Centro de Investigación y de Estudios
Avanzados del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

Ray Tracing en tiempo real con GPGPUs

Tesis que presenta

Carlos Daniel Venegas Tamayo

para obtener el Grado de

Maestro en Ciencias en Computación

Director de la Tesis

Dr. Amilcar Meneses Viveros

México, Distrito Federal **Diciembre 2013**

Resumen

Desde tiempos inmemoriales, el ser humano se ha visto en la necesidad de expresar sus ideas en forma de imágenes. Esta necesidad ha llevado al hombre a la creación de imágenes asistiéndose de la tecnología, desde lápices hasta el uso de supercomputadoras. Recientemente, se ha visto una proliferación de contenidos audiovisuales 3D creados por computadora. Estos contenidos los podemos encontrar en varios campos como la industria del entretenimiento, el diseño de interiores, la medicina y la química; por consiguiente, es de interés tener contenidos cada vez más realistas. Crear estos contenidos 3D conlleva todo un proceso que consiste, a grandes rasgos, en la creación de modelos 3D, especificación de materiales y fuentes de iluminación, síntesis de imágenes y finalmente, si hace falta, la post producción. Dentro de este proceso creativo, la síntesis de imágenes requiere de un algoritmo computacional, de entre los cuales existen el rasterizado y el ray tracing. El ray tracing en específico es capaz de generar imágenes de gran calidad; sin embargo, tiene un costo computacional muy elevado.

Recientemente ha surgido la necesidad de tener imágenes realistas en tiempo real, esto representa un problema debido a las afirmaciones del párrafo anterior. El tiempo real (en computación gráfica) consiste en la generación rápida de imágenes y es el área más interactiva de la computación gráfica. Una imagen es proyectada en la pantalla, el espectador reacciona o actúa y la retroalimentación afecta lo próximo que será generado. La tasa de imágenes desplegadas es medida en frames por segundo (fps) o hertz (Hz), un mínimo de 15 fps podría considerarse como tiempo real [1].

Este problema se ha atacado de diversas formas, desde clústers de CPUs hasta la creación de hardware dedicado. Estas soluciones implican un costo monetario y de consumo energético muy elevado que no es rentable para la mayoría de los usuarios. Una alternativa por la que se ha optado es el uso de GPUs, pues se ha observado que el modelo de trabajo que presenta el ray tracing es altamente paralelizable y se puede adaptar a una arquitectura de SIMD (Single Instruction Multiple Data) que es la arquitectura que manejan las GPUs.

Esta tesis se enfoca en implementar el algoritmo de ray tracing en paralelo utilizando GPUs teniendo como hipótesis que es posible alcanzar el tiempo real. También se pretende adaptar una estructura aceleradora para el mismo algoritmo que disminuye el tiempo de ejecución considerablemente.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Antecedentes

En la actualidad la abundancia de contenidos audiovisuales digitales es evidente. En la vida cotidiana es posible observar desde imágenes simples hasta animaciones complejas que han sido manipuladas o creadas completamente por computadora. Específicamente en la síntesis de imágenes por computadora, existe un algoritmo llamado *ray tracing* que es capaz de alcanzar un alto grado de realismo en sus resultados. A pesar de esto, *ray tracing* representa una inversión muy grande de tiempo, dinero y energía eléctrica; por esta razón, existen otros algoritmos cuya carga de trabajo es más ligera y por consecuencia consumen menos recursos. Sin embargo, estos algoritmos no alcanzan la calidad que produce *ray tracing*.

Con el paso del tiempo, los principales usuarios de estas tecnologías desean obtener imágenes realistas en el menor tiempo posible. Este problema ha sido atacado de diferentes maneras, desde el uso de *clusters* hasta la invención de hardware completamente dedicado al algoritmo. Los resultados obtenidos de estas soluciones son buenos, pero hay que tomar en cuenta que no son soluciones económicas. Recientemente, se ha observado un gran crecimiento en el uso de GPUs para la solución de varios problemas y *ray tracing* no es una excepción. Gracias al modelo que presenta *ray tracing*, es posible proponer una paralelización utilizando un modelo SIMD (*Single Instruction Multiple Data*) que es exactamente el modelo de programación que sigue la programación sobre GPUs. Esta tesis esta enfocada a una disminución de tiempo de ejecución en el algoritmo de *ray tracing*, tal que se obtengan resultados en tiempo real utilizando GPUs aprovechando las propiedades de ambos (el algoritmo y las GPUs) . En específico se va a usar Nvidia CUDA (*Compute Unified Device*

Architecture) como plataforma de trabajo.

1.2. *Ray Tracing*

El algoritmo de *ray tracing* surge a partir de una técnica de dibujo que data del año 1538 creada por el matemático y pintor alemán Albercht Dürer. Esta técnica consistía en colocar una malla entre el dibujante y la escena a dibujar, esto en conjunto con un modelo geométrico de sombreado lograba dibujos con un alto grado de realismo [2].

Mucho tiempo después con el surgimiento de las computadoras, Arthur Appel desarrolla *Ray Casting* [3], un algoritmo computacional que sigue el mismo concepto la técnica de dibujo de Albercht Dürer. *Ray Casting* fue refinado (añadiendo diversos efectos ópticos) hasta convertirse en lo que hoy se conoce como *ray tracing* [4].

Ray tracing es un algoritmo de la computación gráfica cuyo objetivo es la síntesis de imágenes a partir de modelos 3D. Este algoritmo consiste en proyectar una serie de rayos a través de un plano de proyección hacia una escena 3D a partir de un punto de vista. Se debe prestar atención al hecho de que la escena debe ser representada de forma vectorial y como tal, los rayos y el punto de origen también precisan ser representados de la misma manera. El plano de proyección representa la resolución de la imagen resultante, por ejemplo, un plano de 800 por 600 píxeles tendrá como resultado una imagen de 800x600 píxeles.Una vez que los rayos son proyectados hacia la escena se calcula el color correspondiente al píxel del plano, esto es posible encontrando las intersecciones más cercanas entre cada rayo y la escena. Además, para determinar las sombras es necesario proyectar un rayos secundarios hacia las fuentes de luz a partir de las intersecciones antes mencionadas. Finalmente estos rayos pueden o no, dependiendo del material de los objetos dentro de la escena, producir rayos terciarios de reflexión o refracción, los cuales aumentan más la complejidad del algoritmo.

La complejidad de *ray tracing* es medida con respecto al número de primitivas que forman la escena, entiendase por primitivas los objetos que conforman la escena (triángulos, esferas, planos, entre otras). La implementación más sencilla de *ray tracing* tiene una complejidad de $O(n^2)$ donde n es el número de primitivas. Esta complejidad también se puede ver como una relación entre el tamaño de la imagen resultante y el número de objetos resultante alcanzando una complejidad de $O(hwn)$ donde h es la altura de la imagen, w el ancho y n el número de primitivas.

1.3. Motivación y planteamiento del problema

El manejo de imágenes sintetizadas por computadora es fundamental para la mejora de algunas disciplinas cada vez más demandantes. Por ejemplo, en la industria del entretenimiento podemos encontrar cada vez más películas creadas totalmente por computadora y videojuegos con escenarios más realistas. Paralelamente, también se han utilizado imágenes de alta definición en el campo de la medicina para la detección y estudio de patologías. De ahí que el avance tecnológico requiere el uso de imágenes con un realismo de alto nivel.

El proceso de creación de contenidos digitales se divide en varias fases, desde el modelado de objetos hasta la post-producción. El modelado de objetos y texturización son hechos con herramientas de diseño como Autodesk Maya, 3D Studio Max, Blender, entre otras. El renderizado es un proceso que consiste en obtener una imagen a partir de un modelo vectorial de la misma, dentro de esta fase existen varios algoritmos, los dos más utilizados son *ray tracing* y rasterizado. Particularmente, el algoritmo de *ray tracing* es visto como una técnica para obtener imágenes con interacciones complejas de luz, es decir que es posible tener escenas con objetos transparentes y refractantes obteniendo resultados realistas. Debido a la carga de trabajo inherente a esta técnica, se han diseñado soluciones utilizando *clusters* a un alto costo monetario, a su vez se ha optado por el uso de GPUs y también se han diseñado soluciones híbridas.

Analizando el flujo de trabajo y a la distribución de datos que presenta *ray tracing*, es factible proponer una solución que sea masivamente paralela. Se ha visto que cuando las soluciones pueden ser paralelizadas masivamente una de la mejor opción para obtener un alto desempeño, es hacerlo sobre una arquitectura GPU. Podemos encontrar ejemplos muy claros de este tipo de soluciones en diferentes ramas de la investigación como en la bioinformática[5]. Además, se ha visto la factibilidad de paralelizar el algoritmo de *ray tracing* en el trabajo realizado por Lucía Oviedo [6] en el Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional.

El problema principal a resolver es la necesidad que tienen los usuarios de sintetizar imágenes en tiempos cortos, ya que se ha observado que la mayor parte del tiempo dentro del proceso de producción de contenidos digitales, relacionados con imágenes, se utiliza en la fase de renderizado.

1.4. Objetivo de la tesis

Esta tesis tiene como objetivo principal desarrollar un algoritmo paralelo heterogéneo de *ray tracing* que aproveche las ventajas de las GPUs y CPUs, con la hipótesis de que es posible alcanzar el tiempo real. Para poder lograr este objetivo se plantean los siguientes objetivos específicos:

- Identificar regiones altamente paralelizables.
- Identificar regiones que se realizan directamente en la CPU.
- Establecer un esquema de sincronización de kernels y datos.
- Diseñar e implementar de una estructura aceleradora análoga a un árbol para el algoritmo que sea capaz de trabajar en paralelo.
- Diseñar e implementar de un modelo de trabajo en paralelo.
- Realizar pruebas.

1.5. Contribuciones de la tesis

Se realizó un estudio comparativo de algoritmos de *ray tracing* implementados en CPUs y GPUs. Además, las nuevas tecnologías involucran la creación de algoritmos paralelos heterogéneos, tal es el caso de esta tesis. Añadido a esto, se propuso un modelo de sincronización interna y externa entre CPUs y GPUs, es decir, la sincronización que se lleva a cabo entre ambas arquitecturas (GPUs y CPUs) y la sincronización que hay que realizar dentro de la propia arquitectura. Finalmente se realizó una implementación capaz de alcanzar el tiempo real del algoritmo de *ray tracing*.

1.6. Estructura de la tesis

El capítulo 2 es el estado del arte acerca de contenidos digitales y *ray tracing*. En relación a los contenidos digitales se expone una clasificación de contenidos digitales referentes a imagenes. También exponemos a detalle este algoritmo y una comparación con rasterizado. Además, se explican las consideraciones que hay que deben tomarse en cuenta para la implementación en paralelo. Finalmente se exponen los trabajos relacionados a *ray tracing* con el uso de GPUs y *ray tracing* en tiempo real. El

capítulo 3 estudia las tecnologías de programación paralela heterogénea existentes, arquitecturas *multicore* y GPU. El capítulo 4 cubre el desarrollo de la solución, es decir, el modelo de paralelización, las solución a la problemática y el flujo de trabajo que sigue el algoritmo en términos de computación paralela. El capítulo 5 presenta las pruebas, las validaciones realizadas y se estudian los resultados obtenidos. Estos resultados son evaluados en términos de tiempo de ejecución y complejidad de las escenas de prueba.Para finalizar, el capítulo 6 muestra las conclusiones y trabajo a futuro, aquí se plantean las posibles extensiones que este trabajo puede llegar a tener así como nuevas hipótesis obtenidas a través de la experiencia de haber realizado este proyecto de tesis.

Capítulo 2

Estado del arte

En este capítulo se exponen la importancia del *ray tracing* en la creación de contenidos digitales, se comparan los dos algoritmos más usados en el proceso de la sintesis de imágenes 3D y se explica la teoría relacionada a *ray tracing*. Puesto que este algoritmo representa un tema muy amplio, serán expuestos solamente sus puntos principales. Además se revisan algunas estructuras que sirven de apoyo al algoritmo de *ray tracing*. También se da a conocer parte de la infraestructura actual de hardware utilizada para la ejecución de este algoritmo. Finalmente se presentan los trabajos más recientes relacionados al tema.

2.1. Contenidos Digitales

La producción de contenidos digitales en forma de imágenes es el punto clave de esta tesis, mediante estos es posible tener la representación de un entorno virtual creado a partir de la imaginación o de un entorno externo. Para su creación y despliegue es necesario el uso de computadoras y la ayuda de técnicas de la computación gráfica. Además, estos contenidos se pueden encontrar en diferentes áreas, tales como en la arquitectura, el diseño de piezas industriales, la industria del entretenimiento, la química, entre otras.

2.1.1. Clasificación de contenidos digitales

Existen varias formas de clasificar los contenidos digitales; pueden evaluarse factores como la resolución de la imagen, el tipo de formato de imágenes e incluso por el método computacional que haya sido utilizado para crearlos [7]. A continuación damos una

breve descripción de esta clasificación:

- **Clasificación basada en tiempo.** El almacenamiento es en forma de muestras por intervalo de tiempo, conocido como fotogramas por segundo o fps. Se utiliza en contenidos digitales de video.

  Los fps representan la frecuencia en la que los dispositivos producen imágenes consecutivas únicas dando la sensación de animación. Esta clasificación esta fundamentada en el hecho de que el ojo humano es capaz de procesar de 10 a 12 imágenes separadas por segundo, buscando como objetivo garantizar que no se muestre una discontinuidad en esa sensación del movimiento de dichas imágenes.

- **Clasificación basada en el proceso de render.** El almacenamiento es en forma de pixeles fijos, algunos formatos conocidos son jpeg, mapa de bits, png, ppm, entre otros. Se usa para clasificar imágenes, fotografías.

  Una imagen rasterizada es una matriz generalmente rectangular cuyos elementos son definen pixeles o puntos de color; dichas imágenes, a diferencia de las imágenes vectoriales, son dependientes de la resolución, en otras palabras, no se pueden escalar arbitrariamente sin perder calidad. El uso de imagenes rasterizadas es mas práctico para el manejo de fotografías e imágenes foto-realistas, mientras que los gráficos vectoriales son mayormente utilizados para la composición tipográfica o en el diseño gráfico.

- **Clasificación basada en proceso de render.** Este tipo de imágenes es generado mediante ecuaciones matemáticas y conjuntos de datos numéricos, tales como diseños 2D y 3D, modelos y objetos. Se almacenan en forma de vectores o de escenario gráfico. Estos algoritmos se pueden clasificar en dos tipos, los que determinan la superficie visible [8] como z-buffer desarrollado por Catmull [9], árbol BSP desarrollado por Fuchs, Kedem y Naylor [10]; y los que determinan la iluminación y sombra como ray tracing recursivo propuesto por Appel [3] y mejorado por Whitted [4].

2.2. Rasterización

La rasterización es una de las dos técnicas principales para la generación de imágenes 3D. El uso de esta técnica es muy popular en la producción de imágenes 3D en tiempo

real gracias a su rapidez. La rasterización es el proceso de calcular el mapeo de la geometría de la escena hacia los pixeles de la imagen y no especifíca la manera de calcular el color de estos pixeles. El sombreado puede hacerse de muchas formas como el transporte físico de la luz o un sombreado artístico.

La rasterización más básico consiste en transformar los puntos 3D de los polígonos que son representados por una colección de triángulos en sus equivalentes puntos bidimensionales y rellenar los tríangulos resultantes con un color. Las transformaciones necesarias para convertir los puntos de la geometría generalmente son multiplicaciones de matrices. Las transformaciones principales son la traslación, el escalamiento, la rotación y la proyección.

Para convertir un punto tridimensional es necesario homogeneizarlo agregando una variable extra y después multiplicar ese punto homogeneizado por una matriz de 4x4. La mayoría de los sistemas basados en rasterización guardan una pila de transformaciones que se va aplicando a cada vértice, teniendo como resultado una complejidad lineal (O(n)) donde n es el número de vértices.

