SHADERS

Escrito por:

Schneider, José Ignacio

Directora de la tesis:

Dra. Castro, Silvia

UNIVERSIDAD NACIONAL DEL SUR

CIENCIAS DE LA COMPUTACIÓN

Motivación

La tecnología de gráficos 3D en tiempo real avanza a un ritmo frenético. Nuevas herramientas, metodologías, hardware, algoritmos, técnicas etc. etc. aparecen constantemente.
Desafortunadamente, la documentación existente no va de la mano con el ritmo de crecimiento, en especial la documentación inicial. De hecho, por ejemplo, es muy difícil encontrar una buena definición de que es un shader.
Particularmente sufrí este problema al no tener a mi disposición buena documentación para entender el tema de manera precisa y sencilla. Aún, a pesar de que dedique una gran cantidad de tiempo en buscar material.
Inicialmente mi idea era hablar de un tema específico y de forma puntual, hablando de algoritmos y mostrando cómo se codificarían. Pero me di cuenta que podía aportar mucho más si escribía un documento que permitiera entender que es un shader desde el principio, y que posibilidades nos dan y nos darán en el futuro. Afortunadamente también me ayudo a establecer y pulir mis propios conocimientos.

¿Por qué leerlo?

Lo escrito es una recopilación de infinidad de textos sacados de artículos de internet, libros, foros, junto con mis propias experiencias con mi motor gráfico, sumándole mis conocimientos de cómo funciona el mercado tanto de hardware como de software.
Se intenta abordar un enfoque distinto, que no solo trata de definir conceptos, sino que también nos muestra como esos conceptos son utilizados en aplicaciones reales, permitiéndonos medir la importancia de los mismos.
Mi objetivo principal es que tengan una buena base para entender las distintas tecnologías en uso. Enfocándome principalmente en shaders, aunque no exclusivamente.

Prerrequisitos

Se asume como mínimo el nivel de conocimiento de un alumno de la materia computación gráfica.
Aun así, el libro define y repasa conceptos básicos.
Resulta mucho más conveniente y útil, por lo menos desde mi punto de vista, que el material pueda ser entendido sin que el lector este "afilado" con lo que respecta a conocimientos de computación gráfica.

No existen requerimientos de hardware y software dado que este libro no se dedica a discutir algoritmos específicos.

Material incluido

Incluye un DVD con videos, programas y documentación relacionados con shaders.
Entre otras cosas contiene:

- Las demos de ATI y NVIDIA.
- Videos de motores y juegos populares como Crysis y Heavy Rain.
- Programas de diseño de shaders: FXComposer y RenderMonkey.
- Los SDK de ATI, NVIDIA y Microsoft.
- Los tutoriales y código del Rocket Commander de Benjamin Nitschke.
- Videos, demos y programas de varios tipos.

CAPITULO I
PIPELINE GRÁFICO

Devil May Cry 4

1 Introducción

Antes de explicar que es un shader necesitamos entender cómo trabaja el pipeline gráfico y cómo ha evolucionado.

La existencia de los shaders está altamente ligada a las restricciones que imponían los pipeline gráficos de función fija, los predecesores a la era de los shaders. Entender el funcionamiento del pipeline gráfico resulta indispensable para entender que son los shaders, como funcionan y como implementarlos.

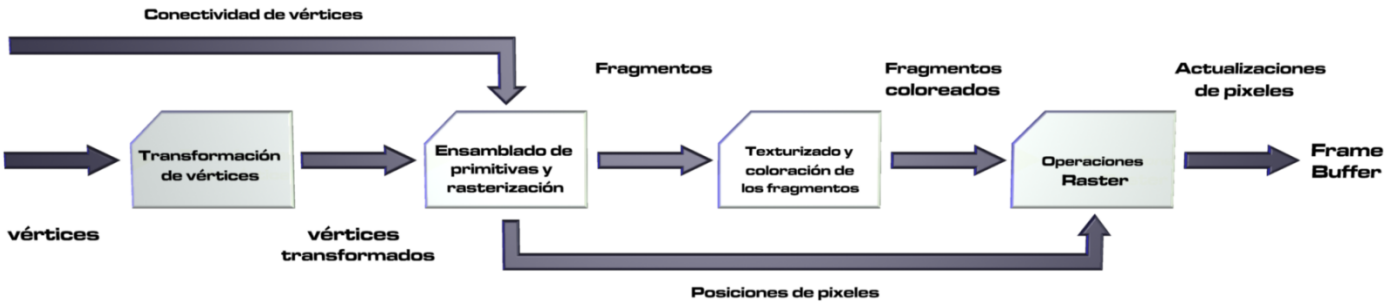
Este capítulo nos explicara entre otras cosas que es un pipeline grafico, cual es el objetivo de cada una de sus partes, que es un pipeline gráfico de función fija y porque existe la necesidad de derivar a uno programable. Para lograr entender todo esto necesitamos complementar esta información con un repaso breve a la historia de las GPUs y de las tecnologías surgidas en cada una de las generaciones de estas GPUs.

1.1 Pipeline gráfico

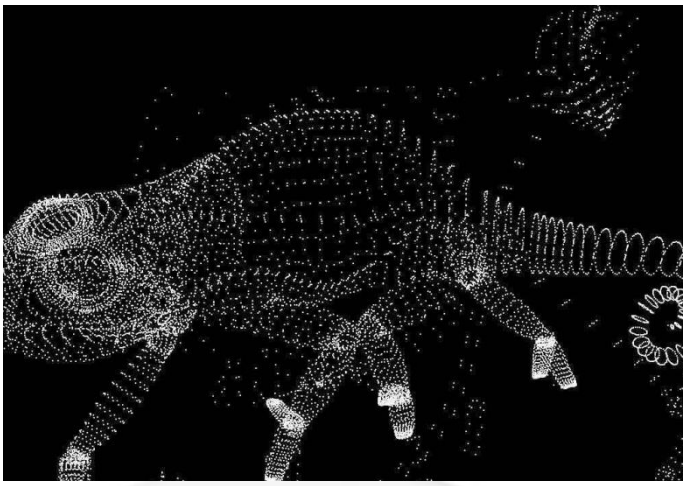
> En computación, un **pipeline**, es un conjunto de elementos o etapas que procesan datos. Estos elementos están conectados en serie y trabajan con el propósito de que la salida de uno de los elementos sea la entrada del próximo. El objetivo del pipeline es ejecutar los elementos en paralelo para aumentar el desempeño.

El pipeline gráfico que explicaremos a continuación es una versión simplificada de un pipeline real, abstrayendo aspectos secundarios que podrían complicar el entendimiento del mismo. Aun así, este pipeline representa el funcionamiento básico de la mayoría de los pipelines gráficos.

La cantidad de etapas con las que se representa un pipeline gráfico puede variar dependiendo del autor. El que describiremos contiene 4 etapas principales:

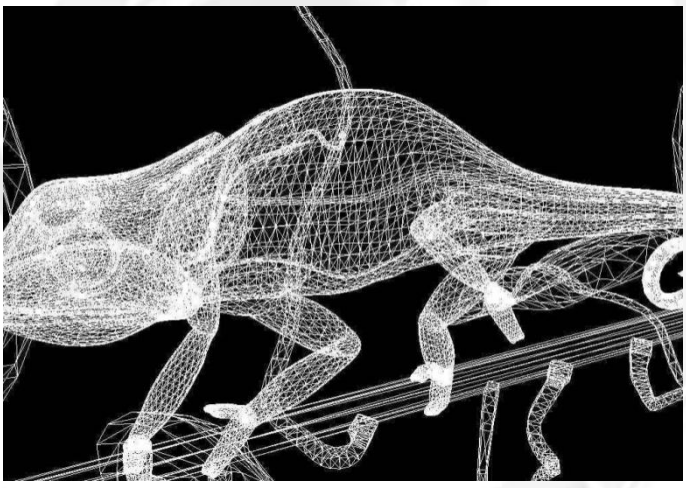


Veamos cada etapa de este pipeline en detalle:



Primero trabajamos sobre vértices

- **Transformación de vértices** (1 vértice -> 1 vértice transformado): la primera etapa realiza una serie de operaciones matemáticas sobre cada vértice. Se encarga de la transformación de la posición del vértice al espacio de pantalla (la posición en que se encuentra con respecto a la cámara) para uso de la etapa de rasterización, también se encarga de la generación de las coordenadas de textura e iluminación del vértice para posteriormente determinar su color.

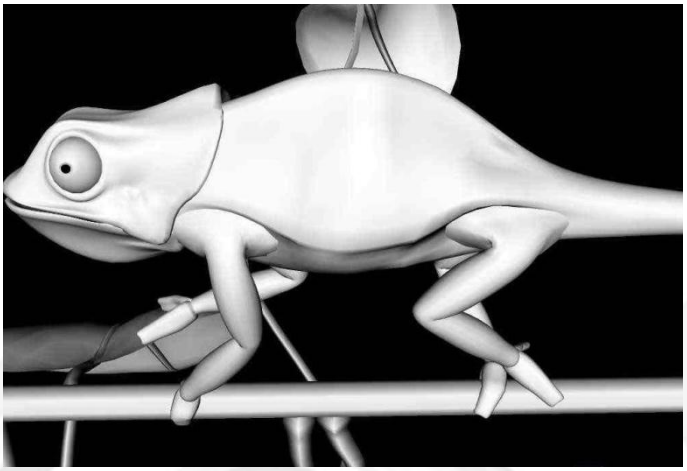


En segunda instancia trabajamos sobre un conjunto de polígonos

- **Ensamblado de primitivas y rasterización** (3 o más vértices -> 1 fragmento): se ensamblan los vértices en primitivas geométricas. La mayoría de los GPU subdividen a las primitivas geométricas de alto orden (primitivas de más de tres vértices) en triángulos. Se aplican técnicas de clipping y culling para reducir el número de polígonos a renderizar, la idea es que si algo no se verá no debe dibujarse. Los polígonos que sobreviven deben rasterizarse, lo que convierte a las primitivas en un conjunto de fragmentos.

Clipping evita dibujar polígonos fuera del rango de visión de la cámara.
Culling (o back face culling) determina si un polígono, dentro del rango de visión de la cámara, es visible. Obviamente, sin considerar el resto de los polígonos de la escena.
Un **pixel** representa el contenido del framebuffer en una determinada localización. El framebuffer contiene la imagen renderizada, en otras palabras, los valores de color para cada pixel.
Un **fragmento**, en cambio, es el estado requerido para actualizar potencialmente un pixel en particular. Un fragmento tiene asociado una localización de pixel, un valor de profundidad, y un conjunto de parámetros

como color, coordenadas de texturas, etc.
Es su definición general, **Rasterización** es la tarea de tomar una imagen descrita en un formato grafico de vectores (en nuestro caso vértices) y convertirlo en una imagen "raster" (pixeles o puntos) para su salida a video o a una impresora. Para nuestro pipeline significa pasar del mundo cuasi infinito de los vectores al mundo finito de nuestra pantalla del monitor.



Por último trabajamos sobre un conjunto de fragmentos

- **Procesamiento de los fragmentos** (1 fragmento -> 1 fragmento coloreado): una vez que una primitiva es rasterizada en un conjunto de fragmentos, y los parámetros de los fragmentos calculados (algunos de los cuales son calculados interpolando los valores de los vértices involucrados), se realiza una secuencia de operaciones matemáticas y de texturizado, para determina el color final de cada fragmento.

- **Operaciones Raster** (1 fragmento coloreado -> 1 actualización de pixel): a cada fragmento se le aplica una cierta cantidad de tests, entre los cuales se encuentran los test de scissor, alpha, stencil y de profundidad. Si alguno de estos test falla, la etapa descarta el fragmento sin actualizar el color del pixel al que corresponde. En esta etapa también se calcula el alpha blending y la niebla, entre otros post procesamientos, si es que están especificados que se calculen. El contenido del frame buffer es el resultado final.

Obtenemos la imagen final, la cual se almacena en el frame buffer

Las etapas no solo toman información de las etapas anteriores, sino que tienen acceso a la memoria de la GPU para buscar y/o modificar datos tales como texturas, el Z-buffer, etc. Además, ciertas etapas pueden tomar datos de más de una ejecución de las etapas anteriores, por ejemplo, la etapa de ensamblado de primitivas y rasterización en cada ejecución utiliza información de tres o más vértices.

De hecho, y para hacer todo un poco más confuso, cada etapa puede tener más de una ejecución corriendo en un mismo instante. Por ejemplo, las GPU tienen varias unidades de procesamiento que calculan la etapa de transformación de vértices, cada una de estas unidades procesando un vértice distinto en el mismo instante. Lo mismo es cierto para otras etapas del pipeline gráfico.

Se puede apreciar que algunas etapas usan información anteriormente calculada. Por ejemplo, la información de un fragmento servirá para actualizar a su pixel asociado solo si no existe otro fragmento con menos profundidad en el Z-Buffer (el test de profundidad).

Aunque en primera instancia resulte confuso el funcionamiento del pipeline gráfico, en el transcurso de las siguientes secciones y capítulos empezaremos a desarrollar en mayor detalle algunas de sus partes.

1.2 Un poco de historia

Repasando la historia de las GPU descubriremos no solo como fueron apareciendo nuevas tecnologías, sino también porque la evolución tecnológica nos llevo a los shaders o, puesto desde otro punto de vista, al pipeline gráfico programable.

La historia de las distintas generaciones de GPUs está simplificada, enfocándonos primordialmente en su evolución funcional.

1.2.1 Voodoo

La Voodoo de 3DFX, aparecida en el año 1996, se la suele reconocer como la primera aceleradora gráfica para el mercado de las PC.

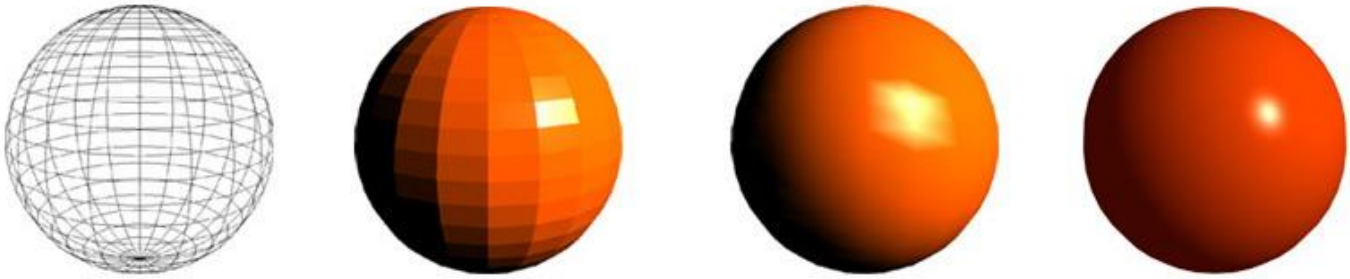
Se limita a procesar solamente triángulos 2D. Esto significa que la primera etapa, transformación de vértices, se realiza completamente sobre la CPU.

La Voodoo posee:

- Un rasterizador que toma como entrada los triángulos 2D pasados por el CPU.
- Una unidad de texturizado y coloreado que utiliza Gouraud shading como sombreado.

Un **modelo de sombreado** es un método de aplicar iluminación local a un objeto. Los siguientes son tres modelos importantes o destacables:

- **Flat**: cada polígono se representa usando el mismo color. Es el modelo más rápido de calcular, pero produce resultados poco realistas.
- **Gouraud**: aplica el patrón de iluminación a cada vértice del polígono y entonces promedia los valores de color en toda la superficie del objeto para así alcanzar un efecto de sombreado suave que parece más realista sobre superficies curvas.
- **Phong**: es mejor que los modelos anteriores pero también es más costoso de calcular. Se utiliza una interpolación bilineal para calcular la normal de cada punto (o pixel en la práctica) del polígono. A cada punto del polígono se le asocia una normal distinta con lo cual su color calculado será muy cercano a la realidad.



Wireframe | Flat shading | Gouraud shading | Phong shading

- Una unidad de operaciones raster.

Como se puede apreciar la Voodoo no implementa por completo el pipeline gráfico, parte del trabajo todavía se realiza en el CPU. A pesar de esto, las mejoras que proponía eran inmensas. Un claro ejemplo es el Need for Speed II SE el cual fue uno de los primeros programas en incorporar soporte para las placas de video Voodoo de 3DFX. Además de incrementar los cuadros por segundo (FPS), se veían mejoras por el uso de Gouraud shading, la incorporación de efectos tales como reflexiones en los autos, manchas en la cámara dejadas por insectos, la existencia de clima, mejoramiento de la niebla, etc.

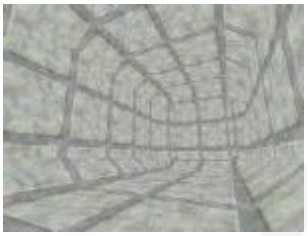
Need for Speed II SE renderizado enteramente por software

Need for Speed II SE renderizado usando la API Glide de 3DFX

1.2.2 TNT y Rage

En 1998, NVIDIA y ATI introducen las GPU TNT y Rage respectivamente. 3DFX en este entonces continuaba liderando el mercado, gracias al popular Voodoo 2. Sin embargo, las GPUs de NVIDIA y ATI fueron las que incluyeron nueva funcionalidad.

Estas incorporan características de multi-texturizado: un pixel puede ser coloreado usando más de una textura sin tener que enviar el triangulo dos veces. Un uso común de esta característica es la técnica de light mapping. La idea es simple, para un objeto no solo se utiliza la textura base del mismo sino también una segunda textura o mapa que contiene información precalculada de la iluminación estática que incide sobre el objeto.

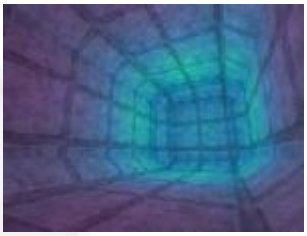


Objeto renderizado solo con su textura base.

X

Objeto renderizado usando solo la textura de iluminación.

=



Resultado final

Otro adelanto importante que solamente tuvieron las GPUs de ATI y NVIDIA es la incorporación de una paleta de colores de 32 bits.

Además, ese mismo año, el ancho de banda entre el CPU y la GPU aumenta considerablemente dado que surge el bus AGP (Accelerated Graphics Port), el reemplazante del bus PCI para conexiones de placas de video.

Una de las principales características del bus AGP es que se encuentra conectado al north-bridge o puente norte de la placa madre, a comparación del bus PCI que se encuentra conectado al south-bridge o puente sur. Esto ubica por primera vez a la GPU en el mismo lugar jerárquico que el CPU y la memoria principal.

Las ventajas del bus AGP incluyen:

- Mayor ancho de banda.
- Una conexión serie, la cual es más barata y escalable.
- Un protocolo punto a punto, con el propósito de que el ancho de banda no sea compartido entre dispositivos.
- Parte de la memoria del sistema se reserva con el propósito de que sirva como memoria de video no local cuando la GPU se queda corta de memoria local.

1.2.3 GeForce, Radeon y Savege3D

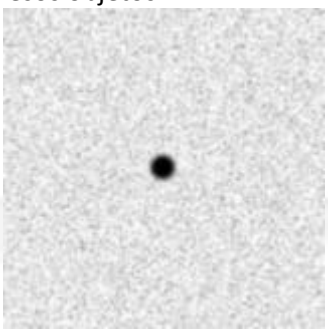
En 1999-2000, con la aparición de la GeForce 256 y GeForce2 de NVIDIA, la Radeon 7500 de ATI y la Savage3D de S3, la etapa de la transformación de vértices se mueve desde el CPU hacia la GPU. Esta nueva incorporación se la llama

unidad de transformación e iluminación, o Transformation and Lighting. La GPU ahora es alimentada con triángulos 3D junto con toda la información para iluminar esos triángulos.

Además, NVIDIA incorpora varias operaciones a nivel de pixel a través de una nueva unidad, llamada register combiner. Con esta unidad podemos, entre otras cosas, realizar phong shading y bump mapping.

Bump mapping es una técnica que consiste en dar un aspecto rugoso a las superficies de los objetos, o en otras palabras, dar un efecto de relieve a esos objetos.

Esfera sin bump mapping



La textura o mapa usada generar el bump mapping

La esfera es geométricamente igual a la primera, pero tiene bump mapping aplicado.

Estas GPUs también soportan nuevos formatos de textura:

- Texturas cubicas se usan para lograr environment mapping.

Environment mapping es un método eficiente de simular reflexiones complejas en una superficie usando texturas precalculadas.

En la carrocera de ambos vehículos se puede ver el poder del Environment mapping

- Texturas proyectivas son usadas justamente para proyectar texturas en la escena con el objetivo de crear sombras y decals.

Los **decals** se pueden ver como texturas que son aplicadas dinámicamente a una superficie. Es normal verlos, por ejemplo, cuando se desea reflejar marcas producidas por impactos de bala o algún tipo de arma.

Decals activados

Decals desactivados

1.3 Pipeline gráfico de función fija

La historia de las GPUs no termina acá. Pero es importante, antes de continuar, ver en qué situación se encontraba el desarrollador de aplicaciones 3D de esa época.

En ese entonces ATI y NVIDIA además de competir por tener la mejor prestación en sus placas de video, competían por la funcionalidad que les brindaban a los desarrolladores de aplicaciones 3D. Ambas empresas incluían funciones que no estaban incluidas en las placas del rival, algo similar a lo que ocurría con Intel y AMD con sus set de instrucciones SSE y 3DNow! Si los desarrolladores querían utilizar todo el potencial de las placas de video de ese entonces debían tener en cuenta las tecnologías de ambas empresas por separado.

Es más, a pesar de que todo el pipeline gráfico se ejecutaba completamente en la GPU, lo cual permitió manejar escenas 3D en tiempo real de relativamente alta complejidad, los desarrolladores estaban limitados a la funcionalidad incluida en las GPUs y no podían hacer nada para evitarlo.

Tanto los vértices como los fragmentos se procesaban siempre de la misma manera. Había muy pocas posibilidades de cambios en el procesamiento de ambos. La unidad de registers combiners, anteriormente nombrada, fue el primer intento de agregar flexibilidad en el procesamiento de los fragmentos. Pero aun así estas posibilidades eran limitadas.

Esta falta de flexibilidad es la que le dio el nombre de **pipeline gráfico de función fija** al pipeline gráfico de las GPUs de ese entonces.

1.4 RenderMan

La necesidad de flexibilidad no es una idea revolucionaria de principios de siglo. Hace ya bastante tiempo que los desarrolladores de gráficos por computadora descubrieron que el proceso de tratar de crear gráficos fotorealistas tiene demasiadas variables que no pueden ser expresadas con un conjunto simple de ecuaciones o representadas por un conjunto finito de estados. De hecho, muchas veces se necesita de la prueba y error, fundamentales cuando se trata de reproducir o simular acontecimientos de la vida real.

Pixar lanzo RenderMan en 1989. RenderMan es un estándar que tiene como propósito especificar un esquema de intercambio de información para permitir la compatibilidad entre un software de modelado 3D y un renderizador. Además de especificar el formato en el cual los datos serán intercambiados, el estándar permite a los desarrolladores especificar cómo debe renderizarse una superficie. Esto se logra a través de un lenguaje simple, basado en el lenguaje C, el cual permite a los desarrolladores tomar los datos entrantes del renderizador y aplicar sus propios algoritmos antes de que sea renderizado. Estos programas son llamados **shaders**.

RenderMan solo define un estándar, pero Pixar también desarrollo en paralelo un software de renderizado basado en el. A través del uso de RenderMan y su propio renderizador, Pixar probó que el uso de shaders puede producir gráficos por computadora impresionantes, y es por eso que RenderMan es usado en una gran cantidad de películas.

RenderMan nunca fue diseñado para ser usado en renderizado de tiempo real, pero sirvió como la base para la futura implementación de shaders en aplicaciones 3D de tiempo real.

Toy Story de Pixar

Monster Inc. de Pixar

1.5 La era del pipeline gráfico programable

En este momento se produce un quiebre tecnológico. Comienza la era del pipeline gráfico programable.

A pesar de que hablaremos de shaders, el tema lo tocaremos superficialmente. Es importante continuar centrados en la evolución general de las GPUs antes de meternos de lleno en el tema principal de este texto.

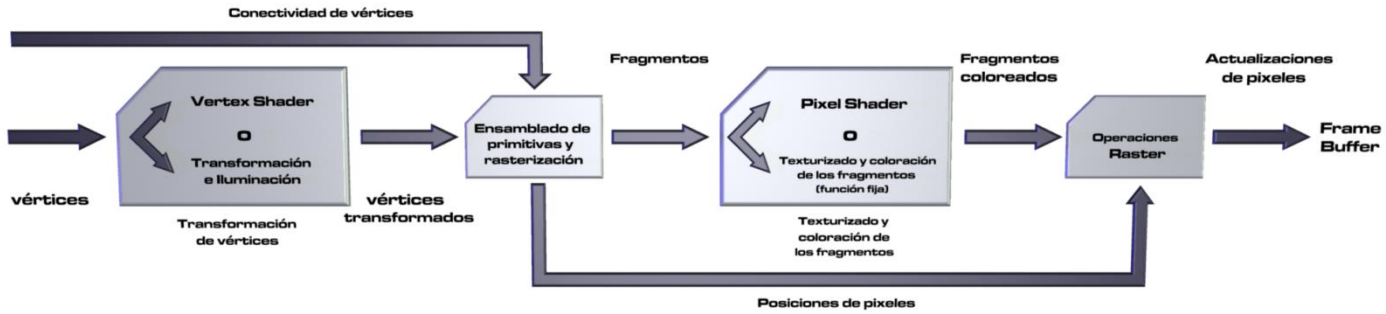
1.5.1 Shader Model 1: GeForce 3, GeForce 4 Ti y Radeon 8500

En el 2001 el procesador GeForce 3 de NVIDIA fue lanzado al mercado. Este incorpora varias novedades, la más importante es el agregado, por primera vez en la historia, de verdaderas etapas programables.

La idea es permitir procesar los vértices y/o fragmentos utilizando una serie de pequeños programas escritos por el desarrollador en vez de usar los algoritmos predefinidos de la GPU. Estos programas son justamente los shaders.

Antes de continuar veamos como repercute esto en nuestro pipeline gráfico:

- Transformación de vértices: a la hora de renderizar un objeto tenemos dos opciones. Podemos seguir usando la unidad de transformación e iluminación o podemos usar un shader programado por nosotros el cual indica como procesar los vértices de ese objeto. Estos shaders los llamaremos **Vertex Shaders**.
- Ensamblado de primitivas y rasterización: sin cambios.
- Texturizado y coloreado de fragmentos: análogamente a la etapa de transformación de vértices, tenemos la opción de elegir usar nuestro propio shader o seguir usando la funcionalidad fija de esta etapa. Estos shaders los llamaremos **Pixel shaders** o **Fragment shaders**.
- Operaciones Raster: sin cambios.



Como se puede apreciar podemos seguir usando el pipeline gráfico enteramente con su funcionalidad fija. Lo que se nos brinda es la posibilidad de reemplazar la funcionalidad fija de una o dos etapas especificas de nuestro pipeline por funcionalidad programada.

En el 2002 la GeForce 4 Ti de NVIDIA y la Radeon 8500 de ATI llegan al mercado mejorando la performance de las etapas programables y agregando nuevas posibilidades a las mismas. Estas GPU, incluida la GeForce 3, soportan la primera generación de shaders, conocida como shader model 1. La GeForce 4 MX no se incluye en esta generación de GPUs dado que sigue como base a la arquitectura de la GeForce 2.

Cada **shader model** específica que restricciones existen en la creación de shaders, como el tamaño máximo de los mismos, las operaciones que se pueden utilizar, la cantidad de registros rápidos presentes, etc.

Need for Speed Underground es uno de los primeros juegos que aprovecha la nueva generación de GPUs. Los desarrolladores optaron por brindar la posibilidad de ejecutarlo con shaders o usando el pipeline gráfico de función fija. A pesar de que los shaders que se podían ejecutar en ese entonces eran limitados, la diferencia era notable. Prestemos atención en la figura siguiente a los reflejos en el chasis del Mazda o a la iluminación proveniente de las luces frontales del mismo.

Need for Speed Underground renderizado usando el pipeline gráfico de function fija

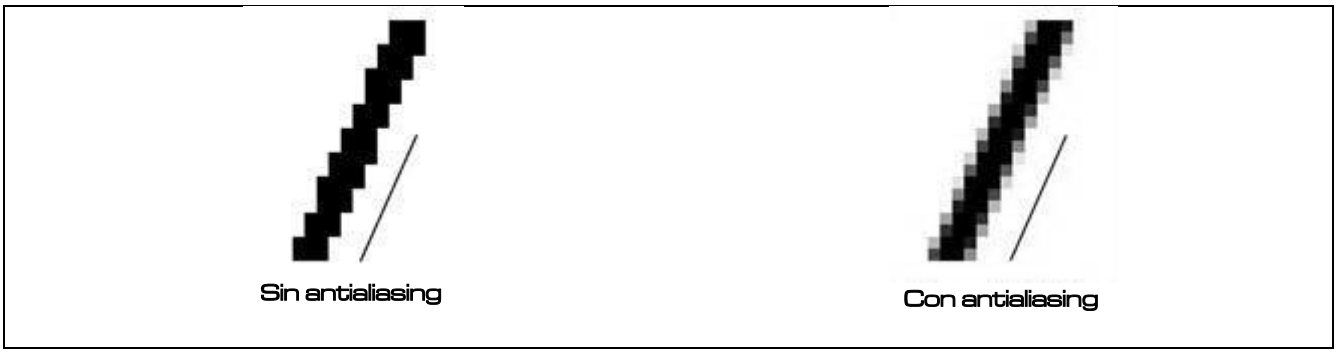
Need for Speed Underground renderizado usando shader model 1

Suspenderemos la explicación de los shaders hasta el próximo capítulo. En el cual definiremos precisamente que es un shader, los tipos de shaders que existen, que son los shader models, entre otras cosas más.

Esta generación de GPUs también incorporo otras novedades además de los shaders. Soportan volume textures, o también llamadas texturas 3D, las cuales agregan una tercera dimensión a las texturas 2D. El uso de volume textures permite entre otras cosas lograr de manera muy realista pelo y pasto.

