UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Tesis de grado

Iluminación global con superficies especulares

Bruno Sena

Tesis de grado presentada en la Facultad de Ingeniería de la Universidad de la República, como parte de los requisitos necesarios para la obtención del título de grado en Ingeniería en Computación.

Directores:

José Pedro Aguerre

Eduardo Fernández

RESUMEN

Los algoritmos de iluminación global simulan el comportamiento de la luz en la naturaleza. La síntesis de imágenes fotorealistas generadas por computadora o la evaluación del diseño lumínico para arquitectura son algunos de los objetivos abarcados por este tipo de algoritmo.

En este contexto, este proyecto se enmarca en el estudio, análisis y adaptación de técnicas que proponen extensiones a el método de radiosidad. Este método se basa en el estudio de la transferencia de energía lumínica entre superficies que componen una escena. Para simplificar el cálculo del intercambio de radiación entre elementos de la escena es subdividida en una cantidad de superficies planas discreta, que se denominan parches.

Este método de radiosidad clásico considera únicamente superficies lambertianas, es decir, reflectores difusos perfectos. Esta limitación supone que el método no pueda aplicarse en una gran variedad de escenas donde la incidencia de la reflexión especular tiene una gran incidencia en la iluminación de la escena.[1].

La extensión planteada propone la construcción de un algoritmo capaz de considerar superficies especulares. Por otro lado, se proponen técnicas que permitan proveer nuevos acercamientos al cálculo de los factores de forma utilizando técnicas de paralelismo para el hardware moderno.

Palabras claves:
Iluminación Global, Radiosidad, Reflexión Especular.

[1]Disponible en GitHub: https://github.com/brunosegiu/radiosum

Tabla de contenidos

Capítulo 1

Introducción

1.1. Motivación y problema

Naturalmente, el fenómeno de la iluminación ocurre cuando una fuente luminosa emite un flujo de fotones, normalmente conocido como rayo o haz de luz, y recorre un camino hasta intersectar una superficie que lo bloquee o desvíe. Según la óptica, los fenómenos que pueden manifestarse debido a esta interacción son la *absorción, reflexión, refracción*; pues una superficie puede conservar parte de la energía transmitida afectando el color percibido, reflejarla en distintas direcciones, modificar la dirección del haz al poseer propiedades de transparencia.

Para simular dicho fenómeno, se han desarrollado y formalizado un conjunto de algoritmos y herramientas que resuelven el problema con distintos niveles de complejidad y realismo físico. Para caracterizar los problemas descriptos, normalmente se utiliza un estándar conocido como expresiones de caminos de luz [1] (o LPEs, por sus siglas en inglés). En estas expresiones regulares se establecen distintos eventos. Que son representados por un conjunto de letras, leídas de izquierda a derecha, y corresponden a un camino de la luz particular. A efectos de este proyecto, interesan los objetos **C** (cámara) y **L** (emisión de luz), así como los eventos **S** (reflexión especular), **D** (reflexión difusa).

Estas formas fundamentales son las que rigen el comportamiento de la luz cuando es reflejada en la naturaleza. En la primera, la luz es reflejada en direcciones aleatorias, que idealmente son distribuidas siguiendo la distribución del coseno. Mientras que en el segundo caso, el ángulo de reflexión es igual al de incidencia como se aprecia en la Figura 1.1.

(a) Difusa **(b)** Especular

Figura 1.1: Posibles formas de reflexión de la luz con superficies.

Los algoritmos de traza de rayos o radiosidad resuelven el problema de la iluminación global con distintos acercamientos. Originalmente, el algoritmo de traza de rayos contempla los caminos de luz de la forma **L(S|D)*E**, es decir, cualquier nivel de reflexiones especulares o difusas; la solución se basa en simular haces de luz (rayos) como semi-rectas, calculando los puntos de intersección con los objetos de la escena recursivamente. Por otro lado, el algoritmo de radiosidad involucra la subdivisión de una escena virtual en superficies finitas conocidas como *parches* a los que se les asignará un valor de radiosidad (energía lumínica) dependiente de la ubicación de las fuentes luminosas y las oclusiones causadas por la disposición de la geometría de la escena. Esto quiere decir, que en su concepción, el algoritmo sólo considera caminos de la forma **LD*E**. No obstante, es deseable que los caminos especulares sean contemplados en el cálculo de la radiosidad.

Estas interacciones observadas son de gran interés para la computación gráfica, dado que su estudio tiene gran relevancia en la arquitectura y la industria del entretenimiento. En particular, la generación de modelos capaces de simular el transporte de la luz en espacios tridimensionales es uno de los mayores motivadores de los avances en computación gráfica. A lo largo de los últimos 50 años se han propuesto distintos modelos ([2], [3]) que aproximan el comportamiento real de la luz en distintos entornos con variados niveles de foto-realismo y desempeño computacional.

Si bien el trazado de rayos tiene el potencial de resolver todos los fenómenos físicos explicados anteriormente, su desempeño computacional empeora con la presencia de superficies difusas. El algoritmo de radiosidad tiene la capacidad de resolver los caminos de la forma **LD*E** (considerando únicamente el fenómeno de la reflexión difusa) de forma eficiente a través del uso de facto-

res de vista para el modelado del fenómeno de la iluminación así como del transporte de energía térmica entre superficies.

El avance del *hardware* obliga a revaluar los algoritmos de iluminación global constantemente, con el objetivo de proveer resultados fotorealistas en tiempos de ejecución cada vez menores. Si bien se han desarrollado algunas técnicas como el *trazado de rayos bi-direccional* y otras basadas en *trazado de rayos de Monte Carlo* que apuntan a solucionar el problema del cálculo de la reflexión difusa utilizando el algoritmo de traza de rayos, también se han propuesto variantes que extienden los métodos de radiosidad para considerar los efectos introducidos por las superficies especulares.

1.2. Objetivos

Este proyecto tiene el objetivo de analizar las técnicas de extensión del método de radiosidad a caminos de la forma **L(S|D)*E**, generando adaptaciones de las extensiones propuestas por trabajos previos y literatura asociada. A su vez, se generará la implementación correspondiente a los distintos algoritmos formulados con la finalidad de comparar cualitativamente el rendimiento computacional observado en hardware moderno así como el error introducido por las aproximaciones que se consideran al discretizar el problema concebido.

En este sentido, se exploran tres implementaciones diferentes para el cálculo de la radiosidad entre las superficies que componen la escena virtual considerada. Se propone aprovechar la implementación en hardware de un conjunto de funcionalidades que facilitan la proyección tridimensional así como el uso de eficiente de los recursos de cómputo a través de bibliotecas que permiten manejar el paralelismo.

1.3. Estructura del documento

El resto del documento se estructura de la siguiente manera. El Capítulo 2 introduce el estado del arte en técnicas de iluminación global, con especial énfasis en la técnica de radiosidad y sus extensiones. Además, se exploran diversos acercamientos alternativos al problema a resolver. El Capítulo 3 refiere al diseño de la solución de los algoritmos implementados. El Capítulo 4 describe la implementación realizada, detallando distintas decisiones tomadas para eludir

un conjunto de obstáculos técnicos observados. En el Capítulo 5, se encuentra una síntesis de los casos de prueba considerados así como los un análisis de los resultados obtenidos junto a un conjunto de ventajas y desventajas que se han observado. Finalmente, se desarrollan las conclusiones y posible líneas de trabajo futuro detectadas a lo largo del desarrollo del proyecto.

Capítulo 2

Estado del arte

Este capítulo introduce un resumen de las áreas más importantes relacionadas al trabajo realizado en este proyecto, incluyendo los modelos de iluminación por computadora, el método de radiosidad y sus posibles implementaciones y extensiones.

2.1. Modelos de iluminación

El proceso de dibujado de gráficos tridimensionales por computadora comprende la generación automática de imágenes con cierto nivel de realismo a partir de modelos que componen una *escena* o *mundo* tridimensional, junto a un conjunto de cualidades físicas que rigen las formas en la que la luz interactúa con los objetos.

Este problema puede ser reducido al problema de cálculo del valor de intensidad lumínica observada en un punto x y proveniente directamente de un conjunto de puntos, representado por x'. En 1986, Kajiya presentó uno de los modelos más aceptado por la comunidad por su generalidad, la denominada «ecuación del *rendering*»:

$$I(x, x') = g(x, x')\left[\epsilon(x, x') + \int_S \rho(x, x', x'')I(x', x'')\delta x''\right] \tag{2.1}$$

donde:

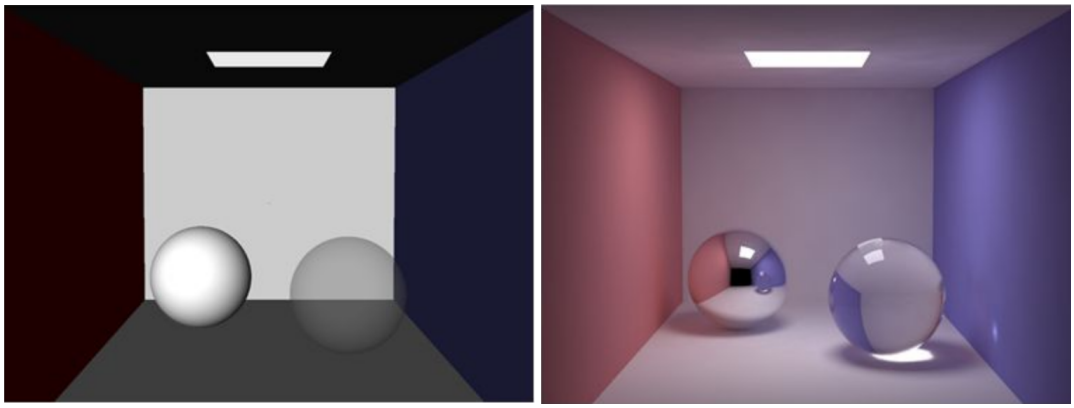
- $I(x, x')$ describe la intensidad lumínica que llega al punto x proveniente de x'.
- $g(x, x')$ es un término geométrico, toma el valor de 0 si existe oclusión

entre x' y x, y en otro caso su valor es $\frac{1}{r^2}$ donde r es la distancia entre ambos puntos.

- $\epsilon(x, x')$ expresa la intensidad lumínica emitida por la superficie en el punto x' en dirección a x.
- $\int_S \rho(x, x', x'')I(x', x'')\delta x''$ está compuesta por dos términos:
  - $\rho(x, x', x'')$ es el término de reflectividad bi-direccional, es decir la proporción de luz que va desde x'' a x pasando por x'.
  - $I(x', x'')$ describe la intensidad lumínica observada en el punto x' proveniente de x''.

  por lo que este término refiere a la intensidad percibida desde x considerando todas las reflexiones de luz posibles para el dominio S.

Existen distintos métodos de resolución de la ecuación del rendering, donde la mayoría implican cálculos aproximados dado el gran costo de cómputo requerido para hallar una solución exacta. Estos métodos balancean el costo computacional de los algoritmos utilizados y el error del valor obtenido. Existen dos categorías principales para el método: *local* y *global*. Un ejemplo de ambos modelos puede ser observado en la Figura 2.1.



(a) Local **(b)** Global

Figura 2.1: Dibujado utilizando distintos modelos de iluminación.

2.1.1. Iluminación Local

Los modelos de iluminación local tienen en cuenta las propiedades físicas de los materiales y las superficies de forma individual. Es decir, al dibujar cada

objeto no se toman en cuenta las posibles interacciones de los haces de luz con los objetos restantes en la escena. Esto implica que no se proyectan sombras, y tampoco se modelan correctamente las cáusticas producidas por la acumulación de la luz ni el sangrado, entre otros fenómenos de la naturaleza. Estos métodos son más sencillos de implementar y son frecuentemente utilizados en problemas cuya resolución debe ser realizada en tiempo real o por decisiones artísticas.

En referencia a la ecuación del rendering, el término geométrico nunca toma el valor 0, es decir, no se toma en cuenta las colisiones de la luz con otros objetos. El término $\epsilon(x, x')$ toma un valor constante únicamente dependiente de x y $\int_S \rho(x, x', x'')I(x', x'')\delta x''$ toma el valor constante 0.

2.1.2. Iluminación Global

El modelo de iluminación global refiere a un conjunto de técnicas que simulan parcial o completamente las interacciones de la luz con todos los objetos que se encuentran en la escena. Es decir, en contraposición a la iluminación local, se consideran los fenómenos de reflexión y refracción de la luz.

Dependiendo de las característica de los modelos y algoritmos empleados, pueden obtenerse resultados fotorealistas para diferentes escenarios.

El algoritmo de *path-tracing* emula completamente cada haz de luz desde su incepción en una fuente luminosa siguiendo el camino de interacciones del rayo con las distintas superficies de la escena. En este caso el grado de granularidad (que depende directamente de la cantidad de muestras utilizadas) influye en la precisión y calidad en la imagen final. En esencia, el cálculo de la iluminación se basa en un método de Monte Carlo donde para cada punto de la escena se calcula cuánta luz se destina al observador (o cámara) en función de la función de distribución de reflectancia de la superficie como lo sugieren los autores Lafortune y Williams [4].

Por otro lado, el algoritmo de *mapeado de fotones* propuesto por Jensen [5] supone simular los efectos producidos por las colisiones de las partículas que componen la luz (fotones) con los objetos. Es decir, en la dualidad del modelo de luz onda/partícula supone la segunda interpretación. Las partículas son disparadas desde las fuentes luminosas bajo cierta función de distribución. El algoritmo se compone de dos etapas, en la primera se lanzan los fotones y se construye un mapa de que los relaciona con distintas posiciones de la escena.

En la segunda se genera la imagen final a partir de las impresiones que las partículas dejan al interactuar con las superficies de la escena.

Existen además distintas variaciones e híbridos de estos métodos ya que los mismos son demasiado costosos como para dibujar imágenes en tiempo real, en sus versiones originales.

2.2. Radiosidad

El método de radiosidad es una técnica de iluminación global que emula el transporte de la luz entre superficies difusas. El mismo nombre se utiliza también para describir la magnitud física definida como radiosidad, que indica el flujo de energía radiada por unidad de área ($\frac{W}{m^2}$).

Originalmente, este modelo de iluminación global fue propuesto por Goral et al.[[6]], y se basa en modelos matemáticos similares a los que resuelven el problema de la transferencia de calor en sistemas cerrados discretos como diferencias finitas o elementos finitos.

2.2.1. Radiosidad en superficies lambertianas

La solución propuesta por Goral et al. implica que todas las superficies son idealmente difusas, también conocidas como lambertianas. Estas superficies se comportan como reflectores difusos ideales, lo que significa que reflejan la energía incidente de forma isotrópica siguiendo la regla del coseno como se observa en la Figura 2.2.

Adicionalmente, se considerará que la energía lumínica irradiada en todas direcciones por cada diferencial de área δ_A, puede ser definida como:

$$I = \frac{\delta P}{\cos\phi \delta\omega} \tag{2.2}$$

donde:

- ω es la dirección de vista.
- I es la intensidad de la radiación para un punto de vista particular.
- δP es lae energía de la radiación que emana la superficie en la dirección ϕ con ángulo sólido $\delta\omega$.

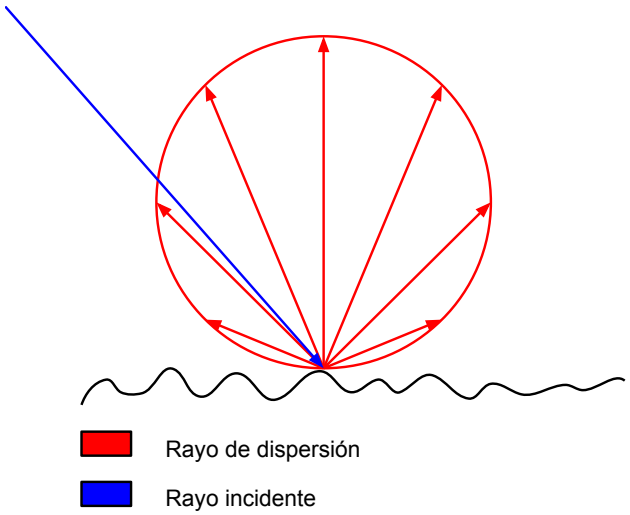


Figura 2.2: Reflector lambertiano

En superficies perfectamente lambertianas, la energía reflejada puede ser expresada como: $\frac{\delta P}{\delta \omega} = k \cos \phi$. Donde k es una constante. Sustituyendo en (2.2) se obtiene: $\frac{\delta P}{\delta \omega} = \frac{k \cos \phi}{\cos \phi} = k$, esto implica que la energía percibida de un punto x es constante, independientemente del punto de vista.

Es por esto que la energía total que deja una superficie (P) puede ser calculada integrando la energía que deja la superficie en cada dirección posible, esto es, se integra la energía saliente en un hemisferio centrado en el punto estudiado:

$$P = \int_{2\pi} \delta P = \int_{2\pi} I \cos \phi d\omega = I \int_{2\pi} \cos \phi d\omega = I\pi \tag{2.3}$$

Por tanto, dada una superficie S_i, es posible calcular la energía lumínica que deja la superficie utilizando (2.3). Para ello, se discretizan las superficies en parches difusos, lo que transforma la Eq. (2.1) en:

$$B_i = E_i + \rho_i^{(d)} \sum_{j=1}^{N} B_j F_{ij} \tag{2.4}$$

donde:

- B_i es la intensidad lumínica (radiosidad) que deja la superficie i.
- E_i es la intensidad lumínica directamente emitida por i.
- $\rho_i^{(d)}$ es la reflectividad difusa del material para la superficie i.
- F_{ij} se denomina *factor de forma*, un término que representa la fracción

de energía lumínica va del parche i al parche j.

Cabe destacar que la naturaleza recursiva de la ecuación anterior (para calcular B_i se debe conocer B_j) implica que se toman en cuenta todas las reflexiones difusas que existan en la escena. Como se puede observar, resolver el sistema de N ecuaciones lineales bastaría para conocer la energía emitida por cada parche.