2.3. *Ray tracing*

Como ya se ha dicho en el capítulo anterior, el algoritmo de *ray tracing* puede datar de los modelos de Albertch Dürer. Sin embargo, es más sencillo ver a *ray tracing* como un modelo de cámara oscura (*pinhole camera*). En la figura 2.1 podemos ver el modelo en cuestión que consiste en tener una película fotográfica encerrada en una caja y aislada de cualquier luz; esta caja tiene un agujero pequeño al otro extremo de donde se encuentra la película que dejará entrar los rayos de luz del exterior, plasmando en la película el color reflejado por los rayos de luz. Esto produce un fenómeno parecido a lo que experimenta el ojo humano; este fenómeno es la capacidad de captar una infinidad de rayos de luz reflejados en los objetos que pasan hacia la retina, teniendo como resultado una imagen de todo aquello que podemos ver.

Por conveniencia, en la computación gráfica, el modelo de la cámara oscura ha sido modificado por el expuesto en la figura 2.2. Podemos observar que en este modelo se toma un punto de vista (Ojo) como el agujero de la cámara obscura, un plano de proyección como la película fotográfica y finalmente una media pirámide o tronco que es donde se presenta la escena. Como podemos observar, es el mismo modelo ya que podría decirse que solo se cambio la posición de la película fotográfica.

Una implementación computacional de este modelo sería ineficiente, ya que se

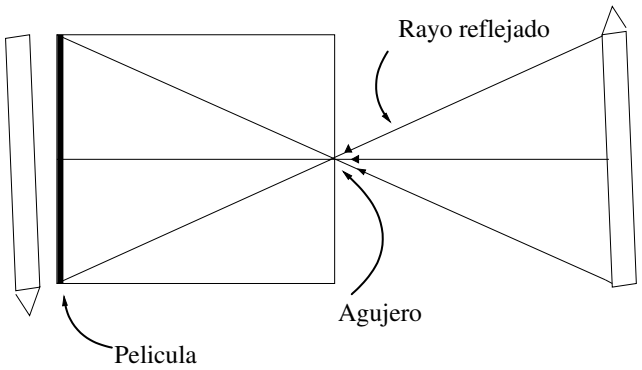


Figura 2.1: Modelo de la cámara oscura.

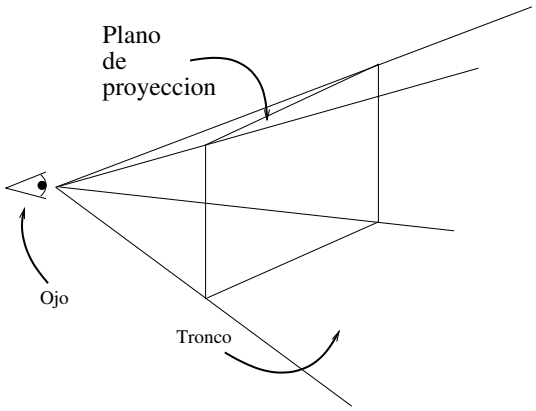


Figura 2.2: Modelo modificado de la cámara oscura.

tendrían que simular la trayectoria de miles de millones (o más) rayos de luz provenientes de la escena hacia la película fotográfica, o en este caso hacia la imagen resultante. A esto se le conoce comúnmente como *forward ray tracing*; afortunadamente, existe otra manera de utilizar el modelo modificado de la cámara oscura. Considerando que cada rayo de luz pasa de la escena a una región específica de la imagen, podríamos trazar el rayo de forma inversa. Es decir, de un punto de vista hacia la escena siguiendo la trayectoria de este y encontrando la intersección más cercana con algún objeto de la escena para averiguar de donde proviene la luz reflejada; este método es conocido como *backward ray tracing* y es el enfoque que ha predominado, haciendo de lado a *forward ray tracing*. Entonces cuando alguien se refiere al algoritmo de *ray tracing*, en realidad está hablando del *backward ray tracing*.

2.4. *Backward ray tracing*

Como vimos en la sección anterior, *backward ray tracing* consiste en proyectar rayos desde un punto de vista a través de un plano de proyección hacia una escena. Proyectar los rayos significa encontrar el objeto más cercano que se interpone en el camino del rayo, es decir, encontrar la intersección rayo-objeto más cercana al plano. Este proceso es posible haciendo un análisis vectorial de lo que sucede físicamente. Los rayos pueden ser representados si se tiene su origen y su dirección de la siguiente manera:

$$R_{origen} = R_o = [X_o, Y_o, Z_o]$$
$$R_{dirección} = R_d = [X_d, Y_d, Z_d]$$
$$y\ X_d^2 + Y_d^2 + Z_d^2 = 1\ (la\ dirección\ está\ normalizada)$$

y entonces el rayo queda definido por la siguiente ecuación:

$$R(t) = R_o + R_d * t \qquad t > 0.$$

El parámetro t de esta ecuación representa la distancia que recorre el rayo, es decir, la distancia a la que es proyectado. Es importante que la dirección sea normalizada ya que es en términos de la longitud de esta que la distancia es recorrida. Viendo la formula podemos deducir que el problema de hallar la intersección más cercana se reduce a hallar una distancia t tal que el rayo desplazado cubra una distancia mínima, cabe destacar que si $t < 0$, el objeto no se encuentra en el camino del rayo o hasta puede ser que se encuentre atrás del plano de proyección.

Una vez que es hallada la intersección más cercana, pueden proceder las siguientes acciones:

- Son proyectados rayos secundarios hacia las fuentes de iluminación dentro de la escena para averiguar si la luz alcanza a tocar al objeto en cuestión o es obstruida por algún otro objeto.

- Si el objeto tiene un grado de reflexión (como un espejo) se proyecta un rayo en dirección y ángulo de incidencia opuestos según el vector normal de la superficie que tendrá un grado de contribución al color del rayo.

- Si el objeto tiene un grado de refracción (como un cristal) se proyecta un rayo en la misma dirección pero desviado en un ángulo de refracción según el vector

normal y el grado de refracción de la superficie del objeto que tendrá un grado de contribución al color del rayo.

De estas acciones, la primera siempre se realiza y las otras dos pueden o no suceder. Es necesario mencionar que estas acciones también se realizan por cada rayo que surge dentro del proceso, llevándonos a la conclusión de que es un algoritmo recursivo. Sin embargo, la contribución de cada rayo al color final va disminuyendo con cada nivel de recursividad y es por eso que generalmente se fija un umbral de paro.

En la figura 2.3 se ejemplifican los casos mencionados del proceso de *backward ray tracing*. La escena está compuesta por un plano semitransparente, dos esferas (una reflejante y otra normal) y dos fuentes de luz; entonces es proyectado el rayo **R0** hacia el plano, al chocar con el plano se proyectan los rayos **S1** y **S2** para detectar las contribuciones de luz y las sombras reflejadas, pero en el caso de **S2** el rayo es bloqueado por una esfera, es decir que la luz es bloqueada por dicho objeto y no contribuye en el rayo principal. Debido a que el plano es semitransparente, se tiene un grado de refracción el cual produce que se proyecte el rayo **T0** en dirección a la esfera metálica y del mismo modo que en el plano se proyectan dos rayos hacia las fuentes de luz (**S3** y **S4**) y ya que la esfera es metálica, tiene un grado de reflexión que provoca el proyectar un tercer rayo **T1** el cual ya no intersecta algún otro objeto y termina la trayectoria del rayo principal. **T0** va a contribuir en un cierto grado el color final del píxel pues es un rayo que fue generado a partir del rayo **R0** gracias a la naturaleza del material del plano.

La acción de buscar la intersección más cercana por cada rayo de la imagen y por cada objeto dentro de la escena representa una pérdida de tiempo muy grande, ya que la mayoría de las veces se encuentra que el rayo no intersecta al objeto. Por ejemplo, supongamos que la escena en cuestión contiene un objeto subdividido en una gran maya de cien mil triángulos de diversos tipos; entonces cada rayo es proyectado y verifica todos y cada uno de los cien mil triángulos en busca del que se encuentre más cerca, en el peor de los casos, esto conlleva cien mil verificaciones que acaban por consumir demasiado tiempo, pues para verificar si existe o no una intersección son necesarias un conjunto de operaciones vectoriales costosas como producto punto y producto cruz. Por esta razón, fueron inventadas varias técnicas aceleradoras de ray tracing y las más notorias son las estructuras aceleradoras como *Octrees*[11], *KD trees*[12] y *Bounding Volume Hierarchy*[13].

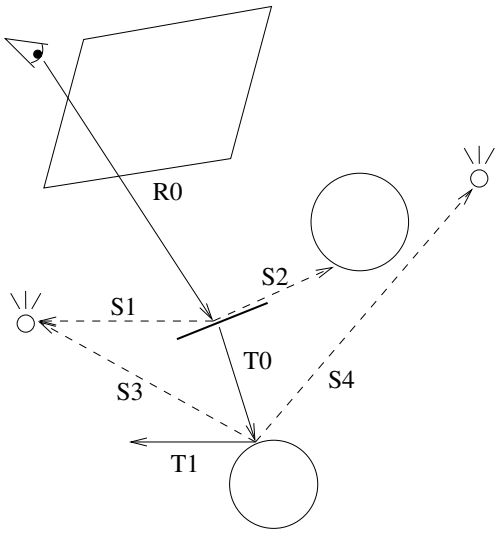


Figura 2.3: Ejemplo de *Backward ray tracing*.

2.5. Comparativa entre *ray tracing* y rasterización

Como ya se ha mencionado en el capítulo anterior, el algoritmo básico de *ray tracing* tiene una complejidad de $O(n^2)$, si comparamos esta complejidad con la de la rasterización básica podríamos concluir que el mejor algoritmo para la sintesis de imágenes 3D es la rasterización. Sin embargo, las imágenes producidas por este algoritmo presentan errores de iluminación y sombreado que con *ray tracing* no son un problema debido a la naturaleza del algoritmo.

La rasterización tiene un consumo de tiempo mucho menor que *ray tracing*, pero por otra parte, el constante avance tecnológico ha permitido el desarrollo de soluciones que reducen el tiempo de ejecución de *ray tracing*. Tal es la reducción de tiempo de ejecución, que algunas compañías han optado por usar *ray tracing* como opción en la creación de contenidos digitales visuales.

2.6. Estructuras Aceleradoras

En la sección anterior mencionamos que existen varias estructuras aceleradoras las cuales disminuyen notablemente el tiempo de ejecución y facilitan la tarea de encontrar la intersección más cercana con el rayo y la escena. Esta sección explica las siguientes estructuras *Octrees*, *KD trees* y *Bounding Volume Hierarchy*. Estas tres estructuras están basadas en la división espacial de la escena, cabe mencionar que las

tres son estructuras de árbol y esto hace que el proceso de ray tracing disminuya su complejidad al orden logaritmico.

2.6.1. *Octrees*

El *octree* es una estructura de árbol, en la cual cada nodo interno tiene exactamente ocho nodos hijos. Esta estructura divide espacios tridimensionales en ocho partes iguales también llamados "octetos", de ahí el nombre. En cada nodo intermedio del árbol se almacena solamente el punto central de cada octeto, mientras que en los nodos hoja se almacenan los puntos que se encuentren en su octeto. En la figura 2.4 podemos apreciar de una forma más clara cómo se comporta en cuanto a subdivisiones de la escena, suponiendo un cubo que sirve como caja delimitadora de la escena.

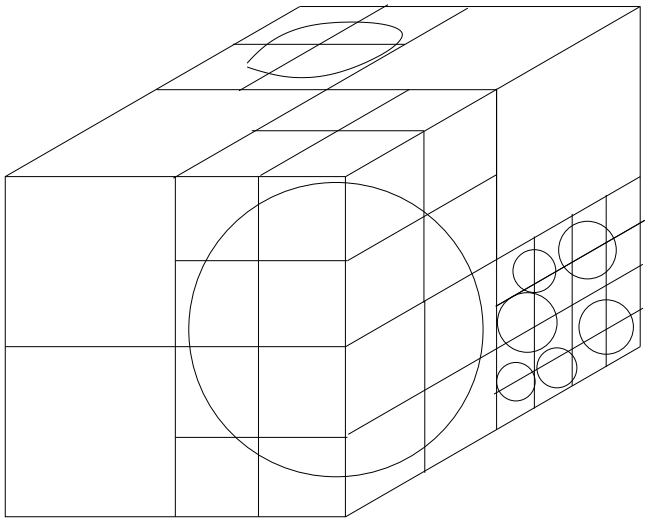


Figura 2.4: Ejemplo de subdivisiones realizadas en una escena compuesta de varias esferas para un *octree*.

Esta estructura es útil, ya que a medida que se va recorriendo el árbol se van descartando todos los nodos hijos de los nodos padre que no fueron seleccionados, reduciendo así el número de verificaciones en busca de la intersección rayo-objeto más cercana.

2.6.2. *KD trees*

Un árbol KD es la representación del espacio dividido de la escena; los nodos internos representan los ejes de corte y los nodos hoja representan las secciones en las que fue

dividido el espacio. La forma clásica para el manejo de árboles KD es dado un rayo, un árbol y un rango delimitador, se recorre el árbol hasta encontrar una intersección del rayo con un nodo hoja el cual es almacenado en una cola y se sigue recorriendo el árbol en busca de más intersecciones; en caso de encontrar otra intersección, esta es encolada y así sucesivamente. Finalmente se buscan intersecciones entre objetos contenidos en los nodos hoja encolados y el rayo, tomando la primera intersección como el resultado del rayo. La figura 2.5 muestra la particion espacial del que se habla y su respectivo árbol.

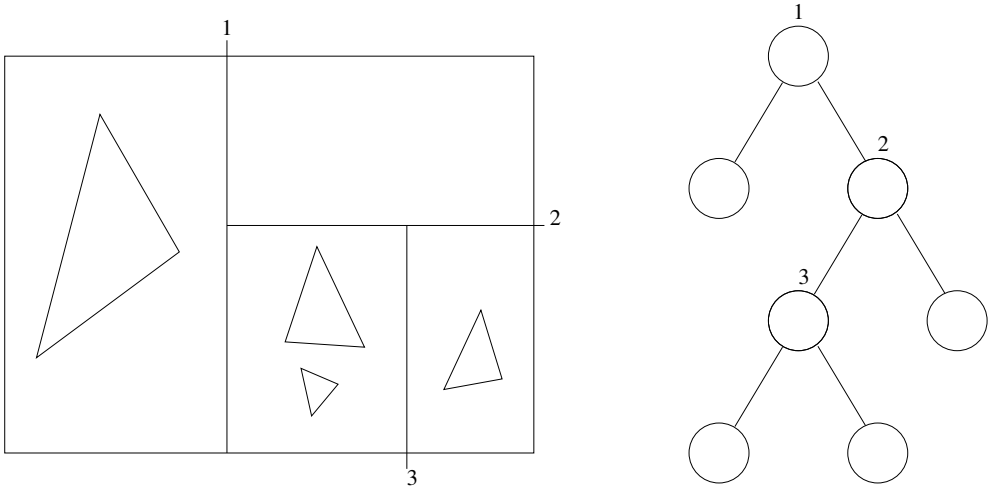


Figura 2.5: Ejemplo de subdivisiones efectuadas en una escena compuesta de varios triángulos para un *KD tree*.

2.6.3. *Bounding Volume Hierarchy*

Bounding Volume Hierarchy (BVH) es una técnica que también está basada en el uso de estructuras de árbol, donde los nodos hoja son objetos geométricos de la escena envueltos por volumen y cada nodo intermedio hace referencia a un volumen envolvente de los nodos hijos. Estos volúmenes son obtenidos con una división espacial 3D similar a la de los *octrees* pero más simple, ya que los objetos no son subdivididos a detalle. En la figura 2.6 se muestra de manera gráfica este tipo de estructura.

