Introducción a OpenGL Shading Language (GLSL)

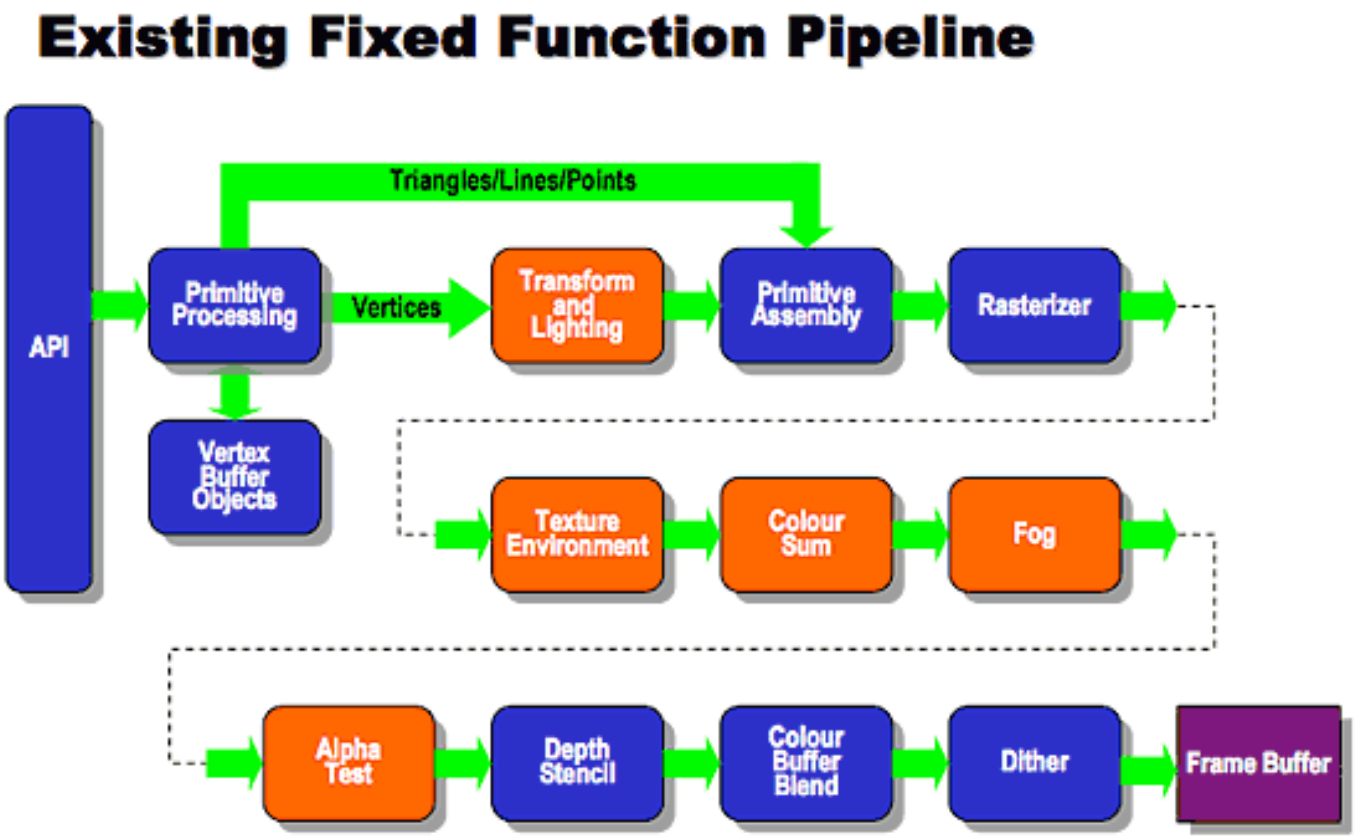
Guido Sánchez

November 20, 2007

Fixed Pipeline

- Todas las versiones de OpenGL hasta la 1.5 inclusive se basan en un funcionamiento fijo del pipeline.
- El usuario puede controlar varios parámetros, pero no la funcionalidad y el orden de procesamiento.
- La única manera de modificar OpenGL es definiendo extensiones.
- Pero únicamente los fabricantes pueden implementar extensiones.
- Utilizar alguna extensión de determinado fabricante implica aprender el lenguaje de cada fabricante y arriesgarse a depender de esa extensión (y de ese hardware).

