Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación

Lecturas en Ciencias de la Computación
ISSN 1316-6239

Rasterización de Escenas 3D con CUDA

Francisco Sans y Rhadamés Carmona

RT 2016-01

Centro de Computación Gráfica
Caracas, noviembre 2016

Rasterización de Escenas 3D con CUDA

Francisco Sans
Computer Graphics Center, Computer Science Department, Faculty of Sciences, Central University of Venezuela Caracas, Venezuela. 1010-A
francisco.sans@ciens.ucv.ve

Rhadamés Carmona
Computer Graphics Center, Computer Science Department, Faculty of Sciences, Central University of Venezuela Caracas, Venezuela. 1010-A
rhadames.carmona@ciens.ucv.ve

***Resumen*—El despliegue de gráficos a través del computador se acelera con el uso de la GPU. Este despliegue se realiza a través de una serie de etapas o *pipeline* que transforman información tridimensional, a una imagen que es desplegada en un monitor. La GPU presenta un implementación por hardware de este *pipeline* gráfico de manera paralela. Inicialmente el *pipeline* poseía una cantidad de etapas fijas, pero ha evolucionado paulatinamente, permitiendo la programación de algunas de sus etapas. Sin embargo, la capacidad de incrementar la programabilidad se ha dificultado debido a que este incremento puede requerir cambios significativos de hardware, lo que puede ser muy complejo. Con el incremento de la capacidad de cómputo de las GPU, especialmente para su uso en el cálculo de propósito general, se ha abierto la posibilidad de recrear el *pipeline* gráfico por software de manera paralela, utilizando APIs de GPGPU como CUDA. En este trabajo se realiza un estudio sobre las alternativas que tiene la GPU para implementar un *pipeline* por software. Se recopilan los estudios recientes y se realizan implementaciones en CUDA utilizando dos vertientes para el despliegue de escenas: rasterización y lanzado de rayos. Aunque las implementaciones propuestas en este trabajo no logran ser competitivas comparadas con el *pipeline* de OpenGL, su estudio abre nuevas posibilidades e investigaciones futuras que puedan ser competitivas.**

***Index Terms*—GPU, OpenGL, CUDA, Rasterization, GPGPU, Ray Casting**

I. Introducción

Las dos vertientes utilizadas para el despliegue de escenas 3D son el uso de la rasterización y el lanzado de rayos (*ray casting*) [1]. Estas dos técnicas producen el mismo resultado para el caso de los rayos primarios [2], por lo que podrían ser utilizados indistintamente para el despliegue de escenas.

La rasterización se encarga de proyectar la geometría (típicamente triángulos) de la escena en el plano imagen y determinar todos los fragmentos asociados a cada triángulo, lo cual se reduce a un problema 2D. Esta técnica típicamente se combina con z-buffer para que sobrevivan los fragmentos visibles por cada píxel. Por otro lado, el trazado de rayos funciona lógicamente en 3D, generando rayos que atraviesan cada píxel de la pantalla y encontrando su intersección más cercana con un triángulo.

Actualmente, todo el proceso de despliegue es realizado en la GPU, ya que está optimizado para la rasterización de geometría de forma paralela con el uso del *pipeline* gráfico. El *pipeline* gráfico corresponde a un conjunto de etapas que permiten transformar una geometría en píxeles que se despliegan en un dispositivo de salida (comúnmente un monitor). Dos de las implementaciones de *pipelines* son OpenGL y Direct3D (OGL y D3D), que en su común denominador se pueden resumir en la Figura 1.

Inicialmente, este *pipeline* consistía en un conjunto de etapas que podían configurarse, pero no programarse, por lo cual se consideraba como un *pipeline* fijo. Con el paso del tiempo se le han añadido etapas que permiten ser programadas por medio de códigos que corren directamente en la tarjeta de video conocidos como *shaders*. Típicamente las etapas programables son el procesador de vértices, utilizando *vertex shaders*, y el procesador de fragmentos, utilizando el *fragment shaders*. En la actualidad existen algunas otras etapas programables como la etapa de geometría o teselado, que aumentan la flexibilidad del despliegue. Sin embargo, el incremento de la programabilidad se ha visto reducido debido a las limitantes de hardware inherentes a la GPU, lo cual implicaría cambios drásticos en su estructura para poder aumentar la programabilidad [3]. Así, etapas como *clipping*, *culling*, o la rasterización, no pueden programarse.

Es posible implementar *pipelines* de despliegue por software, pero debido a la cantidad de cálculo requerido, es ineficiente su implementación en la CPU. Los recientes avances en la GPU han logrado la creación de APIs como CUDA [4] y OpenCL [5], los cuales permiten realizar cálculos de propósito general en la GPU. Con estos APIs, es posible implementar en la actualidad *pipelines* de despliegue por software con lógica paralela en la GPU.

El lanzado de rayos también puede integrarse al *pipeline* gráfico sustituyendo a la etapa de rasterización. En este caso, las etapas de procesamiento de vértices, *clipping*, *culling*, y etapas más modernas como el procesador de geometría y teselado, pueden ser aplicados a cada triángulo en un etapa previa, para posteriormente utilizar el lanzado de rayos sobre todos los triángulos remanentes y determinar el fragmento visible en cada píxel. De esta manera, se produciría un solo fragmento (el de la geometría más cercana) para cada píxel de la imagen final con todas sus propiedades interpoladas, en donde se aplicaría el procesador de fragmentos para realizar el *shading*. Esto contrasta con la rasterización, en donde se podría aplicar *shading* a más de un fragmento por píxel. Realizar el *shading* solo sobre los fragmentos visibles ha demostrado ser eficiente en trabajos recientes de investigación, como en el caso del *Deferred Shading* [6].