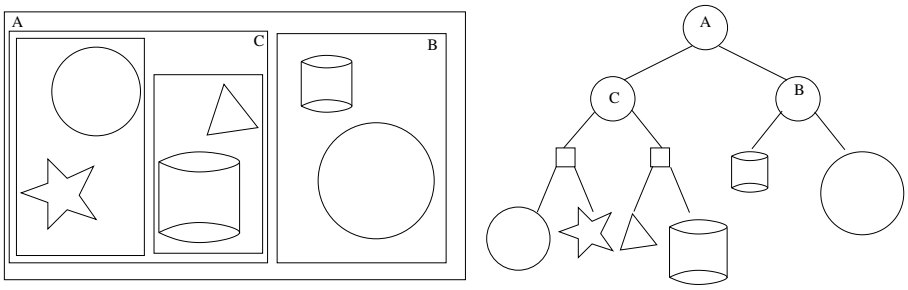


Figura 2.6: Ejemplo de divisiones hechas en una escena compuesta de varias figuras donde se emplea BVH; a la derecha se muestra el árbol que se forma con esta subdivisión.

2.7. Trabajos relacionados

Con el paso del tiempo se han diseñado mejoras para hacer más eficiente el algoritmo, estas van desde el desarrollo de estructuras de datos y algoritmos para determinar de manera eficaz la intersección de los rayos con los objetos hasta la invención de hardware dedicado. La aparición de nuevas tecnologías ha llevado a *ray tracing* a modelos paralelos basados en CPUs multicore, clustering y últimamente GPUs y sistemas híbridos. Entre los trabajos realizados con GPUs encontramos los trabajos [14] y [15] que proponen el uso de una estructura de datos conocida como árbol KD, esta estructura no es nueva a *ray tracing* pero a causa de la arquitectura en GPUs fue rediseñada primero por Tim Foley [14] y mejorada por Stefan Popov [15]. Foley propone reiniciar el recorrido del árbol disminuyendo el rango delimitador cada que el rayo intersecte un nodo hoja pero no intersecte un objeto. El problema principal de Foley fue el uso de una pequeña pila que almacenaba los nodos hoja y esto aunado al número de reinicios que tenía que hacer disminuía el desempeño de su algoritmo. Por otro lado, Popov propuso una cambio en la estructura de árbol y ese cambio era usar ligas entre nodos del mismo nivel que fueran adyacentes en el espacio dividido. Esto evita el reinicio del recorrido ya que el rayo utiliza esas ligas para moverse entre nodos siguiendo la misma lógica de Foley (buscar intersecciones rayo objeto) y si llega a un nodo que está ligado a un nodo nulo, quiere decir que está situado en una esquina y el rayo entonces es terminado. El principal problema de la propuesta de Popov radica en el uso extra de memoria que como sabemos es muy limitada dentro de las GPUs, es decir que Popov sacrificó memoria por desempeño. En relación al hardware dedicado encontramos un artículo de Woop que propone el uso de una FPGA exclusivamente para el proceso de *ray tracing* [16] programable.

En referencia a *ray tracing* en tiempo real Carsten Benthin propone el uso de CPUs para realizarlo en su trabajo [17], la desventaja que tiene esta tesis es que se apoya en tener código optimizado para arquitecturas especificas de procesadores y esto limita las capacidades del algoritmo. Un año más tarde, Popov propone el uso de GPUs [18], para implementar el algoritmo en tiempo real utilizando BVH. En su artículo, Popov concluye que los árboles KD siguen siendo la mejor estructura para acelerar el proceso de *ray tracing* aunque las jerarquías de volumen envolvente alcanzaron un gran desempeño sobre las GPUs. En el 2009, Shin se basa en el trabajo de Popov sobre árboles KD para proponer una solución a *ray tracing* en tiempo real [19]. Shin utilizó varios métodos existentes (tanto de estructuras de datos como algoritmos de intersección) para encontrar una combinación que superara los resultados obtenidos en años pasados.

Además, Nvidia desarrolló un motor programable de *ray tracing* llamado *OptiX*®, la idea principal estaba basada en que es posible implementar algoritmos de *ray tracing* usando una pequeña cantidad de operaciones programables [20]. *OptiX*® es una buena alternativa para implementar varias algoritmos de *ray tracing* que sigue en constante desarrollo.

Capítulo 3

Tecnologías de programación paralela heterogénea

Debido a la naturaleza del problema a atacar con esta tesis, es necesario escoger de manera correcta las herramientas que se utilizaran para el desarrollo de la misma.Dentro de este capítulo se explican las arquitecturas *multicore* y arquitecturas de GPUs. Dentro de dichas arquitecturas se expone el hardware actual, la plataforma de desarrollo necesaria y el diseño de algoritmos para ambos casos.

3.1. Arquitecturas paralelas

Esta sección presenta los fundamentos de la programación paralela. Dichos fundamentos están divididos en dos, la clasificación de problemas que pueden ser paralelizados (taxonomía de Flynn) y la clasificación de plataformas de trabajo (arquitecturas simétricas y asimétricas).

3.1.1. Taxonomía de Flynn

Para el desarrollo de aplicaciones paralelas en arquitecturas *multicore* se han diseñado varias soluciones. La elección correcta entre estas soluciones depende del problema a paralelizar, una forma de identificar el tipo de problema que se tiene es usando la taxonomía de Flynn [21]. La taxonomía de Flynn divide en cuatro arquitecturas los tipos de problemas dependiendo del número de instrucciones y datos diferentes que se tienen. La tabla 3.1 presenta dicha taxonomía.

	Una instrucción	Múltiples instrucciones
Un dato	SISD	MISD
Múltiples datos	SIMD	MIMD

Tabla 3.1: Taxonomía de Flynn

A continuación se explica la tabla anterior:

- **SISD (*Single Instruction, Single Data).*** Son aquellos programas secuenciales donde no es posible el cómputo paralelo.
- **SIMD (*Single Instruction, Multiple Data).*** En este caso, se tienen varios flujos de datos procesados por una sola instrucción de manera paralela. Un ejemplo clásico es un procesador vectorial donde se tienen varios datos de entrada que son procesados simultáneamente.
- **MISD (*Multiple Instruction, Single Data).*** Este tipo de arquitectura es poco común, pero es utilizado cuando se requiere un paralelismo redundante como respaldo. En este caso se tienen operando varias instrucciones sobre el mismo conjunto de datos replicados.
- **MISD (*Multiple Instruction, Multiple Data).*** Este tipo de arquitectura se maneja cuando se tienen múltiples procesadores ejecutando simultáneamente diferentes instrucciones sobre diferentes conjuntos de datos. Los sistemas distribuidos son un ejemplo claro de este tipo de arquitectura.

3.1.2. Arquitecturas simétricas y asimétricas

El multiprocesamiento se puede dividir también en dos tipos de arquitecturas según la configuración del hardware, estas son simétrica y asimétrica. La principal diferencia entre estas dos arquitecturas es el tipo de tareas que realiza cada núcleo. Por una parte, una arquitectura simétrica propone tener núcleos con las mismas capacidades y en contra parte, una arquitectura asimétrica es aquella en la que los núcleos son dedicados a tareas específicas como procesamiento de video, procesamiento de sonido, entre otras.

Una arquitectura simétrica consiste en dos o más procesadores o núcleos idénticos conectados a una única memoria compartida. Cada procesador puede ejecutar diferentes programas de manera independiente y compartir información simultáneamente

entre los demás. También las interfaces son compartidas con todos los procesadores, dígase entradas y salidas. La ventaja de este esquema es que, idealmente, se distribuye la carga de trabajo de una manera uniforme de tal forma que ninguno este inactivo.

Una arquitectura asimétrica es aquella en la que cada núcleo es dedicado a tareas específicas, es decir que, al contrario que una arquitectura simétrica, las interfaces son controladas individualmente. Sin embargo, si puede existir una comunicación entre procesadores.

3.2. Arquitecturas de procesadores *multicore*

En el mercado actual existen varios tipos de procesadores que integran múltiples núcleos, a estos se les conoce como procesadores *multicore*. Tener varios núcleos en un procesador tiene varias ventajas y desventajas. La ventaja principal es la capacidad de procesamiento paralelo, es decir, llevar a cabo tantos procesos como núcleos se tengan en un mismo tiempo; por otra parte, la mayor desventaja es el manejo al acceso de memoria (principalmente en problemas de competencia). Dentro de los procesadores *multicore* podemos encontrar a varios fabricantes, de los cuales los principales son AMD, Intel y ARM.

3.2.1. Intel

La mayoría de los procesadores *multicore* de intel siguen una arquitectura simétrica que manejan una estructura de árbol invertido en sus niveles de caché. La figura 3.1 muestra una arquitectura clásica que ha venido utilizando Intel para el diseño de sus procesadores *multicore*, podemos observar que cada núcleo tiene un nivel de memoria caché propio para la manipulación de variables locales, después cada dos núcleos tienen como canal de comunicación un segundo nivel de caché y finalmente puede o no existir un tercer nivel de caché que este conectado directamente con un bus de comunicación encargado de gestionar la memoria del sistema.

Este tipo de arquitectura tiene como ventaja que el acceso a la memoria del sistema es de de forma uniforme y sirve muy bien para sistemas de memoria compartida, sin embargo, el uso de esta no es recomendable para sistemas de memoria distribuida.

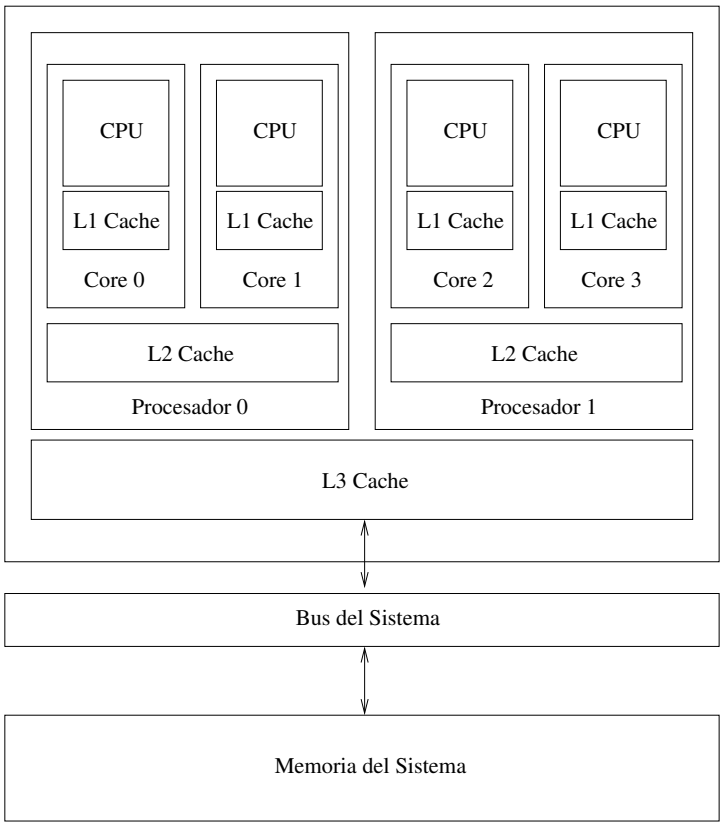


Figura 3.1: Ejemplo de una arquitectura *multicore* convencional de Intel.

3.2.2. AMD

Los procesadores *multicore* usuales de AMD son de arquitectura simétrica siguiendo un esquema diferente al de Intel. Ellos diseñaron una arquitectura donde cada núcleo tiene asociado a si una parte específica de la memoria del sistema. Los núcleos tienen un canal de comunicación directa entre si, puede ser el caso que exista una comunicación total o parcial, es decir, que todos los núcleos estén conectados (todos con todos) o que tengan otro tipo de conexión (en serie por ejemplo). La figura 3.2 muestra la arquitectura propuesta por AMD.

El uso de esta arquitectura es recomendable para sistemas de memoria distribuida, sin embargo, para sistemas de memoria compartida el desempeño no es el óptimo.

3.2.3. ARM

Los procesadores *multicore* de ARM son de bajo consumo energético. Es por esto que la mayoría de estos procesadores son utilizados en dispositivos móviles y recientemente

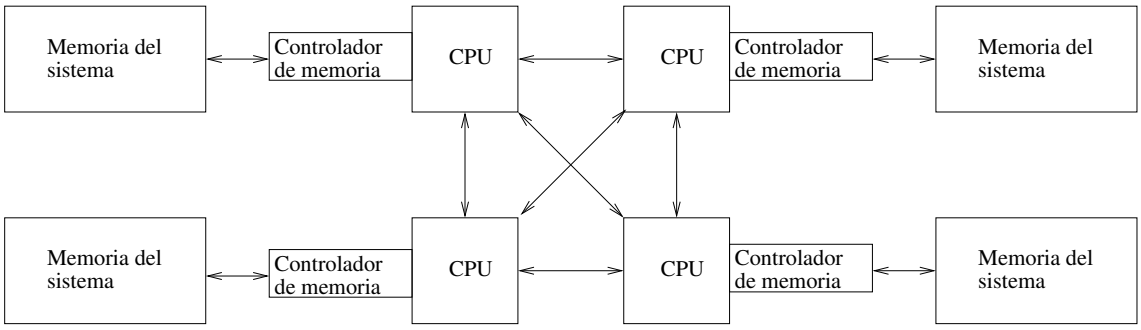


Figura 3.2: Ejemplo de una arquitectura *multicore* convencional de AMD.

en diseños experimentales de servidores de bajo consumo energético. Los procesadores *multicore* más famosos de ARM son los Cortex A9, A8 y Tegra. Mientras que los procesadores Cortex A9 y A8 son del tipo simétrico, los procesadores Tegra son del tipo asimétrico, teniendo unidades de capacidades diferentes dentro de un mismo chip. Los procesadores cortex A9 y A8 están basados en un procesador de 32 bits con una arquitectura ARM en su versión 7. Además cada procesador cuenta con una caché de datos y otra de instrucciones de 16 hasta 64 kilobytes.

La figura 3.3 muestra la arquitectura de un procesador ARM Cortex A9. Esta arquitectura presenta dos niveles de memoria caché utilizando una estructura de árbol invertido. Por el momento, estos procesadores pueden ser de uno a cuatro núcleos.

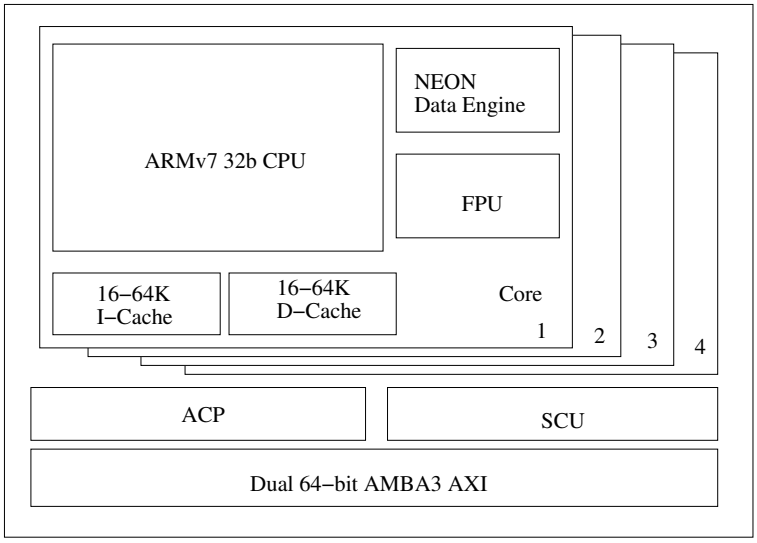


Figura 3.3: Arquitectura *multicore* ARM Cortex A9.

Como ya se ha mencionado, los procesadores Tegra son del tipo asimétrico e

incorporan el uso de GPUs, específicamente GPUs de Nvidia, en la figura 3.4 se muestra la arquitectura que utilizan.

