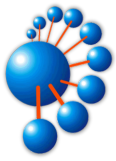
UNIVERSIDAD DE CONCEPCIÓN
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA INFORMÁTICA
Y CIENCIAS DE LA COMPUTACIÓN



Estudio empírico del rendimiento del k^3-tree en la renderización en tiempo real con el método de Voxel Ray Casting

POR

Nicolás Eduardo Rojas Arévalo

Memoria de Título presentada a la Facultad de Ingeniería de la Universidad de Concepción
para optar al título profesional de Ingeniero Civil Informático

Patrocinante
José Fuentes Sepúlveda

Comisión Evaluadora
Jérémy Barbay, Guillermo Cabrera-Vives

Concepción, agosto 2023

A mis queridos abuelos

Agradecimientos

Ocuparé este espacio para referirme a personas que últimamente se han vuelto importantes en mi vida, personas que han influenciado en mi y personas quienes han estado presentes a lo largo de todo este proceso. Todo esto con el fin de agradecerles por toda la ayuda brindada, el apoyo y los ánimos entregados, al igual que la buena actitud que me han hecho mantener, pues hacen que quiera llegar muy lejos, superándome cada vez, para disfrutar cada uno de mis logros en su compañía.

Inicio con los agradecimientos hacia mi profesor guía José Fuentes, por haber sido un pilar en el desarrollo de esta tesis, por mostrarme la mejor forma de encaminar el proyecto y por ayudarme a aterrizar un montón de ideas con salidas altamente divergentes.
Agradezco también a mi abuelita Sonia y a mi tata Hugo por apoyarme desde siempre y por criar a un joven con ambiciones y con la vista en el futuro. Son de las personas más importantes en mi vida y son el principal motivo de que haya logrado esto.
Les agradezco a mi madre y a mi padre por su cariño incondicional y por estar presentes en mi vida, en altos y bajos, hasta el día de hoy.
Finalmente, agradezco a Jookeez, Chelo, Izcan, Edu, Cereal y a todas las personas quienes me ayudaron a mantener la motivación a lo largo del semestre frente a cada dificultad que se presentó.

Resumen

En el método de Voxel Ray Casting, del campo de la renderización en tiempo real, el uso de la estructura de datos compacta k^3-tree es una opción viable frente a la estructura principalmente utilizada en el estado del arte. En los contextos donde se usa dicho método, se hace un intensivo uso de la memoria al cargar en la estructura grandes modelos tridimensionales. Es por esto que, una mejora en la eficiencia del uso del espacio podría ser un gran aporte. Se ha demostrado empíricamente que el k^3-tree brinda un mejor trade-off espacio-tiempo que estructuras de datos más tradicionales.

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Motivación

La *renderización en tiempo real* [1] es un proceso fundamental en el campo de los gráficos por ordenador, que consiste en la rápida generación de imágenes para ser mostradas en pantalla. Se realizan muchos cálculos en un tiempo muy reducido, de forma que múltiples imágenes sean generadas cada segundo y se perciban con fluidez, logrando en el usuario la sensación de interactividad y la percepción de movimiento.

El método de renderización en tiempo real más utilizado en el campo de los gráficos por ordenador es el método de *Rasterization* [2], el cual se basa en el uso de polígonos. Este método se destaca por su rapidez y el hardware actual está optimizado para el mismo, sin embargo, su uso de memoria crece mucho al trabajar con escenas de alto nivel de detalle, dado que se debe hacer uso de grandes cantidades de polígonos, al igual que texturas secundarias muy pesadas para enriquecer la información visible de las superficies. Esto ha impulsado la investigación en profundidad de técnicas alternativas como lo es la llamada *Voxel Ray Casting* [3] (conocida también como Volume Ray Casting), que brinda ciertas ventajas y utilidades al renderizar escenas de gran tamaño y complejidad, y se aproxima al problema de la renderización con un enfoque muy distinto.

Voxel Ray Casting es una técnica de *renderizado de volumen* [4], en la cual, los elementos de la escena se dividen en pequeñas partículas llamadas vóxeles, las cuales se almacenan en estructuras de datos tridimensionales. La estructura de datos usada principalmente es el *octree* [5], estructura cuya eficiencia en espacio vuelve competitiva a la técnica, pues reduce considerablemente el espacio utilizado al visualizar grandes escenas mediante la omisión de información redundante o innecesaria, cuya ilustración es visible en la Figura 1.1.1.

En el método de Voxel Ray Casting, la información en la estructura de datos es accedida al

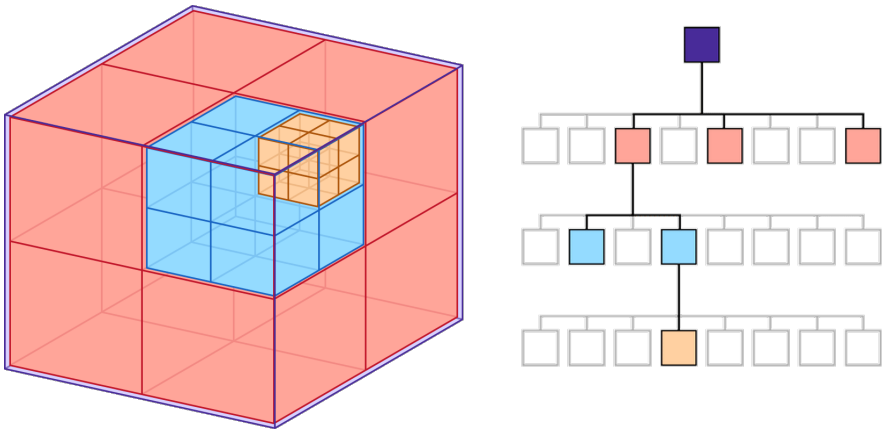


Figura 1.1.1: Visualización de un octree, con subdivisiones en algunos octantes.

simular un gran número de rayos que recorren la escena y obtienen el color correspondiente a cada píxel de la pantalla dentro de la resolución objetivo, como se ilustra en la Figura 1.1.2. De esta forma, los vóxeles accedidos son únicamente aquellos que han sido atravesados por los rayos, a la vez que cada rayo se detendrá cuando se cumpla una cierta condición (ej. si el vóxel accedido no es transparente). Al concluir la simulación de todos los rayos, se habrá generado una imagen 2D que será finalmente plasmada en la pantalla.

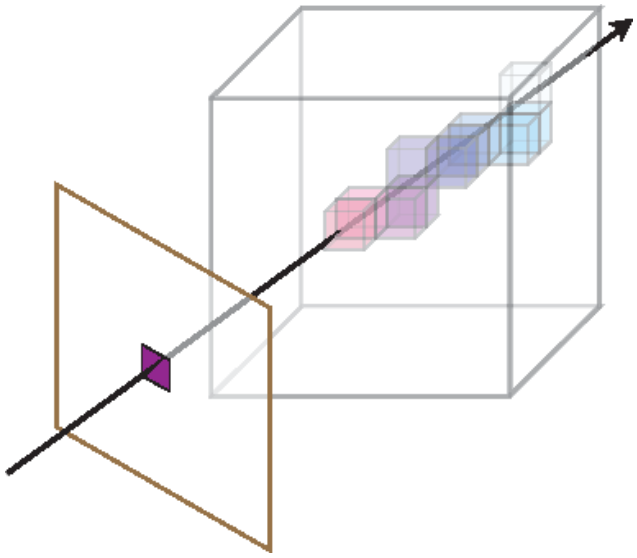


Figura 1.1.2: Ilustración del trazado de vóxeles para obtener el color de un píxel.

En materia de estructuras de datos, además del octree, y en lo que a eficiencia en espacio se refiere, existen mejores alternativas cuyo rendimiento no ha sido estudiado para efectos de la renderización en tiempo-real, motivo por el cual hay poca o nula documentación de su eficiencia espacial y temporal en este área. El objetivo de este trabajo es emparejar un método de renderización basado en vóxeles, como el anteriormente mencionado, con una estructura de datos compacta [6] llamada k^3-*tree* [7], para poder así lograr una implementación que

explote las capacidades de las estructuras de datos compactas cumpliendo el objetivo de la renderización en tiempo real.

1.2. Hipótesis

La hipótesis que se busca validar con el presente trabajo es: "En el método de Voxel Ray Casting es posible utilizar la estructura de datos compacta k^3-tree haciendo un menor uso del espacio y obteniendo un tiempo comparable a la estructura de datos tradicional". Con el fin de comprobar esta hipótesis, se pretende medir el impacto del uso de la estructura de datos compacta k^3-tree, evaluando y comparando su rendimiento contra el octree, tanto en términos de tiempo de renderizado como de uso de memoria.

1.3. Objetivos generales

El propósito de esta memoria de título es comparar empíricamente el impacto del uso del k^3-tree, la versión compacta del octree, en el método de Voxel Ray Casting.

Se busca evaluar el desempeño para observar si las estructuras de datos compactas significan una mejora frente a la estructura de datos principal en la renderización en tiempo real midiendo el rendimiento tanto en términos de uso de espacio como de tiempo de renderizado.

1.4. Objetivos específicos

1. Realizar estudio teórico del método de Voxel Ray Casting consultando el estado del arte.
2. Diseñar e implementar el prototipo del motor gráfico con un arreglo tridimensional.
3. Implementar las distintas estructuras de datos en el motor gráfico y evaluar su impacto en el uso de espacio y en el tiempo del renderizado.
4. Analizar y comparar los resultados obtenidos en las evaluaciones realizadas sobre escenas de distinta complejidad.
5. Identificar las ventajas y desventajas de cada estructura de datos.
6. Proponer posibles mejoras y optimizaciones a realizar para el trabajo futuro.

Capítulo 2

Revisión Bibliográfica

Antes de proceder con el trabajo realizado, se realizará una revisión de todos los aspectos a tener en cuenta, cuya comprensión es necesaria para un correcto entendimiento tanto del desarrollo como de los resultados de este proyecto de título.

2.1. Renderización en tiempo real

Se le llama renderización al proceso de generar imágenes bidimensionales que representen, de manera realista o no realista, una escena tridimensional, equivalente a fotografiar un objeto o paisaje con una cámara fotográfica, pues las imágenes que se obtienen como resultado no son más que proyecciones de la información tridimensional de origen, las cuales perdieron información en el proceso. Para ello se consideran una serie de elementos como lo son la cámara, con su respectiva posición y orientación, toda la geometría de la escena, iluminación, texturas, etc.

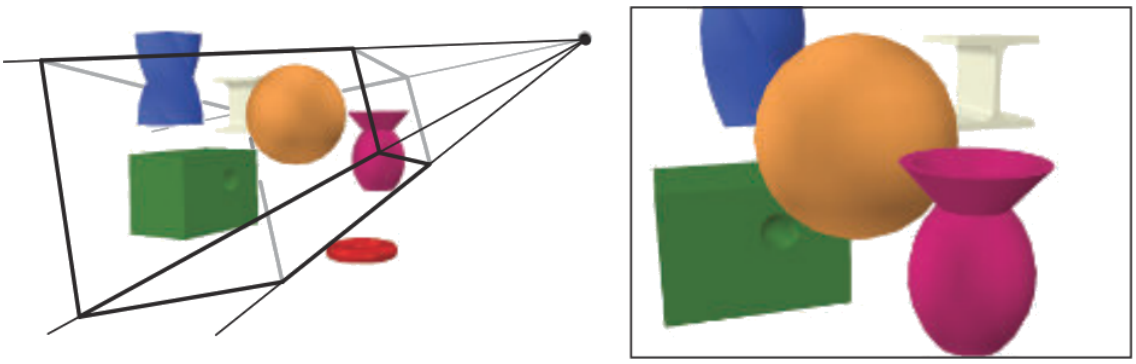


Figura 2.1.1: En la imagen de la izquierda, una cámara virtual está situada en la punta de la pirámide. Sólo se renderiza la geometría dentro del volumen de vista. La imagen de la derecha muestra lo que "ve"la cámara. [1]

La renderización en tiempo real [1], por otra parte, es un subcampo de la renderización que, a grandes rasgos, busca lograr el mismo objetivo pero de manera casi inmediata, para responder así a las acciones del usuario en el momento en que se interactúa con el programa. En la actualidad hay una gran cantidad de ejemplos de renderizados que toman tan poco tiempo que pueden generar decenas y hasta cientos de imágenes por segundo, pues está presente en muchas áreas, como es el caso de la visualización médica, arquitectura, simulaciones, videojuegos y otras aplicaciones.

