Documentación - "Computación Gráfica sobre GPU"

Gonzalo Ordeix, Santiago Cioli

24 de octubre de 2010

Índice general

Capítulo 1

Introducción

Este proyecto consiste en estudiar las distintas estrategias de trabajo en las tarjetas gráficas (GPU Graphic Processing Unit) modernas. Para esto se evaluaran las estrategias a seguir para la resolución de problemas utilizando paralalelismos en las GPUs actuales, las cuales pueden ser consideradas en muchos casos de procesamiento para cualquier propósito (GPGPU de su sigla en inglés General Purpose Graphic Processing Unit). A partir de este punto se diseña y personaliza el algoritmo de *ray tracing* que se encuentra ampliamente extendido en el área de la computación gráfica, siendo la base para otros algoritmos de generación de imágenes fotorealistas como pueden ser *photon mapping* y radiosidad. Además se evaluaron las estrategias para medir la calidad de los resultados. Bajo la premisa de evaluar la calidad del resultado del proyecto tambien se generaron un conjunto de casos de prueba para la aplicación construida con el fin de evaluar los resultados en varios aspectos.

1.1. Objetivos

En los últimos años las tarjetas gráficas (co-procesador gráfico, GPU) han experimentado una evolución explosiva. La evolución no solo fue cuantitativa (capacidad de cómputo) sino que en gran medida ha sido cualitativa (han pasado a computar etapas que antes las computaba la CPU). Este hecho, ha motivado que muchos científicos busquen la utilización de las GPUs para la resolución de problemas generales (bases de datos, computación científica, etc). En particular en áreas de cómputo intensivo, entre las que se destaca la computación gráfica.

En gran medida el aumento en la capacidad de cómputo de las GPUs se sustenta en su arquitectura que permite trabajar con estrategias de cálculo paralelo. La arquitectura de las GPUs evolucionó en forma importante en este siglo en varios aspectos (capacidad, cantidad de procesadores, etc), destacándose especialmente la mejora en la precisión de los datos con los que trabaja. A comienzos de los años 2000 disponían solamente de números de 8 bits, mientras que en el año 2003 pasaron a trabajar con números en punto flotante de simple precisión de 32 bit.

En base a lo mencionado anteriormente, los objetivos del proyecto son estudiar la arquitectura de las GPU modernas, diseñar e implementar algoritmos para computación gráfica sobre GPU y evaluar los algoritmos desarrollados sobre un conjunto de casos de pruebas representativos.

El trabajo del proyecto tiene como principales metas lograr relevar y estudiar los algoritmos implementados en GPU, así como la arquitectura que presentan. Evaluar las implementaciones existentes de los algoritmos de computación gráfica, comparando luego con una implementación propia. También se proyecta evaluar las diferencias que existen de performance entre una GPU y un CPU clásica, cuyos costos en la actualidad son similares. Por otra parte se desea definir un conjunto de casos de prueba que ayuden en estas comparaciones. Por último evaluar el algoritmo desde el punto de vista de la calidad de los resultados así como también en el desempeño computacional de los algoritmos desarrollados.

1.2. Descripción del documento

Este documento se divide en seis capítulos y dos apéndices (Esto va a cambiar si agregamos apéndices para el artículo) cada uno con un objetivo particular, comenzando por introducir al lector en los temas que se tomarán como base para el desarrollo de este proyecto.

El primer capítulo es una introducción a temas generales de la generación de imágenes por computadora. En esta sección se describe el algoritmo a implementar así como también otros algoritmos relacionados y se introducen conceptos básicos necesarios para comprender los objetivos y los pormenores de los algoritmos de computación gráfica. Se descompone en tres sub-secciones que presentarán los conceptos básicos, una clasificación de los modelos de iluminación que siguen para modelar la forma de calcular luces y sombras basándose en los fenómenos naturales de la luz y una introducción

al algoritmo de *ray tracing* en ese orden.

El segundo capítulo toma como punto de partida el algoritmo de *ray tracing* descrito en la sección anterior y evalúa las distintas alternativas para la aceleración del algoritmo, así como también las posibles aplicaciones de paralelismos en su implementación. También introduce al estado del arte en el algoritmo, mostrando cuan potente puede llegar a ser.

En el capítulo tres se presenta la solución planteada para el proyecto. Se presenta la arquitectura de la solución, cómo se divide y como interactuan cada una de las partes para lograr el objetivo final: generar la imagen de manera correcta en el menor tiempo posible. Este capítulo introduce así mismo las distintas variantes implementadas y las mejoras introducidas en cada una de las versiones.

En el capítulo de análisis experimental se presentan las evaluaciones de los métodos relevados para la cuantificación de la calidad de las imágenes generadas y se presenta el conjunto de casos de prueba a utilizar, mostrando la relevancia de cada uno de ellos. Utilizando estos casos de prueba se realizan pruebas al algoritmo comparando distintas escenarios del algoritmo y algunas comparaciones con otras implementaciones, mostrando de forma tabular los resultados obtenidos.

La última sección muestra las conclusiones del trabajo y las posibles mejoras o cambios que se podrán introducir al algoritmo implementado. En esta sección se evalúa el cumplimiento de los objetivos planteados en esta sección así como los aportes a nivel personal y profesional a los autores y a los integrantes de la comunidad de investigadores del área.

Los apéndices muestran la estructura utilizada para el almacenaje de los datos necesarios para el algoritmo en el primero y en el segundo apéndice se muestran otras posibles estructuras que podrían haberse utilizado para acelerar el algoritmo, estas fueron evaluadas para poder decidir que estructura utilizar.

Capítulo 2

Generación de imágenes por computadora

2.1. Introducción

Este proyecto aborda la generación de imágenes fotorealistas en tiempos de cálculo bajos, que es una problemática actual de la computación gráfica. Existen muchos algoritmos para la generación de imágenes fotorrealistas, esto se evidencia por la cantidad de películas de animación con modelos 3D o que simplemente utilizan efectos 3D para realzar las escenas. Un tema importante es el tiempo de procesamiento que requieren este tipo de técnicas, cada una de las imágenes que se van a incluir en la versión final de la película implican varios minutos u horas dependiendo de la complejidad de la imagen a generar. Estos tiempos dependen de si la escena tiene reflejos o no, si tiene transparencias, si tiene muchos fragmentos pequeños de objetos, entre otros aspectos de calidad del modelo 3D a mostrar en la pantalla. No sólo es un problema los tiempos que se requieren para generar las imágenes, sino que además estas imágenes requieren de una capacidad de cómputo enorme. Por esto, en general, se utilizan potentes *clusters* de computadoras para realizar la generación de las imágenes.

Para comprender la forma en que se generan las imágenes foto-realistas por computadora es necesario conocer en profundidad: cómo se especifican los modelos y cuáles son las formas en que se puede, a partir de los modelos, generar las imágenes. En las siguientes secciones se introducen dichas temáticas, abordando conceptos esenciales tales como: qué es una escena y

los algoritmos comúnmente utilizados para generar imágenes. También se mostrarán los modelos para la iluminación que se utilizan, así como la clasificación de los algoritmos en base a estos modelos.

2.1.1. Concepto de escena

Una escena, en computación gráfica, es una colección de objetos y fuentes de luz que será vista por medio de una cámara. Cada una de estas partes está colocada en lo que se llama "mundo", que es un espacio resultante de modelar cuerpos tridimensionales en una imagen bidimensional [20]. Por ejemplo, si se quiere una imagen de una habitación con una mesa, la escena debe estar compuesta por dos objetos principales que representen la habitación y la mesa, una o más fuentes de luz, y la cámara, que es desde donde se ve la escena.

Cada objeto de una escena es una "primitiva geométrica", que por lo general es una figura geométrica simple como un polígono, una esfera ó un cono. Sin embargo las primitivas en una escena pueden ser matemáticamente más complejas, algunos ejemplos pueden ser superficies de Bezier, sub-divisiones de superficies, superficies ISO, etc. Casi cualquier tipo de objeto puede ser usado como primitiva de una escena.

2.1.2. Trazado de rayos

Un rayo es por lo general representado mediante un punto de origen O y una dirección D, $R(t) = O + tD$. En el marco de un algoritmo que traza rayos hay fundamentalmente tres problemas que deben ser resueltos: encontrar la intersección más cercana al origen del rayo O, encontrar alguna intersección a lo largo del rayo[1] y encontrar todas las intersecciones a lo largo de él. La clave de la eficiencia de cualquier tipo de algoritmo trazador de rayos es encontrar eficientemente la intersección de un rayo con una escena compuesta por una lista de primitivas geométricas.

La operación más utilizada en este tipo de algoritmos es obtener la intersección más cercana al origen del rayo. Los datos que se requieren son la primitiva P más cercana que interseca con el rayo y la distancia t_{hit} desde O al punto de intersección. Además pueden determinarse otros parámetros opcionales que serán utilizados en pasos posteriores del algoritmo, como pueden

[1] Se define t_{max}, si $t > t_{max}$ no se consideran las intersecciones.

ser propiedades de la superficie o la normal a la misma en el punto de intersección. Para la mayor parte de las primitivas usadas al construir una escena se dispone de diferentes algoritmos que evalúan la intersección con un rayo. Cada uno de los algoritmos tiene ventajas y desventajas respecto a propiedades como tiempo de ejecución, mantenibilidad, precisión o robustez lo cual hace que no resulte fácil la elección del mismo [1]. Las escenas serán entonces aptas para un algoritmo de trazas mientras sea posible evaluar su intersección con un rayo.

La segunda operación por orden de relevancia es la que determina si existe alguna intersección a lo largo del rayo. El problema que surge a partir de esta operación es igual a la prueba de visibilidad entre dos puntos, en este caso los puntos son: O y $O + t_{max}D$. Encontrar si existe alguna intersección en el camino del rayo es un problema más simple que encontrar la intersección más cercana. Si bien puede emplearse el mismo procedimiento que para encontrar la intersección más cercana, existen algoritmos más eficientes que resuelven este caso especial de trazado de rayo.

El tercer problema, encontrar todas las intersecciones a lo largo de un rayo, es el menos empleado y sólo es requerido para algoritmos de iluminación avanzados. Este problema no es común en los algoritmos trazadores de rayos, excepto para los que implementan modelos de iluminación especiales.

2.2. Modelos computacionales de iluminación

En esta sección se abordan las técnicas más populares para la generación de imágenes foto-realistas, comenzando por los algoritmos en el que se basan la mayoría de los algoritmos actuales:*ray casting* de Appel [4] y *ray tracing* de Whitted [36]. Hay que tener en cuenta que estos algoritmos no son los algoritmos más rápidos para la generación de imágenes. La técnica más popular para la generación de gráficos tridimensionales por computadora es rasterización que funciona en tiempo real. La técnica es simplemente el proceso de computar la correspondencia entre la geometría de la escena y los píxeles de la imagen y no tiene una forma particular de computar el color de esos píxeles. Esta técnica no tiene en cuenta el cálculo de sombras ni las reflexiones entre objetos, como si lo hace, entre otros *ray tracing*. Una de las implementaciones más usadas de la rasterización es *scan lines* [24].

2.2.1. *Ray casting*

El algoritmo de *ray casting* fue introducido por Arthur Appel en 1968 [4]. Es un algoritmo cuyo funcionamiento se basa en lanzar rayos desde el punto de vista del observador hacia un plano de vista que se encuentra entre el observador y la escena. La unidad mínima de visualización en los dispositivos actuales (monitores o dispositivos similares) es el píxel, cada uno de los cuadros de la grilla en la que se basa la visualización de imágenes. Por esto, el algoritmo genera tantos rayos como píxeles haya en el dispositivo de visualización a utilizar. También puede ser que se tenga un tamaño de imagen en píxeles, en este caso se genera un rayo por cada píxel de la imagen a generar. Las coordenadas de los píxeles se mapean a coordenadas del plano de vista, lanzando un rayo desde el punto de vista del observador que pase por la coordenada del plano de vista y calculando el punto de intersección con la escena, en caso de haberlo. Luego de hallado el punto de intersección con la escena se procede a calcular cuánta energía le llega al punto desde las fuentes de luz, sin tener en cuenta los posibles "rebotes" de la luz, como tampoco la posibilidad de que un objeto se encuentre interpuesto entre el objeto y la fuente de luz. Este algoritmo permite calcular fácilmente cuales son los objetos visibles además de facilitar la inclusión de objetos geométricos no planares en las escenas. Este último hecho, en el momento que se propuso el algoritmo, fue muy novedoso porque con los algoritmos que se utilizaban en la generación de gráficos no era posible incluir este tipo de objetos de forma sencilla. Los algoritmos utilizados en esa época eran algoritmos de *scan lines*, que se basan en rasterización mientras que en el algoritmo de *ray casting* los rayos no van más allá del primer objeto encontrado.

2.2.2. Clasificación de los algoritmos de generación de imágenes por computadora

Todos los algoritmos de generación de imágenes, independientemente de la categoría en la que se encuentren, dada una escena, definición matemática o algún tipo de representación abstracta, buscan generar una imagen. En el trabajo se referencia siempre al concepto de generación de imágenes realistas aunque los mismos algoritmos podrían ser utilizados para generar otro tipo de escenas. No obstante la diferencia de enfoque de los algoritmos, todos buscan de alguna manera modelar la cantidad de energía lumínica, o radiancia, que está presente en cada punto de los distintos objetos que forman la

escena, y así poder calcular de que manera se debería ver la imagen según los parámetros de iluminación que se establezcan. Los distintos modelos para calcular la iluminación que se identifican son los siguientes:

- el modelo planteado por Whitted, es el modelo más simple de iluminación y es de iluminación local.
- el modelo basado en elementos finitos, es bastante simple al igual que el modelo de Whitted pero a su vez muy diferente, ya que para calcular la radiancia plantea la división de la escena en pequeñas partes y utiliza alguna solución numérica para aproximar los valores [24]. Además, es un modelo de iluminación global.
- el modelo basado en métodos de Monte Carlo, el cual busca aproximar los valores de radiancia empleando aproximaciones estadísticas basadas en la integración de Monte Carlo [21]. Además, es un modelo de iluminación global. Dentro de los algoritmos que usan este tipo de métodos se encuentra su mayor exponente actualmente en el algoritmo de *photon mapping* [21].

Los algoritmos de generación de imágenes, que surgen en base al algoritmo original de *ray casting* y que emplean alguno de los modelos descritos anteriormente para el cálculo de la iluminación, pueden ser clasificados utilizando distintas estrategias, en este trabajo se presentan agrupados por el modelo de iluminación utilizado. Entonces, se identifican dos categorías de algoritmos:

- algoritmos de iluminación local: incluye los algoritmos que utilizan el modelo simple de iluminación local diseñado por Whitted.
- algoritmos de iluminación global: incluye los algoritmos que utilizan el modelo basado en elementos finitos o el basado en métodos de Monte Carlo.

Para cada modelo de iluminación presentado se describirá el algoritmo más representativo disponible. Estas técnicas que se presentan son: *ray tracing* para el modelo de Whitted, radiosidad para los algoritmos basados en elementos finitos y *photon mapping* para los basados en métodos de Monte Carlo.

Ray tracing

El algoritmo de *ray tracing* propuesto por Turner Whitted en 1980 [36] está basado en el algoritmo de *ray casting*. Whitted extendió la idea proponiendo hacer la traza de rayos recursiva. Entonces el algoritmo no termina cuando el rayo encuentra un objeto en su trayectoria, sino que en ese momento se hace la invocación recursiva del trazado de rayo, desde el punto de la intersección en caso de ser necesario. Con la posibilidad de la invocación recursiva del algoritmo se añade la capacidad de sombreado realista dado que se puede calcular la interposición de otros objetos de la escena entre el objeto y la luz. Al igual que *ray casting* es un algoritmo sencillo para la generación de imágenes, que tiene un modelo de iluminación propio que se basa en emular las características que cumple la luz al llegar a los objetos o al cambiar de un medio de transmisión a otro. Por ejemplo, al pasar del aire al agua, se genera una desviación de la luz, dando la impresión de que los objetos se deforman. Así mismo, introduce los conceptos de reflexión a las imágenes, admitiendo objetos que reflejan luz en las escenas, obteniendo un grado de realismo visual superior al generado por *ray casting*.

Radiosidad

El algoritmo de radiosidad utiliza los principios de *ray tracing* para el cálculo de las superficies visibles y sombras, así como las reflexiones y refracciones pero a diferencia del algoritmo de *ray tracing* básico plantea que para hacer el cálculo de la iluminación es necesario precalcular los valores de iluminación en cada uno de los parches en los que se divide arbitrariamente la escena, por esta división es que este es un método de elementos finitos. Luego de realizado el cálculo de la radiancia de cada uno de los parches se utiliza un rastreo de la escena para el cálculo de los valores de color de los píxeles de la imagen que se quiere generar. Como se precalculan los valores de iluminación en toda la escena se pueden utilizar los mismos valores de radiancia precalculados para generar imágenes desde distintos puntos de vista (variando la posición de la cámara), mientras no se modifique la escena y ni las fuentes de luz.

Photon mapping

El algoritmo de *photon mapping* fue introducido por Henrik Wan Jensen en el año 1996[21]. La técnica ha evolucionado fuertemente en los últimos

años y genera imágenes con una calidad excelente, además de ser menos costosa, en tiempo de cómputo, que el algoritmo de radiosidad. En lugar de utilizar el modelo de elementos finitos utiliza un modelo basado en métodos de Monte Carlo para el cálculo de la cantidad de energía en cada punto. El algoritmo de *photon mapping* tiene dos etapas diferenciadas al igual que el algoritmo de radiosidad, pero tiene una aproximación distinta para el cálculo de la radiancia de los puntos. La primera pasada es similar a la recorrida de la escena por el algoritmo de *ray tracing* con la diferencia que sigue el sentido inverso. Esta primer pasada se llama emisión de fotones, se generan fotones que son lanzados desde los emisores de luz hacia la escena en direcciones que sean factibles, cuantos más fotones se generen más fiable será el resultado de la iluminación. Se calcula el lugar en el que el fotón incide en la escena recursivamente de manera análoga al algoritmo de *ray tracing*. Esto es debido a que en el caso de los objetos reales, estos no absorben toda la luz incidente sino que hay luz que es reflejada y por lo tanto fotones son vueltos a lanzar desde el punto en el que chocaron con un objeto. Para hallar la dirección con la que es emitido el nuevo fotón y la energía que tendrá el mismo se utiliza un modelo para los materiales de los objetos de la escena teniendo que agregar al material de los objetos una función de BRDF (Función de Distribución de Reflectancia Bidireccional). Esta función representa la proporción de radiación reflejada por una determinada superficie en cada dirección del rayo reflejado, proyectada sobre el plano horizontal.

2.3. *Ray tracing*

Este trabajo profundiza el estudio del algoritmo de *ray tracing*, ya que por su simplicidad y versatilidad se presenta como la mejor alternativa para ser implementada. Además, este algoritmo puede verse como el padre de la familia de algoritmos actuales, tal como se presenta en la clasificación de algoritmos de generación de imagen.

En el algoritmo original de *ray tracing*, además de considerar las fuentes de luz para obtener sombras en la escena, el algoritmo de trazado de rayos recursivo de Whitted genera rayos de reflexión y de refracción desde el punto de intersección, como se muestra en la Figura 2.1.

Los rayos de sombra (L_i), reflexión (R_i) y refracción (T_i) son llamados secundarios para diferenciarlos de los primarios que son los que salen desde el punto de vista del observador o cámara.

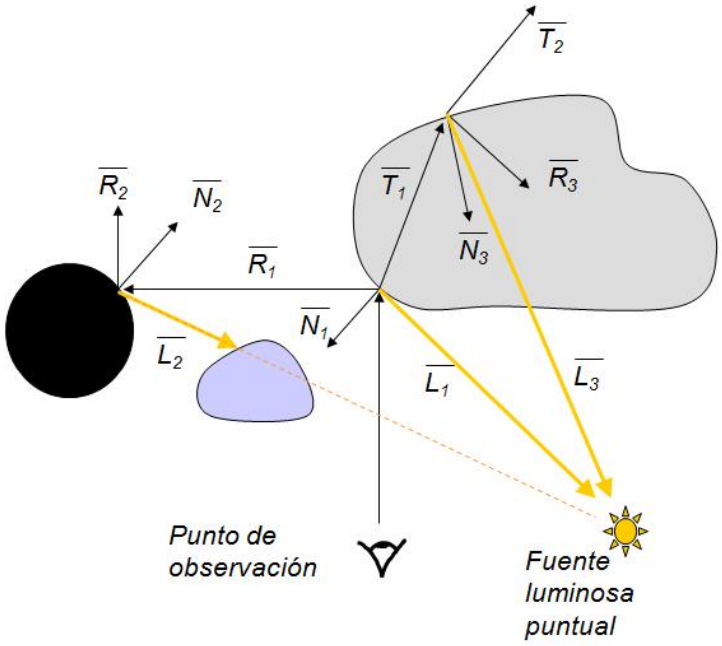


Figura 2.1: Generación de rayos del algoritmo de Whitted a partir de un único rayo primario.

En el primer nivel del algoritmo, cuando se traza un rayo primario, solo se tienen dos posibilidades: el rayo interseca con algún objeto de la escena o no lo hace. Si el rayo no encuentra ningún objeto en su camino, entonces, se debe usar el color de fondo de la escena para pintar ese píxel. Por el contrario, si encuentra un objeto en su trayectoria, se deben realizar los siguientes pasos en el punto de intersección:

- Paso uno: para calcular las sombras, se traza un rayo de sombra (L_1) desde el punto de intersección del rayo con el objeto hacia cada fuente de luz existente en la escena. Si alguno de estos rayos interseca cualquier objeto en su camino hacia la fuente de luz, dependiendo del material del objeto se debe calcular la cantidad de luz que pasa a través de él. Si el objeto es opaco, como es el caso del objeto más pequeño de la Figura 2.1, la luz es bloqueada totalmente y el punto de intersección estará bajo la sombra del objeto. Esto quiere decir que esta fuente de luz no será tomada en cuenta para calcular la iluminación en el punto. Si el objeto es transparente, como es el caso del objeto más grande de

la Figura 2.1, la intensidad de la fuente de luz se ve disminuida, incluso puede ser absorbida totalmente por el objeto. Existen tablas que indican que cantidad de luz es absorbida por cierto material transparente. En caso de que la luz no sea bloqueada totalmente por el objeto, esta contribuirá a la iluminación del punto de intersección del rayo primario.

- Paso dos: si el objeto tiene reflexión especular, como es el caso de la Figura 2.1, un rayo de reflexión es reflejado a partir del rayo primario, con respecto a la normal (N_1) en el punto de intersección, en la dirección del vector R_1. Este rayo permite obtener la cantidad de luz que llega al punto de intersección del rayo primario por el fenómeno de reflexión. Esta cantidad de luz puede verse afectada por el material del objeto, para considerar esto se usa un coeficiente dependiente del material, que escala la cantidad de luz.

- Paso tres: si el objeto es transparente, como es el caso de la Figura 2.1 y no ocurre refracción total, es decir si la luz no es absorbida totalmente por la transparencia que posee el objeto, entonces un rayo de refracción es trazado a través del objeto siguiendo la dirección del vector T_1. Esta dirección es calculada usando la ley de Snell [24]. Este rayo permite obtener la cantidad de luz que llega al punto de intersección del rayo primario por el fenómeno de refracción. Esta cantidad de luz puede verse afectada por el material del objeto, para considerar esto se usa un coeficiente dependiente del material, que escala la cantidad de luz.

Cada uno de los rayos de reflexión genera rayos de sombra, reflexión y refracción. Lo mismo sucede con cada uno de los de refracción. En el ejemplo de la Figura 2.1, para calcular la intensidad de luz aportada por R_1 se usan los mismos pasos que para calcular la intensidad aportada por el rayo primario. Por consiguiente los pasos dos y tres se deben calcular recursivamente. De esta manera se forma un árbol de rayos para cada rayo primario, como se muestra en la Figura 2.2.

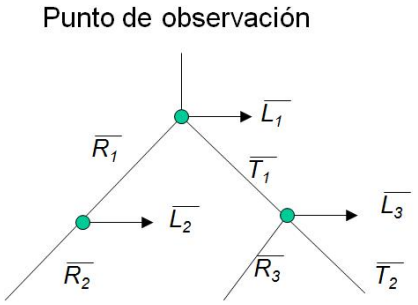


Figura 2.2: Árbol de rayos que surge del ejemplo de la Figura 2.1.

La profundidad del árbol de rayos afecta directamente el tiempo de ejecución del algoritmo y la calidad de la imagen que se obtiene. Dicha profundidad está determinada por distintos aspectos como por ejemplo un máximo dispuesto por el usuario del algoritmo o por no haber intersección entre los rayos reflejados y refractados y algún objeto o por la capacidad de almacenamiento del sistema donde ejecuta el algoritmo.

Luego de obtener la cantidad de luz aportada por cada uno de los pasos anteriores, están dadas las condiciones para calcular la iluminación en el punto de intersección del rayo primario. Para esto se debe recorrer un árbol de rayos (por ejemplo el de la Figura 2.2) de abajo hacia arriba, aplicando la ecuación de iluminación desarrollada por Whitted.

La ecuación de Whitted que se presenta en la Ecuación 2.1, considera tres componentes, la primera es la iluminación local, es decir, la iluminación dada por el ambiente y por las fuentes de luz de la escena pero sin considerar que los objetos reflejan o refractan luz. Esta primera parte usa la ecuación de iluminación de Phong [24]. La segunda ($k_s I_{r\lambda}$) y la tercera ($k_t I_{t\lambda}$) componente consideran la reflexión y la refracción de los objetos respectivamente.

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + \sum_{1 \leq i \leq m} S_i f_{att_i} I_{p\lambda_i} [k_d O_{d\lambda} (\overline{N} \cdot \overline{L_i}) + k_s (\overline{N} \cdot \overline{H_i})^n] + k_s I_{r\lambda} + k_t I_{t\lambda} \tag{2.1}$$

En la siguiente lista se puede observar el significado de cada variable presente en la Ecuación 2.1:

- $I_{a\lambda}$ - Intensidad de la luz ambiente: luz que ha sido esparcida por todo el ambiente y es imposible determinar su origen, cuando golpea una superficie se esparce igualmente en todas direcciones.

- k_a - Coeficiente de reflexión de luz ambiente: se encuentra entre 0 y 1. Determina la cantidad de luz ambiente reflejada por la superficie del objeto. Es una propiedad del material del objeto.
- $O_{d\lambda}$ - Componente difusa del color del objeto.
- m - Cantidad de luces de la escena.
- S_i - Indicador de sombra: indica si hay algún objeto entre la fuente de luz número i y el punto de evaluación. Toma el valor 1 si la luz no está bloqueada y 0 en caso contrario.
- f_{att_i} - Factor de atenuación para la luz número i: soluciona el problema de que dos superficies se vean iguales al estar a distinta distancia de una fuente de luz. Lo más común es usar el inverso del cuadrado de la distancia hacia la luz.
- $I_{p\lambda_i}$ - Intensidad de la fuente de luz número i en el punto de evaluación.
- k_d - Coeficiente de reflexión de luz difusa: se encuentra entre 0 y 1. Determina la cantidad de luz difusa reflejada por la superficie del objeto. Es una propiedad del material del objeto.
- k_s - Coeficiente de reflexión de luz especular: se encuentra entre 0 y 1. Determina la cantidad de luz especular reflejada por la superficie del objeto. Es una propiedad del material del objeto.
- $\overline{H_i}$ - Vector de dirección media o vector de iluminación máxima: vector utilizado por la ecuación de iluminación de Phong [24]. Se calcula como la dirección media entre el vector normal y el vector que indica la dirección del observador.
- n - Exponente de ajuste de la iluminación: este exponente sirve para ajustar la imagen, no es un resultado teórico sino que es resultado de la observación empírica.
- $I_{r\lambda}$ - Intensidad del rayo reflejado: esta intensidad es determinada evaluando recursivamente la Ecuación 2.1.
- k_t - Coeficiente de trasmisión: se encuentra entre 0 y 1. Determina la cantidad de luz que pasa a través del objeto. Es una propiedad del material del objeto. Existen tablas con valores para distintos materiales.

- $I_{t\lambda}$ - Intensidad del rayo refractado: esta intensidad es determinada evaluando recursivamente la Ecuación 2.1.

Algoritmo de *ray tracing*

El algoritmo de *ray tracing* tiene como ventajas la simplicidad de su implementación, así como también el realismo que logra frente a otros métodos como la Rasterización. Las simplificaciones que utiliza el modelo de iluminación no permiten que se generen envolventes de los rayos de luz reflejados o refractados por una superficie curva. A los efectos generados por este fenómeno se les llama cáusticas.

Otra simplificación en el cálculo de la iluminación es la introducción de un componente de color de "luz ambiente", luz que tiene origen en alguna fuente de luz desconocida y parece llegar de todas las direcciones, esto permite no calcular algunos rebotes de la luz en objetos de la escena que harían más complejo al algoritmo. Dada esta última simplificación tampoco se generan efectos de "sangrado de luz", este fenómeno es causado por la reflexión de luz de los objetos en forma parcial que hace que el color de una pared, por ejemplo, sea extendido por la zona del suelo cercana a la pared, dando la idea de que la pared "sangra" color sobre el suelo.

Una de las principales desventajas que muestra el algoritmo es el costo computacional, en especial en los modelos utilizados por la mayoría de las aplicaciones 3D basados en la rasterización de imágenes formadas por polígonos. Por este motivo, *ray tracing* no es una técnica utilizable para la aplicaciones que necesiten mostrar imágenes que se actualicen en tiempo real. Sin embargo, en los últimos tiempos se han desarrollado diferentes esfuerzos por alcanzar tiempo real en aplicaciones basadas en *ray tracing*, como por ejemplo el *framework* para video-juegos *Quake Wars* [14] que emplea el motor *openRT* [13]. Una demostración del *framework* hecha en agosto de 2008 genera entre 20 y 35 imágenes de 1024 por 720 píxeles por segundo, utilizando una plataforma de ejecución *Caneland*, la cual esta compuesta por cuatro sistemas *Dunnington* de seis núcleos.

En el Algoritmo 1 se muestra un pseudo-código del algoritmo de *ray tracing*.