En este trabajo se busca implementar un *pipeline* gráfico por software en CUDA, que aproveche el paralelismo de la GPU,

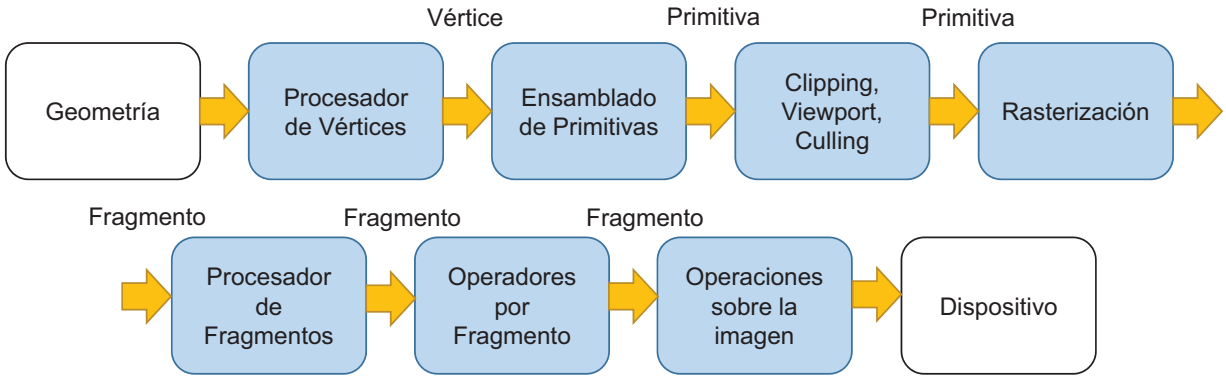


Figura 1. *Pipeline* de visualización 3D.

y que sea altamente programable y flexible a las necesidades. El trabajo está dividido en secciones: la Sección II trata sobre los trabajos relacionados, los cuales sirven como base de esta investigación; la Sección III describe la arquitectura básica de CUDA, la cual va a ser utilizada en las implementaciones; la Sección IV presenta las implementaciones realizadas; la Sección V muestra los resultados obtenidos en las pruebas; y finalmente, la Sección VI presenta las conclusiones y los trabajos futuros.

II. Trabajos Relacionados

En los últimos años se han propuesto diversos trabajos que buscan realizar implementaciones del *pipeline* por software. Por una parte, FreePipe [7] es una implementación paralela donde se asigna un hilo de ejecución para cada triángulo de la geometría, el cual se encargará de rasterizar dicho triángulo. FreePipe es implementado en la GPU utilizando CUDA para aprovechar la naturaleza paralela del *pipeline*. Para escenas con pequeños triángulos, esta implementación se comporta bien, pero para escenas con triángulos de diferentes tamaños posee un mal desempeño, ya que habrá hilos que quedan ociosos esperando a que otros terminen su trabajo.

Posteriormente, Laine y Karras [3] hacen una implementación del *pipeline* de OGL/D3D en CUDA, en donde el trabajo paralelo es asignado al dividir la pantalla en mosaicos (*bins*), buscando mayor equidad de trabajo en los hilos. Inicialmente, la pantalla es dividida en mosaicos de gran tamaño, y los triángulos son asignados, acorde al mosaico que intersecten, a listas de triángulos correspondientes a cada mosaico. Este proceso se repite en una jerarquía de mosaicos hasta obtener un área lo suficientemente pequeña para realizar el proceso de rasterización de manera eficiente. Así, el proceso de rasterización final se realiza en áreas muy pequeñas, lo que evita el cálculo redundante y genera mayor paralelismo. Sin embargo, este software no fue diseñado para permitir la programación de las diferentes etapas del *pipeline* y su desempeño no logra alcanzar la eficiencia de los rasterizadores por hardware. Basados en esta implementación del *pipeline* por software, Chun y Beak [8] han propuesto un prototipo de librería compatible con el API de OpenGL con el proceso de rasterización utilizando CUDA, aunque todavía no hay una implementación disponible.

Ampliando estas ideas, Patney et al. [9] desarrollaron un Framework en CUDA llamado Piko, que permite la división de etapas para la creación de un *pipeline* programable. Utilizan diferentes recipientes para explotar la idea de la localidad espacial y definir etapas de *pipeline*, permitiendo una alta programabilidad. Además, comparan su trabajo con el de Laine y Karras [3], siendo Piko más lento. Esto es debido a que este Framework se centra más en la programabilidad que en la eficiencia.

Por otro lado, Davidovic et al. [2] proponen la integración de la rasterización 2D y el lanzado de rayos para el desarrollo de un método que llaman rasterización 3D, con el cual toman lo mejor de ambos enfoques para el despliegue de geometría. Su idea fue permitir que ambos enfoques pudieran implementarse al mismo tiempo, al realizar primero recorridos de la escena almacenada en una *BVH* (*Bounding Volume Hierarchy*) con *frustums* jerárquicos, descartando geometría a cada paso, y realizar la rasterización cuando el *frustum* actual solo ocupa un pequeño espacio del *viewport*. Los autores realizaron su implementación en CPU, determinando que para algunas de sus escenas de prueba su algoritmo ofrece mejores tiempos a la rasterización 2D implementada en CPU. También realizaron implementaciones de ambas vertientes en GPU (rasterización 2D y rasterización 3D), obteniendo resultados similares entre ambas implementaciones en GPU. Esto indica que su propuesta de rasterización 3D tiene un desempeño similar a la rasterización 2D por software.

En la bibliografía revisada no hay trabajos que utilicen el lanzado de rayos dentro de un *pipeline* programable para la visualización de escenas; aunque si hay diversos trabajos que han ido mejorando tanto el trazado de rayos como el lanzado de rayos [10] [11] [12] [13] [14] [15].

En este trabajo, se estudia la implementación de dos enfoques de *pipeline* por software. El primero de ellos utiliza la rasterización, y el segundo el lanzado de rayos. Ambos enfoques también fueron implementados en la GPU utilizando CUDA.

Luego se discute sus bondades en cuanto a la programabilidad, y se compara el desempeño contra OpenGL, para determinar la factibilidad de uso.

