Instituto de Computación, Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

PROYECTO DE GRADO EN
INGENIERÍA EN COMPUTACIÓN

Resolución de Problemas Inversos de Iluminación Utilizando Photon Mapping

Ignacio Avas

Diciembre de 2015

Dr. Ing. Eduardo Fernández	Supervisor
Ing. José Aguerre	Tribunal
Mag. Ing. Martín Pedemente	Tribunal
Dr. Ing. Marcos Viera	Tribunal

Resumen

La luz es un elemento clave para percibir el entorno, por lo tanto debe ser tenido en cuenta durante el proceso de diseño arquitectónico. En general, quien diseña debe satisfacer ciertas *intenciones de iluminación* en una escena y está limitado a ciertas restricciones, por ejemplo puede necesitar satisfacer restricciones de eficiencia energética, puede preguntársele cuál es la mejor distribución de las luces de una habitación para aprovechar de manera eficiente la energía eléctrica, o también puede requerírsele la ubicación óptima de un objeto de forma que se maximice el uso de la luz natural. Los *problemas inversos de iluminación* (ILPs) son el tipo de problemas propuestos para satisfacer las intenciones de iluminación. Este tipo de problemas se pueden plantear y resolver como problemas de optimización.

La propuesta de este trabajo es implementar un programa que permita la de resolución de ILPs utilizando la técnica de *Photon Mapping* (Mapeo de Fotones). El uso de esta técnica permite manejar restricciones de iluminación y otros aspectos de las escenas que no pueden ser resueltos con otras técnicas como la Radiosidad. Por otra parte, photon mapping es más eficiente para computar restricciones en ILPs que requieran la variación de la geometría, o la inclusión de reflexiones no difusas. También es de interés explorar las ventajas de usar el poder computacional de las tarjetas gráficas (GPUs) para resolver estos problemas, donde es posible implementarlos eficientemente utilizando bibliotecas especializadas en la traza de rayos (ray tracing) como OptiX. Las técnicas desarrolladas podrían ser parte de un paquete de diseño arquitectónico para ser utilizado por arquitectos o ingenieros. [1]

Palabras clave: problema inverso de iluminación, intenciones de iluminación, *photon mapping*, optimización, OptiX, GPU

[1] El código fuente de la implementación y la versión electrónica del documento se encuentran disponibles en `https://github.com/igui/ILPSolver`.

Índice general

1 Introducción

Las *intenciones de iluminación* (LI) refieren a aquellos objetivos a lograr en un espacio arquitectónico, relacionados con la luz. Las LI son restricciones y objetivos que se deben cumplir en la escena, por ejemplo: ¿cuál es la mejor ubicación de una ventana en una habitación para aprovechar de mejor manera la luz natural que llega desde el exterior, o cuál es la posición óptima de las luces en una pasarela de un museo para iluminar los cuadros que se encuentran ya dispuestos en las paredes de la misma? En este sentido quien diseña o proyecta puede tener requisitos como los planteados durante el proceso de diseño. Los problemas inversos de iluminación (*ILP* por sus siglas en inglés) son aquellos donde se busca encontrar las configuraciones óptimas que satisfagan las intenciones de iluminación. En este sentido un ILP puede ser tratado como un problema de optimización en el que se busca cumplir un conjunto de intenciones de iluminación.

Una forma posible de resolver un ILP es recorrer el conjunto completo de posibles configuraciones, evaluarlas una a una para saber si se cumplen las intenciones de iluminación y luego elegir aquella que sea la *mejor* en los términos planteados en el problema. Un planteo como el mencionado resulta prohibitivo, porque recorrer completamente el dominio de todas las configuraciones posibles no es posible desde el punto de vista computacional. En este contexto es beneficioso el uso de alguna metaheurística que guíe el proceso de optimización, seleccionando inteligentemente qué configuraciones del espacio se exploran. Para evaluar cada configuración generalmente se utiliza alguna técnica de *iluminación global*. Estas técnicas modelan diferentes tipos de transporte de la luz para resolver la *ecuación de rendering* (Kajiya, 1986), donde la luz que pasa por un punto depende de la luz que es reflejada y emitida por otros puntos en la habitación. En este sentido los ILP ya fueron abordados en otros trabajos por Contensin (2002), Patow y Pueyo (2003), Fernández (2014), entre otros, utilizando técnicas basadas en radiosidad. Esta técnica, como se verá más adelante, tiene limitaciones para manejar algunas intenciones de iluminación, que incluyen cambiar la posición de los objetos en el espacio, o manejar ciertos tipos de superficies. *Photon Mapping* (PM) o Mapeo de Fotones (Jensen, 2001) es un método para computar la iluminación global de un espacio, que es capaz de capturar efectos como las reflexiones no difusas, refracciones, o cáusticas. Además las técnicas basadas en radiosidad, utilizan matrices para modelar los flujos de luz de un punto al otro del espacio, lo que en algunos casos implica volver a calcular esas matrices si se hacen cambios en la geometría. Eso es costoso y photon mapping no utiliza matrices, si no que usa otras estructuras de datos para representar eso, que son más flexibles al cambio de elementos de la geometría del espacio, y por tanto, parecen prometedoras las posibilidades de photon mapping en la resolución de ILPs.

Las unidades de procesamiento gráfico (*GPU*s) son capaces de realizar en paralelo muchas tareas utilizando una arquitectura de *Single Instruction Multiple Data*. Hace unos

años se ha desarrollado la tendencia de utilizar GPUs para resolver problemas que tienen una naturaleza paralela, pues su rendimiento sobrepasa al de las unidades centrales de proceso (*CPUs*) para ciertos tipos de tareas. Hoy en día, existen bibliotecas como *CUDA* y *OptiX*, que brindan herramientas para construir programas usando las GPUs de manera amigable para alguien que quiera desarrollar programas. Photon mapping es un método que es inherentemente paralelo, dado que cada fotón se comporta de forma independiente, y una simulación puede implicar el modelado de millones de fotones, usando el mismo código para cada fotón. Por tanto resulta natural utilizar la GPU para paralelizar esta tarea.

1.1. Motivación y Contribución

En el contexto de lo explicado anteriormente, la motivación de este proyecto es explorar la utilización de photon mapping en las GPUs para resolver problemas inversos de iluminación. El objetivo es explorar los desafíos implicados en la construcción de una biblioteca de resolución de ILP que pueda usarse como parte de un paquete CAD de diseño arquitectónico. Se desea demostrar que utilizar photon mapping para evaluar la iluminación de un espacio brinda una alternativa válida, y que en algunos casos puede superar a las técnicas basadas en radiosidad.

Se propone el uso la metaheurística *Variable Neighbourhood Search* (VNS) (Mladenović y Hansen, 1997) para la resolución de ILP. Además se estudiarán maneras de mejorar el rendimiento del proceso de optimización, de forma que se puedan obtener los resultados en un menor tiempo. Entre otras mejoras se utiliza un subconjunto básico del photon mapping. También se muestran resultados experimentales, en algunos ILP simples en términos de su complejidad, y se probará la implementación en esos problemas para corroborar su correctitud. Los resultados harán ver que el paquete construido puede ser utilizado como una herramienta de solución de ILPs, y se dejarán planteadas posibles mejoras a futuro.

1.2. Estructura del Informe

El resto del informe se estructura como sigue:

- El Capítulo 2 introduce el trabajo previo y el estado del arte.
- En el Capítulo 3 se profundiza en el estudio de los ILP y se propone una técnica para resolverlos utilizando metaheurísticas de optimización y photon mapping.
- Los detalles de la implementación construida se describen en el Capítulo 4.
- En el Capítulo 5 se presentan los resultados obtenidos y se analiza el rendimiento de la implementación.
- El Capítulo 6 contiene las conclusiones y menciona el trabajo futuro.

2

Trabajo Previo

La intención de este capítulo es introducir los conceptos necesarios para entender la solución que se desarrolló en este trabajo. En este capítulo se introducirán los fundamentos teóricos en los que se basan los métodos de generación de imágenes usados en el proyecto. Luego se presentarán distintos métodos para el cálculo de la radiosidad en una escena, junto con sus ventajas y limitaciones; se hará especial énfasis en photon mapping pues fue el utilizado en la implementación. Luego se mencionarán algunas bibliotecas importantes que se usaron para la implementar la solución. Finalmente se introducirá la metaheurística Variable Neighbourhood Search que es usada durante el proceso de optimización y exploración del espacio de soluciones.

2.1. Teoría de Transporte de la Luz

La naturaleza de cómo la luz se difunde a través del espacio no es completamente conocida. Hay varios modelos que la describen de distintas maneras. Ellos se pueden dividir en tres categorías según se describe en Saleh y Teich (2007):

- **Los modelos ópticos basados en Rayos** que interpretan la luz como rayos que se propagan en el espacio siguiendo reglas geométricas. Estos modelos se pueden usar para describir la mayoría de los efectos comunes que se dan en la renderización de imágenes, como la reflexión y la refracción.

- **Los modelos ópticos basados en Ondas y basados en Electromagnetismo**, que modelan la luz como ondas electromagnéticas, y sirven para explicar todos los efectos del modelo anterior, incluyendo interferencia, difracción, dispersión y polarización.

- **Los modelos basados en fotones** que entienden a la luz como partículas discretas que interactúan con la materia, afectando la forma en que se propagan en el espacio. Este modelo puede lograr interpretar todos los efectos que se logran si se modela con rayos. La difusión de la luz a través de las moléculas y partículas de la materia, se interpreta de manera abstracta como una interacción con una superficie, a partir de ciertas propiedades que la misma posee. Es decir, que se ignoran efectos derivados de los modelos basados en ondas.

2.2. Ecuación de la Luz

Los métodos de iluminación global que se usaron como base, parten de la ecuación de rendering (Kajiya, 1986), que se expresa como:

$$I(x, x') = g(x, x') \left(\varepsilon(x, x') + \int_{s \in S} \rho(x, x', s) I(x', s) ds \right) \tag{2.1}$$

Donde:

- $I(x', s)$ denota la intensidad de la luz desde s hacia x'.
- $g(x, x')$ es un término geométrico relacionado con cuanta luz puede llegar de x' a x.
- $\varepsilon(x, x')$ es la intensidad de la luz emitida desde x' hacia x.
- $\rho(x, x', s)$ proporción de la luz reflejada hacia x desde x' que viene en la dirección de s.

En la Figura 2.1 se ilustran algunos de los elementos de la Ecuación 2.1.

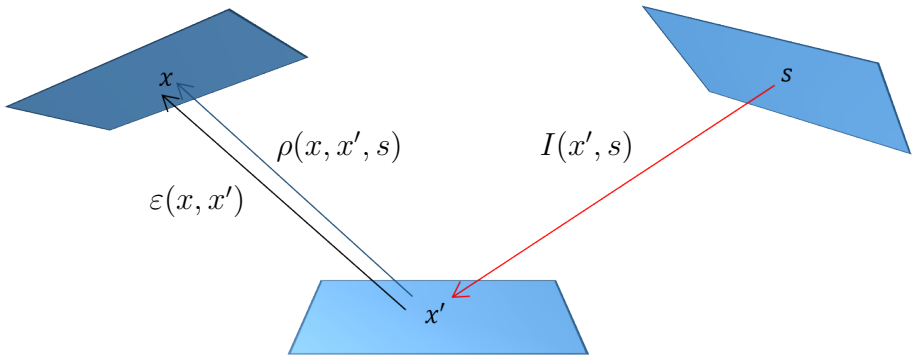
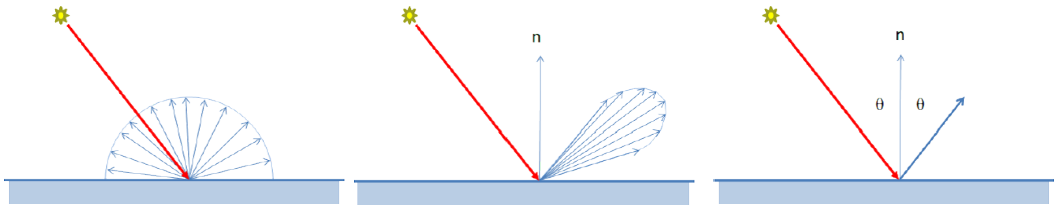


Figura 2.1: Elementos de la ecuación de Rendering

Por ejemplo si x' y x están ocluidos, entonces el valor de $g(x, x')$ es 0, pero si son visibles directamente, es $1/r^2$, donde r es la distancia entre los dos puntos. La función $\rho(x, x', s)$ depende directamente de la naturaleza de la superficie en la que se encuentra el punto x'. Una superficie difusa (lambertiana) tiene un valor constante de ρ para todas las direcciones posibles, lo cual simplifica el cálculo. Otros tipos de superficies con un efecto brillante, espejado, o *glossy*, pueden tener un valor mayor en cierta dirección, pero descendiendo bruscamente en otras. La Figura 2.2 ilustra estos efectos.



(a) Reflexión lambertiana (b) Reflexión glossy (c) Reflexión especular

Figura 2.2: Distintos tipos de reflexiones (Fernández, 2014)

Como se explica en Kajiya (1986) la ecuación Rendering es independiente del tiempo por lo que no se modelan efectos como fosforescencia, donde la luz que se emite fue absorbida previamente, con frecuencia en una longitud de onda diferente. En este modelo e asume que la velocidad de la luz es infinita. Una escena alcanza inmediatamente un

estado estable ante un cambio de la geometría o de la luz existente. Finalmente tampoco modela eventos como polarización o fluorescencia donde la luz cambia su longitud de onda al interactuar con las superficies. Sin embargo todos esos efectos pueden representarse extendiendo el modelo matemático.

2.3. Radiosidad

La técnica de Radiosidad es una alternativa a las técnicas de ray tracing donde se hace hincapié en cómo es el flujo de la luz a través del espacio cuando las superficies son únicamente lambertianas. Este método utiliza matrices para representar esos flujos, y las superficies de la escena se dividen en un conjunto finito de elementos llamados *parches*. La *radiosidad* es el valor de la potencia lumínica emitida y reflejada por unidad de superficie (W/m^2). A su vez, la *irradiancia* es el valor de la potencia lumínica recibida por unidad de superficie. Debido a que las superficies lambertianas (véase la Figura 2.2a) tienen un valor de la función *rho* independiente de la dirección incidente, la ecuación de Rendering (2.1) se puede transformar en la ecuación de radiosidad:

$$B(x) = E(x) + \rho(x) \int g(x, x')B(x')dA' \tag{2.2}$$

donde

- $B(x)$ es la radiosidad en el punto x
- $E(x)$ es la radiosidad emitida desde el punto x. Es 0 si el punto no es emisor de luz
- $\rho(x)$ es la reflectividad de x. Debido a que es una superficie lambertiana el valor es constante en la integral
- $g(x, x')$ es el término geométrico que depende de la distancia entre x y x' y si están o no ocluidos.

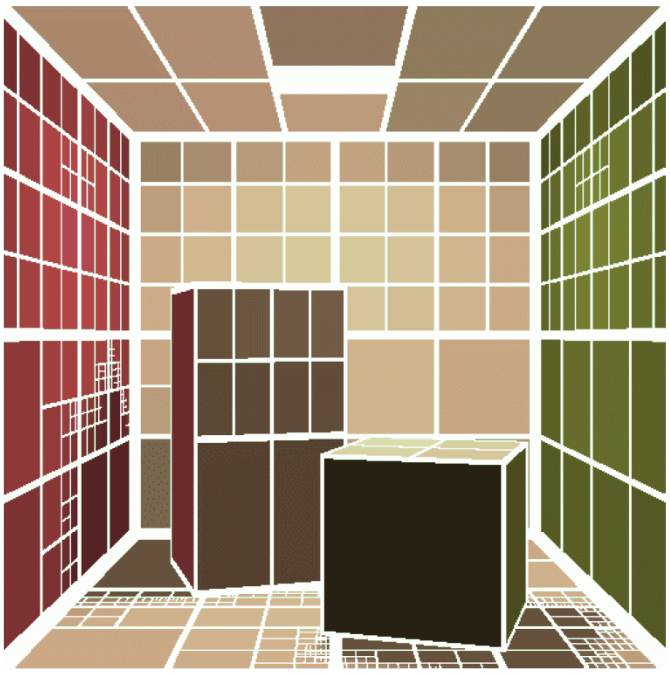


Figura 2.3: División de las superficies de una escena en parches (McCorquodale, 2001)

La ecuación de radiosidad se puede hacer discreta tomando una cantidad finita de parches. Véase la Figura 2.3 para un ejemplo donde se usan cuadrados de diferentes tamaños. Entonces la Ecuación 2.2 se transforma en:

$$B_i = E_i + R_i \sum\nolimits_j B_j F_{i,j} \tag{2.3}$$

donde $i \in \{1, 2, \ldots, n\}$, y n es la cantidad de parches de la escena. B_i representa la radiosidad del i-ésimo parche, que depende de su factor emitivo, representado en E_i y de la contribución de luz que llegue de los otros puntos de la escena. El factor emitivo va a ser distinto de 0 solamente en aquellos parches que representen a fuentes de luz. El segundo término de la Ecuación 2.3 computa la contribución de los demás parches en i. Este término depende de:

- El factor de reflectividad R_i que indica el porcentaje reflejado de la luz que llega al parche. Este valor está en el intervalo $[0, 1]$.
- La radiosidad B_j del resto de los parches
- Los factores de forma de la escena $F_{i,j}$ que son el análogo finito de $g(x, x')$ de la Ecuación 2.2. Describe qué fracción de la energía emitida por el parche i llega al parche j.

Si se considera la Ecuación 2.3 para todos los valores de i, se forma un sistema de ecuaciones que se puede expresar de manera matricial como:

$$B = E + \mathbf{RF}B \tag{2.4}$$

donde $\mathbf{R}$ es una matriz diagonal que representa la reflectividad de cada parche. $\mathbf{F}$ es la matriz de factores de forma que expresa cuánta luz puede ir de un parche al otro. E es el vector de emisión que expresa cuánta luz emite cada parche de la escena. B es un vector que indica la radiosidad en cada parche.

La matriz $\mathbf{F}$ contiene los valores de $F_{i,j}$, que dependen de la geometría de la escena. Tiene en cuenta la distancia entre los parches, el ángulo entre ellos y el área de las superficies. El factor de forma entre dos parches A_i y A_j se calcula como:

$$F_{ij} = \frac{V_{ij}}{A_i} \int_{A_i} \int_{A_j} \frac{\cos\theta_i \cos\theta_j}{\pi |r|^2} dA_i dA_j$$

donde θ_i y θ_j son los ángulos entre la normal de las superficies y la recta que une ambos parches. $V_{i,j}$ es 1 si los parches son visibles entre si y 0 en caso contrario. Finalmente, r es la distancia entre los parches. En la Figura 2.4 se muestra una representación gráfica de esto.

Cálculo de la radiosidad en una escena

Para renderizar una escena usando radiosidad, se necesita saber el valor del vector B de la Ecuación 2.4 cuyo valor es:

$$B = (\mathbf{I} - \mathbf{RF})^{-1} E \tag{2.5}$$

Teniendo en cuenta que $(\mathbf{I} - \mathbf{RF})^{-1}$ puede ser transformado usando series de Neumann en una suma de potencias de RF:

$$B = (\mathbf{I} - \mathbf{RF})^{-1} E \approx \left[\mathbf{I} + (\mathbf{RF}) + (\mathbf{RF})^2 + \cdots + (\mathbf{RF})^n\right] E$$

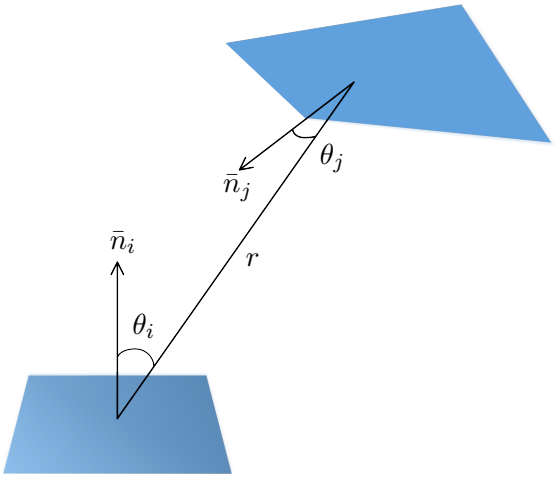
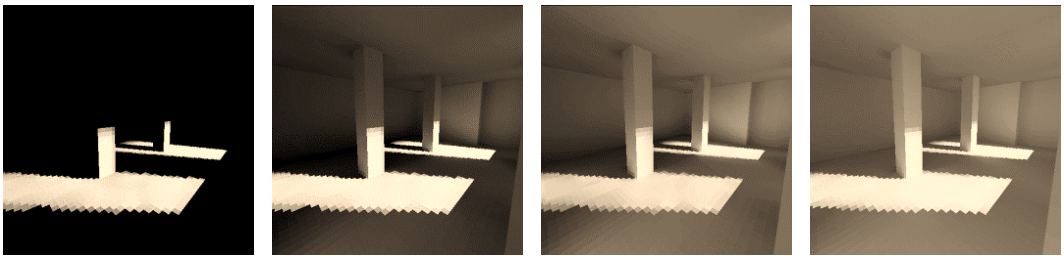


Figura 2.4: Elementos para la derivación del factor de forma

para un n suficientemente grande. El término $\mathbf{I}E = E$ representa la emisión de luz, mientras que $(\mathbf{RF})^i E$ hace referencia a la iluminación recibida en el i-ésimo rebote. Esto lleva a una forma iterativa de calcular la radiosidad de los parches, que se expresa como:

$$B^{(i+1)} = \mathbf{RF}B^{(i)} + E;\ B^{(0)} = E$$

Dependiendo de la escena, la iluminación causada por los rebotes termina no siendo suficientemente significativa como para seguir calculando $B^{(i)}$, a partir de cierto valor de i. En la Figura 2.5 se muestra la generación de una imagen usando distintos valores de n.



(a) 1 rebote (b) 2 rebotes (c) 3 rebotes (d) 16 rebotes

Figura 2.5: Renderización de una habitación con una ventana a la derecha, con distinta cantidad de rebotes (Elias, 2000)

.

Radiosidad para solución de problemas de optimización

En la Ecuación 2.5 el vector B se calcula haciendo una sucesión de sumas que dependen de $\mathbf{R}$ y $\mathbf{F}$. Si se quisiera hacer algún cambio en la geometría, como cambiar de lugar algún objeto de la escena, entonces se tendría que volver a calcular nuevamente B, dado que $\mathbf{F}$ depende de la geometría, pues si se cambia de lugar algún objeto, sus parches, y el valor de los factores de forma también variarán. De la misma forma, si se desea modificar la reflectividad de los parches, en ese caso B tendría que ser computado nuevamente. Esto es importante tenerlo en cuenta debido a que para resolver los ILPs, a menudo resulta necesario cambiar aspectos geométricos y de reflectividad de la escena, para evaluar una configuración específica. En el caso de usar técnicas de Radiosidad esto puede llegar a ser poco eficiente. Sin embargo, las técnicas de Radiosidad resultan útiles cuando sólo se modifica la emisión de las escenas, debido a que basta con cambiar los componentes del vector E de emisión.

2.4. Ray Tracing

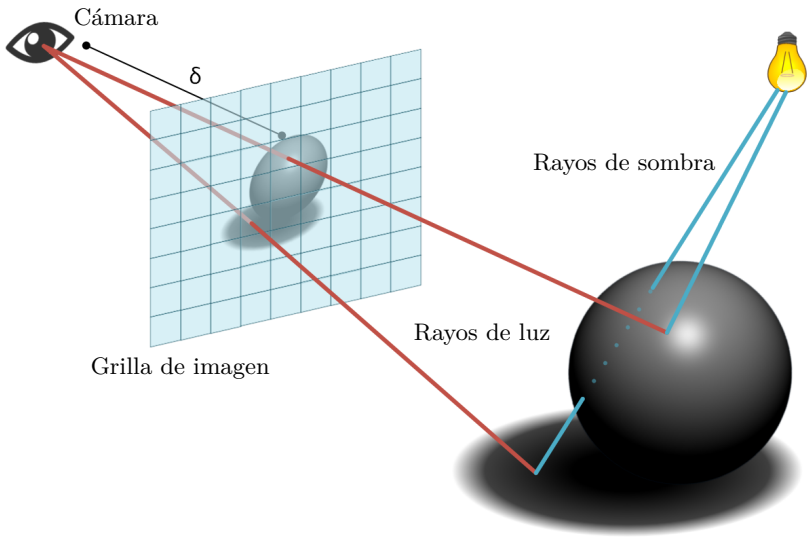


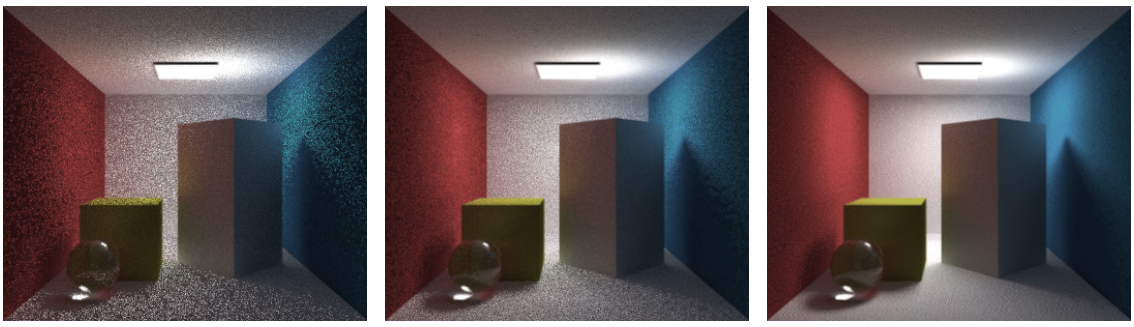
Figura 2.6: Elementos de un Ray Tracing

Ray Tracing, también denominado *traza de rayos* por su traducción al español, es una técnica para renderizar escenas tridimensionales. Esta técnica fue popularizada en computación gráfica por Whitted (1980). La imagen de una escena se genera simulando la forma en que los rayos de luz viajan de un objeto a otro hasta el observador. El proceso consta en lanzar los rayos desde el observador hasta los objetos de la escena y propagarlos hacia las fuentes luminosas. Sin embargo, en la naturaleza el efecto es inverso: la luz se origina en las luces y luego van hacia los objetos, hasta llegar al observador. Dado que el ojo del observador se considera un punto o a lo más una lente de cámara, el proceso puede llevar una cantidad de tiempo que es prohibitiva, debido a que sólo una pequeña fracción de los rayos que impactan en las superficies de la escena terminan llegando al observador. Por eso el proceso se hace de manera inversa: los rayos se generan tomando como punto de partida al ojo del observador. Se hace una traza del rayo para cada píxel de la pantalla. Se toma como referencia una grilla que está a una distancia corta (δ) del observador. El tamaño y la distribución de la grilla de puntos de referencia dependen directamente de las dimensiones de la imagen a generar. Cada píxel se genera trazando un rayo desde el origen, pasando por un punto de la grilla, correspondiente al píxel. Estos rayos se denominan rayos de luz. En una superficie lambertiana para hallar el color final se hace otra traza del rayo a cada una de las luces de la escena. Estos rayos se denominan rayos de sombra. El punto de intersección recibe contribución de luz para aquellas fuentes luminosas, cuyos rayos de sombra no estén obstruidos por algún objeto en la escena. En la Figura 2.6 se ilustran algunos de los elementos de un Ray Tracing.

