Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXXI, págs. 3165-3179 (artículo completo)
Alberto Cardona, Paul H. Kohan, Ricardo D. Quinteros, Mario A. Storti (Eds.)
Salta, Argentina, 13-16 Noviembre 2012

IMPLEMENTACIÓN DE UN RAYTRACER EN UNIDADES DE PROCESAMIENTO GRÁFICO UTILIZANDO BOUNDING VOLUME HIERARCHY

Leonardo Scandolo[a], **Cristian García Bauza**[b,c], **Juan D'Amato**[b,c] **y Marcelo Vénere**[b,d]

[a]*Universidad Nacional de Rosario, Argentina*

[b]*PLADEMA, Universidad Nacional del Centro, Tandil, Argentina* *

[c]*Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)*

[d]*Comisión Nacional de Energía Atómica (CNEA)*

leonardo@fceia.unr.edu.ar , {crgarcia,jpdamato,venerem}@exa.unicen.edu.ar

Palabras Clave: Raytracing, GPGPU, BVH.

Resumen. Los métodos de *ray tracing* permiten generar representaciones gráficas de escenarios tridimensionales con gran calidad, ya que modelan naturalmente sombras, reflexiones y refracciones además de la emisión difusa. Estos algoritmos tienen un costo computacional muy elevado, pero pueden adaptarse para hacer uso del poder de cómputo de arquitecturas masivamente paralelas y de alto rendimiento como las placas gráficas (GPUs).

La implementación clásica de estos algoritmos hace uso de técnicas de clasificación geométrica para reducir al mínimo la cantidad de cómputos de intersecciones entre rayos y triángulos. A su vez, los datos de materiales, tales como texturas, deben organizarse coherentemente para reducir los tiempos de lectura.

En este trabajo se propone utilizar una clasificación de los datos con un árbol de Jerarquía de Volúmenes (BVH) y se describe su implementación en una arquitectura masivamente paralela. Se presenta un *pipeline* de procesamiento implementado en *OpenCL* que permite generar imágenes de alta calidad con tasas de refresco interactivas, y se exponen resultados en resoluciones de 512^2 y 1024^2 píxeles a partir de escenas del orden de 10^5 triángulos.

* http://www.plademe.net

1. INTRODUCCIÓN

La técnica de *ray tracing* para generar imágenes a partir de una escena en 3 dimensiones fue descubierta en la década del 60' por Appel (1968). En su forma más simple el algoritmo modela una cámara en una escena virtual. Por cada píxel de la pantalla se emite un rayo desde el punto de vista de la cámara y se calcula el punto de impacto de dicho rayo con la escena. El color del píxel resulta de la aplicación de un modelo de iluminación al punto de impacto del rayo con la escena. Las escenas son generalmente descriptas por mallas de triángulos, materiales y texturas.

A pesar de ser conocido hace alrededor de 50 años, este algoritmo sigue siendo relevante al día de hoy dado que es la base de la mayoría de los sistemas de renderizado que producen imágenes fotorrealistas. Sin embargo, a pesar de ser tan conocido, sólo se utiliza en sistemas de renderizado *off-line* debido a que es un algoritmo muy intensivo computacionalmente.

Hoy en día, la estrategia más popular para obtener imágenes en tiempo real sigue siendo el renderizado basado en *"triangle rasterization"*, soportado de forma eficiente por las placas gráficas. El problema de este método, es que para obtener efectos realistas de iluminación tales como sombras, reflejos o transparencias debe recurrir a trucos o aproximaciones, que a pesar de dar buenos resultados elevan el grado de complejidad (y de mantenibilidad) de los sistemas de renderizado. El *ray tracing* presenta una estrategia más simple y obtiene mejores resultados, pero a un costo computacional mucho más elevado. De ahí la importancia de poder adaptarlo para escenas interactivas de tiempo real. En este proyecto se propone utilizar placas gráficas comerciales (GPUs) para acelerar el proceso de todas las etapas de un *ray tracer*.

A pesar de que es descripto con frecuencia como un algoritmo trivialmente paralelizable debido a que cada rayo que representa la cámara es procesado por separado, no es así en la práctica debido a que el hilo que procesa cada rayo puede recorrer la memoria que representa una escena en patrones muy diferentes; esto disminuye la velocidad que puede alcanzar un algoritmo de *ray tracing* dada la alta latencia de búsqueda en memoria.

1.1. GPGPU *Ray tracer*

El modelo de cómputo general en placas gráficas (GPGPU) es de reciente aparición, siendo la biblioteca CUDA de la compañía NVIDIA para desarrollar en sus placas gráficas, una de las primeras incursiones en este campo. A partir del éxito de CUDA, se define el estándar *OpenCL* (Khronos Group (2008)) para estandarizar este tipo de cómputos sobre diferentes plataformas.

OpenCL provee una abstracción para diferentes dispositivos (en general GPUs) que les permite exponer una interfaz común para hacer cómputos con alto nivel de paralelismo. Esta abstracción permite leer y escribir la memoria de dichos dispositivos, así como ejecutar cómputos en ellos. Los algoritmos ejecutados en *OpenCL* son llamados *kernels*. Para ejecutar código en GPUs, *OpenCL* define unidades de cómputo individuales, llamados work items. Estas unidades de cómputo se agrupan en work groups, que funcionan como un línea larga de SIMD (*Single Instruction Multiple Data*), es decir que todos los work items de un mismo work group ejecutarán las mismas instrucciones al mismo tiempo, aunque pueden procesar diferentes datos. Dentro del código de un kernel se puede diferenciar que work item se está ejecutando, lo cual permite a diferentes work items operar sobre diferentes datos.