III. CUDA

CUDA (*Compute Unified Device Architecture*), es una plataforma de computación paralela creada por NVidia, que permite el cálculo de propósito general en la GPU. Solo está disponible para las tarjetas gráficas NVidia superior o igual a la serie GeForce 8 [4].

Se constituye por un *device* (GPU), el cual se encargará de realizar los cálculos que le asignó el *host* (CPU) a través de los *kernels*. Un kernel es una función compilada en el GPU. Cuando un *kernel* se ejecuta, se instancian varios hilos (*threads*), todos realizando las instrucciones indicadas en el *kernel*, en el formato de instrucción SIMT (*Single Instruction Multiple Thread*), en donde la capacidad multihilo es simulada por un conjunto de procesadores SIMD (*Single Instruction Mutiple Data*). Los hilos son agrupados en bloques (*blocks*) y los bloques en mallas (*grids*), como se puede observar en la Figura 2. Tanto los bloques como las mallas pueden ser de una, dos o tres dimensiones, permitiendo una mejor organización lógica.

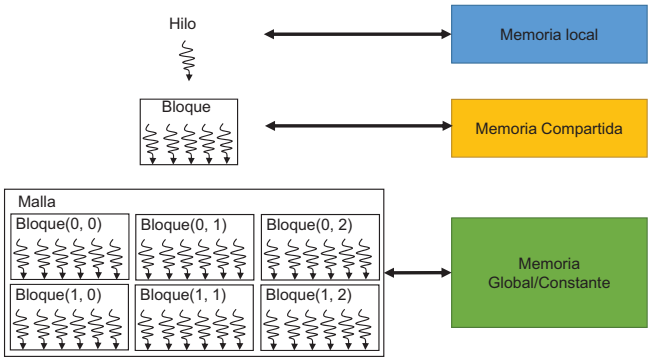


Figura 2. Organización de los hilos y jerarquía de memoria de la GPU en CUDA, donde puede observarse la memoria global, constante, compartida y local.

También hay que agregar que CUDA posee un conjunto de instrucciones que permiten la sincronización entre programas y ejecuciones de los hilos. Además, los hilos dentro de los bloques son agrupados en conjuntos de 32 hilos, llamados *warps*, los cuales ejecutarán la misma instrucción al mismo tiempo. Esto quiere decir, que todos los hilos de un *warp* están sincronizados, lo que permite ciertas optimizaciones a la hora de la programación. Sin embargo, se deben evitar divergencias, ya que disminuyen el desempeño de la aplicación. La divergencia se origina cuando dos hilos ejecutan dos conjuntos de instrucciones diferentes; debido, por ejemplo, a la ejecución de un condicional. Cuando ocurren divergencia entre hilos de un mismo *warp*, algunos hilos deben esperar a que el resto de los hilos del mismo *warp* ejecuten su conjunto de instrucciones para luego ser ejecutados, disminuyendo el paralelismo.

El espacio de memoria del *host* y el *device* está completamente separado. El *device* posee una jerarquía de memoria (ver Figura 2), donde se encuentran los siguientes tipos:

- Memoria global (*global memory*): permite el acceso de lectura/escritura a todos los hilos de todos los bloques. Los hilos pueden leer y escribir cualquier elemento de un objeto de memoria de este tipo. El *host*, podrá escribir y leer en esta memoria para poder copiar la información al GPU.
- Memoria constante (*constant memory*): es una memoria global que se mantiene constante durante la ejecución de un *kernel*. El *host* tiene acceso a ella, y es el único que puede asignar e inicializar objetos en esta memoria.
- Memoria compartida (*shared memory*): la memoria compartida, es local a un bloque, y solo puede ser utilizada por los hilos pertenecientes a ese bloque. Su acceso es extremadamente más rápido que el uso de la memoria global y constante.
- Memoria local (*local memory*): es una memoria privada para un hilo, teniendo un acceso más rápido que la memoria compartida.

Adicionalmente, CUDA tiene soporte de dos tipos de objetos de memoria: memoria de superficie (*surface memory*) y memoria de textura (*texture memory*). Una memoria de superficie almacena una colección unidimensional de elementos, mientras que una memoria de textura es usada para almacenar texturas, *frame buffers* o imágenes de dos o tres dimensiones.

Aunque el objetivo principal de CUDA no es el despliegue gráfico, puede utilizarse para implementar el *pipeline* de visualización. Esta implementación no es trivial, debido a que hay ciertas optimizaciones por hardware que posee el *pipeline* gráfico, que pueden ser difíciles de emular eficientemente por *software*.

IV. Consideraciones de Diseño

En este trabajo se realizaron varias implementaciones por software de un *pipeline* gráfico utilizando CUDA. Inicialmente, la información de la geometría debe ser transformada para poder ser accedida desde CUDA. Es posible codificar la información en arreglos que pueden ser llevados a la memoria global de la GPU. Un *kernel* paralelo de CUDA se encargará de calcular el color final de cada uno de los píxeles de la imagen resultado, la cual estará representada como un arreglo 2D en la memoria de la GPU. El método para calcular el color final puede ser la rasterización o el trazado de rayos, y dependiendo del enfoque utilizado, los algoritmos y las estructuras de datos pueden variar. Finalmente, el arreglo con el resultado final es desplegado para mostrar la imagen por pantalla.

En esta sección se resumirán dos enfoques que se utilizaron para implementar el visualizador de escenas. Para el caso del lanzado de rayo se incluye una versión optimizada con *octree* para reducir el tiempo de respuesta.

IV-A. Rasterización de triángulos

La primera versión de rasterización de escenas 3D realizada en este trabajo consiste en realizar un *pipeline* por software

para el rellenado de triángulos. Todas las etapas del *pipeline* (ver Figura 1) fueron implementadas de manera paralela, donde a cada hilo se le asigna un triángulo para procesar. Cada hilo se encargará de realizar todas las etapas del *pipeline* para el despliegue de dicho triángulo: transformación sobre los vértices, *clipping*, *culling*, rasterización y sombreado (ver Figura 3).