2.5. Path Tracing

Los métodos de Ray Tracing basados en métodos de Montecarlo como el *Path Tracing* (Trazado de Caminos) apuntan a computar una aproximación sin sesgo de la iluminación global de una escena. Path Tracing se basa principalmente en estimar la radiosidad de una superficie haciendo una serie de traza de rayos por cada punto de la imagen. Esto se denomina *point sampling*, que consiste en evaluar la radiosidad de una superficie pasando varios rayos por cada píxel de la imagen a generar. A diferencia del Ray Tracing de Whitted estos métodos pueden manejar cualquier tipo de efectos que causan las superfi-

cies como cáusticas o *color bleeding* en las superficies. Pueden manejar cualquier tipo de reflexión indirecta. Además la calidad de la imagen final se puede controlar de manera arbitraria lanzando más o menos rayos. Estos métodos no requieren una cantidad grande de memoria para ejecutarse. Path Tracing es una extensión del Ray Tracing que hace posible computar cualquier iluminación indirecta lanzando un rayo "aleatorio" sobre el dominio de la integral: todas las direcciones donde un rayo puede caer en un píxel de la imagen generada. La función desconocida a integrar sería la distribución de la radiosidad. Si se usa una cantidad de rayos suficientemente grande se puede estimar el color del píxel usando métodos de Montecarlo. Para estimar la radiosidad de un rayo que cae en un punto de la escena, correspondiente a una superficie que no emite luz, se tiene que hacer un ray tracing partiendo desde ese punto para calcular la radiosidad indirecta. Un problema de este método es que si no se usa una cantidad de muestras por píxel suficientemente grande, se puede apreciar "ruido" en la imagen generada. Lanzar tantos rayos puede causar que la imagen tarde un tiempo prohibitivo, especialmente si la iluminación varía mucho en una superficie a integrar como en el caso de cáusticas o límites de superficies en sombra. Esto hace que usar Path Tracing no sea una opción eficiente para poder calcular la radiosidad de cada punto de la superficie. En la Figura 2.7 se muestran tres renderizaciones para una misma escena usando una cantidad diferente de rayos por píxel.



(a) 25 muestras por píxel (b) 125 muestras por píxel (c) 625 muestras por píxel

Figura 2.7: Path tracing con un número variable de muestras por píxel (Krishnamachari, 2006)

Otros métodos similares como Path Tracing bidireccional y Metrópolis Light Transport apuntan a hacer más rápida la renderización. En algunos tipos de escenas pueden ser más eficientes, pero en otros puede ser más beneficioso usar Path Tracing tradicional[1].

2.6. Photon Mapping

Photon mapping (mapeo de fotones) es una técnica de iluminación global desarrollada por Jensen (2001) que brinda una alternativa viable para escenas que no pueden ser manejadas por los métodos sin sesgo basados en métodos de Montecarlo. Es particularmente valiosa cuando se tienen reflexiones especulares junto a refracciones u otras superficies no lambertianas. El método combina la generación de rayos desde las fuentes luminosas, que se almacenan para luego usarlos al generar la imagen final.

El método de photon mapping tiene dos pasos, ilustrados en la Figura 2.8:

1. *Photon tracing* (Trazado de fotones): En el primer paso se lanzan rayos desde las fuentes luminosas simulando el recorrido de la luz representada como fotones. Al caer en una superficie, esos fotones pueden ser absorbidos o rebotar. En ambos se

[1]Se recomienda consultar Arvo y cols. (2001) para ver estos métodos en mayor detalle.

guarda su información en una estructura llamada *mapa de fotones*, almacenando datos acerca del fotón que incidió como: su posición en la superficie, la potencia incidente y la dirección de donde vino. Esta etapa se desarrolla en la Sección 2.6.1.

2. *Renderización*: En este paso se generan rayos de luz como en un ray tracing clásico, usando la información que está en el mapa de fotones para estimar la radiosidad. El color en cada píxel de la imagen final se calcula a partir de un ray tracing utilizando rayos de luz y sombra para hallar la iluminación directa. Para calcular la contribución de iluminación indirecta se buscan aquellos elementos del mapa de fotones que estén cercanos al punto de intersección del rayo de luz con la escena. Debido a que se necesita frecuentemente saber qué puntos de la superficie están más cercanos al punto de la escena donde impacta el rayo de luz, se utilizan estructuras especiales para tener mejor eficiencia, como árboles *kD* (Bentley, 1975) o grillas 3D uniformes. La etapa de rendering se trata en la Sección 2.6.3.

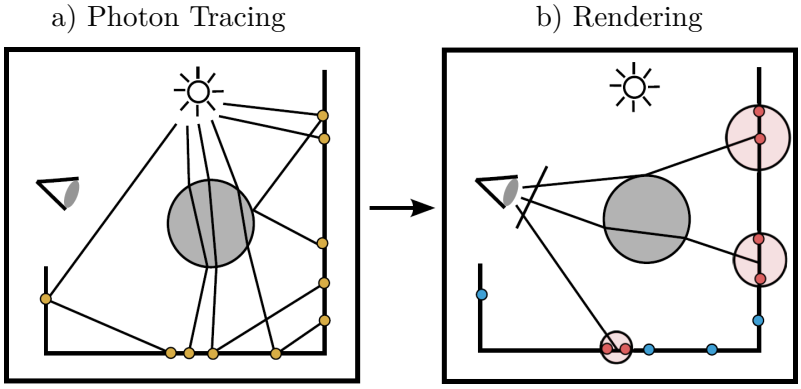


Figura 2.8: Esquema de las etapas de photon mapping (Pedersen, 2013)

2.6.1. Photon tracing

La idea detrás de realizar un photon tracing es generar una estructura que permita calcular la radiosidad indirecta de un punto, durante la etapa de rendering. Durante el photon tracing se toman diversas decisiones para dirimir si el fotón es absorbido o reflejado simulando el comportamiento real de un fotón al interactuar con una superficie. Se podría modelar que el fotón de manera de que pierda energía a medida que interactúa con las superficies, por lo que puede llegar a tener poca energía luego de una cantidad de rebotes. Esto hace no sea importante a la hora de calcular la radiosidad. Por eso se limita el número de rebotes que puede hacer un fotón a una cantidad n fija. Si se llega a ese número se termina el ray tracing de ese rayo.

Ruleta rusa

Para modelar la interacción entre el fotón y la superficie se usa un método llamado *ruleta rusa*(Arvo y Kirk, 1990). La ruleta rusa consiste en generar un número aleatorio ε tal que $0 \leq \varepsilon < 1$. Dependiendo del valor de ε se decide el comportamiento del fotón al interactuar con la superficie. Usar ruleta rusa ayuda a que se tenga una cantidad menor de elementos en el mapa de fotones, lo cual a su vez contribuye en ahorrar memoria, y hacer más eficiente la estimación de radiosidad. Como desventaja se aumenta la varianza de la radiosidad almacenada. Una alternativa para calcular las reflexiones difusas es simular todos los casos ponderando la energía del fotón según la probabilidad de que ocurra cada suceso. En este caso habría en el mapa de fotones muchos elementos “poco importantes”

que tienen poca energía, y que por lo tanto no aportan de manera significativa, al cálculo de la irradiancia en las superficies, usada para calcular el color en la imagen final.

Tratamiento de superficies

El comportamiento del fotón depende del tipo de superficie en el que impacte. A continuación se estudiará su comportamiento para algunos tipos comunes de superficies.

Superficies especulares Si la superficie es especular pura (no tiene un componente difuso), no se almacena el fotón debido a que la probabilidad de que este impacte en una dirección determinada es 0. Los impactos que no vengan de esa dirección específica se ignoran.

Superficies difusas Los fotones que impactan en una superficie lambertiana tienen una probabilidad de ser absorbidos que según la naturaleza del material, expresada generalmente mediante un coeficiente difuso de absorción K_d, que a su vez varía según el color de la superficie. $K_d = (K_{d_r}, K_{d_g}, K_{d_b})$ donde sus componentes son los colores RGB(acrónimo de Rojo, Verde y Azul) del material. Cada uno de los valores del vector están en el intervalo $[0, 1]$. Para decidir si el fotón es absorbido o reflejado se usa el método de ruleta rusa. Siendo ε el valor aleatorio generado, si $\varepsilon \leq \rho_d$ el fotón es reflejado y en caso contrario se absorbe y termina la etapa de photon tracing para ese fotón. La variable ρ_d representa la probabilidad de que el fotón sea reflejado. Un valor posible de ρ_d, es el que se usa en la implementación de este trabajo:

$$\rho_d = \frac{K_{d_r} + K_{d_g} + K_{d_b}}{3}$$

En Jarosz, Jensen, y Donner (2008) se propone usar otra fórmula para calcular ρ_d:

$$\rho_d = \frac{\text{máx}(K_{d_r}P_r, K_{d_g}P_g, K_{d_b}P_b)}{\text{máx}(P_r, P_g, P_b)}$$

siendo $P = (P_r, P_g, P_b)$ la potencia del fotón incidente en la superficie. En cualquiera de los dos casos, la lógica para modelar la interacción se ilustra en el Algoritmo 2.1.

Algoritmo 2.1 Esquema de interacción del fotón con una superficie difusa

```
ε = random([0, 1])
Calcular ρ_d
Almacenar el fotón en el mapa de fotones
if ε ≤ ρ_d then
  Seguir el photon tracing
end if
```

En el caso de la reflexión la potencia del fotón reflejado se tiene que ajustar para tener en cuenta la probabilidad de no ser absorbido.

$$P_{refl_r} = \frac{P_r K_{d_r}}{\rho_d} \qquad P_{refl_g} = \frac{P_g K_{d_g}}{\rho_d} \qquad P_{refl_b} = \frac{P_b K_{d_b}}{\rho_d}$$

donde P es la potencia del fotón incidente y P_{refl} representa la potencia el fotón reflejado.

Superficies refractivas Las superficies puramente refractivas no almacenan fotones, en caso contrario se sigue haciendo el photon tracing modificando la dirección del fotón de acuerdo al coeficiente de refracción K_r de la superficie.

Superficies con combinación de componentes de transmisión, reflectivos y especulares Para este tipo de superficies se puede usar el método de ruleta rusa para calcular si se hace una reflexión especular, difusa o absorción de la misma manera que para superficies difusas puras. El criterio es similar, solo que se usan más casos para decidir qué sucede con el fotón. En el Algoritmo 2.2 se ilustra la lógica usada en este caso. Suponiendo que una superficie tiene un material tal que la probabilidad de que un rayo sea reflejado de manera especular es ρ_s, de que sea refractado es ρ_r y de que sufra una reflexión difusa sea ρ_d. Se debe cumplir que $\rho_s + \rho_r + \rho_d \leq 1$. La Figura 2.9 ilustra el comportamiento según el valor de ε.

Algoritmo 2.2 Interacción en una superficie con componentes refractivo, reflectivo y difuso

```
ε = random([0, 1])
if ε ≤ ρs then
   Calcular reflexión especular
   Seguir photon tracing
else if ε ≤ ρs + ρr then
   Calcular refracción
   Seguir photon tracing
else if ε ≤ ρs + ρr + ρd then
   Calcular reflexión difusa
   Almacenar fotón en el mapa de fotones
   Seguir photon tracing
else
   Absorción. Termina photon tracing
end if
```

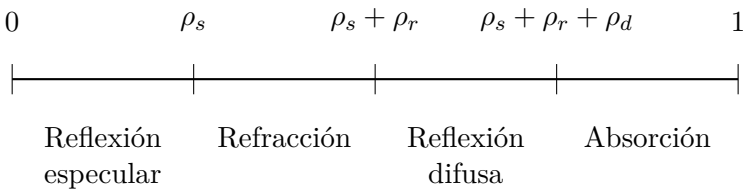


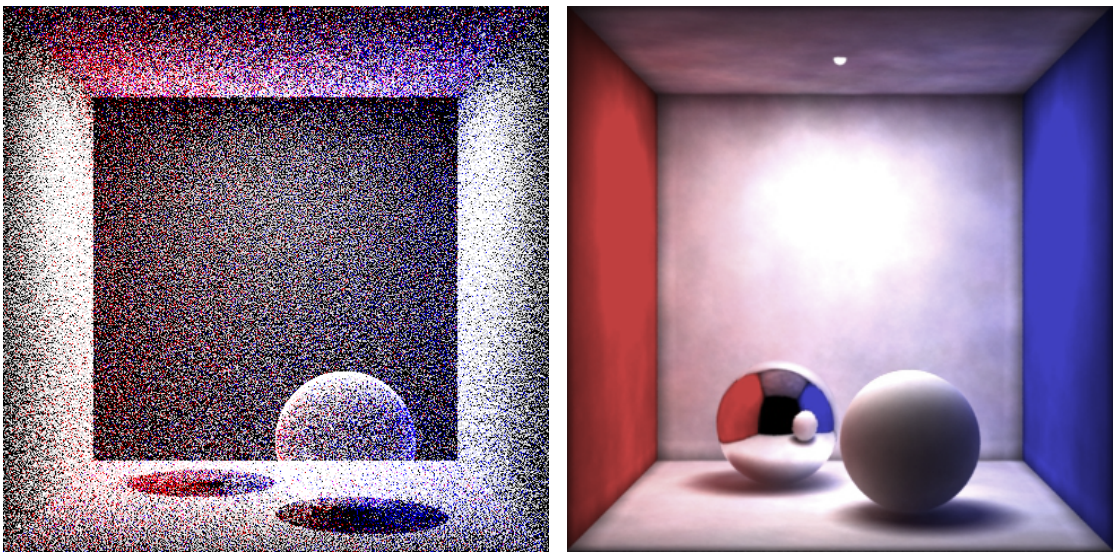
Figura 2.9: Diferentes comportamientos del fotón según el valor de ε

2.6.2. Mapa de fotones

El mapa de fotones es la estructura para almacenar los fotones que se usará luego en la etapa de rendering. Una propiedad interesante del mapa es que la información que se guarda no está asociada a priori a ninguna superficie en particular y es independiente de la geometría de la escena. Los fotones que vienen directamente de una fuente luminosa no se almacenan en el mapa de fotones, debido a que esa información se calcula en la etapa de Rendering, cuando se generan los rayos de sombra (ver Figura 2.6) que van desde las superficies visibles hacia las fuentes luminosas (Jensen, 2001).

Durante la etapa de renderización es necesario identificar aquellos fotones más cercanos a cada punto intersecado de las superficies de la escena. Por esta razón la estructura usada tiene que ser eficiente para ese cometido. Además el mapa de fotones tiene que ser compacto en memoria, debido a que la calidad de la imagen final es proporcional a la cantidad de fotones utilizados. De esto último se desprende que una limitación del photon mapping tradicional, es la cantidad de memoria de acceso rápido (memoria RAM o memoria del

GPU) disponible en el dispositivo. Como consecuencia, la calidad de la imagen final está atada a la memoria del computador que ejecute el algoritmo. Las estructuras de datos más usadas para guardar los fotones son los árboles kD y las grillas uniformes. En el Apéndice A se discute por qué la segunda opción es más viable para implementar photon mapping utilizando la GPU. En la Figura 2.10 se muestra la distribución espacial de los fotones en mapa de fotones y la imagen generada a partir de éste, para la escena de *Cornell Box*.



(a) Mapa de fotones (b) Imagen generada

Figura 2.10: Visualización del mapa de fotones. La esfera del fondo es de un material espejado y la del frente es lambertiana. Los puntos en el mapa de fotones representan la potencia incidente. (Pixar, 2011)

2.6.3. Rendering

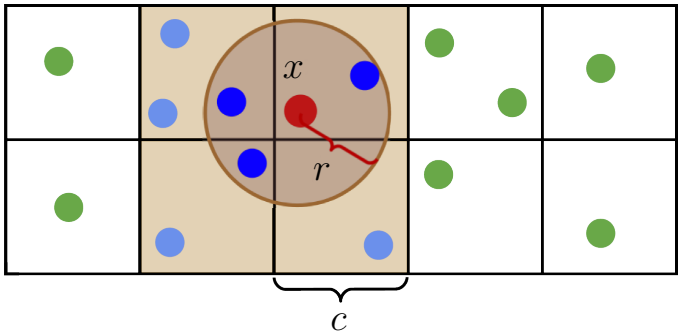


Figura 2.11: Calculo de la radiosidad usando una grilla uniforme 2D. Adaptado de Pedersen (2013)

Para estimar la radiosidad en un punto x de la una superficie de escena se necesita ubicar los n que estén en una esfera de radio R con centro en x. La idea es estimar la irradiancia usando esos fotones. Si la cantidad de fotones es suficientemente grande y la esfera es suficientemente pequeña se podrá tener una buena estimación de la radiosidad $L(x \to \psi)$. Su base matemática consiste en combinar la ecuación de Rendering (2.1) con la definición de radiosidad: El flujo de luz (energía por unidad de tiempo) por unidad de

área, por ángulo sólido ($\frac{d^2\Phi}{dAd\omega\cos\theta}$). [2]

$$\begin{aligned} L(x \to \psi) &= \int_\Omega f_r(x,\psi,\omega) L(x \leftarrow \omega)\cos\theta d\omega \\ &= \int_\Omega f_r(x,\psi \leftrightarrow \omega)\frac{d^2\Phi}{dAd\omega\cos\theta}\cos\theta d\omega \\ &\approx \sum_{p=1}^{n} f_r(x,\psi \leftrightarrow \omega_p)\delta\frac{\Phi_p}{\pi r^2} \end{aligned} \tag{2.6}$$

Esto es correcto si esa esfera no contiene fotones de otras superficies, o aristas que separen caras de un mismo poliedro. En este caso la estimación puede tener un sesgo agregado. Entonces utilizando la Ecuación 2.6 para computar la estimación de la radiosidad de un punto x, se debe buscar aquellos fotones que estén en la esfera de centro x y radio r. Esos fotones van a estar en las celdas que se cruzan con la esfera, por lo que para encontrarlos hay que:

1. Buscar el conjunto de celdas que se cortan con la esfera de centro x y radio r
2. Filtrar aquellos fotones p que cumplan que $|p-x| \leq r$. En la Figura 2.11 puede verse el caso para una grilla en $\mathbb{R}^2$.

2.7. CUDA

CUDA (Compute Unified Device Architecture) (NVIDIA Corporation, 2015) es una plataforma de programación paralela y una *API*(siglas de *Application Programming Interface*) que permite utilizar la GPU como una unidad de procesamiento de uso general. Debido a que las GPUs tienen una eficacia mayor respecto a las CPU del mismo período en términos operaciones teóricas por segundo. Véase la Figura 2.12 para más detalles. En los hechos las GPUs son más rápidas para realizar tareas que pueden paralelizarse en miles de hilos de ejecución. NVIDIA llama al modelo de ejecución SIMT (*Single Instruction Multiple Threads* en inglés). En Kreinin (2011) se describe en detalle cuáles son las ventajas y desventajas de esa arquitectura. Lo cierto es que en photon mapping se hace uso de raytracing y esa operación es altamente paralelizable si se tiene en cuenta que cada rayo puede procesarse de manera independiente. Se guarda la información acerca de la escena en estructuras de datos especiales y se guarda la información de los rayos procesados en *buffers* de salida de la unidad de procesamiento gráfico. Un lado negativo en el rendimiento por uso de la GPU es que hay que transferir datos desde la memoria RAM del sistema hacia la memoria de la GPU para poder realizar tareas, y luego llevarlas a la memoria principal para mostrar los resultados al usuario. Es importante minimizar las ocasiones donde se traspasan datos entre ambas memorias RAM porque el traspaso puede llevar mucho tiempo. En este trabajo se usa CUDA para realizar operaciones que incluyen el armado de las estructuras del mapa de fotones (árboles kD, grillas uniformes), que se usan para generar la imagen final.

Thrust

Thrust (Bell y Hoberock, 2011) es una biblioteca de plantillas de C++ pensada para CUDA y basada en la STL[3]. Thrust permite realizar operaciones paralelas de forma

[2] Se recomienda consultar (Jensen, 2001) para más información acerca de la definición de Radiosidad.

[3] *Standard Template Library* (por sus siglas en inglés) de C++. Es la biblioteca de algoritmos genéricos y estructura de datos que provee el lenguaje

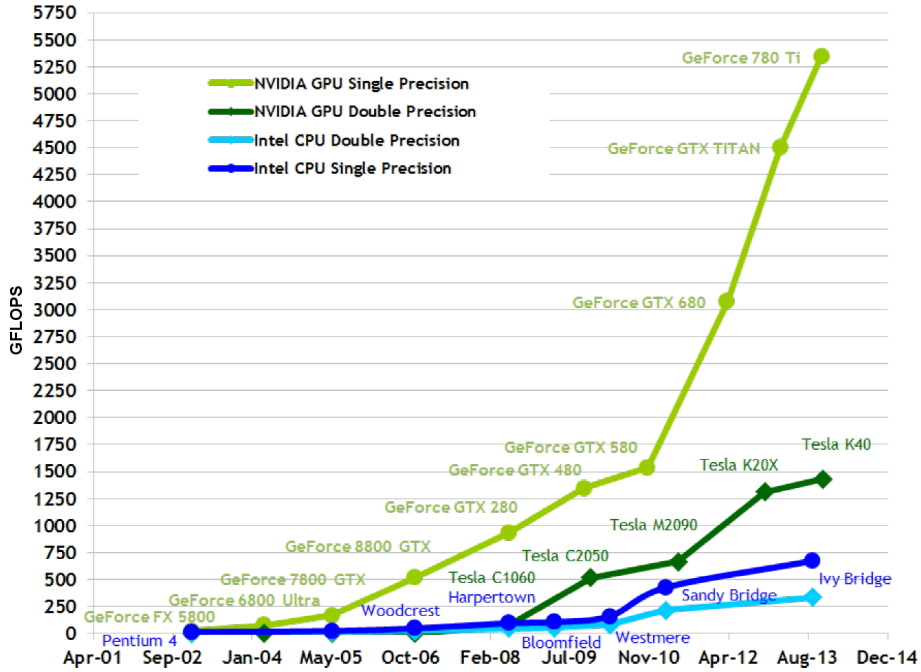


Figura 2.12: Comparación de rendimiento entre diferentes arquitecturas de GPUs y CPUs (NVIDIA Corporation, 2015).

eficiente utilizando los recursos de la GPU y usando las estructuras de memoria que ya provee CUDA. Esencialmente ofrece algoritmos y estructuras de datos como la STL pero pensado para usarse con *buffers* y matrices de CUDA. Ofrece algoritmos de ordenación, transformación, map/reduce y estructuras de datos basados en vectores. Es simple integrar Thrust con programas CUDA. En este trabajo se utiliza esta biblioteca para recorrer las estructuras del mapa de fotones y recabar información acerca de la radiosidad de cada objeto.

2.8. OptiX

Por su parte, OptiX (NVIDIA Corporation, 2014) es un framework para construir aplicaciones basadas en ray tracing. Funciona junto a CUDA y tiene dos componentes principales. En primer lugar se compone de una API que define estructuras, tipos de datos y funciones para realizar el raytracing en una escena fácilmente. En segundo lugar incluye un entorno de programación basado en CUDA para poder escribir programas que utilicen dicha API. Ambas partes dependen entre sí. OptiX es una opción natural debido al uso de photon mapping y el algoritmo en sí implica inherentemente utilizar ray tracing para varias etapas. Durante el proyecto intentó utilizarse la última versión de OptiX disponible (3.8.0), que es compatible con CUDA 7.0.

2.8.1. API

OptiX está partido en dos mitades: el código del anfitrión y los programas de dispositivo. El primero se ejecuta en CPU y el otro en GPU. Los principales elementos son

- Contexto: Representa una instancia de OptiX. Mantiene el resto de los objetos que maneja la biblioteca.
- Programa: Es una función en CUDA que es compilada a un código intermedio. El comportamiento de un contexto de OptiX se forma definiendo programas. Por

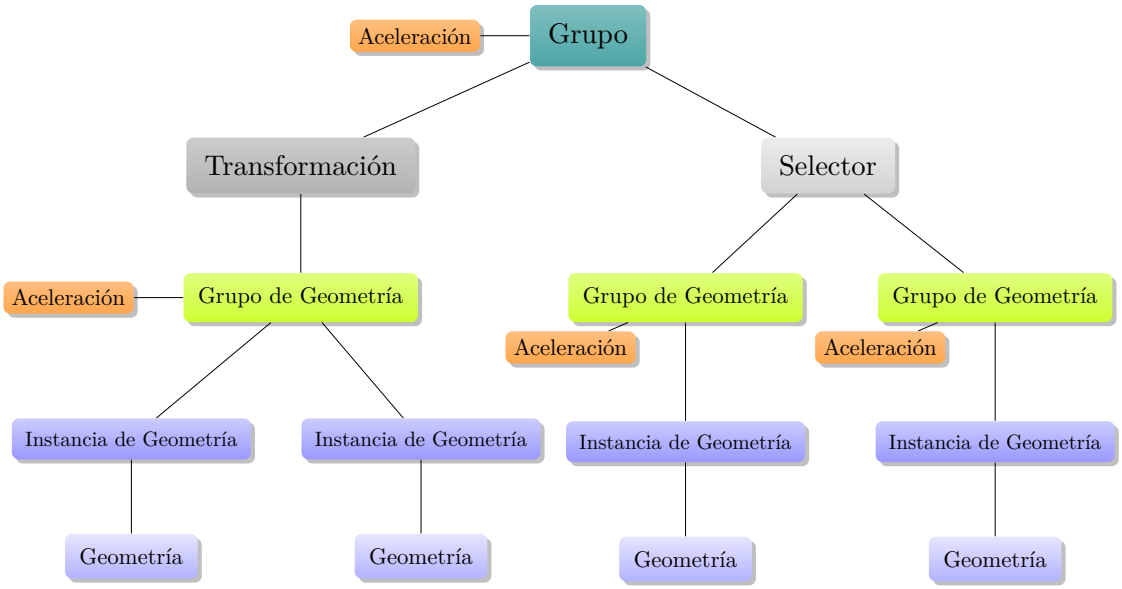


Figura 2.13: Un grafo de ejemplo de una escena en OptiX. Adaptado de NVIDIA Corporation (2014)

ejemplo, los programas se usan para personalizar el comportamiento del rayo y para definir si el rayo impacta con una primitiva geométrica dada de la escena.

- VARIABLE: Son Nombres que se asocian a otros objetos para pasar datos desde los programas de OptiX y el código que ejecuta en la CPU. Las variables pueden estar asociadas tanto al contexto como a otros objetos. Las variables en definitiva son *atributos* de los objetos de OptiX.

- BUFFER: Es un arreglo multi dimensional. Se usa como variable para pasar datos entre la GPU y la CPU. El mapa de fotones, la imagen final y muchos datos que se manejan en OppositeRenderer se tratan como *buffers*.

- GEOMETRÍA: Representa una primitiva con la cual el rayo puede interceptar. Pueden ser esferas, triángulos o cualquier objeto que se quiera. Una geometría se puede pensar como una Clase de un lenguaje orientado a objetos. Tiene una función (programa) que define si el rayo se interseca con la primitiva y un conjunto de variables que definen su posición en la escena y otros datos. Por ejemplo las coordenadas de los vértices del triángulo, o el centro y el radio de la esfera.

- MATERIAL: Es la combinación de un programa y un conjunto de variables que se aplican cuando un rayo interseca con la geometría. Un ejemplo de material puede ser un material lambertiano. Las variables son los coeficientes K_d del material. Por ejemplo en photon mapping, el programa del material guarda el fotón en el mapa de fotones, y define el comportamiento del fotón en la etapa de photon tracing. Se puede tener varios *materiales* para distintas etapas del photon mapping.

- INSTANCIA DE GEOMETRÍA: Una asociación entre un Material y una Geometría. e.g. Un triángulo de color rojo.

- GRUPO: En el grafo de una escena de OptiX es un nodo que puede tener distintos hijos. Un grupo en general tiene una estructura de aceleración. Véase la Figura 2.13 para un ejemplo de un grafo de la escena.

- GRUPO DE GEOMETRÍA: Es un conjunto de instancia de geometrías.

