



UNIVERSIDAD  
DE LA REPÚBLICA  
URUGUAY



FACULTAD DE  
INGENIERÍA

# Fingxels

Informe de Proyecto de Grado presentado por

Francisco Aguirre, Felipe Pizzorno

en cumplimiento parcial de los requerimientos para la graduación de la carrera  
de Ingeniería en Computación de Facultad de Ingeniería de la Universidad de  
la República

Supervisor

Eduardo Fernández

Montevideo, 21 de agosto de 2024



Fingxels por Francisco Aguirre, Felipe Pizzorno tiene licencia [CC Atribución 4.0](#).

# Agradecimientos

Agradecer, siempre es bueno agradecer.



# Resumen

En este trabajo se aborda el problema de la iluminación global en tiempo real, un problema desafiante y relevante en la industria de los videojuegos, cine, simulaciones y la computación gráfica en general. La mayoría de las técnicas de iluminación global se basan en el método de Monte Carlo, y esto las hace computacionalmente costosas, requiriendo hardware especializado y técnicas de aprendizaje automático para alcanzar el tiempo real.

En este contexto, se hace foco en el algoritmo de *voxel cone tracing*, una técnica de trazado de conos que usa una estructura de datos basada en véxeles que se ha demostrado prometedora por su capacidad para producir efectos de iluminación global de alta calidad en tiempo real sin necesidad de hardware especializado.

El objetivo principal de este trabajo es crear una implementación open source del algoritmo de *voxel cone tracing* y probarla en hardware moderno. La implementación realizada no solo demuestra la viabilidad y eficacia de la técnica en un entorno de hardware actual, sino que también es un recurso didáctico valioso para cualquier persona interesada en aprender acerca de computación gráfica, iluminación global y su implementación.

Para esta implementación, se utilizaron principalmente las herramientas Rust y OpenGL, debido a su alto rendimiento, abundancia de documentación y simplicidad. El algoritmo corre exclusivamente en la GPU, aprovechando la alta paralelización que esta provee. Su desarrollo se realizó en Linux, pero el programa no está limitado a este sistema operativo dado que puede ser portado fácilmente a otros.

Para evaluar la implementación, se realizaron experimentos en hardware moderno y se compararon los resultados con implementaciones previas de la técnica.

El código del motor se encuentra en el siguiente repositorio:

<https://github.com/franciscoaguirre/voxel-cone-tracing>

**Palabras clave:** Voxel cone tracing, Iluminación global, OpenGL, Rust

# Índice general

<b>1. Introducción</b>	<b>1</b>
1.1. Organización del documento . . . . .	2
<b>2. Revisión de antecedentes</b>	<b>3</b>
2.1. ¿Qué es la luz? . . . . .	3
2.2. Radiometría: Unidades de la luz . . . . .	5
2.3. Iluminación local e iluminación global . . . . .	7
2.3.1. Iluminación local . . . . .	7
2.3.2. Iluminación global . . . . .	8
2.4. BRDF . . . . .	10
2.5. Trazado de rayos ( <i>ray tracing</i> ) . . . . .	11
2.6. Trazado de conos . . . . .	12
2.7. Photon Mapping . . . . .	14
2.8. Vóxeles . . . . .	16
2.9. Octrees . . . . .	17
2.10. Ducto gráfico . . . . .	17
2.11. Renderizado diferido . . . . .	20
<b>3. Voxel cone tracing</b>	<b>21</b>
3.1. Voxelización . . . . .	22
3.1.1. Rasterización conservativa . . . . .	24
3.2. <i>Octree</i> disperso . . . . .	25
3.2.1. Nodos, bricks y vóxeles . . . . .	26
3.2.2. Construcción . . . . .	27
3.2.3. Border transfer . . . . .	30
3.2.4. Nodos frontera . . . . .	31
3.2.5. Filtrado . . . . .	34
3.2.6. Filtrado anisotrópico . . . . .	36

3.3. Inyección de fotones . . . . .	38
3.4. <i>Cone tracing</i> . . . . .	39
3.5. Oclusión ambiental . . . . .	41
3.6. Conos de sombra . . . . .	41
<b>4. Implementación</b>	<b>43</b>
4.1. <i>Engine</i> . . . . .	44
4.2. Core . . . . .	46
4.2.1. Etapas del algoritmo . . . . .	46
4.2.2. Representación del <i>octree</i> . . . . .	47
4.2.3. Menú . . . . .	51
4.3. CLI . . . . .	51
4.3.1. Archivo de configuración . . . . .	52
4.3.2. Archivos de escena . . . . .	52
4.3.3. Archivos de valores predeterminados . . . . .	53
<b>5. Experimentación</b>	<b>55</b>
5.1. Análisis de resultados . . . . .	56
<b>6. Conclusiones y Trabajo Futuro</b>	<b>61</b>

# Capítulo 1

## Introducción

El problema de la iluminación global en tiempo real ha sido muy estudiado. Resolverlo es uno de los objetivos largamente buscados de la computación gráfica, debido a que tiene una alta importancia en varias industrias, como la de los videojuegos, la del cine, las simulaciones físicas, entre otros. Existen varios métodos para resolver el problema de la iluminación global. La mayoría de ellos se basan en el trazado de rayos, que consiste en trazar rayos a partir de la cámara hacia la escena y simular los caminos que recorren los fotones. Su alto costo computacional limita su uso en tiempo real. Se han realizado varias optimizaciones a lo largo de los años; como por ejemplo simplificaciones en la geometría [Cra<sup>+</sup>09], el uso de estructuras jerárquicas [SA19], clustering [Wan<sup>+</sup>09], entre otras. Más allá de los avances, el problema continúa siendo un área activa de investigación. Métodos recientes utilizan hardware especializado para alcanzar tiempos interactivos. Recientemente, han habido muchos avances en utilizar la inteligencia artificial (AI) para realizar los cálculos en una menor resolución y escalarla a la resolución objetivo [WCH20], ahorrando costos.

El trabajo presente se centra en *voxel cone tracing*, un algoritmo de iluminación global que funciona en tiempo real sin necesidad de hardware específico. El objetivo del trabajo es aprender sobre este algoritmo, crear una implementación del mismo, y probar su eficiencia en hardware moderno. El algoritmo fue propuesto por Crassin et al en 2011 [Cra<sup>+</sup>11], cuando no existían muchas soluciones para el problema de iluminación global en tiempo real, y la demanda estaba creciendo debido a la importancia de la industria de los videojuegos. Se basa en una representación de vértices de la geometría, el trazado de conos y el uso de estructuras jerárquicas de datos y pre-filtrado para reducir los cálculos necesarios y así alcanzar tiempos interactivos. No sufre de problemas de ruido

y provee una buena calidad de imágenes con un rendimiento casi independiente de la complejidad de la escena. Computa hasta dos rebotes de la luz en su camino desde el emisor hacia la cámara, lo que permite incorporar el componente principal de la luz indirecta.

El trabajo surge del interés de los miembros del equipo en técnicas de iluminación global y en véxoles como primitiva de renderizado. Luego de dos cursos de computación gráfica en los que se trata por un lado la creación de ambientes interactivos y por otro la generación de imágenes realistas usando iluminación global, se buscó un algoritmo que permitiera ambas, iluminación global en tiempo real. El proyecto de grado se realiza en el contexto de un ambiente académico, la implementación es open source y apunta a ser un recurso didáctico útil para otras personas interesadas en aprender sobre distintas técnicas de iluminación.

## 1.1. Organización del documento

Las siguientes secciones de este informe se organizan de la siguiente manera. El capítulo 2 se enfoca en la revisión de antecedentes de la técnica de *voxel cone tracing*. El capítulo 3 se enfoca en detallar cómo funciona el algoritmo y la estructura de datos utilizada para implementarlo. El capítulo 4 presenta algunas decisiones tomadas respecto al desarrollo. El capítulo 5 muestra los resultados de los experimentos realizados con la aplicación implementada que se presentan en forma de tablas e imágenes. Finalmente, el capítulo 6 resume los resultados y conclusiones de este trabajo y presenta las funcionalidades y arreglos para implementar a futuro.

## Capítulo 2

# Revisión de antecedentes

En este capítulo se explican en detalle los conceptos teóricos y los antecedentes más importantes para entender el algoritmo de *voxel cone tracing* y la implementación desarrollada.

### 2.1. ¿Qué es la luz?

Antes de definir el problema de la iluminación global en computación gráfica y hablar de diversos trabajos que se han realizado al respecto, conviene dar un paso atrás y preguntarnos, ¿qué es la luz? La pregunta es relevante porque, más allá de las variaciones, la gran mayoría de técnicas de iluminación global se basan en modelar físicamente la luz.

En el libro *Physically Based Rendering*, de Pharr, Jakob y Humphreys [PJH23], se brinda una explicación clara y resumida de la historia de la interacción entre los humanos y la luz. La percepción a partir de la luz es central para nuestra existencia. La incógnita de la naturaleza de la luz ha ocupado las mentes de grandes filósofos y físicos desde el comienzo de los tiempos. La antigua escuela filosófica hinduista de Vaisheshika (siglos VI a V antes de cristo) veía a la luz como una colección de pequeñas partículas viajando a través de rayos a una alta velocidad. Por otra parte, en el siglo V antes de cristo, el filósofo griego Empédocles postulaba que un fuego divino emergía de los ojos humanos y combinado con los rayos de luz del sol producía visión. Entre los siglos 18 y 19, eruditos como Isaac Newton, Thomas Young y Augustin-Jean Fresnel desarrollaron teorías conflictivas, donde algunas modelaban la luz como consecuencia de la propagación de ondas, y otras de partículas. Al mismo tiempo, André-Marie Ampère, Joseph-Louis Lagrange, Carl Friedrich Gauß y Michael

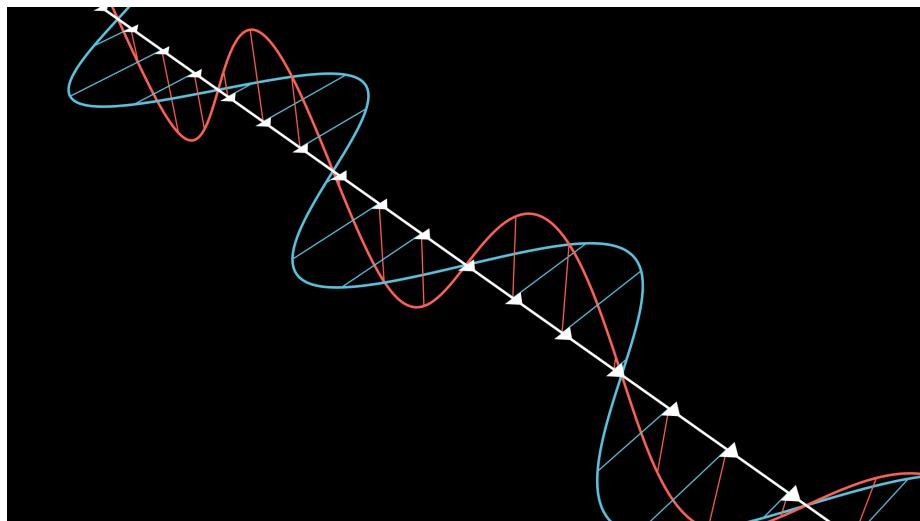


Figura 2.1: Representación de una onda electromagnética propagando por el espacio.

Faraday investigaban las relaciones entre electricidad y magnetismo que culminaron en la repentina y dramática unificación de James Clerk Maxwell en una teoría combinada conocida como **electromagnetismo**, [Max73].

La luz es una manifestación con propiedades de onda. El movimiento de partículas eléctricamente cargadas, como electrones dentro del filamento de una bombilla, produce un disturbio en un campo eléctrico circundante que se propaga hacia fuera de la fuente. La oscilación eléctrica también produce una oscilación secundaria de un campo magnético, que a su vez refuerza la oscilación del campo eléctrico, y así sucesivamente. La interacción entre estos dos campos da lugar a una onda que se autopropaga y puede viajar distancias extremadamente largas. Una representación de esto se puede ver en la Figura 2.1, en la que el campo eléctrico (azul) y el magnético (rojo) son perpendiculares el uno al otro y se propagan, avanzando a lo largo de un eje central.

A principios del siglo XX, los trabajos liderados por Max Planck, Max Born, Erwin Schrödinger y Werner Heisenberg condujeron a otro cambio sustancial en el entendimiento de la luz. A nivel microscópico, las propiedades elementales como energía y momento solo pueden existir como un múltiplo entero de una cantidad base conocida como un **cuanto**. En el caso de las oscilaciones electromagnéticas, este cuanto se conoce como **fotón**. La luz existe tanto como onda y como partícula [Lou65].

Afortunadamente, el complejo comportamiento de onda de la luz aparece en escalas muy pequeñas, por lo que, para la computación gráfica, en la mayoría de los casos se la puede tratar como partícula, lo que simplifica los cálculos [PJH23].

Tratar a la luz como una partícula es el campo de la óptica geométrica, en contraposición a la óptica física. La óptica geométrica trata a la luz como rayos que se mueven en líneas rectas. La óptica física aborda la luz desde el punto de vista de su naturaleza ondulatoria, centrándose en fenómenos como la interferencia, la difracción y la polarización. Si asumimos que las irregularidades de las superficies son en general mucho más grandes que la longitud de onda de la luz, estos efectos no ocurren, y se puede tratar la luz como rayos [Ake<sup>+18</sup>]. A su vez, se ignoran otros fenómenos como la fluorescencia y la fosforescencia.

## 2.2. Radiometría: Unidades de la luz

La radiometría provee una serie de herramientas para describir la propagación de la luz. Para simular luz, es necesario un manejo de las unidades básicas involucradas. Algunas de estas son la energía radiante, el flujo radiante, la radiosidad, la irradiancia, la intensidad radiante y la radiancia. La descripción de estas unidades surgen de [Ake<sup>+18</sup>] y [PJH23].

La **energía radiante** es la energía de la radiación electromagnética de la luz. Se mide en joules y se denota con el símbolo  $Q$ . Las fuentes de iluminación emiten fotones, y cada uno posee una longitud de onda y una energía particular. Un fotón con longitud de onda  $\lambda$  tiene una energía  $Q = \frac{hc}{\lambda}$ , donde  $c$  es la velocidad de la luz y  $h$  es la constante de Planck.

El **flujo radiante**, o potencia, es la energía que pasa por una superficie o región del espacio por unidad de tiempo:  $\Theta = \frac{dQ}{dt}$ . Se mide en joules por segundo, o watts. En la Figura 2.2 se representa en 2D una luz puntual emitiendo fotones en todas las direcciones. Los círculos de la figura son áreas en las que se mide el flujo radiante que pasa por ellas. Por cada círculo pasa el mismo flujo radiante, dado que la energía y el tiempo son los mismos.

Es útil también considerar el área por la cual pasa un flujo radiante. Podemos definir esto como  $E = \frac{\Theta}{A}$ . Esta cantidad se llama o **radiosidad** o **irradiancia** dependiendo de si el flujo está llegando o saliendo de una superficie respectivamente. Estas medidas tienen unidad watts por metro cuadrado. En la Figura 2.2, la irradiancia en el círculo externo es menor que en el interno, dado que el área aumenta cuadráticamente con la distancia.

Para definir la próxima unidad, es necesario definir el **ángulo sólido**, que es

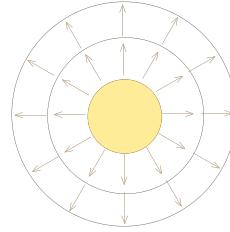


Figura 2.2: Flujo radiante en una luz puntual

la extensión a 3D del ángulo bidimensional. En 2D, un ángulo mide el tamaño de un conjunto continuo de direcciones en un plano. Para medir esto, se mide el largo del arco resultante de la intersección de este conjunto con un círculo de radio 1. El largo de este arco se mide en radianes. De igual manera, un ángulo sólido mide el tamaño de un conjunto de direcciones en el espacio. Para lograr esto, se mide el área de la intersección del conjunto de direcciones con una esfera de radio 1. La unidad de medida es el estereorradián, cuyo símbolo es sr. El ángulo sólido se representa con el símbolo  $\omega$ . En la Figura 2.3 se puede ver un cono con un ángulo sólido de 1 sr removido de una esfera.

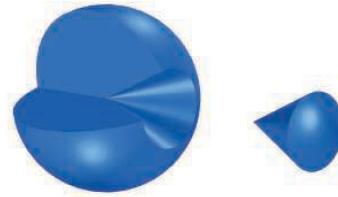


Figura 2.3: Un cono con un ángulo sólido de 1 sr removido de una esfera. Fuente: [Ake<sup>+</sup>18].

La **intensidad radiante** es el flujo radiante dada una dirección, o mejor dicho, un ángulo sólido. Se denota  $I(\omega) = \frac{d\Phi}{d\omega}$  y se mide en watts por estereoradián ( $Wsr^{-1}$ ).

Llegamos a la unidad radiométrica más importante, la **radiancia**. La radiancia es la cantidad de flujo radiante emitida, reflejada, transmitida o recibida por una superficie por unidad de área y por unidad de ángulo sólido. Se denota como  $L$  y se mide en watts por metro cuadrado por estereoradián ( $Wm^{-2}sr^{-1}$ ).

La radiancia es lo que miden los sensores, como los ojos o cámaras. El objetivo

Radiometría (unidad)	Fotometría (unidad)
Flujo radiante ( $W$ )	Flujo luminoso ( <i>lumen, lm</i> )
Radiosidad ( $\frac{W}{m^2}$ )	Emitancia luminosa ( $\frac{lm}{m^2} = lux, lx$ )
Irradiancia ( $\frac{W}{m^2}$ )	Iluminancia ( $lx$ )
Intensidad radiante ( $\frac{W}{sr}$ )	Intensidad luminosa ( <i>candela, cd</i> )
Radiancia ( $\frac{W}{(m^2 sr)}$ )	Luminancia ( $\frac{cd}{m^2} = nit$ )

Cuadro 2.1: Unidades de radiometría y fotogrametría.

de evaluar una ecuación de sombreado es calcular la radiancia a lo largo de un rayo, desde el punto de vista de la cámara.