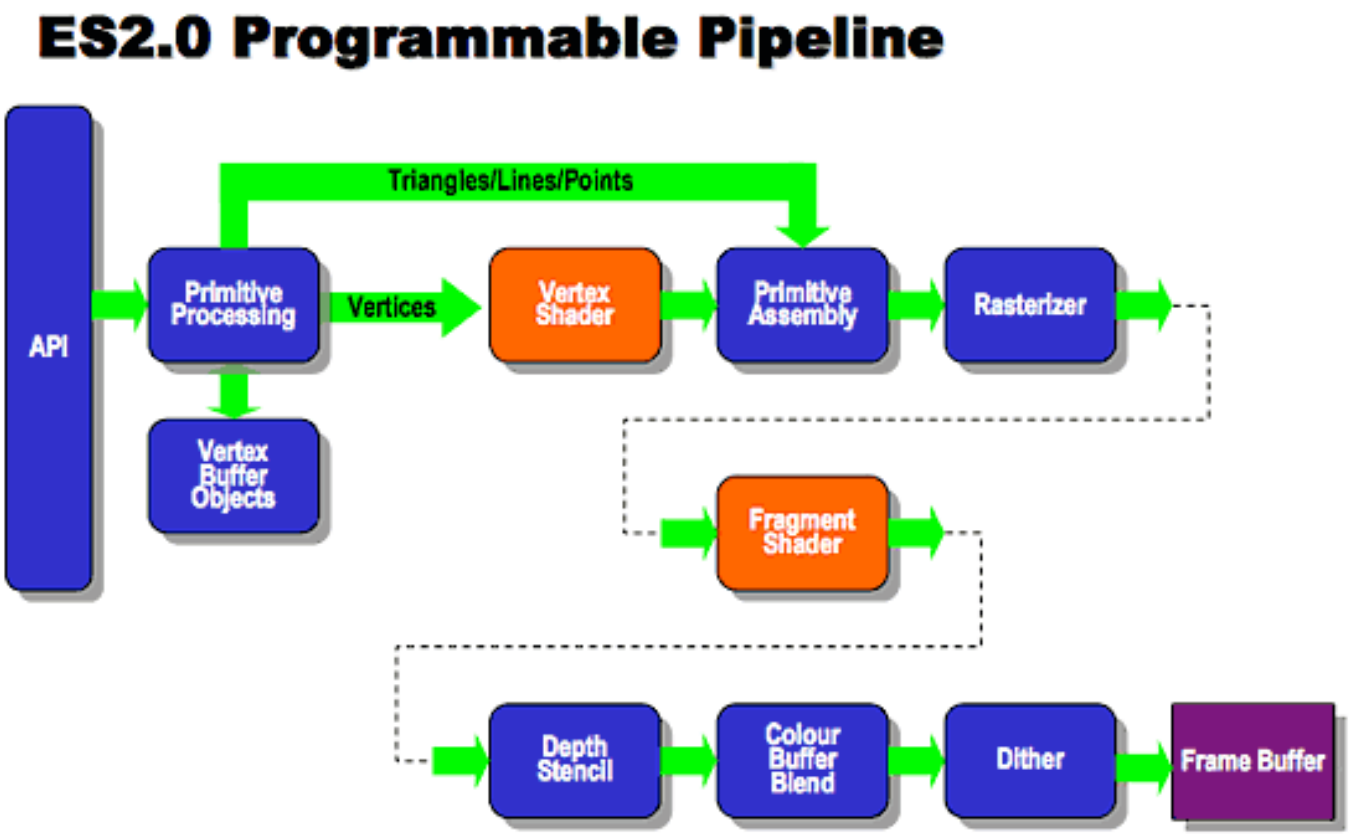
Diagrama del Fixed Pipeline



Programmable Pipeline

- A partir de OpenGL 2.0 se abren las puertas a la programación del pipeline, posibilitando programar la parte de procesamiento de vértices (vertex) y fragmentos (fragment).
- OpenGL Shading Language permite especificar como procesar lo que ocurre en el vertex y fragment processing.
- El código GLSL que se ejecuta en un procesador OpenGL se denomina SHADER.
- Como se definieron dos procesadores programables, tenemos dos shaders: el VERTEX SHADER y el FRAGMENT SHADER.
- Los desarrolladores pueden implementar sus propios algoritmos de renderizado utilizando lenguaje de alto nivel.

Diagrama del Programmable Pipeline



Características de GLSL

- GLSL es un lenguaje procedural de alto nivel.
- Desde OpenGL 2.0 es parte del estándar OpenGL.
- Se utiliza el mismo lenguaje, con unas pequeñas diferencias tanto para vertex como para fragment shaders.
- Soporta operaciones con vectores y matrices y tiene funciones "component wise".

Por qué escribir un Shader?

