CEIG'08, Barcelona, Sept. 3–5 (2008)
L. Matey and J. C. Torres (Editors)

Estimación de Densidades usando GPUs

M. Lastra[1], C. Ureña[1], J. Bittner[2], R. García[1], R. Montes[1]

[1] Universidad de Granada
[2] Czech Technical University in Prague

Abstract

Este artículo presenta un método de estimación de densidades basado en rayos que ha sido implementado completamente en la GPU. La estimación de densidades basada en rayos reduce el sesgo de los enfoques clásicos basados en fotones aunque tiene un costo computacional más alto. Proponemos algoritmos y estructuras de datos para la implementación en GPUs de la estimación de densidades basada en rayos y mostramos que la técnica propuesta da hasta un orden de magnitud de aceleración comparada con su variante CPU. La aceleración obtenida se incrementa al aumentar el número de rayos y por tanto el método es muy útil en aplicaciones que requieran renderización de alta calidad.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

1. Introducción

La generación de imágenes realistas incluye el cómputo de la irradiancia [Kaj86] en determinados puntos de la escena que se está renderizando. Una opción es emular de forma estocástica (usando técnicas de Monte Carlo) las trayectorias de un enorme número de fotones (*simulación de fotones*), y tras ello reconstruir la irradiancia en determinadas posiciones o regiones de la superficie, usando la información de distribución de los fotones producida durante la simulación.

Aunque se usó previamente en física computacional, la simulación de fotones y la estimación de la irradiancia para regiones de superficies fijas fue descrita por primera vez en Informática Gráfica por Arvo [Arv86], y la técnica se mejoró posteriormente usando tamaños de region adaptativos por Heckbert et al. [Hec90]. Estas técnicas se apoyan en particionar la escena en regiones o parches, que se usan para sumar la energía que llevan los fotones que impactan en cada región, para estimar la irradiancia promedio.

Como la partición de la escena en regiones tiene algunas limitaciones, más tarde fueron propuestas técnicas de estimación de densidades para reconstruir la irradiancia a partir de impactos de fotones en las superficies [SWH*95], [Jen96]. Estas técnicas aplican una función kernel a un número de muestras de energía entrantes *en las proximidades de* un punto donde se debe computar la irradiancia para obtener el estimador de la irradiancia.

En este trabajo presentamos una implementación basada en GPUs de un algoritmo de estimación de densidades que tiene claramente mayor rendimiento que la implementación equivalente en CPU y que por tanto permite generar los resultados deseados más rápido. El modelo computacional de la GPU requiere un diseño específico del flujo de trabajo del algoritmo y estructuras de datos específicas. Este artículo contiene una descripción detallada de las estructuras de datos y algoritmos que permiten que la estimación de densidades basada en rayos sea procesada por completo en la GPU.

El artículo se estructura como sigue: En la sección 2 revisamos el trabajo relacionado. En la sección 3 describimos la implementación en GPU de la estimación de densidades basada en rayos. En la sección 4 tratamos la reducción de flujos no uniformes que forma el núcleo del método propuesto. La sección 5 presenta la evaluación experimental del método y una comparación con su implementación CPU.

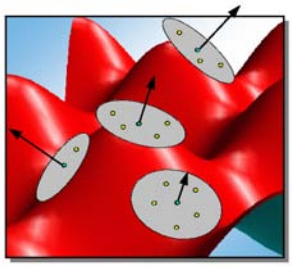

Figura 1: *Estimación de densidades en el plano tangente*

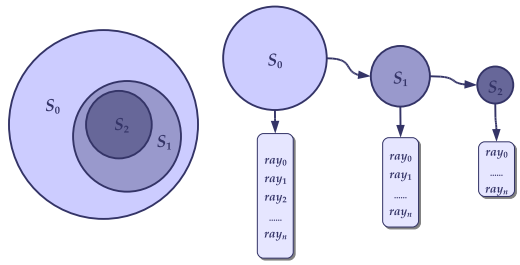

Figura 2: *Algoritmo de la caché de esferas: cada esfera (S_i) de la lista tiene un puntero a la siguiente (S_{i+1}), si existe, y a la lista de rayos que la intersecan. Si el disco actual no está dentro de la última esfera, parte de la lista se destruye y reconstruye para asegurarse de que la última esfera contiene el disco.*

2. Trabajo relacionado

2.1. Cómputo de la irradiancia usando estimación de densidades

La estimación de densidades es una técnica muy utilizada en los algoritmos de iluminación global basados en mapas de fotones [Jen96, Jen01]. Los mapas de fotones almacenan el flujo de luz en las superficies de los objetos. Cada punto del mapa de fotones representa un fotón que lleva una determinada energía. La iluminación de la superficies de los objetos se reconstruye posteriormente a partir de los impactos de los fotones usando estimación de densidades.

El enfoque de mapas de fotones clásico usa estimación de densidades de los fotones almacenados en las superficies. Se puede llevar a cabo un cómputo de la irradiancia más preciso usando un método conocido como *Estimación de Densidades en el Plano Tangente* (DETP) [LURM02a, LURM02b]. En lugar de impactos de fotones, este método usa las trayectorias de los fotones (rayos) generados por una fase de simulación de fotones y un disco centrado en el punto donde se necesita computar la irradiancia. La energía llevada por los rayos que intersecan el disco se usa para obtener un estimador de la irradiancia en ese punto.

Sea $P = \{1 \dots m\}$ el conjunto de índices de todos los rayos generados durante la fase de simulación de fotones. El conjunto $I_r(x) \subseteq P$ se define como el conjunto de índices de todos los rayos que intersecan el disco centrado en x con radio r. El valor ϕ_i es la energía llevada por el rayo i. Usando estas definiciones, la expresión de la irradiancia estimada es:

$$E(x) \approx \frac{1}{\pi r^2} \sum_{i \in I_r(x)} \phi_i \qquad (1)$$

La figura 1 muestra una interpretación visual del algoritmo. Métodos similares para estimación de densidades basados en rayos, que usan funciones kernel ligeramente diferentes, han sido propuestas por Havran et al. [HBHS05] y Herzog et al. [HHK*07].