En el Algoritmo 2 se presenta el pseudo-código de la función *trazarRayo*. Cada objeto de la escena es analizado para probar si el mismo es atravesado por el rayo; del conjunto de objetos atravesados interesa el objeto que tiene el punto de intersección más cercano al observador. Una vez obtenido el punto

de intersección más cercano (si existe) se aplica la Ecuación 2.1.

La función del Algoritmo 2 que verifica si el punto de intersección más cercano esta en sombra se resuelve lanzando un rayo desde el punto de intersección hacia cada uno de los focos de luz para comprobar cuanta luz incide en el objeto. Si todos los rayos intersecan a un objeto antes de llegar al foco de luz entonces el punto está en sombra, caso contrario se tendrá alguna función que calcule cuanto aporta el foco a la iluminación del punto. Si el objeto atravesado más cercano tiene reflexión se genera un rayo reflejado con origen en la intersección y cuya dirección es calculada en función del ángulo de incidencia del rayo original sobre la superficie del objeto. Si la superficie del objeto tiene refracción se genera un rayo refractado con origen en la intersección cuya dirección es calculada en base a las densidades de los medios por los que atraviesa el rayo utilizando, por ejemplo, la ley de Snell [24]. Los rayos reflejado y refractado se usan para invocar recursivamente. Con el color del objeto, el trazado de los rayos de sombra y las dos invocaciones recursivas, de reflexión y refracción se calcula el color del píxel invocando a la función $calcularColorFinal$. Esta función aplica la ecuación de iluminación del modelo de Whitted.

Algoritmo 1 Pseudo-código del algoritmo de *ray tracing*.

para todo píxel p en imagen a generar **hacer**
    $r = rayo(observador, p)$;
    $p.color = trazarRayo(r, 1)$;
fin para

Algoritmo 2 Pseudo-código de la función trazarRayo.

Entrada: Rayo r, Entero $profActual$
Salida: Color *color*

```
si profActual < MAXPROF entonces
    devolver colorNulo;
fin si
objetoMasCercano = ∞;
para todo objeto o en la escena hacer
    si hayInterseccion(o, r) entonces
        si masCercaObservador(o, objetoMasCercano) entonces
            objetoMasCercano = o;
        fin si
    fin si
fin para
si objetoMasCercano <> ∞ entonces
    sombra = verificarSombra(objetoMasCercano, r, luces);
    si objetoMasCercano es reflectivo entonces
        rR = rayoReflejado(objetoMasCercano, r);
        reflex = trazarRayo(rR, profActual + 1);
    fin si
    si objetoMasCercano es transparente entonces
        rT = rayoRefractado(objetoMasCercano, r);
        refrac = trazarRayo(rT, profActual + 1);
    fin si
    color = calcularColorFinal(objetoMasCercano, sombra, reflex, refrac);
si no
    color = obtenerColorFondo(r);
fin si
```

Capítulo 3

Aceleración del algoritmo de *ray tracing*

3.1. Introducción

Esta sección presenta en detalle los algoritmos y técnicas existentes para la aceleración de la generación de imágenes por computadora. Se verán los métodos que fueron analizados durante el desarrollo de este trabajo, centrándose en dos puntos principales que son: las estructuras de aceleración espacial y la paralelización de procesos. Por un lado se busca reducir la cantidad de cálculos requeridos para la generación de una imagen y por otro lado hacer que los cálculos se realicen en menos tiempo. Para la paralelización de algoritmos en este proyecto se optó por utilizar una paralelización del algoritmo en la GPU, en este cápitulo se justifica además por que se consideró esta plataforma para el proyecto.

3.2. Paralelización de algoritmos en GPU

En noviembre de 2006 NVIDIA lanzó CUDA (Compute Unified Device Architecture), una arquitectura de computación paralela de propósito general que hace uso del núcleo de procesamiento paralelo de las GPUs de NVIDIA para resolver una amplia variedad de problemas computacionales de una manera más eficiente que en una CPU. El modelo de programación paralela propuesto por NVIDIA fue totalmente nuevo y la programación sobre el mismo se hace a través del lenguaje de alto nivel CUDA que permite el uso

del lenguaje C para la programación.

La arquitectura CUDA basa su poder de procesamiento en un conjunto de multiprocesadores (variable dependiente del modelo de GPU), donde cada uno de ellos procesa parte de la carga de trabajo en paralelo. Un multiprocesador está compuesto por 8 procesadores, una unidad de memoria compartida y otras 3 unidades que permiten controlar el funcionamiento del mismo. Cada multiprocesador crea, administra y ejecuta hilos concurrentemente en el hardware sin incrementar el tiempo de ejecución por la planificación (*scheduling*).

El modelo de programación esta basado en tres abstracciones fundamentales: una jerarquía de grupos de hilos de ejecución, memoria compartida y barreras de sincronización. Estas abstracciones guían al programador a dividir el problema en sub-problemas que pueden ser resueltos en forma paralela e independiente por medio de bloques de hilos de ejecución. A su vez cada sub-problema de divide en piezas más chicas que pueden ser resueltas en paralelo y cooperando entre ellas, usando los hilos de ejecución de cada bloque. Descomponer el problema de esta forma permite la escalabilidad automática en el número de procesadores, ya que cada bloque de hilos puede ser despachado hacia cualquier conjunto de procesadores (multiprocesador) disponible en cualquier orden, concurrentemente o secuencialmente. De esta manera una aplicación CUDA puede ejecutar sobre un equipo con diferente número de multiprocesadores y sólo se necesita conocer este número en tiempo de ejecución, lo que implica que no sea necesaria una nueva compilación. En la Figura 3.1 se considera una aplicación CUDA que está dividida en cuatro bloques y se muestra como se asignan los bloques de hilos de ejecución en dos GPU distintas, donde una tiene dos multiprocesadores y la otra tiene cuatro.

La extensión del lenguaje C que hace CUDA habilita a definir funciones (de la misma forma que en el lenguaje base) que a diferencia de las funciones comunes de C, cuando son invocadas ejecutan N veces en paralelo, mediante N hilos de ejecución diferentes. Estas funciones propias de CUDA son llamadas *kernels*. Dentro de este tipo especial de funciones se tiene acceso a información propia de CUDA que indica por ejemplo, el identificador del hilo de ejecución o el identificador de bloque que lo contiene. Esta información es de vital importancia ya que es usada para parametrizar la ejecución del *kernel* en función de los hilos de ejecución.

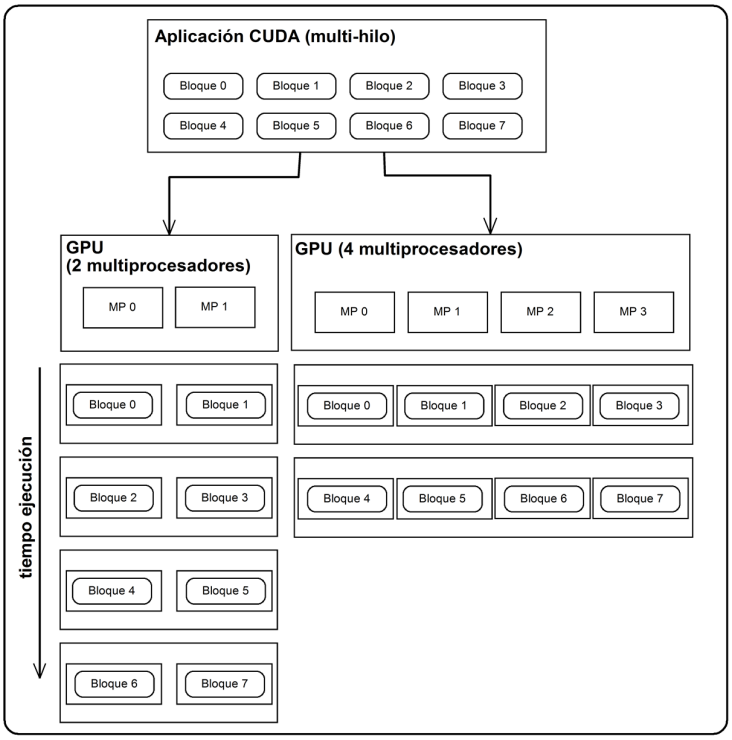


Figura 3.1: Ejemplo de escalabilidad automática en el número de multiprocesadores.

Un *kernel* siempre es invocado desde el *host* (CPU) y ejecuta en el *device* (GPU). La GPU actúa como co-procesador de la CPU y mediante *kernels* la CPU puede asignar trabajo al co-procesador. En la Figura 3.2 se muestra un ejemplo de una aplicación CUDA que posee un *kernel* "*kernel A*". Esta aplicación comienza ejecutando código secuencial en la CPU, en la primer parte secuencial se hace la invocación al *kernel*, el cual ejecuta en la GPU. Una vez terminada la ejecución a nivel de GPU retorna a ejecutar código secuencial en la CPU.

El índice que identifica a un hilo de ejecución es un vector tridimensional, usando el mismo un hilo puede ser identificado usando una, dos o tres componentes del vector. De esta manera los hilos pueden forman un bloque de una, dos o tres dimensiones. Esta forma de agrupar los distintos hilos de un bloque permite invocar *kernels* sobre distintos tipos de dominio (vector, matriz o volumen) de una manera más natural. Los bloques también pueden ser agrupados en una grilla (*grid*) de una o dos dimensiones.

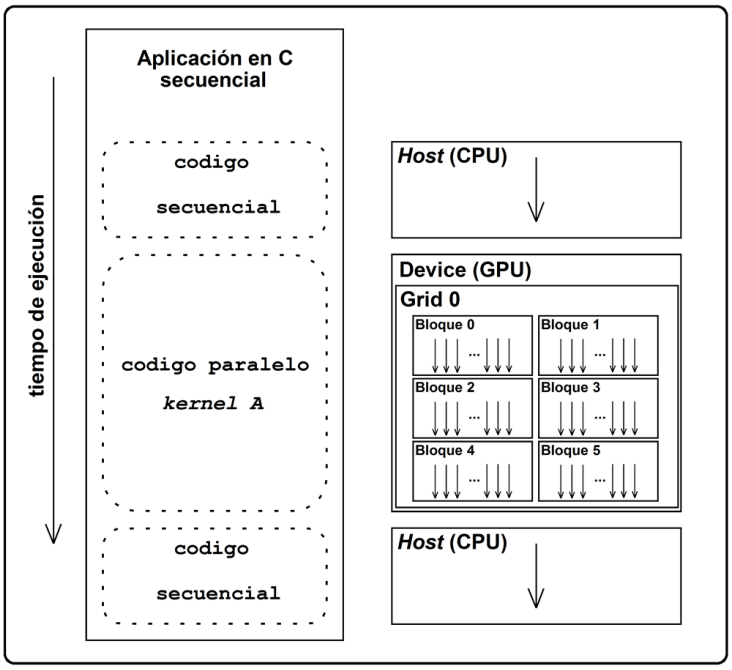


Figura 3.2: Ejemplo de ejecución de una aplicación CUDA.

Como se muestra en la Figura 3.3, cada hilo de CUDA puede acceder a múltiples espacios de memoria durante su ejecución, mientras que los diferentes espacios de memoria forman la jerarquía de memoria de la GPU.

En el primer nivel de la jerarquía, donde se da la latencia más baja y la menor capacidad de almacenamiento, se encuentran los registros. Desde la arquitectura menos avanzada (*Compute Capability 1.0*), cada registro tiene 32 bits para almacenar un entero o un punto flotante. Cada hilo de ejecución tiene acceso a una cierta cantidad de registros, donde la cantidad depende del modelo de la GPU y la cantidad de hilos del bloque, y puede llegar hasta un máximo de 4096 en los modelos más avanzados (*Compute Capability 2.0*). Además cada hilo de ejecución tiene su propio espacio privado de memoria local. La memoria local a cada hilo es pequeña, se dispone de 16 KB en la arquitectura menos avanzada y su latencia es relativamente alta (tanto como la memoria global).

Cada bloque tiene un espacio de memoria compartida entre todos sus hilos, este espacio compartido tiene el mismo tiempo de vida que el bloque, es decir, es válido mientras el bloque se encuentra en ejecución. El espacio de memoria compartida de cada bloque es de 16 KB, posee una latencia

baja, similar a la de los registros. Puede ser controlada totalmente por el programador y puede ser usada como un caché para mejorar los tiempos de acceso a la memoria global.

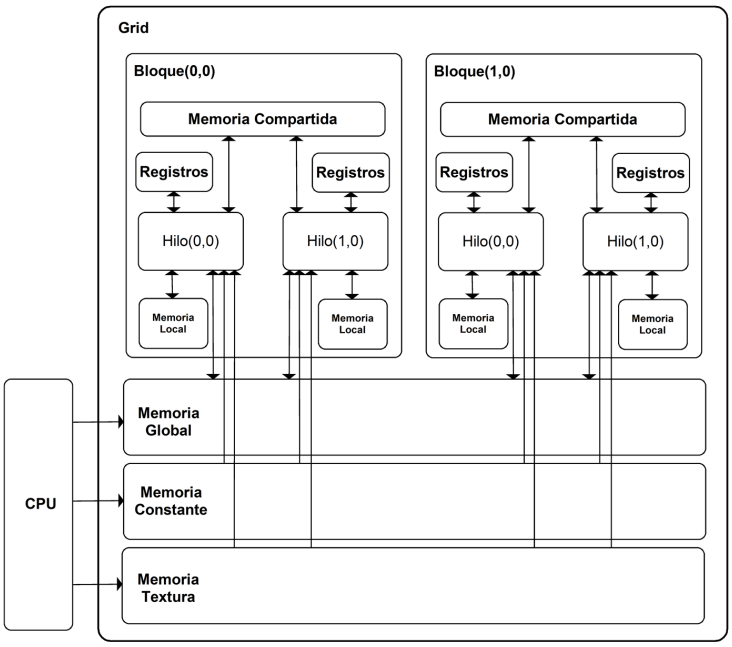


Figura 3.3: Jerarquía de memoria de la GPU.

Además todos los hilos de ejecución de la aplicación tienen acceso a un mismo espacio de memoria global. Este espacio es el que tiene la mayor capacidad de almacenamiento dentro de la jerarquía de memoria llegando hasta 4 GB en los modelos *Tesla C1060*. La memoria global ofrece un alto ancho de banda (superior a 100 GB/s en el modelo *Tesla C1060*) pero padece de latencia alta (cientos de ciclos) y no posee caché.

Existen dos espacios más de sólo lectura en la jerarquía que son accesibles por todos los hilos: el espacio de memoria constante y el espacio de memoria de textura. Ambos espacios de sólo lectura tienen tiempo de vida igual al tiempo de ejecución de la aplicación CUDA y cada uno esta optimizado para distintos usos [12]. Ambos espacios de memoria tienen la misma latencia que la memoria global aunque la de textura posee un caché de 8 KB por cada bloque, lo cual mejora el tiempo de acceso a los datos.

3.3. Métodos de aceleración para *ray tracing*

A partir del gran consumo computacional que significa trazar grandes cantidades de rayos, es vital encontrar métodos de aceleración para este proceso. Whitted al momento de desarrollar su algoritmo (en el año 1980) marcó el alto costo en tiempo en el trazado de rayos, desde ese momento hasta la actualidad se han propuesto diversas técnicas que buscan optimizar este proceso [34].

Los métodos de optimización existentes pueden agruparse en dos categorías, según la forma de abordar el problema. A la primer categoría pertenecen las técnicas que apuntan a reducir el número de rayos a trazar.

Para disminuir la cantidad de rayos a su vez existen dos estrategias:

- Construir la imagen lanzando menos rayos primarios. Un ejemplo de este tipo de técnicas es la llamada *adaptive sampling*. Usando este método, para cada píxel de la imagen, se trazan rayos primarios por cada uno de sus vértices. Si la intensidad de la luz en cada una de las esquinas del píxel varía significativamente con respecto a las otras, entonces el píxel es dividido en cuatro partes iguales. Luego se lanzan rayos primarios por cada nueva parte de la misma forma que se lanzaron en el píxel original, repitiendo esta división hasta lograr la calidad deseada. Una vez terminada la sub-división, el color del píxel es interpolado según el color de cada una de sus partes [34].

- Construir la imagen lanzando menos rayos secundarios. Un ejemplo de este tipo de técnicas es la llamada *shadow caching*. La mayoría de los rayos que se tranzan en un algoritmo de *ray tracing* son rayos de sombra, ya que por cada rayo primario se trazan varios rayos de sombra. Para cada rayo de sombra se debe verificar si hay algún objeto en su camino hacia la fuente de luz y alcanza con que interseque con uno para garantizar oclusión. El caché de sombra explota el hecho de que muchos rayos se sombra son similares (sobre todo los que son originados por la misma fuente de luz) y que además intersecan con el mismo objeto [34].

A la segunda categoría pertenecen las técnicas que buscan acelerar la intersección de los rayos con la escena. Se puede lograr acelerar el núcleo de los algoritmos de *ray tracing* por varios caminos: eligiendo cuidadosamente las primitivas usadas para la construcción de las escenas y sus algoritmos

de intersección, usando volúmenes envolventes que permitan descartar primitivas que no sean alcanzadas por el rayo o utilizando divisiones espaciales (estructuras de aceleración espacial) de la escena que garanticen no recorrer toda su lista de objetos por cada rayo que la atraviesa [34].

Entre las estrategias más importantes para disminuir el tiempo de ejecución del algoritmo de *ray tracing* se encuentra el uso de estructuras de aceleración espacial. Al momento de llevar a cabo cualquier implementación no trivial del algoritmo se deben considerar este tipo de técnicas ya que las mismas permiten reducir su orden (de $O(N)$ a $O(\log N)$) [34]. En la siguiente sub-sección se presentan las estructuras de aceleración espacial más usadas.

3.4. Estructuras de aceleración espacial

Si se consideran todos los chequeos necesarios para generar una imagen utilizando el algoritmo de *ray tracing*, estos pueden llegar a implicar el 95 % del tiempo de cálculo [28]. Se han desarrollado técnicas para optimizar el tiempo que toman las intersecciones rayo-objeto, basadas en tratar de minimizar el número de intersecciones. A continuación se muestran las técnicas más importantes de este tipo según el trabajo de Thrane y Ole [28].

3.4.1. Subdivisión espacial

En el método de sub-división espacial el volumen de la escena se divide en regiones. A cada región se le asigna una lista con todos los objetos que contiene, total o parcialmente. Estas listas se completan asignando a cada objeto la celda o las celdas que lo contienen. Esta técnica requiere un preproceso para crear la estructura de datos donde quedará registrada la información relativa al espacio que ocupan los objetos en la escena.

El preproceso consiste en dividir el volumen total de la escena en pequeños volúmenes o cajas. La forma de definir estas cajas es lo que marca la diferencia entre las técnicas de sub-división espacial. Una vez que las cajas están definidas juegan el mismo papel en todas ellas.

La gran ventaja de esta técnica de sub-división es que solo los objetos asignados a las cajas atravesadas por los rayos deben ser probados para una posible intersección.

Sub-división espacial uniforme

Cuando las particiones son todas del mismo tamaño la técnica se denomina sub-división espacial uniforme (SEU). En la Figura 3.4 se muestra un ejemplo de este tipo de sub-división, que es totalmente independiente de la estructura de la escena.

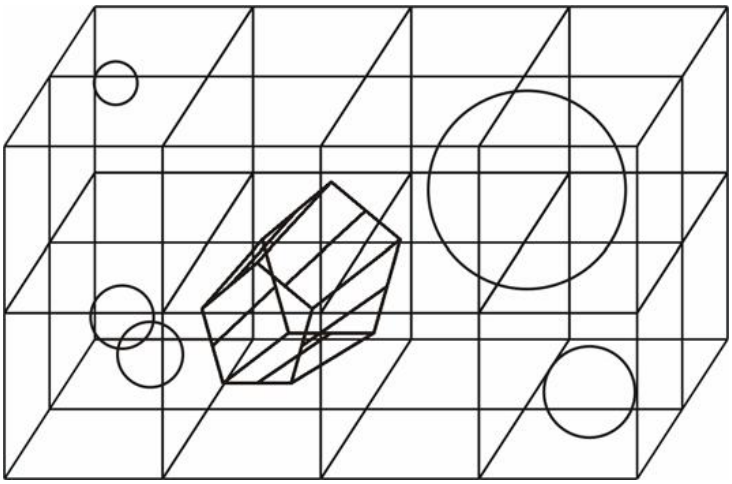


Figura 3.4: Sub-división espacial uniforme aplicada a una escena.

Otro aspecto a tener en cuenta es que las cajas se procesan en el mismo orden en que son encontradas por el rayo, lo que garantiza que cualquier caja atravesada por un rayo estará más cerca del origen del mismo que las restantes. Por consiguiente, una vez encontrado un punto de intersección, por lo general no será necesario considerar el contenido de las restantes cajas. Esto reduce considerablemente el número de objetos que se han de probar para intersección. Cuando un rayo atraviesa una caja, se tiene que averiguar si hay intersección con cada uno de los objetos contenidos en ella, y se debe escoger la intersección que se encuentre más cercana al origen del rayo.

La construcción de la sub-división espacial uniforme se realiza de la siguiente forma: dados los bordes de la escena y la lista de objetos de esta se puede construir la sub-división espacial uniforme, el único parámetro necesario para su construcción es su resolución a lo largo de los tres ejes imaginarios.

Según Thrane y Ole [28] no hay una técnica que garantice la mejor resolución o por lo menos no para todos los casos. En el mismo trabajo se sugiere

que la resolución al utilizar SEU sea $3\sqrt[3]{N}$ cajas a lo largo del eje más corto, donde N es el número de triángulos de la escena. Aunque también se menciona que puede ajustarse empíricamente para lograr mejores resultados en imágenes particulares. Una vez que se determina la resolución, es posible construir una matriz tridimensional de listas de objetos que servirá para manejar las cajas construidas y su contenido. Luego para cada objeto de la escena, se deben encontrar las cajas que lo contienen y agregar una referencia al objeto a cada una de ellas.

La técnica usada para moverse a través de las cajas de la grilla es equivalente (para tres dimensiones) a la técnica para dibujar una línea en dos dimensiones, cuyo algoritmo es denominado Digital Differential Algorithm (DDA) [24]. Basados en DDA, Fujimoto et al. [2] proponen el algoritmo que permite recorrer la grilla, que es usado por Thrane y Ole en su trabajo, con algunas mejoras propuestas por Amanatides y Woo [28]. En el Algoritmo 3 se muestra un pseudo-código de la técnica para dos dimensiones para facilitar la comprensión (extenderla a tres dimensiones es simple).

Algoritmo 3 Pseudo-código de la recorrida de las cajas atravesadas por un rayo.

mientras X y Y estén dentro de la grilla **hacer**
    chequeo de intersección con los triángulos de la caja actual
    si hay intersección en esta caja **entonces**
        se detiene el algoritmo y se retorna la intersección
    fin si
    si $tmax_x < tmax_y$ **entonces**
        $X \leftarrow X + step_x$
        $tmax_x \leftarrow tmax_x + delta_x$
    si no
        $Y \leftarrow Y + step_y$
        $tmax_y \leftarrow tmax_y + delta_y$
    fin si
fin mientras
devolver no hay intersección

Antes de comenzar la ejecución del Algoritmo 3, se debe identificar la caja inicial, es decir la primer caja que atraviesa el rayo. Si el origen del rayo se encuentra dentro de una caja determinada, entonces esta es la inicial. En caso contrario, se busca el primer punto de la grilla que interseca con el rayo

y se usa este punto para localizar la caja inicial. Las coordenadas de este se guardan en las variables X e Y. Además, se deben crear las variables $step_x$ y $step_y$, cuyos valores serán ± 1 dependiendo del signo de las componentes x e y del vector dirección del rayo. Estos valores serán usados para incrementar o decrementar las variables X y Y, y así ir avanzando a lo largo de la trayectoria del rayo. Lo próximo que se necesita es la máxima distancia que se puede avanzar a lo largo de la trayectoria del rayo antes de cruzar un borde vertical u horizontal de una caja. Estas distancias están representadas por las variables $tmax_x$ y $tmax_y$ respectivamente (ver Figura 3.5). El mínimo entre estas dos variables determina la máxima distancia que se puede avanzar a través de la trayectoria del rayo sin salir de los bordes de la caja actual. Por último se calculan $delta_x$ y $delta_y$. La primera indica la distancia horizontal (en la trayectoria del rayo) que se debe avanzar para pasar a la siguiente caja. Esta se calcula de la siguiente manera:

$$delta_x = \frac{tamcaja_x}{rayodireccion_x}$$

La segunda variable indica la distancia vertical y se calcula de la misma forma. Luego de la inicialización de estas variables se usa un algoritmo (Algoritmo 3) incremental simple para avanzar a lo largo de las cajas que el rayo atraviesa.

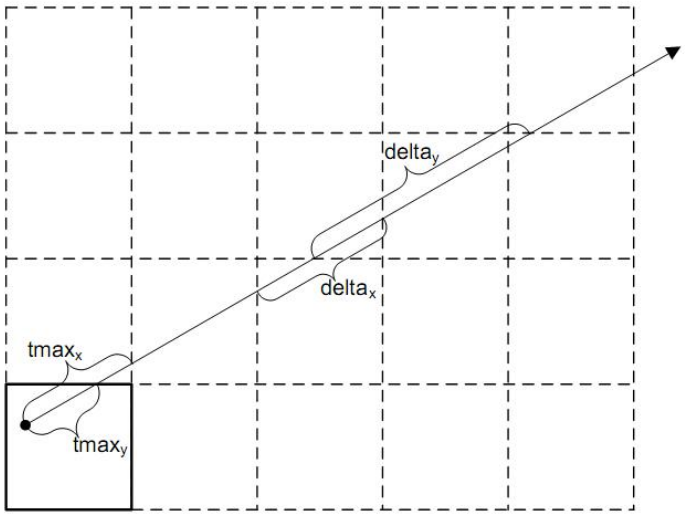


Figura 3.5: Relación entre el rayo, $tmax_x$, $tmax_y$, $delta_x$ y $delta_y$. La caja inferior izquierda contiene el punto origen del rayo.

La sub-división espacial uniforme fue implementada por primera vez sobre una GPU por Purcell y se convirtió en la única estructura de aceleración en ser paralelizada sobre GPU hasta ese momento (año 2004) [29]. En el trabajo de Thrane y Ole se resumieron las principales razones por las cuales se considera a esta estructura buena para ser paralelizada sobre una GPU. Estas razones son:

- Cada caja atravesada por el rayo puede ser localizada y accedida en tiempo constante usando aritmética simple. Esto elimina la necesidad de recorrer árboles (como en otras estructuras de aceleración) y por lo tanto, el manejo de mucha información a nivel de la GPU, lo cual resulta muy costoso.

- El desplazamiento a través de las cajas atravesadas se hace de forma incremental y con sumas sencillas, lo cual elimina la necesidad de una pila y hace posible visitar las cajas en orden, es decir aumentando la distancia desde el origen del rayo.

- Se puede explotar el hecho de que las cajas se recorren en orden para detener la recorrida cuando se de una intersección en la caja actual.

- El algoritmo de recorrido a través de las cajas que interseca el rayo esta dado por un vector lo cual es altamente compatible con el conjunto de instrucciones de una GPU.

En la sub-división espacial se puede dar el caso de que un polígono sea referenciado por más de una caja. Esto genera que en ocasiones especiales, se haga más de una vez el mismo test de intersección. Para resolver esto se han generado técnicas como la denominada *mailboxing*, que mantiene una tabla donde se asocia cada rayo con el último polígono al cual se le realizó el test de intersección [25]. Al emplear estrategias de la paralelización este tipo de técnicas no pueden utilizarse, lo que implica que el algoritmo paralelo debería hacer chequeos repetidos [28].

Kd-Trees

Al igual que la sub-división espacial uniforme la estructura *kd-tree* es una instancia particular de la sub-división espacial. La diferencia que tiene es que representa la escena con una estructura jerárquica basada en un árbol binario. En esta estructura se hace una distinción con respecto al tipo de los nodos,

se distinguen los nodos internos de las hojas. Los nodos hoja se corresponden con las cajas y tienen las referencias a los objetos que se encuentran dentro de las mismas. Los nodos internos se corresponden con la forma en que se divide el espacio. De esta manera, los nodos internos contienen un plano de corte y referencias a cada uno de los dos sub-árboles, mientras que los nodos hojas contienen listas de objetos.

Esta técnica de división del espacio tiene prácticamente las mismas ventajas que la sub-división espacial uniforme. Pero esta división intenta mejorar a la uniforme considerando que los objetos no están uniformemente distribuidos en la escena.

La construcción de un *kd-tree* se hace de forma recursiva, siguiendo un enfoque top-down. Dada una caja que contenga completamente a la escena y una lista de objetos contenidos en ella, se escoge un plano de corte perpendicular a uno de los ejes de coordenadas, que divida la caja en dos. Al dividir se generan dos nuevos volúmenes, y cada uno de ellos es representado agregando un hijo al nodo asociado a la caja original. Cada uno de los objetos que contiene la caja original es asignado al nodo hijo que lo contiene. En caso de que un objeto tenga intersección no vacía con el plano de corte, entonces este objeto es asignado a ambos hijos. Este procedimiento continúa hasta que se alcanza una profundidad definida de antemano o hasta que el número de objetos contenidos en cada caja sea menor a un número definido anteriormente. Havran [33] sugiere que la profundidad máxima sea igual a 16 y que la cantidad de objetos por caja sea 2 para lograr un rendimiento óptimo. Thrane y Ole [28] analizaron estos valores y concluyeron que 16 como profundidad máxima no era un buen valor para las escenas realistas. Se concluyó que, como la escena es dividida en dos en cada nivel del árbol entonces, una profundidad más convincente sería una que fuera resultado de una función logarítmica en el número de triángulos de la escena. Esta consideración también fue adoptada por Pharr y Humphreys [27], los cuales usaron $8 + 1.3 \log(N)$ como profundidad máxima, donde N es el número de triángulos de la escena.

En el Apéndice B se presenta un relevamiento más profundo del estado del arte de la estructura *kd-tree*.

3.4.2. Jerarquía de Volúmenes Envolventes (BVH)

La estructura BVH divide la escena y guarda la información de la división en una jerarquía definida por un árbol. Difiere de las técnicas de sub-división

espacial porque no divide el espacio sino que divide objetos. El volumen envolvente de una pieza de geometría es un objeto geométrico simple que la envuelve, es decir que la contiene en su interior. Claramente, si falla la intersección de un rayo con el volumen envolvente de un objeto, falla la intersección con cualquier cosa que este dentro del mismo y por lo tanto falla la intersección rayo-objeto.