Los factores de emisión y reflexión, para cada parche i: E_i y $\rho_{\mathbf{i}}^{(\mathbf{d})}$ respectivamente, dependen de los materiales que compongan la escena y son parámetros dados. Sólo resta computar la matriz de factores de forma $\mathbf{F}$ para poder calcular el vector de radiosidades B.

Para determinar una entrada de la matriz F_{ij} involucrando a las superficies i y j de área $A(i)$, $A(j)$, considerando los diferenciales infinitesimales de área δA_i, dA_j, representados en la Figura 2.3, el ángulo sólido visto por dA_i es $d\omega = \frac{\cos\phi_j dA_j}{r^2}$. Sustituyendo en (2.3) se obtiene:

$$dP_i dA_i = I_i \cos\phi_i d\omega dA_i = \frac{P_i \cos\phi_i \cos\phi_j dA_i dA_j}{\pi r^2} \tag{2.5}$$

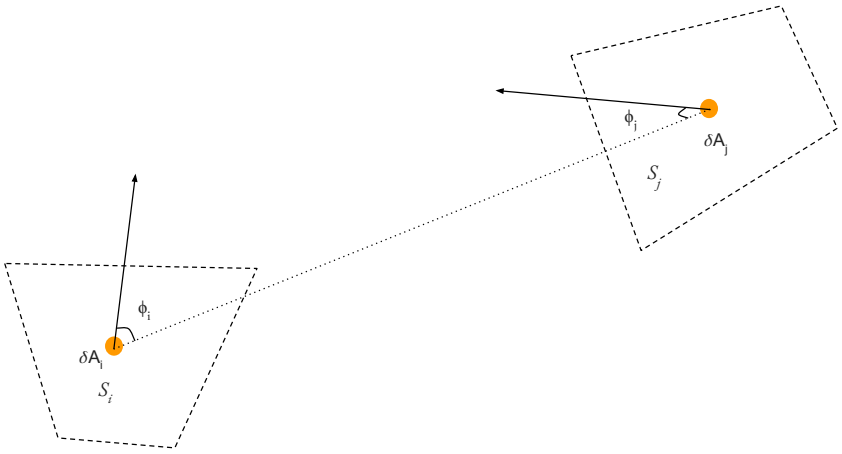


Figura 2.3: El factor de forma entre dos superficies

Considerando que $P_i A_i$ es la energía que deja i, y que el factor de forma F_{ij} representa la fracción de dicha energía que llega a j podemos observar que:

$$F_{\delta A_i - \delta A_j} = \frac{\cos\phi_i \cos\phi_j \delta A_j}{\pi r^2} = \frac{\cos\phi_i \cos\phi_j \delta A_i}{\pi r^2} \tag{2.6}$$

Integrando, para obtener el factor de forma para el área total:

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos \phi_i \cos \phi_j \delta A_i \delta A_j}{\pi r^2} \tag{2.7}$$

De (2.7) se obtienen las siguientes propiedades:

1. $A_i F_{ij} = A_j Fij$, lo que supone una relación simétrica entre los factores de forma.
2. $\sum_{j=1}^{N} F_{ij} < 1$ Es decir, la suma de una de las filas de la matriz de factores de forma no podrá tener un valor superior a la unidad.
3. $F_{ii} = 0$ Esto se debe a que los parches considerados son planos y por tanto no reflejan su propia luz.
4. F_{ij} toma el valor correspondiente a la proyección de j en un hemisferio unitario centrado en i, proyectándola a su vez en un disco unitario.

2.2.2. Métodos de cálculo de la matriz de Factores de Forma

El cálculo analítico de los factores de forma a través de la Eq. (2.7) es inviable en la práctica pues supone la necesidad de conocer la visibilidad entre cada par de parches que componen la escena. Por tanto, es necesario establecer otros métodos que provean aproximaciones lo suficientemente correctas.

Geométricamente, puede establecerse una analogía para la computación de factores de forma conocida como «analogía de Nusselt» (ver Figura 2.4). Se expresará el factor de forma como la proporción de área proyectada de S_j en un hemisferio ubicado en el baricentro de S_i y luego en un disco centrado en S_i.

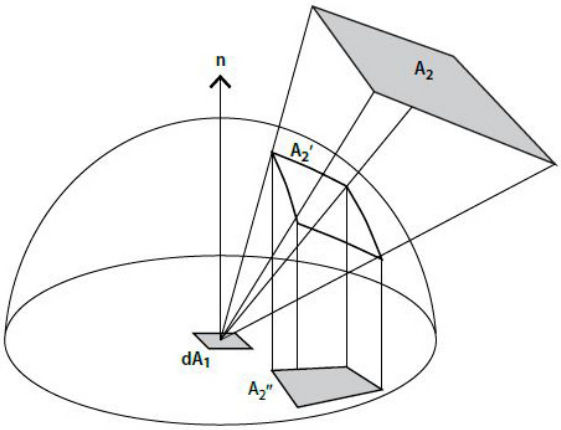


Figura 2.4: La analogía de Nusslet

El cálculo de la matriz de factores de forma **F** supone la proyección de los parches. De aquí en más se asumirá que estos parches son polígonos no curvos, lo que permite utilizar las técnicas de dibujado de objetos tridimensionales tradicionales como en la rasterización.

Rasterización

El «*rendering pipeline*» es un proceso de dibujado estandarizado que consiste en un conjunto de etapas cuyo cometido es la generación de un *frame buffer*. Los fabricantes de los dispositivos aceleradores gráficos y/o sistemas operativos proveen de interfaces de programación (OpenGL, Vulkan, DirectX) que se basan en este modelo para abstraer el uso del hardware.

Si bien el «*rendering pipeline*» es modificable, cada una de sus etapas están definidas. El programador es capaz de modificar pequeñas funciones (también llamadas *kernels* o *shaders*) que son ejecutadas en la GPU en las etapas correspondientes. El cometido de estas funciones es procesar los parámetros de entrada para generar parámetros que recibirá la siguiente etapa, que los recibirá y transformará como corresponda.

A continuación, se describe el proceso para OpenGL 4.5 visualizado en la Figura 2.5, aunque muchas de estas etapas son trasladables a otras tecnologías existentes.

1. Procesamiento de primitivas geométricas:

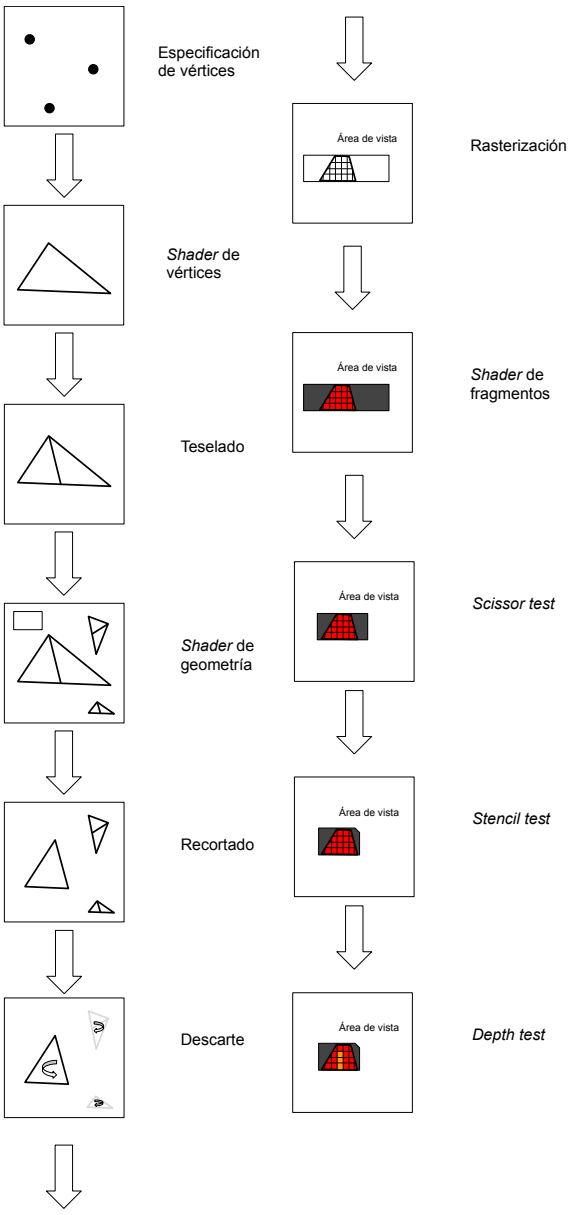


Figura 2.5: El *rendering pipeline* de OpenGL

a) Especificación de vértices: Inicialmente, las aplicaciones indican un conjunto de vértices a dibujar, definiendo cierto conjunto de primitivas geométricas como triángulos, cuadriláteros, puntos, líneas u otros.

b) *Shader de vértices*: Esta etapa transforma los vértices de entrada suministrados por la aplicación. Generalmente se computan las transformaciones lineales necesarias para cambiar la base de las coordenadas de los vértices de un sistema local al sistema global que defina la aplicación. Las coordenadas retornadas deberán corresponderse con coordenadas del espacio de recorte. Es decir, coordenadas correspondientes al volumen de vista.

c) Teselado: En esta etapa se procesan los vértices a nivel de primitiva geométrica, con el objetivo de subdividirlas para mejorar la resolución obtenida.

d) *Shader de geometría*: En esta etapa también se procesan los vértices a nivel de primitiva geométrica con el objetivo de mutarlas y replicarlas.

e) Recortado: Esta etapa es *fija*, es decir, no es programable. Todas las primitivas calculadas anteriormente que residan fuera del volumen de vista serán descartadas en las etapas futuras. Además, se transforma las primitivas a coordenadas de espacio de ventana.

f) Descarte: El proceso de descarte (en inglés *culling*), es también fijo. Consiste en la eliminación de primitivas que no cumplan ciertas condiciones, como por ejemplo el descarte de caras cuya normal tiene dirección opuesta a la del observador.

2. Procesamiento de fragmentos (rasterización):

a) Rasterización: El proceso de rasterización discretiza las primitivas en espacio de pantalla en un conjunto de fragmentos.

b) *Shader de fragmentos*: El procesamiento de cada fragmento se realiza a través del *shader de fragmentos* que calcula uno o más colores, un valor de profundidad, y valores de plantilla (del inglés *stencil*).

c) *Scissor test*: Todos los fragmentos fuera de un área rectangular definida por la aplicación son descartados.

d) *Stencil test*: Los fragmentos que no pasan la función de planilla definida por la aplicación no son dibujados, por ejemplo, simular el

scissor test que requieran primitivas más complejas.

e) *Depth test*: En esta etapa se ejecuta el algoritmo del Z-Buffer, donde sólo se escribirá el resultado en el *frame buffer* de aquellos fragmentos que tengan la menor profundidad. Es decir, los que se encuentren más cerca del observador.

Esta técnica de dibujado es extremadamente rápida, además, la mayoría de dispositivos contienen hardware especializado capaz de acelerar estos cálculos, comúnmente conocidos como Unidades de Procesamiento Gráfico (o GPU por sus siglas en inglés).

Con el objetivo de calcular los factores de forma utilizando este hardware, Cohen y Greenberg [3] idearon el método del hemi-cubo.

El método del hemi-cubo

El hardware optimizado para realizar operaciones de rasterización tiene la capacidad de proyectar escenas tridimensionales sobre ventanas de vista planas a gran velocidad.

El método original de cálculo de factores de forma propone la proyección de la escena sobre un hemisferio centrado en S_i. Sin embargo, los modelos de proyección utilizados no lo facilitan pues las transformaciones se realizan sobre superficies planas. Por esto, Cohen y Greenberg [3] proponen proyectar la escena a un hemi-cubo centrado en S_i, esto supone el dibujado de cinco superficies planas, y por tanto puede ser realizada utilizando la rasterización como se aprecia en la Figura 2.6.

Para utilizar el hardware eficientemente se considera que se calcula una fila completa de $\mathbf{F}$, esto implica que dado el parche S_i, se calcula simultáneamente los factores de forma desde S_i al resto de las superficies restantes.

Este método aprovecha el buffer de profundidad (Z-buffer), para la correcta determinación de visibilidad entre parches tomando en cuenta los fragmentos proyectados para los elementos que se encuentren más cercanos al parche S_i.

Este algoritmo, propone rasterizar la escena tridimensional en cinco texturas correspondientes a las caras de un hemi-cubo. Para cada fragmento renderizado se sumará un valor diferencial del factor de forma (o delta factor de forma), que dependerá de la posición del píxel en el hemi-cubo en relación a el hemisferio que este aproxima. Esta suma genera una fila de la matriz $\mathbf{F}$, específicamente la fila $\mathbf{F}_i$, como se puede observar en la Eq. (2.8).

Por tanto, podremos definir:

$$\mathbf{F}_{ij} = \sum_{q_j=1}^{R_j} \delta F_{q_j} \tag{2.8}$$

donde:

- R_j es la cantidad de píxeles correspondientes a la superficie S_j que cubren el hemi-cubo.
- δF_{q_j} el delta factor de forma asociado al píxel del hemi-cubo q_j.

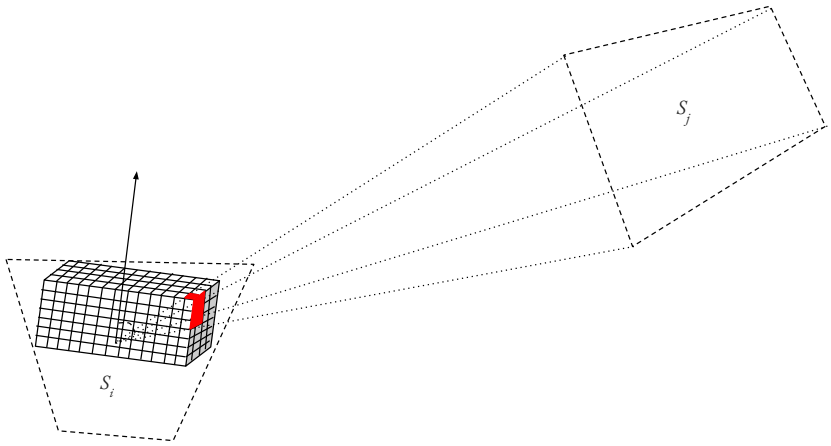


Figura 2.6: Representación gráfica del método del hemi-cubo.

Los delta factores de forma deben corregir la deformación introducida con el cambio de proyección desde un hemisferio a un hemi-cubo. Para ello, para cada píxel que compone el hemi-cubo es necesario calcular la proporción de área que este término ocupa en el hemisferio unitario.

Para la cara superior, los diferenciales se calculan como (ver referencias en la Figura 2.7):

$$\delta F_q = \frac{\cos\phi_i \cos\phi_j}{\pi r^2}\delta A = \frac{\delta A}{\pi(x^2+y^2+1)} \tag{2.9}$$

Para las caras laterales, la fórmula dada es:

$$\delta F_q = \frac{\cos\phi_i \cos\phi_j}{\pi r^2}\delta A = \frac{z\delta A}{\pi(x^2+z^2+1)} \tag{2.10}$$

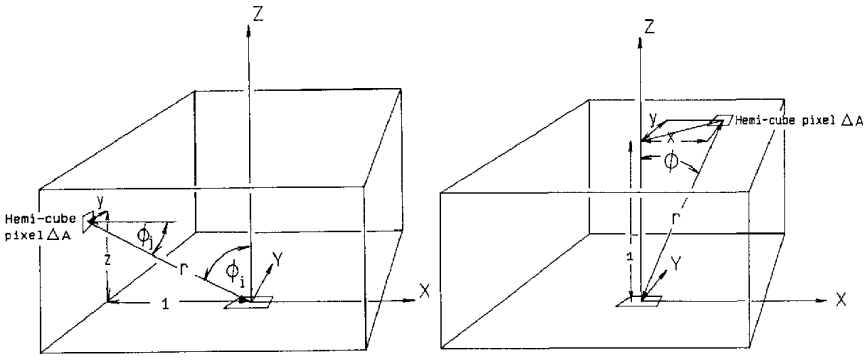


Figura 2.7: Representación gráfica de los ejes considerados para el factor de corrección de los factores de forma. [3]

Trazado de rayos

Otra de las técnicas de simulación de iluminación existente es el ray tracing que consiste en el cálculo de la intersección de una semi-recta (a la que denominaremos rayo) con la geometría de la escena. Cada uno de estos rayos simulará un haz de luz.

Para cada uno de los rayos emitidos, se determinará el punto de intersección más cercano. Dada la primitiva geométrica interceptada, es posible integrar el resultado intermedio al resultado final, dependiendo del modelo de iluminación utilizado.

El método del hemisferio

El trazado de rayos es una técnica efectiva según Kajiya [2], para resolver la ecuación del rendering, utilizando la técnica de *trazado de camino* donde el haz de luz absorbe las propiedades de los materiales con los que interactúa. En este algoritmo, la integral se resuelve con un método de Monte Carlo, donde cada rayo representa una muestra estadísticamente independiente, para resolver la doble integral presentada en la Eq. (2.7).

Es posible pensar el problema original colocando un hemisferio unitario en el centro de S_i orientada en la dirección de la normal de la superficie.

El algoritmo propuesto por Malley [7] consiste realizar un muestreo de la cantidad de rayos que parten desde el centro de S_i e intersecan S_j. Las direcciones de los rayos serán determinadas a partir de la *distribución del coseno* cuya función de densidad es $f(x) = \frac{1}{2}[1 + \cos((x-1)\pi)]$ donde x es una variable aleatoria uniforme en el rango $[0, 1]$.

$$\mathbf{F}_{ij} = \frac{1}{nMuestras} \sum_{k=1}^{nMuestras} \beta(ray(S_i, d), S_j) \tag{2.11}$$

donde:

- d sigue la distribución coseno.
- $\beta(r, S_j)$ toma el valor 1 si el rayo $ray(S_i, d)$ interseca a S_j o 0 en otro caso.

