

CS6533/CS4533 Lecture 5

Slides/Notes

**Programmable Shaders and Shader-Based
OpenGL
(Handout, Ch 1, 6, Appendix A)**

By Prof. Yi-Jen Chiang
CSE Dept., Tandon School of Engineering
New York University

CS6533/CS4533 Handout: Programmable Shaders and Shader- Based OpenGL

By Yi-Jen Chiang

CSE Dept., Tandon School of Engineering
New York University

Programmable Shaders and Shader-Based OpenGL: Overview

Old OpenGL (before version 3.1) v.s. **Shader-Based OpenGL**
(version 3.1 and later) --- 2 major differences

1. Standard way of rendering geometric primitives (points, polygons, etc.):

Immediate mode

```
display(){
```

```
...
```

```
glBegin()
```

```
glVertex()
```

```
...
```

```
glEnd()
```

**Repeat for each
frame:**

the geometric data of
all objects are sent to
GPU for rendering

Slow

Vertex buffer objects (VBO)

Initially, data of each geom.
obj is organized into a VBO
and **sent to GPU once** then
stored in GPU.

In each frame, send **a few
transf. matrices** to GPU to
operate on the stored VBOs

**Much
Faster**

*** Data transfer between CPU & GPU is the speed bottleneck!**

Programmable Shaders and Shader-Based OpenGL: Overview (cont.)

Old OpenGL (before version 3.1) v.s. **Shader-Based OpenGL** (version 3.1 and later) --- 2 major differences

2. Processing Pipeline:

Fixed-Function Commands

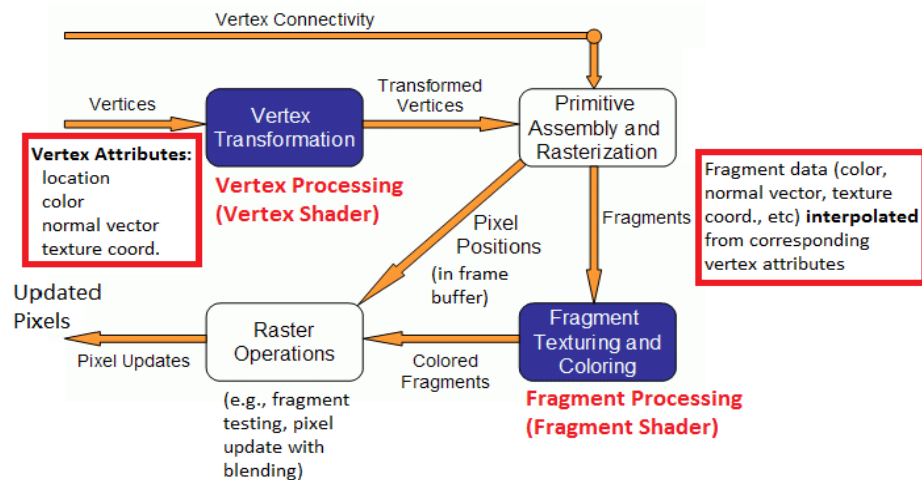
(fixed functions, software support)

Programmable Shaders

(programmable GPU, hardware support)

Much more flexible & powerful (for tasks done, effects achieved, etc) and much faster

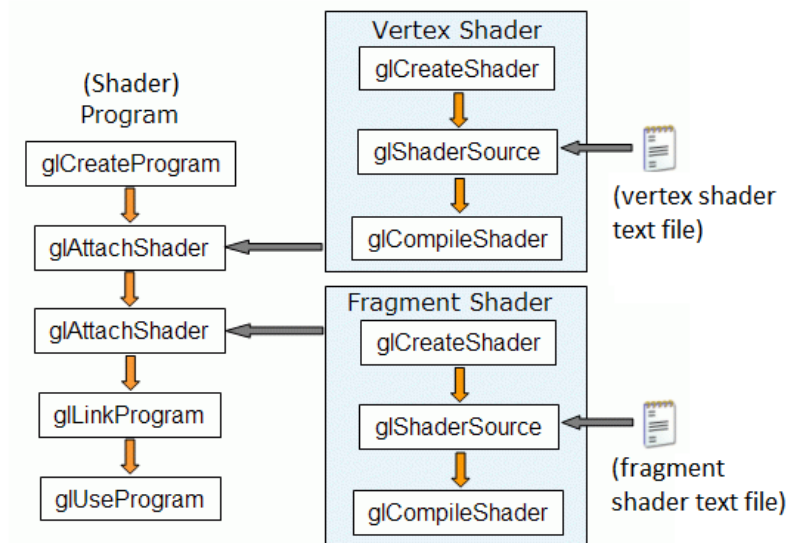
OpenGL Processing Pipeline



* Programmable shaders/GPU language support: assembly -> high-level (Cg -> GLSL)

* **OpenGL + GLSL = Shader-Based OpenGL**

OpenGL Commands to Set Up Shaders (inside an OpenGL Program) ⁵



* Each shader is read & compiled & linked **at run time** of an OpenGL program.

Introduction to GLSL ⁶

- Very similar to C, but functions can not make recursive calls (i.e., **no recursions**).
- Data types: int, float, etc. **New:** vec2, vec3, vec4, mat2, mat3, mat4 (e.g., mat4: 4 x 4 matrix, each element is a float).
- E.g., vec4 t;

t.x = t.r = t.s = t[0]

Position: (x, y, z, w)

t.y = t.g = t.t = t[1]

Color: (r, g, b, a)

t.z = t.b = t.p = t[2]

Texture Coord.: (s, t, p, q)

t.w = t.a = t.q = t[3]

Note: Texture coord. are (s, t, r, q) but r is already used in (r, g, b, a)

→ Replace r with p, and thus (s, t, **p**, q).

- Each vertex/fragment shader must have one `main()` function.
- **Vertex (frag.)** shader is executed for each **vertex (frag.)**.
- Shader min output: vertex: **gl_Position**; frag.: **color** of current frag.

GLSL: Variable Types

1. **Attribute Variables:** Global, read-only, in vertex shader.
 - * For **vertex attributes**. Only used in **vertex shader**.
 - * Declared as **"in"** in vertex shader (e.g., in vec4 vPosition;).
 - * Variable values are **set up/retrieved** from **vertex buffer obj.** (see Handout sample code for more details).
2. **Uniform Variables:** Global, read-only, used in **both** shaders.
 - * For quantities/items whose values **stay the same across different vertices/fragments** (e.g., uniform mat4 model-view; uniform float time;).
 - (See Handout sample code for more details).

GLSL: Variable Types (cont.)

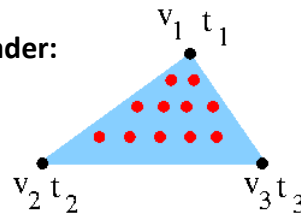
3. Varying Variables:

- * To communicate between vertex & fragment shaders.
- * Declared as **"out"** (**output/writing**) in vertex shader, and declared as **"in"** (**input, read-only**) in fragment shader, with the **same variable name**.

E.g. In vertex shader: | In fragment shader:

out float t;

| in float t;



- * Assign value in vertex shader, and then **interpolate from vertices** to obtain the **value at each fragment** as the **input** to the **fragment shader** for that fragment.