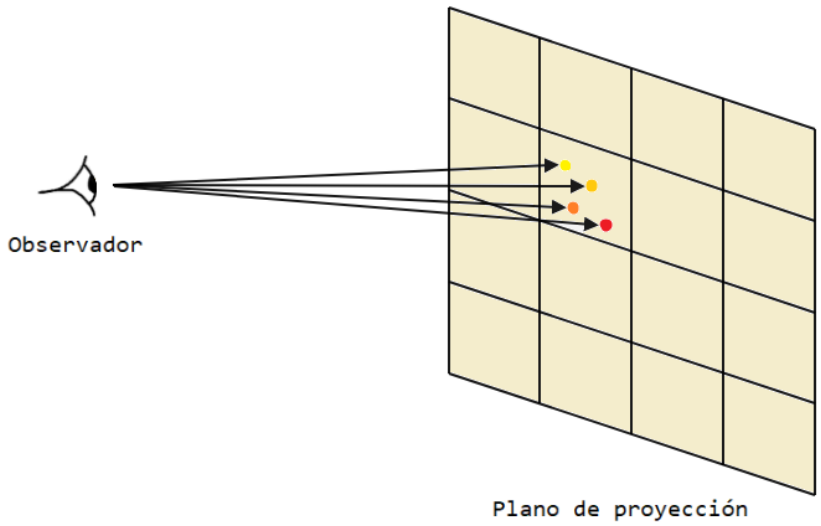


Figura 28: Múltiples muestras calculadas para un mismo píxel.

Entonces, el pseudocódigo del *ray tracer* presentado en la sección *Algoritmo*, luego de implementar *multisampling* queda definido de la siguiente forma:

```
1 List<Hittable> world = loadScene()
2 Camera camera = loadCamera()
3 for (int pixel_i=0; pixel_i < imageWidth; pixel_i++) {
4     for (int pixel_j=0; pixel_j < imageHeight; pixel_j++) {
5         Color pixelColor;
6         for (int sample=0;sample<samplesPerPixel;sample++){
7             float pixel_x = pixel_i + random();
8             float pixel_y = pixel_j + random();
9             Ray cameraRay = camera.getRay(pixel_x, pixel_y);
10            pixelColor += computeRayColor(cameraRay, world);
11        }
12    }
13    pixelColor /= samplesPerPixel;
14    writeColorToOutputFile(pixel_i, pixel_j, pixelColor);
15 }
```

A partir de la línea 6 se introduce un nuevo ciclo, que toma las muestras deseadas. La función `random()` devuelve un número real entre cero y uno con distribución uniforme. Luego se suman los colores obtenidos para cada píxel en la línea 10, y finalmente en las líneas 13 y 14 se promedian los colores y se escribe el resultado en el archivo de salida. Esto se realiza en la función `renderSceneInGpu()` mencionada en la sección anterior.

4.3.10. Volúmenes envolventes

Se implementaron volúmenes envolventes como técnica de optimización en el cálculo de la colisión de rayos de luz con objetos. Previo a la implementación de esta técnica, se iteraba por todos los triángulos de la escena para cada uno de los rayos, devolviéndose el triángulo colisionado. Esto resulta en tiempos de ejecución significativamente elevados, sobre todo en escenas con una gran cantidad de triángulos.

La técnica de volúmenes envolventes consiste en calcular, para cada objeto, un volumen que lo contenga llamado *bounding volume*. Luego, para determinar si un rayo colisiona o no con un objeto, se calcula en primer lugar si existe una colisión con este volumen contenedor. Si el rayo no colisiona con el mismo, se verifica si existen colisiones con otros volúmenes envolventes. De esta forma, se simplifica el proceso al evitar iterar por todos los triángulos del objeto. En caso de que el rayo colisione con el *bounding volume*, se procede a iterar por todos los triángulos contenidos dentro del mismo para encontrar el verdadero punto de colisión.

En la implementación realizada se optó por utilizar cubos como *bounding volumes* los cuales se calculan al momento de instanciar un objeto a partir de dos puntos que representan los extremos del cubo. Estos puntos se calculan tomando las coordenadas mínimas y máximas de todos los triángulos del objeto.

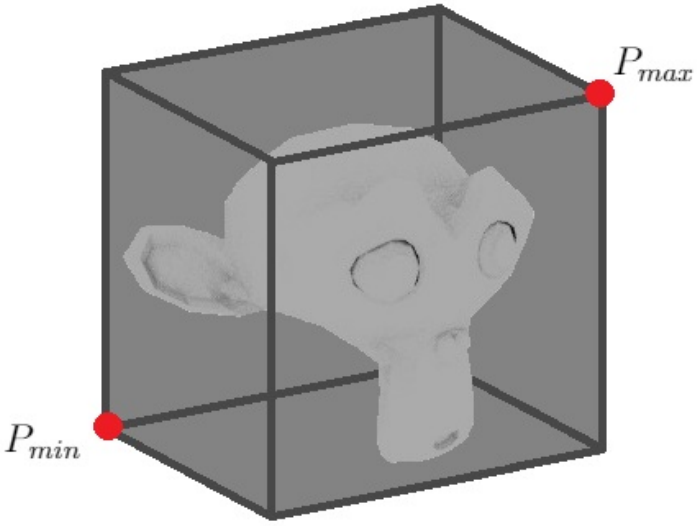


Figura 29: Volumen envolvente calculado para un objeto a partir de dos puntos P_{min} y P_{max}.

A continuación se presenta la implementación de esta técnica dentro del método `hit()` de la clase `PolygonMesh`, que determina si un rayo colisiona con un objeto:

```
1  bool hit(Ray ray)
2  {
3      if (!boundingVolume->hit(ray)) {
4          return false;
5      }
6
7      for (int i=0; i<triangles.length; i++) {
8          if (triangles[i]->hit(ray)) {
9              return true;
10         }
11     }
12
13     return false;
14 }
```

En la línea 3, la variable `boundingVolume` representa el volumen envolvente ya inicializado. Es de tipo `BoundingBox` que a su vez hereda de `Hittable`, por lo cual implementa el método `hit()`. Esta implementación del método calcula la intersección de una recta con un cubo. Por lo tanto, la línea 3 verifica si el rayo colisiona con el *bounding volume* y, en caso de no hacerlo, retorna anticipadamente. En caso de que haya una colisión se procede a iterar por los triángulos del objeto para verificar en qué punto del objeto colisionó dicho rayo.

4.3.11. Librerías matemáticas

Para realizar las operaciones matemáticas presentes en los algoritmos mencionados en las secciones previas se desarrollaron dos clases, `Vector3` y `Vector2`, y un conjunto de funciones independientes en una librería propia llamada `MathUtils`. Tanto `Vector3` como `Vector2` contienen métodos que permiten realizar operaciones aritméticas entre dos vectores o entre vectores y escalares. En particular, `Vector3` cuenta con los métodos `reflect()` y `refract()` que calculan la reflexión y la refracción de un vector respecto de otro. También se utilizó esta clase para representar los colores según sus componentes RGB, donde cada componente del color se asocia a una coordenada del vector. Por último, la librería `MathUtils` contiene métodos de operaciones comunes, como `degreesToRadians()` que convierte un ángulo de grados a radianes, o `getRandomDouble()`, que devuelve un número real pseudoaleatorio entre cero y uno.

4.3.12. Librería de manipulación de imágenes

Los objetos que tienen una imagen como textura asignada requieren de la lectura del color de los píxeles de la imagen, para lo cual se utilizó la librería *open source* de dominio público llamada *stbimage*. Esta librería cuenta con la función `stbi_load()`, que devuelve la lista de píxeles de la imagen elegida. Esta función se utiliza al cargar las texturas que se asignan a objetos de la escena, y la variable `imagePixels` de la clase `GPUImageTexture` toma su valor de retorno.

4.4. Interfaz gráfica

Siguiendo los principios de diseño de la interacción persona-computador (HCI, por sus siglas en inglés), se desarrolló una interfaz gráfica con la intención de que resulte simple e intuitiva para el usuario y que, a la vez, exponga eficientemente toda la funcionalidad de la aplicación dentro de una misma ventana. A continuación se presenta la interfaz implementada habiendo cargado y configurado una escena previamente, y se separan las distintas secciones lógicas a las que de aquí en más se hará referencia por sus nombres.

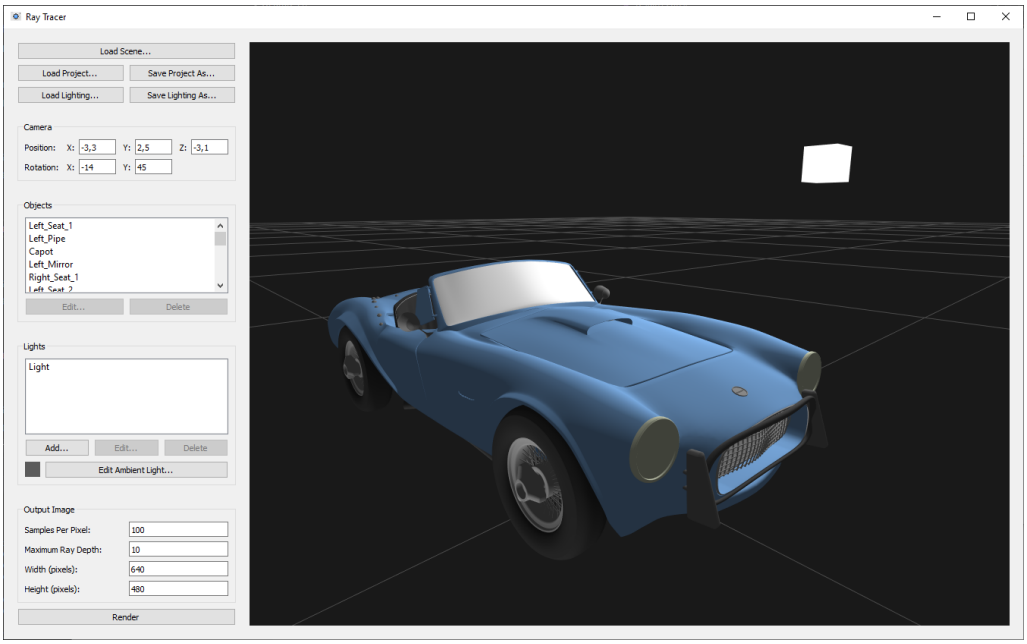


Figura 30: Vista de la interfaz gráfica con una escena cargada y configurada.

Figura 31: Vista de la interfaz gráfica dividida en sus diferentes secciones funcionales.

Para que la previsualización de la escena se destaque respecto del resto de los componentes, se optó por utilizar una mayor proporción del espacio de la ventana para este componente. Asimismo, en la barra lateral se encuentran todas las acciones disponibles agrupadas lógicamente y ordenadas según el flujo de trabajo; en particular, acciones de cargado y guardado de información, objetos de interés en una escena, propiedades lumínicas y parámetros relacionados a la renderización. En primer lugar, se presenta una serie de botones relacionados al manejo de archivos: la opción que permite cargar una escena a partir de un archivo *Wavefront*, la que permite cargar y guardar la iluminación de la escena, y finalmente la opción que permite cargar y guardar el proyecto en su conjunto. Seguidamente se muestran parámetros de configuración de la cámara, en particular, su ubicación en el espacio y su rotación (en grados). A continuación, se cuenta con una lista de los objetos presentes en la escena, desde donde es posible editar sus materiales o eliminarlos. Luego se listan en el panel de luces las geometrías con materiales lumínicos existentes junto con opciones para crear, editar y eliminar las mismas. También es posible editar la luz ambiente. Debajo se proveen opciones de configuración del motor de renderización que afectarán diversos aspectos de la imagen final. En particular, se permite al usuario determinar la cantidad de muestras por píxel a utilizar, la profundidad de rayo (cantidad máxima de veces que un rayo puede rebotar en la escena) y el tamaño de la imagen renderizada. Por último, se presenta un botón para iniciar la renderización, ubicado estratégicamente para indicar el fin del proceso de configuración por parte del usuario.

Dentro de la previsualización se muestra una grilla cuya función es orientar al usuario en el espacio tridimensional. Esto resulta de gran utilidad para complementar la experiencia de navegación mediante el uso del mouse, sobre todo en escenas vacías o de poca complejidad.

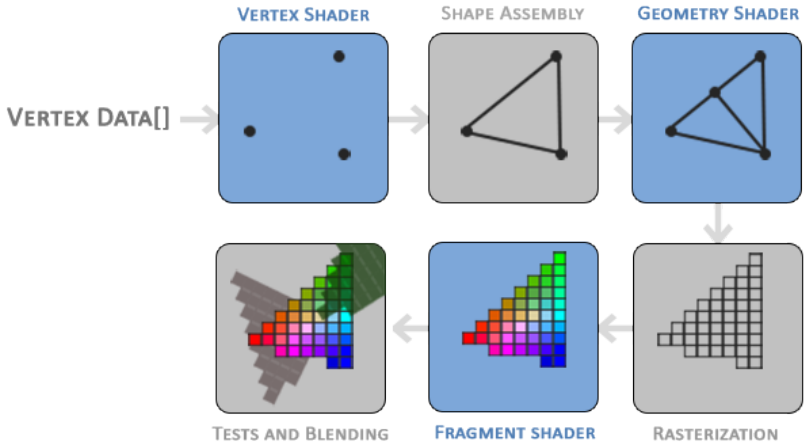
4.4.1. Qt

El *framework Qt* cuenta con clases predefinidas destinadas a representar las diversas componentes presentes en la interfaz, ya sean botones, ventanas emergentes o listas de elementos. Haciendo uso de la programación orientada a objetos, dichos elementos o *widgets* heredan de una misma clase llamada `QWidget` y cumplen distintas funciones según sea el caso. Los botones se instanciaron a partir de `QPushButton`, las etiquetas mediante `QLabel`, la ventana principal con `QMainWindow`, etc.

4.4.2. OpenGL

El *framework Qt* provee *wrappers* que ofrecen compatibilidad con la librería gráfica *OpenGL*, utilizada para implementar la previsualización de escenas. La misma es representada por una instancia de la clase `RenderWindow`, la cual hereda de `QOpenGLWidget`.

A continuación se da una breve explicación acerca del *pipeline* interno de *OpenGL*, desde el momento inicial del cargado de vértices hasta la visualización del objeto completo. Inicialmente, cada vértice leído es recibido por un *vertex shader*, el cual transforma las coordenadas tridimensionales en base a la posición y orientación de la cámara. Dicho *shader* consta de un archivo definido por el usuario (en este caso se utilizaron dos *shaders* similares, `MeshShader.vsh` y `StaticShader.vsh`, para graficar los objetos y las geometrías con materiales lumínicos respectivamente). Luego se agrupan internamente los vértices que forman cada primitiva, pudiendo tratarse de triángulos (para graficar objetos) o rectas (para graficar la grilla horizontal), entre otras. Seguidamente, *OpenGL* permite hacer uso de un *geometry shader* para definir nuevos polígonos a partir de los ya ensamblados. En esta implementación se optó por omitir esta etapa. La fase siguiente (de rasterización) traduce las coordenadas tridimensionales en píxeles de la pantalla, descartándose todos aquellos fragmentos fuera del campo de visión para lograr mayor eficiencia. Los fragmentos restantes atraviesan un *fragment shader* que calcula el color final de cada píxel (etapa de la cual se brindarán más detalles a continuación), y finalmente *OpenGL* realiza pruebas internas para validar qué objetos se encuentran frente a otros y para procesar materiales transparentes.



Fuente: learnopengl.com

Figura 32: *Pipeline* de *OpenGL* y sus etapas en el proceso de renderización.

