



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

RESUMEN

Los algoritmos de iluminación global simulan el comportamiento de la luz en la naturaleza, con objetivos que involucran áreas como la síntesis de imágenes fotorealistas generadas por computadora o la evaluación del diseño lumínico para arquitectura.

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. De modo de simplificar el cálculo del factor de relación geométrico se subdivide la escena en una cantidad de superficies planas discreta. Es decir, las superficies del espacio deben ser subdivididas en otras planas más pequeñas a las que llamaremos parches.

La extensión planteada propone la construcción de un algoritmo capaz de incorporar los efectos observados al considerar superficies especulares. Además de la incorporación de otras técnicas que permitan proveer nuevos acercamientos al cálculo de las relaciones que se establecen entre dos objetos en la escena que consideren las posibles optimizaciones a realizar en el hardware actual.

Palabras claves:

Iluminación Global, Radiosidad, Reflexión Especular.

Tabla de contenidos

1	Introducción	1
1.1	Motivación y problema	1
1.2	Objetivos	3
1.3	Estructura del documento	3
2	Estado del arte	5
2.1	Modelos de iluminación	5
2.1.1	Iluminación Local	6
2.1.2	Iluminación Global	7
2.2	Radiosidad	7
2.2.1	Radiosidad en superficies lambertianas	8
2.3	Métodos de cálculo de la matriz de Factores de Forma	11
2.3.1	Rasterización	12
2.3.2	Trazado de rayos	16
2.4	Superficies especulares	18
2.5	Cálculo del vector de radiosidades	19
2.6	OpenGL	20
2.6.1	Arquitectura	21
2.6.2	Extensiones	21
2.7	Embree	22
2.8	Trabajos relacionados	23
3	Solución propuesta	27
3.1	Alcance y objetivos	27
3.2	Proceso de desarrollo	27
3.3	Diseño	28
3.3.1	Motor de renderizado	29
3.3.2	Interfaz gráfica	31

4 Implementación	33
4.1 Cálculo de factores de forma de la componente difusa	33
4.1.1 Algoritmo del hemi-cubo	33
4.1.2 El algoritmo de la hemiesfera	36
4.2 Cálculo de factores de forma de la componente especular	38
4.2.1 Extensión del método del hemicubo	38
4.2.2 Extensión del método de la hemiesfera	41
4.3 Cálculo del vector de radiosidad	42
4.4 Visualización de resultados y resultados intermedios	45
4.5 Interfaz de usuario	46
5 Experimental	49
5.1 Ambiente de prueba	49
5.2 Escenas	50
5.3 Casos de prueba	52
5.3.1 Métricas consideradas	52
5.3.2 Descripción de casos de prueba	53
5.3.3 Resultados observados	53
6 Conclusiones y trabajo futuro	61
6.1 Conclusiones	61
6.2 Trabajo futuro	62
Lista de figuras	65
Lista de tablas	67
Referencias bibliográficas	69

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, que 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 ello, se han desarrollado y formalizado un conjunto de algoritmos y herramientas que resuelven el problema de forma parcial o incluso completa. Para caracterizar los problemas descriptos, se ha generado 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: lumínicos o materiales. Estas expresiones son 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** (luz), así como los eventos **S** (reflexión especular), **D** (reflexión difusa).

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 **LS-D*C**, 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 occlusiones 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*C**. 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 siglos XX y XXI 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.

Históricamente, se destacan dos acercamientos al cálculo de la iluminación. La traza de rayos y el método de radiosidad. Estas técnicas fueron específicamente diseñadas para simular la iluminación indirecta, es decir, los caminos de iluminación de la forma **L(S|D)*E**. Estos son los caminos trazados por un haz de luz que parte de la fuente luminosa, refleja en superficies especulares o difusas y llega al punto de vista del observador.

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

El avance del *hardware* en los últimos años obliga a reevaluar los algoritmos de iluminación global constantemente, con el objetivo de proveer resultados fotorealistas en tiempos de ejecución cada vez menores. Si bien en los últimos años se han desarrollado las técnicas de *trazado de rayos bi-direccional* y otras técnicas de *trazado de rayos de Monte Carlo* con el objetivo de atacar el problema 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 $\mathbf{L}(\mathbf{S}|\mathbf{D})*\mathbf{E}$, generando adaptaciones de las extensiones propuestas por la Academia a un conjunto de técnicas establecidas en la Industria. 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 eficiente de los recursos de cómputo a través del 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 posibles 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*»:

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

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 su valor preciso. 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.

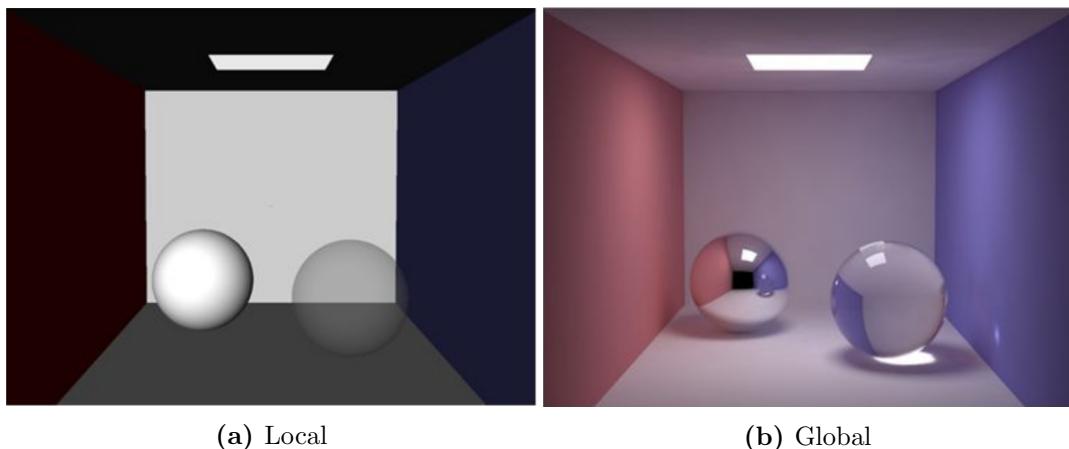


Figura 2.1: Dibujado utilizando distintos modelos de iluminación.

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 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 características 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.

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 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.], 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.

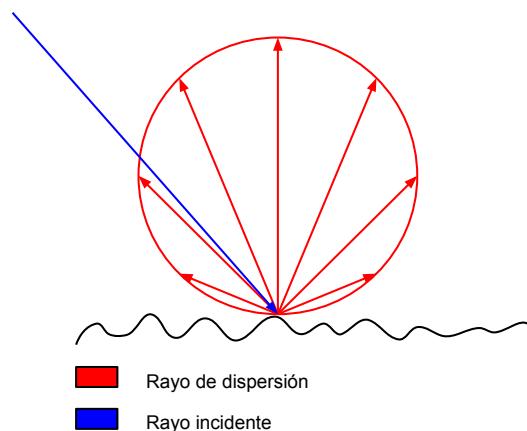


Figura 2.2: Reflector lambertiano

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

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

donde:

- ω es la dirección de vista.
- I es la intensidad de la radiación para un punto de vista particular.

- δP es la energía de la radiación que emana la superficie en la 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). Para ello, se discretizan las superficies en parches difusos, lo que transforma la Eq. (2.3) en:

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

donde:

- B_i es la intensidad lumínica (radiosidad) que deja la superficie i .
- E_i es la intensidad lumínica directamente emitida por i .
- ρ_i es la reflectividad 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_i) 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 ρ_i 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 , δA_j , representados en la Figura 2.3, 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:

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

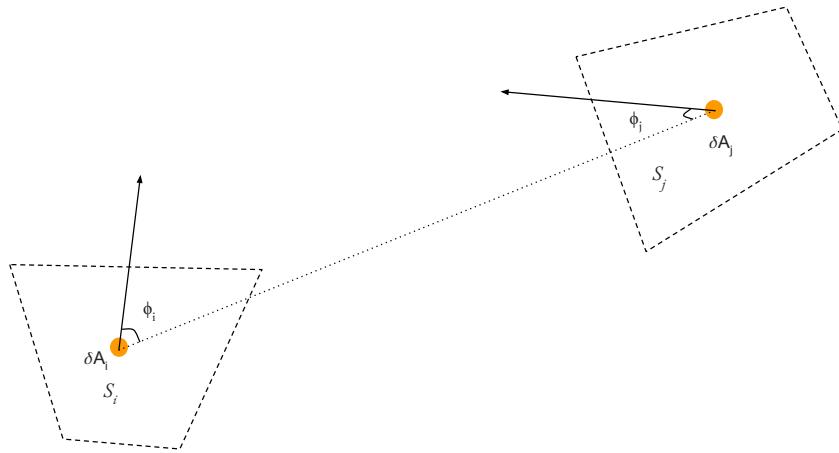


Figura 2.3: El factor de forma entre dos superficies

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

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

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

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

De (2.7) se obtienen las siguientes propiedades:

1. $A_i F_{ij} = A_j F_{ji}$, 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 significa que el factor de forma de cada parche con respecto a sí mismo será siempre nulo.

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 Eq. (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 ubicada en el baricentro de S_i y luego en un disco centrado en S_i .

