CEIG - Spanish Computer Graphics Conference (2012)
Isabel Navazo and Gustavo Patow (Editors)

Generating Coherent Ray Directions in Path Tracing

R. Torres[1] and P. J. Martín[2] and A. Gavilanes[3]

Universidad Complutense de Madrid, Madrid, Spain
[1]r.torres@fdi.ucm.es, [2] pjmartin@sip.ucm.es, [3]agav@sip.ucm.es

Abstract

The quality of the images produced by a global illumination rendering engine is highly connected to the number of paths that are randomly generated from the camera to the light sources. Graphics Processing Units (GPUs) have been used to implement renderers by typically binding each thread to a ray. Nevertheless, this purely random generation does not fit well on the architecture of current GPUs, due to their SIMD nature.
We modify the way paths are extended in order to take the most of GPUs. The arrangement of rays is split into groups of contiguous rays. The hemisphere on each intersection point is divided into spherical patches, and the same patch is chosen for all the rays in the group. The next direction for each path will be randomly chosen on that patch. Thus, if the intersection points of a group are near, the new rays spawned will be similar.
We have implemented different configurations of this idea on a path tracer in CUDA and they have been experimentally tested on usual scenes. In the same amount of time, most of our algorithms complete more paths and, therefore, they produce images of higher quality than those obtained by the purely random generation.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing, I.3.1 [Computer Graphics]: Hardware architecture—Parallel processing

1. Introducción

Simular por ordenador el transporte de luz es necesario para generar imágenes fotorrealistas a partir de escenas tridimensionales. Una de las propuestas para simularlo [Vea98] lo resuelve en términos de la integral

$$I = \int_{\bar{x}\in\Omega} f(\bar{x})\, d\bar{x}$$

donde Ω es el conjunto de todas la rutas de longitud finita que comienzan en una luz y terminan en la cámara, y f es la contribución de una ruta. Una ruta de longitud $n \geq 1$ es una sucesión de puntos $\bar{x} = x_0 x_1 \ldots x_n$, donde x_0 es un punto en la superficie de una luz, x_n está en la lente de la cámara y el resto son puntos que están en la superficie de los objetos de la escena. La contribución f de la ruta $\bar{x}$ es

$$f(\bar{x}) = L_e(x_0,x_1)G(x_0,x_1)\prod_{i=1}^{n-1} f_r(x_{i-1},x_i,x_{i+1})G(x_i,x_{i+1})$$

donde L_e es la *radiance* de emisión de la luz, G es el término geométrico y f_r es la BRDF.

Una aproximación de la integral anterior se puede obtener mediante técnicas numéricas conocidas como Monte Carlo. De esa manera, un estimador de la integral anterior es

$$I \approx \frac{1}{N}\sum_{i=1}^{N}\frac{f(\overline{X_i})}{p(\overline{X_i})}$$

donde cada $\overline{X_i}$ es una ruta aleatoria y p es la función de densidad de probabilidades *pdf* de esa ruta.

Uno de los algoritmos usados para generar aleatoriamente rutas es el algoritmo conocido como *path tracing* [Kaj86]. Este algoritmo comienza eligiendo un punto en la lente de la cámara y trazando un rayo hacia la escena. Una vez que se ha encontrado el punto de intersección más cercano para ese rayo, la ruta se extiende eligiendo una nueva dirección aleatoriamente dentro del hemisferio centrado en la normal de la superficie en ese punto. Así se procede hasta que la ruta alcanza una luz o termina mediante ruleta rusa.

Por otro lado, las GPUs han evolucionado hasta convertirse en dispositivos masivamente paralelos con una gran potencia de cálculo. La implementación de un path tracing en GPU puede parecer trivial ya que cada ruta se traza independientemente de las otras. Sin embargo, las GPUs actuales tienen un modelo de ejecución SIMD (SIMT

DOI: 10.2312/LocalChapterEvents/CEIG/CEIG12/081-090

según la terminología de NVidia [NVI11]), lo que significa que existen dependencias entre hilos durante la ejecución. En concreto, peticiones a memoria y divergencias dentro del código pueden obligar a que ciertas operaciones se realicen secuencialmente en vez de en paralelo, lo que implica una disminución del rendimiento.

Según nuestra experiencia, la etapa más lenta del path tracing es la conocida como *traversal*. Esta etapa se encarga de encontrar la intersección más cercana de cada rayo. Si durante el traversal, las direcciones de los rayos son muy *diferentes*, los accesos a memoria estarán muy alejados entre sí. Esto implica un mal uso de las cachés de memoria y un aumento de la cantidad de memoria transferida.

Una forma de solucionar este problema es reordenar estos rayos aleatorios para que aquellos que tengan direcciones similares se ejecuten en hilos adyacentes. En este trabajo vamos a solucionar el problema favoreciendo esto en la medida de lo posible. Los rayos se generan ya agrupados, evitando, por consiguiente, cualquier tipo de sobrecarga relacionada con la ordenación. A esto lo vamos a llamar *generación coherente*. Sin embargo, esta generación conlleva una correlación entre estimadores, lo que puede incrementar el error medio. En este trabajo demostramos que, aunque el error producido por las muestras es mayor, el número de rayos lanzados es suficientemente grande como para que las imágenes resultantes tengan un error menor.

2. Trabajos relacionados

Una de las principales mejoras para implementar ray tracers más rápidos es aprovechar la *coherencia* de un conjunto de rayos. Decimos que un conjunto de rayos es coherente si todos ellos tienen una característica común que beneficia el trazado de estos por la escena. Esta definición, en realidad, depende de los autores que se han ocupado del tema.

Wald et al. [WBWS01] fueron pioneros en el uso de la coherencia para desarrollar un ray tracer interactivo en CPU. En su trabajo, los rayos primarios generados a través de píxeles cercanos se llaman coherentes ya que tienen direcciones parecidas y un origen común. Por tanto, es muy probable que estos rayos recorran los mismos nodos de la estructura de aceleración e intersequen con los mismos triángulos. Para amortizar ciertas operaciones, estos rayos se agrupan en *paquetes*, constituyendo así la nueva unidad de recorrido. El uso de paquetes permite mejorar la eficiencia de las cachés y el uso de las unidades SIMD de las CPUs.

