

Implementación del algoritmo de path tracing en la GPU

Joaquim Romo

Treball Final de Grau / 2014

DIRECTOR
Javier Agenjo Departamento GTI



Resumen

El presente trabajo tiene como objetivo el estudio y la implementación de un algoritmo de renderizado con raytracing estocástico en la unidad de procesamiento de gráficos (GPU). Se ha elegido realizar la implementación en una arquitectura de este tipo debido a las ventajas que ofrece en cuanto a tiempo de ejecución, gracias a la gran capacidad de cómputo en paralelo que ofrecen las arquitecturas de GPU actuales.

La primera parte del trabajo se dedica al estudio teórico del algoritmo de pathtracing, se comentan algunos conceptos físicos básicos relacionados con el transporte de luz y su interacción con los materiales así como las ecuaciones matemáticas y las técnicas estadísticas necesarias para la comprensión y correcta implementación del algoritmo.

En una segunda parte se discuten las tecnologías involucradas, concretamente el uso que se hace de la arquitectura CUDA, la librería OptiX y su funcionamiento y la implementación del algoritmo que se ha realizado sobre estas.

Índice

Índice de figuras	IX
Índice de tablas	XI
1. INTRODUCCIÓN	1
1.1. Contexto	1
1.2. Algoritmos de iluminación global	3
1.2.1. Ray tracing	3
1.2.2. Radiosity	3
1.2.3. Path tracing	4
1.2.4. Bidirectional path tracing	4
1.2.5. Metropolis light transport	4
1.2.6. Photon mapping	5
1.2.7. Instant radiosity	5
1.3. Tecnologías involucradas	6
1.3.1. Arquitectura de las GPU modernas	6
1.3.2. Shaders	6
1.3.3. Capacidad de computo genérico	7
1.3.4. CUDA	7
1.3.5. OptiX	7
1.4. Objetivos	8
2. FUNDAMENTOS TEÓRICOS	9
2.1. Unidades Radiométricas	9
2.1.1. Flujo	9
2.1.2. Irradiancia	9
2.1.3. Ángulo sólido	10
2.1.4. Radiancia	11
2.2. BRDF	12
2.2.1. Propiedades de la BRDF	13
2.2.2. Isotropía y anisotropía de la BRDF	13

2.2.3. Modelo de Phong modificado	14
2.3. Ecuación de renderizado	15
2.4. El método de Montecarlo	16
2.4.1. Muestreo de importancia	17
2.4.2. Muestreo estratificado	17
2.5. Aplicaciones del muestreo de importancia	18
2.5.1. Muestreo de la BRDF	18
2.5.2. Muestreo del ángulo sólido subtendido	20
3. INTRODUCCIÓN A OPTIX	23
3.1. Device	24
3.1.1. Ray generation programs	24
3.1.2. Intersection programs	24
3.1.3. Bounding box programs	24
3.1.4. Closest hit programs	24
3.1.5. Any hit programs	25
3.1.6. Otros programas	25
3.2. Host	26
3.2.1. Clase Program	26
3.2.2. Clase Geometry	26
3.2.3. Clase Material	26
3.2.4. Clase GeometryInstance	26
3.2.5. Clase Context	26
3.2.6. Buffers	27
4. IMPLEMENTACIÓN	29
4.1. Interfaz de programación	30
4.1.1. Estructura TCamera	30
4.1.2. TSphereLight	30
4.1.3. TOBJModel	30
4.1.4. Clase CPathtracer	30
4.1.5. Uso de la interfaz de programación	30
4.2. Renderizado por bloques	33
4.3. Cámara	34
4.3.1. Host	34
4.3.2. Device	35
4.4. Muestreo de la BRDF	36
4.4.1. Muestreo de la parte difusa de la BRDF	36
4.4.2. Muestreo de la parte especular de la BRDF	39
4.5. Muestreo de luces	41

5. CONCLUSIONES Y RESULTADOS	45
5.1. Algunos resultados	45
5.2. Valoración de los resultados obtenidos	47
5.3. Posibles mejoras	48
5.4. Perspectivas de futuro	49
Bibliografía	51

Índice de figuras

2.1.	Definición de angulo sólido [Haade, 2007]	10
2.2.	BRDF $l = \omega_i, v = \omega_o$ [Timrb, 2008]	12
4.1.	Muestreo de la brdf difusa	38
4.2.	Arriba exponente especular de 64, abajo exponente de 512. Se puede apreciar que cuanto mayor sea el exponente más reflectante sera el material.	40
4.3.	Δ al punto de intersección	42
4.4.	Fuente de luz esférica	43
5.1.	La textura del suelo es un mapa especular	45
5.2.	Tres fuentes de luz poco visibles iluminan toda la escena	46
5.3.	Iluminación por una única fuente de luz	46

Índice de cuadros

Capítulo 1

INTRODUCCIÓN

1.1. Contexto

En el entorno de la imagen generada por computador siempre ha sido un reto tratar de generar imágenes lo más realistas posibles. Para ello un gran número de investigadores se han dedicado a diseñar algoritmos que simulan o imitan el comportamiento y la interacción de la luz con los materiales. Estos algoritmos que tratan de simular de forma realista el comportamiento de la luz son generalmente conocidos como algoritmos de iluminación global.

Estos algoritmos, por lo general, suelen tener una complejidad computacional muy elevada y el tiempo de cómputo necesario para obtener un resultado satisfactorio en escenas complejas era un factor limitador en su aplicación práctica. Por ello las aplicaciones que hacen uso de gráficos 3D en tiempo real típicamente se centran en la iluminación local o directa de los objetos de la escena y simulan la iluminación indirecta mediante técnicas que aun sin tener un fundamento físico ofrecen una mayor credibilidad para el ojo humano. Estas técnicas suelen ser algoritmos de postprocesado que se aplican en espacio de pantalla, por ejemplo *ambient occlusion* o *directional occlusion*.

Sin embargo en los últimos años se han realizado grandes avances en las arquitecturas de las unidades de procesamiento de gráficos (GPUs), en especial la gran capacidad de cómputo en paralelo debido al elevado número de microprocesadores que forman estos dispositivos. Con tal de aprovechar estos avances en el hardware los fabricantes de GPU han desarrollado librerías de computo genérico (OpenCL, CUDA), que ofrecen gran libertad al programador para implementar sus propios algoritmos.

Estas mejoras han permitido realizar implementaciones de algoritmos de iluminación global en las GPUs que son mucho más rápidos que las implementaciones típicas en la CPU, permitiendo reducir el tiempo de cómputo de varias

horas o días a minutos e incluso a tiempos interactivos dependiendo de la GPU y algoritmos utilizados.

1.2. Algoritmos de iluminación global

Se conoce como algoritmos de iluminación global aquellos que tratan de simular distintos aspectos del comportamiento de la luz en su interacción con los objetos de una escena tridimensional. Algunos de ellos están pensados y optimizados para fenómenos concretos, mientras que otros tratan de recrear fielmente todos los aspectos del transporte de luz.

En esta sección revisaremos por encima algunos de los algoritmos clásicos. Téngase en cuenta que no es el objetivo de este trabajo hacer un análisis exhaustivo de todos los algoritmos ni dar una explicación detallada de cada uno de ellos. Si el lector desea mas información sobre alguno de ellos, se han citado las fuentes originales a las que puede remitirse.

1.2.1. Ray tracing

Aunque no se trata de un algoritmo de iluminación global propiamente dicho, el algoritmo de ray tracing original, desarrollado primeramente por Appel (1968) y posteriormente ampliado en [Whitted, 1979], es relevante por la influencia que ha tenido en el campo de los gráficos generados por computador y por que ha servido de base para métodos de iluminación global desarrollados posteriormente.

En este algoritmo se lanzan rayos desde la cámara a través de los pixels de la pantalla y se comprueba si interseccionan con los objetos de la escena. En la versión de Whitted además se lanzan rayos recursivamente si el objeto es reflectante o refractante y hacia las luces, para comprobar si al objeto le llega luz directamente.

1.2.2. Radiosity

Radiosity fue el primero de los algoritmos de iluminación global que se desarrollaron. Inicialmente el algoritmo fue desarrollado en los años 1950 para aplicarlo al problema de la transferencia de calor. En 1984 fue modificado y adaptado por Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg y Bennett Battaille, investigadores de la universidad de Cornell para su aplicación en la generación de imagen sintética.

Este algoritmo trata de resolver el problema de la iluminación indirecta entre superficies puramente difusas sin tomar en cuenta la reflectancia especular.

El funcionamiento del algoritmo, en líneas generales, se basa en dividir la escena en pequeñas unidades de área, llamadas parches, que deberían funcionar como diferenciales de área. Luego a través de una serie de iteraciones se intenta balancear el flujo de luz emitido, reflejado y absorbido entre todos estos parches.

1.2.3. Path tracing

El algoritmo de path tracing [Kajiya, 1986] sea posiblemente el primer algoritmo capaz de solucionar completamente la ecuación de renderizado.

Este algoritmo empieza como el ray tracing clásico, lanzando rayos desde la cámara hacia la escena, pero cuando un rayo intersecciona con un objeto se lanza un rayo en una dirección aleatoria para tener una estimación de cuanta luz indirecta llega a ese punto. Este rayo aleatorio, a su vez es evaluado recursivamente siguiendo el mismo procedimiento.

