



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

Supervisores

Eduardo Fernández
Jose Pedro Aguerre

Montevideo, 17 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/franciscoaguirreperez/voxel-cone-tracing>

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

Índice general

1. Introducción	1
1.1. Motivación	2
1.2. 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	8
2.5. Trazado de rayos (<i>ray tracing</i>)	9
2.6. Trazado de conos	10
2.7. Photon Mapping	11
2.8. Vóxeles	12
2.9. Octrees	14
2.10. Ducto gráfico	16
2.11. Renderizado diferido	17
3. Voxel cone tracing	19
3.1. Voxelización	20
3.1.1. Rasterización conservativa	21
3.2. Sparse Voxel Octree	21
3.2.1. Nodos y bricks	22
3.2.2. Construcción	22
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. Cone tracing	29
3.4. Efectos	29
3.4.1. Oclusión ambiental	29
3.4.2. Conos de sombra	30
3.4.3. Iluminación indirecta	30

4. Implementación	31
4.1. Engine	31
4.2. Core	31
4.2.1. Representación del SVO	31
4.2.2. Vóxels anisotrópicos	34
4.2.3. Nodos frontera	34
4.2.4. Menú	35
4.3. Cli	35
4.3.1. Archivos de escena	35
4.4. Herramientas	36
4.4.1. Rust	36
4.4.2. OpenGL	36
5. Experimentación	39
6. Conclusiones y Trabajo Futuro	41

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 varias técnicas para resolver el problema de la iluminación, como path tracing y photon mapping, pero estas fallan en conseguir tiempos interactivos con el hardware disponible. Se han hecho varias optimizaciones a lo largo de los años para lograr esto, como por ejemplo simplificaciones en la geometría, el uso de estructuras jerárquicas, clustering, entre otras, pero ninguna de estas logró que funcionaran en tiempo real, y sufren de artefactos como el ruido.

El objetivo de este trabajo es aprender sobre y crear una implementación de un algoritmo de iluminación global que logra correr en tiempo real sin necesidad de hardware específico, *voxel cone tracing*. Este algoritmo fue propuesto por Crassin et al en 2011 [Cra+11]. No sufre de problemas de ruido y provee una buena calidad de imágenes con un rendimiento casi independiente de la complejidad de la escena, debido a que la geometría en sí no se usa en los cálculos de luz, si no una aproximación de véxeles. 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 algoritmo surge en un contexto en el que el problema de iluminación global en tiempo real no tenía muchas soluciones, y la demanda para la misma estaba creciendo. Hace uso de varias técnicas que estaban disponibles, la representación de véxeles de la geometría, el trazado de conos y el uso de estructuras jerárquicas de datos y pre-filtrado.

Este trabajo se desarrolla 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.

1.1. Motivación

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.

1.2. 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 las decisiones tomadas respecto al desarrollo y proporciona pseudocódigo para entender el algoritmo en mayor detalle. El capítulo 5 muestra los resultados de los experimentos realizados con la herramienta implementada que se presentan en forma de gráficas, tablas e imágenes. Finalmente, el capítulo 6 resume los resultados y conclusiones de este trabajo y presenta las features y arreglos que se podrían implementar a futuro y por qué no entraron en el alcance de este proyecto.

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 aproximaciones que se puedan hacer, 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 Empedocles 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

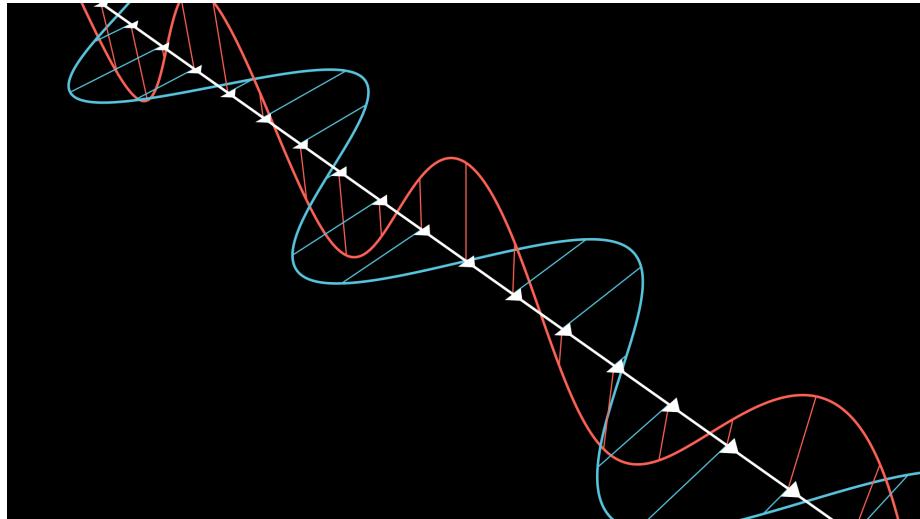


Figura 2.1: Representación de una onda electromagnética propagando por el espacio

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, centrandose 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.

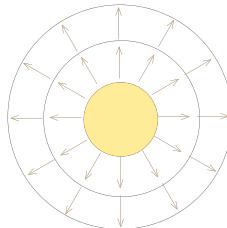


Figura 2.2: Flujo radiante en una luz puntual

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.

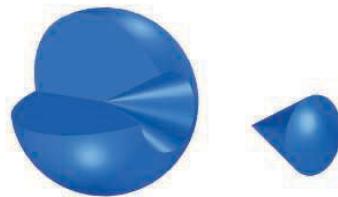


Figura 2.3: Un cono con un ángulo sólido de 1 estereorradián removido de una esfera. Fuente: [Ake⁺18]

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

Llegamos a la unidad radiométrica más importante, la **radiancia**. La radiancia es la cantidad infinitesimal de flujo radiante contenida en un rayo de luz entrante o saliente por unidad de superficie y por unidad de ángulo sólido. Se considera al rayo como un cono “infinitesimal” con su vértice en un punto en una superficie. Es la cantidad de flujo radiante con respecto a tanto área como ángulo sólido, se denota como L y se mide en watts por metro cuadrado por estereorradián. Si el valor X de radiancia emitida por ese rayo se extendiese a todo un estereorradián y a toda una unidad de superficie, entonces se estarían emitiendo X watts.

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

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.

La radiometría trata únicamente con cantidades físicas de la luz. Un campo relacionado, la fotometría, es parecido la radiometría, pero pesa todas sus cantidades por la sensibilidad del ojo humano. 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

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

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.

Sus ventajas son su simplicidad y su eficiencia. Su mayor desventaja es que no abarca fenómenos como la iluminación indirecta, 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, 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.

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.

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.

Esta 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].

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 reflexión, 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 2.4 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. Sucede al representar curvas. Se puede ver en la figura 2.4.