Varios métodos de antialiasing han surgido desde 1995, pero debido a que eran altamente costosos a nivel computacional su uso se vio limitado. Esta generación introdujo un nuevo método, llamado multisampling, que por primera vez permitió tener antialiasing sin disminuir dramáticamente los cuadros por segundos.

Aliasing es un efecto no deseado que se produce cuando las líneas o curvas que forman los bordes de un objeto se ven discontinuas (en forma de escalera). **Antialiasing** es la técnica que permite suavizar todos los bordes y así disminuir el efecto de escalera.



1.5.2 Shader Model 2: GeForce FX, Radeon serie 9000 y Radeon serie X

Comenzamos a llegar al presente y en una época muy interesante para nosotros dado que la evolución se centro casi por completo en el desarrollo de los shaders. Es más, el desarrollo de juegos con shaders se establece como estándar y por una buena razón. La potencia y capacidades de las GPUs comienzan a permitir desarrollar efectos, iluminación y sombras de altísima calidad, entre otras cosas más.

Este periodo abarca entre 2002 y 2003, e incluye a la Radeon serie 9000 (a partir de la Radeon 9500 y superiores) de ATI y la GeForce FX de NVIDIA. En términos de funcionalidad, esta generación se destaco principalmente por la incorporación del shader model 2.

Infinidad de nuevas cosas se pueden lograr con shader model 2, tal vez una de las más importantes es High Dynamic Range o HDR que es un método de iluminación que permite lograr mayor realismo imitando el funcionamiento del iris en el ojo humano. De esta y otras técnicas hablaremos en los próximos capítulos.

Lost Coast con High Dynamic Range desactivado

Lost Coast con High Dynamic Range activado

En términos del mercado esta generación significo el inicio de una guerra entre NVIDIA y ATI que todavía continua. Es verdad que ambas compañías competían cara a cara desde hace ya un tiempo, pero por primera vez ATI tuvo la oportunidad de ganar cierta porción del mercado que pertenecía a NVIDIA. La razón se debe a que la línea FX, que a pesar de haber tenido un aceptable nivel de ventas, era desastrosa en cuestiones de desempeño en aplicaciones que

hacían uso intensivo de los shaders. La importancia de que ATI haya tomado un rol más significativo en el mercado radica en un simple hecho: a mayor competencia mejores productos y mayor evolución tecnológica.

Como dato anecdótico, el nombre GeForce **FX** proviene del hecho de que surgió como un esfuerzo combinado de los ingenieros adquiridos en la compra de 3D**FX** y de los propios ingenieros de NVIDIA. Para muchos fue ver repetir la historia dado que 3DFX, a pesar de que fue el gigante del mundo de las GPUs por años, termino de desaparecer como compañía atribuido entre otros factores al retraso tecnológico de su última generación de GPUs. Lo cierto es que NVIDIA luego de sacar la GeForce 3 comenzó a desarrollar la GeForce 4 Ti, mientras que ATI puso todo su empeño en desarrollar la Radeon 9700. Luego, NVIDIA centro su atención al desarrollo de la GPU de la X-BOX y para colmo de males tuvo problemas en la elección de la tecnología de manufacturación de los chips de las GPUs FX.

La Radeon serie X de ATI a pesar de competir cara a cara con la GeForce 6, la cual pertenece a nuestra siguiente generación, es una mejora de la serie 9000 de ATI y por ende solo soporta shader model 2. Es por esto que la incluiremos como parte de esta generación.

1.5.3 Shader Model 3: GeForce 6, GeForce 7 y Radeon serie X1000

Esta generación, la cual surge entre 2004 y 2006 incluye a la Radeon serie X1000 de ATI y la GeForce 6 y GeForce 7 de NVIDIA. Análogamente a la generación anterior, estas GPUs se destacaron principalmente por la incorporación de un nuevo shader model, en este caso el tercero.

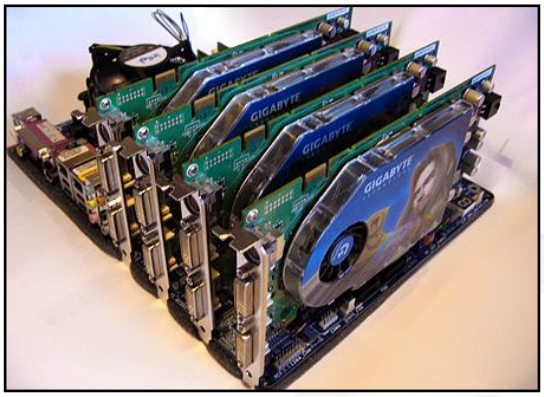
El shader model 3 permitió explorar aun más las posibilidades que brindan los shaders. Tecnologías muy costosas a nivel de cálculos como High Dynamic Range se establecieron en la mayoría de los juegos importantes y se dio paso a una generación de gráficos cada vez más real.

Demo de NVIDIA llamada Nalu, usada para promover a la GeForce 6

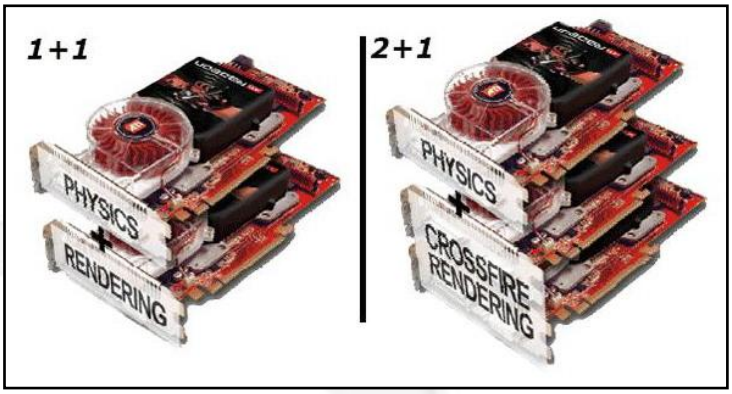
Demo de ATI llamada Ruby: The Assassin, usada para promover a la Radeon serie X1000

Por supuesto que existen otros avances tecnológicos en estas últimas generaciones además de los avances en los shaders. Entre estos adelantos tecnológicos podemos nombrar el uso del bus PCI Express en reemplazo del bus AGP para conexiones de las GPUs a la placa madre, o la posibilidad de trabajar con una paleta de colores de 64 bits.

También podemos destacar el resurgimiento de la tecnología Scan Line Interleave o SLI lanzada en 1998 por 3DFX. SLI es un método que permite conectar dos o más GPUs y en conjunto producir una sola señal de salida. Básicamente es una aplicación de procesamiento paralelo para computación gráfica. NVIDIA llamo a esta tecnología también SLI, pero con otro significado, Scalable Link Interface, debido a que usa una implementación totalmente distinta a la de 3DFX. ATI también tiene su contraparte y la llamo Crossfire.



4 GeForce 7800 GT conectadas en paralelo usando conexión SLI



Sistema Crossfire de ATI en el cual se usa una o dos GPUs para renderizar y una para realizar cálculos físicos.

1.5.4 Shader Model 4: GeForce 8, GeForce 9, Radeon serie HD2000 y serie HD3000

Por fin llegamos al presente y en un momento muy interesante, esperando a que se establezca una nueva generación de GPUs. Al momento de escribir este documento esta generación de GPUs comenzaba a imponerse en el mercado. Solamente la gama alta y media de la línea GeForce 8 y la Radeon HD2900 se encuentran disponibles al público.

Los cambios que traen son muchos. En primera instancia, como es de esperar, incluirá soporte para un nuevo shader model, el cuarto. Este nuevo shader model, además de elevar a un nivel más alto las posibilidades de los shaders, incluirá como cambio significativo una nueva etapa en el pipeline gráfico, los **geometry shaders**. Estos permiten procesar geometrías completas en su conjunto, no solo vértices individuales, y de esta forma poder realizar no solo nuevos efectos, sino procesar en la GPU cosas que se solían hacer en el CPU.

A nivel tecnológico también se ven cambios. Tal vez el más importante es la transición de unidades funcionales separadas para calcular Vertex shaders y Pixel shaders a una colección homogénea de procesadores de punto flotante universales (llamados stream processors) que pueden realizar cualquier tipo de tarea. De forma más sencilla podemos decir que los stream processors pueden calcular Vertex shaders, Pixel Shaders y Geometry shaders según se necesite. Esto permite balancear los recursos y aumentar el desempeño global. Es importante notar que estos procesadores son relativamente simples comparados a las unidades de shaders antiguas debido a su flexibilidad.

Los GPU GeForce 9 y Radeon serie HD3000 están aún en desarrollo, pero se presume que los cambios fundamentales están dados en su arquitectura y su tecnología de fabricación más que en el aspecto funcional.

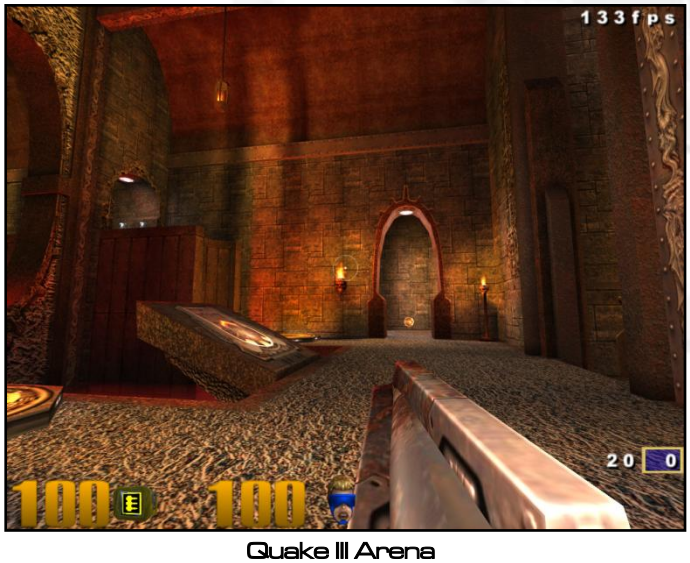
Demo de NVIDIA usada para promover la GeForce 8

Demo de ATI basada nuevamente en Ruby, usada para promover a la Radeon HD2900

1.6 Conclusión

La era del pipeline gráfico programable trajo un nuevo mundo de posibilidades. Muchas veces solo nos basta ver como mejoraron los juegos para entender el poder de los shaders.

Tomemos dos casos extremos. Primero, el Quake III Arena (1999) uno de los últimos FPS (First person shooter) desarrollados usando el pipeline gráfico de función fija, cuyo motor gráfico fue muy popular por unos años. Y comparémoslo con otro FPS, el Crysis (2007), cuyo motor gráfico, el CryEngine 2, hace uso del shader model 4.



Quake III Arena

Crysis

Imaginen, si esto fue solo 8 años de evolución, ¿Qué veremos en los próximos 8 años?

1.7 ¿Qué nos depara el futuro?

Difícil de decir. Obviamente las GPUs serán más potentes y permitirán usar intensivamente la tecnología actual.

Pero como en toda tecnología computacional nueva que alcanzó un cierto grado de madurez a nivel de hardware estimaría que algunos de los próximos adelantos más significativos estarán dados a nivel de software. Mejores herramientas, el surgimiento de nuevas y mejores metodologías de diseño y desarrollo, mejores algoritmos, mayor documentación, etc., etc.

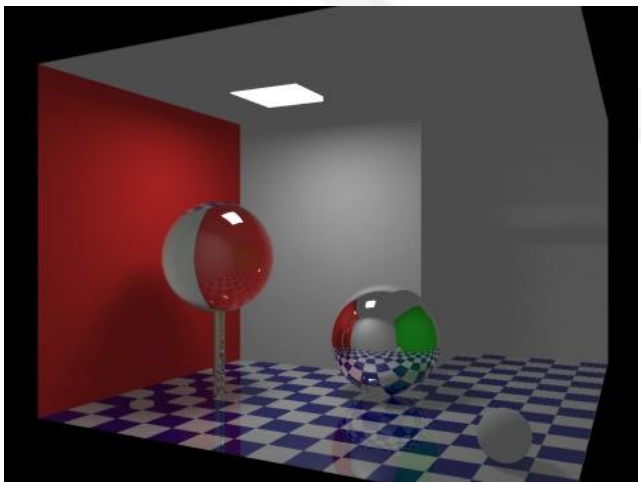
Eso no es todo, muchas veces debemos mirar a su contraparte, los gráficos 3D renderizados sin limitaciones de tiempo para ver en qué dirección iremos. En esta categoría podríamos incluir programas de diseño 3D como Maya, 3D Studio Max, Softimage entre tantos otros. La ventaja de no tener que renderizar un cuadro en un determinado intervalo de tiempo les permite incluir tecnologías costosas a nivel de cálculos. Por esta razón es que en ellos se ven las primeras implementaciones de tecnologías que luego alcanzaran a los gráficos 3D de tiempo real. Tecnologías como los shaders y tantas otras surgieron de esta manera.

Teniendo en cuenta esto, una tecnología que se usa mucho actualmente en los programas de diseño 3D y que sirve para lograr otro nivel más de realismo es la iluminación global. Tal vez el próximo paso importante a lograr.

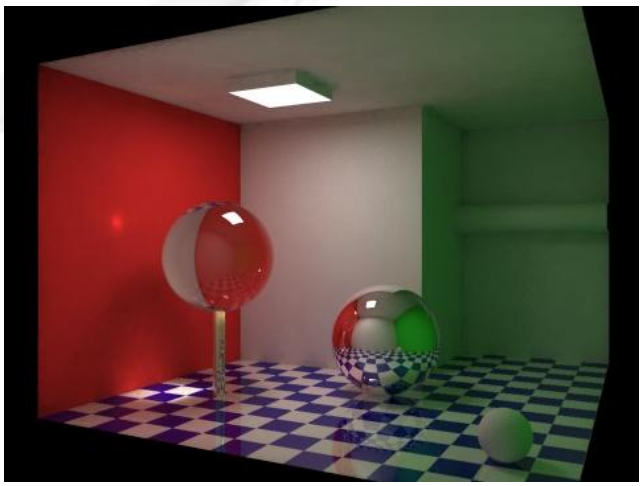
1.7.1 Iluminación global

Los modelos de iluminación que vimos son modelos de iluminación local. Esto significa que la incidencia de la luz en un vértice solo toma en cuenta la relaciones entre el vértice y la fuente de luz, tales como el ángulo de incidencia y la distancia.

Los modelos de iluminación global toman en cuenta no solo la luz que viene directamente desde la fuente de luz (iluminación directa) sino también subsecuentes casos en los cuales haces de luz de la misma fuente son reflejados por otras superficies en la escena (iluminación indirecta).



Sin iluminación global



Con iluminación global

El principal problema de estos modelos de iluminación es que son computacionalmente muy caros. Otro problema que tienen para su implementación en gráficos de tiempo real es que provocarían un cambio radical en el pipeline gráfico que conocemos.

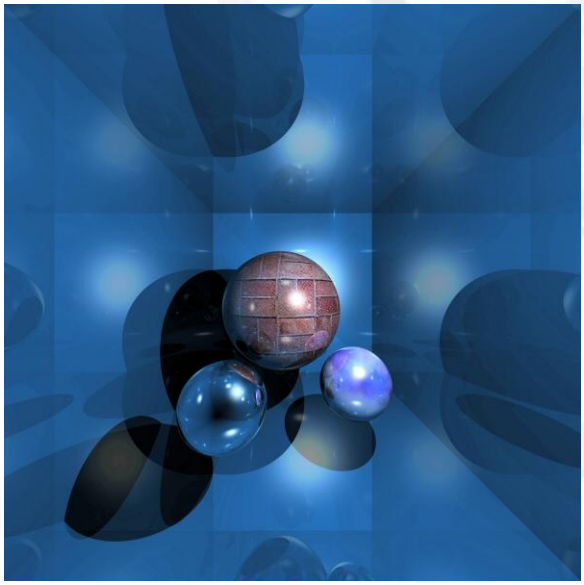
Radiosity, Raytracing, beam tracing, cone tracing, path tracing, ambient occlusion, y photon mapping son ejemplos de algoritmos usados en iluminación global, algunos de los cuales pueden ser usados en conjunto.

Raytracing es una técnica de iluminación global que calcula las trayectorias de los rayos de luz que inciden en la cámara, provenientes de las fuentes de iluminación. Se tienen en cuenta tanto las propiedades de reflexión como las de refracción. Para simular estos efectos de reflexión y refracción se trazan rayos recursivamente desde el punto de intersección del rayo original en el objeto que se está sombreando dependiendo de las características del material del objeto interceptado. Para simular las sombras se lanzan rayos desde el punto de intersección hasta las fuentes de luz.

Es una técnica muy lenta, pero que consigue, en general, resultados muy realistas ya que es capaz de calcular reflexiones, refracciones, transparencias, sombras, etc. Es más, el algoritmo de trazado de rayos es la base de otros algoritmos más complejos.

Dentro del Raytracing, se encuentran básicamente dos submétodos:

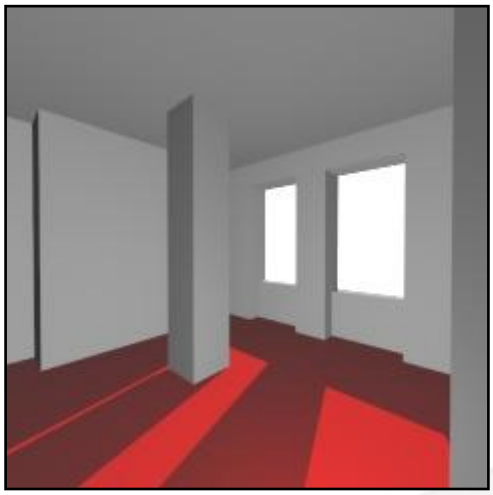
- Photon tracing: se calculan las trayectorias directas, entre las fuentes de iluminación y el observador. Método poco usado.
- Visible tracing: se calculan las trayectorias inversas, entre el observador y las fuentes de iluminación. Es el método más usado. Es el que implementan programas como: RenderMan, Alias, POV y otros.




Ejemplos del poder de Raytracing

La limitación más importante de Raytracing se ve en el sombreado de objetos, que si bien es más realista que con los métodos tradicionales, lo es menos que con técnicas como Radiosity.

Radiosity o radiosidad tiene en cuenta las interreflexiones entre objetos. Se calculan las ecuaciones de radiosidad para cada objeto, en función de la energía que reciben, emiten y propagan. Son métodos independientes del punto de vista del observador, muy lentos, pero que producen los mejores efectos de sombreado e iluminación. Tienen en cuenta las propiedades físicas relacionadas con la reflexión, pero no la refracción.



Escena sin el uso de Radiosity

Radiosity aplicado a una escena

Actualmente un grupo de investigadores logro de manera experimental llevar la técnica de Raytracing a las actuales GPU programables, aunque con alguna que otra incompatibilidad no demasiado grave. Estos investigadores aseguran que las GPUs modernas pueden utilizar esta técnica de renderizado bastante bien. Desde su punto de vista, lo que sucede es que ni ATI ni NVIDIA están interesadas, por el momento, en esta técnica, aunque creen que sólo es cuestión de tiempo que se imponga.

SONY hizo un intento similar cuando uso la fuerza bruta de tres Playstation 3 ejecutándose en paralelo para renderizar usando Raytracing a un Lamborghini Murciélago altísimamente detallado. Desafortunadamente fue necesario utilizar cada uno de los procesadores Cell de las Playstation 3 para poder realizar los cálculos. Cabe aclarar que el procesador Cell es el procesador central de la Playstation 3 y es un procesador extremadamente potente y con buenas prestaciones en cálculos matemáticos de punto flotante. Considerando que estamos hablando de Raytracing, una técnica altamente costosa, el resultado se puede considerar muy satisfactorio.

Lamborghini Murciélago Roadster renderizado usando Raytracing

También existe una demo llamada LightSprint que muestra escenas renderizadas en tiempo real con la técnica de Radiosity. Lo llamativo de esta demo es que corre perfectamente bien en una GPU actual.

1.7.2 Física

Un apartado que está viendo mucha evolución es el de la simulación de física, especialmente usando una PPU o unidad de procesamiento físico. Una PPU es un microprocesador dedicado diseñado para manejar cálculos físicos. Con esta unidad, varios cálculos se pasan de calcular en la CPU a la PPU. Entre estos cálculos podemos nombrar:

- **Rigid body dynamics**: simula el movimiento y equilibrio de sólidos ignorando sus deformaciones. Se entiende por sólido rígido a un conjunto de puntos del espacio que se mueven de tal manera que no se alteran las distancias entre ellos, sea cual sea la fuerza actuante.
- **Soft body dynamics**: se enfoca en simular de forma precisa la física de un objeto flexible. A diferencia de un sólido rígido, el objeto es deformable, lo cual significa que las posiciones relativas de los puntos del objeto pueden cambiar. Como ejemplos de objetos deformables podemos nombrar: ropa, cabello y arena.
- **Fluid dynamics**: simula la física de fluidos.
- **Detección de colisiones**: chequea colisiones, es decir, intersecciones entre dos sólidos.
- **Análisis de Elemento Finito: e**s una tipo de procesamiento matemático especialmente útil para la simulación del comportamiento de variables físicas de un cuerpo. Estas variables físicas son tan diversas como la temperatura punto a punto de un objeto hasta la deformación mecánica provocada por fuerzas ejercidas sobre él.

La idea general es similar a la de usar una GPU. Al usar un procesador especializado se reduce la cargar de la CPU, y de esta manera se permite lograr físicas mucho más realistas. Todavía está por verse si el uso de PPUs será la elección preferida por los desarrolladores y los usuarios. Los usuarios deben sentir que se justifica invertir dinero en una PPU y para ello no solo deben ver las posibilidades que brinda. También deben ver que estas PPUs se utilicen en una gran cantidad de juegos y/o aplicaciones, y a su vez deben sentir que se aprovechan sustancialmente esas posibilidades.

AGEIA PhysX

La PhysX de AGEIA es la primera PPU popular. PhysX también se refiere a la SDK creada por AGEIA para lograr simulaciones físicas. La PPU PhysX está diseñada justamente para acelerar a está SDK.
El 20 de Julio de 2005, Sony firmó un acuerdo con AGEIA para usar la SDK PhysX (en su momento conocida como NovodeX) en la PlayStation 3. Esto provocó que muchos desarrolladores de juegos empezaran a utilizar esta tecnología.
AGEIA afirma que el PhysX es capaz de realizar cálculos físicos cien veces mejor que cualquier PPU creada anteriormente. Aunque esta afirmación pueda parecer algo exagerada, y tal vez lo sea, no se puede discutir la mejora que implica su uso.

Sin AGEIA PhysX PPU

Con AGEIA PhysX PPU

Tom Clancy's Ghost Recon Advanced Warfighter

Havok FX

Havok FX SDK es el competidor más grande que tiene el PhysX SDK. Es usado en más de 150 juegos, incluyendo títulos como Half Life 2 y Company of Heroes.

Para competir con la PPU PhysX, Havok creó Havok FX, el cual permite tomar ventaja de la tecnología multi-GPU de ATI y NVIDIA, Crossfire y SLI respectivamente, utilizando una de las GPU para realizar ciertos cálculos físicos. Básicamente, Havok FX divide la simulación física en física de efectos y jugabilidad. La física de efecto se realiza, si es posible, en la GPU usando el shader model 3. Mientras que la física de jugabilidad se sigue realizando en el CPU.

La distinción más importante entre ambas es que la física de efectos no afecta la jugabilidad. Por ejemplo, polvo o pequeños escombros producidos por una explosión. La mayoría de las operaciones físicas se siguen realizando en software. Este enfoque difiere significativamente del enfoque de PhysX SDK, el cual mueve todos los cálculos físicos a la PPU PhysX, si es que se encuentra presente.

Ahora bien, nos podría surgir la siguiente pregunta: ¿Por qué Havok FX no utiliza a estas GPU para realizar todos los cálculos físicos? La respuesta es que no se puede, dado que los shaders no pueden retornar valores a la aplicación y de esta manera no pueden alterar la jugabilidad. La única forma de retornar valores seria a través de una renderización a una textura, algo que resultaría demasiado rebuscado e ineficiente.

ATI CTM y NVIDIA CUDA

Las últimas GPUs de ATI y NVIDIA pueden usarse como PPUs o para otros tipos de cálculos de propósito general. La idea es usar la nueva arquitectura de stream processors para realizar estos cálculos y obviamente devolver los resultados.

NVIDIA provee un SDK llamado CUDA (Compute Unified Device Architecture), mientras que ATI provee un SDK similar denominado CTM (Close To Metal).

Dual y Quad core

Estos últimos años se ha visto un incremento cada vez mayor de CPUs con tecnología dual core. INTEL y AMD han apostado a CPUs con más de un core como tecnología dominante, y lo han logrado. Es más, AMD, por ejemplo, recientemente acaba de retirar de su gama baja los CPUs single core.

Esto marca un simple hecho, prácticamente todos los usuarios ya tiene tecnología dual core y hasta en algunos casos quad core. Esto permite a los desarrolladores asumir que se puede contar con por lo menos otro core para realizar cálculos. ¿Por qué no usar este core para cálculos físicos? Está bien podría ser la solución a la que se vuelque el mercado, dado que es la más económica (no requiere inversión extra por parte del usuario) y también porque los desarrolladores pueden estar tranquilos que todos dispondrán de esta tecnología. Obviamente, un core no puede alcanzar el poder de cálculo de una PPU para este tipo de tareas.

Como dijimos, está por verse si el uso de PPUs será el futuro de la simulación física. Lo que es indudable es que se está invirtiendo y desarrollando mucho para que esto suceda.

CAPITULO II
SHADERS

The Elder Scrolls IV: Oblivion

2 Introducción

Es tiempo de enfocarnos en el tema principal de este documento. Gracias al capítulo anterior aprendimos que es un shader y cuáles fueron las razones y acontecimientos que dieron inicio a la implementación de esta tecnología en los gráficos 3D de tiempo real. También comenzamos a entender su importancia, y el porqué se convirtió en la tecnología de más avance de estos últimos años.

A diferencia del capítulo anterior, este capítulo se centrará en definir estos temas, permitiéndole al lector no solo entender los distintos matices de esta tecnología, sino también terminar de interpretar conceptos definidos vagamente. En este capítulo definiremos que es un shader, los tipos de shaders y sus posibilidades, y también hablaremos de los distintos shader models y sus características.

2.1 Shaders

> Un **shader** es un conjunto de instrucciones que determinan las propiedades de la superficie de un objeto o imagen. Comúnmente se utilizan para determinar el material que tiene un objeto, aunque también son utilizados para lograr efectos muy realistas como reflejos, transparencias, transformaciones geométricas y una inimaginable cantidad de cosas más.

Como dijimos anteriormente, la idea es simplemente permitir procesar los vértices y/o fragmentos utilizando una serie de pequeños programas escritos por el desarrollador en vez de usar los algoritmos predefinidos de la GPU. Esto, aunque en primera medida podría parecer una pequeña evolución, nos brinda en realidad una gran flexibilidad y la posibilidad de hacer cosas antes imposibles.

En definitiva, debemos tener en claro que:

Los shaders nos dan control sobre el proceso de renderizado

Este es el punto clave de esta tecnología.

2.2 Tipos de shaders

Existen tres tipos de shaders: vertex shader, pixel shader y actualmente geometry shader. A continuación veremos cada uno de ellos, en especial el geometry shader, dado que se describió muy poco del mismo. La razón principal por la que casi no se hablo de él, es porque altera un poco al pipeline gráfico y es el que menos naturalmente se puede esperar como una futura evolución.

2.2.1 Vertex shaders

Los **vertex shaders** se aplican a cada vértice y efectúan operaciones matemáticas sobre los datos de un vértice de un objeto. Cada vértice se define mediante un número de variables, como mínimo se necesita su posición, pero también pueden contener información del color, canal alpha, normal y/o características de la textura e iluminación.

Un vertex shader no genera nuevos vértices, ni elimina existentes, simplemente los procesa. Espera como mínimo la posición del vértice y genera como mínimo la posición transformada a espacio homogéneo (luego de la proyección). El resto de la información que puede contener un vértice, como son las normales, tangentes, etc., también puede ser recibida y devuelta por el vertex shader.

Entre otras cosas, con los vertex shaders se pueden deformar las mallas para lograr ciertos efectos específicos, como los que tienen que ver con la deformación en tiempo real de un objeto; por ejemplo, el movimiento de una ola. Sin embargo, normalmente el vertex shader lo utilizamos para procesar de cierta forma la información que necesita el pixel shader.

2.2.2 Pixel shaders

Un **pixel shaders,** o "**fragment shaders**" en la nomenclatura de OpenGL, básicamente procesa un fragmento y devuelve el color de una posible actualización para un píxel dado.

No existen parámetros mínimos para un pixel shader, pero se espera que devuelva un variable de tipo float4, un vector de cuatro componentes de punto flotante que especifica cada canal de color. También existe la posibilidad de salir sin retornar nada indicando que el fragmento no genera ninguna actualización de color para su pixel asociado.

Este tratamiento individual de los pixeles permite realizar, entre otras cosas, iluminación por pixel, utilizar esquemas complejos de iluminación, y generar efectos muy interesantes.

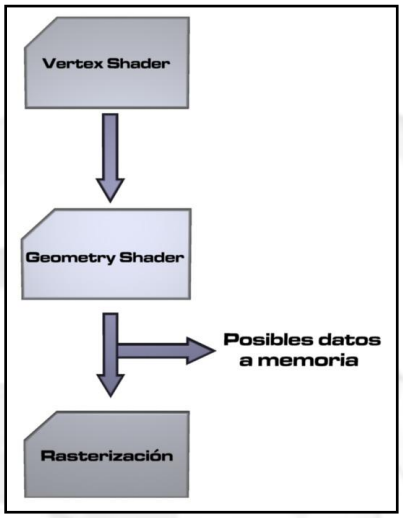
2.2.3 Geometry shaders

El vertex shader toma un solo vértice como entrada y devuelve un solo vértice procesado como la salida. En cambio, el geometry shader permite operar sobre primitivas geométricas enteras, esto es triángulos, lados y vértices, y a su vez permite operar sobre las primitivas vecinas. Esto no quiere decir que el geometry sea un reemplazo del vertex shader, en realidad es un complemento.

