

Documentación - “Computación Gráfica sobre GPU”

Gonzalo Ordeix, Santiago Cioli

13 de agosto de 2010

Índice general

1. Introducción	3
2. Generación de imágenes por computadora	4
2.1. Introducción	4
2.1.1. Concepto de escena	5
2.1.2. Trazado de rayos	5
2.2. Modelos computacionales de iluminación	6
2.2.1. Raycasting	7
2.2.2. Clasificación de los algoritmos de generación de imágenes por computadora	7
2.3. Raytracing	10
3. Aceleración del algoritmo de Raytracing	18
3.1. Introducción	18
3.2. Paralelización de algoritmos en GPU	18
3.3. Métodos de aceleración para raytracing	23
3.4. Estructuras de aceleración espacial	24
3.4.1. Subdivisión espacial	24
3.4.2. Jerarquía de Volúmenes Envolventes (BVH)	29
3.4.3. Conclusiones sobre aceleración espacial	31
3.5. Raytracing interactivo	33
3.5.1. Estado del Arte	34
3.5.2. RI sobre multiprocesadores de memoria compartida . .	34
4. Propuesta	37
4.1. Introducción	37
4.2. Descripción general	37
4.2.1. Preprocesamiento	39

4.2.2.	Intersección	41
4.2.3.	Aceleración espacial	42
4.2.4.	Estructura de datos	45
4.2.5.	Paralelización del algoritmo	47
4.2.6.	Eliminación de la recursión	48
4.2.7.	Estructura del núcleo de Raytracing	49
4.3.	Versiones	51
4.3.1.	Versión 1 - RT(GPU)	51
4.3.2.	Versión 2 - RT(GPU-JM)	52
4.3.3.	Versión 3 - RT(GPU-JM-IR)	52
4.3.4.	Versiones para CPU	52
5.	Experimentación	53
5.1.	Estrategias de evaluación de calidad de imagen	53
5.2.	Casos de prueba	55
5.3.	Equipos utilizados	59
5.4.	Experimentos	60
5.4.1.	Comparación entre versiones	61
5.4.2.	Comparación entre C y CUDA	63
5.4.3.	Comparación entre equipos	67
5.4.4.	Comparación con otras implementaciones	69
6.	Conclusiones y Trabajo a futuro	73
6.1.	Conclusiones	73
6.2.	Trabajo a futuro	73
A.	Estructuras de datos	74
A.1.	Version 1	75
A.2.	Versión 2	77
B.	Algoritmos de Recorrida	79
B.1.	Particularidades del KD-Tree	79
B.2.	Particularidades del BVH	85

Capítulo 1

Introducción

FALTA COMPLETAR.....

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. En la actualidad 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 no menor es el tiempo de procesamiento que requieren este tipo de técnicas, para cada una de las imágenes que se van a incluir en la versión final de la película toman 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 son 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 grandes clusters de computadoras para realizar la generación de las imágenes.

Para poder comprender la forma en que se generan las imágenes fotorealistas 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 o computar las imágenes. En las siguientes secciones se introduce a dichas temáticas, abordando conceptos esenciales tales como:

que es una escena y los algoritmos 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 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 [19]. 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, subdivisiones de superficies, superficies ISO, etc. Casi cualquier tipo de objeto puede ser usado como primitiva de una escena.

2.1.2. Trazado de rayos

El rayo $R(t) = O + tD$ es por lo general representado mediante un punto de origen O y una dirección D . 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 rayo¹ 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

¹Se 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 gran parte de las primitivas usadas para construir escenas existen diferentes algoritmos que evalúan la intersección con un rayo. Cada uno de los algoritmos tienen diferentes valores respecto a propiedades como velocidad, 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 requerido y solo es requerido para algoritmos de iluminación avanzados. Excepto para estos modelos de iluminación especiales, este problema no es común en los algoritmos trazadores de rayos.

2.2. Modelos computacionales de iluminación

En esta sección se abordan las técnicas más populares para la generación de imágenes fotorealistas. Comenzando por los algoritmos en el que se basan la mayoría de los algoritmos actuales: Raycasting de Appel [4] y Raytracing de Whitted [35]. 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 pixels de la imagen y no tiene una forma particular de computar el color de esos pixels. Por ejemplo, esta técnica no tiene en cuenta el cálculo de sombras ni las reflexiones entre objetos, como si lo hace, por ejemplo Raytracing. Una posible implementación de la rasterización es scan lines [23].

2.2.1. Raycasting

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 pixel, 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 pixels haya en el dispositivo de visualización a utilizar. También puede ser que se tenga un tamaño de imagen en pixels, en este caso se genera un rayo por cada pixel de la imagen a generar. Las coordenadas de los pixels 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 importante 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 Raycasting 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, buscan dada una escena, definición matemática o algún tipo de representación abstracta, 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 los enfoques de todos 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 plantea para calcular la radiancia la división de la escena en pequeñas partes y se utiliza alguna solución numérica para aproximar los valores [23]. Además, es un modelo de iluminación global.
- El modelo basado en métodos de Monte Carlo, el cual busca aproximar los valores de radiancia en base a aproximaciones estadísticas basadas en la integración de Monte Carlo [20]. 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 [20].

Los algoritmos de generación de imágenes, que surgen en base al algoritmo original de Raycasting 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. 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 descrito se describirá el algoritmo más representativo para cada estrategia. Estas técnicas que se presentan son: Raytracing para el modelo de Whitted, Radiosidad para los algoritmos basados en elementos finitos y Photon Mapping para los basados en métodos de Monte Carlo.

Raytracing

El algoritmo de Raytracing propuesto por Turner Whitted en 1980 [35] está basado en el algoritmo de Raycasting. 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 Raycasting es un algoritmo sencillo para la generación de imágenes que tiene un modelo de iluminación propio y muy simple 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, ese es el cambio de medio, se genera una desviación de la luz, dando la impresión de que los objetos se deforman. Así mismo introduce también los conceptos reflexión a las imágenes, admitiendo objetos espejados en las escenas obteniendo un grado de realismo visual superior de los generados por Raycasting.

Radiosidad

El algoritmo de radiosidad utiliza los principios de Raytracing para el cálculo de las superficies visibles y sombras, así como las reflexiones y refracciones pero a diferencia del algoritmo de Raytracing 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 pixels de la imagen que se quiere generar. Como se precálculan 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

Este algoritmo fue introducido por Henrik Wan Jensen en el año 1996[20]. Esta técnica que ha evolucionado fuertemente en los últimos años, cuenta con una excelente calidad en las imágenes que genera y es 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 Raytracing 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 Raytracing. 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. Raytracing

Este trabajo profundiza el estudio del algoritmo de Raytracing, 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 Raytracing, 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.

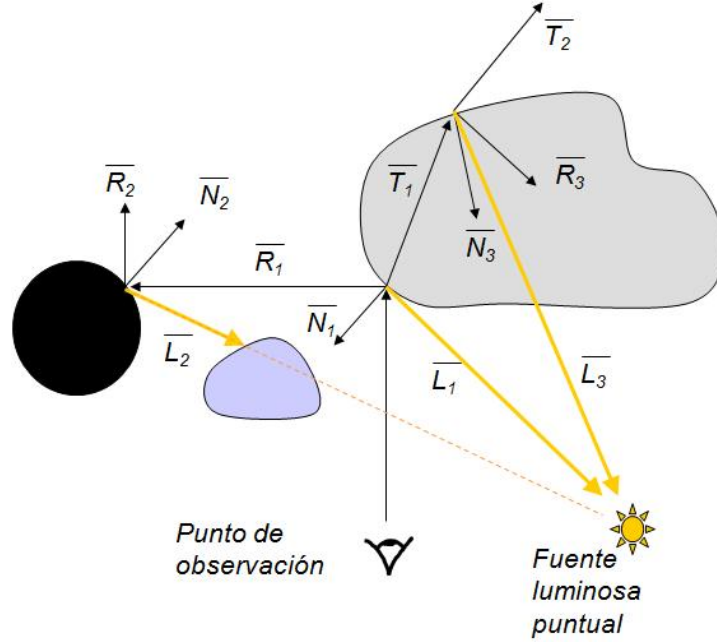


Figura 2.1: Generación de rayos del algoritmo de Whitted a partir de un único rayo primario.

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 pixel. 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 [23]. 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.

Punto de observación

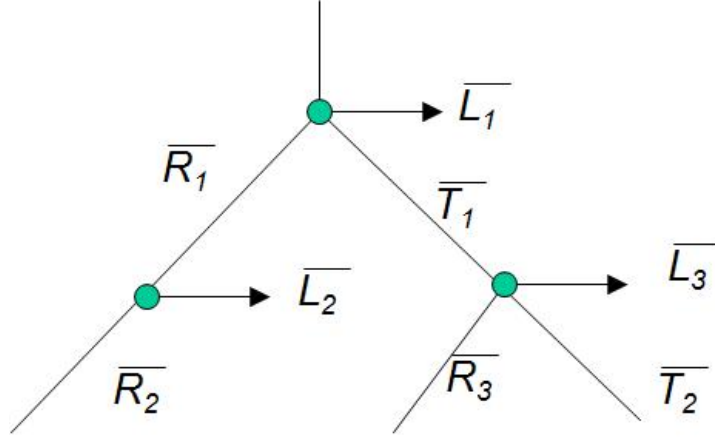


Figura 2.2: Árbol de rayos que surge del ejemplo de la Figura 2.1.

La profundidad del árbol de rayos afecta directamente el tiempo de ejecución del algoritmo y la calidad de la imagen que se quiere obtener. 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 [23]. 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.

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

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 [23]. 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 transmisión: 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 Raytracing

El algoritmo de Raytracing 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 en la mayoría de las aplicaciones 3D basados en la rasterización de imágenes formadas por polígonos. Por este motivo Raytracing 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 Raytracing, como por ejemplo el juego Quake 3 que utiliza el motor openRT [13], que utiliza un cluster

de 20 nodos que cuentan con procesadores Athlon 64. Dependiendo de lo que se esté intentando dibujar en el juego en ese momento puede requerir algo más de capacidad de cómputo para funcionar de manera adecuada.

