



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 Repùblica

Supervisores

Eduardo Fernández
Jose Pedro Aguerre

Montevideo, 23 de diciembre de 2023



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, que 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. Esta implementación 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

1. Introducción	1
1.1. Organización del documento	2
2. Revisión de antecedentes	3
2.1. ¿Qué es la luz?	3
2.2. Radiometría: Unidades de la luz	5
2.3. Iluminación local e iluminación global	6
2.3.1. Iluminación local	7
2.3.2. Iluminación global	7
2.4. BRDF	9
2.5. Trazado de rayos (<i>ray tracing</i>)	10
2.6. Trazado de conos	11
2.7. Photon Mapping	12
2.8. Vóxeles	13
2.9. Octrees	15
2.10. Ducto gráfico	16
2.11. Renderizado diferido	18
3. Voxel cone tracing	19
3.1. Voxelización	20
3.1.1. Rasterización conservativa	21
3.2. <i>Octree</i> disperso	21
3.2.1. Nodos y bricks	22
3.2.2. Construcción	23
3.2.3. Border transfer	24
3.2.4. Nodos frontera	25
3.2.5. Filtrado	25
3.2.6. Filtrado anisotrópico	28
3.3. Inyección de fotones	29
3.4. <i>Cone tracing</i>	29
3.5. Oclusión ambiental	30
3.6. Conos de sombra	31

4. Implementación	33
4.1. <i>Engine</i>	35
4.2. Core	35
4.2.1. Representación del <i>octree</i>	35
4.2.2. Menú	36
4.3. CLI	37
4.3.1. Archivo de configuración	37
4.3.2. Archivos de escena	38
4.3.3. Archivos de predeterminados	38
5. Experimentación	39
6. Conclusiones y Trabajo Futuro	43

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 ellas se basan en el trazado de rayos. Esto consiste en trazar rayos a partir de la cámara hacia la escena y simular los caminos que recorren los fotones. Esto no funciona en tiempo real. Se han hecho varias optimizaciones a lo largo de los años para lograrlo, como por ejemplo simplificaciones en la geometría, el uso de estructuras jerárquicas, clustering, entre otras. Más allá de los avances, el problema continúa siendo un área activa de investigación. Métodos recientes utilizan hardware especializado para lograr alcanzar tiempos interactivos. Una técnica reciente (2019) es DLSS (*Deep Learning Super Sampling*), que consiste en ejecutar estos algoritmos en resoluciones de pantalla mucho menores a la objetivo, y luego, utilizar aprendizaje automático para agrandar esa pantalla a la objetivo.

Este trabajo 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. Este algoritmo fue propuesto por Crassin et al en 2011 [Cra⁺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 vóxeles 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.

Este trabajo surge del interés de los miembros del equipo en técnicas de

iluminación global y en véxeles 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 trabajo 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 intentando aprender sobre distintas técnicas de iluminación.

Se esperaba lograr una implementación funcional del algoritmo y se logró.

1.1. Organización del documento

Las siguientes secciones de este informe se organizan de la siguiente manera. El capítulo 2 se enfoca en analizar trabajos anteriores y proporcionar el trasfondo necesario para entender 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 explicaran 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? Esta pregunta es relevante porque, más allá de las variaciones, la gran mayoría de técnicas de iluminación global se basan en modelar a la luz físicamente.

El libro *Physically Based Rendering*, de Pharr, Jakob y Humphreys [PJH23] 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 5 a 6 antes de cristo) veía a la luz como una colección de pequeñas partículas viajando a través de rayos a una alta velocidad. En el siglo 5 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, 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 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

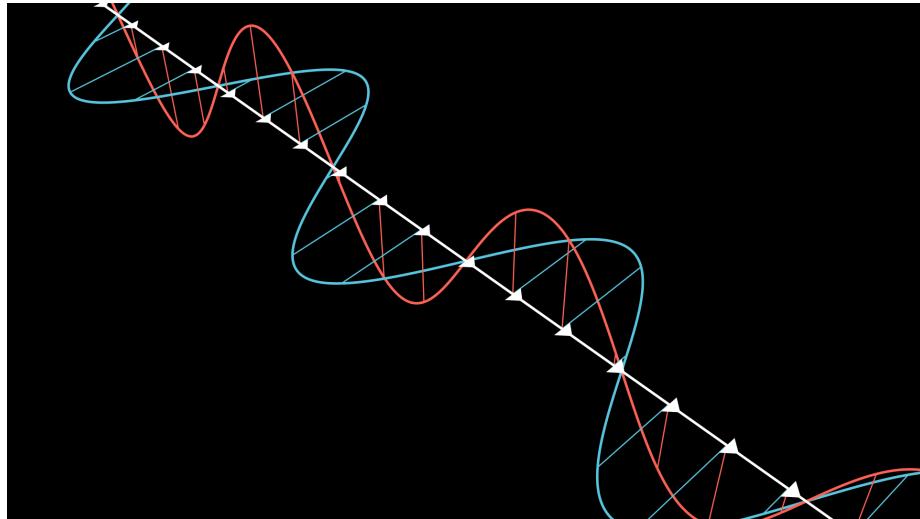


Figura 2.1: Representación de una onda electromagnética propagando por el espacio

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 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. Esto 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, 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⁺18]. Esto remueve otros fenómenos como la fluorescencia, la fosforescencia y la polarización.

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⁺18] 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 λ tiene una energía $Q = \frac{hc}{\lambda}$, donde c es la velocidad de la luz y h es la constante de Planck, el cuanto.

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.

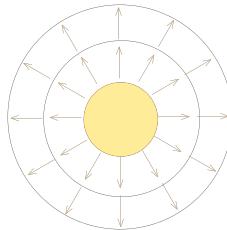


Figura 2.2: Flujo radiante en una luz puntual

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. 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 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. El ángulo sólido se representa con el símbolo ω . En la figura 2.3 se puede ver un cono con un ángulo sólido de 1 estereorradián. El ángulo sólido que abarca todas las direcciones posibles mide 4π estereoradianes.

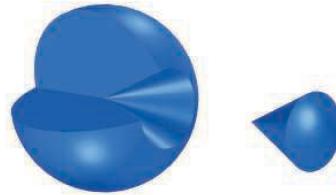


Figura 2.3: Un cono con un ángulo sólido de 1 estereorradián removido de una esfera. Fuente: [Ake⁺¹⁸]

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 estereorradián.

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 estereorradián.

$$L = \frac{d^2\Theta}{(dA \cos \theta)d\omega}$$

La radiancia es lo que miden los sensores, como los ojos o cámaras. El objetivo 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, un campo relacionado, la fotometría 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.

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

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.

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.

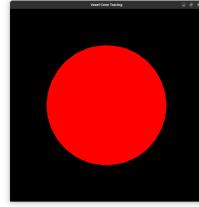
Un modelo ampliamente utilizado para iluminación local es el de Blinn-Phong, [Bli77]. Este posee tres componentes principales: luz **ambiente**, **difusa** y **especular**. La luz ambiente es uniforme y está presente en toda la escena. No proviene de una fuente de luz en particular y simula el efecto 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 especular simula el brillo que se ve cuando la luz se refleja sobre superficies pulidas. Este componente 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.

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. Esto causa que sea difícil alcanzar tiempos interactivos.

