CSC 4356
Interactive Computer Graphics
Lecture 17: Shading Language

Jinwei Ye

http://www.csc.lsu.edu/~jye/CSC4356/

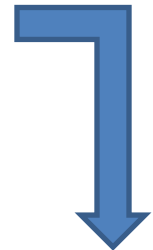
Tue & Thu: 10:30 - 11:50am
218 Tureaud Hall

Why Use Shading Language

- GPU has become increasingly more powerful
- Programming powerful hardware with assembly code is hard
- Most GPUs supports programs more than 1,000 assembly instructions long
- Programmers need the benefits of a high-level language:
  - Easier programming
  - Easier code reuse
  - Easier debugging

Assembly

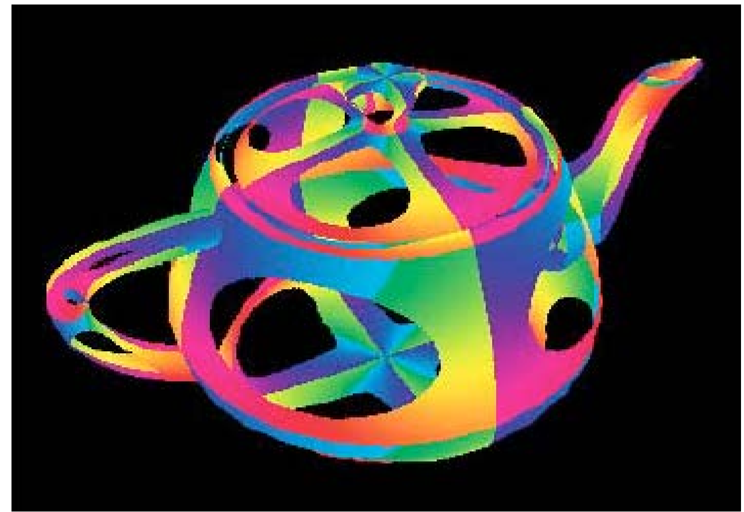
```
…
DP3 R0, c[11].xyzx, c[11].xyzx;
RSQ R0, R0.x;
MUL R0, R0.x, c[11].xyzx;
MOV R1, c[3];
MUL R1, R1.x, c[0].xyzx;
DP3 R2, R1.xyzx, R1.xyzx;
RSQ R2, R2.x;
MUL R1, R2.x, R1.xyzx;
ADD R2, R0.xyzx, R1.xyzx;
DP3 R3, R2.xyzx, R2.xyzx;
RSQ R3, R3.x;
MUL R2, R3.x, R2.xyzx;
DP3 R2, R1.xyzx, R2.xyzx;
MAX R2, c[3].z, R2.x;
MOV R2.z, c[3].y;
MOV R2.w, c[3].y;
LIT R2, R2;
...
```

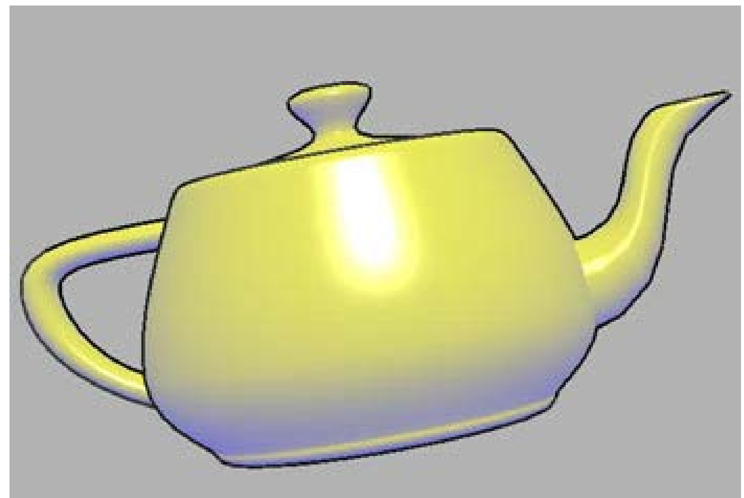


Shading Language

```
float3 cSpecular = pow(max(0, dot(Nf, H)), phongExp).xxx;
float3 cPlastic = Cd * (cAmbient + cDiffuse) + Cs * cSpecular;
```