Lo anteriormente expuesto son algunas unidades de la radiometría que trata únicamente con cantidades físicas de la luz. En contraposición, la fotometría (un campo relacionado) mide la luz en función de la percepción del ojo humano.

La radiometría trata únicamente con cantidades físicas de la luz. En contraposición, un campo relacionado, la fotometría, mide la luz tal como es percibida por el ojo humano, teniendo en cuenta la sensibilidad de este a distintas longitudes de onda. En la tabla 2.1 se muestran algunas unidades equivalentes en radiometría y fotometría.

## 2.3. Iluminación local e iluminación global

En computación gráfica, la iluminación es crucial para agregar realismo a una escena, sin embargo, es también uno de los aspectos más desafiantes desde el punto de vista computacional, debido a la complejidad de simular cómo la luz interactúa con los objetos y el entorno.

### 2.3.1. Iluminación local

La iluminación local es un enfoque más simple y menos demandante computacionalmente, que ayuda a dar una sensación de tridimensionalidad. Consiste en calcular la iluminación directa para cada objeto individualmente, sin considerar la iluminación indirecta que existe entre objetos vecinos. Al no calcular la interacción de la luz con otros objetos, se simplifican significativamente los cálculos necesarios para generar la imagen.

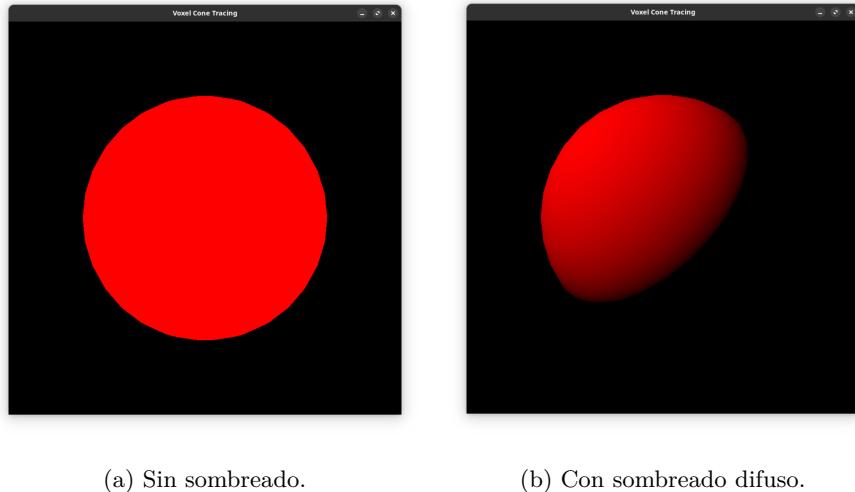


Figura 2.5: Esfera iluminada mediante el modelo local de Blinn-Phong.

Sus ventajas son su simplicidad y su eficiencia. Su mayor desventaja es que no abarca fenómenos como la refracción, las cáusticas y el sangrado, entre otros.

Un modelo ampliamente utilizado para iluminación local es el de Blinn-Phong [Bl177], que posee tres componentes principales: luz **ambiente**, **difusa** y **especular**. La luz ambiente es uniforme y está presente en toda la escena. No proviene de una fuente de luz en particular y simula el efecto de la luz indirecta que proviene del resto de la escena. La luz difusa representa el efecto de la luz directa que incide sobre una superficie rugosa y se refleja en todas las direcciones. Depende del ángulo entre la dirección de la luz y la normal de la superficie. La luz specular depende de la dirección de vista, de la normal de la superficie y de la dirección de la fuente luminosa en cada punto de la superficie. Simula el brillo que se ve cuando la luz se refleja sobre superficies pulidas. En la Figura 2.5 se puede observar este sombreado aplicado a una esfera.

### 2.3.2. Iluminación global

En contraposición a la iluminación local, la iluminación global calcula la interacción completa de la luz con los objetos de la escena. Se calcula la interacción de la luz entre distintos objetos al reflejarse, refractarse y dispersarse en el entorno.

Su principal ventaja es la mejora en el realismo y coherencia de la escena,

mientras que su principal desventaja es la exigencia computacional y complejidad al implementar y optimizar, que causa dificultades en alcanzar tiempos interactivos.

Entre las técnicas más utilizadas se encuentran el trazado de rayos, la radiosidad, el *photon mapping*, y el *path tracing*, que serán presentadas a continuación.

En 1986, James T. Kajiya presentó la **ecuación de renderizado**, formalizando varios métodos para el cálculo de iluminación global como aproximaciones a la solución de una misma ecuación, [Kaj86]:

$$I(x, x') = g(x, x') \cdot \left[ \epsilon(x, x') + \int_S f(x, x', x'') \cdot I(x', x'') \cdot dx'' \right] \quad (2.1)$$

donde:

- $I(x, x')$  se relaciona con la intensidad radiante que pasa del punto  $x'$  al punto  $x$ .
- $g(x, x')$  es un término de "geometría", evalúa si hay algo interponiéndose en el camino de  $x'$  a  $x$ .
- $\epsilon(x, x')$  se relaciona con la intensidad emitida desde  $x'$  en dirección a  $x$ .
- $f(x, x', x'')$  es la función de distribución de reflectancia bidireccional (BRDF, por sus siglas en inglés). Se relaciona con la intensidad de luz reflejada desde  $x''$  hacia  $x$  a través de  $x'$ . En la Sección 2.4 se analizan casos particulares de esta función.
- $S = \cup S_i$ , el dominio de la integral, es la unión de todas las superficies de la escena. Los puntos  $x, x', x''$  varían a lo largo de todas las superficies.

En palabras, la Ecuación 2.1 establece que la intensidad que llega a un punto  $x$  desde otro punto  $x'$ , es la intensidad que  $x'$  emite en dirección a  $x$  más la intensidad reflejada por  $x'$  hacia  $x$  desde cualquier otro punto de la escena ( $x''$ ). Todo sujeto a si hay geometría en el camino de  $x$  a  $x'$ , si es que  $x'$  "ve" a  $x$ .

Los términos "intensidad radiante" e "intensidad emitida" tal y como son planteados no son exactamente ninguna de las unidades radiométricas vistas en la Sección 2.2 pero pueden derivarse de estas.

La Ecuación 2.1 es la base de varios métodos de iluminación global [Ake<sup>+</sup>18]. Tiene en cuenta toda la escena para calcular la luz en un punto. La misma no presenta una solución analítica cerrada para la mayoría de los casos, por lo que

se calculan soluciones numéricas para su resolución. A su vez, considera toda la iluminación de la escena, incluyendo tanto luz directa como indirecta.

En cambio, la **ecuación de reflectancia** (2.2), derivada de la de renderizado, se enfoca en el cálculo de la luz indirecta.

$$L_o(p, \omega_o) = \int_{\Omega} f(p, \omega_i, \omega_o) L_i(p, \omega_i) n \cdot \omega_i d\omega_i \quad (2.2)$$

Calcula la radiancia saliente  $L_o$  de un punto  $p$  y un pequeño ángulo sólido  $\omega_o$  como una integral sobre el hemisferio  $\Omega$  centrado en el punto. Sus términos son muy parecidos a los de la 2.1. Vemos que ya no están ni el término geométrico, dado que se trata de direcciones y no puntos, ni el término de emisión, dado que solo se considera luz indirecta.  $f$  es la BRDF.  $L_i(p, \omega_i)$  es la radiancia entrante hacia  $p$  por otra dirección  $\omega_i$ .  $n$  es la normal en el punto  $p$  y  $n \cdot \omega_i$  es el producto interno entre la normal y la dirección, que equivale al coseno del ángulo entre ellos. La radiancia saliente es la suma por todo el hemisferio visible  $\Omega$  centrado en  $p$ . La suma de todas las radiancias a lo largo del hemisferio es la irradiancia, como se vió anteriormente.

## 2.4. BRDF

**La función de distribución de reflectancia bidireccional** (BRDF, por sus siglas en inglés), es una función  $f(x, \omega, \omega')$  que dado un punto  $x$ , una dirección  $\omega$  entrante y una saliente  $\omega'$ , retorna cuánta luz se refleja desde la dirección entrante hacia la saliente en el punto.

Se puede obtener usando tanto modelos analíticos como midiendo objetos reales con cámaras calibradas y fuentes de luz. A lo largo de los años, se han propuesto varios BRDFs, tanto teóricos como empíricos [MU12]. Los BRDFs son propios de los materiales, y este es uniforme a lo largo de todo el material, por lo que no se diferencia entre dos puntos distintos  $x$  y  $x'$  del mismo material.

Dos modelos teóricos e ideales comúnmente utilizados son los siguientes:

- Reflexión especular:  $f(x, \omega, \omega') = \rho$  cuando  $\omega$  es simétrico a  $\omega'$  respecto a la normal. 0 en otro caso.
- Reflexión difusa:  $f(x, \omega, \omega') = \frac{\rho}{\pi}$  para todas las direcciones donde  $\rho$  es la reflectividad de la superficie, es decir, la fracción de la energía reflejada con respecto a la energía incidente total.

La reflexión especular refleja un rayo solamente en un ángulo específico. La reflexión difusa refleja la luz igual en todas las direcciones, con un mismo valor

de BRDF para cada una de ellas.

Los modelos ideales simplifican mucho los calculos y son fisicamente posibles, aunque no existan materiales reales con estas características de reflectancia.

## 2.5. Trazado de rayos (*ray tracing*)

Uno de los métodos más antiguos y populares para calcular la iluminación global es el de trazado de rayos. Se basa en tratar a la luz como una partícula, y trazar el camino que toman los rayos de luz a través de la escena, calculando reflección, refracción y absorción del rayo cuando interseca con un objeto. La popularidad de este método surge debido a su simplicidad y al gran realismo que agrega a las imágenes generadas. Su primer uso como herramienta de computación gráfica para representar reflexión, refracción y sombras se atribuye a Turner Whitted, [Whi80].

*Ray tracing* lanza rayos desde la cámara a través de la grilla de píxeles hacia la pantalla. Por cada rayo, se busca la intersección con el objeto más próximo. El punto de intersección se prueba si está en sombra lanzando un nuevo rayo hacia todas las luces de la escena y comprobando si intersecan con algún objeto. Pueden surgir otros rayos desde la intersección. Si la superficie es especular, se genera un rayo en la dirección simétrica a la dirección de vista. Si la superficie es transparente, se genera un rayo en la dirección de refracción, gobernada por la ley de Snell<sup>1</sup>. En la Figura 2.6 se puede ver una imagen generada por este método.

El método sufre de *aliasing*, los bordes escalonados o “dentados” presentes en la figura, que ocurre debido a una falta de resolución, cantidad de píxeles, que no logra capturar todo el detalle de la imagen. Suele notarse al representar curvas (ver Figura 2.6b).

Kajiya propuso en 1986[Kaj86], junto con la ecuación de renderizado, un método llamado *path tracing*, una extensión de *ray tracing* que utiliza integración de Monte Carlo para simular la dispersión de la luz. En lugar de únicamente trazar los caminos de reflexión y refracción si corresponden, muestrea aleatoriamente todos los posibles caminos de la luz, esto incluye caminos en los que la luz se dispersa. Es imposible muestrear todos los caminos posibles de la luz, por lo que un parámetro importante en este tipo de algoritmos es la cantidad de muestras tomadas. Debido a la variancia del muestreo aleatorio, el método sufre de ruido, un granulado en la imagen final. El mismo disminuye a medida que se toman más muestras.

---

<sup>1</sup> $n_1 \sin \theta_1 = n_2 \sin \theta_2$

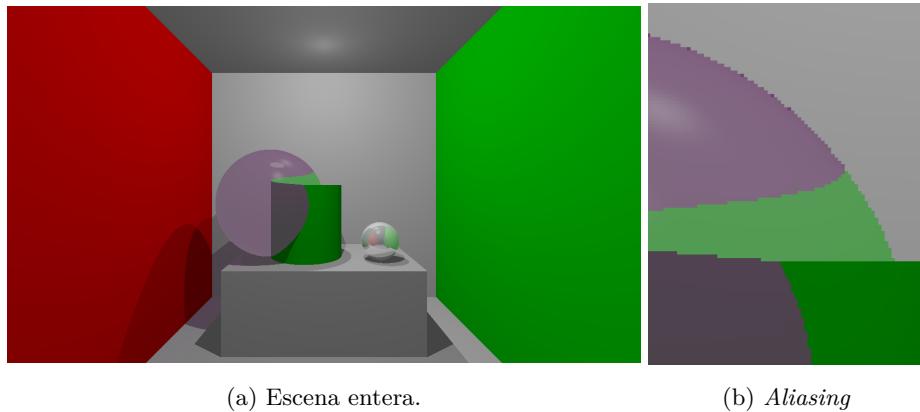


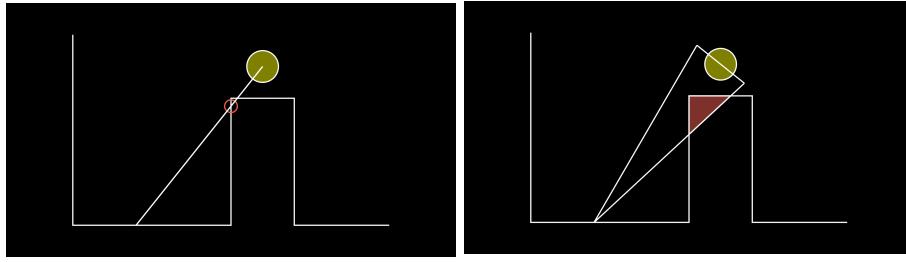
Figura 2.6: Escena renderizada con el trazado de rayos de Whitted. Implementación propia.

Tanto el *aliasing* como el ruido pueden mitigarse con una técnica llamada *supersampling* [Whi80]. Consiste en tomar más de una muestra por píxel, que implica lanzar más de un rayo por píxel, y luego promediar los resultados. Promediar las muestras suaviza las curvas, mitigando el *aliasing*, y provee más muestras aleatorias para la integración de Monte Carlo, lo que reduce el ruido. Es una técnica muy costosa dado que implica lanzar cantidades mucho mayores de rayos. Para imágenes complejas, pueden ser necesarios cientos de rayos por píxel para reducir el ruido a niveles aceptables, imposibilitando la generación de imágenes en tiempo real. Debido a la complejidad computacional agregada por esta técnica, han surgido otras. La estrella de las técnicas de reducción de ruido son los *denoisers* basados en el uso de redes neuronales.

Varios métodos de iluminación global basados en el trazado de rayos han surgido desde entonces.

## 2.6. Trazado de conos

En su artículo de 1984 [Ama84], Amanatides propone una extensión al trazado de rayos en la que redefine los rayos por conos. Los rayos tienen un origen, una dirección y son infinitesimalmente finos. Los conos tienen a su vez un ángulo de apertura, lo que agrega un grosor no despreciable. Usando este grosor, los conos son capaces de no solo probar la existencia de intersecciones, sino también calcular el porcentaje de área de la intersección. El método fue propuesto



(a) Rayo de sombra

(b) Cono de sombra

Figura 2.7: Los conos de sombra no solo devuelven si hay intersección, también devuelven el porcentaje de área de la intersección

para solucionar el *aliasing* presente en el trazado de rayos, como alternativa al *supersampling*. En lugar de lanzar muchos rayos por cada píxel, se lanza un solo cono que integra una mayor área de la escena. Dado que el cono tiene en cuenta la contribución de la luz de varias direcciones dentro de su volumen, reduce el *aliasing* y el ruido al muestrear mayor parte de la escena y reducir la varianza del muestreo.

Trazar conos en lugar de rayos no solo ayuda en el *aliasing* y el ruido, también permite generar sombras suaves. En el trazado de rayos, al probar si un punto se encuentra en sombra o no, se lanza un rayo hacia la fuente de la luz. Si este interseca con algún objeto antes de llegar a la luz, el punto está en sombra. La respuesta binaria, sí o no, a la pregunta de si el punto se encuentra en sombra, resulta en sombras duras. Al trazar un cono en lugar de un rayo hacia la fuente de luz, es posible responder el porcentaje de occlusion de ese punto, lo que produce una escala de grises y sombras suaves. En la Figura 2.7 se puede apreciar esta diferencia.

El trazado de conos, al igual que el trazado de rayos, halla las intersecciones entre cono y objeto de manera analítica. Se toman en cuenta los distintos tipos de objetos (esfera, plano, entre otros) y se resuelve la ecuación de intersección entre ellos.

Para calcular la luz indirecta se halla la radiancia saliente de un punto  $p$  en dirección a la cámara. En *path tracing*, este valor se calcula aproximando la integral sobre el hemisferio con el método de Monte Carlo. Se toman varias muestras, rayos lanzados desde el punto  $p$  en todas las direcciones  $\omega_i$ . En el caso del trazado de conos en lugar de rayos, se partitiona el hemisferio en secciones que pueden ser aproximadas mediante estos conos.

## 2.7. Photon Mapping

*Photon Mapping*, propuesto por Henrik Wann Jensen en 2001 [Jen01], es un algoritmo basado en el trazado de rayos que es capaz de simular de manera más realista la refracción de la luz a través de sustancias transparentes como el vidrio o el agua. Funciona “emitiendo” fotones de la fuente de luz y almacenando en un mapa de fotones la ubicación de cada interacción de estos con las superficies no especulares ni transparentes de la escena.

Con *photon mapping* se pueden generar cáusticas, que son los dibujos que se generan cuando una superficie especular o transparente concentra la luz en una superficie difusa. En la Figura 2.8 se puede ver este efecto.