Wald et al. [WKB*02] usan las técnicas de *photon mapping* [Jen96] e *instant radiosity* [Kel97] para implementar un algoritmo de iluminación global en CPU. Los rayos de sombra trazados a los puntos de luz virtuales son también coherentes ya que son generados de manera parecida a los primarios.

Mansson et al. [MMAM07] presentan varias heurísticas geométricas para organizar los nuevos rayos generados en CPU. También en CPU, Noguera et al. [NUG09] presentan un recorrido donde los rayos son clasificados en función del signo de su dirección. Boulos et al. [BEL*07] proponen varias formas de empaquetar rayos secundarios en función de su tipo.

Con la llegada de las GPUs completamente programables, el concepto de paquete de rayos coherentes fue adaptado a las unidades SIMD de anchura 32 de las GPUs. Günther et al. [GPSS07] realizan un recorrido de una BVH usando una pila por paquete, implementada en memoria compartida. Popov et al. [PGSS07] y Horn et al. [HSMH07] proponen recorridos de KD-trees sin pila. Aila y Laine [AL09] demuestran que el recorrido de una BVH usando una pila individual por rayo es más eficiente que el uso de una pila compartida por todo el paquete.

Garanzha y Loop [GL10] empaquetan los rayos usando un criterio geométrico, basado en la dirección y el origen de los rayos. Para acelerar la clasificación, los rayos son previamente transformados en claves *hash* y posteriormente ordenados rápidamente en GPU. Después, el *frustum* de cada paquete se encarga de recorrer la BVH en anchura.

Otra manera de interpretar la coherencia es tener en cuenta la estructura de aceleración durante el recorrido de los rayos. Así, Pharr et al. [PKGH97] describen un renderizador de Monte Carlo usando una rejilla uniforme como estructura de aceleración. Los rayos quedan esperando en los vóxeles de la rejilla, retrasándose la intersección con la geometría contenida. Un planificador se encarga posteriormente de comenzar los tests de intersección de los rayos de una cola contra la geometría de ese vóxel. Este método reduce el tráfico de las cachés de disco. Similarmente, Navratil et al. [NFLM07] presentan otra técnica para decrementar el tráfico entre DRAM y caché L2. En este caso, las colas de rayos están localizadas en algunos nodos del KD-Tree, cuyos subárboles caben en L2. Boulos et al. [BWB08] presentan técnicas de filtrado para compactar aquellos paquetes durante el recorrido. Torres et al. [TMG11] realizan un corte en una BVH y recorren en paralelo cada uno de los subárboles resultantes.

Otros autores proponen técnicas de empaquetado basadas directamente en las operaciones que los rayos demandan. El objetivo de estas propuestas es aprovechar al máximo las instrucciones SIMD. Trabajos como [WGBK07] y [GR08] se encuentran en esta categoría.

Por otro lado, el uso de un path tracing *unbiased* implica que no todos los rayos tienen la misma longitud. Para no desaprovechar recursos de la GPU, Novák et al. [NHD10] implementan un sistema que permite la regeneración de una ruta que ha terminado. Antwerpen [vA11] añade al trabajo anterior el uso de la coherencia de los rayos primarios mediante una compactación de los rayos regenerados.

Hermes et al. [HHGM10] y Hachisuka [Hac05] realizan

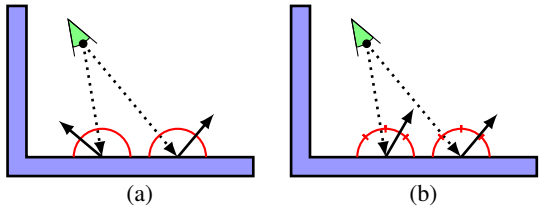

Figure 1: Esquema de generación coherente. Dos rayos (punteados) se generan en la cámara. En (a), los siguientes rayos se generan eligiendo un punto sobre todo el hemisferio. En (b), los siguientes rayos se generan coherentemente. Los hemisferios están divididos en patches y los dos rayos punteados pertenecen al mismo grupo. El siguiente rayo se genera dentro del mismo patch. Los rayos resultantes tienen direcciones parecidas.

un traversal de los rayos cuando todos tienen la misma dirección. Eso les capacita a usar el pipeline gráfico con proyección ortogonal para encontrar el punto de intersección más cercano a cada rayo.

3. Generación de rayos coherentes

Para elegir el siguiente rayo de una ruta, primero se calcula el espacio tangente en el punto de intersección usando la normal de la superficie y un vector no paralelo llamado *up*. Luego, se elige un punto sobre el hemisferio unidad y se transforma a coordenadas globales usando ese espacio tangente. Ese punto se usará para obtener la dirección del siguiente rayo de la ruta.

Si varios rayos tienen intersecciones cercanas y el mismo vector *up*, entonces un mismo punto del hemisferio va a ser transformado en puntos con coordenadas globales también cercanas. Así, las siguientes direcciones de cada ruta serán muy parecidas. Vamos a aprovechar esta propiedad en nuestra generación de rayos coherentes.

La generación coherente es un algoritmo que cambia la manera de elegir el siguiente rayo de una ruta sobre el hemisferio. En primer lugar, el conjunto de rayos se divide en grupos. Posteriormente, cada hemisferio sobre cada punto de intersección se divide en secciones llamadas *patches*. Por último, se selecciona aleatoriamente un patch para cada grupo de rayos. Los nuevos rayos van a ser elegidos como puntos aleatorios dentro de ese patch (Figura 1).

Los rayos primarios pueden ser fácilmente divididos en grupos de rayos coherentes usando los píxeles sobre los que se generan. Como los rayos de estos grupos tienen direcciones similares, entonces sus puntos de intersección van a estar próximos. Los rayos generados sobre estos puntos también van a ser coherentes ya que se generan sobre el mismo patch. Con ello hemos conseguido, con mucha probabilidad, que la coherencia se conserve en sucesivas reflexiones.