En este contexto, es común referirse a la frecuencia de generación de imágenes, más conocida por el término FPS (Frames Per Second), para comparar el rendimiento en tiempo de distintos escenarios de renderización en tiempo real.

2.2. Vóxel

Así como un píxel es la unidad más pequeña en una imagen, un vóxel es la unidad más pequeña de un volumen. Un vóxel es un punto de información en el espacio y la representación de una unidad de volumen, normalmente se visualiza como un cubo que forma parte de una cuadrícula tridimensional uniforme. Los vóxeles son la forma tradicional de almacenar datos volumétricos, y pueden representar objetos como humo, modelos 3D como escáneres óseos o representaciones del terreno. Por lo general, un vóxel no necesita almacenar información de su posición, ya que su índice en la grilla que lo contiene determina su ubicación en esta [1].

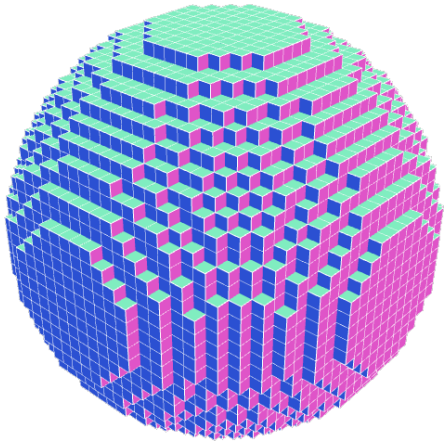


Figura 2.2.1: Esfera compuesta por una gran cantidad de vóxeles.

Esta forma de almacenar cuerpos y figuras vuelve factible un conjunto de distintas técnicas que se aprovechan de la localidad espacial de la información representada, de la simplicidad de la geometría y del formato mismo, para la aceleración de procedimientos como el trazado de rayos, parte importante del algoritmo de Voxel Ray Casting, técnica de renderizado de vóxeles que será vista en profundidad más adelante.

2.3. Estructuras de datos

2.3.1. Arreglo

Un arreglo o array es una estructura de datos estática que puede contener múltiples elementos del mismo tipo. Estos elementos se encuentran almacenados en direcciones de memoria contiguas y pueden ser accedidos con índices numéricos enteros.

2.3.2. Árbol

En ciencias de la computación, un árbol o tree es una estructura de datos conformada por un conjunto de nodos y conexiones entre éstos, cuya estructura logra establecer una jerarquía de árbol.

Los nodos pueden referenciar a otros nodos, esto es, a sus hijos y cada nodo está siendo referenciado siempre por únicamente un nodo padre, exceptuando el nodo raíz. Por último, cabe recalcar que pueden almacenar todo tipo de información en ellos.

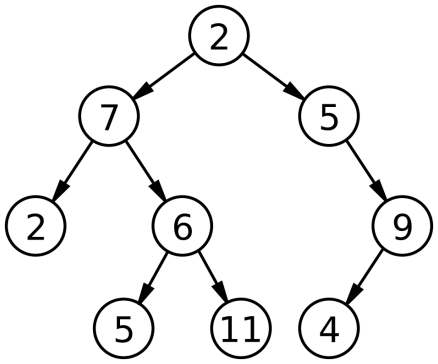


Figura 2.3.1: Representación visual de un árbol.

2.4. Malla poligonal

Una malla poligonal (o mesh) es un objeto compuesto por conjuntos de vértices, aristas y triángulos. Puede contener geometrías más complejas como cuadriláteros u otros polígonos, pero estos se consideran simples conjuntos de triángulos pues pueden ser descompuestos en éstos. Las mallas poligonales son utilizadas en muchas áreas de los gráficos por ordenador para definir la geometría de modelos 3D como formas, objetos, escenas, etc.

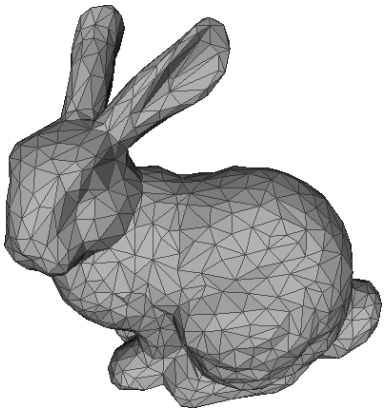


Figura 2.4.1: Malla poligonal de un modelo llamado Stanford Bunny.

2.5. Volumen delimitador

Un volumen delimitador es un cuerpo geométrico que tiene, por lo general, una geometría mucho más simple que los objetos que contiene, cuyo objetivo es mejorar la eficiencia de cálculos y operaciones (ej. detección de colisiones) que, en caso de realizarse sobre la geometría original, suponen una carga computacional mayor.

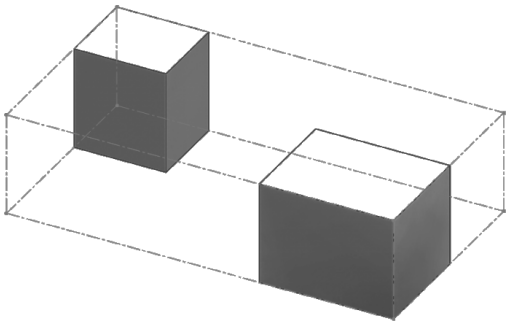


Figura 2.5.1: Ejemplo de volumen delimitador. Contiene dos objetos y se ajusta a ellos con la geometría de un paralelepípedo rectangular.

Capítulo 3

Voxel Ray Casting

La estructura de datos compacta evaluada en este trabajo reemplaza una parte del método de Voxel Ray Casting, específicamente en lo que a representación de la información espacial se refiere. Debido a esto, para entender el impacto que puede causar su uso, es necesario explicar este último de manera detallada, objetivo que se busca lograr en este capítulo.

3.1. Descripción del método

Voxel Ray Casting [3] es el nombre que recibe un método de renderizado de volumen que trabaja sobre escenas cuya información se encuentra descompuesta y representada en vóxeles. Al igual que cualquier método de renderización, lo que busca lograr es la generación de imágenes bidimensionales, específicamente rasters, que representen dicha escena tridimensional, lo cual se consigue mediante la simulación de una gran cantidad de rayos que recorren la escena para pintar dichas imágenes.

Existen varios términos muy similares en la actualidad, los cuales se refieren a diferentes cosas; Ray Casting es el nombre general que reciben las técnicas que realizan trazado de rayos en una escena, independiente de si está compuesta por polígonos o por vóxeles. Otro concepto es el de Ray Tracing, el cual, pese a ser la traducción directa al inglés de trazado de rayos, se refiere a una técnica de renderización muy pesada computacionalmente, que se basa en la simulación de varios rayos por píxel, los cuales rebotan múltiples veces en distintas superficies, principalmente polígonos, intentando lograr una representación de escenas con iluminación realista y físicamente correcta. Luego, dado de que en este trabajo se enfoca específicamente en la renderización de vóxeles con la simulación de rayos primarios únicamente, es que nos referimos al método como Voxel Ray Casting, además, con 'trazado de rayos', no nos referimos a la famosa técnica si no que hablamos específicamente de la simulación del avance de los

rayos en el espacio tridimensional.

La mayor parte del método consiste únicamente en la ejecución del trazado de rayos, los cuales avanzan a través de una grilla. Cada uno de los rayos consulta a la estructura de datos únicamente en las posiciones que colisiona, recolectando información del color a lo largo de su trayecto. Generalmente son simulados tantos rayos como píxeles tiene la imagen de salida, correspondiendo a cada píxel un rayo a simular que eventualmente definirá su color. Opcionalmente, se pueden simular rayos secundarios para brindar mejor calidad a la renderización a cambio de emplear un mayor tiempo en el cómputo.

A continuación se explicará con detalle los pasos en el funcionamiento, al igual que las problemáticas con las que se enfrenta el método de Voxel Ray Casting explicando para cada una la forma en que se solucionó para el desarrollo de la implementación realizada en el presente trabajo.

3.2. Configuración inicial

La ejecución del método se realiza dada una resolución en la cual serán mostradas las imágenes generadas. Dado que nos ubicamos en el contexto de la renderización en tiempo real, el método se encuentra en constante generación de imágenes, repitiendo el renderizado una y otra vez.

La cámara de la escena es la materialización de lo que se suele llamar el *observador*, y está representada con una posición c y un vector dirección $\vec{f}$.

Para una resolución de $n \times m$ píxeles, se simulan $n * m$ rayos, correspondiendo un rayo para cada píxel, cada uno con un determinado origen y una determinada dirección. Para una proyección en perspectiva, que es la proyección usada en este trabajo, el origen de todos los rayos es la misma posición c de la cámara, y la dirección se desvía del vector $\vec{f}$, de forma tal que, mientras más alejado del píxel central se encuentre el píxel correspondiente a un rayo, mayor será la desviación. Las componentes de estas desviaciones son independientes y se mantiene una separación uniforme entre vectores. La cantidad que se separan los vectores está dada por el campo de visión.

Por último, el método tiene acceso a la estructura de datos que contiene la información de los vóxeles, independiente del tipo de estructura que sea. Además, conoce el volumen delimitador o bounding volume que la contiene.

En esta aplicación, se utiliza un volumen delimitador conocido como Axis Aligned Bounding Box (tratado mayormente como AABB), dado que la grilla que contiene a los vóxeles está alineada con los ejes, es decir, no se encuentra rotada de ninguna manera. Luego, el volumen se puede definir con dos de sus esquinas opuestas representadas por dos puntos a y b tal que

$a_x < b_x$, $a_y < b_y$ y $a_z < b_z$.

3.3. Problema de colisión Ray/Box

En la simulación de un rayo cualquiera con origen o y dirección $\vec{d}$, antes que nada se determina si este intersecta el volumen delimitador, esto es, el espacio capaz de contener vóxeles.

Dado un caso hipotético donde el origen o del rayo se encuentre fuera del volumen delimitador, puede darse que el vector $\vec{d}$ apunte hacia una dirección que no intersecta ningún vóxel y ni al volumen delimitador mismo, caso en el cual se debe retornar un color de fondo. Sin embargo, también puede ocurrir que el rayo apunte en dirección al volumen delimitador, y se deberá realizar el trazado de la grilla desde el punto de contacto más cercano. Ambos casos se ilustran en la Figura 3.3.1.

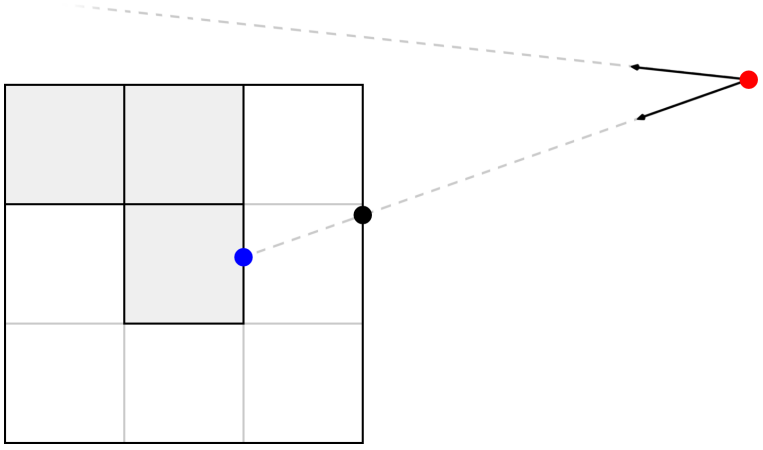


Figura 3.3.1: Ejemplo de dos rayos disparados desde la cámara. Uno se escapa del área con vóxeles y el otro intersecta eventualmente, pese a que su origen está fuera.

Para identificar el caso, es que se realiza una comprobación de colisión entre el rayo y el volumen delimitador usando el método de losas [8]. Este método permite determinar si un rayo intersecta un paralelepípedo rectangular y, en el caso de realizarse intersección, permite conocer los puntos en que el rayo penetra el volumen.

Conociendo el resultado de ejecutar el método de losas, se retorna un color de fondo para aquellos casos en que el rayo no intersecta el volumen y, en caso de haber intersección se continúa con el siguiente paso.