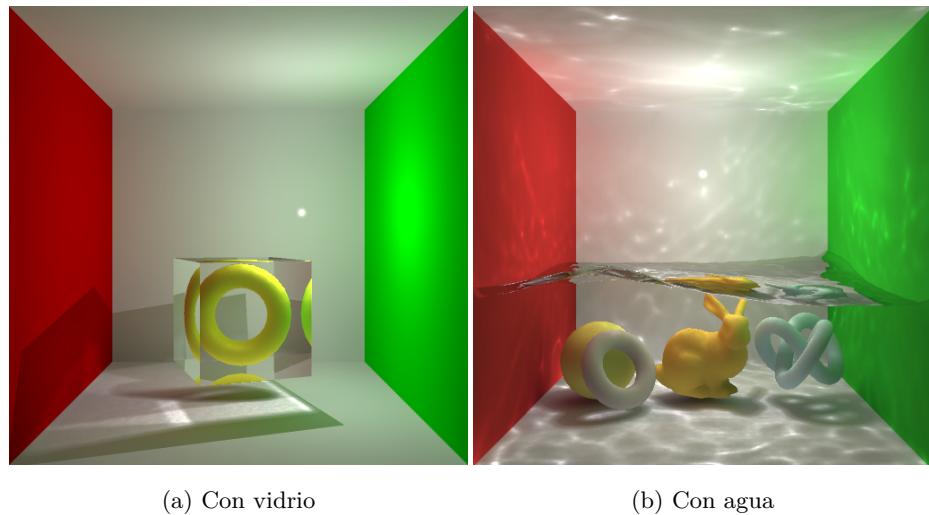


Figura 2.8: Cáusticas. Fuente: [Wan<sup>+</sup>09]

El algoritmo comienza con una etapa en la que se lanzan fotones desde la fuente de luz hacia la escena. Estos se almacenan en las superficies difusas de la escena, creando un **mapa de fotones** y se usan en una segunda etapa cuando se está calculando el color de un píxel. Además de los rayos de ray tracing de reflexión y refracción, se lanzan rayos adicionales en direcciones aleatorias que buscan en el mapa de fotones, simulando la reflexión difusa. Es una extensión a ray tracing, que utiliza el paso adicional de lanzado y evaluación de fotones.

Al igual que con el trazado de rayos, varias mejoras y optimizaciones han surgido a lo largo de los años. Variantes de la técnica original han sido desarrolladas haciendo uso de tarjetas gráficas, alcanzando el tiempo real [SA19]. Imágenes generadas por esta técnica pueden verse en la Figura 2.9.

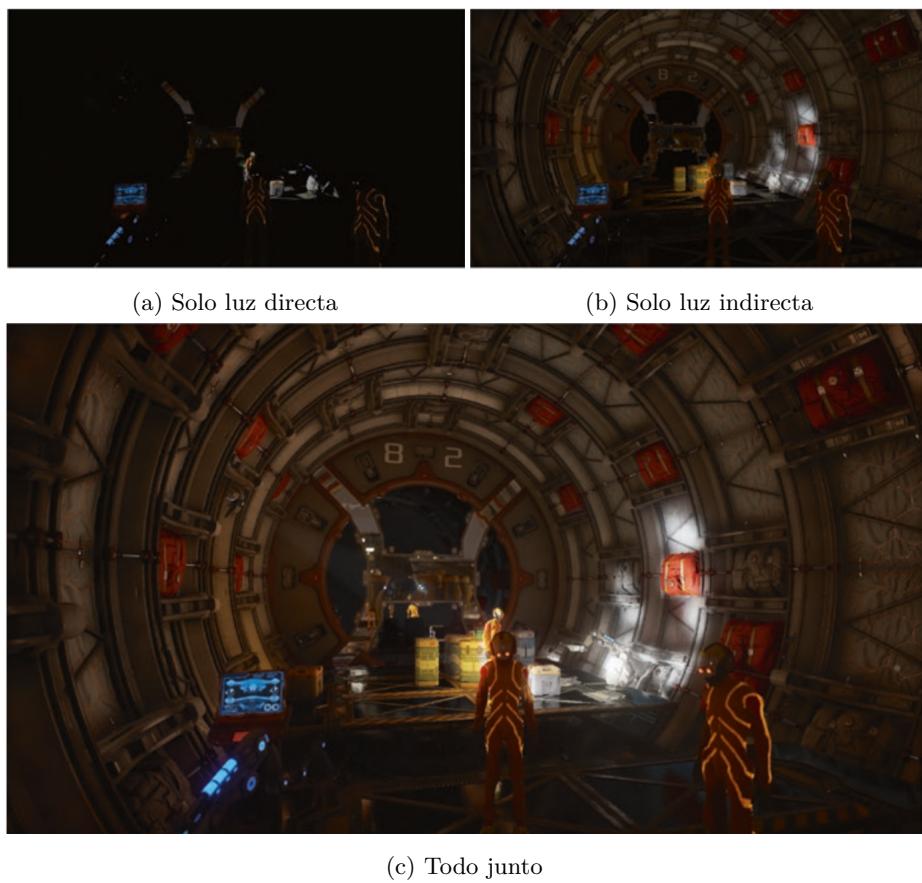
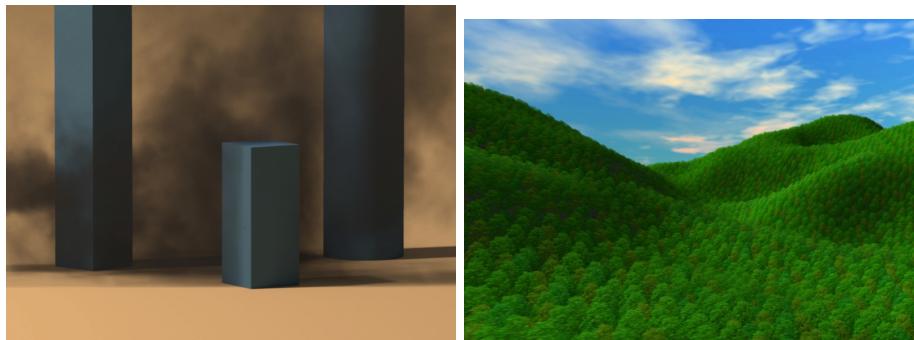


Figura 2.9: Photon mapping en tiempo real. Fuente: [SA19].



(a) Vóxeles representando humo. Fuente: (b) Vóxeles representando vegetación.  
[\[FSJ01\]](#). [\[DN04\]](#).

Figura 2.10: Vóxeles para renderizar humo y vegetación

## 2.8. Vóxeles

Un vóxel es el equivalente de un píxel en el espacio 3D. Así como un píxel es un elemento de imagen o *picture element*, un vóxel es un elemento de volumen, *volume element* [Ake<sup>+</sup>18]. Similar a como los píxeles se ubican en una grilla que divide una superficie 2D en secciones cuadradas, los vóxeles dividen un volumen 3D en cubos.

Tradicionalmente, se usan para guardar datos volumétricos y como primitiva para renderizar una variedad de objetos. Permiten representar volúmenes, en contraposición al triángulo, que se utiliza para representar únicamente superficies. Son un buen candidato para renderizar volúmenes y modelos 3D como el humo, la niebla, el fuego, los huesos y el terreno, entre otros.

Algunos de estos elementos se ven en la Figura 2.10.

Cada vóxel marca si la zona del espacio que representa está ocupada o libre, y por ejemplo en contextos médicos se usan para indicar la opacidad y densidad de un hueso. Para el renderizado, se pueden utilizar para almacenar valores como el color o la irradiancia en cada vóxel. En general, no es necesario guardar la posición de un vóxel, ya que su lugar en la grilla es lo que indica su posición.

El proceso de convertir otra representación, por ejemplo una malla de polígonos, en una estructura de vóxeles se llama voxelización. Involucra interseccar la representación poligonal con la grilla de vóxeles, y marcar los vóxeles que se solapan con esta.

Algunos programas usan grillas completas de vóxeles. Sin embargo, en la

mayoría de los casos no es necesario. Una observación útil es que en muchas aplicaciones es suficiente tener véxoles en el límite entre un objeto y el espacio vacío, siendo innecesario el relleno.

Otra observación útil es que, para obtener una buena representación, alcanza con tener mucho detalle en la frontera entre el espacio vacío y el lleno. Para lograr esto, se puede usar un *octree* disperso, que será explicado a continuación.

## 2.9. Octrees

Un *octree* es una estructura de datos de “árbol”, en la que cada nodo interno (no hoja) tiene 8 hijos [Ake<sup>+</sup>18]. Suele usarse para representar datos espaciales [BD02]. Se puede utilizar para dividir el espacio 3D de manera jerárquica, con varios niveles.

Se parte de un volumen original cúbico o paralelepípedo que se divide en dos partes iguales por dimensión, resultando en 8 octantes iguales. En la estructura de árbol, el nodo raíz representa el volumen original y cada uno de sus 8 hijos representa a cada octante. Aplicando recursivamente, se divide el espacio en  $8^{(n-1)}$  secciones, donde  $n$  es la cantidad de niveles del árbol, y el primer nivel tiene un solo nodo que representa toda la escena.

De manera similar, un espacio 2D se puede dividir utilizando un *quadtree*, en donde cada nodo interno tiene 4 hijos. Los *quadtrees*, al ser bidimensionales, son más fáciles de visualizar, por lo que serán utilizados a lo largo de este informe para explicar aspectos que funcionan igual tanto en ellos como en su equivalente tridimensional. En la Figura 2.11 se muestra un *quadtree* en su forma de árbol y en el espacio que subdivide.

Cuando la naturaleza de la información lo permite, se puede evitar subdividir un nodo del árbol si sus 8 hijos tienen todos la misma información. De esta idea surge el *octree* disperso.

A la hora de voxelizar una escena, los véxoles pueden ubicarse dentro de un *octree* disperso, en lugar de en una grilla. En este caso, los nodos no se subdividen si no hay geometría dentro de la región del espacio que representan, dado que no hay véxoles en esa región.

## 2.10. Ducto gráfico

Dada una escena 3D, una cámara virtual, varios objetos y varias fuentes de luz, ¿cómo se genera una imagen 2D en el monitor de una computadora?

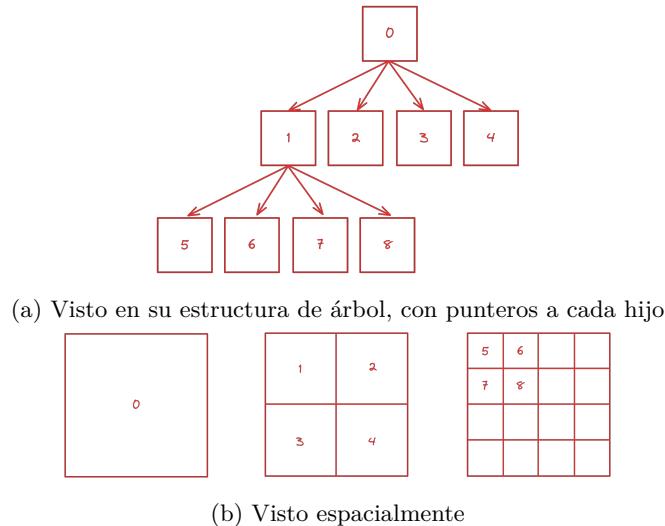


Figura 2.11: Quadtree

Si bien es posible generar una imagen a partir de una escena usando la CPU, las GPUs, o tarjetas gráficas, están especialmente diseñadas para esto.

Las tarjetas gráficas ejecutan un **ducto gráfico** para generar las imágenes, esto es, una secuencia de transformaciones y operaciones que parten de primitivas y generan la imagen final. Existen varios tipos de ductos gráficos, el ducto de rasterización, el de cómputo de propósito general, el de trazado de rayos, entre otros.

El ducto de rasterización, Figura 2.12, es importante, dado que es utilizado en *voxel cone tracing*. Este ducto parte de vértices de mallas poligonales, aplica transformaciones, realiza pruebas de profundidad y calcula el color de cada píxel de la imagen final.

Cada paso de un ducto gráfico puede ser fijo o programable. En el caso de que sea fijo, el mismo ya está implementado en la tarjeta gráfica. En algunos casos, estos pasos fijos proveen parámetros para configurar su comportamiento, este es su mayor grado de libertad. En el caso de que sea programable, se puede escribir un *shader*, un programa de sombreado que corre en la GPU, que implementa el paso en su totalidad, lo que aporta una gran libertad a la hora de renderizar escenas.

Existen varias APIs, interfaces, con las que se puede interactuar con una tarjeta gráfica [Gal21], notablemente Vulkan, Direct3D, Metal, WebGPU y OpenGL. Todas estas permiten acceder al ducto de rasterización.

En OpenGL, por ejemplo, este ducto posee las etapas que se ven en la Figura 2.12 [Ake<sup>+</sup>18].

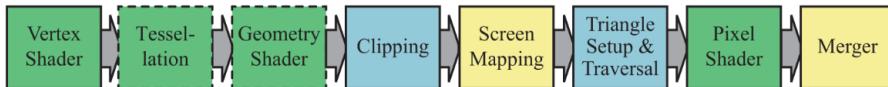


Figura 2.12: Ducto de rasterización. Fuente: [Ake<sup>+</sup>18]

El desarrollador debe escribir *shaders* para las etapas programables, que se pueden escribir en el lenguaje GLSL [Khr23]. Los *shaders* son programas que ejecutan en la GPU con un alto grado de paralelización en la tarjeta gráfica.

Las etapas programables son el *vertex shader* (*shader* de vértices), *geometry shader* (geometría), y *fragment* o *pixel shader* (fragmentos o píxeles). Cada una de estas etapas pueden comunicar datos a etapas posteriores. El resto de las etapas son fijas.

El *vertex shader* toma los vértices de la geometría de la escena y los puede transformar a otro sistema de coordenadas. En general es usado para pasar los vértices de espacio local de coordenadas a espacio global, de vista y luego proyección, lo que se logra usando tres matrices que se conocen como matriz de modelo, matriz de vista y matriz de proyección. Un hilo es ejecutado por cada vértice de las primitivas de entrada.

El *geometry shader* puede generar nuevos vértices, por lo que es útil para agregar complejidad extra a la geometría de la escena. Aquí se ejecuta un hilo por cada primitiva de salida del *vertex shader*, pero estas primitivas pueden tener más de un vértice.

Finalmente, el *fragment shader* (o *pixel shader*) trabaja con píxeles y no con vértices. Es en este *shader* donde se realizan los cálculos de iluminación para calcular el color de cada píxel. Se ejecuta como mínimo un hilo por cada píxel no vacío de la imagen que se quiere generar. Pueden ejecutarse más de uno en el caso que dos objetos aporten color al mismo píxel, en cuyo caso puede ser que uno sea descartado o pueden mezclarse los colores de ambos en caso de que presenten una opacidad menor a 1.

Otro ducto muy relevante, que es usado para implementar la mayoría del algoritmo, es el ducto de cómputo de propósito general. Consiste en la inicialización de datos de entrada, luego la ejecución de un programa en la GPU llamado *compute shader* (*shader* de cómputo) y finalmente el retorno de datos hacia la CPU. Notar que este ducto no es utilizado para generar una imagen, sino para realizar cálculos arbitrarios que toman una entrada y producen una

salida, aprovechando la alta paralelización que provee la tarjeta gráfica.

## 2.11. Renderizado diferido

A la hora de renderizar una escena, hay dos grandes modelos que se pueden utilizar: el clásico, conocido como *forward rendering* y una opción alternativa conocida como renderizado diferido.

En el primero, se procesan todos los vértices en el programa. Cada uno pasa por cada etapa del ducto gráfico y aporta a la imagen final a menos que sea desecharlo por una prueba de profundidad. Cuando un píxel genera varios fragmentos, y se deben correr calculos pesados por fragmento, se puede estar realizando trabajo innecesario, debido a que, en el espacio 3D de la escena, pueden estar a distinta profundidad y sólo el más cercano es el que se debe procesar. Dados dos fragmentos que podrían darle color a un mismo píxel, una prueba de profundidad descartará al que se encuentre detrás del otro, a menos que presente opacidad menor a 1.

En el renderizado diferido, se procesan todos los vértices del programa pero se saltean calculos pesados en el *fragment shader*. Se guarda toda la información necesaria para etapas posteriores en texturas llamadas *geometry buffers*, datos como posicion, normal, color y cualquier otro dato relevante de la geometria a la hora se realizar calculos pesados de sombreado. Se saltean los cálculos para objetos fuera del campo de visión y detrás de otros objetos.

El *forward rendering* es conceptualmente más sencillo y más fácil de implementar. En escenas con pocos objetos y fuentes de luz, o con calculos simples de iluminacion, la complejidad extra del renderizado diferido no suele estar justificada dado que el aumento de rendimiento es despreciable. A su vez, el *forward rendering* soporta mejor la opacidad. Sin embargo, a medida que la complejidad de la escena y los metodos de sombreado aumenta, el renderizado diferido se vuelve cada vez una mejor opción para aumentar el rendimiento. También, el renderizado diferido facilita ciertas técnicas de post-procesamiento de la imagen, como bloom, HDR (High Dynamic Range), corrección gamma, y normalización, entre otras.

## Capítulo 3

# Voxel cone tracing

En este capítulo se detalla el diseño de *voxel cone tracing* [Cra<sup>+</sup>11], un algoritmo de iluminación global en tiempo real, que utiliza conceptos presentes en el trazado de rayos (2.5), trazado de conos (2.6) y *photon mapping* (2.7), pero reduciendo los costos asociados a estos algoritmos clásicos usando una representación de la escena con vóxeles (2.8), almacenada en un *octree* disperso (2.9), para aproximar los conos.

Otra forma de reducir costos y aumentar la velocidad del algoritmo es utilizando el ducto de rasterización 2.10 y renderizado diferido 2.11. Los cálculos de luz se realizan en el *fragment shader* del ducto de raster, sobre los *geometry buffers* para reducir la cantidad de fragmentos o hilos de la GPU sobre los que se corren calculos pesados, sin perder calidad de imagen. Lo anteriormente explicado difiere del trazado de rayos clásico, que renderiza toda la escena puramente mediante intersecciones entre rayos y geometría.