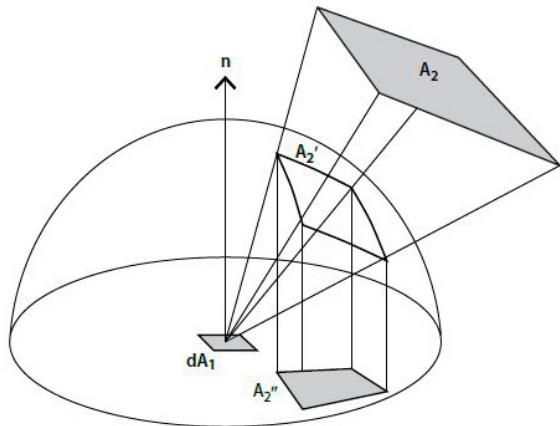


Figura 2.4: La analogía de Nusslet

El cálculo de la matriz de factores de forma \mathbf{F} supone la proyección de los parches, de aquí en más se asumirá que estos parches son polígonos no curvos y lo que permite 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 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:

- 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 replazarlas.
- 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.

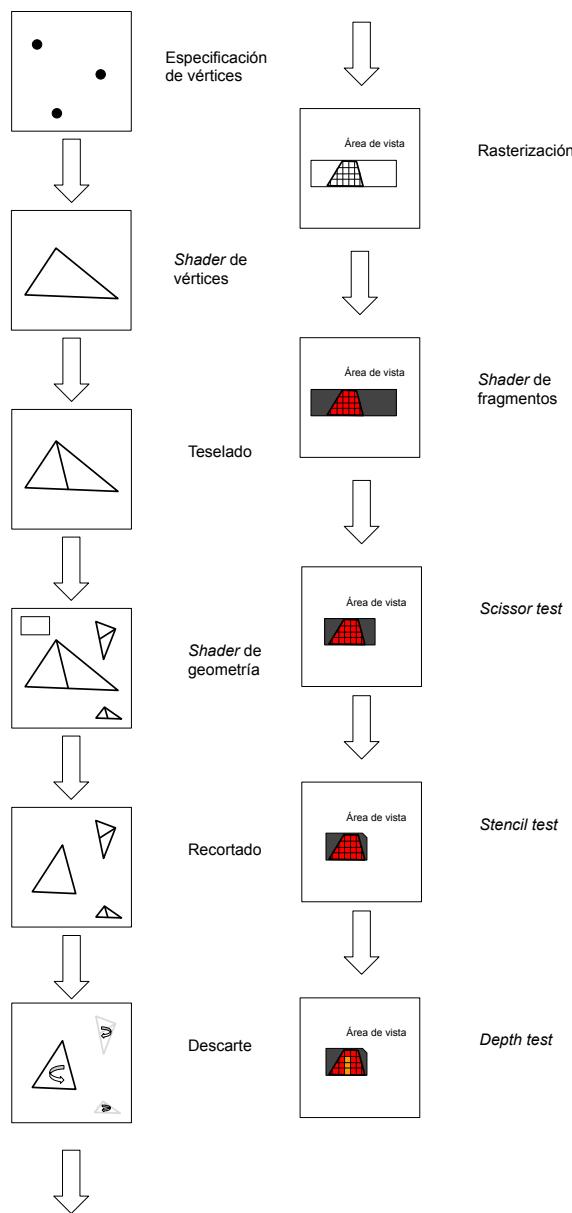


Figura 2.5: El *rendering pipeline* de OpenGL

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 por sus siglas en inglés). Con el objetivo de aprovechar este hardware [Cohen y Greenberg [3]] 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 planas a gran velocidad.

El método original de cálculo de factores de forma propone la proyección de la escena una hemisferia 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 planar, y por tanto puede ser realizada utilizando la rasterización.

Para utilizar el hardware eficientemente consideraremos 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, propuesto originalmente por [Cohen y 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 el hemiesferio 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 Figura 2.8.

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 .

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

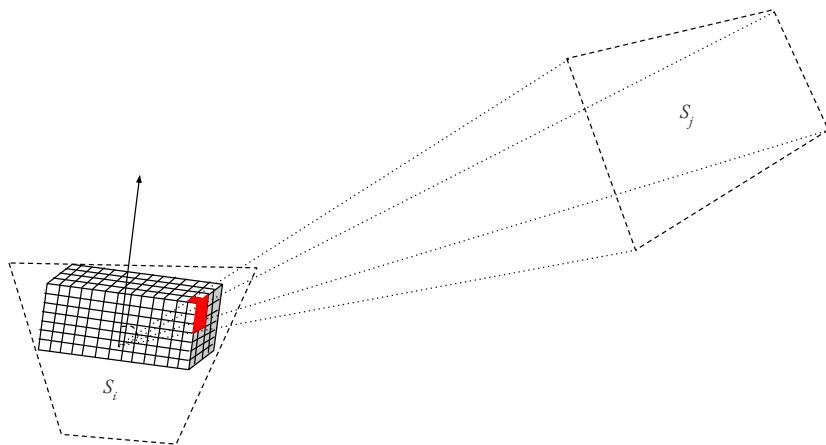


Figura 2.6: Representación gráfica del método del hemicubo

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:

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

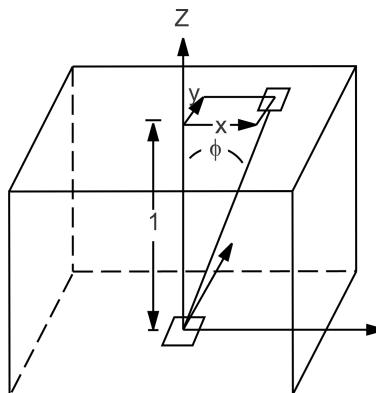


Figura 2.7: Representación gráfica de los ejes considerados para el factor de corrección de los factores de forma

2.3.2. Trazado de rayos

Otra de las técnicas de 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 trazado de rayos es una técnica efectiva [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 interacciona. En este algoritmo, la integral se resuelve con un método de Monte Carlo, donde cada rayo representa una muestra estadísticamente independiente.

El método del hemiesferio

El algoritmo de ray tracing puede ser utilizado para el cálculo de factores de forma, en particular para resolver la doble integral presentada en la Eq. (2.7).

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 $f(x) = \frac{1}{2}[1 + \cos((x - 1)\pi)]$.

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

donde:

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

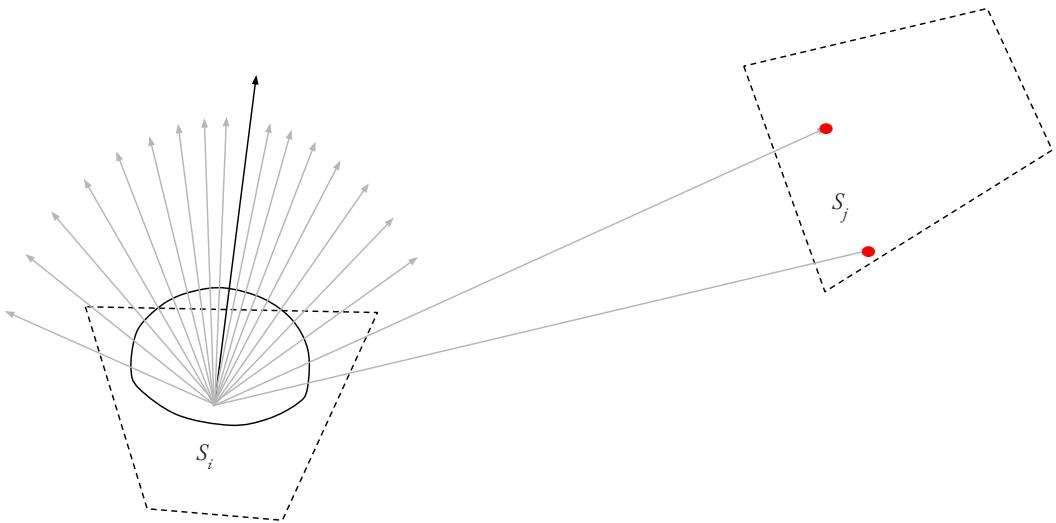


Figura 2.8: Representación gráfica del método de trazado de rayos para el cálculo de factores de forma

Cabe destacar que, no es necesario utilizar una distribución de probabilidad con valores aleatorios o pseudo-aleatorios, sino que, si la cantidad de rayos utilizados es la suficiente y además se utiliza una función que distribuya correctamente cada rayo, los resultados obtenidos se aproximan a los reales.

Para esto, pueden utilizarse otras distribuciones para la dirección de traza. Particularmente, una de ellas es la propuesta por [Beckers and Beckers [6]], presenta un método general de teselación de discos y hemi-esferas. Es deseable el hecho de que la propuesta para hemiesferas genera un conjunto de celdas de igual área 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 trazaran 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, 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 existe la extensión del método para superficies especulares o refractantes propuesto por [Sillion and Puech [7]]. Los autores proponen extender el significado del término *factor de forma* a más que una mera relación geométrica.

ca entre parches. El nuevo factor de forma \mathbf{F}_{ij} corresponde a la proporción de energía que deja la superficie i y llega la superficie j luego de un número de reflexiones y refracciones especulares.

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.

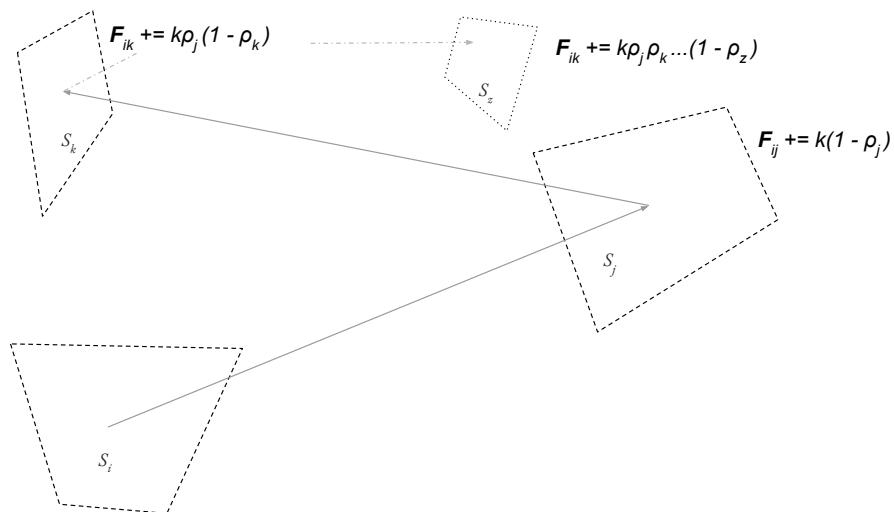


Figura 2.9: Representación gráfica del cálculo del factor de forma extendido donde $k = \frac{1}{N}$, con N muestras tomadas.