Kajiya propuso en 1986[Kaj86], junto con la ecuación de renderizado, un método llamado *path tracing*, una extensión de *ray tracing* que utiliza integración

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

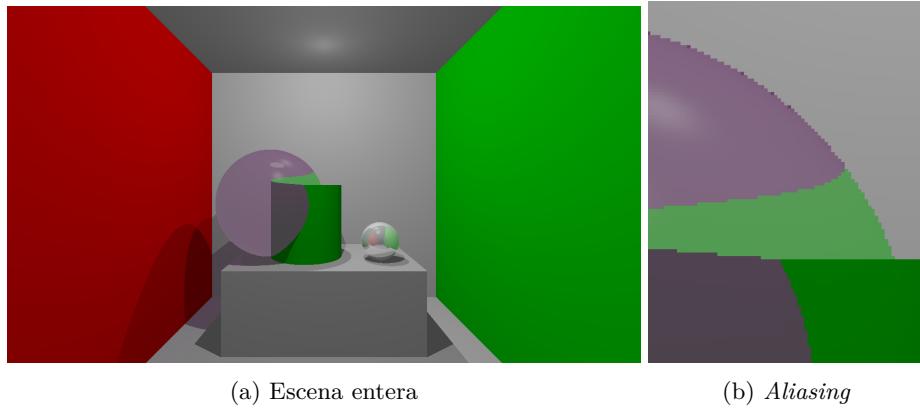


Figura 2.4: Escena renderizada con el trazado de rayos de Whitted. Implementación propia.

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.

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

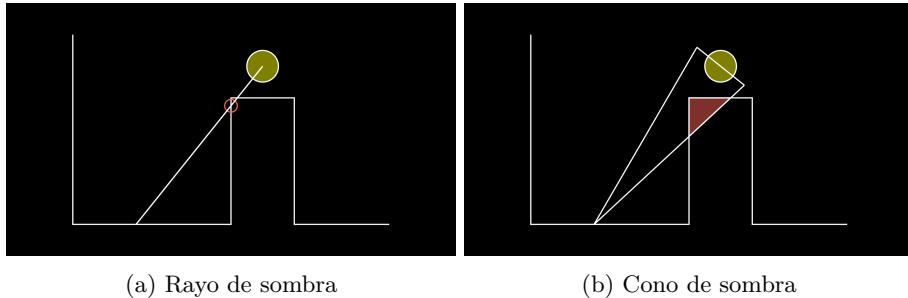


Figura 2.5: Los conos de sombra no solo devuelven si hay intersección, también devuelven el porcentaje de área de la intersección

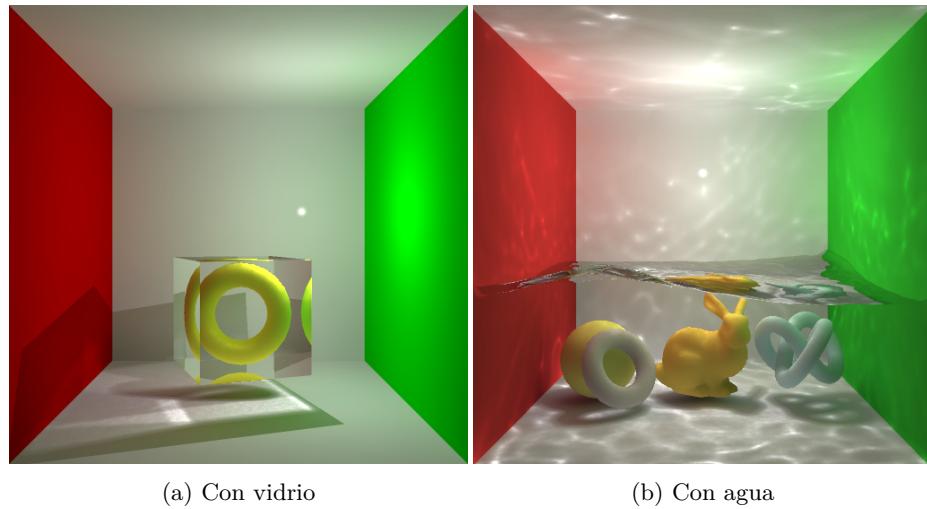
de apertura, lo cual agrega un grosor no despreciable. Usando este grosor, los conos son capaz de no solo probar la existencia de intersecciones, sino también calcular el porcentaje de área de la intersección. Este método fue propuesto para solucionar el *aliasing* presente en el trazado de rayos, como 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 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 2.5 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.

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

Figura 2.6: Cáusticas. Fuente: [Wan⁺⁰⁹]

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 espejular o transparente concentra la luz en una superficie difusa. En la figura 2.6 se puede ver este efecto.

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 2.7.

2.8. Vóxeles

Un vóxel es el equivalente de un píxel en el espacio 3D. Así como un píxel es un elemento de imagen, *picture element*, un vóxel es un elemento de volumen, *volume element* [Ake⁺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.

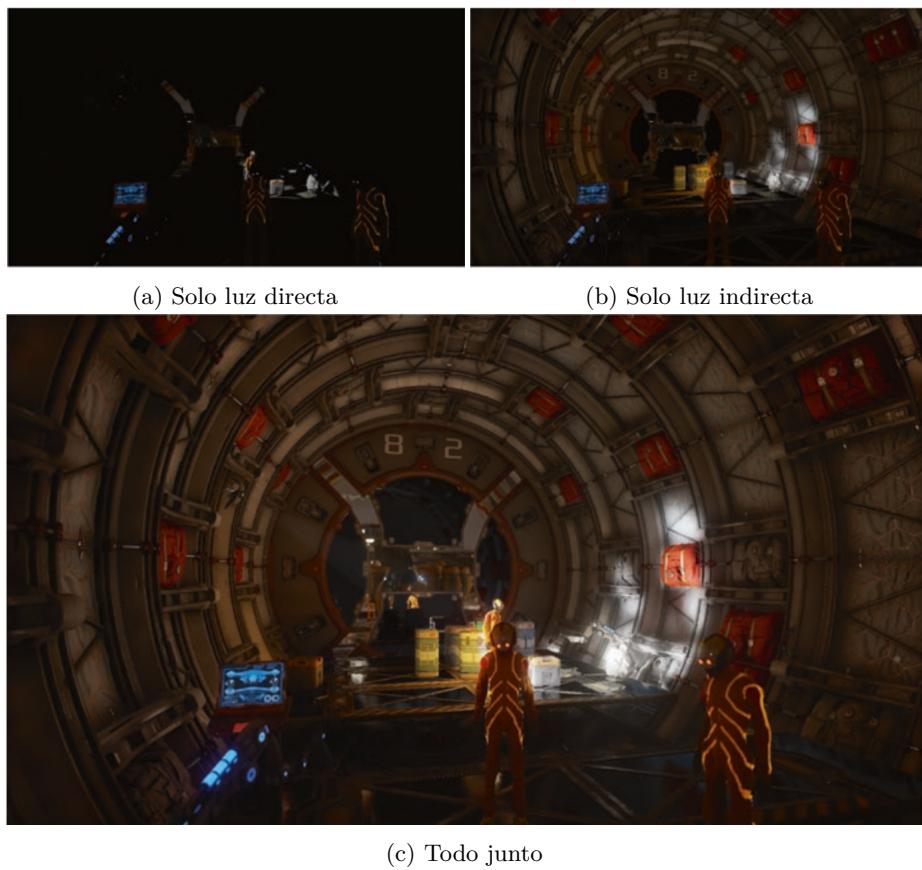
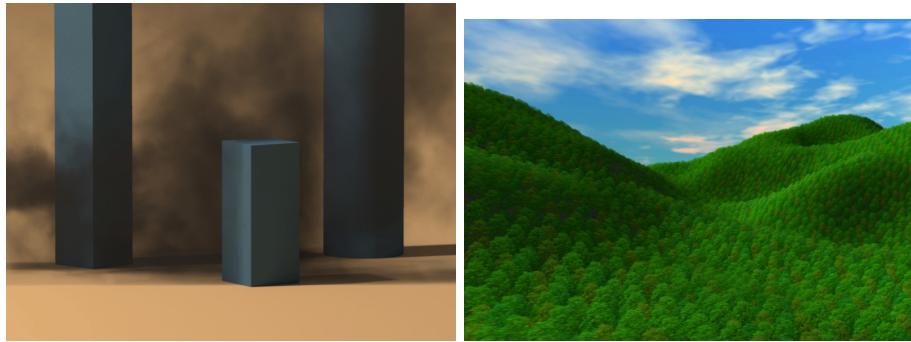