Para determinar el color de los píxeles que corresponden a objetos en la escena, se utilizaron dos *fragment shaders* distintos. En primer lugar, `StaticShader.fsh` es usado para visualizar de color blanco las geometrías que emiten luz, y de esta forma transmitir la idea de irradiación de luz. Para los demás objetos, `MeshShader.fsh` toma en cuenta tanto el color de sus materiales (o el color blanco en el caso de

dieléctricos) como las luces para determinar el color resultante. El programa indica al *shader* la posición actual de todas las luces y sus respectivos radios de iluminación. Para determinar la intensidad lumínica de una luz aplicada sobre un objeto, se utiliza una aproximación del factor de atenuación (o *falloff*) de luz puntual, si bien las luces en el proceso de renderización fueron implementadas bajo un modelo distinto basado en materiales que emiten luz (como se ha mencionado anteriormente). La ecuación original se muestra a continuación:

$$f_{att} = \frac{1}{k_c \ + \ k_l \cdot d + k_q \cdot d^2}$$

siendo d la distancia entre un punto de la superficie del objeto y la luz. De esta forma, se obtiene una intensidad lumínica que decae a medida que un punto se aleja de una geometría cuyo material emite luz, con una fuerte caída inicial que luego se estabiliza. `MeshShader.fsh` omite el término lineal de la atenuación y asigna los factores constante y cuadrático de la siguiente manera:

$$f_{att} = \frac{1}{1 \ + \ b \cdot d^2} \qquad \text{siendo } b = \frac{1}{0{,}01 \cdot r^2}$$

siendo r el radio de iluminación de la luz. Este cálculo se realiza para todas las luces presentes en la escena. Se estableció un rango de radios de iluminación entre 30 y 100, debido a que la intensidad de la luz fuera del mismo no presentaba una variación significativa. En la siguiente figura se observa la curva de atenuación cuando el radio toma el valor mínimo y el máximo, y su efecto visual en la previsualización.

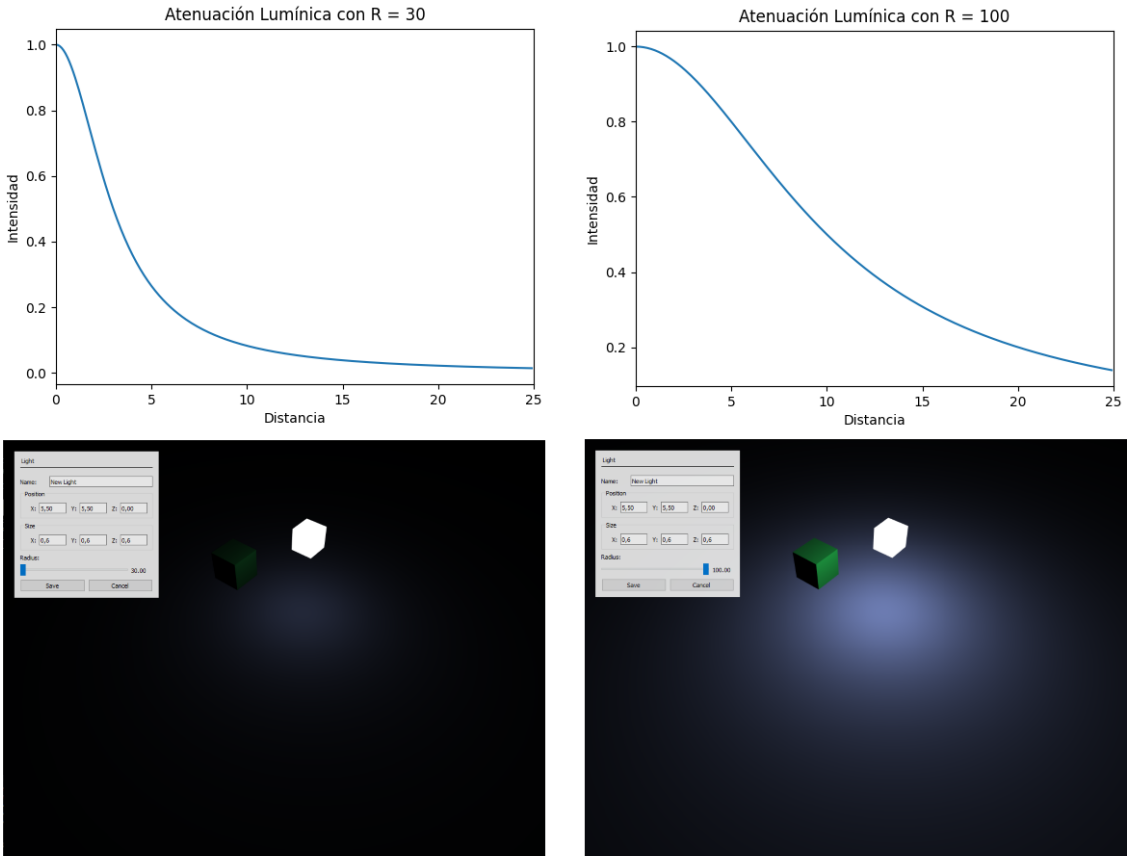


Figura 33: Atenuación de la luz implementada en los *shaders* de *OpenGL*.

La previsualización de la escena muestra resultados que se asemejan a aquellos del *ray tracer*, para que el usuario tenga noción de cómo se verá la imagen final. Sin embargo, a diferencia de lo que sucede en el *ray tracer*, existen ciertos aspectos de la escena graficada por *OpenGL* que no se corresponden con la realidad. Es en estas diferencias donde radica el valor agregado que otorga el *ray tracer*, mediante las técnicas que tienen su sustento en la física y que fueron explicadas en la sección *Estado del arte*. Estos aspectos no realistas abarcan, por ejemplo, la inexistencia de sombras en la previsualización mediante el uso de los *shaders* desarrollados. De todas maneras, existen soluciones para agregar esta funcionalidad como por ejemplo *shadow mapping*, pero fueron descartadas por su complejidad y por estar fuera del alcance del proyecto. Esta ausencia de sombras implica que los objetos no obstruyen la luz a otros objetos. Dicho comportamiento constituye una discrepancia muy significativa con la realidad. Además, como se ha mencionado anteriormente, la iluminación observable en la previsualización (excluyendo la luz ambiental) corresponde a aquella que provendría verdaderamente de luces puntuales. Esto se ha desarrollado con el fin de simplificar los cálculos en tiempo real llevados a cabo

dentro de los *shaders* implementados. Sin embargo, el *ray tracer* interpreta las luces como prismas de cierto tamaño que irradian luz a partir de toda su superficie. Esto puede ocasionar que, en la previsualización, ciertas caras de objetos no sean iluminadas en algunos casos (en especial frente a luces configuradas con grandes tamaños) y que sí sean iluminadas en la imagen renderizada.

Para que los *shaders* tengan acceso a la información de la escena, se instancian ciertas estructuras denominadas *vertex buffer object* (VBO) que pueden almacenar un gran número de vértices en la tarjeta de video. Esto implica un procesamiento de los datos extremadamente rápido y eficiente por parte de *OpenGL*, en lugar de enviar la información de cada vértice a la GPU individualmente. Cada VBO debe explicitar inicialmente cómo leer la información que contenga para que la tarjeta de video la interprete correctamente, y luego dichos datos deben copiarse en la estructura (habiendo reservado previamente el espacio en bytes necesario). Los objetos de la escena se almacenan en un VBO como conjuntos de vértices, vectores normales y coordenadas de texturas según lo observado en la figura 34. Para que los *shaders* puedan dibujar las geometrías que representan las luces y la grilla cuadricular (ambos con sus VBO respectivos), no se requieren vectores normales ni coordenadas de texturas.

VBO																							
VÉRTICE 1								VÉRTICE 2								VÉRTICE 3							
V			VN			VT		V			VN			VT		V			VN			VT	
X	Y	Z	X	Y	Z	X	Y	X	Y	Z	X	Y	Z	X	Y	X	Y	Z	X	Y	Z	X	Y

Figura 34: Estructura del *vertex buffer object* (VBO) utilizado para visualizar los objetos presentes en la escena con *OpenGL* y *shaders* propios.

Cada vez que la escena sufre cambios, como por ejemplo tras la carga de una nueva escena, la eliminación de un objeto o la adición de una luz, el VBO correspondiente vuelve a escribirse con la información actualizada para reflejar en tiempo real los cambios. La función `paintGL` dentro de la clase `RenderWindow` se encarga de interactuar con los *shaders* y redibujar constantemente la escena frente a estos cambios o también cuando el usuario se desplaza por la misma moviendo el mouse.

4.4.3. Movimiento de cámara

La navegación del usuario a través de la escena, tal vez una de las funcionalidades más elementales que provee la interfaz gráfica, involucra ciertos cálculos matemáticos que permiten determinar la posición actualizada de la cámara y su orientación. Para esto, *Qt* provee un sistema de *signals* y *slots* que permite manejar determinados eventos, como la pulsación de un botón del mouse o el maximizado de una ventana.

Mediante este sistema, se reciben las acciones realizadas con el mouse y se ejecutan el desplazamiento o la rotación del punto de vista en una cierta dirección.

Respecto a desplazamientos de la cámara, se definen dos planos en distintos universos, como se puede apreciar en la figura 35. El plano de pantalla P se corresponde, tal como su nombre lo indica, a la pantalla del usuario (en particular, a la previsualización) y es por lo tanto un espacio bidimensional. Los *clics* originados por el mouse poseen componentes en los ejes x e y que forman cierto ángulo θ con el eje horizontal, utilizando siempre el centro de la pantalla como sistema de referencia.

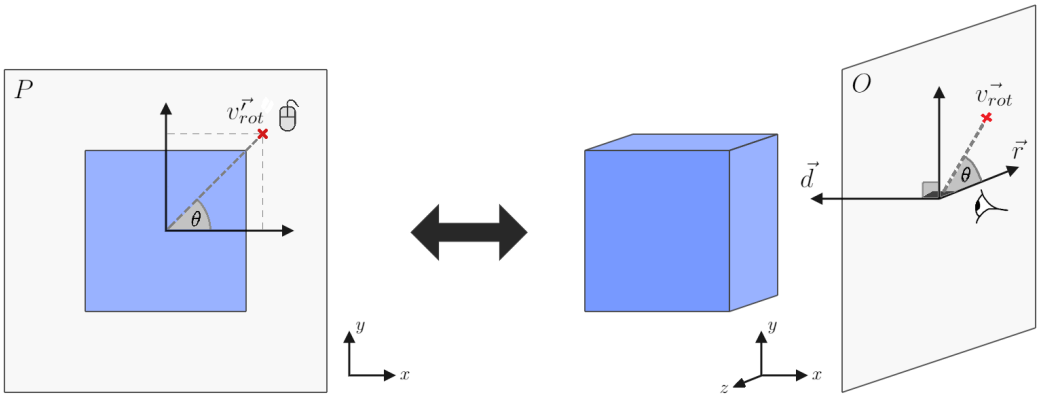


Figura 35: Planos de pantalla y de observación en el cálculo de vectores de movimiento.

Por otro lado, el plano de observación O se define a partir de la dirección normal $\vec{d}$ en la cual el usuario observa la escena tridimensional. El vector ortogonal $\vec{r}$ apunta hacia su derecha y es siempre horizontal. Para calcular la nueva posición de la cámara, resulta necesario rotar $\vec{r}$ sobre el plano O por el ángulo θ. Para lograr esto, se utiliza la fórmula de rotación de Rodrigues[16] que permite rotar un vector sobre cierto plano, dada por:

$$v_{rot}^{\rightarrow} = \vec{r}\cos\theta + (\hat{d} \times \vec{r})\sin\theta + \hat{d}\,(\hat{d}\cdot\vec{r})\,(1-\cos\theta)$$

Al ser perpendiculares los vectores $\vec{d}$ y $\vec{r}$, la ecuación se puede reducir:

$$v_{rot}^{\rightarrow} = \vec{r}\cos\theta + (\hat{d} \times \vec{r})\sin\theta$$

Luego, la posición resultante de la cámara se define como:

$$x = \vec{x_0} - v_{rot}^{\rightarrow} \cdot \Delta t \cdot s_{pos}$$

donde $\vec{x_0}$ representa la posición inicial de la cámara, Δt el tiempo entre llamadas a la función `paintGL` y s_{pos} la velocidad de movimiento prefijada por la aplicación. La resta indica que el desplazamiento está invertido con respecto al movimiento del mouse (por ejemplo, si el movimiento del mouse es hacia abajo se elevará la cámara).

En cuanto al cambio de orientación de la cámara, se hace uso de las coordenadas esféricas α y β para definir el vector de dirección $\vec{d}$, como muestra la figura 36.

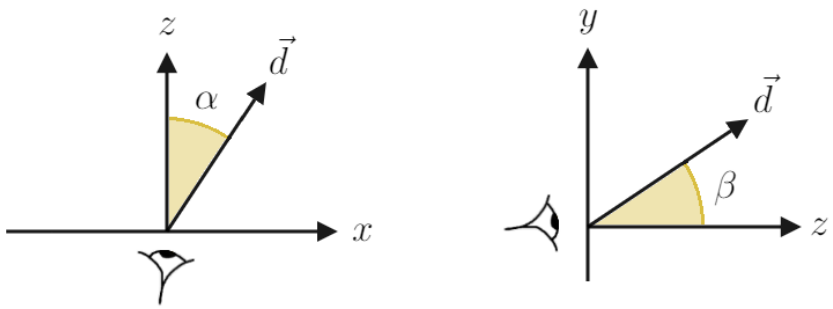


Figura 36: Definición del vector de dirección a partir de coordenadas esféricas.

Frente a un desplazamiento del puntero del mouse, manteniendo presionado el botón derecho, se determinan nuevamente las componentes dx y dy del vector $\vec{v_{rot}}$ formado a partir del centro de la pantalla de visualización, y se actualizan las coordenadas de la siguiente forma:

$$\alpha = \alpha_0 - s_{rot} \,.\, dx$$

$$\beta = \beta_0 - s_{rot} \,.\, dy$$

siendo s_{rot} la velocidad de rotación y α_0 y β_0 los ángulos iniciales. A continuación, se realiza una conversión de coordenadas esféricas a cartesianas para determinar el nuevo valor de los vectores $\vec{d}$ y $\vec{r}$:

$$\vec{d} = (\cos\alpha \cdot \sin\beta,\ \sin\alpha,\ \cos\alpha \cdot \cos\beta)$$

$$\vec{r} = (\sin(\beta - \tfrac{\pi}{2}),\ 0,\ \cos(\beta - \tfrac{\pi}{2}))$$

Frente a movimientos de cámara subsecuentes se hace uso de estos valores actualizados, al haber sido modificado el plano de observación O sobre el cual el desplazamiento ocurre.

4.4.4. Lectura de archivos *Wavefront*

Para este proyecto se implementó un lector de archivos *Wavefront* propio. Esto permitió que fuera posible realizar una conversión desde este formato al formato interno con el que se representan los distintos elementos que conforman la escena.

Como la implementación del lector fue propia y el formato *Wavefront* no es especialmente estricto, el soporte de archivos *Wavefront* es parcial. Por ejemplo, algunos archivos representan los polígonos como cuadriláteros en lugar de triángulos. Por lo tanto es posible que ciertos archivos *Wavefront* válidos no sean soportados por la aplicación.

4.4.5. Persistencia de escenas

A partir de una misma escena tridimensional pueden obtenerse infinitas imágenes considerando todos los parámetros configurables que ofrece la aplicación. Por ello, desde un comienzo se planificó incluir algún tipo de funcionalidad que permita fácilmente guardar el estado instantáneo de la escena en cualquier momento dado. De la misma se guardan específicamente las siguientes propiedades::

- Color de luz ambiente
- Geometrías con materiales lumínicos incluyendo todos sus atributos
- Objetos con sus respectivos materiales (incluyendo texturas) y todos sus atributos
- Ángulo y posición de la cámara

Teniendo en cuenta la complejidad de las escenas y la complejidad inherente a diseñar un protocolo binario se optó por un formato estándar de texto, *JSON*[17]. En lugar de implementar un sistema propietario para escribir archivos *JSON*, se decidió utilizar la librería *JSON for Modern C++* (con licencia MIT)[18]. Esta librería es fácil de incluir en proyectos, fácil de usar y es veloz.