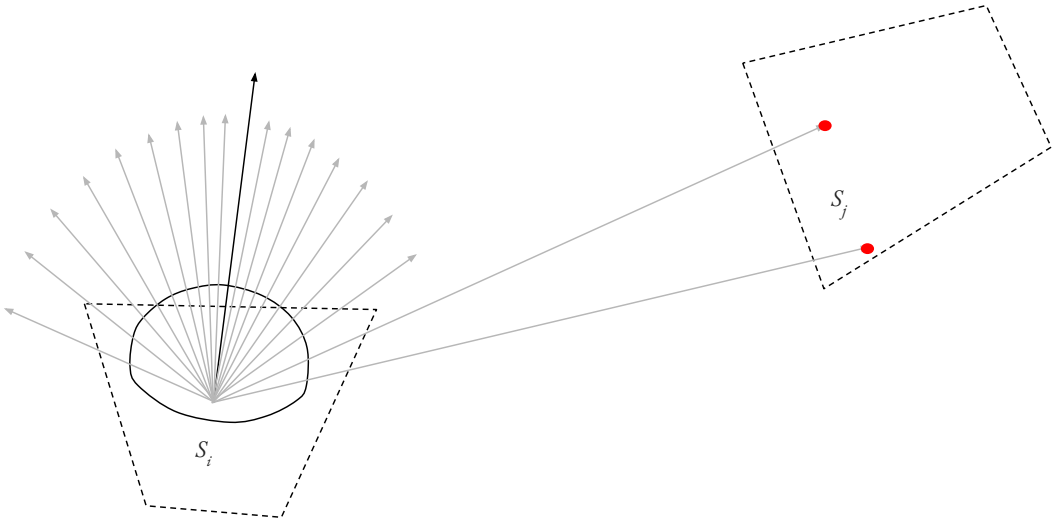


Figura 2.8: Representación gráfica del método de método de trazado de rayos para el cálculo de factores de forma

Cabe destacar que no es necesario utilizar una distribución de probabilidad con valores aleatorios o pseudo-aleatorios, sino que alcanza con utilizar una secuencia determinista de rayos que tengan una distribución que siga la del coseno.

Para esto, pueden utilizarse otras distribuciones para la dirección de traza. Particularmente, una de ellas es la propuesta por Beckers y Beckers [8], que presentan un método general de teselación de discos y hemisferios. Es deseable el hecho de que la propuesta para hemisferios genera un conjunto de celdas de igual factor de forma, es decir, las celdas del disco proyectado según la analogía de Nusselt (Figura 2.4) poseen la misma área. Esto hace que el método presente una calidad adecuada para la elección de las direcciones en la que se trazarán los rayos.

2.2.3. Extensión a superficies especulares

Originalmente, el método de cálculo de la radiosidad asume que todas las superficies son reflectores lambertianos, lo que supone que solo existirán reflexiones difusas cuando la luz interactúa con ellas. Sin embargo, en la mayor parte de las escenas del mundo real es necesario simular reflexiones especulares correctamente para obtener resultados que se asemejen a la realidad.

Por ello se ha desarrollado una extensión del método de radiosidad para superficies especulares o refractantes propuesto por Sillion, et al [9]. Los autores proponen extender el significado del término *factor de forma* a más que una mera relación geométrica entre parches. El nuevo factor de forma $\mathbf{F}_{ij}$ corresponde a la proporción de luz que sale de la superficie i y llega la superficie j luego de un número de reflexiones y refracciones especulares. Esta propuesta modifica completamente los algoritmos de cálculo de factores de forma. Los autores proponen un algoritmo de cálculo que consiste en el trazado de caminos desde S_i en una dirección arbitraria d bien distribuida. Luego, para cada camino trazado, se distribuirá el valor final del factor de forma dependiendo en la cantidad de superficies con las que interaccione el camino y sus coeficientes especulares como se observa en la Figura 2.9.

Este método supone el trazado de rayos recursivo. En primer lugar, se lanza un rayo desde la superficie S_i, si interseca una cara cuyo coeficiente de reflexión especular es mayor a cero se almacenará k (donde $k = \frac{1}{nMuestras}$) como contribuyente del factor de forma $\mathbf{F}_{ij}$. Luego, suponiendo que un rayo impacta S_k desde el camino (S_i, S_j) donde $\rho_j^{(s)} \geq 0$ se agregará $k\rho_j^{(s)}$ a $\mathbf{F}_{ik}$ y se procederá hasta que $\rho_z^{(s)} = 0$ para una superficie intersecada S_z o se alcance el máximo límite de recursión. Cabe destacar, que para que el algoritmo correctamente represente la realidad, se debe respetar la relación $0 < \rho_i^{(d)} + \rho_i^{(s)} \leq 1 \; \forall$ parche $i \in$ escena .

2.2.4. Cálculo del vector de radiosidades

Luego de computar la matriz $\mathbf{F}$ y dado los vectores de emisiones E y reflexiones R, resta computar el vector de radiosidades correspondiente para cada parche, denominado B.

Recordando la Eq. (2.4) y considerando $\mathbf{R}$ a la matriz diagonal que contiene todas las reflectividades, es posible deducir el problema al sistema de

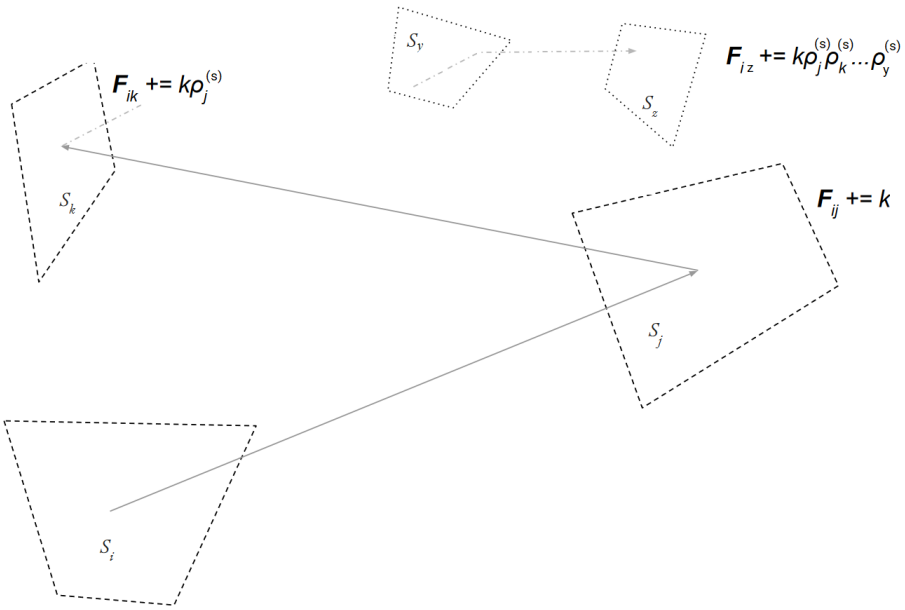


Figura 2.9: Representación gráfica del cálculo del factor de forma extendido donde $k = \frac{1}{N}$, con N muestras tomadas.

ecuaciones dado por:

$$E = (\mathbf{I} - \mathbf{RF})B \tag{2.12}$$

Los estudios de álgebra lineal modernos permiten la resolución de sistemas de ecuaciones de forma optimizada, dependiendo de las propiedades observadas.

Recordando las propiedades en la Sección 2.2.1, podemos observar que:

- $\sum_{j=1}^{N} \mathbf{F}_{ij} \leq 1 \forall i \in [1, N]$
- $\rho_i^{(d)} \leq 1 \rightarrow \mathbf{R}_{ii} \leq 1 \forall i \in [1, N]$ y $\mathbf{R}_{ij} = 0 \ \forall i, j \in [1, N], i \neq j$

Esto implica que las entradas de $\mathbf{RF}$ son siempre menores a 1, por tanto la matriz $(\mathbf{I} - \mathbf{RF}) = M$ es diagonal dominante ya que $R_{ii} \sum_{j=1}^{N} |F_{ij}| \leq 1 \forall i \in [1, N]$. Lo que garantiza la convergencia del uso de métodos de resolución iterativos o de factorización, como el algoritmo de Gauss-Seidel o la factorización LU.

Aunque los algoritmos clásicos de resolución de sistema de ecuaciones aplican a este problema, existen optimizaciones que hacen que su resolución se pueda aproximar de manera razonable con un costo computacional muy me-

nor. Para ello, considerando que la matriz $\mathbf{F}$ es diagonal dominante, podemos utilizar la equación 2.13 pues el *residuo* (el término agregado en cada iteración) se reduce de la siguiente forma: $\left\|\mathbf{RF}B^{(i+1)}\right\| < \left\|\mathbf{RF}B^{(i)}\right\|$. El método planteado en la Eq. (2.13) es el método de Jacobi.

$$B^{(i+1)} = \mathbf{RF}B^{(i)} + E \text{ con } B^{(0)} = E \tag{2.13}$$

Cabe aclarar, que el método planteado hasta el momento resuelve la radiosidad en un único canal. Es decir, no se toma en cuenta todo el espectro electromagnético de la luz, es por ello que puede establecerse una extensión del método. Esta extensión implica la existencia de tres vectores de reflexión, uno para cada canal *RGB* (del inglés *Red - Green - Blue*). Por tanto es necesario que se resuelvan tres y no un único sistema de ecuaciones, aunque es posible destacar que la matriz $\mathbf{F}$ permanece constante pues depende únicamente de la geometría de la escena. El único cambio en el sistema surge en la matriz $\mathbf{R}$ que pasará a depender del canal seleccionado: $\mathbf{R}_c$.

2.3. Bibliotecas gráficas

La computación gráfica es una herramienta sumamente útil para la generación de programas que faciliten tareas. Es por ello que ha sido necesario proveer bibliotecas estándar capaces de facilitar la aplicación de técnicas como la rasterización o el trazado de rayos utilizando las aceleraciones por hardware disponibles y de forma abstracta. Es por esto que generalmente se utilizan bibliotecas gráficas que abstraigan el hardware subyacente para el programador.

2.3.1. OpenGL

Existen un conjunto de bibliotecas de gráficas que soportan la rasterización (OpenGL, Vulkan, DirectX, Metal). En el contexto de este proyecto, se estudia el uso de Open Graphics Library (OpenGL).

OpenGL es una especificación de una Interfaz de Programación de Aplicación (API por sus siglas en inglés) diseñada por la organización Khronos Group. Su cometido es el dibujado de gráficos bidimensionales o tridimensionales utilizando el método de rasterización (ver Sección 2.2.2). Los distintos fabricantes de sistemas operativos y tarjetas gráficas proporcionan implementaciones que

se ajusten al hardware específico. Esta abstracción facilita la compatibilidad de las aplicaciones independientemente del hardware donde sean ejecutadas.

Arquitectura

La arquitectura base de la biblioteca es de cliente/servidor (ver Figura 2.10). El cliente es la aplicación que invoca funciones para el dibujado de gráficos y es ejecutado en la CPU. El servidor, que es ejecutado en la GPU, almacena los distintos buffers y ejecuta las funciones necesarias.

El cliente modifica los atributos a través de invocaciones a las funciones de prefijo `gl`, identificando el recurso afectado con valores enumerados (por ejemplo, `GL_TEXTURE_2D` representa un conjunto de texturas bidimencionales). Dado que la biblioteca es implementada como una máquina de estado, los atributos son recordados hasta que sean modificados nuevamente.

Estas invocaciones no son ejecutadas inmediatamente, sino que de forma similar a un buffer de entrada/salida son almacenados para ser ejecutados cuando sea necesario, es decir, cuando se requiera el dibujo de una nueva imagen. Esto hace que la ejecución de comandos sea asíncrona, y por tanto mejora el rendimiento previniendo la sincronización entre la CPU y GPU.

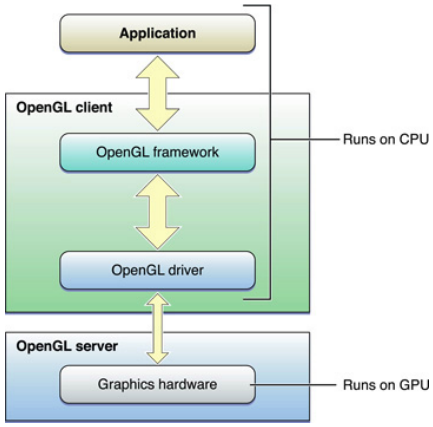


Figura 2.10: Vista general de la arquitectura de OpenGL

Extensiones

La inicialización de la máquina de estados depende directamente de la creación de un contexto que será utilizado para almacenar los datos. Este proceso depende fuertemente de la plataforma donde se ejecute la aplicación,

que depende entre otros del sistema operativo y/o del hardware utilizado. Por este motivo, existen bibliotecas que manejan la creación del contexto en diversas plataformas como SDL y GLFW.

2.3.2. Embree

Los distintos algoritmos para evaluar la intersección entre superficies y rayos han evolucionado en los últimos años, introduciéndose los conceptos de volumen acotante y jerarquías de escena. Esto resulta en un gran trabajo innecesario al momento de implementar algoritmos que se basan en el trazado de rayos de forma eficiente. Es por ello, que de manera similar a las APIs de dibujado de gráficos acelerados por hardware existen interfaces que facilitan la aceleración del trazado de rayos. En particular, Embree es una biblioteca creada por Intel con este propósito.

Embree expone un conjunto de funciones para realizar el trazado de rayos acelerado a través de componentes de hardware y software mediante la utilización del conjunto de instrucciones del paradigma SIMD (del inglés Single Instruction - Multiple Data), donde una única instrucción es ejecutada sobre un gran conjunto de datos. Por ejemplo, la ejecución concurrente de un conjunto de multiplicaciones en punto flotante a nivel de CPU. Además posibilita la generación de estructuras de aceleración, como las BVH (del inglés *Bounding Volume Hierarchies*). La arquitectura de la aplicación, diagramada en la Figura 2.11, demuestra los distintos algoritmos propuestos para la generación de estructuras de aceleración y algoritmos de intersección eficientes.

La biblioteca resuelve un conjunto de dificultades normalmente encontradas en todas las aplicaciones de algoritmos que involucren el trazado de rayos, entre ellas:

- **Multi-hilo:** Con el objetivo de ejecutar distintos kernels de traza de rayos de forma concurrente, la biblioteca provee de funciones *thread-safe* para el dibujado y la generación de estructuras de aceleración.
- **Vectorización:** Con el objetivo de optimizar el uso de la CPU, la biblioteca vectoriza los cálculos necesarios para aprovechar las instrucciones SIMD.
- **Soporte para múltiples CPUs:** La biblioteca provee de una capa de abstracción independiente del hardware donde se utilice.

- **Conocimiento del dominio extenso:** Dado que la biblioteca implementa las estructuras de aceleración y los algoritmos de intersección no es necesario tener un conocimiento completo del dominio para construir aplicaciones utilizando trazado de rayos.
- **Manejo eficiente de la memoria:** Se provee un manejo interno de la memoria RAMo para facilitar la visualización de escenas con gran cantidad de primitivas.

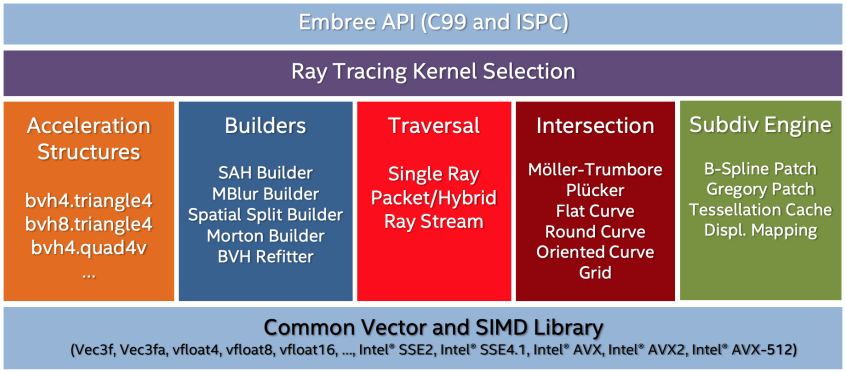


Figura 2.11: Vista general de la arquitectura de Embree

2.4. Trabajos relacionados

En esta sección se discuten alternativas propuestas para resolver el cálculo de la iluminación global en escenas con materiales difusos y especulares.

El algoritmo propuesto por Shirley [10] calcula la iluminación difusa utilizando dos pasadas. Este algoritmo difiere del propuesto por Sillion y Puech [9] en el sentido que se consideran distintos modelos de fuentes luminosas con propiedades particulares (luces puntuales, direccionales, de área).

En la primer pasada, el algoritmo calcula la componente difusa de todos los rayos que rebotan en al menos una superficie especular. Este método calcula los caminos que seguirán los rayos de luz provenientes de fuentes luminosas, es decir, se discretiza la cantidad de rayos emitidos por una fuente luminosa, cada rayo representa una fracción de la energía emitida.

Cuando existe una intersección, se divide la energía entre los cuatro nodos más cercanos (estos nodos almacenan la radiosidad) a través de una estimación para calcular qué área ocupa cada uno de ellos, de esta manera es posible generar un mapa de radiosidad para la superficie. Dado que la iluminación directa

(es decir, aquellos rayos que no se intersecan con superficies especulares) es calculada en la etapa de vista, solo es necesario computar los rebotes especulares, para ello se traza un número bajo de rayos distribuidos de forma uniforme para encontrar las zonas donde existan superficies especulares, luego se trazan rayos en esa dirección de forma "densa", que implica trazar una cantidad de rayos considerable en una dirección que no varía demasiado.

El segundo paso utiliza el método de radiosidad para calcular la iluminación difusa que involucra al menos dos superficies, nuevamente se omite la iluminación directa pues se calculará en la etapa de vista. Para ello, se emiten rayos desde cada superficie utilizando la distribución del coseno de manera equivalente a la propuesta por Malley.

Finalmente, cuando se dibuja la imagen final también se calcula la iluminación directa de forma estándar (ver Whitted [11]) sustituyendo el término de ambiente por el calculado en las pasadas anteriores.

Otro acercamiento al problema es el método propuesto de Kok [12] es una extensión para parches que están formados por superficies de Bézier, estas son superficies delimitadas por curvas de nombre homónimo que para una superficie definida con m puntos siguen la ecuación $c(u,v) = \sum_{i=0}^{m} c_i(v) B_i^m(u)$ donde c es el vector de desplazamientos, y B una función que genera la curva.