3.4. Problema del trazado de rayos

El momento en que un rayo intersecta el volumen delimitador, se debe proceder con la simulación del rayo de una manera más controlada. Ya no es útil realizar intersecciones Ray/Box, pues, comprobar la colisión entre cada par rayo-vóxel dispararía la complejidad temporal del renderizado. En este paso, es mejor idea aprovechar la geometría de la grilla.

Como sabemos que los vóxeles se encuentran ubicados en una grilla uniforme alineada con los ejes, podemos utilizar esa información para consultar únicamente las posiciones por donde el rayo avanza. Esto es fácil en un inicio, pues si el rayo está pasando por un punto P cualquiera dentro de una posición en la grilla, esa posición será aquella que resulte de truncar cada componente del punto P. Sin embargo, si el rayo debe continuar debido a que en esa ubicación no se almacenaba ningún vóxel, entonces se requerirá saber qué distancia debe avanzar para dar con la siguiente posición intersectada.

Una aproximación simple es hacer que el rayo avance pasos de longitud t cada vez, sin embargo, el problema de este acercamiento es que dependiendo del valor de esa longitud, aumentará el tiempo de cómputo o los errores por omisión de vóxeles, como se muestra en la Figura 3.4.1.

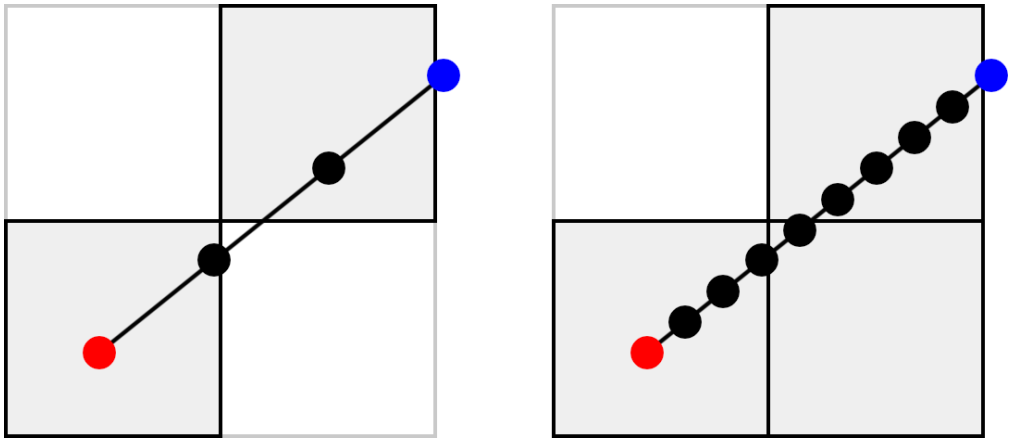


Figura 3.4.1: Visualización de dos casos de trazado de un mismo rayo que inicia en el punto rojo y termina en el punto azul. A la izquierda, el trazado del rayo omite una posición por dar saltos muy largos. A la derecha, el trazado del rayo realiza múltiples consultas en las mismas posiciones vacías debido a que los saltos son muy cortos.

El problema del trazado de rayos en grillas uniformes es conocido y existen varios algoritmos que satisfacen distintas necesidades. Para el caso de Voxel Ray Casting se requiere un algoritmo de trazado preciso, de manera tal que no permita que la visibilidad de los vóxeles sea intermitente o sensible a movimientos leves que puedan perjudicar la correctitud de las interseciones. A continuación se revisarán dos algoritmos muy conocidos en el área de

la renderización que, pese a que su objetivo es el dibujado de líneas, solucionan la misma problemática que presenta el trazado de rayos en una grilla.

3.4.1. Bresenham Algorithm

El algoritmo de dibujado o rasterización de líneas de Bresenham [9] es un método rápido para pintar los píxeles intersectados por un segmento. Consiste en que, dada una línea o un segmento $\overline{\text{AB}}$, busca cuál es el eje a lo largo del cual la diferencia de las componentes de los puntos es mayor. Las coordenadas de ese eje serán las que se considerarán independientes, mientras que las coordenadas del otro eje serán dependientes.

Como lo ilustra la Figura 3.4.2, inicialmente se pintan las posiciones que contienen los puntos A y B. Luego, la componente independiente de la siguiente posición a pintar, estará alejada en una unidad en dirección al punto B. La componente dependiente se calculará con la ecuación de la recta, y así se habrá obtenido un par ordenado que especifique la posición a pintar.

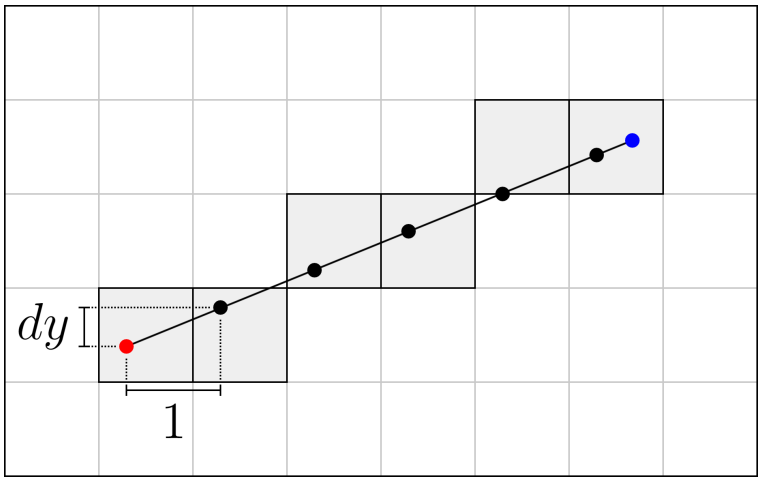


Figura 3.4.2: Ejemplo del algoritmo de Bresenham.

El algoritmo de dibujado de líneas de Bresenham, por su diseño, realiza saltos de longitud 1 en el eje de las abscisas determinado, que se traduce la mayoría de las veces en un salto de longitud mayor a 1 a lo largo de la línea, lo que causa que el algoritmo omita posiciones intersectadas. Por este motivo es que se concluye que no satisface el requerimiento de colisión precisa y, es requerido el siguiente en la lista.

3.4.2. 3D-DDA Algorithm

El algoritmo de dibujado de líneas 3D Digital Differential Analyzer [10] acumula la distancia que se debe avanzar a lo largo del segmento $\overline{\text{AB}}$ para llegar a la siguiente intersección, por

cada eje. Esto lo realiza sumando en cada iteración la longitud Δt_x, Δt_y o Δt_z, dependiendo de en qué eje se realizó la última intersección. Para un eje cualquiera, esta distancia Δt_{eje} es la longitud que debe recorrerse a lo largo de la linea de tal forma que el avance de la proyección de la linea en ese eje sea de 1.

Los valores t iniciales son una fracción de los mismos valores Δt_x, Δt_y y Δt_z que depende de la posición inicial del punto A en la casilla que lo contiene. Luego, en cada iteración, se elige la componente cuyo valor t acumulado sea menor para obtener la siguiente posición a pintar. Se realizan las iteraciones necesarias hasta llegar desde el punto A al píxel que contiene al punto B.

En la Figura 3.4.3 se ilustra el funcionamiento del algoritmo en una versión bidimensional, en donde se muestra la procedencia de los valores Δt_{eje}.

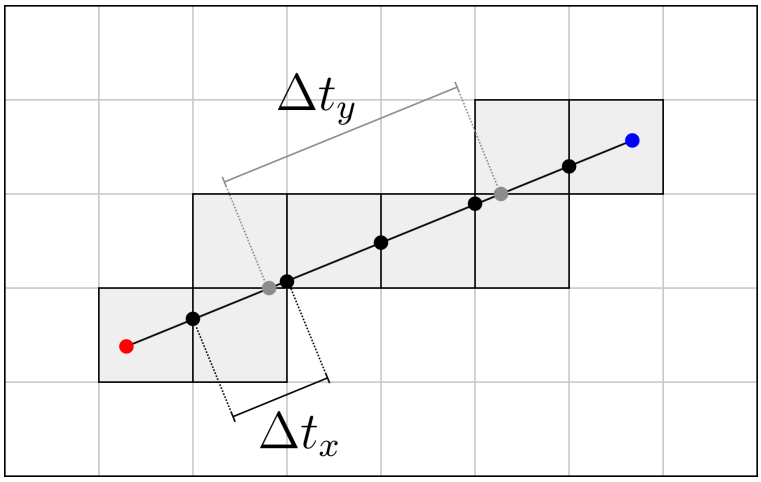


Figura 3.4.3: Representación bidimensional del algoritmo 3D-DDA utilizado.

Este método presenta una solución analítica al problema de dibujado de líneas y, dado que encuentra todas las intersecciones, sin necesidad de hacer saltos que intentan aproximar los puntos de intersección esperando no perderse ninguno, se concluye que sí satisface el requerimiento de colisión precisa.

Para la presente implementación, se tuvo que implementar una versión tridimensional de este algoritmo para ser usado en la implementación del método Voxel Ray Casting de este trabajo.

Capítulo 4

Estructuras de datos

El método de Voxel Ray Casting se basa fuertemente en la estructura con la cual es implementado. Esto es así en primer lugar por que el almacenamiento de grandes cantidades de vóxeles requiere una solución eficiente en espacio, desafío que es típicamente no trivial, además, por que la implementación del trazado de rayos debe solucionar el recorrido de la estructura de datos de una manera correcta para la consulta de las casillas atravesadas de manera precisa, aprovechando a la vez cualquier aspecto que potencie su eficiencia temporal.

A continuación se presentarán tres estructuras de datos con las que se implementó el algoritmo de Voxel Ray Casting, dentro de las cuales se encuentra la estructura de datos compacta k^3-tree.

4.1. Definiciones

4.1.1. Arreglo tridimensional

El arreglo tridimensional es una estructura de datos que consiste básicamente en un arreglo que contiene arreglos bidimensionales, los que a su vez son arreglos de arreglos unidimensionales. Esta estructura se puede ver como una grilla con ancho, alto y largo, como lo muestra la Figura 4.1.1, en donde cada casilla es una variable que puede ser accedida con índices numéricos enteros.

Cabe decir que los arreglos usan memoria para cada una de sus casillas siempre, por lo que, pese a que no se asigne valor en ciertas posiciones, al estar declarado el arreglo, se está usando en todo momento la totalidad de la memoria.

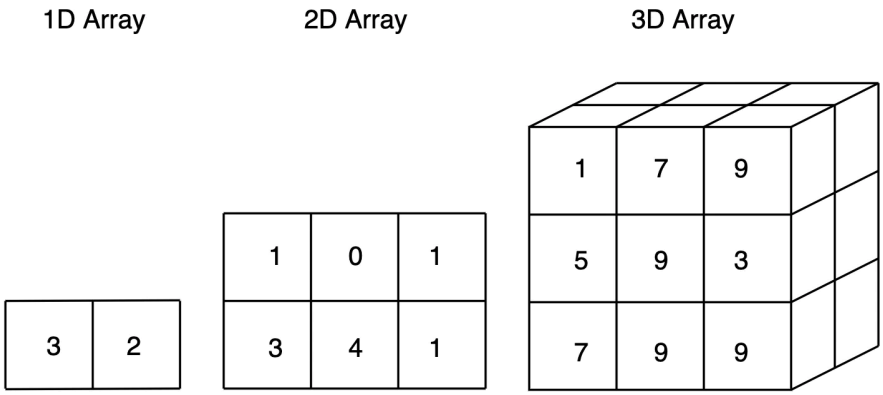


Figura 4.1.1: A la izquierda se encuentra un arreglo unidimensional de números enteros, contiene dos elementos. En el medio, se encuentra un arreglo bidimensional, contiene dos arreglos de números enteros (ej. [1, 0, 1]). A la derecha, se encuentra un arreglo tridimensional, contiene dos arreglos bidimensionales (en la imagen sólo es visible uno).

4.1.2. Octree

El octree es una estructura de datos tipo árbol cuyo principal uso es la división de un espacio tridimensional en 8 partes iguales llamadas octantes. Una unidad cúbica de espacio inscrita en la grilla, independiente de su tamaño, es representada en esta estructura de datos como un nodo. Es decir, los nodos del octree son la representación de cada uno de los octantes que surgen de las divisiones, a excepción del nodo raíz, el cual representa todo el espacio tridimensional encapsulado por el octree y no surge de una división previa.

