



Universidad Central de Venezuela  
Facultad de Ciencias  
Escuela de Computación  
Centro de Computación Gráfica

**Trazado de Conos para el Cálculo de Iluminación Global empleando  
Sombreado de Vóxeles**

Trabajo Especial de Grado presentado ante la Ilustre  
Universidad Central de Venezuela  
por el Br. José Gabriel Villegas Guedez  
para optar al título de Licenciado en Computación.

Tutor:  
Prof. Esmitt Ramírez

Caracas, Mayo del 2016

## Resumen

La iluminación de escenas es fundamental para la generación de imágenes de alta calidad, esta provee inmersión, sensación de profundidad y realismo. La producción de iluminación realista comprende, entre muchas otras cosas, la inclusión de iluminación indirecta para la composición de iluminación global.

Existen varios algoritmos para el cálculo de la iluminación global de forma analítica, sin embargo el costo computacional de estos es alto y dependiente de la complejidad de la escena. Esto hace estas soluciones poco flexibles o simplemente inadecuadas para aplicaciones en tiempo real, altamente interactivas o de considerable complejidad geométrica.

Con el incremento de la capacidad de cómputo en las unidades de procesamiento gráfico modernas también ha aumentado el interés por la inclusión de fenómenos de iluminación global en aplicaciones en tiempo real. De esto han surgido una variada cantidad de aproximaciones explotando características del hardware de procesamiento gráfico y los recursos disponibles en el pipeline de renderizado. Estos algoritmos mantienen cierto grado de coherencia o convergen con la solución analítica al problema de iluminación global.

Este trabajo tiene como enfoque principal el renderizado de iluminación global en tiempo real, por esto se realiza una investigación de estos algoritmos y se presenta el desarrollo de una aplicación con iluminación global utilizando véxeles y trazado de conos.

**Palabras Clave:** Iluminación global, véxeles, trazado de conos, iluminación indirecta, unidad de cómputo gráfico, renderizado en tiempo real.

## **Agradecimientos**

Al profesor Esmitt Ramírez, por su apoyo y dirección, sin el cual este trabajo de investigación se hubiese llevado a cabo.

A mi mamá, Marina Villegas, por su apoyo y soporte incondicional en todos los aspectos de mi vida y por ser mi mejor amiga.

A mi hermano menor Daniel De La Cruz, por su colaboración y ayuda para hacer este trabajo posible e inspirarme a ser un buen ejemplo para él.

A todos mis amigos por estar siempre atentos a este trabajo y su progreso el cual me mantuvo enfocado durante la realización de este trabajo. Y además por ser los causantes de múltiples distracciones cuando las necesitaba y no necesitaba.

# Índice general

<b>Introducción</b>	<b>VII</b>
<b>1. Marco Teórico</b>	<b>1</b>
1.1. Iluminación Global . . . . .	1
1.2. Radiometría . . . . .	2
1.2.1. Unidades en Radiometría . . . . .	2
1.2.1.1. Flujo Radiante . . . . .	2
1.2.1.2. Irradiancia . . . . .	2
1.2.1.3. Emitancia Radiante o Radiosidad . . . . .	2
1.2.1.4. Radiancia . . . . .	3
1.3. Iluminación Directa e Indirecta . . . . .	3
1.4. Representación de Superficies . . . . .	4
1.4.1. Función de Distribución de Reflectancia Bidireccional . . . . .	4
1.4.1.1. Propiedades de la Función BRDF . . . . .	5
1.4.1.2. Ejemplos de BRDF . . . . .	5
1.4.1.3. Modelos de Sombreado. . . . .	7
1.4.2. Función de Distribución Normal . . . . .	8
1.5. Ecuación de Renderizado . . . . .	10
1.5.1. Formulación Hemisférica . . . . .	10
1.5.2. Procedimientos . . . . .	11
1.5.2.1. Radiosidad . . . . .	11
1.5.2.2. Trazado de Rayos e Integración Monte Carlo . . . . .	12
1.6. Pipeline de Renderizado Programable . . . . .	13
1.6.1. Procesador de Vértices . . . . .	13
1.6.2. Procesador de Geometría . . . . .	14
1.6.3. Rasterización . . . . .	14
1.6.4. Procesador de Fragmentos . . . . .	14

1.6.5. Cómputo de Propósito General en la GPU . . . . .	15
1.7. Técnicas en Renderizado de Imágenes . . . . .	15
1.7.1. Mapeado de Sombras . . . . .	15
1.7.2. Sombreado Diferido . . . . .	16
1.7.3. Voxelización . . . . .	17
1.8. Iluminación Global en Tiempo Real. . . . .	18
1.8.1. Luces Puntuales Virtuales . . . . .	18
1.8.2. Mapas de Sombras Reflexivo . . . . .	19
1.8.3. Volúmenes de Propagación de Luz en Cascada . . . . .	20
1.8.4. Iluminación Indirecta con Trazado de Conos y Vóxeles . . . . .	22
1.8.4.1. Construcción del Octree de Vóxeles . . . . .	23
1.8.4.2. Contenido de un Vóxel . . . . .	23
1.8.4.3. Filtrado en Niveles de Detalle . . . . .	24
1.8.4.4. Trazado de Conos y Vóxeles . . . . .	24
1.8.4.5. Filtrado Anisotrópico de Vóxeles. . . . .	25
1.8.4.6. Captura de Iluminación Directa en Vóxeles . . . . .	26
<b>2. Solución Propuesta</b>	<b>27</b>
2.1. Voxelización . . . . .	28
2.1.1. Voxelización Conservativa . . . . .	28
2.1.2. Composición de Fragmentos y Vóxeles . . . . .	29
2.1.3. Voxelización Dinámica . . . . .	30
2.2. Sombreado de Vóxeles . . . . .	31
2.2.1. Trazado y Mapeo de Sombras sobre el Volumen . . . . .	32
2.2.1.1. Trazado de Sombras Suaves sobre el Volumen . . . . .	33
2.3. Estructura Jerárquica . . . . .	34
2.3.1. Filtrado Anisotrópico de Vóxeles . . . . .	34
2.4. Trazado de Conos con Vóxeles . . . . .	35
2.4.1. Reflexión Difusa . . . . .	35
2.4.2. Oclusión Ambiental . . . . .	36
2.4.3. Reflexión Especular . . . . .	36
2.4.4. Sombras Suaves . . . . .	37
2.5. Iluminación Global de Vóxeles . . . . .	37
2.6. Materiales Emisivos . . . . .	38
<b>3. Implementación</b>	<b>39</b>
3.1. Herramientas . . . . .	39

3.2. Arquitectura de la Aplicación . . . . .	40
3.3. Pipeline de Voxelización . . . . .	44
3.3.1. Arquitectura . . . . .	44
3.3.2. Voxelización Conservativa . . . . .	45
3.3.2.1. Matrices de Proyección por Eje . . . . .	45
3.3.2.2. Selección del Eje Dominante . . . . .	46
3.3.2.3. Extensión del Triángulo y Polígono Delimitante . . . . .	47
3.3.2.4. Composición de Fragmentos y Vóxeles . . . . .	49
3.3.2.5. Bandera Estática . . . . .	52
3.3.3. Re-voxelización y Limpieza de Volúmenes . . . . .	53
3.4. Sombreado de Vóxeles . . . . .	54
3.4.1. Mapeo y Trazado de Sombras . . . . .	56
3.4.2. Vóxeles Emisivos . . . . .	59
3.5. Estructura Jerárquica . . . . .	59
3.5.1. Vóxeles Anisótropicos . . . . .	60
3.6. Trazado de Conos con Vóxeles . . . . .	64
3.6.1. Reflexión Difusa . . . . .	68
3.6.2. Reflexión Especular . . . . .	69
3.6.3. Oclusión Ambiental . . . . .	70
3.6.4. Sombras Suaves con Trazado de Conos . . . . .	71
3.6.5. Composición Final . . . . .	72
3.7. Iluminación Global de Vóxeles . . . . .	73
<b>4. Pruebas y Resultados</b>	<b>75</b>
4.1. Entorno de Pruebas . . . . .	75
4.1.1. Configuración de la Aplicación . . . . .	75
4.2. Escenarios de Estudio . . . . .	76
4.2.1. Escenas de Prueba y Objetos . . . . .	76
4.2.1.1. Escenas Completas . . . . .	76
4.2.1.2. Escenas Sandbox . . . . .	79
4.2.1.3. Objetos Precargados . . . . .	79
4.2.1.4. Criterios de Complejidad Geométrica e Iluminación . . . . .	80
4.3. Estudio de Rendimiento . . . . .	81
4.3.1. Prueba Base . . . . .	81
4.3.1.1. Densidad Geométrica y Velocidad de Voxelización . . . . .	83

4.3.1.2. Vacuidad y Velocidad de Trazado para la Iluminación Global de Vóxeles . . . . .	84
4.3.2. Trazado de Sombras y Volumen de Visibilidad . . . . .	85
4.3.3. Apertura del Cono Especular y Cono de Sombras . . . . .	88
4.3.4. Comparaciones . . . . .	88
4.4. Estudio de Calidad de Imagen . . . . .	89
4.4.1. Composición Final de Imagen . . . . .	89
4.4.1.1. Iluminación Global de Vóxeles. . . . .	92
4.4.1.2. Resolución de la Representación en Vóxeles . . . . .	93
4.4.1.3. Factor de Longitud de Marcha del Cono . . . . .	97
4.4.2. Reflexión Especular y Factor de Longitud de Marcha . . . . .	100
4.4.3. Apertura del Cono para Trazado de Sombras Suaves . . . . .	102
4.4.3.1. Mapeo del Volumen de Visibilidad . . . . .	102
4.4.4. Materiales Emisivos . . . . .	104
4.4.5. Defectos o Artefactos Visuales . . . . .	106
4.4.6. Comparación . . . . .	107
4.5. Estudio de Memoria . . . . .	108
<b>5. Conclusiones</b>	<b>109</b>
5.1. Trabajos Futuros . . . . .	110
<b>Bibliografía</b>	<b>112</b>

# Introducción

La síntesis de imágenes realistas siempre ha sido un objetivo de la computación gráfica. La iluminación de escenas es uno de los aspectos importantes para este objetivo. El cálculo preciso de iluminación es un proceso complejo ya que la luz no solo sale de un punto y llega a otro, esta se propaga, rebota y es absorbida por distintos elementos en la escena.

En el pipeline de renderizado estándar se utilizan triángulos los cuales son rasterizados para generar fragmentos que luego son coloreados. La representación de superficies en forma de triángulos es efectiva para cálculos como iluminación directa, sin embargo esta posee grandes limitaciones para incorporar fenómenos de iluminación más complejos como iluminación global.

Técnicas recientes han surgido donde se utiliza una representación de la escena en véxeles para simplificar la geometría en escena y hacer posible el cálculo de iluminación global en tiempo real. Un véxel representa un elemento volumétrico en una cuadrícula tridimensional uniforme, es también referido como píxel volumétrico o tridimensional.

Una escena representada en véxeles permite evitar los problemas de complejidad geométrica y su relación con el rendimiento de varias soluciones al problema de iluminación. Esta representación además puede ser filtrada a menores niveles de detalle bajo un esquema de *mipmapping*.

Estas propiedades son especialmente útiles para el algoritmo de trazado de conos. La complejidad del cálculo de colisión entre geometría poligonal y conos se reduce al aproximar esta al muestreo de un volumen donde según la apertura del cono se utiliza algún nivel de detalle en la estructura de véxeles.

# Objetivos

## Objetivo General

Desarrollar un método para el cómputo de iluminación global en tiempo real basado en trazado de conos y véxeles en escenas interactivas.

## Objetivos Específicos

- Realizar el proceso de voxelización de la geometría en escena.
- Implementar la técnica de *deferred shading*.
- Implementar una estructura de véxeles para representar una simplificación de la escena.
- Construir una descripción jerárquica para esta estructura de véxeles para almacenar distintos niveles de detalle.
- Implementar la actualización dinámica de esta estructura al detectar cualquier cambio relevante en escena.
- Diseñar la representación interna de cada voxel.
- Implementar el trazado aproximado de conos contra véxeles.
- Garantizar fenómenos de iluminación global como oclusión ambiental e iluminación indirecta.
- Implementar sombreado de véxeles para su uso en trazado de conos.
- Incluir oclusión con sombras para el sombreado de véxeles.
- Generar sombras suaves utilizando trazado de conos.
- Aproximar materiales emisivos con la inclusión de emisión durante el sombreado de véxeles.
- Aproximar iluminación indirecta de uno y dos rebotes.
- Realizar pruebas de rendimiento y precisión sobre diversas escenas y resoluciones.

# Capítulo 1

## Marco Teórico

### 1.1. Iluminación Global

Iluminación global se le llama al proceso de calcular la distribución de la energía de la luz sobre escenas tridimensionales desplegadas por computadora. Los efectos de la iluminación global incluyen suave sombreado debajo de objetos y cerca de esquinas, rebotes de luz, mezcla de colores, cáusticas, transluminiscencia, entre otros. Estos efectos son muy sutiles pero afectan el realismo de la imagen final de forma substancial [1]. La Figura 1 muestra algunos de estos efectos.

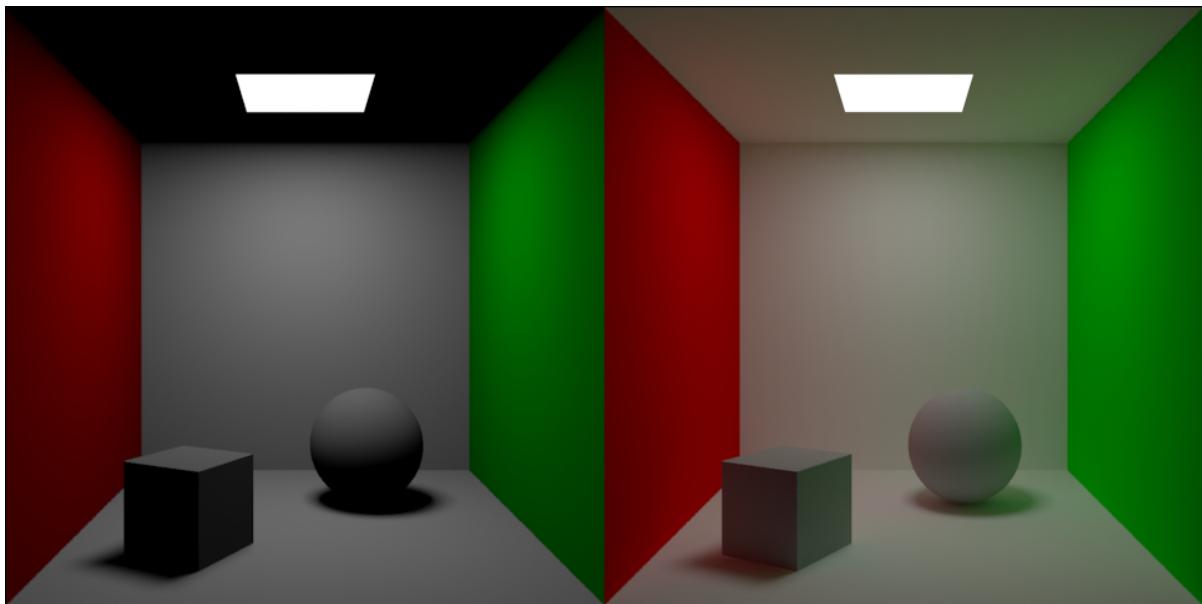


Figura 1: Ejemplo de una escena con solo iluminación directa (izquierda). Iluminación global (derecha)

El cómputo preciso y completo de iluminación global es considerado poco flexible, costoso y lento para ser utilizable en ciertos medios visuales que requieren producción de imágenes en tiempos interactivos o en escenas de gran complejidad, por ejemplo videojuegos o simulaciones. Es por ello que el desarrollo de aproximaciones y algoritmos para simular iluminación global con mejores factores de rendimiento y flexibilidad es un constante tema de investigación.

## 1.2. Radiometría

Radiometría es el campo dedicado al estudio y medición de la radiación electromagnética. Para el cómputo de la distribución de la luz es necesario entender algunas unidades importantes [2].

### 1.2.1. Unidades en Radiometría

#### 1.2.1.1. Flujo Radiante

La unidad fundamental en radiometría, usualmente denotada como  $\Phi$  es expresada en *vatios* o *watts*. Esta cantidad expresa cuanta energía total fluye desde, hasta y a través de una superficie por unidad de tiempo.

#### 1.2.1.2. Irradiancia

La irradiancia  $E$  es el flujo radiante entrante o incidente por unidad sobre el área de una superficie. Esta es expresada en *vatios/m<sup>2</sup>*:

$$E = \frac{d\Phi}{dA} \quad (1.1)$$

#### 1.2.1.3. Emitancia Radiante o Radiosidad

La emitancia radiante  $M$  es el flujo radiante saliente o emitido por unidad sobre el área de una superficie. También es expresada en *vatios/m<sup>2</sup>*:

$$M = \frac{d\Phi}{dA} \quad (1.2)$$

#### 1.2.1.4. Radiancia

La radiancia  $L$  es el flujo radiante emitido por unidad de ángulo sólido y por unidad de área proyectada, expresada en  $\text{vatos/estereorradián} \cdot \text{m}^2$ . De forma intuitiva la radiancia expresa cuanta potencia llega (o sale) de un punto  $x$  por unidad de ángulo sólido y por unidad de área proyectada, en la Figura 2 se ilustra la definición de radiancia. La radiancia varía con la posición  $x$  y el vector dirección  $\Theta$ , es expresada como  $L(x, \Theta)$ :

$$L = \frac{d^2\Phi}{dwdA^\perp} = \frac{d^2\Phi}{dwA \cos \theta} \quad (1.3)$$

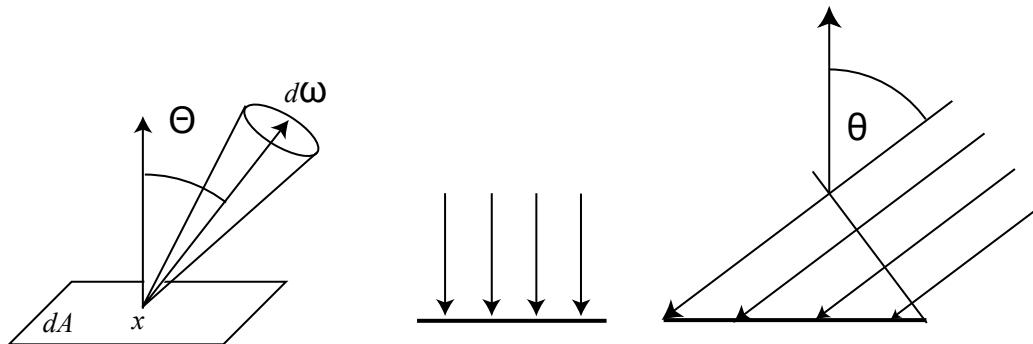


Figura 2: Definición de radiancia  $L(x, \Theta)$ . Flujo radiante emitido por unidad de ángulo sólido  $dw$  y por unidad de área proyectada  $A^\perp$ .

La radiancia es el término más importante para los propósitos de este trabajo e igualmente para una cantidad importante de algoritmos y aproximaciones para el cálculo de iluminación global. Esta unidad es la que captura la apariencia de los objetos en la escena.

### 1.3. Iluminación Directa e Indirecta

La iluminación directa es aquella que se proyecta sobre la superficie de algún objeto directamente desde las fuentes de luz en la escena. Iluminación indirecta es la luz que proviene de los subsecuentes rebotes de luz originados desde las superficies iluminadas, ya sea esta superficie reflectiva o no [2]. La composición de ambas resulta en iluminación global.

## 1.4. Representación de Superficies

Los materiales interactúan con la luz de distintas maneras. Esto hace que la apariencia de ciertos materiales difiera según las condiciones de la luz en una escena. Algunos materiales parecen espejos mientras que otros son totalmente difusos. Son visualmente distinguible materiales como vidrio, madera o metales. Las propiedades de reflectancia de una superficie afectan la apariencia del objeto [2], estos objetos se distinguen por la cantidad de luz reflectada en ciertas direcciones.

En el caso más general, un rayo de luz entra en algún punto  $p$  sobre una superficie en una escena, este rayo de luz tiene una dirección incidente  $\Psi$  y puede salir de esta superficie sobre otro punto  $q$  con dirección saliente  $\Theta$ . La función que define esta relación entre la radiancia incidente y la radiancia reflectada se llama función de distribución de reflectancia y transluminiscencia bidireccional (BSSRDF).

### 1.4.1. Función de Distribución de Reflectancia Bidireccional

Al asumir que la luz incidente en algún punto  $x$  sale del mismo punto  $x$  (ignorando la transluminiscencia) las propiedades de reflectancia de una superficie son entonces descritas por una función de distribución de reflectancia bidireccional (BRDF). En la Figura 3 se puede observar la diferencia entre BRDF y BSSRDF.

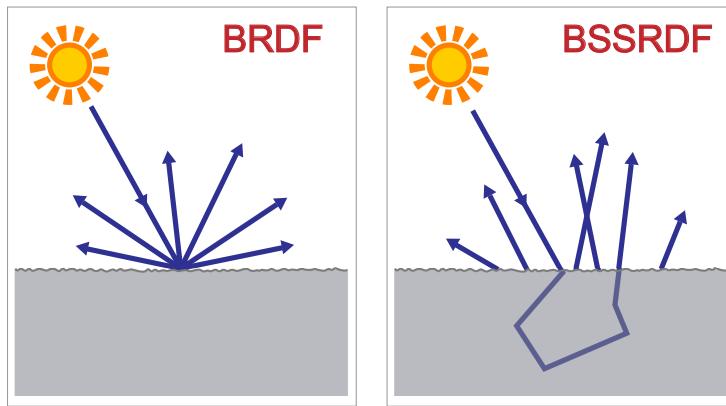


Figura 3: Izquierda: BRDF, punto de salida es igual al punto de entrada. Derecha: BSSRDF, se observa como el punto de salida es distinto al de entrada.

La BRDF en un punto  $x$  se define entonces como la distribución de la radiancia reflectada diferencial en una dirección saliente  $\Theta$  y la irradiancia incidente diferencial a través de un

ángulo sólido  $d\omega_\Psi$ . La función BRDF puede ser escrita de la siguiente forma [2]:

$$\begin{aligned} f_r(x, \Psi \rightarrow \Theta) &= \frac{dL(x \rightarrow \Theta)}{dE(x \leftarrow \Psi)} \\ &= \frac{dL(x \rightarrow \Theta)}{L(x \leftarrow \Psi) \cos(N_x, \Psi) d\omega_\Psi} \end{aligned} \quad (1.4)$$

donde  $\cos(N_x, \Psi)$  es el coseno del ángulo formado entre el vector normal en el punto  $x$  y el vector dirección  $\Psi$ , también llamado atenuación normal.

#### 1.4.1.1. Propiedades de la Función BRDF

La función BRDF tiene una variada cantidad de importantes propiedades:

1. Dimensión: La función BRDF es una función cuatridimensional definida sobre cada punto de una superficie, dos dimensiones corresponden a la dirección entrante y dos a la dirección saliente.
2. Reciprocidad: El resultado de la función BRDF es el mismo si se intercambian la dirección entrante y la dirección saliente:

$$f_r(x, \Psi \rightarrow \Theta) = f_r(x, \Theta \rightarrow \Psi) \quad (1.5)$$

3. Conservación de la energía: La ley de conservación de la energía dicta que la cantidad total de energía reflectada en todas las direcciones debe ser menor o igual a la cantidad de total de energía incidente sobre las superficies:

$$\int_{\Omega^+} f_r(x, \Psi \rightarrow \Theta) \cos(N_x, \Theta) d\omega_\Theta \leq 1 \quad (1.6)$$

#### 1.4.1.2. Ejemplos de BRDF

Dependiendo del comportamiento de la BRDF, el material se verá como una superficie difusa, como un espejo o como una mezcla de ambos (lustroso o *glossy*). En la Figura 4 se ilustra estas propiedades. Los tipos de BRDF relevantes para este trabajo serán listados aquí.

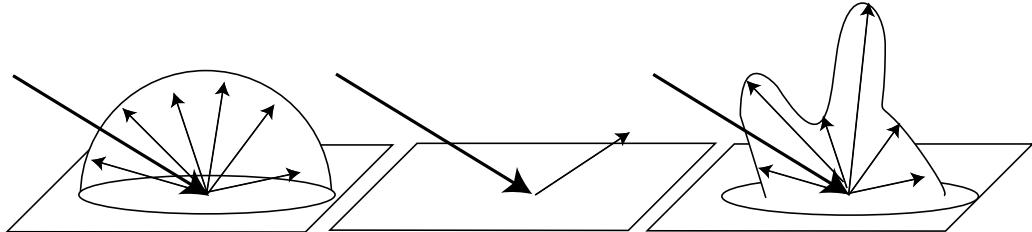


Figura 4: Ejemplos de superficies: totalmente difusa izquierda, totalmente especular centro, lustrosa (glossy) derecha [2].

**Superficies Difusas:** Algunos materiales reflectan la luz de forma uniforme sobre la totalidad de la semiesfera de reflectancia. Esto quiere decir que según la distribución de la irradiancia, la radiancia reflectada es independiente de la dirección de salida. Estos materiales son llamados reflectores difusos y el valor de su BRDF es una constante para todos los valores  $\Theta$  y  $\Psi$ . Para un observador un punto sobre una superficie difusa se ve igual desde todas las direcciones [2]. Para una superficie difusa ideal, la reflexión difusa puede ser representada como:

$$f_r(x, \Psi \leftrightarrow \Theta) = \frac{\rho_d}{\pi} \quad (1.7)$$

La reflectancia  $\rho_d$  representa la fracción de la energía incidente que es reflectada en la superficie. Para materiales físicamente correctos,  $\rho_d$  varía entre 0 y 1.

**Superficies Especulares:** Superficies especulares perfectas solo reflejan o refractan luz en una dirección específica.

**Reflexión Especular:** La dirección de reflexión puede ser obtenida utilizando la ley de reflexión, esta indica que la dirección de la luz incidente y saliente tienen un ángulo equivalente con el vector normal de la superficie. Dado que la luz es incidente con respecto a la superficie con vector dirección  $\Psi$ , y el vector normal de la superficie es  $N$ , la luz incidente es reflectada en la dirección  $R$ :

$$R = 2(N \cdot \Psi)N - \Psi \quad (1.8)$$

Un reflector espectral perfecto tiene solo una dirección de salida donde la BRDF es diferente de 0, esto implica que el valor de la BRDF en esa dirección es infinito.

**Superficies Lustrosas:** La mayoría de las superficies no son ni idealmente difusas ni idealmente especulares, sino que demuestran una combinación de ambas características de reflectancia. Estas superficies son llamadas superficies lustrosas o superficies *glossy*. La BRDF que describe esta clase de materiales es usualmente difícil de modelar de forma analítica [2].

#### 1.4.1.3. Modelos de Sombreado.

Materiales reales pueden tener BRDFs particularmente complejas. En computación gráfica existen varios modelos que intentan capturar la complejidad de las BRDFs. En la siguiente sección se expande sobre ciertos modelos relevantes a este trabajo. Nótese que  $\Psi$  es la dirección de la luz (dirección incidente),  $\Theta$  es la dirección del observador (dirección saliente) y  $N$  el vector normal de la superficie.

**Modelo de Lambert:** Uno de los modelos más simples, este modelo es ideal para superficies difusas, en este modelo la BRDF es una constante como ya fue descrito anteriormente en la ecuación 1.7.

$$f_r(x, \Psi \leftrightarrow \Theta) = k_d = \frac{\rho_d}{\pi} \quad (1.9)$$

donde  $\rho_d$  es la reflexión difusa.

**Modelo de Phong:** La BRDF del modelo de Phong es:

$$f_r(x, \Psi \leftrightarrow \Theta) = k_s \frac{(R \cdot \Theta)^n}{N \cdot \Psi} + k_d \quad (1.10)$$

donde el vector reflectado  $R$  es calculado con la ecuación 1.8.

**Modelo de Blinn-Phong:** El modelo Blinn-Phong utiliza el vector medio  $H$  entre  $\Psi$  y  $\Theta$  de la siguiente manera:

$$f_r(x, \Psi \leftrightarrow \Theta) = k_s \frac{(N \cdot H)^n}{N \cdot \Psi} + k_d \quad (1.11)$$

El valor  $n$  varía según las propiedades del material. Un mayor valor provee un lóbulo especular más pequeño, simulando superficies lisas y pulidas. En la Figura 5 se puede observar este efecto, y en la Figura 6 se puede observar el resultado de este modelo.

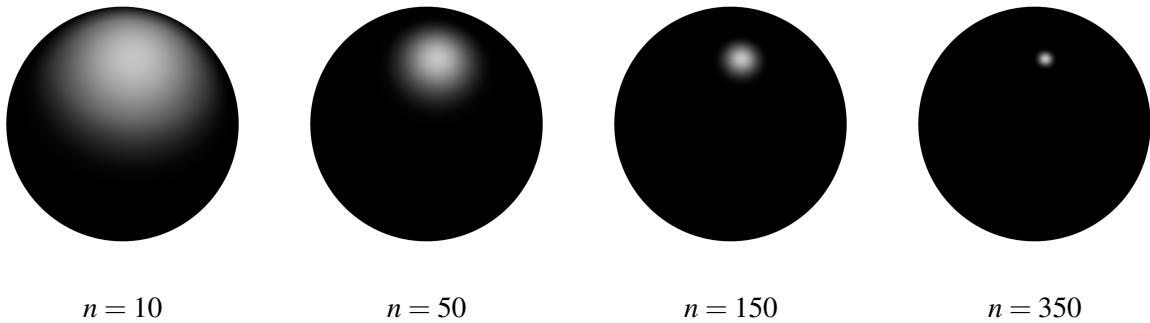


Figura 5: Distribución especular para distintos valores de  $n$  en el modelo Blinn-Phong.

**Modelo de Blinn-Phong Modificado:** El modelo de Phong a pesar de ser simple este tiene ciertas limitaciones, no es ni conservador de energía ni recíproco y tiene problemas para simular materiales reales. El modelo modificado toma en cuenta algunos de estos problemas:

$$f_r(x, \Psi \leftrightarrow \Theta) = k_s(N \cdot H)^n + k_d \quad (1.12)$$

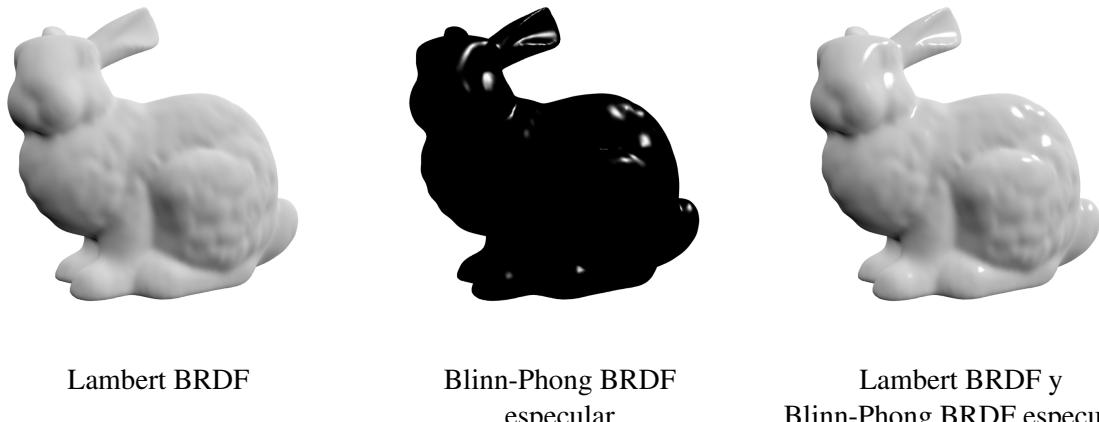


Figura 6: Composición de Blinn-Phong y Lambert. Se observa como estas afectan la apariencia final de la superficie.

#### 1.4.2. Función de Distribución Normal

La función de distribución normal (NDF) introducida por Alain Fournier [3] describe la densidad de las normales como una función de dirección. Funciones gaussianas como la

siguiente es una de las posibles formas de una NDF.

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad (1.13)$$

En la función gaussiana el término  $\sigma^2$  es llamado varianza y el término  $\mu$  es llamado media. Tanto el producto como la convolución de dos distribuciones gaussianas es también una distribución gaussiana [4].

La función gaussiana puede ser utilizada como una representación direccional. En este caso la media es un vector promedio  $D$  del lóbulo gaussiano. Como es descrito en el trabajo de Toksvig para el *mipmapping* de mapas de normales [5], el valor de  $\sigma$  puede ser calculado con la longitud del vector promedio  $D$  utilizando la siguiente ecuación:

$$\sigma^2 = \frac{1 - |D|}{|D|} \quad (1.14)$$

Esto permite obtener lóbulos gaussianos a partir de dos o más vectores para representar su dirección en común como se observa en la Figura 7.

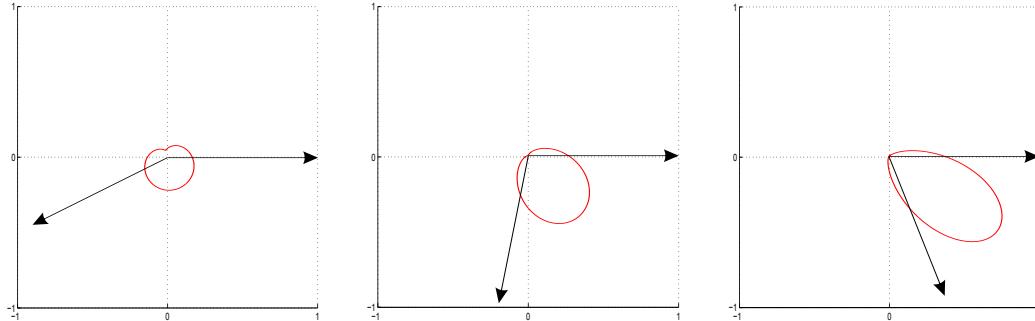


Figura 7: Ejemplo de lóbulos gaussianos según dos vectores, se puede observar como la forma del lóbulo cambia según la dirección promedio.

Algunas BRDF también pueden describirse como distribuciones gaussianas, las propiedades de producto y convolución se mantienen. Esto se puede observar en la Figura 8.



Figura 8: Lóbulos especulares para BRDF Phong y Blinn-Phong para un  $\Theta$  y  $\Psi$  de  $\angle 45$  grados, se puede observar cómo estas describen la distribución de la dirección de reflectancia. Como se explica en 1.4.1.3 el valor de  $n$  afecta la forma del lóbulo mientras mayor es este número más fino y largo es el lóbulo especular. Imágenes renderizadas en Disney's BRDF Explorer [6].

## 1.5. Ecuación de Renderizado

La ecuación de renderizado fue introducida por Kajiya en 1986 [7]. Esta ecuación describe en cada punto  $x$  de una superficie y en cada dirección  $\Theta$ , la radiancia saliente  $L(x \rightarrow \Theta)$  en ese punto y esa dirección.

El objetivo de un algoritmo para el cálculo de iluminación global es aproximar el resultado de esta ecuación. En esta ecuación se asume que no existen medios participantes como objetos translúcidos como ya fue explicado en la sección 1.4. También se asume que la luz se propaga de forma inmediata por tanto la distribución de la luz, ya en un estado estacionario, se obtiene inmediatamente.

### 1.5.1. Formulación Hemisférica

La formulación hemisférica de la ecuación de renderizado es una de las más utilizadas [2]. Esta formulación se obtiene utilizando la propiedad de conservación de energía en el punto  $x$ . Asumiendo que  $L_e(x \rightarrow \Theta)$  representa la radiancia emitida por la superficie en el punto  $x$  con dirección saliente  $\Theta$  y  $L_r(x \rightarrow \Theta)$  representa la radiancia reflectada por la superficie en el punto  $x$  en dirección  $\Theta$ .

Por conservación de energía, el total de la radiancia saliente en un punto y dirección particular es la suma de la radiancia emitida y la radiancia reflectada en este punto de la su-

perficie y dirección. La radiancia saliente  $L(x \rightarrow \Theta)$  es expresada en términos de  $L_e(x \rightarrow \Theta)$  y  $L_r(x \rightarrow \Theta)$  de la siguiente forma:

$$L(x \rightarrow \Theta) = L_e(x \rightarrow \Theta) + L_r(x \rightarrow \Theta) \quad (1.15)$$

Por la definición de BRDF en la ecuación 1.4 se tiene que:

$$\begin{aligned} f_r(x, \Psi \rightarrow \Theta) &= \frac{dL(x \rightarrow \Theta)}{dE(x \leftarrow \Psi)} \\ L_r(x \rightarrow \Theta) &= \int_{\Omega_x} f_r(x, \Psi \rightarrow \Theta) L(x \leftarrow \Psi) \cos(N_x, \Psi) dw_\Psi \end{aligned} \quad (1.16)$$

Colocando estas ecuaciones juntas se obtiene la ecuación de renderizado:

$$L(x \rightarrow \Theta) = L_e(x \rightarrow \Theta) + \int_{\Omega_x} f_r(x, \Psi \rightarrow \Theta) L(x \leftarrow \Psi) \cos(N_x, \Psi) dw_\Psi \quad (1.17)$$

## 1.5.2. Procedimientos

En esta sección se explica dos populares procedimientos clásicos para obtener una aproximación a la ecuación de renderizado, esto es una aproximación de la propagación de la luz en una escena. Estas soluciones no están pensadas para tiempos interactivos y su enfoque principal es la precisión.

Métodos como elementos finitos y Monte Carlo son los grupos de algoritmos más utilizados para aproximar la ecuación de renderizado. El método de elementos finitos utiliza alguna forma de discretización para reducir la ecuación de renderizado a una ecuación de matrices. Los métodos Monte Carlo muestran los caminos que siguen los rayos de luz en una escena, generando un estimado estadístico de la apariencia real de la escena. *Radiosity* es una popular aproximación que utiliza el método de elementos finitos. Trazado de rayos y caminos (*ray tracing* y *path tracing*) son aproximaciones comunes que utilizan el método Monte Carlo [8].

### 1.5.2.1. Radiosidad

Radiosidad es una aproximación con elementos finitos para el cómputo del transporte de luz global. Esta técnica fue introducida por Goral et al. en 1984 [9]. La idea general es discretizar las superficies de la escena en elementos finitos de éstas, estos elementos son usualmente llamados *patches* (parches o trozos) los cuales son utilizados para calcular el transporte

de luz entre ellos como se observa en la Figura 10. Esto conlleva a ciertas implicaciones; de cada *patch* se necesita guardar el valor de radiosidad para las superficies difusas, o la distribución direccional de la luz saliente y entrante para superficies no difusas.

