Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXX, págs. 3009-3019 (artículo completo)
Oscar Möller, Javier W. Signorelli, Mario A. Storti (Eds.)
Rosario, Argentina, 1-4 Noviembre 2011

UN FRAMEWORK EN PLACAS GRÁFICAS (GPU) PARA APLICACIONES BASADAS EN RAYTRACING

Juan P. D´Amato[a,b], Cristian Garcia Bauza[a,b], Marcelo Vénere[a,c]

[a]PLADEMA, Universidad Nacional del Centro, Tandil, Argentina
[b]Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)
[c]Comisión Nacional de Energía Atómica (CNEA)
{mdelfres, venerem}@exa.unicen.edu.ar, http://www.plademа.net

Keywords: Raytracing, GPU, Clasificación de mallas.

Abstract. Las técnicas de seguimiento del rayo o "raytracing" son de gran utilidad en aplicaciones gráficas y geométricas. Estas técnicas pueden ser aceleradas utilizando hardware intensivo como las placas gráficas(GPUs). Pero la utilización de estas tecnologías generalmente requiere realizar una reingeniería del proceso, resultando en aplicaciones más complejas y de difícil mantenibilidad.
En este trabajo se propone un patrón de diseño para aplicaciones de rayCasting que utilizan las placas graficas el cual contempla variar la forma en que se clasifican los elementos en el espacio así como simplifica la interfaz de comunicación con las GPUs. Se proponen variantes del algoritmo de rayTracing aplicadas en la Visualización de Mallas en forma eficiente.

1 INTRODUCTION

Los renderings *foto-realistas* hoy aplicados en video-juegos y simuladores se logran gracias a la utilización de "trucos" o técnicas aproximadas que simplifiquen los tiempos de visualización. Con las nuevas placas gráficas y con el aumento del poder de cálculo, cada vez estas técnicas dan lugar a estrategias más robustas y eficaces como la de *ray-tracing*, en la que se logran efectos de sombras y transparencias mucho más realistas.
Pero aún con esta mayor capacidad de cálculo, es sabido que en los métodos de *ray-tracing* es deseable contar con una estrategia que permita localizar qué elementos se encuentran en la dirección del rayo de forma eficiente (Hunt 2008), especialmente cuando se desea utilizar esta estrategia de forma interactiva. Estas estrategias son métodos de clasificación espacial, los cuales proponen realizar una división del espacio original en particiones o celdas distribuidas de forma conocida y asociarle los elementos o triángulos que son coincidentes con estas celdas. Los métodos de clasificación y búsqueda más conocidos en la literatura son las *Grillas regulares,* los *árboles n-arios y* los *Bounded-objects* (Assarsson 2007), (Lauterbach 2009).
Por cada uno de los métodos de clasificación nombrados, la forma en que se localizan las celdas varía sustancialmente. Uno de los objetivos de este trabajo es poder describir un comportamiento general de dichas estrategias que facilite su posterior implementación.

1.1 Ray-tracing en GPU

En el método de ray-tracing, se utilizan cientos de miles de rayos para generar una visualización de una escena, la cual se muestra en pantalla. La escena se compone generalmente por una malla de superficie con elementos triangulares y vértices. Por cada pixel de la pantalla, se genera un rayo que recorre la escena hasta encontrar una intersección con un triángulo y la posición de esta intersección se almacena. Cuanto mayor es la resolución de la imagen deseada, se requieren mayor cantidad de rayos, aumentando linealmente el costo computacional.
Por cada uno de los rayos, se realiza una búsqueda en una dirección, y se verifica si interseca con algún triángulo de la escena. A partir de la intersección encontrada, se vuelve a tirar un rayo buscando otros elementos, repitiéndose el proceso tantas veces como sea necesario. Cuando los elementos se encuentran clasificados, las búsquedas son más eficientes y el control de intersección debe realizarse pasando a través de la estructura de clasificación propuesta.
Es sabido que cada método de clasificación tiene sus ventajas y desventajas. Los métodos de grillas son fáciles de construir y actualizar, pero requieren mayor cantidad de memoria para lograr más precisión. Los métodos de árboles se adaptan a la distribución de tamaños de los elementos, pero su actualización es costosa y generalmente se utiliza para escenas estáticas. Los métodos de *Bounding Objects* funcionan bien cuando existen muchos objetos en movimiento, pero no son una buena aproximación a objetos muy irregulares. Elegir el método de clasificación adecuado depende en gran medida del dominio en la que se aplicarán los métodos de ray-tracing (Castro 2008).
Desde el punto de vista de extensibilidad de un sistema, es una ventaja poder extraer el comportamiento común de dichas técnicas. Esta tarea siempre se lleva a cabo en los diseños Orientados a Objetos (OO), pero aún no es una práctica común cuando se implementan algoritmos en placas gráficas. La consecuencia es que cada vez que se necesita implementar un nuevo algoritmo de clasificación y búsqueda se requiere implementar todo el código

