UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

VISUALIZACIÓN LIBRE DE MALLAS POR PARTÍCULAS SUAVIZADAS OPTIMIZADA MEDIANTE OCTREE DINÁMICO

TESIS

QUE PARA OPTAR POR EL GRADO DE
DOCTOR EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

PRESENTA:

CÉSAR ADRIÁN VICTORIA RAMÍREZ

TUTOR PRINCIPAL:

DR. EDGAR GARDUÑO ÁNGELES
Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas - IIMAS

Ciudad Universitaria, CD. MX. - Mayo del 2024

PROTESTA UNIVERSITARIA DE INTEGRIDAD Y

HONESTIDAD ACADÉMICA Y PROFESIONAL

De conformidad con lo dispuesto en los artículos 87, fracción V, del Estatuto General, 68, primer párrafo, del Reglamento General de Estudios Universitarios y 26, fracción I, y 35 del Reglamento General de Exámenes, me comprometo en todo tiempo a honrar a ala institución y a cumplir con los principios establecidos en el Código de Ética de la Universidad Nacional Autónoma de México, especialmente con los de integridad y honestidad académica.

De acuerdo con lo anterior, manifiesto que el trabajo escrito titulado "VISUALIZACIÓN LIBRE DE MALLAS POR PARTÍCULAS SUAVIZADAS OPTIMIZADA MEDIANTE OCTREE DINÁMICO", que presenté para obtener el grado de Doctor/a en Ciencia e Ingeniería de la Computación, es original, de mi autoría y lo realicé con el rigor metodológico exigido por mi Programa de Posgrado, citando las fuentes, ideas, textos, imágenes, gráficos u otro tipo de obras empleadas para su desarrollo.

En consecuencia acepto que la falta de cumplimiento de las disposiciones reglamentarias y normativas de la Universidad, en particular las ya referidas en el Código de Ética, llevará a la nulidad e los actos de carácter académico administrativo del proceso de titulación/graduación

Atentamente

César Adrián Victoria Ramírez, 514015002

Contenido

Lista de Tablas

Lista de figuras

Agradecimientos

El presente trabajo no pudo ser realizado sin el apoyo brindado por diversas personas e instituciones, por lo que aprovecho para agradecerles por todo ese apoyo que recibí durante el desarrollo de este proyecto y esta tesis.

Agradezco infinitamente a mis tutores el Dr. Edgar Garduño Ángeles y el Dr. Alfonso Gastélum Strozzi que sin su guía, consejos e ideas jamás hubiera sido posible este proyecto. Encontré en ellos una gran paciencia y conocimientos, espero seguir contando con su apoyo hoy y siempre. Les agradezco su apoyo brindado para realizar una estancia de investigación. También agradezco a los profesores y personal que labora en dicho posgrado que sin ellos nada de esto sería posible. Agradezco su apoyo y orientación brindado para que yo pudiera realizar una estancia de investigación.

Agradezco al laboratorio de Análisis de Imágenes y Visualización en el ICAT-UNAM por brindarme un espacio dentro de sus instalaciones en el cual pude trabajar en este proyecto durante este tiempo.

Agradezco al CONACYT, ahora CONAHCyT, por el apoyo económico que me brindó durante este tiempo con la beca con número 591623.

Resumen

El proyecto que da lugar a la presente tesis desarrolló un sistema de visualización científica orientado a ser usado en simulaciones de cirugía cerebral aprovechando una arquitectura en paralelo usando CUDA® de Nvidia®. Basándose en diferentes trabajos previos acerca del renderizado y visualización científica se construyeron nuevas estructuras de datos que en conjunto con técnicas de ordenamiento y graficación buscan aprovechar al máximo las tecnologías de procesamiento en paralelo.

El sistema presentado busca ser capaz de visualizar los efectos presentes durante una simulación de cirugía cerebral, desde el flujo de los diferentes fluidos hasta la división y corte de tejido cerebral. Se implementaron diferentes técnicas de visualización buscando obtener un compromiso entre la calidad de la visualización y su velocidad de generación de imágenes. Como técnica de renderizado se utilizó *ray tracing*, el cual fue implementado sobre una estructura de datos *octree* el cual es organizado y almacenado en memoria de GPU de manera que también sea posible construirlo, procesarlo y modificarlo de manera paralela. La estructura de datos octree es usada en conjunto con una metodología de organización espacial (clave Morton) con la cual es posible establecer relaciones directas entre las posiciones de los datos en el espacio y su almacenamiento en memoria. Con estas relaciones es posible un acceso directo a la información contenida en el octree, además de que asegura una independencia entre los datos lo cual ayuda al procesamiento en paralelo.

Para obtener una visualización más orgánica se utilizaron meta-esferas, las cuales gracias a su comportamiento permiten una simulación de fluidos y otras estructuras orgánicas de manera más natural. Haciendo uso de las relaciones creadas gracias a las estructuras de datos se describe una metodología que modifica y actualiza la información contenida por el sistema de visualización, lo cual implica una reestructuración de los datos contenidos. Con esta metodología es posible modificar la información de entrada y que estas modificaciones se reflejen en las visualizaciones en tiempo real.

Por último se realizaron pruebas sobre la velocidad y desempeño del sistema, comparando las técnicas con diversas implementaciones actuales de visualización.

Capítulo 1

Introducción

Muchas áreas de la ciencia tienen como necesidad el poder representar de manera visual los datos que obtienen mediante los diferentes experimentos y mediciones que realizan, por lo cual es importante contar con métodos de visualización versátiles y eficientes. Al mismo tiempo, en diversas áreas de la investigación científica, los detalles y patrones presentes dentro de la información a visualizar pueden ser interpretados únicamente cuando existe un nivel de resolución "suficientemente" alto. Una visualización detallada y eficiente ayudará a los usuarios a interpretar dichos patrones, especialmente cuando existe la capacidad de interacción con las visualizaciones finales [1]. La visualización de datos no es exclusiva del área científica y es una parte esencial en las industrias de la publicidad y, en particular, del entretenimiento digital (cine, videojuegos, publicidad, realidad virtual) la cual ha sido una gran promotora de los desarrollos en métodos e investigación en graficación por computadora y en visualización al mismo tiempo de impulsar el desarrollo de *hardware* y *software*.

En este trabajo estamos interesados en la visualización de datos en tres dimensiones (3D o tri-dimensionales) y existen diferentes metodologías para su visualización, pero ellas pueden ser divididas, básicamente, en dos tipos: renderizado por superficie (*surface rendering*) y renderizado por volumen o volumétrico (*volume rendering*). El primero consiste en aproximar una superficie por medio de una malla poligonal que se ajuste lo mejor posible a los datos de variedad (*manifold*). Esta forma de visualizar datos 3D es probablemente la más común y popular debido a que el *hardware* gráfico y los métodos de graficación por proyección se han optimizado para este tipo de renderizado, permitiendo generación de imágenes en tiempo real [2]; sin embargo, estos avances han priorizado la velocidad de la representación visual sobre su precisión, lo que provoca

que en ocasiones las visualizaciones sean poco realistas. *Volume rendering*, por otra parte, tiene como objetivo realizar una proyección bi-dimensional (2D) de todos los elementos espaciales (vóxeles regulares o no) que forman una imagen tri-dimensional. En la actualidad existen múltiples trabajos e investigaciones que proponen métodos para ambos enfoques, buscando optimizar las diferentes técnicas para así obtener salidas gráficas cada vez más realistas con una cantidad de datos en constante aumento. Es importante señalar que, tradicionalmente, realizar el renderizado de una imagen por medio de *volume rendering* implicaba un alto consumo de recursos computacionales por la gran cantidad de datos a visualizar y el bajo rendimiento de técnicas como *raytracing* en dichas condiciones, ocasionando renderizados lentos sin interacción con el usuario final; sin embargo, el desarrollo de *hardware* más sofisticado y poderoso aunado a técnicas nuevas de graficación por computadora han permitido recientemente que se pueda realizar *volume rendering* en tiempo real [3, 4].

Debido a los beneficios de la visualización de datos 3D se ha fomentado el desarrollo de simulaciones de procedimientos médicos, la cual es actualmente un área en constante desarrollo ya que ofrece el potencial de proporcionar una herramienta poderosa para poder diagnosticar enfermedades, desarrollar nuevos tratamientos, realizar seguimiento de tratamientos y, de manera relevante, permitir el entrenamiento a nuevos médicos sin necesidad de estar en contacto directo con un paciente. Sin embargo, los datos obtenidos a través de estudios médicos y sus simulaciones son cuantiosos y complejos por lo que es necesario tener algoritmos y metodologías que puedan procesar de manera eficiente estas cantidades masivas de información. La salida gráfica es una de las partes principales para una simulación y visualización de datos, debido a que proporciona al usuario información visual que le permitirá tomar las decisiones adecuadas, así como desarrollar sus habilidades. En general, entre mejor sea una visualización mayor será el grado de entendimiento de los datos o mayor será el grado de inmersión en la simulación y esto conllevará a un mejor entrenamiento en el caso de entrenadores médicos, diagnostico y planeación [5]. En este aspecto, la simulación gráfica de fluidos es un reto y la opción de renderizado usando superficies a través de mallas ha tenido poco éxito debido a la forma en que los fluidos interactúan entre sí; para el caso de simulaciones médicas, el comportamiento orgánico representa un reto, sobre todo cuando se desea realizar renderizado en tiempo real.

La visualización de datos no se restringe a ambientes de simulación, también puede ser implementada en ambientes clínicos en los cuales es necesario mejorar la presentación de datos pre-existentes con la finalidad de que los expertos puedan tomar mejores

decisiones de tratamiento para los pacientes. Un ejemplo ampliamente usado en la mayoría de las clínicas son las imágenes de ultrasonido, si bien estas han sido usadas desde hace varias décadas como una de las principales herramientas de diagnostico, es posible mejorar su visualización realizando reconstrucciónes de imágenes 3D usando un conjunto de imágenes 2D de ultrasonido. Esto con la finalidad de ofrecer al experto una percepción espacial mayor de las anormalidades que se puedan presentar. La velocidad de estas reconstrucciones, si bien no son tan determinantes como en una simulación, poseer una alta velocidad puede ofrecer muchas ventajas en ambientes aplicados donde no se dispone de mucho tiempo con el paciente o en los cuales es necesario repetir el procedimiento múltiples veces hasta alcanzar el resultado deseado. Las reconstrucciones de imágenes 3D en tiempo real pueden ofrecer una ventaja considerable en ambiente pre operativos y clínicos.

En esta tesis presentamos diferentes metodologías que en conjunto permiten la visualización de la simulación gráfica de sangre que se basa en el uso de estructuras de datos arbóreas *octrees* con ordenamiento y codificación con claves Morton y funciones base tipo meta-esferas (*metaballs*). Debido a que estas metodologías buscan como objetivo acelerar el procesamiento de datos espaciales ellas pueden ser aplicadas a cualquier ambiente 3D, por ello se presenta la implementación que se realizó en algoritmos de reconstrucción 3D para imágenes de ultrasonido adquiridas manualmente (*freehand*).

La tesis esta organizada de la siguiente manera, en el Capítulo 2 se muestran los conceptos y las técnicas usadas para el desarrollo de esta tesis, el Capítulo 3 detalla la implementación eficiente en *hardware* de *octrees* y su codificación con claves Morton; además, se presenta la definición utilizada de *raytracing* y *metaesferas*. El Capítulo 4 muestra a detalle la aplicación de los métodos para la reconstrucción 3D de imágenes de ultrasonido. El Capítulo 5 presenta la aplicación de la metodología para la simulación de líquidos. Finalmente, el Capítulo 6 presenta resultados, conclusiones y posibles direcciones futuras.

Capítulo 2

Conceptos Básicos

En este capítulo se exploraran los conceptos básicos para el desarrollo del proyecto que se presenta en esta tesis.

2.1 Imágenes Digitales

En este trabajo representamos imágenes digitales como funciones $\upsilon : V \to \mathbb{R}$ en donde V es un subconjunto finito de $G_\Delta = \left\{\Delta \boldsymbol{k} \middle| \boldsymbol{k} \in \mathbb{Z}^n\right\}$, en donde Δ es un número real positivo que se conoce como *distancia de muestreo* y es un valor determinado por el equipo generador de imágenes; por ello, obviamos dicho valor a menos que sea necesario hacerlo explicito. En este trabajo estamos interesados en los casos $n = 2$ y $n = 3$; para el caso de $n = 2$, hablamos de imágenes bi-dimensionales o 2D y, obviamente, para el caso $n = 3$, se trata de imágenes tri-dimensionales o 3D (también se les conoce como volúmenes). Es importante hacer notar que el vecindario de Voronoi $\mathcal{V}_{\boldsymbol{x}}$asociado a un punto $\boldsymbol{x}$ de G_Δ para el caso de $n = 2$ es una área cuadrada o un píxel cuadrado, en este trabajo hablaremos simplemente de píxel a menos que exista confusión. De manera similar, para el caso de $n = 3$ el vecindario asociado al punto $\boldsymbol{x}$ es un volumen, o vóxel, cúbico; en este trabajo hablaremos simplemente de vóxel a menos que exista confusión.

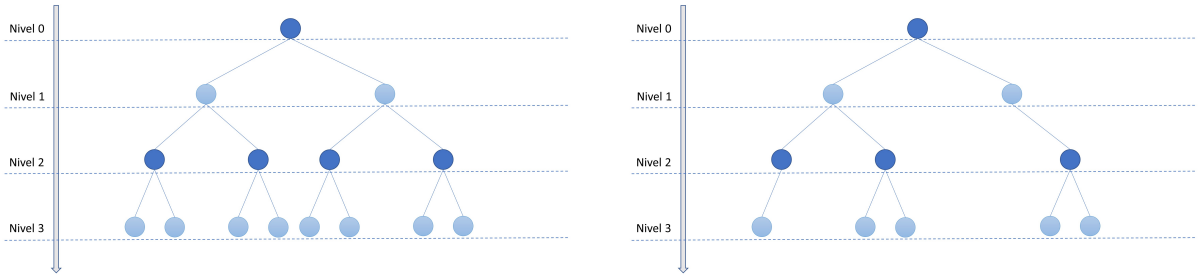
2.2 Claves Morton

En ciencias de la computación y matemáticas, una Clave Morton (*Morton Key*), también conocida como ordenamiento Morton, se refiere a una técnica que mapea datos multidimensionales a una sola dimensión preservando la localidad de los puntos de datos; este

		X							
		0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
Y	0 000	000000	000001	000100	000101	010000	010001	010100	010101
	1 001	000010	000011	000110	000111	010010	010011	010110	010111
	2 010	001000	001001	001100	001101	011000	011001	011100	011101
	3 011	001010	001011	001110	001111	011010	011011	011110	011111
	4 100	100000	100001	100100	100101	110000	110001	110100	110101
	5 101	100010	100011	100110	100111	110010	110011	110110	110111
	6 110	101000	101001	101100	101101	111000	111001	111100	111101
	7 111	101010	101011	101110	101111	111010	111011	111110	111111

Figura 2.1: Distribución de la Clave Morton al entrelazar los valores binarios de las coordenadas de un punto $\boldsymbol{p} = (p_1, p_2)$ en 2D. Se puede observar el patrón en Z formado cuando se busca encontrar un orden en las claves Morton generadas.

mapeo es biyectivo, lo cual implica que es posible regresar a los datos multidimensionales a partir de los respectivos mapeos a datos puntuales. Estos mapeos fueron propuestos en 1966 por Guy Macdonald Morton [6] para ordenar la secuenciación de archivos en datos geodésicos. Actualmente son ampliamente usados para el ordenamiento de información multidimensional ya que simplifican la búsqueda espacial preservando las propiedades espaciales. Estas propiedades hacen que esta codificación sea óptima para el uso de *quadtrees* y *octrees*. Una clave Morton $\mathcal{M}$ mapea las coordenadas, con valores enteros positivos, de un punto $\boldsymbol{p}$ multidimensional a un entero positivo ($\mathbb{Z}_+^n \rightarrow \mathbb{Z}_+$), esto se logra a través de la combinación y entrelazado de los bits de las coordenadas p_j, $1 \leq j \leq n$, del punto. En este punto haremos una aclaración en nuestra notación, por convención usaremos las coordenadas de un punto $\boldsymbol{p} \in \mathbb{R}^2$ como (p_x, p_y) y de uno $\boldsymbol{p} \in \mathbb{R}^3$ como (p_x, p_y, p_z); ocurre algo similar para las coordenadas discretas $\mathbb{Z}^2$ y $\mathbb{Z}^3$. Este tipo de mapeo aplicado a coordenadas en el espacio, se puede observar que el orden de las claves Morton con respecto a las coordenadas en el espacio obtienen un patrón en forma de Z, líneas azules, por lo qué la clave Morton también se conoce como ordenamiento Z, ver Figura 2.1.



(a) (b)

Figura 2.2: Esquemas que muestran (a) un árbol binario completo y (b) un árbol binario parcial.

2.3 Estructura de datos *octrees*

Existen diferentes estructuras de datos usadas para la representación de la información de una manera ordenada. Una de las más usadas son las gráficas de tipo árbol, cuyos nodos y aristas están organizadas que forman una estructura jerárquica cuya organización simula un árbol: un nodo raíz y sub-gráficas (sub-árboles), como hijos, representadas por un conjunto de nodos enlazados por aristas o ramas. En esta representación, las relaciones entre los nodos de un árbol forman una representación jerárquica de la información contenida representada por la estructura de datos. En un árbol, hay tres tipos de nodos: nodo raíz, nodos hoja y nodos interiores. Al mismo tiempo, los nodos pueden ser nodos padre o nodos hijo, un nodo padre es superior en jerarquía a los nodos que serán sus nodos hijos y ambos deben compartir una conexión; cada nuevo grupo de hijos se considera un nuevo nivel de profundidad en el árbol. Los nodos que no tienen hijos propios son llamados nodos hoja y el nodo que se encuentra en el nivel superior (nivel 0) es el nodo raíz, la Figura 2.2 muestra los esquemas de árboles binarios, se tienen dos hijos por cada nodo padre, uno completo y uno parcial.

Existen varios tipos de árboles (gráficas, un conjunto de nodos y vértices, no-dirigidas y acíclicamente conectadas), pero en este trabajo nos interesan los árboles conocidos como *octrees*. Un *octree* $\mathcal{O}$, o árbol octal, es un tipo de árbol donde cada nodo padre tiene entre 0 y 8 nodos. En varias aplicaciones es necesario representar y subdividir una región del espacio $\mathbb{R}^3$ y una de las formas más comunes para llevar a cabo dichas tareas es comenzar encerrando la región a ser representada con un cuboide "mínimo". A continuación, el cuboide se divide en ocho regiones, octantes, con las mis-

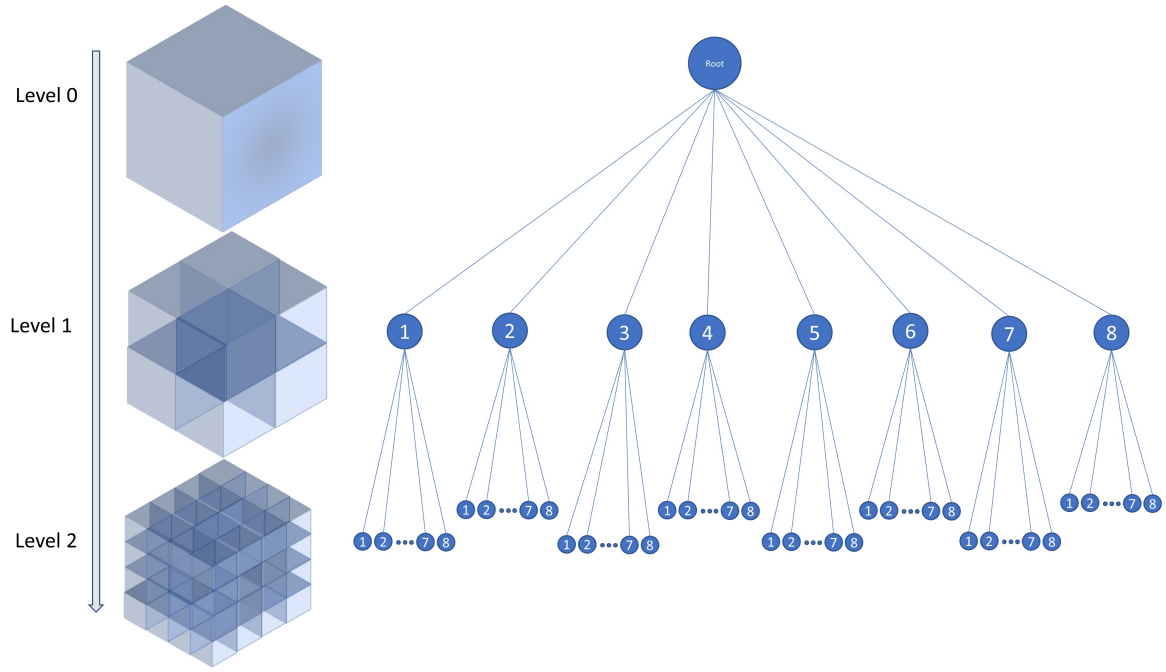
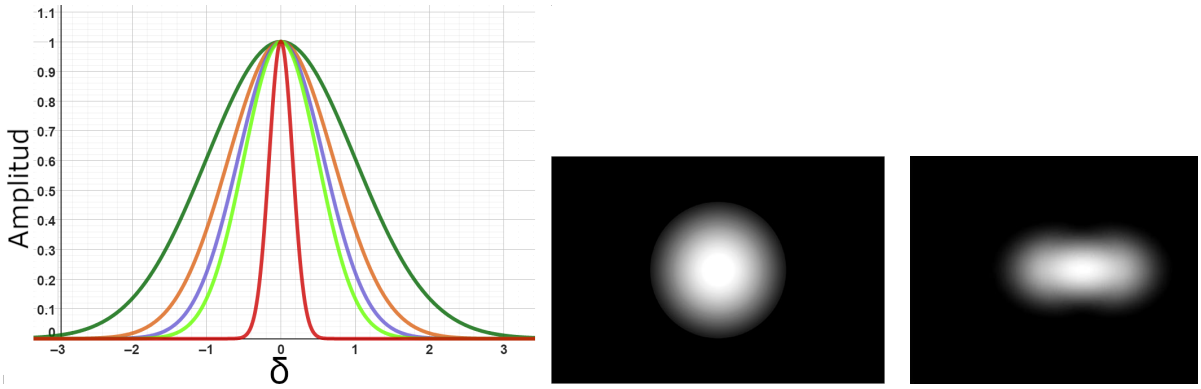


Figura 2.3: Representación espacial y estructural de un *octree*, cada nodo padre tiene ocho hijos que equivalen respectivamente a cada octante en que se divide el espacio.

mas dimensiones. En este momento, el cuboide puede representarse como un nodo o_r, el nodo raíz, y cada octante se puede representar como un nodo hoja o_f; creando un árbol de un sólo nivel. Cada octante, una subregión del cuboide y del espacio $\mathbb{R}^3$, puede subdividir respectivamente en octantes, de menor tamaño, lo que transforma a los nodos hoja o_f en nodos internos o y agrega un nuevo nível al árbol y nuevos nodos hoja al *octree* $\mathcal{O}$. Este proceso puede continuar hasta alcanzar un nivel deseado de subdivisiones (por ejemplo, las subdivisiones corresponden a los vóxeles en una imagen tri-dimensional V) o se alcanza un nivel máximo para el *octree* (se alcanza el número máximo de subdivisiones, o la resolución, del *octree*). Claramente, este tipo de estructura de datos es muy popular en campos en las que es necesario representar "volúmenes" (regiones del espacio $\mathbb{R}^3$), tales como gráficas por computadora y procesamiento de imágenes. Al relacionar los nodos de un *octree* con octantes es posible establecer una organización jerarquíca del espacio 3D, permitiendo una búsqueda rápida de la información en una escena (región 3D Euclideana de interés), ver Figura 2.3.

En la práctica, uno de los inconvenientes principales con los *octrees* es la cantidad de memoria utilizada para almacenar los nodos en una implementación en *software*, cada nivel agregado al *octree* hace que la cantidad de nodos crezca exponencialmente;



(a) (b) (c)

Figura 2.4: Visualización de (a) varios perfiles de *metaballs* en los que varia $\delta(a, r)$, para el radio de influencia a y la distancia al centro de la *metaball* r, (b) la *metaball* con δ igual a 3.4 y (c) la fusión de varias *metaballs* cercanas que entran en contacto creando una forma suave.

por lo que, si existen subregiones del espacio 3D que no tienen información de interés, se estará solicitando el almacenamiento de regiones de memoria de forma innecesarias. Por lo tanto, para representar regiones 3D con varios niveles de subdivisión se puede demandar una gran cantidad de memoria para almacenar una estructura completa y, en algunos casos, con regiones sin utilizar. Una versión alternativa de los *octrees* son los *sparse octrees* los cuales solo almacenan los nodos que contengan información, dejando los demás nodos sin definir, o inexistentes, con el propósito de no usar memoria de manera innecesaria.

2.4 *Metaballs*

Se le conoce como *metaballs*, o meta esferas, a la técnica propuesta por Jim Blim en 1982 [7] y utilizada para visualizar la interacción de modelos orgánicos n-dimensionales. Típicamente, esta técnica es utilizada para la visualización de la interacción entre objetos tri-dimensiones aunque ha sido muy usada en visualización de objetos en dos dimensiones. Las *metaballs* son funciones, esfericamente simétricas, que transitan suavemente de uno a cero y que se encuentran centradas en un punto en el espacio, ver las figuras 2.4a y 2.4b. Para realizar el modelado de objetos, las *metaballs* son utilizadas de la siguiente forma

$$f(\boldsymbol{x}) = \sum_{j=1}^{N} c_j b_j(\boldsymbol{x}), \tag{2.1}$$

donde $\{c_j\}$ es el conjunto de pesos, o coeficientes, $\{b_j\}$ es el conjunto de *metaballs* y N es un número entero positivo; el conjunto de puntos $\left\{\boldsymbol{q}_j\right\}$ es en donde se encuentran centradas las *metaballs*. La combinación lineal de *metaballs* permite que éstas se "fusionen" para crear formas suaves. Cuando dos *metaballs* se combinan, sus funciones de distribución son evaluadas para obtener un valor de distribución total, en la Figura 2.4c se puede observar el resultado de la combinación líneal de dos *metaballs* iguales cuyos centros se encuentran separadas sobre el eje horizontal.

Sin embargo, en la modalidad más empleada en graficación por computadora se utilizan superficies para representar objetos y es por ello que es deseable obtener dicho tipo de representaciones a partir de (2.1); dicha representación se puede obtener a través del uso de iso-superficies como sigue

$$S = \{\boldsymbol{x} | f(\boldsymbol{x}) = \tau\} = \left\{ \boldsymbol{x} \middle| \sum_{j=1}^{N} c_j b_j (\boldsymbol{x}) = \tau \right\}, \tag{2.2}$$

en donde τ es el umbral que determina los puntos que forman la superficie que puede ser renderizada. Por lo tanto, es posible visualizar la fusión entre dos *metaballs* de una manera orgánica (similar al comportamiento de fluidos viscosos) definiendo un umbral τ y encontrando la iso-superficie correspondiente. En la Figura 2.5 se muestra la interacción de dos *metaballs* al entrar en contacto: se utiliza el mismo umbral para mostrar lo que ocurre cuando la distancia de sus centros disminuye.

En la literatura se han hecho muchas propuestas para las funciones base b de (2.1) y de ellas abundaremos en la Sección 3. Como ejemplo, para el caso 2D podemos considerar una función base b definida como

$$\frac{a^2}{(x - q_x)^2 + (y - q_y)^2} \leq 1, \tag{2.3}$$

en donde a es el radio de la función base y $\boldsymbol{q} = (q_x, q_y)$ representa el punto en donde se encuentra centrada. Evidentemente, esta función es la ecuación de un círculo que incluye todos los puntos interiores (aquellos que se encuentran dentro de su perímetro). Es posible incluir un umbral en la ecuación anterior de la siguiente manera

$$\frac{r^2}{(x - q_x)^2 + (y - q_y)^2} \leq \tau. \tag{2.4}$$

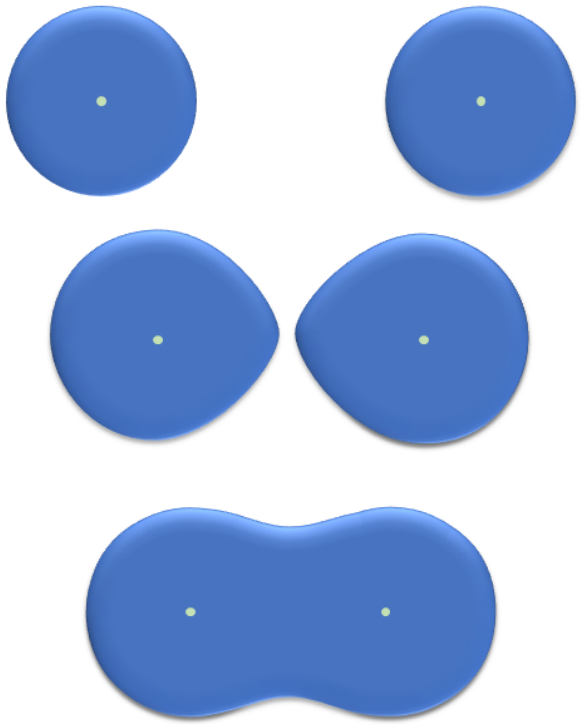


Figura 2.5: *Metaballs* fusionándose al entrar en contacto, creando una forma suave.

Por lo tanto, para obtener funciones suaves a partir de la ecuación anterior, se resuelve

$$\left(f\left(\boldsymbol{p}\right) = \sum_{j=1}^{N} \frac{r_i^2}{\left\|\boldsymbol{p} - \boldsymbol{q}_j\right\|} \right) \leq \tau, \tag{2.5}$$

en donde $\left\|\boldsymbol{p} - \boldsymbol{q}_j\right\|$ es la norma ℓ_2 entre el punto $\boldsymbol{p}$ y el centro de la j-ésima *metaball*. Por lo que la ecuación anterior se reduce a

$$\left(\sum_{j=1}^{N} \frac{r_j^2}{d_i\left(\boldsymbol{p}, \boldsymbol{q}_j\right)} \right) \leq \tau. \tag{2.6}$$

La Figura 2.6 muestra la visualización de *metaballs* individuales y un objeto con uniones suaves que simulan la fusión de las funciones base para un valor de τ fijo.

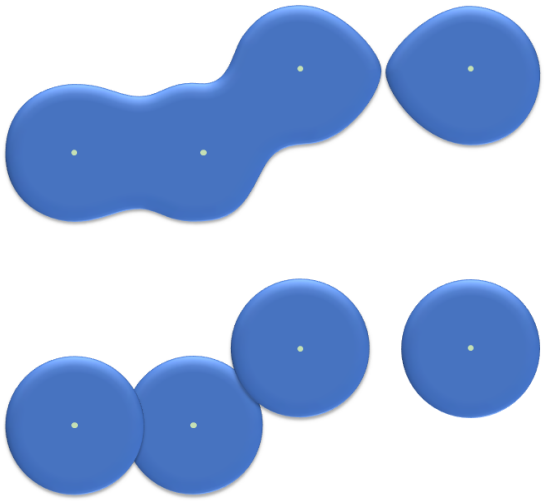


Figura 2.6: Arriba: *Metaballs* bidimensionales fusionándose, graficadas utilizando (2.6). Abajo: Graficación de (2.8), sin considerar todas las *metaballs* en la escena se obtienen círculos independientes.

2.5 *Raytracing*

En graficación por computadora *raytracing* se considera uno de los métodos que genera imágenes de alta calidad, es constantemente presentado como uno de los mejores medios fisicamente precisos que producen imagenes fotorealistas [8]. La idea básica del método consiste en "enviar" haces de luz virtuales desde el observador hacia los objetos de la escena y encontrar las intersecciones entre los rayos y las diferentes superficies; para este modelo, los fotones see consideran partículas y, por ello, los rayos son modelados como líneas rectas. Para entender el funcionamiento de *raytracing*, se puede recurrir al método de *Raycasting*, el cuál es un caso particular de *raytracing* y es uno de los algoritmos más básicos para la renderización de imágenes. El termino fue acuñado por vez primera en el entorno de gráficos por computadora por Roth Scott en 1982 [9]. Este algoritmo consiste en el trazado de rayos desde un punto $\boldsymbol{c}$, la cámara, a través de un plano de imagen Π_I, caracterizado por la normal $\vec{\boldsymbol{n}}_\Pi$, a una escena tridimensional Ω. El plano Π_I está discretizado usando la imagen 2D $\Theta_\Delta = \{\Delta\boldsymbol{k} | \boldsymbol{k} \in \mathbb{Z}^2 \text{ y } 1 \leq k_1 \leq M_\Theta, 1 \leq k_2 \leq N_\Theta\}$, donde M_Θ y N_Θ son números enteros positivos que representan las dimensiones de la imagen que discretiza al plano y Δ es un número real positivo que representa la distancia entre los píxeles. En el modo más básico, el algoritmo de *raycasting* "lanza"

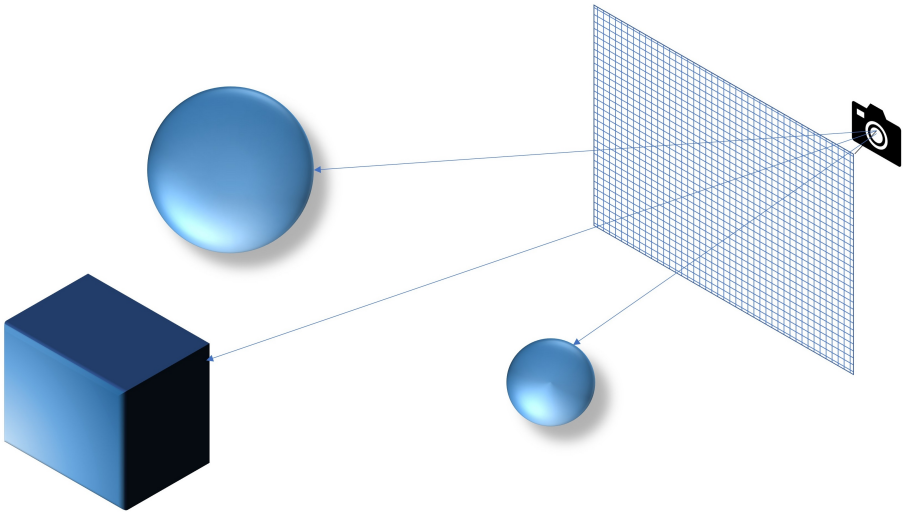


Figura 2.7: Figura que ejemplifica el algoritmo de *raycasting* sobre figuras geométricas. Los rayos trazados desde la cámara buscan determinar el objeto con el que tienen intersección.

un rayo $\boldsymbol{r}$ por cada elemento de Θ_Δ de la siguiente forma

$$\boldsymbol{r} = \boldsymbol{k} + \alpha\ \vec{\boldsymbol{o}}, \tag{2.7}$$

en donde $\boldsymbol{k} \in \Theta_\Delta$, $\vec{\boldsymbol{o}}$ es un vector de dirección (esto es, un vector normalizado), α es un número real y $\langle \vec{\boldsymbol{o}}, \vec{\boldsymbol{n}} \rangle = 0$ (los vectores son perpendiculares). En este modelo, cuando α es igual a cero, el rayo se encuentra sobre el plano y cuando α es mayor a cero, la distancia entre la posición del rayo $\boldsymbol{r_k}$ y los objetos de la escena Ω se reduce. Para cada rayo $\boldsymbol{r_k}$, se busca determinar las intersecciones que pueda tener con los objetos en Ω y en caso de encontrar intersecciones, se busca cual es la intersección más cercana (α_{min}). El objeto asociado con dicha distancia α_{min} es el que será visible desde el punto $\boldsymbol{c}$ a través del píxel $\boldsymbol{k}$. En el punto de intersección entre el rayo y el objeto correspondiente se realizan varios cálculos que determinarán el color que será asignado al píxel $\boldsymbol{k}$, ver Figura 2.7.