Figura 2.7: Photon mapping en tiempo real. Fuente: [SA19]



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

Figura 2.8: Vóxeles para renderizar humo y vegetación

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 2.8.

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, en el que solo se subdividen las regiones que necesitan mayor detalle.

2.9. Octrees

Un *octree* es una estructura de datos espaciales [BD02]. Es un árbol formado por nodos, en los que cada nodo interno (no hoja) tiene exactamente 8 hijos

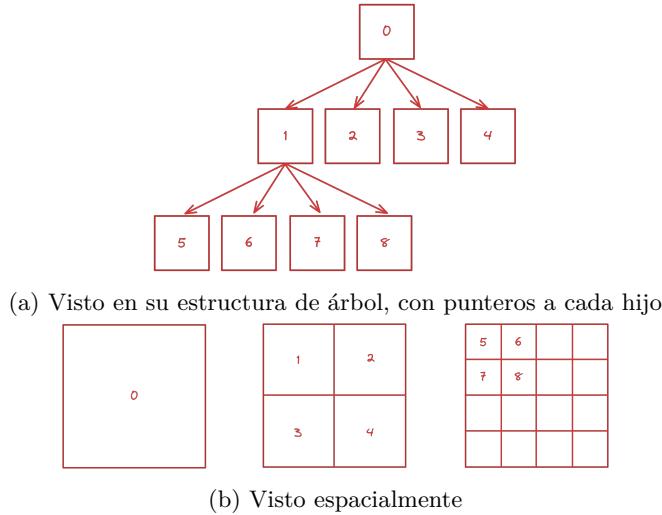


Figura 2.9: Quadtree

[Ake⁺18]. Se utilizan 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 2.9 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 2.10 [Ake⁺18].

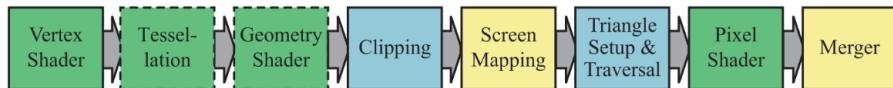


Figura 2.10: Ducto de raster. Fuente: [Ake⁺18]

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.

Los pasos programables son el *vertex shader* (*shader* de vértices), *geometry shader* (geometría), y *fragment* o *pixel shader* (fragmentos o píxeles). 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 ejecuta un hilo por cada píxel de la imagen que se quiere generar.

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 un test de profundidad.

En el renderizado diferido, se procesan todos los vértices del programa solo para la etapa del *vertex shader*. Se guarda toda la información necesaria para etapas posteriores en texturas llamadas *geometry buffers*. Al hacer esto, se ahorrarán los cálculos para todos los vértices que son desechados después del *vertex shader*. 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, 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 aumenta, el renderizado diferido se vuelve cada vez una opción mejor para mejorar la eficiencia. 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* ???. Reduce los costos asociados a estos algoritmos clásicos de trazado de rayos utilizando véxeles ??, *octrees* dispersos 2.9 y conos ??.

Para lograr el renderizado en tiempo real, *voxel cone tracing* realiza aproximaciones. Una de ellas es el uso del ducto de raster visto en la sección 2.10. Los cálculos de luz se realizan en el *fragment shader* del ducto de raster. Esto difiere del trazado de rayos clásico, que renderiza toda la escena puramente mediante intersecciones entre rayos y geometría.

También se utiliza el renderizado diferido, visto en la sección 2.11, para reducir la cantidad de vértices sobre los que se realizan cálculos.

La principal aproximación que realiza el algoritmo es la de

El trazado de conos se usa para agregar efectos de iluminación global a una escena rasterizada utilizando mallas poligonales para sus superficies.

Para reducir la cantidad de cálculos a realizar, se utiliza una aproximación de la escena, en la forma de véxeles almacenados en el *octree*. Al calcular la radiancia de la escena que incide en un punto, se utiliza un nivel de detalle menor de cada parte de la escena, cuanto más alejada se encuentre del punto. Debido a esto, en lugar de trazar rayos, que mantienen el mismo tamaño a medida que se aleja de su punto de origen, se trazan conos, cuyo diámetro crece a medida que se aleja de su origen. Mientras más lejos el punto muestrado del origen del cono, más grande el diámetro del mismo. Este tamaño se usa para determinar qué nivel de detalle será utilizado para esta región de la escena. Este nivel de detalle se almacena en un *octree* disperso.

Para generar esta estructura, se voxeliza la escena y se usan los véxeles como base para el *octree*. Luego, se promedian los valores de los véxeles en el último nivel del *octree* a los niveles superiores, de tal manera que cada nodo aproxime la región del espacio que le corresponde.

El algoritmo se puede dividir en 3 grandes etapas:

1. Voxelización de la escena
2. Construcción del *octree* disperso
3. Trazado de conos

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 véxels. La cantidad de véxels 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 que escribió 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 la rendering pipeline de OpenGL, se voxelizan todos los triángulos de la geometría de la escena. Esto genera una lista de **fragmentos de véxel**, con atributos como posición y color. Cada uno de estos fragmentos será usado para construir el SVO y terminará siendo un véxel 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.

