UNIVERSIDAD DE LAS CIENCIAS INFORMÁTICAS

EMPLEO EFICIENTE DEL HARDWARE GRÁFICO EN LA ILUMINACIÓN DE **ENTORNOS VIRTUALES**

TRABAJO DE DIPLOMA EN OPCIÓN AL TÍTULO DE
INGENIERO INFORMÀTICO

AUTOR

Frank Puig Placeres

TUTOR

MsC. Jose Ignacio Guzmán Montoto

Ciudad de la Habana
Abril 2006

A mis padres…

“Hay cosas grandes, muy grandes. Hay otras pequeñas, muy pequeñas
Pero en ocasiones, las pequeñas hacen que las cosas
Se vean más grandiosas”

Walt Disney

AGRADECIMIENTOS

A mi madre por su apoyo incondicional

A mi padre y hermano por exhortarme a ser cada día mejor.

A Yani por ayudarme a conseguirlo.

A Pepe, por supervisar cada uno de mis pasos.

A Osdalme, por demostrarme que el corrector del Word no es perfecto.

Y a aquellas otras personas que de una forma u otra hicieron posible este trabajo, pero que mi clásico despiste y no mi falta de gratitud, me impiden poner aquí…

A todos ustedes, Gracias.

Declaración de Autoría

Declaro que soy el único autor de este trabajo y autorizo a la Universidad de Ciencias Informáticas (UCI) para que hagan el uso que estimen pertinente con este trabajo.

Para que así conste firmo la presente a los 5 días del mes de abril del 2006.

_____	_____
Frank Puig Placeres	Jose Ignacio Guzmán Montoto
Autor	Tutor

Opinión del Tutor del Trabajo de Diploma

Título: Empleo eficiente del Hardware Gráfico en la iluminación de Entornos Virtuales

Autor: Frank Puig Placeres

Tutor: Jose Ignacio Guzmán Montoto

El tutor del presente Trabajo de Diploma considera que durante su ejecución el estudiante mostró las cualidades que a continuación se detallan.

Por todo lo anteriormente expresado considero que el estudiante está apto para ejercer como Ingeniero Informático; y propongo que se le otorgue al Trabajo de Diploma la calificación de _____

_____ _____

Firma

RESUMEN

Tradicionalmente la Iluminación de escenas virtuales en tiempo real se ha implementado usando módulos especializados llamados *shaders* que calculan el brillo y color generado por las luces en cada superficie. Estos valores son combinados mediante algoritmos de procesamiento de textura para producir materiales que muestren la iluminación. Una de las principales dificultades es que actualmente los shaders no pueden calcular la influencia de un número alto de luces simultáneamente por lo que se procesa sólo un número limitado de estas.

Para mostrar una mayor cantidad de luces en la escena se dividen estas en pequeños grupos que puedan ser manejados independientemente por los shaders y luego al combinarlos forman la iluminación total de la escena. Sin embargo el cálculo de la iluminación requiere que se procese la escena cada vez que se calcula un grupo de luces lo que reduce enormemente el rendimiento cuando las escenas presentan cierta complejidad. Esta clase de algoritmo multi-pasada produce resultados aún más lentos cuando existe un elevado sobrepintado de la escena debido a que la iluminación es calculada para píxeles que no van a mostrarse en la imagen final.

Una alternativa para el cálculo de la iluminación es el uso del *sombreado retardado* (deferred shading). Este método elimina el alto costo que conlleva procesar varias veces la escena para calcular la iluminación, sin embargo exige recursos de programación de shaders, sólo presentes en las tarjetas de video de ultima generación.

Este trabajo propone un conjunto de técnicas para simplificar el *deferred shading* de forma que pueda ser usado en sistemas gráficos con menores prestaciones. Estas técnicas permiten acelerar los cálculos de iluminación en escenas con un elevado sobrepintado, lo cual facilita la simulación sobre mundos virtuales altamente complejos.

INDICE

LISTA DE FIGURAS

CAPÍTULO 1

CAPITULO 2

CAPITULO 3

CAPÍTULO 4

EMPLEO EFICIENTE
del
HARDWARE GRAFICO
en la ILUMINACION de
ENTORNOS VIRTUALES

UCI

Frank Puig Placeres

INTRODUCCION

Desde el surgimiento de las computadoras, el hombre ha aprovechado sus altas prestaciones para representar y simular situaciones del mundo real. El perfeccionamiento del hardware y la implementación de modelos matemáticos avanzados han estimulado el desarrollo de estas representaciones virtuales a gran escala, siendo representados diversos entornos de forma realista; permitiendo al espectador no sólo explorar, sino interactuar con el mundo.

Para la visualización de estos entornos se han utilizado las potencialidades que ofrece la Unidad de Procesamiento Gráfico (GPU) que se encuentra localizada en la tarjeta de video y consiste en un procesador especializado para el tratamiento gráfico, mediante la implementación de rutinas y algoritmos para la transformación y proyección matemática de vértices en 2 y 3 dimensiones, el procesamiento de imágenes y la combinación de texturas, entre otras. Al conjunto de rutinas y algoritmos implementados de forma estándar en estos procesadores se le llama “Tubería de Función Fija” (*Fixed Function Pipeline*).

En los últimos años se ha producido un gran avance tecnológico en el hardware gráfico; las *fixed function pipelines* han sido remplazadas por las tuberías de funciones programables (*Programmable Function Pipeline*) mediante las cuales es posible implementar nuevas rutinas ha ser ejecutadas en el GPU. El uso de estas nuevas tecnologías ha permitido romper las barreras clásicas impuestas por las limitadas funciones implementas en el Procesador Gráfico al permitir personalizar la forma en que serán manejadas las transformaciones de vértices y la determinación del color de cada píxel mediante el uso de los *Vertex Shaders* y los *Píxel Shaders*. Estos son programas que se ejecutan en el GPU (a diferencia de las aplicaciones comunes cuya ejecución se realiza en el CPU) y controlan las dos etapas principales del procesamiento de gráficos tridimensionales.

El surgimiento de los *Programmable Function Pipelines* ha producido a su vez un cambio en la investigación de las aplicaciones de visualizaciones en tiempo real; buscándose no sólo un mayor rendimiento en el procesamiento masivo de polígonos, sino también en la calidad de los materiales con que estos se representan. De esta forma, efectos de render tales como *antialiasing*, desenfoque por movimiento y proyección de sombras comienzan ha ser comunes en las simulaciones.

La iluminación es otro de los efectos visuales que se ha beneficiado con los avances tecnológicos. Unos años atrás las simulaciones virtuales típicas sólo implementaban iluminación estática mediante el uso de mapas de luces (*Light maps*) y algunos objetos iluminados “per-vertex”. Sin embargo, los gráficos se han desplazado al procesamiento de mundos completamente iluminados “per-pixel”, existiendo cada día más luces dinámicas.

Los algoritmos clásicos para la iluminación que son capaces de procesar luces dinámicas comúnmente involucran dibujar los objetos sobre los que incide la iluminación en múltiples pasadas y en cada una de estas pasadas se usan shaders para calcular la influencia de un grupo de las luces del mundo. Conjuntamente con la pérdida de eficiencia que se produce al procesar la escena en cada pasada, la desventaja principal de estos algoritmos es que implican la implementación de complicados shaders para manejar todas las combinaciones posibles de luces (omni, spot, omni y spot, etc.).

Los materiales actuales también manejan efectos cada vez más complejos como son el caso de *Steep Parallax Mapping* y Animación de Coordenadas de texturas en capas múltiples. En ocasiones se usan diferentes combinaciones de estos por, lo que se debe a su vez procesar la escena en varias pasadas. El número de instrucciones necesarias para describir estos materiales es comúnmente elevado y la tendencia es seguir creciendo según aumentan las prestaciones de los hardwares gráficos. Si a los shaders que describen estos materiales, se les suma el código necesario para manejar diferentes

combinaciones de modelos de iluminación y sombreado es muy fácil exceder el número de instrucciones permitidas por la mayoría de los GPU.

Una alternativa para el procesamiento de la iluminación en mundos virtuales actuales es el uso de *sombreado retardado* (Deferred Shading). Este método elimina el alto costo que conlleva procesar varias veces la escena para calcular la iluminación, sin embargo introduce un número de desventajas que frenan el uso extensivo de esta alternativa.

Entre estas desventajas se encuentra un alto consumo de memoria y un procesamiento innecesario de píxeles que se encuentran visibles en la pantalla, pero no reciben influencia de las luces. Esto último deriva en un aumento sustancial del número de píxeles procesados (*fill-rate*), lo que produce un estancamiento en los canales del conjunto de procesos que conduce a la representación de la imagen (*graphics pipeline)*.

Objeto de estudio

La técnica de sombreado retardado (*Deferred Shading*) presenta insuficiencias respecto a: empleo de la memoria y a la técnica empleada para realizar los cálculos de iluminación por píxel, lo cual conduce a la pérdida de eficiencia en la obtención de la imagen en tiempo real.

Problema

¿Como reducir las desventajas del *Deferred Shading* y a su vez mejorar el rendimiento?

Objetivo General

Obtener optimizaciones que permitan resolver las insuficiencias de la técnica de sombreado retardado.

Objetivos Específicos:

1. Obtener técnicas para la reducción del consumo de memoria en el sombreado retardado.

2. Proponer soluciones que aumenten la eficiencia de la representación de una escena tridimensional donde aparezcan múltiples luces dinámicas.

3. Proponer soluciones para implementar la Visualización Retardada en sistemas que carezcan de las altas prestaciones ofrecidas por los últimos GPUs.

Hipótesis

1. Utilizando las potencialidades ofrecidas por los *shaders* es posible implementar métodos para reducir el consumo de memoria.

2. Es posible crear sistema controlador de alto nivel para tratar de forma eficiente el procesamiento de las luces.

Tareas a Desarrollar

1. Realizar un estudio de la situación actual de los sistemas de iluminación y visualización.

2. Estudiar las características de las GPU y sus herramientas de programación.

3. Realizar optimizaciones a la implementación original del *Deferred Shading*. Utilizando los *shaders* para implementar estas directamente en el GPU.

4. Presentar optimizaciones de alto nivel mediante la implementación de un controlador de luces en el CPU.

Restricciones Prácticas

En este documento se presentan las optimizaciones y códigos individuales para la implementación de cada una de estas. Se decidió no presentar un sistema con todas las optimizaciones implementadas debido a que la utilización de muchas de estas mejoras y principalmente las concernientes al controlador de luces de alto nivel dependen estrechamente del tipo de simulación que se desea construir. Siendo diferente la selección de optimizaciones a emplear según el hardware gráfico al que se destina el sistema.

De ahí que en este trabajo se presente un amplio número de optimizaciones válidas para mejorar el funcionamiento de la Visualización Retardada de forma general con el objetivo de que el lector pueda seleccionar las implementaciones más beneficiosas para su simulación en particular, teniendo en cuenta la complejidad de la escena, la memoria disponible, las potencialidades del procesador gráfico y el CPU en el que se prevé la ejecución de la simulación.

Organización del Documento

Este trabajo ha sido organizado de la siguiente manera: El Capítulo 1 presenta una introducción a la iluminación de escenas virtuales; mostrándose los conceptos básicos relacionados con este tópico. A su vez, el Capítulo 2 explica el funcionamiento estándar del *Deferred Shading* para la iluminación de escenas virtuales. En el tercer capítulo se presentan un conjunto de algoritmos y optimizaciones con el objetivo de mejorar el rendimiento del sistema desde diferentes puntos de vista; tanto desde el bajo nivel que representa la implementación de los *shaders* en la tarjeta de video hasta el alto nivel brindado por un controlador de luces.

A su vez, en el Capítulo 4, se relacionan un conjunto de pruebas realizadas al sistema para presentar las mejoras alcanzadas con el uso del sistema propuesto y finalmente se reflejan las conclusiones a las que se llegan luego de implementar esta tecnología e ideas sobre futuras mejoras.

1

CAPÍTULO

ILUMINACION

Este capítulo introduce los conceptos básicos de la iluminación de escenas virtuales así como el empleo de *shaders* para introducir nuevos algoritmos en la tarjeta de video. Además se presentan explicaciones de algunos de los modelos matemáticos más usados en la iluminación de mundos virtuales. Describiéndose

las diferencias principales existentes en los focos de luz y su influencia en la escena.

ILUMINACION

Shaders

Gracias al avance en los hardwares gráficos, 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. Hardweare

La limitante fundamental de los hardwares de aceleración gráfica 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*).

En los últimos 5 años se ha elaborado una alternativa al *Fixed-Function* para aumentar la flexibilidad y capacidades gráficas de los hardwares aceleradores 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 *shaders*, y según su funcionamiento y lugar de ejecución se dividen en *Vertex Shaders* y *Píxel Shaders*.

Los *Vertex Shaders* son los encargados de transformar todos los vértices de la escena. En estos se ejecutan las transformaciones de espacio objeto a espacio de mundo, de cámara, y finalmente se obtiene la posición en la pantalla. Además, se incluyen en estos todas las demás operaciones a nivel de vértices

como son cálculos procedurales de coordenadas de texturas, iluminación per-vertex, entre otras.

La figura 1 muestra a gran escala como trabaja el pipeline gráfico. Mucho de los pasos fueron omitidos para facilitar la comprensión del mismo.

Luego de que el vertex shader transformó los vértices recibidos y realizó otras operaciones a nivel de vértices, el interpolador (Rasterizador) recibe la posición en la pantalla de cada uno de los vértices y genera los triángulos correspondientes. Al dibujar estos triángulos se calcula la posición de cada uno de los píxeles que conforman el triángulo.

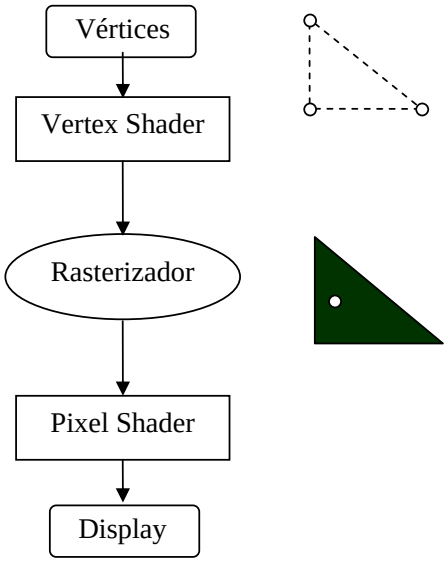

Figura 1. Pipeline gráfico Conceptual usando shaders.

Cada uno de estos píxeles se introduce en el *Píxel Shader* para que este calcule el color del píxel en la pantalla. De esta forma en el *Píxel Shader* 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.

De esta forma, el uso de los *shaders* permite introducir nuevos algoritmos en el hardware de video. Dando la posibilidad de realizar avances en la visualización virtual sin la necesidad de esperar a que estos algoritmos sean codificados en el chip de video para obtener aceleración por hardware.

Modelos de Iluminación

En la naturaleza, cuando la luz es emitida desde un foco luminoso, esta es reflejada desde innumerables objetos antes de alcanzar los ojos del espectador. Cada vez que es reflejada, una parte de la energía es absorbida por la superficie, otra parte es dispersada en direcciones aleatorias y el resto se dirige a otra superficie o a los ojos del espectador. El proceso anterior se repite hasta que la energía se reduce a cero o el espectador percibe la luz.

Los cálculos necesarios para reproducir exactamente el comportamiento de la luz en la naturaleza consumirían demasiado tiempo y recursos como para permitir visualizar escenas en tiempo real. De esta forma, pensando en reducir el tiempo de procesamiento de la iluminación, se han desarrollado modelos matemáticos que aproximan el comportamiento físico de la luz en la naturaleza.

Estos modelos matemáticos han sido divido en dos grandes categorías: aquellos que representan luces ambientales y los que se refieren a luces directas.

Las luces ambientales son las que han sido dispersadas tantas veces producto de la reflexión con las superficies que no se puede determinar la posición del foco desde el cual fueron emitidas. Las luces indirectas usadas por los fotógrafos es un buen ejemplo de luces ambientales. En los gráficos generados por computadoras, estas luces se representan sólo mediante su color y la intensidad, no aportando reflexiones especulares ni degradados difusos.

Contrarias a estas, las luces directas inciden en la superficie sin ser dispersadas por otros objetos. Estas luces producen reflejos especulares en la superficie y su dirección de incidencia es usada para calcular factores de sombreado que modulan la intensidad y el color de la luz en la superficie.

Entre los modelos matemáticos más utilizados para representar contribuciones directas, se destaca el Phong. Este fue creado por Bui Tuong

Pong y permite producir un alto grado de realismo en objetos tridimensionales combinando tres elementos básicos: el componente Ambiental, el Difuso y el Especular.

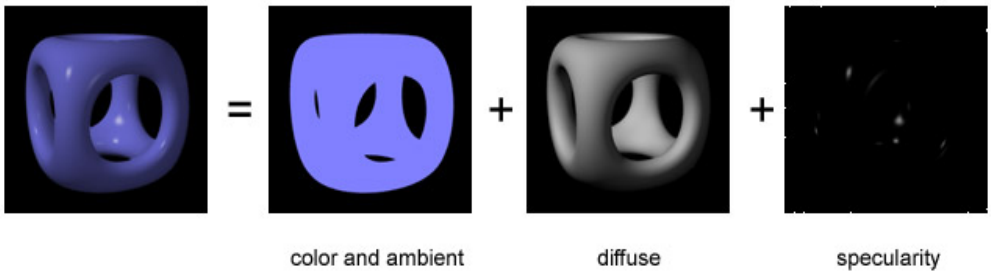

Figura 2. Modelo de Iluminación Phong.

El componente Ambiental produce una iluminación constante en la escena. Todos los píxeles que conforman el objeto reciben el mismo color pues este componente no depende de otros factores como la dirección de la luz, el vector normal a la superficie, etc. Este componente es calculado muy rápidamente, pero, por si solo, produce resultados muy poco realistas, produciendo objetos con apariencia plana.

Contrario al ambiental, el componente Difuso si depende de la dirección de incidencia de la luz y de la normal a la superficie. Resultando en variaciones del color para cada píxel de la superficie, producto del cambio en la dirección de incidencia y de las distorsiones que se pueden presentar en la orientación del vector normal a la superficie.

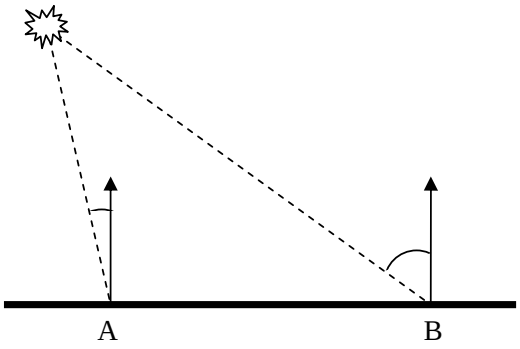

Figura 3. Dirección de incidencia de la luz en dos puntos de la superficie.

El componente difuso usa el ángulo entre el vector normal y la dirección de la luz para calcular la intensidad en cada píxel. Siendo mayor mientras más pequeño sea este ángulo.

De esta forma el punto A de la figura 3 recibirá una iluminación más intensa en el componente difuso que el punto B pues recibe la luz más directamente.

El componente Especular representa los reflejos especulares que se producen en la superficie. Esta componente produce zonas de alta intensidad y brillo en contraste con otras porciones donde genera valores de reducida intensidad. De esta forma, el componente especular simula el efecto producido en la intensidad de la luz que percibimos cuando el rayo de luz incide en la superficie y es reflejado directamente hacia la cámara; produciéndose una mayor intensidad en el punto A de la figura 4 que en el punto B, pues el reflejo de la luz del punto A incide directamente en la cámara, mientras que el reflejo del punto B se aleja de esta. Por esta propiedad es que el produce resultados más intenso que el componente difuso y varía con mayor rapidez en dependencia de la dirección de la luz, el vector normal a la superficie, la posición de la cámara, el factor que describe el poder especular, entre otros.

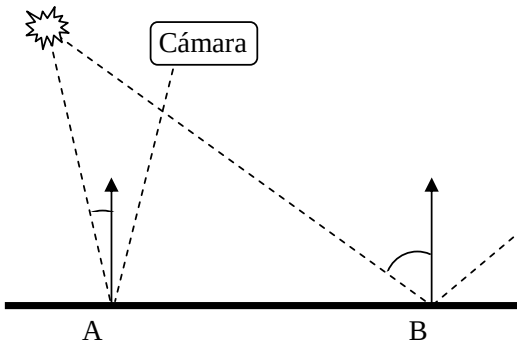

Figura 4. Reflexión especular de la luz

Figura 5. Efecto visual del componente especular de la luz.

El proceso de calcular el componente especular es más costoso y lento que el cálculo de la intensidad difusa y ambiental, pero adiciona un alto realismo a la escena; brindando apariencias metálicas, rugosas, húmedas, etc.

Tipos de Luces

La iluminación de una escena, no sólo depende del modelo usado para calcular los componentes de la luz, sino también de las propiedades de los focos luminosos. En la práctica se destacan tres tipos de focos: los puntuales (*Point Lights*), los direccionales (*Directional Lights*) y los proyectores (*Spot Lights*),

Focos Puntuales

Los focos puntuales son aquellos en los que la luz es emitida desde un sólo punto de la escena. Estos tienen color y posición, pero no una sola orientación; sino que la luz es emitida en todas las direcciones como se muestra en figura 6. Un ejemplo clásico de este tipo de luz son los bombillos, donde la luz parte del centro y es emitido en todas las direcciones.