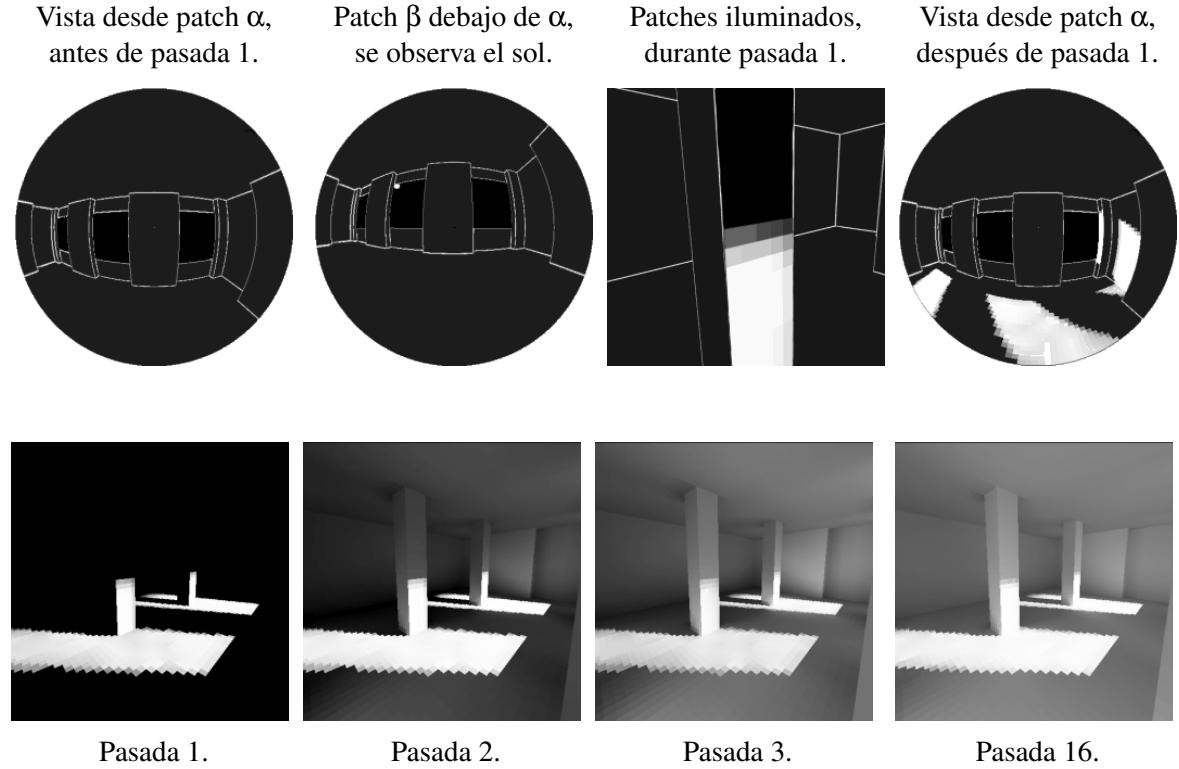


Figura 10: Ejemplo de varias pasadas de radiosidad sobre una escena [10].

### 1.5.2.2. Trazado de Rayos e Integración Monte Carlo

La ecuación de renderizado puede ser aproximada utilizando el algoritmo de trazado de rayos o *ray tracing*, esta es una técnica basada en integración Monte Carlo. Para aproximar la propagación de la luz sobre un punto se crea un número considerable de muestras en variadas direcciones, luego por cada muestra se evalúa la ecuación de renderizado y el promedio de todos los resultados converge hacia la solución analítica de la ecuación de renderizado sobre ese punto. Para evaluar una muestra la luz incidente desde una dirección tiene que ser calculada, para esto un rayo de luz es enviado en una dirección y la luz emitida desde el primer punto de colisión es calculada evaluando la ecuación de renderizado en este punto. En la Figura 11 se puede observar el resultado de esta técnica.

*Ray tracing* está dividido en dos categorías: forward y backward. Forward *ray tracing* consiste en lanzar las trazas/rayos de luz desde las fuentes de luz y usar aquellos que llegan a la

cámara. Backward *ray tracing* por el contrario lanza las trazas/rayos de luz desde la cámara y traza el camino de estos a través de la escena [11].



Figura 11: *Ray tracing* sobre una escena, se puede observar mayor calidad y reducción de ruido al aumentar la cantidad de muestras [12].

## 1.6. Pipeline de Renderizado Programable

La función del pipeline de renderizado es generar imágenes bidimensionales para ser desplegadas en pantalla dada una cámara virtual, objetos tridimensionales, fuentes de luz, ecuaciones sombreado, texturas y otros [13]. Este pipeline está dividido en varias etapas como se observa en la Figura 12.

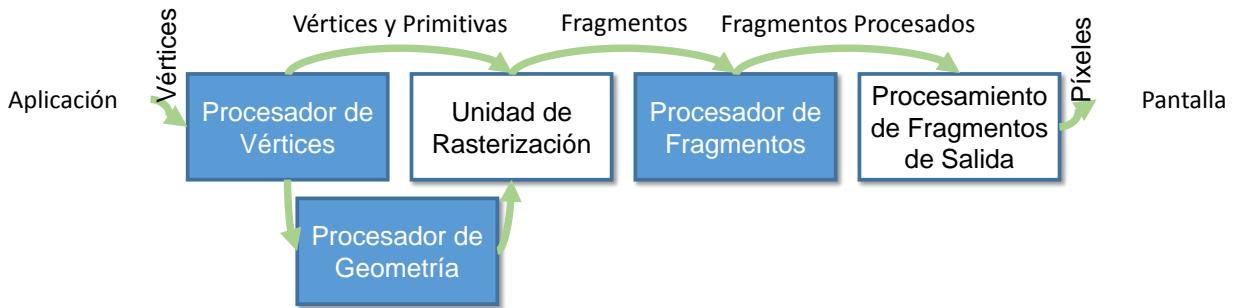


Figura 12: Etapas del pipeline de renderizado.

### 1.6.1. Procesador de Vértices

El procesador de vértices o *vertex shader* realiza operaciones sobre los vértices que definen a la geometría a renderizar. Un vértice puede estar definido por sus coordenadas en

el espacio tridimensional u otra información como color, vector normal, características para la iluminación, coordenadas de texturas, entre otros.

### 1.6.2. Procesador de Geometría

El procesador de geometría o *geometry shader* es una capacidad reciente en la unidad de procesamiento gráfico (GPU). Esta etapa del pipeline permite la manipulación de las mallas que representan la geometría en escena por primitiva. De esta forma los vértices de entrada al pipeline pueden ser enviados al *geometry shader* como vértices individuales, líneas (dos vértices) o triángulos (tres vértices). El *geometry shader* permite la modificación de los parámetros que definen estas primitivas y la generación de nuevas primitivas que serán enviadas al proceso de rasterización.

### 1.6.3. Rasterización

Las primitivas que alcanzan esta etapa del pipeline son procesadas por la unidad de rasterización en la GPU. En esta etapa todos los polígonos son escaneados y convertidos en valores dentro de una cuadrícula regular bidimensional. Durante esta conversión, el triángulo es configurado para generar todos los fragmentos que cubren el área de proyección de este triángulo. Luego por cada uno de estos fragmentos se calculan sus propiedades como color, posición y coordenadas de textura u otros utilizando interpolación lineal entre estas propiedades almacenadas en los tres vértices del triángulo procesado. Los fragmentos generados son luego enviados al procesador de fragmentos.

### 1.6.4. Procesador de Fragmentos

Todos los fragmentos generados durante el proceso de rasterización son procesados en el procesador de fragmentos o *fragment shader*. Durante este proceso se realizan operaciones como mapeo de texturas, cálculo de iluminación por fragmento y otras técnicas de sombreado. Estos fragmentos luego de ser procesados son escritos en un *framebuffer* y enviados al hardware para ser procesados según operaciones de visibilidad y mezclado para finalmente ser mostrados en pantalla.

### 1.6.5. Cómputo de Propósito General en la GPU

La GPU está diseñada específicamente para el renderizado de gráficos en computadora. Esta puede procesar de forma independiente muchos vértices y fragmentos en paralelo. En este sentido la GPU es un procesador de flujo de datos que puede operar en paralelo ejecutando un núcleo o *kernel* sobre los registros en este flujo. Con las nuevas capacidades de cómputo en las GPU se ha incrementado la flexibilidad de estos procesadores, agregando nuevas operaciones y permitiendo su uso para programación de propósito general. Ello facilita el uso de la GPU como un procesador de cómputo paralelo masivo utilizando los muchos núcleos o *cores* de cómputo existentes en las GPU modernas. Esta técnica es llamada cómputo de propósito general en la unidad de procesamiento gráfico (GPGPU).

## 1.7. Técnicas en Renderizado de Imágenes

En esta sección se explican técnicas comunes utilizadas en síntesis o renderizado de imágenes que son relevantes para este trabajo.

### 1.7.1. Mapeado de Sombras

Con la luz representada en forma de rayos las superficies sombreadas reciben menos rayos de luz, ya que estas están ocluidas por otras superficies que se encuentran entre ellas y los emisores de luz. En el pipeline de renderizado estándar, donde las superficies en escena son representadas en geometría poligonal, trazar rayos por cada fragmento para comprobar la visibilidad del mismo no es una operación trivial.

Los mapas de sombras, presentados inicialmente por Lance Williams en 1978 [14] son una solución simple para el cálculo de visibilidad de un fragmento. La técnica consiste en proyectar la escena en una textura bidimensional desde la posición y con la dirección de una fuente de luz. La proyección es calculada utilizando una matriz de proyección  $P_l$ . Por cada píxel de esta textura la profundidad de cada fragmento sobre una superficie es almacenada. Esta textura es llamada mapa de sombra.

Una vez que se procede a renderizar la escena desde el punto de vista del observador, por cada fragmento con posición  $p_{ws}$  en espacio de mundo, la posición en el mapa de sombra

$p_{sh}$  es calculada utilizando la siguiente ecuación:

$$p_{sh} = P_l \cdot p_{ws} \quad (1.18)$$

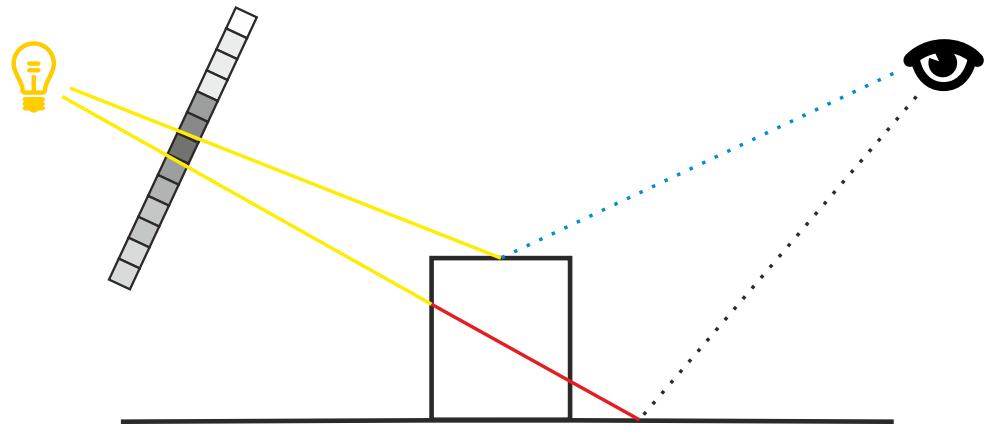


Figura 13: La profundidad almacenada en el mapa de sombra (amarillo) es comparada con la profundidad del punto en la superficie desde la luz (rojo).

Como se muestra en la Figura 13, si la profundidad del fragmento desde la fuente de luz es mayor que el valor almacenado en el mapa de sombra entonces este punto está sombreado.

El mapeado de sombras es una solución sencilla y efectiva al problema de pruebas de visibilidad pero esta técnica tiene dos mayores desventajas. El mapa de sombras está limitado por la resolución de la textura y además este representa una discretización de la profundidad de la escena vista desde la fuente de luz, lo que introduce una variedad de anomalías visuales. A partir de este concepto existe una variedad de algoritmos para el cálculo de sombras que intentan solventar estos problemas u obtener resultados de mejor calidad visual.

### 1.7.2. Sombreado Diferido

En sombreado directo la representación poligonal de la escena es rasterizada y operaciones por píxel como iluminación y sombreado son realizadas por cada fragmento generado por el proceso de rasterización. Esto es poco efectivo cuando se toma en consideración que muchos fragmentos no forman parte de la imagen final.

Con sombreado diferido se pueden realizar estas operaciones por píxel solo sobre los fragmentos visibles. Este concepto está basado en el trabajo de Deering et al. en 1998 [15]. La escena es renderizada solo una vez y varios atributos de la escena son almacenados en buffers.

Este buffer es llamado buffer de geometría (G-Buffer) y fue introducido por Saito et al. en 1990 [16]. El contenido general de un G-Buffer es profundidad, albedo y normal (ver Figura 14). Esto puede cambiar según las necesidades de la aplicación. El propósito de almacenar esta información es separar las operaciones que solo son necesarias sobre los fragmentos visibles de la rasterización de toda la escena, de manera que cálculos como iluminación ahora son realizados en otro paso solo sobre cada píxel almacenado en el G-Buffer.



## Normales.

## Albedo.

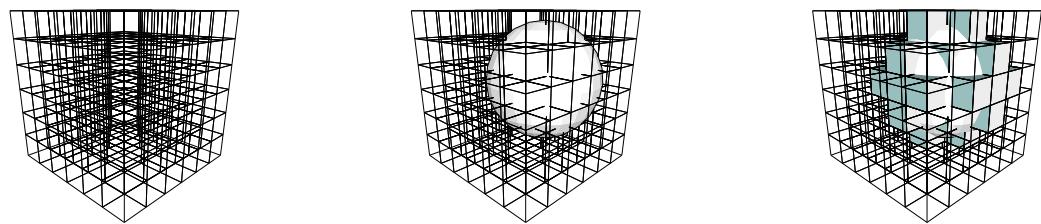
Profundidad.

Figura 14: El contenido de un buffer de geometría.

### 1.7.3. Voxelización

Un voxel o a veces llamado píxel volumétrico representa una muestra singular o elemento volumétrico sobre un grid regular en un espacio tridimensional. Este voxel puede contener cualquier valor definido por la aplicación o incluso múltiples valores.

El proceso de generar superficies discretas en una representación volumétrica a través de véxoles se le llama voxelización (ver Figura 15).



(a) Grid regular del vóxeles.

(b) Superficie en el grid.

(c) Superficie en véxeles.

Figura 15: Representación de una superficie en véxoles con voxelización fina.

Se puede distinguir el proceso de voxelización de superficies en dos clases: voxelización fina con separabilidad factor 6 y voxelización conservativa con todos los vóxeles que tocan la superficie activos o separabilidad factor 26. En el trabajo de Huang et al. en 1998 [17] se describe el proceso de voxelización y terminología con mayor detalle. También existen cuatro tipos de enfoques en voxelización:

- **Voxelización binaria:** Cada vóxel sólo almacena si hay geometría presente o no.
- **Voxelización multi-valor:** Cada vóxel puede almacenar múltiples valores de data arbitraria como opacidad, normal, etc.
- **Voxelización de contorno:** Sólo se voxeliza la superficie o contorno de los objetos.
- **Voxelización sólida:** Además de la superficie también se voxeliza el interior del objeto.

## 1.8. Iluminación Global en Tiempo Real.

En esta sección se examinan algunos algoritmos para el cálculo de iluminación global en tiempos interactivos o *real-time*. Iluminación indirecta con trazado de conos y vóxeles es revisada con detalle ya que esta técnica es de particular interés para este trabajo.

### 1.8.1. Luces Puntuales Virtuales

Una variedad de algoritmos para el cálculo de iluminación global se inspiran o hacen uso del concepto de luz puntual virtual (VPL). Este trabajo fue presentado por Keller en 1997 [18].

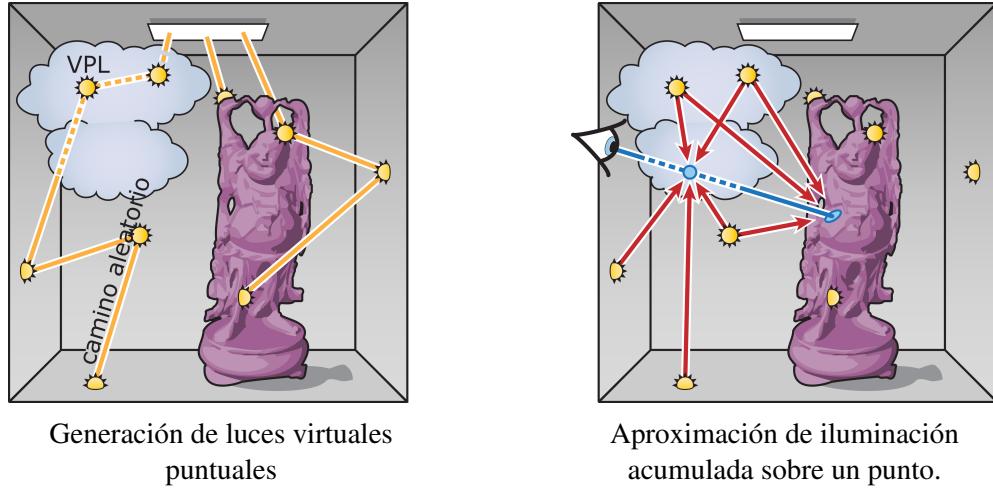


Figura 16: Pasos del algoritmo VPL para la aproximación de iluminación indirecta [19].

En este algoritmo se aproxima la radiancia reflectada en escena utilizando un conjunto de luces virtuales. La radiancia que llega a un punto  $x$  es aproximada por la radiancia que

proviene de las luces virtuales. Pruebas de visibilidad para cada una de estas luces son realizadas utilizando técnicas de sombreado estándar y la radiancia proveniente de cada una de estas es almacenada en un buffer de acumulación. En la Figura 16 se ilustra este algoritmo. Las luces virtuales son generadas a partir de partículas lanzadas por las fuentes de luz principales utilizando la secuencia de Halton para el muestreo. En un principio, un número  $n$  de partículas son generadas, como no toda la radiancia es absorbida algunas de estas partículas son reflejadas. Luego del primer rebote, uno número  $p'n$  de partículas son reflejadas. Luego de  $j - 1$  reflexiones  $p'^j$  son reflejadas. El número  $p'$  es descrito por la siguiente ecuación:

$$p' = \frac{\sum_{k=1}^K p_{d,k} |A_k|}{\sum_{k=1}^K |A_k|} \quad (1.19)$$

donde la escena es compuesta por  $K$  elementos de superficie  $A_k$  con una reflectividad promedio de  $p_{d,k}$ .

### 1.8.2. Mapas de Sombras Reflexivo

Otra técnica utilizada en varios algoritmos de iluminación global es mapas de sombras reflexivos (RSM) o *reflective shadow maps*. La técnica de RSM fue presentada por Dachsbaecher y otros en 2005 [20]. Esta técnica está inspirada en mapeado de sombras, como fue explicado anteriormente en la sección 1.7.1, se utiliza proyección desde la fuente de luz para determinar el primer rebote de luz. Al renderizar la escena desde el punto de vista de la fuente de luz, se entiende que todos los fragmentos en el mapa de sombras son los únicos fragmentos involucrados en el primer rebote de luz. En RSM cada uno de los píxeles en el mapa de sombras es considerado una fuente de luz. Por cada píxel  $p$  además de la profundidad  $d_p$ , se necesita almacenar posición  $x_p$ , normal  $n_p$ , y el flujo de radiancia reflectada  $\Theta_p$  (ver Figura 17).

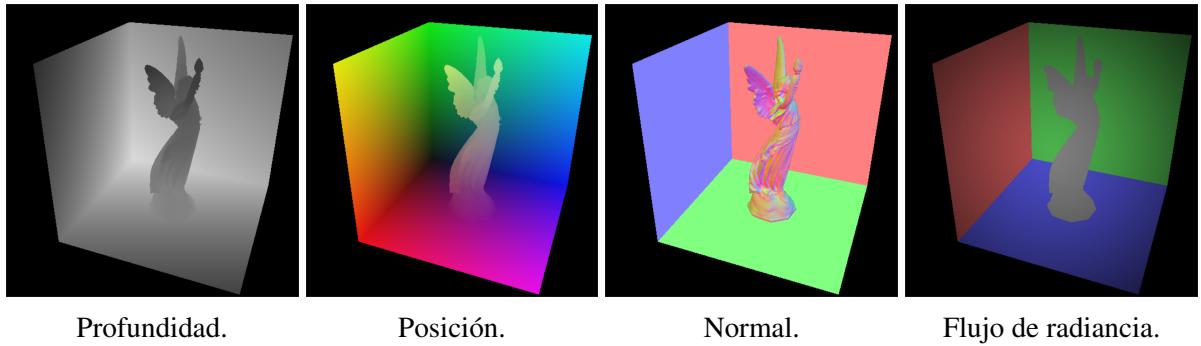


Figura 17: Mapas utilizados por RSM [20].

Si se asume que todas las superficies son reflectores difusos, la intensidad de la radiancia emitida en una dirección  $\omega$  desde un píxel del RSM es descrita por la siguiente ecuación:

$$I_p(\omega) = \Theta_p \max(0, n_p \cdot w) \quad (1.20)$$

La iluminación indirecta de un punto se calcula sumando todas las intensidades de todos los píxeles (considerados ahora como luces) en el RSM visibles. Calcular esto es costoso, por tanto en vez sumar todos los píxeles visibles al punto se toma cierta cantidad de muestras del RSM. La posición del punto iluminado  $x_p$  es proyectado sobre el RSM y las muestras son seleccionadas alrededor de esta posición proyectada. La densidad de las muestras decrece con la distancia cuadrada de la posición proyectada al punto iluminado. Esta técnica asume que dos superficies cercanas proyectadas al RSM también son cercanas en el RSM y que la muestra es directamente visible desde la superficie iluminada.

### 1.8.3. Volúmenes de Propagación de Luz en Cascada

Volúmenes de propagación de luz en cascada (CLPV) o *Cascaded Light Propagation Volumes* presentado por Kaplanyan et al. en 2010 [21] es un algoritmo para el cálculo de iluminación indirecta difusa en tiempo real. En la Figura 18 se observan los resultados de esta técnica.



Figura 18: Iluminación global para la escena *Sponza* utilizando volúmenes de propagación de luz [21].

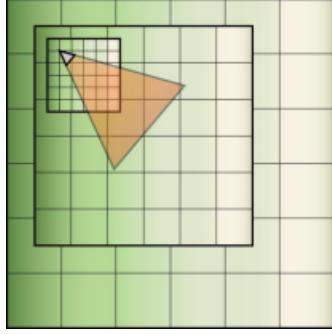


Figura 19: Cuadrículas de propagación anidadas, las cuadrículas se solapan.

utilizando una cuadrícula mucho más fina.

Primero por cada fuente de luz, es necesario renderizar un RSM. Cada téxel del RSM es considerado una VPL. La intensidad del VPL es acumulada y almacenada como un harmónico esférico dentro de las celdas de la cuadrícula.

Para una correcta propagación de la luz el algoritmo necesita conocer la geometría de la escena. Por esto de la misma manera en la que se almacena la intensidad de la luz también se almacena una representación de la escena en una cuadrícula. Esta representación es guardada sobre el GV. Esta cuadrícula es trasladada por la mitad del tamaño de una celda con respecto al LPV, esto asegura que el centro de todas las celdas del GV queden en las esquinas del LPV. Para conocer segmentos de geometría se utiliza un G-Buffer y las fuentes de luz de los RSMs.

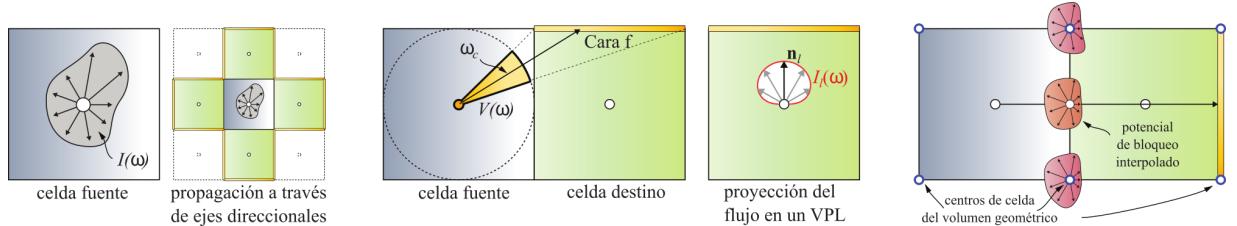


Figura 20: Izquierda: Cada celda del LPV almacena la intensidad de luz que es propagada desde la celda fuente. Centro: El flujo es calculado sobre cada cara de la celda destino para preservar información direccional. Derecha: Oclusión borrosa o *fuzzy* almacenando una representación volumétrica de la escena [21].

La propagación de luz se realiza de forma iterativa. La intensidad de luz en cada iteración es propagada a 6 vecinos por celda según el eje direccional principal. Primero por cada celda adyacente el flujo de radiancia incidente por cada una de las celdas es calculado. Luego el flujo incidente de cada celda es transformado en emitancia radiante. Esto se logra creando

VPLs, cada una de estas luces virtuales son colocadas sobre una de las caras de la celda y emiten un flujo radiancia similar el flujo de radiancia de la celda. Estas VPL son acumuladas dentro del LPV de nuevo y almacenadas como esféricos harmónicos utilizando el mismo proceso de inyección del primer paso. En la Figura 20 se describen los pasos de este algoritmo.

#### 1.8.4. Iluminación Indirecta con Trazado de Conos y Vóxeles

Este algoritmo es presentado por Crassin et al. en 2011 [23] para el cálculo de iluminación indirecta utilizando trazado de conos contra vóxeles o *Indirect Illumination Using Voxel Cone Tracing*. En la Figura 21 se observan los resultados de este algoritmo. En esta técnica se utiliza una estructura de árbol disperso o *sparse* para almacenar ya filtrados los valores necesarios para el cálculo de iluminación indirecta en vóxeles. Este árbol es una representación tridimensional de la escena por tanto cada nodo tiene ocho hijos que representan las ocho particiones de un cubo en partes más pequeñas de forma uniforme. Esta clase de estructuras son llamadas *octrees*. La estructura dispersa requiere menor consumo de memoria ya que solo los vóxeles necesarios son almacenados.



Figura 21: Escena *Sponza* renderizada utilizando trazado de conos con vóxeles para la iluminación indirecta [23].

El algoritmo comprende varios pasos: primero la información de la luz y escena son almacenados en las hojas de la estructura de árbol, este es el nivel más fino de la jerarquía. Luego estos valores son filtrados dentro del árbol disperso hacia todos los niveles de la jerarquía hasta llegar a la raíz. En un último paso para el cálculo de iluminación indirecta por cada fragmento,

los valores dentro de esta jerarquía son recolectados sobre una semiesfera utilizando trazado de conos. En algoritmos como *ray tracing* esta recolección de valores es lenta y es realizada por muchos rayos. Es de notar que todos estos rayos trazados sobre la semiesfera son direccional y espacialmente coherentes. Trazado vóxeles con conos (VCT) hace uso de este concepto para discretizar muchos rayos en simples conos.

#### 1.8.4.1. Construcción del Octree de Vóxeles

El algoritmo está pensando para funcionar con escenas dinámicas. Sin embargo las escenas son divididas entre partes dinámicas y estáticas para acelerar el proceso de voxelización con objetos dinámicos.

Primero la escena es renderizada utilizando proyección ortogonal. Cada triángulo es proyectado sobre uno de los ejes principales, este eje es seleccionado según la normal del triángulo, esto se hace para maximizar el área visible del triángulo con respecto al eje. Cada fragmento producto de esta proyección es almacenado en una lista de vóxeles-fragmentos junto a parámetros como posición en escena, normal y color. Para saber la cantidad de elementos que esta lista debe almacenar, primero se debe realizar una pasada contando el número de fragmentos con un contador atómico.

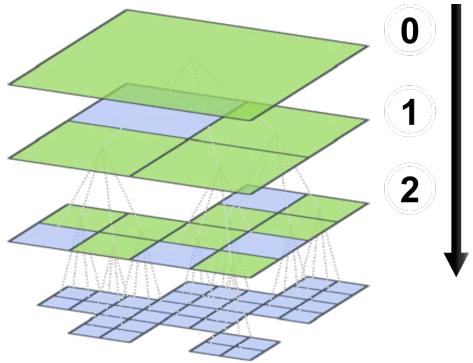


Figura 22: Descripción gráfica del proceso de subdivisión del octree.

Una vez llena la lista de vóxeles-fragmentos, se genera un hilo de procesamiento por cada fragmento en la lista. Los fragmentos son introducidos en el árbol disperso que inicialmente solo tiene un nodo raíz (ver Figura 22). Cada vez que un nodo de este octree necesita ser dividido, un nuevo nodo es creado y se almacena sobre memoria ya reservada en la GPU. La posición del nodo en memoria es determinada por un contador atómico, el cual se incrementa con cada nuevo nodo. Al inicio del proceso de subdivisión de los nodos se genera una gran cantidad de colisiones entre hilos, por esto cada nodo tiene asociado un símbolo mutex.

#### 1.8.4.2. Contenido de un Vóxel

El algoritmo está diseñado para hacer uso de filtrado trilineal por hardware. Sin embargo dos vóxeles vecinos no necesariamente están posicionados de forma subsecuente en me-

moria. Por esto cada nodo contiene un bloque o *brick*. Este bloque representa el entramado  $3^3$  de la celda, donde estas celdas se encuentran en las esquinas de los hijos del nodo (ver Figura 23).

Cada voxel representa varios parámetros, entre ellos color, opacidad, normal, intensidad, etc.

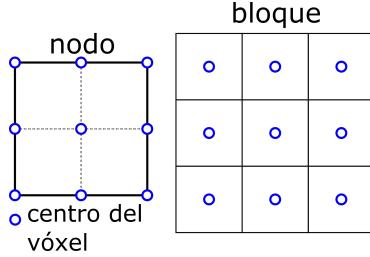


Figura 23: Bloques y nodos, los valores de los bloques se encuentran en las esquinas para rápido acceso y filtrado [23].

#### 1.8.4.3. Filtrado en Niveles de Detalle

Inicialmente cada uno de estos parámetros es almacenado dentro de las hojas del árbol disperso. Luego de forma iterativa estos valores son filtrados desde los niveles más bajos a los niveles más altos de la jerarquía lo que resulta en distintos niveles de detalles para la escena voxelizada, este proceso es llamado *mipmapping*. Cada nodo de un bloque es filtrado a partir de las  $3^3$  celdas del nivel anterior en jerarquía. Para calcular el valor filtrado sobre el actual nodo el algoritmo promedia los valores de los nodos en el nivel anterior. Al calcular el valor filtrado cada voxel debe ser pesado con la inversa de su multiplicidad, resultando en un kernel gaussiano de tamaño  $3^3$ .

#### 1.8.4.4. Trazado de Conos y Vóxeles

Una vez que el árbol disperso está filtrado y completo se utiliza para el cálculo de iluminación indirecta. Por cada fragmento un conjunto de conos es generado. La dirección y apertura de cada cono es determinada por la BRDF del material en ese fragmento. Por ejemplo la BRDF Blinn-Phong vista en la sección 1.4.1.3 puede ser descompuesta como un lóbulo ancho para la parte difusa y un lóbulo especular. Para el lóbulo difuso varios conos son generados orientados por la semiesfera sobre el punto con apertura y dirección maximizadas a tal manera que los conos cubran gran parte de la misma. Para el lóbulo especular se genera un solo cono

con una apertura que varía según el término  $n$  de la BRDF Blinn-Phong. El cono especular tiene como dirección la dirección de la luz incidente reflectada  $R$  (sección 1.4.1.2).

El algoritmo utiliza trazado de conos aproximado para acumular la intensidad de la luz incidente sobre el fragmento. Los valores son recolectados de varias muestras a través del recorrido del cono utilizando composición *front-to-back*. Por cada muestra se examina el árbol disperso de véxeles. El nivel del árbol a examinar es determinado por el diámetro del cono en esa posición.

#### 1.8.4.5. Filtrado Anisotrópico de Vóxeles.

A pesar de que el filtrado gaussiano es suficiente para proveer resultados visuales coherentes, algunos problemas de calidad visual pueden ocurrir bajo ciertas condiciones. El primer problema se conoce como el problema de la pared rojo-verde. Al promediar valores dentro del octree con dos véxeles opacos con diferentes colores provenientes de por ejemplo, dos paredes planas, el color resultante del véxel describe ambas paredes como si estas fueran transparentes. Otro problema resulta también de promediar opacidad entre véxeles totalmente transparentes y véxeles totalmente opacos, resultando un valor filtrado semitransparente. Esto puede resultar en fugas de luz a través de la geometría en la escena.

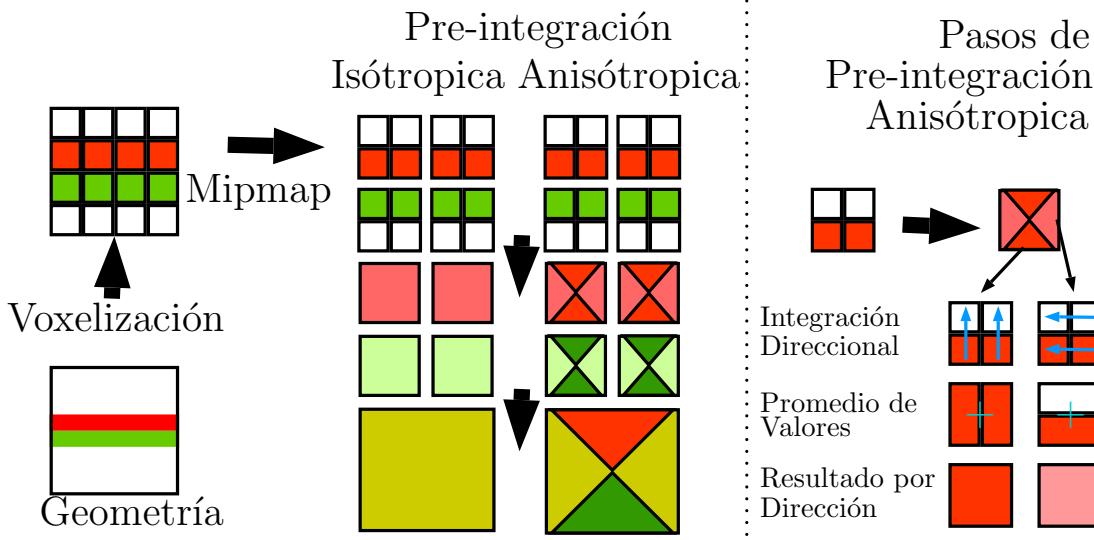


Figura 24: A la izquierda se describe el proceso de *mipmapping* de véxeles sin y con filtrado anisotrópico. Los pasos para la integración direccional se observan a la derecha.

Para solventar este problema se realiza una representación anisotrópica de los véxeles durante el proceso de *mipmapping*. En vez de tener un solo canal de valores filtrados sin direc-

ción, ahora los valores serán filtrados de forma direccional, almacenando seis canales por cada eje positivo y negativo. Un valor direccional es calculado realizando un paso de integración volumétrica en profundidad y promediando los cuatro valores direccionales para obtener el valor resultante según una dirección (ver Figura 24).

#### 1.8.4.6. Captura de Iluminación Directa en Vóxeles

Para el cálculo de iluminación indirecta es necesario describir como la radiancia incidente es almacenada en los nodos del árbol. Este proceso está inspirado en RSM, donde la escena es renderizada desde el punto de vista de la fuente de luz y se utiliza rasterización estándar para almacenar las posiciones de los fragmentos en una textura. Cada píxel en esta textura representa un fotón que rebota en escena. Esta textura se le llama mapa de luz-vista o *light-view map*. Luego de generar este mapa es necesario almacenar los fotones en el árbol octree. Los fotones son almacenados como una distribución direccional y energía proporcional al casquete esférico del ángulo sólido del píxel visto desde la luz.

El procesamiento de la textura de fotones sobre el árbol se realiza en el procesador de fragmentos o *fragment shader*. Como usualmente la dimensión del *light-view map* es mayor a la resolución de la cuadrícula de vóxeles se puede asumir que los fotones serán almacenados directamente en las hojas del árbol octree. Además, los fotones siempre pueden ser almacenados en el nivel más fino de detalle en la representación con vóxeles porque estos describen información de la superficie geométrica. Es posible que muchos fotones terminen sobre un mismo vóxel, por esto es necesario utilizar adicción atómica para garantizar coherencia entre los hilos generados por cada fragmento.

# Capítulo 2

## Solución Propuesta

Un componente importante en la composición de imágenes realistas es la iluminación. Para el cálculo de iluminación directa bajo el pipeline de renderizado estándar ya se tienen décadas de estudio y distintas técnicas que permiten tiempos de renderizado muy cortos en hardware moderno. En contraste, el cálculo de iluminación indirecta sigue siendo una tarea computacionalmente compleja.

Para el cálculo de iluminación indirecta existen varios procedimientos muchos de estos inspirados en los ya explicados en la sección 1.5.2. Sin embargo estos no están pensados para renderizado en tiempo real. En la sección 1.8 se examinó algunas aproximaciones para el cálculo de iluminación indirecta en tiempo real, estas técnicas explotan ciertas características de hardware o recursos del pipeline de renderizado.

Parte de nuestra implementación para el cálculo de iluminación global en tiempo real está basada en el trabajo de Crassin et al. en 2011 [23]. Este trabajo ya fue examinado en la sección 1.8.4. La técnica fue escogida porque además de reflexión difusa también permite el cálculo de superficies lustrosas con reflexión especular, a diferencia, otras técnicas como CLPV solo permiten reflexión difusa. Una ventaja de esta aproximación es que los valores de radiancia ya se encuentran almacenados sobre una representación con véxeles. Esto acelera el cálculo de luz incidente bajo el esquema de integración Monte Carlo visto en la sección 1.5.2.2, en este caso los conos permiten realizar una cruda aproximación de un grupo de rayos. Además de esto el algoritmo puede ser utilizado en escenas totalmente dinámicas.