La elección de *OpenCL* para este proyecto responde a que de trata de una arquitectura basada en un estándar abierto y permite su ejecución en placas gráficas variadas, e incluso en procesadores de múltiple núcleo.

La etapa de generación dinámica de rayos en el *ray tracer* presenta un desafío adicional

para las GPUs, ya que se se deben administrar de forma eficiente el espacio en memoria para alojarlos, así como también la forma en que luego se vuelven a distribuir los threads entre los rayos disponibles. Para esta problemática, proponemos un *pipeline* por pasos que asegura la sincronización de la ejecución y un esquema de asignación de la memoria para asegurar un acceso concurrente correcto a las estructuras.

El trabajo se divide de la siguiente forma: en la siguiente sección se discute brevemente sobre otros trabajos similares. En la sección 3 se resumen los pasos que debe realizar un *ray tracer* que utiliza una estructura de jerarquía de volúmenes, o BVH (*bounding volume hierarchy*). En la sección 4 se presenta el *pipeline* desarrollado y ciertas consideraciones para la implementación en GPU del método. En la sección 5, se discuten los resultados obtenidos en diferentes casos de estudio con diferentes configuraciones de hardware y finalmente se presentan las conclusiones.

2. ANTECEDENTES

Antes de la introducción de bibliotecas para GPGPU, se pueden encontrar ejemplos de *ray tracers* que utilizan bibliotecas gráficas para simular GPGPU y funcionar en tarjetas gráficas. Algunos ejemplos son Christen (2005), Carr et al. (2006) o Bak (2009).

Otros trabajos también presentan resultados sobre plataformas SIMD, pero utilizando CPUs convencionales, como Popov et al. (2006), Shevtsov et al. (2007), Fowler y Collins (2007), Wächter y Keller (2006) o Wald et al. (2007). Las técnicas de estos trabajos suponen un antecedente importante sobre la paralelización de algoritmos de *ray tracing*, aunque los resultados obtenidos no son fácilmente comparables a los presentados en este trabajo, debido a que los algoritmos son implementados en hardware de características esencialmente diferentes.

Los autores de Gunther et al. (2007) presentan una implementación de un *ray tracer* de alto rendimiento en CUDA utilizando BVHs, y obtienen buenos resultados, pero no abordan el problema del cálculo de efectos ópticos reflectivos o refractivos.

El artículo Hapala et al. (2011) muestra diferencias entre un algoritmo que usa una pila para recorrer un BVH y uno que extiende el BVH para no necesitar tal pila (de manera similar al algoritmo que se presenta en este artículo). A través de resultados de una implementación en CUDA se muestra que usar una pila es más eficiente pero presenta limitaciones técnicas. Garanzha y Loop (2010) presentan un *ray tracer* implementado en CUDA que utiliza un BVH de grado 8 como estructura de aceleración, junto con una técnica de reordenamiento de rayos para mantener la coherencia de datos durante el cálculo de rayos secundarios. El trabajo de Zlatuška y Havran (2009) presenta resultados para un *ray tracer* implementado en CUDA, enfocándose en las diferencias de eficiencia y complejidad de utilizar diferentes estructuras de aceleración. Los autores Aila y Laine (2009) presentan una arquitectura de *ray tracing* paralelo basado en reemplazar rayos cuyo procesamiento ya terminó por rayos todavía no procesados en una cola de proceso; demuestran resultados en una implementación de su algoritmo en CUDA.

En D'Amato et al. (2011), los autores presentan una generalización del *ray tracing* que permite variar la estrategia de clasificación de primitivas en placas gráficas, pero no considera los materiales ni texturas de los objetos en el proceso de renderizado.

El sistema *OptiX*, presentado por Ludvigsen y Elster (2010), es un *ray tracer* de calidad comercial, de alto rendimiento y con la capacidad de utilizar CUDA para obtener muy buen desempeño en placas gráficas NVIDIA. Al ser un producto comercial, no se tiene acceso a su código fuente.

En este trabajo, se propone un *pipeline* que considere las diferentes etapas de un *ray tracer* implementado íntegramente en GPU utilizando *OpenCL*, a fin de poder obtener resultados en diferentes configuraciones de hardware.

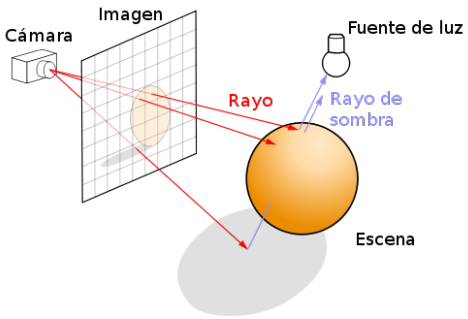


Figura 1: Componentes geométricos de un *ray tracer*

3. ETAPAS DE UN *RAY TRACER*

La mayoría de los *ray tracers* comparten algunas etapas en común que son esenciales para poder producir una imagen a partir de la descripción geométrica de una escena. Dichas etapas se describen brevemente a continuación.

El primer paso para el proceso de *ray tracing* es el de generar los rayos que representan la cámara virtual en la escena. Estos rayos, llamados *rayos primarios*, serán los primeros en intersectar con la geometría de la escena, indicando el punto de impacto, el color de la superficie en ese punto y también información que indica si se producirá reflexión o refracción posterior a la colisión.

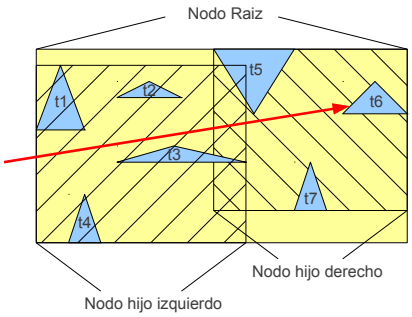
Una vez definidos los rayos primarios, se pasa a la etapa de intersección de rayos. Su función es computar el impacto de los rayos que recibe como entrada la geometría de la escena. Algunas de las características que interesan saber acerca de la intersección normalmente serán el punto de intersección, la superficie impactada y la normal de la superficie en el punto de impacto, entre otras.