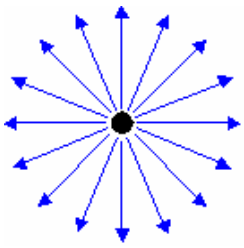

Figura 6. Fuente puntal de luz.

Los focos puntuales presentan un rango de influencia. De forma que la intensidad de la luz es más débil a medida que la posición del foco se encuentre más lejana, siendo nula cuando esta sea mayor que el rango de influencia del foco.

Focos Direccionales

A diferencia de los focos puntuales, los direccionales no presentan posición y emiten la luz de forma paralela. Esto significa que toda la luz emitida por estos focos se desplaza en la misma dirección y sentido como se muestra en la figura 7.

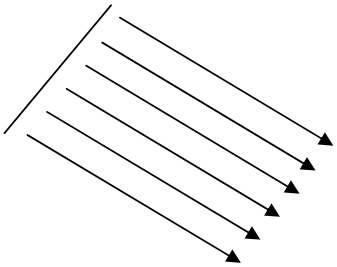

Figura 7. Foco Direccional.

Los focos direccionales se pueden imaginar como emisores de luz situados a una distancia cercana al infinito o muy grande en comparación con la escena a representar. Un ejemplo de estas luces es el Sol, este se encuentra situado a una gran distancia en comparación con las dimensiones del planeta tierra y por tanto el planeta recibe la influencia de este en forma de rayos paralelos.

Debido al reducido número de factores que se tienen en cuenta para realizar el cálculo de la iluminación de los focos direccionales, se consideran como el tipo de luz menos costoso en cuanto a tiempo de procesamiento y recursos necesarios.

Proyectores

Los proyectores presentan color, posición y dirección en la que es emitida la luz. Siendo la región de influencia de estos focos en forma de cono según se muestra en la figura 8.

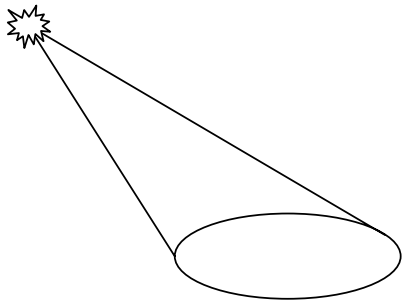

Figura 8. Proyector de Luz.

Estas luces son una de las más costosas en cuanto a la complejidad del cálculo, debido a la cantidad de factores que se han de tener en cuenta para representar estos focos. Entre estos factores se encuentran, el rango de influencia, el ángulo que forma el lateral del cono con su altura, la atenuación

horizontal y la atenuación vertical, la distancia que viaja la luz antes de impactar la superficie, etc.

Un ejemplo de este tipo de luces son las linternas. Estas emiten la luz en forma de cono, siendo atenuada la intensidad de esta según los objetos iluminados se encuentren más alejados.

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.

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 esta, se interpola el valor de la influencia en los vértices y se genera la contribución en cada píxel.

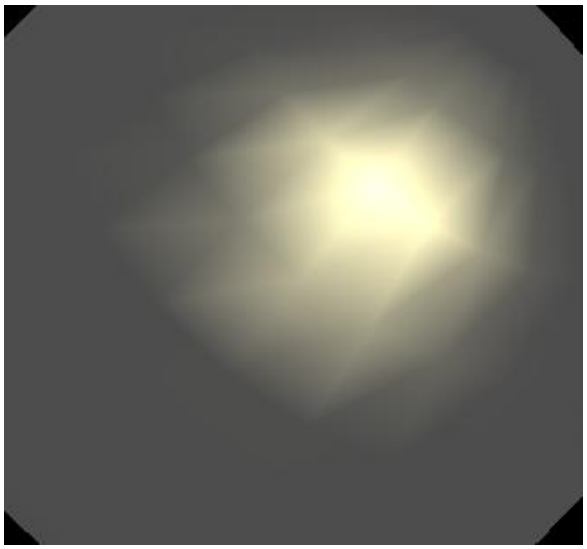

Figura 9. Iluminación "Per-Vertex" en un objeto con pocos polígonos.

La figura 9 muestra un objeto con pocos polígonos 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.

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

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 de 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.

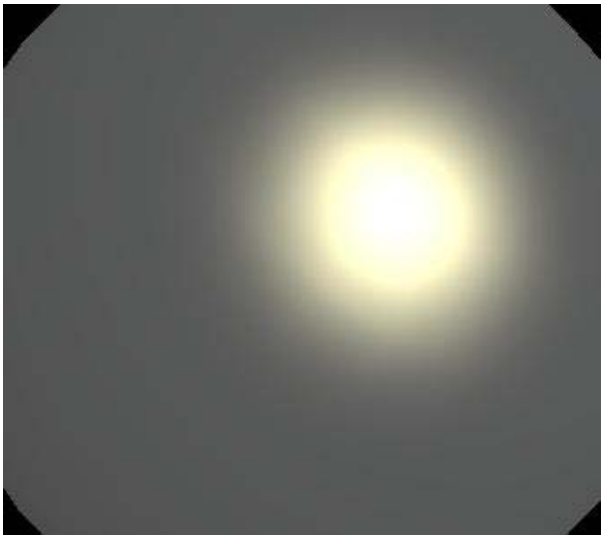

Figura 10. Iluminación "Per-Píxel".

La figura 10 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 esta.

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.

Algoritmo de múltiples pasadas

Los mapas de luces, ayudan a mejorar el rendimiento total en la iluminación de mundos virtuales, sin embargo, ellos también presentan desventajas significativas. Una de ellas es que sólo pueden precalcular la influencia de luces estáticas y sólo en objetos estáticos. De ahí que los objetos animados no puedan ser iluminados usando esta técnica.

Para estos objetos se deben usar las técnicas convencionales de iluminación *per píxel*, en donde la ecuación de iluminación debe ser calculada para cada píxel del objeto en la pantalla mediante el empleo de *shaders*.

El problema de evaluar la ecuación de iluminación de cada luz en el shader es que estos tienen una restricción en cuanto al número máximo de instrucciones que se pueden ejecutar; siendo este número variable en dependencia del hardware gráfico.

La evaluación de cada ecuación de iluminación conlleva la ejecución de varias instrucciones, por lo que a medida que aumenta el número de luces a calcular y la complejidad del modelo matemático con que esta es evaluada, aumentará a su vez el número de instrucciones a ser ejecutados. De esta forma en la mayoría de las ocasiones sólo es posible ejecutar 8-10 luces en el *píxel shader* antes de que el número de instrucciones máximo sea alcanzado.

Para poder ejecutar todas los cálculos necesarios usando los *shaders*, se emplea un algoritmo de múltiples pasadas: Se usa un *shader* que soporte un número fijo de luces (generalmente 8) y se divide las luces en la escena en grupo de ocho luces. Luego para cada uno de estos grupos se calcula la influencia de este en cada píxel y se mezcla el resultado con el generado por los demás píxeles usando transparencia (*blending*).

El proceso anterior se aprecia en el siguiente seudo código:

```
-  Para cada grupo de 8 luces:
     i.  Enviar los datos de las luces al shader
    ii.  Para cada píxel de los objetos procesados:
           1. Evaluar la ecuación de iluminación de cada una
              de las 8 luces
           2. Mezclar la influencia generada con el valor
              que exista en el píxel de la pantalla usando
              blending.
```

Al final, el valor de la influencia de cada luz en cada píxel de la pantalla se encuentra almacenado en el *FrameBuffer*, mostrándose la contribución total de todas las luces en la escena virtual.

Este algoritmo es una forma muy directa de implementar la iluminación, pero se necesita un alto número de pasadas para realizar el cálculo. En cada una de estas pasadas se procesa la misma geometría una y otra vez y sólo se realizan cálculos diferentes en el *píxel shader*, por lo que el *vertex shader* estará repitiendo los mismos cálculos en reiteradas ocasiones durante cada pasada.

En los próximos capítulos se presentarán soluciones alternativas de iluminación que involucran un menor número de pasadas y realizan optimizaciones al *pipeline* de iluminación, aprovechando las capacidades del hardware de video y las propiedades de cada tipo de luz para reducir el número de cálculos necesarios para procesar la iluminación total de las escenas virtuales.

2

CAPÍTULO

DEFERRED SHADING

Este capítulo introduce la implementación básica del *Deferred Shading*, explicándose como se estructuran y funcionan cada uno de sus etapas. A su vez se pretende repasar los conceptos necesarios para la comprensión de la sintaxis

y estructura de los *shaders* que se ejecutan en la Unidad de Programación Gráfica (GPU).

DEFERRED SHADING

Generalmente, en las soluciones tradicionales de iluminación de mundos virtuales se envía la geometría de los objetos al GPU e inmediatamente se calcula la influencia de la luz sobre estos. Cuando se trabaja con un gran número de luces influenciando un objeto, el proceso de iluminación debe ser dividido en varias pasadas debido a que los shaders que realizan los cálculos sólo pueden manejar un conjunto reducido de luces. En cada una de estas pasadas se envía la geometría del objeto al procesador gráfico y se calcula la influencia de un pequeño grupo de luces. El resultado de estas iteraciones es combinado en el *FrameBuffer* de forma que finalmente se obtenga la iluminación total.

La visualización retardada (*Deferred rendering*) se presenta como una solución muy eficaz para procesar un alto número de luces sin tener que enviar múltiples veces la geometría de la escena al GPU. Esta metodología logra combinar las técnicas convencionales de visualización tridimensional con las ventajas del procesamiento en espacio imagen. Es de destacar que el término *Deferred Rendering* es usado para describir un gran número de técnicas; todas ellas comparten el aplazamiento de una de las etapas del *pipeline* gráfico pero se diferencian en cual parte del *pipeline* es diferida. Durante este trabajo se hará referencia sólo al retardo del sombreado y la iluminación, de ahí que el término usado sea *Deferred Shading;* por lo que sólo se trabajará en base a optimizar el aplazamiento de la ejecución del *shader* que describe la superficie a representar.

El proceso de ejecutar el cálculo de la iluminación y el sombreado en espacio imagen presenta un amplio rango de ventajas y desventajas. Entre las ventajas principales se encuentran:

- El costo de la iluminación se basa en el área de la pantalla que afecte en lugar del número de luces u objetos que sean afectados. De ahí que sea igual de eficiente visualizar muchas luces pequeñas que unas pocas con un gran radio de influencia.
- Las luces pueden ser ocluidas (marcadas como invisibles) usando los mismos algoritmos de visualización que se emplean en el cálculo de la visibilidad de objetos.
- La implementación de sombras mediante el método de *Shadow Mapping* es muy simple y eficiente.

No obstante, esta solución también presenta un conjunto importante de desventajas. De ellas se destacan las siguientes:

- Existe un alto consumo de memoria para la creación del G-Buffer.
- Se requiere de sistemas con altas prestaciones del hardware gráfico.
- Existe el peligro de que la aplicación se vea frenada por el alto fill-rate.

En los siguientes epígrafes se describirá con mayor profundidad el funcionamiento del *Deferred Shading* así como las ventajas y desventajas de su implementación. Durante el resto del documento se explicará una metodología para reducir sus efectos colaterales.

Funcionamiento del Deferred Shading

A diferencia de las soluciones tradicionales para el cálculo de iluminación en las que la geometría que describe la escena es enviada en reiteradas ocasiones al GPU; *Deferred Shading* sólo procesa la geometría una sola vez en cada frame, de esta forma se reduce la sobrecarga de los *vertex shader* al minimizar la duplicación del cálculo de las transformaciones de los vértices de la escena así como sus proyecciones a la pantalla.

Esta descarga de los *shaders* de transformación geométrica puede ser usada para mover las animaciones de vértices (*skinning*) del CPU al GPU; siempre y cuando no se cree un cuello de botella en estos programas al sobrecargar nuevamente la tubería de transformaciones de vértices. No obstante, en la mayoría de las escenas la descarga de la tubería geométrica que implica el uso del *deferred shading* compensa la inserción de cálculos más complejos en el GPU, como son los casos del *skinning*, el desplazamiento de vértices según textura (*displacement mapping*), entre otros.

El algoritmo clásico de Deferred Shading requiere la creación de un buffer de pintura (*FrameBuffer*) especial que recibe el nombre de Buffer de Geometría o G-Buffer. En lugar de almacenar el color de cada píxel, en ese bloque de memoria acumulan un conjunto de valores por cada píxel de la pantalla.

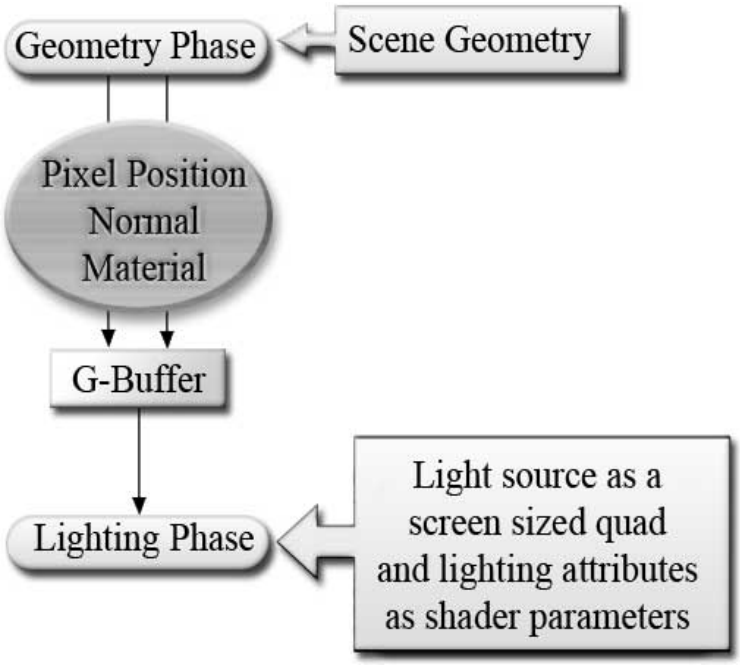

Figura 11. Línea de ejecución del Deferred Shading

Inicialmente los polígonos y demás primitivas que conforman la escena son proyectados a la pantalla y los valores que describen la superficie en cada píxel que sea visible son salvados al Buffer de Geometría. De esta forma no es necesario volver a procesar la geometría de la escena pues ya el G-Buffer contiene los datos que se pueden necesitar para cada píxel.

Una vez rellenado el Buffer de Geometría se usa como si fuese una textura de la cual se extrae la información necesaria para describir geométricamente los píxeles que conforman la pantalla. Teniendo estos datos, se procede a iterar sobre todos los focos luminosos que afecten la escena y para cada uno de estos se dibuja un rectángulo que cubra el *frame-buffer* de forma que los *píxel shaders* correspondientes se ejecuten para cada píxel de la pantalla. En los píxel *shaders* que se introduzcan para procesar el rectángulo anterior se especifican los parámetros de cada uno de los focos de luz. Usando estos parámetros y los datos contenidos en el Buffer de Geometría se evalúa la ecuación de iluminación y se almacena el resultado en el *frame-buffer* final. Combinándose su intensidad con la obtenida de los focos anteriores.

El procedimiento descrito anteriormente es comúnmente implementado mediante la ejecución de dos fases principales como se muestra en la figura 11.

Fase Geométrica

La primera etapa es conocida como Fase Geométrica. Durante su ejecución la geometría que describe la escena es enviada al GPU y mediante la implementación de *shaders* especiales; en lugar de almacenarse el color de cada píxel en el frame-buffer; se almacenan una serie de valores que describen las propiedades de la superficie en cada píxel. De esta forma atributos tales como el vector normal, la potencia especular, el factor de incandescencia, el término de oclusión entre otros; son almacenados en uno o más *frame-buffers* que reciben el nombre de *Geometry Buffer* o *G-Buffer*.

Esta primera etapa es la única que realmente usa la geometría de los objetos que conforman la escena, de forma que una de las mayores ventajas de este sistema es que luego de ejecutada esta fase no interesa que tipo de geometría ha sido enviada al GPU. Permitiendo que se iluminen de igual manera objetos geométricos estáticos, dinámicos, *billboards*, impostores, partículas, terrenos con niveles de detalle implementados, etc.

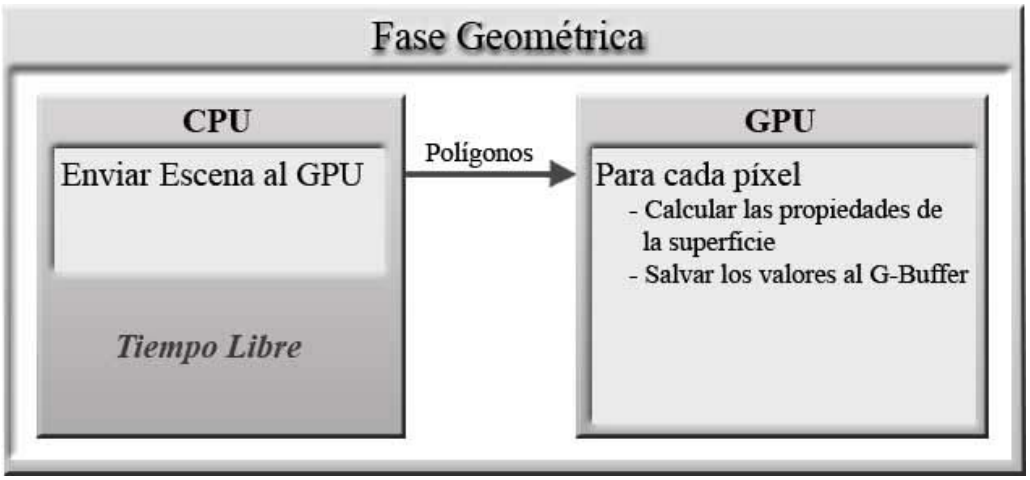

Figura 12. Distribución de Procesadores en la Fase Geométrica.

Otra de las grandes ventajas de la separación del proceso de visualización en dos etapas es que durante la ejecución de esta primera fase no hay que tener en cuenta la iluminación, de forma que los *shaders* que procesan la geometría (*vertex shader*) y los que calculan los colores (*píxel shader*) disminuyen su complejidad al eliminar las ecuaciones y procesos propios de la iluminación. De ahí que sea posible ejecutar estos programas gráficos en sistemas con hardware de video menos poderosos; que resultan inalcanzables para los modelos de visualización anteriores, donde el número de instrucciones de los *shaders* sobrepasan fácilmente las especificaciones de sus GPU.

Fase de Iluminación

Durante la segunda etapa del *deferred shading* se recorre cada una de las luces de la escena y se dibuja un rectángulo que cubra la pantalla de forma que cada píxel de la pantalla sea cubierto. Durante el cálculo del color del píxel se especifican *shaders* particulares que extraen del G-Buffer las propiedades de la superficie en ese punto y usando la posición de la luz, su color, etc. como parámetros; se evalúa la ecuación de iluminación y se combinan los resultados en el buffer de color para producir la iluminación total de la escena.

Al separar el cálculo de iluminación de la geometría, las luces se transforman en entidades independientes de forma que estas no estén sujetas a las mismas restricciones impuestas a sus contrapartidas en los sistemas estándar de iluminación. No siendo necesario establecer un número máximo de luces en cada región de la escena ni fijar que luces influencian los objetos.

Esta ventaja facilita el trabajo de los artistas y diseñadores del entorno virtual al no tener que establecer dependencias entre las superficies a ser iluminadas y las luces que las influencian. De esta forma la implementación de entornos iluminados dinámicamente se torna un proceso altamente intuitivo.

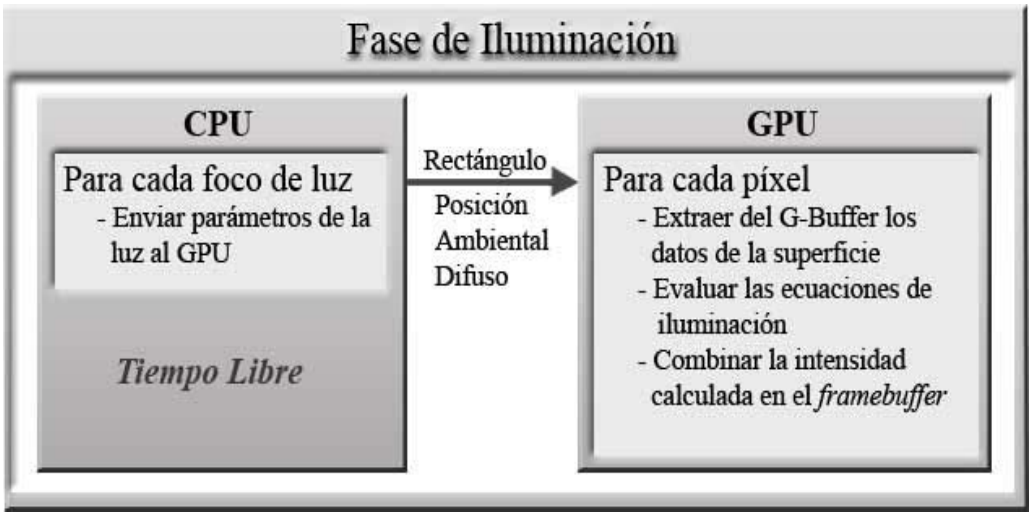

Figura 13. Distribución de Procesadores en la Fase de Iluminación.