Evidentemente este proceso de trazar un rayo desde la cámara y hacerlo rebotar por la escena para obtener un estimación de la luz es muy impreciso, por lo que es necesario repetir el proceso varias veces, y hacer la media entre los resultados obtenidos para obtener una solución satisfactoria.

1.2.4. Bidirectional path tracing

El algoritmo de Bidirectional path tracing [Lafortune and Willem, 1993] fue desarrollado como una extensión al algoritmo de path tracing de Kajiya. En esta modalidad los rayos primarios no solo se lanzan desde la cámara, sino también desde las fuentes de luz. Estos caminos de luz, se calculan del mismo modo que los de la cámara. Se guardan los puntos de intersección de los caminos de la cámara y los de la luz y en una ultima fase se unen estos dos grupos de puntos para obtener la evaluación final del camino.

La principal mejora de este algoritmo respecto a su antecesor, es que es capaz de funcionar mejor y converger mas rápido hacia una solución correcta en escenas complejas, en las que las fuentes de luz no son fácilmente visibles desde la mayoría de puntos de la escena.

1.2.5. Metropolis light transport

Siguiendo en la linea de los dos algoritmos anteriores, otra notable mejora llego con el llamado Metropolis light transport [Veach, 1997]. Este algoritmo parte de la base del path tracing bidireccional, pero en vez de confiar en crear muchos paths hasta converger a una solución aceptable, utiliza un método conocido como algoritmo de Metropolis-Hastings para generar varias mutaciones del mismo path.

Este algoritmo desata todo su potencial cuando se trata de renderizar interacciones complejas entre materiales, que normalmente serian muy costosas de renderizar con los algoritmos que hemos comentado anteriormente. Por ejemplo cáusticas, interreflexiones especulares-difusas, etc.

1.2.6. Photon mapping

Todos los algoritmos que estamos viendo tratan la luz como partículas y no como ondas, pero este algoritmo lo hace de un modo aun mas explicito. El algoritmo de Photon mapping [Jensen, 1996] empieza lanzando rayos (fotones) desde las fuentes de luz. Cuando estos fotones interseccionan con un objeto de la escena se decide aleatoriamente y según las propiedades (BRDF) del material si el fotón sera absorbido, dispersado especularmente o dispersado difusamente. Las posiciones finales donde los fotones son absorbidos se guardan en un mapa (kd-tree) de fotones para la siguiente fase.

La siguiente fase, llamada final gathering, realiza un ray tracing de la escena y en cada intersección consulta el mapa de fotones para ver la cantidad de luz que llega a ese punto.

Este algoritmo sobresale entre todos los demás cuando se trata de renderizar cáusticas, pero en cambio, puede producir errores cuando se renderizan superficies difusas si el numero de fotones no es muy grande.

1.2.7. Instant radiosity

Este algoritmo, desarrollado por Keller en 1997 , combina las ideas de los algoritmos de radiosity y photon mapping. Igual que el radiosity original, este algoritmo, en principio, solo funciona para superficies puramente difusas.

La idea general consiste en lanzar fotones desde las fuentes de luz (como en photon mapping) e ir guardando sus posiciones. La diferencia principal radica en que en la fase de renderizado, estos fotones son tratados como luces puntuales (VPLs, del inglés Virtual Point Lights) con orientación, es decir que tienen un vector normal. La ventaja es que una vez generados estos VPL es posible renderizar la escena mediante una API gráfica tradicional acelerada por hardware, como OpenGL o DirectX, o con ray tracing.

Además del algoritmo de instant radiosity básico existen modificaciones del mismo que combinan las ideas de los algoritmos de path tracing con el de instant radiosity. En este conjunto de algoritmos encontramos el instant radiosity bidireccional y el metropolis instant radiosity.

1.3. Tecnologías involucradas

El propósito de esta sección es explicar las principales tecnologías utilizadas durante el desarrollo de este proyecto. Debido al alcance de este trabajo la mayoría de tecnologías que comentaremos en este apartado son tecnologías específicas de las GPU.

1.3.1. Arquitectura de las GPU modernas

Las arquitecturas de GPU modernas siguen una paradigma conocido como SIMD (del inglés, Single Instruction Multiple Data), que consiste en la capacidad de un procesador de ejecutar la misma instrucción en paralelo sobre datos distintos. Es decir, que se ejecuta el mismo proceso en varias unidades de computo, pero los datos que trata cada unidad pueden ser distintos.

1.3.2. Shaders

Los fabricantes de GPUs empezaron a explotar esta capacidad de computo en paralelo, ofreciendo a los programadores de gráficos la posibilidad de programar ciertos puntos del proceso de renderizado efectuado por las GPU, con los llamados shaders. Los shaders son pequeños programas que se ejecutan en la GPU y sirven para programar funcionalidades. Tradicionalmente existían dos tipos de shaders: los vertex shaders, que se ejecutaban para cada vértice de cada primitiva a pintar, y los pixel o fragment shaders que se ejecutaban para cada pixel rasterizado.

Según el paradigma SIMD, solo puede ejecutarse un único shader en cada GPU pero los datos pueden ser distintos: en el caso de los vertex shaders, por ejemplo, cada unidad de procesamiento tendrá acceso a las coordenadas geométricas de un vértice, las coordenadas de textura de ese vértice, etc. Es decir que un vertex shader ejecutara exactamente las mismas operaciones para cada vértice en paralelo.

1.3.3. Capacidad de computo genérico

La principal limitación de los shaders es que solo trabajan con datos relacionados con el renderizado de gráficos (coordenadas de vértices, coordenadas de textura, vectores normales, texturas, etc). Si un programador quería utilizar la capacidad de computo en paralelo de las GPU para problemas distintos al renderizado por rasterizado de gráficos, debía buscar la manera de codificar los datos del problema en forma de vértices y texturas, lo cual podía resultar tedioso o complicado.

Debido a esta necesidad los fabricantes de GPU empezaron a buscar una forma de ampliar las capacidades de computo que ofrecían sus dispositivos y desarrollaron plataformas de programación genérica en GPUs. La primera de estas plataformas fue la desarrollada por Nvidia, con el nombre CUDA, para sus tarjetas gráficas. Posteriormente se desarrollo OpenCL, un estándar de computo en paralelo tanto para GPUs, como para CPUs, soportado por la mayoría de fabricantes de hardware.

1.3.4. CUDA

CUDA es la plataforma de computo genérico sobre GPU desarrollada por Nvidia para GPUs de Nvidia. El compilador nvcc (Nvidia C Compiler) permite tanto compilar programas con una parte en el host (CPU) y otra parte en el device (la GPU), como compilar kernels, que se ejecutaran en la GPU.

1.3.5. OptiX

OptiX es una librería de ray tracing sobre CUDA desarrollada por Nvidia. Optix esta diseñado con la idea de ofrecer un framework para ray tracing lo más genérico y programable posible, tratando de no limitar las posibilidades del programador.

1.4. Objetivos

El principal objetivo de este trabajo es realizar una implementación de un algoritmo de iluminación global, lo más completo posible, usando tecnologías de GPU para aprovechar las ventajas que ofrecen estos dispositivos en cuanto a tiempo de ejecución.

Para ello, un primer objetivo necesario era estudiar los distintos algoritmos y decidir cual se adaptaba mejor al objetivo de este trabajo, tomando en cuenta la completitud del mismo, es decir, cuan realistas son los resultados ofrecidos por el algoritmo. Otro factor importante en la decisión del algoritmo a utilizar es que sea paralelizable, factor indispensable para poder realizar una implementación eficiente en la GPU.

Una vez decidido el algoritmo, otro objetivo parcial necesario para alcanzar el objetivo primario es realizar un estudio teórico detallado del algoritmo en cuestión, y de las técnicas y conceptos relacionados con el mismo, con tal de tener una comprensión de este lo bastante profunda como para realizar una implementación seria.

Finalmente sera necesario plasmar los conocimientos teóricos adquiridos en una implementación de un renderizador con iluminación global.

Capítulo 2

FUNDAMENTOS TEÓRICOS

2.1. Unidades Radiométricas

Se conoce como radiometría al estudio de las radiaciones electromagnéticas. Ya que la luz visible es una onda electromagnética, los algoritmos de renderizado que buscan el realismo se fundamentan sobre conceptos radiométricos. Por ello, en esta sección, haremos una pequeña introducción sobre algunos conceptos básicos que nos permitirán entender mejor los algoritmos de iluminación global.

2.1.1. Flujo

El flujo radiométrico mide la cantidad de energía radiante por unidad de tiempo. Sus unidades son Watts o Joules/segundo.

$$\Phi = \frac{dQ(t)}{dt} \quad (2.1)$$

2.1.2. Irradiancia

La irradiancia representa el flujo incidente en una superficie y se mide como el flujo radiante por unidad de área y sus unidades son de W/m^2

$$E = \frac{d\Phi}{dA} \quad (2.2)$$

2.1.3. Ángulo sólido

El angulo sólido no es una unidad radiométrica en si mismo, pero es un concepto geométrico necesario para poder explicar otros conceptos radiométricos, además de otros apartados del presente trabajo.

Podemos entender el concepto de angulo sólido como la extension del angulo a las tres dimensiones.