La motivación para usar volúmenes envolventes es que realizar un chequeo de intersección con un objeto simple, como lo es un volumen envolvente, es mucho menos costoso que hacerlo contra el objeto que contiene dentro, que por lo general no es un objeto simple. La aceleración que pueda lograrse mediante esta técnica dependerá de la complejidad de los objetos de la escena y de los volúmenes envolventes que se usen.

Una jerarquía de volúmenes envolventes esta formada por un nodo raíz que contiene un volumen que envuelve a todos los demás volúmenes, y también contiene a todos los objetos de la escena. Cada nodo interno del árbol tiene como hijos a un conjunto de nodos internos, cada uno de ellos con un volumen envolvente asociado, o a un conjunto de nodos hoja, con un número cualquiera de objetos de la escena asociados. En la Figura 3.6 se muestra una estructura BVH como ejemplo, la cual utiliza cajas alineadas a los ejes como volúmenes envolventes. Es posible utilizar otros objetos envolventes, por ejemplo, cajas no alineadas a los ejes, cilindros, esferas, etc.

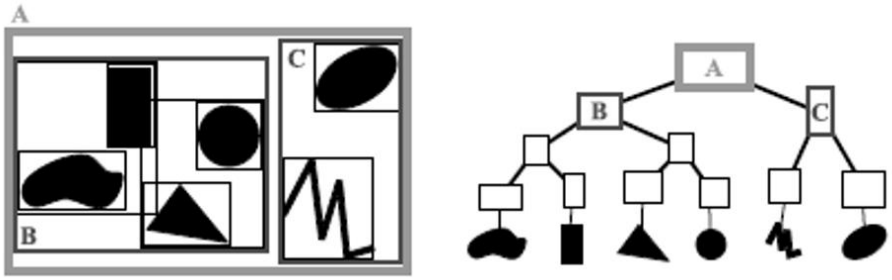


Figura 3.6: Ejemplo de estructura BVH que utiliza cajas alineadas a los ejes.

El algoritmo de recorrida de los volúmenes envolventes es realizado usando un simple e intuitivo descenso recursivo.

Una medida razonable de la calidad de una estructura BVH, es el costo promedio de aplicarle el algoritmo de recorrida, dado un rayo arbitrario. No hay ningún algoritmo conocido que construya estructuras BVH óptimas,

tampoco es obvio como evaluar el costo promedio de atravesar con un rayo arbitrario una estructura de este tipo. Goldsmith y Salmon proponen una función de costo conocida como la heurística del área de las superficies [19]. Está formalizada usando el área de la superficie del nodo padre y del nodo hijo y sigue la relación de la Ecuación 3.1.

$$P(hit(c)|hit(p)) \approx \frac{S_c}{S_p} \tag{3.1}$$

Donde: $hit(n)$ es el evento en que el rayo atraviesa el nodo n, S_n es el área de la superficie del nodo n y c y p son el nodo hijo y padre, respectivamente.

La función da un estimativo del costo de la jerarquía cuando se trata de atravesar por un rayo cualquiera.

Como no existe un algoritmo para construir eficientemente una estructura BVH óptima, se han propuesto heurísticas de construcción. Por lo general, estas heurísticas se basan en una de las dos ideas propuestas por Kay y Kajiya [26] y Goldsmith y Salmon [19], en estos trabajos se puede profundizar sobre el tema.

En el Apéndice B se presenta un relevamiento más profundo del estado del arte de la estructura BVH.

3.4.3. Conclusiones sobre aceleración espacial

En el trabajo de Thrane y Ole se analizan experiencias obtenidas al implementar y usar la sub-división espacial uniforme en la paralelización del algoritmo de trazado de rayos. Para escenas con objetos simples la aceleración lograda a través de esta estructura es mayor que la obtenida a través de *kd-tree* o BVH. Cuando las escenas poseen objetos complejos la aceleración no es tan buena en comparación con las mismas estructuras. Además se menciona que la estructura no es buena para escenas donde se dan grandes variaciones en la densidad de los objetos desde el punto de vista geométrico. También se destaca que el algoritmo para recorrer las cajas es muy simple de implementar y que requiere pocas operaciones aritméticas para avanzar de caja en caja. Así mismo el procedimiento para la construcción de la SEU es más simple que el de las estructuras *kd-tree* o BVH. Estas propiedades del algoritmo de recorrida y de construcción son muy importantes para implementarlo en una GPU, y contrarrestan algunas propiedades negativas que se presentan en el trabajo, como por ejemplo que requiere guardar mayor cantidad de datos para representar el estado del recorrido, en comparación

con las estructuras *kd-tree* y BVH. Los autores concluyen que el algoritmo de recorrida de las cajas se puede paralelizar muy fácilmente en una GPU pero que la implementación tendría un bajo rendimiento para escenas complejas y un buen rendimiento para escenas simples y de mediano porte, en comparación con el obtenido al usar las estructuras *kd-tree* o BVH.

Por otro lado, Thrane y Ole [28] implementaron la estructura de aceleración *kd-tree* sobre GPU. Usaron las técnicas (*restart* y *backtrack*) y concluyeron que para la mayoría de las escenas, esta estructura mejora en rendimiento a la sub-división espacial uniforme. También se recalca que las dos técnicas para realizar la recorrida a través de los *voxeles* sufren de alta complejidad en sus algoritmos (en el contexto de una GPU) y esto no es deseable ya que aumenta el tiempo de ejecución. Aunque también se menciona que el tiempo de ejecución que se agrega por la complejidad puede verse compensado, en cierto grado, por la habilidad de la estructura para adaptarse a los cambios de densidad de objetos en la escena. Con respecto al algoritmo para la construcción de la estructura, se concluye que dada su condición de recursivo su implementación a nivel de GPU es más compleja, en comparación con la sub-división uniforme.

En el trabajo de Lauterbach et al. [5] luego de haber usado la estructura *kd-tree* para diversos casos de prueba se llegó a la conclusión de que con esta se obtienen mejores resultados de los que se obtienen utilizando la estructura BVH, para escenas estáticas (las escenas estáticas se caracterizan por componerse de objetos fijos, que no cambian de posición ni de forma en el tiempo). Esta ganancia en rendimiento tiene como costo asociado un mayor consumo de memoria y una mayor complejidad para implementar y optimizar la construcción de la estructura.

Con respecto a la estructura BVH, Thrane y Ole [28] implementaron sobre una GPU ambas estrategias de construcción, el enfoque *top-down* y el *bottom-up*, para comparar resultados. Usaron siempre volúmenes envolventes alineados a los ejes, en ambas construcciones. La estrategia de recorrida a lo largo de los nodos hijos fue siempre de izquierda a derecha. Esto en algunos casos no resultó muy eficiente ya que si el rayo atraviesa todos los volúmenes envolventes y el volumen donde se da la intersección más cercana es el último de una lista de hermanos, se debe revisar todos los demás volúmenes antes de llegar a un resultado. En un caso de este tipo prácticamente se está utilizando fuerza bruta para encontrar la intersección, situación que se busca evitar con la introducción de estructuras de aceleración. Una alternativa es recorrer la lista de hermanos según la dirección del rayo pero los autores optaron por no

mejorar este aspecto para mantener la simplicidad en el algoritmo (ya que debe ser implementado en una GPU).

La gran ventaja de la estructura BVH es la simplicidad del algoritmo de recorrida y la gran desventaja es el orden fijo en que se recorren los nodos hermanos [28]. Además Thrane y Ole [28] concluyeron que para escenas complejas esta estructura es la que tiene mejor rendimiento sobre la GPU, comparando con la sub-división espacial uniforme y con la *kd-tree*. Como un agregado se destaca que la implementación de la construcción es simple, aunque para llevarla a nivel de GPU hay que resolver el problema de la recursión si se elige el enfoque *top-down* o el problema de encontrar un buen orden para la lista de objetos de la escena si se elige el enfoque *bottom-up*. Además, la construcción en cualquiera de los enfoques es computacionalmente más costosa que la construcción de la grilla uniforme.

Lauterbach et al. [5] luego de haber usado la estructura BVH para diversos casos de prueba concluyeron que con esta estructura se obtienen mejores resultados de los que se obtienen utilizando la *kd-tree*, para escenas dinámicas (las escenas dinámicas se caracterizan por componerse de objetos que a medida que el tiempo avanza cambian de posición, forma, etc.). Además, los autores señalan que a la jerarquía de volúmenes envolventes es más fácil agregarle la optimización basada en paquetes de rayos que a la *kd-Tree*.

En la Tabla 3.1 se resumen las principales conclusiones sobre las estructuras de aceleración espacial.

	SEU	Kd-Tree	BVH
Complejidad del algoritmo de construcción	Baja	Alta	Media
Aceleración lograda para escenas simples	Alta	Baja	Media
Aceleración lograda para escenas complejas	Baja	Media	Alta
Adaptación frente a escenas no uniformes	Baja	Media	Alta
Consumo de memoria	Medio	Alto	Bajo
Complejidad del algoritmo de atravesado	Media	Alta	Baja
Adaptación frente a escenas estáticas	Mala	Buena	Mala
Adaptación frente a escenas dinámicas	Mala	Mala	Buena

Tabla 3.1: Comparación de las estructuras de aceleración.

3.5. *Ray tracing* interactivo

Se llama *ray tracing* interactivo (RI) a una implementación del algoritmo que permita modificar parámetros que afecten a la imagen generada y que el ojo humano pueda percibir las consecuencias de las modificaciones como una animación, para lograr esto se necesitan por lo menos 24 FPS (*frames* por segundo). A diferencia del *ray tracing* en tiempo real este tipo de implementaciones no serviría para hacer video-juegos o generadores de video en tiempo real, dado que la cantidad de imágenes por segundo no sería suficiente.

3.5.1. Estado del Arte

El artículo de Wald et al. [35] plantea cual es el estado del arte en la generación de imágenes para programas en los que uno de los objetivos es la capacidad de interactuar con los usuarios. Según lo relevado en dicho artículo, el algoritmo que generalmente es utilizado para la generación de imágenes es el de Rasterización, pero debido al constante aumento del poder de cómputo del *hardware* el algoritmo de *ray tracing* surge como una alternativa válida.

Para lograr que el algoritmo de *ray tracing* sea interactivo se le deben aplicar estrategias de simplificación y aceleración. En el trabajo de Wald et al. se presentan diversas técnicas que permiten utilizar *ray tracing* para lograr buenos efectos visuales en tiempos interactivos, y se clasifican según las categorías que se presentan a continuación:

- basadas en rasterización. Este tipo de técnicas combinan la alta velocidad de generación de imagen del *hardware* de rasterización de la actualidad con la calidad superior de las imágenes generadas mediante *ray tracing*. Por ejemplo, existe una técnica que utiliza como base la rasterización y únicamente usa *ray tracing* para el cálculo de algunos efectos de iluminación.

- basadas en imagen. Este tipo de técnicas sacan provecho de la coherencia temporal entre las imágenes generadas y se basan en extraer información de la imagen generada en el paso anterior para generar solo algunos píxeles de la imagen siguiente.

- basadas en *ray tracing*. Estas técnicas apuntan a reducir el costo por píxel del algoritmo de *ray tracing* mediante optimizaciones al proceso de trazado de rayos.

- basadas en acelerar *ray tracing*. Las técnicas pertenecientes a esta categoría intentan bajar los costos computacionales asociados al trazado de rayos, por ejemplo utilizando arquitecturas de memoria de las CPU actuales que puedan ser empleadas de manera eficiente o tratando de explotar la capacidad del algoritmo de ser ejecutado en paralelo.

3.5.2. Interactividad sobre multiprocesadores de memoria compartida

El artículo de Wald et al. describe la exploración dentro de las técnicas que buscan acelerar los algoritmos de *ray tracing* de modo de intentar que se puedan ejecutar de manera interactiva. En dicha investigación realizan la implementación de un algoritmo de *ray tracing* que ejecuta en una arquitectura multiprocesador. En la solución desarrollada se logró la interactividad, en gran medida porque la plataforma de ejecución consistió en una plataforma de gran poder de cómputo (SGI Origin 2000). Por otra parte se menciona que aún fuera de estas condiciones (usando PCs de bajo costo) es posible lograr que *ray tracing* sea interactivo por tres características del algoritmo:

- escala bien en cientos de procesadores.
- para escenas estáticas el tiempo de render de los frames (generación de cada cuadro de una animación) el orden del algoritmo es sub-lineal en la cantidad de objetos básicos en la escena.
- permite agregar una gran variedad de objetos básicos y efectos de sombreado programados por el usuario.

Para el trabajo de Wald et al. se utilizó como base el algoritmo clásico de Whitted [36] modificándolo para obtener mejoras visuales y de performance. Las mejoras que afectan directamente a la velocidad del algoritmo se pueden dividir en dos grandes ramas: aceleración o eliminación de cálculos de verificación de intersección entre rayos y objetos, y paralelización. Para optimizar la cantidad de cálculos de la intersección se genera una división espacial de la escena (SEU) y se combina con volúmenes envolventes para los objetos de la misma.

Para paralelizar el algoritmo se emplea un sistema de memoria compartida, y el algoritmo utiliza una estrategia maestro-esclavo para la generación de la imagen. El proceso maestro inicializa la escena a renderizar y genera

rayos que serán lanzados hacia la escena, los cuales se ingresan en una cola. Posteriormente a la etapa de inicialización los procesos esclavos obtendrán a demanda desde la cola los rayos para procesar. Esta estrategia tiene un gran problema en el tiempo necesario para la sincronización entre procesos, por esto los rayos se agrupan de a varios para obtener una mejor performance. Las limitaciones que se pudieron constatar para el algoritmo de *ray tracing* son el balanceo de carga y la sincronización entre los procesos.

En la versión final del algoritmo se logró la interactividad con una cantidad relativamente pequeña de procesadores, 8 procesadores *dual PIII* a *800 MHz* conectados mediante *Gigabit Ethernet*. Esta implementación de *ray tracing* mostró que es un algoritmo muy bueno para presentar efectos de luz dinámicos pero no así para procesar escenas en las cuales los objetos cambian dinámicamente.

3.6. Conclusiones sobre aceleración

La aceleración del algoritmo de *ray tracing* de Whitted puede lograrse empleando diversos métodos. Al estudiar las diferentes técnicas en el relevamiento realizado se identifican las más importantes. La estrategia que se identifica para acelerar *ray tracing* es explotar su gran capacidad de paralelización. En la implementación realizada en este proyecto se paralelizó el algoritmo de Whitted teniendo en cuenta los problemas más comunes de esta estrategia, la sincronización entre hilos y el balanceo de trabajo de procesamiento entre ellos.

Otro problema que se identifica como importante es el de disminuir el tiempo de procesamiento que insume la intersección de los rayos con la escena. Para atacar dicho problema en este proyecto se emplearon estrategias en dos sentidos, disminuir el tiempo que toma cada intersección rayo-primitiva y disminuir la cantidad de intersecciones que se prueban por cada rayo que incide la escena. Para esto se escogió un algoritmo eficiente de intersección rayo-primitiva y se dividió la escena uniformemente según una sub-división espacial uniforme, respectivamente.

Capítulo 4

Propuesta

4.1. Introducción

El principal objetivo de este proyecto es acelerar el algoritmo de *ray tracing* utilizando GPUs, buscando renderizar imágenes en tiempo menor al que insume el algoritmo tradicional en CPU y sin perder calidad en las imágenes generadas.

La primera propuesta de aceleración se basa en paralelizar el trazado de rayos primarios utilizando las capacidades de ejecución multihilo de las GPUs. Recordar que para obtener el color de un píxel con *ray tracing* se debe lanzar un rayo primario (como mínimo), entonces para renderizar una imagen cuya resolución sea de 1024 por 768 píxeles se deben trazar al menos 786432 rayos primarios. En una implementación tradicional del algoritmo se procesa un píxel a la vez, es decir en forma secuencial, mientras que en la propuesta se procesan de forma paralela.

4.2. Descripción general

El algoritmo de *ray tracing* de Whitted requiere como entrada una escena, la cual será renderizada y posee varias etapas. En el Algoritmo 4 se presenta el pseudo-código para generar una imagen describiendo las distintas etapas. En una primera etapa se cargan los datos de la escena y se inicializan las estructuras de datos. En el primer paso de la etapa de preprocesamiento se carga la escena desde archivo. Luego, se construye la grilla de la sub-división espacial uniforme a partir de las primitivas cargadas en el paso anterior. En

el último paso se construyen todos los rayos primarios a partir de los datos de la escena y de la resolución de la imagen a generar. La construcción de rayos se realiza en la GPU, paralelizando el cálculo de cada uno de ellos.

Algoritmo 4 Pseudo-código del proceso para la generación de imagen.

```
e : Escena;
/* Etapa de preprocesamiento */
e = cargarEscenaDesdeArchivoOBJ(rutaArchivo);
e = construirGrillaUnifome(e);
rsPri : Matrix [pxAncho, pxAlto] of Rayo;
rsPri = calcularRayosPrimarios(pxAncho, pxAlto, e);
/* Etapa de renderizado */
img : Matrix [pxAncho, pxAlto] of Color;
para todo idxAncho en [1 ... pxAncho] hacer
    para todo idxAlto en [1 ... pxAlto] hacer
        c : Color;
        c = hallarColorRT(e, rsPri[idxAncho, idxAlto]);
        img[idxAncho, idxAlto] = c;
    fin para
fin para
/* Etapa de presentación de resultados */
SDLMostarImagen(img);
```

En primer término se definió una arquitectura genérica para la solución, en la Figura 4.1 se muestra un diagrama de la arquitectura de la solución implementada. En lo que resta de la sección se detallan las etapas implementadas, se profundiza en las distintas versiones que se desarrollaron, además, se explican las etapas resueltas con otras herramientas.

Los componentes involucrados en la etapa de preprocesamiento son: "*OBJ Loader*", "Cargador Escena" y "Grilla Uniforme". El componente "*OBJ Loader*" es el encargado de leer el archivo que especifica la escena y los materiales utilizados en la misma. El componente "Cargador Escena" que utiliza el "*OBJ Loader*" fue implementado en el marco de este proyecto. Este componente tiene su propia estructura de datos para el almacenamiento de la escena, debido a esto se encarga de inicializar su estructura a partir de los datos leídos. Además se encarga de la construcción de la sub-división espacial uniforme. La división espacial se genera utilizando el componente "Grilla Uniforme", esta parte del sistema construye la grilla a partir de la lista de

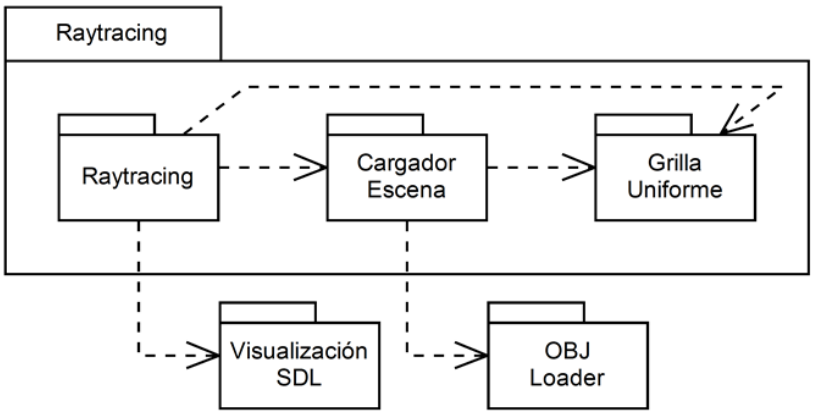


Figura 4.1: Arquitectura del *ray tracing* implementado sobre GPU.

objetos que contiene el encargado de inicializar la estructura de datos de la escena. Por último, el componente "Raytracing" es el encargado de calcular los rayos primarios utilizado la GPU.

En la segunda etapa del proceso de generación de imagen se calcula el color de cada píxel mediante el algoritmo de *ray tracing*. Esta etapa se realiza enteramente a nivel de GPU. Hablando en términos de la arquitectura, el componente "Raytracing" es el encargado de renderizar la imagen. Dentro de este se implementaron los algoritmos que componen el núcleo del *ray tracing*, como por ejemplo el algoritmo de recorrida de un rayo dentro de la sub-división espacial uniforme, el algoritmo de intersección rayo-triángulo, etc.

En la última etapa se presenta en pantalla la imagen que surge como resultado de la etapa anterior. Desde el punto de vista de la arquitectura, el componente "Raytracing" es el encargado de mostrar la imagen generada utilizando la librería externa SDL (Simple Directmedia Layer) [15].

A continuación se presentan los aspectos más relevantes del diseño de la solución implementada.

4.2.1. Preprocesamiento

En este trabajo la escena es cargada desde un archivo de texto usando exclusivamente la CPU. El formato de este archivo esta basado en el Wavefront OBJ version 3.0 [10].

Los datos de la escena son cargados en memoria mediante un cargador

implementado por Micah Taylor [11] llamado *OBJ Loader*. El *OBJ Loader* es capaz de interpretar el formato OBJ completamente, pero también es capaz de leer otros datos que no son parte del estándar de Wavefront, pero que son muy útiles para el algoritmo de *ray tracing*. Las características extras soportadas por el *OBJ Loader* y que extienden el formato OBJ son:

- soporte de nuevas primitivas, por ejemplo planos.
- soporte de elementos extra, como por ejemplo luces puntuales y cámaras.
- soporte de propiedades de material avanzadas, como por ejemplo el coeficiente de reflexión y el de transmisión de un material.

El *OBJ Loader* tiene su propia estructura de datos para almacenar la escena en memoria, pero en la propuesta implementada en este proyecto son utilizadas temporalmente. Después de cargar la escena con el *OBJ Loader*, esta se almacena en una estructura definida especialmente. De esta manera se desacopla el resto del algoritmo de la etapa de carga, lo cual permite cambiar fácilmente la herramienta de carga de la escena.

En la etapa de preprocesamiento se necesitan diferentes parámetros. La resolución de la grilla de aceleración, así como otros parámetros del algoritmo de *ray tracing* se definen mediante el uso de un archivo de configuración. Se tomó la decisión de utilizar un archivo de configuración ya que para las pruebas de rendimiento es interesante cambiar los valores de los parámetros más importantes de cada algoritmo y no tener que re-compilar el código fuente con cada cambio.

En el archivo además del tamaño de la sub-división espacial uniforme, se debe indicar el tamaño de la partición en cada eje de coordenadas, la resolución de la imagen (en píxeles) que genera el algoritmo de *ray tracing*, valores para el cero y para el infinito que usa el algoritmo (el ajuste del cero permite solucionar problemas de errores numéricos mientras que el segundo se usa en algoritmos que necesitan definir una distancia "infinita"). Otro parámetro necesario para el algoritmo de *ray tracing* es la profundidad máxima considerada por el algoritmo para cada árbol de rayos. Por último, se debe especificar la dimensión de los bloques de hilos de ejecución. La Figura 4.2 se muestra una posible configuración del algoritmo.

```
TAMANIO_GRILLA.X=64 /*tamaño de la partición espacial*/
TAMANIO_GRILLA.Y=50
TAMANIO_GRILLA.Z=50
RESOLUCION.X=640 /*resolución de la imagen*/
RESOLUCION.Y=480
INFINITO= 34028234660000000
ZERO=0.000001 /*valor tomado como cero*/
PROFUNDIDAD_RECURSION=3 /*valor máximo rebotes rayo*/
THREADS.X=8 /*hilos por bloque*/
THREADS.Y=16
ESCENA=../escenas/afrodita.obj
```

Figura 4.2: Ejemplo de contenido para el archivo de configuración.

4.2.2. Intersección

En este proyecto se decidió acelerar el algoritmo de *ray tracing* por medio de disminuir el tiempo de ejecución que toman las intersecciones rayo-objeto, pues estas consumen más del 90 % del tiempo de generación de imagen [28]. En este sentido se atacaron los puntos críticos del proceso de intersección rayo-escena, lo que implicó implementar un algoritmo eficiente de intersección rayo-primitiva y reducir la cantidad de intersecciones rayo-primitiva que se prueban por cada rayo primario.

El algoritmo de trazado de rayos implementado únicamente posee un tipo de primitiva, el triángulo, por lo tanto solo se debió escoger un método para la intersección rayo-triángulo. Se escogió únicamente el triángulo como primitiva de construcción de escenas porque es una primitiva sencilla, y a su vez permite construir cualquier objeto tridimensional usándola como base. Asimismo, al ser una primitiva básica su algoritmo de intersección con un rayo es simple, lo cual implica que se requieran pocas operaciones aritméticas para probar su intersección con un rayo. Además, es una primitiva ampliamente soportada por la mayoría de herramientas de modelado tridimensional, como *3D Studio Max* [7], *Maya* [8], *Blender* [9], etc.

El método utilizado para verificar la intersección entre un rayo y un triángulo es el de coordenadas baricéntricas. Dicho método verifica que el rayo interseque con el plano que contiene al triángulo y luego mediante un

cambio de coordenadas verifica que la intersección esté dentro de los límites del mismo [1]. Este método no es el más eficiente que existe pero su consumo de memoria es mínimo, lo cual es importante si se quiere implementar el algoritmo en una GPU.

4.2.3. Aceleración espacial

A pesar de contar con un algoritmo de intersección eficiente es importante implementar una estrategia para no probar para cada rayo primario la intersección con todos los objetos de la escena. En la etapa de relevamiento se evaluaron tres estructuras de aceleración espacial: la sub-división espacial uniforme, la sub-división espacial adaptativa (utilizando *kd-tree*) y la jerarquía de volúmenes envolventes (BVH), que se describen en el Capítulo 3.

La estrategia de aceleración espacial que se adoptó en este proyecto fue la sub-división espacial uniforme. El argumento de mayor peso para la elección de esta estructura fue la simplicidad de construcción y recorrida de la misma, lo cual resulta imprescindible para paralelizar los algoritmos usando una GPU.

Construcción de la grilla

Para la construcción de la grilla se evaluaron dos métodos. Ambos métodos se implementaron exclusivamente en la CPU, aunque podrían ser acelerados ejecutándolos en la GPU.

El primer algoritmo implementado fue la construcción por fuerza bruta. Como se muestra en el Algoritmo 5, este método de construcción recorre todos las cajas de la grilla y para cada una de ellas recorre todos los objetos de la escena para probar si hay intersección caja-objeto. Este método es ineficiente porque por lo general se recorren muchas cajas innecesarias (que no tienen intersección con el objeto) por cada objeto de la escena.

En el Algoritmo 6 se describe el segundo algoritmo de construcción. Este caso es un algoritmo optimizado donde se recorren todos los objetos de la escena y para cada uno ellos se calcula (en coordenadas de la grilla) un volumen alineado a los ejes que lo envuelve. A partir del volumen se obtienen cajas de la grilla candidatas a solaparse con el objeto. Para cada caja candidata se prueba el solapamiento caja-objeto y si da positiva se agrega el objeto a la caja de la grilla. La optimización introducida con respecto al método mostra-

do en el Algoritmo 5 permite desechar una cantidad importante de cajas que no tienen intersección con el objeto. Excepto en el peor caso (cuando todos los objetos de la escena se solapan con todos las cajas de la grilla), el método optimizado es más eficiente que el original, ya que disminuye la cantidad de pruebas de intersección caja-objeto.

Algoritmo 5 Pseudo-código del algoritmo de fuerza bruta de construcción de la grilla uniforme.

```
para todo caja en grilla hacer
    para todo obj en listaObjetos hacer
        //En coordenadas de la grilla
        bbObj = calcularBoundingBoxObjeto(obj);
        si (caja ∩ bbObj) ≠ ∅ entonces
            agregarObjetoEnCaja(caja, obj);
        fin si
    fin para
fin para
```

Algoritmo 6 Pseudo-código del algoritmo eficiente de construcción de la grilla uniforme.

```
para todo obj en listaObjetos hacer
    //En coordenadas de grilla
    bbObj = calcularBoundingBoxObjeto(obj);
    para todo caja ⊂ bbObj hacer
        //En coordenadas de mundo
        cajaMundo = transformarCoorMundo(caja);
        si boundingBoxOverlapObject(cajaMundo, obj) entonces
            agregarObjetoEnCaja(caja, obj);
        fin si
    fin para
fin para
```

Recorrido de la grilla

El algoritmo para recorrer la grilla está basado en el algoritmo de renderizado de una línea en pantalla, que se implementa en las GPUs. El recorrido

de un rayo a través de la grilla de sub-división espacial genera una lista de cajas, estas cajas son las que el rayo atraviesa sucesivamente. El algoritmo se basa en incrementar de manera inteligente un punto a lo largo del rayo, con cada incremento se avanza a la siguiente caja realizando unas pocas sumas y evaluaciones de condición.

La implementación se realiza en 2 pasos, primero se computa el cálculo de la caja inicial y de los incrementos en cada una de las 3 direcciones para un rayo específico y por otra parte la manera en que se computa cada cambio de caja. La caja inicial se calcula de manera sencilla intersecando el rayo con el volumen que acota a la escena que se utiliza para generar la grilla, con el punto de intersección se calcula cual es la caja de la grilla al que pertenece el punto. Para obtener el incremento se calcula la derivada en cada una de las 3 componentes, con esa derivada y el tamaño de las cajas de la grilla en x, y, z se obtiene la distancia que debe recorrer el rayo para poder alcanzar la siguiente, esta se calcula para cada una de las 3 componentes y se almacena en una variable, así como también se hace con las derivadas.

En el Algoritmo 7 se muestra la forma en que se calcula la siguiente caja. Este cálculo está basado en la distancia que tiene que recorrer el rayo desde su origen hasta el siguiente punto de intersección. Se tienen tres distancias que debe recorrer el rayo para pasar a la siguiente caja en el eje X, para pasar a la siguiente caja en el eje Y y para pasar a la siguiente caja en el eje Z. La mínima de estas distancias determina cual de los tres incrementos es el que hay que realizar, la caja que esté más cerca es la siguiente, por lo tanto hay que incrementar 1 en la dirección determinada por dicho mínimo.

