UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

MOTOR DE RENDERIZADO VOLUMÉTRICO CON ARQUITECTURA EN PARALELO

TESIS
QUE PARA OPTAR POR EL GRADO DE:
MAESTRO EN CIENCIA (COMPUTACIÓN)

PRESENTA:
CÉSAR ADRIÁN VICTORIA RAMÍREZ

DR. JORGE ALBERTO MÁRQUEZ FLORES **CCADET, UNAM**
DR. ALFONSO GASTÉLUM STROZZI **CCADET, UNAM**

MÉXICO, D. F. ENERO 2015

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 esta tesis.

Agradezco infinitamente a mis tutores el Dr. Jorge Alberto Márquez Flores 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.

Agradezco al posgrado en ciencias e ingeniería de la computación de la UNAM por el espacio y conocimientos que me fueron otorgados a través de los diferentes cursos tomados. 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 CCADET-UNAM y a la Unidad de Investigación y Desarrollo Tecnológico del CCADET en el Hospital General de México por brindarme un espacio dentro de sus instalaciones en el cual pude trabajar en este proyecto durante este tiempo.

Agradezco al CONACYT por el apoyo económico que me brindó durante este tiempo.

Agradezco al Dr. Patrice Delmas y a la Universidad de Auckland por permitirme realizar una estancia de investigación en el laboratorio “Intelligent Vision System” en Nueva Zelanda.

Agradezco a mi familia por el apoyo recibido durante toda mi vida. Especialmente quiero agradecer a mi mama que me brindo su amor, sabiduría y apoyo incondicional en cada etapa de mi vida. Todos mis éxitos son gracias a que ella estuvo conmigo. Agradezco al Dr. Lázaro Joel Victoria Santiago por sus consejos y apoyo en todo momento, por animarme a no rendirme.

Resumen

En la presente tesis se desarrolló un motor de renderizado volumétrico que hace uso de una arquitectura en paralelo (CUDA), basándose en diferentes trabajos previos acerca del renderizado volumétrico 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 procesar nubes de puntos que sirven como información de entrada para el motor de renderizado. Para la generación de las visualizaciones de la nube de puntos se plantea el uso de la técnica de renderizado conocida como *ray tracing*. Basándose en trabajos previos y con el avance de las tecnologías de procesamiento en paralelo se plantea una nueva implementación de la técnica de *ray tracing.*

El *ray tracing* implementado es ejecutado sobre una estructura de datos *octree* el cual es organizado y almacenado en memoria 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*, también asegura una independencia entre los datos lo cual ayuda al procesamiento en paralelo.

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 motor de renderizado, 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 con implementaciones tradicionales en CPU contra las realizadas en esta tesis.

Tabla de contenido

Introducción

La visualización científica es parte esencial en la mayoría de las áreas de investigación en nuestros días, desde el uso en la industria del entretenimiento (cine, videojuegos), aplicaciones médicas (tomografía computarizada, resonancia magnética) y ciencia (simulación, visualización de datos adquiridos por diversos sensores). La cantidad de datos generados por muchas de estas fuentes es muy grande por lo que la visualización es costosa computacionalmente pero indispensable para la comprensión de estos datos.

Existen diferentes metodologías para la visualización de los datos en tres dimensiones los cuales pueden ser divididos en 2 tipos: renderizado de superficie (suface rendering) y renderizado de volumen o volumétrico (*volume rendering*). El *surface rendering* consiste en aproximar una malla poligonal de una superficie que se ajuste lo mejor posible a los datos. *Volume rendering* por otra parte tiene como objetivo realizar una proyección bidimensional (2D) de una colección de datos tridimensionales (3D). Actualmente existen muchos métodos que siguen este principio entre ellos el principal algoritmo es *ray casting* (sin traducción uniforme, a veces “emisión de rayos”) en el cual unos rayos son “lanzados” desde cada pixel de la imagen donde se proyectará la escena hacia esta misma; su dirección la determina el punto focal en el ojo del observador y cada pixel. Siguiendo la trayectoria de cada rayo se busca la intersección más cercana y se calcula el color de cada pixel (si además de esto se sigue recursivamente la trayectoria de este rayo buscando reflexiones y fuentes de luz, se obtiene una implementación muy realista denominada *trazado de rayos* (*ray tracing*).

Con el creciente desarrollo de hardware para despliegue de gráficos, el desempeño de la unidad de procesamiento de gráficos (GPU) se ha incrementado drásticamente y sigue en constante crecimiento y escalamiento. Este desarrollo nos proporciona la capacidad de poder procesar cada vez mayores cantidades de datos en menos tiempo, cambiando la manera en que se procesan los mismos. La perspectiva de los algoritmos implementados en GPU tiene como objetivo el procesamiento masivo de datos de manera paralela y han demostrado ser eficientes herramientas para el

renderizado volumétrico. A finales del 2006 la compañía NVIDIA [1] desarrolló un nuevo hardware y una arquitectura, llamada CUDA [2] (Computer Unified Device Architecture), presentando en febrero del 2007 el primer SDK (Software development kit) para desarrollo. CUDA busca explotar las ventajas de una GPU frente al CPU utilizando el paralelismo que permite los múltiples núcleos en una GPU. CUDA aun presenta limitaciones frente a un CPU, como lo es la necesidad de independizar procesos, la falta de comunicación entre los procesos, la limitante de memoria y la falta de recursividad (en versiones recientes de CUDA se presentaron algunas soluciones a esto) son algunas de ellas. Sin embargo esta tecnología se ha desarrollado rápidamente en los últimos años; se han desarrollado en diversos campos algoritmos adaptados a esta nueva arquitectura, presentando muy buenos resultados que sobrepasan los alcanzados con la metodología tradicional.

Debido a la gran cantidad de datos que se manejan en diversas aplicaciones, una manera de organizar los datos y su espacio en memoria es importante. Una representación discreta del espacio tridimensional usa elementos de volumen denominados *voxeles* que son una alternativa a los triángulos de mallados superficiales, como primitiva de representación; estos últimos por su naturaleza se vuelven complicados de procesar en GPU. Los voxeles ofrecen ventajas que se acoplan de buena manera a la arquitectura de CUDA. La organización de los datos es otro punto importante, los voxeles pueden ser organizados mediante el uso de la estructura de *octrees,* que actualmente es una de las representaciones discretas más utilizadas gracias a su eficiencia en la división y representación de un espacio 3D.

La visualización en muchos casos no es suficiente, por lo que se requiere una interacción con los objetos representados. Lograr esto requiere muchas adaptaciones a las técnicas actuales y una optimización mayor aunque implique la pérdida de resolución, siempre y cuando esta no produzca submuestreo de los rasgos de interés. Aprovechando las tecnologías actuales y combinando diferentes

[1] www.nvidia.com
[2] www.nvidia.com/object/cuda_home_new.html

técnicas de visualización, clasificación, organización y manejo de datos, en este trabajo se busca lograr satisfacer dichas necesidades.

Planteamiento del problema.

Existen áreas de la ciencia (simulación, *stereo vision*, etc.) donde las necesidades que se requieren en las visualizaciones de diferentes tipos de datos son superiores a las actuales. Los sistemas actuales carecen en muchos casos de versatilidad y velocidad. En su metodología para mostrar los resultados de manera gráfica no ofrecen:

- Una visualización clara de los datos.
- Una interacción en tiempo real con los datos.
- Actualmente solo se pueden encontrar soluciones estáticas que limitan la exploración visual.

Objetivos.

Objetivo Principal.

Esta tesis tiene como objetivo principal utilizar Unidades de Procesamiento Grafico (GPUs) y una arquitectura en paralelo para obtener un método de visualización de grandes cantidades de datos dinámicos en forma volumétrica y que permita la interacción en tiempo real

Objetivos específicos.

Como el propósito de cumplir con el objetivo principal se establecieron los siguientes objetivos específicos:

- Crear un algoritmo de renderizado volumétrico que funcione completamente dentro de un GPU, donde la trasferencia de datos entre GPU y CPU sea mínima.
- Definir un algoritmo que no dependa de la recursividad.

- Crear una estructura de datos lineal que pueda definir un espacio volumétrico y que a la vez esta estructura de datos sea independiente en cada elemento.
- Establecer relaciones directas entre la estructura de datos que define al espacio y la información contenida en el espacio.
- Permitir diferentes tipos de visualización de los volúmenes, implementando filtros que permitan personalizar el resultado final.

Organización de la tesis.

El primer capítulo de esta tesis presenta los antecedentes más importantes de los últimos años con respecto a trabajos de *ray tracing* usando arquitecturas en paralelo, también se presentan algunos trabajos donde se realizaron las primeras implementaciones de las técnicas usadas en esta tesis.

El segundo capítulo se presenta los conceptos básicos necesarios para el desarrollo de esta tesis, se describen las técnicas usadas y las estructuras de datos implementas y como estas son definidas.

El capítulo tercero se presenta las diferentes etapas que se siguieron para la implementación de un motor de renderizado en GPU. Primero se describen las estructuras utilizadas y como fueron implementadas en esta tesis, posteriormente se describen las relaciones entre los datos, como fueron implementadas y la utilidad de las mismas.

El capítulo cuarto muestra la implementación de la metodología utilizada para poder actualizar los datos mostrados por el motor de renderizado en tiempo real.

El quinto capítulo muestra los resultados obtenidos al finalizar el sistema, se muestran algunas imágenes y se midieron los diferentes tiempos que toma generarlas.

El sexto capítulo se presenta las conclusiones obtenidas de acuerdo a los resultados. También se proponen mejoras para el sistema y el trabajo que se puede realizar a futuro.

Capítulo 1. Antecedentes y Estado del arte

A continuación revisamos en detalle trabajos clave, indicando el título del artículo la referencia en la bibliografía.

El renderizado volumétrico fue presentado por primera vez por Kajiya en su trabajo titulado "*Ray tracing complex scenes"* (Kajiya, 1984) y uno de los algoritmos más importantes usados es el *Ray tracing* cuyo algoritmo fue utilizado por primera vez en 1968 (Appel, 1968), esta técnica ha sido desarrollada ampliamente en las últimas décadas y con el surgimiento de nuevas tecnologías para el procesamiento computacional, se encontraron y desarrollaron nuevos enfoques a problemas que anteriormente se creía no podían ser más optimizados.

El trabajo reportado en titulado "*Ray casting deformable models on the GPU"* (Patidar y Narayanan 2009) nos presenta una implementación del *ray casting* y su contribución es el diseño e implementación de un algoritmo para *ray casting* en tiempo real de modelos que pueden ser modificados en su estructura. Se enfocan principalmente en el objetivo de un renderizado en tiempo real de un modelo que contenga un millón de triángulos, realizando el *ray casting* en un millón de pixeles (1 millón de rayos).

Kajiya et al además proponen una estructura que divide el área de renderizado en cuadriculas que representan un conjunto de rayos/pixeles. Se acomodan los triángulos en la cuadricula y se limitan los rayos de cada cuadricula para buscar intersección con los triángulos proyectados en la cuadricula. Esto produce conjuntos de rayos y triángulos que pueden ser procesados de manera independiente en una arquitectura paralela como CUDA. El número de triángulos que recaen en cada cuadricula puede ser muy grande. Si los triángulos en cada cuadricula son ordenados por profundidad, la intersección puede ser detenida cuando se encuentre la primer coincidencia. Se platea un enfoque intermedio, clasificando los triángulos por profundidad creando conjuntos llamados *losas* (*slabs*).

Dentro de la implementación descrita no se toman en cuenta rayos reflectados, por lo que no se tiene una implementación de un modelo de iluminación complejo. El

algoritmo que realiza la clasificación y el *ray casting* están implementados en CUDA, sin embargo es una mezcla entre procesamiento en paralelo y serial. *Ray casting* es una operación altamente paralelizable, en contraste con la rasterización, la cual mapea el mundo en el campo visual. En el proceso de *ray casting*, cada rayo necesita procesar todos los triángulos en una malla triangular e identificar cual es el más cercano. Cuando el modelo a procesar es geométricamente complejo o con una gran cantidad de triángulos que lo describan y la imagen a crear es muy grande (muchos pixeles), este proceso es una operación computacionalmente muy tardada aunque no ocupe tanta memoria. Usando las ventajas que ofrece el procesamiento en paralelo, es posible obtener tiempo real en la visualización de los modelos, incluso mientras estos son modificados. Como resultado se observa que aunque la estructura de datos cambia la estructura de cuadriculas no es necesario cambiarla y permanece constante. Aplicado a un modelo deformable con 1 millón de triángulos se obtienen resultados de 20 fps (*frames per second*), aunque para un modelo con la mitad de triángulos pero complejidad geométrica superior se obtienen 26 fps por lo que se concluye que el *ray casting* depende tanto del número de triángulos como de la complejidad del modelo.

El trabajo propuesto por (Laine y Karras 2010) propone una metodología para codificar una estructura de datos *octree* y procesarlo en paralelo usando una GPU; se concentran principalmente en el uso eficiente de un *octree* de voxeles disperso (SVO: *sparce voxel octree*) obteniendo un gran rendimiento en la visualización y navegación. Al usar un SVO y una estructura de datos para representar cada nodo del *octree*, donde se usan 15 bits para codificar un apuntador al hijo, 1 bit como bandera de distancia, 8 bits como máscara para saber si el nodo tiene información o está vacío, 8 bits como máscara para saber si es un nodo final, 24 bits para representar el contorno del nodo y 8 bits como máscara para los contornos (usando 64 bits por cada nodo). Con estas estructuras se puede obtener una optimización de la memoria usada y, para obtener un resultado visualmente mejor, se utiliza parte de la información del nodo para describir un contorno, este contorno ayuda a

obtener más flexibilidad en los bordes del modelo, dando la posibilidad de definir esquinas muy puntiagudas con menor cantidad de voxeles.

A pesar de estas optimizaciones, la memoria disponible en GPU es mucha menor comparada con la disponible en RAM o en disco duro, por esto se crea una estructura y una técnica de almacenamiento que permita una carga y descarga rápida de información entre disco duro, CPU y GPU. Se crearon 2 clasificaciones para el *octree*: en disco duro el *octree* es almacenado en un conjuntos de rebanadas, cada rebanada puede contener a lo más 512 kb de información (la memoria copia información el bloques de 512kb), de esta manera se transfiere información sin tener transferencias parciales. En memoria de GPU la información del *octree* se almacena en un conjunto de bloques, donde el primer bloque contiene el nodo raíz (o nodo padre) y las ramificaciones principales del *octree*, los demás bloques contienen el resto de la estructura con los nodos-hoja, en estos bloques se agregan las rebanadas que son cargadas desde RAM o disco duro. Cuando la memoria de GPU se llena, las rebanadas que no están siendo usadas son descargadas de la GPU para hacer espacio a las nuevas. Los autores obtienen resultados de 140.8 millones de rayos procesados por segundo para una escena de 2048x1536.

Laine y Karras no toman en cuenta la modificación de la estructura de los datos ni la cantidad de memoria ocupada por la generación del *octree* por lo que sus datos residen en memoria de disco duro, memoria RAM y memoria de GPU (moviéndola cuando sea requerida) la reestructuración del *octree* toma tiempo de escritura y lectura.

El trabajo publicado por Jan Elserber, Dorit Bormann y Andreas Nüchter titulado "*Efficiente processing of large 3d point clouds"* (Elseberg, Borrmann, Ncher 2011) propone la implementación de estructuras y metodologías para procesar grandes cantidades de datos (nube de puntos) obtenidos por *scaners* láser. Se propone usar un *octree* como estructura para organizar los datos de manera espacial y usar *ray casting* como técnica de renderizado. Los autores de este proyecto hacen referencia al trabajo propuesto por (Laine y Karras 2010) ya que presentan una construcción

del *octree* bastante similar, pero a diferencia de otras implementaciones la estructura propuesta no guarda punteros hacia los vecinos de cada nodo (se ahorra memoria) sin embargo el *ray casting* aún puede ser realizado de manera eficiente con un número constante de operaciones de punto flotante (FLOPS por su definición en inglés) por rayo.

Esta implementación se concentra principalmente en la eficiencia de la memoria, procesamiento del *octree* y la estructura de datos para la representación de cada nodo del *octree*, buscando con esto un bajo consumo de memoria y una fácil navegación a través del *octree*. La estructura es comparada contra una implementación tradicional de *ray casting* en la cual se guarda toda la información necesaria por nodo; por ejemplo: centro del nodo, tamaño del nodo, apuntadores a los 8 hijos de nodo, cantidad de puntos en el nodo y apuntador a todos los puntos dentro del nodo. Como se mencionó, la implementación busca una eficiencia de memoria mayor, así que se omite información que pueda ser calculada recorriendo el *octree*, finalmente omiten los apuntadores a los 8 hijos, creando solo los apuntadores a los nodos que tengan información. Usando esto e implementando algoritmos para búsqueda de vecinos se obtiene la capacidad de almacenar un 1000 millones de puntos mientras aún es posible para el usuario navegar a través de la nube de puntos. En adición al almacenamiento y visualización también la estructura y el algoritmo permiten una rápida detección de forma y procesos de *scan matching* que podría traducirse como alineación o registro por correspondencia.

Una enfoque hibrido de procesamiento en paralelo, para una implementación de *ray tracing* es presentado en el trabajo titulado "*Iterative Layer-Based Ray tracing on CUDA*" (Segovia, Li y Gao 2009) buscando combinar las dos metodologías (procesamiento en paralelo y serial) para obtener un mejor rendimiento. Dado que la GPU contiene un gran poder de procesamiento, pero aun no es capaz de tener la versatilidad del procesamiento en CPU, es importante tener un balance entre las responsabilidades delegadas a cada parte. La recursividad del *ray tracing* es un problema importante que es difícil de implementar en GPU, al buscar la acumulación

del color en un pixel esto se vuelve algo complejo, por esto un enfoque donde sólo el procesamiento intensivo se le deje a la GPU puede ser una solución óptima.

Separando en etapas independientes el algoritmo de *ray casting*, Segovia et al proponen encapsular kernels de CUDA para diferentes partes del proceso, estos kernels son usados para calcular el color en cada uno de los pixeles y generar una imagen final. El control sobre la creación de la imagen se realiza completamente en el host (CPU) y este tiene control sobre los kernels, llamándolos cuando es necesario. La escena a visualizar está constituida por esferas y planos descritos en un arreglo lineal de estructuras (centro y dimensiones). Las pruebas realizadas se hicieron sobre imágenes de 512x512 pixeles con un arreglo de 51 esferas. Sus resultados muestran un considerable incremento en el rendimiento comparado con otras implementaciones. Uno de los problemas con esta implementación es la manera en que se manejan los datos, la forma en que procesan todo permite solo el tratamiento de figuras básicas que pueda ser definidas con pocos parámetros, si se busca crear figuras más complejas se vuelve muy complicado o requiere un número demasiado grande de primitivas de representación.

En los últimos años muchos proyectos que hacen uso de GPUs han sido presentados, intentan resolver procesamiento intenso de manera más veloz, dándole un nuevo enfoque a problemas ya anteriormente abordados. Otro trabajo presentado que intenta una aplicación directa a la resolución de un problema presente en el área médica es el titulado “Real-Time Medical Image Volume Rendering Based on GPU Accelerated Method” (Fan y Mei 2008). El área de aplicación para el renderizado de volumen es la imagenología médica, donde la información del volumen es obtenida de rayos-X, tomografía computarizada (CT), tomografía de emisión de positrones (PET), etc. En este trabajo se busca un renderizado volumétrico de los datos en las modalidades mencionadas. El renderizado de estos datos es realizado escaneando los datos con los rayos proyectados a partir de una técnica de *ray casting*, cada rayo va acumulando los resultados de las propiedades ópticas de los datos para al final obtener una textura del volumen. En esta implementación se usan los datos para crear una textura, esta textura es cargada en la memoria de GPU para generar la

imagen final. Esta implementación en datos de 512x512x27 logra un rendimiento de 300 fps con opción para poder interactuar con el modelo. La implementación usa iluminación local para reducir el procesamiento por lo que la iluminación no puede ser modificada.

Una implementación que busca el mismo objetivo que el trabajo mencionado anteriormente es el titulado “High-performance and real-time volume rendering in CUDA” (Zhao, Cui y Cheng 2009) en este trabajo se implementa una optimización al algoritmo de *ray casting* usando GPUs, enfocándose en el manejo de memoria y optimización en CUDA. Dado que el modelo de programación en GPU y CPU es muy diferente, las implementaciones tradicionales de volumen rendering necesitan un enfoque nuevo si se quieren adaptar a un procesamiento en GPU, en este trabajo se muestra una nueva implementación orientada a GPU realizando una comparación contra las implementaciones tradicionales en CPU, buscando determinar si existe una ventaja y de qué depende. Esta implementación alcanza un alto desempeño y velocidad, se realizaron las pruebas en un procesador Intel core2 Duo para la versión en CPU, y la versión en CUDA (GPU) usando una tarjeta NVIDIA GeForce8800GT. Para el modelo más pequeño (256x256x256) se obtiene un desempeño de 59 fps para la versión de CPU y 70.6 para la de GPU, en el modelo más grande (512x512x512) se obtienen 30.3 fps para la de GPU y la versión de CPU no les fue posible desplegarla.

En base a los trabajos revisados y la literatura consultada se decidió tomar un aproximamiento hibrido con respecto a la creación del *octree* en el cual se reservara la memoria únicamente de los nodos llenos y los demás serán solo apuntadores vacíos. El *ray tracing* será realizado completamente en paralelo aprovechando las relaciones existentes entre la información y los nodos en el *octree*.

Capítulo 2. Conceptos básicos

2.1. Arquitectura en paralelo (CUDA).

La información sobre la que se basa la explicación de CUDA está basada en los documentos (NVIDIA cuda c programming guide, 2015) (Cuda api reference, 2015) (Cuda c best practices, 2015) proporcionado por NVIDIA y en el apéndice sobre CUDA presentado en la tesis realizada por el Dr. Alfonso Gastélum (Gastélum Alfonso, 2011).

CUDA (Compute Unified Device Architecture) es una arquitectura de cómputo paralelo desarrollada por NVIDIA como un motor de cómputo para las unidades de procesamiento gráfico (GPUs) fabricadas por NVIDIA, CUDA es usada para obtener una solución paralela a problemas computacionales, algunas de las ventajas que CUDA ofrece son:

- Programación heterogénea serial y paralela: La programación en CUDA involucra ejecutar código en 2 diferentes plataformas de manera simultánea: Un *host*, sistema que contiene uno o más CPUs y uno o más *devices,* dispositivos GPU de NVIDIA. El CPU (host) y el (o los) GPUs (device) se comunican mediante la transferencia de memoria. Esta operación puede ser realizada en cualquier momento durante el programa, permitiendo que las implementaciones en CUDA sean intermitentes entre seriales y paralelas.

- Soluciones escalables: El número de núcleos e hilos (*threads)* pueden ser adaptados a cada problema en específico, haciendo posible escalar (dentro de los límites del hardware) el número de elementos (*blocks* y *threads*) para obtener la mejor solución al problema. Cuando el número de elementos fue definidos por el un hardware en específico la misma implementación puede ser usada en otros modelos de GPU y CUDA se encargará de escalar el problema dentro de los límites de hardware.

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

Las soluciones implementadas en CUDA se ejecutan en el host CPU y es desde el host donde se realizan las llamadas a GPU (*device*), las soluciones en GPU son realizadas con el uso de kernels, los cuales son escritos en C/C++. Un *device* ejecuta un kernel a la vez y el kernel es resuelto en N threads en paralelo. La Figura 1 muestra un esquemático que representa la progresión del código entre host y device, también se muestra la transferencia de memoria necesaria.