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 la Eq. (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 observa-

das.

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

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

Esto implica que las entradas de \mathbf{RF} son siempre menores a 1, por tanto la matriz $(\mathbf{I} - \mathbf{RF}) = M$ 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: $\|\mathbf{RFB}^{(i+1)}\| < \|\mathbf{RFB}^{(i)}\|$. El método planteado en la Eq. (2.13) es el método de Jacobi.

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

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 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 .

2.6. OpenGL

Con el objetivo de proveer interfaces estandarizadas para el uso de tarjetas gráficas y los distintos algoritmos relacionados a la rasterización existen un conjunto de interfaces que abstraen los recursos necesarios (hardware, sistema

operativo). 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 2.3.1). 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 (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 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.

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

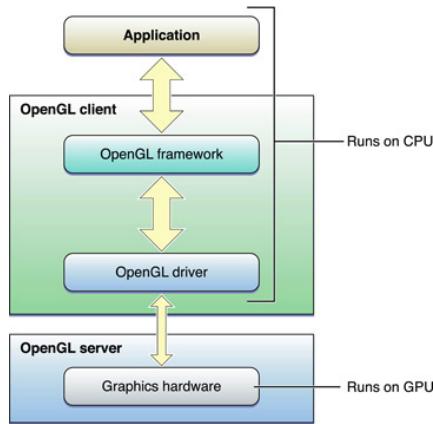


Figura 2.10: Vista general de la arquitectura de OpenGL

2.7. Embree

Los distintos algoritmos para evaluar la intersección entre superficies y rayos han evolucionado a gran velocidad, introduciéndose los conceptos de volumen envolvente y jerarquías de escena. Esto resulta en un gran re-trabajo 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.

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

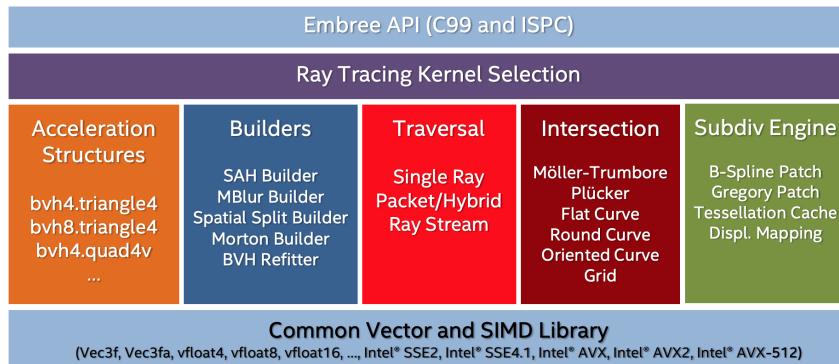


Figura 2.11: Vista general de la arquitectura de Embree

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.

El algoritmo propuesto por [Shirley [8]] calcula la iluminación difusa utilizando dos pasadas. Este algoritmo difiere del propuesto por [Sillion y Pueach [7]] 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 [9]]) sustituyendo el término de ambiente por el calculado en las pasadas anteriores.

Otro acercamiento al problema es el método propuesto de [Kok [10]] 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 y Puech [7]] 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 [11]]. Los autores proponen, nuevamente, un método de dos pasadas donde el factor de forma está compuesto por tres

partes el factor 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 el segundo se relaciona con el fenómeno de la refracción y es calculado integrando en el hemiesferio opuesta 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 hemicubo para calcular los factores de forma delanteros, mientras que los traseros son calculados invirtiendo el hemicubo. Para calcular las componentes especulares, se utiliza un método similar al propuesto en el dibujado de portales, 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 dibujado de portales
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.
- 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.

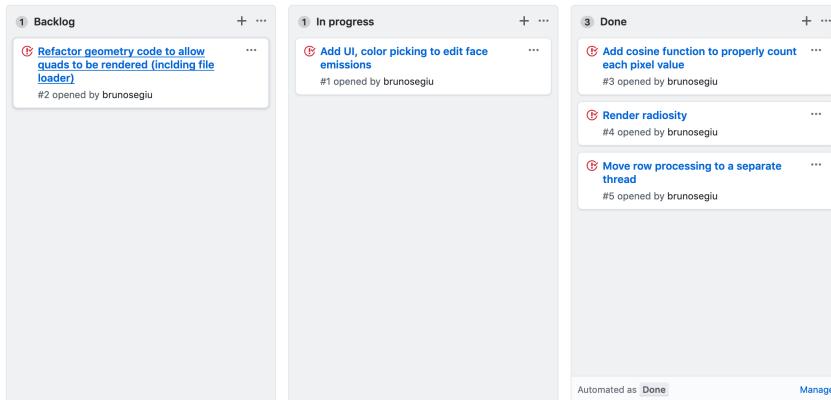


Figura 3.1: Tabla de Kanban utilizada en el proyecto

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 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`, diagramada en la Figura 3.2, cargará distintos objetos desde el disco duro que serán manejados como una instancia de `Mesh`.

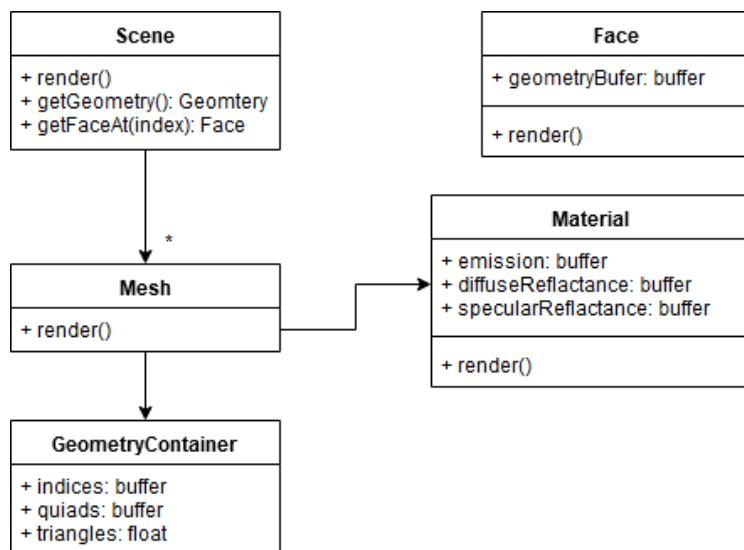


Figura 3.2: Módulo de manejo de geometría

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.

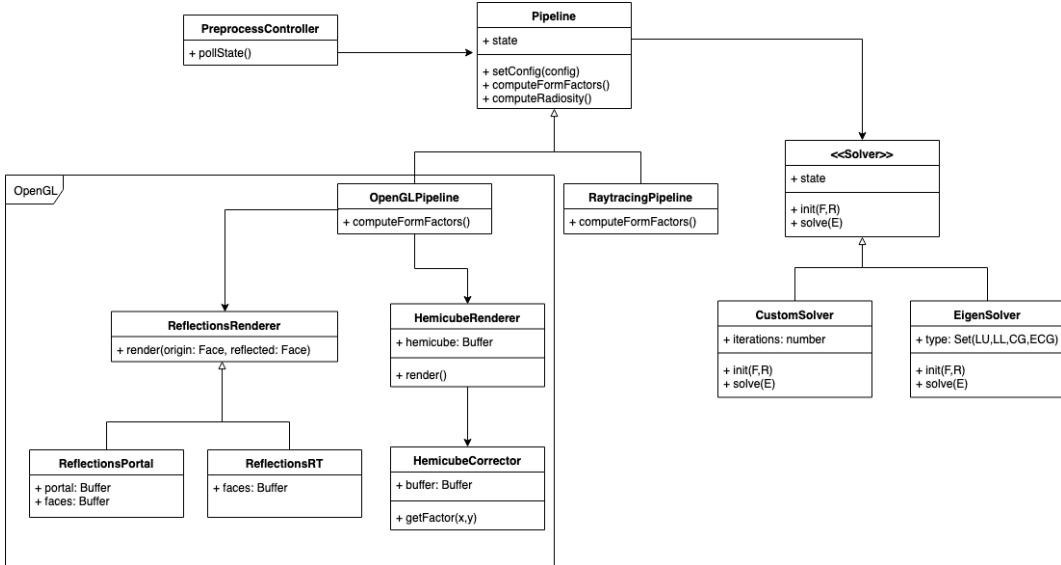


Figura 3.3: Arquitectura del módulo de pre-procesado

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

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

Donde `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.

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 y de sus caras. El proceso de renderizado comienza con una instancia de la clase `DisplayController` 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.