La principal aproximación que realiza este algoritmo es trabajar sobre una representación voxelizada pre-filtrada de la escena. Dadas las coordenadas de un punto de la escena, se recorre el árbol para hallar el vóxel que representa la región del espacio que incluye ese punto. Este vóxel contiene toda la información necesaria sobre oclusión, color e irradiancia debido a que la estructura es previamente filtrada. En el filtrado, la información en las hojas del árbol es promediada hacia niveles mayores hasta que cada nivel del árbol contiene información que aproxima las capas inferiores. La información de un vóxel en cada nivel resume la información de los vóxeles de los nodos hijos que comparten el mismo espacio en la escena. Se vió en la sección sobre trazado de conos (2.5), que en su planteo original, se hallaban las intersecciones del cono con los objetos de la escena de manera analítica, lo que es costoso. Aquí, en lugar de hallar la

intersección analíticamente, se aproxima el cono utilizando la estructura jerárquica de véxeles, permitiendo acumular la información a lo largo del cono mas eficientemente, pero con cierta perdida de detalle.

El algoritmo se puede dividir en 4 grandes etapas:

1. Voxelización de la escena
2. Construcción y filtrado del *octree* disperso
3. Inyección de fotones
4. Trazado de conos

La voxelización, la inyección de fotones y el trazado de conos usan el ducto de rasterización, mientras que la construcción del árbol y el filtrado usan el ducto de cómputo de propósito general. En el resto del capítulo se explicará con mayor grado de detalle cada una de estas etapas.

### 3.1. Voxelización

La escena se divide en una grilla de véxeles. La cantidad de véxeles es configurable, siendo usualmente 512 o 1024 por dimensión, totalizando  $512^3$  o  $1024^3$  véxeles respectivamente. Un ejemplo de voxelización se muestra en la figura 3.1, con 64, 128 y 256 véxeles por dimensión.

Para voxelizar la escena, se realiza el procedimiento detallado en “OpenGL Insights, capítulo 22” [CG12], un aporte escrito por Crassin luego de haber publicado el artículo de *voxel cone tracing*, aprovechando mejoras en las herramientas disponibles en el momento. Ese mismo proceso será explicado a continuación, para más información, consultar la referencia.

Usando el ducto de rasterización de OpenGL, se procesan todos los triángulos de la geometría de la escena utilizando como resolución para el rasterizado la resolución de la grilla de véxeles. Esto genera una lista de **véxeles**, donde cada véxel es una posición dentro de la grilla y un color, que es el color de un triángulo que interseca con el espacio representado por el véxel. Cada uno de estos véxeles será usado para construir el árbol y terminará almacenado en la estructura. Dado lo anteriormente descrito, el mismo véxel, identificado por su posición en la grilla, puede estar presente en la lista multiples veces con colores diferentes si mas de un triángulo interseca con el véxel.

La voxelización de un triángulo  $B$  a un véxel  $V$  puede hacerse si:

1. El plano de  $B$  interseca  $V$ .

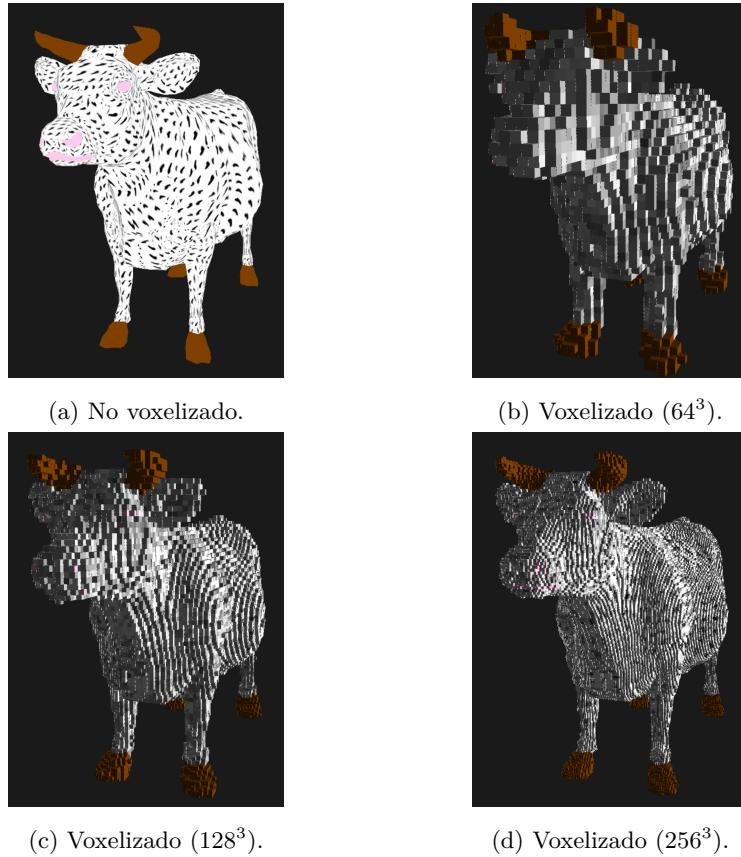


Figura 3.1: Ejemplo de voxelización con la cantidad de véxeles por dimensión entre paréntesis.

2. La proyección 2D del triángulo  $B$  por la dimensión dominante de su normal (la que provee la mayor área proyectada) interseca la proyección 2D de  $V$ .

Basado en esta observación, se sigue la serie de pasos que se muestra en la figura 3.2.

Primero, cada triángulo de la geometría se proyecta ortográficamente en la dimensión dominante de su normal. La dimensión dominante se elige dinámicamente por triángulo en el *geometry shader*, donde la información de los tres vértices de cada triángulo está disponible, maximizando el área del triángulo proyectado.

Cada triángulo proyectado se rasteriza para conseguir fragmentos correspondientes a la resolución 2D de la grilla de véxeles. Se fija el tamaño del *viewport*

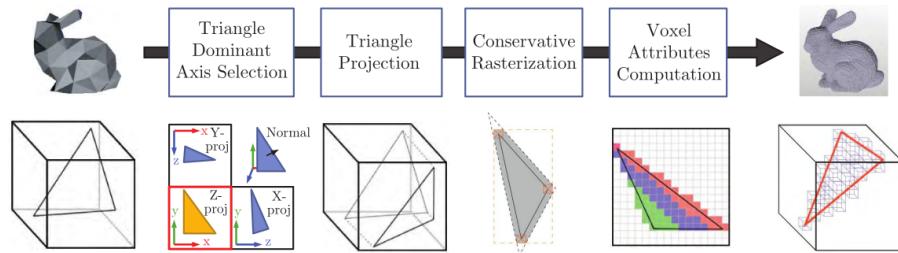


Figura 3.2: Ducto de voxelización. Fuente: [CG12].

a coincidir con la cantidad de véxeles, por ejemplo un *viewport* de tamaño  $512 \times 512$  para una grilla de  $512^3$  véxeles.

Durante la rasterización, cada triángulo genera un conjunto de fragmentos 2D. Estos fragmentos generarian solo un véxel en la grilla si esta fuese 2D, sin embargo al ser una grilla 3D, un triángulo cuyo plano no es perpendicular a su dirección dominante puede intersecar con mas de un véxel en la dirección de proyección(profundidad).

Es necesario calcular la intersección del plano del triángulo con los véxeles para encontrar los valores de profundidad en la grilla, ya que el triángulo fue proyectado perdiendo sus valores de profundidad. Debido a la elección de la dimensión dominante de la normal para la proyección del triángulo, cada fragmento puede generar de 1 a 3 véxeles con mismos valores de  $x$  e  $y$  pero diferentes valores de  $z$ , fuese  $z$  la dirección dominante. Entonces, por cada fragmento 2D, los véxeles que intersecan con el triángulo se calculan en el *fragment shader*, basándose en la posición, la profundidad y las derivadas de la profundidad en espacio de pantalla. Se necesitan las derivadas para calcular la orientación del plano del triángulo, ya que los fragmentos no tiene información global del triángulo que los genero, solo datos del punto responsable de la generación del mismo.

Luego de realizada la voxelización, se obtiene la lista de véxeles necesaria para crear el árbol, con su posición y color. Cada uno tiene sus coordenadas en  $\mathbb{N}^3$  que lo identifica dentro de la grilla 3D de la escena, así como color. Su posición en el espacio de la escena se pude calcular a partir de su posición en la grilla.

### 3.1.1. Rasterización conservativa

El método descrito anteriormente a veces no crea véxeles para elementos muy finos, como un asta de bandera, ya que en el rasterizado solo se prueba

el centro del píxel contra los triángulos para generar fragmentos. Se necesita una manera de generar fragmentos para cada píxel tocado por un triángulo, no necesariamente en el centro. Un algoritmo así se detalla en [HAO05].

La idea es generar, por cada triángulo, un polígono acotante ligeramente más grande, para asegurarse que cualquier triángulo proyectado que interseca con cualquier punto del píxel también interseque con su centro. Se logra trasladando las aristas del triángulo hacia afuera, la mitad de la diagonal de un píxel, y extendiéndolas para generar un triángulo semejante. A su vez se generan fragmentos nuevos que resultan de sobreestimar la cobertura de este triángulo, dado que este nuevo triángulo puede intersecar con el centro de píxeles que antes no intersecaban en ningún punto. Para evitar estos fragmentos también se genera una caja acotante alineada con los ejes, y se descartan fragmentos por fuera de la misma. Este proceso se muestra en la figura 3.3.

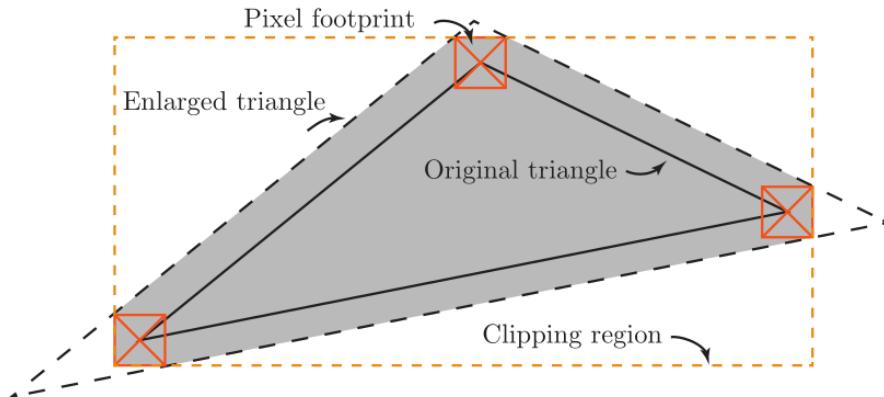


Figura 3.3: Rasterización conservativa. Fuente: [CG12].

## 3.2. Octree disperso

Para almacenar los vértices generados, se usa un *octree* disperso, como los vistos en la sección 2.9. Un octree denso subdivide la escena en 8 cubos y cada cubo se subdivide en 8, y así sucesivamente. Esta estructura escala rápidamente, generando problemas importantes de volumen y velocidad de acceso necesarios a memoria. Para aliviar este problema se crearon los *octrees* dispersos, donde los nodos no se subdividen si no poseen geometría dentro.

Cada elemento del árbol es un **nodo**. Un nodo del árbol representa una

sección de la escena. Cada nivel tiene una cierta cantidad de nodos. En un árbol denso, cada nodo no hoja tiene exactamente 8 hijos, por lo que cada nivel  $n$  tendría  $8^n$  nodos. El nivel 0 tiene  $2^{(0 \times 3)} = 1$  nodo, el nivel 1 tiene  $2^{(1 \times 3)} = 8$ , el 2 tiene  $2^{(2 \times 3)} = 64$  y así sucesivamente. El último nivel del árbol es el que llega a la resolución deseada de 512 o 1024 véxeles. Valores más altos de resolución crean más niveles del *octree* y hacen que la aproximación de véxeles se asemeje cada vez más a la malla de triángulos original.

En la figura 3.4, se muestra un modelo al que luego de ser voxelizado, se creó un *octree* disperso. La resolución utilizada fue de 128 véxeles por dimensión, por lo que el último nivel del *octree* es el 7. En la figura se muestran los niveles 5, 6 y 7.

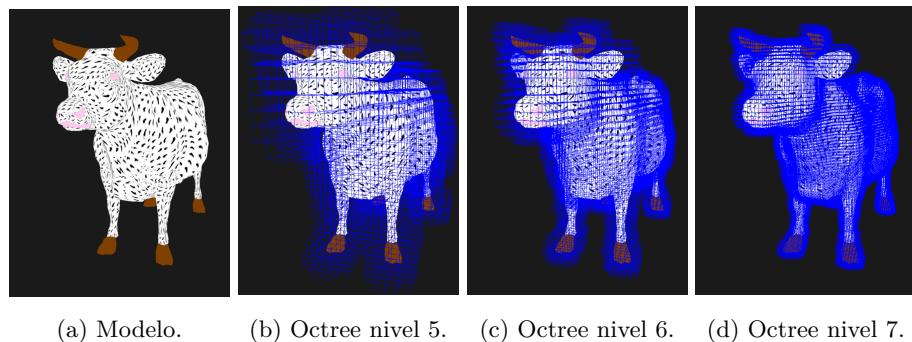


Figura 3.4: Distintos niveles de un octree disperso.

Dado que la estructura tiene como máximo 512 o 1024 véxeles de resolución, sin importar la geometría de la escena, los cálculos sobre ella son independientes de la complejidad de la geometría.

### 3.2.1. Nodos, bricks y véxeles

Los nodos del árbol no almacenan los véxeles mencionados en 3.1. Cada nodo almacena únicamente un puntero a sus, como máximo, 8 hijos.

Cada nodo tiene asociado un **brick**, otra estructura que también representa una región del espacio. Cada brick está dividido en 27 véxeles, distribuidos en una estructura de  $3 \times 3 \times 3$ . Son estos véxeles los que almacenan los valores de la escena. Los nodos solo contienen punteros a sus hijos, existen para obtener la estructura del árbol. Los nodos proveen la estructura mientras que los bricks y véxeles los datos.

En la figura 3.5 se puede observar un nodo con su brick asociado de la

manera en la que se disponen en el espacio. Los bricks ocupan más espacio que sus nodos, ya que los véxeles se centran en los vértices del nodo. Esto es necesario para que los bricks informen con bricks de nodos vecinos, lo que garantiza que la interpolación dentro de un solo nodo tome en cuenta los valores de sus vecinos. Colocar los véxeles en las esquinas de los nodos genera una frontera compartida entre bricks de nodos vecinos, como se puede ver en la figura 3.6. La frontera entre dos nodos adyacentes del mismo nivel son los 9 véxeles de la cara compartida entre los nodos, 3 en el caso 2D apreciable en las figuras, que están presentes en los bricks de ambos nodos. Como consecuencia, la estructura tiene información repetida, necesario para que funcione la interpolación cuando se quiera generar la imagen final. Un *shader* llamado *border\_transfer*, es el que se encarga de lograr la coherencia en la frontera entre dos *bricks*. Se explicará en la Sección 3.2.3.

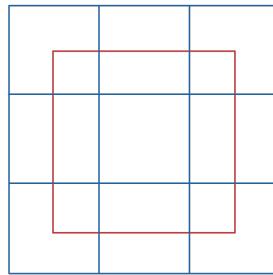


Figura 3.5: Nodo (en rojo) con su brick asociado (azul).

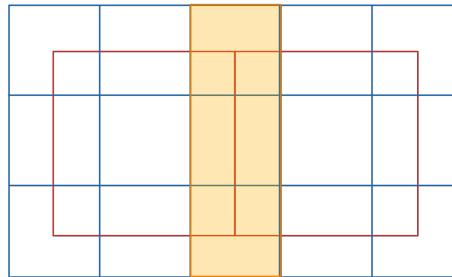


Figura 3.6: Solapamiento entre véxeles de bricks de nodos vecinos.

### 3.2.2. Construcción

Para generar el octree disperso se usa la lista de véxeles generada durante la voxelización. Se empieza con un árbol con un solo nivel, con un solo nodo que

representa toda la escena. Empezando en el primer nivel del árbol y descendiendo a lo largo de los niveles, se subdividen los nodos que contienen geometría. Un nodo debe ser subdividido cuando existe un vértice en la lista con coordenadas dentro del nodo y no se llegó a un cierto nivel deseado. El algoritmo utilizado se detalla a continuación.

Dado un nivel  $i$  del árbol, dos programas principales son ejecutados en secuencia para generar el nivel  $i + 1$ : *flag\_nodes* y *allocate\_nodes*.

Se corre un hilo de *flag\_nodes* por cada vértice de la lista de vértices. Dado un vértice, se recorre el árbol, hasta llegar a  $i$ , el último nivel creado hasta el momento. Para recorrer el árbol, cada hilo comienza en su primer nodo. Al llegar a un nodo en el nivel  $i$ , se marca uno de los 8 punteros a sus hijos. Cada hijo del nodo representa un octante en el espacio dentro de él. Se usa la posición del vértice para determinar a qué octante pertenece, el puntero a este hijo es marcado.

Realizar esta marca es un proceso idempotente, ya que se limita a cambiar el bit más significativo del puntero a 1. Es idempotente porque el resultado de la operación es el mismo sin importar la cantidad de hilos que lo realicen.

Luego, se ejecuta *allocate\_nodes*, que busca en el nivel  $i$  los punteros marcados. Se corre un hilo por cada posible nodo hijo que puede ser creado en esta etapa, 8 por cada nodo del nivel  $i$ . A cada hilo se le asigna uno de los punteros de los nodos del nivel, confirma si fue marcado en el paso anterior, elimina la marca, crea el nodo hijo y agrega al puntero el índice del hijo recién creado. El índice del nuevo hijo se crea usando un contador que es posteriormente incrementado. Para el incremento, se usa una operación atómica que permite que dos hilos puedan crear un nodo concurrentemente sin colisión de índices. Siempre y cuando haya geometría en la región de la escena representada por un nodo, el mismo será subdividido nivel tras nivel hasta el máximo determinado por la resolución de la grilla de vértices.

