Universidad Nacional de Rosario

Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Departamento de Ciencias de la Computación

Tesina de grado

Raytracing interactivo en OpenCL

Director: Dr. Cristian García Bauza
Co-Director: Dr. Juan Pablo D'Amato

Leonardo Scandolo

15 de septiembre de 2013

Resumen

Los métodos de *ray tracing* permiten generar representaciones gráficas de escenas tridimensionales con gran calidad, ya que modelan naturalmente sombras, reflexiones y refracciones además de la emisión difusa de luz. 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).

En este trabajo se presenta una implementación de un *ray tracer* que fue diseñado para hacer uso del poder de cómputo de placas gráficas. El mismo utiliza para la clasificación espacial de sus escenas un árbol de Jerarquía de Volúmenes (BVH). El *pipeline* de procesamiento fue implementado en *OpenCL* y permite generar imágenes de alta calidad con tasas de refresco interactivas. Asimismo se exponen resultados en resoluciones de 512^2 y 1024^2 píxeles a partir de escenas del orden de 10^5 triángulos.

En el capítulo 1 se introducen algunos conceptos básicos de *ray tracing* y programación en GPGPU, asi como una pequeña reseña de los antecedentes en el area.

En el capítulo 2 se presentan los conceptos de *ray tracing* necesarios para describir la solución propuesta.

En el capítulo 3 se presenta la implementación realizada, los problemas encontrados y las soluciones que se propusieron a las mismas.

En el capítulo 4 se presentan los resultados obtenidos con la implementación, así como un análisis de los mismos.

En el capítulo 5 se presenta dos extensiones al trabajo: la creación de la estructura de aceleración completamente en GPGPU, y la integración con una biblioteca de simulación de fluido superficial.

Finalmente, el capítulo 6 presenta las conclusiones que se obtuvieron a partir del trabajo y los resultados, así como las lineas de trabajo futuro posibles.

Índice general

Índice de figuras

Abreviaturas

BVH	Bounding Volume Hierarchy (Arbol de jerarquía de volumenes)
LBVH	Linear Bounding Volume Hierarchy (Arbol de jerarquía de volumenes lineal)
CPU	Central processing unit (Unidad de procesamiento central)
GPGPU	General purpose computing on graphic processing units (Computación de propósito general en unidades de procesamiento de gráficos)
GPU	Graphics processing unit (Unidad de procesamiento de gráficos)
OpenCL	Open Computing Language (Lenguaje de computación abierto)
OpenGL	Open Graphics Library (Biblioteca de gráficos abierto)
SAH	Surface Area Heuristic (Heurística de área superficial)
SIMD	Single instruction multiple data (Instrucción singular ejecutada sobre múltiples datos)

Capítulo 1

Introducción

1.1. Motivación

La técnica de *ray tracing* sirve para generar imágenes a partir de una escena en 3 dimensiones y fue propuesta en la década del 60' por Arthur Appel [2]. En su forma más simple el algoritmo modela una cámara en una escena virtual; por cada pixel 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 pixel correspondiente resulta de la aplicación de un modelo de iluminación al punto de impacto del rayo con la escena.

A pesar de ser conocido hace casi 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, sólo se utiliza en sistemas de renderizado *off-line* debido a que es un algoritmo computacionalmente costoso, como se puede apreciar en [19].

Actualmente la estrategia más popular para obtener imágenes en tiempo real sigue siendo el renderizado basado en *"triangle rasterization"*, soportado de forma eficiente por placas gráficas. Esta técnica permite el renderizado eficiente de una escena desde un punto de vista, pero la reproducción de efectos ópticos como reflexiones, transparencias o sombras es muy costosa y en muchos casos sólo se logra con trucos o aproximaciones. Esto hace que los sistemas de renderizado basado en esta tecnología sean extremadamente complejos. Por otro lado, la técnica de *ray tracing* trata de aproximar de manera más natural la interacción de la luz con una escena, y su proyección en una cámara. Por lo tanto esta técnica logra obtener mejores resultados, pero a un costo computacional mucho más elevado. Esto lleva a que se utilice sólo en aplicaciones que no necesiten resultados instantáneos, como en la generación de películas o efectos especiales, pero no en aplicaciones como videojuegos o recorridos virtuales interactivos. De ahí la importancia de poder generar una solución que permita generar imágenes utilizando la técnica de *ray tracing* en tiempo real. En esta tesina se tratará de utilizar el poder computacional de placas gráficas (GPUs) para acelerar el proceso de todas las etapas de un *ray tracer*.

1.2. *Ray tracing*

Un concepto muy importante para los algoritmos de *ray tracing* es el de *rayos*. Los rayos son segmentos, o semirrectas dependiendo del modelo usado, que modelan la trayectoria de la luz en una escena. A través de los mismos se modela una cámara por la cual se ve la escena tridimensional que se quiere representar como una imagen bidimensional. También permiten representar efectos ópticos de la interacción de las fuentes de luz con la escena.

Un rayo puede ser representado por dos vectores: un vector *origen* y un vector *dirección*.

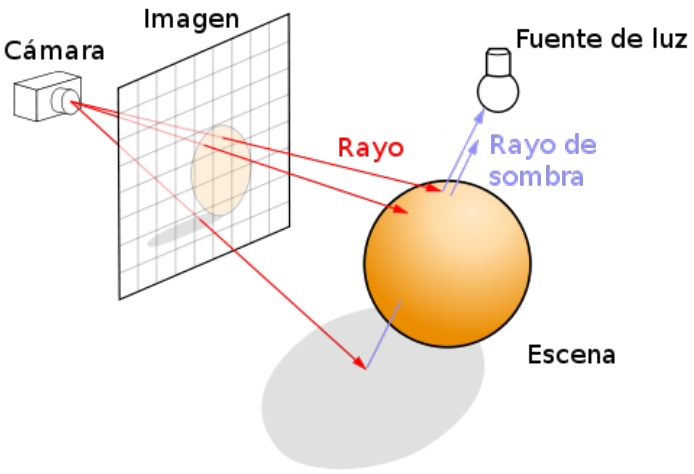


Figura 1.1: *Ray tracing*

Además es común ver otros datos asociados a un rayo, tales como distancia mínima o máxima de intersección, o información sobre el medio en el cual se genera el rayo.

El modelo de cámara más simple es el de una cámara estenopeica[1] en la cual cada rayo, representa un pixel de la imagen. En este modelo todos los rayos tienen el mismo punto de origen, que representa el punto de apertura de la cámara. La dirección de los rayos está en función a la orientación de la cámara y la amplitud del campo visual que se quiera representar. Los rayos que se generan para representar la cámara son llamados *rayos primarios*. La figura 1.1 muestra una representación gráfica de este concepto.

Los algoritmos de *ray tracing* utilizan una estructura de datos llamada *estructura de aceleración* para encontrar más eficientemente la intersección de los rayos con la escena. El tipo de estructura de aceleración utilizada en este trabajo es un árbol de jerarquía de volúmenes, abreviado BVH por sus siglas en inglés (*bounding volume hierarchy*). Esta estructura será descrita en detalle en el capítulo 2.

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 describirán brevemente en el capítulo 2.

1.3. Programación GPGPU en *OpenCL*

El modelo de cómputo general en placas gráficas (GPGPU, por sus siglas en inglés: *General Purpose computing on Graphics Processing Units*) es de reciente aparición y la biblioteca CUDA de la compañía NVIDIA es una de las primeras incursiones en este campo. A partir del éxito de CUDA, se define el estándar *OpenCL* ([12]) para estandarizar este tipo de cómputos sobre diferentes plataformas.

El estándar de *OpenCL* provee una abstracción para diferentes dispositivos (en general GPUs) que les permite exponer una interfaz común que hace explícitas sus capacidades para hacer cómputos con alto nivel de paralelismo. Esta abstracción consiste de una arquitectura abstracta con capacidades para cómputos masivamente paralelos, junto con un lenguaje y bibliotecas que permiten programar cualquier dispositivo que soporta el estándar de la misma forma. De esta

[1] Comúnmente llamada cámara oscura, donde la luz de una escena llega a un sensor en una caja a través de una apertura muy pequeña en la misma.

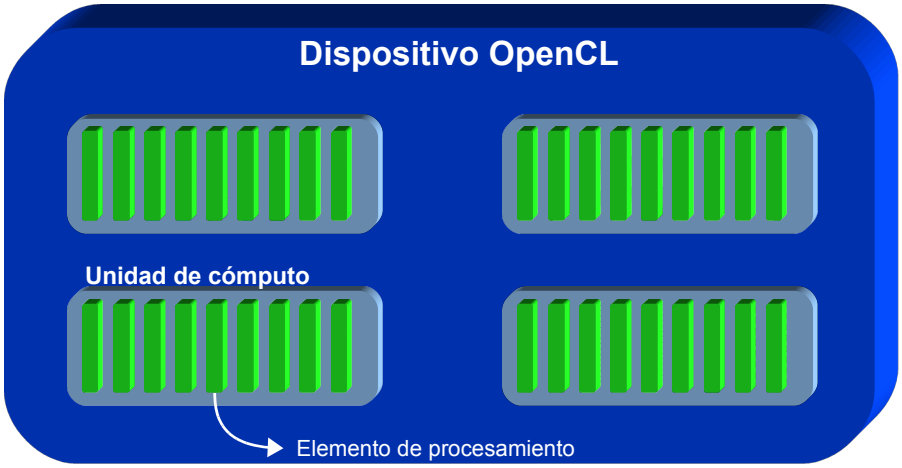


Figura 1.2: Diagrama de la arquitectura básica expuesta por *OpenCL*

manera es posible ejecutar el mismo cómputo sobre una placa gráfica de cientos de procesadores como en un CPU multi-core de 2 o 4 procesadores.

1.3.1. Arquitectura de un dispositivo *OpenCL*

En la arquitectura definida por *OpenCL* cada procesador o placa de gráficos que implementa el estándar es llamada *dispositivo* o *device OpenCL*. Cada *dispositivo* expone una cantidad de *unidades de cómputo*, o *compute units*. Estas *unidades de cómputo* son grupos de uno o más procesadores. Los procesadores que forman parte de una *unidad de cómputo* son llamados *elementos de procesamiento* o *processing elements*. Todos los *elementos de procesamiento* dentro de una misma *unidad de cómputo* comparten recursos, como memoria y cache. Al momento de ejecutar instrucciones se comportan como una linea larga SIMD[2]. Esto significa que todos los *elementos de procesamiento* de una misma *unidad de cómputo* ejecutan las mismas instrucciones, aunque posiblemente leyendo y escribiendo a lugares diferentes de la memoria. En el caso de expresiones condicionales (`IF-THEN-ELSE`) simplemente se desactivan los *elementos de procesamiento* correspondientes durante la ejecución de la rama condicional que no deben ejecutar.

La figura 1.2 muestra un esquema de la arquitectura descripta en el párrafo anterior.

1.3.2. Jerarquía de memoria en *OpenCL*

OpenCL también asume una jerarquía de memoria que está ligada a la forma en que se organizan los procesadores de un dispositivo. Cada *elemento de procesamiento* tiene acceso a su propia memoria privada, que consiste generalmente de registros. A su vez, los *elementos de procesamiento* de una misma *unidad de cómputo* tienen acceso a una memoria compartida que todos los elementos pueden leer y escribir, pero que no es accesible para los elementos de otras *unidades de cómputo*. Finalmente existe una memoria global del *dispositivo* que es accesible a todos los *elementos de procesamiento*.

Como es de esperar cada jerarquía de memoria tiene tamaños y tiempos de acceso diferentes, por lo cual, en la mayoría de los casos, el tiempo de acceso a registros de un *elemento de procesamiento* es menor al de la memoria compartida de una *unidad de cómputo*, que es a la vez

[2]Single Instruction Multiple Data

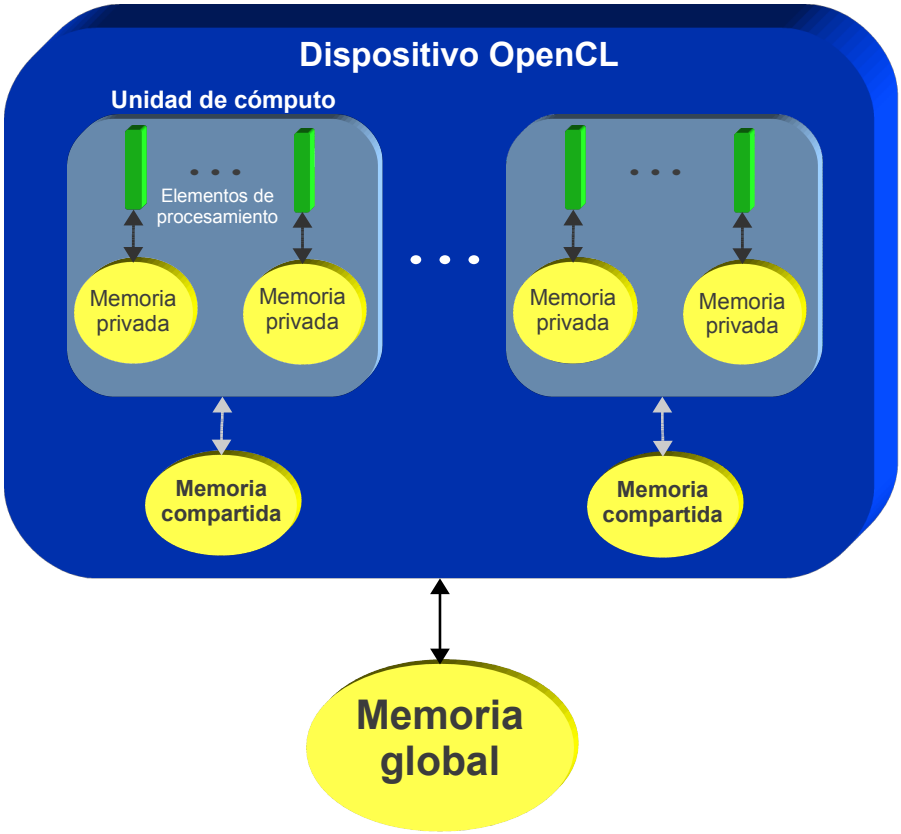


Figura 1.3: Diagrama de la jerarquía de memoria expuesta por *OpenCL*

menor al de la memoria global de un *dispositivo*. El caso contrario se da con el tamaño de cada jerarquía de memoria.

Cada *dispositivo* posee la capacidad de transferir datos desde y hacia la memoria principal de la computadora donde se encuentra. Los tiempos de transferencia en este caso son por lo general altos.

La figura 1.3 ejemplifica la jerarquía de memoria que expone *OpenCL*.

1.3.3. Kernels de *OpenCL*

Los algoritmos ejecutados en *OpenCL* son llamados *kernels*. Éstos son escritos en un lenguaje de programación propio de *OpenCL*, muy parecido al lenguaje C. Dichos algoritmos serán ejecutados por los *elementos de procesamiento* de un *dispositivo*. Al momento de ejecutar un *kernel* en un *dispositivo* es posible definir cuantas instancias de ese cómputo se ejecutarán. De esta manera un mismo cómputo puede ser ejecutado una gran cantidad de veces con una sola llamada a la biblioteca *OpenCL* mediante la capacidad de cómputos paralelos de los dispositivos que implementan el estándar *OpenCL*.

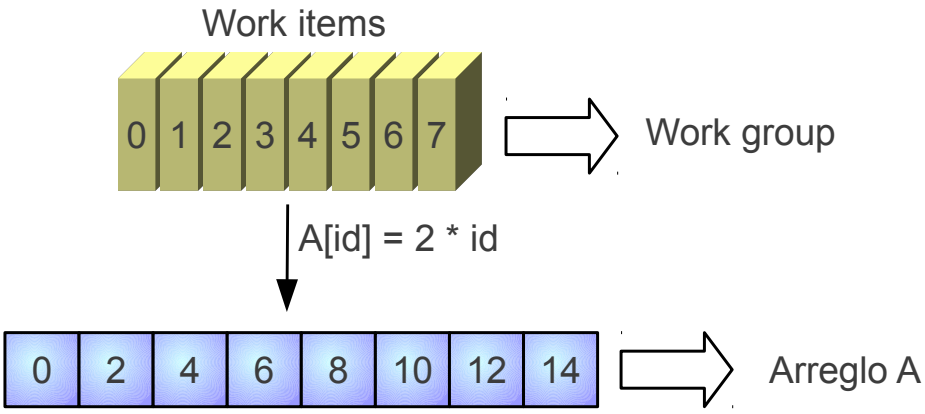
Cada instancia a ejecutar de un *kernel* es llamada un *work item*. Cuando se define la cantidad de work items a ejecutar, también es necesario definir cómo serán agrupadas esas instancias. Los work item se agrupan porque cada instancia será ejecutada por un *elemento de procesamiento*, y cada grupo será ejecutado por una *unidad de cómputo*. Cada grupo de work items es llamado un *work group*. Los work items de un mismo work group tienen acceso a memoria compartida dado a que se ejecutan en la misma *unidad de cómputo*. Los work items de diferentes work groups sólo

```
__kernel void fill_array(global int* A) {

    size_t id = get_global_id(0);

    A[id] = 2 * id;
}
```

(a) Código de un *kernel* de *OpenCL* que modifica los valores de un arreglo global.



