



UNIVERSIDAD
DE LA REPUBLICA
URUGUAY



FACULTAD DE
INGENIERIA

Tesis de grado

Iluminación global con superficies especulares

Bruno Sena

Ingeniería en Computación
Facultad de Ingeniería
Universidad de la República

Montevideo – Uruguay
Agosto de 2019



UNIVERSIDAD
DE LA REPUBLICA
URUGUAY



FACULTAD DE
INGENIERIA

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 Aguerre

Eduardo Fernández

Montevideo – Uruguay
Agosto de 2019

Sena, Bruno

Tesis de grado / Bruno Sena. - Montevideo: Universidad de la República, Facultad de Ingeniería, 2019.

VII, 58 p.: il.; 29, 7cm.

Directores:

José Aguerre

Eduardo Fernández

Tesis de Grado – Universidad de la República, Ingeniería en Computación, 2019.

Referencias bibliográficas: p. 55 – 58.

1. iluminación global, 2. radiosidad, 3. reflexión especular. I. Aguerre, José, Fernández, Eduardo, . II. Universidad de la República, Ingeniería en Computación. III. Título.

INTEGRANTES DEL TRIBUNAL DE DEFENSA DE TESIS

NombreTribunal1 ApellidoTribunal1

NombreTribunal2 ApellidoTribunal2

NombreTribunal3 ApellidoTribunal3

Montevideo – Uruguay

Agosto de 2019

RESUMEN

Aquí va el abstract

Palabras claves:

iluminación global, radiosidad, reflexión especular.

Tabla de contenidos

1	Introducción	1
1.1	Motivación y problema	1
1.2	Objetivos	1
1.3	Resultados esperados	1
1.4	Estructura del documento	1
2	Estado del arte	2
2.1	Modelos de iluminación	2
2.1.1	Iluminación Local	3
2.1.2	Iluminación Global	4
2.2	Radiosidad	4
2.2.1	Radiosidad en superficies lambertianas	5
2.3	Métodos de cálculo de la matriz de Factores de Forma	8
2.3.1	Rasterización	9
2.3.2	Trazado de rayos	14
2.4	Superficies especulares	16
2.5	Cálculo del vector de radiosidades	17
2.6	OpenGL	18
2.6.1	Arquitectura	18
2.6.2	Extensiones	19
2.7	Embree	20
2.8	Trabajos relacionados	21
2.8.1	Un método de trazado de rayos para el cálculo de iluminación en escenas difusas-especulares	21
2.8.2	Iluminación specular rápida incluyendo efectos especulares	22

2.8.3 Un método de dos pasadas para el cálculo de radiosidad en parches de Bezier	22
3 Solución propuesta	24
3.1 Alcance y objetivos	24
3.2 Proceso de desarrollo	24
3.3 Diseño	25
3.3.1 Motor de renderizado	26
3.3.2 Interfaz gráfica	28
4 Implementación	30
4.1 Cálculo de factores de forma de la componente difusa	30
4.1.1 Algoritmo del hemi-cubo	30
4.1.2 El algoritmo de la hemi-esfera	33
4.2 Cálculo de factores de forma de la componente especular	35
4.2.1 Extensión del método del hemi-cubo	35
4.2.2 Extensión del método de la hemi-esfera	38
4.3 Cálculo del vector de radiosidad	39
4.4 Visualización de resultados y resultados intermedios	42
4.5 Interfaz de usuario	43
5 Experimental	45
5.1 Ambiente de prueba	45
5.2 Escenas	45
5.3 Casos de prueba	47
5.3.1 Métricas consideradas	47
5.3.2 Descripción de casos de prueba	48
5.3.3 Resultados observados	49
6 Conclusiones y trabajo futuro	51
6.1 Conclusiones	51
6.2 Trabajo futuro	51
Lista de figuras	52
Lista de tablas	54

Apéndices	55
Referencias bibliográficas	57

Capítulo 1

Introducción

- 1.1. Motivación y problema
- 1.2. Objetivos
- 1.3. Resultados esperados
- 1.4. Estructura del documento

Capítulo 2

Estado del arte

Este capítulo introduce un resumen de las áreas más importantes relacionadas al trabajo realizado en este proyecto, estas incluyen 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.

Trivialmente, esto puede ser reducido al problema de cálculo del valor de intensidad lumínica observada en un punto x y proveniente de otro punto x' . En 1986, Kajiya presentó uno de los modelos más aceptado por la comunidad por su generalidad, comúnmente denominado «la ecuación del *rendering*»:

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

donde:

- $I(x, x')$ describe energía de radiación lumínica observada en el punto x proveniente de x''
- $g(x, x')$ es un término geométrico, toma el valor de 0 si existe oclusión entre x' y x en otro caso su valor es $\frac{1}{r^2}$ donde r es la distancia entre x'

y x

- $\epsilon(x, x')$ mide la energía emitida por la superficie en el punto x' 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 dispersión de la luz que llega desde x'' a x desde el punto x'
 - $I(x', x'')$ describe energía de radiación lumínica observada en el punto x' proveniente de x''

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

Existen distintos métodos de resolución de la «ecuación del *rendering*», la mayoría implican aproximaciones dado el gran costo de cálculo requerido para computar el valor exacto de $I(x, x')$. Estos métodos balancean el costo computacional de los algoritmos utilizados y la fidelidad con el valor final de la función. Dependiendo de las decisiones y simplificaciones consideradas existen dos clasificaciones posibles para el modelo: *local* y *global* 2.1.

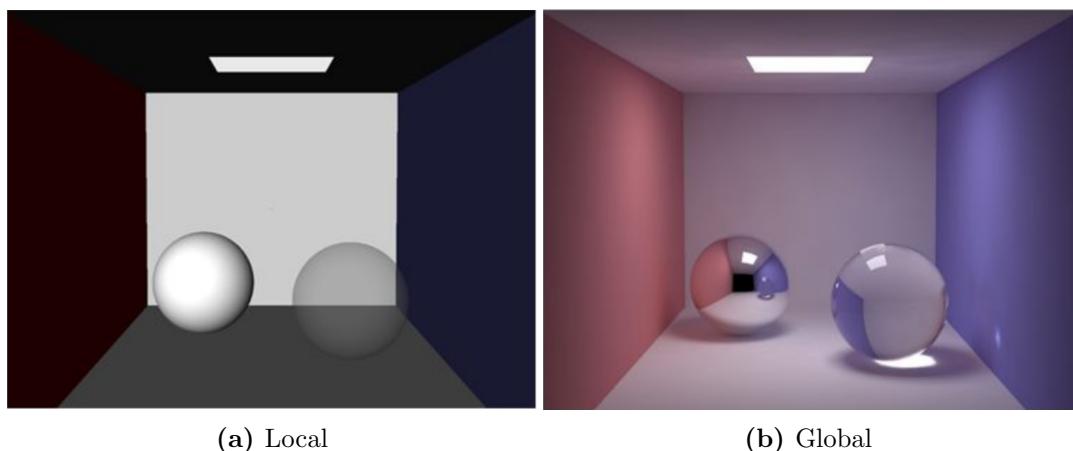


Figura 2.1: Dibujado utilizando distintos modelos de iluminación

2.1.1. Iluminación Local

Los modelos de iluminación local como el propuesto por Phong en 1975 tienen en cuenta las propiedades físicas de los materiales y las superficies de forma individual. Es decir, al dibujar uno de los objetos no se toman en cuenta

las posibles interacciones de los haces de luz con los objetos restantes en la escena. Estos métodos 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, $\epsilon(x, x')$ toma un valor constante únicamente dependiente de x y $\int_S \rho(x, x', x'') I(x', x'') \delta x''$ toma el valor constante 1.

2.1.2. Iluminación Global

El término iluminación global refiere a una modelo de computación gráfica en donde se simulan parcialmente 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 más fieles a la realidad en distintos sentidos.

El algoritmo de *trazado de caminos de rayos* 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) impacta directamente en los errores y calidad en la imagen final.

Por otro lado, el algoritmo de *mapeado de fotones* simula los efectos producidos por las colisiones de las partículas que componen la luz (fotones) con los objetos, que dejan *impresiones* que afectarán el resultado final de la imagen.

Existen además distintas variaciones e híbridos de estos métodos que normalmente dibujan imágenes *fuera de línea* pues son demasiado costosos como para dibujar imágenes en tiempo real.

2.2. Radiosidad

El método de radiosidad es una técnica de iluminación global 2.1.2 que emula el transporte de la luz entre superficies que se rige por la magnitud física definida, a su vez, como radiosidad, que indica el flujo de energía irradiada por unidad de área ($\frac{W}{m^2}$).

Originalmente, este modelo de iluminación global fue propuesto por Goral

et al. en 1984, se basa en modelos matemáticos similares a los que resuelven el problema de la transferencia de calor en sistemas cerrados (MEF).

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.

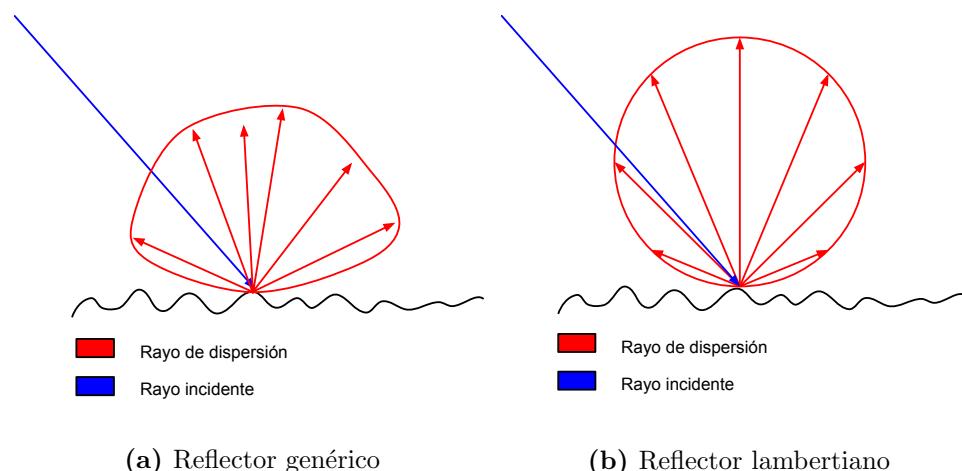


Figura 2.2: Comparación entre tipos de reflectores

Adicionalmente, se considerará que cada superficie irradia energía lumínica en todas direcciones en un diferencial de área δ_A , para una dirección de vista ω puede ser definida como:

$$i = \frac{\delta P}{\cos \phi \delta \omega} \quad (2.2)$$

donde:

- i es la intensidad de la radiación para un punto de vista particular
- δP es la energía de la radiación que hermano la superficie en al dirección ϕ con ángulo sólido $\delta \omega$

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 hemi-esfera centrada en el punto estudiado:

$$P = \int_{2\pi} \delta P = \int_{2\pi} i \cos\phi \delta\omega = i \int_{2\pi} \cos\phi \delta\omega = i\pi \quad (2.3)$$

Por tanto, dada una superficie S_i , es posible calcular la energía lumínica que deja la superficie utilizando (2.3).

Existen tres métodos posibles de cálculo de la ecuación de radiosidad: la integración basada en elementos finitos, la integración basada en reglas de cuadratura y la integración basada en métodos de Monte Carlo.