## 2.1. Voxelización

El trazado de conos contra geometría poligonal compleja es costoso. Encontrar los puntos de intersección entre un cono y un polígono es mucho más complejo que intersecciones rayo-polígono, además de esto un solo cono podría intersectar muchos polígonos.

Para simplificar el trazado de conos se utiliza una discretización de la escena en forma de véxeles. Esta representación puede ser filtrada a niveles más bajos de detalle (ver Figura 25). Esto permite aproximar el efecto de extensión sobre la apertura del cono utilizando cada vez un nivel de detalle más bajo a través del recorrido del mismo.



Figura 25: Distintos niveles de detalle de una escena voxelizada.

Para las partes dinámicas de la escena este proceso de voxelización debe ser realizado cada vez que sucede algún cambio sobre cualquier superficie que pertenece a un objeto dinámico. Por esta razón se requiere un algoritmo de voxelización de alto rendimiento para mantener tiempos interactivos.

### 2.1.1. Voxelización Conservativa

Nuestra implementación realiza voxelización conservativa de geometría de alto rendimiento utilizando la GPU y explotando características del pipeline de renderizado con OpenGL. Para esto se implementó el algoritmo de voxelización utilizando rasterización en hardware explicado en el libro *OpenGL Insights* por Cyril Crassin y Simon Green en *Octree-Based Sparse Voxelization Using the GPU Hardware Rasterizer* [24]. Este algoritmo está basado en el trabajo de Zhang et al. en 2007 [25] para la voxelización conservativa utilizando la GPU y el trabajo de Hasselgren et al. en 2005 [26] sobre rasterización conservativa. En la Figura 26 se puede observar la arquitectura descrita en *Octree-Based Sparse Voxelization Using the GPU Hardware Rasterizer* para el proceso de voxelización.

Para maximizar el área de rasterización, la idea es proyectar cada triángulo utilizando proyección ortogonal por cada eje direccional. El eje dominante es escogido según el vector

normal del plano definido por los vértices del triángulo.

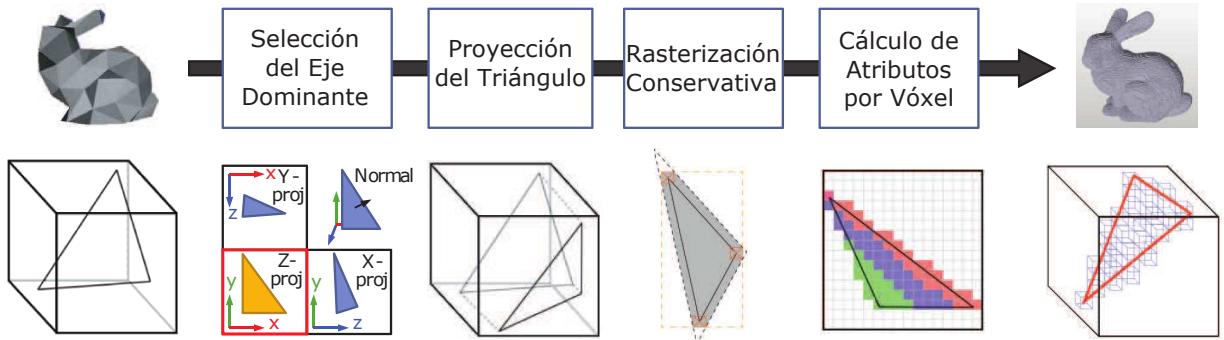


Figura 26: Descripción del pipeline de voxelización [24].

Por cada triángulo proyectado es necesario generar un polígono delimitante un poco más grande que el triángulo para garantizar la voxelización conservativa. Este polígono debe permitir que por cualquier triángulo proyectado, tocando un píxel, este obligatoriamente va a tocar el centro de este píxel, por tanto el pipeline de rasterización generará fragmentos para este triángulo. El polígono se genera expandiendo cada vértice del triángulo hacia afuera utilizando el procesador de geometría o *geometry shader*. El polígono delimitante no sobreestima la cobertura del triángulo por tanto este no tiene forma de triángulo (ver Figura 27). Los fragmentos excedentes de este polígono son descartados en el *fragment shader* utilizando una región delimitante.

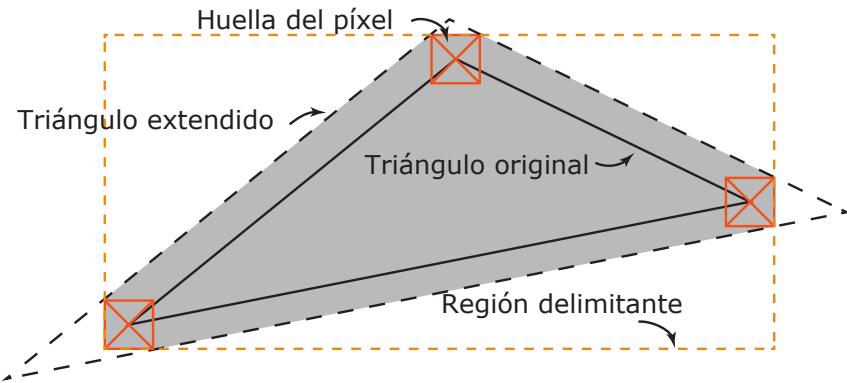


Figura 27: Polígono delimitante de un triángulo utilizado para rasterización conservativa.

### 2.1.2. Composición de Fragmentos y Vóxeles

Una vez que los fragmentos han sido generados en el *fragment shader*, los valores deseados pueden ser almacenados en una textura 3D utilizando operaciones de escritura que

provee la extensión *GL\_ARB\_shader\_image\_load\_store* en OpenGL [27]. Sin embargo múltiples fragmentos de diferentes triángulos pueden caer sobre una misma posición en esta textura 3D. Bajo el ambiente paralelo del *fragment shader* no es posible saber el orden en que los fragmentos son creados y procesados, esto se traduce en resultados arbitrarios cada vez que se re-voxeliza la escena.

Para solventar este problema se utilizaron operaciones atómicas provistas por la misma extensión citada anteriormente. En nuestra implementación los valores necesarios a almacenar en vóxeles son normal, albedo y emisión. Una operación coherente y sencilla con respecto a estas propiedades en el espacio que envuelve un vóxel es un promedio.

### 2.1.3. Voxelización Dinámica

Para escenas complejas y densas en geometría poligonal el proceso de voxelización puede tomar un tiempo de ejecución considerable. Esto afecta el rendimiento general del algoritmo si la escena necesita ser re-voxelizada constantemente. Es por esto conveniente separar la voxelización de superficies estáticas y dinámicas en escena. La idea es voxelizar la parte estática de la escena una sola vez mientras que la parte dinámica es revoxelizada solo cuando sea necesario o constantemente por frame.

En nuestra implementación se utilizan texturas 3D para almacenar los vóxeles en vez de un *octree* disperso como en la propuesta original de Crassin. La separación entre valores estáticos y dinámicos bajo el esquema de un *octree* consiste en dividir el árbol en una parte dinámica y una parte estática. Considerando la cualidad dispersa de esta estructura esto no es de gran peso en memoria.

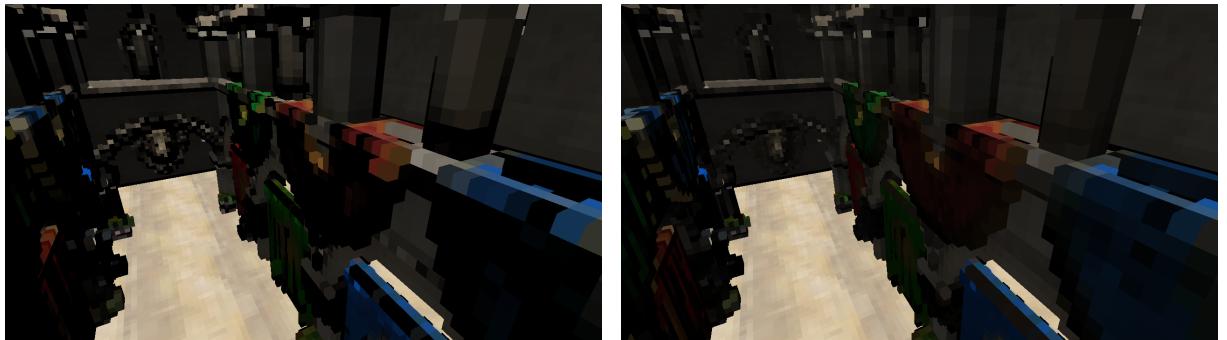
Al emplear texturas 3D esto es equivalente a generar seis texturas en vez de tres para almacenar los valores de albedo, normal y emisión. Tres de estas para la parte estática y tres para la parte dinámica, lo que es extremadamente ineficiente en memoria. Para evitar la creación de estas texturas se utiliza un volumen extra que permite indicar qué vóxel forma parte de la parte estática y cuál no. Durante el proceso de voxelización dinámica solo los fragmentos que se encuentran dentro del espacio de un vóxel dinámico pueden escribir sobre la textura 3D.

## 2.2. Sombreado de Vóxeles

Para el cálculo de iluminación indirecta es necesario sombrear cada voxel. El proceso de sombreado de voxels permite almacenar la radiancia incidente sobre la escena discretizada en voxels. Utilizar mapas de luz-vista por cada fuente de luz como fue explicado en la sección 1.8.4.6, puede resultar ineficiente tanto en consumo de memoria como en rendimiento. Si se considera una escena con muchas luces, cada una de estas luces debe tener un mapa asociado y se debe repetir el proceso de captura por cada una de ellas. Otra desventaja de este método es la dependencia del rendimiento con la resolución del mapa de luz-vista. Al aumentar la resolución de esta textura también se aumenta el número de colisiones por cada fragmento que desea escribir sobre un mismo voxel.

Nuestra implementación utiliza *compute shaders* o el procesador de cómputo en la GPU para el sombreado difuso de cada voxel. Para calcular el término difuso sobre un fragmento utilizando la BRDF de Lambert (ecuación 1.9) se necesita saber el valor de  $\rho_d$  el cual ya es almacenado en el volumen albedo. Esta constante luego debe ser multiplicada por el  $\cos(N_x, \Psi)$  o atenuación normal. Por esto también se crea un volumen de normales. El vector  $\Psi$  se obtiene a partir la dirección de cada fuente de luz en escena.

Para fuentes de luz con dirección no uniforme como luces puntuales o focales es además necesario saber la posición del fragmento a iluminar. Siendo cada voxel una representación discreta de un espacio en escena, almacenado en una textura 3D, esta posición se extrae fácilmente convirtiendo la posición tridimensional del voxel en espacio textura a su equivalente en espacio de mundo.



Lambert.

Lambert direccional ponderado.

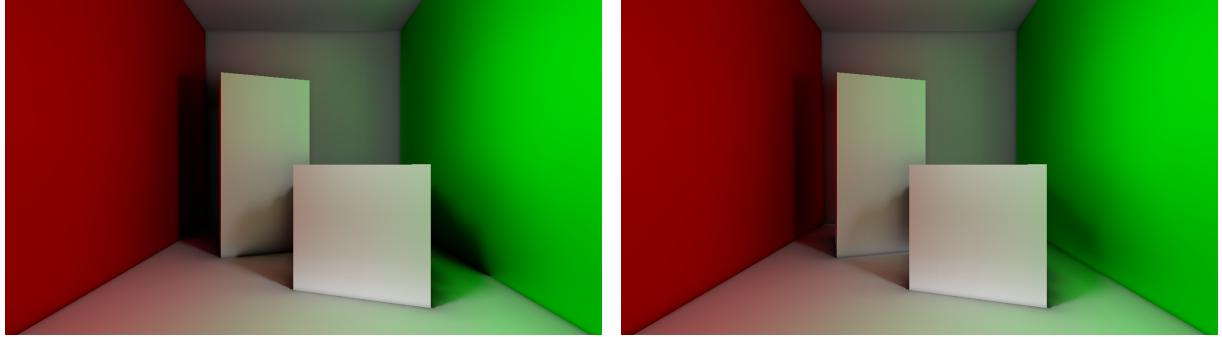
Figura 28: Sombreado difuso de voxels utilizando Lambert clásico y Lambert direccional ponderado.

Al promediar las normales en el espacio de un voxel pueden surgir varios problemas de precisión. Esto sucede especialmente cuando un voxel envuelve superficies finas cercanas

con normales dispares. Para disminuir este problema se implementaron dos modelos de iluminación de véxeles: El modelo de Lambert clásico utilizando el vector normal promedio del voxel directamente y otro denominado Lambert direccional ponderado, donde se calcula la atenuación normal por cada cara del voxel, para promediar este resultado según el peso de cada eje en el vector normal promedio. En la Figura 28 se puede observar el sombreado resultante de ambos modelos. Para el segundo modelo algunos de los véxeles totalmente negros recuperan parte de su color correcto y lugares sin disparidades en el vector normal, como el piso, mantienen el mismo color entre ambos modelos.

### 2.2.1. Trazado y Mapeo de Sombras sobre el Volumen

Para obtener resultados coherentes durante el trazado de conos, es también necesario ocluir los véxeles con sombras generadas a partir de distintas fuentes de luz en escena. En la Figura 29, se puede observar iluminación excesiva en áreas ocluidas al no incluir sombras en la representación en véxeles. Utilizando mapas de luz-vista como se explicó en la sección 1.8.4.6 esto es sencillo, ya que los véxeles ocluidos simplemente no reciben fotones durante el proceso de captura de la iluminación directa.



Con véxeles ocluidos por sombras.

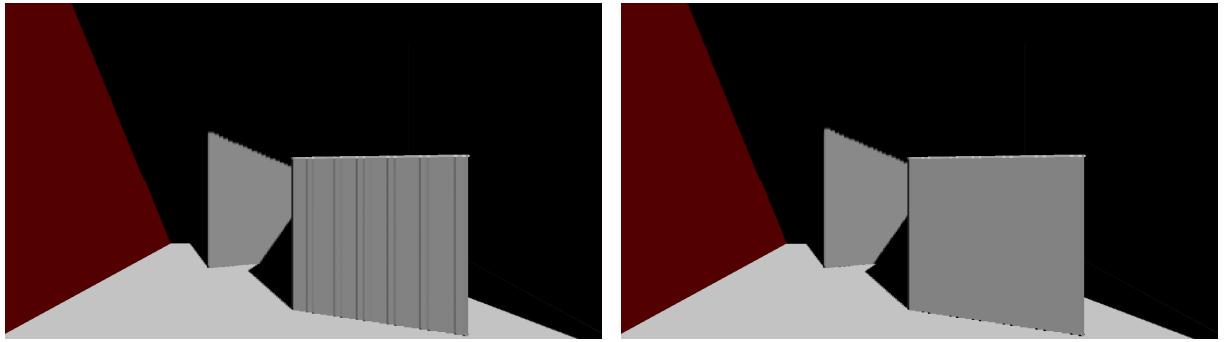
Sin oclusión de véxeles.

Figura 29: Iluminación global al utilizar una representación con véxeles con y sin oclusión por sombras.

Así, existen distintas opciones para trazar sombras sobre los véxeles. Este proceso se realiza durante el sombreado de véxeles explicado anteriormente.

Para luces directas se utiliza mapeo de sombras visto en la sección 1.7.1. En esta técnica es necesario obtener la posición en espacio de mundo del voxel. Para esto la posición tridimensional del voxel en espacio de textura se transforma a espacio de mundo. De la transformación se obtiene la posición central del voxel. Utilizar el centro del voxel para mapeo de sombras, puede ocasionar problemas cuando este punto esté ocluido por otras superficies dentro

del mismo voxel o fragmentos cercanos (ver Figura 30). Esto sucede ya que el mapa de sombras es una representación mucho más detallada de la escena vista desde una fuente de luz. Para solventar este problema se traslada la posición del voxel según el vector normal por el tamaño medio de un voxel.



Utilizando la posición del centro del voxel.

Trasladando la posición medio voxel.

Figura 30: Sombras sobre voxels con y sin traslado de la posición de sombreado.

Estando el mapeo de sombras disponible en nuestra implementación para solo una luz directa, se implementó trazado de sombras para cualquier fuente de luz puntual, focal o direccional. Los volúmenes resultantes del proceso de voxelización pueden ser utilizados para trazar rayos sobre la escena discretizada. Siendo estos volúmenes una representación mucho más simple de la escena original, utilizar técnicas comunes en trazado de rayos (sección 1.5.2.2) es viable.

Para realizar pruebas de occlusion sobre un voxel por una fuente de luz, se lanza un rayo desde la posición del voxel en la dirección opuesta de la luz incidente. Si este rayo colisiona con otro voxel entonces el de origen está ocluido.

### 2.2.1.1. Trazado de Sombras Suaves sobre el Volumen

Una técnica que incrementa la calidad visual de las sombras es la generación de un bordeado suave para las sombras. En el trazado de rayos esto se logra lanzando varios rayos en distintas direcciones a diferencia de uno solo para generar varias muestras de occlusion.

En nuestra implementación se logra obtener bordes suaves para las sombras generadas durante el sombreado de voxels con un solo rayo. Esta técnica se fundamenta en el hecho de que al trazar un rayo hacia una superficie voxelizada, este rayo colisionará más veces contra voxels cerca de los bordes de la superficie vistos desde la fuente de luz. Como se observa en la Figura 31, en lugar de detener el rayo una vez que se ha encontrado una colisión, se asigna

un valor a cada colisión y se acumula este factor a través del recorrido del rayo dividido por la distancia recorrida.

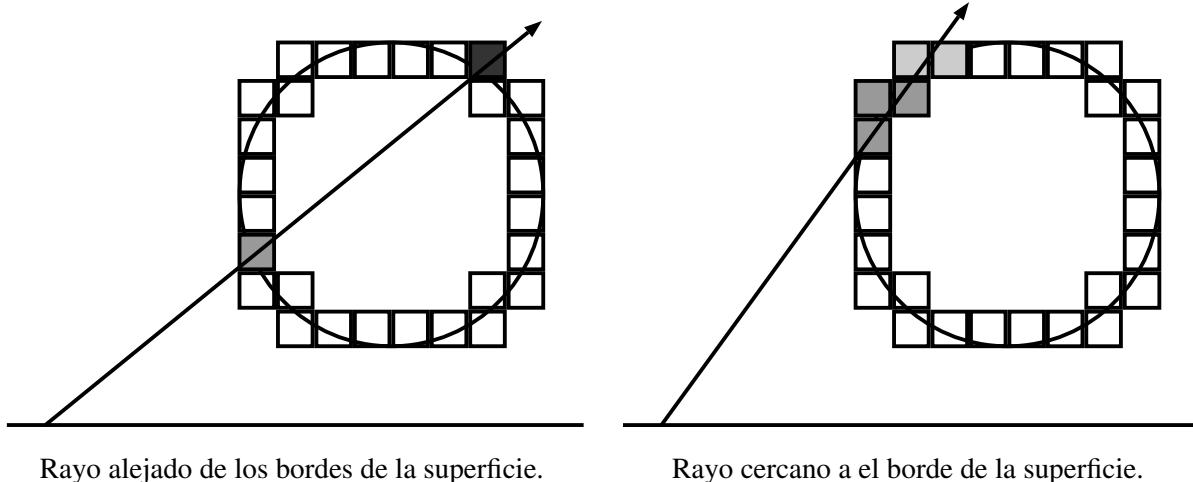


Figura 31: Descripción gráfica del proceso de acumulación de colisiones a través del recorrido de un rayo para prueba de oclusión.

## 2.3. Estructura Jerárquica

Durante el trazado de conos se utilizan distintos niveles de detalle de la escena voxelizada a medida que el diámetro del cono se expande por su recorrido en escena. En el trabajo de Crassin estos niveles de detalle se construyen utilizando la profundidad del *octree* disperso, donde el nodo raíz es el nivel de detalle más bajo y las hojas del árbol contienen el máximo nivel detalle. El proceso de filtrado desde las hojas al nodo raíz fue explicado en la sección 1.8.4.3. En nuestra implementación con texturas 3D esto representa simplemente los distintos niveles de *mipmapping* en una textura, estos pueden ser generados con una invocación al método *glGenerateMipmap* en OpenGL. Una ventaja de utilizar texturas 3D es que el filtrado cuadrilínea es soportado de forma nativa por hardware, sin necesidad de construir bloques por cada voxel como se explica en la sección 1.8.4.2. Esto simplifica de gran manera la construcción de la estructura jerárquica.

### 2.3.1. Filtrado Anisotrópico de Vóxeles

Es posible obtener resultados más precisos durante el trazado de conos utilizando vóxeles direccionales o anisótropicos. Como fue explicada la generación de los niveles de detalle en la sección anterior solo se obtienen vóxeles isotrópicos, esto quiere decir que poseen el mismo

valor sin importar la dirección como son observados. Los problemas que puede ocasionar esta forma de representar los niveles de detalles, fueron explicados en la sección 1.8.4.5. Implementar filtrado direccional consiste en que cada vóxel debe almacenar seis canales, uno por cada eje direccional positivo y negativo. En nuestra implementación con texturas 3D esto se traduce en seis texturas 3D (una por cada dirección) a la mitad de la resolución del volumen original. Para realizar el filtrado anisotrópico de alto rendimiento se utiliza *compute shaders*.

## 2.4. Trazado de Conos con Vóxeles

El proceso de trazado de conos con vóxeles funciona de manera similar a marcha de rayos o *ray-marching*. La diferencia es que el volumen a muestrear por cada paso que da el rayo incrementa según la distancia al punto de origen del cono (ver Figura 32). La forma del cono es esencialmente una discretización de un grupo de rayos trazados desde un punto en una superficie. Para obtener las muestras de volúmenes cada vez más grande se utilizan los niveles de *mipmap* en la estructura de vóxeles.

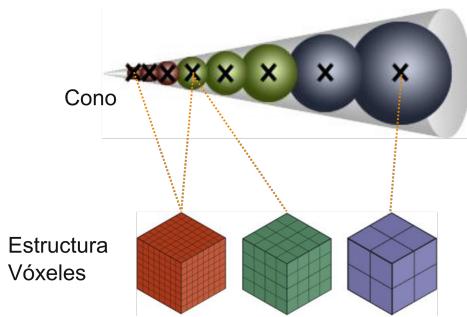


Figura 32: Descripción gráfica del trazado de conos con vóxeles e interpolación entre distintos niveles de detalle [28].

Nuestra implementación realiza trazado de conos con vóxeles de la misma manera que fue explicada en la sección 1.8.4, la mayor diferencia reside en el uso de texturas 3D. Una de las principales ventajas de utilizar texturas 3D es que estas aceleran considerablemente el muestreo de la estructura jerárquica de vóxeles. Mientras que con un *octree* cada vez que se desea trazar un cono el árbol tiene que ser recorrido de forma recursiva. Con texturas 3D esto se reduce a una simple instrucción proveída por OpenGL llamada *textureLod*. Esta función recibe una textura, una coordenada de muestreo y un nivel de *mipmap*. La función también realiza interpolación lineal entre los distintos niveles de *mipmapping* si la textura a muestrear así lo habilita.

### 2.4.1. Reflexión Difusa

La reflexión difusa de un fragmento puede ser aproximada utilizando integración Monte Carlo, trazando un número de conos sobre la semiesfera orientada por el vector normal del fragmento, y acumulando la radiancia a través del recorrido del cono. En nuestra implemen-

tación se utilizan seis conos difusos distribuidos de forma uniforme sobre la semiesfera (ver Figura 33) orientada por el vector normal del fragmento.

### 2.4.2. Oclusión Ambiental

La oclusión ambiental sobre un fragmento puede ser aproximada trazando conos sobre la semiesfera orientada por el vector normal de éste. Para la oclusión ambiental solo es relevante acumular información de oclusión. El cono ambiental es ponderado por una función  $f(r)$  donde su valor decrece según la distancia recorrida. En nuestra aplicación se utiliza la función:

$$f(r) = \frac{1}{1 + \lambda r} \quad (2.1)$$

donde  $r$  representa el radio del cono y  $\lambda$  una variable personalizada que describe la intensidad de declive según la distancia para el término de oclusión ambiental, básicamente la extensión del radio de oclusión.

### 2.4.3. Reflexión Especular

Nuestra implementación utiliza como modelo de iluminación la BRDF Blinn-Phong. Para obtener la reflexión especular es necesario solo un cono en equivalencia al lóbulo especular de la BRDF como se observa en la Figura 33. La apertura del cono especular depende del factor  $n$  en Blinn-Phong, mientras más alto es este valor menor es la apertura del cono especular.

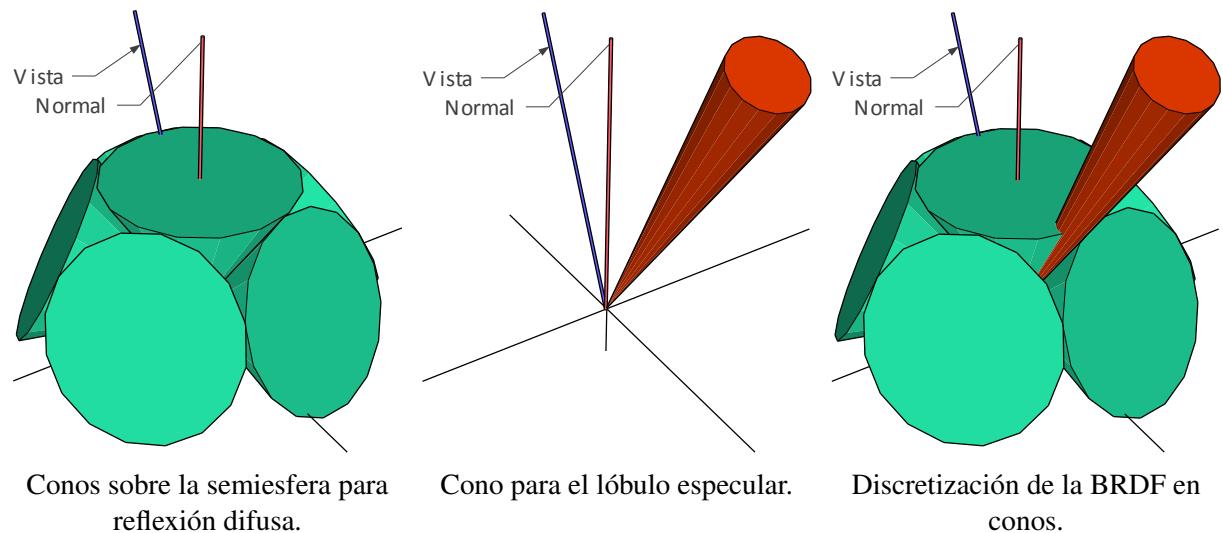


Figura 33: Ilustración de la distribución de los conos utilizados para representar la BRDF Blinn-Phong.

#### 2.4.4. Sombras Suaves

El trazado de conos contra véxeles también puede ser utilizado para realizar pruebas de oclusión sobre una superficie. En nuestra implementación se traza un cono desde la posición del fragmento en dirección opuesta a la dirección de la luz incidente. Para este cono se acumula la opacidad de los véxeles. La apertura del cono permite controlar el umbral de la sombra resultante. A mayor apertura más suave y esparcida es la sombra. En la Figura 34 se describe el trazado de este cono.

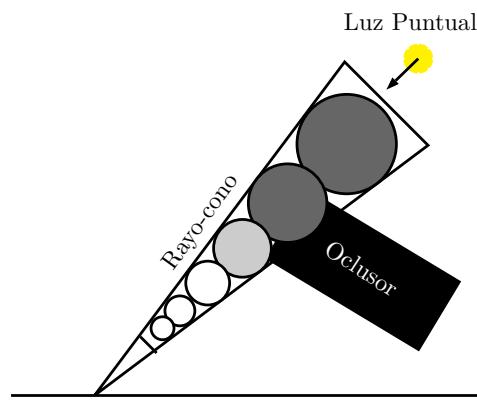


Figura 34: Descripción gráfica del recorrido de un cono empleado para sombreado de superficies.

### 2.5. Iluminación Global de Vóxeles

Almacenar solo la radiancia producto de la iluminación directa, permite obtener iluminación indirecta de un solo rebote durante el proceso de trazado de conos con véxeles. Esto provee buenos resultados visuales ya que el primer rebote es usualmente el que más contribuye en el transporte de luz de una escena.

Para la incorporación de un segundo rebote, nuestra implementación realiza trazado de conos dentro de la misma representación con véxeles utilizando *compute shaders*.

Luego que el proceso de sombreado de véxeles es completado y se filtran estos valores para generar véxeles anisótropos, se agrega otro paso para calcular el primer rebote de iluminación global sobre el volumen de radiancia. Similar al trazado de conos por fragmento explicado en la sección 1.8.4. Por cada véxel se trazan conos acumulando la radiancia incidente sobre el véxel. Este método solo comprende reflexión difusa ya que en esta propuesta no se almacena información especular durante el proceso de voxelización. Al finalizar el cálculo de la iluminación global sobre cada véxel se vuelve a realizar el proceso de filtrado anisotrópico.

El volumen resultante es utilizado durante la composición final de la imagen por el trazado de conos con véxeles, donde ahora estos conos acumulan radiancia producto de tanto iluminación directa como indirecta difusa.

## 2.6. Materiales Emisivos

La estructura de véxeles utilizada durante el trazado de conos almacena un valor de radiancia por cada véxel. Considerando esto, agregar materiales emisivos al proceso de voxelización es simple. Para la voxelización de materiales emisivos se utiliza otro volumen además de los ya existentes (albedo y normal) durante el proceso de voxelización. Este volumen almacena el promedio de emisión de los fragmentos que envuelven el véxel. El valor de estos véxeles es luego agregado al resultado de radiancia directa durante el proceso de sombreado de véxeles. Los materiales emisivos pueden ser utilizados para aproximar cualquier clase de superficie luminosa como luces de área, luces de neón, magma, pantallas digitales, etc (ver Figura 35).



Figura 35: Materiales emisivos en Unreal Engine 4.6 [29].

# Capítulo 3

## Implementación

### 3.1. Herramientas

Nuestra implementación para el cálculo de iluminación global en tiempo real está basada en la GPU. Esta aplicación genera gráficos por computadora utilizando rasterización en tiempo real acelerado por hardware gráfico. Para esto existen principalmente dos API gráficas, OpenGL y Direct3D. Para acceder a distintas características del hardware gráfico, rasterización y pipeline de renderizado se utilizó OpenGL junto al lenguaje de sombreado GLSL para el desarrollo de *shaders*.

La aplicación hace uso de características de OpenGL como lectura y escritura arbitraria y operaciones atómicas sobre texturas utilizando la extensión mencionada en la sección 2.1. Esta extensión es parte del núcleo de OpenGL desde la versión 4.2.

Otro aspecto importante de la aplicación es el uso de GPGPU para distintos cálculos de iluminación, filtrado y sombreado sobre datos almacenados en volúmenes. El uso de GPGPU permite realizar cálculo paralelo masivo utilizando núcleos dedicados en la GPU. Para ello existen distintas alternativas como CUDA y OpenCL o recientemente *compute shaders*. En nuestra aplicación se utilizaron *compute shaders* que provee OpenGL desde la versión 4.3. Estos *shaders* permiten cómputo general en la GPU con la sintaxis ya familiar de GLSL además de mayor compatibilidad en contraste con tecnologías como CUDA que solo están disponibles en tarjetas gráficas NVIDIA.

El lenguaje de desarrollo escogido fue C++ ya que este ofrece acceso directo al manejo de memoria similar al lenguaje C en el que está escrito OpenGL. Este acceso directo además ayuda a eliminar capas de abstracción sobre los datos que existen en otros lenguajes de más alto

nivel. Esto puede ayudar a mejorar el rendimiento de la aplicación lo cual es importante para técnicas de renderizado en tiempo real. Otra razón importante por la que se escoge este lenguaje es que muchas herramientas utilizadas para facilitar el desarrollo de aplicaciones gráficas están escritas en este lenguaje. El entorno de desarrollo fue Visual Studio 2015 Community Edition en el sistema operativo Windows 10.

OpenGL necesita acceder al contexto o framebuffer de una ventana instanciada en algún sistema operativo para poder renderizar. Para la creación, manejo y acceso al contexto de ventanas se utilizó *GLFW* versión 3.1.2. Para cargar las funciones de OpenGL, extensiones y enlazar el contexto se utilizó *GLEW* en su versión 1.13.0. Para OpenGL en C++ a pesar de que se puede acceder directamente a sus funciones en crudo se utilizó un wrapper llamado *OGLplus* en su versión 0.52.0. Este implementa una fachada orientada a objetos sobre OpenGL más acorde con el estilo de programación en C++. *OGLplus* provee administración automática de recursos y objetos de OpenGL, encapsulación y seguridad de tipos, manejo de errores e interoperabilidad con OpenGL en crudo. Para manejo de vectores y matrices además de otras funciones y operaciones matemáticas se utilizó la biblioteca *GLM* versión 0.9.7.

La aplicación hace uso de varios modelos, escenas tridimensionales y texturas. Para estos recursos existen distintos formatos. Implementar soporte para una variedad de ellos es una tarea ardua por esto se utilizaron bibliotecas en C++ que ya proveen esta funcionalidad. Para la carga de mallas 3D e información de escena como cámaras, luces, materiales, jerarquía de objetos y texturas se utilizó la biblioteca *Assimp* en su versión 3.2 por la variedad de formatos que soporta y las múltiples funciones que provee para pre-procesar los mallas e información de la escena. Para cargar datos en crudo de texturas se utilizó la biblioteca *FreeImage*.

Para facilitar el uso de la aplicación esta provee una interfaz gráfica con acceso a distintas funciones relevantes a la manipulación de objetos en escena y parámetros del algoritmo de iluminación global. Para esto se utilizó la biblioteca *dear imgui*, la cual provee distintas rutinas para la composición de interfaces de usuario, esta biblioteca utiliza el mismo pipeline de renderizado para dibujar la interfaz.

## 3.2. Arquitectura de la Aplicación

Detrás de la generación de frames se encuentra un sencillo motor gráfico compuesto por una multitud de clases extensibles. En esta sección se examina un poco la arquitectura de este. Sin embargo se hará énfasis detallado en las partes más relevantes al algoritmo de iluminación global.

Una escena en la aplicación está definida por cámaras, luces, materiales, texturas, mallas y un nodo raíz de jerarquía de escena. En la Figura 36 se muestra un diagrama para las clases que componen una escena.

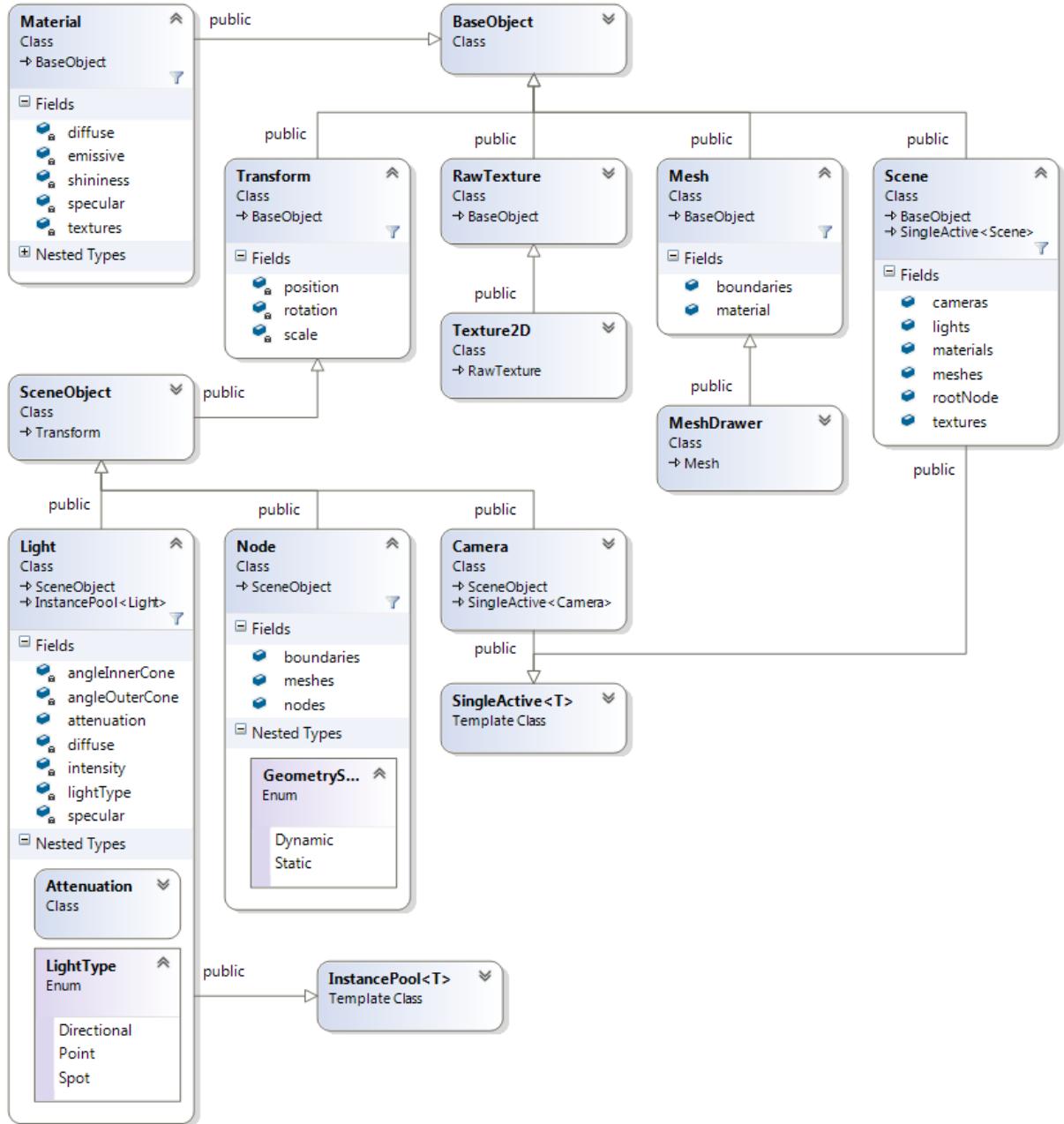


Figura 36: Diagrama de clases relevantes para la definición de escenas en el motor gráfico para el rendimiento de iluminación global.

Como se observa en el diagrama cada *Node* o nodo tiene un grupo *meshes* o mallas asociado. Cada una de estas *Mesh* o mallas tiene un material (clase *Material*) asignado, este ma-

terial puede o no tener uso de texturas. Los nodos en la escena puede definirse como estáticos o dinámicos, esta propiedad es importante para diferenciar entre voxelización estática y dinámica.