Además, el Geometry Shader puede crear nuevas primitivas. Incluso es posible que el Geometry Shader envíe los resultados de nuevo a memoria, permitiendo que los datos vuelvan al inicio del pipeline sin tener que pasar por la CPU. Con esto se acelera muchísimo, por ejemplo, los sistemas de partículas como humo o explosiones que normalmente cargan mucho la CPU, siendo de esta forma independiente este efecto de la CPU.

Se puede usar los geometry shader en combinación con arreglos de texturas para acelerar efectos como cube mapping. Cube mapping simula reflexiones utilizando texturas que reflejan el mundo que hay alrededor del objeto. Para lograr esto se debe determinar qué es lo que rodea a este objeto y mapearlo encima. Normalmente, esta tarea lleva hasta seis pasadas, pero el geometry shader junto con un arreglo de texturas puede lograr el mismo resultado en una sola pasada. Es decir, estamos reduciendo considerablemente el tiempo que tardamos en procesar este efecto tan común. Efecto muy utilizado en agua, objetos metálicos, cristal, etc.

Como dijimos el geometry shader altera un poco el pipeline gráfico. El siguiente es un diagrama simplificado que se enfoca en mostrar los cambios que implica la incorporación de esta unidad.



En conclusión, el geometry shader nos permite realizar de manera más eficiente muchos efectos, dado que además de ejecutar más cálculos sobre la GPU, también reducimos la comunicación innecesaria entre el GPU y la CPU. Lo cual es interesante, debido a que nos muestra una realidad que muchos desconocen. A nivel posibilidades, el shader model 4 no nos permite hacer más cosas que el shader model 3, pero si nos permite hacerlas de manera más eficiente. No solo por la potencia extra de las GPU, o por el aumento del límite del tamaño de los shaders, por nombrar algo. Sino también por el geometry shader y la arquitectura mejorada de estas GPUs (virtualización de la memoria, stream processors, etc.) Todo lo cual, irónicamente, se traduce en más y mejores efectos.

Age of Conan renderizado usando DirectX 9 (shader model 3)

Age of Conan renderizado usando DirectX 10 (shader model 4)

2.3 ¿Qué podemos lograr?

Dado que ahora tenemos el control sobre el proceso de renderizado, podemos lograr una infinidad de cosas. Nombrar todas las posibilidades que disponemos sería difícil, hasta imposible porque siempre puede surgir una nueva técnica.

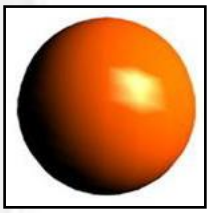
Por eso, a continuación se nombraran solo algunas de las posibilidades que tenemos. El objetivo es brindar una idea de algunas de las principales técnicas existentes y no dar una lista exhaustiva.

2.3.1 Iluminación por pixel

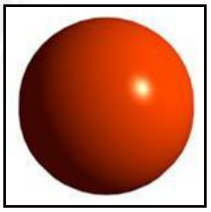
En el capitulo anterior vimos que originalmente la iluminación se calculaba por vértice, que si bien es un método rápido de calcular, los resultados obtenidos no eran perfectos y realistas. Con la aparición de los shaders se puede lograr iluminación por pixel, lo cual nos posibilita disponer de una iluminación más realista.

Antes de continuar, recordemos estos modelos de sombreado:

Gouraud (iluminación por vértice): aplica el patrón de iluminación a cada vértice del polígono y entonces promedia los valores de color en toda la superficie del objeto para así alcanzar un efecto de sombreado suave sobre superficies curvas.



Phong (iluminación por pixel): es el modelo más realista, pero también el más costoso de calcular. Se utiliza una interpolación bilineal para calcular la normal de cada punto (o pixel en la práctica) del polígono. A cada punto del polígono se le asocia una normal distinta con lo cual su color calculado será muy cercano a la realidad.



En definitiva, la iluminación por pixel consiste en realizar gran parte de los cálculos de iluminación en la unidad de pixel shader, en vez de en la unidad de vertex shader. Dado que la unidad de pixel shader nos permite trabajar con ciertos datos de entrada interpolados para cada pixel. En otras palabras, con iluminación por vértice, el color se determina por cada vértice y luego se interpola. En cambio, con iluminación por pixel, se interpola cada componente necesario y luego calculamos la iluminación por cada pixel.

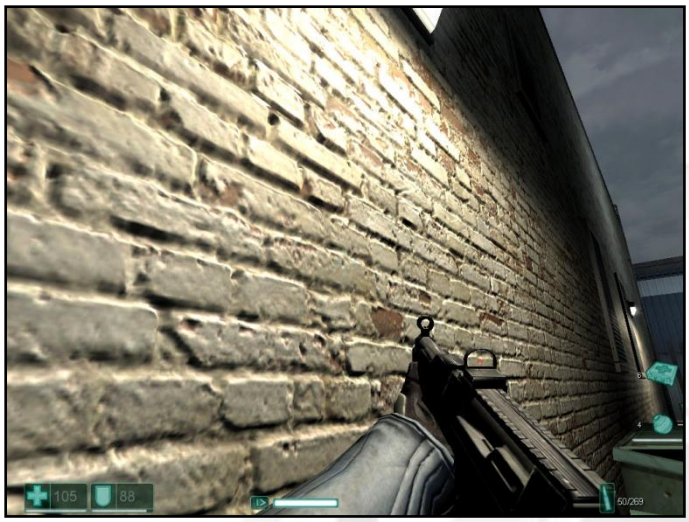
Tengamos en cuenta que la interpolación de cualquier valor es realizada por el hardware de manera transparente. La interpolación es un proceso mecánico y sin misterios, y es por eso que no tenemos control sobre el mismo.

Lo importante aquí, con este tema, es que los shaders no solo permiten incorporar un modelo de sombreado phong, sino trabajar con cualquier tipo de iluminación por pixel que queramos. Por supuesto que podemos trabajar con iluminación por vértice, pero este tipo de iluminación ya casi no tiene sentido en la actualidad.

Entre los tipos de iluminación por pixel que podemos utilizar se encuentra la técnica llamada parallax mapping.

2.3.2 Parallax Mapping

Parallax Mapping es una mejora a la técnica de bump mapping o normal mapping, introducida en el 2001 por Tomomichi Kaneko. Esta mejora se traduce en una mayor sensación de profundidad y un mayor realismo, sin afectar excesivamente al rendimiento del sistema.

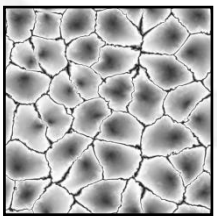



Imágenes del F. E. A. R.
En ambas imágenes las paredes son un simple plano.
La sensación de profundidad se logra usando Parallax Mapping

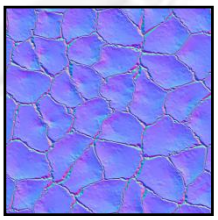
F.E.A.R. hace uso de Parallax Mapping. Desafortunadamente en las capturas anteriores no se llega apreciar completamente la mejora en realismo que conlleva su uso. Se aprecia mucho mejor en movimiento, dado que con Parallax Mapping se genera una sensación de profundidad que hace pensar que existe geometría donde no la hay.

Parallax mapping se implementa desplazando en cada punto del polígono las coordenadas de texturas. Este desplazamiento está basado en una función que se calcula teniendo en cuenta el ángulo de visión en espacio tangente (el ángulo relativo a la normal de la superficie) y el valor del mapa de altura en ese punto. En otras palabras, el color final del pixel se logra usando tres texturas:

Textura base.



Mapa de altura: da la altura de cada texel de la textura.



Mapa de normales: da información sobre las normales de cada texel de la textura.

2.3.3 High Dynamic Range

En la práctica, a High Dynamic Range (o HDR) la podemos ver como una técnica que imita el funcionamiento del iris en el ojo humano. Si nos encontramos en un pasillo muy oscuro, el iris se abre para permitir la entrada de una mayor cantidad de rayos lumínicos dándonos la sensación de que este pasillo se "ilumina" ligeramente. Si nos encontramos en una zona ampliamente iluminada, el iris se cierra, para evitar la entrada masiva de rayos lumínicos (y evitar daños

en la retina), dando una sensación general de que la zona se "oscurece" ligeramente. Si al final de un pasillo ponemos un objeto, y entre éste objeto y nosotros, ponemos un potente "punto de luz", sucede que el objeto "desaparece" detrás de los rayos de luz, y las zonas oscuras que antes se hallaban mas "iluminadas" debido a la apertura del iris, vuelven a oscurecerse.

Básicamente, el iris se adapta a la "luminosidad general" de la zona. Como hay más zonas muy iluminadas que zonas oscuras, el ojo se adapta a las primeras. Si el punto de luz desapareciera, predominarían las zonas oscuras, y nuestra visión se adaptaría a ellas.

HDR desactivado HDR activado

De forma más precisa, High Dynamic Range es la ciencia de reconocer los distintos niveles de intensidad de la luz. Bajo condiciones de renderizado normales, cuando el nivel de iluminación promedio es similar en toda la escena, la precisión de color que solemos usar es suficiente para representar los diferentes colores e intensidades de la luz. Sin embargo, en la vida real existen muchas situaciones donde los niveles de iluminación varían significativamente en la escena, lo que lleva a un escenario donde no hay precisión suficiente para representar todos los fenómenos que suceden de estas diferencias de intensidad.

Para lograr un renderizado de tipo High Dynamic Range necesitamos aplicar dos técnicas:

Tone mapping

Tone mapping es una técnica usada para mapear colores desde un rango dinámico alto (en el cual se realizan los cálculos de iluminación) a un rango dinámico bajo que corresponde con las capacidades del dispositivo en donde se renderizará. Típicamente, el mapeo no es lineal, se preserva un rango suficiente para los colores oscuros y gradualmente se limita el rango dinámico para colores más brillantes. Normalmente esta técnica produce imágenes visualmente atractivas, con buen detalle y contraste.

Las seis muestras usadas para crear la imagen previa

Light bloom

Light bloom es una técnica que trata de imitar un efecto causado por la manera en que trabajan el ojo humano y las cámaras. El exceso de energía lumínica no solo afecta a un punto particular en el receptor sino también a puntos vecinos. Esto produce un efecto tipo glow (resplandor) que se suele llamar blooming.

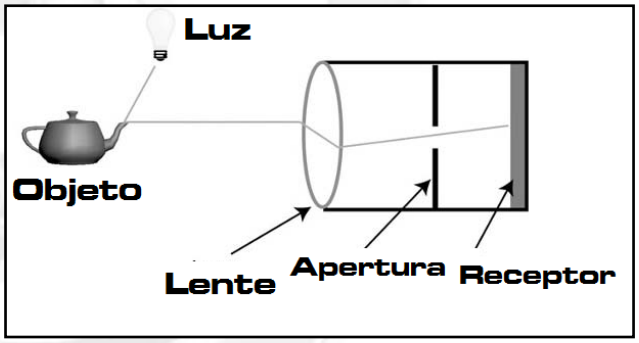
2.3.4 Depth of field

Depth of field, o profundidad de campo, es otro efecto que ocurre por la forma en que trabaja el ojo humano y los equipos de fotografía. Este efecto es el que normalmente llamamos “fuera de foco”. Por ejemplo, miremos fijamente a la distancia y mantengamos un dedo cerca de nuestro ojo, nuestro dedo se verá borroso.

Imágenes de Heavy Rain. Se puede apreciar como el foco cambia de la persona al arma

La razón de este efecto se puede entender si conocemos como funciona una cámara. Lo dicho también vale para el ojo humano dado que su funcionamiento general es idéntico.

Si miramos la figura, lo primero que notamos es que la cámara captura la imagen del objeto capturando la luz que es reflejada por el objeto. La luz rebota en el objeto en todas las direcciones y eventualmente alcanza al lente de la cámara.
El propósito del lente es controlar el zoom del objeto, mientras se asegura que la luz que llega a la cámara converja hacia la apertura. La apertura sirve como una pequeña entrada que controla la cantidad de luz que llega al receptor. El receptor recibe la energía de la luz y la convierte en colores.



En teoría, si la apertura fuera infinitamente pequeña, solo la luz que converge perfectamente a través de la apertura podrá pasar al receptor, y la imagen será nítida para todas las distancias. Sin embargo, en tales casos, esto también significa que muy poca energía lumínica llegara al receptor. Las cámaras no tienen una apertura infinitamente pequeña debido a que se necesita una cierta cantidad de luz para capturar una imagen.

Desafortunadamente, esto tiene un efecto secundario, y es el de permitir que algunos haces de luz que no convergen perfectamente en el centro de la apertura alcancen al receptor. Estos haces indeseados son los que causan las distorsiones.

Ahora nos podemos preguntar la razón por la que existen haces de luz que no convergen perfectamente en la apertura. Para entenderlo consideremos lo siguiente, si miramos a un objeto localizado cerca de la cámara, notaremos que algunos haces de luz que rebotan en el objeto pero que no se dirigen directamente a la cámara, pueden entrar en el lente. El lente está diseñado para refractar la luz que se dirige perpendicularmente hacia él, con el propósito de que converjan a la apertura. Sin embargo, con el objeto cerca del lente, un haz no perpendicular

puede entrar al lente. Este haz de luz es refractado como cualquier otro, pero debido a su dirección, este no converge exactamente al centro de la apertura.

Si la apertura fuera realmente pequeña, estos haces serian bloqueados, pero en la realidad este no es el caso. Es por eso que los objetos cercanos a la cámara se verán borrosos y fuera de foco. Esto se debe a que la luz que llega de direcciones incorrectas también contribuirán a la imagen, aun si estos haces no fueron enfocados apropiadamente. Un fenómeno similar ocurre con los objetos que se encuentran muy lejos de la cámara.

2.3.5 Motion blur

Si movemos nuestra mano de izquierda a derecha rápidamente en frente de nuestros ojos, notaremos que no podremos ver nuestra mano claramente. El mismo fenómeno ocurre con las cámaras. Esto se debe a que el ojo y las cámaras no toman información a una tasa infinita, sino que lo hacen una cierta cantidad de veces por segundo.

El enfoque ideal para simular motion blur es determinar la velocidad de cada pixel de un cuadro a otro, y usar esta información para calcular este fenómeno. Otro enfoque ignoraría la velocidad del movimiento y tomaría los pixeles de cuadros anteriores y los mezclaría con el pixel del cuadro actual. Aunque rápido de calcular, no toma en cuenta la velocidad real del pixel y es por eso que no calcula muy bien este fenómeno. Aun así fue un enfoque muy utilizado en la industria.

2.3.6 Sombras dinámicas y soft shadows

Si no usamos técnicas de iluminación global como Raytracing, es casi imposible determinar de manera precisa las sombras dinámicas. Afortunadamente, podemos simularlas bastante bien utilizando técnicas como shadow mapping y shadow volume.

Shadow mapping

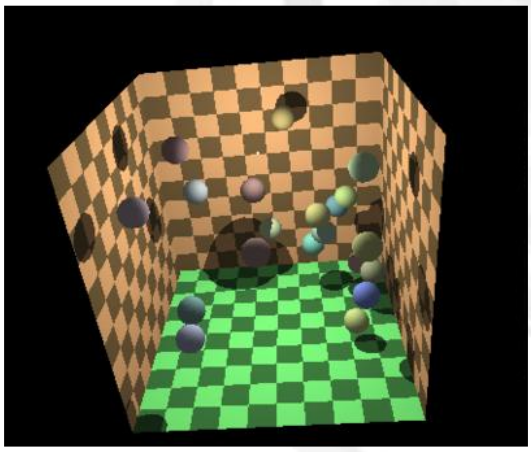
Si viéramos la escena desde el punto de vista de la luz, las regiones bajo sombra corresponden a la porción de la escena que se encuentra detrás de los objetos que ocluyen la luz. En términos de renderizado, cualquier objeto cuya profundidad sea mayor que el objeto que ocluye la luz desde el punto de vista de luz estará bajo sombra.

La técnica de Shadow mapping toma ventaja de este hecho. Si conocemos la profundidad de los objetos que ocluyen la luz desde el punto de vista de la luz, podríamos usar esta información para determinar si existen pixeles que se encuentran bajo sombra.

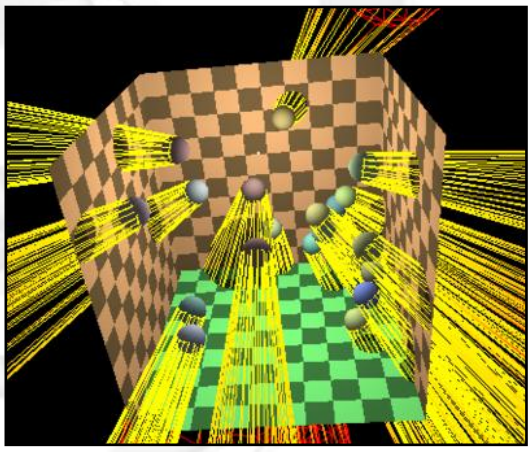
A nivel implementación, podríamos dividir esta técnica en dos partes. Primero necesitamos crear el mapa de sombras (o shadow map) lo cual se logra renderizando a una textura la escena desde el punto de vista de la luz. Segundo, necesitamos aplicar este mapa en la escena.

Shadow volume

Básicamente, se crea un volumen que encierra cada punto que posiblemente podría estar bajo sombra por un objeto. En otras palabras, se necesita encontrar la silueta del objeto, la cual separa la mitad que es iluminada por la luz y la mitad que no lo es, y extender la silueta en la dirección contraria a la fuente de luz. Finalmente, la porción de geometría que intercepta al volumen se encontraría bajo sombra.



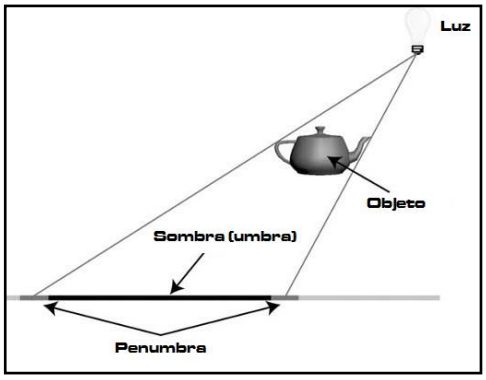
Escena bajo sombra



Shadow volume

Estos esquemas son muy utilizados y simulan muy bien las sombras. Sin embargo, consideran a las luces como luces point perfectas y no crean las regiones de penumbra.

Las sombras tienen dos partes: umbra y penumbra. Umbra es la porción de la sombra que está cubierta completamente, mientras que la penumbra es la porción de la sombra que está parcialmente ocluida.
La penumbra existe porque las luces en la realidad no son luces point. La luz no es emitida desde un solo punto, sino que es emitida desde un área o sector, lo cual causa que la región del perímetro donde la luz no está perfectamente ocluida se vea parcialmente en sombra.



Las sombras tradicionales son llamadas **hard shadows**, mientras que las sombras que consideran la penumbra se las denomina **soft shadows**. La técnica de shadow mapping puede modificarse para producir soft shadows. Esto se logra mesclando varios texels adyacentes del shadow map, según un sistema de pesos.

Hard shadow

Soft shadow

2.3.7 Renderizado no foto realista

Normalmente se busca lograr el mayor realismo, técnicas como HDR bien utilizadas dan una sensación de realismo increíble, imposible de lograr sin los shaders. Muchas otras veces, sin embargo, resulta necesario lograr el efecto contrario. Por ejemplo, cuando se trata de simular el estilo toon o cel shading, estilo que vemos en series como Los Simpson o en películas como Monster House.

XIII

X-Men Legends II: Rise of Apocalypse

Con los shaders podemos lograr la iluminación característica de los materiales toon, o lograr el contorno negro de los objetos. El contorno se usa para destacar a los objetos, dado que la paleta de colores generalmente es reducida y podría costar distinguir un objeto del objeto adyacente. Existen varias técnicas para lograr esto último, cada una con sus ventajas y desventajas.

2.4 Shader models

Un **shader model** es un estándar que define la funcionalidad y restricciones a la que se ajustara el shader a ejecutar. Cada shader model se construye sobre la funcionalidad del modelo anterior a él, implementando más funcionalidad con menos restricciones. Esto implica que existe retro-compatibilidad, por ejemplo, si una GPU soporta shader model 3.0 también soporta shader model 2.0.

Las diferencias entre los shader models tienen que ver, principalmente, con cuestiones internas de programación. Parámetros como la cantidad de registros disponibles, el número de instrucciones permitido por programa y la incorporación de instrucciones aritméticas más complejas, entre otros, son los que diferencian una versión de shader model de otra.

A continuación veremos las diferencias más importantes entre los distintos shader models, sin considerar el primer shader model. La razón radica en que el primer shader model consiste de varios sub modelos, y en la actualidad ya no se tiene en cuenta. Por último vale aclarar que algunos términos se dejaran en ingles para evitar confusiones.

Vertex shaders	VS_2_0	VS_2_a	VS_3_0	VS_4_0
Número máximo de instrucciones por shader	256	256	≥ 512	4096
Número máximo de instrucciones a ejecutar por shader	65536	65536	65536	65536
Branch predication	No	Si	Si	Si
Registros temporales	12	13	32	4096
Número de registros constantes	≥ 256	≥ 256	≥ 256	16x4096
Control de flujo estático	Si	Si	Si	Si
Control de flujo dinámico	No	Si	Si	Si
Profundidad del control de flujo dinámico	No	24	24	Si
Soporte para geometry instancing	No	No	Si	Si
Vertex Texture Fetch	No	No	Si	Si
Número de texture samplers	No	No	4	128

Donde:

- VS_2_0: Especificación original del Shader Model 2.
- VS_2_a: Modelo optimizado de NVIDIA para la GeForce FX.
- VS_3_0: Shader Model 3.
- VS_4_0: Shader Model 4.

Geometry instancing se refiere a la práctica de renderizar múltiples copias de un mismo objeto poligonal en la escena al mismo tiempo.

Pixel shaders	PS_2_0	PS_2_a	PS_2_b	PS_3_0	PS_4_0
Número máximo de instrucciones por shader	32 + 64	512	512	≥ 512	≥ 65536
Número máximo de instrucciones a ejecutar por shader	32 + 64	512	512	65536	Sin limite
Branch predication	No	Si	No	Si	Si
Registros temporales	12	22	32	32	4096
Número de registros constantes	32	32	32	224	16x4096

Control de flujo dinámico	No	No	No	24	Si
Dependent texture limit	4	Sin limite	4	Sin limite	Sin limite
Texture instruction limit	32	Sin limite	Sin limite	Sin limite	Sin limite
Position register	No	No	No	Si	Si
Indirecciones de texturas	4	Sin limite	4	Sin limite	Sin limite
Interpolated registers	2 + 8	2 + 8	2 + 8	10	32
Index input registers	No	No	No	Si	Si
Arbitrary swizzling	No	Si	No	Si	Si
Gradient instructions	No	Si	No	Si	Si
Loop count register	No	No	No	Si	Si
Face register (2-sided lighting)	No	No	No	Si	Si

Donde:

- PS_2_0: Especificación original del Shader Model 2.
- PS_2_a: Modelo optimizado de NVIDIA para la GeForce FX.
- PS_2_b: Modelo de ATI para la Radeon X700, X800, X850.
- PS_3_0: Shader Model 3.
- PS_4_0: Shader Model 4.

La existencia de un estándar es importantísimo para los desarrolladores, dado que no deben tener en cuenta implementaciones distintas realizadas por distintos fabricantes. Desafortunadamente, en la práctica todavía se tiene que tener en cuenta al fabricante por separado, o, lo que es lo mismo, a los productos de ATI y NVIDIA. Mayoritariamente, esto consiste en tener en cuenta a ambas compañías y/o familias de GPUs al momento de realizar las optimizaciones de los shaders más que en la codificación de ellos. La optimización es una etapa de mucha importancia en el desarrollo de un producto. Etapa que muchas veces, por cuestiones de tiempo, presupuesto y/o ignorancia se reduce, y produce que muchos títulos con potencial se ejecuten con un desempeño desastroso.

2.5 Implementación e integración de los shaders

Ahora que tenemos una idea más precisa de que es un shader y que se puede lograr con el mismo, comenzaremos a ver detalles de cómo se implementan y se integran a nuestro motor.

Al momento de renderizar un objeto debemos indicar en nuestro motor que utilizaremos shaders para renderizarlo. Todas estas indicaciones se realizan utilizando nuestra API gráfica, ya sea OpenGL o DirectX. Nuestra API gráfica le suministrara a la GPU el código del shader en un lenguaje maquina especial, el cual es neutral con respecto al hardware y será el mismo no importa donde se compile o ejecute.

Análogamente a la programación tradicional, o desde cierto punto de vista la programación sobre el CPU, los programas a ejecutar, en nuestro caso los shaders, se escriben en algún lenguaje de alto nivel. Lenguajes que tienen cierta similitud con los lenguajes de programación tradicionales, con ciertas diferencias enfocadas primordialmente por la forma en que trabajan las GPUs. También, al igual que la programación tradicional, se comenzó trabajando con un lenguaje ensamblador antes de llegar a los lenguajes de alto nivel.

En definitiva, los shaders se toman desde un archivo, se compilan por la API, y se suministran a la GPU utilizando el lenguaje maquina de ellas. En el próximo capítulo trataremos el tema de los lenguajes a fondo. Por el momento, se seguirá explicando la implementación e integración de los shaders de manera más superficial y general.

Además, existen herramientas que nos permiten desarrollar shaders a un nivel aun más alto. En la práctica, estas herramientas resultan vitales para el desarrollador, dado que no solo nos permiten escribir los shaders de manera más cómoda, sino que también nos permiten, en otras cosas, optimizarlos sin tener que usar nuestro motor gráfico, y de esta manera abstraernos del mismo. Es más, estas herramientas no nos quitan poder, y tampoco producen shaders menos eficientes. En el cuarto capítulo hablaremos más sobre estas herramientas.

2.6 Un ejemplo de cómo integrar un shader a nuestra aplicación

A continuación mostraremos un ejemplo de cómo integrar un shader escrito en HLSL en un motor escrito en C# utilizando como API gráfica a DirectX 9. El objetivo es mostrar, en una situación real, como se especifica y como afecta el uso de shaders al código de nuestro motor.

La razón de la elección de HLSL, C# y DirectX 9 se debe más a una cuestión de comodidad propia, más que a cualquier otro factor. Tranquilamente podríamos ejemplificar esta situación con OpenGL, C++ o Cg para mostrar la integración de shaders a nuestro motor.

Veamos el ejemplo. Este código renderiza un objeto usando un shader determinado.

```
public void OnRender ()
{
    .
    .
    .
    // Cargamos el archivo fx
    Effect effect = Effect.FromFile(device, "codigoshader.fx", null, ShaderFlags.None, null);

    // Elegimos la técnica
    effect.Technique = "ShaderTechnique";

    // Effect.Begin retorna el número de pasadas requeridas para renderizar el shader.
    int passes = effect.Begin(0);

    // Ciclamos a través de todas las pasadas del shader.
    for (int i = 0; i < passes; i++)
    {
        // Seteamos las constantes globales del shader.
        effect.SetValue("WorldMatrix", worldMatrix);

        // Indicamos que comenzaremos la pasada.
        effect.BeginPass(i);

        // Renderizamos el objeto
        device.DrawPrimitives(PrimitiveType.TriangleList, 0, 1);

        // Indicamos que terminamos la pasada.
        effect.EndPass();
    }

    // Indicamos que terminamos
    effect.End();
    .
    .
    .
}
```

Primero se le indica que cargue el shader desde un archivo, en este caso particular codigos*hader.fx*. Esta operación se suele hacer una sola vez, y es por esta razón que en una aplicación real no la veremos en el método de renderizado, pero resulta conveniente ubicarla aquí por propósitos didácticos.

En segunda instancia, se le indica que funciones calcularan el vertex shader y/o pixel shader para renderizar el objeto y sobre cual shader model se trabajara, además de otros parámetros. HLSL incorpora para este propósito un esquema llamado técnica, tema que discutiremos en el capitulo siguiente.

Hay que tener en cuenta que Microsoft llama a los shader **efectos**, y es por esto que en este código vemos el nombre effect en lugar del nombre shader.

Ahora estamos casi listos para poder empezar a renderizar, pero primero se necesita conocer la cantidad de pasadas que tiene el shader. En la próxima sección se profundizara sobre el tema de pasadas, solo necesitaremos saber por el momento que un shader puede dividirse en pasadas, cada una de las cuales genera información útil a la siguiente.

Por cada pasada se setean las variables globales que utilizara el shader, como la posición de la cámara, información de luces, etc. Luego, se indica que comenzara la pasada y se renderiza el objeto invocando las mismas funciones estándares que usábamos cuando utilizábamos el pipeline gráfico de función fija. Al terminar de renderizar el objeto indicamos la finalización de la pasada. Por último, cuando todas las pasadas se hayan calculado se debe indicar que terminamos de procesar el shader. Por cuestiones de eficiencia, si una variable global es utilizada por varias pasadas, se podría pasar el valor de estas variables antes.

D3DX es una API de alto nivel que se ubica sobre la API Direct3D. D3DX fue introducida en Direct3D 8 y fue mejorada cuando salió Direct3D 9. D3DX contiene rutinas para hacer operaciones comunes a varios tipos de aplicaciones 3D.

En el caso particular de DirectX, se utiliza una API llamada D3DX para compilar los shaders escritos en HLSL o en lenguaje ensamblador. DirectX no sabe nada a cerca de HLSL, solo conoce el código maquina ya compilado. Este es un buen esquema, debido a que el compilador puede ser actualizado independientemente de DirectX. De hecho, las distintas evoluciones de DirectX 9 fueron cambios al compilador incluido en D3DX, generalmente cambios que afectaban la optimización del código generado. Esta política cambia en Direct3D 10 donde la única forma de escribir los shaders es en HLSL y donde el sistema de efectos y el compilador de HLSL son parte básica del API y no pertenecen a la librería de extensiones D3DX.