Los nodos de un octree contienen, además de la información necesaria para referenciar a los hijos, datos respecto del vóxel que representan como puede ser un vector normal, color, sombreado, material, etc.

Los nodos en cuyo espacio no exista información u objetos a almacenar, no se subdividen y se consideran hojas del árbol. De esta forma, esta estructura puede ahorrar memoria al omitir por completo el almacenamiento de nodos que representen un espacio vacío.

En la práctica, existen diferentes implementaciones las cuales pueden aprovechar las circunstancias en que se da su uso o satisfacer distintas necesidades; algunas pueden hacer uso de direcciones de memoria secuenciales en el almacenamiento de los nodos hijo para reducir de ocho a uno la cantidad de punteros utilizados, otras pueden cambiar el diseño de sus nodos para incluir más o menos propiedades, como por ejemplo, las que se utilizan en la etapa de renderizado.

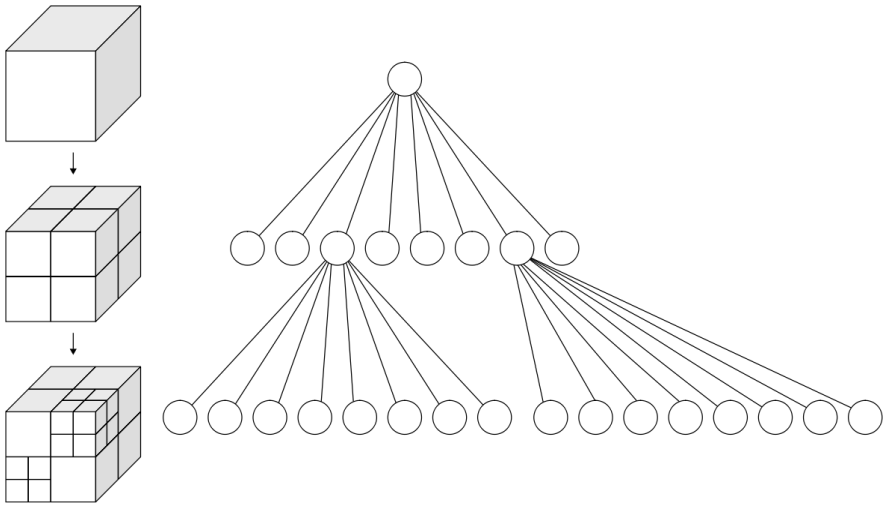


Figura 4.1.2: Representación visual del octree y diagrama del árbol correspondiente. Se puede observar que el nodo raíz tiene exactamente ocho hijos, de los cuales, seis representan octantes sin información y no tienen hijos, mientras que hay dos que representan octantes con información, motivo por el cual se subdividen nuevamente y, por ende, tienen ocho hijos cada uno.

4.1.3. k^2-tree

El k^2-tree es una estructura de datos compacta [6] diseñada para la representación de matrices binarias dispersas. Permite representar de manera eficiente las submatrices resultantes de dividir el espacio en k^2 partes iguales, para un valor entero k, mediante el uso de vectores de bits. Cabe recalcar que la división de un espacio de tamaño $n \times n$ en k^2 partes iguales tiene como requisito que el tamaño de la matriz sea una potencia de k.

Los nodos del árbol contienen información de la división de la submatriz que representan. Específicamente, un nodo cualquiera, al que le corresponde una submatriz de tamaño $n \times n$, tiene k^2 hijos cuyos valores indican si las submatrices de tamaño $\frac{n}{k} \times \frac{n}{k}$, resultantes de la división recursiva de ese espacio, contienen al menos un 1.

Un ejemplo que ayuda a comprender el funcionamiento es el siguiente: Si un nodo A es un 1, entonces tiene 4 nodos hijo. Si estos son 0010, entonces el tercer nodo hijo de A es el único que tiene otros 4 hijos, y sólo el tercer cuadrante del área que A representa, está subdividido en 4 partes. Los otros nodos de valor 0 se quedan como hojas, pues el espacio vacío no se subdivide. En resumen, cada nodo de valor 1 tendrá tendrá k^2 hijos.

En la siguiente Figura 4.1.3 se puede ver una matriz de valores binarios, a la derecha el árbol que la representa, y abajo los dos vectores de bits que almacenan la codificación del árbol.

La codificación del k^2-tree utiliza dos vectores de bits compactos. El primero guarda los bits de cada nodo interno ordenados por nivel, y el segundo guarda los bits de los nodos en el

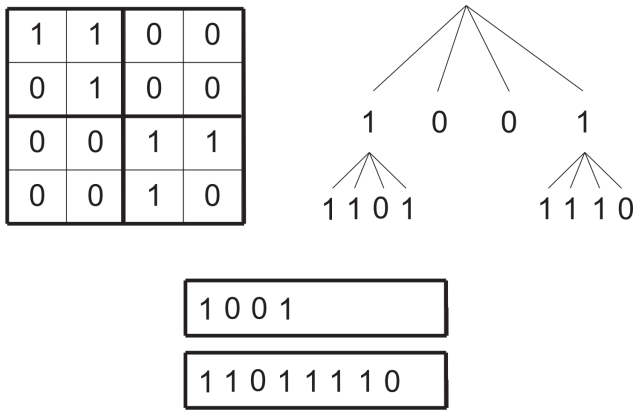


Figura 4.1.3: Visualización de un k^2-tree, k igual a 2.

último nivel. El motivo de la separación se encuentra en técnicas dentro de la implementación que buscan mejorar la eficiencia de las consultas, haciendo uso de funciones como *rank* y *select*.

4.1.4. k^3-tree

El k^3-tree [7] es la versión tridimensional del k^2-tree, y se puede considerar de cierta manera una versión compacta del octree. El principio de división del espacio es similar, y prácticamente equivalente si se utiliza un valor de k igual a 2, aunque existen varias diferencias como lo es la información extra que se empaqueta en los nodos del octree, la complejidad de la implementación, la forma en que se codifica la información almacenada, el tipo de consultas que permite, etc.

Por otra parte, las diferencias con el k^2-tree son básicamente los valores que dependen de la cantidad de dimensiones involucradas. Por ejemplo, dado el mismo valor k, el espacio se dividirá en k^3 partes iguales en lugar de k^2 partes y, como es de esperarse, el mismo cambio aplica a la cantidad de bits almacenados por nodo. Ese es el aspecto que le da el nombre a las estructuras k^d-tree.

Conceptualmente, cualquier algoritmo o programa diseñado para usar un octree, puede ser modificado para usar el k^3-tree, dado que ambos soportan las mismas operaciones y sólo se diferencian en su modo de codificar los datos.

4.2. Adaptado del trazado de rayos

En esta sección se profundizará en la implementación base que se desarrolló, además de los cambios que fue requerido realizar para adaptar el método a las distintas estructuras de datos

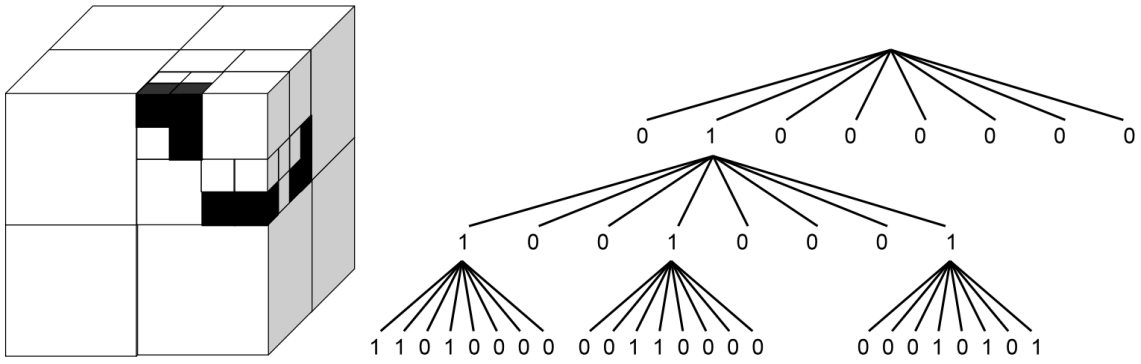


Figura 4.1.4: Matriz tridimensional de valores binarios de tamaño 8^3 y la representación del árbol y los conjuntos de bits por nodo. Se usa un k igual a 2.

de manera que aprovechase las características de las mismas para un mejor funcionamiento en los casos donde fue posible.

4.2.1. Trazado del arreglo tridimensional

La implementación de base se realizó teniendo en mente la estructura de datos más simple de las tres con las que se trabajó en este proyecto, la cual es el arreglo tridimensional.

Se decidió así de modo que se lograse tener una versión funcional que posteriormente sería alterada para desarrollar la versión alterna de la misma, adaptada a las estructuras de datos en árbol, siguiendo el proceso de desarrollo incremental de software.

Como se explicó en el Capítulo 3, el algoritmo de dibujado de líneas seleccionado fue el algoritmo 3D Digital Differential Analyzer, el cual trabaja sobre una grilla uniforme. Esto calza con la geometría de la representación visual de la matriz tridimensional, que tiene casillas de tamaño uniforme.

En la siguiente Figura 4.2.1 se muestran los pasos de la simulación de un rayo sobre una grilla bidimensional, cuyo origen es el punto o, y dirección es el vector $\vec{v}$. Se pueden identificar el primer paso, que es la comparación de intersección entre un rayo y una caja, la cual retorna el punto h desde donde se comienza el segundo paso, que es el trazado de rayos en la grilla. La primera consulta se realiza en la casilla 1 y dado que no contiene un vóxel, el trazado continúa. El rayo atravieza todas las posiciones enumeradas, ilustradas en rojo las posiciones alcanzadas por una intersección con planos verticales y con azul aquellas alcanzadas por intersecciones con planos horizontales. Cada posición es consultada en la estructura de datos, pero no es hasta que se corta el plano del costado izquierdo de la casilla 9 que la consulta retorna un vóxel válido. Posteriormente, el rayo retorna el color correspondiente.

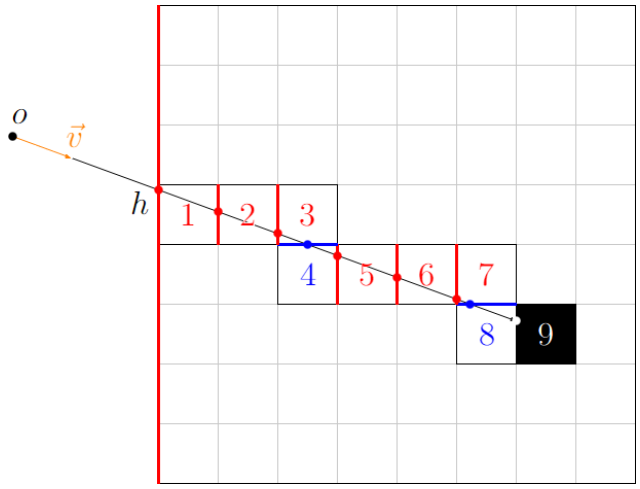


Figura 4.2.1: Ejemplo ilustrado del trazado de un rayo.

4.2.2. Trazado de estructuras en árbol

Para la versión alterna adaptada a las estructuras de datos de tipo árbol, se implementó una característica que consiste en la omisión del espacio vacío representado por nodos de tamaño mayor al vóxel mínimo.

En el contexto del uso de las estructuras de datos octree o k^3-tree, se mencionó anteriormente que, los octantes en el caso del octree, y las submatrices en el caso del k^3-tree, sufren una división únicamente cuando contienen información, esto es, cuando hay presencia de uno o más vóxeles ubicados en el espacio que representan.

Esta característica permite la optimización del trazado de rayos, dado que, en casos de intersección de un rayo con un nodo vacío con tamaño $n > 1$ en cada eje, seguir avanzando a la siguiente celda de tamaño 1 puede causar que se consulte múltiples veces en un mismo nodo vacío. La implementación del método aprovecha esto realizando saltos entre planos de una nueva grilla uniforme, pero en este caso, de tamaño n. De este modo, el valor t de la siguiente intersección de cada eje considera la siguiente intersección de plano fuera del espacio representado por el nodo en cuestión, solucionando el problema del trazado de una grilla irregular.