Una implementación poco sofisticada de un *ray tracer* debería tratar de intersectar cada uno de miles de rayos producidos con cada uno de miles de triángulos en una escena. Claramente tratar de lograr esto en tiempo real no es posible. Para reducir los tiempos de búsqueda se utilizan estrategias de clasificación, que consisten en subdividir las primitivas de la escena en subconjuntos que pueden rápidamente ser descartados para propósitos de intersección sin tener que probar cada uno de sus elementos. Los kd-trees, octrees, y grillas regulares son algunas de las estructuras que se utilizan para realizar esta clasificación. En este trabajo se utilizan los árboles de jerarquía de volúmenes, o BVH, dado que poseen características que los hacen deseables para su uso en programación GPGPU.

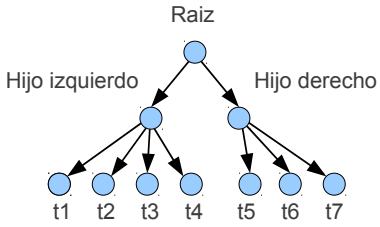
Un BVH es un árbol creado a partir de la escena en el cual la raíz representa la totalidad de las primitivas en dicha escena. Cada nodo interno tiene dos hijos que representan una subdivisión de la geometría de la escena en dos partes. Para cada nodo interno del árbol se guarda un prisma que engloba a todas las primitivas que el nodo representa. Dicho prisma es llamado BBox (*Bounding Box*), y servirá para descartar todas las primitivas del nodo si podemos asegurar que un rayo no impacta con el mismo. Una vez que se llega a tener pocas primitivas asignadas a un nodo, se crea un nodo hoja. Un ejemplo de la estructura de un BVH se puede apreciar en la figura 2b. En la figura 3 se superimpuso los BBoxes de un BVH sobre el modelo de una mano[1].

Para saber si el punto de impacto es iluminado por una fuente de luz, se repite el proceso con un rayo cuyo origen es el punto de impacto y en dirección a la fuente de luz. Si dicho rayo

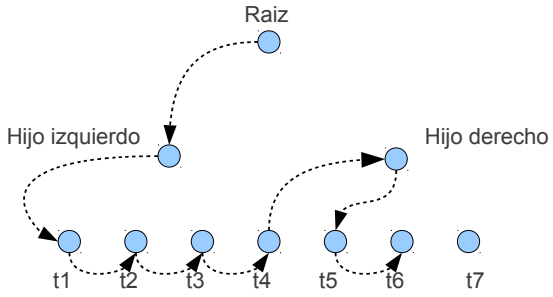
[1]Este modelo, como otros usados para medir la velocidad de la solución fueron obtenidos del repositorio de animación 3D de la universidad de Utah en http://www.sci.utah.edu/~wald/animrep/.



(a) BBoxes para un BVH de dos niveles



(b) Representación del BVH



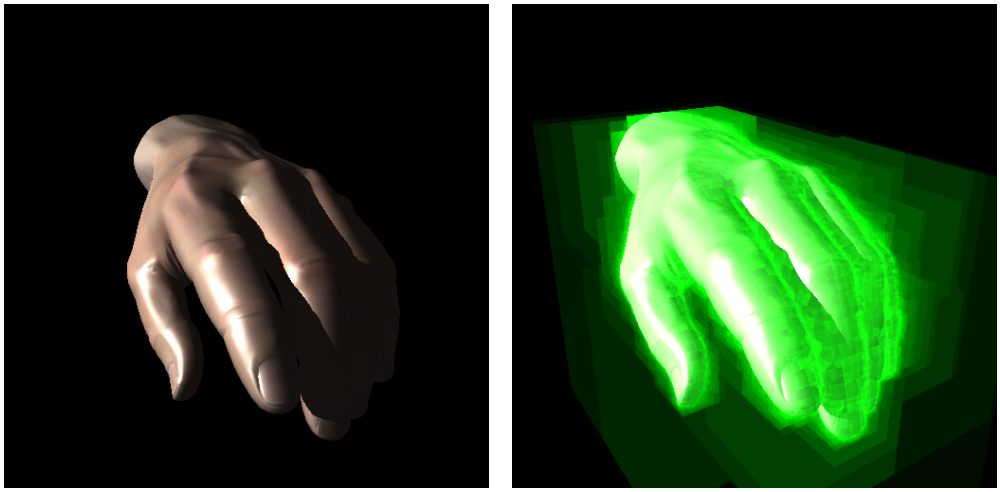
(c) Secuencia de lectura del BVH para el rayo de la figura

Figura 2: BVH

intersecta alguna primitiva en la escena, entonces el punto de impacto no está iluminado. Dado que esta etapa no necesita retornar la intersección más cercana, sino sólo la existencia de alguna, el código es ligeramente más simple.

Una vez que se obtiene la información de intersección de un rayo, en el caso de intersectar primitivas cuyo material es reflectivo o refractivo, se lanzarán nuevos rayos representando la reflexión o refracción de la luz que se produce en esos materiales; dichos rayos son llamados *rayos secundarios*. Los rayos secundarios son procesados de la misma manera que los rayos primarios: se calcula su intersección con la escena y se crean nuevos rayos secundarios a partir de los mismos. Por lo tanto se repite un ciclo que terminará cuando no se creen más rayos secundarios, o se llegue a un máximo permitido de iteraciones.