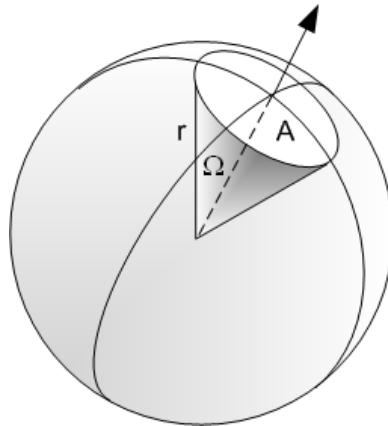


Figura 2.1: Definición de angulo sólido [Haade, 2007]

El angulo sólido se mide como el área proyectada sobre una esfera de radio unitario. Sus unidades son adimensionales y son llamadas estereoradianes [*sr*].

$$\Omega = \frac{A}{r^2} \quad (2.3)$$

Usando coordenadas esféricas $\Theta = (\phi, \theta)$ podemos definir el angulo sólido diferencial como:

$$d\omega_\Theta = \sin \theta d\theta d\phi \quad (2.4)$$

Informalmente resulta sencillo entender el angulo sólido si pensamos en *cuan grande se ve un objeto*. Supongamos una superficie perpendicular a la dirección de visión del observador: si este objeto esta muy cerca, diremos que subtienede un angulo sólido mayor que la misma superficie a una mayor distancia en la misma dirección.

2.1.4. Radiancia

La radiancia, también llamada intensidad por algunos autores, es probablemente la unidad radiométrica más importante en lo que concierne al presente trabajo, y a muchos de los algoritmos de iluminación global, ya que su valor es invariante a lo largo de la longitud de un rayo.

Esta unidad mide la irradiancia por unidad de angulo sólido.

$$L = \frac{dE}{d\omega} = \frac{d^2\Phi}{d\omega dA \cos \theta} \quad (2.5)$$

2.2. BRDF

La función de distribución de reflectancia bidireccional (de ahora en adelante BRDF, por sus siglas en inglés), definida por primera vez por [Nicodemus, 1965] Nicodemus (1965), es un función que define la respuesta a la luz de una superficie opaca, tomando como parámetros dos vectores unitarios que definen las direcciones de entrada y salida de la luz. Más formalmente, la BRDF mide la relación entre la radiancia diferencial reflejada en la dirección de salida, y la irradiancia diferencial entrante en el ángulo sólido diferencial alrededor del vector de entrada.

$$f(x, l, v) = \frac{dL(x \rightarrow v)}{dE(x \leftarrow l)} = \frac{dL(x \rightarrow v)}{L(x \leftarrow l) \cos \theta d\omega_i} \quad (2.6)$$

Donde l es el vector unitario que apunta en la dirección opuesta a la de entrada de la luz, y v es el vector unitario que apunta en la dirección de salida de la luz.

La BRDF solo esta definida para vectores l y v tales que $n \cdot v > 0, n \cdot l > 0$, siendo n la normal de la superficie.

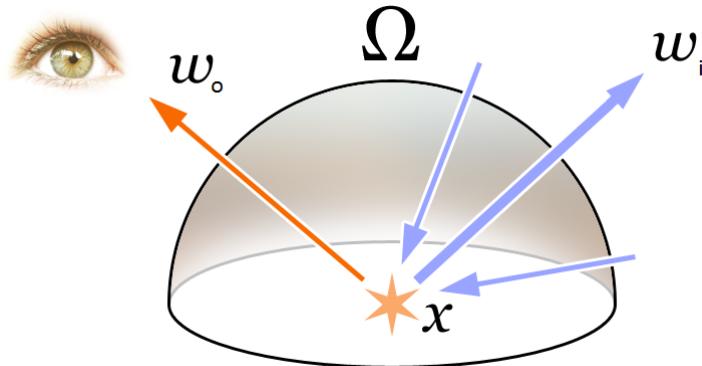


Figura 2.2: BRDF $l = \omega_i, v = \omega_o$ [Timrb, 2008]

Para obtener la radiancia total reflejada en un punto x en la dirección saliente v , es necesario integrar sobre el ángulo sólido en el dominio de la hemiesfera centrada en x .

$$L_o = \int_{\Omega_x} f(x, l, v) L_i(l) (l \cdot n) d\omega_i \quad (2.7)$$

2.2.1. Propiedades de la BRDF

Una BRDF debe cumplir ciertas propiedades para que sea físicamente plausible. En primer lugar debe cumplir la ley de conservación de la energía. En el caso que nos ocupa, esto significa que una superficie puede absorber luz, transformándola en calor, o puede reflejarla, pero en ningún caso puede reflejar más energía lumínica que la que recibe.

$$\forall l, \int_{\Omega_x} f(x, l, v)(n \cdot v) d\omega_o \leq 1 \quad (2.8)$$

En términos informales esta ecuación significa que la integral de toda la luz reflejada debido a un rayo de luz entrante nunca podrá ser superior a la luz entrante por ese rayo.

Además también debe cumplir el *principio de reciprocidad de Helmholtz*, esto significa que si intercambiamos los vectores l y v , su valor se mantiene. Este hecho cobra sentido si pensamos que la BRDF es una característica intrínseca de cada material y que al intercambiar los vectores v y l el ángulo entre ellos sigue siendo el mismo.

$$f(x, l, v) = f(x, v, l) \quad (2.9)$$

2.2.2. Isotropía y anisotropía de la BRDF

Una BRDF isotrópica es aquella en la que su valor se mantendrá constante si aplicamos la misma rotación a v y a l alrededor de la normal de la superficie. Por el contrario una BRDF anisotrópica cambiará su valor dependiendo de la rotación de v y l alrededor de la normal.

2.2.3. Modelo de Phong modificado

Una de las BRDFs más usadas en los algoritmos de ray tracing estocástico es la BRDF basada en el modelo de Phong modificado, que fue descrito por primera vez por Lewis (1994) y posteriormente explorado en más profundidad en [Lafortune and Willems, 1994]. Esta BRDF está basado en el conocido modelo de reflexión local de Phong [Phong, 1975] que fue adaptado por Lewis para cumplir con el principio de conservación de la energía.

En la forma usada por Lafortune y Willems la función aparece como:

$$f(x, l, v) = \frac{k_d}{\pi} + k_s \frac{n+2}{2\pi} \cos^n \alpha \quad (2.10)$$

donde α es el ángulo entre la dirección de reflexión especular perfecta y l . En esta forma, la función esta normalizada para conservar la energía, pero además para que esto sea cierto se debe cumplir $k_d + k_s \leq 1$.

2.3. Ecuación de renderizado

La ecuación de renderizado fue desarrollada en los años 80 simultáneamente y de forma independiente por distintos autores [Kajiya, 1986, Immel et al., 1986]. Se trata de una ecuación integral que unifica y formaliza los distintos algoritmos de renderizado, ya que hasta ese momento no existía un marco de trabajo teórico común.

Existen varias versiones de esta ecuación, según el autor que la use, que en general se pueden clasificar en dos tipos: las que integran sobre la hemiesfera, que se corresponde con la ecuación propuesta por Immel y las que integran sobre la unión de las superficies de la escena, que es la versión propuesta por Kajiya.

Consideremos la ecuación 2.7 y consideremos que además de dispersar luz una superficie también puede emitir luz, siendo L_e la radiancia de la luz emitida, entonces tenemos la ecuación de renderizado.

$$L_o = L_e + \int_{\Omega_x} f(x, l, v) L_i(l)(l \cdot n) d\omega_i \quad (2.11)$$

Es decir, que la radiancia total L_o que sale de un punto x es igual a la radiancia emitida por ese punto en la dirección de salida v más la integral de toda la radiancia que llega a ese punto y es reflejada en la dirección de salida.

Lo significativo de esta ecuación, es que resulta muy intuitivo derivar algoritmos de renderizado de la misma: se evalúa para cada punto a pintar y se evalúa L_i recursivamente hasta que se cumpla determinada condición.

El problema es que no parece factible encontrar una solución analítica de esta ecuación y por este motivo se aplican métodos de integración numérica para aproximar una solución.

2.4. El método de Montecarlo

El método de Montecarlo se trata de un método de integración numérico para integrales definidas sobre un dominio de dimensión arbitraria, del tipo:

$$I = \int_D f(x)dx, D \subseteq \mathbb{R}^m \quad (2.12)$$

Sabemos que la esperanza de una función continua se define como la integral de la función por la probabilidad de x . Y que podemos estimar la esperanza calculando la media de los valores que toma la función en puntos aleatorios escogidos independientemente y con la misma distribución.

$$E(f(x)) = \int f(x)p(x)dx \approx \frac{1}{N} \sum_{i=1}^N f(x_i) \quad (2.13)$$

El método de Montecarlo se basa en este hecho para estimar el valor de una integral definida tomando muestras aleatorias sobre el dominio $x_1, x_2, \dots, x_n \in D$ y aplicando:

$$I = \int_D f(x)dx \approx \frac{1}{N} \sum_{i=1}^N \frac{f(x_i)}{p(x_i)} \quad (2.14)$$

Siendo $p(x_i)$ la probabilidad de tomar una muestra x_i concreta de entre todas las posibles en el dominio D . En el caso de tomar las muestras sobre una distribución uniforme en D :

$$\forall x_i, p(x_i) = \frac{1}{\int_D dx} \quad (2.15)$$

$$I \approx \frac{\int_D dx}{N} \sum_{i=1}^N f(x_i) \quad (2.16)$$

El error en una estimación de este tipo se reduce a medida que N crece.