Cada kernel define el código que será resuelto por todos los threads en paralelo, cada thread resuelve el mismo código con diferente información. El uso de condicionales o condiciones de control (switch, do, for, while) es posible pero puede afectar considerablemente el rendimiento de ejecución de instrucciones ya que se crean ramificaciones, si esto ocurre cada ramificación será resuelta de manera serial.

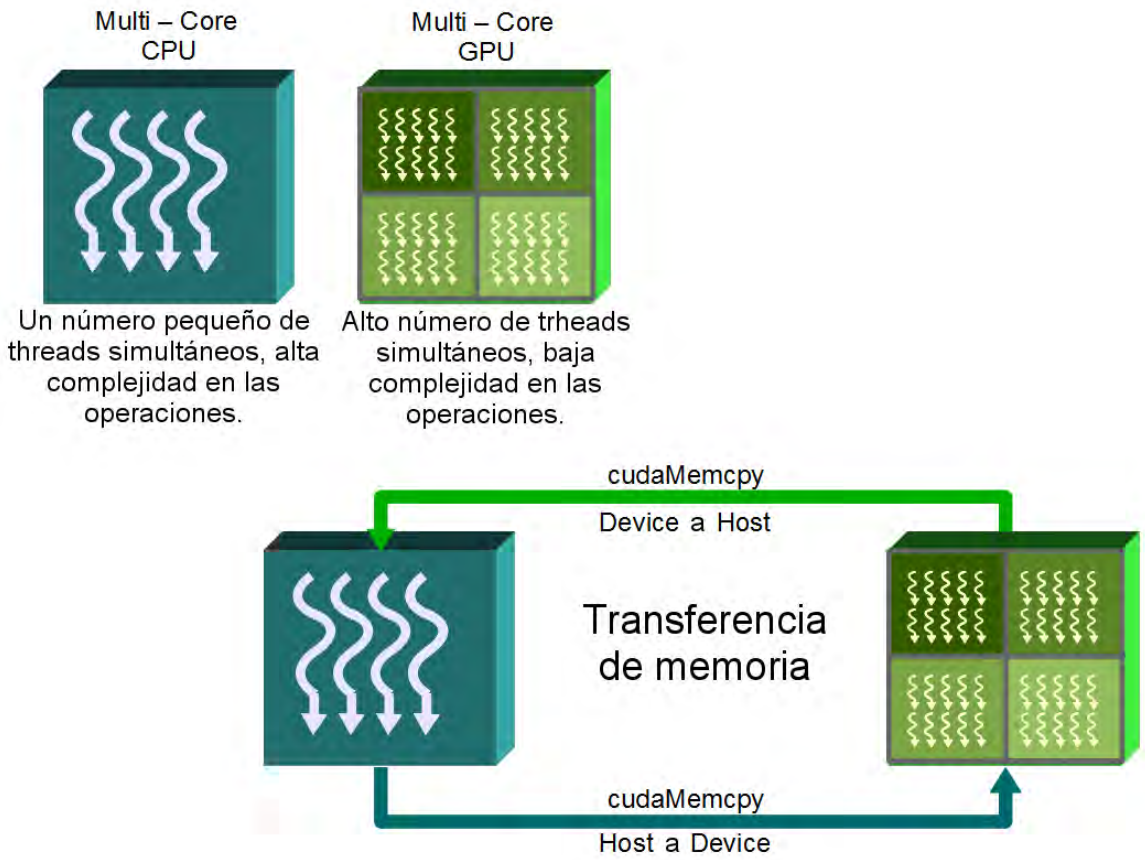

Figura 1 Progresión heterogénea de una implementación en CUDA – La progresión heterogénea de una implementación en CUDA permite mezclar soluciones entre componentes seriales y paralelos. El costo computacional para un sistema incrementa con el número y tamaño de la memoria transferida.

CUDA organiza las solución en paralelo usando una representación en *grid* (cuadricula) del dispositivo GPU, cada retícula (*grid*) es dividida en *blocks* (bloques) y cada *block* contiene *threads* de CUDA. Para poder hacer uso de la organización de *threads* en CUDA, el API (*Application Programming Interface*) ofrece tres vectores tridimensionales variables: threadIdx, blockIdx y blockDim.

Cada *block* contienen un numero N de *threads*, el tamaño de cada bloque es definido por el usuario en cada dimensión. El usuario también proporciona el número de blocks por grid. Cada block puede contener hasta 1024 threads, y para poder ejecutar más threads el kernel puede ser ejecutado con muchos blocks del mismo tamaño a la vez (la cantidad de bloques depende de la capacidad computacional de

la GPU, los detalles pueden ser encontrado en la tabla 13 de la guía de programación de NVIDIA (NVIDIA cuda c programming guide)).

Un GPU multiprocesador NVIDIA está diseñado para ejecutar cientos de threads de manera simultánea usando la arquitectura llamada SIMT (Single-Intruction, Multiple-Thread). La figura 2 muestra la relación que existe entre grid, block y thread y cómo se organizan. El grid representa al device siendo usado y la figura muestra la estructura 3D de los threads y blocks.

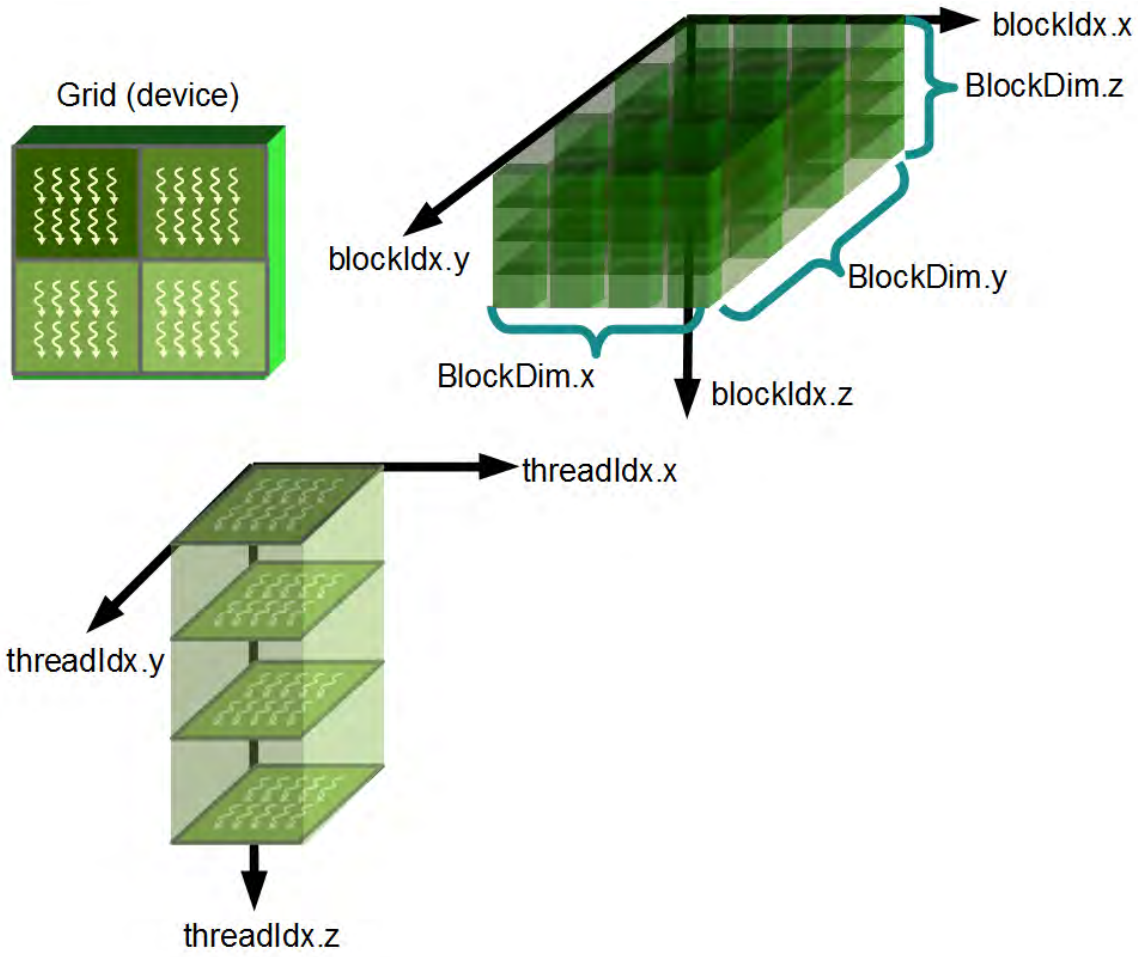

Figura 2 estructura lógica de CUDA- CUDA hace uso de elementos grid, block y threads para organizar el procesamiento en paralelo

Si cada uno de los threads individualmente fuera usado para leer directamente un elemento en un arreglo lineal, el índice lineal para cada una de las 3 dimensiones

se obtiene usando el índice del thread y block. El índice es definido por la ecuación A.1 para 2D y la ecuación A.2 para 3D:

$$\begin{cases} unsignedintx = threadIdx.x + (blockIdx.x \times blockDim.x) \\ unsignedinty = threadIdx.y + (blockIdx.y \times blockDim.y) \\ unsignedintw = array\,width \\ unsignedintindex = x + (y \times w) \end{cases} \quad (A.1)$$

$$\begin{cases} unsigned\ int\ x = threadIdx.x + (blockIdx.x * blockDim.x) \\ unsigned\ int\ y = threadIdx.y + (blockIdx.y * blockDim.y) \\ unsigned\ int\ z = threadIdx.z + (blockIdx.z * blockDim.z) \\ unsigned\ int\ v = array\,width \\ unsigned\ int\ w = array\,depth \\ unsigned\ int\ index = x + (y * v) + (z * v * w) \end{cases} \quad (A.2)$$

Usando la combinación de threadIdx, blockIdx y blockDim cada uno de los threads en el kernel pueden ser identificados. La figura 3 muestra la relación entre los blocks y threads, nótese que dentro de cada block el índice para cada thread inicia en 0.

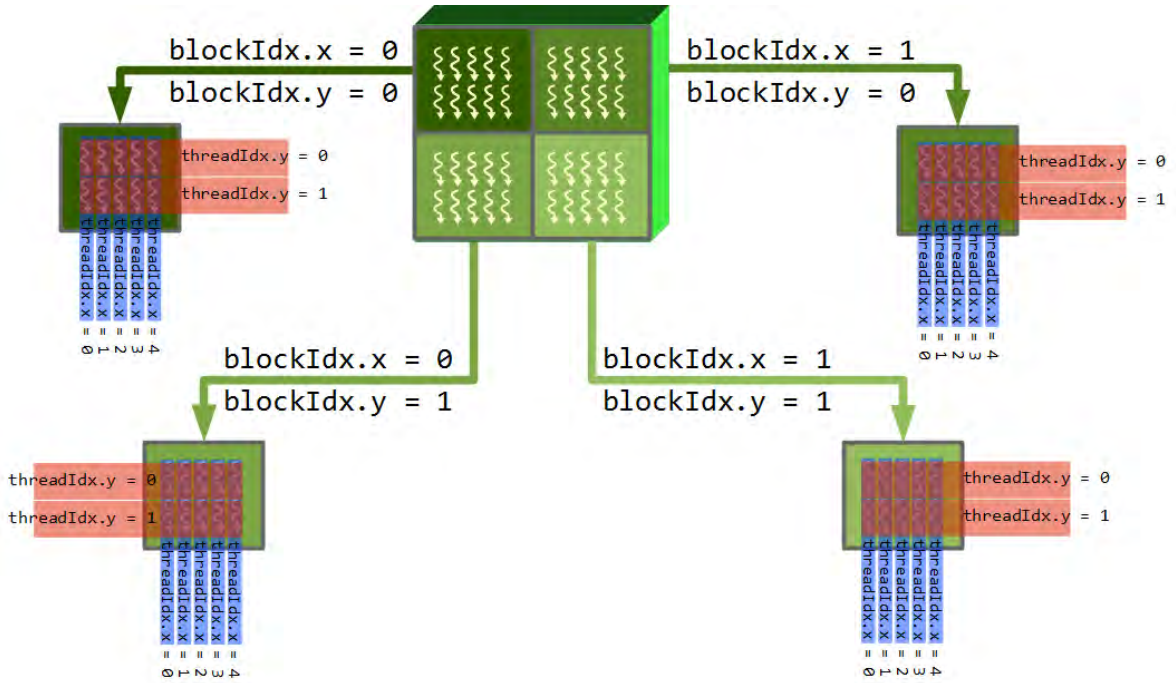

Figura 3 CUDA Blocks y threads índices - Cada uno de los blocks contiene un grupo de threads que serán ejecutados en paralelo, el índice de cada thread dentro de un block inicia en 0 y el índice completamente individual de cada thread es una combinación de los valores contenidos en threadIdx, blockIdx y blockDim.

Las funciones dentro de un kernel son ejecutadas una a la vez dentro del grid. Las operaciones dentro del device pueden tener diferente tiempos de cómputo para diferentes threads. Para forzar al algoritmo a esperar por la finalización de todos los threads dentro de un kernel antes de continuar con la ejecución, se utiliza el comando syncthreads(). Los blocks por otra parte no pueden ser sincronizados, esto se debe a que los block son resueltos en cualquier orden, y más importante, los blocks pueden ser resueltos de manera serial o en paralela. CUDA maneja lo blocks 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 los permite el uso de diferentes tipos de GPUs con diferentes especificaciones ni necesidad de modificar el código dentro de los kernels. La figura 4 muestra un ejemplo de 2 sistemas diferentes, uno con 2 núcleos y otro que contiene 4 núcleos.

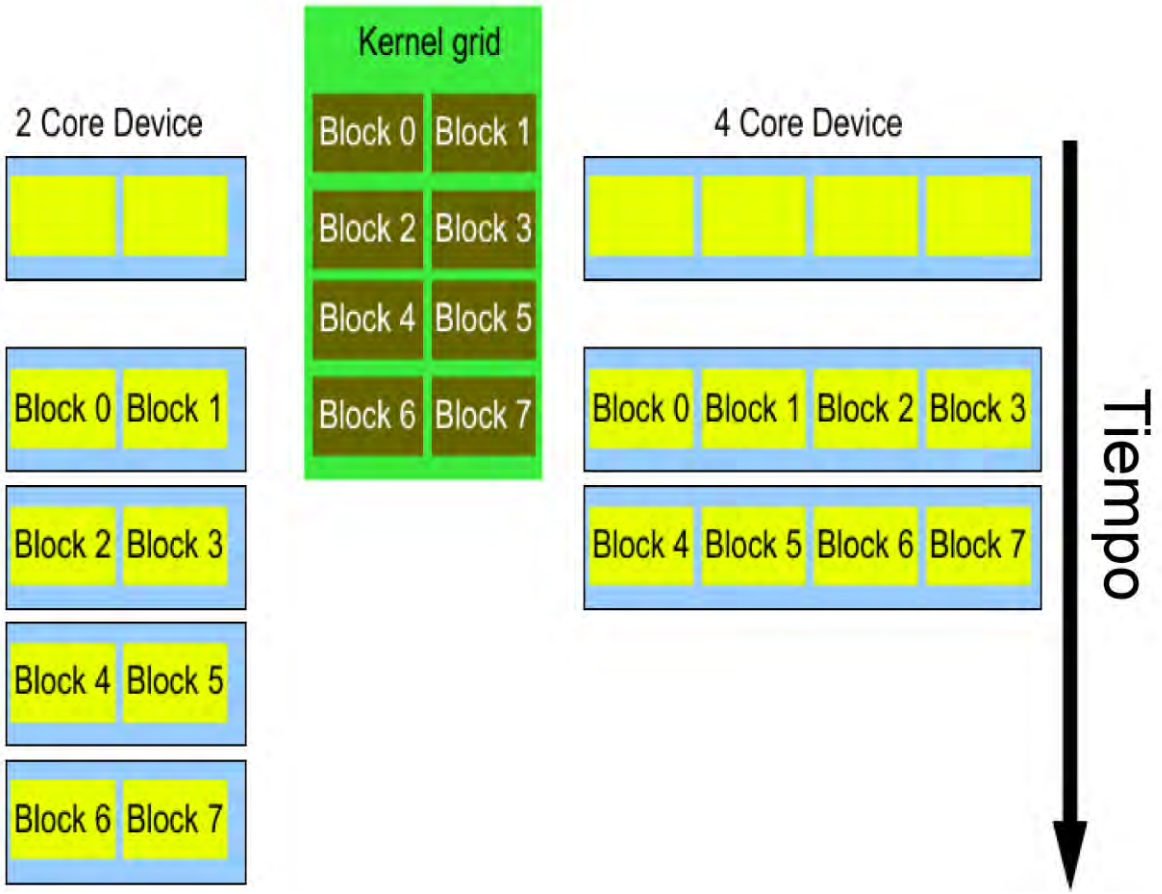

Figura 4 Las implementaciones realizadas en arquitecturas CUDA será escalables a las capacidades de la GPU en la que es ejecutada. Entre más núcleos dentro del sistema, menos tiempo computacional será requerido por el sistema para dar solución al algoritmo.

2.2. Clave Morton.

En ciencias de la computación la *Clave Morton (Morton* Key) también llamada *ordenamiento Z* se refiere a una técnica que mapea datos multidimensionales a una sola dimensión y viceversa. Fue introducida en 1966 por G.M. Morton (Morton, G. M. 1966) y actualmente es muy usada para el ordenamiento de información multidimensional ya que una vez implementado este ordenamiento, es más simple la búsqueda espacial. Estas propiedades hacen a esta codificación óptima para el uso de quadtrees y *octrees*.

Para obtener el valor de la clave Morton de datos multidimensionales a una dimensión se calcula intercalando las representaciones binarias de los valores en cada dimensión para obtener un único valor. Si aplica esta codificación 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 por lo cual la clave Morton es ampliamente llamada “ordenamiento Z”, figura 5.

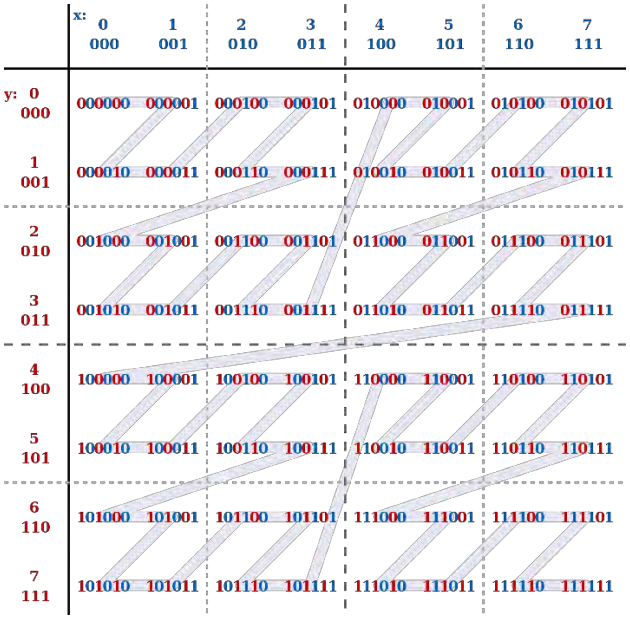

Figura 5 Claver Morton. Se puede observar el patrón en Z formado cuando se busca encontrar un orden en las claves Morton generadas. [3]

[3] Imagen obtenida de: https://en.wikipedia.org/wiki/Z-order_curve#/media/File:Z-curve.svg

2.3. Estructuras de datos Octrees (árboles).

Una estructura de datos usada para la representación jerarquita de manera gráfica son los grafos de tipo árbol. La estructura de árbol está compuesta por elementos individuales llamados nodos que se encuentran conectados por líneas llamadas ramas. En una representación de árbol se usan las relaciones entre los nodos para representar la jerarquía. El nodo en el nivel superior (nivel 0) del árbol es el nodo raíz, el árbol está formado por padres e hijos nodos. Un nodo padre es superior en jerarquía que su hijo (hijos) y ambos comparten una conexión. Los nodos que no tienen hijos propios son llamados nodos hoja y se encuentran en el nivel inferior de cada división del árbol la figura 1 muestra un árbol binario completo y la figura 2 muestra uno parcial.

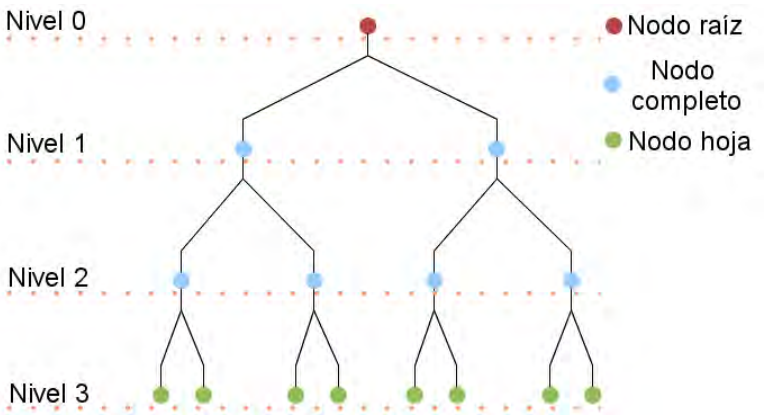

Figura 6 Árbol binario completo

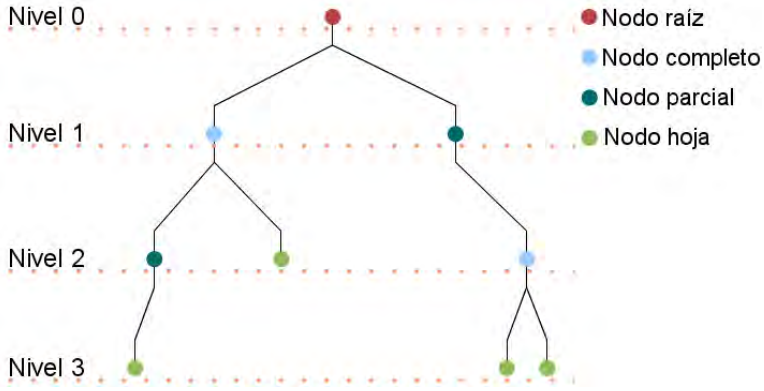

Figura 7 Árbol binario parcial

2.4. Octree.

Un *octree* (árbol octal) es un tipo de árbol donde cada nodo tiene entre 0 y 8 nodos hijos. Esta estructura de datos es usada principalmente para dividir un espacio tridimensional donde todo consiste en que cada nodo padre tiene exactamente 8 nodos hijos, de esta manera es posible particionar el espacio dividiendo recursivamente en ocho octantes por nivel, cada división es realizada con respecto al centro del nodo original para obtener 8 regiones exactas (figura 3).