- Transformación: Implementa una forma eficiente de aplicar una transformación lineal a los hijos que tenga en un Grupo de Geometría. Es muy útil en este trabajo para *mover* instancias de geometría como los objetos de la escena sin la necesidad de volver a rearmar completamente las estructuras de aceleración. No es obligatorio usarlas en OptiX aunque puede llegar a ser poco eficiente no hacerlo.
- Selector: Un nodo que tiene un programa que permite seleccionar si un rayo va a interceptar un nodo u otro. No se usa en este trabajo. Puede utilizarse para implementar algunos efectos de tipo *LoD*[4] teniendo varios nodos hijos con diferentes cantidades de polígonos.
- Estructura de Aceleración: Una estructura que se asigna a un nodo del grafo de la escena. Se estudian en más detalle en la Sección 2.8.2.

2.8.2. Estructuras de aceleración

Una gran ventaja de OptiX es que permite mantener lo que se llaman *Estructuras de aceleración* (NVIDIA Corporation, 2014). Estas estructuras permiten organizar los objetos de la escena de forma eficiente para que las búsquedas de elementos sean rápidas, específicamente cuando se hace las intersecciones de rayos con la geometría. Algunas estructuras de aceleración son árboles kD y variantes de BVH[5]. Algunas estructuras interesantes que se probaron durante este trabajo son:

- *SBVH*: Una jerarquía de volúmenes acotantes de alta calidad pero con tiempo de armado lento. Es útil cuando la geometría es estática.
- *TRBVH*: Una variante de BVH que es ligeramente más rápido que el SBVH debido a que se construye en la GPU. Puede tener problemas si la GPU se queda sin memoria disponible. Esto es algo que pasa frecuentemente en el photon mapping ya que se utiliza la memoria para el mapa de fotones.
- *TriangleKdTree*: Un árbol kD de alta calidad pero muy lento en términos de rendimiento si se lo compara con SBVH. El tiempo de armado y el uso de memoria es más grande para un árbol kD.

Cada una de las estructuras ofrece distintos tiempos de armado y de consulta dentro de ellas. Una buena elección de las estructuras influye en el tiempo de ejecución del photon mapping:

$$T_{algoritmo} = T_{armado} + T_{consulta}$$

donde T_{armado} es el tiempo que toma armarse la estructura elegida, y $T_{consulta}$ es el tiempo que toma procesar los rayos generados usando la estructura de aceleración. Si se generan pocos rayos el tiempo de armado puede dominar la ecuación y puede ser beneficioso usar una estructura fácil de armar aunque no sea eficiente a la hora de calcular intersecciones de rayos con la geometría. Stich, Friedrich, y Dietrich (2009) discuten la eficiencia de diferentes tipos de BVHs en cuánto a la eficiencia.

2.9. OppositeRenderer

OppositeRenderer es un programa realizado Stian Pedersen (2013), que implementa photon mapping Progresivo (véase el Apéndice B) en la GPU utilizando OptiX como motor de Ray Tracing. Está escrito en C++ y CUDA. El programa implementa diferentes

[4] *Level of Detail* por sus siglas en inglés
[5] Jerarquía de volúmenes acotantes o BVH (*Bounding Volume Hierarchy* por sus siglas en inglés)

maneras para almacenar el mapa de fotones como árboles kD, grillas uniformes y diccionarios estocásticos. Estas estructuras se describen en detalle en el Apéndice A. También posee la capacidad de distribuir el proceso de desarrollo en diferentes GPUs que estén en una misma red local. Un programa actúa de servidor y divide el photon mapping progresivo en iteraciones que se ejecutan de manera independiente en distintos clientes. El código se divide en tres partes.

- Biblioteca de renderización: Se encarga de implementar el photon mapping.
- Cliente y servidor de renderización distribuida: Son dos componentes que se encargan de implementar la generación de imágenes por photon mapping de manera distribuida.
- Interfaz gráfica: Es un cliente gráfico que permite cargar una escena y visualizar su representación gráfica luego de cada iteración del photon mapping progresivo.

2.10. Heurísticas de Optimización

En este trabajo se busca resolver problemas de optimización, para ello fue necesario elegir un método o herramienta que permita solucionar esos problemas. En las secciones siguientes se detallan los tipos más comunes de problemas de programación matemática: problemas de programación lineal y problemas de programación *no* lineal (Luenberger y Ye, 2008). Por último se describe el método usado en este trabajo: VNS.

2.10.1. Tipos de problemas de optimización

Los problemas de optimización pueden clasificarse de distintas maneras. Usualmente se lo hace según la forma función de optimización (si tiene una solamente) y según el tipo de sus restricciones. En la Figura 2.14 se puede ver una clasificación de ellos, la clasificación no es excluyente. Por ejemplo, puede haber problemas de programación sin restricciones que se puedan atacar con los mismos métodos que los problemas con restricciones.

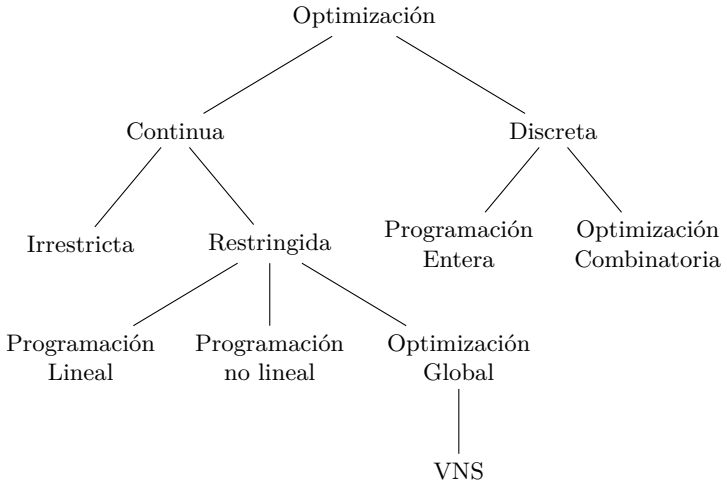


Figura 2.14: Clasificación parcial de problemas de optimización. Adaptado de Universidad de Wisconsin (2015).

Problemas de programación lineal

Los problemas de programación lineal constan en optimizar una función lineal en base a las variables, donde también son lineales las restricciones: son ecuaciones o bien inecuaciones lineales en las variables. Es la forma natural de atacar una gran cantidad de problemas de optimización, y la teoría es muy rica en esa área. En un problema de este tipo la función de optimización puede definirse como Wx donde x es un vector columna de $\mathbb{R}^n$ y W es el vector fila de pesos. Las ecuaciones o inecuaciones que definen las restricciones se expresan de manera similar. Si se tiene una igualdad la ecuación es del tipo: $A^{(i)}x = b^{(i)}$, donde $A^{(i)}$ es un vector fila con los coeficientes de la ecuación. Del mismo modo si se tiene una inecuación se tiene algo similar $A^{(i)}x \leq b^{(i)}$. Generalizando se puede anotar una inecuación como una igualdad si se agrega una variable y a la ecuación: $A^{(i)}x + c^{(i)}y = b^{(i)}$. Si es una inecuación $c \neq 0$, y si es una igualdad $c = 0$. De esta forma se tiene un sistema de m ecuaciones, que se puede agrupar en una matriz A donde cada una de sus filas son los coeficientes del sistema $A^{(i)}$. La matriz A es de tamaño $(m \times n)$. $c^{(i)}$ se deja en un término aparte y el sistema de ecuaciones se compacta en la notación $Ax + Cy = B$. Sintetizando, un problema de programación lineal puede anotarse de la siguiente manera:

$$\begin{aligned} \text{máx} \quad & W^t x \\ s.a. \quad & Ax + Cy = B \\ & y \geq 0 \end{aligned}$$

Problemas de programación no lineal

Los problemas de programación no lineal son una forma más genérica de formulación del descrito anteriormente. En estos problemas, las funciones de optimización y las restricciones tienen una forma no necesariamente lineal, por lo cual se podría ver a este tipo de problemas como una generalización del caso lineal:

$$\begin{aligned} \text{máx} \quad & f(\mathbf{x}) \\ s.a. \quad & h_i(\mathbf{x}) = 0, \quad i \in \{1, 2, \ldots, m\} \\ & g_j(\mathbf{x}) \geq 0, \quad j \in \{1, 2, \ldots, r\} \\ & \mathbf{x} \in S \end{aligned}$$

Figura 2.15: Problema de programación no lineal

En esta formulación $\mathbf{x}$ es un vector de incógnitas: $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, las funciones f, h_i, y g_j con $i \in \{1, 2, \ldots, m\}$, $j \in \{1, 2, \ldots, r\}$ son $S \to \mathbb{R}$. En general $S \subseteq \mathbb{R}^n$, debido a que puede modelarse las restricciones que dan forma a S por medio de (in)ecuaciones. Puede darse que el dominio S sea finito. En ese caso se dice que el problema de optimización es de carácter *combinatorio*.

2.10.2. Metaheurísticas de optimización

Muchos de los problemas de optimización no lineal genéricos no pueden resolverse haciendo un análisis de la función f de optimización, debido a que su forma no es conocida. Por eso es beneficioso hacer una búsqueda en el dominio S de forma inteligente. En este sentido surgen las metaheurísticas de optimización, que son marcos de referencia para generar algoritmos de optimización específicos a un problema. Según Osman y Laporte (1996), una metaheurística se define como un proceso iterativo de generación, que

guía a una heurística subordinada combinando inteligentemente diferentes conceptos, para explorar y explotar el espacio de búsqueda, aplicando estrategias que son usadas para estructurar la información, y encontrar eficazmente soluciones óptimas. Según Blum y Roli (2003), las metaheurísticas se caracterizan por:

- Ser algoritmos aproximados y usualmente no deterministas.
- Aplicar técnicas para evitar quedar atrapados en ciertas áreas del espacio de búsqueda.
- No ser específicas a un problema.
- Usar conocimiento del dominio en forma de heurísticas, que a su vez son controladas por estrategias de más alto nivel.

2.10.3. Variable Neighbourhood Search

El método Variable Neighbourhood Search (*VNS*) es una metaheurística para resolver problemas de optimización global, introducido por Mladenović y Hansen (1997). Un problema de optimización global se define como la maximización de cierta función continua:

$$\max f(x) \quad s.a.\ x \in X$$

donde X es el conjunto de soluciones factibles. Se define que una solución x^* es óptima si

$$f(x^*) \geq f(x) \quad \forall x \in X$$

Como se explicará en el Capítulo 3, en el problema de optimización de iluminación inversa f es una función en $\mathbb{R}^n$ y toma valores reales. El dominio de f son las diferentes posiciones de un objeto, colores de una superficie, posiciones de una fuente luminosa o la combinación de estos elementos. El valor de f es la radiosidad de cierta superficie de una escena. Se puede utilizar VNS para problemas combinatorios. Debido a la naturaleza de los problemas que se pretenden resolver, en esta sección se introduce a VNS para funciones reales.

Para el caso continuo es bueno admitir cierto grado de tolerancia en la solución x^*, por lo que es suficiente establecer que $x^* \in X$ es óptimo si

$$\frac{f(x) - f(x^*)}{f(x^*)} < \varepsilon, \quad \forall x \in X$$

donde ε es la tolerancia: un número real pequeño y positivo.

VNS funciona buscando una dirección de ascenso para encontrar un máximo local y luego escapar de los picos que tienen a esos máximos para encontrar un máximo global. Para esto se usa el concepto de *vecindario* de una solución x. Un vecindario $\mathcal{N}(x)$ es un conjunto de soluciones x' que están "cerca" de la solución x. Por ejemplo si x representa la posición de un objeto en la escena, su vecindario se compone de aquellas posiciones *factibles* que están a una distancia menor que r, y donde r es un real constante. Por ejemplo en un problema donde se busca encontrar la mejor posición de la luz en un techo para maximizar la luminosidad de una mesa, r puede ser el radio de una circunferencia centrada en la posición actual de la luz. VNS explota ciertos hechos que en general cumplen las funciones continuas o con pocas discontinuidades, definidas en $\mathbb{R}^n$.

1. Un máximo local de un vecindario con centro x no es necesariamente el mismo que para otro vecindario del mismo centro.
2. Un máximo global es un máximo local para todos los vecindarios del mismo centro.

3. Para muchos problemas los máximos locales para uno o varios vecindarios del mismo centro están relativamente cerca entre sí.

VNS es muy simple de implementar para los problemas de ILP debido a que no requiere de muchos parámetros para funcionar. El desafío está en definir correctamente cuál es la estructura de un vecindario para un conjunto de variables de optimización. Además, como los problemas de optimización de iluminación son en general contínuos o con pocas discontinuidades, se cumplen los tres lemas definidos anteriormente.

Heurística de primera mejora

Algoritmo 2.3 Aplicación de la heurística de primera mejora

```
buscarMejora(x):
x_mejor ← x
repeat
  hayMejora ← false
  while No se cumpla condición de terminación do
    Se genera x_candidato ∈ N(x)
    if f(x_mejor) < f(x_candidato) then
      x_mejor ← x_candidato
      hayMejora ← true
      break
    end if
  end while
until not hayMejora
return x_mejor
```

VNS funciona en dos pasos. En el primero de ellos se busca un máximo local buscando el primera solución x' que sea mejor que la solución x actual. Esta heurística se denomina *de primera mejora*. Cuando se llega a un máximo local este algoritmo se detiene.

El Algoritmo 2.3 trata de recorrer todo el vecindario $\mathcal{N}(x)$ para encontrar un valor mejor. Si se encuentra un mejor *máximo* se cambia el centro del vecindario y se sigue nuevamente hasta agotar esa estructura. Vale la pena aclarar que esa estructura se genera estocásticamente dado que el dominio de soluciones es infinito y no numerable. Solo se recorren algunos elementos del vecindario para determinar que está agotado y se recorrió completamente. Si el vecindario es infinito la condición de terminación puede consistir "en hacer I intentos", donde I es un entero fijo. En un problema de dominio finito se podría recorrer todos los elementos del vecindario si se quiere, y la condición de terminación podría ser "Recorrer el vecindario $\mathcal{N}(x)$ completamente".

VNS Básico

Teniendo en cuenta el lema 1 de la heurística cuando se aplica el algoritmo de primera mejora, se encuentra un máximo local. Para salir del pico y encontrar otro máximo, es necesario volver a explorar otros vecindarios más grandes, pero con el mismo centro. Esto lleva a considerar varios vecindarios para buscar un máximo global. Siendo $\mathcal{N}_k$ con $k = (1, \ldots, k_{max})$ un conjunto finito de vecindarios anidados. $\mathcal{N}_k(x)$ es el k-ésimo vecindario con centro en x. Se considera que $X \subseteq \mathcal{N}_{k_{max}}(x),\ \forall x \in X$. Una solución x es óptima global si es óptima local en k_{max}. En el Algoritmo 2.4 se ilustra la heurística de VNS básico. La condición de terminación puede ser un tiempo límite, cantidad de vecindarios explorados, o la calidad de la solución. Se puede extender este algoritmo utilizando la técnica de *shaking*

que consiste en elegir un vecino aleatorio cercano al centro dentro del vecindario, antes de realizar la primera mejora, para evitar recorrer varias veces los mismos puntos.

Algoritmo 2.4 VNS Básico

```
VNS(x):
while No se haya alcanzado la condición de terminación do
  k ← 1
  repeat
    y ← buscarMejora(x)
    if f(y) > f(x) then
      y ← x
      k ← 1
    else
      k ← k + 1
    end if
  until k > k_max
end while
return x
```

3

Problemas Inversos de Iluminación

Un problema inverso de iluminación -*Inverse Lighting Problem* (ILP)- es un problema donde se debe determinar una configuración válida de un conjunto de parámetros para lograr así la iluminación deseada y se suele resolver con técnicas de optimización. En general se busca optimizar la luminosidad de cierta superficie sujeto a ciertas restricciones en las luces o la colocación de objetos en la escena. Por ejemplo: ¿cuál es la mejor posición en el techo de una fuente luminosa de forma tal de maximizar la cantidad de luz recibida en una mesa? Problemas similares pueden plantearse con aberturas, superficies reflectivas, o con la geometría. Matemáticamente el problema se expresa como:

$$\text{máx}\, f(x) \;\; s.a. \;\; x \in S; \;\; S \subseteq X \tag{3.1}$$

donde $f : X \to \mathbb{R}$ es la función objetivo, y S es el conjunto de soluciones factibles, que definen las condiciones que debe cumplir la solución para que sea aceptable como candidata. El conjunto de soluciones factibles refieren a la ubicación de las fuentes luminosas (techo), restricciones para la luz recibida, etc. Se verá luego que el espacio de soluciones X es isomorfo a un subconjunto de $\mathbb{R}^n$ cuya dimensionalidad depende de la cantidad y el tipo de variables de optimización. En las secciones que siguen se explicará qué aspectos de los problemas de optimización se atacaron durante este trabajo, y cómo se adaptó VNS para poder lidiar con la naturaleza de un problema no determinista debido al uso de photon mapping como técnica para evaluar la función objetivo.

Usar photon mapping tiene algunos beneficios en comparación con la técnica de Radiosidad, principalmente porque es más eficaz para cambiar aspectos relacionados con la geometría. En photon mapping mover un objeto es computacionalmente más simple que usando Radiosidad, porque en este último caso es necesario volver a calcular la matriz de radiosidad $(\mathbf{I} - \mathbf{RF})^{-1}$ de la Ecuación 2.5.

3.1. Intenciones de Iluminación

Las intenciones de iluminación son los objetivos y restricciones que los usuarios quieren conseguir en el proceso de optimización. Dada una escena, existe un conjunto de variables a modificar a través de los cuales se pretende conseguir los objetivos fijados.

3.1.1. Variables y restricciones de iluminación

Una variable de iluminación define qué aspectos de la escena pueden cambiarse durante el proceso de optimización. Usualmente son aspectos de la escena que pueden tener un efecto directo en la iluminación (como propiedades de las fuentes luminosas) y en los objetos de la escena (como su posición o su color). Ejemplos de variables de optimización

pueden ser la posición de una abertura en una pared, la posición de una escultura en un pasillo, o el color del piso de una habitación. Una restricción puede estar asociada a una variable o a la función de optimización. Ejemplos de la primera pueden ser que un objeto pueda posicionarse solamente en cierta sección de la escena, que el color de una superficie pueda variar solamente en su luminosidad pero no en la tonalidad. Un ejemplo de restricción de la función de optimización puede ser limitar la luminosidad en cierta superficie. En definitiva las variables dan forma al dominio X y las restricciones determinan al conjunto S de la Ecuación 3.1.

3.1.2. Objetivos de iluminación

Los objetivos de iluminación determinan qué aspectos de la iluminación desea alcanzarse durante el proceso de optimización. Matemáticamente se expresa por medio de una función f (ver Ecuación 3.1) que quiere maximizarse (o minimizarse). La función que se usa en este trabajo es precisamente optimizar la radiosidad total en una superficie, que es el cociente de la suma de luz entrante y la superficie total. Otra función de optimización que podría haberse incluido es minimizar la varianza en la irradiancia de la superficie, que mide la "uniformidad" de la luz recibida en toda una superficie de interés.

Medición de irradiancia

La medición de la irradiancia total de una superficie puede aproximarse contando cuántos fotones en la etapa del photon tracing impactaron con una superficie de interés s. La medida de la irradiancia de una superficie s se estima como:

$$\mathcal{I}_s(x) \approx F_N(x) = \frac{\sum_{p \in P} W_{p_r} + W_{p_g} + W_{p_b}}{AN} \tag{3.2}$$

donde P es el conjunto de fotones que impactaron sobre la superficie s, A es el área de s, N es la cantidad de fotones generados, y W_{p_r},W_{p_g},W_{p_b} es la potencia de cada fotón en los colores rojo (r), verde (g) y azul (b).

3.2. Optimización Utilizando VNS

El algoritmo elegido para hacer la optimización es VNS. Como se discutió en la Sección 2.10.3, VNS es simple de implementar y puede utilizarse para resolver problemas de iluminación. La optimización se da en dos pasos: en uno se busca una dirección de mejora, donde se aplica la heurística de primer ascenso, y en el siguiente se toma la decisión de saltar de un vecindario al siguiente. El método adaptado para este trabajo se explica en alto nivel en el Algoritmo 3.1.

La función *buscarMejora* es equivalente a la función del mismo nombre del Algoritmo 2.3. La operación se detiene luego de un número determinado de iteraciones. Entonces si se quiere limitar el tiempo de ejecución puede ajustarse el número de iteraciones. Una iteración lleva como máximo un par de segundos en el caso de las escenas de prueba, como se muestra en la Sección 5.4.

En las secciones que siguen se profundizará sobre el significado del radio y la función *buscarMejora*.

Procedimiento *buscarMejora*

El procedimiento de primera mejora, correspondiente a la llamada a la función *buscar* en el Algoritmo 3.1. En esa función es en donde se encuentran los máximos locales y se atacan algunos de los problemas que un VNS clásico puede tener al aplicarse a un problema

Algoritmo 3.1 Adaptación de VNS usado en el trabajo

```
VNSAdaptado(x):
iteracion ← 1
while iteracion ≤ maxIteraciones do
  radio ← 0,05
  while radio < 1 do
    x_mejor ← buscarMejora(x, radio)
    hayMejora ← x_mejor ≠ x
    if hayMejora then
      x ← x_mejor
      radio ← 0,05
    else
      radio ← radio + 0,05
    end if
  end while
end while
return x
```

de optimización del tipo que se abarcó en este trabajo, por ejemplo problemas donde se tiene un componente estocástico.

En primer lugar, para la búsqueda en el vecindario de una configuración x es necesario encontrar un criterio para determinar vecindarios $\mathcal{N}(x)$. Como se explicó en la Sección 3.2 se usa un radio para enumerar los vecindarios. Debido a que los problemas manejados en este trabajo son de carácter continuo, y por la naturaleza de las variables que se manejan, es necesario limitar la cantidad de intentos I a realizar durante la visita de un vecindario. En el algoritmo se eligió hacer un número máximo de visitas por vecindario, antes de intentar con otro diferente.

Algoritmo 3.2 Versión simplificada de *buscarMejora*

```
buscarMejora(x, radio):
repeat
  x' ← buscarVecino(x, radio)
  if f(x') > f(x) then
    return x'
  end if
until no se haya repetido I veces
return x
```

En el Algoritmo 3.2 se puede ver el método de *buscarMejora*, en una versión simplificada. I es la cantidad de intentos que se hace para buscar un vecino en el vecindario. Podría tomarse en cuenta la complejidad del ILP para elegir I. Por ejemplo, I puede ser proporcional al número de variables del problema y la cantidad total de iteraciones a realizar. La función *buscarVecino* genera una configuración x' en el vecindario $\mathcal{N}(x, r)$, usando las técnicas que se detallan en la Sección 4.4.

Vecindarios $\mathcal{N}(x)$

Otro aspecto es la elección de vecindarios. En el Algoritmo 3.1 se menciona un radio, pues durante el desarrollo se quiso preservar cierta independencia entre el VNS y las variables de optimización. Por ello se eligió construir una forma genérica de vecindarios para todas las variables definiéndolos a través de *radios* que emulan ser los de una n-

circunferencia en un espacio euclídeo. Los vecindarios están definidos para $0 \leq r \leq 1$ y cumplen que: (a) el vecindario de radio 0 tiene solo al elemento del centro: $\mathcal{N}(x, 0) = \{x\}$, (b) Si $r < r'$ entonces $\mathcal{N}(x, r) \subseteq \mathcal{N}(x, r')$ y (c) El vecindario de radio 1 contiene al conjunto de todas las soluciones factibles: $S \subseteq \mathcal{N}(x, 1) \subseteq X$.

El radio r del Algoritmo 3.1 define qué vecindario $\mathcal{N}(x, r)$ se está explorando. El vecindario se construye componiendo los vecindarios de cada una de las variables de optimización. Por ejemplo si se quiere calcular el vecindario en un problema donde hay dos variables de optimización: la posición de una luz x_1, y la posición de un objeto x_2, las soluciones que estén a una "distancia" r de x_1 y a una "distancia" r de x_2, entonces estarán en el vecindario $\mathcal{N}(x, r)$. Matemáticamente eso se puede expresar como: $\mathcal{N}(x, r) = (\mathcal{N}_1(x_1, r) \times \mathcal{N}_2(x_2, r) \times \ldots \times \mathcal{N}_n(x_n, r))$, donde x_i con $i \in 1, 2, \ldots, n$ es la i-ésima variable de optimización. En la Sección 4.4 se especifican cómo se calculan los vecindarios para cada tipo de variable de optimización.

3.3. Estimación de la Función Objetivo

En el Algoritmo 3.2 puede verse que se evalúa la función objetivo f. En el caso de querer maximizar la irradiancia de una superficie f sería F_N: la estimación de la irradiancia de esa superficie s. Esta estimación se genera utilizando photon mapping. Más específicamente se utiliza solamente la etapa de photon tracing, hallando la cantidad de fotones que impactan sobre la superficie s luego de generar N fotones, y usando la Ecuación 3.2. No es necesario aplicar la etapa de Ray Tracing para evaluar F_N. En el algoritmo de photon mapping clásico no se guardan aquellos fotones que impactan en s directamente desde una fuente luminosa porque se usa raytracing para relevar esa información acerca de ellos, como se explica en la Sección 2.6.2. Por esto se tiene que modificar el algoritmo para sí guardarlos. De lo contrario la estimación de irradiancia puede dar una cifra diferente a la cantidad real. La Figura 3.1 resume el proceso iterativo de optimización.

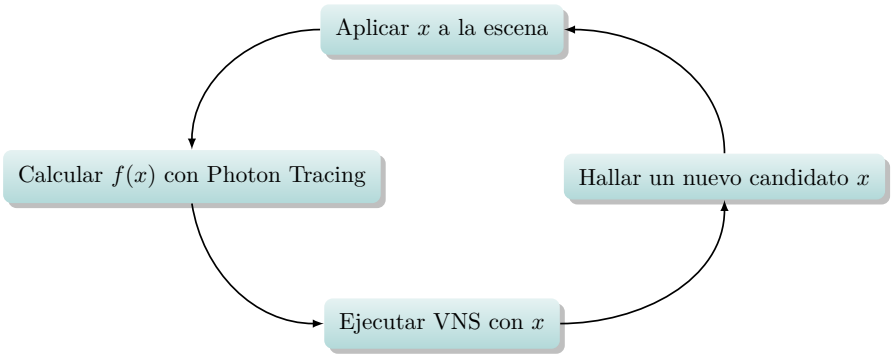


Figura 3.1: Proceso de optimización

3.3.1. Optimización estocástica

Usando los métodos que se propusieron en este trabajo no se puede tener un valor real de la función $\mathcal{I}_s$, pero sí se tiene una *estimación* de su valor. En las secciones siguientes se desarrolla un modelo estadístico de $\mathcal{I}_s$ y se estudia cómo modificar VNS para poder soportar el uso de estimaciones en lugar de valores reales de la función objetivo.

La estimación de la irradiancia que se calcula utilizando photon mapping es consistente. Como explica Crane (2014), un estimador Γ_N para una cantidad I es consistente si

$$\lim_{N \to \infty} P\left([\Gamma_N - I] > \varepsilon\right) = 0$$

Por lo que se puede concluir que la esperanza $E(\Gamma_N) = I + \varepsilon$. Para el caso de la estimación de la irradiancia $\mathcal{I}_s(x)$ se puede decir que $E(F_N(x))$ es I y $\varepsilon = 0$. [1]

A pesar de tener una aproximación del valor de $\mathcal{I}_s(x)$ a través de la Ecuación 3.2, no se sabe su valor real. Sin embargo, durante el uso de VNS es necesario poder comparar dos soluciones para saber si una es mejor que la otra. Con estimaciones no se puede determinarlo si sus valores son muy similares.