(b) Esquema del código ejecutado por el *kernel*.

Figura 1.4: Ejemplo de un cómputo para modificar un arreglo usando un *kernel* de *OpenCL*

pueden comunicarse a través de la memoria global.

Para poder ejecutar código sobre diferentes datos, cada work item tiene acceso a un identificador global que lo diferencia de todos los otros work items y a un identificador local que lo identifica dentro de su work group.

La figura 1.4 ejemplifica la manera en que un *kernel* puede modificar datos distintos mientras ejecuta las mismas instrucciones. El *kernel* contiene sólo una instrucción que establece el valor de una posición de un arreglo. Tanto la posición como el valor son función del identificador de cada work item que se ejecute.

La cantidad de work items en un mismo work group está limitada por la cantidad de *elementos de procesamientos* por *unidad de cómputo* en el dispositivo *OpenCL*. Sin embargo, la cantidad total de work items no está acotada, dado que en caso de haber más work groups que *unidades de cómputo*, simplemente se ejecutarán primero tantos como *unidades* haya disponible, y luego que se completen sus cálculos, se continuará con los siguientes, hasta culminar la ejecución de todos los work groups. Se desprende de esta lógica de ejecución que existe una expectativa de concurrencia para la ejecución de los work items de un mismo work group, pero no para work items de diferentes work groups.

Una característica importante de la definición de instancias de un *kernel* a ejecutar es que pueden definirse grupos multidimensionales, es decir que el identificador de cada work item y cada work group puede tener generalmente hasta 3 dimensiones. Esto facilita la división de trabajo en una gran cantidad de algoritmos paralelos.

1.3.4. *OpenCL* y bibliotecas gráficas

Además de proveer una interfaz para computación general, *OpenCL* incluye extensiones para interactuar con otras bibliotecas usualmente encontradas en GPUs, cómo *OpenGL* o *DirectX*.

De esta manera la memoria del dispositivo que implementa *OpenCL* puede simplemente ser utilizada para otros propósitos por otra biblioteca sin la necesidad de copiarla.

1.3.5. Limitaciones de *OpenCL*

Las propiedades del modelo de cómputo de *OpenCL* afectan el diseño de la solución que se propone. Por lo tanto a continuación se discutirán algunas limitaciones y factores que afectan la eficiencia de los programas que utilizan *OpenCL*.

Un factor importante que afecta la eficiencia de *OpenCL*es que la ejecución de un *kernel* es ineficiente cuando diferentes work items de un mismo work group ejecuten ramas distintas de una expresión condicional. Esto es debido al modelo de cómputo expuesto basado en SIMD. Cuando un *kernel* no sufre este problema, se dice que los datos que está procesando son *coherentes*. Por lo tanto es siempre mejor agrupar los datos a procesar de manera que sean lo más coherentes posibles. A pesar de que no es una limitación que no permita hacer un cálculo, el modelo de *OpenCL* fue creado con aplicaciones de alto desempeño como objetivo, por lo cual si no es posible obtener un buen rendimiento es preferible hacer cómputos en unidades de procesos tradicionales, como CPUs.

El lenguaje de definición de *kernels* en *OpenCL* es un lenguaje muy parecido al lenguaje C. Sin embargo presenta dos limitaciones muy importantes: la falta de recursión y la falta de memoria dinámica.

La falta de recursión implica que cualquier algoritmo usado en *OpenCL* debe ser completamente iterativo. Esta es sólo una limitación sobre la facilidad de escribir código, dado que cualquier cómputo recursivo (que termine) puede escribirse de forma iterativa. Sin embargo el algoritmo de *ray tracing* se presta más a ser escrito de forma recursiva, por lo cual se hace notar esta limitación para entender la razón de las decisiones que llevan a la solución que se presenta.

La segunda limitación es más problemática, dado que implica que debemos establecer la cantidad de memoria que un *kernel* va a utilizar antes de comenzar su ejecución. Para un algoritmo de *ray tracing* el rebote de la luz en algunos materiales puede llevar a la creación de rayos secundarios, y por lo tanto se deberá prestar mucha atención a la manera en que se procesan estos datos cuya existencia no se puede determinar a priori.

Otra limitación importante del modelo es la separación entre memoria del dispositivo que implementa *OpenCL* y la memoria convencional del sistema. La memoria del dispositivo puede ser accedida rápidamente durante la ejecución de un *kernel* en *OpenCL*. Sin embargo escribir o leer la memoria del dispositivo desde fuera del mismo puede ser lento, y por lo tanto se debe minimizar en lo posible estas operaciones.

Un último inconveniente resultante del uso de arquitecturas masivamente paralelas es que dado que están diseñadas para hacer cálculos utilizando muchos procesadores, la subutilización de los mismos decrementa la velocidad en la cual se ejecuta un algoritmo. En particular, dado que cada work group es asignado a una *unidad de cómputo* diferente, es siempre más eficiente crear work groups que utilicen todos los recursos de la *unidad de cómputo* donde se ejecutarán.

Resumiendo, las limitaciones más importantes del modelo *OpenCL* son:

- Necesidad de tener coherencia de datos.
- Falta de recursión.
- Falta de memoria dinámica.
- Lentitud de traspaso de datos entre sistema y dispositivo *OpenCL*.
- Necesidad de utilizar eficientemente los *elementos de procesamiento* disponibles.

1.4. Antecedentes

Existen muchos antecedentes de trabajos tratando el tema de *ray tracing* en arquitecturas paralelas. A continuación se dará una pequeña reseña de las publicaciones más importantes en el área.

Muchos basan su trabajo en tratar de utilizar las características SIMD de los CPU modernos para acelerar el proceso de creación de estructuras de aceleración e intersección de rayos. En [20], se presenta un método eficiente para calcular la SAH[3] de una partición de una BBox[4]. Este método consiste en agrupar primitivas y calcular SAH incrementalmente cuando la cantidad de primitivas es muy grande, y usar *radix sort* y calcular SAH de la manera estándar cuando la cantidad de primitivas es menor. En [21], se presenta una técnica de creación de kd-trees paralela junto con un algoritmo de *ray tracing* basado en creación de pirámides truncadas (*frustums*) para paquetes de rayos. Se muestran resultados del algoritmo implementado para CPUs usando instrucciones SIMD.El proyecto RT^2, presentado en [7] es un *ray tracer* construido para poder obtener eficiencia suficiente para producir imágenes en tiempo real. Implementa varias técnicas de paralelización en diferentes partes de la implementación para hacer uso de las instrucciones SIMD de los CPU modernos. En [24] se presenta un algoritmo de *ray tracing* que utiliza una estructura de aceleración que es mezcla entre un BVH y un kd-tree. Cada nodo interior tiene dos hijos, y cada hijo define un plano que corta la BBox del padre. Los planos de ambos hijos son paralelos y definen dos subespacios que pueden solaparse. De esta manera obtienen una eficiencia parecida a la obtenida usando un kd-tree, junto con un ordenamiento espacial de los nodos para el proceso de intersección de rayos. El *ray tracer* presentado en [9] utiliza BVHs como estructura de aceleración. Dicho BVH se construye en CPU usando una técnica de agrupamiento para agilizar el cálculo de SAH. La estructura resultante es recorrida en un GPU, utilizando un work item por rayo para paralelizar el trabajo. Los resultados muestran que esta estrategia obtiene tiempos de renderización bajos. El trabajo presentado en [22] discute la implementación de un algoritmo basado en arquitecturas SIMD que permite procesar paquetes de rayos de manera eficiente compactando implícitamente paquetes de rayos a medida que se procesa el árbol de rayos primarios y secundarios. El trabajo fue hecho con CPUs, utilizando instrucciones SIMD de largo 4, pero propone utilizarlo para líneas SIMD de mayor tamaño, como las que se encuentran en GPUs.

Los primeros trabajos utilizando GPUs para implementar algoritmos de *ray tracing* utilizaban bibliotecas gráficas de maneras no convencionales para poder hacer uso de las arquitecturas paralelas existentes. La tesis de doctorado de Martin Christen ([6]) implementa muchos de los conceptos de interfaces de GPGPU usando sólo OpenGL y DirectX, dado que al momento de escritura del trabajo, no existían CUDA, OpenCL ni otras bibliotecas GPGPU. El *ray tracer* que describe el trabajo está basado en grillas regulares y sólo usa efectos de reflexión. Obtiene buenos resultados para las placas gráficas usadas. En [11], los autores presentan varias modificaciones al algoritmo básico de *ray tracing* basado en paquetes y utilizando kd-trees como estructura de aceleración, y prueban sus resultados en un GPU comercial. El artículo [5] explica la construcción de un *ray tracer* que utiliza imágenes de geometría como primitivas y un BVH como estructura de aceleración. El programa corre en GPU usando bibliotecas gráficas para simular el uso de bibliotecas GPGPU.

Existen antecedentes de trabajos describiendo la aceleración de diferentes partes de un algoritmo de *ray tracing* utilizando GPGPU utilizando CUDA. El trabajo [13] describe dos formas de crear un BVH en una arquitectura GPU. Uno de los algoritmos propuestos utiliza códigos

[3] SAH (Surface Area Heuristic) es una heurística usada en la creación de estructuras de aceleración. Será explicada en el capítulo 2.

[4] *Bounding Box*, un espacio prismático en una escena que engloba algunos elementos de la misma.

de Morton para ordenar primitivas de una manera eficiente cuando hay muchas para calcular. También muestra un segundo algoritmo que divide el trabajo de calcular el próximo nivel de un BVH asignando a cada procesador el trabajo de procesar un nodo distinto. El trabajo muestra que procesar los primeros niveles de un BVH con el primer algoritmo y los niveles más bajos con el segundo da buenos resultados. El artículo [10] describe resultados de probar la diferencia en eficiencia entre un algoritmo de recorrido de un BVH que usa una pila para guardar los nodos que debe recorrer y un algoritmo que extiende la estructura del árbol para no usar una pila (de manera parecida a como funciona la lógica en este trabajo). El trabajo muestra que en CUDA el algoritmo que mantiene una pila es más rápido, pero tiene limitaciones a la hora de implementarlo en una arquitectura de GPU. El artículo [8] describe una estrategia de *ray tracing* usando grupos de rayos coherentes para mayor eficiencia. Se basa en generar un BVH de grado 8 para reducir el número de niveles. Los rayos principales son ordenados de acuerdo al recorrido de una curva-z en la imagen final y los rayos secundarios son ordenados de acuerdo a su cercanía espacial y direccional. A partir de esto, se calcula una pirámide que englobe a cada grupo de rayos a procesar y se utiliza para evitar calcular la intersección de los rayos con los nodos del BVH cuya BBox no interseca. El algoritmo guarda una lista de todos los nodos hoja donde puede haber una intersección y calcula la intersección con primitivas luego de haber atravesado todo el BVH. El trabajo muestra resultados de una implementación en CUDA. El sistema *OptiX* ([14]) es un producto comercial de la compañía NVidia. 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. En el artículo [25] los autores presentan resultados de su implementación de un *ray tracer* que funciona en placas gráficas. Utilizan diferentes estructuras de aceleración (kd-trees, grillas, BVH) y comparan los resultados obtenidos con cada una. En [1] se presenta una arquitectura para *ray tracing* paralelo basado en reemplazar rayos de cuyo procesamiento ya terminó por rayos todavía no procesados en una cola de proceso. Demuestran resultados en una implementación de su algoritmo en CUDA.

Otros trabajos en el área incluyen [23], en el cual Woop et al. proponen una estructura llamada *B-kd-trees*, que es una estructura de aceleración similar la de los BVH, sólo un nodo interior del árbol se subdivide en dos subnodos de la misma manera que se subdividen los nodos interiores de un *kd-tree*, permitiendo así recorrer el árbol de manera ordenada, obteniendo mayor eficiencia. Al igual que los BVHs, pueden ser actualizados en caso de modificarse la posición de las primitivas en la escena. A partir de esta estructura, crean una arquitectura en FPGA para hacer *ray tracing* dinámico. En [3] se presenta un *ray tracer* que utiliza DirectX 11 para acelerar partes del proceso de *ray tracing*. Utiliza kd-trees como estructura de aceleración, eligiendo el plano de corte cíclicamente, y usa el rasterizador de DirectX para generar la información de la intersección de rayos primarios.

Aporte del trabajo presentado

La mayoría de los trabajos citados anteriormente son investigaciones experimentales que tratan en su mayoría con las etapas de creación de estructuras de aceleración y búsqueda de intersecciones, luego reemplazando esas etapas en un *ray tracer* tradicional. Sin embargo, para lograr una solución que funcione en tiempos interactivos es necesario que todo el *pipeline* de *ray tracing* utilice las capacidades de GPGPU. Los proyectos que así lo hacen son de código cerrado, como OptiX. Por lo tanto un aporte de este trabajo es la disponibilidad del código completo de un *ray tracer* implementado completamente en *OpenCL*, ya sea para extenderlo o para su uso didáctico.

Asimismo, la bibliografía encontrada es escasa acerca de la implementación de la etapa de síntesis de imágenes a partir de los resultados de *ray tracing* y la creación eficiente de rayos

secundarios en el contexto de GPGPU, por lo cual las soluciones a los diferentes problemas que se presentan en esas etapas constituyen un aporte al campo.

Capítulo 2

Marco Teórico

En esta sección se describirán brevemente las etapas más importantes en el proceso de generar una imagen a través de la técnica de *ray tracing*. Debido a que dicha técnica posee una historia muy larga, existe gran cantidad de implementaciones de *ray tracers* que generan imágenes utilizando distintas variaciones del algoritmo originalmente propuesto. Sin embargo, las etapas descritas a continuación representan los bloques constitutivos para la mayoría de los *ray tracers*.

2.1. Creación de rayos primarios

El primer paso para el proceso de *ray tracing* es el de producir los rayos que representan la cámara virtual en la escena. Estos rayos son llamados *rayos primarios*, y como se mencionó en el capitulo 1, en el modelo básico de *ray tracing* cada rayo primario es asignado a un pixel de la imagen final a producir. Los rayos primarios serán los primeros en ser intersecados con la geometría de la escena para descubrir el punto de impacto (si existe), el color de la superficie que impacta en ese punto, y si se producirá algún efecto óptico de reflexión o refracción. Cada rayo consistirá de su representación geométrica, así como otra metadata que puede incluir el píxel al que dicho rayo corresponde, el tiempo en que es lanzado (si se trata de una escena animada), información sobre el medio en que se origina, etc.

En la figura 2.1 se pueden apreciar las características geométricas de un rayo. Se define un plano cercano a la cámara que representa la imagen a representar. A partir de dicho plano se define la dirección de cada rayo a partir de la dirección desde el origen o hasta el centro del plano

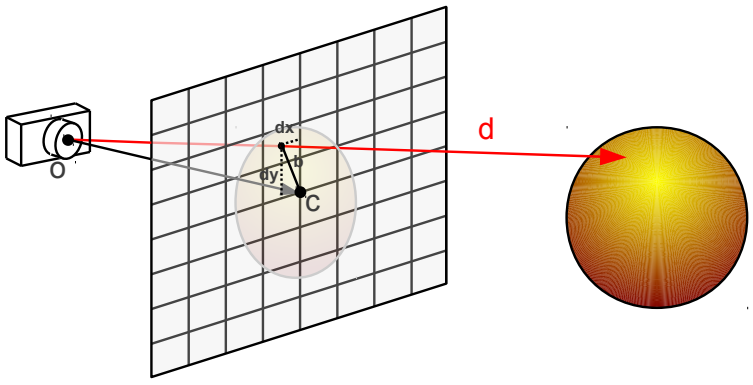


Figura 2.1: Rayo primario

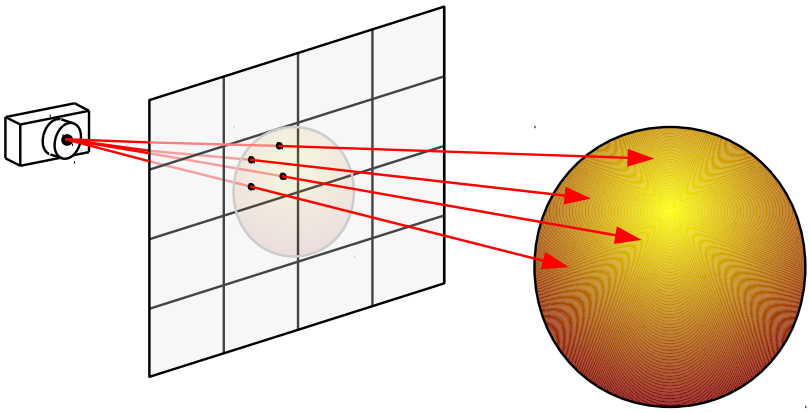


Figura 2.2: Pixel asociado a múltiples rayos

c, sumado al vector desde el centro del plano al centro del pixel correspondiente b. Este último vector se calcula como un desplazamiento del centro del plano hacia el punto que representa el pixel del rayo. Por lo tanto la dirección (no normalizada) de un rayo se calcula como $d = c - o + b$.