Los autores proponen la discretización de las superficies en puntos de muestreo dependiendo del área, luego simplemente se calcula el factor de forma de la superficie utilizando el método del hemisferio, agregando los resultados para cada punto. En caso de que un rayo intersecta una superficie especular, los autores proponen un método similar al de Sillion y Puech [9] donde se seguirá el camino del rayo mientras rebote en superficies especulares y arribe en una difusa, distribuyendo el factor de forma entre las superficies involucradas dependiendo del coeficiente de reflexión especular.

Una formulación distinta del problema, que también se focaliza en el uso de radiosidad es Holly y Torrance [13]. Los autores proponen, nuevamente, un método de dos pasadas donde el factor de forma está compuesto por tres partes, el factor de forma difuso y los factores de forma delantero y trasero. El primero, está intrínsecamente relacionado a la reflexión difusa y tiene el mismo comportamiento que el factor de forma definido por Cohen [3] mientras que los segundos se relacionan con el fenómeno de la refracción y son calculados integrando en el hemisferio opuesto por la normal del parche. La última componente son los factores de forma de ventana, que contienen la información

referente a la reflexión especular.

El método propone el uso del hemi-cubo para calcular los factores de forma delanteros, mientras que los traseros son calculados invirtiendo el hemi-cubo. Para calcular las componentes especulares, se utiliza un método comúnmente denominado dibujado de portales, donde se coloca una ventana virtual que distorsiona la proyección de la escena al mover la cámara, con la salvedad de que se opta por duplicar la geometría simétricamente en lugar de simplemente trasladar la cámara.

Capítulo 3

Solución propuesta

3.1. Alcance y objetivos

Este proyecto se centra en la implementación completa de una aplicación capaz de calcular tanto la matriz de factores de forma como el vector de radiosidad final. Se agrega especial énfasis en la comparación del rendimiento de los distintos métodos en la Sección 2 para escenas compuestas por triángulos y cuadriláteros.

Se propone comparar cuatro métodos para el cálculo de factores de forma:

1. Cálculo de factores de forma utilizando el hemi-cubo
2. Cálculo de factores de forma utilizando trazado de rayos
3. Cálculo de factores de forma extendidos utilizando rasterización
4. Cálculo de factores de forma extendidos utilizando trazado de rayos

Además, se propone implementar una interfaz de usuario que facilite la carga, edición y visualización de los objetos que componen la escena y sus respectivas propiedades (geometría, emisión inicial, coeficientes de reflexión difusa y especular, y radiosidad).

3.2. Proceso de desarrollo

Dada la naturaleza del proyecto, es deseable establecer una metodología de desarrollo para facilitar el proceso de seguimiento del progreso incluso cuando el equipo de desarrollo fue compuesto por un único integrante.

Para ello, internamente, se utilizó una metodología ágil de desarrollo similar a la conocida como *Kanban*.

Los principios claves del método aplicado a este proyecto fueron:

- La visualización sencilla del curso de trabajo (una lista de tareas a realizar conocida como *Backlog*)
- La limitación de las tareas en progreso con el objetivo de eliminar la sobrecarga de trabajo.
- Dirigir y gestionar el flujo de trabajo implica la priorización de tareas a realizar dada una cantidad finita de recursos.

La gestión de las tareas a realizar se llevó a cabo en el repositorio del proyecto, con tareas como las vistas en la Figura 3.1, donde se consideran un conjunto de tareas:

- Backlog: Las tareas a realizar, en orden de importancia.
- In progress: Las tareas actualmente en desarrollo.
- Done: Las tareas cuya funcionalidad fue completamente desarrollada y probada.

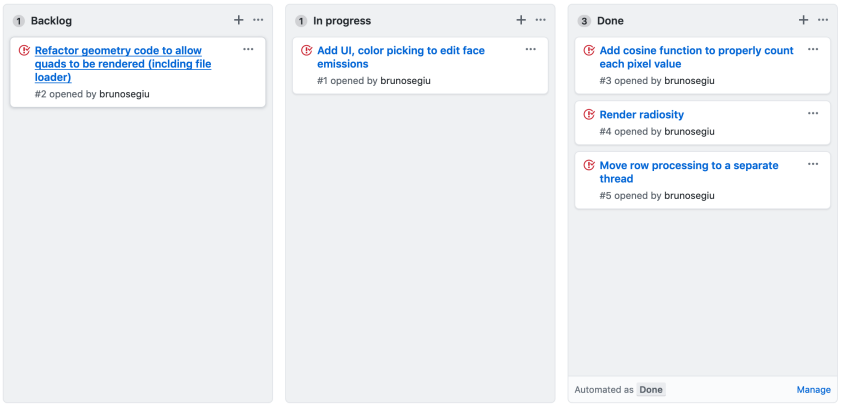


Figura 3.1: Tabla de Kanban utilizada en el proyecto

3.3. Diseño

Con la finalidad de evitar el alto acoplamiento, facilitar la extensión y reducir la cantidad de errores de integración se tomó la decisión de utilizar distintos módulos y sub-módulos que ofrezcan un conjunto de funcionalidades bien definido utilizando programación orientada a objetos. Esta decisión permite el

añadido de nuevas características y la optimización de ciertas funcionalidades independientemente de los demás módulos construidos.

El diseño de la solución comprende dos componentes principales, la interfaz gráfica de usuario (GUI, por su nombre en inglés) y el motor de renderizado.

3.3.1. Motor de renderizado

El paquete del motor de renderizado se compone de un conjunto de sub-módulos, el primero de ellos que maneja el pre-procesado de una escena, es decir, el cálculo de la matriz de factores de forma y la radiosidad. El siguiente conjunto de sub-módulos se ocupan del renderizado en tiempo real de la escena, así como la carga de modelos desde el disco duro, la modificación de materiales, entre otras funcionalidades detalladas en 3.3.2.

Módulo de geometría

El módulo de geometría encapsula la información de las escenas leídas desde el disco duro, además de adaptar y optimizar los formatos de las primitivas geométricas para ser utilizados en las APIs de dibujado de terceros. La clase `Scene`, diagramada en la Figura 3.2, carga distintos objetos desde el disco duro que son manejados como una instancia de `Mesh`.

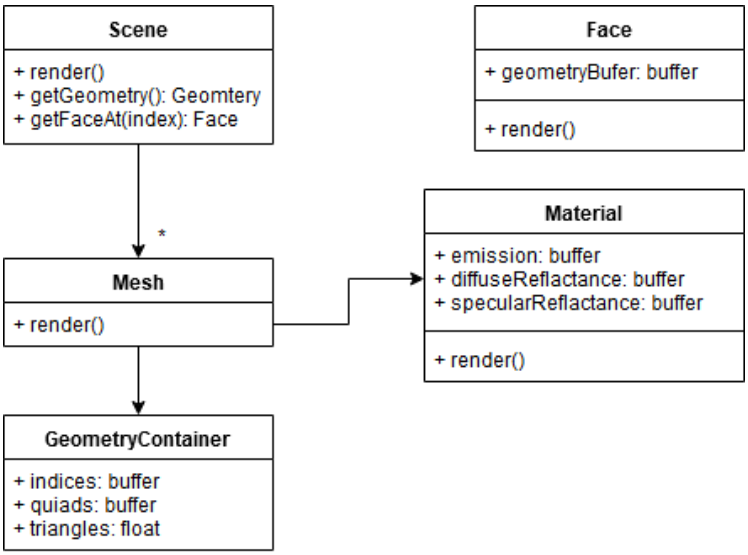


Figura 3.2: Módulo de manejo de geometría

Módulo de pre-procesado

El módulo de preprocesado se compone de un controlador principal (`PreprocessController` en la figura 3.3), que maneja el estado y la ejecución de los comandos de los distintos *pipelines* implementados que resuelven el cálculo de la radiosidad.

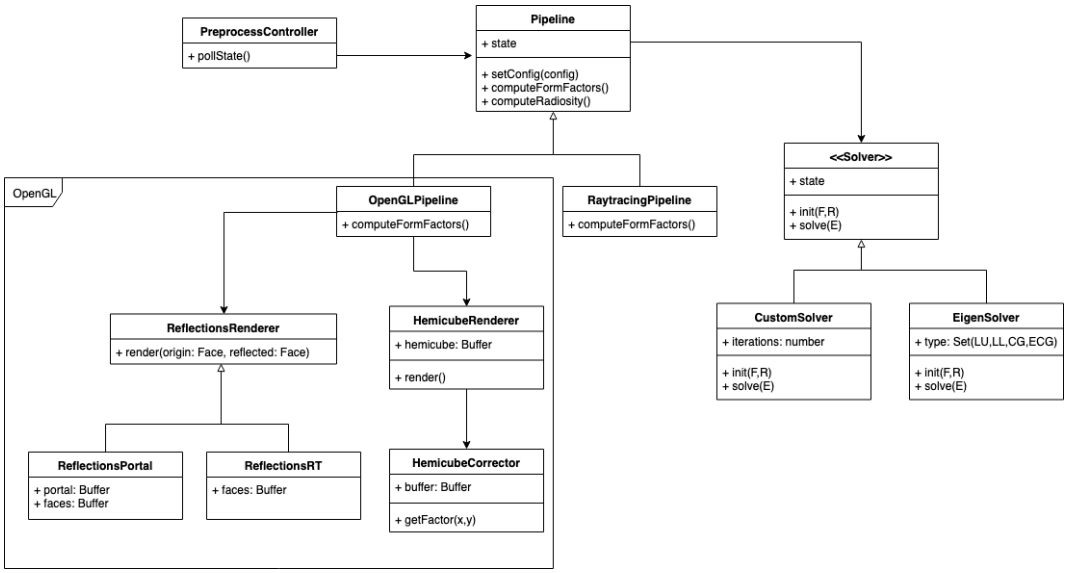


Figura 3.3: Arquitectura del módulo de pre-procesado

La clase Pipeline y sus derivadas de rasterización y traza de rayos definen un conjunto de tareas que se realizarán para calcular los factores de forma y el vector de radiosdiad. Las tareas suponen incializar el pipeline con información de la escena, calcular los factores de forma (eventualmente considerando la reflexión especular) y calcular el vector de radiosidad para los parches de la escena. Por ello, una instancia del pipeline es definida a partir de un conjunto de funciones ejecutadas en el siguiente orden:

1. `setConfig(scene, intrp, ref, chan, sol)`
2. `computeFormFactors()`
3. `computeRadiosity()`

La función `computeFormFactors()` variará dependiendo del método de cálculo elegido (Figura 3.3), donde puede utilizarse el método del hemi-cubo o el de raytracing. El primero de ellos utilizará un *pipeline* configurado utilizando una API de rasterización, mientras que el segundo utilizará una API capaz de calcular intersecciones utilizando rayos.

La ejecución de `computeRadiosity()` dependerá directamente del manejador `Solver` seleccionado por el usuario, este último ejecutará el algoritmo que calculará el vector de radiosidades para la escena, resolviendo el sistema lineal de radiosidad.

Módulo de visualización

El módulo de visualización se encarga de renderizar la escena actual desde el punto de vista seleccionado por el usuario. Además, debe tener la capacidad de mostrar las distintas propiedades de los materiales, tales como valor de emisión inicial, valor de reflexión especular y, visualización de geometría, para así facilitar la edición de las propiedades de los objetos y de sus caras. El proceso de renderizado comienza con una instancia de la clase `DisplayController` que dibuja un conjunto de imágenes correspondiente a las propiedades de los materiales, `SceneRenderer` en la Figura 3.4. Luego un dibujante de texturas `TextureRenderer` selecciona y convierte correctamente el resultado anterior a valores de tres canales (RGB). El módulo de visualización siempre toma una textura bidimensional como parámetro de salida.

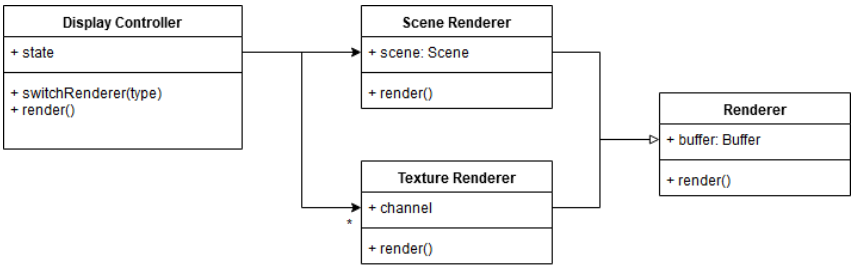


Figura 3.4: Arquitectura del módulo de visualización

3.3.2. Interfaz gráfica

El módulo de visualización (UI) utiliza una arquitectura basada en el paradigma del bucle de eventos (Figura 3.5), consiste en un bucle que detecta y maneja los distintos eventos recibidos por el sistema. Este método es útil para el manejo sencillo de la concurrencia en sistemas con múltiples hilos en ejecución, y es de fácil implementación pues procesa cada uno de los eventos completamente antes de procesar el siguiente.

El *bucle de eventos* procesa todos los eventos de la aplicación, lo que desencadena un conjunto de acciones que modifican su estado de la interfaz de usuario. Este estado es dibujado por un conjunto de componentes, que no son

más que presentadores del estado actual. Es decir, a partir de un conjunto de valores los presentan en un formato gráfico adecuado y sencillo de comprender.

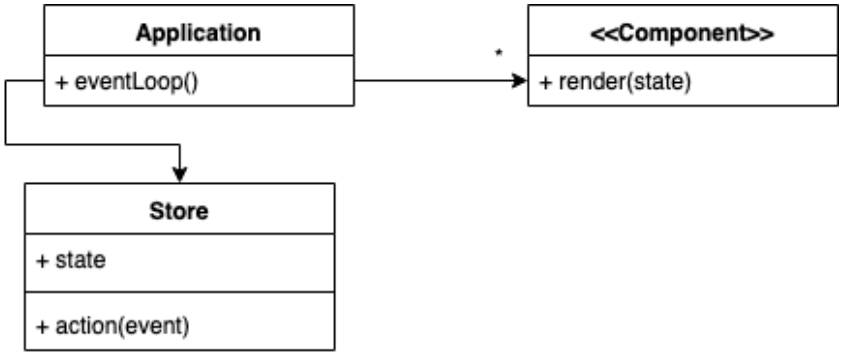


Figura 3.5: Arquitectura general del módulo de interfaz de usuario

Capítulo 4

Implementación

El siguiente capítulo desarrolla los detalles de implementación y optimización de los algoritmos para computar los factores de forma simples y extendidos.

4.1. Cálculo de factores de forma de la componente difusa

Tal como se expresa en la Sección 2.2.2, se implementaron dos algoritmos para el cálculo de factores de forma. El primero está basado en el método del hemi-cubo y el segundo está basado en el trazado de rayos de forma determinista utilizando un hemisferio de Beckes y Beckers [8].

4.1.1. Algoritmo del hemi-cubo

El algoritmo del hemi-cubo (Sección 2.2.2) fue implementado utilizando la API OpenGL que provee de interfaces de alto nivel para la programación de tarjetas gráficas, facilitando el uso del algoritmo del Z-Buffer y el manejo de memoria en la GPU.

Para implementar el cálculo de factores de forma se implementó la función `computeFormFactors` de la Figura 3.3. Recordando la arquitectura diseñada, la función debe computar completamente la matriz de factores de forma. Esta función sigue las siguientes de etapas:

1. En primera instancia, son configurados los *buffers* en memoria necesarios para representar el hemi-cubo. Para ello, se crea un *Frame Buffer Object*

en la GPU que está compuesto de 5 texturas, cada una de ellas correspondiente a una de las caras a dibujar. Cabe destacar, que estas texturas se componen de dos imágenes: una conteniendo enteros sin signo que son utilizados para representar el `id` de los parches vistos desde el centro del hemi-cubo y la restante contiene los valores de profundidad necesarios para el algoritmo del Z-Buffer.

2. En la segunda etapa se establecen las matrices de transformación de vista, es decir, las transformaciones que alinean el volúmen de vista al hemi-cubo. Esto implica trasladar el origen de vista hacia el baricentro de la cara en cuestión, y alinearla a su normal.
3. En la tercera etapa se procede a dibujar cada objeto en la escena desde el parche considerado en las cinco texturas que componen el hemi-cubo. Con el objetivo de tener el mejor rendimiento posible, se hace uso de los shaders de geometría para realizar una única llamada de dibujado por objeto. Este método solamente realiza un cambio de textura de dibujado, es decir, solo se liga las texturas correspondientes al hemi-cubo una única vez, en lugar de ligar una a una cada cara. Esto se debe a que esta es una de las operaciones más costosas según la Figura 4.1.

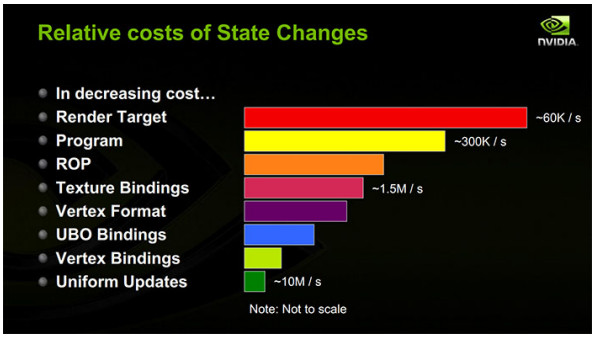


Figura 4.1: Costo de cambios de estado en OpenGL. Se muestra el costo de las distintas operacioens en orden inverso de cantidad de operaciones realizables por segundo para la tarjeta Nvidia GTX 480. [14]

Como puede apreciarse en el Algoritmo 1 solo se realiza una única llamada de ligamiento del hemi-cubo. Específicamente, la implementación sigue el siguiente patrón:

a) El shader de vértices es simplemente *passthough* lo que significa que

conecta las entradas provistas por la CPU con su salida.

b) El shader de geometría genera cinco primitivas donde cada una estará en las coordenadas correspondientes a los volúmenes de vista de las caras del hemicubo además de añadir un plano adicional de corte del dibujo necesario debido a la imposibilidad de que las caras laterales posean una resolución menor a la cara superior.

c) El shader de fragmentos corrige el identificador local `gl_PrimitiveId` a un identificador global a partir de variables uniformes que conservan el valor de caras cúbicas y triangulares que componen el objeto. Luego, escribirá el identificador de la cara detectada en la textura que le corresponda según el valor asignado por el *geometry shader*.