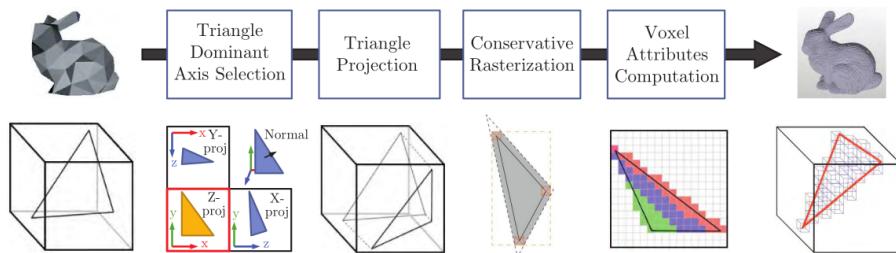


Figura 3.1: Ducto de voxelización. Fuente: [CG12]

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.

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

Durante la rasterización, cada triángulo genera un conjunto de fragmentos 2D. Estos fragmentos pueden intersecar con a lo sumo 3 vóxeles de la grilla. A su vez, debido a la elección de la dimensión dominante de la normal para la proyección, solo pueden intersecar con 3 vóxeles en la profundidad también. Entonces, por cada fragmento 2D, los vóxeles que intersecaron con el triángulo se calculan en el *fragment shader*, basándose en la posición, la profundidad y las derivadas en espacio de pantalla.

Esta información se usa para generar una lista de **fragmentos de vóxel**. Estos son una generalización en 3D de los fragmentos 2D y corresponden a un vóxel que interseca con un triángulo. Cada fragmento de vóxel tiene una coordenada que lo identifica dentro de la grilla 3D de vóxeles, así como color y posición.

3.1.1. Rasterización conservativa

El método descrito anteriormente a veces no crea fragmentos para elementos muy finos, como un asta de bandera. Esto pasa porque 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 triángulo (en cualquier punto) también toca su centro. Esto se hace alargando las aristas del triángulo hacia afuera. Hay fragmentos que resultan de sobreestimar la cobertura de este triángulo, los cuales se descartan. Este proceso se muestra en la figura 3.2.

3.2. Sparse Voxel Octree

Para almacenar los fragmentos de vóxeles generados, se usa un *octree* disperso. 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 subdividan 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 fuera denso, cada nivel n tendría 8^n nodos. El nivel 0 tendría 1, el nivel 1 tendría 8, el 2 tendría 64 y así sucesivamente. En este caso, 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 vóxeles. Valores más altos de resolución crean más niveles del *octree* y se asemeja cada vez más la aproximación a la geometría real.

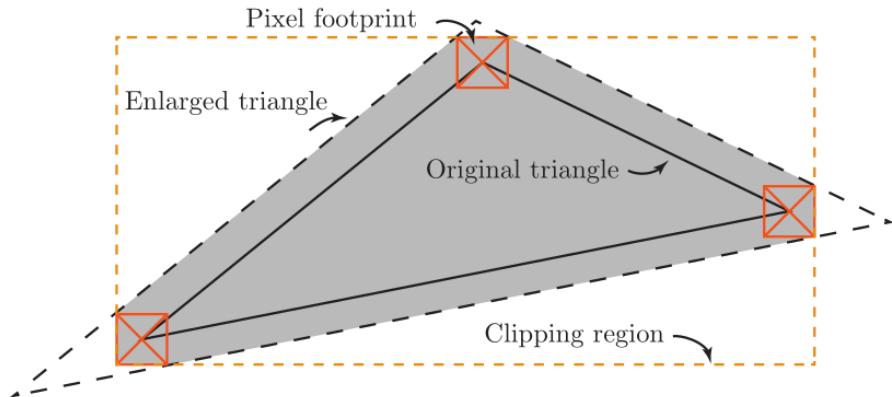


Figura 3.2: Rasterización conservativa. Fuente: [CG12]

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 y bricks

El SVO no solo está compuesto por los nodos del árbol. A su vez, cada nodo tiene asociado un *brick*, una estructura que también representa una región del espacio, pero que almacena valores distintos. Cada nodo del árbol almacena únicamente un puntero a sus, como máximo 8, hijos. Los bricks almacenan los valores necesarios que representan esa región del espacio, por ejemplo el color.

Cada brick está dividido en 27, $3 \times 3 \times 3$, véxeles. Son estos véxeles los que almacenan los valores de la escena. En la figura 4.3 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 para que los bricks puedan obtener valores de sus vecinos, para garantizar que la interpolación dentro de un solo brick toma en cuenta a los vecinos. Esto resulta en una frontera compartida entre vecinos, como se puede ver en la figura 4.4. Esta frontera debe almacenar lo mismo para que la interpolación a la hora de generar la imagen final funcione. Esto se logra mediante un programa llamado *border_transfer*, que se explicará en la sección 3.2.3.

3.2.2. Construcción

Para generar esta estructura se usan los fragmentos de véxeles. 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.

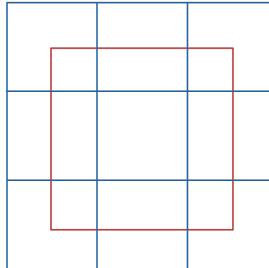


Figura 3.3: Nodo con su brick asociado (en 2D)

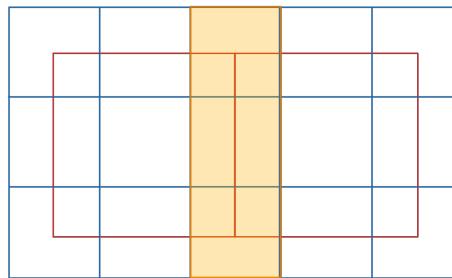


Figura 3.4: Solapamiento entre véxels de bricks de nodos vecinos

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 fragmento de la lista de fragmentos. Dado un fragmento, 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 marcas 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 fragmentos en los bricks de las hojas del árbol, promediando cuando más de uno cae en la misma hoja. Esto último pasa más mientras más triángulos tiene la escena original y menos resolución tiene la grilla de véxels. Las hojas no 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 fragmentos, estos se almacenan únicamente en las esquinas de los bricks. Se almacenan en la esquina más cercana a la posición del fragmento. Luego, se aplica un programa *spread_leaves*, para esparcir 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:

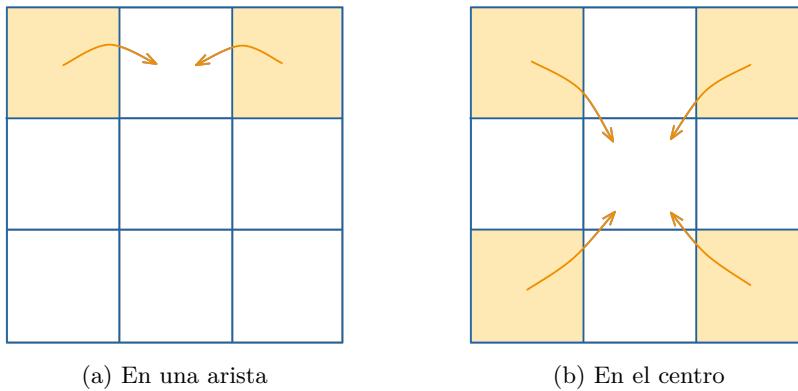


Figura 3.5: Funcionamiento de *spread_leaves* en 2D. Las flechas indican aporte al promedio

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

De esta manera, se esparsen los valores de las esquinas a todo el brick. Este es el algoritmo que expande los voxels generados para llenar los bricks $3 \times 3 \times 3$.