En el Algoritmo 1 se muestra un pseudocódigo del algoritmo de Raytracing.

En el Algoritmo 2 se presenta el pseudocódigo 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 *verificarSombra* del Algoritmo 2 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 [23]. 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 pixel invocando a la función *calcularColorFinal*. Esta función aplica la ecuación de iluminación del modelo de Whitted.

Algoritmo 1 Pseudocódigo del algoritmo de Raytracing.

para todo pixel p en imagen a generar **hacer**

$r = \text{rayo}(\text{observador}, p);$

$p.\text{color} = \text{trazarRayo}(r, 1);$

fin para

Algoritmo 2 Seudocódigo de la función trazarRayo.

Entrada: Rayo r , Entero $profActual$

Salida: Color $color$

```
    si  $profActual < MAXPROF$  entonces
        devolver  $colorNulo$ ;
    fin si
    objetoMasCercano =  $\infty$ ;
    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  $\neq \infty$  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 Raytracing

3.1. Introducción

Esta sección presenta en detalle de 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 capítulo 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 GPU 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 de un lenguaje de alto nivel, CUDA permite el uso

del lenguaje C para la programación.

La arquitectura CUDA basa su poder de procesamiento en un conjunto variable (dependiente del modelo de GPU) de multiprocesadores, donde cada uno de ellos procesa parte de la carga de trabajo en paralelo con los demás. Un multiprocesador esta 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*).

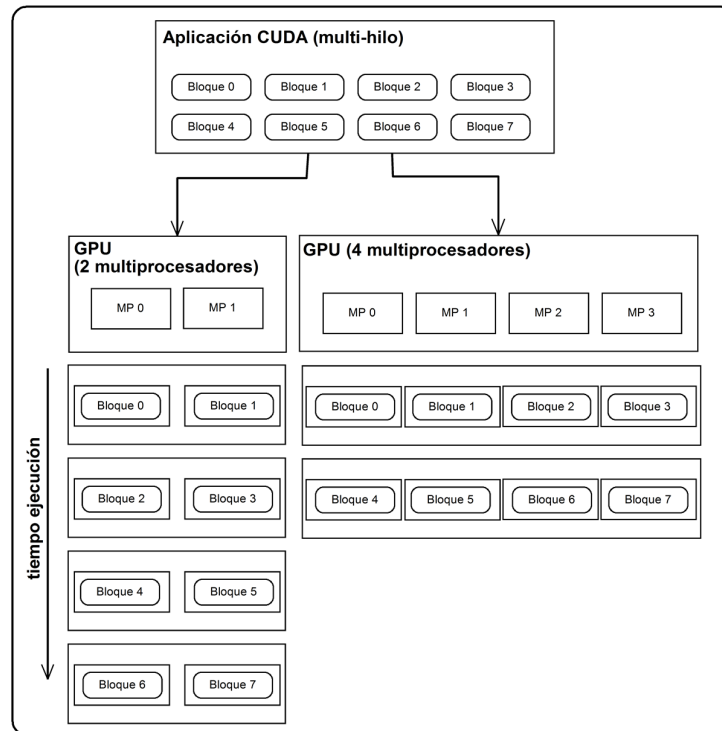


Figura 3.1: Ejemplo de escalabilidad automática en el número de multiprocesadores.

El modelo de programación esta basado en tres abstracciones fundamentales, dispone de una jerarquía de grupos de hilos de ejecución, memoria compartida y barreras de sincronización de la ejecución. 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 se 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 cualquier aplicación CUDA puede ejecutar sobre cualquier 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.

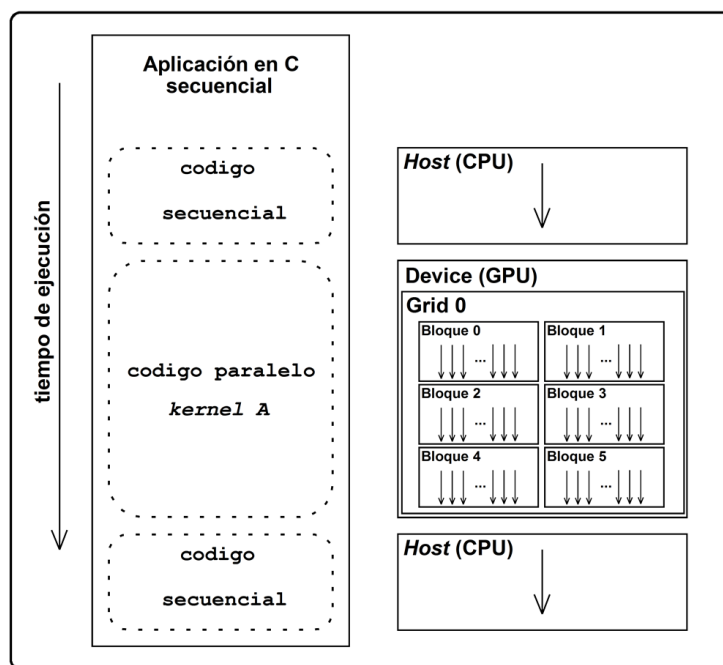


Figura 3.2: Ejemplo de ejecución de una aplicación CUDA.

En la extensión del lenguaje C que hace CUDA es posible definir funciones (de la misma forma que en el lenguaje base) que a diferencia de las funciones normales de C, cuando son invocadas ejecutan N veces en paralelo, mediante N hilos de ejecución de CUDA 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.

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.

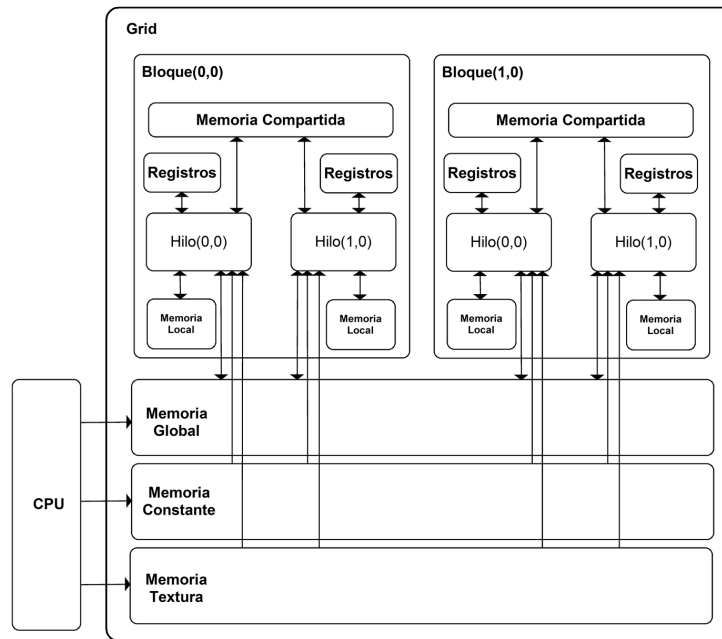


Figura 3.3: Jerarquía de memoria de la GPU.

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 formar 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.

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 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 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.

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 está 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 raytracing

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[33].

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 pixel 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 pixel varía significativamente con respecto a las otras, entonces el pixel es dividido en cuatro partes iguales. Luego se lanzan rayos primarios por cada nueva parte de la misma forma que se lanzaron en el pixel original. Repitiendo esta división hasta lograr la calidad deseada. Una vez terminada la subdivisión el color del pixel es interpolado según el color de cada una de sus partes[33].
- El otro camino es reducir el número de rayos que deben ser trazados por cada rayo primario, es decir la cantidad de rayos secundarios. Un ejemplo de este tipo de técnicas es la llamada *shadow caching*. La mayoría de los rayos que se tratan en un algoritmo de raytracing 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 de sombra son similares (sobre todo los que son originados por la misma fuente de luz) y que además intersecan con el mismo objeto [33].

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 raytracing 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 [33].

3.4. Estructuras de aceleración espacial

Si se consideran todos los chequeos necesarios para generar una imagen utilizando el algoritmo de Raytracing, estos pueden llegar a tomar el 95 % del tiempo de cálculo [27]. 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 algunas, las más importantes según el trabajo de Thrane y Ole [27], de estas técnicas.

3.4.1. Subdivisión espacial

En el método de subdivisión 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 voxeles (el término voxel es la extensión a tres dimensiones de su homónimo en dos dimensiones pixel). La forma de definir estos voxeles es lo que marca la diferencia entre las técnicas de subdivisión espacial. Una vez que los voxeles están definidos juegan el mismo papel en todas las técnicas.

La gran ventaja de esta técnica de subdivisión es que solo los objetos asignados a los voxeles atravesados por los rayos deben ser probados para una posible intersección.

Subdivisión espacial uniforme.

Cuando las particiones (voxeles) son todas del mismo tamaño la técnica se denomina subdivisión espacial uniforme. En la Figura 3.4 se muestra un

ejemplo de este tipo de subdivisión, que es totalmente independiente de la estructura de la escena.

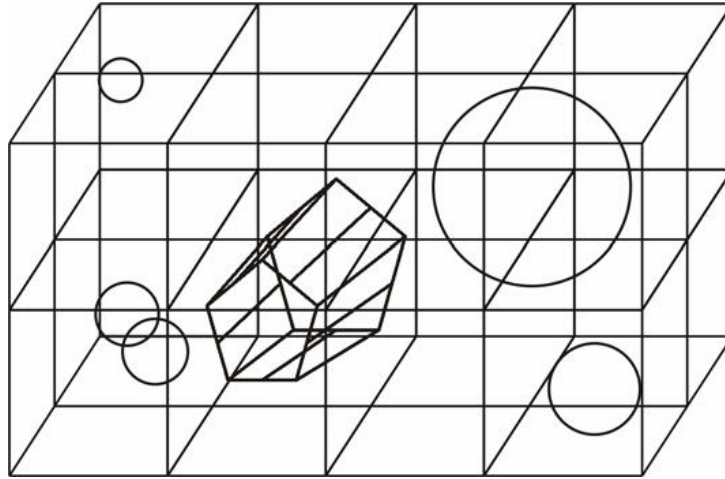


Figura 3.4: Subdivisión espacial uniforme aplicada a una escena.

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

La construcción de la subdivisión 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 subdivisión espacial uniforme, el único parámetro necesario para su construcción es su resolución a lo largo de los tres ejes imaginarios.

Según Thrane y Ole 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 sea $3\sqrt[3]{N}$ voxeles a lo largo del eje más corto, donde N es el número de triángulos de la escena. Aunque también se sugiere que puede ajustarse empíricamente para lograr mejores resultados en imágenes particulares. Una vez que la resolución es determinada, podemos construir una matriz tridimensional de listas de objetos que servirá para manejar los

voxels construidos y su contenido. Luego para cada objeto de la escena, se deben encontrar los voxels que lo contienen y agregar una referencia al objeto a cada uno de ellos.

La técnica usada para moverse a través de los voxels 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) [23]. Basados en DDA, Fujimoto et al. [2] proponen el algoritmo que permite recorrer la grilla y este es usado por Thrane y Ole, con algunas mejoras propuestas por Amanatides y Woo [27]. En el Algoritmo 3 se muestra un pseudocódigo de la técnica para dos dimensiones para facilitar la comprensión (extenderla a tres dimensiones es simple).

Algoritmo 3 Recorrida de los voxels atravesados por un rayo.