Esta separación entre marcar y subdividir es necesaria para prevenir problemas de concurrencia, al ser esto ejecutado en múltiples hilos. Durante la construcción de un nivel interno del *octree*, es probable que muchos hilos marquen la creación de un mismo nodo hijo. Si el mismo hilo que se encarga de crear la marca, también crease el nodo, se tendrían nodos duplicados perdidos dentro de la memoria. En lugar de lidiar con que todos esos hilos intenten subdividirlo, se limitan a marcarlo de manera idempotente, evitando así utilizar locks entre hilos. Correr un solo hilo para cada posible hijo que se debe crear asegura que no se creen nodos duplicados, resolviendo así el problema de concurrencia.

Un paso de este algoritmo se puede observar en la figura 3.7. Corriendo el

algoritmo para un vóxel  $x$ , este debe ser ubicado en el último nivel, en el nodo punteado. El nodo inmediatamente anterior se marca para subdividir y luego se subdivide.

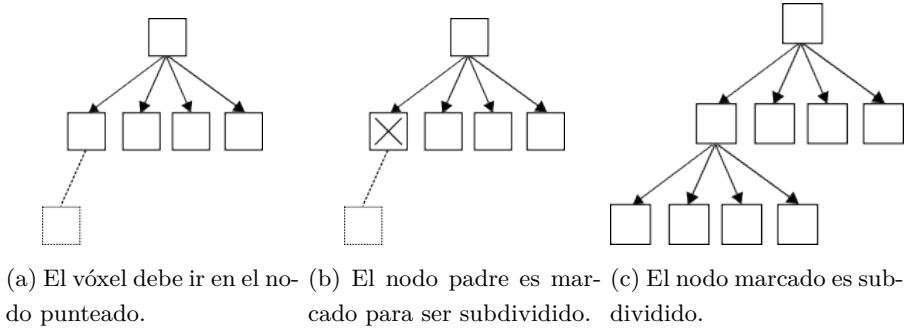


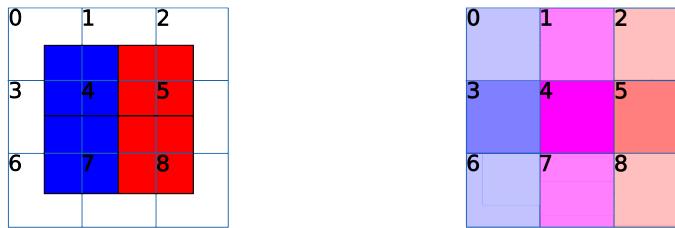
Figura 3.7: Un paso del algoritmo de *flag* y *allocate*.

Una vez alcanzado el último nivel, se escriben los atributos de los vóxeles en los bricks de las hojas del árbol, promediando cuando más de un vóxel tiene la misma posición en la grilla de vóxeles. Mientras más triángulos tenga la escena original y menos resolución la grilla de vóxeles, más ocurre lo anteriormente mencionado.

En 3.2.1, se vió como los *bricks* ocupan una región más amplia del espacio que su nodo correspondiente. Además los vóxeles conseguidos en la voxelización no están alineados con los vóxeles de los bricks. En la primer parte de la figura 3.8 se puede observar un brick, cuyos vóxeles están numerados, solapado con 4 vóxeles generados en el paso de voxelización, cada uno con su color. En el espacio de la escena estos coinciden con el nodo del brick, ya que la grilla utilizada para la voxelización es la misma que la determinada por los nodos del nivel mas bajo, pero llegando a un nivel mayor de resolución. Por lo que a cada nodo le corresponden a lo sumo 8 vóxeles del paso de voxelización, si consideramos que triángulos que se solapan en un vóxel de la grilla devuelven un único vóxel con el promedio de sus colores.

Para la versión 2D a cada nodo le corresponden a lo sumo 4 vóxeles del paso de voxelización. Se le asignan valores a todos los vóxeles del brick, utilizando la figura 3.8 como ejemplo, como se detalla a continuación:

- El vóxel 4 almacena el promedio de azul y rojo
- El vóxel del medio de cada arista almacena el promedio de los 2 valores que le corresponden y espacio vacío, siendo el vóxel 3 un azul semi transparente con un alfa de 0.5



(a) Solapamiento en el espacio de la escena (b) Brick con el valor de los vóxeles cargados entre el brick y los vóxeles

Figura 3.8: Vóxeles de la voxelización y su mapeo a un brick

- El vóxel de cada esquina almacena tres cuartas partes de espacio vacío, y el color correspondiente, siendo el vóxel 2 de color rojo con un alfa de 0.25

Como resultado, se espacien los valores de los vóxeles del paso de voxelización a los vóxeles del *brick*. De esta figura también se desprende la necesidad del border transfer 3.2.3, ya que los espacios vacíos de los vóxeles del brick deben ser rellenados con información de nodos vecinos, aunque de forma transitória afectan el alfa del vóxel. Lo anteriormente descrito es el algoritmo que expande los vóxeles para poder llenar las  $3 \times 3 \times 3$  subdivisiones del *brick*. A partir de aquí, el término “vóxel” refiere a una de las 27 subdivisiones de un *brick*, dado que ya se procesó la lista generada durante la voxelización.

### 3.2.3. Border transfer

Como se mostro en la figura 3.6, los vóxeles frontera entre *bricks* adyacentes corresponden al mismo espacio en la escena. Sin embargo, cada nodo tiene su propio *brick*, pero esos *bricks* no comparten memoria entre ellos, habiendo así vóxeles que pertenecen a *bricks* diferentes pero corresponden al mismo espacio en la escena. Es necesario asegurarse que los valores almacenados en esos vóxeles frontera tengan el mismo valor para asegurar coherencia y una correcta interpolación a la hora de generar la imagen final. Sin embargo, luego de aplicar *spread\_leaves*, los vóxeles frontera pueden tener valores distintos entre *bricks* vecinos. Para igualarlos, se promedian los valores de la frontera con la del *brick* vecino. Además este paso es necesario para llenar los espacios vacíos vistos en la parte anterior.

El shader *border\_transfer* se encarga de promediar los valores en la frontera de cada *brick* con la de sus vecinos, en los ejes X, Y y Z. De esta manera, aún cuando un vóxel puede estar en varios *bricks*, en 8 como máximo para los vóxeles

en las esquinas de los *bricks*, su valor va a ser siempre el mismo en cada uno de ellos. En la figura 3.9 se puede apreciar este proceso, realizado para el eje X, entre dos *bricks* vecinos con los véxels que comparten enmarcados.

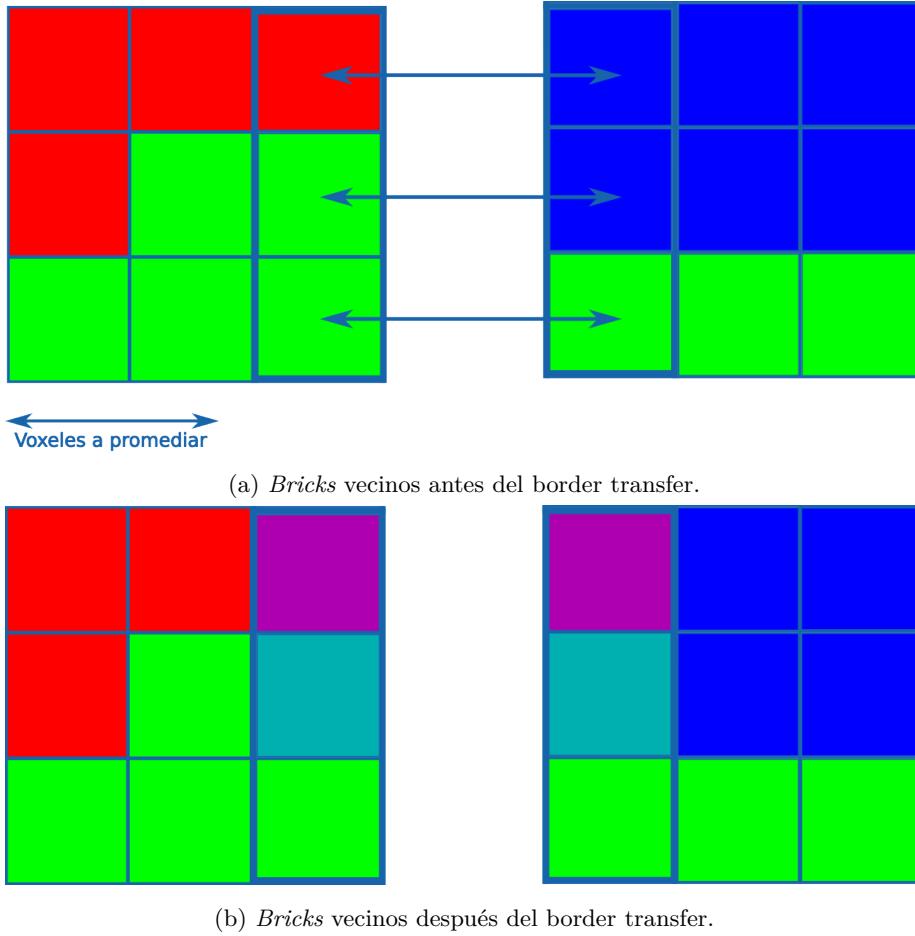


Figura 3.9: *Bricks* vecinos en el *eje X*.

### 3.2.4. Nodos frontera

El paso anterior tiene un problema, asume que todos los nodos tienen vecinos. Al usar un *octree* disperso, no existen nodos en donde no hay geometría. Como se vio en la sección 3.2.3, para mantener coherencia dentro de la estructura todo par de *bricks* vecinos deben tener valores iguales en sus véxels frontera. En el límite entre la geometría y el espacio vacío existen nodos sin vecinos. Estos

nodos presentan un problema que impide una correcta interpolación, dado que no se puede promediar el valor de sus véxeles con su vecino. Para una correcta interpolación, es necesaria una capa de nodos extra, los que llamaremos **nodos frontera**, que representa el espacio vacío adyacente a la geometría.

Los nodos frontera se añaden en cada nivel del árbol a la hora de construirlo. Sus *bricks* en principio no contienen valores, existen para asegurar una correcta interpolación en los extremos de la geometría. Se realiza un border transfer extra entre los nodos frontera y sus vecinos, de manera que los véxeles compartidos entre los dos tengan el mismo valor. El mismo problema no se vuelve a generar entre estos nodos frontera y el espacio vacío, dado que los véxeles compartidos entre ellos tienen opacidad nula. El resultado de la introducción de los nodos frontera se observa en la figura 3.14.

En el cone tracing, se toma la intersección del cono con el nodo. Luego se toma una muestra del *brick*, en el lugar donde interseca con el cono. Para mejor entender el problema que es solucionado por los nodos frontera, veamos un ejemplo y en vez de un cono usemos un rayo ya que eso se acerca mejor a lo que hace el algoritmo.

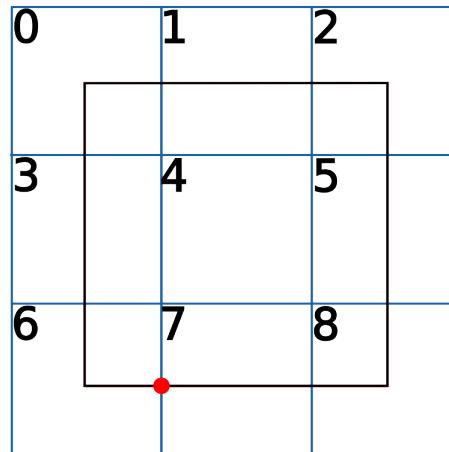


Figura 3.10: Muestreo interpolado.

Consideremos un nodo como el de la figura 3.10, cuyo *brick* asociado tiene sus véxeles numerados. Este nodo no tiene vecinos. Digamos que su véxel 6 es de color azul y su véxel 7 color rojo, ambos con alfa 1. Si se tiene un rayo que interseca en algún punto entre ambos véxeles, como se puede ver representado con un punto rojo en la figura 3.10, el valor muestrado sera violeta con un mayor componente azul, la interpolación lineal entre ambos.

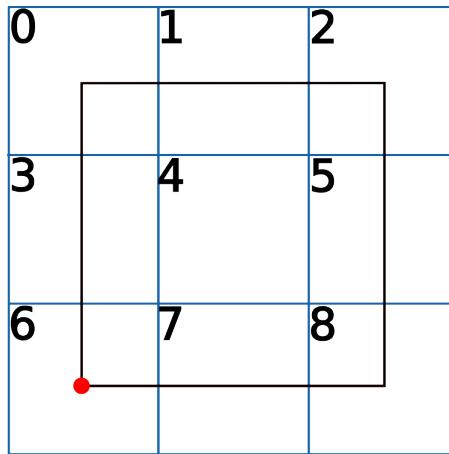


Figura 3.11: Muestreo del centro de un véxel.

Si un rayo interseca con el centro del véxel como en la figura 3.11, el color muestreado sera azul con alfa 1.

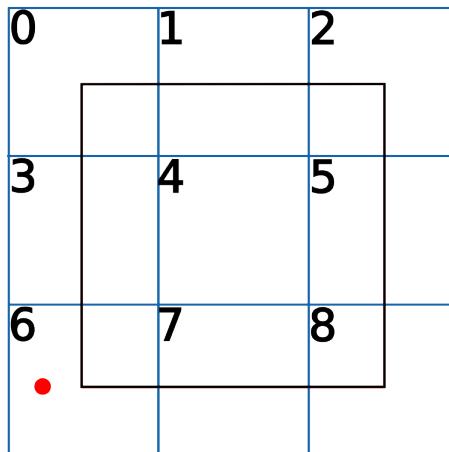


Figura 3.12: Muestreo con resultado vacío.

¿Pero que ocurre cuando el rayo interseca como se ve en la figura 3.12? Siguiendo la misma lógica, debería ser una interpolación entre azul y vacío, el cual sería azul con un alfa menor a 1. Sin embargo, el rayo solo puede intersecar con nodos, no *bricks*, por lo que la intersección es vacía. Pero eso es inconsistente, dado que ese véxel tiene un valor, aunque no haya nodo ahí. Si el nodo tuviese un vecino en X, el rayo interseccionaría con el vecino, evitando el problema.

Y esa es la solución a nuestro problema, agregar uno nodo en este espacio sin geometría.

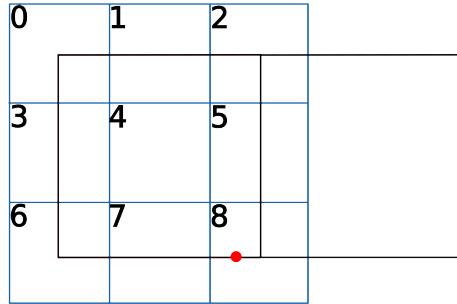


Figura 3.13: Muestreo con nodo frontera.

En la figura 3.13 se tiene el nodo frontera y su brick con sus véxeles numerados, al lado del nodo original que no tenia vecinos y el mismo rayo de la figura 3.12. El véxel 8 va a tener color azul y alfa 1, dado que los véxeles compartidos entre vecinos siempre deben tener el mismo valor, y la intersección del rayo con la estructura ya no va a ser vacía llevando al resultado deseado del muestreo.

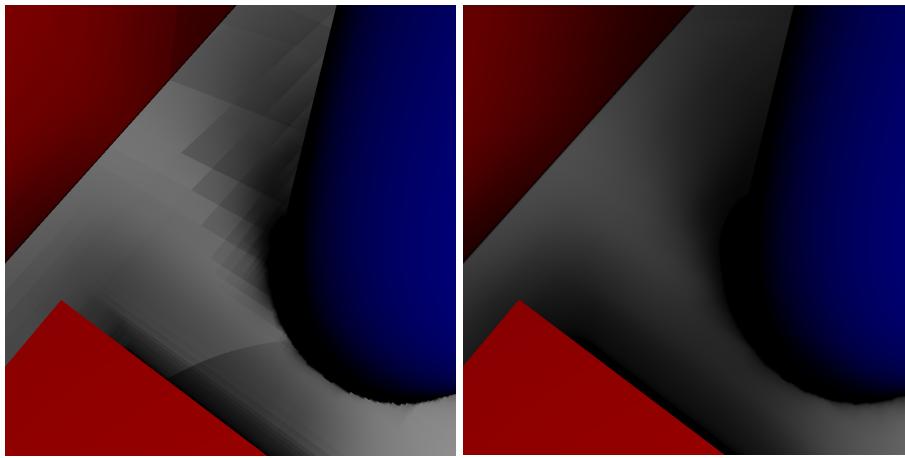
### 3.2.5. Filtrado

Una vez que todos los atributos se encuentran en las hojas del octree, los mismos seran filtrados a posiciones superiores. Filtrarlos implica promediarlos de tal manera que para un nodo interior (no hoja)  $A$ , su brick tenga un promedio de la información contenida en los bricks de todos sus hijos. El proceso se realiza en  $n - 1$  pasos, siendo  $n$  el nivel máximo del octree. En cada paso, se calcula el valor de cada véxel del brick del padre, usando los bricks de los hijos.

Consideremos un nodo en el penúltimo nivel, con su brick asociado y sus hijos, como muestra la figura 3.15. En la figura se muestran solo 4 hijos porque se usa como ejemplo un *quadtree*, la versión 2D del *octree*, ya que es mas facil de visualizar. Cada hijo tiene a su vez su propio *brick* asociado como se muestra en la figura 3.16.

Los valores de los véxeles del brick padre se calculan en 4 etapas distintas, dependiendo de dónde se ubican en el *brick*: Esquinas, bordes, caras, centro. Cada una de las etapas calcula un valor parcial para un tipo de véxel. Es parcial porque para los véxeles limítrofes con otro nodo, este valor tiene que luego ser agregado con el de los vecinos, con un tipo de *border\_transfer*.

Dado el véxel superior izquierdo de la figura 3.18, se considera solo el brick



(a) Sin nodos frontera. (b) Con nodos frontera.

Figura 3.14: Diferencia causada por nodos frontera.

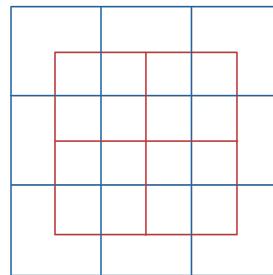


Figura 3.15: Nodo con su *brick* asociado y sus hijos.