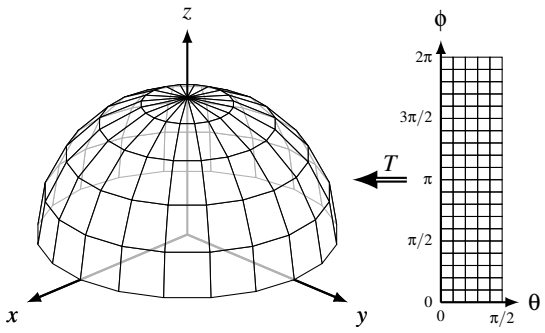

Figure 2: Patches del hemisferio con 5 divisiones de θ ($N = 5$) y 20 de ϕ ($M = 20$).

Para implementar la generación coherente, dividimos el hemisferio en $N \times M$ secciones, donde N y M son enteros. Consideremos la siguiente función T que transforma coordenadas esféricas en cartesianas

$$T(\theta,\phi) = (\sin\theta\cos\phi,\ \sin\theta\sin\phi,\ \cos\theta)$$

donde $\theta \in [0, \frac{\pi}{2}]$ y $\phi \in [0, 2\pi]$. Si dividimos el rango de la coordenada θ en N partes y el de la coordenada ϕ en M, el hemisferio queda dividido en $N \times M$ *patches* (Figura 2). Cada patch, por tanto, está definido como

$$Patch_{ij} = \{T(\theta,\phi) | \theta \in [i\Delta\theta, (i+1)\Delta\theta], \phi \in [j\Delta\phi, (j+1)\Delta\phi]\}$$

donde $\Delta\theta = \frac{\pi/2}{N}$, $\Delta\phi = \frac{2\pi}{M}$, $i \in \{0,1,\ldots,N-1\}$ y $j \in \{0,1,\ldots,M-1\}$. Es decir, un patch es la sección de hemisferio que genera T aplicada a cada rectángulo del espacio de coordenadas θ y ϕ. Por simplicidad, estableceremos $M = 4N$ durante el resto del artículo.

Las superficies *diffuse* son las que generan rayos más incoherentes, por tanto, a lo largo de este trabajo sólo vamos a considerar generación coherente en los puntos de intersección sobre superficies diffuse. Si algún rayo dentro de un grupo interseca con alguna superficie no diffuse, su siguiente rayo se genera según su BRDF, independientemente del resto y sin usar generación coherente.

Típicamente, el muestreo de un hemisferio para una superficie diffuse se lleva a cabo con una pdf proporcional al coseno del ángulo con la normal, lo que es conocido como *importance sampling*. Para hacer uso de esta característica vamos a hacer que la probabilidad de que el $Patch_{ij}$ sea elegido sea proporcional a dicho coseno. Para ello, se genera un punto o sobre todo el hemisferio con pdf $p(o) = \frac{(N \cdot o)}{\pi}$. El patch elegido es aquel que contiene al punto o. Posteriormente, se tiene que elegir un punto aleatorio dentro del patch. Esa elección se realiza con probabilidad uniforme dentro del propio patch.

4. Detalles de implementación

4.1. Descripción general

El renderizador usado es un path tracing implícito implementado en CUDA, usando una BVH construida con SAH como estructura de aceleración. La implementación de este algoritmo sigue las líneas de Antwerpen [vA11].

Para cada ruta es necesario guardar la información de su último rayo (origen y dirección) y el estimador parcial del color que aporta. Esta información se guarda en un array llamado *rays*. El orden de los rayos en este array coincide con el código Morton de los píxeles a los que pertenece. Sin embargo, los rayos van a ser reordenados durante el renderizado. Dicha reordenación no se realiza directamente sobre el array *rays*, sino usando un nivel más de indirección a través del array de enteros *idrays*.

Al comienzo del renderizado se lanza un hilo por cada píxel de la imagen. El id_g global de cada hilo representa el código Morton de las coordenadas de su píxel en la imagen. Cada hilo genera un rayo que sale de la cámara por dicho píxel id_g. Ese rayo se guarda en $rays[id_g]$ y su índice id_g en $idrays[id_g]$. Además, todos los hilos ponen a cero la imagen. Durante el renderizado, el array *idrays* va a ser reordenado (ver más abajo), por tanto, el valor $idrays[id_g]$ indicará el índice del rayo asociado a cada hilo.

La recursión de cada rayo termina en alguno de los siguientes casos: sale de la escena, encuentra una superficie luminosa o no es extendido debido a la ruleta rusa. En los dos primeros casos, la contribución de la ruta se acumula en la imagen. Si el rayo termina por ruleta rusa, su contribución es 0. En cualquiera de estos casos, el hilo vuelve a generar otro rayo desde la cámara por su píxel, regenerándose la ruta [NHD10].

Después de esta fase, los rayos regenerados se compactan al final del array y los extendidos quedan al principio. De esta manera, se intenta aprovechar la coherencia de los nuevos rayos primarios. Como ya se ha mencionado, esta ordenación no se hace directamente sobre el array de rayos *rays* sino sobre el array de índices *idrays*.

El algoritmo se ha implementado como un path tracing iterativo, de manera que se siguen extendiendo y regenerando rutas mientras no se haya alcanzado el tiempo de ejecución máximo. Sólo al principio del programa o cuando la cámara cambia de posición, el algoritmo tiene que comenzar de nuevo.

4.2. Kernels

El algoritmo completo se ha implementado en CUDA en cuatro kernels: *extends*, *compact*, *traversal* y *display*. Estos se ejecutan secuencialmente en este orden. Las imágenes renderizadas tienen una resolución de 1024×1024 y cada kernel se lanza con una configuración de rejilla de 2^{20} hilos (un hilo por rayo).

Extends genera el siguiente rayo de la ruta cuando se ha encontrado un punto de intersección. Si no se usa generación coherente, entonces el siguiente rayo se genera aleatoriamente usando la BRDF de la superficie pesada con el coseno. El caso de la generación coherente se explicará en la Subsección 4.3. Este kernel también se encarga de regenerar una ruta desde la cámara cuando la ruta ha terminado. *Compact* agrupa todos los rayos regenerados al final del array y los extendidos al principio. Está implementado con el *radixsort* de CUDPP 1.1.1 [HOS*10] usando sólo el bit menos significativo. *Traversal* encuentra el punto de intersección más cercano de cada rayo. Para este kernel se ha seguido el trabajo de Aila y Laine [AL09] que implementa el recorrido mediante *persistent threads*. *Display* muestra la información acumulada en la imagen parcial del path tracing. Su implementación se ha realizado para facilitar la interacción y depuración.