nuevamente tanto en CPU como en GPU, con la complejidad adicional que tiene programar y probar estrategias en las GPUs (Garanzha 2010).
En este trabajo proponemos un patrón o *framework* para algoritmos de búsqueda y clasificación espacial, lo cual facilita enormemente el desarrollo y prueba de nuevos algoritmos. Para definir este patrón, es necesario realizar una abstracción del método de ray-casting y por otro lado un proceso estándar de comunicación y ejecución CPU-GPU.

En la siguiente sección se presentarán los métodos más conocidos de clasificación y búsqueda, de los cuales se extraerá un comportamiento común o patrón. A continuación, se discute una metodología para convertir las estructuras a un formato compatible con las placas gráficas y se realizan instanciaciones del patrón propuesto. Finalmente se aplicarán dichos algoritmos a casos tipo y se tomarán los tiempos de corrida de los algoritmos, tanto en CPUs como en GPUs.

2 MÉTODOS DE CLASIFICACIÓN Y BÚSQUEDA

Los elementos a buscar en el espacio serán atravesados por un rayo, el cual es definido como un segmento *origen-fin*. Si por cada rayo (m) es necesario verificar con cada elemento de la escena (n), la tarea se torna costosa computacionalmente, con una complejidad de $O(nm)$. Para lograr búsquedas eficientes, es necesario clasificar los elementos aplicando una estrategia de discretización del espacio utilizando estructuras rectangulares denominadas *celdas*.
El método de clasificación debe recorrer todos los elementos y asignarlos a las celdas, por lo cual siempre tendrá un costo de a lo menos la cantidad de elementos, $O(n)$. A continuación, el algoritmo de *ray-casting* recorre la estructura para localizar el punto de intersección entre una recta y un polígono, tarea ejecutada en tres pasos claramente diferenciados:

- Encontrar la celda donde se origina el rayo
- Buscar las celdas vecinas que son intersectadas por el rayo
- Verificar en estas celdas, qué elementos son efectivamente atravesados por el rayo.

El primer paso se corresponde con una búsqueda puntual sobre la estructura. El siguiente paso es recorrer los vecinos a la celda origen y si el rayo interseca la celda vecina, marcarla como visitada y añadirla a una cola de verificación. Ya que no todos los rayos que intersecan la celda también lo hacen con los polígonos que contienen, es necesario realizar una verificación *triángulo-rayo*. El algoritmo de intersección del rayo con un triángulo usado es el que describe (Jones 2000). En promedio, localizar un elemento tiene un costo quasi-constante o a lo sumo logarítmico con la cantidad de elementos. A continuación se plantea la localización de celdas con dos tipos de estructuras de clasificación.

2.1 Búsquedas direccionales en Grillas Uniformes (GU)

Una Grilla Uniforme o GU es una partición del espacio en celdas del mismo tamaño. En memoria se almacena y accede como una matriz, con un costo de acceso constante. Dicho método propone asignar los elementos a las celdas cuando los *bounding-box* de ambos son coincidentes. Esta metodología puede encontrarse en varios trabajos (Purcell 2002). Para una búsqueda sobre una GU, primero se computa el *bound-box* que contiene al rayo y se marcan todas las celdas incidentes con este. En un segundo paso, se descartan las celdas que no tienen intersección con el rayo, quedando sólo las celdas visitadas, como se muestra en la Figura 1.