2.2. Cómputo de las intersecciones kernel-rayo

DETP evita varios problemas inherentes a los mapas de fotones usando rayos en lugar de impactos de fotones, pero por otra parte, introduce una sobrecarga de computación porque requiere el cálculo de una gran cantidad de intersecciones rayo-disco. El número de intersecciones rayo-disco debe reducirse usando una estructura de datos eficiente que indexe los rayos y/o los discos.

Una posibilidad es usar una estructura de datos similar a una cache llamada la *caché de esferas* [LURM02b]. La caché de esferas consiste de una lista dinámica de esferas. Cada esfera contiene una lista de rayos que intersecan la esfera. Las esferas se numeran, comenzando en 1. El centro de la esfera i-ésima se llama c_i, y su radio es r_i. La primera esfera (de índice 1) tiene su centro c_1 en el centro de la caja englobante de la escena. El radio r_1 se elige de modo que cualquier disco esté completamente contenido en la primera esfera. La lista asociada con la primera esfera contiene todos los rayos que la intersecan, es decir, la lista completa de rayos a procesar. La longitud de la lista y las esferas que contiene cambian bajo demanda durante el procesamiento de los discos.

Cuando se procesa un nuevo disco, se comprueba si está incluido en la esfera n-ésima. En caso afirmativo, la lista de esferas no se actualiza y la lista de rayos candidatos es exactamente la lista de rayos que intersecan la última esfera. Esta lista ya ha sido precalculada y se devuelve para que se realize explícitamente un test rayo-disco.

En caso de que el disco no esté incluido en la última esfera, se calcula el máximo valor entero i tal que el disco esté incluído en la esfera i-ésima (i es al menos 1). En ese momento todas las esferas a partir de ésta se borran, y n se hace igual a i. Esta situación se llama un *fallo de caché* en el nivel i, porque significa que hay un disco que no está incluído en los últimos niveles de la estructura de caché y que estos niveles deben ser reconstruídos. La ilustración visual de la caché de esferas se muestra en la figura 2.

Otro enfoque para reducir el número de intersecciones rayo-disco usa *mapas de rayos*, i.e. una subdivisión es-

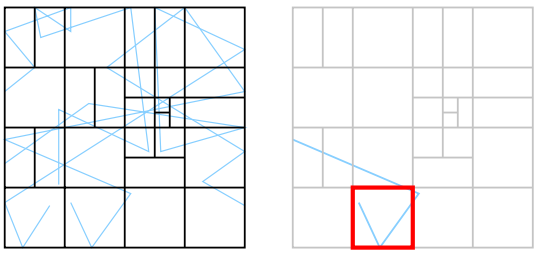

Figura 3: *Ilustración del mapa de rayos. El mapa de rayos consiste en un kD-tree en el que cada celda hoja se asocia con un conjunto de rayos como se muestra a la derecha.*

pacial que indexe los rayos que intersecan las celdas de la subdivisión. Esta técnica fue propuesta por Havran et al. [HBHS05], donde se usó un kD-tree construído bajo demanda para indexar los rayos. El método funciona de la siguiente forma: cuando se procesa un disco, primero se localizan las celdas de la subdivisión que intersecan el disco. Las intersecciones de los rayos asociadas con estas celdas y el disco se calculan. Si una celda contiene más rayos que un umbral dado, la celda se subdivide recursivamente y los rayos se asocian con los hijos de las celdas que se han subdividido.

Para limitar los requerimientos de memoria de este método Havran et al. [HBHS05] propusieron colapsar las partes no usadas del kD-tree si se llega a un límite de memoria dado. Una ilustración del mapa de rayos se muestra en la figura 3.

Nuestro método usa una técnica similar para construir un índice espacial de rayos en la GPU, pero en lugar de un kD-tree usamos un enfoque similar a un octree. Adicionalmente, suponemos que todos los discos son conocidos a priori y realizamos un indexado de los discos junto con el de los rayos usando la misma estructura de datos.

3. Implementación de DETP en la GPU

Como se mencionó antes, el núcleo del algoritmo DETP es el cómputo de una serie de intersecciones rayo-disco. Este cómputo puede ser implementado en la GPU ya que incluye una gran cantidad de operaciones de coma flotante que se pueden llevar a cabo en paralelo. Portar el algoritmo de una arquitectura de propósito general a una plataforma especializada requiere varias adaptaciones.

En esta sección presentamos primero un esbozo del algoritmo. Cada uno de los pasos individuales se explican con detalle posteriormente en esta sección.

3.1. Resumen del algoritmo

El diseño del algoritmo propuesto se guía por la necesidad de proporcionar a la GPU conjuntos de datos grandes para que realice cómputo con una intensidad aritmética alta. También reducimos las transferencias CPU-GPU a sólo aquellas que se requieren para enviar los datos iniciales a la GPU y para devolver los resultados finales.

El algoritmo funciona procesando recursivamente un octree que en el nodo raíz contiene todos los rayos y los discos a procesar. Usamos una estructura octree en lugar de una lista de esferas porque los voxels del octree tienen posiciones fijadas a priori (en el método de caché de esferas, las posiciones de las esferas se generan bajo demanda cuando se procesa la lista de discos). Esto facilita el diseño de un algoritmo que pueda beneficiarse de las capacidades de procesamiento paralelo de las GPUs (se dan más detalles en la sección 3.3). Para reducir la cantidad de memoria GPU que se usa para texturas, no almacenamos el octree completo en la memoria; sólo el conjunto de cajas de la lista desde la raíz hasta la que se está procesando.

