Ray Tracing sobre Hardware Gráfico Programable

Leandro Fal, Guillermo Vasconcelos y Claudio Delrieux*
Departamento de Computación
FCEN-UBA - Ciudad Universitaria - Buenos Aires - ARGENTINA
leandrofal@gmail.com, guillevasco@fibertel.com.ar, claudio@acm.org
*(También Depto. Ing. Eléctrica y Computadoras, UNS, parcialmente financiado por SECyT-UNS)

Resumen

El objetivo de este trabajo es implementar el algoritmo de ray tracing en hardware gráfico programable. El método de ray tracing tiene la ventaja de que posibilita obtener imágenes con alto grado de realismo, permitiendo implementar reflexiones mutuas, sombras e integrar distintos modelos de iluminación. Los procesadores gráficos están diseñados para renderizar utilizando el modelo *scan-line*. Pese a esta especificidad computacional, es posible utilizar dichos procesadores para realizar *streaming*. Además, este hardware tiene características que se pueden aprovechar para reducir los tiempos de renderizado, como el procesamiento en paralelo y las operaciones vectoriales. Adicionalmente, se está desarrollando muy rápidamente, mejorando su performance y ampliando sus funcionalidades. En este trabajo describimos una implementación del ray tracing basada en computación de *streams*. Los resultados muestran que, con el ritmo natural de avance en esta tecnología, el objetivo de contar con ray tracing en tiempo real es una posibilidad de inminente realización.

1 Introducción

Hasta hace algunos años, era impensable obtener aplicaciones interactivas con imágenes fotorrealistas generadas en tiempo real, utilizando computadoras convencionales. Por lo tanto, a la hora de diseñar aplicaciones gráficas, se debía elegir entre solo una de esas dos características. Sin embargo, en los últimos años, con el advenimiento del hardware gráfico programable (GPU), esta brecha entre el fotorrealismo y la interactividad se fue acortando. Este hardware permite implementar los más elaborados algoritmos de rendering y modelos de iluminación. Al ser un hardware especializado en la generación de imágenes y tener un modelo de procesamiento en paralelo (SIMD), realiza las operaciones gráficas con mayores velocidades que en la CPU.

El primer modelo de iluminación global, y actualmente entre los más difundidos y utilizados, es el ray tracing (RT) [3]. Mediante este método se pueden generar imágenes con iluminación global y shading de alto realismo. La característica más importante del RT es que

es un algoritmo recursivo que computa el *shading* (los distintos matices del color de los objetos) teniendo en cuenta la interacción de los mismos con las fuentes de iluminación y otros objetos de la escena. Hasta hace algún tiempo, sólo era posible computar RT en tiempo real utilizando hardware específico [4]. Nuestro objetivo en este trabajo es implementar RT en tiempos interactivos en computadoras PC de bajo costo, utilizando la programabilidad de la GPU. Si bien este hardware está diseñado específicamente para renderizar *scan-line*, provee suficiente poder de programación para implementar RT. Los resultados muestran que con el estado actual de la tecnología es posible ejecutar el algoritmo a suficiente velocidad como para desarrollar aplicaciones interactivas que muestren escenas interesantes en un tiempo aceptable.

En este trabajo describimos una implementación del RT aprovechando las capacidades del hardware gráfico, así como variantes del mismo. Esta implementación se realizó con el lenguaje de programación de shaders CG de NVidia [2]. También realizamos una comparación en cuanto a la performance con respecto al RT implementado en la CPU. Luego analizamos los resultados obtenidos y estudiamos posibles mejoras en la implementación. También mostramos las conclusiones obtenidas sobre el estado actual de esta tecnología y la distancia a la que nos encontramos de alcanzar la meta del fotorrealismo en tiempo real en aplicaciones prácticas.

2 Ray Tracing y computación de *streams*

Debido a que el hardware gráfico programable está diseñado para optimizar la ejecución del modelo *scan-line*, para implementar RT hay que mapearlo a un modelo que sea ejecutable por el hardware. En este trabajo se propone pensar al RT como un *pipeline* entre *kernels*, con distintas etapas por las que va pasando cada pixel hasta que queda definido su color final. Para poder implementar el método, son necesarias algunas suposiciones iniciales para simplificar el problema y hacerlo más manejable por el hardware. El método implementado utiliza como única primitiva el triángulo, debido a que es la primitiva más utilizada para describir objetos y la que se suele usar en el método de scan-line. Además, el hardware gráfico está organizado para polígonos convexos, cuanto más simples mejor.