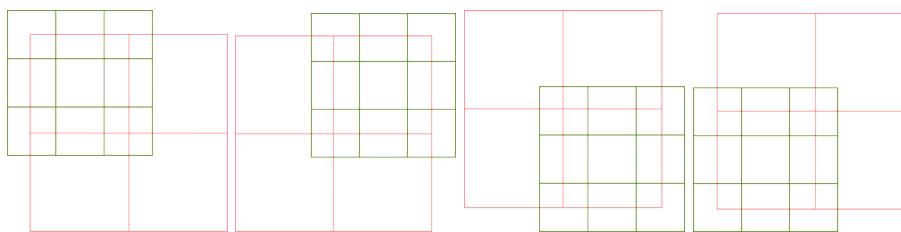


Figura 3.16: Bricks asociados a cada hijo del nodo.

del hijo superior izquierdo del nodo. De ese brick, se consideran los véxeles amarillos en la figura. El valor final del véoxel del padre se calcula promediando los valores de los véxeles del hijo, pesados por el porcentaje de solapamiento.

a	b	c
d	e	f
g	h	i

Figura 3.17: Una posible forma de referirse a cada voxel de un brick.

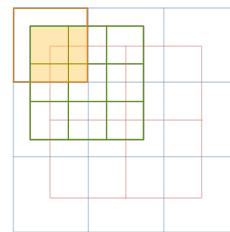


Figura 3.18: Filtrado para un voxel esquina. Se puede ver el voxel del brick padre cuyo valor se quiere calcular, junto con el brick del hijo y su solapamiento con este voxel.

Si nombramos los voxels del brick hijo  $a, \dots, i$  y los del brick padre  $a', \dots, i'$ , como en la figura 3.17, entonces el valor del voxel del padre se calcula como:

$$a' = a + b * \frac{1}{2} + d * \frac{1}{2} + e * \frac{1}{4}$$

En el caso tridimensional, hay que agregar un quinto factor multiplicado por  $\frac{1}{8}$ .

De la misma manera se calculan los voxels de los bordes y de las caras, solo que en esos casos se usan más de un brick hijo, como se puede ver en las figuras 3.19 y 3.20.

### 3.2.6. Filtrado anisotrópico

El filtrado descrito en la sección anterior resulta en voxels con valores independientes del punto de vista, o isotrópicos. Una característica deseable es que los valores de oclusión y opacidad dependan del punto de vista del observador, lo que se logra en este caso guardando mas de un valor por voxel para estas propiedades. Es muy útil a la hora de representar un atributo en una escena 3D.

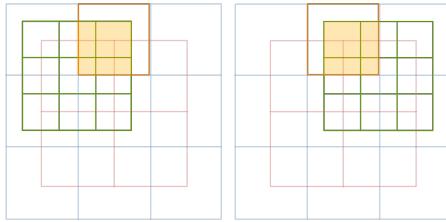


Figura 3.19: Filtrado para un véoxel borde.

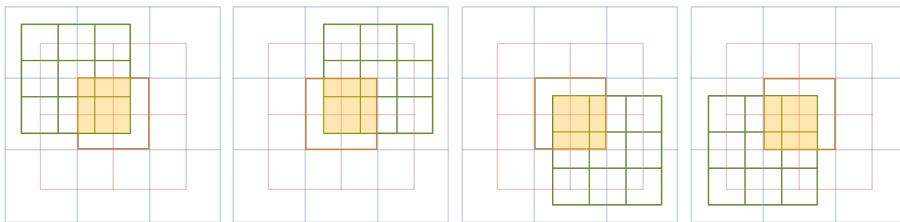


Figura 3.20: Filtrado para un véoxel cara.

Para ilustrar el punto anterior, consideremos una escena compuesta únicamente por una pared fina. Dado un véoxel de un nodo en un nivel alto del árbol, el nodo representa una gran región del espacio. La región contiene únicamente una pared que pasa por su centro, y el resto de la escena es espacio vacío. Con el filtrado presentado en las secciones anteriores, el véoxel tendrá un único valor de opacidad que, al representar un espacio mayormente vacío, tendrá un valor bajo. Sin embargo, se observa que la percepción de la pared es muy distinta si se la ve de frente o de costado. Como se muestra en la figura 3.21, la pared vista de frente es opaca, pero vista de costado su superficie visible es prácticamente nula, asemejándose a una superficie transparente.

La solución propuesta por Crassin [Cra+11] consiste en realizar el filtrado 6 veces por véoxel, uno por cada dirección alineada con los ejes: X, -X, Y, -Y, Z, -Z; y tener en cuenta la dirección a la hora de promediar los valores. Cada uno de los 6 valores representa el véoxel visto desde una de las 6 direcciones previamente mencionadas. Por lo tanto si se observa el véoxel desde una dirección arbitraria, se calcula su oclusión y color a través de la interpolación de las tres direcciones más cercanas. Este tipo de filtrado se denomina **anisotrópico**, dado que depende de la dirección con que se observe al véoxel.

Dada una dirección, por ejemplo, de izquierda a derecha, se parte de los véoxeles de la izquierda y se calcula un valor para cada fila, partiendo de estos

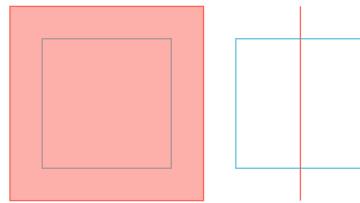


Figura 3.21: Vóxel que contiene una pared fina. En la izquierda, la pared se ve de frente. En la derecha, se ve de costado. Se espera que el valor del vóxel refleje esta diferencia entre las direcciones de vista.

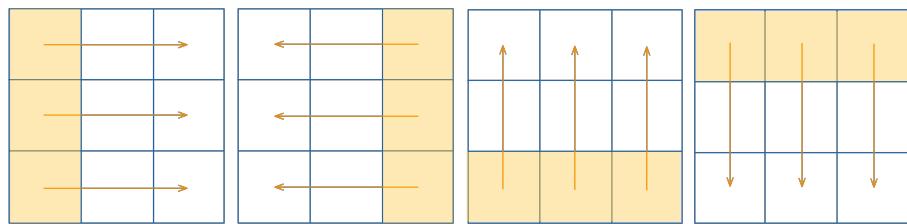


Figura 3.22: Filtrado anisotrópico en todas las direcciones.

yiendo hacia los véxeles de la derecha. Llamémosle al valor de cada fila **valor direccional**. En la figura 3.22 se pueden ver todos los valores direccionales que deben ser calculados para un brick en 2D, para todas las direcciones. Para cada fila, se ejecuta un algoritmo de acumulación de opacidad que va avanzando en la dirección dada. Si el algoritmo llega a opacidad 1, termina y devuelve el valor direccional para esa fila.

El filtrado anisotrópico soluciona situaciones como la de la pared mencionada anteriormente. Para el caso de la pared fina, el vóxel que contiene la pared es opaco en la dirección paralela a la normal de la pared y prácticamente transparente en cualquiera de las perpendiculares.

### 3.3. Inyección de fotones

Hasta ahora tenemos la estructura de datos creada, conteniendo el color y opacidad de toda la escena en las hojas y un promedio de los niveles inferiores en todos los nodos interiores. El objetivo del algoritmo es la iluminación global de una escena, por lo que necesitamos información de la luz. En este paso, lanzamos

fotones a partir de la fuente de luz de la escena, similar a como se hace en *photon mapping*, pero utilizando el ducto de rasterización.

Se rasteriza la escena desde el punto de vista de la luz para generar una textura 2D. En lugar de tener colores en cada téxel de la textura, se almacenan las posiciones de los objetos de la escena. La existencia de una posición en un texel de la textura significa que esa posición es visible desde la luz, por lo que debe recibir un fotón. Las posiciones son utilizadas para recorrer el octree y almacenar los fotones en vóxeles de sus hojas.

Los fotones que se almacenan en los vóxeles del árbol, la **irradiancia**, es el flujo lumínico recibido por cada superficie. Luego pasan por el mismo proceso de *border\_transfer* y filtrado que el color. El *border\_transfer* suma en lugar de promediar en este caso, dado que ambos lados de la frontera aportan a la cantidad de fotones total. El filtrado funciona de la misma manera.

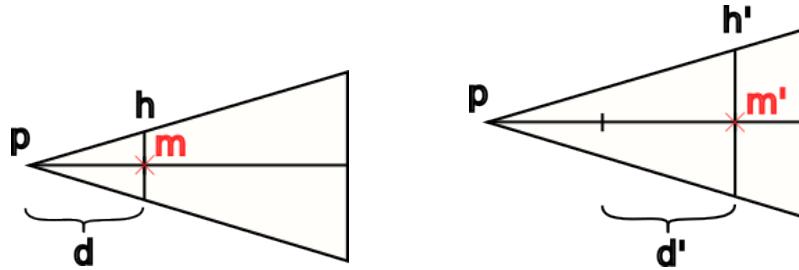
La etapa de construcción debe realizarse solo una vez, mientras que, al soportar luces dinámicas, esta etapa debe ejecutarse cada vez que la luz se mueva. En esos casos, toda la irradiancia del árbol vuelve a cero y se vuelve a correr el programa que lanza los fotones, y lo llamaremos la actualización de la estructura.

### 3.4. Cone tracing

Como se comentó brevemente al inicio del capítulo, el trazado de conos se realiza en el *fragment shader* del ducto de rasterización. La entrada al algoritmo de *cone tracing* son los *geometry buffers* que contienen los valores de la escena, ya habiendo descartado los vértices fuera de vista. El algoritmo de trazado de conos es ejecutado para cada píxel de los *geometry buffers* para calcular el color final.

Dado un punto de origen  $p$ , se lanza uno o varios conos con cierta dirección y apertura, dependiendo del efecto que se quiera lograr. Para cada cono, se parte desde su origen y se avanza en la dirección de su eje tomando pasos de cierto tamaño variable. Esto se conoce como *ray marching*.

Se toma un paso de tamaño  $d$  a lo largo del eje del cono y se consigue un nuevo punto  $p + d$ . Se calcula el diámetro del cono en ese punto,  $h$ . Luego se encuentra el nivel del octree  $i$  con el menor tamaño posible de vóxel que sea mayor al diámetro del cono  $h$ . Dado ese nivel  $i$  y la posición del punto  $p + d$ , se recorre el octree y se encuentra el nodo correspondiente a ese nivel que incluye el punto. Ese nodo tiene un brick asociado, cuyos vóxeles tienen los valores prefiltrados, conseguidos en 3.2.5. Se usa el valor del vóxel que contiene al punto  $p + d$ , este

(a) Toma de una primer muestra  $m$ . (b) Toma de una segunda muestra  $m'$ .Figura 3.23: Toma de muestras a lo largo del cono en *cone tracing*.

valor es una muestra  $m$ .

Este proceso se vuelve a repetir. Se toma un nuevo paso, con un largo potencialmente diferente,  $d'$  y se llega a un nuevo punto  $p + d + d'$ . Se halla el diámetro del cono en ese punto,  $h'$ . Se halla el nivel del octree  $i'$ , el brick, el voxel que contiene el punto y su valor es una segunda muestra  $m'$ .

Estas muestras se acumulan a lo largo del cono. En caso de voxels anisotrópicos, también depende de la dirección del eje del cono. Se sigue avanzando paso a paso acumulando valores hasta satisfacer un criterio de parada, que puede ser llegar a un máximo valor acumulado, o haber recorrido una distancia máxima a lo largo del eje del cono.

Este algoritmo se puede observar en la figura 3.23.

Para calcular luz indirecta, las muestras son un color  $c$  y una opacidad  $\alpha$ . Si en cada paso consideramos  $c$  y  $\alpha$  como los valores hasta el momento, y  $c'$  y  $\alpha'$  como los valores de la nueva muestra  $m'$ , entonces en cada paso los valores de  $c$  y  $\alpha$  se calculan de la siguiente manera ([Cra<sup>+</sup>11]):

$$\begin{cases} c = \alpha c + (1 - \alpha)\alpha' c' \\ \alpha = \alpha + (1 - \alpha)\alpha' \end{cases}$$

Para la luz indirecta difusa, se lanzan conos para cubrir el hemisferio centrado en la normal del punto, como se puede ver en la figura 3.24. En la mayoría de los casos, 5 conos anchos difusos dan un buen resultado. Cada cono acumula el color de los voxels con los que se encuentra multiplicado por la cantidad de fotones, que fue almacenada en el paso de inyección de fotones (3.3). Esto logra un efecto de *color bleeding*, donde las superficies adquieren color de otras superficies cercanas que reciben y dispersan luz.

Para la luz indirecta especular, se lanza un solo cono fino en la dirección de

reflexión. El cono, al ser más fino, es rápidamente ocluido por nodos de niveles más bajos, con lo que el reflejo tiene mejor definición. Si se utiliza un mayor ángulo de apertura del cono, el reflejo se ve más turbio, simulando una superficie menos pulida.

### 3.5. Oclusión ambiental

La oclusión ambiental es una técnica de rendering que se usa para calcular qué tan expuesto está cada punto de una escena a la luz ambiental. *Cone tracing* se puede usar para calcular este valor. El efecto no aporta más al realismo de una escena una vez que se usan técnicas de iluminación indirecta, pero es un buen paso previo para ver el funcionamiento del algoritmo.

Para calcularlo, se lanzan varios conos, cubriendo el hemisferio centrado en la normal de la superficie en el punto. El único valor necesario es la opacidad. A medida que se viaja a lo largo de un cono, se va acumulando la opacidad de los vértices correspondientes. Se define una distancia máxima y el criterio de parada es cuando el punto a lo largo del cono pasa esa distancia máxima, o la oclusión llega a 1.

### 3.6. Conos de sombra

De la misma manera que el trazado de rayos logra sombras lanzando un rayo hacia la fuente de luz, *cone tracing* lo logra lanzando un cono hacia la fuente de luz. El cono toma en cuenta únicamente la opacidad y su criterio de parada es alcanzar la luz o 1 de opacidad antes. El beneficio de que sea un cono en lugar de un rayo, y de tener la estructura jerárquica del *octree*, es que se logran sombras suaves, sin necesidad de lanzar varios y promediar sus resultados.

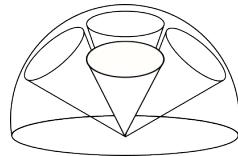


Figura 3.24: Conos intentando cubrir el hemisferio en dirección de  $n$ , centrado en  $p$ .



## Capítulo 4

# Implementación

La primera decisión importante en el proceso de desarrollo fue implementar el algoritmo en GPU. La principal razón para esto es el alto grado de paralelismo que la misma ofrece. A su vez, el ducto de rasterización, ampliamente implementado y optimizado en las GPUs, puede ser utilizado, aprovechando así algoritmos y cálculos optimizados a la hora de generar una imagen.

La programación en GPU precisa un lenguaje para la **CPU**, una **API de gráficos** y un lenguaje de **sombreado**. El lenguaje de CPU es el que realiza y coordina las llamadas a funcionalidades de la GPU. La API de gráficos es la que provee la interfaz que utiliza el lenguaje de CPU para comunicarse con la GPU. La misma es una especificación que los fabricantes de las tarjetas de video deben implementar si deciden soportarla. El lenguaje de sombreado es el que se utiliza para escribir los *shaders*, programas que ejecutan directo en la GPU y hacen uso del paralelismo de la misma.

Como lenguaje de CPU se eligió **Rust**, un lenguaje de sistemas moderno multiparadigma. Fue escogido debido a su gran comunidad, su amplia documentación y a su poderoso manejador de paquetes *cargo*. Se caracteriza por ser eficiente, al permitir un acceso a primitivas de bajo nivel, pero manteniendo una buena experiencia de desarrollo, debido a las abstracciones de costo cero que provee. Otra característica a mencionar es la seguridad de memoria que impide comportamientos indefinidos en tiempo de ejecución como punteros nulos en situaciones que se espera un valor, o accesos a espacios de memoria no inicializados o liberados.

También se consideró el lenguaje C++ para CPU, dado que es el lenguaje más popular para aplicaciones gráficas. Si bien ambos lenguajes presentan una eficiencia similar, el factor que más pesó en la decisión fue la presencia de un

manejador de paquetes en Rust. El mismo permitió instalar varias dependencias más rápido y cambiarlas a lo largo del ciclo de desarrollo cuando fue necesario. Dada la implementación en la GPU, la eficiencia y el manejo de memoria fue secundario.

Como API de gráficos se eligió **OpenGL**, junto con el lenguaje de sombreado **GLSL**, el más usado para OpenGL, aunque soporte otros lenguajes. También fueron consideradas CUDA y Vulkan, pero se terminó por escoger OpenGL debido a la facilidad de desarrollo, el conocimiento previo del equipo y la facilidad de trabajo con *compute shaders*. Si bien Vulkan es una API más moderna, es conocida por ser más compleja que OpenGL y el equipo del trabajo no contaba con la suficiente experiencia para utilizarlo adecuadamente. Su uso hubiera llevado a mayores tiempos de desarrollo y no necesariamente a una implementación más eficiente. CUDA suele usarse para cómputo de propósito general en la GPU, sin embargo, no posee ducto de rasterización que fue ampliamente utilizado. Por lo tanto, se optó por utilizar los *compute shaders* de OpenGL para el cómputo de propósito general necesario.

El desarrolló se realizó en el sistema operativo **Linux** debido a los ambientes de desarrollo utilizados. De igual manera, tanto Rust como OpenGL son multiplataforma, con lo que la aplicación es fácilmente portable a otros sistemas operativos.

La arquitectura de la aplicación consta de tres paquetes: *CLI*, *Core* y *Engine*. *Engine* contiene todas las abstracciones sobre OpenGL utilizadas, provee tipos como *Transform*, *Light*, *Camera*, comunes en aplicaciones interactivas, que permiten manipular objetos en el espacio 3D. *Core* contiene todas las etapas y programas del algoritmo que han sido mencionados: voxelización, construcción del *octree*, filtrado, actualización y el trazado de conos en si. El paquete *CLI* (*Command-Line Interface*, interfaz por línea de comandos) es el punto de entrada de la aplicación, procesa los argumentos pasados por linea de comandos y archivos de configuración y utiliza las funcionalidades expuestas por *engine* y *core* para crear un ambiente 3D y ejecutar el algoritmo. La arquitectura se muestra en la figura 4.1.