Se puede observar también que tanto nodos como mallas tienen un miembro *boundaries* o límites. Este miembro define una caja delimitante alineada a los ejes (AABB), utilizada para determinar si estos objetos se encuentran dentro del tronco cuadrado que define el volumen de proyección de la cámara. Esta técnica es llamada *frustum culling*.

Existen tres tipos de luces en nuestra implementación como ya fue mencionado anteriormente: direccionales, puntuales y focales. Una de las ventajas de realizar el sombreado a través de *compute shaders* es que se pueden colocar muchas luces en escena sin necesidad de crear mapas luz-vista por cada uno de estos.

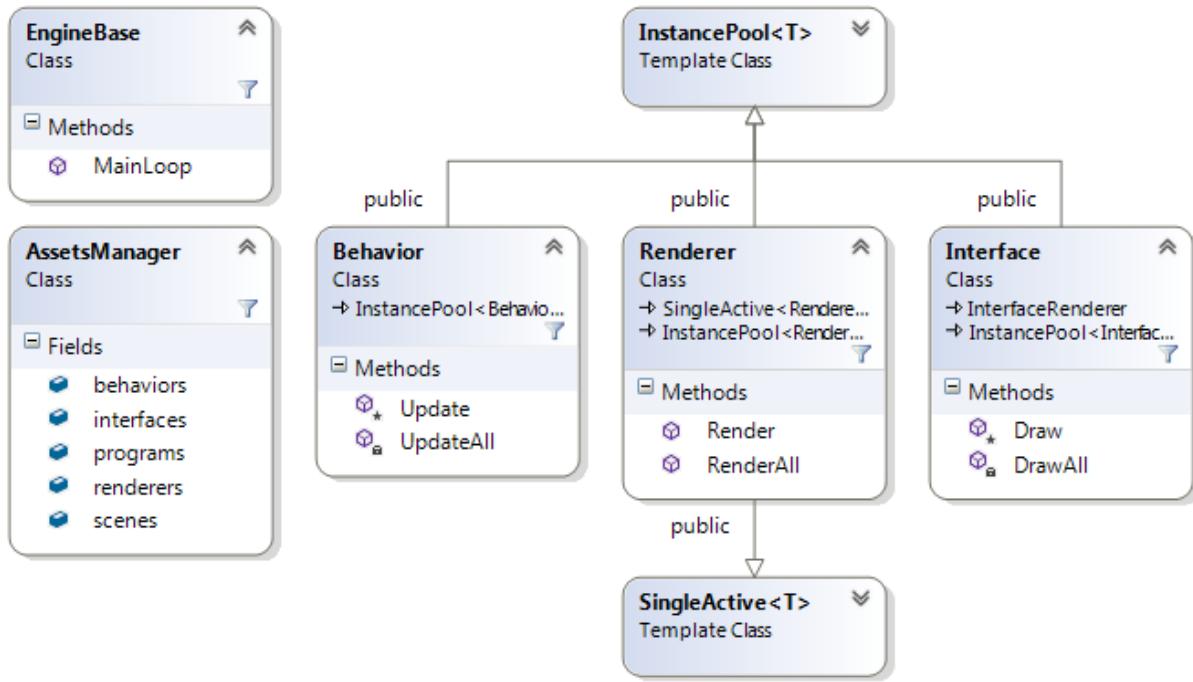


Figura 37: Diagrama de clases principales con métodos abstractos relacionados con la actualización de cada frame.

La aplicación utiliza abstracción en el *loop* o ciclo de renderizado. Existen tres clases bases que proveen un método abstracto a implementar por los hijos de estas. Las clases que implementan estos métodos son instanciadas en la clase *AssetsManager*. Por cada instancia de estas clases cada uno de estos métodos es llamado con cada iteración del ciclo de renderizado. Este ciclo incluye la actualización de interfaces, actualización de lógica y actualización de renderizado.

En nuestra implementación la voxelización, actualización del mapa de sombras y el cálculo de iluminación global son implementados en clases que heredan de *Renderer*. La clase *Interface* es la base de todas las interfaces en la aplicación. Mientras que la clase *Behavior* solo es utilizada para lógica de cámara libre flotante en escena.

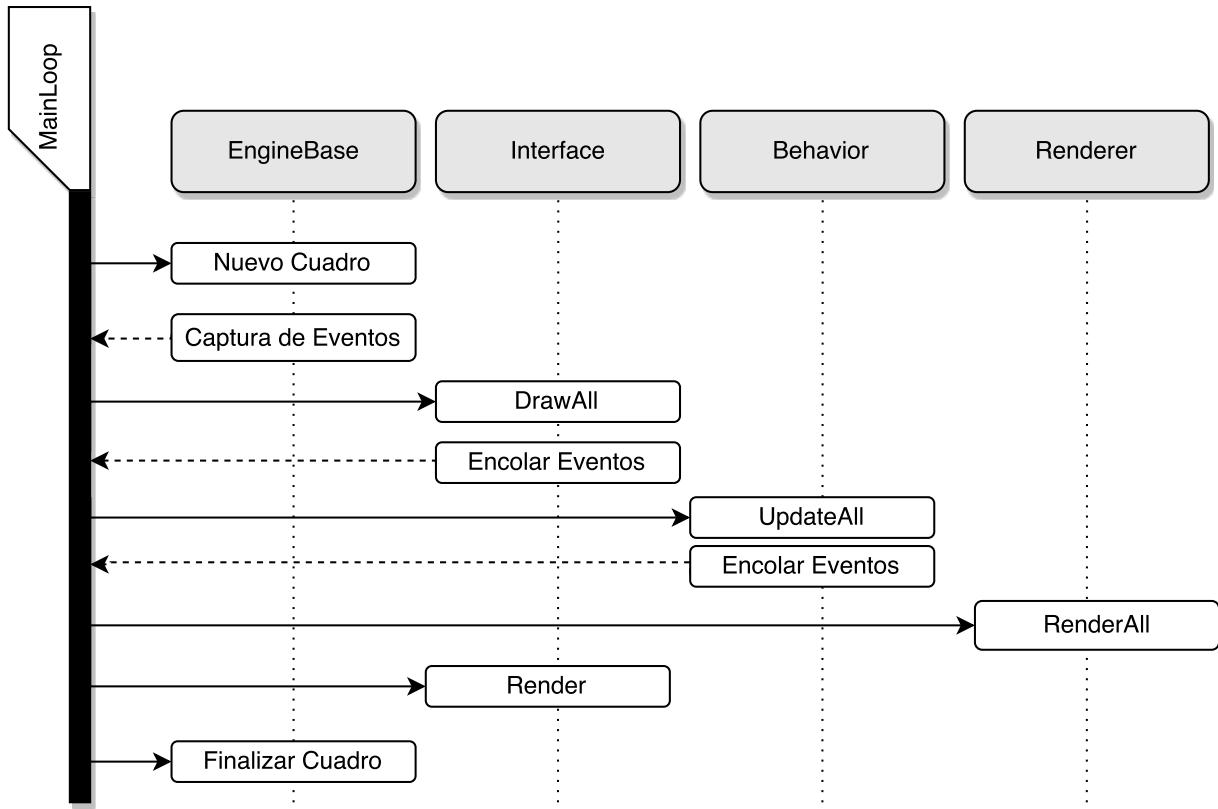


Figura 38: Descripción del flujo de ejecución del método *MainLoop* en donde reside el ciclo de renderizado.

Los métodos *Render*, *Draw* y *Update* son abstractos. Mientras que los métodos *RenderAll*, *UpdateAll* y *DrawAll* son métodos estáticos públicos accesibles por la clase *EngineBase*. Estos métodos estáticos invocan a todos las implementaciones de los métodos abstractos por cada clase que hereda de *Renderer*, *Interface* o *Behavior* respectivamente. El método *MainLoop* en *EngineBase* por tanto se abstrae de la implementación de estos métodos y simplemente invoca todas las implementaciones por cada nuevo frame. En la Figura 38 se observa el uso de los métodos estáticos para la composición de un frame.

### 3.3. Pipeline de Voxelización

El algoritmo de voxelización de escenas es implementado en una clase que hereda de la clase `Renderer`. En el método `Render`, residen la lógica por el lado de la CPU de voxelización, sombreado de véxeles, *mipmapping* para véxeles anisótropos e iluminación global de véxeles. En esta sección se describe en detalle el proceso de voxelización y su arquitectura.

#### 3.3.1. Arquitectura

La arquitectura de nuestro proceso de voxelización describe un pipeline, donde la entrada es geometría, materiales, texturas y matrices de proyección, y la salida es una serie de volúmenes de véxeles los cuales describen la discretización de esta geometría.

El algoritmo de voxelización se ejecuta entre el *geometry shader* y el *fragment shader*. En el *geometry shader* se proyecta y traslada los vértices de cada triángulo para generar un triángulo expandido. La rasterización conservativa de cada triángulo es tarea de ambos *shaders*. El *fragment shader* primeramente descarta fragmentos excedentes de la expansión del triángulo en el *geometry shader* para formar un polígono delimitante. La voxelización de geometría estática y dinámica se realiza sobre este mismo pipeline, el *fragment shader* debe garantizar que los véxeles estáticos no sean sobrescritos durante la voxelización dinámica, finalmente se realiza una operación de promedio sobre todos los atributos de cada fragmento que forma parte del espacio que envuelve un véxel, este valor es almacenado en cada véxel por atributo. En la Figura 39 se puede observar la composición de esta arquitectura.

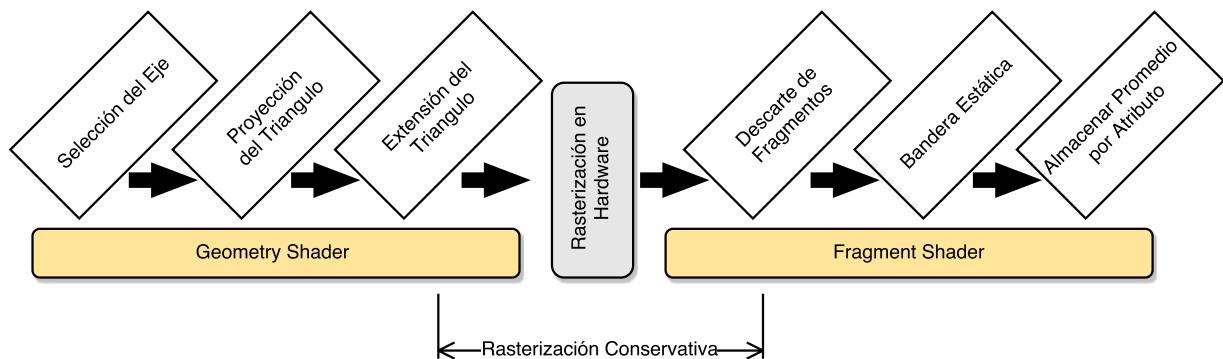


Figura 39: Descripción gráfica del pipeline de voxelización.

### 3.3.2. Voxelización Conservativa

Nuestra implementación utiliza una representación simplificada de la escena en véxels. Esta representación es generada de forma conservativa, por tanto si existe algún triángulo dentro del espacio de un véxel se generara un véxel por más pequeño que sea este triángulo.

#### 3.3.2.1. Matrices de Proyección por Eje

Como se explicó en la sección 2.1.1 cada triángulo debe ser proyectado sobre un eje direccional. El primer paso a realizar es definir estas matrices de proyección. Estas están definidas por una AABB uniforme. En nuestra implementación se toma la AABB que envuelve toda la escena por simplicidad. En el Código 1 se encuentra el algoritmo de la función que realiza esto. Este método se invoca cada vez que se carga una escena o cuando se cambia la resolución de la representación en véxeles.

La AABB puede estar definida de dos formas: por un punto que indica el centro de la caja y un vector extensión que indica la longitud media de la caja en cada eje a partir del centro o por un punto mínimo y un punto máximo.

```
void VoxelizerRenderer::UpdateProjectionMatrices(const BoundingBox &sceneBox)
{
    // longitud del aabb en cada eje
    auto axisSize = sceneBox.Extent() * 2.0f;
    // centro del aabb
    auto &center = sceneBox.Center();
    // tamaño de la cuadrícula de véxeles
    volumeGridSize = glm::max(axisSize.x, glm::max(axisSize.y, axisSize.z));
    // tamaño de un véxel en la cuadrícula
    voxelSize = volumeGridSize / volumeDimension;
    auto halfSize = volumeGridSize / 2.0f;
    // proyección ortogonal
    auto projection = glm::ortho(-halfSize, halfSize, -halfSize, halfSize, 0.0f, volumeGridSize);
    // matrices de vista por cada eje
    viewProjectionMatrix[0] = lookAt(center + glm::vec3(halfSize, 0.0f, 0.0f), center, glm::vec3(0.0f, 1.0f, 0.0f));
    viewProjectionMatrix[1] = lookAt(center + glm::vec3(0.0f, halfSize, 0.0f), center, glm::vec3(0.0f, 0.0f, -1.0f));
    viewProjectionMatrix[2] = lookAt(center + glm::vec3(0.0f, 0.0f, halfSize), center, glm::vec3(0.0f, 1.0f, 0.0f));
    // multiplicación de ambas matrices para obtener la matriz de proyección para los triángulos
    for (auto &matrix : viewProjectionMatrix) { matrix = projection * matrix; }
}
```

---

Código 1: Creación de matrices de proyección ortogonal por cada eje direccional

En el Código 1 se observa que primero se obtiene la longitud total en cada eje que define la AABB que envuelve la escena. Luego en la misma función se actualizan dos variables: *volumeGridSize* que indica el tamaño de la cuadrícula tridimensional de vóxeles en espacio de mundo y *voxelSize* que indica el tamaño de cada voxel dentro de esta cuadrícula.

El tronco de proyección es un cubo uniforme por tanto se utiliza la mayor longitud según cada eje. Esto asegura que desde cualquier eje direccional la proyección abarca toda la escena. De esta longitud se extrae la matriz de proyección ortogonal. Luego se configura cada matriz de vista por cada eje direccional. Finalmente se almacena la multiplicación de ambas matrices.

Una vez obtenidas las matrices de proyección el algoritmo procede a voxelizar la escena. El proceso de voxelización estática y dinámica utilizan el mismo programa de sombreado o *shader*, la diferencia reside sobre cual geometría es enviada al pipeline de rasterización, y durante el *fragment shader* si se lee o se escribe sobre una textura que indica las posiciones de vóxeles estáticos.

### 3.3.2.2. Selección del Eje Dominante

Para maximizar el área de voxelización y generar la mayor cantidad de fragmentos posibles cada triángulo es proyectado sobre uno de los ejes direccionales (ver Figura 40). Este eje se escoge según el vector normal definido por el plano formado por los tres vértices del triángulo. Este proceso se realiza en el *geometry shader*. En el *geometry shader* se pueden realizar operaciones sobre los vértices generados por el procesador de vértices o *vertex shader*. Esto es de particular interés para el proceso de voxelización conservativa, ya que como fue explicado anteriormente cada vértice de un triángulo necesita ser expandido.

```
int CalculateAxis()
{
    // normal del plano formado por vertices del triángulo
    vec3 p1 = gl_in[1].gl_Position.xyz - gl_in[0].gl_Position.xyz;
    vec3 p2 = gl_in[2].gl_Position.xyz - gl_in[0].gl_Position.xyz;
    vec3 faceNormal = cross(p1, p2);

    // valor por eje direccional
    float nDX = abs(faceNormal.x);
    float nDY = abs(faceNormal.y);
    float nDZ = abs(faceNormal.z);
```

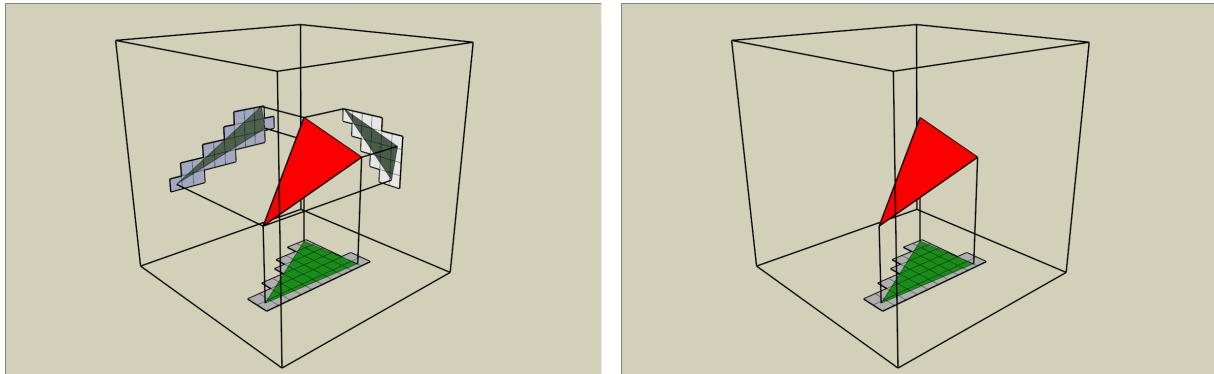
```

// índice para la matriz de proyección
if( nDX > nDY && nDX > nDZ )
{
    return 0;
}
else if( nDY > nDX && nDY > nDZ )
{
    return 1;
}
else
{
    return 2;
}
}

```

Código 2: Selección del eje dominante para la proyección ortogonal.

En el Código 2 primero se obtiene el vector normal del triángulo. El arreglo *gl\_in* contiene información de todos los vértices generados por el vertex shader. Para la aplicación esto es forzado a solo triángulos, por tanto la longitud de *gl\_in* siempre es tres. Luego según el peso en cada eje del vector normal del plano formado por el triángulo se escoge un eje direccional. Esto se retorna como un número entero. Este número indica cuál de las matrices de proyección generadas en la sección anterior va ser utilizada para proyectar cada vértice del triángulo.



Tres direcciones potenciales de proyección.

Según el vector normal, la proyección al eje Y es lo indicado.

Figura 40: Descripción gráfica del proceso de selección del eje de proyección y maximización de fragmentos [30].

### 3.3.2.3. Extensión del Triángulo y Polígono Delimitante

Una vez proyectados todos los vértices del triángulo según el eje direccional escogido, (esto es multiplicar cada vértice por la matriz escogida) entonces se procede a expandir los vér-

tices de este triángulo. Antes de esto primero es importante definir una AABB para el triángulo. Esta área delimitante es utilizada en el *fragment shader* para descartar fragmentos excedentes del triángulo expandido. El punto mínimo y máximo se expanden según la longitud diagonal de un píxel, en nuestra implementación este valor es  $\frac{1}{V_{res}}$  donde  $V_{res}$  representa la resolución del volumen. El Código 3 expone como se genera este AABB. El arreglo *pos* contiene los vértices del triángulo proyectado.

```
vec4 AxisAlignedBoundingBox(vec4 pos[3], vec2 pixelDiagonal)
{
    vec4 aabb;
    // punto mínimo y máximo para definir el aabb
    aabb.xy = min(pos[2].xy, min(pos[1].xy, pos[0].xy));
    aabb.zw = max(pos[2].xy, max(pos[1].xy, pos[0].xy));
    // se extiende el aabb por la longitud diagonal de un píxel
    aabb.xy -= pixelDiagonal;
    aabb.zw += pixelDiagonal;
    return aabb;
}
```

Código 3: Creación de un AABB para el triángulo proyectado.

Luego en el Código 4, primero se calculan los planos perpendiculares al triángulo formados por cada par de vértices del triángulo. Estos planos son trasladados medio píxel hacia afuera con respecto al triángulo proyectado.

```
vec2 halfPixel = vec2(1.0f / volumeDimension);
// cálculo de planos perpendiculares al triángulo
vec3 planes[3];
planes[0] = cross(pos[0].xyw - pos[2].xyw, pos[2].xyw);
planes[1] = cross(pos[1].xyw - pos[0].xyw, pos[0].xyw);
planes[2] = cross(pos[2].xyw - pos[1].xyw, pos[1].xyw);
planes[0].z -= dot(halfPixel, abs(planes[0].xy));
planes[1].z -= dot(halfPixel, abs(planes[1].xy));
planes[2].z -= dot(halfPixel, abs(planes[2].xy));
```

Código 4: Planos por cada par de vértices del triángulo proyectado.

Luego se calcula la intersección entre los planos perpendiculares al triángulo como se muestra en el Código 5.

```
// puntos de intersección entre los planos perpendiculares
vec3 intersection[3];
intersection[0] = cross(planes[0], planes[1]);
intersection[1] = cross(planes[1], planes[2]);
intersection[2] = cross(planes[2], planes[0]);
intersection[0] /= intersection[0].z;
```

```

intersection[1] /= intersection[1].z;
intersection[2] /= intersection[2].z;

```

Código 5: Intersección entre planos perpendiculares al triángulo proyectado.

En el Código 6 finalmente se obtienen los vértices del triángulo expandido al dilatar cada vértice según el plano del triángulo y los puntos de intersección.

```

vec4 trianglePlane;
// plano del triángulo proyectado en formato normal-distancia
trianglePlane.xyz = cross(pos[1].xyz - pos[0].xyz, pos[2].xyz - pos[0].xyz);
trianglePlane.xyz = normalize(trianglePlane.xyz);
trianglePlane.w = -dot(pos[0].xyz, trianglePlane.xyz);
// dilatación de los vértices del triángulo
float z[3];
z[0] = (-intersection[0].x * trianglePlane.x - intersection[0].y * trianglePlane.y -
         trianglePlane.w) / trianglePlane.z;
z[1] = (-intersection[1].x * trianglePlane.x - intersection[1].y * trianglePlane.y -
         trianglePlane.w) / trianglePlane.z;
z[2] = (-intersection[2].x * trianglePlane.x - intersection[2].y * trianglePlane.y -
         trianglePlane.w) / trianglePlane.z;
// posición final de los tres vértices del triángulo expandido
pos[0].xyz = vec3(intersection[0].xy, z[0]);
pos[1].xyz = vec3(intersection[1].xy, z[1]);
pos[2].xyz = vec3(intersection[2].xy, z[2]);

```

Código 6: Vértices del triángulo expandido.

Este nuevo triángulo expandido es enviado al pipeline de rasterización donde cada fragmento generado será procesado por el *fragment shader*. En el *fragment shader* se almacenarán los datos de la escena sobre cada volumen de véxeles.

### 3.3.2.4. Composición de Fragmentos y Vóxeles

Este proceso se realiza en el *fragment shader*. En la sección anterior se mencionó la creación de una AABB para el triángulo expandido. Lo primero a realizar durante la composición de fragmentos es descartar los fragmentos excedentes del triángulo:

```

// posición del fragmento debe encontrarse dentro del aabb
if( In.position.x < In.triangleAABB.x || In.position.y < In.triangleAABB.y ||
    In.position.x > In.triangleAABB.z || In.position.y > In.triangleAABB.w )
{
    discard;
}

```

Código 7: Descarte de fragmentos excedentes en el *fragment shader*.

En el Código 7 se compara la posición del fragmento con los límites del AABB, si este se encuentra fuera de la AABB del triángulo entonces este fragmento es descartado. Esto resulta finalmente en un polígono delimitante ya descrito en la Figura 27.

Una vez finalizada la rasterización conservativa de cada triángulo, se procede a almacenar la información de la escena en texturas tridimensionales. En nuestra aplicación se almacena en vértices el albedo, normal, y emisión de la geometría en escena. Un vértice puede envolver a varios fragmentos y cada uno de estos fragmentos puede tener distintos valores para los atributos mencionados. Una forma simple de almacenar todos estos atributos es utilizar un promedio de todos los valores que envuelven el vértice.

Como se mencionó en la sección 2.1.2: cada fragmento es tratado como un hilo individual, por tanto no hay forma de saber el orden como se envían fragmentos a la posición de un vértice. La extensión *GL\_ARB\_shader\_image\_load\_store* provee instrucciones atómicas para solventar esto. La forma más simple de obtener un promedio consiste en utilizar la función *imageAtomicAdd* y luego dividir por la cantidad de fragmentos. Para mantener un conteo de estos fragmentos se necesita un contador por cada vértice. La idea es utilizar el canal alfa en un formato RGBA como contador.

Sin embargo todas las funciones atómicas en esta extensión solo están reservadas para imágenes con formato entero de 32 bits. Generalmente el formato apropiado para estos valores sería RGBA8 o RGBA16F. Por tanto la función *imageAdd* no puede ser utilizada para este propósito.

Se utiliza entonces la función *atomicCompSwap* para emular la adición atómica. La idea es iterar por cada escritura sobre el volumen hasta que ya no haya más conflictos entre fragmentos, y el valor actual del vértice no ha sido cambiado por otro hilo.

El promedio en cada vértice debe ser calculado de forma incremental. Con el formato RGBA8 solo 8 bits están disponibles por canal. Por tanto es sencillo sobrepasar el valor límite de este formato mientras se suman valores. Para hacer esto se utiliza la siguiente fórmula:

$$C_{i+1} = \frac{iC_i + x_i + 1}{i + 1} \quad (3.1)$$

El Código 8 expone este procedimiento y la conversión entre formato RGBA8 y R32UI, sobre este último formato es posible realizar operaciones atómicas.

```
// conversión de formato de entero en r32ui a vec4
vec4 convRGBA8ToVec4(uint val)
{
```

```

        return vec4(float((val & 0x000000FF)),
        float((val & 0x0000FF00) >> 8U),
        float((val & 0x00FF0000) >> 16U),
        float((val & 0xFF000000) >> 24U));
    }
    // conversion de formato vec4 a entero r32ui
    uint convVec4ToRGBA8(vec4 val)
    {
        return (uint(val.w) & 0x000000FF) << 24U |
        (uint(val.z) & 0x000000FF) << 16U |
        (uint(val.y) & 0x000000FF) << 8U |
        (uint(val.x) & 0x000000FF);
    }
    // promedio con atomicidad
    void imageAtomicRGBA8Avg(layout(r32ui) volatile coherent uimage3D grid, ivec3 coords, vec4
    value)
    {
        // conversión de 0->1 a 0->255
        value.rgb *= 255.0;
        // valor equivalente en entero sin signo
        uint newVal = convVec4ToRGBA8(value);
        uint prevStoredVal = 0;
        uint curStoredVal;
        uint numIterations = 0;

        // mientras el valor almacenado sea diferente al actual y
        // no se sobrepase el numero máximo de iteraciones
        while((curStoredVal = imageAtomicCompSwap(grid, coords, prevStoredVal, newVal))
            != prevStoredVal
            && numIterations < 255)
        {
            prevStoredVal = curStoredVal;
            vec4 rval = convRGBA8ToVec4(curStoredVal);
            rval.rgb = (rval.rgb * rval.a); // De-normalizar
            vec4 curValF = rval + value;    // Agregar nuevo valor
            curValF.rgb /= curValF.a;      // Re-normalizar
            newVal = convVec4ToRGBA8(curValF);
            ++numIterations;
        }
    }
}

```

Código 8: Conversión entre RGBA8 y R32UI y promedio incremental.

La función *imageAtomicRGBA8Avg* es llamada por cada fragmento para almacenar el vector normal, el albedo y la emisión del promedio de todos los fragmentos sobre el espacio que envuelve un voxel, estos voxels se almacenan en texturas tridimensionales como se muestra en el Código 9.

---

```

// volúmenes utilizados para almacenar información de escena en voxels
layout(binding = 0, r32ui) uniform volatile coherent uimage3D voxelAlbedo;
layout(binding = 1, r32ui) uniform volatile coherent uimage3D voxelNormal;
layout(binding = 2, r32ui) uniform volatile coherent uimage3D voxelEmission;

```

```

void main()
{
    :
    // llamadas al promedio atómico por fragmento
    imageAtomicRGBAAvg(voxelNormal, position, normal);
    imageAtomicRGBAAvg(voxelAlbedo, position, albedo);
    imageAtomicRGBAAvg(voxelEmission, position, emissive);
    :
}

}

```

Código 9: Composición de fragmentos y vóxeles

### 3.3.2.5. Bandera Estática

El proceso de voxelización para geometría estática y dinámica es el mismo. Como se explicó en la sección 2.1.3 en nuestra implementación se evitó la creación de nuevos volúmenes para vóxeles estáticos y dinámicos. Para garantizar esto se utiliza un volumen con un solo componente entero, formato R8.

Durante la voxelización estática se almacena en este volumen un valor para marcar esta posición como estática (en este caso valor 1) como se observa en el Código 10.

```

layout(binding = 3, r8) uniform image3D staticVoxelFlag;

void main()
{
    :
    // durante voxelización estática se escribe la bandera sobre el volumen
    if(flagStaticVoxels)
    {
        imageStore(staticVoxelFlag, position, vec4(1.0));
    }
    :
}

```

Código 10: Escritura de la bandera estática durante voxelización de geometría estática

En contraste durante la voxelización dinámica se lee este volumen para descartar la escritura sobre vóxeles estáticos como se observa en el Código 11.

```

:
// durante voxelización dinámica se lee la bandera sobre el volumen
if(!flagStaticVoxels)
{
    bool isStatic = imageLoad(staticVoxelFlag, position).r > 0.0f;
}

```

```

    // si la posición del voxel es estática se descarta el fragmento cancelando la
    escritura
    if(isStatic) { discard; }
}
;

```

Código 11: Lectura de la bandera estática durante voxelización de geometría dinámica.

### 3.3.3. Re-voxelización y Limpieza de Volúmenes

Cada vez que se re-voxeliza la escena, los volúmenes utilizados durante el proceso de voxelización deben ser limpiados antes de usarse. Para la voxelización estática esto es sencillo, OpenGL provee una función llamada *glClearImage* desde la versión 4.4 la cual permite llenar una textura con un valor indicado. Debido a que se utiliza el mismo volumen para la voxelización dinámica, no se puede utilizar esta función durante la voxelización dinámica, ya que se eliminarían todos los voxels estáticos. Para solventar esto, antes de llamar al proceso de voxelización con la geometría dinámica, se limpian del volumen todos los voxels que no son estáticos utilizando un *compute shader*.

```

void main()
{
    int volumeDimension = imageSize(voxelAlbedo).x;

    if(gl_GlobalInvocationID.x >= volumeDimension ||
       gl_GlobalInvocationID.y >= volumeDimension ||
       gl_GlobalInvocationID.z >= volumeDimension) return;

    ivec3 writePos = ivec3(gl_GlobalInvocationID);
    // voxel vacío
    if(imageLoad(voxelAlbedo, writePos).a < EPSILON) { return; }
    // voxel es estático
    if(texelFetch(staticVoxelFlag, writePos, 0).r > EPSILON) { return; }
    // voxel no es estático entonces se procede a limpiar el volumen con ceros
    imageStore(voxelAlbedo, writePos, vec4(0.0));
    imageStore(voxelNormal, writePos, vec4(0.0));
    imageStore(voxelEmissive, writePos, vec4(0.0));
}

```

Código 12: Limpieza de voxels no estáticos.

En el Código 12 primero se verifica si ya el voxel está vacío, luego si este es estático y finalmente si este no lo es, se le coloca el valor cero en todos los canales de color y el alfa, esto indica que allí no hay un voxel.

### 3.4. Sombreado de Vóxeles

Para calcular la iluminación indirecta es necesario sombrear los véxeles. Cada voxel representa una simplificación de la radiancia saliente sobre un espacio en la escena. La textura tridimensional resultante del sombreado de véxeles sera llamada *textura base*. Esta información es utilizada por el algoritmo de trazado de conos para calcular la radiancia incidente sobre un fragmento.

En nuestra implementación la iluminación de los véxeles comprende solo el componente difuso. En el proceso de voxelización se puede observar que ningún volumen está relacionado al componente especular de un material. Esto debido a dos razones: primero memoria, ya que para incluir la especular durante el sombreado de véxeles tendría que crearse otro volumen, y segundo que usualmente el componente especular es un detalle fino de alta frecuencia. La voxelización siendo una simplificación de la escena podría perder detalle en especulares con lóbulos finos e incluso ocasionar errores de sombreado.

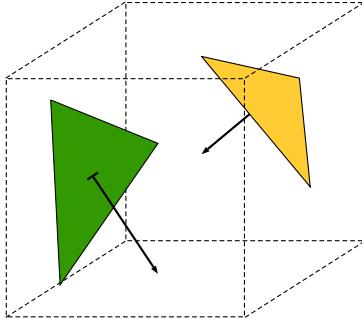
Para la iluminación de los véxeles se utilizan técnicas de iluminación estándar en computación gráfica. La BRDF de Lambert ya se encuentra almacenada en nuestro volumen albedo. Es necesario entonces multiplicar este valor por la atenuación normal. Para la ecuación de renderizado ignorando la emitancia esto resulta en la siguiente ecuación:

$$\begin{aligned} L(x \rightarrow \Theta) &= \int_{\Omega_x} f_r(x, \Psi \rightarrow \Theta) L(x \leftarrow \Psi) \cos(N_x, \Psi) d\omega_\Psi \\ &= \int_{\Omega_x} \frac{\rho}{\pi} L(x \leftarrow \Psi) \cos(N_x, \Psi) d\omega_\Psi \end{aligned} \tag{3.2}$$

El Código 13 expone este cálculo con solo iluminación directa para los véxeles.

```
vec3 BRDF(Light light, vec3 normal, vec3 albedo)
{
    // atenuación normal
    float nDotL = max(dot(normal, light.direction), 0.0f);
    // iluminación directa
    return light.diffuse * albedo * nDotL;
}
```

Código 13: Sombreado estándar para un voxel



Calcular el sombreado de véxeles de esta forma es suficiente para generar resultados de buena calidad durante el trazado de conos. Sin embargo uno de los problemas que surgen al simplificar las normales en véxeles, es que algunos véxeles tienen normales incorrectas o desviadas. Esto sucede especialmente cuando un véxel contiene geometría con normales muy disparejas (ver Figura 41).

Figura 41: Normales disparejas en un véxel.

Para disminuir este problema se calcula la atenuación normal por cada cara del véxel según la dirección de la luz  $\Psi$ . Luego se pondera cada atenuación normal por cara frontal con el peso de cada eje en el vector normal promedio del véxel. Este procedimiento se puede observar en el Código 14. En la Figura 42 se puede observar la composición para el sombreado final y la diferencia con Lambert tradicional. Además del método Lambert tradicional a este método se le denominó Lambert direccional ponderado.

```
vec3 BRDF(Light light, vec3 normal, vec3 albedo)
{
    // peso en cada eje para el vector normal del véxel
    vec3 weight = normal * normal;
    // atenuación normal por cada cara del véxel
    float rDotL = dot(vec3(1.0f, 0.0f, 0.0f), light.direction);
    float uDotL = dot(vec3(0.0f, 1.0f, 0.0f), light.direction);
    float fDotL = dot(vec3(0.0f, 0.0f, 1.0f), light.direction);
    // se encuentra la cara dominante según el vector normal
    rDotL = normal.x > 0.0f ? max(rDotL, 0.0f) : max(-rDotL, 0.0f);
    uDotL = normal.y > 0.0f ? max(uDotL, 0.0f) : max(-uDotL, 0.0f);
    fDotL = normal.z > 0.0f ? max(fDotL, 0.0f) : max(-fDotL, 0.0);
    // se hace un ponderado del sombreado por el peso
    nDotL = rDotL * weight.x + uDotL * weight.y + fDotL * weight.z;
    // iluminación directa
    return light.diffuse * albedo * nDotL;
}
```

Código 14: Sombreado direccional y ponderado según el vector normal para un véxel

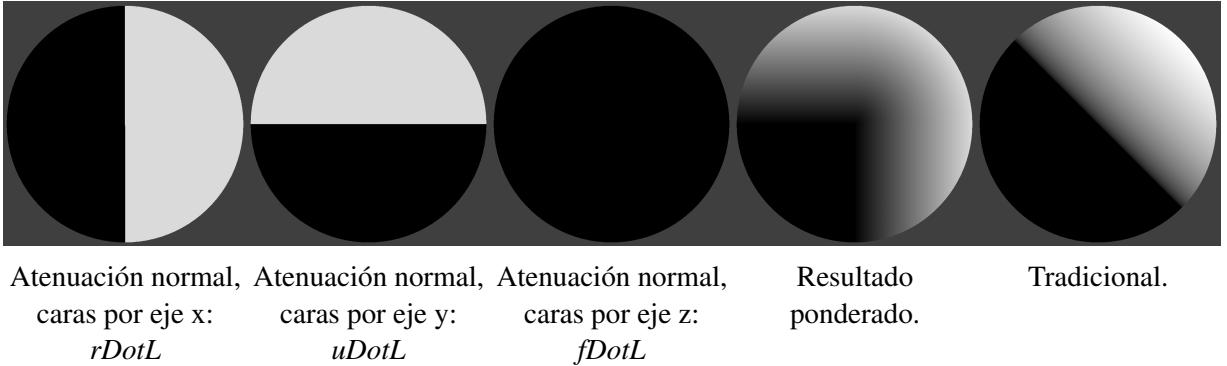


Figura 42: Composición gráfica para el Código 14 con un ángulo incidente  $\Theta$  y  $\Psi$  de  $\angle 45$  grados en BRDF Explorer [6].

### 3.4.1. Mapeo y Trazado de Sombras

Para obtener resultados coherentes es necesario ocluir véxeles. En nuestra implementación es posible utilizar mapeado de sombras solo para una luz direccional. Esta técnica es explicada en la sección 1.7.1. La posición a proyectar  $p_v$  es trasladada según el vector normal del véxel  $n_v$  por una distancia equivalente al tamaño medio de un véxel en espacio de mundo  $V_{wsSize}$ , esto es equivalente a la siguiente ecuación:  $p_v = p_v + n_v \cdot V_{wsSize} \cdot 0.5$ . Esto se hace para evitar problemas ya mencionados en la sección 2.2.1.

Una de las ventajas de este trabajo es la capacidad de incluir muchas luces con iluminación global en escena. Por tanto es necesario alguna forma de incluir sombras para otros tipos de luces y más de una fuente de luz. Nuestra implementación realiza trazado de rayos sobre el volumen para incorporar sombras para cualquier tipo de luz dentro del área de proyección del volumen de véxeles.

El algoritmo para detectar oclusión sobre un véxel es simple *raycasting* de un solo rayo como prueba de oclusión. Por cada fuente de luz con trazado habilitado se inicia un rayo desde la posición del véxel a iluminar con la dirección de la luz, si este rayo colisiona con algún otro véxel, entonces este véxel está ocluido.

