GRAPHIC SHADERS FOR SCIENTIFIC VISUALIZATION

DAVID FERNÁNDEZ ALCOBA

DOBLE GRADO EN INGENIERÍA INFORMÁTICA - MATEMÁTICAS
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTESNE DE MADRID

Trabajo de Fin de Grado en Ingeniería Informática - Matemáticas

26 de julio de 2019

Directora:
Ana Gil Luezas

Autorización de difusión

David Fernández Alcoba

26 de julio de 2019

El/la abajo firmante, matriculado/a en el Doble Grado en Ingeniería Informática y Matemáticas de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Grado "GRAPHIC SHADERS FOR SCIENTIFIC VISUALIZATION", realizado durante el curso académico 2018-2019 bajo la dirección de Ana Gil Luezas en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

David Fernández Alcoba

Resumen

En el mundo actual, las investigaciones y estudios científicos generan gran cantidad de datos que han de ser interpretados de una manera eficaz, con el fin de sacar las mejores conclusiones y obtener resultados fiables que no den lugar a la duda. En este contexto, una de las disciplinas más importantes es la de la visualización, ya que puede ayudar a entender, ilustrar y obtener información relevante acerca del fenómeno que se está estudiando.

De igual forma, en los últimos años se ha dado una expansión considerable de las capacidades de las GPUs, ofreciendo nuevas posibilidades dentro de la informática gráfica e incrementando el rendimiento tanto en computación paralela como en aplicaciones de visualización.

En este trabajo se exploran estas ideas, haciendo hincapié en las posibilidades que nos ofrecen los distintos tipos de shaders gráficos dentro de la especificación OpenGL, y la manera en la que nos pueden ser útiles a la hora de interpretar datos y obtener representaciones para problemas típicos de visualización científica, como puede ser la visualización de datos en tres dimensiones, visualización de volúmenes, renderizado de curvas y superficies, etc.

Palabras clave

Visualización científica, GPU, Shader, OpenGL, Bézier, Superficies.

Abstract

Nowadays, scientific studies and investigations generate a great amount of data that has to be well interpreted, so as to extract the best possible conclusions and obtain reliable results. Within this context, one of the most important disciplines is that of visualization, since it can help understand, illustrate and obtain relevant information about the phenomenon being studied.

Similarly, in the last few years, a considerable expansion of the capabilities of GPUs has been taking place, offering new possibilities within graphics computing and increasing performance both in parallel computing and in visualization applications.

In this text those ideas are explored, emphasizing the possibilities that the different kinds of graphic shaders have to offer within the OpenGL specification, and the way these can help interpret data and obtain representations for common scientific visualization topics, such as three dimensions data Visualization, volume visualization, surface rendering, etc.

Keywords

Scientific Visualization, Graphic Shaders, GPU, OpenGL.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Tal y como se cuenta en Defanti and Brown [1], los científicos computacionales basan su trabajo en fuentes de datos de gran volumen. Sin embargo, estos datos tienen tal magnitud que los científicos se ven, a menudo, superados. Entre las fuentes de datos de gran volumen se encuentran:

- Supercomputadores
- Inteligencia militar, satélites, datos astronómicos y de tiempo atmosférico
- Sondas enviando datos desde otros planetas
- Radio telescopios terrestres
- Instrumentos capturando temperaturas oceánicas, movimientos tectónicos y actividad volcánica y sísmica
- Escáneres médicos empleando distintas técnicas de imagen como tomografía, resonancias magnéticas, etc

Simplemente con un formato numérico, el cerebro humano es incapaz de interpretar gigabytes de datos cada día, resultando en mucha información desperdiciada. De aquí surge la necesidad de una alternativa a los números. La posibilidad de los científicos para visualizar cómputos complejos y simulaciones es absolutamente esencial para asegurar la integridad de análisis y predicciones, así como presentar esta información al resto.

Esta capacidad de visualización se hace especialmente importante en el ser humano, puesto que, de todas nuestras funciones cerebrales, nuestro sistema de visión es el que mayor capacidad de procesamiento de información tiene. Según expertos en conocimiento, el procesamiento de información en humanos tiene dos formas: preconsciente y consciente. El procesamiento de información preconsciente es involuntario, similar a la respiración. Este es el tipo de procesamiento que se da en información gráfica. Rohrer [2]

Teniendo esto en cuenta y el hecho de que cada persona tiene una capacidad de vision espacial diferente, la informática gráfica puede ayudar a aquellos que tienen una mayor dificultad y que, de otro modo, serían incapaces de visualizar conceptos complejos.

Estos hechos muestran una necesidad ha resultado en el surgimiento, en la última década, de una disciplina totalmente independiente, la visualización científica.

1.1. Motivación

La importancia de lo expuesto anteriormente sirve como suficiente motivación, aunque a esto se ha de añadir el reto personal de, con este trabajo, aprender y entender un área de la informática que no forma parte del itinerario en mi formación, como es la informática gráfica, y que engloba muchas de las materias vistas hasta ahora tanto en ingeniería informática como en matemáticas.

Además, esta rama dentro de la investigación científica es relativamente reciente, asociándose su nacimiento en 1987 al artículo McCormick et al. [3], por lo que aún hay muchos retos y problemas por resolver, haciendo su estudio muy interesante.

1.2. Objetivos

El objetivo principal de este trabajo es el de aprender el funcionamiento básico de los gráficos y la aplicación de éstos a la investigación científica. Este objetivo se puede desglosar en otros subobjetivos más concretos y que marcan la línea de trabajo:

1. Comprender el pipeline de gráficos y la utilidad y funcionamiento de los shaders, así como aprender el lenguaje GLSL para su escritura.

2. Aprender las técnicas más conocidas de visualización científica y cómo desarrollar shaders que las implementen.

3. Desarrollar una aplicación que ponga de manifiesto lo aprendido, desarrollando shaders que ilustren algunas de las técnicas vistas.

1.3. Plan de trabajo

Con estos objetivos en mente, se desarrolló el siguiente plan de trabajo, acordado en reuniones iniciales entre tutora y autor del trabajo.

