3.PIPELINE DE RENDERIZADO Y SHADERS

Computación Gráfica
Ing. José Roque Román Guadarrama 2020-2

CONCEPTOS BÁSICOS

Antes de empezar es importante tener claro los siguientes conceptos:

- El pixel (picture element) es la unidad mínima de representación gráfica.
- El frame buffer se caracteriza por su resolución y por su profundidad.
- La resolución es la cantidad total de pixeles que puede mostrar el dispositivo dada por

$$R_T = R_H \times R_V$$

  medidas en pixeles..
- La profundidad es el numero de bits que utilizamos para guardar la información de cada pixel . Este número dependerá de la cantidad de colores que deseemos mostrar en nuestra aplicación. Tipicamente si queremos "color real" necesitaremos ser capaces de mostrar 16,7 millones de colores simultáneamente que es la capacidad aproximadada de nuestro sistema visual.

CONCEPTOS BÁSICOS

Ejemplo: este caso y suponiendo una resolución de 800 x 600 pixeles en pantalla necesitaremos:

```
800 píxeles/fila x 600 filas x 24 bits/píxel = 1.37 Megabytes
                de memoria para el frame buffer
```

Dado que el color real implica 256 posibles valores de rojo, 256 de verde y 256 de azul por pixel y esto implica un byte/pixel para cada una de estas componentes, es decir, 3 byte por pixel.

INTRODUCCIÓN

El proceso completo de visualización de una escena 3D se conoce como pipeline de renderizado.

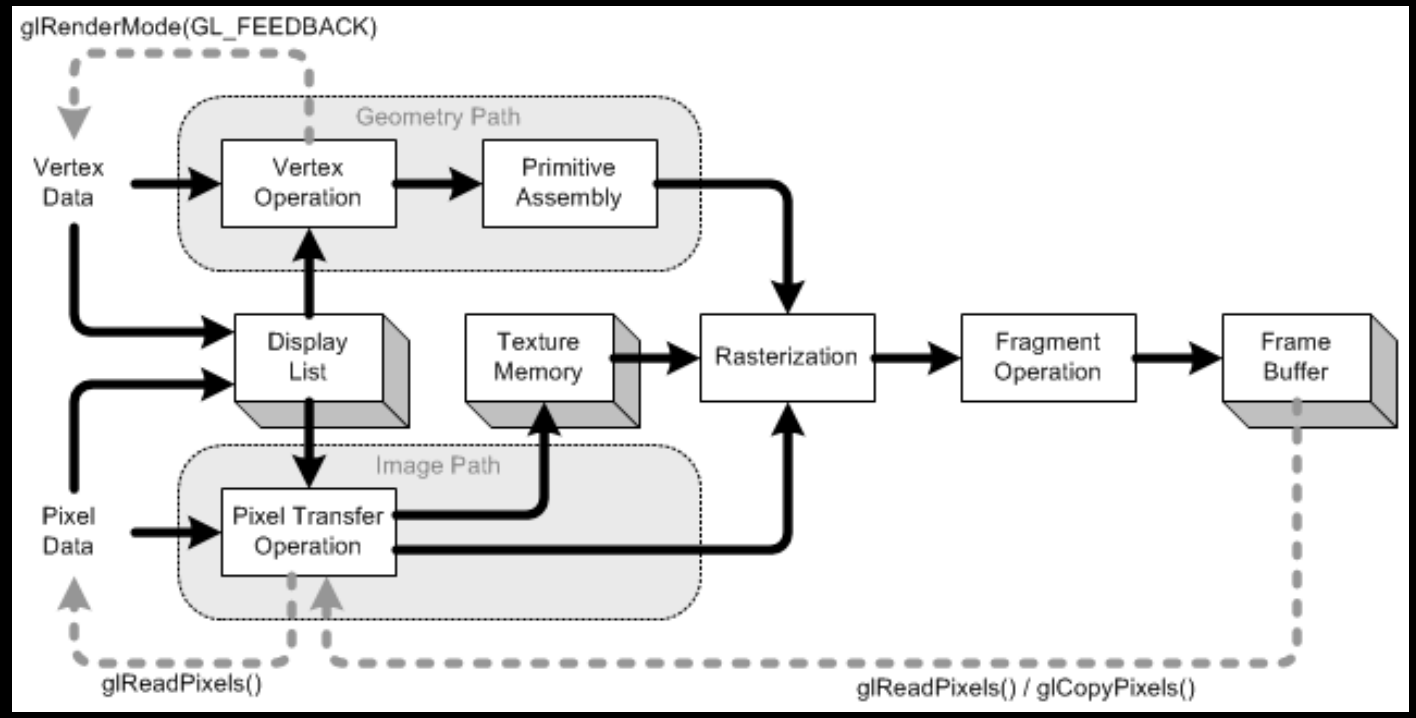
En gráficos 3D por computadora, el Pipeline de renderizado (rendering pipeline)se refiere comúnmente a la renderización basada en la implementación de hardware de gráficos.