Además, para que los métodos tengan un rendimiento aceptable, se deben organizar los objetos en una estructura aceleradora, de manera que no sea necesario comprobar las intersecciones de los rayos con todos los objetos de la escena, sino con un subconjunto relevante. La implementación utiliza como estructura aceleradora una grilla uniforme, debido a que el recorrido de esta grilla es implementable por hardware con relativa sencillez (utilizando una técnica de recorrido de voxels DDA 3D similar a la presentada en [10]) y da en general buenos resultados de performance (como toda estructura aceleradora, su performance depende de la escena en particular). Cada uno de los cubos definidos por esta grilla es llamado *voxel*. La comunicación entre *kernels* se realiza enviando todos los pixels en paralelo a través de una textura. Los diferentes *kernels* de la implementación se describen a continuación.

2.1 Generador de rayos

El generador de rayos es el kernel más simple de los cuatro, tanto en sus funciones como en su implementación. Este kernel se ejecuta una vez al comienzo del ciclo de renderizado. Genera un rayo por pixel de la imagen final en base a la posición y orientación de la cámara, y guarda la información de los rayos generados en una textura. Como los rayos son vectores, se aprovecha la naturaleza vectorial de la aritmética de la GPU. Además, como las direcciones de los rayos varían linealmente dentro del viewport, con un único paso desde el procesador de vértices se puede utilizar el convertidor scan de la GPU para generar los rayos como *texcoords* en el procesador de fragmentos.

Eventualmente, si por alguna razón se dispusiera un esquema diferente de muestreo [5], entonces habría que modificar la programación de este kernel. Situaciones donde puede ocurrir un muestreo diferente son, por ejemplo, realizar varias muestras ponderadas (oversampling) por pixel para luego aplicar anti-aliasing, realizar subdivisiones adaptativas para optimizar el cómputo (undersampling adaptativo) [7], o utilizar alguna de las técnicas aceleradas de RT como beam tracing [8], pencil tracing [12], etc.

2.2 Recorrido

La etapa de recorrido o *traversal* cumple dos funciones diferenciadas. La primera consiste en encontrar los voxels de entrada de los rayos en la estructura aceleradora. Esto implica extender los rayos hasta que lleguen a los bordes de la grilla, y encontrar el voxel que corresponde a esta intersección. Esta tarea se realiza justo después del generador de rayos, y una sola vez por ciclo de renderizado. La segunda función que tiene este kernel es la de recorrer la grilla, atravesando los sucesivos voxels en la dirección de los rayos, hasta encontrar voxels que contengan triángulos o llegar al final de la grilla.

El recorrido de la grilla se realiza utilizando un algoritmo DDA 3D [6], donde cada iteración del algoritmo se corresponde con una ejecución del kernel. Esto significa que por cada ejecución del kernel, el rayo avanza un voxel solamente. Durante la iteración se decide si el próximo kernel que se debe ejecutar para el rayo en cuestión es el detector de intersecciones, de nuevo el traversal o ninguno, si el rayo sale fuera de la grilla.

2.3 Detector de intersecciones

Esta etapa recorre los triángulos de cada voxel buscando intersecciones entre los mismos y el rayo que se está analizando. En cada ejecución de este kernel se evalúa la intersección del rayo con un triángulo del voxel en particular. Cuando ya se evaluó la intersección con todos los triángulos del voxel, se decide si el próximo kernel a ejecutar es el traversal (si no hubo ninguna intersección) o el shader.

En el caso en que dos o más triángulos son intersectados por el rayo, se toma como triángulo de intersección al que sea alcanzado por el rayo a menor distancia del origen del mismo. También puede suceder que el punto de intersección del rayo con un triángulo caiga

fuera del voxel que se está analizando. En este caso esta intersección no se tomará como válida en este momento y sólo será válida si el rayo llega hasta el voxel al que pertenece dicho punto de intersección.