Como se mencionó anteriormente, las fronteras entre bricks son compartidas, corresponden al mismo espacio en la escena. Por lo tanto, los valores almacenados en esos voxels deben ser los mismos. Para lograr esto, una vez que los valores son almacenados en cada brick, se transfieren los valores de la frontera al brick vecino, asegurándose que sea coherente el nivel. En el caso del color y las normales, se guarda en las fronteras de cada brick, el promedio de estos valores.

3.2.3. Border transfer

Recordar que los bricks comparten una frontera con sus vecinos. Luego de *spread_leaves*, esta frontera tiene cosas distintas en cada brick vecino. Es necesario que tengan el mismo valor para cada brick, para garantizar una suave interpolación a la hora de generar la imagen final. 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 voxel 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. Esto permite ahorrar memoria, pero tiene un problema. A la hora de interpolar para generar la imagen final, esta interpolación de efectos logrados por el algoritmo, por ejemplo sombras y reflejos, solo llega hasta el final de los nodos existentes, luego se corta abruptamente. Para que la interpolación continúe y logre difuminar estos efectos, es preciso una capa de nodos extra, nodos frontera, entre la geometría y el espacio vacío. Estos nodos 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 difuminar los bordes de la geometría.

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 no hoja A , lograr que su brick tenga un resumen 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 vértice del brick del padre, usando los bricks de los hijos.

Consideremos un nodo de un nivel mayor al último, 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.

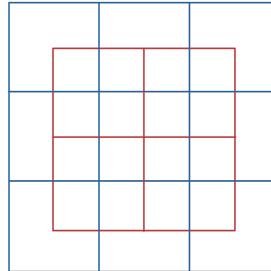


Figura 3.6: Nodo con su brick asociado y sus hijos

Los valores de los vértices se calculan en 4 pasajes distintos:

- Esquinas
- Bordes
- Caras
- Centro

Cada uno de estos pasajes calcula un valor parcial de un tipo de vértice. Es parcial porque para los vértices limítrofes con otro nodo, este valor tiene que

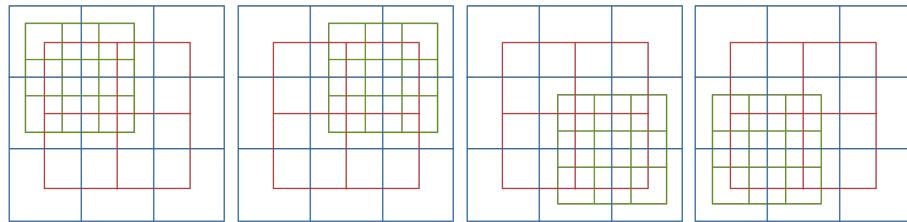


Figura 3.7: Bricks de todos los hijos del nodo

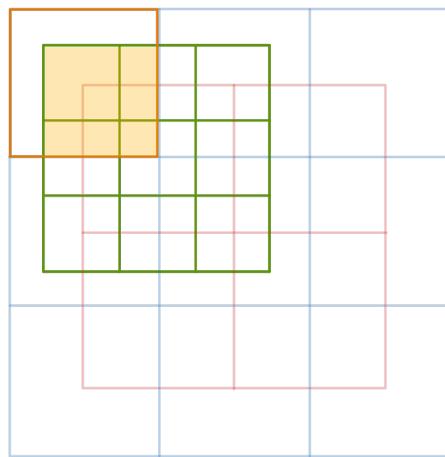


Figura 3.8: Filtrado para un voxel esquina

luego ser agregado con el de los vecinos. Luego este valor parcial se utiliza *border_transfer* para conseguir los valores totales.

El cálculo para cada pasaje es el mismo para todos los véxeles dentro de su grupo, así que se mostrará cómo se calcula solamente para un ejemplo tipo de cada grupo (esquinas, bordes, caras, centro). En el caso 2D, no existe el voxel centro.

Dado el voxel superior izquierdo de la figura 3.8, se considera solo el brick del hijo superior izquierdo del nodo. De ese brick, se consideran los véxeles iluminados. El valor final del voxel del padre se calcula promediando los valores de los véxeles del hijo, pesados por el porcentaje de solapamiento. Esto resulta en un kernel gaussiano.

De la misma manera se calculan los véxeles 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 3.9 y 3.10.