En este caso, los autores decidieron utilizar un método de elementos finitos. Estos métodos consisten en la subdivisión del espacio en partes discretas más simples conocidas como **elementos finitos** aproximando el valor real de la función a través de un sistema de ecuaciones. Por tanto, para poder aplicarlo resta definir la *cerradura* de una superficie, definiremos la cerradura de una superficie como los límites que definen los puntos internos y externos de esta, llamaremos **parche** a cada una de estas superficies cerradas. Esta reformulación del problema, es fácilmente trasladable a la ecuación (2.1).

$$B_j = E_j + \rho_j \sum_{i=1} N B_i F_{ij} \quad (2.4)$$

donde:

- B_j es la intensidad lumínica (radiosidad) que deja la superficie j .
- E_j es la intensidad lumínica directamente emitida por j .
- ρ_j es la reflectividad del material para la superficie j .
- F_{ij} se denomina *factor de forma*, un término que representa la fracción de energía lumínica que deja la superficie i y llega a j .

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

E, ρ dependen de los materiales que compongan la escena, son parámetros dados. Sin embargo, resta computar la matriz de factores de forma \mathbf{F} para finalmente obtener 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 , δA_j , representados en la figura 2.8, el ángulo sólido visto por δA_i es $\delta\omega = \frac{\cos \phi_j \delta A_j}{r^2}$. Sustituyendo en (2.3) se obtiene:

$$\delta P_i \delta A_i = i_i \cos \phi_i \delta\omega \delta A_i = \frac{P_i \cos \phi_i \cos \phi_j \delta A_i \delta A_j}{\pi r^2} \quad (2.5)$$

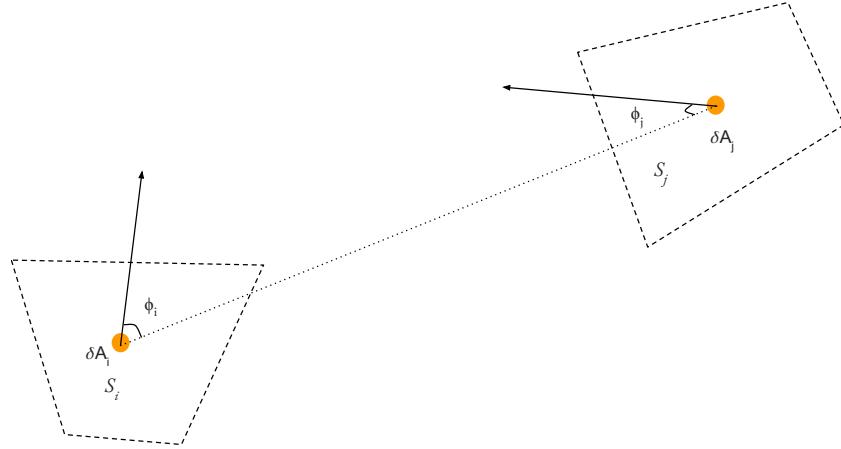


Figura 2.3: El factor de forma entre dos superficies

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

$$F_{\delta A_i - \delta A_j} = \frac{\frac{P_i \cos \phi_i \cos \phi_j \delta A_i \delta A_j}{\pi r^2}}{P_i \delta A_i} = \frac{\cos \phi_i \cos \phi_j \delta A_i}{\pi r^2} \quad (2.6)$$

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

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

De (2.7) se obtienen las siguientes propiedades:

1. $A_i F_{ij} = A_j F_{ji}$
2. $\sum_{j=1}^N F_{ij} = 1$

3. $F_{ii} = 0$
4. F_{ij} toma el valor correspondiente a la proyección de j en una hemiesfera unitaria centrada en i , proyectándola a su vez en un disco unitario.

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

El cálculo de los factores de forma a través de la ecuación (2.7) analíticamente es inviable en la práctica pues supone la necesidad de calcular 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, puese 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 una hemi-esfera centrada en S_i y luego en un disco centrado en S_i .

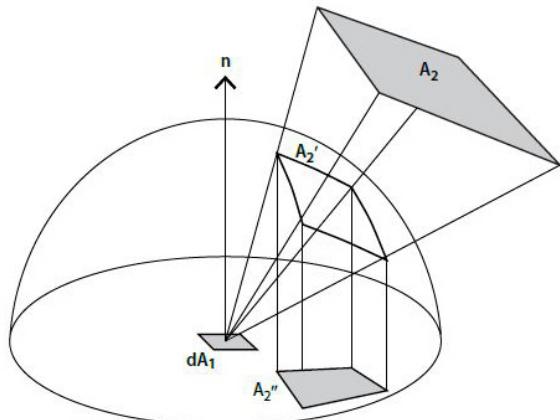


Figura 2.4: La analogía de Nusslet

El cálculo de la matriz de factores de forma \mathbf{F} supone la proyección de los parches, de aquí en más se asumirá que estos parches son poligonales y por tanto es posible utilizar las técnicas de dibujado de objetos tridimensionales tradicionales.

2.3.1. 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 la transformación los parámetros de entrada en parámetros que recibirá la siguiente etapa, que los recibirá y transformará como corresponda.

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

1. Procesamiento de primitivas geométricas:

- 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.
- Vertex shader*: 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.
- Teselado: En esta etapa se procesan los vértices a nivel de primitiva geométrica, con el objetivo de subdividirlas para mejorar la resolución obtenida.
- Geometry shader*: En esta etapa también se procesan los vértices a nivel de primitiva geométrica con el objetivo de mutarlas y replicarlas.
- Recortado: Esta etapa es *fija*, es decir, no es programable. Todas las primitivas calculadas anteriormente que residan fuera del volu-

men 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 plantilla 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.

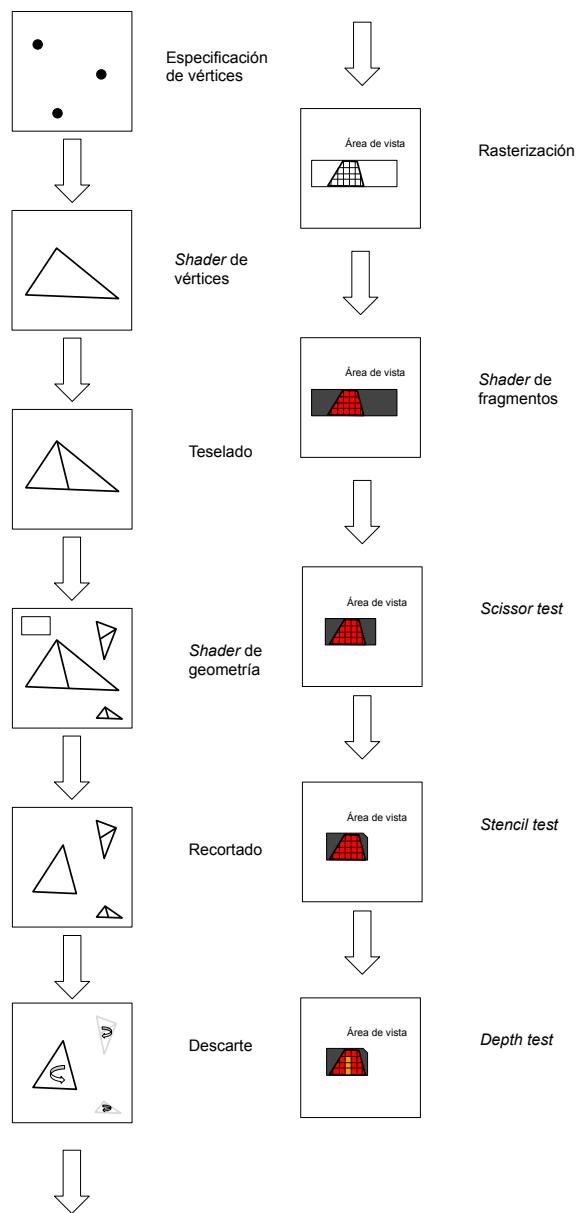


Figura 2.5: El *rendering pipeline* de OpenGL

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 en sus siglas en inglés). Con el objetivo de aprovechar este hardware Cohen and Greenberg idearon el método del hemi-cubo para el cálculo de factores de forma.

El método del hemi-cubo

El hardware optimizado para realizar operaciones de rasterización tiene la capacidad de proyectar escenas tridimensionales en imágenes bidimensionales a gran velocidad.

El método original de cálculo de factores de forma propone la proyección de la escena una hemiesfera centrada en S_i , sin embargo los modelos de proyección utilizados no lo permiten. Por esto es necesario proyectar la escena a un hemi-cubo centrado en S_i , esto supone el dibujado de cinco superficies bidimensionales, y por tanto puede ser realizada utilizando la rasterización.

Para utilizar el hardware eficientemente consideraremos que se calculará una fila completa de \mathbf{F} , esto implica que dada S_i , una superficie, calcularemos simultáneamente los factores de forma para 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, propuesto originalmente por Cohen and Greenberg en 1985, 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, que dependerá de la posición del píxel en el hemi-cubo en relación a la hemiesfera que este aproxima. Esta suma genera una fila de la matriz \mathbf{F} , específicamente la fila \mathbf{F}_i .

Por tanto, podremos definir:

$$\mathbf{F}_{ij} = \sum_{q=1}^R \delta F_q \quad (2.8)$$

donde:

- R es la cantidad de píxeles correspondientes a la superficie S_j que cubren el hemi-cubo.
- δF_q el diferencial de factor de forma asociado al píxel del hemi-cubo q .

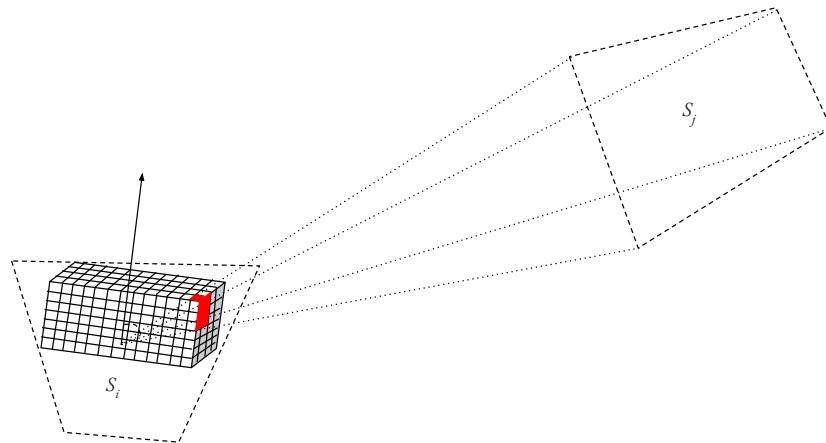


Figura 2.6: Representación gráfica del método del hemicubo

Los diferenciales de factores de forma deben corregir la deformación introducida con el cambio de proyección desde una hemiesfera 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 la hemiesfera unitaria.