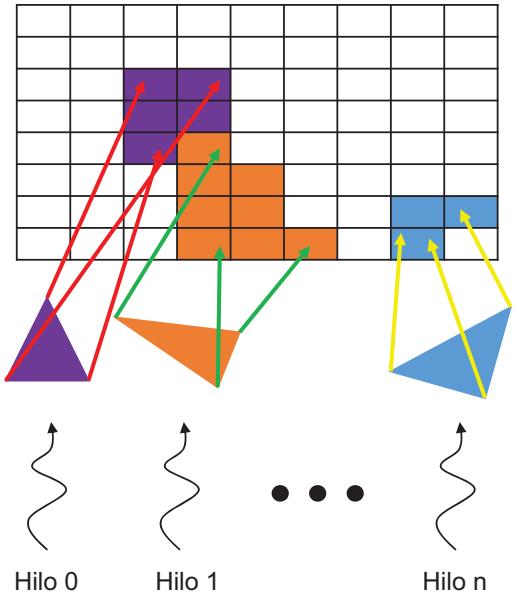


Figura 3. Rasterización paralela utilizando CUDA. Cada hilo procesa un triángulo de la geometría, llevándolo por todas las etapas del *pipeline* y calculando su contribución para cada uno de los píxeles finales.

La información de la geometría es llevada a la memoria global de la GPU como un arreglo de triángulos, que contiene índices a un arreglo de vértices, el cual tiene la información tridimensional de cada punto. Primeramente cada hilo accede a su correspondiente triángulo obteniendo las coordenadas de los vértices en espacio objeto. Con las matrices de transformaciones modelo, vista y de proyección, cada vértice es llevado al espacio de *clipping*. En este espacio, se realiza el descarte de caras traseras (*back face culling*) y posteriormente se realiza el *clipping*, para solo quedarse con las partes del triángulo que intersectan con el área de despliegue. Este proceso puede generar de 0 a 6 triángulos a partir del triángulo original, debido al proceso de re-triangularización después del *clipping*. Cada uno de estos nuevos triángulos posteriormente son rasterizados, mediante el algoritmo de *Scanline*, y para cada uno de los fragmentos generados, se calcula su color y se verifica si es visible utilizando el *z-buffer*. Si es visible, el color es escrito en el *framebuffer*. Se implementaron clases y funciones en CUDA para el manejo del *z-buffer* y el *framebuffer*.

La programabilidad en este caso se reduce a modificar las funciones que implementan cada una de las etapas del *pipeline*.

Uno de los problemas principales que muestra este enfoque es que el trabajo es desigual para cada uno de los hilos. La etapa que se ve más impactada por ello es la rasterización. Hilos con triángulos que abarcan mayor área en la pantalla tardarán más tiempo en el proceso de rasterización y sombreado que triángulos con menor área, especialmente si en la etapa de *clipping* hay mucha subdivisión. Debido a la organización de los hilos en bloques, un solo hilo con mucho trabajo puede retrasar la ejecución de un *kernel*, debido a que todo el bloque de hilos debe terminar para poder empezar con otro bloque.

Además, debido a la ejecución paralela, muchos hilos pueden tratar de escribir al mismo tiempo sobre el mismo píxel del *framebuffer*, lo que puede resultar en incongruencias al momento del despliegue, ya que el *z-buffer* podría no funcionar correctamente. Para evitar las incongruencias en la imagen final y las incongruencias con el *z-buffer*, se implementó un sistema de exclusión mutua para el uso del *z-buffer*, tratando de permitir el uso atómico del mismo. Sin embargo, debido a la organización de los hilos en unidades de ejecución simultáneas (*warps*), se originan abrazos mortales debido a la divergencia en la ejecución de los *warps* [16], por el uso atómico del *z-buffer*, haciendo inviable esta solución.

IV-B. Lanzado de Rayos Paralelo

Debido a los problemas expuestos anteriormente, se decidió realizar otra versión del visualizador utilizando otra técnica de visualización: el lanzado de rayos. En este caso, el paralelismo se realiza por píxel en vez de por triángulo, donde cada hilo tendrá asignado un píxel final de la pantalla, por el cual se lanzará un rayo hacia la escena. Este rayo debe recorrer la escena y conocer con cuales triángulos intersecta. Inicialmente se realizó un enfoque de fuerza bruta en donde un hilo se encargará de determinar el triángulo más cercano a la pantalla, calculando la intersección del rayo con todos los triángulos en la escena (ver Figura 4).

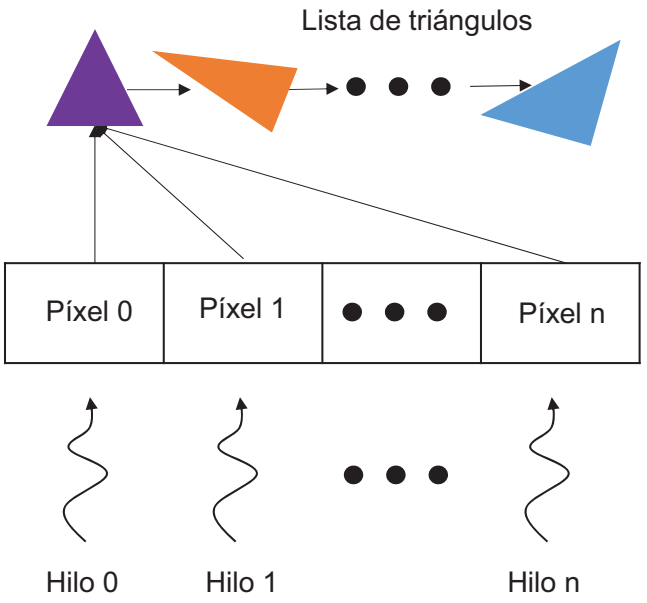


Figura 4. Lanzado de rayos paralelo. Cada hilo se encarga de determinar cuál es el triángulo visible, y calcula su contribución en el color final.