2.4.1. Muestreo de importancia

Otra forma de reducir el error a parte de tomar más muestras es tomarlas de forma más inteligente. Anteriormente hemos supuesto que tomamos las muestras de una distribución uniforme sobre el dominio, pero el método de Montecarlo no impone ninguna limitación en este aspecto. Esto implica que podemos tomar las muestras de otro tipo de distribuciones, que sean más apropiadas para cada caso. Por ejemplo tomando más muestras en aquellas partes del dominio de integración que sean más interesantes o importantes para nuestros propósitos.

Para ello basta con tomar las muestras x_1, x_2, \dots, x_n según la distribución usada y substituir $p(x_i)$ en la ecuación 2.14 por la probabilidad correspondiente.

2.4.2. Muestreo estratificado

El muestreo estratificado es otro método para reducir la varianza de la estimación. En este caso lo que se hace es dividir el dominio de la integral en regiones y aplicar Montecarlo para cada región.

2.5. Aplicaciones del muestreo de importancia

En esta sección veremos dos aplicaciones prácticas en los algoritmos de ray tracing estocástico del muestreo de importancia y como usando esta técnica es posible reducir el número de muestras necesarias para obtener una buena estimación de la integral.

2.5.1. Muestreo de la BRDF

Aunque algunos de los conceptos explicados aquí son aplicables a otros modelos, para esta explicación nos centraremos en la BRDF de Phong modificada, ya que es la usada en nuestra implementación, además de una de las más exploradas [Lafortune and Willems, 1994].

El problema que nos ocupa se nos presenta cuando tenemos que estimar la luz que llega a un punto del espacio. Una aproximación *naive*, sin usar muestreo de importancia, sería muestrear uniformemente direcciones en la hemiesfera y evaluar la BRDF para cada dirección. Esto puede funcionar bien cuando se trata de materiales puramente difusos, sin componente especular o con un lóbulo especular muy abierto. Por el contrario en el caso de encontrarnos con una superficie altamente especular, como por ejemplo un metal, la cantidad de muestras necesarias para obtener una estimación decente sería desorbitada, ya que la probabilidad de elegir una dirección dentro del lóbulo especular sería muy pequeña.

Cuando nos encontramos en la tesitura de tener que muestrear proporcionalmente a la BRDF, lo primero que habrá que tener en cuenta será decidir si muestrear la parte difusa o la parte especular de la BRDF, de forma proporcional a las características del material en cuestión. Es decir que para un rayo, la probabilidad de muestrear el lóbulo difuso sera de k_d , la probabilidad de muestrear el lóbulo especular sera k_s y la probabilidad de ser absorbido sera $1 - k_d - k_s$.

Para ello generaremos un número aleatorio $0 \leq r \leq 1$ y si $r \leq k_s$ muestreamos la parte especular, si $k_s < r \leq k_s + k_d$ muestreamos la parte difusa y si $k_s + k_d < r$ el rayo sera absorbido por el material.

Una vez decidido el evento a evaluar procederemos de forma distinta según cada caso. En el caso de evaluar la parte difusa, podemos muestrear uniformemente sobre la hemiesfera o podemos muestrear usando una distribución coseno como defienden algunos autores.

Una forma sencilla de obtener puntos distribuidos con densidad de coseno en la hemiesfera es generar puntos uniformemente sobre un círculo unitario y luego proyectarlos a la hemiesfera. Para ello generamos dos números aleatorios r_1 y r_2 en el intervalo $[0, 1]$ y hacemos:

$$\begin{aligned}x &= r_1 \cos(2\pi r_2) \\y &= r_1 \sin(2\pi r_2) \\z &= \sqrt{1 - x^2 - y^2} \\l &= (x, y, z)\end{aligned}$$

En este caso la función de densidad de probabilidad (pdf) es

$$pdf(l_{diff}) = \frac{(l \cdot n)}{\pi} \quad (2.17)$$

La parte más interesante es muestrear el lóbulo especular de la BRDF. Una buena explicación del procedimiento se encuentra en [Lafortune and Willems, 1994]. En este caso la función de densidad de probabilidad es

$$pdf(l_{spec}) = \frac{n+1}{2\pi} \cos^m \theta \quad (2.18)$$

Siendo θ el ángulo entre l y el vector $R = -v + 2n(n \cdot v)$, es decir v reflejado respecto a la normal, que es la dirección de reflexión especular perfecta. De este modo conseguiremos direcciones más parecidas a R cuanto mayor sea el exponente especular.

El vector resultante de muestrear esta pdf, en coordenadas esféricas:

$$(\phi, \theta) = (2\pi r_1, \arccos(r_2^{\frac{1}{n+1}})) \quad (2.19)$$

En ambos casos se prosigue trazando el rayo con la dirección obtenida y se calcula la radiancia reflejada en la dirección v en función de la radiancia entrante $L(x \leftarrow l)$ del siguiente modo:

si estamos evaluando el factor difuso

$$L(x \rightarrow v) = \frac{L(x \leftarrow l) k_d (n \cdot l)}{q_1 pdf_{diff}(l)} \quad (2.20)$$

si evaluamos el factor especular

$$L(x \rightarrow v) = \frac{L(x \leftarrow l) k_s (n \cdot l) (R \cdot l)^n}{q_2 pdf_{spec}(l)} \quad (2.21)$$

siendo R el vector v reflejado respecto a la normal.

si el rayo es absorbido, la contribución es 0.

2.5.2. Muestreo del ángulo sólido subtendido

Como en este trabajo estamos calculando la integral de la luz dispersada integrando la hemiesfera, cuando tengamos que muestrear una fuente de luz será necesario integrar el ángulo sólido subtendido por dicha fuente de luz. A priori, no sabemos si la fuente de luz estará ocluida, pero si que podemos saber que ángulo subtende con respecto al punto sobre el que estamos calculando la luz que llega. Por lo tanto, la idea será lanzar solo rayos dentro de este ángulo sólido para que vayan dirigidos hacia la fuente de luz.

Para proceder de este modo sera necesario calcular el ángulo sólido y definir una estrategia de muestreo para cada tipo de luz presente. En este trabajo hemos utilizado luces esféricas, pero es posible aplicar un método similar para luces poligonales.

Esfera

Para el muestreo del ángulo sólido subtendido por una luz esférica nos basaremos en el método propuesto en [Shirley et al., 1996].

En primer lugar tenemos que calcular el ángulo sólido en relación al punto x subtendido por una esfera de radio r centrada en c . Llamaremos θ_{max} al ángulo entre la dirección del vector $c - x$ y una recta tangente a la esfera que pasa por x . Entonces

$$\cos \theta_{max} = \sqrt{1 - \sin^2 \theta_{max}} = \sqrt{1 - \left(\frac{r}{\|c - x\|} \right)^2} \quad (2.22)$$

El ángulo sólido subtendido por la esfera en x según su definición es:

$$\omega_x = 2\pi(1 - \cos \theta_{max}) \quad (2.23)$$

La función de densidad de probabilidad de muestrear una dirección cualquiera l dentro del ángulo sólido será la inversa del ángulo sólido subtendido es decir $p(l) = \frac{1}{\omega_x}$.

Para obtener direcciones dentro del ángulo sólido ω_x generamos dos números aleatorios r_1 y r_2 uniformemente en el rango $[0, 1]$, la dirección en coordenadas esféricas será:

$$(\theta, \phi) = (\arccos(1 + r_1(\cos \theta_{max} - 1)), 2\pi r_2); \quad (2.24)$$

Para obtener la dirección del vector l en coordenadas absolutas, creamos una base ortonormal (u, v, w) con $w = \frac{c-x}{\|c-x\|}$ y multiplicamos la matriz de la base ortonormal por la representación en coordenadas cartesianas de la dirección que hemos obtenido.

$$l = \begin{pmatrix} u_x & v_x & w_x \\ u_y & v_y & w_y \\ u_z & v_z & w_z \end{pmatrix} \begin{pmatrix} \cos \phi \sin \theta \\ \sin \phi \sin \theta \\ \cos \theta \end{pmatrix} \quad (2.25)$$

Capítulo 3

INTRODUCCIÓN A OPTIX

OptiX es una librería desarrollada por Nvidia para hacer ray tracing en la GPU. OptiX fue desarrollada con el propósito de ser lo más flexible posible y adaptarse a las necesidades del programador. Por ello sólo ofrece la funcionalidad de lanzar rayos y es responsabilidad del programador implementar el funcionamiento de esos rayos: como serán lanzados, que datos portarán, que sucederá cuando interseccionen con un objeto, etc.

Esta funcionalidad se implementa mediante lo que en OptiX se llaman programas. Los programas de OptiX son fragmentos de código CUDA con acceso a las funciones de OptiX, básicamente para el trazado de rayos, que serán compilados por el compilador de Nvidia (nvcc) y se ensamblaran en kernels CUDA para ser ejecutados en la GPU.