2.1.1. Múltiples rayos por píxel

En una cámara real, la superficie del sensor que representa un píxel en la imagen producida será afectado por una gran cantidad de rayos de luz. En el modelo básico de *ray tracing*, el color de un píxel es aproximado por la información de intersección de un rayo que impactaría en el centro de dicha superficie. Es posible mejorar la aproximación del color del píxel utilizando múltiples rayos para calcular su color. Cada rayo representa el impacto en un lugar diferente de la superficie del sensor, y el color final del píxel resulta de combinar de manera adecuada los resultados de intersección de cada uno. Esta técnica ayuda a mitigar los problemas de tratar de representar una función continua (una escena real) con una función discreta (píxeles en una imagen).

La figura 2.2 muestra un esquema ejemplificando la creación de múltiples rayos para representar un pixel.

2.2. Intersección

La etapa de intersección de rayos es la parte más importante de un *ray tracer*. Su función es calcular la intersección de los rayos con la geometría de la escena. Algunas de las características que interesan saber acerca de la intersección son el punto de intersección, la superficie impactada y la normal de la superficie en el punto de impacto, entre otras.

2.2.1. Estructuras de aceleración

La geometría de las escenas que debe intersecar un rayo consisten de un conjunto de formas geométricas denominadas *primitivas*. Por lo general las primitivas más usadas son triángulos, aunque las escenas pueden definirse utilizando también polígonos, prismas, esferas, cónicas e inclusive fractales. De hecho, puede utilizarse cualquier forma geométrica para la cual se pueda calcular al menos el punto de intersección más cercano con una semirrecta. Cada una de estas

primitivas tendrá asignado también un conjunto de propiedades físicas que tratan de modelar el material del cual está compuesto el elemento que representan en una escena.

Para encontrar la intersección de un rayo con la escena una implementación poco sofisticada de un *ray tracer* debería tratar de intersecar el rayo con cada primitiva de la escena, y devolver la intersección más cercana a su origen. Debido a que las escenas pueden contener centenares de miles de primitivas, y necesitarse millones de rayos para crear una imagen de buena calidad, claramente no es posible lograr esto en tiempo real. Para reducir los tiempos de búsqueda se utilizan estrategias de clasificación, que consisten en dividir las primitivas de la escena en conjuntos que pueden ser rápidamente descartados para propósitos de intersección sin tener que visitar cada uno de sus elementos. Las estructuras de datos que se crean para clasificación son llamadas *estructuras de aceleración*. Algunas de las más usadas son kd-trees, grillas regulares y árboles de jerarquía de volúmenes (BVH, o bounding volume hierarchy). 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.

La estrategia de todas las estructuras de aceleración es dividir al conjunto de primitivas en subconjuntos más pequeños y para cada subconjunto calcular una *bounding box*, o *BBox*, que es el prisma más pequeño que contiene a todas las primitivas del subconjunto. Calcular la intersección entre un rayo y un prisma se puede hacer rápidamente, y si se determina que la BBox de un subconjunto no interseca a un rayo, entonces se pueden descartar todas las primitivas de ese subconjunto.

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 división de la geometría de la escena en dos partes. Para cada nodo interno del árbol se calcula 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. Los nodos hojas contienen información acerca de las primitivas que representan. Un ejemplo de la estructura de un BVH se puede apreciar en la figura 2.3b. En la figura 2.5 se superimpuso los BBoxes de un BVH sobre el modelo de una mano[1].

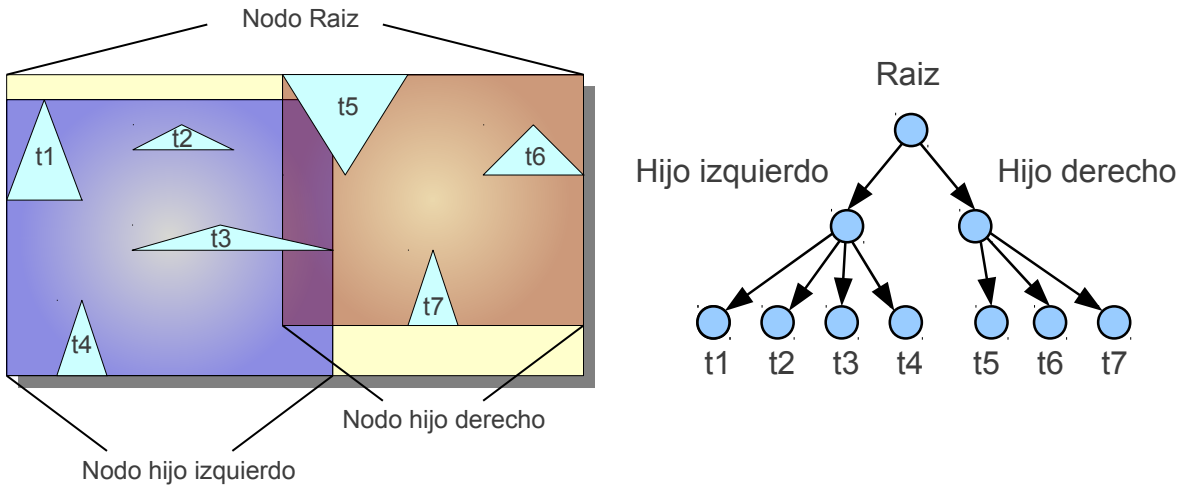
Para construir un BVH se comienza calculando la BBox de la escena completa y asignando al nodo raíz esa BBox. Una vez hecho esto recursivamente se aplica el siguiente método:

- Si el nodo contiene menos de un mínimo preestablecido de primitivas, se lo toma como nodo hoja y se termina el proceso.
- Si el nodo contiene suficientes primitivas :
  - Se elige un eje por el cual dividirlas
  - Se ordenan las primitivas del nodo por orden del menor valor de sus puntos en el eje elegido
  - Se elige un valor apropiado en el eje para dividir el subconjunto
  - Todas las primitivas cuyo punto menor en ese eje sea menor que el valor elegido irán a un subconjunto y el resto al otro.

Este método posee algunas características importantes a notar:

- El árbol generado por este método es un árbol binario.

[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

Figura 2.3: BVH

- Los subconjuntos en que se divide un nodo son disjuntos, por lo cual no es necesario dividir primitivas ni repetir información.
- Las BBoxes de los subconjuntos en que se divide un nodo no son necesariamente disjuntas.

El segundo de estos items es beneficioso dado que implica que no es necesario modificar la escena dividiendo primitivas, y dado que no hay primitivas repetidas, ninguna de ellas será usada más de una vez tratando de intersecar un rayo.

El tercero de los items es una desventaja respecto a otras estructuras como *grillas regulares* o *kd-trees* donde las BBoxes de nodos de un mismo nivel[2] son disjuntas. Esto es un problema porque implica que se recorre parte del espacio dos veces para encontrar una intersección primitiva-rayo.

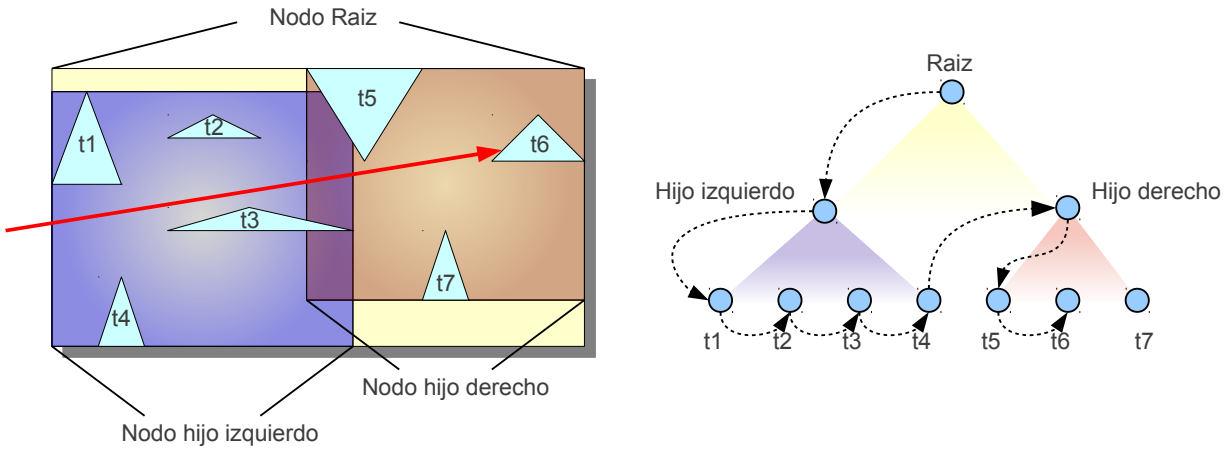
Para intersecar un rayo con una escena usando un BVH se procede recursivamente comenzando con el nodo raíz y ejecutando los siguientes pasos:

- Se verifica si existe intersección de la BBox del nodo con el rayo :
  - Si no hay intersección con la BBox, entonces no hay intersección con ninguna de las primitivas del nodo.
  - Si hay intersección con la BBox :
    - Si se trata de un nodo interior, se repite el procedimiento con ambos hijos.
    - Si se trata de un nodo hoja, se interseca el rayo con todas las primitivas que contiene.

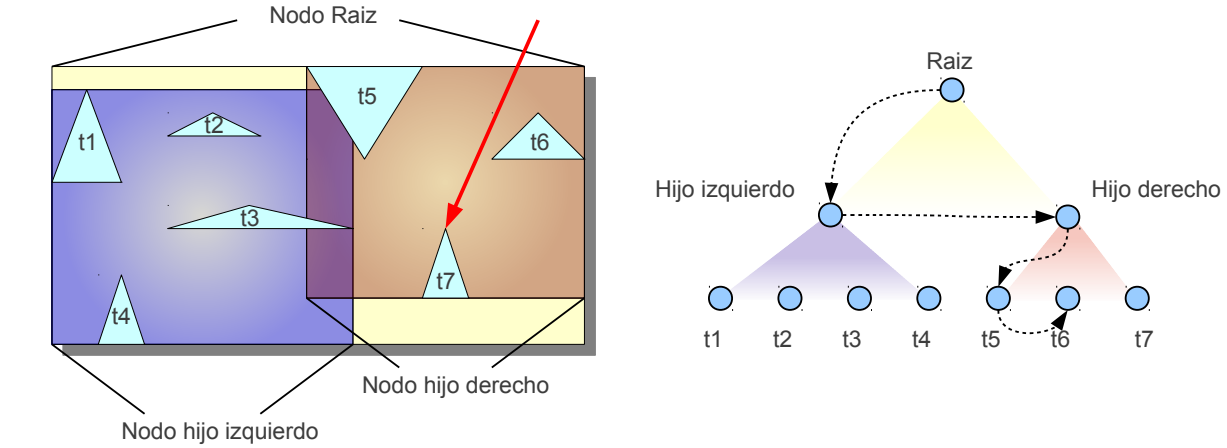
El algoritmo comienza con el nodo raíz. Al final de este algoritmo en caso de haber alguna intersección, se toma la más cercana al origen.

La figura 2.4 muestra un ejemplo de un BVH y la forma en que se recorren sus nodos para encontrar la intersección con el rayo visualizado.

[2] en el caso de *grillas regulares*, se asume que todos los nodos están al mismo nivel



(a) Recorrido para un rayo que atraviesa todos los hijos de un nodo



(b) Recorrido para un rayo que solo atraviesa un hijo de un nodo

Figura 2.4: Ejemplo de recorridos de un BVH para diferentes rayos

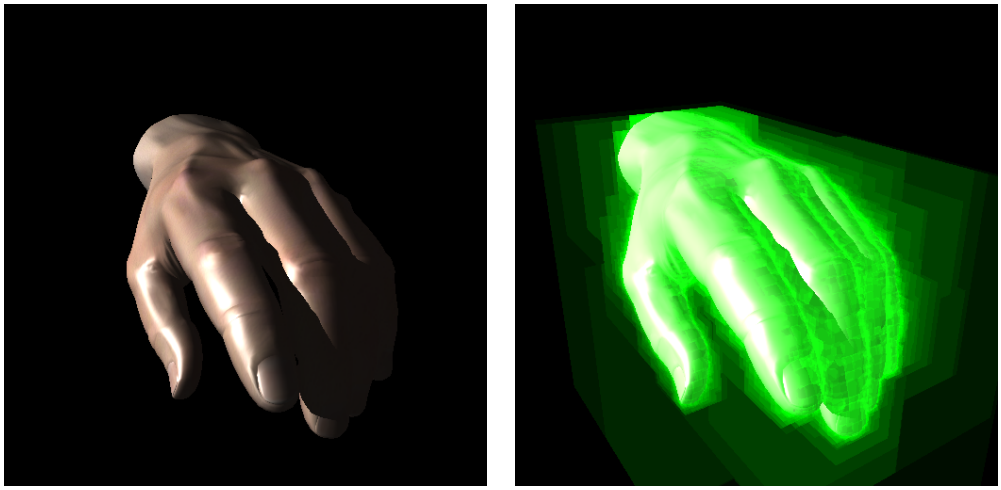
2.2.2. Estrategias para dividir el espacio de una escena

Al momento de elegir como dividir las primitivas de una escena para crear un BVH es necesario tener en cuenta el impacto que tiene la división al momento de ser atravesado durante la búsqueda de una intersección con un rayo. Por lo tanto se debe tener en cuenta el costo computacional de evaluar una intersección rayo-triángulo (t_{tri}) y el costo computacional de decidir que hijo de un nodo interior recorrer (t_{int}).

Considérese el caso en que tengamos un sólo nodo en nuestra escena con n primitivas asociadas. El costo de atravesar dicho nodo será:

$$c(N) = \sum_{t=1}^{n} t_{tri}(t)$$

Ahora considérese una escena con un sólo nodo interior N y dos nodos hijos N_1 y N_2 respectivamente. Si se desea calcular el costo que tendrá un rayo aleatorio que atraviesa dicha escena, podemos estimarlo a partir de la probabilidad p_1 y p_2 de que dicho rayo atraviese N_1 y N_2 respectivamente. En dicho caso el costo para atravesar nuestro nodo será:



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

Figura 2.5: BVH superimpuesto sobre el modelo de una mano

$$c(N) = t_{int}(N) + p_1 \times c(N1) + p_2 \times c(N2)$$

La estrategia óptima para dividir una escena es dividir la escena de manera de que el costo de atravesar la raíz sea mínimo. Sin embargo minimizar la función costo descripta, cuando un BVH puede contener cientos de miles de nodos, es restrictivo. Por lo tanto existen heurísticas que permiten decidir de manera local como (y si) dividir cada nodo.

Una de las maneras más simples es dividir cada nodo tratando de mantener en cada subnodo la misma cantidad de primitivas, o con una diferencia de una primitiva. Otra heurística es dividir un nodo siempre por la mitad a través de alguno de sus ejes. Aunque estas heurísticas son muy rápidas a la hora de construir un árbol, en el caso de que las primitivas no estén distribuidas uniformemente en la escena serán lentas de atravesar en el momento de usar la estructura.

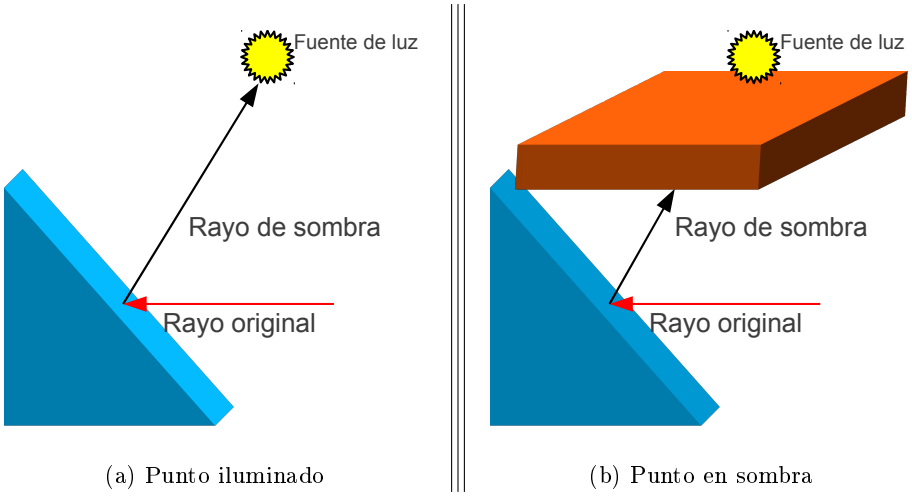
En [15] los autores describen SAH (Surface Area Heuristic), una heurística para dividir el espacio que toma en cuenta la superficie de las BBoxes del nodo padre y los nodos hijos. Es una de las heurísticas más utilizadas al día de hoy dado que permite crear BVHs que pueden ser atravesados rápidamente en la mayoría de los casos. Dado que se usa para estimar la mejor manera de dividir localmente un nodo, intenta minimizar el costo de atravesar un nodo si los dos subnodos fuesen nodos hojas. Esto implica que se quiere minimizar la siguiente función de costo:

$$c(N) = t_{int} + p_1 \times \sum_{t=1}^{n_1} t_{tri}(t) + p_2 \times \sum_{t=1}^{n_2} t_{tri}(t)$$

Para esto, provee una manera de calcular p_1 y p_2: la probabilidad de un rayo aleatorio de atravesar N_1 será estimada como S/S_1, donde S es la superficie de la BBox de N, y S_1 la de N_1. De la misma manera se calcula p_2. Si se asume t_{int} constante para todos los nodos interiores y t_{tri} constante para cada triángulo en la escena, el algoritmo necesario para elegir las divisiones de cada nodo se puede implementar eficientemente.