Un ejemplo equivalente al trazado del arreglo tridimensional se ilustra en la siguiente Figura 4.2.2, donde se puede observar que la cantidad de iteraciones necesarias para alcanzar al vóxel de color negro disminuyó de 9 a 4.

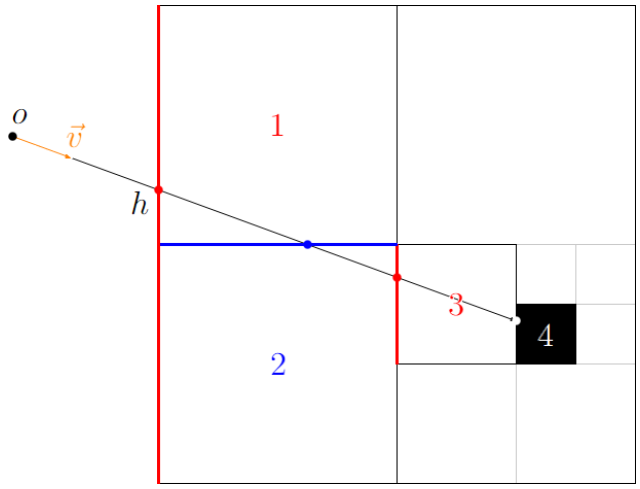


Figura 4.2.2: Ejemplo ilustrado del trazado de un rayo en una estructura en árbol.

4.3. Análisis teórico

Con la explicación previa de cada estructura de datos, cuyo rendimiento será comparado en el capítulo de análisis experimental, se profundizará a continuación en materia de complejidad espacial y temporal de los distintos escenarios que se pueden llegar a dar. Para ello, nos enfocaremos en notación Big-O y distintas medidas que se obtendrán en los experimentos para tener una mejor comprensión de las ventajas y desventajas de las estructuras, al igual que sus fortalezas y debilidades.

4.3.1. Factor de ramificación

Samuli Laine y Tero Karras [5] presentan, en el análisis de espacio del octree, una medida relacionada al factor de ramificación del árbol, a la cual denominan *dimensionalidad*.

$$D_{local}(l, s) = log_2 \frac{N(l, s)}{N(l, s + 1)}$$

Sea q la proporción entre la cantidad de nodos no vacíos de tamaño 2^s y la cantidad de nodos no vacíos de tamaño 2^{s+1}, valor el cual se encuentra entre 0 y 8, se define la dimensionalidad como el logaritmo en base 2 de q. De esta forma, la dimensionalidad será un valor en el intervalo $]0, 3]$. Notar que, para datasets de superficies planas la dimensionalidad será cercana a 2 por la bidimensionalidad de la geometría, mientras que para datasets mascizos se acercará a un valor de 3.

Esta medida propuesta en su trabajo es equivalente a lo que puede mostrar el factor de ramificación promedio y, la diferencia recae únicamente el que se consideran nodos no vacíos, mientras que un promedio simple consideraría factores de ramificación iguales a 0 gracias a la existencia de nodos vacíos.

Se considerará esta medida para obtener más información de los experimentos, en los cuales se espera poder visualizar una tendencia de la dimensionalidad según va aumentando la resolución.

4.3.2. Factor de división del espacio

En términos de espacio, la medida de dimensionalidad brinda información para relacionar los distintos datasets y su rendimiento en memoria, sin embargo, no es un factor directamente relacionado con la complejidad temporal del método de renderización en su totalidad.

Dado que el algoritmo de trazado de rayos debe realizar un recorrido por la estructura de datos, cortando los planos que conforman la grilla tridimensional en la que se ubican los vóxeles, la forma de ésta afecta directamente al tiempo de renderizado. Esta influencia es positiva en ciertos casos, brindando al método la capacidad de omitir muchos saltos, correspondientes al espacio representado por nodos hoja vacíos de gran tamaño, mientras que es negativa en otros, al tener grandes cantidades de divisiones que aumentan el parecido a una grilla uniforme.

Un comportamiento similar se puede ver con distintos valores de resolución de la grilla. El rendimiento en tiempo cae cuando esta resolución es muy grande, sin embargo, esta es una característica necesaria para lograr el objetivo de generar imágenes detalladas que representen la geometría sin un aspecto cúbico.

Esta medida de bifurcación del espacio está directamente relacionada con la cantidad de nodos internos del octree y del k^3-tree, debido a que el propósito mismo de los nodos es el de dividir el espacio. Es por eso que la cantidad de nodos internos será considerada en los experimentos con el objetivo de hacer visible esta relación.

4.3.3. Recorrido de árboles

Un aspecto ligado al uso de estructuras de datos de tipo árbol es la necesidad de recorrer los nodos del mismo al realizar consultas y búsquedas. Este es un aspecto que aumenta la demanda computacional y, en este caso, el tiempo de renderizado del método.

Para cada nueva consulta del árbol, en los casos del k^3-tree y del octree, el método inicia el recorrido a partir de la raíz. Sin embargo, existen distintas formas de evitar la repetición del recorrido, haciendo uso de pilas o stacks para almacenar el orden de visitado de los nodos y

reutilizarlo posteriormente. Además, en este contexto, los nodos cuyo espacio es intersectado por un mismo rayo a lo largo de su simulación, son contiguos y cuentan con una cierta localidad espacial, lo que hace este aspecto algo a tener en cuenta para el trabajo futuro. Esta carga computacional extra no existe en el caso del arreglo tridimensional pues cuenta con acceso de tiempo constante.

4.3.4. Análisis de complejidad

Todos los aspectos recién mencionados son importantes de considerar para poder efectuar una mejor definición de la complejidad tanto espacial como temporal del método de Voxel Ray Casting con cada una de las tres estructuras de datos.

La complejidad del algoritmo será expresada con la notación Big-O, que es utilizada en ciencias de la computación para referirse a qué tan rápido crece el tiempo o la memoria que requiere un algoritmo o programa, en términos de, por ejemplo, el tamaño de la entrada.

Se listan a continuación las distintas variables que participan en la complejidad temporal del método:

1. Dado que se simula un rayo por cada píxel, se considera la cantidad de total de éstos en la imagen resultante, que está dado por el producto de las variables w y h que especifican la resolución.
2. El valor i_{max}, que se define como la máxima cantidad de posiciones de la grilla que una recta contenida dentro del volumen delimitador puede intersectar, esto es una cota superior de las intersecciones realizadas por cada rayo simulado.
3. La altura h_t del árbol, ya que, por cada posición atravesada por un rayo, éste es recorrido para encontrar al nodo hoja cuya submatriz representada contiene al punto de intersección.
4. Se define p como la proporción de rayos que efectivamente entran al volumen delimitador. Siendo $1 - p$ la cantidad de rayos que omiten por completo la etapa de trazado de la grilla.

Con esto en mente, la complejidad temporal está acotada superiormente por $\mathrm{O}(p * w * h * i_{max} * h_t + (1 - p) * w * h)$ para las versiones del método que implementan las estructuras k^3-tree y octree, mientras que para el arreglo tridimensional, el cual es accedido en tiempo constante, es de $\mathrm{O}(p * w * i_{max} + (1 - p) * w * h)$.

La complejidad espacial, por otra parte, expresada en términos de la cantidad de nodos n para el k^3-tree y octree, está acotada superiormente por O(n). Esto es así debido a que la cantidad de nodos del árbol hacen crecer el uso de espacio de manera lineal. La complejidad espacial

del arreglo tridimensional, por otro lado, está acotada por $O(n^3)$, donde n es el tamaño del arreglo en una dimensión espacial, asumiendo que su ancho, alto y largo son iguales, como fue requerido para las otras estructuras.

Capítulo 5

Evaluación experimental

5.1. Implementación

El código desarrollado en el transcurso de esta memoria de título se encuentra en el repositorio de GitHub disponible en `https://github.com/zetaso/VOXEL`.

Los experimentos se llevaron a cabo en una máquina con las siguientes características:

- Procesador Intel Core i3-10105@3.8GHz.
- Memoria RAM de 16 gigabytes.
- Sistema operativo Linux Mint (kernel 5.15.0-56).
- Compilador gnu/g++ 11.3.0.

5.2. Datasets

El dataset está compuesto por 7 modelos de distinta complejidad en formato *Standard Triangle Language* (STL), cuya renderización se adjunta en el anexo B, para los cuales existen recorridos de cámara automatizados que buscan capturar información de distintas zonas dentro del escenario.

Virtualmente, se realiza la voxelización de estos modelos en resoluciones de 64, 128, 256, 512 y 1024 vóxeles para cada dimensión de la grilla. La voxelización es el proceso de descomponer un modelo tridimensional cualquiera en unidades de volumen que son las que finalmente pueden ser almacenadas en las estructuras de datos. Para cumplir este objetivo de descomponer un cuerpo, objeto u escena en vóxeles, existen distintos algoritmos de voxelizado.

Dado que esta etapa se lleva a cabo solo una vez al inicio y no se considera parte del renderizado en sí, se diseñó un algoritmo directo no necesariamente eficiente para satisfacer esta necesidad. El algoritmo consiste en almacenar en la estructura de datos aquellos vóxeles que intersecten al menos un triángulo del modelo, para modelos en formato de malla poligonal, para lo cual se hacen comparaciones de intersección triángulo-caja entre cada vóxel dentro del volúmen delimitador del triángulo y el triángulo mismo, proceso que se repite para cada triángulo del modelo tridimensional.

De los distintos datasets se puede recalcar que cumplen el requisito de aportar variabilidad en la geometría. Hay modelos cuya estructura centra el objeto en el espacio, maximizando el potencial de las estructuras en árbol y hay otros que fuerzan los peores casos, como lo es el dataset *city* o el dataset *cube* basado en un fractal, los cuales se expanden en la escena y causan muchas subdivisiones en el espacio.

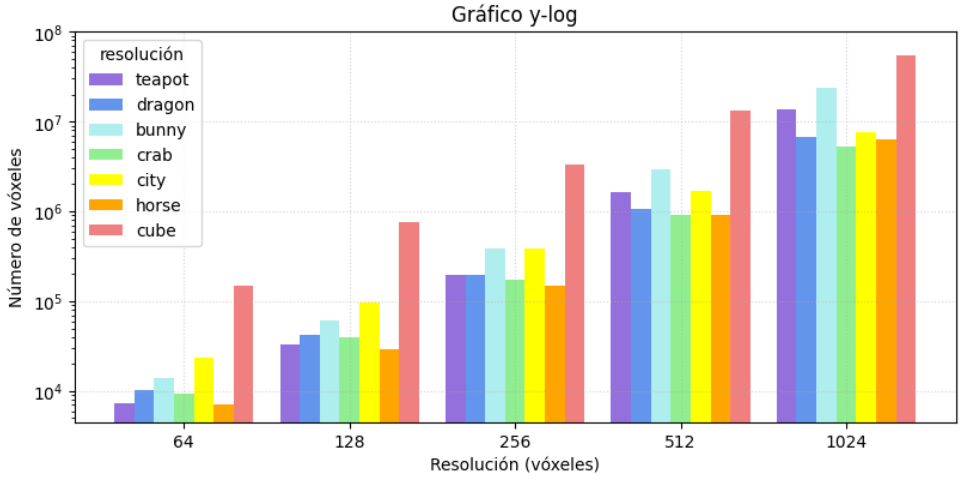


Figura 5.2.1: Gráfico de tamaño de los datasets de prueba. El eje vertical está expresado en escala logarítmica.

5.3. Experimentos

La implementación del k^3-tree usada en este proyecto [11], es en específico una versión binaria, cuya información representada por posición en el espacio es la existencia o ausencia, esto debido a que usa vectores de bits como se vio en el capítulo anterior, esto significa que la información que puede guardar no es equivalente al octree. Para permitir la comparación en igualdad de condiciones, todas las medidas de espacio realizadas al k^3-tree consideran un múltiplo de 32, ya que la cantidad de 1 bit actual falta de 31 bits más para almacenar valores del mismo tipo de dato que el octree y el arreglo tridimensional ya almacenan, y además, por que no supone un cambio en el diseño del algoritmo que requiera evaluar su factibilidad.