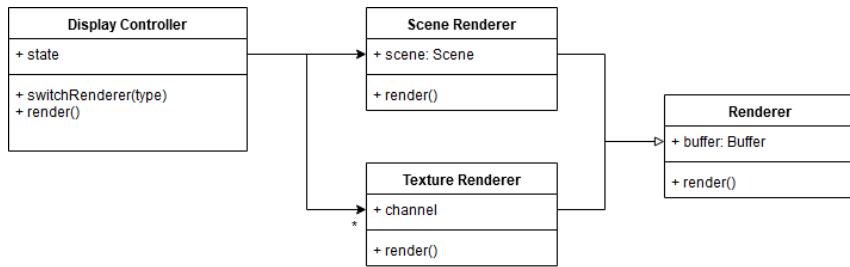


Figura 3.4: Arquitectura del módulo de visualización

3.3.2. Interfaz gráfica

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

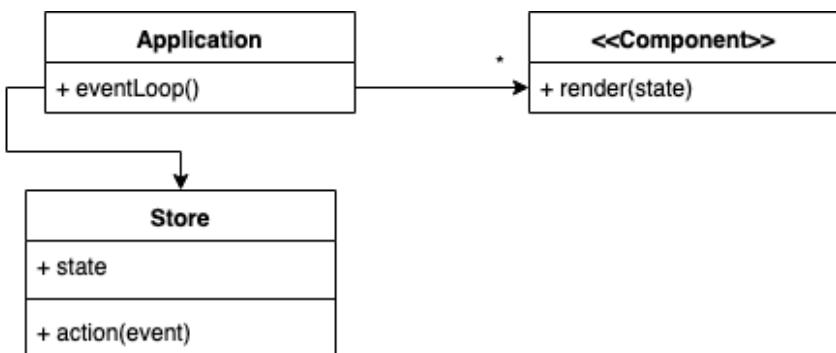


Figura 3.5: Arquitectura general del módulo de interfaz de usuario

Capítulo 4

Implementación

El siguiente capítulo 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

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 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 seguirá las siguientes de etapas:

1. En primera instancia, serán configurados los *buffers* en memoria necesarios para representar el hemi-cubo. Para ello, se creará 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 cada parche de la

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

2. En la segunda etapa se establecerán 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 *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 la Figura 4.1. Como puede apreciarse en el Algoritmo 1 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 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 *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 el Algoritmo 2. 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 (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. Es necesario que esta operación sea atómica para garantizar que dos procesadores no escriban en la misma posición del arreglo de forma concurrente. Es posible acceder al 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 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. Esto se realiza concurrentemente con el procesamiento de nuevos hemi-cubos en la GPU, generando un buen nivel de paralelismo entre los dispositivos.

Algoritmo 1 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 2 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
```

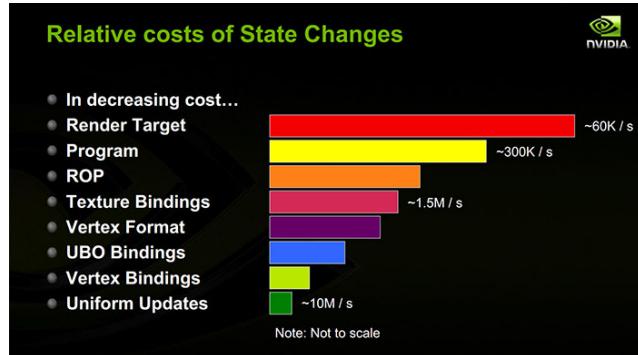


Figura 4.1: Costo de cambios de estado en OpenGL referencia [12]

4.1.2. El algoritmo de la hemiesfera

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

```

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

```

De forma similar a 2.3.1 es necesario posicionar el origen de cada rayo a trazar en el baricentro de la superficie. Luego, recordando la Eq. (2.11), se generan 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 hemiesfera [Beckers y Beckers [6]]. 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` 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 $\frac{1}{n_{\text{Muestras}}}$ para cada rayo.

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, y de una única manera para el algoritmo de la hemiesfera.

4.2.1. Extensión del método del hemicubo

Ambas variantes son métodos de “dos pasadas”, es decir, se dibujará el hemicubo normalmente y luego de determinar cuales de las caras visibles tienen componente especular se proyectará nuevamente la escena para determinar qué parches son visibles a través de la 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 la implementación propuesta, el primer paso consiste en dibujar un parche cuyo coeficiente de reflexión especular es mayor a cero desde la dirección simétrica a aquella donde se encuentra el hemicubo, como se aprecia en la Figura 4.2. La imagen resultante se almacenará en un *stencil buffer*, como se aprecia 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 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.

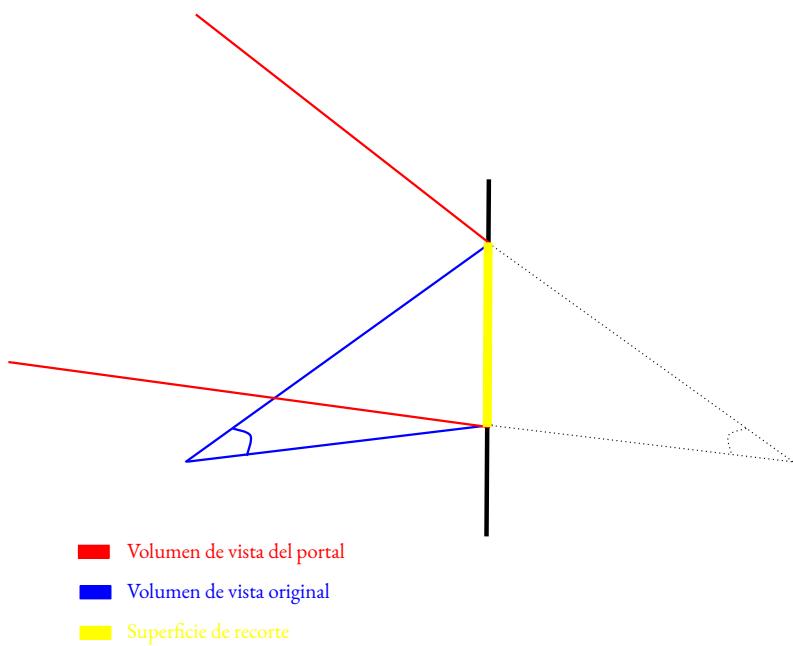


Figura 4.2: Generación del volumen de vista para espejos

Finalmente, se obtienen los identificadores de las caras reflejadas. En caso de que existan caras con valores de reflexión especular no nulos se vuelven a procesar. Este procesamiento recursivo puede ocurrir un número finito de veces, particularmente en el contexto de este proyecto se podrían dibujar caminos de hasta 10 rebotes. En caso de no existir superficies especulares en la imagen se utilizarán los identificadores obtenidos para distribuir el factor de forma correspondiente al fragmento del hemicubo entre los parches visualizados.

Algoritmo 4 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(direction, targetFace.getNormal())
    refOrig ← symmetrical(origin, targetFace.getPlane())
    setCamera(refOrig, refDir)
    bindVertices(scene)
    setStencilFunction(== 0xFF)
    render()
    seenFaces ← readBuffer()
    loop faceId ∈ seenFaces
        if notVoid(faceId) then
            row[faceId] ← +(1/nSamples)
            if isSpecular(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 hemicubo a través de parches especulares. 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 del 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 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 del rayo).

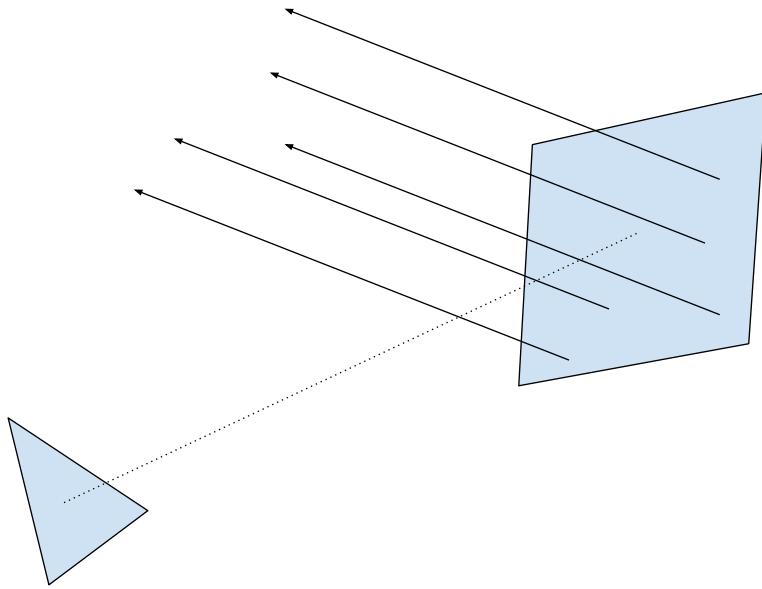


Figura 4.3: Visualización del rebote de rayos al impactar en parches especulares

Algoritmo 5 Cálculo de las caras vistas utilizando trazado de rayos

```

function renderReflections(face, targetFace)
    direction  $\leftarrow$  normalize(targetFace.getNormal()  $-$  face.getNormal())
    refDir  $\leftarrow$  reflected(dir, targetFace.getNormal())
    origins  $\leftarrow$  getUniformSamples(targetFace)
    loop in parallel origin  $\in$  origins
        hit  $\leftarrow$  traceRay(origin, refDir)
        if isValid(hit) then
            row[face]  $\leftarrow$   $+(\frac{1}{nSamples})$ 
            if isReflective(faceId) then
                renderReflections(targetFace, faceId)
            end if
        end if
    end loop
end function

```

4.2.2. Extensión del método de la hemiesfera

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 mayor a cero se almacenará el total de $k(1 - \rho_j)$ (donde $k = \frac{1}{n_{\text{Muestras}}}$) 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 $\rho_j \geq 0$ se agregará $k\rho_j(1 - \rho_k)$ 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 la Sección 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 $(\mathbf{I} - \mathbf{RF} = E)$, mientras que el segundo método está regido por el esquema iterativo dado por la Eq. (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 dispersas. 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 **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 **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

- Gradiante 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 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 largo fijo.

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 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 la Figura 4.4) en que se encuentre como muestra el Algoritmo 6.

Algoritmo 6 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  $\leftarrow$  + $B$ (face)
    end loop
    vertexRadiosity  $\leftarrow$   $\frac{\text{temp}}{\text{length}(\text{faces})}$ 
  end loop
end function