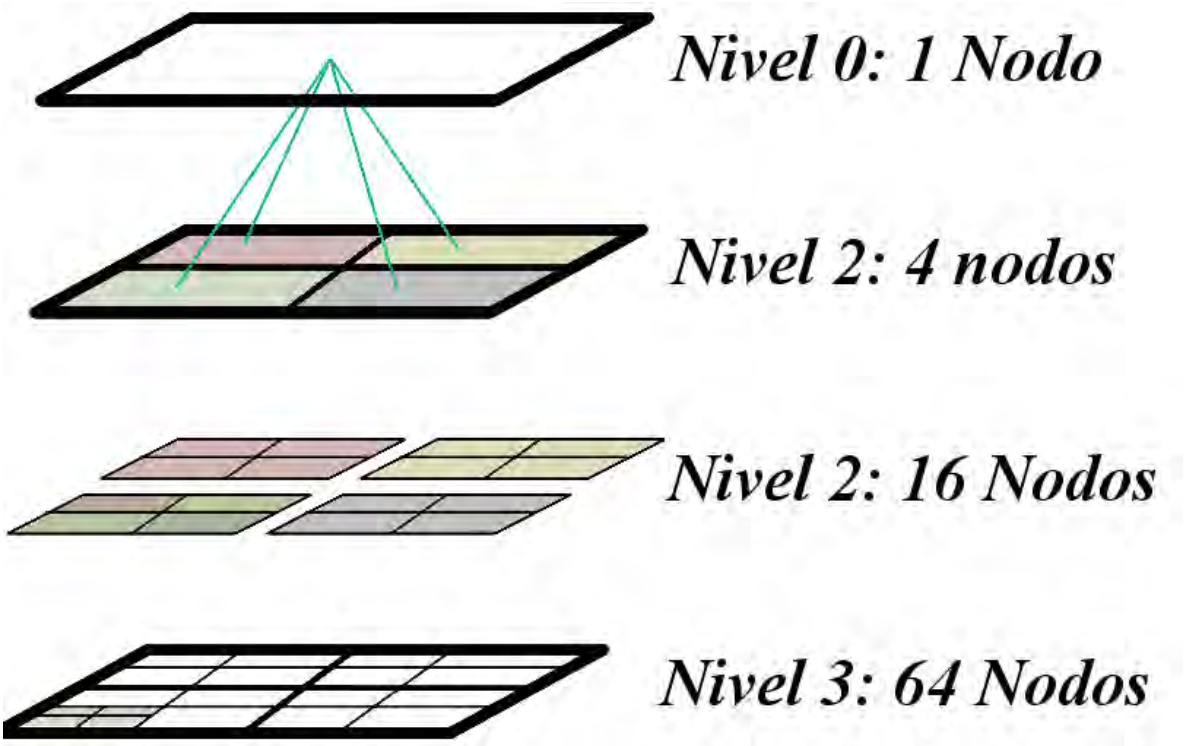

Figura 8 Estructura de niveles dentro del Octree

Uno de los problemas al usar este tipo de estructuras es la cantidad de memoria utilizada, ya que la cantidad de nodos crece exponencialmente conforme se aumenta la profundidad (incrementando la resolución del *octree*) este crecimiento conlleva un gran uso de memoria para poder almacenar la estructura completa; dado que se está trabajando con CUDA la memoria que se tiene disponible es aún más limitada que en las implementaciones tradicionales, utilizamos una variación de esta estructura de datos llamada *Sparce Octrees* (arboles dispersos) en la cual solo se crean los nodos que contengan información, dejando los demás nodos sin

definir o inexistentes con el propósito de ahorrar memoria. Los *sparce Octrees* son un tipo de *octree* parcial.

2.5. *Ray tracing*

Ray tracing es un algoritmo de renderizado de imágenes a partir de una escena tridimensional, esta técnica busca emular el comportamiento de la luz, trazando rayos desde el observador (cámara) hacia la escena, así como también toma en cuenta las fuentes de iluminación y los rayos que estas producen. Es una extensión del algoritmo *ray casting*.

2.5.1. Ray casting

El algoritmo *ray casting* consiste en tener un plano correspondiente a la imagen a generar, cada pixel de este plano es un rayo que es proyectado en dirección a la escena. Para todo rayo de la imagen se busca determinar su intersección con todos los objetos en la escena tridimensional, una vez obtenidos las intersecciones se busca para cada rayo, el objeto más cercano a la cámara con el que se tenga intersección, el color del objeto determinara el color del pixel, figura 9.

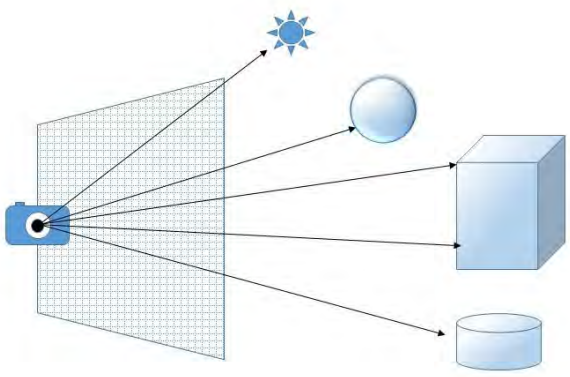

Figura 9 Ray casting.

Ray tracing busca expandir esta idea incluyendo puntos de iluminación en la escena los cuales afectan a las intensidades de todos los pixeles, agregando en la escena sombras y diferentes efectos visuales como pueden ser reflexiones y difracciones.

Esta técnica busca emular el comportamiento de los rayos de luz, los cuales se reflejan en el espacio afectando su color dependiendo de los objetos que intersectan y las propiedades del mismo.

El objetivo final es determinar para cada rayo de luz que intersecta el plano de la imagen a generar, el color del mismo. Este comportamiento de los rayos ignora muchos factores físicos y es una generalización del comportamiento de la luz, figura 10.

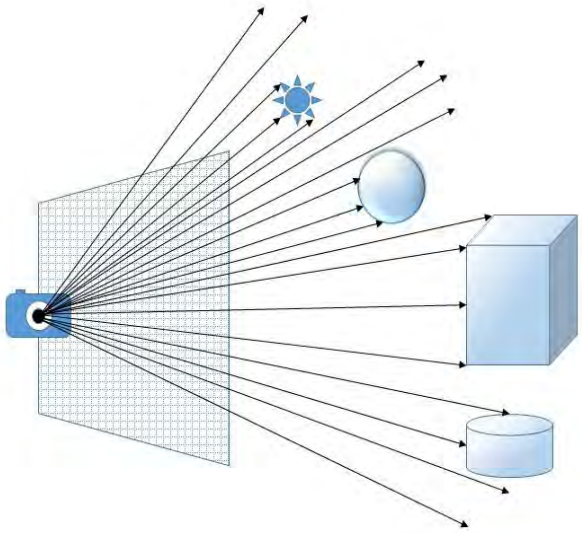

Figura 10 Todos los rayos proyectados desde la cámara a través del plano de la imagen generaran un color dependiendo del objeto con que intersecten.

La suposición natural con respecto a esta técnica es que los rayos de luz que determinan la intensidad del pixel de la pantalla son trazados desde el punto de origen (la fuente de luz) hacia su destino que es el plano de la imagen. Esta forma es bastante precisa en el cálculo de la intensidad para cada pixel pero hace más complejo el procesamiento necesario ya que se pueden tener una cantidad muy grande de rayos que no chocaran con el plano de la imagen.

Siguiendo este razonamiento en una escena, es necesario determinar cuántos rayos proyecta cada fuente de iluminación, una vez determinado esto es necesario asignar una dirección a cada rayo. Como se mencionó solo algunos de estos rayos alcanzarán el plano de la imagen, ya sea de manera directa o indirecta. La cantidad

de rayos puede ser infinita por cada fuente de luz y para todos es necesario determinar cuáles logran alcanzar el plano de la imagen y cuales se pierden en el espacio. Los rayos perdidos no tiene sentido calcularlos ni seguir su trayectoria en la escena ya que serían desperdicio de procesamiento, pero no es posible determinar de manera previa cuáles intersectan y cuales no lo cual hace esta forma poco viable computacionalmente hablando, figura 11.

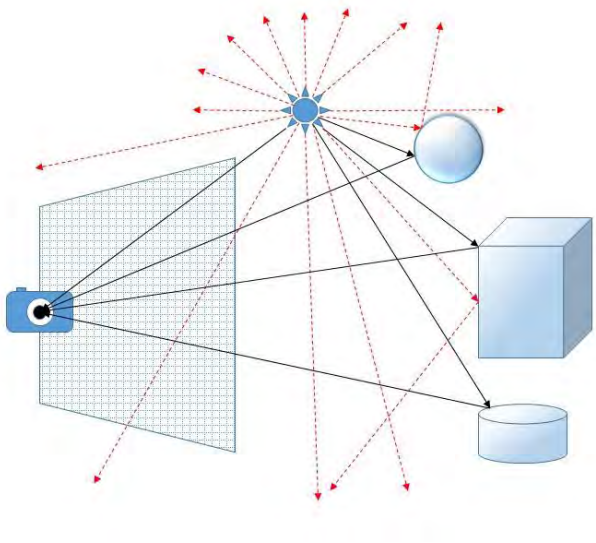

Figura 11 Los rayos proyectados desde la fuente de luz, pueden tender a infinito y entre más fuentes de luz se tenga más procesamiento es necesario.

Con el propósito de realizar solo los cálculos necesarios, se busca procesar solo los rayos que llegan al plano de la imagen. Para lograr esto se aborda el problema de manera inversa al proceso planteado previamente, en vez de probar con todos los rayos proyectados por las fuentes de iluminación, se proyectan rayos desde el plano de la imagen (cantidad de rayos a proyectar depende del tamaño del plano de imagen) hacia la escena, se sigue la trayectoria de estos rayos buscando un punto de iluminación como objetivo final. De acuerdo a todas las intersecciones que se encontraron, antes de que el rayo tocara el punto de iluminación, se determina la intensidad del pixel. De esta forma se prueban únicamente los rayos que intersectan el plano de la imagen, los cuales tendrán la misma trayectoria que los proyectados por las fuentes de luz, pero en dirección contraria.

Como se observa, también existen rayos que no alcanzan ningún punto de iluminación y por lo tanto se pierden en el espacio, estos rayos no son desperdiciados ya que también asignan un color a la imagen que se genera. También existen rayos que pueden reflejarse en la escena una gran cantidad de veces antes de alcanzar una fuente luminosa, por esto también es necesario establecer un límite en la distancia que puede viajar un rayo y la cantidad de reflexiones que puede tener.

Esta técnica permite agregar diferentes efectos visuales durante el seguimiento del rayo, ya que es posible asignar propiedades a los diferentes objetos en la escena, como es refracción y reflexión. La reflexión constituye una propiedad de los objetos para reflejar los rayos que choquen con él, el factor de reflexión de un objeto determina en qué proporción el rayo será reflejado y afectado por las fuentes de iluminación, un factor de reflexión alto implica que el rayo es reflejado con la misma intensidad inicial (como sería el caso de un espejo) mientras un factor bajo implica un rayo reflejado en menor intensidad por lo tanto será afectado en menor cantidad por las fuentes de iluminación. Con la propiedad de reflexión se puede simular fácilmente objetos brillosos y opacos.

La propiedad de refracción por otra parte, implica que un rayo puede atravesar un objeto pero su dirección e intensidad es modificada dependiendo del factor de refracción del objeto, los objetos actúan como modificadores; un ejemplo es el agua u objetos que presenten propiedades de transparencia. Dependiendo del factor de refracción es qué tanto es modificada la dirección del rayo, el nuevo rayo refractado es calculado usando la ley de Snell.

Capítulo 3. Implementación del motor de renderizado en GPU.

3.1. *Sparce Octree* unidimensional

Con la finalidad de tener una estructura más versátil que permita el ahorro de memoria, así como una creación y manejo más óptimo al ser implementado en CUDA, el *Sparce Octree* es creado de manera que pueda ser visto como un arreglo de una dimensión en el cual el nodo al nivel 0 ocupará la posición 0 del arreglo, los 8 nodos del nivel 1 ocuparán la posición 1-8 para el nivel 2 que contiene 64 nodos se le asignará las posiciones 9-72 y así sucesivamente hasta el nivel deseado.

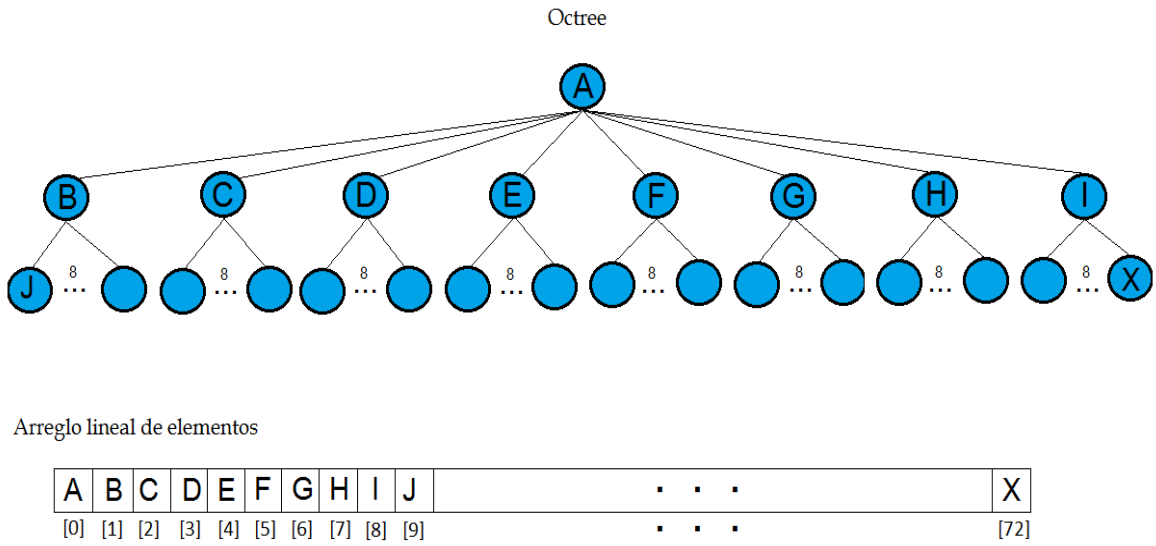

Figura 12 Octree y su representacion dentro de una estructura lineal.

3.2. Implementación de claves Morton.

Para propósitos de esta tesis se utilizara el término partículas para referirse a puntos en el espacio con diferentes propiedades que permitirán su representación e interacción. Las partículas contienen propiedades de posición en el espacio, color en rgb (rojo, verde y azul), factor de refracción y difracción.

3.2.1. Clave Morton para partículas.

Las partículas no tienen un volumen definido ya que son puntos existentes en el espacio. Se definen únicamente por sus coordenadas (x, y, z) más algunas propiedades útiles para la visualización como color (r, g, b) refracción y difracción. Estas últimas 2 propiedades corresponden a diferentes comportamientos de los rayos de luz que chocan con una superficie, dependiendo de las propiedades de la superficies se refleja la luz de diferente manera como lo es en objetos brillantes (como vidrio o metal) y objetos suaves (como madera).

Dado que se busca una manera de ordenar las partículas y relacionarlas directamente con la estructura de datos usada en esta tesis se eligió representar los datos espaciales de las partículas con su clave Morton, lo cual nos permite relacionar de mejor manera la posición de las partículas con la estructura de datos.

Las partículas tienen una posición de (x, y, z) que indica su ubicación en el espacio. Para convertir las coordenadas de estas partículas a su correspondiente clave Morton debemos tomar cada uno de los elementos en binario y combinarlos de la siguiente manera:

$$(xxx, yyy, zzz) = (zyxzyxzyx)$$

Ejemplo:

$$(x, y, z) = (10, 3, 15) = (1010, 0011, 1111) = (101100111110)$$

Para cada partícula se ocupara una clave Morton de 32 bits de los cuales los 2 bits más significativos será ocupados para indicar que la partícula existe en el árbol y que pertenecen al espacio del *Octree*. Los 30 bits restantes se ocuparán para la coordenadas x, y y z, usando 10 bits por coordenada, lo cual nos dará a lo más 1025 posibles valores para cada coordenada. Dada la limitante por coordenada se debe normalizar con respecto a 1024 que es el valor mayor que se puede alcanzar, esto se realiza usando las dimensiones del *octree*, tomando el máximo en cada dimensión para usarlo como valor de normalización (1024 será equivalente al valor mayor) y tomando todo valor para transformarlo a su correspondiente entre 1024 y

0. Esto se realiza dividiendo el valor requerido entre el valor mayor para luego multiplicarlo por 1024 obteniendo el nuevo valor normalizado.

Una vez obtenido el valor normalizado se trasformaran directamente las coordenadas x, y y z a su respectiva clave Morton, dejando los 2 bits más significativos en 01 (indicando la pertenencia al espacio del *octree*).

Ejemplo:

(x, y, z) = (10, 3, 15)= (0000001010,0000000011,0000001111)
= (000000000000000000101100111110)

El resultado de las combinaciones de las coordenadas nos dará los bits del 1 al 30. Asignamos los 2 bits más significativos faltantes y así obtenemos la clave Morton para la partícula con coordenadas (10, 3, 15):

0 1 000 000 000 000 000 000 101 100 111 110

3.2.2. Clave Morton para nodos.

Al igual que las partículas los nodos del *octree* usaran una clave Morton como identificador en el espacio del *octree*, esto nos permitirá un ordenamiento sobre los nodos y una relación entre el espacio del nodo y sus partículas.

Se usará una clave Morton de 32 bits, de los cuales los 2 bits más significativos indicarán el nodo padre o nodo raíz (el cual describe todo el espacio ocupado por el *octree*). Para generar la llave Morton completa (usando los 32 bits) para los nodos en cualquier nivel, se usará la esquina inferior más cercana al origen como coordenada (x,y,z) y se combina de la misma manera que las partículas dado que el nodo padre es el inicial y se describirá con 01 cuyo centro es (512, 512, 512) y cuya esquina es (0,0,0) la cual es representada en cada dimensión con 0 000 000 000 (0) al combinar nos da 01 000 000 000….000 la cual es la calve Morton del nodo padre, esta clave también nos permite filtrar todas sus partículas contenidas

dentro del nodo como se explicara más adelante. Al realizar este proceso también se crea un orden sobre las llaves Morton de todos los nodos.

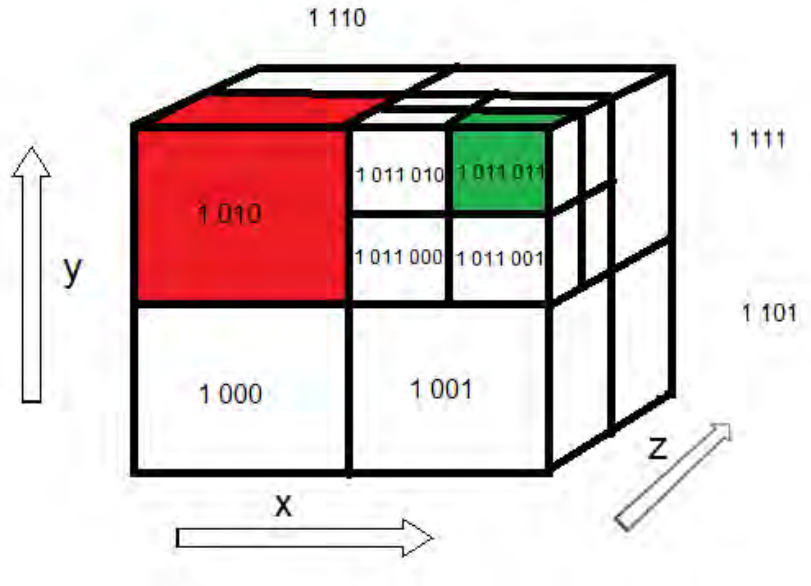

Figura 13 Distribución de las claves Morton dentro del octree.

Los centros de los nodos son con respecto un volumen de 1024 unidades en los 3 ejes por lo que los centros siempre serán potencias de 2, esto nos permite identificar rápidamente las Morton sin necesidad de leer toda la Morton y se pueden calcular todos los centros usando una máscara de bits justo antes de decodificar las claves Morton.

Las claves Morton siguen un orden especifico, para el nivel 1 se tienen 8 nodos, a ese nivel solo se consultan los primeros 4 bits de la llave Morton completa (que es de 32 bits), estos 4 bits componen la llave Morton a ese nivel de profundidad, por lo que se tienen 8 posibles nodos: 01 000, 01 001, 01 010, 01 011, 01 100, 01 101, 01 110, 01 111. Si se desea aumentar 1 nivel de profundidad solo se toman las combinatorias de los 3 bits siguientes lo cual darán los 8 hijos siguientes.

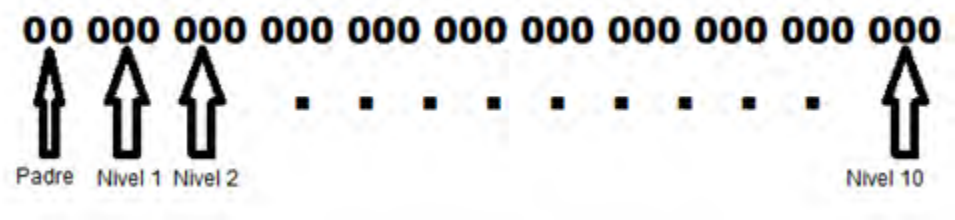

Figura 14 Distribución de la clave Morton. Cara nivel del Octree representado por la clave Morton ocupara 3 bits.

Cada 3 bits indicaran un nivel de profundidad en el árbol por lo que se tendrá a lo más 10 niveles de profundidad, los cuales son suficientes para obtener una resolución de más de mil millones de nodos.

Ejemplo:

Tomando como origen una de las esquinas del nodo raíz (como se ilustra en la figura 1, donde el origen es la esquina inferior izquierda de la cara frontal) al nivel 0, donde todo es solo un área. El centro estará ubicado en (512, 512, 512) lo cual nos dará una clave Morton de:

(512, 512, 512)= (1000000000, 1000000000, 1000000000) = 01 111 000 000 000 000 000 000 000 000 000 000.

Agregando el corrimiento de los 3 bits del nodo padre nos queda

01 000 000 000 000 000 000 000 000 000 000 000

El nivel cero (solo el nodo padre) será identificado por 01 en sus 2 bits más significativos; dada la naturaleza de un *octree*, al subir un nivel de profundidad, tendremos 8 divisiones que descienden del nodo padre a los cuales se les denomina hijos, estos hijos serán identificados mediante los 3 bits siguientes de la clave Morton (000, 001, 010, 011,...111) a su vez en cada nivel extra de profundidad se usaran los 3 bits siguientes y de esta manera de tendrá una descripción del árbol y una relación entre padres e hijos. Dada una clave Morton es posible obtener las claves de los nodos hijos como también la del nodo padre.

3.3. Relaciones entre nodos y partículas

La estructura de datos *Sparce Octree* tiene la propiedad de no crear nodos que no tienen información con el propósito de ahorrar memoria lo cual en una tarjeta de video es necesario. Para lograr crear solo los nodos con información se necesitan saber que nodos contienen partículas y la cantidad contenida, una de las maneras más comunes es mediante una búsqueda dentro del árbol para cada partícula, este método es lento cuando se tiene un numero grande partículas así que en esta tesis se plantea una relación directa entre las partículas y sus nodos contenedores la cual nos permite obtener una descripción del *Sparce Octree*.

Uno de los propósitos principales de esta tesis es una manera rápida y eficaz de filtrar la información, lo cual nos permite identificar rápidamente la posición de las partículas dentro de la estructura de datos. Entre más directa sea esta relación más rápido se podrá actualizar la estructura de datos cuando la información cambie de posición.

Se está usando claves Morton para las partículas como para los nodos dentro del *octree* lo cual nos permite obtener una relación directa entre ambos y aplicando únicamente una máscara de bits podemos determinar a cualquier nivel de profundidad del *octree* a que nodo pertenece cada partícula, esta relación es ejemplificada en la figura 15 donde se pude observar que dependiendo de la profundidad de los nodos será la cantidad de bits usados para representar la clave Morton, estos bits serán los mismos en la clave Morton de las partículas y pueden ser usados para filtrar las partículas contenidas en el nodo.