4.3. Generación coherente

Un grupo de rayos consiste en G hilos adyacentes, es decir, con índices globales id_g consecutivos. Por tanto, los hilos que pertenecen al mismo grupo tienen el mismo valor $\lfloor id_g/G \rfloor$. Los tamaños de G que hemos probado están relacionados con la forma en que se agrupan físicamente los hilos en CUDA. Concretamente, hemos considerado grupos con el tamaño de un warp ($G = 32$), un bloque ($G \in \{256, 512, 1024\}$) y el de toda la rejilla ($G = 2^{20}$).

Cuando el tamaño de G es un warp, un hilo del grupo se encarga de seleccionar el patch común al grupo. Posteriormente, el resto de hilos del warp recogen esa información de memoria compartida y cada uno prolonga su ruta hacia un punto sobre ese patch. No es necesario sincronización explícita por cómo se ejecutan los warps en GPU. Al path tracing que realiza la generación coherente de esta manera le hemos llamado `warp`. En esta configuración, el tamaño de bloque CUDA es de 256 hilos.

Cuando G es un bloque, un hilo se encarga también de elegir el patch común al grupo, pero esta vez es necesaria una sincronización explícita. Hemos usado la notación `block_256`, `block_512` y `block_1024` para referirnos a estos niveles de agrupación cuando $G = 256$, 512 y 1024, respectivamente.

Cuando G es la rejilla entera, todos los rayos pertenecen a un único grupo. En este caso es el *host* el que se encarga de elegir aleatoriamente el patch común a todos los rayos. Esto se implementa a través de una variable en la memoria global del dispositivo. El nombre que usaremos para este path tracing es `grid`.

Por último, hemos implementado también el path tracing tradicional en el que cada ruta se prolonga independientemente. A este algoritmo lo hemos llamado `normal` y lo usaremos como referencia.

Figure 3: Imágenes de referencia de las escenas (ConfRoom, FairyForest, Sponza y Stanford) usadas en los tests.

4.4. Ruleta rusa por grupo

Después del kernel *traversal*, es posible que algunos rayos de un grupo tengan que regenerarse y otros extenderse. Si esto ocurre, los rayos extendidos van a ser compactados al principio del array. Así, la siguiente vez que se realice la generación coherente, los grupos van a estar formados por rayos generados sobre diferentes patches, perdiéndose parte de la propiedad de la coherencia.

De las tres causas de terminación de una ruta, la salida de un rayo fuera de la escena o la llegada a una superficie luminosa dependen tanto de la generación aleatoria de los rayos como de la escena, por tanto, no son controlables. Sin embargo, podemos implementar una ruleta rusa que no sea independiente por rayo. En este caso, el test de la ruleta rusa se realiza una sola vez para todo el grupo. Si se pasa este test, todos los rayos se extienden. Si no, todos terminan y se regeneran. La probabilidad de pasar el test es la misma que en el caso individual (ver Sección 5). En `warp` y `block`, sólo un hilo del grupo realiza la ruleta rusa en el kernel *extends*. En `grid`, el proceso es idéntico excepto que es el host el que decide si terminar todos los rayos.

Con este método de ruleta rusa por grupo, los estimadores siguen siendo *unbiased*. Además, el número de rayos consecutivos coherentes es más probable que no cambie, con lo que la compactación puede no ser necesaria. Veremos esto experimentalmente en la Sección 5. Los algoritmos que incorporan la ruleta rusa por grupo llevan el sufijo `_rr` y si no realizan el kernel *compact* entonces terminan en `(NO)`.

5. Resultados

Las escenas que hemos usado para los experimentos son *ConfRoom*, *FairyForest*, *Sponza* y *Stanford* (Figura 3). Las imágenes renderizadas tienen una resolución de 1024×1024 y se ha usado una cámara *pinhole*. En las tres primeras, los materiales de todas las superficies son diffuse. En la escena Stanford sólo lo son las paredes, mientras que el Buda tiene un material *glossy*, el Dragón, de espejo perfecto y el Conejo, de refracción perfecta. El *albedo* de todas las superficies está fijo a 0.8 y usamos este valor como probabilidad de continuación en todos los tests de ruleta rusa.

Con estas escenas probamos la generación coherente sobre rutas de diferentes características. Las escenas ConfRoom y FairyForest están muy bien iluminadas así que la mayoría de las rutas tienen longitud 2 (una reflexión). Para la primera, todo el techo de la habitación es un área de luz, mientras que la segunda está abierta por arriba. Por otra parte, Sponza tiene un área de luz encima del atrio y la cámara está enfocando un lugar donde parte de la iluminación es indirecta (más de una reflexión). Finalmente, Stanford tiene un área de luz en la parte superior de la escena y superficies no diffuse. En esta escena se mezclan superficies sobre las que se hace generación coherente (superficies diffuse) con las que no (resto de superficies).

Los algoritmos anteriormente descritos se han probado sobre una GeForce GTX 580 con 1.5 GB de DRAM. Todos los experimentos se han llevado a cabo renderizando la escena con un límite de 30 segundos para el tiempo acumulado de los kernels *extends*, *compact* y *traversal*. Se han probado diferentes valores de N entre 4 y 1000, mientras que se ha tomado la variable $M = 4N$ siempre.

Las gráficas de la Figura 4 muestran el número de rutas terminadas por píxel (de media). El algoritmo `normal` corresponde a una recta constante debido a que su ejecución no depende de N. Los algoritmos sin ruleta rusa ni compactación no se han incluido porque su rendimiento cae significativamente.