2.2.3. Rayos de sombra

Una vez que se determina el punto de intersección de un rayo con la escena, es necesario determinar si dicho rayo es iluminado por una fuente de luz. Para ello se genera un nuevo rayo,



(a) Punto iluminado (b) Punto en sombra

Figura 2.6: Rayo de sombra

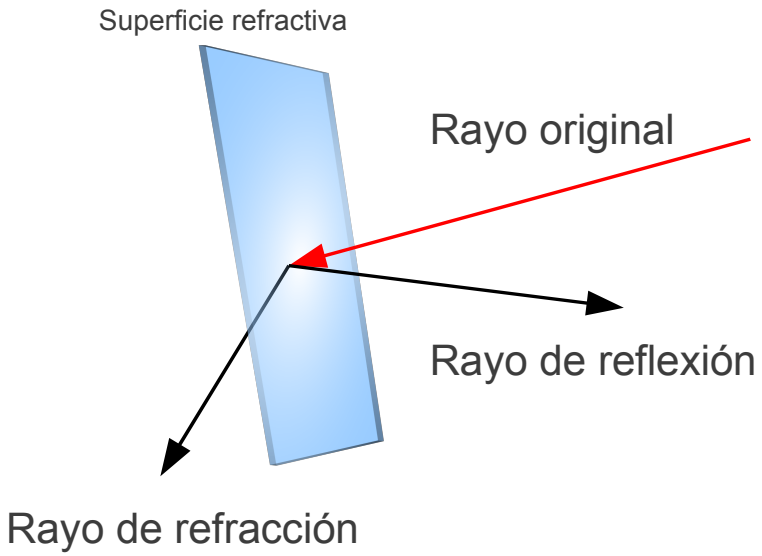
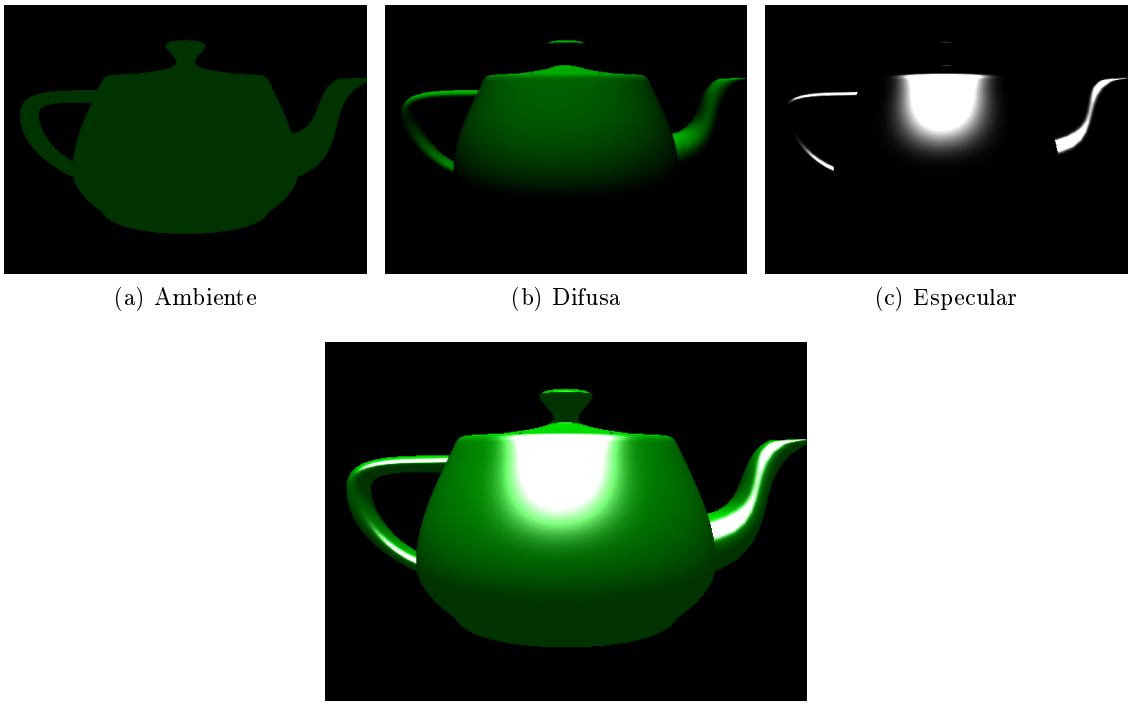


Figura 2.7: Creación de rayos secundarios

llamado *rayo de sombra*, cuyo origen es el punto de intersección y cuya dirección apunta hacia la fuente de luz. Si dicho rayo interseca alguna primitiva en la escena, entonces el punto de impacto no está iluminado. Dado que esta etapa no necesita devolver la intersección más cercana, sino sólo la existencia de alguna, su implementación es ligeramente más simple. La figura 2.6 da un ejemplo gráfico del uso de un rayo de sombra para determinar si un punto alcanzado por un rayo está iluminado por una fuente de luz.

2.3. Rayos Secundarios

En una escena real, existen efectos ópticos en los cuales intervienen los rayos de luz. Algunos de los más comunes son la reflexión (rebote de la luz) y la refracción (cambio de dirección de



(a) Ambiente (b) Difusa (c) Especular

(d) Resultado

Figura 2.8: Componentes del modelo de iluminación de Phong

la luz al pasar de un medio a otro). Cuando un rayo interseque una superficie cuyo material es reflectivo o refractivo, se crearán nuevos rayos que representan la reflexión o refracción de la luz que se produce en esos materiales. Estos rayos son llamados *rayos secundarios*. Dichos rayos serán procesados de la misma manera que los rayos primarios: se calculará su intersección con la escena y posiblemente se crearán nuevos rayos secundarios a partir de los mismos. Por lo tanto se entrará en un ciclo que terminará cuando no se creen más rayos secundarios, o se llegue a un máximo permitido de niveles. El conjunto de un rayo primario y los rayos secundarios creados a partir del mismo forman un *árbol de rayos* cuya raíz es el rayo primario. La información conjunta de intersección de todos los rayos del árbol será usada para determinar la contribución de color al píxel con el que está asociado.

La figura 2.7 muestra dos rayos secundarios creados a partir del impacto de un rayo contra una superficie refractiva.

2.4. Composición de la imagen final

Un *modelo de iluminación* es una función que calcula el color de un punto en una superficie al ser observada desde un ángulo particular. El paso final para procesar un rayo es aplicar el modelo de iluminación elegido para obtener el color perteneciente al punto de impacto del rayo con la escena. El color calculado de esta manera para un rayo primario y todo el árbol de rayos que le corresponde es usado para definir el color del pixel al que corresponden.

Existen modelos de iluminación muy variados que permiten modelar diferentes características de la interacción de la luz presente en una escena con una superficie de la misma. Dichos modelos pueden ser fotorrealistas, tratando de modelar solamente efectos ópticos reales, o artísticos, tratando de crear una representación abstracta de la escena. En cualquiera de los casos existirá

un vínculo entre la definición de una escena, la etapa de búsqueda de intersección, y la etapa de composición de la imagen: esta última necesitará información de las dos primeras para poder realizar sus cálculos. Por lo tanto el diseño del algoritmo de intersección en un *ray tracer* debe tener en cuenta toda la información que necesitarán los modelos de iluminación que pueden ser usados.

Un ejemplo de un modelo de iluminación es el modelo de *Phong*. Es un modelo de iluminación local, dado que el color que asigna a cada superficie es independiente del color o la luz que reciben otras. Este modelo define el color final de un punto en una superficie como la combinación de tres componentes distintas: luz ambiente, reflejo difuso, y reflejo especular. El material de cada superficie define como se combinan dichas componentes para obtener el color en cada uno de sus puntos.

La figura 2.8 ejemplifica los componentes de iluminación del modelo de Phong y su resultado final. La figura 2.8a muestra la componente ambiental, que es la representación de la luz dispersa en el ambiente de la escena que impacta al objeto. En la figura 2.8b se aprecia la componente difusa, que es la luz reflejada por la superficie de un objeto opaco, y representa la dispersión de luz en muchas direcciones a partir del impacto de rayos de luz en el objeto. Finalmente, la figura 2.8c representa el reflejo especular, que es la luz que se reflejan en una sola dirección, característico de los objetos lustrados o brillosos.

Para describir modelos cuyas características materiales varíen en diferentes puntos de su superficie se utilizan generalmente funciones bidimensionales llamadas *texturas*. Dichas funciones generalmente se codifican como imágenes que se vinculan a una superficie. Para utilizar texturas cada punto de la superficie tiene asignado un par de parámetros que se corresponden a un punto que será evaluado en la textura. A partir de dicha evaluación se podrán obtener o modificar características del punto en la superficie tales como color, brillo, reflexividad o el vector normal en ese punto.

En el siguiente capítulo será presentado el diseño propuesto en este trabajo y la implementación de un *ray tracer* que permita realizar todas las tareas descriptas anteriormente en arquitecturas paralelas mediante el uso de *OpenCL*.

Capítulo 3

Implementación

3.1. Arquitectura de la solución

El diseño propuesto encapsula las diferentes etapas descriptas en el capítulo 2 como módulos individuales. Cada uno de esos módulos implementa *kernels* de *OpenCL* para acelerar los cómputos que debe realizar. Debido a esto los diferentes módulos se comunican a través de estructuras de datos que existen en memoria global dentro del *dispositivo OpenCL*. A continuación se va a discutir algunas características del diseño que surgen debido a la arquitectura paralela donde se ejecutará.

3.1.1. Características generales

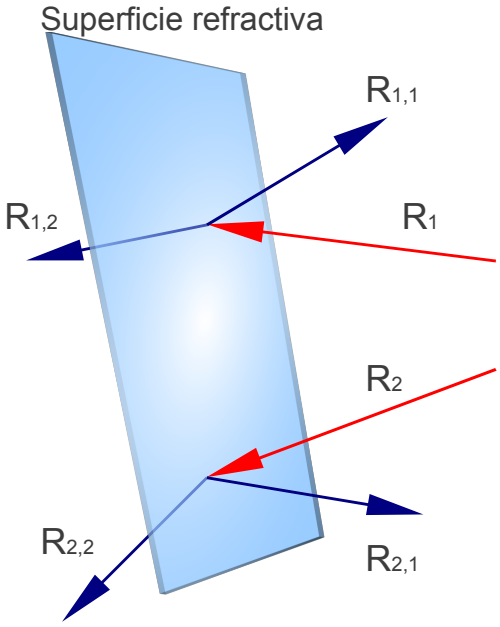
Esquema iterativo

Una de las limitaciones más importantes de *OpenCL* es la falta de recursión en los kernels. Normalmente un algoritmo de *ray tracing* funciona de manera recursiva: la función que calcula el impacto de un rayo se llama recursivamente para calcular el impacto de los rayos secundarios resultantes. Una vez que las llamadas recursivas retornan, el algoritmo tiene toda la información necesaria para determinar el color y otras propiedades de la muestra perteneciente a ese rayo. Dada la falta de recursividad en *OpenCL*, es necesario utilizar un esquema iterativo para lograr el mismo fin.

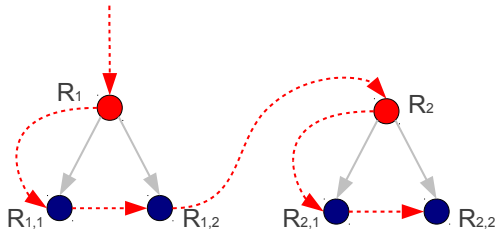
Para lograr un esquema iterativo de *ray tracing*, será necesario atravesar un nivel de un grupo de árboles de rayos por iteración. La figura 3.1 muestra dos árboles de rayos y la diferencia de recorrido de los nodos en un esquema recursivo y un esquema iterativo para procesar los rayos en un *ray tracer*.

Organización en tiles

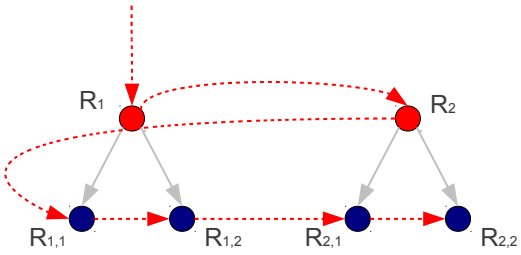
La arquitectura paralela que expone *OpenCL* hace que sea ventajoso procesar gran cantidad de rayos simultáneamente. Por lo tanto la mayoría de las etapas de la solución trabajan con grandes grupos de rayos a la vez para mejorar su eficiencia. Sin embargo, mantener en memoria todos los rayos de una imagen de grandes dimensiones en algunos casos no es posible, especialmente si se quiere procesar más de un rayo por pixel. Para solucionar este problema se adopta la estrategia de procesar rayos primarios y sus rayos derivados en grandes grupos o *tiles*. Esto es posible porque durante el proceso de *tracing* la información obtenida de un rayo primario y sus rayos derivados no afecta a la información obtenida del resto. Los *tiles* deben ser suficientemente grandes como para asegurar que todas las unidades de proceso del dispositivo sean utilizadas, al menos durante la etapa de *tracing* primario. De esta manera es posible disminuir



(a) Dos rayos y sus respectivos rayos secundarios



(b) Recorrido recursivo de los arboles de rayos



(c) Recorrido iterativo de los arboles de rayos

Figura 3.1: *Ray tracing* iterativo vs *ray tracing* recursivo

la cantidad de memoria que el programa necesita para estructuras intermedias sin disminuir su eficiencia. El tamaño de los *tiles* dependerá de la cantidad total de *elementos de procesamiento* en el dispositivo *OpenCL*.

Uso de triángulos como primitivas

Como se comentó en el capítulo 1, existen multitud de primitivas posibles para utilizar en la definición de una escena. Sin embargo, la vasta mayoría de los modelos tridimensionales que se definen para trabajar en gráficos por computadora consisten únicamente de triángulos. Ésto se debe a que cualquier polígono puede formarse a partir de triángulos. Por lo tanto cualquier superficie puede aproximarse por polígonos compuestos por triángulos.

Tener un tipo de primitiva única en uso permite optimizar el uso de memoria y los algoritmos que procesan la escena. El modelo de cómputo SIMD también se beneficia debido a que el mismo código deberá ejecutarse para tratar con todas las primitivas.

Pasos de la solución

La figura 3.2 muestra un diagrama de los módulos del *pipeline* de la solución. Las flechas en el diagrama indican el flujo de datos entre las diferentes etapas.

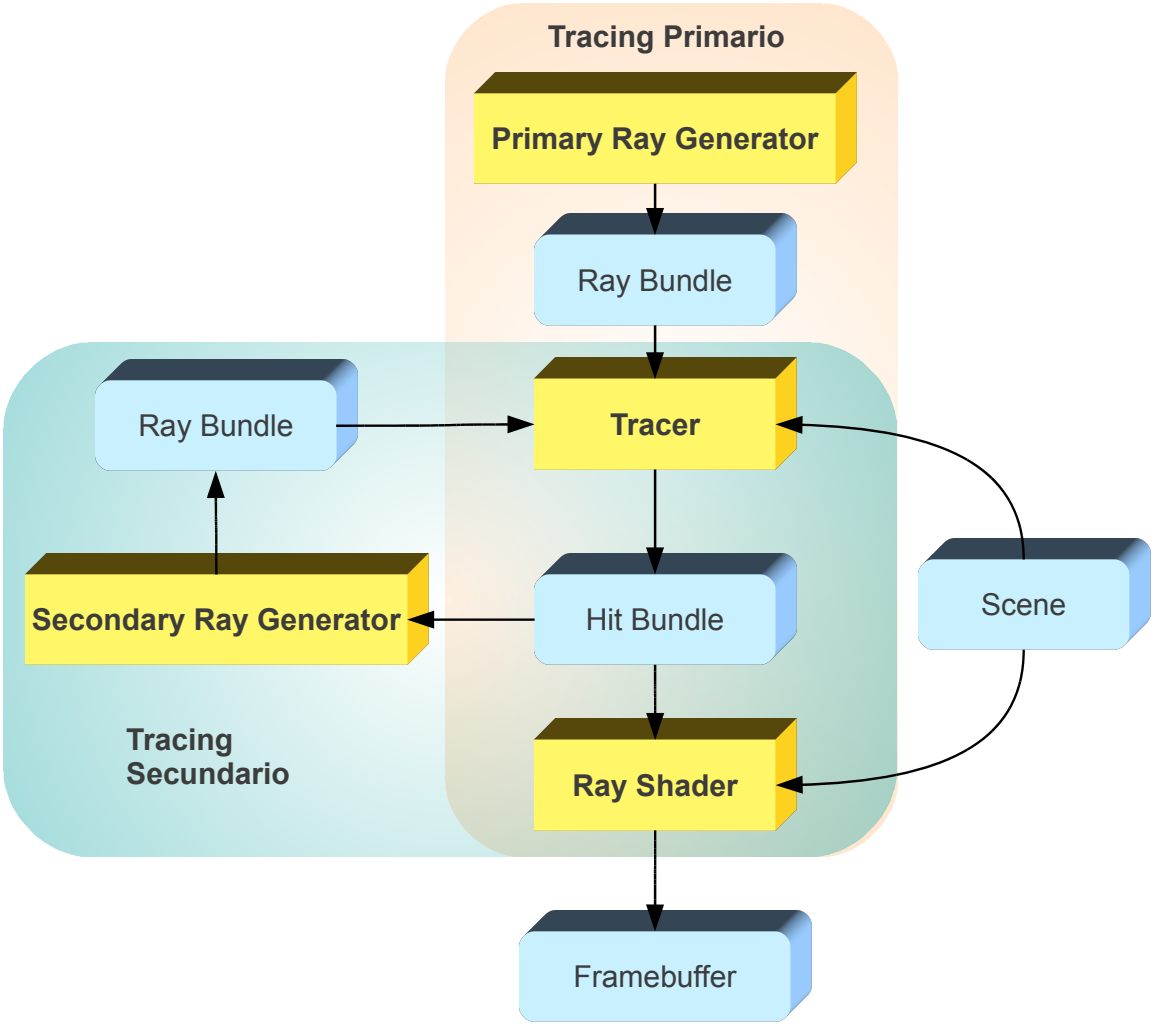


Figura 3.2: Diagrama de ejecución del *pipeline* de *ray tracing*

La entrada al *pipeline* de *ray tracing* es la escena que se quiere representar y las propiedades que representan la cámara virtual que será usada. La descripción de la escena consiste de uno o más modelos geométricos que tendrán asociadas diferentes propiedades. Las propiedades geométricas de los modelos en la escena está definida por un conjunto de vértices y un conjunto de triángulos formado por esos vértices. Cada vértice, además de representarse por su posición geométrica puede poseer propiedades tales como un vector normal a la superficie, un vector tangencial a la superficie y coordenadas para texturas. Los modelos también tendrán asociado un material que describe como interactúa con la luz en la escena. Dicho material define las propiedades de reflexión y refracción del modelo, así como su brillo, color y texturas asociadas.