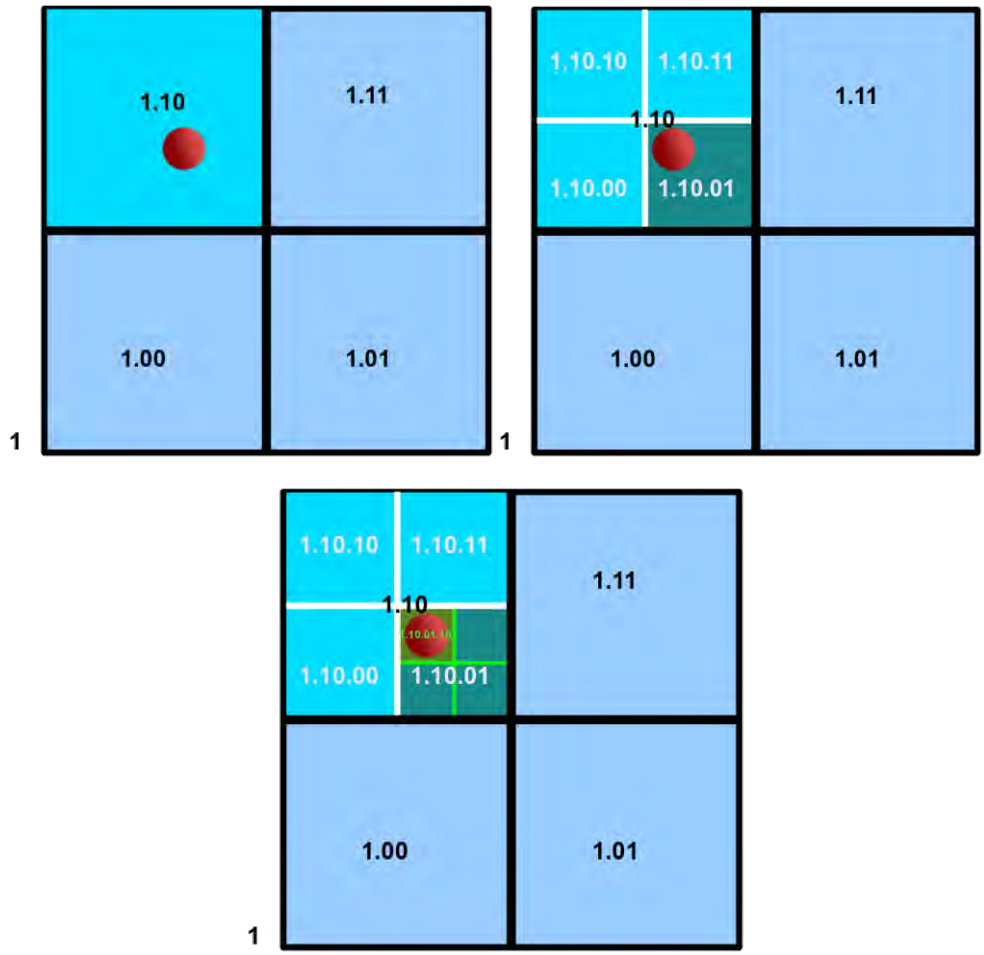

Figura 15 Relación entre la clave Morton de los nodos y las partículas.

Ejemplo, si buscamos generar al nivel 1 la clave Morton para el nodo 8 (111) se sigue el mismo proceso, la esquina inferior del nodo estará en (512, 512, 512):

Por lo que su llave Morton será igual a:

(512, 512, 512) = (1000000000, 1000000000, 1000000000) = 111 000 000 000 000 000 000 000 000 000 000

Agregamos los 2 bits más significativos

Llave Morton = 01 111 000 000 000 000 000 000 000 000 000

Dado que solo buscamos las partículas para este nodo a nivel 1 de profundidad ocupamos solo los primeros 4 bits: 01 111, este valor nos sirve de mascara para poder filtrar todas las partículas que estén dentro del espacio del nodo ya que cualquier partícula que este dentro del nodo tendrá los mismo 4 bits en su clave Morton.

Esto se cumple ya que el nodo con llave Morton 1 000, abarca de 0 a 256 en x y y z. Las partículas contenidas deben estar dentro de esa área así que ninguna partícula contenida en ese nodo tendrá alguna coordenada mayor a 256 y al momento de generar su llave Morton tendrá el bit más significativo en 1 pero los 3 siguientes nunca estarán en 1 (por el proceso al combinarlos y generar la llave).

3.4. Conversión de la llave Morton de un nodo a un índice dentro de un arreglo.

Para poder trasformar la llave Morton a un índice dentro de un arreglo, usaremos el valor correspondiente a las tripletas, sin tomar en cuenta los bits dedicados al padre, también depende del nivel al que nos encontremos ya que esto nos indicara la cantidad de tripletas que tomaremos en cuenta p.e. si nos encontramos a nivel 2 tomaremos solo las primeras 2 tripletas de izquierda a derecha (6 bits).

01.000.000.000.000.000.000.000.000.000.000

Estos bits nos representaran el índice del nodo a ese nivel. Para convertirlo a un índice lineal dentro de un solo arreglo que contiene todos los nodos a todos los niveles necesitamos ajustar la posición dependiendo del nivel al que se encuentre el nodo.

Como se mencionó previamente el nodo al nivel 0 ocupara la posición 0 del arreglo unidimensional, los 8 nodos del nivel 1 ocuparan la posición 1-8 para el nivel 2 que contiene 64 nodos se le asignara las posiciones 9-72 y así sucesivamente hasta el nivel deseado. Este crecimiento por nivel es en potencias de 8 donde en cada nivel de profundidad extra se agregan 8^nivel nodos extra por lo tanto, la posición del nodo en el arreglo estará descrita por una sumatoria de potencias de 8.

Para obtener el índice directamente se ocupara la siguiente fórmula:

Si el nivel es 0 el índice es 0, para cualquier otro nodo:

$$Indice = \left(\sum_{i=0}^{\frac{(L(MK))}{3}-1} 8^i \right) + V(MK) \qquad \text{(B.1)}$$

Donde MK es la llave Morton del nodo, y V(MK) obtiene el valor decimal de las tripletas usadas sin tomar en cuenta los bits usados para describir al padre. L(MK) da como resultado la cantidad de bits que se están usando (3 multiplicado por la cantidad de tripletas que se usa). En la división contenida en las sumatoria solo se tomara la parte entera del resultado.

Ejemplo:

Se tiene la llave Morton a nivel 2:

MK = 01 001 000 000 000 000 000 000 000 000 000

Se toman únicamente el valor de las primeras 2 tripletas (de izquierda a derecha) sin tomar en cuenta los 2 bits más significativos (bits para el nodo raíz).

MK = 01 001 000 000 000 000 000 000 000 000 000

V(MK) nos dará como resultado 8 que es el valor en decimal de 001000b y L(MK) nos dará 6. Usando la ecuación 1, tenemos:

$$Indice = \left(\sum_{i=0}^{(6)/3)-1} 8^i \right) + 8 = \left(\sum_{i=0}^{1} 8^i \right) + 8 = 8 + 1 + 8 = 17$$

Por medio del uso de esta fórmula podemos obtener la posición de cualquier nodo en el arreglo unidimensional, esto nos permitirá una implementación más rápida en una arquitectura en paralelo.

3.5. Conversión del índice dentro de un arreglo a llave Morton.

De manera inversa también es posible obtener la clave Morton para cualquier nodo en el arreglo unidimensional.

Para obtener a partir del índice de un arreglo su llave Morton debemos tomar en cuenta que entre más grande sea el valor del índice, a un nivel más profundo se encuentra el nodo con esa llave Morton. Para esta conversión ocuparemos las formulas siguientes:

$$Pos(indice) = Indice - \sum_{i=0}^{log_8(indice)} 8^i \quad \text{(B.2)}$$

$$(log_8(200) + 1) \quad \text{(B.3)}$$

La ecuación B.2 nos da el valor de las tripletas que representan la llave de Morton (sin tomar en cuenta los bits más significativos que corresponden al nodo raíz), una vez obtenido este valor se necesita saber la profundidad a la que se encuentra el nodo ya que esto nos indicara cuantas tripletas de bits usar para representar la clave Morton, esto se obtiene con la ecuación B.3, así sabremos cuantas tripletas usar para representar el valor obtenido en la ecuación, solo se agregan como último paso los 2 bits más significativos que corresponden al nodo raíz.

Ejemplo:

Índice = 200

$$Pos(200) = 200 - \sum_{i=0}^{log_8(200)} 8^i = 126$$

y

$(log_8(200) + 1)$ = 3

Por lo tanto sabemos que el nodo está a nivel 3 de profundidad con la posición 126 dentro de este nivel del árbol. El nodo se encuentra en profundidad 3 así que se usaran 9 bits (nivel 3, se usan 3 tripletas) para representar el 126 = 000 011 110, este es el valor de llave Morton sin tomar en cuenta los 2 bits más significativos. Al final queda:

MK = 01 000 011 110 000 000 000 000 000 000 000

Solo se agregaron los bits correspondientes al nodo raíz. Ahora en ese nivel de profundidad se busca el nodo con esos 10 bits de izquierda a derecha.

3.6. Lectura de las partículas y almacenamiento en memoria

La información perteneciente a todas las partículas es almacenada en un archivo de texto que describe en cada renglón una partícula.

Ejemplo de archivo para partículas:

30 21 220 222 21 90 0.5 0

10 122 100 1 1 1 0 0

130 126 4 4 4 4 0 0

50 127 15 67 45 15 0 0

200 128 23 130 145 23 0 0

En este ejemplo se describen 5 partículas con el siguiente formato: Los primero 3 valores, separados por un espacio cado uno, representan las coordenadas en el espacio de la partícula; los siguiente 3 valores representan el color para esa partícula; por último los 2 valores restantes representan los valores de reflexión y refracción respectivamente. Se usa un renglón por cada partícula y se debe respetar este formato, automáticamente se detectara la cantidad de partículas por lo que no es necesario describir más datos en el archivo.

Después de leer la información contenida en el archivo texto (el cual almacena todos los datos correspondientes a las partículas) se guardara toda la información en una

matriz donde cada fila de la matriz contiene información de una partícula que es equivalente a una línea en archivo leído.

Con la matriz generada se usa una estructura de datos para encapsular la información de una manera más práctica para el manejo de la misma dentro de la GPU, la estructura es declarada de la siguiente manera:

```
struct Particles
{
        float m_x;
        float m_y;
        float m_z;
        unsigned int morton_key;
        float r;
        float g;
        float b;
        int totalvecinos = 0;
        Particles * vecinos;
};
```

Esta estructura es inicialmente declarada dentro del flujo normal del programa para posteriormente ser transferido a memoria de GPU donde será procesado. Los elementos que componen a la estructura son 3 flotantes que guardan las coordenadas de la partícula en el espacio, un entero sin signo para almacenar la clave Morton, 3 flotantes para almacenar el color, un contador entero que determinar la cantidad de partículas vecinas y por ultimo un apuntador a los vecinos que pueda tener la partícula en un radio de búsqueda definido.

Con la definición de esta estructura se crea un arreglo de tamaño igual a la cantidad de partículas leídas en un archivo texto, se pasa la información de cada partícula a un elemento del arreglo para posteriormente trasferir la memoria de este arreglo a su equivalente en memoria de GPU. Con esto tenemos un arreglo de estructuras de dimensión igual a la cantidad de partículas existentes.

3.7. Construcción de la estructura de datos

Para que un nodo exista dentro de la estructura de datos es necesario que primero contenga información dentro de su espacio (partículas). Como se mencionó previamente la estructura de datos *Octree* es representada como un arreglo lineal de elementos donde cada elemento contiene un nodo y un contador de partículas para el nodo.

Como primer paso es necesario determinar que nodos deben existir y por lo tanto crearlos, para lograr esto es necesario filtrar las partículas y determinar que nodos las contienen. Esto se realiza implementando un kernel de CUDA el cual será ejecutado con tantos threads como partículas existan, cada thread se encargara de procesar una partícula.

Para procesar una partícula se usa la clave Morton la cual dependiendo de la cantidad de tripletas usadas corresponderá a la clave Morton de un nodo en diferentes profundidades del árbol, lo cual indicara que esta partícula es contenida por esos nodos. Se calcula la pertenencia de la partícula a todos los niveles del *octree*.

Ejemplo:

Se tiene un *Octree* de profundidad 5 y una clave Morton de una partícula: 01 000 010 010 000 000 000 000 000 000 000.

Se calcula a que nodo pertenece esta partícula a todos los niveles del *Octree*.

Para determinar si la partícula existe en el nodo padre del *Octree* solo basta revisar los 2 bits más significativos, los cuales deben ser 01 respectivamente si esto se cumple la partícula pertenece al espacio del *octree* y por lo tanto el nodo padre la contiene. Para los demás niveles del *octree* necesitamos buscar el índice del nodo que contiene a la partícula en cada nivel de profundidad.

Para calcular el índice del nodo que contiene a la partícula basta tomar la clave Morton de la partícula y convertirla a un índice lineal empleando la fórmula 1, usando la cantidad de bits correspondiente a la profundidad del árbol donde se está buscando el nodo que la contiene.

A profundidad 1 se toma los siguientes 3 bits después de los 2 bits más significativos

01 000 010 010 000 000 000 000 000 000 000

$$Indice = \left(\sum_{i=0}^{0} 8^i\right) + 0 = \left(\sum_{i=0}^{0} 8^i\right) + 0 = 1 + 0 = 1$$

A profundidad 2 se toman los siguiente 3 bits

01 000 010 010 000 000 000 000 000 000 000

$$Indice = \left(\sum_{i=0}^{1} 8^i\right) + 2 = 1 + 8 + 2 = 9 + 2 = 11$$

A profundidad 3

01 000 010 010 000 000 000 000 000 000 000

000 010 010b = 18d

$$Indice = \left(\sum_{i=0}^{2} 8^i\right) + 18 = 1 + 8 + 64 + 18 = 90$$

Para las demás profundidades se sigue el mismo proceso, obteniendo:

Nivel 4 = 729

Nivel 5 = 5833

Estos valores: 0, 1, 11, 90, 729, 5833 corresponden a las posiciones en el arreglo de los nodos que contienen a la partícula. De esta manera podemos acceder directamente a este nodo en el arreglo y asignarle una partícula más a los respectivos contadores. Se realiza este procedimiento para todas las partículas,

dado que se realiza dentro de un kernel el trabajo se realizara en paralelo y no en serie.

Una vez procesadas todas las partículas se tiene la estructura de datos con punteros vacíos (los nodos aun no existen) pero sus contadores fueron modificados por el procesamiento de las partículas. Con esto es posible construir un kernel que lance un thread por elemento del arreglo unidimensional, cada thread procesara un elemento del arreglo y si el contador es mayor a 0 se crea el nodo asignando memoria al puntero actualmente nulo.

Ya que todos los nodos que tienen información dentro de ellos fueron instanciados, se pasa a procesarlos y calcular todos los datos necesarios para el nodo, el diagrama 1 describe la creación del nodo como tal.

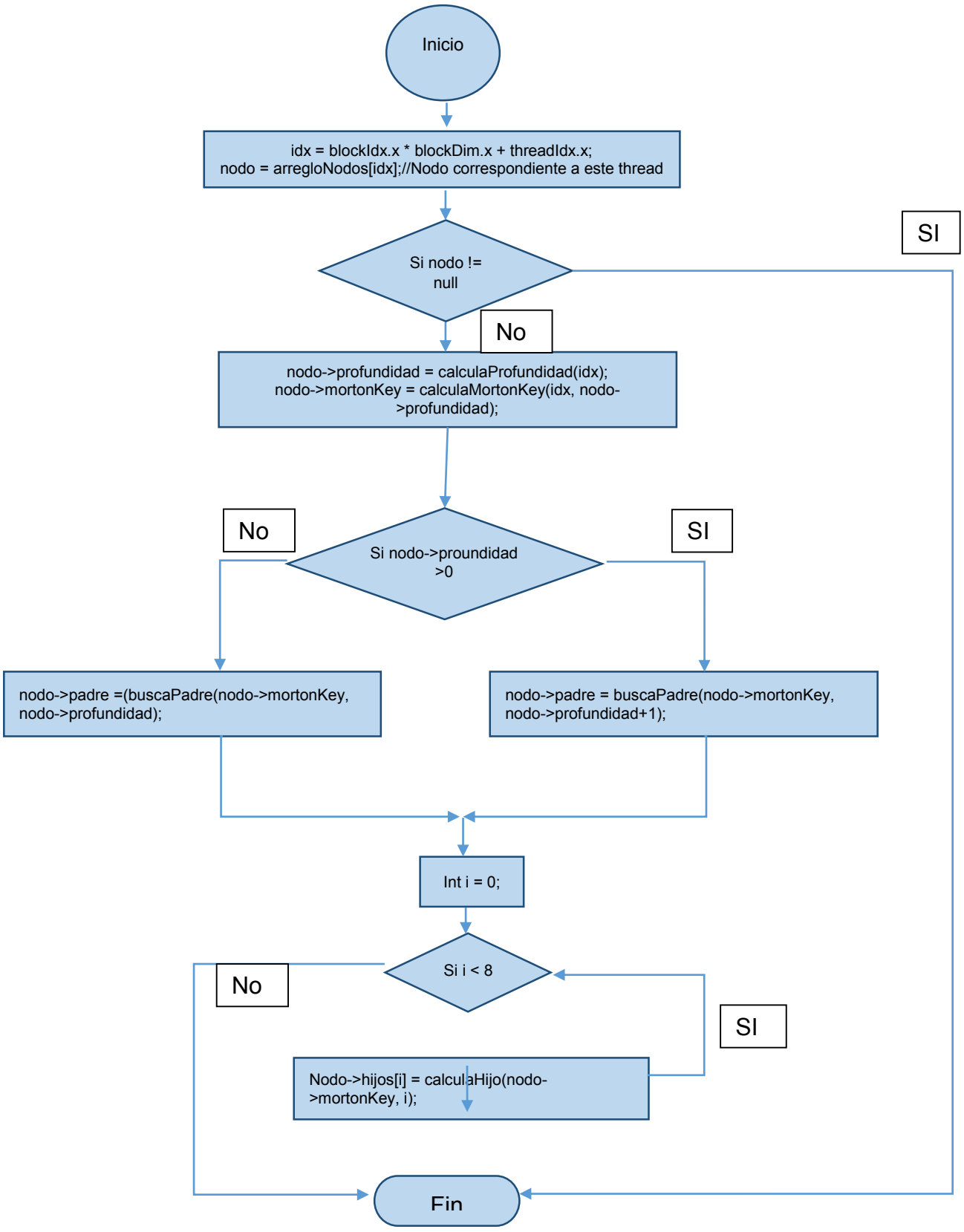

Diagrama 1. Creación de un nodo en GPU.

Terminado el kernel de procesamiento de nodos, todos los nodos son creados correctamente y todos los datos que describen al nodo son asignados incluyendo apuntadores (los apuntadores a nodos padres e hijos están apuntando a sus respectivos nodos).

3.8. Implementación de ray tracing en paralelo

Para la implementación de ray tracing (descrita en la sección 3.5) usando CUDA se realizaron algunos kernels y múltiples funciones para diferentes aspectos al generar la imagen. Una de las características de la técnica de *ray tracing* es el uso de recursividad para recalcular los rayos o en nuestro caso navegar sobre nuestra estructura de datos *octree*. La recursividad es un recurso que no se puede usar en GPUs salvo algunas excepciones (los modelos más actuales de GPUS), por esta razón se decidió diseñar un algoritmo que no ocupara recursividad.

Cada pixel de la imagen a generar será procesado de manera independiente en un kernel de CUDA. Cuando el algoritmo de *ray tracing* inicia se establece un punto de origen inicial y un tamaño de pantalla de imagen (esto determina la cantidad de pixeles y por lo tanto la cantidad de threads con los cuales será ejecutado el kernel). Cada thread procesara un pixel de la pantalla iniciando por calcular la posición del pixel en la pantalla, esto se realiza usando el punto de origen de la cámara menos la mitad de la pantalla en *x* y *y* lo cual nos posicionara en la esquina de la pantalla a esto le sumamos el número del thread multiplicado por un factor de escalamiento el cual nos dará al final el tamaño de la pantalla en unidades del mundo. En el ejemplo siguiente el tamaño de la pantalla es de 800 por 600, con tamaño en coordenadas del mundo de 8 x 6 unidades por lo tanto el factor de escalamiento es igual a 0.01 (8/800 y 6/600). Para la posición en el eje Z se usa el punto de origen de la cámara más la distancia focal lo cual nos posiciones en el plano de la imagen.

```
globalTidY = threadIdx.y + blockIdx.y*blockDim.y;
globalTidX = threadIdx.x + blockIdx.x*blockDim.x;
xInf = 8/2;
yInf = 6/2;
dest.x = orig.x - xInf + globalTidX * 0.01f;
```

```
dest.y = orig.y - ylnf + globalTidY * 0.01f;
dest.z = orig.z + focalDist;
```

Después de este proceso se tiene la posición de cada pixel en el plano de la imagen en coordenadas del mundo se puede calcular el rayo que es proyectado por cada pixel. Para definir el rayo que se maneja como un vector necesitamos un punto y una dirección, el origen de todos los rayos proyectados por cada pixel es el origen de la cámara y dado que tenemos la posición del pixel en coordenadas del mundo podemos calcular la dirección de cada rayo que pasa por dicho pixel. Para realizar esto se realiza la resta de la posición del pixel menos el origen de la cámara en cada uno de los ejes, lo cual nos da un vector de dirección el cual se normaliza.

```
vector3 dir; //Dirección del vector
dir.x = dest.x - orig.x;
dir.y = dest.y - orig.y;
dir.z = dest.z - orig.z;
//Normalizando vector
aux = 1 / sqrtf(dir.x *dir.x + dir.y*dir.y + dir.z*dir.z);
dir.x *= aux;
dir.y *= aux;
dir.z *= aux;
```

Con esto tenemos el rayo descrito y podemos realizar el *ray tracing* sobre la estructuras de datos *octree*, dado que no se utilizara recursividad se necesita una manera de almacenar el camino que recorre el rayo por la estructura de datos para esto se crea una pila de nodos la cual va agregando nodos intersectados por el rayo y nos presenta una descripción del camino del rayo. Este proceso será descrito con detalle más adelante en esta sección.

Para detectar las intersecciones con los voxeles dentro del *octree* se utiliza la técnica basada en *slabs* desarrollada por Kay and Kayjia (Kay y Kajiya 1986) donde un *slab* (losa) se considera al espacio entre 2 planos que son paralelos. Esta técnica busca la intersección del rayo entre los 3 *slabs* que componen a un voxel, si el rayo atraviesa los 3 *slabs* al mismo tiempo entonces el rayo choca con el voxel ya que

el rayo en algún momento de su trayectoria se encontró dentro de los 3 *slabs* simultáneamente.