- **Toma de contacto con OpenGL** Durante esta fase se leyeron tutoriales sobre OpenGL y se experimentó con diversos shaders y librerías para familiarizarse con la tecnología, a la vez que se aprendía el lenguaje GLSL.

- **Documentación** Durante la duración completa del proyecto se llevó a cabo una documentación acerca de las distintas fuentes de información, con el objetivo de no olvidar incluir partes importantes en la memoria.
- **Comunicación con el tutor** Se concretaron diversas reuniones con la tutora durante las partes intensivas del proyecto con el fin de mostrar avances y acordar los siguientes pasos. Asimismo, se mantuvo una comunicación mediante correo electrónico para aquellas dudas menores que surgieron durante la realización del trabajo.
- **Preparación del entorno de desarrollo** Durante esta fase se preparó el equipo, instalando las librerías y programas necesarios para el correcto funcionamiento de la aplicación.
- **Desarrollo de la aplicación** Una vez preparado el entorno, se continuó durante toda la duración del trabajo con el desarrollo de la aplicación, incluyendo cada vez nuevas capacidades.
- **Redacción de la memoria** Se inició la redacción de la memoria una vez se tenían conocimientos suficientes, a mitad de la elaboración del trabajo. Una vez comenzada la redacción, se fue reeditando y mejorando en un proceso iterativo.

1.4. Estructura de la memoria

El siguiente capítulo, **OpenGL y DirectX** 2, presenta las dos grandes especificaciones dentro de la informática gráfica, centrándose en OpenGL y analizando sus características, capacidades y debilidades, así como las diferencias entre ambas.

Posteriormente, el capítulo **Shaders y Visualización Científica** 3 explora las técnicas más comunes dentro del campo de visualización científica y qué tipos de shaders son útiles para cada una de ellas, introduciendo algunos de los que más adelante se presentarán junto a la aplicación.

En el capítulo **Aplicación** 4 se presenta la aplicación desarrollada, explicando el diseño, capacidades, experimentos realizados...

Por último el capítulo **Conclusiones y Trabajo Futuro** 5 incluye un análisis del trabajo realizado, el nivel de cumplimiento de los objetivos propuestos y posibles líneas de trabajo futuro.

El código de la aplicación desarrollada puede encontrarse en el siguiente enlace:
http://github.com/davidfdezalcoba/TFG

Capítulo 2

OpenGL y Direct3D

El objetivo de este capítulo es explicar en qué consiste OpenGl, así como su pipeline de gráficos, los tipos de shaders que incluye y las diferencias que presenta con DirectX, su principal competidor.

2.1. ¿Qué es?

OpenGL [4] se define como una API *(application programming interface)*, que es simplemente una librería de software para acceder a capacidades del hardware de gráficos (ver Shreiner et al. [5]).

OpenGL está diseñado como una interfaz independiente del hardware que puede ser implementada en muchos sistemas hardware de gráficos diferentes, o completamente como software, en el caso de que el sistema no posea hardware de gráficos. OpenGL no proporciona ninguna funcionalidad para describir modelos en tres dimensiones ni operaciones para leer ficheros (como imágenes JPEG, por ejemplo). En su lugar, se deben construir los objetos tridimensionales a partir de un pequeño conjunto de primitivas geométricas—puntos, líneas, triángulos y parches.

2.2. Breve historia de OpenGL

OpenGL nace a principios de los años 90, desarrollada por Silicon Graphics (SGI). En los años 80, Silicon Graphics poseía una API privada denominada IRIS GL, utilizada para producir gráficos en sus estaciones de trabajo IRIS. Posteriormente, debido a la pérdida de cuota de mercado, decidió hacer su API pública. Sin embargo, a causa de problemas con patentes y el hecho de tener características poco relevantes para los gráficos 3D como la funcionalidad de ventanas, se decidió reescribir algunas de las partes y se lanzó lo que ahora se conoce como OpenGL.

Esta nueva especificación consiguió logros importantes para la informática gráfica, como estandarizar el acceso al hardware gráfico, trasladar a los fabricantes la responsabilidad del desarrollo de las interfaces con el hardware y delegar la funcionalidad de ventanas al sistema operativo. Todo esto supuso un gran impacto en la industria, al ofrecer a los desarrolladores una plataforma de alto nivel sobre la que trabajar.

En 1992, Silicon Graphics lideró la creación del OpenGL Architecture Review Board (OpenGL ARB) [6], un grupo de empresas del sector que sería la encargada de mantener y extender la especificación en los años siguientes. El OpenGL ARB estaba formado por 3Dlabs, Apple, ATI, Dell, IBM, Intel, Nvidia, SGI and Sun Microsystems.

En otoño de 2006, el OpenGL ARB y los directores de Khronos votaron para transferir el control de OpenGL a Khronos Group. El secretario de la ARB Jon Leech observó: *"Hemos decidido mover OpenGL a Khronos para asegurar la salud futura de OpenGL en todas sus formas."* [6]

2.3. Diseño de OpenGL

OpenGL implementa el llamado pipeline de gráficos o pipeline de renderizado. Este pipeline consiste en una secuencia de etapas de procesamiento para convertir datos de una aplicación en una imagen final renderizada.