Algoritmo 7 Pseudo-código del algoritmo para avanzar en las cajas de la grilla.

```
si (Minimo(DX, DY, DZ) == DX) entonces
    DX = DX + incrementoX;
    CajaActualX = CajaActualX + signo(incrementoX) * 1;
fin si
si (Minimo(DX, DY, DZ) == DY) entonces
    DY = DY + incrementoY;
    CajaActualY = CajaActualY + signo(incrementoY) * 1;
fin si
si (Minimo(DX, DY, DZ) == DZ) entonces
    DZ = DZ + incrementoZ;
    CajaActualZ = CajaActualZ + signo(incrementoZ) * 1;
fin si
```

4.2.4. Núcleo de *ray tracing* sobre GPUs

Considerando la arquitectura del sistema, el núcleo principal del *ray tracing* para GPU se encuentra implementado dentro del componente "Ray-tracing". Para lograr que el algoritmo tenga un buen rendimiento es necesario utilizar adecuadamente la jerarquía de memoria de la GPU. Los datos más utilizados por el algoritmo, como por ejemplo la lista de objetos de la escena o las cajas de la sub-división espacial, deben estar almacenados en un tipo de memoria que tenga tiempo de lectura bajo, y pudiendo ser de solo lectura, ya que esta información es frecuentemente accedida y nunca debe ser actualizada. Es por ello que la lista de triángulos y sus normales, las luces, las cajas de la grilla de sub-división espacial y la lista de materiales se copian a la memoria de textura de la GPU. Otra información como los datos de la cámara, la cantidad de luces, la dimensión de la grilla se copian a la memoria constante de la GPU, que también es de solo lectura. La lectura en los dos tipos de memoria mencionados es mucho más rápida que una lectura en memoria global.

El uso de la jerarquía de memoria que provee CUDA afecta directamente a la estructura de datos que almacena la escena. Por ejemplo, la lista de triángulos de la escena ocupa una cantidad grande de memoria y al cargarla desde archivo insume un tiempo de ejecución considerable. Toma prácticamente el mismo tiempo transformar toda la lista de triángulos desde el formato en

que esta almacenada hacia el formato que requiere CUDA para cargarla en memoria de textura. Es por esto que se decidió almacenarla directamente en el formato que debe tener para copiarla a memoria de la GPU (ver Apéndice A).

En la memoria de textura solo se pueden copiar vectores de dos o cuatro elementos, debido a esto cada vértice de triángulo se almacena como un vector de cuatro componentes, desperdiciando una componente (4 bytes) por cada vértice. En algunos casos se aprovecha la memoria de la cuarta componente, por ejemplo el identificador de material se guarda en la cuarta componente del primer vértice de cada triángulo. De todas formas para usar la jerarquía de memoria y mejorar el tiempo de ejecución del algoritmo se pierde la generalidad en las estructuras de datos y la legibilidad del código fuente.

Una vez copiados todos los datos de entrada del algoritmo de *ray tracing* a la GPU, se procede a la invocación del *kernel* encargado de calcular los rayos primarios. Como se muestra en la Figura 4.3, los datos necesarios para calcular los rayos primarios son: el plano de vista, la cámara y la resolución de la imagen a renderizar. Toda esta información se encuentra en memoria constante y puede ser accedida en cualquier instante del tiempo de vida de la aplicación CUDA. Es importante resaltar que la invocación al *kernel* que calcula rayos se hace desde la CPU y cuando este finaliza su procesamiento retorna nuevamente a la CPU.

Los rayos primarios son datos de entrada para el *kernel* encargado de calcular el color de cada píxel. Inmediatamente después que se tienen calculados los rayos primarios, se invoca el *kernel* (desde la CPU) que genera la imagen. Dentro de este se encuentra implementado el corazón del algoritmo de *ray tracing*. Los datos de entrada necesarios para calcular el color de cada píxel se muestran en la Figura 4.3 y son leídos desde memoria de textura, cuyo tiempo de vida es igual al de la aplicación CUDA.

El *kernel* principal es invocado según una división en parches (conjunto de píxeles) de la imagen a generar, tal como se detalla en la Sección 4.2.5. De esta forma, cada parche es asociado con un bloque de ejecución de CUDA y la cantidad de hilos que ejecutan el procedimiento de cálculo de color es igual a la cantidad de píxeles. Cada uno de los rayos lanzados recorrerá la estructura de aceleración espacial, siguiendo los reflejos y refracciones. Al finalizar esta ejecución es que se realiza la copia de los datos generados en la GPU nuevamente a la CPU para ser mostrados en pantalla.

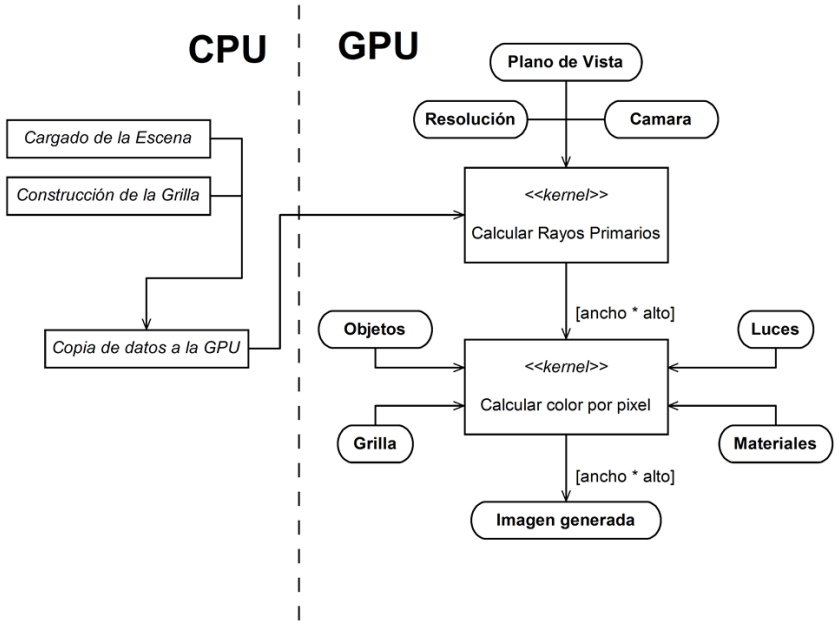
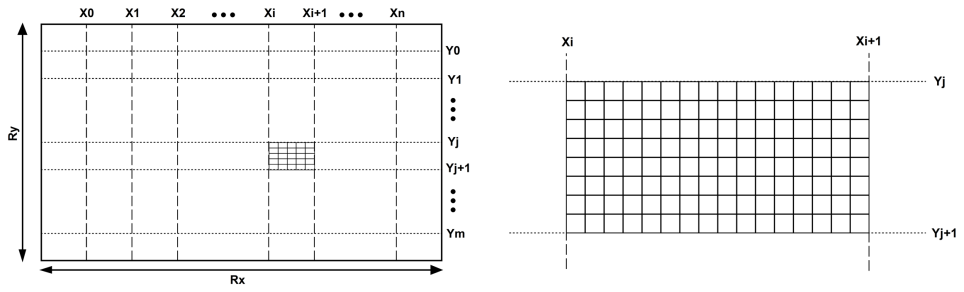


Figura 4.3: Estructura del algoritmo de *ray tracing* implementado en CUDA.

4.2.5. Paralelización del algoritmo

Para resolver el problema en términos del modelo de programación usado por CUDA se hizo una descomposición en sub-problemas, dividiendo la imagen a renderizar. La imagen se divide usando una grilla uniforme de dos dimensiones, donde cada celda de la grilla tiene la misma cantidad de píxeles. En la Figura 4.4(a) se muestra una posible división de la imagen. En este ejemplo la imagen se divide en $n \times m$ celdas, quedando así cada celda con $\frac{R_x}{n} \times \frac{R_y}{m}$ píxeles. En la Figura 4.4(b) se muestra en detalle la celda definida por los intervalos $[X_i, X_{i+1}]$ y $[Y_j, Y_{j+1}]$, la cual contiene una pequeña parte de los píxeles de la imagen. Cada celda de la grilla se corresponde directamente con un bloque de hilos de ejecución de CUDA, es decir cada división de la imagen es procesada por un bloque de CUDA diferente. Además, como cada píxel de la imagen es procesado por un hilo de ejecución diferente, la cantidad de hilos por bloque es igual a la cantidad de píxeles que contiene cada división. Es por esta razón que la partición queda totalmente establecida cuando se fija la cantidad de hilos de ejecución por bloque (en el archivo de

configuración), además de la resolución de la imagen a renderizar. Si se tiene una resolución de imagen de 640×480 píxeles y el tamaño de los bloques es de 16×8 hilos la imagen quedará dividida en una grilla de 40×60 celdas.



(a) (b)

Figura 4.4: Sub-division de la imagen para lograr paralelismo.

Mediante esta forma de descomponer el problema el *ray tracing* para CUDA puede escalar fácilmente en el número de procesadores de la GPU sin necesidad de recompilar su código fuente. Además mediante el archivo de configuración es posible cambiar la partición de la imagen a renderizar de manera de optimizarla para cualquier GPU.

La cantidad máxima de hilos de ejecución por bloque es 512, por consiguiente 512 es la cantidad máxima de píxeles que pueden ser procesados por un mismo bloque, límite impuesto por CUDA (versión 2.3). También hay que tener en cuenta que la cantidad de hilos por bloque puede verse limitada por la cantidad de registros que consuma cada hilo, ya que por ejemplo la cantidad máxima de registros por bloque es 8192 (CUDA versión 2.3).

4.2.6. Eliminación de la recursión

El algoritmo de *Ray tracing* es un algoritmo inherentemente recursivo, esto es una limitación a la hora de hacer que se ejecute en la GPU dado que la ejecución debe ser secuencial, ya que CUDA no tiene soporte para recursión. Este algoritmo puede analizarse como una recorrida en un árbol binario (el árbol de rayos originado por un rayo primario y sus sucesivos rebotes debido a los fenómenos de reflexión y refracción).

Para resolver este problema se evaluaron dos opciones, la primera consiste en implementar una pila la cual sirva de apoyo para la recorrida del árbol

binario. La segunda consiste en simplificar el árbol de manera que degenere en una lista, esta forma de simplificar el árbol implica una pre-condición sobre la escena de entrada, que en la escena no existan objetos que reflejen y transmitan la luz al mismo tiempo.

La opción elegida fue la segunda ya que simplifica el algoritmo a implementar en la GPU a cambio de incluir una limitación aceptable a nivel de las escenas de entrada. Otra razón importante para optar por simplificar el árbol de rayos, es que en caso de implementar la primer opción cada hilo de ejecución debe contar con una pila y la cantidad de memoria local de cada hilo de ejecución de CUDA es muy limitada.

Para implementar el algoritmo simplificado e iterativo se modificó el original, de forma que cada rayo sea el encargado de llevar el estado de su trayectoria de manera iterativa. En caso de que el rayo interseque con algún objeto cuyo material posea reflexión o refracción, se debe modificar la dirección del mismo según las propiedades del material, para que en la siguiente iteración se trace en la trayectoria correcta. Mientras el rayo siga rebotando en los objetos de la escena y no se supere la cantidad máxima de rebotes se debe continuar con este proceso iterativo.

4.2.7. Estructura de datos

Durante el proyecto se desarrollaron diferentes estructuras de datos para trabajar en GPU, las cuales se muestran en detalle en el Apéndice A. A continuación se presentan las principales características de las estructuras de datos utilizadas.

La primera versión de la estructura de datos usada para trabajar en GPU busca almacenar los datos de la escena de forma "prolija". Entre los principios de su concepción se encuentran: sencillez de acceso a los datos al momento de implementar los algoritmos, simple introducción de nuevos tipos de primitivas y uso eficiente de memoria del sistema.

Entre los campos más importantes de la estructura utilizada para almacenar la escena se encuentra la lista de objetos de la misma, que se guardan en un arreglo (*objetos*) con tope (*cant_objetos*). Otro campo importante donde se guarda la información referente a la división espacial de la escena, es el campo *grilla*, que es a su vez otra estructura de datos llamada *UniformGrid*. Esta estructura tiene varios campos:

- *dimension*: almacena la dimensión de la grilla que divide el espacio de

la escena.

- *bbEscena*: almacena una caja alineada a los ejes de coordenadas que contiene a toda la escena.
- *voxels*: array de punteros a entradas de *listasGrid*. Cada entrada de *voxels* se corresponde con un *voxel* de la escena, donde se almacena un puntero a la lista de objetos que contiene el *voxel*.
- *listasGrid*: array de listas de punteros a objetos.

Se puede ver un ejemplo esquemático de la estructura *UniformGrid* en la Figura 4.5. En este ejemplo se considera una escena con seis objetos, los cuales están completamente contenidos es una caja alineada a los ejes definida por los puntos $min = (-10, -10, -10)$ y $max = (10, 10, 10)$. Al aplicarle una sub-división espacial uniforme con dimensiones $2 \times 2 \times 3$ a la escena se generan 12 cajas, tal como se muestra en el vector "voxels". Cada objeto de la escena se encuentra contenido en una o más divisiones, por lo tanto por cada división se tiene una lista de objetos contenidos. El vector "listasGrid" contiene las listas de todas las cajas de la sub-división y mediante las entradas del vector "voxels" se accede a cada una de ellas. El caso especial en que la lista de objetos de una caja es vacía se marca con un valor especial en el vector "voxels".

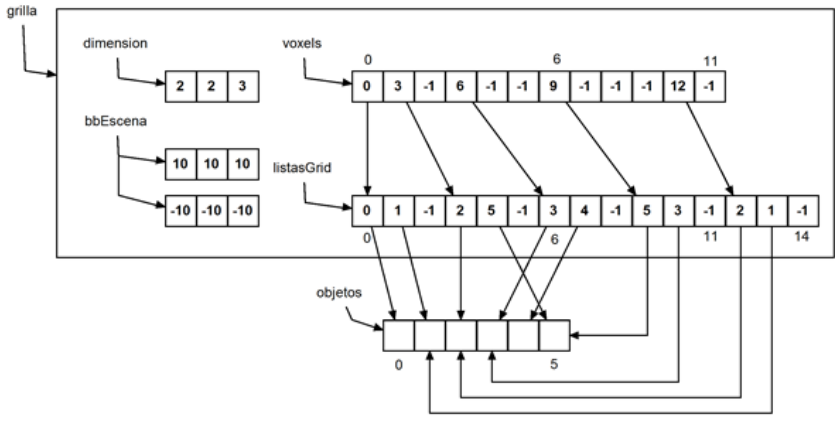


Figura 4.5: Ejemplo de estructura *UniformGrid*.

La segunda versión de la estructura de datos usada para trabajar en GPU es una evolución de la primera. Los cambios que generaron la evolución

estuvieron determinados por la forma de utilizar la jerarquía de memoria de la GPU. La estructura tuvo que ser reordenada de manera de minimizar las operaciones de memoria al momento de cargar los datos de la escena en la GPU. Lo que más afecta la eficiencia es reordenar los datos antes de cargarlos en memoria de textura de la GPU. Es importante que el copiado de memoria de la CPU hacia la GPU sea lo más eficiente posible ya que de esta manera se reduce el tiempo de ejecución consumido por la etapa de preprocesamiento.

Los campos más importantes de la segunda versión de la estructura son los mismos que en la primera versión, ya que sólo se ordenaron de manera diferente. El único campo importante que se modificó fue la lista de objetos, se paso de almacenar una lista agrupada por objetos (lista de pares vértices-normales) a almacenar varias listas, una por cada propiedad de los objetos (una lista de vértices y otra de normales).

Otra diferencia importante entre la primera y la segunda versión es que en la segunda algunos tipos de datos se modificaron para adaptarlos a los tipos de datos soportados por la jerarquía de memoria de CUDA. Por ejemplo, en la primera versión la lista de luces es una lista de *float3*, donde los primeros dos definen la posición y el color de la primer luz respectivamente, los segundos dos la posición y el color de la segunda y así sucesivamente. Como la lista de luces se carga en memoria de textura y la memoria de textura no permite cargar vectores de tres componentes, en la segunda versión se optó por transformar la colección de luces en una lista de *float4*, donde cada propiedad es definida utilizando únicamente las primeras tres componentes.

4.3. Versiones implementadas

El desarrollo del *ray tracing* para CUDA fue iterativo-incremental, desarrollando tres grandes versiones. En forma general la primera versión implementa el *ray tracing* de Whitted completo, la segunda versión está marcada por la introducción de mejoras que tienen que ver con la optimización en la explotación de la jerarquía de memoria de la GPU. Por último, la tercera está señalada por la mejora del algoritmo de cálculo de la intersección rayo-triángulo.

4.3.1. Versión 1: RT-GPU

Las principales caracteristicas de esta versión son:

- Implementa el algoritmo de Whitted con reflexión y refracción de forma iterativa.
- Utiliza la estructura de aceleración espacial *Uniform Grid*.
- La generación de rayos primarios se hace en un *kernel* de CUDA.
- Para la recorrida de la grilla, el trazado de rayos de sombra y secundarios se utiliza un sólo *kernel* de CUDA.
- Las operaciones sobre vectores se hacen usando las operaciones propias de CUDA.

Las principales restricciones de la versión RT-GPU son:

- las escenas deben tener como máximo una luz puntual.
- las escenas no pueden tener objetos reflexivos y transparentes a la vez.
- la sombra proyectada por un objeto transparente no tiene en cuenta el color del objeto.
- no se explota al máximo la estructura de memoria de la GPU.
- sufre el problema que genera el sistema operativo al ejecutar un *kernel* por más de 5 segundos. El sistema operativo impone que un *kernel* puede ejecutar como máximo por 5 segundos, en caso de que se exceda este tiempo el sistema operativo reinicia el *driver* de video al considerar que este se encuentra inactivo, cancelando la ejecución del *kernel*.

4.3.2. Versión 2: RT-GPU-JM

Esta versión presenta las mismas características que RT-GPU y elimina la restricción de "no explota al máximo la estructura de memoria de la GPU", usando la jerarquía de memoria para almacenar las estructuras de datos que representan la escena a procesar por el algoritmo.

4.3.3. Versión 3: RT-GPU-JM-IR

Esta versión presenta las mismas características que RT-GPU-JM y modifica el algoritmo de intersección rayo-triángulo para disminuir el tiempo de ejecución de cada test de intersección.

4.3.4. Versiones para CPU

Para cada versión desarrollada del *ray tracing* sobre GPU se implementó una versión similar en cuanto a las cualidades de optimización para ejecutarla sobre CPU. De esta manera se dispone de versiones de referencia sobre arquitecturas tradicionales para comparar el desempeño.

Capítulo 5

Análisis experimental

5.1. Introducción

En esta sección se detallan las estrategias seguidas para evaluar los algoritmos implementados, explicando las pruebas realizadas, así como también los resultados obtenidos en ellas. La evaluación también incluye el estudio comparativo de la eficiencia alcanzada por las implementaciones desarrolladas con otras implementaciones similares.

5.2. Estrategias de evaluación de calidad de imagen

Para evaluar los algoritmos implementados es necesario, además de una evaluación de la velocidad de generación de las imágenes, una medida de la calidad de las mismas. Por esta razón se relevaron los métodos con los que se suele evaluar la calidad de los generadores de imágenes. Como resultado de esta investigación no se pudieron determinar estrategias sólidas para evaluar la calidad de las imágenes generadas. Si bien no se encontraron metodologías de evaluación que se adaptaran específicamente a las necesidades del trabajo, se presentan a continuación ideas interesantes que podrían ser útiles a la hora de desarrollar métodos para evaluar la calidad de imágenes generadas por trabajos similares a este.

Se realizó una categorización de las medidas de evaluación de calidad, basada en el artículo de Avcibas y Sankur [23] y analizando el resto de la

información disponible [17, 18, 30, 37]. En el trabajo de Avcibas y Sankur se proponen medidas de calidad de imagen ordenadas según la estrategia en que se basan. Cabe señalar que si bien el trabajo no es sobre generación de imágenes, se pueden establecer ciertas similitudes en los objetivos de todas y cada una de las medidas de calidad de imágenes. Las categorías en las que se dividen los algoritmos de evaluación de calidad son:

- Basados en diferencias a nivel de píxeles.
- Basados en correlación.
- Basados en aristas.
- Basados en análisis espectral.
- Basados en contexto.
- Basados en el sistema visual humano (HVS por su sigla en inglés).

Las estrategias basadas en diferencias a nivel de píxeles son las más simples. Consisten en calcular la diferencia entre dos imágenes tomando como referencia que un píxel en una imagen se corresponde con el mismo píxel de la imagen objetivo, dicho valor indica la diferencia que hay entre ambas imágenes, la generada y la imagen objetivo.

Las estrategias basadas en correlación son muy similares a los anteriores pero pueden introducir una nueva variable: los píxeles se pueden mover y no estar en el mismo lugar en ambas imágenes. Este tipo de algoritmos son útiles en muchos casos para el área de procesamiento de imagen, en especial porque una misma imagen puede ser generada vista de distintos ángulos y en el análisis de calidad de las mismas considerar que no tienen diferencias.

Otra opción para la evaluación, se basa en notar que las aristas (bordes que separan los objetos de la imagen) que componen las imágenes pueden ser utilizadas para el análisis de la calidad de la misma. Esta técnica toma como referencia que para dos imágenes generadas por la misma escena, si una es la correcta (imagen objetivo) entonces en la imagen de la cual se quiere saber si es correcta también deben aparecer las mismas aristas.

Las estrategias que se basan en el análisis espectral miden la distorsión de la señal en fase y magnitud, siguiendo un enfoque del tratamiento de señales. Si bien son particularmente útiles en el análisis de algoritmos de compresión

en los que se da este tipo de distorsión, no se encontró una aplicación directa a este trabajo.

En el análisis de contexto para medir la calidad de imagen se analiza para cada píxel sus vecinos en una cantidad de niveles arbitraria. Estos píxeles en caso de que difieran de alguna manera (varía para cada algoritmo dentro de la familia) modificarán, no solamente la calidad de ellos mismos como se analiza con las estrategias de diferencia por píxel, sino también la calidad de los vecinos. A modo de ejemplo no será lo mismo un píxel negro entre píxeles rojos que un píxel negro entre píxeles blancos.

Por último, para brindar una medida de calidad de la imagen generada existen métodos basados en el análisis de la percepción humana, los cuales utilizan los modelos que se han generado para la percepción del ojo humano. En este tipo de estrategias se considera que dos imágenes son iguales si para la percepción del ojo humano no tienen diferencias. Este es un modelo razonable en muchos aspectos, en particular para la industria audiovisual por ejemplo películas, video juegos, generación de imágenes fotorealistas, entre otras.

Como se puede ver, todas estas estrategias sirven para comparar imágenes y la que más se acerca a los requerimientos de evaluación de este proyecto es la basada en las capacidades de percepción del sistema visual humano, que apuntan a evaluar que imágenes son iguales para el ojo humano. Dentro de esta categoría se analizaron las técnicas más recientes, las cuales permiten calcular una medida de calidad de una imagen distorsionada con respecto a la imagen original. Por más detalles sobre dichas técnicas se pueden consultar los artículos propuestos por Zhai et al. [37], Ghanem et al. [18] y Rao et al. [30]. Ninguna de estas medidas de calidad es aplicable directamente en el contexto de este proyecto, debido a que no se cuenta con una imagen original que sirva de base para realizar la comparación con la imagen generada por el algoritmo.

Dirik et al. [17] abordan el problema de la evaluación de imágenes generadas por un algoritmo. En dicho trabajo se plantea la distinción entre imágenes generadas por un algoritmo de generación de imágenes e imágenes reales tomadas con una cámara. El objetivo que persiguen los autores es similar al que se persigue en el análisis de este trabajo, de esta manera se puede utilizar la medida propuesta por Dirik et al. para decidir si la imagen generada por el algoritmo implementado en este proyecto tiene buena calidad (es decir si parece tomada con una cámara fotográfica). El problema que se presenta es que en este proyecto no se abordan las imágenes fotorealistas dado que el modelo (*ray tracing*) no es un modelo tan preciso. Por este motivo no

se puede utilizar el método propuesto en el artículo de Dirik et al.

5.3. Casos de prueba

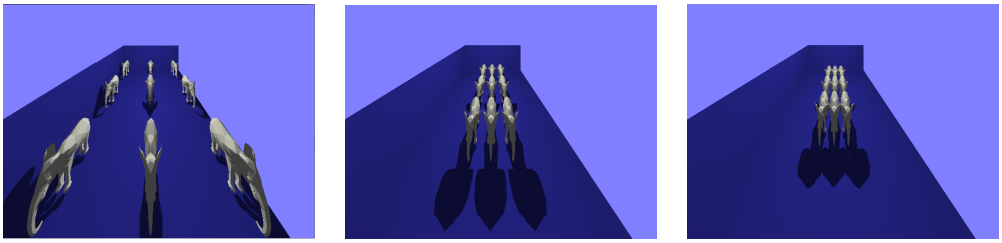
Para evaluar el desempeño de los algoritmos de generación de imágenes implementados es necesario disponer de un conjunto de casos de prueba con distintas características. Además la comparación con implementaciones similares es importante para establecer la calidad del algoritmo de *ray tracing* desarrollado en el marco de este proyecto. Por este motivo se incluyen dentro de los casos de prueba escenas pertenecientes a distintas instituciones que realizan investigación y desarrollo sobre el algoritmo de *ray tracing*. Dentro de esta clase de escenas externas al proyecto, hay escenas usadas comúnmente en los proyectos de generación de imágenes, por ejemplo "*stanford bunny*" y también hay escenas únicas de proyectos particulares.

Al no disponer de un conjunto estándar de pruebas a realizar sobre el algoritmo, ya que no están establecidas por la diversidad de pruebas a realizarse, es que se eligieron y diseñaron casos de prueba según criterios justificados que se adapten a los objetivos del trabajo. En este sentido, al momento de diseñar los casos de prueba para las versiones de *ray tracing* se buscó cubrir los aspectos críticos del algoritmo. Un aspecto importante a tener en cuenta es que los algoritmos implementados usan una grilla uniforme como estructura de aceleración. Como se analizó anteriormente este tipo de estructura no es buena cuando la escena tiene una distribución espacial no uniforme de sus elementos. Por ello resulta importante probar las versiones con un conjunto de escenas que mantengan fija la cantidad de objetos, pero que varíen la distribución de ellos.

La cantidad de objetos de la escena es un aspecto que afecta directamente el tiempo de ejecución de un algoritmo de *ray tracing*. Por este motivo es importante verificar el tiempo de ejecución del algoritmo implementado con escenas que tengan distinta cantidad de objetos pero que mantengan fijas todas las demás propiedades.

5.3.1. Distribución de los objetos en la escena

Para verificar el comportamiento del algoritmo implementado frente a la uniformidad espacial de los objetos de la escena se diseñaron tres casos de prueba. Como se muestra en la Figura 5.1 los tres casos son similares, la única



(a) (b) (c)

Figura 5.1: Escenas con distinta disposición espacial de los objetos.

diferencia entre ellos es la posición de los objetos (cada uno está compuesto por 1148 triángulos) en la escena.

En la Tabla 5.1 se muestran algunas características importantes de estos casos de prueba, en especial en la primer columna se muestra el nombre con el que se hará referencia a la escena de aquí en más.

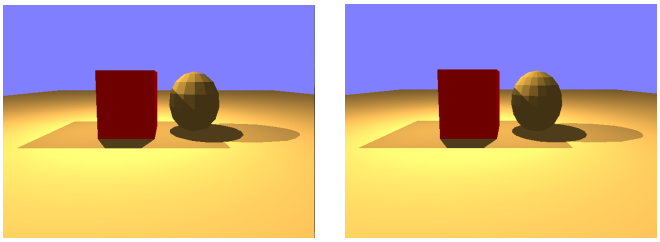
Escena	Objetos	Luces	Triángulos	Archivo	Imagen
DIST_I	9	1	10338	elefantesChicosDistUniforme.obj	5.1(a)
DIST_II	9	1	10338	elefantesChicosDistNoUniforme.obj	5.1(b)
DIST_III	9	1	10338	elefantesChicosDistNoUniformeSOLAP.obj	5.1(c)

Tabla 5.1: Datos de entrada para pruebas de distribución.

5.3.2. Cantidad de primitivas de la escena

Para verificar el comportamiento del algoritmo implementado frente a la cantidad de objetos de la escena de entrada se diseñaron cinco casos de prueba. Los cinco casos de prueba definen la misma escena, la única propiedad que cambia entre uno u otro es la cantidad de primitivas (triángulos) usadas para construir los objetos de la misma. Como se muestra en las Figuras 5.2(a) y 5.2(b) cada escena contiene una esfera y un cubo, y existe una diferencia entre las imágenes generadas dada por la variación de la cantidad de triángulos.

En la Tabla 5.2 se muestran algunas características importantes de estos casos de prueba, en especial en la primer columna se muestra el nombre con el que se hará referencia a la escena de aquí en más.



(a) (b)

Figura 5.2: Imágenes de los casos de prueba con distintas cantidades de primitivas.

Escena	Objetos	Luces	Triángulos	Archivo	Imagen
PRI_I	2	2	194	cajaEsfera1.obj	5.2(a)
PRI_II	2	2	274	cajaEsfera2.obj	-
PRI_III	2	2	348	cajaEsfera3.obj	-
PRI_IV	2	2	482	cajaEsfera4.obj	-
PRI_V	2	2	606	cajaEsfera5.obj	5.2(b)

Tabla 5.2: Características de los casos de prueba con distintas cantidades de primitivas.

5.3.3. Comparación con otras implementaciones

Para la evaluación de desempeño de los algoritmos implementados en el marco de este proyecto, resulta imprescindible la comparación con otros algoritmos de *ray tracing* similares. Por ello se buscaron algoritmos que se ajustaran al modelo de Whitted, implementados sobre CUDA por desarrolladores de *ray tracing*.

Los integrantes del grupo de Computación Gráfica del *Alexandra Institute* de Dinamarca [16] implementaron un algoritmo de *ray tracing* y se encuentra publicado en su página web. Este algoritmo no permite cambiar la escena que renderiza de forma sencilla, ya que su cargador de escena es distinto al que se usa en este proyecto. Como se dispone de información (cantidad de triángulos de cada uno de los elementos) sobre la escena del algoritmo del *Alexandra Institute*, se decidió replicar manualmente dicha escena en el formato que usa el algoritmo implementado en este proyecto. Esta escena esta formada por un conjunto de 13 cajas y una esfera como se muestra en la Figura 5.3(a). Cada caja tiene 2 triángulos por cara y la esfera tiene 80 caras, por lo tanto

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

Figura 5.3: Imágenes de los casos de prueba de cantidad de primitivas.

la escena completa tiene 236 triángulos.

