Raytracing en CUDA

Cabrera López Oscar Emilio

2018-12-05

Introducción

Ray casting y ray tracing

El ray tracing o trazado de rayos es una técnica para la creación de de imágenes por computadora con una calidad de iluminación más realista que la lograda con pasterización. La técnica de ray tracing se basa en el algoritmo de ray casting presentado por Arthur Appel en 1968.

La idea detrás del algoritmo de ray casting es lanzar rayos desde el ojo de un observador y encontrar la intersección con el objeto más cercano, éste objeto es entonces el objeto que ve en el pixel de la imagen que corresponde a ese rayo.

El algoritmo de ray tracing añade recursividad e iluminación al algoritmo de ray casting. La técnica de ray tracing puede generar rayos secundarios, reflejados o refractados, dependiendo de la interacción con el material asociado con la geometría en la que impactó el rayo. Por ello las implementaciones más comunes de ray tracing son recursivas.

El ray tracing tiene la ventaja de poder crear iluminación realista en cualquier geometría para la que se pueda definir una función de intersección con un rayo.

CUDA

CUDA (antes compute unified device architecture), es una plataforma de desarrollo de aplicaciones de computo paralelo desarrollada por Nvidia.

Nvidia es una empresa especializada en el desarrollo de hardware y software dedicado a los gráficos por computadora. Desarrolla CUDA como una plataforma que permite el computo paralelo usando únicamente sus GPUs. Permite la programación de código tanto de lado del CPU (host) como del GPU (device).

A los programas que se ejecutan de forma paralela en el GPU se les llama kernels.

Objetivos

- Crear un renderizador de una escena aplicando técnicas de graficación por ray-tracing.

Desarrollo

Antes de desarrollar el proyecto se investigó más a fondo el planteamiento del ray tracing, las diferentes técnicas existentes y los enfoques que se pueden dar en la creación de imágenes.

Durante esta investigación se encontró el libro de Raytracing in a weekend de Peter Shirley, perteneciente a una serie de 3 libros donde desarrolla de forma incremental la infraestructura para la creación de un programa de ray tracing cada vez más serio.

Debido a la sencillez del planteamiento en sus libros se decidió replicar este desarrollo, pero añadiendo técnicas de cómputo paralelo en el camino.

En los libros de Peter Shirley el programa se desarrolla en C++ sin ningún tipo de paralelismo y usando recursividad. Debido a las limitaciones de CUDA, un planteamiento recursivo no es posible, debido al tamaño limitado del stack de llamadas dentro de cada núcleo de la GPU.

Por ello se tuvo además del reto del desarrollo del ray tracing, la adaptación de los algoritmos presentados en el libro a un modelo de cómputo paralelo y iterativo en lugar de recursivo.

Otros caminos posibles que se consideraron fue hacer uso de la infraestructura ya existente dentro de OpenGL, aprovechando el paralelismo que brindan los shaders de OpenGL.

Los shaders son pequeños programas que se pueden ejecutar en diferentes etapas del pipeline de OpenGL. Los shaders más usados son los Vertex shaders, que se ejecutan por cada vértice en la geometría; los fragment shaders, ejecutados por cada fragmento generado después de la pasterización; y los compute shaders, estos últimos son los más recientes, y pueden ejecutarse en cualquier etapa del pipeline e incluso reemplazar por completo el pipeline de OpenGL, implementado por si mismo un pipeline.

Este enfoque brinda la capacidad de aprovechar una bases de código bastante estables como glut o glm para dibujar la geometría, realizar transformaciones, aplicar texturas, materiales y manejar bufferes en la GPU.

Sin embargo fue un opción que se exploró después del comienzo del proyecto, y el costo de portar el código de CUDA a OpenGL se consideró demasiado alto, y el tiempo muy reducido.

Una ventaja de seguir los libros Peter Shirley es que su sencillez ha llamado la atención de muchas personas, entre ellos el ingeniero de Nvidia Roger Allen, que en uno de sus blogs da una introducción a la programación en CUDA portando el código del primer libro a CUDA. Esto sirvió como una gran introducción a CUDA y nos permitió reusar parte de su código para enfocarnos en los problemas más complejos e interesantes de los otros 2 libros.