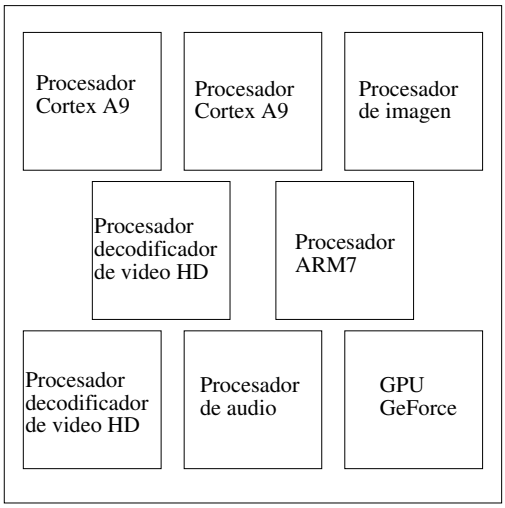


Figura 3.4: Arquitectura *multicore* ARM Tegra.

3.2.4. Herramientas de desarrollo en procesadores *multicore*

A continuación se exponen algunas de las herramientas que sirven de apoyo para la programación paralela en procesadores *multicore*:

- **Hilos.** Un hilo es la unidad más pequeña de procesamiento que puede ser planificada por un sistema operativo y es posible crear varios hilos para realizar varias tareas al mismo tiempo; hay que tomar en cuenta que el número real de hilos que se ejecutan simultáneamente es igual al número de núcleos del procesador, los demás hilos son ejecutados de manera concurrente según el planificador de tareas del sistema operativo. El uso de hilos implica saber el uso de semáforos o candados y programación concurrente. Existen varias implementaciones de hilos, tales como los hilos de POSIX [22], TBB (*Thread Building Blocks*) [23] de Intel, hilos de Java [24], entre otros.

- **OpenMP.** Es una interfaz de programación de aplicaciones (API) capaz de manejar memoria compartida con múltiples procesos. OpenMP esta compuesta por directivas de compilador, rutinas de biblioteca y variables de entorno que influyen en el comportamiento en tiempo de ejecución. Esta interfaz es simple,

flexible y escalable. Es flexible ya que el usuario puede o no realizar una sincronización de procesos parcial. Es simple puesto que su sintaxis es fácil de recordar y es escalable ya que puede usarse desde computadoras de escritorio hasta supercomputadoras (con el uso conjunto de otras herramientas como OpenMPI). Existen también varias implementaciones de OpenMP de las cuales depende su eficiencia.

- **OpenCL.** Es un lenguaje de programación y una extensión de C que permite la programación paralela en múltiples plataformas. Su principal característica es que puede ejecutarse de manera transparente, es decir que no importa la arquitectura, ya sean procesadores AMD, Intel, GPUs o una combinación de CPUs y GPUs. OpenCL no es tan eficiente debido a la gama de arquitecturas que debe manejar, lo cual atrofia la capacidad de computo en casos de programación heterogénea.

3.2.5. Arquitecturas *multicore* de caché

Como se menciono al inicio de este capítulo, es necesario hacer una selección correcta de las herramientas para el desarrollo de esta tesis. Dentro de estas herramientas, la importancia del procesador recae en el esquema de memoria caché que maneja ya que esto repercute en el acceso a datos por parte del CPU que pueda existir dentro de la solución propuesta. A continuación se discuten cuatro de las arquitecturas actuales de caché con el proposito de aclarar porque usar procesadores Intel para la propuesta de solución.

Árbol invertido

Esta arquitectura es la utilizada por la mayoría de los procesadores *multicore* de intel. Propone un esquema de comunicación en distintos niveles de caché, desde la memoria caché propia de cada núcleo hasta una memoria cache compartida por todos los nucleos que sirve de canal de intercambio de datos. Cada nivel de memoria caché esta conectado directamente al siguiente nivel de memoria, es decir que no es necesario que cuando se requiera una copia de datos en la memoria cache de otro núcleo estos pasen por el núcleo como se ve a continuación en la arquitectura basada en *Crossbar*. Este detalle resulta importante ya que la copia de datos en los distintos niveles de caché es directa. La figura 3.1 mostrada anteriormente muestra un ejemplo de este tipo de arquitectura..

Crossbar

En esta arquitectura cada núcleo tiene su propia memoria caché y el intercambio de datos se hace a través de un canal de comunicación entre los procesadores, es decir que el intercambio de datos en la memoria caché es de la siguiente manera, primero los datos se pasan de la memoria caché a un núcleo, despues estos datos son pasados por un canal de comunicación entre núcleos al otro núcleo donde son requeridos y finalmente se copian a la memoria caché de dicho núcleo. Como se vio en la subsección anterior la mayoría de los procesadores AMD utilizan este tipo de arquitectura. En la figura 3.2 presentada anteriormente, se puede ver este tipo de arquitectura.

Arquitectura de caché de MIC

La arquitectura MIC es la más nueva propuesta de arquitectura de Intel, está propone que cada nucleo tenga un nivel interno de memoria caché y haya un segundo nivel de memoria caché comunicado en una especie de red con topología *token ring* doble a la cual se conecta cada núcleo. La figura 3.5 representa esta arquitectura de memoria caché. A primera vista, esta arquitectura parecería ineficiente, pero si se toma en cuenta la velocidad en la que trabaja la red *token ring* interprocesador nos podemos dar cuenta de que la memoria cache es bien manipulada.

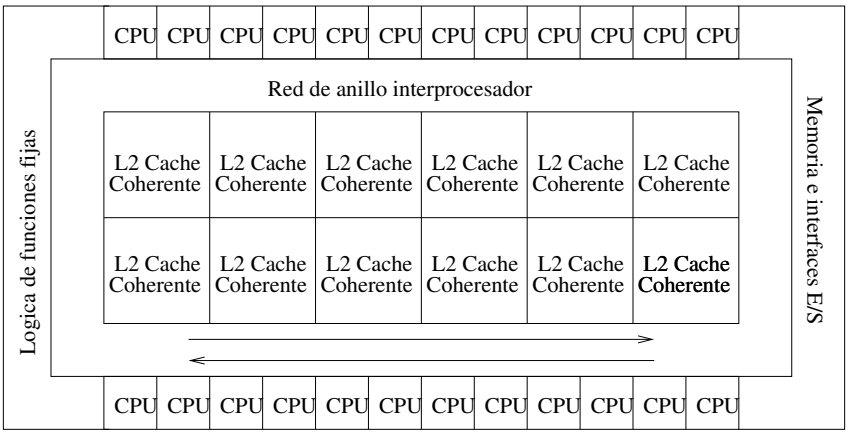


Figura 3.5: Arquitectura MIC.

3.3. Arquitecturas de GPUs

Los avances tecnológicos han hecho posible utilizar las tarjetas gráficas como recurso de cómputo. Una GPU, como un procesador *multicore*, está compuesta por una cantidad superior de núcleos que una CPU y esto nos brinda una capacidad superior de cómputo, mas está capacidad se ve limitada a la programación paralela. Dentro de los principales fabricantes de GPUs se encuentran Nvidia y AMD (con ATI). En esta sección damos a conocer un poco de las arquitecturas que proponen ambos fabricantes, una comparación más a fondo entre ambas arquitecturas se puede encontrar en [25].

3.3.1. Nvidia

Nvidia propone una arquitectura para cómputo paralelo masivo ofreciendo una gran cantidad de unidades de procesamiento paralelo. Está arquitectura es conocida como *Compute Unified Device Architecture* (CUDA). CUDA divide sus unidades de procesamiento en conjuntos de bloques de hilos. Además presenta una jerarquía de memoria dividida en tres niveles, en la figura 3.6 se puede apreciar dicha jerarquía. Cada hilo de ejecución tiene su propia memoria local y privada, es decir que solo puede ser accedida por el hilo. Un nivel más arriba existe la memoria compartida entre bloques de hilos, esta memoria es accedida por un conjunto de hilos que comparten el mismo bloque. Finalmente esta la memoria global que puede ser accedida por grupos de bloques de hilos llamados rejillas (*grids*).

En cuanto a las unidades de procesamiento, CUDA provee la unidad *Streaming Multiprocessor* (SM) que esta compuesta por muchos núcleos (el número depende de la arquitectura por ejemplo en la arquitectura *Fermi* tiene 16 SMs con 32 núcleos cada uno [26] y la arquitectura *Kepler* contiene 15 SMs con 192 núcleos cada uno [27]), que a su vez están compuestos por una unidad de punto flotante y una unidad aritmetica lógica. En la figura 3.7 se muestra la estructura de un SM.

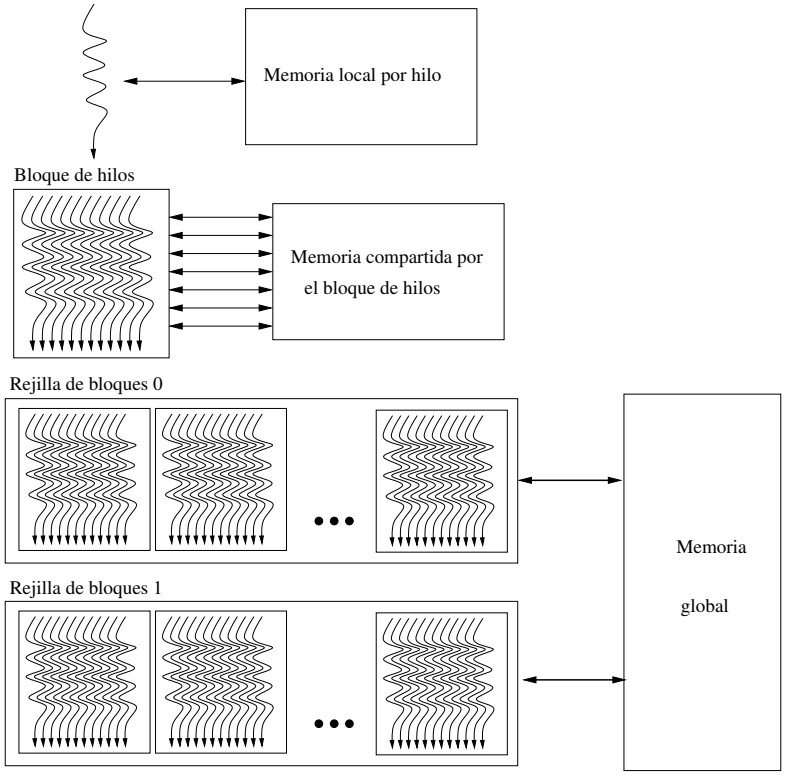


Figura 3.6: Distribución de la memoría en CUDA.

Como se ve en la figura 3.7 un SM también incluye otros componentes como unidades de carga y almacenamiento de datos (LD/ST) y unidades de funciones especiales (SFU) que permiten la ejecución de funciones como seno, coseno, reciproco y raiz cuadrada. Para el manejo de hilos, un SM tiene un planificador y despachador de hilos. El archivo de registros que se muestra en la figura es dividio entre los núcleos del SM y sirven como la memoria local de cada hilo. Lo que hace de esta arquitectura una opción muy poderosa en el campo del cómputo de alto desempeño.

Finalmente, la figura 3.8 muestra la interconexión que hay de los SMs y la memoria de la tarjeta gráfica y el canal de comunicación entre estos es un nivel de memoria caché. También se muestra la interfaz con la CPU que en este caso es BUS PCI.

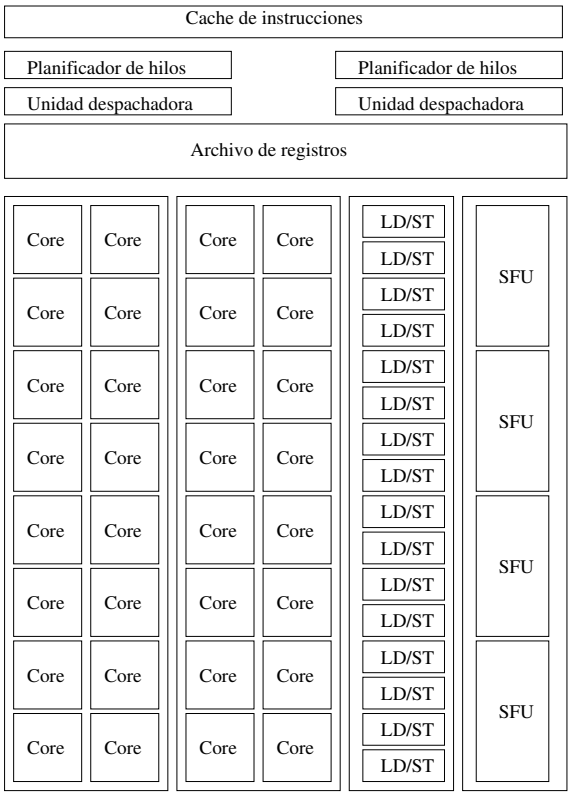


Figura 3.7: Estructura de un SM.

3.3.2. AMD ATI

Por su parte ATI también ofrece una arquitectura de GPUs capaz de realizar cómputo paralelo. Esta arquitectura esta basada en tener una comunicación total entre unidades de procesamiento, es decir tener un canal entre cada una. En la figura 3.9 se observa la arquitectura de una tarjeta gráfica ATI con arquitectura *Cypress*. Esta arquitectura generalmente esta compuesta de 20 unidades SIMD con 16 hilos cada uno, basicamente una SIMD es equivalente a un SM de CUDA.

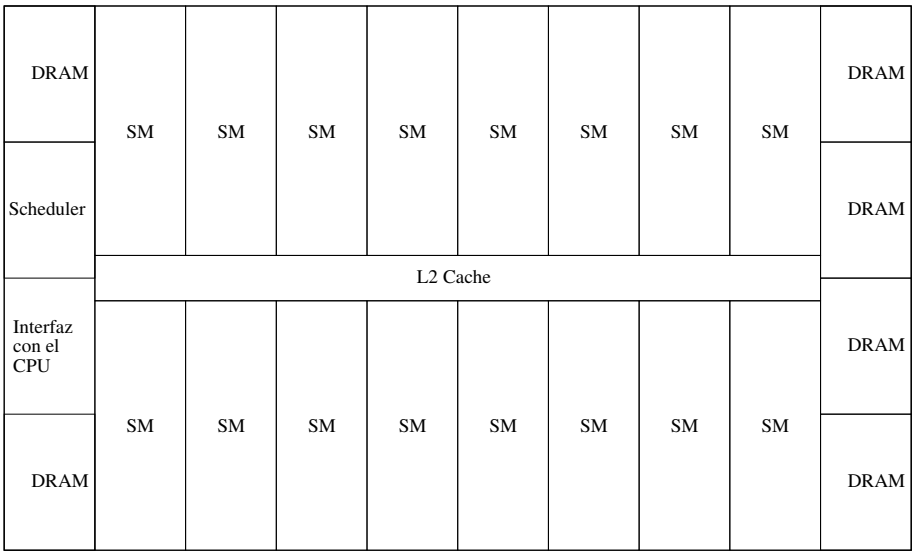


Figura 3.8: Interconexión de SMs.

3.3.3. Herramientas de desarrollo en GPUs

A continuación se presentan algunas herramientas que permiten la programación bajo GPUs:

- **CUDA (Compute Unified Device Architecture).** Es la plataforma de cómputo paralelo y modelo de programación inventado por Nvidia. Además, CUDA es una extensión al lenguaje C, C++ y fortran. CUDA permite el desarrollo de aplicaciones paralelas en GPUs y es de uso exclusivo para GPUs de Nvidia.

- **DirectCompute.** Es una API de Microsoft que apoya el cómputo de propósito general en GPUs bajo Windows Vista, Windows 7 y Windows 8. DirectCompute forma parte de las APIs de DirectX.

- **BrookGPU.** Es un compilador e implementación del lenguaje Brook creado por el laboratorio de gráficos de la universidad de Stanford. Este proyecto tiene como objetivos, demostrar el uso de GPUs como plataforma de desarrollo de propósito general e investigar el modelo computacional del lenguaje de programación Brook sobre GPUs y la implementación de sistemas.

- **OpenCL.** Mencionado en la sección anterior.