Por último, a partir de la información de intersección de un rayo primario y los rayos secundarios generados a partir de éste, la última etapa de un *ray tracer* consiste en calcular el color del píxel (o la contribución al color de un píxel si existen más de un rayo por píxel) originado por esos rayos. Para esto se utilizan las propiedades de la superficie impactada, el punto y ángulo de impacto, y las luces en la escena. En el caso que un rayo haya creado rayos secundarios, el color será una composición del color calculado a partir de sí mismo, y el calculado a partir de los rayos secundarios.



(a) Modelo de una mano (b) BBoxes del BVH construido para el modelo

Figura 3: BVH superimpuesto sobre el modelo de una mano

4. IMPLEMENTACIÓN

La arquitectura del *ray tracer* está basada en clases que abstraen las estructuras de datos que se transmiten al dispositivo que ejecuta los kernels de *OpenCL*, y clases que abstraen la lógica de ejecución de los kernels de *OpenCL*. Un diagrama de los módulos del *pipeline* de la solución se puede ver en la figura 4.

La entrada al *pipeline* de *ray tracing* es la geometría de la escena que se quiere representar, así como las propiedades que representan la cámara virtual que será usada.

Dado que no es posible (ni práctico) implementar un esquema recursivo para realizar la tarea de *ray tracing* en *OpenCL*, es necesario adoptar un esquema iterativo. Por lo tanto los rayos se generan en paquetes llamados *ray bundles* y se procesan en paralelo. El primer paquete generado corresponderá a los rayos primarios. A partir de la intersección de los rayos primarios, se crearán sucesivamente otros paquetes que corresponden a rayos secundarios.

El primer paso en el *pipeline* es generar el conjunto de rayos primarios para representar la imagen de la escena. Ese conjunto de rayos será procesado por otro módulo, llamado *tracer*, que calcula la información de intersección de los rayos con la escena. En esta etapa también se determina si el punto de intersección es alcanzado por una fuente de luz a partir de generar y procesar rayos de sombra.

Toda la información resultado de esta última etapa es almacenada en una estructura llamada *hit bundle*, que identifica las propiedades de las intersecciones encontradas para los rayos de un *ray bundle*.

Con esa información, el módulo llamado *ray shader* puede buscar entre las propiedades de la escena, las propiedades materiales de la superficie de intersección. Una vez realizado este paso, se aplica el modelo de iluminación para obtener el color de la superficie en el punto de impacto. Este color se guarda en una estructura llamada *framebuffer*, que es una representación de la imagen final.

La información de intersección y propiedades materiales de la escena también son utilizadas por el módulo llamado *secondary ray generator* para generar rayos de reflexión y refracción.

Los rayos secundarios pasarán nuevamente por el módulo de *tracing* y el resultado nuevamente será evaluado para actualizar la imagen y crear nuevos rayos secundarios. Este proceso se

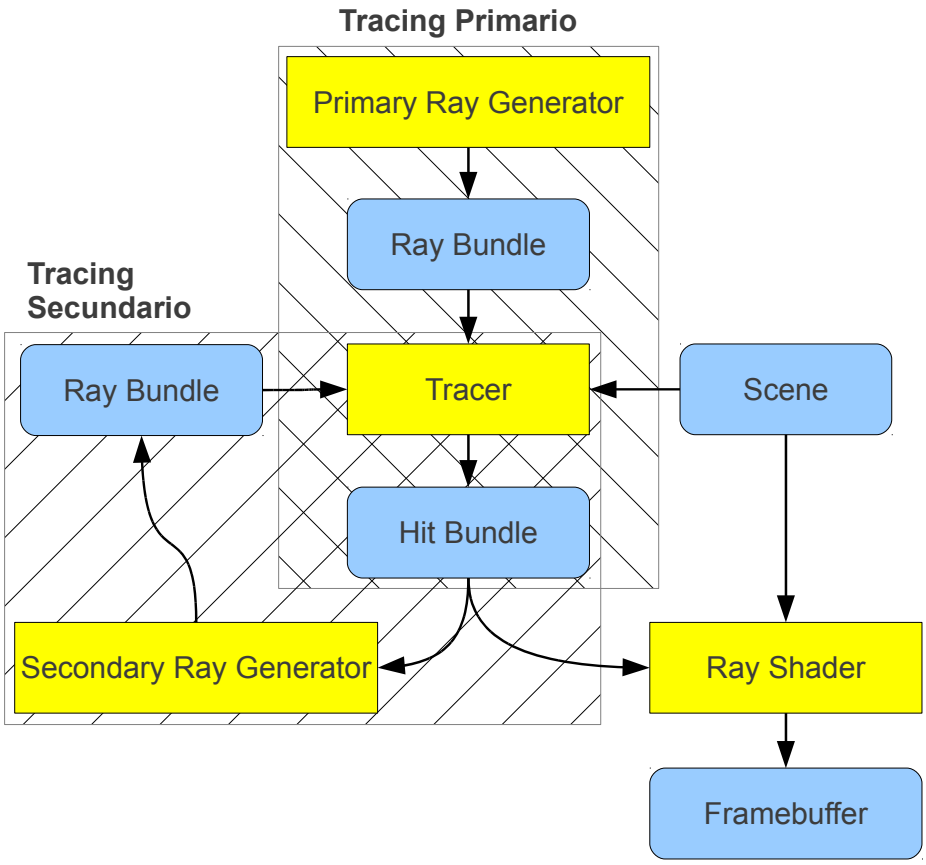


Figura 4: Diagrama de ejecución del *pipeline* de *ray tracing*

repite hasta que no se crean nuevos rayos secundarios o se llega a un número máximo prefijado de iteraciones.