Las escenas cuyos renders se muestran en las Figuras 5.3(b), 5.3(c) y 5.3(d), se encuentran dentro de las escenas clásicas de gran cantidad de proyectos de generación de imágenes. Contar con estas escenas dentro de los casos de prueba de este proyecto es muy importante porque permite comparar con otros proyectos similares. Además es importante que el algoritmo implementado en este proyecto soporte este tipo de casos de prueba, que por lo general se componen de una cantidad considerable (100.000) de primitivas.

En la Tabla 5.3 se muestran algunas características destacables de estos casos de prueba, en especial en la primer columna se muestra una referencia que será usada de aquí en más.

Escena	Objetos	Luces	Triángulos	Archivo	Imagen
ALEXANDRA	14	1	236	escenaAlexandra.obj	5.3(a)
BUDDHA	1	1	100.000	buddhaRT.obj	5.3(b)
DRAGON	1	1	100.000	dragonRT.obj	5.3(c)
BUNNY	1	1	69.698	StanfordBunny.obj	5.3(d)

Tabla 5.3: Escenas que permiten la comparación con otros proyectos.

5.4. Plataformas de ejecución

Las características de los equipos utilizados para ejecutar los casos de prueba se muestran en la Tabla 5.4. Todos los equipos utilizados usan *Windows* como sistema operativo, a excepción del equipo GTX9800 que utiliza *Linux*.

Todos los equipos utilizados para ejecutar los casos de prueba del proyecto usan la versión 2.3 del *driver* de CUDA. Los equipos poseen tarjetas gráficas distintas lo cual implica que las propiedades que afectan la ejecución de las

Equipo	CPU	Memoria Ram	GPU	Memoria GPU
9500M	Core 2 Duo T7500 2.20GHz	4GB DDR2 667 MHz	GeForce 9500M GS	512 MB
9600M	Core 2 Duo P8400 2.26GHz	4GB DDR2 667 MHz	GeForce 9600M GT	512 MB
GTX260	Core 2 Duo E7500 2.93GHz	4GB DDR2 667 MHz	GeForce GTX 260	896 MB
GTX9800	Core 2 Duo E5200 2.50GHz	2GB DDR2 667 MHz	GeForce GTX 9800	512 MB

Tabla 5.4: Equipos utilizados para ejecutar los casos de prueba.

aplicaciones CUDA sobre ellas también lo sean. En la Tabla 5.5 se muestran las principales propiedades relacionadas con CUDA de cada tarjeta gráfica utilizada en el proyecto.

Equipo	Multiprocesadores	Cores	Clock (MHz)	Shader clock (MHz)	Memory clock (MHz)
9500M	4	32	475	950	400
9500M	4	32	500	1250	400
GTX260	24	192	576	1242	999
GTX9800	16	128	675	1688	1100

Tabla 5.5: Características de las GPUs utilizadas.

5.5. Resultados experimentales

Las pruebas realizadas en este proyecto se pueden dividir en dos grandes líneas. La primera es comparar resultados dentro del propio proyecto, por ejemplo la comparación entre los algoritmos implementados, para CPU y para GPU. La otra línea de prueba es la comparación con algoritmos similares implementados por terceros. En esta clase de pruebas se hicieron comparaciones con implementaciones que pudieron ser ejecutadas en los equipos utilizados en el proyecto y también con resultados de otras experiencias similares extraídos de artículos científicos.

La mayoría de las pruebas realizadas se hicieron fijando la resolución de la imagen a generar en 640 por 480 píxeles. Las únicas excepciones a esta regla se dan cuando se hacen pruebas de comparación con algoritmos implementados por terceros. Para las pruebas de comparación con el *Alexandra Institute* se uso una resolución de 800 por 600, mientras que para la comparación con los resultados del artículo de Günther et al. [31] se uso una resolución de 1024 por 1024 píxeles. En el caso del *Alexandra Institute* la resolución quedó determinada por su implementación del algoritmo de *ray tracing*, que no permite variar la misma. En el caso del artículo de Günther la resolución

quedó determinada por los resultados descritos en él, ya que fueron obtenidos usando una resolución fija (1024 por 1024).

Las pruebas realizadas a lo largo del proyecto mostraron que una buena elección del tamaño de la grilla puede incrementar notablemente la velocidad de generación de imágenes, siendo esto un parámetro crítico que se debe definir correctamente. Como primer aproximación se toma la medida sugerida por Thrane y Ole [28], la cual indica que la resolución sea $3\sqrt[3]{N}$ *voxeles* a lo largo del eje más corto, donde N es el número de triángulos de la escena. Después de varias pruebas se comprobó que esta división no siempre es la mejor y que una buena resolución para la grilla se encuentra entre $\sqrt[3]{N}$ y $3\sqrt[3]{N}$ a lo largo del eje más corto. Dentro de este intervalo se debe buscar empíricamente la grilla de mejor rendimiento. En todas las pruebas realizadas en esta sección se siguió esta metodología para encontrar el tamaño de grilla óptimo (o grilla óptima), así mismo se muestran resultados para otros tamaños de grilla para cada escena.

5.5.1. Estudio de las versiones implementadas

La comparación de rendimiento entre las diferentes versiones del algoritmo para GPU se hizo usando los casos de prueba PRI_I a PRI_V. Esta comparación entre versiones consta de dos partes, en la primera se determina la grilla óptima para cada caso de prueba y en la segunda se evalúan los casos de prueba con cada una de las versiones del algoritmo utilizando su grilla óptima.

Para determinar la grilla óptima de cada escena se usa la Versión RT-GPU-JM-IR del algoritmo, ejecutando en el equipo GTX260. En la Tabla 5.6 se muestran los resultados obtenidos al ejecutar los casos de prueba sobre GPU. Observando los resultados se puede concluir que la grilla óptima para todos los casos de prueba se construye partiendo en dos cada eje.

Los resultados obtenidos en estas primeras pruebas muestran que a medida que aumenta la cantidad de primitivas con que está construida la escena aumenta el tiempo de generación de imagen, y por lo tanto disminuyen los *frames* por segundo (FPS) alcanzados. Antes de ejecutar los casos de prueba se planteó la hipótesis de que a mayor cantidad de primitivas en la escena mayor es el tiempo de generación de imagen, la cual es validada por las pruebas realizadas en este sentido. También se pensaba que al aumentar la cantidad de primitivas de la escena aumentaría la cantidad de *voxeles* que debía tener la grilla óptima, pero esto no fue validado por los resultados

obtenidos. Esto puede deberse a que el incremento de la cantidad de primitivas no es suficientemente grande como para obligar a aumentar la resolución de la grilla.

Escena	**Tamaño de grilla**					
	1x1x1	2x2x2	4x4x4	6x6x6	10x10x10	15x15x15
PRI_I	18.7	**36.7**	32.7	29.7	27.3	24.7
PRI_II	13.9	**27.6**	25.4	23.3	21.5	20.1
PRI_III	11.3	**24.5**	22.8	20.9	19.2	18.0
PRI_IV	8.4	**18.5**	18.0	16.5	15.9	15.1
PRI_V	6.7	**16.6**	15.5	14.6	13.8	13.5

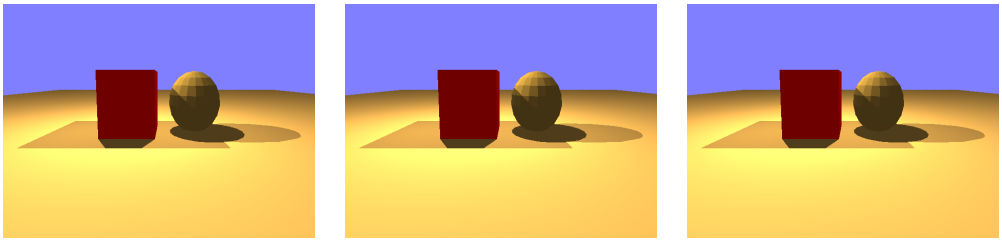
Tabla 5.6: FPS de los casos PRI_X en el equipo GTX260 sobre GPU.

Una vez determinada la grilla óptima para cada caso de prueba, se evalúan los casos, utilizando su grilla óptima en cada una de las versiones del algoritmo para GPU, en la Tabla 5.7 se muestran los resultados obtenidos. Los resultados de la Tabla 5.7 reflejan la evolución del algoritmo a medida que se fue mejorando la implementación, incrementándose los FPS para todos los casos de prueba.

Escena	Tamaño Grilla Óptimo	RT-GPU (FPS)	RT-GPU-JM (FPS)	RT-GPU-JM-IR (FPS)
PRI_I	2x2x2	5.8	26.2	**36.7**
PRI_II	2x2x2	5.1	20.2	**27.6**
PRI_III	2x2x2	4.9	18.2	**24.5**
PRI_IV	2x2x2	4.3	14.1	**18.5**
PRI_V	2x2x2	3.9	12.6	**16.6**

Tabla 5.7: FPS de PRI_X en el equipo GTX260 sobre GPU para cada versión del algoritmo.

El incremento del tiempo de generación de imagen es mayor en el primer cambio de versión, esta diferencia está determinada por el uso de la jerarquía de memoria de la GPU. En la Versión RT-GPU todos los accesos a memoria son a memoria global mientras que en la Versión RT-GPU-JM la mayoría de los datos de entrada del algoritmo de *ray tracing* se encuentran en memoria de textura. Considerando estos casos de prueba, el correcto uso de la jerarquía de memoria permite que el algoritmo pierda menos tiempo accediendo a memoria, siendo tres veces y media en promedio más rápido que el algoritmo que no la usa.



(a) (b) (c)

Figura 5.4: Render de PRI_V en GPU con la Versión RT-GPU, RT-GPU-JM y RT-GPU-JM-IR respectivamente.

En el segundo cambio de versión se mejora el algoritmo de intersección rayo-triángulo, el nuevo test de intersección logra el mismo objetivo que el anterior pero con menos operaciones aritméticas, lo cual implica una disminución del tiempo de generación de la imagen. Considerando los resultados obtenidos, esta mejora del algoritmo de *ray tracing* ayuda a que la Versión RT-GPU-JM-IR genere la imagen un 30 % más rápido que en la Versión RT-GPU-JM.

En la Figura 5.4 se muestran los renders del caso de prueba PRI_V con cada una de las versiones del algoritmo implementado en CUDA. No se observan diferencias importantes entre las imágenes generadas por las diferentes versiones del algoritmo. El aumento en la cantidad de FPS de las diferentes versiones implementadas no implica una perdida de calidad de imagen.

5.5.2. Estudio de las implementaciones en C y en CUDA

La comparación de rendimiento entre el algoritmo para CPU y el algoritmo para GPU se hizo usando los casos de prueba DIST_I, DIST_II, DIST_III y BUNNY. Para esta comparación se usa la versión más eficiente de los algoritmos la Versión RT-GPU-JM-IR y la Versión RT-CPU-IR, ejecutando en el equipo GTX260. En la Tabla 5.8 se muestran los resultados obtenidos al ejecutar los casos de prueba sobre CPU, mientras que en la Tabla 5.9 se muestran los resultados obtenidos al ejecutar los mismos casos sobre GPU.

Los resultados obtenidos sugieren que el tamaño de la grilla depende exclusivamente de la cantidad de triángulos con que esta construida la escena, ya que en los tres primeros casos (que tienen la misma cantidad de triángulos) el tamaño de grilla donde se logran más FPS es siempre el mismo. Además

Escena	Tamaño de grilla					
	10x10x10	22x22x22	50x50x50	65x65x65	100x100x100	200x200x200
DIST_I	0.3	0.9	**1.4**	1.3	1.1	0.3
DIST_II	0.3	1.0	**1.5**	1.4	1.1	0.3
DIST_III	0.4	1.3	**1.6**	1.4	1.1	0.3
	20x20x20	41x41x41	80x80x80	123x123x123	200x200x200	300x300x300
BUNNY	1.2	3.0	**4.2**	4.0	3.2	2.1

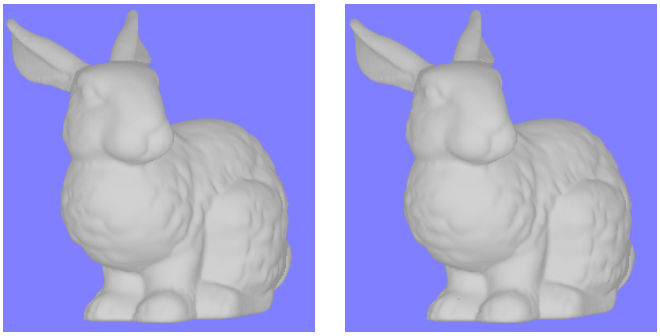
Tabla 5.8: FPS de DIST_I, DIST_II, DIST_III y BUNNY en el equipo GTX260 sobre CPU.

Escena	Tamaño de grilla					
	10x10x10	22x22x22	50x50x50	65x65x65	100x100x100	200x200x200
DIST_I	10.5	15.4	**17.2**	14.2	10.7	5.1
DIST_II	10.7	16.7	**20.1**	17.5	12.0	5.1
DIST_III	8.9	18.5	**21.1**	18.2	12.5	5.1
	20x20x20	41x41x41	80x80x80	123x123x123	200x200x200	300x300x300
BUNNY	13.9	20.7	**23.8**	20.8	14.4	10.1

Tabla 5.9: FPS de DIST_I, DIST_II, DIST_III y BUNNY en el equipo GTX260 sobre GPU.

como lo muestra el caso BUNNY, al incrementarse la cantidad de primitivas de la escena aumenta la resolución de la grilla óptima. Se concluye también que el tamaño de la grilla óptima es independiente al *hardware* de ejecución, la grilla que permite más FPS tiene igual resolución en CPU y en GPU.

Los casos de prueba DIST_I, DIST_II y DIST_III fueron pensados para buscar una debilidad de la estructura de aceleración. La debilidad de la estructura de sub-división espacial uniforme se da cuando los objetos de la escena están distribuidos de forma no uniforme en la escena. Es por esto que se pensaba que con el caso DIST_III, que tiene todos los objetos concentrados en el centro de la escena, se lograrían menos FPS que con el DIST_II y con el DIST_I. De la misma forma se pensaba que con el caso DIST_II se lograrían menos FPS que con el caso DIST_I. Los resultados obtenidos reflejan totalmente lo contrario a lo que se pensaba de antemano. La explicación para este comportamiento en este tipo de escenas, es que cuánto más uniformemente distribuidos en la escena estén los objetos, más sombra arrojan. Como el cálculo de sombra es computacionalmente costoso, se cree que este costo contrarresta al beneficio que brinda la grilla cuando los objetos están distribuidos de forma uniforme en la escena.



(a) (b)

Figura 5.5: Render de BUNNY en CPU y en GPU respectivamente.

Los resultados obtenidos con los casos de prueba DIST_I, DIST_II, DIST_III y BUNNY demuestran que el algoritmo para GPU genera imágenes en un tiempo menor que el algoritmo para CPU. En la Tabla 5.10 se muestra la aceleración lograda por el algoritmo para GPU para cada caso de prueba. En promedio, considerando los cuatro casos de prueba, el algoritmo para GPU es más de 11 veces más rápido que el algoritmo para CPU. En la Figura 5.5(a) se muestra la imagen generada por el algoritmo implementado en C, mientras que en la Figura 5.5(b) se muestra la imagen generada por el algoritmo implementado en CUDA. Observando estas imágenes no se percibe diferencia alguna, por lo que es posible concluir que el algoritmo para GPU logra una muy buena aceleración con respecto al que ejecuta en CPU y sin pérdida en la calidad de imagen.

Escena	Tamaño Grilla	CPU (FPS)	GPU (FPS)	Aceleración
DIST_I	50x50x50	1.4	17.2	12
DIST_II	50x50x50	1.5	20.1	**14**
DIST_III	50x50x50	1.6	21.1	13
BUNNY	80x80x80	4.2	23.8	6

Tabla 5.10: Aceleración lograda por algoritmo para GPU.

5.5.3. Estudio entre distintos equipos

La comparación de rendimiento entre los diferentes equipos del proyecto se hizo usando la Versión RT-GPU-JM-IR del algoritmo implementado para

GPU. Los casos de prueba de esta comparación son: DRAGON, BUDDHA y ALEXANDRA. En cada equipo utilizado en el proyecto, se corren los casos de prueba usando varios tamaños de grilla, de esta forma se puede determinar el tamaño de grilla óptimo para cada caso y equipo. El mejor tiempo de generación de imagen para cada caso y equipo es analizado y comparado con los demás.

En las tablas 5.11, 5.12, 5.13 y 5.14 se muestran los resultados obtenidos para los equipos 9500M, 9600M, GTX260 y GTX9800 respectivamente.

Escena	**Tamaño de grilla**					
	20x20x20	46x46x46	92x92x92	138x138x138	180x180x180	230x230x230
DRAGON	1.4	2.3	**2.8**	2.0	1.6	1.3
BUDDHA	1.3	2.4	**2.5**	2.1	1.7	1.3
	1x1x1	3x3x3	6x6x6	10x10x10	15x15x15	30x30x30
ALEXANDRA	4.6	11.5	**15.6**	13.1	12.3	9.1

Tabla 5.11: FPS de DRAGON, BUDDHA y ALEXANDRA en el equipo 9500M sobre GPU.

Escena	**Tamaño de grilla**					
	20x20x20	46x46x46	92x92x92	138x138x138	180x180x180	230x230x230
DRAGON	1.7	3.3	**3.4**	2.6	2.0	1.6
BUDDHA	1.4	2.8	**3.1**	2.6	2.1	1.7
	1x1x1	3x3x3	6x6x6	10x10x10	15x15x15	30x30x30
ALEXANDRA	5.9	14.0	**20.0**	18.4	17.5	12.8

Tabla 5.12: FPS de DRAGON, BUDDHA y ALEXANDRA en el equipo 9600M sobre GPU.

Escena	**Tamaño de grilla**					
	20x20x20	46x46x46	92x92x92	138x138x138	180x180x180	230x230x230
DRAGON	5.0	12.2	**16.8**	14.2	11.4	9.3
BUDDHA	4.9	9.8	**12.4**	11.4	10.3	8.4
	1x1x1	3x3x3	6x6x6	10x10x10	15x15x15	30x30x30
ALEXANDRA	28.0	49.3	**71.2**	67.6	62.5	49.4

Tabla 5.13: FPS de DRAGON, BUDDHA y ALEXANDRA en el equipo GTX260 sobre GPU.

Escena	Tamaño de grilla					
	20x20x20	46x46x46	92x92x92	138x138x138	180x180x180	230x230x230
DRAGON	5.6	12.3	**12.8**	9.0	6.8	5.0
BUDDHA	5.0	10.0	**10.4**	7.9	6.1	4.5
	1x1x1	3x3x3	6x6x6	10x10x10	15x15x15	30x30x30
ALEXANDRA	29.5	57.7	**74.4**	70.5	69.6	50.1

Tabla 5.14: FPS de DRAGON, BUDDHA y ALEXANDRA en el equipo GTX9800 sobre GPU.

De los resultados obtenidos se desprende que para los casos de prueba DRAGON, BUDDHA y ALEXANDRA el tamaño de grilla donde se logran más FPS es el mismo independientemente del equipo en donde se ejecuten. El modelo de programación de CUDA es capaz de ejecutar más hilos en paralelo cuanto más procesadores tenga la GPU. Basado en esto se puede afirmar que el tamaño de grilla óptimo no depende del grado de paralelismo que se logre.

En la Figura 5.6 se muestra un gráfico de la cantidad de FPS en función de la cantidad de procesadores de la GPU para cada caso de prueba utilizando su grilla óptima.

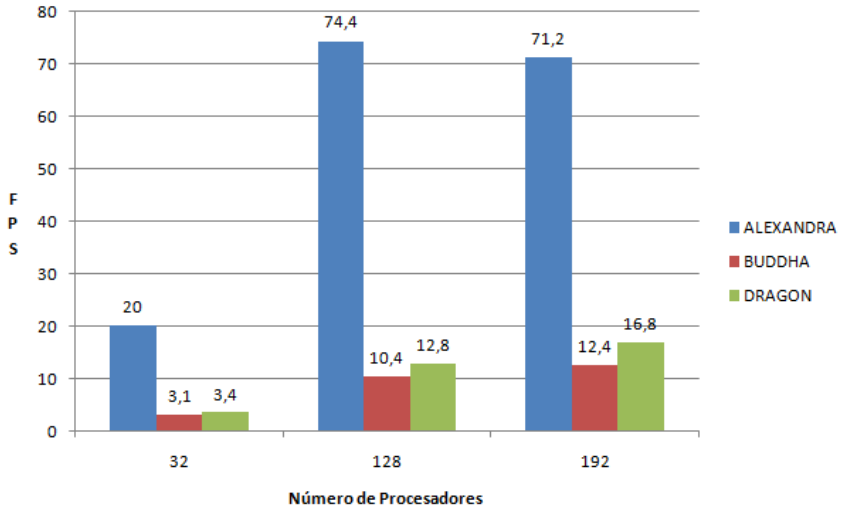


Figura 5.6: FPS en función de la cantidad de procesadores de la GPU.

Todas las pruebas realizadas en esta comparación fueron realizadas con los casos de prueba DRAGON y BUDDHA, los cuales se consideran com-

plejos por estar conformados por 100.000 triángulos, confirmaron que cuanto más procesadores posea la GPU más rápido será la generación de imágenes mediante el algoritmo de *ray tracing* para CUDA. Por otro lado los resultados de las pruebas realizadas con el caso ALEXANDRA (considerado una escena sencilla por estar conformado por 236 triángulos) permiten afirmar que el *overhead* generado por el uso de más procesadores afecta al tiempo de generación de imagen. Igualmente los resultados demuestran que la aplicación CUDA implementada puede escalar automáticamente en el número de procesadores de la GPU, esta importante característica surge como consecuencia del modelo de programación de CUDA que permite lograrlo muy fácilmente.

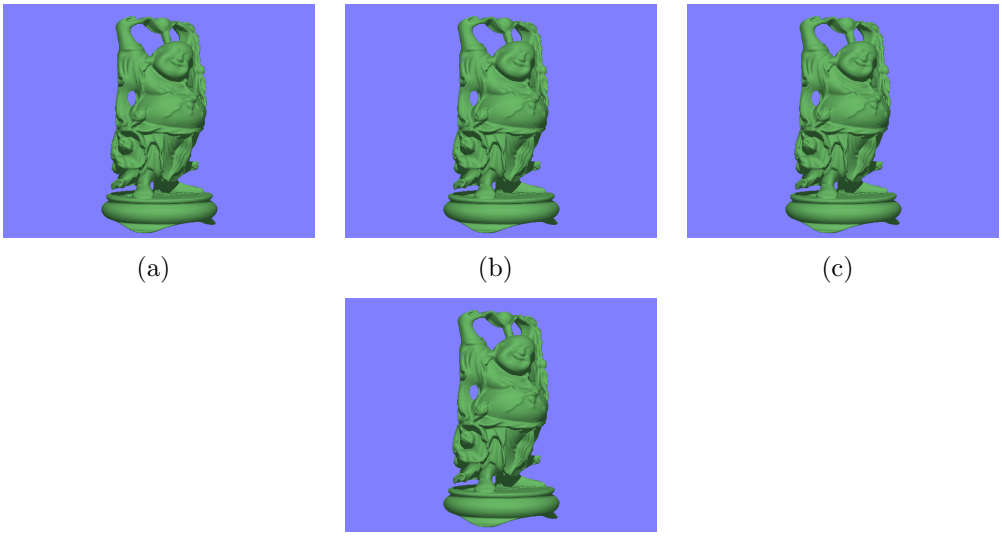
La GPU del equipo GTX260 tiene seis veces más procesadores que la del equipo 9600M. Como se muestra en la Figura 5.6, si consideramos los resultados de los casos DRAGON, BUDDHA y ALEXANDRA con sus grillas óptimas, se observa que la aceleración en la generación de imagen nunca llega a 6, sino que es aproximadamente 5, 4 y 4 respectivamente. Basado en esta información se puede concluir que la ganancia de FPS no es lineal con respecto al aumento de procesadores. Esto se debe al tiempo empleado para la lectura de datos de entrada, a retrasos por serialización de los accesos a memoria o por sincronizaciones entre hilos al momento de escribir los resultados.

En la Figura 5.7 se muestran renders del caso de prueba BUDDHA con el algoritmo implementado en CUDA. El render de la Figura 5.7(a) fue generado usando el equipo 9500M, el de la Figura 5.7(b) fue generado usando el equipo 9600M, el de la Figura 5.7(c) usando el equipo GTX260 y el de la Figura 5.7(d) con el equipo GTX9800. No se observan diferencias importantes entre las imágenes generadas por los diferentes equipos utilizados en el proyecto. Entonces, el aumento del poder de cómputo de la GPU implica un aumento en la cantidad de FPS, esta disminución del tiempo de generación de imágenes no implica perdida de calidad de las mismas.

5.5.4. Estudio comparativo con otras implementaciones

En el trabajo de Günther et al. [31] se desarrolló una implementación del algoritmo de *ray tracing* similar a la de este proyecto. En este trabajo se generan renders de una escena que es igual a la escena BUNNY y se presentan resultados de los tiempos de generación de imágenes.

Se deben considerar algunas excepciones con respecto a la igualdad de



(a) (b) (c)

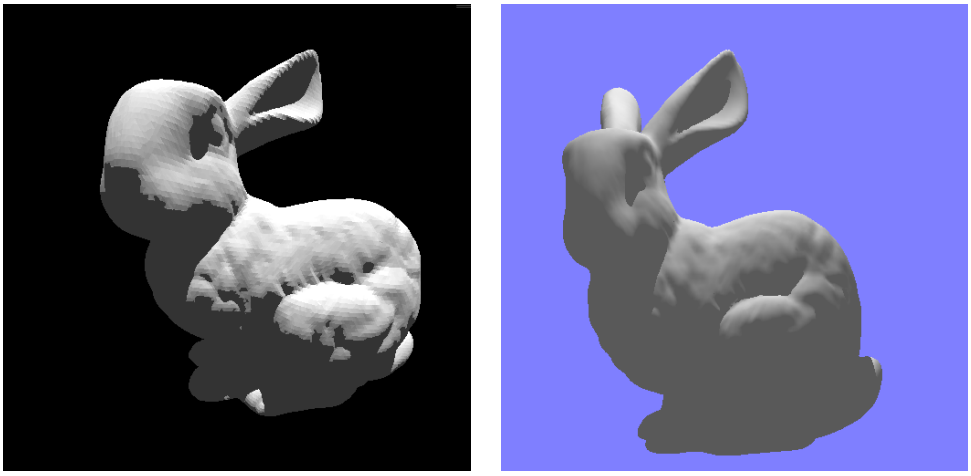
(d)

Figura 5.7: Render de BUDDHA en GPU sobre los equipos 9500M, 9600M, GTX260 y GTX9800 respectivamente.

las escenas, la escena BUNNY utilizada en este proyecto está construida mediante la observación minuciosa de un render del trabajo de Günther et al. Por ejemplo, la cantidad de luces es la misma pero la posición de estas no son exactamente las mismas, ya que en el artículo no se brinda este tipo de información. Asímismo, el material del objeto principal de la escena no pudo ser reproducido con exactitud debido a que en el trabajo no se brinda dicha información.

Debido a que en el trabajo de Günther et al. solamente se presentan resultados y no se tiene acceso al código fuente y a la especificación de la escena, las pruebas con el algoritmo implementado en este proyecto se adaptaron a dicho trabajo para lograr experimentos comparables. En dicho trabajo se usa una resolución de 1024 por 1024 píxeles, por lo tanto las pruebas de comparación se hacen utilizando esta resolución.

Günther et al. usan la estructura *kd-tree* como método de aceleración espacial, como se dijo en la sección de relevamiento de estructuras de aceleración, esa estructura es más eficiente que la utilizada en este proyecto. Debido a esto, para la comparación con el algoritmo implementado en este proyecto se usa el tamaño de grilla que permite generar la imagen en el menor



(a) (b)

Figura 5.8: Render de BUNNY en el trabajo de Günther et al. y en este proyecto respectivamente.

tiempo posible (dicho tamaño se definió empíricamente en la Sección 5.5.2).

La GPU utilizada en el trabajo de Günther es una nVidia modelo GeForce 8800 GTX, la cual posee 128 núcleos. En el marco de este proyecto la GPU que más se adapta para este caso es la del equipo GTX9800, ya que posee la misma cantidad de núcleos de procesamiento. Asimismo se realizaron pruebas en el equipo GTX260 de forma de evaluar el comportamiento del algoritmo implementado en un equipo mas potente y bajo las mismas condiciones que en el trabajo de Günther et al.

En la Figura 5.8(a) se muestra el render extraído del trabajo de Günther et al. y en la Figura 5.8(b) se muestra el render de la escena BUNNY generado con la Versión RT-GPU-JM-IR del algoritmo implementado para GPU.

El rendimiento en FPS del algoritmo implementado en este proyecto bajo las condiciones descritas anteriormente es de 4.7 FPS utilizando el equipo GTX9800 y de 6.1 FPS utilizando el equipo GTX260, mientras que el rendimiento del algoritmo del trabajo de Günther es de 5.9 FPS. Si se considera la comparación utilizando el equipo GTX9800, el algoritmo implementado en el trabajo de Günther es un $25\,\%$ más rápido que el algoritmo implementado en este proyecto. Es importante resaltar que utilizando la misma cantidad de núcleos de procesamiento, la estrategia de aceleración espacial pasa a ser un aspecto importante en el desempeño de los algoritmos y que tal como

se relevó al inicio del proyecto (Sección 3.4) la estructura Kd-tree permite lograr más aceleración que la sub-división espacial uniforme. Considerando las pruebas realizadas en el equipo GTX260, el desempeño de ambos algoritmos para este caso de prueba es muy similar. El algoritmo implementado en este proyecto tiene a favor que la GPU donde ejecuta es más potente y en contra que usa una estructura de aceleración menos eficiente. El algoritmo implementado en el trabajo de Günther tiene a favor que usa una estructura de aceleración más eficiente y en contra que ejecuta en una GPU de menor capacidad de cálculo. De todas formas es importante que el algoritmo implementado en este proyecto tenga rendimientos competitivos con algoritmos desarrollados en otros proyectos similares.