2.7 Renderizar a textura

A lo largo del texto asumimos que el contenido del frame buffer se reproducía siempre en la pantalla, lo cierto es que se puede almacenar en una textura, lo cual puede ser increíblemente útil. Por ejemplo, podríamos usar esta característica para producir environment mapping dinámico. Pero como podrán imaginar este no es el caso por el cual se dijo que es increíblemente útil.

Esta característica tiene otra importante cualidad, y es la de permitir el concepto de pasadas. Como dijimos anteriormente, un shader puede dividirse en pasadas, cada una de las cuales genera información útil a la siguiente.

La idea es renderizar una pasada en una textura, luego renderizar la próxima utilizando la información de la pasada anterior almacenada en esa textura, y así siguiendo hasta calcular la pasada final.

Históricamente, las múltiples pasadas servían para sobrepasar el límite máximo en el tamaño de los shader de ese momento, dado que ese límite era muy reducido. Por supuesto que no siempre se utilizaba este esquema, dado que las GPUs de ese entonces no eran tan potentes como para poder ejecutar siempre shaders de gran longitud. Es más, el shader debía poder dividirse en pasadas, algo no siempre posible. Este esquema fue muy utilizado en su entonces para calcular las contribuciones de luces por separado que influían a un objeto. Por otro lado, muchos shaders se mapean mejor en pasadas, como por ejemplo HDR, glow, por nombrar algunos.

Pero tal vez el uso más importante de renderizar a una textura es el de poder crear efectos de post procesado con los shaders. Estos efectos trabajan sobre una escena ya renderizada, y para que esto sea posible se debe poder acceder a la escena ya renderizada. Otra vez podemos nombrar como ejemplos a HDR y glow, pero tengamos en claro que son ejemplos de distintas cosas.

Por último es conveniente aclarar que se necesita un poco de trabajo extra en nuestra aplicación 3D para trabajar con renderizaciones a texturas. Siempre debemos indicar que vamos a renderizar a una textura, en vez de a la pantalla. Y también, siempre debemos pasarle la textura con la imagen renderizada al shader.

2.8 Pipeline gráfico de función fija vs. pipeline gráfico programable

Hemos hablado de lo bueno que es el pipeline programable, pero no hemos hablado sobre la desventaja que tiene sobre el pipeline de función fija.

El programador que usa el pipeline gráfico de función fija está acostumbrado a usar las herramientas que le brinda la API para definir luces, cámaras, la posición del objeto y el material del mismo. Todas estas herramientas no tienen efecto sobre el pipeline programable, las ignora completamente. Obviamente esta “limitación” se debe justamente a la filosofía de los shader. El shader calcula todos estos atributos como la iluminación, proyección y posición usando su propio código y su propia información.

Por otro lado, mucho del software actual no necesita de los shaders para lograr su cometido, por ejemplo, una aplicación industrial. Es por eso que muchos programadores siguen prefiriendo al pipeline gráfico de función fija como la opción ideal. Todo esto no tiene demasiados inconvenientes mientras se permita la opción de elegir entre los dos pipelines. Sin embargo, DirectX 10 solo permite el uso del pipeline gráfico programable. Más allá de los justificativos que Microsoft pueda tener, los cuales podrían ser válidos, considero una mala idea no proveer herramientas al programador para evitar el uso de shaders, aun cuando estas herramientas se traduzcan internamente a shaders. En la práctica, sin embargo, no es un tema de mayor importancia dado que la mayoría de las aplicaciones actuales que usan el pipeline gráfico de función fija utilizan OpenGL.

CAPITULO III
LENGUAJES

Command & Conquer 3 Tiberium Wars

3 Introducción

En el primer capítulo hablamos de RenderMan, el primer lenguaje utilizado para programar shaders. Este lenguaje, junto con otros de su estilo, está enfocado en dar la mayor calidad de imagen sin importar tanto los aspectos de rendimiento. Se requiere poco conocimiento de programación y de hardware, lo cual resulta ideal para los artistas. Y como es de imaginar, es utilizado en aplicaciones de diseño 3D como 3D MAX, Softimage o Maya dado que no tienen restricciones tan ajustadas como los gráficos 3D de tiempo real. Houdini VEX y Gelato son otros dos lenguajes populares y similares a RenderMan.

En contra partida, en el mundo de los gráficos en tiempo real, se tuvo que adoptar un esquema totalmente diferente. En un comienzo se escribían usando lenguajes ensambladores especiales para tal propósito. Además, el primer shader model era muy limitado en cantidad de instrucciones por shader. Todo esto hacia que programar shader en los inicios de esta nueva tecnología sea algo muy tedioso y poco práctico.

Es más, el hardware en si mismo diverge de los estándares del lenguaje ensamblador, por lo que los lenguajes ensambladores en realidad son una representación intermedia. No hay razón de realizar optimizaciones a bajo nivel en un lenguaje ensamblador si el código será transformado por el driver de la GPU.

Afortunadamente, como es normal en la evolución de la programación, no tardo mucho en surgir lenguajes de alto nivel para programar shaders de tiempo real. Estos lenguajes permiten realizar shaders de manera más accesible, evitándonos preocupaciones sobre detalles del hardware, tales como que registros internos usaremos. Además, entre otros beneficios, le relegamos varias decisiones de optimización al compilador.

Los lenguajes existentes utilizan sintaxis similar a C, permiten definir funciones, crear tipos de datos definidos por el usuario y usar sentencias como *if* o *for*. También, incluyen un amplio conjunto de funciones predefinidas. Todo esto hace al desarrollo de shaders más orientado hacia el diseño de algoritmos y menos enfocado en como codificar estos algoritmos.

Dado su corta historia, no podemos esperar que los lenguajes de shaders sean tan sofisticados y maduros como los lenguajes de programación tradicionales. Como podrán imaginar, tampoco sirve usar un lenguaje de programación general como lenguaje para desarrollar shaders, por dos razones particulares. Primero, las GPUs tienen algunas características especiales, tales como operaciones en tuplas en vez de escalares, el acceso y procesado de los datos que guarda una textura (característica comúnmente llamada texture lookups) y computación del tipo stream (computación altamente paralela). Esas características no están reflejadas en el diseño de un lenguaje de programación de propósito general. Segundo, algunas características de los lenguajes de programación general no son soportadas por el GPU. Por ejemplo, los shaders están limitados a cierta longitud, longitud que día a día es más larga pero aún limitada. Otro ejemplo es la incapacidad en las primeras generaciones de GPUs programables de realizar saltos en el código.

3.1 Los primeros lenguajes

A pesar de que los primeros lenguajes no resultaron ampliamente usados, tuvieron un rol importante como primer paso para lograr el uso de shaders en gráficos 3D de tiempo real. A continuación se hablara de estos lenguajes, pero no se describirán en profundidad.

Pfman shading language es el primer lenguaje de shaders de alto nivel de tiempo real. Fue diseñado específicamente para el hardware gráfico PixelFlow, ambos desarrollados por la universidad de Carolina del Norte en 1995. Pfman fue modelado después de RenderMan y es bastante similar a él. En ese entonces, RenderMan era usado intensivamente en la industria del cine, y es por eso que los desarrolladores de Pfman quisieron hacer fácil portar shaders a PixelFlow. A pesar de que PixelFlow se adapta bien para la programación de shaders, por varias razones, se usaron esquemas distintos para futuras implementaciones de shaders en tiempo real.

En el año 2000, SGI, diseñadora de OpenGL, desarrollo su propio lenguaje de alto nivel para programar shaders. Este lenguaje, llamado **Interactive Shading Language** o **ISL**, está construido sobre las capacidades existentes de OpenGL. El compilador de ISL no traduce archivos escritos en ISL a código maquina directamente. En cambio, convierte archivos ISL en archivos descriptores de pasadas, y luego, traduce esos archivos a código C/OpenGL. Básicamente, su idea fue usar a la API OpenGL como un lenguaje ensamblador para ejecutar shaders.

ISL no es lenguaje complejo como lo es RenderMan o Pfman debido a que se ejecuta sobre hardware con pipeline gráfico de función fija. Aún así, el esquema es interesante dado que trata de utilizar shaders en un pipeline no programable usando conceptos como las múltiples pasadas.

Paralelo a este, surgió el **Real-Time Shading Language** o **RTSL**, el cual es otro lenguaje de alto nivel diseñado sobre OpenGL, pero esta vez considerando las extensiones de OpenGL que utilizan las unidades programables de vértices y fragmentos. RTSL se podría ver como el padre de los lenguajes actuales, dado que todos ellos siguen su filosofía.

3.2 Los lenguajes actuales

Con el surgimiento de la primera generación de GPUs con pipeline programable el mercado comenzó a centrar su atención en esta tecnología. Además, al mismo tiempo surgía una nueva generación de consolas formada por la Playstation 2, la X-BOX y la GameCube. Pronto, títulos como el Prince of Persia: The sands of Time solo correrían en GPUs con soporte para shader model 1 o superior.

Como resultaron nacieron tres lenguajes que se establecerían como las opciones para programar shaders de tiempo real. Estos lenguajes son HLSL de Microsoft, Cg de NVIDIA y GLSL (OpenGL).

A continuación hablaremos de estos lenguajes, mostrando sus características y diferencias principales. Al principio describiremos de forma completa a HLSL y luego mostraremos las diferencias con Cg y GLSL. HLSL será mostrado sin tener en cuenta las repercusiones de DirectX 10, debido a que se analizaran por separado.

3.3 HLSL

En el 2002, el **High Level Shading Language** es introducido por Microsoft en la versión 9.0 de DirectX. Comparte tanto la sintaxis como semántica con el lenguaje Cg de NVIDIA, haciéndolos compatibles a nivel de código fuente. Desde cierto punto de vista se puede ver a HLSL como un subconjunto de Cg, aún si esto nunca es mencionado ni en la documentación de HLSL ni la especificación de Cg.

La razón de esta similitud se basa en que ambos originalmente fueron diseñados por el mismo grupo de desarrolladores. Fue un proyecto en conjunto de Microsoft con NVIDIA que evoluciono por caminos separados. Esta colaboración estuvo motivada por la XBOX, la consola de Microsoft, en la cual NVIDIA desarrollo el GPU de la misma.

HLSL está basado en la sintaxis del lenguaje C con algunas características de C++ tales como el tipo de dato booleano, definiciones de estructuras y comentarios. Como ya saben, dado las limitaciones del GPU, HLSL no puede implementar el subconjunto completo de características del lenguaje C. Entre otras cosas, por ejemplo, no puede operar con punteros.

3.3.1 Palabras reservadas

La siguiente es una lista de las palabras reservadas por el lenguaje. Las marcadas con asterisco no son case sensitive.

asm*	bool	break	continue
compile	const	decl*	do
double	else	extern	false
float	for	half	if
In	inline	inout	int
matrix*	out	pass*	pixelshader*
return	sampler	shared	static
string*	struct	switch	technique*
texture*	true	typedef	uniform
vector*	vertexshader*	void	volatile
while			

La siguiente lista son palabras reservadas para uso potencial en el futuro:

Auto	case	catch	char
class	compile	const_cast	default
delete	dynamic_cast	enum	explicit
friend	goto	long	mutable
namespace	new	operator	private
protected	public	register	reinterpret_cast
short	signed	sizeof	static_cast
template	this	throw	try
typename	union	unsigned	using
virtual			

3.3.2 Tipos de datos

El preprocesador de C brinda **compilación condicional**. Esto permite tener diferentes versiones del mismo código en el mismo archivo fuente. Típicamente, se usa para personalizar el programa con respecto a la plataforma de compilación, o habilitar código de testeo. Veamos un ejemplo:

```
#define x
...
#ifdef x
...
#else
...
#endif
```

Además de que HLSL soporta un preprocesador como el de C que nos permite compilación condicional. También extiende al lenguaje C introduciendo características que permiten utilizar aspectos específicos de la GPU. Mayormente es cuestión de tipos de datos nuevos y sus operaciones.

A continuación veremos una descripción de los tipos de datos discriminados en categorías.

Escalares

Tipo Escalar	Valores posibles
bool	True o false
int	Entero signado de 32 bits
uint	Entero no signado de 32 bits
half	Número en punto flotante de 16 bits
float	Número en punto flotante de 32 bits
double	Número en punto flotante de 64 bits

No todos los shader models tienen soporte nativo para int, half o doublé. Si el shader es compilado para un shader model que no soporta un formato especifico, será emulado a través del uso del tipo float, por lo que el resultado podría no ser preciso. Similarmente, el tipo de dato int es en muchos casos reemplazado por el tipo float, con excepción de las variables para bucles o la indexación de arreglos, debido a que el GPU muchas veces no dispone de suficientes registros temporales de lectura escritura para enteros.

Vectores

Tipo Vector	Valores
Vector	Un vector de cuatro componentes de tipo float
vector <tipo, tamaño>	Un vector conteniendo n componentes del tipo especificado

El tipo vector es un arreglo unidimensional compuesto de un tipo escalar particular con cuatro componentes como máximo. Por defecto, un vector es un arreglo compuesto por cuatro valores en punto flotante. Sin embargo, se puede definir manualmente vectores arbitrarios.

Sin embargo, la manera más común de declarar vectores es usar el nombre del tipo seguido por un número entero entre dos y cuatro. HLSL define por defecto estos tipos usando typedef para brindar más comodidad al desarrollador. Por ejemplo, para declara un tupla de cuatro floats, podríamos usar cualquiera de las siguientes declaraciones:

```
float4 fVector0;
float  fVector1[4];
vector fVector2;
vector <float, 4> fVector3;
```

Como se ve en el ejemplo anterior, se podría implementar vectores con la ayuda de arreglos, el cual es otro tipo disponible en HLSL. Sin embargo, en tales casos el compilador no podría distinguir entre arreglos reales de cuatro floats y un vector de cuatro componentes de tipo float.

Los componentes de los vectores pueden accederse de varias maneras. A continuación se muestra las distintas posibilidades para acceder a estos componentes:

```
Por componente:       vector.x,  vector.y,  vector.z,  vector.w
Por color:            vector.r,  vector.g,  vector.b,  vector.a
Por índice:           vector[0], vector[1], vector[2], vector[3]
```

Por conveniencia, también se puede acceder a los componentes del vector combinando múltiples componentes (por ejemplo *vector.xzzy*). De esta forma, los componentes deben especificarse usando o {x, y, z, w} o {r, g, b, a} pero no ambos. También se puede repetir componentes, siempre y cuando se use al vector para lectura y no escritura. Esta forma de acceder a los datos a menudo se llama **swizzle**.

```
float4 pos  = {3.0f, 5.0f, 2.0f, 1.0f};
float2 vec0 = pos.xy; // vec0 es {3.0f, 5.0f}
float2 vec0 = pos.xx; // vec0 es {3.0f, 3.0f}
float2 vec1 = pos.ry; // Invalido
float2 vec2;
vec2.xx = pos.xy;     // Invalido
```

Matrices

Tipo Matriz	Valores
Matrix	Un vector de cuatro por cuatro componentes de tipo float
matrix <tipo, filas, columnas>	Un vector conteniendo n filas y m columnas de componentes del tipo especificado

Son similares a los vectores. Las principales diferencias radican en los modos de acceso. Permiten acceder a determinadas filas usando un acceso del tipo arreglo. Por ejemplo, podríamos acceder a una sola fila usando

Matrix[3]. Dado que el resultado es un vector podríamos acceder a un componente especifica usando *Matrix[3].x* o *Matrix[3][0]*.

Los componentes de una matriz también pueden accederse usando alguna de las siguientes dos notaciones:

```
1-based:
_11 _12 _13 _14
_21 _22 _23 _24
_31 _32 _33 _34
_41 _42 _43 _44

0-based:
_m00 _m01 _m02 _m03
_m10 _m11 _m12 _m13
_m20 _m21 _m22 _m23
_m30 _m31 _m32 _m33
```

Las matrices, al igual que los vectores, permiten el acceso tipo swizzle. Por ejemplo: *Matrix._m01_m02_m03_04*. Como es de suponer, tampoco podremos usar las dos notaciones a la vez. Por ejemplo: *Matrix._m01_11* no es permitido.

Estructuras

Las estructuras, al igual que en C, son tipos compuestos usados para agrupar variables comunes en una única entidad. Las estructuras se declaran y se usan usando la siguiente sintaxis:

Declaración del tipo estructura:

```
struct [ID] {miembros}
```

Uso:

```
ID.miembro
```

Ejemplo:

```
struct Circulo          // Defino la estructura circulo
{
   float4 Posicion;
   float Radio;
};

Circulo MiCirculo;      // Declaro una variable de tipo circulo

MiCirculo.Radio = 2.0; // Asigno a la variable radio de la estructura MiCirculo el valor 2
```

Samplers

Por cada textura que deseamos usar en un pixel shader se debe declarar un sampler. Un sampler es responsable de indicar que textura se usara, y que filtrados y direccionamientos se le aplicara a dicha textura.

Por ejemplo, el siguiente código define un sampler para la textura *TexturaDifusa*:

```
sampler2D TexturaDifusaSampler = sampler_state
{
        Texture = <TexturaDifusa>;
        MinFilter = Linear;
        MagFilter = Linear;
        MipFilter = Linear;
        AddressU = WRAP;
        AddressV = WRAP;
};
```

Los distintos tipos de samplers son sampler1D, sampler2D, sampler3D y samplerCube, lo cual depende de si la textura es de una, dos o tres dimensiones o si es cubica.

Modos de direccionamiento de la textura

Las texturas 2D, por ejemplo, son mapeadas a la geometría usando las coordenadas de texturas U y V. Un valor UV de 1,1 se encuentra abajo a la derecha de la textura, mientras que un valor 0,0 se encuentra arriba a la izquierda. En otras palabras, el valor U se incrementa hacia la derecha, mientras que el valor V se incrementa hacia abajo.

Si el valor es mayor a 1, podemos tomar tres decisiones:

- WRAP: indica que la textura será repetida si los valores son mayores a 1. De esta manera si usamos un valor U de 10, la textura será repetida 10 veces en la dirección U. WRAP es el valor por defecto.
- MIRROR: es similar a WRAP, con la diferencia que cada repetición de la textura será la versión espejada de la repetición previa.
- CLAMP: restringe los valores en el rango 0 a 1. Es decir, el color de los valores que superan a 1 será el mismo que el usado para 1.

Los modos de direccionamiento pueden especificarse independientemente para U, V y W usando AdressU, AdressV y AdressW, respectivamente.

Filtrados

Cuando la textura es renderizada a un tamaño más grande de la que fue creada se dice que ha sido magnificada. Por el contrario, si es renderizada a menor tamaño diremos que ha sido, usando el termino en ingles, minified. Los mipmaps y los tipos de filtrado serán definidos de manera completa en el titulo “LOD en texturas: Mipmapping” que forma parte del apéndice.

Por lo tanto, podemos definir el filtrado a usar cuando la textura ha sido magnificada (MagFilter), cuando ha sido minified (MinFilter), y también podemos decir cómo combinar los diferentes niveles de mipmaps (MipFilter).

Los tipos de filtrados disponibles son:

- None: no se utiliza ningún tipo de filtrado.
- Point: no hay interpolación entre texels. Básicamente es el filtrado lineal sin considerar los mipmaps.
- Linear: hay interpolación lineal entre texels. Básicamente es el filtrado bilineal.
- Anisotropic: es el filtrado anisotrópico.

MagFilter y MinFilter pueden usar cualquiera de los tres tipos de filtros. MipFilter solo puede usar Point y Linear.

Por lo tanto si quisiéramos utilizar un filtrado bilineal haríamos las siguientes combinaciones:

MagFilter = MinFilter = Linear y MipFilter = Point.

Y si quisiéramos trilineal:

MagFilter = MinFilter = Linear y MipFilter = Linear.

Texturas

Por su parte, las texturas se deben declarar. Para nuestro ejemplo anterior, podríamos usar alguna de estas dos formas de declarar texturas:

```
texture TexturaDifusa;
 o
Texture2D TexturaDifusa;
```

Los tipos posibles para las texturas son: Texture1D, Texture1DArray, Texture2D, Texture2DArray, Texture3D y TextureCube. Además, se incluye el tipo texture, el cual es un supertipo de los anteriores, que está para mantener la compatibilidad con especificaciones anteriores.

Tipos definidos por el usuario

El typedef de HLSL funciona exactamente igual que en C++:

typedef [const] Tipo ID [Índice];

const marca explícitamente al tipo como constante.

Como se dijo antes, varios tipos son automáticamente definidos por conveniencia, por ejemplo:

```
typedef vector <float, 4> float4;
```

3.3.3 Variables

Las variables se definen siguiendo la siguiente sintaxis:

[storage_class] [modificador_de_tipo] tipo id [índice] [: semántica] [= inicialización] [: registro];

Los modificadores storage class dan información sobre el alcance y tiempo de vida de la variable. Los modificadores static y extern no pueden usarse en simultáneo, como tampoco uniform y volatile. Los tipos de modificadores son:

Prefijo	Significado
static	Para variables globales significa que el valor es interno y no puede ser accedido por la aplicación, solo por el shader. Para variables locales, sin embargo, indica que el valor persiste. La inicialización de variables estáticas es hecha una sola vez.
extern	El modificador extern puede usarse en variables globales para indicar que puede ser modificado a fuera del shader a través de la API. Este es el comportamiento por defecto para variables globales.
uniform	Las declaraciones de variables globales con el prefijo uniform indican que no su valor no cambia, excepto entre llamadas de renderizado. Uniform es el comportamiento por defecto de todas las variables globales.
volatile	Indica que el valor de la variable cambiara frecuentemente. Este modificador solo se aplica a parámetros globales.
shared	Indica que la variable será compartida entre efectos (ver effect framework, sección 3.6).
nointerpolation	No interpolar la salida de un vertex shader antes de pasársela al pixel shader.

Los modificadores de tipos (llamados así por Microsoft) son:

Prefijo	Significado
const	Específica que el valor de la variable no cambiara. Por esta razón, el valor debe inicializarse en la declaración de la variable.
row_major	Indica que cada fila de la matriz será almacenada en un único registro constante.
colum_major	Indica que cada columna será almacenada en un único registro constante. Esta organización facilita las operaciones de matrices y por eso es la opción por defecto.

Los modificadores row_major y col_major se utilizan solo para matrices, y sirven para especificar la organización esperada de dicha matriz dentro del almacenamiento de hardware.

La inicialización de variables es similar a la usada en C. Por ejemplo:

```
float2x2 fMat    = {3.0f, 5.0f,  // Columna 1
                    2.0f, 1.0f}; // Columna 2
float4   vPos    = {3.0f, 5.0f, 2.0f, 1.0f};
float    fFactor = 0.2f;
```

También podemos especificar a qué registro constante particular se mapeara la variable. Si el tipo de la variable no encaja con el tamaño de un único registró, subsecuentes registros serán usados.

3.3.4 Semántica

HLSL introduce un nuevo constructor, el cual especifica la semántica de una variable particular y/o un parámetro de una función. Este constructor permite al código de un shader conectarse con un registro particular de la GPU

permitiendo de esta manera acceder y modificar los datos procesados por el pipeline. La semántica es utilizada mayormente por un shader para acceder a los datos del pipeline y para especificar que datos devolverá al pipeline. Es más, la semántica permite entender mejor el código, dado que sabremos más de la variable viendo solo su declaración.

Sintácticamente, la semántica se escribe después del identificador del elemento de código particular para la cual la semántica es aplicada. Por ejemplo, la posición del vértice que será procesado por el vertex shader es denotada como *float4 pos: POSITION* en la lista de parámetros de la función que calcula dicho vertex shader.

También permite conectar una variable global a un registro constante particular, el cual puede setearse por la aplicación sobrepasando directamente la interfaz de un shader. Esto también permite una compatibilidad hacia atrás con aplicaciones que utilizan código de shader en lenguaje ensamblador, debido a que en tales casos la aplicación tiene una referencia constante por el nombre del registro en vez de usar su nombre lógico en el código del shader.

Las posibles semánticas de entrada de los vertex shaders:

Semántica	Descripción
POSITION*n*	Posición
BLENDWEIGHT*n*	Blend Weight
BLENDINDICES*n*	Indices Blend
NORMAL*n*	Vector normal
PSIZE*n*	Tamaño del punto
COLOR*n*	Color
NORMAL*n*	Vector normal
TEXTCOORD*n*	Coordenadas de Textura
TANGENT*n*	Vector tangente
BINORMAL*n*	Vector binormal
TESSFACTOR*n*	Factor de teselación

Las posibles semánticas de entrada de los pixel shaders:

Semántica	Descripción
COLOR*n*	Color
TEXTCOORD*n*	Coordenadas de Textura

Las posibles semánticas de salida de los vertex shaders:

Semántica	Descripción
POSITION*n*	Posición
PSIZE*n*	Tamaño del punto
COLOR*n*	Color
FOG*n*	Vertex Fog
TEXTCOORD*n*	Coordenadas de Textura

Las posibles semánticas de salida de los pixel shaders:

Semántica	Descripción
COLOR*n*	Color para el render target n
DEPTH*n*	Valor de profundidad

Donde n es un entero opcional que sirve para especificar a qué registro especifico mapea, y el cual varía entre 0 y el número de recursos soportados. Como por ejemplo: *TEXTCOORD3* o *COLOR0*.

3.3.5 Conversiones de tipo

Existen una gran cantidad de conversiones incorporadas en el lenguaje.

Conversión	Valides
Escalar a escalar	Esta conversión siempre es válida. Cuando aplicamos una conversión de un tipo booleano a un tipo entero o punto flotante, false es considerado cero y true uno. Cuando convertimos desde un tipo entero o punto flotante a uno booleano, un cero es considerado false, y cualquier otro valor es considerado true. Cuando aplicamos una conversión de un tipo en punto flotante a un tipo entero, el valor es redondeado al entero más cercano.
Escalar a vector	Esta conversión siempre es válida. Replica al escalar de manera de llenar al vector.
Escalar a matriz	Esta conversión siempre es válida. Replica al escalar de manera de llenar la matriz.
Escalar a objeto	Conversión invalida.
Escalar a estructura	Replica al escalar de manera de llenar la estructura.
Vector a escalar	Esta conversión siempre es válida. Se selecciona el primer componente del vector y se lo asigna al escalar.
Vector a vector	El vector destino no debe ser más grande que el vector fuente. La conversión mantiene los primeros componentes y descarta el resto. Para este tipo de conversión, las columnas de las matrices, las filas de las matrices y las estructuras numéricas son tratadas como vectores.
Vector a matriz	Para que esta conversión sea válida, el tamaño del vector debe ser igual al tamaño de la matriz.
Vector a objeto	Conversión invalida.
Vector a estructura	Es válida si la estructura no es más grande que el vector, y si todos los componentes de la estructura son numéricos.
Matriz a escalar	Esta conversión siempre es válida. Se selecciona el primer componente de la matriz (primera fila y primera columna) y se lo asigna al escalar.
Matriz a vector	Para que esta conversión sea válida, el tamaño de la matriz debe ser igual al tamaño del vector.
Matriz a matriz	Para que esta conversión sea válida, la matriz destino no debe ser más grande que la matriz fuente, en ambas dimensiones. La conversión funciona manteniendo los valores de más arriba y de más a la izquierda, descartando el resto.
Matriz a objeto	Conversión invalida.
Matriz a estructura	Para que esta conversión sea válida, el tamaño de la estructura debe ser igual al tamaño de la matriz, y los componentes de la estructura deben ser todos de tipo numérico.

Objeto a escalar	Conversión invalida.
Objeto a vector	Conversión invalida.
Objeto a matriz	Conversión invalida.
Objeto a objeto	Este tipo de conversión es válida solo si ambos objetos son del mismo tipo.
Objeto a estructura	Para que esta conversión sea válida, la estructura no debe contener más de un miembro. El tipo del miembro debe ser idéntico al tipo del objeto.
Estructura a escalar	Para que esta conversión sea válida, la estructura debe contener al menos un miembro y este miembro debe ser numérico.
Estructura a vector	Para que esta conversión sea válida, la estructura debe tener al menos el tamaño del vector. Y las primeras componentes deben ser numéricas hasta el tamaño del vector.
Estructura a matriz	Para que esta conversión sea válida, la estructura debe tener al menos el tamaño la matriz. Y las primeras componentes deben ser numéricas hasta el tamaño de la matriz.
Estructura a estructura	Para que esta conversión sea válida, la estructura fuente no debe ser más grande que la estructura destino. Y además, una conversión válida debe existir para cada uno de los componentes.

Objeto se refiere a tipos como samplers o texturas.

3.3.6 Expresiones, operadores y sentencias

El manejo de expresiones, operadores y sentencias en HLSL es muy similar al de C. Al igual que C, HLSL soporta el concepto de bloque de sentencias (*{ sentencias 1; … sentencias n; }*) en donde cada sentencia está compuesta de expresiones y operadores.

La siguiente es una lista de los operadores disponibles y su significado. No se describirá como se arman los distintos tipos de sentencias y expresiones, dado que resulta fácil de deducir dado el conjunto de operadores disponibles.

Operador	Uso	Significado	Asociatividad
()	(valor)	Sub expresión	A izquierda
()	id(argumento)	Llamada a función	A izquierda
()	tipo(argumento)	Constructor de tipo	A izquierda
[]	arreglo[entero]	Indexación de arreglo	A izquierda
.	estructura.id	Selección de un miembro	A izquierda
.	valor.swizzle	Swizzle	A izquierda
++	variable++	Incremento posfijo (por componente)	A izquierda
--	variable--	Decremento posfijo (por componente)	A izquierda
++	++variable	Incremento prefijo (por componente)	A derecha
--	--variable	Decremento prefijo (por componente)	A derecha
!	!valor	No lógico (por componente)	A derecha
-	-valor	Menos unario (por componente)	A derecha
+	+valor	Más unario (por componente)	A derecha
()	(tipo) valor	Conversión o cast	A derecha
*	valor * valor	Multiplicación (por componente)	A izquierda
/	valor / valor	División (por componente)	A izquierda

%	valor % valor	Modulo (por componente)	A izquierda
+	valor + valor	Suma (por componente)	A izquierda
-	valor - valor	Resta (por componente)	A izquierda
<	valor < valor	Menor (por componente)	A izquierda
>	valor > valor	Mayor (por componente)	A izquierda
<=	valor <= valor	Menos igual (por componente)	A izquierda
>=	valor >= valor	Mayor igual (por componente)	A izquierda
==	valor == valor	Igual (por componente)	A izquierda
!=	valor != valor	Distinto (por componente)	A izquierda
&&	valor && valor	Y lógico (por componente)	A izquierda
||	valor || valor	O lógico (por componente)	A izquierda
?:	bool?valor:valor	Condicional	A derecha
=	variable = valor	Asignación (por componente)	A derecha
*=	variable *= valor	Asignación/multiplicación (por componente)	A derecha
/=	variable /= valor	Asignación/división (por componente)	A derecha
%=	variable %= valor	Asignación/modulo (por componente)	A derecha
+=	variable += valor	Asignación/suma (por componente)	A derecha
-=	variable -= valor	Asignación/resta (por componente)	A derecha
,	valor, valor	Coma	A izquierda

A diferencia del lenguaje C, la evaluación de las expresiones &&, || y ?: no es en corto circuito debido a la forma en que son evaluadas por el hardware. Además, muchos operadores están etiquetados “por componente”, lo cual indica que el operador es aplicado a cada componente independientemente. Por ejemplo, una suma de vectores dará como resultado un vector con la suma de cada componente por separado.