Para convertir a *JSON* cada una de las entidades involucradas en una escena se determinó que cada clase implemente un método que permita convertir una instancia de dicha clase en un objeto de la clase `nlohmann::json`, así como un constructor que reciba un único parámetro de tipo `nlohmann::json`. A continuación se incluyen los prototipos de ambos métodos (donde `Class` refiere al nombre de alguna clase con esta funcionalidad):

```
nlohmann::json serialize() const

Class(nlohmann::json j)
```

A la hora de guardar una escena en un archivo se llama recursivamente a las distintas implementaciones de este método desde la instancia de la clase `Scene` hasta las clases complementarias, como `Vector3`, `Vector2` y los tipos primitivos.

De manera similar, para cargar una escena se parte de tres elementos iniciales: el color de la luz ambiente (cuya única propiedad es el color por lo que solo se crea una instancia de la clase `Color`), las luces (instanciadas dentro de un ciclo) y los objetos de la escena (instanciados también en un ciclo).

Como el formato *JSON* es un formato de texto, resulta complejo incorporar información binaria dentro del mismo, dado que se requiere alguna transformación de información binaria a caracteres imprimibles. Una solución implementada frecuentemente en el mundo del desarrollo web (donde el formato *JSON* está muy difundido) consiste en codificar el contenido binario en *base64* [19]. Dado que las texturas suelen ser imágenes almacenadas en formatos binarios, se decidió inicialmente guardar las texturas de los distintos objetos como arreglos de números enteros en el rango $[0, 256)$ (considerando que un byte permite expresar hasta 256 valores). Esta solución es más sencilla de implementar que la codificación en *base64*. Sin embargo, demostró ser ineficiente porque un byte de la imagen puede ser representado en texto con hasta tres bytes si el valor decimal que lo representa tiene tres dígitos, por lo que almacenar la textura de esta forma ocupa más bytes que hacerlo en forma binaria. Lo mismo sucede si se utiliza *base64*. Dado que ambas soluciones incrementan considerablemente el tamaño del archivo del proyecto, fueron descartadas y se optó por una implementación más eficiente. Mediante la librería *Miniz*[20] (de licencia MIT) se combinan los archivos de texturas utilizados en la escena con el archivo en formato *JSON* que contiene toda la información de la escena en un único archivo comprimido. Al momento de cargarlos, se extraen en una carpeta temporal en la ruta `./temp/` (creándose la carpeta en caso que no exista) y se eliminan una vez finalizado el proceso de carga.

4.4.6. Visualización de resultados

Una vez finalizado el proceso de renderización, se presenta al usuario una ventana que muestra la imagen obtenida por el *ray tracer*. Dicha ventana, una instancia de `ImageViewer` que hereda de `QMainWindow`, incluye la opción de guardar el archivo en formato PPM en el directorio de preferencia. La imagen visualizada en esta ventana corresponde a un archivo guardado en un directorio temporal dentro del directorio de trabajo de la aplicación. Al cerrarse la aplicación, este directorio es eliminado por lo cual es necesario que el usuario guarde la imagen si desea conservarla. Inicialmente, se planteó solicitarle al usuario la ruta donde se guardaría el resultado al iniciar el proceso de renderización, pero luego se desestimó debido a que esto obligaba al usuario a eliminar manualmente el archivo si el resultado no fuera el deseado.

Todo esto contribuye a una mejor experiencia de usuario ya que es frecuente probar múltiples configuraciones de escena hasta alcanzar la deseada.

5. Proceso de Desarrollo

En las secciones 4.3 y 4.4 se describieron en detalle los dos principales componentes de la aplicación desarrollada: el motor de renderización (*ray tracer*) y la interfaz gráfica. Ambos fueron desarrollados en paralelo, con una activa participación de todos los miembros del equipo. Inicialmente se comenzó a trabajar en los mismos como proyectos separados e independientes, donde uno sería una aplicación y el otro una librería. Esto se debió principalmente a que ningún miembro del equipo contaba con experiencia considerable trabajando con las librerías *Qt* ni *OpenGL*, ni con un amplio conocimiento relacionado a la computación gráfica. Esto hizo que se contemplara la posibilidad de que las tecnologías escogidas no resultaran lo suficientemente prácticas o que no fueran las adecuadas para el proyecto, por lo que fue esencial desde el comienzo revalidar continuamente las principales tecnologías elegidas. Tras aproximadamente tres meses de desarrollo se decidió comenzar con la integración funcional de ambos proyectos. Aunque se consideró compilar el motor de renderización como una librería dinámica o estática y luego vincular ambos proyectos en la etapa de linkedición del proyecto de la interfaz gráfica, finalmente se optó por combinar ambos proyectos en uno solo.

Si bien no se definió un proceso estricto a seguir durante la etapa de desarrollo, se respetaron algunas de las pautas establecidas por la metodología de desarrollo ágil *SCRUM*. Las tareas del proyecto fueron definidas a medida que fue progresando el desarrollo del mismo, es decir, no fueron predefinidas. Asimismo, con muy esporádicas excepciones, se mantuvieron una o dos reuniones virtuales por semana desde el comienzo hasta el fin del proyecto. Estas reuniones fueron, en muchas ocasiones, sesiones colaborativas de desarrollo, y sirvieron también para refinar las tareas pendientes. Algunas tareas fueron llevadas al formato *card* de *Jira* (la herramienta de seguimiento de proyectos de *Atlassian*), mientras que otras se mantuvieron en archivos de texto no sincronizados en la nube.

El sistema de control de versiones escogido fue *git* al ser posiblemente el más popular y al integrarse cómodamente con *Bitbucket*, la plataforma de *Atlassian* utilizada para alojar el código.

Como se ha mencionado en la sección *Arquitectura*, la principal IDE utilizada para realizar el desarrollo fue *Microsoft Visual Studio*. Sin embargo, se mantuvo la configuración necesaria para poder importar el proyecto en *Qt Creator*, la IDE de *Qt*, aunque no se terminó de configurar para funcionar con *CUDA*. Esto permitió, entre otras cosas, probar la aplicación en otros sistemas operativos y fue útil para encontrar errores no detectados en los entornos de desarrollo principales. También se mantuvieron dos ramas de código distintas: una que realiza la renderización utilizando un único hilo de la CPU mientras que la otra lo hace con *CUDA*.

6. Resultados

Para realizar distintas pruebas de rendimiento y de calidad de resultados se utilizaron dos configuraciones de hardware distintas. La configuración **A** (la más potente) cuenta con una tarjeta de video *NVIDIA GTX 1060* con 6 GB de VRAM y con un procesador *Intel Core i7-8700 @3.20 GHz*. La configuración **B** cuenta con una tarjeta de video *NVIDIA GTX 960* con 4 GB de VRAM y con un procesador *Intel Core i5-4690k @3.50 GHz*. Cada una de las pruebas realizadas fue ejecutada únicamente en una de las configuraciones de hardware anteriormente mencionadas. Cabe aclarar que las resoluciones de las imágenes presentadas en esta sección han sido adaptadas al formato de este documento y no necesariamente coinciden con las resoluciones con las que fueron renderizadas.

6.1. Muestras por píxel

A continuación se incluyen ejemplos que demuestran las diferencias entre distintas imágenes renderizadas a partir de la misma escena, alterando únicamente la cantidad de muestras por píxel utilizadas en el proceso de renderización.

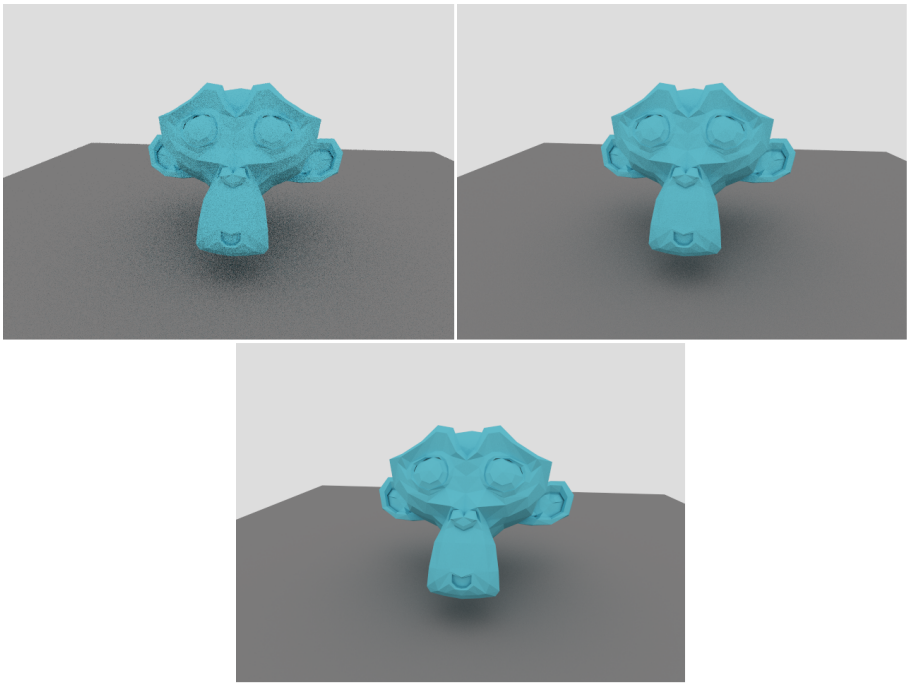


Figura 37: Imágenes renderizadas con s muestras por píxel y un valor de cinco como profundidad de rayo. Arriba a la izquierda: $s = 10$; arriba a la derecha: $s = 100$; abajo: $s = 1000$. Resolución: 640x480.

En la figura 37 se observan mejoras en la calidad de las sombras mientras más muestras se utilizan. La imagen superior izquierda fue renderizada con 10 muestras, y en ella se puede apreciar (particularmente en la sombra) un fenómeno de distorsión o ruido el cual es característico cuando se usan pocas muestras por píxel. En la imagen superior derecha, renderizada con 100 muestras por píxel, este fenómeno se observa en menor medida y puede ser necesario ampliar la imagen para notarlo. Luego, en la imagen inferior, la sombra aparece más lisa y el efecto es prácticamente imperceptible.

Además de diferencias en la calidad de la imagen resultante, el parámetro tiene un impacto en el tiempo necesario para generar la imagen. En la siguiente figura resulta evidente que aumentar la cantidad de muestras por píxel implica un mayor costo computacional:

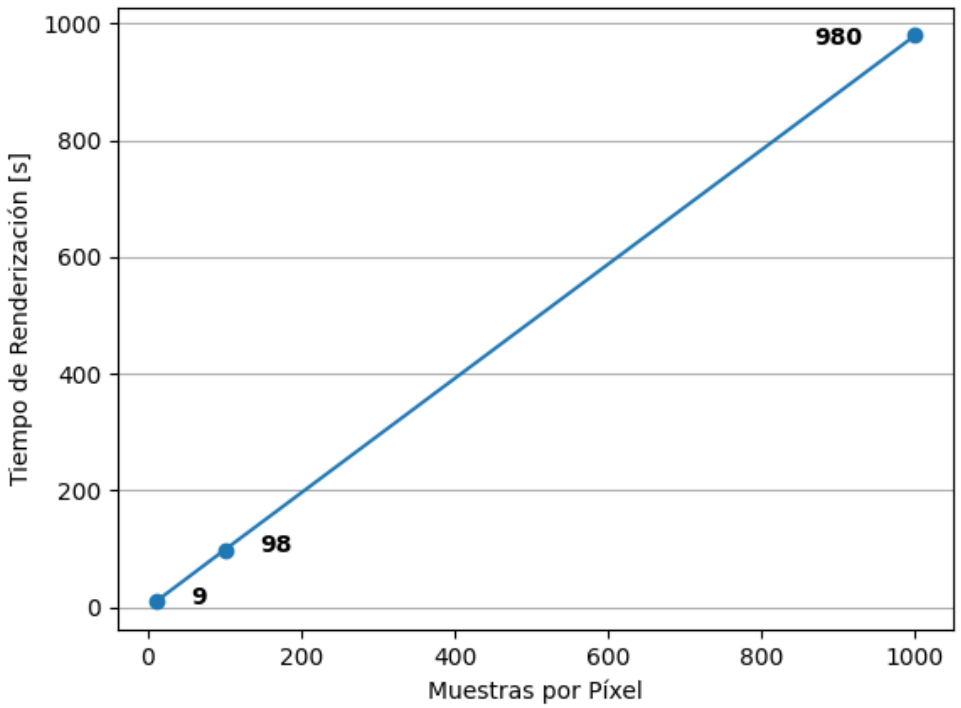


Figura 38: Tiempos de ejecución del proceso de renderización de imágenes expuestas en la figura 37 utilizando la configuración **A**.

Resulta interesante la evolución lineal del tiempo en base a la cantidad de muestras por píxel. Este patrón se repetirá a lo largo de las siguientes pruebas.

Visualmente, el impacto en la variación de la cantidad de muestras por píxel

es destacadamente notable cuando se tienen escenas oscuras con una fuente de luz pequeña, como se muestra a continuación:

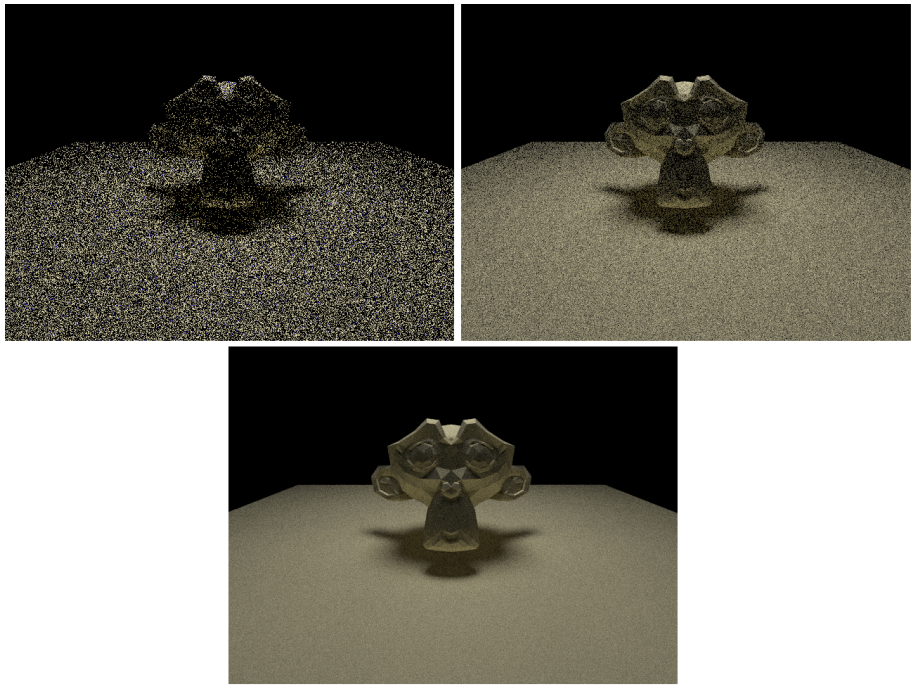


Figura 39: Imágenes renderizadas con s muestras por píxel y un valor de cinco como profundidad de rayo. Arriba a la izquierda: $s = 10$; arriba a la derecha: $s = 100$; abajo: $s = 1000$. Resolución: 640x480.

Cuanto mayor es la cantidad de muestras, mayor es la probabilidad de que un rayo colisione aleatoriamente con cualquier luz presente en la escena. En escenas oscuras, hay varios píxeles que no reciben iluminación cuando esto no ocurre, por lo que su color acaba siendo negro.