Entre las técnicas más utilizadas se encuentra el trazado de rayos, la radiosidad



(a) Sin sombreado



(b) Con sombreado difuso

Figura 2.4: Esfera iluminada mediante el modelo local de Blinn-Phong

dad, 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. Esto significa que 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. 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 son cantidades que pueden derivarse de estas.

Esta ecuación es la base de varios métodos de iluminación global [Ake⁺18]. Se puede observar que esta ecuación tiene en cuenta toda la escena para calcular la luz en un punto.

La ecuación no presenta una solución analítica cerrada para la mayoría de los casos, por lo que se calculan soluciones numéricas para su resolución.

La ecuación 2.1 considera toda la iluminación de la escena, incluyendo tanto luz directa como indirecta. La **ecuación de reflectancia** se deriva de la de renderizado y se enfoca en el cálculo de la luz indirecta.

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

Esta ecuación calcula la radiancia saliente L_o de un punto p y un pequeño ángulo sólido ω_o como una integral sobre el hemisferio Ω centrado en el punto. Sus términos son muy parecidos a los de la ecuación de renderizado. Vemos que ya no están 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 ω_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 Ω 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 ω entrante y una saliente ω' , retorna cuánta luz se refleja desde la dirección entrante hacia la saliente en el punto.

Está función 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 casos teóricos e ideales son los siguientes:

- Reflexión especular: $f(x, \omega, \omega') = \rho$ cuando ω es simétrico a ω' respecto a la normal. 0 en otro caso.

- Reflexión difusa: $f(x, \omega, \omega') = \frac{\rho}{\pi}$ para todas las direcciones donde ρ 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.

Estos BRDFs son muy utilizados, dado que simplifican mucho los cálculos y son físicamente 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. Este 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, en 1980, [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 se 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 Snell¹. En la figura 2.5 se puede ver una imagen generada por este método.

Este método sufre de *aliasing*, los bordes irregulares. Esto ocurre debido a una falta de resolución, es decir, cantidad de píxeles, que no logra capturar todo el detalle de la imagen. Suele notarse al representar curvas. Se puede ver en la figura 2.5.

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 que se toman. Debido a la variancia del muestreo aleatorio, este método sufre de ruido, un granulado en la imagen final. El mismo disminuye a medida que se toman más muestras.

¹ $n_1 \sin \theta_1 = n_2 \sin \theta_2$

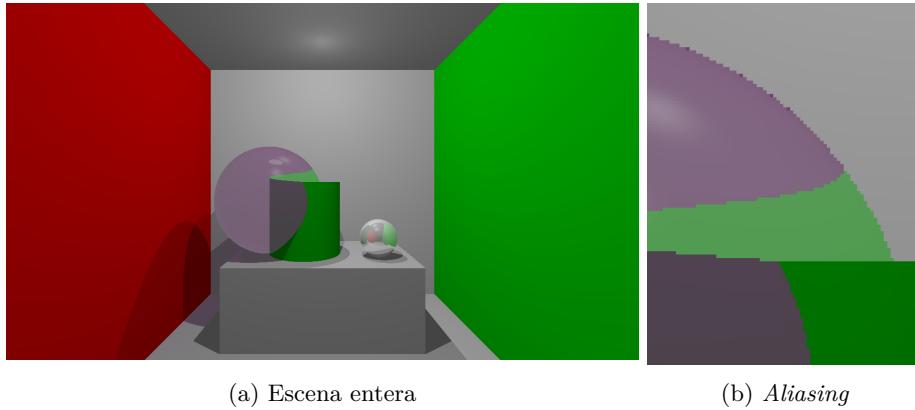


Figura 2.5: Escena renderizada con el trazado de rayos de Whitted. Implementación propia.

Tanto el *aliasing* como el ruido pueden mitigarse con una técnica llamada *supersampling*. Consiste en tomar más de una muestra por píxel, lo que significa lanzar más de un rayo por píxel y luego promediar los resultados. Esto 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. Esto imposibilita 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 cual agrega un grosor no despreciable. Usando este grosor, los conos son capaz de no solo probar la existencia de intersecciones, sino también calcular el porcentaje de área de la intersección. Este método fue propuesto 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 variancia

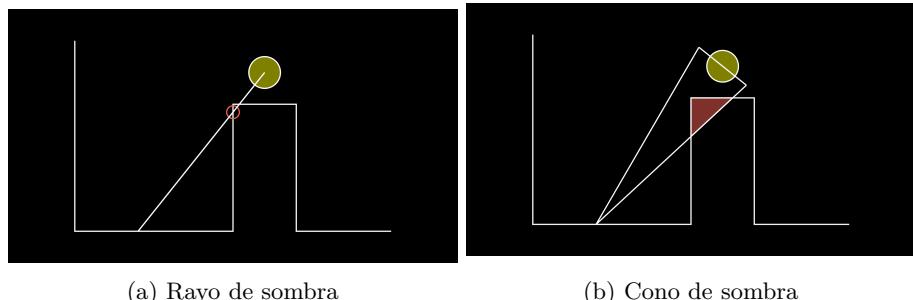


Figura 2.6: Los conos de sombra no solo devuelven si hay intersección, también devuelven el porcentaje de área de la intersección

del muestreo.

Esta idea de 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. Esta respuesta binaria, si 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 ocultación de ese punto, lo cual lleva a una escala de grises y a sombras suaves. En la figura 2.6 se pueden ver diagramas mostrando esta diferencia.

Esta técnica sigue hallando la intersección de manera analítica, por lo que las ecuaciones son mucho más complejas. Tanto es así que originalmente Whitted había considerado usar conos pero la complejidad añadida de las ecuaciones lo hizo descartarlos. Aún así, los beneficios superan a las desventajas en la mayoría de los casos.

Para calcular la luz indirecta se halla la radiancia saliente de un punto en dirección a la cámara. En *path tracing*, este valor se calcula approximando la integral sobre el hemisferio con el método de Monte Carlo. Esto implica tomar varias muestras, que son los rayos que son lanzados desde p en todas las direcciones ω_i . En el caso de 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 vidrio o 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 2.7 se puede ver este efecto.

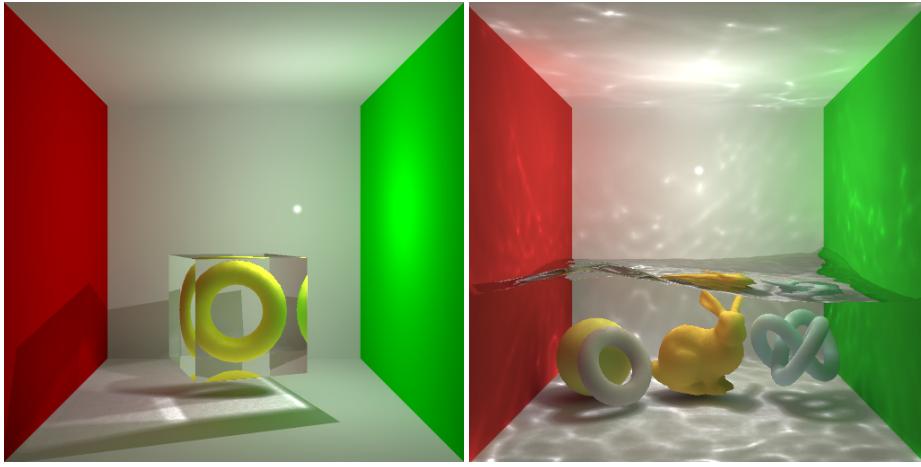


Figura 2.7: Cáusticas. Fuente: [Wan⁺09]

El algoritmo comienza con una etapa en la que se lanzan fotones desde la fuente de luz hacia la escena. Estos fotones se almacenan en las superficies difusas de la escena, creando un **mapa de fotones**. Estos fotones se usan en una segundo 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 2.8.