3.3.7 Estructuras de control

HLSL soporta un amplio rango de estructuras de control con excepción de las no estructuradas, es decir los goto. Las estructuras de control soportadas son:

- break
- continue
- do
- for
- if
- stop
- switch
- while

La sentencia continue termina la ejecución de la iteración actual del ciclo, actualiza las condiciones del loop y comienza a ejecutar nuevamente una iteración desde el comienzo del ciclo. La sentencia stop finaliza la ejecución en la sentencia actual y retorna la salida. A nivel global, stop es idéntica a return. En cambio, dentro de una función, stop finaliza la ejecución del shader en vez de finalizar la ejecución de la función.

Tengamos en cuenta que recién a partir del shader model 3 se brinda la posibilidad de usar control de flujo dinámico, característica inexistente en las GPUs de las generaciones anteriores. Afortunadamente, el compilador puede evitar algunas de estas limitaciones. Por ejemplo, en un ciclo estático (ciclo que se ejecutara un número constante de veces) el compilador puede expandir el ciclo para satisfacer los requerimientos de la GPU particular. En este caso, el compilador se podría encontrar con la limitación en la longitud de código, problema que no puede evitar.

Branch predication es una estrategia que usualmente se utiliza para mitigar el costo asociado con saltos condicionales, particularmente saltos entre secciones de código cortas. Básicamente, permite que una instrucción realice condicionalmente una operación o no haga nada. Su principal desventaja es que cada instrucción ocupa más espacio.

Algunos shader models permiten el uso de branch predication, estrategia que no requiere ningún tipo de control de flujo. A su vez, branch predication se puede usar en saltos entre secciones de código cortas para mejorar el desempeño en shader models avanzados, o mejor dicho, en GPUs que tienen soporte de control de flujo dinámico. Esta mejora en el desempeño está dada por cuestiones a nivel arquitectura de los procesadores, en la cual se mejora la efectividad de la ejecución del pipeline (el pipeline del procesador que calcula el shader, no el pipeline gráfico).

El shader model 2, por ejemplo, da soporte al control de flujo estático. El cual permite que ciertos bloques de código se ejecuten o no basados en una contante booleana. Este es un método muy conveniente para habilitar o deshabilitar código que podría resultar potencialmente costoso, basado en el objeto a renderizar en ese momento. Entre llamadas de renderizado (la llamada que hace la API gráfica para renderizar al objeto), se puede decidir que características queremos soportar y de esta manera setear las banderas booleanas requeridas para producir ese comportamiento. La mejor parte de este método es que las instrucciones “deshabilitadas” serán completamente salteadas durante ejecución. La desventaja es que solo se puede cambiar esas banderas a baja frecuencia, es decir, entre llamadas de renderizado. En contraste, al usar branch predication es posible tomar esta decisión dinámicamente.

En caso de que no se puedan evitar las limitaciones del shader model elegido, el compilador dará un error en compilación.

3.3.8 Funciones

HLSL permite el uso de funciones definidas por el usuario, además de incorporar un conjunto de funciones propias. Definir funciones en HLSL es similar a la manera en que se definen en C. La sintaxis a seguir es:

```
[static inline target] tipo_a_retornar id ( [lista_de_parametros] ) { [sentensias] };
```

La función puede prefijarse con uno o más de las siguientes palabras reservadas:

Prefijo	Significado
Static	Indica que la función existirá dentro del alcance del shader actual y no será compartida.
Inline	Indica que las instrucciones de la función serán copiadas dentro del código llamador en vez de usar una verdadera llamada a función. Esto en realidad es una sugerencia al compilador, nada garantiza ese funcionamiento. En ciertos shader models, debido a sus limitaciones, esta es la única opción.
Target	Indica en que versión de pixel o vertex shader se pensara usar esta función. Esto permite al compilador tomar mejores decisiones cuando construye el código. Notemos que no escribiremos la palabra "target" en nuestro shader, en cambio, usaremos un nombre que denota la versión que queremos usar. Por ejemplo, usaremos ps_3_0 si pretendemos que va a correr usando pixel shader 3.0.

Además, todos los parámetros declarados pueden prefijarse con las siguientes palabras reservadas:

Prefijo	Significado
In	Es el comportamiento por defecto e indica que el parámetro será usado solo para lectura.
Out	Indica que el parámetro será usado para retornar un valor.
Inout	Combina el comportamiento de In y Out.
Uniform	Indica que el valor proviene de un registro constante. Generalmente, las variables globales se almacenan en registros constantes. Los valores de estas variables globales muchas veces se setean en las llamadas de renderizado del objeto, o mantienen el valor inicial asignado en el código del shader.

Cada shader debe contener al menos una función que sirva como punto de partida del shader. Además, las funciones pueden sobrecargarse de manera similar a C++, es decir, la función puede tener el mismo nombre pero debe diferir en el número y/o tipo de sus parámetros. Las funciones no pueden llamarse recursivamente, esto se debe a la manera en la cual las funciones son procesadas, compiladas y ejecutadas por el hardware.

Para una descripción de las funciones incluidas en HLSL consultar:

http://msdn2.microsoft.com/en-us/library/bb509611.aspx

3.3.9 Effect framework

El **Effects framework** permite agrupar shaders (o efectos, término que le gusta usar mucho a Microsoft) y otros datos relacionados en una unidad lógica. Los dos effects frameworks más populares son DirectX FX (o DXFX) y CgFX los cuales son una extensión del los lenguajes HLSL y Cg, respectivamente. Por cuestiones de simplicidad consideraremos a estos frameworks como parte de ambos lenguajes, dado su importancia y su amplio uso. Y es por eso que en este caso lo describiremos como parte de HLSL.

Los archivos de efectos usan la extensión .fx, y se los denomina comúnmente como archivos fx. Muchas veces llamemos efecto a esta unidad lógica. La estructura de un archivo fx es similar a la siguiente:

Variables Globales
Funciones
Técnicas
Pasadas
Estados del efecto
Especificación de que funciones calcularan los vertex y pixel shaders.

- **Variables globales**: son seteadas por el efecto o por la aplicación, y su alcance abarca a todo el efecto.
- **Funciones**: las funciones pueden escribirse tanto en HLSL como en lenguaje ensamblador.
- **Estados del efecto:** especifican las condiciones de renderizado del pipeline. En otras palabras, indican al pipeline como procesar la información. Muchas veces, algunos de estos estados se suelen setear para lograr un comportamiento particular y es por eso que su importancia es grande. Podrían indicar, por ejemplo, que se deshabilite el Z-Buffer, o especificar que las caras de un objeto poligonal se armen de cierta forma (cull mode). Para ver la lista completa de estados del efecto ver:
  http://msdn2.microsoft.com/en-us/library/bb173347.aspx#Light_States
- **Técnicas**: básicamente, una técnica nos dice funciones vamos a utilizar a la hora de renderizar cada pasada de un shader, especificando a su vez las condiciones de renderizado del pipeline. Las técnicas permiten a los desarrolladores agrupar distintas versiones de un mismo shader. Estas versiones contendrán distintos niveles del shader, variando desde un shader más potente y costoso, a shaders de menor calidad. Cada técnica puede definir una o más pasadas. Cada pasada especificara variables globales (para esa pasada), estados del efecto y también especificará que funciones calcularán los vertex y pixel shaders. El estado del pipeline se restablecerán a su estado anterior cada vez que una técnica finalice.

3.4 Cg

Su nombre proviene de “C for Graphics” y fue desarrollado por NVIDIA. Cg estuvo disponible antes que HLSL y GLSL. Es muy similar a HLSL y de acuerdo a sus características se lo podría considerar un super conjunto del lenguaje HLSL. Esto convierte a Cg y HLSL compatibles a nivel condigo fuente. De esta manera, solo describiremos las diferencias entre ambos lenguajes, en vez de enumerar todas sus características.

Su característica clave y única es su independencia a la API gráfica elegida, dado que es posible utilizar código escrito en Cg tanto en OpenGL como en DirectX. Es más, el código del compilador Cg es abierto, permitiendo a un tercero desarrollar un compilador con el objetivo de brindar soporte a una nueva plataforma de software y/o hardware. También de esta manera podemos asegurarnos la calidad de la compilación, si encontramos ineficiencias o bugs podemos corregirlos sin necesidad de depender del vendedor.

Además, Cg utiliza el concepto de perfiles. Un perfil define las capacidades y restricciones particulares de un shader model, de una GPU y de una API gráfica. Al momento de compilar el shader, podemos especificar el perfil del sistema en el que se ejecutara el shader y de esta manera obtener portabilidad y una mejor compilación para esa configuración particular de software y hardware. Notemos que la especificación del perfil es un comando al compilador Cg, no forma parte del código escrito en Cg.

Cg también permite la sobrecarga de funciones de acuerdo a perfiles. En otras palabras, permite especificar distintas versiones de una misma función para distintos perfiles. Esto puede resultar útil debido a que diferentes perfiles soportan diferentes subconjuntos de capacidades del lenguaje, y debido a que las implementaciones más eficientes de una función podrían ser diferentes para diferentes perfiles. El objetivo de todo esto es brindar soporte a GPUs más antiguas y limitadas.

```
perfilA float mifunc(float x) {...};

perfilB float mifunc(float x) {...};
```

Pero tengamos en claro que esta última característica no es una verdadera ventaja de Cg sobre HLSL, dado que HLSL agrega el concepto de effect framework, el cual vimos anteriormente. De hecho, las distintas ventajas de incorporar un effect framework dieron origen a CgFX, el effect framework de Cg.

Una interesante característica sacada de C++ disponible en el lenguaje Cg, es la posibilidad de agregar funciones a una estructura, convirtiendo a esa estructura en una versión limitada de una clase. Cg también soporta un mecanismo de interfaces que es utilizado para definir la funcionalidad general disponible en esas estructuras.

3.4.1 Diferencias entre Cg y HLSL

Además de las antes nombradas, existen otras diferencias entre ambos lenguajes. A continuación, se especificara las diferencias de código para convertir un shader escrito en HLSL a Cg, y viceversa. En ambos casos, consideraremos la inclusión del effect framework como parte del lenguaje.

La próxima es una lista utilizando solo ejemplos para entender las diferencias:

HLSL	*vsOut.Pos = mul(float4(pos,1) , Wvp);*
Cg	*vsOut.Pos = mul(Wvp, float4(pos,1));*

Donde Wvp y Pos se definen como: *float4x4 Wvp : WORLDVIEWPROJECTION; float4 Pos : POSITION;*

HLSL	*VertexShader = compile vs_1_1 myVS();* *PixelShader = compile ps_2_0 myPS();*
Cg	*VertexProgram = compile arbvp1 myVS();* *FragmentProgram = compile arbfp1 myPS();*

HLSL	*Texture[0] = NULL;*
Cg	*Texture2DEnable[0] = false;*

HLSL	*LightEnable[0] = false;*
Cg	*LightingEnable = false;* *LightEnable[0] = false;*

HLSL	*MipFilter = LINEAR;* *MinFilter = LINEAR;*
Cg	*MinFilter = LINEARMIPMAPLINEAR;*

HLSL	*float2 f2 = f3; //conversión implícita*
Cg	*float2 f2 = f3.xy;*

Donde f3 se define como: *float3 f3;*

HLSL	*float4x4 wit : INV_TWORLD;*
Cg	*float4x4 wi : INV_WORLD;* *static float4x4 wit = transpose(wi);*

Tener en cuenta estas diferencias puede resultar muy útil dado que ambos son los lenguajes de mayor uso, y por ende, los de mayor cantidad de ejemplos. A menudo, podría resultar conveniente transforma un código al lenguaje elegido para desarrollar shaders. Por supuesto que también resulta útil para notar las mínimas diferencias entre ambos lenguajes.

3.5 GLSL

GLSL es el último de los tres lenguajes en salir al mercado. Es parte de la especificación OpenGL 2.0 y es específico a esta plataforma. Similarmente a los otros dos lenguajes, GLSL está basado en C y comparte muchas características de HLSL y Cg.

Las diferencias de GLSL con HLSL radican en la manera en que se acceden los datos del pipeline, la sintaxis de los tipos de datos definidos por el usuario, pequeñas diferencias en el significado de algunos operadores matemáticos, la falta de perfiles y por último, la falta de un effect framework sin ningún reemplazante.

La falta de un mecanismo de perfiles es una diferencia llamativa. GLSL espera una funcionalidad mínima la cual se especifica por una extensión particular de OpenGL requerida para compilar y ejecutar el código del shader. La desventaja de este enfoque está basada en el hecho de que el estándar de OpenGL especifica solo la funcionalidad de las acciones válidas, en vez de dar una especificación completa. Esto permite a los vendedores de hardware manejar estas situaciones inválidas a su manera.

La implementación real de las extensiones es provista por los vendedores de hardware. Lo cual podría generar un escenario en el cual GLSL se ejecute en hardware que no provee toda la funcionalidad esperada, tal como, por ejemplo, la precisión numérica. Todo esto podría provocar la inestabilidad de la aplicación o la aparición de artifacts. Sin embargo, este problema puede evitarse si la aplicación testea que el hardware subyacente en el que se está ejecutando cumple con los requerimientos de las extensiones a través de su cadena de identificación.

3.6 Conclusiones

Las limitaciones de GLSL no lo hacen inutilizable, aunque también es cierto que es el menos utilizado de estos lenguajes. GLSL perdió mucho mercado cuando SONY decidió usar Cg como lenguaje de shaders. Por otra parte, en el mercado de PC tampoco es usado demasiado, probablemente por DirectX sea la elección a elegir por la mayoría de los desarrolladores. Y por supuesto, como es de imaginar, la opción elegida por Microsoft para su X-BOX 360 es HLSL.

Cg es utilizado en la Playstation 3 y en algunos, pero no muchos, títulos de PC. La Playstation 3 utiliza como API gráfica a PSGL, una variación de OpenGL ES que desplaza a GLSL en favor de Cg. La razón de este cambio radica en que NVIDIA desarrollo para SONY a la GPU de la PlayStation 3, la RSX.

A nivel práctico podemos decir que la elección de un lenguaje es influenciada por factores como la API a utilizar, el poder de las herramientas, la facilidad de integración al proyecto y los ejemplos disponibles, más que con factores específicos del lenguaje. En gran parte esto se debe a que estos lenguajes son muy similares.

Como opinión personal diría que el hecho de no tener un effect framework hace menos atractivo a GLSL, pero probablemente se deba a una costumbre, más que una verdadera desventaja. Herramientas de desarrollo de shaders como RenderMonkey evitan el uso del effect framework, utilizando una organización visual en vez de código.

Un tema que puede preocupar, es el soporte futuro de los lenguajes. Los shaders es una tecnología que todavía tiene que madurar y es por eso que es un tema que no debe tomarse tan a la ligera. Cg tal vez sea el que más duda puede generar, dado que David Kirk de NVIDIA manifestó en su momento la falta de interés de competir por imponer su lenguaje. Textualmente dijo:

> "As we started out with Cg it was a great boost to getting programmers used to working with programmable GPUs. Now Microsoft has made a major commitment and in the long term we don't really want to be in the programming language business"

Con la utilización de Cg en la PlayStation 3 la expectativa de vida de Cg cambio dramáticamente, nuevamente NVIDIA comenzó a prestarle atención. Este nuevo resurgimiento se puede ver en la nueva versión del FXComposer, la herramienta de NVIDIA que en su primera versión era solo exclusiva para HLSL. Además, Cg ya tiene definido los perfiles para Shader Model 5, lo cual es llamativo dado que el publico general se entero de la existencia de la definición de este shader model gracias a la nueva versión de Cg.

3.7 Otros lenguajes

Existen otros lenguajes además de HLSL, Cg y GLSL, de los cuales hablaremos a continuación. Primero se hablara de Sh, un enfoque para programar shaders totalmente distinto al de los lenguajes anteriormente vistos. Luego, se verán enfoques para la programación no gráfica, o cálculos de propósito general, sobre GPUs.

Desafortunadamente, solo se dará un breve pantallazo de estos enfoques, solo se tratará de conocer su naturaleza. No es el objetivo de este libro profundizar sobre este tema.

3.7.1 Sh

Sh es una librería que utiliza un mecanismo de metaprogramación, el cual permite escribir shaders dentro del programa principal, en C++. Parte del código es compilado cuando la aplicación principal se compila, y el resto (con ciertas optimizaciones) se compila en tiempo de ejecución. Esta es una idea similar a la que usamos cuando incorporamos consultas SQL en el código de nuestra aplicación.

Entre las características que posee podemos nombrar:

- Es Open source.
- Tiene soporte ya implementado para las plataformas Windows, Linux y Mac OS X. Además, también tiene soporte en OpenGL y DirectX.
- Dado que fue desarrollado por investigadores universitarios, Sh no está altamente ligado a ninguna plataforma particular, API gráfica, o fabricante de GPU.
- Está diseñado para poder usarlo para cálculos de propósito general.

Sh es un proyecto investigativo con un enfoque interesante. Aunque, en mi opinión, no tan útil como para usarlo en la práctica. ¿Por qué? Entre otras cosas, porque nos quita la gran ventaja de poder desarrollar shaders por separado, utilizando herramientas que no solo nos permiten desarrollarlos muy cómodamente, sino optimizarlos sin tener en cuenta a nuestro motor.

3.7.2 Cálculos de propósito general sobre GPUs

En el primer capítulo hablamos de usar a las GPU para realizar cálculos físicos. En realidad, las GPUs podrían ser útiles para realizar cualquier tipo de cálculos de propósito general. De hecho, actualmente es un área que está llamando mucha la atención, dado que el poder de cálculo de las GPUs es altísimo.

Los lenguajes anteriormente vistos (excepto Sh) reflejan la estructura de la GPU bastante bien. Y es por esto que necesitamos enfoques totalmente distintos. Actualmente existen dos enfoques interesantes, el CUDA de NVIDIA y el CTM de ATI.

Lamentablemente, no continuaremos con este tema, a pesar de que es verdaderamente muy interesante, y probablemente consiga repercutir considerablemente en el mercado en los próximos años.

CAPITULO IV
HERRAMIENTAS

Colin McRae DIRT

4 Introducción

Tener un lenguaje de alto nivel para shaders es un paso fundamental para tener un ambiente de desarrollo confortable y útil, pero no es el último. Crear shaders no es un trabajo mecánico, es una actividad que envuelve ciencia y arte, y es por esto que necesitamos herramientas que nos faciliten el trabajo.

Necesitamos un entorno de trabajo donde podamos ver los shaders que estamos creando o modificando sin necesidad de recurrir a nuestro motor gráfico. Esto nos posibilitara:

- Ver el resultado del shader de forma aislada.
- Despreocuparnos por errores en nuestro motor gráfico.
- Poder ver el rendimiento del shader y realizar posibles optimizaciones.
- Tener una interfaz amigable, con herramientas útiles.

Para poder hacer posible el uso de estas herramientas, debe existir alguna forma de pasar cierta información crucial con el propósito de ver el shader en funcionamiento. Esta información puede abarcar matrices de proyección, de vista y/o de mundo, texturas, y por supuesto la geometría a renderizar. Existen distintos enfoques para lograr este cometido, dependiendo de la herramienta en particular.

4.1 Herramientas

Existen varias herramientas populares, entre ellas podemos nombrar:

- **FX Composer**: desarrollada por NVIDIA.
- **RenderMonkey**: desarrollada por ATI.
- **ShaderWorks**: adquirida en el 2005 por Activision.
- **Dark Tree**: desarrollada por una pequeña empresa llamada Darkling simulations.

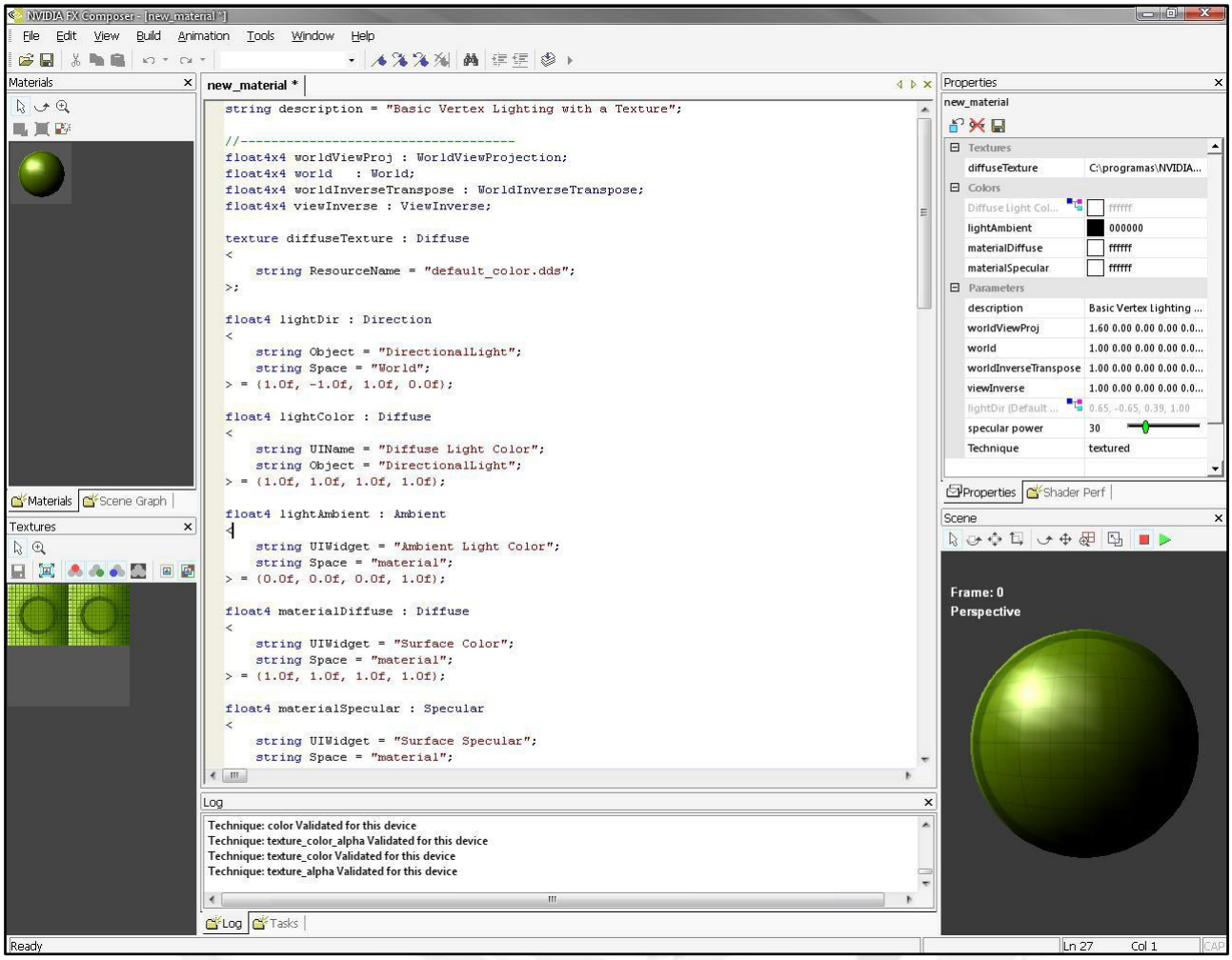
A continuación veremos FX Composer y RenderMonkey, las dos herramientas más populares.

4.2 FX Composer

FX Composer, la herramienta desarrollada por NVIDIA es una de las más utilizadas y completas del mercado. Actualmente, FX Composer se encuentra en su segunda versión, la cual incluye mejoras considerables. Por primera vez se ve una herramienta de creación de shaders con calidad similar a otras herramientas de desarrollo de otras disciplinas. Esto es algo muy bueno dado el poco tiempo de vida que tiene esta tecnología.

Comenzaremos viendo las características de la primera versión. La segunda versión sigue la misma filosofía, agrega y mejora la funcionalidad pero sigue utilizando el mismo esquema de trabajo. Más tarde discutiremos las cualidades de la segunda versión.

4.2.1 FX Composer 1



FX Composer 1.8

La primera versión de FX Composer trabaja sobre shaders escritos en HLSL. Como se puede apreciar en la captura de pantalla, el eje central de la IDE es el código HLSL. Lo cual no indica que sea un simple editor de texto, ya que incluye varias características interesantes.

FX Composer dispone, entre otras cosas, de las siguientes cualidades:

- Resaltado de sintaxis.
- Vista previa de uno o más shaders en una escena de prueba.
- Editor gráfico de propiedades del shader.
- Compilador interactivo.
- Administración de texturas usadas para las pruebas.
- Herramientas de análisis y optimización.
- Distintas herramientas de medición: FPS, conteo de ciclos del GPU, uso de registros y tasa de utilización.

La primera versión de FX Composer, a pesar de ser algo tosca y simple, es bastante cómoda de usar. Y es por esto que se gano un lugar importante entre las herramientas de creación de shaders. A continuación, veremos la forma en que FX Composer especifica la información que el shader necesita para poder probarlo en la vista previa.

4.2.2 Standard Annotations and Semantics (DXSAS)

FX Composer trabaja sobre código en HLSL, el cual respeta un estándar desarrollado por Microsoft conocido como standard annotations and semantics. DXSAS permite especificar en el propio código del shader cierta información sobre el significado de ciertas variables y sus valores iníciales. Esta información es obviada por el compilador de HLSL, solo tiene utilidad para herramientas o programas que entiendan DXSAS. Además, el uso de DXSAS tiene aparejada otra ventaja dado que nos permite entender el código más fácilmente.

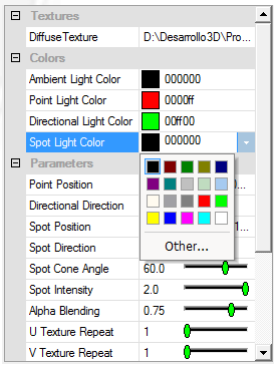
Para entender que es DXSAS veamos el siguiente ejemplo el cual muestra una definición clásica de una textura usada en un programa HLSL.

```
texture diffuseTexture;
```

Esta información no le dice mucho a una herramienta, solo sabe que es una textura, no sabe para qué es usada ni cuál es el contenido que lleva. Agreguémosle información siguiendo el estándar DXSAS.

```
texture diffuseTexture : Diffuse // Semántica
<
        string ResourceName = "default_color.dds"; // Anotación
>;
```

DXSAS permite especificar la semántica de ciertas variables globales, en este caso usando el identificador *Diffuse* le indicamos que la variable *diffuseTexture* es usada como una textura que especifica el color difuso.



Muy bien, ahora la herramienta sabe que es y para qué sirve *diffuseTexture,* pero ¿qué debe mostrar? Acá entra en juego las anotaciones. En este caso, las anotaciones permiten asignarle a la variable una textura desde un archivo. Esta información se enmarca con los signos menor y mayor. Notemos que nos permite asignar como valor inicial el contenido de un archivo externo, esto es extremadamente útil para inicializar texturas.
Además, las semánticas tienen otros propósitos. Por ejemplo, podemos indicar que el valor de una variable podrá ser controlado con el editor gráfico de propiedades del shader. En la figura de la derecha podemos apreciar un ejemplo del editor gráfico de propiedades del shader. Fijémonos que podemos modificar, a través de una interfaz gráfica, el color de las luces, su posición, dirección y otros parámetros del shader.

Entender DXSAS es muy importante para entender cómo funciona FX Composer. Además, muchos de los shaders que se encuentran en internet siguen el estándar DXSAS (en parte gracias a FX Composer), lo cual permite visualizarlos directamente y sin modificaciones sobre FX Composer.