```

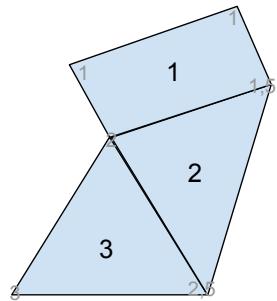
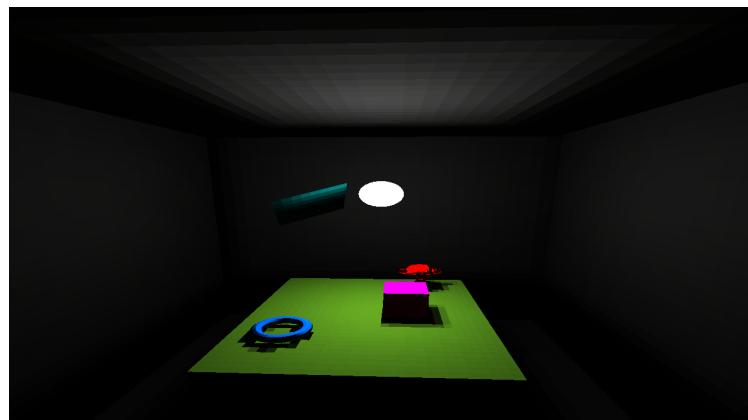
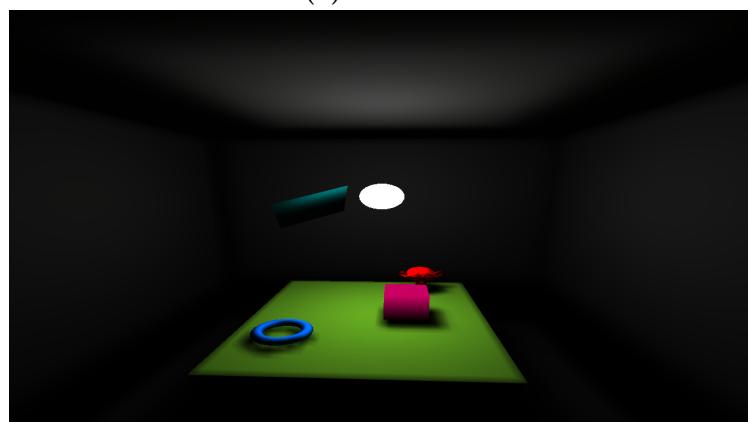


Figura 4.4: Ejemplificación del algoritmo de interpolación

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



(a) Facetado



(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. 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 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.

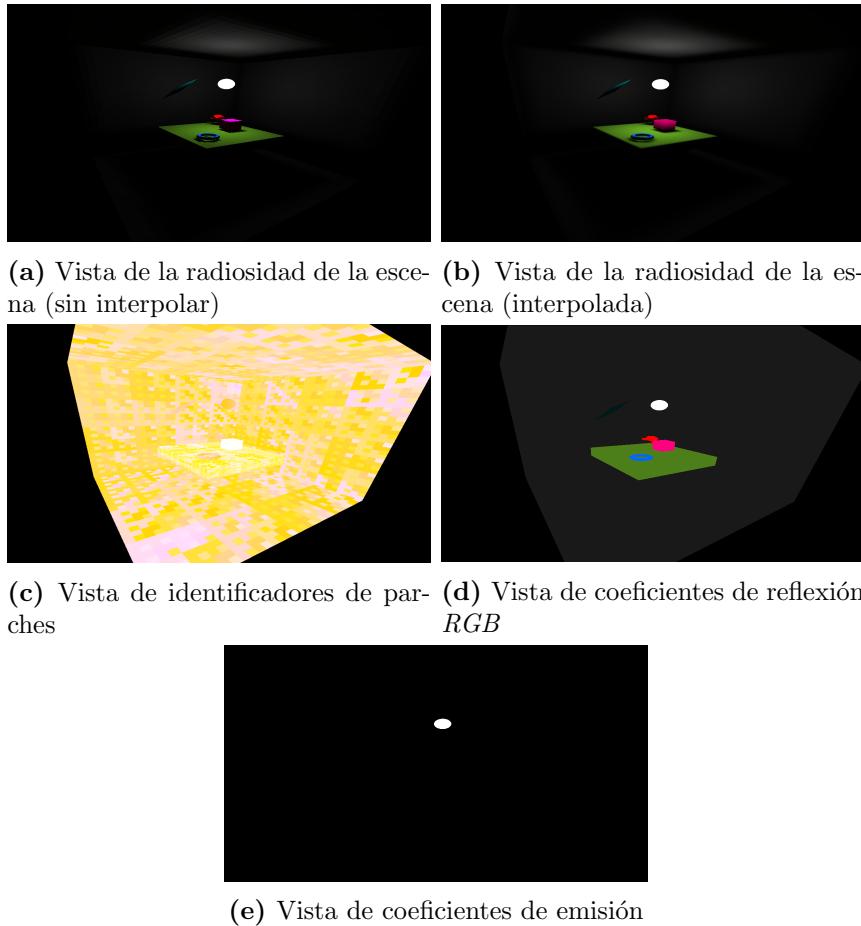


Figura 4.6: Vistas seleccionables por el usuario a través de la interfaz gráfica

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 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 fue 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 *Log* imprime un conjunto de propiedades y detalles del proceso que pueden resultar interesantes, como por ejemplo los tiempos empleados.

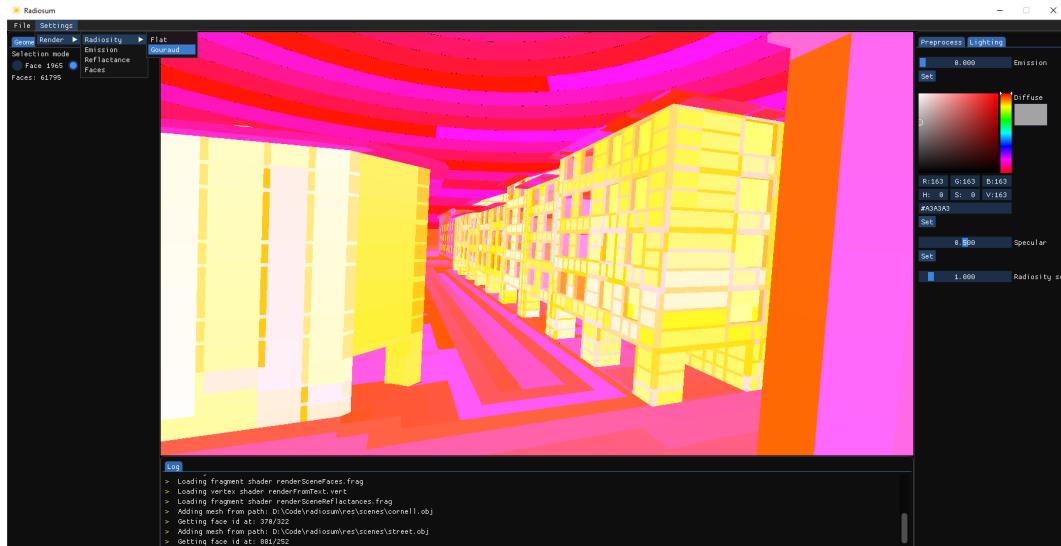


Figura 4.7: Interfaz gráfica implementada

Capítulo 5

Experimental

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

5.1. Ambiente de prueba

De forma que los resultados sean extrapolables a otros ambientes de prueba y además puedan entenderse en términos relativos los tiempos de ejecución se presentan tanto el hardware utilizado en las pruebas (Tabla 5.1) así como las versiones del software utilizado en la Tabla 5.2

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

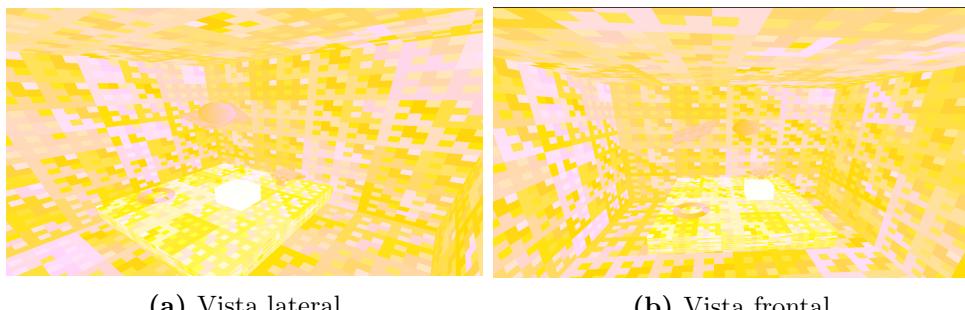
Tabla 5.1: Características del hardware utilizado

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) Vista lateral

(b) Vista frontal

Figura 5.1: Vistas de la escena *Conrnell Box*

Se denominará *Escena - Conrnell Box* a la mostrada en 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.