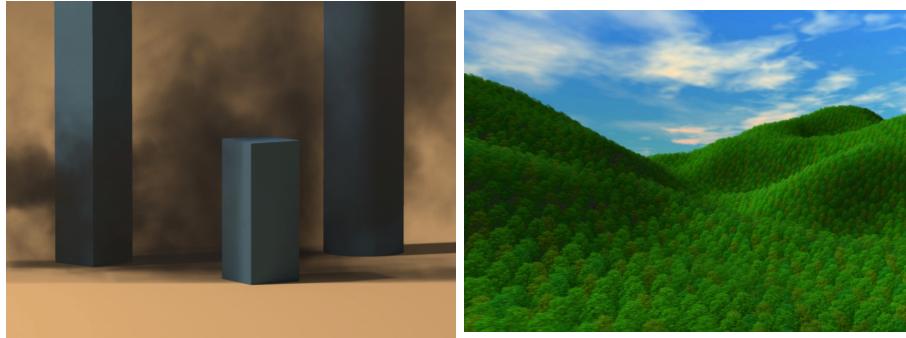
2.8. Vóxeles

Un voxel es el equivalente de un píxel en el espacio 3D. Así como un píxel es un elemento de imagen, *picture element*, un voxel es un elemento de volumen, *volume element* [Ake¹⁸]. Similar a como los píxeles se ubican en una grilla que divide una superficie 2D en secciones cuadradas, los véxeles dividen un volumen 3D en cubos.

Tradicionalmente, se usan para guardar datos volumétricos y como primitivas para renderizar una variedad de objetos. Permiten representar volúmenes, en



Figura 2.8: Photon mapping en tiempo real. Fuente: [SA19]



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

Figura 2.9: Vóxeles para renderizar humo y vegetación

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 2.9.

Cada voxel 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 voxel. En general, no es necesario guardar la posición de un voxel, 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 voxels se llama voxelización. Esto involucra interseccar la representación con la grilla de voxels, y marcar como ocupados los voxels que se solapan con esta.

Algunos programas usan grillas completas de voxels. Sin embargo, en la mayoría de los casos no es necesario. Una observación útil es que en muchas aplicaciones es suficiente tener voxels en el límite entre un objeto y el espacio vacío, siendo innecesario el relleno.

Otra observación útil es que, para una buena representación, es suficiente con tener mucho detalle en la frontera entre espacio vacío y 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⁺18]. Suele usarse para representar datos espaciales [BD02]. Se puede utilizar para dividir el espacio 3D de manera jerárquica, con varios niveles.

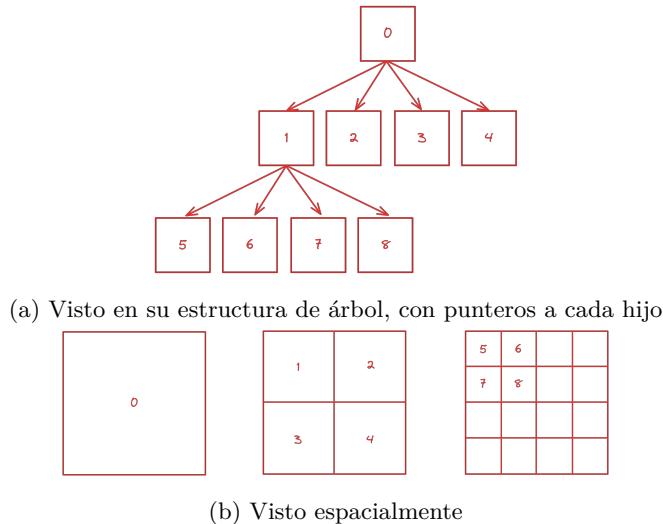


Figura 2.10: Quadtree

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 2.10 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 véxeles 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 véxeles 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?

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 raster, el de cómputo de propósito general, el de trazado de rayos, entre otros. El ducto raster es importante, dado que es utilizado en *voxel cone tracing*.

El ducto raster 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. Esto aporta un gran grado de 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 raster.

En OpenGL, por ejemplo, este ducto posee las etapas que se ven en la figura 2.11 [Ake⁺18].

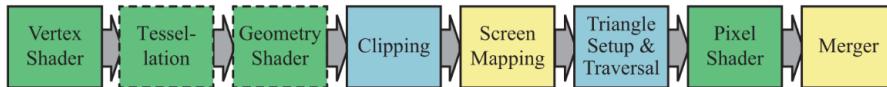


Figura 2.11: Ducto de raster. Fuente: [Ake⁺18]

De estas, tres son programables. El desarrollador debe escribir *shaders* para estos pasos, que se pueden escribir en el lenguaje GLSL [Khr23]. Los *shaders* son programas que ejecutan en la GPU. Estos programas son ejecutados 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. Esto se logra usando tres matrices que se conocen como modelo, vista y 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 ejecutan mínimo un hilo por cada píxel 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 sean transparentes.

Otro ducto muy relevante, que es usado para implementar la mayoría del algoritmo, es el ducto de cómputo de propósito general. Este es muy simple, 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 cálculos pesados por fragmento, se puede estar realizando trabajo innecesario, debido a una prueba de profundidad. Dados dos fragmentos que podrían darle color a un mismo píxel, una prueba de profundidad descarta el que se encuentra detrás del otro, a menos que halla transparencia.

En el renderizado diferido, se procesan todos los vértices del programa pero se saltean cálculos pesados en el *fragment shader*. Se guarda toda la información necesaria para etapas posteriores en texturas llamadas *geometry buffers*, datos como posición, normal, color y cualquier otro dato relevante de la geometría a la hora de realizar cálculos pesados de sombreado. Estos suelen ser un buen porcentaje del total, teniendo en cuenta los objetos fuera del ángulo de vista 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 cálculos simples de iluminación, la complejidad extra del renderizado diferido no está justificada dado que el aumento de rendimiento es despreciable. A su vez, *forward rendering* soporta mejor la transparencia. Sin embargo, a medida que la complejidad de la escena y los métodos 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⁺11], un algoritmo de iluminación global en tiempo real, que utiliza conceptos presentes en el trazado de rayos (2.5) y *photon mapping* (2.7). Reduce los costos asociados a estos algoritmos clásicos trazando conos en lugar de rayos (2.6) y usando una representación de la escena de vóxeles (2.8) almacenada en un *octree* disperso (2.9).

Otra forma de reducir costos y aumentar la velocidad del algoritmo es reutilizar el ducto raster 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. Esto difiere del trazado de rayos clásico, que renderiza toda la escena puramente mediante intersecciones entre rayos y geometría.