El algoritmo de *raytracing* es una extensión del algoritmo *raycasting*; sin embargo, cuando se encuentra la primera intersección, α_{min}, para el rayo $\boldsymbol{r_k}$ se calcula la reflexión y refracción que ocurre en ese punto utilizando leyes físicas (por ejemplo, Ley de Snell) con base en las propiedades del material del objeto intersecado. El rayo que potencialmente es reflejado, o refractado, (rayos secundarios) a partir de ese punto vuelve a generar una búsqueda de alguna intersección con algún objeto en la escena;

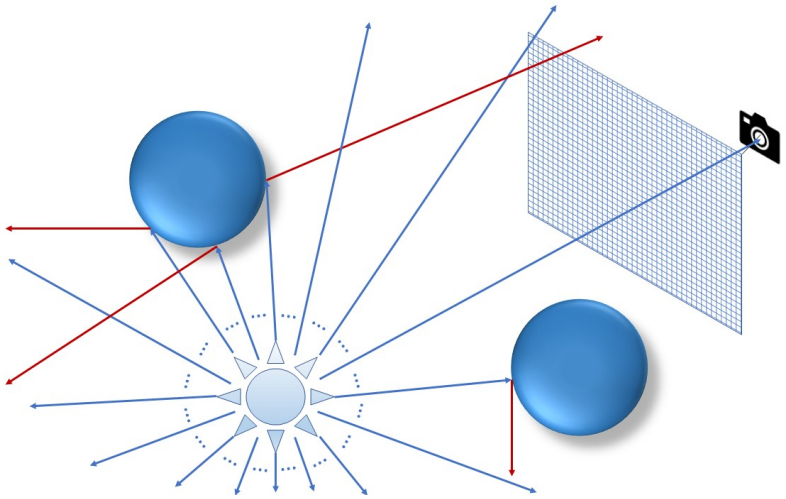


Figura 2.8: Rayos emitidos por una fuente de iluminación. La cantidad de rayos emitidos tiene puede tender a infinito y solo algunos de ellos colisionarán con la cámara, los demás se perderán. En rojo, rayos reflejados al colisionar con un objeto en la escena.

este proceso continua hasta que se llega a un número máximo, pre-determinado, de rayos secundarios, o iteraciones, o hasta que la luz pierde energía. El color asignado al píxel $\boldsymbol{k} \in \Theta_\Delta$ es la composición de las intersecciones de todos los rayos secundarios generados a partir del rayo original $\boldsymbol{r_k}$. El valor que se asigna al píxel $\boldsymbol{k}$ va a depender del modelo físico que se utilice para la interacción luz - materia; por ejemplo, se puede utilizar refracción, esparcimiento o sombras suaves. Sin embargo, en la actualidad es poco común encontrar modelos que consideren la naturaleza dual de los fotones.

Es importante hacer notar que, en principio, en *raytracing* cuando la escena tiene fuentes de iluminación (por ejemplo, lamparas) los rayos son emitidos desde ellas en "todas" las direcciones determinadas por la geometría de las fuentes. Estos rayos potencialmente intersecan con los objetos en la escena generando rayos secundarios que, potencialmente, pueden llegar a intersecar con el plano Π_I (un proceso conocido como *forward raytracing*). Sin embargo, este esquema no es imposible de seguir en la práctica porque el número de rayos emitidos por una fuente de iluminación es infinito y, adicionalmente, de esos rayos son muy pocos los rayos secundarios que eventualmente llegan al plano Π_I, ver la Figura 2.8. Por lo tanto, se opta por un enfoque invertido.

La técnica de *raytracing* que se utiliza es conocida como *backward raytracing* y funciona de la siguiente manera. Como hemos mencionado, se emite un rayo $\boldsymbol{r}$ por cada píxel $\boldsymbol{k} \in \Omega_\Delta$ hacia la escena Ω. Nos referiremos al rayo emitido desde el píxel $\boldsymbol{k}$ como $\boldsymbol{r}_{\boldsymbol{k}}^{(0)}$ por razones que quedaran claras más adelante. En caso de que el rayo $\boldsymbol{r}_{\boldsymbol{k}}^{(0)}$

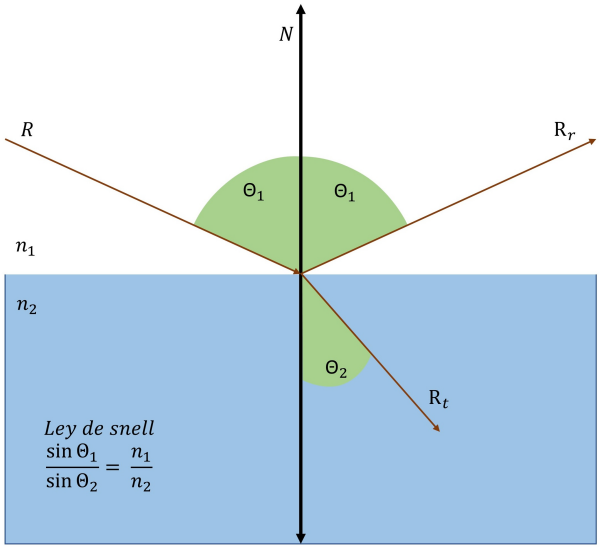


Figura 2.9: Representación de la ley de snell para calcular rayos refractados y reflejados sobre una superficie. n_1 y n_2 corresponden a los índices de refracción de los materiales. R corresponde al rayo incidente, R_t corresponde al rayo refractado, R_r corresponde al rayo reflejado y N a la normal de la superficie.

interseque con un objeto de la escena, en ese punto de intersección $\boldsymbol{p}^{(0)}$ se realiza un cálculo con base en un modelo de interacción luz con materia, como la Ley de Snell (ver Figura 2.9.). El resultado de dicho cálculo puede producir hasta M rayos secundarios $\boldsymbol{r}_{\boldsymbol{k}}^{(1)}$, ya sea de reflexión o de refracción, que con considerados como rayos emitidos desde el punto $\boldsymbol{p}^{(1)}$, Figura 2.10. Entonces, para cada rayo secundario $\boldsymbol{r}_{\boldsymbol{k},\ell}^{(1)}$, para $1 \leq \ell \leq M^{(1)}$, se vuelve a buscar la intersección con los objetos de la escena. Por lo tanto, el ℓ-ésimo rayo secundario $\boldsymbol{r}_{\boldsymbol{k},\ell}^{(1)}$ al intersecar con un objeto en el punto $\boldsymbol{p}_{\ell}^{(2)}$ puede generar hasta $M_{\ell}^{(2)}$ rayos secundarios $\boldsymbol{r}_{\boldsymbol{k},\ell}^{(2)}$. Este proceso se repite hasta que el rayo sale de la escena o se alcanza un número máximo K de rayos re-emitidos desde el rayo original $\boldsymbol{r}_{\boldsymbol{k}}^{(0)}$; es decir, $\boldsymbol{r}_{\boldsymbol{k},\ell}^{(k)}$, para $0 \leq k \leq K$, donde K es definido por alguno método. Alternativamente, se puede utilizar algún modelo físico de reducción de energía para los fotones y dejar que el número de rayos re-emitidos continue hasta que la energía de los fotones sea despreciable. Vale la pena resaltar que el conjunto de puntos de intersección $\left\{\boldsymbol{p}_{\ell}^{(k)}\right\}$ y el conjunto de rayos $\left\{\boldsymbol{r}_{\boldsymbol{k},\ell}^{(k)}\right\}$ forman un árbol cuya raíz es el píxel $\boldsymbol{k}$ y el resto de los puntos de intersección componen el resto de los vértices, en donde los puntos con un mismo valor k se encuentran en el mismo nivel del árbol. Claramente, los puntos de intersección, los vértices, se encuentran conectados por el conjunto de rayos, las aristas.

En este proceso, el valor final que se asigna al píxel $\boldsymbol{k}$ a partir del cálculo del modelo

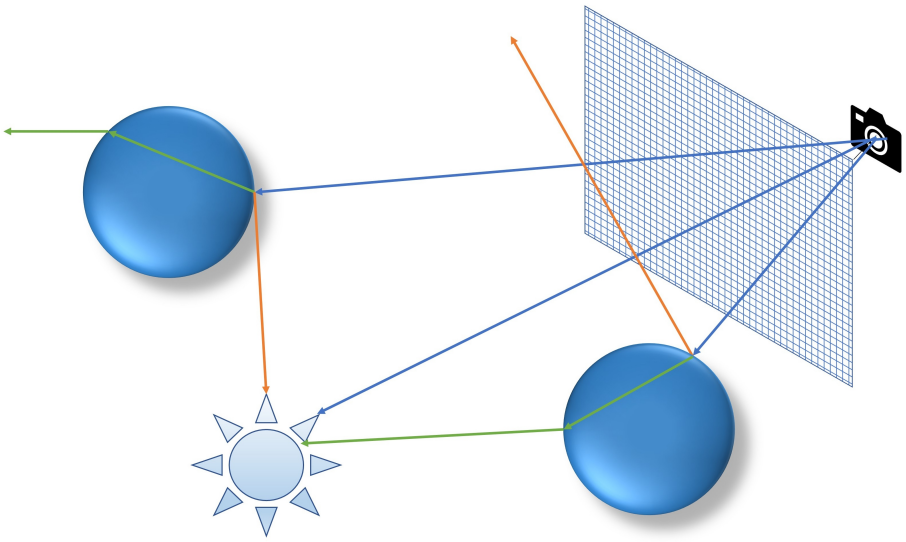


Figura 2.10: Ejemplo de rayos primarios y secundarios. En azul los rayos primarios, en verde rayos secundarios producidos por el efecto de refracción y en rojo los rayos secundarios producidos por el efecto de reflección.

de intersección luz - materia en cada una de las intersecciones $\left\{\boldsymbol{p}_{\ell}^{(k)}\right\}$, ya sea recorriendo el árbol de las hojas a la raíz o viceversa; siendo el primer esquema el más común. Si el rayo $\boldsymbol{r}^{(0)}$ no interseca a ningún objeto de la escena, entonces se asigna un color de fondo pre-establecido al píxel $\boldsymbol{k}$.

2.6 Arquitectura en paralelo (CUDA®)

La arquitectura de dispositivos de cómputo unificado (Compute Unified Device Architecture o CUDA®) es una plataforma de computo en paralelo y un modelo de programación, desarrollada por Nvidia® para aprovechar sus unidades de procesamiento gráfico (GPUs) con gran cantidad de núcleos; una plataforma que desde hace varios años ha sido usada ampliamente para aplicaciones de investigación e industriales. Algunas de las ventajas que CUDA® ofrece son [10–12]:

- Programación heterogénea serial y paralela. La programación en CUDA® involucra ejecutar código en dos diferentes plataformas de manera simultánea. Por un lado se requiere de un huésped (*host*), el cual es un sistema que contiene uno o más CPUs. Por otra parte, se pueden tener uno o más dispositivos GPU de Nvidia®. Bajo este esquema, el huésped y los dispositivos GPU se comunican mediante la transferencia de información en memoria. Esta operación puede ser

realizada en cualquier momento durante la ejecución del programa en CUDA®, permitiendo que las implementaciones en este paradigma se alternen entre seriales y paralelas.

- Soluciones escalables. El número de núcleos e hilos (*threads*) pueden ser adaptados a cada problema específico, haciendo posible escalar, dentro de los límites del *hardware*, el número de elementos bloques e hilos (*blocks* y *threads*) para obtener la mejor solución al problema. Por ello, cuando una aplicación define el número de elementos para un tipo de *hardware* específico, la misma implementación puede ser usada en otros modelos de GPU y CUDA® gracias a que la Interfaz de Programación de Aplicaciones, o *Application Programming Interface*, (API) se encargará de escalar el problema dentro de los límites que impondrá el *hardware*.

- Lenguaje de programación de CUDA®. El lenguaje usado para la programación es muy similar a C/C++, existiendo también una versión que es similar a FORTRAN.

- Eficiencia de costo. La relación entre el precio y la capacidad computacional decrece con cada generación nueva de *hardware* lanzado por Nvidia®, y en cada generación es más accesible el uso de soluciones implementadas en GPU.

Como mencionamos antes, las soluciones implementadas en CUDA® funcionan usando una comunicación entre un CPU huésped y dispositivos GPU, los programas se ejecutan inicialmente en el huésped y desde él se realizan las llamadas a los diferentes componentes en un dispositivo GPU. En este momento hacemos una pausa para anunciar una decisión para el resto del documento, para evitar confusión de ahora en adelante nos referiremos al CPU huésped como *host* y a los dispositivos GPU como *devices* en el esquema de CUDA®.

Las instrucciones ejecutadas en un *device* se escriben en una extensión del lenguaje C/C++ y son realizadas a través de la implementación de funciones especiales y *kernels*. Un *kernel* es un segmento de código que será resuelto en paralelo, cada ejecución paralela del *kernel* será resuelta por una unidad de procesamiento (núcleo o *core*) del *device* correspondiente. La cantidad de veces que un *kernel* será resuelto por los núcleos del *device* será definida por la implementación y configuración del código. Cada ejecución del *kernel* será administrada por un hilo (*thread*). El número de núcleos disponibles dependerá del modelo y versión del *device*: modelos más robustos tendrán más núcleos disponibles. Si bien existe la posibilidad que un *device* no contenga los

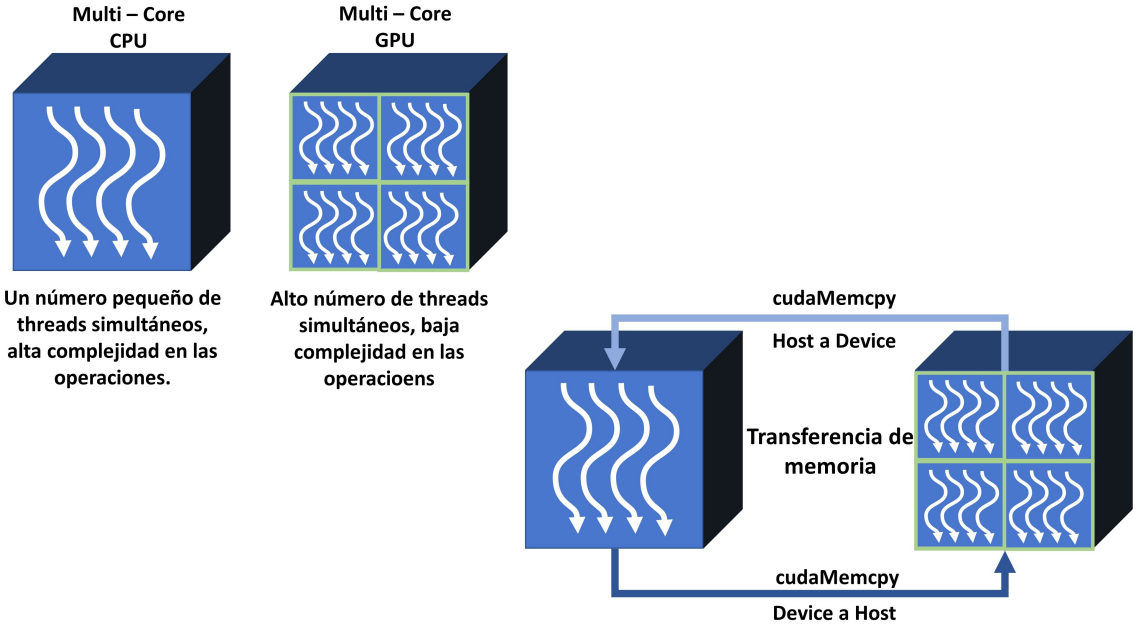
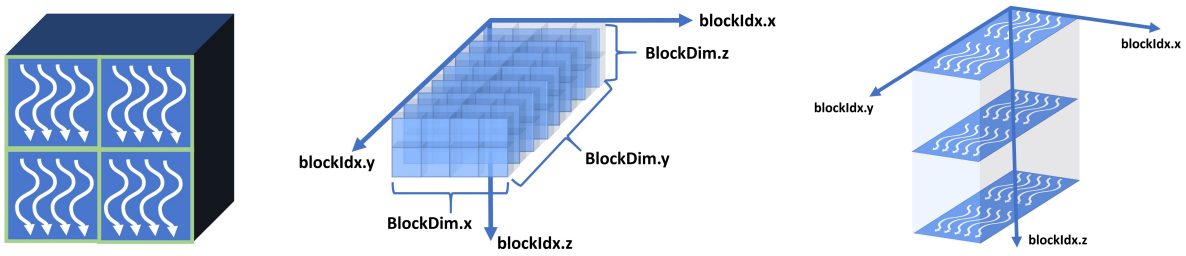


Figura 2.11: Esquema de la progresión heterogénea de una implementación en CUDA®. Esta arquitectura permite mezclar soluciones entre componentes seriales y paralelos. El costo computacional para un sistema aumenta con el número y tamaño de la memoria transferida, esto es conocido como latencia.

núcleos disponibles para la cantidad de ejecuciones solicitadas para un *kernel*, estas ejecuciones deberán ser segmentadas en bloques (*blocks*) para ser atendidos en grupos; los grupos de bloques son procesados realizando tantas iteraciones como sean necesarias para atender los bloques solicitados. La Figura 2.11 muestra un esquema que representa la progresión del código entre *host* y *device*, también se muestra la transferencia de memoria necesaria, esto es realizadado a través de una instrucción de trasferencia *cudaMemcpy*, instrucción proporionada por CUDA®.

Cada *kernel* define el código que será resuelto por todos los hilos en paralelo, cada hilo resuelve el mismo código con diferente información. Es posible utilizar condicionales o condiciones de control (tales como *switch*, *do*, *for*, *while*), pero puede afectar el rendimiento de ejecución de instrucciones ya que su uso crea ramificaciones.

CUDA® organiza la solución en paralelo usando una representación en retícula (*grid*) del *device*, cada retícula representa la forma en que se utilizará el dispositivo; la retícula es, a su vez, dividida en bloques y cada bloque contiene hilos de CUDA®. Como se mencionó previamente la cantidad de bloques ejecutados al mismo tiempo dependerá de los núcleos disponibles en el *device*, la cantidad de bloques que se usarán



(a) (b) (c)

Figura 2.12: Estructura lógica de CUDA® en la que se muestra como se hace uso de los elementos (a) retícula (*grid*), (b) bloques (*blocks*) e (c) hilos (*threads*) para organizar el procesamiento en paralelo.

por reticula dependerá de la implemnentación y la cantidad de datos a procesar. Cada bloque puede contener hasta 1,024 hilos, por lo qué para poder ejecutar más hilos, el *kernel* puede ser ejecutado con muchos bloques del mismo tamaño a la vez (la cantidad de bloques depende de la capacidad computacional de la GPU y los detalles pueden ser encontrados en la Tabla 13 de la guía de programación de Nvidia® [11]). Para poder hacer uso de la organización de hilos en CUDA®, las API ofrece tres vectores tridimensionales variables: *threadIdx*, *blockIdx* y *blockDim*.

Un GPU multiprocesador Nvidia® está diseñado para ejecutar cientos de hilos de manera simultánea usando la arquitectura llamada *Single-Instruction, Multiple-Thread* (SIMT). La Figura 2.12 muestra la relación que existe entre retícula, bloque e hilo, cómo se organizan y la estructura 3D de los hilos y bloques.

La selección de un bloque (p_x, p_y, p_z) dentro de una retícula 3D, con dimensiones $R_u \times R_v \times R_w$, en CUDA® se obtiene usando por medio de la siguiente ecuación $b = p_x + (p_y \times R_v) + (p_z \times R_v \times R_w)$. Adicionalmente, CUDA® ofrece las estructuras *threadIdx*, *blockIdx* y *blockDim* para poder obtener las coordenadas de un hilo dentro de un bloque determinado; para ello se realizan los siguientes calculos

$$\begin{aligned} t_x &= threadIdx.x + (blockIdx.x \times blockDim.x), \\ t_y &= threadIdx.y + (blockIdx.y \times blockDim.y), \\ t_z &= threadIdx.z + (blockIdx.z \times blockDim.z), \end{aligned} \tag{2.8}$$

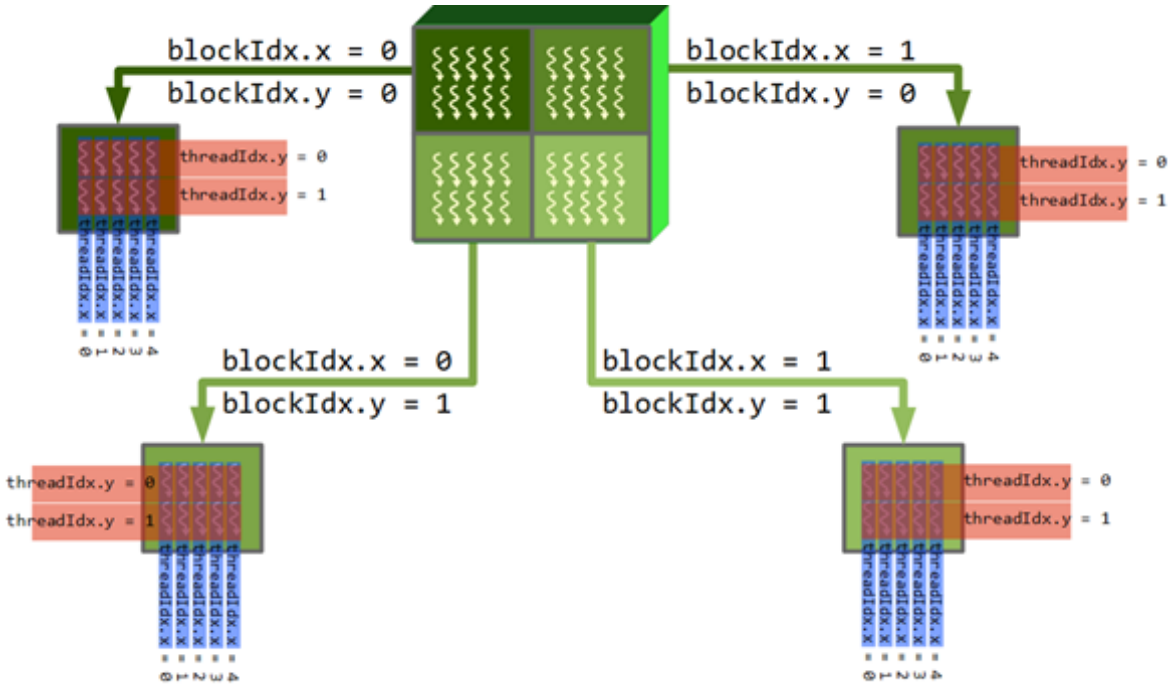


Figura 2.13: Esquema de operación de los índices de bloques e hilos en CUDA®. Cada uno de los bloques contiene un grupo de hilos que serán ejecutados en paralelo, el índice de cada hilo dentro de un bloque inicia en 0 y el índice completamente individual de cada hilo es una combinación de los valores contenidos en *threadIdx*, *blockIdx* y *blockDim* (ver (2.8)).

donde (t_x, t_y, t_z) son las coordenadas del hilo; haciendo notar que dentro de cada bloque el índice para cada hilo inicia en 0. La Figura 2.13 muestra la relación entre los bloques e hilos.

Las funciones dentro de un *kernel* son ejecutadas una a la vez dentro de la retícula. Las operaciones dentro del *device* pueden tener diferente tiempos de cómputo para diferentes hilos. Para forzar al algoritmo a esperar por la finalización de todos los hilos dentro de un *kernel* antes de continuar con la ejecución se pueden utilizar diversas barreras de sincronización. Los bloques, por otra parte, no pueden ser sincronizados, esto se debe a que los bloques son resueltos en cualquier orden, y, más importante, los bloques pueden ser resueltos de manera serial o paralela. CUDA® maneja los bloques de manera que las funciones del *kernel* puedan ser escalables a través de cualquier número de núcleos en paralelo (dependiendo de la aplicación y de las capacidades de la GPU). La escalabilidad también permite el uso de diferentes tipos de GPUs con diferentes especificaciones sin necesidad de modificar el código dentro de los *kernels*.

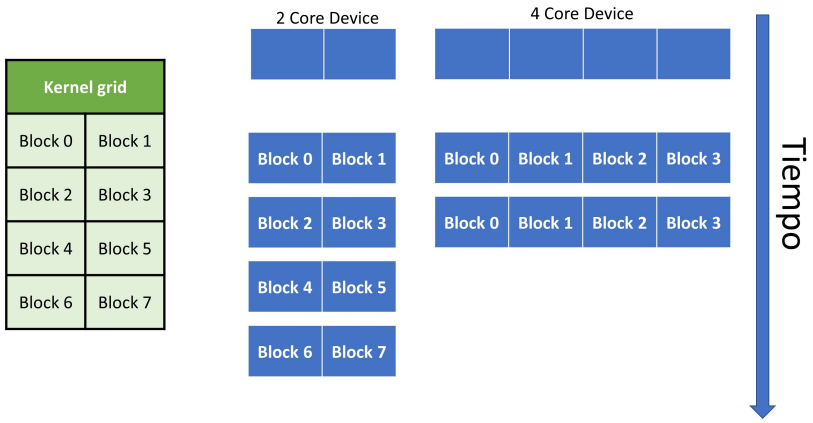


Figura 2.14: Las implementaciones realizadas en arquitecturas CUDA® serán escalables a las capacidades de la GPU en la que es ejecutada. Entre más núcleos dentro del sistema, menos tiempo computacional será requerido por el sistema para dar solución al algoritmo.

La Figura 2.14 muestra un ejemplo de dos sistemas diferentes, uno con dos núcleos y otro que contiene cuatro. Es importante aclarar que la escalabilidad de cualquier implementación en CUDA® no depende únicamente del *hardware*, sino, de como se planea e implementa la solución a un problema. Las implementaciones en paralelo son considerablemente diferentes a las seriales en su comportamiento por lo que no todos los algoritmos pueden ser paralelizables para obtener resultados más rápido; la dependencia de datos, sincronización de procesos y distribución de trabajo juegan un papel importante en cualquier implementación en paralelo.

Capítulo 3

Metodología

En este capítulo se describe la implementación eficiente que usamos para los *octrees* y otras estructuras que necesitamos a lo largo de la tesis.

3.1 Implementación eficiente de *octrees*

En este proyecto se pone énfasis en el cómputo en paralelo, para lo cual utilizamos la infraestructura de CUDA® por ser la más extendida; sin embargo, los métodos que proponemos pueden ser implementados en otras plataformas con características similares tales como ROCm® de AMD. Por ello, en el resto del texto sólo nos referiremos a la infraestructura de CUDA® y a *hardware* de Nvidia®.

Una característica de los *kernels* de CUDA® es que su rendimiento es más eficientes cuando cada *thread* procesa información independiente, sin esperar a los datos procesados o producidos por otros *threads*.

Por otro lado, uno de los principales retos para la utilización de un *octree* es encontrar una forma para realizar una navegación eficiente de sus nodos; por ejemplo, una búsqueda de nodo siempre comienza por el nodo raíz y se desciende, de forma jerárquica, hasta alcanzar el nodo deseado. Este proceso, implica la realización de varias comparaciones a los diferentes niveles del *octree*. Además, el almacenamiento de un *octree* puede ser complejo debido a que se debe mantener su jerarquía en todo momento, lo cual implica relaciones complejas entre sus nodos y un alto consumo de memoria. Para resolver estas dificultades es necesario buscar la independencia de los datos tanto como sea posible.

En particular, para obtener un alto rendimiento en aplicaciones que requieren de la

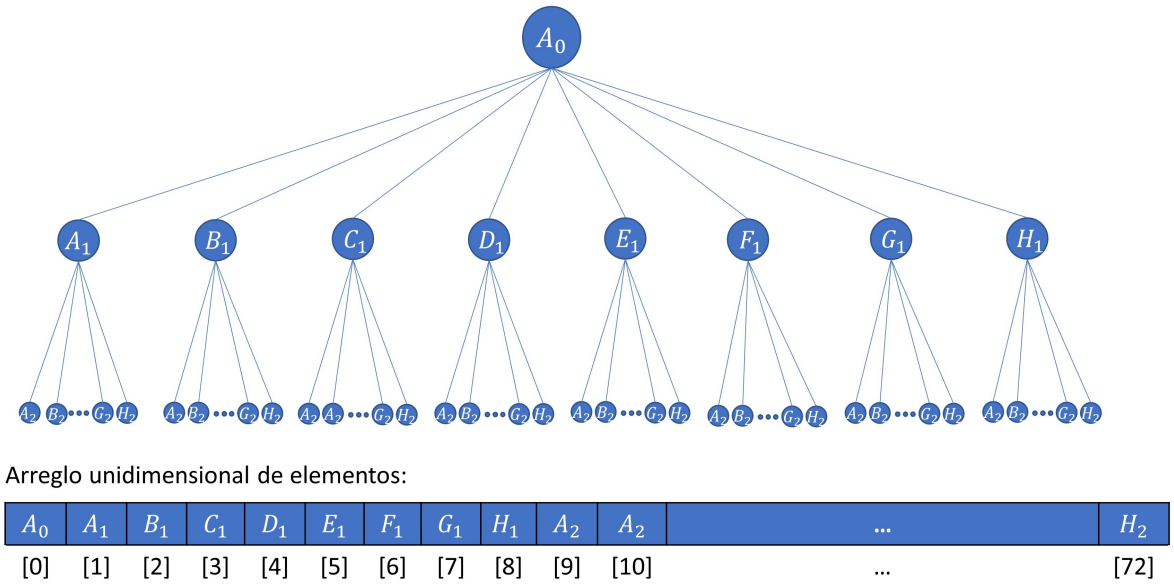


Figura 3.1: Esquema de un *octree* con dos niveles de profundidad representado, almacenado, en un arreglo unidimensional.

utilización de *octrees* es deseable tener la mayor independencia posible entre nodos de tal forma que puedan ser procesados de manera individual y paralela por *kernels* de CUDA® y para que cada *thread* pueda procesar únicamente un nodo de manera directa. Una forma de procesar independientemente cada nodo es reestructurar el *octree* de tal forma que pueda ser representado como arreglo unidimensional $\mathcal{L}$ en el cual cada uno de sus elementos corresponde a un nodo del *octree*, ver Figura 3.1.

El número máximo de nodos en el k-ésimo nivel de un *octree* es 8^k y el máximo número de nodos en un *octree* de L niveles es

$$\mathcal{T} = \sum_{k=0}^{L} 8^k, \tag{3.1}$$

por lo que el tamaño de $\mathcal{L}$ será igual a $\mathcal{T}$. Un nodo o en la j-ésima posición dentro del k-ésimo nivel (es decir, es uno de los 8^k nodos posibles en el nivel k) de un *octree* $\mathcal{O}$ con L niveles será asignado al elemento con posición i, calculada como sigue

$$i = \begin{cases} 0, & \text{si } k = 0, \\ \sum_{j=0}^{k-1} 8^j + m - 1, \text{para } 1 \leq m \leq 8^k, & \text{si } k > 0. \end{cases} \tag{3.2}$$

En otras palabras, el nodo padre ocupará la posición 0 de $\mathcal{L}$, los ocho nodos correspon-

dientes al nivel 1 ocuparan las posiciones 1 al 9 de $\mathcal{L}$, los 64 nodos del nivel 2 ocuparán las posiciones 9 a 72 y así sucesivamente hasta llevar a la profundidad máxima del *octree*. Esta forma forma de representar un *octree* permite, en combinación de otras técnicas de manejo de datos de las cuales hablaremos más adelante, permiten aprovechar CUDA® de manera eficiente.

3.2 Claves Morton para partículas

En esta tesis nos referiremos a puntos $\boldsymbol{u} \in \mathbb{R}^3$ sin volumen pero que poseen diversas propiedades tales como color, valores de intensidad, factores de refracción o de difracción. La utilización de claves Morton para reducir la dimensionalidad de datos ha probado mejorar las implementaciones de métodos de cómputo en paralelo, acelerando su procesamiento [13]. En el caso de una partícula $\boldsymbol{u}$ representamos su posición alternadamente por sus coordenadas (u_1, u_2, u_3) o por medio de una clave Morton que combina las representaciones binarias de cada una de las coordenadas de la partícula. Cada coordenada u_j se puede representar por medio de una secuencia de números binarios $a_j^{(m-1)}, \ldots, a_j^{(0)}$, en donde $a_j^{(k)} \in \{0, 1\}$ for $1 \leq k \leq m$; en este orden, el bit $a_j^{(m-1)}$ es el más significativo y el bit $a_j^{(m-1)}$ es el menos. Por lo que la clave Morton $\mathcal{M}_{\boldsymbol{u}}$ de la partícula $\boldsymbol{u}$ se puede obtener combinando y mezclando todos los digitos binarios de sus coordenadas de la siguiente manera: $a_3^{(m-1)}\ a_2^{(m-1)}\ a_1^{(m-1)}\ \ldots\ a_3^{(l)}\ a_2^{(l)}\ a_1^{(l)}\ \ldots\ a_3^{(0)}\ a_2^{(0)}\ a_1^{(0)}$, ver la Figura 3.2.

En nuestra implementación de claves Morton utilizamos 10 bits para representar cada coordenada (es decir, m es igual a 10), lo que permite la representación de un valor entero máximo igual a 1024. Con esta codificación, cada clave Morton usa 30 bits, que son almacenados como enteros de 32 bits, dejando dos bits sin usar. En lenguajes inspirados en el lenguaje de programación C [14], o asociados a él, se utilizan variables del tipo `int` sin signo para dicha representación. En nuestra implementación, los 2 bits restantes son utilizados como identificador espacial: estos bits sirven de referencia para saber si una partícula está dentro del espacio representado por un *octree*. Si una partícula, por razones de simulación, sale del espacio representado por un *octree*, entonces debe ser excluida del procesamiento (por ejemplo, de la visualización). Por lo tanto, se tienen dos casos: *i*) si una partícula se encuentra dentro del espacio representado por un *octree* los dos bits más significativos de la representación de su clave Morton serán 01, por otro lado *ii*) si la partícula está fuera de dicho espacio, los bits

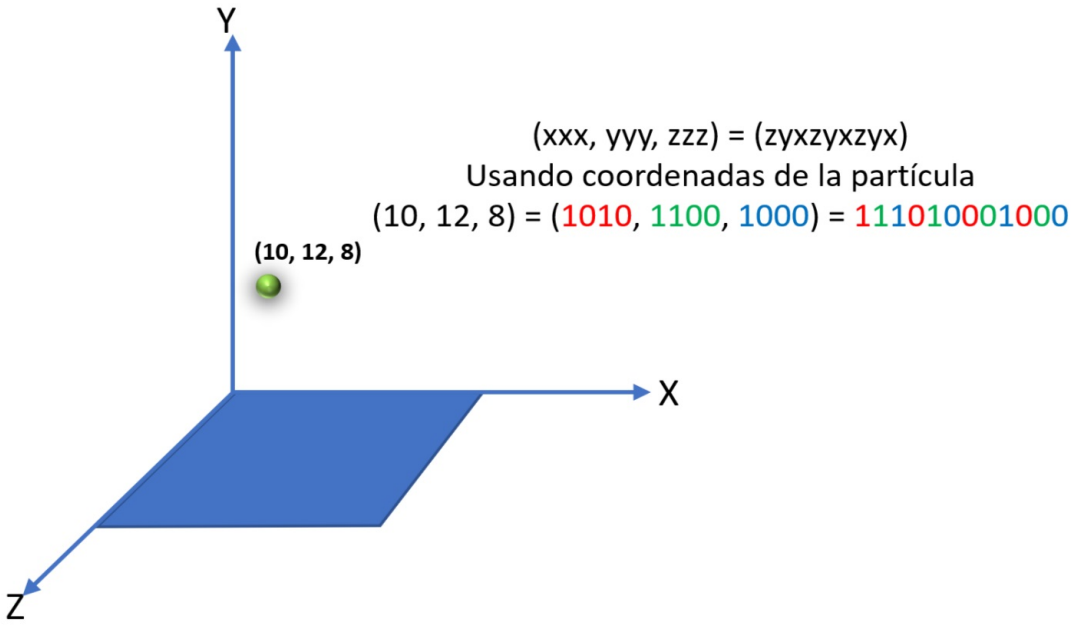


Figura 3.2: Construcción de una clave Morton para un partícula.

más significativos de la representación de la clave Morton de la partícula serán 00 y por lo tanto no será considerada para el procesamiento.