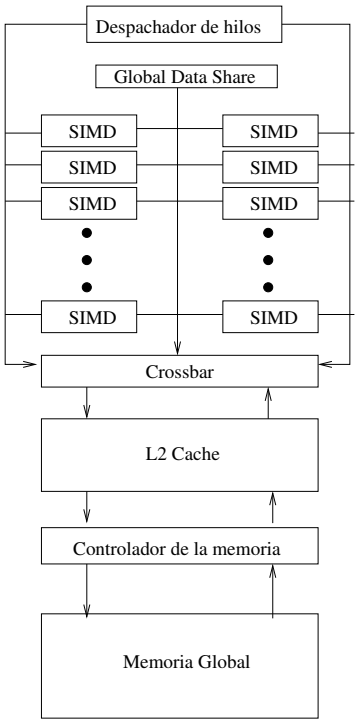


Figura 3.9: Arquitectura de GPUs de ATI.

3.4. Clústers de *High Performance Computing* (HPC) Heterogéneos

En esta sección se habla de la situación actual de los clústers que existen para realizar HPC heterogéneo. Para esta sección revisamos los primeros lugares de las listas top500 [28] y green500 [29]. La lista del top500 está basada en el poder computacional que se tiene, mientras que el green500 está basada en el consumo energético. En las tablas 3.2 y 3.3 se listan las supercomputadoras preparadas para cómputo paralelo heterogéneo que se encuentran en los diez primeros lugares de ambas listas. Como podemos observar, apenas existen muy pocas supercomputadoras de HPC que tengan opción para el cómputo paralelo heterogéneo. A pesar de el bajo consumo energético que representa utilizar GPUs, la mayoría de los clústers existentes siguen siendo basados en CPUs. Esto puede ser el resultado del poco conocimento que se tenga de programación con GPUs o por la costumbre de utilizar CPUs como herramienta principal.

Top500		
Supercomputadora	TFlops	Energía(kW)
Titan - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x	17590.0	8209.0
Tianhe-1A - NUDT YH MPP, Xeon X5670 6C 2.93 GHz, NVIDIA 2050	2566.0	4040.0

Tabla 3.2: Supercomputadoras listas para cómputo paralelo heterogéneo(Top500).

Green500		
Supercomputadora	MFlops/W	Energía(kW)
Eurora - Eurotech Aurora HPC 10-20, Xeon E5-2687W 8C 3.100GHz, Infiniband QDR, NVIDIA K20	3208.83	30.70
Aurora Tigon - Eurotech Aurora HPC 10-20, Xeon E5-2687W 8C 3.100GHz, Infiniband QDR, NVIDIA K20	3179.88	31.02

Tabla 3.3: Supercomputadoras listas para cómputo paralelo heterogéneo (Green500).

En la figura 3.10 se muestra un diagrama de una arquitectura clásica de cómputo paralelo heterogéneo usando GPUs. Se puede observar que la comunicación entre CPU y GPU es realizada a partir de directicas de control, es decir que se tienen comandos clave para ejecutar algún programa en GPU que comunmente son llamados *kernels*. Además, la comunicación en la memoria del sistema y la de la tarjeta gráfica es realizada por medio de la característica *direct memory access (DMA)* del sistema, lo cual permite copiar rapidamente datos en memoria del GPU.

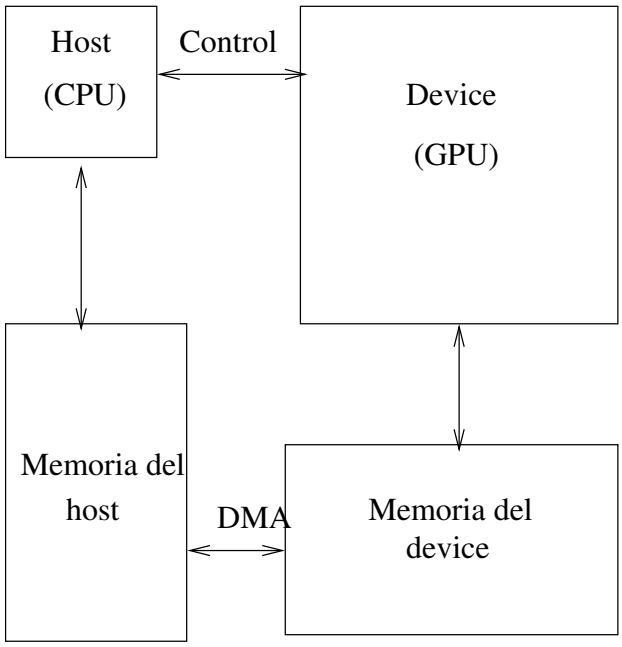


Figura 3.10: Esquema de una arquitectura para cómputo paralelo heterogéneo basado en GPUs.

Capítulo 4

Desarrollo

Como se ha comentado en capítulos anteriores, esta tesis tiene como objetivo la creación de un algoritmo de *ray tracing*. Este capítulo detalla el diseño de cada sección del algoritmo propuesto.

4.1. Analisis de estructuras aceleradoras

En el capítulo 2 describimos tres estructuras aceleradoras para el algoritmo de *ray tracing*: *octrees*, *KD trees* y *bounding volume hierarchies* (BVH). En esta propuesta se decidió utilizar la estructura *KD tree*. Está decisión fue tomada por eliminación, a continuación explicamos por qué no se optó por usar *octrees* ni BVH y las ventajas de usar *KD trees*.

Las principal desventaja de usar *octrees* recae en el uso de la memoria ya que por cada nuevo nivel de árbol se requieren forzosamente ocho nodos hijo de cada nodo que se vaya a subdividir. Esto conlleva un gran uso de memoria lo cual es limitado en la GPU. Además, los nodos hoja del *octree* pueden tener primitivas o no (al menos uno debe tener primitivas o partes de estas). Por ejemplo, si tenemos que en los ocho nodos hijo de una rama del *octree* solo uno de estos tiene primitivas las otras siete no contienen nada, es decir que hay un desperdicio de memoria. Este comportamiento es debido a que las divisiones de la escena se hacen de tal forma que queden ocho partes iguales. A pesar de esto, si es posible definir una estructura a partir de octrees si se utiliza una serie de identificadores de nodos, un bosquejo de esta estructura seria tener ocho identificadores por nodo y almacenar los nodos en memoria; aun así esta estructura requiere una gran cantidad de memoria.

Otra opción que se tuvo fue usar BVH, sin embargo, el uso de BVH conlleva

utilizar geometría constructiva de sólidos. La geometría constructiva de sólidos se refiere a construir los objetos de la escena con operaciones booleanas y objetos más simples como cubos, cilíndros, esféras, pirámides, entre otros. Este proceso es sencillo para construir escenas, pero el proceso contrario es mucho más difícil ya que puede existir una gran cantidad de opciones y hay que escoger una que pueda usarse para generar un árbol balanceado con el uso de BVH. Otra razón para no usar BVH es el trabajo de Popov [18] que concluye que los *KD trees* pueden ser una mejor opción.

Finalmente está la estructura *KD tree*, la cual si es bien diseñada se puede almacenar en un espacio menor al que ocuparia un *octree*. Construir un *KD tree* puede ser de diversas formas y es posible tener un árbol balanceado si se sigue un método correcto, como se explica en el transcurso de este capítulo. Además, el uso de *KD trees* en aplicaciones gráficas se ha estudiado más a fondo en el pasado obteniendo buenos resultados, lo que nos lleva a pensar que es una buena opción para lapropuesta de solución.

4.2. Propuesta de la solución

El objetivo principal de esta tesis es diseñar un algoritmo heterogéneo de *Ray Tracing* basado en *KD trees* capaz de aprovechar las propiedades de cómputo paralelo de las GPUs. Para diseñar esta propuesta seguimos la metodología de Ian Foster que establece ciertos criterios a considerar cuando se hace cómputo paralelo. En a siguiente sección se detalla esta metodología mientras que en las secciónes subsecuentes se explica como se usó la metodología en el desarrollo de esta tesis.

Construir *KD trees* en el CPU resulta sencillo debido a que se puede utilizar memoria dinámica, sin embargo, en GPU esto se vuelve más complicado ya que la memoria se debe manejar de manera estática. Es decir que los datos que se copian a la GPU deben ser de tamaño fijo y no es posible utilizar memoria dinámica en ese entorno. Al observar esto, nos encontramos con el problema de hallar una forma de manipular el *KD tree* de forma tal que quede de un tamaño fijo para poder copiarlo a la memoria de la GPU. Es por eso que en esta tesis se desarrollo una método de preprocesamiento para poder utilizar árboles creados con CPU en GPU.

4.3. Diseño de Algoritmos Paralelos

Existen una metodologías que cubre los detalles que hay que tomar en cuenta a la hora de diseñar un algoritmo paralelo. Esta metodología es conocida como metodología de Ian Foster [30], a continuación exponemos una breve explicación de los pasos de dicha metodología.

4.3.1. Metodología de Ian Foster

La metodología de Ian Foster propone cuatro fases de procesamiento, que son el particionamiento, la comunicación, la aglomeración y el mapeo. En el texto subsecuente, procederemos a dar una breve explicación de estas fases.

Particionamiento

El particionamiento consiste en dividir los datos y las computaciones en varias partes. Un buen particionamiento divide ambas en muchas piezas muy pequeñas. El particionamiento tiene dos acercamientos, el que se centra en la parte computacional y el que se centra en los datos.

Desde el punto de vista de los datos generalmente se busca dividir el más grande o más usado conjunto de datos y determinar una asociación entre estos y las computaciones sin afectar los resultados.

Por otra parte, el punto de vista funcional busca dividir la computación en pequeñas tareas y después determinar la asociación de estas con los datos. Muchas veces esta descomposición lleva a una colección de tareas que se ejecutan concurrentemente mediante un modelo de *pipeline*.

Para considerar que un particionado es óptimo se deben verificar los siguientes puntos:

- La minimización de tareas y almacenamiento redundantes.
- Las tareas en las que fueron divididas las computaciones deben ser del mismo tamaño.
- El número de tareas es una función creciente del tamaño del problema.

Comunicación

Después de haber hecho un buen particionamiento, es necesario establecer un sistema de comunicación entre tareas. Cuando existe una dependencia funcional entre tareas (una tarea necesita una porción de los resultados parciales o totales de otra), tenemos que crear canales de comunicación entre tareas y a esto se le conoce como comunicación local. En contraste, la comunicación global sucede cuando un mayor número de tareas deben contribuir para realizar un cálculo. Un claro ejemplo de este tipo de comunicación es cuando se requiere sumar los resultados de todos las tareas. La comunicación siempre conlleva cierto *overhead* y es necesario minimizarlo al momento de diseñar un algoritmo paralelo. Para determinar que se tiene un buen esquema de comunicación, se toman en cuenta los siguientes aspectos:

- La comunicación debe ser balanceada entre las tareas.
- Cada tarea debe comunicarse con el menor número posible de tareas.
- Las tareas deben ser capaces de comunicarse concurrentemente.
- Las tareas deben ser capaces de realizarse concurrentemente.

Aglomeración

La aglomeración es el proceso de agrupar pequeñas tareas en tareas más grandes con el fin de mejorar el desempeño o simplificar la programación.

Una de las metas de la aglomeración es disminuir el *overhead* de la comunicación. Si aglomeramos las tareas que requieren comunicarse, entonces eliminamos esa comunicación (junto con su *overhead*) porque el valor de los datos de esas tareas ya esta en la memoria de la tarea aglomerada. Otra forma de reducir el *overhead* de la comunicación es combinar grupos de tareas que envían y reciben datos, así reducimos el número de mensajes enviados. Enviar menos mensajes de mayor tamaño toma menos tiempo que enviar muchos mensajes pequeños.

Una segunda meta de la aglomeración es mantener la escalabilidad del algoritmo paralelo. Es decir que es deseable asegurarse de que no hayamos combinado muchas tareas al punto de que no sea posible ejecutar nuestro programa en una arquitectura con más procesadores.

Finalmente la aglomeración busca reducir el tiempo de desarrollo, ya que es posible utilizar código que haya sido de manera secuencial haciendo pocas modificaciones.

Para evaluar esta fase, Foster propone revisar los siguientes detalles:

- Las computaciones replicadas toman menos tiempo que la comunicación que reemplazan.
- El conjunto de datos replicados es lo suficientemente pequeño para permitir el escalamiento del algoritmo.
- Las tareas aglomeradas deben tener costos computacionales y de comunicación similares.

Mapeo

El mapeo es el proceso de distribuir las diferentes tareas entre los procesadores. El fin del mapeo es maximizar el uso de los procesadores y minimizar la comunicación entre procesadores. La comunicación entre procesos aumenta cuando las tareas conectadas por un canal son mapeadas a distintos procesadores, sin embargo, es posible reducirla si estas tareas son mapeadas al mismo procesador.

4.4. Esquema de paralelización

Ray Tracing con *KD trees* consiste a grandes rasgos en las siguientes tareas:

- Leer y construir la geometría de la escena.
- Construir un *KD tree*.
- Por cada pixel en la escena trazar un rayo que busque dentro del árbol la primitiva geométrica más cercana al punto de vista.

Se propuso un esquema de paralelización donde la lectura de datos, la construcción de un *KD tree* y el despliegue de las imágenes son realizadas por la CPU, mientras que la GPU se encarga de trazar los rayos dentro de la escena y determinar el color de los pixeles.

4.4.1. Esquema de trabajo con CUDA

En CUDA existen dos términos que se usan para referirse al sistema que alberga la tarjeta gráfica y para referirse a la tarjeta gráfica en si, estos terminos son *host* y *device*

respectivamente. Para poder trabajar con CUDA se deben seguir los siguientes pasos: asignación de memoria, copia de memoria del *host* al *device*, ejecución del *kernel* (unidad de ejecución en CUDA, es decir, el código que se ejecuta en el *device*) de CUDA, copia de memoria del *device* al *host*, liberar memoria asignada en el *device*. A continuación se describen dichos pasos y como se ven reflejados en el desarrollo de la propuesta de solución:

- **Asignación de Memoria.** Consiste en reservar el espacio de memoria donde se guardan los parámetros del kernel. En el caso de la soución propuesta se asigna memoria para almacenar los triángulos del objeto, el *KD tree* y los pixeles de la imagen resultante.

- **Copia de memoria *host-device*.** Consiste en copiar los datos que existen en la memoria del *host* a la memoria que fue asignada en el *device*. En esta solución los datos que se copian son los triángulos y el *KD tree*.

- **Ejecución del *kernel*.** En esta fase el *device* se encarga de trabajar con los datos copiados y realizar todo el proceso en paralelo. Para esta solución el *kernel* se encarga del algoritmo de *ray tracing*, es decir, proyectar los rayos hacia la escena, recorrer el árbol, determinar la intersección más cercana, procesar el color del pixel y guardarlo en el pixel correspondiente (esto es en la memoria asignada en el *device* para la imagen resultante).

- **Copia de memoria *device-host*.** Ya que se ejecutó el *kernel*, corresponde copiar los resultados en la memoria del *host*. En este caso solo se copian los pixeles de la imagen resultante.

- **Liberar memoria asignada.** Finalmente se libera toda la memoria que se haya asignado en el *device* para que pueda usarse en algún otro proceso que requiera la tarjeta gráfica.

4.4.2. División de datos

Para proponer una división de datos para trabajo en paralelo, debemos analizar cuales son los datos, en el caso de *ray tracing* con *KD trees*, los datos son los siguientes:

- *KD tree*.

- Pixeles de la imagen.

- Primitivas de la escena.

Gracias a que el color de cada pixel es determinado de manera independiente de los demás pixeles y cada rayo realiza el mismo proceso, la división de datos propuesta es por pixel, es decir que cada hilo de la GPU procesa un pixel al mismo tiempo. Además, el *KD tree* y las primitivas de la escena son datos de solo lectura, así que estos datos son compartidos por todos los hilos.

Con esta división de datos evitamos el número de mensajes que hay que mandar entre la CPU y la GPU, ya que copiar el *KD tree* y las primitivas solo se realiza una vez, dejando como *overhead* de comunicación la constante transmisión de los pixeles de la imagen.