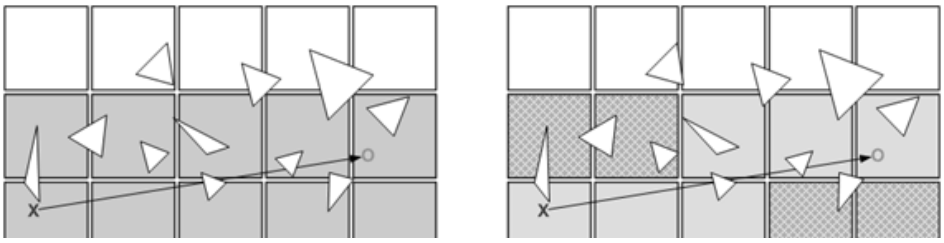

Figura 1 : (izq.) Celdas a ser visitadas potencialmente (der.) Celdas descartadas en gris y activas en azul.

El algoritmo de búsqueda se propone a continuación en dos dimensiones. Su extensión a 3D es inmediata.

Algoritmo 2.1: Búsqueda direccional en GU

Parámetros : *Src, Dst* is a Point3D

step 1. Buscar la celda que contiene a Src

h =root

step 2. Computar la caja(*bound-box*) que contiene al segmento Src-Dst

BoxTr = box(Src, Dst)

$$i_{min} = \left\lfloor \frac{(Box.min.x - D_{min}.x)}{d.x} \right\rfloor ; i_{max} = \left\lceil \frac{(Box.max.x - D_{min}.x)}{d.x} \right\rceil$$

$$j_{min} = \left\lfloor \frac{(Box.min.y - D_{min}.y)}{d.y} \right\rfloor ; j_{max} = \left\lceil \frac{(Box.max.y - D_{min}.y)}{d.y} \right\rceil$$

step 3. Verificar que el *bound-box* de la celda intersecta con el segmento

```
For i= imin to imax
 For j=jmin to jmax
   If cells[i,j].intersects(src, dst)
     resCells.add(cells[i,j]);
  End for
```

step 4. Verificar los triángulos de las celdas en este rango que son intersectadas por el rayo

```
For each c in resCells
 For each t in c
    If t.intersects(, src,dst)
      Result.add(t)
end for
```

Esta estrategia es quizá las más utilizada en GPUs, dada su distribución regular. El costo de creación y actualización de dicha estructura es lineal con el número de elementos. Por otro lado adolece del problema no balancear correctamente la cantidad de elementos que entran por celda.

2.2 Búsquedas direccionales en Árboles n-arios

Este método de clasificación propone realizar una división inicial del dominio en 4 celdas regulares (8 si es en 3 dimensiones) y asignar los elementos a dichas celdas con el mismo criterio que las GU, por superposición de *bounding-box*. Si alguna de las celdas contiene muchos elementos (definido por una tolerancia), la misma se divide en 4 nuevas celdas, denominadas hijas, y los elementos que la contienen se a asignan a estas celdas. Para evitar divisiones reiteradas, se añade una cota de mínimo tamaño de división; en caso que la celda hija sea menor que dicha cota, no se divide. El resultado es una estructura jerárquica de celdas.

Las búsquedas en árboles plantean retos adicionales, por su estructura en niveles. Para localizar un elemento en una dirección, se inicia por el nodo raíz, y se visitan de forma recursiva aquellos nodos hijos que son intersectados. Este recorrido visita primero un nodo padre y luego cada nodo hijo, siendo un recorrido conocido como *top-down*. Cuando el nodo intersectado es *terminal*, se verifica si alguno de los triángulos contenidos, intersecan con el rayo (como en el caso de las GU). En la Figura 2 se muestran las celdas visitadas para el caso propuesto.

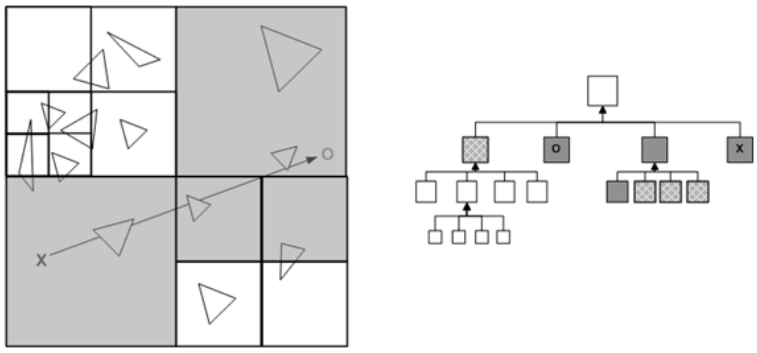

Figura 2 : (izq.) Caso de una búsqueda direccional (der.) Celdas alcanzadas en la estructura