Un esquema del pseudocódigo del algoritmo que controla el *ray tracer* se puede apreciar como algoritmo 1.

Algoritmo 1 Pseudocódigo del cálculo de un frame

```
1: Cargar escena
2: PrimaryRayGenerator → Crear rayos primarios
3: while (Existen rayos para procesar) do
4:     Tracer → Calcular impacto de los rayos
5:     Ray Shader → Actualizar imagen con la información de impacto
6:     if (No se alcanzó la cantidad máxima de rebotes) then
7:         Secondary Ray Generator → Crear rayos secundarios
8:     end if
9: end while
```

4.1. Generación de rayos primarios

La función del módulo llamado *Primary Ray Generator* es simple: a partir de una configuración de la cámara, se genera un *ray bundle* con la información necesaria para que el resto del

pipeline pueda procesarlos. Sin embargo, dado el modelo de cómputo de *OpenCL*, el orden en que se generan estos rayos impacta en el desempeño del *ray tracer*.

Por lo tanto en esta etapa se intenta generar rayos en grupos cuyas direcciones sean lo más similares posibles y sean procesados en un mismo *work group* de forma eficiente, intentando reducir la divergencia en el recorrido de los rayos. Además de la tarea de generar la geometría de los rayos primarios, este módulo también genera los información adicional de un rayo para poder ser procesados por los siguientes módulos en el *pipeline* del *ray tracer*.

4.2. Tracer

El módulo llamado *tracer* es la parte más crítica del *pipeline*. Su función es calcular la intersección entre los rayos de un paquete de rayos con la geometría de la escena. Una vez encontrada la intersección (o la falta de ella) de los rayos del paquete, escribe el resultado de sus cómputos en un *hit bundle*. La información de intersección que se guarda en estos paquetes contiene:

- Un indicador de si hubo o no intersección con la escena.
- Un indicador de si el punto de intersección está iluminado
- Un indicador del lado de la superficie que intersectó el rayo
- Un identificador de la primitiva intersectada
- El punto de intersección
- La normal de la superficie en el punto de intersección
- La coordenadas para texturado del punto de intersección

Para determinar si un rayo intersecta con una primitiva de la escena, se utiliza un BVH. Al recorrerlo se comienza con el nodo raíz. Si no hay intersección con el BBox de la raíz, se sabe que el rayo no intersecta con la escena. Si hay una intersección con el BBox de ese nodo, se desciende al nodo izquierdo y se repite el proceso. Si se llega a un nodo hoja, se intenta intersectar el rayo con todas las primitivas que representa esa hoja. Si el rayo no intersecta un nodo (ya sea por no intersectar su BBox o no intersectar nada en el subárbol debajo suyo), entonces hay dos opciones: si se trata de un nodo izquierdo, se pasa al nodo derecho, y si es un nodo derecho, se sube un nivel y se repite el proceso.

La solución genera *work groups* en los cuales cada *work item* procesa un rayo diferente. El bucle principal se ordena como el bucle *if-if* descripto en Aila y Laine (2009); cada *work item* sigue su propio recorrido dentro del BVH, pero el cómputo de los rayos de un *work group* no terminará hasta que todos sus *work item* hayan terminado su recorrido del árbol. Por lo tanto se tendrá mejor desempeño cuando los rayos asignados a un mismo *work group* hagan el mismo recorrido del BVH. De esto se desprende que el *tracing* de rayos primarios, que son coherentes, será más eficiente que el *tracing* de rayos secundarios.

Una vez que se conoce la primitiva que intersecta el rayo, se puede calcular toda la información que el resto del *pipeline* del *ray tracer* necesita, esto es, la normal en el punto de impacto, coordenadas de textura, color del material, entre otros.

El algoritmo de intersección para rayos de sombra utiliza la misma estructura de aceleración usada para la intersección de rayos primarios y secundarios. Sin embargo, el algoritmo terminará en cuanto se encuentre la primer intersección.

4.3. Generación de rayos secundarios

Una vez calculada la intersección entre un paquete de rayos y la escena, el módulo llamado *Secondary Ray Generator* es responsable de la creación de rayos secundarios a partir de la información de intersección resultante.

La cantidad de rayos secundarios a crear no es conocida a priori, ya que depende de las propiedades de reflectividad de cada objeto. En principio, se estima que cada rayo generará por refracción y reflexión 2 rayos, por lo que se reserva un espacio de memoria suficientemente grande para contener la generación de todos los rayos posibles. La manera más sencilla de decidir el lugar en memoria en que se aloja cada rayo nuevo creado es utilizando un contador sincronizado globalmente que se incrementa atómicamente cada vez que se desea añadir un nuevo rayo. Esta solución, aunque simple, es muy poco eficiente debido a la alta latencia introducida por las funciones atómicas.

Para evitar la necesidad de sincronizar todos los *work item*, la creación de rayos secundarios tendrá 3 fases:

- Por cada rayo procesado de la etapa anterior, un *work item* calcula la cantidad de rayos secundarios que producirá, y guarda esa cantidad en un arreglo auxiliar.
- El arreglo se procesa para guardar en cada posición i la sumatoria del prefijo i de dicho arreglo.
- Con la información del arreglo auxiliar se puede deducir la posición donde guardar los rayos secundarios producidos sin necesidad de sincronizar los *work items*.