4.4.3. División de tareas

Como se mencionó en secciones anteriores a este capítulo, *ray tracing* se divide en tres grandes tareas. De estas tres, el CPU se encarga de la lectura de datos y la construcción del *KD tree*, esto porque estas tareas son secuenciales y paralelizarlas no aumentaría el desempeño notablemente. Las dos tareas restantes (trazado de rayos) resultan ser altamente paralelizables ya que cada pixel de la imagen es independiente, es decir, que no existe una dependencia funcional. Esta división de tareas se hizo considerando la aglomeración de la que trata la metodología de Ian Foster. Se estudiaron las dos tareas realizadas en la GPU y se trato de dividir en funciones de tamaño medio que redujeran la comunicación entre ellas mismas.

4.5. Descripción de tareas

Las tres tareas expuestas al inicio de este capítulo fueron divididas en varias subtareas con el propósito de simplificar la programación y la explicación. A continuación se listan dichas subtareas:

- **Construcción de la escena**
  - Construir las primitivas.
- **Construcción del *KD tree***
  - *Heurística del área de la superficie (SAH).*

- Determinar la posición del plano de corte.
- Determinar primitivas de nodos hijo.

- **Trazado de rayos**
  - Buscar dentro del *KD tree* la intersección rayo-triángulo más cercana.
  - Test de intersección rayo-triángulo.

En las siguientes secciones se explican detalladamente cada una de las tareas anteriores.

4.5.1. Construcción de la escena

Para la solución propuesta, las escenas se leen a partir de archivos en formato `obj`. Estos archivos contienen una lista de puntos 3D que representan los vértices de varios triángulos. Dichos triángulos forman una malla que da forma a los objetos de la escena. Además, el formato `obj` especifica la información necesaria para construir cada triángulo, es decir, grupos de tres vértices de la lista. El formato `obj` ofrece también otro tipo de información como coordenadas u,v para el mapeo de texturas, normales de vértices, entre otras (Ver apéndice A). La forma en la que son declarados vértices en un archivo `obj` es estándar, fue necesaria la creación de un analizador sintáctico.

Construir las primitivas

Como ya se dijo, el formato `obj` ofrece los índices de los vértices que forman los triángulos que serán las primitivas de la escena. De este modo, construimos los triángulos, también se calcula el vector normal de cada triángulo con el fin de reducir los cálculos de intersectar un rayo. Además se busco que el tamaño ocupado en bytes de cada triángulo fuera el mínimo para hacer una implementación más económica en términos del uso de memoria. Estos triángulos se guardan en la siguiente estructura:

```
struct Triangle{
  float3 vertex[3]; //Tres vertices del triángulo
  float3 normal; //Vector normal a la superficie del triángulo
};
```

4.5.2. Construcción del *KD tree*

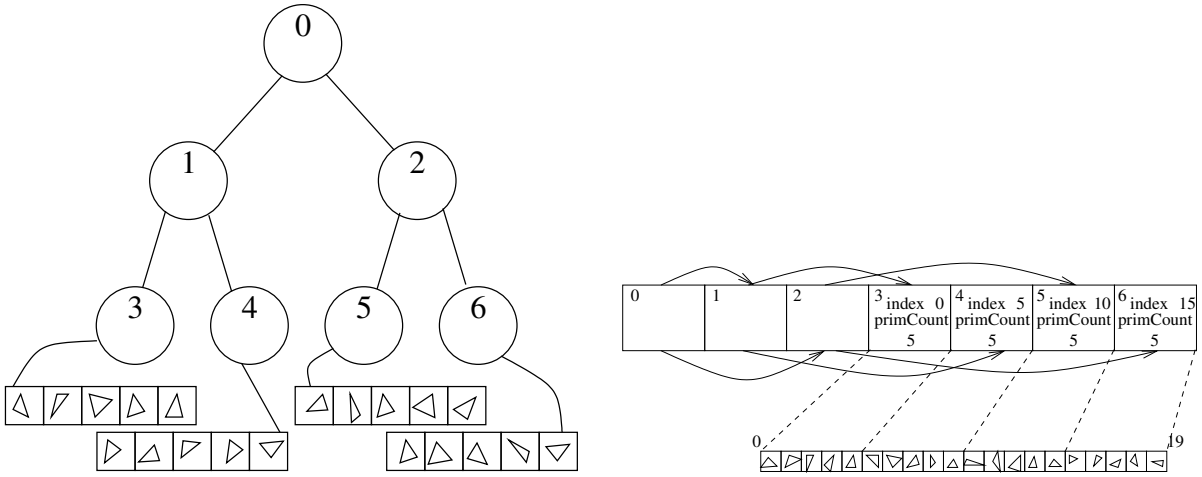
En la construcción de *KD trees* se toma en cuenta un volumen V que se puede ver como una caja delimitadora de la escena que es construida al evaluar cada vértice independientemente y determinar el mayor y el menor valor de las componentes X, Y y Z de todo el conjunto de vértices así podemos determinar el ancho, alto y profundidad de dicha caja delimitadora, más adelante se explica la razón por la cual se calcula este volumen. Para esta solución proponemos las siguientes estructuras:

```
struct KDTreeNode{
    int eje; //Eje de subdivisión
    int isLeaf; //Determina si el nodo es un nodo hoja
    int* prims; //Indices de primitivas del nodo
    aabb bounds; //Caja contenedora consiste de dos puntos 3D
 };

struct CUDAKDTreeNode{
    int eje; //Eje de subdivisión
    int isLeaf; //Determina si el nodo es un nodo hoja
    int primCount; //Contador de primitivas en el nodo
    int index; //Indice de la primer primitiva del nodo
    aabb bounds; //Caja contenedora consiste de dos puntos 3D
 };
```

La primer estructura es propia de la CPU ya que contiene un apuntador a una lista de índices de primitivas que va aumentando conforme se va construyendo el árbol y la segunda es para el trabajo de la GPU. La construcción del *KD tree* se realiza en la CPU de manera recursiva y después se construye un arreglo de estructuras de GPU que se copian en la memoria de la GPU para su uso en el algoritmo de *ray tracing*.

En la figura 4.1 se muestran los diagramas de ambas estructuras. Es posible observar que para el caso de los árboles de la CPU requieren de memoria dinámica, mientras que la estructura utilizada en la GPU es manejada como un arreglo estático de nodos.



(a) *KD tree* dinámico. (b) *KD tree* estático.

Figura 4.1: Estructuras de *KD trees*.

Heurística del área de la superficie (SAH)

La solución propuesta utiliza una metaheurísitca llamada *Surface Area Heuristic*(SAH) que fue presentada en [31] y después mejorada en [32]. Esta metaheurísitca es un método eficaz para estimar el costo de un *KD tree* para *ray tracing*, esto resulta en un *KD tree* aproximadamente óptimo. Esta metaheurísitca aplicada a la construcción de *KD trees* se basa en que las probabilidades de que un rayo intersecte un V_1 o V_2 ambos subvolumenes de un volumen V que es dividido en dos es dada por el área de la superficie de dichos subvolumenes, entonces se tiene lo siguiente:

$$P(V_1|V_s) = \frac{SA(V_1)}{SA(V_s)} \text{ y } P(V_2|V_s) = \frac{SA(V_2)}{SA(V_s)},$$

donde P es la probabilidad de la intersección, V es un subvolumen de V_s y $SA()$ es el área de la superficie relacionada a cada volumen. Esta área es calculada usando la formula siguiente:

$$SA(V) = 2(V_w V_d + V_w V_h + V_d V_h),$$

donde V_w, V_h y V_d son el ancho, el alto y la profundidad de un volumen V. Una vez que se han calculado las dos probabilidades, se procede a determinar el costo de una división. Asumiendo que el paso del rayo y la intersección rayo-triángulo tienen un costo promedio C_s y C_i, el costo de la división de un volumen V en V_1 y V_2 es estimado con la siguiente formula:

$$Costo(V_1, N_1, V_2, N_2) = C_s + C_i(P(V_1|V)N_1 + P(V_2|V)N_2),$$

donde N_1 y N_2 son el número de triángulos que hay en los subvolumenes V_1 y V_2 respectivamente.

Determinar la posición del plano de corte

En la construcción de *KD trees* es posible determinar la posición de los cortes que se realizan en la escena de diversas maneras. La más rápida y sencilla es cortar los subvolumenes en mitades escogiendo un eje diferente por cada corte, por ejemplo se puede escoger que primero sea un corte en X, luego en Y y finalmente en Z repitiendo ese orden hasta alcanzar algún determinado criterio de paro. Sin embargo, está forma de determinar los planos de corte no garantiza una distribución óptima de las primitivas dentro del árbol, así que pueden producir árboles desbalanceados, nodos hoja con demasiadas primitivas y viceversa. Otra forma es utilizando la metaheurísitca mencionada anteriormente (SAH) que es la que seleccionamos para esta solución ya que puede generar *KD trees* que son más balanceados y por ende más rápidos de recorrer.

Para determinar la posición del plano de corte primero es necesario saber en que eje se va a colocar dicho corte. Comparando el ancho, alto y la profundidad del volumen V que contiene las primitivas podemos determinar el eje más conveniente para realizar la subdivisión de la escena, esto es, si $V_w >= V_h$ y $V_w >= V_d$ se escoge el eje X, si $V_h >= V_w$ y $V_h >= V_d$ se escoge el eje Y en otro caso se escoge a Z como eje de corte.

Ya que se tiene el eje de corte, se hace un calculo de costos de SAH entre las posibles posiciones de dicho corte. Estas posiciones se determinan otra vez a partir de los vértices de cada triángulo pero esta vez se toman como el conjunto de vértices que forman al triángulo. A partir de la posición obtenida, se calculan los dos subvolumenes de V y se utiliza SAH para determina si es la mejor posición, en caso de no ser la mejor posición obtenida hasta ese momento, está es descartada y se calcula la siguiente posición. Hacer este calculo, conlleva hacer el calculo de los subvolumenes V_1 y V_2 que serán los volúmenes respectivos de cada nodo hijo.

Determinar primitivas de nodos hijo

Para finalizar el proceso de construir un *KD tree* se deben determinar las primitivas de cada nodo hijo que será procesado recursivamente. Gracias a los resultados del proceso anterior (posición del plano de corte y subvolumenes V_1 y V_2) podemos separar las

primitivas con un prueba de intersección entre triángulo y caja delimitadora, para esta solución utilizamos el método propuesto por Tomas Akenine-Möller [1]. Este método determina si un triángulo T esta contenido en una caja V.

4.5.3. Trazado de rayos

El trazado de rayos en un algoritmo de *ray tracing* con *KD tree* se puede dividir en dos tareas. Estas tareas son realizadas en la GPU en la solución propuesta. Cada hilo de la GPU maneja un pixel de la imagen resultante, es decir que para este caso, cada hilo proyecta un rayo hacia la escena y determina si hay una intersección con el modelo o no. Ya que se encontró una intersección, se determina el color del pixel proyectando otro rayo desde el punto de intersección hacia la luz en la escena para encontrar el tipo de sombreado de la imagen.

Un rayo es definido si se tienen su origen y su dirección de la siguiente manera:

$$R_{origen} = R_o = [X_o, Y_o, Z_o]$$
$$R_{dirección} = R_d = [X_d, Y_d, Z_d]$$
$$y\ X_d^2 + Y_d^2 + Z_d^2 = 1\ (la\ dirección\ está\ normalizada)$$

y entonces el rayo queda definido por la siguiente ecuación:

$$R(t) = R_o + R_d * t \qquad t > 0$$

El parámetro t de esta ecuación representa la distancia (o tamaño de paso) que recorre el rayo, es decir, la distancia a la que es proyectado. En una implementación ideal, al inicio del algoritmo, se propone una distancia t infinita, pero ya que esto no es posible, proponemos un tamaño de paso t_max demasiado grande que servirá de referencia para comparar mientras se recorre el *KD tree*. Antes de realizar la búsqueda dentro del *KD tree* primero acotamos el tamaño de paso $t = t_max$ propuesto al inicio con la caja contenedora del modelo. Esto se puede considerar como el primer parámetro para saber si existe una intersección del rayo con la escena y nos ahorra la búsqueda en el *KD tree* pues se utiliza los tamaños de paso $t_n = 0$ y $t_f = t$ al inicio. Si resulta que $t_n > t_f$ entonces el rayo está dando un paso negativo, es decir que se aleja de la escena.

Búsqueda dentro del *KD tree*

En esta tesis adaptamos el algoritmo *KD-restart* propuesto por Foley [14]. Este algoritmo toma como base el trabajo de Wald [12] y elimina el uso de una pila. El algoritmo original de Wald propone delimitar el rayo en un rango $(tmin, tmax)$ que es tomado a consideración con el nodo que se está atravesando actualmente. Cuando un rayo atraviesa un nodo, el rango $(tmin, tmax)$ es clasificado con respecto al plano de corte del nodo. Si este rango se encuentra en su totalidad de un lado del plano, el recorrido continua por el nodo hijo respectivo de ese lado. Por otra parte, si este rango se encuentra en ambos lados del plano, el siguiente nodo hijo a visitar es el que atraviesa primero el rayo mientras que el nodo hermano es almacenado en una pila. Así es como un rayo recorre el *KD tree* hasta llegar a un nodo hoja donde se verifica si alguna primitiva es intersectada por el mismo, en el caso de que ninguna primitiva sea intersectada se saca un elemento de la pila y se sigue el mismo algoritmo empezando desde el nodo referenciado por el elemento sacado de la pila.

Foley observó que eliminando la operación de apilado del algoritmo, el recorrido del árbol resulta en el primer nodo hoja atravesado por el rayo. Además que $tmax$ solo es modificado en el caso que se tenga que apilar un elemento y que este valor será guardado en la pila como $tmin$. Esto implica que cuando el rayo llega a un nodo hoja $tmax$ es el valor global de $tmax$ o es el valor $tmin$ con el cual el rayo estaría atravesando el siguiente nodo hoja, en la figura 4.2 se puede observar con más claridad esta aseveración. Con estas observaciones, Foley modificó el algoritmo de Wald de tal forma que en el caso en que un rayo llegue a un nodo hoja y este no atraviese ninguna primitiva se reinicie la búsqueda dentro del árbol intercambiando el valor de $tmin$ por $tmax$. En el algoritmo 1 mostramos el pseudocódigo del algoritmo *KD-restart*.

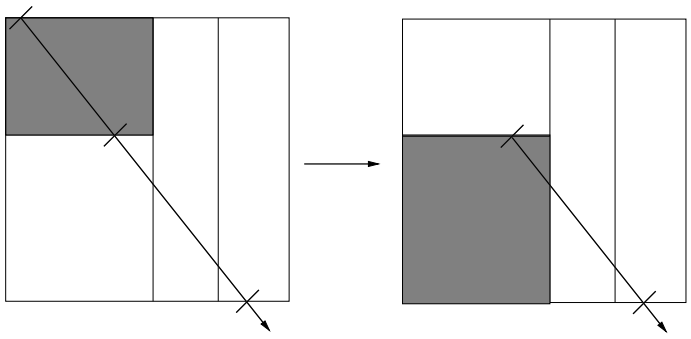


Figura 4.2: El valor de $tmax$ de un nodo hoja es el valor $tmin$ de otro.

Algoritmo 1 *KD-restart*

Require: $arbol, rayo$
  $global\_tmax \Leftarrow \infty$
  $global\_tmin \Leftarrow -\infty$
  $nodo \Leftarrow nodo\_raiz$
  if $(tmin, tmax) = intersecta(nodo\_raiz.bounds, rayo)$ **then**
    while $tmax \neq global\_tmax$ **do**
      if $nodo.isLeaf$ **then**
        if $intersect$ **and** $intersect.t < tmax$ **then**
          return $SUCCEED$ {El rayo intersecta alguna primitiva}
        else
          $tmin \Leftarrow tmax$
          $tmax \Leftarrow global\_tmax$
          $nodo \Leftarrow nodo\_raiz$
        end if
      else
        $a \Leftarrow nodo.eje$
        $t \Leftarrow (nodo.bounds.value[a] - rayo.origen[a])/rayo.direccion[a]$
        $(first, second) = ordena(rayo.direccion[a], nodo.left, nodo.right)$
        if $t \geq tmax$ **or** $t < 0$ **then**
          $nodo \Leftarrow first$
        end if
        if $t \leq tmin$ **then**
          $nodo \Leftarrow second$
        else
          $nodo \Leftarrow first$
        end if
      end if
    end while
  end if
  return $FAIL$ {El rayo no intersecta alguna primitiva}

Test de intersección rayo-triángulo

Existen varias formas de determinar si un rayo intersecta un triángulo, en esta propuesta de solución utilizamos la propuesta por Möller y Trumbore [33] ya que es de las más rápidas y que requieren poco espacio en memoria (solo requiere los tres vértices del triángulo y el rayo definido con términos de su origen y dirección). Esta prueba se basa en hallar las coordenadas baricéntricas del triángulo y en el proceso también se calcula el tamaño de paso t de la ecuación del rayo.

