Implementación de algoritmos para luces y sombras dinámicas usando la GPU

Implementation of algorithms for dynamic light and shadow using GPU

Lien Muguercia, Yaima Nodarse

Universidad de las Ciencias Informáticas

{lmuguercia, ynodarse}@uci.cu

Resumen

Tradicionalmente lograr la iluminación y el sombreado eficiente ha sido un tema de discusión en los gráficos por computadora.
Lograr el nivel de realismo deseado requiere de un alto consumo de recursos y a veces no se cumple con las expectativas.
Este trabajo propone una solución para incorporar mayor realismo a las escenas virtuales a través de luces y sombras dinámicas utilizando la GPU para ello.
Partiendo del estudio de varios algoritmos que dan solución al problema de la iluminación y sombreado dinámico de modo eficiente, se proponen dos algoritmos para el trabajo con luces y sombras a través de los *shader* (códigos que permiten introducir nuevos algoritmos en el hardware de video) y su implementación en la GPU.

Palabras clave: Dinámicas, GPU, *shader*, hardware de video.

Abstract

Traditionally, efficient shading and light mapping has been an obstacle in computer graphics.
Reaching desired realism levels requires a high source consume and sometimes it doesn't accomplish existing expectative.
The intention of this work is adding more realism to virtual scenes through dynamic light and shadow using GPU.
Starting off of the study of many algorithms, it proposes two algorithms to work whit light and shadow through shader (program used to determine the final surface properties of an object or image. This can include arbitrarily complex descriptions of light absorption and diffusion, texture mapping, reflection and refraction, shadowing, surface displacement and post-processing effects).

Key words*: GPU, shader, post-processing effects.*

Introducción

La tecnología ha progresado más rápido que nuestra habilidad para siquiera imaginar que vamos a hacer con ella. Hoy la Realidad Virtual (RV), ha trastocado la percepción y está revolucionando el mundo, no sólo de la informática sino también de diversidad de áreas como la medicina, la arquitectura, la educación y la ingeniería entre otros.

La RV entra en un exclusivo rango de herramientas para hacer, en el cual el usuario puede incursionar creativamente, hasta donde el límite de su imaginación se lo permita. Allí radica muy posiblemente el mayor atractivo, por cuanto la imaginación y la creatividad tienen la oportunidad de ejecutarse en un "mundo" artificial e ilimitado.

Para lograr una alta sensación de credulidad en este mundo ilimitado pero artificial, se hace necesario que esté correctamente iluminado tal que su semejanza con el mundo real sea lo más certera posible.

Tradicionalmente lograr esto de manera eficiente ha sido un tema de discusión en los gráficos por computadora.

Lograr el nivel de realismo deseado requiere de un alto consumo de recursos y a veces no se cumple con las expectativas.

El objetivo principal del trabajo es, después de un largo estudio, proponer dos algoritmos que den respuesta a la dinamicidad de las luces y sombras pero que los cálculos se realicen sobre la GPU; proponer su implementación mediante el lenguaje GLSL y utilizando la API Opengl.

Desarrollo

Shader

Gracias al avance en el hardware gráfico, el rendimiento de las aplicaciones de visualización se ha podido incrementar notablemente permitiendo representar escenas virtuales de mayor tamaño y complejidad. Sin embargo, aunque el aumento en el rendimiento ha posibilitado ejecutar las secuencias de visualización en un tiempo cada vez menor, se puede argumentar que al mismo tiempo estos sistemas no se han desarrollado en cuanto a tecnologías de visualización.

La limitante fundamental del hardware de gráfico es que su estructura no se puede cambiar. Cuando se crean estos hardwares, los ingenieros prefijan un conjunto de instrucciones y algoritmos en el chip de video. Cada uno de estos algoritmos es acelerado por el hardware gráfico. Pero no se brinda la posibilidad de ejecutar en estos chips otras instrucciones que no sean las codificadas en el momento de la creación del hardware. A esta estructura estática se le denomina sistema de funciones fijas (*Fixed-Function*). [3]

En los últimos 5 años se ha elaborado una alternativa al *Fixed-Function* para aumentar la flexibilidad gráfica del hardware de video. En dos momentos claves del *pipeline* gráfico se ha permitido introducir códigos que permitan ejecutar algoritmos no diseñados inicialmente en el hardware. A estos códigos se le llaman *shader*, y según su funcionamiento y lugar de ejecución se dividen en *Vertex Shader, Geometry Shader* y *Píxel Shader*.