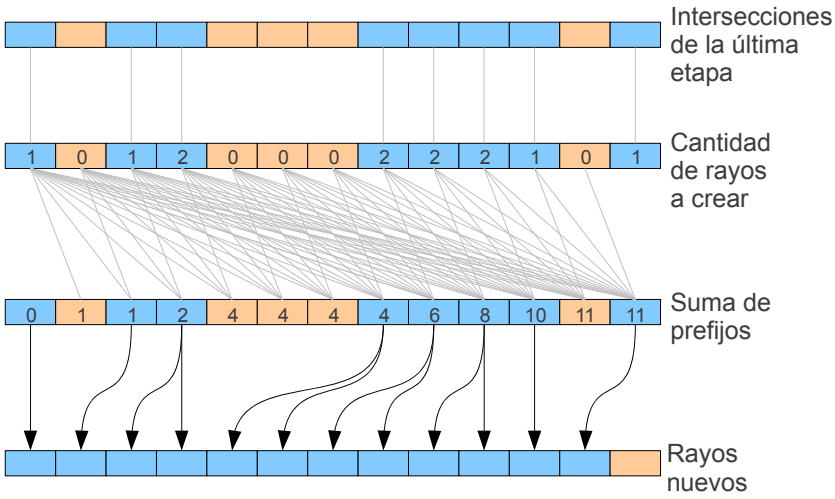


Figura 5: Pasos para crear y guardar rayos secundarios

4.4. Composición de la imagen

Finalmente, la etapa de composición de imagen recibirá como entrada un paquete de rayos, con información de intersección y el material de la primitiva interceptada. El material esta descripto por un color y por una textura. A su vez, por convención, toda la escena se encuentra

limitada por un cubo con texturas (también llamado $cubemap$), para determinar el color de un rayo que no intercepta ninguna primitiva en la escena.

En caso de que haya una intersección, se usará la información del material de la primitiva interceptada, junto con las propiedades de las luces de la escena para determinar el valor del color que se usará. La iluminación se basa en el modelo de *phong* por lo cual hay tres contribuciones principales al color de un píxel: el color proveniente de la luz ambiente de la escena, el color del reflejo difuso y especular de la luz en el material intersectado.

Cada píxel es procesado y almacenado por un *work item* diferente, evitando la sincronización de escritura. Por cada píxel, se computa el color final como la contribución de los rayos primarios, secundarios y de sombra. La contribución de color de un rayo R_p será al final del proceso de *tracing* equivalente a la aplicación de la siguiente fórmula recursiva:

$$c(R_p) = [Phong(T_P, R_p) \times R_p.C \times T_P.A \times T_P.Tex] + (1 - T_P.A) \sum_{R_s} c(R_s)$$

Donde T_p son las propiedades de la superficie en el punto de intersección, $T_p.A$ es la transparencia de dicha superficie, $T_p.Tex$ es el valor de la textura aplicada al punto de intersección, y R_s son los rayos secundarios que se generan a partir de la intersección. La función $Phong$ expresa la aplicación del modelo de iluminación de *phong* al punto de intersección. $R_p.C$ es la contribución al color final que tendrá el color calculado para un rayo; para rayos primarios será 1, y será calculado para los rayos secundarios al momento de su creación.

5. RESULTADOS

Las escenas utilizadas en esta propuesta son descriptas por triángulos, los cuales son luego clasificados por el BVH. Cada triángulo tiene un índice a un material, el cual describe el color, el tipo de material (reflectivo u opaco) y una textura, todos alojados en una estructura global. Todos estos datos son copiados a memoria de GPU para su ejecución.

El *pipeline* propuesto funciona en *OpenCL*, a excepción de la creación del BVH, que todavía se realiza en CPU . Las escenas utilizadas son estáticas aunque pueden ser observadas desde distintos puntos de vista. Para mostrar el resultado del renderizado de una escena, se utiliza OpenGL la cual se integra fácilmente con *OpenCL*.

Para mostrar la aplicación de este *pipeline*, primero se evaluan los requerimientos de memoria necesarios para ejecutar este tipo de problemas en una GPU y luego el tiempo de generar una imagen para distintas escenas y con distintos puntos de vista.

5.1. Pruebas de requerimientos de memoria

Para la primera serie de pruebas, se define un tamaño de buffer de 512x512 píxeles y una cantidad de rebotes máxima de 5, para lo cual se generan 262144 rayos primarios, y un máximo de 4 millones de rayos secundarios, aunque en la práctica es raro encontrar escenas donde se produzcan más de 1 millón. Por cada rayo se alojan aproximadamente 144 bytes para almacenar su información y la información de intersección necesaria para calcular su contribución de color. En la tabla 1 se puede apreciar los requerimientos de memoria para el renderizado de una escena. Para procesar imágenes en resolución Full HD (1920x1080) se requieren menos de 150 MB de memoria de video.

Procesar una imagen de grandes dimensiones puede requerir una gran cantidad de memoria prealocada si queremos mantener todos los rayos de un mismo nivel en memoria. Sin embargo,

dado el hecho de que durante la ejecución del *pipeline* la información obtenida de un rayo primario y sus rayos derivados no afecta a la información obtenida del resto, es posible procesar rayos primarios en grupos o *tiles* suficientemente grandes como para asegurar que todas las unidades de proceso sean utilizadas (al menos durante el *tracing* primario). De esta manera se puede mantener la eficiencia del sistema a la vez que se reduce la cantidad de memoria que el programa necesita.

Resolución	512x512	1024x1024
Tamaño de tile	81920	81920
Memoria para rayos	39.32 MB	39.32 MB
Memoria para intersección	31.45 MB	31.45 MB
Memoria para framebuffer	4.2 MB	16.77 MB

Tabla 1: Estadísticas de memoria

Escena	Triángulos	Vértices	Memoria para geometría
Mano	34270	10039	1.2 MB
Humano	78029	49964	5.9 MB
Bote	10979	6884	0.68 MB
Dragón	109248	54761	5.7 MB
Buddha	100020	50010	5.2 MB

Tabla 2: Cantidad de primitivas en las escenas de prueba

Por otra parte, la tabla 2 muestra los requerimientos en memoria de las escenas de prueba. El tamaño en memoria es proporcional a la cantidad de triángulos de la escena, e independiente de la resolución del *ray tracer*. No se incluye el espacio de alojar las texturas.