Un punto, $T(u, v)$, en un triángulo es dado por:

$$T(u, v) = (1 - u - v)V_0 + uV_1 + vV_2,$$

donde V_n son los puntos que conforman al triángulo y (u, v) son las coordenadas baricéntricas, estas deben cumplir $u \geq 0, v \geq 0$ y $u + v \leq 1$. Encontrar la intersección rayo-triángulo es básicamente encontrar los valores u, v y t de $R(t) = T(u, v)$ es decir:

$$R_o + R_d t = (1 - u - v)V_0 + uV_1 + vV_2,$$

Acomodando los términos, la ecuación queda de la siguiente forma:

$$[-R_d, V_1 - V_0, V_2 - V_0] \begin{bmatrix} t \\ u \\ v \end{bmatrix} = R_o - V_o,$$

Con esto observamos que hallar las coordinadas baricéntricas y el tamaño de paso del rayo se obtienen al resolver el sistema de ecuaciones lineales anterior. Este algoritmo es utilizado en la búsqueda de intersecciones rayo-triángulo dentro del algoritmo *KD-restart*.

Capítulo 5

Pruebas y resultados

En este capítulo se expone la infraestructura de hardware y software que fue utilizada para realizar las pruebas realizadas. También se analizan las escenas de prueba de tal forma que se pueda dar una idea de la complejidad de estas. Además, se explica en que consistieron las pruebas realizadas asi como un estudio de los resultados obtenidos. Finalmente, se presentan los resultados de las pruebas realizadas.

5.1. Infraestructura del sistema de pruebas

Las pruebas del algoritmo desarrollado fueron efectuadas en dos entornos distintos de hardware y software para poder observar su comportamiento en diferentes entornos y tener varios puntos de comparación. Esta sección explica a detalle dichos entornos de hardware y software. También se presentan algunas diferencias que afectaron en la realización de las pruebas en ambos entornos.

5.1.1. Hardware

Para las pruebas realizadas de esta tesis se contó con dos equipos. El primero fue un servidor y el segundo un equipo de escritorio. Se escogieron estos entornos de hardware por dos razones, primero por la disponibilidad del equipo y segundo para poder comparar las diferentes capacidades entre distintas tarjetas gráficas. En las tablas 5.1 y 5.2 continuación se detallan las características de ambos equipos:

Equipo 1. Servidor	
Procesador	
Modelo	Intel(R) Xeon(R) CPU X5675
Número de Procesadores	12
Número de Núcleos por Procesador	6
Velocidad de Reloj	3.0GHz
Memoria de caché	12MB
Tarjetas de Video NVIDIA	
Modelo	Geforce GTX 460
Versión del controlador de CUDA	5.0
Capacidad de computo CUDA	2.1
Memoria global	1GB
Cantidad total de Núcleos	336 (7 Multiprocesadores x 48 Núcleos)
Frecuencia de reloj de GPU	1530MHz (1.53GHz)
Modelo	Tesla C2070
Versión del controlador de CUDA	5.0
Capacidad de computo CUDA	2.0
Memoria global	5375MB
Cantidad total de Núcleos	448 (14 Multiprocesadores x 32 Núcleos)
Frecuencia de reloj de GPU	1147MHz (1.15GHz)

Tabla 5.1: Detalles del servidor.

Equipo 2. Equipo de escritorio	
Procesador	
Modelo	Intel(R) Core 2 Duo(R) CPU E8400
Número de Núcleos	2
Velocidad de Reloj	3.0GHz
Memoria de caché	6MB
Tarjeta de Video NVIDIA	
Modelo	Geforce 9500 GT
Versión del controlador de CUDA	4.2
Capacidad de computo CUDA	1.1
Memoria global	512MB
Cantidad total de Núcleos	32 (4 Multiprocesadores x 8 Núcleos)
Frecuencia de reloj de GPU	1400MHz (1.4GHz)

Tabla 5.2: Detalles del Equipo de escritorio

5.1.2. Software

El software que fue utilizado para el desarrollo y ejecución del algoritmo es el siguiente:

- **Software del sistema.** En ambos equipos de prueba se contó con un sistema operativo Linux. El equipo de escritorio cuenta con la distribución Ubuntu 12.04 LTS, mientras que el servidor tiene la distribución CentOS 5.8 (Final)

- **Software de compilación/ejecución.** La solución propuesta se puede dividir en dos partes según el hardware en el que se ejecuta. La parte que se ejecuta en la CPU y la parte que se ejecuta en la GPU. La parte correspondiente a la CPU es encargada de leer y cargar el archivo de geometría en memoria, crear el árbol correspondiente y desplegar la imagen; esta parte fue realizada en C y C++. La parte correspondiente al trazado de rayos, el recorrido del árbol, la determinación del color son ejecutadas en la GPU utilizando CUDA en su versión 4.2.

5.1.3. Diferencias de los entornos de prueba

El desempeño del algoritmo se ve afectado por las diferentes características que tienen los entornos de prueba. Por consiguiente se muestran las diferencias y se explica brevemente el impacto que sufre el desempeño del algoritmo a causa de estas.

Por una parte, las diferencias entre la capacidad de cómputo de las CPUs afecta al algoritmo en la construcción del árbol. Por otra parte, la velocidad de lectura y escritura de datos afectan la velocidad de carga del modelo y el despliegue de imagenes.

En relación a las tarjetas gráficas existen dos grandes diferencias. La principal diferencia es el número de hilos que se pueden asignar debido a la cantidad de multi-procesadores y núcleos de CUDA, mientras más hilos puedan ejecutarse en paralelo, más rápida será la generación de imagenes ya que cada hilo procesa un pixel de la imagen a la vez. También se tiene en cuenta la capacidad de cómputo CUDA, esto influye en las limitaciones que tiene la tarjeta gráfica y las optimizaciones específicas que se realizan en tiempo de compilación.

5.2. Escenas de prueba

Para realización de las pruebas sobre el algoritmo desarrollado se utilizaron varias escenas que varían en el número de triángulos y vertices que las conforman. Todas estas escenas son escenas comunmente utilizadas como *benchmarking* en trabajos de investigación de render. Las escenas utilizadas se detallan en la tabla 5.3 y en la figura 5.1 se grafica la relación entre las escenas y el número de vertices y triángulos que las forman. Tomando en cuenta estos datos es posible hacerse una idea del tiempo que tomara el proceso de render que implíca la carga del modelo, la creación del *KD tree* y la ejecución del algoritmo de *ray tracing*. Por ejemplo, los modelos con menos triángulos demostraron ser aquellos con menor tiempo de ejecución. Además es posible deducir que mientras más triángulos se tengan, más memoria ocuparán; a pesar de esto, el *KD tree* ocupa un espacio máximo fijo de memoria definido por un umbral de profundidad.

	Escena	**Vertices**	**Triángulos**
1	Monkey[1]	507	968
2	Bunny[2]	2503	4968
3	Teapot[3]	3644	6320
4	Sibenik[4]	83490	75284
5	Dragon[2]	50000	100000
6	Happy Buddha[2]	49990	100000

Tabla 5.3: Especificación de las escenas de prueba.

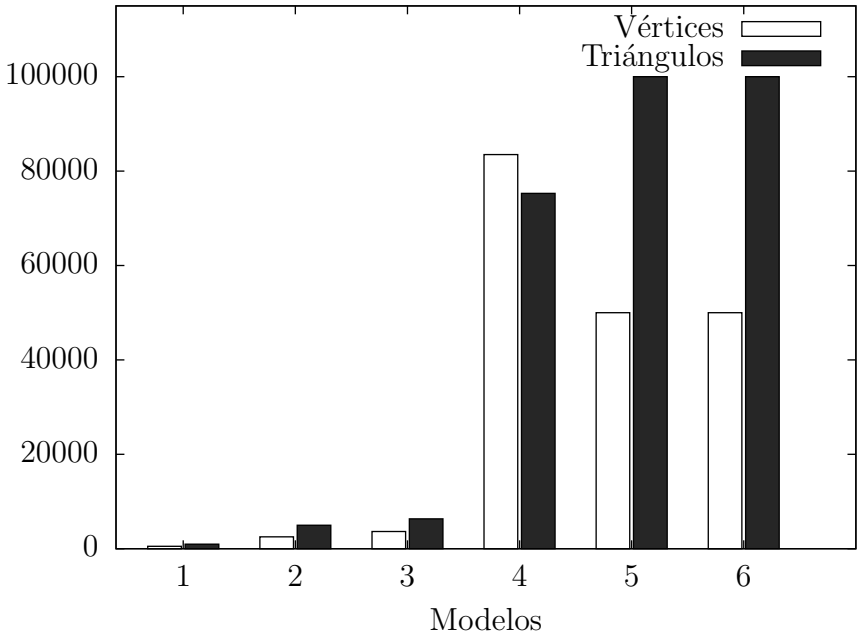


Figura 5.1: Detalles de los modelos de prueba.

[1]Modelo exportado de *Blender*.
[2]Modelo obtenido de *The Stanford 3D Scanning Reppository*[34].
[3]Modelo obtenido del grupo de computación gráfica del MIT[35].
[4]Modelo obtenido de *Computer Graphics Data* [36].

5.3. Resultados de las pruebas

Las pruebas efectuadas consistieron en ejecutar el algoritmo e ir moviendo el punto de vista para hacer un muestreo de tiempo de ejecución por cada frame generado. Se tomaron medidas de los diferentes tiempos, tiempo de carga del modelo, tiempo de construcción del *KD-tree* y tiempo de ejecución en GPU. La carga del modelo y la construcción del árbol solo se realizan una vez, una vez cargado el modelo y construido el árbol el algorítmo se puede ejecutar cambiando el punto de vista cuantas veces sea necesario, en este caso el muestreo de tiempo constó de 20 muestras las cuales se promediaron y con eso podemos calcular una tasa de fps estimada.

En las tablas 5.4 y 5.5 se presentan los tiempos y fps estimados para las escenas de prueba en los diferentes entornos en los que se trabajo. Los tiempos de carga hacen referencia a la carga del modelo y la construcción del *KD tree* y el tiempo promedio es referente a la ejecución del algoritmo en la GPU. *T. promedio 1* es el tiempo de la tarjeta Tesla C2070 y *T. promedio 2* es el de la tarjeta Geforce GTX 460.

Como se esperaba, el tiempo de ejecución fue mayor en el equipo de escritorio debido a su antiguedad y sus capacidades. Por otra parte, las diferencias entre los tiempos de las dos tarjetas del servidor no son tan grandes debido a que tienen capacidades similares. Esto nos da una idea de la capacidad de cómputo necesaria para alcanzar tiempos considerables para establecer tiempo real. El tiempo de carga y generación del árbol pueden considerarse despreciables ya que estos solo ocurren una vez.

Escena	**Equipo 1. Servidor**				
	T. carga	**T. promedio 1**	**FPS**	**T. promedio 2**	**FPS**
Monkey	1.1501	0.0071	140.8450	0.0083	120.4819
Bunny	1.3037	0.0089	112.3595	0.0096	104.1667
Teapot	1.2491	0.0075	133.3333	0.0078	128.2051
Sibenik	1.4157	0.0355	28.1690	0.0380	26.3158
Dragon	1.6079	0.0362	27.6243	0.0375	26.6666
Happy Buddha	1.6803	0.0371	26.9541	0.0379	26.3852

Tabla 5.4: Tiempos medidos (segundos) y FPS estimados del servidor.

Escena	**Equipo 2. Escritorio**		
	T. carga	**T. promedio**	**FPS**
Monkey	1.3027	0.0408	24.5098
Bunny	1.5090	0.0702	14.2450
Teapot	1.3062	0.0560	17.8571
Sibenik	1.5080	0.0856	11.6822
Dragon	2.0591	0.1037	9.6432
Happy Buddha	2.1009	0.1648	6.0679

Tabla 5.5: Tiempos medidos (segundos) y FPS estimados del equipo de escritorio.

En la figura 5.2 se aprecia una gráfica que muestra los tiempos de carga y creación del *KD tree*. Se puede observar que mientras más triángulos tiene la imagen más tiempo toma este paso, sin embargo, el tiempo consumido no solo depende de la cantidad de triángulos en la escena sino también en su distribución dentro de esta. La distribución de los triángulos en la escena es un factor que influye en la creación del *KD tree* pues si se encuentra una aglomeración de triángulos en cierto espacio de la escena la tarea de encontrar un plano de corte utilizando SAH se vuelve más laboriosa. Tal es el caso de las escenas *Bunny* y *Happy Buddha* (o *Buddah*) este fenómeno es más notorio en el caso de *Buddah* ya que tiene la misma cantidad de triángulos que *Dragon* y a pesar de esto tarda un poco más en ser procesada.

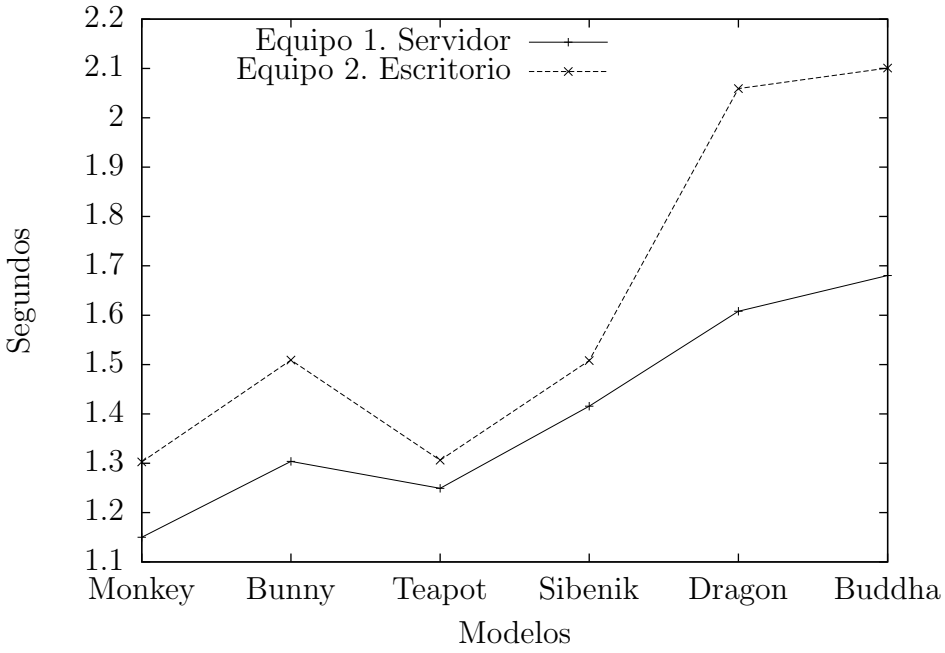


Figura 5.2: Tiempo de carga.

Analizando el tiempo de ejecución podemos observar que, al igual que en los tiempos de carga, estos aumentan según el número de triángulos en la escena. En la figura 5.3 se muestran los tiempos de ejecución en forma de gráfica, podemos observar más claramente que las tarjetas gráficas del servidor obtuvieron tiempos similares mientras que el equipo de escritorio con la tarjeta Geforce 9500 GT tomó mucho más tiempo y es posible concluir que en está tarjeta el ray tracing no podría ser ejecutado en tiempo real para modelos complejos.