El proceso comienza con una llamada al procedimiento recursivo principal para el nodo raíz, con la caja envolvente de la escena como caja asociada. Este procedimiento recursivo principal acepta dos parámetros: una referencia al nodo padre, P (que es nulo para el nodo raíz), y la caja envolvente del nodo actual, B (que es la caja envolvente de la escena para el nodo raíz). Este procedimiento puede resumirse de la siguiente forma:

1. Crear un nuevo nodo actual N, asignar B a su caja envolvente, e insertar en él una referencia a un nuevo vector vacío de índices de rayos R, y un nuevo vector de índices de discos D
2. Si P es nulo entonces:
   - insertar en R los índices de todos los rayos a procesar
   - insertar en D los índices de todos los discos a procesar

   en caso contrario, P es una referencia válida al nodo padre con un vector de índices a rayos (R_P) y un vector de índices a discos D_P, procedemos a:
   - insertar en R los índices de los rayos de R_P que intersequen la caja B.
   - insertar en D los índices de los discos de D_P que intersequen la caja B.
3. Si N es un nodo hoja, entonces:
   - Intersecar todos los discos de D con todos los rayos de R.
   - Distribuir las energías de los rayos: para cada rayo r que interseque a un disco d, añadir la energía de r a la energía de d en la textura resultante.

   en caso contrario, para i con los valores enteros desde 0 hasta 7, hacer:
   - Sea B_i la caja envolvente del i-ésimo nodo hijo de N
   - Llamar recursivamente a este procedimiento con B_i como caja envolvente y una referencia a N como caja padre

4. Borrar el nodo actual *N* (liberar la memoria usada para *N*, *R* y *D*)

3.2. Almacenamiento de rayos y discos

Las estructuras de datos principales para almacenar datos en la memoria de la GPU son *flujos*. En este contexto, un flujo es una textura, i.e., un array bidimensional de tuplas, cada tupla con cuatro valores de coma flotante (usando el formato IEEE 754 de simple precisión). Las entradas pueden indexarse por dos valores enteros (número de fila y columna), o usando dos flotantes sin parte fraccionaria (estos valores normalmente deben de normalizarse para acceder al flujo). Podemos ver también el flujo como un vector de una dimensión o una lista de tuplas.

Los rayos y los discos se almacenan en flujos en la GPU y estos elementos se transfieren sólo una vez para evitar sobrecargas innecesarias. Los rayos se representan por su origen, su vector director y su energía. Se necesita almacenar 9 elementos de coma flotante. Hemos decidido almacenar los orígenes, los vectores directores y las energías de los rayos en tres texturas diferentes.

Los discos se definen por su centro, vector normal y radio. En este caso, todos los centros se almacenan en una textura y las normales (tres números de coma flotante) y los radios (un número de coma flotante) se empaquetan en otra textura.

Durante el cómputo, hay que crear los subconjuntos de rayos y discos. Para evitar tener copias de los elementos que definen los discos y los rayos, se usan índices a los rayos y los discos. Hay que crear texturas de índices en la que cada elemento contiene las coordenadas 2D de cada disco, y lo mismo debe hacerse para los rayos (véase la figura 4). Durante el proceso, cuando se crean subconjuntos de rayos o de discos, se usan subconjuntos de estas texturas de índices.

Esto obviamente añade una pequeña sobrecarga a la operación de acceso a los rayos o los discos, ya que antes hay que acceder a la textura de índices para obtener las coordenadas 2D y se necesita un acceso a textura adicional para acceder al dato final de rayo o de disco.

3.3. Implementación de un índice espacial en la GPU

Portar directamente el índice espacial de la caché de esferas (que se explica en la sección 2.2) a la GPU presenta algunos problemas debido a su naturaleza dinámica. Mantener la lista de esferas actualizada en la memoria de la tarjeta gráfica es una tarea mucho más compleja que en memoria principal. Para evitar estos problemas elegimos finalmente usar una estructura octree clásica. El octree da una reducción equivalente en el número de operaciones de intersección realizadas pero se crea estáticamente y no requiere actualizaciones una vez creado.

El espacio 3D de la escena a renderizar se divide por tanto

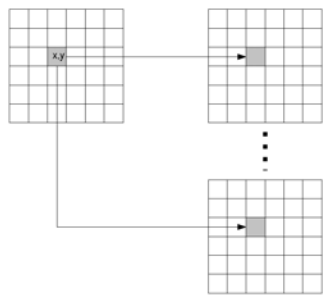

Figura 4: *Una textura de índices se usa para referenciar los rayos y los discos*

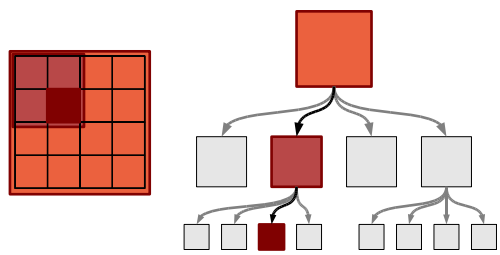

Figura 5: *Árbol de indexación espacial para la versión GPU. Cuando se visita un nodo hoja, sólo sus nodos padre se almacenan en la memoria de la GPU (recuadros con sombra en la figura). Para cada caja, se almacena un par de arrays de índices (uno para los rayos y uno para los discos).*

de forma jerárquica. El nodo raíz del octree se asocia a la caja envolvente de la escena y este espacio se subdivide en ocho nodos. Cada uno de estos nodos se subdivide de la misma forma. Cada nodo guarda información acerca de los discos y los rayos contenidos en él completa o parcialmente. La construcción del octree se detiene cuando se llega a una profundidad del octree determinada, o si el número de rayos o de discos es menor que un umbral predefinido. Cada uno de los nodos hoja o terminales se procesa computanto el test de intersección usando los discos y rayos de ese nodo (véase la sección 3.5).

El octree se crea y se procesa usando una estrategia primero en profundidad. Obtener primero todos los nodos hojas y procesarlos todos *de golpe* sería el enfoque más eficiente; sin embargo, como los datos están almacenados en texturas, no siempre es posible mantener todas las texturas almacenadas al mismo tiempo. Debido a esta limitación, los nodos hoja se procesan justo tras ser creados (antes de que se cree el siguiente nodo hoja). Con este enfoque, cuando se procesa un nodo hoja, sólo están almacenados en la memoria de la GPU su caja y la de su nodo padre (véase la figura 5).