Se llevó a cabo un total de 105 experimentos, consistiendo en las combinaciones posibles donde se mide el rendimiento de cada estructura de datos con cada uno de los datasets de prueba, voxelizados para las 5 resoluciones especificadas. De estos experimentos se extrajo una serie de valores entre los cuales se encuentran la tasa de cuadros por segundo promedio, la desviación estándar de la misma, saltos promedio realizados en el trazado de rayos, promedio de rayos simulados por segundo, uso de memoria, número de nodos internos, número de vóxeles, y dimensionalidad.

En los gráficos que muestran el rendimiento en tiempo mediante la tasa de cuadros por segundo promedio, se puede ver una mejor respuesta general tanto del octree como del k^3-tree, frente a los escenarios con cada vez más nivel de detalle, como lo muestra la gráfica visible en la Figura 5.3.1.

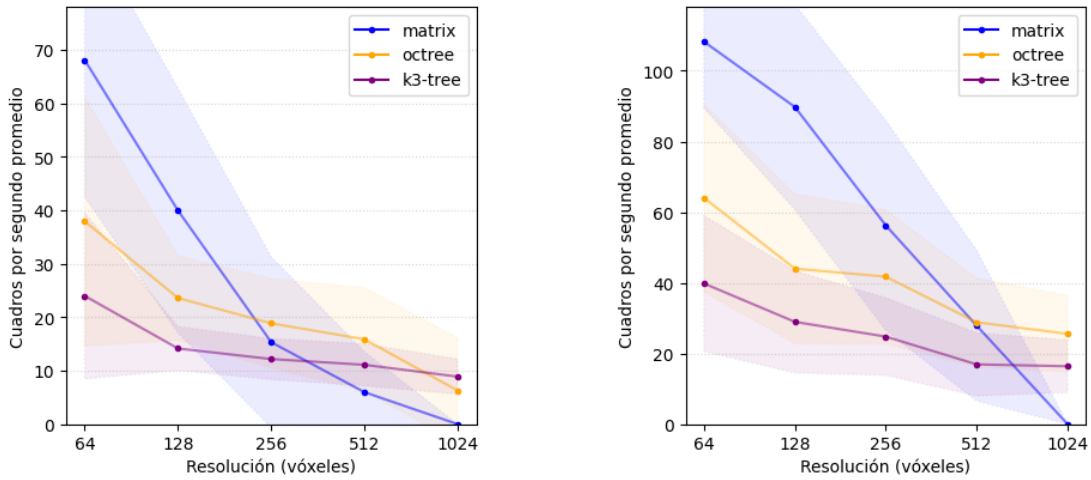
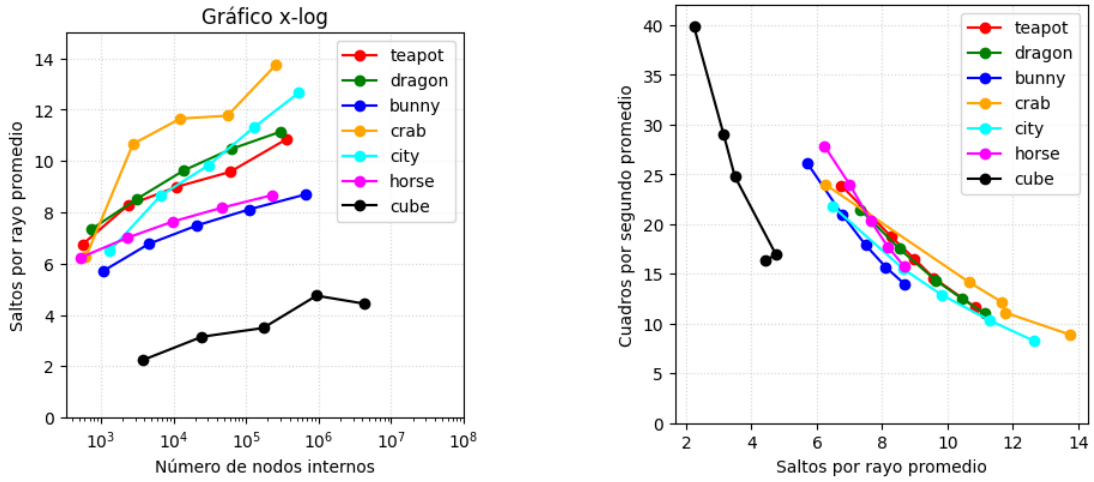


Figura 5.3.1: Gráfico de cuadros por segundo promedio para las distintas resoluciones de los dataset *crab* y *cube*. El área pintada corresponde a la desviación estándar.

La clase de resultados que se obtuvieron de todos los gráficos de cuadros por segundo promedio sobre las distintas resoluciones, los cuales se adjuntan en el anexo A, son muy similares. Luego, otros gráficos muestran relaciones que sirven para concluir la razón de las suposiciones mencionadas en el análisis teórico como lo es la relación entre la cantidad de nodos internos y el rendimiento en tiempo del método.

Como se muestra en el segundo gráfico de la Figura 5.3.2 Se puede ver que existe un aumento en la cantidad de saltos promedio que efectúan los rayos simulados cuando crece el orden de magnitud de la cantidad de nodos internos, lo que también está conectado al rendimiento de manera directa, el cual evidencia una caída de cuadros por segundo al aumentar levemente el número promedio de saltos por rayo.

La dimensionalidad, por otra parte, que se definió como una medida que daba cuenta de la



(a) Número de saltos por rayo promedio según número de nodos internos.

(b) Cuadros por segundo promedio según número de saltos por rayo promedio.

Figura 5.3.2: Gráficos que relacionan los saltos promedio con el número de nodos internos y la tasa de cuadros por segundo promedio para el k^3-tree. Las pendientes de las curvas se pueden ver en más detalle en el anexo A.

superficialidad de la geometría de los modelos, alcanzó valores superiores a lo esperado. Esto es debido a que mientras aumenta la resolución, más definida se vuelve la representación de la información espacial y con ello menos ocurrencias hay de clústeres de vóxeles producto de una voxelización de bajo nivel de detalle, motivo por el cual se esperaba ver una tendencia hacia un valor de dimensionalidad cercano a 2. Sin embargo, como se da a conocer en la Figura 5.3.3, ésta se aproxima hacia valores cercanos a 2.4 pese al aumento de la resolución. Esta diferencia puede adjudicarse a errores introducidos en el algoritmo de voxelizado, pues hay evidencia de casos de mal funcionamiento y apariciones de pequeños artefactos cúbicos recurrentes en los modelos, los cuales afectan a este valor al contener dentro de sí conjuntos macizos de vóxeles.

Por otro lado, en materia de espacio usado por el método, las relaciones entre los valores de los atributos medidos para el k^3-tree sugieren, con los gráficos de la Figura 5.3.4, que el uso de memoria aumenta de manera lineal conforme crece el número de vóxeles, siendo así también para la cantidad de nodos internos.

No obstante, pese a la obtención de resultados que apunten a una complejidad espacial lineal al utilizar el k^3-tree, existen gráficas como la de la Figura 5.3.5 que muestran otra relación a notar. Se observa que conforme va creciendo la cantidad de vóxeles disminuye la cantidad de memoria promedio se emplea por unidad de vóxel, es decir, mientras más vóxeles existan en una escena, menor será la demanda de memoria promedio que supone el añadido de vóxeles nuevos.

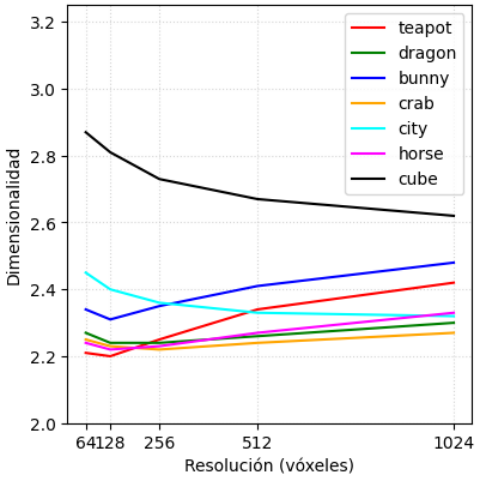


Figura 5.3.3: Gráfico de dimensionalidad.

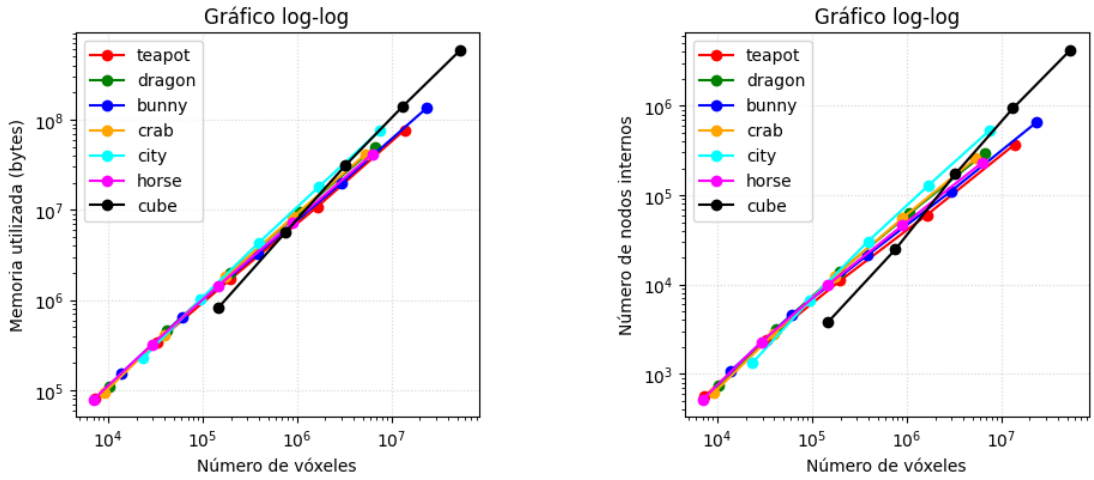


Figura 5.3.4: Gráficos del k^3-tree, de memoria utilizada y de nodos internos respectivamente. Ambos con sus dos ejes en escala logarítmica.

Sin embargo, esto ocurre solo en los datasets simples y de información puramente superficial y, tiene sentido que no exista esta reducción de memoria empleada por vóxel en los datasets *city* y *cube*, pues con el aumento de resolución, se incluyen cada vez más vóxeles en bifurcaciones que no existían antes dado el alto nivel de detalle del primero y la naturaleza repetitiva del segundo.

La gráfica de la Figura 5.3.6 compara directamente el uso de espacio y el tiempo de renderizado de las tres estructuras de datos, promediando los valores medidos para todos los datasets y para cada una de las resoluciones evaluadas.

Finalmente, la Figura 5.3.7 hace explícito el rendimiento del k^3-tree, en tiempo y en espacio, en proporción al del octree. Se puede observar que el rendimiento en términos de tiempo,

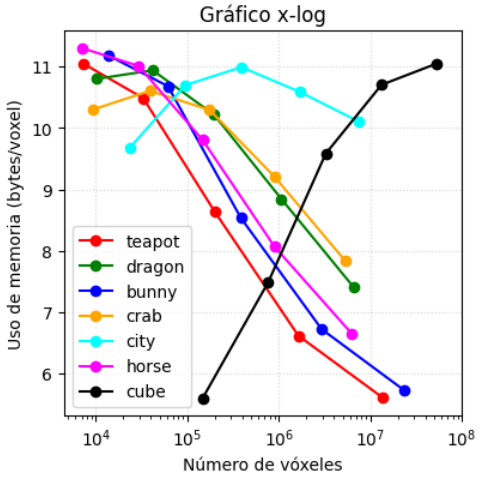


Figura 5.3.5: Gráfico de uso de memoria por vóxel. Eje x en escala logarítmica.