2.4 Shader

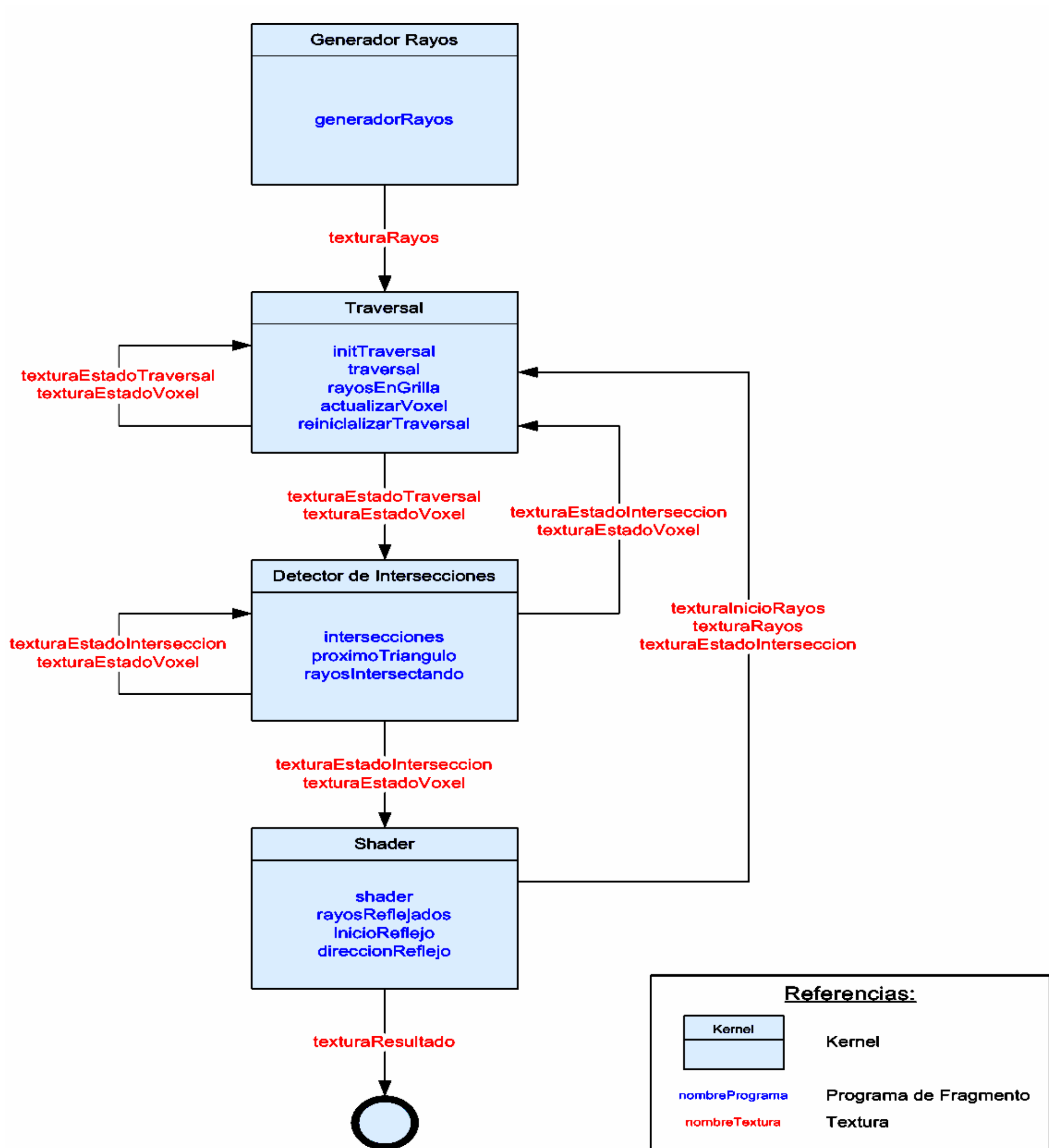
Esta etapa tiene como función generar el color de los pixels de la imagen final. Para esto toma el triángulo intersectado por el rayo, y calcula la iluminación utilizando las propiedades de ese triángulo y su normal interpolada en el punto de intersección teniendo en cuenta también la posición de las luces. El kernel puede utilizar diferentes modelos de iluminación para obtener el color del pixel en la imagen final. Adicionalmente este kernel puede generar nuevos rayos de sombras o reflexiones que son pasados al kernel traversal para continuar el cálculo. El resultante de estos nuevos rayos es combinado con el original para obtener el color final para ese pixel.