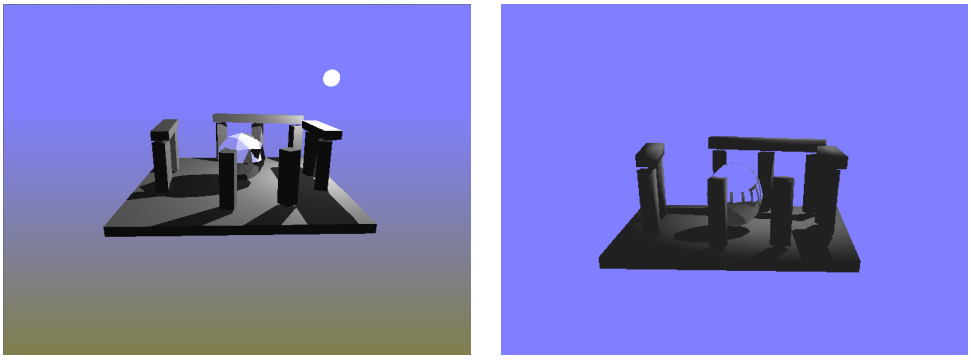
El *Alexandra Institute* [16] ha desarrollado un algoritmo de *ray tracing* para CUDA que permite comparar su rendimiento con el implementado en este proyecto. La comparación se realiza únicamente mediante el caso de prueba ALEXANDRA, debido a que el algoritmo implementado por este instituto como se mencionó anteriormente no permite cambiar de escena fácilmente. Cabe destacar que la escena ALEXANDRA fue construida mediante observación de imágenes generadas por el *ray tracing* desarrollado en dicho instituto, por ende las escenas de entrada de ambos algoritmos no son formalmente iguales, aunque sí muy similares.

La resolución de la imagen que genera el algoritmo de *ray tracing* implementado por el *Alexandra Institute* es de 800 por 600 píxeles y no puede ser modificada, por lo tanto se usa esta resolución para las pruebas de comparación. El *ray tracing* del *Alexandra Institute* no usa estructura de aceleración espacial, mientras que el algoritmo implementado en este proyecto usa un tamaño de grilla de 6 *voxeles* por dimensión. Este tamaño de grilla genera el mejor rendimiento del algoritmo y fue hallado empíricamente en la sección 5.5.3.

Debido a que se tiene acceso al ejecutable del algoritmo del *Alexandra Institute* y que esta compilado para *Windows x86* se optó por ejecutar la comparación de rendimiento entre ambos algoritmos en el equipo 9600M.

En la Figura 5.9(a) se muestra un render generado por la aplicación desarrollada por el *Alexandra Institute* y en la Figura 5.9(b) se muestra el render de la escena ALEXANDRA generado con la Versión RT-GPU-JM-IR del algoritmo implementado para GPU sobre el equipo 9600M.

El rendimiento en FPS del algoritmo implementado en este proyecto bajo las condiciones descritas anteriormente es de 13 FPS, mientras que el rendimiento del *ray tracing* del *Alexandra Institute* es de 11. 7 FPS. Es nece-



(a) (b)

Figura 5.9: Render de la aplicación del *Alexandra Institute* y render de ALEXANDRA en GPU sobre equipo 9600M usando la Versión RT-GPU-JM-IR, respectivamente.

sario recalcar que el algoritmo implementado en este proyecto renderiza escenas genéricas, es decir, no fue concebido para generar imágenes de sólo un tipo de escena, lo cual implica que para una misma escena se requiere mayor espacio de almacenamiento y más accesos a memoria. Parte de la escena del algoritmo desarrollado por el instituto de Dinamarca se encuentra en el código fuente del mismo, lo cual brinda mayor eficiencia para el caso particular. Es posible afirmar que para este caso de prueba ambos algoritmos tienen rendimientos similares, esto demuestra que el algoritmo implementado es competitivo con otros algoritmos de *ray tracing* implementados en CUDA.

En las imágenes generadas por ambos algoritmos se pueden apreciar diferencias, como por ejemplo el color de fondo que no pudo ser correctamente reproducido en la escena de prueba ALEXANDRA. Se nota otra diferencia en el brillo especular de la esfera, esto se debe a la reproducción incorrecta del material de la misma. Aunque en el render generado por el algoritmo implementado en este proyecto se aprecia mejor el reflejo de los objetos cercanos a la esfera sobre la superficie de la misma. Otra diferencia es que el algoritmo del *Alexandra Institute* trata a las luces de la escena como un objeto más de la misma, esto no es considerado por el algoritmo implementado en este proyecto. Se puede concluir que no existen diferencias de calidad significativas entre ambos renders.

5.6. Conclusiones

Entre los resultados más importantes de las pruebas realizadas se puede destacar que el algoritmo para GPU es más de 10 veces más rápido que el algoritmo implementado para CPU, lo cual demuestra que la paralelización a nivel de rayos primarios realizada sirve para acelerar el algoritmo de *ray tracing*.

Por otro lado, el método de aceleración mediante GPU permitió lograr tiempos interactivos de generación de imagen para escenas simples. Asimismo, es posible lograr tiempos interactivos para escenas complejas debido a que aún existen puntos del algoritmo por mejorar, como por ejemplo la estructura de aceleración espacial.

Otro punto relevante es la comparación con proyectos similares, en este sentido se obtuvieron buenos resultados debido a que el tiempo de generación de imagen del algoritmo implementado en este proyecto resultó competitivo con los tiempos logrados por los algoritmos desarrollados en otros esfuerzos.

Por último, las pruebas realizadas en los diversos equipos utilizados en este proyecto permiten afirmar que el algoritmo de *ray tracing* implementado para CUDA escala de buena manera y automáticamente en el número de procesadores de la GPU. Esta propiedad permite mejorar de forma automática los tiempos de generación de imagen a medida que el poder de cómputo de las tarjetas gráficas se incrementa.

Capítulo 6

Conclusiones y trabajo a futuro

Tomando en cuenta los objetivos definidos al inicio de este trabajo, es posible afirmar que las metas propuestas se lograron exitosamente.

El trabajo se inició con un estudio general de los distintos métodos de generación de imágenes por computadora, con mayor profundidad en el algoritmo de *ray tracing* propuesto por Turner Whitted. Posteriormente se continuó con un estudio sobre la utilización de GPUs como plataforma de ejecución de aplicaciones paralelas. Se investigaron aspectos de la arquitectura de las GPUs, y especialmente herramientas para su programación, y en particular el lenguaje de programación CUDA. Luego se continuó con un relevamiento de los diversos métodos de aceleración para el algoritmo de *ray tracing*, haciendo énfasis en los métodos de aceleración espacial y sus implementaciones sobre GPUs. Por último, se realizó un relevamiento del estado del arte de algoritmos de *ray tracing* interactivos, haciendo hincapié en los algoritmos interactivos implementados sobre arquitecturas multiprocesador de memoria compartida. En estas etapas se adquirieron valiosos conocimientos que luego permitieron realizar el diseño e implementación de las soluciones propuestas.

Siguiendo un proceso de desarrollo iterativo e incremental, se realizó la implementación del algoritmo de *ray tracing* sobre GPU paralelizando a nivel de rayos primarios y utilizando una sub-división espacial uniforme para disminuir la cantidad evaluaciones de intersección rayo-triángulo. De la primera iteración del proceso surgió como resultado la versión llamada RT-GPU, la cual permite generar imágenes utilizando la GPU para acelerar el cómputo. Sin embargo no explota al máximo la jerarquía de memoria de la GPU y no utiliza un algoritmo de intersección rayo-triángulo optimizado. En la si-

guiente iteración se implementó una estrategia que permite el uso eficiente de la jerarquía de memoria de la GPU por parte del algoritmo de *ray tracing*, dando lugar a la versión llamada RT-GPU-JM. Posteriormente, en la última iteración se optimizó el algoritmo de intersección rayo-triángulo y se obtuvo la versión final del *ray tracing* implementado sobre GPU, RT-GPU-JM-IR. Asimismo para cada versión del algoritmo implementado sobre GPU se implementó una versión equivalente para CPU, de forma de evaluar la aceleración lograda al paralelizar el algoritmo sobre GPU.

Otro resultado importante del proyecto es un conjunto de casos de prueba para evaluar algoritmos de iluminación. En una primera etapa se relevó la existencia de *benchmarks* para evaluar este tipo de algoritmo, este relevamiento no permitió establecer un conjunto de casos de prueba que se adaptaran a la realidad de este proyecto. Por esta razón, durante el proceso de implementación se diseñaron y construyeron diversos casos de prueba con el propósito de evaluar el desempeño de cada versión del algoritmo. Los casos de prueba fueron diseñados para cubrir aspectos que se consideraron importantes, por ejemplo puntos débiles de la estructura de aceleración espacial empleada, desempeño del algoritmo frente a variaciones de la cantidad de triángulos de la escena, comparación de desempeño con el estado del arte en la materia y la escalabilidad en las plataformas de prueba.

Los resultados de las pruebas de comparación entre las diferentes versiones del algoritmo para GPU permitieron concluir que el correcto uso de la jerarquía de memoria de la GPU es muy importante, ya que la aceleración lograda al pasar de la versión RT-GPU a la versión RT-GPU-JM fue de más de 3x. Se observa también que es importante minimizar los accesos a la memoria global de la GPU, utilizando la memoria compartida disponible entre bloques de threads o la memoria de textura siempre que sea posible, ya que la diferencia de velocidades entre ambas memorias y la memoria global es muy importante. Los resultados obtenidos al realizar pruebas de desempeño comparando las últimas dos versiones implementadas para GPU permiten concluir que es determinante que el *ray tracing* incluya un algoritmo eficiente de chequeo de intersección rayo-triángulo, ya que reduciendo un cuarto la cantidad de operaciones aritméticas del chequeo se logró un 30 % más de velocidad en la generación de la imagen.

Las pruebas de comparación de desempeño entre la versión final del algoritmo para GPU y su correspondiente para CPU permitieron concluir que el objetivo de acelerar el algoritmo *ray tracing* fue alcanzado con éxito. Todas las pruebas realizadas en este sentido arrojaron que el algoritmo para GPU

es más rápido que su correspondiente para CPU, llegando en el mejor caso hasta una aceleración que supera los 13x.

Por otro lado, la comparación entre GPU y CPU permitió concluir que el tamaño de la grilla de aceleración espacial es independiente del *hardware* donde ejecute el algoritmo y que solo depende de la cantidad de triángulos que componen la escena, ya que para todas las escenas de prueba el tamaño de grilla donde se logran más FPS (grilla óptima) es el mismo en GPU que en CPU.

Se utilizaron cuatro GPUs sobre *Windows* y *Linux* para ejecutar los casos de prueba diseñados, los datos obtenidos permitieron concluir resultados importantes. Los datos de desempeño obtenidos al ejecutar los mismos casos de prueba sobre distintas plataformas usando la versión final del algoritmo para GPU permiten concluir que, para escenas complejas (100.000 triángulos en el contexto de este proyecto), cuanto más procesadores posea la GPU más rápido será la generación de imágenes mediante el algoritmo de *ray tracing* para CUDA. Asimismo se concluye que para escenas simples el *overhead* introducido por demasiado paralelismo puede afectar el tiempo de generación de imagen. Sin duda la propiedad del algoritmo más sobresaliente que los resultados sobre diversas plataformas permitieron visualizar, es la escalabilidad automática en el número de procesadores de la GPU. Este es un beneficio del modelo de programación de CUDA y permite lograr mejores resultados día a día dado el vertiginoso crecimiento del poder de computo de las GPUs actuales. Por otro lado, al evaluar los resultados sobre diferentes plataformas se concluyó que el tamaño de la grilla óptima no depende del grado de paralelismo que se logre. Además se observó que la aceleración lograda por el aumento de la cantidad de procesadores no es lineal, lo cual puede deberse a retrasos por serialización de los accesos a memoria o por sincronizaciones entre hilos al momento de escribir los resultados.

Los resultados de las pruebas de comparación entre la versión final del algoritmo implementado en este proyecto y algoritmos desarrollados en otros proyectos similares demostraron que este proyecto es competitivo con el estado del arte en la materia, tanto en tiempo de generación como en calidad de imagen. Algunos aspectos del trabajo fueron validados mediante la presentación de un artículo de divulgación científica, *Improving the Performance of the Ray Tracing Algorithm with a GPU*, en las JCC 2010 (Jornadas Chilenas de Computación de 2010). El artículo fue aceptado por la revisión, y será incluido en las actas de conferencia publicadas anualmente por IEEE. La versión preliminar del artículo se encuentra en el Apéndice C.

Como conclusión final se puede destacar que el método de aceleración mediante GPU alcanzó tiempos de generación de imágenes que permiten abordar estrategias interactivas para escenas simples. Además la aceleración lograda no implica pérdida de calidad de imagen, ya que en ninguna de las ejecuciones de los casos de prueba se verificaron diferencias entre imágenes generadas en GPU y en CPU.

A partir del trabajo realizado y las conclusiones extraídas, es posible identificar diversas líneas de trabajo futuro que se presentan a continuación.

Una primera línea de trabajo a futuro es mejorar la estructura de aceleración espacial del *ray tracing* implementado. La primer opción es optimizar la grilla uniforme que se utiliza actualmente, para optimizarla se puede agregar una etapa más a su algoritmo de construcción para convertir en un solo *voxel* varios *voxeles* vacíos (o que contengan menos de k triángulos en su interior). Otra opción es cambiar la estructura uniforme por una estructura que se adapte más a la escena, como la kd-tree. Luego de analizado el estado del arte en esta materia en el transcurso de este proyecto de grado, se piensa que con la estructura kd-tree se obtendrían mejores resultados. De esta manera se podría elevar el limite de cantidad de triángulos que tiene el algoritmo actual para renderizar escenas en tiempos interactivos.

En cuanto a mejorar la calidad de las imágenes una línea de trabajo es introducir algún método de *antialiasing*, como por ejemplo *supersampling*, *adaptive sampling* o *stochastic sampling*. Otra opción de mejora en este sentido es eliminar la restricción que tiene el algoritmo actual en cuanto a que las escenas no pueden tener objetos reflexivos y transparentes a la vez. Para remover esta restricción es necesario implementar un *stack* a nivel de la GPU para manejar el un árbol de rayos en lugar de una lista como se maneja actualmente.

En cuanto a diseño global del algoritmo una línea de trabajo es re-diseñar los *kernels* utilizados por el algoritmo actual. Actualmente existe un *kernel* principal que se encarga de lanzar los rayos primarios, luego lanzar los de sombra, posteriormente lanzar los rayos de reflexión y refracción y por último calcular el color del píxel. Este diseño tiene problemas porque para escenas complejas el *kernel* principal tiene un tiempo de ejecución demasiado alto y es abortado por el sistema operativo. La solución que podría solucionar este problema es dividir el *kernel* principal en varios *kernels* más pequeños, donde cada uno tenga la responsabilidad de hacer parte del trabajo que hace el principal en la actual implementación.

La siguiente línea de trabajo es a más largo plazo y consiste en estudiar

en profundidad otros algoritmos de generación de imagen, como radiosidad o *photon mapping*, y sus implementaciones sobre GPUs, de modo de aprovechar la experiencia en programación sobre GPUs adquirida en este proyecto de grado aplicada a otras herramientas del área.

El impulso de OpenCL como estándar de programación de GPUs, para todas las tecnologías y no solo para NVIDIA, sugiere otra línea de trabajo futuro a más largo plazo, que consiste en evaluar la utilización de otros lenguajes de programación, en particular OpenCL. También sería interesante lograr un *ray tracing* que utilice varias GPUs y que sea capaz de dividir la generación de la imagen entre ellas.

Apéndice A

Estructuras de datos

En las próximas dos secciones se presentan las dos versiones de las estructuras de datos usadas durante el desarrollo de este proyecto.

A.1. Versión 1

Algoritmo 8 Estructura de datos para almacenar la escena. Parte I.

```
typedef struct {
    ObjetoEscena* objetos;
    int cant_objetos;
    Camara camara;
    Triangulo plano_de_vista;
    Luz* luces;
    int cant_luces;
    UniformGrid grilla;
    Material* materiales;
    int cant_materiales;
} Escena;

typedef struct {
    float3 v1;
    float3 v2;
    float3 v3;
} Triangulo;

typedef struct {
    TipoObjeto tipo;
    Triangulo tri;
    Triangulo normales;
    int id_material;
} ObjetoEscena;

typedef enum {
    Triangle,
    Sphere
} TipoObjeto;
```

Algoritmo 9 Estructura de datos para almacenar la escena. Parte II.

```
typedef struct {
    float3 ojo;
    float3 direccion;
    float3 up;
} Camara;

typedef struct {
    float3 posicion;
    float3 color;
} Luz;

typedef struct {
    float3 dimension;
    BoundingBox bbEscena;
    int* voxels;
    int* listasGrid;
} UniformGrid;

typedef struct {
    float3 diffuse_color;
    float3 ambient_color;
    float3 specular_color;
    float refraction;
    float reflection;
    float transparency;
    int coef_at_especular;
} Material;

typedef struct {
    float3 minimum;
    float3 maximum;
} BoundingBox;
```

A.2. Versión 2

Algoritmo 10 Estructura de datos para almacenar la escena. Parte I.

```
typedef struct {
    Triangulo* triangulos;
    Triangulo* normales;
    int cant_objetos;
    Camara camara;
    Triangulo plano_de_vista;
    Luz* luces;
    int cant_luces;
    UniformGrid grilla;
    Material* materiales;
    int cant_materiales;
} Escena;

typedef struct {
    float4 v1;
    float4 v2;
    float4 v3;
} Triangulo;
```

Algoritmo 11 Estructura de datos para almacenar la escena. Parte II.

```
typedef struct {
    float3 ojo;
    float3 direccion;
    float3 up;
} Camara;

typedef struct {
    float4 posicion;
    float4 color;
} Luz;

typedef struct {
    float3 dimension;
    BoundingBox bbEscena;
    int* voxels;
    int* listasGrid;
} UniformGrid;

typedef struct {
    float4 diffuse_color;
    float4 ambient_color;
    float4 specular_color;
    float4 others;
} Material;

typedef struct {
    float3 minimum;
    float3 maximum;
} BoundingBox;
```

Apéndice B

Algoritmos de Recorrida

Este apéndice contiene la especificación de los algoritmos descriptos en el Capítulo 3 agregando información que puede ser útil para el lector. Los algoritmos y técnicas de este anexo no fueron implementados en el presente trabajo.

B.1. Particularidades del KD-Tree

La dificultad de construir esta estructura dado un volumen a dividir radica en escoger el lugar donde colocar el plano de corte. Thrane y Ole [28] usan una función de costo para evaluar en donde se coloca el plano. Esta se basa en que la probabilidad de que un rayo atraviese un nodo hijo es proporcional a la proporción entre el área de la superficie del nodo hijo y el área de la superficie del nodo padre. Luego de algunos refinamientos la función pasa a tener en cuenta también, lo que sucede cuando un objeto es cortado por el plano de corte. Esto es importante porque los objetos cortados se propagan a través de los dos volúmenes hijos y por lo tanto se incurre en un alto costo de procesamiento. La forma de escoger una posición para el plano de corte es evaluar la función de costo a lo largo de todos los ejes de las cajas que envuelven a los objetos de la escena. La posición con menos costo según la función es elegida para posicionar el plano. A modo de ejemplo se puede considerar la Figura B.1 donde se muestra el procedimiento para un solo eje, en el procedimiento real se deben considerar también los dos restantes [28].

En la Figura B.1 la función de costo es evaluada en los puntos $a, b, \ldots, j$. Estos puntos se corresponden con los puntos iniciales y finales de los intervalos

que definen las cajas que envuelven a los objetos de la escena y el eje que será cortado. Vale la pena resaltar el intervalo $[c, f]$, que es generado por un objeto cortado por un plano, y es formado a través de un recorte generado por el *voxel* actual.

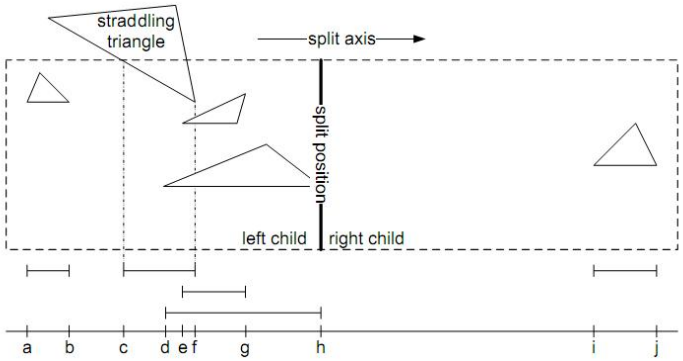


Figura B.1: Ejemplo de búsqueda del plano de corte considerando solo un eje.

En el caso que un objeto no esté completamente contenido en el *voxel* que se esta analizando para dividir, como sucede en el ejemplo de la Figura B.1, no se debe usar la caja que lo envuelve totalmente. En este caso se debe cortar el objeto y usar solo la parte de este que queda contenida en el *voxel* que queremos dividir. De esta forma solo se considera la caja que envuelve totalmente a esta nueva parte. Esta técnica es denominada "*split clipping*" [33]. Aunque se considere el corte con respecto al *voxel* para obtener un nuevo volumen envolvente, el objeto que es transferido hacia los hijos del nodo actual es el objeto original y no el corte. Thrane y Ole [28] usan esta técnica para la construcción de la estructura kd-tree y obtuvieron muy buenos resultados.

En el Algoritmo 12 se puede ver un pseudo-código de la construcción de esta estructura. El algoritmo es recursivo, tiene como entrada un *voxel* y como salida una estructura kd-tree. En cada paso de la recursión se divide al *voxel* de entrada (nodo padre) en dos sub-*voxeles* (nodos hijos). Para construir una estructura que permita lograr una buena aceleración, el algoritmo busca el mejor plano de corte utilizando el método que ilustra la Figura B.1. Cada objeto perteneciente a la lista de objetos del nodo padre es agregado a la lista de objetos del nodo hijo que lo contiene total o parcialmente. Luego,

para cada nodo hijo se invoca el algoritmo recursivamente.

El paso base de la recursión se da cuando se cumple algún criterio de parada. El algoritmo usa dos criterios de parada; cuando el número de objetos del *voxel* de entrada es menor a cierto valor predefinido y cuando la profundidad de la recursión alcanza cierto valor predefinido.

Para construir una estructura kd-tree se invoca el algoritmo de construcción con un *voxel* que contenga toda la escena. El algoritmo construye la estructura a partir de este *voxel*, dividiéndolo hasta que se cumplan los criterios de parada para cada una de las ramas del árbol de recursión.

Dado un nodo N de un kd-tree, el algoritmo para moverse a lo largo del sub-árbol con raíz N debe seguir los siguientes pasos:

- Si N es un nodo hoja, todos los objetos de N se prueban para ver si tienen intersección con el rayo. En caso de que existan intersecciones, se retorna la más cercana al observador.

- Si N es un nodo interno, es decir un nodo que esta dividido en dos y que tiene dos hijos, se debe determinar cual hijo de N es atravesado primero por el rayo. Luego, se llama de forma recursiva con este nodo. Si esta llamada encuentra intersección, será la más cercana al punto del observador entonces, se retorna. En caso contrario, se debe llamar de forma recursiva con el otro nodo hijo de N.

Utilizando el algoritmo para moverse en el árbol, siempre se visitan los *voxeles* en el orden en que son visitados por el rayo. Esto permite que se pueda parar el algoritmo de recorrida de *voxeles* tan pronto como se encuentre la intersección rayo-objeto más cercana al observador.

Algoritmo 12 Construcción de la estructura KD-Tree. Pseudo-código de la función *construir*(*voxel*).

```
si numObjetos(voxel) ≤ MIN_OBJETOS entonces
    se retorna nueva hoja con su lista de objetos
fin si
si profundidad(arbol) ≥ PROFUNDIDAD_MAX entonces
    se retorna nueva hoja con su lista de objetos
fin si
mejorCorte ← ∅
mejorCosto ← ∞
para todo eje in {x, y, z} hacer
    posiciones ← []
    para todo objeto en voxel hacer
        recortar objeto segun voxel
        calcular caja envolvente del objeto recortado
        encontrar puntos extremos a y b segun eje
        agregar a y b a la lista posiciones
    fin para
    para todo punto p en posiciones hacer
        si costo(p) < mejorCorte entonces
            mejorCorte ← (p, eje)
            mejorCosto ← costo(p)
        fin si
    fin para
fin para
(voxelIzq, voxelDer) ← dividir voxel segun mejorCorte
para todo objeto o en voxel hacer
    si interseccion(o, voxelIzq) entonces
        agregar o a voxelIzq
    fin si
    si interseccion(o, voxelDer) entonces
        agregar o a voxelDer
    fin si
fin para
hijoIzq ← construir(voxelIzq)
hijoDer ← construir(voxelDer)
se retorna nuevoNodoInterno(hijoIzq, hijoDer, mejorCorte)
```

Paralelismo en GPU

El primer problema que surge al querer paralelizar el algoritmo de atravesado de *voxeles* en una GPU es que este es recursivo. Esto es un problema porque en la GPU no se cuenta con una pila.

Una solución es considerar a la estructura kd-tree como un caso especial de la estructura de jerarquía de volúmenes envolventes (BVH, Bounding Volume Hierarchy) y usar su algoritmo de recorrida. Esto no es bueno porque se pierde la capacidad de recorrer los *voxeles* en el orden que son visitados por el rayo, y por lo tanto se pierde la capacidad de detener el algoritmo tan pronto como se encuentre una intersección. Además, como una estructura kd-tree es usualmente más grande que su correspondiente BVH para la misma escena, se estaría creando un BVH ineficiente [28]. Una posible solución es emplear una estrategia diferente de recorrido de los *voxeles*. Se puede emplear la estrategia usada por Foley y Sugerman [32], en la cual se cambia el algoritmo recursivo por uno secuencial. Esta estrategia consiste en mover un intervalo $[t_{min}, t_{max}]$ a lo largo del rayo e ir descendiendo desde la raíz del árbol hasta que una hoja que contenga al intervalo sea encontrada. Inicialmente, el intervalo abarca todos los valores de t tal que el punto $o + tv$ esta contenido en la caja del nivel superior del árbol, es decir la caja que contiene a toda la escena (o es el origen del rayo y v es la dirección del rayo). Para cada nivel del árbol que se desciende, se le asigna a t_{max} el mínimo entre t_{max} y t_{split}, donde t_{split} es la distancia a lo largo del rayo desde t_{min} hasta el plano de corte del nodo actual. Cuando se llega a un nodo hoja, el intervalo es el rango paramétrico en el cual el rayo se encuentra dentro del *voxel* determinado por el nodo.

Si en un nodo hoja se encuentra intersección rayo-objeto, se debe retornar; por como lleva a cabo la recorrida el algoritmo, está garantizado que esta será la intersección más cercana al origen del rayo. En caso contrario, se debe actualizar el intervalo para continuar con la recorrida. El nuevo intervalo comienza en el fin del *voxel* actual y finaliza en el fin de la caja que contiene a la escena completa, como se muestra en el ejemplo de la Figura B.2. En la parte (a) del ejemplo luego de que fallan todas las intersecciones en un nodo hoja, el nuevo intervalo es construido con el punto de fin del *voxel* actual y el punto de fin de la caja que envuelve a toda la escena. De esta manera, se llega a la siguiente hoja comenzando nuevamente el algoritmo de recorrida.

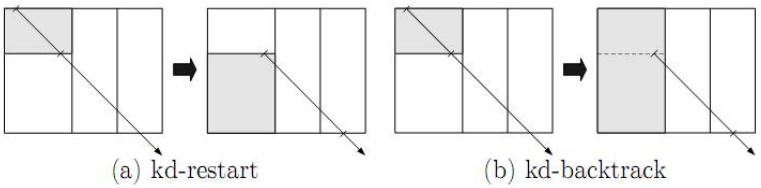


Figura B.2: Actualización del intervalo en cada variante del algoritmo de atravesado.

Foley y Sugerman presentan su algoritmo en dos variantes *restart* y *backtrack*. La variante *restart* usa el enfoque de Glassner, en el cual se comienza desde la raíz del árbol cada vez que se avanza un *voxel* en la recorrida de los mismos [3]. Esta técnica, en general, presenta un tiempo de ejecución alto y en este sentido es peor que el algoritmo recursivo. Para remediar esto la técnica de *backtrack* modifica la de *restart* permitiendo moverse hacia arriba en el árbol en vez de moverse hacia la raíz cada vez. Cuando en un nodo hoja fallan todos los intentos por encontrar una intersección, hay que moverse hacia arriba en la estructura de árbol hasta que se encuentre un *voxel* ancestro que tenga intersección con el nuevo intervalo. En la Figura B.2 parte (b) se muestra dicho ancestro marcado en color en la estructura de más a la derecha. Foley y Sugerman reportan una pequeña mejora utilizando esta técnica pero también afirman que se incurre en un grado más alto de complejidad en la implementación.

En el trabajo de Horn et al. [6] se presentan optimizaciones que pueden realizarse sobre el algoritmo *restart*. Los autores señalan que en las pruebas realizadas, las optimizaciones propuestas permitieron llegar a un algoritmo de *ray tracing* con la capacidad de generar de 12 a 18 frames por segundo. Dado esto las consideran como buenas optimizaciones. Se llevaron a cabo tres optimizaciones sobre el algoritmo de Foley y Sugerman en su versión *restart*:

- Paquetes de rayos: esta optimización se basa en la idea de paquetes de rayos para CPUs, descrita por Wald [22]. Wald buscó la manera de sacar provecho de las instrucciones SIMD de las CPUs modernas agrupando los rayos en paquetes. El tamaño de los paquetes queda determinado por la cantidad de datos que tengan como entrada las instrucciones. La mejora que introduce esta optimización es que todos los rayos de un mismo paquete se trazan en paralelo. Esta misma idea se puede llevar

a una GPU, donde la cantidad de rayos por paquete dependerá de las características de la misma.

- *Push-Down*: esta optimización busca no recomenzar siempre desde la raíz del árbol. Por ejemplo, a menudo un rayo atraviesa el volumen que contiene a la escena y solo pasa a través de un sub-árbol de la estructura kd-tree. Si se arranca el algoritmo siempre desde la raíz se esta analizando muchas veces un sub-árbol que ya se sabe que no es atravesado por el rayo. Esta técnica permite recomenzar el algoritmo desde el sub-árbol más profundo que encierra al rayo, y de esta manera no se vuelven a analizar sub-árboles que no lo contienen.