En primer lugar, se observa que el número de rutas terminadas con todos los algoritmos de generación coherente supera a `normal`. Se ha comprobado experimentalmente (usando *NVidia Visual Profiler*) que la etapa de traversal es más rápida debido a que las cachés tienen una menor tasa de fallos y a que el número de transferencias de memoria off-chip es menor. En segundo lugar, según crece N, las direcciones generadas se hacen más parecidas dentro de cada grupo porque los patches son más pequeños. En consecuencia, las curvas de nuestros algoritmos son crecientes, es decir, el traversal es más rápido según crece N.

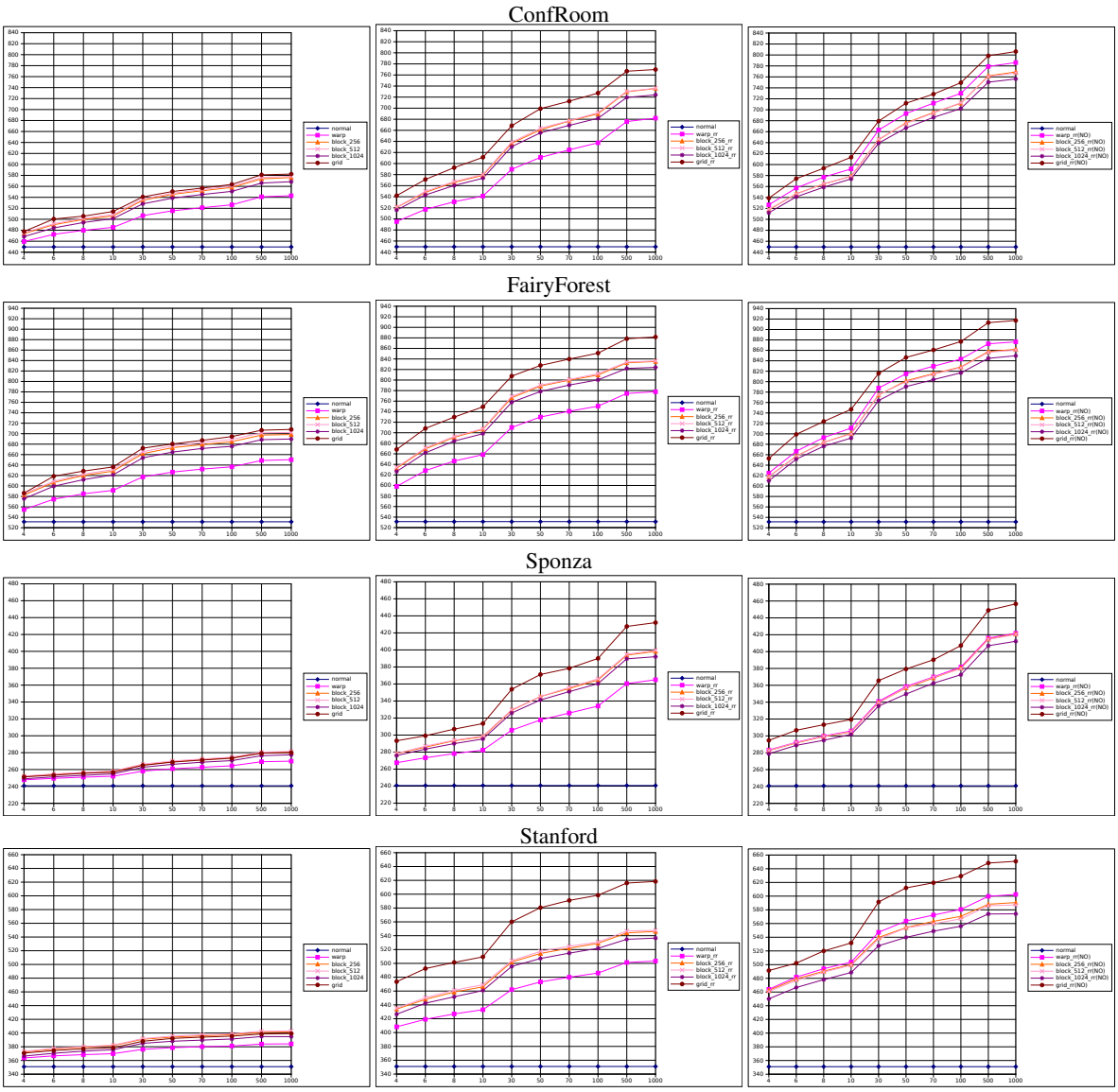

Figure 4: Rutas terminadas por píxel en media.

Los algoritmos que generan a nivel de grid son los que ofrecen mejor rendimiento debido a que su tasa de aciertos de caché es la mayor. Después se encuentran los algoritmos basados en bloque y en warp. En las tres columnas, las curvas de los algoritmos que generan en bloque son muy parecidas. Las curvas correspondientes a los tamaños 256 y 512 discurren juntas, y la asociada al tamaño 1024 queda ligeramente por debajo. El motivo es que con bloques de 1024 hilos, sólo un bloque se asigna a cada multiprocesador de la GPU, por lo que las barreras de sincronización suponen una penalización mayor que para los otros tamaños.

En cuanto a las curvas correspondientes a la agrupación a nivel de warp, su ubicación depende de la columna en cuestión. En las dos primeras columnas ofrecen el peor rendimiento mientras que en la tercera quedan en segundo lugar, superando a los algoritmos por bloque. Recuérdese que la única diferencia entre las columnas segunda y tercera consiste en que en la tercera no se compacta. Así el beneficio obtenido cuando se elimina la compactación es mayor para la agrupación en warps que para la agrupación en bloques. La explicación es que los grupos pequeños sufren más que los grandes tras la compactación. Si, por ejemplo, uno de los primeros rayos decide regenerarse, entonces se coloca al final del array *idray*. Esto supone desorganizar los