La principal aproximación que realiza el algoritmo es trabajar sobre una representación voxelizada pre-filtrada de la escena. Se recorre el árbol usando las coordenadas para hallar un vóxel que represente esa región del espacio. Este vóxel contiene toda la información necesaria sobre oclusión, color, irradiancia, debido a que la estructura es previamente filtrada. En el filtrado, la información parte de las hojas del árbol y es promediada hacia niveles mayores hasta que todo el árbol tiene información. La información del vóxel en ese nivel resume la información de todos los hijos. Se vió en la sección sobre trazado de conos, 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 muestran distintos puntos a lo largo del cono. En cada punto, se toma el diámetro del cono y se usa para calcular el nivel del *octree* necesario, dado que niveles mas altos del octree son una aproximación de sus hijos con menos lujo de detalle. Esto permite acumular la información a lo largo del cono de manera rápida.

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 raster, 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.

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 raster de OpenGL, se procesan todos los triángulos de la geometría de la escena. Esto genera una lista de **vóxeles**, en los que se almacena la posición y el color del triángulo original. Cada uno de estos véxeles será usado para construir el árbol y terminará almacenado en la estructura.

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

1. El plano de B interseca V .
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 3.1.

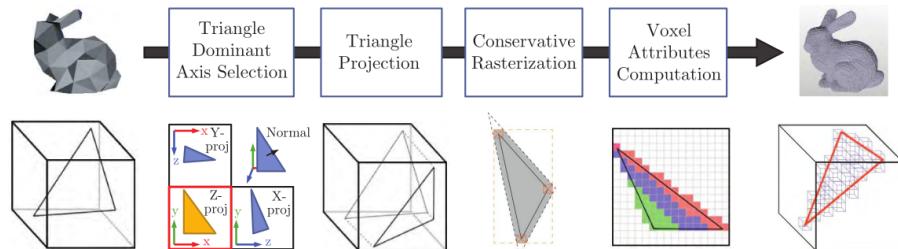


Figura 3.1: Ducto de voxelización. Fuente: [CG12]

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. Esto se realiza para maximizar el área del triángulo proyectado.

Cada triángulo proyectado se rasteriza para conseguir fragmentos correspondientes a la resolución 3D de la grilla de vértices. Se fija el tamaño del *viewport* a coincidir con la cantidad de vértices, por ejemplo un *viewport* de tamaño 512×512 para una grilla de 512^3 vértices.

Durante la rasterización, cada triángulo genera un conjunto de fragmentos 2D. Debido a la elección de la dimensión dominante de la normal para la proyección, solo pueden intersecar con 3 vértices en profundidad. Entonces, por cada fragmento 2D, los vértices que intersecan con el triángulo se calculan en el *fragment shader*, basándose en la posición, la profundidad y las derivadas en espacio de pantalla.

Luego de realizada la voxelización, se obtiene la lista de vértices necesaria para crear el árbol, con su posición y color. Estos vértices son una generalización en 3D de los fragmentos 2D. Cada uno tiene una coordenada que lo identifica dentro de la grilla 3D de la escena, así como color y posición.

3.1.1. Rasterización conservativa

El método descrito anteriormente a veces no crea vértices para elementos muy finos, como un asta de bandera. Esto pasa porque 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 toca un píxel (en cualquier punto) también toca su centro. Esto se logra alargando las aristas del triángulo hacia afuera, la mitad de la diagonal de un píxel, generando un triángulo semejante. Hay fragmentos que resultan de sobreestimar la cobertura de este triángulo, dado que este nuevo triángulo puede tocar el centro de píxeles que antes no tocaba. 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 3.2.

3.2. Octree disperso

Para almacenar los vértices generados, se usa un *octree* disperso, como los vistos en la sección 2.9. Esta estructura subdivide la escena en 8 y cada hijo en 8 y así sucesivamente. Al ser disperso, puede ser que ciertos hijos no se subdividen si no hay más geometría dentro de la región de la escena que representan.

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. Si el árbol

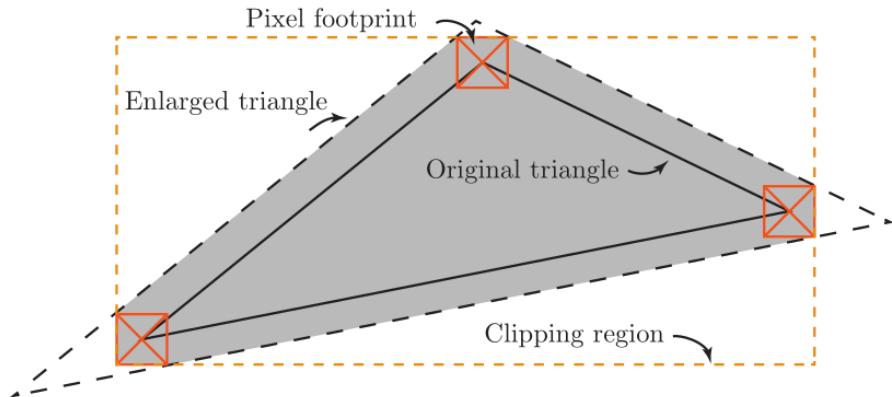


Figura 3.2: Rasterización conservativa. Fuente: [CG12]

fuerza denso, cada nivel n tendría 8^n nodos. El nivel 0 tendría 1 nodo, el nivel 1 tendría 8, el 2 64 y así sucesivamente. Al ser un árbol disperso, no se crean nodos para regiones de la escena que no contienen geometría. El último nivel del árbol es el que llega a la resolución deseada de 512 o 1024 vértices. Valores más altos de resolución crean más niveles del *octree* y hacen que se asemeje cada vez más la aproximación de vértices a la geometría real.

Dado que la estructura tiene como máximo 512 o 1024 vértices 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 y bricks

Los nodos del árbol no almacenan los vértices 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, $3 \times 3 \times 3$, vértices. Son estos vértices los que almacenan los valores de la escena. En la figura 3.3 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, esto es porque los vértices se centran en sus vértices. Esto es útil para que los bricks puedan obtener valores de sus vecinos, lo que garantiza que la interpolación dentro de un solo brick toma en cuenta los valores de sus vecinos. Esto resulta en una frontera compartida entre vecinos, como se puede ver en la figura 3.4. Esta frontera debe almacenar lo mismo para que la interpolación a la hora de generar la imagen final funcione. 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.

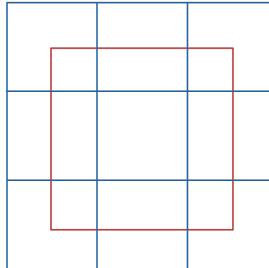


Figura 3.3: Nodo (en rojo) con su brick asociado (azul)

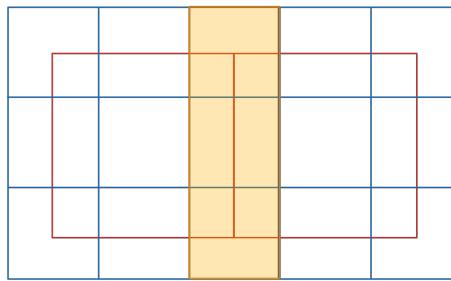


Figura 3.4: Solapamiento entre vóxels de bricks de nodos vecinos

3.2.2. Construcción

Para generar esta estructura se usa la lista de vóxeles generada durante la voxelización. Se empieza con un árbol con un solo nivel, con un solo nodo que ocupa toda la escena. Se ejecuta el algoritmo a continuación sobre este nivel para generar el siguiente, y luego se continúa aplicándolo en cada nivel del árbol hasta completarlo.