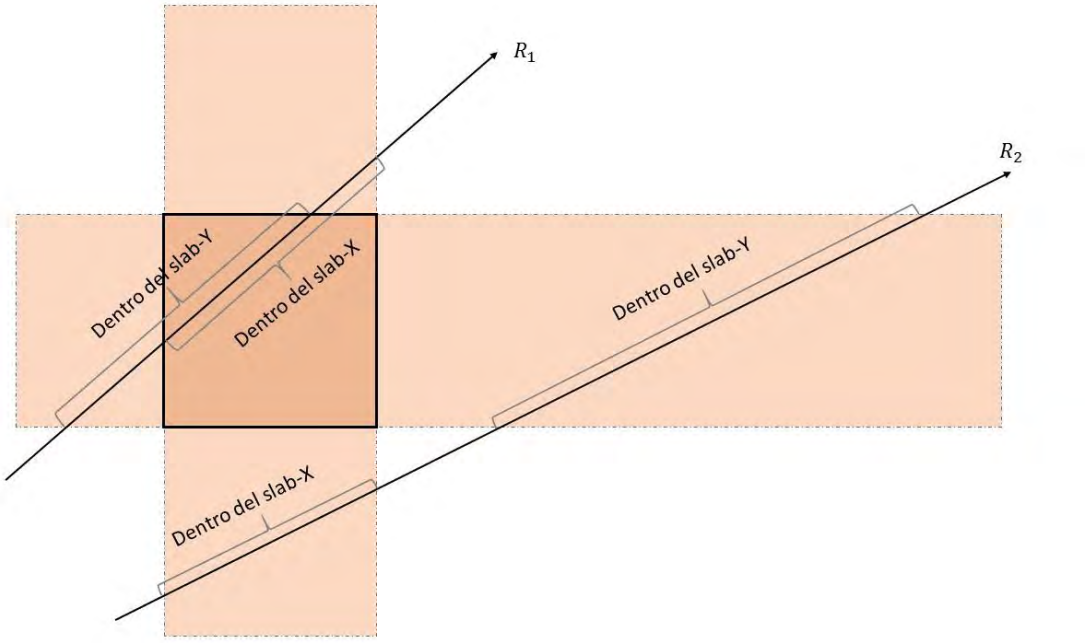

Figura 16 Ejemplo de intersección de slabs en 2 dimensiones (x, y).

Para poder determinar la intersección de un rayo contra un voxel se necesitara calcular los intervalos de intersección con cada uno de los pares de planos que conforman los *slabs*. Todas las intersección son necesarias para determinar las distancias a las que intersectan y con estas determinar si existe intersección simultanea entre los *slabs*. Si el punto de entrada más lejano de un *slab* se encuentra alguna vez mas lejos que la salida más cercana, entonces rayo no intersecta con el voxel y se puede finalizar la ejecución del algoritmo. El algoritmo tomado como base para implementación en esta tesis fue tomado del libro escrito por Christer Ericson (Ericson 2005, pag. 179).

La forma en que los intervalos de intersección son obtenidos es intersectando la ecuación paramétrica del rayo R(t) = P + t**d** en las ecuaciones de los planos paralelos que componen a cada *slab*, $R * n_i = d_i$ y resolviendo la ecuación para t, dado $t = (d - P * n_i)/(\mathbf{d} * n_i)$. Donde P se refiera al punto de origen del rayo P =

(p_x, p_y, p_z) y d la dirección del rayo **d=** (d_x, d_y, d_z), en el caso de los slabs en el eje X los planos son perpendiculares y por lo tanto las normales tienen 2 componentes con valor igual a cero y el componente en x igual a 1 por esto podemos simplificar la ecuación a $t = (d - p_x)/(\mathbf{d}_x)$. Donde *d* se refiere a la posición del plano en el eje x. Uno de los problemas que se pueden presentar es cuando la dirección del rayo sea igual 0 en el eje que se esté procesando, para evitar un error este caso es tratado por separado y simplemente se revisa que el punto de origen en el eje a procesar se encuentre dentro del *slab*. Para los propósitos de esta tesis se utilizó el siguiente algoritmo.

```
__device__ bool intersectaVoxel(vector3 *orig, vector3 *dir, OctreeNodo *nodo)
{
        vector3 limiteSup; //límite superior que define al cubo
        vector3 limiteInf; //límite inferior que define al cubo
        float t1, t2; ///dos posible intersecciones una por plano

        ///Se calculan las esquinas inferior y superior del nodo, con esto sabemos
        los //límites del espacio de cada nodo.
        limiteSup = calculaEsquinaSup(nodo->mortonKey, nodo->depth) ;
        limiteInf = calculaEsquinaInf(nodo->mortonKey, nodo->depth);

        //Probamos en los 3 ejes (x, y, z)
        for(int i =0; i < 3 ; i++)
        {
                if (dir[i] == 0) //El rayo es paralelo a los planos
                {
                        if (orig[i] < limiteInf[i] || orig[i] > limiteSup[i])
                                return false;
                }
                else
                {
                  //limite inferior
```

```
                t1 = (limiteInf[i] - orig[i]) / dir[i];
                //limite sup
                t2 = (limiteSup[i] - orig[i]) / dir[i];
              //En caso de que la distancia al límite inferior sea mayor que la
        superior las cambiamos.
                if (t1 > t2)
                {
                        float aux = t1;
                        t1 = t2;
                        t2 = aux;
                }
                //plano más cercano a la cámara, t1 mayor que la distancia más
cercana
                if (t1 > distNear)
                {
                        distNear = t1;
                }
                //plano más lejano a la cámara
                if (t2 < distFar)//t2 menor que la distancia más lejana
                        distFar = t2;
                if (distNear > distFar) //el rayo no intersecta en el voxel
                        return false;
        }
}
```

Una vez ejecutado este algoritmo se obtiene la distancia a la que se entra al voxel y la distancia a la que el rayo sale. Si se desea obtener el punto de intersección exacto a la que el rayo choca con el voxel se puede usar la distancia y sustituir en la ecuación del rayo para obtener el punto de intersección usando la siguiente formula.

$$q = orig + dir * distNear. \qquad (B.4)$$

Para determinar el camino que sigue un rayo, se hace la prueba de intersección con los nodos por nivel y todos los nodos alcanzados son almacenados en una pila para después ordenarlos por distancia (del más cercano al más lejano usando la ecuación B4 para obtener la distancia). A nivel 1 se prueba la intersección del rayo con los 8 hijos del nodo padre, los nodos alcanzados por el rayo a nivel 1 son almacenados en una pila y luego ordenados. Una vez ordenados se saca el primer nodo de la pila (que debe ser el que obtuvo la intersección más cercana con el rayo) y se hace la prueba de intersección con sus 8 hijos para después ordenar los nodos intersectados y almacenarlos en la pila de nodos. Esto se realiza hasta que no existan más nodos en la pila o que se encuentre el nodo más cercano en el último nivel de profundidad.

3.8.1. Visualización, iluminación y color

Para la iluminación y el color se utilizan diferentes técnicas y métodos para asignar diferentes tipos de valores a los pixeles que proyectan la imagen. Cuando los voxeles son creados es posible asignarles un color dependiendo de las partículas que contienen, para esto cuando las partículas son filtradas y se determina cual es el nodo que la contiene a las diferentes profundidades del *octree*, se suman los valores de color de cada partícula contenida en los nodos y se promedia con lo cual nos da un color único para cada nodo y que depende de la profundidad del *octree*, si la profundidad es suficiente como para que cada nodo contenga a una única partícula el color de los nodos será igual al de la partícula. Esto al realizar *ray casting* sobre el *octree* nos da un color sólido, pero sin propiedades de profundidad en caso de imágenes con un color sólido, figura 16.

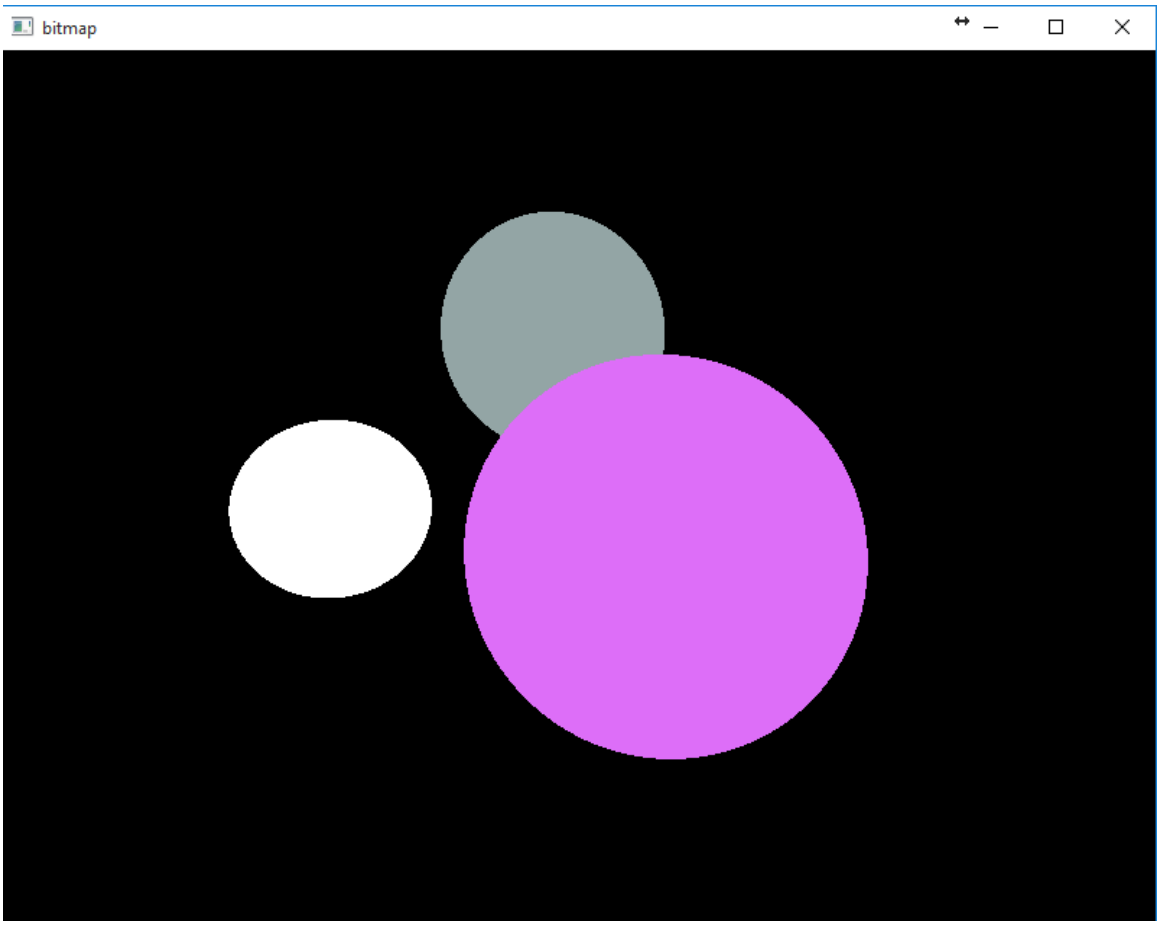

Figura 17 Imagen generada únicamente realizando ray casting sobre los objetos. Se genera un color solido

.

Para poder tener una pista de profundidad es posible usar el eje Z como referencia al color asignado, obscureciendo el color conforme más lejos se encuentre en la coordenada Z el nodo. Se utiliza un regla simple donde el tamaño del *octree* en el eje Z representa la referencia para la escala de color, si el nodo se encuentra en el borde del *octree* entonces será casi negro mientras que los que se encuentre en el borde frontal serán de tono casi blanco.

Este método da una percepción de profundidad en la imagen y permite ver los cambios en la superficie del volumen. Es posible mapear directamente un mapa de profundidad como puntos en el espacio, por ejemplo la siguiente imagen en la figura 17 es un mapa de profundidad

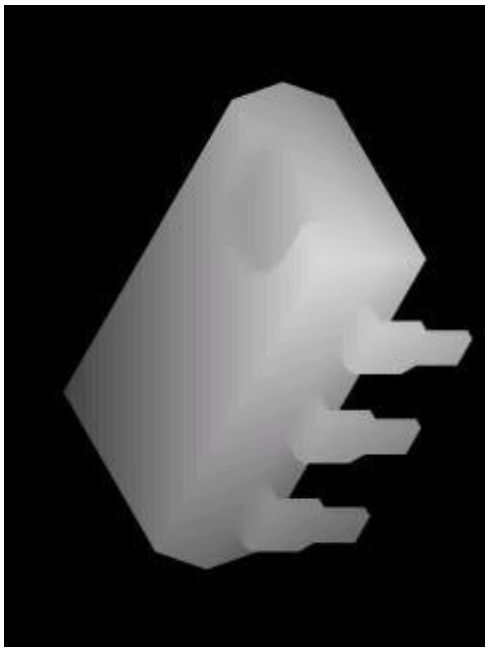

Figura 18 Mapa de profundidad.

Esta imagen de profundidad es mapeada a una nube de puntos donde cada pixel de la imagen es un punto en el espacio. Se usan como dimensiones del volumen el tamaño de la imagen, para el caso de la profundidad se asigna como coordenada en z el tono de gris del pixel. Los tonos completamente negros son omitidos ya que no contienen información y representan el fondo de la imagen. Con esta nube de puntos generada podemos construir un volumen y visualizarlo, figura 19. El volumen visualizado contiene algunas imperfecciones debido a la perspectiva desde la que se observa ya que la cámara debe ser posicionada exactamente en la misma posición de la imagen original donde fue generado el volumen.

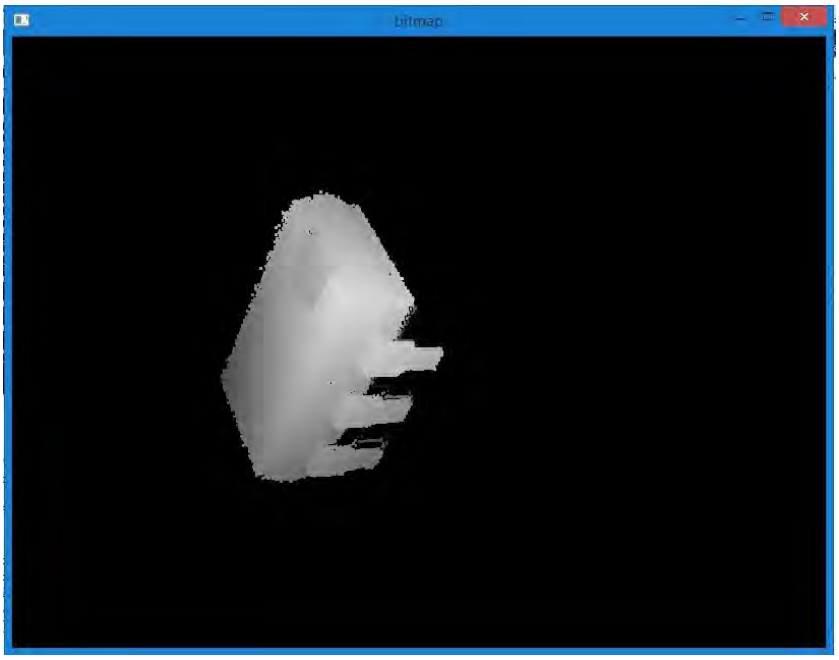

Figura 19 Visualización del mapa de profundidad, realizando ray casting sobre el volumen.

Las representaciones anteriores son visualizaciones utilizando solo el color de las partículas o asignando una intensidad de gris que indique la posición de la partícula en el volumen.

Alternativamente a usar solo el color de las partículas se implementó el modelo de iluminación de Phong que permite el uso de puntos de iluminación que crearan sombras y reflexiones sobre las superficies.

El modelo de iluminación de Phong es un modelo empírico que permite calcular como la luz es reflejada por las superficies dependiendo de diferentes propiedades asignadas a la superficie. Cada partícula que compone la nube de puntos tiene asignado un valor de reflexión el cual indicara cuanta luz refleja la partícula, es posible promediar este factor dependiendo de la cantidad de partículas contenidas en cada nodo del *octree* y de esta manera obtener un valor de reflexión propio para cada nodo.

Para poder calcular los patrones de iluminación es necesario primero describir como viajan los diferentes tipos de rayos a través del *octree*.

La imagen mostrada en la figura 19 muestra los rayos primarios que son proyectados por el plano de la cámara hacia los diferentes objetos en la escena, de estos rayos proyectados existen algunos que se van al infinito y nunca interceptan con ningún objeto, otros interceptan directamente con un punto de iluminación. Por otra parte los rayos que chocan con algún objeto proyectan el color del objeto como se mostró en los ejemplos anteriores usando solo una escala de grises para mostrar los objetos.

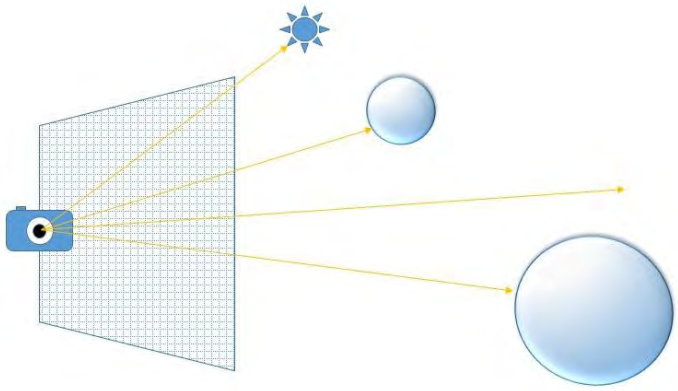

Figura 20 Diferentes tipos de rayos primarios proyectados por la cámara a través del plano de la imagen.

Existen otros rayos que son creados como consecuencia de la propagación de los rayos primarios (figura 20), diferentes factores de las superficies dan como resultado estos rayos, la imagen 21 muestra algunos de ellos.

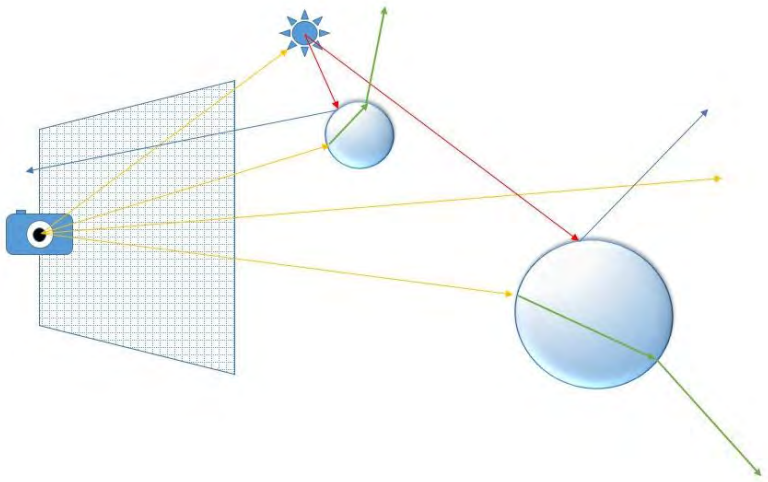

Figura 21. Rayos creados a partir de los rayos primarios.

En la imagen de la figura 20 las líneas de color amarillo corresponden a los rayos primarios que son lanzados por la cámara hacia el espacio. Las líneas azules representan los rayos reflejados, estos rayos simplemente chocan con una superficie y se reflejan de acuerdo al ángulo de incidencia, la intensidad con la que el rayo es reflejado también depende del factor de reflexión de la superficie.

Las líneas verdes son rayos refractados, estos rayos involucran un factor de refracción que indica como los rayos atraviesan los objetos como puede ser agua u objetos semitransparentes, son calculados usando la ley de Snell.

Por otro lado las líneas rojas son rayos que son usados para probar sobre una fuente de luz, esto con el propósito de calcular sombras.

Si se sigue una línea amarilla (rayo primario) que inicia su trayectoria en la cámara, se puede observar que cada rayo puede dar como resultado muchos otros rayos secundarios: Rayo reflejado, un rayo refractado y un rayo de sombreado por cada fuente de luz que exista en la escena. Cuando estos rayos son lanzados, cada uno de ellos es tratado como un rayo normal (excepto el rayo de sombra). Uno de los problemas que se presentan es que los rayos pueden ser reflejados un número indefinido de veces y es necesario seguir la trayectoria del mismo, esto es llamado *ray tracing* recursivo. Cada rayo que se lanza agrega color al pixel de la pantalla

que lanzo el rayo original, y finalmente construye un color final de acuerdo a todas las propiedades de los objetos. Esta recursividad es muy costosa computacionalmente hablando y depende directamente de la recursividad por lo que es mejor establecer un límite para las reflexiones posibles.

El modelo de iluminación de Phong comprende 3 componentes que darán como resultado la iluminación final, figura 22.

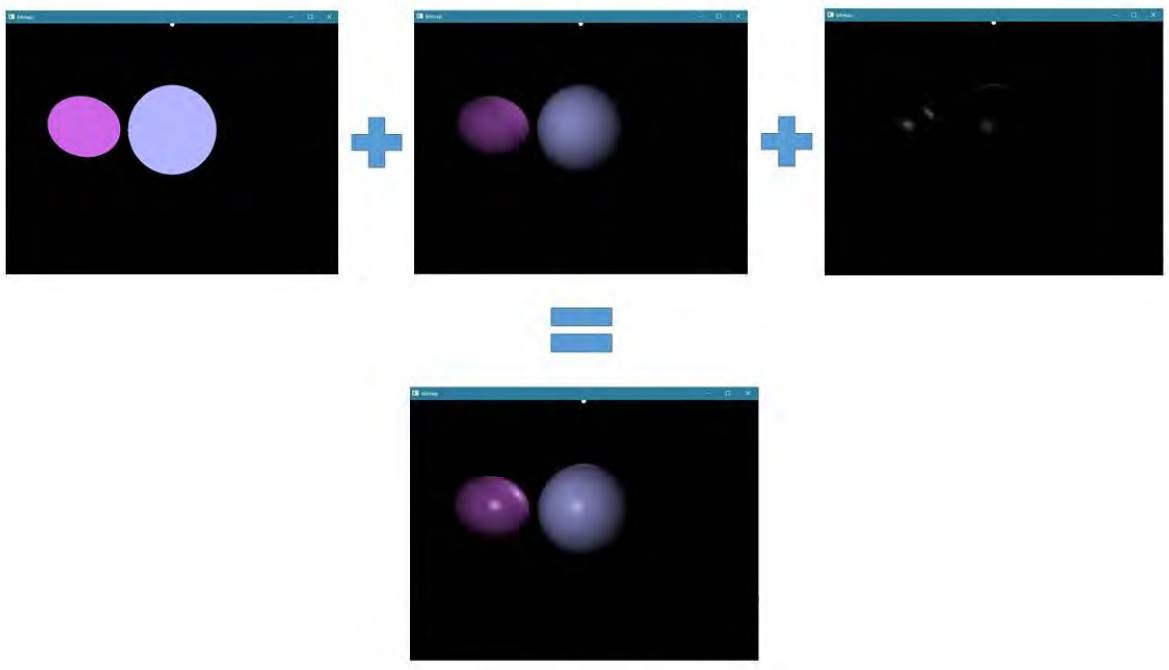

Figura 22. Modelo de iluminación de Phong

La iluminación ambiental se utiliza para simular el efecto "radiante" en la iluminación, es decir, el efecto de la luz que está reflejándose en el medio ambiente.

La iluminación difusa se utiliza para simular la reflexión de luz desde una superficie donde el rebote de la luz es igual en todas las direcciones. La reflexión difusa es la reflexión de la luz desde una superficie donde el rayo de luz que choca contra la superficie es reflejado en muchos ángulos en lugar de solo uno como el caso de la reflexión especular.

Una superficie "sin brillo" que refleja todo lo que le llega (como la nieve, o una tabla de madera, por ejemplo) tendría un valor muy alto de "difuso" para su valor de iluminación. Una superficie difusa emite luz en “todas” las direcciones cada vez es golpeada por un rayo de luz, pero la cantidad de luz que emite es controlada por el ángulo en que la luz llega a la superficie.

Al hablar de reflexión especular significa que los rayos de luz que chocan contra la superficie se reflejan exactamente en un ángulo igual al ángulo formado por el rayo que choca y la superficie de impacto, por esto es también conocida como reflexión en espejo. La reflexión especular es una propiedad que se observa en las superficies brillantes como lo son objetos metálicos.