Nuestra implementación también incluye una variación de esta técnica donde en vez de detener el rayo al colisionar, se acumulan valores a través del recorrido del rayo, estos valores decrecen según la distancia recorrida. Esto se hace para explotar el hecho de que un rayo trazado desde la posición a iluminar con la dirección de luz, tendrá más colisiones al pasar por los bordes de algún objeto voxelizado visto desde esta fuente de luz (Figura 31). El propósito de esto es generar valores más claros en los bordes de la sombra para aproximar sombras suaves. El Código 15 expone el trazado de rayos sobre el volumen para pruebas de oclusión utilizando

ambas técnicas:

```
float TraceShadow(vec3 position, vec3 direction, float maxTracingDistance)
{
    // tamaño de un voxel en espacio de textura
    float voxelTexSize = 1.0f / volumeDimension;
    // se empieza el recorrido un voxel hacia adelante para evitar auto-colisión
    float dst = voxelTexSize * 2.0f;
    vec3 samplePos = direction * dst + position;
    // visibilidad del voxel
    float visibility = 0.0f;

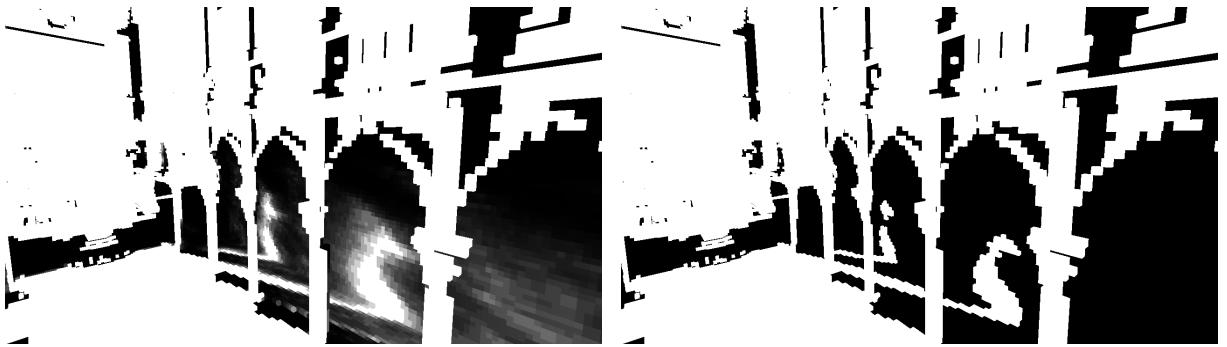
    while (visibility <= 1.0f && dst <= maxTracingDistance)
    {
        if (samplePos.x < 0.0f || samplePos.y < 0.0f || samplePos.z < 0.0f
            || samplePos.x > 1.0f || samplePos.y > 1.0f || samplePos.z > 1.0f)
        { break; }

        // peso de colisión si hay algún voxel
        float traceSample = ceil(texture(voxelAlbedo, samplePos).a) * traceShadowHit;
        // sombras duras si el valor es 1
        if(traceSample > 1.0f - EPSILON) { return 0.0f; }
        // acumulación de opacidad dividida por la distancia
        visibility += (1.0f - visibility) * traceSample / dst;
        // se avanza un voxel hacia adelante
        dst += voxelTexSize;
        samplePos = direction * dst + position;
    }

    return 1.0f - visibility;
}
```

Código 15: Trazado de rayos sobre volumen albedo para sombras sobre voxels

El valor de *traceShadowHit* es definido por el usuario. Este controla el peso de cada colisión del rayo con algún voxel. Si este valor es 1, esto equivale a detener el rayo apenas se encuentra una colisión. La acumulación de valores se hace de manera *front-to-back*, el peso de los valores acumulados disminuye con la distancia *dst* recorrida del rayo.



Con trazado utilizando acumulación de colisiones.

Deteniendo el rayo de oclusión apenas colisiona.

Figura 43: Valor de oclusión almacenado en el componente alfa del volumen de normales.

El valor de oclusión promedio por cada luz que traza sombras es almacenado en el componente alfa del volumen de normales. Esta información puede ser utilizada para realizar mapeado de sombras con una textura tridimensional. Este componente de la textura será llamado volumen de visibilidad, en la Figura 43 se puede observar su contenido. La proyección de la posición en el volumen es mucho más sencilla que en el mapeo de sombras tradicional. Sin embargo la calidad es mucho más baja ya que el volumen es de baja resolución. Para proyectar la posición de un voxel en espacio de mundo y viceversa se utilizan las dos funciones expuestas en el Código 16.

```
// de espacio de textura a espacio de mundo
vec3 VoxelToWorld(vec3 pos)
{
    vec3 result = pos;
    result *= voxelSize;
    return result + worldMinPoint;
}
// de espacio de mundo a espacio de textura
vec3 WorldToVoxel(vec3 position)
{
    vec3 voxelPos = position - worldMinPoint;
    return voxelPos * voxelScale;
}
```

Código 16: Transformación de espacio entre coordenadas de textura y posiciones de mundo.

Estas funciones son utilizadas en varias partes del algoritmo en su totalidad. *VoxelToWorld* convierte una coordenada en espacio de textura a una posición en espacio de mundo, mientras que *WorldToVoxel* hace lo contrario. La variable *voxelSize* representa el tamaño de un voxel dentro de la cuadrícula de voxels, *voxelScale* representa la escala de la cuadrícula de voxels, *worldMinPoint* es el mínimo punto de la *AABB* que comprende el volumen de voxelización.

### 3.4.2. Vóxeles Emisivos

La forma como se almacenan materiales emisivos en la representación con vóxeles es simple. Al culminar el sombreado de un vóxel, al color resultante se le suma el valor de emisión obtenido del proceso de voxelización. El propósito de esto es aproximar de forma muy cruda el valor de  $L_e(x \rightarrow \Theta)$ . Este describe la luz emitida por un punto  $x$  sobre una superficie. Entonces continuando la ecuación 3.2 ahora considerando emisión tenemos:

$$L(x \rightarrow \Theta) = L_e(x \rightarrow \Theta) + \int_{\Omega_x} \frac{\rho}{\pi} L(x \leftarrow \Psi) \cos(N_x, \Psi) dw_\Psi \quad (3.3)$$

## 3.5. Estructura Jerárquica

La representación de la escena en vóxeles puede ser filtrada hacia niveles de menor detalle utilizando distintos niveles de *mipmap*. Esto describe una estructura piramidal la cual es utilizada para acelerar el trazado de conos contra vóxeles, donde según la apertura del cono se interpola entre distintos niveles de detalle como se observa en la Figura 32.

En nuestra implementación se utilizan seis nuevos volúmenes por cada eje direccional positivo y negativo para obtener vóxeles anisótropos. Cada uno de estos volúmenes tiene una resolución de  $V_{res}/2$ . Los niveles de *mipmapping* no se encuentran en la textura base de radiancia sino en estos volúmenes, por tanto estos volúmenes son los que residen los distintos niveles de detalle de la escena voxelizada (ver Figura 44). A estos volúmenes se les llamará *volúmenes direccionales*.

Como se utilizan texturas tridimensionales en vez de un *octree* el proceso de filtrado es mucho más simple ya que la interpolación cuadrilínea es soportada nativamente. Un problema surge para la interpolación entre la textura base y los niveles de *mipmapping* en los volúmenes direccionales. Ya que estas son texturas separadas no hay interpolación lineal entre la textura base y los volúmenes direccionales. Esto es simple de solventar utilizando la función *mix()* de GLSL cuando el nivel de detalle se encuentre entre cero y uno, cero siendo el máximo detalle (textura base) y uno el nivel cero del *mipmap* en los volúmenes direccionales.

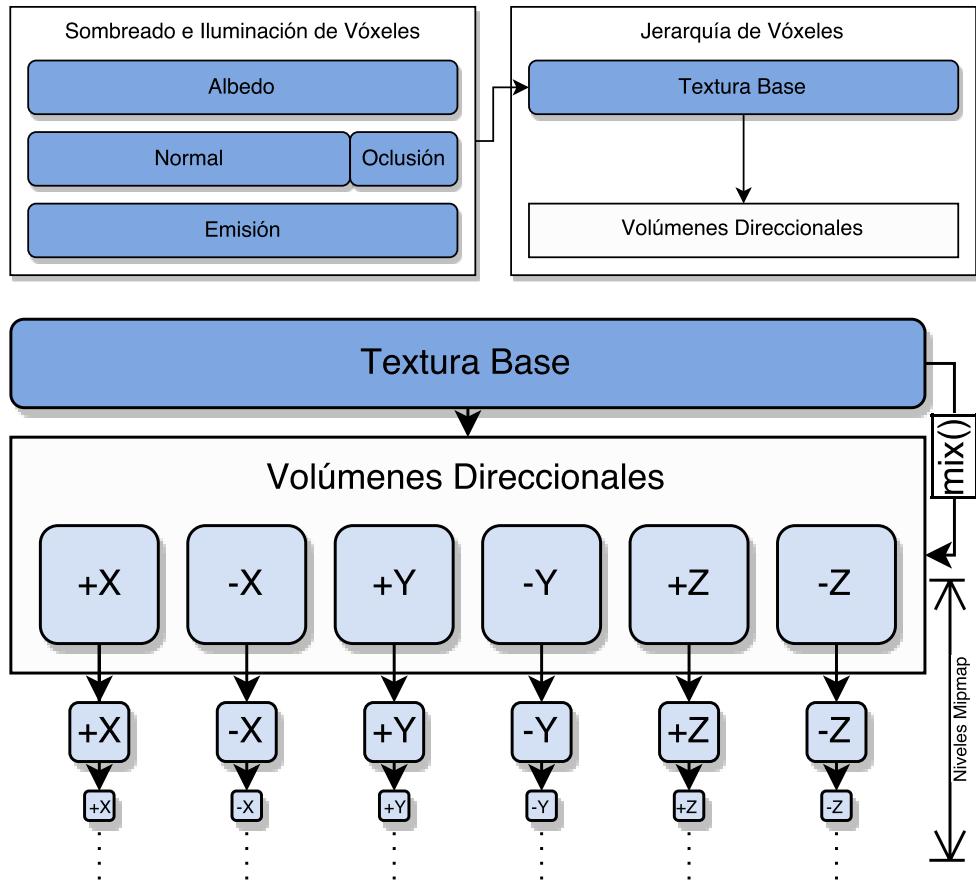


Figura 44: Descripción gráfica de la estructura jerárquica de véxeles utilizada para el trazado de conos. Arriba se observa el origen de la textura base y abajo la relación de la textura con los volúmenes direccionales.

### 3.5.1. Vóxeles Anisótropicos

El proceso para generar véxeles anisótropicos es explicado de forma general en la sección 1.8.4.5. Una representación con véxeles anisótropicos provee mayor calidad visual y precisión durante el trazado de conos. Cada cono trazado tiene una dirección, la idea es utilizar esta dirección para saber cuáles volúmenes direccionales van a utilizarse para interpolar. Para una dirección arbitraria se representa como tres volúmenes que serían las caras frontales según esta dirección del véxel simplificado. La dirección del cono tiene un peso en cada eje direccional, por tanto los valores de cada uno de estos volúmenes direccionales deben ser ponderados al momento de muestrearse. Una representación isotrópica no tiene concepto de direccionalidad del cono, por tanto para algunos casos esto puede generar una serie de problemas visuales ya mencionados. Las desventajas de esta implementación es que el trazado es más lento ya que se deben realizar tres muestras por cada paso del cono y mayor consumo de memoria.

Una forma simplificada de observar este proceso es con texturas bidimensionales. Suponiendo que se tiene una cuadrícula de  $5^2$  la cual se quiere reducir a un nivel de detalle más bajo de  $3^2$  como se observa en la Figura 45.

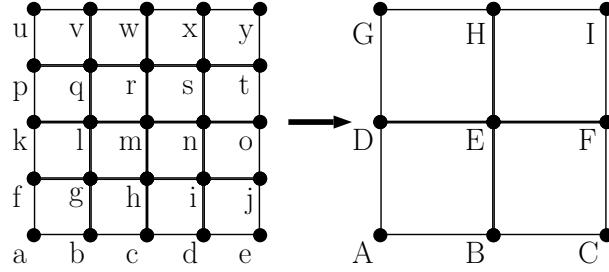


Figura 45: Cuadrícula de vértices de  $5^2$  a menor detalle  $3^2$ .

Para filtrar el valor del vértice  $E$  en la dirección  $X$  positiva es necesario considerar los nueve vértices cercanos en el nivel anterior de detalle. Estos vértices son divididos en grupos de cuatro como se observa en la Figura 46.

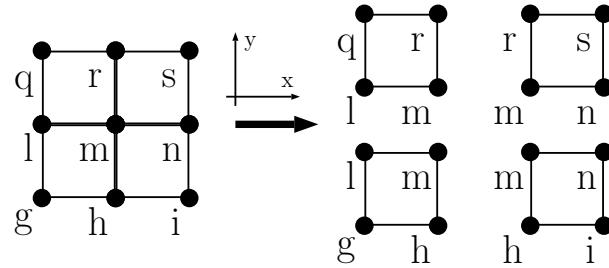


Figura 46: Separación de vértices cercanos a  $E$  en grupos a filtrar.

Por cada grupo se filtra en la dirección  $X$  positiva. Por ejemplo en el primer grupo, en la parte superior izquierda. Primero se realiza mezclado alfa o *alpha blending* en la dirección  $X$  positiva y luego se calcula un promedio para obtener el valor de este grupo. La siguiente ecuación describe este proceso:

$$\begin{aligned}
 \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix}_{q \rightarrow r} &= \begin{pmatrix} R \\ G \\ B \\ 1 \end{pmatrix}_r A_r + \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix}_q (1 - A_r) \\
 \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix}_{l \rightarrow m} &= \begin{pmatrix} R \\ G \\ B \\ 1 \end{pmatrix}_m A_m + \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix}_l (1 - A_m) \\
 \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix}_{q \rightarrow r, l \rightarrow m} &= \left[ \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix}_{q \rightarrow r} + \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix}_{l \rightarrow m} \right] \cdot 0.5
 \end{aligned} \tag{3.4}$$

Una vez se obtiene cada valor de los cuatro grupos se puede ahora calcular el valor del vóxel filtrado repitiendo este proceso con los cuatro valores resultantes. En nuestra implementación esto se realiza con un *compute shader*, el Código 17 expone este proceso sobre texturas tridimensionales.

```

// posiciones de vóxeles cercanos
const ivec3 anisoOffsets[] = ivec3[8]
{
    ivec3(1, 1, 1),
    ivec3(1, 1, 0),
    ivec3(1, 0, 1),
    ivec3(1, 0, 0),
    ivec3(0, 1, 1),
    ivec3(0, 1, 0),
    ivec3(0, 0, 1),
    ivec3(0, 0, 0)
};

// obtiene vóxeles cercanos
void FetchTexels(ivec3 pos, int dir, inout vec4 val[8])
{
    for(int i = 0; i < 8; i++)
    {
        val[i] = texelFetch(voxelMipmapSrc[dir], pos + anisoOffsets[i], mipLevel);
    }
}

void main()
{
    :
}

```

```

if(gl_GlobalInvocationID.x >= mipDimension.x ||
   gl_GlobalInvocationID.y >= mipDimension.y ||
   gl_GlobalInvocationID.z >= mipDimension.z) return;

// posición de escritura
ivec3 writePos = ivec3(gl_GlobalInvocationID);
// posición de lectura, mip de mayor detalle
ivec3 sourcePos = writePos * 2;
// Arreglo de valores cercanos a filtrar
vec4 values[8];
// obtiene valores del volumen -X
FetchTexels(sourcePos, 0, values);
// filtrado en dirección -X
imageStore(voxelMipmapDst[0], writePos,
(
    values[0] + values[4] * (1 - values[0].a) + // Mezclado Alfa
    values[1] + values[5] * (1 - values[1].a) +
    values[2] + values[6] * (1 - values[2].a) +
    values[3] + values[7] * (1 - values[3].a)) * 0.25f // Promedio
);
// obtiene valores del volumen +X
FetchTexels(sourcePos, 1, values);
// filtrado en dirección +X
imageStore(voxelMipmapDst[1], writePos,
(
    values[4] + values[0] * (1 - values[4].a) +
    values[5] + values[1] * (1 - values[5].a) +
    values[6] + values[2] * (1 - values[6].a) +
    values[7] + values[3] * (1 - values[7].a)) * 0.25f
);
// obtiene valores del volumen -Y
FetchTexels(sourcePos, 2, values);
// filtrado en dirección -Y
imageStore(voxelMipmapDst[2], writePos,
(
    values[0] + values[2] * (1 - values[0].a) +
    values[1] + values[3] * (1 - values[1].a) +
    values[5] + values[7] * (1 - values[5].a) +
    values[4] + values[6] * (1 - values[4].a)) * 0.25f
);
// obtiene valores del volumen +Y
FetchTexels(sourcePos, 3, values);
// filtrado en dirección +Y
imageStore(voxelMipmapDst[3], writePos,
(
    values[2] + values[0] * (1 - values[2].a) +
    values[3] + values[1] * (1 - values[3].a) +
    values[7] + values[5] * (1 - values[7].a) +
    values[6] + values[4] * (1 - values[6].a)) * 0.25f
);
// obtiene valores del volumen -Z
FetchTexels(sourcePos, 4, values);
// filtrado en dirección -Z
imageStore(voxelMipmapDst[4], writePos,
(

```

```

        values[0] + values[1] * (1 - values[0].a) +
        values[2] + values[3] * (1 - values[2].a) +
        values[4] + values[5] * (1 - values[4].a) +
        values[6] + values[7] * (1 - values[6].a)) * 0.25f
    );
    // obtiene valores del volumen +Z
    FetchTexels(sourcePos, 5, values);
    // filtrado en dirección +Z
    imageStore(voxelMipmapDst[5], writePos,
    (
        values[1] + values[0] * (1 - values[1].a) +
        values[3] + values[2] * (1 - values[3].a) +
        values[5] + values[4] * (1 - values[5].a) +
        values[7] + values[6] * (1 - values[7].a)) * 0.25f
    );
}

```

Código 17: Filtrado sobre los volúmenes direccionales para obtener véxeles anisótropos

Este proceso se repite por cada nivel del *mipmap* en los volúmenes direccionales. Estas son enlazadas según el nivel que va a ser filtrado con la instrucción *glBindImageTexture*. La variable *mipLevel* le indica a la instrucción *texelFetch* cual es el nivel anterior de detalle como fuente de filtrado. Cuando se filtra desde la textura base al nivel 0 de los volúmenes direccionales el proceso es similar. La diferencia reside en dos detalles: primero solo se hace una llamada a *FetchTexels* porque solo hay una textura fuente y segundo que el *mipLevel* es cero ya que la textura base no tiene niveles de *mipmap* y esta representa el máximo nivel de detalle de la estructura jerárquica.

### 3.6. Trazado de Conos con Vóxeles

El trazado de conos es utilizado para distintos aspectos visuales de la aplicación. En nuestra implementación este proceso se realiza durante el cálculo de iluminación utilizando sombreado diferido (sección 1.7.2). Utilizar sombreado diferido es muy conveniente ya que durante el cálculo de iluminación solo es necesario trazar conos sobre cada píxel en vez de cada fragmento incluyendo los no visibles.

Como se menciona en la sección 2.4. El trazado de conos es similar a *ray-marching* con la diferencia que el volumen a muestrear incrementa de tamaño según la distancia recorrida. Esto es producto de la expansión de la apertura del cono a través de su recorrido. Las muestras de mayor tamaño se obtienen utilizando los niveles de *mipmap* descritos en la sección anterior.

Al utilizar véxeles anisótropos se necesita saber que volúmenes direccionales van a

ser utilizados para muestrear a través del recorrido del cono. Esto se determina según el signo de cada eje del vector direccional del cono como se observa en el Código 18. Es además necesario calcular el peso de cada eje de este vector para obtener un resultado ponderado entre los tres volúmenes direccionales, este peso esta expresado por la variable *weight* en el Código 18.

```
// volúmenes direccionales
layout(binding = 8) uniform sampler3D voxelTexMipmap[6];
// trazado de cono contra véxeles
vec4 TraceCone(vec3 position, vec3 normal, vec3 direction, float aperture)
{
    // índice de los tres volúmenes direccionales
    uvec3 visibleFace;
    // selección de índices según el signo
    visibleFace.x = (direction.x < 0.0) ? 0 : 1;
    visibleFace.y = (direction.y < 0.0) ? 2 : 3;
    visibleFace.z = (direction.z < 0.0) ? 4 : 5;
    // peso por eje de la dirección del cono
    vec3 weight = direction * direction;
    :
}
```

Código 18: Lógica para determinar volúmenes direccionales a utilizar durante el trazado de conos y peso por eje.

Durante el recorrido del cono se utiliza la función de GLSL *textureLod*. Esta función tiene como entrada una textura, una coordenada y un nivel *mipmap*. Durante la marcha del cono el nivel *mipmap* se obtiene del diámetro del cono dado una distancia desde el origen con la siguiente ecuación:

$$V_{level} = \log_2 \left( \frac{d}{V_{size}} \right) \quad (3.5)$$

donde *d* es el diámetro del círculo del cono según la distancia recorrida y *V<sub>size</sub>* es el tamaño de un véxel en la cuadrícula de véxeles en el máximo nivel de detalle. El valor de *d* se puede obtener de la siguiente ecuación:

$$d = 2t \cdot \tan \left( \frac{\theta}{2} \right) \quad (3.6)$$

donde *t* es la distancia recorrida por el cono desde el punto de origen y  $\theta$  es el ángulo de apertura del cono. En la Figura 47 se observa una representación visual de este recorrido:

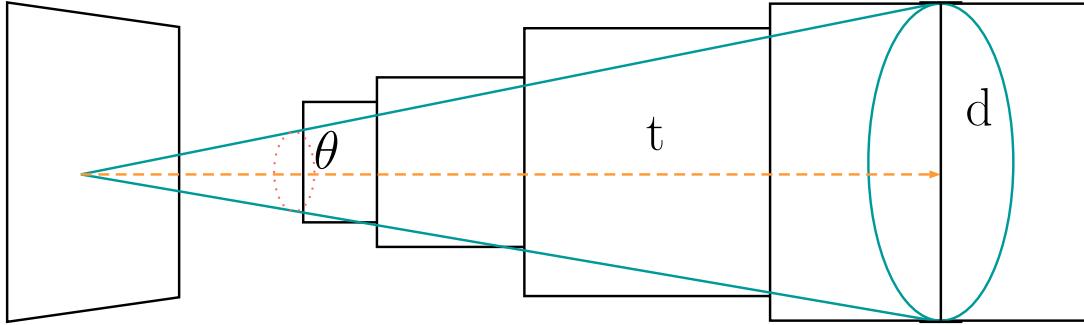


Figura 47: Visualización del recorrido de un cono.

En nuestra implementación el punto de origen del cono es trasladado según el tamaño de un voxel como se observa en el Código 19. Esto se hace para evitar que el cono colisione con el voxel de origen.

```
vec4 TraceCone(vec3 position, vec3 normal, vec3 direction, float aperture)
{
    :
    // distancia inicial recorrida de un voxel
    float dst = voxelWorldSize;
    // posición inicial
    vec3 startPosition = position + normal * dst;
    :
}
```

Código 19: Traslado de origen del cono.

Finalmente en el Código 20 se expone el resto del código para el trazado del cono a través de la escena.

```
// volumen de radiancia o textura base
layout(binding = 7) uniform sampler3D voxelTex;
// volúmenes direccionales
layout(binding = 8) uniform sampler3D voxelTexMipmap[6];
// obtiene una muestra de la representación en véxoles
vec4 AnisotropicSample(vec3 coord, vec3 weight, uvec3 face, float lod)
{
    // nivel mip en los volúmenes direccionales
    float anisoLevel = max(lod - 1.0f, 0.0f);
    // muestra direccional ponderada por el peso de cada eje
    vec4 anisoSample = weight.x * textureLod(voxelTexMipmap[face.x], coord, anisoLevel)
        + weight.y * textureLod(voxelTexMipmap[face.y], coord, anisoLevel)
        + weight.z * textureLod(voxelTexMipmap[face.z], coord, anisoLevel);

    // interpolación lineal con la textura base
    // cuando el nivel de detalle es menor a 1
```

```

    if(lod < 1.0f)
    {
        vec4 baseColor = texture(voxelTex, coord);
        anisoSample = mix(baseColor, anisoSample, clamp(lod, 0.0f, 1.0f));
    }

    return anisoSample;
}

vec4 TraceCone(vec3 position, vec3 normal, vec3 direction, float aperture)
{
    :
    // resultado de acumulación
    vec4 coneSample = vec4(0.0f);
    // máxima distancia para el recorrido de un cono
    float maxDistance = maxTracingDistanceGlobal / voxelScale

    while(coneSample.a < 1.0f && dst <= maxDistance)
    {
        vec3 conePosition = startPosition + direction * dst;
        // expansión del diámetro del cono
        float diameter = 2.0f * aperture * dst;
        // nivel de detalle según el diámetro
        float mipLevel = log2(diameter / voxelWorldSize);
        // conversión de posición en espacio de mundo a espacio de textura
        vec3 coord = WorldToVoxel(conePosition);
        // muestra anisotrópica
        vec4 anisoSample = AnistropicSample(coord, weight, visibleFace, mipLevel);
        // composición front-to-back
        coneSample += (1.0f - coneSample.a) * anisoSample;
        // aumento de la distancia recorrida
        dst += diameter * samplingFactor;
    }

    return coneSample;
}

```

Código 20: Trazado de cono con véxeles.

En el método *AnistropicSample* se puede observar el muestreo direccional utilizando los tres volúmenes direccionales. Cada muestra es ponderada por el peso de cada eje en la dirección del cono. También se puede observar el uso de la función *mix()* para interpolar entre los volúmenes direccionales y la textura base cuando el nivel de detalle es menor a 1 como se mencionó en la sección 3.5.

Durante el trazado se observa cómo se obtiene la posición del cono según la distancia recorrida y el uso de las operaciones ya mencionadas para obtener el diámetro y el nivel de *mipmap*. La función *WorldToVoxel* aparece para cambiar una posición en espacio de mundo a espacio de textura como fue mencionado para el Código 16. La distancia recorrida es alterada por una variable *samplingFactor* esto permite disminuir la distancia entre muestreos para lograr

resultados más suaves con un costo en rendimiento.

Para acumular los valores muestrados través del recorrido del cono se utiliza acumulación volumétrica *front-to-back*. Para esto es necesario llevar pista de un valor oclusión  $a$  y color  $c$ . La actualización de estos valores por cada paso con un nuevo valor  $a_2$  y  $c_2$  se realiza de la siguiente manera:  $c = c + (1 - a)c_2$  y  $a = a + (1 - a)a_2$ . En el algoritmo se puede observar que esto se realiza con la variable *coneSample*.

### 3.6.1. Reflexión Difusa

Para el cálculo de reflexión difusa se utilizan seis conos dentro de la semiesfera orientada por el vector normal como se describen en la Figura 33. El Código 21 demuestra el proceso de integración de los conos difusos.

```
// direcciones de conos difusos
const vec3 diffuseConeDirections[] =
{
    vec3(0.0f, 1.0f, 0.0f),
    vec3(0.0f, 0.5f, 0.866025f),
    vec3(0.823639f, 0.5f, 0.267617f),
    vec3(0.509037f, 0.5f, -0.7006629f),
    vec3(-0.50937f, 0.5f, -0.7006629f),
    vec3(-0.823639f, 0.5f, 0.267617f)
};

// pesos de conos difusos
const float diffuseConeWeights[] =
{
    PI / 4.0f,
    3.0f * PI / 20.0f,
    3.0f * PI / 20.0f,
};

vec4 CalculateIndirectLighting(vec3 position, vec3 normal, vec3 albedo, vec4 specular)
{
    // resultado de integración del trazado difuso
    vec4 diffuseTrace = vec4(0.0f);
    // dirección del cono
    vec3 coneDirection = vec3(0.0f);
    :

    // albedo es mayor a cero, si albedo es cero no hay radiancia
    if(any(greaterThan(albedo, diffuseTrace.rgb)))
    {
        // apertura del cono es la tangente del ángulo medio
        const float aperture = 0.57735f // tan(60/2);
        // vector guía para obtener vectores direccionales arbitrarios
        vec3 guide = vec3(0.0f, 1.0f, 0.0f);
```

```

    if (abs(dot(normal,guide)) == 1.0f)
    {
        guide = vec3(0.0f, 0.0f, 1.0f);
    }

    // obtener hacia la derecha y hacia arriba
    vec3 right = normalize(guide - dot(normal, guide) * normal);
    vec3 up = cross(right, normal);

    for(int i = 0; i < 6; i++)
    {
        // rotación de la dirección según cada dirección del cono difuso con vector normal
        coneDirection = normal;
        coneDirection += diffuseConeDirections[i].x * right + diffuseConeDirections[i].z *
            up;
        coneDirection = normalize(coneDirection);
        // se acumula el resultado de cada cono y se hace un ponderado por peso
        diffuseTrace += TraceCone(position, normal, coneDirection, aperture) *
            diffuseConeWeights[i];
    }

    diffuseTrace.rgb *= albedo;
}

:
}

```

Código 21: Conos para reflexión difusa.

Las direcciones, pesos de integración y tangente del ángulo ya se encuentran pre-calculados. Para rotar a lo largo del vector normal se obtienen vectores direccionales perpendiculares al vector normal utilizando un vector guía.

### 3.6.2. Reflexión Especular

Para la reflexión especular se utiliza un solo cono con la dirección  $R$  del lóbulo especular (sección 1.4.1.2) y una apertura adecuada a la potencia especular  $n$  de un material para la BRDF Blinn-Phong (sección 1.4.1.3) como se observa en el Código 22.

```

vec4 CalculateIndirectLighting(vec3 position, vec3 normal, vec3 albedo, vec4 specular)
{
    // resultado del cono especular
    vec4 specularTrace = vec4(0.0f);
    // dirección del cono
    vec3 coneDirection = vec3(0.0f);
    :

    // component greater than zero
    if(any(greaterThan(specular.rgb, specularTrace.rgb)))

```

```

{
    vec3 viewDirection = normalize(cameraPosition - position);
    // dirección del lobulo especular
    vec3 coneDirection = reflect(-viewDirection, normal);
    coneDirection = normalize(coneDirection);
    // apertura del cono especular segun la potencia para blinn-phong
    float aperture = max(tan(HALF_PI * (1.0f - specular.a)), 0.0174533f);
    // resultado
    specularTrace = TraceCone(position, normal, coneDirection, aperture);
    specularTrace.rgb *= specular.rgb;
}

```

Código 22: Cono para reflexión especular.

Para controlar la apertura del cono especular se utiliza la función tangente. En el componente alfa de la variable *specular* se encuentra la potencia especular sin escalar, esto quiere decir que esta tiene un rango de cero a uno. También se limita el cono especular a mínimo un grado de apertura. Los conos extremadamente finos puede afectar mucho el rendimiento del programa y la apertura no puede ser cero.

### 3.6.3. Oclusión Ambiental

En nuestra implementación los conos utilizados para calcular la reflexión difusa son también utilizados para calcular la oclusión ambiental. En el Código 23 se puede observar una pequeña adición realizada a la función *TraceCone* del Código 20 para incluir el cálculo de oclusión ambiental como es descrito en la sección 2.4.2:

```

vec4 TraceCone(vec3 position, vec3 normal, vec3 direction, float aperture, bool traceOcclusion)
{
    :
    // resultado de oclusión ambiental
    float occlusion = 0.0f;
    // declive del cono ambiental
    float falloff = 0.5f * aoFalloff * voxelScale;

    while(coneSample.a < 1.0f && dst <= maxDistance)
    {
        :
        if(traceOcclusion && occlusion < 1.0)
        {
            // acumulación de opacidad, multiplicación por la función f(r)
            occlusion += ((1.0f - occlusion) * anisoSample.a) / (1.0f + falloff * diameter);
        }
        :
    }
}

```

```

    return vec4(coneSample.rgb, occlusion);
}

```

Código 23: Oclusión ambiental para el algoritmo de trazado de conos.

El trazado del cono ambiental es muy similar a la acumulación completa la diferencia esta en que solo se acumula la opacidad y la multiplicación de cada muestra por la función  $f(r)$  descrita en la sección 2.4.2. Acá  $\lambda$  es la variable *falloff* pre-multiplicada por 0.5 para obtener el radio del cono. La oclusión ambiental es almacenada en el alfa del vector returned por *TraceCone* esta será utilizada luego para la composición final de la imagen.

### 3.6.4. Sombras Suaves con Trazado de Conos

El trazado de conos puede ser utilizado para obtener sombras suaves. La traza de este cono es exactamente igual al método *TraceCone*, sin embargo lo único que es necesario acumular es la opacidad de los véxeles. El Código 24 para el trazado de sombras describe la diferencia con respecto al Código 20.

```

float TraceShadowCone(vec3 position, vec3 direction, float aperture, float maxTracingDistance)
{
    :
    // distancia de trazado del cono
    maxDistance = min(maxDistance, maxTracingDistance);
    // visibilidad del fragmento
    float visibility = 0.0f;

    while(visibility.a < 1.0f && dst <= maxDistance)
    {
        :
        // acumulación de solo opacidad, multiplicada por un factor de escala
        visibility += (1.0f - visibility) * anisoSample.a * k;
        :
    }

    return 1.0f - visibility;
}

```

Código 24: Trazado de sombras con conos.

Cada muestra de opacidad es multiplicada por un valor  $k$  definido por el usuario que indica que tan suave es la sombra.

Este cono es trazado por cada fuente de luz que habilita trazado de sombras. La posición del cono es la posición del fragmento a iluminar y la dirección es la dirección de la luz.

La función también recibe la distancia de trazado, esto es útil para luces puntuales y focales las cuales comprenden un volumen de influencia en escena.

### 3.6.5. Composición Final

Incluyendo oclusión ambiental, reflexión especular y reflexión difusa la función *CalculateIndirectLighting* vista en el Código 22 y 21 culmina como se expone en el Código 25.

```
vec4 CalculateIndirectLighting(vec3 position, vec3 normal, vec3 albedo, vec4 specular, bool ambientOcclusion)
{
    :
    // iluminación indirecta multiplicada por un factor de escala
    vec3 result = bounceStrength * (diffuseTrace.rgb + specularTrace.rgb);

    // se retorna la oclusión ambiental en el alfa si se habilita, rgb contiene luz indirecta
    return vec4(result, ambientOcclusion ? clamp(1.0f - diffuseTrace.a + aoAlpha, 0.0f, 1.0f) : 1.0f);
}
```

Código 25: Composición para la iluminación indirecta.

La variable *bounceStrength* tiene un valor personalizado por el usuario. Este factor indica la intensidad de la iluminación indirecta. La oclusión ambiental es almacenada en el componente alfa del vector returned por la función. La variable *aoAlpha* es también indicada por el usuario, esta determina la minima intensidad para la oclusión ambiental resultante.

En el Código 26 una vez finalizado el cálculo de la iluminación directa a esta se le suma la iluminación indirecta, se multiplican ambas por la oclusión ambiental y luego se suma la emisión para obtener como resultado final el valor del fragmento.

```
layout(location = 0) out vec4 fragColor;
:

void main()
{
    :
    // cálculo de iluminación indirecta
    indirectLighting = CalculateIndirectLighting(position, normal, albedo, specular, true);
    // cálculo de iluminación directa
    directLighting = CalculateDirectLighting(position, normal, albedo, specular);
    // composición final
    compositeLighting = (directLighting + indirectLighting.rgb) * indirectLighting.a;
    // se agrega emisión del píxel
    compositeLighting += emissive;
    :
}
```

```

    // color mostrado en pantalla
    fragColor = vec4(compositeLighting, 1.0f);
}

```

Código 26: Composición final de imagen.

Después de obtener el valor del fragmento se puede aplicar corrección gamma, mapeo de tonos o *tonemapping* o cualquier otro post-proceso sobre la imagen. En nuestra implementación se hace solo corrección gamma y Reinhard *tonemapping* [31] sobre la variable *compositeLighting* sin post-procesamiento.

## 3.7. Iluminación Global de Vóxeles

En la sección anterior se expone el cálculo de iluminación indirecta de solo un rebote. Si se observa el Código 26 los únicos datos necesarios para este cálculo con solo el componente difuso son: albedo, normal, posición e iluminación directa.

Todos estos datos se encuentran disponibles en nuestra representación en vóxeles. La iluminación directa en vóxeles se obtiene luego del proceso de sombreado de vóxeles, mientras que albedo y normal son volúmenes producto del proceso de voxelización. La posición puede ser fácilmente proyectada utilizando la función *VoxelToWorld* vista en el Código 16.

Nuestra implementación realiza iluminación global de solo el componente difuso sobre los vóxeles utilizando *compute shaders*. El algoritmo de trazado es exactamente el mismo al expuesto en la sección anterior. La única diferencia es que este proceso se realiza por vóxel en vez de por píxel. El Código 27 expone este proceso:

```

// volúmenes de la representación en vóxeles
layout(binding = 0, rgba8) uniform image3D voxelComposite;
layout(binding = 1, rgba8) uniform sampler3D voxelAlbedo;
layout(binding = 2, rgba8) uniform sampler3D voxelNormal;
layout(binding = 3) uniform sampler3D voxelTexMipmap[6];
    :

void main()
{
    if(gl_GlobalInvocationID.x >= volumeDimension ||
       gl_GlobalInvocationID.y >= volumeDimension ||
       gl_GlobalInvocationID.z >= volumeDimension) return;

    ivec3 writePos = ivec3(gl_GlobalInvocationID);
    vec4 albedo = texelFetch(voxelAlbedo, writePos, 0);

    if(albedo.a < EPSILON) { return; }

```

```

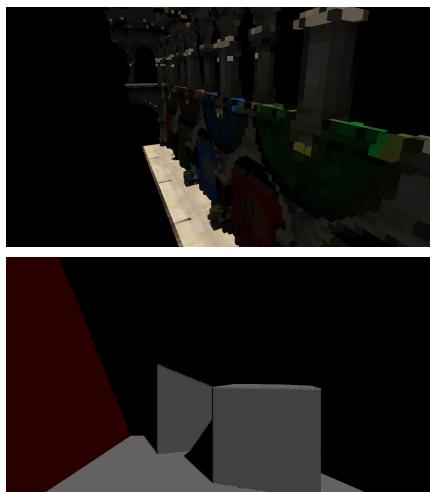
vec4 directLight = imageLoad(voxelComposite, writePos);
// normal promedio del proceso de voxelización
vec3 normal = texelFetch(voxelNormal, writePos, 0).xyz;
// vector normal esta almacenado en formato 0->1, se convierte a -1->1
normal = normalize(DecodeNormal(normal));
// calcular el primer rebote de luz sobre el voxel
vec3 position = VoxelToWorld(writePos);
vec4 indirectLighting = CalculateIndirectLighting(position, normal);
// multiplicación por el albedo producto de la voxelización
indirectLighting *= albedo;
// composición luz directa + luz indirecta
vec4 radiance = directLight + indirectLighting;
radiance.a = directLight.a;
// almacenar en la textura base de radiancia
imageStore(voxelComposite, writePos, radiance);
}

```

Código 27: Calculo de iluminación global sobre véxeles.