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 vóxel de la lista de vóxeles. Dado un vóxel, se recorre el árbol construido hasta el momento, hasta que se llega a una sección del nodo no subdividida. Esta sección del nodo se marca para ser subdividida.

Luego, se ejecuta *allocate_nodes*, que busca en el nivel i secciones marcadas para subdividir. Al encontrar una sección de un nodo marcada, crea un nuevo nodo en la estructura y cambia la marca por un puntero a ese nuevo nodo.

Siempre y cuando haya un fragmento en la región de la escena representada por un nodo, este será subdividido nivel tras nivel.

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 uno tiene la misma posición en la grilla de vóxeles. Esto último pasa más mientras más triángulos tiene la escena original y menos resolución tiene la grilla de vóxeles. Las hojas no

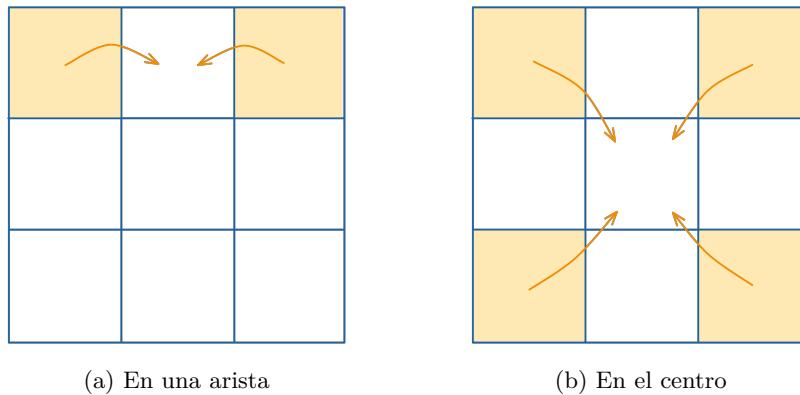


Figura 3.5: Funcionamiento de *spread_leaves* en 2D. Las flechas indican aporte al promedio

tienen hijos.

En 3.2.1, se vió como los bricks ocupan una región más amplia del espacio. Para consolidar esto con el tamaño de los véxels, estos se almacenan únicamente en las esquinas de los bricks. Se almacenan en la esquina más cercana a la posición del véxel. Luego, se aplica un programa *spread_leaves*, para espacer estos valores a lo largo de todo el brick. Esto funciona de la siguiente manera, la cual se muestra en 2D en la figura 3.5:

- El véxel central almacena el promedio de todas las 8 esquinas
- El véxel del medio de cada cara almacena el promedio de las 4 esquinas de su cara
- El véxel del medio de cada arista almacena el promedio de las 2 esquinas de esa arista

De esta manera, se espacian los valores de las esquinas a todo el brick. Este es el algoritmo que expande los véxeles generados para llenar los bricks $3 \times 3 \times 3$. A partir de aquí, el término “véxel” 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 mencionó anteriormente, las fronteras entre bricks son comparadas, corresponden al mismo espacio en la escena. Por lo tanto, los valores almacenados en esos véxeles deben ser los mismos. Esto garantiza una correcta interpolación a la hora de generar la imagen final. Sin embargo, luego de *spread_leaves*, esta frontera tiene valores distintos en cada brick vecino. Para igualarlos, se promedian los valores de la frontera con la del brick vecino, asegurándose que el nivel sea coherente. De esto se encarga *border_transfer*. Este

programa promedia los valores en la frontera de cada brick con la de sus vecinos, en X, Y y Z. De esta manera, aún cuando un vóxel puede estar en varios bricks, en 8 como máximo, su valor va a ser siempre el mismo en cada uno de ellos.

3.2.4. Nodos frontera

Al usar un octree disperso, no existen nodos donde no hay geometría. *Border transfer* asume que todo nodo tiene un vecino, pero esto no es siempre verdad, por ejemplo en el límite entre la geometría y el espacio vacío. Estos nodos si un vecino causan un problema, dado que no se puede promediar el valor de sus vóxeles con el vecino, lo que causa problemas de interpolación. Para que la interpolación continue y logre difuminar el sombreado, es necesaria una capa de nodos extra, los que llamaremos **nodos frontera**, entre la geometría y el espacio vacío.

Los nodos frontera se añaden en cada nivel del árbol a la hora de construirlo. Sus bricks no contienen valores, existen solo para interpolar los valores con 0 y así difuminar los bordes de la geometría. De esta forma los vóxeles compartidos entre espacio vacío y geometría tienen un promedio de ambas, similar al resto de los vóxeles compartidos entre nodos. El mismo problema no se genera entre estos nuevos nodos y el espacio vacío, dado que los vóxeles compartidos con el espacio vacío van a tener como valor total transparencia, por lo que se tiene una estructura consistente.

3.2.5. Filtrado

Una vez que todos los atributos se encuentran en las hojas del octree, estos deben ser 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. Esto se realiza en $n - 1$ pasos, con n el máximo nivel del octree. En cada paso, se calcula el valor de cada vóxel 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 3.6. En este caso se muestran solo 4 hijos porque es en 2D, en lugar de 8 como en el caso tridimensional. Cada hijo tiene a su vez su propio brick asociado como se muestra en la figura 3.7.

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

Dado el vóxel superior izquierdo de la figura 3.9, se considera solo el brick del hijo superior izquierdo del nodo. De ese brick, se consideran los vóxeles amarillos en la figura. El valor final del vóxel del padre se calcula promediando los valores de los vóxeles del hijo, pesados por el porcentaje de solapamiento. Si nombramos los vóxeles del brick hijo a, \dots, i y los del brick padre a', \dots, i' , como en la figura 3.8, entonces el valor del vóxel del padre se calcula como:

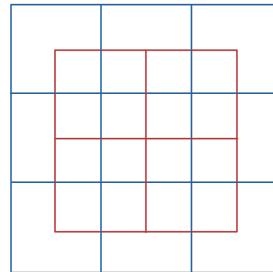


Figura 3.6: Nodo con su brick asociado y sus hijos

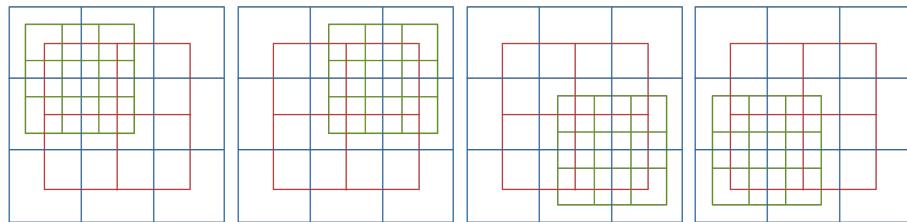


Figura 3.7: Bricks de todos los hijos del nodo

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

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

De la misma manera se calculan los vóxels 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 3.10 y 3.11.

a	b	c
d	e	f
g	h	i

Figura 3.8: Una posible forma de referirse a cada vóxel de un brick

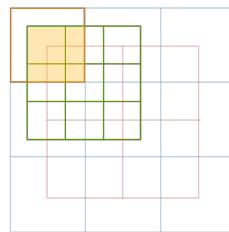


Figura 3.9: Filtrado para un vóxel esquina. Se puede ver el vóxel del brick padre cuyo valor se quiere calcular, junto con el brick del hijo y su solapamiento con este vóxel.