```

mientras  $X$  y  $Y$  estén dentro de la grilla hacer
    chequeo de intersección con los triángulos del voxel actual
    si hay intersección en este voxel 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 con el Algoritmo 3 se debe identificar el voxel inicial, es decir el primer voxel que atraviesa el rayo. Si el origen del rayo se encuentra dentro de un voxel determinado, entonces este es el inicial. En caso contrario, se busca el primer punto de la grilla que interseca con el rayo y se usa este punto para localizar el voxel 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 un voxel. 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 del voxel 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 al siguiente voxel. Esta se calcula de la siguiente manera:

$$delta_x = \frac{voxelsize_x}{raydirection_x}$$

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 los voxeles que el rayo atraviesa.

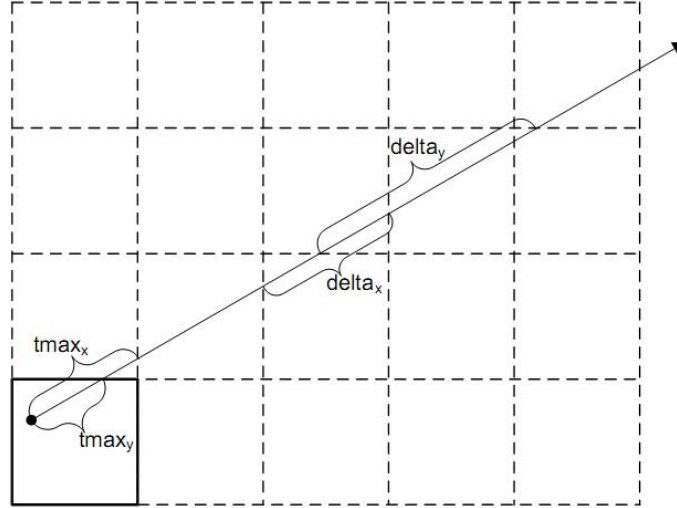


Figura 3.5: Relación entre el rayo, $tmax_x$, $tmax_y$, $delta_x$ y $delta_y$. El voxel inferior izquierdo contiene el punto origen del rayo.

La subdivisión 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) [28]. 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 voxel atravesado por el rayo puede ser localizado y accedido 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 los voxeles atravesados se hace de forma incremental y con sumas sencillas, lo cual elimina la necesidad de una pila y hace posible visitar los voxeles en orden, es decir aumentando la distancia desde el origen del rayo.
- Se puede explotar el hecho de que los voxeles se recorren en orden para detener la recorrida cuando se da una intersección en el voxel actual.
- El algoritmo de recorrido a través de los voxeles que interseca el rayo está dado por un vector lo cual es altamente compatible con el conjunto de instrucciones de una GPU.

En la subdivisión espacial se puede dar el caso de que un polígono sea referenciado por más de un voxel. 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 [24]. Al emplear estrategias de la paralelización este tipo de técnicas no pueden utilizarse, lo que implica que el algoritmo paralelo debería hacer chequeos repetidos [27].

KD-Trees

Al igual que la subdivisión espacial uniforme la estructura kd-tree es una instancia particular de la subdivisión 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 los voxeles y tienen las referencias a los objetos que se encuentran dentro de los mismos. 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 subárboles, mientras que los nodos hojas contienen listas de objetos.

Esta técnica de división del espacio tiene prácticamente las mismas ventajas que la subdivisión 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 voxel sea menor a un número definido anteriormente. Havran [32] sugiere que la profundidad máxima sea igual a 16 y que la cantidad de objetos por voxel sea 2 para lograr un rendimiento óptimo. Thrane y Ole [27] 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 [26], los cuales usaron $8 + 1,3 \log(N)$ como profundidad máxima, donde N es el número de triángulos de la escena.

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 subdivisión 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.

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

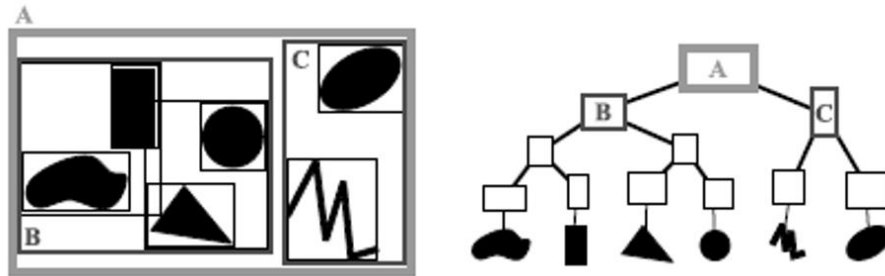


Figura 3.6: Ejemplo de estructura BVH que utiliza cajas alineadas a los ejes.

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 [18]. Está formalizada usando el área de la superficie del nodo padre y del nodo

hijo y sigue la relación de la Ecuación 3.1.

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

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 [25] y Goldsmith y Salmon [18], en estos trabajos se puede profundizar sobre el tema.

3.4.3. Conclusiones sobre aceleración espacial

En el trabajo de Thrane y Ole se analizan experiencias obtenidas al implementar y usar la subdivisión 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 los voxeles es muy simple de implementar y que requiere pocas operaciones aritméticas para avanzar de voxel a voxel. Estas propiedades del algoritmo de recorrida 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 los voxeles 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, en comparación con el obtenido al usar las estructuras Kd-Tree o BVH. Además, concluyen que el algoritmo para la construcción de la subdivisión uniforme es más simple que el de las estructuras Kd-Tree o BVH y que puede ser paralelizado en GPU con mayor facilidad.

Por otro lado, Thrane y Ole [27] 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 subdivisión espacial uniforme. También se recalca que las dos técnicas para realizar la recorrida a través de los voxels 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 subdivisión 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.

	SEU	Kd-Tree	BVH
Complejidad del algoritmo de construcción	Media	Alta	Baja
Aceleración lograda	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.

Con respecto a la estructura BVH, Thrane y Ole [27] 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, cosa que se buscaba 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 [27]. Además Thrane y Ole [27] concluyeron que para escenas complejas esta estructura es la que tiene mejor rendimiento sobre la GPU, comparando con la subdivisión 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 mas costosa que la construcción de la grilla uniforme.

Lauterbach et al. [5] luego de haber usado la estructura BVH para diversos casos de prueba concluyó 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.

3.5. Raytracing interactivo

Se llama Raytracing 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 Raytracing en tiempo real esta implementación no serviría para hacer videojuegos 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. [34] 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 este 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 Raytracing surge como una alternativa válida.

Para lograr que el algoritmo de Raytracing sea interactivo se le deben aplicar estrategias de simplificación y aceleración. Existe una estrategia que consiste en utilizar un algoritmo basado en rasterización, asumiendo que la velocidad de la rasterización es mejor que la de Raytracing se puede utilizar Raytracing solamente para el cálculo de algunos efectos de iluminación y no realizar toda la generación de la imagen con este algoritmo. Otra estrategia saca provecho de la coherencia temporal entre las imágenes generadas, esta técnica se basa en extraer información de la imagen generada en el paso anterior para generar solo algunos píxeles de la imagen siguiente. Utilizando como base Raytracing otra de las formas de mejorar el tiempo de procesamiento es tratando de reducir el costo de cálculo de cada uno de los píxeles. Otra forma de aceleración es la utilización de algoritmos que permitan bajar los costos computacionales asociados al trazado de rayos, estos pueden ser utilización de algún tipo de particionamiento espacial, utilización de arquitecturas de memoria de las CPU actuales que puedan ser utilizadas de manera eficiente. Por último la utilización de la capacidad del algoritmo de ser ejecutado en paralelo, esto requiere una buena optimización de los algoritmos teniendo en cuenta el balanceo de carga y las latencias de sincronización.

3.5.2. RI sobre multiprocesadores de memoria compartida

El artículo de Wald et al. describe la exploración dentro de las técnicas de optimización para los algoritmos de Raytracing de modo de intentar

que se puedan ejecutar de manera interactiva. En esta investigación realizan la implementación de un Raytracer que corre sobre una arquitectura multi-procesador. En la solución desarrollada se logró la interactividad, en parte porque el sistema corre en una máquina de gran poder de cómputo (SGI Origin 2000). Por otra parte es posible aún en estas condiciones lograr que Raytracing sea interactivo por tres características del algoritmo:

- Raytracing 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 sublineal 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.

Estos ítems permiten: que la implementación sea interactiva, poder generar imágenes para escenas de gran porte y obtener imágenes con las características de realismo clásicas del Raytracing.

Para el trabajo se utilizó como base el algoritmo clásico de Whitted [35] 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: 1) Acelerar o eliminar cálculos de verificación de intersección entre rayos y objetos. 2) Paralelización. Utilizan ambas técnicas buscando la combinación de ambas que les brinde un mejor desempeño. Para optimizar en la cantidad de cálculos de intersección se utiliza división espacial de la escena, utilizando no solamente una estructura sino que se combinan una división en grilla de la escena con volúmenes acotantes para los objetos de dicha escena.