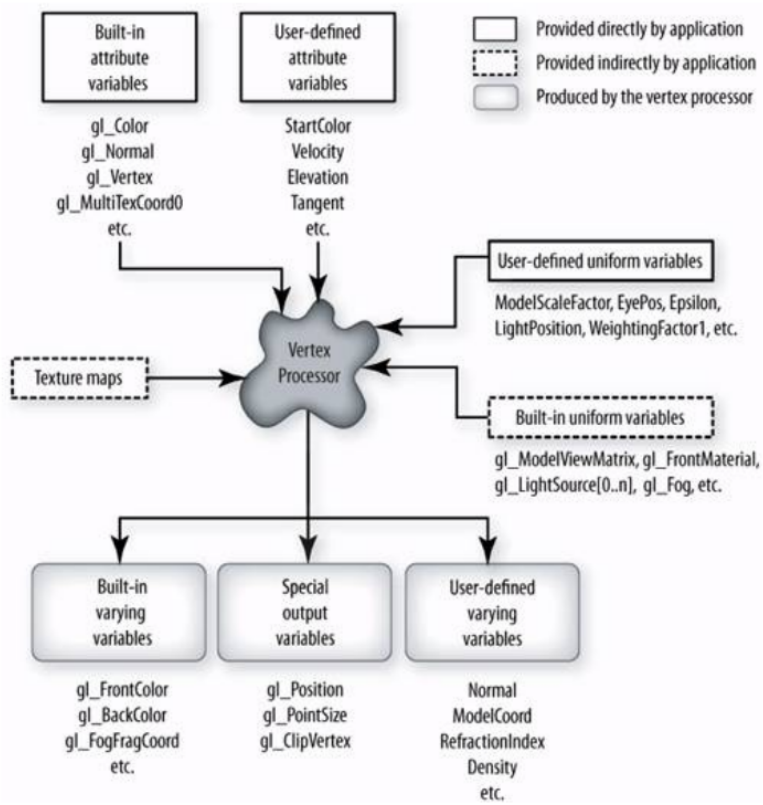
- Porque a través del API de OpenGL no se puede cambiar la manera en que opera el Pipeline Gráfico ni el orden de las operaciones.
- Si, por ejemplo, se frustró porque OpenGL no le permitía definir la manera en que los cálculos de iluminación se realizan por vértice en vez de por fragmentos.
- O si se encontró con alguna limitación del modelo tradicional de renderizado de OpenGL.

Vertex Processor

Opera sobre los valores de los vértices y sus datos asociados.
Generalmente realiza:

- Transformaciones de vértices.
- Transformación de la normal y normalización.
- Generación y transformación de coordenadas de texturas.
- Iluminación.
- Aplicación del color de materiales.

E/S del Vertex Processor

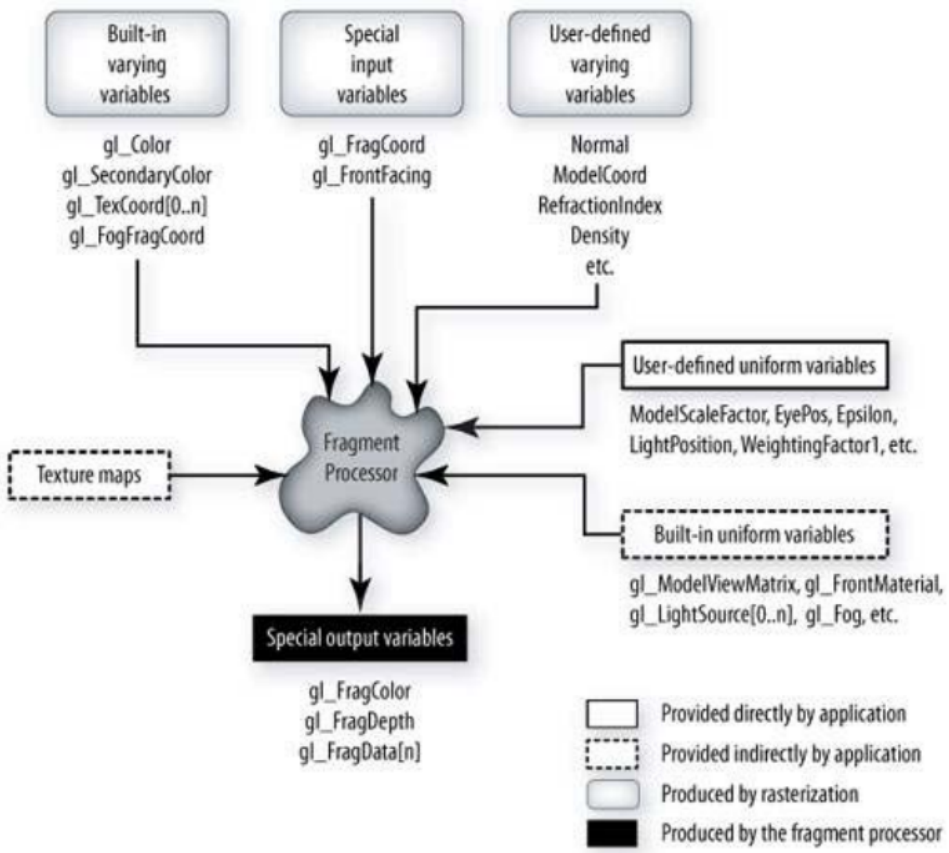


Fragment Processor

Los fragmentos son estructuras de datos por pixel que son creados por la resterizacion de primitivas gráficas. Opera con valores de fragmentos y sus datos asociados. Generalmente realiza:

- Operaciones en valores interpolados.
- Acceso a texturas.
- Aplicación de texturas.
- Niebla.
- Suma de colores.

E/S del Fragment Processor



Descripción básica del lenguaje

- Basado en la sintaxis de ANSI C.
- El punto de entrada de un shader es la funcion `void main();` cuyo cuerpo se delimita con llaves. Las constantes, operadores, identificadores, expresiones y declaraciones, el control de flujo para lazos, if-then-else y las llamadas a funciones son básicamente identicos a C.
- Se añaden tipos de datos vectoriales y matriciales, tales como: **vec2** (dos float), **vec3**, **vec4**, **mat2**, **mat3**, **mat4**.

Descripción básica del lenguaje

- *Samplers* para acceder a texturas. **Sampler1D** y **Sampler2D** para texturas en 1D y 2D respectivamente.
- Calificativos para especificar que tipo de entrada o salida realiza una variable: **attribute**, **uniform** y **varying**.
  - **attribute**: comunica un valor que cambia frecuentemente, desde la aplicación al vertex shader.
  - **uniform**: comunica un valor que no cambia frecuentemente, desde la aplicación a cualquier shader.
  - **varying**: comunica un valor interpolado (resultado de rasterización de primitivas) desde el vertex shader al fragment shader.
- Las variables predefinidas comienzan con **"gl_"**. Por ejemplo: **gl_ModelViewMatrix**, **gl_LightSource[i]**, **gl_Fog.Color**.

Funciones incorporadas

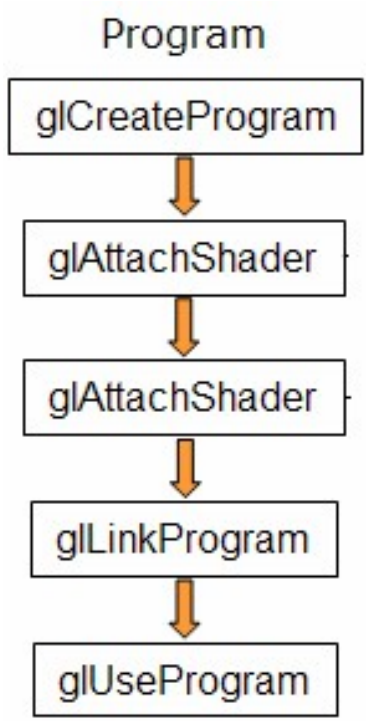
- Trigonométricas: seno, coseno, tangente, etc.
- Exponenciales: potencia, logaritmo, raiz cuadrada, etc.
- Floor, ceiling, parte fraccionaria, módulo, etc.
- Geométricas: distancia, producto punto, producto cruz, normalización, etc.
- Racionales y vectorcomponent-wise: mayor que, menor que, igual a, etc.
- Funciones especializadas del fragment shader para calcular las derivadas y estimar el ancho de filtros para antialiasing.
- Funciones para acceder a los valores de las texturas en memoria.
- Y más...

Hardware

Para correr un programa que utilice shaders se necesita una GPU que acepte el lenguaje de shading. Casi todas las tarjetas gráficas desde la GeForce3 soportan shaders. Se necesita al menos una Nvidia GeForce 5200 o una ATI 9500 para trabajar correctamente con OpenGL 2.x. Asegurese que los drivers de su GPU estén actualizados, el soporte OpenGL y el compilador de Shaders estan integrados en los drivers.

Código OpenGL y C

GLSL necesita un programa OpenGL. Tambien hay que activar el lenguaje de Shading. A continuación se presenta un esquema con los pasos necesarios para activar los shaders.



Ejemplos de GLSL

- Iluminación para "Cartoonist Rendering" con vertex y fragment shaders.
- Uso de "Shader Designer" para editar y testear shaders. Shader Designer se obtiene de http://www.typhoonlabs.com/.

Bibliografía útil

- OpenGL Shading Language, Second Edition by By Randi J. Rost.
- GLSL Tutorial, http://www.lighthouse3d.com/opengl/glsl/.
- NeHe Productions, http://nehe.gamedev.net/.