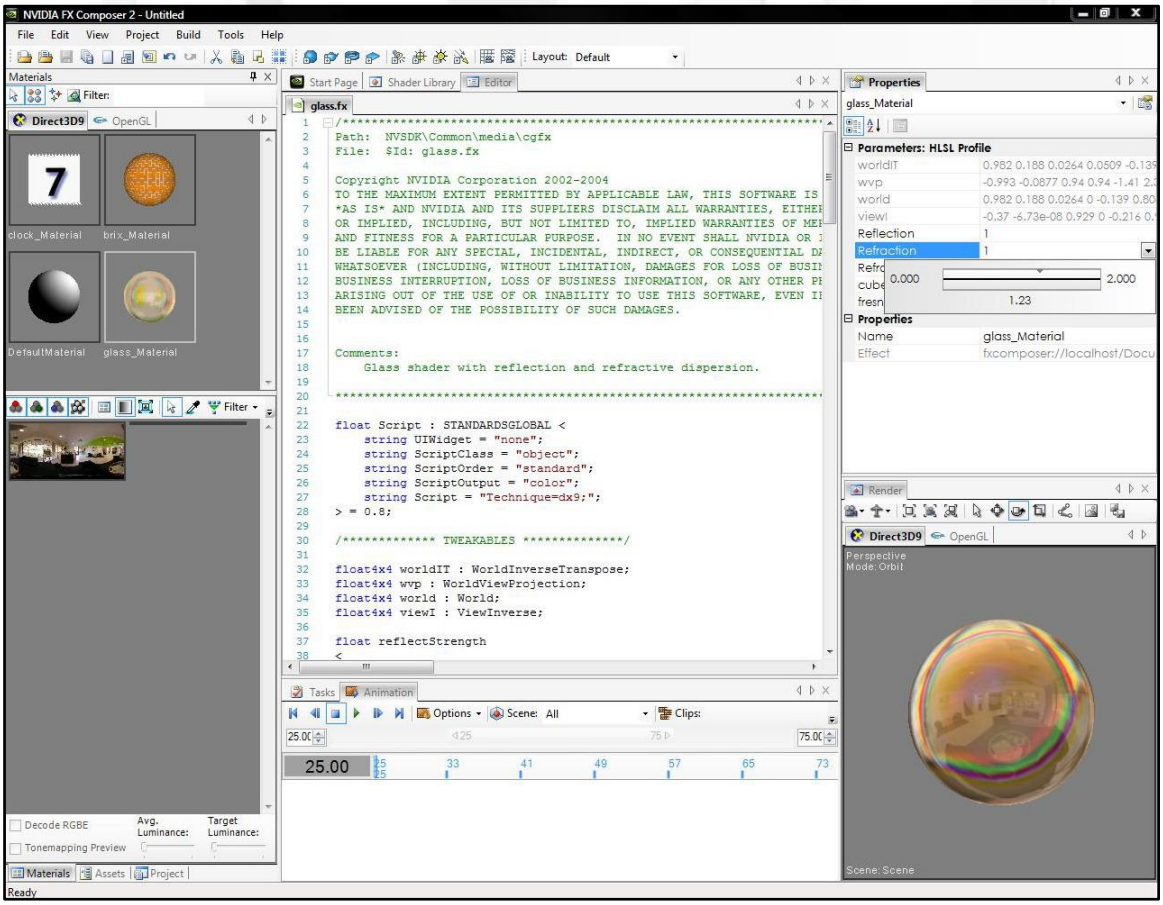
4.2.3 FX Composer 2

Posiblemente gracias a la asociación reciente de NVIDIA con SONY para el desarrollo de la GPU de la Playstation 3 es que vemos tantos cambios en FX Composer 2 con respecto a su versión anterior. Ahora disponemos de la posibilidad de desarrollar shaders en ambas APIs, OpenGL y DirectX, utilizando no solo HLSL, sino también CgFX. Además, se incorporo soporte del estándar COLLADA, el cual parece estar pisando muy fuerte en la comunidad de desarrolladores 3D.

COLLADA (**COLLA**borative **D**esign **A**ctivity) es un estándar abierto basado en XML para el intercambio de información entre varias aplicaciones 3D. El objetivo es disponer de un formato universal que permita intercambiar recursos fácilmente entre las distintas aplicaciones que se utilizan en el desarrollo de un proyecto.
COLLADA incluye entre otras cosas:

- Soporte para animación y skinning.
- COLLADA FX: intercambio de shaders, incluyendo soporte para múltiples lenguajes como Cg, GLSL y HLSL.
- COLLADA Physics: permite definir varios atributos físicos en escenas. Por ejemplo, se podría definir un material con propiedades como la fricción del mismo.

Asimismo, toda la interfaz ha sido re diseñada, dándole un aspecto más similar a aplicaciones como el Visual Studio .NET, lo cual se traduce en una mejor organización y en la incorporación de mejores controles. Incluye, por ejemplo, una página de inicio, muy al estilo de herramientas como Macromedia Studio y Visual Studio .NET.



FX Composer 2

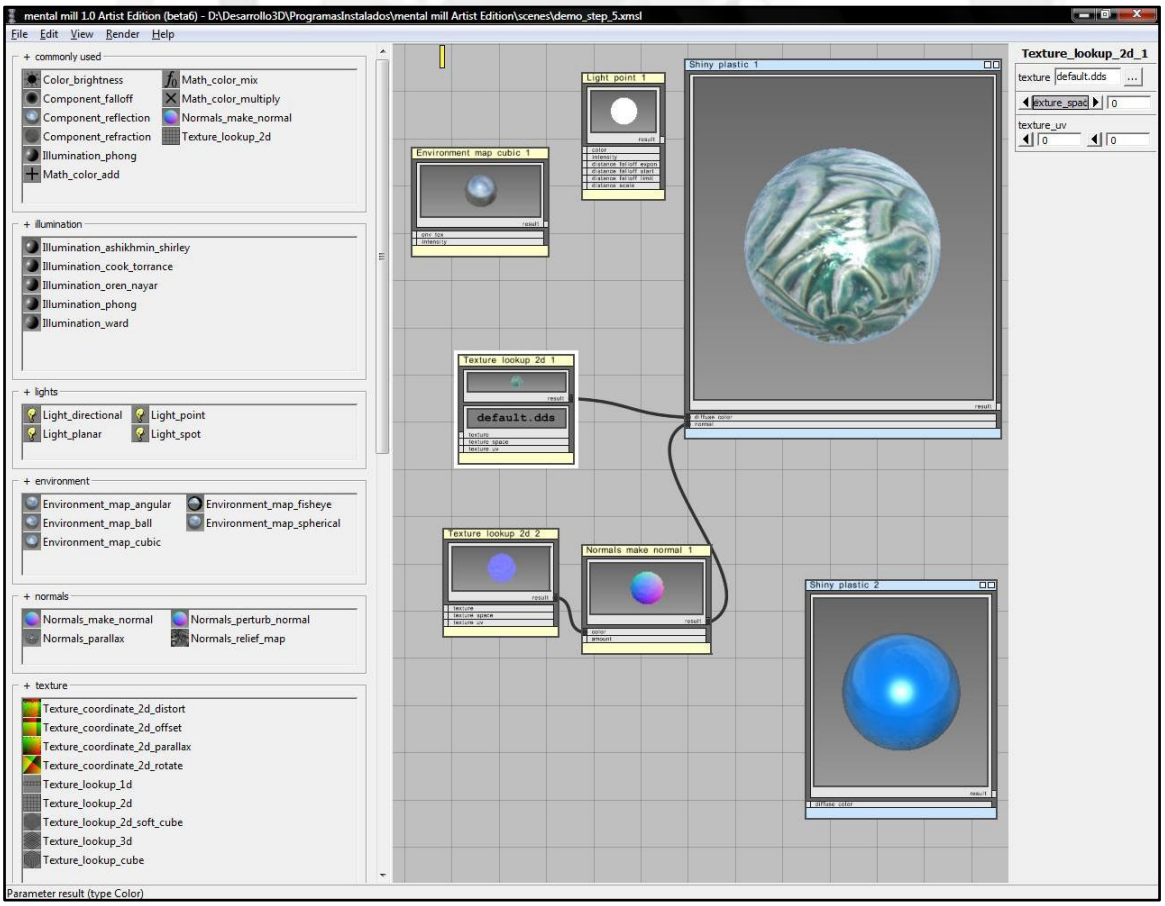
Las incorporaciones de FX Composer 2 no terminan acá, podríamos destacar varias cosas más. Detalles importantes como por ejemplo una mejor integración con la herramienta de optimización ShaderPerf 2, o detalles menores como la incorporación de distintas disposiciones (o layouts) de la interfaz.

En conclusión, se puede decir sin temor que FX Composer es una herramienta muy interesante para el desarrollo de shaders.

4.2.4 Mental Mill Artist Edition

En el paquete de FX Composer 2 también se agrega la herramienta Mental Mill Artist Edition, la cual permite desarrollar shaders a través de una interfaz gráfica. El eje central de esta herramienta es la posibilidad de conectar diferentes módulos para crear nuevos shaders.

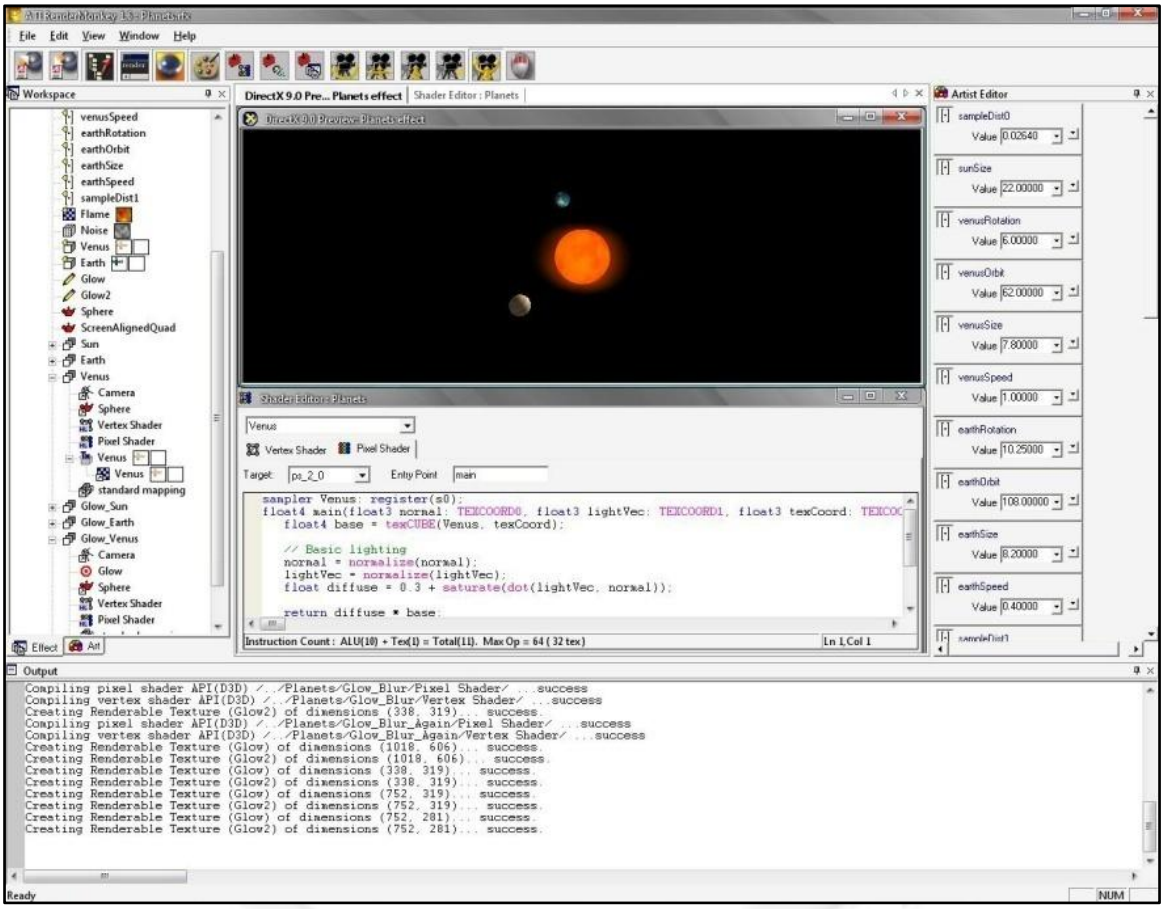
Mental Mill se lo podría considerar un enfoque adecuado para artistas que no prefieren trabajar directamente con el código. Los shaders desarrollados en Mental Mill pueden exportarse luego a CgFX o HLSL, y cargarse en FX Composer 2 para, por ejemplo, realizar optimizaciones.



Mental Mil Artist Edition

4.3 RenderMonkey

RenderMonkey, la contrapartida de ATI, aunque similar en muchos aspectos difiere en uno clave. No utiliza el effect framework, ni DXSAS. En otras palabras, el único código visible al desarrollador es el de los vertex shaders y pixel shaders. Todo lo demás son elementos representados gráficamente: pasadas, perfiles, constantes globales, texturas, etc. Además, RenderMonkey permite trabajar sobre DirectX y OpenGL, y sobre HLSL y GLSL.



RenderMonkey 1.6

Uno de los objetivos iníciales de los desarrolladores de RenderMonkey era crear una herramienta que pueda ser usada cómodamente tanto por programadores como artistas. Los programadores se beneficiarían de la organización jerárquica de los distintos elementos que componen el shader junto con una IDE bien organizada. A los artistas se les brinda una herramienta para modificar ciertos parámetros de los shaders, la cual permite modificar estos parámetros sin tocar ninguna línea de código, con el objetivo de poder probar variaciones del mismo shader. Esta última herramienta también está presente en FX Composer.

Al momento de escribir este texto, no hay noticias sobre una nueva versión de RenderMonkey que considere entre otras cosas a los geometry shaders y al shader model 4 en su totalidad. Lo cual no quiere decir que no aparezca una nueva versión de esta herramienta pronto. Y, ¿Porque no con cambios más profundos como los que vimos en FX Composer 2?

Aunque no es mi favorita, debo reconocerlo, no puedo negar que es una herramienta poderosa, con un esquema interesante para el manejo de la organización del shader, y que aún es utilizada por una gran cantidad de desarrolladores.

En conclusión, la herramienta a elegir depende mucho del gusto personal y de ciertas características particulares que puedan necesitarse. Para un desarrollador que recién está comenzando, FX Composer 2 puede verse como la opción más tentadora, dado que está muy actualiza y dispone de soporte para varias plataformas y varios formatos.

4.4 Un ejemplo simple

A continuación se mostrara un shader de ejemplo sacado de mi motor gráfico. El shader representa un material phong simple (iluminación por pixel), el cual permite influenciarse por cuatro luces, la luz ambiente, una luz punto, una luz direccional y una luz spot. El código está escrito en HLSL, utilizando FX Composer, y por lo tanto respeta la notación DXSAS.

Veremos el código por partes para entender mejor cada una de ellas. Las partes en cuestión serán:

- Variables globales (matrices de transformación, luces, superficie, texturas).
- Estructuras.
- Vertex shader.
- Pixel shader.
- Técnicas.

4.4.1 Variables globales

Debemos definir las variables globales que utilizara el shader. En general, las variables globales las suele setear la aplicación principal. Y es por esta razón que necesitamos en la herramienta de desarrollo de shaders un mecanismo cómodo y rápido para cambiar estos valores, con el objetivo de realizar distintas pruebas en tiempo real. En nuestro caso particular utilizaremos DXSAS y el editor gráfico de propiedades del shader para este propósito.

Matrices de transformación

```
////////////////////////////////////////////
/////////////// Matrices ///////////////////
////////////////////////////////////////////

float4x4 WorldIT       : WorldInverseTranspose <string UIWidget="None";>;
float4x4 WorldViewProj : WorldViewProjection   <string UIWidget="None";>;
float4x4 World         : World                 <string UIWidget="None";>;
float4x4 ViewI         : ViewInverse           <string UIWidget="None";>;
```

Primero definimos cuatro variables globales que representan a las matrices de transformación que utilizara el shader. Notemos que esta información no es pasada automáticamente por la API gráfica, debemos hacer todo el trabajo a mano. En la práctica esto no suele ser un gran problema, una buena organización del código permite solucionar todos estos "inconvenientes".
Tomemos una matriz de ejemplo para analizarla mejor:

```
    (1) float4x4 WorldViewProj : (2) WorldViewProjection   (3) <string UIWidget="None";>;
```

(1) La declaramos como una matriz de 4 x 4 componentes de punto flotante.
(2) Definimos su semántica. Esta semántica le indica al FX Composer que cargue la variable con cierta información automáticamente para propósitos de prueba. En este caso, la variable es cargada con una matriz muy útil, la *WorldViewProjection*, que surge como la matriz de mundo, multiplicada por la matriz que contiene la posición y orientación de la cámara, multiplicada por la matriz que contiene la información de proyección (cómo trabaja el lente de la cámara).
(3) Por último, se define la anotación. En este caso, le indicamos que esta variable no tiene ningún control gráfico asociado en el editor gráfico de propiedades del shader. No tiene sentido permitir cambiar estos valores.

Luces

```
////////////////////////////////////////////////
/////////////////// Luces //////////////////////
////////////////////////////////////////////////

// Ambient light //

float3 AmbientLightColor : SPECULAR <
        string UIName = "Ambient Light Color";
        string UIWidget = "Color";
> = {0.05f, 0.05f, 0.05f};

// Point light //

float3 PointLightPos : POSITION <
        string UIName = "Point Position";
        string Object = "PointLight";
        string Space = "World";
> = {500.0f, 500.0f, -500.0f};

float3 PointLightColor : SPECULAR <
        string UIName = "Point Light Color";
        string UIWidget = "Color";
> = {0.0f, 0.0f, 0.0f};

// Directional light //

float3 DirectionalLightDir : DIRECTION <
        string UIName = "Directional Direction";
        string Object = "DirectionalLight";
        string Space = "World";
> = {0.65f, -0.65f, 0.39f};

float3 DirectionalLightColor : SPECULAR <
        string UIName = "Directional Light Color";
        string UIWidget = "Color";
> = {0.0f, 0.0f, 0.0f};

// Spot light //

float3 SpotLightPos : POSITION <
        string UIName = "Spot Position";
        string Object = "SpotLight";
        string Space = "World";
> = {500.18f, -510.10f, -510.12f};

float3 SpotLightDir : DIRECTION <
        string UIName = "Spot Direction";
        string Object = "SpotLight";
        string Space = "World";
> = {0.0f, -0.91f, -0.42f};
```

```
float SpotLightCone <
    string UIWidget = "slider";
    float UIMin = 0.0;
    float UIMax = 90.5;
    float UIStep = 0.1;
    string UIName = "Spot Cone Angle";
> = 60.0;

float3 SpotLightColor : Specular <
        string UIName = "Spot Light Color";
        string Object = "SpotLight";
        string UIWidget = "Color";
> = {0.0f, 0.0f, 0.0f};

float SpotLightIntensity <
        string UIName = "Spot Intensity";
        string UIWidget = "slider";
        float UIMin = 0.0;
        float UIMax = 2;
        float UIStep = 0.1;
> = 2;
```

Definimos las variables globales concernientes a las luces. En este caso, nuestro shader utilizará cuatro luces, cada una de distinto tipo:

- Ambiente.
- Punto.
- Direccional.
- Spot.

Análogamente, tomemos una variable global de ejemplo, el color de la luz ambiente:

```
// Ambient light //

(1) float3 AmbientLightColor : (2) SPECULAR <
(3)     string UIName = "Ambient Light Color";
        string UIWidget = "Color";
> = (4) {0.05f, 0.05f, 0.05f};
```

(1) Definimos al color de la luz ambiente como un vector de tres componentes de punto flotante.
(2) Definimos su semántica, la cual le indica al FX Composer que la variable representa a un color especular. En este caso, a diferencia del anterior, FX Composer cargará el valor de la variable con lo establecido en (4), la asignación inicial de la variable.
(3) Luego, definimos la anotación. Le indicamos que la variable tendrá un control gráfico asociado en el editor gráfico de propiedades del shader. Este control se llamará "*Ambient Light Color*" y permitirá modificar el valor utilizando una herramienta de selección de color.
(4) Por último, especificamos el valor inicial de la variable. En este caso, la variable tendrá un color gris muy oscuro. El valor inicial de la variable tiene más sentido para el FX Composer que para la aplicación principal, dado que esta ultima suele asignarle el valor de la misma.

Superficie

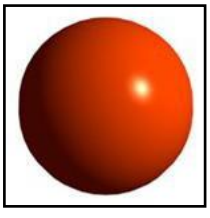
```
////////////////////////////////////////////////
/////////////// Superficie ////////////////////
////////////////////////////////////////////////

float AlphaBlending
<
    string UIWidget = "slider";
    float UIMin = 0.0;
```

```
    float UIMax = 1.0;
    float UIStep = 0.05;
    string UIName = "Alpha Blending";
> = 0.75f;

float UScale
<
    string UIWidget = "slider";
    float UIMin = 1.0;
    float UIMax = 16.0;
    float UIStep = 1.0;
    string UIName = "U Texture Repeat";
> = 1.0;

float VScale
<
    string UIWidget = "slider";
    float UIMin = 1.0;
    float UIMax = 10.0;
    float UIStep = 1.0;
    string UIName = "V Texture Repeat";
> = 1.0;

float SpecIntensity
<
    string UIWidget = "slider";
    float UIMin = 0.0;
    float UIMax = 10.0;
    float UIStep = 0.01;
    string UIName = "Specular Intensity";
> = 2.0;

float SpecExponent : SpecularPower
<
    string UIWidget = "slider";
    float UIMin = 0.0;
    float UIMax = 100.0;
    float UIStep = 0.01;
    string UIName = "Specular Exponent";
> = 30.0;
```

Definimos las variables globales que permiten ajustar algunos parámetros relacionados con la superficie del objeto, o lo que es lo mismo, el material del objeto. Por ejemplo, permitimos especificar parámetros que definen como reacciona el material ante las reflexiones especulares. Estas reflexiones se suelen simular a groso modo, considerando solamente los haces de luz provenientes de las fuentes de luz. En realidad podríamos simularlas mejor, pero necesitaríamos técnicas de Raytracing, o al menos environment mapping, lo cual normalmente no es necesario, simplemente el costo/beneficio no lo justifica.

Tomemos en cuenta que modificar los parámetros concernientes a las reflexiones especulares nos permite representar distintos materiales, algo muy importante.

El fenómeno de reflexión se produce cuando un rayo de luz rebota contra la superficie de un objeto. La **reflexión especular** no es más que un caso particular de este fenómeno, que se produce cuando el ángulo de salida del rayo es igual al ángulo de llegada, tomando como referencia la normal a la superficie en el punto de incidencia. Es decir, cuando el rayo sale rebotado con el mismo ángulo con el que incide sobre la superficie.
Lo particular de este tipo de reflexión es que al conservarse los ángulos también se conserva la disposición original de los rayos, y esto hace que se impresionen imágenes nítidas de los objetos que se encuentran alrededor de la superficie sobre la que incide la luz, tal y como ocurre en los espejos o en los metales muy pulidos.



En este shader también se define el alpha blending, o transparencia de la superficie, junto con un par de variables que permiten modificar levemente el mapeado de texturas.

Texturas

```
////////////////////////////////////////////
/////////////////// Texturas ///////////////////
////////////////////////////////////////////

texture DiffuseTexture : DIFFUSE
<
    string ResourceName = "default_color.dds";
    string ResourceType = "2D";
>;

sampler2D DiffuseSampler = sampler_state
{
        Texture = <DiffuseTexture>;
        MinFilter = Linear;
        MagFilter = Linear;
        MipFilter = Linear;
        AddressU = WRAP;
        AddressV = WRAP;
};
```

Definimos las variables globales que tiene relación con las texturas. Lo más destacable de esto es la posibilidad de referenciar a texturas almacenadas en un archivo externo, algo muy importante para el desarrollo de shaders en una herramienta. También definimos la semántica, en este caso especificamos que la textura será utilizada como textura difusa, o dicho de otra manera, le indicamos que la textura le dará el color principal a los objetos a los cuales le aplicamos este material. Además, definimos el sampler de la textura, indicando el tipo del filtrado utilizado y los modos de direccionamiento de la textura.

4.4.2 Estructuras

Es hora de definir las estructuras que especifican los parámetros de entrada y salida de los shaders. En realidad, los parámetros de entrada los podríamos haber declarado individualmente en el cuerpo de la función que calcula el shader. Esta decisión generalmente va en gustos. Personalmente considero que es más organizado definir los parámetros de entrada como estructuras. Mas teniendo en cuenta que los parámetros de salida del vertex shader son los parámetros de entrada del pixel shader.

```
////////////////////////////////////////////
/////////////// Estruturas ////////////////////
////////////////////////////////////////////

struct vertexInput {
    float3 Position   : POSITION;
    float4 UV         : TEXCOORD0;
    float4 Normal     : NORMAL;
};

struct vertexOutput {
    float4 HPosition    : POSITION;
    float2 UV           : TEXCOORD0;
    float3 WorldNormal  : TEXCOORD1;
    float3 WorldView    : TEXCOORD2;
    float3 PointLightVec: TEXCOORD3;
    float3 SpotLightVec : TEXCOORD4;
};
```

4.4.3 Vertex shader

Llego el momento de analizar el vertex shader. Es importante notar que a pesar de que este shader calcula la iluminación por pixel, ciertos cálculos referentes a la iluminación deberán hacerse en el vertex shader.

Básicamente calculáramos en el vertex shader ciertos valores necesarios para determinar la iluminación, como la normal en ese vértice o la posición de las fuentes de luz con respecto a ese vértice. Pero en vez de realizar el resto de los cálculos aquí, dejaremos que se calculen en el pixel shader. Dado que ahí tendremos estos valores interpolados por todo el polígono del cual el vértice forma parte, considerando obviamente los valores calculados en los otros dos vértices. De esta manera podremos realizar un cálculo mucho más preciso.

Desafortunadamente, escapa al alcance del libro discutir sobre como calcula el shader está iluminación. El objetivo es hablar de aspectos generales.

```
////////////////////////////////////////////
/////////////// Vertex Shader //////////////
////////////////////////////////////////////

vertexOutput VertexShader(vertexInput IN)

{

    // Definimos la estructura de salida //
    vertexOutput OUT;

    // Calculo de la normal en el mundo //
    float3 Nw = normalize(mul(IN.Normal,WorldIT).xyz);
    OUT.WorldNormal = Nw;

    // Posiciones del vértice teniendo en cuenta distintos puntos de origen //
    // Necesarias para cálculos auxiliares concernientes a las luces //
    float4 Po = float4(IN.Position.xyz,1.0);        // object coordinates
    float3 Pw = mul(Po,World).xyz;                  // world coordinates

    // Posiciones de las luces con respecto al vértice //
    OUT.PointLightVec = PointLightPos - Pw;
    OUT.SpotLightVec = SpotLightPos - Pw.xyz;

    // Coordenadas de texturas //
    OUT.UV = float2(UScale,VScale) * IN.UV.xy;

    float3 Vn = normalize(ViewI[3].xyz - Pw);       // obj coords
    OUT.WorldView = Vn;

    // Posición del vértice en la escena
    OUT.HPosition = mul(Po,WorldViewProj);

    return OUT;
}
```

Observemos que para pasar varios valores auxiliares al pixel shader utilizamos vectores con semántica TEXTCOORD. Esta es una práctica muy común debido a que muchas veces no disponemos de semánticas que reflejen el significado de estos valores. No existe ningún problema grave en realizar esto, dado que nosotros en el pixel shader le daremos la semántica que queramos. Distinto seria si el procesamiento de fragmentos se calcula utilizando el pipeline gráfico de función fija, en este caso, el GPU no sabría cómo interpretar esta información.

Sin embargo tenemos que tener cuidado en hacer esto, muchas veces nos lleva a utilizar la semántica equivoca, y esto podría provocar la reducción de la legibilidad del código. Pensemos lo siguiente: ¿Qué semántica podríamos haber utilizado para *WorldNormal* o *WorldView*?

4.4.4 Pixel shader

Por fin llegamos al pixel shader. Como vemos, aquí se realiza la mayor parte de los cálculos, algo que se da en la mayoría de los shaders.

El proceso se puede resumir de la siguiente manera. Para cada luz, calculamos como ella incide en el color difuso y en el color especular del material en ese fragmento en particular. Por último, utilizamos estos valores más las texturas involucradas (en este caso solo una) para calcular el color final del fragmento.

```
////////////////////////////////////////////
/////////////// Pixel Shader ///////////////
////////////////////////////////////////////

float4 PixelShader(vertexOutput IN) : COLOR

{

    float3 diffContrib;
    float3 specContrib;

    // La interpolación no normaliza los valores //
    float3 Nn = normalize(IN.WorldNormal);
    float3 Vn = normalize(IN.WorldView);

    // Luz punto //
    float3 Ln = normalize(IN.PointLightVec);
    float3 Hn = normalize(Vn + Ln);
    float hdn = dot(Hn,Nn);
    float ldn = dot(Ln,Nn);
    float4 litVec = lit(ldn,hdn,SpecExponent);
    diffContrib = litVec.y * PointLightColor;
    specContrib = SpecIntensity * litVec.z * diffContrib;

    // Luz direccional //
    float3 Ln3 = DirectionalLightDir;
    float3 Hn3 = normalize(Vn + Ln3);
    float hdn3 = dot(Hn3,Nn);
    float ldn3 = dot(Ln3,Nn);
    float4 litVec3 = lit(ldn3,hdn3,SpecExponent);
    diffContrib += litVec3.y * DirectionalLightColor;
    specContrib += ((litVec3.z * SpecIntensity) * litVec3.y * DirectionalLightColor);

    // Luz spot //
    float3 Ln2 = normalize(IN.SpotLightVec);
    float CosSpotAng = cos(SpotLightCone*(float)(3.141592/180.0));
    float dl = dot(SpotLightDir,Ln2);
    dl = ((dl-CosSpotAng)/(((float)1.0)-CosSpotAng));
    if (dl>0)
    {
        float ldn2 = dot(Ln2,Nn);
        float3 Hn2 = normalize(Vn + Ln2);
        float hdn2 = dot(Hn2,Nn);
        float4 litVec2 = lit(ldn2,hdn2,SpecExponent);
        ldn = litVec2.y * SpotLightIntensity;
                ldn *= dl;
                diffContrib += ldn * SpotLightColor;
        specContrib += ((ldn * litVec2.z * SpecIntensity) * SpotLightColor);
    }

    // Calculamos el color final del pixel //
    // Considerando la información anteriormente calcula más la textura utilizada //
    float3 colorTex = tex2D(DiffuseSampler, IN.UV).xyz;
    float3 result = colorTex * (diffContrib + AmbientLightColor) + specContrib;
    return float4(result.xyz, AlphaBlending);
}
```

Desafortunadamente, como dijimos antes, escapa al alcance de este texto discutir en detalle cómo se calculan los distintos pasos.

4.4.5 Técnicas

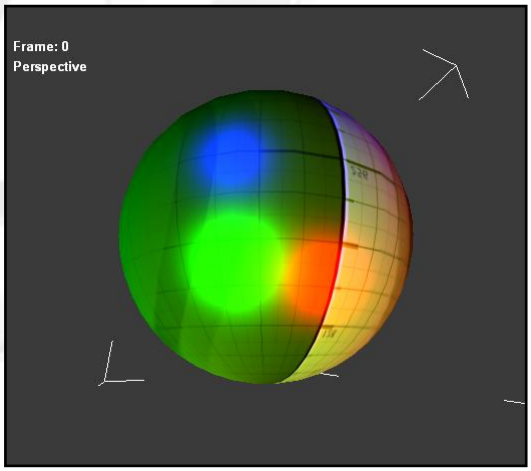
Finalizando el shader se encuentran las técnicas, en nuestro caso particular solo una. Está específica que la función que calculara el vertex shader será la función *VertexShader* utilizando la especificación del shader model 2.0. Y que la función que calculara el pixel shader será la función *PixelShader* utilizando la especificación del shader model 3.0.