Para paralelizar el algoritmo se utiliza un sistema de memoria compartida, y el algoritmo utiliza una estrategia maestro-esclavo en donde el proceso maestro inicializa la escena a renderizar y se generan los rayos a ser lanzados en una cola de rayos de la cual los procesos esclavos obtendrán a demanda 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 Raytracing 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) y se logró el objetivo de tiempo real con 64 procesadores. Esta implementación de raytracing mostró que es un algoritmo muy bueno para mostrar efectos de luz dinámicos pero no así para procesar escenas en las cuales los objetos cambian dinámicamente.

Capítulo 4

Propuesta

4.1. Introducción

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

La primera propuesta de aceleración se basa en paralelizar el trazado de rayos primarios utilizando las prestaciones multihilo de las GPUs. Recordar que para obtener el color de un pixel con Raytracing se debe lanzar un rayo primario (como mínimo), entonces para renderizar una imagen cuya resolución sea 1024 por 768 pixeles se deben trazar 786432 rayos primarios. En una implementación tradicional del algoritmo se procesa un pixel a la vez, es decir en forma secuencial.

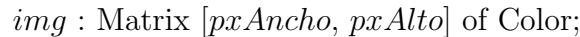
4.2. Descripción general

El algoritmo de Raytracing de Whitted requiere como entrada una escena, la cual será renderizada. Como muestra el pseudocódigo del Algoritmo 4, para generar una imagen hay una primera etapa en la cual 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. En el segundo se construye la grilla de la subdivisión 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 GPU, paralelizando

el cálculo de cada uno de ellos.

Algoritmo 4 Pseudocódigo 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 */
 : 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);

```

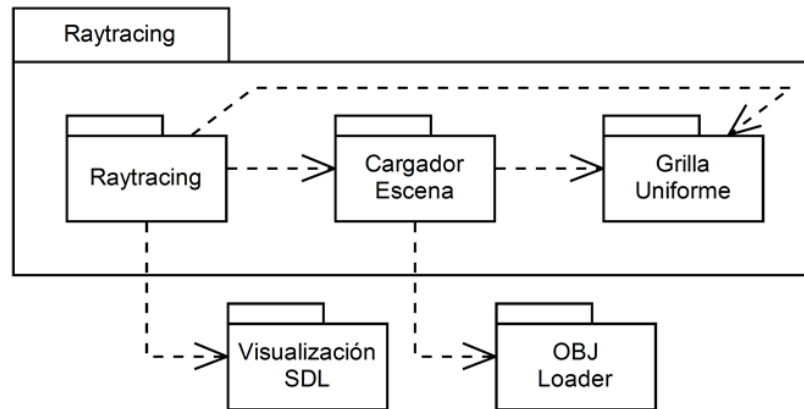


Figura 4.1: Arquitectura del Raytracing implementado sobre GPU.

En la Figura 4.1 se muestra un diagrama de arquitectura de la solución implementada. 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, el cual especifica la escena y los materiales utilizados en la misma. El componente “Cargador Escena” fue implementado en el marco de este proyecto y utiliza el “*OBJ Loader*” para leer la escena desde archivo. Este componente generado en el proyecto, 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 por el cargador externo al proyecto. Además se encarga de la construcción de la subdivisión 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 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 utilizando la GPU.

En la segunda etapa del proceso de generación de imagen se calcula el color de cada pixel mediante el algoritmo de Raytracing. 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 hacen al núcleo del raytracing, como por ejemplo el algoritmo de recorrida de un rayo dentro de la subdivisión 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) [14].

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 Raytracing. Las características extras soportadas por el *OBJ Loader* y que extienden el formato OBJ son:

- soporte de nuevas primitivas, como por ejemplo plano.

- soporte de elementos extra, se soportan 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 para el contexto de este proyecto. De esta manera se desacopla el resto del algoritmo de la etapa de carga, lo cual permite cambiar la herramienta de cargar la escena fácilmente.

Archivo de configuración

```
#= tamaño de la partición espacial
TAMANIO_GRILLA.X=64
TAMANIO_GRILLA.Y=50
TAMANIO_GRILLA.Z=50
#= resolución de la imagen
RESOLUCION.X=640
RESOLUCION.Y=480
INFINITO= 340282346600000000
#= valor tomado como cero
ZERO=0.000001
#= valor máximo rebotes rayo
PROFUNDIDAD_RECURSION=3
#= hilos por bloque
THREADS.X=8
THREADS.Y=16
ESCENA=../escenas/afrodita.obj
```

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

En la etapa de preprocesamiento se usan parámetros como la resolución de la grilla de aceleración. Este valor así como también otros parámetros

del algoritmo de raytracing 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 recompilar el código fuente con cada cambio. La Figura 4.2 se muestra una posible configuración del algoritmo.

En el archivo se debe configurar el tamaño de la subdivisión espacial uniforme, se debe indicar el tamaño de la partición en cada eje de coordenadas. Es posible definir también la resolución de la imagen (en píxeles) que genera el algoritmo de raytracing. En este archivo también se definen valores para el cero y para el infinito que usa el algoritmo de raytracing, el ajuste del primer valor 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 raytracing y que es interesante variar para cada escena, es la profundidad máxima considerada por el algoritmo para cada árbol de rayos que genera cada rayo primario (debido a los sucesivos rebotes por los fenómenos de reflexión y refracción). Por último, se debe especificar la dimensión de los bloques de hilos de ejecución.

4.2.2. Intersección

En este proyecto se decidió acelerar el algoritmo de Raytracing 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 [27]. Para esto se tomaron medidas en los caminos posibles para lograr acelerar el algoritmo: implementar un algoritmo de intersección rayo-primitiva eficiente y reducir la cantidad de intersecciones rayo-primitiva que se prueban por cada rayo primario. porque es una Es importante escoger un buen método para la verificación de intersección entre los rayos y las primitivas de construcción de escenas, dado que este es parte central del Raytracing. 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 el triángulo como primitiva de construcción de escenas porque permite construir a partir de ella cualquier objeto tridimensional que se quiera modelar pero a su vez es una primitiva sencilla y por lo tanto el algoritmo de su intersección con un rayo también lo es, lo que implica poca cantidad de operaciones aritméticas. 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 se de 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 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 la intersección con todos los objetos de la escena, para cada rayo primario. En la etapa de relevamiento y evaluación de las distintas estrategias se presentan tres estructuras de aceleración espacial: la subdivisión espacial uniforme, la subdivisión espacial adaptativa (utilizando Kd-trees) y la jerarquía de volúmenes envolventes (BVH).

La estrategia de aceleración espacial que se adopto en este proyecto fue la subdivisión 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 se podría acelerar ejecutándolo en la GPU.

El primer algoritmo implementado para generar la subdivisión espacial uniforme fue la construcción por fuerza bruta. Como se muestra en el Algoritmo 5 este método de construcción recorre todos los voxels de la grilla y para cada uno de ellos recorre todos los objetos de la escena para probar si hay intersección voxel-objeto. Este método es ineficiente porque por lo general se recorren muchos voxels innecesarios (que no tienen intersección con el objeto) por cada objeto de la escena.

En el Algoritmo 6 se describe un algoritmo de construcción optimizado, en este método de construcción se recorren todos los objetos de la escena,

Algoritmo 5 Pseudocódigo del algoritmo menos eficiente de construcción de la grilla uniforme.

```
para todo voxel en grilla hacer
  para todo obj en listaObjetos hacer
    //En coordenadas de grilla
    bbObj = calcularBoundingBoxObjeto(obj);
    si (voxel  $\cap$  bbObj)  $\neq \emptyset$  entonces
      agregarObjetoEnVoxel(voxel, obj);
    fin si
  fin para
fin para
```

donde para cada uno ellos se calcula (en coordenadas de grilla) una caja alineada a los ejes que lo envuelve. A partir de la caja se obtienen voxels candidatos a solaparse con el objeto. Para cada voxel candidato se prueba el solapamiento caja-objeto y si da positiva se agrega el objeto al voxel que dio origen a la caja. La optimización introducida con respecto al método mostrado en el Algoritmo 5 permite desechar una cantidad importante de voxels que no tienen intersección con el objeto. Si no consideramos el peor caso de este método (cuando todos los objetos de la escena se solapan con todos los voxels de la grilla), que es muy poco probable, este método siempre supera al anterior en cantidad de pruebas de intersección voxel-objeto.

Algoritmo 6 Pseudocódigo del algoritmo eficiente de construcción de la grilla uniforme.

```
para todo obj en listaObjetos hacer
  //En coordenadas de grilla
  bbObj = calcularBoundingBoxObjeto(obj);
  para todo voxel  $\subset$  bbObj hacer
    //En coordenadas de mundo
    voxelMundo = transformarCoorMundo(voxel);
    si boundingBoxOverlapObject(voxelMundo, obj) entonces
      agregarObjetoEnVoxel(voxel, obj);
    fin si
  fin para
fin para
```

Recorrido de la grilla

El algoritmo para la recorrida de la grilla está basado en el renderizado de una línea en pantalla que se implementa en las tarjetas gráficas. El recorrido de un rayo a través de la grilla de subdivisión espacial genera una lista de voxels, estos voxels son los 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 al siguiente voxel realizando unas pocas sumas y evaluaciones de condición.

La implementación se realiza en 2 pasos, primero se computa el cálculo del voxel 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 voxel.

El voxel inicial se calcula de manera sencilla intersecando el rayo con el cubo que acota toda la escena que se utiliza para generar la grilla, con el punto de intersección se calcula cual es el voxel 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 los voxels en x, y, z se obtiene la distancia que debe recorrer el rayo para poder alcanzar el siguiente voxel, esta se calcula para cada una de las 3 componentes y se almacena en una variable, así como también se hace con las derivadas.

Algoritmo 7 Pseudocódigo del algoritmo para avanzar en los voxels.