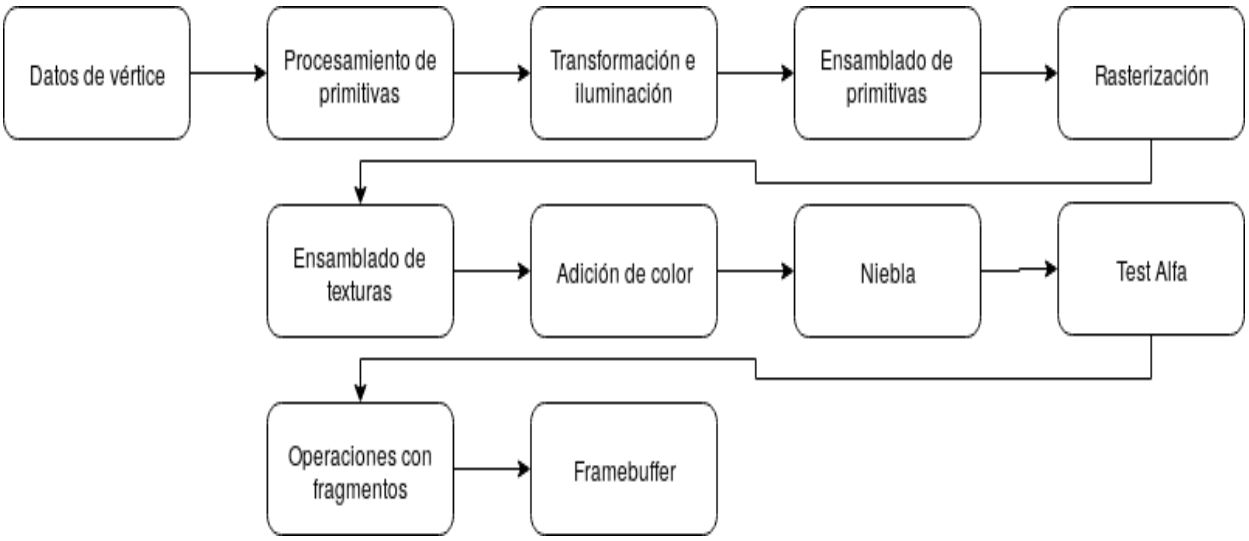


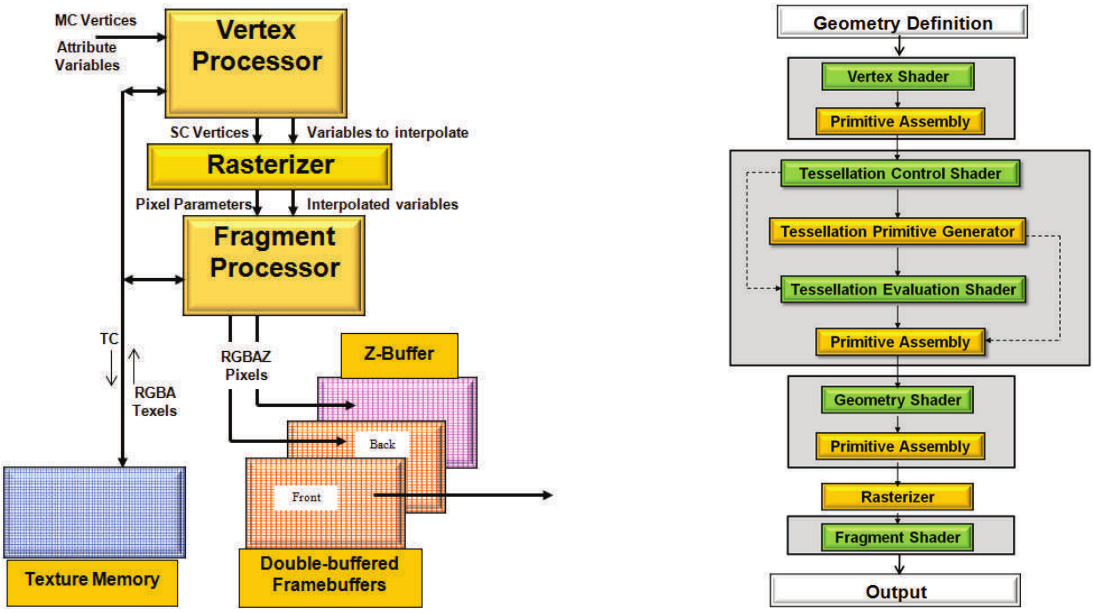
Figura 2.1: Pipeline no programable. OpenGL 1.5

En un principio, este pipeline de renderizado consistía en varias etapas fijas y no programables, en las que el programador simplemente elegía una serie de configuraciones para

que OpenGL realizase las operaciones propias de cada etapa. Las etapas principales de este pipeline eran las siguientes (Ver Figura 2.1):

- Procesamiento de primitivas
- Transformación e iluminación
- Ensamblado de primitivas
- Rasterización
- Ensamblado de texturas
- Adición de color
- Niebla
- Test Alfa
- Operaciones con fragmentos

Sin embargo, con la versión 2.0 de OpenGL se introdujeron los *shaders*, explorados en detalle en el Capítulo 3, que sustituyeron algunas de estas etapas fijas por etapas programables, dando mayor flexibilidad al programador. La Figura 2.2 muestra el pipeline asociado a la versión 4.3 de OpenGL.



(a) Pipeline de OpenGL (b) Pipeline detallado

Figura 2.2: Pipeline de gráficos de OpenGL

El renderizado de una imagen comienza con una llamada, desde la aplicación principal, a una función de dibujado de OpenGL. Con esta llamada, OpenGL coge datos sobre vértices contenidos en diferentes objetos y renderiza con estos datos una o más *primitivas* [7].

Las primitivas geométricas son las interpretaciones que OpenGL puede hacer sobre lo que representa un flujo de vértices. Por ejemplo, tres vértices pueden significar tres puntos independientes, dos líneas, un triángulo, etc. OpenGL contiene los siguientes tipos principales de primitivas:

2.3.1. Primitivas de Puntos

OpenGL contiene una única primitiva de puntos: `GL_POINTS`. Cuando se llama a una función de dibujado con esta primitiva, OpenGL interpreta el flujo de vértices como puntos independientes.

2.3.2. Primitivas de Línea

En cuanto a las primitivas de línea, hay tres tipos, basándose en distintas interpretaciones del flujo de vértices.

- `GL_LINES`: Los dos primeros vértices se consideran una línea, los dos siguientes otra línea, etc.
- `GL_LINE_STRIP`: Los vértices adyacentes se consideran una línea. Es decir el primero y el segundo forman una línea, el segundo y el tercero forman otra línea, etc.
- `GL_LINE_LOOP`: Como `GL_LINE_STRIP`, pero el primer y último vértices se consideran también una línea.