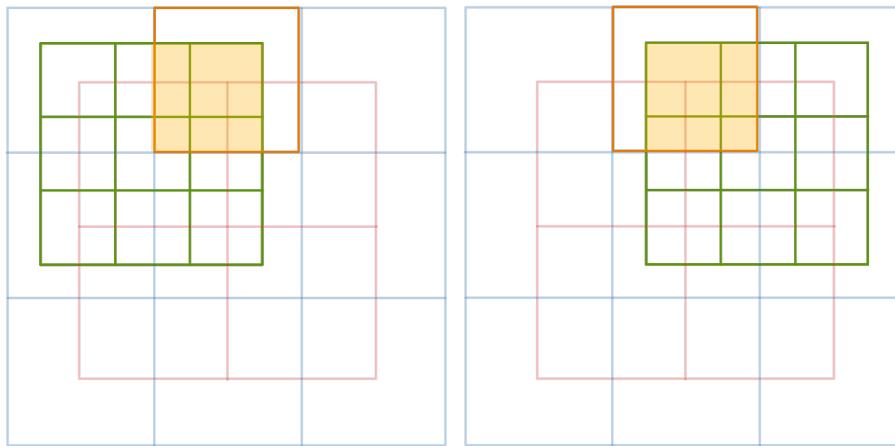


Figura 3.9: Filtrado para un vóxel borde

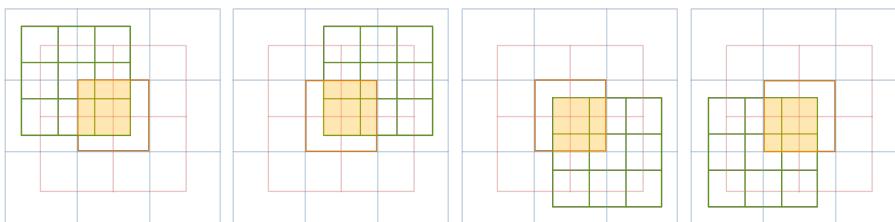


Figura 3.10: Filtrado para un vóxel cara

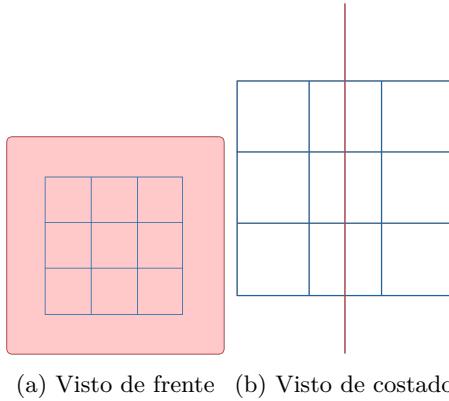


Figura 3.11: Brick que contiene una pared fina

3.2.6. Filtrado anisotrópico

El filtrado descrito en la sección anterior resulta en véxeles cuyos valores son los mismos vistos en todas las direcciones. A pesar de que el voxel es un cubo, almacena solo un valor, en lugar de tener distintos valores según de dónde se mira. Esto es muy útil a la hora de representar un atributo como el color en una escena 3D.

Consideremos un voxel de un nodo alto en el árbol, que representa una gran región del espacio. Esta región contiene únicamente una pared que pasa por su centro, y el resto es todo espacio vacío. En este caso, entonces visto de la dirección paralela a la normal de la pared, el voxel se verá del color de la pared y será completamente opaco, mientras que visto de la dirección perpendicular a la normal de la pared, el voxel será completamente transparente. Esta situación se muestra en la figura 3.11.

Este comportamiento no ocurre con el filtrado anterior, que se conoce como isotrópico, significando “igual en todas las direcciones del espacio”. La solución es mejorar el filtrado y hacerlo 6 veces por voxel, uno por cada dirección alineada con los ejes, X, -X, Y, -Y, Z, -Z. Luego, se puede usar la dirección de vista para interpolar usando las tres direcciones más cercanas. Este comportamiento es anisotrópico, no es igual para todas las direcciones.

Dada una dirección, por ejemplo, de izquierda a derecha, se encuentra una base de véxeles y se usan para calcular **valores direccionales**. En la figura 3.12 se pueden ver todos los valores direccionales que deben ser calculados para un brick en 2D, para todas las direcciones. Para cada voxel de la base, 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 ese voxel base. Este algoritmo es el que soluciona situaciones como la de la pared mencionada anteriormente.

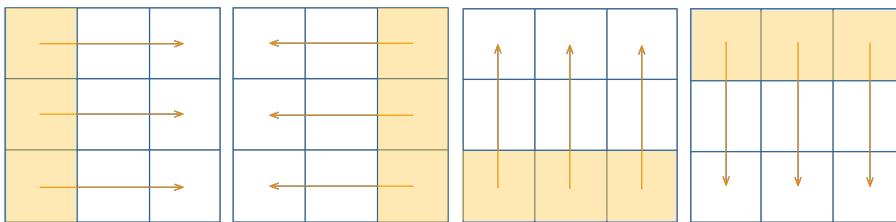


Figura 3.12: Filtrado anisotrópico en todas las direcciones

3.3. Cone tracing

La entrada al algoritmo de cone tracing no son los vértices de la escena, si no los *geometry buffers* que contienen los valores de la escena ya habiendo descartado los vértices fuera de vista, como se explicó en ??.

El algoritmo de cone tracing es ejecutado para cada píxel de los geometry buffers para calcular un color que se escribe en otra textura, la que finalmente se muestra sobre un quad que cubre toda la pantalla.

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. 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 voxel 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 en si es simple dado todos los pasos anteriores.

3.4. Efectos

Dependiendo de la cantidad, el tamaño, la dirección y el criterio de parada de los conos, se pueden lograr varios efectos con cone tracing.

3.4.1. 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 vértices 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.

3.4.2. 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 promediárlas.

3.4.3. Iluminación indirecta

Para producir efectos de iluminación indirecta, se toma un enfoque parecido al de *photon mapping*, y se lanzan fotones a partir de la fuente de luz.

Para hacer esto, se renderiza la escena desde cada fuente de luz, con esto se genera un mapa de luz. Todos los mapas de luz se guardan en texturas, donde se almacena, para cada texel de la textura, la posición global de la geometría. Luego, cada una de estas texturas se recorren texel a texel y se recorre el octree para guardar en el brick correspondiente un fotón. Con esto, se almacenan en todas las hojas del octree fotones. Estos fotones se transfieren a través de las fronteras (esta vez sumando y no promediando) y se filtran al igual que los otros atributos.

Con esta información de iluminación, se pueden generar reflejos.

Reflejos difusos

Para reflejos difusos, 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 vértices 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 luz.

Reflejos especulares

Para los reflejos especulares, 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.

Capítulo 4

Implementación

La implementación utilizó el lenguaje de programación Rust ([Rus23]) y la API de gráficos OpenGL ([Khr22]).

La arquitectura principal consta de tres paquetes: *cli*, *core* y *engine*. *Engine* contiene todas las abstracciones sobre OpenGL, provee tipos como *Transform*, *Light*, *Camera* que permiten manipular objetos en el espacio 3D. *Core* contiene todos los algoritmos de voxel cone tracing que hemos mencionado: voxelización, construcción del SVO, filtrado, actualización y el trazado de conos en si. El paquete *cli* es el punto de entrada a la aplicación y utiliza las funcionalidades expuestas por *engine* y *core* para crear un ambiente 3D con los efectos logrados por voxel cone tracing. Estos tres paquetes se muestran en la figura 4.1.

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

4.1. Engine

4.2. Core

4.2.1. Representación del SVO

La forma de representar este árbol es con una textura lineal en GPU, conocida como la node pool. Cada texel (píxel de textura) de esta textura es un puntero a otro nodo. Se toma la convención de que cada grupo de 8 texels es un nodo, cada texel es un puntero al hijo correspondiente. Los primeros 4 texels 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, primero es menor x , último mayor. Si un texel tiene el valor 0, entonces ese hijo del nodo no existe. Si un texel tiene un valor $x! = 0$, entonces en la posición $x * 8$ comienza un nuevo nodo, que termina en $x * 8 + 7$.