```
DM = Minimo(DX, DY, DZ);
si (DM == DX) entonces
    DX = DX + incrementoX;
    VoxelActualX = VoxelActualX + signo(incrementoX) * 1;
fin si
si (DM == DY) entonces
    DY = DY + incrementoY;
    VoxelActualY = VoxelActualY + signo(incrementoY) * 1;
fin si
si (DM == DZ) entonces
    DZ = DZ + incrementoZ;
    VoxelActualZ = VoxelActualZ + signo(incrementoZ) * 1;
fin si
```

Como se muestra en el Algoritmo 7, la forma en que se calcula el siguien-

te voxel es en base a 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 al siguiente voxel en el eje X, para pasar al siguiente voxel en el eje Y y para pasar al siguiente voxel en el eje Z. La mínima de estas distancias determina cual de los tres incrementos es el que hay que realizar, el voxel que esté más cerca es el siguiente, por lo tanto hay que incrementar 1 en la dirección determinada por dicho mínimo.

4.2.4. 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 B.

Versión 1

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 con tope *objetos* (el tope es *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 y en cada una de sus entradas 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 4.3. 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 subdivisión espacial uniforme con dimensiones $2 \times 2 \times 3$ a la escena se generan

12 voxels, 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 todos los voxels de la subdivisión y mediante las entradas del vector “voxels” se accede a cada una de ellas. El caso especial en que la lista de objetos de un voxel es vacía se marca con un valor especial en el vector “voxels”.

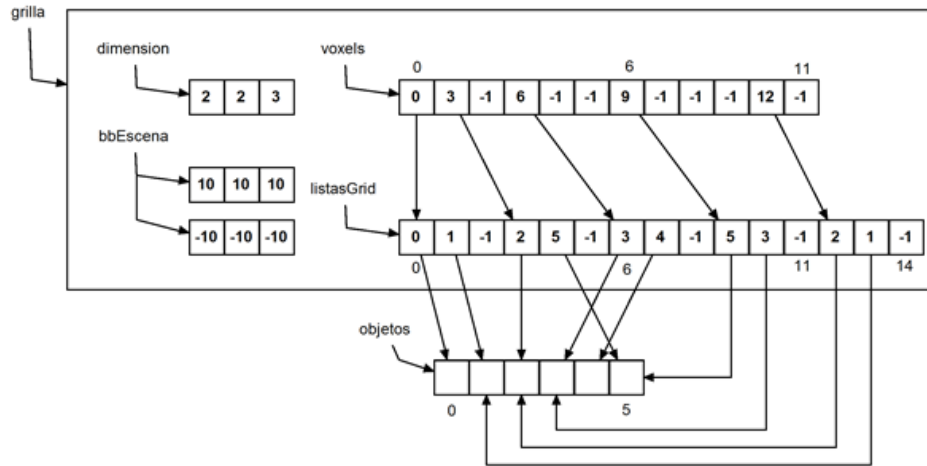


Figura 4.3: Ejemplo de estructura *UniformGrid*.

Versión 2

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 modificada 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 el algoritmo puede tornarse interactivo sin mayores costos computacionales.

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.

4.2.5. Paralelización del algoritmo

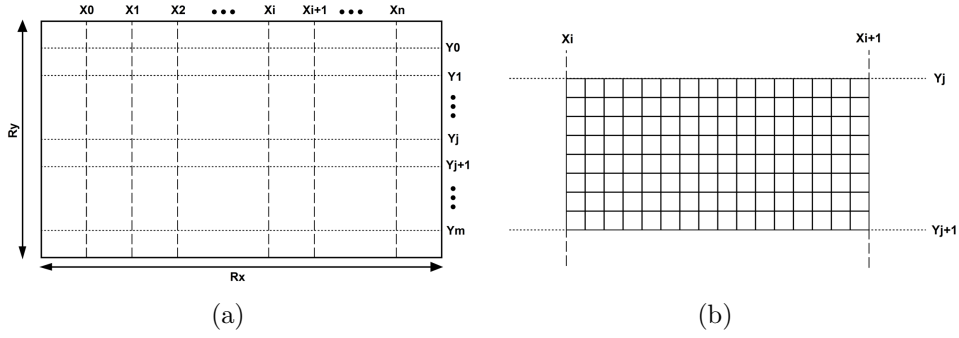


Figura 4.4: Subdivision de la imagen para lograr paralelismo.

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 pixeles. 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 $\frac{R_x}{n} \times \frac{R_y}{m}$ pixeles. 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 pixeles 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 pixel de la imagen es procesado por un hilo de ejecución diferente, la cantidad de hilos por bloque es igual a la cantidad de pixeles 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 es dividida en una grilla de 40×60 celdas.

Mediante esta forma de descomponer el problema el raytracing para CUDA puede escalar fácilmente en el número de procesadores de la GPU sin necesidad de re-compilar 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 pixels que pueden ser procesados por un mismo bloque, esto es un limite 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 consume 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 Raytracing 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 y de esta forma no se necesita del apoyo de una pila. La forma de simplificar el árbol es exigiendo una pre-condición sobre la escena de entrada, la misma exige 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 hubo que hacer modificaciones al algoritmo original, generando que cada rayo sea el encarga-

do de hacer su trayectoria de manera iterativa. Para esto hay que hacer que en cada iteración, en caso de que haya algún tipo de reflejo o refracción, el rayo tenga que ser modificado para que en la siguiente iteración se trace, de manera de simular la interacción con materiales con reflexión o refracción.

4.2.7. Estructura del núcleo de Raytracing

Considerando la arquitectura del sistema, el núcleo principal del Raytracing para GPU se encuentra implementado dentro del componente “Raytracing”.

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 los voxeles de la subdivisión espacial, deben estar almacenados en un tipo de memoria que tenga tiempo de lectura bajo y puede 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, los voxeles de la grilla de subdivisión 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.

A 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 raytracing a la GPU, se procede a la invocación del *kernel* encargado de calcular los rayos primarios. Como se muestra en la Figura 4.5, 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.

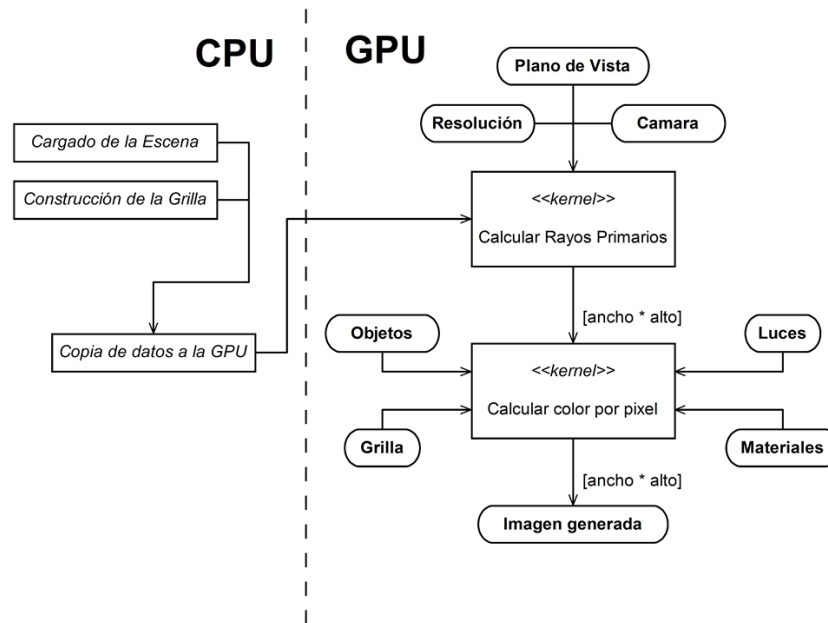


Figura 4.5: Estructura del algoritmo de raytracing implementado en CUDA.

Los rayos primarios son datos de entrada para el *kernel* encargado de calcular el color de cada pixel. 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 raytracing. Los datos de entrada necesarios para calcular el color de cada pixel se muestran en la Figura 4.5 y son leídos desde memoria de textura, cuyo tiempo de vida es igual al de la aplicación CUDA.

La forma de invocación al *kernel* principal es según la división en bloques de la imagen que se presentó anteriormente 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.

4.3. Versiones

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

4.3.1. Versión 1 - RT(GPU)

- Implementación del 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 solo *kernel* de CUDA.
- Las operaciones sobre vectores se hacen usando las operaciones propias de CUDA.
- Restricción de la versión: las escenas deben tener una luz puntual como máximo.
- Restricción de la versión: las escenas no pueden tener objetos reflexivos y transparentes a la vez.
- Restricción de la versión: la sombra proyectada por un objeto transparente no tiene en cuenta el color del objeto.

- Restricción de la versión: no explota al máximo la estructura de memoria de la GPU.
- Restricción de la versión: sufre el problema que genera el sistema operativo al ejecutar un *kernel* por mas de 5 segundos.

4.3.2. Versión 2 - RT(GPU-JM)

- Idem. que la versión RT(GPU).
- Se elimina la restricción “no explota al máximo la estructura de memoria de la GPU”, usando la jerarquía de memoria (memoria de textura y constante principalmente) para almacenar las estructuras de datos que representan la escena a procesar por el algoritmo.

4.3.3. Versión 3 - RT(GPU-JM-IR)

- Idem. que la versión RT(GPU-JM).
- Se cambia 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 Raytracing 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 referencia sobre arquitecturas tradicionales para comparar desempeño.

Capítulo 5

Experimentación

Esta sección detalla las estrategias tomadas para evaluar el trabajo realizado, explicando las pruebas realizadas, así como también los resultados obtenidos en estas pruebas. También se describen otros trabajos y se compara contra estos para evaluar la distancia en cuanto a eficiencia de nuestro algoritmo comparado con otras implementaciones similares.

5.1. Estrategias de evaluación de calidad de imagen

Para evaluar la solución implementada 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 evaluaba la calidad de los generadores de imágenes. Como resultado de esta investigación no se pudieron determinar estrategias solidas para evaluar la calidad de las imágenes generadas. Si bien no se encontraron algoritmos que se adaptaran específicamente a las necesidades del trabajo, se presentan a continuación ideas interesantes que podrían servir a la 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 [22] y analizando el resto de la información disponible [36] [16] [17] [29]. Cabe señalar que si bien el trabajo de Avcibas y Sankur [22] 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 los más simples, calculan la diferencia entre 2 imágenes tomando como referencia que un pixel en una imagen se corresponde con el mismo pixel de la imagen objetivo y, dependiendo de cuál de los algoritmos se trate, calcula alguna ponderación de los píxeles para retornar un valor que indicará 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 se basa en notar que las imágenes en general en su composición de aristas (bordes que separan los objetos de la imagen), estas pueden ser utilizadas para el análisis de calidad de la imagen. 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, esto pertenece al área de tratamiento de señales, base del tratamiento de imágenes. 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 pixel sus vecinos en una cantidad de niveles arbitraria. Estos píxeles en caso de que difieran de alguna manera (esto varía para cada algoritmo dentro

de la familia) modificaran, no solo la calidad de ellos mismos como analizan las estrategias de diferencia por pixel, sino también la calidad de los vecinos, dado que no será lo mismo, por ejemplo, un pixel negro entre píxeles rojos que un pixel negro entre píxeles blancos.