El primer paso en el *pipeline* es generar el conjunto de rayos primarios para representar la cámara virtual. Dicho conjunto es llamado paquete de rayos, o *ray bundle*, y será procesado por otro módulo, llamado *tracer*, que calcula la información de intersección de los rayos con la

Algoritmo 1 Pseudocódigo del cálculo de un frame

```
1: Cargar escena
2: while (Existen tiles para calcular) do
3:     PrimaryRayGenerator → Crear rayos primarios
4:     Tracer → Calcular impacto de los rayos
5:     Ray Shader → Actualizar imagen con la información de impacto
6:     while (No se llegó a la cantidad máxima de rebotes) do
7:         Secondary Ray Generator → Crear rayos secundarios
8:         if (No se crearon rayos secundarios) then
9:             Break
10:        end if
11:        Tracer → Calcular impacto de los rayos
12:        Ray Shader → Actualizar imagen con la información de impacto
13:    end while
14: end while
```

escena y la guarda en una estructura llamada *hit bundle*. Esta estructura contiene información que identifica las propiedades de las intersecciones encontradas para los rayos de un ray bundle. En esta etapa también se determina si el punto de intersección está alumbrado por una fuente de luz a partir de generar y procesar rayos de sombra.

Con esa información, el módulo llamado *ray shader* puede buscar 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 que se guardan en un nuevo *ray bundle*.

Los rayos secundarios creados 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 repetirá hasta que no se creen nuevos rayos secundarios o se llegue a un número máximo prefijado de iteraciones.

Un esquema del pseudocódigo del algoritmo que controla el *ray tracer* se puede apreciar como algoritmo 1 y la figura 3.3 muestra el diagrama de secuencia del cálculo de un tile.

3.2. Componentes de la solución

En esta sección se detallarán las propiedades y la lógica de ejecución de los módulos principales del programa.

3.2.1. Primary Ray Generator

La función de este módulo es generar un *ray bundle* que contiene los rayos para el *tile* que se está procesando a partir de los parámetros de la cámara a utilizar.

Sin embargo, dado el modelo de cómputo de *OpenCL*, el orden en que se generan estos rayos impacta en el desempeño del *ray tracer*. Los work items de un mismo work group procesarán rayos adyacentes en memoria. Si se generan rayos de manera que los rayos adyacentes en memoria no sean muy similares, entonces durante la tarea de *tracing*, el recorrido de cada rayo a través de la escena divergirá mucho para diferentes work items de un mismo work group. Ésto ocasionará que disminuya el rendimiento de esa etapa. Por lo tanto se intenta generar rayos en paquetes cuyas direcciones sean lo más similares posibles.

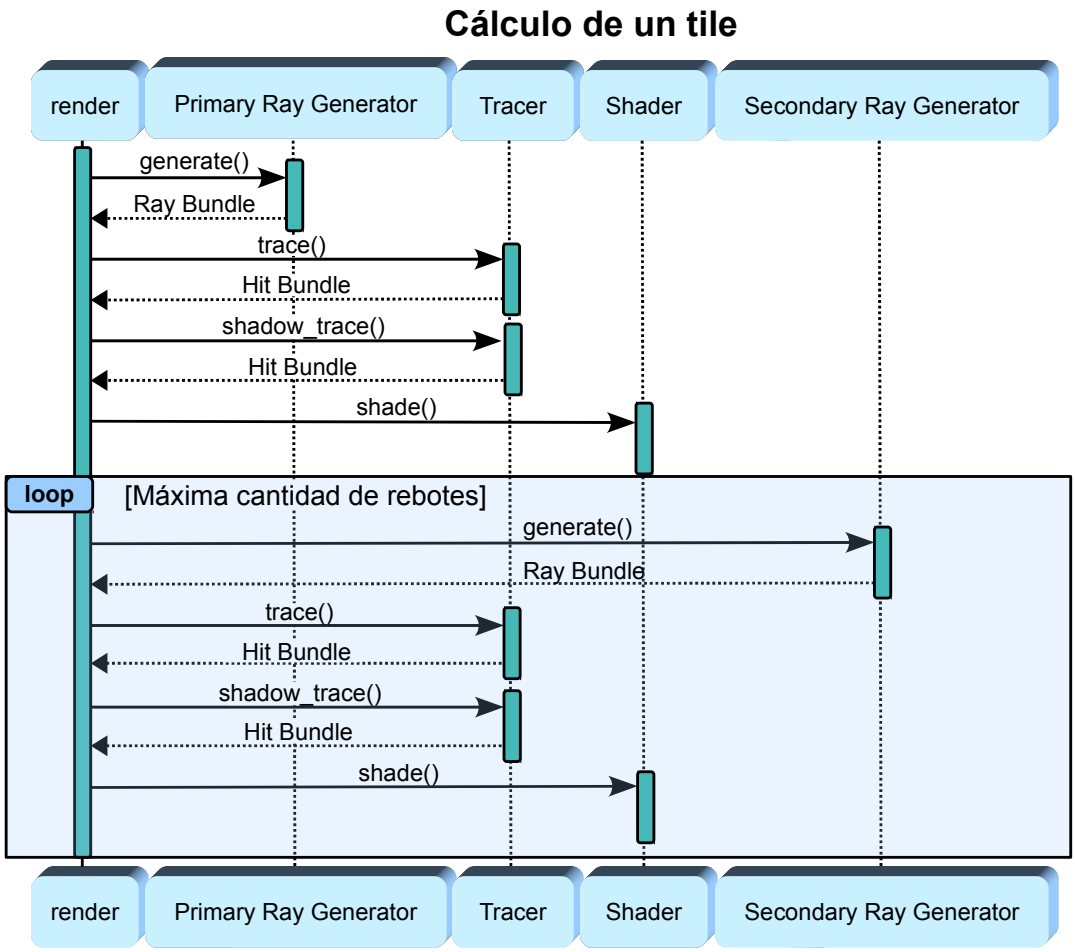


Figura 3.3: Diagrama de secuencia para el cálculo de un tile de una imagen

La figura 3.4 ejemplifica el esquema utilizado para generar los rayos primarios. De esta manera se trata de mejorar la coherencia de acceso a memoria entre los work items de un mismo work group. Los grupos generados son cuadrados de tamaño fijo. Dado que es imposible asegurar que los lados de dichos cuadrados dividan exactamente el ancho o largo de la imagen, los rayos correspondientes a las barras horizontales y verticales que sobran se agruparán en rectángulos de manera de tratar de mantener el tamaño de los grupos.

Además de generar las propiedades geométricas de los rayos primarios, este módulo también genera los metadatos asociados a un rayo para poder ser procesados por los siguientes módulos en el *pipeline* del *ray tracer*. Para cada rayo que se genera en un *ray bundle* se guardan:

- Las propiedades geométricas del rayo
- El pixel de la imagen al que corresponde este rayo
- La contribución del rayo al color del pixel que le corresponde

La contribución es una representación del porcentaje del color del píxel que representa este rayo. En el caso en que se cree un rayo por píxel de la imagen este valor será 1, indicando que el 100 % del color del píxel proviene de la información de este rayo y sus rayos derivados. Si el rayo interseca con una superficie reflectiva o refractiva, se generarán rayos secundarios que tendrán

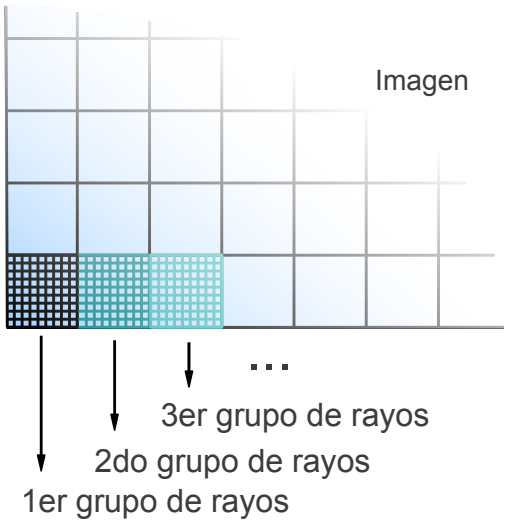


Figura 3.4: Orden de rayos primarios

una contribución menor o igual a la contribución del rayo original. Esta información luego será utilizada por el *ray shader* para actualizar la imagen final.

Generación de múltiples rayos por pixel

Para poder generar múltiples rayos por píxel, este módulo puede ser configurado con una lista de parámetros para los rayos que serán creados para un mismo píxel. Estos parámetros incluyen el desplazamiento de cada rayo respecto del rayo ideal que impacta en el centro del sensor de una cámara ideal, y la contribución del rayo al color del píxel.

De esta manera para pasar de un modelo de *ray tracing* clásico a un modelo distribuido sólo es necesario cambiar la configuración de generación de rayos primarios. De la misma manera, agregar otros efectos de cámara no presentes en la implementación, como *depth of field*[1], también implicaría cambiar la implementación de este módulo únicamente.

3.2.2. *Tracer*

El módulo llamado *Tracer* es la parte más critica del *pipeline*. Su función es calcular la intersección entre los rayos de un *ray bundle* y las primitivas de la escena. Una vez encontrada la intersección (o su ausencia) de todos los rayos, guarda el resultado de sus cómputos en un *hit bundle*. La información de intersección que se guarda en estos paquetes es:

- 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 intersecó el rayo
- Un identificador de la primitiva intersecada
- El punto de intersección

[1] Efecto que emula el punto de enfoque de una cámara.

- La normal de la superficie en el punto de intersección
- La coordenadas para texturado del punto de intersección

Uno de los algoritmos más usados para calcular la intersección entre rayos y triángulos (la primitiva que se utiliza en el trabajo presentado) fue descripto por Möller y Trumbore en [16]. Este método es eficiente tanto en tiempo como en memoria utilizada para encontrar la intersección, y fue por lo tanto elegido para calcular intersecciones en este trabajo.

En el capítulo 1 se presentaron diferentes estructuras de aceleración: kd-trees, BVHs y grillas regulares. Cada una posee ventajas y desventajas que las hacen útiles para diferentes situaciones. Los kd-trees por lo general obtienen la mayor eficiencia a la hora de atravesarlos, pero para su construcción es necesario en algunos casos modificar las primitivas de la escena, y una vez creado no es en la mayoría de los casos posible modificarlo si las propiedades geométricas de la escena cambia. Las grillas regulares son creadas rápidamente y pueden crearse nuevamente cuando la escena se modifica, pero su rendimiento es mucho más bajo que estructuras jerárquicas como kd-trees y BVHs.

Estructura de aceleración propuesta

Se eligió utilizar BVHs para este trabajo porque posee la mejor combinación de rendimiento y flexibilidad. Su rendimiento es similar al de kd-trees, pero no sufre de la necesidad de alterar las primitivas de la escena.

Para determinar si un rayo interseca con una primitiva de la escena se comienza por examinar el nodo raíz del BVH. Si no hay intersección con el BBox de la raíz, se sabe que el rayo no interseca con la escena. Si existe una intersección, se desciende al nodo izquierdo y se repite el proceso. Si llegamos a un nodo hoja, se trata de intersecar el rayo con todas las primitivas que representa esa hoja. Si el rayo no interseca un nodo (ya sea por no intersecar su BBox o no intersecar 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.

Para poder llevar a cabo esta búsqueda, cada nodo del árbol contiene información sobre cual es su nodo padre. Si en algún momento se sube nuevamente al nodo raíz habiendo atravesado sus dos hijos, es porque no se intersecó ninguna primitiva y termina 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 [1]; 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 items 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 representación gráfica del modo de atravesar el BVH se ejemplifica en la figura 2.4 con un ejemplo sencillo de un BVH de dos niveles.

El pseudocódigo correspondiente a este algoritmo se lista como algoritmo 2, y una representación gráfica del mismo se ejemplifica en la figura 2.4.

Una vez que se sabe la primitiva que interseca un rayo, se puede calcular toda la información que el resto del *pipeline* del *ray tracer* va a necesitar, como la normal de la primitiva en el punto de impacto, valores interpolados de las propiedades de los vértices de la primitiva, y su material, entre otros.

Algoritmo 2 Pseudocódigo del algoritmo de *tracing*

```
1: function TRACE(rayo,bvh)
2:     mejorInterseccion ← Ninguna
3:     subiendo ← false
4:     nodoAnterior ← bvh.raz
5:     nodoActual ← bvh.raz
6:     while (true) do
7:         if (subiendo) then
8:             if (ultimo = raz) then
9:                 break
10:            end if
11:            if (ultimo = nodoActual.hijoIzquierdo) then
12:                nodoAnterior ← nodoActual
13:                nodoActual ← nodoActual.hijoDerecho
14:                subiendo ← false
15:                continue
16:            else
17:                nodoAnterior ← nodoActual
18:                nodoActual ← nodoActual.padre
19:                subiendo ← true
20:                continue
21:            end if
22:        end if
23:        if (Rayo interseca a nodoActual.BBox) then
24:            if (nodoActual es un nodo terminal) then
25:                Actualizar mejorIntersección si hay otra mejor entre las primitivas de nodoActual
26:                continue
27:            else
28:                nodoAnterior ← nodoActual
29:                nodoActual ← nodoActual.hijoIzquierdo
30:                subiendo ← false
31:                continue
32:            end if
33:        else
34:            nodoAnterior ← nodoActual
35:            nodoActual ← nodoActual.padre
36:            subiendo ← true
37:            continue
38:        end if
39:    end while
40:    return mejorInterseccin
41: end function
```

Orden de atravesado de los hijos de un nodo

Una vez encontrada una intersección de un rayo con la escena, se puede descartar cualquier intersección más lejana al origen del rayo que la primera. Dado que cualquier intersección con las primitivas de un nodo sólo puede ocurrir dentro del espacio determinado por su BBox, en caso de que el BBox de un nodo sea más lejano que un punto de intersección ya hallado, puede ignorarse dicho nodo (y todos sus descendientes). Esto lleva a la posibilidad de introducir una pequeña optimización para el algoritmo de *tracing*: si se atraviesan los hijos de un nodo de más cercano a más lejano, se maximiza la posibilidad de poder ignorar el hijo lejano si se encuentra una intersección en el hijo más cercano. Para poder hacer esto durante la fase de creación del BVH de la escena se guarda información en los nodos acerca del orden geométrico de los hijos de un nodo. Esa información se utilizará para determinar el orden de atravesado.

Algoritmo 3 Pseudocódigo de la lógica de generación de rayos secundarios

```
1: function GenerarRayosSecundarios(rayos,intersecciones,escena)
2:     Arreglo RayosGenerados ← MarcarRayosAProducir(rayos,intersecciones,escena)
3:     Arreglo Índices ← SumaDePrefijos(RayosGenerados)
4:     Arreglo RayosNuevos ← GenerarRayos(rayos,intersecciones,escena,Índices)
5:     return RayosNuevos
6: end function
```

Consideraciones para el *tracing* de rayos secundarios

Los rayos primarios son generados de manera que para un mismo work group, los rayos tengan recorridos similares y por tanto recorran los nodos del BVH en el mismo orden. Al contrario, los rayos secundarios son creados a partir de la simulación de efectos ópticos en la escena, por lo cual no se tiene control acerca de su coherencia. Por lo tanto, si usamos work groups muy grandes para calcular el impacto de rayos secundarios, los rayos de un mismo work group pueden llegar a recorrer los nodos del BVH en ordenes muy diferentes. Mientras más grande sea un work group, más recorridos diferentes existirán, y por lo tanto un work item con un recorrido particular estará inactivo mientras se visitan nodos de un recorrido diferente. Por otro lado, si se reduce demasiado la cantidad de work items en un work group, se perderá concurrencia y por lo tanto rendimiento.