El cómputo de qué rayos y discos están dentro de la caja envolvente de cada nodo del octree se hace en la GPU. La

computación en sí es trivial y sólo el hecho de que el conjunto de entrada debe ser reducido teniendo en cuenta los resultados de la intersección requieren algo de esfuerzo extra.

El proceso es el siguiente. Como se explicó antes, los rayos y los discos contenidos en una caja envolvente se representan por dos texturas de índices. Cada elemento referenciado por esas texturas se interseca con la caja envolvente del *siguiente nivel* (los discos y los rayos se procesan en distintos pasos). La textura resultante de este proceso de intersección es otra textura del mismo tamaño de la entrada, donde un valor de -1.0 representa que no hay intersección, y un valor de 1.0 representa que se ha encontrado un punto de intersección o que el elemento está dentro de la caja envolvente.

3.4. Proceso de reducción Rayo-Disco

Como se mencionó en el resumen del algoritmo, cuando se procesa un nodo del octree es necesario filtrar las listas de rayos y de discos del nodo padre. Esto implica la creación de nuevos flujos con los índices de los rayos y discos que intersecan la caja envolvente de ese nodo, usando como entrada los flujos de índices de rayos y de discos que ya se han obtenido del nodo padre. La operación puede realizarse en dos pasadas: primero, se realizan todos los cómputos de intersecciones; posteriormente los flujos de entrada se filtran y se crean nuevos flujos con sólo los elementos del flujo original que intersecan.

Debido a las características de procesamiento paralelo de la GPU, la operación de filtrado (o empaquetado) no se puede implementar directamente en este tipo de hardware como búsquedas simples secuenciales. Por tanto, se han diseñado y descrito en la literatura algoritmos que realizan esta operación. La sección 4 incluye una explicación más detallada de cómo hemos implementado estos algoritmos en nuestro sistema.

3.5. Proceso de intersección Rayo-Disco

El proceso de intersección rayo-disco se realiza en cada nodo terminal del octree. En este punto, los discos y los rayos asociados con el nodo deben intersecarse. Todo este cómputo se realiza en el procesador de fragmentos donde cada fragmento computa la intersección de cada disco con todos los rayos. El procesador de fragmentos realiza un bucle que itera sobre todos los rayos calculando las intersecciones. Existe por tanto una relación de 1 a 1 entre los fragmentos y los discos que se establece en cada fragmento.

Dependiendo del número de rayos, este cómputo no puede realizarse exactamente, como se mencionó antes. El problema es que las GPUs actuales tienen un número limitado de instrucciones que un procesador de fragmentos o de vértices puede ejecutar. Una vez que se llega a este límite el cómputo se detiene y el proceso termina. Para evitar esta limitación, no todos los rayos se procesan de una vez y se ejecutan varios pasos de rendering. En cada paso de rendering cada fragmento computa la intersección de un disco con un subconjunto de los rayos (véase la figura 6). En una tarjeta Nvidia 7800GTX, el número máximo de rayos que se puede procesar en una pasada es 50×50. En una tarjeta Nvidia 8800GTX, este número se podía incrementar pero no había diferencia en el rendimiento obtenido a pesar de aumentar aun más la intensidad aritmética.

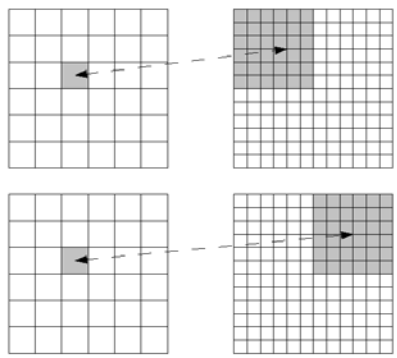

Figura 6: *Proceso de intersección Rayo-Disco. Cada fragmento representa la intersección de un disco y un subconjunto del conjunto de rayos*

3.6. Distribución de la energía

Cuando un nodo hoja ha sido procesado, para cada disco contenido en ese nodo y representado por sus coordenadas en las texturas correspondientes, se obtiene un valor de energía RGB. Estos valores de energía también se almacenan en una textura. Todos estos valores de energía deben transferirse a otra textura que contiene el valor de irradiancia de todos los discos y que será transferida de vuelta a memoria principal como resultado del proceso completo.

Así que los valores de energía obtenidos al procesar un nodo deben sumarse en las posiciones adecuadas en la textura de energía con el resultado final. La posición correspondiente a cada valor de energía se determina por el índice de cada disco. Esta es por tanto una operación de *scattering* típica en la que hay que distribuir una serie de valores de energía hacia la textura resultado.

Creamos un procesador de vértices para realizar esta operación de scattering en la GPU. Este procesador de vértices recibe una textura de índices de discos y una textura de energías (ambas del mismo tamaño). Para cada elemento de la textura, se crea y se procesa un vértice. En cada vértice, se lee el índice correspondiente y se envía la energía a un frag-

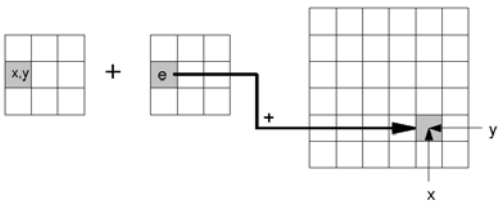

Figura 7: *Distribución de la energía*

mento con las mismas coordenadas del disco. El resultado se acumula en la textura final de energías. Véase la figura 7.

Desde la introducción de la arquitectura unificada de Nvidia, tanto los procesadores de vértices como los de fragmentos se ejecutan en los mismos procesadores. Por tanto el incremento de costo por usar un procesador de vértices en tarjetas de generaciones anteriores (que solían tener menos procesadores de vértices que de fragmentos) ya no es un problema. La simplicidad del algoritmo en el procesador de vértices es un beneficio añadido.