De ahora en adelante cuando hablamos de programas nos estaremos refiriendo a estos fragmentos de código CUDA, cuando queramos referirnos al conjunto del sistema de renderizado, que es el objeto de este trabajo usaremos la expresión aplicación o sistema. Además en el contexto de CUDA y OptiX se conoce como Device a la GPU y como Host a la CPU.

3.1. Device

OptiX tiene un conjunto de programas que pueden ser implementados. Cada uno de ellos se encarga de una tarea específica dentro del flujo de ejecución de una aplicación OptiX. En esta sección veremos cuales son estos tipos de programas y que utilidad tiene cada uno de ellos.

3.1.1. Ray generation programs

Este tipo de programas son los puntos de entrada de una aplicación OptiX y es desde donde se crean y lanzan los rayos primarios. Típicamente, en una aplicación de renderizado se implementa la cámara en un programa de este tipo.

3.1.2. Intersection programs

Los programas de intersección se encargan de determinar si un rayo intersecciona con un objeto y en caso afirmativo retorna la distancia a la que se ha producido la intersección. Además, el programador tiene flexibilidad para calcular y retornar datos relativos a esta intersección, típicamente las coordenadas de textura, la normal a la superficie en el punto de intersección, etc.

El hecho de que el programador pueda determinar si se ha producido una intersección ofrece gran flexibilidad para implementar distintos tipos de superficies, desde un simple triángulo o esfera, a complejas superficies procedimentales.

3.1.3. Bounding box programs

Estos programas van ligados a los programas de intersección y su función es la de calcular una AABB (del inglés Axis Aligned Bounding Box) que contenga el objeto con el que están asociados. La implementación de estos programas no es obligatoria para que OptiX pueda determinar la intersección, pero aceleran el proceso y son necesarios si se quiere construir una estructura de aceleración.

3.1.4. Closest hit programs

Este es el tipo de programa más interesante para el caso que nos ocupa, ya que aquí es donde se calcula el resultado final de una intersección, normalmente el color de un punto en el espacio. Aquí es donde se hacen los accesos a texturas, se hace el shading y se pueden lanzar rayos recursivamente.

Estos programas, como su nombre indica, se ejecutan solo para la intersección más cercana del rayo con la escena.

3.1.5. Any hit programs

Por el contrario, los programas any hit, se ejecutan para todas las intersecciones que encuentre un rayo en su camino, a menos que explicitamente se termine la ejecución del rayo. Un uso típico de estos programas es para el calculo de sombras, si cualquier punto de la escena ocluye la luz, se termina el rayo y se retorna este hecho. Lo cual ofrece un mayor rendimiento en contraposición a tener que esperar a encontrar la intersección más cercana.

3.1.6. Otros programas

Los programas que hemos visto hasta ahora son los más relevantes y los que ofrecen la mayoría de funcionalidades, pero existen otros programas que se pueden implementar para ofrecer otras funcionalidades.

3.2. Host

Además de la parte del device que hemos visto hasta ahora, OptiX también ofrece una API en el host que sirve para encapsular y ensamblar todos los programas del device, además de para hacer las transacciones de memoria entre el host y el device. Aunque la API de host es en lenguaje C, OptiX también proporciona un envoltorio de la misma en C++ que sera el que usaremos.

3.2.1. Clase Program

La clase Program es la representación en lado host de los programas que irán al device. Se crea una instancia de un Programa proporcionando una cadena de caracteres con el código del programa compilado o una ruta al fichero que contiene el código compilado.

3.2.2. Clase Geometry

La clase geometry representa un objeto geométrico, se crea proporcionandole un Programa de intersección y un programa de bounding box.

3.2.3. Clase Material

Esta clase representa el material de un objeto y puede contener un closest hit program y un any hit program.

3.2.4. Clase GeometryInstance

La clase GeometryInstance sirve para crear una asociación entre una instancia de la clase Material y una instancia de la clase Geometry. Semánticamente lo que estamos haciendo al crear un GeomtryInstance es asociar un objeto geométrico de la escena con su material.

3.2.5. Clase Context

Una instancia de la clase context encapsula todos los datos necesarios para la ejecución de una aplicación optix, todos los objetos que hemos comentado anteriormente se ensamblan dentro de este y además el context contiene el programa de generación de rayos que sirve como punto de entrada. Cuando queramos realizar una ejecución de OptiX lo haremos a través del método launch del context.

3.2.6. Buffers

Además de las clases que hemos visto, OptiX también permite crear buffers para pasar datos de la memoria del sistema a la VRAM del device.

Capítulo 4

IMPLEMENTACIÓN

En este capítulo analizaremos la implementación realizada de los conceptos teóricos vistos hasta ahora y el uso práctico de las tecnologías involucradas.

El diseño de la aplicación está basado en el patrón de software *Facade* con la idea de ofrecer una interfaz en C++ de tipo librería de alto nivel, reutilizable y fácil de usar, que permita realizar renderizados realistas sin tener que preocuparse de los detalles de bajo nivel relativos a OptiX y CUDA, ni de todos los conceptos teóricos involucrados.

Además sobre esta interfaz se ha construido un parser de escenas XML, para hacer aún más sencillo el acto de configurar y crear escenas.

4.1. Interfaz de programación

4.1.1. Estructura TCamera

Este tipo de datos representa la cámara de la escena y encapsula todos los datos relativos a la cámara. Para construir un objeto de este tipo le pasamos la posición de la cámara, la posición del objetivo y el ángulo de campo de visión.

4.1.2. TSphereLight

Esta estructura representa una luz esférica, tomando como parámetros la posición del centro de la esfera, su radio y una tripleta RGB representando su emisión de luz.

4.1.3. TObjModel

Es una estructura que representa un modelo 3D en formato .obj. Se construye pasándole la ruta del modelo.

4.1.4. Clase CPathtracer

Este es el tipo de datos más relevante de nuestra interfaz. Todos los datos relativos a la escena y al renderizado son finalmente encapsulados en un objeto de este tipo. Una vez configurado, es el responsable de cargar los programas de OptiX y construir el *Context* de OptiX y controlar la ejecución del mismo.

4.1.5. Uso de la interfaz de programación

Para explicar el funcionamiento de la interfaz lo haremos mediante un sencillo ejemplo que contiene las clases y métodos principales de la interfaz.

```
1 CPathtracer pt;
2
3 pt.setRenderSize(width, height);
4 pt.setSqrtSamplesPerPixel(sqrtSpp);
5 pt.setBlockSize(200u);
6 pt.setMaxDepth(7);
7
8
9 pt.setCamera(TCamera(optix::make_float3(8.0, 9.0, 1.0),
10                      optix::make_float3(7.0, 9.0, 0.5), 60.0f));
11 pt.addObjModel(
12     TObjModel("assets/dabrovic-sponza/sponza.obj"));
13 pt.addLight(TSphereLight(optix::make_float3(2.0f, 2.0f, 2.0f),
```

```

15     optix::make_float3(0.0f, 5.0f, 0.0f), 1.0f));
16 pt.prepare();
17 pt.renderAllSamples();
pt.saveToTGA("render_sponza.tga");

```

Este ejemplo sirve para ilustrar el funcionamiento de la interfaz con el motor de renderizado. Veamos el funcionamiento de este ejemplo: En primer lugar creamos una instancia de la clase **CPathtracer** para seguidamente especificar las dimensiones, en pixeles, de la imagen que vamos a renderizar usando el metodo *void setRenderSize(unsigned int width, unsigned int height)*.

Seguidamente usamos la instrucción *pt.setSqrtSamplesPerPixel(sqrtSpp)* para especificar el número de muestras por pixel. El valor que toma este método es la raíz cuadrada del número de muestras. Lo hacemos de este modo para asegurarnos de que la división de cada pixel se hará en regiones uniformes, dividiendo el pixel en *sqrtSpp* filas y *sqrtSpp* columnas.

En la siguiente linea, especificamos un tamaño aproximado de los bloques en los que dividiremos la imagen. Entraremos en más detalle sobre este comportamiento en las secciones siguientes, pero de momento quedémonos con la idea de que la instrucción *pt.setBlockSize(200u)*; le dice al renderizador que debe dividir la imagen en bloques de aproximadamente 200x200 pixels.

A continuación especificamos la profundidad de recursión máxima. Este valor equivale al número de veces que un rayo rebotará por la escena antes de ser terminado. Hay que calibrar bien este valor, ya que un valor muy bajo produciría una iluminación pobre de la escena y un valor demasiado alto incrementaría el tiempo de ejecución innecesariamente.

Seguidamente le pasamos al pathtracer un objeto de tipo **TCamara** inicializado con los valores de posición, objetivo y ángulo de visión deseado. Sólo puede haber una cámara activa a la vez por lo que la clase **CPathtracer** siempre usará la última que le especifiquemos.

Los métodos *addObjModel* y *addLight* añaden modelos .obj y luces a la escena. A diferencia de la cámara, podemos llamar a estos métodos tantas veces como queramos para añadir luces y geometría a la escena.

El método *prepare* toma todos los datos de configuración que hemos especificado anteriormente y los compila en un *Context* de OptiX para poder empezar el proceso de renderizado.