Búsqueda de intersecciones para rayos de sombra

Los rayos de sombra no necesitan un módulo para ser creados de la misma manera que los rayos primarios, dado que pueden ser creados a partir de la información de intersección del rayo al que corresponden y la información acerca de las fuentes de luz de la escena.

El algoritmo de intersección para rayos de sombra utiliza la misma estructura de aceleración que se utiliza para la intersección de rayos primarios y secundarios. La lógica para este algoritmo también es muy similar, con la diferencia de que no se hará el cálculo de las propiedades de intersección y el algoritmo terminará en caso de encontrarse cualquier intersección.

3.2.3. Secondary Ray Generator

El módulo llamado *Secondary Ray Generator* es responsable de la creación de rayos secundarios a partir de la información de intersección calculada en la etapa de *tracing*. Además de las propiedades geométricas de los rayos secundarios creados, esta etapa también calculará la contribución de los nuevos rayos al color del píxel al que corresponden.

La figura 2.7 muestra dos rayos secundarios creados a partir del impacto de un rayo contra una superficie refractiva.

El pseudocódigo de la lógica de creación de un rayo secundario se lista como algoritmo 4.

Dado que la creación de rayos secundarios no está asegurada a priori, el kernel *OpenCL* que hace los cómputos debe sincronizar el uso de la memoria de salida. La manera más sencilla de decidir el lugar en el espacio de memoria de salida que debe ocupar cada rayo nuevo creado es utilizar un contador sincronizado globalmente que se incremente atómicamente cada vez que se desea escribir 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 al momento de crear rayos secundarios, se optó por un esquema asíncrono de 3 fases:

- Por cada rayo de la etapa anterior procesado, un work item calcula la cantidad de rayos secundarios que producirá, y guarda esa cantidad en un arreglo auxiliar.

Algoritmo 4 Pseudocódigo para la generación de un rayo secundario

```
1:  function GenerarRayoSecundario(rayo,intersección,material)
2:      if (material es refractivo) then
3:          R ← aproximación de Schlik
4:          if (no hay reflexión total interna) then
5:              rayoRefractivo ← rayo refractivo
6:              rayoRefractivo.contribución ← (1-R) × rayo.contribución × material.reflectividad
7:              Guardar rayoRefractivo
8:          end if
9:          rayoReflexivo ← rayo reflectivo
10:         rayoReflexivo.contribución ← R × rayo.contribución × material.reflectividad
11:         Guardar rayoReflexivo
12:     else if (material es reflectivo) then
13:         rayoReflexivo ← rayo reflectivo
14:         rayoReflexivo.contribución ← rayo.contribución × material.reflectividad
15:         Guardar rayoReflexivo
16:     end if
17: end function
```

- 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.

La figura 3.5 da un esquema visual de este proceso y el pseudocódigo de su lógica se lista como algoritmo 3. El algoritmo 4 muestra el pseudocódigo para la generación de posibles rayos secundarios a partir de un solo rayo.

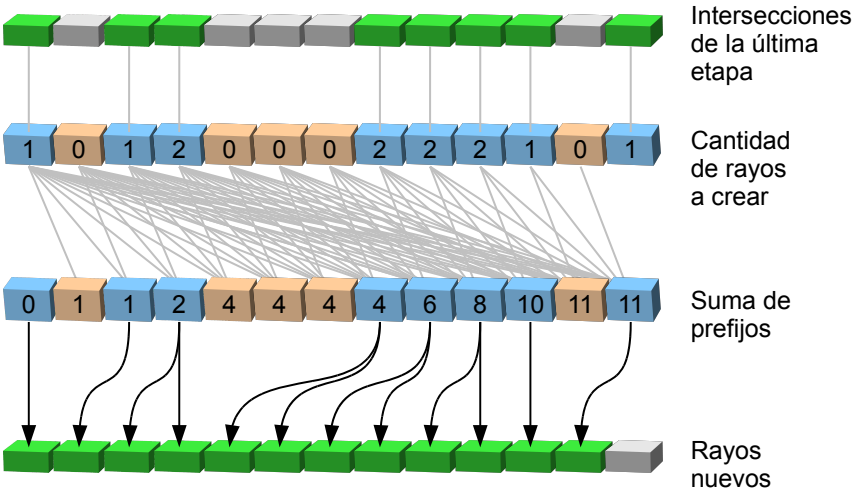


Figura 3.5: Pasos para crear y guardar rayos secundarios

3.2.4. Ray Shader

El módulo llamado *Ray Shader* tiene como objetivo actualizar la imagen a partir de la información de intersección de un grupo de rayos. En un *ray tracer* convencional el color de cada pixel se calcula una vez que la información de intersección está disponible para todos los

Algoritmo 5 Pseudocódigo del cálculo de la contribución de color de un rayo

```
1: function SHADE(rayo,intersección,material,framebuffer)
2:     Color color
3:     if (existe intersección) then
4:         color ← Color ambiente
5:         if (la intersección no está en la sombra) then
6:             color = color + reflejo difuso
7:             color = color + reflejo especular
8:         end if
9:         color ← color × (1 - material.reflectividad)
10:    else
11:        Actualizar color con el valor correspondiente del cubemap de la escena
12:    end if
13:    Calcular la representación entera de color
14:    Actualizar el framebuffer con la representación entera del color
15: end function
```

rayos en un árbol de rayos. Sin embargo no es posible ni práctico retener en memoria toda la información de intersección de todos los arboles de rayos de un tile. Por lo tanto, esta etapa del *ray tracer* debe actualizar la imagen final a partir de un rayo independientemente de los otros rayos en su mismo árbol. Para que esto sea posible, durante la creación de rayos secundarios se calcula un valor de contribución. Dicha contribución es la medida de cuan importante es el valor de color para un rayo particular en relación a todos los rayos del árbol. Con este parámetro es simple llevar a cabo la actualización de la imagen: simplemente es necesario multiplicar el valor de color obtenido para un rayo por su contribución y sumarlo al valor de color del pixel. Para que este esquema funcione, la imagen debe ser restaurada a un valor nulo antes de empezar a procesar los rayos primarios.

La entrada por cada rayo para esta etapa del *pipeline* es el rayo y su metadata, la información del impacto del rayo, el material de la primitiva interceptada y el cubemap[2] que se debe usar en caso de que el rayo no haya interceptado ninguna primitiva en la escena.

En caso de que haya una intersección se usarán la información del material de la primitiva interceptada y las propiedades de las luces de la escena para determinar el valor de color. La iluminación se basa en el modelo de *phong* (presentada en el capítulo 2) 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 el color del reflejo especular de la luz en el material intersecado.

Por cada píxel, se calcula el color final como la contribución de los rayos primarios, secundarios, y de sombra. Al final del proceso de *tracing* la contribución de color de un rayo R_p será equivalente a la aplicación de la siguiente fórmula recursiva:

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

Donde T_p son las propiedades de la superficie en el punto de intersección, $T_p.A$ es la transparencia de dicha superficie, $T_p.Col$ es el valor de color del material (que puede ser constante o puede ser obtenido de una textura) en el 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.

El pseudocódigo para el cálculo de la contribución de color de un pixel se encuentra listado como algoritmo 5.

[2]Imagen de fondo de la escena.

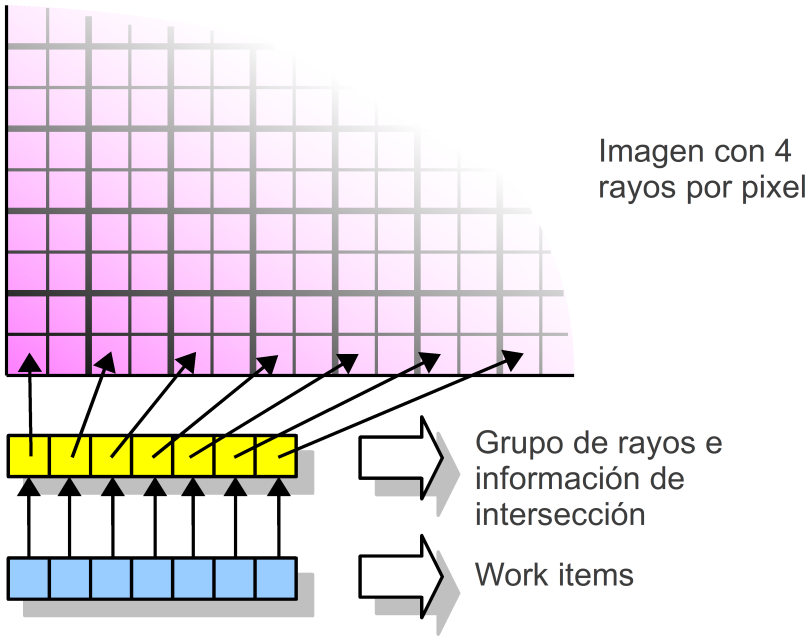


Figura 3.6: Distribución de tareas para *shading* de rayos primarios

Sincronización

Un paquete de rayos puede contener más de un rayo asignado a un mismo píxel. Si esta etapa asignase un work item para cada rayo del paquete, podrían tratar de modificar el valor del mismo píxel de la imagen al mismo tiempo sin ningún tipo de sincronización y generar resultados incorrectos. Una manera sencilla de solucionar este problema es actualizar atómicamente la imagen. Sin embargo, dicha solución es muy ineficiente dado el alto costo de las operaciones atómicas en la mayoría de los dispositivos que soportan *OpenCL*.

Para procesar rayos primarios sin utilizar operaciones atómicas se pueden ordenar los rayos de manera de que en cada tile exista sólo un rayo perteneciente a cada pixel. Por lo tanto es posible asignar un work item diferente para calcular el color que contribuye cada rayo y actualizar el framebuffer sin necesidad de sincronización. Este esquema se ejemplifica en la figura 3.6. Sin embargo, este esquema no permite crear tiles de tamaño mayor a la resolución de la imagen final y genera grupos de rayos menos coherentes que la alternativa de empaquetar los rayos de un mismo pixel juntos para ser procesados. La primer limitación no es grave dado que incluso en bajas resoluciones, la cantidad de pixeles en pantalla es mayor al tamaño ideal para un tile. La segunda limitación es más severa, dado que la coherencia en los datos a procesar impacta sobre la eficiencia de todas las etapas del *ray tracer*. No obstante, el costo de sincronización necesario en caso de permitir más de un mismo rayo por pixel en cada paquete de rayos es mucho mayor a la ganancia en velocidad obtenida de empaquetar juntos dichos rayos, debido al alto costo computacional de las operaciones atómicas.

Cuando esta etapa recibe un paquete de rayos secundarios es inevitable la posible existencia de rayos asignados a un mismo pixel, por lo cual no es posible utilizar la misma estrategia utilizada para los rayos primarios. Sin embargo, es posible que cada work item procese todos los rayos asignados a un mismo pixel de manera secuencial, dado que es conocido que serán adyacentes. Esto es debido a que la etapa de generación de rayos secundarios guardará todos

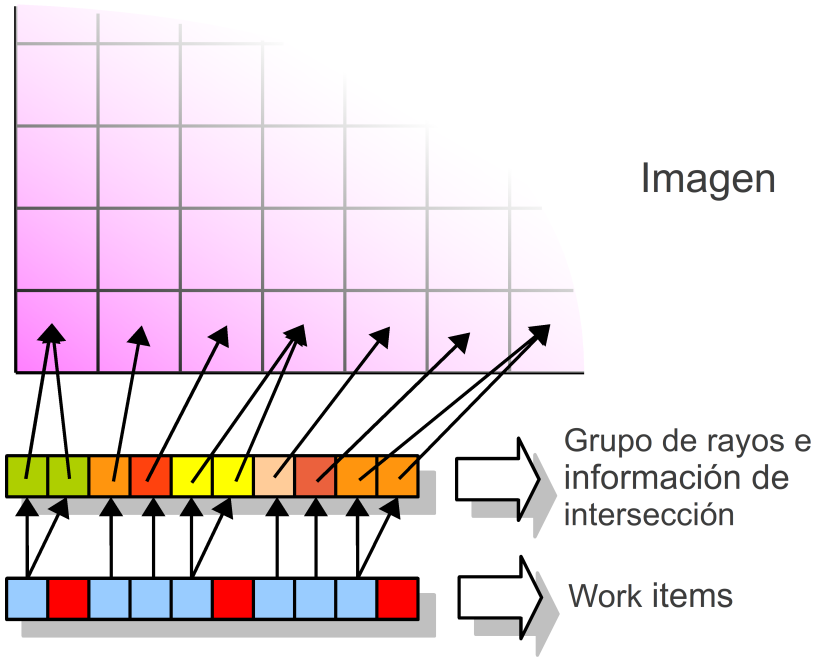


Figura 3.7: Distribución de tareas para *shading* de rayos secundarios

los rayos secundarios pertenecientes a un mismo pixel en espacios consecutivos en memoria. Por lo tanto las actualizaciones a un mismo índice del framebuffer serán hechas por un mismo work item y no será necesaria la sincronización para la escritura. Sin embargo, un problema presente al utilizar este esquema de proceso es que se subutilizan los *elementos de procesamiento* disponibles.

La figura 3.7 muestra el esquema usado para los rayos secundarios. Cada work item se asigna a un resultado de intersección de la etapa de *tracing*, y si este resulta ser el primero de todos los resultados para el pixel al que corresponde entonces procesará todos los resultados hasta encontrar un resultado de un pixel diferente. En caso de que el pixel asignado a un work item no sea el primero, entonces el work item no realizará ningún cómputo.

Capítulo 4

Resultados

Para mostrar el rendimiento de este *pipeline* se evaluarán sus requerimientos de memoria y los tiempos de las diferentes etapas para generar una imagen en distintas escenas y con distintos puntos de vista.

Las escenas utilizadas por la solución están compuestas en su totalidad por triángulos, los cuales son clasificados por un BVH. Cada triángulo tiene un índice a un material que describe el color, el tipo de material (reflectivo u opaco) y un identificador para texturas, todos alojados en una estructura global. Todos estos datos son copiados a memoria de GPU para su ejecución.

El *pipeline* propuesto funciona enteramente en *OpenCL*. Los tiempos que se mencionan no incluyen el tiempo de traspaso de la escena a la memoria del *dispositivo OpenCL*. Tampoco se incluye el tiempo de creación del BVH, que en esta etapa del trabajo todavía se hace en CPU. Por lo tanto no es posible modificar la geometría de la escena de un frame a otro, pero sí el punto de vista. Para mostrar el resultado del renderizado de una escena, se optó por utilizar OpenGL como interfaz y biblioteca gráfica para la solución, la cual está integrada a *OpenCL*.

A continuación se describirán las escenas de prueba usadas para evaluar el desempeño de la solución propuesta. Luego se presentan resultados de uso de memoria para el renderizado de dichas escenas en distintas resoluciones. Finalmente se presentan los tiempos de cómputo obtenidos, discriminados por etapa, para el renderizado de las escenas propuestas, junto con las conclusiones obtenidas.

4.1. Escenas de prueba

Se usaron escenas con diferentes características y diferentes configuraciones del pipeline para permitir evaluar el desempeño de cada etapa del *ray tracer* en particular.

(a) (b) (c)

Figura 4.1: Escena 1

(a) (b) (c)

Figura 4.2: Escena 2

(a) (b) (c)

Figura 4.3: Escena 3

La escena número 1 es un modelo de una mano, de alrededor de 34000 triángulos, que no posee superficies reflectivas ni refractivas, por lo cual sólo es necesario que el pipeline ejecute la instancia de *tracing* primario. Esta escena se corresponde a la figura 4.1.

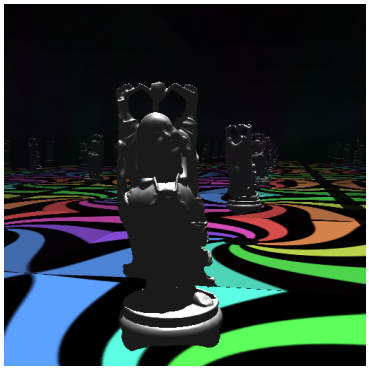
La escena 2 es el modelo de un humano de gran cantidad de triángulos (∼156000). Consiste por completo de superficies no reflectivas o refractivas, y permite juzgar el desempeño del *tracing* primario frente a una escena de alta complejidad. Esta escena puede visualizarse en la figura 4.2.

El modelo de la escena 3 presenta un bote de superficie opaca sobre una superficie de agua refractiva, que permite evaluar la eficacia de la solución presentada en presencia de gran cantidad de rayos secundarios que a su vez pueden impactar con un modelo complejo (el bote). Puede apreciarse en la figura 4.3.

La figura 4.4 muestra la escena 4. En esta escena el modelo sobre la superficie de agua es

(a) (b) (c)

Figura 4.4: Escena 4



(a)

Figura 4.5: Escena 5

de un dragón de alta complejidad geométrica (100000 triángulos) que es a la vez completamente reflectivo, por lo cuál esta es una escena de alta complejidad con gran cantidad de rayos secundarios que son en su mayoría poco coherentes.