Claramente, la representación de coordenadas está limitada por la restricción de que cada coordenada debe ser entera positiva y la utilización de m-bits (10 bits en nuestras implementaciones), por lo que en nuestra implementación se lleva a cabo una normalización por el valor 2^m. de la siguiente manera. Para todas las partículas $\boldsymbol{u} \in \mathcal{C}$, en donde $\mathcal{C} \subset \mathbb{R}^3$ (la región del espacio representada por el *octree*) se busca el valor máximo de cada dimensión $v_j = \max_{u_j}(u_j | \boldsymbol{u} \in \mathcal{C})$, para $1 \leq j \leq 3$, y el valor de la j-ésima coordenada de todas las partículas $\boldsymbol{u} \in \mathcal{C}$ sin normalizadas de la siguiente manera $u'_j = 1024 \times \frac{u_j}{v_j}$, ver Figura 3.3. Las nuevas coordenadas las llamaremos *coordenadas del* octree.

Una vez que se obtienen las coordenadas normalizadas de todas las partículas $\boldsymbol{u} \in \mathcal{C}$ (las coordenadas del *octree*) se procede a obtener sus correspondientes claves Morton. La Figura 3.4 muestra la clave Morton $\mathcal{M}_{\boldsymbol{u}}$ obtenida para una partícula $\boldsymbol{u} = (200, 520, 50)$.

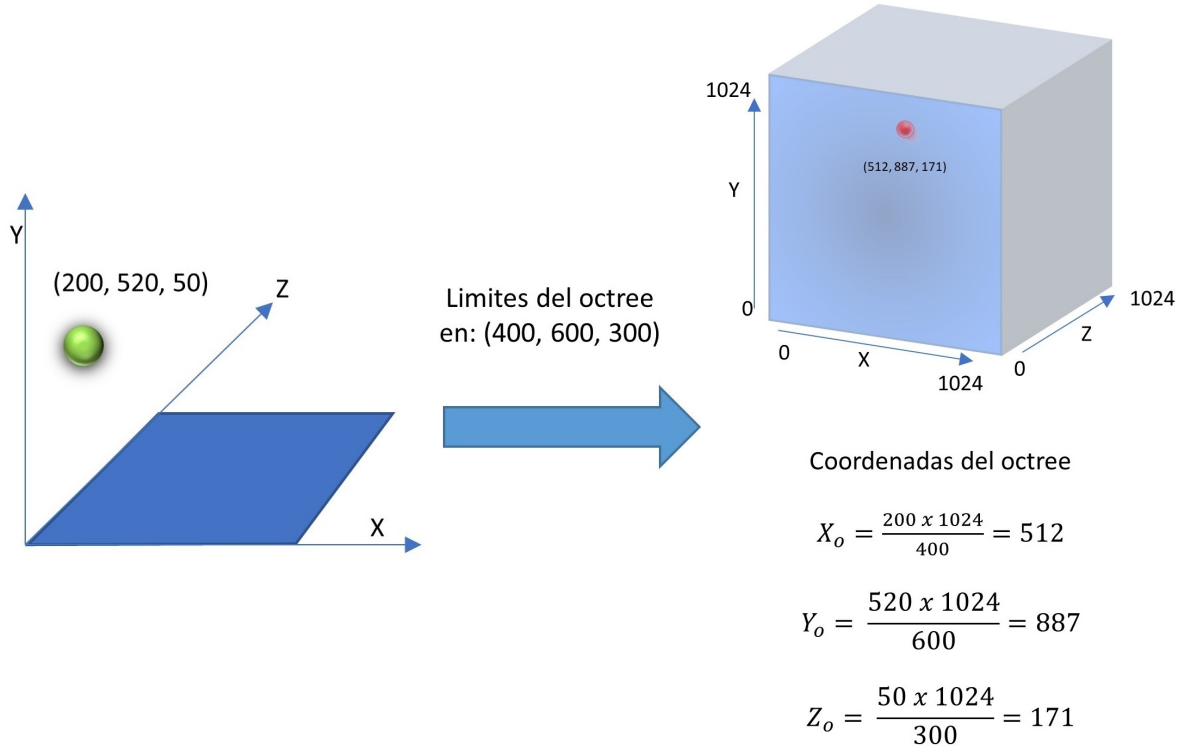


Figura 3.3: Conversión de coordenadas espaciales a coordenadas del *octree* para una partícula (a) $\mathsf{u} = (200, 520, 50)$ dentro de una región cúbica del espacio con dimensiones máximas (b) $[400, 600, 300]$ que producen la partícula en coordenadas del *octree* (c) $\mathsf{u}' = (512, 887, 171)$.

3.3 Claves Morton para subdivisiones espaciales

De manera similar al caso de partículas, también deseamos asignar claves Morton a las subdivisiones espaciales representadas por un *octree*; recordando que estas estructuras de datos representan subdivisiones del espacio que pueden ser equivalentes a vóxeles (elementos de una rejilla G_Δ). Por ello, por ahora nos referiremos a los nodos $o \in \mathcal{O}$, considerando que existe una relación directa entre un nodo o y una subdivisión del espacio (por ejemplo, un vóxel $\boldsymbol{v}_i \in V$ con el correspondiente nodo hoja $o_{f,i}$). Debido a que cada nodo $o \in \mathcal{O}$ representa una subdivisión espacial, la cual puede concebirse como el vecindario de Voronoi $\mathcal{V}$ del punto $\boldsymbol{c}_o \in \mathbb{R}^3$, es posible utilizar la misma aproximación que utilizamos para las partículas y asignar una clave Morton $\mathcal{m}_o$ a cada uno de los nodos $o \in \mathcal{O}$ de la misma manera que se hizo con una partícula $\boldsymbol{u} \in \mathcal{C}$. Es importante resaltar que la clave Morton asignada a un nodo o sirve como identificador único, conteniendo la información espacial que representa el nodo. Además, la clave Morton

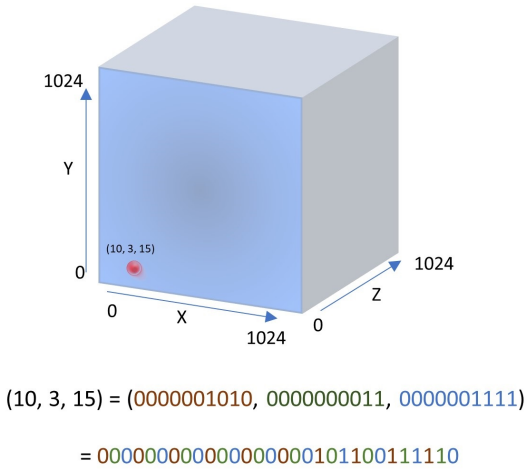


Figura 3.4: Ejemplo de la generación de claves Morton para un punto $\boldsymbol{c}$ (punto rojo) que tiene coordenadas en$(10, 3, 15)$. Esta representación es igual a $(1010, 0011, 1111)$ cuando se utiliza 4 bits por coordenada. Las claves Morton son obtenidas realizando una combinación y reordenación de todos los dígitos binarios para producir la secuencia: $(\, a_3^{(3)}\ a_2^{(3)}\ a_1^{(3)}\ a_3^{(2)}\ a_2^{(2)}\ a_1^{(2)}\ a_3^{(1)}\ a_2^{(1)}\ a_1^{(1)}\ a_3^{(0)}\ a_2^{(0)}\ a_1^{(0)}\,)$. Por lo tanto, la clave Morton para el punto $\boldsymbol{c}$ es (101100111110) o 0×B3E en notación hexadecimal.

asignada a un nodo también permite obtener un orden jerárquico entre los nodos del *octree*. Finalmente, la clave Morton $\mathcal{m}_o$ del nodo o relaciona directamente las posiciones espaciales $\boldsymbol{x} \in \mathbb{R}^3$ que se encuentran dentro del vecindario de Voronoi representada por el nodo.

Al igual que ocurre con las claves Mortón asignadas a partículas, las claves Mortón asignadas a todos los nodos $o \in \mathcal{O}$ serán representadas en números enteros de 32 bits, de los cuales los 2 bits más significativos indicaran que los nodos pertenecen a $\mathcal{O}$. Estos bits se usarán como máscara para un punto $\boldsymbol{x} \in \mathbb{R}^3$, indicando que el punto está dentro del espacio representado por $\mathcal{O}$ y por lo tanto debe tomarse en cuenta en el procesamiento (por ejemplo, en su visualización). Los nodos o y puntos $\boldsymbol{x}$ que se encuentran dentro del espacio $\mathcal{C}$ representado por $\mathcal{O}$.

Para generar la clave Morton $\mathcal{m}$ para un nodo $o \in \mathcal{O}$ se utilizan las coordenadas del punto $\boldsymbol{o}$ en la esquina inferior del vecindario de Voronoi $\mathcal{V}_o$ asignado al octante representado por el nodo o en lugar de la posición central $\boldsymbol{c}_o$, como se ve en la Figura 3.5; las coordenadas del punto $\boldsymbol{o}$ siempre se manejarán en las coordenadas del *octree*, ver

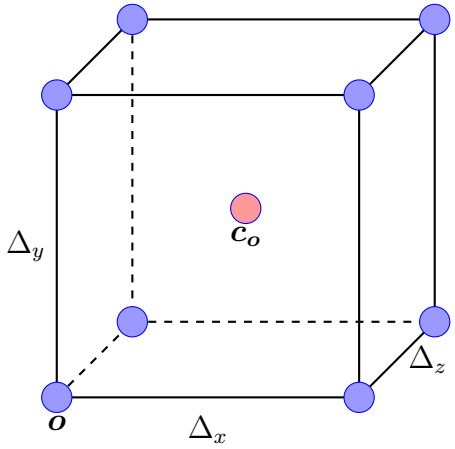


Figura 3.5: Esquema del vecindario de Voronoi asociado a un nodo $o \in \mathcal{O}$ con sus vértices (puntos azules) y el centro del vecindario $\boldsymbol{c}_o$ (punto rojo).

la Sección 3.2. Como ejemplo, consideremos el nodo raíz o_r, cuyos bits más significativos son iguales a 01, de un espacio de $512 \times 512 \times 512$. Como o_r contiene a todo el espacio abarcado por el *octree* su esquina más cercana será el punto $(0, 0, 0)$, el origen. Por lo que, los bits de la clave Morton $\mathcal{M}_{o_r}$ serán iguales a 01000000000...00. Esta clave no solo nos servirá como identificador de o_r, sino también como filtro para determinar los puntos que estén dentro de la región representada por el *octree* $\mathcal{O}$. Debido a que en nuestro trabajo todos los puntos en el espacio $\mathbb{R}^3$ que son representados en el *octree* $\mathcal{O}$, sus coordenadas son manejadas en las *coordenadas del octree*; lo que implica qué, en nuestra implementación, sus coordenadas son representadas como potencias de dos y su rango es $[0, 1024]$.

Las claves Morton tradicionales siguen un patrón en Z, como el mostrado en la Figura 2.1, lo cual implica un patrón complejo de expansión. Por ello, relacionamos las claves Morton de todo $o \in \mathcal{O}$ con la estructura jerárquica de $\mathcal{O}$ de la siguiente manera: en el nivel 1 de $\mathcal{O}$ existen 8 nodos, y para su representación se utilizan 3 bits, para el nivel 2 se tienen 64 nodos y para su representación son necesarios 6 bits y así sucesivamente hasta la altura de $\mathcal{O}$ ($o_j^{(k)}$, para el k-ésimo nivel y $1 \leq j \leq 8^l$). Bajo este esquema, para el primer nivel sólo es necesario consultar 5 bits de una clave Morton asignada a un nodo $o^{(1)}$: $01 \ldots 000$, $01 \ldots 001$, $01 \ldots 010$, $01 \ldots 011$, $01 \ldots 100$, $01 \ldots 101$, $01 \ldots 110$, $01 \ldots 111$ (recordando que representamos una clave Morton en 32 bits, con los 2 bits más significativos como indicadores espaciales). Para verificar nodos a un nivel mayor sólo es necesario considerar los 3 bits siguientes; por ejemplo, para el

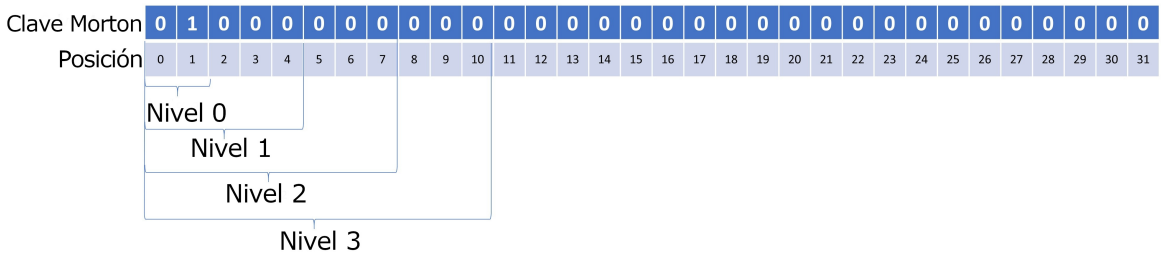


Figura 3.6: Distribución de los niveles del *octree* en la clave Morton.

segundo nivel ser revisan el segundo set de 3 bits, lo cual nos da los 8 hijos siguientes para cada hijo del ese nivel lo que da 64 en total, ver Figura 3.6. La clave Morton final debe tener una longitud de 30 bits, por lo que los bits no utilizados serán rellenados con valores iguales a cero.

Por lo tanto, para una posición, o partícula, $\boldsymbol{u}$ su clave Morton $\mathcal{M}_{\boldsymbol{u}}$ se puede crear utilizando la jerarquía de la estructura de $\mathcal{O}$, en donde la posición k del conjunto de cada 3 bits en $\mathcal{M}_{\boldsymbol{u}}$ describen los nodos del nivel k de profundidad en $\mathcal{O}$. En nuestras implementaciones, se tiene un máximo de 10 niveles de profundidad, lo cual es suficientemente profundo para representar más de mil millones de nodos y nos da uniformidad con las claves Morton utilizadas para nubes de puntos o partículas. La Figura 3.7 muestra la jerarquía establecida por esta distribución para todos los nodos en el espacio 3D. De esta manera es posible generar las claves Morton para partículas aprovechando la jerarquía de un *octree* sin necesidad de hacer demasiados cálculos.

En caso de que se deseen utilizar los centros de los octantes asociados a los nodos o para generar una clave Morton $\mathcal{M}_{\boldsymbol{u}}$ para una partícula $\boldsymbol{u}$, basta con ajustar los bits en clave: para utilizar los centros de los nodos o^k se generan la clave Morton como hemos descrito arriba y luego se toman los $(3k + 2)$ bits más significativos.

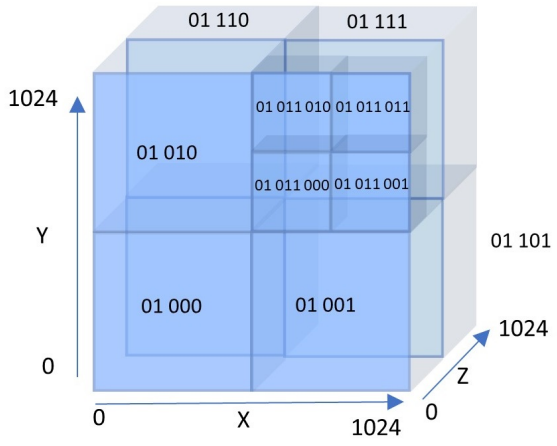


Figura 3.7: Distribución de las claves Morton dentro de los octantes generados por la subdivisión de un *octree*.

Ejemplo: Tomamos como ejemplo el espacio del octante definido por un nodo o, con centro en (384, 128, 128) y esquina más cercana al origen en (256, 0, 0) a profundidad 2 del *octree*. Usando el centro obtenemos una clave Morton igual a:
(384, 128, 128) = (0110000000, 0010000000, 0010000000) = 000 001 111 000 000 000 000 000 000 000.
Agregamos los bits descriptores del *octree* (01) y obtenemos
01 000 001 111 000 000 000 000 000 000 000.
Dado que estamos procesando un nodo que se encuentra a profundidad 2 del *octree* solo utilizamos 2×3 bits más los bits descriptores del *octree*, dando un total de 7 bits. La clave Morton final nos queda como:
01 000 001 000 000 000 000 000 000 000 000.
Si calculamos la clave Morton usando la esquina más cercana al origen del mismo nodo obtenemos la clave Morton:
(256, 0, 0) = (0100000000, 0000000000, 0000000000) = 01 000 001 000 000 000 000 000 000 000 000.
Esta clave es exactamente igual a la calculada usando el centro del nodo con el ajuste correspondiente.

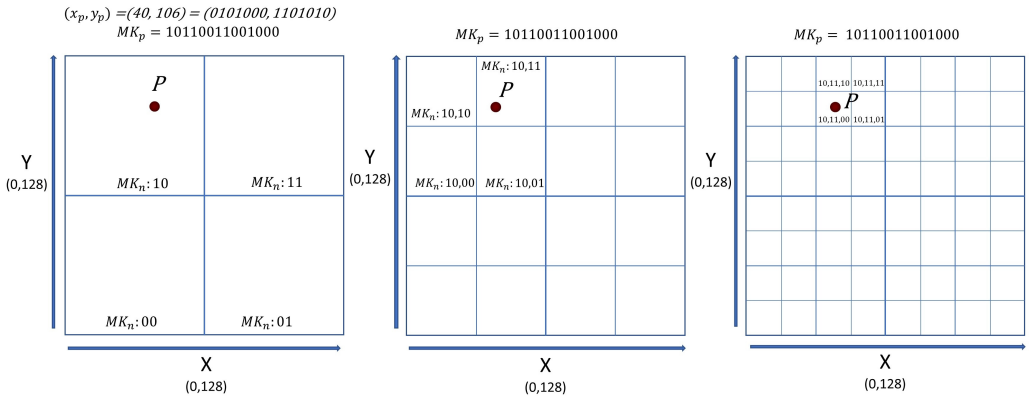


Figura 3.8: Ejemplo en 2D de la correspondencia entre las clave Morton asignada a una partícula y el nodo en el que se encuentra contenida a diferentes niveles de profundidad.

3.4 Interacción entre claves Morton para nodos y partículas

Una de las características más importantes que tienen los *octrees* para la representación del espacio es la capacidad de subdividirlo en octantes con el propósito de representar regiones de ese espacio cada vez más pequeñas, lo que permite, entre otras cosas, reducir el espacio de búsqueda; por ejemplo, cuando se implementa el algoritmo de *raytracing* esta característica es indispensable para un funcionamiento óptimo.

Para la búsqueda de un conjunto $\mathcal{C}$ de puntos, o partículas, dentro de una región del espacio representada por un *octree* $\mathcal{O}$ es necesario encontrar todos los nodos $o \in \mathcal{O}$ cuyas regiones contienen puntos o información; en otras palabras, para un nodo o se busca el conjunto $\mathcal{H} = \{\boldsymbol{u} \in \mathcal{C} \mid \boldsymbol{u} \cap \mathcal{V}_o \neq \emptyset\}$, en donde $\mathcal{V}_o$ es el vecindario de Voronoi asociado al nodo o. Una forma inocente, pero una de las más comunes, de buscar dichas intersecciones es seleccionar uno nodo o y recorrer todos los puntos en $\mathcal{C}$ para determinar si están dentro de $\mathcal{V}_o$, esto determina si la región representada por o tiene asociada puntos. Este proceso es extremadamente lento, especialmente cuando $|\mathcal{C}|$ es grande. Otra aproximación consiste en almacenar en cada nodo o la información de todos los puntos del conjunto $\mathcal{H} = \{\boldsymbol{u} \in \mathcal{C} \mid \boldsymbol{u} \cap \mathcal{V}_o \neq \emptyset\}$ al crear el octree $\mathcal{O}$; sin embargo, el uso de memoria se vuelve prohibitivo, además de que conlleva procesamiento extra al tener que reasignar puntos cuando se aplica una transformación a los puntos que puede

moverlos de posición en el espacio y, por lo tanto, en el *octree* (por ejemplo, rotaciones o traslaciones). Una forma más eficiente para buscar puntos dentro de regiones asociadas a los nodos o asociar los puntos a los diferentes nodos del *octree* es tener un sistema que permita filtrar su información. Las claves Morton en combinación con *octrees* justo permite realizar este tipo de filtrado y es una de las contribuciones de este trabajo.

Al utilizar claves Morton para los puntos $\boldsymbol{u} \in \mathbb{R}^3$ y para los nodos $o \in \mathcal{O}$ se tiene una relación espacial entre ambos tipos de datos y es posible crear una máscara que filtre la información de los datos. Para los puntos $\mathcal{H} = \{\boldsymbol{u} \in \mathcal{C} \mid \boldsymbol{u} \cap \mathcal{V}_o \neq \emptyset\}$ es posible utilizar la clave Morton $\mathcal{M}_{\boldsymbol{u}}$ del punto $\boldsymbol{u} \in \mathcal{H}$ como máscara para encontrar al resto de los puntos en $\mathcal{H}$. Esta relación claves Morton parar puntos en el espacio y claves Morton para los nodos del *octree* es ejemplificada en la Figura 3.8, ahí se puede visualizar cómo el número de bits necesarios para representar un enmascaramiento se encuentra en relación a la profundidad en el *octree* a la que se encuentra un nodo. Todos los puntos $\boldsymbol{u}$ que se encuentran en una región $\mathcal{V}_o$ (el conjunto $\mathcal{H}$) tienen la misma máscara en su claves Morton $\mathcal{M}_{\boldsymbol{u}}$. Por ello, es posible realizar un ordenamiento de los puntos $\boldsymbol{u} \in \mathcal{H}$ utilizando su respectiva clave $\mathcal{M}_{\boldsymbol{u}}$, lo cual nos permitirá un búsqueda rápida de los puntos utilizando la máscara; se puede acceder directamente a una posición dentro de $\mathcal{C}$ al filtrar por medio de la máscara hasta hallar el punto deseado.

3.5 Intercambio clave Morton e índice de arreglo unidimensional

Para lograr la independencia de procesamiento de los nodos $o \in \mathcal{O}$, y por ello lograr procesamiento masivo en paralelo, los almacenamos en un arreglo unidimensional $\mathcal{L}$ ordenandolos con base en sus claves Morton. De esta forma es posible acceder a y modificar las propiedades de cualquier nodo de forma independiente, sin importar el nivel en que se encuentre el nodo dentro de $\mathcal{O}$. Para poder hacer uso del arreglo $\mathcal{L}$ es necesario poder transformar la clave Morton $\mathcal{M}_o$ al índice apropiado dentro de $\mathcal{L}$. Para lograr esto, se utilizan los valores de las tripletas que conforman la clave Morton (sin incluir los 2 bits más significativos que sirven como identificadores). La cantidad de valores o tripletas que usaremos dependerá de la profundidad a la que se encuentre o dentro de $\mathcal{O}$, p. ej., si $o^{(4)}$, entonces usaremos las primera 4 tripletas de $\mathcal{M}_o$ de izquierda a derecha (12 bits): 01 111 111 111 111 000 000 000 000 000 000. Estos bits nos permiten determinar el índice de la posición del arreglo $\mathcal{L}$ en la cual se encuentra

el nodo o que se busca. Como se mencionó en la Sección 3.1, el nodo raíz o_r ocupa la posición 0 del arreglo unidimensional, los 8 nodos descendientes (nodos en el nivel 1) con posiciones 1-8 en $\mathscr{L}$, los 64 nodos en el nivel 2 ocupan las posiciones 9-72 y así sucesivamente hasta la altura de $\mathcal{O}$. Claramente, cada que se agrega un nivel k de $\mathcal{O}$ al arreglo $\mathscr{L}$, se crean 8^k elementos al arreglo. Por ello, la posición de un nodo o dentro del arreglo $\mathscr{L}$ se obtiene por medio de

$$i_{\mathscr{L}}(o) = \left(\sum_{j=0}^{\lfloor \frac{(\beta(\mathcal{M}_o))}{3} \rfloor} 8^j \right) + \nu(\mathcal{M}_o), \tag{3.3}$$

donde $\nu(\mathcal{M}_o)$ representa el valor decimal de las tripletas usadas sin considerar los bits descriptores y $\beta(\mathcal{M}_o)$ representa la cantidad de bits que serán usados de la clave Morton (3 multiplicados por la cantidad de tripletas usadas). Usando (3.3) es posible obtener la posición del elemento en $\mathscr{L}$ para cualquier nodo $o \in \mathcal{O}$ simplemente por medio de su clave Morton.

Ejemplo: Para un nodo o en el segundo nivel de $\mathcal{O}$ con $\mathcal{M}_o$ igual a

01 001 011 000 000 000 000 000 000 000 000

Se toman únicamente los primeros 6 valores (2 tripletas) de izquierda a derecha sin considerar los 2 bits descriptores.

$\mathcal{M}_o$ = 01 001 011 000 000 000 000 000 000 000 000

$\nu(\mathcal{M}_o)$ es igual a 11, valor en decimal de 001011b, y $\beta(\mathcal{M}_o)$ es igual a 6. Usando (3.3), obtenemos:

$$i_{\mathscr{L}} = (\sum_{j=0}^{\lfloor \frac{6}{3} \rfloor} 8^j) + 11 = 8 + 1 + 11 = 20.$$

También es posible transformar el índice de la posición dentro del arreglo $\mathscr{L}$ a una clave Morton. Para realizar este proceso es importante recordar el proceso que convierte la clave Morton de un nodo a un índice de posición dentro del arreglo $\mathscr{L}$, entre mayor sea el valor del índice mayor será la profundidad a la que se encuentra en $\mathcal{O}$, ver la Figura 3.6, y cada k-ésimo nivel de $\mathcal{O}$ contiene 8^k nodos. Dado un índice $i_{\mathscr{L}}$, el valor decimal de la clave Morton se obtiene por medio de

$$D_{\mathcal{M}_o}(i_{\mathscr{L}}) = i_{\mathscr{L}} - \sum_{j=0}^{\log_8(i_{\mathscr{L}})} 8^j. \tag{3.4}$$

El valor obtenido por la ecuación anterior no considera los dos bits descriptores del nodo pariente, por lo que es necesario obtener el número de tripletas que se necesitan para representar el valor decimal de la clave Morton obtenida con (3.4). Dado un índice $i_{\mathcal{L}}$, el número de tripletas se obtiene con

$$T_{m_o}(i_{\mathcal{L}}) = \lfloor \log_8(i_{\mathcal{L}}) \rfloor + 1. \tag{3.5}$$

Si el valor obtenido con (3.4) no ocupa todos los bits usados para representar una clave Morton, los bits no utilizados serán llenados con ceros. Una vez generada la clave Morton, se asignan los 2 bits descriptores.

Ejemplo: Para un índice $i_{\mathcal{L}}$ igual a 180 se tiene
$D_{m_o}(180) = 180 - \sum_{j=0}^{\log_8(180)} 8^j = 106$
y
$T_{m_o}(180) = \lfloor \log_8(180) \rfloor + 1 = 3.$

Por lo que se sabe que el nodo o se encuentra en el nivel 3 de $\mathcal{O}$ con la posición 126 dentro de ese nivel del árbol. Para construir la clave m_o se usan 9 bits (3 tripletas), por lo que el valor decimal 106 resulta en $(001101010)_b$, resultando en la clave Morton final:

$m_o = 01\ 001\ 101\ 010\ 000\ 000\ 000\ 000\ 000\ 000\ 000.$

3.6 Implementación de *metaballs*

Como vimos en el capítulo anterior, es posible generar isosuperficies suaves a través de (2.2) y para ello existen diversas funciones base b propuestas en la literatura; algunos ejemplos son las siguientes

$$\text{Blinn [7]:} \qquad b(r) = \mathrm{e}^{-ar^2}, \tag{3.6a}$$

$$\text{Nishimura [15]:} \qquad b\,(r) = \begin{cases} 1 - 3(\frac{r}{a})^2 & , 0 \leq r \leq \frac{a}{3}, \\ \frac{3}{2}[1 - (\frac{2}{a})]^2 & , \frac{a}{3} \leq r \leq a, \end{cases} \tag{3.6b}$$

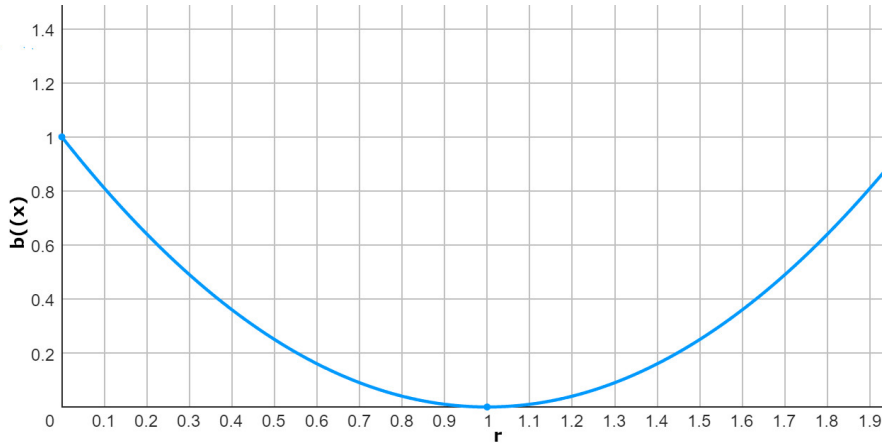
$$\text{Murakami [16]:} \qquad b\,(r) = \begin{cases} [1 - (\frac{r}{a})]^2 & , 0 \leq r \leq a, \\ 0 & , r > a, \end{cases} \tag{3.6c}$$

$$\text{Wyvill [17]:} \qquad b\,(r) = \begin{cases} -\frac{4}{9}(\frac{r}{a})^6 + \frac{17}{9}(\frac{r}{a})^2 + 1 & , 0 \leq r \leq a, \\ 0 & , r \geq a, \end{cases} \tag{3.6d}$$

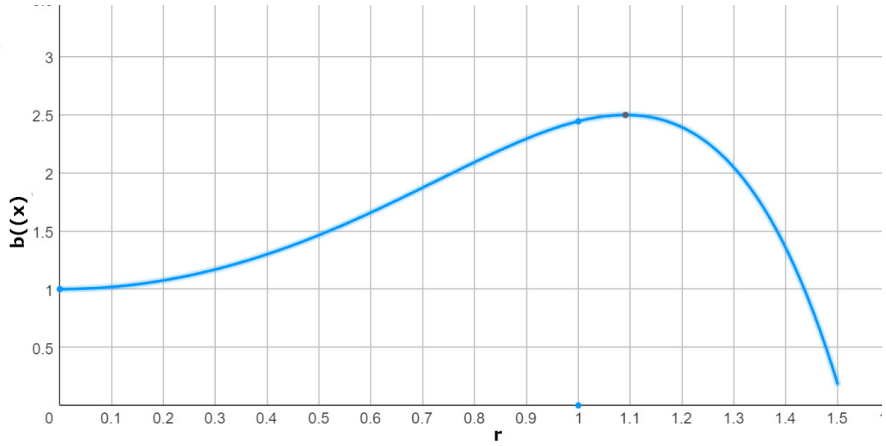
donde a representa el radio del soporte de la función y r la distancia al centro de la *metaball*. La primera función base propuesta para esta aplicación fue realizada por Blinn, la cual proporciona un modelado suave de manera natural pero debido a que es una ecuación exponencial su procesamiento representa un alto consumo computacional. Otro problema que presenta esta ecuación se debe a que no importa que tan lejos se evalúe el valor resultante será mayor a 0, esto implica que todas las *metaballs* deben ser consideradas sin importan que tan lejos se encuentren del punto de evaluación. Para las ecuaciones (3.6b), (3.6c) y (3.6d) el campo de intensidad donde $r \geq a$ es igual a 0 por lo que no es necesario evaluar la ecuación. Esto corrige uno de los inconvenientes presentes en la ecuación Blinn y permite un mejor control sobre la interacción entre las *metaballs*. Todas las ecuaciones propuestas en trabajos recientes evitan el uso de exponenciales con la finalidad de reducir el costo computacional y son implementadas en diversos escenarios y con propósitos distintos, p. ej., simulación de líquidos, humo o incluso solidos.

La selección de la función base se basa, principalmente, en el comportamiento que tiene la función dentro de su soporte. La Figura 3.9 muestra el comportamiento para las funciones de Murakami y Wyvill con $a = 1$, la gráfica de la izquierda, Murakami, muestra una ecuación decreciente monótonamente que encuentra su punto más bajo cuando $r = a$ mientras que la gráfica de la derecha muestra la función de Wyvill que es creciente con su punto más alto cuando $r > a$. Debido a que la forma de la función seleccionada afecta el comportamiento de la función resultante, f en (2.2) y, por lo tanto, de la visualización es importante seleccionar la función base adecuada para la tarea a realizar; p. ej., si se desea la simulación de humo la función propuesta por

Murakami es más apropiada ya que la intensidad es más grande conforme la distancia a la *metaball* se hace más pequeña, esto puede utilizarse para modelar la densidad del humo a visualizar.



(a)



(b)

Figura 3.9: Funciones base propuestas por (a) Murakami y (b) Wyvill evaluadas con $a = 1$.

Para el trabajo realizado en este proyecto se realizaron pruebas con diferentes funciones y se decidió utilizar la ecuación propuesta por Murakami ya que presenta el comportamiento más apropiado para nuestra aplicación, es decreciente monótonamente y encuentra su punto más bajo cuando $r = a$ por lo que podemos utilizar el valor a como punto limite para las *metaballs* consideradas. Modificando la notación para aplicación de este proyecto, la ecuación final queda de la siguiente forma:

$$b(r) = \begin{cases} [1 - (\frac{r}{a})]^2 - \tau & , 0 \leq r \leq a, \\ 0 & , r > a, \end{cases} \tag{3.7}$$

donde τ es el umbral para encontrar la isosuperficie deseada, ver (2.2). El valor del soporte a sirve de límite del área de interacción entre las *metaballs*, esto está directamente relacionado con la resolución bajo la cual una isosuperficie será formada. Si el valor de a es más pequeño que la distancia a la que se encuentran los puntos que representan los centros de las *metaballs*, no se generará una unión entre las mismas y la isosuperficie no será formada o tendrá huecos. Para poder seleccionar un valor apropiado para a, se usará el *octree* $\mathcal{O}$ y sus nodos, específicamente sus nodos hoja o_f, la resolución del *octree* y por lo tanto la dimensiones de los nodos hoja determinarán el valor de a con el propósito de formar una isosuperficie completa a la resolución en la que el *octree* esté segmentado el espacio. Los nodos o_f de $\mathcal{O}$ tienen forma cuboide y sus aristas están definidas por un tamaño $\Delta_x, \Delta_y, \Delta_z$ en cada dirección, la distancia a de influencia será definida por:

$$a = \frac{min(\Delta_x, \Delta_y, \Delta_z)}{2}.$$

Esto nos permite la interacción apropiada de *metaballs* en cada nodo. La calidad de la visualización será dictada por la profundidad a la que el *octree* sea definido ya que entre más profundidad menor será el radio de influencia y podremos observar *metaballs* con interacciones menores.