En las siguientes secciones se verán más a detalle cada uno de estos paquetes.

## 4.1. *Engine*

*Engine* es el paquete en el que se implementaron las abstracciones usuales al trabajar con APIs de gráficos, que son utilizadas por el resto de la aplicación. Se pueden ver con sus relaciones en la Figura 4.2.

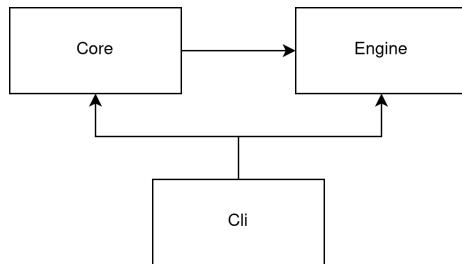
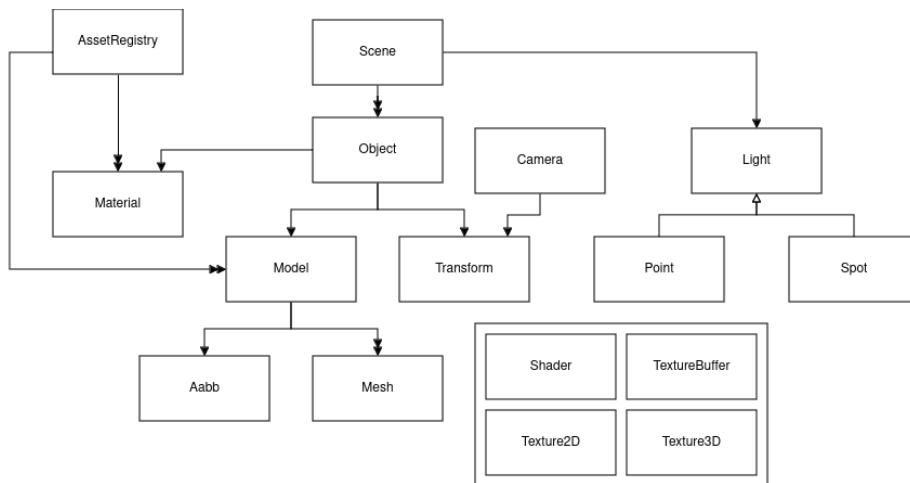


Figura 4.1: Arquitectura alto nivel de la aplicación.

Figura 4.2: Arquitectura de *Engine*.

La abstracción principal es la escena (*Scene*). Una escena consta de una luz (*Light*) y de muchos objetos (*Object*). Es la representación del mundo tridimensional en el que se encuentra todo lo visible de la aplicación.

Cada objeto de la escena tiene un material (*Material*), un modelo (*Model*) y un *Transform*.

*Transform* es un tipo que contiene la posición, rotación y escala de un objeto en la escena. Esto simplifica el lidiar con matrices de traslación, rotación y escala a lo largo de todo el programa.

Tanto los modelos como los materiales son cargados una vez al inicio de la aplicación en el *AssetRegistry* para no duplicarlos en caso de que sean utilizados por más de un objeto. Los objetos solo necesitan almacenar referencias a estos.

Se representan los modelos como mallas poligonales, distinto al trazado de rayos en el que se usan ecuaciones geométricas. Para cargar estos modelos, se

utilizó la librería *tobj* [Ush22], que permite cargar archivos con extensión *obj*. Cada *Model* contiene muchos *Mesh* y tiene un *Aabb*, un Axis-Aligned Bounding Box, que se calcula a partir de cada uno de los vértices del mismo. Este volumen acotante se calcula en el momento que se carga el modelo y se expone para ser utilizado por el resto de la aplicación.

*Camera* provee el punto de vista, la dirección de vista, la dirección hacia arriba, la ventana de vista, entre otros, del que se renderiza toda la escena. Se soportan tanto cámaras en perspectiva como ortogonales.

Para las luces, se soportan tanto luces direccionales como puntuales. Estas luces tienen toda la lógica necesaria para que luego *core* realice la inyección de fotones.

La estructura de árbol, los bricks, y todas las estructuras auxiliares, se almacenan en la memoria de la GPU. Para esto, se ofrecen abstracciones sobre los distintos tipos de memorias y texturas de la GPU. *TextureBuffer*, *Texture2D* y *Texture3D* son ejemplos de estas, que manejan memoria lineal, bidimensional y tridimensional respectivamente. Se provee la abstracción *Shader* para interactuar con la GPU mediante programas escritos en GLSL.

Finalmente, se exponen las primitivas necesarias para crear una interfaz de usuario. Luego, estas son usadas por *core* para modificar parámetros del algoritmo.

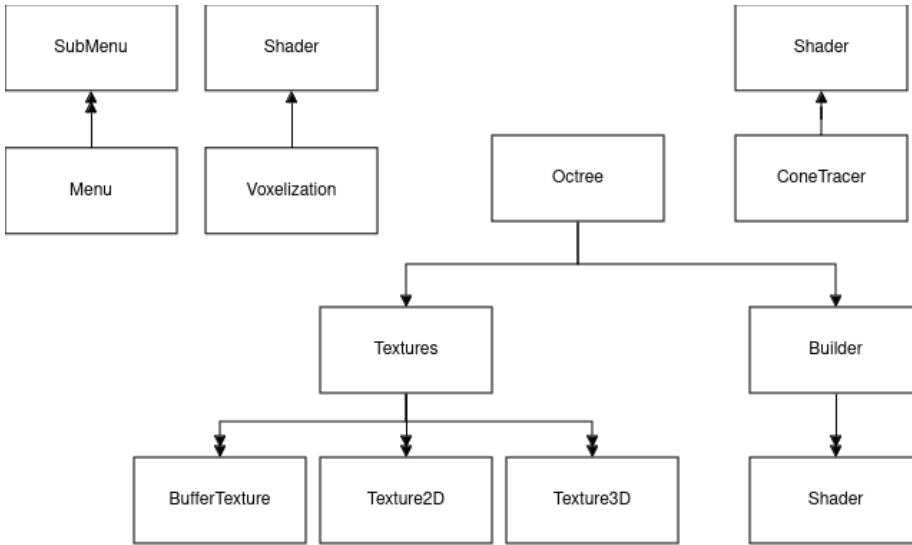
## 4.2. Core

*Core* es el corazón de la implementación, donde se implementa el algoritmo. Contiene la lógica de la voxelización, toda la implementación del *octree*, el trazo de conos y un menú para poder ajustar parámetros en tiempo de ejecución. Debido a que el algoritmo está implementado en la GPU, este paquete hace mucho uso de los *Shaders*. Sus componentes se pueden ver en la Figura 4.3.

### 4.2.1. Etapas del algoritmo

En la Figura 4.3 se pueden ver los componentes encargados de cada etapa del algoritmo.

La voxelización tiene su propio módulo que hace uso del ducto de rastreización como se vió en 3.1. Para voxelizar la escena, esta primero debe ser normalizada usando el *Aabb* expuesto por *Engine*. Se crea un *Aabb* para la escena, y se le aplican transformaciones de escalado y translación de manera que sea lo más grande posible pero quede incluida dentro de la grilla de véxeles.

Figura 4.3: Arquitectura de *Core*.

Las mismas transformaciones luego se aplican a los modelos al momento de voxelizar, asegurando así que todo objeto quede dentro de la grilla, y que se mantengan las proporciones.

La abstracción de *Octree* realiza tanto construcción como el filtrado de los valores de la escena a lo largo de él. Guarda referencias a todas las texturas utilizadas y a todos los *Shaders* que operan sobre esas texturas. También guarda una referencia a la lista de véxeles generada por el módulo anterior.

El *ConeTracer* reúne todos los conceptos anteriores e implementa el algoritmo de trazado de conos en el ducto de rasterizado, donde se genera la imagen final.

#### 4.2.2. Representación del *octree*

La estructura del *octree*, cada nodo y sus hijos, se representa en la GPU con una textura lineal. Esta textura se llama *node pool*. Cada téxel (elemento de textura) de esta textura es un puntero a otro nodo. Se toma la convención de que cada grupo de 8 téxeles es un nodo, donde cada téxel es un puntero al hijo correspondiente. Los primeros 4 téxeles representan los hijos con valor de  $z = 0$  mientras que los últimos 4 representan los que tienen valor de  $z = 1$ . De manera similar de cada grupo de 4, los primeros 2 son los hijos con valor de  $y = 0$  menor y los otros dos  $y = 1$ . Por ultimo de cada uno de los 4 grupos de 2, el primer

téxel representa al hijo con coordenada  $x = 0$  y el último con coordenada  $x = 1$ . Si un téxel tiene el valor 0, entonces ese hijo del nodo representa espacio vacío, y ese nodo no será guardado en la *node pool*. Si un téxel tiene un valor  $n \neq 0$ , entonces en la posición de memoria  $n * 8$  comienza ese nodo, que termina en la posición de memoria  $x * 8 + 7$ .

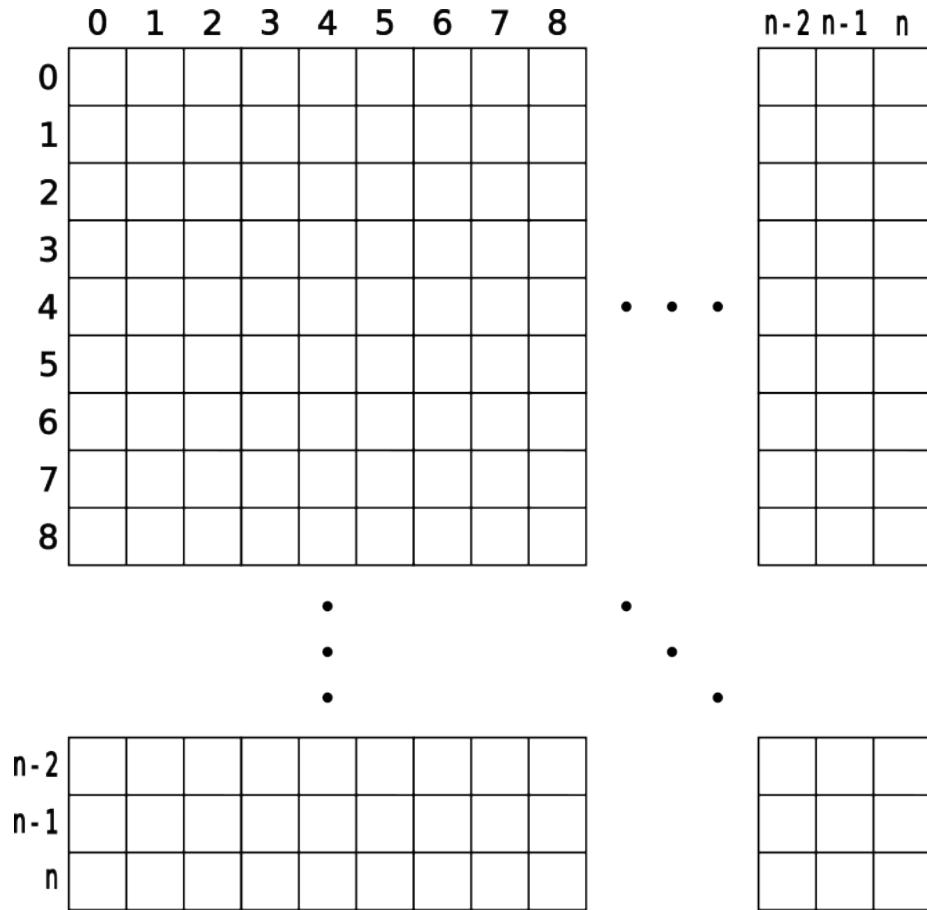


Figura 4.4: Ejemplo de *brick pool* (en 2D), con índices de memoria. El verdadero es 3D.

Para representar los *bricks*, se usa una gran textura 3D, llamada la *brick pool*. Cada brick es una sección de  $3^3$  téxeles de esta textura 3D. En cada uno de estos téxeles se almacenan valores de la escena, por ejemplo color. Para otros valores que deben ser almacenados, como la irradiancia, se crea otra *brick pool*. Cada téxel dentro de un brick se llama *vóxel*, porque además de ser un píxel

de textura, también es un píxel de volumen, a los efectos de la aplicación. En la figura 4.4 se muestra una *brick pool*, con una textura 2D para facilitar la visualización, donde cada téxel esta identificado por sus coordenadas  $x$  e  $y$  (y  $z$  en su versión 3D).

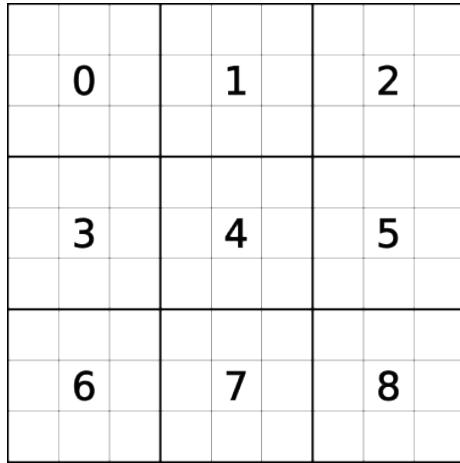


Figura 4.5: Ejemplo de bricks, cada uno con un identificador único que se asocia a un nodo.

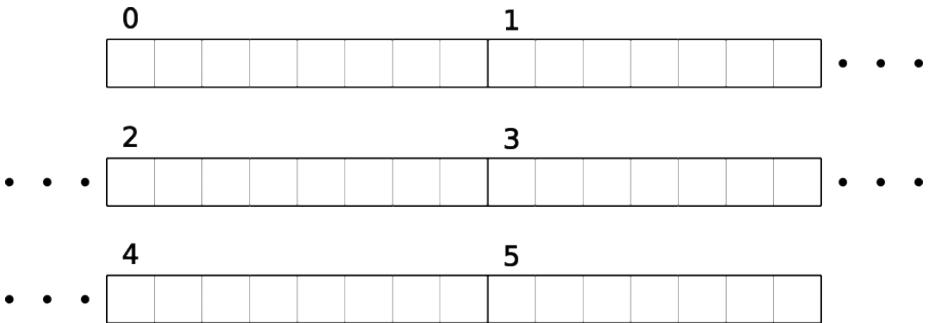


Figura 4.6: Ejemplo de nodos, cada uno con un identificador único.

Cada nodo tiene asociado un brick, ambos identificados de forma única por un mismo índice, como se puede ver en las figuras 4.5 y 4.6 respectivamente. Dado que los bricks existen en una textura 3D, la manera de localizarlo en memoria es con un vector en  $\mathbb{N}^3$ . Para esto, si se quiere encontrar el brick correspondiente a un nodo dado su identificador, dado que nodo y brick comparten un mismo identificador en  $\mathbb{N}$ , se usa una función que convierte el índice del nodo en un vector en  $\mathbb{N}^3$ . El vector de  $\mathbb{N}^3$  ( $n, m, s$ ) es el índice del primer voxel del brick

buscado. El resto del *brick* se encuentra en las posiciones desde  $(n, m, s)$  hasta  $(n + 2, m + 2, s + 2)$ .

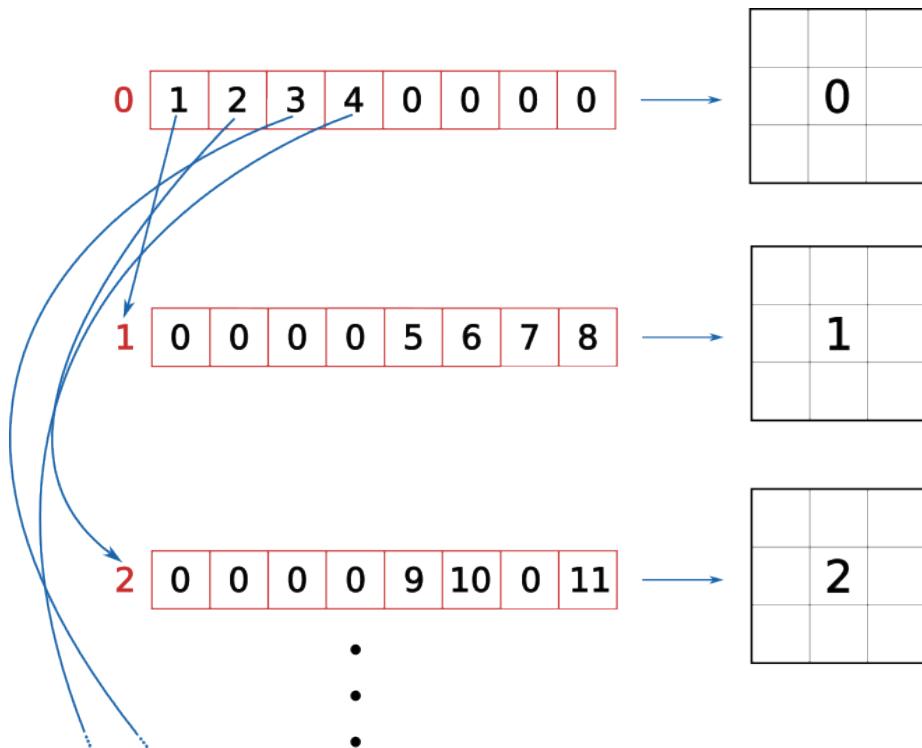


Figura 4.7: Ejemplo node pool y brick pool.

En la figura 4.7 se muestra un ejemplo de nodos, con sus bricks asociados. Además cada nodo contiene 8 téxeles, los cuales representan los hijos del nodo, donde el valor de cada téxel es el identificador de un nodo.

Ya que el árbol es disperso se necesita un valor que represente que un hijo no contiene geometría, que representa espacio vacío. Se usa el valor 0. Cada téxel apunta a un nodo mediante su índice en la *node pool*. En la figura se muestran estos índices dentro de cada téxel.