Igual que para la versión anterior, los triángulos están organizados en un arreglo, donde se tienen índices a otro

arreglo que contiene la información de los vértices. Ambos arreglos se encuentran en memoria global de la GPU. Como cada hilo se encarga del coloreado de un píxel particular, no es necesario tener un método de acceso excluyente al *framebuffer*, y tampoco es necesario utilizar un *z-buffer*. Sin embargo, para escenas con muchos triángulos, cada hilo debe calcular la intersección con todos los triángulos, lo cual puede llegar a ser pesado y no dar resultados en tiempo real. Por ello, se realizó otra versión donde se realiza un recorrido de la escena utilizando un *octree*, el cual acelera el cálculo de interesección del rayo con la escena.

IV-C. Optimización del Lanzado de Rayos con un Octree

Una manera de optimizar el cálculo de la intersección de los rayos de visualización con la geometría es utilizando alguna estructura jerárquica de subdivisión espacial como el *octree*. Sin embargo, uno de los problema de utilizar un *octree* es la necesidad de recursión para el recorrido del árbol, la cual no es permitida en algunas versiones de CUDA. Por ello, se hace necesario adaptar esta estructura para que pueda ser recorrida de manera iterativa en la GPU. Diversos autores han hecho propuestas para atacar este problema [15][17].

En este trabajo se optó por transformar el *octree* a un arreglo lineal como puede observarse en la Figura 5. Para obtener esta representación lineal se utilizó el Algoritmo 1. Primero, se crea un arreglo donde cada posición contiene una celda que puede tener información de una hoja, un nodo interno o de un triángulo. Para cada nodo que se va a procesar, la función tiene un índice a la posición donde debe almacenarse el nodo actual en el arreglo. El llenado del arreglo se realiza en anchura. Al procesar un nodo interno, se almacenan sus hijos "no vacíos" de modo contiguo al final del arreglo. A su vez, estos hijos pueden ser nodos internos, u hojas con primitivas. Así, en cada nodo se almacena la cantidad de hijos o primitivas que tiene en la variable `numChilds`, y se coloca el índice en el arreglo del primer hijo en la variable `firstChild`. Finalmente, cada primitiva es almacenada como un índice (dado por `firstChild`) a un arreglo donde están los triángulos de la geometría. Así, toda la información del árbol quedará almacenada en un solo arreglo lineal.

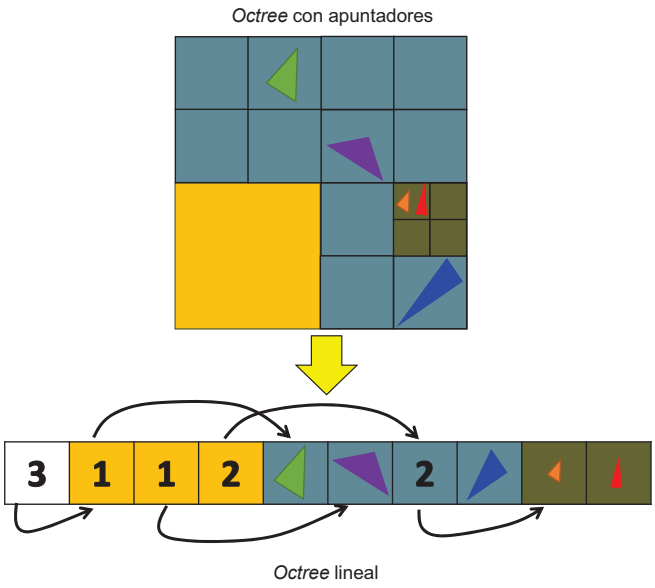


Figura 5. Transformación de un *octree*, a un arreglo continuo. Cada nodo almacenará la posición de su primer hijo, y el número de hijos, y los hijos inmediatos se almacenarán contiguos en memoria. En el caso de las hojas, se almacenarán identificadores de los triángulos. Solo se almacenan las hojas que posean primitivas (no vacías).

```
enum CellType {LEAF, TRIANGLE, INTERNAL};

struct Cell{
  CellType type;
  BoundingBox boundingBox;
  int firstChild;
  int numChilds;
};

void Octree::toLinearChild(std::vector<Cell> &linearOc, ↩
    unsigned int pos){
  linearOc[pos].boundingBox = this->boundingBox;

  //
  linearOc[pos].firstChild = linearOctree.size();

  //si este nodo del octree es hoja
  if(this->Hoja){
    //colocar el tipo de nodo en el arreglo es hoja
    linearOc[pos].type = LEAF;
    //colocar que la cantidad de hijos es igual a la ↩
        cantidad de primitivas
    linearOc[pos].numChilds = this->primitivas.size();

    //Guardar cada una de sus primitivas en el arreglo ↩
        lineal
    for(unsigned int i = 0; i< primitivas.size();++i){
      Cell tri;
      //indicar que es una primitiva
      tri.type = TRIANGLE;
      //indicar el índice de la primitiva
      tri.firstChild = primitivas[i];
      //colocarle su caja envolvente
      tri.boundingBox = this->boundingBox;
      linearOc.push_back(tri);
    }
  }else{
    //si no es hoja, es un nodo interno del árbol
    linearOc[pos].type = INTERNAL;

    int numhijos = 0;

    //contar en número de hijos no nulos y se colocan en ↩
        el arreglo
    for(unsigned int i = 0; i < 8;++i){
      if(!hijos[i]->Hoja || (hijos[i]->Hoja && hijos[i]->↩
          primitivas.size() > 0)){
        Cell dommy;
        ++numhijos;
        //se reserva su espacio en el arreglo
        linearOc.push_back(dommy);
      }
    }

    //se coloca en la posición actual el numero de hijos
    linearOc[pos].numChilds = numhijos;

    int count = 0;

    //realizar el mismo procedimiento recursivamente para↩
        todos los hijos
    for(unsigned int i = 0; i < 8;++i){
      if(!hijos[i]->Hoja || (hijos[i]->Hoja && hijos[i]->↩
          primitivas.size() > 0)){
        hijos[i]->toLinearChild(linearOc, linearOc[pos].↩
```

```
            firstChild + count);
      }
    }
  }
}
```

Algoritmo 1. Algoritmo para la construcción lineal del *octree*.