El recorrido tal como se presentó se describe como **Algoritmo 2.2**

Algoritmo 2.2: Búsqueda direccional en AN

Parámetros : *Src, Dst* is a Point

step 1. Buscar la celda que contiene a Src

h =root

step 2. Verificar si alguna de las celdas es intersectada por el rayo

```
If  h is leaf
  resCells.add(h)
else
for each child in h.children
  if child.intersects(Src,Dest)
      h = child;
      goto step 2;
end while
```

step 3. Verificar el contenido de las celdas en este rango que son intersectadas por el rayo

Idem Algoritmo 2.1

El costo de creación y actualización de dicha estructura es un tanto más elevado que en las GU, siendo del orden de $O(n \log n)$ y las búsquedas son $O(\log n)$. Por otra parte, las ventajas de estos métodos radican en lograr un adecuado balance de celdas y elementos por celdas, pero no son populares por su estructura en niveles.

2.3 Raytracing en GPU

La utilización de GPUs para el cálculo de ciertas partes de un algoritmo, añade al procedimiento original etapas intermedias de generar estructuras y copiar datos de un espacio de memoria (CPU) al otro (GPU). A continuación, cada rayo es procesado en paralelo y el resultado se almacena en una imagen. Los pasos del lado de la CPU para correr el algoritmo de búsqueda son:

- Generar estructuras compatibles para GPU
- asignar espacio de memoria de salida.
- asignar cada búsqueda a un único proceso a un único proceso/*thread*.
- Copiar datos CPU->GPU
- resolver la búsqueda en la GPU
- Copiar datos GPU->CPU
- leer y procesar el resultado en forma secuencial

Una vez inicializadas las estructuras y el programa para ejecutar en GPU, se copian los datos a la memoria de la GPU; se carga el algoritmo de búsqueda programado en forma de códigos *scripts* compatible con *openCL* y se ejecutan y leen los resultados nuevamente a memoria CPU. La mayor complejidad se encuentra en generar las estructuras de clasificación de forma tal que sean compatibles con la arquitectura de la GPU.

3 GENERALIZACIÓN DE LOS MÉTODOS

Para resolver la problemática de generar y probar diversos algoritmos para *ray-tracing* se propone una estructura patrón con las representaciones comunes a dichas estructuras. Estos algoritmos nombrados conviven en dos arquitecturas, la CPU y la GPU, por lo que es necesario plantear una solución integral, que contemple los pasos mostrados, tal cual se muestra en la Figura 3.

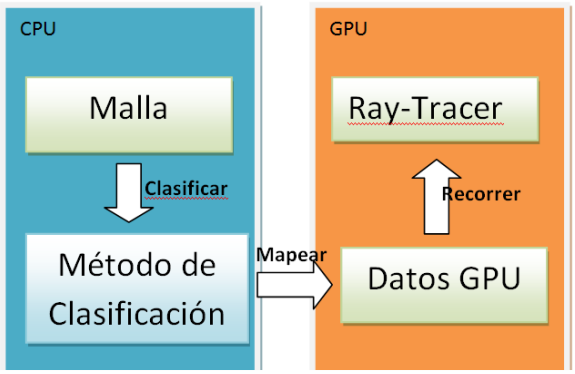

Figura 3 : Esquema del circuito de los datos en CPU y GPU

Dicha solución contempla por el lado de la CPU generalizar la forma de convertir las estructuras; por el lado de la GPU, un algoritmo patrón sobre el cual corre dicho ray-tracing. La forma de clasificar es propia del método a implementar. A continuación se describe el método de preparación de datos para llevar las estructuras a GPU y el método general de *ray-tracing*.