Típicamente recibe la representación de una escena tridimensional como entrada y genera una imagen en dos dimensiones como salida.

OpenGL y Direc3D son dos estándares gráficos que proporcionan pipeline de renderizado y permiten además la programación de shaders para modificar su comportamiento

PIPELINE OPENGL 3.3 Y 4

- Vertex Specification
- Vertex Shader
- Tessellation
- Geometry Shader
- Vertex Post-Processing
- Primitive Assembly
- Rasterization
- Fragment Shader
- Per-Sample Operations



ETAPAS QUE LO COMPONEN

Vertex Specification:

- Posiciones de Vértices*
- Vertex Array Objects(VAO)
- Vertex Buffer Objects(VBO)
- Attribute Pointers: Shader*
- Se genera VAO ID y VBO ID
- Se hace Bind del VAO y del VBO
- Se asignan datos al VBO
- Se definen Attribute Pointers
- Se habilitan Attribute Pointers
- Se hace unbind del VAO y VBO
- Dibujo:
  - Se activa el Shader Program
  - Se hace bind del VAO al objeto
  - Se llama g glDrawArrays para continuar

Vertex Shader

- Obligatorio
- Manejo de Vértices
- gl_Position almacena datos
- Se pueden especificar outputs adicionales:

```
#version 330
layout (location =0) in vec3 pos;
void main()
{
        gl_Position=vec4(pos,1.0f);
}
```

ETAPAS QUE LO COMPONEN

Tesellation
- Dividir data en pequeñas primitivas
- A partir de OpenGL 4.0
- Gran nivel de detalle dinámico

Geometry Shader
- Grupos de vértices
- Instancias de las primitivas
- Puede recibir datos para modificar datos las primitivas
- Puede modificar las primitivas

Vertex Post-Processing
- Clipping
- Primitive Assembly
  - Vértices generan primitivas
  - Face culling

Rasterización
- Convierte primitivas en Fragmentos: datos de cada pixel
- Se interpolan los fragmentos de acuerdo a su posición relativa a cada vértice