$F_N(x)$ se calcula usando la Ecuación 3.2. Como el conjunto de los fotones que cayeron en la superficie luego del photon tracing varía de un experimento al siguiente, el valor $F_N(x)$ puede variar. Esto es debido a que los fotones se generan pseudo-aleatoriamente, pero en promedio el valor de F_N es $\mathcal{I}_s(x)$. El problema radica en que no se sabe cuán precisa es esa estimación. Naturalmente, cuanto mayor es el valor de N: los fotones lanzados, más preciso es el valor de $F_N(x)$, dado que la varianza del error ε es menor. Se necesita poder comparar el valor de dos configuraciones para ejecutar el VNS en su versión determinista. Algunas alternativas para comparar dos configuraciones x' y x que consisten en hallar estimaciones de alta calidad de $\mathcal{I}_s(x)$ que permitan compararlas de manera categórica. Dos métodos posibles son usar un número de fotones N suficientemente grande para poder tener un error pequeño ε que permita comparar ambas soluciones con cierta seguridad, o realizar un promedio de m estimaciones de la misma configuración y por la Ley de los Grandes Números el promedio será una estimación más cercana al valor real (Pollard, 2001). En ambos casos no es posible realizarlo de manera práctica, por lo que se puede optar por tener una certeza estadística p, estudiando la función de distribución de F_N. Si se tiene una certeza con una probabilidad p de ser cierta, también está la probabilidad $1-p$ de que se esté errado. Por eso la heurística de optimización tiene que tener en cuenta los falsos positivos y los falsos negativos en la optimización. Es necesario poder ir atrás en el proceso de búsqueda del óptimo local y global. Además se pueden plantear interrogantes acerca de cuán precisa es la estimación $F_N(x)$ generada, y cuál es el valor óptimo de N para comparar dos soluciones. A mayor número de fotones lanzados la estimación es de mayor calidad, pero es más lenta de generar. Otro punto a tener en cuenta es que el algoritmo de photon mapping es limitado por la memoria por lo que hay un valor máximo de N admisible por la máquina.

A continuación se estudia algunas propiedades estadísticas de $F_N(x)$ para contestar algunas de las preguntas y adaptar el VNS a las características del problema.

3.3.2. Propiedades estadísticas de la estimación de irradiancia

Diferentes ejecuciones de la etapa de photon tracing pueden dar distintos valores de $F_N(x)$, por lo que se puede aseverar que $F_N(x) : Y \to \mathbb{R}$ es una variable aleatoria, donde Y es el espacio muestral: todos posibles experimentos de photon mapping para una configuración x.

En el Apéndice C se demuestra que la distribución de $F_N(x)$ es binomial, por lo que puede ser aproximada por una distribución normal $\mathcal{N}(\mu, \sigma^2)$ usando el teorema de De Moivre-Laplace (Feller, 1950). En este caso μ es la esperanza y σ^2 es la varianza. Dichos valores dependen de la probabilidad $p_s(x)$ de que un fotón impacte en la superficie de interés. Dado que el valor de $p_s(x)$ no es conocido directamente, se aproxima a través de $\bar{p_s}(x)$: el cociente entre la cantidad de fotones que impactaron en la superficie durante el experimento (la etapa de photon tracing) y el número total de fotones emitidos N.

Teniendo el valor de $\bar{p_s}(x)$ se puede estimar μ y σ. Se denotará $\bar{\mu}$ y $\bar{\sigma}$ (respectivamente)

[1] En Crane (2014) se dice que el photon mapping es un método sesgado, es decir que tiene un error no nulo en la estimación. Esto refiere únicamente al cálculo del color de cada píxel en la imagen final y no a la estimación de la irradiancia de la superficie.

a esas estimaciones y se calcularán usando las ecuaciones de la Sección C.3:

$$\mu(F_N(x)) \approx \bar{\mu}(F_N(x)) = \frac{W}{A}\bar{p_s}(x) \tag{3.3}$$

$$\sigma(F_N(x)) \approx \bar{\sigma}(F_N(x)) = \sqrt{\left(\frac{W}{A}\right)^2 \frac{\bar{p_s}(x)(1-\bar{p_s}(x))}{N}} = \frac{W}{A}\sqrt{\frac{\bar{p_s}(x)(1-\bar{p_s}(x))}{N}} \tag{3.4}$$

Donde:

- W es la suma de las potencias de los fotones que impactaron en la superficie.
- A es el área de la superficie.
- N es la cantidad de fotones lanzados.
- $\bar{p_s}(x)$ es la probabilidad estimada de que un fotón impacte en la superficie.

3.3.3. Comparación de configuraciones

$F_N(x)$ tiene una distribución aleatoria como fue explicado en la Sección 3.3.2. Al ser $F_N(x)$ binomial se puede aproximar por una distribución normal. A pesar de que no se puede tener una certeza de probabilidad 1 de que $\mathcal{I}_s(x)$ es mayor o menor que $\mathcal{I}_s(y)$, se pueden usar los intervalos de confianza de $\bar{\mu}$, calculados a partir de la desviación estándar de $F_N(x)$ y $F_N(y)$, para comparar ambas configuraciones con cierta probabilidad estadística. Si se tiene un intervalo de confianza para ambas distribuciones de confianza de nivel $1-\alpha$, se puede decir que decir que $F_N(x) > F_N(y)$ si ocurren las siguientes condiciones:

1. Usando la Ecuación 3.3 es equivalente a que $\bar{p_s}(x) > \bar{p_s}(y)$ debido a que W y A son fijos para cualquier valor de x.

$$\bar{\mu}\left(F_N(x)\right) > \bar{\mu}\left(F_N(y)\right) \Longleftrightarrow \bar{p_s}(x) > \bar{p_s}(y)$$

2. Sus estimaciones estén lo suficientemente apartadas entre sí, es decir que sus intervalos de confianza de probabilidad $1-\alpha$ no se intersequen:

$$\left|\bar{\mu}\left(F_N(x) - F_N(y)\right)\right| > \bar{\sigma}\left(F_N(x) - F_N(y)\right) z_{1-\alpha} \tag{3.5}$$

   donde $\bar{\sigma}(F_N(x) - F_N(y))$ es la desviación estándar calculada de $F_N(x) - F_N(y)$, calculada usando la Ecuación 3.4.

Debido que debido a que las muestras de $F_N(x)$ y $F_N(y)$ son independientes entre sí, entonces su covarianza es nula. Lo cual implica que

$$\overline{Var}\left(F_N(a) - F_N(b)\right) = \overline{Var}\left(F_N(a)\right) + \overline{Var}\left(F_N(b)\right)$$

Entonces puede deducirse que

$$\bar{\sigma}\left(F_N(x) - F_N(y)\right) z_{1-\alpha} = \sqrt{\bar{\sigma}\left(F_N(x)\right)^2 + \bar{\sigma}\left(F_N(y)\right)^2} z_{1-\alpha} \tag{3.6}$$

Aplicando la propiedad de que la esperanza es lineal[2], y utilizando la Ecuación 3.6, la Ecuación 3.5 es equivalente a

$$\left|\bar{\mu}\left(F_N(x)\right) - \bar{\mu}\left(F_N(y)\right)\right| > \sqrt{\bar{\sigma}\left(F_N(x)\right)^2 + \bar{\sigma}\left(F_N(y)\right)^2} z_{1-\alpha} \tag{3.7}$$

[2] $E(X-Y) = E(X) - E(Y)$

Si se cumple la anterior desigualdad entonces x e y son *distinguibles* con probabilidad $1-\alpha$ ¿Qué sucede si no se cumple la desigualdad? Entonces x e y no son distinguibles entre sí para el tamaño N de la muestra y el intervalo de confianza $1-\alpha$ dado.

Entonces se consideran los siguientes casos:

1. Si $\bar{\mu}\left(F_N(x)\right) > \bar{\mu}\left(F_N(y)\right)$ y se cumple (3.7) entonces se afirma que $\mathcal{I}_s(x) > \mathcal{I}_s(y)$ con confianza $1-\alpha$. Su notación es $\mathcal{I}_s(x) >_{1-\alpha} \mathcal{I}_s(y)$
2. Si $\bar{\mu}\left(F_N(x)\right) < \bar{\mu}\left(F_N(y)\right)$ y se cumple (3.7) entonces se afirma que $\mathcal{I}_s(x) < \mathcal{I}_s(y)$ con confianza $1-\alpha$. Su notación es $\mathcal{I}_s(x) <_{1-\alpha} \mathcal{I}_s(y)$
3. Si no se cumple (3.7) entonces se afirma que $\mathcal{I}_s(x) = \mathcal{I}_s(y)$ con confianza $1-\alpha$. Su notación es $\mathcal{I}_s(x) =_{1-\alpha} \mathcal{I}_s(y)$

Todas estas afirmaciones son estadísticas debido a que pueden haber falsos positivos y falsos negativos: por ejemplo pueden haber casos donde se afirme que $\mathcal{I}_s(x) >_{1-\alpha} \mathcal{I}_s(y)$ pero en realidad $\mathcal{I}_s(x) < \mathcal{I}_s(y)$.

Una forma más gráfica de visualizar los items enunciados anteriormente es mediante la Figura 3.2.

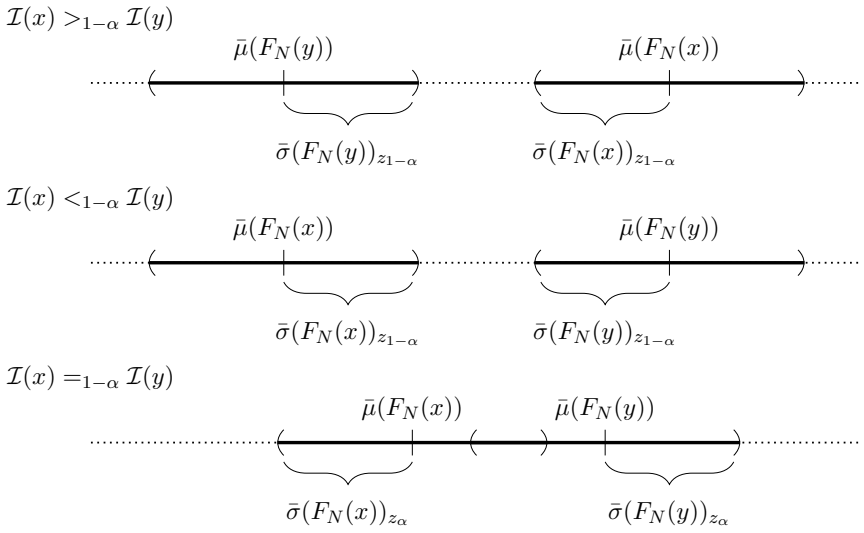


Figura 3.2: Comparación de dos configuraciones x e y

3.3.4. Conjunto de soluciones óptimas en VNS

Hasta ahora se tiene una manera estocástica de comparar soluciones, pero la heurística que se usa requiere saber si una solución es mejor que la otra. Como no es posible saberlo con probabilidad 1, se opta por tener un conjunto de soluciones óptimas, cada una de esas soluciones óptimas es mejor que el resto de las soluciones no óptimas, pero entre ellas no se puede distinguir cuál es mejor, para un valor de N dado. En este trabajo se llama a este conjunto *ISOC*[3]. Al final de la recorrida del VNS el objetivo es hallar cuáles configuraciones recorridas son óptimas e indistinguibles entre sí. Para el caso de hallar el máximo esto es equivalente a:

$$\forall x, y \in ISOC : \mathcal{I}_s(x) =_{1-\alpha} \mathcal{I}_s(y)$$
$$\forall x \in ISOC, z \notin ISOC : \mathcal{I}_s(x) >_{1-\alpha} \mathcal{I}_s(z)$$

[3]Acrónimo de *Indistinguishable Set of Optimum Configurations* que quiere decir Conjunto Indistinguible de Configuraciones Óptimas

4

Solución Propuesta

El objetivo del capítulo es explicar detalladamente la solución que se implementó para poder resolver ILPs, como los descritos en el Capítulo 3, utilizando OptiX para implementar photon mapping, y VNS como metaheurística de optimización para explorar el espacio de soluciones.

La solución consta de dos grandes componentes: *OppositeRenderer* (véase Sección 2.9) e *ILPSolver*. El primero se encarga de cargar y renderizar escenas usando photon mapping y de recabar información estadística generada durante la etapa de recorrida del mapa de fotones. El segundo, ILPSolver es el encargado de leer y resolver los ILPs usando las funciones de OppositeRenderer. En la Figura 4.1 se puede ver la relación entre ambos. OppositeRenderer tuvo que ser modificado para soportar un conjunto de funcionalidades para ser útil durante proceso de optimización: implementar photon tracing, modificar la escena y recabar información de esta.

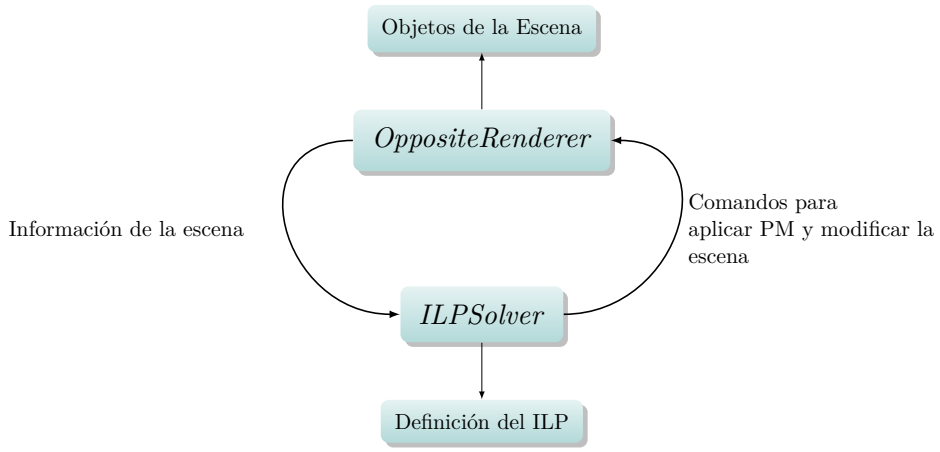


Figura 4.1: Relación entre los componentes OppositeRenderer e ILPSolver

4.1. Intenciones de Iluminación Implementadas

En la implementación se incluyeron diferentes tipos de restricciones, variables y funciones de optimización. En el futuro se podrían agregar más tipos de variables de optimización, para que el programa sea capaz de poder resolver una gama de ILP más compleja. ILPSolver define una interfaz, donde se pueden agregar de manera relativamente simple nuevos tipos de variables. Agregar nuevas funciones objetivo es más complejo, pues debería modificarse el OppositeRenderer para que sea capaz de recabar la información de la

escena para poder evaluarlas.

En términos generales, las variables que se pueden utilizar en ILPSolver son las siguientes:

- **Posición de objetos**: Se puede cambiar la posición de un objeto sujeto a que esté dentro de la geometría de la escena.
- **Posición de luces**: Se puede acotar la posición de una luz a una superficie determinada de la escena. Las luces pueden ser puntuales, o bien emisores en forma de rectángulos.
- **Luces direccionales**: Para luces direccionales es posible variar su dirección.
- **Aberturas**: Se puede cambiar la posición de una abertura en una pared para simular una ventana, dentro de una superficie determinada
- **Color de superficies**: Se puede cambiar el coeficiente de reflexión difusa K_d de una superficie para optimizarlo.

Para cada tipo de variable de optimización (excepto la de la dirección de la luz) se debe definir una restricción. Las restricciones dan forma a los vecindarios al aplicar el VNS, esto se detalla en la Sección 4.4.

- Para la posición de objetos, luces y aberturas se debe definir en qué superficie de la escena se debe ubicar.
- Para las variables de tipo color, se debe definir el rango de valores que puede tomar el color.

Estas restricciones se expresan en un archivo de configuración que luego se lee por el programa ILPSolver. El archivo es un XML donde el usuario puede configurar las variables que se quiere dejar libres, qué superficie se va a optimizar y otros parámetros del problema. En la Sección 4.2.1 se especifica el esquema del archivo. Se implementó la función de optimización que maximiza la irradiancia de una superficie s: $\mathcal{I}_s$, que se estima mediante la función F_N, y que a su vez es calculada utilizando photon tracing. Ambas fueron definidas en la Sección 3.1.2.

4.2. Funcionamiento del Programa

El programa funciona tomando como entrada un archivo que define las intenciones de iluminación y da como salida otro archivo con las soluciones encontradas, y un conjunto de imágenes de las soluciones óptimas: aquellas que pertenecen al ISOC que se detalló en la Sección 3.3.4.

4.2.1. Archivo de definición de un ILP

El archivo de definición describe el ILP: variables, restricciones, la superficie cuya iluminación se quiere maximizar y otros parámetros de ajuste. Es un archivo XML que sigue un esquema específico para ser leído por el programa ILPSolver. El archivo contiene las siguientes secciones:

- La escena de entrada: es la ruta a un archivo en formato Collada (The Khronos Group Inc., 2008) que contiene la escena a optimizar. Otros formatos soportados son los que puede importar la biblioteca *AssImp*[1].

[1] AssImp es la abreviación de *Open Asset Import Library*, una biblioteca de importación de objetos en diferentes formatos de modelado 3D estándar.

- La carpeta de salida donde se generarán las imágenes y la información sobre las soluciones encontradas.
- Las variables de optimización y sus restricciones.
- El tamaño de la grilla del espacio discreto de configuraciones. Su significado se discute en la Sección 4.8.
- El nombre de la superficie cuya irradiancia se va a optimizar.
- La cantidad de iteraciones que se va a hacer en el VNS. Este número sirve para limitar la cantidad de tiempo que durará el proceso de búsqueda de configuraciones. En general habría que ajustar ese número a la cantidad de variables, y la complejidad de la escena dado, que escenas con más variables pueden tardar más tiempo en ejecutar.

En el Apéndice D se encuentra un archivo de configuración de ejemplo que puede consultarse.

4.2.2. Salida del programa

La salida del programa son dos tipos de elementos: un archivo en formato CSV que contiene todas las soluciones encontradas, y un conjunto de imágenes auxiliares para cada una de las soluciones del ISOC, que son útiles para visualizar cómo se ve la escena con cada una de esas configuraciones. Las imágenes se generan utilizando la ubicación de la cámara provista por archivo de la escena. El archivo CSV de salida tiene el siguiente esquema de columnas:

1. El número de iteración i donde se generó esa estimación.
2. Un número de columnas para describir el estado la configuración x, por ejemplo la posición de una luz y de un hoyo en una pared.
3. 3 columnas para definir el intervalo de confianza de la estimación de $F_N(x)$: el centro $\bar{\mu}(F_N(x))$ y sus extremos: $\bar{\mu}(F_N(x)) \pm \bar{\sigma}(F_N(x))z_{1-\alpha}$.
4. La cantidad de fotones usados para generar la estimación (N).
5. El tiempo t (en segundos) que tardó en generarse la estimación.

En el Cuadro 4.1 puede verse las primeras filas del resultado de una ejecución.

i	H_x	H_y	H_z	$F_N(x)_{min}$	$F_N(x)$	$F_N(x)_{max}$	N	t
0	-1,00	0,00	1,00	7,896	7,941	7,986	4.000.000	6,686974
1	-1,00	0,05	0,86	7,962	8,008	8,053	4.000.000	1,888766
2	-1,00	0,04	0,81	7,962	8,008	8,053	4.000.000	1,938297
...	...	...	...	...	...	...	...	...

Cuadro 4.1: Ejemplo de las primeras filas de una ejecución que consiste en hallar la mejor posición de una abertura (H_x, H_y, H_z) en una superficie.

El intervalo de confianza de las soluciones $1-\alpha$ es fijo y se define en 95 %. Esto permite encontrar soluciones de calidad y limitar el tamaño del ISOC.

4.3. Implementación de Photon Mapping

El código del paquete OppositeRenderer fue el punto inicial de este trabajo para construir un programa que resuelva ILPs. El paquete *ILPSolver* que resuelve el problema de optimización de ILP depende de OppositeRenderer para funcionar. ILPSolver toma como entrada los datos de la escena luego de la etapa de photon tracing generados por OppositeRenderer y los utiliza para decidir la calidad de la configuración. Luego de la optimización se usa OppositeRenderer de nuevo para generar las imágenes finales de las soluciones óptimas.

4.3.1. Criterio de selección

En este trabajo se partió de un paquete que ofrece la capacidad de hacer photon mapping utilizando OptiX. La implementación del algoritmo es compleja, y se necesitan tener en cuenta muchos aspectos como la carga de las escenas en formatos estándar, manejo de texturas, y materiales diversos. Debido a que no se tenía inicialmente un conocimiento suficiente de la técnica de photon mapping, implementarla enteramente desde cero podría llevar un tiempo considerable. En ese marco, se investigaron diversas alternativas de paquetes, que implementaran photon mapping con OptiX.

SDK de OptiX

Una opción posible era implementar photon mapping desde cero utilizando OptiX, que es una biblioteca especializada en Ray Tracing usando la GPU (Sección 2.8). Una primera alternativa fue utilizar directamente los ejemplos de la SDK de OptiX, entre los que se incluye una implementación básica de photon mapping progresivo. Sin embargo la implementación no era fácil de entender, estaba atada a una escena de ejemplo, y hacía uso de árboles kD, que si bien son simples, no son muy eficientes como estructuras de photon mapping en la GPU.

OppositeRenderer

Otra alternativa fue usar código que implementara photon mapping progresivo y modificarlo de forma tal que implementara photon mapping clásico. En ese contexto se consideró OppositeRenderer. La ventaja de este paquete es que permite la carga de escenas desde formatos conocidos como Collada (The Khronos Group Inc., 2008) o Wavefront OBJ (Bourke, 2012). Esto hace posible el uso de editores gráficos para modificar las escenas, sin necesidad de modificar los archivos manualmente. Además utiliza un método de renderización distribuido donde la imagen final puede ser generada por varios ordenadores que están en una misma red de computadoras local. El código original tiene algunas características que podrían utilizarse en el futuro, como efectos volumétricos, distintos tipos de estructuras para el armado del mapa de fotones, y una implementación de Path Tracing bidireccional. Sin embargo, fue hecho usando versiones relativamente antiguas CUDA, OptiX y de otras bibliotecas por lo que habría que migrarlas a las nuevas versiones. Además no tiene instrucciones sobre la compilación del código. Todo lo mencionado podría generar complicaciones al hacer funcionar el programa por primera vez. Sin embargo, el conjunto de características antes mencionadas determinó que se eligiera como base para la implementación de photon mapping.

4.3.2. Modificaciones a OppositeRenderer

El paquete original de OppositeRenderer no estaba diseñado para ser utilizado para un proceso de optimización de ILP, y sufrió un número importante de modificaciones que

se resumen en el listado que sigue:

- Se actualizó sus dependencias para usar las últimas versiones disponibles de OptiX, CUDA y Qt (Qt Company Ltd., 2015). En la implementación se usa OptiX 3.8.0, CUDA 7.0 y Qt 5.5.
- Se documentó el código original y se publicó en el portal de GitHub para que fuera disponible públicamente (Avas, 2015).
- Se agregaron diversas optimizaciones para permitir la carga a partir de escenas creadas con la herramienta Blender (Blender Foundation, 2015).
- Se agregó soporte para modelar luces dimensionales, y permitir tener aberturas en las escenas.
- Se modificaron las estructuras de photon mapping para recabar datos de irradiancia y de los fotones durante la etapa de recorrida del mapa de fotones para poder implementar ILP.
- Se construyó una implementación de photon mapping clásico a partir de la implementación de photon mapping progresivo.
- Se quitó la funcionalidad de Rendering en Red que tenía la implementación inicial, pues no se podía probar si la funcionalidad era compatible con ILPSolver. Se debía adaptar la funcionalidad y conseguir hardware para probarla. Puede que en un futuro se vuelva a integrar, pudiendo aumentar el rendimiento del ILPSolver usando varias GPUs distribuidas en diferentes computadoras.
- Se agregó la posibilidad de modificar la geometría de una escena, utilizando Transformaciones de OptiX.

Durante el transcurso del proyecto se mantuvieron actualizadas las bibliotecas en las últimas versiones disponibles. Actualizarlas no fue un proceso complicado, dado que están debidamente documentadas. Una de las dificultades que se encontró es el poco soporte que tiene OptiX para el manejo de errores, además del hecho de que no es posible depurar un programa hecho en OptiX, de la misma manera que un proceso que ejecuta sobre la CPU o en CUDA. Esto afectó el proceso de desarrollo dado que muchas veces era complicado detectar cuál era la raíz del problema.

4.4. Vecindarios de VNS

Hay tres tipos de vecindarios $\mathcal{N}(x, r)$ según el tipo de variable de optimización. Para las variables de optimización que se mapean a una superficie siempre se asume que esa superficie es un rectángulo cuyos puntos están en un mismo plano. Esto se hace por simplicidad en el cálculo. Las posiciones de objetos, luces y aberturas caen en esta categoría. El segundo tipo son las variables que denotan una dirección que consiste mapear un vector unitario de $\mathbb{R}^3$ a $\mathbb{R}^2$, usando sus coordenadas esféricas. Finalmente para las variables de color se usa el tono del color representado según el modelo HSV. En las secciones siguientes se profundizará sobre cada uno de los tipos de vecindarios.

4.4.1. Mapeo de punto en superficie

Las variables de posición de objetos, aberturas y luces dentro de una superficie se mapean usando el mismo método, que consta en mapear un rectángulo en el espacio $\mathbb{R}^3$ a

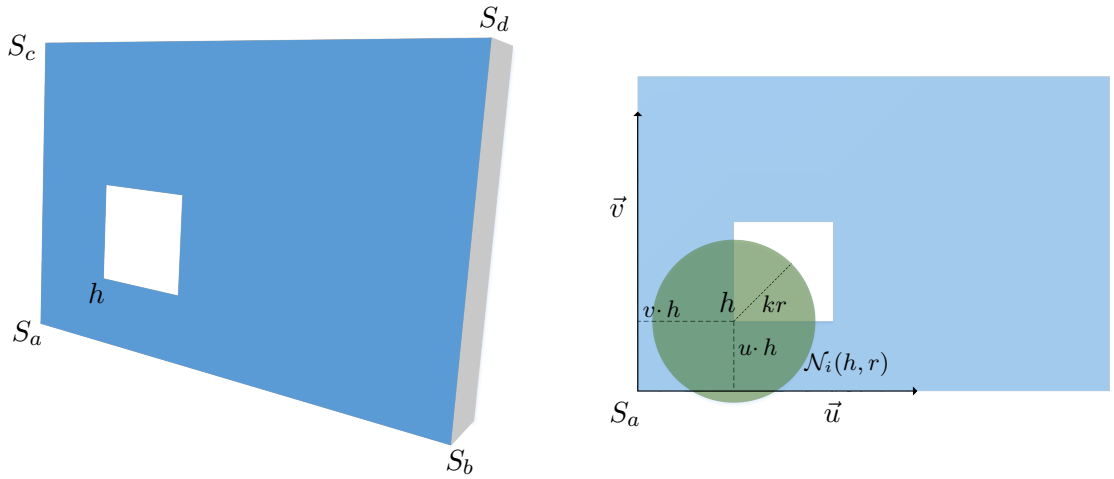


Figura 4.2: Mapeo de la posición de una luz en un cuadrilátero de una malla poligonal.

uno en $\mathbb{R}^2$, utilizando una base ortonormal de vectores para representar los puntos que se encuentren en el plano del rectángulo. En todos los casos se tiene un rectángulo donde la luz, el objeto o la abertura pueden estar. Para mapear la posición del objeto/luz/abertura, se calcula la posición de un punto de referencia de dicho elemento, si es una luz puntual, entonces el punto de referencia es su posición, pero si se trata de una abertura, un emisor difuso o un objeto, se toma como punto de referencia a uno de sus vértices y se lo considera como un objeto puntual. El objetivo es mapear un punto de un cuadrilátero que está en la escena 3D a otro cuadrilátero, pero en $\mathbb{R}^2$. Luego el vecindario es una circunferencia con centro en las coordenadas mapeadas. Para mapear un objeto, una abertura o una luz, se mapea solamente su punto de apoyo, que es algún vértice de la geometría. Para una luz puntual el punto de apoyo es la misma posición de la luz. Esto quiere decir que el objeto se puede mover libremente siempre y cuando el punto de apoyo caiga en el cuadrilátero de la restricción.