Además, se le indica al pipeline gráfico el modo de culling, el cual evita procesar caras innecesarias, elevando considerablemente el rendimiento. También se le indica otros parámetros relacionados con el Z Buffer.

```
////////////////////////////////////////////
/////////////// Techniques /////////////////
////////////////////////////////////////////

technique ModelShader30
{
        pass P0
        {
                VertexShader = compile vs_2_0 VertexShader();
                ZEnable = true;
                ZWriteEnable = true;
                CullMode = None;  // Para FX Composer
                //CullMode = CCW; // Para el motor (mayor rendimiento). DirectX y FX Composer
                                  // trabajan de manera contraria en este apartado)
                PixelShader = compile ps_3_0 PixelShader();
        }
}
```

4.4.6 Resultado final

Este ejemplo nos permitió comprender la complejidad y magnitud de un shader real. Al principio puede resultar algo intimidante, y no es para menos, mucha de la simplicidad del pipeline gráfico de función fija se pierde.
Pero la ganancia es increíble, más aun si consideramos, por ejemplo, que con algunas pocas líneas más se puede crear un material que utiliza parallax mapping.
Las formulas para el cálculo de iluminación, como para cualquier otra cosa que se quiera lograr con un shader no son tan difíciles de seguir si se utiliza una buena base como documentación. Existen libros que explican de manera sencilla y entendible como lograr gran cantidad de los shader actuales. Shaders que solemos ver todos los días en aplicaciones reales.



APENDICE
NIVEL DE DETALLE

GTA IV

A1 Introducción

Cada año las escenas son más complejas. Utilizan sistemas de iluminación más realistas, incrementan la cantidad de efectos, e incluyen mejores simulaciones físicas. Paralelo a esto, como es de imaginar, se utilizan modelos con mayor poligonaje y texturas con mayor resolución. Todo lo cual agrega mayor inmersión y realismo, y por supuesto mayor costo computacional.

Los recursos son y seguirán siendo limitados y muy preciados, al menos en un futuro cercano. Es por esto que se ha invertido un gran esfuerzo en tratar de reducir la complejidad de las escenas con el propósito de reducir el costo computacional y al mismo tiempo tratando de mantener una calidad similar.

Veámoslo de este modo: ¿Para qué procesar gran cantidad de vértices o procesar texturas con gran resolución para objetos que ocuparan algunos pocos pixeles en la pantalla?

Este campo de la computación gráfica que trata de balancear complejidad con performance se lo denomina **nivel de detalle** o más comúnmente **LOD** por “Level Of Detail”.

No solo existen técnicas de nivel de detalle geométrico o de texturas, también existen técnicas LOD para shaders o cualquier apartado que permita reducir su calidad en beneficio de la performance.

A1.1 Nivel de detalle geométrico

LOD geométrico fue el primer tipo de LOD en aparecer y es el que más investigaciones tiene. Data desde 1976, año en el cual James Clark presento el primer artículo sobre el tema. En el transcurso del tiempo y a medida que los gráficos 3D fueron tomando mayor importancia fueron apareciendo nuevas investigaciones. Al principio su utilidad recaía en software como simuladores de vuelo, los cuales hacían uso intensivo de LOD. Recién a principios de los 90s fue cuando el tema entro en furor y empezó a aplicarse a una gran rama de aplicaciones.

> Es sumamente normal que al LOD geométrico se lo denomine solo como LOD dada su alta popularidad. Por simplicidad muchas veces me referiré al LOD geométrico como LOD, dado que no es difícil discernir a que me estoy refiriendo dado el contexto.

En el 2001 apareció en el mercado el GTA III. Este fue el primer paso hacia una nueva generación de juegos que incluyen mundo abierto. Desarrollar un mundo abierto tiene varios retos, el tamaño del terreno a cubrir, las grandes distancias de renderizado y el dinamismo del mismo. Dinamismo que podría complicar predecir el número de polígonos en una escena en determinado momento. Otra razón más por la que necesitamos tener una herramienta para controlar la complejidad de una escena.

> Para no tener conflictos históricos, se debe aclarar que en 1998 el mismo equipo de desarrolladores del GTA III realizó el Body Harvest, el cual es para muchos el primer juego de mundo abierto, aunque careció de gran popularidad.

Imágenes del GTA: San Andreas. Se puede apreciar la magnitud del mundo creado

Controlar el nivel de detalle también permite ejecutar software en una gran variedad de computadoras, cada una con distintos niveles de recursos. Problema de vital importancia para un desarrollador de juegos para PC que debe soportar no solo la generación actual de GPUs, sino también las anteriores.

Ahora bien, está claro que LOD geométrico es un tema de gran importancia, pero es necesario destacar que también puede influir a desarrolladores de shaders. Esto se debe a que cada vértice de cada objeto es siempre procesado por la etapa de transformación de vértices, no importa su ubicación o ningún otro factor. Por lo que reducir la cantidad de vértices a procesar también nos permitirá tener una mayor cantidad de shaders y de mejor calidad.

Más aún si consideramos la arquitectura actual de las GPUs. En las nuevas GPUs, el procesamiento de shaders se distribuye a los distintos stream processors los cuales pueden calcular Vertex shaders, Pixel Shaders y Geometry shaders según se necesite. Permitiendo balancear los recursos y aumentando el desempeño global. Por lo tanto, si reducimos la cantidad de procesamiento a nivel de vértice tendremos una mayor cantidad de stream processors disponibles para realizar cálculos a nivel de pixel y/o geometría.

Existen varias técnicas para lograr LOD geométrico, algunas de las cuales veremos en las próximas secciones. La idea básica de las técnicas que veremos es tener distintas versiones del modelo geométrico cada una con diferentes niveles de detalle y usar la versión que en ese momento de los mejores resultados considerando calidad final y rendimiento.

A1.1.1 LOD discreto

Es el esquema más simple, propuesto por James Clark en 1976. Esta técnica crea múltiples versiones pre procesadas de cada objeto, cada una con diferentes niveles de detalle. En tiempo de ejecución se selecciona el nivel de detalle apropiado para representar al objeto en ese instante. Los objetos distantes usaran versiones con poco detalle, el resto de los objetos usaran versiones cada vez más detalladas dependiendo, por ejemplo, de su distancia relativa a la cámara o alguna otra métrica.

Debido a que el LOD es pre calculado, el proceso de simplificación no puede predecir desde que dirección será visto el objeto. Por lo tanto, la simplificación típicamente reduce el detalle uniformemente a través del objeto. Por esta razón es que muchas veces se refiere al LOD discreto como LOD isotrópico o LOD independiente de la vista (view independent).

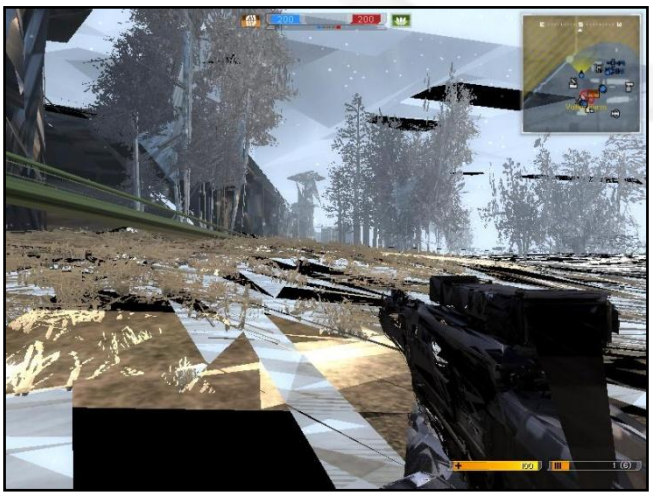
60000 polígonos 6000 polígonos 600 polígonos 60 polígonos

LOD discreto con cuatro versiones del modelo

LOD discreto tiene sus ventajas:

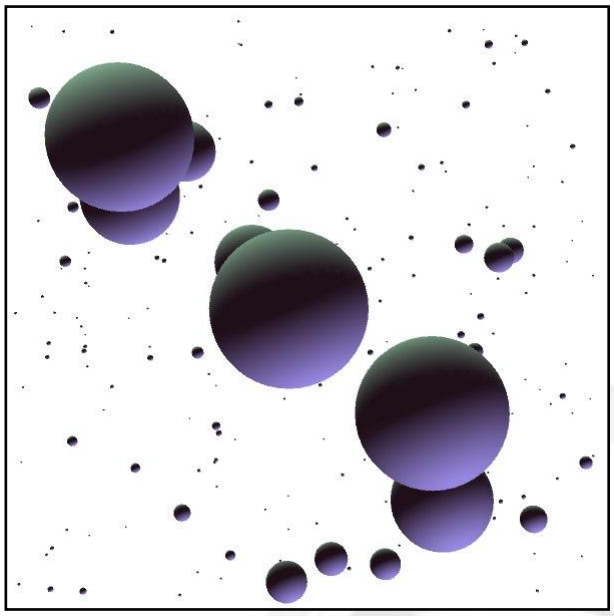
- No es importante cuanto tardara el algoritmo de simplificación, dado que se realizará solo una vez y fuera del entorno de ejecución. Además, en tiempo de ejecución simplemente se necesitara elegir que LOD renderizar por cada objeto.
- Dependiendo del entorno en donde se ejecutara, los desarrolladores podrán convertir los modelos para usar características tales como triangle strips, display lists y vertex arrays. Lo cual normalmente implica un mejor tiempo de renderización en comparación al renderizado de los LODs como una lista de polígonos desordenados.
- Permite crear los distintos niveles de detalle de forma manual, brindando control total sobre el proceso de simplificación. No solo permitiendo controlar la calidad del modelo, sino también asegurando que no existan artifacts tales como un mapeo de texturas incorrecto.

Artifacts, o artefactos en castellano, son distorsiones en una imagen o sonido causado por limitaciones o mal funcionamiento en hardware o en software.

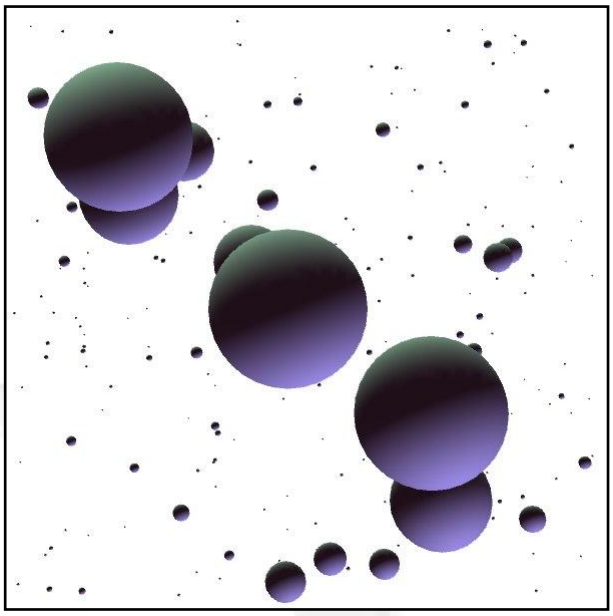


Artifacts generados por funcionamiento errático en el hardware

Artifacts producidos en el mapeo de texturas



Escena renderizada a fuerza bruta
Vértices = 2.328.480
Tiempo de renderización = 27.27 ms



Escena renderizada usando LOD discreto
Vértices = 109.440
Tiempo de renderización = 1.29 ms

A pesar de ser una comparativa abstracta podemos apreciar la ganancia del uso de LOD

LOD discreto ha sido históricamente el esquema más utilizado. Sin embargo, dista mucho de ser perfecto. Una de sus principales desventajas se ve en las transiciones de un nivel de detalle a otro. Debido a que se necesita cambiar de una malla a otra en un momento específico es probable que se note la transición, aún cuando las dos mallas tengan buena calidad. Para reducir estos artifacts llamados poppings existen técnicas para fundir ambos modelos, tales como alpha blending y geomorphing, pero aun así los resultados podrían producir algunos problemas visuales. Otra forma de solucionar este problema es utilizar un esquema de LOD continuo.

- **Alpha blending**: esta técnica define un rango de transición entre dos niveles de detalles. Llegado a este rango, en vez de reemplazarse un LOD por el otro, se renderizan ambos niveles de detalles con distintos valores alpha de forma tal que la suma de ambos valores alpha de siempre 1. Supongamos que el rango va de A a B, si se está en el extremo A el valor alpha del primer LOD será 1 y el del segundo LOD será 0, en cambio si se está entre A y B el valor alpha de ambos será 0.5. Un problema de este esquema es que necesita renderizar dos modelos cuando se encuentra en la etapa de transición, lo cual implica que para que sea implementable en la practica el rango de transición debe ser corto. Otro problema es la aparición de artifacts que surgen al renderizar los niveles de detalle con valor alpha distinto de 1. Afortunadamente existe en muchas plataformas soporte en hardware para el LOD con alpha blending que ayudan a reducir estos artifacts.
- **Geomorphing**: esta técnica "simplifica" nuestra malla en tiempo real, normalmente uniendo vértices unidos por la misma arista. Cada vez que se unen dos vértices, se eliminan tres aristas y dos triángulos. Si aplicamos este método con cuidado, podemos eliminar completamente el efecto de "popping", haciendo que los vértices se aproximen poco a poco, a lo largo de la arista, en vez de un salto. Sin duda, la dificultad de este algoritmo radica en saber qué vértices "colapsar". Incluso permite eliminar el efecto de "popping" si en vez de colapsar los vértices de golpe, hacemos que se vayan aproximando poco a poco.

Popping en Half Life 2. Notar la transición de modelos en el camión.
Este efecto se suele notar más en movimiento

A1.1.2 LOD continúo

Este esquema surgió en el año 1996, 20 años más tarde que el esquema de LOD discreto. En LOD continúo el sistema de simplificación crea una estructura de datos que codifica un espectro continuo de detalle. Luego, en tiempo de ejecución, el nivel de detalle deseado es extraído desde esta estructura.

> Muchas veces podemos encontrar al LOD continúo con el nombre de progressive LOD, debido a la estructura de datos progressive mesh creada por Hugues Hoppe.

Una gran ventaja de este esquema es su mejor granularidad, dado que el nivel de detalle para cada objeto se especifica exactamente en vez de seleccionarlo desde unas pocas opciones precalculadas. En pocas palabras, no se utilizan más polígonos que los necesarios. Esto permite usar esos polígonos ahorrados para renderizar otros objetos, que a su vez usaran solo la cantidad de polígonos necesarios para el nivel de detalle deseado, liberando más polígonos para otros objetos, y así siguiendo. Mejor granularidad lleva a mejor uso de los recursos y mejor fidelidad general.

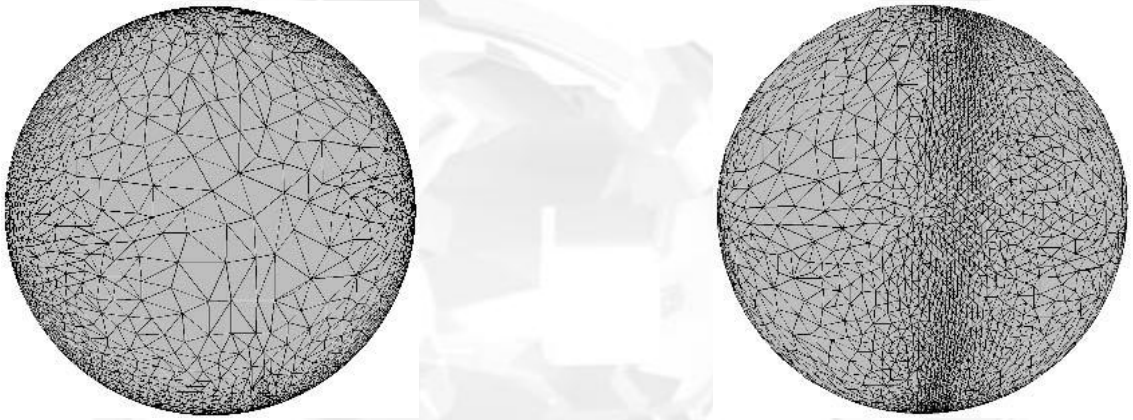
LOD continuo también permite streaming de modelos poligonales, en el cual a un modelo base simple le siguen una sucesión de refinamientos que serán integrados dinámicamente. Cuando se deben cargar modelos de gran poligonaje desde un disco o sobre una red, LOD continúo provee renderizado progresivo y carga interrumpible, muchas veces propiedades muy deseadas.

LOD continuo impone un costo en procesamiento y memoria requerida, razón que limita su uso. A pesar de esto podemos ver su uso de ya hace tiempo. Por ejemplo, el Unreal engine 1 (1998) usa LOD continuo para manejar el nivel de detalle de los personajes.

A1.1.3 LOD dependiente de la vista

Surgido en 1997, LOD dependiente de la vista extiende el concepto de lo LOD continúo, dado que usa criterios de simplificación dependientes de la vista para seleccionar el nivel de detalle más apropiado para la posición actual de la cámara. De esta forma LOD dependiente de la vista es anisotrópico. Por ejemplo, partes cercanas de un objeto tendrán mayor resolución que partes distantes, o la silueta del objeto puede mostrar mayor resolución que las regiones internas. Esto nos lleva a una mejor granularidad: el detalle se encuentra en donde más se necesita. Esto a su vez nos lleva a mejor fidelidad para un conteo de polígonos dado, optimizando la distribución de los polígonos.

Con este esquema, modelos muy complejos que representan objetos físicamente muy grandes, como terrenos, a menudo no pueden ser correctamente simplificados sin técnicas dependientes de la vista. Los esquemas anteriores no nos sirven dado que la posición de la cámara típicamente se encuentra muy cercana a una parte del terreno y distante de otras partes, así un alto nivel de detalle proveerá calidad a una inaceptable tasa de cuadros por segundo, mientras que un bajo nivel de detalle proveerá buenos cuadros por segundo pero al precio de una terrible calidad.



Una esfera simplificada con preservación de silueta. La primeara imagen muestra la esfera renderizada desde la posición de la cámara, mientras que la segunda desde el costado.

El procesamiento extra requerido para evaluar, simplificar y refinar el modelo en tiempo real, y la memoria extra requerida por las estructuras de datos, limitan su uso en la práctica. Los desarrolladores de juegos se reusan a adoptar este esquema excepto en situaciones que casi lo requieren, tales como renderizado de terreno.

A1.1.4 Otros esquemas

Existen muchos más esquemas de LOD geométrico que los anteriores nombrados, algunos de los cuales son muy usados. Los siguientes esquemas de LOD geométrico generalmente se usan en conjunto con los anteriores para mejorar aún más el rendimiento.

Screen LOD: es la técnica más sencilla de LOD y una que casi siempre se usa. Cuando un objeto se aleja a cierta distancia de la cámara, deja de dibujarse. No obstante, se produce abruptamente el efecto de "popping" cuando el objeto sale o entra de la escena, transición en la cual aparece de repente.

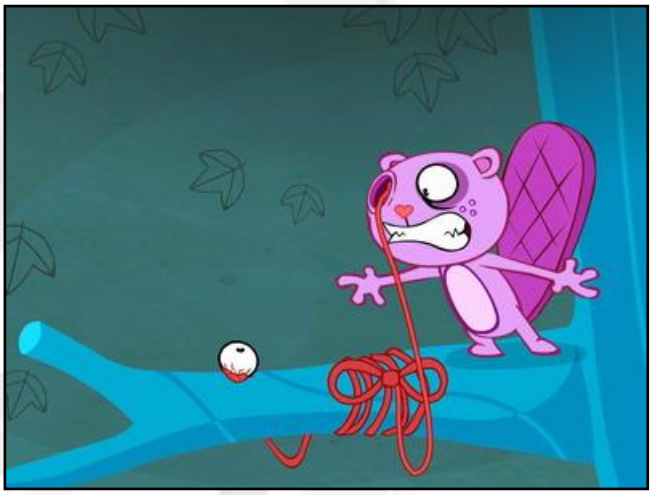
Alpha LOD: actúa como la anterior, salvo que en vez de descartar el modelo directamente, va reduciendo su valor alpha, hasta que el modelo se hace invisible, momento en que deja de dibujarse en pantalla.

En la siguiente sección se tratara otro esquema muy utilizado en la actualidad, los imposters.

A1.1.5 Imposters

En computación gráfica, **sprite** es una imagen o animación bidimensional que es integrada en una escena.
Actualmente a las sprites se las suele mapear en un plano especial. A diferencia del mapeo de texturas, este plano está siempre perpendicular a la cámara. La imagen puede ser escalada para simular perspectiva, puede ser rotada bidimensionalmente, puede solaparse con otros objetos y puede ocluir otros objetos, pero solo puede ser vista desde un mismo ángulo. Este método de renderización se denomina **billboarding** y a las sprites renderizadas de esta manera se las suele llamar **billboards**.

The Legend of Zelda: The Wind Waker hace uso de billboards para renderizar objetos como plantaciones y también para generar efectos de partículas.



Las animaciones realizadas en flash es un ejemplo clásico del uso de sprites

Un **imposter** es una técnica de simplificación que reemplaza un objeto geométrico con un billboard que es texturizado con la renderización del objeto. Muchas veces se usan imposters como último nivel de detalle de un objeto. De esta manera, podemos ver a los imposters como la máxima posibilidad de simplificación, en la que la geometría se reemplaza por una simple imagen. Muchos juegos también usan una combinación de partículas y billboards para simular efectos volumétricos como explosiones o volúmenes de luz.

Dado que los billboards son imágenes en un plano, estos deben tener el canal alpha para marcar la silueta del objeto. Además, normalmente los billboards contienen la renderización del objeto utilizando una iluminación brillosa, lo cual implica en muchos casos que deben ser oscurecidos para simular una correcta iluminación. Desafortunadamente, la iluminación nunca es tan buena como la que se tiene al usar el verdadero objeto geométrico, dado que el oscurecimiento es uniforme en toda la imagen.

Una forma de implementar imposters involucra el prerenderizado de una imagen o conjunto de imágenes que representan al objeto. Los principales desafíos se encuentran cuando lidiamos con los cambios de apariencia del objeto debido a cambios en el punto de vista de la cámara y también con las animaciones de los objetos.

Podemos simplificar el problema del cambio del punto de vista renderizando al objeto entre cuatro y dieciséis veces alrededor del eje Y. Lo cual es adecuado si usamos imposters principalmente para renderizado de objetos distantes y verticales como árboles en un terreno. En tiempo de renderizado el motor determina el ángulo entre la cámara y el objeto y selecciona la textura correcta para usar como imposter. Si el visor cambia de posición de manera que ve al objeto desde un ángulo significativamente alto o bajo, esto es desde arriba o desde abajo, debemos utilizar la versión geométrica. Por supuesto que es posible expandir este esquema a tres ejes.

La animación del objeto presenta un gran problema. El costo de renderizar múltiples ángulos y múltiples cuadros de la animación hacen al esquema prohibitivo en muchos casos. Por esta razón, se suele optar cuando es posible por un conjunto de animaciones restrictivas para imposters, o restringir ciertas animaciones a ciertos ángulos fijos.

Las texturas 3D o volume textures proveen un tercer eje de información que puede ser usado para almacenar múltiples ángulos para un imposter. Pero más importante aún, estas texturas se pueden beneficiar por el mezclado (blending) entre cuadros, el cual es asistido por hardware. En tiempo de ejecución, en vez de seleccionar un cuadro basado en el ángulo de visión calculamos una coordenada de textura w que mezclara linealmente los dos cuadros más cercanos en la textura 3D. Además, si renderizamos varios imposters del mismo tipo podemos evitar cambios en los registros de estado de la textura, cambios que serían requeridos si cada cuadro está almacenado en una textura 2D. Por último, también nos podemos beneficiar por las técnicas de compresión de las texturas 3D.

Otra técnica que podemos utilizar consiste en renderizar a una textura en tiempo real una versión del modelo con nivel de detalle relativamente bajo. Textura que reusaremos un número de cuadros. Este método tiene la habilidad de generar una imagen del modelo sin restringir el ángulo de visión o la animación del mismo. También usa menos memoria, dado que no dedica almacenamiento a sprites que no son usadas activamente. Desafortunadamente, este esquema solo puede incrementar el rendimiento si la imagen resultante es usada en varios cuadros, lo cual no siempre es posible.

Imagen del CryEngine 2. Los árboles alejados se renderizan usando billboards

El uso de billboards como imposters para representar el menor nivel de detalle en árboles o algún otro objeto que se encuentre en un terreno es uno de los tantos ejemplos que marca la importancia de los imposters. SpeedTree y Bionatics natFX son dos herramientas populares utilizadas para generar en tiempo real árboles y plantas. SpeedTree, por ejemplo, se usa en una gran cantidad de juegos, entre los cuales podemos nombrar títulos tales como The Elder Scrolls IV: Oblivion y Project Gotham Racing 3.

Estas herramientas nos brindan un gran abanico de posibilidades para controlar de manera precisa el nivel de detalle. Hacen uso intensivo de billboards aplicando técnicas muy avanzadas, como por ejemplo los esquemas híbridos. En las siguientes imágenes se ve como parte del árbol es renderizado usando geometría y otra parte es renderizado usando imposters. Con estos esquemas híbridos se puede lograr un mejor balance entre prestación y realismo.

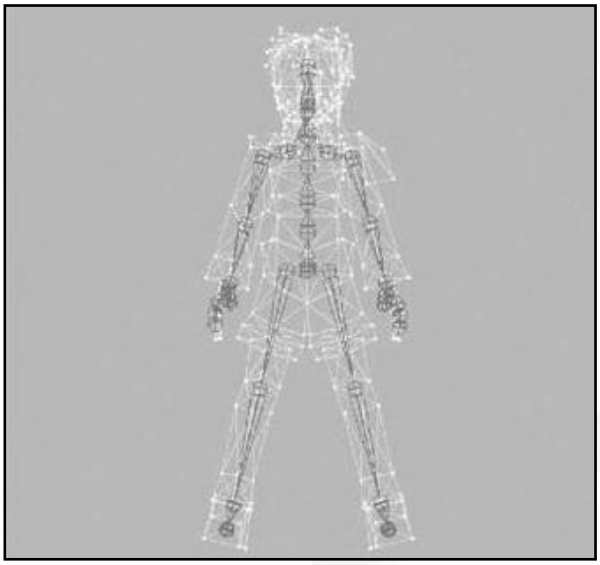
Esquema híbrido que representa al tronco por medio de geométrica y a las hojas por medio de billboards

Otra forma de esquema híbrido, en la cual la parte del árbol que es más visible por la cámara se renderiza usando la versión geométrica y el resto del árbol usando un billboard

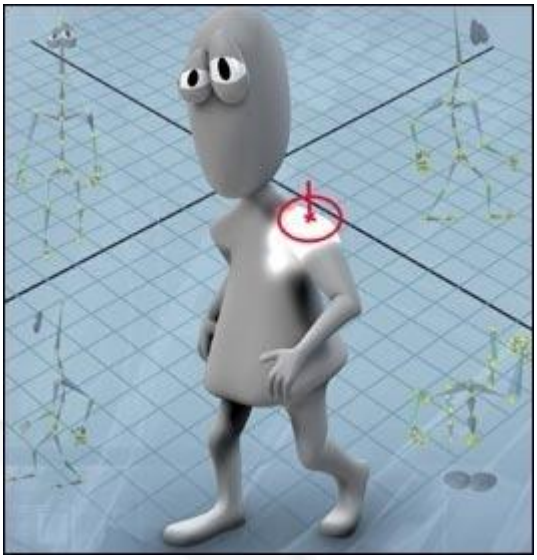
Existe otra clase más de imposters, los **imposters geométricos**, los cuales son usados para reemplazar modelos con sistemas de animación complejos por objetos rígidos. Esto tiene gran importancia dado que los sistemas de animación con esqueletos y skinning son costosos de calcular en cada cuadro.

El uso de imposters geométricos tiene sus ventajas a comparación de los imposters convencionales, dado que la silueta del objeto será siempre la correcta sin importar el ángulo de visión y sin necesidad de almacenar múltiples versiones. Además, un imposter geométrico se puede iluminar correctamente.

El **esqueleto** en un personaje 3D tiene una correlación directa con el esqueleto humano. Consiste de huesos y articulaciones, y puede ser usado como un mecanismo de control para deformar sus mallas asociadas.
Un esqueleto, de hecho, está compuesto de nodos nulos, usualmente llamados dummy nodes o grouping nodes, los cuales no son renderizables y que en conjunto forman una jerarquía. Esta jerarquía permite pasar transformaciones de un nodo a sus hijos o de un nodo a sus padres, según se use cinemática directa o inversa respectivamente.
Skinning es una técnica para crear deformaciones suaves, eliminando el efecto de “muñeco de juguete articulado”. Esencialmente el skinning es la relación entre los vértices de una malla y los huesos del esqueleto, y cómo afectarán las transformaciones de cada hueso a la posición de los vértices de la malla. Al usar skinning vinculamos el esqueleto a una única malla y en cada vértice de esta malla se le indicara por medio de pesos cuanto afecta el movimiento de un hueso a ese vértice.