Usamos la instrucción *pt.renderAllSamples()* para efectuar un renderizado completo, de la imagen que tomará tantas muestras como hayamos especificado, y por

último guardamos el resultado en una fichero TGA. Por simplicidad en este ejemplo hemos usado el método *renderAllSample*, pero también existe la opción de renderizar sólo una muestra por pixel por llamada con el método *void renderNextSample()*. Este método sera útil cuando queramos dar un feedback visual del progreso del renderizado, calculando una muestra y pintando en pantalla el resultado cada vez.

4.2. Renderizado por bloques

Durante el proceso de implementación nos encontramos con una dificultad cuando se trataba de renderizar escenas con muchos polígonos: el sistema operativo impone un límite al tiempo de ejecución de una llamada a la GPU de unos pocos segundos. Si llamábamos a la ejecución de un Context de OptiX contra toda la imagen, es decir con tantos CUDA threads como pixels, el sistema operativo paraba la ejecución antes de que esta terminara.

La solución que encontramos fue dividir la imagen en bloques de menor tamaño y lanzar una ejecución del Context para cada uno de estos bloques. Es como pintar varias imágenes de menor tamaño y luego juntarlas, con la diferencia de que mantenemos un buffer en la memoria del device, del tamaño de la imagen final, para no tener que hacer transacciones de memoria entre el host y el device ya que esto reduciría notablemente el rendimiento.

4.3. Cámara

4.3.1. Host

En el host precalculamos los datos de la cámara que serán constantes durante la ejecución: los vectores de la base ortonormal del sistema de coordenadas de cámara (front, left, up) y la relación (viewd) entre el vector front y el vector up en función del ángulo de visión fov.

```
1  TCamera::TCamera(optix::float3 aposition, optix::float3 atarget,
2  	float afov)
3  {
4  	is_ok = false;
5  	if(afov <= 0.0f || afov >= 179.0f) return;
6
7  	position = aposition;
8  	target = atarget;
9  	fov = afov;
10  viewd = 0.0f;
11
12  front = optix::normalize(target - position);
13  left = optix::normalize(optix::cross(optix::make_float3(0.0f,
14  	1.0f, 0.0f), front));
15  up = optix::normalize(optix::cross(front, left));
16
17  viewd = 1.0 / tanf(fov * 0.5f * DEG2RAD);
18  is_ok = true;
19 }
```

4.3.2. Device

En el device tenemos un programa encargado de generar los rayos primarios que se lanzaran desde la cámara, para ello se le pasan desde el host una serie de parámetros necesarios para lanzar el rayo correcto.

```
1 float2 offset = make_float2(offset_x, offset_y);
2 float2 startSample = make_float2(launch_index) + offset
3     + make_float2(
4         (float)(currentSample % sqrtSPP) * (1.0f / (float)sqrtSPP),
5         (float)(currentSample / sqrtSPP) * (1.0f / (float)sqrtSPP)
6     );
7
8 float2 centerSample = make_float2(1.0f / float(2 * sqrtSPP));
9 float2 d = (startSample + centerSample) / make_float2(screen_dim
10    .x, screen_dim.y) * 2.f - 1.f;
11
12 d.x *= aspect_ratio;
13 float3 ray_origin = eye;
14 float3 ray_direction = normalize(d.x*U + d.y*V + W*viewD);
```

El offset son las coordenadas absolutas en espacio de imagen del pixel top-left del bloque que estamos pintando y el launch_index es el indice del CUDA thread actual que se corresponde con las coordenadas relativas al bloque actual del pixel que estamos pintando. Usamos estas variables para calcular las coordenadas top-left del pixel que estamos pintando, en coordenadas absolutas de la imagen entera.

Para que el muestreo múltiple de cada pixel proporcione antialiasing es necesario dividir el pixel en regiones y lanzar el rayo a través de la región que se corresponde con la muestra actual. Para calcular las coordenadas de la región actual lo hacemos a partir del parámetro currentSample y de la raíz cuadrada del total de muestras (la variable sqrtSPP).

Efectuados estos cálculos tenemos las coordenadas top-left absolutas de la región de pixel. A estas coordenadas le sumamos el valor de media región de pixel para que el rayo salga por el centro de dicha región, en vez de por su esquina superior-izquierda.

En la variable d normalizamos esas coordenadas para obtener las coordenadas de pantalla correctas en el intervalo $[-1, 1]$. Corregimos la relación de aspecto de la imagen y calculamos la dirección del rayo resultante.

4.4. Muestreo de la BRDF

Para el muestreo de la BRDF usamos el modelo propuesto en [Lafortune and Willem, 1994] que hemos explicado en la sección 2.5.1.

En primer lugar decidimos aleatoriamente si muestrear la parte difusa o la parte especular de la BRDF. Para ello proponemos usar como probabilidades las medias de los valores RGB de las componentes difusa y especular: $p_{diff} = \frac{1}{3}(k_{dR} + k_{dG} + k_{dB})$ y $p_{spec} = \frac{1}{3}(k_{sR} + k_{sG} + k_{sB})$

Generamos un número aleatorio r en $[0, 1]$ y si $r \leq p_{spec}$ muestreamos la parte especular, si $p_{spec} < r \leq p_{spec} + p_{diff}$ la parte difusa, en caso contrario el rayo es absorbido y terminamos la recursión del mismo.

Esta forma de terminar la recursión se conoce como ruleta rusa y es una técnica de reducción de la varianza usado en integraciones por Montecarlo. Para compensar esta terminación anticipada y la separación entre rayos difusos y especular dividiremos el resultado entre la probabilidad de que se lance ese rayo. Ya que se muestrea con más probabilidad el evento más probable esto servirá para dar más peso a aquellas muestras menos probables.

4.4.1. Muestreo de la parte difusa de la BRDF

En el programa de closest hit, muestreamos la hemiesfera centrada en el punto de intersección para tener una estimación de la luz incidente.

```
1  unsigned int seed = prd_radiance.seed;
2  float r1 = rnd(seed);
3  float r2 = rnd(seed);
4  float3 p;
5  cosine_sample_hemisphere(r1, r2, p);
6
7  float3 u, v;
8  createONB(ffnormal, u, v);
9  float3 rd = normalize(u * p.x + v * p.y + ffnormal * p.z);
10
11 PerRayData_radiance diffuse_refl_prd;
12 diffuse_refl_prd.seed = seed;
13 diffuse_refl_prd.depth = prd_radiance.depth + 1;
14 diffuse_refl_prd.contribution *= Kd;
15 diffuse_refl_prd.is_light = false;
16 optix::Ray diffuse_refl_ray( hit, rd, radiance_ray_type,
17     scene_epsilon );
18 rtTrace(top_object, diffuse_refl_ray, diffuse_refl_prd);
19 brdfLight = diffuse_refl_prd.result;
```

```
19||    color += brdfLight * Kd / prob_diff;
```

Básicamente lo que hace es, en el caso de que no hayamos llegado a la profundidad de recursión máxima, muestrear un punto en la hemiesfera con densidad de coseno, generar una base ortonormal en función de la normal de la superficie y transformar el punto de la hemiesfera con esa base. Se traza un rayo recursivamente en la dirección obtenida y el color devuelto por ese rayo se multiplica por el color difuso de la superficie.

Hay que hacer notar que en la instrucción `color += brdfLight * Kd / prob_diff;` no estamos multiplicando por el típico factor coseno de la normal y la luz. No lo hacemos porque al tomar la muestra con densidad coseno y dividir entre la densidad de probabilidad de esa muestra los factores coseno se anulan. Siguiendo con ese razonamiento y teniendo en cuenta que la densidad coseno es $\frac{\cos(\vec{l}, \vec{n})}{\pi}$ deberíamos multiplicar el resultado por π para ser coherentes con la teoría, pero lo hemos quitado del cálculo porque hemos comprobado empíricamente que el resultado final es más satisfactorio sin ese factor, que en caso de incluirlo en el cálculo generaba imágenes excesivamente iluminadas y quemadas.

Usando solamente esta función de iluminación ya es posible generar renderizados, aunque al no muestrear las luces directamente tardarán más en converger. En las primeras fases de la implementación realizamos algunos renderizados con este closest hit program, y un closest hit program para las fuentes de luz que devolvía su emisión de luz. Un ejemplo de ello puede verse en la siguiente figura.



Figura 4.1: Muestreo de la brdf difusa

4.4.2. Muestreo de la parte especular de la BRDF

En el caso de la parte especular usamos como densidad de probabilidad una potencia de coseno de forma que el exponente sea mayor cuanto más reflectante sea el material. Ese vector es posteriormente transformado con una base ortonormal orientada hacia la dirección de reflexión perfecta.

```
1 float3 eye = -ray.direction;
2 float3 perfect_specular = normalize(2.0 * ffnormal * dot(
3     ffnormal, eye) - eye);
4 createONB(perfect_specular, u, v);
5 p = sample_specular2(phong_exp, r1, r2);
6 float3 rd = normalize(u * p.x + v * p.y + perfect_specular * p
7 .z);
8 if(dot(rd, ffnormal) > 0.0f) {
9     PerRayData_radiance specular_refl_prd;
10    specular_refl_prd.seed = seed;
11    specular_refl_prd.depth = prd_radiance.depth + 1;
12    specular_refl_prd.contribution *= Ks;
13    specular_refl_prd.is_light = false;
14    optix::Ray specular_refl_ray( hit, rd, radiance_ray_type,
15        scene_epsilon );
16    rtTrace(top_object, specular_refl_ray, specular_refl_prd);
17    brdfLight = specular_refl_prd.result;
18
19    color += ( brdfLight * Ks * (phong_exp + 2) * dot(rd,
20        ffnormal) ) / ( (phong_exp + 1) * prob_spec );
21 }
```

La función *sample_specular2* genera una dirección aleatoria con una densidad de probabilidad que depende del exponente especular, tal como hemos visto en la sección 2.5.1.