Por último, los métodos basados en el análisis de la percepción humana para brindar una medida de calidad de la imagen generada. Este tipo de estrategias utilizan los modelos que se han generado para la percepción del ojo humano declarando 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, videojuegos, generación de imágenes fotorealistas, entre otras.

Como se puede ver, todas estas estrategias son para comparar imágenes y no son aplicables directamente en el contexto de este proyecto donde el objetivo es evaluar la calidad de las imágenes generadas por un algoritmo.

En evaluación de imágenes generadas es que se basa el artículo de Dirik, Bayram, Sencar y Memon [16]. Pero plantea, al contrario de lo que se requiere en este proyecto, detectar que imágenes son generadas por un algoritmo de generación de imágenes y cuales son imágenes reales tomadas con una cámara. Aunque el objetivo que persigue este artículo es muy similar al que se persigue en el análisis de esta sección, el área de trabajo de este proyecto no son las imágenes fotorealistas dado que el modelo abordado, Raytracing, no es un modelo tan preciso. Por esto es que no se podrían utilizar los métodos propuestos en el artículo de Dirik, Bayram, Sencar y Memon.

Las dos aproximaciones que se acercan más a los requerimientos de evaluación del proyecto son el planteado por Dirik, Bayram, Sencar y Memon [16] y los basados en las capacidades de percepción del sistema visual humano, que apuntan a evaluar que imágenes son iguales para el ojo humano. En ambos casos las estrategias no permiten evaluar la calidad en la forma que se requiere.

5.2. Casos de prueba

Para probar el desempeño de los algoritmos de generación de imágenes implementados es necesario disponer de un conjunto de casos de prueba con distintas características.

La comparación con implementaciones similares es importante para establecer la calidad del algoritmo de Raytracing 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 Raytracing. Dentro de esta clase de escenas externas al proyecto, hay escenas usadas en todo proyecto 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 Raytracing 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 Raytracing. 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.

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 diferencia entre ellos es la posición de los objetos (cada uno esta 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 una referencia que será usada de aquí en más.

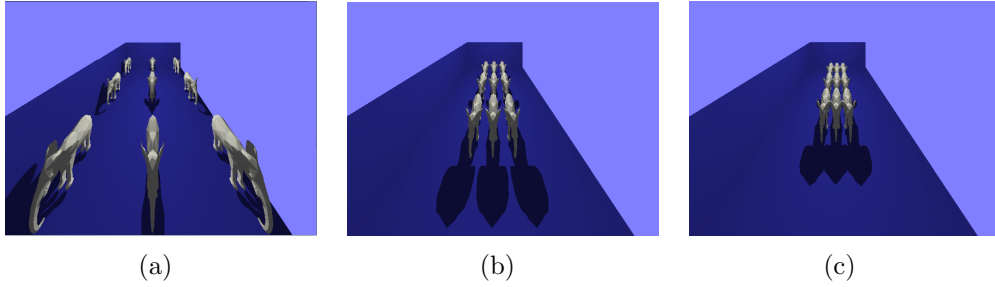


Figura 5.1: Escenas con distinta disposición espacial de los objetos.

Ref	Objetos	Luces	Triángulos	Archivo	Imagen
DIDT_I	9	1	10338	elefantesChicosDistUniforme.obj	5.1(a)
DIDT_II	9	1	10338	elefantesChicosDistNoUniforme.obj	5.1(b)
DIDT_III	9	1	10338	elefantesChicosDistNoUniformeSOLAP.obj	5.1(c)

Tabla 5.1: Datos de entrada para pruebas de distribución.

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 o 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

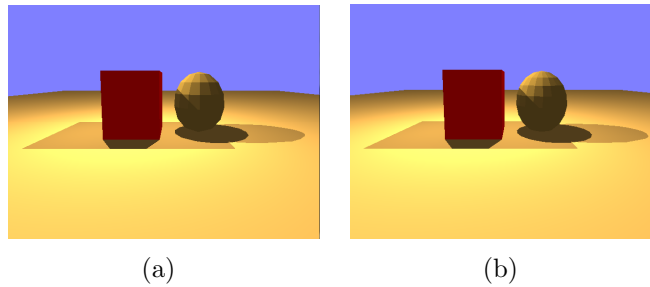


Figura 5.2: Imágenes de los casos de prueba de cantidad de primitivas.



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

casos de prueba, en especial en la primer columna se muestra una referencia que será usada de aquí en más.

Ref	Objetos	Luces	Triángulos	Archivo	Imagen
PRLI	2	2	194	cajaEsfera1.obj	5.2(a)
PRLII	2	2	274	cajaEsfera2.obj	-
PRLIII	2	2	348	cajaEsfera3.obj	-
PRLIV	2	2	482	cajaEsfera4.obj	-
PRLV	2	2	606	cajaEsfera5.obj	5.2(b)

Tabla 5.2: Datos de entrada para pruebas de cantidad de primitivas.

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 Raytracing similares. Por ello se buscaron algoritmos que se ajustaran al modelo de Whitted, implementados sobre CUDA por desarrolladores de Raytracing.

Los integrantes del grupo de Computación Gráfica del *Alexandra Institute* de Dinamarca [15] implementaron un algoritmo de Raytracing 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 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 todo proyecto 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 el hecho de que el algoritmo implementado en este proyecto soporte este tipo de casos de prueba, que por lo general se componen de una cantidad importante (100.000) de primitivas, es importante.

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.

Ref	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	69698	StanfordBunny.obj	5.3(d)

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

5.3. Equipos utilizados

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.

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

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

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 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	Núcleos	Clock (MHz)	Shader clock (MHz)	Memory clock (MHz)
9500M	4	32	475	950	400
9500M	4	32	500	1250	400
GTX260	27	216	576	1242	999

Tabla 5.5: Características de las GPUs utilizadas.

5.4. Experimentos

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 dos algoritmos implementados, uno para CPU y el otro 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. [30] 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 Raytracing, que no permite variar la misma. En el caso del artículo de Günther la resolución quedo 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 [27], la cual indica que la resolución sea $3\sqrt[3]{N}$ voxels a lo largo del eje más corto, donde N es el número de triángulos de la escena. Después varias pruebas realizadas 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

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 PRI.I, PRI.II, PRI.III, PRI.IV, PRI.V en el Equipo GTX260 sobre GPU.

óptimo (o grilla optima), así mismo se muestran también otros tamaños de grilla para cada escena.

5.4.1. Comparación entre versiones

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 ejecuta cada caso de prueba en cada una de las versiones del algoritmo utilizando su grilla óptima.

Para determinar la grilla óptima para cada escena de prueba se usa la Versión RT(GPU-JM-IR) del algoritmo para GPU, ejecutando en el Equipo GTX260. En la Tabla 5.11 se muestran los resultados obtenidos al ejecutar los casos de prueba sobre GPU. Observando los resultados se puede concluir que la grilla optima 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 esta construida la escena aumenta el tiempo de generación de imagen, y por lo tanto disminuyen los FPS. Esto se pensaba antes de ejecutar estos casos de prueba y fue corroborado por los mismos. También se pensaba que al aumentar la cantidad de primitivas de la escena aumentaría la cantidad de voxels que debía tener la grilla optima, 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.

Una vez determinada la grilla optima para cada caso de prueba, se eje-

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.I, PRI.II, PRI.III, PRI.IV, PRI.V en el Equipo GTX260 sobre GPU para cada versión del algoritmo.

cuta cada caso utilizando su grilla optima en cada una de las versiones del algoritmo para GPU, en la Tabla 5.12 se muestran los resultados obtenidos.

Los resultados de la Tabla 5.12 reflejan la evolución positiva del algoritmo ya que a medida que se fue sofisticando el algoritmo se incrementaron los FPS para todos los casos de prueba.

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 Raytracing 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 así (en promedio) tres veces y media más rápido que el algoritmo que no la usa.

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 Raytracing 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 5.6 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 de número de versión del algoritmo implica un aumento en la cantidad de FPS, esta disminución del tiempo de generación de imágenes no implica perdida de calidad de imagen.

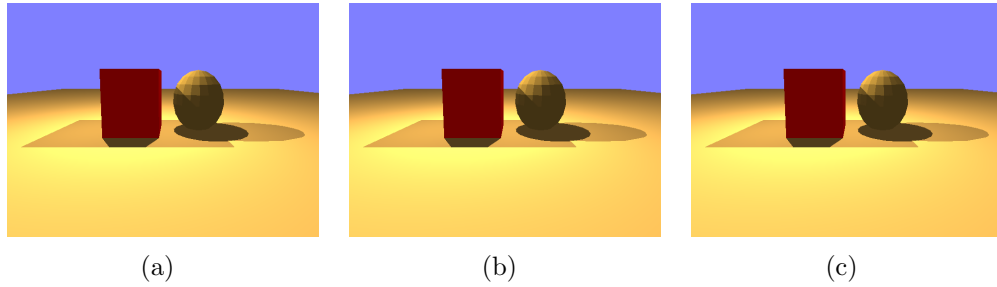


Figura 5.4: Render de PRLV en GPU con la Versión RT(GPU), RT(GPU-JM) y RT(GPU-JM-IR) respectivamente.

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.

5.4.2. Comparación entre C y 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 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

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.

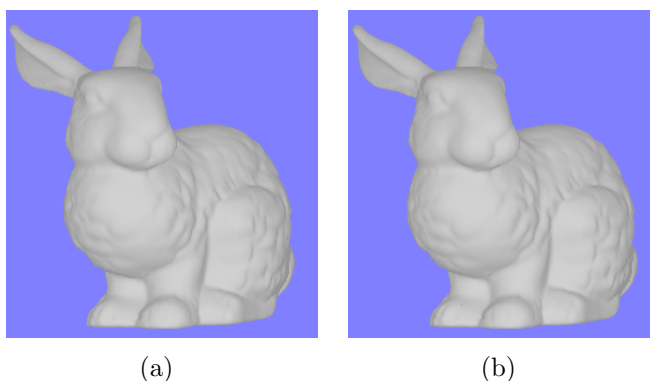


Figura 5.5: Render de BUNNY en CPU y en GPU respectivamente.

buscar una debilidad de la estructura de aceleración. La debilidad de la estructura de subdivisión espacial uniforme se da cuando los objetos de la escena están distribuidos de forma no uniforme en la misma. 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 que se encuentra y que es para el caso de este tipo de escenas, es que cuanto mas uniformemente distribuidos en la escena estén los objetos, más sombra arrojan. Como el calculo de sombra es un cálculo computacionalmente costoso, se cree que este costo contrarresta al beneficio que brinda la grilla cuando existe distribución espacial uniforme de los objetos en la escena.