3.7 Implementación de *raytracing*

Como se mencionó en la Sección 2.5 la técnica de *raytracing* nos permite la generación de imágenes a partir de modelar la interacción de la luz con los materiales de los objetos en la escena a partir de el lanzamiento y rastreo de rayos de luz dentro de la escena. Sin embargo, este proceso es costoso computacionalmente ya que es necesario calcular tanto las intersecciones entre haces de luz y objetos como los rebotes para cada uno de los rayos lanzados por la cámara; entre más compleja es la escena, mayor será el costo computacional. Por lo tanto, para acelerar el proceso de *raytracing* se realizó una implementación que aprovecha todas las técnicas presentadas anteriormente para obtener visualizaciones más fluidas y con el mejor detalle gráfico posible.

Una de las características del modelo de *raytracing* es que el procesamiento de cada

rayo lanzado desde el plano es independiente, lo cuál permite procesarlos en una unidad de procesamiento computacional (ya sea GPU o CPU). En este trabajo, aprovechamos los varios *kernels* disponibles en CUDA® para alcanzar un buen rendimiento para la generación de imágenes.

En nuestra implementación usamos un modelo de proyección en perspectiva y al inicio del algoritmo seleccionamos un punto origen $\boldsymbol{c}$ desde el cual se inician todos los rayos. En este esquema, todos rayos $\overline{\boldsymbol{r}}_k$, para $1 \leq k \leq K$, pasarán por el k-ésimo píxel $\boldsymbol{k}_k$ del plano de imagen Π_I para luego seguir su camino hacía la escena S. Por lo que cada rayo puede ser descrito de la siguiente manera

$$\overline{\boldsymbol{r}}_k^{(j)} = \boldsymbol{c}_k^{(j)} + d_k^{(j)}\ \overrightarrow{\boldsymbol{o}}_k^{(j)}, \tag{3.8}$$

donde j es el j-ésimo nivel del rayo; en este esquema, el rayo en el nivel 0 es aquel que sale del plano Π_I, el rayo en el primer nivel es aquel que ha sido generado después de la intersección entre el rayo del nivel 0 y algún objeto y así sucesivamente. El vector unitario $\overrightarrow{\boldsymbol{o}}_k^{(j)}$ determina la dirección del rayo y $d_k^{(j)}$ es un real positivo cuyo valor permite representar cualquier punto sobre el rayo; en las implementaciones de *raytracing*, el valor de $d_k^{(j)}$ representa la distancia a la cual el rayo del nivel j interseca con algún objeto o fuente de iluminación desde su origen $\boldsymbol{c}_k^{(j)}$. Para los rayos iniciales, aquellos que parten desde el plano Π_I y $j = 0$, el vector $\overrightarrow{\boldsymbol{o}}_k$ se obtiene normalizando la diferencia $\boldsymbol{k}_k - \boldsymbol{c}_k^{(0)}$; para rayos secundarios (aquellos con nivel $j > 0$ y generados cuando un rayo es reflejado o trasmitido desde una superficie) el vector de dirección es generado a partir de las Leyes de Snell.

Uno de las tareas que consume más recursos computacionales es la búsqueda de objetos que pueden intersecar con un rayo $\overline{\boldsymbol{r}}_k^{(j)}$. Una de las formas más comunes para mejorar el rendimiento de la búsqueda de objetos es la utilización de una estructura de datos que divida el espacio de la escena S, el cual es un subconjunto de $\mathbb{R}^3$. Una de las estructuras de datos más utilizadas en la literatura es el *octree*. Claramente, bajo este esquema los rayos se moverán a través de los diferentes vecindarios de Voronoi asociados a los nodos del *octree*. Por lo tanto, es útil describir la interacción entre los rayos y los nodos del *octree*.

Intersección rayos – nodos del *octree*

Como hemos visto antes, el vecindario de Voronoi asociado a un nodo o es un cuboide de $\Delta_x \times \Delta_y \times \Delta_z$, como se puede ver en la Figura 3.5. Las dimensiones de cada cuboide se

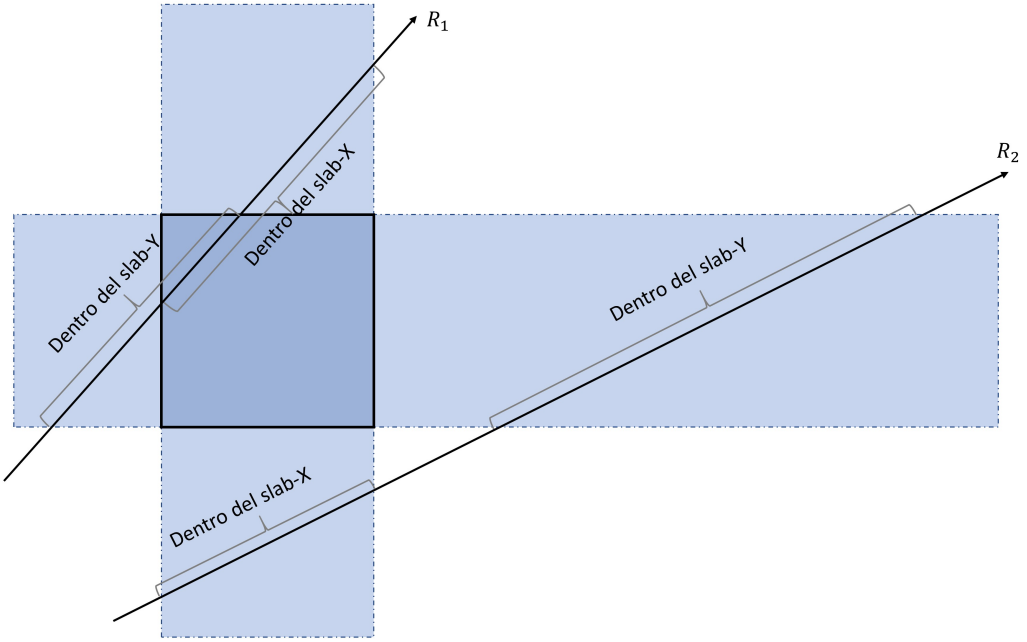


Figura 3.10: Intersección de rayos con losas en los planos $\vec{\boldsymbol{y}}\vec{\boldsymbol{z}}$ y $\vec{\boldsymbol{x}}\vec{\boldsymbol{z}}$.

pueden normalizar por el nivel j al que se encuentra el nodo asociado o dentro de $\mathcal{O}$ de la siguiente manera $\frac{\Delta_x}{j+1}$, $\frac{\Delta_y}{j+1}$, $\frac{\Delta_z}{j+1}$. Es importante resaltar que en nuestra aplicación estas dimensiones son manejadas en coordenadas del *octree* y por ello todos los vecindarios de Voronoi $\mathcal{V}_o$ asociados a $o \in \mathcal{O}$ se encuentran alineados con los ejes coordenados $\vec{\boldsymbol{x}}$, $\vec{\boldsymbol{y}}$ y $\vec{\boldsymbol{z}}$. Para encontrar la intersección entre un cuboide y un rayo (en otras palabras, una línea) utilizamos la técnica propuesta por Kay y Kajiya en [18], la cual es una técnica basada en losas (del inglés *slabs*) y que considera una losa como el espacio entre dos planos paralelos; por lo tanto, es posible utilizar los planos que forman las caras de los vecindarios de Voronoi $\mathcal{V}_o$, que con paralelos a los ejes coordenados, como los planos de las losas, la Figura 3.10 muestra un ejemplo análogo en 2D.

Por lo tanto, para determinar la intersección de un rayo $\overline{\boldsymbol{r}}_k$ con el vecindario de Voronoi $\mathcal{V}_o$ es necesario determinar la intersección de la línea representando el rayo con cada uno de las losas que contienen a las caras de $\mathcal{V}_o$, este proceso se realiza en pares buscando la intersección con losas paralelas. Las caras de $\mathcal{V}_o$ que son intersecadas por el rayo $\overline{\boldsymbol{r}}_k$ se encuentran utilizando los ejes del sistema de coordenadas $\vec{\boldsymbol{x}}$, $\vec{\boldsymbol{y}}$ y $\vec{\boldsymbol{z}}$, correspondientes a planos paralelos a $\vec{\boldsymbol{y}}\vec{\boldsymbol{z}}$, $\vec{\boldsymbol{x}}\vec{\boldsymbol{z}}$ y $\vec{\boldsymbol{x}}\vec{\boldsymbol{y}}$, respectivamente. La distancia $d_k^{(j)}$

del rayo $\overline{\boldsymbol{r}}_k$ a la intersección a un plano paralelo a $\vec{\boldsymbol{y}}\vec{\boldsymbol{z}}$ se obtiene por medio de

$$d_{k,x}^{(j)} = \frac{(d_{\mathcal{V}_{o,x}} - c_{k,x})}{\langle \vec{\boldsymbol{o}}_k, \vec{\boldsymbol{x}} \rangle}, \tag{3.9}$$

en $c_{k,x}$ es el primer elemento de las coordenadas del punto de origen del rayo $\overline{\boldsymbol{r}}_k^{(j)}$ y $d_{\mathcal{V}_{o,x}}$ es el desplazamiento de la losa perpendicular al eje $\vec{\boldsymbol{x}}$. El producto interno en el denominador permite saber si hay intersección con los planos asociados: cualquier valor mayor a cero implica que el rayo interseca un plano perpendicular a uno de los ejes $\vec{\boldsymbol{x}}$, $\vec{\boldsymbol{y}}$ o $\vec{\boldsymbol{z}}$. Para obtener las distancias a un plano paralelo a $\vec{\boldsymbol{x}}\vec{\boldsymbol{z}}$ y $\vec{\boldsymbol{x}}\vec{\boldsymbol{y}}$ se utiliza la coordenada apropiada del punto de origen $\boldsymbol{c}_k^{(j)}$ y el producto interno con el vector correspondiente $\vec{\boldsymbol{y}}$ y $\vec{\boldsymbol{z}}$.

Pseudo Código 1 Intersecciones con $\mathcal{V}_o$

```
1: t_min_x = min(d^(j)_{k,x1}, d^(j)_{k,x2})
2: t_max_x = max(d^(j)_{k,x1}, d^(j)_{k,x2})
3: t_min_y = min(d^(j)_{k,y1}, d^(j)_{k,y2})
4: t_max_y = max(d^(j)_{k,y1}, d^(j)_{k,y2})
5: t_min_z = min(d^(j)_{k,z1}, d^(j)_{k,z2})
6: t_max_z = max(d^(j)_{k,z1}, d^(j)_{k,z2})
7: t_min = t_min_x
8: t_max = t_max_x
9: if t_min ≥ t_max_y ∨ t_min_y ≥ t_max then
10:     intersección fuera del octante
11: else
12:     t_min = max(t_min, t_min_y)
13:     t_max = min(t_max, t_max_y)
14:     if t_min ≥ t_max_z ∨ t_min_z ≥ t_max then
15:         intersección fuera del octante
16:     else
17:         t_min = max(t_min, t_min_z)
18:         t_max = min(t_max, t_max_z)
19:     end if
20: end if
21: return t_min y t_max
```

Debido a que un vóxel $\mathcal{V}_o$ tiene dos caras paralelas que son perpendiculares a cada eje $\vec{\boldsymbol{x}}$, $\vec{\boldsymbol{y}}$ y $\vec{\boldsymbol{z}}$, se procesan dos losas por cada eje y por ello, se encuentran dos distancias $d_{k,x_1}^{(j)}$ y $d_{k,x_2}^{(j)}$ para las losas perpendiculares al eje $\vec{\boldsymbol{x}}$; lo mismo ocurre para los ejes $\vec{\boldsymbol{y}}$ y $\vec{\boldsymbol{z}}$,

ver Figura 3.10. Para encontrar el punto de entrada y salida del vóxel $\mathcal{V}_o$ es necesario realizar comparaciones entre los 3 pares de puntos de intersección de las losas para determinar los puntos de intersección delimitados por las paredes correspondientes al cuboide. Dado que los pares de losas son tratadas como planos perpendiculares a cada eje coordenado, la intersección entre cada par puede suceder fuera de los límites del cuboide asociado al octante o. Por ello, es necesario utilizar el procedimiento en el Pseudo Código 1, el cual proporciona los valores t_{min} y t_{max} para los 3 pares de losas asociados al vóxel $\mathcal{V}_o$, valores que corresponden a la distancia a la que el $\overline{\boldsymbol{r}}_k$ interseca las caras del vóxel $\mathcal{V}_o$ asociado al nodo $o \in \mathcal{O}$. Es importante destacar que en la práctica la implementación del procedimiento Pseudo Código 1 mantiene las referencias a las distancias $d^{(j)}_{k,x_1}$, $d^{(j)}_{k,x_2}$, $d^{(j)}_{k,y_1}$, $d^{(j)}_{k,y_2}$, $d^{(j)}_{k,z_1}$ o $d^{(j)}_{k,z_2}$ para regresar los valores t_{min} y t_{max}.

3.7.1 Navegación del *octree* usando claves Morton

La navegación tradicional de un *octree* consiste en una búsqueda en profundidad a través de la estructura de datos generada: Se visita un nodo para luego revisar sus ocho hijos, de lo cuales se seleccionan aquellos con información y se visitan sus ocho nodos hijo correspondientes (la siguiente generación de hijos); este proceso se repite hasta encontrar el primer nodo hoja o_f con información deseable. Este proceso implica que la estructuras de datos sea algo sofisticada que permita el movimiento entre nodos de manera descendente y ascendente, tal como listas ligadas.

Por el contrario, la implementación que se ha hecho de *octrees* en este proyecto busca aprovechar las relaciones creadas por las claves Morton asociadas a con los nodos de la estructura de datos. Como se menciono en la Sección 3.3 el *octree* sigue un patron de asignación de claves Morton jerárquico donde los bits asignados, considerando desde el más significativo al menos significativo, permiten describir los diferentes niveles del *octree* por medio de las claves asignadas. Por lo tanto, es posible subir o bajar desde un nodo, realizando una búsqueda de posición, usando únicamente las claves Morton de los nodos, la Figura 3.11 muestra la jerarquía de claves Morton en el *octree*.

Siguiendo esta idea el primer paso para todo rayo $\overline{\boldsymbol{r}}_k$ es verificar la intersección con el nodo raíz o_r del *octree*, los rayos que no intersecan con $\mathcal{V}_{o_r}$ pueden ser descartados inmediatamente ya que no tendrán interacción con la información asociada al *octree*. Por el contrario, si el rayo $\overline{\boldsymbol{r}}_k$ interseca con $\mathcal{V}_{o_r}$ se procede a buscar la intersección con alguno de los ocho vóxeles $\mathcal{V}_{o^{(1)}}$ asociados a los nodos hijo en el nivel 1 de $\mathcal{O}$. Recordando que en nuestra implementación un *octree* es representado como un arreglo

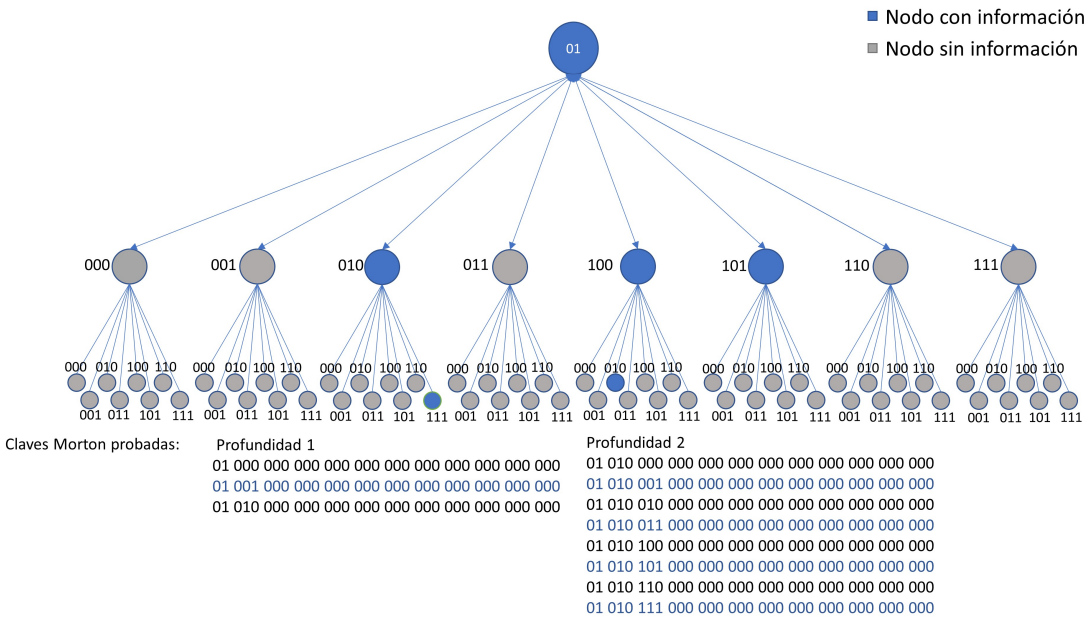


Figura 3.11: Jerarquía de nodos usando las claves Morton asignadas. Para encontrar un nodo hoja basta con consultar las claves Morton de los diferentes nodos en la estructura de datos.

unidimensional $\mathcal{L}$, ver Sección 3.1, entonces para poder acceder a un nodo hijo dentro de $\mathcal{L}$ basta con conocer el índice en el que se encuentra el nodo; dicho índice se obtiene usando (3.3). Las claves Morton de los hijos de un nodo o se calculan usando la clave Morton de o, cada tripleta de bits representa un nivel del *octree*, por lo que es posible utilizar las permutaciones de los bits de la clave $\mathcal{M}_o$ para encontrar las claves Morton de los nodos hijo; para el nodo raíz o_r el proceso es el siguiente.

La clave Morton $\mathcal{M}_{o_r}$ del nodo raíz es "01 000 000 000 000 000 000 000 000 000 000". Las combinaciones posibles de la primera tripleta, después de descartar los bits descriptores, corresponden al primer nivel de $\mathcal{O}$. Existen ocho posibles combinaciones para tres bits que corresponden a los ocho hijos del nodo o_r, por lo tanto las claves Morton de los hijos serán

Nodo hijo 1:	01 000 000 000 000 000 000 000 000 000 000
Nodo hijo 2:	01 001 000 000 000 000 000 000 000 000 000
Nodo hijo 3:	01 010 000 000 000 000 000 000 000 000 000
Nodo hijo 4:	01 011 000 000 000 000 000 000 000 000 000
Nodo hijo 5:	01 100 000 000 000 000 000 000 000 000 000
Nodo hijo 6:	01 101 000 000 000 000 000 000 000 000 000
Nodo hijo 7:	01 110 000 000 000 000 000 000 000 000 000
Nodo hijo 8:	01 111 000 000 000 000 000 000 000 000 000

Tabla 3.1: Claves Morton para los ocho hijos de un nodo raíz o_r.

Una vez que se han obtenido las claves Morton de los nodos hijo es posible obtener sus índices dentro del arreglo $\mathcal{L}$. Este proceso aplica para cualquier nodo $o \in \mathcal{O}$ hasta llegar a los nodos hoja. No sólo es posible obtener la clave Morton de un nodo hijo a partir de la clave de su nodo padre, también es posible obtener la clave Morton de un nodo padre a partir de un nodo hijo haciendo el proceso inverso. Para este proceso se reducen la cantidad de tripletas que se usan, usando una tripleta más por cada nivel que se desee incrementar hasta ascender al nivel deseado del *octree*.

La navegación del rayo $\overline{\boldsymbol{r}}_k$ dentro del *octree* continua procesando los nodos hijo cuyos vóxeles que son intersecados por el rayo de manera similar al proceso descrito antes. Dado que se realiza una navegación en profundidad dentro del *octree* es necesario guardar los nodos que han sido visitados durante el proceso de búsqueda de intersección por el rayo $\overline{\boldsymbol{r}}_k$ con los nodos del *octree* para el caso de no encontrar un nodo hoja durante el proceso de búsqueda a lo largo de una ramificación visitada. Para este proceso se utiliza una pila de índices $\mathcal{S}$ en la cual se guardarán los índices de los nodos visitados, la Figura 3.12 muestra el proceso de almacenamiento en la pila de nodos.

Cuando un rayo $\overline{\boldsymbol{r}}_k$ interseca con un nodo o, se procede a revisar la intersección con los nodos hijo que tengan información; los nodos hijo intersecados por el rayo $\overline{\boldsymbol{r}}_k$ son agregados a la pila de nodos $\mathcal{S}$. Entonces, se extrae un nodo o de $\mathcal{S}$ y se procede a repetir el proceso de búsqueda de intersección entre el rayo $\overline{\boldsymbol{r}}_k$ y los hijos nodo de o; en caso de que ocurra una intersección con alguno de los nodos hijo, éstos son agregados a la pila $\mathcal{S}$. Si el rayo $\overline{\boldsymbol{r}}_k$ interseca con un nodo o a una profundidad s de la pila $\mathcal{S}$ pero el rayo no interseca con ninguno de sus nodos hijo con información, entonces el nodo o desechado y se procede a procesar el siguiente nodo o en la pila. Este proceso eventualmente encuentra nodos hoja con información sobre los cuales se buscaran las intersecciones con *metaballs*.

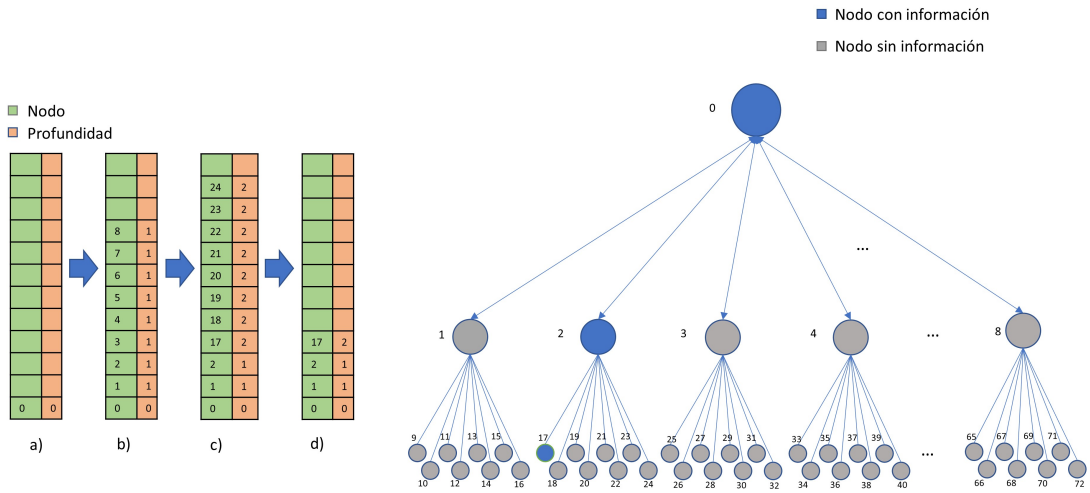


Figura 3.12: Pila de nodos utilizada para encontrar un nodo hoja con el cual se tenga intersección. Los nodos son agregados y removidos conforme se comprueba la intersección con un rayo.

Capítulo 4

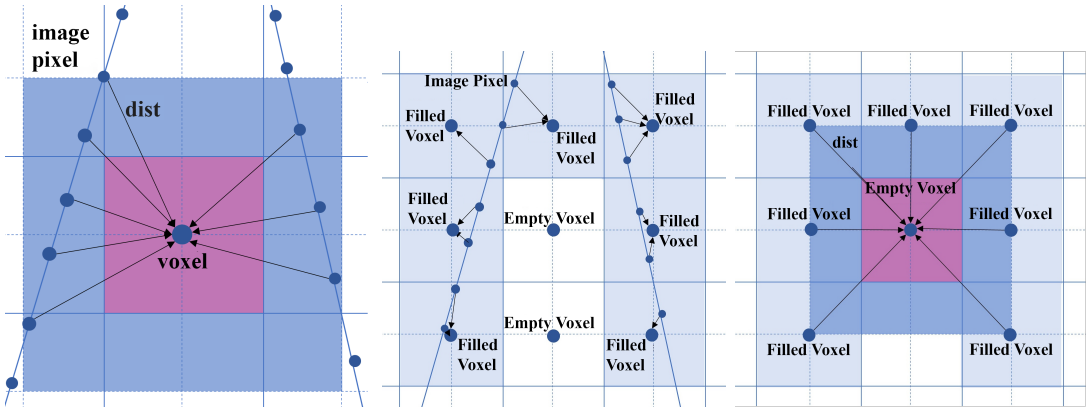
Reconstrucción 3D de ultrasonido

Las técnicas propuestas en esta tesis, si bien, fueron diseñadas con el propósito de ser utilizadas para mejorar las visualización de objetos a través del uso de *metaballs*, su uso no se limita a este tipo de aplicaciones. Los algoritmos desarrollados permiten la navegación de datos de manera eficiente por lo que pueden ser utilizados para cualquier aplicación que requiera el procesamiento eficiente de datos en el espacio 3D. Gracias a la versatilidad de los métodos desarrollados es posible utilizar nuestra implementación eficiente en *hardware* de *octrees*, con las relaciones creadas con claves Morton, para la reconstrucción de imágenes 3D de ultrasonido a partir de imágenes 2D obtenidas con la técnica *freehand* [19]. En nuestra presentación anterior, usamos $\boldsymbol{c}$ para los puntos en $\mathbb{R}^3$, esta notación es extensible para puntos en n dimensiones y para un punto que se encuentran en $\mathbb{R}^2$ también podemos usar $\boldsymbol{c}$. Por ejemplo, se puede decir que el punto $\boldsymbol{c} \in \mathbb{R}^2$ también pertenece al espacio $\mathbb{R}^3$ si se considera que el punto se encuentra sobre el plano $\vec{\boldsymbol{x}}\vec{\boldsymbol{y}}$ y, por lo tanto, su tercer coordenada c_3 es igual a cero.

4.1 Métodos

Para aprovechar los métodos que hemos desarrollados en la aplicación de reconstrucción de imágenes 3D de ultrasonido, decidimos utilizar variaciones de los métodos Voxel-Based Reconstruction (VBR) y Pixel-Based Reconstruction (PBR) para la reconstrucción de imágenes 3D de ultrasonido. Estos métodos son los más simples basados en búsqueda de vecindades que se usan al hacer reconstrucción 3D de imágenes de ultrasonidos [20], que, junto con los métodos basados en funciones (p. ej., Bézier) presentan los mejores tiempos reportados en la literatura [19]. Ambos métodos están basados en la búsqueda

de información en una región especifica (*kernel*) del espacio para poder asignar los valores a una imagen 3D de ultrasonido. Es importante resaltar que se seleccionaron los métodos VBR y PBR sobre los basados en funciones debido a que se basan en la búsqueda de información utilizando un *kernel* $\mathcal{K}$ en un área específica. Esta búsqueda puede ser resuelta de manera directa utilizando los métodos propuestos en esta tesis. Con esto logramos reducir el tiempo computacional requerido para realizar reconstrucciones 3D sin comprometer la precisión del resultado final.



(a) (b)

Figura 4.1: (a) En el método VVN-OM, un *kernel* $\mathcal{K}$ (rosa) es centrado en un vóxel $\boldsymbol{v} \in V$ que no contiene un píxel de $\mathcal{C}$ dentro de su vecindad de Voronoi. El tamaño del *kernel* es incrementado (azul) hasta que se intersecte un píxel dentro de $\mathcal{C}$; los valores de estos píxeles contribuirán al valor final de $\boldsymbol{v}$. (b) En el método PVN-OM, primero (izquierda), los valores de cada píxel mapeado $\boldsymbol{c} \in \mathcal{C}$ son asignados a su vóxel más cercano $\boldsymbol{v} \in V$ (etapa de rellenado de intervalos o *bin-filling*). Después (derecha), cada $\boldsymbol{v} \in V$ que no haya sido llenado con anterioridad, es visitado y se le asigna un valor ponderado por la distancia de la combinación de valores de todos lo vóxeles que ya hayan sido llenados y que estén dentro del *kernel* $\mathcal{K}$ centrado en $\boldsymbol{v}$ (etapa de rellenado de hoyos o *hole-filling*).

4.1.1 Reconstrucción Basada en Vóxeles

Este método produce una imagen 3D a partir de un conjunto de imágenes 2D de ultrasonido. El método inicia atravesando todo vóxel $\boldsymbol{v}$ en una rejilla (*grid*) V que contendrá la reconstrucción final. Para cada vóxel $\boldsymbol{v}$ se buscarán todos los píxeles cercanos al conjunto de imágenes 2D que contribuyan al valor final que será asignado al vóxel. Para realizar la búsqueda de estos píxeles cercanos, se utiliza un proceso iterativo donde se

inicia con un *kernel* de búsqueda $\mathcal{K}$, el tamaño de este *kernel* está predefinido previamente, centrado en el $\boldsymbol{v}$ correspondiente. Con esto se filtrarán todos los píxeles que estén dentro de $\mathcal{K}$. Si esta búsqueda no obtiene ningún píxel dentro del *kernel*, el tamaño de $\mathcal{K}$ es incrementado y la búsqueda inicia de nuevo. Este proceso se repite hasta que se encuentre al menos un píxel o el tamaño del *kernel* alcance un tamaño máximo. En la Figura 4.1(a) se presenta este proceso de manera general.

4.1.2 Reconstrucción Basada en Píxeles

La implementación más común de este método es denominado PNN-VN2, el cual consiste en realizar dos pasos para realizar la reconstrucción final. Al iniciar el método se realiza la asignación de todos los píxeles $\boldsymbol{c}$ en el conjunto de imágenes de ultrasonido a un vóxel $\boldsymbol{v}$ dentro de V, conjunto de vóxeles que componen la imagen 3D final. Esto se realiza buscando el vóxel más cercano a cada píxel. En la practica existen diferentes formas para considerar los valores de varios píxeles para asignar el valor a un vóxel, tales como asignar el valor máximo, promediar los valores de intensidad o elegirlos acorde al orden en que son encontrados. El segundo paso del método consiste en asignar valores a todos los vóxeles que no fueron visitados en el paso inicial, este proceso es similar al primer paso del método VBR explicado previamente, con la excepción de que en esta ocasión se buscarán valores de intensidad de los vóxeles que estén dentro del *kernel* $\mathcal{K}$ (cuyos valores ya han sido calculados en el primer paso). Un esquema ilustrando este proceso es mostrado en la imagen 4.1(b).

4.1.3 Implementación usando *Octrees* y claves Morton

Como se exploró en capítulos anteriores, los *octrees* son útiles para dividir el espacio en octantes de manera recursiva hasta un profundidad específica. Los algoritmos VBR y PBR transforman imágenes 2D a un espacio 3D para crear una imagen 3D. Los *octrees* con claves Morton pueden ser adaptados a este proceso con la intención de acelerar el proceso de reconstrucción. Para lograr esto es necesario primero crear una *octree* con las dimensiones necesarias para almacenar las imágenes de ultrasonido.

El proceso consiste en tres pasos principales ilustrados en la Figura 4.2, el primero consiste en construir una nube de puntos basado en el conjunto de imágenes de ultrasonido. Esto es realizado primero aplicando una matriz de transformación, obtenida durante la calibración del sistema *freehand*, esta transformación nos permitirá localizar

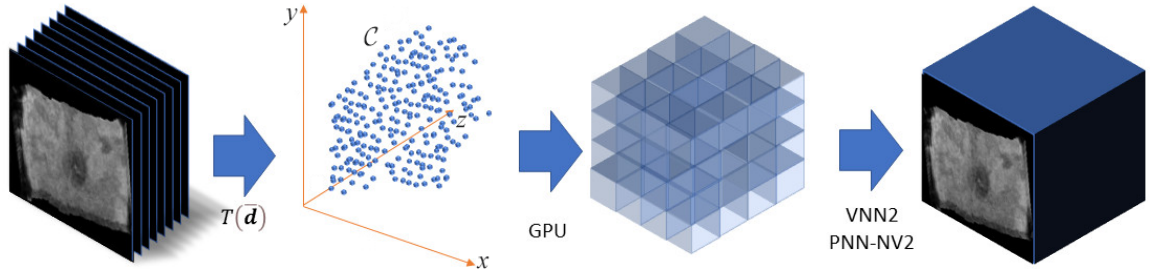


Figura 4.2: Secuencia metodológica propuesta para la generación de imágenes 3D después de adquirir (a) un conjunto de imágenes 2D de US. (b) Los píxeles en todas las imágenes son usados para generar una nube de puntos delimitada $\mathcal{C}$. (c) Un *octree* con claves Morton que completamente encierra a la nube de puntos $\mathcal{C}$. (d) La imagen final 3D f producida por un método de reconstrucción basado en un *octree*.

las imágenes 2D de ultrasonido como planos en un espacio 3D. Tomando en cuenta los parámetros internos de la sonda de ultrasonido es posible asignar un tamaño al píxel de cada imagen. Este paso nos dará como resultado un conjunto de planos en el espacio y usando los centros de cada píxel en las imágenes 2D de ultrasonido es posible construir un nube de puntos $\mathcal{C}$.

Haciendo uso de la nube de puntos encontrada es necesario, primero, encontrar el mínimo volumen V que englobe a todos lo puntos en la nube, esto con el propósito de determinar el tamaño del *octree*. La región obtenida de este volumen será representada por el nodo raíz o_r del *octree*. Como se mencionó con anterioridad en la Sección 2.3, el *octree* se compone de una subdivisión recursiva de octantes. La subdivisión del cuboide representado por o_r obtiene los octantes que componen al nivel 1 del *octree*. Cada octante del nivel 1 es subdividido en octantes más pequeños agregando nuevos niveles al *octree*. Este proceso se continuara hasta alcanzar una profundidad establecida previamente o hasta que el nivel deseado para la nube de puntos sea alcanzado (p.ej., hasta que cada octante en los nodos hoja contenga un solo punto de la nube $\mathcal{C}$). Debido a que nuestra implementación de *octrees* utiliza codificación de claves Morton usando una variable de tipo *int*, se tiene un máximo de 32 bits y cada nivel del *octree* ocupa 3 bits. Por lo que sólo se dispondrá de un máximo de 10 niveles del *octree*, ver la Sección 3.1, aunque esto puede ser incrementado usando una variable de mayor tamaño para almacenar las claves Morton; para la reconstrucción de imágenes de ultrasonido se determinó que 10 niveles del *octree* (1024×1024×1024) vóxeles son suficientes para las resoluciones de imágenes de sondas de ultrasonido. Al terminar la construcción y

subdivisión del *octree*, esté contendrá el espacio ocupado por las nube de puntos $\mathcal{C}$.

Al tiempo que se procesa la nube de puntos $\mathcal{C}$ es posible asignar una clave Morton a cada punto, cada vóxel del *octree* contará con su propia clave Morton asignada. Debido a las relaciones explicadas en el Capítulo 3 es posible encontrar la correspondencia directa entre cada punto en el espacio y el correspondiente vóxel que engloba el espacio que lo contiene, esto usando las claves Morton como filtros, ver Figura 3.8. Una vez que el *octree* ha sido construido y la claves Morton han sido asignadas tanto a los vóxeles como a los puntos en $\mathcal{C}$, esto nos servirá como base para iniciar el procesamiento de los algoritmo PBR y VBR respectivamente. En este trabajo aplicamos la búsqueda de vecindades dentro de los nodos hoja del *octree* para producir la imagen 3D final. Debido a que nuestra implementación realiza la búsqueda de vecindades usando las claves Morton para cada uno de lo métodos implementados, nos referimos a la implementación con claves Morton y *octree* del método VBR como método *Voxel Vicinity Neighbor* con *octree* y claves Morton (VVN-OM) y a la implementación del método PBR como *Pixel Vicinity Neighbor* con *octree* y claves Morton (PVN-OM).

4.1.4 Reconstrucción de imágenes 3D

Se implementaron los *octree* y claves Morton para ambos métodos ya que no existía información preliminar de que alguno de los métodos pudiera tener un mejor desempeño usando nuestras metodologías y debido a que ambos métodos eran dependientes de una búsqueda de vecindades. Cabe resaltar que al finalizar el proceso previamente mencionado, muchos vóxeles asociados a nodos hoja del *octree* quedarán vacíos o contendrán en su espacio más de un punto $\boldsymbol{c}$. El *octree* generado nos servirá como base para realizar la implementación de los algoritmos VVN-OM y PVN-OM, usando los nodos hoja y sus respectivas claves Morton para filtrar puntos y buscar vecindades con la finalidad de obtener una imagen 3D al finalizar cada algoritmo de reconstrucción. Los algoritmos VVN-OM y PVN-OM realizan búsquedas dentro del *octree* en lugar de realizarlo en el espacio 3D directamente, esto acelerará el proceso de reconstrucción ya que las claves Morton nos llevarán a encontrar directamente la información que se necesita sin realizar búsquedas exhaustivas en la nube de puntos.