Se usa el índice del nodo para identificar su *brick*, que es parte de una textura 3D. Cada brick se muestra al costado de su nodo asociado, con sus coordenadas dentro de la textura en el margen derecho. Cada téxel individual de un brick tiene sus propias coordenadas, como es el caso con los téxeles individuales de la *node pool*.

Propiedades de la escena, ya sea color o irradiancia, se almacenan en los *bricks*. Existe una función que lleva posiciones dentro del nodo, que representa

una sección de la escena, a posiciones dentro de su brick asociado. En la figura 3.5 se vé porque es necesario, las esquinas del nodo se mapean a los centros de los vóxeles del brick. A la hora de tomar una muestra de una propiedad en un punto de la escena, se ubica el nodo correspondiente, se aplica la función y se toma el valor de la propiedad en el *brick*. Si el punto no es el centro de uno de los vóxeles del brick, se debe interpolar el valor utilizando los vóxeles adyacentes. Almacenar los bricks en una textura 3D en lugar de en una textura lineal trae beneficios de rendimiento, ya que las GPUs proveen interpolación trilineal acelerada por hardware, lo que elimina la necesidad de implementarla, y se obtiene mayor rendimiento.

#### 4.2.3. Menú

A lo largo de la implementación, fue necesario depurar varios errores y correr pruebas. Para esto, fue muy útil contar con una interfaz gráfica o menú para seleccionar varias opciones y poder ver valores de la GPU en tiempo real. Se desarrolló este menú utilizando un paquete del ecosistema de Rust llamado *Egui* [Ern20].

*Egui* permite rápidamente crear una interfaz gráfica basada en ventanas que se renderiza junto con la aplicación en cada frame. Es muy sencillo conectar valores del código a etiquetas en el menú y botones en el menú a acciones.

El menú de la aplicación cuenta con varias ventanas, o submenús, para ver valores, ajustar parámetros, y renderizar distintas imágenes. Uno de submenús para depurar muestra todos los nodos de la *node pool*, permite buscarlos por índice y coordenadas, y visualizarlos en el espacio.

Esta se puede ver en la figura 4.8. En el menú se muestran todos los nodos con su índice en el buffer y sus coordenadas dentro de la escena, entre 0 y la cantidad de vóxeles por dimensión, 512 o 1024. Al apretar cualquier nodo, se muestra en la escena un cubo delimitando la región de la escena representada por ese nodo. Se pueden mostrar muchos nodos a la vez.

También se usó la interfaz gráfica para reportar los FPS que fueron usados en el capítulo 5.

### 4.3. CLI

*CLI* (*Command-Line Interface*, interfaz por linea de comandos), es el punto de entrada de la aplicación. Consta de un archivo *main* que inicializa el contexto de OpenGL, la ventana de la aplicación, y utiliza *engine* y *core* para crear

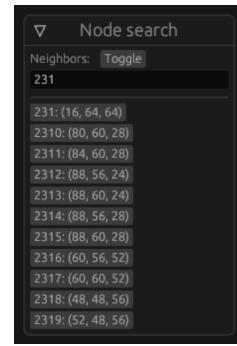


Figura 4.8: Menú de nodos.

la escena con iluminación global. También procesa varios tipos de archivos de configuración, que serán explicados a continuación. Todos estos tipos de archivos contienen información en formato RON, por *Rusty Object Notation*, una notación similar a JSON, *JavaScript Object Notation*, pero diseñada para Rust.

### 4.3.1. Archivo de configuración

Cada ejecución de la aplicación carga un archivo de configuración que se encarga de definir ciertos parámetros. Estos son la cantidad de vértices que se utilizarán, 256, 512 o 1024; las dimensiones de la pantalla, entre otros. A partir de esto, se inicializan otras variables, como la cantidad de niveles del octree, que es definida por la cantidad de vértices utilizados. Estos parámetros son utilizados a lo largo de toda la aplicación, por lo que se usó el patrón singleton, creando un tipo *Config*, que es inicializado una vez por *CLI* y luego utilizado en el resto de la aplicación.

### 4.3.2. Archivos de escena

Para poder fácilmente cargar distintos modelos y probar el algoritmo en ellos, se creó un formato de archivos de escena.

En estos archivos se definen la lista de objetos y luces de la escena. También se especifican listas de recursos para que los objetos refieran, modelos y materiales. Estos recursos son cargados primero y los objetos pueden reutilizarlos sin necesidad de copiarlos.

Utilizando estos archivos, se pueden definir muchas escenas distintas para probar la implementación.

#### 4.3.3. Archivos de valores predeterminados

Para poder iterar más rápido, se crearon archivos de valores predeterminados. Estos archivos contienen valores predeterminados de opciones que cambian con el transcurso de la ejecución. Por ejemplo, la posición de la cámara, qué tipo de imagen se está renderizando, qué nodos están siendo visualizados. Estos archivos también pueden guardarse a partir del menú.

Al iniciar el programa, se puede especificar un archivo de valores predeterminados, que altera el estado inicial de la escena. Son muy útiles cuando se está investigando algo en específico o viendo una imagen desde un punto de vista particular. Es posible guardar los valores utilizados, cambiar algo en el código, recompilar y observar cómo ese cambio afectó lo que se estaba observando.



# Capítulo 5

## Experimentación

Para analizar el rendimiento de la implementación, se realizaron experimentos en distintas tarjetas gráficas, las cuales se ven en el cuadro 5.1. Las escenas utilizadas se muestran en las figuras 5.1 y 5.2, las llamaremos  $S1$  y  $S2$  respectivamente. Se puede observar en cada una de ellas la luz indirecta difusa, luz indirecta especular y sombras suaves.

Nombre	GPU	Núcleos	Clock (GHz)	VRAM (GB)
$H1$	Intel Mesa ADL GT2	96	1.45	???
$H2$	Nvidia GTX 1660 Ti mobile	1536	1.45	6
$H3$	Nvidia Geforce RTX 4070	5888	1.92	12

Cuadro 5.1: Hardware utilizado.

Los experimentos varían la cantidad de véxeles por dimensión, y registran los cuadros por segundo alcanzados (FPS, por sus siglas en inglés) y el tiempo de construcción del *octree* en segundos. Para conseguir los FPS se ejecutó el programa durante un minuto, tomando una muestra cada segundo, luego se promediaron. Para el tiempo de construcción del *octree*, se construyó 50 veces y se calculó el tiempo promedió. Cada ejecución se realizó independientemente, para evitar posibles mejoras debido al uso del cache.

En las tablas 5.2, 5.3 y 5.4 se muestran los resultados de estos experimentos en tarjetas Intel Mesa ADL GT2 integrada de laptop, Nvidia GTX 1660 Ti de laptop y Nvidia RTX 4070 respectivamente. En cada tabla se muestran los resultados para cada una de las escenas  $S1$  y  $S2$ .

Vóxeles	Escena	FPS	Construcción del <i>octree</i> (s)
256	S1	14,47	4,62
	S2	17,05	3,82
512	S1	11,30	5,04
	S2	14,08	3,66
1024	S1	9,22	4,40
	S2	12,57	3,78

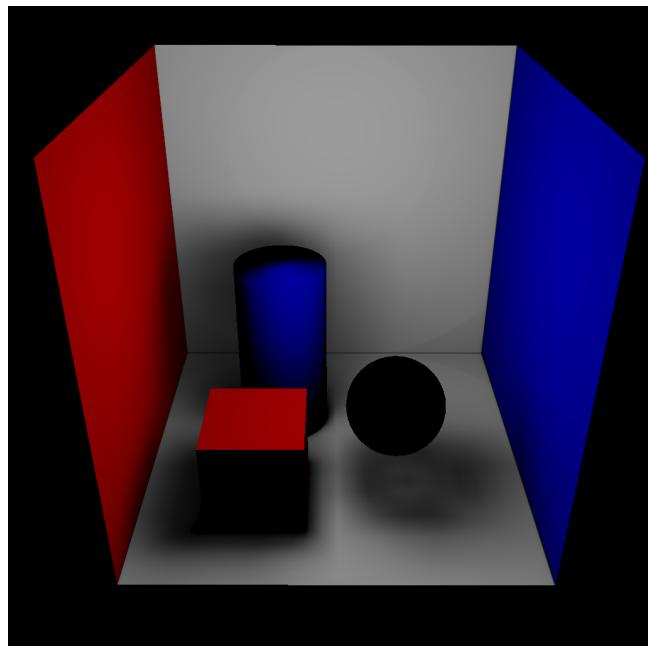
Cuadro 5.2: Experimentos usando *H1*.

## 5.1. Análisis de resultados

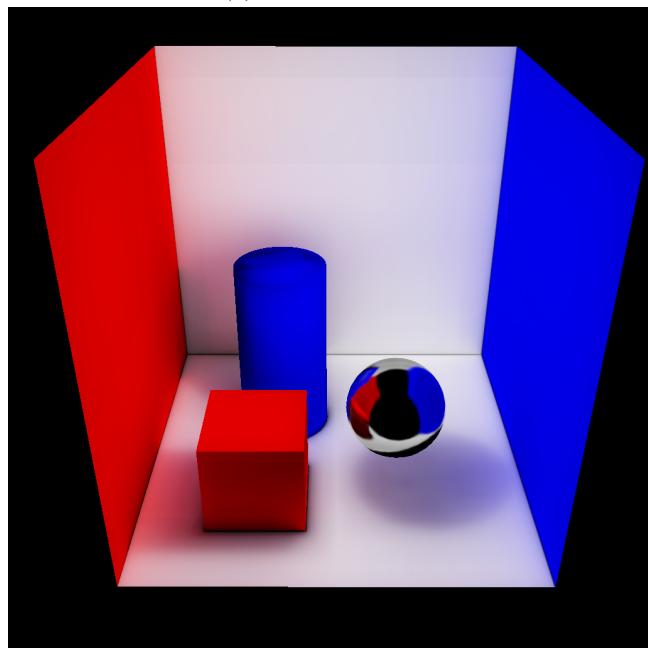
Una observación es la falta de rendimiento, dado que en las tarjetas gráficas de laptops en las que se ejecutó, se consiguieron muy pocos cuadros por segundo y en todas las tarjetas los tiempos de construcción del *octree* fueron muy altos. Los valores de estas métricas son mucho mejores en el artículo original del algoritmo, en el que se mostraron entre 20 y 30 cuadros por segundo y 280 milisegundos de construcción del *octree*, en lugar de segundos.

Una observación es que la tarjeta Nvidia RTX 4070 trae excelentes resultados en términos de cuadros por segundo, pero sorprendentemente es más lenta que la Intel Mesa ADL GT2 en construir el *octree*. Esto puede deberse a que la primer tarjeta posee mucho más paralelización, por lo que puede realizar todos los trazados de conos sin problemas, pero este no es el cuello de botella para la construcción del *octree*, sino la comunicación entre CPU y GPU.

Fingxels logró correr escenas a aproximadamente 15 FPS en tarjetas gráficas de laptops, mientras que el artículo original logró aproximadamente 30 FPS. En una tarjeta de última generación se lograron aproximadamente 140 FPS. Sin embargo, el tiempo de construcción de la estructura de datos fue mucho más lento que en el trabajo original para todas las tarjetas probadas.



(a) Solo luz directa.



(b) Luz directa e indirecta.

Figura 5.1: Escena de prueba con geometría simple.



Figura 5.2: Sponza.

Vóxeles	Escena	FPS	Construcción del <i>octree</i> (s)
256	<i>S1</i>	13,70	2,46
	<i>S2</i>	13,79	2,48
512	<i>S1</i>	11,87	2,63
	<i>S2</i>	11,92	2,57
1024	<i>S1</i>	10,58	2,96
	<i>S2</i>	10,65	2,92

Cuadro 5.3: Experimentos usando *H2*.

Vóxeles	Escena	FPS	Construcción del <i>octree</i> (s)
256	<i>S1</i>	141,57	1,88
	<i>S2</i>	141,59	1,90
512	<i>S1</i>	141,47	1,90
	<i>S2</i>	141,51	1,91
1024	<i>S1</i>	140,88	1,89
	<i>S2</i>	141,21	1,99

Cuadro 5.4: Experimentos usando *H3*.



## Capítulo 6

# Conclusiones y Trabajo Futuro

Se logró pasar de un artículo académico a una implementación práctica, open source. Se enfocó en aprender a fondo el algoritmo y en documentar la implementación para que sea un recurso útil para otros estudiantes interesados en este tema. Dicho esto, se cumplieron los objetivos principales.

La mayoría del tiempo fue dedicado a entender los conceptos y la programación en GPU mediante el uso de *compute shaders*. Esta fue la principal dificultad encontrada en el transcurso del trabajo, implementar un algoritmo principalmente en la GPU. El equipo tiene mayoritariamente experiencia programando en la CPU y manipulando etapas del pipeline gráfico, pero el uso extensivo de *compute shaders*, texturas (lineales, 2D, 3D), imágenes y *samplers* fue algo que sin dudas enlenteció el desarrollo.

A su vez, en el transcurso del trabajo, la falta de planificación, fijación de objetivos y fechas límite fue otro problema.

Se plantean posibles tareas de un trabajo futuro:

- Objetos dinámicos

Es posible tener objetos dinámicos en la escena, que al moverse subdividen el *octree* nuevamente. La estructura fue implementada con esto en mente, pero quedó fuera del alcance.

- Pasar a Vulkan

Pasar el programa a Vulkan sería un buen experimento para aprender y medir la mejora en eficiencia que resulta, dado que Vulkan permite mejor control de la GPU lo que ofrece más oportunidades de optimización.

- Mejores herramientas interactivas de exploración  
Mejorar las herramientas interactivas para la exploración del algoritmo.

# Bibliografía

- [Ake<sup>+</sup>18] Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michał Iwanicki y Sébastien Hillaire. *Real Time Rendering*. Boca Raton, London y New York: CRC Press, 2018.
- [Ama84] John Amanatides. “Ray tracing with cones”. En: *ACM SIGGRAPH Computer Graphics* 18.3 (1984), págs. 129-135.
- [BD02] David Benson y Joel Davis. “Octree Textures”. En: *ACM Transactions on Graphics* 21.3 (2002), págs. 785-790.
- [Bli77] James F. Blinn. “Models of light reflection for computer synthesized pictures”. En: *ACM SIGGRAPH Computer Graphics* 11.2 (1977), págs. 192-198.
- [CG12] Cyril Crassin y Simon Green. “Octree-Based Sparse Voxelization Using the GPU Hardware Rasterizer”. En: *OpenGL Insights*. Ed. por Christophe Riccio Patrick Cozzi. Boca Raton, Florida: CRC Press, 2012.
- [Cra<sup>+</sup>09] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre y Elmar Eisemann. “GigaVoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering”. En: *ACM Symposium on Interactive 3D Graphics and Games*. Boston, United States: ACM, 2009, págs. 15-22.
- [Cra<sup>+</sup>11] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green y Elmar Eisemann. “Interactive Indirect Illumination Using Voxel Cone Tracing”. En: *Computer Graphics Forum* 30.7 (2011), págs. 1921-1930.
- [DN04] Philippe Decaudin y Fabrice Neyret. “Rendering forest scenes in real-time”. En: *Rendering Techniques (EGSR)*. 2004, págs. 93-102.
- [Ern20] Emil Ernerfeldt. *Egui: an easy-to-use GUI in pure Rust*. <https://github.com/emilk/egui>. Última actualización: 8 de mayo 2023 (TODO: Actualizar). 2020.

- [FSJ01] Ronald Fedkiw, Jos Stam y Henrik Wann Jensen. “Visual Simulation of Smoke”. En: *SIGGRAPH* (2001).
- [Gal21] Alain Galvan. *A Comparison of Modern Graphics APIs*. <https://alain.xyz/blog/comparison-of-modern-graphics-apis>. Accedido el 14 de diciembre 2023. 2021.
- [HAO05] Jon Hasselgren, Tomas Akenine-Möller y Lennart Ohlsson. “Conservative Rasterization”. English. En: *GPU Gems 2*. Addison-Wesley, 2005, págs. 677-690.
- [Jen01] Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping*. Wellesley, Massachusetts: AK Peters, 2001.
- [Kaj86] James T. Kajiya. “The Rendering Equation”. En: *ACM SIGGRAPH Computer Graphics* 20.4 (1986), págs. 143-150.
- [Khr23] Khronos. *The OpenGL Shading Language*. 4.6. Khronos Group. Ago. de 2023.
- [Lou65] Rodney Loudon. *The Quantum Theory of Light*. 1965.
- [Max73] James Clerk Maxwell. *A Treatise on Electricity and Magnetism*. Clarendon Press, 1873.
- [MU12] Rosana Montes y Carlos Ureña. *An Overview of BRDF Models*. <http://hdl.handle.net/10481/19751>. Mar. de 2012.
- [PJH23] Matt Pharr, Wenzel Jakob y Greg Humphreys. *Physically Based Rendering*. Cambridge, Massachusetts: The MIT Press, 2023.
- [SA19] Niklas Smal y Maksim Aizenshtein. “Real-Time Global Illumination with Photon Mapping”. English. En: *Ray Tracing Gems*. Springer Science+Business Media, 2019, págs. 409-434.
- [Ush22] Will Usher. *Tiny OBJ Loader in Rust*. <https://github.com/Twinklebear/tobj/tree/3.2.2>. Abr. de 2022.
- [Wan<sup>+09</sup>] Rui Wang, Rui Wang, Kun Zhou, Minghao Pan y Hujun Bao. “An Efficient GPU-based Approach for Interactive Global Illumination”. En: *ACM Transactions on Graphics* 28.91 (2009), págs. 1-8.
- [WCH20] Zhihao Wang, Jian Chen y Steven C. H. Hoi. “Deep Learning for Image Super-Resolution: A Survey”. En: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 43.10 (2020).
- [Whi80] Turner Whitted. “An improved illumination model for shaded display”. En: *Communications of the ACM* 23.6 (1980), págs. 343-349.