4. Reducción no uniforme de flujos rayo/disco

Como se mencionó antes, la característica principal de este método es que permite reducir el número total de intersecciones rayo-disco calculadas, descartando grandes conjuntos de rayos y de discos mediante el uso de la lista de esferas envolventes en la CPU o el octree de cajas en la GPU. Esto implica que hay que manejar arrays (texturas o flujos) que contienen datos o bien de rayos o de discos, y que hay que descartar de esos arrays las entradas con índices que apunten a rayos o discos que no intersequen la caja envolvente del nodo actual del octree. Este proceso debe generar nuevos flujos (más pequeños) con datos de rayos o discos que intersequen la caja. Estos datos se añaden al octree como nodos hijos del nodo actual. El proceso se puede llamar filtrado de flujos o empaquetado, aunque en la literatura de la GPU normalmente se le llama *reducción no uniforme de flujos*.

En el modelo de procesado secuencial clásico, basado en CPU, esto se puede llevar a cabo mediante un procesado secuencial de los arrays, simplemente ignorando las entradas no interesantes, y copiando los datos útiles en el array de salida. La complejidad algorítmica es $O(n)$ para un array de entrada con n elementos. No hay operaciones aritméticas, y casi todo el tiempo se emplea en transferencias de datos. Sin embargo, el método produce un patrón de acceso a memoria coherente, y esto asegura constantes ocultas muy pequeñas, lo que a su vez implica que este paso necesita una fracción minúscula del tiempo total de computación, haciendo por tanto a la caché de esferas una técnica eficiente en CPUs.

Nótese que se asume que determinar si un elemento debería filtrarse o no es un proceso que se realiza antes del filtrado en sí; de este modo, cuando comienza el filtrado, a cada entrada de datos no válidos ya se le ha asignado un valor *nulo* especial (hemos elegido para el valor nulo usar una tupla de cuatro elementos -1, ya que -1 nunca es un índice válido), así que una simple comparación con ese valor es suficiente para saber si la entrada debería copiarse en la salida o no.

El algoritmo descrito antes es inherentemente secuencial, porque para elemento de entrada que no se descarta, necesitamos conocer la posición donde guardar ese dato en el flujo de salida. Obviamente, esta posición sólo puede calcularse conociendo cuántas entradas de datos válidas se almacenan en la entrada a la izquierda de la actual. Por tanto, un procesamiento del flujo simple, secuencial y de izquierda a derecha, necesita usar un contador de entradas válidas, y esto impide una paralelización directa del algoritmo, ya que las transferencias de los elementos no pueden realizarse independientemente del resto de los elementos. Además, el algoritmo tiene una intensidad aritmética bastante baja.

Hubo que diseñar un método específico para las implementaciones en GPU, para aprovechar las capacidades paralelas de este hardware. Varios autores han propuesto técnicas de reducción de flujos no uniformes para conseguir esto. En esta línea, la propuesta de Horn y otros [Hor05], usa una técnica que considera un stream como un vector unidimensional. Posteriormente, otros autores han propuesto técnicas relacionadas, más eficientes, ya que consideran los streams como matrices bidimensionales y usan pirámides de imágenes [AG06, ZTTS06]. Nuestra implementación se basa en estas últimas técnicas. Lo que buscamos es un algoritmo que tome un flujo o vector I como entrada, y que genere otro vector O de salida con todas las entradas válidas en I empaquetadas en entradas adyacentes, probablemente junto con algunas entradas no válidas usadas para relleno (dado que en las GPUs los flujos normalmente se almacenan como arrays rectangulares).

Estas técnicas se basan en el cómputo de un vector que guarda la posición donde debería almacenarse cada elemento de la entrada en el flujo de salida. Esto se llama el vector de distribución [Buc05, Hor05]. Este proceso puede interpretarse como el cómputo del contador mencionado antes para cada elemento de entrada, y obtiene un flujo S de valores enteros no decrecientes. El valor $S[i]$ es el contador del número total de entradas válidas en I a la izquierda de i, o lo que es lo mismo, la posición donde $I[i]$ debería almacenarse. Este vector puede usarse para empaquetar las entradas válidas de la forma deseada mediante una búsqueda binaria, en un paso final de cálculo.

La reducción no uniforme se puede llevar a cabo por un algoritmo de tres fases. Asumimos un vector de entrada I con 2^m elementos. En el caso de que I no tenga esta propiedad, I puede incluirse en un flujo mayor que sí tenga esta propiedad rellenando con valores nulos. El proceso completo se describe a continuación:

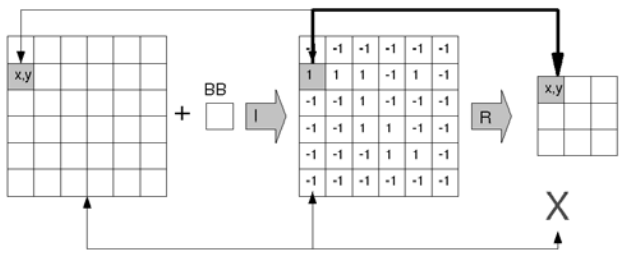

Figura 8: *Proceso de intersección de cajas envolventes con rayos o discos*

1. La primera fase tiene m pasos, donde el paso i-ésimo genera un flujo U_i que, a su vez, es la entrada al paso $i+1$. El flujo U_i tiene 2^{m-i} elementos, con i moviéndose entre 0 y m. La entrada j-ésima del flujo U_0 es 1 si la entrada j-ésima de A es válida, y 0 en caso contrario. A partir del flujo U_i, el flujo U_{i+1} se calcula sumando las entradas adyacentes: $U_{i+1}[j] = U_i[2j] + U_i[2j+1]$. Como consecuencia, cada entrada en U_i guarda el número total de entradas válidas de un bloque de 2^i entradas adyacentes de A. Cuando se calcula U_m, termina esta primera fase, y $U_m[0]$ contiene el contador con el número total de entradas válidas de A.
2. La segunda fase se realiza calculando una serie de vectores D_m,D_{m-1},...,D_0, donde el tamaño de D_i es 2^{m-i}. El vector D_m tiene una única entrada que se inicializa a cero. En cada paso, D_i se calcula a partir de D_{i+1} y U_i, asignando a $D_i[k]$ el valor $D_{i+1}[k/2]$, si k es par, y el valor $D_{i+1}[(k-1)/2] + U_i[(k-1)/2]$, si k es impar. El valor entero $D_i[k]$ es el número total de entradas válidas en las primeras $2^i k$ entradas de I. Como consecuencia, el vector D_0 es exactamente el vector S que queríamos calcular (es el vector de distribución antes mencionado).
3. Los datos de S pueden usarse como entrada para la fase tercera final (que se llama fase de *recolección*). En esta fase, y para cada elemento de salida (de posición l), se hace una búsqueda binaria en S para la última posición p (la más a la derecha) en donde esté almacenado l; por tanto estamos buscando el máximo valor p tal que $S[p] = l$. Si se encuentra alguno, entonces $O[l] = I[p]$, en caso contrario $O[l]$ contiene el valor nulo.