El método *CalculateIndirectLighting* es el mismo visto en la sección anterior, igualmente el trazado de conos pero sin oclusión ambiental. La textura *voxelComposite* es la textura resultante del sombreado de véxeles (Figura 44). Una vez terminado el cálculo de iluminación global sobre los véxeles debe volver a realizarse el proceso de filtrado anisótropo descrito en la sección 3.5.1. Almacenar la iluminación global producto del primer rebote sobre los véxeles nos permite aproximar el segundo rebote durante el proceso de trazado de conos de la sección anterior. En la Figura 48 se puede observar la escena voxelizada con iluminación global de véxeles.

Solo iluminación directa.



Con iluminación indirecta.

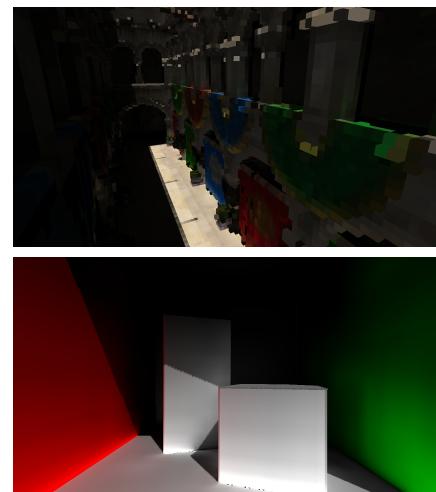


Figura 48: Resultado de véxeles con solo iluminación directa y con iluminación global difusa.

# **Capítulo 4**

## **Pruebas y Resultados**

### **4.1. Entorno de Pruebas**

Todos los experimentos realizados en esta sección fueron ejecutados en un computador de escritorio con las siguientes características de hardware:

1. Procesador AMD Phenom II X6 1055T 2.8 Ghz
2. 8 GB de Memoria RAM DD3
3. Disco duro de 1TB
4. Tarjeta gráfica AMD R9 380
5. Sistema Operativo Windows 10 de 64 bits.

#### **4.1.1. Configuración de la Aplicación**

Con respecto a la representación en véxeles distintos pasos del algoritmo solo se realizan dependiendo de ciertos eventos en escena. Sin embargo cada paso de este algoritmo es dependiente de pasos anteriores. El sombreado se véxeles solo necesita volver a realizarse bajo algún cambio en los parámetros de iluminación, al actualizarse el sombreado también deben realizarse todos los pasos siguientes. Igualmente sucede con la voxelización dinámica bajo algún cambio sobre un objeto dinámico. La aplicación también permite el cambio de parámetros en la escena estática, esto implica realizar todos los pasos del algoritmo. En contraste el trazado de conos se realiza constantemente por frame durante el paso de iluminación del sombreado diferido.

Para ejecutar pruebas que comprendan todos los aspectos del algoritmo es necesario que luces y objetos se encuentren registrando cambios constantemente. Para simplificar este proceso, la aplicación provee un modo de actualización forzosa por frame. Esto permite simular situaciones de estrés donde tanto objetos como luces en escena se encuentran bajo constantes cambios. Para ciertos experimentos este modo será desactivado de manera que solo se obtengan datos relevantes a ese ambiente de prueba.

## 4.2. Escenarios de Estudio

En esta sección se describe las distintas escenas y configuraciones a realizar para las distintas pruebas sobre la aplicación.

### 4.2.1. Escenas de Prueba y Objetos

Para la realización de pruebas de precisión y rendimiento se utilizan siete escenas las cuales se clasifican en dos categorías: escenas completas y ambientes de pruebas llamados escenas *sandbox*. Todas las escenas comprenden varios niveles de complejidad con respecto a geometría e iluminación. La aplicación también incluye una serie de modelos precargados los cuales serán agregados a las escenas, estos objetos son considerados dinámicos.

#### 4.2.1.1. Escenas Completas

Como escenas completas se consideraron cuatro escenas comunes para pruebas de iluminación global. Estas escenas presentan varios niveles de complejidad geométrica y de propagación de luz. En la Tabla 1 se listan sus nombres y atributos.

Nombre	Vértices	Triángulos	Texturas	Geometría	Iluminación
Sponza	153.635	278.163	Sí	Compleja	Compleja
Conference	194.399	331.179	No	Compleja	Compleja
Sibenik	40.479	75.283	Sí	Media	Media
Cornell Box	72	36	No	Simple	Simple

Tabla 1: Escenas completas y sus atributos.

**Sponza:** Modelo del atrio del palacio Sponza en Dubrovnik. Este modelo originalmente realizado por Marko Dabrovic y luego remodelado por Frank Meinl de Crytek con nuevos elementos como cortinas, vasos y plantas, además de mapas especulares, albedo y normales [32]. Esta es una escena de dimensión considerable con propagación de luz compleja, especialmente en las áreas ocluidas por las cortinas y los pasillos superiores donde la luz difusa rebota luego de pasar a través de varias columnas. En la Figura 49 se puede observar esta escena.

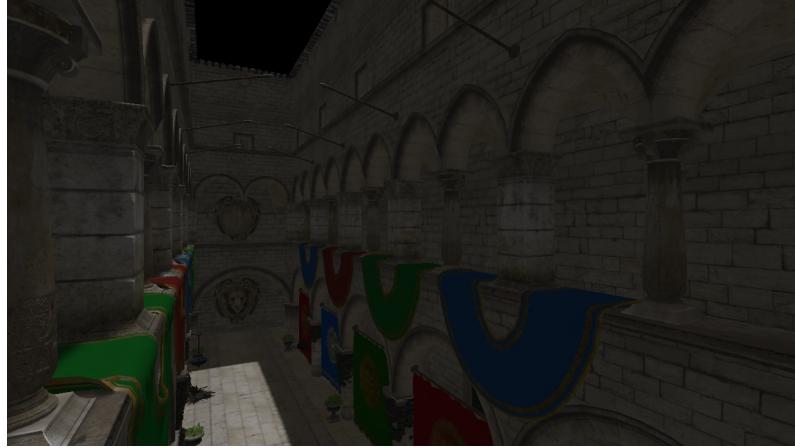


Figura 49: Sponza con solo iluminación directa desde una luz direccional más luz ambiental para visualizar las áreas sombreadas.

**Conference:** Un modelo 3D basado en una sala de conferencias del *Lawrence Berkeley National Laboratory* [33]. Esa es una escena interior pequeña pero de gran complejidad geométrica con muchos objetos similares. El transporte de luz es particularmente complejo en ciertas partes como debajo de sillas o la mesa central. En la Figura 50 se puede observar esta escena.



Figura 50: Conference con solo iluminación directa desde una luz direccional más luz ambiental para visualizar las áreas sombreadas.

**Sibenik:** El interior de una catedral [34]. La luz entra a ella a través de ventanas y se propaga por toda la escena. La escena tiene una sección con columnas ideal para probar sombras. En esta misma sección de columnas hay un piso de mármol que permite apreciar reflexión especular. Una alfombra roja cubre gran parte de la catedral esta permite la visualización de reflexión difusa. En la Figura 51 se puede observar esta escena.



Figura 51: Sibenik con solo iluminación directa desde una luz direccional más luz ambiental para visualizar las áreas sombreadas.

**Cornell Box:** Este es un modelo para la visualización de iluminación global. Creado por Donald Greenberg y estudiantes de *Cornell University* [35]. El Cornell Box está compuesto por una caja con dos paredes de colores, a la izquierda una roja y a la derecha una verde, el resto de la caja es blanca. Dentro de esta caja se encuentran dos cajas más, una pequeña y otra más alta. En la Figura 52 se puede observar esta escena.

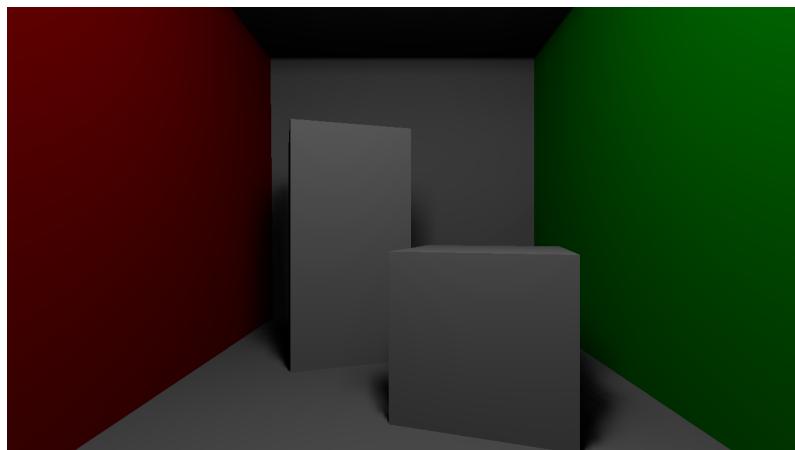


Figura 52: Cornell Box con solo iluminación directa desde una luz puntual con trazado de sombras.

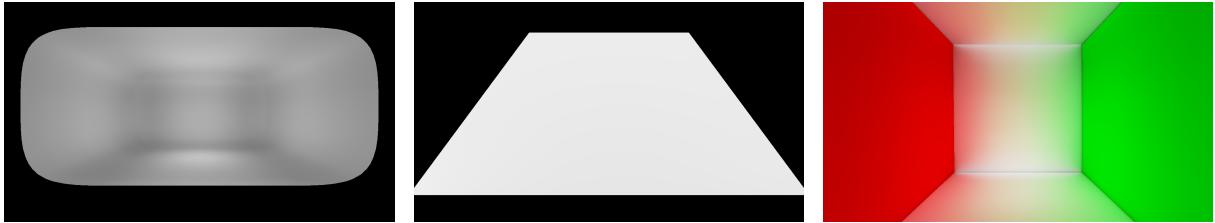
#### 4.2.1.2. Escenas Sandbox

Las escenas sandbox son entornos simples y vacíos donde se realizaran pruebas de aspectos específicos de la aplicación. Estas escenas no presentan mayor complejidad geométrica o complejidad de iluminación. En la Tabla 2 se listan sus nombres y atributos.

Nombre	Vértices	Triángulos	Texturas	Geometría	Iluminación
Light Box	4.098	8.192	No	Simple	Simple
Plane Test	16	24	No	Simple	Simple
Cornell Box Vacío	24	12	No	Simple	Simple

Tabla 2: Escenas Sandbox y sus atributos.

Light Box es un interior donde los bordes son tan suaves que no hay oclusión ambiental, el interior es totalmente blanco y no hay cambios bruscos en geometría, esto maximiza la propagación de luz en esta escena. Plane Test es sencillamente un plano, en una esquina se encuentra un cuboide fino pero de gran altura para permitir la voxelización de objetos sobre el plano. Cornell Box Vacío es la escena completa Cornell Box sin las cajas internas. En la Figura 53 se puede observar todas las escenas sandbox.



Escena Light Box con solo iluminación directa desde una luz puntual de baja intensidad.

Escena Plane Test con solo iluminación directa desde una luz direccional.

Escena Cornell Box Vacío con iluminación indirecta y solo una luz puntual.

Figura 53: Escenas sandbox.

#### 4.2.1.3. Objetos Precargados

La aplicación cuenta con una serie de objetos tridimensionales que pueden ser agregados a las escenas. Cada uno de estos objetos puede tener su propio material y matriz espacial. Estos objetos son considerados dinámicos por la aplicación. Los objetos están divididos en dos categorías: primitivas y modelos. Las primitivas son: Icosaedro, Cubo, Esfera, Cono, Toro, Plano y Cilindro. Los modelos son: Stanford Happy Buddha, Stanford Bunny, Stanford Dragon y Utah Teapot. En la Figura 54 se puede observar todos estos objetos.

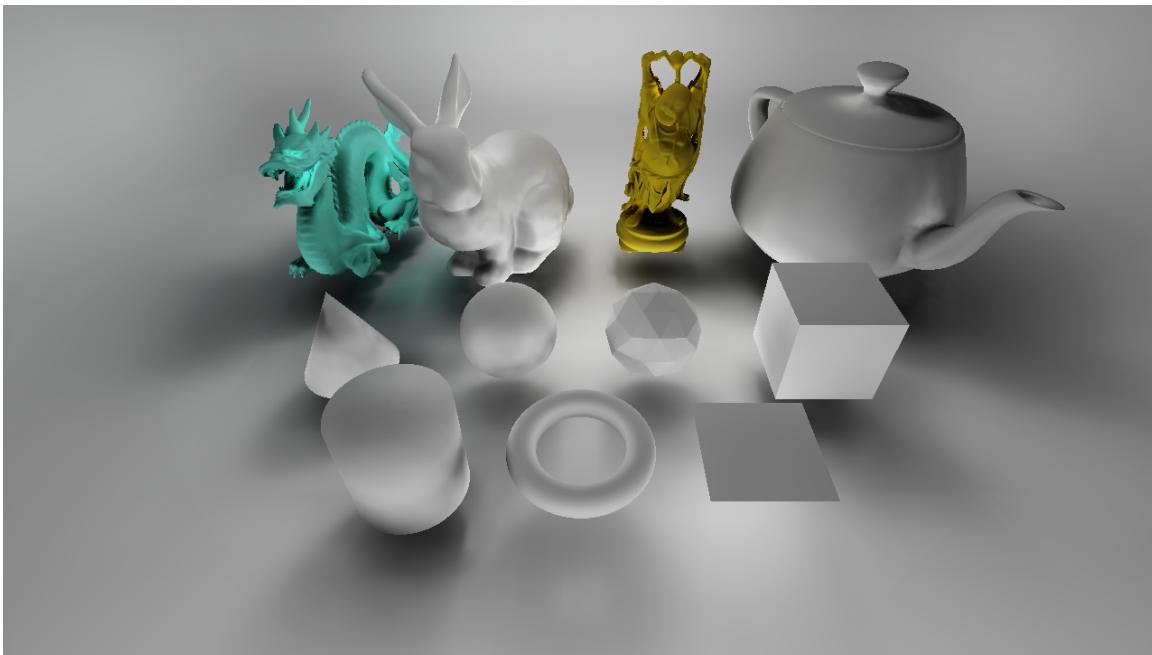


Figura 54: Objetos precargados. Arriba se encuentra los modelos Stanford Dragon, Stanford Bunny, Stanford Happy Buddha y Utah Teapot. El resto son primitivas, de izquierda a derecha: Cono, Esfera, Icosaedro, Cubo, Cilindro, Toro y Plano. Todos los objetos se encuentran en la escena Light Box utilizando una luz focal con trazado de sombras e iluminación indirecta.

#### 4.2.1.4. Criterios de Complejidad Geométrica e Iluminación

Para determinar la complejidad geométrica de cada escena se consideró la cantidad de triángulos que estas poseen y no la cantidad de objetos a renderizar ya que este es el factor que mayor afecta el proceso de voxelización.

Para determinar la complejidad de iluminación se consideraron los siguientes aspectos:

1. El nivel de detalle de la representación en vértices de la escena. En la escena Cornell Box la representación en vértices es muy similar a la geometría original. En contraste la escena Sponza difiere considerablemente de la geometría original en lugares con detalles finos como los vasos y plantas de esta escena.
2. Secciones ocluidas y facilidad de propagación de luz sobre distintas áreas de la escena. En la escena Cornell Box la luz se propaga fácilmente desde cualquier superficie iluminada a otras partes de la escena. En contraste escenas como Conference o Sponza tienen complicadas secciones de geometría como las sillas en Conference y las cortinas inferiores en Sponza que reducen la luz a pequeños haces.

## 4.3. Estudio de Rendimiento

Para el estudio de rendimiento se utilizó la cantidad de tiempo entre rutinas relevantes a cada prueba en milisegundos. Para obtener tiempos precisos en la GPU se utilizó el software GPU PerfStudio y análisis de frame. Este software permite observar entre muchas otras cosas el tiempo total de cada invocación a la función de dibujo o *draw call* al API gráfico OpenGL.

### 4.3.1. Prueba Base

La prueba base de rendimiento comprende todos los pasos del algoritmo sin trazado de sombras o modificaciones sobre la escena estática. Para esta prueba se colocó la aplicación en actualización forzosa de tal manera que todos los pasos del algoritmo se realicen por frame. También se varían tres aspectos de la aplicación: resolución de pantalla, dimensión de la representación en vóxeles y factor de longitud de marcha del cono.

Para estas pruebas se utilizaron todas las escenas completas. La configuración del escenario constó de una luz direccional con mapeado de sombras y tres modelos precargados. Los modelos utilizados fueron Stanford Happy Buddha, Stanford Dragon y Stanford Bunny, seleccionados por su complejidad geométrica. Cada modelo tiene su propio material con un cono especular con apertura de 45, 27 y 9 grados respectivamente. La cámara en escena fue colocada de tal forma que todos los píxeles visibles formen parte del trazado de conos.

Escena	Voxelización Estática	Limpieza de Vóxeles Dinámicos	Voxelización Dinámica	Sombreado de Vóxeles	Filtrado Anisotrópico	Illuminación Global de Vóxeles	Filtrado Anisotrópico	Trazado de Conos con Vóxeles	Tiempo Dinámico
Sibenik	1.80	0.58	2.11	0.95	1.39	3.88	1.38	7.31	17.60
Cornell Box	0.51	0.78	1.30	1.33	1.38	8.41	1.37	7.23	21.81
Conference	46.04	0.56	1.52	0.86	1.38	3.23	1.37	7.50	16.42
Sponza	11.29	0.60	2.03	1.13	1.37	5.44	1.38	7.01	18.97

Tabla 3: Rendimiento (en milisegundos) de todas las partes del algoritmo en distintas escenas utilizando volúmenes de resolución  $256^3$  vóxeles, factor de longitud de marcha de 0.5 y resolución de pantalla de  $1280x720p$ .

		Resolución de Volumenes												
	Escena	Voxelización Estática		Limpieza de Vóxeles Dinámicos		Voxelización Dinámica		Sombreado de Vóxeles		Filtrado Anisotrópico	Iluminación Global de Vóxeles	Filtrado Anisotrópico	Trazado de Conos con Vóxeles	Tiempo Dinámico
	64 <sup>3</sup> vóxeles	Sibenik	3.00	0.01	9.17	0.04	0.11	0.12	0.10	5.17	14.74			
	64 <sup>3</sup> vóxeles	Cornell Box	0.07	0.02	2.13	0.06	0.11	0.29	0.11	4.79	7.51			
	64 <sup>3</sup> vóxeles	Conference	39.68	0.01	5.98	0.03	0.11	0.10	0.11	5.38	11.72			
	64 <sup>3</sup> vóxeles	Sponza	22.87	0.01	13.93	0.05	0.17	0.13	0.17	5.49	19.97			
	128 <sup>3</sup> vóxeles	Sibenik	2.17	0.08	3.93	0.17	0.26	0.62	0.26	2.17	3.93			
	128 <sup>3</sup> vóxeles	Cornell Box	0.16	0.10	1.38	0.28	0.26	1.62	0.26	5.75	1.38			
	128 <sup>3</sup> vóxeles	Conference	39.19	0.07	2.47	0.15	0.26	0.54	0.26	6.42	2.47			
	128 <sup>3</sup> vóxeles	Sponza	15.15	0.08	4.00	0.20	0.26	0.82	0.26	5.47	4.00			
	512 <sup>3</sup> vóxeles	Sibenik	2.36	4.51	1.35	6.06	10.92	23.23	10.91	8.57	65.55			
	512 <sup>3</sup> vóxeles	Cornell Box	6.18	6.08	1.80	7.54	10.70	41.14	10.87	7.87	86.00			
	512 <sup>3</sup> vóxeles	Conference	47.44	4.46	1.28	5.63	11.57	18.70	11.35	8.56	61.56			
	512 <sup>3</sup> vóxeles	Sponza	9.05	4.55	1.39	6.87	11.16	28.09	10.79	7.64	70.49			

Tabla 4: Rendimiento (en milisegundos) en contraste con la Tabla 3 considerando distintas resoluciones para la representación en vóxeles, todos los pasos del algoritmo se ven afectados por este parámetro.

Resolución Pantalla	1920x1080p	1280x720p
Escena	Trazado de Conos con Vóxeles	Trazado de Conos con Vóxeles
Sibenik	14.30	7.31
Cornell Box	15.17	7.23
Conference	16.27	7.50
Sponza	16.69	7.01

Tabla 5: Rendimiento (en milisegundos) con una mayor resolución de pantalla. Esto solo afecta el trazado de conos con vóxeles.

Longitud de Marcha	0.1	0.25	0.5	2.5
Escena	Trazado de Conos con Vóxeles			
Sibenik	29.55	12.73	7.31	2.67
Cornell Box	26.80	11.57	7.23	2.68
Conference	29.73	12.93	7.50	2.85
Sponza	29.81	12.92	7.01	2.81

Tabla 6: Rendimiento (en milisegundos) con distintas configuraciones de la longitud de marcha del cono. Esto solo afecta el trazado de conos con vóxeles.

Considerando como tiempos interactivos todo resultado por debajo de  $33.3\text{ ms}$  (aproximadamente 30 cuadros por segundo). Nuestra implementación logra colocarse por debajo de este tiempo en todos los casos excepto los casos que comprenden resoluciones de volúmenes mayores a  $256^3$  vóxeles.

#### 4.3.1.1. Densidad Geométrica y Velocidad de Voxelización

La cantidad de triángulos dentro del espacio que representa un vóxel afecta la velocidad de voxelización. Esto se debe a que este proceso requiere sincronización entre distintos hilos por fragmento. A mayor cantidad de triángulos en este espacio mayor es la cantidad de fragmentos generados por el proceso de rasterización. Cada uno de estos hilos debe esperar a escribir en la misma posición del volumen para garantizar atomicidad.

En la Figura 55 se puede observar esta condición, especialmente en la escena Conference. Esta es una escena pequeña en escala, sin embargo de todas las escenas completas es la que posee mayor cantidad de triángulos como se puede observar en la Tabla 1. También es notable como disminuye el tiempo de voxelización para algunas escenas al aumentar la resolución del volumen y como aumenta al utilizar menores resoluciones.

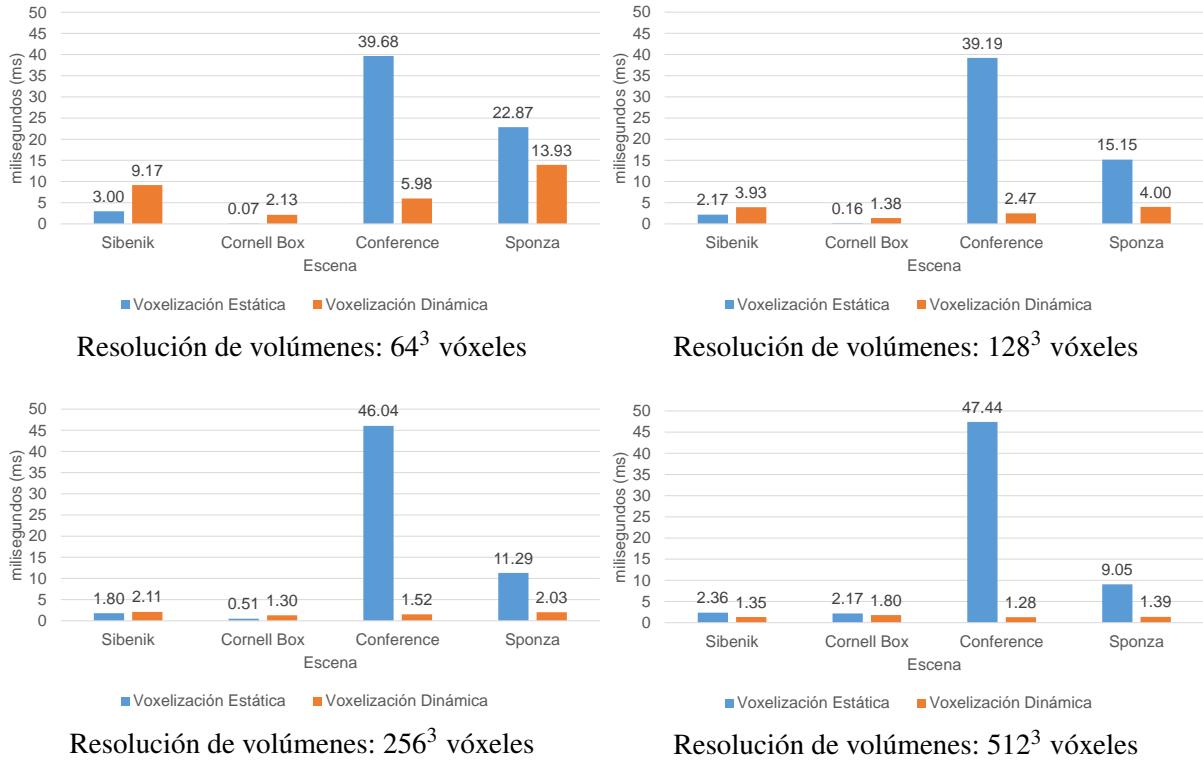


Figura 55: Grafica para los tiempos de voxelización dinámica de la Tabla 4.

#### 4.3.1.2. Vacuidad y Velocidad de Trazado para la Iluminación Global de Vóxeles

En escenas donde existen muchos espacios vacíos el trazado de rayos suele tardar un poco más que en escenas densas de objetos. Esto se debe a que mientras la representación en vóxeles sea vacía en la posición que describe la apertura, dirección y punto de origen del cono, este cono debe seguir expandiéndose. A consecuencia mientras más espacios vacíos la representación en vóxeles tenga, más distancia tiene que recorrer cada cono aumentando la cantidad de operaciones de lectura sobre la representación en vóxeles.

Este problema se puede observar en la Figura 56 para las escenas Sponza y Cornell Box. La escena Cornell Box es en gran parte vacua con solo dos cuboides internos. La escena Sponza es densa en su interior por tanto la falta de objetos no es el principal problema. El problema de la Sponza reside en que la escena es larga y alta pero no ancha. Durante la voxelización cada triángulo se proyecta de forma ortogonal sobre cada eje para maximizar la voxelización, este tronco cuadrado de proyección es un cubo uniforme. Para las dimensiones de esta escena siempre quedará una cantidad considerable de vóxeles vacíos fuera de la geometría principal de la escena.

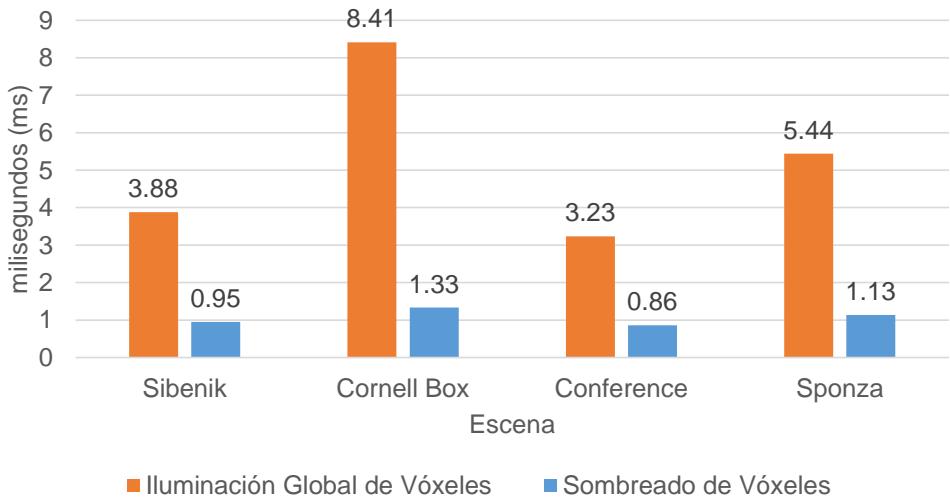


Figura 56: Tiempos de cálculo de iluminación global sobre la representación en vóxeles y tiempos de sombreado (solo iluminación directa) para todas las escenas completas.

### 4.3.2. Trazado de Sombras y Volumen de Visibilidad

La aplicación provee sombras suaves para luces direccionales, puntuales y focales utilizando trazado de conos. Si no se sombra la representación en vóxeles la imagen resultante es incorrecta a pesar de definir los contornos de sombreado. La razón es que el trazado de conos para la iluminación indirecta consideraría estos vóxeles como iluminados en vez de ocluidos por sombras. Nuestra implementación provee un método de inclusión de sombras suaves durante el sombreado de vóxeles. En esta sección se examina el rendimiento de las distintas opciones para el trazado de sombras que provee nuestra implementación excepto mapeado de sombras (sección 1.7.1) disponible solo para luces direccionales.

Para estas pruebas se utilizaron todas las escenas completas. La configuración del escenario constó de solo una luz puntual con trazado de sombras habilitado. La cámara en escena fue colocada de tal forma que todos los píxeles visibles formen parte del trazado de conos.

Con respecto a la configuración de la aplicación, se activó el modo de actualización forzosa para simular luces en constante movimiento y se desactivó la iluminación global de vóxeles. La resolución de la representación en vóxeles utilizada fue de  $256^3$  vóxeles y el factor de longitud de marcha del cono fue 0.5.

Este estudio considera las distintas opciones en la aplicación relevantes al trazado de sombras. Estas opciones comprenden: trazado de sombras durante el sombreado de vóxeles, trazado de sombras con trazado de conos y vóxeles o muestreo del volumen de visibilidad.

Escena	Sombreado de Vóxeles	Sombreado y Trazado de Sombras
Sibenik	0.95	4.57
Cornell Box	1.33	20.32
Conference	0.86	4.77
Sponza	1.13	3.31

Tabla 7: Tiempos (en milisegundos) de sombreado de vóxeles con y sin trazado de sombras sobre el volumen.

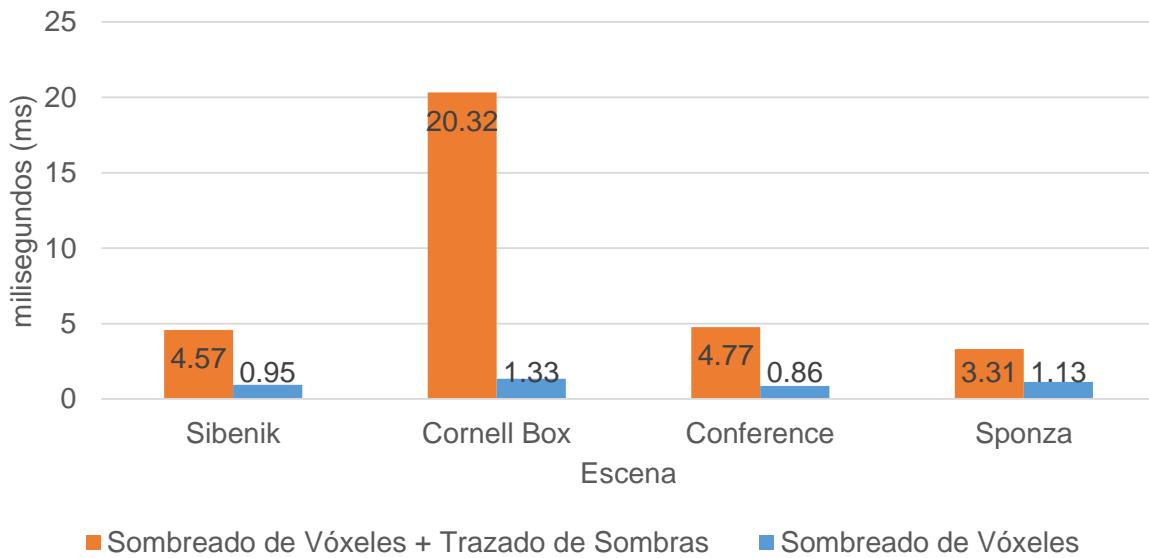


Figura 57: Gráfica para la Tabla 7 describe el tiempo agregado al sombreado de vóxeles a utilizar trazado de sombras.

En la Figura 57 se puede observar un incremento del tiempo de cómputo invertido durante el sombreado de vóxeles al incluir sombras trazadas sobre el volumen. En la escena Cornell Box este incremento es mucho mayor al de las demás escenas. La razón es que esta escena es en gran parte vacía, por lo tanto la mayoría de los rayos trazados tienen que recorrer un largo camino antes de culminar. Sin embargo, la escena Sponza a pesar de tener los mismos problemas de vacuidad ya explicados en la sección 4.3.1.2 tiene un tiempo agregado similar a las demás escenas sin estos problemas. Esto se debe principalmente a que la fuente de luz puntual se encuentra en el interior de la escena. La posición de la fuente de luz a trazar en la escena afecta directamente el rendimiento, ya que esta determina en promedio que tan rápido la mayoría de los rayos terminan su recorrido.

Escena	Trazado de Conos	Volumen de Visibilidad	Conos para Sombras
Sibenik	7.31	6.03	8.58
Cornell Box	7.23	6.50	10.31
Conference	7.50	5.65	5.65
Sponza	7.01	6.28	8.30

Tabla 8: Tiempo (en milisegundos) agregado al paso de trazado de conos con véxeles al incluir sombras bajo distintos métodos.

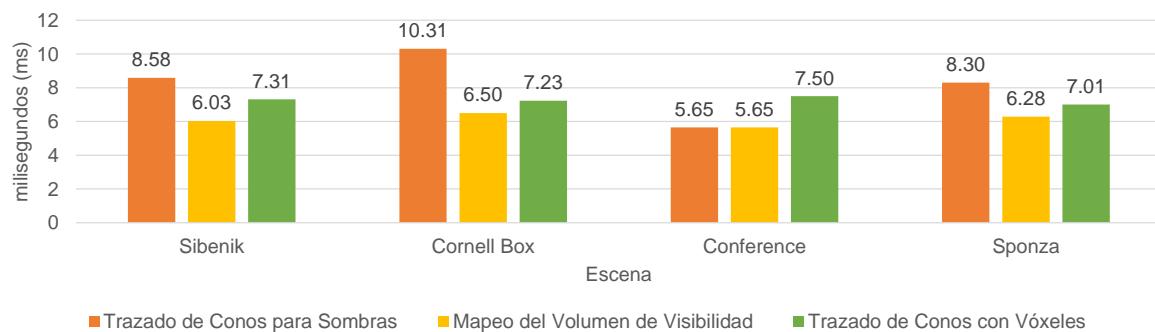


Figura 58: Gráfica para la Tabla 8 describe el tiempo agregado al trazado de conos con véxeles al utilizar conos para sombras suaves o mapeo del volumen de visibilidad.

El mapeo del volumen de visibilidad ofrece mayor rendimiento a cambio de menor calidad de sombras. El trazado de conos con véxeles incluye el tiempo invertido en mapeo de sombras en la prueba base 3.

En la Figura 58 se observa que el mapeo del volumen de visibilidad es mucho más rápido que el resto de los métodos expuestos. Esto se debe a que este es solo una instrucción de lectura más una simple operación aritmética para convertir la posición en espacio de mundo a una coordenada en espacio de textura como se observa en el Código 16.

En algunos casos el trazado de conos para sombras puede incluso ser más rápido que la técnica de mapeo de sombras. Esto se observa en la Tabla 8 para la escena Conference. La razón es que ésta es una escena pequeña en escala y densa en geometría, los conos trazados suelen terminar rápidamente.

### 4.3.3. Apertura del Cono Especular y Cono de Sombras

La apertura de los conos a trazar afecta directamente el rendimiento del trazado de conos con véxeles. Mientras más fino es este cono más lenta es su expansión a través del espacio en escena ya que el incremento de la longitud de marchado es mucho más lento. También mientras más fino es este cono mayor tiempo se invierte en los niveles más altos de detalle en la representación de véxeles, en la Tabla 4 se puede observar que a mayor resolución del volumen mayor es el tiempo de trazado.

El objetivo de estas pruebas es demostrar la diferencia en rendimiento según distintas configuraciones de apertura del cono especular y el cono utilizado para el trazado de sombras suaves.

Para ambas pruebas se utilizó solo la escena completa Cornell Box. Para la prueba del cono de sombras se colocó dentro de la escena una luz puntual con trazado de sombras habilitado. Para la prueba del cono especular se colocó dentro la escena una esfera con un material especular y una luz puntual con trazado de sombras inhabilitado. Para ambas pruebas la resolución de la representación en véxeles y la longitud de marcha del cono es de  $256^3$  véxeles y 0.5 respectivamente. El ángulo de apertura para ambas pruebas es 1, 5, 20, y 45 grados. En la Figura 59 se puede observar los resultados.

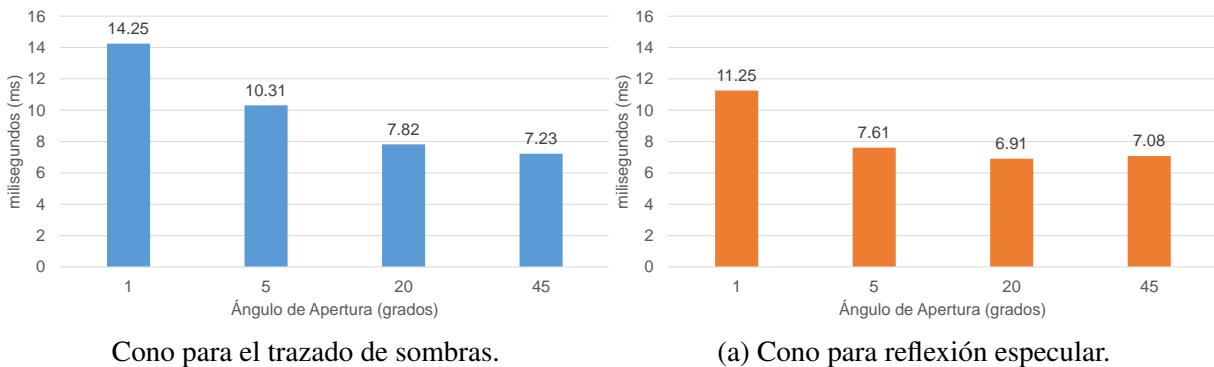


Figura 59: Tiempo sobre el trazado de conos con voxels para distintos ángulos de apertura con conos para sombras y reflexión especular.