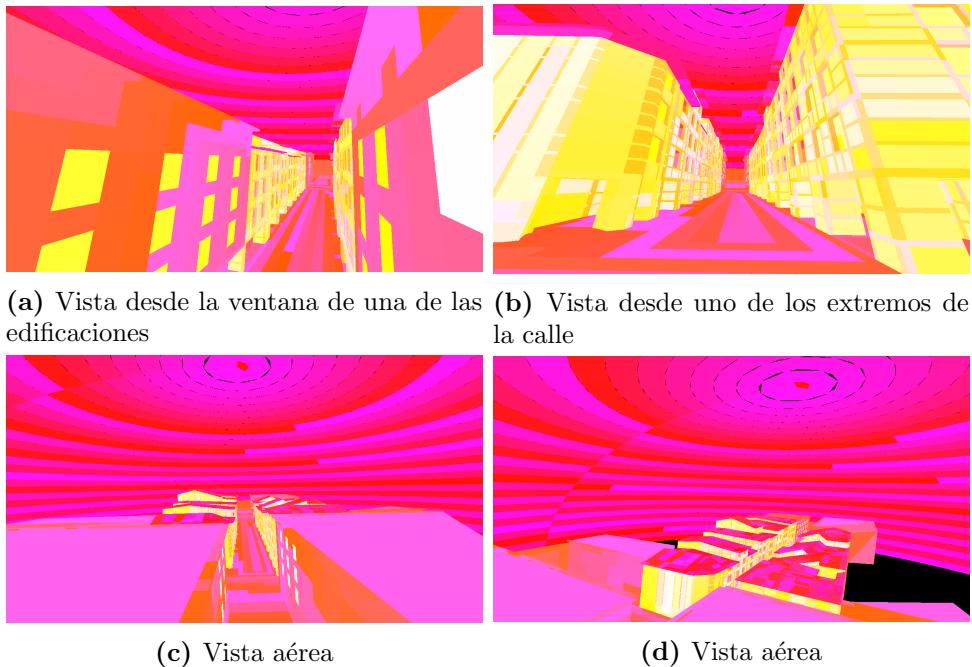


Figura 5.2: Vistas de la escena *Calle*

Se denominará *Escena - Calle* a la referente a la figura 5.2. La escena está constituida por dos objetos, el primero de ellos es una cúpula (hemiesfera) subdividida en 2.407 cuadriláteros de área equitativa, 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, Minion, et al. [13]] con el objetivo de estudiar el fenómeno de la radiación. En total, la escena cuenta con 61.795 caras cuadriláterales.

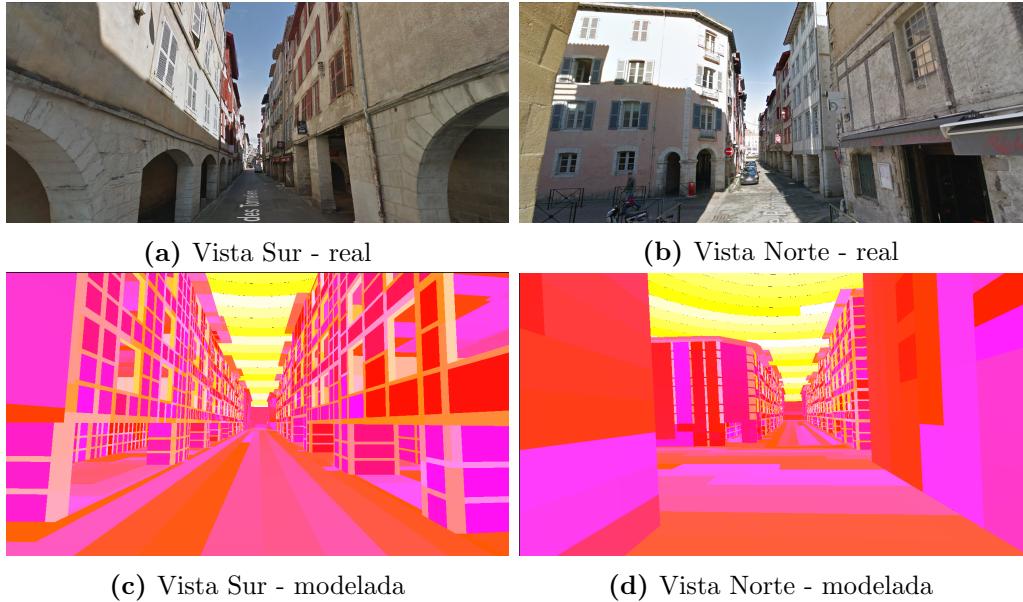


Figura 5.3: Comparación entre el fotografías reales del modelo *Calle* y su representación tridimensional

5.3. Casos de prueba

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

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 definirá un conjunto de métricas para evaluar su optimialidad en distintas dimensiones. Cada dimensión se aplicará dependiendo del caso considerado.

- Rendimiento
 - Tiempo de ejecución: Se registrará el tiempo empleado en calcular completamente la matriz de factores de forma.
- Matriz de factores de forma: Se comparará la matriz de control \mathbf{F}_C , calculada utilizando la técnica de trazado de rayos con una gran resolución.
 - Error promedio por fila: $Ep_i = \sum_{j=1}^N \frac{|\mathbf{F}_{C_{ij}} - \mathbf{F}_{ij}|}{N \mathbf{F}_{C_{ij}}}$
 - Error máximo por fila: $Em_i = \max_{j=1}^N \frac{|\mathbf{F}_{ij} - \mathbf{F}_{C_{ij}}|}{\mathbf{F}_{C_{ij}}}$

- Error estándar por fila: $Em_i = \max_{j=1}^N \frac{(\mathbf{F}_{ij} - \mathbf{F}_{C_{ij}})^2}{\mathbf{F}_{C_{ij}}}$
- Dimensión vector de radiosidad (dado el vector R , y el vector de control Rc):
 - Error promedio de radiosidad: $Ep = \sum_{i=1}^N \frac{|R_i - R_{Ci}|}{NR_{Ci}}$
 - Error máximo de radiosidad: $Em = \max_{j=1}^N |R_j - R_{Cj}|R_{Ci}$
 - Error estándar de radiosidad: $Em = \max_{j=1}^N (R_j - R_{Cj})^2$
- Visualización:
 - 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 invariantes. 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 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 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 desatascar 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 hemicubo o hemiesfera según corresponda. La cantidad de muestras tomadas depende de: $3*x^2$ donde x es el largo de un lado del hemicubo y en el caso de la hemiesfera el número dependerá de la cantidad de rayos trazados.

Caso de prueba I

Muestras	Tiempo de ejecución (s)			
	Cornell Box		Calle	
	OpenGL-D	Embree-D	OpenGL-D	Embree-D
3072	7	3	68	14
49512	10	30	128	174
196608	31	116	248	665
786432	251	446	1213	2565
3145728	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 5.3 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 %. 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 los núcleos del procesador 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

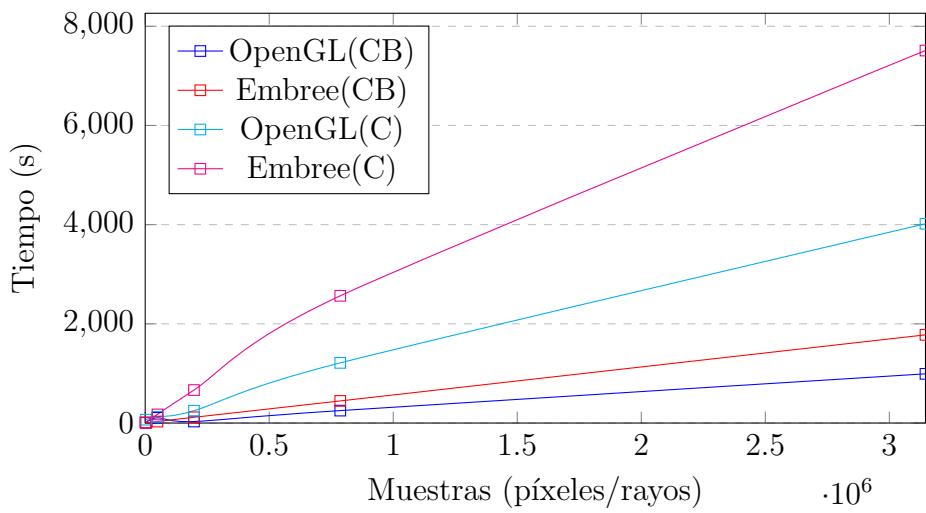


Figura 5.4: Comparación del rendimiento de los algoritmos en escenas exclusivamente difusas

métodos del hemicubo y la hemiesfera 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.