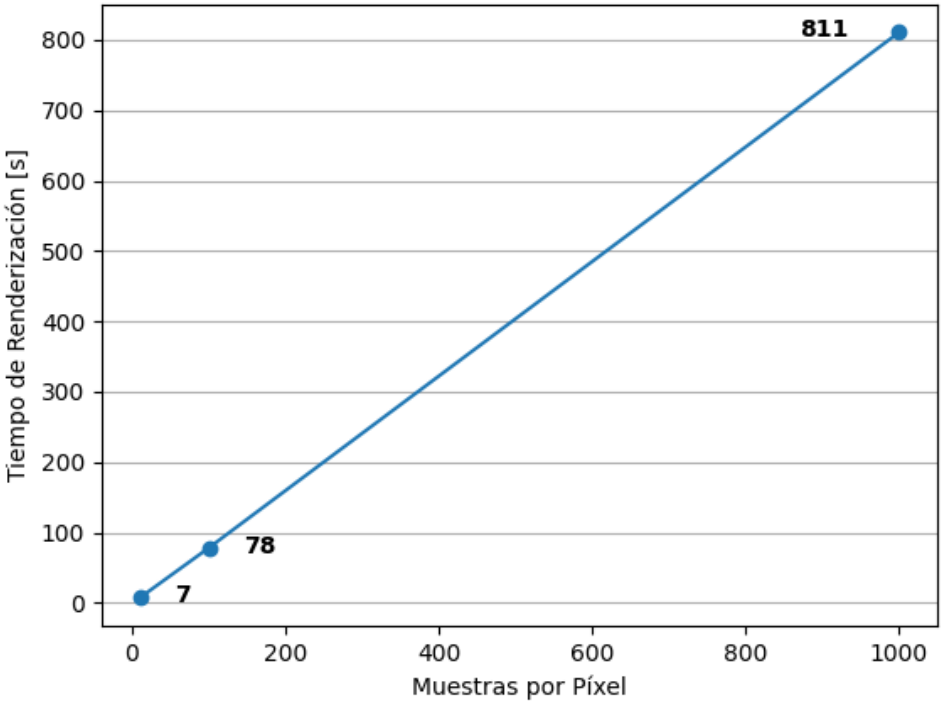


Figura 40: Tiempos de ejecución del proceso de renderización de imágenes expuestas en la figura 39 utilizando la configuración **A**.

6.2. Profundidad de rayo

Para evaluar la influencia de la cota superior de la cantidad de rebotes por rayo se utilizó una escena diseñada específicamente para este fin. La misma consiste en un espacio cerrado que cuenta con una luz ubicada detrás de una pared, la cual ilumina indirectamente los objetos que se encuentran detrás de la obstrucción.

Figura 41: Escena renderizada con 300 muestras por píxel y d como profundidad de rayo. Arriba a la izquierda: $d = 5$; arriba a la derecha: $d = 10$; abajo a la izquierda: $d = 15$; abajo a la derecha: $d = 20$. Resolución: 373x280.

La luz de la escena aporta mayor luminosidad cuanto mayor es la profundidad de rayo. Esto se debe a que los rayos necesitan rebotar múltiples veces para alcanzar a la luz debido a la obstrucción presente. Cuantos más rebotes se permitan, mayor será la probabilidad de alcanzar la luz. Existe cierta cota superior de profundidad, propia de cada escena, a partir de la cual el efecto visual no sufre alteraciones significativas.

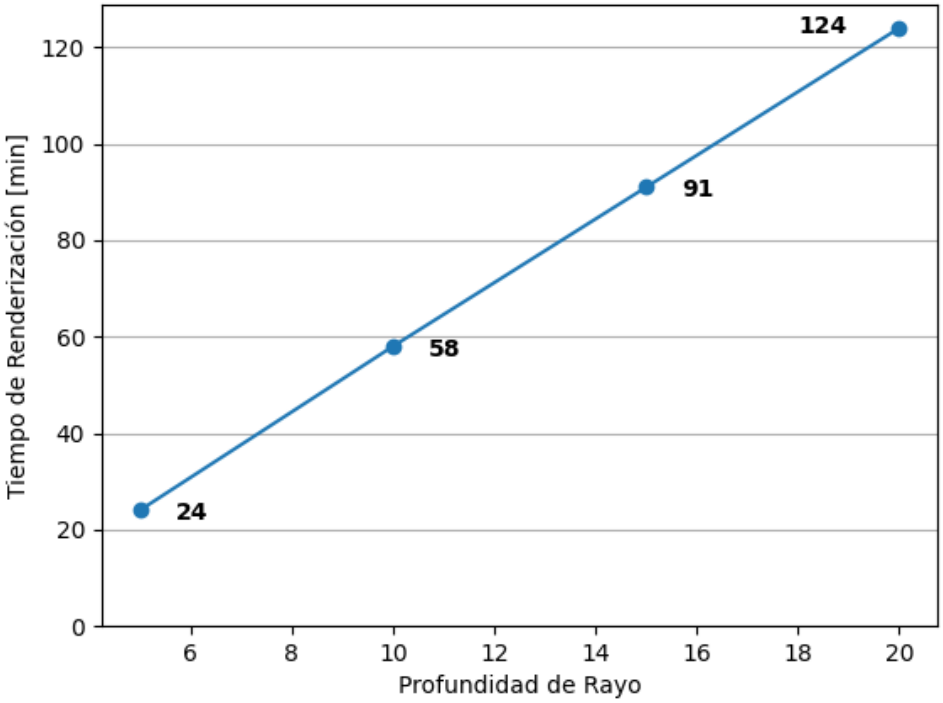


Figura 42: Tiempos de ejecución del proceso de renderización de imágenes expuestas en la figura 41 utilizando la configuración **B**.

6.3. Materiales y texturas

A continuación se presentan imágenes renderizadas a partir de una misma geometría, pero haciendo uso de los diferentes materiales implementados para demostrar la forma en la que los mismos se comportan al interactuar con su entorno.

Figura 43: Conejo y base renderizados con material lambertiano, 1000 muestras por píxel y un valor de cinco como profundidad de rayo. Resolución: 640x480. Duración de 12 minutos utilizando la configuración **A**.

Se puede observar en la figura 43 el color uniforme a lo largo de la superficie del piso y del conejo debido al comportamiento del material lambertiano.

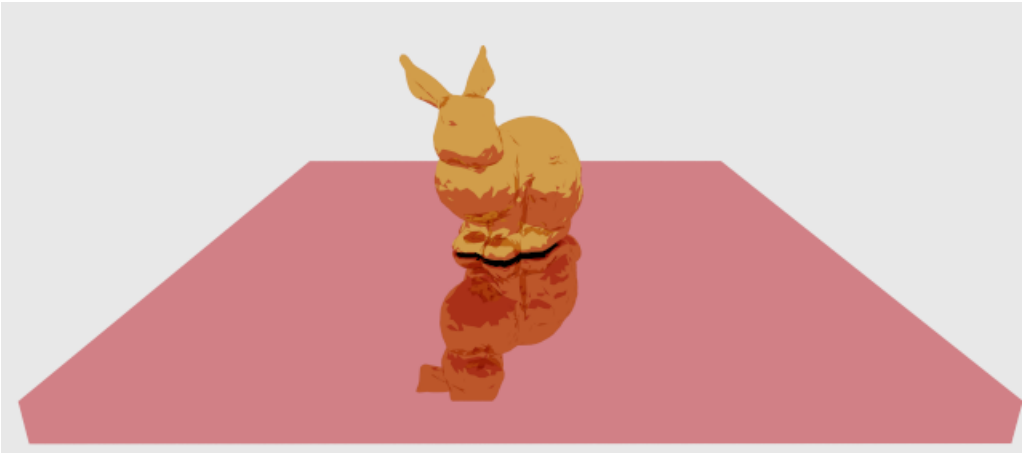


Figura 44: Conejo y base metálicos, ambos renderizados con difuminación nula, 1000 muestras por píxel y un valor de cinco como profundidad de rayo. Resolución: 640x480. Duración de 5 minutos utilizando la configuración **A**.

En la figura 44 el piso refleja perfectamente la luz, al tratarse de un metal con un valor de difuminación nulo. A continuación, se muestra una secuencia de imágenes

donde se varía este valor únicamente para la base:

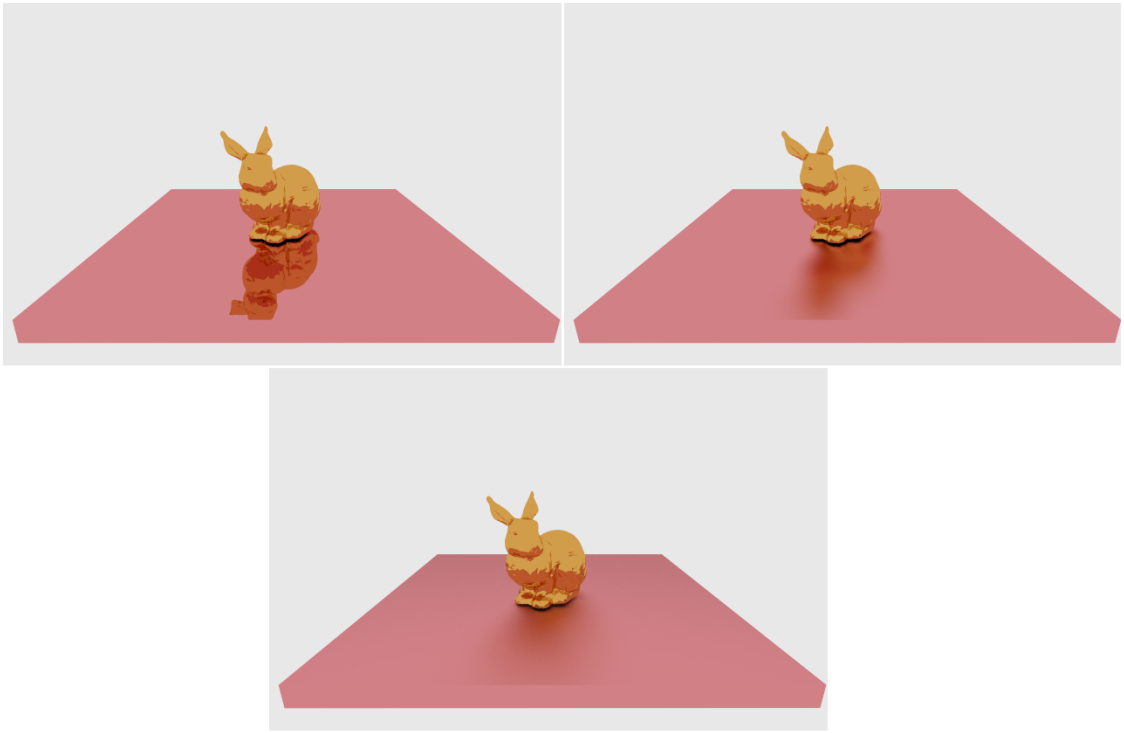


Figura 45: Conejo y base metálicos, renderizados con 1000 muestras por píxel, un valor de cinco como profundidad de rayo, difuminación del conejo nula y f como difuminación de la base. Arriba a la izquierda: $f = 0$; arriba a la derecha: $f = 0{,}2$; abajo: $f = 0{,}5$. Resolución: 640x480. Configuración: **A**.

Se puede observar en la figura 45 que la luz incidente se refleja menos a medida que se incrementa el parámetro de difuminación del metal.

En la siguiente prueba se observa la misma geometría con el material dieléctrico, utilizando el índice de refracción del vidrio crown:

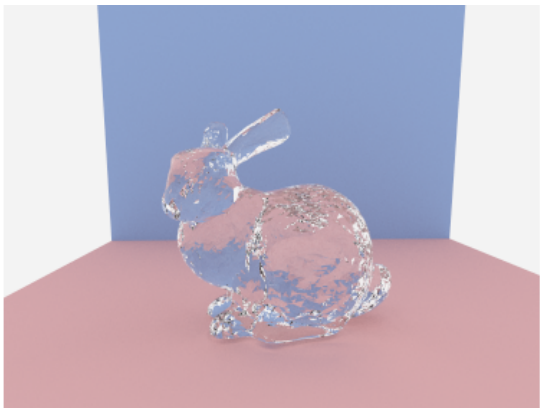


Figura 46: Conejo con material dieléctrico, renderizado con 1000 muestras por píxel, un valor de diez como profundidad de rayo y un índice de refracción de 1,517 (vidrio crown). Resolución: 373x280. Duración de 147 minutos utilizando la configuración **B**.

Como puede apreciarse en la figura 46, la luz se refracta al incidir sobre la superficie del conejo. A continuación se muestra una secuencia de imágenes cuyo objetivo es visualizar el contraste entre el índice de refracción del agua y del diamante:

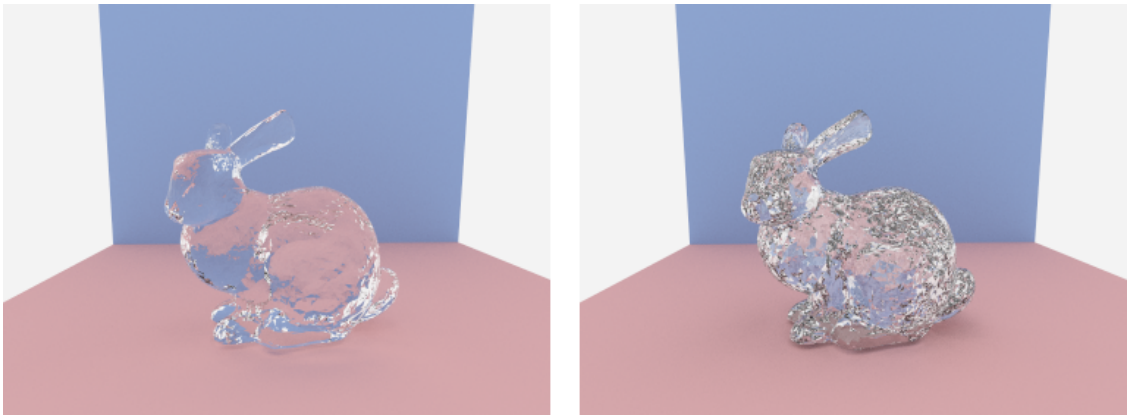


Figura 47: Conejo con material dieléctrico, renderizado con 1000 muestras por píxel, un valor de diez como profundidad de rayo y n como índice de refracción. Izquierda: $n = 1{,}333$ (agua); derecha: $n = 2{,}417$ (diamante). Resolución: 373x280. Duración de 137 y 168 minutos respectivamente utilizando la configuración **B**.

En lo que respecta al uso de texturas, se llevaron a cabo pruebas con distintas escenas y geometrías.

Figura 48: Imagen renderizada con 1000 muestras por píxel y un valor de cinco como profundidad de rayo. Resolución: 640x480. Duración de 16 minutos utilizando la configuración **A**.

Figura 49: Imagen renderizada con 2000 muestras por píxel y un valor de cinco como profundidad de rayo. Resolución: 640x480. Duración de 17 minutos utilizando la configuración **A**.