Los ***Vertex Shader*** son los encargados de transformar todos los vértices de la escena. En ellos se ejecutan las transformaciones de espacio objeto a espacio de mundo, de cámara, y finalmente se obtiene la posición en la pantalla. Además, en ellos se incluyen todas las demás operaciones a nivel de vértices como son los cálculos procedurales de coordenadas de texturas, iluminación *per-vertex*, entre otras.

Los ***Geometry Shader*** pueden generar nuevas estructuras primitivas a partir de primitivas existentes como pixeles, líneas y triángulos. Este es ejecutado después del *vertex shader* y su entrada es completa primitiva o primitiva con información de adyacencia. Un uso muy común incluye generación de *point sprite*, teselacion, calculo de sombra volumétrica y pasadas simples de *rendering* para el *cube map*. Por ejemplo, cuando trabajamos con triangulos, tres vértices son entradas para el *geometry shader* el cual puede generar cero o mas primitivas, estas son rasterizadas y el resultado de la interpolación es pasado al ***Pixel Shader*** donde se realizan todas las operaciones a nivel de píxel como es el cálculo de la iluminación *per-píxel*, mapeo de texturas, uso de los mapas de normales para la determinación de la normal del píxel, etc.

Soluciones de Iluminación

Contando con los modelos matemáticos para describir el comportamiento de la luz y el foco luminoso, es posible calcular la iluminación resultante en un objeto. Para esto se pueden tomar dos vías fundamentales: el cálculo de iluminación puede ser realizado para cada vértice del objeto o para cada píxel de este. [5]

Iluminación Per-Vertex

Cuando el cálculo se realiza en los vértices la iluminación se denomina "*Per Vertex Lighting*". Pues las ecuaciones que describen la contribución de la luz en el objeto son evaluadas en cada vértice de la geometría y luego para cada triángulo de ésta, se interpola el valor de la influencia en los vértices y se genera la contribución en cada píxel.

La figura 1 muestra un objeto al que se le ha iluminado usando esta técnica. Como se aprecia, la luz se nota un tanto distorsionada, pues la interpolación que ocurre para cada vértice produce gradientes de colores más oscuros o más claros de los que realmente describen la influencia de la luz.

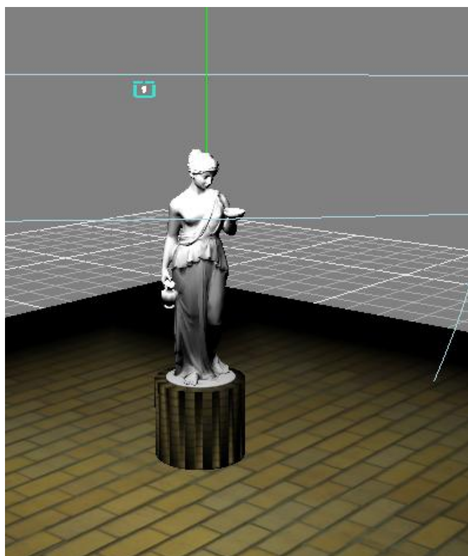

Fig. 1: Iluminación "Per-Vertex"

Es de destacar que la solución *per vertex* tiende a mostrar resultados más precisos y con menos errores de aproximación a medida que los objetos iluminados son descritos con mayor cantidad de polígonos (presentan una mayor teselación).

Lo interesante de usar *per vertex lighting* es que el pipeline fijo contiene una implementación de esta, donde es posible calcular la contribución de hasta 8 luces en los vértices de un objeto. De ahí que las soluciones dirigidas al cálculo por vértices generalmente puedan ser ejecutadas en un mayor número de sistemas gráficos.