3.1 Conversión de datos

Del lado de la CPU es más común utilizar listas de elementos que administran la posición de los objetos en memoria dispersa y punteros entre objetos para indicar una relación. Del lado de las placas gráficas, los mismos datos deben encontrarse en alguna forma consecutiva, es decir, alojados en una estructura tipo matriz o vector. Para lograr una compatiblidad CPU-GPU, los elementos con sus propiedades deben agruparse y las relaciones *celdas-triángulo-vértice* que existían entre los elementos deben ser convertidas mediante direccionamientos por índices. Aunque es una tarea un tanto tediosa, es fácilmente generalizable utilizando un mecanismo denominado *serialización.* La serialización, como se define en la Programación Orientada a Objetos, es el proceso de codificación de un Objeto en un medio de almacenamiento (como puede ser un archivo, o un buffer de memoria) con el fin de transmitirlo como una serie de bytes o en un formato más legible.
Este mecanismo contempla que por cada elemento de la malla (*Vertex, Triangle,Cell*) se les asocie un *identificador (ID)* o indicador de *posición* del elemento en memoria. Los atributos de interés de estos elementos (posición, normal, color y otros) se deben almacenar en la posición del arreglo/vector que indica dicho identificador utilizando un tipo de dato compatible con el estándar de *OpenCL*. Como los vértices y los triángulos tienen diferentes tipos de datos es conveniente separarlos en diferentes estructuras. Generalmente los datos de los triángulos son índices a vértices (tipo *Entero)* y los vértices son número reales (tipo *Float* para las 3 coordenadas de la posición, las normales y texturas).
El primer paso es por lo tanto convertir las estructuras originales en un formato compatible o "inmediato", asignando un ID a cada elemento de forma incremental comenzando en el 0. A continuación, se define el espacio de memoria requerido por los triángulos y por los vértices generando una estructura tipo buffer, y finalmente se recorren todos los elementos y almacenan sus datos en la posición indicada por dicho ID en el *buffer* correspondiente. La distribución en memoria de los datos al aplicar este esquema resulta en dos bloques de memoria separados (vértices por un lado indicados en color *rojo*, triángulos que referencian a los vértices indicados en *azul* por otro); sabiendo que cada elemento ocupa la misma cantidad de memoria, como en la Figura 4.

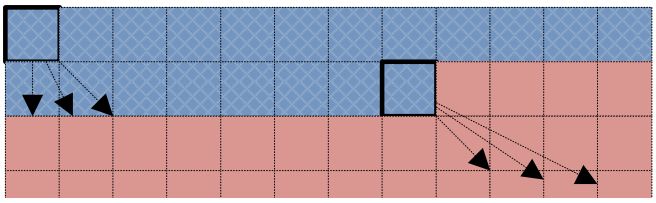

Figura 4 : Espacio de memoria resultante de una representación compacta

Cuando los elementos de las mallas se encuentran clasificados, se agrega un nivel de indirección a los elementos; primero se deben acceder a las celdas, luego a los triángulos y finalmente a los vértices. De forma similar al proceso anterior, las referencias de las celdas a los triángulos se convierten en un índice. Esta indirección se almacena en otra estructura que indica la celda y los triángulos a los que referencia. Cuando un triángulo pertenece a más de una celda, se replica su información. La conversión se esquematiza en la Figura 5