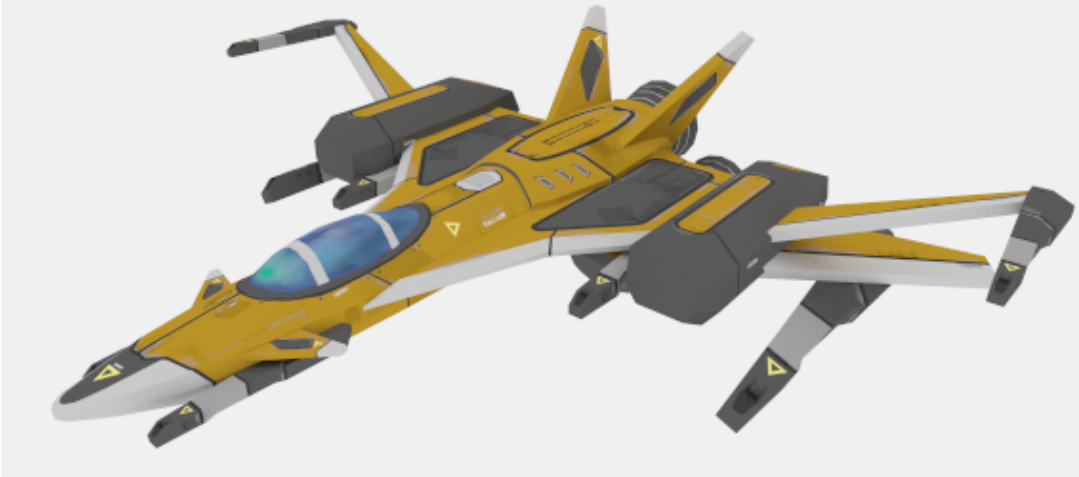


Figura 50: Imagen renderizada con 1000 muestras por píxel y un valor de cinco como profundidad de rayo. Resolución: 640x480. Duración de 5 minutos utilizando la configuración **A**.

6.4. Iluminación

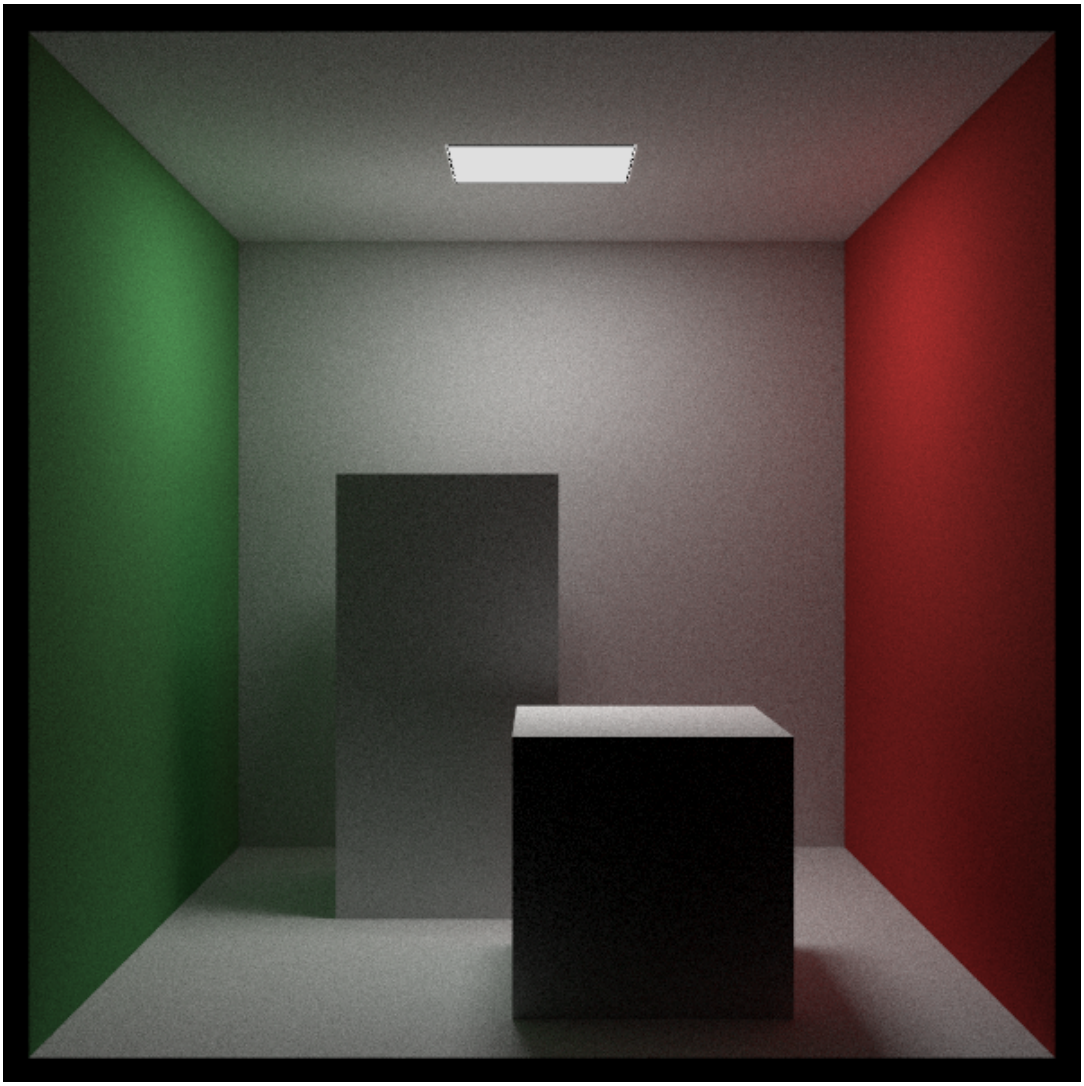


Figura 51: *Cornell Box* renderizada con 2000 muestras por píxel y un valor de diez como profunidad de rayo. Resolución: 640x480. Duración de 200 segundos utilizando la configuración **A**.

En la figura 51 se observa una escena conocida como *Cornell Box*, introducida en 1984 en el paper *Modelling the interaction of Light Between Diffuse Surfaces*[21]. Todas las superficies en la escena tienen el material lambertiano. En la esquina inferior izquierda del prisma se puede destacar un tono verde debido a la colisión de los rayos de luz emitidos con la pared izquierda. Lo mismo sucede con el cubo, donde se observa un tono levemente rojo en la base debido a la interacción de los rayos

con la pared roja. Asimismo, son destacables las sombras proyectadas de ambos volúmenes contra las paredes y el piso.

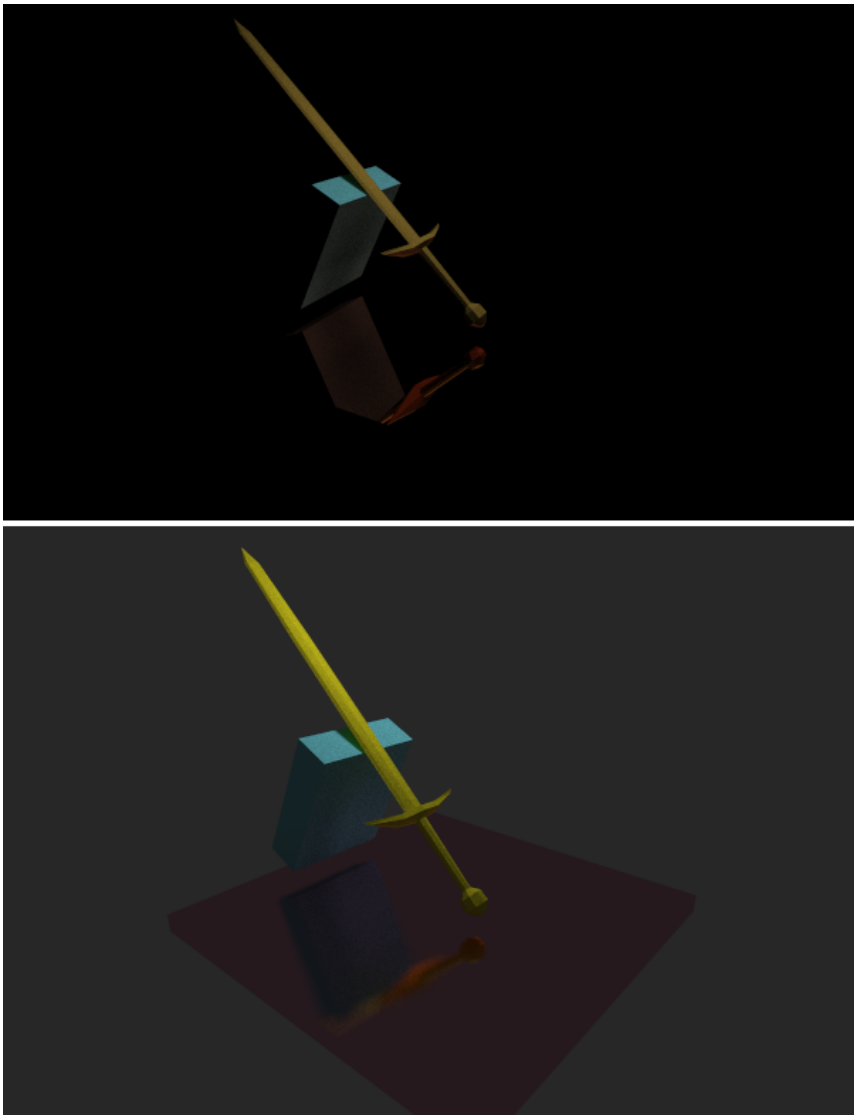


Figura 52: Imágenes renderizadas con 5000 muestras por píxel y un valor de cinco como profundidad de rayo. Resolución: 640x480. Ambas fueron generadas en 4 minutos utilizando la configuración **A**.

Como puede apreciarse en la figura 52, la imagen superior muestra una escena sin iluminación ambiente, con una luz ubicada por encima de la espada. La base posee un material metálico y su difuminación es nula, razón por la cual se refleja

perfectamente la luz sobre el mismo. En la imagen inferior se observa la misma espada en una escena con iluminación ambiente oscura y una luz idéntica a la de la imagen superior. La base también es metálica, pero con un valor de difuminación de 0,15 por lo que no se refleja perfectamente la luz sobre la misma. En ambas imágenes es destacable el comportamiento de la luz cerca de la zona de contacto entre la espada y el bloque azul donde se observa la sombra proyectada por la espada sobre la cara superior del bloque, mientras que el resto de esta cara se encuentra iluminada. Las demás caras del bloque no reciben tanta iluminación y se ven más oscuras.

6.5. Estructuras de aceleración

A continuación, se describen los resultados obtenidos al evaluar el impacto de las distintas técnicas de optimización implementadas. Como se mencionó en las secciones anteriores, se llevaron a cabo dos optimizaciones de rendimiento. Por un lado se logró ejecutar el código de forma paralela sobre la GPU, y por el otro se implementaron volúmenes envolventes como estructura de aceleración. Las pruebas realizadas a continuación reflejan los tiempos de renderización conseguidos al ejecutar en primer lugar la aplicación en la CPU sin estructuras de aceleración, y luego haciendo uso de la ejecución en paralelo sobre la GPU tanto sin estructuras de aceleración como con ellas.

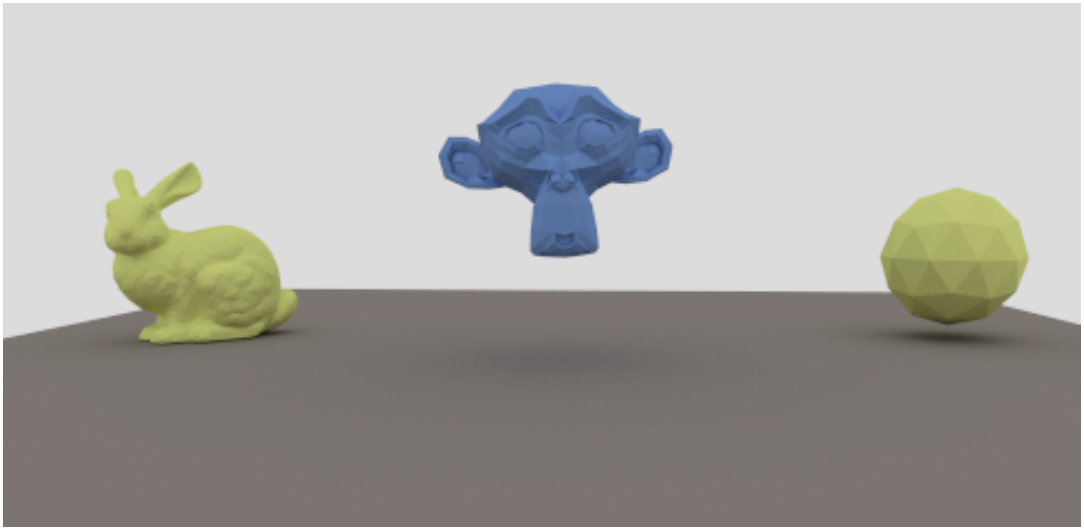


Figura 53: Imagen renderizada con 1000 muestras por píxel y un valor de cinco como profundidad de rayo. Resolución: 640x480.

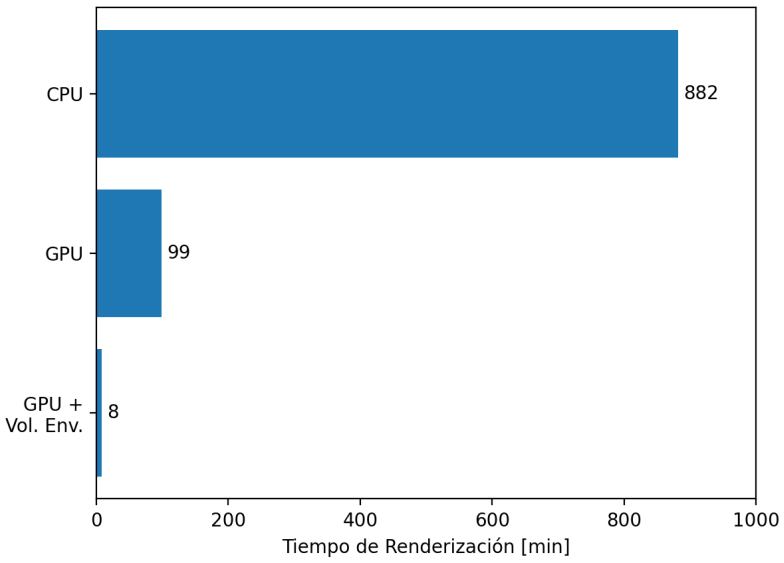


Figura 54: Tiempos de renderización obtenidos utilizando la CPU sin volúmenes envolventes y utilizando la GPU con y sin volúmenes envolventes. Configuración: **A**.

Los resultados conseguidos muestran variaciones de tiempo muy significativas entre los tres escenarios. La ejecución de la aplicación sobre la CPU demandó la mayor cantidad de tiempo, lo cual resulta lógico al no implementar técnicas de paralelización y realizar todos los cálculos de manera secuencial. Resulta notable, sin embargo, la disminución del tiempo conseguida al hacer uso de la GPU con paralelización en *CUDA*. Fue posible obtener tiempos de renderización aún menores con la incorporación de volúmenes envolventes como estructura de aceleración.

6.6. Cantidad de triángulos en escena

Se desarrollaron pruebas para medir el impacto de la complejidad de las escenas en el tiempo de renderización total. En particular, se tomó una escena y se alteró la cantidad de triángulos que la componen utilizando la herramienta *Blender*. Para cada una de las escenas generadas se mantuvieron constantes los materiales de los objetos y sus propiedades, así como también el posicionamiento y la rotación de la cámara.