Finalmente presentamos las imágenes resultantes de ejecutar las pruebas en la figura 5.4.

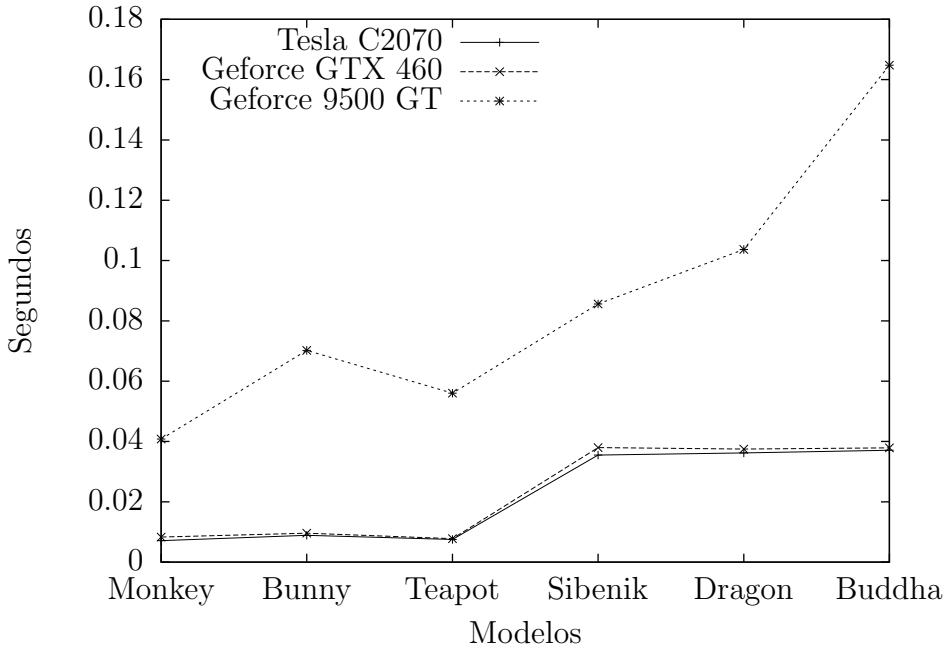
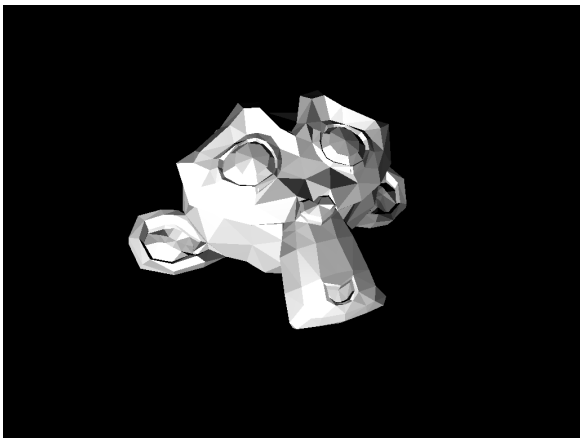
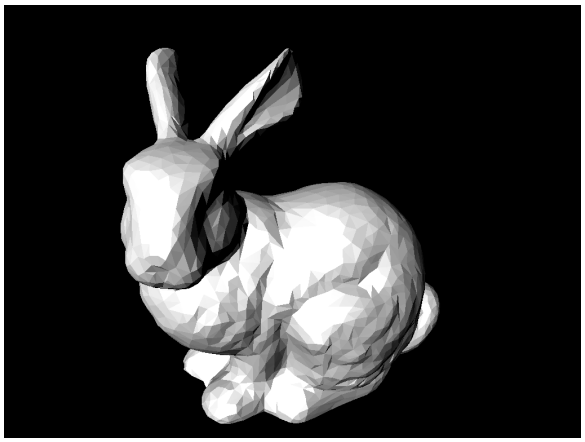


Figura 5.3: Tiempo de procesamiento.



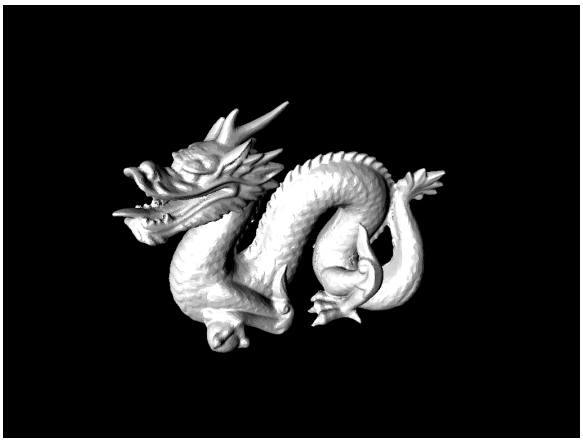
(a) Monkey



(b) Bunny

(c) Teapot

(d) Sibenik



(e) Dragon

(f) Buddha

Figura 5.4: Escenas de prueba.

Capítulo 6

Conclusiones y trabajo a futuro

Esta tesis tuvo como propósito aplicar el algoritmo de *ray tracing* en GPUs con CUDA para objetos formados por una malla de tríangulos para demostrar la aceleración que es posible alcanzar gracias al uso de las GPUs.

6.1. Conclusiones finales

A continuación se listan las conclusiones finales de este trabajo de tesis:

- La distribución de las primitivas en una escena afecta la construcción del *KD tree* por lo tanto se puede tener un árbol desbalanceado y con eso la velocidad del algoritmo de recorrido se ve afectada. Esto lo podemos observar claramente en las escenas de prueba *teapot* y *bunny* donde a pesar de que el modelo de *teapot* utilizado en las pruebas está compuesta por más triángulos que el modelo *bunny* el tiempo de ejecución es mayor en esta segunda escena. Esto es debido a que al tener un árbol desbalanceado aumenta el tiempo de busqueda dentro de los nodos hoja, es decir que algunos nodos hoja tienen muchas más primitivas que otros nodos hoja del mismo árbol.

- En el procesamiento y sintesis de imágenes se ha visto que el poder computacional que brinda la GPU impacta de manera sustancial reduciendo tiempos de ejecución. Esta aceleración que ofrece la GPU no esta limitada solamente al procesamiento de imágenes sino también a diversas áreas como son la bioinformática, la química computacional, la física cuántica, entre otros.

- Para lograr el tiempo real utilizando *ray tracing* es necesario considerar varios

factores, desde el hardware mínimo que se debe utilizar hasta los efectos finales que se quieran aplicar a las imagenes. Por lo tanto, es difícil, garantizar tiempo real para todo tipo de modelos y de entornos. Sin embargo, es posible alcanzar el tiempo real hasta cierto punto en el que las escenas se vuelven complejas.

- Utilizar *KD trees* para acelerar al algoritmo de *ray tracing* en GPUs es efectivo si se modelan de tal forma que sea una estructura compacta y a su vez se pueda copiar de manera sencilla y rápida a la memoria del GPU. Se requiere una estructura compacta debido a la gran cantidad de parámetros que se requieren para ejecutar el algoritmo y a la memoria limitada por la GPU ya que a diferencia del cómputo en CPU no se cuenta con memoria virtual. Esta estructura también debe ser sencilla y rápida de copiar a la GPU debido al *overhead* generado al realizar operaciones de carga y descarga de memoria entre la GPU y CPU.

- El algoritmo *KD-restart* funcionó adecuadamente y a pesar de sus desventajas tuvo un mejor desempeño que su predecesor. La gran desventaja de este algoritmo es la repeticion de operaciones ya que por cada vez que no se encuentra una intersección en un nodo hoja del *KD tree* hay que reiniciar el algoritmo. Además eliminar la pila que manejaba su predecesor deja libre un poco más de espacio en la memoria de la GPU.

6.2. Contribuciones

- Se adaptó de un algoritmo heterogéneo de *ray tracing* que usa *KD trees* como estructura aceleradora capaz de ejecutarse en paralelo utilizando GPUs.

- Se diseñó una estructura de tipo árbol con la cual se pudo trabajar en GPUs.

6.3. Trabajo a futuro

Referente al trabajo a futuro propuesto de esta tesis tenemos:

- Implementación de alguna técnica de antialias que no tenga un gran impacto en el desempeño del algoritmo. Esto para mejorar la calidad de las imágenes resultantes.

- Implementación de objetos refractantes y reflejantes. Esta tesis ofrece un algoritmo de *ray tracing* que puede ser modificado para trabajar con distintos materiales de manera sencilla ya que cada tarea esta dividida en módulos, tal es el caso del recorrido del árbol para la búsqueda de intersecciónes que al ser una función genérica se puede usar en casos de escenas con objetos reflejantes y refractantes.

- Diseño de algun método para seleccionar los criterios de paro en la construcción de *KD trees* basado en el número de primitivas que componen la escena. Estos criteros de paro son dos, una profundidad máxima del árbol y una cantidad máxima de primitivas dentro de los nodos, generalmente ambos criterios son definidos manualmente.

- Extensión a más primitivas geométricas más complejas como esféras, toroides, superficies de Bézier, entre otras.

- Integración con alguna herramienta CAD como Autodesk Maya(R), Autodesk 3ds MAX, Blender.

Ya que se tiene un algoritmo en GPUs se puede experimentar con distintos parámetros, por ejemplo determinar en tiempo de ejecución un cierto número de hilos para mejorar el desempeño dependiendo del tamaño de la imagen.

Apéndice A

Archivos `obj`

OBJ es un formato de archivo de definición de geometría que fue desarrollado primero por la compañía *Wavefront Technologies* para uso de su paquete de animación *Advanced Visualizer* [37]. Este formato es abierto y ha sido adoptado por otras aplicaciones de gráficos en 3D.

El formato obj es un formato simple de datos que solo representa la geometría 3D, esto es, la posición de cada vertice, la posición UV de cada coordenada de textura de los vertices, normales y las caras que definen los polígonos definidos por una lista de vertices. Las coordenadas en un archivo obj no tienen unidades, asi que estos pueden contener la información de la escala comentadas. La información mínima que debe contener un archivo `obj` son los vertices de los poígonos y las caras.

A.1. Formato

Las lineas que comienzan con un caracter hash (#) son comentarios.

```
# esto es un comentario
```

Los vertices son definidos en sus coordenadas (x,y,z,w) aunque la coordenada w es opcional y por defecto es 1.0. Las lineas que comienzan con el caracter **v** denotan un vertice.

```
v 0.123 0.234 0.345 1.0
v 0.456 0.567 0.678
```

Las coordenadas de texturas UV son descritas por (u,v,w), varian entre 0 y 1 y w es opcional, por defecto es 0. Las coordenadas UV son lineas que comienzan con los caracteres **vt**.

```
vt 0.500 1 0
vt 1 0.500
```

Los vectores normales se escriben en formato (x,y,z) y pueden o no estar normalizadas. Dentro del archivo las podemos encontrar en las liineas que comienzan con los caracteres **vn**.

```
vn 0.707 0.000 0.707
```

Las caras de los polígonos se pueden declarar de diversas formas, dependiendo de cuanta información contenga el archivo `obj`. Las lineas que comienzan con el caracter **f** denotan las listas de los vertices que forman la cara, la lista completa de vertices es indexada empezando con el 1. Si el archivo contiene solo los vertices de los poligonos, las caras se expresan de la siguiente manera:

```
f v1 v2 v3,
```

donde $v1$, $v2$ y $v3$ son los indices de los vertices que forman la cara. Si además se tienen las coordenadas UV, las caras se declaran asi:

```
f v1/vt1 v2/vt2 v3/vt3,
```

donde $vt1$, $vt2$ y $vt3$ son los indices de la lista de coordenadas UV. Si también se tienen los vectores normales, las caras se escriben asi:

```
f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3,
```

donde $vn1$, $vn2$ y $vn3$ son los indices de la lista de vectores normales. Si no se tienen las coordenadas UV pero las normales si, las caras se declaran asi:

```
f v1//vn1 v2//vn2 v3//vn3
```

Bibliografía

[1] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. *Real-Time Rendering, Third Edition*. AK Peters, 3 edition, July 2008.

[2] Georg Rainer Hofmann. Who invented ray tracing? *The Visual Computer*, 6:120–124, 1990. 10.1007/BF01911003.

[3] Arthur Appel. Some techniques for shading machine rendering of solids. *AFIPS Conference Proc.*, 32:37–45, 1968.

[4] J. D. Foley and Turner Whitted. An improved illumination model for shaded display, 1979.

[5] Ling Sing Yung, Can Yang, Xiang Wan, and Weichuan Yu. Gboost: a gpu-based tool for detecting gene-gene interactions in genome-wide case control studies. *Bioinformatics*, 27(9):1309–1310, 2011.

[6] Lucía Araceli Oviedo Díaz. Gestor de tareas para un render farm basado en GPUs. Master's thesis, CINVESTAV, Zacatenco, 2012.

[7] Jon Peddie Research. Digital content creation software market. Technical report, 2007.

[8] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer graphics: principles and practice (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[9] Edwin Earl Catmull. *A subdivision algorithm for computer display of curved surfaces.* PhD thesis, 1974. AAI7504786.

[10] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. In *Proceedings of the 7th annual conference on*

Computer graphics and interactive techniques, SIGGRAPH '80, pages 124–133, New York, NY, USA, 1980. ACM.

[11] D. Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19(2):129–147, June 1982.

[12] Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in o(n log n). In *IN PROCEEDINGS OF THE 2006 IEEE SYMPOSIUM ON INTERACTIVE RAY TRACING*, pages 61–70, 2006.

[13] Ingo Wald. On fast construction of sah-based bounding volume hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, RT '07, pages 33–40, Washington, DC, USA, 2007. IEEE Computer Society.

[14] Tim Foley and Jeremy Sugerman. KD-tree acceleration structures for a GPU raytracer. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS*, pages 15–22, 2005.

[15] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless KD-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26:415–424, 2007.

[16] Sven Woop, Jörg Schmittler, and Philipp Slusallek. Rpu: a programmable ray processing unit for realtime ray tracing. *ACM Trans. Graph.*, 24(3):434–444, July 2005.

[17] Carsten Benthin. *Realtime ray tracing on current CPU architectures*. PhD thesis, Saarländische Universitäts- und Landesbibliothek, Postfach 151141, 66041 Saarbrücken, Germany, 2006.

[18] J. Gunther, S. Popov, H.-P. Seidel, and P. Slusallek. Realtime ray tracing on gpu with BVH-based packet traversal. In *Interactive Ray Tracing, 2007. RT '07. IEEE Symposium on*, pages 113 –118, sept. 2007.

[19] Min Shih, Yung-Feng Chiu, Ying-Chieh Chen, and Chun-Fa Chang. Real-time ray tracing with cuda. volume 5574, pages 327–337. 2009.

[20] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin

Robison, and Martin Stich. Optix: a general purpose ray tracing engine. *ACM Trans. Graph.*, 29(4):66:1–66:13, jul. 2010.

[21] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, September 1972.

[22] Ingo Molnar. The native posix thread library for linux. Technical report, Tech. Rep., RedHat, Inc, 2003.

[23] James Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[24] Scott Oaks and Henry Wong. *Java Threads, Second Edition*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2nd edition, 1999.

[25] Y. Zhang, L. Peng, B. Li, J.K. Peir, and J. Chen. Architecture comparisons between nvidia and ati gpus: Computation parallelism and data communications. 2011.

[26] C. Nvidia. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Technical report, 2009.

[27] C. Nvidia. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. Technical report, 2012.

[28] http://www.top500.org.

[29] http://www.green500.org.

[30] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.

[31] David J. MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *Vis. Comput.*, 6(3):153–166, May 1990.

[32] Vlastimil Havran and Jiri Bittner. On improving kd-trees for ray shooting. In *In Proc. of WSCG 2002 Conference*, pages 209–217, 2002.

[33] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *J. Graph. Tools*, 2(1):21–28, October 1997.

[34] Marc Levoy. The Stanford 3D Scanning Repository.

[35] MIT computer graphics group.

[36] Marko Dabrovic. Computer graphics data.

[37] Wavefront Technologies. Obj specification. Technical report, Wavefront Technologies.