2.3.3. Primitivas de Triángulo

Un triángulo es una primitiva formada por tres vértices. También existen tres primitivas de triángulo:

- `GL_TRIANGLES`: los tres primeros vértices forman un triángulo, los tres siguientes otro, etc.
- `GL_TRIANGLE_STRIP`: cada grupo de tres vértices adyacentes forman un triángulo. El (0, 1, 2), el (1, 2, 3), etc.
- `GL_TRIANGLE_FAN`: El primer vértice queda fijo, y cada grupo de dos vértices adyacentes a partir de él forma un triángulo con el primero. Por ejemplo el (0, 1, 2), (0, 2, 3), etc.

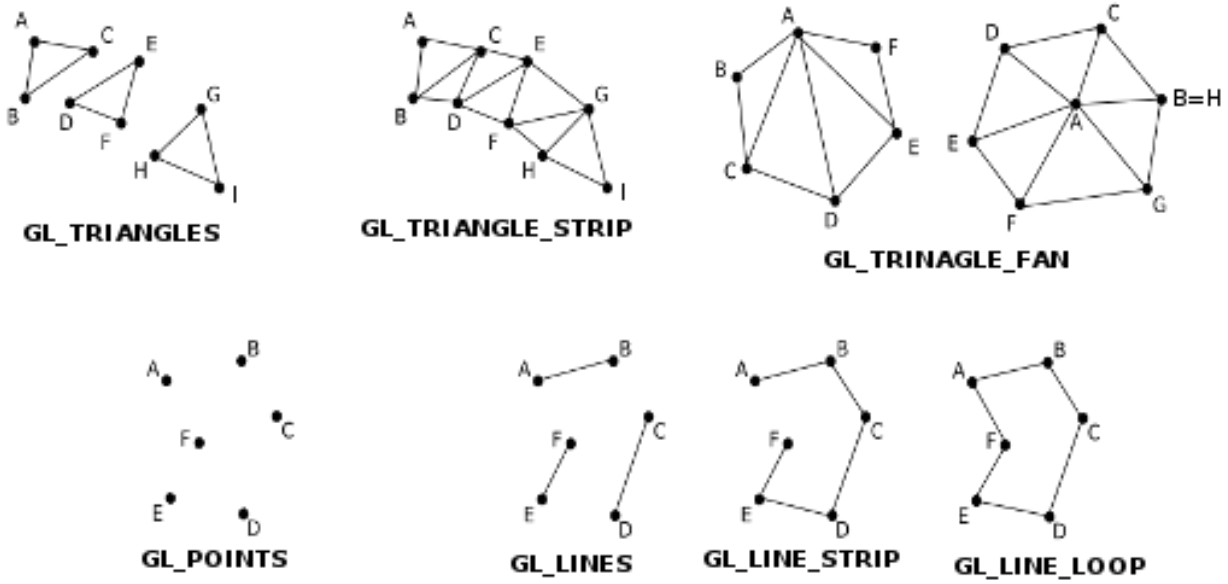


Figura 2.3: Primitivas de OpenGL

En la Figura 2.3 se pueden ver las diferentes interpretaciones en forma de primitiva que puede realizar OpenGL para renderizar un flujo de vértices.

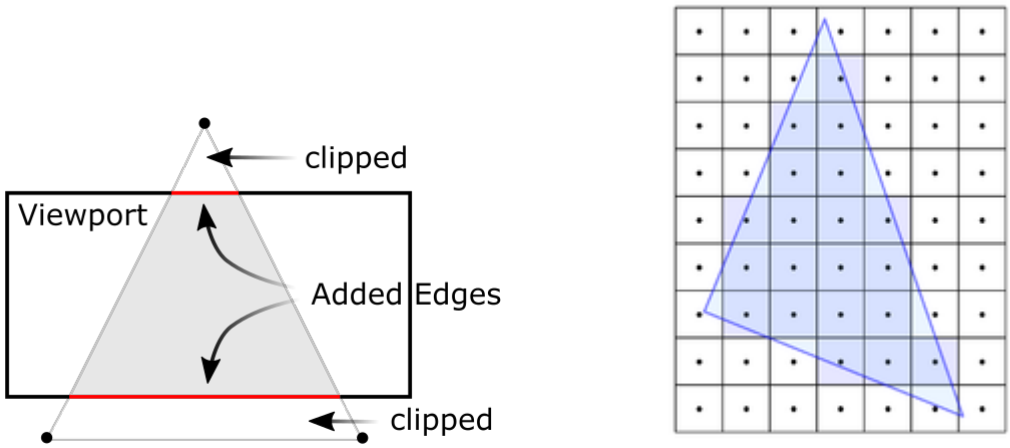
Una vez se ha hecho la llamada a la función de dibujado, comienza el proceso de renderizado, recorriéndose las diferentes etapas que forman el pipeline de renderizado.

2.3.4. Procesamiento de Vértices

La primera etapa del pipeline programable —Vertex Processor, Figura 2.2a—, está formada a su vez por los shaders programables de vértice, teselación y geométrico, que sustituyen a las etapas de transformación e iluminación del pipeline no programable, así como la etapa fija realizada por OpenGL conocida como Ensamblado de Primitivas. Obtiene como entrada el flujo de vértices, normales, definiciones de primitivas geométricas, colores, parámetros de iluminación, materiales y coordenadas para las texturas. Esta etapa opera sobre estos datos y los organiza en las primitivas geométricas asociadas en preparación para la etapa de post-procesado de vértices, en la que se realizan, entre otras cosas, su recorte y rasterización. Así, la salida es un conjunto de vértices como píxeles, con su color, profundidad y coordenadas de textura.

2.3.5. Clipping

Posteriormente se realiza el recorte o *clipping*, que ocurre cuando, ocasionalmente, algunos vértices quedan fuera de lo que se denomina *viewport*—la región de la ventana donde se permite dibujar— modificando las primitivas geométricas para que nada quede fuera de ese espacio. (Ver Figura 2.4a).



(a) Clipping en OpenGL

(b) Rasterización en OpenGL

Figura 2.4: Clipping y rasterización en OpenGL

2.3.6. Rasterización

La siguiente etapa es la rasterización (ver Figura 2.4b). Este paso implementa la transición de vértice a fragmento. Inmediatamente después del clipping, las primitivas geométricas actualizadas se mandan al rasterizador para la generación de fragmentos. Podemos considerar un fragmento como un "candidato a pixel", en el sentido de que un pixel reside en el *framebuffer*— un espacio de memoria manejado por el hardware gráfico que manda al dispositivo de salida digital— mientras que un fragmento puede ser rechazado y nunca actualizar su localización de pixel asociada.