Hasta el momento se ha presentado el *Deferred Shading* como una técnica de iluminación dinámica de entornos. Sin embargo esa solución presenta una cara mucho más general. Como se ha visto *Deferred Shading* lleva el proceso de cálculo del color de los píxel a espacio imagen de esta forma la iluminación viene a ser un post-procesamiento de texturas. Al igual que se trabaja la iluminación; en esta segunda etapa, se pueden definir diversos efectos especiales como son la niebla, la distorsión de la profundidad, entre otros y

siendo tratados mediante el mismo proceso aplicado a las luces, se aplican en las escena.

Esta técnica de post-procesamiento, no sólo permite implementar numerosos algoritmos y efectos especiales sino que también posibilita combinarlos para producir una escena foto-realista. Introducir un nuevo efecto como Distorsión de Calor a la tubería de visualización del motor de realidad virtual usando esta metodología no involucra procesar múltiples veces la geometría de la escena en varias pasadas, sino que se transforma en algo tan intuitivo como la implementación de *shaders* específicos que realicen el efecto usando los datos almacenados en el G-Buffer. De esta forma, adicionar el efecto especial al sistema de visualización puede llegar a ser tan simple como adicionar una nueva entidad de post-procesamiento a la lista de efectos de post-procesamiento contenida en el manejador de esta segunda etapa (cuando se dice entidad de post-procesamiento se refiere a cualquier efecto especial o foco de luz).

Buffer de Geometría

Uno de los pasos más importantes en la planificación de un sistema de visualización retardada es determinar que parámetros son necesarios para realizar la evaluación de las ecuaciones de iluminación. Estos parámetros deben ser separados en dos grupos.

En un primer conjunto están los que definen el foco de iluminación y que generalmente son pasados al sistema como parámetros constantes de los shaders de la fase de iluminación. Ejemplo de estos valores son la Posición de la luz, su color Ambiental, Difuso y Especular, etc.

En un segundo grupo se encontrarían aquellos parámetros que definen las propiedades de la superficie sobre la que incide la iluminación. Aquí se pueden encontrar parámetros tales como el Vector Normal, Color Difuso, Posición en coordenadas de mundo, etc. Estos valores son calculados durante la ejecución de la fase geométrica y almacenada en uno o varios *FrameBuffers* según las

capacidades del hardware gráfico que se disponga. Al *FrameBuffer* generado o al conjunto de estos se le da el nombre Buffer de Geometría o G-Buffer y es una de las características más importantes del *deferred shading* ya que contiene toda la información geométrica necesaria para los cálculos de iluminación de la escena, reduciendo el procesamiento de la parte visible de la escena a un solo pase.

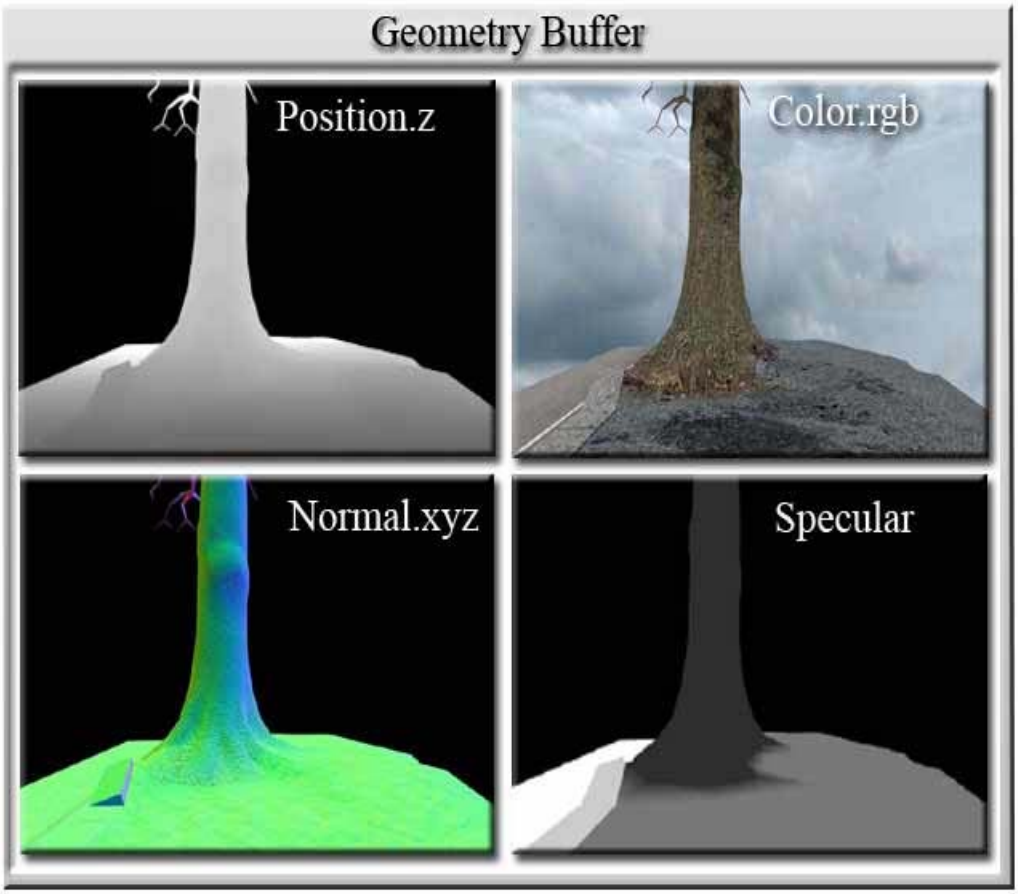

Figura 14. Representación Visual de algunos de los parámetros almacenados en el G-Buffer.

Como se ha visto, el Buffer de Geometría permite almacenar los valores necesarios de cada píxel de la pantalla para luego ejecutar las ecuaciones de iluminación o los cálculos de otros efectos de post-procesamiento. Sin embargo

se ha de prestar un especial cuidado en la cantidad de parámetros a almacenar, debido al alto consumo de memoria que implica almacenar estos valores para cada píxel. De ahí que siempre se busque un consenso entre memoria disponible y calidad gráfica de las ecuaciones a representar.

Implementación Estándar del Deferred Shading

Como punto de partida para la serie de optimizaciones y mejoras que plantea este trabajo; durante el resto de este capítulo se presentaran ejemplos de implementaciones tradicionales del *Deferred Shading* así como los principales logros alcanzados en el desarrollo de esta tecnología.

Los códigos de los shaders que se presentarán han sido creados en HLSL que es un lenguaje de alto nivel para la programación de la GPU con una sintaxis similar al C.

Buffer de Geometría

Como se explicó con anterioridad, el Buffer de Geometría almacena los valores necesarios para evaluar la ecuación de iluminación en cada píxel de la pantalla. Una vez definidos estos valores, aún se necesita especificar el formato más adecuado para almacenar estos parámetros y a la vez permitir usar el Buffer de Geometría como un buffer o textura dibujable (*Render Target)*, válida en las librerías gráficas que soporten aceleración por hardware.

El formato más difundido de estos *Render Targets* es el $ARGB$ que se presenta como cuatro bytes por píxel conteniendo la información de los canales rojo, verde, azul y alpha (transparencia). Sin embargo, tener tan sólo cuatro bytes para almacenar el cúmulo de parámetros necesarios restringe enormemente la fidelidad y precisión a usar. De ahí que se necesiten formatos con mayor capacidad de almacenamiento.

Afortunadamente, los recientes adaptadores gráficos implementan formatos de mayor capacidad, donde cada canal puede ser un valor flotante de 16 o 32 bits. Estos últimos adaptadores, no sólo amplían la capacidad de almacenamiento de los *Render Targets*, sino que además permiten usar varios de estos al mismo tiempo para almacenar los resultados del procesamiento en el GPU.

Para este primer ejemplo, se usaran estas capacidades, de forma que todos los parámetros necesarios puedan ser almacenados en el Buffer de Geometría. De esta forma, se usaran dos *Render Targets* con los formatos RGBAF32. Este formato permite almacenar cuatro valores reales con 32 bits de precisión en cada textura dibujable. Lo que permite un total de 8 posiciones por píxel para almacenar valores flotantes.

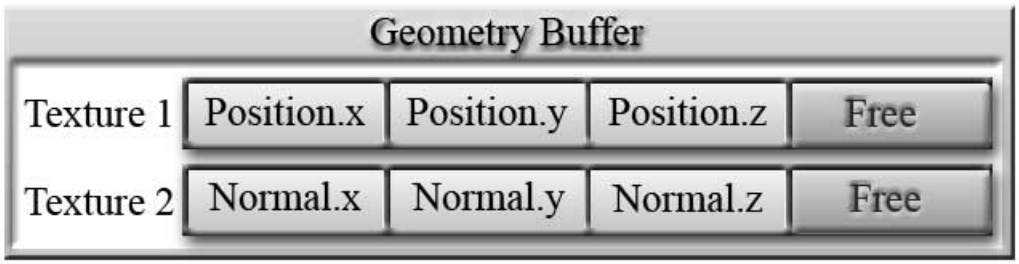

Figura 15. Distribución usada en el primer ejemplo de Deferred Shading para almacenar los parámetros en el Buffer de Geometría empleando dos Render Targets.

En la figura 15 se muestra la distribución de los parámetros de iluminación en el Buffer de Geometría que será usada en el primer ejemplo de Deferred Shading. Para cada píxel los tres componentes de la posición en espacio de mundo serán almacenados en los canales Rojo, Verde y Azul del primer Render Target y los valores que definen el vector normal en los mismos canales pero de la segunda textura. El canal alpha de ambas texturas permanece sin utilizar en aras de mantener la simpleza de este primer acercamiento al Sombreado Retardado pero en estas casillas se podría colocar otros valores como son el

poder especular y el término de oclusión para enriquecer la descripción del píxel y permitir cálculos más complejos que aumenten la calidad visual.

Fase de Geometría

Como se ha visto, durante esta primera etapa, los diferentes *Render Targets* que definen el Buffer de Geometría son activados para permitir escribir en ellos y la geometría de la escena es enviada al GPU de forma que los shaders efectuados en esta fase almacenen los parámetros necesarios para describir la superficie en el Buffer de Geometría.

Este proceso puede ser tan simple como el que se presenta a continuación donde el vector normal a la superficie y la posición en coordenadas de mundo de cada vértice son calculados en el *vertex shader* y enviados al *píxel shader*, quien a su vez interpola estos valores para extraer la posición y normal de cada píxel. Estos dos últimos valores son almacenados en el Buffer de Geometría usando la distribución representada en la figura 15.

```
float4x4 matWorld;
float4x4 matWorldViewProjection;

struct VS_OUTPUT
{
   float4 Pos             : POSITION;
   float3 NormalInWorld   : TEXCOORD0;
   float3 PosInWorld      : TEXCOORD1;
};

VS_OUTPUT vs_main(float4 inPos: POSITION, float3 inNormal: NORMAL)
{
   VS_OUTPUT Out;
   Out.Pos           = mul (matWorldViewProjection, inPos);
   Out.PosInWorld    = mul (matWorld,               inPos);
   Out.NormalInWorld = mul (matWorld,               inNormal);

   return Out;
}
```

El código anterior muestra la implementación del *vertex shader*. Este inicialmente define el uso de dos matrices con cuatro filas y cuatro columnas. La primera de estas matrices, almacena la matriz de transformación de espacio

objeto a espacio mundo (*matWorld*). En la segunda matriz se mantiene la concatenación de las matrices de Mundo, de Cámara y la de Proyección (*matWorldViewProjection*) resultando en una matriz de transformación de espacio objeto a coordenadas normalizadas de pantalla.

La estructura *VS*_OUTPUT describe que valores serán enviados desde este *vertex shader* al interpolador y posteriormente al *píxel shader*. Primeramente se define que se enviará la posición del vértice en coordenadas normalizadas de pantalla, para esto se usa la palabra reservada *POSITION*. Este parámetro en realidad es el único que la tubería de visualización necesita para calcular la posición exacta de los píxeles que conforman los diferentes polígonos, de forma que se procesen cada uno de estos en el píxel shader.

En la estructura anterior también se plantea el envío de dos datos adicionales con tres componentes cada uno. Estos datos no son usados directamente por la tubería de visualización sino que son interpretados en el *píxel shader*. De ahí que no se especifique semánticamente los datos a enviar. Tan sólo se declara que se enviarán por los canales 0 y 1 de las coordenadas de textura. Estos canales son usados comúnmente como conductos de comunicación entre los *vertex* y *los píxel shaders*.

Al igual que la mayoría de los lenguajes de programación lineal, los *shaders* presentan una función principal que es por donde comienza la ejecución de estos. En el caso del ejemplo anterior esta función es *vs_main* y en esa línea se declara que como entrada se recibirán desde la aplicación los valores de la posición y la normal en espacio de objeto. Enviando al píxel shader los valores establecidos en la estructura VS_OUTPUT, que serían los mismos valores recibidos pero cambiados a espacio de mundo.

Al comenzar la ejecución del *vertex shader* anterior, primeramente se crea un objeto que contendrá los datos a enviar por la tubería gráfica. Posteriormente se calculan cada uno de estos datos y al final se envían al *píxel shader* al representar estos la salida de la función principal del *shader*.

La función intrínseca mul (*Matriz*, *Vector*) como su nombre sugiere realiza la multiplicación entre la matriz y el vector retornando el vector transformado al espacio definido por la matriz indicada. En el código del shader se realizan tres multiplicaciones: La primera transforma la posición del vértice en coordenadas de objeto a coordenadas de pantalla normalizada, para que el interpolador pueda calcular correctamente la posición de los píxel dentro de la geometría descrita y envíe al *píxel shader* los valores esperados. La segunda multiplicación transforma la posición del vértice de espacio objeto a espacio mundo y la tercera realiza la misma operación anterior pero usando el vector normal en lugar de la posición.

Este *vertex shader* es ejecutado para cada vértice que se introduce en la tubería gráfica y al terminar las instrucciones anteriores, cada uno de los tres datos especificados como salida son enviados al interpolador. El interpolador recibe los datos de cada vértice y para cada conjunto de estos que forme un polígono (triángulo, cuadrado u otro tipo de geometría previamente especificada en la aplicación) se calculan los píxeles que lo conforman realizando una interpolación entre sus vértices proyectados a espacio de pantalla. Es por esto último que el dato señalado semánticamente como *POSITION* es obligatorio.

Una vez que el interpolador determina la posición de cada uno de los píxeles que conforman el polígono realiza la misma operación para cada uno de los datos recibidos, de esta forma los valores enviados mediante el canal 0 y 1 de texturas también serán interpolados. De ahí que se obtengan estos datos para cada píxel. Luego de obtener los valores de cada dato para cada píxel del polígono, el interpolador llama al *píxel shader* con estos parámetros. A continuación se especifica el *píxel shader* correspondiente a la Fase Geométrica clásica del *Deferred Shading*.

```
struct PS_INPUT
{
   float3 Normal    : TEXCOORD0;
   float3 Pos       : TEXCOORD1;
};
```

```
struct PS_OUTPUT
{
   float4 Color0 : COLOR0;
   float4 Color1 : COLOR1;
};

PS_OUTPUT ps_main( PS_INPUT Input )
{
   PS_OUTPUT o;

   o.Color0.xyz = Input.Pos;
   o.Color1.xyz = normalize( Input.Normal );

   o.Color0.w = 0;
   o.Color1.w = 0;

   return o;
}
```

El *píxel shader* anterior muestra otra forma en la que es válido escribir la función principal. En lugar de declarar directamente en la función que tipo de parámetros recibirá el *píxel shader* del interpolador, se declara una estructura que contiene dos datos, uno que toma valores de las coordenadas de textura 0 y otro del canal 1. Esta estructura define la entrada del shader de forma que el programa recibe estos datos cada vez que es ejecutado.

Como salida de la función principal se especifica otra estructura conteniendo dos valores que semánticamente definen el color que se almacenará en el píxel correspondiente del *Render Target* 0 y 1 respectivamente. Estos *Render Targets* como se ha dicho deben ser previamente activados en la aplicación principal (CPU).

Como se presenta en la figura 16, los tres componentes que describen el vector posición en el espacio de mundo son almacenados en los tres primeros componentes del *Color0* y los tres componentes del vector normal son depositados en el *Color1*. El cuarto componente (w) de ambos colores queda sin usar para implementaciones posteriores, pero como los shaders obligan a llenar todos los componentes de los parámetros que son devueltos entonces se le asigna el valor 0.

Como se aprecia en el código anterior, el proceso de almacenar la posición se realiza mediante una simple asignación vectorial, pero cuando se repite el proceso para la normal se le aplica a esta una normalización pues la normal llega al *píxel shader* desnormalizada, aún cuando se introduzca en el *vertex shader* como un vector normalizado.

Lo anterior ocurre pues el rasterizador recibe las normales de longitud uno de cada vértice e interpola estas normales para obtener la correspondiente de cada píxel que de los polígonos que usen ese vértice. Durante el proceso de interpolación, los valores de la posición no son afectados pues estos se desplazan por el plano del polígono, sin embargo, este movimiento lineal afecta la longitud del vector normal como se aprecia en figura 6. En donde los dos vértices representados reciben una normal de longitud uno y durante la interpolación, los píxeles situados en la mitad de la primitiva formada reciben el vector con una longitud menor.

Comentario [c1]:

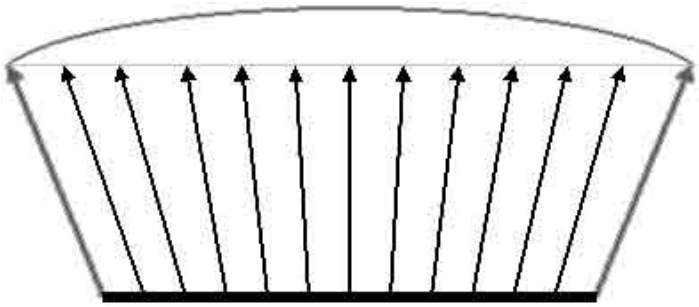

Figura 16. Al interpolar linealmente las normales de dos vértices los vectores resultantes no mantienen la misma longitud.

Para resolver este problema el vector normal recibido en el píxel shader es normalizado de forma que recupere la longitud uno, manteniendo la dirección del vector interpolado. Como se ve en la cuarta línea de la función principal anterior, los tres componentes del vector normal son almacenados en el color 1 una vez normalizado.

Luego de ejecutar el *vertex* y *píxel shader* anterior para cada vértice y píxel de la escena respectivamente, el Buffer de Geometría contiene los valores de posición y normal para cada píxel de la pantalla.

Fase de Iluminación

La fase de iluminación es una de las etapas más importantes del Deferred Shading. En este ejemplo sólo se calculará la influencia difusa de una luz direccional; más adelante se introducirá la implementación de otros efectos especiales como son las sombras y niebla por píxel.

Una vez que todos los parámetros son almacenados en el Buffer de Geometría durante la primera etapa del proceso de visualización retardada; las texturas que conforman el Buffer son enviadas a esta fase como parámetros de entrada en el *píxel shader*. Todas las luces son recorridas y para cada una de estas se calcula de cada píxel de la pantalla la ecuación de iluminación que la describe. Este proceso es ejecutado usando un rectángulo que cubra el *viewport* visible y ejecutando la siguiente combinación de *shaders*.

```
float4x4 matWorldViewProjection;


struct VS_INPUT
{
   float4 pos       : POSITION;
   float2 texCoord  : TEXCOORD0;
};

struct VS_OUTPUT
{
   float4 pos       : POSITION;
   float2 texCoord  : TEXCOORD0;
};

VS_OUTPUT vs_main( VS_INPUT Input )
{
   VS_OUTPUT Out = (VS_OUTPUT) 0;

   Out.pos      = mul (matWorldViewProjection, Input.pos);
   Out.texCoord = Input.texCoord;

   return o;
}
```

El *vertex shader* anterior recibe como entrada las coordenadas de un rectángulo que al ser proyectadas al *viewport* visible, cubre exactamente la pantalla. A su vez, cada una de esas coordenadas vendrá acompañada de un coordenada de textura que será leída del canal de textura 0. Al ser interpoladas estas coordenadas de textura, se usarán como las coordenadas de los píxeles en el Buffer de Geometría según se aprecia en la figura 17. De esta forma el *píxel shader* tendrá las coordenadas del texel donde se encuentran los datos de superficie de cada píxel en el G-Buffer.

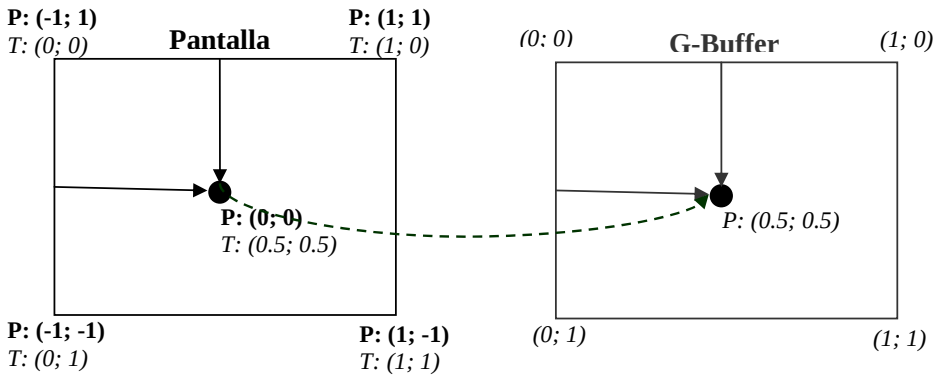

Figura 17. Transformación desde el espacio de pantalla [-1, 1] al Buffer de Geometría usando la interpolación de las coordenadas de textura.

