Un algoritmo de recorrido vectorizado para Ray-Tracing

José María Noguera
Dept. de Informática
Escuela Politécnica Superior
Universidad de Jaén
23071 Jaén
jnoguera@ujaen.es

Carlos Ureña
Dept. Lenguajes y
Sistemas Informáticos
ETS Ingeniería Informática
Univ. de Granada
18071 Granada
curena@ugr.es

Resumen

En este artículo presentamos un algoritmo optimizado de ray-tracing que es capaz de mejorar a los algoritmos estándar existentes. Para ello, este algoritmo procesa simultáneamente un conjunto amplio de rayos y realiza un único recorrido del índice espacial de la escena para todos ellos, lo cual permite aprovechar eficientemente las funcionalidad SIMD del hardware y produce accesos coherentes a memoria. Además, durante dicho recorrido único se agrupan los rayos de tal forma que las citadas ventajas se mantienen incluso para conjuntos de rayos no coherentes.

Se ha comprobado que el algoritmo reduce los tiempos de cálculo respecto a otras soluciones estándar, especialmente para conjuntos de rayos no coherentes. Cabe destacar que sus características lo hacen especialmente adaptable a hardware gráfico (GPUs)

1. Introducción

Ray-tracing es la técnica dominante en el ámbito de la síntesis de imágenes realistas o de gran calidad, cuando el tiempo de cómputo es un factor secundario. Los grandes tiempos de procesamiento se deben a los elevados costes computacionales que requiere el trazado de rayos. Para cada píxel que conforma la imagen es necesario lanzar un o varios rayos que atraviesen la escena, intersectarlos con los objetos que la conforman, calcular el color, y finalmente mostrar todos los píxeles así obtenidos en la pantalla.

Para aplicaciones que interactúen en tiempo real con el usuario, es necesario disponer de algoritmos que permitan producir imágenes muy rápidamente. En estos entornos usualmente se emplean otro tipo de técnicas de renderizado, tales como rasterización de triángulos con z-buffer. Además, en los últimos años ha tenido un gran auge el desarrollo de hardware gráfico específicamente optimizado que permite acelerar aún más la renderización mediante rasterización de triángulos.

No obstante, ray-tracing ofrece un gran número de características ventajosas sobre los algoritmos basados en rasterización, y sería beneficioso disponer de ellas en el renderizado en tiempo real:

- Calidad y corrección de la imagen: Los efectos visuales que se obtienen con métodos de rasterización de triángulos son muy limitados. Muchos de los efectos que se consiguen se deben a un meticuloso diseño manual de la escena. La aparición de hardware gráfico programable (GPUs) ha permitido un gran avance en este aspecto, pero es difícil obtener ventaja de estas características pues son difíciles de implementar y cuando se consigue, se trata simplemente de aproximaciones a la realidad. Esto contrasta con la facilidad y naturalidad con que ray-tracing permite calcular

refracciones, reflexiones, sombras, iluminación indirecta y otros efectos de forma físicamente correcta.

- Escenas complejas: Para escenas complejas, el hardware gráfico es limitado pues la rasterización presenta costo lineal al número de primitivas de la escena. Por ello requiere técnicas de preprocesamiento sofisticadas para reducir la geometría. El problema se agrava si se emplean efectos que requieren múltiples pasadas de renderizado para la misma escena. Frente a ello, ray-tracing presenta complejidad logarítmica respecto al número de primitivas de la escena e incorpora *occlusion culling*. Tan solo es preciso realizar aquellos cálculos que sean realmente necesarios para la escena final pues los *shaders* solo se procesan una vez determinada la visibilidad. Por último, la naturaleza del algoritmo de ray-tracing permite una paralelización trivial.

El trazado de rayos no solo resulta útil para la visualización. Existen multitud de algoritmos, especialmente en el área de la síntesis de imágenes realistas, que basan su funcionamiento en el trazado de rayos, como por ejemplo, *photon-mapping* [15]. Además en estos casos se suelen trabajar con rayos no coherentes. Disponer un algoritmo de ray-tracing eficiente permitiría también obtener mejoras sustanciales en los tiempos de ejecución de estos algoritmos.

Por todas estas razones, creemos interesante el desarrollo de nuevas técnicas que permitan aproximar el ray-tracing al mundo de las aplicaciones en tiempo real.