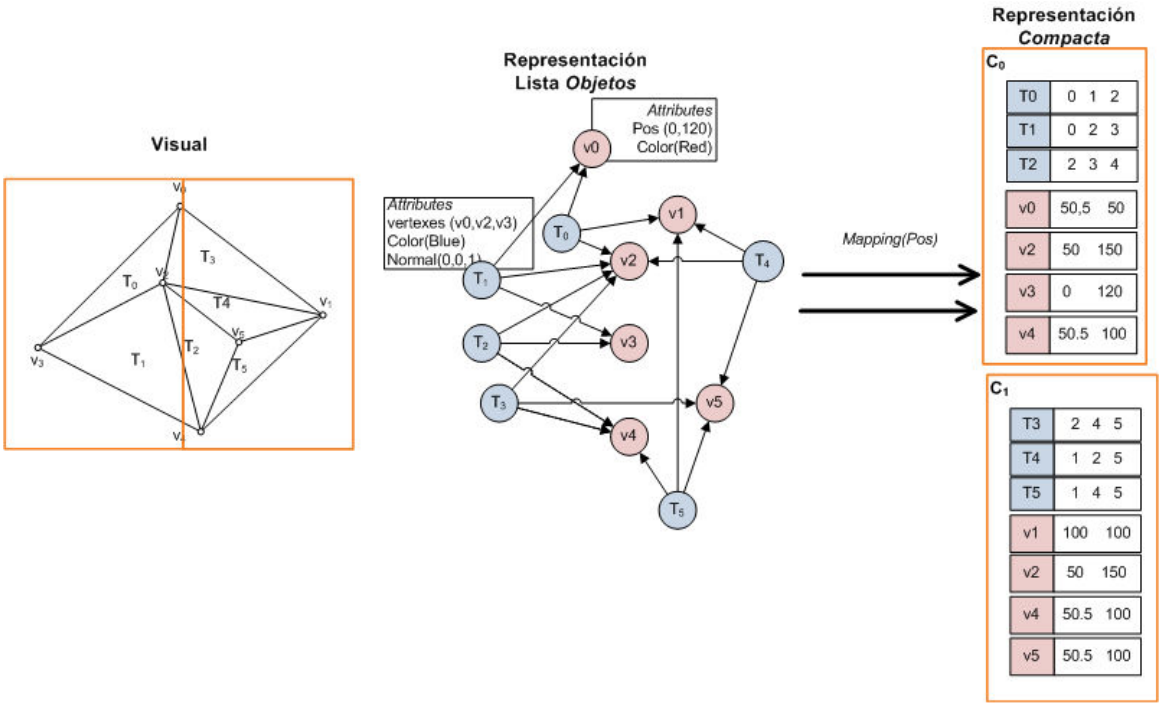

Figura 5 : Mapeo de listas de celdas/triángulos/vértices a su forma compacta

Esta representación compacta son bloques de memoria contiguos, los cuales se copian y manipulan en una clase *GPUBuffer* la cual implementa los métodos para copiar a GPU (copyToGPU) y leerse a memoria de CPU(readFromGPU) . La representación resultante no es la única, es posible eliminar las indirecciones de los triángulos a los vértices, reemplazando los triángulos por la información de posición del vértice. Esta alternativa aumenta la redundancia y utiliza más memoria.

Como la estructura interna de las celdas de clasificación varía, este método debe implementarse por cada método de clasificación propuesto. Para los árboles, se añade un nivel de indirección en las celdas, para acceder a las celdas hijas, para lo cual se utiliza un valor de tipo entero como índice. Cuando este índice es negativo, la celda es terminal.

3.2 Algoritmo de ray-tracing

De los métodos propuestos, se observa que la diferencia principal se encuentra en la forma de

localizar la primera celda y acceder luego a las celdas vecinas. En el método de grilla, los accesos a vecinos son implícitos, y se acceden variando los índices. En los árboles, existen nodos hijos de otros, los terminales son aquellos que incluyen los elementos.
En este sentido, se propone que de acuerdo al método escogido se implemente la forma de recorrer las celdas que son alcanzadas con el rayo y se retornen en esta lista (paso 1 al 3 del Algoritmo 2.1 y el paso 1 y 2 del algoritmo 2.2)., logrando que el resto del método de *ray-tracing* se conserva invariante. Finalmente, se recorren todas las celdas para localizar el triángulo que intersecta con el rayo y se guarda la posición más cercana (función *nearer*) al origen del rayo. Como mejora, se podrían ordenar las celdas con respecto a la distancia al origen.
Aunque la forma de estructurar y recorrer las celdas puede sistematizarse aún más utilizando grafos direccionales, el problema es que una estructura más general afecta considerablemente la performance. Por esta cuestión se concluye que el método de localización de celdas es mejor re-implementarlo por cada método de clasificación. Por otro lado, el proceso de *ray-casting* propuesto tiene la siguiente forma.

Algoritmo 3.1 : Pattern Algorithm

```
For each r in rays
{
 for each iteration
  cL= locateCells(ray.src, ray.dst)* //método a implementar de acuerdo a la clasificación
  For each c in cL
    // checkContent
    For each t in c.content
     If intersect(t,ray) && nearer(Ray. bestTriangle,t)
       buffer[r.id, iteration] = t;
       r.dir = r.dir x t.normal
       bestTriangle = t;
}
```

El método de *ray-tracing* es iterativo; a partir de una intersección con un triángulo encontrada el algoritmo realiza una nueva búsqueda direccional. Si no encuentra ninguna intersección, la búsqueda para ese rayo termina. Como cada triángulo puede tener gran cantidad de propiedades adicionales a la información geométrica(transparencia, reflectancia, texturas, colores) que puede demandar un esfuerzo adicional para llevar a memoria de GPU, se propone simplificar la interfaz y únicamente almacenar las posiciones y el ID del elemento de la intersección encontrada.
Cada iteración se almacena en un buffer de salida independiente. En la primera iteración (*i0*) los rayos provienen de la posición de los pixeles y la dirección se encuentra dada por la matriz de vista. El resultado es equivalente a dibujar el triángulo en pantalla. En la segunda iteración (*i1*), los rayos inician en las intersecciones encontradas y el destino se encuentra en la posición de la luz (la cual es única).