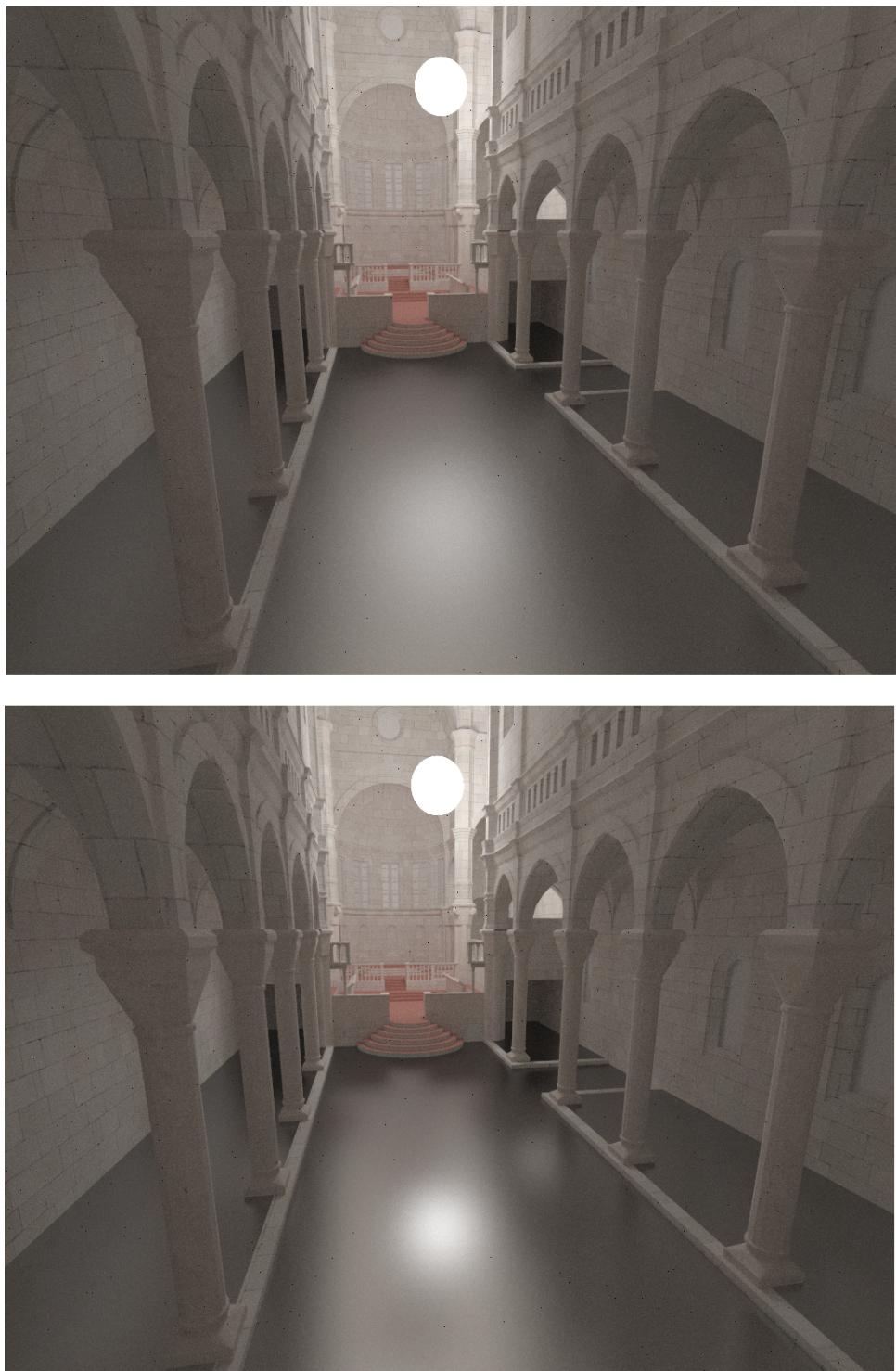


Figura 4.2: Arriba exponente especular de 64, abajo exponente de 512. Se puede apreciar que cuanto mayor sea el exponente más reflectante sera el material.

4.5. Muestreo de luces

Una mejora importante que hemos implementado ha sido el muestreo directo de fuentes de luz. Muestrear las luces directamente provoca que el renderizado converja más rápido al lanzar rayos hacia zonas de la escena que es más probable que aporten luz al computo final.

```
2  for(int i = 0; i < spherical_lights.size(); ++i) {
3      float3 light_dir = spherical_lights[i].center - hit;
4      float dist2 = dot(light_dir, light_dir);
5      float radius2 = spherical_lights[i].radius * spherical_lights[
6          i].radius;
7      if(dist2 - radius2 < scene_epsilon) {
8          continue;
9      }
10     unsigned int seed = prd_radiance.seed;
11     float cos_theta_max = sqrtf(1 - radius2/dist2);
12     float inv_pdf = 2.0f * PI * (1.0f - cos_theta_max);
13
14     float r1 = rnd(seed);
15     float r2 = rnd(seed);
16     float cos_theta = 1 + r1 * (cos_theta_max - 1);
17     float sin2theta = 1 - cos_theta * cos_theta;
18     float sin_theta = sqrtf(sin2theta);
19     float sin_phi = sinf(2 * PI * r2);
20     float cos_phi = cosf(2 * PI * r2);
21     float3 w = normalize(light_dir);
22     float3 u, v;
23     createONB(w, u, v);
24     float3 dir = normalize( u * cos_phi * sin_theta + v * sin_phi
25         * sin_theta + w * cos_theta );
26
27     if(dot(dir, ffnormal) < 0) continue;
28     PerRayData_shadow shadow_prd;
29     shadow_prd.contribution = spherical_lights[i].emission;
30     float delta = sqrtf(radius2 - sin2theta * dist2);
31     Ray shadow_ray = Ray( hit, dir, shadow_ray_type, scene_epsilon
32         , cos_theta * length(light_dir) - delta );
33     rtTrace(top_object, shadow_ray, shadow_prd);
34     color += inv_pdf * shadow_prd.contribution * dot(dir, ffnormal
35         ) * (Kd + Ks * powf(fmaxf(dot(dir, perfect_specular), 0.0f),
36         phong_exp) );
```

Para cada luz esférica calculamos la distancia del punto que estamos evaluando al centro de la esfera y en caso de encontrarnos dentro de la misma la omitimos y continuamos con la siguiente.

A continuación procedemos a generar una dirección dentro del ángulo sólido subtendido por la esfera, tal como se ha explicado en la sección 2.5.2. Si dicha dirección forma un ángulo superior a noventa grados con respecto a la normal la omitimos.

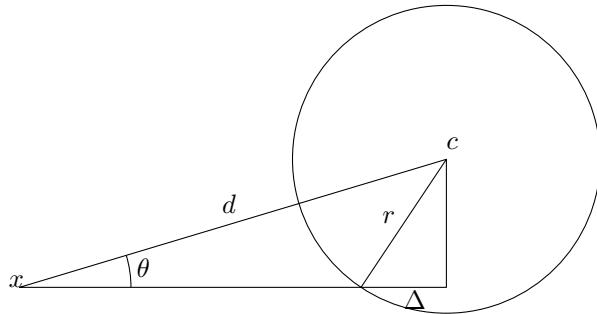


Figura 4.3: Δ al punto de intersección

La distancia a la que se encuentra el primer punto de intersección de la esfera con el rayo será $d_{intersect} = d \cos \theta - \Delta$. Este valor $\Delta = \sqrt{r^2 - (d \sin \theta)^2}$ lo calculamos en la instrucción `float delta = sqrtf(radius2 - sin2theta * dist2);`. De este modo cuando creamos el rayo lo haremos con una distancia máxima que es la que nos interesa con tal de que no interseccione con objetos que estén dentro o detrás de la esfera. En las líneas 29 y 30 creamos y lanzamos el rayo. El valor `shadow_prd.contribution` será la emisión de luz de la fuente que estamos muestreando o 0 si ha interseccionado con algún objeto.

Finalmente en la ultima linea calculamos la contribución de luz de la muestra evaluando la BRDF.