Iluminación Per-Píxel

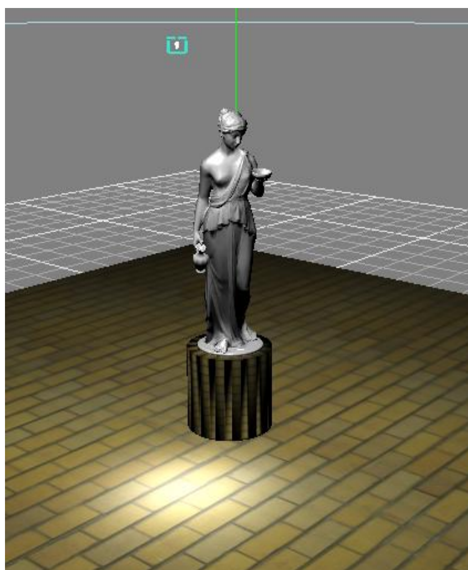

Fig. 2: Iluminación "Per-Píxel"

Contrario a la iluminación "*per-vertex*", este método ejecuta las ecuaciones directamente en el píxel. De ahí que los resultados sean más precisos, pero el proceso total de iluminación tiende a ser más lento pues las ecuaciones son calculadas muchas veces por cada triángulo en lugar de tres veces (una por cada vértice) y luego interpolar el resultado para cada píxel.

La figura 2 muestra una escena iluminada *per-píxel*, como se aprecia en esta figura no aparecen errores de interpolación ni brillos en los bordes de los polígonos. Sin embargo, la iluminación "*per-píxel"* tiene el inconveniente que de manera estándar no se encuentra implementada en el *pipeline* fijo, de ahí que haya que usar los *shaders* para introducir esta funcionalidad en las tarjetas de video.

Mapas de Luces

De manera general la iluminación *per píxel* produce resultados de mayor calidad en comparación con la iluminación *per vertex*. Sin embargo, el cálculo de esta requiere mayores recursos y necesita evaluar la ecuación de iluminación muchas más veces que la en la iluminación *per vertex* de ahí que los tiempos de procesamiento de la luz sean mayores.

Cuando existen un número de luces y objetos estáticos, es posible precalcular la influencia de las luces en estos, siendo viable almacenar en una textura llamada mapa de luz (*Light MAP*) la contribución luminosa en la superficie y las sombras generadas por otros objetos en ésta.

El uso de mapa de luces reduce notablemente el número de cálculos a realizar para iluminar la escena pues ya se cuenta con el valor resultante de la iluminación en los píxeles de los objetos, por tanto no hace falta evaluar la ecuación de iluminación en cada píxel. Sin embargo se mantiene el carácter de iluminación per píxel.

Iluminación con Esferas Harmónicas

En 2001, *Ravi Ramamoorthi* y *Pat Hanrahan* presentan un método que usa esferas harmónicas para calcular la llamada iluminación difusa. Este método utiliza reflexión difusa, basada en el contenido de una imagen de luz sin acceder a ella en tiempo de procesamiento. La imagen de luz antes de ser procesada produce coeficientes que son usados en una representación matemática de la imagen en tiempo de procesamiento. El resultado es notablemente simple, certero y realista, y puede ser fácilmente codificado en un shader de OpenGL.

La iluminación con Esferas Harmónicas (*Spherical Harmonics Lighting*) proporciona una representación de un espacio periódico de una imagen sobre una esfera. Esta representación es seguida a través de rotaciones invariantes. Usando esta representación por una imagen de luz, se puede reproducir exactamente la reflexión difusa desde una cara con solo nueve funciones básicas de esferas harmónicas. Éstas son obtenidas con constantes, lineales, y polinomios cuadráticos de la superficie normalizada.

Cada función básica de esferas harmónicas posee un coeficiente que depende de la imagen de luz que se esté usando. Los coeficientes son diferentes para cada color de canal, así se puede pensar en cada coeficiente con un valor RGB. Un paso del procesamiento por adelantado requiere calcular los nueves coeficientes RGB para que la imagen de luz sea usada.

$$\text{Difuso} = c_1 L_{22} (x^2 - y^2) + c_3 L_{20} z^2 + c_4 L_{20} - c_5 L_{20} + 2c_1 (L_{2-2}\, xy + L_{21}\, xz + L_{2-1}\, yz) + 2c_2 (L_{11}\, x + L_{1-1}\, y + L_{10}\, z$$

El *vertex shader* que codifica la fórmula para las nueve funciones básicas es actualmente bastante simple. Un compilador optimizado reduce todas las constantes involucradas en las operaciones. El resultado es bastante eficiente porque contiene un pequeño número de suma y operaciones de multiplicación que involucran los componentes de la normal del plano.