Algoritmo 1 Algoritmo de proyección de la escena en un hemi-cubo

```
function computarFF
    inicializarHemicubo() //inicializar los píxeles en 0
    loop parche ∈ escena
        alinearCamara(parche) //alinear la cámara a la normal del parche
        dibujar(escena) //dibujar la escena en el hemi-cubo (GPU)
        hemicubo ← leerHemicubo() //leer los píxeles del hemi-cubo
        procesarHemicubo(parche, hemicubo) //hallar la fila asociada al parche de F
    end loop
end function
```

4. En última instancia es necesario procesar la información del hemi-cubo dibujado para obtener una nueva fila de la matriz **F**. Para esto, se suman los delta factores de forma de los pixeles para cada parche de la escena. Este proceso puede ser realizado tanto en GPU como CPU y sigue el patrón que se muestra en el Algoritmo 2. Ambos métodos fueron implementados y se detallan a continuación:

   - Computar factores de forma en GPU: Se utilizan *compute shaders* para reducir las cinco texturas que componen el hemi-cubo en un único *Shader Buffer Object* que representa un arreglo de bytes en la GPU. En este caso, el arreglo representará una fila completa de la matriz. Para calcular cada entrada se utiliza una textura inmutable (es decir, no modificable) auxiliar que contiene los valores de corrección expresados en las Eqs. (2.7) y (2.8) en conjunción con la función `atomicAdd` para sumar las componentes de los factores de forma de cada elemento. Al ser la GPU un multiprocesador puede ocurrir que varios hilos intenten sumar en la misma celda de **F** al mismo tiempo y la operación `atomicAdd` obliga a realizar esas

operaciones secuencialmente. Es posible acceder a este buffer desde la CPU mediante la función `glMapBuffer`, que a través del dispositivo DMA (del inglés *Direct Memory Access*) permite la lectura de la memoria VRAM (localizada físicamente en la GPU) de forma directa, aunque requiere de la sincronización entre GPU-CPU. No obstante, dada la naturaleza de las tarjetas gráficas el uso de estructuras de control que evalúen condiciones en tiempo de ejecución generan divergencia de hilos y por tanto reducen drásticamente el rendimiento del algoritmo.

- Computar factores de forma en CPU: Con el objetivo de aumentar el rendimiento del algoritmo se utiliza la reducción en CPU, en este caso, se utiliza la función `glReadPixels` que sincroniza la GPU y copia el contenido de la memoria VRAM en la memoria RAM. Luego, se inician hilos de CPU que procesan la información de manera similar a la GPU, aunque de forma secuencial para eliminar la necesidad de sincronización. Esto se realiza concurrentemente con el procesamiento de nuevos hemi-cubos en la GPU, generando un buen nivel de paralelismo entre los dispositivos.

Algoritmo 2 Procesamiento de la fila de la matriz **F** asociada al parche, a partir de la información almacenada en una textura cúbica.

```
function procesarHemicubo(parche, hemicubo)
    fila ← [0, ..., 0] //inicializar fila
    loop pixel ∈ hemicubo :
        factor ← obtenerDeltaFF(pixel) //buscar el valor del delta ff
        parcheVisto ← obtenerIdParche(pixel) //computar identificador del parche
        if esValido(parcheVisto) then
            fila[parcheVisto] ← +factor //añadir la fracción de ff
        end if
    end loop
    F[parche] ← fila
end function
```

4.1.2. El algoritmo del hemisferio

El algoritmo del hemisferio (Sección 2.2.2) fue implementado utilizando la biblioteca de traza de rayos Embree que implica una solución alternativa al método del hemi-cubo. Esta biblioteca soporta el trazado de rayos en la CPU en múltiples superficies, en particular, triángulos y cuadriláteros utilizando BHV (del inglés *Bounding Volume Hirarchies*). Estas estructuras de datos se basan en árboles que sub-dividen la escena en un conjunto de volúmenes simples que encapsulan un grupo de primitivas geométricas y cada nivel garantiza

la reducción de tamaño de dichas estructuras. Su utilidad radica en la simplificación del cálculo de la intersección rayo-objeto, pues el cómputo de la intersección de un rayo con un volumen envolvente tiene un costo despreciable en comparación al muestreo de todas las primitivas que están contenidas en él. La aceleración se da cuando no hay intersección entre el volumen y el rayo, pues en caso de "fallo"se asegura a su vez el fallo en la intersección del rayo con todas las primitivas que el volumen contiene.

Algoritmo 3 Cálculo de una fila de los factores de forma utilizando traza de rayos

```
function computarFactores(face)
    fila ← [0, ..., 0]
    direcciones ← beckers(nMuestras) //pre-computo de las direcciones en hemisferio
    loopdireccion ∈ direcciones : //traza de rayos desde el baricentro
        interseccion ← trazarRayo(escena, baricentro, direcciones)
        parcheVisto ← obtenerIdParche(interseccion)
        if esValido(parcheVisto) then//si el id es válido (no es el id del vacío)
            fila[parcheVisto] ← + 1/nMuestras //añadir la fracción de ff
        end if
    end loop
    F[parcheVisto] ← fila
end function
```

De forma similar al método del hemi-cubo, es necesario posicionar el origen de cada rayo a trazar en el baricentro de la superficie. Luego, recordando la Eq. (2.11), se genera un conjunto de direcciones utilizando la distribución coseno o similar. Si bien originalmente se utilizó la generación de números aleatorios para obtener direcciones de rayos correctamente distribuidos, el uso de estos generadores perjudicó el rendimiento del algoritmo debido al grado de complejidad que tienen estos algoritmos además de resultar en la presencia de ruido en la solución final. Por esto que se decidió utilizar otro algoritmo que generan de direcciones deterministas en un hemisferio propuesto por Beckers y Beckers [8]. Estas direcciones son pre-calculadas y almacenadas previo al procesamiento. Esta solución puede considerarse como un muestreo determinista que permite obtener resultados con menor ruido, además de mejorar el desempeño computacional del sistema.

Luego se procede a la traza de rayos. $\mathbf{F}_{ij}$ se calculará como $\frac{nIntersecciones_{ij}}{nMuestras}$, esto significa que por cada rayo que parta de la superficie S_i impactando S_j se adiciona $\frac{1}{nMuestras}$ al valor de la entrada correspondiente en $\mathbf{F}$.

El muestreo de puntos partiendo del origen del hemisferio en las direcciones determinadas se calcula utilizando la función `rtcIntersect1` de la biblioteca Embree, que retorna, de forma similar a OpenGL, un identificador de primitiva relativo al objeto que se intersecta. Para transformarlo en un identificador

global se utilizan los valores obtenidos `geomID` y `primID` además de un mapa de *offsets* que contienen un número con la cantidad de primitivas que anteceden a un objeto. Obtenido el conjunto de primitivas vistas, resta reducirla para generar una fila de la matriz adicionando $\frac{1}{nMuestras}$ para cada rayo.

Para manejar los distintos hilos de ejecución se utilizó una combinación entre la biblioteca estándar `std::threading` y OpenMP. La primera se utilizó en combinación con `std::lock` para manejar los distintos hilos de la interfaz de usuario y del procesamiento de hemisferios. Mientras que por otro lado para manejar la traza de rayo se utilizó la extensión provista por OpenMP para paralelizar ciclos `for`, por lo que cada rayo se maneja en un hilo independiente de forma transparente.

4.2. Cálculo de factores de forma de la componente especular

El cálculo de factores de forma extendidos fue implementado en dos variantes para el método del hemicubo. La primera de ellas utiliza la técnica de rasterización para dibujar los rebotes especulares, mientras que la otra utiliza el trazado de rayos. Además, se implementó la extensión para el método del hemisferio utilizando trazado de rayos.

4.2.1. Extensión del método del hemicubo

Ambas variantes son métodos de "dos pasadas" donde se dibuja el hemicubo normalmente, y luego de determinar cuáles de las caras visibles tienen componente especular, se proyecta nuevamente la escena para determinar qué parches son visibles a través de la reflexión.

Los métodos utilizados son heurísticos pues se basan en aproximaciones a la reflexión para aprovechar el hardware al máximo. Hay un factor de error que depende drásticamente del área de los parches y de la disposición general de la geometría, ya que al contrario de la técnica de trazado de rayos, se conoce qué parches son visibles pero se desconoce el punto exacto del espacio que es el correspondiente a cada pixel del hemi-cubo ya que solamente se almacenan los identificadores de los parches. A su vez, con motivo de simplificar el algoritmo no se considerará el delta factor de forma en el calculo de la componente especular (ver Algoritmos 4 y 5).

La primer variante utiliza el método de dibujado de portales en la GPU, mientras que la segunda fue implementada utilizando *trazado de rayos* en la CPU.

Dibujado de portales

El dibujado de portales es una técnica que emplea la rasterización para recortar el *frame buffer* en cierta área arbitraria. Particularmente, estos puntos son aquellos cubiertos por un polígono que llamaremos portal. Este portal será ubicado en el entorno tridimensional, su comportamiento emulará aquel de una ventana o una puerta. Normalmente se utiliza esta técnica para optimizar el dibujado de escenas donde la oclusión entre objetos es alta. También se utiliza en la simulación de espejos planos.

Este método puede ser realizado de forma sencilla utilizando la GPU debido a los `Stencil Buffers`, que son similares a los *buffers* de profundidad, pero almacenan información que indique el programador. El programador almacenará cierta información en este buffer, que será utilizada para decidir qué *fragmentos* pasan la prueba del `Stencil Test`. Esta prueba es una función booleana que decidirá si un elemento será dibujado o no. En este caso, los elementos que se encuentren fuera del portal fallarán la prueba.

En la implementación propuesta, el primer paso consiste en dibujar un parche cuyo coeficiente de reflexión especular es mayor a cero, como se aprecia en la Figura 4.2. La imagen resultante se almacenará en un *stencil buffer*, como se nota en el Algoritmo 4. Luego, manteniendo el mismo volumen de vista, se dibujarán los identificadores de los parches de forma similar al dibujado del hemicubo, salvo que en una textura bidimensional y utilizando el stencil buffer anteriormente mencionado. Es necesario además establecer un plano de corte en la superficie especular, con el objetivo de truncar el volúmen de vista para evitar el dibujado de los objetos que se encuentren detrás del espejo. Este proceso se realiza a una menor resolución, con el objetivo de mejorar el rendimiento y minimizar el costo de transferencia de memoria.

Finalmente, se obtienen los identificadores de las caras reflejadas. En caso de que existan caras reflejadas que también tengan un componente especular, se vuelven a procesar. Este procesamiento recursivo ocurre un número finito de veces, porque en este proyecto se dibujan caminos de hasta 10 rebotes. En caso de no existir superficies especulares en la imagen se utilizarán los

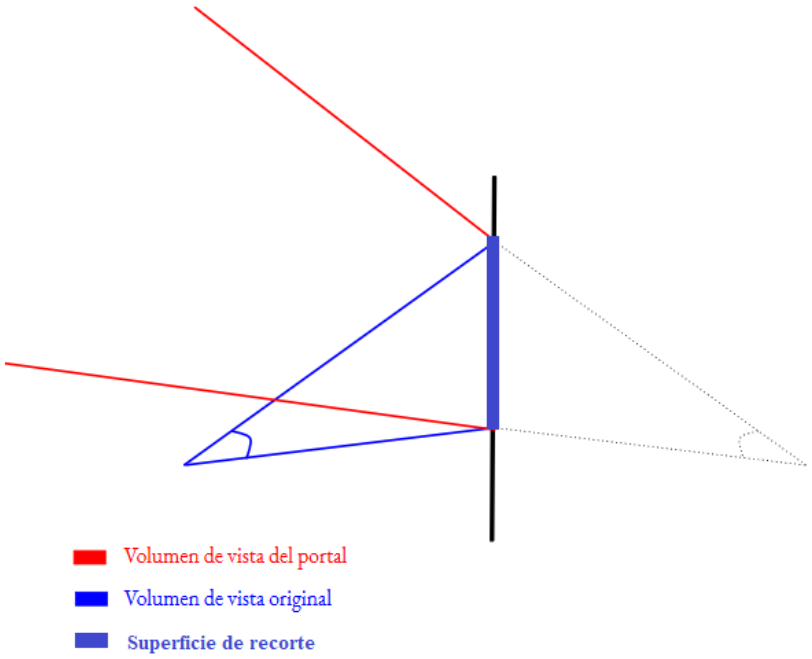


Figura 4.2: Generación del volumen de vista para un espejo. El volúmen de vista refiere a uno de los planos del hemi-cubo.

identificadores obtenidos para distribuir el factor de forma correspondiente al fragmento del hemicubo entre los parches visualizados.

Método híbrido

El método híbrido consiste en la utilización del trazado de rayos para computar qué parches son visualizados desde el hemicubo a través de parches especulares. Es decir, en lugar de la rasterización se utiliza la técnica de traza de rayos para calcular únicamente los caminos de reflexión especular que toma cada rayo que parte del hemicubo. Para ello, se trazarán rayos desde un conjunto de puntos pertenecientes al parche especular de manera uniformemente distribuida. La dirección es la que corresponde a la del volumen de vista, es decir, la que está dada por la diferencia entre los baricentros del parche de origen y puntos aleatorios en una grilla uniforme en el distribuidos en el parche especular. Las caras vistas, de tener componentes difusas, son las que aportarán fracciones de valor al factor de forma.

Esto emula el fenómeno de la reflexión, aunque introduce errores al aproximar la dirección real de reflexión. Debido a que no se han de considerar las posibles oclusiones parciales entre las caras, dado que no se trazan rayos hacia

Algoritmo 4 Cálculo de las caras vistas utilizando dibujado de portales

```
function dibujarPortal(parche, parcheObj)
    origen ← parche.obtenerBaricentro()
    direccion ← parche.obtenerNormal()
    configurarCamara(origen, direccion)
    seleccionarVertices(parcheObj)
    dibujarStencil() // dibujar área que se recortará
    refDir ← reflejar(direccion, parcheObj.obtenerNormal())
    refOrig ← simetrico(origen, parcheObj.obtenerPlano())
    configurarCamara(refOrig, refDir)
    seleccionarVertices(escena)
    dibujar(escena)
    parchesVistos ← leerBuffer()
    loopparcheRef ∈ parchesVistos
        if esValido(parcheRef) then
            fila[parcheRef] ← +(1/nMuestras) // en lugar de delta FF se usa 1 / nMuestras
            if esEspecular(parcheRef) then // si el coeficiente de reflexión especular es no nulo
                dibujarPortal(parcheObj, parcheRef)
            end if
        end if
    end loop
end function
```

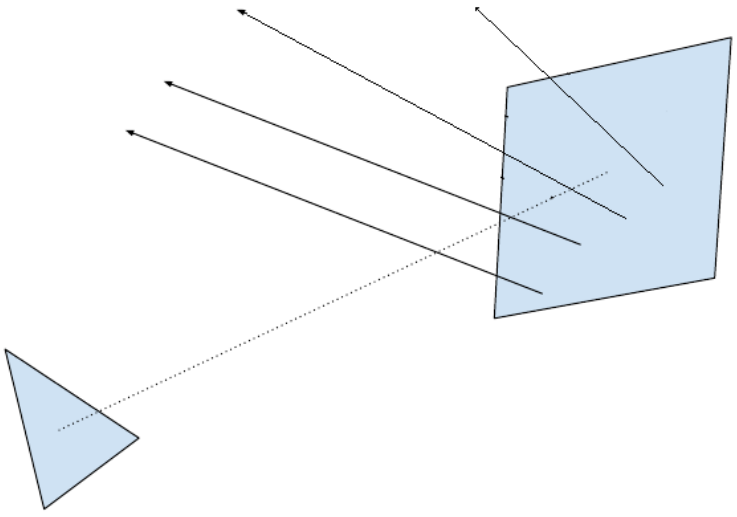


Figura 4.3: Visualización del rebote de rayos al impactar en parches especulares

el espejo en la dirección de los pixeles visibles en el hemi-cubo, sino que se eligen puntos al azar.

Algoritmo 5 Cálculo de las caras vistas utilizando trazado de rayos

```
function dibujarReflexiones(parche, parcheObj)
    origenes ← puntosEnGrilla(parcheObj)
    loop origen ∈ origenes
        dir ← normalizar(parche.baricentro() − parcheObj.baricentro())
        refDir ← reflejar(dir, parcheObj.obtenerNormal())
        parcheRef ← trazarRayo(origen, refDir)
        if esValido(parcheRef) then
            fila[parcheRef] ← +(1/nMuestras)
            if esEspecular(parcheRef) then
                dibujarReflexiones(parcheObj, parcheRef)
            end if
        end if
    end loop
end function
```

4.2.2. Extensión del método del hemisferio

En el caso del trazado de rayos, la extensión de los factores de forma es prácticamente trivial. Comprende la extensión de la función `traceRay()`, que en lugar de retornar un único valor para la cara vista, retornará un conjunto de pares de identificadores de caras y fracción de factor de forma. Básicamente, si el rayo inicial interseca una cara j cuyo coeficiente de reflexión especular ($\rho_j^{(s)}$) es mayor a cero se almacenará de k (donde $k = \frac{1}{nMuestras}$) como contribuyente del factor de forma $\mathbf{F}_{ij}$ y se calcularán las siguientes intersecciones con el *residuo* de la reflexión que se distribuirá entre los parches reflejados. Es decir, suponiendo que un rayo impacta S_k desde el camino (S_i, S_j) donde $\rho_j^{(s))} \geq 0$ se agregará $k\rho_j^{(s))}$ y se procederá de forma recursiva hasta que $\rho_z^{(s)} = 0$ para una superficie intersecada S_z o se alcance el máximo límite de recursión cómo se aprecia en la Figura 2.9.

4.3. Cálculo de la radiosidad

Recordando la Sección 2.2.4, se han propuesto dos métodos para resolver el sistema, y calcular la radiosidad $\mathbf{B}$. El método 'exacto' supone la resolución del sistema de ecuaciones dado por $((\mathbf{I} - \mathbf{RF})\mathbf{B} = E$, mientras que el segundo método está regido por el esquema iterativo dado por la Eq. (2.13).