- *Short-Stack*: Horn et al. [6] observaron que era posible usar una pila que guarde los últimos N nodos visitados (tamaño fijo) y pasarse al algoritmo sin pila cuando esta se desborda. Dado esto, introdujeron como forma de optimizar el algoritmo una pequeña pila de tamaño fijo, cuya manipulación puede tomar dos caminos. Cuando se introduce un nuevo nodo y la pila esta llena, se descarta el nodo que se encuentra más alejado del tope. Cuando se saca un nodo de la pila vacía el algoritmo no termina, vuelve a comenzar desde la raíz del árbol. Esta pila es como un *caché* de nodos y puede ser usado para disminuir la frecuencia de recomienzos a costa de sacrificar el tiempo de procesamiento de un rayo.

B.2. Particularidades del BVH

Antes de presentar las ideas para construir una estructura BVH es importante considerar que en la práctica, los volúmenes más usados para construir una BVH son los volúmenes envolventes alineados a los ejes de coordenadas (AABB). Los AABB pierden rendimiento porque no se ajustan perfectamente a los objetos, pero lo ganan por el lado de permitir un chequeo de intersección simple y rápido. También son muy buenas estructuras en términos de simplicidad de implementación [28]. Los dos enfoques de construcción que se presentan a continuación utilizan este tipo de volumen envolvente.

Kay y Kajiya sugieren un enfoque recursivo top-down. Esta idea se muestra aplicada al algoritmo de construcción en el Algoritmo 13. Para construir una estructura BVH el algoritmo necesita como entrada la lista de objetos que conforman la escena. La salida del algoritmo es una jerarquía de

volúmenes envolventes alineados con los ejes, como la que se muestra en la Figura 3.6. Si la escena tiene un solo objeto, la estructura se construye con un solo volumen envolvente. En caso contrario, se busca el mejor eje de corte y la mejor posición de corte. Se pueden usar diversas estrategias para encontrar el plano de corte, una de ellas es usar la función de costo que se muestra en la Ecuación 3.1. Se puede adoptar otra estrategia como cortar siempre por el punto medio o como dejar la misma cantidad de objetos de cada lado del plano. Por último, se construyen los sub-árboles izquierdo y derecho de forma recursiva, así como también se construye el volumen envolvente que contiene a todos los objetos.

Algoritmo 13 Construcción de la estructura BVH según Kay y Kajiya [26]. Pseudo-código de la función $consArbol(objetos)$

```
BVNODE res
si cantidad(objetos) == 1 entonces
    res.hijoIzq ← arbolVacio
    res.hijoDer ← arbolVacio
    res.volEnvolvente ← volumen que contiene a todo o ∈ objetos
si no
    Calcular el mejor eje de corte y por donde se debe cortar
    res.hijoIzq ← consArbol(objetos del lado izquierdo del corte)
    res.hijoDer ← consArbol(objetos del lado derecho del corte)
    res.volEnvolvente ← volumen que contiene a todo o ∈ objetos
fin si
se retorna res
```

Goldsmith y Salmon proponen un enfoque de construcción bottom-up que resulta más complicado. El algoritmo comienza asignando el primer objeto de la escena como la raíz del árbol. Para cada objeto adicional en la escena, se busca la mejor posición en el árbol mediante la evaluación de una función de costo (por ejemplo, usando la Ecuación 3.1). La posición se busca mediante un recorrido recursivo descendente en el árbol, siguiendo el camino que resulte menos costoso según la función. Finalmente, el objeto es insertado de alguna manera: como una nueva hoja o se reemplaza una hoja existente por un nodo interno que contiene al nodo hoja viejo y al nuevo objeto como hijos. Como resultado de este enfoque un nodo interno puede tener un número arbitrario de hijos, contrariamente a lo que pasa con el enfoque de Kay y Kajiya, que produce árboles binarios.

Goldsmith y Salmon advierten que la calidad de la estructura BVH generada por su algoritmo depende fuertemente del orden de los objetos pasados como entrada. Como una solución, recomiendan distribuir aleatoriamente el orden de los objetos antes de construir la estructura.

La forma estándar de recorrer una estructura BVH es a través de una recursión. Para los nodos internos se debe probar la intersección del rayo contra el volumen envolvente asociado. Si se encuentra intersección, se debe probar la intersección recursivamente contra los nodos hijos. A diferencia de la estructura kd-tree, se deben visitar todos los nodos hijos, dado que estos se pueden solapar y no siguen ningún criterio de ordenación. Si el rayo no atraviesa el volumen envolvente del nodo no es necesario probar los nodos hijos. Para los nodos hoja, solo se debe probar si el rayo tiene intersección con alguno de los objetos de la escena asociados al nodo.

El principal problema que se encuentra cuando se quiere acceder a los volúmenes envolventes para probar su intersección con el rayo, es el orden en el cual los nodos hijos son accedidos. Kay y Kajiya proponen un método por el cual se intenta seleccionar el nodo más cercano al origen del rayo, siguiendo la dirección del mismo. Esta técnica es un poco complicada porque requiere por ejemplo, mantener una cola de prioridad, de donde se extraen los nodos a ser atravesados por el rayo. Thrane y Ole [28] concluyen que esta técnica no implica una ganancia de rendimiento considerable.

Paralelismo en GPU.

Para implementar en una GPU el algoritmo que atraviesa una estructura BVH dado un rayo hay que resolver dos problemas. El primero corresponde a encontrar un método para atravesar la estructura de árbol eficientemente sin contar con una pila. El segundo problema es encontrar una representación adecuada de la estructura BVH para usar sobre la GPU.

La representación de la estructura y el algoritmo que atraviesa el árbol son independientes. La solución propuesta por Thrane y Ole se basa en una cuidadosa elección de los datos que se deben guardar dados los tipos de almacenamiento que provee la GPU. En la GPU se debe guardar el estado del recorrido en vez de guardar el árbol en si mismo. La idea para llevar a cabo esto proviene de observar que los rayos atraviesan los nodos del árbol siempre en *pre-order*. Una recorrida de un árbol es en *pre-order* cuando se recorre primero el nodo raíz, luego el sub-árbol izquierdo y por último el sub-árbol derecho, como se muestra en el ejemplo de la Figura B.3.Los nodos

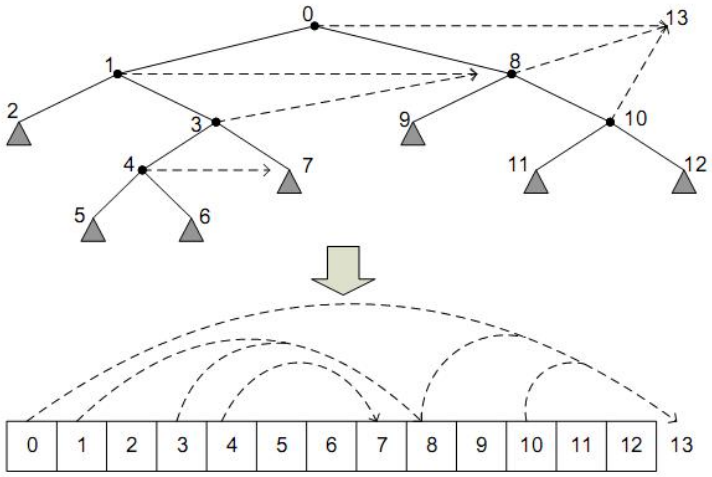


Figura B.3: Ejemplo de codificación de los datos para atravesar una estructura BVH.

del árbol son numerados secuencialmente de acuerdo al orden mencionado anteriormente. Esta numeración coincide con la forma en que son guardados los datos de los nodos en la estructura que maneja la GPU, un array.

Una línea punteada ($a \dashrightarrow b$) representa la situación en la que el rayo no atraviesa al volumen a y se debe seguir probando con los demás nodos hermanos, en este caso el volumen b. Como se muestra en la Figura B.3, cada línea punteada es guardada como un par de índices, donde cada componente del par hace referencia al array. Thrane y Ole [28] llaman a este puntero índice de escape. En el ejemplo de la Figura B.3, si un rayo no atraviesa el volumen número 1 se debe seguir probando con los volúmenes hermanos para ver si el rayo atraviesa a alguno. Para pasar del volumen número 1 a su próximo hermano (volumen número 8) sin tener que recorrer todo el sub-árbol izquierdo se usa el índice de escape ($1 \dashrightarrow 8$). A través del índice de escape se navega el árbol de forma eficiente. Se puede ver que todos los nodos hoja tienen un índice de escape relativo igual a 1. Como consecuencia de esto, no se necesita guardar a la vez el índice de escape y los objetos que contiene el volumen. Notar la convención indirecta en la Figura B.3 donde los nodos internos del sub-árbol derecho tienen índice de escape igual al número total de nodos del árbol. Esto se usa para tener un criterio de parada en el algoritmo de recorrida.

Un algoritmo para atravesar una estructura BVH dado un rayo que siga este enfoque es simple e iterativo, lo cual es muy bueno para ejecutarlo en una GPU. El algoritmo requerido para ejecutar la recorrida de los volúmenes envolventes de una BVH, guardados en la estructura de array, es mostrado en el Algoritmo 14. La iteración siempre termina con un índice actual mayor al que existía cuando se inició, esto se da como consecuencia de que los índices de escape siempre van en la dirección de aumento.

Algoritmo 14 Recorrida de los volumenes envolventes para GPU.

```
S ← secuencia de recorrida
r ← el rayo
indiceActual ← 0
mientras indiceActual < largo(S) hacer
    nodoActual ← S[indiceActual]
    si hayInterseccion(r, nodoActual) entonces
        indiceActual ← indiceActual + 1
        guardar datos interseccion si nodoActual es hoja
    si no
        indiceActual ← indiceEscape(nodoActual)
    fin si
fin mientras
```

Apéndice C

Articulo presentado en JCC 2010

Improving the Performance of the Ray Tracing Algorithm with a GPU

Santiago Cioli*, Gonzalo Ordeix†, Eduardo Fernández‡, Martín Pedemonte§, Pablo Ezzatti¶
Instituto de Computación, Facultad de Ingeniería,
Universidad de la República
Montevideo, Uruguay
*Email: *sancioli@gmail.com, †gonzalo.ordeix@gmail.com*
‡eduardof@fing.edu.uy, §mpedemon@fing.edu.uy, ¶pezzatti@fing.edu.uy

***Abstract*—This article presents the application of parallel computing techniques using a Graphic Processing Unit (GPU) in order to improve the computational efficiency of the ray tracing algorithm. In particular, three different GPU implementations of the ray tracing algorithm are presented. The experimental evaluation of the proposed methods demonstrates that a significant reduction on the computing time can be attained when compared with a full CPU implementation, making a step forward to the calculation of scene brightness in real time on desktop computers.**

***Keywords*-Ray tracing; GPU; Real-time**

I. Introduction

A scene is a collection of objects and light sources that are seen through a camera. Each object in a scene is a geometric primitive, which is usually a simple geometric shape like a polygon, a sphere or a bicubic surface (Hermite, Bézier, Spline). Additionally, the surface of the object has material properties, textures, etc. All global illumination techniques try to solve the same problem consisting in, given a scene, finding a set of images as photorealistic as possible. These algorithms usually differ in how they handle the lighting of the scene.

Several kind of global illumination algorithms can be identified, based on the different light elements considered. Radiosity [11], *ray tracing* [25] and multipass methods (like RADIANCE [24] and photon mapping [13]) produce realistic images in diverse scenarios with several kind of surfaces. Radiosity works well in scenes with Lambertian surfaces, *ray tracing* produces good images in scenes with specular surfaces and the multipass methods are more versatile based on the mixture of methodologies used in them. Developing an algorithm that generates at least twenty images per second -that is, in real time- is a great challenge for the computer graphics research community.

Ray tracing algorithm calculates the brightness of each pixel of an image by throwing rays, and evaluating their bounces in the different surfaces of the image. Each bounce produce new rays that impact in other objects and so on. The color of the pixel is composed with the color of the first object considered plus the color added with each bounce. This algorithm was the first step into photorealistic rendering, and its success is due to its ability to generate good quality images with an algorithm easy to implement.

Graphics Processing Units (GPUs) are devices designed originally for graphics processing only, lightening the workload on the CPU in applications such as video games. Thus, the CPU can be used to perform other computations while most of the graphic processing calculations are performed on the graphic device. GPUs are currently very powerful platforms, provided by tens or hundreds of cores with acceptable clock frequencies (500-600MHz). Additionally, the computing power of GPUs is enhanced due to its intrinsically parallel architecture.

Initially, the progress in the design of GPUs was not accompanied by an advance in the software for programming these devices until 2006, when NVIDIA released CUDA (Compute Unified Device Architecture) [7]. CUDA is an architecture for general purpose parallel computing that allows using the parallel processing core in these devices to solve a wide variety of computational problems more efficiently than a CPU. The *ray tracing* algorithm is highly parallelizable since the calculation of lighting in each pixel is an independent process, and therefore is very suitable for GPUs.

This article studies a *ray tracing* algorithm implemented in GPU for speeding up the computation time. Three parallel versions were developed, in order to exploit different characteristics of the *ray tracing* algorithm and GPU architecture. An analysis of the performance was conducted, measuring the number of frames that could be calculated per second. The preliminary results show that large improvements could be obtained (up to $13\times$) when using a GPU cheaper than a standard multicore computer, such as the ones used in this analysis.

The content of the article is structured as follows. The next section describes the main features of modern GPUs and CUDA architecture. Then, Section III introduces the *ray tracing* algorithm. Section IV describes the three different GPU implementations of *ray tracing* presented in this article. The experimental evaluation of the proposed methods is reported in Section V, where the results are also analyzed. Finally, Section VI presents the conclusions of this research and formulates the main lines for future work.

II. Graphic processing units

Based on the facilities provided by CUDA [7] for GPU programming, GPUs can be viewed as a set of shared memory multicore processors. GPUs are usually considered *many-cores* processors due to the large number of small cores that contain. GPUs follow the single-program multiple-data (SPMD) parallel programming paradigm in which cores execute the same program on multiple parts of the data, but do not have to be executing the same instruction at the same time [8]. The number of threads that currently graphics card can execute in parallel is in the order of hundreds and is expected to continue increasing rapidly, what makes these devices a powerful and low cost platform for implementing parallel algorithms.

CUDA [15] consists of a stack of software layers including: a hardware driver, a C language application programming interface and the CUDA driver that is dedicated to transfer data between the GPU and CPU. It is available for NVIDIA's GeForce 8 series of graphics cards and superiors. It is compatible with operative systems Linux 32/64 bit and Windows XP and superiors of 32/64 bits.

CUDA architecture is built around a scalable multiprocessor array. Each multiprocessor on current GPUs, consists of eight scalar processors as well as additional units like the multithreading instruction unit and the shared memory chip. When a part of an application could be run many times but independently on different data, it can be isolated in a function, called kernel function, to be executed on the device through many different threads. For this purpose, the kernel function is compiled using the device instruction set and the resulting program is transferred to the device.

When a kernel function is called, a large number of threads are generated on the GPU. The group of all the threads generated by a kernel invocation is called a grid, which is partitioned in many blocks. The blocks group threads that are executed concurrently on a single multiprocessor of the card. There is no fixed order of execution between blocks. If there are enough multiprocessors available on the card, they are executed in parallel. Otherwise, a time-sharing strategy is used.

Threads can access data across multiple memory spaces during their execution. Nowadays, GPUs based on G80 architecture have six different memory spaces: registers, local memory, shared block memory, global memory, constant memory and texture memory. Table I presents the main features of the different GPU memory spaces that are briefly commented next.

Registers, that are located in the chip, are the fastest memory on the card and are only accessible by each thread. In addition to this, each thread has its own local memory but is one of the slowest memories on the card, because it is located in the device memory and is not cached. Both memory spaces are entirely managed by the compiler. Each block has a shared memory space that is almost as fast as registers and could be accessed by any thread of the block. The shared memory is located on the chip and its lifetime is equal to the lifetime of the block.

Table I
Features of the different GPU memory spaces.

Memory	Scope	Lifetime	Size	Speed
Registers	Thread	Kernel	Very small	Very fast
Local	Thread	Kernel	Small	Slow
Shared	Block	Kernel	Very small	Very fast
Global	Grid	Application	Big	Slow
Constant	Grid	Application	Very small	Fast
Texture	Grid	Application	Very small	Fast

All the threads on the GPU have access to the same global memory on the card. The global memory is one of the slowest memories on the card and is not cached. On the other hand, constant memory is fast although for the device is read-only. It is located in the device memory even though it is cached. In fact, constant memory can be seen more as a cache of the global memory than a different memory space. Finally, the texture memory has the same characteristics that constant memory.

Figure 1 presents the CUDA architecture diagram, including the six different memory spaces. The figure shows local memory close to the threads as local memory is private to each thread. However, local memory is really located in the device memory as the global memory.

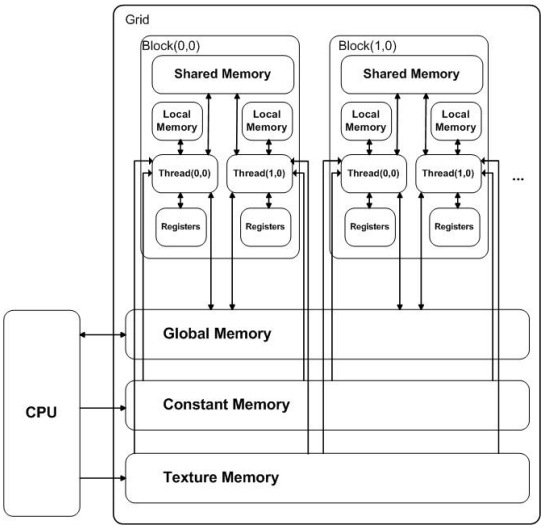


Figure 1. CUDA memory model.

III. Ray tracing algorithm

The ray casting algorithm [3] was proposed by A. Appel and is based in tracing rays from the observer's viewpoint to a view plane between the observer and the scene. The *ray tracing* algorithm [25] extends the idea of ray casting

by making the ray tracing process recursive when the ray intersects an object in its way.

Ray tracing achieves a great realism in the images generated even though its implementation is quite simple. However, the simplifications used in the lighting model do not allow generating caustics caused by light rays reflected or refracted by curved surfaces. Similarly, the calculation of the color component of "ambient light" [14], that is a simplification in the lighting calculation, makes the algorithm unable to produce some effects like "color bleeding" (phenomenon caused by light reflection making the color of a wall spread over the floor area near the wall).

The *ray tracing* algorithm works as follows. For each pixel of the image, a ray is traced from the observer's viewpoint to the pixel (called primary ray). If a ray does not intersect an object in its way, then the pixel is painted with the background color of the scene. On the contrary, if the ray intersects with an object, the effect of the shadows, refraction and reflection are calculated. Figure 2 shows the rays generation of *ray tracing* in a scene with a single light source from a single primary ray.

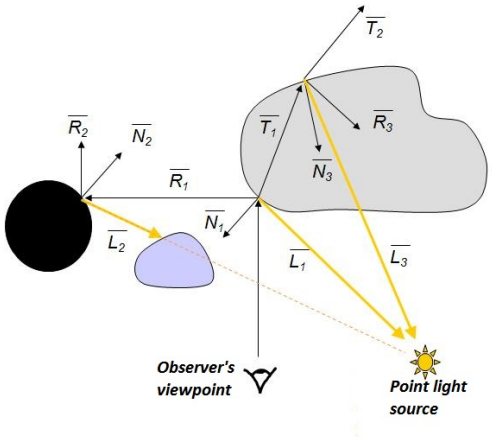


Figure 2. Ray generation from a single primary ray.

To calculate the shadows, a "shadow" ray ($\overline{L_1}$) is traced from the intersection point of the primary ray to each existing light source on the scene. If any of these rays intersects with any object in the scene, the amount of light that passes through the object is calculated, depending on the material of the object. If the object intersected is opaque (as the smallest object in the Figure 2), the intersection point of the primary ray is under the shadow of the object, so the light source is not considered to calculate the illumination at the point. If the object intersected is transparent (as the largest object in the Figure 2), the intensity of the light source is reduced, or may even be totally absorbed by the object. When the light is not completely blocked by the object, it contributes to the illumination of the intersection point of the primary ray.

When the object has specular reflection, a ray ($\overline{R_1}$, called reflection ray) is reflected from the primary ray at the point of intersection with respect to the normal ($\overline{N_1}$). This ray enables to get the intensity of the light that reaches the intersection point of the primary ray due to the phenomenon of reflection.

When the object is transparent and the light is not totally absorbed by the transparency of the object, a ray ($\overline{T_1}$, called refraction ray) is traced through the object. This ray enables to get the intensity of the light that reaches the intersection point of the primary ray due to the phenomenon of refraction.

Each one of the reflection and refraction rays when intersects with an object, can generate new "shadow", reflection and/or refraction rays. Therefore, the steps for the calculating the effect of refraction and reflection should be calculated recursively. For example, the same steps used to calculate the intensity of the light provided by the primary ray should be used to calculate the intensity of the light provided by $\overline{R_1}$. Thus, a ray tree is build for each primary ray, as shown in Figure 3.

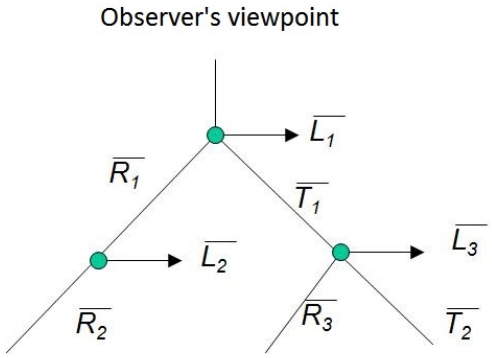


Figure 3. Ray tree resulting from the Figure 2.

A. Spatial acceleration structures: uniform spatial subdivision

Two families of strategies could be used to improve the performance of *ray tracing* algorithm; one which reduces the number of rays and another that optimizes the number of intersection checks performed. Space division of the scene by means of spatial acceleration structures helps to reduce the number of intersection checks, since it guarantees that the entire list of objects of the scene should not be checked for each ray.

In the space division method, the scene is divided into regions. Each region has a list of all the objects that it contains, either in whole or in part. This technique requires a preprocessing stage to create a data structure in which the position and space occupied by each object in the scene are stored. In the preprocessing stage, the total volume of the

scene is divided in small volumes or boxes. The criteria used to define the boxes is what distinguishes the different spatial subdivision techniques, having all the techniques exactly the same behavior.

The main advantage of the space division method is that only the objects belonging to boxes traversed by the ray should be checked for possible intersections. As a result, and depending on the object distribution in the scene, it avoids doing many unnecessary calculations.

When the scene is divided in boxes of the same size, the technique is called uniform spatial subdivision. This strategy of space division can effectively accelerate the calculation of the intersections despite of being simple. While there are other alternatives like the kd-tree [23], the uniform spatial subdivision improves the performance of *ray tracing*, it is easy to implement and it does not add extra issues to the algorithm; therefore, it is a spatial acceleration structures interesting to be included in the implementation of *ray tracing* in a GPU.

B. Related work

The *ray tracing* algorithm has a high computational cost especially in the models used in most 3D applications, based on the rasterization of images formed by polygons. For this reason, *ray tracing* technique, until recent times, was not suitable for real-time applications. However, nowadays, some new works have achieved real-time *ray tracing* implementations over CPU architectures, such as a Quake Wars game engine [21] implemented using openRT [18]. A demo of the engine showed in August 2008 runs approximately between 20 - 35 FPS with an image resolution of 1024 by 720 pixels on a Caneland system that has four Dunnington with six cores.

On the other hand, in recent years, applying the computational resources delivered by modern GPUs to *ray tracing* has resulted in a number of implementations that allow rendering scenes in reasonable times. Researchers have introduced many techniques to speed up the construction of acceleration structures and the traversal of rays through an acceleration structure. Works, such as those done by Horn et al. [12], Popov et al. [20], Parket et al. [19], Aila and Laine [1] and the 3D engine developed by researchers of the Alexandra Institute [4].

The Horn et al. work is based on the use of Boundary Volume Hierarchy (BVH), such technique is not cover in our work. Meanwhile, the proposal of Aila and Lane is implemented using a combination of Brook [6] and Direct3D [9], involving a different conceptual abstraction of the GPU model. Parket et al. [19] have proposed a general framework to develop *ray tracing* algorithms, but their work is more focused on developing a flexible and adaptable framework than on the performance of the resulting algorithm. For these reasons, none of the three previous works were considered for the development of our proposal. On the other hand, the algorithm implemented by the *Alexandra Institute* is based on the traditional *ray tracing* algorithm whereas in the work of Popov et al. the kd-tree spatial acceleration structure is used, therefore both works are closely related with our approach.

A good survey of the state of the art in the *ray tracing* techniques can be found in the works of Parker et al. [19], and McGuire and Luebke [16].

IV. Our Proposal

The *ray tracing* algorithm is inherently suitable for parallelization with SPMD techniques, since the calculation of lighting in each pixel is an independent process. This feature makes possible its semi-direct implementation on graphics cards, using a separate thread to calculate each ray. Three different versions of the *ray tracing* algorithm were developed in order to exploit different characteristics of the algorithm and the GPU architecture. The versions were implemented following an incremental approach, incorporating in each version a considerable improvement over the previous one.

This section describes the main characteristics of the different versions implemented. First, a description of some general features of all versions implemented is introduced. Later, the differences between all versions implemented are detailed.

All versions were implemented using the C language and CUDA (version 2.3) to manage the GPU. The general structure of the different versions of the *ray tracing* implemented is presented in the Figure 4. It has five different steps that are discussed below.

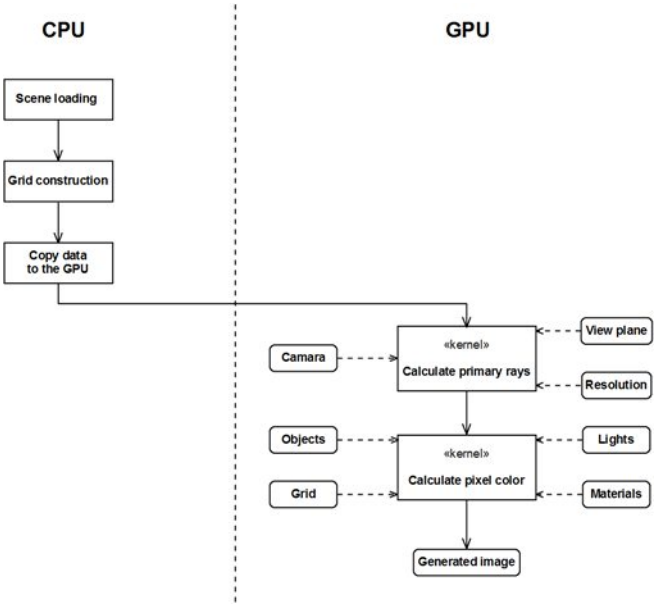


Figure 4. Structure of the *ray tracing* algorithm implemented in CUDA.

In the first step of the algorithm, the data of the scene is loaded from a text file. The file format is based on Wavefront OBJ (version 3.0) [17] that has been adopted by most 3D graphics application vendors. It makes possible to define the elements of the scene (e.g. vertices, points, lines, polygons, curves, etc.) and the materials of the elements. Also at this stage, a configuration file is loaded, containing parameters required to execute such as image resolution, division size for the acceleration grid, maximum number of ray bounces, etc.

In the second step, a spatial acceleration structure corresponding to the uniform space subdivision it is built, because of the simplicity of its construction and its traversal. The grid construction algorithm has been optimized, so that for each object in the scene, it is calculated (grid coordinates) an axis-aligned box that surrounds it. From the box, candidate boxes that may overlap with the object are obtained. For each candidate box, it is tested the box-object overlapping, and if it happens, the object is added to the box.

The third stage involves the transfer of data from the memory space of the CPU to the GPU. The transferred data are: the view plane, the camera, the image resolution, the list of triangles and its normals, the light sources, the boxes of the grid for spatial subdivision and the material of the objects.

After copying the data to the GPU, the kernel that calculates the primary rays is invoked. The data required to calculate the primary rays are the view plane, the camera and the image resolution.

The primary rays are input to the kernel which calculates the color of each pixel. This kernel implements the core of the *ray tracing* algorithm. The data required to calculate the color of each pixel are the list of triangles and its normals, the light sources, the boxes of the grid for spatial subdivision and the material of the objects.

The kernel that calculates the color of each pixel is invoked following a division in patches (group of pixels) of the image to render. The image is divided uniformly, having each patch the same number of pixels. Each patch corresponds directly to a block of threads in CUDA in order to process each division of the image by a different block. Moreover, since each pixel of the image is processed by a different thread, the number of threads per block is equal to the number of pixels contained in each division. For this reason, the division is completely established when fixing the number of threads per block and the image resolution. For example, if the image resolution is 640×480 pixels and the block size is 16×8 threads, the image must necessarily be divided into 40×60 patches.

Each ray threw traverse the spatial acceleration structure, following the reflections and refractions in the objects. The *ray tracing* algorithm implemented only has one type of elements, the triangle, therefore just a method of ray-triangle intersection was required. As it only works with a basic element, the intersection algorithm is simple and requires only a few arithmetic operations to test the intersection with a ray.

An important aspect is that the *ray tracing* algorithm is recursive, but current GPUs do not support recursion. As a consequence, the algorithm has to be implemented iteratively. There are two alternatives to achieve this, implementing a stack to store the recursion tree or simplifying the tree by making it degenerate into a list. The first alternative was ruled out because each thread must have its own stack and the size of the local memory is very limited. To degenerate the recursion tree into a list, it requires as a precondition that the scene has no objects that reflect and transmit light at the same time. This was the alternative implemented, since the limitation imposed on the scene is acceptable.

Finally, after the execution of the kernel that calculates the color of each pixel, the data generated in the GPU is copied to the CPU to be displayed on the screen.

The differences between the versions are discussed in the next subsections.

A. `RT(GPU)` *version*

The first version of *ray tracing* algorithm (`RT(GPU)`) is a GPU-analogue to the CPU implementation. In `RT(GPU)`, all the data is stored in global memory.