2.3.7. Procesamiento de Fragmentos

En la última etapa, el procesado de fragmentos se realiza a su vez en dos pasos, uno de ellos programable con el shader de fragmentos y otras operaciones sobre fragmentos realizadas automáticamente por OpenGL. Durante este último procesamiento, la visibilidad de cada fragmento es determinada utilizando diferentes tests (profundidad, color, plantilla,

ruido...).

Cuando un fragmento pasa por todas estas etapas y todos los test activos, puede ser escrito directamente en el buffer de fragmentos, actualizando su color y valor de profundidad de su pixel o, si el *blending* está activado, mezclando su color con el del pixel actual para generar un nuevo color que se escribe en el buffer de fragmentos.

2.4. Diferencias con Direct3D

Direct3D es parte del conjunto de la API multimedia DirectX, propiedad de Microsoft [8]. Es la principal competidora de OpenGL, ofreciendo ambas un conjunto similar de funcionalidades. Aun así, se pueden observar varias diferencias en términos de disponibilidad, portabilidad, facilidad de uso, rendimiento, estructura y usuarios finales. En la tabla 2.1, se muestran algunas de ellas resumidas.

	OpenGL	Direct3D
Soporte de escritorio	Multiplataforma	Microsoft Windows Xbox
Soporte sistemas empotrados	Multiplataforma (OpenGL ES)	Windows Embedded Windows CE
Licencia	Libre (con características patentadas)	Privativa
Usuario final	Profesionales	Juegos

Tabla 2.1: Diferencias entre OpenGL y Direct3D

En cuanto a la portabilidad, DirectX solo está disponible para la familia de sistemas operativos Microsoft Windows, mientras que OpenGL tiene implementaciones en muchas plataformas, incluyendo Microsoft Windows y sistemas Unix.

En términos de facilidad de uso hoy en día ambas APIs se encuentran bastante parejas, habiendo evolucionado desde las primeras versiones, en las que los desarrolladores instaban a Microsoft a unirse a la iniciativa OpenGL, debido a que suponía menos esfuerzo trabajar con esta última.

El diseño de ambas es también similar. Tras muchos años de evolución, el pipeline gráfico es bastante parecido, como puede apreciarse en la Figura 2.5.

En conclusión, hoy en día ambas se consideran bastante similares en cuanto a rendimiento y capacidades, siendo el soporte a otros sistemas y el carácter de las licencias las únicas grandes diferencias que hacen decantarse a los desarrolladores por una u otra.

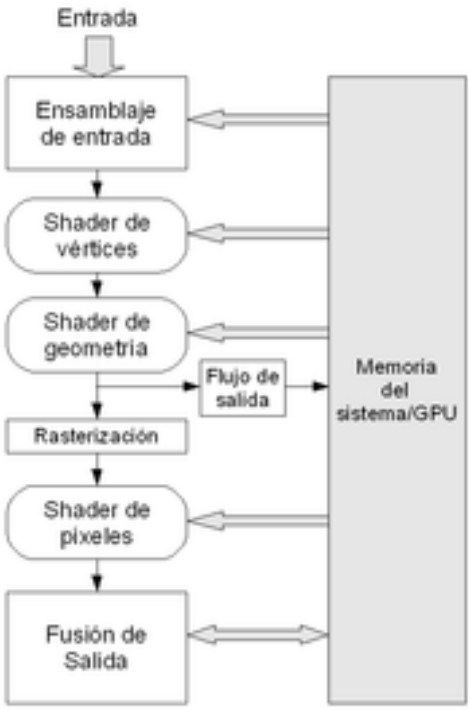


Figura 2.5: Pipeline de gráficos de Direct3D

Capítulo 3

Shaders y Visulización Científica

Como se ha visto en la sección 2.3, el pipeline de gráficos de OpenGL tiene cuatro etapas programables:

- Shader de Vértices (Vertex Shader)
- Shaders de Teselación

  Shader de Control de Teselación (Tessellation Control Shader)

  Shader de Evaluación de Teselación (Tessellation Evaluation Shader)
- Shader Geométrico (Geometry Shader)
- Shader de Fragmento (Fragment Shader)

En este capítulo se explicará cómo funcionan, cómo desarrollarlos y cómo utilizarlos para resolver problemas de visualización científica habituales.

3.1. Shaders

Los shaders gráficos son un tipo de programa utilizado inicialmente para producir niveles apropiados de luz, oscuridad y color en una imagen. Sin embargo, hoy en día se utilizan con diversas finalidades diferentes como efectos especiales, post procesado de vídeos, videojuegos, etc.

Los shaders se introdujeron en OpenGL en la versión 2.0, incluyendo el lenguaje de programación centrado en shaders OpenGL Shading Language, también conocido como GLSL [9]— un lenguaje tipo C creado específicamente para que los desarrolladores tuviesen más control sobre el pipeline de renderizado.—

Durante el proceso de desarrollo de shaders no es necesario incluir todas las etapas que se muestran en la Figura 2.2b, aunque normalmente, si se decide utilizar alguno de ellos, se requiere utilizar, al menos, un Vertex Shader.