Para recorrer el *octree* en la GPU se utilizó la lógica mostrada en el Algoritmo 2. Esta función calcula la intersección entre un rayo y todos los triángulos de la escena utilizando el *octree*. Para ello, se utilizó una pila implementada con un arreglo de tipo `stackNode`, el cual va a tener un índice (`index`) a la posición del *octree* que va a ser utilizada, y va a almacenar un índice (`actualChild`) al hijo del nodo que se está procesando actualmente. Debido a la poca flexibilidad con la memoria dinámica que puede ser utilizada en un hilo, esta pila tiene un tamaño definido por `MAXDEPTH`.

```
struct stackNode{
  int index, actualChild;
};
__device__
int octreeInter(    const float4 & O,  //Ray origin
                    const float4 & D,  //Ray direction
                    const Cell * octree,
                    const uint3 * const id,
                    const float4 * const pos,
                    float *dist, float &u, float &v){
  int actual = 0, init, num, idInter = -1, child_index;
  uint3 idtri;
  stackNode Stack[MAXDEPTH], *Node;
  float4 V0, V1, V2;
  BoundingBox boundingBox;
  float t, v1, u1;
  Stack[actual].index = 0;
  Stack[actual].actualChild = 0;
  actual++;
  while(actual > 0){
    Node = &Stack[actual - 1];
    if(Node->actualChild >= octree[Node->index].numChilds↩
        )
      --actual;
    else if(octree[Node->index].type == LEAF){
      init = octree[Node->index].firstChild;
      num = octree[Node->index].numChilds;
      for(int i = 0; i < num; ++i){
        idtri = id[octree[init + i].firstChild];
        V0 = pos[idtri.x];    V1 = pos[idtri.y];
        V2 = pos[idtri.z];    t = FLT_MAX;
        if(ray_triangle(V0, V1, V2, O, D, &t, u1, v1) &&
            t < *dist){
          *dist = t;
          u = u1; v = v1;
          idInter = octree[init + i].firstChild;
        }
      }
      --actual;
    }else if(octree[Node->index].type == INTERNAL){
      child_index = octree[Node->index].firstChild +
              Node->actualChild;
      boundingBox = octree[child_index].boundingBox;
      if(ray_box(O, D, &t, boundingBox) && t < *dist){
        //Insert the new child in the stack
        Stack[actual].index = child_index;
        Stack[actual].actualChild = 0;
        ++actual;
      }
      ++Node->actualChild;
    }
  }
  return idInter;
}
```

Algoritmo 2. Recorrido del *octree* de manera paralela con CUDA.

El algoritmo empieza insertando el nodo raíz del árbol en la pila. Mientras la pila no esté vacía (`actual` $>$ 0) todavía quedan nodos del árbol por recorrer. Se toma el nodo del tope de la pila, y procesamos cada uno de sus hijos. Cada nodo hijo puede ser un nodo interno o una hoja. Si el nodo hijo (`actualChild`) es un nodo interno, se calcula la intersección del rayo con la caja envolvente. En caso de intersección, se inserta en el tope de la pila para continuar el algoritmo con dicho nodo hijo. Si no hay intersección, se ignora el nodo hijo y se procesa el próximo, hasta procesar todos los hijos del nodo, en donde de desapila dicho nodo, dando paso al nodo previo de la pila. Si el nodo hijo es una hoja, se accede a la información de cada uno de los triángulos, y se calcula su intersección con el rayo. Se utilizó el algoritmo de intersección rayo-caja mostrado en [18], el cual está optimizado para evitar divergencias. Una vez procesado todos los triángulos de la hoja, el nodo actual es desapilado.

V. Pruebas y Resultados

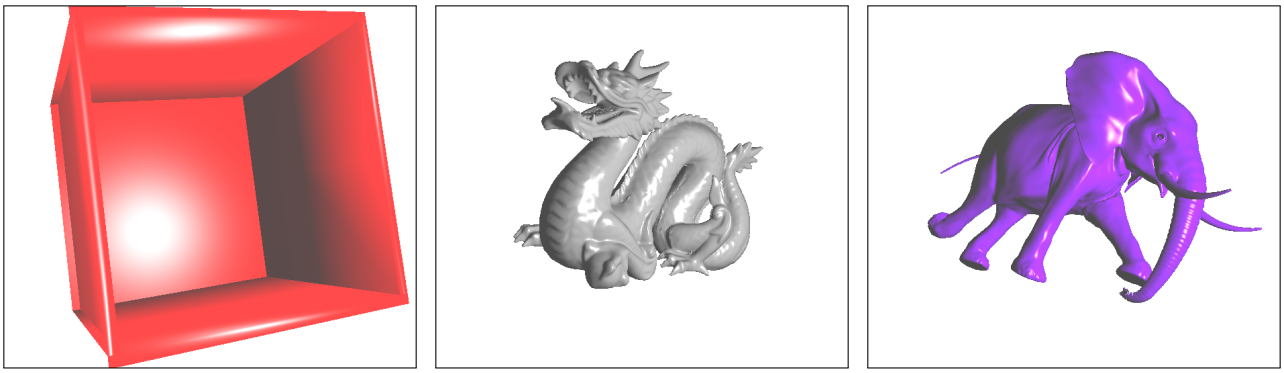
Se midió el desempeño del visualizador y comparó con el desempeño del *pipeline* gráfico por defecto utilizando OpenGL, para verificar si realmente un *pipeline* por software puede ser competitivo con un *pipeline* por hardware. Las pruebas se realizaron en un PC convencional con un procesador Intel(R) Core(TM) i7-3770 de 3.40GHz, 8 GB de RAM, Windows 7 64 bits, y una Nvidia GeForce GTX 660. Esa tarjeta de video soporta OpenGL 4.5, OpenCL 1.2, y CUDA con capacidad de cómputo 3.0. Los objetos utilizados para realizar las pruebas pueden observarse en la Figura 6, y el número de triángulos para cada uno de ellos se encuentran reflejados en la Tabla I.