Figura 4.4: Fuente de luz esférica

Capítulo 5

CONCLUSIONES Y RESULTADOS

5.1. Algunos resultados



Figura 5.1: La textura del suelo es un mapa especular



Figura 5.2: Tres fuentes de luz poco visibles iluminan toda la escena

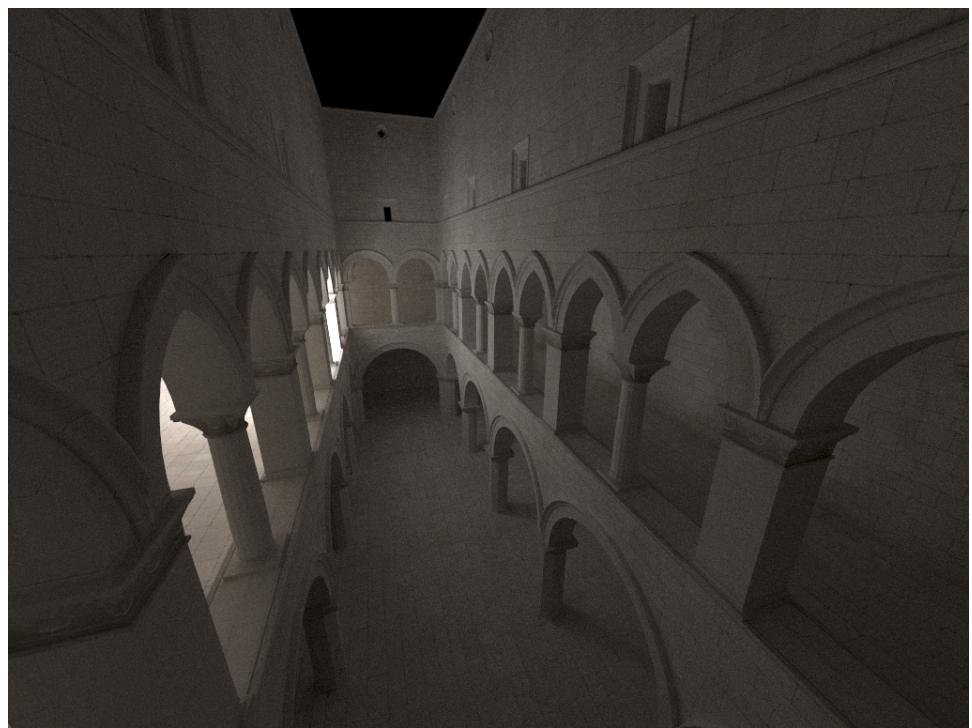


Figura 5.3: Iluminación por una única fuente de luz

5.2. Valoración de los resultados obtenidos

El campo de estudio en el que se sitúa el presente trabajo es muy amplio y ha sido muy estudiado por un gran número de investigadores. A medida que profundizábamos en la materia de estudio iban surgiendo cada vez más detalles, mejoras y variaciones en la teoría que incrementaban el abasto del proyecto y generaban nuevas dudas. Por ello hemos llegado a un punto en el que, por el alcance de un TFG, hemos tenido que empezar a descartar cosas que al autor le hubiese gustado investigar en más profundidad. Aún así, valoramos positivamente los resultados obtenidos, pues se ha alcanzado el objetivo principal de este trabajo que era implementar un algoritmo de renderizado realista. Además la implementación en la GPU ha demostrado ser bastante rápida, más aún si consideramos que la GPU usada en el desarrollo de este trabajo y en las pruebas realizadas es de una gama baja entre las presentes en el mercado.

5.3. Posibles mejoras

Planteamos tres grupos de posibles mejores bien diferenciados: En primer lugar tenemos las mejoras al algoritmo usado. El algoritmo de path tracing que hemos implementado es una de sus versiones más básicas y tal como hemos comentado en la introducción existen variaciones del mismo que ofrecen mejores resultados en un menor número de muestras. Por ejemplo, una primera mejora sería implementar el transporte de luz bidireccional que ofrecería mejores resultados cuando se trata de iluminar escenas en las que la fuente de luz está muy escondida con respecto a la zona enfocada por la cámara. Otra importante mejora sería explorar e implementar modelos de BRDFs más realistas, pues el que hemos usado es de los más básicos que hay. Además, en este trabajo hemos omitido el importante fenómeno de la refractancia de la luz.

Otro grupo de mejoras, más allá de este algoritmo en concreto, sería buscar formas de combinar distintos algoritmos. Aunque en principio el algoritmo utilizado es capaz de simular los fenómenos de la luz más habituales esto no significa que sea el mejor para todos ellos. Existen algoritmos que destacan en simular aspectos específicos del transporte lumínico y un buen motor de renderizado puede aprovechar ese hecho para combinar algoritmos de forma inteligente. Por ejemplo, versiones del algoritmo de radiosity o instant radiosity pueden usarse en una primera fase para computar un mapa de luz difusa de la escena. En una segunda fase se puede utilizar alguna de las variantes de path tracing para calcular la luz especular y la luz refractada, haciendo consultas al mapa de luz difusa cuando sea necesario. Finalmente una ejecución de photon mapping puede usarse para calcular las causticas con un mayor nivel de detalle. Una buena explicación de este uso combinado de algoritmos se puede encontrar en [Hery et al., 2013].

Por último y aunque no era el objetivo de este proyecto, una mejora interesante sería implementar una interfaz de usuario que permitiese configurar la escena de forma fácil y cómoda. Por ejemplo una interfaz con Qt con una vista OpenGL, que permita previsualizar la escena, cargar modelos tridimensionales y colocar las luces y la cámara facilitaría mucho el hecho de configurar escenas y crear nuevos renderizados.

5.4. Perspectivas de futuro

Durante el desarrollo de este proyecto hemos podido comprobar que las ganancias de velocidad al ejecutar el algoritmo de path tracing en la GPU, frente a hacerlo en la CPU, son muy notorias. Con nuestra implementación es posible renderizar escenas de varios millones de polígonos, con cientos de muestras por pixel en unos cuantos minutos. Realizar el mismo tipo de renderizado en una CPU actual podría tardar varios días.

Además, considerando que los experimentos se han realizado con una tarjeta gráfica de ordenador portátil y que la implementación es muy optimizable, creamos que en pocos años sera habitual disponer de software capaz de renderizar escenas con iluminación global en tiempo real.

Bibliografía

- [Appel, 1968] Appel, A. (1968). Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 37–45.
- [Ashikhmin and Shirley, 2000] Ashikhmin, M. and Shirley, P. (2000). An Anisotropic Phong BRDF Model.
- [Cook et al., 1984] Cook, R., Porter, T., and Carpenter, L. (1984). Distributed ray tracing. *ACM SIGGRAPH Computer Graphics*, 18(3):137–145.
- [Dutr , 2003] Dutr , P. (2003). Global Illumination Compendium. Technical report.
- [Goral et al., 1984] Goral, C. M., Torrance, K. E., Greenberg, D. P., and Battaile, B. (1984). Modeling the interaction of light between diffuse surfaces.
- [Haade, 2007] Haade (2007). Solid_Angle. Obtenido de <http://commons.wikimedia.org/wiki/File:Solid\Angle.png>.
- [Hery et al., 2013] Hery, C., Villemin, R., and Studios, P. (2013). Physically Based Lighting at Pixar. *Citeseer*, pages 1–23.
- [Immel et al., 1986] Immel, D., Cohen, M., and Greenberg, D. (1986). A radiosity method for non-diffuse environments. *ACM SIGGRAPH Computer ...*, 20(4):133–142.
- [Jensen, 1996] Jensen, H. (1996). Global illumination using photon maps. *Rendering Techniques' 96*, 96:21–30.
- [Kajiya, 1986] Kajiya, J. (1986). The rendering equation. *ACM Siggraph Computer Graphics*, 20(4):143–150.
- [Keller, 1997] Keller, A. (1997). Instant radiosity. *Proceedings of the 24th annual conference on Computer graphics and interactive techniques SIGGRAPH 97*, 31:49–56.

- [Lafortune and Willem, 1993] Lafortune, E. and Willem, Y. (1993). Bi-directional path tracing. *Proceedings of CompuGraphics*.
- [Lafortune and Willem, 1994] Lafortune, E. and Willem, Y. (1994). Using the Modified Phong brdf for Physically Based Rendering. Technical report.
- [Lewis, 1994] Lewis, R. R. (1994). Making Shaders More Physically Plausible. *Computer Graphics Forum*, 13(2):109–120.
- [Nicodemus, 1965] Nicodemus, F. (1965). Directional reflectance and emissivity of an opaque surface. *Applied Optics*, 4(7):767–773.
- [Phong, 1975] Phong, B. T. (1975). Illumination for computer generated pictures.
- [Shirley et al., 1996] Shirley, P., Wang, C., and Zimmerman, K. (1996). Monte Carlo techniques for direct lighting calculations. *ACM Transactions on Graphics (TOG)*, pages 1–28.
- [Timrb, 2008] Timrb (2008). Rendering_eq. Obtenido de http://commons.wikimedia.org/wiki/File:Rendering_eq.png.
- [Torrance and Sparrow, 1967] Torrance, K. and Sparrow, E. (1967). Theory for off-specular reflection from roughened surfaces. *JOSA*.
- [Veach, 1997] Veach, E. (1997). *Robust Monte Carlo methods for light transport simulation*. PhD thesis.
- [Veach and Guibas, 1995] Veach, E. and Guibas, L. J. (1995). Optimally combining sampling techniques for Monte Carlo rendering. *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques - SIGGRAPH '95*, pages 419–428.
- [Walter et al., 2007] Walter, B., Marschner, S., Li, H., and Torrance, K. (2007). Microfacet models for refraction through rough surfaces. *Proceedings of the 18th . . .*
- [Whitted, 1979] Whitted, T. (1979). An improved illumination model for shaded display.