Para implementar interreflexión en el RT tradicional se utiliza recursión. Como el modelo de ejecución de las placas gráficas no permite recursión, esto se implementó volviendo atrás en el pipeline, redefiniendo el origen y la dirección de cada rayo que intersecte a una cara reflectiva, en base al punto y el ángulo de intersección. El resultado final de cada pixel se obtiene mediante una combinación de los resultados parciales ponderados por los coeficientes de reflexión de las distintas superficies intersectadas durante toda la ejecución del pipeline.

3 Implementación

3.1 Componentes del Pipeline

La organización del pipeline se basa en ideas similares a las presentadas en [1], donde se aprovechan las características de paralelismo en las GPU durante el rendering sobre texturas. Por lo tanto, la comunicación entre los kernels del pipeline se realiza por medio de texturas. La descripción de las mismas está en la Sec. 3.3, mientras que los programas de fragmentos que implementan los kernels se detallan en la Sec. 3.5.



3.2 Estructura aceleradora

Como mencionamos anteriormente, para mejorar la performance del programa trabajamos con una estructura aceleradora que se carga *off-line* (se carga una vez al comenzar el programa, y permanece invariante durante toda la ejecución). La finalidad de esta estructura es analizar solamente combinaciones rayo-triángulo que tienen chances de intersectarse, y no calcular intersecciones de todos los rayos con todos los triángulos. Existen distintos tipos de estructuras aceleradoras. En este caso se emplea una grilla tridimensional uniforme que secciona la escena en *n* cubos, llamados voxels, y ubica dentro de cada cubo los triángulos que tienen alguna parte en su interior. De esta manera, mientras los rayos van atravesando la

escena, sólo se realiza el cálculo de intersección con los triángulos que pertenecen al cubo en el que se encuentra cada rayo. Si bien esta estructura acelera el proceso de renderizado de manera notable, el tiempo que se consume durante el armado de la misma es alto, sobre todo en escenas con gran complejidad. Por dicha razón esta estructura es estática.

3.3 Texturas empleadas

Para almacenar datos durante la ejecución del programa se utilizan texturas que son modificadas y accedidas desde OpenGL y CG. De acuerdo a la persistencia que tienen los datos que se utilizan a lo largo de la ejecución del programa, podemos clasificarlas en dos tipos. Las texturas estáticas se calculan al comienzo de la ejecución del programa y perduran invariantes hasta el final de la misma. Dentro de estas se encuentran las texturas que representan a la escena.

Nombre de la textura	**Contenido**
`texturaVertices`	Arreglo de 3 texturas con los 3 vertices de cada triangulo. Se las accede con un ID de triángulo (*id*), y cada una de estas texturas tiene la posición (X,Y,Z) de un vértice del triángulo *id***.**
`texturaColores`	Se accede con el ID de triángulo (*id*) y se obtiene el color (R, G, B) y el coeficiente de reflexión del triángulo *id***.**
`texturaNormales`	Arreglo de 3 texturas con las normales de los 3 vertices de cada triangulo. Se las accede con un ID de triángulo (*id*), y cada una de estas texturas tiene la normal (X,Y,Z) de un vértice del triángulo *id*.
`texturaTriangulos`	Contiene los números de triángulos que se encuentran en cada voxel. Los voxels estan uno atrás del otro y separados por *-1*. Se indexa con la `texturaTriangulosEnVoxel`
`texturaTriangulosEnVoxel`	Contiene el puntero al comienzo de cada voxel en la `texturaTriangulos`. Para hacer la transformacion de 3D a 2D, para el voxel X,Y,Z, tomamos *X*tamGrillaY+Y, Z.*

Las texturas dinámicas, en cambio, varían durante el renderizado de la imagen, y llevan los estados en los que se encuentran los distintos rayos. Dentro de esta categoría se encuentran todos los datos que son modificados por la ejecución de los distintos kernels.