Hay que tener en cuenta que usar este algoritmo para Ray-Tracing, o, en general, Iluminación Global, supone que la aplicación debe generar conjuntos grandes de rayos para calcular las intersecciones simultáneamente en todos ellos. Esto supone que el algoritmo se debe organizar siguiendo un esquema "primero a lo ancho" en lugar de "primero en profundidad". Por ejemplo, para Ray-tracing, se deben generar todos los rayos primarios, luego todos los secundarios, etc...

Si bien esto puede suponer un incremento en uso de memoria, creemos que la reducción de los tiempos de cálculo hace que merezca la pena. Otro beneficio adicional es que este esquema permite la implementación vectorizada y eficiente de otras partes del algoritmo, como puede ser la evaluación del modelo de iluminación local, o la generación de los rayos secundarios.

El artículo se organiza de la siguiente forma: la sección 2 muestra una descripción general de la filosofía empleada en el algoritmo. La sección 3 describe con más profundidad el funcionamiento del mismo. Por último, en la sección 4 se ofrece una comparación de eficiencia frente a otros métodos.

1.1. Trabajos previos

Pese a que ray-tracing es una técnica muy antigua [1], su uso para aplicaciones interactivas es relativamente nuevo.

Las GPUs están optimizadas para rasterización de triángulos, aunque no obstante, ha habido diversos intentos de adaptar su uso para acelerar el ray-tracing [3, 8, 5, 4]. Actualmente, los resultados obtenidos en GPUs apenas mejoran ligeramente los resultados que se obtienen en CPUs convencionales. Ello es debido a que si bien las GPUs son arquitecturas paralelas de cómputo muy poderosas, presentan dificultades para aplicaciones con alto control de flujo. Si bien esto posiblemente cambiará en el futuro por el continuo desarrollo de las GPUs. Por tanto, consideramos que sigue siendo interesante desarrollar técnicas de aceleración de ray-tracing en CPUs que sean adaptables a su uso en GPUs.

Wald [14, 13] presenta un algoritmo basado en CPU que explota la coherencia existente en los rayos primarios. Se basa en la idea de que los rayos coherentes, con mucha posibilidad, harán el mismo recorrido a través de la estructura jerárquica de indexación espacial que contiene la escena, e intersectarán con los mismos objetos.

Su propuesta, pues, consiste en recorrer el índice espacial y realizar las intersecciones rayo-triángulo con paquetes de cuatro rayos coherentes simultáneamente. De esta forma, y

empleando las instrucciones SIMD de los procesadores actuales, reduce el tiempo de cálculo del algoritmo al realizar los cómputos para los cuatro rayos en paralelo. Además, Wald presenta un meticuloso diseño orientado a minimizar la complejidad del código y las fallas de caché.

Esta idea es válida para rayos primarios pues es fácil localizar rayos que a priori tengan altas posibilidades de ser coherentes (aquellos que atraviesen el mismo píxel o píxeles consecutivos de la imagen). Pero falla para rayos secundarios donde no es posible hacer esta presunción. En este artículo presentamos una ampliación de esta idea, que permite paralelizar los cálculos de distintos rayos en cualquier situación, aún cuando éstos no sean coherentes.

2. Implementación del trazado de rayos

Las siguientes secciones describen nuestra implementación del trazador de rayos. Primeramente se muestra una descripción general de la filosofía empleada y cómo se a aplicado al algoritmo. En las siguientes secciones se describe los detalles de su implementación.

2.1. Motivaciones

Dado un árbol de indexación espacial que contiene la escena, el algoritmo clásico de raytracing recorre una vez por cada rayo el árbol de forma recursiva. Wald en cambio, propone recorrer el árbol una vez por cada paquete de cuatro rayos. Nuestra propuesta avanza en esa idea, y propone recorrer una única vez el árbol pero con todos los rayos simultáneamente.

Este acercamiento proporciona las siguientes ventajas. Siempre que existan dos o más rayos que necesiten atravesar el mismo nodo del árbol, lo harán a la vez, permitiendo paralelizar los cómputos mediante las instrucciones SIMD del hardware. Esto se traduce en unas posibilidades mucho mayores de disponer en cada nodo del árbol de varios rayos a los que procesar en paralelo, especialmente en los nodos superiores el árbol. Incluso aunque no se presente coherencia entre los rayos, al ir avanzando por la estructura, éstos se irán clasificando de forma automática permitiendo mantener las citadas ventajas.