Customized Rendering Effect





GPU Shading Languages

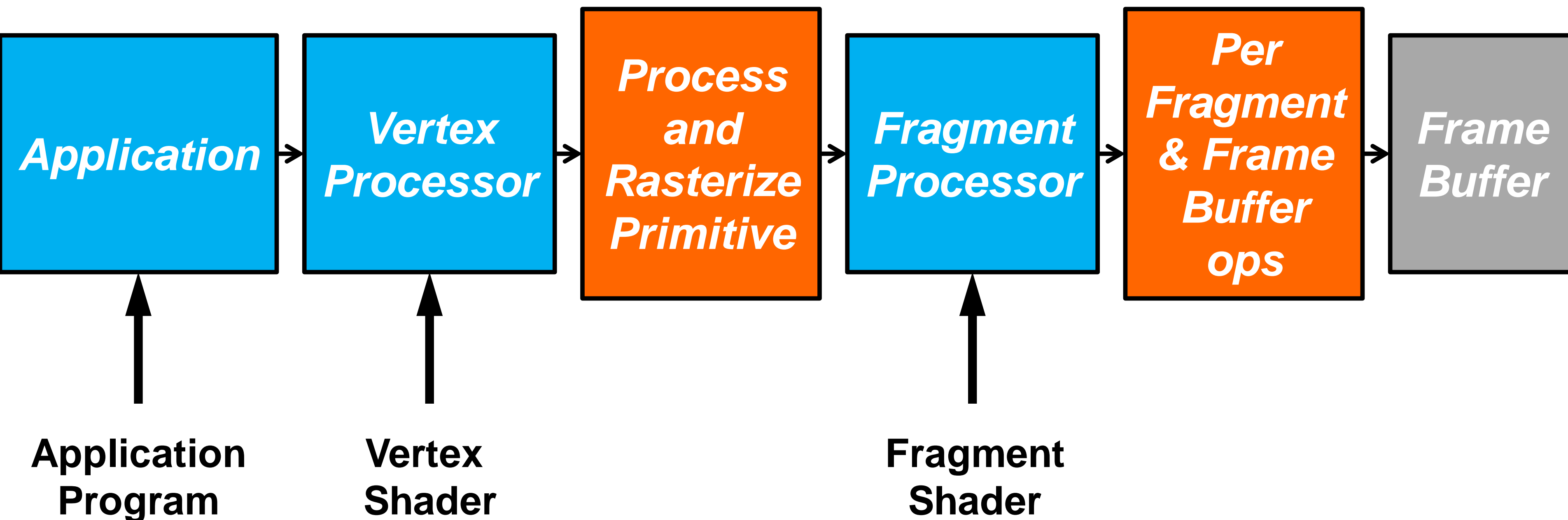
- CG
  - NVIDIA's shading language
  - Works on OpenGL and DirectX
  - Uses hardware profiles that may limit language constructs
- HLSL
  - Shading language used in DirectX
  - Very similar to CG
- GLSL
  - OpenGL's built in shading language
- Sh
  - A C++ library rather than a language
  - Can cross-compile to the GPU

OpenGL Shading Language (GLSL)