Para el caso del método VVN-OM, primero es necesario atravesar todo nodo hoja o_f en el *octree* $\mathcal{O}$, esto con el objetivo de encontrar el valor final asociado a cada $\boldsymbol{v}(o_f)$; este valor es obtenido buscando todos los puntos en $\mathcal{C}$ en una área de búsqueda especifica, en el caso de nuestra implementación dicha búsqueda es realizada, primero, buscando

la vecindad de nodos hoja de o_f usando un *kernel* $\mathcal{K}_{\boldsymbol{v},\rho}$ de radio ρ que está centrado en el vóxel v (es decir, $\{\boldsymbol{c} \in \mathcal{C} | \boldsymbol{c} \cap \mathcal{K}_{\boldsymbol{v},\rho} \neq \emptyset\}$) y segundo, usando las claves Morton de los nodos encontrados usando $\mathcal{K}_{\boldsymbol{v},\rho}$ para filtrar los puntos en la nube $\mathcal{C}$ y por último, asignando un valor de intensidad a $\boldsymbol{v}$. Este proceso inicia con la búsqueda de un *kernel* $\mathcal{K}_{\boldsymbol{v},\rho}$ cuyo tamaño ρ cubre completamente solo los vóxeles adyacentes a $\boldsymbol{v}$; se usan las claves Morton de los vóxeles adyacentes para filtrar los puntos de la nube y si no se encuentran puntos en $\mathcal{C}$ usando estas claves Morton entonces el *kernel*, cuyo tamaño es ρ, es incrementado, y la búsqueda es restringida a los nuevos vóxeles agregados, ver la Figura 4.1(a). Esta búsqueda es realizada usando operaciones binarias sobre la clave Morton del nodo $\boldsymbol{v}(o_f)$ debido a que la información espacial de cada vóxel está codificado en la clave Morton y esta debe ser similar en vóxeles adyacentes. Cabe resaltar que el *kernel* de búsqueda $\mathcal{K}_{\boldsymbol{v},\rho}$ estará compuesto de las Claves Morton de todos los nodos adyacentes y el nodo central $\boldsymbol{v}$. Usando esta lista de claves Morton es posible filtrar los puntos en la nube $\mathcal{C}$. El arreglo $\mathcal{L}$ que contiene todas claves Morton de $\mathcal{C}$ son previamente ordenadas acorde al valor de su clave Morton, el indice del primer punto de cada nodo o_f puede ser directamente almacenado por su nodo o_f respectivo. Usando la clave Morton de cada nodo o_f se puede recorrer el arreglo $\mathcal{L}$ a partir del índice almacenado, una vez que la clave Morton en el arreglo cambia lo suficiente (esta sensibilidad del cambio depende de la profundidad del *octree*) significa que hemos pasado a otro nodo o_f por lo que nos detenemos y concluimos que hemos encontrado todos los puntos en $\mathcal{C}$ para el nodo hoja con la clave Morton respectiva. Este proceso es realizado para todas la claves Morton de los nodos encontrados usando el *kernel* de búsqueda $\mathcal{K}_{\boldsymbol{v},\rho}$ para que al terminar encontremos todas los puntos en $\mathcal{C}$ dentro del área de búsqueda.

Esta búsqueda está representada por el conjunto dado por

$$\mathcal{Q}_{\boldsymbol{u}} = \left\{ o_f | \mathcal{V}_{\boldsymbol{v}(o_f)} \cap \mathcal{K}(\boldsymbol{u}, \rho) \neq \emptyset \right\}, \tag{4.1}$$

donde el *kernel* $\mathcal{K}$ está centrado en $\boldsymbol{u}$ y se ajusta a un radio esférico ρ, y $\mathcal{V}_{\boldsymbol{v}(o_f)}$ es la vecindad de Voronoi del vóxel $\boldsymbol{v}$ en V asociado con el nodo hoja o_f. El *kernel* de búsqueda puede tomar múltiples formas, incluyendo cubico o esférico, siempre y cuando esté contenido en la esfera formada por el radio ρ. Debido al que en nuestra implementación el uso de claves Morton es la base de la búsqueda y filtrado, el tamaño del *kernel* de búsqueda podrá únicamente ser incrementado acorde al tamaño de los vóxeles asociados a los nodos hoja o_f en el *octree* en cada eje coordenado. Esto nos da

como resultado un *kernel* de búsqueda cúbico y que solo podrá mantener esta forma conforme se incrementa su área de búsqueda.

Después de obtener $\mathcal{Q}_{\boldsymbol{u}}$ para todos los vóxeles vecinos a $\boldsymbol{u}$, el método buscará todos los píxeles mapeados al conjunto. Este proceso crea, para todo nodo hoja $o_f \in \mathcal{Q}_{\boldsymbol{u}}$, un grupo de puntos de la siguiente forma:

$$\mathcal{P}_{\boldsymbol{u},o_f} = \left\{ \boldsymbol{c} \in \mathcal{C} | \boldsymbol{c} \cap \mathcal{V}_{\boldsymbol{v}(o_f)} \neq \emptyset \text{ and } \|\boldsymbol{u} - \boldsymbol{c}\| \leq \rho \right\}. \tag{4.2}$$

Este proceso restringe la búsqueda a puntos contenidos en los vecindarios de Voronoi asociados a los nodos hoja $o_f \in \mathcal{Q}_{\boldsymbol{u}}$; así que se descartan grandes cantidades de puntos que no son relevantes para la búsqueda [21], ver la Figura 3.11 para una ilustración del proceso de búsqueda de vecindades.

Es importante mencionar que cuando un píxel es encontrado este no aportará directamente su valor de intensidad al vóxel $\boldsymbol{v}$, sino que será ponderado utilizando la siguiente función de distancia-peso [20]

$$\phi(\boldsymbol{v}) = \frac{1}{|\mathcal{K}_{\boldsymbol{v},\rho}|} \sum_{\boldsymbol{u} \in \mathcal{K}_{\boldsymbol{v},\rho}} g(\boldsymbol{u})\, e^{-\left(\frac{\|\boldsymbol{u}-\boldsymbol{v}\|}{\sigma}\right)^2}, \tag{4.3}$$

donde $|\mathcal{K}_{\boldsymbol{v},\rho}|$ es el número de puntos en $\mathcal{C}$ que han sido encontrados dentro del *kernel* de búsqueda $\mathcal{K}_{\boldsymbol{v},\rho}$, la función g obtiene el valor de intensidad correspondiente a cada punto $\boldsymbol{u} \in \mathcal{C}$, $\|\bullet\|$ es la norma l_2 y σ es la distancia que existe entre el centro de $\mathcal{K}_{\boldsymbol{v},\rho}$ y la posición valida más lejana.

En contraste, el algoritmo PVN-OM inicia procesando directamente todos los puntos dentro de $\mathcal{C}$ y, haciendo uso de sus claves Morton, encuentra los vóxeles más cercanos a ellos. En esta etapa el método asigna el valor de cada punto $\boldsymbol{c}$ en $\mathcal{C}$ a cada vóxel $\boldsymbol{v}$ en $\mathcal{V}$, cuando el punto $\boldsymbol{c}$ está dentro de la región definida por el vóxel $\boldsymbol{v}$. Cabe resaltar que este proceso es altamente paralelizable debido a la independencia de datos que tenemos usando las claves Morton, las claves de los puntos pueden ser directamente convertidas a posiciones de los nodos del *octree* dentro del arreglo lineal de nodos, esto debido a que tanto los nodos como los puntos comparten claves Morton que fueron creadas bajo la misma discretización del espacio definida por el *octree*.

Cuando existe más de un punto $\boldsymbol{c} \in \mathcal{C}$ dentro de $\mathcal{V}_{\boldsymbol{v}}$ (o sea., $|\{\boldsymbol{c} \in \mathcal{C} | \mathcal{V}_{\boldsymbol{v}} \cap \boldsymbol{c} \neq \emptyset\}| > 1$), los valores asociados a todos los puntos se promedian para determinar el valor final de $\boldsymbol{v}$. Al terminar este proceso denominado etapa de rellenado de intervalos o *bin-filling*, todos los vóxeles que no han sido visitados por el recorrido en la nube de

puntos son considerados como nodos vacíos. Con el propósito de llenar estos nodos vacíos considerados "huecos", se realizará un proceso similar al aplicado en el VVN-OM usando (4.1). Sin embargo, ahora una vez encontrada la vecindad de nodos se usarán los valores de intensidad asociados a estos nodos y ya no será necesario buscar los puntos en $\mathcal{C}$. La búsqueda se realizará de forma similar al método VVN-OM, usando un *kernel* $\mathcal{K}$ pero buscando todos los vóxeles $\boldsymbol{v}(o_f)$ para todos los nodos o_f en $\mathcal{O}$ cuyos valores fueron asignados durante el proceso *bin-filling*. Es importante que cualquier vóxel cuyo valor fue asignado durante el proceso de etapa de rellenado de hoyos o *hole-filling* es considerado como un vóxel sintético y es excluido para el resto del proceso, ver Figura 4.1(b).

Las implementaciones de ambos métodos fueron realizados en CUDA®, aprovechando la capacidad de poder pasar entre el espacio de los puntos y el espacio de los vóxeles usando las claves Morton, esto permite una independencia de datos que lo vuelve altamente paralelizable en cada etapa de la reconstrucción. Debido a esto la implementación de ambos métodos pudo ser realizada utilizando únicamente 4 *kernels* de CUDA® que permiten evitar conflictos entre las diversas etapas de los algoritmos. El primer *kernel* de CUDA® será responsable de la creación del *octree* sin información; esto se realiza procesando todo el arreglo lineal $\mathcal{L}$ de nodos mientras se crean las claves Morton correspondientes. El segundo *kernel* de CUDA® procesa todos los puntos, asigna claves Morton y ordena el arreglo $\mathcal{L}$ de puntos. Finalmente, el proceso de *bin-filling* y *hole-filling* son realizados en los dos *kernels* de CUDA® restantes. Para el caso del método PVN-OM es necesario realizar operaciones atómicas de CUDA® para asegurar que los diferentes hilos (*threads*) no accedan de manera concurrente al mismo nodo. En ambos métodos, diferentes *kernels* son responsable por el proceso *hole-filling*.

4.1.5 Resultados

Para realizar la validación de las técnicas implementadas en los métodos PVN-OM y VVN-OM se realizaron experimentos con el propósito de medir el desempeño de las implementaciones. Se construyó *software* para ambos métodos usando GPUs. Buscando un punto de comparación se realizaron las implementaciones de los métodos sin hacer uso de *octrees* y claves Morton, estas versiones serán referidas como PVN y VVN, en ambas implementaciones sin *octrees* se utilizó una rejilla 3D y la nube de puntos directamente. Todas las implementaciones hacen uso de CUDA® para hacer uso eficiente de los recursos proporcionados por el hardware (tarjetas de video) de Nvidia®. Con

el propósito de realizar validaciones no solo de velocidad, se diseñaron pruebas de precisión y resolución con el propósito de probar lo robusto del método y que la velocidad alcanzada no haya sido sacrificando precisión o resolución en el método.

El *hardware* utilizado para todos nuestro métodos consistió en dos sistemas que usan un procesador Intel® Core™ i7 con 4 núcleos, con una velocidad de 4.2 GHz y 16 GiB de RAM. Uno de estos equipos de cómputo usa una tarjeta de video GeForce® GTX 1080, con arquitectura Pascal™ y 6 GiB de memoria, mientras el otro usa una GeForce® RTX 2060 como tarjeta de video, con arquitectura Turing™ y también 6 GiB de memoria.

La información a utilizar para la pruebas era esencial como punto de referencia. Esta información se obtuvo de diversas fuentes, una de las principales fue la generada en el trabajo de F. Torres [22]. Se utilizó un sistema de ultrasonido 3D de adquisición manual (FUS) que hace uso de un tracker óptico (Polaris Spectra by Northern Digital Inc., Canada) y un sistema 2D US (Aloka SSD-1000 Diagnostic Ultrasound System by ALOKA Co., Ltd., U.S.A.) para generar parte de la información experimental. Los sistema deben ser calibrados con el propósito de obtener la nube de puntos $\mathcal{C}$. Para la calibración del sistema FUS se utilizo la técnica del hilo cruzado (2 hilos cruzados sumergidos en agua que se cruzan en el centro), se seleccionó esta metodología debido a su facilidad de construcción y su simplicidad. El proceso de calibración se realizó siguiendo el procedimiento descrito en [23]; una sonda 2D de ultrasonido fue usada para adquirir el conjunto de imágenes 2D en diferentes orientación sobre la región del hilo cruzado y calcular los valores para la transformación afín para cada plano escaneado. Con el equipo mencionado anteriormente se realizaron escaneos de 2 phantoms distintos. El primero fue un phantom de alcohol polivinilico (PVA) que fue construido para realizar la simulación del tejido mamario con una lesión (tumor). El segundo phantom usado fue uno de propósito general para ultrasonido, modelo 044 manufacturado por CIRS, Inc., E.U.A.; este phantom consiste en un cuboide con medidas de 18×13×20 cm^3, con diversos cilindros acomodados en dos planos. El diseño del material del phantom tiene el propósito de simular las propiedades del tejido de hígado humano. Para propósitos de los experimentos mostrados en esta sección solo se utilizaron solo uno de los planos mencionados que contienen cilindros.

Las imágenes de ultrasonido, producidas en los experimentos reportados en [22] tienen todas con dimensiones de 387×400 píxeles (px), para los phantoms mencionados previamente, esto usando el sistema Aloka con una frecuencia de 7 MHz. Para el caso del phantom de PVA, se obtuvo un solo conjunto de datos que contiene 131 imágenes.

Para el caso del *phantom* CIRS (GPUP), se adquirieron dos conjuntos de datos. El primero con un total de 66 imágenes de ultrasonido, enfocándonos en el plano que contienen los cilindros de 1.5 mm de diámetro, cubriendo solo 6 de los cilindros. El segundo conjunto de datos con un total de 51 imágenes de la región que contiene los cilindros de 3 mm, cubriendo solo nueve de los cilindros.

4.1.6 Tiempo de reconstrucción y uso de memoria en GPU

# Imágenes 2D	Images 3D (vóxeles)	Tiempo (segundos)			
		PVN-OM	VVN-OM	PVN	VVN
131	128×128×128	0.72	1.07	4.98	2.71
	256×256×256	7.86	8.71	24.47	16.22
	512×512×512	79.79	86.28	121.71	239.55
66	128×128×128	0.54	0.84	5.07	1.53
	256×256×256	6.19	6.93	26.07	17.15
	512×512×512	70.57	70.84	107.40	185.89
51	128×128×128	0.44	0.80	4.56	1.02
	256×256×256	5.22	6.36	23.89	12.95
	512×512×512	71.16	67.64	104.48	150.23

Tabla 4.1: Tiempos medidos para realizar la reconstrucción de imágenes 3D de ultrasonido con diferentes dimensiones, basados en los conjuntos de imágenes 2D de ultrasonido con dimensiones 387×400 px usando los métodos basados en píxeles y vóxeles con y sin *octrees* y claves Morton.

Para registrar los tiempos computacionales de nuestro métodos PVN-OM y VVN-OM se midió el tiempo requerido para reconstruir las imágenes 3D con dimensiones de 128×128×128, 256×256×256, y 512×512×512 vóxeles. Comparamos los tiempos de ejecución con los tiempos medidos para los métodos PVN y VVN que no hacen uso de claves Morton ni de *octrees*. Los experimentos mencionados fueron realizados usando el sistema que hace uso de una GPU GTX 1080 GPU y las imágenes 2D US de entrada fueron obtenidas del CIRS y el phantom PVA. En la Tabla 4.1, se muestran los tiempos que necesitan cada métodos para reconstruir las diversas imágenes 3D. Una imagen de dichas reconstrucciones es mostrada en la Figura 4.3, esta es un ejemplo de una imagen 3D reconstruida del phantom PVA, con dimensiones de 256×256×256 vóxeles.

Para realizar un medición más precisa de la aceleración lograda al implementar

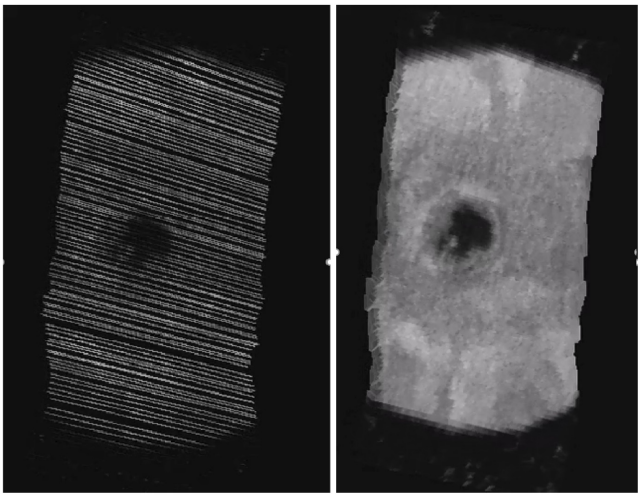


Figura 4.3: Corte axial de una imagen 3D con dimensiones 256×256×256 vóxeles producida a partir de del conjunto de datos adquirido del phantom PVA. El valor de cada vóxel en la imagen 3D es asignado (izquierda) usando directamente las imágenes adquiridas por la sonda 2D de ultrasonido y (derecha) aplicando el método VVN-OM a las imágenes 2D adquiridas.

octree y claves Morton, se utilizó la siguiente medida:

$$SpeedUp = \frac{T_n}{T_o}, \tag{4.4}$$

donde T_n es el tiempo usado para el método que no hace uso de un *octree* y T_o es el tiempo requerido para el método que hace uso de un *octree* con claves Morton. Los resultados mostrados en la Tabla 4.1 muestran una aceleración de 10.28× para el método PVN-OM con el conjunto de datos de 51 imágenes 2D de ultrasonido, reconstruyendo una imagen 3D con 128×128×128 vóxeles. La tasa de aceleración más baja fue de 1.25×, la cual fue obtenida con el método VVN-OM usando el conjunto de datos con 51 imágenes 2D de ultrasonido y reconstruyendo una imagen 3D de ultrasonido con dimensiones de 128×128×128 vóxeles.

En una revisión del estado del arte, el trabajo reportado por Huang et al. [19] reportó que la implementación secuencial más rápida del método basado en píxeles y usando búsqueda de vecindades requirió 0.031 s para procesar imágenes 2D de ultrasonido de 302×268 px y reconstruir una imagen 3D US con 90×81×192 vóxeles. En comparación, nuestro método PVN-OM toma solo 0.005 s para imágenes 2D US de 387×400 px y reconstruir una imagen 3D US con 128×128×128 vóxeles (aproximadamente 40% más vóxeles).

Como parte de las pruebas realizadas, también se realizaron mediciones para determinar el uso de memoria para ambos métodos al reconstruir imágenes 3D US. Los resultados obtenidos son mostrado en la Tabla 4.2. En nuestras implementaciones, cada nodo del *octree* usar un total de 0.015625 KiB, y un *octree* de nivel (256×256×256 nodos) requiere 292.57 MiB de memoria. Si este *octree* es incrementado en un nivel, el numero adicional de nodos será igual a 512×512×512, y la memoria adicional requerida será igual a 2.28 GiB.

Imágenes 2D (387×400 px)	Imagen 3D (vóxeles)	Memoria (MiB) PVN-OM	VVN-OM
131	128×128×128	1,390	1,538
	256×256×256	1,614	1,762
	512×512×512	3,406	3,554
66	128×128×128	1,278	1,388
	256×256×256	1,502	1,612
	512×512×512	3,294	3,404
51	128×128×128	1,196	1,280
	256×256×256	1,420	1,504
	512×512×512	3,212	3,296

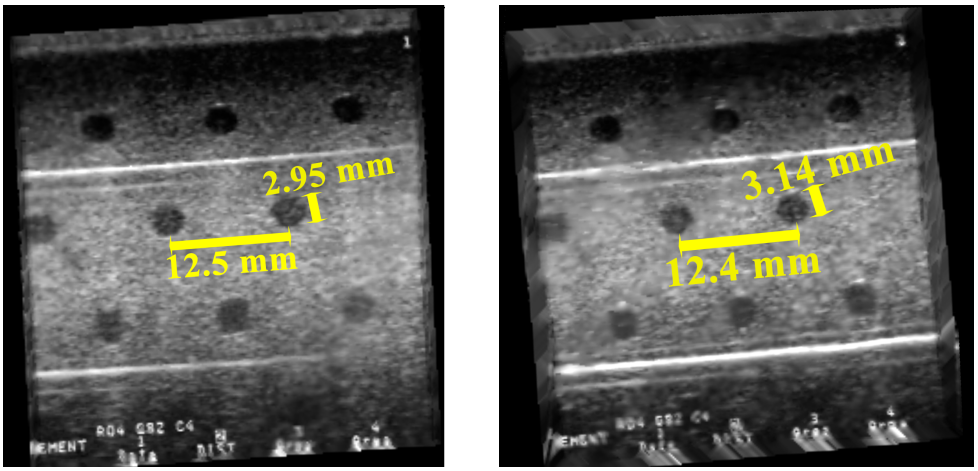
Tabla 4.2: Memoria usada para reconstruir imágenes 3D de ultrasonido con diferentes dimensiones, usando los métodos PVN-OM y VVN-OM.

4.1.7 Pruebas de resolución

Las reconstrucciones de imágenes 3D también fueron medidas en términos de la resolución que pueden lograr, para estas pruebas de utilizaron los datos adquiridos de planos usando el phantom GPUP. Las imágenes 3D obtenidas tienen dimensiones de 64×64×64, 128×128×128, 256×256×256, y 512×512×512 vóxeles. El proceso de reconstrucción se realizó utilizando un *kernel* de búsqueda de tamaño máximo igual a 9×9×9 vóxeles, esto para ambos métodos. Después de obtener las imágenes 3D, manualmente se realizaron medidas de los cilindros y sus respectiva separación. La Tabla 4.3 presenta los valores promediados del diámetro de los cilindros y su desviación estándar para las diferentes imágenes 3D reconstruidas. La Figura 4.4 muestra dos imágenes 3D reconstruidas de 256×256×256 de la sección donde los cilindros de 3 mm están localizadas.

	Grupo de cilindros de 1 (mm)		Grupo de cilindros de 2 (mm)	
	Diámetro	Separación	Diámetro	Separación
VVN-OM	1.49±0.05	12.60±0.13	2.95±0.08	12.60±0.10
PVN-OM	1.54±0.03	12.60±0.10	3.14±0.02	12.60±0.19

Tabla 4.3: Media de los valores medidos del diámetro y separación de los cilindros en las reconstrucciones 3D del phantom CIRS (GPUP).



(a) (b)

Figura 4.4: Reconstrucciones 3D con dimensiones de 256 × 256 × 256 vóxeles en la región del Phantom CIRS que contienen los cilindros de 3 mm; es importante notar que las líneas rectas delimitan las región que contiene los cilindros. Las reconstrucciones fueron obtenidas con los métodos (a) VVN-OM and (b) PVN-OM. Las líneas amarillas y el texto ilustran las medidas adquiridas y sus correspondientes valores.

4.1.8 Error de reconstrucción

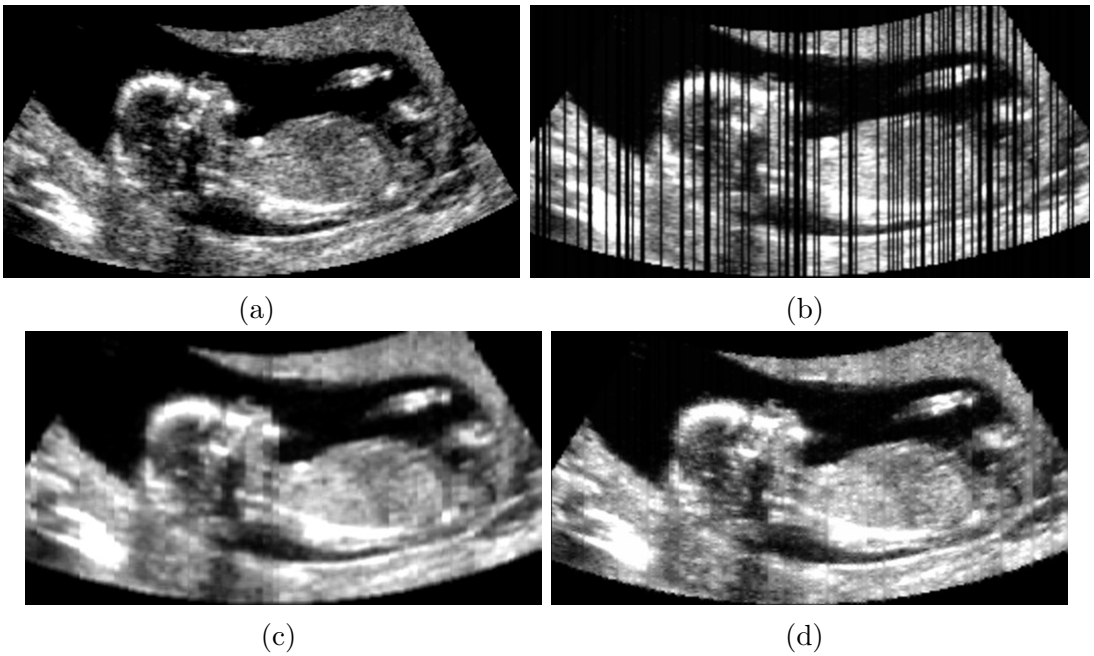
Por último se determinó la precisión de los métodos desarrollados, midiendo que tan buenas precisos son los valores en las imágenes 3D reconstruidas, esto se realizó utilizando imágenes de ultrasonido 3D que fueron usadas como punto de referencia. Para cada imagen 3D de ultrasonido, se seleccionaron diversos cortes perpendiculares a los planos $\vec{x}\vec{y}$, $\vec{x}\vec{z}$ o $\vec{y}\vec{z}$. Esta metodología fue elegida debido a que replica las manera en el que los conjuntos de datos son generados a partir de la técnicas de adquisición *freehand* para imágenes de ultrasonido 2D. Sin embargo, para poder simular un escenario más realista, huecos de información fueron creados de manera aleatoria, esto al remover información (imágenes 2D de ultrasonido) en diferentes cantidades. Esta metodología permite determinar el error asociado a los algoritmos de reconstrucción, mientras se evita introducir cualquier error que pueda ser generado durante la adquisición de imágenes

2D de ultrasonido. Una ventaja adicional de este método es la robustez de las pruebas, permitiendo un punto de referencia para las imágenes 3D reconstruidas con ambos métodos.

Se midió la diferencia de los valores en la imagen 3D reconstruida con respecto a la imagen 3D original, y se calculó el error cuadrático medio normalizado (MSE):

$$\mathrm{MSE} = \frac{1}{N}\sum_{i=1}^{N}\left(v_r\left(\boldsymbol{v}_i\right) - v_b\left(\boldsymbol{v}_i\right)\right)^2, \tag{4.5}$$

donde $v_r\left(\boldsymbol{v}_i\right)$ y $v_b\left(\boldsymbol{v}_i\right)$ son los valores para el i-ésimo vóxel $\boldsymbol{v}$ en la imagen 3D reconstruida y la original, respectivamente, y N es el número total de vóxeles. Dada una imagen 3D original de ultrasonido, varios grupos de datos con imágenes 2D de ultrasonido son creados, esto con diferentes cantidades de información cada uno, usando rangos desde la no remoción de información hasta la exclusión del 100%. El error producido sin información removida se utilizó para normalizar el error obtenido en la reconstrucción de las imágenes 3D restantes. Denotamos a este error base como $\mathrm{MSE_M}$. El error final reportado para una reconstrucción 3D de ultrasonido es la relación entre MSE y $\mathrm{MSE_M}$, y denotamos a esta relación como $\overline{\mathrm{MSE}}$.



(a) (b) (c) (d)

Figura 4.5: (a) Plano sagital del imagen 3D original de un feto. (b) El mismo plano en la imagen 3D con 30% de vóxeles removidos. (c, d) Los planos correspondiente en las imágenes 3D reconstruidas con los métodos (c) PVN-OM y (d) VVN-OM.

Para realizar estas pruebas se obtuvieron 92 imágenes 3D de ultrasonido como punto de referencia: 20 cerebros, 61 fetos, y 11 corazones de fetos; estas imágenes fueron generadas con un sistema General Electric Voluson 3D US usando una frecuencia que oscila entre 2 to 7 MHz. También se obtuvieron 31 imágenes de referencia que representan tejido de mamas con diferentes características. Estas imágenes están disponibles de manera pública en la base de datos con co-autoría del Departamento de Radiología de la Universidad de Tahammasat y el Queen Sirikit Center de Cancer de Mama en Tailandia [24]. En la Figura 4.5, se presentan diversos planos sagitales en una de las imágenes 3D de fetos, ilustrando las medidas del proceso para calculo de error.

Estas pruebas se realizaron usando el sistema de cómputo con una tarjeta de video RTX 2060 para ambos métodos, se reportaron los valores de la $\overline{\text{MSE}}$ y sus desviaciones estándar para todas las imágenes 3D de ultrasonido reconstruidas en la Figura 4.6.

Con el propósito de poder determinar si la utilización de *octree* y Claves Morton introducen errores en los métodos basados en píxeles y vóxeles. Se compararon las implementaciones de los métodos PVN y VVN con sus respectivas versiones usando *octrees* y claves Morton. La Figura 4.7 muestra una comparación de los valores del $\overline{\text{MSE}}$ obtenidos por los métodos PVN y PVN-OM, los métodos VVN y VVN-OM muestras un comportamiento similar. Las diferencias promediadas entre los errores producidos por ambos métodos es de 0.00089, con el método PVN-OM teniendo un error más pequeño. Estas diferencias pueden ser consideradas como despreciables. Los errores presentados por ambos métodos puede ser atribuido a la precisión al calcular las distancias a las vecindades, en particular, el método empleado para cuantificar distancias al usar *octree* y claves Morton. Estos resultados sugieren que los métodos que hacen uso de claves Morton y *octrees* no introducen ningún error significativo durante el proceso de reconstrucción.

4.2 Discusión

A pesar de las variadas implementaciones y propuestas que han surgido entorno al procesamiento de ultrasonido en 3D, aún existe la necesidad de realizar optimizaciones que permita reconstrucciones 3D de ultrasonido de manera más rápida y eficiente con el propósito de proporcionar resultados en tiempo real o cercanos a tiempo real, esto a pesar del incremento en la complejidad y tamaño de los datos obtenidos de los pacientes. Diversas aproximaciones han sido propuestas para lograr este mejor rendimiento, la

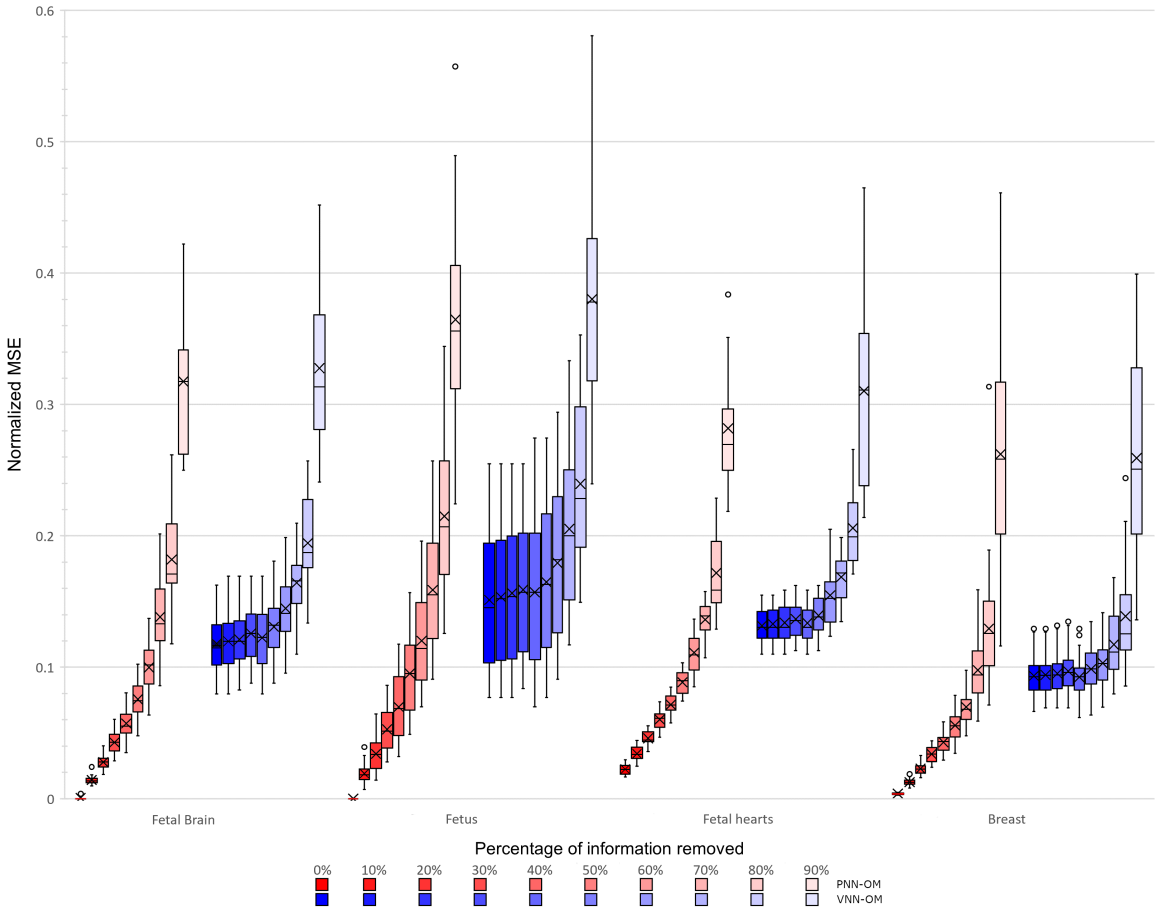


Figura 4.6: Incremento en el $\overline{\text{MSE}}$ como función de la cantidad de información eliminada de los conjuntos de datos (representado por las diferentes tonalidades de color) cuando se producen imágenes 3D con los métodos (red) PVN-OM y (azul) VVN-OM. Para estas pruebas se utilizaron los conjuntos de datos: Cerebros Fetales, Fetos, Corazones Fetales y Mamas.

implementación planteada en esta sección propone una nueva implementación de algoritmos de reconstrucción basados en búsqueda de vecindades. Las implementaciones que fueron descritas prueban aportar un aceleramiento al proceso de reconstrucción, esto con la implementación de un *octree* para la representación de vóxeles y el uso de claves Morton como método de codificación para los vóxeles y su información contenida. Esto permite implementaciones directas de los algoritmos de reconstrucción, logrando una velocidad de reconstrucción y uso de memoria más eficiente que métodos previamente propuestos.