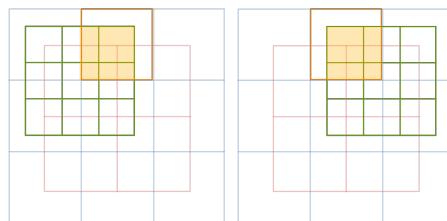


Figura 3.10: Filtrado para un vóxel borde

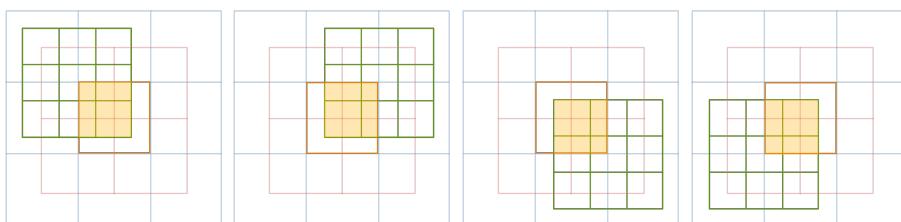


Figura 3.11: Filtrado para un vóxel cara

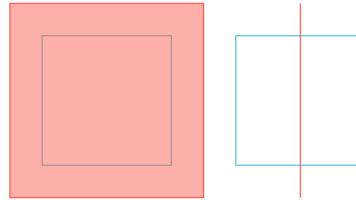


Figura 3.12: Vóxel que contiene una pared fina. En la izquierda, la pared se ve de frente. En la derecha, se ve de costado. Se espera que el valor del vóxel refleje esta diferencia entre las direcciones de vista.

3.2.6. Filtrado anisotrópico

El filtrado descrito en la sección anterior resulta en vóxeles cuyos valores son los mismos vistos en todas las direcciones. Una propiedad deseable es que se almacenen distintos valores, pudiendo usarse según el ángulo del que se mira. Esto es muy útil a la hora de representar un atributo como el color en una escena 3D.

Para ver por qué esta propiedad es deseable, consideremos una escena compuesta únicamente por una pared fina. Dado un vóxel de un nodo alto en el árbol, este representa una gran región del espacio. Esta región contiene únicamente una pared que pasa por su centro, y el resto es todo espacio vacío. Con el filtrado presentado anteriormente, llamado **isotrópico**, este vóxel tendrá un muy bajo valor de opacidad, dado que la mayoría de su espacio es vacío. Sin embargo, es fácil ver que es muy distinto ver la pared de frente que de costado. Como se muestra en la figura 3.12, la pared vista de frente es opaca pero vista de costado es mayoritariamente transparente la región que ocupa.

La solución es realizar el filtrado 6 veces por vóxel, 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. Una vez conseguidos los 6 valores, es posible conseguir cualquier dirección de vista interpolando las tres direcciones más cercanas. Este filtrado es **anisotrópico**, no es igual para todas las direcciones.

Dada una dirección, por ejemplo, de izquierda a derecha, se parte de los vóxeles de la izquierda y se calcula un valor para cada fila, partiendo de estos y yendo hacia los vóxeles de la derecha. Llamémosle al valor de cada fila **valor direccional**. En la figura 3.13 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.

Este tipo de filtrado soluciona situaciones como la de la pared mencionada anteriormente. Con él, el vóxel que contiene la pared es opaco en la dirección paralela a su normal y transparente en la dirección perpendicular.

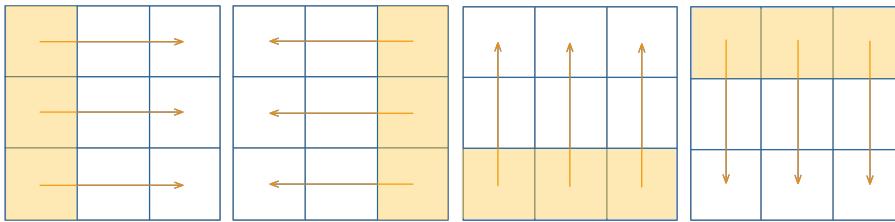


Figura 3.13: Filtrado anisotrópico en todas las direcciones

3.3. Inyección de fotones

Hasta ahora tenemos la estructura de datos creada, conteniendo el color 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*. Para hacer esto, se usa el ducto raster.

Se rasteriza la escena del punto de vista de la luz usando el ducto raster para generar una textura 2D. En lugar de tener colores en cada téxel de esta textura, se almacenan las posiciones de los objetos de la escena. La existencia de una posición en esta textura significa que esa posición es visible desde la luz, por lo cual debe recibir un fotón. Las posiciones son utilizadas para recorrer el árbol y almacenar los fotones en véxeles del mismo.

Los fotones se almacenan en los véxeles del árbol, esto es la **irradiancia**, el flujo recibido por la 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. A esto se le llama 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 raster. 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, se lanza uno o varios conos con cierta dirección y apertura, dependiendo del efecto que se quiere lograr.

Dado un cono, se parte desde su origen y se avanza en su dirección con un cierto tamaño de paso. Esto se conoce como *ray marching*. Después de cada paso tomado, se calcula el diámetro del cono en ese punto. Dado el diámetro, se calcula un nivel del octree, y dado ese nivel y la posición a lo largo del cono, se recorre el octree y se encuentra el nodo que corresponde a ese nivel y a esa posición. Ese nodo tiene un brick asociado, cuyos véxeles tienen los valores prefiltrados, conseguidos en 3.2.5. Se usa el valor del véxel que corresponde con la posición y se acumula. Se sigue avanzando paso a paso en el cono acumulando valores hasta satisfacer un criterio de parada. El algoritmo de *cone tracing* es simple dado todos los pasos anteriores.

Cada cono calcula un color c y una opacidad α . Si en cada paso consideramos c y α como los valores hasta el momento, y c_2 y α_2 como los valores nuevos encontrados en el véxel del paso, entonces en cada paso los valores de c y α se calculan de la siguiente manera:

$$\begin{cases} c = \alpha c + (1 - \alpha) \alpha_2 c_2 \\ \alpha = \alpha + (1 - \alpha) \alpha_2 \end{cases}$$

Para la luz indirecta difusa, se lanzan conos para cubrir el hemisferio centrado en la normal del punto. En la mayoría de los casos, 5 conos anchos difusos dan un buen resultado. Cada cono acumula el color de los véxeles con los que se encuentra multiplicado por la cantidad de fotones. Esto logra un efecto de *light bleed*, 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. Este cono, al ser más fino, se encuentra con 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. Este efecto no aporta más al realismo de una escena una vez que se usan los 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 del punto. El único valor necesario es la opacidad. A medida que se va viajando a lo largo de un cono, se va acumulando la opacidad de los véxeles que se tocan. Se define una distancia máxima y el criterio de parada es cuando el punto a lo largo del cono pasa esa distancia máxima.

3.6. Conos de sombra

De la misma manera que el trazado de rayos logra sombras lanzando un rayo hacia la fuente de luz, en este caso se logran 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 es que se logran sombras suaves, sin necesidad de tener que tomar muchas muestras y promediarlas.

Capítulo 4

Implementación

La primera decisión de relevancia en el proceso de desarrollo fue implementar el algoritmo en la GPU. La principal razón para esto es el alto grado de paralelismo que la misma ofrece. A su vez, el ducto raster, ampliamente implementado y optimizado en las GPUs, puede ser reutilizado. Esto reduce los cálculos necesarios a la hora de generar una imagen.