En el primer caso, debido a que las características de algunas escenas de prueba utilizadas muestran que las matrices de factores de forma pueden ser dispersas (es decir, una gran cantidad de entradas de la matriz son nulas) la

implementación se realizó utilizando la biblioteca de álgebra lineal *Eigen*. En este caso, dadas las características de la matriz se optó por añadir soporte para los siguientes métodos:

- De factorización
  - Descomposición LU: En Matrices no singulares (se ha demostrado que $\mathbf{I} - \mathbf{RF}$ no lo es), la descomposición $\mathbf{LU}$ genera dos matrices tal que $\mathbf{I} - \mathbf{RF} = \mathbf{M} = \mathbf{LU}$ con $\mathbf{L}$ triangular inferior y $\mathbf{U}$ triangular superior. Fácilmente se puede comprobar que $\mathbf{M}^{-1} = \mathbf{U}^{-1}\mathbf{L}^{-1}$. Por tanto, $B = (\mathbf{U}^{-1}\mathbf{L}^{-1})E$.
- Iterativos
  - Método de Jacobi: Dada una matriz cuyo radio espectral menor a uno, el método supone el uso de fórmulas como iteración de punto fijo. Dada la matriz $\mathbf{M} = \mathbf{D} + \mathbf{R}$ donde $\mathbf{D}$ es diagonal y $\mathbf{R}$ es la suma de las matrices $\mathbf{L}$ y $\mathbf{U}$. Se resuelve el sistema escribiéndolo de forma tal que la sucesión $x^{(k+1)} = \mathbf{D}^{-1}(r - \mathbf{R}^{(k+1)})$ converge al valor final.

Por otro lado, se implementó el método completamente iterativo dado por la Eq. (2.13). Este método es similar en precisión a los métodos iterativos mencionados anteriormente pues no se calcula el valor exacto de B. Para su implementación se utilizó la biblioteca *Eigen* para realizar la multiplicación de matrices dispersas con vectores de radiosidad, que tienen un largo fijo pues cada entrada representa el valor de la radiosidad en cada parche.

Finalmente, luego de calcular el vector de radiosidad se procede a la interpolación y ajuste de resultados. Dado que en OpenGL solo es posible agregar atributos a nivel de vértice y no a nivel de primitiva (parche o polígono), es necesario generar un vector extendido donde se replica el valor asignado por cara a cada uno de los vértices.

Este proceso puede realizarse de forma trivial, simplemente copiando valores o aplicando interpolación a nivel de geometría como el usado en el modelo de iluminación de *Gouraud* [15]. Este proceso implica balancear el valor de radiosidad para cada vértice asignándole el promedio del valor de radiosidad de cada cara adyacente(como se aprecia en la Figura 4.4) en que se encuentre como muestra el Algoritmo 6.

Algoritmo 6 Algoritmo de interpolación de radiosidad para vértices

```
function interpolar(B)
    loop vertice ∈ escena
        temp ← 0
        parches ← parches donde vertice ∈ parche
        loop parche ∈ parches
            temp ← +B(parche)
        end loop
        radiosidadVertice ← temp / cantidad(parches)
    end loop
end function
```

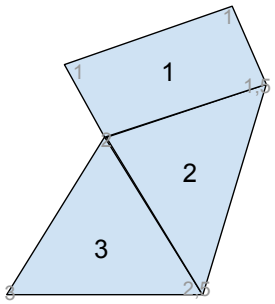


Figura 4.4: Ejemplificación del algoritmo de interpolación

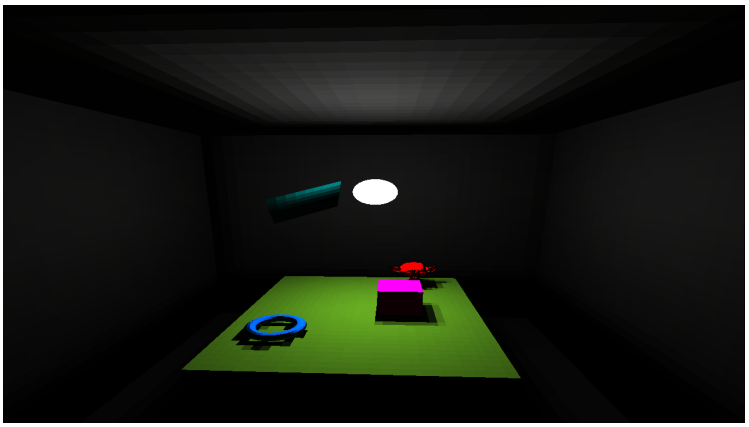
Esta técnica genera resultados con figuras de colores menos facetados, ocultando la discretización que se realizó para aplicar el método, como se aprecia en la Figura 4.5.

4.4. Visualización de resultados y resultados intermedios

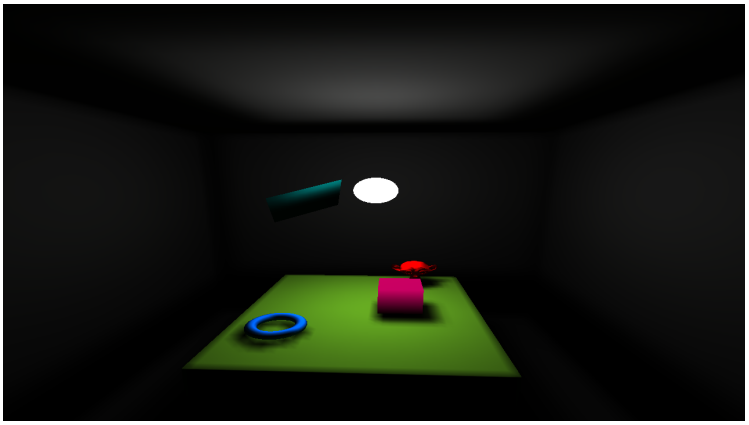
La visualización de resultados finales e intermedios se implementó utilizando el método de *dibujado a textura en capas*. Las texturas en capas son un conjunto de imágenes de igual resolución que contienen distinta información.

El algoritmo implementado no dibuja la escena directamente en el *frame buffer* global de la pantalla. Por el contrario, se dibuja en una textura auxiliar de varios niveles (cada nivel corresponde a una propiedad distinta de la escena). El primer nivel contiene la información del identificador de las caras, el segundo nivel el valor de radiosidad, el tercero el valor de emisión inicial, y el último nivel contiene los valores de los coeficientes de reflexión difusa.

Finalmente, para generar la textura que se mostrará en pantalla se utiliza un cuadrilátero unitario. Esta es una técnica estándar para proyectar un valor



(a) Facetado

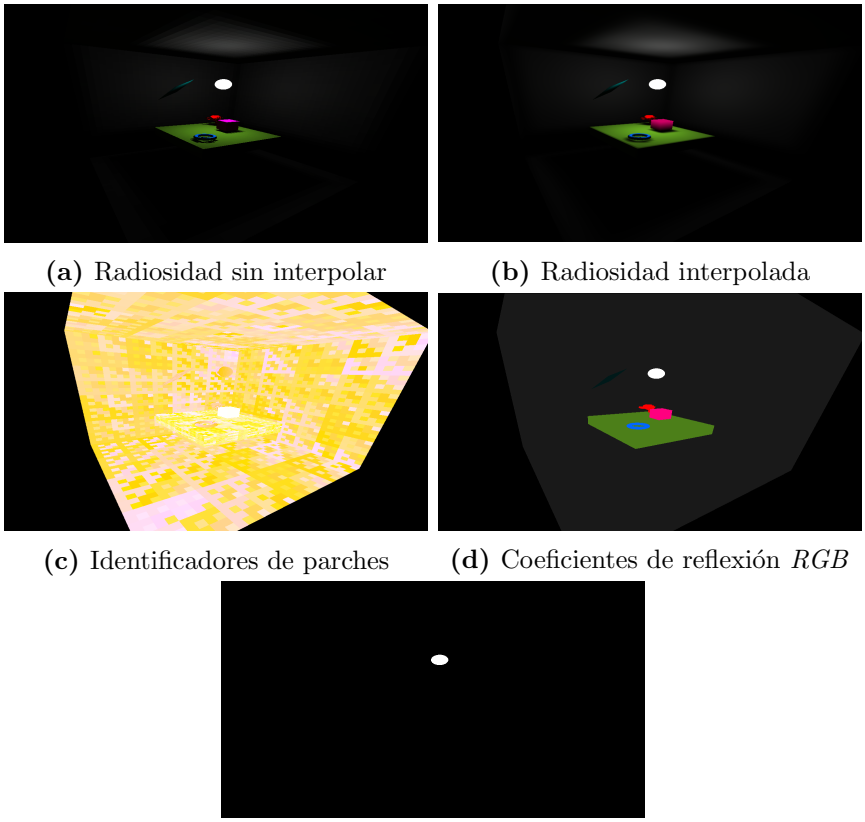


(b) Interpolación de Gouraud

Figura 4.5: Dibujado utilizando distintas funciones de interpolación

contenido en un buffer interno. Dependiendo de la propiedad que seleccione el usuario, se seleccionará uno de estos niveles para desplegar en pantalla.

Este método, además, añade la posibilidad de implementar la técnica de *picking* que involucra el reconocimiento de la selección que realiza el usuario. Para ello, basta obtener el valor del fragmento (`glReadPixels`) que se encuentra en las coordenadas del puntero dentro de la textura.



(a) Radiosidad sin interpolar **(b)** Radiosidad interpolada

(c) Identificadores de parches **(d)** Coeficientes de reflexión *RGB*

(e) Coeficientes de emisión

Figura 4.6: Vistas seleccionables por el usuario a través de la interfaz gráfica.

4.5. Interfaz de usuario

La interfaz de usuario fue implementada utilizando la biblioteca de dibujado de interfaces gráficas en modo inmediato *ImGui*. Este método de dibujado implica que los comandos de la interfaz se ejecutan inmediatamente, de forma tal que los resultados son guardados en una máquina de estados. Los componentes dibujados dependen directamente del estado interno de la aplicación. Este método fue utilizado en versiones anteriores de OpenGL o Direct3D. Esta

biblioteca implica la simplificación de la programación dada su capacidad de extenderse a diversas plataformas y por minimizar el impacto en el rendimiento de la aplicación en caso de ser utilizada.

Entre los componentes implementados que se observan en la Figura 4.7 se encuentran:

- Menú: El menú principal permite importar o exportar geometría (en formato Waveform OBJ utilizando un parser también implementado) y sus propiedades además de la matriz de factores de forma y editar las configuraciones del motor de dibujado.
- Panel de geometría: El panel de geometría permite editar el modo de seleccionado (cara, objeto) y visualizar qué cara se ha seleccionado.
- Panel de preprocesado: Este panel permite configurar y ejecutar las dos etapas de preprocesado (cálculo de factores de forma y radiosidad).
- Panel de iluminación: Permite editar características de los materiales de los objetos como los coeficientes de reflexión difusa, especular y emisión.
- *Log*: El *Log* imprime un conjunto de propiedades y detalles del proceso que pueden resultar interesantes, como por ejemplo los tiempos empleados.

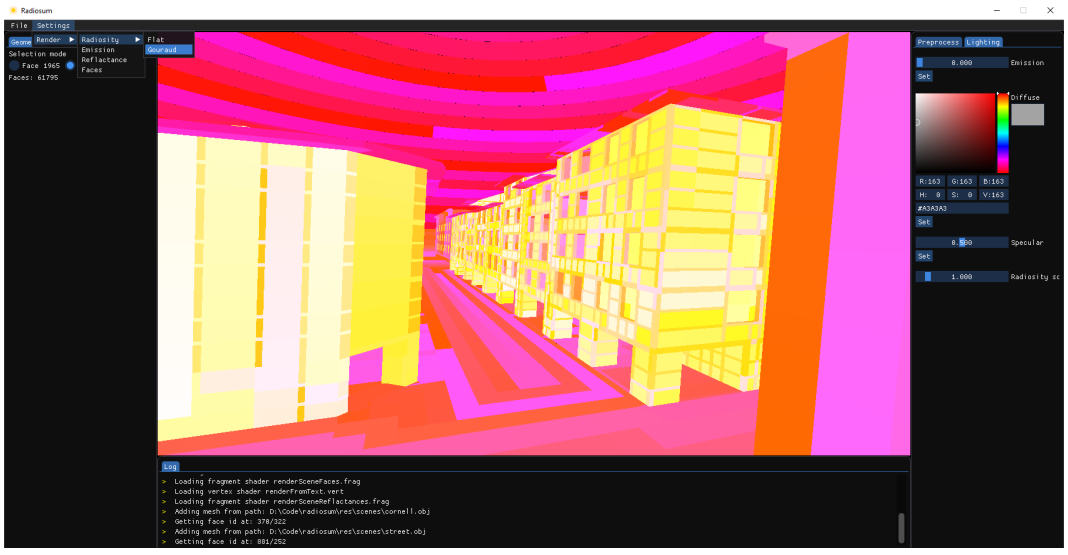


Figura 4.7: Interfaz gráfica implementada.

Capítulo 5

Experimental

En este capítulo se muestran detalles de las pruebas realizadas, con el objetivo de determinar las características positivas y negativas de los algoritmos implementados en las dimensiones de rendimiento computacional y precisión de los resultados.

5.1. Ambiente de prueba

A continuación, se presentan tanto el hardware utilizado en las pruebas (Tabla 5.1) así como las versiones del software utilizado (Tabla 5.2), de forma que los resultados puedan entenderse en términos relativos al entorno de ejecución utilizado.

Procesador	Intel i7 8700K - 12 CPUs - 3.7 GHz
GPU	Nvidia GeForce GTX 1070 Ti - 8 GiB VRAM
RAM	32 GiB - 2667 MHz

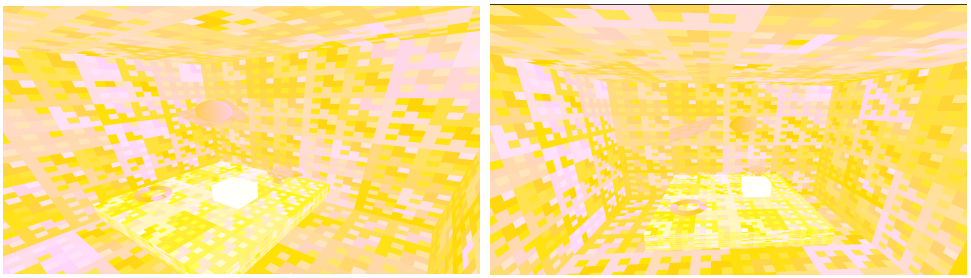
Tabla 5.1: Características del hardware utilizado

SO	Windows 10 Pro
Embree	v3.5.2
OpenGL	v4.5

Tabla 5.2: Características del entorno de desarrollo utilizado

5.2. Escenas

Con el objetivo de obtener resultados comparables para los distintos algoritmos y configuraciones se plantea el uso de dos escenas particulares de prueba, con distintas variaciones en los materiales que componen cada una de ellas.



(a) Lateral **(b)** Frontal

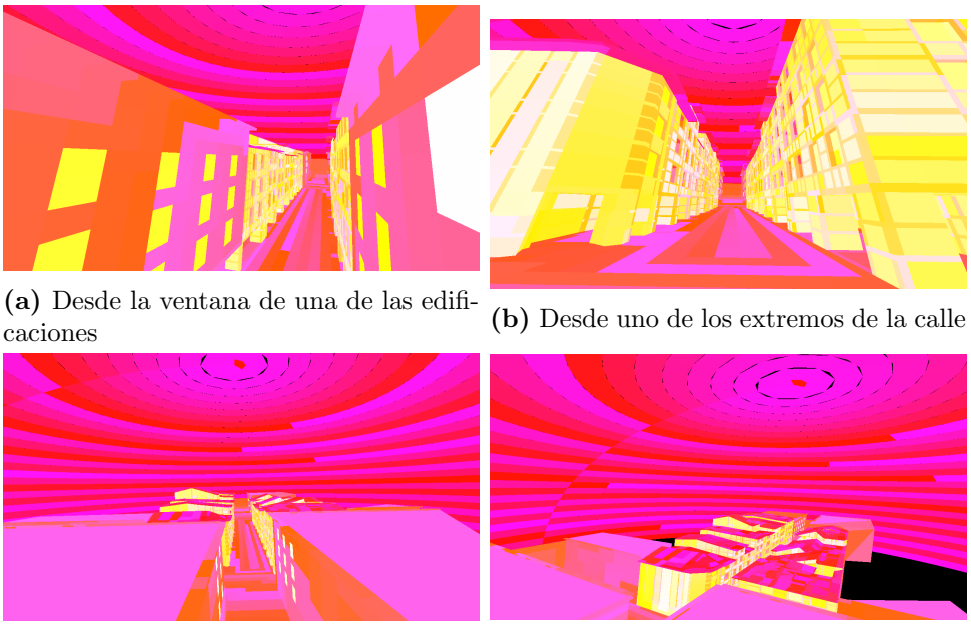
Figura 5.1: Vistas de la escena *Conrnell Box*.

Se denomina *Escena - Cornell Box* a la mostrada en la Figura 5.1. Se basa en un tipo de escena comúnmente usado en el que se ubican objetos en el interior de un cubo, donde debajo están los objetos de prueba y en el nivel superior reside el objeto que emitirá luz. Esta escena cuenta con siete objetos: el cubo, una esfera que oficia de luz, y cinco objetos compuestos por diversas primitivas. En total, existen 12.922 polígonos de las cuales 96 son triángulos y 12.826 son cuadriláteros.

Se denomina *Escena - Calle* a la referente a la Figura 5.2. La escena está constituida por dos objetos, el primero de ellos es una cúpula (hemisferio) subdividida en 2.407 cuadriláteros de igual área, cuyo objetivo es representar el cielo. Por otro lado, el segundo objeto es una representación de una porción de una calle en el barrio de Petit Bayonne, localizado en Bayona, Francia cuyas imágenes se aprecian en la figura 5.3. El modelo fue construido por Beniot, Acuña, et al. [16] con el objetivo de estudiar el fenómeno de la transferencia de calor a la escala de calle utilizando el método de elementos finitos. En total, la escena cuenta con 61.795 parches cuadrangulares.