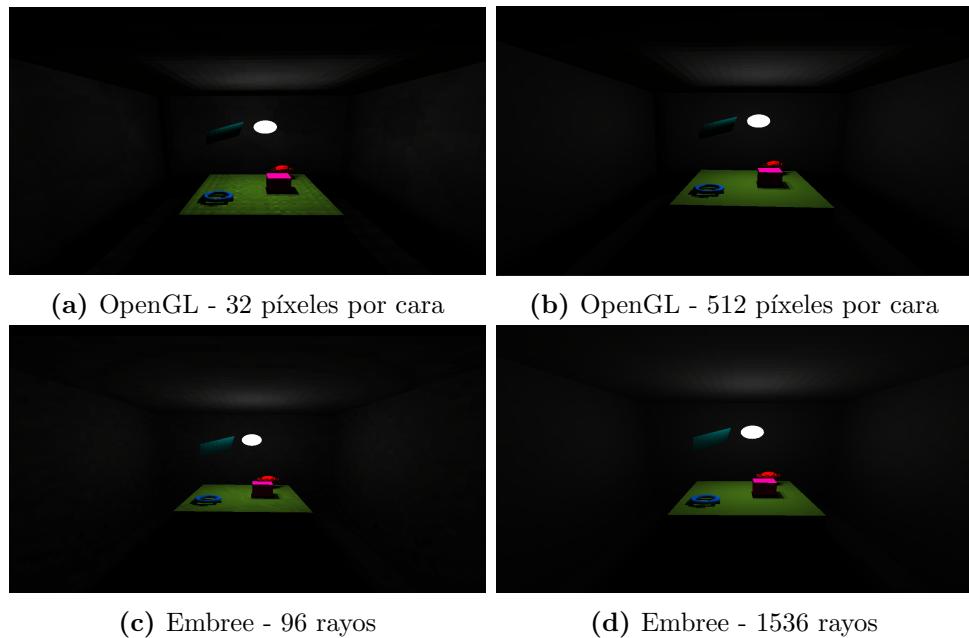


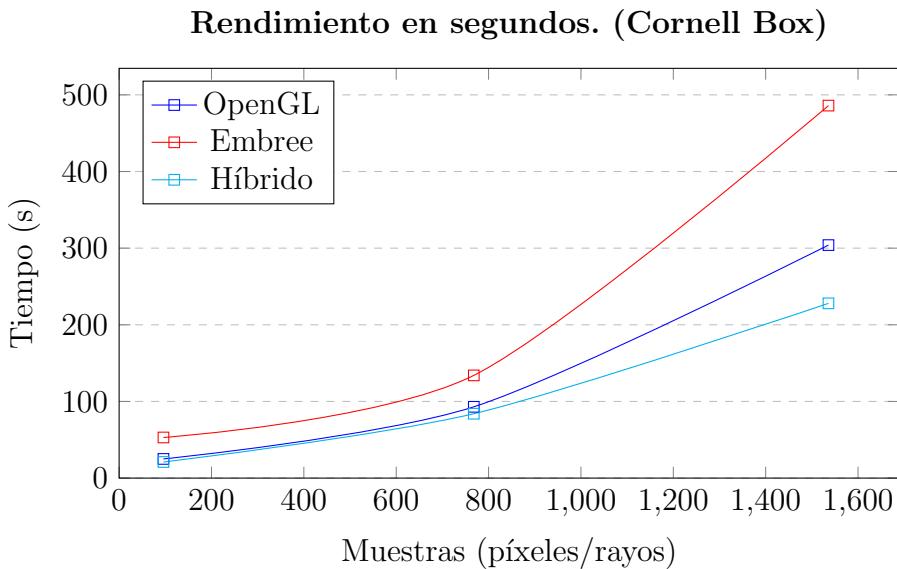
Figura 5.5: Diferencias visuales ajustando la cantidad de muestras

Caso de prueba II

En este caso, en cuanto a rendimiento, se observa en la tabla 5.4 que el mejor rendimiento observado 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 hemicubos. Esto se ve respaldado 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.

Muestras	Tiempo de ejecución (s)					
	Cornell Box			Calle		
	GL-D+S	E-D+S	Híb	GL-D+S	E-D+S	Híb
96 - 32	25	53	21	2563	1221	943
768 - 32	93	134	84	1054	2512	1643
1536 - 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; D+S indica que se utilizaron reflexiones especulares y difusas.



El paralelismo de las implementaciones basadas en la GPU garantiza el mejor rendimiento, sin embargo, se pudo notar que el uso exclusivo de traza de rayos provee hasta 14 veces menor error máximo que los otros algoritmos implementados (en *Cornell Box*, con 786432 muestras se notó una diferencia

de error máxima Ep de 0.0138 con el método híbrido y 0.0091 utilizando el método de dibujado de portales). 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 rendimiento un tanto menor (que se debe mayormente al hecho de que se ejecuta únicamente en la CPU) se observa una calidad de datos órdenes de magnitud superior a la de los otros métodos.

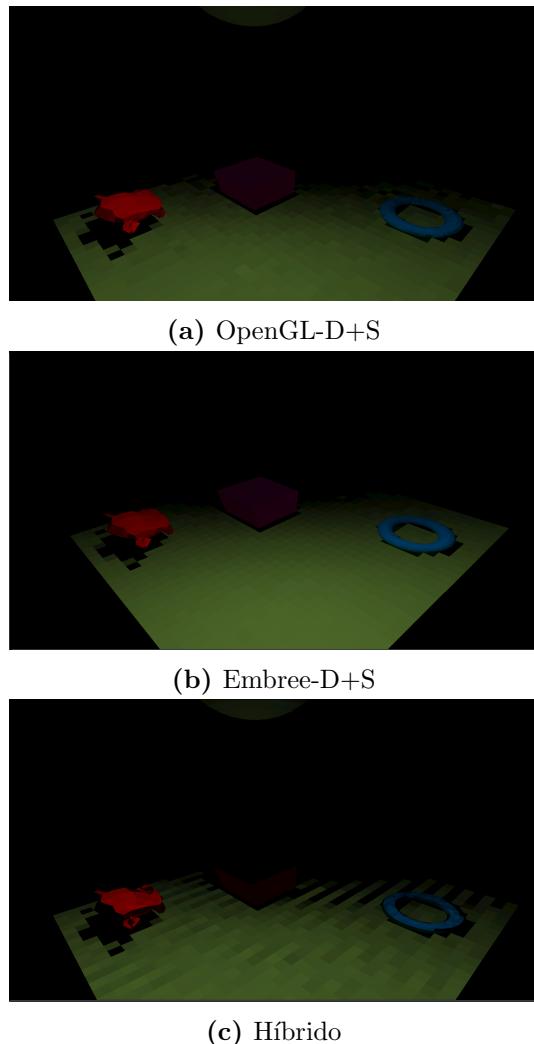


Figura 5.6: Diferencias visualizadas utilizando las distintas implementaciones de cálculo de factores de forma extendido. 1536 muestras iniciales y 64 para rebotes especulares.

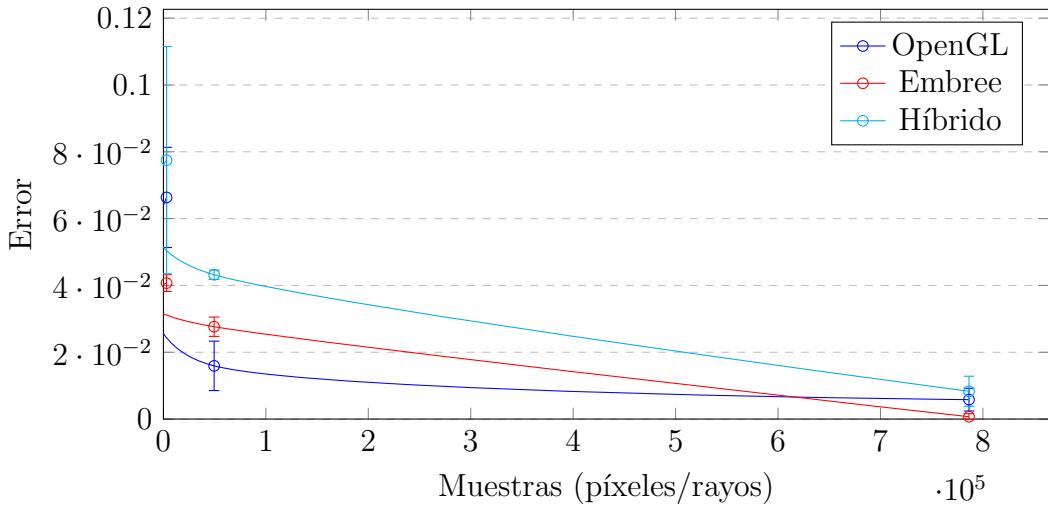


Figura 5.7: Error promedio observado en valor final de radiosidad en Híbrido

Con el objetivo de cuantificar los datos de error observado se analizó el error estándar y promedio observado en el vector de radiosidad final (comparado a una muestra utilizando exclusivamente trazado de rayos con 3276 muestras para la hemiesfera) y se observó que, tal como se había supuesto, el método es significativamente menos propenso a generar errores como se ve en la Figura 5.6.

Caso de prueba III

Esta prueba se construyó con el objetivo de comparar el rendimiento observado utilizando los algoritmos de extensión de factores de forma implementados. 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 (196608 para el hemicubo o rayos y 4096 para el portal).

Tabla 5.5: Pruebas realizadas en *Cornell Box* para identificar incidencia en la cantidad de caras especulares utilizadas en el tiempo de ejecución

Parches especulares	OpenGL	Embree	Híbrido
0	38	126	-
16	41	118	39
32	48	120	40
62	68	121	43

En table 5.5 puede observarse cómo la traza de rayos supera en dos veces el

tiempo a los otros algoritmos. Sin embargo, se ha de destacar (al igual que en las pruebas anteriores) que los resultados observados difieren completamente si se utilizaran resoluciones mayores para el dibujado de portales los tiempos serían más afines, aunque se conseguirían resultados peores. Esto se demuestra al emparejar la cantidad de muestras tomadas por el portal con las de cada cara del hemicubo (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.

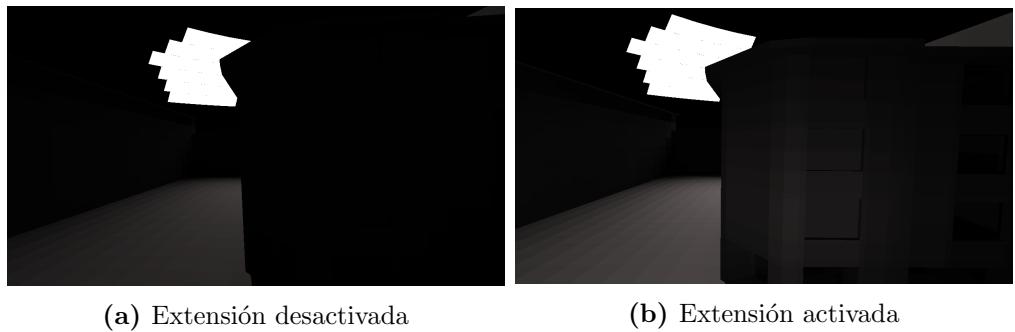


Figura 5.8: Diferencias observadas activando y desactivando extensiones

Adicionalmente, como se puede observar en la figura 5.8, 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 negligible en el tiempo de ejecución. En esta prueba en particular se notó un aumento de aproximadamente 20 segundos de 670 segundos considerando únicamente la iluminación difusa a 690 segundos activando la extensión utilizando trazado de rayos.