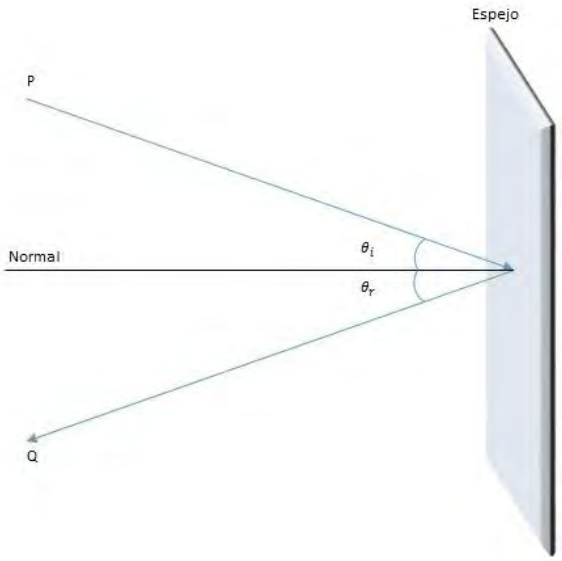

Figura 23 Reflexión de un rayo que choca contra una superficie en este caso un espejo.

3.8.2. Reflexiones

Para que un rayo pueda ser reflejado por una superficie es necesario conocer su normal, si esta normal es conocida se utiliza la siguiente formula:

$$\vec{R} = \vec{V} - 2 * (\vec{V} \cdot \vec{N}) * \vec{N}$$

4.

Donde el vector $\vec{R}$ corresponde al vector (rayo) reflejado por la superficie, $\vec{V}$ es el vector original que choca con la superficie y $\vec{N}$ corresponde a la normal de la superficie.

Aplicando esta fórmula podemos obtener todos los rayos reflejados, también es posible probar las fuentes de iluminación y crear sombras.

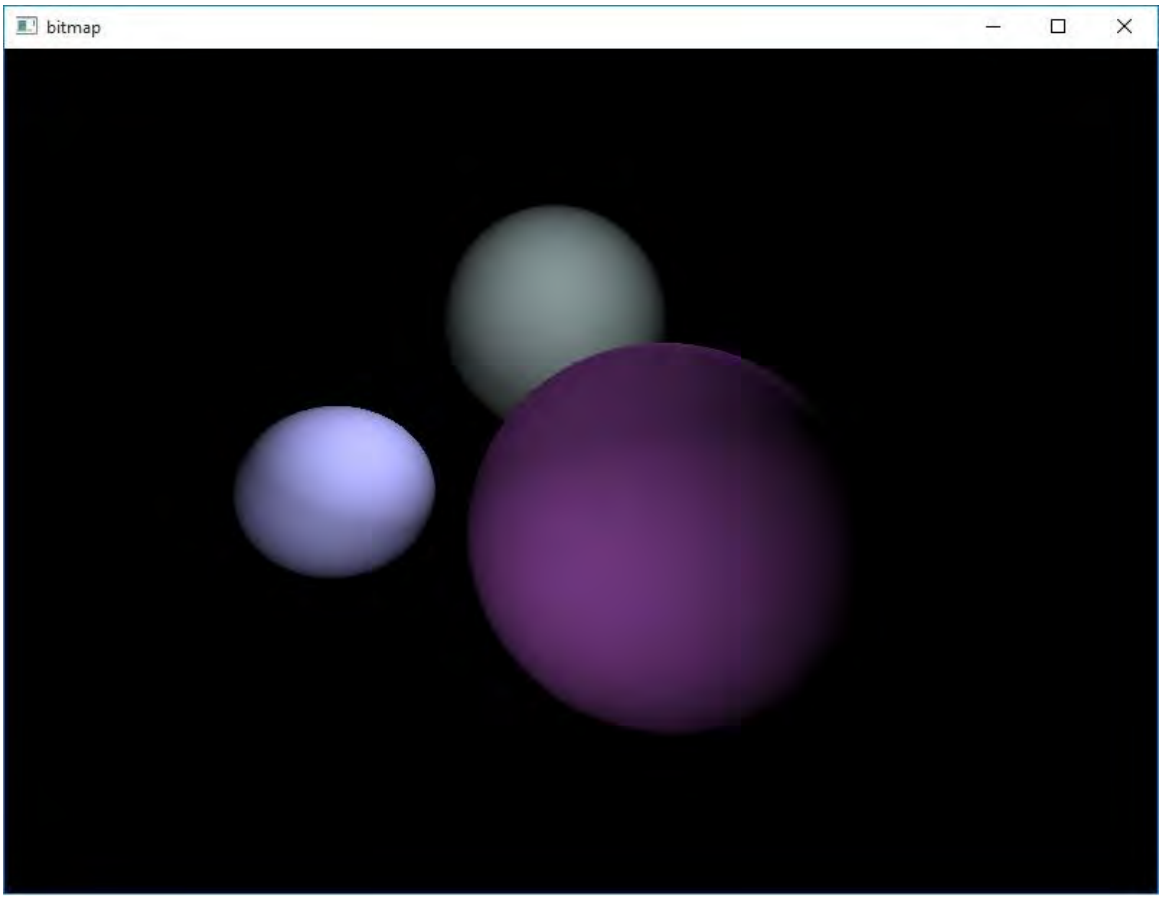

Figura 24. Imagen generada con iluminación difusa.

La imagen generada en la figura 23 tiene un solo punto de iluminación en la parte frontal pero la luz es esparcida por todo el objeto dependiendo de la distancia a la

que se encuentre el punto de intersección (solo cuenta con iluminación ambiental y difusa).

Para el caso de objetos brillantes es necesaria la iluminación especular, donde podremos tener más control sobre la intensidad de la iluminación, esto significa calcular y tomar en cuenta el vector reflejado de la luz por la superficie (no solo usarlo como referencia de distancia para calcular un color).

El modelos de iluminación de Phong usa la siguiente formula donde se toma en cuenta el vector reflejado.

$$Intensidad = diffuse \times (L \cdot N) + specular \times (V \cdot R)\text{^}n \qquad \text{B.5.}$$

Donde L es el vector de un rayo de luz que intersecta en la superficie, N es la normal de la superficie, V es el vector de dirección, R es L reflejado en la superficie y n es un exponente.

Aplicando esta fórmula podemos obtener una nueva imagen donde se observa claramente el punto de iluminación que se encuentra al frente del objeto, figura 24.

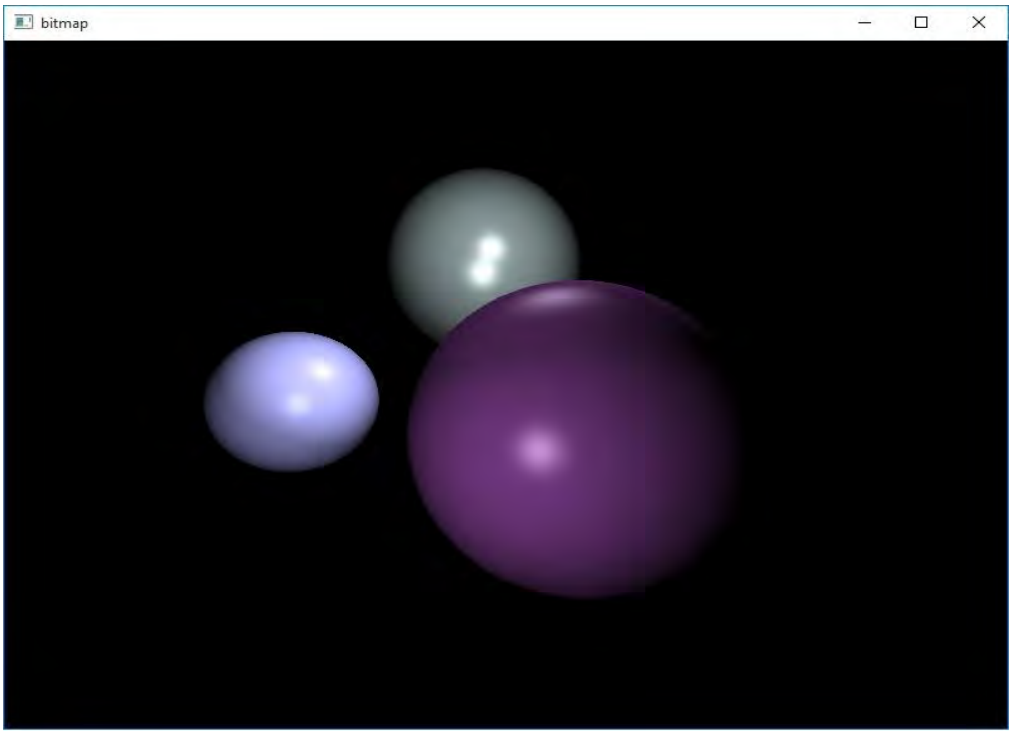

Figura 25. Imagen generada utilizando el modelo de iluminación de Phong (iluminación ambiental, difusa y especular)

Capítulo 4. Actualización del Octree

Gracias a la estructura en la que está constituida el octree es posible un mejor manejo de la memoria y la información que contiene. Cuando las partículas se mueven a través del espacio abarcado por el octree es necesario modificar la estructura de memoria, liberando la memoria ocupada por nodos que ya no contengan partículas y creando nodos en los cuales las partículas ingresen a su espacio. Para lograr esto se emplea la ventaja que se tienen usando una estructura lineal de nodo y un mapeo directo de la clave Morton a cualquier posición del arreglo.

Cuando una partícula cambia su posición en el espacio definido por el *octree* su clave Morton cambia en proporción a su movimiento, cuando una partícula abandona el espacio perteneciente a un nodo, su clave Morton dejara de coincidir con la clave Morton del nodo que contenía a la partícula a la profundidad del nodo (El espacio definido por el nodo depende de la profundidad a la que se encuentre el nodo).

Para comparar las claves Morton de la partícula y los nodos se utiliza la profundidad a la que se encuentra el nodo con el que se está comparando para saber la cantidad de bits a comprar. La figura 26 muestra como las coordenadas de las partículas son utilizadas para generar su clave Morton dependiendo de la representación en binarios de sus coordenadas en el espacio, si una partícula se mueve de posición sus coordenadas cambian. El cambio en su clave Morton depende del desplazamiento de la partícula, a mayor desplazamiento la clave Morton cambiara en sus bits más significativos y si el desplazamiento es pequeño (en relación al tamaño del *octree*) solo se verán afectados sus bits menos significativos. Al momento de comparar claves Morton entre partículas y nodos, se usan la cantidad de bits dependiendo de la profundidad a la que se encuentre el nodo que se desea comparar.

En la figura 26 (2) se puede ver que si se desea saber si la partícula salió del nodo en el que se encuentra (1.10) a profundidad 1 sus bits más significativos en su clave Morton cambiaran, si la partícula se moviera al nodo de la izquierda sus bits más significativos serán 1.11, los mismos que el nodo a esa profundidad. Entre más profundos se muevan por el *octree* más pequeños tendrán que ser los movimientos para que la partícula cambie de nodo, la figura 26 (4) muestra la clave Morton de la partícula que es igual a 1.10.01.10 (la misma clave Morton corresponde al nodo que la contiene). Si la partícula se moviera una unidad en el eje x sus nuevas coordenadas serian (3,5) y su nueva Morton seria 1.10.01.11 la cual corresponde al nodo que se encuentra a la derecha del que la contenía previamente. En este caso en particular 3 niveles de profundidad son suficiente para detectar movimientos unitarios de las partículas.

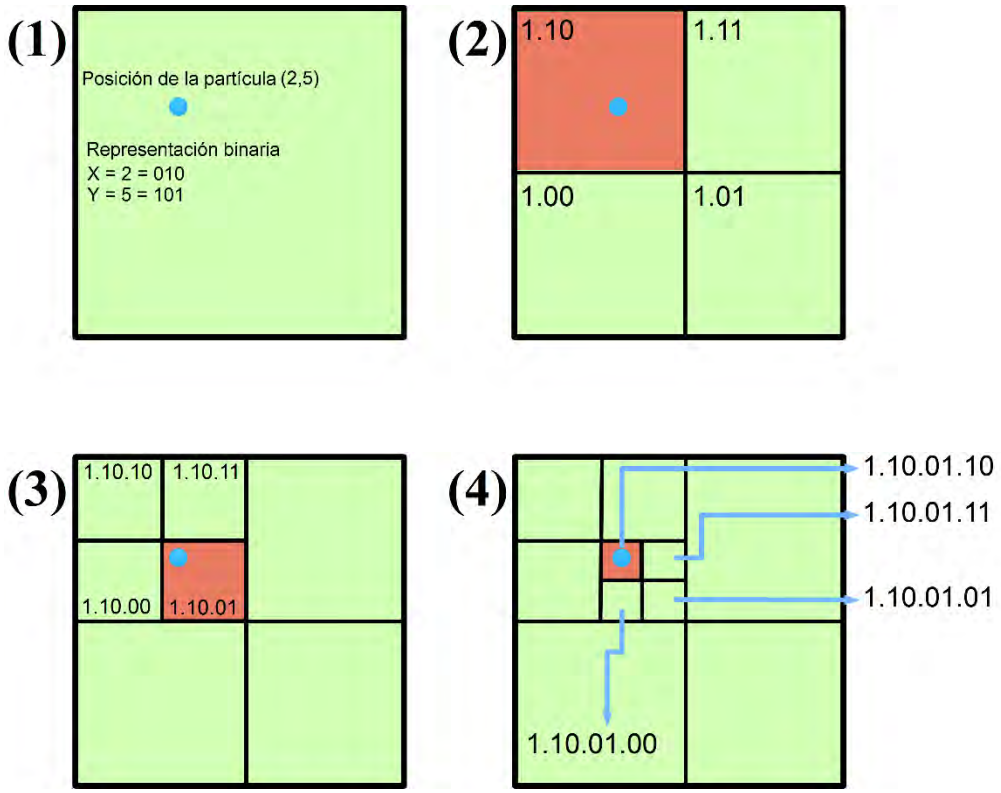

Figura 26 Generación de las claves Morton para las partículas dependiendo de la posición en el octree y la profundidad a la que se está comparando.

El proceso de creación y liberación de memoria para los nodos se realiza dentro de un kernel donde cada thread revisara las partículas contenidas por el *octree* y si estas cambiaron de posición lo suficiente como para significar que están contenidas

en un nuevo nodo. El proceso se realiza usando el algoritmo que se muestra a continuación.

```
void actualizaOctree(arregloNodos *arbol, Particles particula)
{
   //calculamos la Morton de esta partícula
   unsigned int tempMorton = generaMKParticula(particula);

   //Revisamos el cambio de la clave Morton en la partícula
   for (int i = 0; i <= profundidadMaxima; i++)
   {
        //Comparamos la clave Morton a la profundidad actual
        if (comparaMortons(points[idx].morton_key, tempMorton, i))
        {
            //Calculamos los índices del nodo que contenía a la partícula y del
            //nodo que ahora la contiene
          int indice_old, indice_new, aux;
          indice_old = mortonToArrayIndex(particula.morton_key, i);
          indice_new = mortonToArrayIndex(tempMorton, i);

          //le restamos al antiguo nodo una partícula
          sub(arbol[indice_old].contador);

          if (arbol[indice_old].contador == 0)
          {
                //borramos el nodo
                borraNodo(arbol[indice_old].nodo);
        }
          //aumentamos una partícula al nuevo nodo
          add(arbol[indice_new].contador);
        if (arbol[indice_new].contador == 1)
          {
              //Creamos un nuevo nodo
                creaNodo(arbol[indice_new].nodo);
          }
        }
   }
   //Asignamos las nueva clave Morton
   points[idx].morton_key = tempMorton;
}
```

El algoritmo anterior se ejecuta en paralelo para todas las partículas contenidas. Cada thread dentro del kernel ejecutara su propia versión del código y procesara una partícula. Cuando las partículas se mueven su posición en el espacio cambia, pero su clave Morton no es actualizada inmediatamente, por esto cuando el algoritmo inicia se calcula la clave Morton actual en base a las coordenadas de la partícula esto nos dará una nueva clave Morton si las coordenadas de la partícula fueron modificadas. Se inicia un ciclo que se ejecuta tantas veces como la profundidad máxima del *octree*, iniciando por el nodo padre. Se compara la clave Morton a la profundidad que se está revisando por lo que solo se comparan una cantidad determinada de bits, para la profundidad 1 solo se revisa los 3 primeros bits (sin tomar en cuenta los bits que definen al nodo padre 01), si estos 3 bits cambiaron con respecto a la Morton anterior registrada en la partícula quiere decir que la partícula abandono un nodo a esta profundidad y entro en uno nuevo por lo que es necesario ajustar el *octree*.

Para saber el nodo en el que se encuentra la partícula y el nodo en el que va a entrar, se calcula el índice del nodo dentro del arreglo usando únicamente su clave Morton y la profundidad que se está revisando, esto nos da dos índices: el del nodo antiguo que contenía la partícula (usando la clave Morton que tiene la partícula) y el nodo al que está ingresando (usando la clave Morton que se acaba de calcular).
Cuando una partícula abandona un nodo se procede a restar una partícula al contador de partículas del nodo que la contenía y aumentar el contador del nodo al que ingresa la partícula. Cabe mencionar que esto se realiza con una operación atómica ya que existe la posibilidad de que dos partículas abandonen un nodo al mismo tiempo y dado que todo el proceso se realiza en paralelo puede representar un riesgo de pérdida de datos. Si dos threads intentan modificar una variable al mismo tiempo toman el valor contenido por la variable y lo modifican para después asignarlo de nuevo, pero ya que ambos threads accedieron a la variable al mismo tiempo, tomaron el mismo valor y lo modificaron sin tomar en cuenta la modificación efectuada por el otro thread, al final el valor que queda en la variable será el del

ultimo thread en guardar su valor modificado y con esto se omite la operación de uno de los dos threads por lo que se pierde información.

Cuando se utiliza una operación atómica en un kernel de CUDA el valor de la variable sobre la que se está operando es bloqueado para los demás threads que deseen modificarlo y estos deberán esperar por su turno para modificar la variable, con esto se asegura que no exista perdida de información.

Ya que el contador del nodo al que pertenecía la partícula fue modificado, se verifica el valor del contador y en caso de que éste sea 0, indica que este nodo no tiene información para el *octree* y es necesario borrarlo, se elimina primero todos los punteros que puedan hacer referencia a este nodo asignándolos a nulo y posteriormente se libera la memoria ocupada por el nodo.

Cuando la partícula ingresa a un nodo también se debe modificar su contador, aumentando una partícula (también este procedimiento es realizado con una operación atómica), si el contador después de aumentarlo es igual a 1 indica que el nodo no existe aún y que es la primera partícula que entra a su espacio. Para crear un nuevo nodo es necesario primero que nada asignar memoria al puntero del nodo, después se asignan dato como su clave Morton, un puntero al padre y a los hijos, un puntero de su padre y su profundidad. Al finalizar este proceso ya se tiene ajustada la estructura del *octree* y se procede a revisar la siguiente profundidad del *octree*. El proceso de ajuste en el *octree* se realiza de manera descendente con respecto a la profundidad ya que de esta manera se evita problemas de acceso de memoria en caso de que se busque asignar punteros a nodos que aún no existen o usar la información del mismo.

Al terminar de revisar todas las profundidades para la partícula se asigna la nueva clave

Morton a la partícula para que en caso de que sus coordenadas sean modificadas de nuevo la comparación entre claves Morton detecte un cambio de nodos.

Capítulo 5. Resultados

5.1. Velocidad y desempeño.

Se realizó la comparación de la implementación en GPU para el *ray tracing* (sin usar un *Octree*) contra una implementación usando las mismas técnicas pero siendo ejecutada completamente en CPU, en la tabla 1 se muestran los tiempos obtenidos.

Para la versión en CPU usando solo iluminación difusa y ambiental, se obtuvo una velocidad de 6-7 fps para un total de 2 primitivas (esferas) y 2 fuentes de iluminación, figura 25.

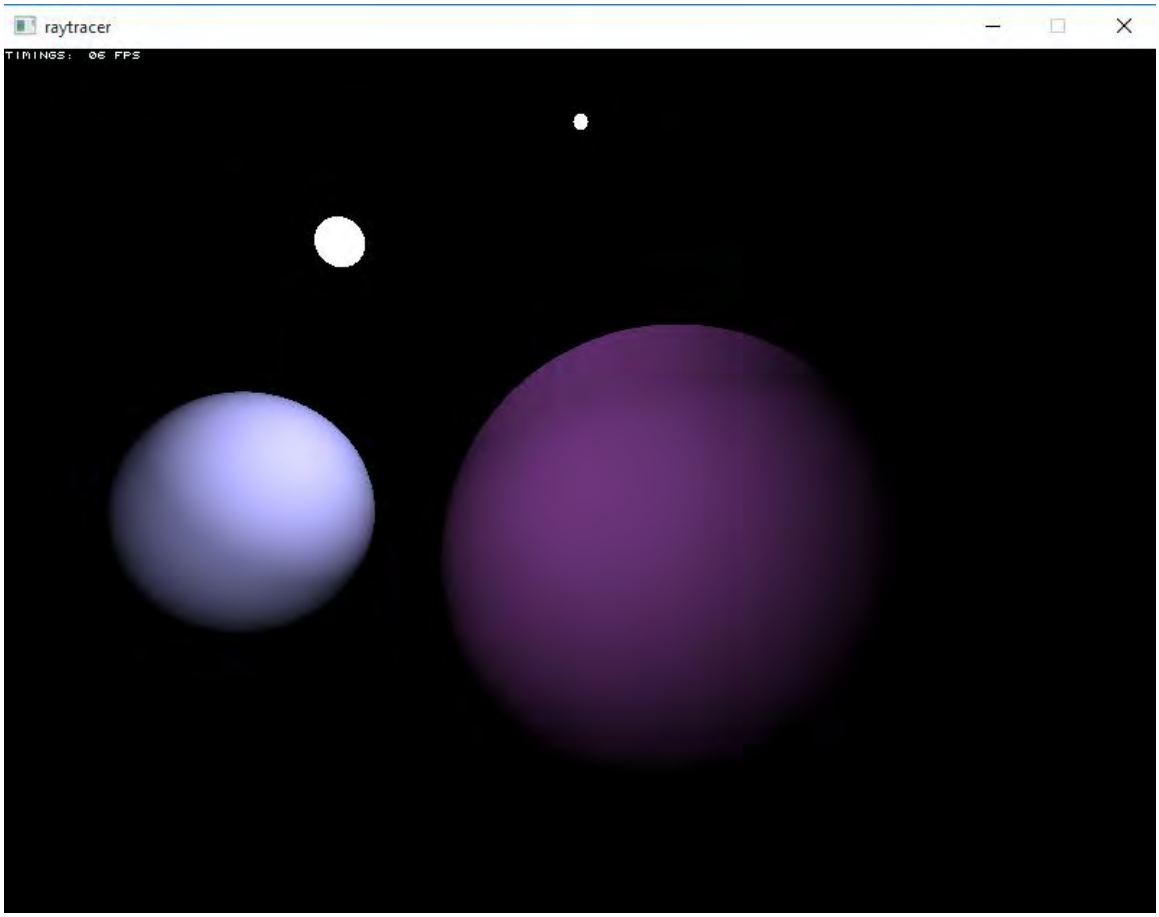

Figura 27. Ray tracing sobre 2 esferas y 2 fuentes de iluminación. Implementación usando solo CPU.

La misma escena fue probada con la implementación en GPU mostrada en la figura 26, en la cual un *kernel* de CUDA es usado para procesar cada pixel de la pantalla

independientemente. Se obtuvo un resultado de 86 fps, la escena es la misma: 2 esferas y 2 fuentes de iluminación.