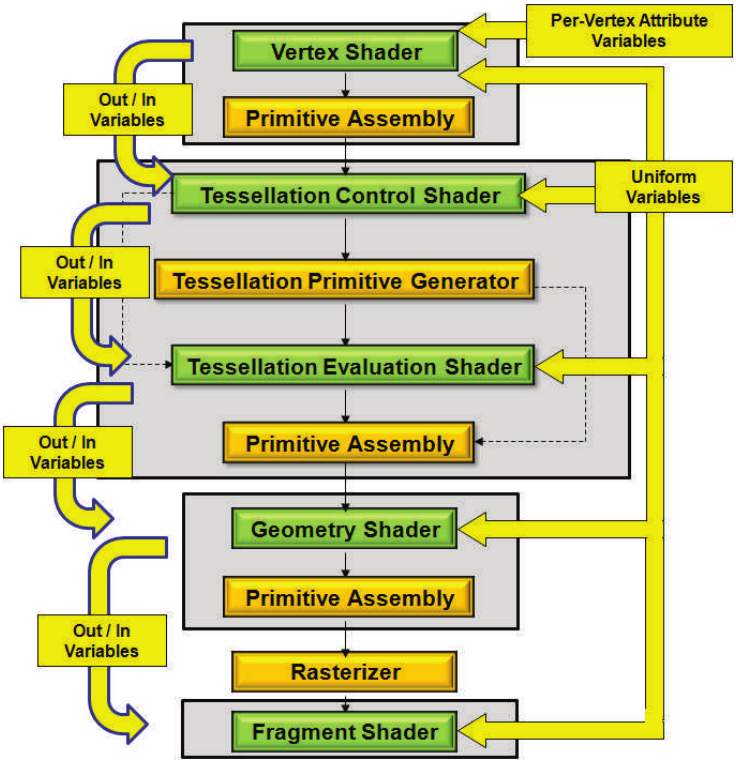


Figura 3.1: Comunicación entre shaders del pipeline

Los shaders del pipeline se comunican entre ellos mediante variables proporcionadas por GLSL, siendo la salida de un shader la entrada del siguiente, como se muestra en la Figura 3.1. Un breve resumen acerca del lenguaje GLSL se incluye en el Apéndice A.

3.1.1. Vertex Shader

El Vertex Shader es la etapa de sombreado en el pipeline de renderizado que se encarga del procesamiento de los vértices individuales [10]. El Vertex Shader tiene como entrada unos atributos de vértice especificados desde un *Vertex Array Object (VAO)* por un comando de dibujo. Recibe un único vértice, formado por sus atributos, del flujo de vértices y genera un único vértice al flujo de salida. Por cada vértice de entrada ha de haber, necesariamente, uno de salida.

Este shader se invoca una vez por cada vértice en el flujo de entrada, exceptuando el caso en el que OpenGL detecte que una invocación a este shader con las exactamente las mismas entradas ya ha sido realizada, en cuyo caso se reutilizan los resultados de la invocación previa, resultando en un ahorro de tiempo valioso.

Normalmente, las operaciones que se realizan en el Vertex Shader son transformaciones para el espacio de post-proyección, iluminación por vértice o preparación para las siguientes

etapas del pipeline.

3.1.2. Tessellation Shaders

La Teselación es la etapa, opcional, del pipeline de renderizado que consiste en subdividir un parche de algún tipo y computar los valores de los nuevos vértices creados en el proceso. Está compuesta, a su vez, por otras tres etapas, dos de ellas programables en forma de shader, y una intermedia fija. Cada una de estas etapas se encarga de una parte del proceso de teselación.

En esta sección se explican los dos shaders involucrados, además de la etapa intermedia, llamada generador de primitivas de teselación, pues resulta importante para entender el proceso y las entradas y salidas a los shaders.

Tessellation Control Shader

El Tessellation Control Shader (TCS) [11] es la primera etapa del proceso de teselación, en el caso de ser utilizado. Se sitúa inmediatamente posterior al Vertex Shader e inmediatamente anterior al generador de primitivas de teselación. Controla cuánta teselación provocar en un parche determinado, así como el tamaño del parche, permitiendo aumentar la cantidad de datos. Su función principal es la de comunicar al generador de primitivas de teselación el nivel de teselación deseado, así como proveerle los datos del parche al Tessellation Evaluation Shader mediante sus variables de salida.

Como entrada, el TCS obtiene la salida del Vertex Shader organizada en un vector de tantos vértices como tenga el parche de entrada. Cada invocación al TCS produce un único vértice como salida al parche de salida. Por cada vértice en el parche de entrada se realiza una invocación al TCS, resultando en tantas invocaciones como vértices hay en dicho parche.

En el caso de no utilizar un TCS, se pueden pasar valores por defecto a las siguientes etapas de teselación.

Generador de primitivas de teselación

El generador de primitivas de teselación [12] es la etapa que se encuentra entre los dos shaders de teselación, el TCS y el Tessellation Evaluation Shader. Esta etapa, fija en el pipeline, es la encargada de crear nuevas primitivas a partir del parche de entrada. La función principal de este sistema es la de determinar cuántos vértices crear, en qué orden hacerlo y qué clase de primitivas construir con ellos. Los datos reales de estos vértices, como color, posición, etc., han de ser generados por el TES. Debido a esto, el generador no tiene en cuenta el parche de salida producido por el TCS, sino que solo opera en términos de teselar

un cuadrado o triángulo abstracto, o un bloque de isolíneas.

Esta etapa, está supeditada al Tessellation Evaluation Shader, puesto que solo se ejecutará en el caso de que exista uno activo. La generación de primitivas en esta etapa se ve afectada por distintos factores:

- Niveles de teselación marcados por el TCS (O por defecto si no hay TCS)
- Espaciado de los vértices teselados, definido en el TES
- Tipo de primitiva, definido en el TES
- Orden de generación de primitivas, definido en el TES

La cantidad de teselación a realizar se define en niveles de teselación internos y externos. Funcionan de la siguiente manera: un nivel de teselación 4 indica que un borde se convertirá en 4 bordes (2 vértices se convertirán en 5). El nivel externo define el grado de teselación para los bordes externos de la primitiva. Esto permite que dos parches distintos se conecten apropiadamente, a pesar de tener distintos niveles de teselación dentro del parche. El nivel interno hace referencia el número de teselaciones a realizar dentro del parche abstracto.

Cabe destacar que no todos los parches abstractos utilizan los mismos niveles de teselación. Por ejemplo, los triángulos utilizan un único nivel interno y tres niveles externos. El resto de posibles niveles son ignorados.

El espaciado entre vértices puede realizarse de las siguientes maneras: espaciado equidistante, espaciado fraccional par o espaciado fraccional impar.