La evaluación de los dos métodos propuestos sugieren que el método PVN-OM presenta un mejor desempeño, esto debido a su alta dependencia en los vóxeles al

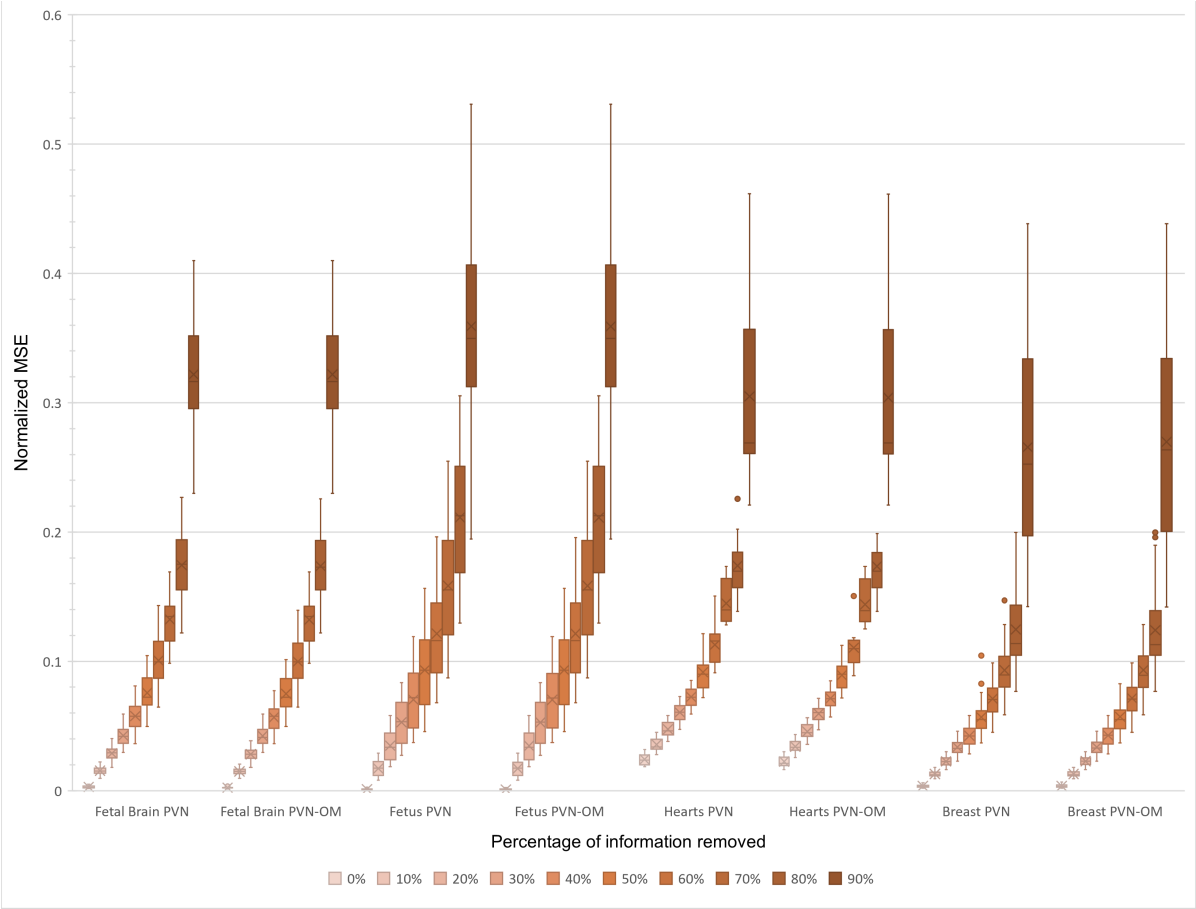


Figura 4.7: Comparación de los incrementos en el $\overline{\text{MSE}}$ como función de la información removida de los conjuntos de datos (representado por las diferentes tonalidades de color) cuando se producen imágenes 3D con los métodos PVN y PVN-OM. Para estas pruebas se utilizaron los conjuntos de datos: Cerebros Fetales, Fetos, Corazones Fetales y Mamas.

reconstruir la imagen 3D. Esto permite hacer uso total de la representación creada por el *octree* y sus claves Morton, resultado en una implementación más eficiente y directa al usar las capacidades de computa en paralelo. Los errores presentados durante la reconstrucción de imágenes 3D de ultrasonido se deben a diversos factores, como puede ser el método de interpolación utilizado para para calcular el valor final de cada vóxel o el tamaño del *kernel* usado para realizar una búsqueda de vecinos, pese a esto, nuestros resultados muestra que podemos reconstruir una imagen 3D con hasta un 60% de información removida, aunque este extremo lleva a la aparición de artefactos. Estos artefactos se presentan principalmente en las huecos de información más grandes y dependen del tamaño del *kernel* de búsqueda de vecindades.

Ambos métodos pueden presentar algunos bordes suavizados, pero en general la información de la imagen es preservada. Estos bordes suavizados se generan durante el proceso de reconstrucción y los consideramos como artefactos. Estos artefactos aparecen debido a la naturaleza de los algoritmos de reconstrucción. Debido a esto se presenta un error de reconstrucción mayor a cero cuando no hay información removida de las imágenes 3D base. Comparando ambos métodos, el VVN-OM produce una precisión de reconstrucción peor, incluso cuando nada de información es removida. Sin embargo, este algoritmo puede presentar imágenes visualmente más atractivas. Este efecto se produce debido a como el algoritmo VVN-OM funciona, toma el promedio de los valores en los vóxeles dentro de un *kernel* de búsqueda, para producir los valores de la imagen 3D final. Al realizar este promedio se están modificando los valores de la imagen 3D final como un promedio de los valores vecinos, lo cual nos lleva a diferencias entre la imagen base original y la imagen 3D reconstruida. Esto nos lleva a que un error más grande es encontrado al comparar ambas imágenes. En nuestros experimentos se muestra que el proceso de paralelización y el uso de *octree* junto con claves Morton no lleva a un error significativo en comparación con sus versiones paralelas sin uso de *octrees*.

Como se ha discutido previamente en esta tesis, una de las limitaciones de los *octrees* es que el espacio representado es dividido de manera equitativa entre todos los nodos hijo, esto sucede en las tres direcciones del espacio 3D. Si la distribución de punto no está balanceada, será necesario incrementar la profundidad del *octree* para obtener la resolución necesaria aunque este incremento será uniforme y no en un area localizada, incrementando la cantidad de tiempo de procesamiento necesario para obtener un reconstrucción final. Basado en los resultados obtenidos, el máximo tiempo de reconstrucción fue de 79.79 s utilizando el método PVN-OM produciendo una imágen de 512×512×512 vóxeles, con base en un conjunto de 131 imágenes 2D de ultrasonido. En el extremo opuesto, el mismo método requirió un tiempo de 0.44 s para obtener una imagen 3D de dimensiones 128×128×128 vóxeles basada en un conjunto de 51 imágenes con dimensiones 387×400 px, esto es equivalente a un velocidad de 2 cuadros por segundo (fps). Este tiempo se considera aceptable para procesos de planeación pre operativa y en la mayoría de los procesos quirugicos. Clinicamente hablando es un tiempo aceptable de reconstrucción cuando se encuentra en el orden segundos ya que esto no extiende de manera significativa la duración del estudio para estudios de imagenología 3D. El trabajo presentado por Hiep et al. [25] reporta que la adquisición de imágenes 2D de ultrasonido que toman de 3.5 ± 0.8 a 16.9 ± 6.9 minutos se consideran tiempos extra aceptables en procedimientos quirúrgicos. Los autores de [26] reportan

que en el caso de procedimientos intra-operatorios, 14 minutos es un tiempo promedio que se agrega debido a navegación 2D asistida, los cual no significa un impedimento para cirujanos y se considera insignificante.

Si bien el objetivo de esta implementación es mejorar la velocidad de la reconstrucción 3D de los métodos basados en búsqueda de vecindades, se realizó una comparación de desempeño con respecto al método basado en funciones de Bézier, realizando la misma reconstrucción. Nuestra implementación basada en Bézier produce imágenes más rápidamente que los métodos propuestos, pero acorde nuestras pruebas los errores producidos por el método basado en píxeles produce menos errores en la imagen 3D final. Durante nuestros experimentos se encontró que el método de Bézier es más sensible a la orientación de las imágenes 2D de ultrasonido, probando que nuestro métodos son más robustos con respecto a las orientaciones. Estas situaciones son relevantes especialmente en ambientes clínicos, ya que restringir la dirección de escaneo al adquirir imágenes 2D de ultrasonido podría no ser practico. Nuestra metodología es una buen alternativa y una opción competitiva.

Nuestros experimentos muestran que nuestras implementaciones aceleran la reconstrucción 3D de ultrasonido y pueden mejorar los tiempos para los métodos basados en píxeles y vóxeles para la búsqueda de vecindades. Como se mencionó anteriormente, las mejoras realizadas no afectan la calidad e la reconstrucción y pueden alcanzar velocidades en tiempo real. Estas implementaciones son viables alternativas a los métodos basados en funciones ya que presenta mayor robustez que puede significar un gran ventaja en entornos clínicos. El tiempo adicional requerido para obtener las imágenes 3D usando nuestras implementaciones no representa un incremento significativos para procedimiento quirúrgicos, durante los cuales los médicos expertos tienen tiempo limitado para examinar pacientes y es frecuentemente necesario repetir los escaneos para examinar lesiones y enfermedades. Se considera que algunas de las técnicas presentadas también pueden ser implementadas a métodos basados en funciones de Bézier, esto será explorado en trabajo futuro.

Capítulo 5

Visualización de simulación de fluidos

En el capítulo anterior hemos presentado la aplicación de nuestros métodos para manipular y almacenar información tri-dimensional en *octrees* de manera eficiente en conjunción con su implementación en paralelo aprovechando las capacidades de computación paralela que ofrecen los GPUs actuales. Sin embargo, el problema que se planteó originalmente consistía en desarrollar métodos para manipular y almacenar información tri-dimensional para lograr una simulación realista de líquidos, sobre todo para la simulación de sangre en ambientes gráficos. En este capítulo revisaremos a detalle nuestra propuesta para resolver dicho problema de visualización.

5.1 Antecedentes y estado del arte

Desde que las computadoras nos brindaron la posibilidad de visualizar información hemos buscado la manera de hacerlo de la manera más eficiente posible, esto con el propósito de ofrecer una mejor comprensión de la información obtenida por diversos medios. En la literatura existen diversos trabajos enfocados en proponer métodos para mejorar visualizaciones. En este punto merece la pena mencionar que en el campo de visualización se prefiere tener transiciones suaves entre las diferentes regiones de los objetos visualizados. En los comienzos de este campo de estudio, las limitaciones en recursos computacionales obligaron a proponer visualizaciones de objetos basadas en crear mallas de los objetos que se proyectaban sobre la pantalla para crear la imagen deseada; esta metodología se utilizaba buscando optimizar la velocidad de visualización

a costa de la calidad y la apariencia visual (por ejemplo, no tenían transiciones suaves entre regiones de los objetos). Estas aproximaciones obtenían visualizaciones burdas de los objetos y era necesario explorar otros métodos para mejorar sus visualizaciones, tales como el uso de texturas y mapas de normales así como otras técnicas que actuaban sobre la imagen final y no sobre la escena original. Por ello, se han propuesto diferentes técnicas con la finalidad de ofrecer opciones alternativas a la visualización basada en mallas; una de las más populares es la propuesta por Blinn [7] en 1982, la cual propone la representación de formas suaves utilizando una combinación lineal de funciones base con transiciones suaves (*meataballs*) que es visualizada en conjunto con una técnica de renderizado como *raytracing*. Sin embargo, uno de los principales problemas con esta metodología es el costo computacional que conllevaba, si bien *raytracing* es un método de renderizado que en sí, es demandante computacionalmente, al visualizar funciones de densidad creadas a partir de la combinación lineal de *metaballs* su costo computacional se ve incrementado drásticamente.

Desde entonces, el uso de *metaballs* ha sido estudiado, si bien, no para el uso de visualizaciones en tiempo real, sí para la representación de objetos orgánicos como la visualización de fluidos [27], gases [28] y modelado de objetos con transiciones suaves [29]. La mejora del *hardware* y la aparición de *hardware* nuevo, tal como GPUs, ha permitido expandir la capacidad computacional y explorar nuevas metodologías que no eran posibles previamente, tales como las técnicas de *raytracing* en combinación con el uso de *metaballs*. La utilización de eficiente de datos a través de estructuras de datos optimizadas han permitido que sea posible realizar visualizaciones en tiempo real de objetos modelados con *metaballs* y renderizados por medio de *raytracing*. A continuación se exploran algunos de los trabajos más recientes y relevantes que hacen uso de las técnicas ya mencionas en combinación con algunas otras propuestas por diversos autores presentados a continuación.

Una de las estructuras de datos más usadas para la visualización de escenas en $\mathbb{R}^3$ son los *octrees* [4] porque permite subdividir el espacio de manera fácil y eficiente, lo que permite mejorar el rendimiento de los procesos de visualización. El trabajo propuesto por Cyril Crassin [30] demuestra la capacidad de esta estructura de datos en el proceso de renderizado usando *raytracing* de vóxeles que varian con el tiempo utilizando codificaciones en los nodos del *octree*; este método les permitió almacenar escenas con datos masivos de vóxeles y texturas, obteniendo resultados de 20 fps para un volumen de $1024 \times 1024 \times 1024$ vóxeles, utilizando una tarjeta gráfica Nvidia® 8800 GTS 512. Sin embargo, todas las escenas presentadas son estáticas y no están

optimizadas para ser modificadas durante la ejecución.

Aquí vale la pena hacer notar que el estándar para películas y video transmitido es de 24 fps, 30 fps para transmisión de eventos y programas de televisión y 60 fps para video con resolución de 3840 $\times$ 2160 píxeles (conocido como 4K UHDTV); por lo que es deseable producir secuencias de imágenes con tasas entre 30 y 60 fps.

Por otra parte, el trabajo presentado por Flynn et al [31] presenta una implementación de *octrees* para la visualizar la interacción de fluidos, alcanzando tiempos de renderizado de 4.69 fps para un volumen de 64 $\times$ 64 $\times$ 64 vóxeles; si bien esta implementación utiliza únicamente un *octree* como base de la visualización los resultados obtenidos no se pueden considerar como tiempo real por lo que no existe una interacción fluida con la escena.

Los trabajos mencionados arriba utilizan directamente un *octree* como base para la visualización sin proponer una codificación especifica. De forma contraria, el trabajo presentado por Laine and Karras [32] propone la codificación de un *octree* para su rápido acceso. Dicha codificación permite tener descriptores de los nodos directamente en claves asignadas a los mismos, lo cual acelera el proceso de acceso de los datos para su visualización al contener información descriptiva de los nodos que ayuda a decidir si es necesario acceder a la información dentro de un nodo. También permite cargar partes del *octree* en memoria, ya sea en CPU o GPU, alternando según sea necesario. Con este método los autores obtuvieron resultados de alta resolución, permitiendo una profundidad máxima de 13 niveles en el *octree* con una velocidad de 77.1 millones de rayos por segundo utilizando *raycasting*. Las escenas en niveles más bajos permiten completa interacción y demuestra la mejora que proporciona la implementación de una codificación eficiente en *octrees*.

Los trabajos de Pätzold y Kolb [33] y Baert et al [34] presentan implementaciones de *octrees* codificadas con claves Morton para la aceleración de la búsqueda espacial, sin embargo, ambas implementaciones están enfocadas al renderizado de mallas y la estructura de datos es usada para la búsqueda de triángulos dentro de mallas. El trabajo presentado por Songo et al. [35] también presenta un método para renderizar imágenes 3D provenientes de estudios médicos en el cual utilizan un *octree* codificado con claves Morton; este método busca optimizar el almacenamiento de información y no tanto en la velocidad de la visualización, demostrando la capacidad de un *octree* para la representación de grandes cantidades de información. El trabajo reporta 37% de radio de compresión para volúmenes de 512 $\times$ 512 $\times$ 400 vóxeles. Por otra parte, los trabajos mencionados presentan métodos que implementan *octree* para la visualización de in-

formación estática o que carecen de una interacción en tiempo real, presentan escenas donde no se visualizan objetos orgánicos o suavizados y, a excepción del trabajo Flynn et al, ninguno muestra la interacción entre objetos en alguna forma. El uso de *metaballs* para representar objetos orgánicos se ha presentado en diversos trabajos, la tesis de doctorado presentada por Jiménez Parga [36] explora la representación de nubes de puntos usando *metaballs* como función de suavizado, sus resultados permiten la contrucción de diferentes tipos de nubes de puntos con diferentes propiedades demostrando la versatilidad de las *metaballs*. Los autores de Pan et al [37] proponen un método de visualización que utiliza *metaballs* para la representación de órganos deformables, con énfasis en el hígado, que tiene la capacidad de realizar cortes sobre el órgano visualizado.

Uno de los campos donde las *metaballs* son ampliamente usadas es la representación de fluidos. El trabajo de Mercier et al. [38] presenta el uso de *metaballs* para la representación de turbulencia en fluidos, logrando preservar los detalles del comportamiento del agua pero obteniendo 1.5 segundos por imagen generada para un total de 390 mil puntos (partículas).

El trabajo presentado por Xiao et al [39] presenta una implementación de *metaballs* para la simulación de líquidos en tiempo real, esta implementación presenta los mejores tiempos para el uso de *metaballs* junto con *raycasting*, alcanzando un tiempo de 25 fps para imágenes con una resolución de 1920×1920 píxeles con más de 770 mil puntos. Otra implementación presentada en el mismo contexto es la realizada por Santos Brito et al [40], en la cual se presenta un algoritmo de *raycasting* para la visualización de grandes cantidades de puntos. Los autores utilizan *metaballs* para la representación de líquidos, pero, calculan iso-superficies utilizando el gradiente de las *metaballs* en combinación con los campos de intensidad. Su implementación da resultados de 5 fps para 750 mil puntos simuladas, pero es importante mencionar que las visualizaciones generadas con su método incluyen efectos de reflexión y refracción.

5.2 Implementación

5.2.1 Intersección de rayos con *metaballs*

Una forma de visualizar un conjunto, o nube, de puntos $\mathcal{C}$ es por medio de renderizar la superficie que representa su contorno. Por ello, lo primero que se necesita hacer es encontrar la superficie, algo que se obtiene encontrando una iso-superficie de la combinación lineal de (2.1) en donde las *metaballs* se encuentran centradas sobre los

puntos $\{\boldsymbol{c}\}$ de $\mathcal{C}$, $c_j = 1$, para $1 \leq j \leq |\mathcal{C}|$ y $N = |\mathcal{C}|$. La forma más trivial de extraer la iso-superficie es por medio de subdividir el soporte V del conjunto $\mathcal{C}$ en vóxeles de forma tal que un punto $\boldsymbol{c} \in \mathcal{C}$ se encuentra en el vecindario de Voronoi $\mathcal{V}_{\boldsymbol{v}}$ de al menos un punto $\boldsymbol{v} \in V$ (el vóxel representado por el punto $\boldsymbol{v}$). Entonces, para generar la isosuperficie de (2.2) para el conjunto $\mathcal{C}$ se avanza a través de cada vóxel $\boldsymbol{v}$ en $\mathcal{V}$ hasta encontrar un punto $\boldsymbol{x}$ en el cual el valor de f es mayor, o igual, a un valor predeterminado (umbral τ). Esta metodología tiene desventajas cuando se usa *raytracing* para realizar la visualización (el uso de una malla no ofrece visualizaciones suaves, además de que es complicado simular el movimiento de puntos), pero la principal es que es necesario establecer la distancia Δ a la cual un rayo $\boldsymbol{r}$, ver (2.7) y (3.8), avanzará dentro de un vóxel $\boldsymbol{v}$ para encontrar el punto $f(\boldsymbol{x}) \geq \tau$. El valor de Δ determinará la precisión con la que la iso-superficie será encontrada: si el valor de Δ seleccionado es grande, se obtendrán intersecciones escalonadas alrededor de la iso-superficie; por otro lado, si el valor es muy pequeño la intersección será precisa pero conllevará un alto costo computacional, ver la Figura 5.1.

Figura 5.1: Visualización de la combinación lineal de un par de *metaballs*. usando (*izquierda*) con un valor grande para Δ, es posible visualizar la iso-supeficie escalonada formada por los intervalos grandes de Δ, y (*derecha*) con un valor pequeño para Δ.

El costo computacional elevado viene del proceso realizado en cada punto probado sobre $\boldsymbol{r}$, debido a que para cada punto es necesario calcular la función (2.1) utilizando todas las *metaballs* cercanas hasta encontrar un punto de intensidad suficientemente alto. Si el $\boldsymbol{r}$ no encuentra el valor apropiado de la función (2.1) dentro de un nodo, repetirá el proceso en el siguiente nodo y así sucesivamente hasta encontrar un punto con la intensidad suficiente o salir del espacio del *octree*, Figura 5.2.

Diferentes opciones alternativas han sido exploradas; como el uso de funciones por

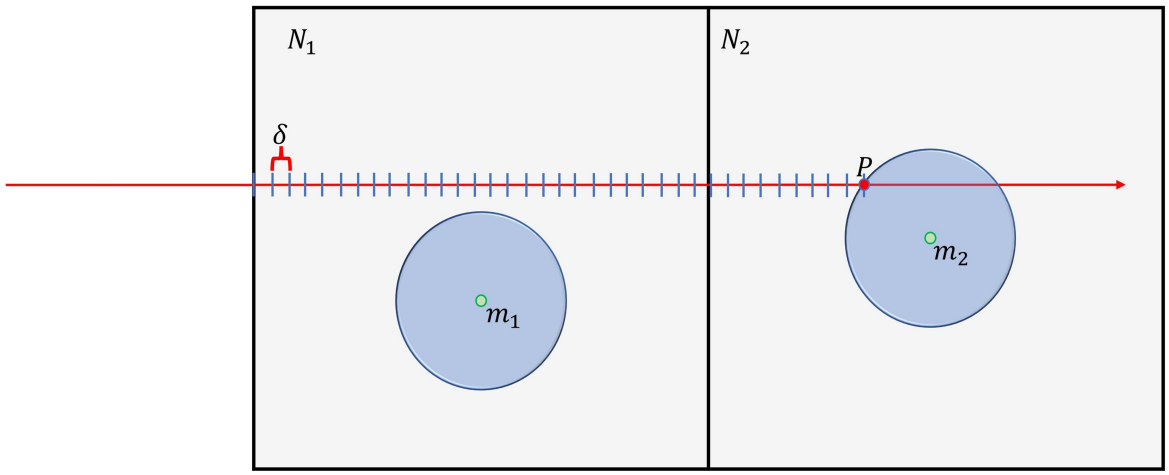


Figura 5.2: El punto $\boldsymbol{p}$ mostrado en la figura representa el punto en el cual el rayo $\boldsymbol{r}$ proyectado usando un incremento δ choca con la iso-superficie generada por las *metaballs* contenidas en el nodo.

partes [15] el cual propone el uso de puntos de control (3 puntos por *metaball*) para calcular una función que aproxime la curva generada por la iso-superficie, esta metodología se vuelve costosa computacionalmente para una cantidad elevada de *metaballs*. Otra propuesta consiste en utilizar *Bézier Clipping* [41] el cual consiste en utilizar puntos de control distribuidos a través de la intersección para encontrar la curva de Bézier que mejor describa el campo de intensidad generado por las *metaballs*. Esta metodología es bastante rápida e implica pocas iteraciones para encontrar un punto de intersección apropiado, sin embargo implica resolver múltiples veces un polinomio de grado 6 por lo que si el punto de intersección se encuentra lejos del punto inicial de búsqueda puede tomar una cantidad considerable de iteraciones encontrar una convergencia. En la implementación realizada en esta tesis se decidió optar por una metodología más simple pero que, sin embargo, nos permite encontrar el punto de intersección realizando menos cálculos.

Para cada nodo $\boldsymbol{v}$ a procesar es necesario delimitar el área de búsqueda con las *metaballs* que van a tener influencia sobre la trayectoria del rayo $\boldsymbol{r}$. De acuerdo con la definición de la función base (3.7), la combinación lineal (2.1) será igual a 0 cuando $r > a$ por lo tanto todas las *metaballs* que se encuentren en una distancia mayor al

radio a no tendrán influencia y se pueden excluir del procesamiento. Para lograr esto es posible realizar el calculo de la intersección con una esfera cuyo radio es igual a a y a la cual llamaremos *esfera exterior*; este método nos ayuda a descartar los rayos que no entran en contacto con la cmbinación lineal de *metaballs* y nos permite construir un esqueleto sobre la iso-superficie final, ver Figura 5.3.

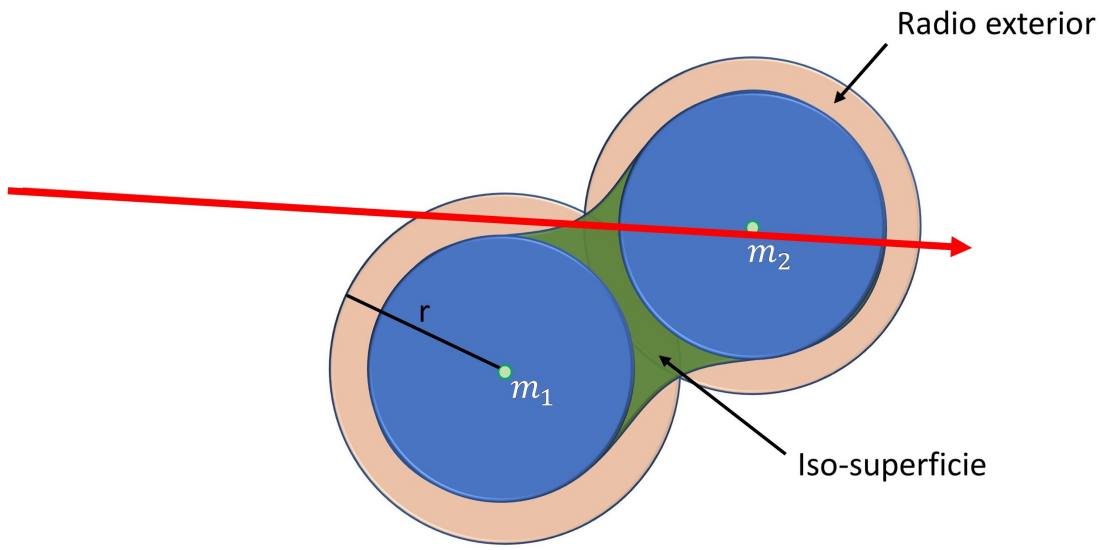


Figura 5.3: Se crean esferas exteriores (naranja) que crean un esqueleto que encierra la iso-superficie generada por las *metaballs*. Si un rayo choca con alguna de las esferas exteriores es candidato para impactar con la iso-superficie.

Los rayos que sigan teniendo relevancia son aquellos que tienen una mayor probabilidad de intersecar con la iso-superficie, para obtener el punto de intersección final se obtienen 2 casos distintos: (i) El rayo interseca con una esfera exterior y (ii) el rayo choca con más de una esfera exterior. Para el primer caso, es suficiente con buscar la distancia d_m a la cual la función (2.1) es igual al umbral τ, ver la Figura 5.3. El valor de d_m puede ser encontrado usando un *clipping* de Bezier básico o avanzando a lo largo de la trayectoria del rayo. Una ventaja de la intersección con una única *metaball* es que el valor d_m puede ser precalculado antes de iniciar el proceso de *raytracing* dado

que este valor será constante para todas las *metaballs*. Para el segundo caso de que existan múltiples intersecciones con múltiples esferas exteriores, es necesario encontrar la primera intersección con la iso-superficie contenida, ver la Figura 5.4.

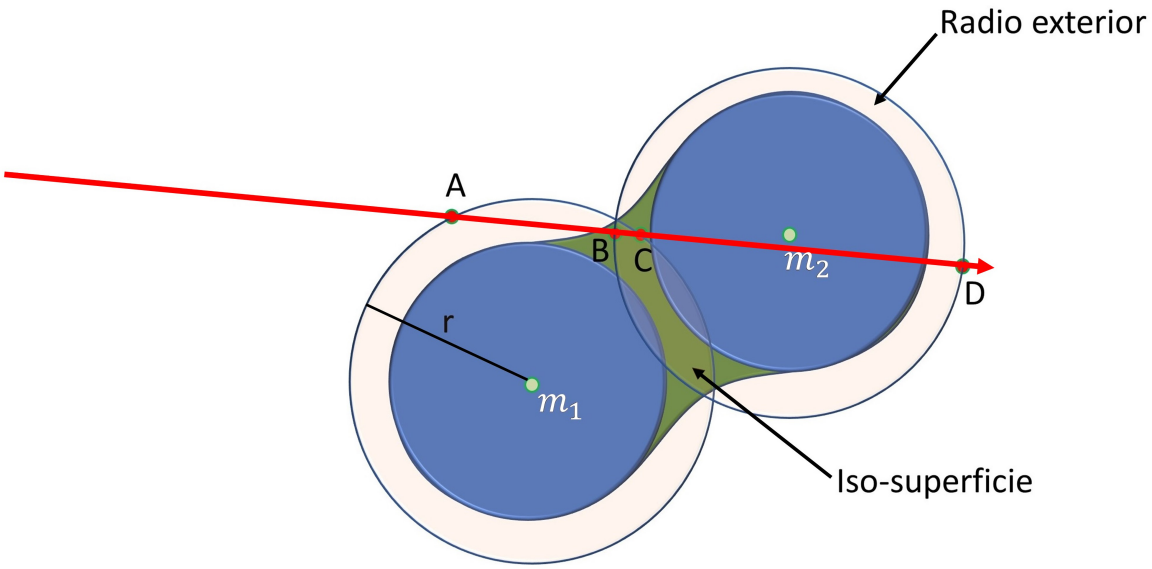


Figura 5.4: Un rayo que interseca con rayo exteriores puede tener más de una colisión, la figura muestra 4 colisiones (A, B, C, D) por lo que es necesario ordenar las distancias y empezar la búsqueda de la iso-superficie a partir del punto de intersección más cercano.

Para lograr esta tarea se decidió realizar un esquema de avance a lo largo de un rayo y buscar un punto en el cual el umbral τ es cruzado (en otras palabras, pasa de $f(\boldsymbol{r}) < \tau$ a $f(\boldsymbol{r}) > \tau$). Para evitar un sobre muestreo sobre el rayo durante la búsqueda, utilizamos la metodología propuesta por J.M. Singh and P.J. Narayanan [42] llamada *marching points*, en la cual se establecen múltiples umbrales que determinarán el muestreo realizado a lo largo de un rayo. Está técnica busca establecer umbrales que, conforme sean alcanzados, cambiarán el tamaño del incremento al que será muestreado el rayo $\boldsymbol{r}$. Si el umbral inicial no es superado el rayo será muestreado con los incrementos de mayor tamaño, debido a que significa que no estamos cerca de ningún campo de intensidad producido por alguna *metaballs*, conforme nos acercamos a algún compo de intensidad el umbral será superado y procederemos a incrementar el muestreo de $\boldsymbol{r}$, si nos acercamos aún más a algún campo de intensidad un segundo umbral será superado y el muestreo será incrementado de nuevo. Usando esta metodología, los rayos que no entren en el umbral final serán muestreados rápidamente y descartados, ver la Figura

5.5.

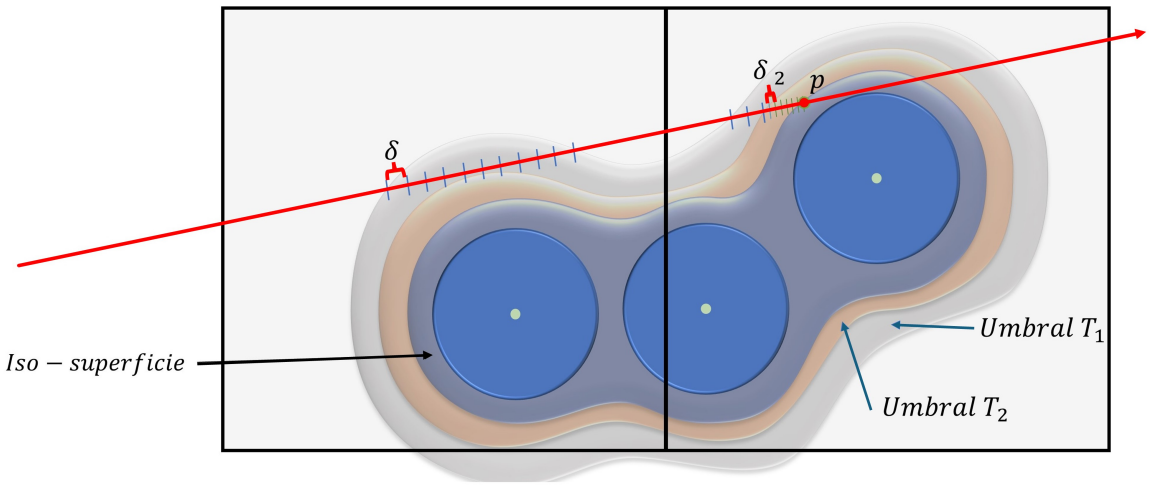


Figura 5.5: Figura que muestra la interacción del rayo con diferentes umbrales. Si el umbral es más grande el muestreo será más pequeño para encontrar un punto de intersección más preciso con la iso-superficie.

Utilizando estos métodos es posible encontrar las intersecciones de un rayo $\boldsymbol{r}$ con las *metaballs* involucradas de una manera rápida y precisa, pero, sobre todo, descartando las *metaballs* que no intervienen en la formación de la superficie deseada.

5.3 Búsqueda de vecindades y nodos fantasma

Al procesar los nodos de manera aislada se presentan artefactos no deseados debido al espacio delimitado por los nodos. Los rayos que impactan con el espacio asociado a un nodo con información son los únicos que son considerados para el procesamiento y el renderizado (aquellos que no intersecan serán, potencialmente, tomados en cuenta para otros nodos), ver Figura 5.6a. En el caso en que el centro de una *metaball* se encuentra proximo a alguna de las caras del vóxel asociado a un nodo, entonces sólo una parte de la distribución de la *metaball* será considerada. Algo similar sucede en el caso de tener dos *metaballs* lo suficientemente cerca como para que sus distribuciones se sobrepongan, pero si se encuentran en vóxeles asociados a nodos distintos no se realizará un calculo

entre ellas, ver Figura 5.6b.

Si un nodo $\boldsymbol{o}$, con vóxel asociado $\mathcal{V}_{\boldsymbol{o}}$, contiene *metaballs* en su espacio y un rayo $\boldsymbol{r}$ lo interseca, es necesario considerar todas las *metaballs* que potencialmente contribuyen a la suma de *metaballs* sobre el rayo $\boldsymbol{r}$ y por ello es necesario buscar el conjunto de nodos $\{\boldsymbol{u}\}$ que se encuentran en la vecindad de $\boldsymbol{o}$; en vista de que los nodos tienen un vecindario de Voronoi cúbico, cada nodo tiene 26 vecinos (caras, aristas y vértices) que deben ser revisados para determinar si las *metaballs* que se encuentren en su espacio tienen alguna intersección con el rayo $\boldsymbol{r}$. Este proceso puede ser computacionalmente costoso por lo que se consideran algunas optimizaciones.

Para buscar a los 26 vecinos a un nodo $\boldsymbol{o}$ se puede utilizar claves Morton, utilizando operaciones sobre los bits que componen una clave; como se vio en la Sección 2.2 una clave Morton $\mathcal{M}$ está representada por 30 bits organizados en 10 triplas de bits, en dónde el valor de la j-ésima tripla $\mathcal{T}_j$, para $1 \leq j \leq 10$, representa el número de descendencia en el nivel, o profundidad, j del *octree*.

Con este método sólo es posible encontrar los nodos cara-adyacentes a $\boldsymbol{o}$: aquellos que se encuentran a su derecha o izquierda (eje $\vec{\boldsymbol{x}}$), arriba o abajo (eje $\vec{\boldsymbol{y}}$) o enfrente o detrás (eje $\vec{\boldsymbol{z}}$). Para hallar a los restantes 20 nodos vecinos es necesario, primero, pasar a uno de los nodos cara-adyacente a $\boldsymbol{o}$ para luego llegar a uno de los 12 nodos arista-adyacente; para los restantes 8 nodos vértice-adyacente será necesario primero encontrar un nodo arista-adyacente para luego llegar desde ahí a uno de los nodos vértice-adyacente. Este proceso es descrito a detalle en el trabajo de Valdez et al. [43], donde se presenta una comprobación matemática del método que demuestra su validez para cualquier $\mathcal{M}$ asociada a un nodo dentro del *octree*. El método para hallar cada nodo cara-adyacente se considera de la siguiente manera.