Matemáticamente el mapeo del punto en la superficie consiste en encontrar una función $f_i : \mathbb{R}^3 \to \mathbb{R}^2$ tal que $f(p)$ sea el mapeo del punto en la superficie. El vecindario $\mathcal{N}_i(p, r)$ es una circunferencia de centro p y radio kr, donde k depende del tamaño del cuadrilátero.

Mapeo de punto

Teniendo como vértices del rectángulo los puntos (S_a, S_b, S_c, S_d) (ver Figura 4.2), se calculan los vectores de sus ejes:

$$\vec{j} = S_b - S_a$$
$$\vec{k} = S_c - S_a$$

Luego se forma una base ortonormal que está formada por:

$$\vec{u} = \frac{\vec{j}}{\|\vec{j}\|}$$
$$\vec{v} = \frac{(\vec{j} \times \vec{k}) \times \vec{j}}{\|(\vec{j} \times \vec{k}) \times \vec{j}\|}$$

Entonces las coordenadas mapeadas de un punto p del cuadrilátero están dadas por sus proyecciones en los ejes $\vec{u}$ y $\vec{v}$.

$$(\vec{u} \cdot p, \vec{v} \cdot p)$$

Vecindario

Una vez que tiene el punto en coordenadas en $\mathbb{R}^2$ se puede calcular su vecindario $\mathcal{N}_i(p, r)$ como la circunferencia de centro $(\vec{u} \cdot p, \vec{v} \cdot p)$ y radio kr donde k es el largo de la diagonal del rectángulo. En la Figura 4.2 se puede observar la correspondencia cuando se mapea una abertura en una pared. Las coordenadas del punto p'' en la escena correspondiente a un punto $p' = (x', y')$ en el vecindario mapeado son $p'' = x'\vec{u} + y'\vec{v} + S_a$.

4.4.2. Vecindario para una luz direccional

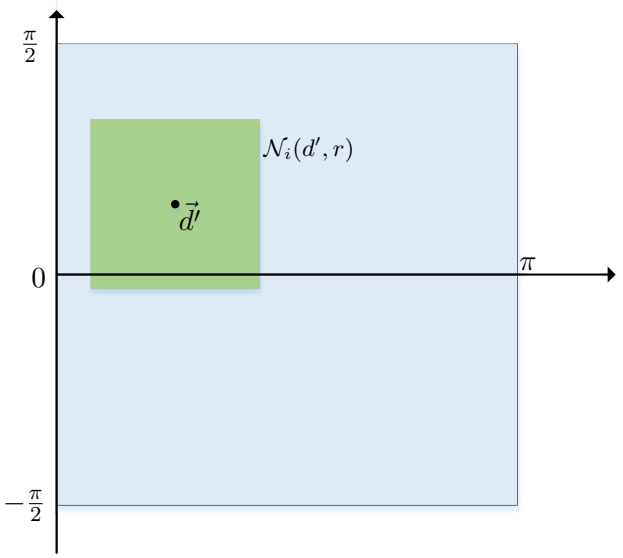


Figura 4.3: Vecindario de una variable de dirección.

Para mapear una dirección y hallar su vecindario se usan sus coordenadas esféricas. La ventaja de usarlas frente a las coordenadas cartesianas, es que se puede mapear un vector unitario, transformándolo a una forma en $\mathbb{R}^2$, donde es más fácil encontrar un vecindario. En este caso el mapeo $\vec{d'}$ de un vector unitario $\vec{d} = (x, y, z)$ usando coordenadas esféricas es $\vec{d'} = (\cos^{-1}(z), \tan^{-1}(\frac{y}{x}))$. Véase la Figura 4.4 para ver una versión gráfica de la representación. Para cualquier valor de $\vec{d}$ sus coordenadas están en el cuadrado $[0, \pi] \times [-\frac{\pi}{2}, \frac{\pi}{2}]$ de $\mathbb{R}^2$. Para calcular su vecindario de radio r se usa un cuadrado de lado πr y centro $\vec{d'}$. Las coordenadas cartesianas de una dirección $d'' = (x', y')$ dada en coordenadas esféricas son $d' = (\sin x \cos y, \sin x \sin y, \cos x)$. En la Figura 4.3 se ve cómo es el vecindario en el espacio mapeado, para una variable de dirección.

4.4.3. Vecindario para una variable de color

Las variables de color se representan en la notación Tono Saturación Valor (HSV). HSV es una representación alternativa del color que es más intuitiva que el cubo cartesiano de la representación RGB, donde los colores se representan en un cilindro (Figura 4.5). Además en HSV las diferentes tonalidades de un color se pueden computar de manera más fácil que en RGB, pues las variables de tipo de color varían su componente de *Valor*, representarlo en RGB es complejo debido que hay que combinar los distintos componentes de los colores para generar los distintos valores. El vecindario de un color RGB es el segmento de recta paralelo al eje del cilindro HSV que pasa por el color deseado.

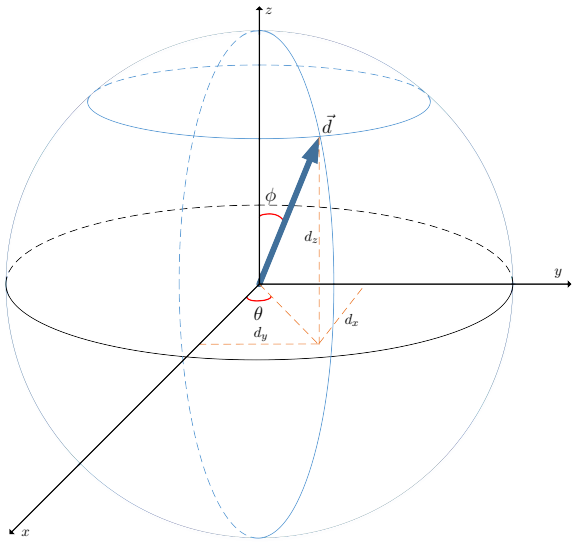


Figura 4.4: Coordenadas esféricas para un vector unitario.

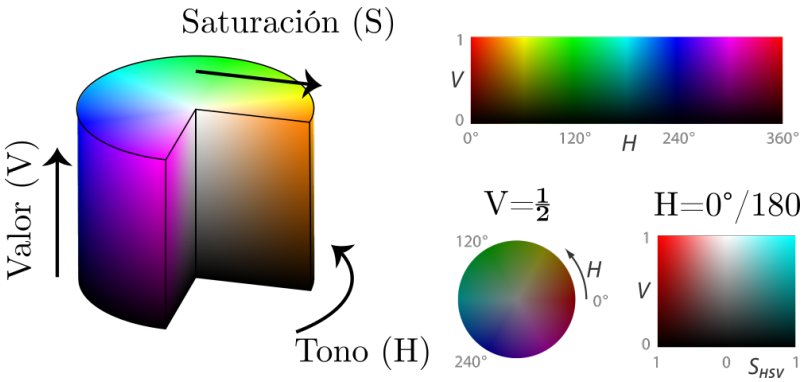


Figura 4.5: Representación de color HSV

4.5. Aplicación de Configuraciones usando OptiX

Para poder estimar la irradiancia $\mathcal{I}_s(x)$ de una superficie s según una configuración x, es necesario modificar los atributos de los nodos de la escena de OptiX. El objetivo de esta sección es desarrollar cómo se hacen estos cambios en la escena para cada tipo de variable.

Configuraciones de tipo posición

Para evaluar $\mathcal{I}_s$ hay que mover los objetos a sus posiciones correspondientes. Esto aplica para las variables de posición de luces y objetos en la geometría. Para hacerlo se usa el objeto de tipo Transformación de OptiX, introducido en la Sección 2.8.1, que permite aplicar una transformación lineal a un nodo del grafo de la escena. Por ejemplo si se quiere mover la posición de una mesa es más simple aplicarle un nodo transformación que modificar cada uno de los vértices manualmente, debido a que lo segundo tiene efectos negativos sobre el rendimiento. Esto se debe a que si se mueve el vértice de un objeto se tiene que volver a calcular completamente su estructura de aceleración. En la Sección 2.8.2 se describe en más detalles las estructuras de aceleración usadas en la implementación. Por ello se construye el grafo de OptiX de la escena asociando un nodo de Transformación a cada objeto que fuera necesario mover durante la optimización. Cada uno de estos nodos tiene asociado una matriz de Transformación homogénea 4×4, que permite aplicar cualquier transformación lineal como traslaciones o rotaciones. Inicialmente se asigna la matriz

de identidad I, que corresponde a no transformar el objeto. Si se desea mover el objeto asociado de la posición x hasta y, entonces se tiene que usar la matriz de transformación $\mathbf{I}+\mathbf{T}$ donde:

$$\mathbf{I}+\mathbf{T} = \begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Y $\Delta d = y - x = (d_x, d_y, d_z)$. En las Figuras 4.6 y 4.7 se puede ver la transformación de un nodo utilizando la matriz $\mathbf{I}+\mathbf{T}$ y su resultado en una escena de ejemplo.

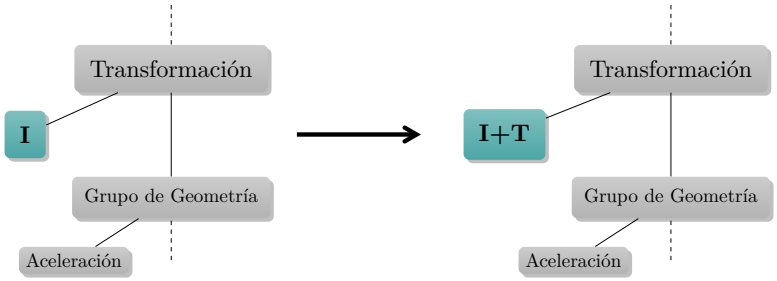


Figura 4.6: Aplicación de una configuración usando un nodo de tipo Transformación en el grafo de la escena de OptiX

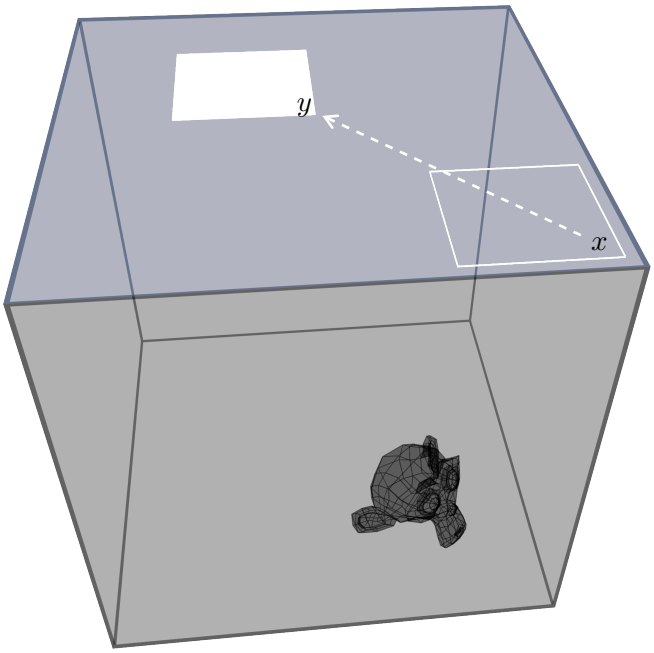


Figura 4.7: Aplicación de una transformación de traslación a una luz de la escena, moviéndola del punto x al punto y

Dirección de luz

Los fotones se generan desde un disco que se encuentra en un plano normal a la dirección de la luz, y ubicada la dirección opuesta a la dirección de la luz lejos del centro de

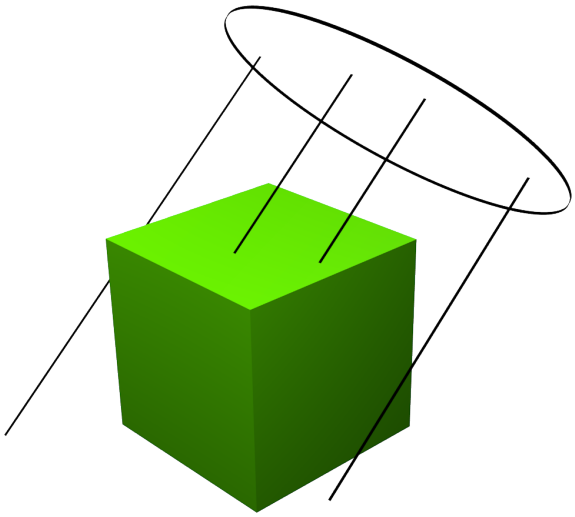


Figura 4.8: Generación de fotones para una luz direccional.

la escena, de forma tal de que todos los puntos de la superficie expuestos sean iluminados. La Figura 4.8 ilustra los distintos elementos. Las líneas indican los recorridos de algunos fotones que se generan en el disco. Si se desea modificar la dirección de la luz, solamente se debe cambiar de lugar la circunferencia donde se generan, sin necesidad de modificar los elementos del grafo de la escena de OptiX.

Color de la superficie

Para asignar el color de una superficie se cambian los valores del coeficiente difuso K_d almacenadas en el objeto OptiX de tipo Material correspondiente a la superficie afectada.

4.6. SIISOC

El objetivo del ILPSolver es encontrar un conjunto de soluciones óptimas, indistinguibles entre sí, llamado *ISOC* para un problema dado como se describió en la Sección 3.3.4. Ese conjunto de soluciones óptimas cumplen que $\forall x, y \in ISOC : x =_{1-\alpha} y$. Usando la Ecuación 3.7 se deduce que dos configuraciones x e y son indistinguibles si sus márgenes de confianza $1 - \alpha$ intersecan. Entonces todas las soluciones que estén en el ISOC van a tener una intersección común de sus intervalos de confianza. Se llama a esa intersección *SIISOC* que es el acrónimo del inglés de *Superposition Interval of ISOC* (Figura 4.9). En la implementación se compara una configuración candidata contra el SIISOC para saber si puede estar en el ISOC, debido a que es más eficiente que compararla una a una contra todas las soluciones. Cuando se tiene una solución candidata x' se pueden tener los siguientes casos:

A) x' no interseca con el SIISOC y su intervalo de confianza es menor. Entonces no va a poder pertenecer al ISOC, debido a que $x' <_{1-\alpha} x_i^*$ para algún elemento x_i^* del ISOC. La Figura 4.9 ilustra este fenómeno. Este caso es común después de haber encontrado un óptimo local de buena calidad.

B) x' interseca con el SIISOC, o su intervalo de confianza es mayor. Entonces va a pertenecer al ISOC, *puede* ser mejor que *algunas* celdas del ISOC, a las que hay que eliminar del nuevo ISOC. Matemáticamente $ISOC_{new} = x' \cup \{x \in ISOC_{old}, x =_{1-\alpha} x'\}$.

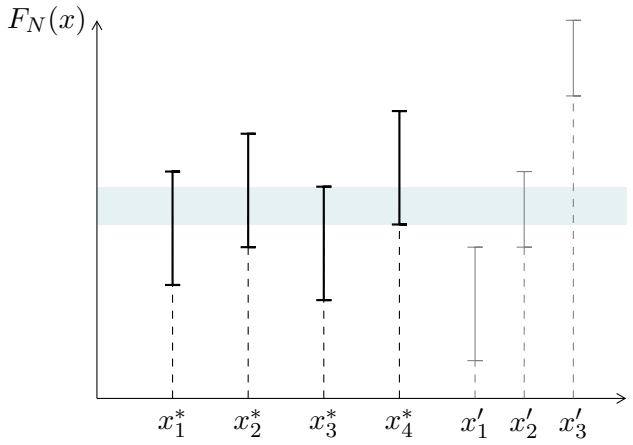


Figura 4.9: SIISOC para 4 configuraciones $x_1^*, \ldots, x_4^*$ cuando se pretende maximizar $F_N(x)$. La configuración candidata x_1' no es óptima y queda fuera del ISOC (caso A), mientras que x_2' y x_3' entran en el nuevo ISOC (caso B).

Tener el SIISOC facilita la validación de si una configuración candidata se encuentra en el ISOC, pues evita compararla contra todos los elementos del ISOC si se está en el caso A.

4.7. Evaluación de la Función Objetivo

En esta sección se explicará cómo se evalúa la función de estimación de $\mathcal{I}_s(x)$: $F_N(x)$ de forma de aumentar el rendimiento de la aplicación. Debido a que para evaluar $F_N(x)$ se necesita ejecutar una etapa de photon tracing, a mayor cantidad de fotones lanzados (N), más tiempo toma generar una evaluación. Por ese motivo es importante manejar un valor pequeño de N, y aumentarlo sólo si es necesario. Otra pregunta que puede plantearse es cuál es el valor de N adecuado a usar.

Determinación del tamaño del mapa de fotones

El valor de N determina la cantidad de fotones que se lanzan desde las fuentes luminosas, lo que a su vez determina el tamaño del mapa de fotones que se tiene. Una limitación del tamaño de N es la memoria disponible en la GPU, debido a que el mapa de fotones se almacena allí. Además, el mapa de fotones se representa a partir de una grilla uniforme ordenada, tal como se explica en la Sección A.2. Por otra parte hay que tener espacio para otras estructuras auxiliares que se usan en el photon mapping y que también están en la GPU, como el histograma que indica cuántos fotones hay en cada celda de la grilla uniforme, entre otras estructuras. Es muy complejo hallar una medida exacta de cuánto ocupan todos esos *buffers* en memoria de la GPU. Además la memoria puede sufrir de fragmentación externa. La fragmentación externa es el fenómeno que ocurre cuando la memoria está fragmentada en bloques discontinuos de tal forma que no pueda alojarse un *buffer* de tamaño T de forma continua en la memoria. Esto que puede limitar su capacidad efectiva. Debido a eso se estima una cota superior para N. Se comprobó experimentalmente que en GPUs de 2GB de memoria física N no debe ser mayor que 2^{22} bytes $\simeq 4,19 \cdot 10^6$ bytes.

En la implementación, si Q es la cantidad máxima de rebotes que se consideran en la etapa de recorrida del mapa de fotones, entonces la grilla uniforme va a tener a lo sumo QN fotones. El tamaño de la grilla uniforme en memoria de la GPU va a ser $T_P QN$ donde T_P es el tamaño de cada elemento de la grilla: un fotón. En el Apéndice E se especifica la estructura concreta de un fotón tal como se guarda en el mapa de fotones. En

la implementación Q es 4, y T_p son 40 bytes. Usando como N 2^{22} el tamaño de la grilla es aproximadamente 671 *MB*. Hay que tener en cuenta los otros *buffers* necesarios y la fragmentación externa.

La fórmula para calcular la cota superior de N teniendo una memoria disponible de M bytes es la siguiente:

$$N = 0,075\frac{M}{T_P}$$

La implementación de OppositeRenderer requiere que N sea un cuadrado perfecto, y además es bueno que ese cuadrado perfecto sea además múltiplo de 256, para mejorar el rendimiento ya que los bloques de procesamiento de CUDA funcionan más rápido. Por lo que la fórmula extendida para hallar N es la siguiente:

$$N = \left(\sqrt{0,075\frac{M}{T_P}} \texttt{ \& ~0xF}\right)^2$$

donde la operación `& ~0xF` implica redondear hacia abajo al número entero múltiplo de 16.

El término que multiplica es un factor de ajuste para que funcione correctamente en los equipos que se usaron para probar la solución implementada. Alternativamente se podría dar al usuario la posibilidad de determinar el valor de máximo de N manualmente.

Ajuste dinámico de N

Como se explicó en la Sección 4.6, puede haber 2 resultados cuando se compara una configuración x con el SIISOC. Durante el proceso de búsqueda lo más común es que la configuración se descarte debido a que no es mejor que SIISOC, por lo que si se genera una estimación $F_N(x)$, esta se descarta. El tiempo de cómputo dedicado a generar la estimación es proporcional a N. Para eso se toma como criterio siempre evaluar la configuración con un número de fotones $n < N$, si se cae en el caso B, se vuelve a calcular usando N fotones. De esta manera se ahorra tiempo, generando estimaciones de mayor calidad cuando es necesario. Al igual que N, n es un cuadrado perfecto múltiplo de 256.:

$$n = \text{máx}\left(\sqrt{qN} \texttt{ \& ~0xF}, 16\right)^2$$

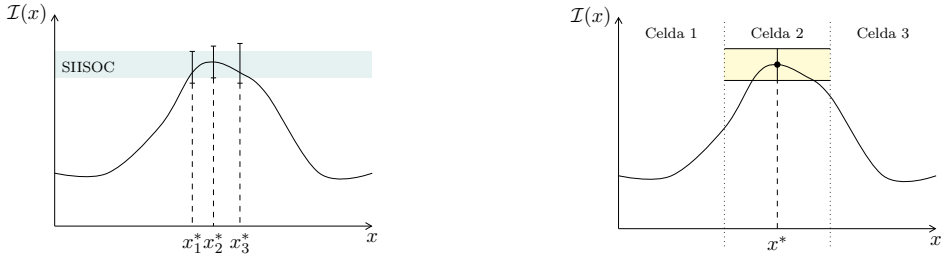
donde $0 < q < 1$ es una constante que determina la fracción de la cantidad máxima de fotones. En la implementación $q = 1/4$, lo que implica que $n \simeq \frac{N}{4}$.

Al evaluar una configuración se genera su estimación: $F_n(x)$. Si el intervalo de confianza de $F_n(x)$ se interseca con el SIISOC, entonces sí se calcula $F_N(x)$ $(n < N)$ que tiene un intervalo de confianza más pequeño, y por lo tanto es un mejor estimador. Luego se decide si entra o no en el ISOC. De esta manera, todas las soluciones del ISOC se calculan con máxima precisión y las configuraciones que no pueden ser candidatas a ser óptimas locales se calculan con mínima precisión, que es más económico y no afectan a la búsqueda de las soluciones finales. Hay que tener en cuenta que un valor muy pequeño de q puede causar que se tengan más situaciones de "falsos positivos": se genera un $F_N(x)$ que intersecan al SIISOC pero que luego al generar un valor de $F_N(x)$ termine no intersecándolo y siendo una solución no óptima. La elección de q tiene que ser balanceada teniendo en cuenta lo dicho anteriormente.

4.8. Espacio de Soluciones como Conjunto Discreto

Uno de los aspectos a considerar de la optimización estocástica consiste en que el ISOC pueda tener un gran número de configuraciones prácticamente iguales. Si el espacio

de soluciones X tuviese alguna métrica $|\ |: X \times X \to \mathbb{R}$, entonces habrá muchas soluciones en el ISOC que cumplirían que $|x - x'| < \varepsilon$ para un valor pequeño de ε. Esto sucede porque en general el valor de $\mathcal{I}_s$ es continuo en las inmediaciones del óptimo global y porque VNS tiende a intentar soluciones cercanas a las óptimas en el algoritmo de primera mejora, en los vecindarios $\mathcal{N}(x, r)$ cuando r es pequeño que ocurre al encontrar una solución óptima local por primera vez. Hacer discreto el espacio de soluciones en *celdas* de X ayuda a que el algoritmo VNS evalúe solamente una vez las inmediaciones de una configuración x. Luego de encontrar una solución óptima local x^*, se puede intentar probar soluciones ubicadas en celdas diferentes y en lugares alejados de x^*. En las Figuras 4.10 y 4.11 se puede ver la comparación entre un espacio continuo y otro discreto y cómo afecta al conjunto de soluciones que genera el VNS.



(a) ISOC de espacio continuo

(b) ISOC de espacio de celdas

Figura 4.10: Conversión de un intervalo real de configuraciones, a un conjunto discreto de 3 celdas. Las soluciones x_1^*, x_2^* y x_3^* se resumen en una sola solución x^* en el caso discreto

Partición en celdas

Para definir las celdas se parte cada dimensión de las variables de optimización en C secciones de igual tamaño, luego una configuración en el espacio continuo se mapea a su celda correspondiente.

- Variables de posición: Este tipo de variables consisten en tener un objeto o luz en alguna posición de un rectángulo. Por lo que se parte el rectángulo en $C \times C$ celdas.
- Variable de dirección de la luz: Las variables de dirección se manejan en coordenadas esféricas, partiéndose el rectángulo de dimensiones $2\pi \times \pi$ en $C \times C$ celdas.
- Variable de color: Las variables de color varían su valor. Entonces se divide el intervalo del Valor HSV $[0, 1]$ en C segmentos.

El VNS se ejecuta de la misma manera que en el caso continuo, con la salvedad de que antes de evaluar un elemento se busca la *celda* a la que pertenece, y se usa un valor de dicha celda. El problema de optimización pasa a tener un carácter combinatorio: se trata de encontrar los valores $(x_1, \ldots, x_m)$ donde $0 \leq x_i < C, x_i \in \mathbb{N}\ \forall i = 1, \ldots, m$. En la Figura 4.11 puede verse la definición de celdas para un espacio continuo. El valor de la celda en el punto de referencia en b) (marcado como puntos blancos) define el valor de la función en la celda en a).

En el archivo de configuración el usuario debe definir el valor de C. La elección de este número representa un compromiso entre la exactitud de la solución encontrada y el rendimiento del algoritmo. Si el valor de C es muy pequeño se puede llegar a agotar el dominio de soluciones luego de pocas iteraciones.

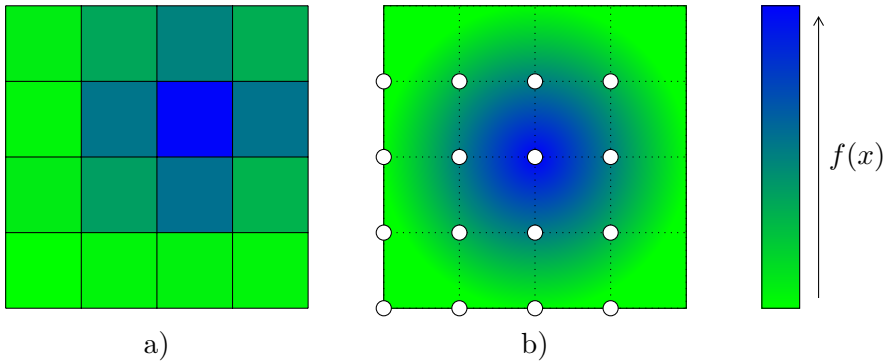


Figura 4.11: Definición de celdas de un espacio de soluciones en $\mathbb{R}^2$ con $C = 4$.

4.9. Optimizaciones de Rendimiento

A continuación se describen algunas optimizaciones hechas durante el transcurso del trabajo para mejorar el rendimiento del proceso de optimización.

4.9.1. Tabú search

Si se tiene dominio discreto, se puede almacenar el valor de $F_N(x)$ para evitar invertir tiempo en volver a calcular el valor cuando se evalúa una configuración. En este contexto se implementó un caché de largo plazo de todas las soluciones evaluadas, como se hace en la metaheurística de Tabú Search (Glover y Laguna, 2013), donde se guardan evaluaciones para que la heurística de búsqueda (en este caso VNS) vuelva a evaluar lugares que ya fueron considerados. En este sentido en la implementación, se tiene diccionario $M : X \to \mathbb{R}$ que guarda todas las soluciones evaluadas hasta el momento. Al evaluar la función $F_N(x)$ se verifica que x esté en M, si no lo está se evalúa y se guarda su resultado en M como se ve en el Algoritmo 4.1. La función *calcularCelda* devuelve la configuración correspondiente a la celda a la que pertenece x y *evaluar* da un valor para $F_N(x)$.

Algoritmo 4.1 Algoritmo de cálculo con cache, de $F_N(x)$

```
evaluarConTabuYCelda(x, N):
x̄ ← calcularCelda(x)
if x̄ ∉ M then
  y ← evaluar(x̄, N)
  M[x̄] ← y
end if
return M[x̄]
```

4.9.2. Uso parcial de photon mapping

Debido a que para hallar el valor de $F_N(x)$ no es necesario construir el mapa de fotones, puede utilizarse solamente el *buffer* de fotones para generar el valor de $F_N(x)$. El mapa de fotones sólo es útil cuando se genera una renderización de la escena, para estimar la irradiancia, solamente es necesario recorrer el *buffer* de fotones y ver cuáles fotones corresponden a la superficie de interés. Debido a que cada fotón contiene una referencia a la superficie en la cual impactó (que es independiente de su posición de la escena, usada para la renderización), se puede hacer un filtro en el *buffer* por el valor de la superficie, omitiendo el mapa de fotones completamente. Esto hace que el proceso de optimización se reduzca considerablemente, debido a que no se arma una estructura de mapa de fotones.