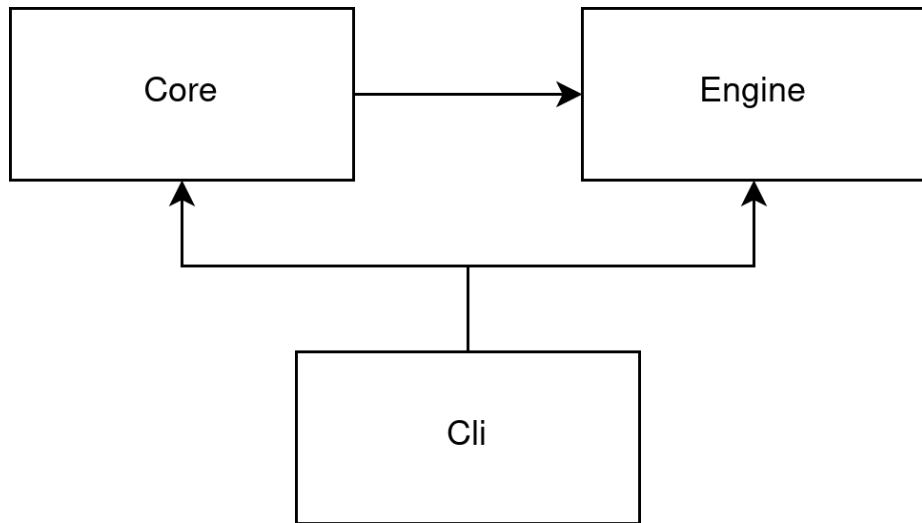


Figura 4.1: Arquitectura de la implementación

Cada nodo tiene asociado un brick. Los bricks son texturas 3D de 3^3 texels que buscan aproximar la región de la escena que representa el nodo. Se almacenan en una gran textura 3D llamada la brick pool, con lo cual cada brick es una región de 3^3 texels de esta gran textura. Mientras los nodos tienen únicamente punteros a sus hijos, los bricks son los que almacenan los atributos, como el color y la normal. Cada texel dentro de un brick se llama véxel, porque además de ser un píxel de textura, también es un píxel de volumen. Es en estos véxels que se almacenan los atributos con los que se va a trabajar.

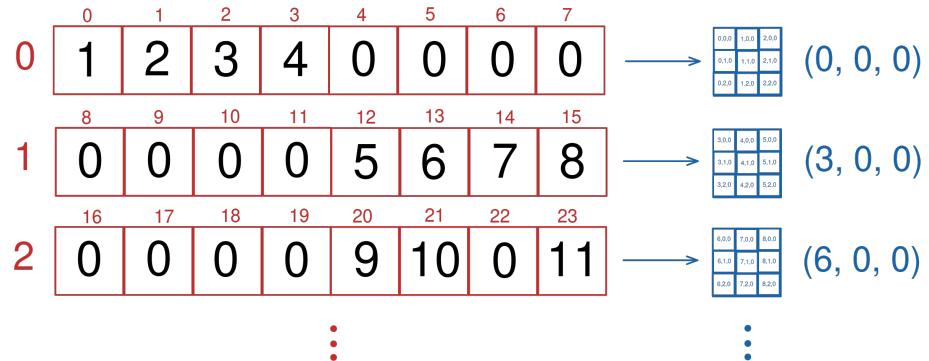


Figura 4.2: Ejemplo node pool y brick pool

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 texels (los pequeños números arriba de cada caja). Los números dentro de cada texel de un nodo son los índices del nodo (que debe ser multiplicado por 8 para conseguir el texel correspondiente) hijo. Los texels que tienen 0 indican que esa subdivisión no tiene un hijo. Acá es donde se ve que la estructura es esparza.

Cada brick es una sección de 3^3 texels de una gran textura 3D. 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 .

Cada nodo podría tener asociado solo un valor escalar para el atributo que se quiera guardar en él, en lugar de asociarles una textura 3D (el brick). Sin embargo, usando texturas podemos conseguir los beneficios de la interpolación trilineal provista por el hardware de la GPU al tomar un sample dentro del brick. Si el sample cae entre dos texels, se interpola. Esto permite mejorar la calidad de la imagen, como veremos en el capítulo 5.

La node pool y brick pool son la forma en la que se almacenan en memoria los nodos y bricks respectivamente. Si bien los nodos y los bricks se almacenan usando indices, la región de la escena que representan es totalmente distinta. Los nodos representan secciones cada vez más chicas de la escena, que resultan de subdividir al nodo padre, ocupando el nodo con indice 0 toda la escena. Los bricks se ubican en la escena sobre sus nodos asociados, y se extienden un poco más hacia cada dirección. Esto es porque los vóxels de los bricks están centrados en los vértices de los nodos. En la figura 4.3 se muestra esta disposición espacial.

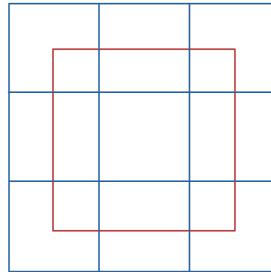


Figura 4.3: Nodo con su brick asociado (en 2D)

En la figura 4.3 también se ve la razón por la que los bricks son de tamaño 3^3 , para poder cubrir el espacio del nodo pero también extenderse un poco más allá de su límite. Esto es necesario para poder conseguir valores de más allá del espacio del nodo y traerlos al brick, así luego, al interpolar los valores dentro del brick, se están teniendo en cuenta valores de una región más grande. El resultado es que dos nodos vecinos comparten una frontera de vóxels de sus bricks asociados. Esto se muestra en la figura 4.4. Dado que estos vóxels representan la misma región del espacio, es necesario que sus valores sean los mismos. Si bien en memoria son dos entidades distintas que existen en distintos lugares,

espacialmente son las mismas, con lo cual debe asegurarse que sus valores sean iguales.

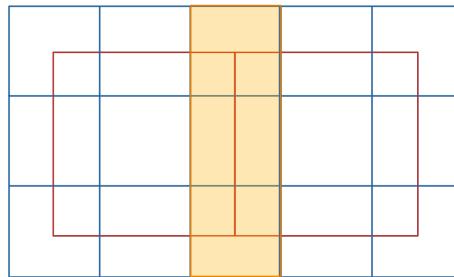


Figura 4.4: Solapamiento entre vóxels de bricks de nodos vecinos

4.2.2. Vóxels anisotrópicos

El cálculo del filtrado anisotrópico requiere conocimiento de los bricks vecinos a la hora de calcular el valor de un vóxel limítrofe. El shader que realiza el filtrado anisotrópico ejecuta un hilo por brick, con lo cual no se tiene acceso a los bricks vecinos.

Para resolver esto, se carga el vecino en la dirección de filtrado para poder completar los valores direccionales. Luego se usa un shader que promedia los valores direccionales con los vecinos de las otras dos direcciones.