Problemas y soluciones

Para comprender la estructura del programa presentado se recomienda la lectura de al menos el primer libro de Peter Shirley, Raytracing in a Weekend.

Correccion gamma

Uno de los primeros problemas que se presentó fue la distorsión aparente de los colores en las imágenes generadas. Esto se debe a que la mayoría de los visores de imágenes esperan que las imágenes mostradas tengan previamente aplicada una corrección gamma.

La corrección gamma es una técnica que se usa para dar cierta ponderación a los colores de la misma forma que lo hacen los ojos de las personas. La técnica de corrección gamma más sencilla involucra elevar cada componente de color al inverso de un factor gamma, que usualmente vale 2.2, entonces cada componente de color se eleva a la
$frac12.2$. Para hacer los cálculos mas rápidos se uso una aproximación de gamma a 2, de esta forma solo hay que obtener la raíz cuadrada de las componentes de color.

Suavizado

Otro problema que se encontró fue una apariencia de la imagen demasiado cuadrada, esto se debe a que no se estaban suavizando las imágenes durante el trazado de rayos. En el libro el autor aborda este problema usando un muestreo de múltiples rayos por pixel, usando una distribución uniforme entre 0 y 1 para enviar rayos alrededor del centro del pixel a dibujar. Sin embargo en nuestro programa usamos una distribución normal con media 0 y desviación estándar de 0.3 para muestrear no solo en el pixel, sino también en sus alrededores, para suavizar los cambios en los colores sin desenfocar demasiado la imagen. La elección de estos valores fue a través de prueba y error.

Materiales

Probablemente la parte más interesante e ilustrativa del proyecto ha sido el modelado de los materiales y su interacción con los rayos.

Inicialmente se consideraron para los materiales 3 tipos de interacciones:

- Material difuso (acabado mate) - Material especular (brillo metálico) - Material dieléctrico (cristal)

Material difuso

De los materiales modelados el mas sencillo fue el material difuso ya que únicamente refleja los rayos en direcciones aleatorias dentro una media esfera.

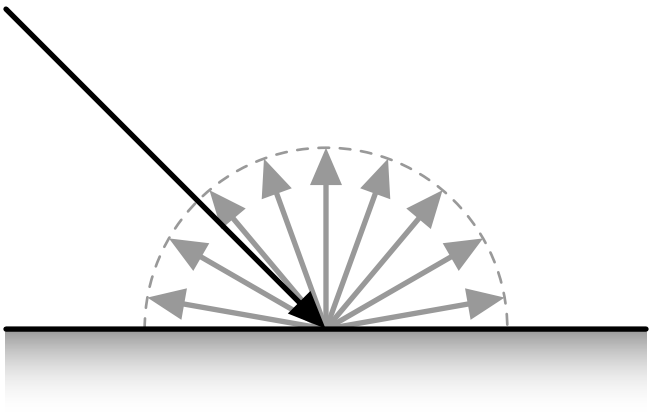

Figure 1: Modelo de un material difuso

Un modelo más cercano a la realidad absorvería algún porcentaje de rayos de forma aleatoria en lugar de solo reflejarlos aleatoriamente. Un modelo que aproxima la cantidad de rayos obsorvidos o reflejados y sus direcciones es la función de densidad de probabilidad de reflectancia bidireccional.

Una gran introducción al ray tracing usando métodos probabilísticos para la refracción, reflexión y generación de rayos es el tercer libro de la serie donde se introduce al método de Montecarlo y se generan sombras usando importance sampling en lugar de rayos de sombra.

Material especular

Para el material metálico se simuló como un material perfectamente reflectivo al principio. Posteriormente se añadió también algo de aleatoriedad a la reflexión

de los rayos de forma que apareciera cierta distorsión en la superficie metálica como la observada en el aluminio anodizado.

Este ultimo efecto se logra reflejando los rayos con una pequeña desviación alrededor del rayo de la reflexión perfecta, en este caso los rayos se desviaban dentro de una esfera cuyo radio es controlado por el parámetro fuzziness dentro de la clase specular.