5

Pruebas y Resultados Obtenidos

El objetivo de este capítulo es presentar los resultados obtenidos durante el transcurso de este trabajo. En primer lugar se muestran las escenas que se usaron para probar la implementación. Junto a los resultados numéricos del proceso de optimización también se mostrarán imágenes de cada una de las soluciones encontradas. Se analizará cuánto tiempo se invierte en cada una de las etapas del proceso. Finalmente se compara la eficiencia de la implementación en diferentes sistemas, y según la complejidad de la escena.

5.1. Escenas de Prueba

Se probó la implementación con 3 escenas con distinta cantidad de polígonos. Su propósito es probar las diferentes variables de optimización implementadas (posición de luces, de objetos, aberturas y color de una superficie), y demostrar los distintos tipos de iluminación y de superficies.

- *Cornell Box* (Figura 5.1). Es una representación de una Cornell Box, una escena de prueba usada generalmente para medir cualitativamente la precisión de un método de renderización, comparando el resultado de una escena generada contra una foto tomada de la misma escena. Fue creada por Donald P. Greenberg y otros estudiantes de la Universidad de Cornell. La escena fue usada para un artículo presentado en una Conferencia de *SIGGRAPH*[1] (Battaile, Goral, Greenberg, y Torrance, 1984). La escena que se usó en este proyecto es ligeramente diferente, dado que es cerrada. Se tienen varios cubos y prismas, además de dos esferas: una espejada y otra de material cristalino. Finalmente se tiene un cono que se usará en las pruebas. La única fuente luminosa es un panel en el techo de la que tiene un componente emitivo. La escena tiene 16428 polígonos.

- *Sponza Atrium* (Figura 5.2). Es una representación simplificada del atrio de Sponza ubicado en Dubrovnik, Croacia. Por la complejidad de la geometría del modelo es muy útil para probar métodos de iluminación global. La escena fue creada inicialmente por Marko Dabrovic en el 2002, y la versión utilizada se basa en el trabajo de Frank Meinl y está disponible en (McGuire, 2014). El modelo fue modificado para reducir la cantidad de caras, y tiene 102738 polígonos.

- *Sala de Conferencias*. La tercera escena es una representación de una sala de reunión del edificio del *Lawrence Berkeley National Laboratory*, creado por Anat Grynberg y

[1] Acrónimo de *Special Interest Group on GRAPHics and Interactive Techniques*, es el grupo de interés en infografía y computación gráfica en español y es también una conferencia anual creada por el grupo del mismo nombre.

Grer Ward, circa 1991. El modelo fue creado tomando medidas de la sala original y entradas manualmente en un editor de texto. Se usa comúnmente para experimentos de iluminación global. La versión que se usa fue modificada de la original y tiene texturas tanto en el piso como en el pizarrón, y tiene 644508 polígonos.

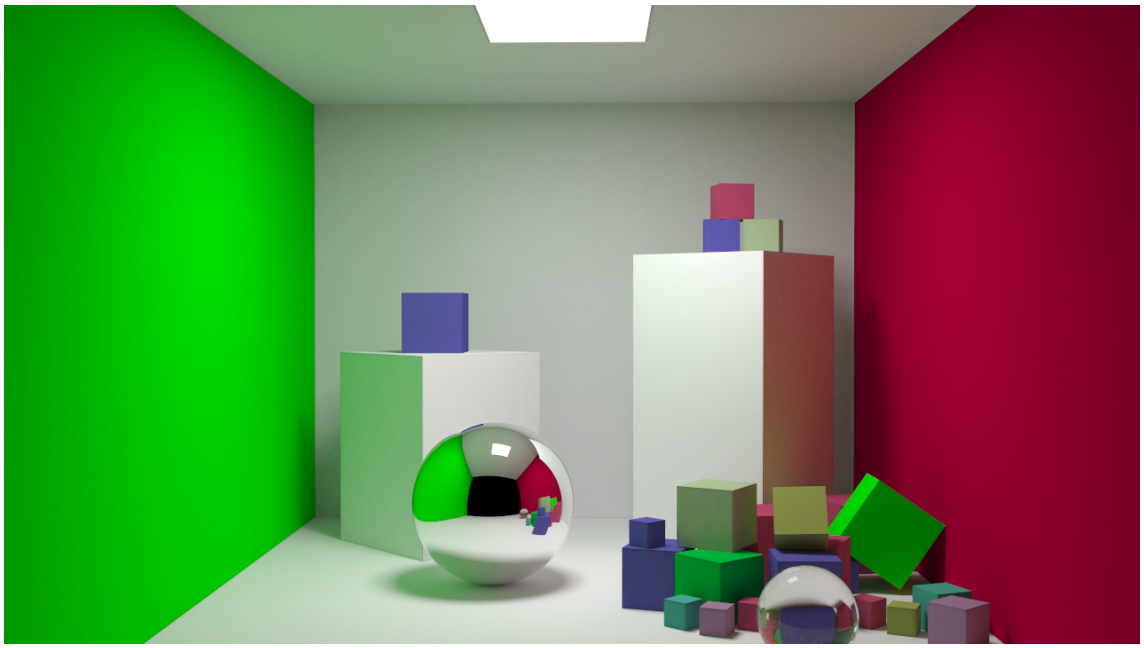


Figura 5.1: Imagen de la escena *Cornell Box* generada utilizando photon mapping progresivo

Figura 5.2: Imagen de la escena *Sponza* generada utilizando photon mapping progresivo

Figura 5.3: Imagen de la escena *Sala de Conferencias* generada utilizando photon mapping progresivo

5.2. Hardware de Prueba

Se probó la aplicación de ILPSolver en tres sistemas distintos. Estos sistemas fueron elegidos teniendo en cuenta la diversidad. A priori el más rápido de ellos es el Equipo 1, que es un computador de escritorio del 2014. Seguido por una portátil de alto rendimiento del mismo año. El tercero es una computadora de escritorio más antigua. En el Cuadro 5.1 se detallan las especificaciones de ambos sistemas. Una diferencia importante es que el Equipo 3 tiene menor memoria de vídeo que los otros dos ordenadores, lo cual hace que soporte mapa de fotones más pequeño. Esto causa que la calidad de las soluciones obtenidas sea menor.

	Equipo 1	Equipo 2	Equipo 3
Modelo CPU	Intel Core i5 3475	Intel Core i7 4710	Intel Core i5 2400
Frec. CPU	3200MHz	2500MHz	3100Mhz
Memoria RAM	8192MB	16384MB	8192MB
Modelo GPU	GeForce GTX 760	GeForce GTX 860M	GeForce GTX 480
Memoria GPU	2048MB	2048MB	1536MB
Frecuencia GPU	980Mhz	1019MHz	701MHz
Núcleos CUDA GPU	1152	640	480
Rendimiento bruto GPU	2258 GFLOPS	1306 GFLOPS	1345 GFLOPS
Sistema Operativo	Windows 10	Windows 10	Windows 8.1

Cuadro 5.1: Especificaciones de los equipos usados en las pruebas

5.3. Problemas de Prueba

En esta sección se describen los problemas de prueba para comprobar el funcionamiento del programa. El objetivo de cada uno de ellos es probar al menos un tipo de variable de optimización (véase Sección 4.1). Se apunta a describir problemas cuya solución sea verificable viendo renderizaciones de las soluciones que se generan.

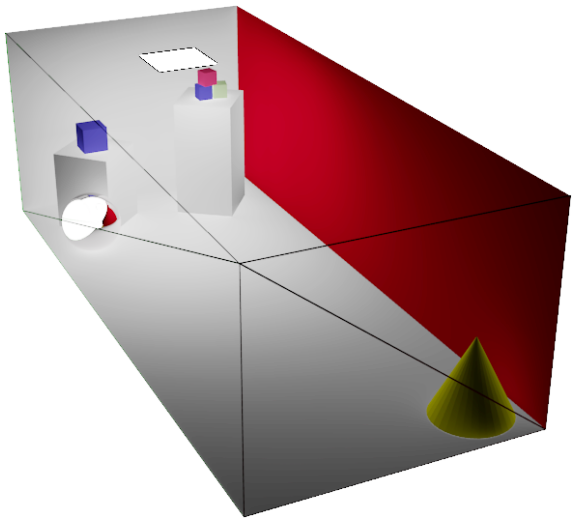


Figura 5.4: Imagen de la escena *Cornell Box* generada utilizando Blender. Se ocultan las caras de 3 de las paredes para poder ver el interior de la caja.

En la escena de Cornell Box se prueba el funcionamiento del algoritmo cuando se modifican variables que cambian la geometría de la escena y la posición de una fuente luminosa. El objetivo en ambos es maximizar la iluminación de cierto cono que está en la esquina de la caja. En el Sponza Atrium se tienen dos problemas: en uno se modifica la posición de una abertura, mientras que en el otro se cambia el color de una superficie. Finalmente, en la escena de la Sala de Conferencias se busca optimizar la irradiancia de un pizarrón usando una variable de luz direccional. En el Cuadro 5.2 se listan las escenas utilizadas.

	Nombre	Tipo de variable utilizada
1	Cono en Cornell Box	Posición de objeto
2	Luz en Cornell Box	Posición de luz
3	Abertura en Sponza Attrium	Posición de abertura
4	Color en Sponza Attrium	Color de superficie
5	Sala de Conferencias	Dirección de luz

Cuadro 5.2: Listado de los problemas de prueba definidos

Para cada problema se presentará su especificación que consta de las siguientes partes:

- Objetivo: Define de qué intención de iluminación se quiere alcanzar en el problema.
- Variables: Qué variables pueden modificarse para alcanzar el objetivo.
- Restricciones: Pueden ser de las variables o de la propia función objetivo.
- Iteraciones: La cantidad de iteraciones que ejecuta el VNS antes de terminar. Es el valor de *maxIteraciones* del Algoritmo 3.1.
- Tamaño del cuadrante: Define en cuantas partes se divide el espacio de soluciones para formar las grillas discretas. Es el valor de C, definido en la Sección 4.8.

5.3.1. Cono en Cornell box

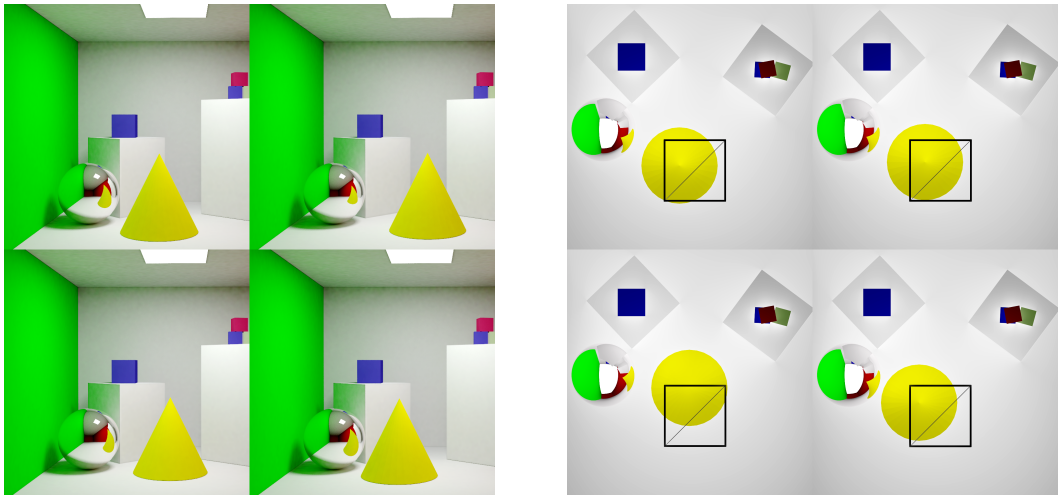
El problema es conseguir la mejor ubicación de un cono en la escena de la Cornell Box (Figura 5.1) de manera de que se maximice la irradiancia en su superficie. El cono

se posiciona inicialmente en la esquina más lejana de la caja. Se espera que el cono se ubique debajo de la fuente luminosa. El cono puede variar su posición siempre y cuando esté apoyado sobre el piso (no puede flotar en el aire). Las especificaciones del problema están definidas en el Cuadro 5.3.

Objetivo	Maximizar irradiancia del Cono
Variables	Posición del cono
Sujeto a	El cono debe estar en el piso
Escena	Cornell Box
Iteraciones	1000
Tám. del cuadrante	20

Cuadro 5.3: Definición del problema *Cono en Cornell Box*

Luego de ejecutar ILPSolver, las posiciones óptimas del cono se encuentran bajo la luz, como se esperaba. En la Figura 5.5 se pueden ver las soluciones que generaron durante una ejecución del algoritmo. En 5.5a se muestran las soluciones vistas desde una cámara en perspectiva dentro del cubo. Las renderizaciones fueron generadas con photon mapping progresivo. En 5.5b se muestran las mismas soluciones vistas desde una proyección ortográfica, en este caso fueron generadas utilizando Blender. Se puede ver que las soluciones se ubican debajo de la luz, pero ligeramente cerca de los prismas. Esto se debe a que la reflexión de la fuente luminosa en los prismas ayuda a que el cono obtenga más irradiancia debido a los rebotes que los rebotes de los fotones en los prismas impactan sobre la superficie el cono.



(a) (b)

Figura 5.5: Visualización de la configuración óptima encontrada durante un experimento de *Cono en Cornell Box*.

En la Figura 5.6 se graficaron los resultados intermedios generados usando ILP durante el proceso de primera mejora (véase Sección 2.10.3).El punto de partida se marca como un cuadrado azul, que corresponde a la imagen de la Figura 5.4. Las soluciones intermedias que se encontraron durante el VNS se marcan como rombos naranjas. Se indican en círculos verdes al conjunto de soluciones óptimas resultado del algoritmo, que se muestran en la Figura 5.5. Se muestran los límites del dominio en rojo (posiciones posibles del cono), que coinciden con las paredes de la caja. El eje vertical indica el valor de la función objetivo, mientras que los otros dos indican la posición de las soluciones en el espacio. En la gráfica de la Figura 5.6 se puede ver que las soluciones encontradas tienden a formar una línea

que apunta hacia el conjunto de soluciones óptimas. Esto ocurre porque la función de iluminación en el cono crece de manera casi constante con la cercanía a la región donde están las soluciones óptimas por lo que el VNS encuentra mejoras en esa dirección.

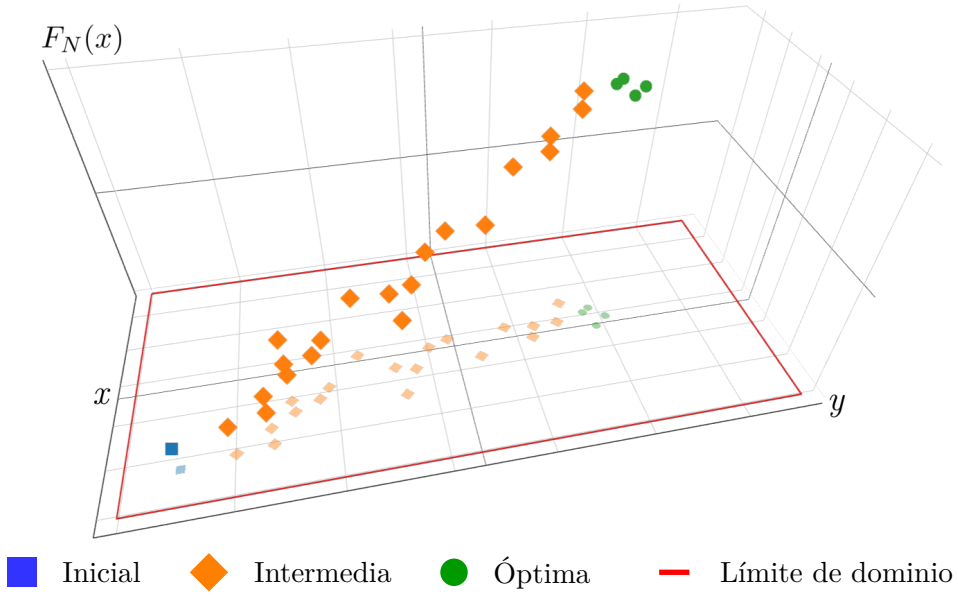


Figura 5.6: Gráfica de la distribución espacial de las soluciones intermedias generadas por el programa para una ejecución del experimento de Cono en Cornell Box

5.3.2. Luz en Cornell box

El objetivo de este problema es maximizar la luz incidente en el cono, pero moviendo de lugar el panel luminoso en el techo de la caja. El problema es muy similar al de *Cono en Cornell Box*, solamente que en vez de moverse el cono en este caso se mueve la luz. Mover una fuente luminosa es distinto trasladar un objeto, debido a que la luz no forma parte de la geometría: cambiar una luz de lugar implica solamente cambiar cómo se generan los fotones en la etapa de photon tracing. La fuente luminosa es un cuadrado que tiene un componente emitivo y por lo tanto genera fotones. En el problema, el panel puede ubicarse en cualquier punto del techo de la escena. El resultado esperado es que el panel se ubique sobre el cono se encuentra en la esquina de la escena. Las especificaciones del problema están definidas en el Cuadro 5.4.

Objetivo	Maximizar irradiancia del Cono
Variables	Posición de la luz
Sujeto a	La luz debe estar en el techo
Escena	Cornell Box
Iteraciones	1000
Tam. del cuadrante	20

Cuadro 5.4: Definición del problema *Luz en Cornell Box*

Luego de ejecutar el ILPSolver el panel luminoso se posiciona sobre el cono, moviéndose desde su posición inicial, hacia la esquina de la escena. Las soluciones resultantes se pueden ver en la Figura 5.7. En la ejecución realizada se encontraron dos soluciones muy similares. En la Figura 5.7b se tiene la imagen vista desde arriba en proyección ortogonal, de ambas soluciones. El panel se encuentra ligeramente más cerca de las paredes que del centro

geométrico del cono. Esto se puede deber a que se aprovecha los rebotes de los fotones en las paredes, aumentando la irradiancia de la parte que está más contra la esquina de la escena. En la Figura 5.7a se muestra el resultado de una renderización en proyectiva, hecha con photon mapping progresivo.

(a) (b)

Figura 5.7: Visualización de las configuraciones óptimas encontradas durante el experimento de *Luz en Cornell Box*.

En la gráfica de la Figura 5.8 se ilustra la distribución espacial de las soluciones encontradas, donde se ven las soluciones finales (las soluciones al problema), las soluciones intermedias, el punto de partida en naranja, y el límite de la región de configuraciones factibles, que coincide con el techo de la habitación. La forma en la que las soluciones se distribuyen es similar al experimento de Cono en Cornell Box. Principalmente porque la función de iluminación crece de manera similar y al aplicar VNS se encuentra una dirección de mejora de manera relativamente fácil.

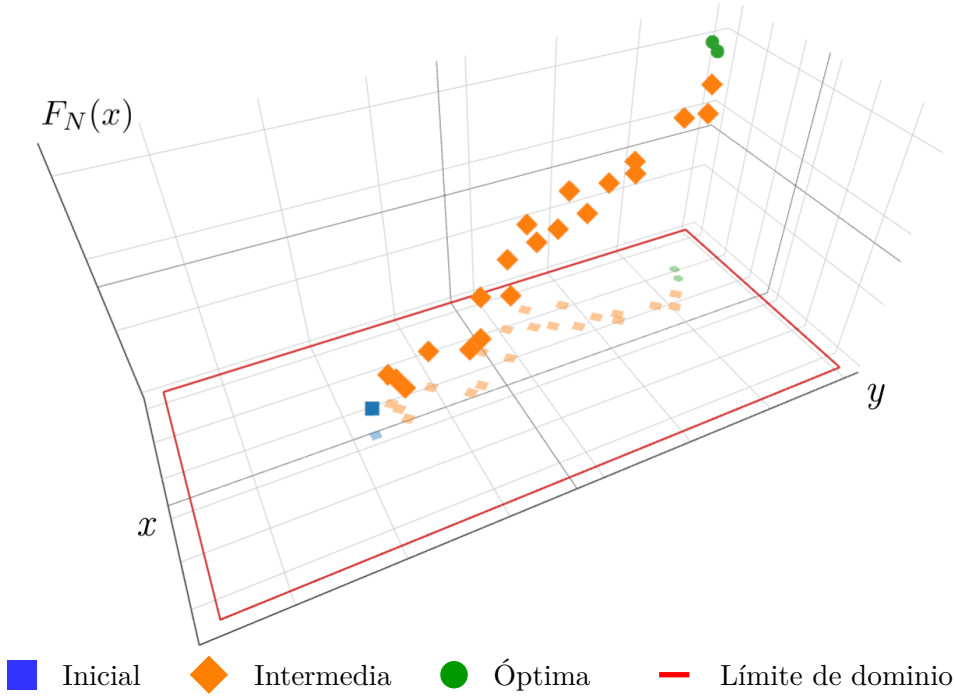


Figura 5.8: Gráfica de la distribución espacial de las soluciones intermedias generadas por el programa para el experimento de Luz en Cornell Box

5.3.3. Abertura en Sponza atrium

Objetivo	Maximizar irradiancia de la jarra
Variables	Posición del abertura
Sujeto a	El abertura debe estar en el rectángulo delimitado en el Techo
Escena	Sponza Atrium
Iteraciones	1000
Tam. del cuadrante	20

Cuadro 5.5: Definición del problema *Abertura en Sponza Atrium*

El objetivo de esta escena es optimizar la posición de una abertura en el techo de la escena, de forma tal de iluminar una jarra que se encuentra en el medio de la escena, sobre el piso. La escena tiene 2 luces puntuales dentro del atrio y una luz direccional vertical girada 15 grados en dirección del eje longitudinal del atrio. La solución óptima es la posición de la abertura de forma que la luz direccional exterior pase por ella e ilumine la jarra. En la Figura 5.9 se muestra una vista de la escena en modo de marco de alambre, resaltando la superficie donde puede posicionarse la abertura. Esta es cuadrada y mide aproximadamente la tercera parte del ancho del rectángulo del dominio. Las especificaciones del problema se muestran en el Cuadro 5.5.

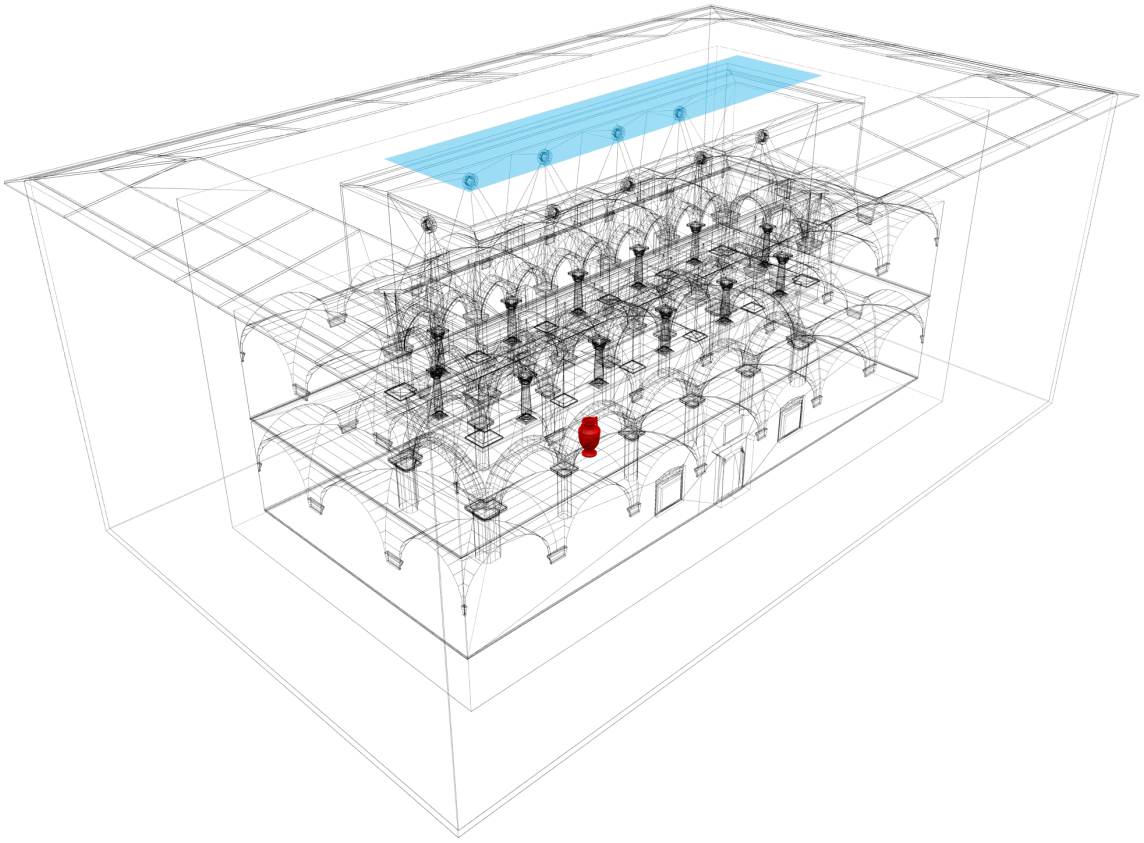


Figura 5.9: Vista exterior de la escena Sponza Atrium, en modo de marco de alambre (*wireframe*). Se resalta la superficie donde puede posicionarse la abertura, y en rojo opaco a la jarra que desea iluminarse.

Al ejecutar ILPSolver el problema converge a las soluciones óptimas, que pueden verse en la Figura 5.10. Las soluciones comparten que la abertura está posicionada de forma tal de que la luz ilumina directamente a la jarra.

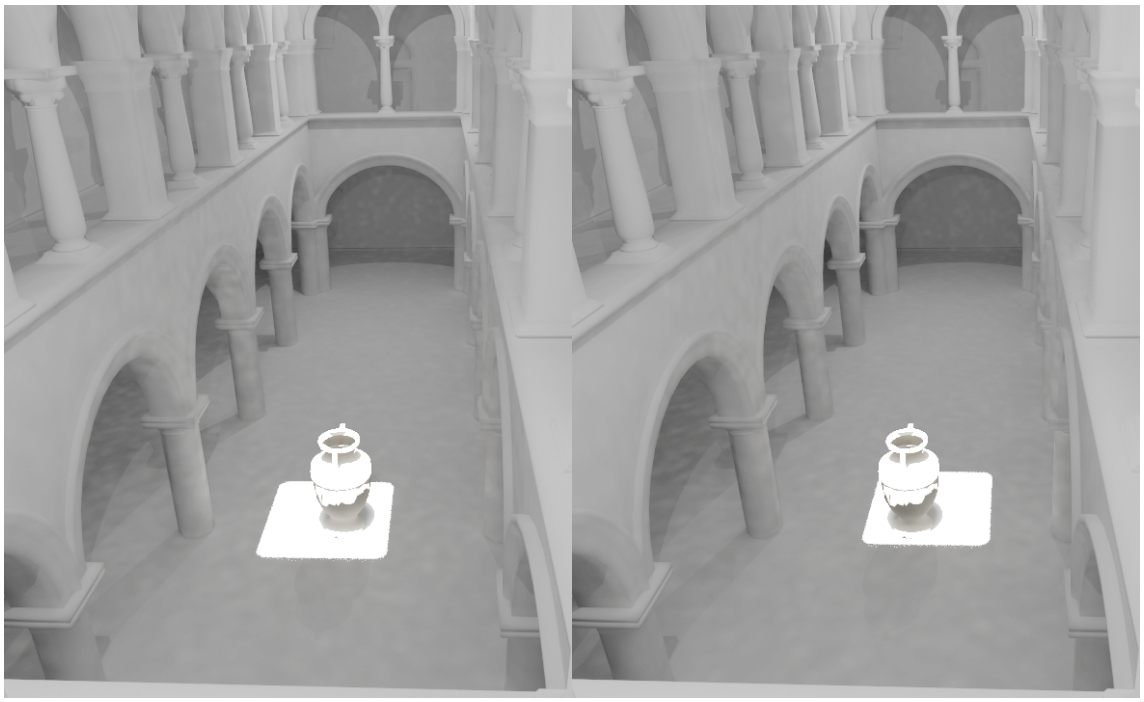


Figura 5.10: Imágenes de las soluciones encontradas para el experimento Abertura en Sponza Atrium, generadas utilizando photon mapping progresivo.