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

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

Para las caras laterales, la fórmula dada es:

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

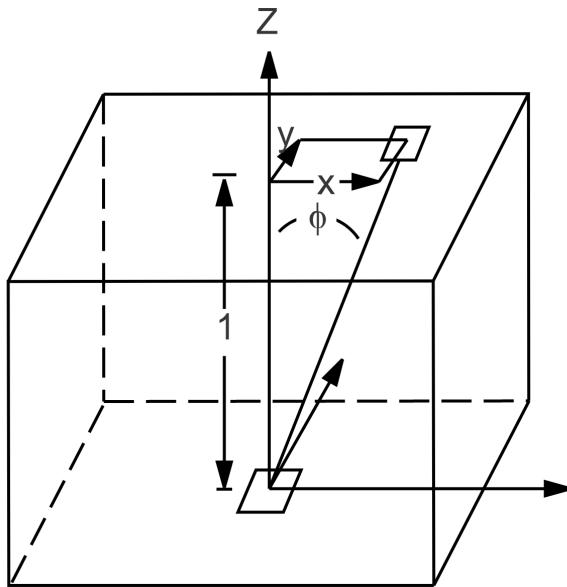


Figura 2.7: Representación gráfica de los ejes considerados para el factor de corrección de los factores de forma

2.3.2. Trazado de rayos

Otra de las técnicas de emulación de iluminación existente es el trazado de rayos, consiste en la computación de los puntos de intersección de una semirecta (a la que denominaremos rayo) con la geometría de la escena, cada uno de estos rayos emulará el transporte de los haces de luz emitidos.

Para cada uno de los rayos emitidos, se determinará el punto de intersección más cercano, luego es posible establecer qué primitiva geométrica fue intercepcionada es posible integrar el resultado intermedio producido por la intersección al resultado final, dependiendo del modelo de iluminación utilizado.

El trazado de rayos es una técnica efectiva [1] para computar 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 interacciona. En este algoritmo, cada uno de los rayos emitidos computa uno de los integrandos de la ecuación.

El método de la hemi-esfera

En el caso de la radiosidad, es posible utilizar esta técnica para calcular los factores de forma, es decir, para resolver la ecuación (2.7).

Recordando, un factor de forma F_{ij} representa la energía que llega a la

superficie S_i desde S_j , es posible re-imaginar el problema original colocando una hemi-esfera unitaria en el centro de S_i orientada en la dirección de la normal de la superficie.

El algoritmo propuesto por Malley 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 $f(x) = \frac{1}{2}[1 + \cos((x - 1)\pi)]$.

$$\mathbf{F}_{ij} = \sum_{k=1}^{nMuestras} \frac{\beta(ray(S_i, d), S_j)}{nMuestras} \text{ donde } d \text{ tiene una distribución apropiada.} \quad (2.11)$$

donde:

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

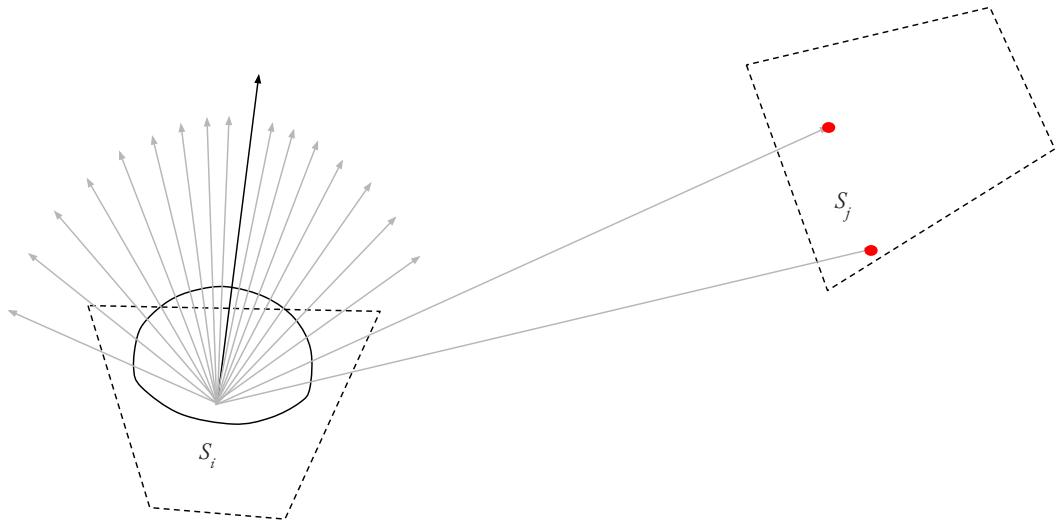


Figura 2.8: Representación gráfica del método de trazado de rayos

Sin embargo, no es necesario utilizar una distribución de probabilidad con valores aleatorios o pseudo-aleatorios, sino que si la cantidad de rayos utilizados es la suficiente y además se utiliza una función que distribuya correctamente cada rayo, entonces los resultados obtenidos serán similares.

Por esto, pueden utilizarse otras distribuciones para la dirección de traza. Particularmente, una de ellas es la propuesta por Beckers and Beckers, los autores presentaron un método general de teselación de discos y hemiesferas. En particular, la propuesta para hemiesferas genera un conjunto de celdas

de igual área con cuatro lados que comparten la misma relación de aspecto. Esto hace que el método presente una calidad adecuada para la elección de las direcciones en la que se trazarán los rayos.

2.4. 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, es necesario simular reflexiones especulares correctamente para obtener resultados que se asemejen a la realidad.

Por ello existe la extensión del método para superficies especulares o refractantes propuesto por Sillion and Puech en 1989. Los autores proponen extender el significado del término *factor de forma* a más que una mera relación geométrica entre parches. Sino que un factor de forma F_{ij} será la proporción de energía que deja la superficie i y llega la superficie j luego de un número de reflexiones y refracciones.

Esto modifica completamente los algoritmos de cálculo de factores de forma, los autores proponen un algoritmo de que cálculo consiste en el trazado de rayos desde S_i en una dirección arbitraria d bien distribuida. Luego, una vez que se conozca el **camino** trazado se distribuirá el valor final del factor de forma dependiendo en la cantidad de superficies con las que interaccione el rayo y sus coeficientes especulares como se observa en la figura 2.9.

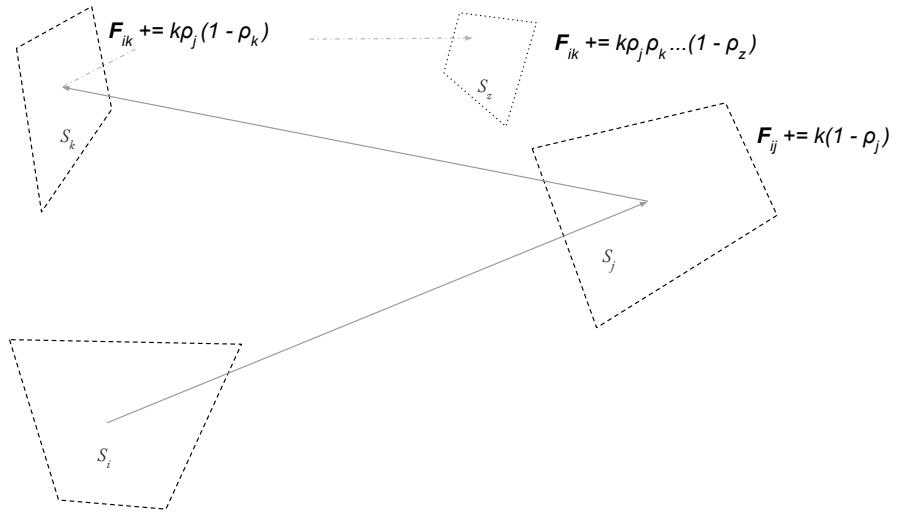


Figura 2.9: Representación gráfica del cálculo del factor de forma extendido donde k corresponde al inverso de la cantidad de muestras tomadas.

2.5. Cálculo del vector de radiosidades

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

Recordando (2.4), es posible deducir el problema al sistema de ecuaciones dado por:

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

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 2.2.1, podemos observar que:

- $\sum_{j=1}^N \mathbf{F}_{ij} \leq 1 \forall i \in [1, N]$
- $\rho_i \leq 1 \rightarrow \sum_{j=1}^N \mathbf{R}_{ij} \leq 1 \forall i \in [1, N]$

Esto implica que las entradas de \mathbf{RF} son siempre menores a 1, por tanto, $(\mathbf{I} - \mathbf{RF}) = M$ es diagonal dominante ya que $\sum_{j=1}^N |R_{ij}F_{ij}| \leq 1 \forall i \in [1, N]$ y $R_{ii}F_{ii} = 0 \forall i \in [1, N]$. Esto 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 menor. Para ello, considerando que la matriz \mathbf{F} es diagonal dominante, podemos utilizar la ecuación 2.13 pues el *residuo* (el término agregado en cada iteración) se reduce de la siguiente forma: $\|\mathbf{R}\mathbf{F}B^{(i+1)}\| < \|\mathbf{R}\mathbf{F}B^{(i)}\|$.

$$B^{(i+1)} = \mathbf{R}\mathbf{F}B^{(i)} + E \text{ con } B^{(0)} = E \quad (2.13)$$

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 del coeficiente de reflexión especular y 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 .

2.6. 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 2.3.1. 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.

2.6.1. Arquitectura

La arquitectura base de la biblioteca es de cliente/servidor 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, que 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 imágenes bidimensionales). 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.

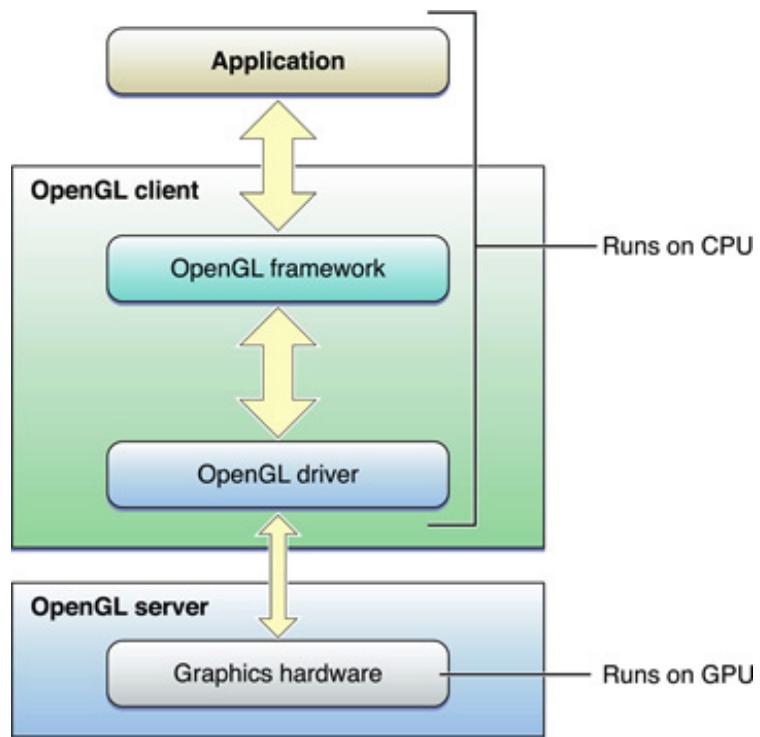


Figura 2.10: Vista general de la arquitectura de OpenGL

2.6.2. 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 el hardware utilizado. Por este

motivo, existen bibliotecas que manejan la creación del contexto en diversas plataformas como SDL y GLFW.

2.7. Embree

Embree es una biblioteca que 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) y la generación de estructuras de aceleración, como las BVH (del inglés *Bounding Volume Hierarchies*).

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:** Para la visualización de escenas con gran cantidad de primitivas.