Caso de prueba IV

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 para considerar este tipo de superficie, los resultados pueden observarse en la figura 5.9 con una diferencia máxima entre 0 y 1500 superficies especulares de 3 % en el tiempo de cálculo de los factores de forma. 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.

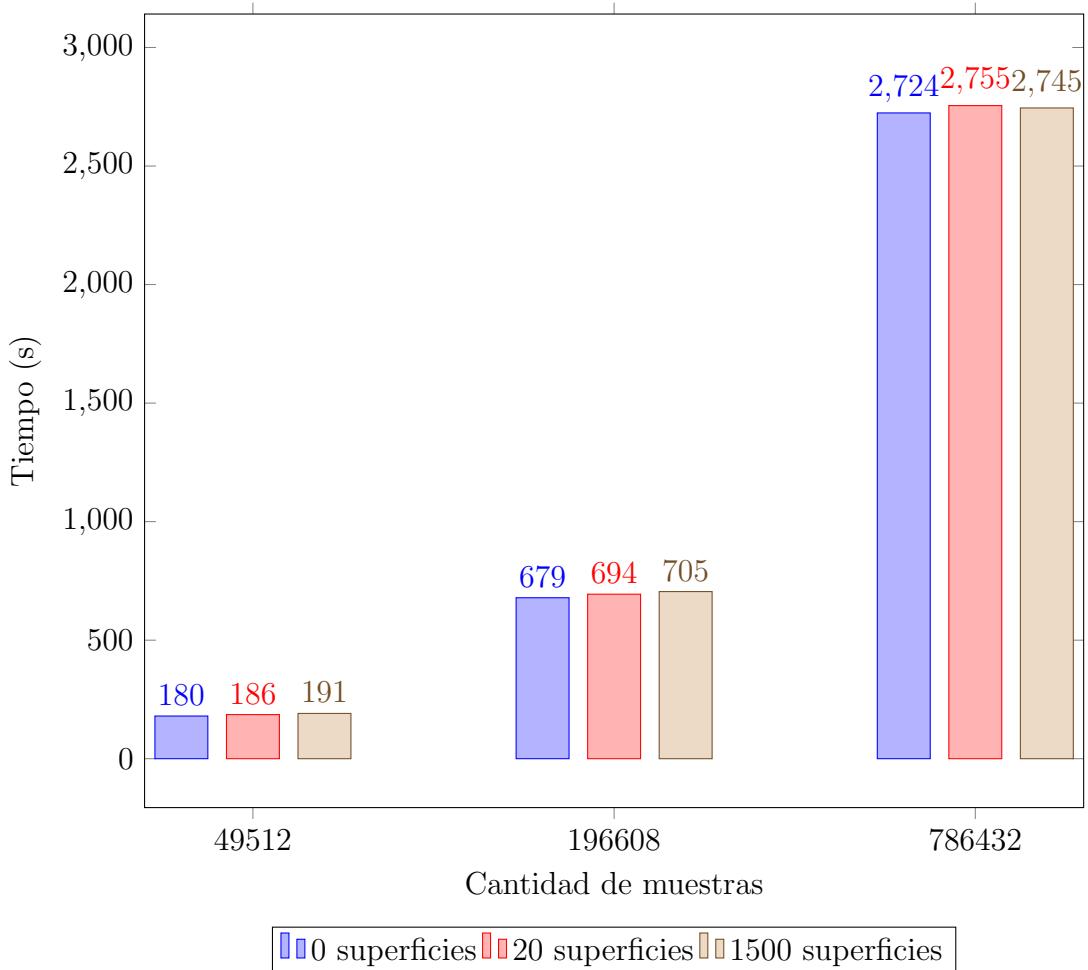


Figura 5.9: Variación en el tiempo de ejecución en función de la cantidad de muestras y superficies especulares

Capítulo 6

Conclusiones y trabajo futuro

Este capítulo presenta 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 para extender los algoritmos implementados.

6.1. Conclusiones

Este proyecto se dedicó al estudio, adaptación, implementación, extensión y comparación de distintos modelos de iluminación global que han surgido durante los años basados 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 hemicubo comúnmente usado en la técnica no presenta buenas características para ser integrado en el 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 de su uso utilizando bibliotecas de manejo 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 re-utilización de componentes y extensión de los métodos. Como por ejemplo, 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 [14]]. 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, además provee facilidades que permiten enviar comandos de dibujado desde distintos hilos, lo que disminuiría 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 aunque provee de funcionalidades que calculan la iluminación global. Además, como se aprecia en la Figura 6.1 el uso de texturas proporciona detalles visuales que enriquecen la imagen generada, aunque la técnica de radiosidad es utilizada para el cálculo de la iluminación, es decir, es una de las etapas necesarias para generar imágenes fotorealistas.

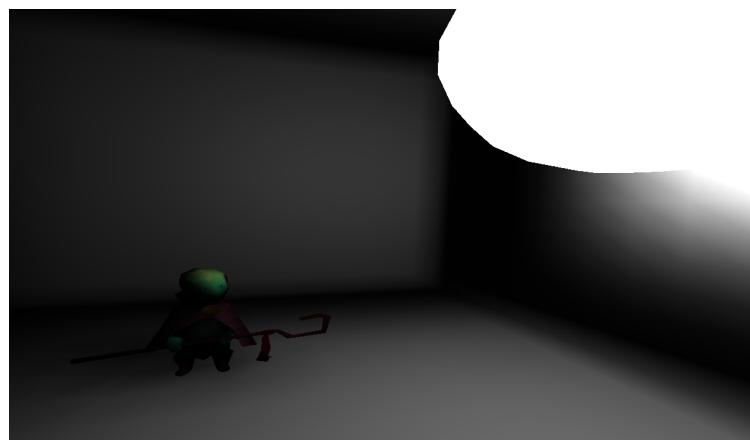


Figura 6.1: Ejemplificación de la adición de texturas a los objetos de la escena

Finalmente, el desarrollo de nuevo *hardware* permite la que 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.

Lista de figuras

2.1	Dibujado utilizando distintos modelos de iluminación.	6
2.2	Reflector lambertiano	8
2.3	El factor de forma entre dos superficies	10
2.4	La analogía de Nusslet	11
2.5	El <i>rendering pipeline</i> de OpenGL	14
2.6	Representación gráfica del método del hemicubo	16
2.7	Representación gráfica de los ejes considerados para el factor de corrección de los factores de forma	16
2.8	Representación gráfica del método de trazado de rayos para el cálculo de factores de forma	18
2.9	Representación gráfica del cálculo del factor de forma extendido donde $k = \frac{1}{N}$, con N muestras tomadas.	19
2.10	Vista general de la arquitectura de OpenGL	22
2.11	Vista general de la arquitectura de Embree	23
3.1	Tabla de Kanban utilizada en el proyecto	28
3.2	Módulo de manejo de geometría	29
3.3	Arquitectura del módulo de pre-procesado	30
3.4	Arquitectura del módulo de visualización	31
3.5	Arquitectura general del módulo de interfaz de usuario	32
4.1	Costo de cambios de estado en OpenGL referencia [12]	36
4.2	Generación del volumen de vista para espejos	39
4.3	Visualización del rebote de rayos al impactar en parches especulares	41
4.4	Ejemplificación del algoritmo de interpolación	44
4.5	Dibujado utilizando distintas funciones de interpolación	44
4.6	Vistas seleccionables por el usuario a través de la interfaz gráfica	46

4.7	Interfaz gráfica implementada	47
5.1	Vistas de la escena <i>Conrnell Box</i>	50
5.2	Vistas de la escena <i>Calle</i>	51
5.3	Comparación entre el fotografías reales del modelo <i>Calle</i> y su representación tridimensional	52
5.4	Comparación del rendimiento de los algoritmos en escenas exclusivamente difusas	55
5.5	Diferencias visuales ajustando la cantidad de muestras	55
5.6	Diferencias visualizadas utilizando las distintas implementaciones de cálculo de factores de forma extendido. 1536 muestras iniciales y 64 para rebotes especulares.	57
5.7	Error promedio observado en valor final de radiosidad en Híbrido	58
5.8	Diferencias observadas activando y desactivando extensiones . .	59
5.9	Variación en el tiempo de ejecución en función de la cantidad de muestras y superficies especulares	60
6.1	Ejemplificación de la adición de texturas a los objetos de la escena	63

Lista de tablas

5.1	Características del hardware utilizado	49
5.2	Características del entorno de desarrollo utilizado	49
5.3	Resultados obtenidos para el primer caso de prueba en ambas escenas consideradas	54
5.4	Resultados obtenidos en el segundo caso de prueba. Notación: GL, E, Híb corresponden a OpenGL, Embree e Híbrido; D+S indica que se utilizaron reflexiones especulares y difusas.	56
5.5	Pruebas realizadas en <i>Cornell Box</i> para identificar incidencia en la cantidad de caras especulares utilizadas en el tiempo de ejecución	58

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] 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.
- [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] 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.
- [11] 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.
- [12] 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>.
- [13] 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.
- [14] Joel Vazquez and Pablo Guartes. Aceleracion del calculo de la matriz de factores de forma utilizando visibilidad jerarquica. 2017.