4.2.3. Nodos frontera

El procedimiento para producir los nodos frontera reutiliza lo ya existente para generar nodos dado una lista de voxel fragments³, en lugar de modificar directamente con la node pool¹³ ya existente. Al proceso de construcción del octree se le agregan dos pasos:

- Construir lista de voxel fragments
- Construir nodos del octree a partir de la lista de voxel fragments
- NUEVO: Construir una lista de voxel fragments frontera a partir de la lista original de voxel frags
- NUEVO: Agregar nodos frontera al octree usando la lista de voxel frags frontera

Se toma la lista de voxel frags que fue generada a partir de la geometría de la escena y se hace pasar por un nuevo shader. Este nuevo shader toma esta lista de voxel frags y crea una nueva lista de voxel frags “ficticia”, porque no son parte de la geometría. Por cada voxel fragment de la lista original, se recorre el octree y, por cada una de las seis direcciones principales (X, -X, Y,

$(-Y, Z, -Z)$, se busca si tiene un nodo vecino. En el caso que no lo tenga, se agrega a la nueva lista un voxel fragment cuya posición es la del original desplazada hacia la dirección considerada, para que al crearse el nodo en la estructura, este tome el lugar del vecino faltante.

Aprovechar este shader logra que solo sea necesario crear nodos para el último nivel y se subdividen nodos de niveles anteriores automáticamente a medida que se desciende por el árbol.

Una vez se obtiene la nueva lista de voxel fragments, esta se hace pasar por el mismo shader que convierte voxel fragments en nodos del octree. Estos nuevos nodos se agregan al final del buffer de la estructura. Dado que los nodos nuevos están en distintos niveles, se mantiene una lista de índices distinta para mantener de dónde a dónde va un nivel N en el buffer.

4.2.4. 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.

La herramienta más importante del menú a la hora de depurar fue una ventana que muestra todos los nodos de la node pool. Esta se puede ver en la figura 4.5. En el menú se muestran todos los nodos de la node pool con su índice $(0, 1, 2, \dots)$ 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 tener muchos nodos mostrándose a la vez.

Este menú se puede filtrar, tanto por índice como por coordenadas. También se pueden mostrar los vecinos de cada nodo y los bricks, que para no llenar la pantalla mucho, se pueden mostrar individualmente cada capa del brick, con $z = 0, 1, 2$.

También se usó la interfaz gráfica para reportar los FPS que fueron usados en el capítulo 5 y mostrar otros valores en pantalla, relevantes para el algoritmo.

4.3. Cli

4.3.1. Archivos de escena

Para poder fácilmente cargar distintos modelos y probar el algoritmo en ellos, se creó un formato de archivos de escena. Estos archivos usan extensión RON, por Rusty Object Notation, una notación similar a JSON, JavaScript Object Notation, pero para Rust.

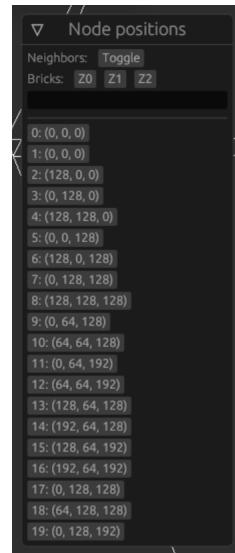


Figura 4.5: Menú de nodos

4.4. Herramientas

4.4.1. Rust

Se eligió el lenguaje de programación Rust para la implementación del algoritmo debido a varias razones:

- Es rápido
- Tiene un manejador de paquetes por defecto con muchas herramientas disponibles
- Tiene una comunidad muy activa y buena documentación

Obviamente, se consideró C++ dado que es el lenguaje más popular para aplicaciones gráficas. El factor que más pesó en la decisión fue la existencia de un manejador de paquetes por defecto. Esto permitió instalar varias dependencias mucho más rápido y cambiarlas a lo largo del ciclo de desarrollo.

4.4.2. OpenGL

Como se mencionó en el capítulo 3, el algoritmo fue mayoritariamente implementado en la GPU. Si bien el lenguaje de programación de la CPU es importante, la elección de la API de gráficos fue, sin duda, la más importante. Se consideraron Vulkan y CUDA pero se terminó optando por OpenGL debido a varios factores:

- Facilidad de desarrollo

- Conocimiento previo
- Facilidad de trabajo con compute shaders

Si bien Vulkan es una API más moderna, el equipo del presente trabajo no contaba con la suficiente experiencia como para utilizarlo adecuadamente, lo cual hubiera llevado a mucho tiempo desperdiciado aprendiendo la herramienta. CUDA es usualmente utilizado para cálculos arbitrarios en la GPU pero debido a la necesidad de también usar el pipeline gráfico, se optó por usar los compute shaders de OpenGL.

Capítulo 5

Experimentación

Puede ser necesario incluir un capítulo de Experimentación, incluyendo las pruebas realizadas (casos de prueba) y los resultados obtenidos con su respectivo análisis, que puede incluir comparaciones.

Capítulo 6

Conclusiones y Trabajo Futuro

Luego de todo el trabajo, se logró pasar de un paper académico a una implementación práctica, open source. Se enfocó en aprender bien lo que se estaba implementando y de documentarlo bien en la implementación para que otros puedan aprenderlo también. Fingxels logró ejecutar oclusión de ambiente a X FPS e iluminación indirecta a Y FPS, mientras que el paper original logró FPS de Z y W respectivamente. Esto se debe a la calidad de la implementación y al hardware moderno utilizado. En conclusión, se logró satisfactoriamente lo que se propuso cumplir y se aprendió mucho haciéndolo.

La principal dificultad que fue encontrada en el transcurso de la tesis, fue la dificultad de implementar un algoritmo enteramente 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 y desmotivó a la hora de hacer la tesis.

El primer año de desarrollo tuvo avance muy lento. Esto se debió más que nada a los problemas con la GPU, pero a su vez a la falta de fijación de objetivos y fechas límite para lograrlos.

Proximos pasos incluyen:

- Optimizaciones varias

Varias optimizaciones mencionadas en el paper no se pudieron alcanzar debido a falta de tiempo. Para poder comparar la verdadera mejora de FPS, habría que implementar como mínimo las mismas optimizaciones. Esto incluye cosas como: usar estructuras especiales para reducir la cantidad de hilos lanzados para algunos shaders, cambiar código para reducir llamadas innecesarias, entre otros.

- Objetos dinámicos

Más allá de la capacidad de renderizar imágenes muy buenas de ilumina-

ción global en tiempo real, es posible tener objetos dinámicos en la escena, que al moverse subdividen el octree nuevamente.

- Pasar a Vulkan

Esto se explicó que no se pudo lograr por temas de experiencia en la tecnología. Pero sería un buen experimento para aprenderla el pasar el programa a Vulkan. También sería un buen experimento ver la mejora en eficiencia que resulta, dado que Vulkan tiene más herramientas de optimización.

- Mejores herramientas de exploración

Poder elegir nodos con el mouse en lugar de seleccionarlos en el menú. 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.
- [Khr22] Khronos. *OpenGL Core Profile*. 4.6. Khronos Group. Mayo de 2022.
- [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.
- [Rus23] Rust. *Rust*. <https://www.rust-lang.org/>. 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.
- [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.