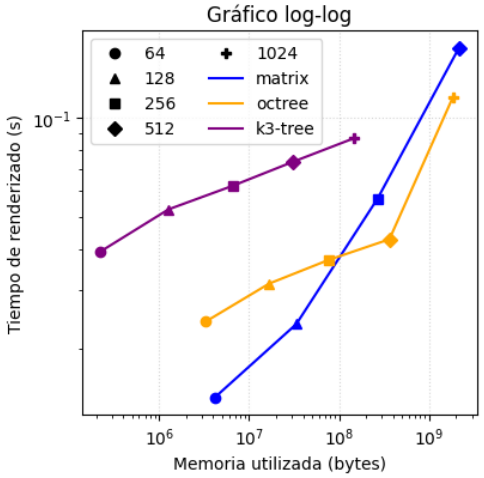
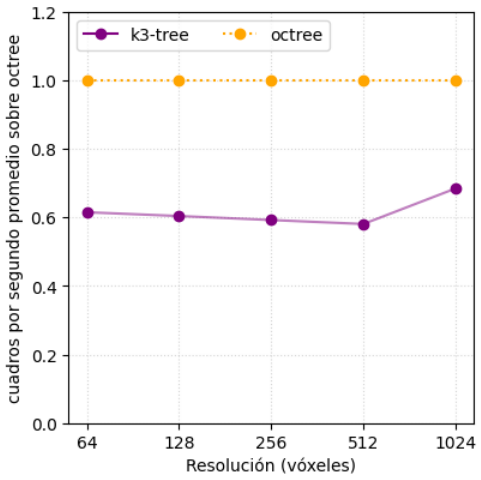


Figura 5.3.6: Trade-off de espacio empleado y tiempo de renderización. Ambos ejes en escala logarítmica.

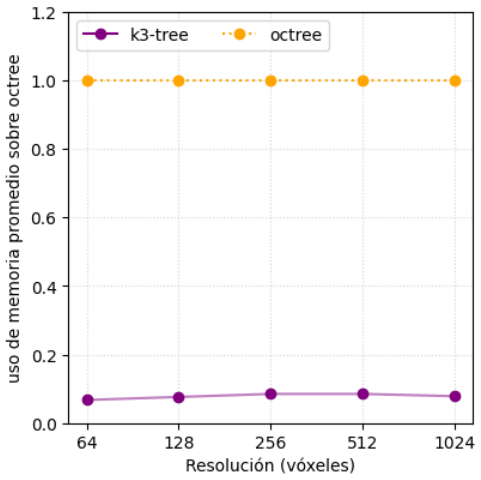
evidenciado en la gráfica de cuadros por segundo promedio, es del 60 % aproximadamente, obteniendo una leve mejora para la última resolución medida, mientras que el uso de espacio promedió aproximadamente un 10 % del uso que hizo el octree.

5.4. Discusión

En términos de espacio, los vóxeles son conocidos por consumir grandes cantidades de memoria, lo que es cierto para datasets de información realmente volumétrica como lo son las imágenes de resonancia magnética, del campo de la medicina. Sin embargo, si se utilizan para representar información de superficies únicamente, y son almacenados con una estructura de datos ad hoc,



(a) Cuadros por segundo promedio k^3-tree.



(b) Uso de memoria k^3-tree.

Figura 5.3.7: Gráficos relativos al rendimiento del octree. Se considera el promedio de todas las pruebas.

el uso de memoria baja en gran medida. Afortunadamente éste es el caso en la renderización en tiempo real, pues para la mayoría de aplicaciones, el tipo de elementos a renderizar son superficies planas que forman objetos huecos, ya que cualquier información volumétrica no causa, por lo general, ninguna diferencia en la experiencia del usuario.

Esto da una idea de la utilidad de esta estructura de datos compacta emparejada con el método visto; tiene un rendimiento menor en términos de tiempo, pero mejora considerablemente el uso del espacio. De todas formas, para utilizar el método en un entorno real de renderización se requiere el uso de la GPU dado que para obtener estos resultados se requirió usar una baja resolución.

Capítulo 6

Conclusiones

El principal motivo por el que el k^3-tree supera al Octree en términos eficiencia de uso de memoria es el hecho de que utiliza punteros en su implementación. No obstante, pese a que es a costa de una mayor demanda de memoria, es posible lograr, en otras implementaciones, una renderización más rica en detalles debido a que la estructura de sus nodos les brinda la capacidad de almacenar sus propios valores tanto de vectores normales que permite que la iluminación no se encuentre estrictamente determinada por la geometría cúbica de los vóxeles, como de contornos que brindan mejor detalle y permiten reducir la cantidad de subdivisiones realizadas en cada octante, además de otros atributos útiles.

6.1. Trabajo futuro

Se buscará llevar a cabo el desarrollo de una versión aún más elaborada del método, que no quede por detrás en materia de detalles; se propone reemplazar el vector de bits en el último nivel con un vector cuyo tipo de dato represente un color u estructura más compleja con el fin de equiparar a los nodos del Octree. Además, se buscará igualar o superar las tasas de fotogramas por segundo alcanzadas por el Octree probando métodos de recorrido de árbol que aprovechen la localidad espacial, así como la técnica de cómputo en lotes también conocida como batch computing, cuyo objetivo es el de minimizar la necesidad de accesos repetidos a memoria inferior en la jerarquía. Se considera además la aplicación de una estrategia de optimización conocida como beam optimization, para reducir aún más los saltos innecesarios en la etapa de recorrido de la estructura de datos. Finalmente, para maximizar el rendimiento del método, se explorará la posibilidad de emplear la capacidad de procesamiento de la GPU.

Estos cambios potenciarían tanto la calidad como el rendimiento de la actual implementación del método de Voxel Ray Casting, el cual cubrió el rol de una prueba de concepto.

Glosario

campo de visión ángulo que delimita el área que se puede percibir del mundo virtual generado en el dispositivo de visualización asociado a la posición del punto de visión. 9

GPU coprocesador dedicado al procesamiento de gráficos u operaciones de coma flotante, para aligerar la carga de trabajo del procesador en aplicaciones como los videojuegos o aplicaciones 3D interactivas. 30

píxel elemento más pequeño de una imágen digital, es un punto de color y se ubica en una grilla uniforme. 5

resolución es el tamaño en píxeles del área de la pantalla donde se reproducirá algún contenido, usualmente especificado con los valores de ancho y alto. 9

Bibliografía

[1] Tomas Akenine-Mller, Eric Haines, and Naty Hoffman. *Real-Time Rendering, Fourth Edition*. A. K. Peters, Ltd., USA, 4th edition, 2018.

[2] N. Gharachorloo, S. Gupta, R. F. Sproull, and I. E. Sutherland. A characterization of ten rasterization techniques. *SIGGRAPH Comput. Graph.*, 23(3):355–368, jul 1989.

[3] Rama Karl Hoetzlein. Gvdb: Raytracing sparse voxel database structures on the gpu. In *Proceedings of High Performance Graphics*, HPG '16, page 109–117, Goslar, DEU, 2016. Eurographics Association.

[4] Randima Fernando. *Chapter 39. Volume Rendering Techniques. GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

[5] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '10, page 55–63, New York, NY, USA, 2010. Association for Computing Machinery.

[6] Gonzalo Navarro. *Compact Data Structures – A practical approach*. Cambridge University Press, 2016. ISBN 978-1-107-15238-0. 536 pages.

[7] Susana Ladra, José R. Paramá, and Fernando Silva-Coira. Scalable and queryable compressed storage structure for raster data. *Information Systems*, 72:179–204, 2017.

[8] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. *SIGGRAPH Comput. Graph.*, 20(4):269–278, aug 1986.

[9] Shota Vashakmadze. Modeling the line: Bresenham's algorithm, 1962–87. *Architectural Theory Review*, 24(3):262–278, 2020.

[10] Kai Xiao, Danny Chen, X Hu, and Bo Zhou. Efficient implementation of the 3d-dda ray traversal algorithm on gpu and its application in radiation dose calculation. *Medical physics*, 39:7619–25, 12 2012.

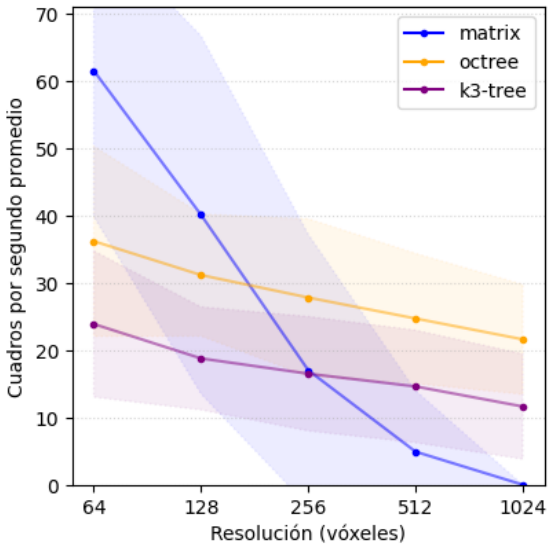
[11] Susana Ladra, Miguel R. Luaces, José R. Paramá, and Fernando Silva-Coira. Space- and time-efficient storage of lidar point clouds. In Nieves R. Brisaboa and Simon J. Puglisi, editors, *String Processing and Information Retrieval*, pages 513–527, Cham, 2019. Springer International Publishing.

Anexos

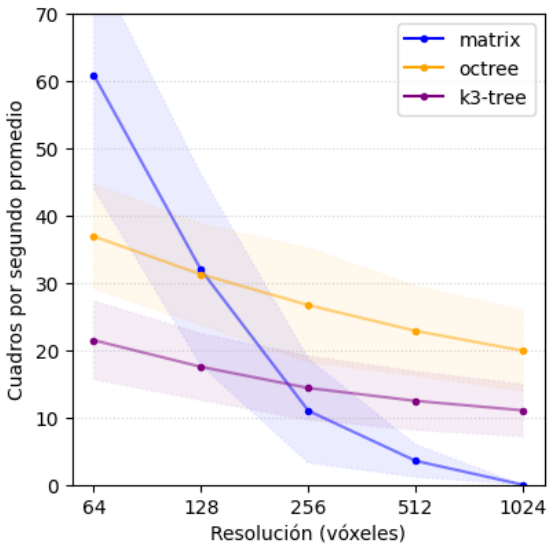
Anexo A

Gráficos

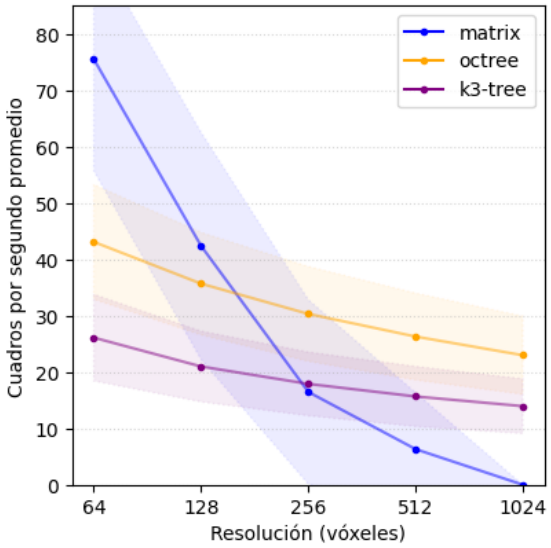
Se adjuntan los gráficos de cuadros por segundo promedio para los todos los modelos del dataset y los gráficos más detallados de la Figura 5.3.2.



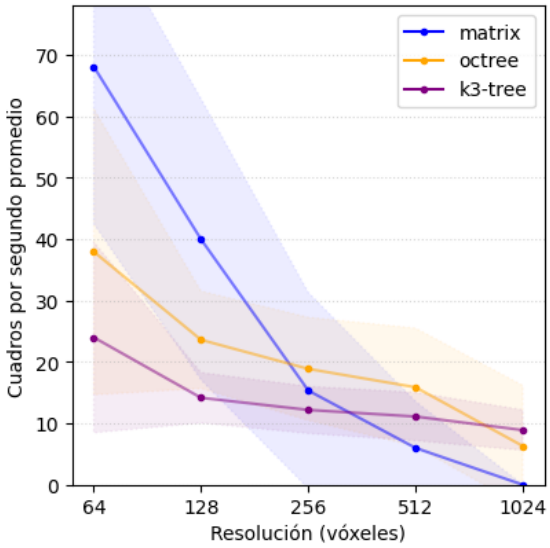
Dataset teapot.