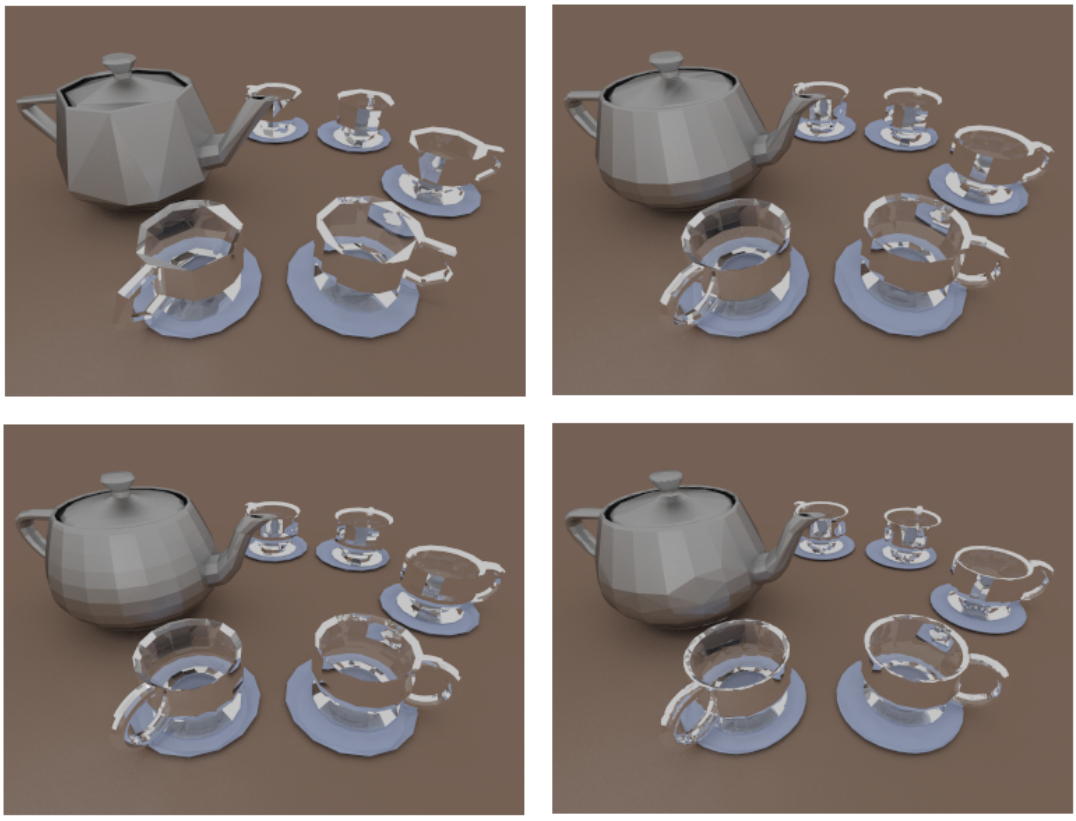


Figura 55: Representaciones de la misma escena con distintas cantidades de triángulos utilizando 2000 muestras por píxel y un valor de ocho como profundidad de rayo. Arriba a la izquierda: 1972 triángulos; arriba a la derecha: 3920 triángulos; abajo a la izquierda: 7922 triángulos; abajo a la derecha: 15821 triángulos. Resolución: 373x280.

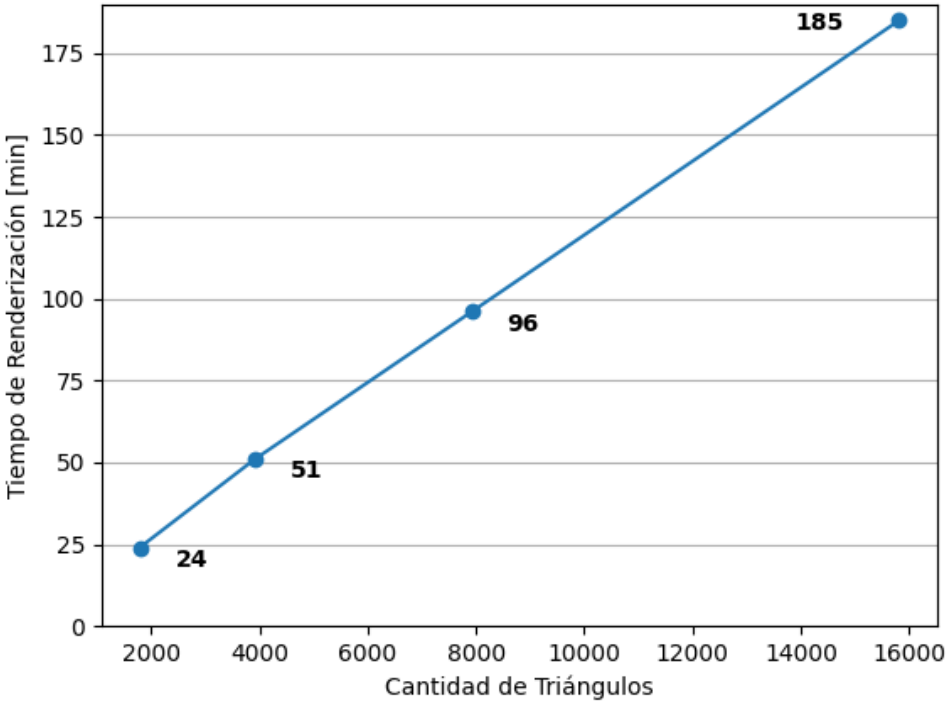


Figura 56: Tiempos de renderización obtenidos variando la cantidad de triángulos en la escena. Configuración: **B**.

Puede concluirse a partir de estos resultados que los tiempos de renderización se incrementan linealmente a medida que aumenta la cantidad de triángulos que componen una escena, principalmente debido al aumento de cálculos de colisiones entre rayos y triángulos.

6.7. Resultados integrados

A continuación se exponen imágenes renderizadas de escenas complejas que integran toda la funcionalidad de la aplicación, haciendo uso de distintos materiales y texturas así como también variando la iluminación. Para todas ellas se utilizaron valores elevados de muestras por píxel con el fin de maximizar la calidad de los resultados. Esto implicó tiempos de renderización particularmente altos.

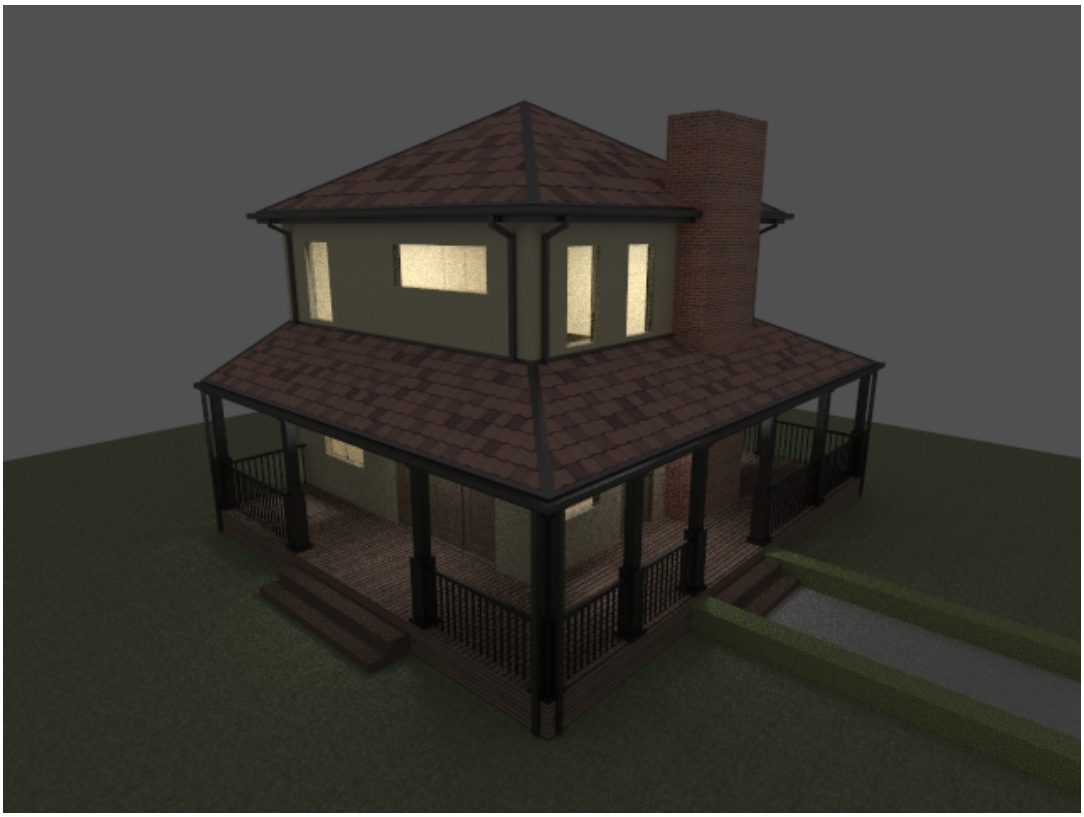


Figura 57: Escena compuesta por 5498 triángulos renderizada con 1000 muestras por píxel y un valor de ocho como profundidad de rayo. Resolución: 640x480. Duración de 115 minutos utilizando la configuración **B**.

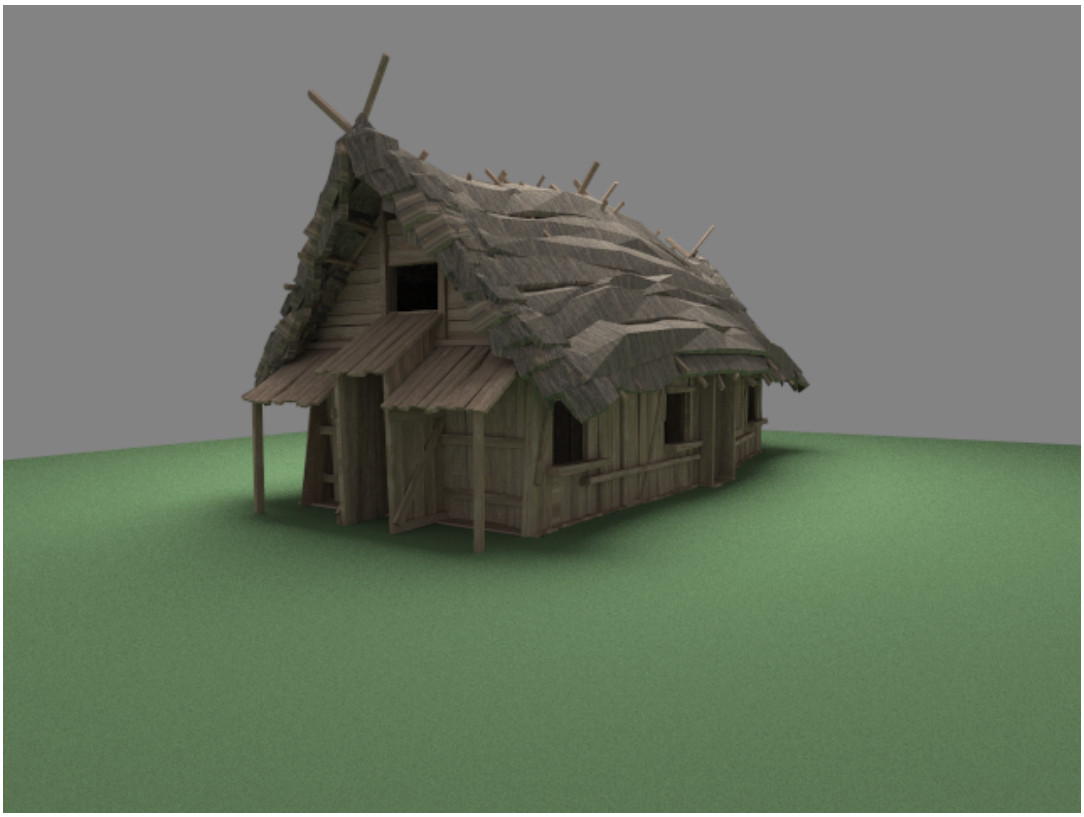


Figura 58: Escena compuesta por 7122 triángulos renderizada con 1000 muestras por píxel y un valor de ocho como profundidad de rayo. Resolución: 640x480. Duración de 4 horas utilizando la configuración **A**.

Figura 59: Escena compuesta por 1706 triángulos renderizada con 5000 muestras por píxel y un valor de cinco como profundidad de rayo. Resolución: 640x480. Duración de 1 hora utilizando la configuración **A**.

Figura 60: Escena compuesta por 3718 triángulos renderizada con 300 muestras por píxel y un valor de diez como profundidad de rayo. Resolución: 640x480. Duración de 28 minutos utilizando la configuración **B**.

7. Manual de uso

En esta sección se detallan los pasos que se deben seguir para cargar, configurar y renderizar una escena, explorando las diversas funcionalidades que la aplicación ofrece en cada etapa.

7.1. Requisitos de sistema

Para ejecutar la aplicación se requiere una tarjeta de video *NVIDIA* con al menos 2 GB de VRAM y arquitectura Maxwell o superior, es decir, compatible con *CUDA* 11. Además, debe instalarse previamente *CUDA*[22] y algún driver de video compatible con *OpenGL* 4.3 o superior. Finalmente, se recomienda utilizar el sistema operativo *Windows 10*, aunque pueda funcionar en otros sistemas operativos.

7.2. Carga de escenas y navegación

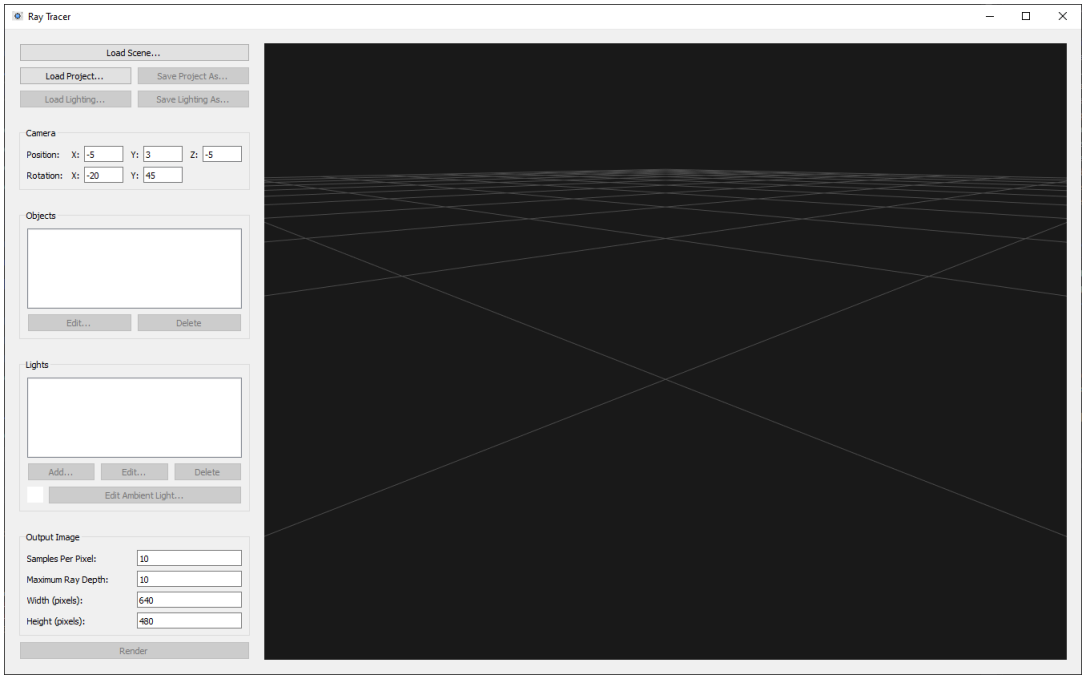


Figura 61: Estado inicial de la interfaz gráfica al ejecutar el programa.

Para cargar una escena se debe contar previamente con un archivo *Wavefront* (.obj) que describa correctamente la misma, y cuyos objetos contengan caras trianguladas. Dichos archivos pueden ser creados utilizando otras aplicaciones diseñadas para ese fin, o pueden descargarse de múltiples sitios web. Al seleccionar la opción

Load Scene, se abre un cuadro de diálogo que permite seleccionar el archivo que contiene la escena que se desea cargar.