Otra ventaja es que al tener que visitar cada nodo del árbol una única vez, y no decenas de miles como en algoritmos previos, se reduce en gran medida en número de accesos que hay que realizar a la estructura de indexación espacial de la escena. Un menor número de accesos a memoria principal se traduce en una reducción de latencias y del número de fallos de caché.

Se requiere de una nueva estructura de datos que contenga los rayos a procesar. Esto no incurre en un mayor número de accesos a memoria frente a los algoritmos clásicos de raytracing, pues tan solo se organiza de forma agrupada la misma información que también es necesario almacenar y acceder en dichos algoritmos clásicos.

2.2. Paralelismo mediante SIMD

Los procesadores actuales incorporan extensiones SIMD que permiten realizar múltiples operaciones en coma flotante (usualmente cuatro) con una sola instrucción, aumentando así el rendimiento de este tipo de cálculos. Por su gran disponibilidad, hemos decidido emplear el juego de instrucciones SSE de Intel para explotar el paralelismo de datos del trazado de paquetes de rayos.

3. Recorrido del kd-tree

En nuestra implementación, para crear un índice espacial de la escena empleamos un kd-tree en el que el plano que divide a un vóxel pasa justo por el centro, dividiendo el vóxel en dos subvóxeles del mismo tamaño. El algoritmo se puede modificar fácilmente para trabajar con planos de separación no situados en el centro. La principal razón de esta elección frente a otras estructuras tales como Octrees o BVHs (Bounding Volume Hierarchies) es la simplicidad del algoritmo de recorrido. Para cada nodo y rayo solo hay que tomar dos decisiones: o seguir o no seguir por cada uno de los dos nodos hijos. Además este tipo de estructura jerárquicas se adaptan bien a la com-

plejidad de la escena. Y es fácil de paralelizar empleando instrucciones SSE.

Para poder paralelizar los cálculos, en cada nodo hemos de tener disponible todos los rayos que lo atraviesen. Primeramente se expondrá el funcionamiento del algoritmo para un único rayo, y posteriormente se expandirá para su uso con todos los rayos simultáneamente.

3.1. Algoritmo de recorrido para un rayo

Se emplea una adaptación a árboles-kd (con planos divisores en la mitad de los voxels) del algoritmo de recorrido paramétrico incremental para octrees presentado en [10]. Sea $r = p + td$ la ecuación de un rayo expresado en forma paramétrica, siendo p el vector origen, d el vector director unitario y $t \geq 0$ el parámetro real que determina un punto a lo largo del rayo.

Para cada nodo, t_{x0}, t_{x1} son los valores del parámetro del rayo para el que éste intersecta con dos planos perpendiculares al eje X que delimitan al voxel asociado al nodo (igualmente, llamamos t_{y0}, t_{y1} a los correspondiente valores en Y, y t_{z0}, t_{z1} a los valores en Z). Asumimos que $t_{x0} < t_{x1}$. Dichos valores se calculan para el nodo raíz y a partir de ahí se recalculan de forma incremental en cada paso del recorrido del árbol.

Sea t_{xm} el parámetro para el cual el rayo atraviesa el plano (perpendicular a X) que divide al vóxel en dos subvóxeles iguales. Puede calcularse como: $t_{xm} = (t_{x0} + t_{x1})/2$. En el caso de un voxel del árbol-kd dividido en dos por un plano perpendicular a X, los valores de interseccion para el primer sub-nodo visitado son t_{x0}, t_{xm}, y el segundo de ellos t_{xm}, t_{x1}. Los valores t_{ym}, t_{zm} se definen, calculan y usan de forma análoga, en los casos de voxels divididos por planos perpendiculares a Y o Z. En general, el cálculo de los valores de los nodos hijos de un nodo solo requiere sumar y dividir por dos. Puede demostrarse fácilmente ([10]) que el rayo intersecta al nodo si se cumple que:

$$max(t_{x0}, t_{y0}, t_{z0}) \; < \; min(t_{x1}, t_{y1}, t_{z1})$$

El algoritmo funciona de forma recursiva. Para cada nodo interno, se actualizan los parámetros del rayo para su primer hijo, y se determina si lo intersecta o no. Si esto ocurre, se procede inmediatamente con dicho nodo. Posteriormente, se repite el proceso para el segundo hijo.

Si se recorre el árbol seleccionando siempre primero el primer nodo situado en la dirección del rayo, una vez se encuentre una intersección en un nodo visitado puede terminarse la recursividad pues cualquier otra intersección que pudiera encontrarse será siempre más lejana de la ya localizada.