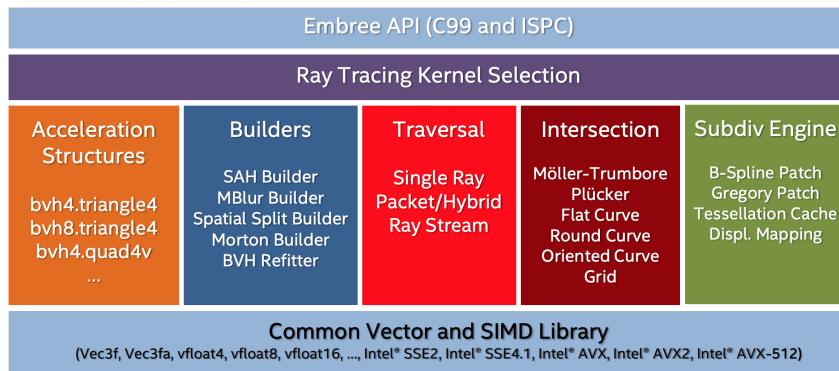


Figura 2.11: Vista general de la arquitectura de Embree

2.8. Trabajos relacionados

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

2.8.1. Un método de trazado de rayos para el cálculo de iluminación es escenas difusas-especulares

El algoritmo propuesto por Shirley en 1990 calcula la iluminación difusa utilizando dos pasadas. Este algoritmo difiere del propuesto por Sillion and Puech 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 utilizando el método de Arvo. Este método considera calcular 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 [9]) sustituyendo el término de ambiente por el calculado en las pasadas anteriores.

2.8.2. Iluminación specular rápida incluyendo efectos especulares

Otro acercamiento a la emulación de efectos de reflexiones especulares implica el uso de sistemas de partículas o fotones para seguir el camino de la luz cuando interacciona con este tipo de superficies. El algoritmo propuesto por Granier et al. implica la construcción de un grafo jerárquico representando la geometría de la escena, cada nodo del árbol representa un subespacio donde se calculará la radiosidad, el refinamiento (cantidad de niveles) incide directamente en el error observado.

Al observar la transferencia de luz entre los nodos que componen la escena, las transferencias difusas-especulares son computadas a través de trazado de fotones entre el subconjunto de objetos seleccionado. Esto significa que las partículas dejarán trazas de energía dependiendo del tipo de superficie donde incidan, además rebotarán en función de distribución asignada a la superficie.

2.8.3. Un método de dos pasadas para el cálculo de radiosidad en parches de Bezier

El método propuesto por Kok et al. 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 de la hemi-esfera, agregando los resultados para cada punto. En caso de que un rayo interseque una superficie especular, los autores proponen un método similar al de Sillion and Puech 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.

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. Comparando el rendimiento de los distintos métodos en 2, en escenas compuestas por triángulos y cuadriláteros.

Los métodos involucrados incluyen:

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 dibujado de portales
4. Cálculo de factores de forma extendidos utilizando trazado de rayos

Además, será necesario 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). Se soportará la carga de datos geométricos y materiales utilizando el formato estándar `Waveform`.

3.2. Proceso de desarrollo

Dada la naturaleza del proyecto, fue deseable establecer una metodología de desarrollo para facilitar el proceso de seguimiento del progreso incluso cuando el equipo de desarrollo fue individual.

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.
- 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 respository del proyecto, con tareas como las vistas en 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.

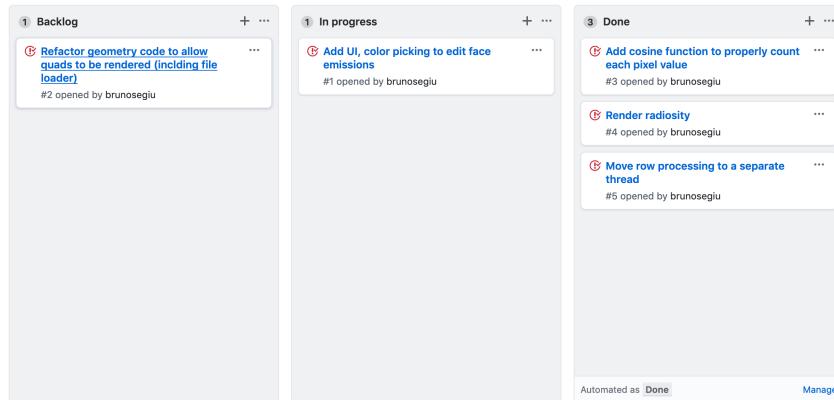


Figura 3.1: Tabla de Kanban utilizada en el proyecto

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 contruidos.

El diseño de la solución comprende dos componentes principales, la interfaz gráfica de usuario (GUI, 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** 3.2 cargará distintos objetos desde el disco duro que serán manejados como una instancia de **Mesh**.

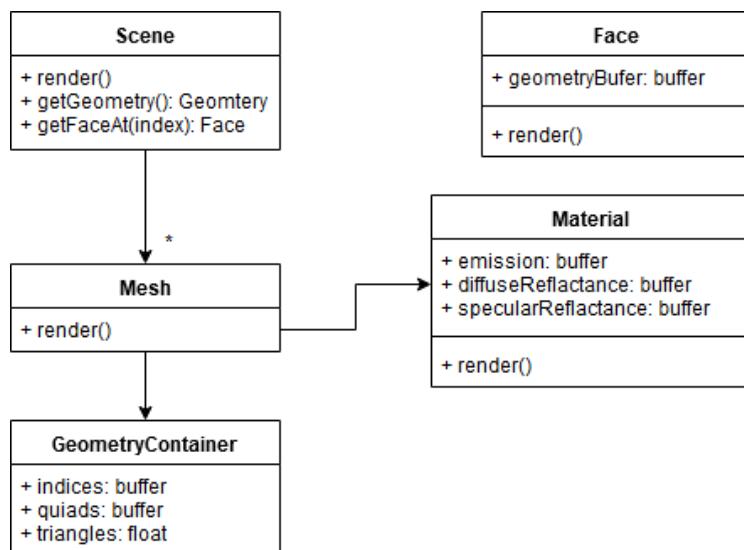


Figura 3.2: Módulo de manejo de geometría

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 ejecu-

ción de los comandos de los distintos *pipelines* implementados que resuelven el cálculo de la radiosidad.

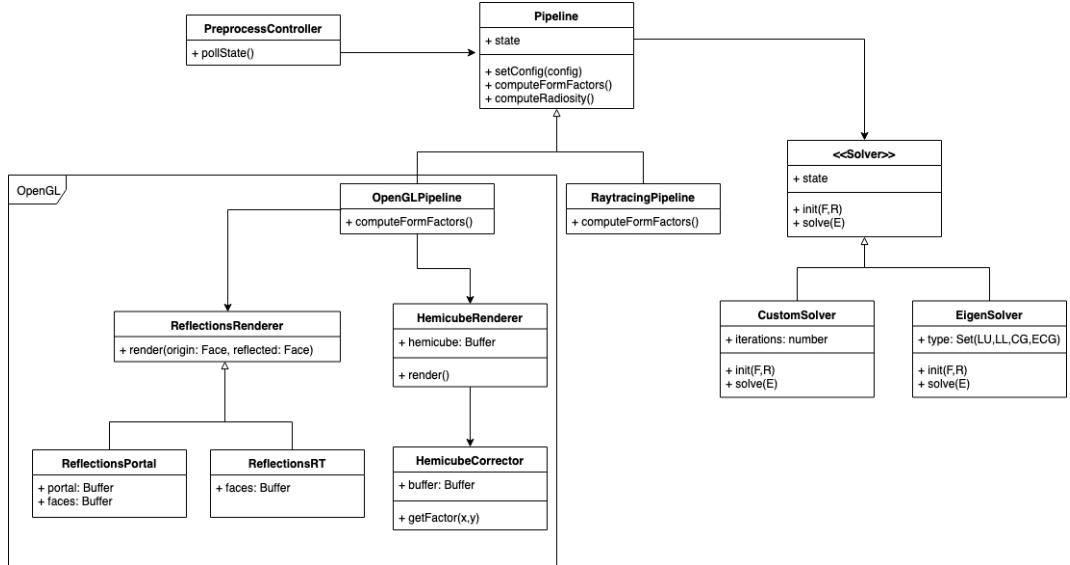


Figura 3.3: Arquitectura del módulo de pre-procesado

Un pipeline es definido a partir de un conjunto de funciones ejecutadas en el siguiente orden:

- 1.
2. `setConfig(scene, interpolator, reflections, n_channels, solver)`
3. `computeFormFactors()`
4. `computeRadiosity()`

Donde `computeFormFactors()` variará dependiendo del método de cálculo elegido (veáse 3.3), donde puede utilizarse el método del hemi-cubo o el de la hemi-esfera. 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 *trazado de 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.

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 como valor de emisión

inicial, valor de reflexión especular, visualización de geometría para facilitar la edición de las propiedades de los objetos o sus caras. El proceso de dibujado comienza con el dibujante de la escena que dibujará un conjunto de imágenes correspondiente a las propiedades de los materiales **SceneRenderer** en la figura 3.4, luego un dibujante de texturas **TextureRenderer** seleccionará y convertirá correctamente el resultado anterior a valores tres canales (RGB). El módulo de visualización siempre tendrá una textura bidimensional como parámetro de salida.

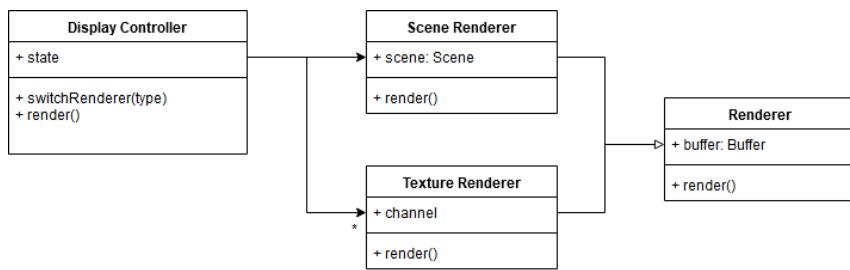


Figura 3.4: Arquitectura del módulo de visualización

3.3.2. Interfaz gráfica

El módulo de visualización (UI) utiliza una arquitectura basada en el paradigma del *bucle de eventos* 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 como se aprecia en 1.

Algoritmo 1 El algoritmo del bucle de eventos

```

while queue.waitForEvent() do
    queue.processEvent()
end while
  
```

El *bucle de eventos* procesará todos los eventos de la aplicación, lo que desencadena un conjunto de acciones que modificarán su **estado** de la interfaz de usuario. Este estado será 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 presentarán en un formato gráfico adecuado y sencillo de comprender.

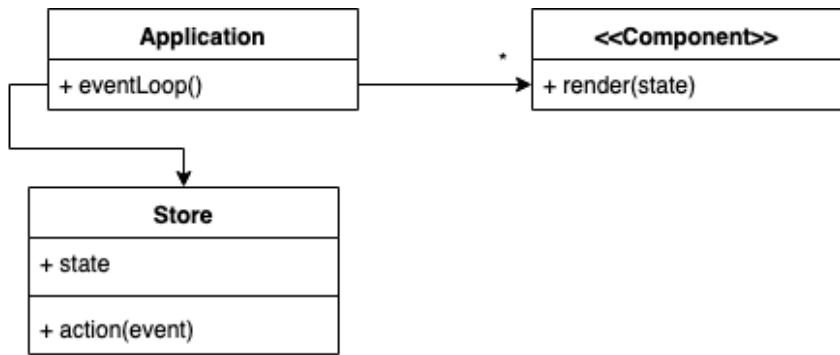


Figura 3.5: Arquitectura general del módulo de interfaz de usuario

Capítulo 4

Implementación

El siguiente capítulo encapsula los detalles de implementación y optimización de los algoritmos que se han desarrollado con el objetivo de computar los factores de forma simples y extendidos.