El resultado de las iteraciones se compone en CPU sólo una vez finalizado el *ray-tracing* en la cual además se aplica un modelo de iluminación *flat*; como se muestra en la Figura 6.

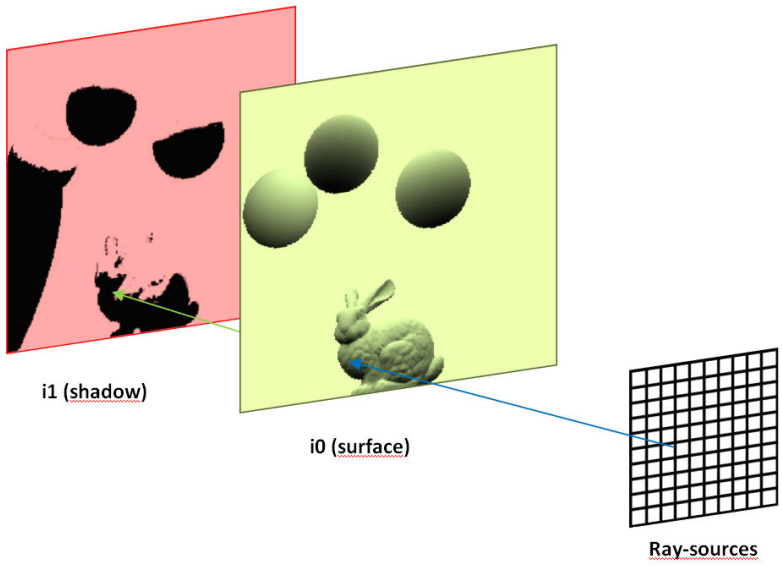

Figura 6 : Composición del resultado

4 RESULTADOS Y PRUEBAS

La metodología fue aplicada utilizando 2 métodos nombrados (GU y Árboles) los cuales corrieron tanto en CPU como en GPU. El algoritmo de ray-tracing se aplicó con una profundidad 2 (logrando sombras), aunque puede fácilmente aplicarse a más niveles para obtener reflejos sobre objetos. En la Figura 7 se muestran imágenes obtenidas con los distintos métodos en una resolución de 1024x1024, los cuales no difieren en el resultado. En la imagen de la derecha(conejo), se encuentran algunos artefactos por triángulos mal definidos en la malla.