En la Figura 9 se puede observar el diagrama de flujo de datos para las primeras dos fases, en un caso simple donde el vector de entrada sólo tiene cuatro elementos. El proceso puede interpretarse como un esquema jerárquico, donde el algoritmo recorre un árbol binario; la primera fase es un proceso de abajo hacia arriba, mientras que la segunda fase es de arriba abajo.

Cada una de las tres fases puede verse como un proceso iterativo con $\log_2(n)$ pasos básicos, para un vector de entrada de tamaño n. En cada paso básico, el número de entradas

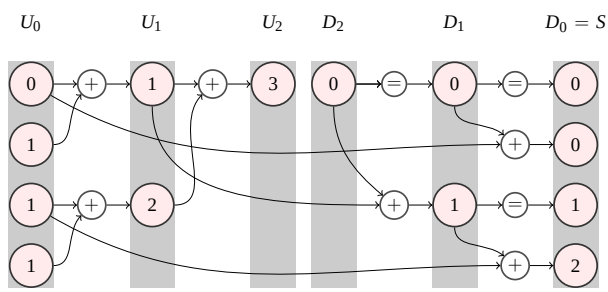

Figura 9: *Diagrama de flujo de datos para un vector de entrada de muestra I con 2^2 entradas; se muestran las primeras dos fases de la reducción de flujos (los vectores se muestran como vectores columna).*

procesadas es proporcional a n. Por tanto la complejidad de la reducción de flujos es $O(n\log_2(n))$.

En nuestra implementación, hemos usado un esquema modificado. La construcción jerárquica en dos pasos de S se hace usando un quadtree en lugar de un árbol binario [AG06, ZTTS06]. Los flujos se procesan como arrays cuadrados 2D, en lugar de como vectores lineales 1D, ya que en la memoria de la GPU un flujo o vector se guarda como una textura 2D. En el primer paso, el algoritmo se parece a la construcción de un mip-map, con cada entrada calculándose como la suma de cuatro entradas. Este esquema encaja mejor con el hardware actual de las GPUs, porque permite aprovechar el hecho de que las GPUs que hemos empleado usan a nivel de hardware tuplas de cuatro flotantes tanto para las operaciones aritméticas (usando sumas y comparaciones SIMD) como para las transferencias desde y hacia la memoria de texturas.

Teniendo en cuenta lo anterior, en nuestro algoritmo usamos tuplas de cuatro elementos para almacenar cuatro entradas en las primeras dos fases, de forma que, en estas fases, todas las operaciones se realizan sobre dichas tuplas, operaciones que por tanto que se realizan simultáneamente, a nivel hardware, sobre cuatro flotantes en paralelo, reduciendo así

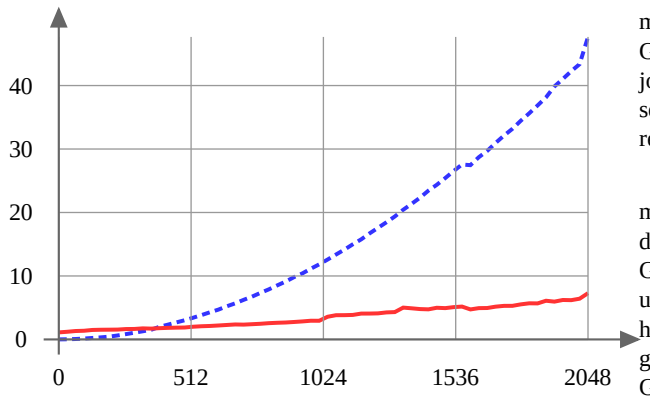

Figura 10: *Tiempo (en milisegundos) empleado en la reducción en la CPU (azul, con guiones) y en la GPU (roja, contínua), en función de la raíz del número de entradas en el flujo.*

tiempo empleado en sumas y comparaciones y otras operaciones ya el hardware de las GPUs está optimizado para trabajar con estas tuplas.

La Figura 10 muestra una comparación del tiempo empleado en la reducción de un flujo para una implementación CPU y la implementación GPU descrita antes. El test se realizó creando un array cuadrado en memoria principal, con n filas y n columnas. Cada entrada (una tupla de cuatro flotantes) se inicializó con un valor válido aleatorio con probabilidad p (con $p \in [0,1]$), y con el valor nulo con probabilidad $1-p$. El valor entero n y la probabilidad p son parámetros del programa de test.

Tras la creación del array de entrada en memoria, ejecutamos la reducción tanto en la CPU como en la GPU. El proceso genera un nuevo array que contiene todos los valores válidos del array de entrada, junto con algunas entradas nulas necesarias para rellenar este nuevo array resultante. Este array es casi cuadrado, en el sentido de que la diferencia entre el número de filas y de columnas es como máximo uno. El programa de la CPU simplemente recorre el array de entrada almacenado en RAM y copia las entradas válidas al array de salida. Lo hace de forma eficiente mediante un puntero de lectura, que recorre el array de entrada, y un puntero de escritura que se mueve por el array de salida.