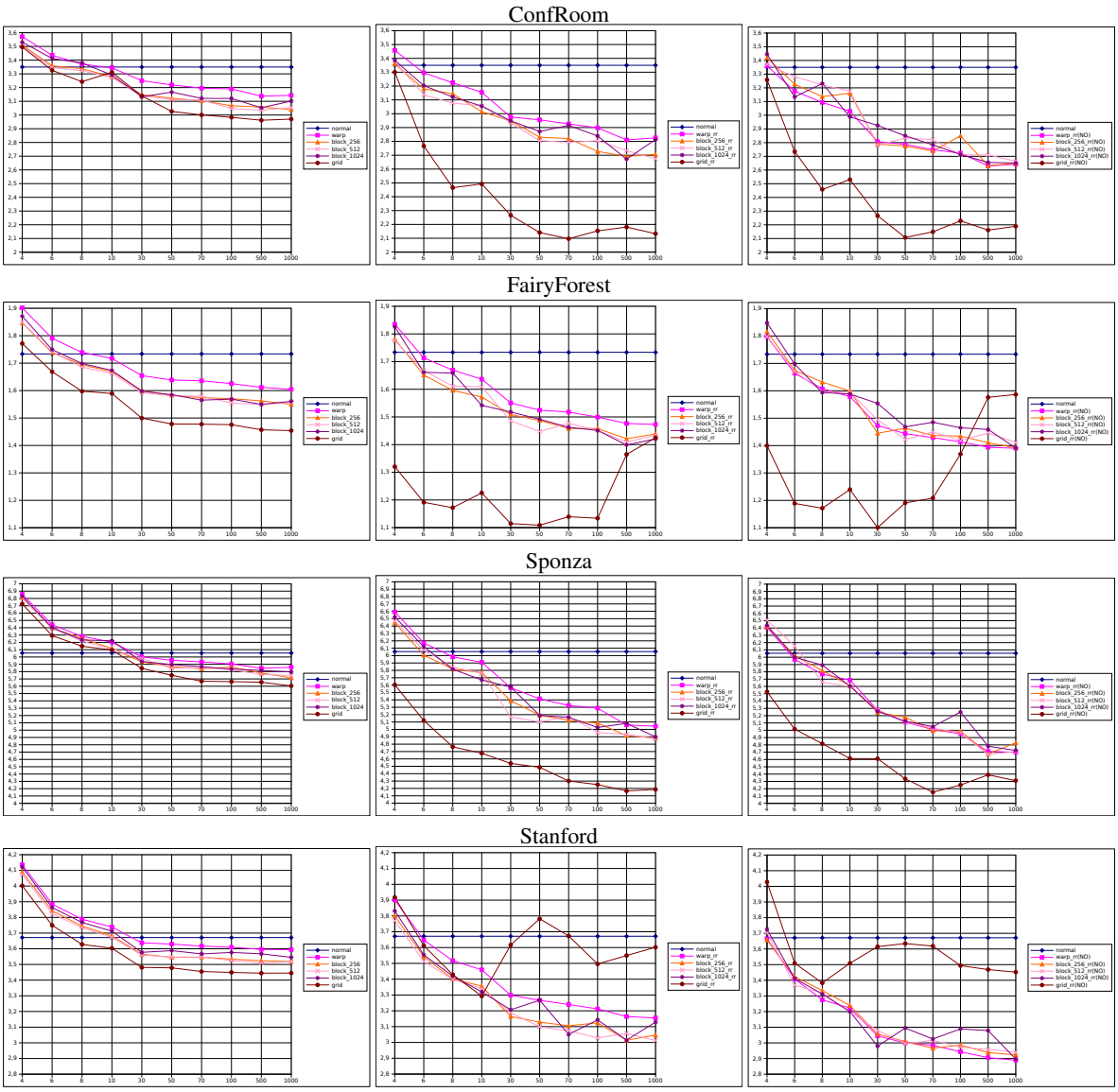

Figure 5: RMS (en %) de los algoritmos con respecto a las imágenes de referencia de la Figura 3.

actuales grupos de rayos, ya que pierden uno de sus rayos e introducen otro del siguiente grupo. Como cada grupo se ejecuta en la GPU en warps de 32 hilos (ejecución en SIMT) esta reorganización afecta más al agrupamiento basado en warp porque todos los grupos han visto modificada la composición de sus rayos.

En la Figura 5 se analiza el error de las imágenes resultantes medido como la raíz del error cuadrado medio (*root mean square* o *RMS*) de cada píxel (canales RGB) con respecto a las imágenes de referencia (Figura 3). Estas imágenes se han obtenido aplicando el algoritmo `normal` durante 20 minutos.

Cuando N crece, el número de rutas terminadas aumenta, lo que supondría mayor calidad en la imagen. Pero al mismo tiempo la correlación entre las muestras de los píxeles del mismo grupo también será mayor, lo que implicaría un mayor error. Las curvas de la Figura 5 indican que tiene mayor influencia la ganancia que aporta un número mayor de rutas terminadas, ya que se trata de curvas decrecientes en general. De hecho, comienzan con más error a pesar de que el número de rutas terminadas es mayor. A medida que el número de rutas terminadas crece, el error cae por debajo de `normal`. Sin embargo, los algoritmos basados en grid tienen un comportamiento más imprevisible, como se

Figure 6: Detalles de algunas capturas de la escena Sponza. Las imágenes han sido obtenidas con $N = 100$. Todas las imágenes tienen un RMS menos que `normal`.

puede ver en las gráficas de la segunda y tercera columna en FairyForest y Stanford.

En la Figura 6 presentamos capturas de las imágenes generadas por algunos de nuestros algoritmos. La influencia de la correlación entre las muestras de los píxeles del mismo grupo aparece visualmente como un patrón de rejilla cuadrada para los algoritmos basados en bloque. Para los basados en warp sucede algo parecido aunque no lo mostramos por falta de espacio. Este efecto se hace especialmente visible para valores grandes de N y con ruleta rusa por grupo. Aunque el RMS es menor que el del algoritmo `normal`, este patrón resulta molesto desde un punto de vista subjetivo.

Para aliviar su efecto, hemos usado una técnica relacionada con el *interleaved sampling* [KH01]. Consiste en modificar las coordenadas del píxel al que cada ruta aporta, multiplicando cada coordenada por un entero impar D. Los valores que hemos usado para nuestros experimentos son $D = 3$ y $D = 5$. Esta técnica tiene la ventaja de que substituye el patrón de rejilla por un ruido que se reparte por la imagen. Sin embargo, su uso implica que los rayos primarios van a tener menos coherencia ya que son generados en píxeles más alejados entre sí. Esto se aprecia experimentalmente como una disminución del número de rutas terminadas. En la Figura 7, se muestra el RMS usando $D = 5$. Se aprecia que el error de los algoritmos que usan interleaved sampling (con sufijo `D5`) es mayor que los que no lo usan. Los algoritmos de la primera columna son los más lentos y su error no supera al error de `normal`. En las otras columnas, los algoritmos también tienen un error mayor, pero sí superan a `normal` a medida que crece N.