4.1. Cálculo de factores de forma de la componente difusa

4.1.1. Algoritmo del hemi-cubo

El algoritmo del hemi-cubo fue implementado utilizando la API OpenGL que provee de interfaces de alto nivel para la programación de tarjetas gráficas que facilitan el uso de algoritmos como el del Z-Buffer además de facilitar el manejo de memoria en la GPU.

Para implementar el cálculo de factores de forma se implementará la función `computeFormFactors` como se aprecia en la figura 3.3. Recordando la arquitectura diseñada, la función debe computar completamente la matriz de factores de forma. Esta función seguirá un conjunto de etapas:

1. En primera instancia, se configurarán los *buffers* de memoria necesarios para representar el hemi-cubo que se dibujará. Para ello, se crea un *Frame Buffer Object* en la GPU que estará compuesto de 5 texturas, cada una de ellas correspondiente a una de las caras a dibujar. Cabe destacar, que estas texturas se compondrán de dos imágenes, una de ellas contiene enteros sin signo que serán utilizados para representar un *id* de cara

parche de la escena y la restante contiene los valores de profundidad necesarios para el algoritmo del Z-Buffer.

2. En la segunda se establecen las matrices de transformación de vista, es decir, las transformaciones lineales que alinean el volúmen de vista al hemicubo. Esto implica trasladar el origen de vista a el baricentro de la cara en cuestión, y alinearla a su normal como se ve en la figura.
3. En la tercer 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 *geometry shaders* para realizar una única llamada de dibujado por objeto. Este método solamente realiza un cambio de *render target*, una de las operaciones más costosas según 4.1. Como puede apreciarse en 2 solo se realiza una única llamada de *binding* del hemi-cubo. Específicamente, la implementación sigue el siguiente patrón:
 - a) El *vertex shader* es simplemente *passthrough* lo que significa que conecta las entradas proveídas por la CPU con su salida.
 - b) El *geometry shader* genera cinco primitivas donde cada una estará en las coordenadas correspondientes a los frustums 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 *fragment shader* corregirá 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*.
4. En última instancia es necesario procesar la información del hemi-cubo dibujado para obtener una nueva fila de la matriz **F**. Este proceso puede ser realizado tanto en GPU como CPU y sigue el patrón visto en 6. Ambos métodos fueron implementados y se detallan a continuación:
 - Reducción 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 auxiliar que

contiene los valores de corrección propuestos por Cohen and Greenberg en conjunción con la función `atomicAdd` para sumar las componentes de los factores de forma de cada elemento. Es necesario que esta operación sea atómica para garantizar que dos procesadores no escriban en la misma posición del arreglo de forma concurrente. Este arreglo accesible 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 de forma directa aunque requiere de la sincronización entre GPU-CPU. No obstante, dada la naturaleza de las tarjetas gráficas la adición de condicionales hace que las instrucciones SIMD generen divergencia de hilos y por tanto reducen drásticamente el rendimiento del algoritmo.

- Reducción 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 a información de manera similar a la GPU, aunque de forma secuencial para eliminar la necesidad de barreras de sincronización, mientras tanto se sigue procesando nuevos hemi-cubos en la GPU generando un máximo de concurrencia entre los dispositivos.

Algoritmo 2 Algoritmo de proyección de la escena en un hemi-cubo

```

function computeFormFactors
  bindHemicube()
  loop face  $\in$  scene
    alignCamera(face)
    clearBuffers()
    render(scene)
    hemicube  $\leftarrow$  getHemicube()
    startThread(processHemicube, face, hemicube)
  end loop
end function

```

Algoritmo 3 Procesamiento de una fila de la matriz \mathbf{F} a partir de la información almacenada en una textura cúbica.

```
function processHemicube(face, hemicube)
    row ← [0, ..., 0]
    loop pixel ∈ hemicube :
        factor ← getCorrectionFactor(pixel)
        seenFace ← getFaceId(pixel)
        if isValid(seenFace) then
            row[seenFace] ← +factor
        end if
    end loop
    formFactorMatrix[face] ← row
end function
```

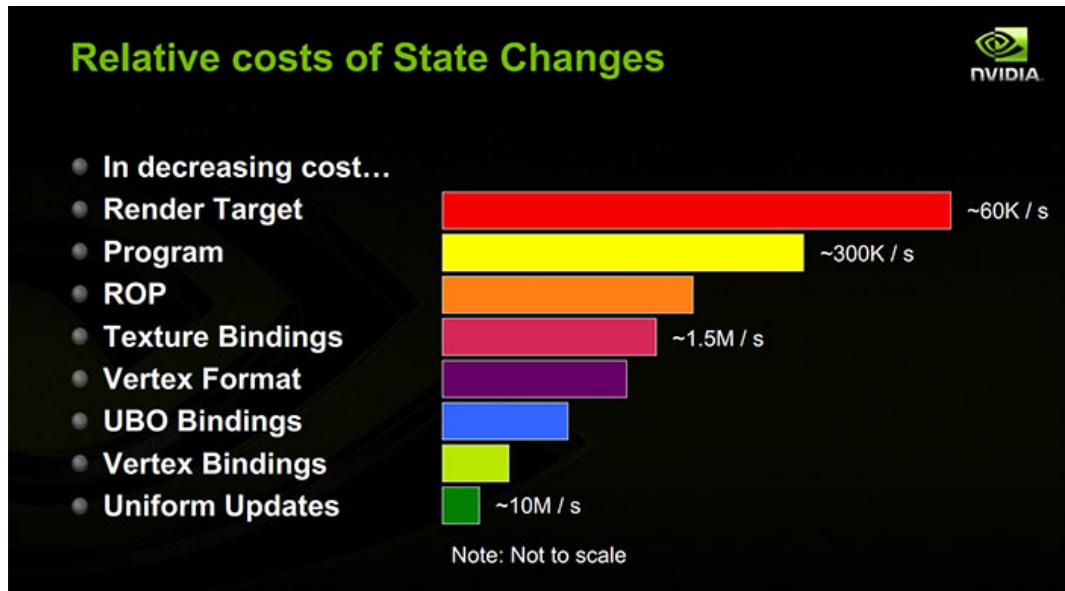


Figura 4.1: Costo de cambios de estado en OpenGL. Fuente: Nvidia

4.1.2. El algoritmo de la hemi-esfera

El algoritmo de la hemi-esfera fue implementado utilizando la biblioteca de traza de rayos Embree. Esta biblioteca soporta el trazado de rayos en la CPU en múltiples superficies, en particular, triángulos y cuadriláteros utilizando BHV (del inglés *Bounding Volume Hierarchies*). 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, cada nivel garantiza la reducción de tamaño de dichas estructuras. Su utilidad radica en la simplificación del cálculo de la intersección rayo-objeto, pues esta la computación de la

intersección de un rayo con un volumen envolvente tiene un costo de cómputo despreciable en comparación al muestreo en una gran cantidad de primitivas. La aceleración proviene en caso de fallo, pues un fallo en la intersección del rayo con el volumen envolvente supone a su vez el fallo de la intersección con todos las primitivas que contiene.

Algoritmo 4 Cálculo de una fila de los factores de forma utilizando traza de rayos

```

function computeFormFactors(face)
    row  $\leftarrow [0, \dots, 0]$ 
    directions  $\leftarrow \text{beckers}(nSamples)$ 
    baricenter  $\leftarrow \text{face.getBaricenter}()$ 
    loop in parallel direction  $\in \text{directons}$  :
        intersection  $\leftarrow \text{traceRay}(\text{scene}, \text{baricenter}, \text{directions})$ 
        seenFace  $\leftarrow \text{getFaceId}(\text{intersection})$ 
        if isValid(seenFace) then
            row[seenFace]  $\leftarrow +\frac{1}{nSamples}$ 
        end if
    end loop
    formFactorMatrix[face]  $\leftarrow \text{row}$ 
end function

```

De forma similar a ?? es necesario posicionar el origen de cada rayo a trazar en el baricentro de la superficie. Luego, recordando (??), es necesario generar un conjunto de direcciones utilizando la distribución del coseno o similar. Si bien originalmente se utilizó la generación de números pseudo-aleatorios para generar rayos correctamente distribuidos, el uso de números aleatorios perjudicó el rendimiento del algoritmo dada la complejidad de los algoritmos que lo computan. Es por esto que se decidió utilizar otro algoritmo de generación de direcciones determinísticas en una hemi-esfera, en este caso la *regla general de particionado de una hemi-esfera en celdas de área equitativa* propuesta por ?. Estas direcciones son pre-calculadas y almacenadas pues previo al procesamiento de se conoce la definición (o cantidad de rayos) que se utilizarán en el cálculo.

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

El muestreo de puntos partiendo del origen de la hemi-esfera en las direcciones determinadas se calcula utilizando la función `rtcIntersect1`, que retorna,

de forma similar a OpenGL un identificador de primitiva relativo al objeto que se dibuja. 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.

4.2. Cálculo de factores de forma de la componente especular

El cálculo de factores de forma extendido en fue implementado en dos variantes para el método del hemi-cubo, y de una única manera para el algoritmo de trazado de rayos.

4.2.1. Extensión del método del hemi-cubo

Ambas variantes son métodos de "dos pasadas", es decir, se dibujará el hemicubo normalmente y luego de determinar cuales de las caras visibles son reflectivas se proyectará nuevamente la escena para determinar qué parches son visibles debido al fenómeno de reflexión.

Los métodos utilizados son heurísticos y su factor de error depende drásticamente del área de los parches, ya que al contrario de la técnica de trazado de rayos se desconoce el punto exacto en que los rayos emitidos desde el hemicubo colisionan con los parches del entorno.

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 los puntos cubiertos por cierta superficie ubicada en el entorno tri-dimensional (como una ventana o una puerta). Normalmente se utiliza esta técnica para optimizar el dibujado de escenas donde la oclusión entre objetos es alta o 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 arbitraria que puede ser utilizada para decidir qué *fragmentos* pasan la prueba del **Stencil Test**.

En el caso de la implementación propuesta, el primer paso consiste en dibujar un *stencil buffer* donde se encuentre el parche cuyo coeficiente de reflexión espectral es mayor a cero desde la dirección simétrica como se aprecia en 4.2, solo se dibujará la sección del espejo. Luego, manteniendo el mismo volumen de vista, se dibujarán los identificadores de los parches de forma similar al dibujado del hemi-cubo salvo que en una textura bi-dimensional, es necesario además establecer un plano de corte en la superficie espectral para evitar el dibujado de los objetos que se encuentren detrás de esta. Este proceso se realiza a una resolución muy baja con el objetivo de preservar el rendimiento y minimizar el costo de transferencia de memoria.