Así el *fragment shader* hace muy poco trabajo; porque típicamente la reflexión difusa cambia lentamente, para escenas de grande polígonos se calcula razonablemente el *vertex shader* e interpola durante el proceso de la imagen.

Mapeo de Sombras

La oclusión ambiental es muy eficiente para resaltar el realismo dirigido a objetos bajo condiciones de iluminación difusa, pero a menudo la escena necesita incorporar iluminación desde uno o más fuentes de luces bien definidos. En el mundo real, se sabe que una fuente de luz fuerte causa en los objetos una sombra a su semejanza. Produciendo similares sombras en la computadora, las escenas generadas tomarán mayor realismo.

En este algoritmo, la escena es rendereada en múltiples ocasiones para cada luz que es capaz de dar sombra y una vez para generar la escena final, incluyendo la sombra. Cada paso de la luz es rendereado desde cada punto de luz. Los resultados son almacenados en una textura llamada Mapa de Sombra (*Shadow Map*) o Mapa de Profundidad (*Depth Map*). Esta textura esencialmente representa una superficie visible de un punto de luz. Estas superficies son completamente iluminadas por una fuente de luz, no son

visibles desde un punto de luz que esté en la sombra. Cada textura generada es accedida en el último paso del proceso de render para crear la escena final con sombras desde más luces. [1]

Puesto que este algoritmo involucra un rendereado extra para cada fuente de luz, su rendimiento depende del número de luces que halla en la escena. Pero para aplicaciones interactivas, con dos luces adicionadas para el realismo y comprensibilidad de la escena es eficiente.

Si se adiciona más de dos luces puede darle complejidad a la escena y a la vez quitarle realismo. Como otros algoritmos que usan texturas, el mapeo de sombras está propenso a severos problemas de *aliasing* (líneas, especialmente las que están casi horizontales o verticales, que aparecen dentadas o irregulares debido a su representación por *píxeles*) a menos que se tome el cuidado apropiado.

La comparación entre aplicaciones interactivas puede también conducir a problemas. Desde que los valores son comparados son generados en distintos pasos con diferentes transformaciones matriciales, por lo tanto, debe usar un valor épsilon en la comparación. [1] Por ejemplo, en OpenGL se puede usar el comando glPolygonOffset para predefinir los valores de profundidad cuando el Mapa de Sombra es creado.

El camino para evitar problemas de precisión de profundidad con caras iluminadas es dibujar caras traseras cuando se esté construyendo un Mapa de Sombra. El valor de profundidad para estas superficies es normalmente bastante diferente desde el Mapa de Sombra.

A pesar de los inconvenientes, el MS es aun popular y efectivo en la generación de sombras.

Conclusiones

El mundo de los gráficos por computadora es muy lindo y rico en información, basta solo con querer para poder hacer.

Como diariamente se proponen nuevas técnicas para darle solución a distintos problemas como son lograr una iluminación y sombreado realista, lograr alta eficiencia de manera que el *render* no sufra retardos y otros tantos problemas derivados de ello, este trabajo solo propone la implementación de dos algoritmos que si bien no logran ambos propósitos, por lo menos el realismo y semejanza con el mundo real está garantizado.

Aunque no se tocó en esta investigación queda como recomendación el estudio y análisis del *Deferred Shading*, algoritmo muy eficiente para lograr iluminación dinámica ahorrando trabajo en el *render* de la escena, el cual también tiene una variante para implementar sombras dinámicas. En trabajos futuros se expondrá la investigación y explicación detallada de este algoritmo.

Referencias Bibliográficas

[1]. J. Rost, Randi. *OpenGL® Shading Language, Second Edition*, 2006.

[2]. Caro, Zamir. *Realidad Virtual*, 2005.[http://www.monografias.com/trabajos28/realidad-virtual/realidad-virtual.shtml#element]

[3]. *Shader*, 2006. [http://en.wikipedia.org/wiki/Shader]

[4]. ASTLE, Dave y HAWKING, Kevin. *OpenGL Game Programming.* USA, Prima Tech Publishing. 2001.

[5].Puig Placeres, Frank. *Empleo eficiente del hardware gráfico en la iluminación de entornos virtuales*. Universidad de las Ciencias Informáticas. Ciudad de La Habana, 2006. 106.