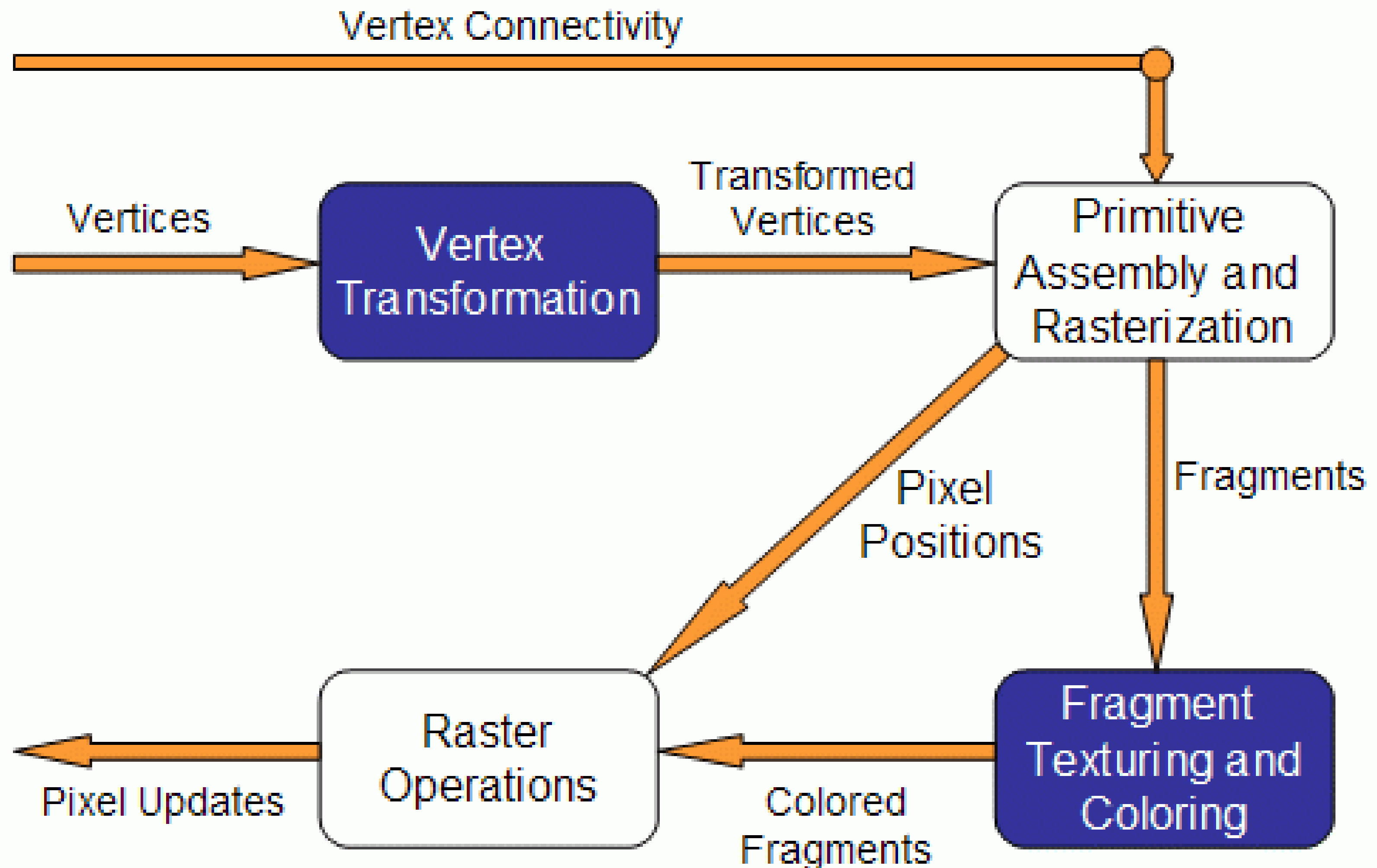
- A C-like language and incorporated into OpenGL 2.0
- Used to write vertex shader and fragment shader
- No distinction in the syntax between a vertex shader and a fragment shader
- Platform independent compared to CG

GPU Programmability

- Programmable Processing units
  - Programmable per-Vertex Processors
  - Programmable per-Fragment Processors



Graphics Pipeline



Vertex Transformation

- A vertex is a set of attributes such as its location in space, color, normal, texture coordinates, etc.
- Inputs: individual vertices attributes
- Operations:
  - Vertex position transformation
  - Lighting computations per vertex
  - Generation and transformation of texture coordinates