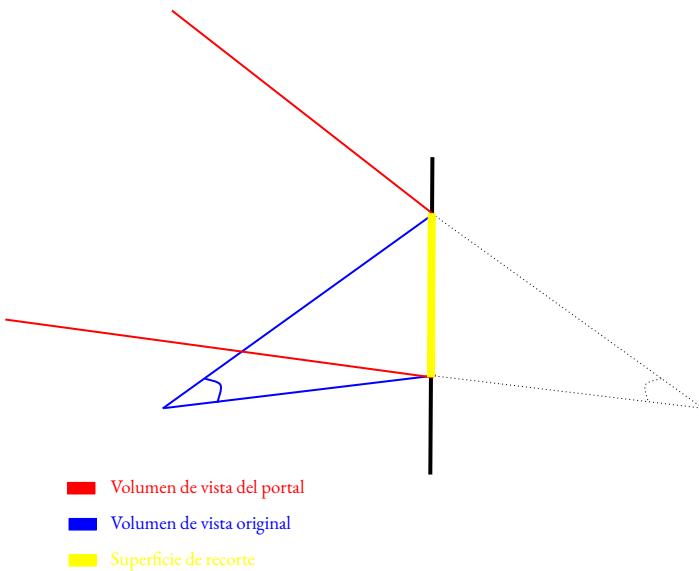


Figura 4.2: Generación del volumen de vista para espejos

Finalmente, se obtienen los identificadores de las caras reflejadas. En caso de que existan caras con valores de reflexión espectral no nulos se vuelven a procesar, en otro caso se utilizarán los identificadores obtenidos para distribuir el factor de forma correspondiente al fragmento del hemi-cubo entre los parches visualizados.

Algoritmo 5 Cálculo de las caras vistas utilizando dibujado de portales

```
function renderPortal(face, targetFace)
    origin ← face.getBarycenter()
    direction ← face.getNormal()
    setCamera(origin, direction)
    bindVertices(targetFace)
    renderStencil(0xFF)
    refDir ← reflected(dir, targetFace.getNormal())
    refOrig ← symmetrical(origin, targetFace.getPlane())
    setCamera(refOrig, refDir)
    bindVertices(scene)
    setStencilFunction(== 0xFF)
    render()
    seenFaces ← readBuffer()
    loop faceId ∈ seenFaces
        if isValid(faceId) then
            row[faceId] ← +( $\frac{1}{nSamples}$ )
            if isReflective(faceId) then
                renderPortal(targetFace, faceId)
            end if
        end if
    end loop
end function
```

Método híbrido

El método híbrido consiste en la utilización del trazado de rayos para computar qué parches son visualizados desde el parche considerado. Para ello, se trazarán rayos desde un conjunto de puntos pertenecientes al parche reflectivo 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 del especular. Las caras vistas, de ser difusas, son las que aportarán fracciones de valor al factor de forma.

Esto emula el fenómeno de la reflexión, aunque introduce pequeños errores al aproximar la dirección real de reflexión (aquella dada por la diferencia entre el baricentro del parche considerado y el punto de intersección con el rayo).

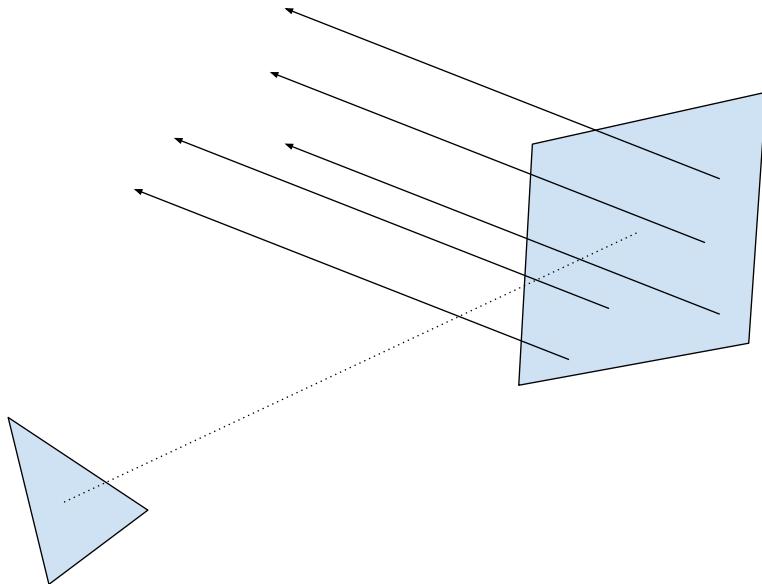


Figura 4.3: Visualización del rebote de rayos al impactar en parches especulares

4.2.2. Extensión del método de la hemi-esfera

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 cuyo coeficiente de reflexión especular es estrictamente positivo se almacenará el total de $k(1-\rho_j)$ (donde $k = \frac{1}{nMuestras}$)

Algoritmo 6 Cálculo de las caras vistas utilizando trazado de rayos

```
function renderReflections(face, targetFace)
    direction ← normalize(targetFace.getNormal() – face.getNormal())
    refDir ← reflected(direction, targetFace.getNormal())
    origins ← getUniformSamples(targetFace)
    loop in parallel origin ∈ origins
        hit ← traceRay(origin, refDir)
        if isValid(hit) then
            row[face] ← +(1/nSamples)
            if isReflective(faceId) then
                renderReflections(targetFace, faceId)
            end if
        end if
    end loop
end function
```

como contribuyente del factor de forma \mathbf{F}_{ij} y se calcularán las siguientes intersecciones con el *residuo* de la reflexión que se distribuirá en los factores de forma que correspondan a la reflexión. Es decir, suponiendo que un rayo impacta S_k desde el camino (S_i, S_j) donde $\rho_j \geq 0$ se agregaría $k\rho_j(1 - \rho_k)$ y se procederá de forma recursiva hasta que $\rho_z = 0$ para una superficie intersecada S_z o se alcance el máximo límite de recursión como se aprecia en la Figura 2.9.

4.3. Cálculo del vector de radiosidad

Recordando 2.5, se han propuesto dos métodos para resolver el sistema. El método exacto supone hallar el vector solución del sistema de ecuaciones dado por $(\mathbf{I} - \mathbf{RFB}) = \mathbf{E}$, el segundo método está regido por el método iterativo dado por (2.13).

En el primer caso, la implementación se realizó utilizando la biblioteca de álgebra lineal *Eigen* pues soporta la resolución de sistemas con matrices esparzas. 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 (hemos demostrado que $\mathbf{I} - \mathbf{RF}$ no lo es), la descomposición **LU** genera dos matrices tal que $\mathbf{M} = \mathbf{LU}$ con **L** triangular inferior y **U** triangular superior.

Fácilmente se puede comprobar que $\mathbf{M}^{-1} = \mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{I}$. Por tanto, $B = (\mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{I})E$.

- Descomposición de Cholesky: La factorización supone que $\mathbf{I} - \mathbf{RF}$ se puede descomponer como \mathbf{LL}^T donde \mathbf{L} es triangular inferior. Esta descomposición permite resolver el sistema trivial a través de las ecuaciones $\mathbf{LB}' = E$ y $\mathbf{L}^T B = B'$.

■ Iterativos

- Gradiente conjugado: Dada una matriz cuadrada y definida positiva (como $\mathbf{I} - \mathbf{RF}$) es posible expresar el vector solución del sistema como $\sum_{i=1}^N \alpha_i p_i$ donde p_k es un conjunto de vectores ortonormales. El método de cálculo de α_k comprende el cálculo de $\frac{p_k \cdot E}{\|p_k\|^2}$.
- Gradiente conjugado estabilizado: Este método ofrece mayor velocidad de convergencia que el anterior, es decir, se necesitan menos iteraciones para alcanzar el resultado deseado.

Por otro lado, se implementó el método completamente iterativo dado por (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 esparzas.

Finalmente, luego de calcular el vector de radiosidades 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 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 en el modelo de iluminación de *Gouraud*. Este proceso implica balancear el valor de radiosidad para cada vértice asignándole el promedio del valor de radiosidad de cada cara (como se aprecia en 4.4) en que se encuentre como muestra el algoritmo 7.

Algoritmo 7 Algoritmo de interpolación de radiosidad para vértices

```
function interpolate( $B$ )
    loop vertex  $\in$  scene
        temp  $\leftarrow$  0
        faces  $\leftarrow$  faces such as vertex  $\in$  face
        loop face  $\in$  faces
            temp  $\leftarrow$   $+B(\text{face})$ 
        end loop
        vertexRadiosity  $\leftarrow$   $\frac{\text{temp}}{\text{length}(\text{faces})}$ 
    end loop
end function
```

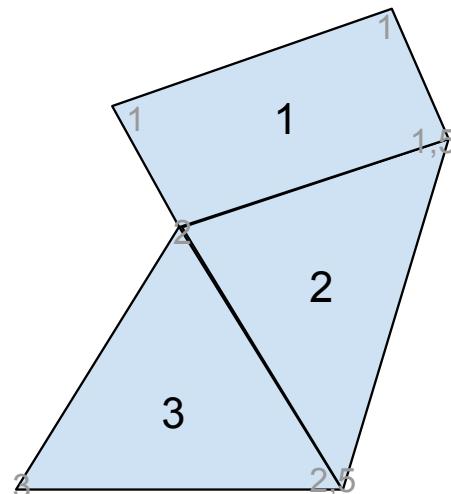
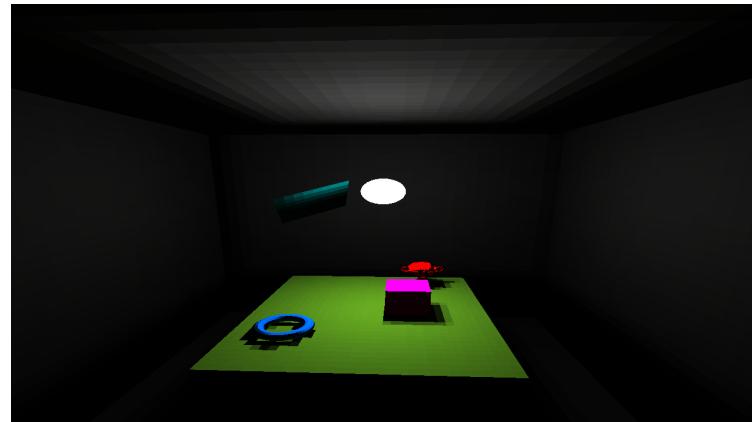
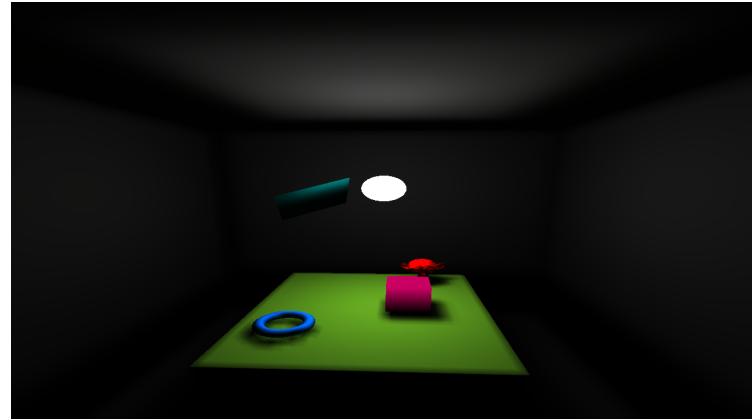


Figura 4.4: Ejemplificación del algoritmo de interpolación

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



(a) Interpolación plana (sin interpolación)



(b) Interpolación de Gouraud

Figura 4.5: Dibujado utilizando distintas funciones de interpolación

4.4. Visualización de resultados y resultados intermedios

La visualización de resultados y resultados 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. Sin embargo, 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, el ultimo 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 contenido en un buffer interno en el que corresponde al estándar. 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, que son suministradas por el Sistema Operativo.