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

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.

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 el ojo humano no percibe diferencia alguna, entonces el algoritmo para GPU logra una muy buena aceleración con respecto al que ejecuta en CPU y sin pérdida en la calidad de imagen.

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 optima para cada caso de prueba y en la segunda se ejecuta cada caso de prueba en cada una de las versiones del algoritmo utilizando su grilla optima.

Para determinar la grilla optima para cada escena de prueba se usa la Versión RT(GPU-JM-IR) del algoritmo para GPU, ejecutando en el Equipo GTX260. En la Tabla 5.11 se muestran los resultados obtenidos al ejecutar los casos de prueba sobre GPU. Observando los resultados se puede concluir que la grilla optima 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 esta construida la escena aumenta el tiempo de generación de imagen, y por lo tanto disminuyen los FPS. Esto se pensaba antes de ejecutar estos casos de prueba y fue corroborado por los mismos. También se pensaba que al aumentar la cantidad de primitivas de la escena aumentaría la cantidad de voxels que debía tener la grilla optima, 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 5.11: FPS de PRI_I, PRI_II, PRI_III, PRI_IV, PRI_V en el Equipo GTX260 sobre GPU.

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 5.12: FPS de PRI_I, PRI_II, PRI_III, PRI_IV, PRI_V en el Equipo GTX260 sobre GPU para cada versión del algoritmo.

Una vez determinada la grilla optima para cada caso de prueba, se ejecuta cada caso utilizando su grilla optima en cada una de las versiones del algoritmo para GPU, en la Tabla 5.12 se muestran los resultados obtenidos.

Sin duda los resultados de la Tabla 5.12 reflejan que a medida que se fue mejorando el algoritmo se incrementaron los FPS para todos los casos de prueba.

El incremento del tiempo de generación de imagen es mayor en el primer cambio de versión, esta diferencia esta 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 Raytracing 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 así (en promedio) tres veces y media más rápido que el algoritmo que no la usa.

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 dismi-

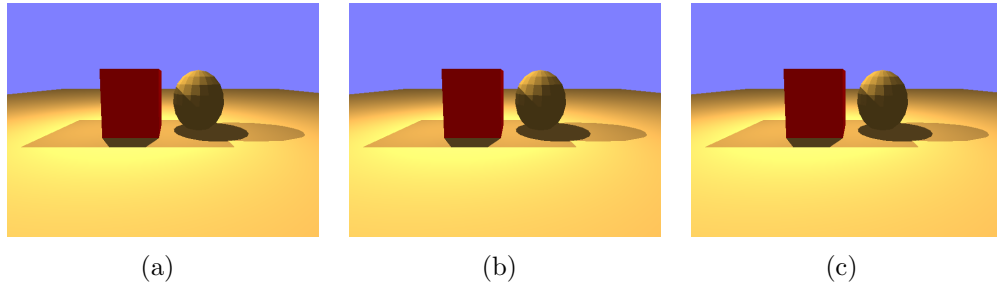


Figura 5.6: Render de PRLV en GPU con la Versión RT(GPU), RT(GPU-JM) y RT(GPU-JM-IR) respectivamente.

ucción del tiempo de generación de la imagen. Considerando los resultados obtenidos esta mejora del algoritmo de Raytracing 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 5.6 se muestran los renders del caso de prueba PRLV 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 de número de versión del algoritmo implica un aumento en la cantidad de FPS, esta disminución del tiempo de generación de imágenes no implica pérdida de calidad de imagen.

5.4.3. Comparación entre 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 del 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 5.13, 5.14 y 5.15 se muestran los resultados obtenidos para los equipos 9500M, 9600M y GTX260 respectivamente.

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

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 5.13: 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 5.14: FPS de DRAGON, BUDDHA y ALEXANDRA en el equipo 9600M sobre GPU.

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.

Todas la pruebas realizadas en esta comparación confirmaron que cuanto más procesadores posea la GPU más rápido será la generación de imágenes mediante el algoritmo de Raytracing para CUDA. 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 un poco más de seis veces más procesadores que la del equipo 9600M. 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

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
ALEXANDRA	1x1x1	3x3x3	6x6x6	10x10x10	15x15x15	30x30x30
	28.0	49.3	71.2	67.6	62.5	49.4

Tabla 5.15: FPS de DRAGON, BUDDHA y ALEXANDRA en el equipo GTX260 sobre GPU.

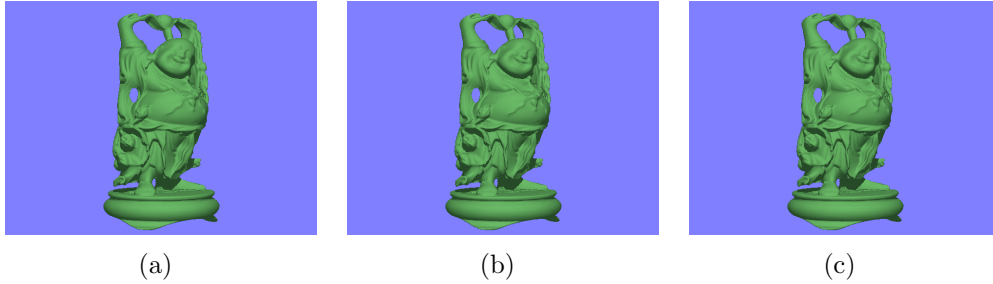


Figura 5.7: Render de BUDDHA en GPU sobre los equipos 9500M, 9600M y GTX260 respectivamente.

el equipo 9600M y el de la Figura 5.7(c) con el equipo GTX260. No se observan diferencias importantes entre las imágenes generadas por los diferentes equipos utilizados en el proyecto. 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 pérdida de calidad de las mismas.

5.4.4. Comparación con otras implementaciones

En el trabajo de Günther et al. [30] se desarrolló una implementación del algoritmo de Raytracing 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 tiempos de generación de imágenes.

Se deben considerar algunas excepciones con respecto a la igualdad de las escenas, la escena BUNNY esta 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. El material del objeto principal de la escena no pudo ser reproducido con exactitud debido a que

en el trabajo no se brinda esta información.

Debido a que en el trabajo de Günther et al. solo 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 resultados comparables. En el trabajo de Johannes Günther 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, la Kd-Tree 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 tiempo posible, dicho tamaño se definió empíricamente en la Sección 5.4.2.

La GPU utilizada en el trabajo de Günther es nVidia modelo GeForce 8800 GTX, la cual posee 112 núcleos. En el marco de este proyecto la GPU que más se adapta para este caso es la del equipo GTX260, ya que si bien posee más cantidad de núcleos de procesamiento esta cantidad se ve compensada por el uso de la estructura Kd-Tree por parte del algoritmo 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 6,1 FPS, mientras que el rendimiento del algoritmo del trabajo de Günther es de 5,9 FPS. 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.

Otro caso de prueba que permite evaluar un resultado positivo del algoritmo implementado para GPU es la escena ALEXANDRA.

La resolución de la imagen que genera el algoritmo de Raytracing 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 compara-

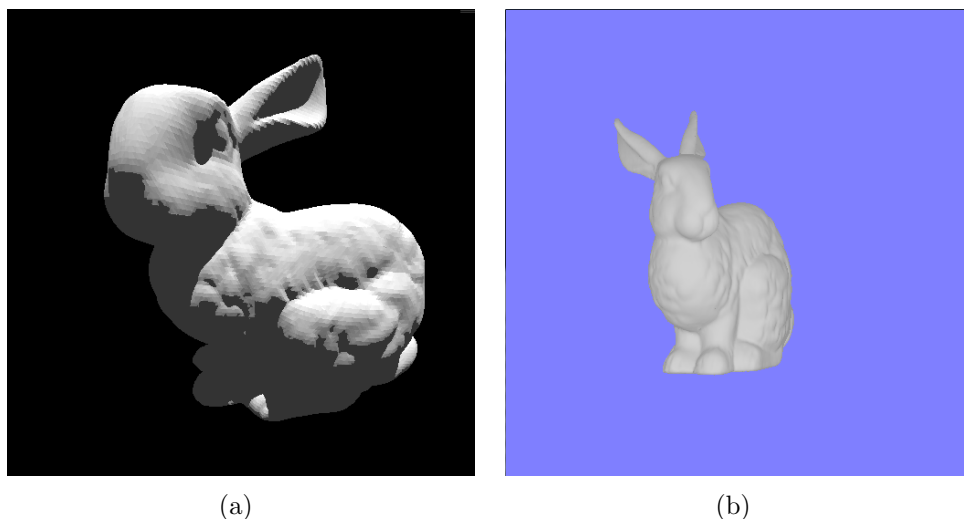


Figura 5.8: Render de BUNNY en el trabajo de Günther et al. y en este proyecto respectivamente.

ción. El Raytracing del *Alexandra Institute* no usa estructura de aceleración espacial, mientras que el algoritmo implementado en este proyecto usa un tamaño de grilla de 6 voxels por dimensión. Este tamaño de grilla genera el mejor rendimiento del algoritmo y fue hallado empíricamente en la sección 5.4.3.

Debido a que se tiene acceso al ejecutable del algoritmo del *Alexandra Institute* 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 raytracing del *Alexandra Institute* es de 11,7 FPS. Es necesario recalcar que el algoritmo implementado en este proyecto renderiza escenas genéricas, es decir, no fue concebido para generar imágenes de solo 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

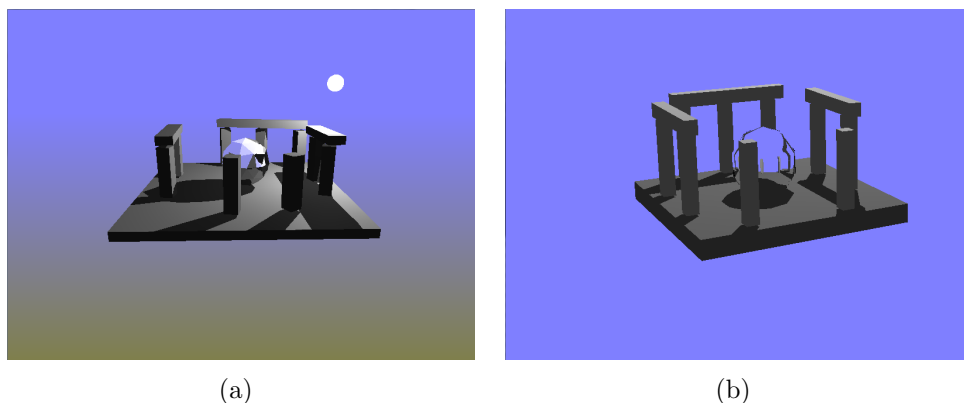


Figura 5.9: Render de la aplicación del *Alexandra Institute* y render de ALEXANDRA en GPU sobre Equipo 9600M usando la Versión RT(GPU-JM-IR), respectivamente.

en el código fuente del mismo, lo cual brinda mayor eficiencia para el caso puntual. 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 raytracing 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 que 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.

Capítulo 6

Conclusiones y Trabajo a futuro

6.1. Conclusiones

Falta completar...

6.2. Trabajo a futuro

Falta completar...

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. Version 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 descritos 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 [27] 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 [27].

En la Figura B.1 la función de costo es evaluada en los puntos a, b, \dots, 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.

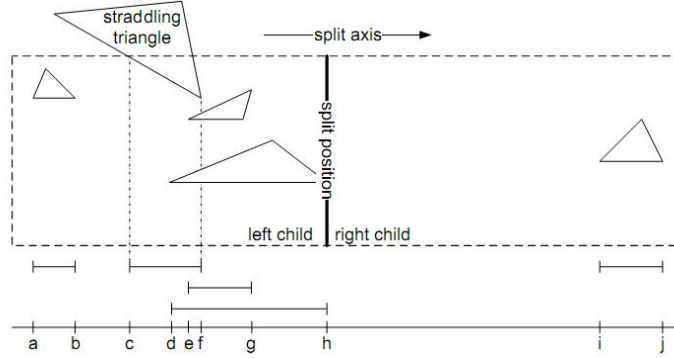


Figura B.1: Ejemplo de búsqueda del plano de corte considerando solo un eje.

En el caso que un objeto no esté completamente contenido en el voxel que se está 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*” [32]. 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 [27] usan esta técnica para la construcción de la estructura kd-tree y obtuvieron muy buenos resultados.

En el Algoritmo 12 se puede ver un pseudocódigo 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-voxels (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 subárbol 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 voxels en el orden en que son visitados por el rayo. Esto permite que se pueda parar el algoritmo de recorrida de voxels tan pronto como se encuentre la intersección rayo-objeto más cercana al observador.

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