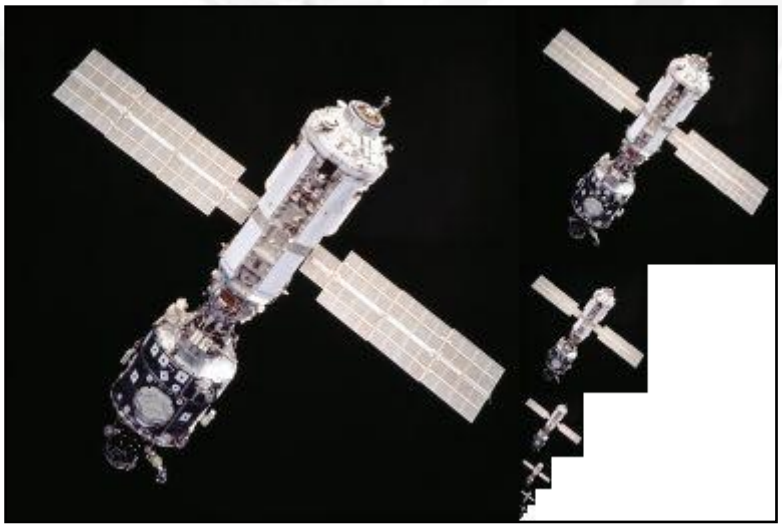
Una malla simple asociada a un esqueleto



Con skinning las deformaciones en las articulaciones son suaves y realistas.

A1.2 LOD en texturas: Mipmapping

Los **mipmaps** son colecciones de imágenes que acompañan a una textura principal para aumentar la velocidad de renderizado y reducir artifacts. Cada imagen del conjunto mipmap es una versión de la textura principal, pero a menor nivel de detalle. Aunque la textura principal se seguirá usando cuando la posición de la cámara es suficiente para renderizarla en detalle completo, el renderizador intercambiara a una imagen más adecuada (o de hecho, la interpolación de los dos niveles más cercanos) cuando la textura es vista desde cierta distancia, o vista en tamaño pequeño.



Ejemplo de mipmap. La imagen principal, a la izquierda, va acompañada de copias filtradas reducidas.

Un **texel** es la unidad mínima de una textura aplicada a una superficie. Proviene del inglés, texture element o también texture pixel. De forma práctica lo podríamos ver como un pixel de una textura.

La velocidad de renderización se incrementa dado que el número de texels procesados es mucho menor que al usar la textura original.

Si no usamos mipmaps, cuando la textura es vista de manera que varios texels caen en el espacio ocupado por un pixel podríamos tener artifacts. Dado que para calcular correctamente el color de ese pixel deberíamos combinar la contribución de todos los texels que caen en ese pixel.

En cambio, si usamos mipmap, con seleccionar la versión de la textura que tenga el tamaño adecuado en texels (relativo al tamaño de pixeles) tendríamos el cálculo automáticamente resuelto. De esta manera eliminamos artifacts a bajo costo.

Desafortunadamente, la distribución de texels en pixeles no siempre es perfecta. Dado este caso, nos encontramos con ocho texels para elegir:

- Los cuatros texels más cercanos de la textura en la cual los texels son apenas más grandes que el pixel actual.
- Los cuatros texels más cercanos de la textura en la cual los texels son apenas más chicos que el pixel actual.

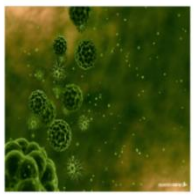
Si estamos haciendo una imagen estática, podemos elegir el texel más cercano y listo. Sin embargo, para imágenes en movimiento, este resultado nuevamente produciría aliasing dado que un pequeño movimiento en la cámara podría cambiar bruscamente de un texel a otro. Para mejorar el resultado, debemos tomar más de un texel y combinar sus valores.

De esta manera, surgen tres filtrados:

- **Filtrado lineal**: para ambos mipmaps se elije el texel más cercano y se los combina.
- **Filtrado bilineal**: elije los cuatro texels del mipmap más cercano y combina sus valores.
- **Filtrado trilineal**: elije los ocho texels y combina sus valores.

Debemos tener en cuenta que la combinación de valores no es un simple promediado. Se usa un sistema de pesos, en el cual, cuanto más cercano esta el pixel del texel, mayor será la contribución del mismo.

El filtrado trilineal es el mejor esquema de los tres expuestos, pero requiere tomar 8 texels como muestra. Desafortunadamente, el filtrado trilineal no es perfecto. Trabaja bien con polígonos que están perpendiculares a la cámara pero no con polígonos que se encuentran de manera oblicua. Para evitar este tipo de aliasing necesitamos el filtrado anisotrópico.



Textura vista de forma perpendicular

Textura vista de forma oblicua

Supongamos que vemos a una textura como en el caso de la segunda figura. En este caso la textura que mapearemos esta comprimida varias veces en dirección vertical. La textura ideal necesitaría mucha más información horizontal que vertical. Aquí es donde surge el problema, dado que esa información horizontal no es considerada ya que los mipmaps se escalan uniformemente en ambos ejes.

El **filtrado anisotrópico** toma en consideración más información dependiendo de la perspectiva en que se mire la textura, en nuestro caso tomaría más información horizontal.

Este tipo de filtrado da como resultado imágenes más claras y definidas en los puntos lejanos y con inclinación. El piso, por ejemplo, es normalmente uno de los lugares donde más se nota la presencia de filtrado anisotrópico.

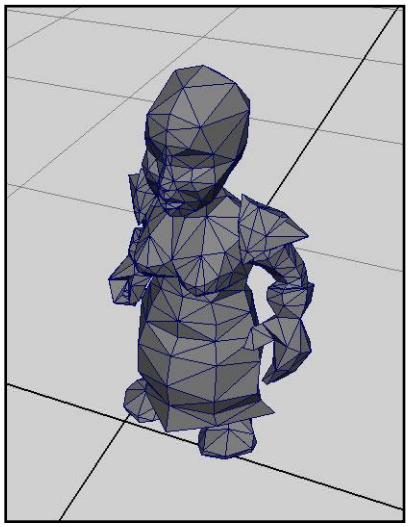
El filtrado anisotrópico, al igual que el trilineal, toma texels de los dos mipmaps más cercanos. El de menor calidad toma 16 texels como muestra para cada pixel, y se lo conoce como anisotrópico 2X. Cada versión de mejor calidad va tomando el doble de texels que su predecesor. De esta manera, por ejemplo, un esquema anisotrópico 16X utiliza 128 texels de muestra. Por supuesto que más muestras implican más calidad, pero a más uso del ancho de banda de memoria, lo cual afecta considerablemente el rendimiento, aún si la unidad que calcula el filtrado no afecta al resto de la GPU a nivel desempeño.

Usualmente, los motores nos permiten elegir entre el filtrado bilineal, el trilineal y varias opciones del anisotrópico, dejando de lado el filtrado lineal. Las opciones del anisotrópico normalmente varían entre 2X a 16X.

Bilineal Trilineal Anisotrópico 4X Anisotrópico 16X

Notemos que si la textura es vista de muy cerca, no podremos evitar con mipmaps los artifacts generados. Dado que para evitarlos necesitamos una versión de mayor resolución que la textura original.

Antes de terminar este tema, se merece la pena mencionar otra técnica referente a texturas, la **composición de texturas**. Su objetivo es crear una textura maestra que contenga todas las texturas que usa un modelo o, en algunos casos, conjunto de modelos. Esto reduce el número de cambios de estado (tanto en los registros de estado de textura y en los de shaders) requeridos para renderizar el modelo. También permite tandas de renderizado más largas, dado que solo caras con propiedades idénticas pueden renderizarse en una sola tanda.


Objeto a renderizar

+


Textura compuesta

=


Resultado final

A1.3 LOD en shaders

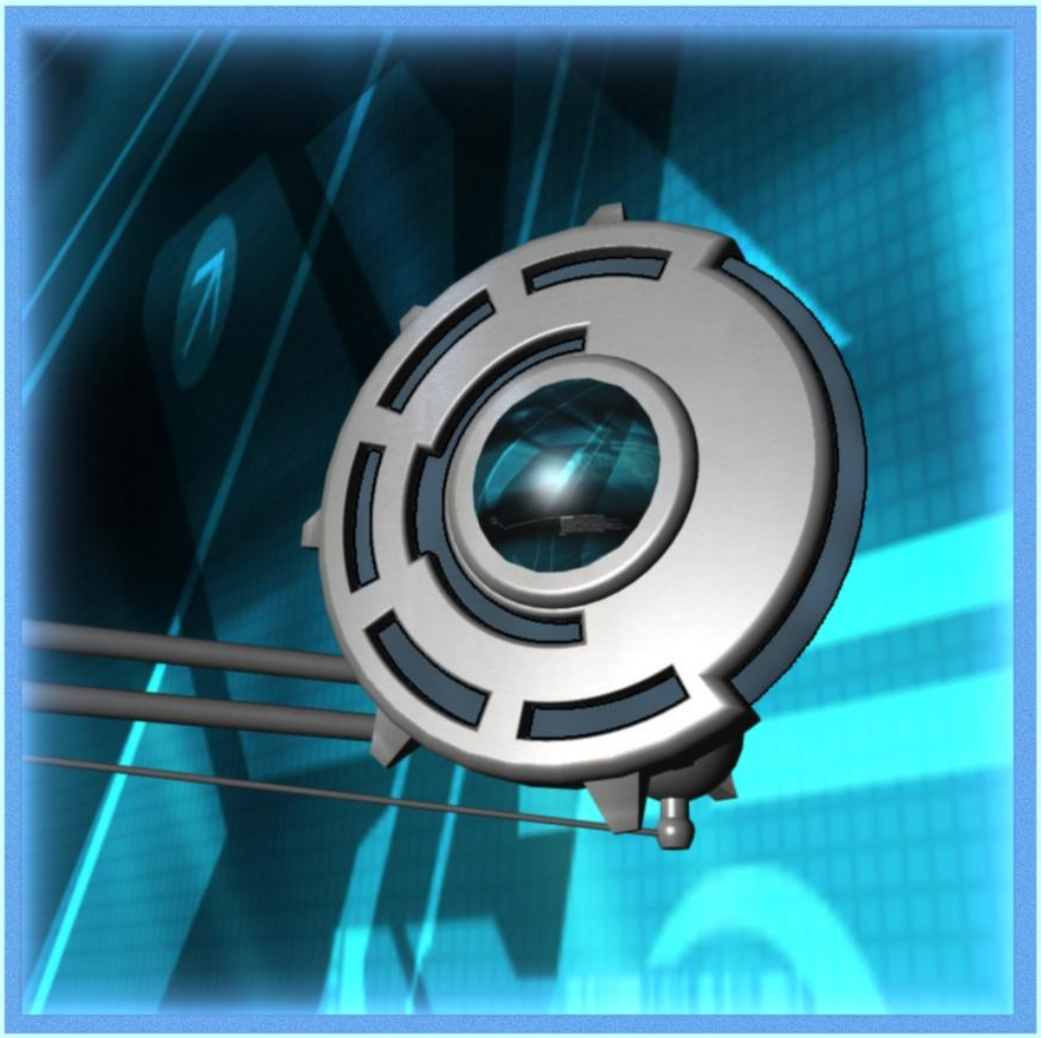
Como dijimos, no solo existen técnicas de nivel de detalle geométrico o de texturas, también existen técnicas LOD para shaders. El objetivo es exactamente el mismo, crear distintas versiones de un shader, cada una de estas con mejor calidad pero con mayor costo computacional.

Estos niveles de detalles se pueden crean de forma manual y a criterio del desarrollador. Por ejemplo, podríamos tener un shader que en su mayor nivel de detalle trabaje con parallax mapping, en su nivel intermedio con normal mapping y en el nivel más bajo con iluminación por pixel estándar.

También se está dedicando un esfuerzo considerable en automatizar la generación de los distintos niveles de detalle. Investigadores de la universidad de Maryland fueron de los primeros en buscar una forma de automatizar esta tarea. Particularmente se enfocaron en reducir los accesos a memoria dado que son muy costosos. Como efecto secundario, al reducir los accesos a memoria también se logra reducir un poco las operaciones en el propio shader. No obstante, falta mucho por lograr en este apartado, sería interesante ver técnicas automáticas que reduzcan el número de instrucciones y la cantidad de memoria usada.

En conclusión, cada aproximación tiene sus ventajas y desventajas, y queda a criterio del desarrollador cual es la más conveniente en cada caso. Lo que sí es un hecho es que el LOD en shaders es un tema de importancia, de tanta importancia como el LOD geométrico y el LOD en texturas.

APENDICE
CASO DE ESTUDIO



FINAL ENGINE

A2 Introducción

La mayor parte del tiempo que utilice para esta tesis lo dedique en crear un motor gráfico. Esta fue para mí una experiencia interesante y muy gratificante. Y es sobre este motor gráfico de lo que se hablara en este apartado. Partiremos hablando de los objetivos, discutiremos el porqué de la plataforma de trabajo elegida, veremos los recursos de los que disponía y por último se hablara de la realización del mismo.

A2.1 Objetivos

El objetivo principal de la tesis desde el principio fue desarrollar un motor gráfico reducido que permitiera aprender no solo sobre shaders, la prioridad del proyecto, sino sobre todos los temas relacionados con el desarrollo de un motor gráfico.

Acepte que el motor gráfico estaría basado en algún otro motor existente, y ese fue un gran acierto. Tratar de encarar un motor gráfico, por más pequeño que sea, es una tarea compleja que requiere de cierta experiencia. Por supuesto, que una vez que uno empieza a tener un mejor entendimiento de las tecnologías, la API gráfica, las herramientas, y de las formas convenientes de organizar las clases, uno comienza a separarse de esta base y aportar cada vez más.

La mayor parte del desarrollo de dicho motor se centro en el manejo gráfico de los objetos, dejando de lado, al menos parcialmente, sonido, IA y física. La priorización de objetivos fue tomada bien en cuenta desde el principio. Debía aceptar que muy probablemente el proyecto desbordaría mis recursos, en especial el tiempo. Afortunadamente, los puntos claves del motor se lograron hacer, lo cual resulto fundamental para interpretar mejor los conceptos de los shaders. Muy difícil se hace entender algo simplemente leyendo libros, la práctica es irremplazable en algunos casos.

A2.2 Plataforma de trabajo elegida

La API gráfica elegida fue DirectX 9, dado que era la que mayor actualizada estaba en lo referente a shaders, y es también la de mayor utilización para este tipo de tareas. Es más, gran parte de la documentación y herramientas existentes estaban enfocadas en DirectX y HLSL. Por su lado, OpenGL continuamente trata de ponerse al día, y lo logra bastante bien, en especial si OpenGL 3 resulta ser como se dice. Aún así DirectX generalmente está un paso más adelante, y después de la reestructuración que sufrió a partir de DirectX 9 la convierte en una opción bastante tentadora para este tipo de desarrollo, obviamente si es que el desarrollo multiplataforma no es nuestra prioridad. Por otra parte, DirectX 10 estuvo disponible (en versión beta) mucho después de comenzar con el proyecto, y es por eso que no se tomo en cuenta.

Por el lado de las herramientas vale aclarar que dispongo de ciertos conocimientos sobre RenderMonkey, es más, mucho de la programación de shaders lo aprendí con él. Sin embargo, me siento mucho más cómodo utilizando FX

Composer. Además, el motor sobre el que me base utiliza a FX Composer como herramienta principal de desarrollo de shaders. Por estas razones, indiscutiblemente la opción elegida fue FX Composer.

El lenguaje de la aplicación es C# dado que es un lenguaje muy limpio y cómodo de usar. Aún así fue mi decisión más difícil, dado que tenía aparejada una gran desventaja: muchos de los ejemplos están escritos en C++ debido a que es el lenguaje utilizado por excelencia en el desarrollo de aplicaciones 3D, particularmente por su velocidad y esparcimiento en la comunidad de desarrolladores desde ya hace años. Por suerte, más tarde descubrí que fue una buena elección elegir C#. El código es mucho más fácil de seguir, la performance del motor es buena y además encontré un muy buen motor escrito en C# para seguir de ejemplo, motor en el que me base.

A2.3 Recursos y aprendizaje inicial

Probablemente la etapa más difícil, la documentación es muy variada y poco regular en materia de calidad. Es más, a pesar de que disponía de una buena base en lo referente a conceptos de computación gráfica y en lo referente a programación general, en el tema shaders estaba prácticamente en blanco.

Comencé leyendo algún que otro libro, varios artículos y trate de basarme en los ejemplos que incluía el SDK de DirectX. Esos ejemplos tenían la ventaja de estar enfocados en el tema que querían mostrar, con la desventaja de una pobre organización, lo que impedía el escalamiento del código hacia un motor gráfico.

Gracias a estos ejemplos y a toda la documentación que tenia a mi disposición es como nació mi primer motor gráfico (segundo si considero un muy pequeño motor que hice en OpenGL hace ya un tiempo en la materia computación gráfica). En síntesis, el motor era muy limitado, permitía cargar objetos desde un archivo, aplicarles texturas, renderizarlos con iluminación por vértice y luego por pixel, utilizando un sistema de cámaras primitivo.

En ese momento me di cuenta de que el motor no podía progresar más sin un serio reestructuramiento. Descartarlo, era una decisión difícil de tomar dado que tenía que comenzar nuevamente de cero. Sin embargo, fue la decisión correcta, una buena base es vital para este tipo de proyecto. Aún así, la evaluación del proyecto en si fue más que positiva, me permitió entender muchos conceptos y organizar mejor mis ideas, especialmente sobre la estructuración de las clases.

Paralelamente a esto, seguía mi búsqueda de nuevo material y documentación, y es aquí donde encontré la página de Benjamin Nitschke. Por ese entonces, Benjamin había lanzado muy recientemente un tutorial de cómo crear un juego llamado Rocket Commander en C#, utilizando DirectX y FX Composer. Los tutoriales consisten de 10 videos, cada uno de 40 minutos de duración aproximadamente, los cuales están muy bien explicados. Los tutoriales son gratis, y además disponen de todo el código fuente del Rocket Commander. Benjamin Nitschke tiene experiencia real en el desarrollo de juegos, y eso se nota. El código es muy bueno, es complejo donde se enfoca el proyecto, y muy simple en temas secundarios, y por si no es poco la organización general es impecable. Definitivamente una opción muy tentadora.

Es aquí donde comenzó realmente el desarrollo de mi motor. Ya sabía lo que quería lograr, sabia aproximadamente cómo y tenía a mi disposición una guía de muy buena calidad. El Final Engine comenzó a escribirse, utilizando como base al Rocket Commander.

A2.4 Realización del motor

No pretendía hacer una copia del Rocket Commander, además había ciertas cosas que diferían con mis objetivos:

Rocket Commander	Final Engine
Puede utilizarse bajo el pipeline gráfico de función fija y programable.	Solamente sobre el pipeline gráfico programable.
Pobre organización a nivel de código sobre los objetos gráficos, debido a que maneja muy pocos objetos.	Los objetos gráficos tenían prioridad en mi motor. El objetivo era realizar una base completa y escalable.
La lógica del juego y otras partes se mesclan con el motor gráfico en sí. La magnitud del proyecto justificaba esta decisión.	El objetivo era disponer de una buena base, priorizando la organización estructurada.

Así mismo, existían omisiones que creía importantes en mi motor:

- Distintos tipos de luces: punto, direccional, spot, ambiental.
- Distintos tipos de animaciones: cinemática inversa, cinemática directa, animaciones cargadas desde archivos y animaciones primitivas como mover, rotar, escalar.
- Posibilidad de cargar distintos tipos de formatos para objetos poligonales: .x fue el único implementado. Sin embargo, se agregó la posibilidad de cargarlos utilizando el esquema de LOD denominado Progressive Meshes.
- Distintos tipos de shaders: material constante, phong, parallax bump mapping, distintos tipos de pre y post screen shaders. Y otros shaders más específicos, que simulaban, por ejemplo, la atmosfera de un planeta.

Teniendo todo esto en cuenta y algunos que otros detalles más, se comenzó a codificar este motor. Por supuesto que todavía me faltaba comprender mucho sobre el tema, de hecho todavía me falta. Pero disponía de la suficiente base de conocimientos como para realizar un trabajo mucho mejor, y así fue.

Así mismo, complemente mi trabajo de codificación creando modelos poligonales en Softimage XSI. Estos modelos se utilizaron en un menú, una pantalla de carga y en un escena en el espacio. Tratar de utilizar el motor es importante, no solo para mostrar las posibilidades y motivarse a uno mismo, sino también para probarlo en una situación real y arreglar pequeños, o tan pequeños, problemas que puedan surgir.

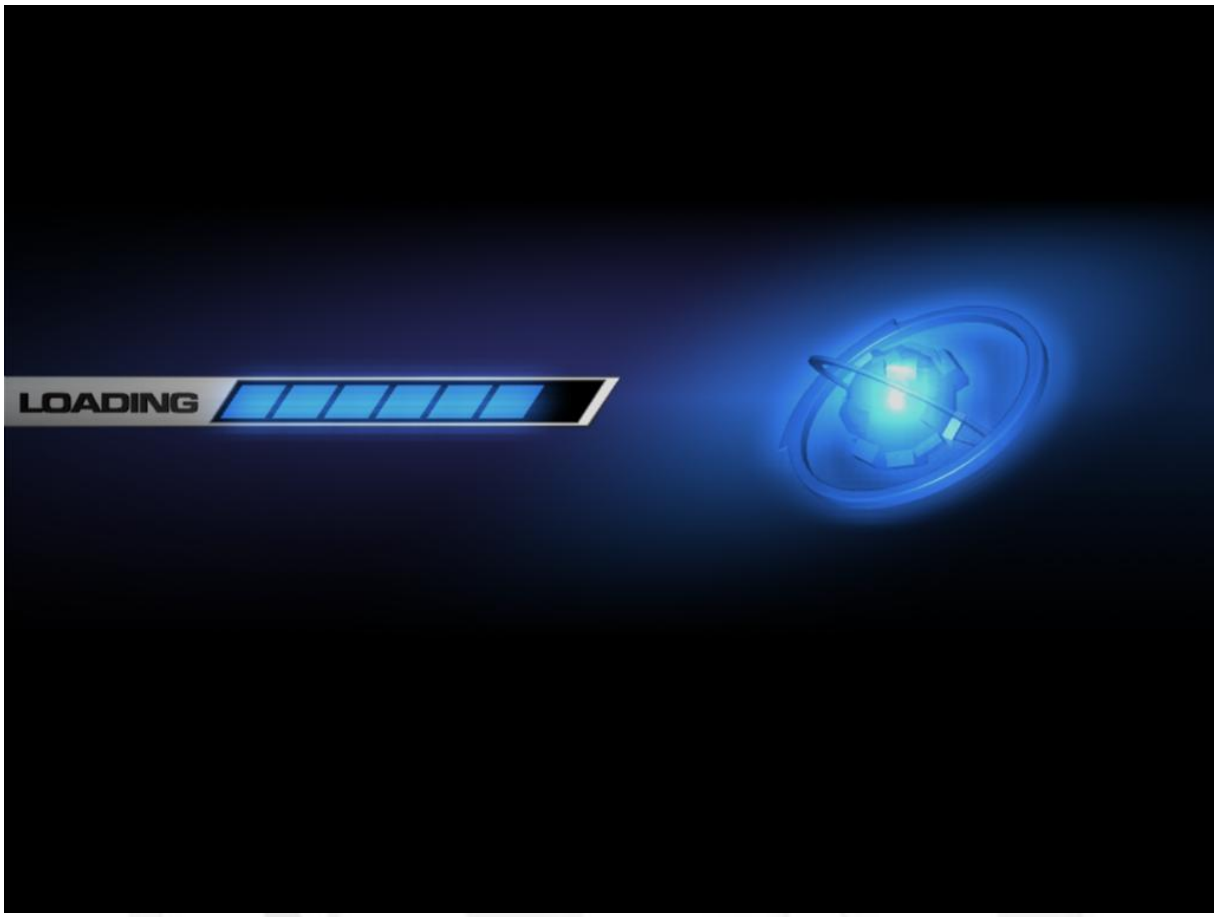
No hablare sobre los detalles del motor dado que no es importante para el tema actual. Es más, el motor está lejos de su versión final (de ahí su nombre) y no tendría una verdadera utilidad describir la estructura del mismo.

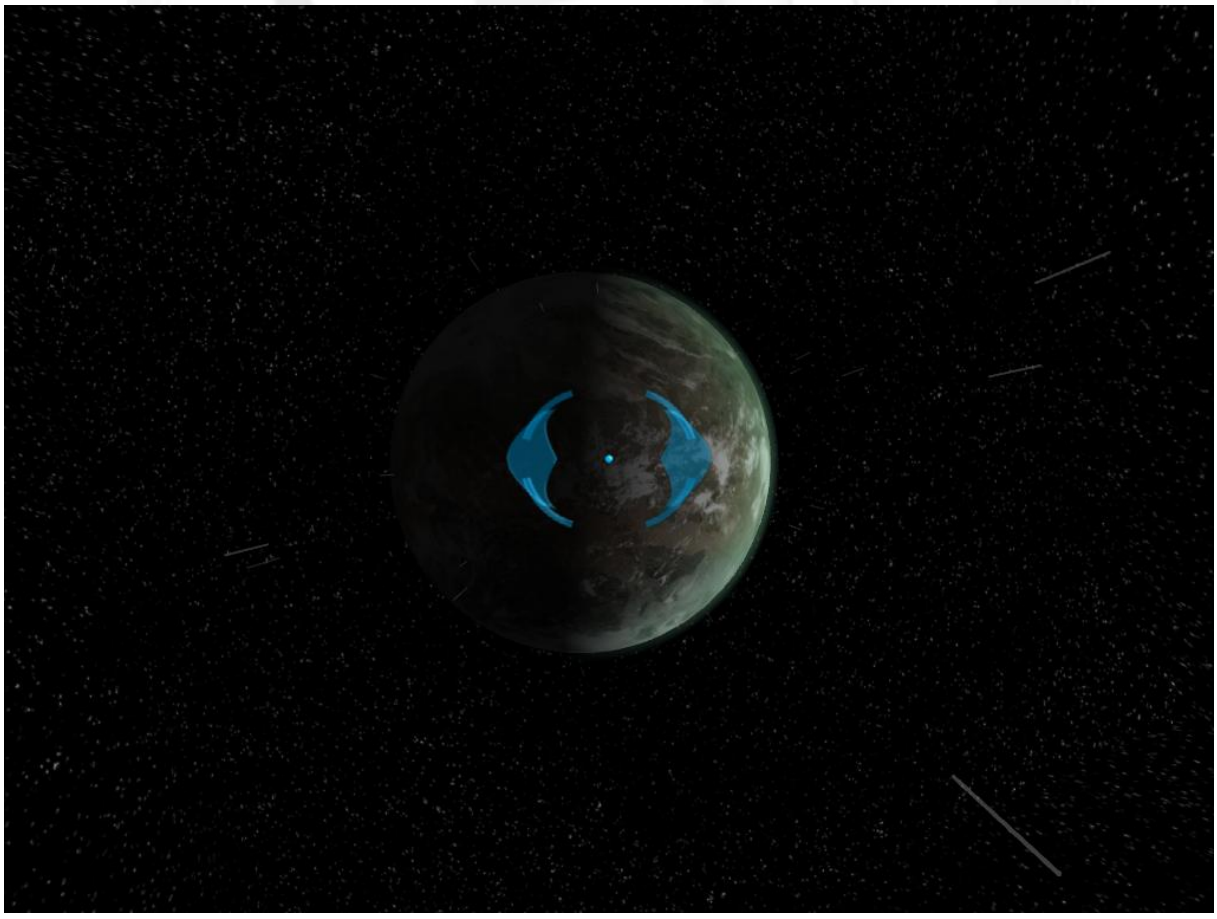
En definitiva, a partir de lo aprendido en estos últimos meses de escritura de la tesis he pulido mis conocimientos y he incorporado muchos nuevos. Y es por esto, que sería interesante que pudiera seguir con este motor, reestructurar ciertas clases, incorporar ciertas cosas aprendidas y terminar algunas que ni siquiera se empezaron. Con el objetivo de dejarlo al nivel que me gustaría y ¿porque no? utilizarlo en una aplicación real, ya sea académica, industrial o de entretenimiento. Ojala, disponga del tiempo para continuarlo.

A continuación, se muestran algunas capturas de pantalla del Final Engine:


LOADING


FINALENGINE
START
OPTIONS
CREDITS
EXIT


LOADING



BIBLIOGRAFÍA

Antes que nada, debería destacar a la comunidad de internet como una de mis grandes fuentes de información, gracias a que me brindo una innumerable cantidad de artículos, foros, tutoriales, etc. Ha sido de un aporte indiscutible para poder realizar este libro.

- Rocket Commander por Benjamin Nitschke (http://blogs.msdn.com/coding4fun/archive/2006/11/06/997852.aspx)
- Shaders: for game programmers and artists por Sebastien ST-Laurent.
- Wikipedia.
- ShaderX2: Introductions & Tutorials with DirectX 9.
- Shader Programming por Dr. Jian Huang.
- Programming Graphics Hardware por Randy Fernando, Mark Harris, Matthias Wloka y Cyril Zeller.
- Level of Detail for 3D Graphics (Morgan Kaufmann) por David Luebke, Martin Reddy, Jonathan D. Cohen, Amitabh Varshney, Benjamin Watson y Robert Huebner.
- A Real-Time Procedural Shading System for Programmable Graphics Hardware por Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkovy, Pat Hanrahan.
- El pipeline gráfico por César Mendoza.
- A Comparison of Real Time Graphical Shading Languages por Anthony Lovesey.
- Fixed function pipeline using vertex programs por Lars Ivar Igesund and Mads Henrik Stavang.
- Learn Vertex and Pixel Shader Programming with DirectX® 9 por James C. Leiterman.
- An Embedded Shading Language por Zheng Qin.
- GPU Programming “Languages” por Suvenkat.
- The GPU and Graphic Algorithms - State of the Art and Concept por Ivo Hanák.
- Microsoft Developer Network o MSDN.

- Automatic Shader Level of Detail por Marc Olano, Bob Kuehne y Maryann Simmons.
- Especificación del lenguaje Cg.
- ASC2GPU: Stream Compilation to Graphics Cards por Lee Howes.
- Gráficas Interactivas por Isaac Rudomín y Erik Millán.
- Level of detail Management for games por David Luebke.
- DevMaster.net (http://www.devmaster.net/)
- GameDev.net (http://www.gamedev.net/)
- The Z Buffer (http://www.thezbuffer.com/)

INDICE

CAPITULO I PIPELINE GRÁFICO

CAPITULO II SHADERS

CAPITULO III LENGUAJES

CAPITULO IV HERRAMIENTAS

APENDICE I NIVEL DE DETALLE

APENDICE II CASO DE ESTUDIO