Comentario [c2]:

Luego de ejecutado el *vertex shader* anterior para cada vértice del rectángulo introducido, el rasterizador situado en la tubería gráfica de la Tarjeta de Video interpola estas coordenadas y calcula sus valores para todos los píxeles que conforman el polígono. Ejecutando el siguiente *píxel shader* para cada uno de estos.

```
sampler2D G_Buffer_0;
sampler2D G_Buffer_1;

float3 LightPos;
float4 DiffuseLightColor;

float4 ps_main( float2 texCoord  : TEXCOORD0 ) : COLOR
{
```

```
    // la posición de píxel en coordenadas de mundo y
    // su normal son extraídas del Buffer de Geometría

     float3 PíxelPos = tex2D(G_Buffer_0, texCoord);
     float3 Normal   = tex2D(G_Buffer_1, texCoord);

     // Ecuación de iluminación

      float3 LightDir         = normalize(LightPos - PíxelPos);
      float  DiffuseInfluence = dot(LightDir, Normal);

     // Calcular el color final

      return DiffuseLightColor * DiffuseInfluence;
}
```

El píxel shader anterior recibe como entrada las coordenadas de texturas de cada píxel que como se ha descrito con anterioridad, coincide con la posición del píxel en el Buffer de Geometría. Estas coordenadas son usadas para extraer de este los datos paramétricos de la superficie en el punto procesado. Para esto se accede a los valores almacenados en la posición indicada de las dos texturas que conforman el Buffer de Geometría y que al inicio del código fueron declaradas con los nombres de *G_Buffer_0* y *G_Buffer_1* refiriéndose a las texturas 0 y 1 estructuradas como en la figura 15.

El segundo segmento del *píxel shader* usa los parámetros extraídos del Buffer de Geometría para calcular la influencia de la luz en el vértice, apoyándose en valores suministrados por la aplicación como es el caso de la posición en coordenadas de mundo de la luz. En este ejemplo sólo se presenta la ecuación de las luces direccionales como es el caso del sol y otros focos luminosos cuya lejanía virtualmente no afecta la intensidad de la luz suministrada por lo que sólo la dirección de incidencia es usada.

En la primera línea del segundo segmento se calcula la dirección de la luz con respecto al píxel. Esta dirección es normalizada para que su longitud no afecte los cálculos posteriores de la intensidad.

Durante la segunda línea la intensidad de la luz es obtenida. Para esto se asigna como el valor de máxima intensidad o intensidad 1 a la dirección en donde la normal al píxel y la dirección de la luz coinciden, o sea, ambos vectores apuntan en la misma dirección. Este valor disminuye a medida que la dirección de la luz se aleja de la orientación del vector normal, siendo 0 cuando los dos vectores son perpendiculares y negativo cuando su ángulo es mayor de 90 grados. El cálculo anterior es obtenido usando la función intrínseca *dot* que realiza un producto escalar entre los dos vectores obteniéndose los valores descritos.

Este producto escalar se define como el coseno del ángulo de incidencia de la luz. Ángulo este que se forma entre el vector de la dirección y el de la normal. Siendo esto válido pues los dos vectores son normalizados. El coseno del ángulo tiene la propiedad requerida siendo 1 cuando el este es 0 (ambos vectores miran en la misma dirección) y 0 cuando se forma un ángulo de 90 grados.

Según se ha explicado, los *píxel shaders* son procesados para obtener el color de cada píxel que el interpolador procese. Durante la fase de iluminación el color retornado por el *píxel shader* anterior es la contribución luminosa del foco procesado. Para esto el color de la luz cuyo valor es introducido como un parámetro por la aplicación en la variable *DiffuseLightColor* es modulado por la intensidad calculada. Siendo el color final negro (0) para los píxeles que no son iluminados y manteniendo el color de la luz para los que son expuestos a una iluminación directa.

De esta forma, cada vez que una luz es procesada con los *shaders* anteriores mediante la visualización de un rectángulo que cubre la pantalla, Se calcula la contribución luminosa del foco de luz y se acumulada en el buffer de dibujo actual (*FrameBuffer*). Esta acumulación se realiza mezclando el valor nuevo con los acumulados en iteraciones anteriores. De esta forma, al terminar

la iteración por todos los focos de luz, el *FrameBuffer* guarda la contribución luminosa de todos los focos de luz en la escena visible.

Es válido destacar que es muy fácil implementar diferentes modelos luminosos. Teniéndose focos cuyo cálculo involucre el componente difuso, el especular, el ambiental, combinaciones de estos o diferentes sistemas de cálculos para procesar los componentes anteriores. Tan sólo se necesita reemplazar el *píxel shader* anterior por uno más especializado para soportar luces posicionales o spot. La modulación inducida en el *FrameBuffer* combina las diferentes contribuciones calculadas y de esta forma varios modelos luminosos y tipos de focos pueden coexistir en el mismo entorno sin requerir procesar la escena en varias pasadas.

3

CAPÍTULO

DEFERRED SHADING OPTIMIZADO

En este capítulo se presentan una serie de algoritmos y técnicas para reducir los efectos secundarios del uso de la Visualización Retardada como son el alto consumo de memoria y el fill-rate. Además se optimiza el rendimiento gráfico y la velocidad de ejecución de visualización del sistema.

DEFERRED SHADING OPTIMIZADO

La visualización retardada se ha presentado como un sistema capaz de iluminar mundos altamente complejos en tiempos interactivos. Esta tecnología separa los procesos de iluminación y visualización de materiales permitiendo procesar escenas independientemente de su complejidad estructural, lo cual es una de las características más atractivas de este sistema frente a la creciente complejidad de los mundos virtuales en los video-juegos y visualizadores gráficos.

En el capítulo anterior se presentó el funcionamiento básico de estos sistemas de Visualización Retardada así como la implementación clásica usando las posibilidades que brindan los desarrollos en el hardware gráfico y los *shader programs*. Esta implementación usa algunas de las características más avanzadas de las unidades de procesamiento gráfico por lo que su uso se ve comprometido en sistemas con menores prestaciones. De ahí que el uso de la visualización retardada ha sido exclusivo de los sistemas de avanzada lo que ha frenado su uso extensivo en aplicaciones virtuales en general.

En las siguientes secciones se propone un conjunto de técnicas y algoritmos para optimizar la implementación del Deferred Shading. El objetivo principal de esta propuesta será implementar de manera eficiente la visualización retardada en un sistema con capacidades de procesamiento gráfico inferiores a las

requeridas por las implementaciones actuales. Así como acelerar el funcionamiento general de este.

Las optimizaciones propuestas se han divido en dos categorías principales. En la primera se realiza un estudio de los procesos de bajo nivel optimizándose los *shaders programs* para disminuir sus requerimientos, tiempos de procesamiento y uso de la memoria. Mientras que la segunda categoría se presentan estructuras y algoritmos para implementar un sistema de visualización de alto nivel cuyo objetivo sea optimizar el trabajo de los *shader programs*, controlando el número de píxel a ser procesados por cada luz así como el número de luces y la calidad de estas en dependencia de su localización espacial.

Optimización de los *Shader Programs*

En la implantación presentada en el capítulo anterior, se utilizó un Buffer de Geometría estructurado a partir de dos texturas. En implementaciones profesionales donde se simulan sistemas de luces más complejos y que involucran más parámetros para evaluar las ecuaciones de iluminación; el número de texturas que conforman el Buffer de Geometría tiende a ser más elevado.

Durante la fase geométrica estas texturas se definen como *Render Targets* de forma que los colores obtenidos durante el procesamiento de los *píxel shaders* sean almacenados en cada uno de los *render targets* activos. Las tarjetas gráficas actuales cuentan con una tecnología llamada *Multiple Render Targets* que permite activar más de una textura a la vez. Siendo posible activar hasta cuatro *render targets* simultáneamente. Desafortunadamente tan sólo los últimos procesadores gráficos permiten el uso de esta tecnología, de ahí que en

la mayoría de las tarjetas de videos sólo se pueda especificar un único *render target*.

En esos sistemas el deferred shading tradicional se implementa mediante el uso de soluciones alternativas que involucran ejecutar la fase geométrica tantas veces como texturas se necesiten en el Buffer de Geometría y en cada pasada calcular sólo los valores a ser salvados en esta. Dada la alta complejidad estructural de los mundos virtuales actuales y el creciente aumento en el número de polígonos usados para describir las escenas; esta solución alternativa implica realizar las numerosas transformaciones de vértices varias veces por cada *frame* lo que conllevaría a un punto crítico en el rendimiento de las tuberías gráficas a medida que aumenta la complejidad geométrica de las escenas.

Otra solución alternativa que no presenta la desventaja anterior sería empaquetar los diferentes valores a almacenar, de forma que estos puedan ser acomodados en una sola textura sin incurrir en un costo muy elevado de compactación. Esto se puede realizar de dos formas, reduciendo el número de elementos a almacenar para describir cada atributo necesario y ajustando la precisión necesaria para describir estos.

Reducción del número de parámetros

La siguiente lista presenta los parámetros almacenados comúnmente en el Buffer de Geometría para describir la superficie geométrica en cada píxel de la pantalla.

- Vector normal a la superficie.
- Posición en coordenadas de mundo.
- Atributos del material como la emisión, el poder especular, el factor de oclusión, entre otros.

Durante las siguientes secciones se presentan algoritmos y métodos para reducir la cantidad de valores a almacenar manteniendo los datos necesarios para la evaluar la ecuación de iluminación.

Empaquetado de la Normal

La normal a la superficie en un píxel es un vector tridimensional de ahí que se han de almacenar tres valores reales en el Buffer de Geometría para describir los componentes x, y, z del vector. Otra de las propiedades de las normales a la superficie es que son vectores unitarios; o sea, su longitud es uno.

Esta restricción puede ser descrita matemáticamente usando el teorema de Pitágoras en tres dimensiones; donde la longitud de la hipotenusa (*L*) es la longitud del vector normal.

$$X^2 + Y^2 + Z^2 = L^2$$
$$X^2 + Y^2 + Z^2 = 1^2 \qquad (L = 1)$$
$$X^2 + Y^2 + Z^2 = 1$$

Esta última ecuación permite descartar uno de los componentes a almacenar y recuperarlo en la fase de iluminación aplicando el despeje adecuado. De esta forma es posible descartar el valor del componente z y recuperarlo nuevamente usando:

$$Z = \pm\sqrt{1 - X^2 - Y^2}$$

El problema que se presenta al usar la ecuación anterior es que la z obtenida puede ser un valor positivo o negativo. Por lo que también se necesitará almacenar el signo del valor descartado en el Buffer de Geometría para luego aplicárselo al componente obtenido en la fase de iluminación mediante la fórmula anterior. Este signo sólo va a contener dos estados: positivo o negativo por lo que se puede representar mediante un bit. Donde cero indicaría negativo y 1 positivo.

Usando este procedimiento es posible reducir el espacio de memoria requerido para almacenar el vector normal de cada píxel. Inicialmente se almacenarán tres valores reales siendo cada uno de estos un número de punto flotante de 32 bits. Aplicando el truco de Pitágoras se reduce a sólo dos valores

reales de 32 bits y 1 bit extra. Desafortunadamente los *drivers* de video actuales no permiten definir render *targets* con componentes reales de 32 bits y de 1 bit al mismo tiempo. Por lo que ese bit extra deberá ser empaquetado en uno de los dos componentes restantes.

Hasta el momento se ha trabajado con la normal en espacio de mundo por lo que los valores del componente z de ese vector pueden oscilar arbitrariamente entre los sub-espacios positivo y negativo como se muestra en la figura 18a. Si tenemos en cuenta que la ecuación de iluminación sólo ha de calcularse para las caras que están mirando la cámara pues sólo estas tienen la posibilidad de ser visibles, entonces se puede realizar un cambio de espacio para descartar el bit extra.

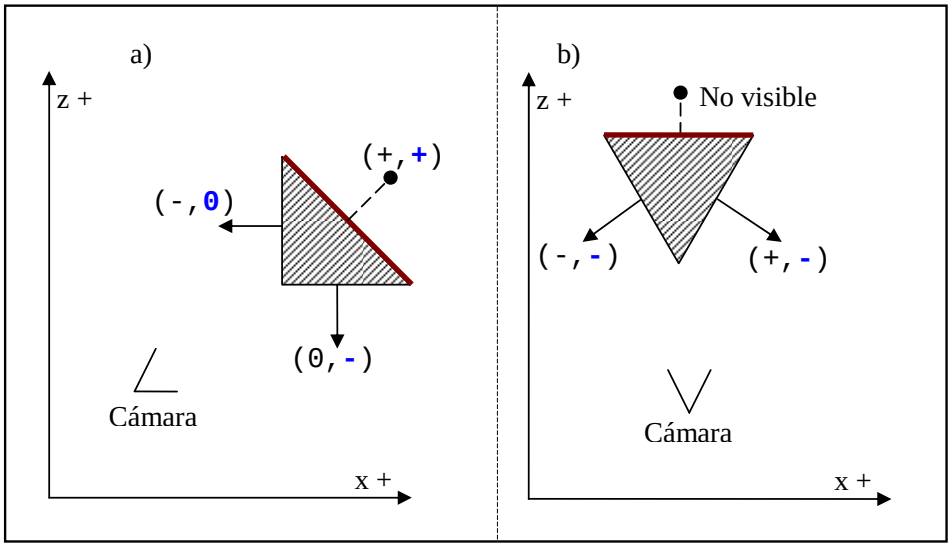

Figura 18. Normales vistas desde diferentes espacios. a) Espacio de mundo, b) Espacio de Cámara.

En el espacio de mundo, la cámara tiene una posición y una rotación arbitraria, sin embargo, en el espacio de cámara; esta se encuentra situada en el punto cero y mirando en la dirección positiva del eje z o la dirección negativa según el sistema de referencia fijado. En este documento se trabaja con el

sistema de referencia conocido como Sistema de Mano Izquierda donde la cámara se encuentra mirando en la dirección positiva del eje z.

Una vez que las normales son transformadas a espacio de cámara todas los vectores que pertenezcan a caras visibles, estarán situados en el semi-espacio negativo como se muestra en la figura 18b y por ende el componente z de estos vectores será negativo por lo que no hace falta almacenar el bit extra.

```
Normal.xy = G_Buffer.xy;
float cuadrados = Normal.x*Normal.x + Normal.y*Normal.y;
Normal.z  = -sqrt(1-cuadrados);
```

Sustituyendo el cálculo de los cuadrados puede ser realizando usando la función *dot*(a, b*)* que realiza un producto escalar entre los dos vectores suministrados y que además se encuentra optimizada intrínsicamente.

```
Normal.xy = G_Buffer.xy;
Normal.z  = -sqrt(1-dot(Normal.xy,Normal.xy));
```

Hasta este momento se ha presentado como reducir el número de valores a ser almacenados en el Buffer de Geometría para representar el vector normal a la superficie en un píxel. Sin embargo, esta reducción de memoria requiere aumentar el número de cálculos necesarios para desempaquetar el vector normal del Buffer de Geometría en la fase de iluminación. A diferencia de los programas que se ejecutan en la Unidad Central de Procesamiento (CPU), los segmentos de código que se evalúan en la Unidad de Procesamiento Gráfico (GPU) se encuentran restringidos en cuanto al número de instrucciones que pueden contener. De ahí que cuando el número de operaciones necesarias para realizar los cálculos de iluminación aumente, será necesario reducir la cantidad de instrucciones que permitan obtener los valores del Buffer de Geometría.

Optimizando un tanto más el empaquetamiento de las normales, es posible precalcular algunos valores para reducir el número instrucciones en la extracción de los vectores normales. Este precálculo implica el uso de memoria auxiliar para almacenar los valores obtenidos, sin embargo, esta sólo representará una pequeña fracción de la memoria reducida al almacenar los componentes z de la normal en cada píxel del Buffer de Geometría.

Como el vector normal es un vector unitario, cada uno de sus componentes oscila en el rango [-1, 1] pero cuando estos valores se almacenen en el buffer de Geometría es útil y en algunas ocasiones necesario como se mostrará más adelante almacenarlos en el rango [0, 1] usando la siguiente transformación:

```
float3 NormalTransformada = (Normal + 1)/2.0f;
```

El rango [0, 1] es compartido por el espacio de textura de ahí que una vez almacenado el componente z del vector normal en cada texel de esta, es posible extraerlo luego usando los componentes (x, y) de la normal transformada como los valores (u, v). Usando el procedimiento siguiente:

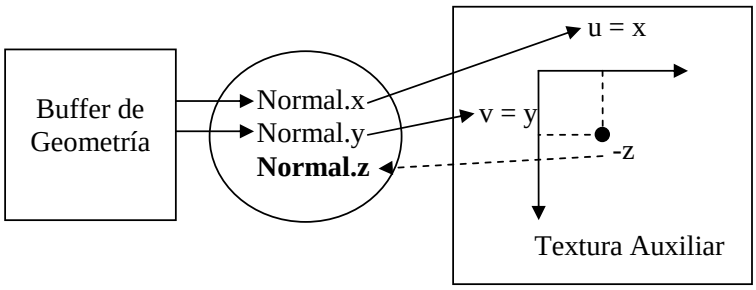

Figura 19. Proceso de desempaquetamiento de la normal usando una textura auxiliar con los valores del componente z de la normal para cada texel. Representando las normales en el rango unitario positivo.

```
Normal.xy = G_Buffer.xy;
Normal.z = -tex2D( Normal_Texture, Normal.xy );
```

El código anterior muestra el procedimiento descrito. Como se puede apreciar no se usan costosas funciones matemáticas como son la Raíz Cuadrada ni el producto escalar de vectores. En general este último procedimiento permite reducir hasta cinco instrucciones en el código de los píxel shaders de la fase de iluminación manteniendo la ventaja de sólo almacenar los valores de los componentes x, y en el Buffer de Geometría.

Comentario [c3]:

Empaquetado de la Posición

Al igual que el vector normal, la posición del píxel es un vector tridimensional. Pero a diferencia de las normales, estos no son vectores unitarios por lo que la optimización basada en el teorema de Pitágoras no puede ser aplicada. No obstante, transformado la posición del píxel desde espacio de mundo a espacio de cámara es posible empaquetar la posición en un sólo valor.

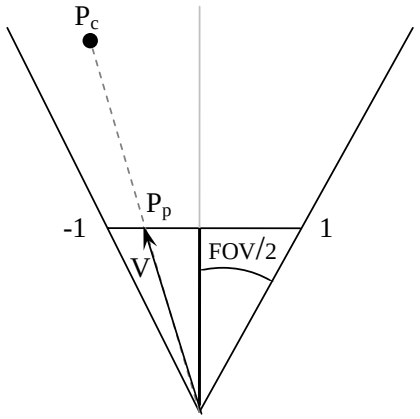

Figura 20. Derivando la posición del píxel P dada sus coordenadas en espacio pantalla y la distancia del píxel al ojo de la cámara.

Cuando se habla de espacio cámara, se hace referencia al espacio como se aprecia desde el sistema de referencia de la cámara. El sistema de referencia de la cámara ubica a su punto de observación en la posición cero del espacio y mirando en la dirección positiva del eje de las Z, teniendo el eje X positivo a la derecha y el Y hacia arriba.

Dada la posición del píxel en coordenadas de pantalla es factible construir un vector V desde el origen (ojo de la cámara) hasta la posición del píxel en la pantalla (P_p). Multiplicando este vector normalizado por la distancia a la que originalmente se encontró el píxel con respecto al ojo de la cámara se reconstruye la posición original en coordenadas de cámara del píxel (P_c), según se muestra en la figura 20.

Durante la ejecución de la fase de iluminación es posible construir un vector normalizado desde la posición de la cámara y en dirección de la posición del píxel en coordenadas de pantalla. De esta forma el único valor que realmente necesita almacenarse en el Buffer de Geometría es la distancia del píxel al ojo de la cámara. Esta distancia es calculada en la fase geométrica y es implementada mediante el cálculo de la longitud del vector que contiene la posición del píxel en coordenadas de cámara. La longitud de este vector es la distancia desde el píxel al punto cero, que siendo el espacio definido el espacio de cámara, representa la posición del ojo de esta. Siendo este último el valor almacenado en el Buffer de Geometría.

Píxel Shader (Fase Geométrica)

```
G_Buffer.z = length(Input.PosInView);
```

Calcular el vector normalizado desde el ojo de la cámara hasta las coordenadas del píxel en la pantalla es un tanto más complejo. Las coordenadas de pantalla como se muestran en la figura 21 describen cada posición de esta en los ejes (x, y) tomando valores entre 0 y 1. El espacio pantalla es muy útil

cuando se mapean los píxel de la pantalla a los texels de una textura, como se puede apreciar cuando se estableció una dependencia entre cada píxel de la pantalla y su correspondiente texel en el Buffer de Geometría. Estas facilidades vienen dadas por el rango de las coordenadas pues las texturas también comparten el rango [0,1] en sus valores u y v. De ahí que dada la posición (x, y) de la pantalla, el texel correspondiente en la textura tiene también las mismas coordenadas (u=x, v=y).