3.2. Algoritmo de recorrido para múltiples rayos

El algoritmo de recorrido del kd-tree para múltiples rayos funciona de forma similar al anterior. La idea consiste en procesar simultáneamente todos los rayos disponibles, realizando un único recorrido del índice espacial para todos ellos.

Su pseudocódigo puede verse en el algoritmo 1. Cada nodo recibe una lista de rayos que lo intersectan, con sus correspondientes parámetros del rayo ajustados para ese nodo. Si el tamaño de la lista es cero, retornamos. Si el nodo es hoja, la función intersecTrg calcula si los rayos que han llegado hasta el nodo intersectan con los triángulos indexados dentro del nodo.

En caso contrario, estamos en un nodo interno. La función calcNodo emplea SSE para actualizar los parámetros de todos los rayos para el primer hijo y para determinar cuáles de ellos lo intersectan. Todos aquellos rayos que intersecten a dicho hijo son copiados a una nueva lista. Por tanto, esta nueva lista contendrá un número de rayos comprendido entre cero y el número de rayos que atraviesan al nodo padre. A continuación, y de forma recursiva, se recorre el primer hijo suministrándole la nueva lista de rayos calculada. Una vez se termine de recorrer ese hijo, se procede de forma análoga con el segundo hijo.

```
Recorrer( nodo, ListaRayos )
   if numRayos = 0
      return;
```

```
if esHoja(nodo)
   intersecTrg( nodo, ListaRayos );
else foreach hijo h_i of nodo
   ListaRayosH_i=calcNodo(h_i,ListaRayos);
   Recorrer( h_i, ListaRayosH_i );
```

Algoritmo 1: Recorrido para múltiples rayos

A dichas listas no se copiarán aquellos rayos para los que ya se haya encontrado una intersección, ni aquellos rayos situados más lejos del parámetro máximo del rayo permitido, evitando así cálculos innecesarios.

El número total de operaciones requeridas para determinar si cuatro rayos intersectan un nodo son: una suma, una multiplicación, una operación de máximo, otra de mínimo y una de comparación entre el resultado de las dos anteriores, todas ellas SSE.

Al trabajar con múltiples rayos de manera simultánea puede ocurrir que diferentes rayos requieran recorrer el árbol de indexación espacial en distinto orden. El orden en el que un rayo ha de visitar los nodos en cada paso del algoritmo depende de los signos de las tres componentes del vector director del rayo. Por tanto, esto podría ocurrir si existen rayos con vectores directores con distinto signo, lo cual es muy probable. En estos casos no nos es posible recorrer una única vez el árbol con todos los rayos simultáneamente sin perder la deseable característica de que una vez localizada la primera intersección, ésta sea la más próxima.

La solución empleada para solventar este problema consiste en que en un primer paso se clasifican todos los rayos en ocho grupos en función de los signos de las componentes de sus vectores directores. Ahora, como todos los rayos de un mismo grupo recorrerán el árbol de indexación exactamente en el mismo orden, puede aplicarse el algoritmo de recorrido sin problema para cada uno de los grupos.

Con esta idea, tendremos que recorrer el árbol como máximo ocho veces, número que aún resulta muy inferior a las decenas de miles de veces que requieren recorrerlo los algoritmos convencionales. En la práctica, para rayos primarios nunca es necesario recorrerlo más de cuatro veces, siempre que el vector normal al plano de visión sea paralelo a uno de los ejes.

3.3. Disposición de los rayos en memoria

A la hora de se diseñar las estructuras de datos que almacenen los rayos, se debe elegir una disposición de los datos en memoria que permita maximizar la eficiencia y tiempo, teniendo en cuenta la implementación mediante instrucciones SIMD [6].

Se emplean dos estructuras de datos para gestionar los rayos. Una es global, puede accederse desde cualquier nodo y almacena todos los rayos y sus atributos, mientras que la otra pertenece a cada nodo en concreto y solo almacena algunos datos necesarios para el recorrido, así como punteros a la estructura global.

Para cada rayo es necesario almacenar de forma global:

- Su vector director y origen.
- El valor máximo del parámetro del rayo permitido.
- Un flag indicando si se ha encontrado una intersección con un triángulo.
- El identificador del triangulo más cercano intersectado.
- El valor del parámetro del rayo para el cual lo intersecta.
- Las coordenadas paramétricas del triángulo en el punto de intersección.