Nombre de la textura	**Contenido**
`texturaRayos`	Guarda en X,Y,Z la dirección de cada rayo.
`texturaEstadoTraversal`	En X,Y,Z guarda el voxel al cual apunta el rayo, en W va el coeficiente por el cual se multiplica la dirección del rayo para entrar al voxel actual que se está analizando.
`texturaEstadoVoxel`	En X está la posición a la que apunta en texturaTriangulos. En W está el número de triángulo activo para el rayo.

texturaEstadoInterseccion	En X está el coeficiente por el cual se multiplica la dirección del rayo para alcanzar el punto de intersección y en W está el triángulo con el cual se produce la intersección.
texturaResultado	Contiene el color actual de cada pixel. En alfa tiene el coeficiente que aporta el color actual al color final de la imagen para cada pixel.

3.4 Interreflexión

Se produce una reflexión cuando un rayo intersecta un objeto cuyo coeficiente especular es mayor a 0. Esta reflexión genera un rayo con origen en el punto de intersección y reflejado con respecto a la normal del objeto en dicho punto.

Para calcular el color final del pixel, se combinan los resultados parciales de los rayos correspondientes a dicho pixel (el original y sus reflejos), según la siguiente fórmula:

$$\text{ColorFinal} = C_1 (1 - \alpha_1) + \alpha_1 (C_2 (1 - \alpha_2)) + \alpha_1 \cdot \alpha_2 (C_3(1 - \alpha_3)) + \ldots$$

C_i es el color del i-ésimo objeto intersectado por el rayo, y α_i es el coeficiente especular del i-ésimo objeto intersectado por el rayo.

El cálculo de los rayos reflejados se realiza una vez que todos los rayos terminaron su recorrido por la grilla, ya sea intersectando con algún objeto o saliendo de la misma. Luego se realiza una nueva iteración hasta que estos nuevos rayos terminen su recorrido. Esta operación se repite hasta que ningún rayo sea reflejado, o se alcance el número máximo de iteraciones establecido.

3.5 Programas de fragmentos

En la figura anterior puede verse que, asociados a estos kernels y texturas existe un conjunto de programas o *shaders* que el procesador de fragmentos de la GPU computa durante las distintas etapas del rendering.

Programa de fragmentos	**Propósito**
generadorRayos Modifica: texturaRayos	Se ejecuta solamente una vez por ciclo de renderizado. Genera la dirección inicial de cada rayo. Esta dirección es obtenida como la normalización del vector comprendido entre las coordenadas de cada pixel con z = 0 y el punto (0, 0, -1).

initTraversal Modifica: texturaEstadoTraversal	Se ejecuta solamente una vez por ciclo de renderizado. Calcula el voxel de entrada de cada rayo en la estructura aceleradora. Para esto calcula el mínimo factor por el que se multiplica a la dirección del rayo para alcanzar un borde de la grilla. Con esto calcula el punto por el que ingresa a la grilla, y a qué voxel pertenece.
traversal Modifica: texturaEstadoTraversal	Encuentra el próximo voxel por el que tiene que pasar el rayo. Para esto se utiliza un algoritmo DDA modificado para que considere todos los voxels por los cuales pasa el rayo.
rayosEnGrilla Modifica: cantActivos	Calcula la cantidad de rayos que están activos, es decir, que no encontraron todavía un triángulo y están en algún voxel de la grilla.
actualizarVoxel Modifica: texturaEstadoVoxel	Asigna el primer triángulo con el cual se tiene que hacer la intersección para cada rayo, de acuerdo al voxel en que se encuentra. Esta información es obtenida de *texturaTriangulosEnVoxel* y *texturaTriangulos*.
reinicializarTraversal Modifica: texturaEstadoTraversal	Luego de que se modifica el inicio y la dirección de algún rayo, este programa actualiza el factor por el cual se debería multiplicar la dirección de dicho rayo para entrar al voxel actual.
intersecciones Modifica: texturaEstadoInterseccion	Analiza si existe intersección entre cada rayo y el triángulo que le corresponde en la iteración actual (este triángulo es obtenido de *texturaEstadoVoxel*). Si se produce una intersección analiza también si ésta se produce más cerca del inicio del rayo que la intersección más cercana calculada hasta el momento, y si se produce dentro del voxel que se está considerando.
proximoTriangulo Modifica: texturaEstadoVoxel	Obtiene el próximo triángulo a analizar para encontrar intersecciones con cada rayo. Si no hay más triángulo en el voxel para un rayo, éste queda marcado como listo para traversal.
rayosIntersectando Modifica: cantIntersectando	Calcula la cantidad de rayos que tienen al menos un triángulo pendiente para intersectar en el voxel actual.
shader Modifica: texturaResultado	Calcula el color que se debe mostrar en cada pixel de la imagen final. Dependiendo del coeficiente de reflexión del objeto con el que interseca el rayo, el valor obtenido puede ser sólo un porcentaje del color final.
rayosReflejados Modifica: cantReflejados	Calcula la cantidad de rayos que han sido reflejados en la ejecución del shader. Es decir, los que intersecaron con un objeto reflectante.
inicioReflejo Modifica: texturaInicioRayos	Establece el punto de inicio de los rayos reflejados, utilizando el punto de intersección anterior.
direccionReflejo Modifica: texturaRayos	Establece la dirección de los rayos reflejados, utilizando la dirección anterior y la normal en el punto de intersección.