Además de las coordenadas de pantalla, existen las coordenadas de pantalla normalizadas. Estas describen también la posición de los píxel de la pantalla pero usando el rango [-1, 1] en lugar del rango [0,1]. Cuando se ejecuta la proyección de los vértices y la división por la coordenada w de estos, se obtiene la posición en coordenadas de pantalla normalizada. La figura 21 muestra la diferencia de rangos de los ejes (x, y) en los espacios de pantalla y pantalla normalizada.

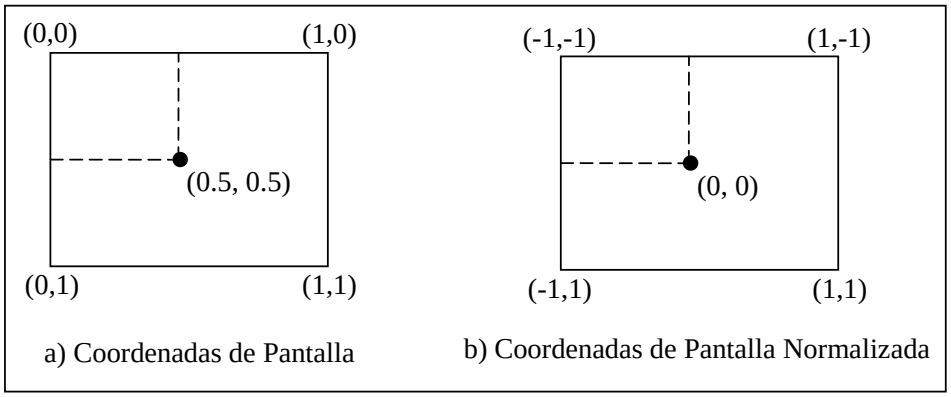

Figura 21. Coordenadas de Pantalla y Coordenadas de Pantalla Normalizada

Durante la fase de iluminación, el *píxel shader* que describe la luz, recibe las coordenadas de textura del píxel activo. Estas coordenadas se mapean directamente a la pantalla, describiendo la posición en espacio de pantalla del píxel. Para determinar el vector desde el ojo de la cámara hasta la posición de la pantalla, se necesitan las coordenadas del píxel en espacio de pantalla

normalizada como se aprecia en la figura 21. Transformar la posición de espacio de pantalla a espacio de pantalla normalizada se reduce a multiplicar por dos y restar uno. De esta forma se transforma el rango de los valores (x, y) desde [0, 1] hasta [-1, 1].

Al transformar las coordenadas de textura al espacio de pantalla normalizada, se obtiene la posición original del píxel que se describe en el Buffer de Geometría una vez que se proyectó a la pantalla durante las transformaciones de vértices que tienen lugar en la tubería gráfica.

Asumiendo que el vector que se desea calcular y que va desde la posición del ojo de la cámara hasta la posición del píxel proyectado a la pantalla se encuentra en espacio de cámara, la situación se reduce a determinar el vector desde el punto 0 del espacio hasta la proyección del píxel en la pantalla cuyas coordenadas son conocidas. De esa forma los componentes (x, y) del vector coinciden con los valores (x, y) de la posición del píxel. Pero aún faltaría determinar el valor del componente z.

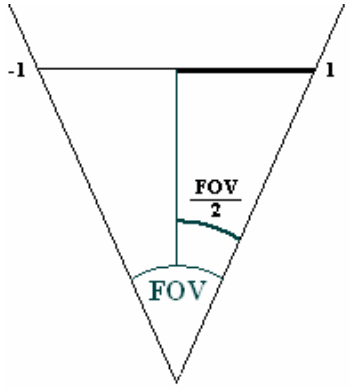

Figura 22. Uso del Ángulo de Visión (FOV) para calcular la distancia del plano de la pantalla al ojo de la cámara.

Este valor representa a qué distancia se encuentra el plano de la pantalla del ojo de la cámara. Para calcularlo se necesitará usar un parámetro empleado para definir la matriz de proyección: El Ángulo de Visión (*Field of View*: FOV).

Este ángulo describe la zona de visibilidad de la cámara como se muestra en la figura 22.

Dado que el plano de la pantalla es completamente visible en su rango [-1, 1] entonces es posible usar la tangente de la mitad del Ángulo de Visión para calcular a que distancia (z) la mitad de la pantalla tiene longitud uno mediante la siguiente ecuación:

$$\tan\left(FOV/2\right) = \frac{1}{z}$$

$$z = \frac{1}{\tan\left(FOV/2\right)}$$

Usando los valores anteriores se obtienen los componentes (x, y, z) del vector buscado. Pero aún existe otro punto a tener en cuenta en la construcción del mismo. Las coordenadas de pantalla normalizadas, describen los valores de las coordenadas dentro de un plano situado en el espacio de mundo o de cámara. Este plano representa la pantalla física en el universo virtual. Aún cuando se distribuyan matemáticamente los valores del -1 al 1 en cada uno de sus ejes, esto no implica que el área visible de este plano sea un cuadrado. La pantalla física contiene más píxeles horizontalmente de los que existen en la vertical. De ahí que el área visible del plano sea un rectángulo con la base más ancha que la altura.

Durante la construcción de la matriz de proyección este parámetro es introducido mediante la variable *ViewAspect* que representa cuantas veces es más ancha que larga la ventana visible. La figura 23 describe un ejemplo en donde la pantalla física contiene 640 píxeles horizontales y 400 verticales, lo que supone que el rectángulo visible es 1.6 veces más ancho que alto.

Como los valores de las coordenadas de pantalla normalizada mantienen el rango [-1,1] para cada uno de sus componentes, al construir un vector del punto

cero en el universo virtual a la posición (x, y) del rectángulo visible; estando (x, y) definidas en el rango [-1, 1], se debe distorsionar el vector para que se ubique en la posición correcta del espacio. Para esto se multiplica el componente x por el ViewAspect. De forma que su nuevo rango sea [-*ViewAspect*, *ViewAspect*] mientras el componente y mantenga su rango normalizado para reflejar así la diferencia de longitudes entre la base y la altura de la ventana visible en el espacio virtual.

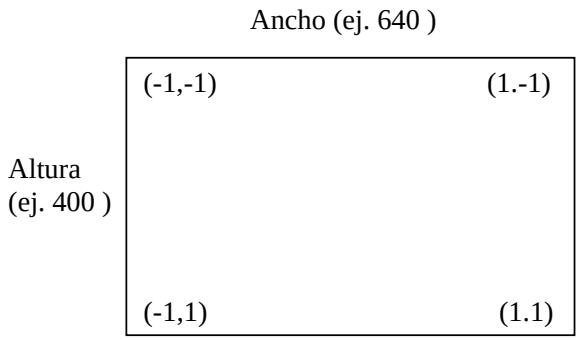

Figura 23. Ejemplo de ventana visible donde el *ViewAspect* es igual a 1.6

Comentario [c4]:

Usando la anterior es posible calcular el vector que va desde el punto cero, que en espacio cámara describe la posición del ojo, hasta la proyección del píxel en el plano virtual, que representa la pantalla normalizada.

Comentario [c5]:

$$vEyeToScreen = \left(x \cdot ViewAspect, \quad y, \quad \frac{1}{\tan\left(FOV/2 \right)} \right)$$

El proceso de empaquetamiento y extracción de la posición del píxel del Buffer de Geometría usando el procedimiento descrito anteriormente se puede implementar de la siguiente forma: usando el componente z del Buffer de Geometría para almacenar la distancia del Píxel en el espacio a la posición del ojo de la cámara y la variable *invTanHalfFOV* como una constante en la fase de Iluminación, describiendo el recíproco de la tangente de la mitad del Ángulo de Visión.

Empaquetamiento en la Fase Geométrica

```
G_Buffer.z = length(Input.ViewPos);
```

Extracción de la Posición en la Fase de Iluminación

(Vertex Shader)

```
out.vEyeToScreen   = float3( Input.Texcoord.x * ViewAspect,
                             Input.Texcoord.y,
                             invTanHalfFOV );

out.Depth          = G_Buffer.z;
```

(Píxel Shader)

```
float3 PíxelPos = normalize(Input.vEyeToScreen) * Input.Depth;
```

Este procediendo permite reducir el número de valores a ser almacenado en el Buffer de Geometría para describir la posición del píxel en el espacio de tres valores representando los componentes (x, y, z) de la posición a sólo uno que contiene la distancia del píxel al ojo de la cámara.

Empaquetado de los atributos de los materiales

Al especificar los atributos de los materiales, se hace referencia al grupo de valores que definen las propiedades de la superficie. Ejemplo de estos atributos son: la potencia especular, el término de oclusión, el factor de incandescencia o de irradiación, entre otros. Almacenar cada uno de estos valores en el Buffer de Geometría, implica usar un vasto espacio de almacenamiento a medida que se usen más parámetros para describir el material.

En el ejemplo de Deferred Shading mostrado en el capítulo anterior sólo se especificó una iluminación direccional difusa, por lo que no se almacena ningún atributo del material en el Buffer de Geometría. Sin embargo, el ejemplo mostrado tenía como meta introducir la Visualización Retarda usando la más simple de las situaciones posibles. En las aplicaciones gráficas, comúnmente se usan más ecuaciones de iluminación, para lograr una imagen final foto-realista. Esta elevada calidad de la imagen obliga a la consideración de numerosos parámetros que describen propiedades intrínsecas del material.

Una forma sencilla de disminuir la cantidad de parámetros a almacenar en el Buffer de Geometría es descartar algunos de estos o fijarlos a un valor constante para toda la escena. Pese a que esto resolvería el problema de almacenamiento, el uso indiscriminado de la reducción de valores trae implícito la disminución de la calidad gráfica al prohibir la evaluación de ecuaciones más complejas de iluminación o la eliminación de variedad, pues todos los materiales de la escena compartirían el mismo valor del material.

Sin embargo, haciendo un uso más racional del número de atributos a almacenar es posible mitigar en gran medida los efectos anteriores. En muchas de las escenas representadas virtualmente, existen parámetros que varían muy poco y casi imperceptiblemente, de ahí que su contribución gráfica sea pobre. Ejemplo de esto son las escenas donde se muestran entornos completamente metálicos o de poca variedad en los que atributos como la potencia especular y el factor de incandescencia tienden a permanecer constante, reflejando propiedades comunes entre los diferentes materiales de la escena. En esos casos, es posible fijar esos atributos como constantes en la ecuación de iluminación, de forma que no es necesario almacenarlos en el Buffer de Geometría.

Existen otros atributos como el factor de refracción y el de transferencia de luz cuyos valores no permanecen constantes en los diferentes materiales de la escena, pero que su contribución gráfica es descartada para mantener un mayor rendimiento e interactividad al disminuir los cálculos necesarios para su evaluación. Los términos que implican el procesamiento de estos valores en las ecuaciones de iluminación pueden ser eliminados para acelerar el proceso de cálculo, lo que implicaría no tener que almacenar estos valores en el Buffer de Geometría.

Además, existen otros parámetros cuyos efectos ocultan o disimulan las variaciones producidas por los demás atributos. Ejemplos clásicos son los factores de incandescencia y de emisión de luz que alteran notablemente la

coloración y reflectividad del material, ocultando los cambios producidos por el término difuso y especular. De ahí que valores como la potencia especular y el factor de oclusión puedan ser eliminados cuando los materiales de la escena presentan un alto poder de emisión de luz.

En general existen miles de parámetros que definen los atributos de los diferentes materiales de la escena; pero sólo es necesario almacenar los valores que serán usados posteriormente en las diferentes etapas de la Visualización Retardada. Valorándose con especial cuidado si los parámetros a ser almacenados tienen una influencia notable en la escena final. De esta forma se puede reducir la cantidad de parámetros a ser almacenados en el Buffer de Geometría con una reducción mínima de la calidad gráfica.

Conjuntamente con lo anterior, es posible realizar una reducción más conservativa y al mismo tiempo más agresiva en cuanto al número de valores almacenados en el Buffer de Geometría para describir los atributos de los materiales. Esta reducción se basa en que los valores almacenados en el Buffer de Geometría son los parámetros que describen la superficie en cada píxel de la pantalla.

En la mayoría de los casos, las propiedades de los materiales no van a variar en cada píxel, sino que se mantendrán constantes en todos los píxeles pertenecientes a un mismo material. Si el número de materiales en la escena es relativamente pequeño, entonces es posible crear un arreglo (*array) de materiales* el cual realiza la función de una paleta de datos y contiene los atributos de cada uno de los materiales. El *array* anterior puede ser introducido como constante en los *píxel shaders* de la fase de iluminación. De esta forma el único valor que debe ser almacenado en el Buffer de Geometría es la posición en el array del material a la que pertenece cada píxel, como se muestra en la figura 24.

Los hardwares gráficos actuales, imponen una limitante en cuanto al número de valores constantes que se pueden enviar a los *shader programs*, de ahí que

esta solución sólo sea aplicable cuando la cantidad total de atributos a almacenar es menor que la cantidad máxima de valores permitidos por el hardware gráfico. En la mayoría de las situaciones se cumple lo anterior, pues cuando se hace referencia al material en la Visualización Retardada, no se implica el material gráfico común que varía por cada superficie con diferente texturizado, sino que se refiere al material como propiedades intrínsecas. O sea que los materiales varían sólo según cambien los atributos usados en las ecuaciones de iluminación y no sus colores o texturas. De esta forma el número de materiales a usar en una escena estándar es relativamente pequeño y perfectamente almacenable en los parámetros de entrada de los shaders utilizados.

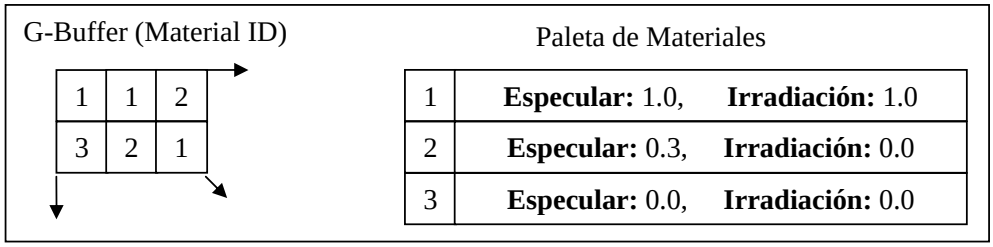

Figura 24. Uso de una paleta de materiales para describir los atributos del material en cada píxel del Buffer de Geometría.

En el caso de que el número de materiales y atributos a almacenar exceda la cantidad de parámetros de entrada de los shaders del hardware gráfico usado, es posible usar una solución alternativa. En lugar de introducir la paleta de materiales como parámetros constantes en los shaders, se puede almacenar esta en una textura donde cada línea horizontal representa un material y contiene todos los atributos de estos. Esta textura es usada en los *shaders* de iluminación para desempaquetar los valores necesarios de los materiales. Para esto se extrae el *Material_ID* del Buffer de Geometría y se busca la fila de la textura indexada con ese valor.

Comentario [c6]:

Esta solución permite especificar un mayor número de materiales, pero presenta el inconveniente de usar parte de la memoria de video e impone un retraso adicional al tener que realizar movimientos de la memoria de video a la caché gráfica para extraer los valores deseados de la textura. Sin embargo una de las ventajas que brinda este sistema es que los valores de los atributos no cambian durante la simulación. En ese caso, esa textura puede ser creada por los artistas durante la generación de la escena y no en tiempo de ejecución.

Ajuste de la precisión de los parámetros

Usando las optimizaciones anteriores, se ha reducido notablemente el número de valores almacenar en el Buffer de Geometría para representar geométricamente cada píxel. Una solución gráfica común en la que sólo se utiliza la posición, el vector normal a la superficie y el material, es descrita usando 4 valores. De esta forma es posible usar sólo una textura para almacenar estos atributos en sus canales (rojo, verde, azul y alpha). Eliminándose la necesidad de usar *Multiple Render Targets,* que como se ha explicado anteriormente, es una herramienta que sólo se encuentra disponible en los últimos procesadores gráficos.

Aún con la eliminación de los Multiple Render Targets, la implementación descrita hasta el momento utiliza texturas de punto flotante de 32 bits de precisión en cada canal (32 *Bits Floating Point Texturas*). Estas texturas también son una de las posibilidades gráficas de los procesadores de video más recientes. En las próximas secciones, se describirán una serie de optimizaciones para eliminar la necesidad de estas texturas y usar formatos más comúnmente soportados en las tarjetas de video.

El uso de texturas flotantes de 32 bits con cuatro canales de almacenamiento ha permitido usar un total de 16 bytes (128 bits) para almacenar los atributos en el Buffer de Geometría. El formato de texturas que más frecuentemente soportan los procesadores gráficos es el llamado R8G8B8A8, donde se cuenta con cuatro canales de valores enteros de 8. El uso de este formato restringe la capacidad

de almacenamiento a tan sólo 4 bytes (32 bits). De esto se deriva que para implementar el Buffer de Geometría con este formato se necesita no tan sólo reducir el número de valores almacenar sino que además es imprescindible reducir la cantidad de bits que describen el valor. A esto se le llama variar la precisión del dato.

Como los valores almacenar en el Buffer de Geometría no serán introducidos con el número exacto de bits que lo describen, sino que serán compactados de forma no conservativa para mantener el alto rendimiento; se deberá comprometer un tanto la calidad gráfica en aras de poder utilizar el sistema en un mayor número de procesadores gráficos.

En la mayoría de las ocasiones, algunas de las optimizaciones que se presentan a continuación son usadas como soluciones auxiliares para los casos en los que la ejecución de la aplicación ocurra en sistemas gráficos que no presenten las capacidades de texturas flotantes de 32 bits. No obstante la precisión de los valores resultantes luego de las próximas optimizaciones ha sido escogida de forma que la calidad gráfica no se vea extremadamente comprometida. De hecho, en la mayoría de las representaciones de escenas

Comentario [c7]:

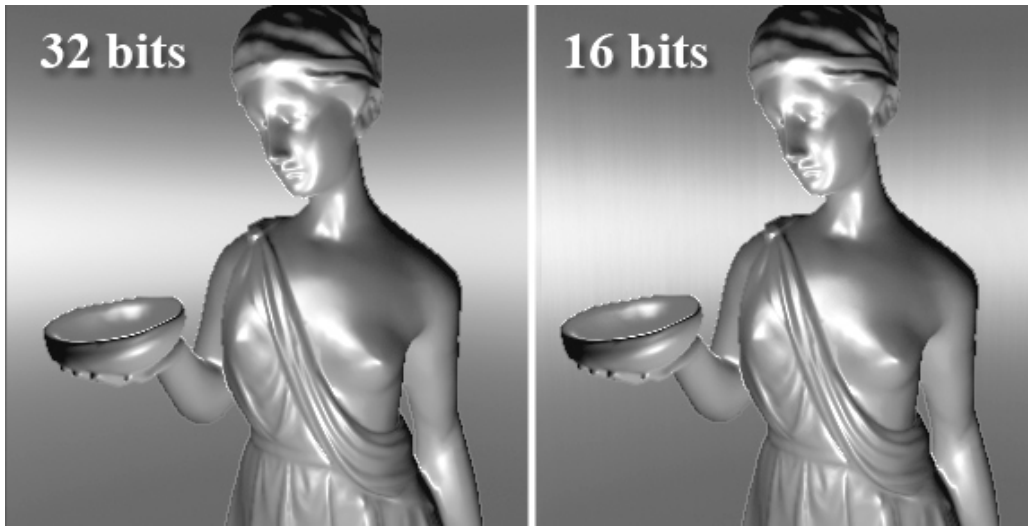

Figura 25. Escena Virtual vista con diferente precisión de datos. a) Alta precisión, b) Datos compactados a una menor precisión.

virtuales es preferible observar un alto número de luces dinámicas interactuando en la escena con algunos pequeños errores de intensidad en lugar de representaciones pobremente iluminadas pero con una asombrosa precisión física en la iluminación. Lo cual, generalmente, apenas es distinguible de la versión de baja calidad de iluminación como se aprecia en la figura 25 donde se muestras dos versiones de la misma escena, una con alta precisión de datos y otra con la menor precisión descrita en este trabajo.

Como un ejemplo de las optimizaciones sólo se almacenará en el Buffer de Geometría la posición y el vector normal a la superficie. Esto permite implementar una alta variedad de luces que respondan a un mismo material. Esta configuración es la implementada comúnmente cuando se escoge la opción de calidad media en la mayoría de los juegos de video.

Empaquetamiento de la Normal

Cuando se utiliza mapa de normales para implementar *bump-mapping*, cada texel de la textura representa un vector de longitud uno que describe la normal a la superficie en ese punto. Los componentes de ese vector son almacenados en los 8 bits de cada canal de la textura. Esa precisión produce tan sólo pequeños errores en las normales a la vez que reduce el número de bits necesarios para almacenarlas, desde 64 bits (dos componentes de 32 bits) a tan sólo 16 (dos componentes de 8 bits). Es por esto que en la generalidad de los sistemas de visualización actuales, las normales se almacenan con una precisión de 8 bits por canal.