Como se explicó en la descripción del algoritmo, durante el recorrido del árbol, cada nodo recibe una lista de rayos que le intersectan. Dicha lista es una estructura de datos local para cada nodo, y almacena para cada rayo los valores de los parámetros del rayo de entrada y de salida del vóxel $(t_{0x}, t_{0y}, t_{0z}, t_{1x}, t_{1y}, t_{1z})$, pues dependen del nodo en el que nos encontremos. También se almacena para cada rayo un puntero a la estructura global que almacena el resto de información del rayo.

A estos datos se les aplicará operaciones SIMD, por lo que para obtener el máximo rendimiento deben de almacenarse con una organización concreta denominada "estructura de vectores" tal y como se aprecia en la figura 1.

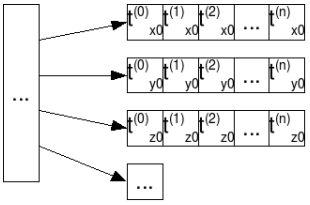

Figura 1: Estructura de Vectores

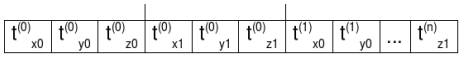

Figura 2: Vector de Estructuras

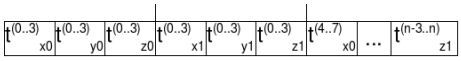

Figura 3: Mixto

En esta organización, se almacenan en un vector los t_{0x} de todos los rayos, en otro vector los t_{0y} y así sucesivamente. Esta estructuración es menos natural que la clásica "vector de estructuras" que se muestra en la figura 2, pero es necesaria para alcanzar la máxima eficiencia con SSE.

En dichas figuras, $t_{xi}^{(j)}$ representa el valor de la componente x del parámetro t para el rayo j-ésimo. i vale 0 para los parámetros de entrada y 1 para los de salida. Lo mismo se aplica para las componentes y, z.

Un problema que presenta es que para recuperar los distintos parámetros de un rayo, habrá que realizar accesos a posiciones distantes de memoria, especialmente si la lista de rayos es larga. Esto provoca un mal aprovechamiento de la memoria caché. Para solventar el problema, empleamos una organización de memoria mixta entre las dos anteriores, tal y como se aprecia en la figura 3. Los rayos se almacenan en paquetes de cuatro. Cada paquete es una estructura que contiene un vector para las componentes x de los parámetros de entrada de los cuatro rayos, otro para las componentes y, etc.

Si el número de rayos no es múltiplo de cuatro, el último paquete estará incompleto y se rellenará con datos que no serán tenidos en cuenta a la hora de obtener los resultados.

A cada nodo llega una lista de rayos, y debe de generar una nueva lista para cada hijo. Si para cada nodo visitado hubiera que solicitar memoria de forma dinámica para crear las listas de los hijos, el proceso sería ineficiente. Para evitarlo, se emplea una pila de rayos. Al construir el árbol, se reserva memoria para una pila de tamaño suficiente. Para cada nuevo recorrido, se añaden a esa pila todos los rayos a intersectar y se envía al nodo raíz. Las listas de rayos para los sucesivos nodos que se visiten se irán añadiendo y retirando del tope de la pila según sea necesario, eliminando la necesidad de ir reservando memoria en cada paso.

Cada nodo recibe en el tope de la pila los rayos que le intersectan y que debe de procesar. Calcula cuáles de esos rayos intersectan con su primer hijo, y los añade al final de la pila. A dicho hijo se le suministran estos nuevos rayos recién apilados. Al terminar el procesamiento del primer nodo hijo, se desapilan los rayos que le fueron enviados, y se procede de forma análoga con el segundo hijo. De esta forma, en cada momento solo se necesita tener almacenado en la pila los rayos que intersectan a uno de los hijos.

3.4. Cómputo rayo-triángulo

Para las intersecciones rayo-triángulo, hemos usado dos variantes:

- El algoritmo de Moller-Trumbore [7] estándar, realizando un test de 1 rayo con 1 triángulo.
- Una variante del anterior, optimizada, con las normales y diversos resultados intermedios precalculados, con los datos de 4 triángulos empaquetados, y haciendo simultáneamente la intersección de 1 rayo con 4 triángulos con SSE.

Se obtiene mejores resultados (en cuanto a tiempo y uso de memoria) con la segunda opción, (que requiere árboles menos profundos),

Figura 4: *Stanford Bunny*

Figura 5: *Dragon*

Figura 6: *Happy Buddha*

y por tanto dicha segunda opción es la que hemos usado en las pruebas.