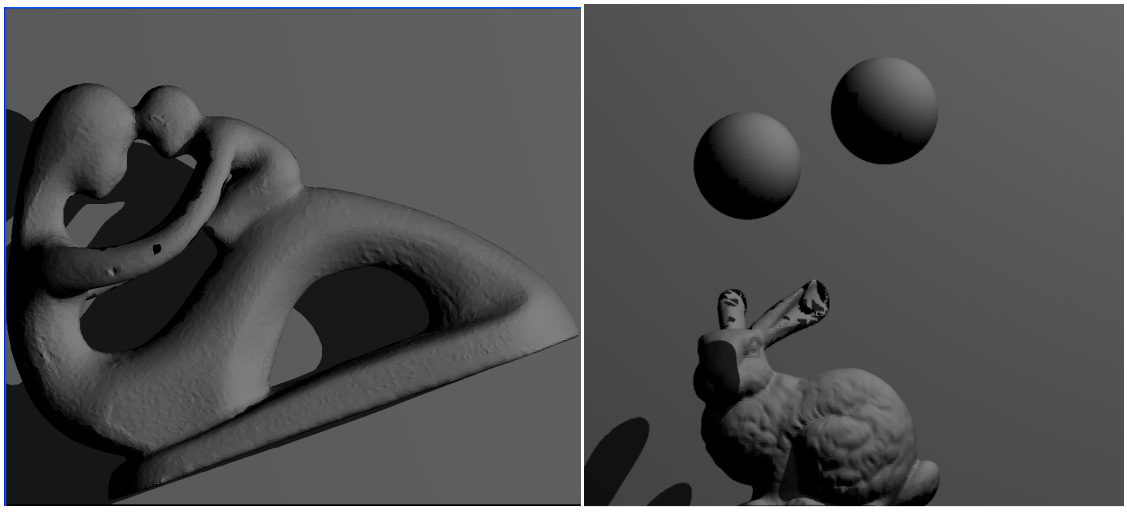

Figura 7 : Imágenes obtenidas con los métodos (izq.) Mujer e hijo de 50,000 triángulos (der.) Conejo de 90,000 triángulos.

A continuación, en la Tabla 1, se enumeran los tiempos obtenidos utilizando una resolución de 256x256 rayos en una placa gráfica NVIDIA GTX260 y una CPU de 2.0 GHZ.

	#Triángulos	GRID	Octree	GPUGrid	GPUTree
Conejo	90k	7909 ms	10401 ms	459 ms	672 ms
Mujer e hijo	50k	5724 ms	5898 ms	154 ms	210 ms
Gears	19k	3378 ms	2792 ms	93 ms	129 ms

Tabla 1 : Tiempos del ray-tracing

Observando los tiempos de actualización, con la placa utilizada es posible lograr imágenes sobre escenarios con una cantidad interesante de triángulos (50K) que pueden actualizarse de forma casi interactiva, de 8 a 10 cuadros por segundo. La estructura propuesta facilitó la incorporación de un tercer algoritmo de clasificación y búsqueda de mallas comprimidas (GUc) el cual se encuentra en proceso de evaluación.

5 CONCLUSIONES

Se ha propuesto un un patrón de trabajo para diseñar, implementar y extender un algoritmo de *ray-tracing* variando las estructuras de búsquedas. Se propuso utilizar celdas de forma rectangulares, pero el algoritmo puede extenderse para soportar celdas esféricas, añadiendo únicamente el tipo de celda utilizada.

Los *speed-upds* alcanzados, aunque no son tan altos como los presentados en otros trabajos, son razonables para obtener imágenes en tiempo real, a la vez que la metodología facilitó el desarrollo y prueba de varios algoritmos. Como trabajos futuros se espera añadir ciertas características visuales, tales como reflejos y transparencias que permitan generar imágenes mucho más realistas

REFERENCES

Assarsson U., Holzschuch N. Whitted Ray-Tracing for Dynamic Scenes using a RaySpace Hierarchy on the GPU. In Proceedings of Eurographics Symposium on Rendering (2007)

Castro, R., Lewiner, T., Lopes, H., Tavares, G., And Bordignon, A. L. Statistical optimization of octree searches. Computer Graphics Forum 27, 1557–1566 (2008).

Garanzha K. , Loop C., Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing EUROGRAPHICS 2010 , Volume 29, Number 2, (2010).

Jones R., Intersecting a Ray and a Triangle with Plücker Coordinates, Ray Tracing News, 13. webSite : http://www1.acm.org/pubs/tog/resources/RTNews/.(2000)

Hunt W., Mark W. Ray-Specialized Acceleration Structures for Ray Tracing. In Proceedings of the 2008 IEEE/EG Symposium on Interactive Ray Tracing (2008).

Lauterbach C., Garland M., Sengupta S.,Luebke D., Manocha D. Fast Bvh Construction On Gpus. In Proceedings Of Eurographics (2009)

Mametsa H.J., Laybros S., Bergès A., Asymptotic Formulations and Shooting Ray Coupling for Fast 3D Scattering Field Evaluation from Complex Objects and Environment–Applications, ICEAA, Turin (Italy), (2003)

Möller T. and Trumbore B., Fast, Minimum Storage Ray-Triangle Intersection, Journal of Graphics Tools, vol. 2, no. 1, pp. 21–28, 1997.

Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P. Ray tracing on programmable graphics hardware. ACM Trans. Graph., 703-712 (2002).

Subramanian K. R. and Fussel D. S., A search structure based on k-d trees for efficient ray tracing. Technical Report (Ph.D. Dissertation), Tx 78712-1188, The University of Texas at Austin, (1990).