No obstante, aunque 8 bits por cada componente del vector normal a la superficie representa un punto aceptable entre la calidad gráfica y el espacio de almacenamiento, también pueden funcionar precisiones menores. En algunas escenas 7, 6 o incluso 5 bits producen también buenos resultados; pero esto último depende estrechamente de la complejidad requerida en las escenas a visualizar. Es por esto que en los ejemplos que se describen a lo largo de este

documento se mantiene 8 bits de precisión como un estándar para almacenar el vector normal a la superficie.

Un punto al que se le debe prestar especial atención es que al reducir la cantidad de bits de cada componente del vector normal, este se almacena como un valor entero sin signo y no como un valor real (punto flotante). Como se ha explicado con anterioridad, cada componente del vector normal se encuentra restringido al rango [-1, 1]. Pero para almacenarlo en un byte este debe encontrarse en el rango [0..255], de esta forma el rango inicial debe ser primero transformado a un rango en el cual no influya el signo. De esta forma se le adiciona uno para llevar el componente al [0, 2] y luego normalizado al dividirlo por 2. El nuevo valor del componente cuyo rango es [0..1] es introducido dentro del *shader* en el valor flotante que representa el canal de la textura en la que se desea almacenar este. El rasterizador de la tubería gráfica se encarga automáticamente de transcribir el valor anterior al rango [0..255]. Pero esta última operación no puede ser llevada acabo directamente en el *shader*, pues las especificaciones de estos indican que los valores resultantes para ser almacenados en *render targets* cuyos componentes sean bytes, deben ser valores flotantes en el rango [0, 1].

Las operaciones descritas anteriormente se implementan usando las siguientes instrucciones en el *píxel shader* de la Fase Geométrica.

```
Gbuffer.xy = (Input.Normal.xy + 1)/2;
```

Ó

```
Gbuffer.xy = Input.Normal.xy * 0.5 + 0.5;
```

Esta segunda forma de escribir las instrucciones realiza el mismo proceso matemático que la primera línea, pero se amolda más elegantemente a la función *mad* del lenguaje ensamblador al que son traducidos los *shaders* cuando son compilados. Esto aporta un beneficio extra al ayudar al compilador a entender mejor el código y realizar optimizaciones más elaboradoras en este.

Empaquetamiento de la Posición

Como se ha especificado el vector posición puede ser almacenado como un valor flotante que representa la distancia desde el ojo de la cámara hasta la posición del píxel. Esa magnitud, no esta restringida al rango [-1, 1] como lo estaban los componentes de la normal. En su lugar, este toma valores en el intervalo definido por los planos que representan la distancia mínima y máxima de visibilidad de la cámara [*Near_Plane*, *Far_Plane*]. Por lo tanto la cantidad de bits que deben ser asignados para almacenar este valor dependerá estrechamente de esas distancias.

El siguiente ejemplo de empaquetamiento asume que las distancias usadas son de 1 unidad como lo visible más cercano y de 1000 como el rango de visibilidad máximo. Esta configuración es la usada en una gran parte de los visualizadores de entornos virtuales, no obstante, otros rangos se pueden implementar usando el mismo procedimiento con la introducción de pequeños cambios para especificar el intervalo de visibilidad usado.

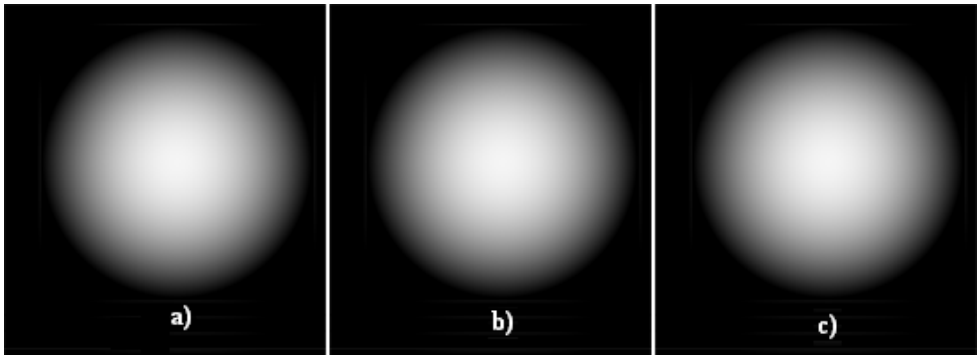

Figura 26. Precisión de la Posición en el Buffer de Geometría: a) 32 bits, b) 24 bits, c) 16 bits

La figura 26 muestra una escena virtual en la que se ha calculado la iluminación mediante la Visualización Retardada usado diferentes precisiones para almacenar el valor de la posición en el Buffer de Geometría. Es notable que la versión de 16 bits no presente tantos errores visuales como se podría esperar inicialmente. De hecho esta versión resulta muy convincente para ser usada

como la opción de media y baja calidad gráfica en las configuraciones gráficas de la aplicación.

La clave para que no existan tantos errores e irregularidades en la escena al reducir la precisión del valor almacenado en el Buffer de Geometría; es que lo salvado en este no es la posición vectorial del píxel ni el componente z de esta, sino la distancia del píxel a la cámara. Lo anterior se traduce al hecho de que si dos píxeles consecutivos almacenan el mismo valor en el Buffer de Geometría, no retornarán el mismo componente z o incluso la misma posición, debido a que la diferencia de los valores de los componentes (x, y) generan diferentes valores z como se muestra en la figura 27. Esta propiedad disimula los errores que se producen debido a la baja precisión del valor que describe la posición del píxel.

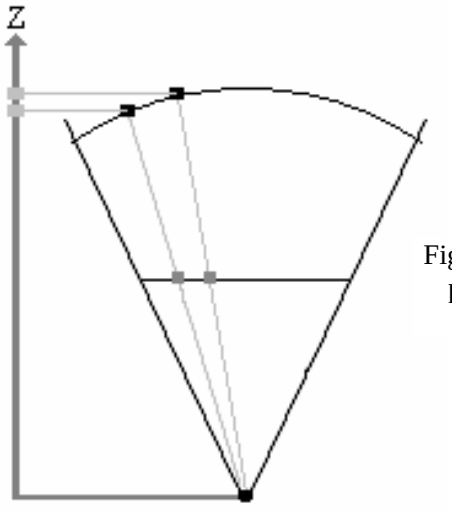

Figura 27. Dos píxeles consecutivos con la misma profundidad en el G-Buffer, producen valores diferentes en el componente z de la posición.

La siguiente función es usada en los píxel shaders de la fase de geometría para compactar la distancia del ojo de la cámara a la posición del píxel en dos bytes, lo cual es equivalente a 16 bits de precisión.

```
float2 PackFloat16( float depth )
{
    depth /= 4;
    float Byte1    = floor(depth);
    float Fraction = frac(depth);
    return float2( Byte1 / 256, Fraction );
}
```

La función anterior recibe como parámetro la distancia del píxel a la cámara. Esta profundidad se dividirá en dos partes, la parte entera y la parte fraccionaria. Según las restricciones que se impusieron inicialmente, la distancia recibida se encontrará en el intervalo [1, 1000], por lo tanto la parte entera nunca será mayor de 1024. De ahí que tan sólo se necesiten 10 bits para almacenarla, permitiendo utilizar los 6 bits restantes para la parte fraccionaria.

El número será almacenado en dos bytes. De esta forma el primero contendrá los 8 últimos bits de la parte entera. Mientras que el segundo los primeros 2 bits que describen la parte entera junto con los 6 bits de la parte fraccionaria, como se aprecia en la figura 28.

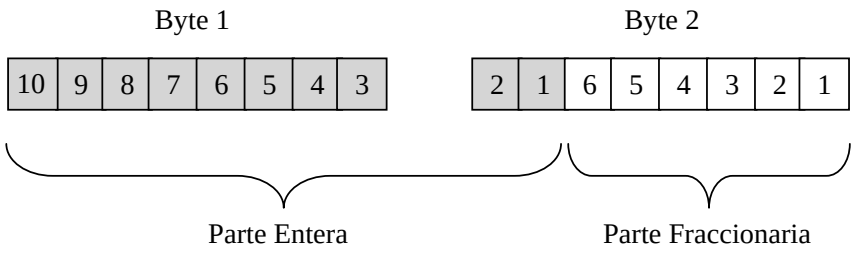

Figura 28. Distribución de la profundidad en 16 bits.

Para introducir los 8 últimos bits del número, primeramente se debe correr hacia abajo 2 bits de los 10 que describen inicialmente la parte entera de este. Esto se realiza dividiendo la distancia inicial entre 4. La función intrínseca floor(x) extrae la parte entera del resultado que luego de la división anterior se encontrara descrito a lo máximo por 8 bits por lo que puede ser introducido en el primer byte del resultado. Es de destacar que al igual que sucedió con la normal el resultado deseado es un byte, o sea el rango [0..255], pero los shaders especifican que la salida debe ser realizada en el rango [0..1] y el rasterizador de la tarjeta de video realizará automáticamente el mapeo al intervalo [0..255]. De ahí que finalmente se divida el primer valor entre 256 para llevar el número al rango normalizado.

Una vez que se realizó la división entre 4, la parte fraccionaria contendrá los primeros 2 bits de la parte entera junto con los que describen la parte fraccionaria inicial. Utilizando la función frac(x) esta es extraída e introducida directamente en el segundo byte. Y como este valor ya se encuentra en el rango [0, 1] no se necesita normalizar.

La siguiente función muestra el procedimiento para desempaquetar la distancia del ojo de la cámara al píxel durante la fase de iluminación, dados los dos bytes almacenados en el Buffer de Geometría.

```
float UnpackFloat16( float2 depth )
{
   const float2 unpack = {1024.0f, 4.0f};

   return dot(unpack, depth);
}
```

La función intrínseca dot(a, b) realiza el producto escalar entre dos vectores. En este caso se trata de vectores con dos componentes, uno de ellos contiene los valores {1024, 4} y el otro los bytes en el rango [0..1] extraídos del Buffer de Geometría.

$$\mathrm{Pr}\,ofundidad\_Original = dot\left(\begin{bmatrix}1024\\4\end{bmatrix},\begin{bmatrix}byte1\\byte2\end{bmatrix}\right)$$

$$\mathrm{Pr}\,ofundidad\_Original = 1024\cdot byte1 + 4\cdot byte2$$

$$\mathrm{Pr}\,ofundidad\_Original = (256*4)\cdot byte1 + 4\cdot byte2$$

La ecuación anterior rige el proceso de extracción de la distancia original. Primeramente se multiplica el primer byte por 1024 (o sea, por 256 para llevar el número del rango [0,1] al rango [0,256] y luego nuevamente por 4 para correr los 8 bits de este dos veces a la izquierda). Posteriormente se le suma el segundo byte multiplicado por 4 para obtener los dos bits iniciales de la parte entera junto con la parte fraccionaria.

Optimizaciones adicionales

Aún cuando los códigos mostrados hasta el momento han sido optimizados para permitir su ejecución en procesadores gráficos con menores prestaciones, estos no han sido completamente optimizados para mantener un fácil entendimiento del funcionamiento de la Visualización Retardada.

Existen numerosos trabajos sobre como optimizar los shaders que pueden ser aplicados satisfactoriamente en los programas mostrados anteriormente. De esa forma se pueden introducir otras optimizaciones que aunque no son tratadas directamente en este documento, pues no optimizan el proceso relacionado con la Visualización Retardada sino el código general; permiten obtener mejores rendimientos.

Entre algunas de estas, se encuentran optimizaciones relacionados con el cálculo de las ecuaciones de iluminación como es el caso del precálculo del factor de atenuación en las luces que puede ser almacenado en una textura y accedido directamente ahorrándose en este proceso algunas instrucciones aritméticas.

Además el uso del tipo de datos half en lugar del float cuando sea posible prescindir de la precisión de 32 bits que proporcionan estos últimos. Half es un tipo de dato flotante con tan sólo 16 bits, que resulta muy adecuado para almacenar valores como la normal, los índices a los materiales y la distancia del ojo a la cámara cuando esta es empaquetada en 16 bits. La ventaja del uso de este tipo de datos es que los compiladores de los vertex y píxel shaders pueden realizar optimizaciones más agresivas, mezclando operaciones y cálculos lo que reduce notablemente el número de instrucciones a ejecutar y aumenta el rendimiento total del código.

A su vez, optimizaciones tan simples como el reajuste del orden en que son ejecutadas las instrucciones pueden significar una gran diferencia en el

rendimiento total. Al permitir aprovechar más eficientemente la naturaleza puramente paralela de los procesadores gráficos así como evitar los conflictos y las demoras productos de la caché al requerir datos desde texturas y que se usen inmediatamente luego de la petición, lo cual impone que el procesador detenga la ejecución del código hasta que el dato esté disponible.

De esta forma, un gran número de optimizaciones generales de los shaders pueden ser aplicadas directamente en los programas que se ejecuten en un Sistema de Visualización Retardada, para producir códigos que se ejecuten más rápido y usen de formas más eficientes los recursos disponibles en los procesadores gráficos.

Ejemplo de Deferred Shading Optimizado

Fase Geométrica

Vertex Shader:

```
float4x4 matViewProjection;
float4x4 matWorldView;

struct VS_OUTPUT
{
   float4 Pos        : POSITION;
   float3 Normal     : TEXCOORD0;
   float3 WorldPos   : TEXCOORD1;
};

VS_OUTPUT vs_main( float4 inPos: POSITION, float3 inNormal: NORMAL)
{
   VS_OUTPUT Out;

   Out.Pos    = mul( matViewProjection,  inPos);
   Out.Normal = mul( matWorldView,       inNormal);
   float4 pp  = mul( matWorldView,       inPos );

   Out.WorldPos = pp.xyz / pp.w;

   return Out;
}
```

Píxel Shader:

```
struct PS_INPUT
{
   float3 Normal   : TEXCOORD0;
   float3 WorldPos : TEXCOORD1;
};

float4 ps_main( PS_INPUT Input ) : COLOR0
{
   float4 G_Buffer;

   G_Buffer.xy = normalize( Input.Normal ).xy;
   G_Buffer.z  = length(Input.WorldPos);

   G_Buffer.w = 0;

   return G_Buffer;
}
```

Fase de Iluminación:

Vertex Shader:

```
sampler2D G_Buffer;

struct PS_INPUT
{
    half2 texCoord      : TEXCOORD0;
    half3 EyeScreenRay  : TEXCOORD1;
    half3 LightPos      : TEXCOORD2;
};

half  InvSqrLightRange;
half4 DiffuseLightColor_0;

half UnpackFloat16( half2 depth )
{
   const half2 unpack = {1024.0f, 4.0f};

   return dot(unpack, depth);
}

half4 ps_main( PS_INPUT Input ) : COLOR
{
   half4 G_Buffer = tex2D( G_Buffer, Input.texCoord );

   // Compute pixel position


   half  Depth = UnpackFloat16( G_Buffer.zw );
   float3 PixelPos = normalize(Input.EyeScreenRay.xyz) * Depth;
```

```
    // Compute normal

    half3 Normal;
    Normal.xy = G_Buffer.xy*2-1;
    Normal.z = -sqrt(1-dot(Normal.xy,Normal.xy));

    // Computes light attenuation and direction

    float3 LightDir = (Input.LightPos - PixelPos)*InvSqrLightRange;
    half Attenuation = saturate(1-dot(LightDir, LightDir));
    LightDir = normalize(LightDir);

    // Lighting equation

    half  DiffuseInfluence = dot(LightDir, Normal)*Attenuation;
    return DiffuseLightColor_0 * DiffuseInfluence;
}
```

Píxel Shader:

```
float2 ViewportDimensions;
float TanFOV;

float3 LightPos_0;

float4x4 matWorldView;

struct VS_OUTPUT
{
    float4 pos          : POSITION0;
    float2 texCoord     : TEXCOORD0;
    float3 EyeScreenRay : TEXCOORD1;
    float3 LightPos     : TEXCOORD2;
};

VS_OUTPUT vs_main( float4 inPos: POSITION )
{
    VS_OUTPUT o = (VS_OUTPUT) 0;

    // Compute vertex positions of a screen aligned quad

    inPos.xy = sign( inPos.xy);
    o.pos = float4( inPos.xy, 0.0f, 1.0f);

    // RenderMonkey don't have a ViewAspect variable,
    // but it can be computed when on the engine, the
    // ViewAspect is provided instead of the Dimensions

    float ViewAspect = ViewportDimensions.x/ViewportDimensions.y;

    o.EyeScreenRay = float3(inPos.x * ViewAspect, inPos.y, TanFOV);
```

```
    // get the screen coords into range [0,1]
    o.texCoord = float2(o.pos.x, -o.pos.y) * 0.5 + 0.5;

    float4 p = mul( matWorldView, float4(LightPos_0,1) );
    o.LightPos = p.xyz/p.w;

    return o;
}
```

Optimizaciones de Alto Nivel

Implementando las optimizaciones mostradas anteriormente en las diferentes *shaders* de cada fase de la Visualización Retardada ayuda a soportar un amplio número de procesadores gráficos así como mejorar el rendimiento total del sistema. Sin embargo, si el proceso es implementado exactamente como se ha especificado y cada foco luminoso es enviado a la fase de iluminación como un cuadrado alineado con la cámara y que cubre exactamente toda la pantalla; sin realizar ningún otro tipo de optimización que trabaje a un nivel más alto, el sistema podría incurrir en un cuello de botella en el rasterizador al sobrecargarse producto del alto número de píxeles procesados.

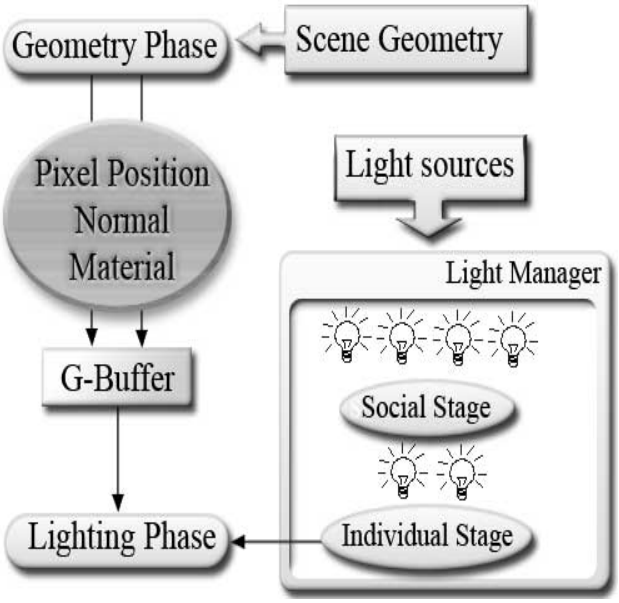

Figura 29. Funcionamiento del *Deferred Shading* usando un Manejador de Alto Nivel.

En las próximas secciones se definirá como implementar un manejador de luces (Light Manager ó *High Level Manager*) que funcione como un *firewall* que filtre los datos enviados desde el Motor de Visualización del Sistema y los *shaders* de la fase de iluminación, como se muestra en la figura 29. Este manejador de luces es implementado usando dos fases principales: La fase social y la fase individual.

Fase Social

En esta primera fase, el sistema accede a todos los focos de luces y aplica optimizaciones generales para reducir el número de estas que serán procesadas. Para ello, cada foco mantiene información actualizada sobre datos generales como pueden ser el factor que representa cuan brillante es la luz y una Geometría Auxiliar, describiendo que zonas del mundo virtual son afectadas

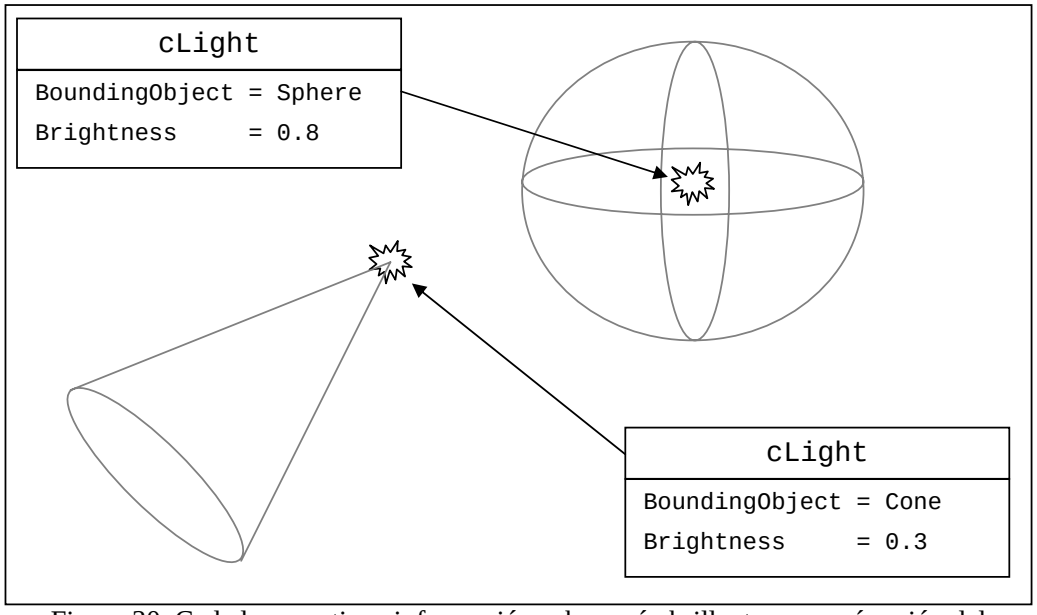

Figura 30. Cada luz mantiene información sobre cuán brillante es y qué región del espacio es afectada por ella.

por esta (*Light Bounding Object*). Un ejemplo visual de lo anterior se muestra en la figura 30.