ETAPAS QUE LO COMPONEN

Fragment Shader

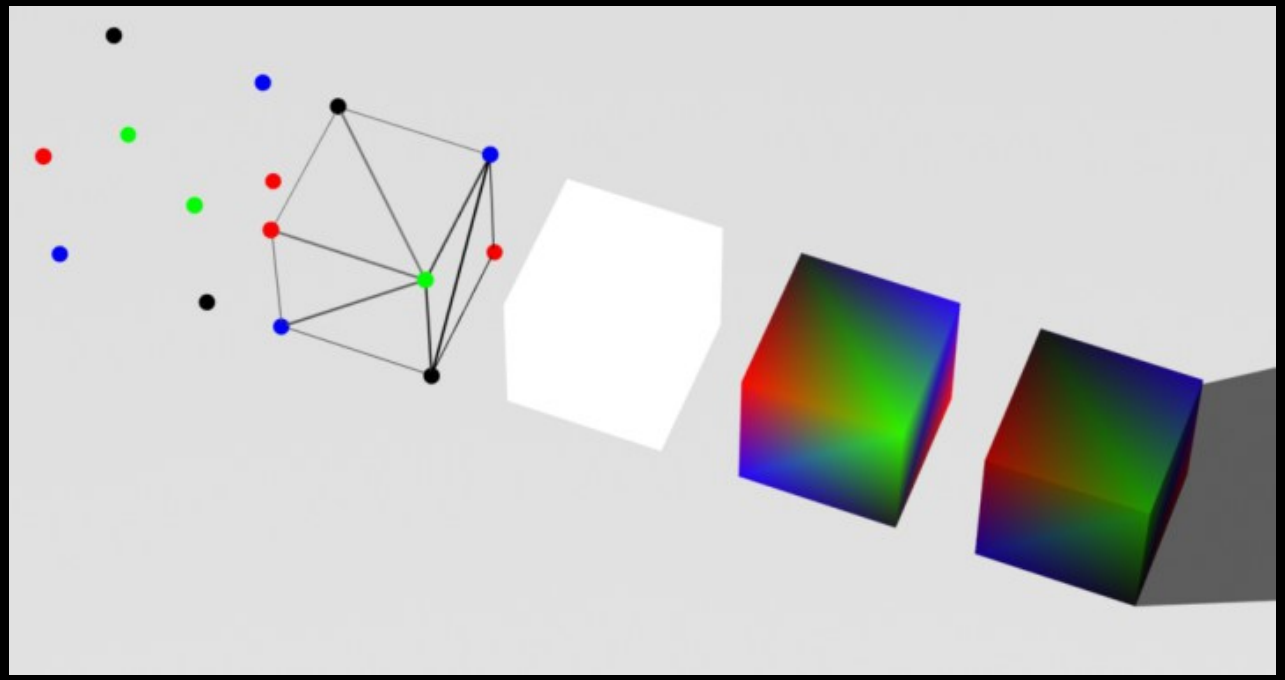
- Maneja datos para cada fragmento
- Opcional pero de muy frecuente uso
- Color del fragmento
- Los programas más simples manejan Vertex Shader y Fragment Shader

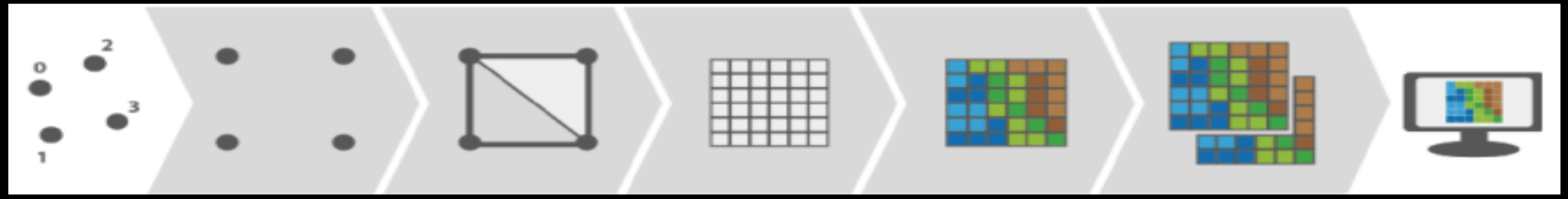
```
#versión 330
out vec4 color;
void main()
{
color=vec4(0,0,0,0,1,0,1,0);
}
```

Per Sample Operations

- Tests para ver si se dibuja un fragmento
- Depth Test.
- Color Blending.
- Datos de fragmentos asignados al FrameBuffer
- SwapBuffer

PIPELINE GRÁFICO





CREANDO UN PROGRAMA DE SHADER

- Se crea un programa vacío
- Se crea shader vacío
- Se añade el shader source al shader
- Se compila el shader
- Se liga el shader al programa
- Se linkea el programa
- Se valida el programa