B. `RT(GPU-ml)` *version*

Regarding the GPU architecture, and in particular the importance of the correct use of the memory levels, this version (`RT(GPU-ml)`) uses the different memory levels of GPU accessible through CUDA. In particular, texture and constant memories are used, ensuring an improve in the performance.

The data that is most used by the algorithm, such as the list of triangles or the boxes of the grid for spatial subdivision, must be stored in a memory level with fast read access. For this reason, the list of triangles and its normals, the light sources, the boxes of the grid for spatial subdivision and the list of material of the objects are copied to the texture memory, since these data is accessed frequently and do not need updating. Other data such as the view plane, the camera and the image resolution is stored in the constant memory. Reading data from these types of memory is much faster than reading from global memory.

C. `RT(GPU-ii)` *version*

The third version (`RT(GPU-ii)`) improves the procedure for calculating the ray-triangle intersection using the barycentric coordinates method [2]. This method verifies that the ray intersects the plane containing the triangle and then by a change of coordinates verifies that the intersection point is within the limits of the triangle.

V. Experimental analysis

In this section, we present the test cases and hardware platforms used to evaluate the different versions implemented of the *ray tracing* algorithm. Then, we describe in detail the various experiments conducted to validate the proposal.

In addition to the GPU versions of the *ray tracing* algorithm described in Section IV, a CPU implementation of *ray tracing* `RT(CPU-ii)` was developed to evaluate the comparative performance versus the GPU versions.

A. Test cases

In a first instance, we studied the existing strategies for measuring the quality of the images generated. The survey did not obtain any comprehensive strategy. The choice of the method for measuring the quality of the image depends heavily on the objective of the study. Avcibas et al. [5] and Dirik et al. [10] present a good survey of strategies and discuss their limitations, but none is applicable for the purposes of this study.

In addition to this, there are no standardized test cases or benchmarks that could be used to evaluate the different implementations of the *ray tracing* implemented in this work. For this reason, a set of images were designed trying to cover several aspects of the image generation process, in order to contribute to measure different characteristics of the implemented algorithms. The test set of images designed is divided into three different groups. The first group consists of images that allow evaluating the effect of the distribution of objects in the scene. The test cases of the second group consists of images that allow evaluating the impact of the number of triangles in the scene. Although there are no benchmarks, some images have been sporadically used by the research community (such as the Bunny from Stanford University). Therefore, the third group includes some of those scenes and images that have been used in studies similar to this.

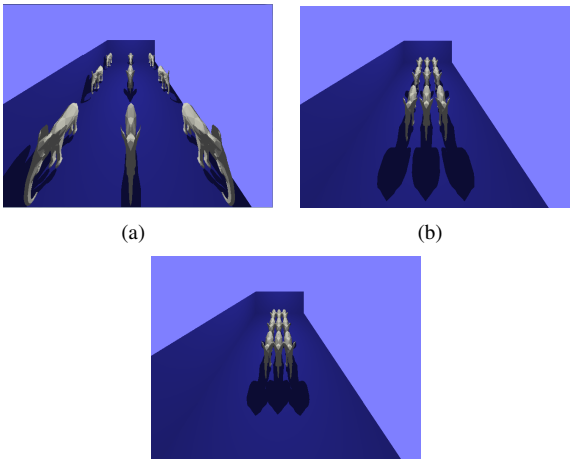
1) Test cases with different object distribution: All scenes have the same number of triangles but a different distribution in the scene. Table II presents the main features of the test cases considered.

2) Test cases with different number of primitives: All images in this group are exactly the same, but were discretized using a different number of primitives. Table III presents the main features of the test cases considered.

3) Test cases from similar studies: All images are taken from similar studies or are images commonly used by the research community. Table IV presents the main features of the test cases considered.

Table II
Test cases with different distribution in the scene.

Scene name	# Objects	# Lights	# Triangles	Figure
Dist_I	9	1	10,338	5(a)
Dist_II	9	1	10,338	5(b)
Dist_III	9	1	10,338	5(c)

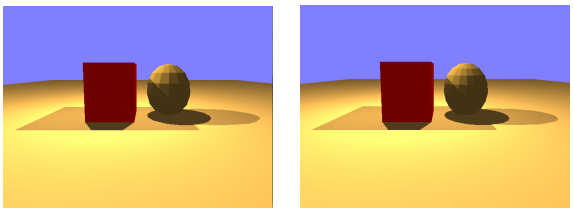


(a) (b) (c)

Figure 5. Scenes with different spatial distribution of the objects.

Table III
Test cases with different number of primitives.

Scene name	# Objects	# Lights	# Triangles	Figure
Pri_I	2	2	194	6(a)
Pri_II	2	2	274	N/S
Pri_III	2	2	348	N/S
Pri_IV	2	2	482	N/S
Pri_V	2	2	606	6(b)



(a) (b)

Figure 6. Scenes with a different number of primitives.

Table IV
Test cases used for comparing with other *ray tracing* implementations.

Scene name	# Objects	# Lights	# Triangles	Figure
Alexandra	14	1	236	7(a)
Buddha	1	1	100,000	7(b)
Dragon	1	1	100,000	7(c)
Bunny	1	1	69,698	7(d)

B. Hardware platform

Several hardware platforms were employed to evaluate the implemented algorithms. Each platform consists of a PC

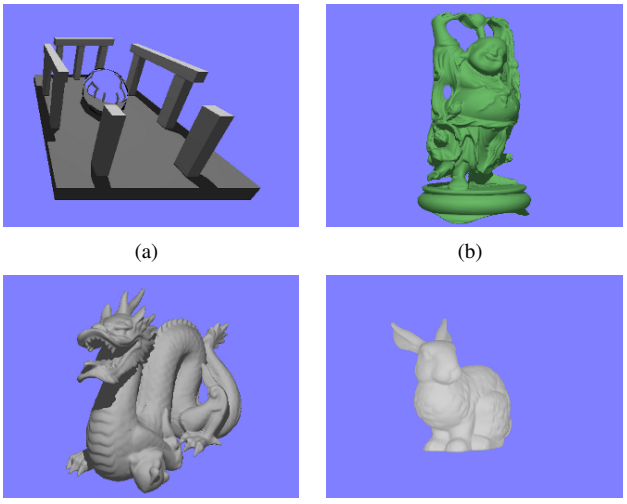


Figure 7. Scenes used for comparing with other *ray tracing* implementations.

Core 2 Duo with a GPU of the NVIDIA GeForce series. The main details of the hardware platforms used are presented in the Table V. All the PCs were running the Windows operating system.

Table V
HARDWARE PLATFORMS USED FOR EXPERIMENTAL ANALYSIS.

GPU	GPU memory	CPU	RAM memory
9500M GS	512 MB	T7500 2.20GHz	4GB DDR2 667 MHz
9600M GT	512 MB	P8400 2.26GHz	4GB DDR2 667 MHz
GTX 260	896 MB	E7500 2.93GHz	4GB DDR2 667 MHz

The Table VI provides more details of each one of the GPUs used during the evaluation.

Table VI
GPUS USED FOR EXPERIMENTAL ANALYSIS.

GPU	Multi processors	Cores	Clock (MHz)	Shader clock (MHz)	Memory clock (MHz)
9500M GS	4	32	475	950	400
9600M GT	4	32	500	1250	400
GTX260	27	216	576	1242	999

C. Experimental results

Most of the experiments were conducted with an image resolution of 640 by 480 pixels. However, in the case of the comparison with algorithms implemented by other authors and this work, it was essential to use other resolutions. For comparing with the implementation of the *Alexandra Institute*, an image resolution of 800 by 600 pixels was used. The resolution was determined by the implementation of the *ray tracing* algorithm as it could not be modified. While for comparison with results obtained by Popov et al. [20], an image resolution of 1024 by 1024 pixels was used. The resolution was determined from the experiments described in the article by Popov et al., since the authors worked with that fixed resolution.

Experiments conducted during the evaluation confirmed that the choice of the grid size can increase the performance of image generation. As a first approximation we considered the value suggested by Thrane and Simonsen [22], which indicates that the resolution is $3\sqrt[3]{N}$ boxes along the shortest axis, where N is the number of triangles in the scene. After several tests it was found that this division is not always the best, and that a good refinement for the grid size is between $\sqrt[3]{N}$ and $3\sqrt[3]{N}$ along the shortest axis. For each of the images, it must be determined empirically the optimal grid size that obtains the better performance within the range of values.

1) Evaluation of the different GPU versions implemented: The performance comparison between different versions of the algorithm implemented on GPU was made using the test cases Pri _I, Pri _II, Pri _III, Pri_IV and Pri _V. This evaluation has two stages. In the first stage, the optimal grid size for the test cases considered is determined, while in the second stage, each one of the GPU versions are executed for each of the test cases using the optimal grid size, founded in the previous stage.

The `RT(GPU-ii)` version was used to determine the optimal grid size for each test case. The results obtained executing in the PC with a GTX260 are summarized in Table VII. The table shows the number of frames per second (FPS) that can be computed for each of the images. From these results, we can conclude that the optimal grid size for all test cases considered, is obtained by halving each axis ($2 \times 2 \times 2$).

Table VII
FPS OF `RT(GPU-ii)` VERSION FOR DIFFERENT TEST CASES.

Scene	1x1x1	2x2x2	4x4x4	6x6x6	10x10x10	15x15x15
Pri_I	18.7	36.7	32.7	29.7	27.3	24.7
Pri_II	13.9	27.6	25.4	23.3	21.5	20.1
Pri_III	11.3	24.5	22.8	20.9	19.2	18.0
Pri_IV	8.4	18.5	18.0	16.5	15.9	15.1
Pri_V	6.7	16.6	15.5	14.6	13.8	13.5

Once the optimal grid size for each of the test cases was found, all the versions implemented on GPU are executed for the same test cases. Table VIII presents the performance obtained by the different GPU versions in the PC with a GTX260, measured in frames per second. The results show that the performance improves with the version, and that `RT(GPU-ii)` achieved the best performance.

The results obtained shown the importance of exploiting the different memory levels of GPU. In `RT(GPU)` version all the data is stored in the global memory while in the `RT(GPU-ml)` version most of the data is allocated in the texture and the constant memories. For the test cases considered, the use of the different memory levels of the GPU

Table VIII
FPS COMPARISON BETWEEN THE DIFFERENT GPU VERSIONS.

Scene	Optimal grid size	RT(GPU) (FPS)	RT(GPU-ml) (FPS)	RT(GPU-ii) (FPS)
Pri_I	2x2x2	5.8	26.2	36.7
Pri_II	2x2x2	5.1	20.2	27.6
Pri_III	2x2x2	4.9	18.2	24.5
Pri_IV	2x2x2	4.3	14.1	18.5
Pri_V	2x2x2	3.9	12.6	16.6

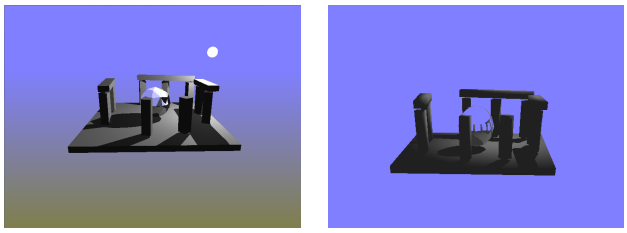
enables to improve the performance due to the reduction in memory access time, making the algorithm on average three and a half times faster than the algorithm that does not use it.

On the other hand, `RT(GPU-ii)` version improves the algorithm of ray-triangle intersection, with an algorithm that requires less arithmetic operations, thereby reducing the time needed to generate images. From the results, it is possible to notice that the improved intersection algorithm helps to make `RT(GPU-ii)` generate images 30% faster than `RT(GPU-ml)` version.

Finally, it can be seen, in Table VII as well as in Table VIII, that the increase in the number of triangles in a scene, increases the time required for generating the image and therefore reduces the FPS.

2) Comparative study with other ray tracing *implementations:* First, it was made a test to compare our proposal with a *ray tracing* implemented in GPU by *Alexandra Institute* [4] considering a scene provided with its implementation. The results obtained in the PC with a 9600M GT graphics card, showed that `RT(GPU-ii)` reach 13.0 FPS, while the implementation of the *Alexandra Institute* obtained 11.7 FPS.

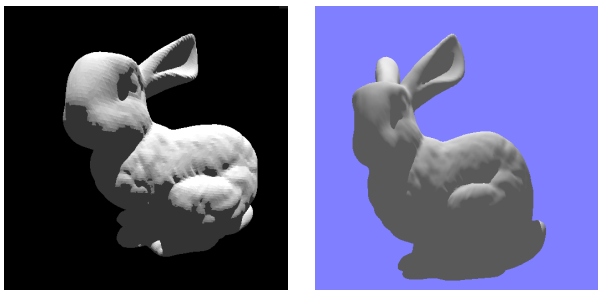
Figure 8(a) shows the rendering generated by the implementation of the *Alexandra Institute* and Figure 8(b) shows the rendering generated by our implementation for the same scene. It should be emphasized that the our implementation renders generic scenes unlike the *Alexandra Institute* implementation that was designed to produce images of a single type of scene, which generally causes that for the same scene our implementation requires more storage space and more memory accesses. Another difference in the implementations is that *Alexandra Institute* implementation considers the light sources as objects of the scene, while this was not considered in our implementation. In the images generated it can be seen subtle differences, such as the background color that could not be properly reproduced for the test case. Also there is a noticeable difference in the specular brightness of the sphere motivated by the incorrect reproduction of material used in the *Alexandra* image. On the other hand, in the image generated by our implementation it is best seen the reflection of objects near the sphere on its surface. Based on an analysis of the images, it could be concluded that there are no significant differences between the two images.



(a) (b)

Figure 8. Render of the image *Alexandra* with *Alexandra Institute* implementation (left) and with our implementation (right).

Then, a test was conducted to compare our proposal with a *ray tracing* implemented in GPU developed by Popov et al. [20] that used the kd-tree structure as spatial acceleration method. In their work, the rendering of the scene *Bunny* is presented as well as the time required for the image generation. The scene *Bunny* considered in our experiments was built from the observation of the rendering presented in the work of Popov et al.. The scene could not be exactly the same because of the limitations of the construction method used and the omission in the article of some relevant aspects of the scene. For example, the number of light sources is the same but the position is not identical and the material of the main object could not be accurately reproduced, since the article does not provide this information. The GPU used by Popov et al. is a NVIDIA GeForce 8800 GTX which has 112 cores. The GTX260 is the best suited GPU from the ones available for our experiments but has superior features. Figure 9(a) shows the render presented by Popov et al. and Figure 9(b) shows the render of the scene *Bunny* generated with our implementation.



(a) (b)

Figure 9. Render of the image *Bunny* with Popov et al. [20] implementation (left) and with our implementation (right).

The performance of our implementation of the *ray tracing* under the conditions described above is 6.1 FPS, while the performance of the work of Popov et al. is 5.9 FPS. The performance of both implementations for the test case considered is very similar. The results obtained

show that the *ray tracing* implemented is competitive with other ray tracing algorithms implemented in GPU.

3) Comparative study between different platforms: The comparative study of performance between the different platforms used consisted in the execution of the `RT(GPU-ii)` version to the test cases *Dragon*, *Buddha* and *Alexandra*. In each of the platforms, the algorithm is executed for each of the test cases using various grid sizes, determining the optimal grid size for each case and platform.

Tables IX, X and XI show the results obtained for the equipment 9500M GS, 9600M GT and GTX260, respectively.

Table IX
FPS OF `RT(GPU-ii)` VERSION IN A PC WITH A 9500M GS GRAPHICS CARD.

Dragon		Buddha		Alexandra	
Grid size	FPS	Grid size	FPS	Grid size	FPS
20x20x20	1.4	20x20x20	1.3	1x1x1	4.6
46x46x46	2.3	46x46x46	2.4	3x3x3	11.5
92x92x92	2.8	92x92x92	2.5	6x6x6	15.6
138x138x138	2.0	138x138x138	2.1	10x10x10	13.1
180x180x180	1.6	180x180x180	1.7	15x15x15	12.3
230x230x230	1.3	230x230x230	1.3	30x30x30	9.1

Table X
FPS OF `RT(GPU-ii)` VERSION IN A PC WITH A 9600M GT GRAPHICS CARD.

Dragon		Buddha		Alexandra	
Grid size	FPS	Grid size	FPS	Grid size	FPS
20x20x20	1.7	20x20x20	1.4	1x1x1	5.9
46x46x46	3.3	46x46x46	2.8	3x3x3	14.0
92x92x92	3.4	92x92x92	3.1	6x6x6	20.0
138x138x138	2.6	138x138x138	2.6	10x10x10	18.4
180x180x180	2.0	180x180x180	2.1	15x15x15	17.5
230x230x230	1.6	230x230x230	1.7	30x30x30	12.8

Table XI
FPS OF `RT(GPU-ii)` VERSION IN A PC WITH A GTX260 GRAPHICS CARD.

Dragon		Buddha		Alexandra	
Grid size	FPS	Grid size	FPS	Grid size	FPS
20x20x20	5.0	20x20x20	4.9	1x1x1	28.0
46x46x46	12.2	46x46x46	9.8	3x3x3	49.3
92x92x92	16.8	92x92x92	12.4	6x6x6	71.2
138x138x138	14.2	138x138x138	11.4	10x10x10	67.6
180x180x180	11.4	180x180x180	10.3	15x15x15	62.5
230x230x230	9.3	230x230x230	8.4	30x30x30	49.4

The results obtained suggest that for the test cases considered the optimal grid size ($92 \times 92 \times 92$ in *Dragon*, and *Buddha* images, and $6 \times 6 \times 6$ in *Alexandra* image) is the same regardless of the platform used. On the other hand, the experiments confirmed that the generation of images by the *ray tracing* algorithm implemented is faster, when the GPU has a greater number of cores. In addition to this, the results also show that the algorithm implemented can automatically scale with the number of cores of the GPU, this important property arises as a consequence of the CUDA programming model that helps to achieve it very easily.

4) Comparative study between CPU and GPU implementations: The experiments for comparing the performance between the implementations for CPU and GPU used the test cases Dist_I, Dist_II, Dist_III and *Bunny*. The evaluation consisted in the execution of `RT(GPU-ii)` the most efficient version on GPU, and `RT(CPU-ii)` the CPU version of *ray tracing* in the PC with a GTX260 graphics card. Table XII presents the FPS obtained by `RT(GPU-ii)` and `RT(CPU-ii)` versions and the acceleration (defined as $\frac{\texttt{FPS of GPU implementation}}{\texttt{FPS of CPU implementation}}$) achieved by using the GPU.

Table XII
FPS OF THE CPU AND GPU IMPLEMENTATIONS.

Scene	Grid size	CPU (FPS)	GPU (FPS)	Acceleration
Dist_I	50x50x50	1.4	17.2	12.29
Dist_II	50x50x50	1.5	20.1	13.40
Dist_III	50x50x50	1.6	21.1	13.19
Bunny	80x80x80	4.2	23.8	5.67

The results obtained for the test cases Dist_I, Dist_II, Dist_III and *Bunny* show that the GPU implementation produces images in less time than the CPU implementation, and therefore achieves a higher number of frames generated per second. In particular, the GPU implementation is on average more than 11 times faster than the CPU implementation for the four test cases considered in this study.

5) Evaluation of the effect of the object distribution in the scene: The test cases Dist_I, Dist_II and Dist_III were designed for detecting a possible weakness in the spatial acceleration structure chosen, when working with scenes in which objects are not evenly distributed. We assumed that in the case of Dist_III, which has all the items concentrated in the center of the scene, would reach less FPS than in the cases Dist_I and Dist_II, which are more evenly distributed. However, the results obtained (presented in Table XII) show otherwise. One possible explanation for this behavior in such scenes, is that when the objects are more evenly distributed in the scene, they cast more shadows (as it can be seen in Figure 5). Since the calculation of the shadows is computationally expensive, this cost counteracts the benefit gained with uniform distribution of objects in the scene.

VI. CONCLUSIONS AND FUTURE WORK

This work has presented an initial study on applying GPU computing in order to speed up the execution of the *ray tracing* algorithm. Three version of *ray tracing* were implemented in GPU using CUDA and were evaluated on different platforms using several images.

The experimental analysis showed that the GPU implementation increased significantly the number of frames that could be generated per second over the traditional CPU implementation (the `RT(GPU-ii)` version obtained an acceleration of up to $13\times$). This result shows the importance of making a good use of the different levels of memory on current GPUs.

We can also conclude that the performance achieved by our proposal (`RT(GPU-ii)` version) is competitive with the state of the art in real time *ray tracing* implementations on GPU, such as the one developed by the *Alexandra Institute* and the proposal of Popov et al.. In addition to this, our proposal showed a good scalability on the platforms used in this study. This property is very important because the GPUs improve its power at a vertiginous rate, which predicts that our implementation of *ray tracing* will achieve a better performance in new GPUs.

The main line for current and future work consists in evaluating the use of a different spatial acceleration structure, being the kd-trees the structure that best seems to suit. In addition to this, extending this work to a multi-GPU scenery or a hybrid multicore-GPU approach should be addressed in order to attain real time calculation of the frames in more complex scenes.

REFERENCES

[1] Aila, T. and Laine, S., *Understanding the efficiency of ray traversal on GPUs*, Proceedings of the Conference on High Performance Graphics (HPG '09), pp. 145-149, ACM, 2009.

[2] Akenine-Möller, T. and Haines, E., *Real-time rendering*, A. K. Peters Ltd., 2002.

[3] Appel, A., *Some Techniques for Shading Machine Renderings of Solids*, Proceedings of the American Federation of Information Processing Societies (AFIPS '68), pp. 37-45, ACM, 1968.

[4] *Alexandra Institute*, Computer Graphics Group, Denmark. Available at http://cg.alexandra.dk/tag/gpgpu/. Accessed on October 2010.

[5] Avcibas, I., Sankur, B. and Sayood, K., *Statistical Evaluation of Image Quality Measures*, Journal of Electronic Imaging, vol. 11, no. 2, pp. 206-223, 2002.

[6] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M. and Hanrahan, P., *Brook for GPUs: Stream computing on graphics hardware*, ACM Transactions on Graphics - Proceedings of ACM SIGGRAPH 2004, vol. 23, no. 3, pp. 777-786, 2004.

[7] *CUDA website*. Available at http://www.nvidia.com/object /cuda_home_new.html. Accessed on October 2010.

[8] Darema, F., *The SPMD Model: Past, Present and Future*, Proceedings of the 8th European PVM/MPI Users' Group Meeting, Lecture Notes in Computer Science 2131, p. 1, 2001.

[9] *DIRECTX website*. Available at http://developer.nvidia.com/page/directx.html. Accessed on October 2010.

[10] Dirik A., Bayram, S., Sencar, H. and Memon, N., *New Features to Identify Computer Generated Images*, Proceedings of the International Conference on Image Processing (ICIP 2007), pp. 433-436, IEEE, 2007.

[11] Goral, C., Torrance, K., Greenberg, D. and Battaile, B., *Modeling the Interaction of Light Between Diffuse Surfaces*, SIGGRAPH Computer Graphics, vol. 18, no. 3, pp. 213-222, ACM, 1984.

[12] Horn, D., Sugerman, J., Houston, M. and Hanrahan, P., *Interactive K-D Tree GPU Raytracing*, Proceedings of the Symposium on Interactive 3D Graphics and Games (I3D '07), pp. 167-174, ACM, 2007.

[13] Jensen, H., *Realistic Image Synthesis Using Photon Mapping*, A. K. Peters Ltd., 2001.

[14] Kay, T. and Kajiya, J., *Ray tracing complex scenes*, SIGGRAPH Computer Graphics, vol. 20, no. 4, pp. 269-278, ACM, 1986.

[15] Kirk, D. and Hwu, W., *Programming Massively Parallel Processors: A Hands-on Approach.*, Morgan Kaufmann, 2010.

[16] McGuire, M. and Luebke, D., *Hardware-Accelerated Global Illumination by Image Space Photon Mapping*, Proceedings of the Conference on High Performance Graphics (HPG '09), pp. 77-89, ACM, 2009.

[17] Murray, J. and Van Ryper, W., *Encyclopedia of Graphics File Formats*, O'Reilly Media, 1996.

[18] *OpenRT website*. Available at http://openrt.de/. Accessed on October 2010.

[19] Parker, S., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A. and Stich, M., *OptiX: A General Purpose Ray Tracing Engine*, ACM Transactions on Graphics - Proceedings of ACM SIGGRAPH 2010, vol. 29, no. 4, pp. 1-13, 2010.

[20] Popov, S., Günther, J., Seidel, H.-P., and Slusallek, P., *Stackless KD-Tree Traversal for High Performance GPU Ray Tracing* Computer Graphics Forum - Proceedings of Eurographic, vol. 26, no. 3, pp. 415-424, 2007.

[21] *Quake Wars: Ray Traced website*. Available at http://www.qwrt.de/. Accessed on October 2010.

[22] Thrane, N. and Simonsen, L., *A Comparison of Acceleration Structures for GPU Assisted Ray Tracing*, Master's thesis, Department of Computer Science, Faculty of Science, University of Aarhus, Denmark, 2005.

[23] Wald, I., *Realtime Ray Tracing and Interactive Global Illumination*, PhD. Thesis, Computer Graphics Group, Saarland University, 2004.

[24] Ward, G., Rubinstein, F. and Clear, R., *A Ray Tracing Solution for Diffuse Interreflection*, Proceedings of the 15th annual conference on Computer graphics and interactive techniques (SIGGRAPH '88), pp. 85-92, ACM, 1988.

[25] Whitted, T., *An Improved Illumination Model for Shaded Display*, Communications of the ACM, vol. 23, no. 6, pp. 343-349, 1980.

Bibliografía

[1] Tomas Akenine-Moller, Tomas Moller, and Eric Haines. Real-time rendering. *A. K. Peters, Ltd.*, 2002.

[2] Akira Fujimoto and Takayuki Tanaka and Kansei Iwata. ARTS: Accelerated Ray-Tracing System. *IEEE Computer Graphics and Applications*, 6, 1986.

[3] Andrew S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 1984.

[4] Arthur Appel. Some techniques for shading machine renderings of solids. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 37–45, New York, NY, USA, 1968. ACM.

[5] Christian Lauterbach, Dinesh Manocha, David Tuft, Sung-Eui Yoon. Interactive Ray Tracing of Dynamic Scenes using BVHs. *IEEE Symposium on Interactive Ray Tracing*, 2006.

[6] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, Pat Hanrahan. Interactive k-D Tree GPU Raytracing. *Stanford University*, 2007.

[7] Página Web de Autodesk 3ds Max. http://www.autodesk.es/adsk/servlet/pc/index?siteID=455755&id=14626995.

[8] Página Web de Autodesk Maya. http://www.autodesk.es/adsk/servlet/pc/index?siteID=4557

[9] Página Web de Blender. http://www.blender.org/.

[10] Página Web de FileFormat.Info. http://www.fileformat.info/format/wavefrontobj/egff.htm.

[11] Página Web de Micah Taylor. http://kixor.net.

[12] Página Web de nVidia CUDA. http://developer.nvidia.com/object/gpucomputing.html.

[13] Página Web de OpenRT. http://openrt.de/.

[14] Página Web de Quake Wars. http://www.qwrt.de/.

[15] Página Web de SDL (Simple Directmedia Layer). http://www.libsdl.org/.

[16] Blog del grupo de Computación Gráfica del Alexandra Institute de Dinamarca. http://cg.alexandra.dk/.

[17] A.E. Dirik, S. Bayram, H.T. Sencar, and N. Memon. New features to identify computer generated images. In *ICIP07*, pages IV: 433–436, 2007.

[18] B. Ghanem, E. Resendiz, and N. Ahuja. Segmentation-based perceptual image quality assessment (spiqa). In *ICIP08*, pages 393–396, 2008.

[19] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl.*, 7(5):14–20, 1987.

[20] Donald Hearn and M. Pauline Baker. Computer graphics / d. hearn, m.p. baker. *Prentice-Hall*, 1988.

[21] Henrik Wann Jensen. Realistic Image Synthesis Using Photon Mapping. *AK Peters, Ltd.*, 2001.

[22] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive Rendering with Coherent Ray Tracing. *Saarland University*, 2001.

[23] Ismail Avcibas, Bülent Sankur. Statistical Analisys of Image Quality Measures. *Department of Electrical and Electronic Engineering, Bogaziçi University, Istanbul, Turkey*, 2001.

[24] James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes, Richard L. Phillips. Introducción a la graficación por computador. *Addisson-Wesley Iberoamericana, S.A.*, 1996.

[25] John Amanatides and Andrew Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. 1987.

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

[27] Matt Pharr, Greg Humphreys. Physically based rendering. *Morgan Kauffman*, 2004.

[28] Niels Thrane, Lars Ole Simonsen. A Comparison of Acceleration Structures for GPU Assisted Ray Tracing. Master's thesis, Department of Computer Science. Faculty of Science. University of Aarhus., April 2005.

[29] Timothy John Purcell. Ray tracing on a stream processor. Technical report, Stanford University, Stanford, CA, USA, 2004. Adviser-Hanrahan, Patrick M.

[30] D.V. Rao and L.P. Reddy. Image quality assessment based on perceptual structural similarity. In *PReMI07*, pages 87–94, 2007.

[31] Stefan Popov, Johannes Günther, Hans-Peter Seidel, Philipp Slusallek. Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Saarland University and MPI Informatik, Saarbrücken, Germany*, 2007.

[32] Jeremy Sugerman Tim Foley. Kd-tree acceleration structures for a gpu raytracer. In *Graphics Hardware*, pages 15–22, 2005.

[33] Vlastimil Havran, Jan Prikryl, Werner Purgathofer. Statistical comparison of ray-shooting efficiency schemes. Technical report, Institute of Computer Graphics, Vienna University of Technology, 2000.

[34] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination.* PhD thesis, Computer Graphics Group, Saarland University, 2004.

[35] Ingo Wald and Philipp Slusallek. State of the art in interactive ray tracing. In *State of the Art Reports, EUROGRAPHICS 2001*, pages 21–42. EUROGRAPHICS, Manchester, United Kingdom, 2001.

[36] Whitted Turner. An improved illumination model for shaded display. *Communications of the ACM*, 1980.

[37] G. Zhai, W. Zhang, X. Yang, and Y. Xu. Image quality metric with an integrated bottom-up and top-down hvs approach. *VISP*, 153(4):456–460, August 2006.