Una vez terminado el proceso de carga, se muestra una previsualización de la escena la cual le permite al usuario explorar y navegar por la misma haciendo uso del mouse. Manteniendo presionado el botón izquierdo sobre la previsualización es posible trasladar la cámara en la dirección opuesta al movimiento realizado. Con el botón derecho, el usuario puede rotar la dirección de observación de la cámara. Por último, mediante la rueda del mouse es posible trasladar la cámara hacia delante y hacia atrás. También es posible navegar la escena utilizando los parámetros de cámara en la barra lateral. Al ingresar un valor en cualquiera de los campos de texto, la cámara se posiciona automáticamente según lo indicado. Con las teclas direccionales del teclado (flechas de arriba y abajo) se pueden aumentar o disminuir gradualmente estos parámetros, seleccionando previamente alguno de los campos. Este comportamiento se puede aplicar a todos los campos numéricos de la aplicación.

Al momento de renderizar una escena, se utilizan los valores actuales de los parámetros de cámara para generar la imagen final. Es decir, la imagen corresponderá a lo que el usuario pueda estar visualizando en la previsualización de la escena.

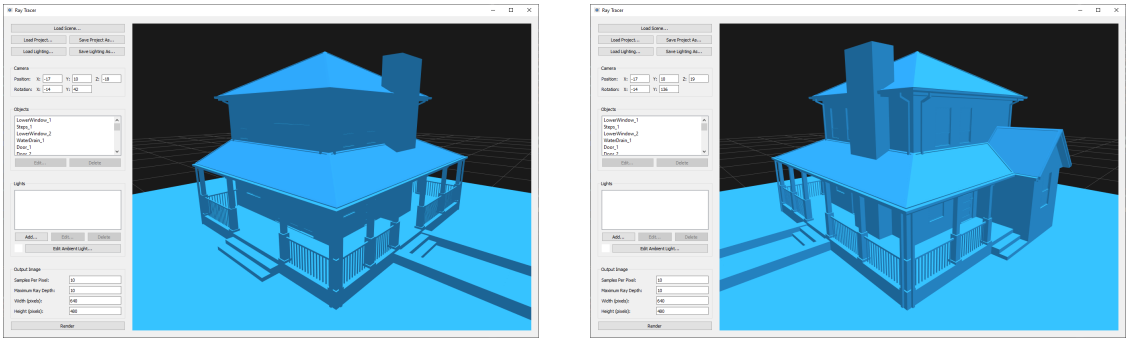


Figura 62: Visualización de una misma escena desde distintos puntos de vista.

7.3. Edición de objetos

Una escena está compuesta por diversos objetos que poseen distintos nombres y materiales. Al seleccionar algún objeto de la lista de objetos (o bien utilizando el botón *Edit*) se muestra un panel de edición que permite modificar el material del objeto y sus propiedades. Los cambios se ven reflejados en tiempo real en la previsualización. Sin embargo, hay dos propiedades que sólo tendrán efecto en las imágenes resultantes del proceso de renderización: el valor de refracción de un material dieléctrico y el difuminado de un material metálico.

Los materiales disponibles que pueden elegirse son lambertianos de color sólido, lambertianos con texturas, dieléctricos y metales, cuyas características han sido

explicadas en la sección *Materiales*. Por defecto, los materiales comienzan siendo lambertianos, con un color elegido al azar al cargar el archivo *Wavefront* de la escena.

En el caso de texturas, se presenta una opción que permite elegir un archivo de imagen con soporte para múltiples formatos (.png, .jpg, .ppm, entre otros). De no elegir un archivo, el programa asigna una textura por defecto de color blanco.

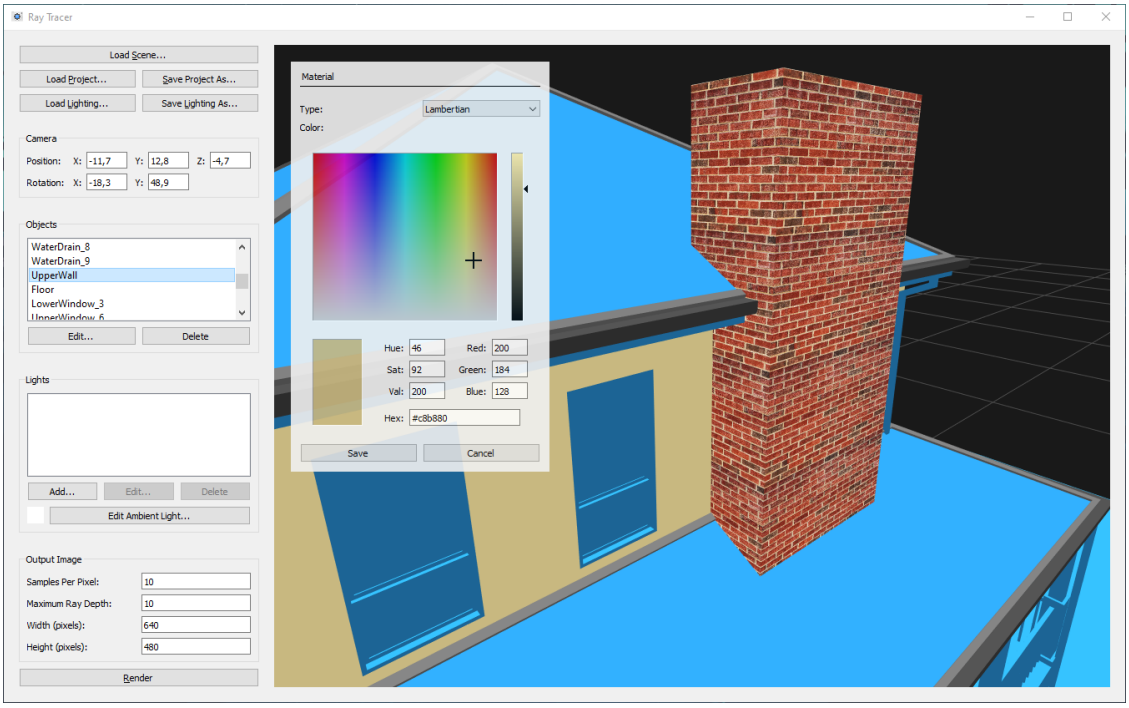


Figura 63: Edición parcial de una escena con asignación de colores y texturas.

También se encuentra disponible la opción de eliminar un objeto. Para esto se debe seleccionar el objeto en el panel de objetos y presionar el botón *Delete*. Si bien se solicita confirmación antes de proceder, se debe tener precaución ya que una vez confirmada no es posible revertir esta acción.

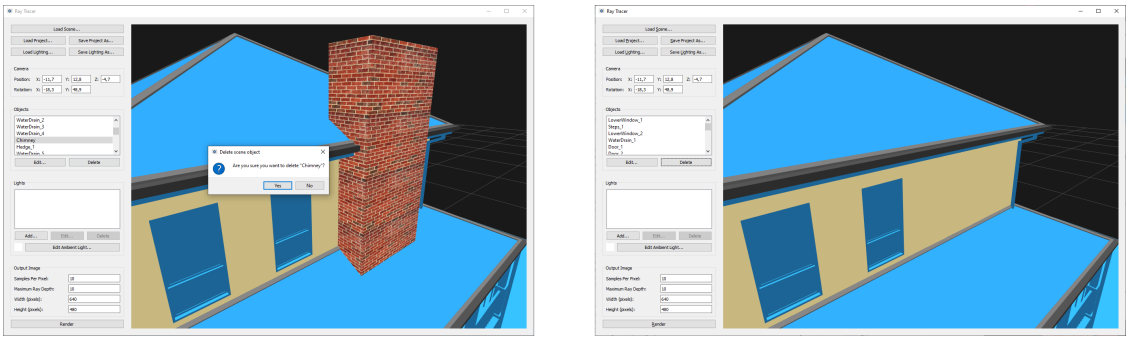


Figura 64: Eliminación de un objeto presente en la escena.

Una vez que todos los materiales han sido configurados se puede proceder a la etapa de edición de luces, si bien el usuario puede seguir los pasos en el orden que deseé. A continuación se visualiza una escena cuyos materiales han sido completamente configurados.

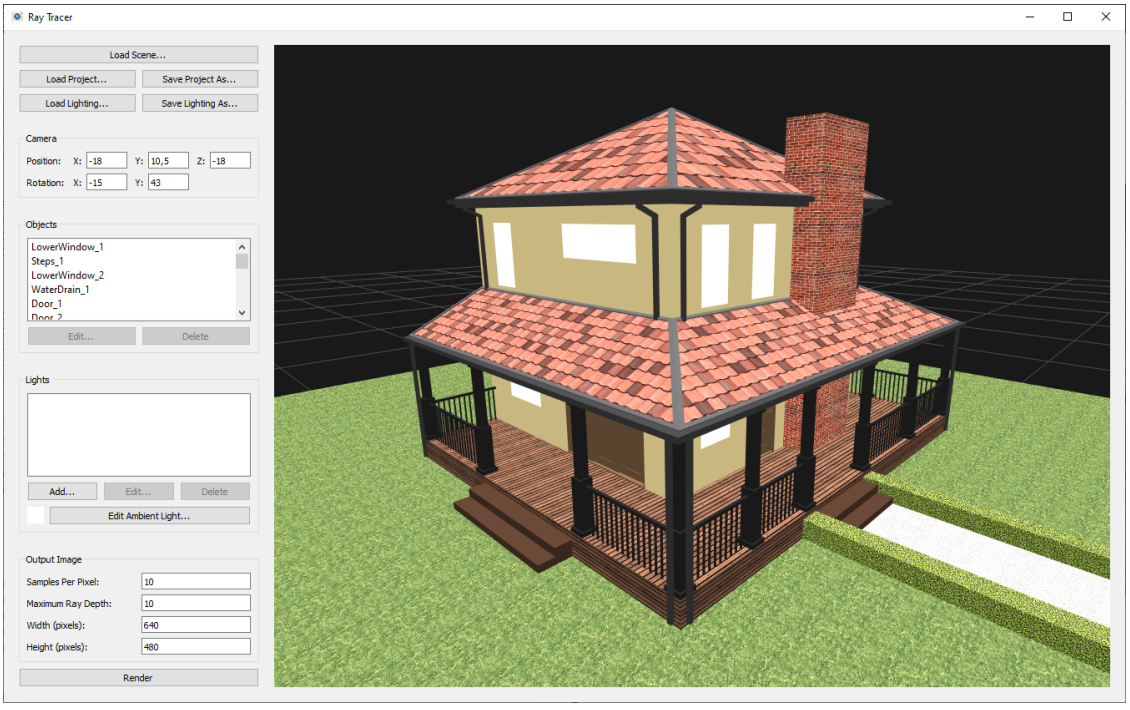


Figura 65: Visualización de una escena una vez que sus materiales han sido configurados.

7.4. Edición de luces

La luz ambiente siempre se encuentra presente en una escena, y su color puede alterarse desde el panel de luces presionando el botón *Edit Ambient Light*. Para representar escenas completamente oscuras debe seleccionarse el color negro. Si se desea que la luz ambiente no modifique significativamente el color de los objetos es recomendable seleccionar valores que se encuentren en escala de grises, como se muestra a continuación.

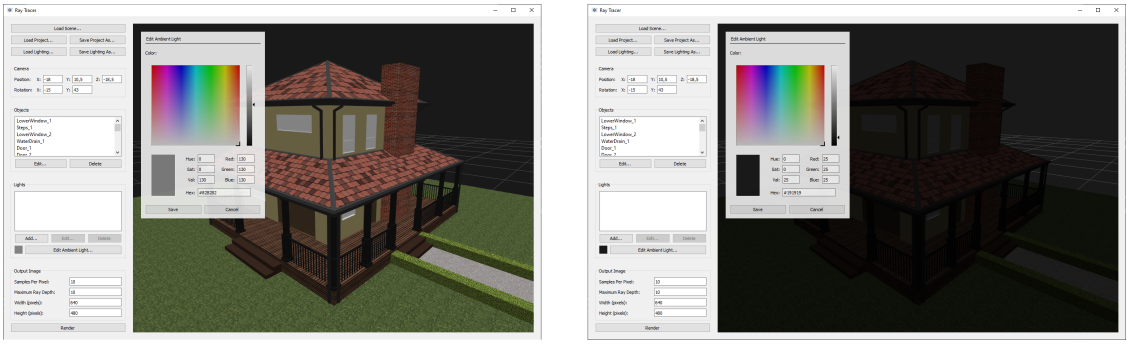


Figura 66: Visualización de una escena con distintos colores de luz ambiente.

Para agregar a la escena geometrías con materiales lumínicos se debe utilizar la opción *Add* en el panel de luces, lo cual crea instantáneamente una luz en el origen de coordenadas del espacio. Esto abre también un panel de edición desde el cual es posible modificar la posición, la escala y la intensidad de la luz, así como también su nombre. Asimismo, se pueden eliminar seleccionando el ítem correspondiente de la lista en el panel de luces y presionando la opción *Delete*.

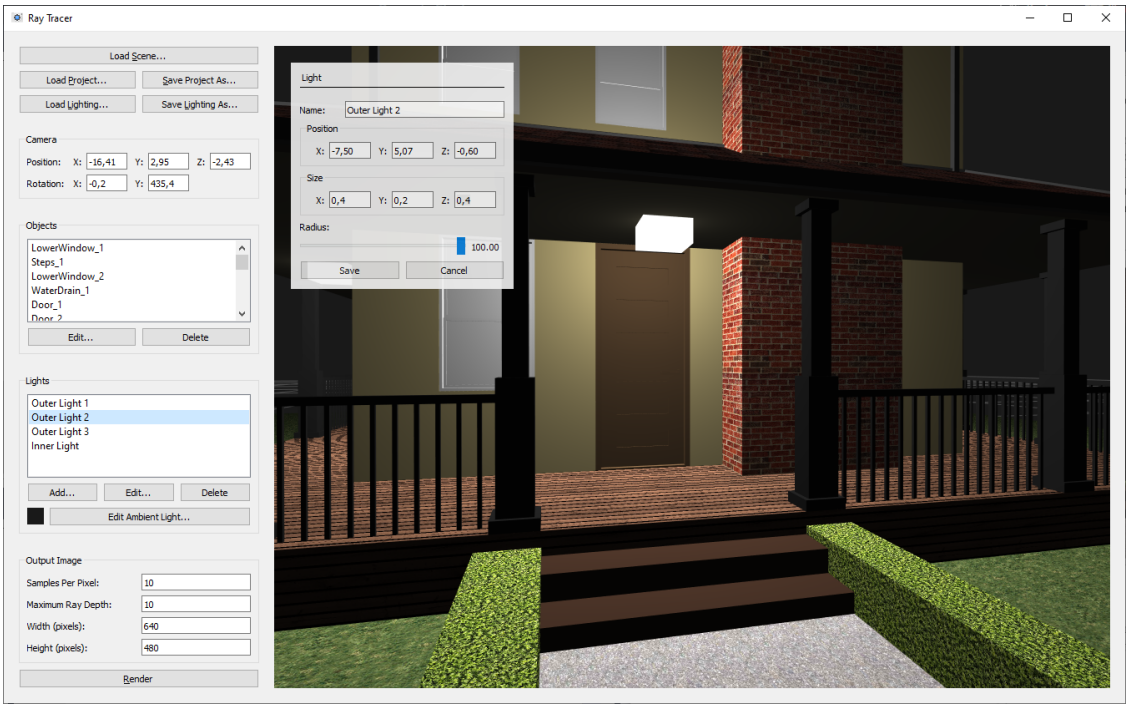


Figura 67: Edición de una geometría con material lumínico.

A continuación se visualiza una escena cuyas propiedades lumínicas han sido configuradas apropiadamente, para luego proceder al paso de renderización y generación de imágenes fotorrealistas.