En el caso del programa CPU, el array de salida está en memoria principal, mientras que para la GPU, se almacena en memoria de la GPU. Tanto para la CPU como para la GPU, calculamos el tiempo real que se gasta en la reducción, incluyendo el tiempo necesario para reservar memoria para el array resultante. Para la versión GPU, no incluímos el tiempo que se gasta en la transferencia inicial de CPU a GPU del array de entrada, porque la reducción se diseñó para usarse como una fase de programas GPU que transformen flujos, que ya están almacenados en la memoria de la GPU. Tampoco incluímos el tiempo empleado en crear flujos auxiliares usados en la reducción en la GPU, ya que esto se hace únicamente una vez para todas las operaciones de reducción, que pueden compartir esos flujos.

La Figura 10 muestra los resultados obtenidos. La gráfica muestra el tiempo de cómputo (en milisegundos) en función de n, la raíz del número de entradas. Como se esperaba, la GPU tiene mayor rendimiento que la CPU para flujos con un tamaño mayor de un umbral, que depende del entorno hardware y software y de la implementación. Para flujos grandes, las capacidades de procesamiento paralelo de las GPUs hacen la implementación GPU más rápida; sin embargo, para flujos pequeños, la inicialización de los procesadores de fragmentos y el tiempo de construcción del flujo de salida domina. Realizamos el test para varios valores de p, la fracción del número de entradas válidas del array de entrada; sin embargo este parámetro no tiene un efecto medible en los tiempos de los programas. El test se ejecutó en un PC con una GPU Nvidia 8800 GTX y una CPU AMD Athlon 64 5200+.

5. Resultados

El esquema de estimación de densidades que ha sido presentado se ha comparado con la versión correspondiente del algoritmo en la CPU. En ambos casos se usó la misma técnica de estimación de densidades (excepto por el tipo de indexación espacial), pero para hacer la comparación completamente justa, también hemos creado una versión CPU basada en SSE del módulo de intesersección rayo-disco. Esto permite usar todo el poder de cómputo de la CPU al usar también las capacidades SIMD de los procesadores de propósito general. Las instrucciones SSE de aritmética en coma flotante se usan para calcular la intersección entre un disco y un paquete de cuatro rayos. Como era de esperar, observamos que esto es más eficiente que una intersección directa entre un único disco y un único rayo.

El entorno de test consiste en un AMD Athlon 64 5200+ CPU (2.6Ghz) y una GPU Nvidia 8800GTX. Es por tanto un entorno en el que se ha usado un sólo núcleo CPU y una única GPU. Aunque actualmente son comunes configuraciones multi-núcleo hay que recordar también que existen configuraciones que permiten el uso de hasta cuatro GPUs en paralelo. El ordenador ejecuta Linux con un kernel 2.6, el compilador usado fue gcc 4.1 y se usaron Nvidia Cg 1.5 y 2.0 para implementar los procesadores de vértices y de fragmentos.

Para obtener los resultados mostrados en la tabla 1 usamos rayos y discos generados aleatoriamente y con distribución uniforme dentro de un cubo de tamaño unidad; el radio de los discos era 0.01.

Hay que tener en cuenta que el tiempo de cálculo empleado en cualquier esquema de indexación espacial depen-

Tabla 1: *Resultados*

#discos × #rayos	GPU	CPU(SSE)	CPU
$10^4 \times 10^4$	0.82s	0.18s	0.22s
$10^5 \times 10^5$	1.9s	8.6s	13s
$10^5 \times (5 \times 10^5)$	7.3s	37.6s	60.1s
$10^5 \times 10^6$	13.2s	88.8s	124.7s

derá de la distribución espacial del conjunto de rayos. En el contexto de los algoritmos de photon-mapping o estimación de densidades, esta distribución depende, a su vez, de la situación y características de las fuentes de luz, de la geometría de la escena y de las propiedades reflectivas de los materiales. Este hecho impediría, en principio, valorar la eficiencia de las indexaciones espaciales de forma independiente de un escenario concreto, y aparentemente haría imposible obtener conclusiones válidas sobre el esquema de indexación más adecuado en general, o los mejores valores de los parámetros involucrados en la construcción. Esto es válido también en nuestro caso, pues las distribuciones de discos y rayos dependen igualmente de esos mismos parámetros (escena, luces y materiales).

Para solventar este problema, se puede estudiar la eficiencia de las indexaciones asumiendo una distribución uniforme de los rayos, lo cual implica que la probabilidad de un rayo cualquiera de intersecar un objeto cerrado es proporcional a su área). Esta distribución uniforme no se da en ningún caso concreto, pero representa bien el caso promedio. Cuando se asume esta distribución de rayos, se puede usar la conocida *heuristica de área de la superficie* para obtener resultados formales sobre la forma óptima de construir las indexaciones espaciales [GS87]. De hecho, experimentos recientes demuestran que, si durante la construcción de indexaciones espaciales se usa SAH teniendo en cuenta distribuciones concretas no uniformes de los rayos, los tiempos de recorrido son solo marginalmente inferiores para raycasting (rayos primarios), o bien incluso superiores para ray tracing [BH07].

Por todo lo anterior, hemos decidido verificar la eficiencia del método propuesto para distribuciones uniformes del conjunto de rayos. Tal verificación podría llevarse a cabo en un escenario concreto, y esta prueba sería ilustrativa, pero entendemos que en ese caso los resultados no serían concluyentes por falta de generalidad.

La tabla 1 y la figura 11 muestran claramente cómo la versión GPU del algoritmo de estimación de densidades tiene más rendimiento que ambas implementaciones CPU (con y sin instrucciones SSE). El uso de las instrucciones SSE por supuesto representa una mejora sobre la versión simple CPU, pero los tiempos en la GPU son de todas formas casi un orden de magnitud más bajos.