Primitive Assembly and Rasterization

- Inputs: transformed vertices and connectivity information
- Operations:
  - Clipping against view frustum and back face culling
  - The actual rasterization determines the fragments, and pixel positions of the primitive
- Output:
  - Position of the fragments in the frame buffer
  - Interpolated attributes for each fragment

Fragment Texturing and Coloring

- Input: interpolated fragment information
- A color has already been computed in the previous stage through interpolation, and can be combined with a texel
- Texture coordinates have also been interpolated in the previous stage
- Fog is also applied at this stage
- Output: a color value and a depth for each fragment

Replacing Fixed Functionalities

- Vertex Transformation:
  Vertex shader
- Primitive Assembly and Rasterization:
  Geometry shader
- Fragment Texturing and Coloring:
  Fragment shader

How Does It Work

- You specify vertices as usual
  - Vertex positions, texture coordinates, etc.
  - And some user variables if you want
- The vertex shader modifies/calculates these variables
- Each fragment gets the interpolated values
- The fragment shader can now work on the interpolated values, including the user defined variables

Vertex Shader

- A vertex shader is executed on each vertex triggered by glVertex*()
- Each vertex shader must output the information that the rasterizer needs
  - At a minimum: transforms the vertex position
- The program can access all OpenGL states
  - Current color, texture coordinates, material properties, transformation matrices, etc
- The application can also supply additional input variables to the vertex program

Vertex Program Capabilities

- General processing that a vertex shader can do include:
  - Vertex transformation
  - Normal transformation, normalization and rescaling
  - Lighting
  - Color material application
  - Clamping of colors
  - Texture coordinate generation
  - Texture coordinate transformation
  - Etc.

Vertex Program Capabilities

- The vertex program does NOT do:
  - Perspective divide and viewport mapping
  - Frustum and user clipping
  - Backface culling
  - Two sided lighting selection
  - Polygon mode
  - Etc.

Fragment Shader

- The fragment shader is executed after rasterization and operate on each fragment
  - Per-pixel operations
- Vertex attributes (colors, positions, texture coordinates, etc.) are interpolated across a primitive automatically as the input to the fragment program
- Fragment shader can access OpenGL state, (interpolated) output from vertex program, and user defined variables

Fragment Shader Capabilities

- General processing that a fragment shader can do include:
  - Operations on interpolated values
  - Texture access
  - Texture application
  - Fog
  - Color sum
  - Color matrix
  - Discard fragment
  - Etc.

Fragment Shader Capabilities

- The fragment shader does NOT replace:
  - Scissor
  - Alpha test
  - Depth test
  - Stencil test
  - Alpha blending
  - Etc.

GLSL Data Types

- Three basic data types in GLSL: float, bool, int
  - float and int behave just like in C and bool types can take on the values of true or false
- Vectors with 2,3 or 4 components, declared as:
  - vec2, vec3, vec4
- Square matrices 2x2, 3x3 and 4x4:
  - mat2, mat3, mat4
- A set of special types are available for texture access, called sampler
  - sampler1D, sampler2D, sampler3D, samplerCube
- Arrays can be declared using the same syntax as in C, but can't be initialized when declared
- Structures are supported with exactly the same syntax as C

GLSL Variables

- Declaring variables in GLSL is mostly the same as in C

```
float a,b;
int c = 2; // c is initialized with 2
bool d = true; // d is true
```

- Differences: GLSL relies heavily on constructor for initialization and type casting

```
float e = (float)2;// incorrect, requires constructors for type casting
int a = 2;
float c = float(a); // correct. c is 2.0
vec3 f; // declaring f as a vec3
```

- GLSL is pretty flexible when initializing variables using other variables

```
vec3 g = vec3(1.0,2.0,3.0); // declaring and initializing g
vec2 a = vec2(1.0,2.0);
vec2 b = vec2(3.0,4.0);
vec4 c = vec4(a,b) // c = vec4(1.0,2.0,3.0,4.0);
vec2 g = vec2(1.0,2.0);
float h = 3.0;
vec3 j = vec3(g,h);
```