4 Resultados

En las siguientes imágenes se ven los resultados obtenidos al ejecutar la aplicación para diferentes escenas, utilizando una GPU NVidia GF 5700. Junto a las imágenes se incluye información relevante a la misma, como el tiempo de renderizado (sin incluir la inicialización) y la cantidad de divisiones de la grilla, por ejemplo. También se muestra el resultado de la comparación de algunos aspectos que se modifican al variar diferentes parámetros o condiciones de ejecución. Todas las imágenes fueron renderizadas en 256 x 256 pixels.

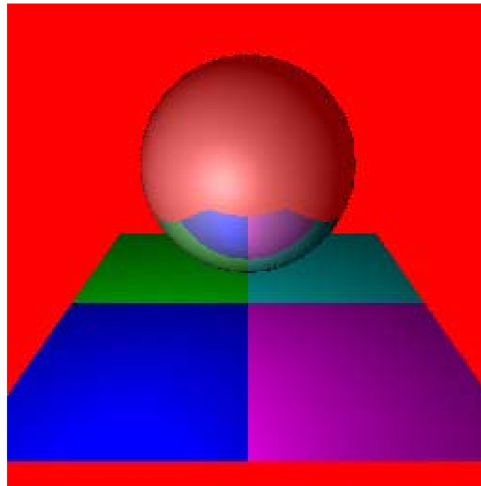


Figura 1: 848 caras, 3 iteraciones de RT, grilla aceleradora 15x15x15, 7852 ms.

En la Fig. 1 se puede ver la escena tradicionalmente utilizada para testear RT, junto con los parámetros específicos del renderizado. En la siguiente figura se compara el tiempo de renderizado con distinto nivel de detalle en un modelo geométrico de mayor complejidad. Se ve cómo la variación en la cantidad de caras no es proporcional con el tiempo que insume el renderizado.

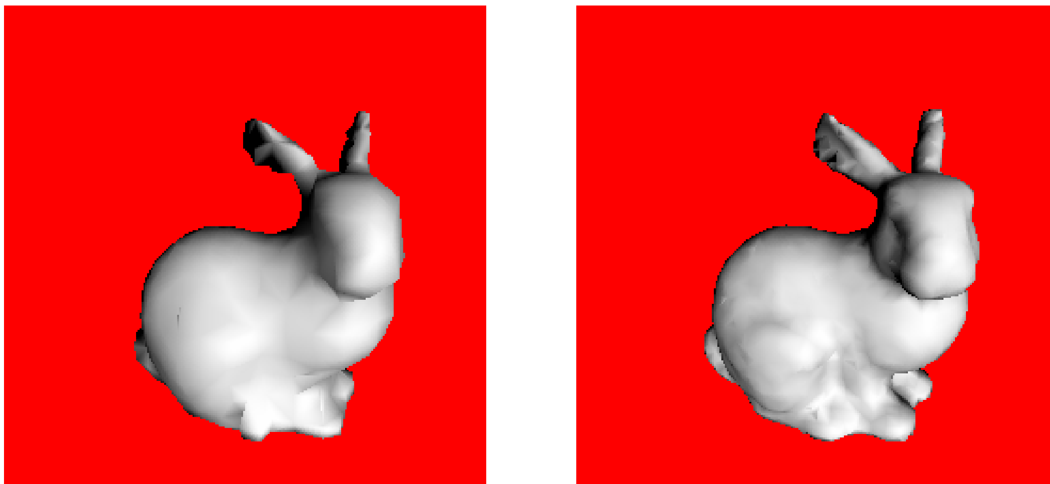


Figura 2: Modelo geométrico con 948 y 3851 caras. Los tiempos fueron 3485 ms. y 6068 ms.