Tabla I
Número de triángulos de los objetos a utilizar en las pruebas.

Objeto	Número de triángulos
Caja de Cornel	32
Dragón de Stanford	100000
Elefante	84698

Los tiempos de respuesta del visualizador fueron probados utilizando el rasterizador por software en la CPU, el rasterizador por software en la GPU con CUDA, el rasterizador por hardware de OpenGL, el lanzador de rayos con *octree* en la CPU y el lanzador de rayos en la GPU con CUDA. La idea de las pruebas es comparar el incremento de rendimiento que proporciona la implementación de estas técnicas utilizando GPGPU, y comparar su desempeño con la rasterización por hardware de OpenGL. En el caso de la rasterización por software en el GPU, el problema de la exclusión mutua para la escritura en el *framebuffer* no pudo solucionarse, por lo que esta implementación podría generar ciertas incongruencias entre cuadros. Los resultados obtenidos se muestran en la Tabla II.

Como se puede observar, el rasterizador por software en la GPU presenta peor comportamiento que el rasterizador por software en la CPU, a pesar del alto paralelismo. La versión en CPU es entre 4 y 23 veces más rápido con respecto a la versión en GPU, debido a la alta divergencia entre cada hilo en la versión GPU.



(a) (b) (c)

Figura 6. Objetos utilizados para realizar las pruebas con : (a) caja de Cornel, (b) dragón de Standfor, y (c) elefante.

Tabla II
COMPARACIÓN DE RESULTADOS EN MILISEGUNDOS Y CUADROS POR SEGUNDO (FPS)

	Caja de Cornel		**Dragón de Stanford**		**Elefante**	
	Milisegundos	FPS	Milisegundos	FPS	Milisegundos	FPS
Rasterización CPU	11,30	88,50	80,50	12,42	101,30	9,87
Rasterización GPU	262,50	3,81	354,00	2,82	458,60	2,18
Rasterización OpenGL	0,39	2564,10	1,04	961,54	0,79	1265,82
Lanzado de rayos CPU	254,00	3,94	270,00	3,70	332,00	3,01
Lanzado de rayos GPU	5,65	176,99	8,80	113,64	18,10	55,25

Tabla III
COMPARACIÓN DE RESULTADOS EN MILISEGUNDOS Y CUADROS POR SEGUNDO (FPS), CON UN ACERCAMIENTO A LOS OBJETOS DEL 50 %

	Caja de Cornel		**Dragón de Stanford**		**Elefante**	
	Milisegundos	FPS	Milisegundos	FPS	Milisegundos	FPS
Rasterización CPU	29,40	34,01	87,70	11,40	126,50	7,91
Rasterización GPU	900,10	1,11	438,46	2,28	648,56	1,54
Rasterización OpenGL	0,46	2173,91	1,35	740,74	0,83	1204,82
Lanzado de rayos CPU	334,00	2,99	348,00	2,87	660,00	1,52
Lanzado de rayos GPU	7,50	133,33	18,24	54,82	47,40	21,10

Por otro lado, el lanzado de rayos implementado sobre la GPU presenta una mejora de tiempo en milisegundos de entre 18 y 44 veces en comparación con su versión en CPU, lo cual es una mejora significativa. Esto se debe a que el lanzado de rayos es altamente paralelo, lo cual puede ser aprovechado por la GPU, y los cálculos que deben realizarse no originan tanta divergencia.

Comparando las versiones basadas en rasterización por GPU y lanzado de rayos por GPU, podemos observar que el lanzado de rayos es entre 25 y 46 veces más rápido. Sin embargo, ambas implementaciones se encuentran lejos del rendimiento que presenta el rasterizador por hardware de OpenGL, siendo el rasterizador por software en GPU entre 28 y 128 veces más lento, y el lanzado de rayos en GPU entre 8 y 22 veces más lento.

Posteriormente, se realizó una prueba adicional con un 50 % de acercamiento a la pantalla, cuyos resultados se pueden observar en la Tabla III. Debido a que al aumentar el acercamiento más píxeles de la imagen final son cubiertos por geometría, el trabajo del rasterizador y del lanzador de rayos es mayor. El comportamiento de esta tabla es similar al comportamiento de la Tabla II. Tiene especial mención el caso de la caja de Cornel, que a pesar de ser el objeto con menos geometría, es el que mayor tiempo requiere para realizar su rasterización por software en el GPU. Esto es debido a que se tienen pocos hilos realizando mucho trabajo, ya que un solo triángulo abarcará casi toda la pantalla, por lo que el paralelismo no es explotado correctamente.

VI. CONCLUSIONES Y TRABAJOS FUTUROS

En este trabajo se realizó un estudio del estado del arte sobre la implementación de un visualizador de escenas por software, y se hicieron algunas implementaciones a modo de prueba. Primero, se implementó un visualizador utilizando la rasterización y paralelizando cada una de las etapas del *pipeline* OGL/D3D. La versión paralela que utiliza la rasterización, no funcionó correctamente debido a problemas de sincronización a la hora de escribir en el *framebuffer* final. Por ello, se implementó una versión utilizando trazado de rayos en la GPU, ya que no hay necesidad de implementar sincronización para la escritura del *framebuffer*.