4. Rendimiento

Para estudiar la eficiencia de este algoritmo, hemos empleado tres diferentes escenas: *Stanford Bunny* (69451 triángulos), *Dragon* (0,8 millones de triángulos) y *Happy Buddha* (1,1 millones de triángulos). Las escenas empleadas han sido las mismas para todos los algoritmos.

Para comparar el algoritmo propuesto, se ha simulado el comportamiento del algoritmo de Wald ([14, 13]) empleando nuestro propio algoritmo pero limitando el número máximo de rayos que pueden recorrerse juntos a cuatro. De esta forma, se necesita recorrer el índice espacial repetidas veces con paquetes de cuatro rayos.También se compara el algoritmo propuesto con un algoritmo recursivo clásico que recorre los rayos por la estructura de uno en uno.

Se han obtenido los resultados para una profundidad máxima permitida del árbol de indexación espacial de 16, 18, 20, 22 y 24 y se han seleccionado los mejores resultados para cada algoritmo y escena. Para la mayoría de los casos, la profundidad de 20 ha demostrado ser la más óptima. Todas las mediciones han sido tomadas en un Pentium-M 1.8 GHz con 512 MB de RAM.

4.1. Comparación para rayos coherentes

Para cada caso, se han generado una animación de cien imágenes con la cámara rotando alrededor de la escena y se han tomado los resultados medios. Al tener los rayos el mismo punto de origen y atravesar píxeles consecutivos del plano de la imagen, los rayos generados presentan alta coherencia entre sí.

El cuadro 1 muestra los resultados medios obtenidos en MegaRayos por segundo para la visualización de las diferentes escenas por los tres distintos algoritmos. La figura 7 expresa estos datos de forma gráfica.

De igual forma, el cuadro 2 muestra los resultados medios obtenidos en animaciones por segundo para la visualización de las diferentes escenas por los tres distintos algoritmos. La figura 8 expresa estos datos de forma gráfica.

Como puede apreciarse, el algoritmo clásico resulta menos eficaz en todas las situaciones. Los otros dos algoritmos le superan ampliamente al ser capaces de paralelizar los cálculos. Entre estos dos algoritmos, nuestra propuesta consigue resultados ligeramente superiores a la versión que recorre la estructura con paquetes de cuatro rayos. Concretamente para la escena *Stanford Bunny* conseguimos alcanzar una

	n-Rayos	4-Rayos	1-Rayo
Bunny	0,72	0,58	0,24
Dragon	0,51	0,42	0,19
Buddha	0,56	0,45	0,23

Cuadro 1: Rendimiento medio en MegaRayos por segundo de los tres diferentes algoritmos a resolución de 256^2 píxeles para visualizar las diferentes escenas.

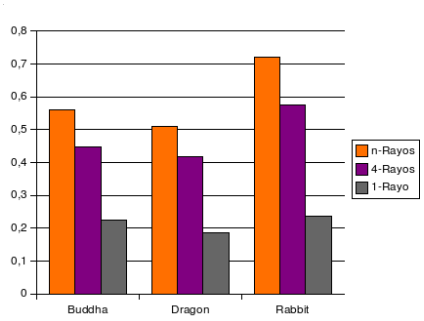

Figura 7: Rendimiento medio en MegaRayos por segundo a resolución de 256^2 píxeles para la visualización de las escenas.

	n-Rayos	4-Rayos	1-Rayo
Bunny	8,31	7,06	3,31
Dragon	6,22	5,34	2,63
Buddha	6,9	5,78	3,17

Cuadro 2: Rendimiento en imágenes por segundo de los tres diferentes algoritmos a resolución de 256^2 píxeles para visualizar las diferentes escenas.

ganancia cercana a 3 con respecto al algoritmo recursivo clásico, siendo 4 la máxima ganancia teórica que podríamos obtener si se paralelizaran todos los cálculos.

Esta ligera mejora de nuestro algoritmo se debe a que la versión que emplea paquetes de 4 rayos, al ser éstos coherentes, consigue paralelizar casi siempre su cómputo. Si bien, ésto no se garantiza que siempre ocurra y en ocasiones se ve obligado a partir dicho paquete, reduciendo el grado de paralelismo. Nuestro algoritmo consigue extraer para cada nodo todo el volumen de datos que ha de procesarse, por lo que las posibilidades de tener que trabajar con paquetes de tamaño inferior a cuatro son menores.