5.3. Casos de prueba

Se proponen casos de prueba utilizando las escenas descritas en la Sección 5.2, compuestas por diversos materiales.



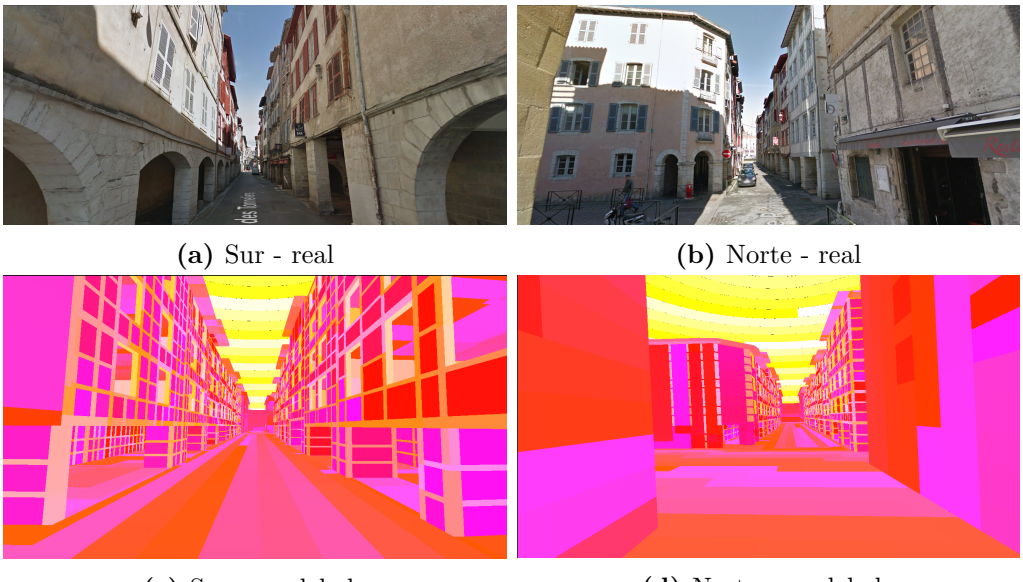
(a) Desde la ventana de una de las edificaciones

(b) Desde uno de los extremos de la calle

(c) Aérea I

(d) Aérea II

Figura 5.2: Vistas de la escena *Calle*.



(a) Sur - real

(b) Norte - real

(c) Sur - modelada

(d) Norte - modelada

Figura 5.3: Comparación entre el fotografías reales del modelo *Calle* y su representación tridimensional.

5.3.1. Métricas consideradas

Con el objetivo de medir correctamente las ventajas y desventajas de cada método de cálculo de factores de forma simples y extendidos que se han propuesto, se define un conjunto de métricas para evaluar su optimalidad en distintas dimensiones. Cada dimensión se aplica dependiendo del caso considerado.

- Rendimiento
  - Tiempo de ejecución: Se registra el tiempo empleado en calcular completamente la matriz de factores de forma.
- Matriz de factores de forma: Se compara la matriz de control $\mathbf{F_C}$, calculada utilizando la técnica de trazado de rayos con una gran resolución (3.145.728 rayos).
  - Error relativo promedio por fila: $Ep_i = \sum_{j=1}^{N} \frac{|\mathbf{F_C}_{ij} - \mathbf{F}_{ij}|}{N\mathbf{F_C}_{ij}}$
  - Error relativo máximo por fila: $Em_i = \text{máx}_{j=1}^{N} \frac{|\mathbf{F}_{ij} - \mathbf{Fc}_{ij}|}{\mathbf{F_C}_{ij}}$
- Vector de radiosidad (dado el vector B, y el vector de control Bc, calculado a partir de la matriz de factores de forma $\mathbf{F_C}$):
  - Error relativo promedio de radiosidad: $Ep = \sum_{i=1}^{N} \frac{|B_i - B_{Ci}|}{NB_{Ci}}$
  - Error máximo de radiosidad: $Em = \text{máx}_{j=1}^{N} |B_j - Bc_j| B_{Ci}$
- Valuación cualitativa:
  - Calidad de resultados: Se evaluarán los resultados esperando que se asemejen a la realidad.

Las definiciones fueron concebidas con la idea de que es necesario controlar el error de cada etapa del método. Se debe tener especial consideración con los términos geométricos, pues el resultado obtenido en la matriz de factores de forma incidirá en los cálculos posteriores. Para ello, se consideró una buena opción obtener errores relativos por fila, es decir, por parche. Un error promedio bajo asegura que en la generalidad los resultados sean aceptables, mientras que el error máximo controla la varianza del error percibido con el objetivo de evitar casos excepcionales. De la misma manera se considera el error observado en el resultado final, es decir, en el vector de radiosidad. Estos valores incidirán directamente en la calidad de la imagen, aunque pueden ser afectados por el error de la etapa anterior.

5.3.2. Descripción de casos de prueba

1. *Prueba difusa*: Se utilizan materiales estrictamente difusos en ambas escenas, cuyos colores no varían a lo largo de las pruebas realizadas. De esta manera se desactiva cualquier interacción especular. Se escoge un conjunto de parches que ofician de fuente luminosa. En caso de la escena *Calle*, se seleccionan 10 parches de la cúpula hemisférica para emular al sol. Por otro lado, para la escena *Cornell Box* se utiliza la bola central como fuente luminosa.
2. *Prueba especular*: Se utilizan materiales difusos y especulares en ambas escenas, con una cantidad reducida de estos últimos. En caso de la escena *Cornell Box* se utiliza el plano ubicado en el centro como reflector, mientras que en la escena *Calle* se utiliza una selección de ventanas. Para cada *pipeline* (completo) implementado se computa la radiosidad registrando el tiempo de renderizado según la cantidad de muestras configurada.
3. *Prueba conjunta*: En la escena *Cornell Box* se computarán dos variantes. En una de ellas se utilizan superficies con materiales exclusivamente difusos y en el segundo caso se añaden espejos, con el objetivo principal de destacar diferencias visuales percibidas al utilizar la extensión implementada.
4. *Prueba de stress*: Se utiliza gran cantidad de espejos en ambas escenas.

5.3.3. Resultados observados

En esta sección se presentan los resultados observados para los casos de prueba planteados. En particular, los resultados se detallan en función de la cantidad de muestras tomadas en cada hemi-cubo o hemisferio según corresponda. Para realizar una comparación justa, se utiliza la misma cantidad de pixeles totales del hemi-cubo como la cantidad de rayos lanzados en el hemisferio. Por ejemplo, para un hemi-cubo donde la cara frontal tiene dimensiones 32x32 pixeles el total de pixeles es de 32x32x3, totalizando 3072 pixeles. Por lo tanto, su caso equivalente en Embree corresponde a lanzar 3072 rayos.

Caso de prueba I (Reflexión difusa)

En este caso, se observa en la Tabla 5.3 que el método del hemi-cubo tiene un rendimiento considerablemente superior al de la traza de rayos. Cabe des-

Muestras		Tiempo de ejecución (s)			
		Cornell Box		*Calle*	
OpenGL	Embree	OpenGL-D	Embree-D	OpenGL-D	Embree-D
32x32x3	**3072**	7	**3**	68	14
64x64x3	**49512**	**10**	30	**128**	174
128x128x3	**196608**	**31**	116	**248**	665
256x256x3	**786432**	**251**	446	**1213**	2565
1024x1024x3	**3145728**	**992**	1778	**4018**	7511

Tabla 5.3: Resultados obtenidos para el primer caso de prueba. El número de muestras representa la cantidad de píxeles (en OpenGL) y rayos (en Embree) dibujados.

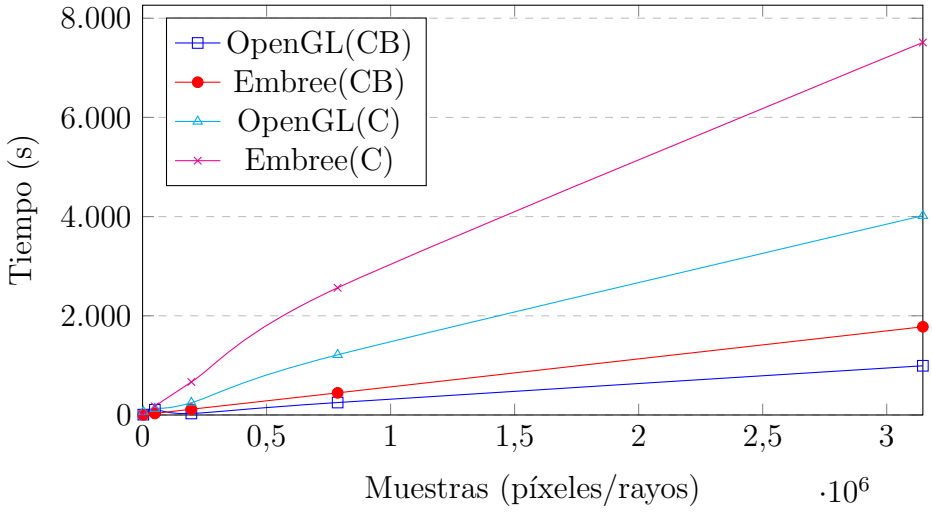
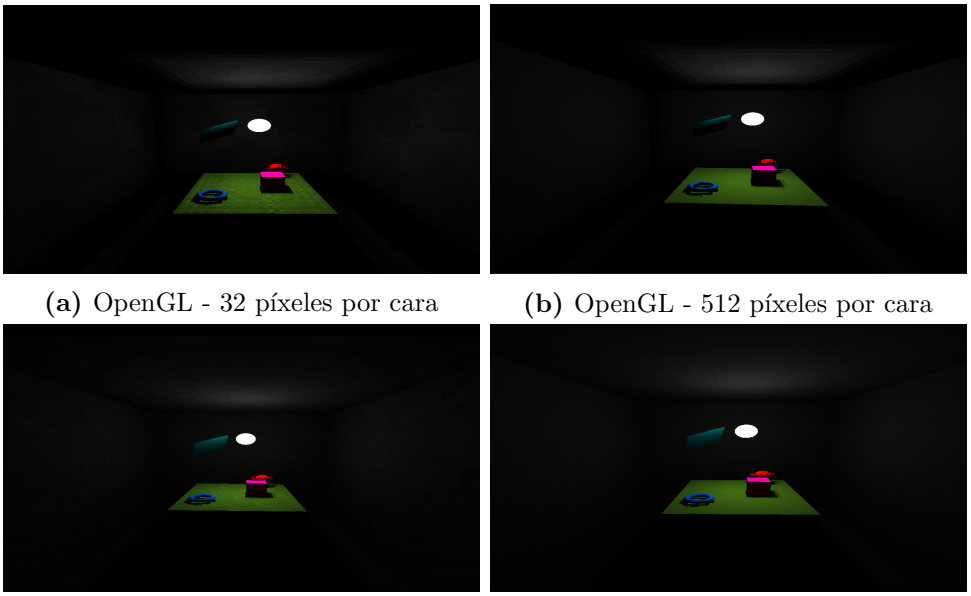


Figura 5.4: Comparación del rendimiento de los algoritmos en escenas exclusivamente difusas

tacar que se observó una ocupación promedio de la GPU del 30 % y CPU 90 %. Esto probablemente se deba a la gran cantidad de sincronizaciones necesarias entre el dispositivo y el controlador. El uso de traza de rayos presentó una ocupación de la CPU del 99 %. Se destaca la diferencia observada entre OpenGL y Embree en la Figura 5.4.

Una de las consecuencias más interesantes a ser analizadas para detectar la cantidad de muestras óptimas a considerar es la calidad de la imagen final, y qué tan pronunciadas son las diferencias en la iluminación entre los parches, es decir, en qué medida difiere la radiosidad entre parches. Para este caso, se pudo observar (véase la Figura 5.5) que si bien las resoluciones más bajas consumen menor cantidad de recursos los resultados tienen una calidad sustancialmente menor. Considerando que el modelo generalmente es utilizado para el cálculo de

iluminación en una etapa de pre-procesado (fuera de línea), es recomendable evitar el uso de factores de muestreo tan bajos. Esto se ve acentuado en el análisis de la matriz de factores de forma, según las métricas establecidas en 5.3.1 se pudo comprobar que máximo error promedio apreciado (medida que se ha denominado Ep) fue de $0,05$ y $0,04$ utilizando 786.432 muestras para los métodos del hemi-cubo y el hemisferio mientras que el uso de 3.072 muestras generó errores del entorno de los $0,14$ y $0,06$ respectivamente. Estos se ven aún más acentuados al realizar el cálculo de la radiosidad para cada parche.



(a) OpenGL - 32 píxeles por cara **(b)** OpenGL - 512 píxeles por cara

(c) Embree - 3072 rayos **(d)** Embree - 786432 rayos

Figura 5.5: Diferencias visuales ajustando la cantidad de muestras

Caso de prueba II (Reflexión Difusa + Especular)

En este caso, se observa en la tabla 5.4 que el mejor rendimiento se obtiene utilizando el método híbrido, esto se debe a los hilos que ejecutan los cálculos correspondientes al rebote especular (utilizando traza de rayos) son ejecutados en la CPU mientras la GPU procesa hemi-cubos. Esta observación se ve respaldada por el hecho de que, en promedio se observó una ocupación rondando en los entornos de 100 % de la CPU y 35 % GPU en el método híbrido, 75 % de la CPU y 30 % GPU en el método utilizando OpenGL y 99 % en la implementación que solo utiliza traza de rayos.

El paralelismo de las implementaciones basadas en la GPU garantiza el mejor rendimiento, sin embargo, se pudo notar que el uso exclusivo de traza

Muestras		**Tiempo de ejecución (s)**					
		Cornell Box			*Calle*		
OpenGL	Embree	GL-D+S	E-D+S	Híb	GL-D+S	E-D+S	Híb
32x32x3 - 32	**3072x3 - 32**	25	53	**21**	2563	1221	**943**
256x256x3 - 32	**196608 - 32**	93	134	**84**	**1054**	2512	1643
512x512x3 - 64	**786432 - 64**	304	486	**228**	-	5342	**4725**

Tabla 5.4: Resultados obtenidos en el segundo caso de prueba. Notación: GL, E, Híb corresponden a OpenGL, Embree e Híbrido. La notación 'D+S' indica que se utilizaron reflexiones difusas y especulares. Los valores de muestras a la derecha indican la resolución de los espejos. '-' indica casos de prueba que tomaron tiempos excesivos.

de rayos provee hasta 800 veces menor error máximo que los otros algoritmos implementados. En *Cornell Box*, con 786.432 muestras se notó una diferencia de error máxima Ep de $0,0080$ con el método híbrido y $0,0055$ utilizando el método de dibujado de portales, en comparación con el método de traza de rayos que logró un error de $0,1 \times 10^{-3}$.

Muestras		**Error relativo (normalizado)**		
		Cornell Box		
OpenGL	Embree	GL-D+S	E-D+S	Híb
32x32x3 - 32	**3072 - 32**	0.0138	0.000481	0.0091
256x256x3 - 32	**196608 - 32**	0.0093	0.000108	0.0073
512x512x3 - 64	**786432 - 64**	0.0080	0.000010	0.0055

Tabla 5.5: Resultados obtenidos en el segundo caso de prueba (Cornell Box).

Esto se debe a que tanto en el uso de dibujado de portales o el algoritmo híbrido se utilizan estimaciones de la dirección en la que rebotaría el rayo en el espejo considerado, lo que degrada la autenticidad final de los datos obtenidos, sobre todo en el dibujado de portales donde la granularidad de las muestras obtenidas es inferior (se toman muestras por área y no por rayo). Por lo tanto, incluso si el método de traza de rayos posee un tiempo de ejecución un tanto mayor (que se debe mayormente al hecho de que se ejecuta únicamente en la CPU) se observa una calidad de datos de órdenes de magnitud superior a la de los otros métodos.

Con el objetivo de cuantificar los datos de error observado se analizó el error promedio en el vector de radiosidad final (comparado a una muestra utilizando exclusivamente trazado de rayos con 786.432 muestras para el hemisferio) y se constató que, tal como se había supuesto, el método es significativamente

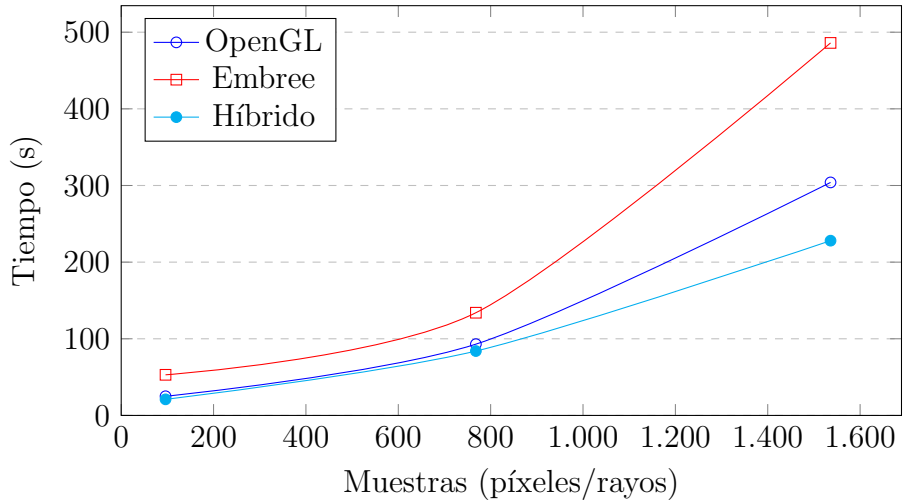


Figura 5.6: Rendimiento en segundos, considerando superficies especulares

menos propenso a generar errores como se ve en la Figura 5.7.