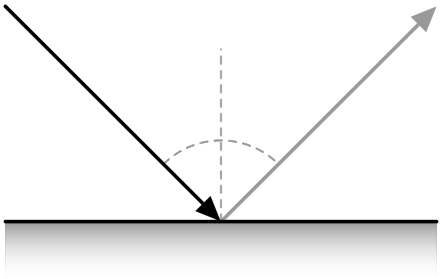

Figure 2: Reflexión perfecta

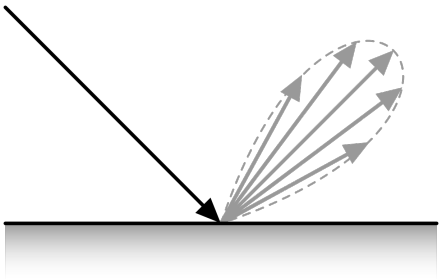

Figure 3: Reflexión distorsionada

Por ahora un material solo puede ser especular o difuso, implementar un material que se comporte como ambos podría hacerse de forma sencilla, simplemente decidiendo de forma aleatoria por cada rayo incidente si el material se comportara de forma difusa o especular.

Material dieléctrico

El tercer material es el material dieléctrico, este material puede refractar o reflejar la luz. Este fue definitivamente el más interesante de los 3.

Para modelar este material se tiene que recurrir a la Ley de Snell. La cual pone varias condiciones a la reflexión o refracción de la luz dependiendo del ángulo de incidencia y de los índices de refracción de los medios que atraviesa el rayo. Para entender mejor como funciona es mejor revisar en el código fuente, el

Figure 4: Brillo metálico en el aluminio anodizado

encabezado material.cuh, en la clase dielectric se da un resumen del algoritmo usado y las consideraciones tomadas, también se recomienda leer el articulo de Wikipedia en inglés de la ley de Snell.

Una consideración adicional a la ley de Snell es que los materiales cambian su refractancia con el ángulo en el que se miran. Que tanto cambia depende del coeficiente de Fresnel del material. Para modelar este efecto se uso la aproximación de Schlick, una técnica común en los gráficos por computadora.

Una mejora a este material seria añadir la posibilidad de definir una textura también en ellos con el fin de añadir el efecto de aberración cromática, como la del agua, en la que los colores refractados y reflejados son mas azules.

Vectores aleatorios

Como ya se discutió, para el modelado de los materiales, se usaron vectores aleatorios en varias ocasiones. y aunque al principio parace trivial generar vectores aleatorios, este problema se vuelve complicado cuando se desea una distribución uniforme dentro de una esfera unitaria.

Uno de los apartados que requiere optimización en el programa es la generación de estos vectores, por ahora ocupa el método de Montecarlo (o método de descarte) para generarlos. Básicamente consiste en crear vectores aleatorios dentro de un cubo unitario usando una distribución uniforme, y descartar aquellos vectores que no se encuentren dentro de la esfera unitaria inscrita en el cubo unitario.

Si recordamos que la relación entre el volumen de una esfera inscrita en un cubo es de $\frac{\pi}{6}$ vemos que aproximadamente la mitad de los rayos generados se descartan. En la computadora generar números aleatorios es costoso, y este es uno de los apartados del programa que mas necesitan atención.

Una solución sencilla y la mas eficiente seria generar vectores aleatorios con una distribución normal en lugar de con una distribución uniforme. Se puede ver la

comparación de los algoritmos en este paper.

Subdivisión del espacio

Hasta antes de introducir esta característica el programa aplicaba un ray tracing por fuerza bruta. Es decir enviaba millones de rayos a la escena y la mayoría de ellos no impactaba en ningún modelo. Esto es sumamente ineficiente, por ello se han desarrollado técnicas de búsqueda en la escena con el fin de enviar rayos solo a aquellas partes de la escena donde realmente se va a golpear un objeto.

En el segundo libro de Peter Shirley se añaden varias estructuras de datos para poder organizar el espacio usando la técnica de Bounding Volumes Hierarchies. Esta técnica consiste en generar cajas que contienen a un modelo y otras cajas que contienen a otro modelo y así sucesivamente hasta terminar en una caja que contiene a toda la escena. Esto genera un árbol binario que una vez ordenado, se puede usar como un árbol binario de búsqueda, haciendo de esta forma la complejidad de gran parte del programa logarítmica en lugar de exponencial.