En la Fig. 3 se muestra el efecto de cambiar el modelo de iluminación en el programa que implementa el shader. Estos shaders son fácilmente intercambiables, y no generan una diferencia apreciable en el tiempo de renderizado.

Figura 3: Modelo geométrico con 5030 caras y sombreado difuso, difuso+especular, y difuso, especular y atenuación. El tiempo de renderizado en todos los casos es de 7990 ms.

En la siguiente figura se observa cómo influye la cantidad de iteraciones de reflexión cuando se sitúan objetos que se reflejan entre sí. La primera imagen no tiene reflejo. En la segunda se muestran los reflejos luego de una iteración de los rayos reflejados y en la tercera se agrega la siguiente iteración.

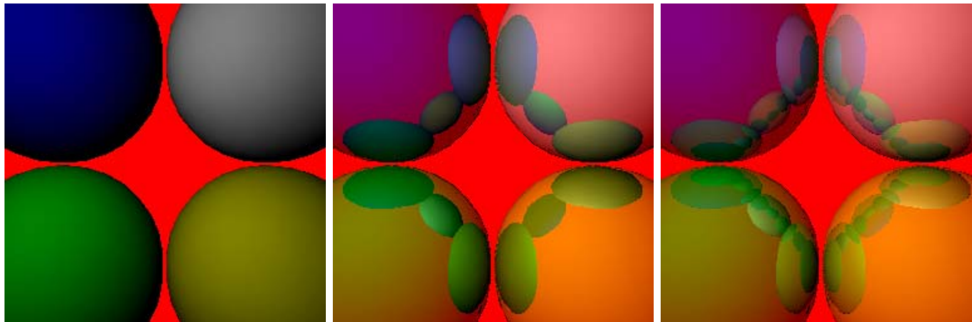


Figura 4: Modelo geométrico con una, dos y tres iteraciones del kernel de trazado de rayos.

Es importante considerar que estos tests se están realizando en una GPU de generación antigua. Una comparación de los tiempos de ejecución de algunas escenas en tres hardware diferentes muestra resultados que es necesario considerar. En la siguiente tabla se comparan tiempos entre CPU y GPUs de dos diferentes generaciones en tres escenas diferentes. Como es esperable, en GPU el tiempo de cómputo es sensible al contexto de la escena. Con la GPU antigua, el *overhead* de la implementación en GPU no alcanza a compensar la ventaja de la paralelización, por lo que los tiempos son similares o peores a los obtenidos con CPU. Por otra parte, el salto de una generación de GPU a la siguiente muestra una reducción en más de 5 veces en los tiempos. Este fenómeno no es sorprendente, dado que la evolución de las GPU supera ampliamente la ley de Moore. Esto permite predecir que con las GPU actuales (que por cuestiones económicas no pudimos adquirir), los tiempos se reducirían 5 veces más (es decir, al

4% de los tiempos originales), y que por lo tanto el objetivo de renderizar escenas razonablemente complejas en tiempo real es factible.

Una consideración adicional importante es que nuestra implementación utiliza los perfiles de shaders de fragmentos compatibles con GF 5700, lo cual no aprovecha las ventajas inherentes a los nuevos perfiles, específicamente el soporte extendido para `render_objects`, multitexturas, y condicionales. Otro cuello de botella que se evitaría con placas más avanzadas es la necesidad de cargar diferentes programas de fragmento para procesar los diferentes kernels, lo cual congestiona el PCI. Una reescritura de los shaders de nuestra implementación para aprovechar estas características determinaría una reducción aún más dramática en los tiempos de cómputo.

Escena	**GPU 5700**	**CPU**	**GPU 6600**
Sphere	2800	2864	500
Bunny3	5075	2013	1079
BunnyRefl	10355	2504	1788

Tabla 1: comparación del tiempo de cómputo en diferentes hardware.