En resumen, una escena con hasta 10^6 triángulos y en máxima resolución, requiere no más de 150 Mbytes de memoria, por lo cual cualquier GPU comercial puede alojar esta estructura.

5.2. Pruebas de rendimiento

Para el segundo conjunto de pruebas se espera analizar la posibilidad de utilizar el *ray tracer* en tiempo real en diferentes situaciones, también generando imágenes de 512x512. Se usaron escenas típicas de benchmarks de este tipo de aplicaciones (mostradas en el Apéndice A), con diferentes características de los materiales y en el cual se fue variando el ángulo de vista. Esta variación del ángulo de vista impacta en los tiempos finales del renderizado, especialmente cuando se está observando un objeto con propiedades refractivas, como es el caso del agua. En este caso, la cantidad de rayos secundarios aumenta, y esto se ve reflejado en tiempos de proceso más largos.

El *pipeline* fue probado en una computadora con una placa gráfica NVIDIA GeForce GTX 560 junto a un procesador Intel i5 2500K.

En las escenas 4 y 5 se presentan los casos más complejos. En el caso de la escena 4, una malla del orden de 10^5 primitivas reflectivas se ubica sobre una superficie refractiva, generando gran cantidad de rayos secundarios. En el caso de la escena 5, una malla compleja se encuentra dentro de un cubo reflectivo. A pesar de que en ambos casos se generan gran cantidad de rayos secundarios, es de notar que los rayos de la escena 5 son mucho más coherentes que los de la

Escena	Resolución	Rayos secundarios	Creación de rayos	*Tracing* primario	*Tracing* secundario	Sombras	*Shading*	Total
1(a)	512x512	0	2.1	8.8	0	2.6	1.2	14.7
1(b)	512x512	0	2.1	4.6	0	2.6	1.1	10.4
1(c)	512x512	0	2.1	4.9	0	1.3	1.2	9.5
1(a)	1024x1024	0	4.4	16	0	4.9	3.2	28.5
1(b)	1024x1024	0	4.4	9.3	0	5.2	3.1	22
1(c)	1024x1024	0	4.4	9.3	0	2.9	3	19.6
2(a)	512x512	0	2	10.8	0	5.2	1.1	19.1
2(b)	512x512	0	2.1	13.9	0	7.6	1.2	24.8
2(c)	512x512	0	2	13	0	5.5	1.2	21.7
2(a)	1024x1024	0	4.4	11.3	0	8.3	2.9	26.9
2(b)	1024x1024	0	4.4	14.5	0	12.2	3.2	34.3
2(c)	1024x1024	0	4.4	18.7	0	8.6	3.1	34.8
3(a)	512x512	191031	4.5	5	4.4	5.7	3.3	22.9
3(a)	1024x1024	585409	13.6	10.4	13.6	7.3	9.3	54.2
4(a)	512x512	366475	10.8	7.6	35.4	21.8	6.3	81.9
4(b)	512x512	525440	13.6	9.1	61.9	39.3	7.7	131.6
4(a)	1024x1024	1034397	21.1	17.6	94.8	70	13.9	217.4
4(b)	1024x1024	1425040	24.5	22.2	178.9	136.5	17.2	379.3
5(a)	512x512	708760	14.3	8.6	11.5	15.1	7.6	57.1
5(a)	1024x1024	2280820	34.3	18.4	31.4	32.9	19.9	136.9

Tabla 3: Resultados obtenidos en las escenas de prueba (tiempo en milisegundos)

escena 4, resultando en tiempos de proceso menores para la etapa de *tracing* de rayos secundarios. Las mallas 3D para las escenas de prueba fueron obtenidas del repositorio de Stanford[2] y del repositorio de animación 3D de la universidad de Utah[3].

Los tiempos de ejecución para las diferentes etapas de la solución son de la proporción esperada. En la escena 1 y la escena 2, el tiempo de la etapa de *tracing* es el mayor.

En escenas donde existen materiales reflectivos, se producen una gran cantidad de rayos secundarios que a la vez son poco coherentes. El tiempo de cómputo del *tracing* para esos rayos es entre 3 y 5 veces mayor que la etapa de *tracing* primario.

La etapa de *tracing* de sombras toma entre 70 % y 50 % menos tiempo que las etapas de *tracing* normal. En primer lugar esto se debe a que el *tracing* de sombra puede terminar prematuramente al encontrar la primera intersección, y también que en la etapa de *tracing* normal se calculan características de la intersección, como la normal en el punto de intersección y las coordenadas para texturas. Adicionalmente, a través de una herramienta de *profiling* fue posible descubrir que en la etapa de *tracing* se declara mayor cantidad de variables automáticas para realizar los cómputos extra. Dichas variables son almacenadas en registros, y cuando no existen registros disponibles, son guardados en memoria global de video, la cual es mucho más lenta.

El tiempo que requiere la etapa de creación de rayos es proporcional a la cantidad de triángulos con materiales refractivos y reflectivos. En las escenas 5 y 4, se puede apreciar en la tabla 3 que esta etapa es la más costosa. Esto se puede mitigar disminuyendo la cantidad de rebotes máximos permisible, a costo de disminuir la calidad de la imagen producida.