Usando la información anterior, durante la Fase Social se ejecuta el código descrito por el siguiente pseudo código.

- Aplicar algoritmos de visibilidad para descartar las luces cuya área de influencia no es apreciable.
- Proyectar los *Bounding Objects* de las luces visibles a espacio pantalla.
- Combinar las luces que están muy cerca en espacio pantalla.
- Descartar las luces que no contribuyen significantemente debido a que su área de influencia en la pantalla es muy pequeña o se halla muy lejos.
- Comprobar que cada región de la pantalla no sea influenciada por más de un número predefinido de luces. Escoger tan sólo las luces más grandes, brillantes y cercanas.

De esta forma, primeramente se descartan que luces son ocultadas por la geometría de la escena o no se encuentran en el campo visual de la cámara, el cual esta definido por el *frustum*. Para esto se ejecutan algoritmos especializados de oclusión y determinación de visibilidad usando los *Bounding Objects* que describen el campo de influencia de la luz.

Comentario [c8]:

Una vez que se ejecuta este proceso, sólo las luces que contribuyen visualmente son enviadas al siguiente paso. Descartándose un conjunto de focos luminosos cuyo procesamiento no retornaría resultados visibles en la imagen final.

Sin embargo, aún cuando sólo se procesan las luces que iluminan píxeles visibles de la pantalla, no todas realizan una contribución significante a la imagen final. En ocasiones es preferible ganar notablemente en eficiencia y velocidad sacrificando un tanto de calidad gráfica. Para esto se ejecutan una

serie de pasos que funcionan en espacio de pantalla. Por lo que todos los *Bounding Objects* que describen el volumen de influencia de la luz son convertidos en figuras bidimensionales que especifican el área de la pantalla afectada por cada uno de los focos luminosos.

Usando las proyecciones de las luces en la pantalla se pueden eliminar el procesamiento de las luces que se encuentren relativamente cerca en espacio de pantalla y que casi afecten la misma región. Estos focos pueden ser cuidadosamente reemplazados por luces más grandes de forma que se afecte la región representada por la unión de las proyecciones originales y con un brillo que simule el resultado de la fusión de los focos iniciales.

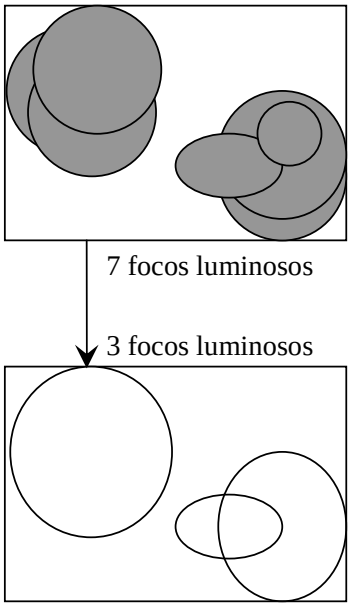

Figura 31. Combinación de Luces similares.

Mediante este procedimiento se reduce el número de luces a representar sin comprometer demasiado la calidad gráfica final, introduciéndose errores aún menos notables si las luces fusionadas se encuentran a gran distancia de la cámara.

Otro de los factores que ayuda aumentar el rendimiento global del sistema es no tomar en cuenta las luces cuya contribución a la escena es muy pobre. En este caso se tienen aquellas cuya área de influencia se restringe a tan sólo unos pocos píxeles de la pantalla o se encuentran muy lejos de la cámara como para ser visibles por esta.

Incluso cuando los pasos anteriores reduzcan la cantidad total de luces a ser procesadas por el sistema, si el rendimiento es aún demasiado bajo. Es posible adjuntar el número máximo de luces que afectarán los píxeles. De ahí que cada punto en la pantalla sólo ejecute los códigos de iluminación para las 8 o 10 luces más influyentes teniendo en cuenta la intensidad de estas. La cantidad máxima

de luces anterior puede incluso disminuirse aún más de ser extremadamente necesario un aumento en el rendimiento.

Este procedimiento brindaría la posibilidad de establecer un costo máximo total en el cálculo de la iluminación. Además permite descartar luces y regiones de influencia que pese a estar cercanas a la cámara su contribución es solapada por un cúmulo de luces más brillantes.

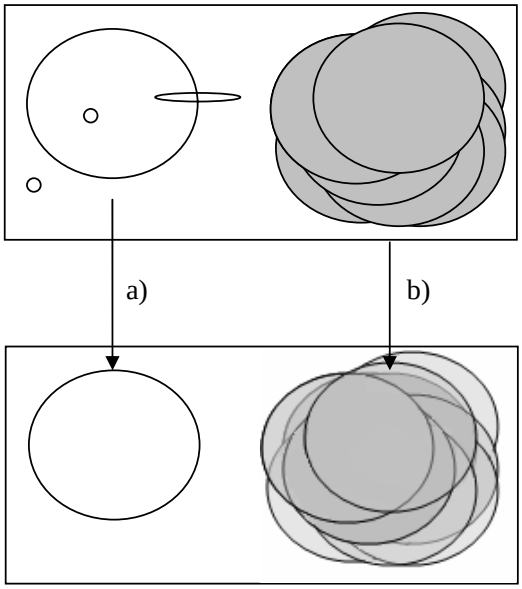

Figura 32. a) Descartar luces que influencian pocos píxeles. b) Fijar número máximo de luces por píxel.

Como ya se ha especificado, la contribución principal de esta fase del Manejador de Luces es la reducción del número de focos luminosos a ser procesados por el sistema. Esta disminución del número de luces trae como consecuencia directa que se realicen menos cálculos durante la fase de iluminación del *Deferred Shading*. Reduciéndose la posibilidad de que ocurran embotellamientos en el rasterizador gráfico debido al alto costo ocasionado por el procesamiento reiterado de los cálculos de iluminación en un alto número de píxeles.

Fase Individual

Al concluir la fase de social, se obtiene una lista de luces ha ser procesadas por el sistema. La siguiente fase del Manejador de Luces recorre esta lista y para cada una de las luces contenidas en ella, realiza una serie de optimizaciones con el objetivo de reducir el costo de procesamiento de esta cuando se ejecute la Fase de Iluminación.

Para realizar esta operación, las luces son primeramente catalogadas en dos grandes grupos: Las luces globales y las luces locales.

Comentario [o9]:

Luces Globales

Las luces globales son aquellas que afectan toda la escena visible. Un ejemplo clásico de este tipo de luz es el sol. Para las luces globales se debe calcular la ecuación de iluminación en cada píxel de la pantalla, pues estas afectan toda la escena.

Dado que en los sistemas de Visualización Retardada el costo de iluminación es proporcional al número de píxeles afectados, entonces este tipo de luces son

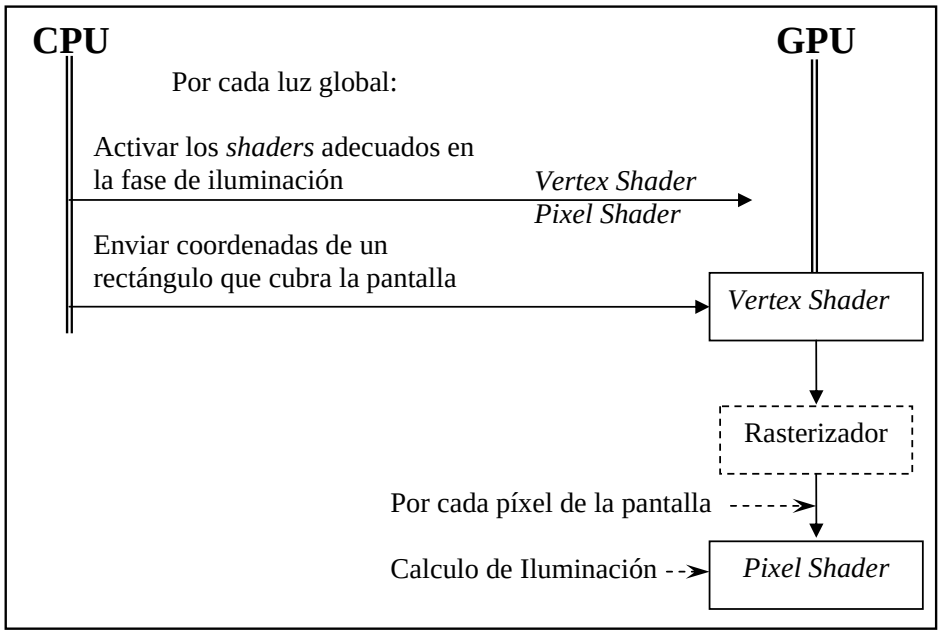

Figura 33. Pipeline de interacción entre el CPU y el GPU par alas Luces Globales

unas de las más costosas. Afortunadamente en la mayoría de las escenas sólo interactúan un grupo reducido de estas.

En las luces globales se debe calcular la ecuación de iluminación en cada píxel de la pantalla. De ahí que se deban ejecutar los *shaders* de la Fase de

Iluminación en cada uno de los píxeles. La forma más sencilla de ejecutarlos en cada píxel de la pantalla es dibujar un rectángulo que cubra exactamente la pantalla, de esta forma el rasterizador del *pipeline* de la tarjeta de video ejecutará los *shaders* para calcular el color resultante en cada píxel que conforma este rectángulo. Lo cual en este caso representa realizar el cálculo de iluminación en cada píxel de la pantalla.

Como se ve, estas luces casi no permiten ejecutar optimizaciones específicas para reducir el número de píxeles a procesar, de ahí que dependan estrechamente de cuan optimizados sean los *shaders* que se utilicen. El procedimiento general para calcular la influencia de las luces globales en la escena, es el siguiente:

Luces Locales

En esta categoría se encuentran las luces que presentan posición y área de influencia. Contrario a las Luces Globales que influenciaban toda la escena, las Luces Locales iluminan sólo una porción de esta. De esta forma, sólo en los píxeles que se encuentran en el interior del *Bounding Object* de la luz se deben realizar los cálculos de la ecuación de iluminación.

Al mantener un área definida de influencia, en la mayoría de las ocasiones, estas luces no afectan todos los píxeles de la pantalla. Lo cual permite realizar algunas optimizaciones para reducir el número de píxeles que se debe procesar durante la fase de iluminación.

A continuación se describen los pasos generales que se realizan durante la Fase Individual del Manejar de Luces:

Por cada Luz Local

- Seleccionar el nivel de detalle apropiado (Light LOD).
- Activar y configurar los *Píxel* y *Vertex Shader* que realizaran el cálculo de iluminación.
- Calcular el máximo y mínimo valor en coordenadas de pantalla del Área de Influencia de la luz.
- Activar la prueba de *Scissor* usando los valores máximo y mínimo anteriores.
- Dibujar el Área de Influencia. (enviar el GPU sus coordenadas)

La prueba de *Scissor* es un mecanismo muy rápido de descartar los píxeles que no sean influenciados por la luz. Usando un rectángulo que cubra la proyección en la pantalla del Área de Influencia de la Luz como la región de *Scissor*, permite descartar algunas de las zonas de la pantalla no afectadas por el foco luminoso. Esto se puede apreciar en la figura 34.

La desventaja de la prueba de *Scissor* es que como el Área de Influencia de las luces no siempre se proyecta a coordenadas de pantalla como un cuadrado, existirán regiones que pese a no ser influenciadas por la luz serán procesadas pues entran dentro del rectángulo de *Scissor*. Usando tan sólo esta prueba, los píxeles que se encuentren dentro de estas regiones son procesados, aún cuando no contribuyen a la iluminación de la escena, pues no se encuentran influenciados por la luz.

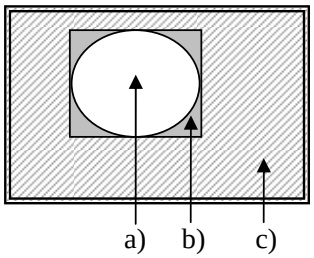

Figura 34. Prueba de *Scissor*. a) Área Iluminada, b) Área procesada sin necesidad. c) Área descartada

En lugar de enviar al GPU un rectángulo que cubra la pantalla, es posible pintar directamente el *Bounding Object* que describe la región de Influencia de la luz. El cual, para un foco luminoso como el descrito en la figura 34, será una esfera. De esta forma el rasterizador sólo generará los píxeles que se encuentran dentro del área proyectada y sólo estos serán procesados durante la fase de iluminación.

Aún cuando esta última solución se acerca más a la ideal donde sólo los píxeles afectados por la luz son procesados, aún deben puntualizarse algunos aspectos. Generalmente los *Bounding Objects* que describen el área de influencia de la luz son volúmenes convexos como esferas, conos, prismas, etc.

Al dibujar los polígonos que conforman estos volúmenes, cada píxel es procesado dos veces. Una cuando se rasterizan las caras frontales y otra al realizar la misma operación para las caras traseras. Esto ocurre pues el rasterizador al proyectar los polígonos funciona como si se lanzara desde la cámara un rayo intersecando el volumen convexo. Las propiedades matemáticas de estos describen que al ser intersecados por una línea, se generan dos puntos de intersección, el de entrada (generada con las caras frontales) y el de salida (generada con las caras traseras).

Si la influencia de la luz en el píxel es procesada y acumulada dos veces, este se sobre-iluminará. De ahí que haya que evitar realizar dos veces el cálculo. Para esto sólo se enviaran al rasterizador las coordenadas de las caras frontales o de las caras traseras. La mayoría de los sistemas de *render* actuales presentan parámetros para controlar por hardware que sólo sean rasterizados los polígonos cuyas caras apuntan a la cámara (caras frontales) o en dirección contraria (caras traseras).

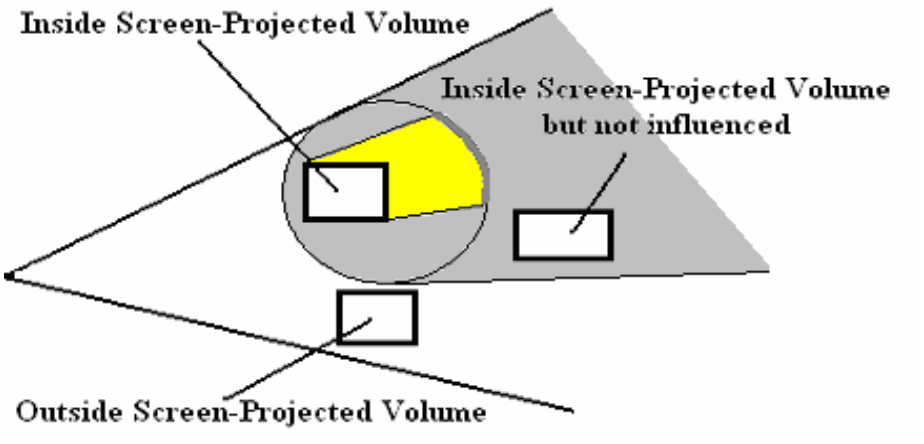

Figura 35. Ubicación de los objetos respecto a la proyección del área de influencia de la luz.

Como se muestra en la figura 35, al proyectar a la pantalla el volumen que describe la región de influencia de la luz se restringe los píxeles a procesar a sólo el área de influencia. Sin embargo, si se procesan todos los píxeles que están dentro de esta área, también se calcularán las ecuaciones de iluminación para objetos que se encuentran lejos de esta y por tanto no influenciados pero cuya proyección cae dentro de esta área en la pantalla.

Una vía de descartar estos píxeles que no reciben influencia de la luz, pero que son proyectados al área de influencia de esta, es usando las diferentes funciones del buffer de profundidad (Z-Buffer).

En el caso de que se estén procesando sólo las caras traseras de la región convexa del *Bounding Object* de la Luz, estas pueden ser dibujadas usando la función *GREATHER_THAN* del Z-Buffer para descartar los píxeles que se encuentran “flotando en el aire”. De lo contrario, cuando se procesen las caras frontales, se puede seleccionar la función *LESS_THAN* para descartar los píxeles que se encuentren “enterrados bajo la superficie”.

La elección de procesar las caras frontales o traseras del *bounding object* convexo de la luz dependerá de las proporciones en las que se encuentren en la escena los objetos intersecando otros o flotando.

Lo anterior proporciona resultados cercanos al ideal en la mayoría de las escenas virtuales. Sin embargo, cuando es posible contar con procesadores gráficos avanzados se cuenta con una tecnología llamada *Dynamic Branching*. Esta característica permite evaluar en el *píxel shader* de la fase de iluminación si el píxel se encuentra dentro del radio de influencia de la luz en caso de que este sea una esfera o si se encuentra dentro del volumen de esta. En caso negativo el *dynamic branching* permite interrumpir los cálculos de iluminación de este píxel sin realizar más operaciones.

Uniendo esta potencialidad con las anteriores, la cantidad de píxeles obtenidos no sólo se ajusta al ideal, en donde sólo los píxeles influenciados son

procesados, sino que además es acelerado el proceso de selección usando técnicas básicas de los gráficos por computadoras.

En los procesadores que no soportan de forma intrínseca *Dynamic Branching*, es posible simularla usando el *Stencil Buffer* y creando máscaras de selección según determinadas reglas definidas con anterioridad y que describirán cuales de los píxeles se encuentran dentro de la región de influencia. Luego sólo se procesan los píxeles de la escena que sean proyectados en esta máscara. Este proceso se describe en detalle en los papeles del SDK de la ATI [ATI00] y proporciona una técnica muy útil para implementar no sólo el proceso de selección sino también otros efectos especiales para los cuales la Visualización Retardada ofrece un gran número de facilidades.

La fase individual del Manejador de Luces también realiza otra optimización no conservativa al proceso de iluminación. Esta es ejecutada al inicio de la fase y permite ajustar la calidad visual de la representación en función del rendimiento esperado del sistema. Su funcionamiento general se basa en los niveles de detalle (LOD) usados para dibujar los objetos de la escena.

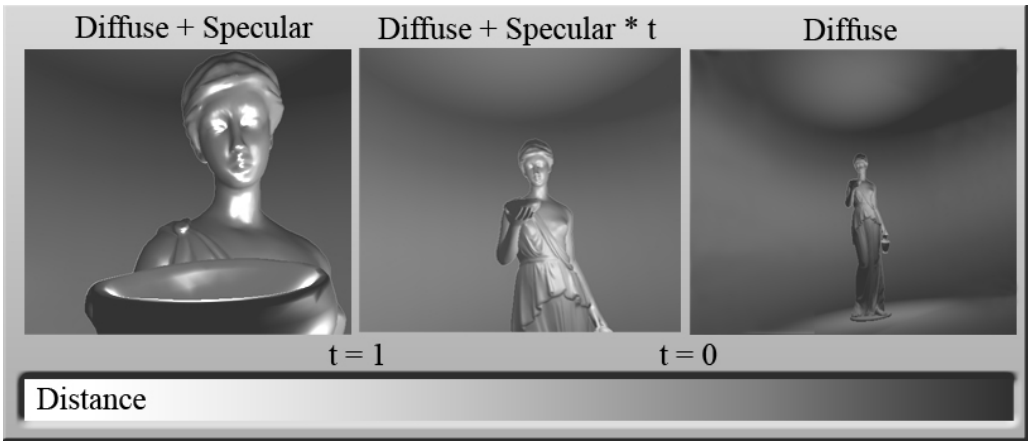

Figura 36. Niveles de detalle de la luz según la distancia a la que se encuentre de la cámara.

De esta forma, las luces que se encuentran muy lejos de la cámara no son calculadas con la misma calidad que las luces cercanas a esta. En la figura 36 se muestran tres niveles de detalle.

Como se puede apreciar el primer nivel de detalle se activa cuando el píxel a procesar se encuentre a una distancia cercana a la pantalla. En este primer nivel la luz es calculada usando la configuración de mayor calidad gráfica, en el ejemplo mostrado se usa una mezcla de contribución difusa y especular. Este nivel de detalle es el más costoso en cuanto a rendimiento del sistema, pues se realizan numerosos cálculos para determinar la influencia exacta de la luz en el píxel.

Cuando el píxel procesado se encuentre a una distancia mucho mayor de la cámara se activa otro nivel de detalle. Pongamos como ejemplo el nivel 3. En este nivel sólo se procesa la contribución difusa de la luz, pues de todas formas si a esa distancia se calculara también la contribución especular, esta no aportaría mucho detalle a la escena ya que el objeto se encuentra demasiado lejano como para apreciar los detalles de esta última contribución. Sin embargo al eliminar la influencia especular de la ecuación, esta se simplifica notablemente, acelerando el rendimiento y disminuyendo el tiempo requerido para iluminar estas regiones.

Para representar los niveles de detalle, cada uno de estos almacena el rango de activación. Por ejemplo, el nivel 1 se usa para los píxeles que se encuentren a una distancia menor de 20 metros y el nivel 2 para los que se encuentren entre los 20 y lo 40 metros; y así sucesivamente.