Se realizaron pruebas sobre la versión del visualizador en GPU y CPU, utilizando la rasterización y el lanzado de rayos optimizado con el uso de un *octree*, comparando los resultados obtenidos con la rasterización por hardware de OpenGL. Como se pudo observar en los resultados, el uso de la GPU mejora considerablemente el rendimiento del lanzado de rayos con respecto a la versión basada en CPU (entre 18 y 44 veces más rápido), debido al paralelismo inherente al lanzado de rayos. En el caso de la rasterización, la versión en CPU ofrece mejor rendimiento que la basada en GPU (entre 4 y 23 veces más rápido) debido a la alta divergencia en la ejecución de los hilos que procesan triángulos de distintos tamaños o con distintos niveles de visibilidad (por *clipping* y/o *culling*). Además, el lanzado de rayos en el GPU presenta los mejores resultados de las técnicas implementadas. Sin embargo, los tiempos de respuesta son lejanos a los tiempos obtenidos por el *pipeline* OGL/D3D, siendo este último entre 8 y 128 veces más rápido que cualquiera de las técnicas implementadas.

Como trabajo futuro se propone un estudio más exhaustivo acerca de la división de la pantalla de visualización en mosaicos [3], para paralelizar de manera más eficiente el visualizador. Además, se deben considerar otras estructuras de datos para el recorrido de rayos en escena, como lo son el *BSP*, *Kd-trees*, y *BVH*, los cuales representan alternativas al *octree*. Especialmente en el caso del uso de *BSP* o *Kd-trees*, puede representar una buena opción debido a que los recorridos en estas estructuras organizan la información de la más cercana a la más lejana. También puede ahondarse en optimizaciones en el método de acceso de las estructuras de datos, utilizando accesos coherentes a la información, que ayudan a una lectura más rápida de la información por la GPU, o considerando otro algoritmos de recorrido de las estructuras de datos [17] [19].

REFERENCIAS

[1] S. D. Roth, "Ray casting for modeling solids," *Computer Graphics and Image Processing*, vol. 18, no. 2, pp. 109 – 144, 1982.

[2] T. Davidovic, T. Engelhardt, I. Georgiev, P. Slusallek, and C. Daschsbacher, "3D rasterization: a bridge between rasterization and ray casting," in *Proceedings of Graphics Interface 2012 (GI'12)*, Toronto, Canada, May 2012, pp. 201–208.

[3] S. Laine and T. Karras, "High-performance software rasterization on GPUs," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics (HPG'11)*, Vencouver, Canada, August 2011, pp. 79–88.

[4] Nvidia, "CUDA," [Accedido: 02-07-2015]. [Online]. Available: https://developer.nvidia.com/cuda-zone

[5] Khrones Group, "OpenCL," [Accedido: 02-07-2015]. [Online]. Available: http://www.khronos.org/opencl

[6] M. Deering, S. Winner, B. Schediwy, C. Duffy, and N. Hunt, "The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics," *SIGGRAPH Comput. Graph.*, vol. 22, no. 4, pp. 21–30, Jun. 1988.

[7] F. Liu, M.-C. Huang, X.-H. Liu, and E.-H. Wu, "FreePipe: A Programmable Parallel Rendering Architecture for Efficient Multi-fragment Effects," in *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ser. I3D '10. New York, NY, USA: ACM, 2010, pp. 75–82.

[8] Y. Chun and N. Baek, "A CUDA-based implementation of OpenGL-compatible rasterization library prototype," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC'14)*, Gyeongju, Korea, March 2012, pp. 1747–1748.

[9] A. Patney, S. Tzeng, K. A. Seitz, Jr., and J. D. Owens, "Piko: A Framework for Authoring Programmable Graphics Pipelines," *ACM Trans. Graph.*, vol. 34, no. 4, pp. 147:1–147:13, Jul. 2015.

[10] T. Aila and S. laine, "Understanding the efficiency of ray traversal on GPUs," in *Proceedings of the Conference on High Performance Graphics 2009 (HPG '09)*, Saarbrucken Germany, June 2009, pp. 145–149.

[11] C. Benthin, S. Woop, M. Niessner, K. Selgard, and I. Wald, "Efficient Ray Tracing of Subdivision Surfaces Using Tessellation Caching," in *Proceedings of the 7th Conference on High-Performance Graphics*, ser. HPG '15. New York, NY, USA: ACM, 2015, pp. 5–12.

[12] D. Kopta, K. Shkurko, J. Spjut, E. Brunvand, and A. Davis, "An Energy and Bandwidth Efficient Ray Tracing Architecture," in *Proceedings of the 5th High-Performance Graphics Conference*, ser. HPG '13. New York, NY, USA: ACM, 2013, pp. 121–128.

[13] O. Mattausch, J. Bittner, A. Jaspe, E. Gobbetti, M. Wimmer, and R. Pajarola, "CHC+RT: Coherent Hierarchical Culling for Ray Tracing," *Comput. Graph. Forum*, vol. 34, no. 2, pp. 537–548, May 2015.

[14] S. Woop, J. Schmittler, and P. Slusallek, "RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 434–444, Jul. 2005.

[15] M. Hapala, T. Davidovič, I. Wald, V. Havran, and P. Slusallek, "Efficient Stack-less BVH Traversal for Ray Tracing," in *Proceedings of the 27th Spring Conference on Computer Graphics*, ser. SCCG '11. New York, NY, USA: ACM, 2013, pp. 7–12.

[16] H. Wong, M. M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, March 2010, pp. 235–246.

[17] A. L. dos Santos, V. Teichrieb, and J. Lindoso, "Review and Comparative Study of Ray Traversal Algorithms on a Modern GPU Architecture," in *22nd International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision in co-operation with EUROGRAPHICS Association*, ser. WSCG' 14. Vaclav Skala - UNION Agency, 2014, pp. 203–212.

[18] T. Aila, S. Laine, and T. Karras, "Understanding the Efficiency of Ray Traversal on GPUs - Kepler and Fermi Addendum," NVIDIA Research, Tech. Rep., 2012.

[19] A. L. dos Santos, J. M. X. N. Teixeira, T. S. M. C. de Farias, V. Teichrieb, and J. Kelner, "kd-tree traversal implementations for ray tracing on massive multiprocessors: A comparative study," in *Computer Architecture and High Performance Computing, 2009. SBAC-PAD '09. 21st International Symposium on*, Oct 2009, pp. 41–48.