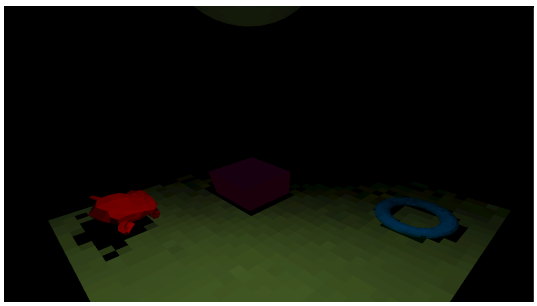
Caso de prueba III (Conjunta)

Esta prueba se construyó con el objetivo de comparar el rendimiento observado utilizando los algoritmos para el cálculo de factores de forma extendidos implementados. Es interesante comparar las diferencias en los tiempos de ejecución para una cantidad de muestras fijas, incluso si se han notado grandes discrepancias entre los resultados observados entre los métodos. Se ha puesto especial énfasis en las diferencias producidas al variar la cantidad de caras especulares de la escena, es decir, los parches cuyo coeficiente de reflexión es positivo. Las pruebas se realizaron con una cantidad de muestras fijas (256x256x3 para el hemi-cubo o 196.608 rayos para Embree y 4.096 muestras para el portal).

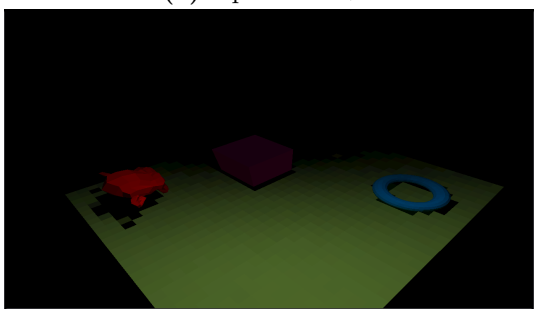
Tabla 5.6: Pruebas realizadas en *Cornell Box* para identificar incidencia en la cantidad de caras especulares utilizadas en el tiempo de ejecución

Parches especulares	OpenGL (s)	Embree (s)	Híbrido (s)
0	**38**	126	-
16	41	118	**39**
32	48	120	**40**
64	68	121	**43**

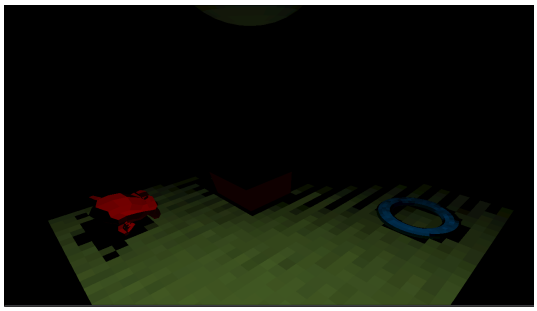
En la Tabla 5.6 puede observarse cómo la traza de rayos supera en dos



(a) OpenGL-D+S



(b) Embree-D+S



(c) Híbrido

Figura 5.7: Diferencias visualizadas utilizando las distintas implementaciones de cálculo de factores de forma extendido. 1536 muestras iniciales y 64 para rebotes especulares.

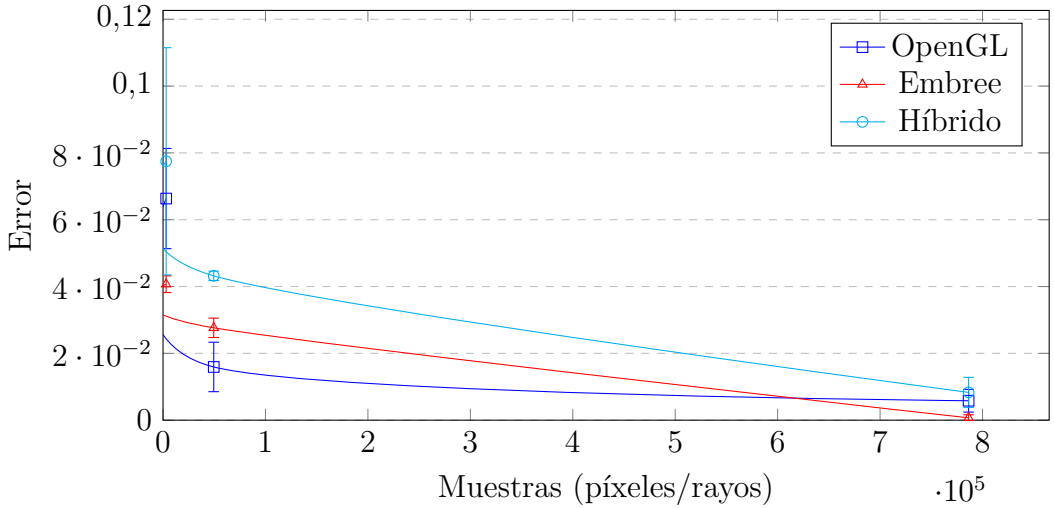
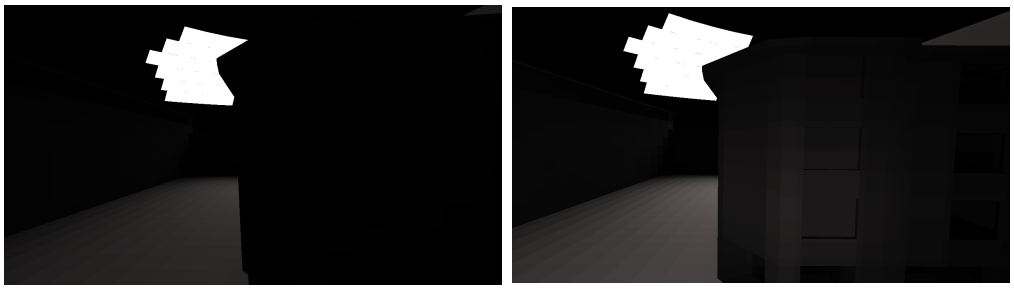


Figura 5.8: Error promedio observado en valor final de radiosidad en Híbrido. Los intervalos muestran el error máximo.

veces el tiempo a los otros algoritmos. Sin embargo, se ha de destacar (al igual que en las pruebas anteriores) que los resultados observados tendrían variaciones en caso de utilizarse resoluciones mayores para el dibujado de portales. En cuyo caso, los tiempos entre los casos de prueba serían más afines, aunque se conseguirían resultados peores dada las aproximaciones realizadas en el dibujado de portales. Esto se demuestra al emparejar la cantidad de muestras tomadas por el portal con las de cada cara del hemi-cubo (256); en este caso los tiempos de ejecución para 62 parches especulares es de 150 segundos. Por lo tanto, si bien existe un tiempo de ejecución mayor la estabilidad (en tiempo de ejecución) y la calidad de los datos obtenidos hacen que el método de traza de rayos sea superior a las otras dos propuestas.



(a) Extensión desactivada **(b)** Extensión activada

Figura 5.9: Diferencias observadas activando y desactivando extensiones

Adicionalmente, como se puede observar en la Figura 5.9, las diferencias

obtenidas en el valor final de la iluminación en cada parche pueden ser sutiles no obstante asemejan la simulación a escenarios reales con un costo despreciable en el tiempo de ejecución. En esta prueba en particular, se notó un aumento de aproximadamente 20 segundos (de 670 a 690). Donde en el primer caso se utilizó únicamente la iluminación difusa mientras que en segundo caso se activó la extensión utilizando trazado de rayos.

Caso de prueba IV (Stress)

Finalmente, con el objetivo de evaluar el impacto de contar con muchas superficies especulares se probó qué tanto tiempo adicional insume la carga impuesta al algoritmo del hemisferio (Embree) para considerar este tipo de superficie. Los resultados pueden observarse en la Figura 5.10. Se notó una diferencia máxima de 3 % del tiempo de cálculo de los factores de forma al variar la cantidad de superficies especulares entre 0 y 1500 parches. Es por ello que se puede concluir que al utilizar algoritmos de traza de rayos no se detecta un impacto significativo en el tiempo de ejecución.

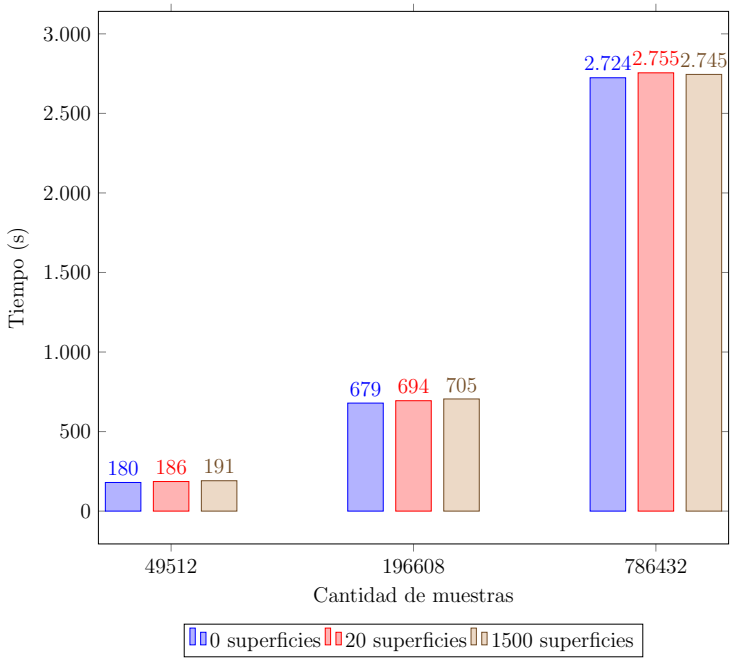


Figura 5.10: Variación en el tiempo de ejecución en función de la cantidad de muestras y superficies especulares

Capítulo 6

Conclusiones y trabajo futuro

En este capítulo se presentan las conclusiones principales en relación a la investigación realizada sobre el modelo de iluminación global implementado y sus extensiones para considerar superficies especulares, junto a un conjunto de posibles líneas de trabajo futuro para extender los algoritmos desarrollados.

6.1. Conclusiones

Este proyecto se dedicó al estudio, adaptación, implementación, extensión y comparación de distintos modelos de iluminación global basados en el método de radiosidad, así como posibles estrategias que optimicen el tiempo de ejecución observado. En este sentido, se logró comprobar que las extensiones del método de radiosidad que toman en consideración el fenómeno de la reflexión especular logran resultados con mayor realismo, observándose un costo adicional de computación despreciable, sobre todo al utilizar la técnica de traza de rayos.

Adicionalmente, se comprobó que el método del hemi-cubo presenta mayores dificultades para adaptarse al cálculo de los factores de forma extendidos. Recurrir a métodos auxiliares para computar las direcciones de rebote son aproximaciones que pueden degradar la calidad de los resultados, tanto en el uso de traza de rayos o dibujado de portales. No obstante, el método se muestra superior en escenas exclusivamente difusas aún cuando no se han implementado estructuras de aceleración para el cálculo.

En lo que refiere a la interfaz de usuario y las funcionalidades auxiliares implementadas se ha podido comprobar que fueron de utilidad para facilitar

la ejecución y diseño de las pruebas permitiendo la exportación e importación de los datos obtenidos de forma sencilla y genérica. El uso de formatos estandarizados fue sin lugar a dudas una elección que enriqueció la usabilidad del programa.

Sobre las distintas tecnologías utilizadas se destaca la facilidad de manejo de la geometría en Embree como a su vez la flexibilidad con la que se maneja el paralelismo mediante el uso de hilos. Además, se destacan las distintas extensiones de OpenGL que hicieron posible el uso de *compute shaders* para reducir los hemicubos proyectados a filas de la matriz de factores de forma y la simpleza del uso de *geometry shaders* y arreglos de texturas para reducir al mínimo los costos de las mutaciones del estado en lo que respecta de *render targets*.

Finalmente, se destaca el uso de la metodología *Kanban* para el control y seguimiento de requerimientos y tareas, junto al uso de programación orientada a objetos para abstraer los conceptos y ofrecer una capacidad mayor de reutilización de componentes y extensión de los métodos. Un ejemplo de esto fue la clase *Pipeline* concebida exclusivamente para superficies difusas utilizando OpenGL, que fue luego extendida de forma tal que se incluyeran métodos de traza de rayos con Embree y extensiones de los algoritmos de cálculo de factores de forma.

6.2. Trabajo futuro

Este proyecto puede ser continuado en distintas líneas de trabajo que fueron surgiendo a lo largo de la implementación y estudio de las soluciones desarrolladas.

En primer lugar, sería de gran interés la adición de estructuras de aceleración en OpenGL como por ejemplo, el uso de *octrees*. Estas estructuras arborescentes permiten el descarte de dibujado de ciertas agrupaciones de primitivas a nivel de CPU y por tanto eliminan el costo del cálculo del algoritmo del Z-Buffer en gran medida [Vazquez y Guarte [17]]. Además, sería beneficioso investigar la posible implementación del *renderer* de OpenGL en la API Vulkan, que ha sido optimizada para minimizar el impacto del controlador de la GPU en la CPU, y que además provee facilidades que permiten enviar comandos de dibujado desde distintos hilos, disminuyendo la necesidad de sincronización entre el controlador de pre-procesamiento (*PreprocessorController*)

y el controlador del dispositivo.

Por otro lado, sería beneficioso implementar un *plugin* de los algoritmos implementados para *software* de terceros, como por ejemplo el programa *Blender*, que se ha utilizado para editar los objetos 3D en las escenas de prueba y que ya provee de funcionalidades que calculan la iluminación global [18].

Además, como se aprecia en la Figura 6.1 el uso de texturas proporciona detalles visuales que enriquecen la imagen generada. Sin embargo, no es necesario modificar el algoritmo de radiosidad desarrollado, simplemente se adicionan las texturas en la última etapa del pipeline. Es decir, es posible integrar el método de radiosidad como una de las etapas necesarias para generar imágenes fotorealistas.

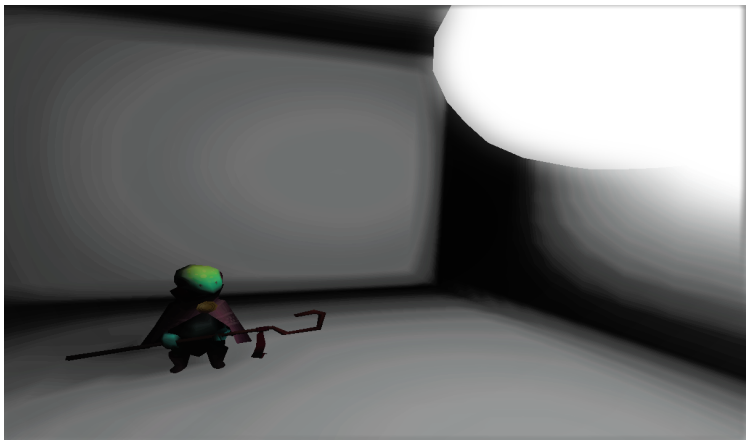


Figura 6.1: Ejemplificación de la adición de texturas a los objetos de la escena

En tercer lugar, el desarrollo de nuevo *hardware* permite que la traza de rayos pueda ser acelerada por la GPU a través de las extensiones DirectX DXR y Vulkan VK_NVX_raytracing, que son predominantemente utilizadas con tarjetas gráficas NVIDIA RTX. Por ello se propone el análisis de los algoritmos implementados en este tipo de dispositivos.

Finalmente, se propone considerar las posibles oclusiones parciales en el método híbrido, de forma de aumentar su precisión.

Lista de figuras

Lista de tablas

Referencias bibliográficas

[1] Light path expressions, 2019. URL `https://rmanwiki.pixar.com/display/REN22/Light+Path+Expressions`.

[2] James T Kajiya. The rendering equation. In ACM SIGGRAPH computer graphics, volume 20, pages 143–150. ACM, 1986.

[3] Michael F Cohen and Donald P Greenberg. The hemi-cube: A radiosity solution for complex environments. In ACM SIGGRAPH Computer Graphics, volume 19, pages 31–40. ACM, 1985.

[4] Eric P. Lafortune and Yves D. Willems. Bi-directional path tracing. In Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93), pages 145–153, Alvor, Portugal, December 1993.

[5] Henrik Wann Jensen. Realistic image synthesis using photon mapping. AK Peters/CRC Press, 2001.

[6] Cindy M Goral, Donald P Torrance, Kenneth E'; Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. In ACM SIGGRAPH computer graphics, volume 18, pages 213–222. ACM, 1984.

[7] TJ Malley. A shading method for computer generated images. Master's thesis, Dept. of Computer Science, University of Utah, 1988.

[8] Benoit Beckers and Pierre Beckers. Fast and accurate view factor generation. In FICUP, An International Conference on Urban Physics, volume 9, 2016.

[9] Francois Sillion and Claude Puech. A general two-pass method integrating specular and diffuse reflection. In ACM SIGGRAPH Computer Graphics, volume 23, pages 335–344. ACM, 1989.

[10] Peter Shirley. A ray tracing method for illumination calculation in di use-specular scenes. In Proceedings of Graphics Interface, volume 90, pages 205–212, 1990.

[11] Turner Whitted. An improved illumination model for shaded display. In ACM Siggraph 2005 Courses, page 4. ACM, 2005.

[12] Arjan JF Kok, Celal Yilmaz, and Laurens HJ Bierens. A two-pass radiosity method for bézier patches. In Photorealism in Computer Graphics, pages 115–124. Springer, 1992.

[13] Holly E Rushmeier and Kenneth E Torrance. Extending the radiosity method to include specularly reflecting and translucent materials. ACM Transactions on Graphics (TOG), 9(1):1–27, 1990.

[14] Beyond porting: How modern opengl can radically reduce driver overhead (steam dev days 2014), 2019. URL `https://www.youtube.com/watch?v=-bCeNzgiJ8I&list=PLckFgM6dUP2hc4iy-IdKFtqR9TeZWMPj`.

[15] Henri Gouraud. Continuous shading of curved surfaces. IEEE transactions on computers, 100(6):623–629, 1971.

[16] Jairo Acuña Paz y Miño, Vincent Lefort, Claire Lawrence, and Benoit Beckers. Maquette numérique d'une rue du vieux bayonne pour son étude thermique par éléments finis. A la pointe du BIM: Ingénierie et architecture, enseignement et recherche, page 103, 2018.

[17] Joel Vazquez and Pablo Guartes. Aceleración del cálculo de la matriz de factores de forma utilizando visibilidad jerarquica. Proyecto de grado de Ingeniería en Computación. 2017.

[18] Cycles - global illumination engine, 2019. URL `https://docs.blender.org/manual/en/latest/render/cycles/index.html`.