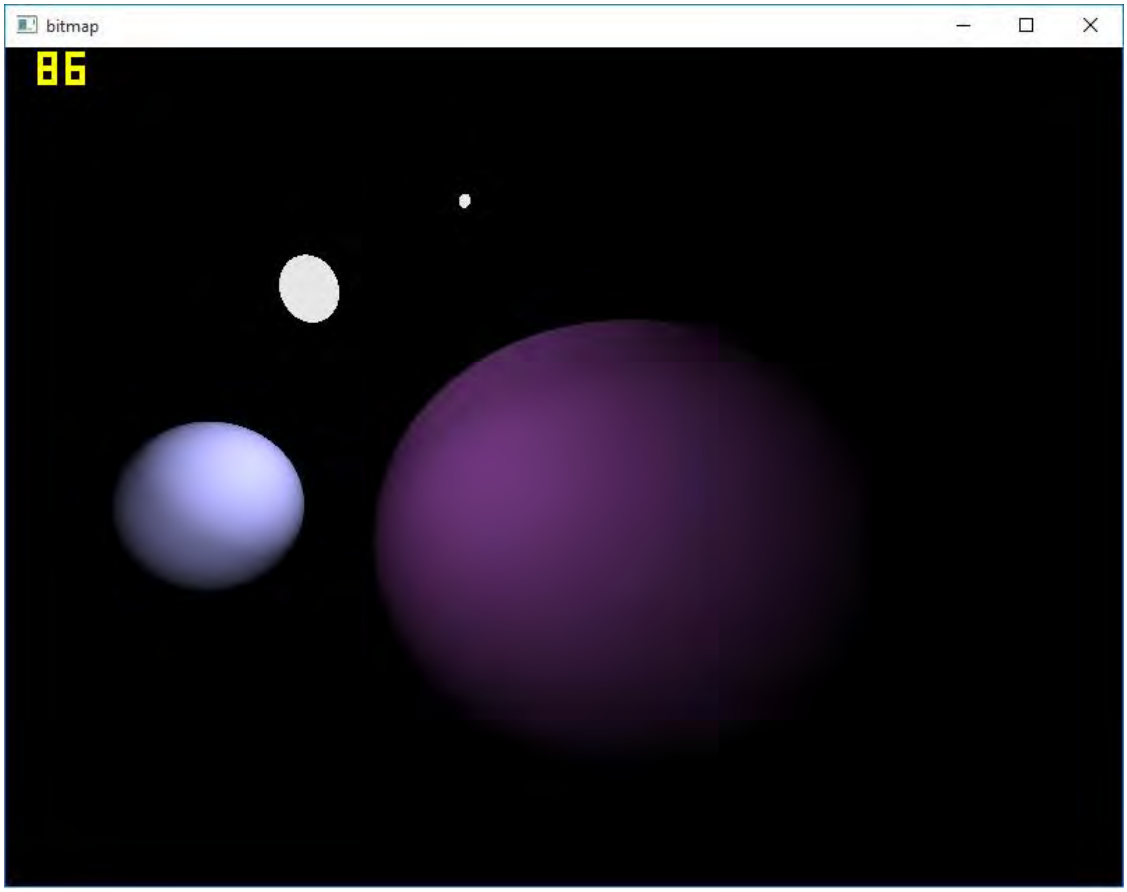

Figura 28. Ray tracing sobre 2 esferas y 2 fuentes de iluminación. Implementación usando GPU.

Se puede observar una aceleración en la velocidad del cálculo, bastante grande (aproximadamente 13 veces más veloz).

Para medir el desempeño de ambas implementaciones, se hicieron pruebas con diferentes cantidades de esferas, los resultados se muestran en la tabla 1 y en la figura 29.

cantidad de esferas	tiempo CPU (segundos)	Tiempo GPU (segundos)
1	0.125	0.0072
2	0.156	0.008
3	0.172	0.0088
5	0.234 s	0.0105
10	0.329	0.013
30	0.75	0.027
50	1.4	0.04

Tabla 1. Tiempos de creación de cada imagen para 2 implementaciones de ray tracing. Para la implementación del ray tracing en CPU se utilizó una versión desarrollada por Jacco Bikker (ver referencia).

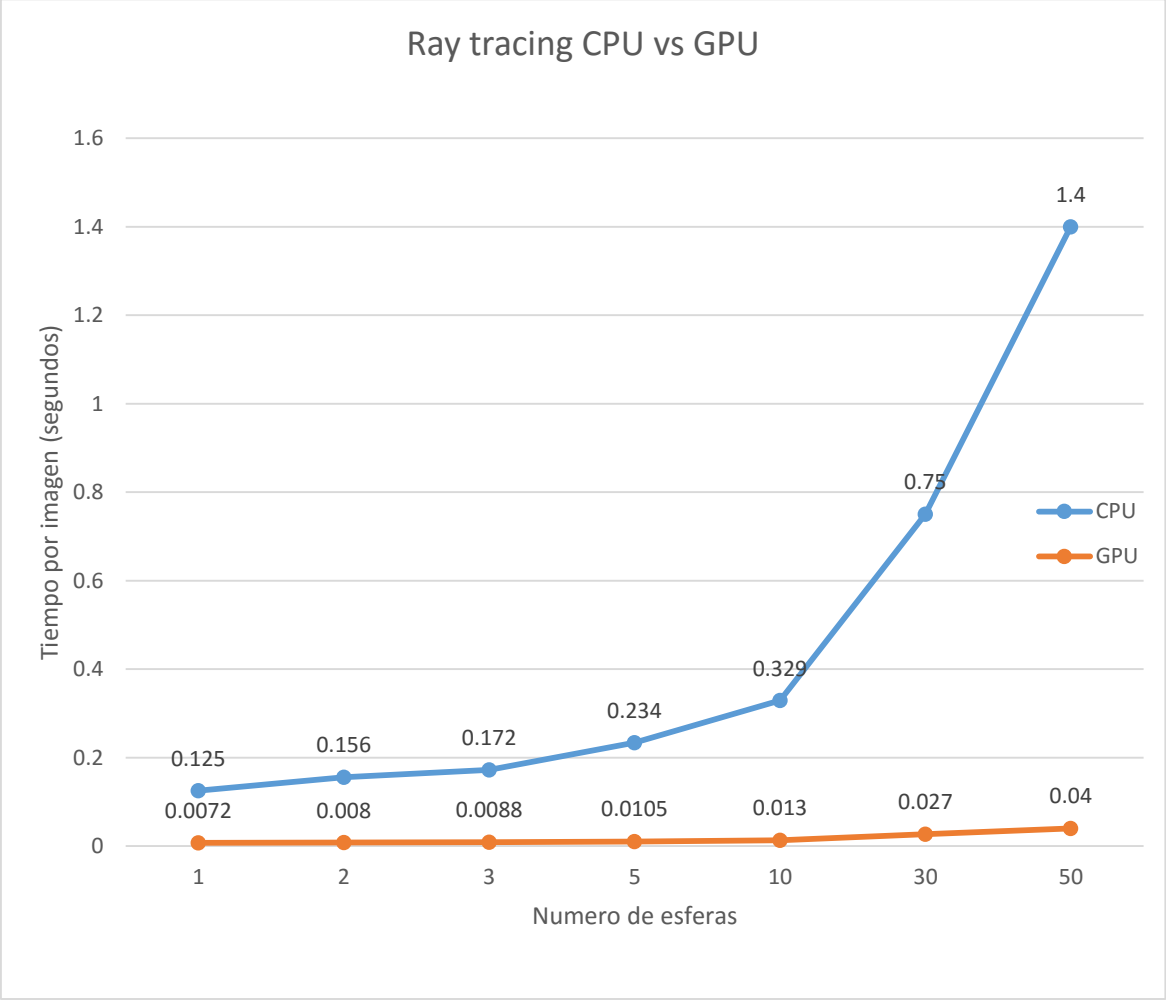

Figura 29. Grafico comparativo del tiempo que toma generar una imagen en 2 implementaciones de ray tracing, GPU y CPU.

El grafico mostrado en la figura 29 nos describe el incremento en el consumo de tiempo para generar una imagen en dos implementaciones de *ray tracing* (GPU y CPU). El grafico muestra como el crecimiento en el consumo de tiempo para la implementación en CPU aumenta de manera más rápida que la implementación en GPU, llegando a duplicarse con el aumento de esferas.

Una vez implementado el *octree* se puede alcanzar una profundidad de 7 niveles, esto se debe a la limitante de memoria con que se cuenta. Se probó con diferentes tamaños de nubes de puntos, entre más grande la nube de puntos esto solo incrementaba el tiempo de creación del *octree*. Pero no afecta el tiempo del *ray tracing* sobre el árbol.

La velocidad a la que se genera el *ray tracing* depende también sobre los pixeles (rayos proyectados) si los pixeles colisionan con nodos con información, será más tardado obtener el resultado por lo que dependiendo de la cantidad de rayos colisionados será la velocidad a la que se generen la imagen.

Para los siguientes resultados se ocupó una tarjeta de video NVIDIA GeForce GTX Titan con 6 gb de memoria. Se utilizó una nube de puntos generada a partir de un volumen segmentado de un cerebro, el volumen consta de 148 cortes o imágenes. Cada corte fue procesado considerando únicamente los pixeles que no pertenezcan al fondo. Usando las coordenadas del pixel en la imagen se obtiene la posición (x, y) de la partículas, para la posición en el eje z se utilizó el número de la imagen dentro del volumen (o número del corte), de esta manera se creó una nube de puntos que representa al volumen completo. La figura 30 muestra algunas de las imágenes usadas para generar el volumen.

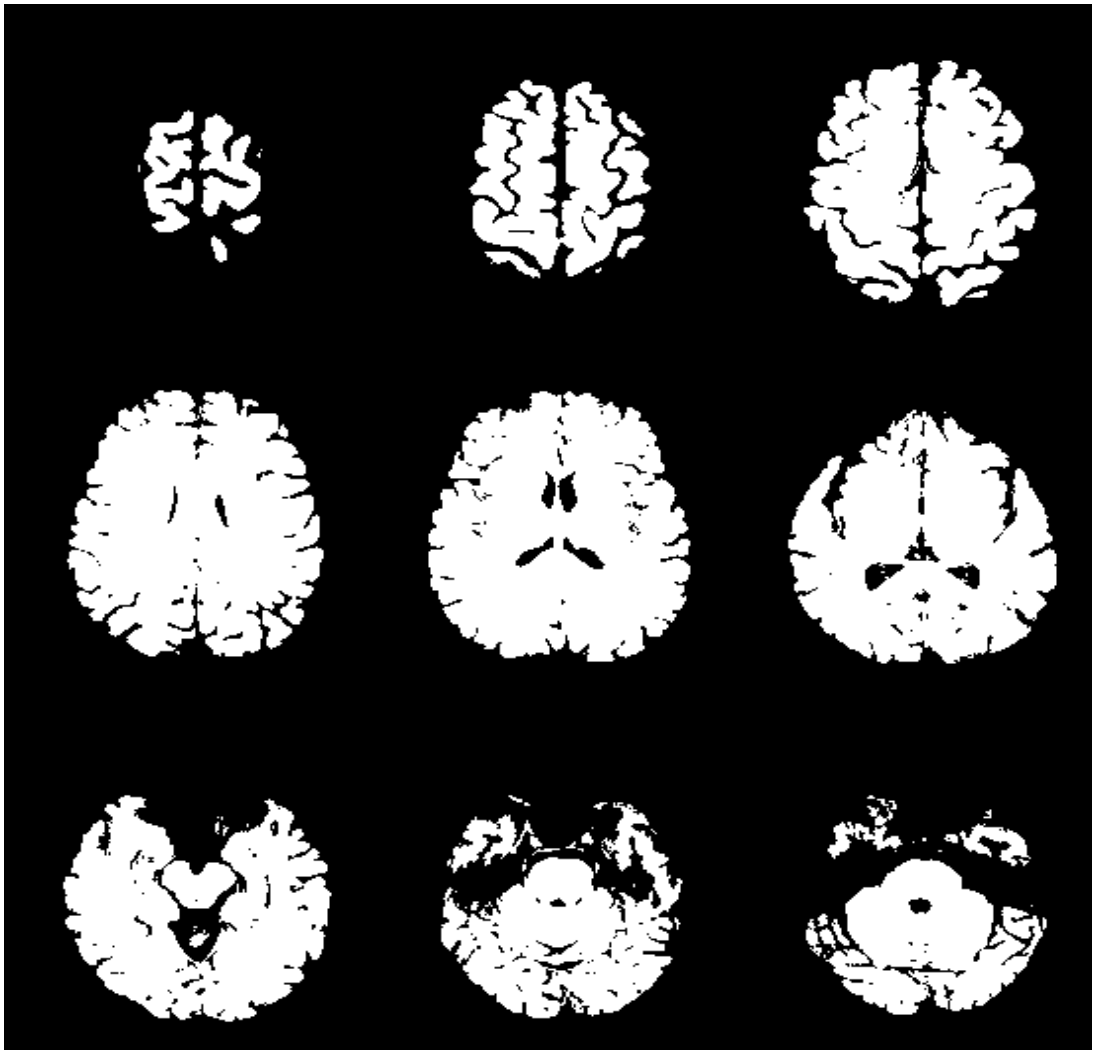

Figura 30. Ejemplo de imágenes (algunas rebanadas) utilizadas para generar una nube de puntos.

El tiempo que tarda en generarse una imagen es suficiente para alcanzar una visualización tiempo real (un mínimo de 18 fps) y da posibilidad a la navegación a través del modelo, la velocidad de renderizado depende de los rayos que interceptan el modelo y que tan disperso se encuentre el volumen dentro del *octree*.

La figura 29 (a) muestra una imagen en la que se procesó menos de la mitad de pixeles de la pantalla, alcanzando 30 fps, si la cámara es acercada, más rayos colisionaran con el objeto en pantalla (b) y por lo tanto más rayos deberán ser procesados por lo que la velocidad de generación para cada imagen disminuirá dependiendo de la cantidad de rayos que impacten con el volumen.

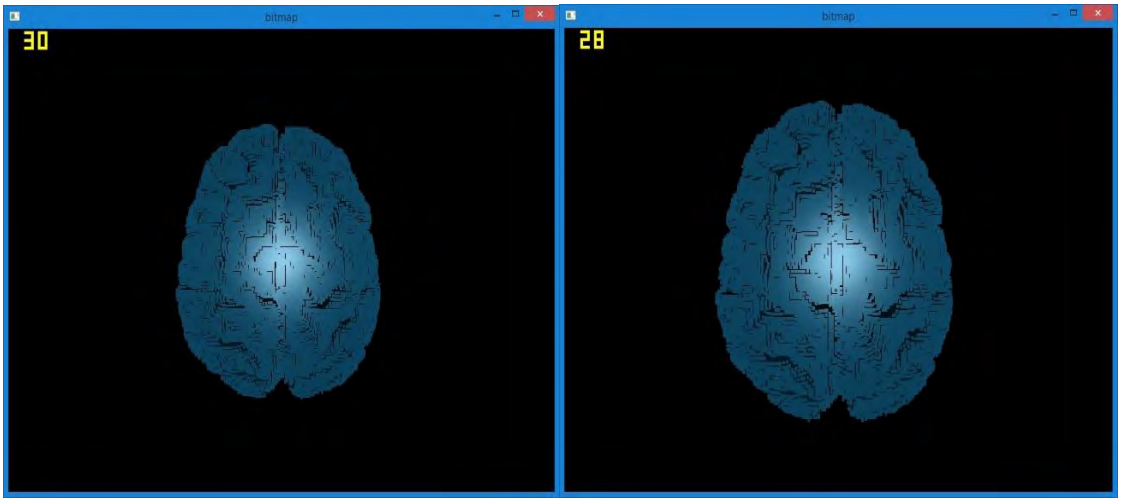

Figura 31 fps tomados para el renderizado del volumen de un cerebro (a) 30 fps (b) 28 fps.

Si se observa la imagen desde una perspectiva donde todos los rayos procesen un nodo en la imagen (calculando reflexiones) se obtiene un resultado mínimo de 18 fps (figura 30) para una profundidad 7 niveles dentro del árbol.

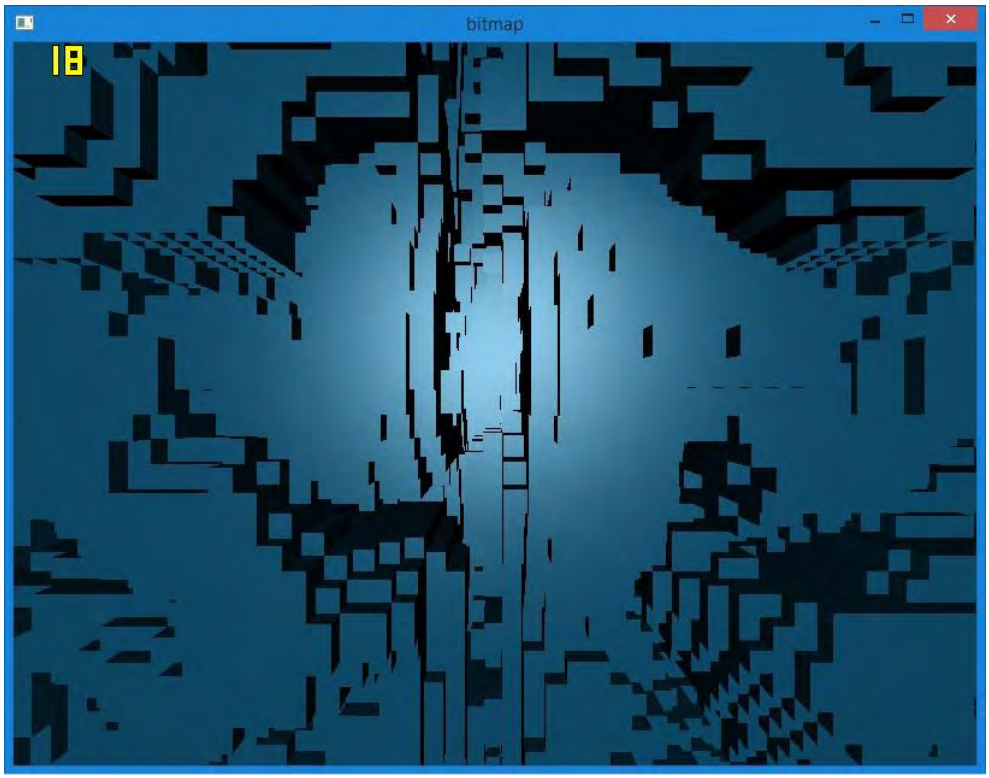

Figura 32 Renderizado del volumen del cerebro desde una perspectiva donde todos los rayos de la imagen colisionan con información del volumen (ninguno es proyectado al infinito).

5.2. Pista de profundidad.

Cuando en lugar de calcular reflexiones, usamos directamente la posición en el eje Z del nodo para determinar se obtienen imágenes donde es posible observar claramente la pista de profundidad, figuras 30 y 31. Las imágenes en la figura 32 muestran 2 diferentes perspectivas del volumen de un cerebro que fue generado a partir de 148 imágenes similares a las mostradas en la figura 27.

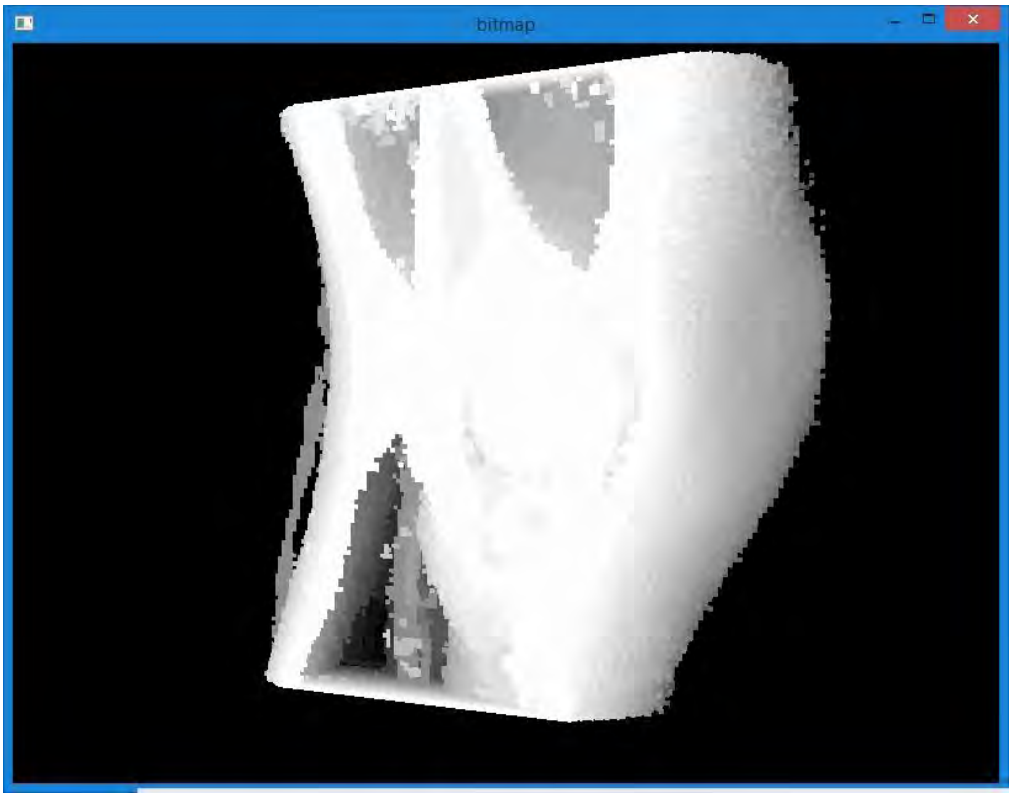

Figura 33 Volumen generado a partir de la tomografía de una rodilla, se usa la coordenada del eje Z como color.

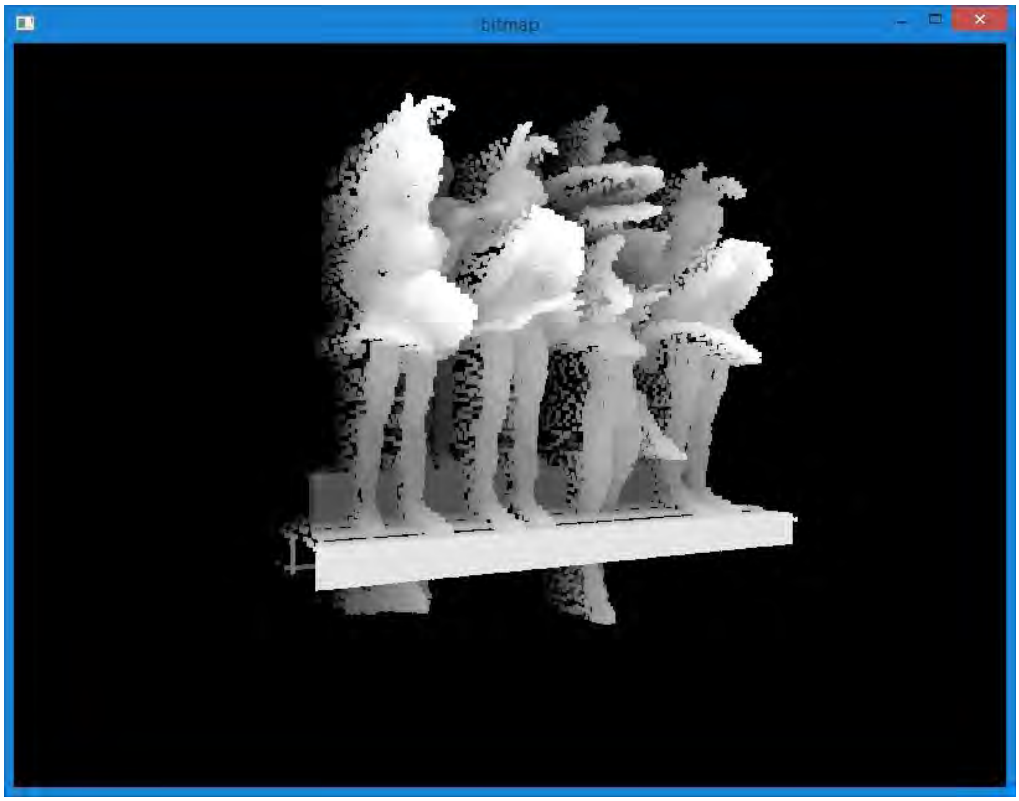

Figura 34 Volumen generado a partir de un mapa de profundidad, se usa la coordenada del eje Z como escala de gris.