El problema principal del uso de esta optimización ocurre cuando se cambia de un nivel a otro. Normalmente a la distancia seleccionada en el nivel de detalle la ecuación usada para calcular la influencia de la luz en el píxel debe producir resultados casi indistinguibles de los de la ecuación de alta calidad (nivel 1), dado que el objeto representado se haya a una cierta distancia. Sin embargo pese a que el ojo acepta estos resultados casi como si fuesen los mismos,

cuando se transita de un nivel a otro ocurre un cambio brusco en la intensidad del píxel y ese cambio si es visible.

Para resolver este problema, se pueden introducir niveles de transición o intermedios donde se parte de la influencia del nivel inicial y se transita a la influencia del próximo nivel. Un ejemplo de este tipo de transición se aprecia en el segundo nivel de la figura 36. Este hace una transición desde el cálculo de la contribución especular y difusa hasta que la especular desparece de la ecuación. Para esto multiplica el componente especular por un valor t que toma valores entre 0 y 1 y representa cuan avanzada se encuentra la transición.

De esta forma si el nivel de transición comienza a los 20 metros, el componente t será de 1 y la ecuación se calculará completamente, a medida que la distancia aumenta, disminuye el valor de t hasta que se llega a la frontera de activación del nivel siguiente que en este caso serían los 40 metros. En ese punto, el valor de t es 0 y por tanto el componente especular ya no se le suma a la influencia de la luz en el píxel. Como no existe un cambio brusco en la intensidad, sino que esta es interpolada entre el final del nivel 1 y el inicio del nivel 3; entonces el cambio de nivel apenas será percibido.

La implementación de esta optimización en el sistema de Visualización Retardada se produce configurando los *shaders* de la fase de iluminación para que se ejecuten las ecuaciones correspondientes. Para esto se puede contar con diferentes versiones de los *shaders* de esta fase y según la distancia y el radio de influencia del foco luminoso, se activa el *shader* que calcula la ecuación deseada para el nivel de detalle que represente la distancia del punto más cercano del área de influencia de este.

Optimizaciones Adicionales de Alto Nivel

El manejador de luces descrito en las secciones anteriores, representa un gran paso de avance en la optimización del *Deferred Shading*. Sin embargo aún

existen muchas otras técnicas que pueden ser combinadas con el sistema de Visualización Retardada presentado para obtener resultados aún más óptimos.

El *Deferred Shading* permite implementar luces dinámicas a nivel de píxel. O sea, luces que se mueven y cuya ecuación de iluminación es evaluada para cada píxel que se encuentre en el área influenciada por esta. Pero aún cuando la tendencia de los sistemas gráficos es aumentar exponencialmente el número de luces dinámicas de la escena, muchos de los focos luminosos que influencian el mundo virtual tienen una naturaleza puramente estática.

Ejemplo de estas luces estáticas son: las lámparas de una casa, el Sol cuando su movimiento no es implementado, etc. Una de las propiedades más prometedoras de este tipo de luces es que su influencia en los objetos estáticos que la rodean es constante. De esta forma es posible precalcular su contribución en estos y almacenarlos en texturas de luces llamadas *LightMaps*.

Los mapas de luces reducen el *fillrate* del sistema pues la ecuación de iluminación no necesita ser calculada para los píxeles de los objetos estáticos de la escena. Estos objetos en la mayoría de las visualizaciones virtuales representan un alto por ciento de la geometría del mundo.

Aún cuando el uso de texturas de luces optimiza el proceso de iluminación, también aumenta la cantidad de memoria requerida por el sistema para almacenar estas texturas. Cuando el rendimiento es extremadamente necesario y se puede gastar un poco más de memoria, los mapas de luces se presentan como una herramienta con grandes potencialidades.

Otra técnica que permite reducir el *fill-rate* del sistema es el uso de iluminación por vértices en lugar de por píxel cuando los objetos presenten una alta teselación (gran cantidad de polígonos) o se hallen lejos de la cámara. Esto funcionaría como los niveles de detalles de la luces (LOD) en el que se distribuye el cálculo de iluminación, visualizando con mayor detalle las luces más cercanas mientras que se sacrifica un tanto la calidad de las más lejanas.

En el caso de la iluminación por vértice, la ganancia en velocidad viene dada el cálculo de la iluminación en cada vértice y luego interpolando el resultado para cada píxel que conforman los polígonos del objeto. Cuando el objeto se encuentra suficientemente teselado el resultado visual es muy semejante al obtenido aplicando iluminación por píxel. Es por esto que en los objetos más alejados de la cámara la pérdida en calidad gráfica no es extremadamente notable.

4

CAPÍTULO

PRUEBAS Y RESULTADOS

En el siguiente capítulo, se presentan los resultados obtenidos al aplicar las optimizaciones discutidas en este trabajo. Además, se realizan comparaciones entre las soluciones de iluminación más significativas en cuanto al rendimiento

total y el uso de la memoria. Resaltándose el impacto que tienen las mejoras presentadas en el *pipeline* de iluminación.

Rendimiento

En los capítulos anteriores se ha presentado una solución para la iluminación de entornos virtuales de alta complejidad en tiempo real mediante el uso de la Visualización Retardada (*Deferred Shading*). Esta solución aumenta el rendimiento de sistema al no ser necesario realizar las trasformaciones geométricas de los vértices de escena repetidas veces para calcular la contribución de las luces.

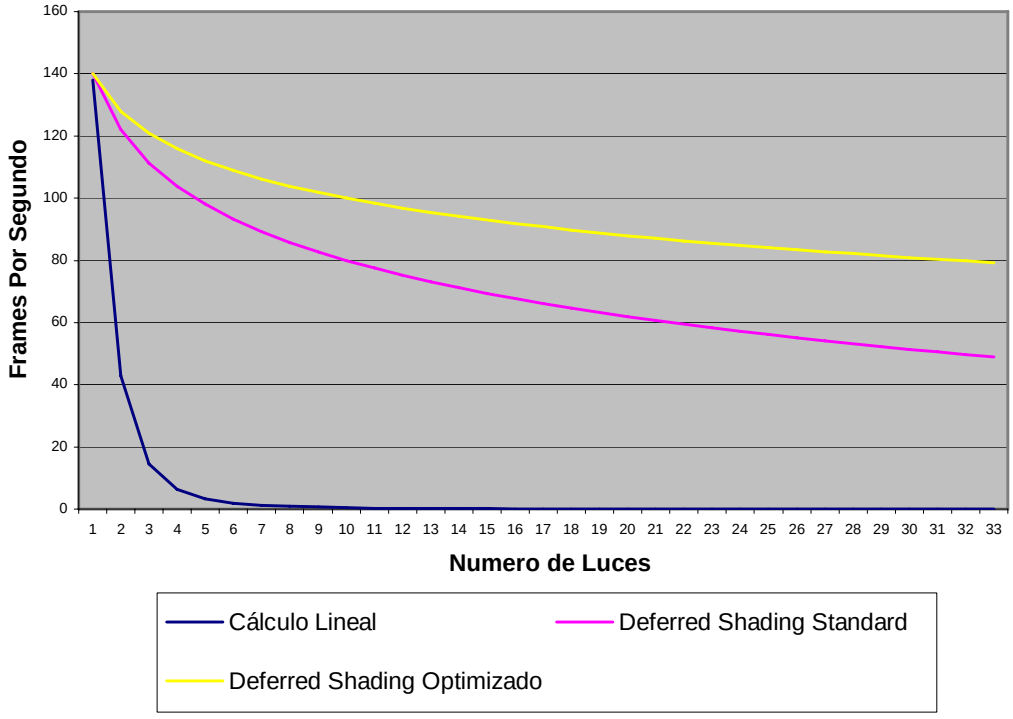

Gráfico 1. Comparación del rendimiento alcanzado por diferentes técnicas de iluminación según la cantidad de luces que influencian la escena.

En el gráfico 1, se muestra el rendimiento de tres soluciones de iluminación en tiempo real; La visualización lineal, la visualización retardada y la visualización retardada optimizada usando las implementaciones descritas en este trabajo.

El algoritmo de iluminación más usado en la actualidad, es la iluminación lineal. Esta solución aboga por procesar la escena para cada luz. Activándose los *shaders* necesarios para el cálculo de la influencia de la luz y se envía la escena a la tarjeta de video. De esta forma se realiza repetidamente el cálculo de transformaciones geométricas y de la influencia para cada foco de luz.

Esta solución tiene la ventaja de ser muy sencilla de implementar pero como se aprecia en el gráfico 1a, el rendimiento a medida que aumenta el número de luces tiende muy rápido a cero, de ahí que no sea una solución efectiva para representar gran número de luces en la escena.

Otra solución de iluminación es la Visualización Retardada. Como se aprecia en el gráfico 1b, este sistema mantiene un comportamiento más estable y eficiente. Permitiendo visualizar un gran número de luces sin disminuir el rendimiento tan bruscamente como en la solución lineal. Este comportamiento más estable se produce al eliminar el procesamiento de la geometría de la escena para cada luz. En este sistema la escena sólo se procesa una vez y para cada foco de luz sólo se transforma un rectángulo que cubre la pantalla.

Durante el desarrollo de este documento, se han especificado un conjunto de optimizaciones para mejorar el rendimiento de la Visualización Retardada y disminuir sus efectos secundarios. El gráfico 1c muestra el rendimiento alcanzado al usar las optimizaciones presentadas. Es de destacar que gran parte de estas se encuentran dirigidas a disminuir el uso de la memoria, con el objetivo de permitir ejecutar la solución en sistemas con menores prestaciones.

En general, las optimizaciones que conllevan a un uso más racional de memoria también reducen el rendimiento del sistema, al ser necesarias

operaciones extras para empaquetar y desempaquetar los valores almacenados en el Buffer de Geometría.

Sin embargo, como se aprecia en el gráfico 1c, el rendimiento de la Visualización Retardada una vez que se optimiza según se presenta en este documento es aún más eficiente que el Deferred Shading Clásico. Esto se debe a que una segunda parte de las optimizaciones están destinadas a disminuir el procesamiento intensivo de píxeles (disminuir el Fill-rate); descartando los píxeles que no necesitan ser calculados o evaluando cuan complejos deben ser los cálculos según heurísticas que definan las propiedades de visibilidad de estos.

Además, como se verá a continuación, la disminución de la memoria también tiene un impacto indirecto en el rendimiento gráfico del sistema.

Uso de la Memoria

Comúnmente, las aplicaciones tridimensionales en tiempo real son ejecutadas en pantallas con una resolución entre 800x600 y 1600x1200 lo que representa entre 480,000 y 1, 920,000 de píxeles.

Si durante el Deferred Shading para cada uno de estos píxeles se almacena en el Buffer de Geometría usando tres valores reales para representar la posición del píxel en coordenadas de mundo y otros tres valores reales para representar el vector normal, teniendo en cuenta que cada valor real ocupa 4 bytes; se empleará un total de 24 bytes por cada píxel.

De ahí que en las resoluciones de pantalla típicas se gaste entre 11,520,000 y 46,080,000 bytes. Lo que representa entre 10 y 44 Mega bytes (Mb) de memoria sólo en datos temporales para la visualización.

Ese ha sido uno de los factores que más ha reducido la popularidad del Deferred Shading sobre todo en consolas de video en donde la memoria

disponible es extremadamente limitada (generalmente entre 32 y 64 Mb) y esta debe ser empleada en todas las demás funciones del simulador (almacenaje de texturas, modelos, shaders, scripts, etc.).

El gráfico 2 muestra la cantidad de memoria usada en la implementación original del Deferred Shading. Como se puede apreciar, a medida que la pantalla es configurada para mayores resoluciones; el consumo de la memoria aumenta exponencialmente. Siendo de hasta 44 megas para la resolución de 1600 x 1200 píxeles que es la más extendida en la actualidad en los sistemas de Realidad Virtual y Video Juego.

Como se ha especificado, el uso de la memoria de video no sólo se destina al Buffer de Geometría. También compiten por este recurso, el almacenaje de modelos, texturas, entre otras. De esta forma en una tarjeta de video de 64 o 128 megas la cantidad de memoria restante luego de creado el Buffer de Geometría no sería suficiente para ejecutar eficientemente la aplicación. Hay que tener en cuenta que tan sólo una textura de 512x512 consume 768 Kb, pues cada texel de la textura contiene generalmente tres bytes para describir los componentes rojo, verde y azul. Y a medida que aumentan las expectativas de los usuarios, también aumenta el tamaño de la textura para garantizar la calidad visual que esta aporta.

De esta forma en la actualidad la mayoría de los Sistema Gráficos y de Video Juegos mantienen texturas de 1024 x 1024 e incluso se esta transitando hacia la resolución de 2048 x 2048 de manera masiva. Consumiéndose entre 3 y 12 Mb de memoria de video por cada textura. Estos números disminuyen si la tarjeta de video permite compactar la textura. Sin embargo esta función no es soportada por todas los sistemas gráficos.

Una escena de baja complejidad puede ser fácilmente descrita por más de 15 texturas. Esto desbordaría rápidamente la memoria de video disponible en el sistema y aún se necesitarán almacenar los modelos, el código de los shaders, entre otros.

Cuando la cantidad de memoria disponible en la tarjeta de video no es suficiente para almacenar los recursos de la aplicación, la parte que se considere heurísticamente menos prioritaria es mapeada a un segmento de la memoria RAM o a un fichero del disco duro en caso de que la RAM se encuentre ocupada. Cada vez que la aplicación necesite esos datos que se encuentran temporalmente almacenados en el disco duro o RAM (páginas auxiliares), libera un bloque de la memoria de video llevando este a las páginas auxiliares y trayendo de estas los bloques de memoria que se necesiten.

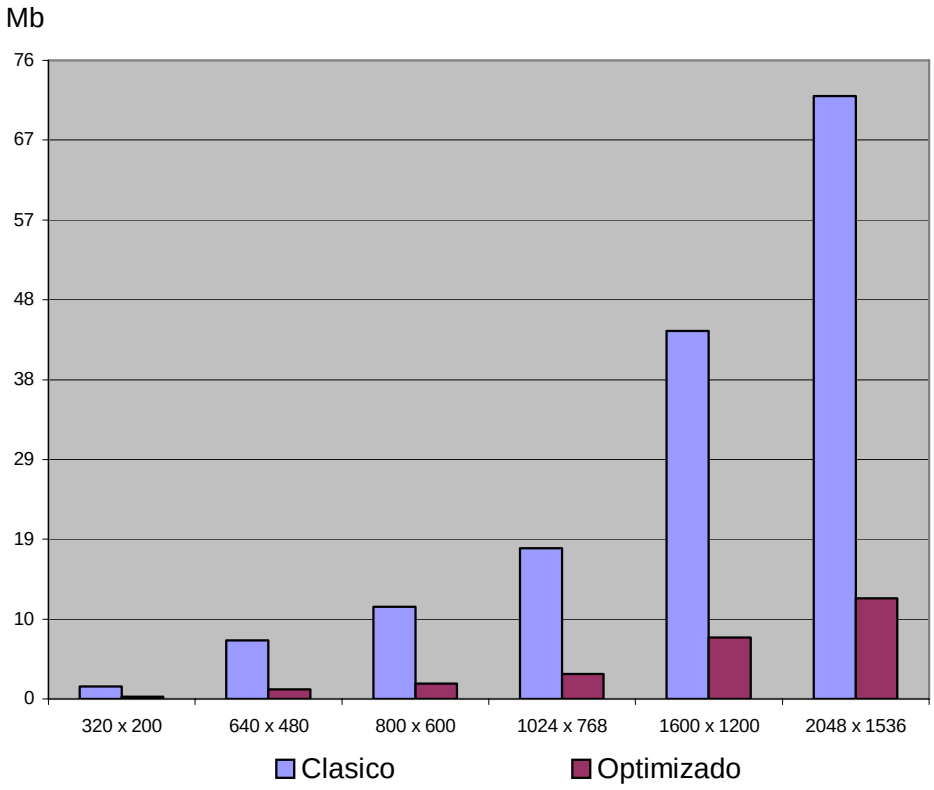

Gráfico 2. Relación de la memoria usada en las diferentes resoluciones de pantalla.

Si en cada frame se necesita acceder a todas las texturas y demás recursos almacenados en memoria, el tiempo de espera que ocurre mientras se realiza los movimientos de reacomodación de la memoria representará un retardo que

incidirá notablemente en el rendimiento del sistema. Esto se aprecia fácilmente en las escenas descritas mediante un gran número de texturas o cuando muchos objetos hagan referencia a una misma imagen, siendo la escena ordenada por objetos y no por materiales.

En el gráfico 2b se muestra la memoria utilizada al realizar las optimizaciones propuestas en este documento a la Visualización Retardada. Como se puede apreciar el consumo de memoria se reduce a tan sólo un 17% de la utilizada por el sistema original. De esta forma, las optimizaciones presentadas brindan la posibilidad de liberar 60 de los 72 Mb utilizados originalmente en la resolución de 2048 x 1536. Siendo posible utilizar esta memoria liberada para almacenar texturas y otros recursos de alta prioridad para el Procesador de Video.

La ganancia reportada por las optimizaciones propuestas no sólo se cuantifican en la reducción de la cantidad de memoria, sino que esta reducción, como se especificó anteriormente, representa un impacto importante en el Rendimiento del Sistema; según se aprecia en el gráfico 1 al reducirse los movimientos necesarios del disco duro o RAM a la memoria de video. Pues mayor cantidad de recursos se podrán almacenar en la memoria de video y no se necesitará realojarlos en las páginas auxiliares.

CONCLUSIONES

Las soluciones tradicionales de iluminación involucran algoritmos de múltiples pasadas donde la escena es procesada por cada foco luminoso o por cada pequeño grupo de estos. A medida que la cantidad de luces dinámicas en la escena aumenta, la carga ocasionada por el excesivo procesamiento geométrico produce un cuello de botella en el pipeline gráfico.

Comentario [o10]:

Este trabajo ha demostrado la importancia de utilizar la Visualización Retardada como una vía para reducir los cuellos de botella en el pipeline gráfico. Sin embargo, la implementación general de esta técnica impone un número de desventajas entre las que se encuentran un alto consumo de memoria y la necesidad de utilizar funcionalidades presentes sólo en los últimos hardwares gráficos.

Para mitigar estos efectos negativos, y cumplir con el objetivo general, se ha realizado un estudio de la Visualización Retardada, aplicada a la iluminación de entornos virtuales. Como resultado de esta investigación se han propuesto un conjunto de optimizaciones y técnicas a la implementación del *Deferred Shading* tradicional.

Entre las optimizaciones y técnicas desarrolladas se encuentran:

1. Optimización de memoria a partir de técnicas de compresión de datos. Empleándose solo un 17% de la memoria utilizada por la técnica original.

2. Controlador de alto nivel para el tratamiento eficiente del procesamiento de las fuentes de luz. Esta nueva solución logra reducir el número de luces a procesar, así como su complejidad matemática y de esta acelerar el proceso de visualización.

Comentario [o11]:

3. Soluciones que permiten la implementación retardada en sistemas que carecen de altas prestaciones.

Siendo este último punto uno de los principales aportes de este trabajo, ya que se brindan soluciones para que sistemas de bajo costo puedan utilizar las ventajas de la visualización retardada. De esta manera una técnica que era prohibitiva hasta hoy, puede ser utilizada en computadoras que no tienen las características de los últimos GPU, lo que permite aumentar los beneficios para diferentes aplicaciones de realidad virtual a bajos costos.

Se han realizado además, un conjunto de pruebas a estas optimizaciones para evaluar la magnitud de las mejoras en cuanto a las desventajas claves del sistema tradicional. Concluyéndose que el modelo propuesto reduce notablemente las desventajas originales. Las pruebas realizadas han permitido demostrar que las hipótesis planteadas al inicio del trabajo eran válidas.

Sin embargo, aún existen muy pocas investigaciones y trabajos en este campo. Esperamos que el perfeccionamiento del hardware gráfico junto a las potencialidades de este sistema, ofrezcan un punto de partida para futuros estudios que involucren resaltar aún más los beneficios del *Deferred Shading*. Pues siguiendo la tendencia al aumento en la complejidad de los mundos virtuales, es posible comprender que La Visualización constituirá una de las soluciones claves para la iluminación de mundos virtuales de la próxima generación gráfica.

BIBLIOGRAFIA

[ATI05] ATI, “Dynamic branching using stencil test”. ATI Software Developer’s Kit, June 2005.

[Calver03] Calver, Dean “Photo-realistic Deferred Lighting” Aparece online en *http://www.beyond3d.com/articles/deflight/*, July 31, 2003.

[Delphi3D] “Deferred Shading” Aparece online en *http://www.delphi3d.net/articles/viewarticle.php?article=deferred.htm*, October 24, 2002.

[Fpuig2006] Frank Puig Placeres, “Fast per-pixel lighting with many lights”, Aparece en el libro “Game Programming Gems 6”, Marzo 2006.

[Geldreich04] Geldreich, Rich “Gladiator Deferred Shading Demo” Aparece online en *http://www.gdconf.com/archives/2004/geldreich_rich.ppt*, March 16, 2004.

[NVIDIA04] Hargreaves, Shawn and Harris, Mark “Deferred Shading” Aparece online en *http://download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues__Deferred_Shading.pdf*, March 26, 2004.

[Pritchard04] Pritchard, Matt “Deferred Lighting and Shading” Aparece online en *http://www.gdconf.com/archives/2004/pritchard_matt.ppt*, March 7, 2004.

[RMonkey05] Render Monkey. ATI