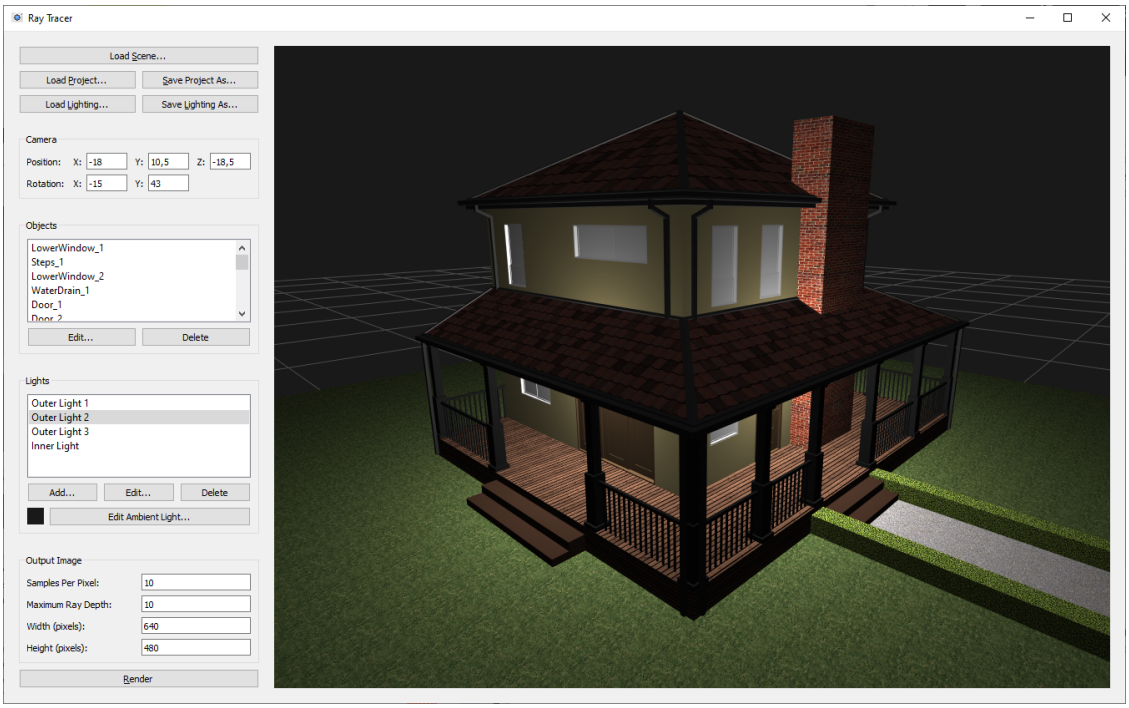


Figura 68: Visualización de una escena con múltiples luces e iluminación ambiente oscura.

7.5. Renderización de imágenes

Una vez configurados los objetos y las luces en la escena, el usuario puede renderizar distintas imágenes en base a la misma escena variando los parámetros de renderización, los cuales se encuentran en la parte inferior de la barra lateral. Es posible especificar la cantidad de muestras por píxel que el *ray tracer* utilizará para determinar el color de cada píxel. Cuanto más elevado sea, mejor será la calidad de la imagen resultante pero requerirá mayores tiempos de procesamiento. Asimismo, puede limitarse la cantidad de veces que cada rayo del *ray tracer* rebota sobre las superficies con las que colisiona. Valores cercanos a cinco (o diez cuando hay metales o dieléctricos) resultan apropiados en la mayoría de los casos. Finalmente, se permite especificar el tamaño en píxeles que tendrá la imagen de salida. Al igual que sucede con la cantidad de muestras por píxel, incrementar este valor mejorará la calidad de la imagen resultante pero requerirá mayores tiempos de procesamiento.

Luego de definir estos parámetros y de posicionar la cámara apropiadamente, el usuario puede iniciar el proceso de renderización mediante la técnica de *ray tracing* presionando el botón *Render*. Una vez finalizado el mismo se abre una nueva ventana que contiene la imagen resultante, permitiendo guardar la misma en un archivo.

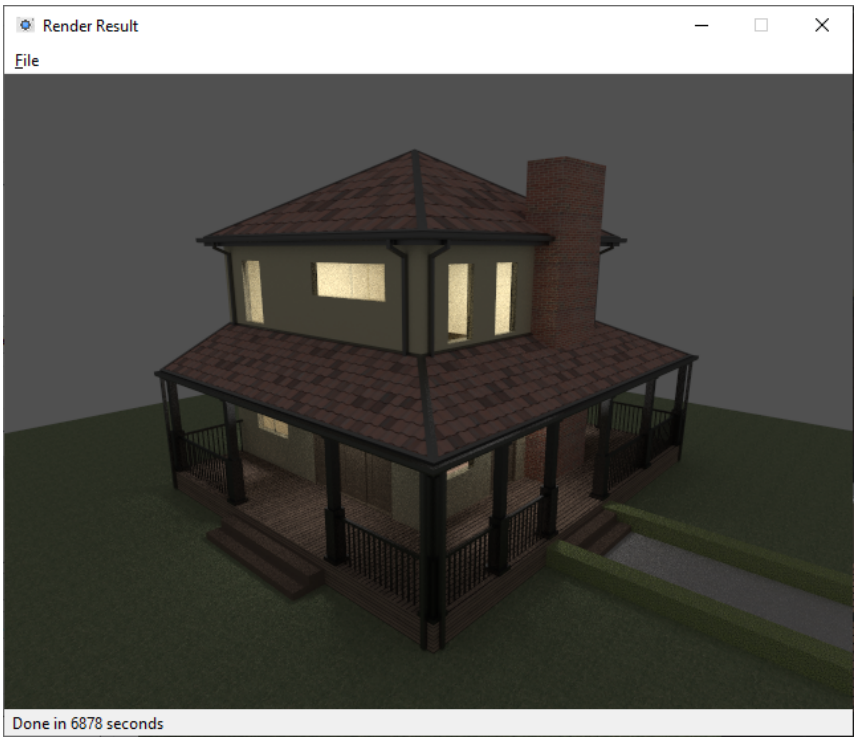


Figura 69: Ventana de visualización de imágenes renderizadas.

7.6. Guardado y cargado de proyecto

La aplicación ofrece la posibilidad de persistir escenas en archivos externos, lo cual resulta especialmente útil al trabajar con escenas de alta complejidad. Para guardar una escena se debe utilizar la opción *Save Project As* y luego especificar el archivo de destino. No es necesario conservar ningún archivo adicional.

Para cargar una escena que ha sido guardada previamente, el usuario debe pulsar el botón *Load Project* y seleccionar el archivo correspondiente. De esta forma, queda guardada una copia exacta de la escena, teniendo en cuenta todas sus propiedades. No se persisten, sin embargo, los parámetros de configuración del *ray tracer*.

Existe también la posibilidad de persistir las propiedades lumínicas de un proyecto, independientemente de la escena. Para esto debe utilizarse la opción *Save Lighting As* y seleccionar el archivo de destino. Asimismo, la opción *Load Lighting* permite cargar dichas propiedades en la escena actual. De esta forma pueden persistirse tanto las geometrías de materiales lumínicos (y sus propiedades) como el color de la luz ambiente para ser utilizados en otros proyectos.

8. Conclusiones y trabajo futuro

Los resultados obtenidos y expuestos en la sección anterior son consistentes con los objetivos y las expectativas inicialmente planteadas. La aplicación logra generar imágenes con un grado alto de realismo, logrando una mejor eficiencia haciendo uso de la GPU y estructuras de aceleración respecto al uso de CPU. El programa es interactivo, respeta los lineamientos de HCI y prácticamente no requiere conocimiento previo para poder ser utilizado correctamente, por lo que puede concluirse que cumple con el criterio de usabilidad que forma parte del alcance del proyecto. A continuación se exploran posibles mejoras que podrían implementarse en el futuro.

La teoría física detrás del *ray tracer* implementado provee una base sólida para generar imágenes de alta calidad y con iluminación realista. Sin embargo, existen algunas técnicas más recientes que, basándose en los mismos conceptos o en conceptos muy similares, obtienen aún mejores resultados. Además del algoritmo de *ray tracing* implementado en este trabajo existen otros dos algoritmos surgidos a fines de los años noventa: *Bidirectional Path Tracing* (BDPT)[23] y *Metropolis Light Transport* (MLT)[24]. Si bien ambos mantienen como base el trazado de rayos entre una fuente de luz y un objeto para generar imágenes, lo hacen de manera distinta. El primero traza rayos tanto de la fuente de luz hacia el objeto como del objeto a la fuente de luz, tratando de hallar alguna manera de conectarlos. Por otra parte, MLT parte de BDPT y genera nuevos caminos entre ambas entidades en base a los ya encontrados. En consecuencia, ambos métodos logran una iluminación aún más fiel a la realidad con un costo computacional adicional relativamente bajo. Cualquiera de estos dos métodos podrían ser explorados en profundidad partiendo de las técnicas de *ray tracing* implementadas para obtener mejores resultados.

La calidad de los resultados también puede ser mejorada, siendo que en ocasiones aún es visible cierta cantidad de ruido en los resultados finales. El ruido de la imagen produce un efecto de granularidad, y puede removerse utilizando técnicas de *denoising*. En particular, *NVIDIA* provee una herramienta llamada *NVIDIA OptiX™ AI-Accelerated Denoiser*[25], diseñada con este propósito.

Si bien la aplicación abarca todos los requisitos funcionales definidos en el alcance del proyecto, es posible incorporar nueva funcionalidad que cubra casos de uso no contemplados. En primer lugar, resultaría de interés agregar atributos adicionales a los materiales implementados (como por ejemplo el color en dieléctricos) al igual que configurar perfiles de materiales preestablecidos, como por ejemplo el cobre, el agua o el vidrio crown. Además, es posible implementar mejoras al motor de renderización respecto a la iluminación incorporando nuevos modelos como luces puntuales y direccionales, o permitiendo cambiar el color de las mismas.

Otro aspecto funcional que permitiría a los usuarios personalizar aún más las escenas consiste en añadir la posibilidad de modificar las mismas luego de ser importadas. En particular, resultaría útil mover y rotar objetos de una escena así como

alterar su tamaño. Asimismo, se podría permitir al usuario configurar atributos de la cámara como por ejemplo el campo de visión o la apertura.

Resultaría también conveniente mejorar la previsualización para poder representar con mayor afinidad las imágenes renderizadas. Para esto, los *shaders* implementados podrían soportar iluminación basada en geometrías que emiten luz así como también sombras generadas por la obstrucción de luz mediante técnicas como *shadow mapping*. Ambas mejoras requerirían una cantidad considerable de tiempo de desarrollo.

Adicionalmente se podría ampliar el soporte de archivos externos tanto de entrada como de salida. Por un lado, el lector de archivos *Wavefront* no soporta algunos archivos válidos y en ciertos casos pueden presentar fallas, como se ha mencionado en la sección *Lectura de archivos Wavefront*. Por otro lado, resultaría conveniente soportar otros formatos de imagen de salida, además de PPM. El formato PPM no cuenta con el mismo soporte que JPG o PNG, los cuales pueden ser manipulados por una infinidad de programas y dispositivos. Algunos sistemas operativos, como Windows, no proveen herramientas nativas para abrir imágenes PPM. Sin embargo, pueden abrirse y visualizarse fácilmente con software libre e incluso en ciertos sitios web.

Por último, también se podría mejorar la eficiencia del código que se ejecuta en la GPU. En *CUDA*, se denomina *warp* a bloques de 32 hilos ubicados en la GPU. Cuando se ejecuta código en paralelo, todos los hilos en un mismo *warp* se ejecutan simultáneamente si los mismos siguen un mismo flujo de control, es decir, si ejecutan las mismas instrucciones. En caso de que en un hilo el flujo de la ejecución diverja respecto de otro, la paralelización no se llevará a cabo y los hilos se ejecutarán en serie. Este fenómeno se denomina divergencia y, de no tenerse en cuenta esta problemática, no se hará un uso eficiente de la capacidad de paralelización de la GPU. Un *profiling* de la aplicación reveló que se estaba utilizando un 33 % de la capacidad de los *warps* debido a la divergencia. Por lo tanto, es posible trabajar este punto implementando técnicas de reducción de divergencia como las propuestas en *Reducing Branch Divergence in GPU Programs*[26].

Bibliografía

(1) Dürer, A., *Four Books on Measurement*, 1525.

(2) Goldstein, R. A. y Nagel, R. *3-D Visual simulation*, sagepub.com, [Online; accedido el 25 de Junio de 2021], 1971.

(3) Moore, G. E. *Cramming more components onto integrated circuits*, intel.com, [Online; accedido el 25 de Junio de 2021], 1965.

(4) Christensen, P. H. y Jarosz, W. *The Path to Path-Traced Movies*, pixar.com, [Online; accedido el 25 de Junio de 2021], 2016.

(5) Pharr, M.; Wenzel, J. y Humphreys, G. *Physically Based Rendering: From Theory To Implementation (3rd Edition)*, pbr-book.org, [Online; accedido el 25 de Junio de 2021], 2018.

(6) Library of Congress. *Sustainability of Digital Formats: Wavefront OBJ file format information*, loc.gov, [Online; accedido el 25 de Junio de 2021], 2020.

(7) Netpbm. *PPM Format Specification*, sourceforge.net, [Online; accedido el 25 de Junio de 2021], 2016.

(8) Qt Official Site, https://www.qt.io/.

(9) OpenGL Official Site, https://www.opengl.org/.

(10) Dimolarov, N. *On the state of Deep Learning outside of CUDA's walled garden*, towardsdatascience.com, [Online; accedido el 25 de Junio de 2021], 2019.

(11) Nemire, B. *NVIDIA is Now OpenCL 3.0 Conformant*, nvidia.com, [Online; accedido el 25 de Junio de 2021], 2021.

(12) Möller, T. y Trumbore, B., *Fast, Minimum Storage Ray-Triangle Intersection*, 1997.

(13) Acharya, T. y Ray, A. K., *Image Processing: Principles and Applications*, 2005.

(14) Schlick, C., *An inexpensive BRDF model for physically-based rendering*, 1994.

(15) Ley de Snell, ehu.es, [Online; accedido el 25 de Junio de 2021].

(16) Rodrigues, O. *Des lois géométriques qui régissent les déplacements d'un système solide dans l'espace, et de la variation des coordonnées provenant de ces déplacements considérés indépendamment des causes qui peuvent les produire*, Journal de mathématiques pures et appliquées 1re série, tome 5, 1840.

(17) ECMA. *ECMA-404: The JSON data interchange syntax*, ecma-international.org, [Online; accedido el 25 de Junio de 2021].

(18) Repositorio de librería *JSON for Modern C++*, github.com, [Online; accedido el 25 de Junio de 2021].

(19) IETF. *RFC 2045 - 6.8: Base64 Content-Transfer-Encoding*, ietf.org, [Online; accedido el 25 de Junio de 2021].

(20) Repositorio de librería *Miniz*, github.com, [Online; accedido el 25 de Junio de 2021].

(21) Goral, C.; Torrance, K. y Greenberg, D., *Modeling the interaction of Light Between Diffuse Surfaces*, 1984.

(22) CUDA Download Site, nvidia.com.

(23) Lafortune, E. P. y Will, Y. D. *Bidirectional Path Tracing*, kuleuven.be, [Online; accedido el 25 de Junio de 2021], 1993.

(24) Veach, E. y Guibas, L. J. *Metropolis Light Transport*, stanford.edu, [Online; accedido el 25 de Junio de 2021], 1997.

(25) *NVIDIA OptiX™ AI-Accelerated Denoiser*, nvidia.com, [Online; accedido el 25 de Junio de 2021].

(26) Han, D. y Abdelrahman, T., *Reducing Branch Divergence in GPU Programs*, 2011.

(27) Shirley, P. *Ray Tracing in One Weekend*, github.io, [Online; accedido el 25 de Junio de 2021], 2018.

(28) OpenGL Tutorial, opengl-tutorial.org, [Online; accedido el 25 de Junio de 2021].