La programación en GPU precisa un lenguaje **motor**, una API de gráficos y un lenguaje de **sombreado**. El lenguaje motor ejecuta en la CPU y 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 motor 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 que esta ofrece.

Como lenguaje motor se eligió **Rust**, un lenguaje de sistemas moderno multi-paradigma. 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. Además, provee abstracciones de costo cero lo cual mejora la experiencia de desarrollo. Otra característica a mencionar es la seguridad de memoria que impide comportamientos indefinidos en tiempo de ejecución.

Se consideró a su vez C++ 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. El mismo permitió instalar varias dependencias más rápido y cambiarlas a lo largo del ciclo de desarrollo cuando esto fue necesario. En este caso, dada la implementación en la GPU, la eficiencia y el manejo de memoria es secundario.

Como API de gráficos se eligió **OpenGL**, junto con el lenguaje de sombreado **GLSL**, el más usado para OpenGL, aunque este 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,

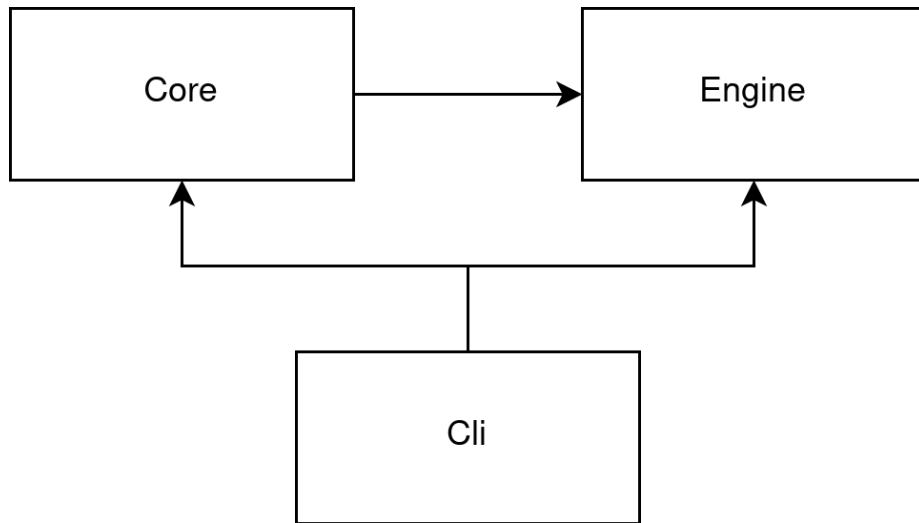


Figura 4.1: Arquitectura de la aplicación

es conocida por ser más compleja que OpenGL y el equipo del trabajo no contaba con la suficiente experiencia para utilizarlo adecuadamente. Esto 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, el cual hubo mucho, sin embargo, también fue utilizado el ducto raster, algo que CUDA no tiene. Por esto 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* 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* 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. Estos tres paquetes se muestran en la figura 4.1.

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

4.1. Engine

En este paquete, se implementaron las abstracciones usuales al trabajar con APIs de gráficos.

Transform es un tipo que contiene la posición, rotación y escala de un objeto en la escena. Esto simplifica el lidar con matrices de traslación, rotación y escala por todo el programa, todo se maneja en el *Transform*.

Camera es el punto de vista del cual se renderiza toda la escena. Se soportan tanto cámaras en perspectiva como ortogonales.

Debido al uso del ducto raster, la aplicación representa 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*. Se usa una abstracción *Model*, que contiene muchos *Mesh*.

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.

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

Este paquete es el corazón de la implementación, dado que es aquí donde se implementa el algoritmo. Contiene la lógica de la voxelización, los tipos de datos del *octree*, el trazado de conos y un menú para poder ajustar parámetros en tiempo de ejecución.

4.2.1. Representación del *octree*

La forma de representar el árbol en GPU es con una textura lineal. Esta textura que contiene los nodos toma el nombre de *node pool*. Cada téxel (píxel 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, cada téxel es un puntero al hijo correspondiente. Los primeros 4 téxeles representan una subdivisión con valor de *z* menor mientras que los últimos 4 representan una con valor de *z* mayor. Dentro de cada grupo de 4, los primeros 2 son un valor menor de *y* y los últimos uno mayor. Dentro de los grupos de 2, el primero es menor *x* y el último mayor. Si un téxel tiene el valor 0, entonces ese hijo del nodo no existe. Si un téxel

tiene un valor $x \neq 0$, entonces en la posición $x * 8$ comienza un nuevo nodo, que termina en $x * 8 + 7$.

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. Si otro valor quiere ser almacenado, por ejemplo la irradiancia, entonces 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.

Cada nodo tiene asociado un brick, que es identificado únicamente por su índice. Dado que los bricks existen en una textura 3D, la manera de identificarlos es con un vector de \mathbb{R}^3 . Para esto, se usa una función que convierte cada índice de \mathbb{R} en un vector de \mathbb{R}^3 .

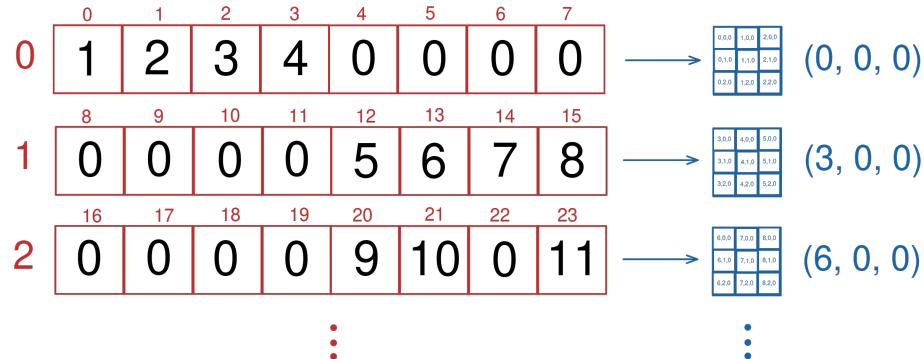


Figura 4.2: Ejemplo node pool y brick pool

En la figura 4.2 se muestra un ejemplo de *node pool* y *brick pool*. En este ejemplo, vemos que la *node pool* está pintada de rojo, mientras que la *brick pool* está pintada de azul. Cada nodo de la *node pool* está numerado en el margen izquierdo (0, 1, 2), está formado por 8 téxeles, los pequeños números arriba de cada caja. Los números dentro de cada téxel de un nodo son los índices del nodo hijo, que debe ser multiplicado por 8 para conseguir el téxel correspondiente.

Al almacenar los valores en una textura 3D, conseguimos interpolación trilineal acelerada por el hardware de la GPU a la hora de tomar una muestra dentro de un brick. Esto mejora la calidad de imagen.

4.2.2. 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

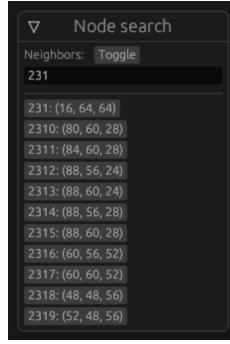


Figura 4.3: Menú de nodos

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 más útiles para depurar es uno que 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 4.3. En el menú se muestran todos los nodos de la node pool con su índice (0, 1, 2, ...) y sus coordenadas dentro de la escena (entre 0 y 255). 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.