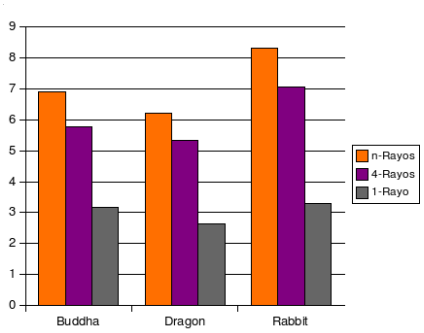

Figura 8: Rendimiento en imágenes por segundo a resolución de 256^2 píxeles para la visualización de las escenas.

4.2. Comparación para rayos no coherentes

En este artículo se ha dicho que la principal ventaja que se obtendría de recorrer el árbol con todos los rayos simultáneamente se produce cuando se trabaja con rayos no coherentes. Éstos se irían clasificando automáticamente al ir descendiendo por el árbol de indexación y las posibilidades de tener varios rayos en un mismo nodo (y así paralelizar sus cálculos) son mucho mayores, especialmente en nodos cercanos a la raíz.

Para demostrar ésto se ha generado una distribución uniforme para lanzar rayos aleatorios sin que sean más densos en unas regiones que otras. Para cada rayo a generar, se toman dos puntos con probabilidad uniforme respecto al área de la esfera envolvente mínima de la escena [11, 12]. Un punto se emplea como punto origen del rayo y el otro para generar un vector director.

El cuadro 3 muestra los resultados medios obtenidos en MegaRayos por segundo para el cómputo de rayos no coherentes que atraviesen las diferentes escenas por los tres distintos algoritmos. La figura 9 expresa estos datos de

forma gráfica.

	n-Rayos	4-Rayos	1-Rayo
Bunny	0,39	0,18	0,13
Dragon	0,28	0,15	0,11
Buddha	0,27	0,15	0,11

Cuadro 3: Rendimiento medio en MegaRayos por segundo para los tres diferentes algoritmos. Los rayos no son coherentes.

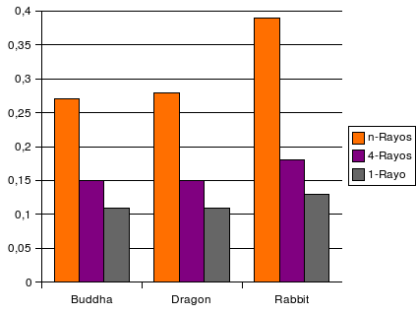

Figura 9: Rendimiento medio en MegaRayos por segundo para rayos no coherentes.

Como puede apreciarse claramente, los resultados obtenidos por la versión que recorre la estructura con paquetes de cuatro rayos han descendido notablemente en comparación con los obtenidos con rayos coherentes. Ofrece resultados muy similares a los obtenidos por el algoritmo recursivo clásico. La razón de este degradamiento en la eficiencia se debe a que al coger paquetes de cuatro rayos no coherentes, las posibilidades de que el paquete se rompa en las partes altas del árbol son muy elevadas, y el algoritmo degenera en el algoritmo convencional no paralelo.

Frente a ésto, nuestro algoritmo sigue ofreciendo una ganancia muy elevada frente a la versión clásica del algoritmo. Los rayos se van clasificando al descender por el árbol, y las posibilidades de paralelizar cómputos son mucho mayores al haber mayor cantidad de rayos con los que trabajar.

Los resultados obtenidos con rayos coherentes eran similares entre nuestra propuesta y la de Wald, pero con rayos no coherentes los resultados se distancian notablemente. Esto demuestra la mayor capacidad de nuestro algoritmo de encontrar paralelismo en el cómputo de los rayos en todo tipo de situaciones.

5. Conclusiones y trabajo futuro

Hasta ahora, las técnicas de aceleración de ray-tracing resultaban eficaces tan solo cuando era fácil encontrar rayos coherentes a los que poder procesar en paralelo, por ejemplo, en rayos primarios. En este artículo hemos mostrado una algoritmo de trazado de rayos que en vez de recorrer el árbol de indexación espacial de la escena una vez por rayo o por paquete de rayos, lo que hace es recorrer el árbol tan solo una vez, pero con todos los rayos simultáneamente (o hasta un máximo de ocho veces si los rayos presentan vectores directores de distinto signo).