1. **Nodo cara-adyacente $\vec{\boldsymbol{x}}$.** Para hallar a los vecinos en esta dirección se aplica una operación de conjunción lógica, o AND, entre la clave Morton del nodo $\boldsymbol{o}$ y una máscara $\boldsymbol{\mathcal{L}}$ (que contiene los bits 001 en la tripla $\mathcal{T}_j$ y el resto de triplas igual a 000): $\mathcal{T}_j \wedge \boldsymbol{\mathcal{L}}$. En caso de que el resultado de dicha operación sea igual a 0×00, entonces el nodo se encuentra a la izquierda en el espacio definido por su nodo superior (nodo padre); en caso de que el resultado de la operación lógica sea igual a 001 para la j-ésima la tripla, entonces el nodo se encuentra a la derecha del espacio definido por su nodo superior. Tomando en cuenta que el nodo $\boldsymbol{o}^{(j)}$ se encuentra en el nivel j (tiene un nodo superior a profundidad $j-1$) este nodo $\boldsymbol{o}^{(j-1)}$ hereda cuatro nodos que comparten la cara izquierda de su vóxel asociado

$\mathcal{V}_{\boldsymbol{o}^{(j-1)}}$ y otros cuatro que comparten la cara derecha de $\mathcal{V}_{\boldsymbol{o}^{(j-1)}}$.

(a) Cuando el vóxel asociado con el nodo $\boldsymbol{o}^{(j)}$ comparte la cara izquierda de $\mathcal{V}_{\boldsymbol{o}^{(j-1)}}$, entonces siempre existe un nodo cara-adyacente $\boldsymbol{q}$ a la derecha de $\boldsymbol{o}^{(j)}$ que es también hijo del de nodo $\boldsymbol{o}^{(j-1)}$. Para hallar la clave Morton $\mathcal{M}_{\boldsymbol{q}}$ se utiliza la operación de disyunción exclusiva, o XOR, entre $\mathcal{T}_j$ y la máscara $\boldsymbol{\mathcal{L}}$ ($\mathcal{T}_j \oplus \boldsymbol{\mathcal{L}}$).

(b) En el caso de que el vóxel asociado al nodo $\boldsymbol{o}^{(j)}$ comparta la cara derecha de $\mathcal{V}_{\boldsymbol{o}^{(j-1)}}$, sí existe un nodo cara-adyacente $\boldsymbol{q}$ a la derecha de $\boldsymbol{o}^{(j)}$ y se encuentra en la cara izquierda del vóxel $\mathcal{V}_{\boldsymbol{p}^{(j-1)}}$ (el vóxel asociado al nodo cara-adyacente a la derecha de $\mathcal{V}_{\boldsymbol{o}^{(j-1)}}$). Para encontrar la clave Morton $\mathcal{M}_{\boldsymbol{q}}$, primero, se utiliza la operación $\mathcal{T}_j \oplus \boldsymbol{\mathcal{L}}$. Segundo, usando la clave Morton resultante $\mathcal{M}_{\boldsymbol{q}}$, se iniciará de nuevo el proceso para calcular el vecino derecho, con la diferencia de que se usarán $j-1$ triplas para realizar el proceso; lo cual significa que estaremos procesando el nodo $\boldsymbol{o}^{(j-1)}$ que tiene la clave Morton $\mathcal{M}_{\boldsymbol{q}}$ como punto de partida y buscando su nodo cara-adyacente derecho, en el caso que este vecino también se encuentre del lado izquierdo de su nodo superior, el proceso se repetirá ahora buscando el vecino a la derecha de $\boldsymbol{o}^{(j-2)}$. Este proceso será repetido de, ser necesario, hasta llegar a cuando $j = 1$, ya que esto significa que $j-1$ será igual a cero y esto correspondería al nodo raíz o_r, por lo que no es posible repetir el proceso ya que el nodo o_r no tiene vecinos, y podemos decir que el nodo $\boldsymbol{q}$ buscado no existe ya que nos encontramos en el límite del espacio del *octree*. Si el vecino derecho es encontrado en algún nivel antes de llegar a $j = 1$, la clave Morton encontrada en este vecino automáticamente corresponderá a la clave Morton de $\boldsymbol{q}$. Debido a que todos los bits han sido conservados y únicamente las triplas involucradas fueron modificadas.

2. **Nodo cara-adyacente $\vec{\boldsymbol{y}}$**. Para hallar a los vecinos en esta dirección se aplica la operación $\mathcal{T}_j \wedge \boldsymbol{\mathcal{L}}$, en donde la máscara $\boldsymbol{\mathcal{L}}$ contiene los bits 010 en la tripla $\mathcal{T}_j$ y el resto de triplas igual a 000. En caso de que el resultado de dicha operación sea igual a 0×00, entonces el nodo se encuentra a la izquierda en el espacio definido por su nodo superior (nodo padre); en caso de que el resultado de la operación lógica sea igual a 0×02 en la tripla $\mathcal{T}_j$, entonces el nodo se encuentra a la derecha del espacio definido por su nodo superior. Tomando en cuenta que el nodo $\boldsymbol{o}^{(j)}$

se encuentra en el nivel j, tiene un nodo superior a profundidad $j-1$, este nodo $\boldsymbol{o}^{(j-1)}$ hereda cuatro nodos que comparten la cara izquierda de su vóxel asociado $\mathcal{V}_{\boldsymbol{o}^{(j-1)}}$ y otros cuatro que comparten la cara derecha de $\mathcal{V}_{\boldsymbol{o}^{(j-1)}}$.

(a) Cuando el vóxel asociado con el nodo $\boldsymbol{o}^{(j)}$ comparte la cara izquierda de $\mathcal{V}_{\boldsymbol{o}^{(j-1)}}$, entonces siempre existe un nodo cara-adyacente $\boldsymbol{q}$ a la derecha de $\boldsymbol{o}^{(j)}$ que es también hijo del de nodo $\boldsymbol{o}^{j-1}$. Para hallar la clave Morton $\mathcal{M}_{\boldsymbol{q}}$ se vuelve utilizar la operación $\mathcal{T}_j \oplus \mathcal{L}$.

(b) En el caso de que el vóxel asociado al nodo $\boldsymbol{o}^{(j)}$ comparta la cara derecha de $\mathcal{V}_{\boldsymbol{o}^{(j-1)}}$, sí existe un nodo cara-adyacente $\boldsymbol{q}$ a la derecha de $\boldsymbol{o}^{(j)}$, este se encuentra en la cara izquierda del vóxel $\mathcal{V}_{\boldsymbol{p}^{(j-1)}}$, que es vóxel asociado al nodo cara-adyacente a la derecha de $\mathcal{V}_{\boldsymbol{o}^{(j-1)}}$. Para encontrar la clave Morton $\mathcal{M}_{\boldsymbol{q}}$, primero, se realiza la operación $\mathcal{T}_j \oplus \mathcal{L}$. Segundo, usando la clave Morton resultante $\mathcal{M}_{\boldsymbol{q}}$, se iniciará de nuevo el proceso para calcular el vecino derecho, con la diferencia de que usarán $j-1$ triplas al realizar el proceso, lo cual significa que estaremos procesando el nodo $\boldsymbol{o}^{(j-1)}$ que tiene la clave Morton $\mathcal{M}_{\boldsymbol{q}}$ como punto de partida y buscando su nodo cara-adyacente derecho, en el caso que este vecino también se encuentre del lado izquierdo de su nodo superior, el proceso se repetirá ahora buscando el vecino a la derecha de $\boldsymbol{o}^{(j-2)}$. Este proceso será repetido de, ser necesario, hasta llegar un límite de $j=1$, ya que esto significa que $j-1$ será igual a cero y esto correspondería al nodo raíz o_r, por lo que no es posible repetir el proceso ya que el nodo o_r no tiene vecinos, y podemos decir que el nodo $\boldsymbol{q}$ buscado no existe ya que nos encontramos en el límite del espacio del *octree*. Si el vecino derecho es encontrado en algún nivel antes de llegar a $j=1$, la clave Morton encontrada en este vecino automáticamente corresponderá a la clave Morton de $\boldsymbol{q}$. Debido a que todos los bits han sido conservados y únicamente las triplas involucradas fueron modificadas.

3. **Nodo cara-adyacente $\vec{z}$**. Para hallar a los vecinos en esta dirección se aplica una operación $\mathcal{T}_j \wedge \mathcal{L}$, en donde la máscara $\mathcal{L}$ contiene los bits 100 en la tripla $\mathcal{T}_j$ y el resto de triplas igual a 000. En caso de que el resultado de dicha operación sea igual a 0×00, entonces el nodo se encuentra a la izquierda en el espacio definido por su nodo superior (nodo padre); en caso de que el resultado de la operación lógica sea igual a 0×04 en la tripla $\mathcal{T}_j$, entonces el nodo se encuentra a la derecha

del espacio definido por su nodo superior. Tomando en cuenta que el nodo $\boldsymbol{o}^{(j)}$ se encuentra en el nivel j, tiene un nodo superior a profundidad $j-1$, este nodo $\boldsymbol{o}^{(j-1)}$ hereda cuatro nodos que comparten la cara izquierda de su vóxel asociado $\mathcal{V}_{\boldsymbol{o}^{(j-1)}}$ y otros cuatro que comparten la cara derecha de $\mathcal{V}_{\boldsymbol{o}^{(j-1)}}$.

(a) Cuando el vóxel asociado con el nodo $\boldsymbol{o}^{(j)}$ comparte la cara izquierda de $\mathcal{V}_{\boldsymbol{o}^{(j-1)}}$, entonces siempre existe un nodo cara-adyacente $\boldsymbol{q}$ a la derecha de $\boldsymbol{o}^{(j)}$ que es también hijo del de nodo $\boldsymbol{o}^{(j-1)}$. Nuevamente, para hallar la clave Morton $\mathcal{M}_{\boldsymbol{q}}$ se utiliza la operación $\mathcal{T}_j \oplus \boldsymbol{\mathcal{L}}$.

(b) En el caso de que el vóxel asociado al nodo $\boldsymbol{o}^{(j)}$ comparta la cara derecha de $\mathcal{V}_{\boldsymbol{o}^{(j-1)}}$, sí existe un nodo cara-adyacente $\boldsymbol{q}$ a la derecha de $\boldsymbol{o}^{(j)}$, este se encuentra en la cara izquierda del vóxel $\mathcal{V}_{\boldsymbol{p}^{(j-1)}}$, que es el vóxel asociado al nodo cara-adyacente a la derecha de $\mathcal{V}_{\boldsymbol{o}^{(j-1)}}$. Nuevamente, se encuentra la clave Morton $\mathcal{M}_{\boldsymbol{q}}$, primero, utilizando $\mathcal{T}_j \oplus \boldsymbol{\mathcal{L}}$. Después, se usa la clave Morton resultante $\mathcal{M}_{\boldsymbol{q}}$ y se iniciará de nuevo el proceso para calcular el vecino derecho, con la diferencia de que usarán $j-1$ triplas al realizar el proceso. Esto significa que se estará procesando el nodo $\boldsymbol{o}^{(j-1)}$ que tiene la clave Morton $\mathcal{M}_{\boldsymbol{q}}$ como punto de partida y se busca su nodo cara-adyacente derecho, en el caso que este vecino también se encuentre del lado izquierdo de su nodo superior, el proceso se repetirá ahora buscando el vecino a la derecha de $\boldsymbol{o}^{(j-2)}$. Este proceso será repetido de, ser necesario, hasta llegar al límite de $j=1$, ya que esto significa que $j-1$ será igual a cero y esto correspondería al nodo raíz o_r, por lo que no es posible repetir el proceso ya que el nodo o_r no tiene vecinos, y podemos decir que el nodo $\boldsymbol{q}$ buscado no existe ya que nos encontramos en el límite del espacio del *octree*. Si el vecino derecho es encontrado en algún nivel antes de llegar al $j=1$, la clave Morton encontrada en este vecino automáticamente corresponderá a la clave Morton de $\boldsymbol{q}$. Debido a que todos los bits han sido conservados y únicamente las triplas invoucradas fueron modificadas.

Pese a que únicamente es posible buscar las claves Morton de los nodos cara-adyacente a $\boldsymbol{o}$, es posible usar este método de manera iterativa hasta encontrar los nodos cara-adyacente, arista-adyacente, vértice-adyacente.

Al momento de procesar las *metaballs* contenidas en los nodos vecinos una optimización realizada es comparar la distancia de la *metaballs* al nodo $\boldsymbol{o}$, si esta distancia

es superior al radio a significa que estas *metaballs* se encuentran alejadas del nodo principal y por lo tanto no es necesario que sean procesadas. Una optimización adicional consiste en considerar la cara por la cual el rayo entra al nodo $\boldsymbol{o}$, si dicha cara corresponde es la que se busca y coincide a la dirección por la cual el rayo $\boldsymbol{r}$ entra al nodo, nos indica que el rayo $\boldsymbol{r}$ ya pasó por el nodo conectado por esta cara y por lo tanto no es necesario que sea revisado de nuevo. Finalmente, es posible descartar los nodos que se encuentren conectados únicamente a las aristas de la cara que son intersecadas por el rayo $\boldsymbol{r}$ y por lo tanto podemos eliminar nueve nodos de la búsqueda de vecinos, ver la Figura 5.7.

5.4 Calculo de normales

Una vez encontrado el punto en el cual un rayo interseca con la iso-superficie generada por las *metaballs*, es necesario obtener la normal para cada punto alcanzado sobre la iso-superficie. La normal de una superficie describe los cambios en cada punto definido sobre la superficie, esto con respecto a los ejes $\vec{\boldsymbol{x}}$, $\vec{\boldsymbol{y}}$, $\vec{\boldsymbol{z}}$. Debido a que la iso-superficies de las *metaballs* es descrita como la sumatoria de todas las funciones base, es posible calcular los cambios de dicha función, esto es calculado utilizando el gradiente ∇f. En la implementación presentada se utilizó la función de (3.7) la cual depende del radio r. Siguiendo esta idea es posible calcular el cambio presentado por (3.7) en el punto de intersección $\boldsymbol{p}$ con respecto a $\vec{\boldsymbol{x}}$, $\vec{\boldsymbol{y}}$, $\vec{\boldsymbol{z}}$.

El gradiente de una función expresa el cambio de la función en cada dirección y es útil para obtener la normal en cada punto de la función. Para calcular el gradiente de (3.7) se expresa en términos de sus componentes en las direcciones $\vec{\boldsymbol{x}}$, $\vec{\boldsymbol{y}}$ y $\vec{\boldsymbol{z}}$. Considerando que usamos la *metaball* de (3.7) en la combinación lineal de (2.1) para aproximar una función, entonces el radio desde un punto $\boldsymbol{p}$ a la j-ésima *metaball* se calcula con

$$r\left(\boldsymbol{p}, \boldsymbol{q}_j\right) = \sqrt{(p_x - q_{j,x})^2 + (p_y - q_{j,y})^2 + (p_z - q_{j,z})^2}, \tag{5.1}$$

donde $(q_{j,x}, q_{j,y}, q_{j,z})$ representan las coordenadas del centro $\boldsymbol{q}_j$ de la j-ésima *metaball*, como se menciona en la definición de (2.1). Sustituyendo (5.1) en (3.7) tenemos el

gradiente como

$$\nabla b\left(\boldsymbol{p};\boldsymbol{q}_j\right) = \left[1 - (\frac{\sqrt{\left(p_x - q_{j,x}\right)^2 + \left(p_y - q_{j,y}\right)^2 + \left(p_z - q_{j,z}\right)^2}}{r})\right]^2 .$$

De esta manera es posible calcular el gradiente de la combinación lineal de las *metaballs* de (3.7) con respecto a un punto $\boldsymbol{p}$. Para calcular el gradiente de una función es necesario calcular las derivadas parciales con respecto a cada eje coordenado de la siguiente forma

$$\nabla f\left(\boldsymbol{p};\boldsymbol{q}_j\right) = \left(\frac{\partial f}{\partial x}\left(\boldsymbol{p};\boldsymbol{q}_j\right), \frac{\partial f}{\partial y}\left(\boldsymbol{p};\boldsymbol{q}_j\right), \frac{\partial f}{\partial z}\left(\boldsymbol{p};\boldsymbol{q}_j\right)\right).$$

Calculando las derivadas parciales de la ecuación la ecuación de Murakami definida en (3.7), obtenemos

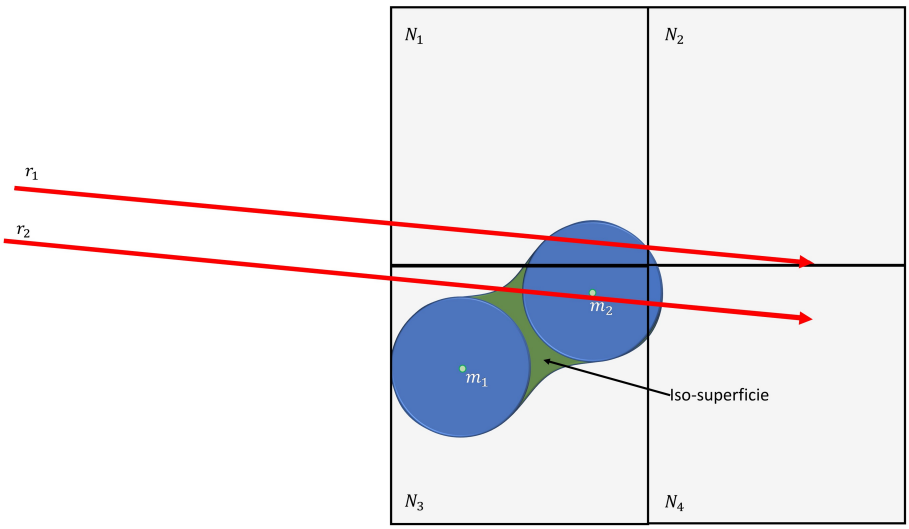
$$\begin{aligned} b\left(\boldsymbol{p};\boldsymbol{q}_j\right) = (&-\frac{4}{r^2}\left(\left(p_x - q_{j,x}\right)^3 + \left(p_x - q_{j,x}\right) + \left(p_x - q_{j,x}\right)\left(p_y - q_{j,y}\right)^2 + \left(p_x - q_{j,x}\right)\left(p_z - q_{j,z}\right)^2\right), \\ &-\frac{4}{r^2}\left(\left(p_y - q_{j,y}\right)^3 + \left(p_y - q_{j,y}\right) + \left(p_y - q_{j,y}\right)\left(p_x - q_{j,x}\right)^2 + \left(p_y - q_{j,y}\right)\left(p_z - q_{j,z}\right)^2\right), \\ &-\frac{4}{r^2}\left(\left(p_z - q_{j,z}\right)^3 + \left(p_z - q_{j,z}\right) + \left(p_z - q_{j,z}\right)\left(p_x - q_{j,x}\right)^2 + \left(p_z - q_{j,z}\right)\left(p_y - q_{j,y}\right)^2\right), \end{aligned} \tag{5.2}$$

donde r representa la distancia a la *metaball* de (3.7). Usando (5.2) se calcula la normal para cada rayo que tenga una intersección con la iso-superficie generada por las *metaballs*. Por lo que para obtener la normal de la combinación lineal de las *metaballs* definidas por (3.7) en el punto $\boldsymbol{p}$ se calcula

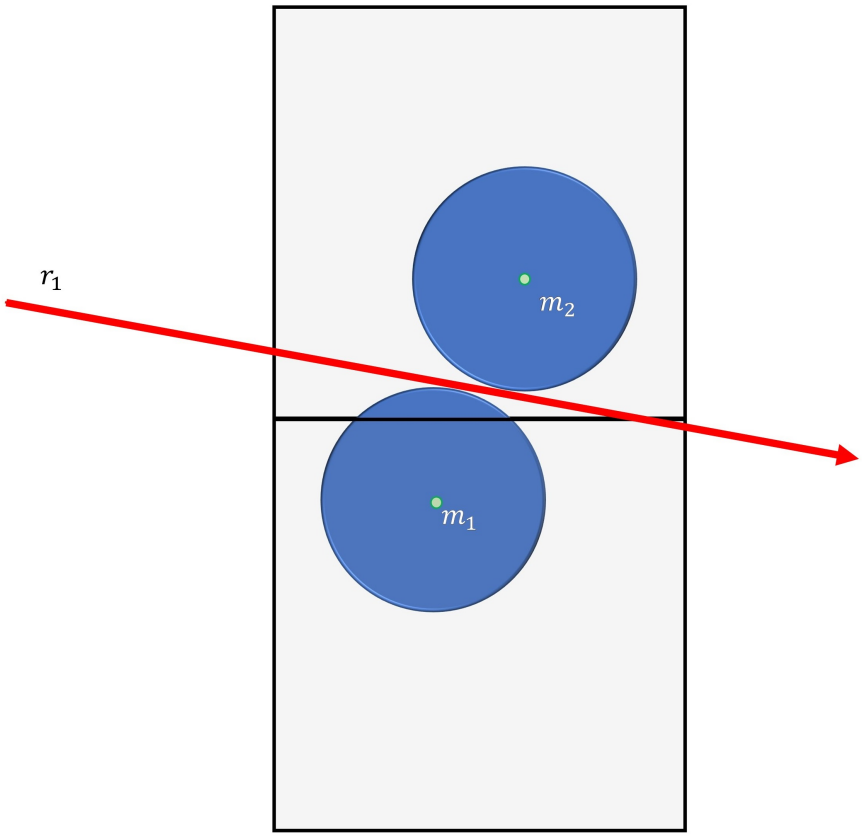
$$\begin{aligned} \vec{\boldsymbol{n}}\left(\boldsymbol{p}\right) = \sum_{j=1}^{m} \Bigg(&-\frac{4}{r^2}\left(\left(p_x - q_{j,x}\right)^3 + \left(p_x - q_{j,x}\right) + \left(p_x - q_{j,x}\right)\left(\left(p_y - q_{j,y}\right)^2 + \left(p_x - q_{j,x}\right)\left(\left(p_z - q_{j,z}\right)\right)^2\right)\right), \\ &-\frac{4}{r^2}\left(\left(p_y - q_{j,y}\right)^3 + \left(p_y - q_{j,y}\right) + \left(p_y - q_{j,y}\right)\left(p_x - q_{j,x}\right)^2 + \left(p_y - q_{j,y}\right)\left(p_z - q_{j,z}\right)^2\right), \\ &-\frac{4}{r^2}\left(\left(\left(p_z - q_{j,z}\right)^3 + \left(p_z - q_{j,z}\right) + \left(p_z - q_{j,z}\right)\left(p_x - q_{j,x}\right)^2 + \left(p_z - q_{j,z}\right)\left(p_y - q_{j,y}\right)^2\right)\right)\Bigg), \end{aligned} \tag{5.3}$$

donde m representa el número de *metaballs* usadas para calcular la normal correspondiente en el punto $\boldsymbol{p}$. La Figura 5.8a muestra una superficie donde las normales son calculadas usando únicamente el promedio de las normales emitidas por cada *metaballs*

y la Figura 5.8b muestra la misma superficie con las normales calculadas usando (5.3).



(a) Pese a que el rayo $\boldsymbol{r}_1$ debe intersecar la iso-superficie generada por las dos *metaballs*, ambas *metaballs* no son consideradas debido a que los centros de las *metaballs* no se encuentran en los vóxeles asociados a los nodos N_1 y N_2, que se encuentran en la trayectoria del rayo y ambos vóxeles se consideran espacio vacío.



(b) Las *metaballs* m_1 y m_2 se encuentran en nodos distintos por lo que el rayo $\boldsymbol{r}_1$ solo considerara una *metaballs* a la vez mientras viaja por cada uno de los nodos.

Figura 5.6: Figuras que muestran 2 situaciones en la que el espacio delimitado por el nodo provoca que los rayos no intersequen con la iso-superficie.

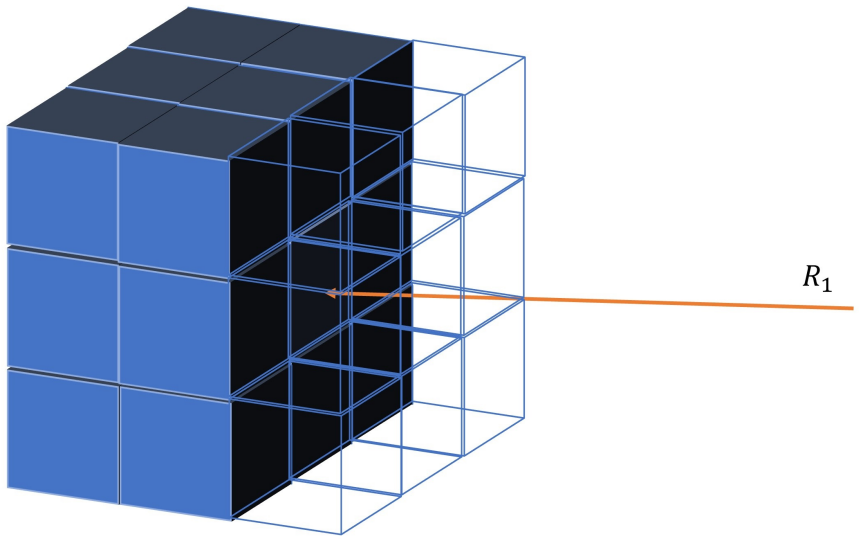
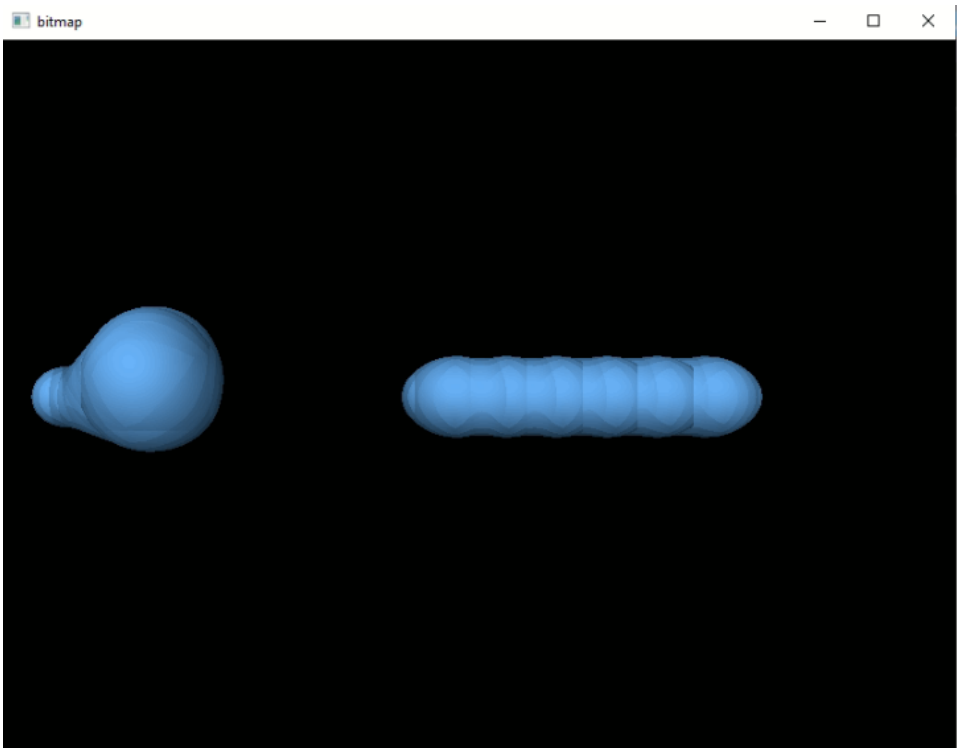
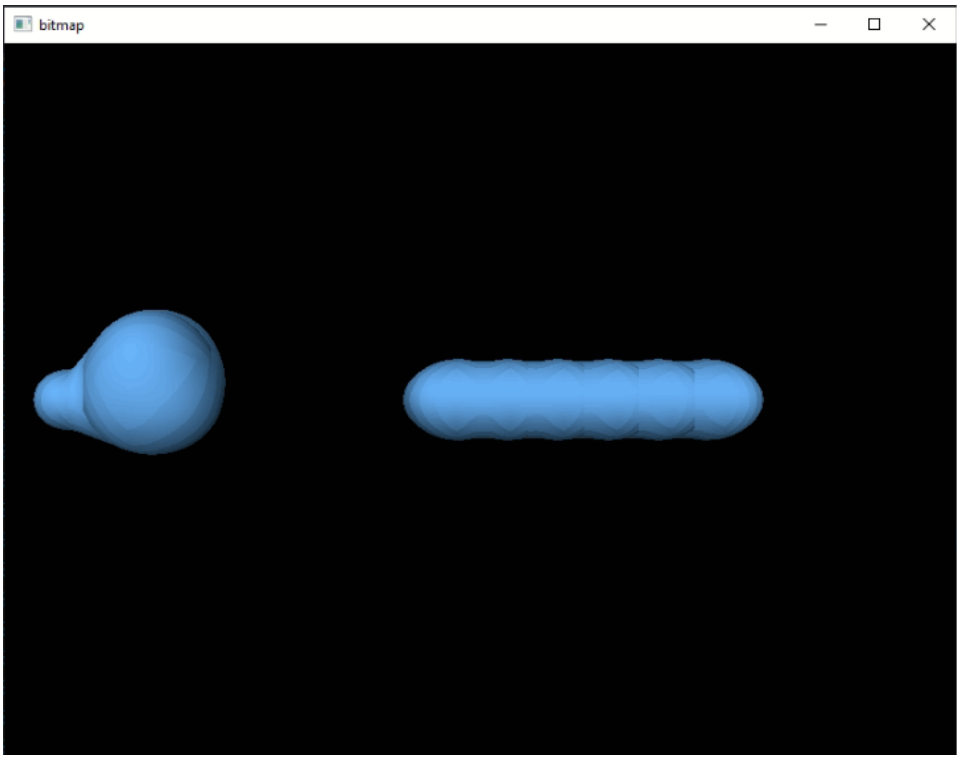


Figura 5.7: Figura que muestra los nueve nodos descartados como parte de la vecindad. Los nodos descartados ya fueron consultados debido a la trayectoria del rayo $\boldsymbol{r}_1$.



(a) Superficie calcula con normales promediadas utilizando los centros de la *meta-balls*.



(b) Superficie calcula con normales usando el gradiente de la función de campo.

Capítulo 6

Resultados y conclusiones

Para la evaluación de las técnicas implementadas se realizaron pruebas de tiempo de renderiza así como memoria consumida por la implementación del *octree*. Dado que nuestra implementación funciona completamente en GPU la memoria es una aspecto importante a considerar. Para la simulación de interacción físicas se utilizó una implementación de SPH para partículas libres de mallas, la cual funciona también en GPU. Los tiempos reportados no toman en cuenta el tiempo utilizado para simulación física ni tampoco tiempos de lectura de datos.

6.0.1 Archivo de datos

Antes de la ejecución del programa es necesario obtener la información de las partículas, esto es realizado utilizando un archivo de texto en el cual se almacenan las diferentes propiedades de las partículas: posición, color, factor de reflexión y factor de refracción. Cada partícula y sus propiedades ocupan un renglón dentro del archivo de texto para y sus propiedad son separadas por un espacio en blanco. El primer renglón del archivo indicará el numero de partículas que serán utilizados, los renglones posteriores contendrán un total de 9 elementos: 3 elementos que componen la posición en el espacio de la partícula, 3 elementos que describen el color (RGB), 1 elemento para el factor de reflexión, 1 para el factor de refracción y 1 elemento usado para la simulación que indica el factor de rigidez de la partícula.

Ejemplo:

1432

0.450000 0.400000 0.450000 0.4 0.6 0.1 0.6 0.2 1

0.450000 0.400000 0.469812 0.4 0.6 0.1 0.6 0.2 1

0.450000 0.400000 0.489625 0.4 0.6 0.1 0.6 0.2 1
0.450000 0.400000 0.509438 0.4 0.6 0.1 0.6 0.2 1

Cuando el archivo de texto es cargado en memoria se procesan las posiciones de las partículas buscando las posiciones más cercanas al origen en las 3 dimensiones (x, y, z) y la más lejana para de esta manera determinar las dimensiones que totales que tendrá el *octree*.

6.0.2 Resultados de renderizado

Para todas las pruebas realizadas se utilizó una tarjeta de video Nvidia 1080 con 6 Gb de memoria de video, un procesador intel Core i7 6700 y 16 Gb de memoria RAM. Se realizaron diferentes pruebas para determinar la calidad de la visualización así como los tiempos obtenidos.

La Figura 6.1 muestra la visualización de 100 *metaballs* interactuando. Esta imagen fue capturada realizando un acercamiento a una sección pequeña del *octree*, la interacción de las *metaballs* sucede en 9 nodos únicamente. Es posible observar que algunas discontinuidades se forman sobre la superficie, esto se debe a la distancia que se utiliza para dividir el rayo para encontrar el punto de intersección mas cercano. Es posible observar una *metaball* aislada de las demás del lado derecho y como su forma generada es una esfera, esto se debe a la función de campo utilizada y al hecho de no tener interacción con ninguna otra *metaball*.

Para poder observar de una mejor manera la interacción entre las *metaballs* se hicieron pruebas con partículas distribuidas por el espacio del *octree*, la Figura 6.2 muestra 100 mil *metaballs* interactuando, debido a características especificas de la simulación la aglomeración de *metaball* en ciertos puntos no permite que su distribución de manera uniforme, esto lleva a tener espacios vacíos.

Con las técnicas presentadas es posible la representación de sólidos, para realizar esto necesario aumentar el campo de intensidad requerido cuando se encuentren más de una intersección con una *metaball*, esto lleva a una superficie más uniforme si los centros de las *metaballs* se encuentran apropiadamente distribuidas. En la Figura 6.3 se muestra una sección de un cubo, se observan algunas secciones sobresalientes.

El sistema de renderizado presentado presenta opciones versátiles de configuración por lo que es posible la representación de diversas formas y figuras. Si se desea obtener diversos comportamientos es posible modificar la función de intensidad usada así como

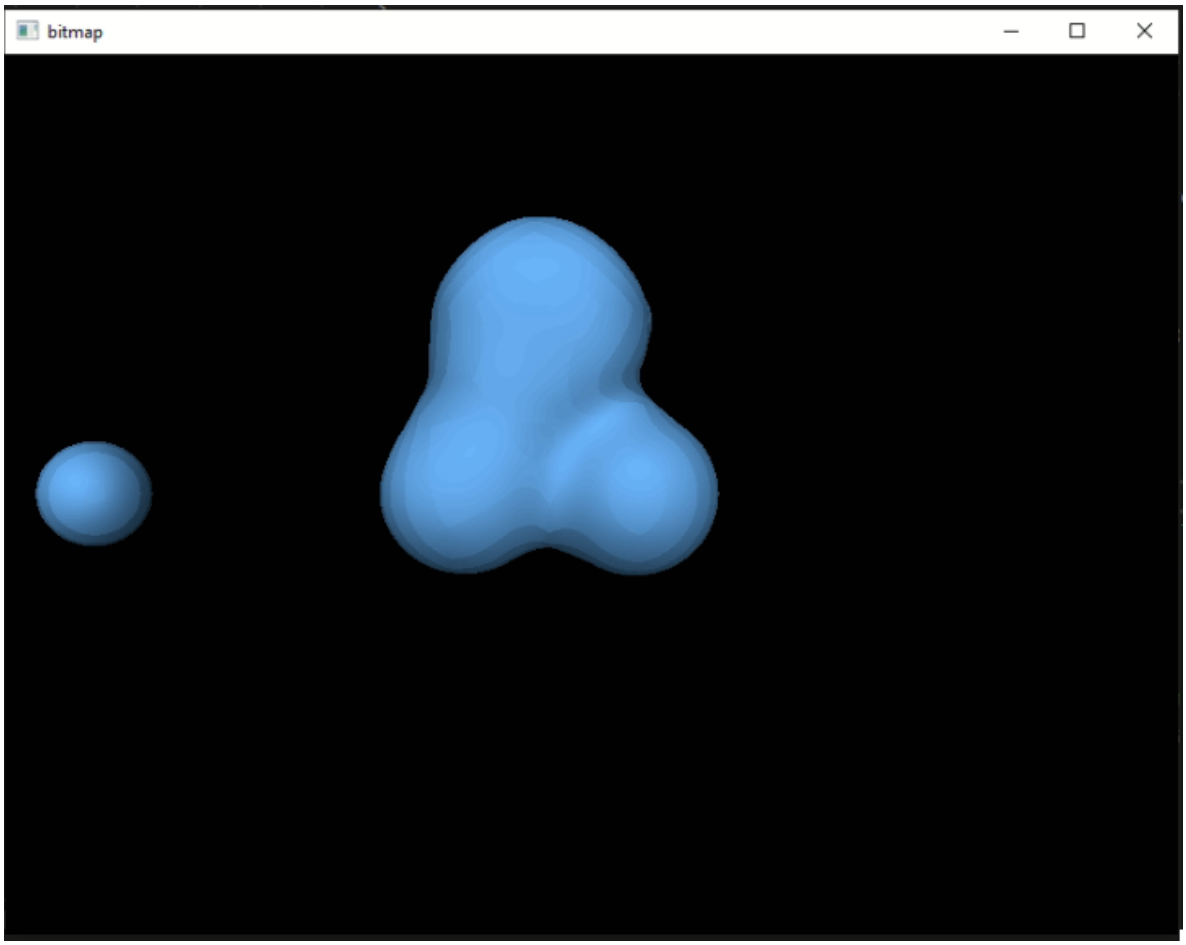


Figura 6.1: Figura que muestra 100 *metaballs* interactuando y formando una iso-superficie completa.

ajustar el los valores de intensidad usados para construir la iso-superficie. Las figuras 6.4 y 6.5 muestran 2 ejemplos más de las visualizaciones realizadas.