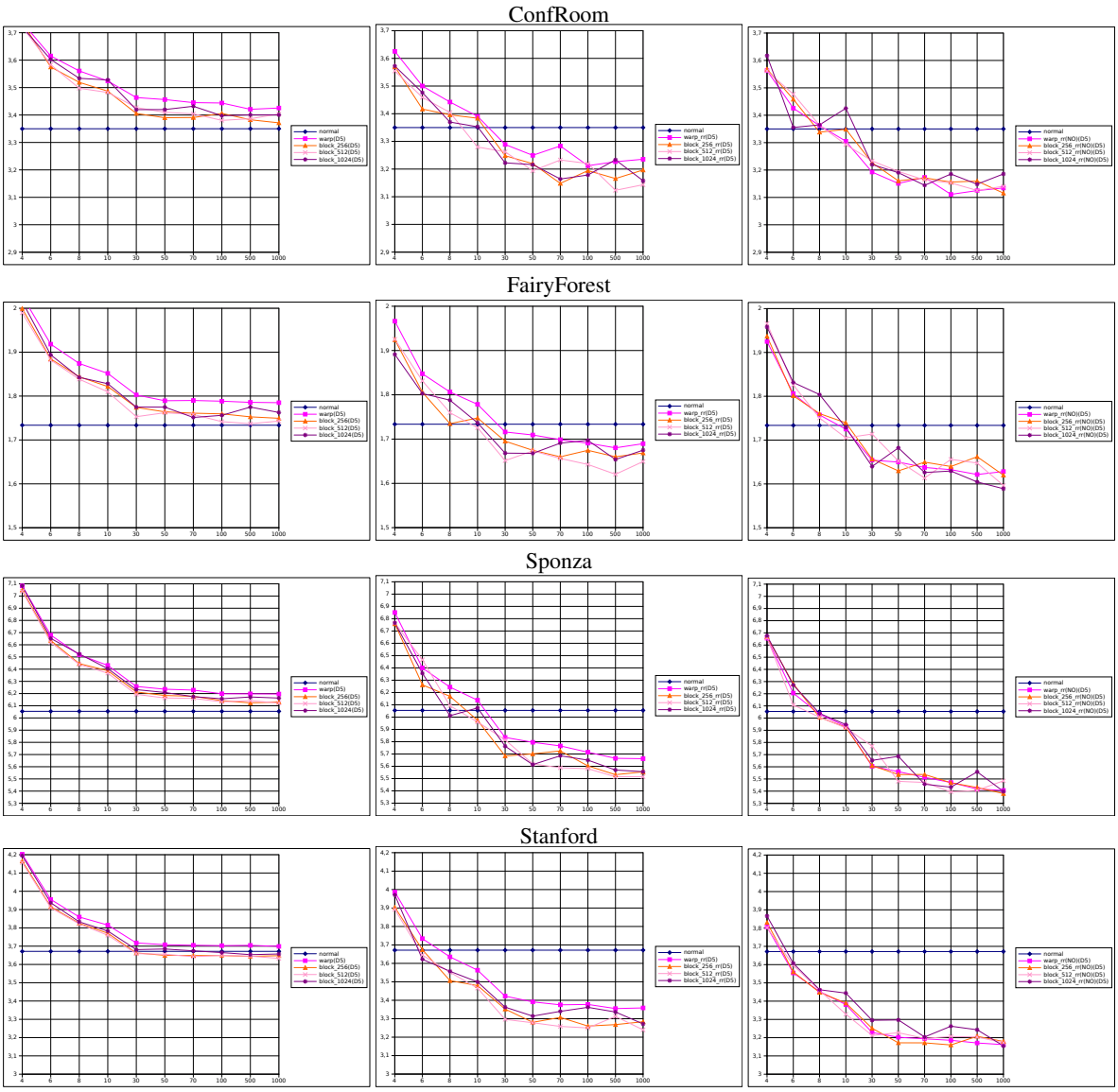

Figure 7: RMS (en %) de los algoritmos con respecto a las imágenes de referencia con D=5.

6. Trabajo futuro

Los patches en los que está dividido el hemisferio no poseen la misma área. Habría que comprobar la técnica de generación coherente cuando los patches son todos iguales (como en las esferas geodésicas) o cuando se genera por una función que preserva áreas (como en [SC97]).

La generación coherente se puede aplicar siempre que se tenga que muestrear un conjunto de direcciones. En este sentido, se podría usar sobre superficies con BRDFs no lambertianas o con medios participativos.

El rendimiento se puede degradar en cada reflexión de las rutas. Eso se debe a que los puntos de intersección están cada vez más alejados entre sí, intersecan superficies con normales muy diferentes o algún rayo del grupo tiene que terminar. Para aliviar estos problemas se proponen dos soluciones. La primera es implementar una fase de reordenación de rayos en función de sus cualidades geométricas. Esta fase se podría realizar sólo cada cierto número de reflexiones para que no sea el cuello de botella del rendimiento. La segunda es la terminación de todos los rayos del grupo siempre que alguno tuviera que terminar. En este trabajo se ha realizado esto en la ruleta rusa por grupo, pero se podría extender a los otros dos casos de terminación.

7. Conclusiones

En este trabajo hemos planteado una manera diferente de extender rutas en path tracing llamada *generación coherente*. Esta generación es fácil de implementar y aprovecha mejor la forma de computación de las GPUs. En concreto, el conjunto de rayos es dividido en grupos. Para determinar la extensión de cada ruta, el hemisferio sobre cada punto de intersección se divide en patches y cada grupo elige uno de ellos. El siguiente rayo de cada ruta se genera aleatoriamente sobre ese patch. Con ello se consigue que la coherencia del grupo se conserve a lo largo del trazado de las rutas de cada grupo, acelerando así la fase de traversal. En consecuencia, se terminan más rutas y las imágenes obtenidas, en un mismo período de tiempo, son de mayor calidad.