Finalmente, la eficiencia de la etapa de composición de imagen, o *shading*, está directamente ligada a la cantidad de rebotes que se calculen en la escena. Sin embargo, relativo a una iteración del *pipeline* de *ray tracing*, esta etapa representa aproximadamente entre el 3 % y el 8 % del

[2]http://graphics.stanford.edu/data/3Dscanrep/
[3]http://www.sci.utah.edu/~wald/animrep/

tiempo total de cómputo.

La etapa de *tracing* primario llega a procesar en las escenas de prueba entre 18 y 112 millones de rayos primarios por segundo, dependiendo de la complejidad de la escena. Los resultados son similares a los presentados en Parker et al. (2010) para el *ray tracer* en GPU OptiX de la compañía NVIDIA, el cual procesa entre 45 y 192 millones de rayos por segundo para escenas de hasta 280000 triángulos.

6. CONCLUSIONES Y TRABAJO FUTURO

Se ha presentado una implementación de un *ray tracer* que utiliza *OpenCL* para tomar ventaja del poder de cómputo disponible en las placas gráficas .

En cada cada etapa del *ray tracer* se utilizó distintos niveles de granularidad, buscando maximizar el uso de la GPU. En este sentido, se propuso un método para la organización de rayos secundarios y un esquema para la composición final de la imagen. A su vez, el recorrido del BVH fué adaptado para ser ejecutado de manera iterativa.

La solución fue probada en un GPU comercial de última generación, obteniéndose resultados satisfactorios en cuestiones de rendimiento, que permite navegar escenas en forma interactiva.

Si bien la performance obtenida es un poco menor a otros trabajos similares, nuestra propuesta al utilizar *OpenCL*permite ser ejecutada en otras arquitecturas tales como ATI e Intel.

Como trabajo futuro, se puede citar la creación del BVH en *OpenCL*y otras heurísticas, de manera de poder trabajar con escenas dinámicas y animaciones en tiempo real.

REFERENCIAS

Aila T. y Laine S. Understanding the efficiency of ray traversal on gpus. En *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09. ACM, 2009.

Appel A. Some techniques for shading machine renderings of solids. En *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, AFIPS '68 (Spring). 1968.

Bak P. *Real time ray tracing*. Tesis de Maestría, IMM, DTU, 2009.

Carr N.A., Hoberock J., Crane K., y Hart J.C. Fast gpu ray tracing of dynamic meshes using geometry images. En *Proceedings of Graphics Interface 2006*, GI '06. 2006.

Christen M. *Ray tracing on GPU*. Tesis de Doctorado, Chalmers, 2005.

D'Amato J., Garcia Bauza C., y Vénere M. Un framework en placas gráficas (gpu) para aplicaciones basadas en ray-tracing. *Mecánica Computacional*, 2011.

Fowler C. y Collins S. Implementing the rt^2 real-time ray-tracing system. En *Proceedings of Eurographics Ireland*. 2007.

Garanzha K. y Loop C. Fast ray sorting and breadth-first packet traversal for gpu ray tracing. *Computer Graphics Forum*, 29, 2010.

Gunther J., Popov S., Seidel H.P., y Slusallek P. Realtime ray tracing on gpu with bvh-based packet traversal. *Symposium on Interactive Ray Tracing*, 2007.

Hapala M., Davidovic T., Wald I., Havran V., y Slusallek P. Efficient stack-less bvh traversal for ray tracing. En *27th Spring Conference on Computer Graphics (SCCG 2011)*. 2011.

Khronos Group. *The OpenCL Specification, version 1.0.29*, 2008.

Ludvigsen H. y Elster A. Real-time ray tracing using nvidia optix. En *2010 Eurographics*. 2010.

Parker S.G., Bigler J., Dietrich A., Friedrich H., Hoberock J., Luebke D., McAllister D., McGuire M., Morley K., Robison A., y Stich M. Optix: a general purpose ray tracing engine. *ACM Trans. Graph.*, 29, 2010.

Popov S., Gunther J., Seidel H.P., y Slusallek P. Experiences with streaming construction of sah kd-trees. *Symposium on Interactive Ray Tracing*, 0, 2006.

Shevtsov M., Soupikov A., y Kapustin A. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. *Computer Graphics Forum*, 26, 2007.

Wald I., Gribble C., Boulos S., y Kensler A. Simd ray stream tracing – simd ray traversal with generalized ray packets and on-the-fly re-ordering. Informe Técnico, SCI Institute, 2007.

Wächter C. y Keller A. Instant ray tracing: The bounding interval hierarchy. En *IN RENDERING TECHNIQUES 2006 – PROCEEDINGS OF THE 17TH EUROGRAPHICS SYMPOSIUM ON RENDERING*. 2006.

Zlatuška M. y Havran V. Ray tracing on a gpu with cuda – comparative study of three algorithms. Informe Técnico, Czech Technical University in Prague Faculty of Electrical Engineering, 2009.

A. APÉNDICE: ESCENAS DE PRUEBA

(a) Escena 1(a)

(b) Escena 2(b)

(c) Escena 2(c)

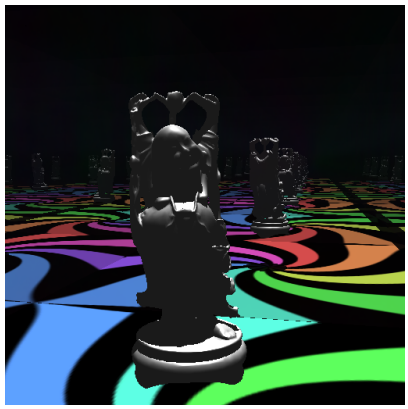
(d) Escena 3(a)

(e) Escena 3(c)

(f) Escena 4(a)

(g) Escena 4(b)

(h) Escena 4(c)



(i) Escena 5(a)