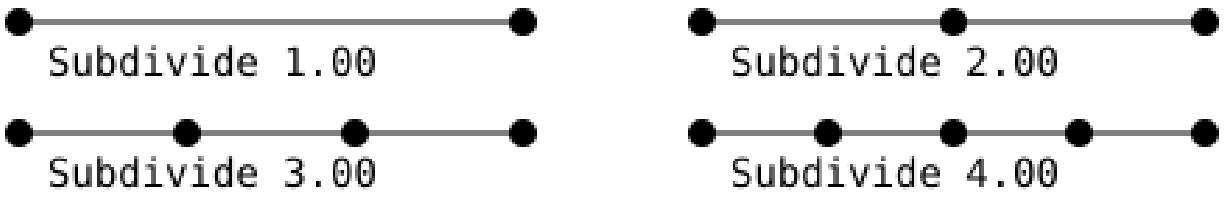


Figura 3.2: Teselación - Espaciado equidistante

El espaciado equidistante (ver Figura 3.2) divide el borde a teselar en segmentos de igual longitud. Solo acepta valores enteros, por lo que redondea el nivel de teselación hasta el siguiente entero. Este hecho causa que los segmentos aparezcan instantáneamente de un nivel a otro.

Para conseguir un comportamiento mas “suave” se tienen los otros dos modos de espaciado. Estos últimos son útiles especialmente cuando el nivel de teselación es dependiente del área vista desde la cámara. En el espaciado fraccional par el número de segmentos en

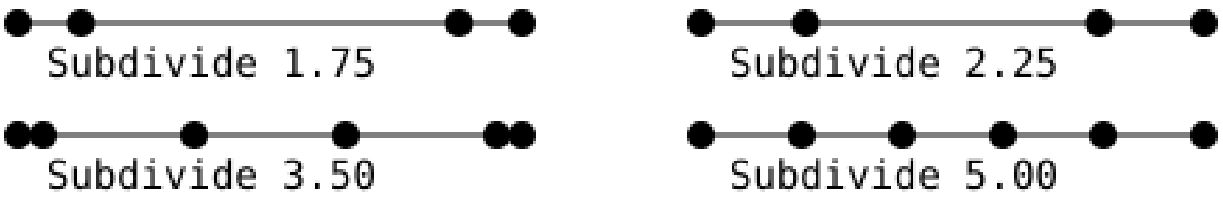


Figura 3.3: Teselación - Espaciado fraccional par

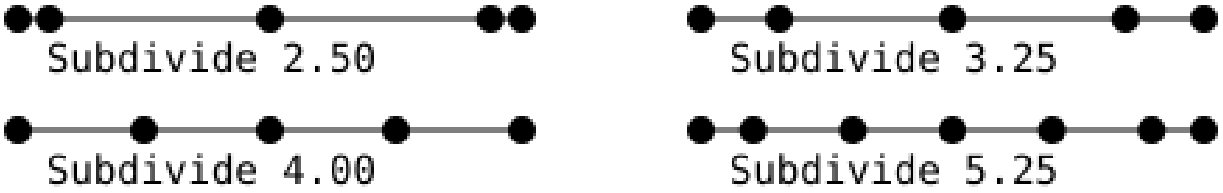


Figura 3.4: Teselación - Espaciado fraccional impar

los que dividir el borde (nivel de teselación efectivo) se redondea al siguiente entero par, mientras que en el espaciado fraccional impar se redondea al siguiente entero impar. Para estos modos de espaciado se necesita definir dos valores:

- n, el nivel de teselación efectivo, redondeado según lo anterior.
- f, el valor computado antes del redondeo. Un valor potencialmente fraccionario.

Según este esquema, los bordes a teselar se subdividen en dos conjuntos de segmentos. El primero con $n-2$ segmentos de igual longitud, el otro con 2 segmentos de igual longitud entre ellos, pero no necesariamente de igual longitud que los del el otro conjunto. Estos dos segmentos tendrán menor longitud que los otros en general. La longitud de estos es exactamente $n-f$. Por tanto, cuando se cumple que $n-f=0$ se tiene que todos los segmentos tienen igual longitud. Estos comportamientos se pueden observar en las Figuras 3.3, 3.4.

Tessellation Evaluation Shader

El Tessellation Evaluation Shader (TES) [13] es la etapa opcional que se encuentra entre el generador de primitivas de teselación y el geometry shader. Su función es la de coger los resultados obtenidos en la etapa anterior y computar las posiciones interpoladas y otros datos vértice a vértice a partir de ellos.

El TES obtiene del generador de primitivas de teselación un parche abstracto, así como datos de los vértices para todo el parche, junto con otros datos, provenientes del TCS. Cada invocación a este shader produce un vértice particular y es invocado una vez por cada vértice en el parche abstracto.

Esta es la etapa donde el programador implementa el algoritmo que se usa para computar las nuevas posiciones, normales, coordenadas de texturas, etc. Como se ha expuesto antes, este shader es el que determina si ocurrirá o no la etapa de generación de primitivas de teselación, puesto que solo se ejecutará si existe un TES activo.

El TES, en caso de ser utilizado, debe especificar el tipo de primitiva que servirá como entrada al geometry shader. Este tipo puede ser puntos, isolíneas, triángulos o cuadriláteros.

3.1.3. Geometry Shader

El Geometry Shader [14] es un programa escrito en GLSL que corresponde a la etapa del pipeline programable que se encuentra entre el TES o el Vertex Shader (dependiendo de si existe o no teselación) y la etapa fija de post-procesado de vértices. Este shader es opcional y no es necesaria su utilización.

Las invocaciones a este shader toman como entrada una única primitiva geométrica y puede dar como salida cero o más primitivas, aunque existe un límite de primitivas que se pueden generar en cada invocación, dependiendo de la implementación. Los shaders geométricos están diseñados para aceptar como entrada una primitiva específica y dar como salida otra.

Sus usos varían bastante, pudiéndose utilizar como una manera de amplificar la geometría, sirviendo como una especie de teselación, así como para realizar un renderizado por capas o incluso para la realización de tareas de cómputo en la GPU.