Tambien es importante que el uso de la GPU no merece

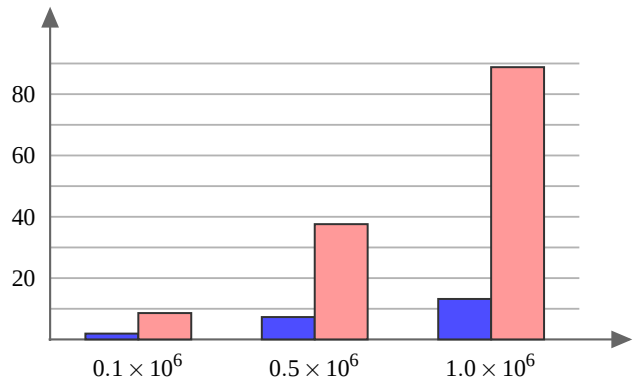

Figura 11: *Tiempo (en segundos) necesario para intersecar 10^5 discos, tanto para la GPU (izquierda) como para la CPU (derecha). El test se ejecutó variando el número de rayos (mostrado bajo cada par de barras). Véase la tabla 1 para los números exactos.*

Tabla 2: *Tiempo requerido por cada fase*

Tiempo total (10^5 discos × 10^6 rayos)	13.2s
Intersecciones Rayo-Nodo octree	53ms
Intersecciones Disco-Nodo octree	17ms
Intersecciones Rayo-Disco	**11.8s**
Filtrado	380ms
Distribución de energía	32ms

la pena en el caso de que el tamaño del problema no sea lo suficientemente grande. Esto puede verse en la primera fila de la tabla 1.

Hemos intentado obtener una primera estimación de cómo de eficientemente se estaba usando la unidad de procesamiento gráfico. Usando las herramientas suministradas por Nvidia realizamos algunos tests haciendo tanto *underclocking* como *overclocking* de la GPU. Hemos notado que los tiempos obtenidos escalan casi linealmente con la velocidad de la GPU. Este hecho muestra que la GPU se está usando eficientemente y que no hay periodos de tiempo grandes en los que la GPU esté sin actividad durante el proceso de cómputo.

Para dar más información sobre el algoritmo presentado, el tiempo requerido por cada una de las fases del proceso de estimación de densidades se muestra en la tabla 2. Esta tabla muestra claramente como el 90% del tiempo se gasta en el proceso de intersecciones rayo-disco y que las otras fases no introducen ninguna sobrecarga.

6. Conclusión

Este artículo introduce un nuevo esquema para implementar la estimación de densidades en la GPU que tiene mejor rendimiento que implementaciones comparables en la CPU. Este trabajo muestra que aplicar adecuadamente técnicas de indexación espacial bien conocidas a la estructura inherentemente paralela de la GPU puede resultar en programas eficientes que aprovechan al máximo este tipo de hardware. La técnica también permite aumentar el uso de técnicas de estimación de densidades avanzadas, porque las hace más competitivas en términos de eficiencia en tiempo en comparación con soluciones estandar.

7. Agradecimientos

Este trabajo ha sido financiado por el proyecto de investigación TIN2004-07672-C03-02 (Comisión Interministerial de Ciencia y Tecnología).

References

[AG06] A. GRESSR. K.: Gpu-based collision detection for deformable parametrized surfaces. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2006) 25*, 3 (2006), 497–506.

[Arv86] ARVO J.: Backward ray-tracing. In *ACM Siggraph Course Notes* (1986), pp. 259–263.

[BH07] BITTNER J., HAVRAN V.: Rdh: Ray distribution heuristics for construction of spatial data structures. In *Proceedings of Symposium on Interactive Ray Tracing 2007 (posters)* (2007).

[Buc05] BUCK I.: *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison Wesley, 2005, ch. Taking the Plunge into GPU Computing, pp. 509–519.

[GS87] GOLDSMITH J., SALMON J.: Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications 7(5)* (1987), 14–20.

[HBHS05] HAVRAN V., BITTNER J., HERZOG R., SEIDEL H.-P.: Ray maps for global illumination. In *Rendering Techniques 2005 (Eurographics Symposium on Rendering)* (Konstanz, Germany, June 2005), Bala K., Dutre P., (Eds.), Eurographics Association, pp. 43–54,311.

[Hec90] HECKBERT P. S.: Adaptive radiosity textures for bidirectional ray tracing. *ACM SIGGRAPH Computer Graphics 24*, 4 (1990), 145–154.

[HHK*07] HERZOG R., HAVRAN V., KINUWAKI S., MYSZKOWSKI K., SEIDEL H.-P.: Global illumination using photon ray splatting. *Computer Graphics Forum (Proceedings of Eurographics 2007)* (2007), 503–513.

[Hor05] HORN D.: *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison Wesley, 2005, ch. Stream Reduction Operations for GPGPU Applications, pp. 573–589.

[Jen96] JENSEN H.: Global illumination using photon maps. In *Rendering Techniques'96* (1996), Pueyo, Schroeder, (Eds.), Springer-Verlag, pp. 21–30.

[Jen01] JENSEN H.: *Realistic Image Synthesis Using Photon Mapping*. AK Peters, 2001.

[Kaj86] KAJIYA J.: The rendering equation. In *Computer Graphics. ACM Siggraph'86 Conference Proceedings* (1986), vol. 20(4), pp. 143–150.

[LURM02a] LASTRA M., UREÑA C., REVELLES J., MONTES R.: A density estimation technique for radiosity. In *1st Ibero-American Symposium in Computer Graphics (SIACG'2002). Guimaraes, Portugal* (2002), pp. 163–172.

[LURM02b] LASTRA M., UREÑA C., REVELLES J., MONTES R.: A particle-path based method for monte-carlo density estimation. In *Poster at: 13th EUROGRAPHICS Workshop on Rendering. Pisa, Italy* (2002), pp. 33–40.

[SWH*95] SHIRLEY P., WADE B., HUBBARD P., ZADESKI D., WALTER B., GREENBERG D.: Global illumination via density estimation. In *Rendering Techniques'95* (1995), Hanrahan, Purgathofer, (Eds.), Springer-Verlag, pp. 219–230.

[ZTTS06] ZIEGLER G., TEVS A., THEOBALT C., SEIDEL H.: On-the-fly point clouds through histogram pyramids. In *Proceedings of Vision, Modeling and Visualization 2006* (11 2006), pp. 137–144.