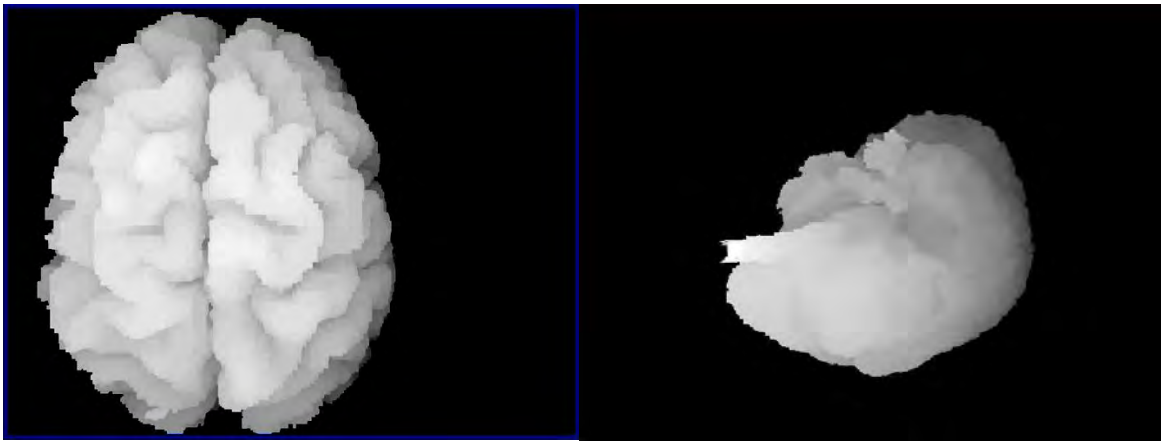

Figura 35 Diferentes perspectivas del volumen de un cerebro, se usa la coordenada del eje Z como escala de gris.

5.3. Consumo de memoria.

A pesar de solo estar construyendo el esqueleto del *octree* y generando solo los nodos que contengan información, se realiza un consumo de memoria en GPU que

aumenta de manera exponencial dependiendo de la profundidad con que sea construido el *octree*. La tabla 1 muestra el incremento en el consumo de memoria cuando el *octree* aumenta de profundidad y como crece exponencialmente. Después del nivel 9 de profundidad el uso de memoria necesario para almacenar la estructura vacía es superior al 1 gb lo cual es un gasto considerable.

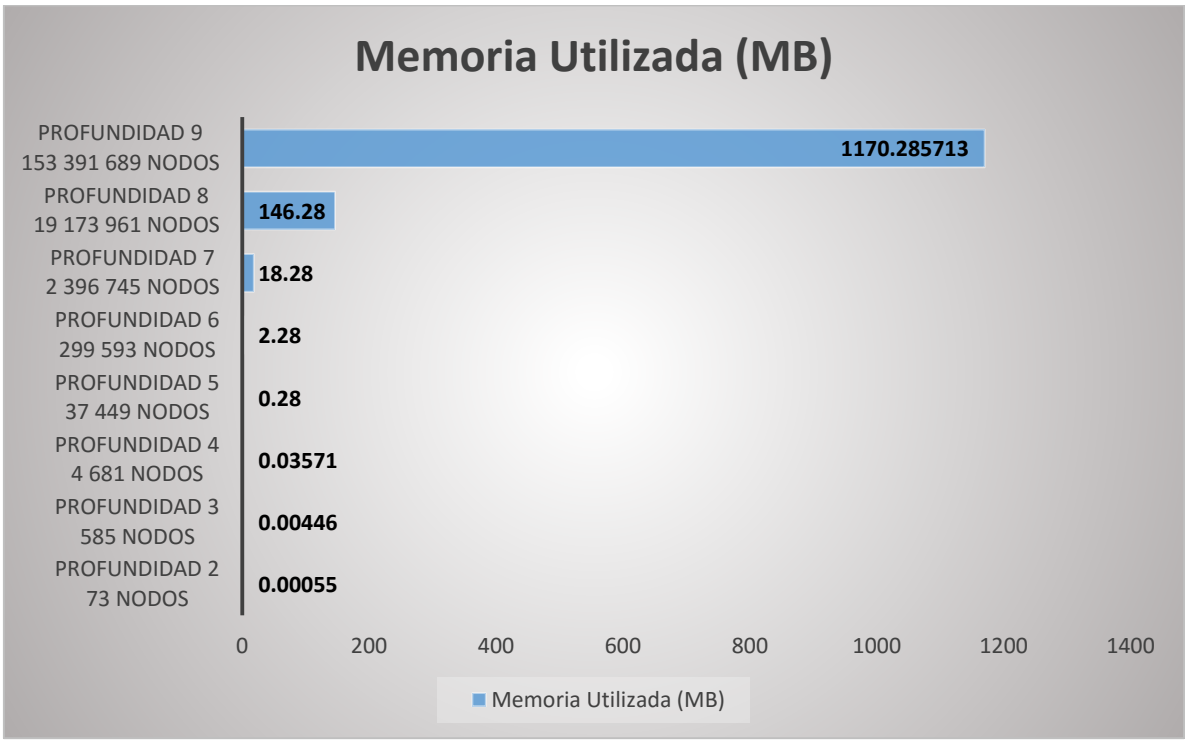

Tabla 2 Memoria utilizada para crear la estructura vacía del octree.

5.4. Tiempo de creación del *Octree*.

El consumo de memoria mostrado es solo para la estructura del *octree* vacía, una vez construida la estructura se probó con diferentes cantidades de información y se midieron los diferentes tiempos que toma procesar las partículas, reservar memoria para nodos, construir el *octree* y realizar el renderizado de la imagen. Todos los procesos fueron encapsulados en diferentes kernels de CUDA y son independientes entre sí. El ray tracing fue realizado con un total de 480 mil rayos.

Cantidad de partículas procesadas.	Procesamiento de Partículas (segundos)	Reserva de memoria para el *octree* (segundos)	Construcción del *octree*. (segundos)	*Ray tracing* por imagen (segundos)	Memoria usada (mb)
30641	0.0005	0.0195	0.0012	0.0255	2425
303374	0.0017	0.2003	0.0033	0.0445	2435
951879	0.0042	0.7084	0.0043	0.0443	2460
1076596	0.0046	6.0923	0.0053	0.0244	2464
3283639	0.0217	1.9085	0.0071	0.0368	2549

Tabla 2. Se muestran los diferentes tiempos utilizados para ejecución de diferentes kernels en CUDA, se probó con diferentes cantidades de información y también se proporciona información de la memoria ocupada por todo el programa durante la ejecución.

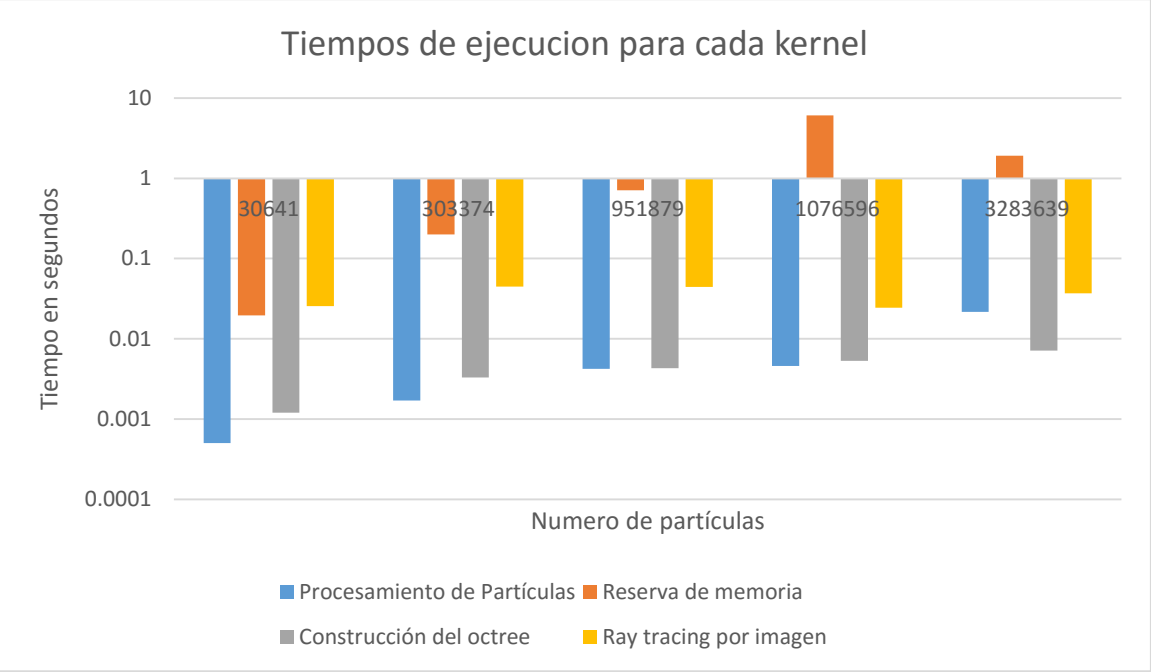

Figura 36 Mediciones de los tiempos que toma cada kernel en ser ejecutado.

La tabla 2 muestra los resultados de las mediciones realizadas y la figura 36 muestra una comparación en el incremento de tiempo consumido por las diferentes partes

del proceso de creación del *octree*. Se puede observar un aumento constante en el procesamiento de las partículas conforme la cantidad de partículas aumenta pero no sucede lo mismo para la reserva de la memoria del *octree* ni para el *ray tracing*. El tiempo que toma reservar la memoria y realizar el *ray tracing* no depende directamente de la cantidad de partículas procesadas, dependen de otros factores diversos como la dispersión de la nube de puntos y en algunos casos la perspectiva desde la que se esté viendo el volumen.

Una de las ventajas que ofrece el enfoque tomado para esta tesis es la capacidad de modificar la posición de las partículas en tiempo real y poder actualizar el *octree* de manera que el cambio de posición se vea reflejado directamente en las imágenes generadas. Si un grupo de partículas viaja a través del *octree*, existe un kernel de CUDA que se encarga de revisar si las partículas cambiaron de posición y en caso de hacerlo actualiza la estructura del *octree*. La figura 37 muestra algunas partículas moviéndose dentro del *octree*, atraviesan todo el espacio del *octree* liberando y reservando memoria dependiendo si el nodo se queda vacío o entra una nueva partícula y debe ser creado. El rendimiento del renderizado depende directamente del camino que las partículas recorran. Si las partículas se mueven hacia lugares vacíos todo el tiempo entonces es necesario crear muchos nodos nuevos y liberar nodos ya no usados, por lo tanto la velocidad de renderizado decrece considerablemente.

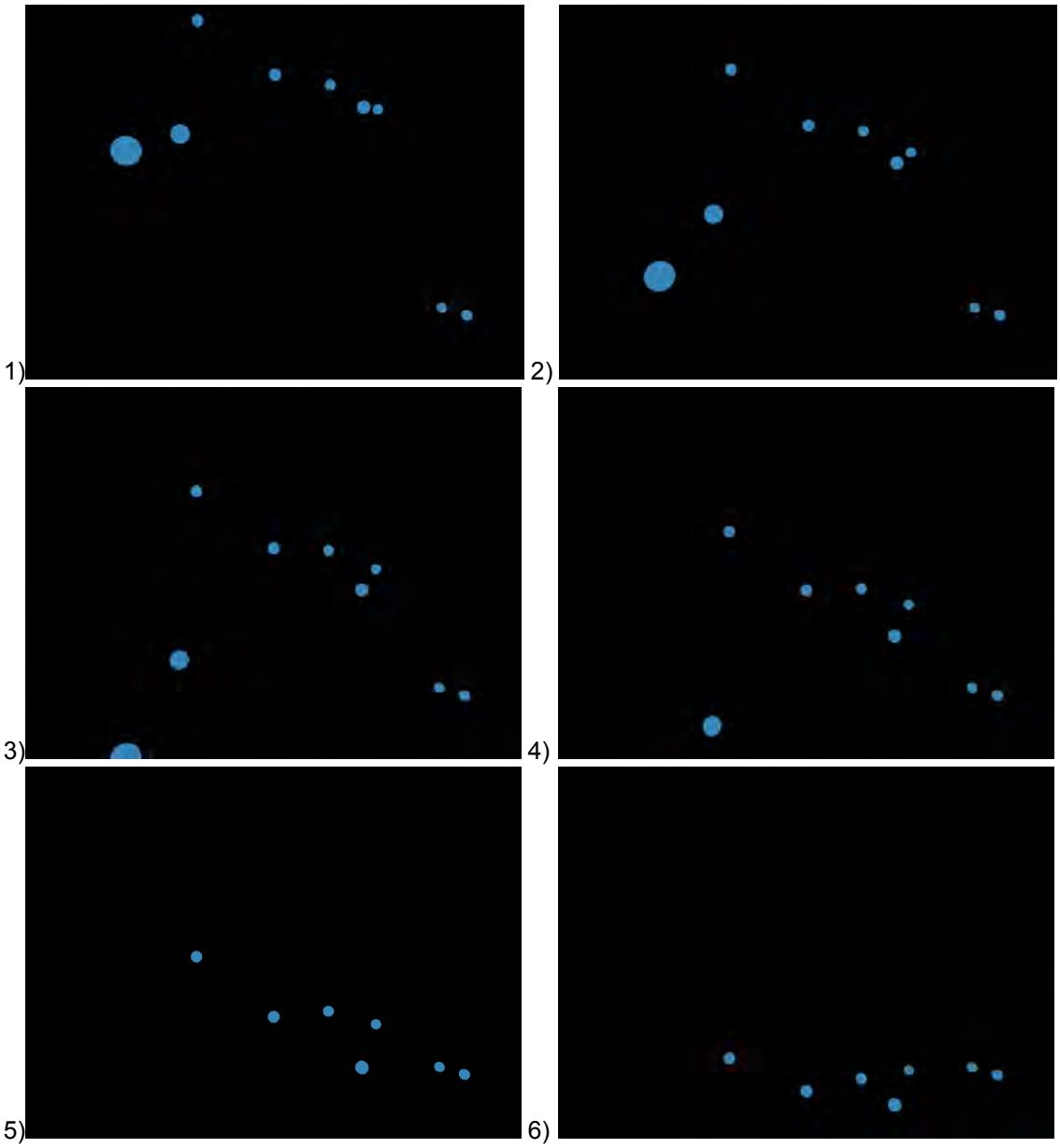

Figura 37. Imágenes mostrando como las partículas viajan a través del octree, liberando memoria y creados nuevos nodos cuando sea necesario.

Capítulo 6. Conclusiones y trabajo a futuro.

6.1. Conclusiones

En esta tesis se presenta una nueva metodología para creación y manejo de una estructura de datos *octree* usando una arquitectura en paralelo (CUDA). A diferencia de las implementaciones tradiciones en la cuales se manejan 2 estrategias principales: si el *octree* es creado completamente se tienen un consumo de memoria bastante grande y para las aplicaciones que se realizan completamente usando memoria de GPU representa una limitante; cuando el *octree* es construido de manera que solo los nodos que contengan información sean creados, se optimiza el consumo de memoria pero dado que el *octree* se crea acorde a los datos iniciales, todo se acomoda y acopla de acuerdo a ellos, por lo tanto se vuelve complejo de modificar. La estrategia abordada en esta tesis propone un enfoque hibrido creando un esqueleto vacío del *octree* y solo creando nodos que tengan información, de esta manera se optimiza el uso de memoria sin sacrificar la versatilidad del *octree* y es posible modificar las información dentro del *octree* en tiempo real. De acuerdo a los resultados obtenidos el rendimiento observado depende directamente de la aplicación y las variables que esta tengo.

Al crear la estructura del *octree* a manera de un esqueleto vacío que contenga solo apuntadores a los nodos pero si los nodos no contienen información se convierte en una apuntador *null* y solo los nodos que contenga información son creados, se optimiza considerablemente la memoria pero el consumo de ella sigue siendo exponencial conforme el *octree* se vuelve más profundo. A profundidad 7 se obtiene un consumo de aproximadamente 18 MB solo para almacenar la estructura vacía pero al pasar a la siguiente profundidad se obtiene un consumo de 146 MB aproximadamente, lo cual muestra el crecimiento considerable en el consumo de memoria mientras el nivel del *octree* aumenta. Esto nos lleva a la conclusión de que a pesar de existir un consumo considerablemente menor de memoria en GPU, esta sigue siendo una limitante para las generaciones actuales de tarjetas de video.

La implementación de la clave Morton y una estructura de datos lineal, para poder mapear el *octree*, vuelven menos compleja la creación, navegación y modificación del *octree*. Se ahorran operaciones y consumo de memoria pudiendo guardar

relaciones entre los nodos de una manera estructurada. Es posible modificar el *octree* y la información contenida en tiempo real.

La velocidad de creación del *octree* depende principalmente de la cantidad de nodos que contengan información, por lo tanto si la nube de puntos se encuentra muy dispersa por el espacio del *octree*, esto llevara a la creación de una gran cantidad de nodos y tomara más tiempo la creación de la estructura completa.

El *ray tracing* implementado en GPU demuestra ser superior a las implementaciones tradicionales en CPU, acelerando el proceso considerablemente. Sin embargo el aceleramiento en los diferentes procesos depende directamente de que tan independientes son las operaciones entre ellas, si al momento de utilizar un kernel de CUDA los diferentes threads lanzados tienen alguna dependencia de información o buscan acceso a zonas de memoria al mismo tiempo, la solución en paralelo se vuelve serial.

Las soluciones presentadas implementadas en una arquitectura en paralelo son soluciones escalables y dependen de la tarjeta de video donde se estén ejecutando. Si la tarjeta de video contiene 128 núcleos de procesamiento, esto significa que 128 threads serán procesados al mismo tiempo y una vez terminados se procesaran los siguientes 128. Conforme aparezcan en el mercado nuevas generaciones de tarjetas de video, con más núcleos y más memoria, el alcance de estas soluciones crecerá y mejoraran en rendimiento.

Si las partículas salen del espacio generado originalmente para el *octree*, no es posible visualizarlas ya que ahora no pertenecen al *octree* como tal. El espacio inicial generado para el *octree* no puede ser ampliado.

6.2. Trabajo a futuro.

La implementación presentada muestra un renderizado volumétrico que aprovecha las ventajas del uso de CUDA, existe mucho trabajo por realizar en diferentes aspectos. Es necesario optimizar el sistema para obtener una implementación aún más veloz.

Usando técnicas de procesamiento de imágenes que puedan ser encapsuladas en kernels puede aplicarse diferentes filtros a las imágenes.

Dado que es posible modificar la información dentro del octree es posible implementar diferentes kernels que simulen física para las partículas y de esta manera obtener diferentes efectos como flujo de líquidos, humo, destrucción de sólidos, simulación de terrenos, etc. De ser necesario se puede usar variables de mayor tamaño para la representación de las claves Morton para obtener un resolución superior (pero con un consumo de memoria mayor).

Los mallados pueden ser procesados para ser transformados a una nube de puntos usando los vértices como puntos en el espacio, de esta manera se pueden renderizar mallados.

La aplicación a diferentes áreas es posible adaptando el sistema a las diferentes aplicaciones, dado que se trabajan con partículas que no tienen dependencia unas de las otras, diferentes propiedades pueden ser asignadas de manera independiente como puede ser atracción, fuerza, peso, emisión de luz, etc. En el caso de aplicaciones médicas es posible asignar las propiedades de tejidos a las partículas y realizar cortes sobre las nubes de puntos que permitan la simulación de heridas o cirugías.

Si las tecnologías superiores presentan un acceso al uso de recursión dentro de GPU se puede actualizar esta implementación para que funciones de manera recursiva lo cual daría entrada a muchas otras optimizaciones como:

Implementar una búsqueda de vecinos que ayude a propósitos de simulación, ubicando las partículas y sus vecinos.

Calculo de propiedades de color, normales, refracción y difracción dependiendo de una vecindad.

Búsqueda de partículas dentro de los nodos sin necesidad de ir a un nivel muy profundo dentro del octree.

Esta tesis presento una primera aproximación a una implementación en GPU para renderizado volumétrico, sin embargo existe mucho trabajo por realizar.

Bibliografía

Appel A. (1968) Some techniques for shading machine renderings of solids. AFIPS Conference.

Bikker Jacco "Raytracing Topics & Techniques". Noviembre 2004. http://www.flipcode.com/archives/Raytracing_Topics_Techniques-Part_2_Phong_Mirrors_and_Shadows.shtml.

Christer Ericson "Real-Time Collision Detection". 2005, editorial CRC Press. Sony Conputer Entertainment America.

Gastélum Alfonso (2011) "Smoothing Particle Hydrodynamics parallel implementation for numerical modelling of solid-fluid interactions". Tesis de doctorado. Universidad de Auckland, Nueva Zelanda.

Elseberg Jan, Borrmann Dorit, Nüchter Andreas (2011) "Efficient Processing of Large 3D Point Clouds"; Jacobs University Bremen. Bremen, Alemania.

Morton, G. M. (1966), "A computer Oriented Geodetic Data Base"; and a New Technique in File Sequencing, Technical Report, Ottawa, Canada: IBM Ltd.

NVIDIA "NVIDIA cuda c programming guide version 7.5", NVIDIA. Última actualización Septiembre 1, 2015.

Guía de programación NVIDIA "Cuda api reference manual version 7.5", NVIDIA. Última actualización Septiembre 1, 2015.

Guía de programación NVIDIA "Cuda c best practices guide", NVIDIA. Last updated September 1, 2015.

Roth, Scott D. (Febrero 1982), "Ray casting for Modeling Solids", Computer Graphics and Image Processing. Vol 18, no., pagina 109–144*.*

Samuli Laine y Tero Karras (2011) "Efficient Sparse Voxel Octrees"; IEEE Transactions on visualization and computer graphics. Vol 17.no, pp.

Segovia Alejandro, Li Xiaoming, Gao Guang (2009) “Interative Layer-Based Raytracing on CUDA”, Performance Computing and Communications Conference (IPCCC); IEEE 28th International.

Suryakant Patidar y P.J. Narayanan (2008) “Ray casting Deformable Models on the GPU”. Computer Vision, Graphics & Image Processing. Sixth Indian Conference.

Kay T. L. y Kajiya J. T. Ray tracing complex scenes. In SIGGRAPH ’86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques, pagina 269–278, New York, NY, USA, 1986. ACM Press.

Yue Zhao, Xiaoyu Cui, Ying Cheng (2009) “High-Performance and Real-Time Volume Rendering in CUDA”; Northeastern University, Shenyang, China.

Zhang Fan y Xie Mei (2008) “Real-time Medical Image Volume Rendering Based on GPU Accelerated Method”; Computational Intelligence and Design. ISCID '08. International Symposium on (Volume:2).