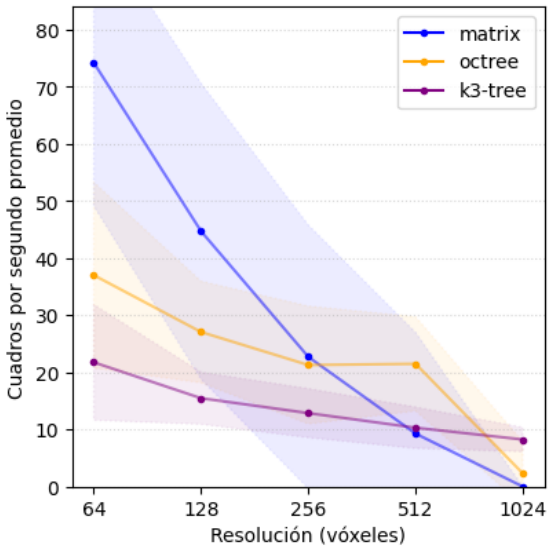
Dataset dragon.



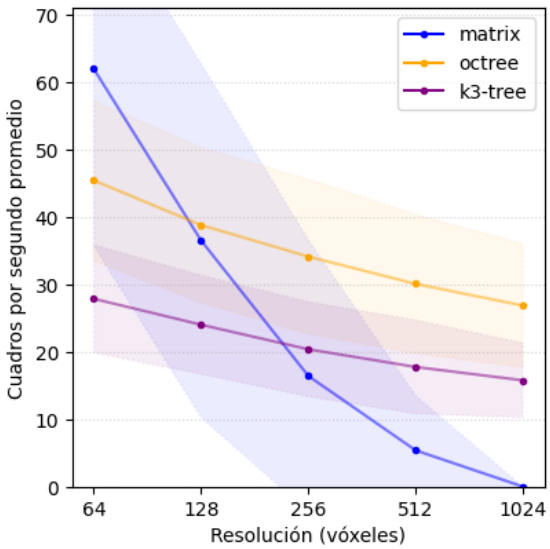
Dataset bunny.



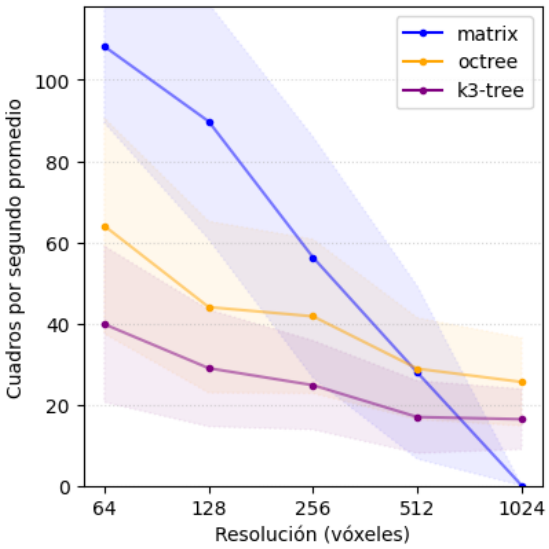
Dataset crab.



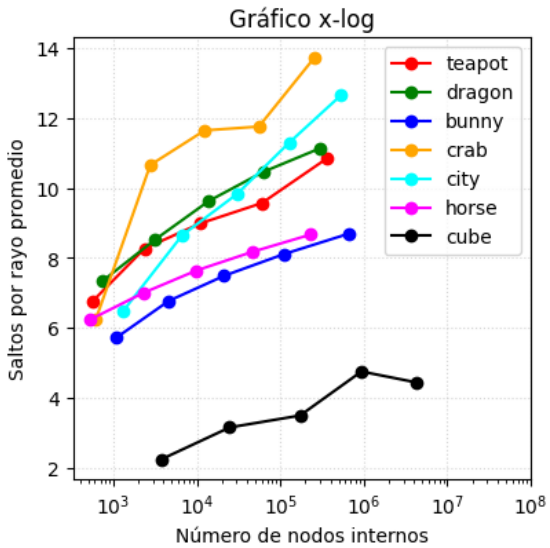
Dataset city.



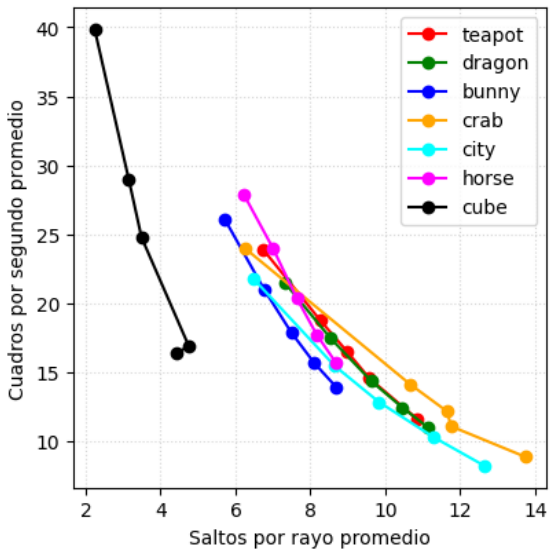
Dataset horse.



Dataset cube.



(a) Número de saltos por rayo promedio según número de nodos internos.



(b) Cuadros por segundo promedio según número de saltos por rayo promedio.

Anexo B

Modelos de prueba

A continuación se muestran los modelos del dataset con capturas del software desarrollado. Para cada modelo se muestra la textura de profundidad renderizada por el programa, esto es, iniciando desde un tono negro, conforme aumenta la distancia total recorrida por un rayo cualquiera, más se acerca al blanco.

Además, al final se adjuntan dos capturas que corresponden a la textura de subdivisión en el trazado de rayos. El color rojo se hace más notable según aumenta la cantidad de saltos realizados por un rayo en su simulación.

Figura B.0.1: Modelo *teapot* (Utah Teapot).

Figura B.0.2: Modelo *dragon* (Stanford Dragon).

Figura B.0.3: Modelo *bunny* (Stanford Bunny).

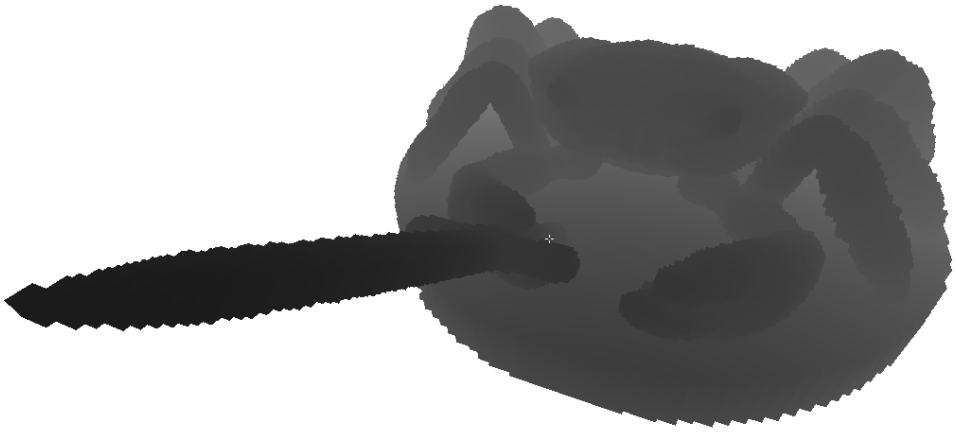


Figura B.0.4: Modelo *crab*.

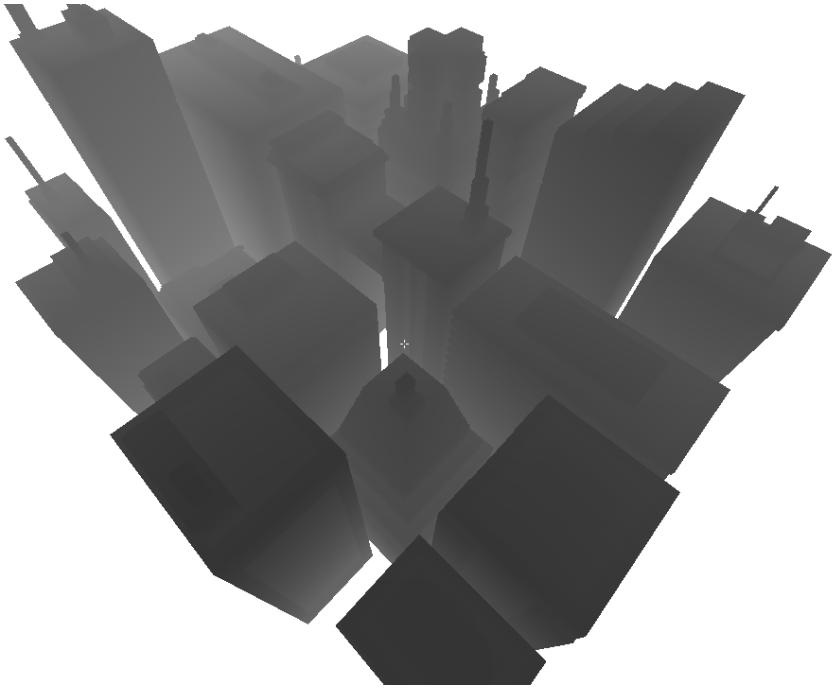


Figura B.0.5: Modelo *city*.

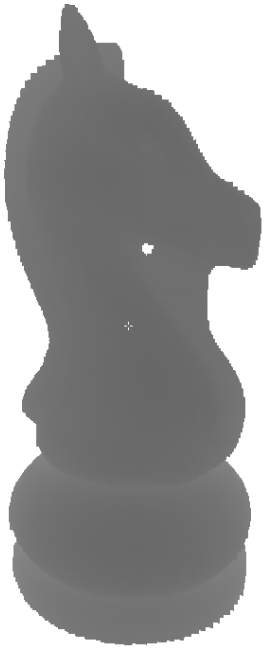


Figura B.0.6: Modelo *horse*.

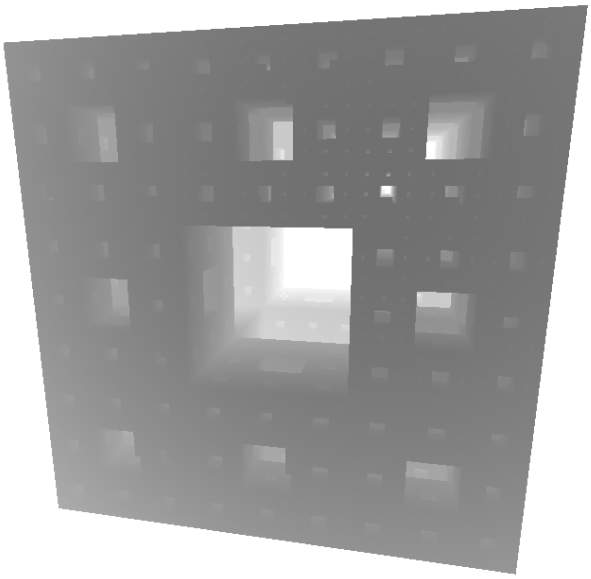


Figura B.0.7: Modelo *cube* (Esponja de Menger).

Figura B.0.8: Subdivisiones del espacio para el modelo *teapot*, resolución 256.

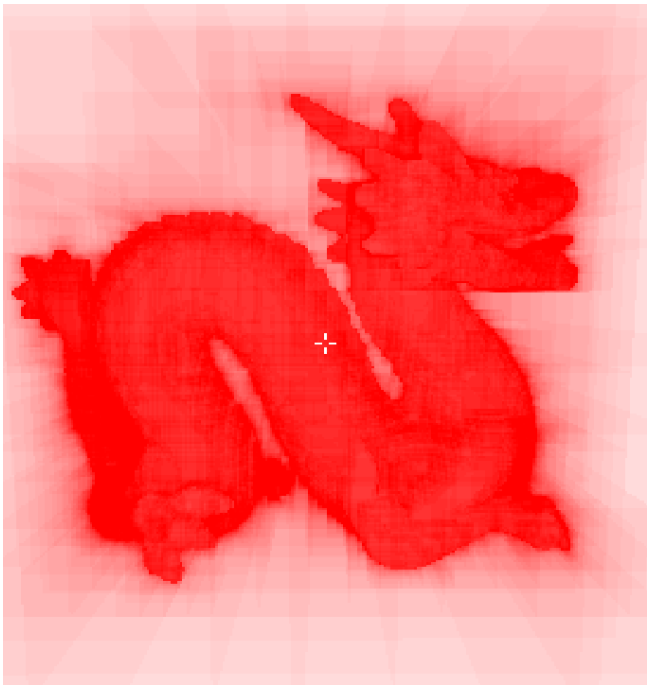


Figura B.0.9: Subdivisiones del espacio para el modelo *dragon*, resolución 256.