Entre las primitivas de entrada aceptadas por el geometry shader se encuentran las siguientes:

Entrada	Primitiva	Parámetro TES	Vértices
`points`	`GL_POINTS`	`point_mode`	1
`lines`	`GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_LIST`	`isolines`	2
`lines_adjacency`	`GL_LINES_ADJACENCY`, `GL_LINE_STRIP_ADJACENCY`	`N/A`	4
`triangles`	`GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`	`triangles`, `quads`	3
`triangles_adjacency`	`GL_TRIANGLES_ADJACENCY`, `GL_TRIANGLE_STRIP_ADJACENCY`	`N/A`	6

Tabla 3.1: Primitivas de entrada al Geometry Shader

Las primitivas de salida pueden ser únicamente alguna de las siguientes:

- `points`
- `line_strip`
- `triangle_strip`

Los shaders geométricos pueden generar tantos vértices como permita el límite de implementación. Para ello, el programador genera los valores que necesite para el nuevo vértice y, una vez estos valores sean correctos, una llamada a la función `EmitVertex()` produce el vértice deseado. Una vez llamada esta función, los valores escritos para el vértice son reseteados, teniendo que volver a escribirlos para generar otro vértice.

De igual modo, para generar una primitiva, debemos especificar del modo anterior todos los vértices que forman esa primitiva y posteriormente llamar a la función `EndPrimitive()`. De esta forma, si se desea generar más de una primitiva, se deben especificar los vértices que forman la primera, llamar a `EndPrimitive()`, generar los vértices que forman la segunda y llamar de nuevo a `EndPrimitive()` para generar la segunda primitiva.

3.1.4. Fragment Shader

El Fragment Shader [15] es la etapa posterior a la rasterización. Por cada uno de los píxeles cubiertos por una primitiva, se genera un fragmento. Cada uno de estos fragmentos tiene una posición en la espacio de ventana, así como otros valores procedentes de la etapa de procesamiento de vértices.

La salida del fragment shader consta de un valor de profundidad, un posible valor de plantilla (que no es modificado por el shader) y cero o más valores de color para ser potencialmente escritos en los buffers del frame buffer actual. Estos shaders toman como entrada un único fragmento, producido por el rasterizador, y dan como salida otro único fragmento.

Técnicamente, la utilización de estos shaders es también opcional, puesto que de no utilizarlo, los valores de color del fragmento de entrada quedarán indefinidos, pero los valores de profundidad y plantilla en la salida serán los mismos que los de entrada. Esto puede ser interesante en el caso de solo estar interesados en los valores de profundidad computados por el sistema en lugar de otro valor calculado por el programador.

Este shader también tiene operaciones especiales no presentes en los otros tipos de shader, como puede ser la instrucción `discard`, cuyo objetivo es descartar los valores de salida generados durante la ejecución del shader para un fragmento en concreto, haciendo que este fragmento no pase a las siguientes etapas del pipeline. Esto puede ser útil para descartar fragmentos cuyos valores generados en la ejecución se queden fuera de unos límites impuestos por el programador.

3.2. Uso en Visualización Científica

Capítulo 4

Aplicación

En este capítulo se presenta la aplicación que se ha desarrollado para ilustrar y poner en práctica las ideas aprendidas.

Capítulo 5

Conclusiones y Trabajo Futuro

Bibliografía

[1] Thomas A. Defanti and Maxine D. Brown. Visualization in scientific computing. volume 33 of *Advances in Computers*, pages 247 – 307. Elsevier, 1991. doi: https://doi.org/10.1016/S0065-2458(08)60168-0. URL http://www.sciencedirect.com/science/article/pii/S0065245808601680.

[2] Matthew W. Rohrer. Seeing is believing: The importance of visualization in manufacturing simulation. In *Proceedings of the 32Nd Conference on Winter Simulation*, WSC '00, pages 1211–1216, San Diego, CA, USA, 2000. Society for Computer Simulation International. ISBN 0-7803-6582-8. URL http://dl.acm.org/citation.cfm?id=510378.510552.

[3] B. H. McCormick, T. A. DeFanti, and M. D. Brown. Visualization in scientific computing. *Computer Graphics*, 20(6), 1987.

[4] The Khronos Group Inc. OpenGL Website. https://www.opengl.org/, . Accessed: 23/07/2019.

[5] Shreiner, Dave, and The Khronos OpenGL ARB Working Group. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. Addison-Wesley Professional, 7th edition, 2009. ISBN 0321552628, 9780321552624.

[6] The Khronos Group Inc. About the OpenGL ARB. https://www.opengl.org/archives/about/arb/, . Accessed: 19/07/2019.

[7] The Khronos Group Inc. Primitivas - OpenGL wiki. https://www.khronos.org/opengl/wiki/Primitive, . Accessed: 26/07/2019.

[8] Microsoft. Microsoft Website. https://www.microsoft.com/es-es. Accessed: 23/07/2019.

[9] The Khronos Group Inc. GLSL Website. https://www.khronos.org/opengl/wiki/Core_Language_(GLSL), . Accessed: 24/07/2019.

[10] The Khronos Group Inc. Vertex Shader - OpenGL wiki. https://www.khronos.org/opengl/wiki/Vertex_Shader, . Accessed: 24/07/2019.

[11] The Khronos Group Inc. Tessellation Control Shader - OpenGL wiki. https://www.khronos.org/opengl/wiki/Tessellation_Control_Shader, . Accessed: 24/07/2019.

[12] The Khronos Group Inc. Tessellation Primitive Generator - OpenGL wiki. https://www.khronos.org/opengl/wiki/Tessellation#Tessellation_primitive_generation, . Accessed: 26/07/2019.

[13] The Khronos Group Inc. Tessellation Evaluation Shader - OpenGL wiki. https://www.khronos.org/opengl/wiki/Tessellation_Evaluation_Shader, . Accessed: 25/07/2019.

[14] The Khronos Group Inc. Geometry Shader - OpenGL wiki. https://www.khronos.org/opengl/wiki/Geometry_Shader, . Accessed: 26/07/2019.

[15] The Khronos Group Inc. Fragment Shader - OpenGL wiki. https://www.khronos.org/opengl/wiki/Fragment_Shader, . Accessed: 26/07/2019.

Apéndice A

El lenguaje GLSL