Una de los desafíos del problema es que la iluminación que recibe la jarra no varía de manera significativa al cambiar la posición la abertura, por lo que el VNS tarda un poco más en encontrar direcciones de ascenso comparado con los experimentos de Cornell Box, dado que las mejoras son suficientemente significativas cuando la luz se encuentra cerca de la posición ideal. En la Figura 5.11 se pueden ver cómo están distribuidas las soluciones que encontró el VNS. El límite del dominio es el indicado en la Figura 5.9. Como puede verse, el valor de la función objetivo cambia de manera significativa en el entorno del ISOC, pero no varía demasiado lejos de esa área. Las configuraciones en esas posiciones hacen que la jarra esté en la penumbra.

5.3.4. Color en Sponza atrium

Objetivo	Maximizar irradiancia del piso
Variables	Color de las paredes
Sujeto a	El color de las paredes debe tener tono *HSV* 39 grados, y saturación del 23 %. La irradiancia del piso no debe superar los $3 \times 10^{-4} W/m^2$
Escena	Sponza Atrium
Iteraciones	100
Tam. del cuadrante	50

Cuadro 5.6: Definición del problema *Color en Sponza Atrium*

Esta escena intenta optimizar el valor del color (el componente difuso K_d) de una superficie de manera tal de maximizar la irradiancia en otra. Tiene algunas diferencias con la anterior: no está la jarra, ni una abertura en el techo, la escena tiene dos luces interiores puntuales que iluminan de manera más o menos uniforme el piso inferior de la escena, y el piso tiene una textura que simula ser un piso de baldosas. Los objetivos

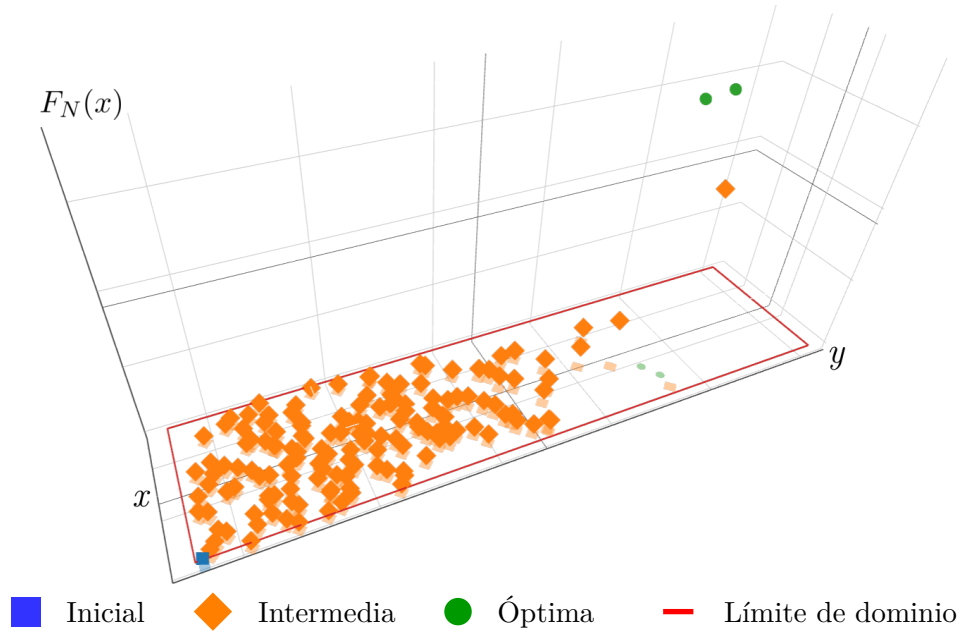


Figura 5.11: Gráfica de las soluciones intermedias encontradas por ILPSolver

concretos de la escena se muestran en el Cuadro 5.6. En el problema se limita el valor de la irradiancia del piso. Dado que la única función de optimización implementada es maximizar la irradiancia de una superficie, si no se restringe el Color de una superficie, su solución siempre va a ser la que tenga Valor HSV de 100 %. Eso ocurre porque un fotón no adquiere energía al interactuar con las superficies: mantiene su energía o la pierde. Por esta razón siempre va a ser conveniente el color más "claro" si se quiera maximizar la irradiancia de una superficie. Por eso, se restringe también el valor de la función objetivo, para que se elija un color pero cuyo Valor HSV óptimo para cumplir las restricciones sea menor que 100 %. En la Figura 5.12 se puede ver la solución que encontró ILPSolver al problema planteado.

Figura 5.12: Imagen de la solución encontrada para el experimento Color en Sponza Atrium, generada utilizando photon mapping progresivo.

Objetivo	Maximizar irradiancia del pizarrón
Variables	Dirección de la luz direccional exterior
Escena	Sala de Conferencias
Iteraciones	1000
Tam. del cuadrante	20

Cuadro 5.7: Definición del problema *Sala de Conferencias*

5.3.5. Sala de conferencias

La intención de este experimento es probar la optimización de una luz direccional. A diferencia de las luces puntuales, las luces direccionales tienen su origen en el infinito. El problema puntual que se resuelve es encontrar la mejor posición de la luz de forma tal de iluminar el pizarrón que se encuentra sobre la pared más lejana a la cámara en la Figura 5.13. Una aplicación práctica para este problema sería saber en qué momento del día la sala de conferencias recibe más luz, siendo el Sol la luz direccional. La luz puede entrar en la habitación a través de una ventana que está sobre la izquierda. Además la sala de conferencias tiene 8 paneles que emiten luz sobre el techo, por lo que se totaliza en 9 fuentes luminosas. La escena tiene una textura que determina el color del pizarrón. Las especificaciones del problema se determinan en el Cuadro 5.7. Se espera que la solución del problema sea una dirección de la luz que pueda pasar por el ventanal e ilumine directamente el pizarrón. Esa luz debe venir de la izquierda y de manera casi horizontal para evitar generar sombra sobre el pizarrón por el techo o el borde inferior del ventanal.

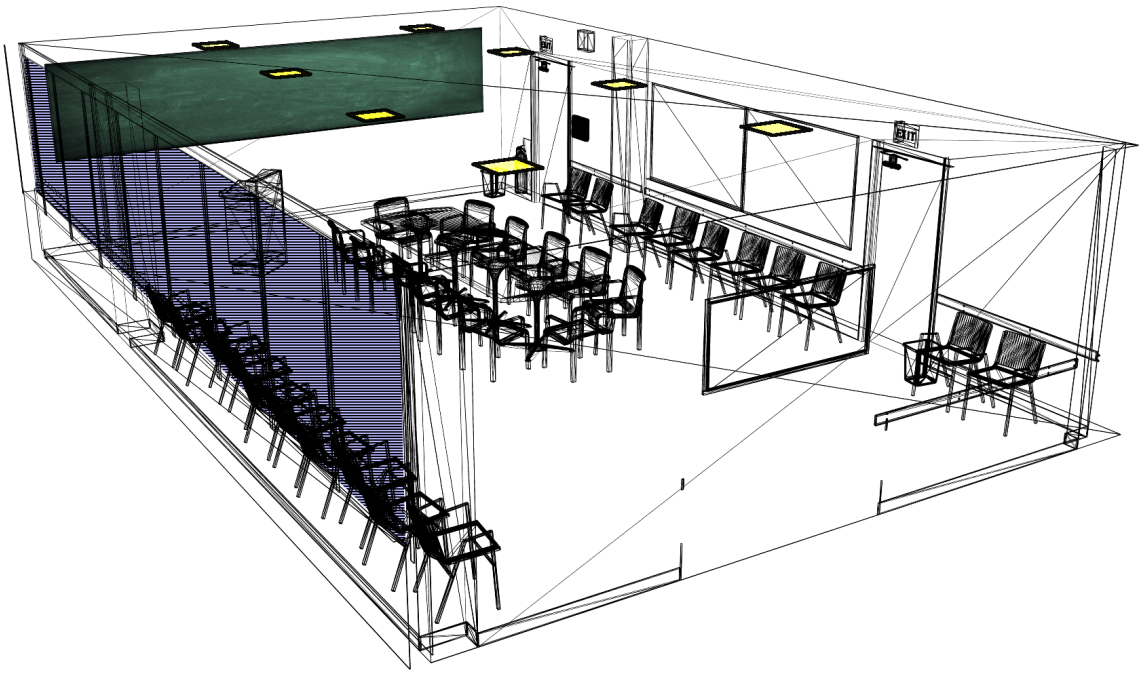


Figura 5.13: Representación de la escena de Sala de Conferencias en marco de alambre. El pizarrón se muestra con su textura original, se marca en amarillo a los paneles luminosos del techo, y a la izquierda en azul el espacio que ocupa el ventanal.

Luego de ejecutar el ILPSolver se encontró una solución que coincide con lo que se esperaba. La luz pasa por el ventanal e ilumina el pizarrón. Se puede ver en la renderización de la Figura 5.14b la sombra del marco de la ventana en la pared y el pizarrón. En la Figura 5.14a se muestra la dirección de la luz exterior marcada con la flecha 3d.

A pesar de que en algunos problemas puede ser útil buscar direcciones sin restricciones,

el VNS prueba todas las direcciones, incluso aquellas que no tienen sentido físico, como aquellas que vengan debajo del horizonte. Pero para el Sol es beneficioso implementar restricciones, por ejemplo, para permitir sólo direcciones provenientes de un hemisferio. Esto aceleraría el proceso de optimización evitando evaluaciones innecesarias.

(a) (b)

Figura 5.14: Visualización de las configuraciones óptimas encontradas durante el experimento de *Sala de Conferencias*.

5.4. Análisis de Rendimiento

En el proceso de optimización la mayoría del tiempo se dedica a generar el mapa de fotones de la escena. En esta sección se compara el rendimiento de las tres escenas que se presentaron en la Sección 5.1. El objetivo es conocer cuánto tiempo se invierte en cada actividad del proceso de optimización. Luego de ejecutar los experimentos se encontraron tres procesos que invierten la gran parte del tiempo:

- Redimensionamiento de *buffers*: Es el acto de cambiar los tamaños de los *buffers* que usa la GPU. Estos *buffers* se usan para almacenar los fotones y otras estructuras de datos, utilizadas durante el photon tracing. Cada vez que es necesario evaluar una solución con una cantidad de fotones diferente se redimensiona los *buffers* de la GPU. Por ejemplo cuando se evalúa una configuración de manera rápida para saber si está en el ISOC, y luego para confirmarlo se la vuelve a evaluar con más fotones.

- Photon tracing: Engloba el proceso de generar fotones desde las fuentes luminosas, introducido en la Sección 2.6.1.

- Cálculo de impactos: Implica recorrer el *buffer* de fotones que se generó en la etapa de photon tracing para contar los impactos que tuvo cada superficie, de forma de poder calcular la irradiancia de la solución.

Las pruebas se hicieron en los tres equipos descritos en el Cuadro 5.1. Para cada experimento se tomó la proporción del tiempo invertido en cada parte sobre el tiempo total de ejecución, haciéndose 10 veces y tomándose promedios de los tiempos. El tiempo total es el tiempo desde que se inicia la optimización del VNS hasta que se termina. No se tomó en cuenta el tiempo dedicado a interpretar la escena, ni el tiempo que se invierte luego cuando finaliza el programa. Lo que sorprende del resultado, mostrado en la Figura 5.15, es que el hecho de cambiar el tamaño de los *buffers* lleve más tiempo que el cálculo de impactos, que implica justamente, recorrer el *buffer* de fotones. El redimensionamiento de un *buffer* implica borrar su contenido y volver a reservar espacio, que es muy poco

eficiente. Esto indica que el programa se podría beneficiar de no tener que cambiar el tamaño de los *buffers* cada vez que se cambia la cantidad de fotones a lanzar. Se puede reservar espacio y evitar redimensionar al cambiar la cantidad de fotones lanzados.

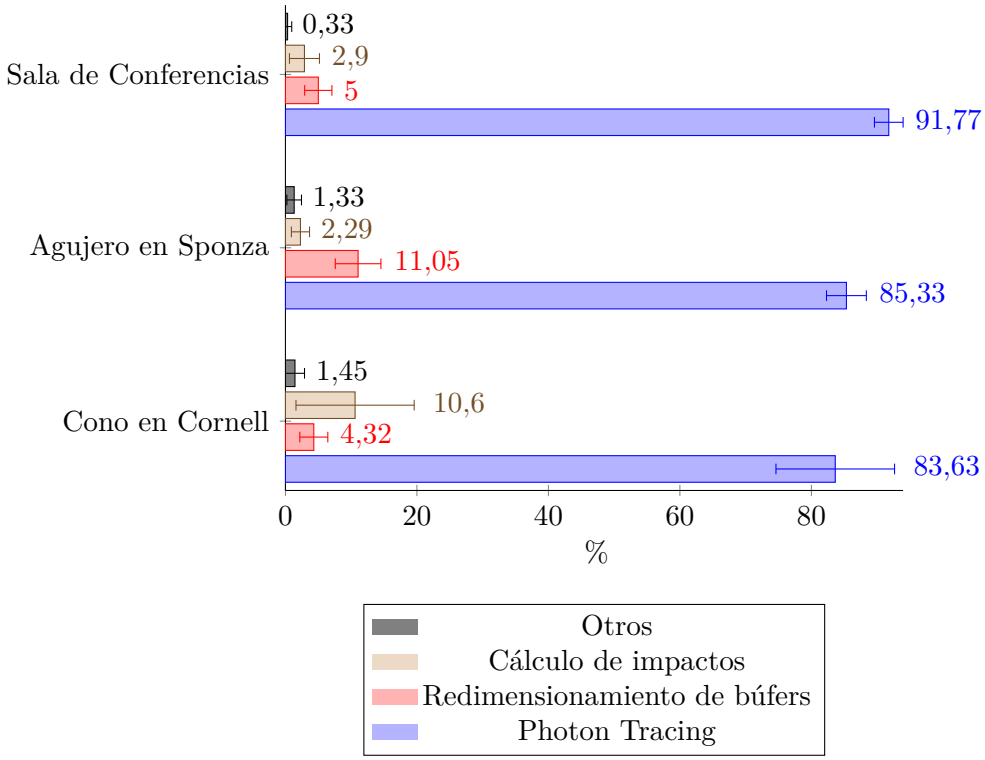


Figura 5.15: Proporción del tiempo dedicado a cada etapa de la optimización

Adicionalmente se hizo un análisis del rendimiento individual de cada equipo en cada una de las escenas. Los datos son los mismos que en el ejemplo anterior, tomando el promedio de tiempo por iteración que tomó el VNS. En la gráfica se puede apreciar que el tiempo por iteración del equipo 3 es ligeramente mejor al de los demás. Esto ocurre porque el equipo 3 tiene menor memoria que los otros dos, lo cual implica que se lanzan menos fotones por iteración, como se explica en la Sección 4.7. Cada iteración es más rápida por la menor cantidad de fotones, pero el margen de error de las estimaciones es mayor, lo cual hace que las soluciones sean de menor calidad. De todos modos los equipos 1 y 2 tienen la misma cantidad de memoria en la GPU (2 GB), lo cual hace que se lancen la misma cantidad de fotones y sus rendimientos sean comparables. La gráfica con los tiempos por iteración puede verse en la Figura 5.16.

5.5. Discusión

El objetivo de esta sección es discutir algunos de los resultados encontrados durante el armado y la ejecución de los problemas de prueba.

Las escenas con una alta cantidad de polígonos y de texturas no pueden ser cargadas en ILPSolver por restricciones de memoria. El problema es que el programa requiere una gran cantidad de memoria de GPU para funcionar. Tanto las estructuras de aceleración de OptiX, cuyo tamaño depende de la escena, como el *buffer* de fotones, y otras estructuras necesitan residir en la memoria de la GPU durante todo momento. Además si se definen múltiples objetos con la misma textura, ILP solver OppositeRenderer carga una copia de cada textura para cada nodo de la escena que la defina, lo que es ineficiente. Por este motivo, la escena de Sponza Atrium fue modificada de forma de usar texturas en un

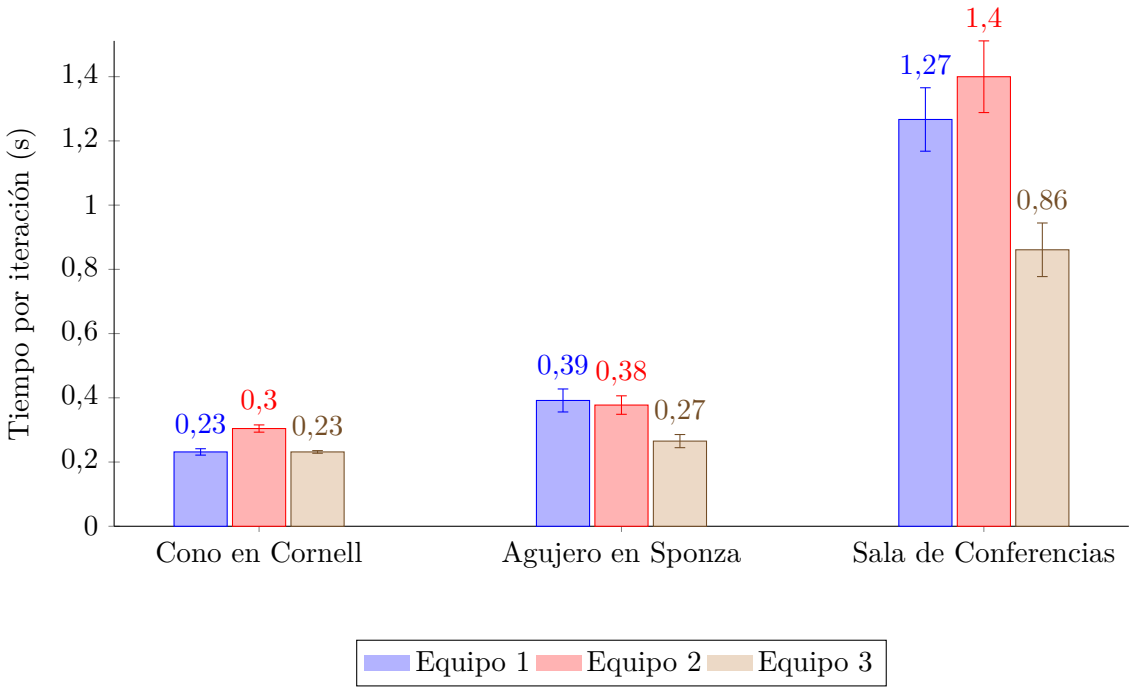


Figura 5.16: Comparación del tiempo por iteración para cada equipo.

número acotado de objetos.

Otra limitación asociada con OptiX y con las estructuras de aceleración, es que la cantidad de polígonos de cada objeto de la escena tiene que ser acotada. De lo contrario pueden ocurrir errores de desbordamiento de pila en el código de intersección rayo-geometría. Como se vio anteriormente el manejo de memoria es no es eficiente cuando se hace el redimensionamiento de *buffers*. Una alternativa que podría manejarse es reservar espacio para los *buffers* del mapa de fotones al inicio del programa y no usar OptiX para redimensionar los *buffers*.

Cuando un ILP tiene múltiples variables de optimización el tiempo de cómputo aumenta considerablemente, debido a que se modelan como independientes, y por tanto cada combinación que prueba el VNS es el producto de las combinaciones de las variables de optimización. Se puede mitigar el tiempo que tarda el programa partiendo el espacio de soluciones en trozos más pequeños (usando un valor de C más pequeño), pero al costo de tener soluciones de menor calidad.

6 Conclusiones y Trabajo Futuro

6.1. Conclusiones

En este trabajó se logró implementar un programa que resuelve un problema de iluminación inversa utilizando photon mapping y la biblioteca OptiX. El proceso funciona de forma relativamente rápida y tiene ventajas sobre el uso del método de radiosidad al poder manejar cambios en la geometría, en la reflectividad, texturas, y superficies con componentes no difusos. Se utilizó un paquete que implementa photon mapping progresivo, para implementar el método photon mapping. Luego de disponer de un método basado en photon mapping, se lo modificó para generar información de la escena y también para permitir cambiar aspectos del modelo original, con el objetivo de poder evaluar distintas configuraciones. Finalmente se implementó la heurística de VNS, en el programa que fuera capaz de resolver problemas inversos de iluminación. Se desarrolló una variante estocástica de optimización para manejar múltiples soluciones indistinguibles entre sí. Se priorizó la posibilidad de extenderla agregando nuevas variables de optimización. ILPSolver se integró con el paquete de renderización OppositeRenderer para tener la solución final, que permite resolver ILPs y que puede ser extendida de manera relativamente fácil. Se estudió la distribución de probabilidad de las evaluaciones y modifico el VNS para que tomara en cuenta que las soluciones no son exactas y tienen un margen de error. Luego, se trató de hacer más eficiente el proceso de optimización, aplicando pequeñas mejoras al rendimiento, como la utilización parcial del proceso de photon mapping y el uso de Tabú Search.

El código del proyecto fue publicado para uso del público en general y documentado. El código fue probado en diferentes tipos de máquinas, y versiones de sistemas operativos, por lo que se puede considerar que la solución es portable y compatible. Durante el proceso de desarrollo se utilizaron las últimas versiones de las bibliotecas disponibles de OptiX, CUDA.

6.2. Trabajo Futuro

En primer lugar se puede paralelizar el proceso de optimización utilizando varios equipos ejecutando en manera paralela, aprovechando la implementación original de OppositeRenderer. Luego se puede mejorar el manejo de la memoria de la GPU. Se encontró que el programa ILPSolver no maneja de manera correcta escenas muy complejas. Si bien las escenas que se utilizaron tienen una complejidad bastante grande, tuvieron que ser modificadas para poder ser utilizadas en el programa.

Otros aspectos a ser mejorados incluyen soportar más tipos de variables de optimización, como poder rotar objetos de la geometría, cambiar las dimensiones de una abertura, entre otros. También se pueden agregar nuevos criterios de optimización, como por ejem-

plo la irradiancia percibida por un observador en cierto punto de la habitación, en cierta dirección. El manejo eficiente de problemas con múltiples variables de optimización puede ayudar a que ILPSolver resuelva problemas más complejos. Otro tipo de mejoras posibles es el agregado de parámetros de configuración que permitan al usuario controlar mejor cómo se realiza la optimización. Por ejemplo poder controlar mejor cuál es el intervalo de confianza que se busca en las soluciones. Otro aspecto que se puede implementar es el ajuste de la cantidad de fotones de forma automática, según el valor de la función objetivo, de forma de no lanzar una cantidad innecesariamente grande de fotones para distinguir las soluciones. Finalmente puede evitarse el redimensionado de los *buffers* de fotones cuando se cambia la cantidad de fotones generados en cada iteración. Esto acelerará el proceso de evaluación de cada solución generada.

Referencias

Arvo, J., Fajardo, M., Hanrahan, P., Jensen, H. W., Mitchell, D., y Shirley, P. (2001, agosto). *State of the art in monte carlo ray tracing for realistic image synthesis* (Inf. Téc.). Los Angeles: SIGGRAPH 2001.

Arvo, J., y Kirk, D. (1990, agosto). Particle transport and image synthesis. *Computer Graphics*, *24*(4), 63–66.

Avas, I. (2015). *ILPSolver.* Descargado 2015-12-04, de `https://github.com/igui/ILPSolver`

Battaile, B., Goral, C. M., Greenberg, D. P., y Torrance, K. E. (1984, julio). Modeling the interaction of light between diffuse surfaces. *Computer Graphics*, *8*(13), 213-222.

Bell, N., y Hoberock, J. (2011). Thrust: A Productivity-Oriented Library for CUDA. En W. W. Hwu (Ed.), *GPU Computing Gems* (Jade ed., p. 359-371). Morgan Kaufmann.

Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM*, *18*(9), 509–517.

Blender Foundation. (2015). *About blender.org.* Descargado 2015-11-22, de `https://www.blender.org/about/`

Blum, C., y Roli, A. (2003, septiembre). Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys (CSUR)*, *35*(3), 268-308.

Bourke, P. (2012). *Object Files (.obj).* Descargado 2015-08-04, de `http://paulbourke.net/dataformats/obj/`

Contensin, M. (2002). Inverse lighting problem in radiosity. *Journal of Inverse Problems in Engineering*(2), 131-152.

Crane, K. (2014, 24 de enero). *Bias in rendering.* Descargado 2015-07-19, de `http://www.cs.columbia.edu/~keenan/Projects/Other/BiasInRendering.pdf`

Elias, H. (2000). *TGLTLSBFSSP: Radiosity.* Descargado 2015-05-31, de `http://freespace.virgin.net/hugo.elias/radiosity/radiosity.htm`

Feller, W. (1950). *An introduction to probability theory and its applications* (3ra ed., Vol. I). John Wiley and Sons.

Fernández, E. (2014). *Efficient global illumination calculation for inverse lighting problems* (Tesis de Doctorado en Informática). Facultad de Ingeniería, Universidad de la República, Montevideo.

Fleisz, M. (2009). *Photon mapping on the GPU* (Master of Science). University of Edinburgh, Ireland.

Glover, F., y Laguna, M. (2013). Tabu search. En P. M. Pardalos, D.-Z. Du, y R. L. Graham (Eds.), *Handbook of combinatorial optimization* (p. 3261-3362). Springer New York.

Hachisuka, T., Ogaki, S., y Jensen, H. W. (2008, diciembre). Progressive photon mapping. En *Acm transactions on graphics (tog)* (Vol. 27). ACM.

Jarosz, W., Jensen, H. W., y Donner, C. (2008). Advanced global illumination using photon mapping. En ACM (Ed.), *Siggraph 2008 classes.* Los Angeles.

Jensen, H. W. (1996). Rendering caustics on non-lambertian surfaces. En *Proceedings of graphics interface '96* (pp. 116–121).

Jensen, H. W. (2001). *Realistic image synthesis using photon mapping* (1ra ed.). Natick: A K Peters.

Kajiya, J. T. (1986, agosto). The rendering equation. En *SIGGRAPH '86* (Vol. 20, pp. 143–150). Dallas.

Kreinin, Y. (2011, 10 de noviembre). *SIMD <SIMT <SMT: parallelism in NVIDIA GPUs.* Descargado 2015-06-24, de `http://yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html`

Krishnamachari, P. (2006, 30 de mayo). *Global illumination in a nutshell.* Descargado 2015-05-30, de `http://www.thepolygoners.com/tutorials/GIIntro/GIIntro.htm`

Luenberger, D. G., y Ye, Y. (2008). *Linear and non linear programming* (3ra ed.). Springer.

McCorquodale, J. (2001, mayo). *Radiosity homework.* Descargado 2015-06-15, de `http://www.cs.utah.edu/~mcq/radiosity/`

McGuire, M. (2014). *Computer graphics data archive.* Descargado 2015-11-26, de `graphics.cs.williams.edu/data/meshes.xml`

Mladenović, N., y Hansen, P. (1997, noviembre). Variable neighborhood search. *Computers and Operations Research*, *24*(11), 1097–1100.

NVIDIA Corporation. (2014, enero). NVIDIA OptiX^{TM} Ray Tracing Engine Programming Guide [Manual de software informático].

NVIDIA Corporation. (2015, marzo). CUDA C Programming Guide [Manual de software informático].

Osman, I., y Laporte, G. (1996). Metaheuristics: A bibliography. *Annals of Operational Research*, *63*, 513-628.