Se ha mostrado que esta técnica presenta los siguientes beneficios:

- Principalmente, permite extraer un mayor grado de paralelización en el procesamiento de los rayos. Ello ocurre aún en el caso de rayos no coherentes, pues éstos se irán clasificando al recorrer el árbol, y basta con que dos o más rayos visiten un mismo nodo para que se detecte y se pueda paralelizar su cómputo. Esto se hace especialmente evidente sobre todo en los nodos más elevados del árbol, que serán visitados por una gran cantidad de rayos.

- Además, se visitará un nodo como máximo ocho veces, y no decenas de miles de veces como ocurre con el resto de algoritmos. De esta forma, se minimiza el número de accesos a memoria que hace falta para cargar los nodos del árbol y los triángulos que contienen.

Como trabajo futuro, el esquema de la pila de rayos no resulta eficiente cuando el número de rayos que llega a un nodo es igual o inferior a cuatro. Esto puede ocurrir en nodos cercanos a las hojas. Un algoritmo que combine el uso

de la pila de rayos cuando el número de éstos sea mayor a cuatro, y que no la emplee cuando sea inferior o igual a cuatro podría solventar este problema.

Por otro lado, este algoritmo persigue aumentar la vectorización en el procesamiento de los rayos. Se consigue extraer para cada nodo del índice espacial todo el volumen de datos que ha de procesarse, pero las operaciones deben efectuarse de 4 en 4 que es lo máximo que permite vectorizar SSE. Es de esperar que con arquitecturas que sean capaces de vectorizar mayor cantidad de operaciones, la ganancia de rendimiento sea aún mayor ([2]).

Las GPUs son procesadores especializados en este tipo de operaciones vectoriales, por lo que este algoritmo resulta idóneo para ser implementado en GPUs y poder aprovechar su mayor capacidad de procesamiento en paralelo.

Agradecimientos

La realización de este artículo ha sido llevada a cabo parcialmente con financiación proveniente del proyecto TIN2004-07672-C03-02 del Ministerio de Educación y Ciencia, y por la Junta de Andalucía y la Unión Europea (FEDER) con el proyecto P06-TIC-01403.

Referencias

[1] A. Appel. Some Techniques for Shading Machine Renderings of Solids. SJCC, pages 27-45, 1968.

[2] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich. Ray Tracing on the CELL processor. Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, 2006.

[3] N. Carr, J. Hall and J. Hart. The ray engine. In Proceedings of the ACM Siggraph/Eurographics conference on Graphics hardware, 37-46. Eurographics Association, 2002. ISBN: 1-58113-580-7.

[4] M .Christen: Ray Tracing on GPU. Diploma thesis, University of Applied Sciences Basel, Switzerland, 2005.

[5] T. Foley and J. Sugerman. Kd-tree Acceleration Structures for a GPU Raytracer. In Proc. Graphics Hardware, 15-22, 2005.

[6] A. Klimovitski. Using SSE and SSE2: Misconceptions and Reality. Intel DeveloperUPDATEMagazine, 2001. http://www.intel.com/technology/ magazine/computing/sw03011.pdf.

[7] T. Möller and B. Trumbore. Fast, Minimum Storage Ray-Triangle Intersection. Journal of Graphics Tools, 2(1):21-28, 1997.

[8] T. Purcell, I. Buck, W. Mark, and P. Hanrahan. Ray Tracing on Programmable Graphics Hardware. ACM Transactions on Graphics, 21(3):703-712, 2002. (Proceedings of SIGGRAPH 2002).

[9] T. Purcell. Ray Tracing on a Stream Processor. PhD Thesis, 2004 UMI Order Number: AAI3128683.

[10] J. Revelles, C. Ureña and M. Lastra. An Efficient Parametric Algorithm for Octree Traversal. Proc. WSCG, pp. 212-219, 2000.

[11] L. A. Santaló. Integral Geometry and Geometric Probability. Addison-Wesley, Reading (MA), USA, 1976.

[12] M. Sbert. An integral geometry based method for fast formfactor computation. Computer Graphics Forum (Proceedings of Eurographics'93), 12(3):409-420, 1993.

[13] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive Rendering with Coherent Ray Tracing. Computer Graphics Forum, 20(3):153-164, 2001. (Proceedings of Eurographics).

[14] I. Wald. Realtime Ray Tracing and Interactive Global Illumination. PhD thesis, Saarland University, 2004.

[15] Henrik Wann Jensen. Global Illumination using Photon Maps. In Rendering Techniques '96. Eds. X. Pueyo and P. Schröder. Springer-Verlag, pp. 21-30, 1996