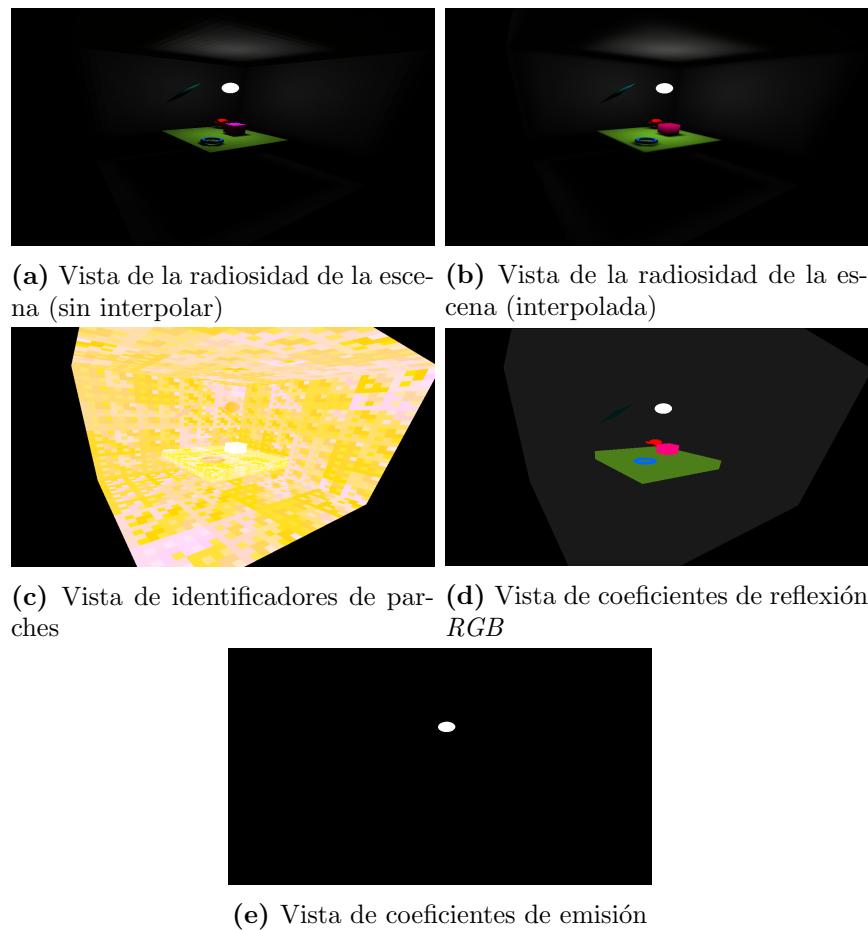


Figura 4.6: Vistas seleccionables por el usuario a través de la interfaz gráfica

4.5. Interfaz de usuario

La interfaz de usuario fue implementada utilizando la biblioteca de dibujo de interfaces gráficas en modo inmediato *ImGui*. Este método de dibujado

implica que los comandos de dibujado 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 era el utilizado en versiones anteriores de OpenGL o Direct3D. Esta biblioteca es reconocida por 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 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 preprocessado: Este panel permite configurar y ejecutar las dos etapas de preprocessado (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 componente *Log* imprime un conjunto de propiedades y detalles del proceso que pueden resultar interesantes, como por ejemplo, los tiempos empleados.

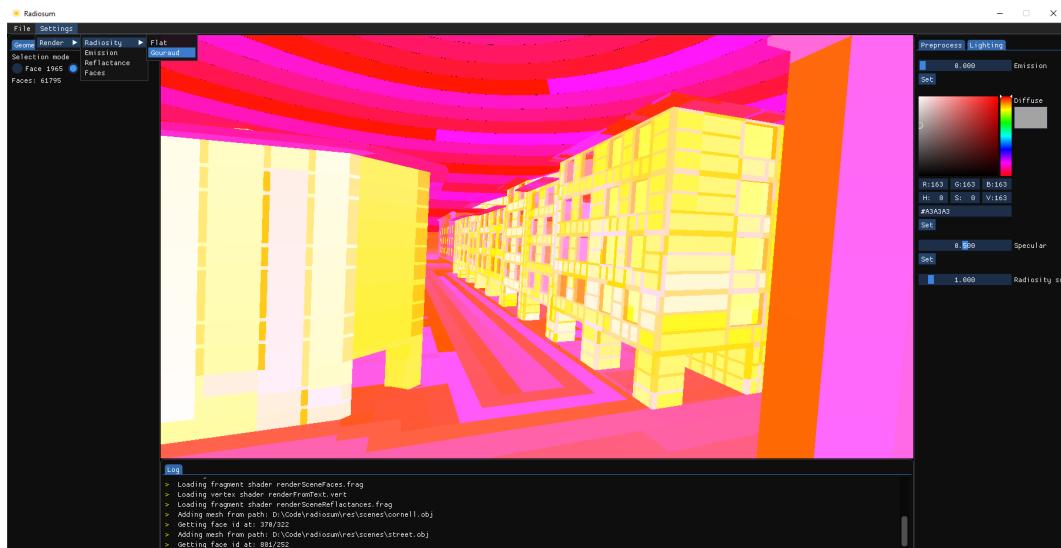


Figura 4.7: Interfaz gráfica implementada

Capítulo 5

Experimental

5.1. Ambiente de prueba

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

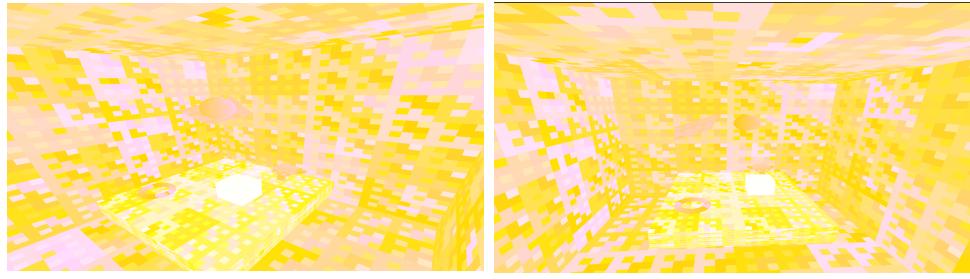
Tabla 5.1: Características del hardware utilizado

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

Tabla 5.2: Características del hardware 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.

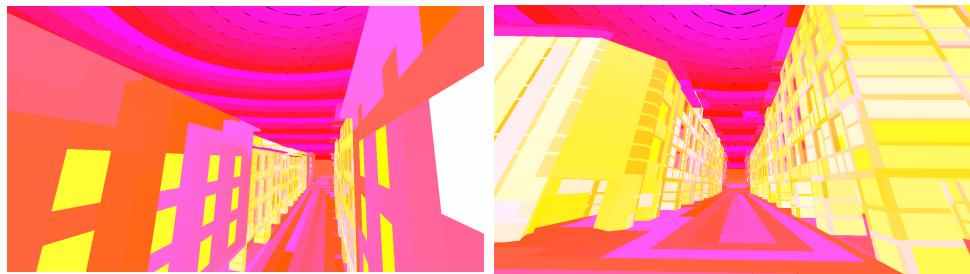


(a) Vista lateral

(b) Vista frontal

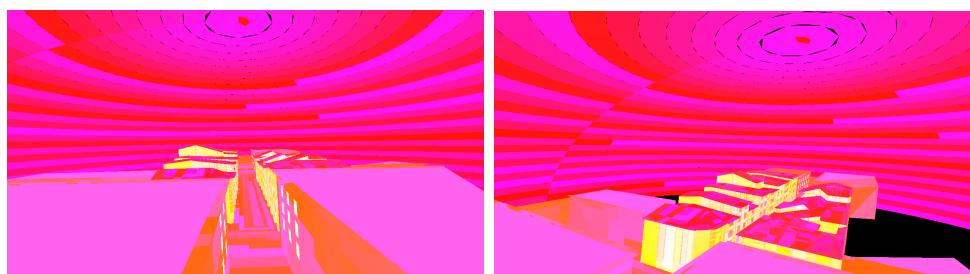
Figura 5.1: Vistas de la escena *Conrnell Box*

Se denominará *Escena - Conrnell Box* a la referente a la figura 5.1. Se basa en un tipo de escena comúnmente usado en el se ubican objetos en el interior de una caja (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 caras de las cuales 96 son triangulares y 12.826 son cuadrilaterales. Cabe notar, que de haber soportado únicamente caras triangulares se procesarían 25.748 parches.



(a) Vista desde la ventana de una de las edificaciones

(b) Vista desde uno de los extremos de la calle



(c) Vista aérea

(d) Vista aérea

Figura 5.2: Vistas de la escena *Calle*

Se denominará *Escena - Calle* a la referente a la figura 5.2. La escena está construida por dos objetos, el primero de ellos es una cúpula (hemi-esfera)

subdividida en 2.407 cuadriláteros de área equitativa, su 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 y Miño et al. con el objetivo de estudiar el fenómeno de la radiación. En total, las escena cuenta con 61.795 caras cuadrilaterales.

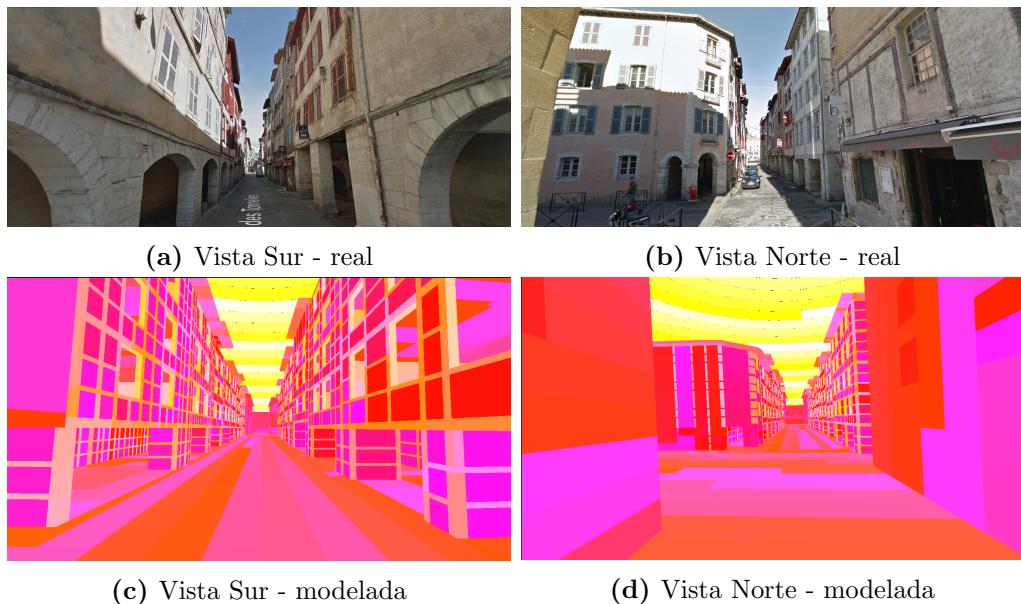


Figura 5.3: Comparación entre el fotografías reales del modelo *Calle* y su representación tridimensional

5.3. Casos de prueba

Se proponen casos de prueba utilizando las escenas descritas en 5.2 utilizando diversos materiales.