### 4.3.4. Comparaciones

Para el estudio el rendimiento con respecto a otras aproximaciones para cálculo de iluminación indirecta se consideró una aplicación con trazado de conos con véxeles utilizando un octree disperso para la representación con véxeles y otra aplicación utilizando CLPV visto

en la sección 1.8.3. Para ambos programas se utilizó la escena Sponza con cámara y luces en posiciones y direcciones semejantes. La aplicación fue colocada bajo actualización forzosa. Solo se muestra el tiempo final de renderizado en milisegundos. La Tabla 9 demuestra los resultados de esta comparación.

Para comparar esta solución con CLPV se utilizó la implementación realizada por NVIDIA en su DirectX 11 SDK [36]. Para la comparación con una implementación con octrees se utilizó la aplicación desarrollada por Simon Yeung para trazado de conos con véxeles [37].

Escena	Nuestra Implementación	Representación con un Octree Disperso	CLPV
Sponza	20.83	89.13	18.44

Tabla 9: Comparación de rendimiento (en milisegundos) con otras aplicaciones.

En la Tabla 9 nuestra implementación obtiene un rendimiento mayor comparado con una representación en octree disperso. Esto se debe a que el muestreo de la estructura de véxeles consiste simplemente en la lectura de una textura, mientras el uso de octree implica el recorrido de un árbol cada vez que se quiere obtener un valor. CLPV obtiene 2 ms más que nuestra implementación sin embargo es bueno mencionar que esta técnica no comprende reflexión especular.

## 4.4. Estudio de Calidad de Imagen

En esta sección se estudiaron configuraciones que afectan la calidad de imagen resultante. Para el estudio de diferencia entre imágenes se utilizó el software PerceptualDiff basado en el trabajo de Hector Yee et al. en 2001 [38], este programa utiliza un modelo computacional imitando al ojo humano para generar la diferencia perceptual entre dos imágenes.

### 4.4.1. Composición Final de Imagen

Todas las imágenes en esta sección fueron renderizadas con una resolución de pantalla de  $1920 \times 1080p$ , con una resolución para la representación de véxeles de  $512^3$  véxeles y con un factor de longitud de marcha del cono de 0.5. La representación de véxeles solo contiene iluminación directa.

En todas las imágenes, para el resultado final por escena, se puede apreciar la aparición de distintos fenómenos de iluminación global como mezclado de colores, reflexión difusa, reflexión especular para superficies reflectivas y oclusión ambiental.

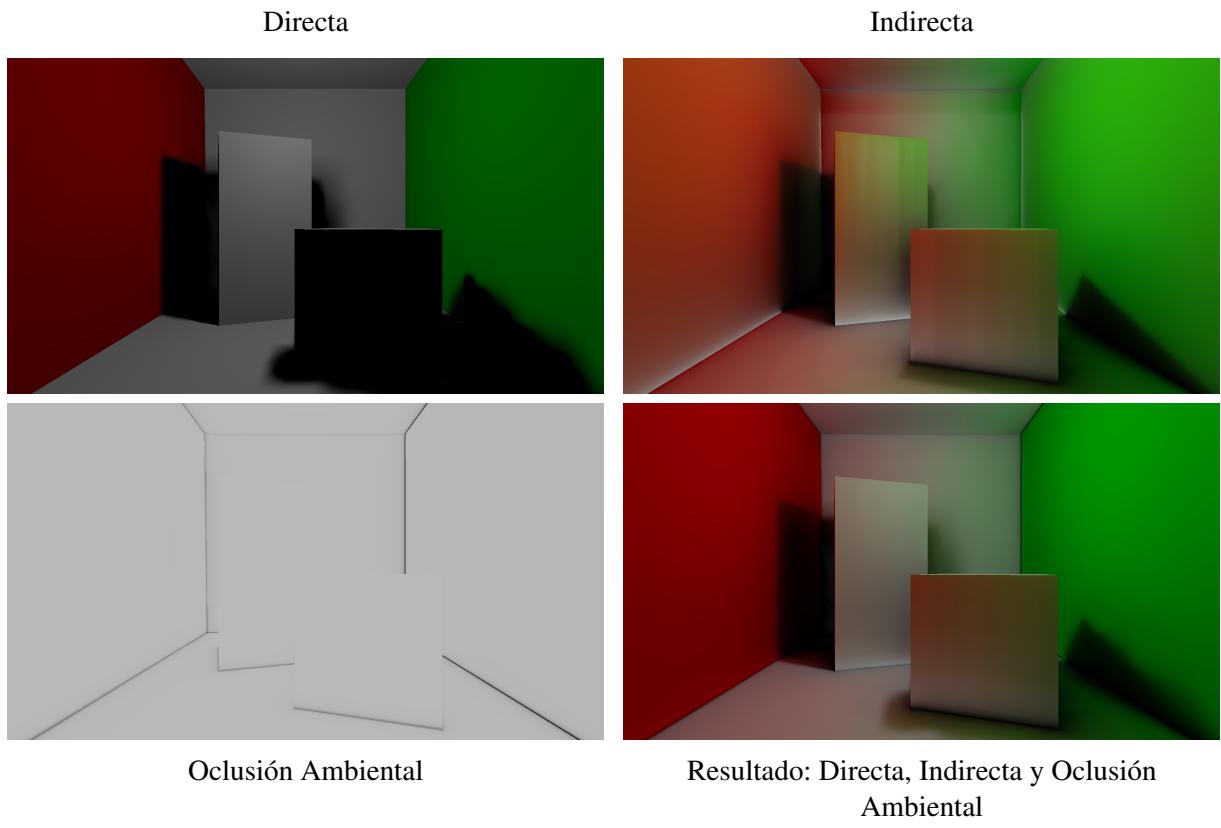


Figura 60: Composición para la escena Cornell Box.

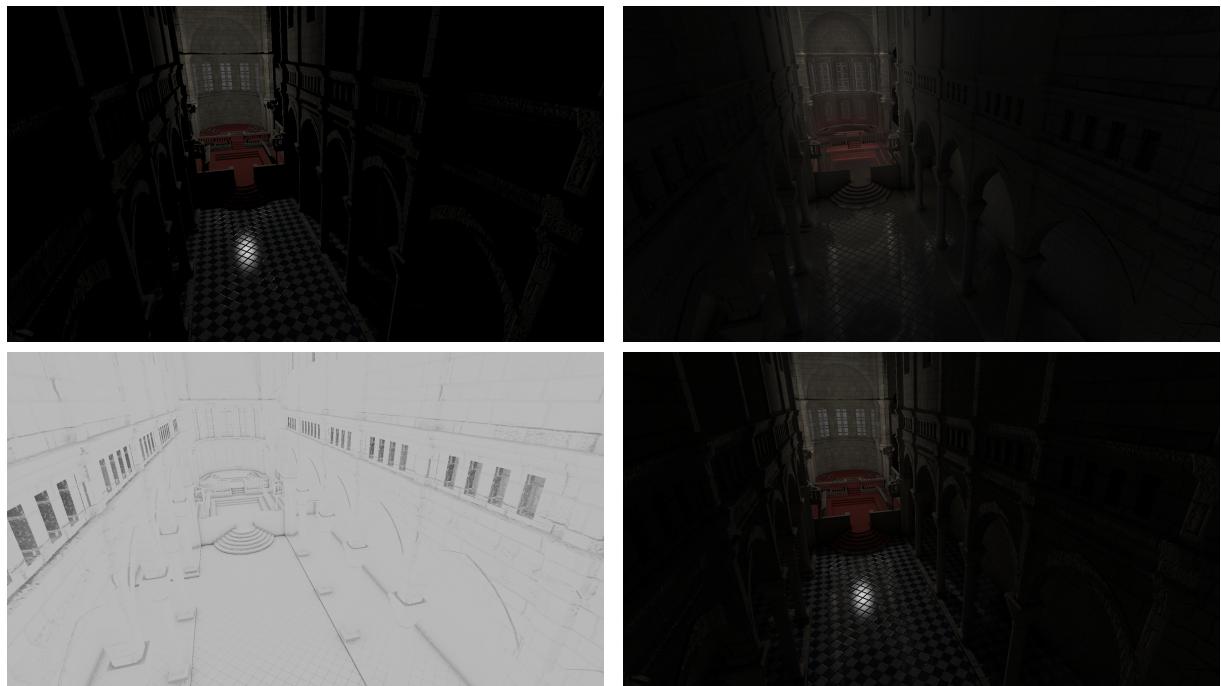


Figura 61: Composición para la escena Sibenik.



Figura 62: Composición para la escena Sponza.



Figura 63: Composición para la escena Conference.

#### 4.4.1.1. Iluminación Global de Vóxeles.

En esta sección se demuestra la diferencia de la imagen final al agregar iluminación global a la representación con vóxeles. La inclusión de iluminación global sobre los vóxeles nos permite aproximar el segundo rebote de luz. La diferencia entre imágenes fue procesada utilizando PerceptualDiff. Los píxeles azules describen donde se encuentran las áreas con mayor diferencia al ojo humano.

En todas las imágenes se puede observar una diferencia considerable con respecto a los resultados de la sección anterior al incluir iluminación global de vóxeles. Los detalles de reflexión especular aumentan al estar la representación en vóxeles mas iluminada por la inclusión del rebote difuso, y la reflexión difusa alcanza regiones anteriormente ocluidas en todas las escenas.

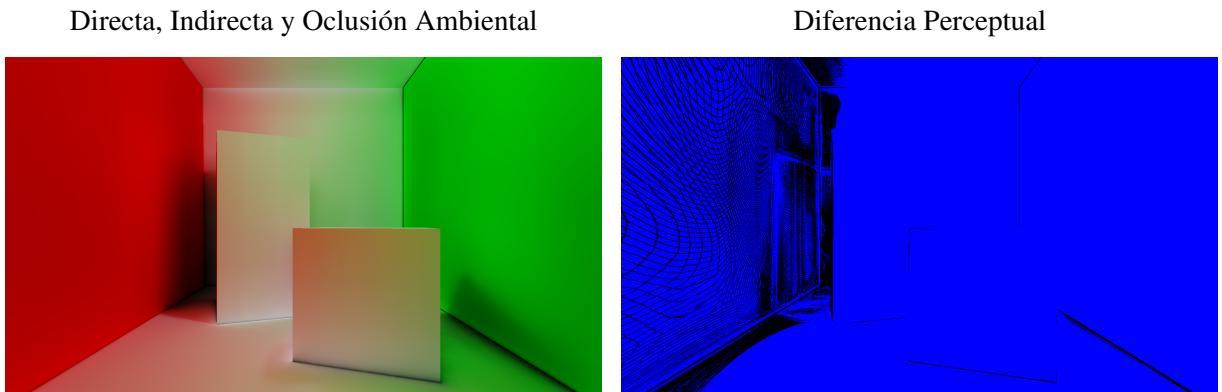


Figura 64: Diferencia con respecto al resultado en la Figura 60 con iluminación global de vóxeles

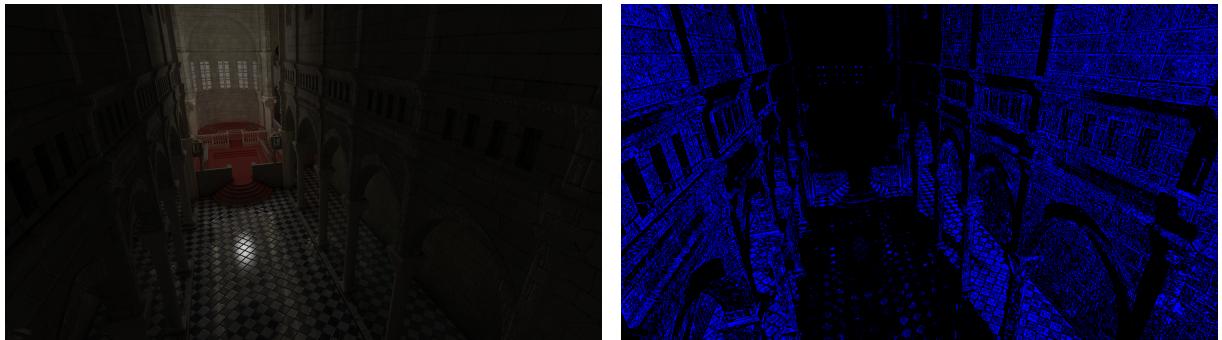


Figura 65: Diferencia con respecto al resultado en la Figura 61 con iluminación global de vóxeles



Figura 66: Diferencia con respecto al resultado en la Figura 62 con iluminación global de vóxeles



Figura 67: Diferencia con respecto al resultado en la Figura 63 con iluminación global de vóxeles

#### 4.4.1.2. Resolución de la Representación en Vóxeles

En esta sección se demuestra la diferencia de la imagen final con distintas resoluciones para la representación en vóxeles. Todas las imágenes serán comparadas con los resultados de la sección 4.4.1 donde se utiliza una resolución para los volúmenes de  $512^3$  vóxeles sin iluminación global de vóxeles. En las imágenes se puede apreciar como la diferencia aumenta a medida que se disminuye la resolución.

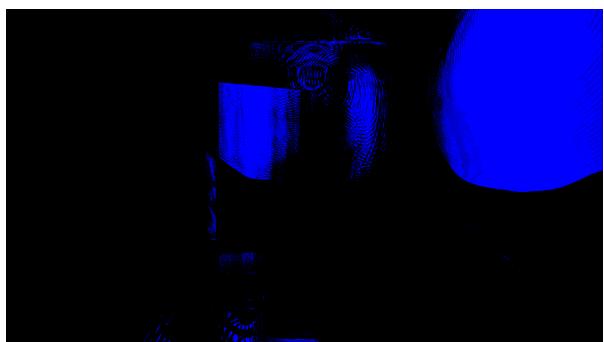
Todas las imágenes en esta sección presentan perdida de precisión durante el cálculo de iluminación indirecta, con respecto a los resultados de la sección 4.4.1. Para escenas de gran escala como Sponza este problema de precisión es mayor. En la escena Sponza, se observa la aparición de colores erróneos durante el cálculo de iluminación indirecta, especialmente sobre las columnas superiores en la escena, al utilizar una resolución de  $64^3$  vóxeles.

Directa, Indirecta y Oclusión Ambiental

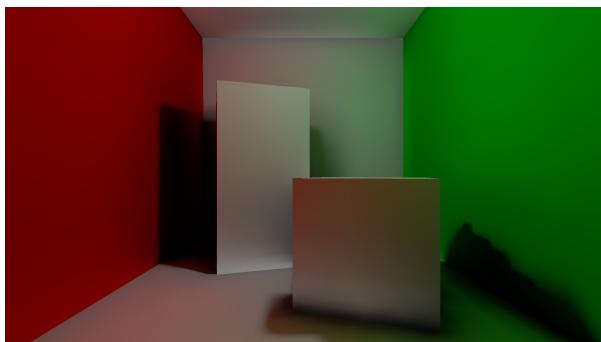


$256^3$  vóxeles

Diferencia Perceptual



$128^3$  vóxeles



$64^3$  vóxeles



Figura 68: Diferencia con respecto al resultado en la Figura 60 con distintas resoluciones para la representación en vóxeles.

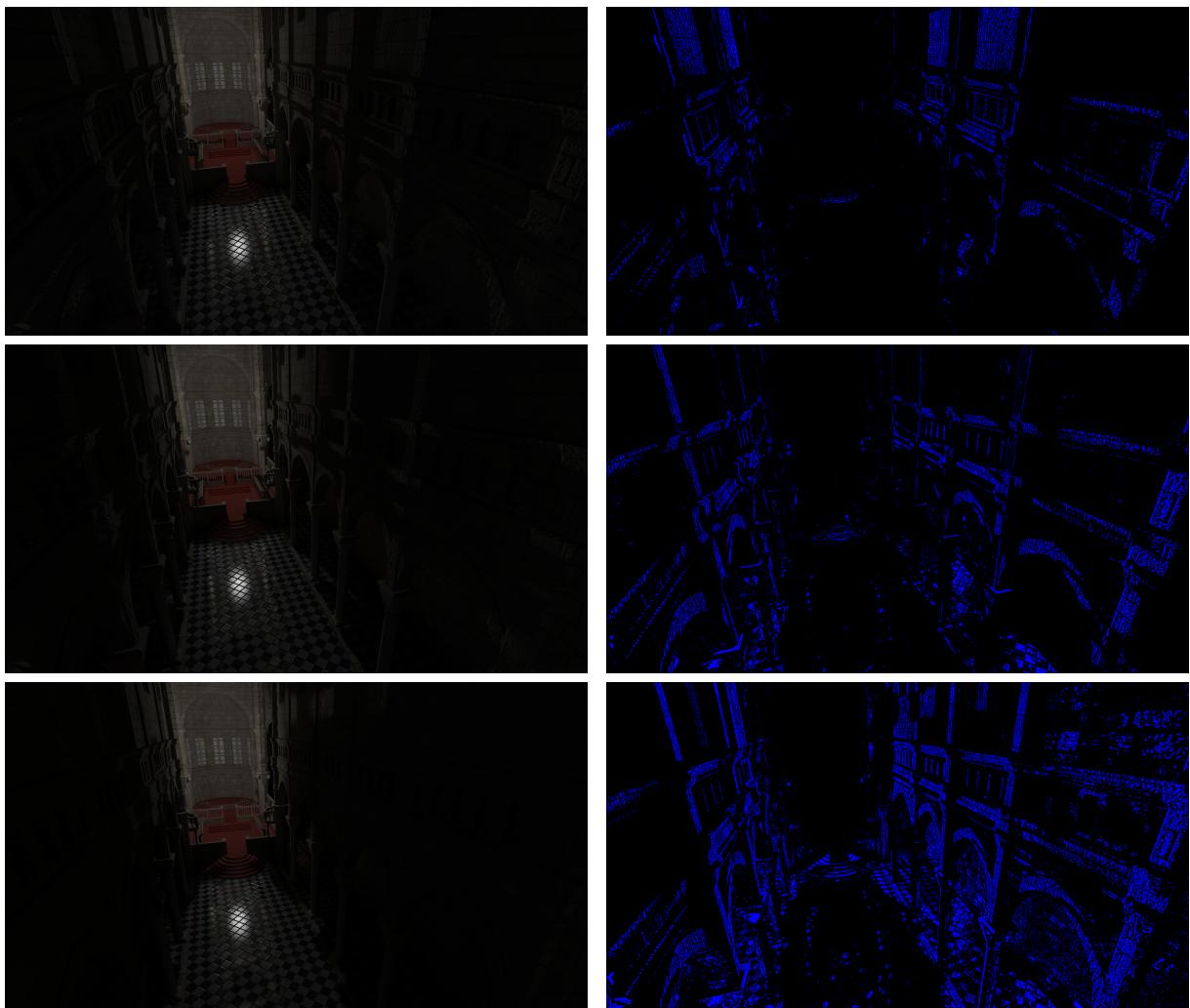


Figura 69: Diferencia con respecto al resultado en la Figura 61 con distintas resoluciones para la representación en véxeles.

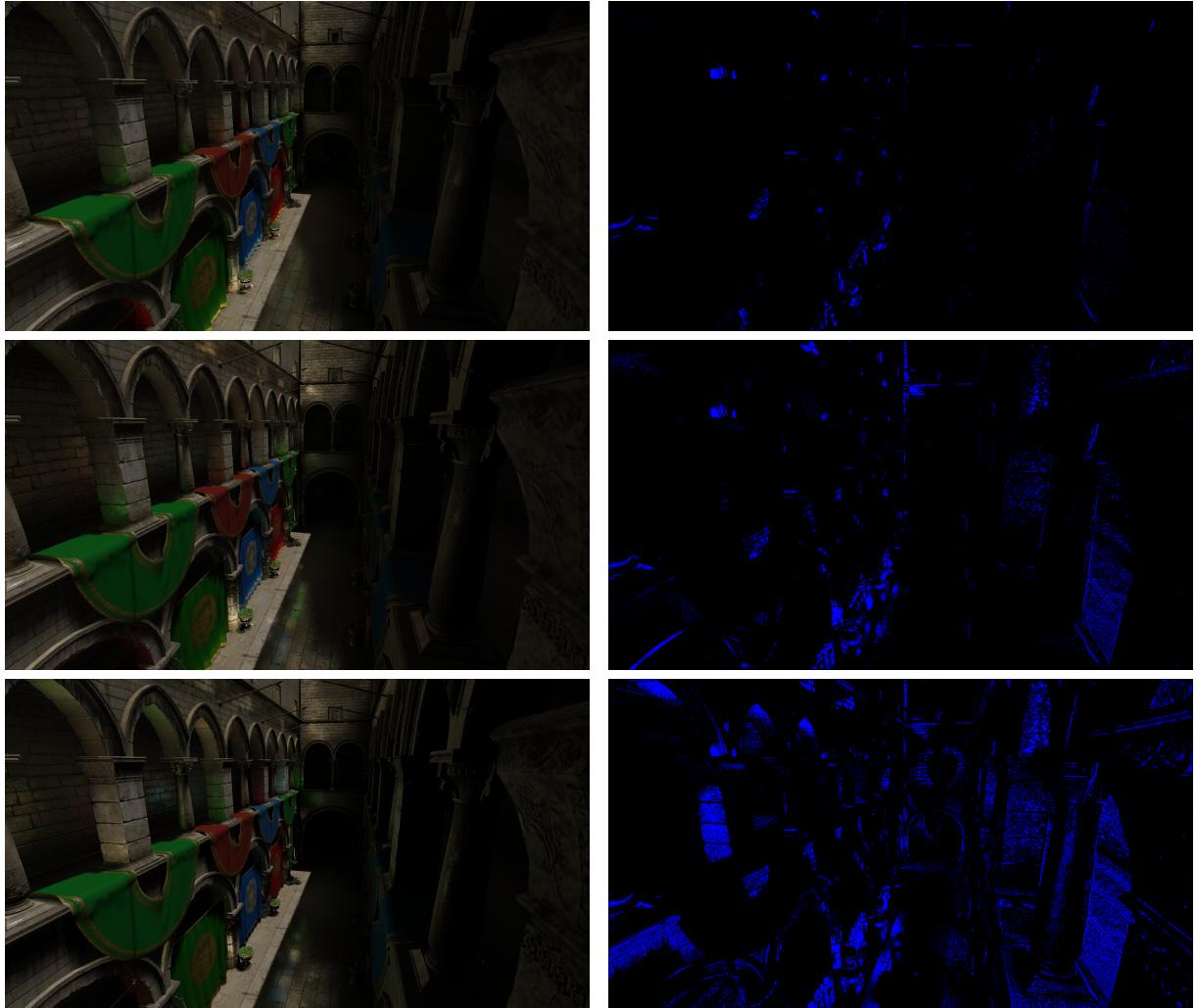


Figura 70: Diferencia con respecto al resultado en la Figura 62 con distintas resoluciones para la representación en véxeles.

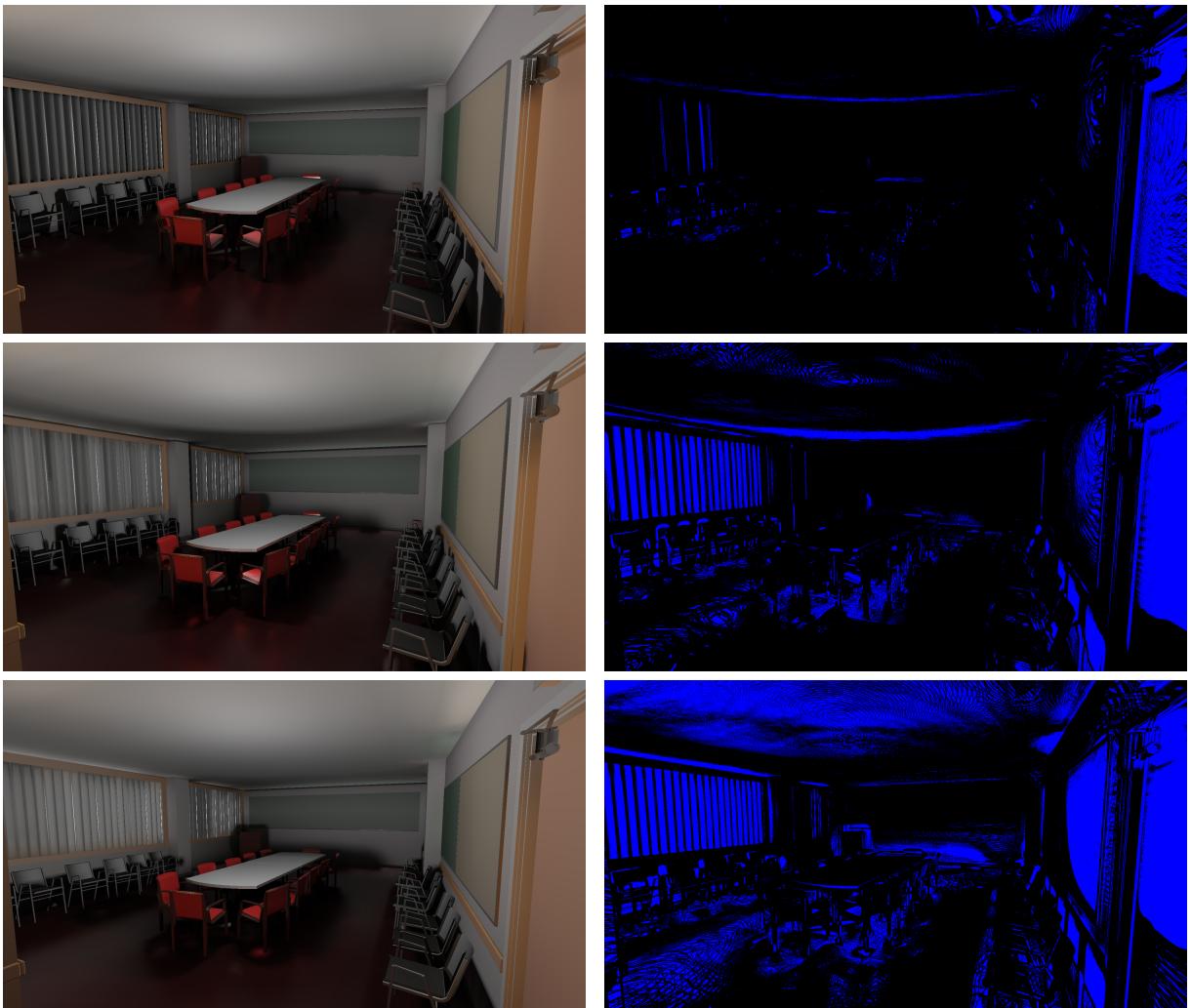


Figura 71: Diferencia con respecto al resultado en la Figura 63 con distintas resoluciones para la representación en véxeles.

#### 4.4.1.3. Factor de Longitud de Marcha del Cono

El factor de longitud de marcha del cono afecta tanto el rendimiento como la calidad visual final. Valores mayores a 1 no son recomendados ya que saltarían véxeles constantemente durante el trazado. En esta sección se compara contra el factor 0.5 en la sección 4.4.1 con valores de 1 y 2.5.

Los defectos de aumentar el factor de longitud son notables al utilizar valores mayores a 1. Ello asegura que habrá salto de muestras durante el recorrido del cono. Esto es notable en la escena Cornell Box con factor de longitud de 2.5, especialmente sobre las sombras suaves que son generadas con trazado de conos. Para la mayoría de las escenas, aumentar el factor de 0.5 (sección 4.4.1) a 1 no afecta de gran manera la diferencia perceptual, sin embargo esta diferencia

puede aumentar dependiendo de dónde se encuentre el observador.

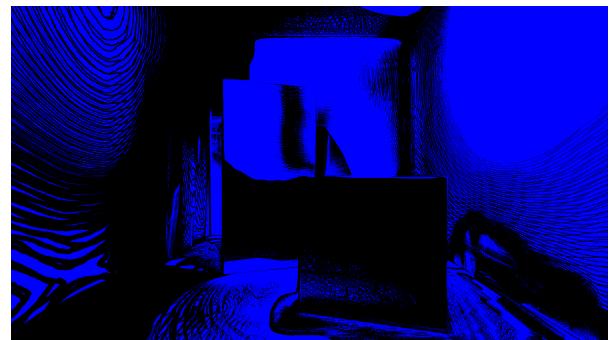
Directa, Indirecta y Oclusión Ambiental



Diferencia Perceptual



Factor de longitud de marcha: 1



2.5

Figura 72: Diferencia con respecto al resultado en la Figura 60 utilizando distintos factores de longitud.

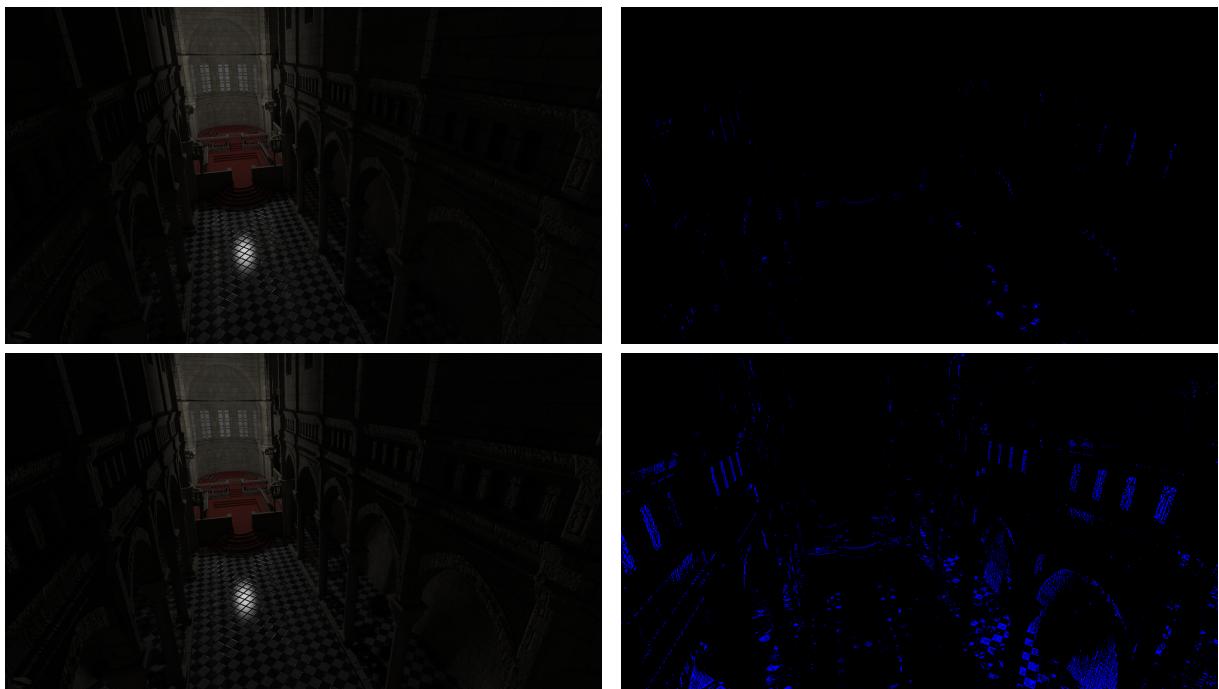


Figura 73: Diferencia con respecto al resultado en la Figura 61 utilizando distintos factores de longitud.

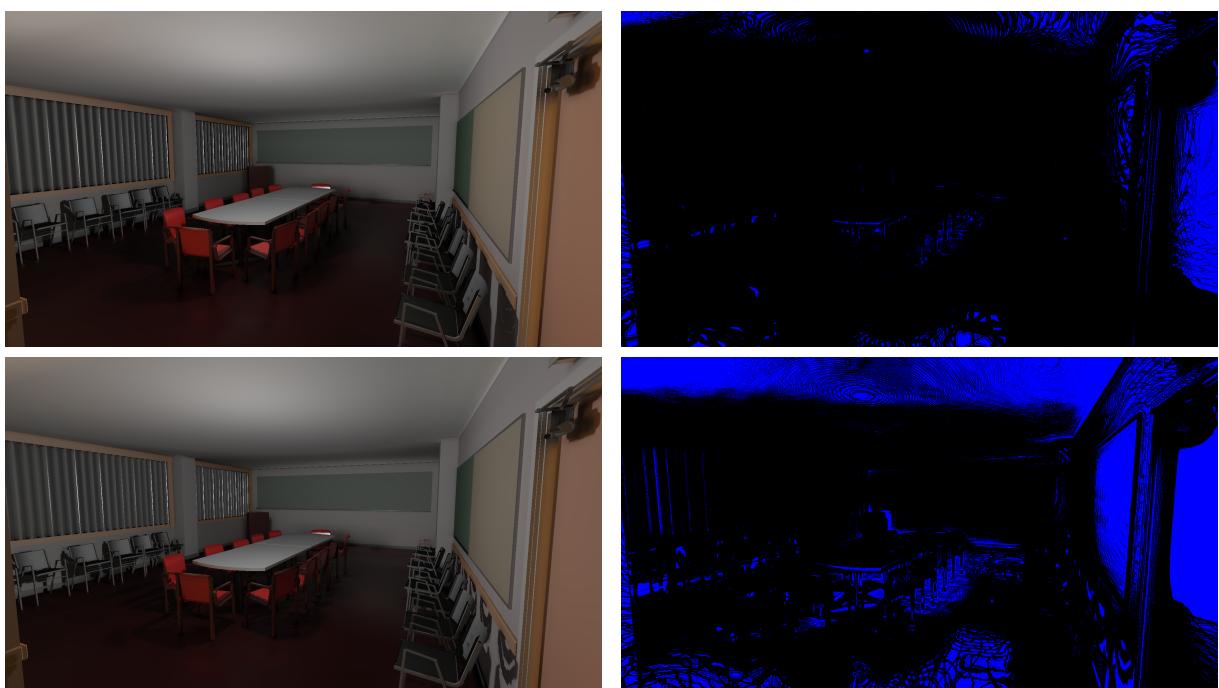


Figura 74: Diferencia con respecto al resultado en la Figura 63 utilizando distintos factores de longitud.

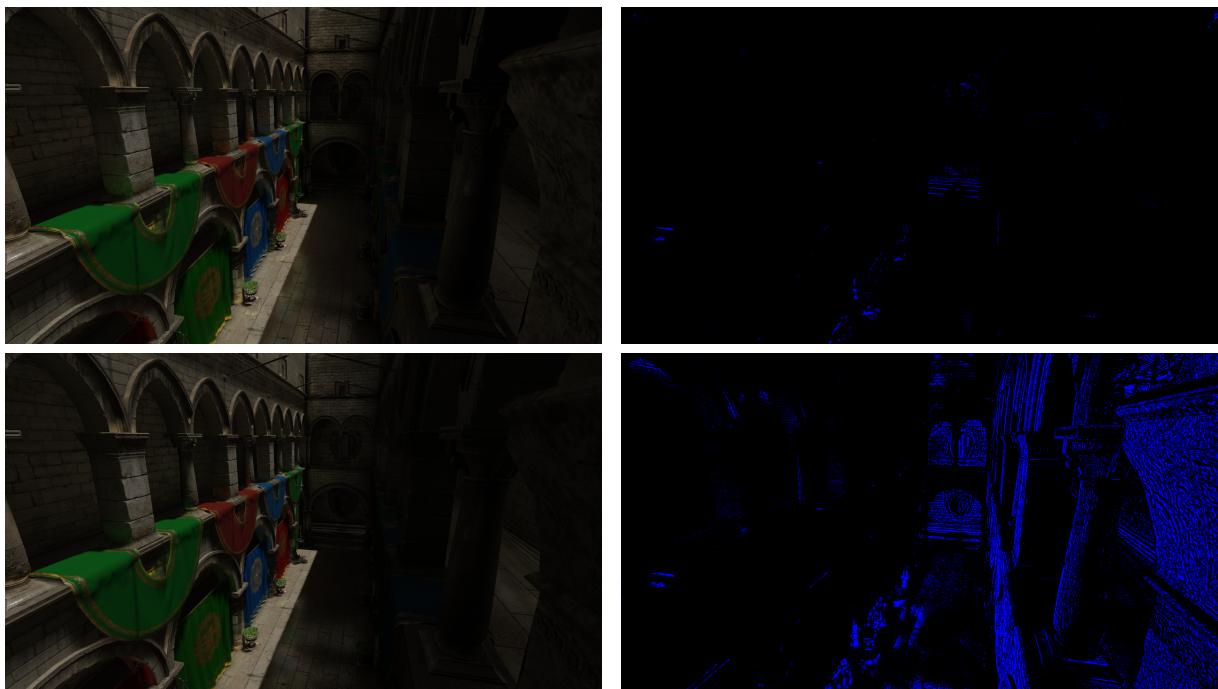


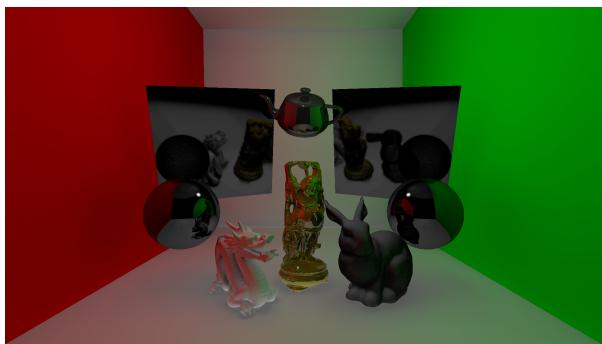
Figura 75: Diferencia con respecto al resultado en la Figura 62 utilizando distintos factores de longitud.

#### 4.4.2. Reflexión Especular y Factor de Longitud de Marcha

El factor de longitud afecta especialmente la calidad de las reflexiones especulares finas. En la Figura 77 se puede observar como empiezan a aparecer anomalías y vacíos en la reflexión specular a medida que se aumenta el factor de longitud.



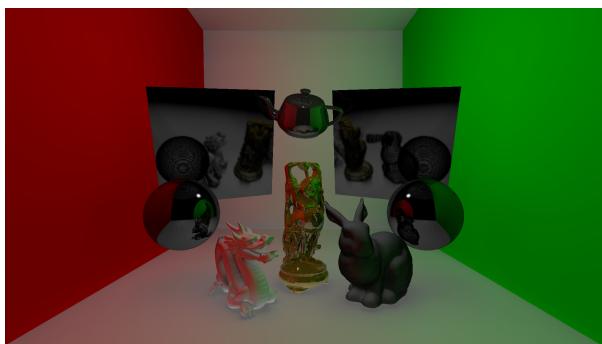
Figura 76: Demostración de reflexión especular con factor de longitud de marcha 0.1 y volúmenes con resolución de  $512^3$  véxoles.