Debido a que el sistema de renderizado se basa en la visualización a partir de puntos en el espacio que representen los centros de las *metaballs*, es posible adaptar un mallado a este formato. Los mallados usan como primitiva de representación triángulos, estos están compuestos de vértices que pueden funcionar como centros de *metaballs*. Siguiendo esta idea es posible interpretar un mallado y generar una iso-superficie a partir de los vértices de cada trinagulo. El color asignado a cada vértice será el color emitido por cada *metaball*, para obtener una distribución de color suave, se promedian los colores de todas las metaballs que generar un campo de en cada punto sobre la superficie, este color es ponderado por la cantidad de influencia que esté aportando cada *metaball*, aquellas que se encuentre a una distancia mayor aportarán menos color. La figura 6.6

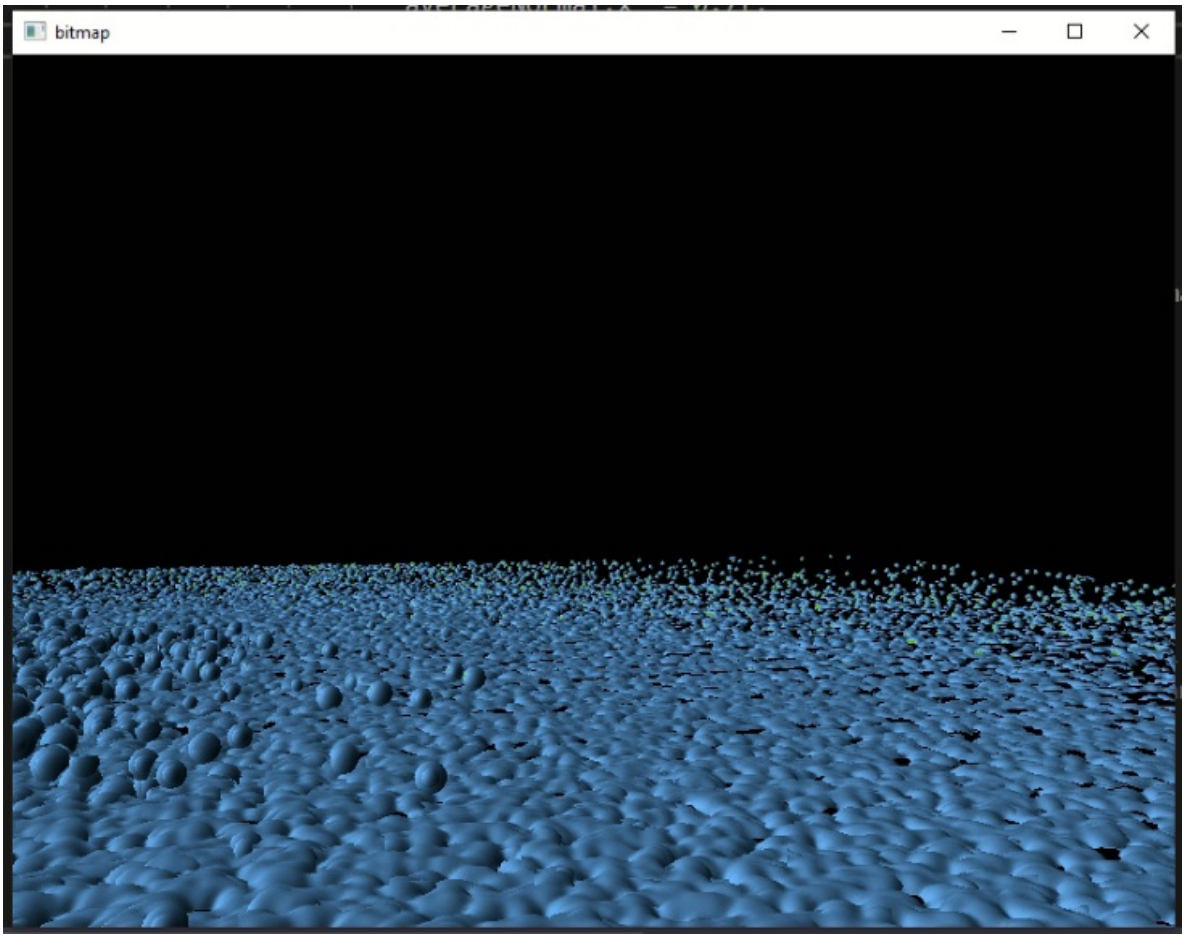


Figura 6.2: Imagen que muestra 100 mil *metaballs* distribuidas de manera uniforme.

muestra un ejemplo de esta visualización, se utilizó un mallado de una cráneo humano con una textura de color. Es posible observar pequeño montículos que se generan por la naturaleza esférica de las *metaballs* pero estos no afectan a los detalles importantes del mallado ni agregan artefactos que puedan afectar la interpretación de la visualización.

6.0.3 Tiempos de visualización

Para medir los tiempos que toma el sistema en generar una imágenes se utilizaron 5 archivos de datos con 100, 1342, 21822, 145777 y 320120 partículas respectivamente, cada partículas representará el centro de una *metaball*. Todos los resultados fueron probados con un total de 800,000 rayos proyectados. La Tabla 6.1 muestra los cuadros por segundo (fps por sus siglas en ingles) obtenidos al renderizar una escena completa y la Tabla 6.2 muestra la memoria usada para almacenar el *octree* a diferentes profun-

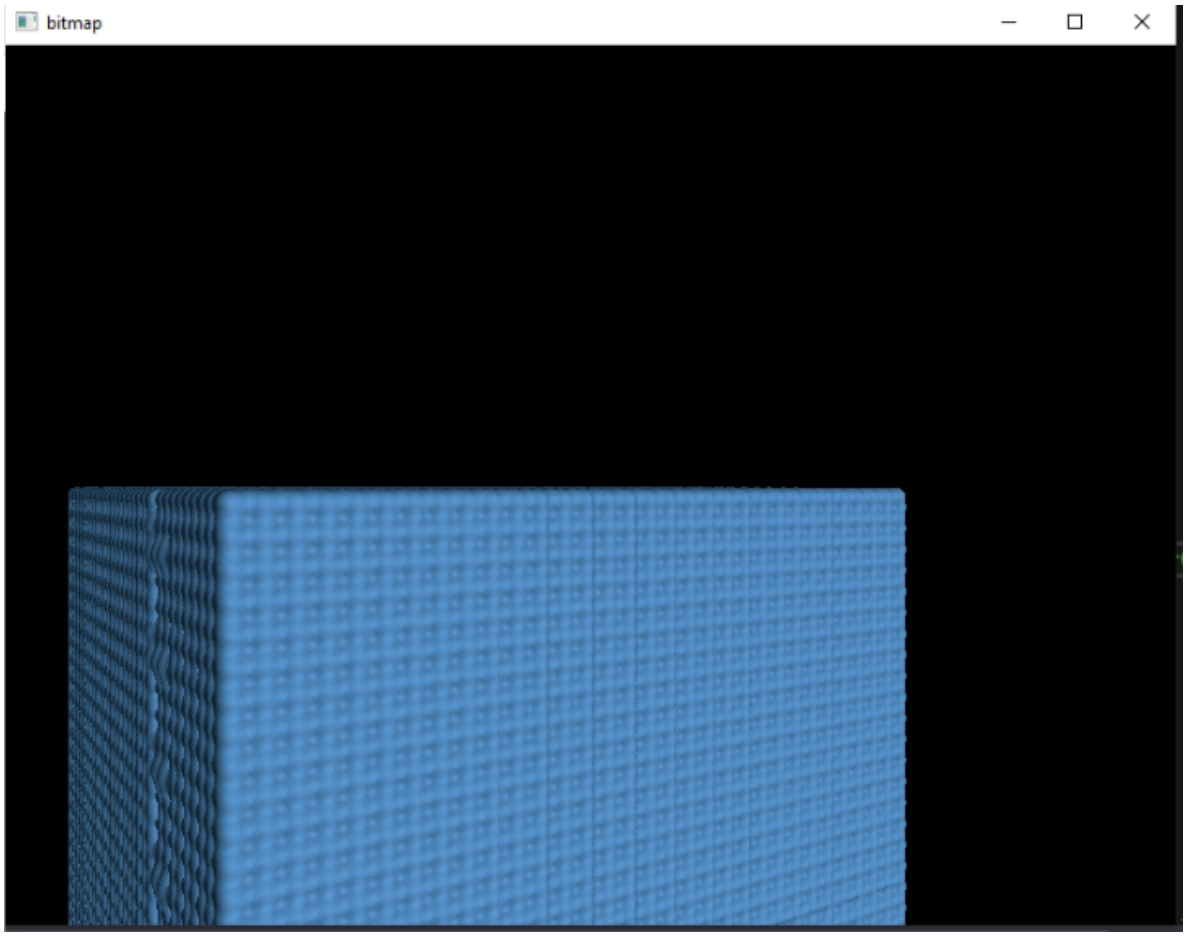


Figura 6.3: *Metaballs* distribuidas simulando una superficie solida, es posible definir bordes afilados, sin embargo, se requiere una cantidad elevada de *metaballs*.

didades máximas.

Los resultados obtenidos muestran un desempeño interactivo incluso para más de 300 mil *metaballs* interactuando con un *octree* de resolución 8 ($256 \times 256 \times 256$) para segmentar el espacio. El trabajo reportado por Szécsi et al. [44] alcanza 6 fps al representar 200 mil *metaballs*. Es importante mencionar que los tiempos obtenidos dependen directamente de la distribución de las partículas dentro del *octree*. Si las partículas están distribuidas de manera separada sin que los campos de intensidad se mezclen o existan menos interacciones, los rayos navegaran a través de una cantidad mayor de nodos buscando información contenida por lo que los tiempos aumentaran considerablemente.

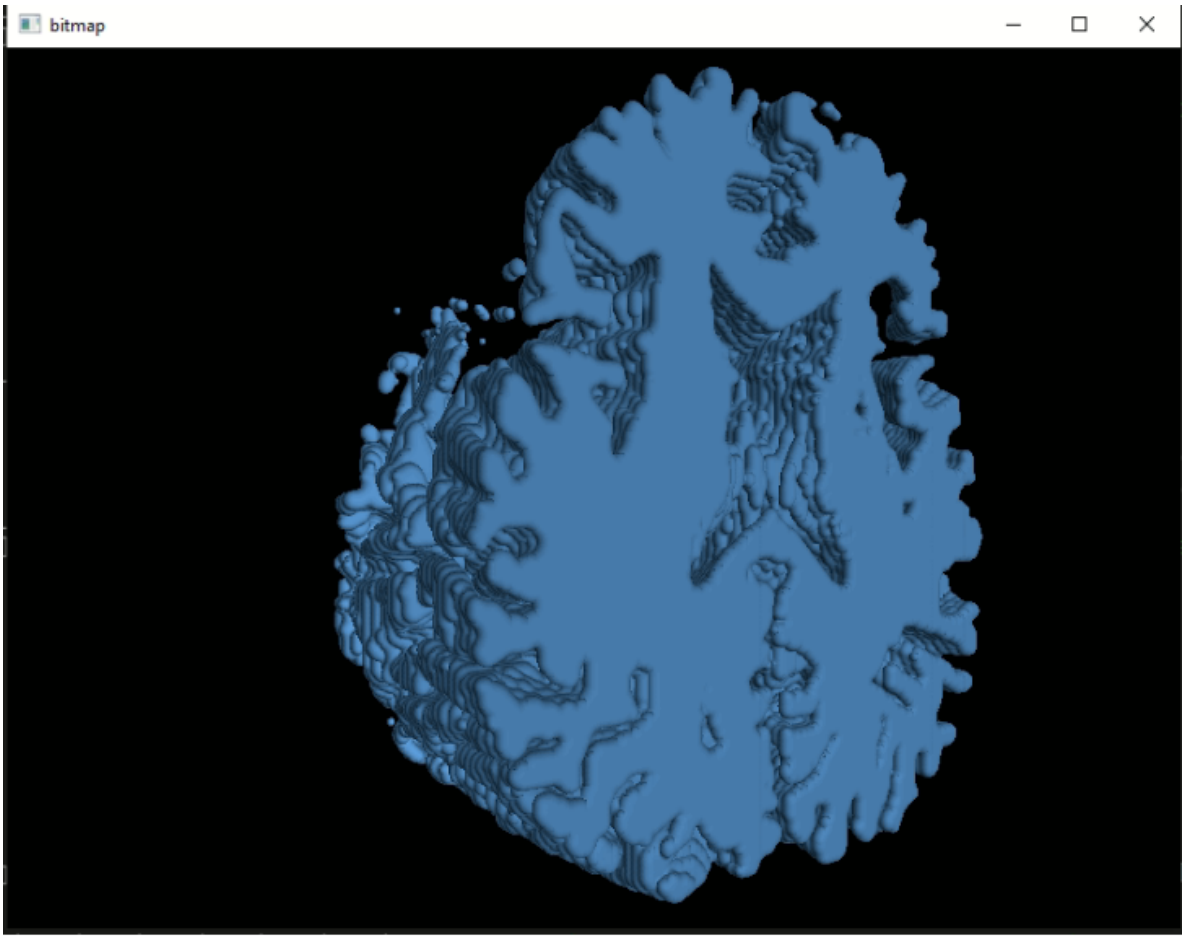


Figura 6.4: Visualización del corte realizado a un modelo del cerebro. El modelo del cerebro es obtenido usando imágenes de tomografía donde cada píxel con información de las imágenes de tomografía representa el centro de una *metaball*.

6.0.4 Artefactos e imprecisiones

Como se ha mencionado anteriormente, artefactos e imprecisiones pueden llegar a presentarse en la visualización. Una de ellas es una falta de uniformidad en la superficie cuando se realiza un acercamiento, esto se debe a muestreo realizado al encontrar la intersección con la iso-superficie generada por las *metaballs*. Es posible corregir este error generando un submuestreo más pequeño o usando otro metodología como el bezier clipping, sin embargo, usar estas opciones impactarían significativamente en el tiempo de renderizado y los artefactos generados no son perceptibles al momento de visualizar la escena completa.

Una impresión generada se encuentra al momento de transformar las coordenadas de las partículas a coordenadas del *octree*, este proceso implica una normalización que

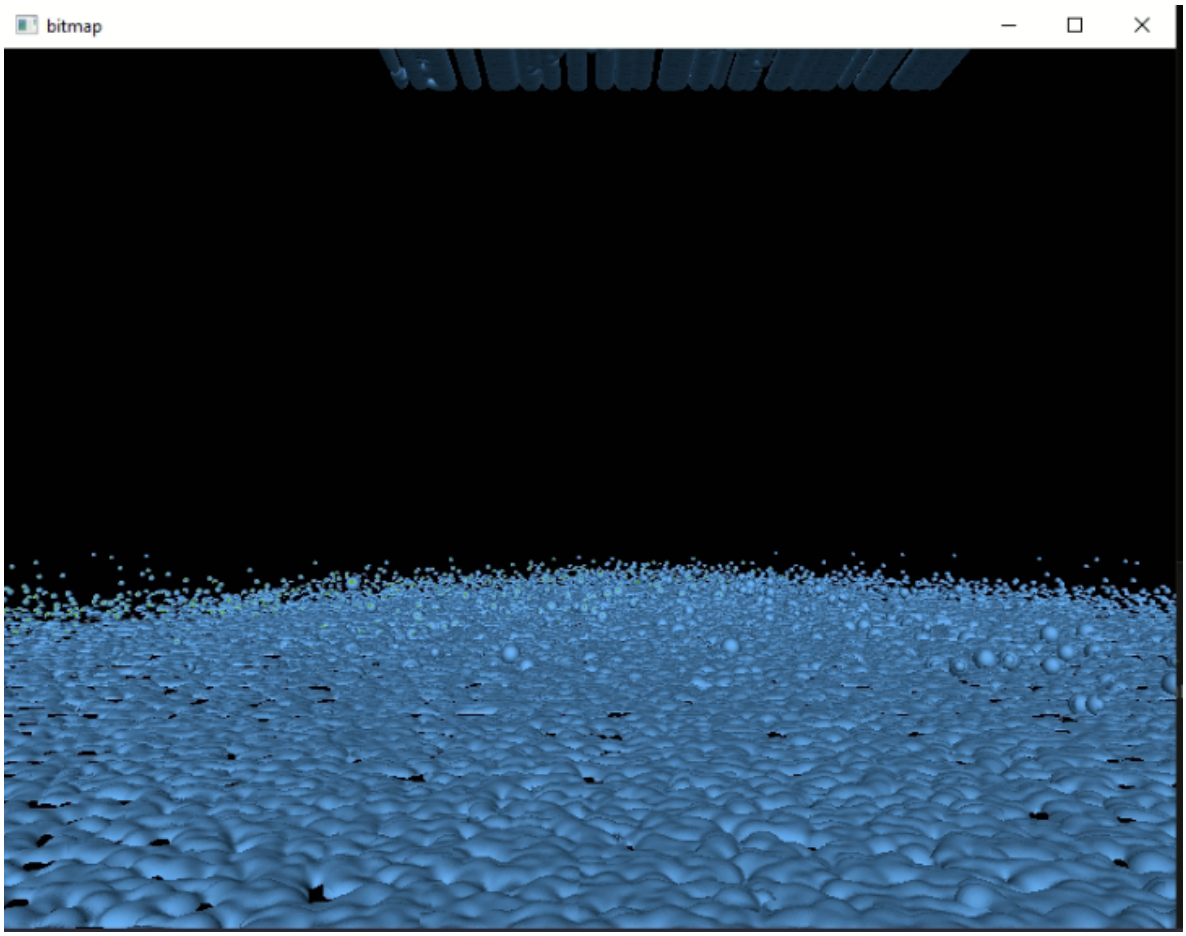


Figura 6.5: *Metaballs* interactuando de manera aleatoria sobre el limite del octree.

va de 0 a 1024 por lo que la posición de las partículas puede contener una imprecisión la posición que ocupan dentro del espacio del *octree*. El margen de error generado con la normalización incrementara conforme la dimensión del *octree* aumentan debido a que se normalizan una cantidad mayor de valores en cada dirección del *octree*. Es posible solucionar esto utilizando claves Morton más grandes (64 bits) pero esto implica un uso de memoria más grande para el almacenamiento de tanto partículas como nodos.

6.1 Conclusiones

En la tesis presentada se realizó la implementación de una nueva metodología de organización de datos, modificando la manera en que las estructuras de datos son almacenadas en memoria y llenadas con información, así como la manera en la que los datos y su estructura contenedora se relacionan. Se implemento una metodología de codificación que

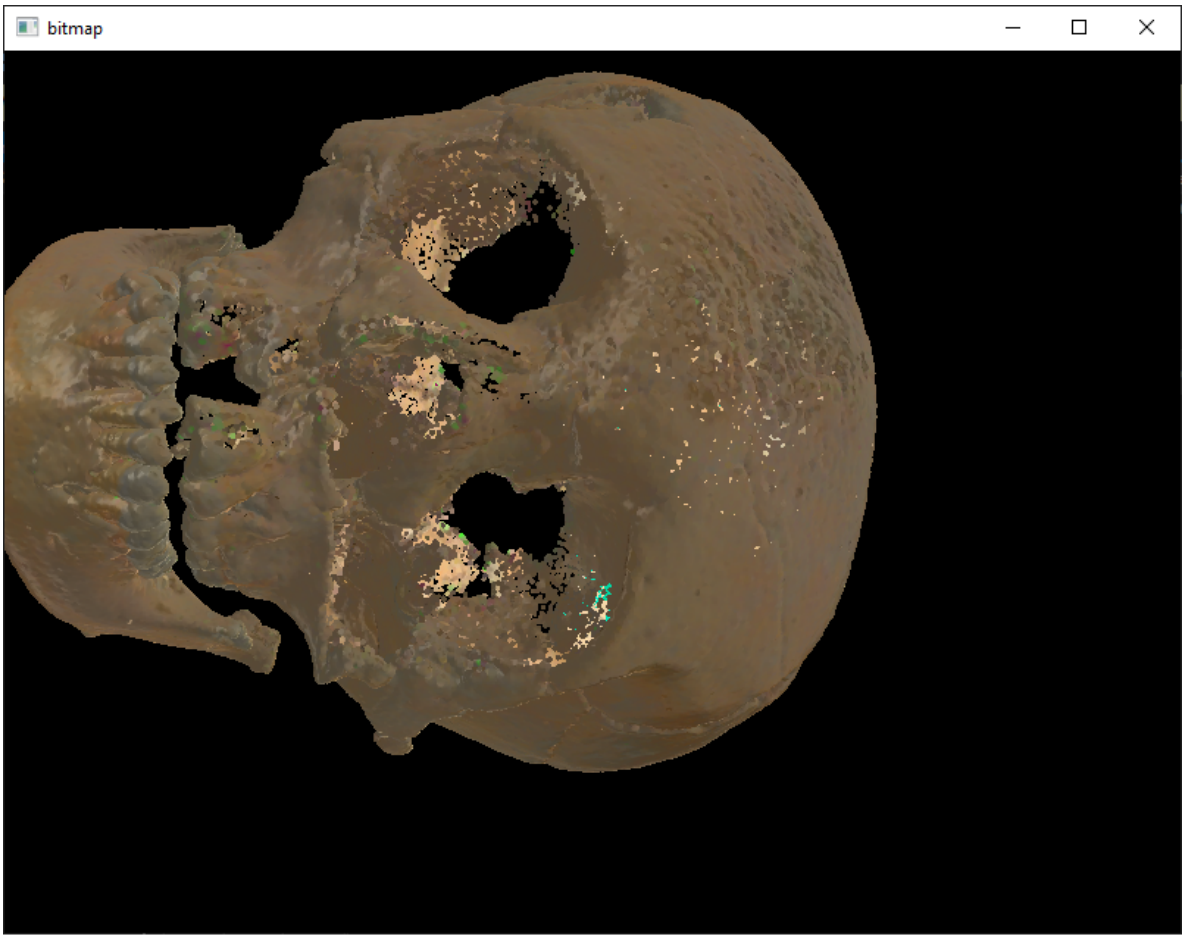


Figura 6.6: Visualización de un mallado 3D representando un cráneo humano. Se utilizó los vértices de cada triangulo como centro de cada *metaball*. Los colores base fueron obtenidos de cada vértice en el mallado y suavizado utilizando las normales para obtener una distribución de color uniforme.

permitía la creación de relaciones directas entre datos y su estructura, lo que permite una interacción más rápida utilizando operaciones binarias en algunos casos. Además, se realizó una implementación del algoritmo de raytracing que permite la visualización de los datos almacenados, utilizando la codificación implementada para acelerar el proceso de trazado de rayos y la interacción con una iso-superficie creada por la fusión de múltiples *metaballs*.

El sistema presentado demuestra ser robusto y versátil para la personalización de las visualizaciones. Las implementaciones presentadas son procesadas completamente en paralelo usando CUDA, alcanzando una paralelización natural debido a la estructuración y codificación de los datos. Los resultados obtenidos mejoran los presentados

Renderizado sin búsqueda de vecinos		
Cantidad Partículas	Profundidad Octree	Renderizado con vecindad(fps)
100	6	164
	7	159
	8	154
1432	6	56
	7	48
	8	35
145777	6	34
	7	25
	8	22
302751	6	24
	7	14
	8	11

Tabla 6.1: Velocidades de renderizado para diferentes cantidades de *metaballs* y nivels del *octree*

Memoria utilizada Octree		
Profundidad Octree	Cantidad Nodos	Memoria usada (mb)
6	299523	4.57
7	2396745	36.57
8	19173961	292.57

Tabla 6.2: Memoria utilizada para almacenar la estructura de datos *octree* usando diferentes profundidades máximas

en el trabajo de Szécsi et al. [44], demostrando el potencial de las técnicas presentadas.

La presencia de artefactos en las visualizaciones solo es perceptibles a escalas bajas por lo que no reducen la calidad visual cuando se visualiza la escena completa o secciones especificas de la misma. Se presentaron problemas de precisión debido a la normalización realizada y a la limitante de representación causada por la cantidad de bits utilizados para almacenar las claves Morton, debido a que se utilizan campos de intensidad para crear la iso-superficie renderizada estos problemas de precisión no producen problemas o una visualización de calidad baja, para lograr esto es importante la selección del radio del campo de intensidad emitida.

Se propusieron nuevas metodologías que pueden llevar a la visualización de campos de intensidad utilizando *octrees* y codificaciones a nivel binario. Obteniendo resulta-

dos comparable con el estado del arte y abriendo nuevas posibilidades para este tipo de visualización. Si bien, los resultados presentados contienen artefactos, estos son únicamente perceptibles cuando se realiza un acercamiento a un área pequeña de la escena representada por el *octree* por lo que son imperceptibles para las aplicaciones finales.

Las técnicas propuestas representan la base para realizar un visualización de campos de intensidad utilizando CUDA®, obteniendo resultados que des muestran el potencial de estas técnicas. Esto abre la posibilidad de un desarrollo futuro para la implementación de algoritmos y técnicas que puedan expandir el potencial de las ya propuestas.

Referencias

[1] D. Streeb, M. El-Assady, D. A. Keim, and M. Chen, "Why visualize? arguments for visual support in decision making," *IEEE Computer Graphics and Applications*, vol. 41, no. 2, pp. 17–22, 2021.

[2] L. Orazi and B. Reggiani, "A novel algorithm for a continuous and fast 3d projection of points on triangulated surfaces for CAM/CAD/CAE applications," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 4, pp. 1240–1245, apr 2022.

[3] V. G. Menon and K. Raju, "Performance analysis of ray tracing based rendering using OpenCL," in *2017 Innovations in Power and Advanced Computing Technologies (i-PACT)*. IEEE, apr 2017.

[4] T. Akenine-Möller, *Real-time rendering*, 4th ed., E. Haines, N. Hoffman, and D. C. Abbey, Eds. Milton: CRC Press LLC, 2018, description based on publisher supplied metadata and other sources.

[5] F. Vidal, F. Bello, K. Brodlie, N. John, D. Gould, R. Phillips, and N. Avis, "Principles and applications of computer graphics in medicine," *Computer Graphics Forum*, vol. 25, no. 1, pp. 113–137, mar 2006.

[6] G. M. Morton, "A computer oriented geodetic data base; and a new technique in file sequencing," International Business Machines Corporation (IBM), Tech. Rep., 1996.

[7] J. F. Blinn, "A generalization of algebraic surface drawing," *ACM Transactions on Graphics*, vol. 1, no. 3, pp. 235–256, jul 1982.

[8] J. Peddie, *Ray Tracing: A Tool for All*. Springer International Publishing, 2019.

[9] S. D. Roth, "Ray casting for modeling solids," *Computer Graphics and Image Processing*, vol. 18, no. 2, pp. 109–144, feb 1982.

[10] NVIDIA, "Cuda c++ best practices guide," May 2021. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

[11] ——, "Cuda c++ programming guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, May 2021. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[12] ——, "Cuda runtime api," May 2021. [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api/index.html

[13] K. Evans, "Morton order improves performance," in *High Performance Parallelism Pearls*. Elsevier, 2015, pp. 477–490.

[14] D. M. Ritchie, "The development of the c language," *ACM SIGPLAN Notices*, vol. 28, no. 3, pp. 201—208, mar 1993.

[15] H. Nishimura, A. Hirai, T. Kawai, T. Kawata, I. Shirakawa, and K. Omura, "Object modeling by distribution function and a method of image generation." *Journal of papers given at the Electronics Communications*, vol. 68, no. 4, pp. 718–725, 1985.

[16] I. H. Murakami S., "On a 3d displaymethod by metaball technique," *Journal of papers given byat the Electronics Communication (in Japanese)*, vol. 68, no. 8, 1987.

[17] G. Wyvill, C. McPheeters, and B. Wyvill, "Data structure forsoft objects," *The Visual Computer*, vol. 2, no. 4, pp. 227–234, aug 1986.

[18] T. L. Kay and J. T. Kajiya, "Ray tracing complex scenes," in *Proceedings of the 13th annual conference on Computer graphics and interactive techniques - SIGGRAPH '86*. ACM Press, 1986.

[19] Q. Huang and Z. Zeng, "A review on real-time 3d ultrasound imaging technology," *BioMed Research International*, vol. 2017, pp. 1–20, 2017.

[20] O. V. Solberg, F. Lindseth, H. Torp, R. E. Blake, and T. A. N. Hernes, "Freehand 3D ultrasound reconstruction algorithms—a review," *Ultrasound in Medicine & Biology*, vol. 33, no. 7, pp. 991–1009, jul 2007.

[21] J. Behley, V. Steinhage, and A. B. Cremers, "Efficient radius neighbor search in three-dimensional point clouds," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, may 2015.

[22] F. Torres, Z. Fanti, and F. A. Cosío, "3D freehand ultrasound for medical assistance in diagnosis and treatment of breast cancer: preliminary results," in *IX International Seminar on Medical Information Processing and Analysis*, J. Brieva and B. Escalante-Ramírez, Eds., vol. 8922, International Society for Optics and Photonics. SPIE, 2013, p. 89220K. [Online]. Available: https://doi.org/10.1117/12.2041806

[23] R. Prager, R. Rohling, A. Gee, and L. Berman, "Rapid calibration for 3-D freehand ultrasound," *Ultrasound in Medicine & Biology*, vol. 24, no. 6, pp. 855–869, jul 1998.

[24] A. Rodtook, K. Kirimasthong, W. Lohitvisate, and S. S. Makhanov, "Automatic initialization of active contours and level set method in ultrasound images of breast abnormalities," *Pattern Recognition*, vol. 79, pp. 172–182, jul 2018.

[25] M. A. J. Hiep, W. J. Heerink, H. C. Groen, and T. J. M. Ruers, "Feasibility of tracked ultrasound registration for pelvic–abdominal tumor navigation: a patient study," *International Journal of Computer Assisted Radiology and Surgery*, may 2023.

[26] K. Atesok, J. Finkelstein, A. Khoury, M. Liebergall, and R. Mosheiff, "CT (ISO-c-3d) image based computer assisted navigation in trauma surgery: A preliminary report," *Injury Extra*, vol. 39, no. 2, pp. 39–43, feb 2008.

[27] M. Brill, H. Hagen, H.-C. Rodrian, W. Djatschin, and S. V. Klimenko, "Streamball techniques for flow visualization," in *Proceedings of the Conference on Visualization '94*, ser. VIS '94. Washington, DC, USA: IEEE Computer Society Press, 1994, p. 225–231.

[28] Y. Dobashi, T. Nishita, H. Yamashita, and T. Okita, "Modeling of clouds from satellite images using metaballs," in *Proceedings Pacific Graphics '98. Sixth Pacific*

Conference on Computer Graphics and Applications (Cat. No.98EX208). IEEE Comput. Soc, 1998.

[29] X. Jinq, Y. Li, and Q. Peng, "General constrained deformations based on generalized metaballs," in *Proceedings Pacific Graphics '98. Sixth Pacific Conference on Computer Graphics and Applications (Cat. No.98EX208)*. IEEE Comput. Soc, 1998.

[30] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, "GigaVoxels," in *Proceedings of the 2009 symposium on Interactive 3D graphics and games - I3D '09*. ACM Press, 2009.

[31] S. Flynn, P. Egbert, S. Holladay, and J. Oborn, "Adaptive fluid simulation using a linear octree structure," in *Proceedings of Computer Graphics International 2018 on - CGI 2018*. ACM Press, 2018.

[32] S. Laine and T. Karras, "Efficient sparse voxel octrees," in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games - I3D 10*. ACM Press, 2010.

[33] M. Pätzold and A. Kolb, "Grid-free out-of-core voxelization to sparse voxel octrees on GPU," in *Proceedings of the 7th Conference on High-Performance Graphics - HPG '15*. ACM Press, 2015.

[34] J. Baert, A. Lagae, and P. Dutré, "Out-of-core construction of sparse voxel octrees," in *Proceedings of the 5th High-Performance Graphics Conference on - HPG '13*. ACM Press, 2013.

[35] W. Song, S. Hua, Z. Ou, H. An, and K. Song, "Octree based representation and volume rendering of three-dimensional medical data sets," in *2008 International Conference on BioMedical Engineering and Informatics*. IEEE, may 2008.

[36] C. J. de Parga Bernal Quirós, "High-perfomance algorithms for real-time gpgpu volumetric cloud rendering from an enhanced physical-math abstraction approach," Ph.D. dissertation, Universidad Nacional de Educación a Distancia, España, 2019.

[37] J. Pan, S. Yan, and H. Qin, "Interactive dissection of digital organs based on metaballs," in *Proceedings of the 33rd Computer Graphics International*. ACM, jun 2016.

[38] O. Mercier, C. Beauchemin, N. Thuerey, T. Kim, and D. Nowrouzezahrai, "Surface turbulence for particle-based liquid simulations," *ACM Transactions on Graphics*, vol. 34, no. 6, pp. 1–10, nov 2015.

[39] X. Xiao, S. Zhang, and X. Yang, "Real-time high-quality surface rendering for large scale particle-based fluids," in *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. ACM, feb 2017.

[40] C. J. dos Santos Brito, A. L. B. V. e Silva, J. M. Teixeira, and V. Teichrieb, "Ray tracer based rendering solution for large scale fluid rendering," *Computers & Graphics*, vol. 77, pp. 65–79, dec 2018.

[41] T. Nishita and E. Nakamae, "A method for displaying metaballs by using bezier clipping," *Computer Graphics Forum*, vol. 13, no. 3, pp. 271–280, aug 1994.

[42] J. Singh and P. Narayanan, "Real-time ray tracing of implicit surfaces on the GPU," *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, no. 2, pp. 261–272, mar 2010.

[43] D. A. S. Valdez, P. Delmas, T. Gee, P. Gutierrez, J. L. Punzo-Diaz, R. Ababou, and A. G. Strozzi, "CUDA implementation of a point cloud shape descriptor method for archaeological studies," in *Advanced Concepts for Intelligent Vision Systems*. Springer International Publishing, 2020, pp. 457–466.

[44] L. Szécsi and D. Illés, "Real-time metaball ray casting with fragment lists," 2012.

Índice Alfabético