5 Conclusiones

Se mostró una implementación de RT en GPU, la cual muestra tiempos de cómputo interactivos con escenas de complejidad razonable. La implementación se basa en programar a la GPU por medio de *streams* de datos para aprovechar el paralelismo inherente, donde cada *stream* es un conjunto de vectores almacenado en texturas (lo cual aprovecha la aritmética vectorial), y los diferentes *streams* son obtenidos con la aplicación paralela de *kernels* de procesamiento, implementados en programas de fragmento escritos en el runtime del lenguaje CG de NVIDIA. Con el ritmo actual de evolución de los procesadores gráficos, en una o dos generaciones de GPU estos tiempos estarán dentro de lo que podría llamarse RT en tiempo real.

A lo largo del desarrollo del trabajo, nos enfrentamos a algunos problemas o limitaciones. Las herramientas de desarrollo para los programas de fragmentos y vértices no estás muy avanzadas. Por esta razón, en algunos momentos se dificulta la programación y el depuramiento del código. El rápido avance de la tecnología hace que una implementación para una determinada generación de procesadores gráficos quede desactualizada y no sea la óptima para la siguiente generación. Además, el RT requiere mucha precisión en los cálculos y para ciertas operaciones las placas actuales provocan errores de precisión que se traducen en pixels renderizados erróneamente.

Al ser el campo de la programación del hardware gráfico tan novedoso, y al tener el RT tantas variantes, son muchas las posibilidades de expansión y mejoras

- Utilizar múltiples *rendering targets* (MRT), una función de los procesadores gráficos más modernos que permite en un mismo programa de fragmento escribir en más de una textura. Esto permitiría reducir el número de programas necesarios y por lo tanto reducir el tiempo de renderizado.

- Utilizar las máscaras de la GPU para desactivar la ejecución de los programas de fragmento sobre pixels que ya llegaron a un resultado. De esta manera se evitarían ejecuciones y cálculos innecesarios [7, 9].
- Implementar métodos de anti-aliasing para mejorar la calidad de las imágenes obtenidas, evitando los característicos bordes dentados, pixels mal sombreados en las esquinas de los objetos, etc. [5].
- Permitir el renderizado de objetos semi-transparentes que refracten la luz. Esta es una extensión típica del ray tracing.
- Evaluar diferentes estructuras aceleradoras que permitan modificar la escena dinámicamente (bounding volumes, subdivisión adaptativa, etc.) [11].
- Evaluar diferentes métodos de evaluación sobre grupos de pixels (pencil tracing, beam tracing, cone tracing), dado que nuestra implementación es naturalmente extensible para incorporar evaluaciones paralelas [8, 12].

Bibliografía

1. Purcell, Timothy John, Buck, Ian, Mark, William R., Hanrahan Pat. "Ray tracing on programmable graphics hardware", ACM Transactions on Graphics. 21 (3), pp. 703-712, 2002. (Proceedings of ACM SIGGRAPH 2002).
2. Randima Fernando and Mark J. Kilgard. "The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics" ISBN 0-321-19496-9 Addison-Wesley, 2003.
3. Turner Whitted. "An Improved Illumination Model for Shaded Display" Communications of the ACM 23, 6, 343–349.
4. Purcell, Timothy John. "Ray Tracing on a Stream Processor" Ph.D. dissertation, Stanford University, 2004.
5. Genetti Jon and Gordon Dan. "Ray Tracing with Adaptive Supersampling in Object Space" N. Jaffe, editor, Graphics Interface '93, 70-77.
6. Amanatides, J., and Woo, A. "A fast voxel traversal algorithm for ray tracing." In Eurographics '87, 3–10.
7. T. Akimoto, K. Mase, and Y. Suenaga. "Pixel-Selected Ray Tracing. IEEE Computer Graphics and Applications, 11(4):14-22, 1991.
8. John Amanatides. Ray Tracing with Cones. ACM Computer Graphics, 18(3):129-138, 1984.
9. J. Arvo and D. Kirk. Fast Ray Tracing by Ray Classification. Computer Graphics, 21(4):55-64, 1987.
10. A. Fujimoto, T. Tanaka, and K. Iwata. ARTS: Accelerated Ray Tracing System. IEEE Computer Graphics and Applications, 6(4):16-26, 1986.
11. A. S. Glassner. Space Subdivision for Fast Ray Tracing. IEEE Computer Graphics and Applications, 4(10):15-22, 1984.
12. M. Shinya, T. Takanashi, and S. Naito. Principles and Applications of Pencil Tracing. ACM Computer Graphics, 21(4):45-54, 1987.