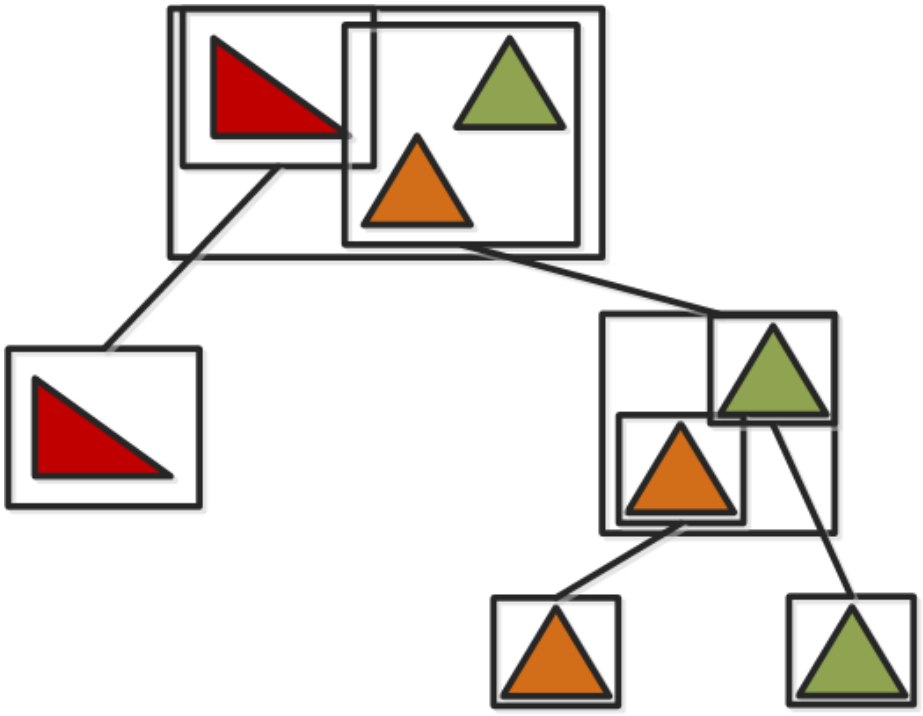

Figure 5: Ejemplo de la organizacion de la geometria usando BVH

Para organizar los objetos se usan cajas delimitadoras alineadas con los ejes (axis-aligned bounding boxes).

Una mejora al código actual seria construir la escena en un kernel de CUDA, y en otro kernel ordenar el árbol usando la función qsort que provee CUDA. En este momento qsort es llamado dentro del kernel que construye el mundo, el cual tiene solo un hilo de ejecución.

Texturas

Las texturas solidas fueron sencillas de implementar ya que siempre devuelven el mismo color sin importar las coordenadas (u, v). Sin embargo las texturas de imágenes dieron problemas en la parte de programación, ya que la imagen se carga en el CPU y posteriormente debe ser movida a la memoria del GPU.

Hay dos opciones para lograr esto, la primera es generar un espacio de memoria compartida entre la CPU y GPU que es lo que se hizo en el programa, para posteriormente copiar los datos de la imagen a este segmento de memoria y luego sincronizar los dispositivos.

Otra forma de hacerlo que usa menos memoria es apartar espacio solo en la memoria del GPU y hacer una copia usando cudaMemcpy, sin embargo no se logro hacer funcionar este método.

Luces

Los dos tipos de luces solicitados en el proyecto fueron spotlight y pointlight. El primer tipo es una luz cuyos rayos son siempre paralelos entre sí, el segundo es una luz que emite en todas direcciones.

Hay dos enfoques tradicionales para generar luces y sombras. El primero es enviar rayos hacia la fuente de luz (rayos de sombra), el segundo es enviar rayos desde la fuente de luz. En nuestro programa buscábamos emplear un enfoque híbrido, ya que enviar rayos desde la fuente de luz genera luces muy intensas y sombras muy débiles y viceversa. Sin embargo, el tiempo no lo permitió.