Este menú se puede filtrar, tanto por índice como por coordenadas, y se pueden mostrar los vecinos de cada nodo.

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

4.3. CLI

Este paquete 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 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 específicamente 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 vóxeles 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 referencien, 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 predeterminados

Para poder iterar más rápido, se crearon archivos de 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. Son muy útiles cuando se está viendo algo en específico, y es necesario cambiar el código y recomilar, porque inmediatamente se vuelve al mismo lugar.

Capítulo 5

Experimentación

Para analizar el rendimiento de la implementación, se realizaron experimentos en distintas tarjetas gráficas. La escena utilizada se muestra en la figura 5.1. Esta escena presenta luz indirecta difusa, luz indirecta especular y sombras suaves.

Los experimentos varian la cantidad de véxeles por dimensión, y registran los cuadros por segundo alcanzados (FPS) y el tiempo de construcción del *octree* en segundos. Para conseguir los FPS se corrió el programa por un minuto, tomando una muestra de este valor cada segundo, luego se promediaron. Para el tiempo de construcción del *octree*, se construyó 50 veces y se promedió el tiempo. Para que el cache no afectara estas ejecuciones en serie, se realizaron en procesos separados.

En las tablas 5.1, 5.2 y 5.3 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. Una observación es la falta de optimización, 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 menores que en las del 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.

Otra 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 primera 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. En la construcción, probablemente lo que más este beneficiando a la segunda tarjeta es la cercanía física entre CPU y GPU, que implica una comunicación más rápida.

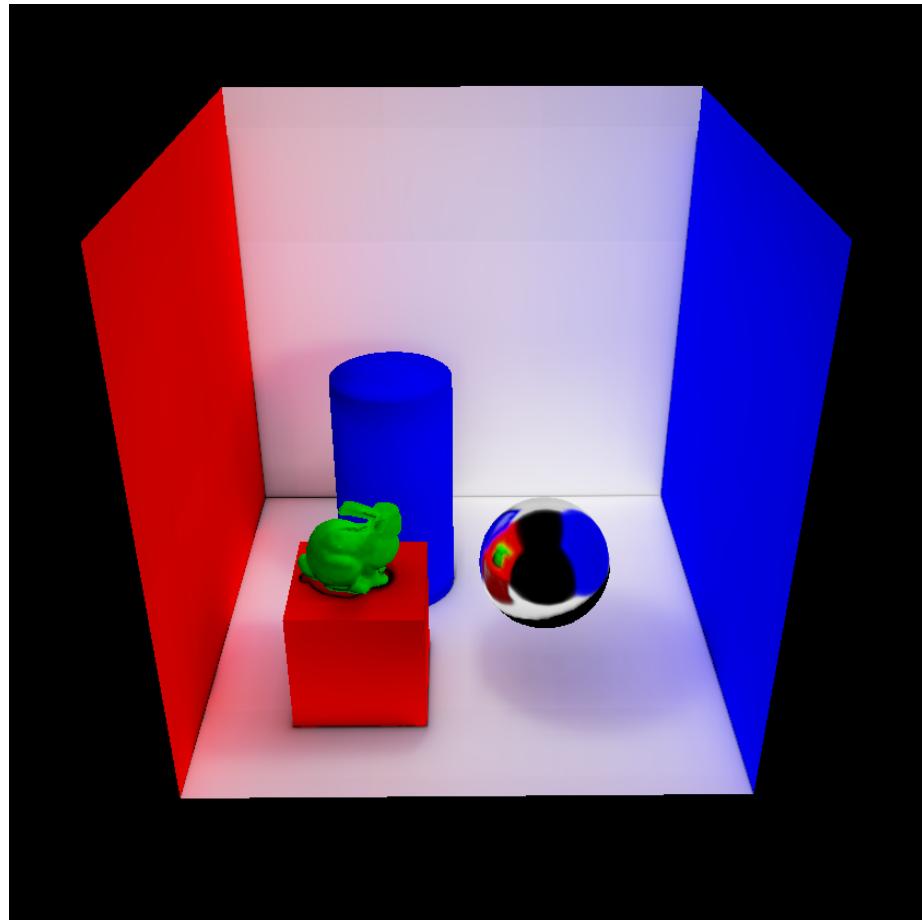


Figura 5.1: Escena con iluminación indirecta mediante *voxel cone tracing*

Vóxeles	FPS	Construcción del <i>octree</i> (s)
256	14,470	1,451
512	11,301	1,490
1024	9,215	1,656

Cuadro 5.1: Experimentos para una Intel Mesa ADL GT2

Vóxeles	FPS	Construcción del <i>octree</i> (s)
256	12,632	2,461
512	10,363	2,780
1024	8,825	3,245

Cuadro 5.2: Experimentos para una Nvidia GTX 1660 Ti de laptop

Vóxeles	FPS	Construcción del <i>octree</i> (s)
256	141,570	1,949
512	141,496	1,936
1024	141,406	1,950

Cuadro 5.3: Experimentos para una RTX 4070

Capítulo 6

Conclusiones y Trabajo Futuro

Luego de todo el trabajo, 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.

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. De esto se concluye que no fue dedicado suficiente tiempo a la optimización de la implementación.

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:

- Optimizaciones varias

Como fue visto en los experimentos, la implementación realizada, si bien funciona, no está optimizada. Varias optimizaciones fueron identificadas pero no se pudieron alcanzar debido a falta de tiempo. Para que la comparación de eficiencia sea más fiel, habría que implementar estas optimizaciones. Esto incluye cosas como: usar estructuras especiales para reducir

la cantidad de hilos lanzados para algunos *shaders*, reducir llamadas innecesarias, reducir comunicación entre CPU y GPU, entre otros.

- **Objetos dinámicos**

Más allá de la capacidad de renderizar imágenes buenas de iluminación global en tiempo real, es posible tener objetos dinámicos en la escena, que al moverse subdividen el *octree* nuevamente. La estructura fue implementada con esto en mente, pero no se llegó por temas de tiempo. Dicho esto, no debería ser extremadamente complejo agregar esta funcionalidad en un trabajo futuro.

- **Pasar a Vulkan**

Esto se explicó que no se pudo lograr por temas de experiencia en la tecnología. Pasar el programa a Vulkan sería un buen experimento para aprenderla. También sería un buen experimento ver 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**

Por último, el menú fue una funcionalidad muy útil para entender el funcionamiento de todos los pasos involucrados. Dado más tiempo, hubiera sido de interés extender el mismo incorporando funcionalidades como: habilitar y deshabilitar etapas para ver cómo afectan a la imagen, poder elegir objetos con el mouse para facilitar explorar sus materiales y nodos, entre otros. Esto sería particularmente útil para la exploración del programa por programadores interesados en aprender cómo funciona todo.

Bibliografía

- [Ake⁺18] Tomas Akenine-Möller y col. *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⁺11] Cyril Crassin y col. “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⁺09] Rui Wang y col. “An Efficient GPU-based Approach for Interactive Global Illumination”. En: *ACM Transactions on Graphics* 28.91 (2009), págs. 1-8.
- [Whi80] Turner Whitted. “An improved illumination model for shaded display”. En: *Communications of the ACM* 23.6 (1980), págs. 343-349.