La escena número 5 presenta un modelo de una estatua de Buddha dentro de una caja cuyos lados son reflectivos. Se evalúa en esta escena la eficiencia de la solución cuando la cantidad de rayos secundarios se acerca a la máxima cantidad posible, a la vez que se intersecan con un modelo de alta complejidad (100000 triángulos). Esta escena pueden apreciarse en la figura 4.5.

En las escenas 4 y 5 se presentan los casos más complejos, y 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 4, resultando en tiempos de proceso menores para la etapa de *tracing* de rayos secundarios.

Las mallas 3D para las escenas de prueba fueron obtenidos del repositorio de Stanford[1] y del repositorio de animación 3D de la universidad de Utah[2].

El *pipeline* fue probado en en una computadora con una placa gráfica NVIDIA GeForce GTX 590 junto a un procesador Intel i5 2500K.

4.2. Requerimientos de memoria

Para la primer serie pruebas se utilizan tamaños de imágenes de 512×512 y 1024×1024 píxeles y una cantidad de rebotes máxima de 5, para lo cual se generan 262144 y 1048576 rayos primarios respectivamente, 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 144 bytes para almacenar su información y la información de intersección necesaria para calcular su contribución de color. En la tabla 4.1 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. Los valores de la tabla se corresponden con el uso de *tiles* de 81920 rayos. Dicho tamaño surge de comprobar empíricamente que un tile de tamaño 128 veces el de la cantidad de *elementos de procesamiento* provee un equilibrio aceptable entre la eficiencia lograda y el tamaño de las estructuras intermedias para las características de placas gráficas actuales.

La tabla 4.2 muestra los requerimientos en memoria de las escenas de prueba. El tamaño en

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

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 4.1: Estadísticas de memoria

Escena	Triángulos	Vértices	Memoria para modelos geométricos
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 4.2: Cantidad de primitivas en las escenas de prueba

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.

A partir de las pruebas realizadas se desprende que una escena con hasta 10^6 triángulos y en alta resolución requiere menos de 150 MB de memoria, por lo cual cualquier GPU comercial puede alojar esta estructura.

4.3. Rendimiento

Para el segundo conjunto de pruebas se analiza el rendimiento del *ray tracer* en las escenas ya presentadas y generando imágenes de 512×512 y 1024×1024 pixeles de resolución. Se usaron diferentes puntos de vista (mostrados en el Apéndice A) y se presentan los tiempos promedios de cómputo para cada etapa. Es de notar que la variación del ángulo de vista impacta en los tiempos finales del renderizado, especialmente cuando se está observando un objeto con propiedades refractivas. En este caso la cantidad de rayos secundarios puede variar mucho, y esto se ve reflejado en tiempos de proceso más largos.

Las tablas 4.3 y 4.4 muestran el promedio de los tiempos para las etapas del *ray tracer* a partir de la renderización de imágenes desde diferentes puntos de vistas, correspondientes a las figuras 4.1 a 4.5. En particular la tabla 4.3 muestra los tiempos para las escenas 1 y 2, que no presentan materiales reflexivos o refractivos, por lo cual no se generan rayos secundarios. En todas las escenas de la tabla 4.4 se generan rayos secundarios debido a efectos de reflexión o refracción. Es de notar que en las escenas de la tabla 4.3 los tiempos máximos de proceso no se alejan mucho del promedio, pero no ocurre lo mismo para escenas de la tabla 4.4. Por ejemplo, en la escena 2, para resoluciones de 1024×1024, el tiempo máximo de proceso fue sólo 10 % mayor que el promedio, mientras que para la escena 4 la diferencia de tiempo de proceso entre el promedio y el máximo fue de más del 30 %. Esto es debido a que en este último caso la posición de la cámara resultó en la generación de aproximadamente 15 % más rayos secundarios que el promedio. Se incluye una aproximación de la tasa de refresco promedio esperada (FPS[3]) en las diferentes escenas a partir de los tiempos totales de procesamiento.

Para la etapa de *tracing* secundario se experimentó con diferentes tamaños de work groups para evaluar el cambio de rendimiento debido a las razones expresadas en la sección 3.2.2, y empí-

[3]Del inglés "Frames Per Second"

Escena	Resolución	Creación de rayos	*Tracing* primario	Sombras	*Shading*	Total (FPS)
1	512x512	2.1	6.1	2.2	1.2	11.6 (86.2)
1	1024x1024	4.4	11.5	4.3	3.1	23.3 (42.9)
2	512x512	2	12.6	6.1	1.2	21.9 (45.6)
2	1024x1024	4.4	14.8	9.7	3	31.9 (31.3)

Tabla 4.3: Estadísticas de tiempo para las escenas de prueba con materiales opacos (en milisegundos)

Escena	Resolución	Rayos secundarios	Creación de rayos	*Tracing* primario	*Tracing* secundario	Sombras	*Shading*	Total (FPS)
3	512x512	191031	4.5	5	4.4	5.7	3.3	22.9 (43.7)
3	1024x1024	585409	13.6	10.4	13.6	7.3	9.3	54.2 (18.4)
4	512x512	445958	12.2	8.4	48.7	30.6	7	106.9 (9.3)
4	1024x1024	1229718	15.2	19.9	136.9	103.3	15.6	290.9 (3.4)
5	512x512	708760	14.3	8.6	11.5	15.1	7.6	57.1 (17.5)
5	1024x1024	2280820	34.3	18.4	31.4	32.9	19.9	136.9 (7.3)

Tabla 4.4: Estadísticas de tiempo para las escenas con rayos secundarios (en milisegundos)

ricamente se llegó a la conclusión de que procesar rayos secundarios en work groups de tamaño 64 es la mejor opción. Por lo tanto, al procesarse rayos secundarios se define el tamaño de los work groups como el mínimo entre 64 y el tamaño máximo para un work group del dispositivo que los procesa.

La figura 4.6 muestra el porcentaje relativo de tiempo de proceso utilizado por cada etapa del *pipeline* de *ray tracing* para algunas de las configuraciones de cámara para las escenas analizadas.

Finalmente se registró el tiempo de proceso de un recorrido de las escenas con una cámara cuya posición y orientación cambia en cada cuadro. Se obtuvieron los tiempos promedio de proceso para los recorridos a diferente resolución de imagen. La figura 4.7 muestra que los resultados son cercanos a lineales respecto al tamaño de imágenes generadas.

4.3.1. Conclusiones

Como muestran las tablas 4.3 y 4.4, se alcanzan tasas de refresco interactivas para casi todas las escenas, manteniéndose en todos los casos por sobre 3 cuadros por segundo.

Los tiempos de ejecución para las diferentes etapas de la solución son de la proporción esperada para un *ray tracer*, con la etapa de *tracing* ocupando siempre el mayor tiempo de cómputo. En escenas donde se producen una gran cantidad de rayos secundarios, el tiempo de cómputo de *tracing* para esos rayos, debido a que dichos rayos son poco coherentes, es el más elevado, tomando entre 3 y 5 veces más tiempo que la etapa de *tracing* primario.

La etapa de *tracing* de sombras toma entre 70 % y 40 % menos tiempo que la etapas de *tracing* normal. Esto se debe a que el *tracing* de sombra puede terminar prematuramente al encontrar la primer intersección y a 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.

El rendimiento de la etapa de creación de rayos secundarios será afectado directamente por la cantidad de rayos que se generen. En las escenas donde hay una gran cantidad de rebotes, como las escenas 5 y 4, se puede apreciar en la tabla 4.4 que esta etapa será 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

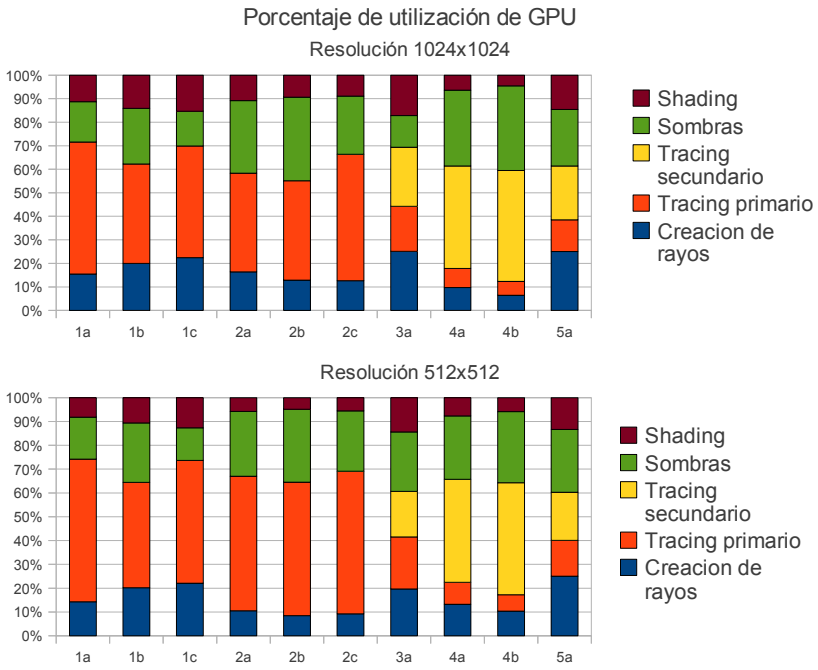


Figura 4.6: Tiempo de cómputo relativo de las etapas de la solución

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 en todos los casos menos del 15 % del tiempo total de cómputo.

La etapa de tracing primario es capaz de procesar entre 18 y 112 millones de rayos primarios por segundo en las escenas de prueba, dependiendo de la complejidad de la escena. Esto es

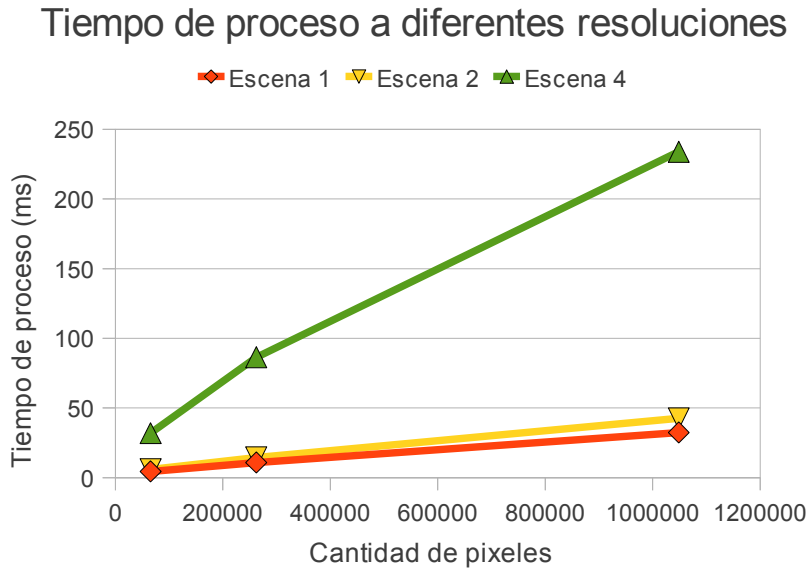


Figura 4.7: Tiempos de cómputo relativos a la resolución de las imágenes obtenidas

comparable con los resultados obtenidos en [1], que utiliza GPUs más antiguos pero consiste de código de ensamblaje de CUDA optimizado manualmente para mejorar su eficiencia. Es imposible optimizar el código de ensamblaje en *OpenCL* debido a que el driver de cada dispositivo *OpenCL* compila los kernels a ejecutar de manera distinta. Asimismo, son similares los resultados presentados en [18] 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 en placas gráficas de eficiencia cercana a la usada para obtener los resultados presentados en este trabajo.

Capítulo 5

Extensiones y aplicación del trabajo

5.1. Aplicación de la arquitectura en escenas dinámicas

El trabajo presentado hasta ahora no contempla la creación de la estructura de aceleración en GPGPU, sino que lo considera parte del preprocesamiento de la escena. Debido a que la estructura de aceleración no se modifica la escena debe ser estática. Si se tratase de agregar nuevas primitivas o modificar las existentes, no coincidirían con la BBox de un nodo, y en consecuencia el algoritmo de *tracing* no encontraría las intersecciones de los rayos con ellas. Por lo tanto, un requisito para poder visualizar escenas dinámicas es que la estructura de aceleración debe ser reconstruida cada frame.

La construcción de la estructura de aceleración enteramente en GPU es una tarea de alta complejidad. Al contrario que la tarea de *ray tracing*, donde encontrar paralelismo para explotar es relativamente fácil debido a que los rayos son hasta cierto punto independientes, durante la creación de un BVH no es evidente la división de tareas posible.

La estructura de aceleración construida en esta etapa del *ray tracer* es llamada a veces *árbol de jerarquía de volúmenes linear*, o LBVH. Esto se debe a que el espacio será dividido sucesivamente por la mitad de uno de sus ejes, sin tener en cuenta la cantidad de primitivas que se encuentran en cada mitad. Esto genera árboles ligeramente menos eficientes a la hora de recorrerlos, pero permite mejorar visiblemente la eficiencia a la hora de creación de la estructura.

La creación del BVH está basada en el trabajo de Pantaleoni y Luebke en [17], que describe un algoritmo eficiente de creación de un LBVH, implementado para el sistema Optix de NVIDIA en CUDA.

La división espacial de las primitivas en la implementación está basada en un función que mapea valores discretos n-dimensionales a una dimensión, tratando de conservar la localidad de los puntos recorridos. Dicha función es llamada *curva-z* o *código de Morton*, y obtiene ese nombre debido a que en dos dimensiones recorre el espacio en un recorrido jerárquico que hace recordar a la forma de una letra Z; este recorrido se ejemplifica en la figura 5.1.

Como muestra la figura 5.1, el código de Morton de un punto específico en el espacio n-dimensional se obtiene intercalando la representación binaria de cada dimensión. Por lo tanto, el dígito más significativo de un código de Morton indica de que lado de una partición en uno de sus ejes se encuentra el punto que representa. El segundo dígito indica de que lado de una partición en otro eje se encuentra el punto dentro de la partición definida por el dígito anterior. Si se sigue este razonamiento, se llega a que cada dígito leído del más significativo al menos significativo restringe el espacio que puede ocupar el punto a un subespacio más pequeño. Esto es equivalente al uso que se obtiene de un BVH, donde se subdivide el espacio en regiones cada vez más pequeñas y con menos primitivas.

El algoritmo de creación del BVH inicialmente generará el nodo raíz que representará toda la

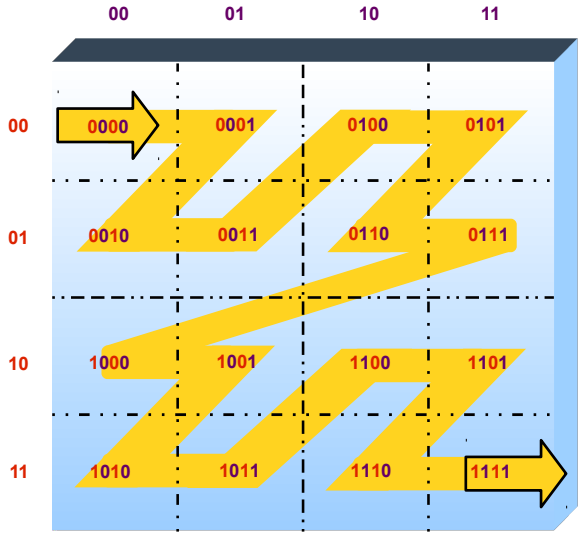


Figura 5.1: Códigos de Morton de 4 bits.

escena. Seguidamente dividirá el espacio en dos subespacios de igual tamaño por uno de sus ejes. Esta división se corresponde al dígito más significante de una curva-z que recorre todo el espacio de la escena. A continuación dividirá el subespacio para los nodos que contengan elementos y creará nuevos nodos internos a partir de los siguientes dígitos en el código de Morton. Una vez se llega al dígito menos significativo se generan nodos hojas con las primitivas que se corresponden al código de Morton del nodo.

5.1.1. Pasos de la creación del BVH

El primer paso para la creación del BVH es el cómputo del código de Morton para cada triángulo en la escena. El espacio total sobre el cual calcular los códigos para cada triángulo es la BBox de la escena entera. Cada eje se divide en 1024 divisiones, por lo cual se necesitan 30 bits para el código. La posición del centro de cada triángulo determina el código que le será asignado.

Una vez calculados los códigos para todas las primitivas en la escena, deben ser ordenados. Ordenar los códigos interpretándolos como números provee efectivamente el recorrido de la curva-z que atraviesa la escena a través de los triángulos. El orden de los códigos de Morton se hace a través de un algoritmo de ordenamiento que utiliza una red bitónica debido a que es posible implementarlo eficientemente en *OpenCL*.

El proceso de creación de los nodos a partir del reparto de las primitivas a los sectores determinados por sus códigos de Morton no es trivial. Por un lado sería simple generar un BVH completo y luego descartar los nodos que no posean primitivas asociadas. Sin embargo, un árbol completo para códigos de 30 bits tendría 2^{31} nodos, lo cual necesitaría más de 2GB de memoria y claramente no es una alternativa viable en GPUs convencionales. Usar menos de 30 bits para los códigos de Morton implicaría agrupar muchas más primitivas juntas, lo cual haría que se pierda eficiencia a la hora de recorrerla. Por lo tanto, es necesario crear el árbol nivel a nivel, teniendo en cuenta los nodos que quedarán vacíos y tratando de explotar el paralelismo disponible en *OpenCL*.