Para ver otros enfoque más modernos para generar sombrar vea el tercer libro de la serie de Peter Shirley.

Implementar sombreado usando rayos de sombra no es complicado con el diseño actual del programa, la forma mas sencilla sería (en la función color o en la función hit de los objetos que heredan de hitable) elegir de forma aleatoria entre calcular el color de la superficie como ahora lo hace o generar un rayo de sombra en dirección a cada una de las luces en ese punto y verificar por cada luz si el rayo intersecta a alguna, si no lo hace, se resta una cierta cantidad de luz al color calculado de acuerdo con la distancia a la fuente de luz. La infraestructura para crear listas de luces ya existe (ver hitable_list.cuh) y el código seria similar al usado para verificar intersecciones en el mundo.

Para crear luces se añadió un nuevo material cuya textura emite luz en lugar de atenuarla. Además puede valer más de 1 en cualquiera de sus componente, para dar mas intensidad a la luz.

Salida del proyecto

En este momento el código genera la imagen en formato PPM e imprime el archivo en la salida estándar. Una mejora que se busca hacer es usar la biblioteca stb_image_write para generar una salida en formato JPG o PNG.

Construcción del proyecto

Para construir el proyecto es necesario instalar CUDA y git en Linux.

En una terminal escribir lo siguiente:

```
$ git clone https://github.com/emilio1625/raytracing.git
$ cd src
$ make
$ time ./cudart > test.ppm
$ eog test.ppm &
```

La salida debería ser similar a esta:

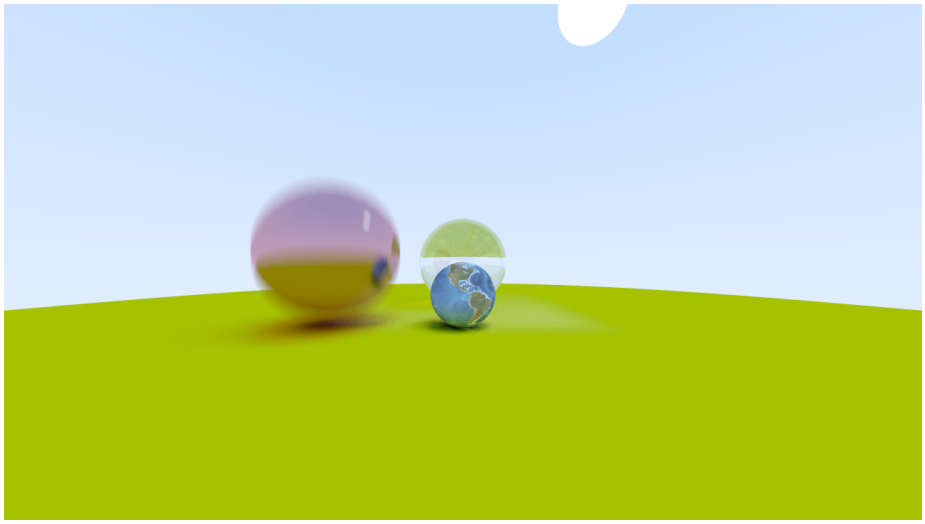

Figure 6: Salida del programa

Conclusiones

Este proyecto ha sido uno de los más interesante de mi carrera y de los más ilustrativos. En él he logrado puesto a prueba no solo mis conocimientos en computación gráfica sino también de probabilidad, óptica, computo paralelo y gestión de memoria y otro temas que de no ser por el proyecto jamás me habrían interesado o habría sabido de ellos. Aun hay muchas cosas que me gustarían agregar a este proyecto, como la posibilidad de cargar modelos obj o stl, o integrarlo en OpenGL para aprovechar la posibilidad de crear transformaciones y rotaciones en los modelos de la escena. También hay muchísimas áreas en las que se pueden optimizar la cantidad y la complejidad de los cálculos realizados.

Si se desea revisar el código desarrollado, éste se encuentra en Github en este enlace. Esta liberado bajo una licencia CC0, equivalente a dedicar este trabajo, el código y su documentación al dominio publico.