Patow, G., y Pueyo, X. (2003). A survey of inverse rendering problems. *Computer Graphics Forum*, *22*(4), 663-687.

Pedersen, S. A. (2013). *Progressive photon mapping on GPUs* (Master of Science in Computer Science). Norwegian University of Science and Technology, Trondheim.

Pixar. (2011, enero). *Renderman 19 documentation.* Descargado 2015-06-13, de `http://renderman.pixar.com/resources/current/RenderMan/globalIllumination.html`

Pollard, D. (2001). *A user's guide to measure theoretic probability* (Vol. 8). Cambridge University Press.

Purcell, T. J., Donner, C., Cammarano, M., Jensen, H. W., y Hanrahan, P. (2003). Photon mapping on programmable graphics hardware. En *ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (pp. 41–50). Eurographics Association.

Qt Company Ltd. (2015). *Qt for application development.* Descargado 2015-11-22, de `http://www.qt.io/application-development/`

Saleh, B. E., y Teich, M. C. (2007). *Fundamentals of photonics* (2da ed.). New York: John Wiley and Sons.

Stich, M., Friedrich, H., y Dietrich, A. (2009, agosto). Spatial splits in bounding volume hierarchies. En *Proc. high-performance graphics 2009.*

The Khronos Group Inc. (2008). *Collada – Digital Asset Schema Version 1.5.0.*

Universidad de Wisconsin. (2015). *NEOS: Optimization Taxonomy.* Descargado 2015-12-02, de `http://www.neos-guide.org/content/optimization-taxonomy`

Whitted, T. (1980). An improved illumination model for shaded display. *Graphics and Image Processing*, 343–349.

A

Estructuras de Datos Para el Mapa de Fotones

En este apéndice se describe las diferentes estructuras de datos que se pueden usar para almacenar los fotones en el mapa de fotones, comparándolas entre sí según que tan eficientes son para hacer un photon mapping usando la GPU.

A.1. Árboles kD

Un árbol kD es un árbol de búsqueda multi dimensional donde cada elemento se usa para partir una dimensión del espacio de búsqueda Cada nodo contiene un fotón, dos hijos, y una dimensión de partición. La dimensión de partición define un plano que pasa por ese fotón y es está alineado a uno de los ejes. Todos los elementos en el subárbol izquierdo tienen están en un lado del subespacio delimitado por el plano y todos los elementos del subárbol derecho pertenecen en el plano o están en el otro subespacio. Por ejemplo si un elemento del árbol es un fotón p con posición $P = (p_x, p_y, p_z)$ si la dimensión de división del nodo es y los elementos en el subárbol izquierdo cumplen que la su coordenada y es menor que p_y y los del subárbol derecho son tienen sus componentes y mayores o iguales a p_y. El caso de un árbol kD unidimensional es simplemente un árbol binario de búsqueda. Esta estructura es muy usada para photon mapping implementado en CPU porque es muy eficiente para buscar los k fotones más cercanos a un punto dado de la escena. El tiempo para realizar esa operación es de $O(k + \log n)$ en promedio. Si no se balancea el árbol luego de la etapa de photon tracing el árbol puede estar sesgado, teniendo ramas largas y enlenteciendo el proceso de búsqueda, y de renderización de la imagen final. Sin embargo balancear un árbol binario puede llevar un tiempo considerable. Por lo cual es bueno tener un equilibrio y ponderar si realmente es necesario realizar el pre procesamiento árbol. En Jensen (1996) se hacen experimentos con diferentes escenas y tamaños de mapas de fotones, llegando a la conclusión que volver a balancear el árbol tarda poco tiempo en comparación a lo que se ahorra luego en la etapa de renderización.

A.2. Grillas Uniformes

Una grilla 3D uniforme es una división de la escena en prismas de igual tamaño (*vóxels*). Cada celda de la grilla contiene aquellos fotones que espacialmente corresponden a esa grilla. Esta estructura tiene la ventaja de que es efectiva si los fotones están distribuidos más o menos de manera uniforme en la geometría. Sin embargo lo más usual es que los fotones se concentren más en algunas celdas y haya celdas sin fotones. Considerando el caso de que la escena sea una *Cornell Box*, como la que se ilustra en la Figura 2.10, los fotones van a estar concentrados en las paredes y no podrá haber fotones en aquellas zonas

que no tengan geometría. Esto causa una distribución desigual de los fotones. Este efecto se acentúa si hay efectos como cáusticas que concentren los fotones en un solo lugar. Esto hace a las grillas uniformes un mal candidato si se hace una implementación de photon mapping con CPU exclusivamente.

Grillas uniformes vs árboles kD

OppositeRenderer aplica photon mapping totalmente en la GPU. En ese caso utilizar árboles kD para almacenar el mapa de fotones no es beneficioso porque ni el armado del árbol, ni el balanceo puede paralelizarse si se usan múltiples hilos (*threads*) para hacer el photon tracing. Esto se debe a la naturaleza de ambos procedimientos que implican lecturas y escrituras concurrentes en una misma estructura. Si se usa la GPU en el photon tracing y se desea armar un árbol kD se deben guardar los fotones en una estructura sin organizar y luego armar el árbol balanceado usando la CPU. El árbol se almacena en la memoria RAM del sistema lo cual implica que los fotones deben trasladarse de la RAM del GPU a la memoria principal del sistema. Esto agrega una latencia extra. Además debe usarse un solo hilo de la CPU para armar el árbol. La idea de utilizar la GPU para el photon mapping no es nueva y en Purcell, Donner, Cammarano, Jensen, y Hanrahan (2003) se explica por qué es beneficioso utilizar grillas uniformes e introduce una forma de generarlas usando *shaders* en la GPU. Puede consultarse Pedersen (2013) donde se hace una comparación similar usando diferentes escenas a las que se probaron en este trabajo.

Método para armar grillas uniformes

A continuación se explicará el algoritmo que se implementó en el paquete usado en este trabajo, que a su vez es una variante del algoritmo propuesto por Fleisz (2009).

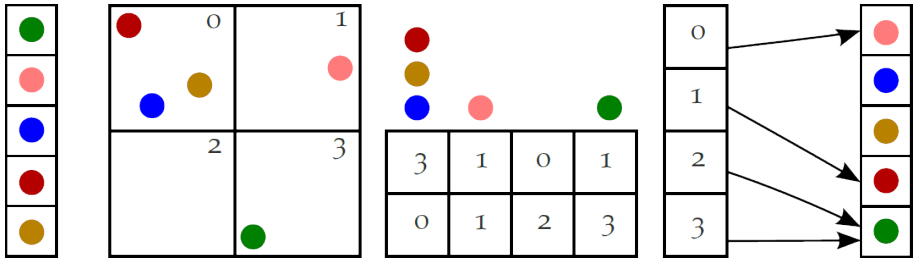


Figura A.1: Una grilla ordenada usando 5 fotones en una escena 2D. *a)* Fotones generados, *b)* Grilla de la escena, *c)* Tabla de desplazamientos, *d)* Histograma, *e)* Fotones ordenados. (Pedersen, 2013)

El método consta de los siguientes pasos

1. Se calcula el *AABB* (Axis Aligned Bounding Box) de la escena. Un AABB para un conjunto de puntos es prisma con las aristas paralelas a los ejes que contenga a un conjunto de objetos. La idea es encontrar las medidas del AABB mínimo que encapsule a todos los fotones de la escena. Este AABB va a tener como vértices a $(x_{min}, y_{min}, z_{min})$ y $(x_{max}, y_{max}, z_{max})$, donde

$$x_{min} = \min_p p_x \qquad x_{max} = \max_p p_x$$
$$y_{min} = \min_p p_y \qquad y_{max} = \max_p p_y$$
$$z_{min} = \min_p p_z \qquad z_{max} = \max_p p_z$$

y $p = (p_x, p_y, p_z)$ pertenece al conjunto P de fotones generados en la etapa de photon tracing.

2. Se organizan los fotones según el índice de la grilla tratando las celdas de la grilla como un arreglo tridimensional. Siendo C el tamaño de la grilla cuadrada, el índice de un elemento $p = (p_x, p_y, p_z) \in P$ es

$$\begin{aligned} I(p) = & \left\lfloor (C-1)\frac{p_x - x_{min}}{x_{max} - x_{min}} \right\rfloor + \\ & C \left\lfloor (C-1)\frac{p_y - y_{min}}{y_{max} - y_{min}} \right\rfloor + \\ & C^2 \left\lfloor (C-1)\frac{p_z - z_{min}}{z_{max} - z_{min}} \right\rfloor \end{aligned}$$

   Las fracciones son reales en el intervalo $[0, 1]$, por lo que se multiplica por $C - 1$ para generar un índice entero. De esta manera los valores de $I(p)$ van a estar en el conjunto $\{0, \ldots, C^3 - 1\}$

3. Se ordenan los fotones por el índice $I(p)$ generándose un conjunto de tuplas $(p, I(p))$. De esta manera todos los elementos de una misma celda son contiguos.

4. Se cuentan los elementos con un mismo valor de I(p) creando un histograma que permita mapear la celda c y el número de elementos que está en la celda para formar un histograma de los elementos del mapa de fotones. Se genera un mapeo $H : \mathbb{Z} \to \mathbb{Z}$ donde $H(c)$ es la cantidad de elementos de una celda c.

5. Luego se usa el histograma H para formar un tabla de desplazamientos $D : \mathbb{Z} \to \mathbb{Z}$ tal que $D(c)$ es el índice en la matriz donde comienzan los fotones de la celda c:

$$D(i) = \begin{cases} 0 & \text{si } i = 0 \\ H(i-1) + D(i-1) & \text{si } 1 \leq i < C^3 \end{cases}$$

   Conociendo $D(c)$ y $H(c)$ pueden enumerarse todos los fotones de una celda del mapa de fotones.

Todos los pasos se pueden realizar utilizando *CUDA* y la biblioteca *Thrust* que se discutirán en mayor detalle en la Sección 2.7. Thrust permite usar la GPU para realizar operaciones de ordenamiento, búsqueda, transformación y reducción de arreglos sobre *buffers* de elementos que están en la memoria de la GPU. En general es más rápido utilizar esta biblioteca que sus implementaciones en la CPU.

A.3. Estructura del *Buffer* de Fotones

Como se explicó en la Sección A.2 se usó una grilla uniforme para implementar el mapa de fotones. De hecho como se vio en la Sección 4.9.2 no es necesario tener una estructura para el mapa de fotones. Igualmente es útil organizar el mapa de fotones de tal manera que las renderizaciones de las imágenes de las soluciones finales sean lo más rápidas posibles. El objetivo de esta sección es describir como se forma el *buffer* de fotones durante el proceso de un photon mapping.

En primer lugar el *buffer* de fotones son N celdas de tamaño Q, donde Q es la cantidad de máxima de rebotes que se toman en cuenta en la etapa de recorrida del mapa de fotones. Al realizar el proceso de raytracing para el fotón i, donde $i \in 0, 1, \cdots, N$, todos sus rebotes se guardan en la i-ésima celda. En general el j-ésimo rebote del i-ésimo fotón está en la posición $Qi+j$. En la Figura A.2 se puede ver un ejemplo de un *buffer* de fotones. El *buffer*

de fotones no es continuo. Esto se debe a que cada celda almacena un número de rebotes diferente que puede variar si el fotón fue absorbido por una superficie o si no intersecó con la escena.

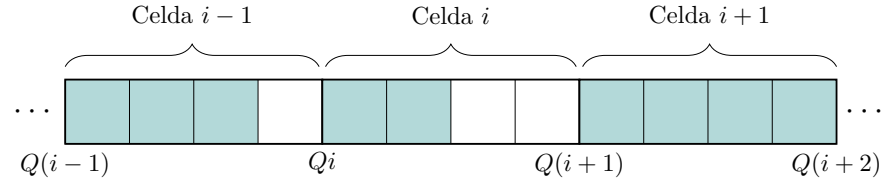


Figura A.2: *buffer* de fotones en las inmediaciones de la celda i, con $Q = 4$. Las celdas en celeste tienen fotones. En la parte inferior se notan las coordenadas dentro del arreglo.

B

Photon Mapping Progresivo

Una de las limitaciones de photon mapping es que los fotones tienen que estar en memoria hasta que la imagen esté lista para renderizar. Esto hace que sea un algoritmo muy limitado en términos de memoria. Puede no ser posible renderizar una escena lo suficientemente compleja aceptando una cantidad de ruido limitada. Esto se debe a que se necesitarían almacenar más fotones que los que la memoria del dispositivo (la GPU) puede almacenar. La variante llamada *Photon Mapping Progresivo*, introducido en Hachisuka, Ogaki, y Jensen (2008) elimina esa limitación de memoria haciendo la evaluación de la radiosidad en cada píxel de la imagen de manera iterativa e incremental. Se eligió presentar este algoritmo porque es una evolución reciente del photon mapping clásico y se utiliza en el paquete OppositeRenderer. Sin embargo no es utilizado directamente en este trabajo. El photon mapping progresivo se organiza en iteraciones: La primera es un raytracing y las subsiguientes son una cantidad arbitraria de photon tracing. Cada etapa de captura de fotones ayuda a refinar la estimación de la radiosidad calculada en la iteración anterior. Es decir que la calidad de la imagen va mejorando a medida que se ejecuta el algoritmo. Una propiedad interesante del photon mapping progresivo es que el algoritmo luego de cada etapa de photon tracing puede generar una imagen de la renderización de la escena. El algoritmo puede terminarse cuando se quiera, por ejemplo si la calidad de la escena cumpla las expectativas de tener poco ruido. En la Figura B.1 puede verse un esquema del algoritmo. Como puede verse la pasada de raytracing se hace *antes* que el photon tracing a diferencia del photon mapping clásico.

(a) Raytracing (b) Photon tracing y estimación

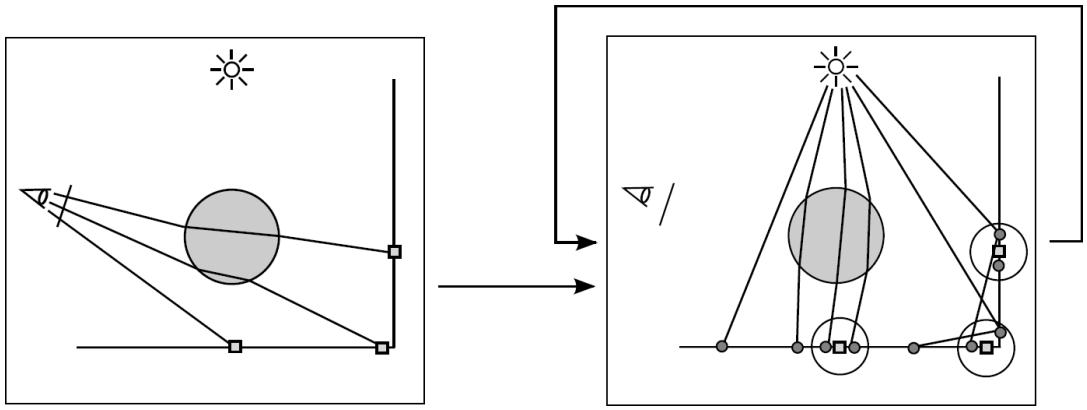


Figura B.1: Esquema de las etapas de photon mapping progresivo. Adaptado de Hachisuka y cols. (2008)

Ray Tracing

En esta etapa que se hace al principio el objetivo es conocer qué superficies son visibles desde el observador. Cada píxel de la imagen que se va a generar es influenciado por una o más *regiones* de la superficie. Cada región es simplemente un intervalo centrado en un punto p que tiene los siguientes valores:

- El punto p centro de la región.
- El radio R_p actual de la esfera de influencia. Inicialmente es un valor fijo para todas las regiones. El valor inicial es altamente dependiente de la escena.
- La cantidad de fotones N_p que impactaron en la última etapa de photon tracing. Inicialmente es 0.
- El color τ_p de la región, que es acumulado en cada etapa de estimación de radiosidad.

Pasada de photon tracing

Esta etapa es igual a la etapa de un photon mapping clásico. Se emiten fotones desde las fuentes luminosas y se almacenan aquellos fotones en un mapa. Se emite una cantidad fija en cada iteración de forma tal de acotar la cantidad de memoria necesaria para almacenarlos.

Estimación de radiosidad

Después de cada pasada de se calcula la radiosidad estimada tomando los fotones que están dentro de cada región definida inicialmente. Se acumula la potencia de esos fotones en τ_p. Se reduce el radio R_p según la cantidad de fotones N_p que cayeron en la región (si no cayó ninguno el radio no se achica). Luego de finalizados los cálculos se descartan los fotones y se vuelve a hacer otro photon tracing.

Los dos últimos pasos pueden realizarse las veces que se quiera. Los valores de τ_p determinan los colores de cada píxel y son vueltos calcular en cada iteración, teniéndose una imagen que *progresivamente* se va acercando a la solución final.

C

Análisis de la Función de Distribución de Probabilidad de la Irradiancia de una Superficie

Para calcular el valor de la irradiancia de una superficie, representado por la variable aleatoria $F_N(x)$, introducida en la Sección 3.3.1, se tienen que lanzar N fotones independientes. Se puede pensar que cada superficie s de la escena tiene una probabilidad asociada $p_s(x)$ de que un fotón caiga en ella, y además se cumple que $\sum_s p_s(x) \leq 1$.

C.1. Una Luz con una Banda de Color

Si se considera una sola banda de color (por ejemplo rojo) y una sola luz de potencia W en la banda del rojo, la suma de las potencias de los fotones en la superficie s tiene una distribución de binomial debido a que es la suma de N experimentos independientes que tienen distribución de Bernoulli con la misma probabilidad $p_s(x)$ dependiente de la configuración x elegida. Aplicando la Ecuación 3.2, teniendo en cuenta que $F_N(x)$ es una estimación del valor real de $\mathcal{I}_s(x)$, entonces:

$$F_N(x) = \frac{W}{AN} Y \tag{C.1}$$

donde Y es una variable aleatoria que indica la cantidad de fotones que impactan sobre la superficie s. Por lo discutido anteriormente, Y tiene una distribución binomial debido a que son N experimentos independientes e idénticamente distribuidos de probabilidad $p_s(x)$. Entonces $Y \sim \mathcal{B}(N, p_s(x))$[1] donde $\mathcal{B}(N, p_s(x))$ es una distribución binomial de N experimentos con probabilidad $p_s(x)$. Por lo tanto:

$$F_N(x) \sim \frac{W}{AN} \mathcal{B}(N, p_s(x))$$

C.2. Una Luz con Múltiples Bandas de Colores

Si consideramos las otras bandas de colores su distribución sería el promedio una suma de 3 distribuciones binomiales. Cada una de ellas para un color RGB:

$$\begin{aligned} F_N(x) &\sim \frac{W_r \mathcal{B}(N, p_s(x)) + W_b \mathcal{B}(N, p_s(x)) + W_g \mathcal{B}(N, p_s(x))}{AN} \\ &= \frac{W_r + W_b + W_g}{AN} \mathcal{B}(N, p_s(x)) \end{aligned}$$

donde W_r, W_g y W_b son las potencias en las bandas de rojo, verde y azul, respectivamente.

[1] $X \sim Y$ denota que X tiene una distribución de probabilidad de Y

C.3. Múltiples Luces con Múltiples Bandas de Colores

Si se hay más de una luz en la escena, el caso se torna un poco más complejo, y tiene que tenerse en cuenta como la forma en la que se generan fotones. En la implementación utilizada en este trabajo, se generan los rayos tomando en cuenta las potencias relativas de cada una de las luces respecto al resto. Entonces la probabilidad de que un fotón salga de la luz i-ésima es igual al ratio de la potencia de la luz i y la potencia de todas las luces combinadas. Siendo $W_i = W_{ri} + W_{g_i} + W_{bi}$ la suma de las potencias de cada banda de color para una luz, entonces la probabilidad P_i es:

$$P_i = \frac{W_i}{\sum_{j=1}^{m} W_j}$$

donde m es la cantidad de fuentes luminosas de la escena. Se puede asumir, simplificando el modelo, que de los N fotones generados aproximadamente NP_i saldrán de la i-ésima luz; de hecho se puede forzar a que en la etapa de photon tracing se genere esa cantidad exacta de fotones. Sin embargo, la solución implementada elige la luz al aleatoriamente, teniendo en cuenta las probabilidades P_i, por lo que se tiene un componente estocástico.

Debido a que las luces están en distintos puntos de la escena se tiene también que considerar que un fotón que sale de una luz no tiene la misma probabilidad de impactar la superficie s que otro fotón que sale de una luz diferente. Por eso se tienen valores $p_{s_i}(x)$ para cada fuente luminosa. Cada $p_{s_i}(x)$ es la probabilidad de que el fotón de la i-ésima luz llegue a la superficie s.

Considerando los valores de P_i y $p_{s_i}(x)$ para cada luz, se puede comprobar que la estimación $F_N(x)$ es una suma de i distribuciones binomiales independientes, debido a que la elección de la luz que genera el fotón es independiente a si el fotón pega en la superficie s, esto es porque se asumió que se generan exactamente NP_i rayos para cada emisor de luz, entonces:

$$F_N(x) \sim \sum_{i=1}^{m} \frac{W_i}{ANP_i} \mathcal{B}\left(NP_i, p_{s_i}(x)\right) = \sum_{i=1}^{m} \frac{W_i}{AN\frac{W_i}{W}} \mathcal{B}\left(NP_i, p_{s_i}(x)\right)$$

Si se toma $W = \sum_{j=1}^{m} W_j$ se puede deducir una fórmula más simple para la distribución:

$$F_N(x) \sim \frac{W}{AN} \sum_{i=1}^{m} \mathcal{B}\left(NP_i, p_{s_i}(x)\right)$$

La esperanza de la distribución $F_N(x)$ es:

$$\begin{aligned} E(F_N(x)) &= E\left(\frac{W}{AN} \sum_{i=1}^{m} \mathcal{B}\left(NP_i, p_{s_i}(x)\right)\right) \\ &= \frac{W}{AN} \sum_{i=1}^{m} E\left(\mathcal{B}\left(NP_i, p_{s_i}(x)\right)\right) \\ \Rightarrow E(F_N(x)) &= \frac{W}{AN} \sum_{i=1}^{m} NP_i \; p_{s_i}(x) \end{aligned}$$

Usando las propiedades básicas de la varianza:

- $Var(\mathcal{B}(n,p)) = np(1-p)$
- $Var(aX) = a^2 Var(X)$
- $Var(X+Y) = Var(X) + Var(Y) + 2\, Cov(X,Y)$, donde $Cov(X,Y)$ es la covarianza de X e Y. Si X y Y son independientes entre sí como pasa en este caso su covarianza es nula.

Se puede calcular la varianza de la distribución:

$$\begin{aligned} Var(F_N(x)) &= \left(\frac{W}{AN}\right)^2 \sum_{i=1}^{m} Var\left(\mathcal{B}\left(NP_i, p_{s_i}(x)\right)\right) \\ &= \left(\frac{W}{AN}\right)^2 \sum_{i=1}^{m} NP_i p_{s_i}(x)(1-p_{s_i}(x)) \\ &= \left(\frac{W}{AN}\right)^2 \sum_{i=1}^{m} N\frac{W_i}{W} p_{s_i}(x)(1-p_{s_i}(x)) \\ \Rightarrow Var(F_N(x)) &= \frac{W}{A^2N} \sum_{i=1}^{m} W_i p_{s_i}(x)(1-p_{s_i}(x)) \end{aligned}$$

Alternativa al caso de múltiples luces

Se puede simplificar el modelo anterior observando que la probabilidad de que un fotón que caiga en la superficie s, para una configuración x es fija y depende solo de x. Se llamará a esta probabilidad $p_s(x)$. Entonces usando el mismo razonamiento que para el caso de una luz, usando la Ecuación C.1. El valor de $F_N(x)$ depende de los fotones que caen en la superficie que tienen una distribución de probabilidad $\mathcal{B}(N, p_s(x))$ entonces:

$$F_N(x) \sim \frac{W}{AN}\mathcal{B}(N, p_s(x))$$

Que es independientemente del número de luces de la escena. Haciendo los mismos cálculos que en el caso de múltiples luces la esperanza de $F_N(x)$ es

$$E(F_N(x)) = E\left(\frac{W}{AN}\mathcal{B}(N, p_s(x))\right) = \frac{W}{AN}Np_s(x) = \frac{W}{A}p_s(x) \tag{C.2}$$

Y su varianza es:

$$Var(F_N(x)) = Var\left(\frac{W}{AN}\mathcal{B}(N, p_s(x))\right) = \left(\frac{W}{AN}\right)^2 Np_s(x)(1-p_s(x))$$

$$\Rightarrow Var(F_N(x)) = \left(\frac{W}{A}\right)^2 \frac{p_s(x)(1-p_s(x))}{N} \tag{C.3}$$

Estos valores son más simples de calcular dado que no hay que disgregar por las diferentes luces, y son independientes de la forma en la que se modela la generación de fotones en la etapa de photon tracing. De hecho esta fue la estimación que se utilizó en este trabajo.

D

Archivo de Definición de ILPSolver

Se incluye un archivo de definición de un problema de optimización de ejemplo para la ejecución del programa ILPSolver. En ese archivo se muestran los diferentes tipos de variables de optimización soportados.

```
<?xml version="1.0" encoding="utf-8"?>
<input>
  <!-- Ruta al archivo de la escena (formato Collada) -->
  <scene path="cube.dae"/>

  <!-- Carpeta donde se deja el log de ejecución, y las soluciones encontradas -->
  <output path="output"/>

  <!-- Variables de optimización -->
  <conditions>
    <!-- Abertura en una pared -->
    <holeInSurface id="Hole" surface="Wall"
        vertexAIndex="1"
        vertexBIndex="0"
        vertexCIndex="2"
        vertexDIndex="3"/>
    <!-- Luz direccional -->
    <directionalLight id="Sun"/>
    <!-- Luz puntual o emisor difuso en una superficie -->
    <lightInSurface id="EmitiveLight" surface="Ceiling"/>
    <!-- Objeto que puede moverse en una superficie -->
    <objectInSurface id="Chair" surface="Floor"/>
    <!-- Color que puede ser modificado -->
    <color id="Walls" saturation="0.23" hue="39"/>
  </conditions>

  <!-- Objetivos de optimización  -->
  <objectives maxIterations=1000>
    <!-- Sobre qué superficie se va optimizar la irradiancia -->
    <maximizeRadiance surface="Suzzane"/>
  </objectives>
</input>
```

E

Declaración de Struct Photon

A continuación se incluye la definición del archivo `Photon.h` que define la estructura de un fotón que se almacena en el mapa de fotones. El tamaño de la estructura en la implementación es de 40 *bytes*, este número puede variar si se usa otro compilador diferente al que se recomienda. El tipo `optix::float3` es una terna de números `float`.

```
struct Photon
{
    #ifdef __CUDACC__
        __device__ __inline Photon(
          const optix::float3 & power,
          const optix::float3 & position,
          const optix::float3 & rayDirection,
          const optix::uint & objectId):
            power(power),
            position(position),
            rayDirection(rayDirection),
            objectId(objectId)
        {

        }

        __device__ __inline Photon(void)
        {

        }
    #endif

    // La potencia del fotón en las bandas RGB.
    // Se usa tanto para generar imágenes como
    // en ILPSolver
    optix::float3 power;

    // la posición en la escena donde impactó.
    // Se usa para generar imágenes en la renderización
    optix::float3 position;

    // la dirección de impacto donde proviene el impacto.
    // Se usa para generar imágenes en la renderización
```

```
    optix::float3 rayDirection;

    // El índice de la superficie en que impactó el fotón.
    // Se usa solo en ILPSolver.
    optix::uint   objectId;

#if ACCELERATION_STRUCTURE == ACCELERATION_STRUCTURE_KD_TREE_CPU
    // Solo se usa cuando se tiene un árbol KD.
    // Indica el eje de partición
    optix::uint   axis;
#endif
};
```