```
si numObjetos(voxel)  $\leq$  MIN_OBJETOS entonces
    se retorna nueva hoja con su lista de objetos
fin si
si profundidad(arbol)  $\geq$  PROFUNDIDAD_MAX entonces
    se retorna nueva hoja con su lista de objetos
fin si
mejorCorte  $\leftarrow \emptyset$ 
mejorCosto  $\leftarrow \infty$ 
para todo eje in  $\{x, y, z\}$  hacer
    posiciones  $\leftarrow []$ 
    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  $\leftarrow (p, eje)$ 
            mejorCosto  $\leftarrow$  costo(p)
        fin si
    fin para
fin para
(voxelIzq, voxelDer)  $\leftarrow$  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  $\leftarrow$  construir(voxelIzq)
hijoDer  $\leftarrow$  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 [27]. Una posible solución es emplear una estrategia diferente de recorrido de los voxeles. Se puede emplear la estrategia usada por Foley y Sugerman [31], 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 descende, 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.

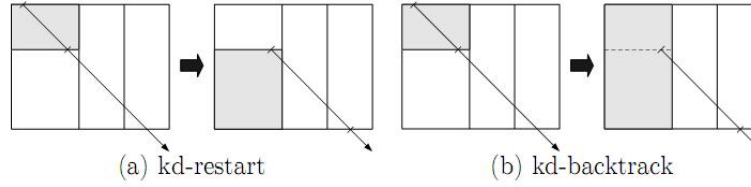


Figura B.2: Actualización del intervalo en cada variante del algoritmo de atravesado.

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 Raytracing 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 [21]. 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 subárbol de la estructura kd-tree. Si se arranca el algoritmo siempre desde la raíz se esta analizando muchas veces un subárbol que ya se sabe que no es atravesado por el rayo. Esta técnica permite recomenzar el algoritmo desde el subárbol más profundo que encierra al rayo, y de esta manera no se vuelven a analizar subárboles 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 [27]. 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 subárboles 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 [25].
Seudocódigo 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 [27] 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 subárbol izquierdo y por último el subárbol derecho, como se muestra en el ejemplo de la Figura B.3. Los nodos

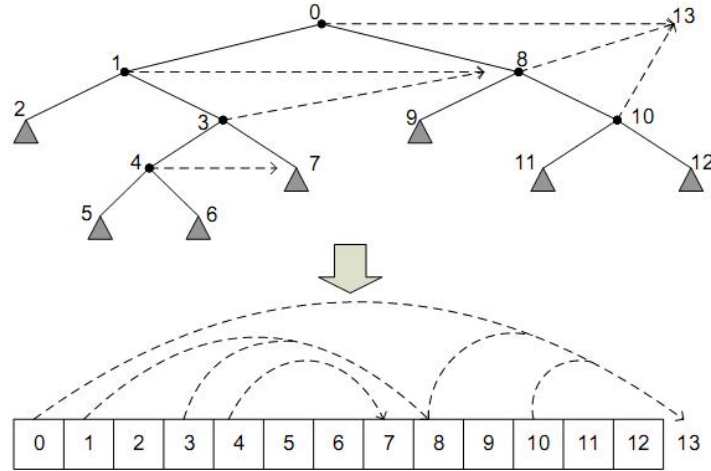


Figura B.3: Ejemplo de codificación de los datos para atravesar una estructura BVH.

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 [27] 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 subárbol 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 subárbol 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 volúmenes 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

```

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=455755&id=14626995>.
- [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 SDL (Simple Directmedia Layer). <http://www.libsdl.org/>.
- [15] Blog del grupo de Computación Gráfica del Alexandra Institute de Dinamarca. <http://cg.alexandra.dk/>.
- [16] 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.
- [17] B. Ghanem, E. Resendiz, and N. Ahuja. Segmentation-based perceptual image quality assessment (spiq). In *ICIP08*, pages 393–396, 2008.
- [18] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl.*, 7(5):14–20, 1987.
- [19] Donald Hearn and M. Pauline Baker. Computer graphics / d. hearn, m.p. baker. *Prentice-Hall*, 1988.
- [20] Henrik Wann Jensen. Realistic Image Synthesis Using Photon Mapping. *AK Peters, Ltd.*, 2001.
- [21] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive Rendering with Coherent Ray Tracing. *Saarland University*, 2001.
- [22] Ismail Avcibas, Bülent Sankur. Statistical Analysis of Image Quality Measures. *Department of Electrical and Electronic Engineering, Bogaziçi University, Istanbul, Turkey*, 2001.
- [23] James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes, Richard L. Phillips. Introducción a la graficación por computador. *Addison-Wesley Iberoamericana, S.A.*, 1996.
- [24] John Amanatides and Andrew Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. 1987.

- [25] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. *SIGGRAPH Comput. Graph.*, 20(4):269–278, 1986.
- [26] Matt Pharr, Greg Humphreys. Physically based rendering. *Morgan Kaufman*, 2004.
- [27] 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.
- [28] Timothy John Purcell. Ray tracing on a stream processor. Technical report, Stanford University, Stanford, CA, USA, 2004. Adviser-Hanrahan, Patrick M.
- [29] D.V. Rao and L.P. Reddy. Image quality assessment based on perceptual structural similarity. In *PReMI07*, pages 87–94, 2007.
- [30] 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.
- [31] Jeremy Sugerman Tim Foley. Kd-tree acceleration structures for a gpu raytracer. In *Graphics Hardware*, pages 15–22, 2005.
- [32] Vlastimil Havran, Jan Prikryl, Werner Purgathofer. Statistical comparison of ray-shooting efficiency schemes. Technical report, Institute of Computer Graphics, Vienna University of Technology, 2000.
- [33] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004.
- [34] 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.
- [35] Whitted Turner. An improved illumination model for shaded display. *Communications of the ACM*, 1980.
- [36] 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.