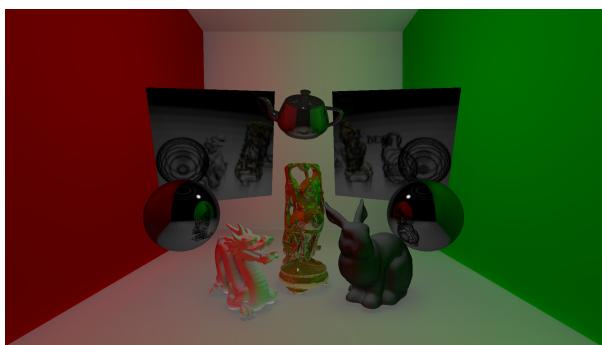
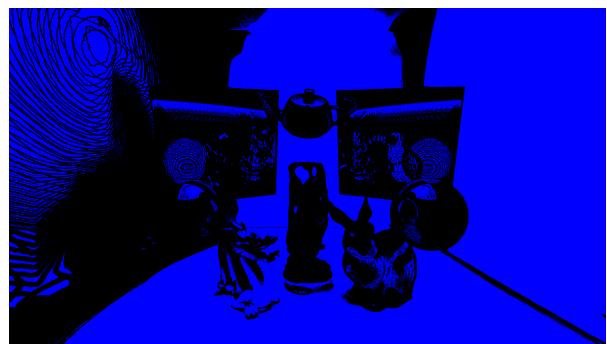
Factor de longitud de marcha: 0.5



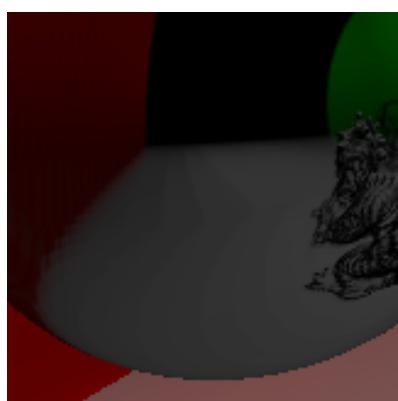
Diferencia con la Figura 76



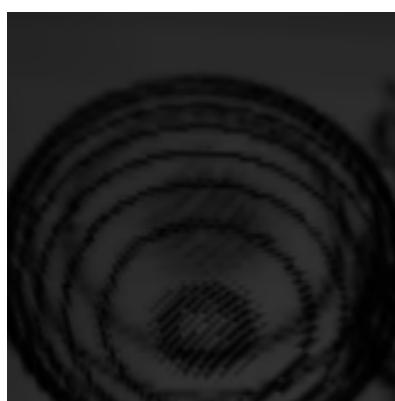
1.0



2.5



Bandas de Color



Salto de Vóxeles.



Rayado o *Striping*

Figura 77: Diferencia con respecto a la Figura 76 considerando distintos factores de longitud. En la última imagen se puede observar en detalle algunos problemas visuales que surgen al incrementar longitud de marcha del cono.

#### 4.4.3. Apertura del Cono para Trazado de Sombras Suaves

Una de las características del trazado de conos es que mientras mayor es la apertura del cono más rápido es el trazado (ver Figura 59). Esto presenta una ventaja para el trazado de sombras suaves ya que a medida que se abre el cono de sombreado más suaves y esparcidas son las sombras resultantes como se observa en la Figura 78.

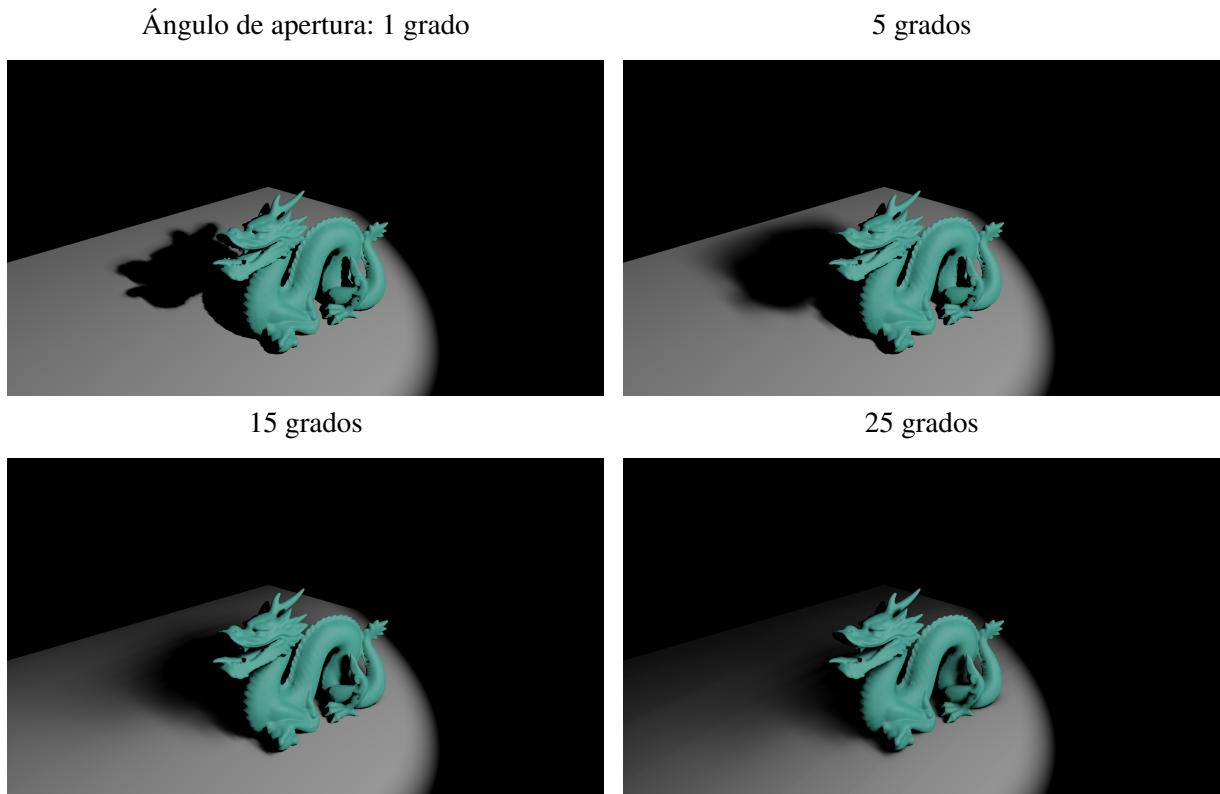


Figura 78: Sombras suaves generadas bajo distintas aperturas del cono de sombreado.

##### 4.4.3.1. Mapeo del Volumen de Visibilidad

Durante el proceso de sombreado se almacena la visibilidad de cada voxel en el volumen de visibilidad. Estos valores pueden ser utilizados para mapeo de sombras proyectando la posición del fragmento en espacio de textura para obtener un valor de oclusión aproximado. Debido a la baja resolución de la representación del volumen, esta técnica provee sombras de baja calidad, pero a un rendimiento más alto como se observa en la Figura 59. En esta sección se mostrarán las tres técnicas de sombreado implementadas: mapeo de sombras, sombras con trazado de conos, mapeo del volumen de visibilidad. Para el mapeo de sombras se utilizó un mapa con resolución de  $512 \times 512$  píxeles, la resolución de la representación en voxels fue de

$512^3$  vóxeles y el factor de longitud de marcha fue 0.5. Se utilizó la escena sandbox Plane Test con distintas primitivas de diferente complejidad geométrica y una fuente de luz direccional.

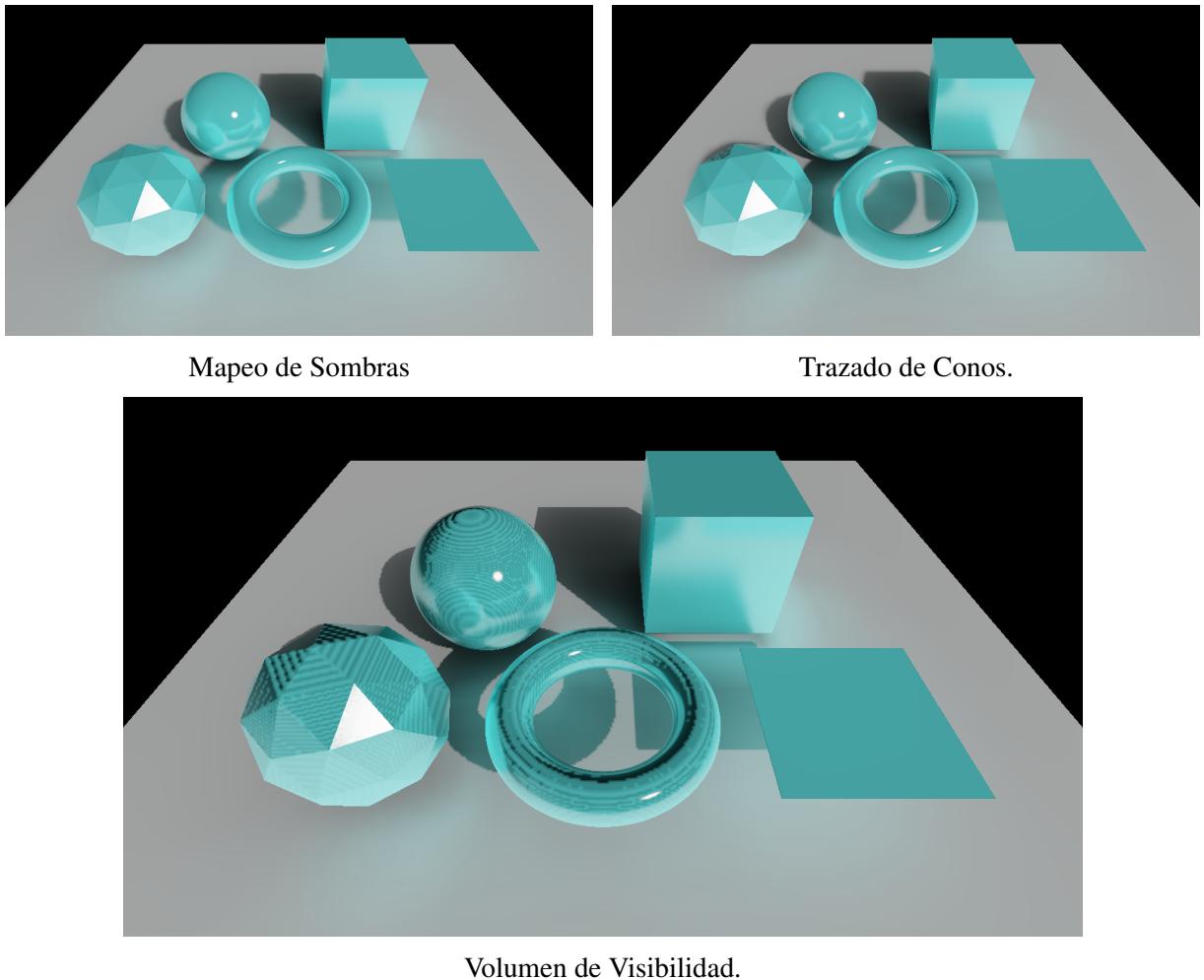


Figura 79: Distintas técnicas para la generación de sombras sobre objetos de distinta complejidad geométrica.

Para la generación de sombras tanto el trazado de conos como el mapeo del volumen de visibilidad, puede ocasionar errores sobre geometría suave como la esfera o el toro. Mientras el mapeo de sombras no parece presentar estos problemas. En la Figura 79 el mapeo del volumen de visibilidad produce considerables artefactos visuales para toda geometría no definida por ángulos rectos, como el icosaedro, el toro y la esfera, haciendo su uso muy limitado. Sin embargo para objetos como el cubo y el plano, la técnica produce resultados similares a las otras técnicas implementadas.

#### 4.4.4. Materiales Emisivos

Nuestra implementación permite la aproximación de materiales emisivos con la adición de otro volumen durante el proceso de voxelización. Estos materiales pueden ser utilizados para simular luces de área. En esta sección se puede observar estos materiales en acción.



Figura 80: Modelo Utah Teapot emisivo junto a otros objetos precargados.

En la Figura 80 se observa la escena sandbox Cornell Box Vació sin fuentes de luz tradicionales, el modelo de Utah Teapot tiene un material emisivo blanco que ilumina el resto de los objetos cercanos. La oclusión ambiental esta desactivada para esta imagen. Las sombras debajo de los objetos son generadas de forma natural por el trazado de conos contra véxeles.



Figura 81: Escena Sponza iluminada por varios materiales emisivos de distintos colores.

En la Figura 81 se puede observar en la escena completa Sponza mezcla de colores por distintos materiales emisivos y reflexión especular de estos materiales.

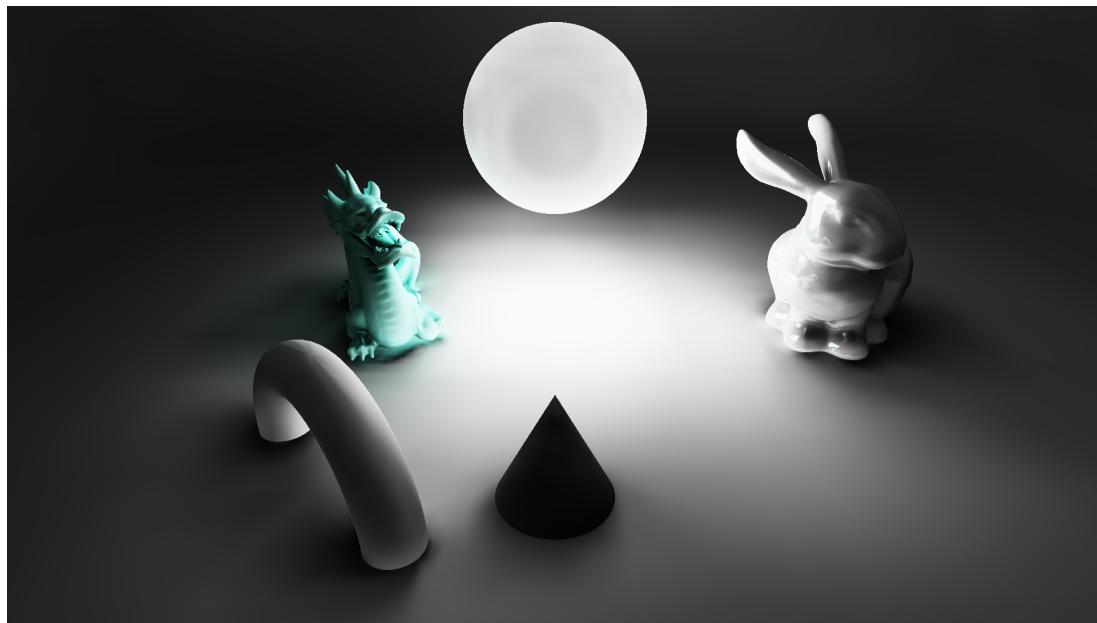


Figura 82: Sombras suaves generadas por materiales emisivos.

En la Figura 82 se puede observar en la escena sandbox LightBox la generación de sombras suaves y su direccionalidad con respecto al material emisivo en la esfera.

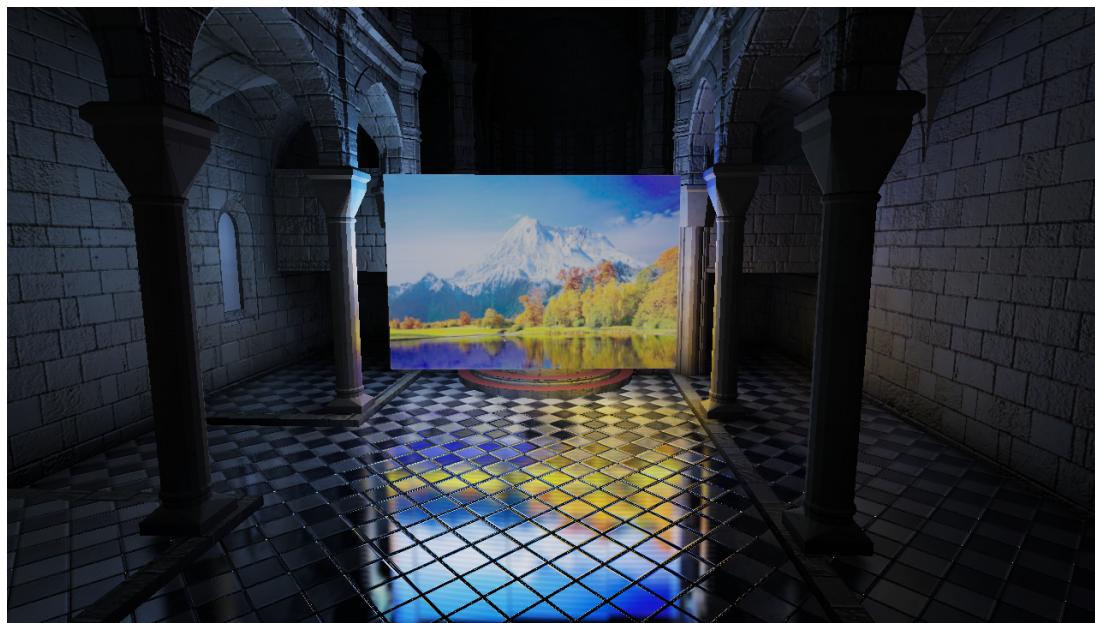


Figura 83: Detalles finos en materiales con emisión.

En la Figura 83 se puede observar en la escena completa Sibenik los detalles de emisión para materiales con texturizado, el color resultante en la pared izquierda y derecha son distintos.

#### 4.4.5. Defectos o Artefactos Visuales

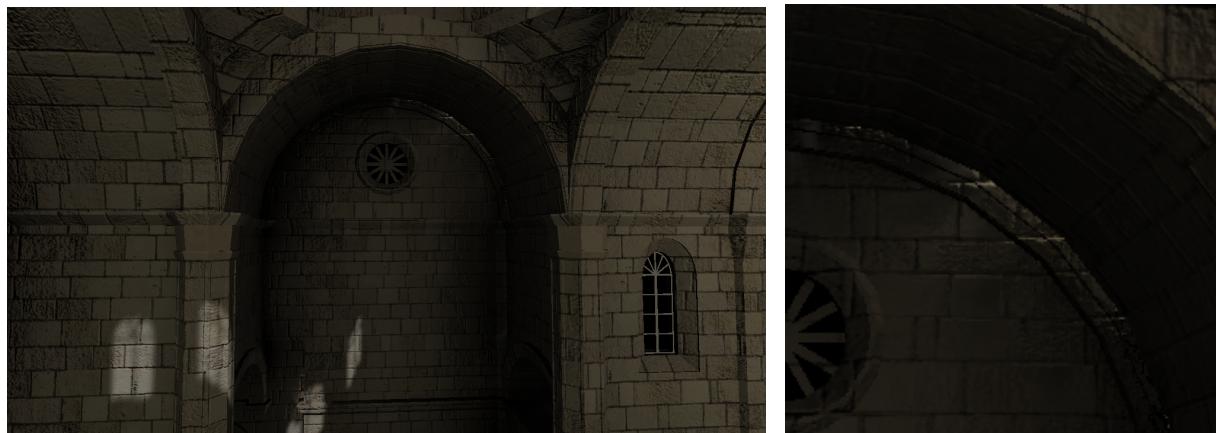


Figura 84: Fuga de luz o *light-leaking* en escena Sibenik.

En la Figura 84 se puede observar un defecto visual común del algoritmo. Este puede producirse por defectos de voxelización cuando se falta o excede la cantidad de vértices. Esto

sucede usualmente por errores en la geometría de la malla. Sin embargo también puede ser causado por el mismo algoritmo de voxelización por problemas de precisión en el proceso de rasterización al generar véxeles en los bordes de la geometría poligonal. Para este caso una forma de reducir este problema es utilizando anti-aliasing con multimuestreo (MSAA) [30]. Los véxeles faltantes o sobrantes pueden causar que se iluminen véxeles que no deberían iluminarse durante el proceso de sombreado y/o de iluminación global de véxeles.

Otra condición en la que esto puede suceder es durante el trazado de conos con véxeles cuando el trazado salta algunas de las muestras durante el recorrido ya sea dentro del nivel más alto de detalle o por falta de precisión en la interpolación de los niveles de mipmap. La utilización de véxeles anisotrópicos reduce en gran medida esta posibilidad.

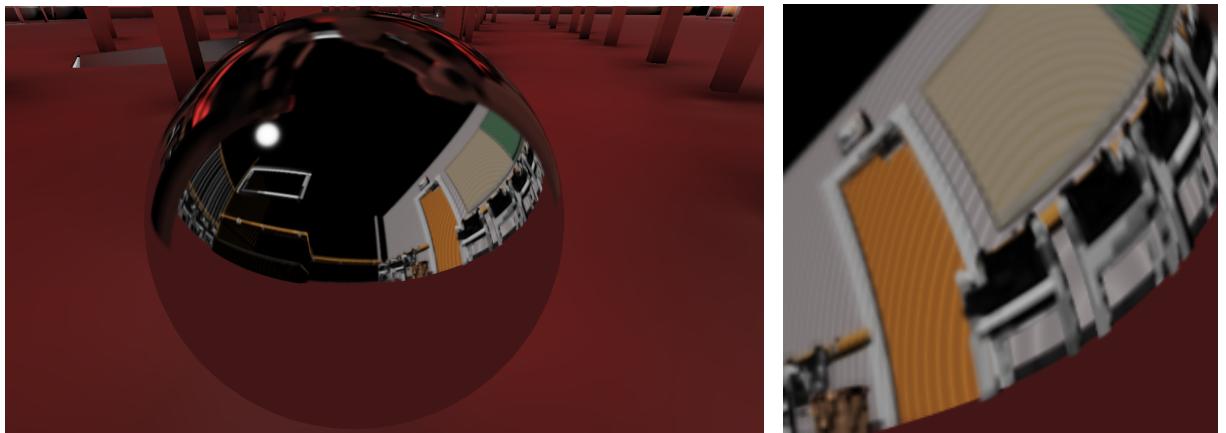


Figura 85: Bandas de colores o *color banding*.

La aparición de bandas de colores sucede por problemas de muestreo durante el trazado de conos. La distancia entre muestras produce estas bandas súbitas producto de la interpolación trilineal entre un mismo nivel de detalle o interpolación cuadrilínea entre varios. Una forma de reducir esto es utilizando un cono de menor apertura sacrificando reflectividad en el caso de conos especulares o reduciendo la longitud de marcha del cono a cambio de menor rendimiento.

#### 4.4.6. Comparación

Para realizar una comparación de calidad se utilizó el algoritmo CLPV el cual tiene resultados en rendimiento similares a nuestra implementación. Se utilizó la implementación realizada por NVIDIA en su DirectX 11 SDK [36]. Esta aproximación solo comprende iluminación indirecta difusa. En la figura 86 se puede observar que los bloques debajo de las columnas

reflejan las áreas iluminadas del pasillo superior en la escena completa Sponza por reflexión especular. La reflexión difusa sobre las columnas es un poco más extensa en CLPV causando mezclado de colores hasta los arcos entre las columnas y el techo del pasillo. En nuestra implementación la reflexión difusa y especular permite una mayor iluminación del pasillo. Se puede observar también la reflexión especular de la luz directa en el pasillo sobre la pared debajo de las columnas, esto no está presente en CLPV.



Figura 86: Comparación de nuestra implementación (a) con algoritmo para el cálculo de iluminación indirecta con volúmenes de propagación de luz en cascada (b).

## 4.5. Estudio de Memoria

Para examinar el consumo de memoria en la GPU se utilizó la herramienta ProcExplorer. Nuestra implementación no utiliza una estructura de octree disperso por tanto se almacenan véxeles sean estos vacíos o no. Esto implica un mayor uso de memoria. En la Tabla 10 se puede observar la memoria reservada por resolución de la representación en véxeles. Estos valores son constantes sobre cualquier escenario en la aplicación. En la Tabla 10 se pueden observar los resultados.

Resolución de la Representación en Vóxeles	Memoria Reservada
$512^3$ véxeles	2800 MB
$256^3$ véxeles	540.3 MB
$128^3$ véxeles	253 MB
$64^3$ véxeles	216.7 MB

Tabla 10: Consumo de memoria en la GPU para distintas resoluciones de la representación en véxeles.

# Capítulo 5

## Conclusiones

En este trabajo se han estudiado distintos algoritmos para el cálculo de iluminación indirecta en tiempo real. Se ha implementado una aproximación utilizando vóxeles y trazado de conos. El trazado de conos para *ray tracing* fue propuesto por Amanatides en 1984 [39]. El mayor problema de esta aproximación es que calcular la colisión entre un cono y polígonos, es usualmente una operación muy compleja. Utilizando una representación simplificada de la escena en vóxeles, se evita esta desventaja realizando ahora un trazado aproximado. Por cada punto sobre una superficie se trazan conos para acumular la iluminación indirecta difusa y especular. Estos conos son trazados sobre la semiesfera orientada según el vector normal de este punto basándose en integración Monte Carlo. Esta acumulación es realizada utilizando muestreo sobre una estructura de vóxeles. A medida que el cono realiza su recorrido su diámetro se expande por tanto aumenta el tamaño del volumen a muestrear. Para esto se utiliza una estructura de vóxeles piramidal basada en *mipmapping* con distintos niveles de detalle.

Nuestra implementación provee de manera efectiva una aproximación para la iluminación indirecta de uno y dos rebotes de luz. Esta implementación comprende distintos fenómenos de iluminación como reflexión difusa, reflexión especular, oclusión ambiental, sombras suaves, sombras indirectas, mezclado de colores y materiales emisivos. A pesar de esto, algunos defectos son de esperar al tratarse de aproximaciones. Los mayores problemas visuales surgen por discontinuidad entre vóxeles. Es posible obtener valores erróneos por distintas causas como longitud de marcha o errores de voxelización. Esto usualmente ocasiona *light leaking* o escape de luz en algunas partes de ciertas escenas. Otro problema notable es la aparición de *color banding* o bandas de colores sobre reflexiones especulares finas. Esto es un error de muestreo sobre los vóxeles, que puede solucionarse disminuyendo la longitud de marcha del cono. Sin embargo esto implica un alto costo en rendimiento.

Este trabajo utiliza texturas tridimensionales para almacenar la representación en vóxeles. Esto simplifica la construcción de la estructura jerárquica y el filtrado cuadrilíneal utilizado durante el muestreo en el trazado de conos. La captura de la iluminación directa sobre los vóxeles se realiza utilizando *compute shaders*; para esto se voxeliza también las normales de la escena. Debido a que un promedio de normales puede traer ciertos problemas de disparidad, se implementó un modelo de sombreado para los vóxeles basado en la atenuación normal por cada cara del voxel ponderada. Para obtener resultados precisos durante el trazado de conos es necesario incluir oclusión durante el sombreado de vóxeles. Nuestra propuesta realiza *ray tracing* sobre un volumen producto de la voxelización y además se implementó un método para la generación de sombras suaves, basado en el número de colisiones de un rayo contra geometría voxelizada. Con la voxelización de las normales este trabajo permite el cálculo de iluminación global sobre la representación de vóxeles. Esto se hace utilizando el mismo algoritmo de trazado de conos contra vóxeles en un *compute shader*, solo que este se realiza desde vóxeles en vez de fragmentos.

## 5.1. Trabajos Futuros

Nuestra implementación provee buenos resultados tanto en calidad visual como en rendimiento, sin embargo podría beneficiarse de futuros estudios y ciertas mejoras en la aplicación. En nuestra implementación el método de voxelización utilizado es conservativo con solo el contorno. Para solventar problemas de *light-leaking*, explorar alternativas de voxelización sólida (incluyendo vóxeles internos en la geometría) podría ayudar a reducir estos problemas, ya que este es causado usualmente por la opacidad de vóxeles cercanos. Además de esto un esquema de voxelización sólida podría abrir la posibilidad a explorar otras técnicas de iluminación, como objetos con transluminiscencia [40].

Un problema en nuestra propuesta es el consumo en memoria. Esto podría reducirse utilizando la estructura *octree* dispersa. Sin embargo esto trae un costo agregado para la construcción de esta estructura y su recorrido. Una posibilidad reciente es el uso de texturas dispersas. Sin embargo, las extensiones que proveen esta funcionalidad en OpenGL son muy recientes, y para el momento de este trabajo la reserva de espacios de memoria sobre estas texturas todavía se hace del lado del CPU, lo cual puede ser muy lento cuando se considera la actualización dinámica de esta estructura.

El nivel de detalle de voxelización depende del espacio de voxelización y la resolución de la cuadrícula de vóxeles. Usualmente la información más relevante al usuario se encuentra

cercana al observador. Esta cualidad se podría explotar para realizar voxelización por distintas cuadrículas, donde las cuadrículas cercanas al observador proveen mayor nivel de detalle para la representación en vértices. Esto permitiría manejar escenas de gran escala con mayor facilidad. Existen ya algunos recursos relacionados a esto como volúmenes de vértices en cascada [41].

El trazado de conos puede ser acelerado utilizando variadas técnicas para reducir el número de píxeles a muestrear. Una técnica sencilla es realizar el cálculo de iluminación indirecta en un paso a parte a menor resolución y luego integrar este resultado como un post-proceso. Esto puede hacer que se pierda precisión especialmente en los bordes de la geometría en escena. Para estos casos se puede utilizar algún algoritmo de detección de bordes y realizar el trazado de conos a resolución completa en estos píxeles. Los problemas de color banding en la reflexión especular pueden ser solventados de manera más eficiente utilizando *dithering* sobre la reflexión especular resultante en este post-proceso.

# Bibliografía

- [1] P. Christensen. (2003) Global Illumination and All That: Pixar Animation Studios. <http://renderman.pixar.com/view/global-illumination-and-all-that>.
- [2] P. Dutre, K. Bala, P. Bekaert, y P. Shirley, *Advanced Global Illumination*, 2da. ed. AK Peters Ltd, 2006.
- [3] A. Fournier, “Normal distribution functions and multiple surfaces,” en *Graphics Interface '92 Workshop on Local Illumination*, Vancouver, BC, Canadá, 11 Mayo 1992, pp. 45–52.
- [4] P. Bromiley, “Products and convolutions of gaussian distributions,” TINA Vision, Internal Report 2003-003, 2003.
- [5] M. Toksvig, “Mipmapping normal maps,” *J. Graphics Tools*, vol. 10, núm. 3, pp. 65–71, 2005. [En línea]. Disponible en: <http://dx.doi.org/10.1080/2151237X.2005.10129203>
- [6] Disney. (2012) Disney’s BRDF Explorer. <https://github.com/wdas/brdf>.
- [7] J. T. Kajiya, “The Rendering Equation,” en *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH ’86. New York, NY, USA: ACM, 1986, pp. 143–150. [En línea]. Disponible en: <http://doi.acm.org/10.1145/15922.15902>
- [8] D. Ritchie y R. Ramamoorthi, “Global Illumination and the Rendering Equation,” <https://inst.eecs.berkeley.edu/~cs294-13/fa09/lectures/scribe-lecture3.pdf>, 2009, university of California, Berkeley.
- [9] C. M. Goral, K. E. Torrance, D. P. Greenberg, y B. Battaile, “Modeling the Interaction of Light Between Diffuse Surfaces,” *SIGGRAPH Comput. Graph.*, vol. 18, núm. 3, pp. 213–222, Ene. 1984. [En línea]. Disponible en: <http://doi.acm.org/10.1145/964965.808601>
- [10] H. Elias, “Radiosity: The Workings of a Radiosity Renderer,” <http://freespace.virgin.net/hugo.elias/radiosity/radiosity.htm>, 2000.

- [11] J. Arvo, “Backward Ray Tracing,” en *In ACM SIGGRAPH '86 Course Notes - Developments in Ray Tracing*, 1986, pp. 259–263.
- [12] L. Do, “HW6: Ray Tracing Extension,” <https://inst.eecs.berkeley.edu/~cs184/sp12/assignments/Archive/HW6/>, 2013.
- [13] T. Akenine-Moller, T. Moller, y E. Haines, *Real-Time Rendering*, 2da. ed. Natick, MA, USA: A. K. Peters, Ltd., 2002.
- [14] L. Williams, “Casting curved shadows on curved surfaces,” en *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '78. New York, NY, USA: ACM, 1978, pp. 270–274. [En línea]. Disponible en: <http://doi.acm.org/10.1145/800248.807402>
- [15] M. Deering, S. Winner, B. Schediwy, C. Duffy, y N. Hunt, “The triangle processor and normal vector shader: A vlsi system for high performance graphics,” en *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '88. New York, NY, USA: ACM, 1988, pp. 21–30. [En línea]. Disponible en: <http://doi.acm.org/10.1145/54852.378468>
- [16] T. Saito y T. Takahashi, “Comprehensible rendering of 3-d shapes,” *SIGGRAPH Comput. Graph.*, vol. 24, núm. 4, pp. 197–206, Sep. 1990. [En línea]. Disponible en: <http://doi.acm.org/10.1145/97880.97901>
- [17] J. Huang, R. Yagel, V. Filippov, y Y. Kurzion, “An accurate method for voxelizing polygon meshes,” en *Proceedings of the 1998 IEEE Symposium on Volume Visualization*, VVS '98. New York, NY, USA: ACM, 1998, pp. 119–126. [En línea]. Disponible en: <http://doi.acm.org/10.1145/288126.288181>
- [18] A. Keller, “Instant radiosity,” en *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997, pp. 49–56. [En línea]. Disponible en: <http://dx.doi.org/10.1145/258734.258769>
- [19] C. Dachsbacher, J. Křivánek, M. Hašan, A. Arbree, B. Walter, y J. Novák, “Scalable realistic rendering with many-light methods,” *Computer Graphics Forum*, vol. 33, núm. 1, pp. 88–104, 2014.
- [20] C. Dachsbacher y M. Stamminger, “Reflective shadow maps,” en *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, I3D '05. New

York, NY, USA: ACM, 2005, pp. 203–231. [En línea]. Disponible en: <http://doi.acm.org/10.1145/1053427.1053460>

- [21] A. Kaplanyan y C. Dachsbacher, “Cascaded light propagation volumes for real-time indirect illumination,” en *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D ’10. New York, NY, USA: ACM, 2010, pp. 99–107. [En línea]. Disponible en: <http://doi.acm.org/10.1145/1730804.1730821>
- [22] K. Crane, I. Llamas, y S. Tariq, *Real Time Simulation and Rendering of 3D Fluids*. Addison-Wesley, 2007, cap. 30.
- [23] C. Crassin, F. Neyret, M. Sainz, S. Green, y E. Eisemann, “Interactive indirect illumination using voxel cone tracing,” *Computer Graphics Forum (Proceedings of Pacific Graphics 2011)*, vol. 30, núm. 7, sep 2011. [En línea]. Disponible en: <http://maverick.inria.fr/Publications/2011/CNSGE11b>
- [24] P. Cozzi y C. Riccio, *OpenGL Insights*. CRC Press, July 2012, <http://www.openglinsights.com/>.
- [25] L. Zhang, W. Chen, D. S. Ebert, y Q. Peng, “Conservative voxelization,” *The Visual Computer*, vol. 23, núm. 9-11, pp. 783–792, 2007.
- [26] J. Hasselgren, T. Akenine-Möller, y L. Ohlsson, “Conservative rasterization,” *The Visual Computer*, 2005.
- [27] P. B. Jeff Bolz. (2011) ARB\_shader\_image\_load\_store. [https://www.opengl.org/registry/specs/ARB/shader\\_image\\_load\\_store.txt](https://www.opengl.org/registry/specs/ARB/shader_image_load_store.txt).
- [28] P. Oliver, “Unreal engine 4 elemental,” en *ACM SIGGRAPH 2012 Computer Animation Festival*, SIGGRAPH ’12. New York, NY, USA: ACM, 2012, pp. 86–86. [En línea]. Disponible en: <http://doi.acm.org/10.1145/2341836.2341909>
- [29] B. Sieprawski. (2014) Unreal Engine 4.6 Release Notes. <https://www.unrealengine.com/blog/unreal-engine-46-released>.
- [30] M. Takeshige. (2015) The Basics of GPU Voxelization. <https://developer.nvidia.com/content/basics-gpu-voxelization>.
- [31] E. Reinhard, M. Stark, P. Shirley, y J. Ferwerda, “Photographic tone reproduction for digital images,” en *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH ’02. New York, NY, USA: ACM, 2002, pp. 267–276. [En línea]. Disponible en: <http://doi.acm.org/10.1145/566570.566575>

- [32] F. Meinl. (2010) Sponza Model. <http://www.crytek.com/cryengine/cryengine3/downloads>.
- [33] A. Grynberg y G. Ward. (1991) Conference Room. <http://graphics.cs.williams.edu/data/meshes.xml>.
- [34] M. Dabrovic. (2002) Sibenik Cathedral. <http://graphics.cs.williams.edu/data/meshes.xml>.
- [35] W. College. (2012) Cornell Box. <http://graphics.cs.williams.edu/data/meshes.xml>.
- [36] NVIDIA. (2016) DX11 Samples. <https://developer.nvidia.com/dx11-samples>.
- [37] S. Yeung. (2013) Implementing Voxel Cone Tracing. <http://simonstechblog.blogspot.com/2013/01/implementing-voxel-cone-tracing.html>.
- [38] H. Yee, S. Pattanaik, y D. P. Greenberg, “Spatiotemporal sensitivity and visual attention for efficient rendering of dynamic environments,” *ACM Trans. Graph.*, vol. 20, núm. 1, pp. 39–65, Ene. 2001. [En línea]. Disponible en: <http://doi.acm.org/10.1145/383745.383748>
- [39] J. Amanatides, “Ray tracing with cones,” en *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH ’84. New York, NY, USA: ACM, 1984, pp. 129–135. [En línea]. Disponible en: <http://doi.acm.org/10.1145/800031.808589>
- [40] E. Eisemann y X. Décoret, “Single-pass gpu solid voxelization for real-time applications,” en *Proceedings of Graphics Interface 2008*, GI ’08. Toronto, Ont., Canadá, Canadá: Canadian Information Processing Society, 2008, pp. 73–80. [En línea]. Disponible en: <http://dl.acm.org/citation.cfm?id=1375714.1375728>
- [41] J. McLaren y T. Yang, “The tomorrow children: Lighting and mining with voxels,” en *ACM SIGGRAPH 2015 Talks*, SIGGRAPH ’15. New York, NY, USA: ACM, 2015, pp. 67:1–67:1. [En línea]. Disponible en: <http://doi.acm.org/10.1145/2775280.2792546>