Sobre esta idea, hemos implementado en CUDA distintas configuraciones de un path tracing. Distinguimos casos por la forma en que se agrupan los rayos, y el número de divisiones del hemisferio, así como por la forma de organización del algoritmo de renderizado, incluyendo o no técnicas como la compactación y la ruleta rusa. Concluimos que, en general, nuestras propuestas tienen un rendimiento mejor que el algoritmo clásico basado en la generación puramente aleatoria de rayos.

References

[AL09] AILA T., LAINE S.: Understanding the Efficiency of Ray Traversal on GPUs. In *High-Performance Graphics* (2009), pp. 145–149. 2, 4

[BEL*07] BOULOS S., EDWARDS D., LACEWELL J. D., KNISS J., KAUTZ J., WALD I., SHIRLEY P.: Packet-based Whitted and Distribution Ray Tracing. In *Graphics Interface* (2007), pp. 177–184. 2

[BWB08] BOULOS S., WALD I., BENTHIN C.: Adaptive Ray Packet Reordering. *Symposium on Interactive Ray Tracing* (2008), 131–138. 2

[GL10] GARANZHA K., LOOP C.: Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. In *Eurographics* (2010). 2

[GPSS07] GÜNTHER J., POPOV S., SEIDEL H.-P., SLUSALLEK P.: Realtime Ray Tracing on GPU with BVH-based Packet Traversal. In *Interactive Ray Tracing* (2007), pp. 113–118. 2

[GR08] GRIBBLE C. P., RAMANI K.: Coherent Ray Tracing via Stream Filtering. In *Eurographics Symposium on Interactive Ray Tracing* (2008), pp. 59–66. 2

[Hac05] HACHISUKA T.: High-Quality Global Illumination Rendering Using Rasterization. In *GPU Gems 2*. Addison-Wesley, 2005, pp. 615–633. 2

[HHGM10] HERMES J., HENRICH N., GROSCH T., MUELLER S.: Global Illumination using Parallel Global Ray Bundles. In *Vision, Modeling and Visualization* (2010), pp. 65–72. 2

[HOS*10] HARRIS M., OWENS J. D., SENGUPTA S., TSENG S., ZHANG Y., DAVIDSON A., SATISH N.: CUDA Data Parallel Primitives Library (CUDPP 1.1.1), 29 April 2010. http://code.google.com/p/cudpp/. 4

[HSMH07] HORN D. R., SUGERMAN J., MIKE H., HANRAHAN P.: Interactive KD-Tree GPU Raytracing. In *I3D* (2007), pp. 167–174. 2

[Jen96] JENSEN H. W.: Global Illumination Using Photon Maps. In *Eurographics Workshop on Rendering Techniques* (1996), pp. 21–30. 2

[Kaj86] KAJIYA J. T.: The Rendering Equation. *SIGGRAPH Computer Graphics 20*, 4 (1986), 143–150. 1

[Kel97] KELLER A.: Instant Radiosity. In *Computer Graphics and Interactive Techniques* (1997), pp. 49–56. 2

[KH01] KELLER A., HEIDRICH W.: Interleaved Sampling. In *Eurographics Workshop on Rendering* (2001). 8

[MMAM07] MANSSON E., MUNKBERG J., AKENINE-MOLLER T.: Deep Coherent Ray Tracing. In *Symposium on Interactive Ray Tracing* (2007), pp. 79–85. 2

[NFLM07] NAVRATIL P. A., FUSSELL D. S., LIN C., MARK W. R.: Dynamic Ray Scheduling to Improve Ray Coherence and Bandwidth Utilization. In *Symposium on Interactive Ray Tracing* (2007), pp. 95–104. 2

[NHD10] NOVÁK J., HAVRAN V., DASCHBACHER C.: Path Regeneration for Interactive Path Tracing. In *Eurographics, short papers* (2010), pp. 61–64. 2, 4

[NUG09] NOGUERA J. M., UREÑA C., GARCÍA R. J.: A Vectorized Traversal Algorithm for Ray-Tracing. In *GRAPP* (2009), pp. 58–63. 2

[NVI11] NVIDIA: NVidia CUDA C Programming Guide 4.1 www.nvidia.com, 2011. 2

[PGSS07] POPOV S., GÜNTHER J., SEIDEL H.-P., SLUSALLEK P.: Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum 26*, 3 (2007), 415–424. 2

[PKGH97] PHARR M., KOLB C., GERSHBEIN R., HANRAHAN P.: Rendering Complex Scenes with Memory-Coherent Ray Tracing. In *SIGGRAPH* (1997), pp. 101–108. 2

[SC97] SHIRLEY P., CHIU K.: A Low Distortion Map Between Disk and Square. *J. Graphics Tools 2*, 3 (1997), 45–52. 9

[TMG11] TORRES R., MARTIN P., GAVILANES A.: Traversing a BVH Cut to Exploit Ray Coherence. In *GRAPP* (2011), pp. 140–150. 2

[vA11] VAN ANTWERPEN D.: Improving SIMD Efficiency for Parallel Monte Carlo Light Transport on the GPU. In *High Performance Graphics* (2011), pp. 41–50. 2, 4

[Vea98] VEACH E.: *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, 1998. 1

[WBWS01] WALD I., BENTHIN C., WAGNER M., SLUSALLEK P.: Interactive Rendering with Coherent Ray Tracing. In *Computer Graphics Forum (Proceedings of Eurographics'01)* (2001), vol. 20, pp. 153–164. 2

[WGBK07] WALD I., GRIBBLE C. P., BOULOS S., KENSLER A.: *SIMD Ray Stream Tracing - SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering*. Tech. rep., 2007. 2

[WKB*02] WALD I., KOLLIG T., BENTHIN C., KELLER A., SLUSALLEK P.: Interactive Global Illumination Using Fast Ray Tracing. In *Eurographics Workshop on Rendering* (2002), pp. 15–24. 2