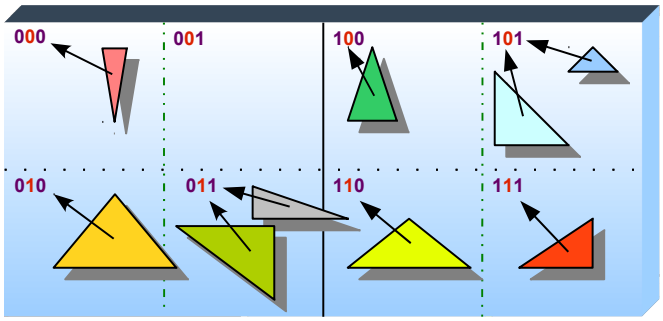
Una vez ordenados los códigos se procesan tres bits del código de Morton de las primitivas por vez, comenzando con los tres más significativos. Inicialmente sólo existe el nodo raíz, que

representa a todos los triángulos de la escena, y el primer paso será encontrar el elemento **Φ** en el cual el bit más significante cambiará de 0 a 1. Debido a que las primitivas están ordenadas de acuerdo a su código de Morton, sabemos que las primitivas anteriores a **Φ** tendrán todas valor 0 en su bit más significativo y las posteriores tendrán valor 1. El índice del elemento **Φ** será guardado en una estructura temporal. Este proceso se repite para cada uno de los dos rangos resultantes, y nuevamente sobre los que resultan en ese paso. Por lo tanto, al finalizar este proceso se tiene los rangos de las particiones de las primitivas a partir de los primeros tres bits de su código de Morton. Esta información se guarda en una estructura intermedia y se utiliza para saber cuantos nodos nuevos es necesario crear, dado que algunos de los rangos pueden ser nulos.

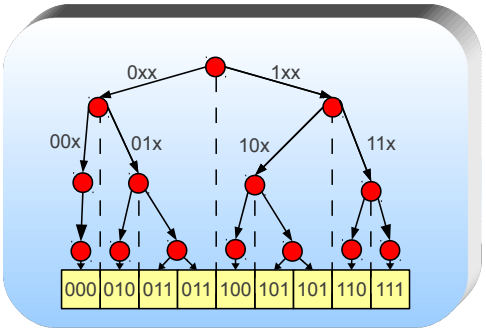
Este proceso se repetirá con los bits restantes de los códigos de Morton hasta agotarlos, momento en el cual se producirán nodos hojas. Debido a que los rangos se procesan por separado para aprovechar el paralelismo de *OpenCL*, se requerirá una etapa de sincronización para calcular el lugar en memoria que ocupará cada nodo.

Esta forma de construcción de los nodos es necesaria para equilibrar los costos de sincronizar la creación de nodos en el BVH y el costo de crear cada nodo en particular: si se procesa un bit a la vez, el trabajo de sincronización sería muy costoso debido a que debería realizarse una vez por bit; si se procesaran muchos bits por paso, las estructuras intermedias construidas para guardar los rangos creados a partir de la similitud de los códigos de Morton serían muy grandes y el costo de recorrerlas aumentaría.

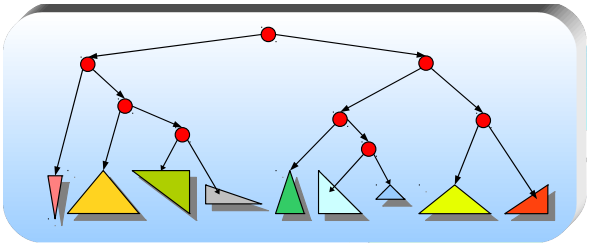
Finalmente, las BBoxes de los nodos son calculados comenzando de las hojas y sucesivamente calculando un nivel superior de la jerarquía hasta llegar a la raíz.



(a) Asignación de códigos de Morton a triángulos en una escena



(b) Árbol inicial formado a partir de los códigos de Morton



(c) Árbol final que se obtiene de la escena inicial

Figura 5.2: Ejemplo de construcción de un BVH con un código de Morton de 3 bits

Escena	Triángulos	Codificación en códigos de Morton	Creación de la estructura del BVH	Cálculo de BBoxes	Tiempo total
1	34270	5.3	2.1	7.1	16.9
2	78029	18.2	9.8	9.3	38.4
4	109248	11.8	8.6	10.2	31.4
5	100020	10.9	9.2	6	27.2

Tabla 5.1: Tiempos de proceso del algoritmo de creación de BVH(en milisegundos)

La figura 5.2 provee un ejemplo de la creación de un bvh mediante el método descripto para una escena simple. La tabla 5.1 muestra los tiempos de proceso en la creación del BVH para algunas de las escenas de prueba ya presentadas, distinguiendo las diferentes etapas del algoritmo. Puede apreciarse que para las escenas del orden de 10^5 triángulos los tiempos de creación permiten obtener una tasa de refresco interactiva.

5.2. Integración con un simulador de fluido superficial

Con el objetivo de probar la solución propuesta con un ejemplo práctico se propuso la integración del *ray tracer* con un simulador de fluido superficial. Esta integración pretende demostrar la capacidad de la solución para producir imágenes fotorrealistas con una tasa de refresco interactiva cuando es incluida en un programa que tiene sus propios tiempos de proceso. Para este propósito se utilizó el simulador de liquido superficial presentado en [4], basado en el método de Lattice-Boltzman.

En la demostración se utiliza una grilla de triángulos plana cuyos vértices serán modificados cada cuadro de acuerdo a los parámetros de la simulación, a fin de crear una secuencia que muestre el movimiento de agua a partir de la caída de gotas de lluvia y la interacción del usuario.

Figura 5.3: Superficie líquida sintetizada

Grilla	Triángulos	FPS promedio
100x100	20000	18
200x200	80000	12
300x300	180000	6

Tabla 5.2: Características de las escenas de prueba de fluido

Los pasos que sigue esta prueba de concepto para generar cada frame es:

1. Tomar el estado de los dispositivos de entrada (mouse).
2. Llamar al simulador de fluido superficial para modificar la posición de los vértices de la malla.
3. Crear el BVH a partir de la malla.
4. Renderizar la escena.

Se realizaron pruebas con grillas de diferentes tamaños para poder constatar el desempeño tanto en tasa de cuadros por segundo como en realismo de las imágenes obtenidas. La tabla 5.2 detalla algunas características de las grillas usadas para simulación, y los resultados obtenidos.

La figura 5.3 muestra una imagen de una superficie líquida obtenida a través de este método.

Capítulo 6

Conclusiones y trabajos futuros

En este trabajo se ha presentado una implementación de un *ray tracer* que utiliza *OpenCL*, el cual aprovecha el poder de cómputo de las placas gráficas hoy disponibles en casi cualquier computadora personal.

El esquema de cómputo paralelo de *OpenCL* obliga a escribir cada parte del *pipeline* de la solución de manera de aprovechar al máximo la gran cantidad de unidades de cómputo que poseen dichas placas gráficas. Debido a la jerarquía de memoria de *OpenCL* y a la división de trabajo en grupos de unidades de cómputo independientes, la arquitectura del *ray tracer* presentado difiere en muchos aspectos de lo convencional.

La solución fue probada en un GPU comercial de alta gama, obteniéndose resultados satisfactorios en lo que respecta a rendimiento y uso de memoria. Se llegó a poder mostrar escenas de gran cantidad de triángulos con un framerate alto.

En el caso de la simulación de fluidos, se mostró que la solución es capaz de ser usada de forma interactiva, respondiendo en tiempo real a la información de dispositivos de entrada.

Algunos de los problemas encontrados durante la creación de este trabajo fueron los esperados para proyectos que realizan cómputos con placas gráficas, tales como altos costos de sincronización de cómputos y acceso a memoria. Estos problemas fueron explicados en el texto y las soluciones encontradas para cada caso ha aumentado la eficiencia de la etapa correspondiente. Sin embargo, quedan lineas de trabajo para seguir mejorando la solución presentada.

Uno de los problemas más grandes en el campo de *ray tracing* en ambientes *multi-core* es la poca coherencia entre rayos secundarios. Este problema fue mitigado parcialmente gracias al algoritmo de ordenación de rayos secundarios mostrado, pero sigue siendo poco eficiente para el caso de creación de rayos secundarios poco coherentes a partir de un mismo rayo. Para ese problema otras posibles soluciones incluyen el reordenamiento global de rayos a partir de algún criterio adecuado, como en [8].

A partir del estado funcional de la solución, existen muchas vias de trabajo futuro a explorar. Uno de los más importantes es mejorar el estado de la solución a fin de soportar modelos y materiales más complejos. Otras lineas de trabajo a considerar incluyen la optimización de los diferentes módulos de la solución, la inclusión de efectos de cámara como *motion blur* o *depth of field*, y el uso de estructuras de aceleración diferentes. Finalmente, el trabajo aquí presentado puede utilizarse como base para implementar otros algoritmos de síntesis de imágenes basados en rayos, como *path tracing* o *photon mapping*.

El código del *ray tracer* presentado en este trabajo puede descargarse libremente en la siguiente dirección: *www.github.com/lscandolo/Tesina-RT*.

Bibliografía

[1] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 145–149, New York, NY, USA, 2009. ACM.

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

[3] Pawel Bak. Real time ray tracing. Master's thesis, IMM, DTU, 2009.

[4] Cristian García Bauza, Gustavo Boroni, Marcelo Vénere, and Alejandro Clausse. Real-time interactive animations of liquid surfaces with lattice-boltzmann engines. *Australian Journal of Basic & Applied Sciences*, 4:3730–3740, 2010.

[5] Nathan A. Carr, Jared Hoberock, Keenan Crane, and John C. Hart. Fast gpu ray tracing of dynamic meshes using geometry images. In *Proceedings of Graphics Interface 2006*, GI '06, pages 203–209, Toronto, Ont., Canada, Canada, 2006. Canadian Information Processing Society.

[6] Martin Christen. *Ray tracing on GPU.* PhD thesis, Chalmers, January 2005.

[7] Colin Fowler and Steven Collins. Implementing the rt^2 real-time ray-tracing system. In *Proceedings of Eurographics Ireland*, pages 1–8, Dec 2007.

[8] Kirill Garanzha and Charles Loop. Fast ray sorting and breadth-first packet traversal for gpu ray tracing. *Computer Graphics Forum*, 29(2):289–298, 2010.

[9] Johannes Gunther, Stefan Popov, Hans-Peter Seidel, and Philipp Slusallek. Realtime ray tracing on gpu with bvh-based packet traversal. *Symposium on Interactive Ray Tracing*, 0:113–118, 2007.

[10] Michal Hapala, Tomas Davidovic, Ingo Wald, Vlastimil Havran, and Philipp Slusallek. Efficient stack-less bvh traversal for ray tracing. In *27th Spring Conference on Computer Graphics (SCCG 2011)*, 2011.

[11] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree gpu raytracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, I3D '07, pages 167–174, New York, NY, USA, 2007. ACM.

[12] Khronos Group. *The OpenCL Specification, version 1.0.29*, December 2008.

[13] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast bvh construction on gpus. *Computer Graphics Forum*, 28(2):375–384, 2009.

[14] H. Ludvigsen and A.C. Elster. Real-time ray tracing using nvidia optix. In *2010 Eurographics*, 2010.

[15] J.David MacDonald and KelloggS. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6:153–166, 1990.

[16] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *journal of graphics, gpu, and game tools*, 2(1):21–28, 1997.

[17] J. Pantaleoni and D. Luebke. Hlbvh: hierarchical lbvh construction for real-time ray tracing of dynamic geometry. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 87–95, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[18] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: a general purpose ray tracing engine. *ACM Trans. Graph.*, 29, 2010.

[19] M. Pharr and G. Humphreys. *Physically Based Rendering: From Theory to Implementation.* Morgan Kaufmann. Elsevier Science, 2010.

[20] S. Popov, J. Gunther, H.-P. Seidel, and P. Slusallek. Experiences with streaming construction of sah kd-trees. *Symposium on Interactive Ray Tracing*, 0:89–94, 2006.

[21] Maxim Shevtsov, Alexei Soupikov, and Alexander Kapustin. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. *Computer Graphics Forum*, 26(3):395–404, 2007.

[22] Ingo Wald, Christiaan Gribble, Solomon Boulos, and Andrew Kensler. Simd ray stream tracing – simd ray traversal with generalized ray packets and on-the-fly re-ordering. Technical report, SCI Institute, 2007.

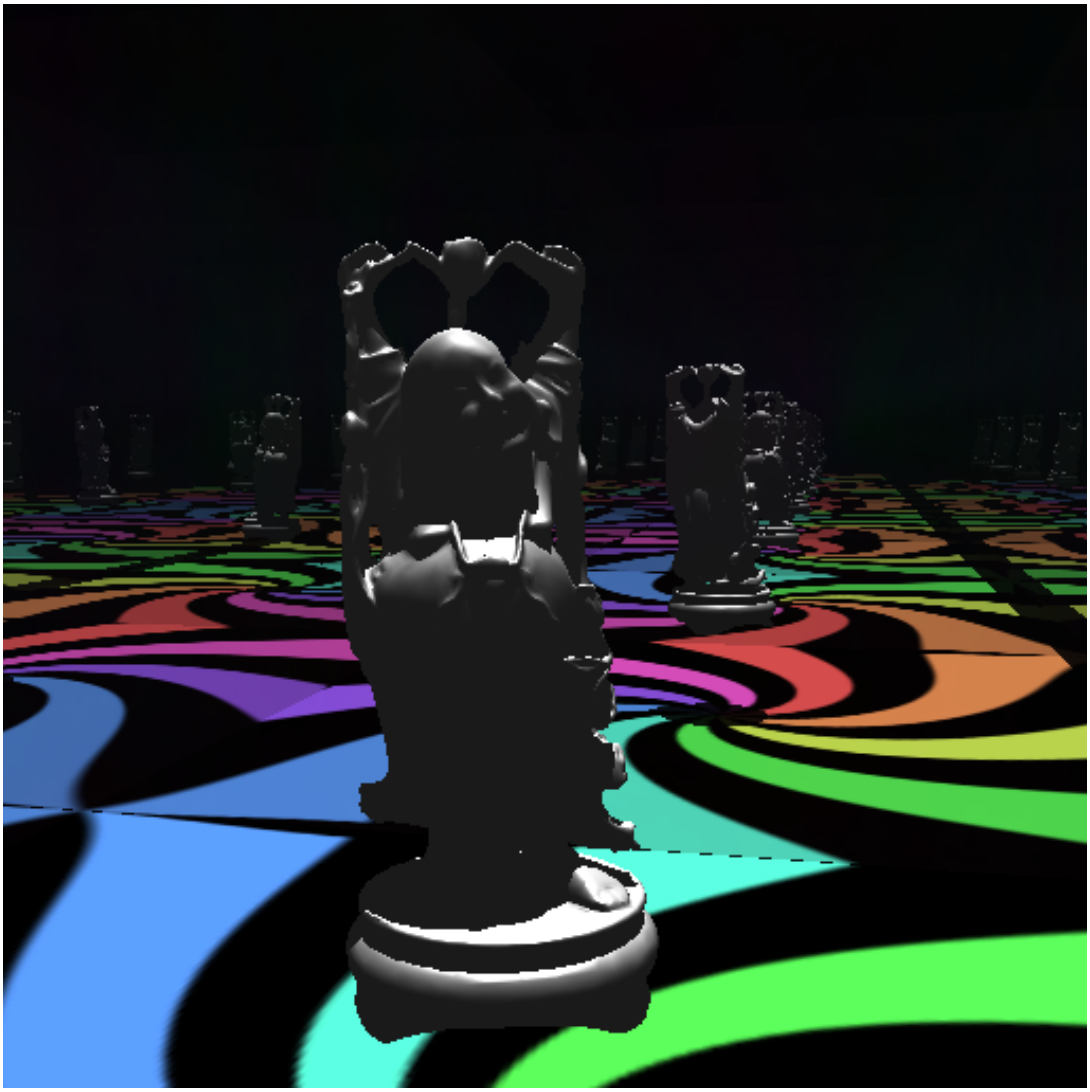
[23] Sven Woop, Gerd Marmitt, and Philipp Slusallek. B-kd trees for hardware accelerated ray tracing of dynamic scenes. In *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 67–77, New York, NY, USA, 2006. ACM.

[24] Carsten Wächter and Alexander Keller. Instant ray tracing: The bounding interval hierarchy. In *IN RENDERING TECHNIQUES 2006 – PROCEEDINGS OF THE 17TH EUROGRAPHICS SYMPOSIUM ON RENDERING*, pages 139–149, 2006.

[25] Martin Zlatuška and Vlastimil Havran. Ray tracing on a gpu with cuda – comparative study of three algorithms. Technical report, Czech Technical University in Prague Faculty of Electrical Engineering, 2009.

Apéndice A

Escenas de prueba



Escena	5
Triángulos	100020
Vértices	50010
Rebotes	5

Escena	1
Triángulos	34270
Vértices	10039
Rebotes	0

Escena	2
Triángulos	156058
Vértices	49964
Rebotes	0

Escena	3
Triángulos	10979
Vértices	6884
Rebotes	5

Escena	4
Triángulos	109248
Vértices	54761
Rebotes	5