5.3.1. Métricas consideradas

Con el objetivo de medir correctamente las ventajas y desventajas de cara acercamiento al cálculo de factores de forma simples y extendidos que se han propuesto, se definirán un conjunto de métricas para evaluar su optimialidad en distintas dimensiones. Cada dimensión considerada se aplicará dependiendo del caso considerado.

- Dimensión rendimiento

- Tiempo de ejecución: Se registrará el tiempo empleado en calcular completamente la matriz de factores de forma.
- Dimensión matriz de factores de forma: Se computará la matriz de factores de forma de control \mathbf{F}_C utilizando la técnica de trazado de rayos con una gran resolución.
 - Error promedio de factor de forma por fila: $Ep_i = \sum_{j=1}^N \frac{|\mathbf{F}_{Cij} - \mathbf{F}_{ij}|}{N}$
 - Error máximo de factor de forma por fila: $Em_i = \max_{j=1}^N |\mathbf{F}_{ij} - \mathbf{F}_{Cij}|$
 - Error estándar de factor de forma por fila: $Em_i = \sqrt{\frac{1}{N} \sum_{j=1}^N (\mathbf{F}_{ij} - \mathbf{F}_{Cij})^2}$
- Dimensión vector de radiosidad (dado el vector R , y el vector de control Rc):
 - Error promedio de radiosidad: $Ep = \sum_{i=1}^N \frac{|R_i - R_{Ci}|}{N}$
 - Error máximo de radiosidad: $Em = \max_{j=1}^N |R_j - R_{Cj}|$
 - Error estándar de radiosidad: $Em = \sqrt{\frac{1}{N} \sum_{j=1}^N (R_j - R_{Cj})^2}$
- Dimensión visual:
 - Calidad de resultados: Se evaluarán los resultados esperando que se asemejen a la realidad.

5.3.2. Descripción de casos de prueba

1. *Prueba difusa*: Se utilizarán materiales estrictamente difusos en ambas escenas, con colores aleatorios. Desactivando, entonces, cualquier interacción especular. Se escogerá un conjunto de parches que oficiarán de fuente luminosa. En caso de la escena *Calle*, se seleccionan 10 parches que emulan la luz solar. Por otro lado, para la escena *Cornell Box* se utiliza la bola central como fuente luminosa.
2. *Prueba especular*: Se utilizarán 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.
3. *Prueba híbrida*: Únicamente en la escena *Cornell Box* se computarán dos variantes. En una de ellas se desactivan los reflectores especulares (plano) y en el otro caso no.
4. Prueba de estrés: Se utiliza gran cantidad de espejos en ambas escenas.

- Prueba de rendimiento: Para cada *pipeline* (completo) implementado se computa la radiosidad registrando el tiempo de renderizado según la cantidad de muestras configurada.

5.3.3. Resultados observados

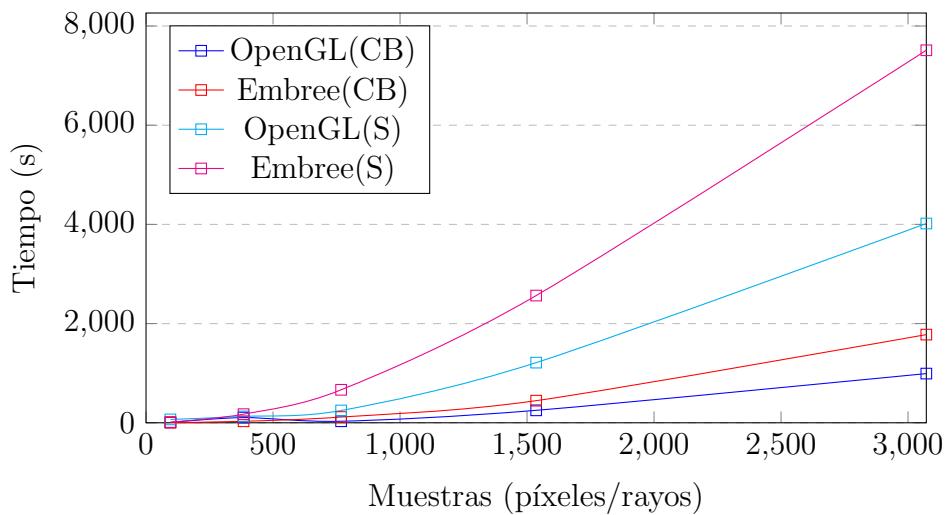
Caso de prueba I

Muestras	Tiempo de ejecución (s)			
	<i>Cornell Box</i>		<i>Street</i>	
	OpenGL	Embree	OpenGL	Embree
96	7	3	68	14
384	10	30	128	174
768	31	116	248	665
1536	251	446	1213	2565
3072	992	1778	4018	7511

Tabla 5.3: Resultados obtenidos para el primer caso de prueba en ambas escenas consideradas

En este caso, se observa en la tabla ?? que el método del hemicubo tiene un rendimiento considerablemente superior al de la traza de rayos. Cabe destacar que se observó una ocupación promedio de la GPU del 30 % y CPU 90 % (esta última probablemente se deba a la gran cantidad de sincronizaciones necesarias entre el dispositivo y el controlador); mientras que el uso de traza de rayos presentó una ocupación de los núcleos del procesador del 99 %. Se destaca la diferencia observada entre OpenGL y Embree en 5.3.3.

Caso de prueba I, rendimiento en segundos.



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 pronunciados son las diferencias en de iluminación entre los parches, es decir, en qué medida difiere la radiosidad entre parches. Para este caso, se pudo observar (véase la figura 5.4) que si bien las resoluciones más bajas consumen menor cantidad de recursos los resultados tienen una calidad sustancialmente menor a la esperada. Considerando que el modelo generalmente es utilizado para el cálculo de mapas de iluminación en una etapa de pre-procesado, 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 .

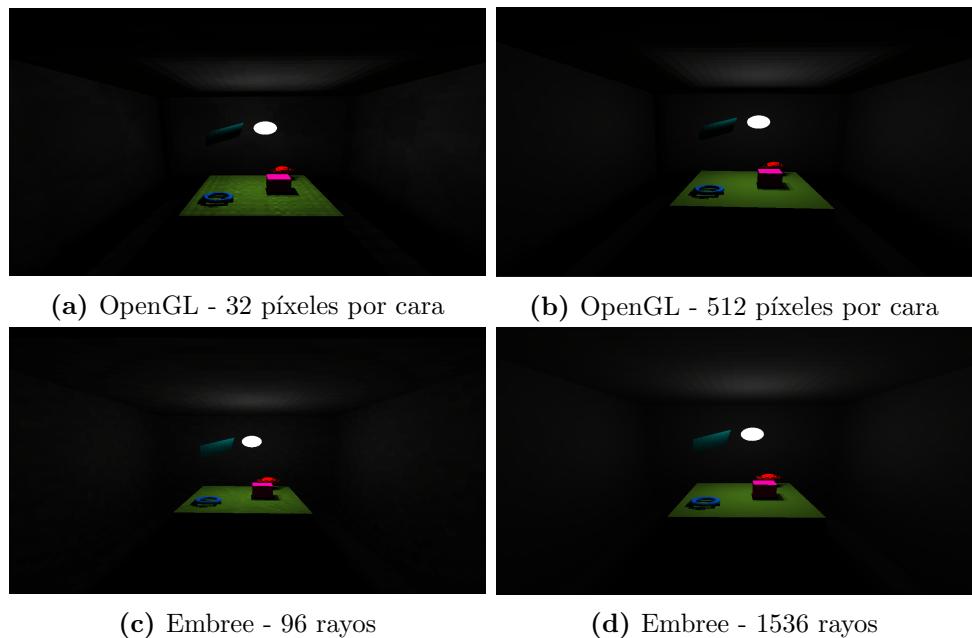


Figura 5.4: Diferencias visuales ajustando la cantidad de muestras

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

6.2. Trabajo futuro

Lista de figuras

2.1	Dibujado utilizando distintos modelos de iluminación	3
2.2	Comparación entre tipos de reflectores	5
2.3	El factor de forma entre dos superficies	7
2.4	La analogía de Nusslet	8
2.5	El <i>rendering pipeline</i> de OpenGL	11
2.6	Representación gráfica del método del hemicubo	13
2.7	Representación gráfica de los ejes considerados para el factor de corrección de los factores de forma	14
2.8	Representación gráfica del método de trazado de rayos	15
2.9	Representación gráfica del cálculo del factor de forma extendido donde k corresponde al inverso de la cantidad de muestras tomadas.	17
2.10	Vista general de la arquitectura de OpenGL	19
2.11	Vista general de la arquitectura de Embree	21
3.1	Tabla de Kanban utilizada en el proyecto	25
3.2	Módulo de manejo de geometría	26
3.3	Arquitectura del módulo de pre-procesado	27
3.4	Arquitectura del módulo de visualización	28
3.5	Arquitectura general del módulo de interfaz de usuario	29
4.1	Costo de cambios de estado en OpenGL. Fuente: Nvidia	33
4.2	Generación del volumen de vista para espejos	36
4.3	Visualización del rebote de rayos al impactar en parches especulares	38
4.4	Ejemplificación del algoritmo de interpolación	41
4.5	Dibujado utilizando distintas funciones de interpolación	42
4.6	Vistas seleccionables por el usuario a través de la interfaz gráfica	43

4.7	Interfaz gráfica implementada	44
5.1	Vistas de la escena <i>Conrnell Box</i>	46
5.2	Vistas de la escena <i>Calle</i>	46
5.3	Comparación entre el fotografías reales del modelo <i>Calle</i> y su representación tridimensional	47
5.4	Diferencias visuales ajustando la cantidad de muestras	50

Lista de tablas

5.1	Características del hardware utilizado	45
5.2	Características del hardware utilizado	45
5.3	Resultados obtenidos para el primer caso de prueba en ambas escenas consideradas	49

APÉNDICES

Apéndice

Referencias bibliográficas

- [1] James T Kajiya. The rendering equation. In ACM SIGGRAPH computer graphics, volume 20, pages 143–150. ACM, 1986.
- [2] Bui Tuong Phong. Illumination for computer generated pictures. Communications of the ACM, 18(6):311–317, 1975.
- [3] 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.
- [4] 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.
- [5] TJ Malley. A shading method for computer generated images. Master's thesis, Dept. of Computer Science, University of Utah, 1988.
- [6] Benoit Beckers and Pierre Beckers. A general rule for disk and hemisphere partition into equal-area cells. Computational Geometry, 45(7):275–283, 2012.
- [7] 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.
- [8] 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.
- [9] Turner Whitted. An improved illumination model for shaded display. In ACM Siggraph 2005 Courses, page 4. ACM, 2005.

- [10] Xavier Granier, George Drettakis, and Bruce Walter. Fast global illumination including specular effects. In Rendering Techniques 2000, pages 47–58. Springer, 2000.
- [11] Arjan JF Kok, Celal Yilmaz, and Laurens HJ Bierens. A two-pass radiosity method for b  zier patches. In Photorealism in Computer Graphics, pages 115–124. Springer, 1992.
- [12] Jairo Acu  a Paz y Mi  o, Vincent Lefort, Claire Lawrence, and Benoit Beckers. Maquette num  rique d'une rue du vieux bayonne pour son  tude thermique par  l ments finis. A la pointe du BIM: Ing nierie et architecture, enseignement et recherche, page 103, 2018.