GLSL Variables

- Matrices also follow this pattern

```
mat4 m = mat4(1.0) // initializing the diagonal of the matrix with 1.0
vec2 a = vec2(1.0,2.0);
vec2 b = vec2(3.0,4.0);
mat2 n = mat2(a,b); // matrices are assigned in column major order
mat2 k = mat2(1.0,0.0,1.0,0.0); // all elements are specified
```

- The declaration and initialization of structures is demonstrated below

```
struct dirlight {
vec3 direction;
vec3 color;
};
dirlight d1;
dirlight d2 = dirlight(vec3(1.0,1.0,0.0),vec3(0.8,0.8,0.4));
```

GLSL Variables

- Accessing a vector can be done using letters as well as standard C selectors

```
vec4 a = vec4(1.0,2.0,3.0,4.0);
float posX = a.x;
float posY = a[1];
vec2 posXY = a.xy;
```

- One can the letters x,y,z,w to access vectors components; r,g,b,a for color components; and s,t,p,q for texture coordinates
- As for structures the names of the elements of the structure can be used as in C

```
d1.direction = vec3(1.0,1.0,1.0);
```

GLSL Variable Qualifiers

- Qualifiers give a special meaning to the variable. In GLSL the following qualifiers are available:
  - ***attribute***: per-vertex data values provided to the vertex shader
  - ***uniform***: variables set for the entire primitive, i.e., may not be set inside glBegin()/glEnd()
  - ***varying***: used for interpolated data between a vertex shader and a fragment shader (output for vertex shader but input for fragment shader). Defined on a per vertex basis but interpolated over the primitive for the fragment shader

Built-in Variables

- For ease of programming
- OpenGL state mapped to variables
- Some special variables are required to be written to, others are optional

Built-in Attributes

```
attribute vec4 gl_Vertex;
attribute vec3 gl_Normal;
attribute vec4 gl_Color;
attribute vec4 gl_SecondaryColor;
attribute vec4 gl_MultiTexCoordn;
attribute float gl_FogCoord;
```

Built-in Uniforms

```
uniform mat4 gl_ModelViewMatrix;
uniform mat4 gl_ProjectionMatrix;
uniform mat4 gl_ModelViewProjectionMatrix;
uniform mat3 gl_NormalMatrix;     //transpose of the
                                  //inverse of the upper
                                  //leftmost 3x3 of
                                  //gl_ModelViewMatrix
uniform mat4 gl_TextureMatrix[n];

struct gl_MaterialParameters {
          vec4 emission;
          vec4 ambient;
          vec4 diffuse;
          vec4 specular;
          float shininess;
};

uniform gl_MaterialParameters gl_FrontMaterial;
uniform gl_MaterialParameters gl_BackMaterial;
```

Built-in Uniforms

```
struct gl_LightSourceParameters {
      vec4 ambient;
      vec4 diffuse;
      vec4 specular;
      vec4 position;
      vec4 halfVector;
      vec3 spotDirection;
      float spotExponent;
      float spotCutoff;
      float spotCosCutoff;
      float constantAttenuation
      float linearAttenuation
      float quadraticAttenuation
};

uniform gl_LightSourceParameters gl_LightSource[gl_MaxLights];
```

Built-in Varyings

- For vertex shader

```
varying vec4 gl_FrontColor;
varying vec4 gl_BackColor;
varying vec4 gl_FrontSecondaryColor;
varying vec4 gl_BackSecondaryColor;
varying vec4 gl_TexCoord[];
varying float gl_FogFragCoord;
```

- For fragment shader

```
varying vec4 gl_Color;
varying vec4 gl_SecondaryColor;
varying vec4 gl_TexCoord[];
varying float gl_FogFragCoord;
```

Special Built-ins

- Vertex shader

```
vec4 gl_Position; // must be written
vec4 gl_ClipPosition; // may be written
float gl_PointSize; // may be written
```

- Fragment shader

```
float gl_FragColor; // may be written
float gl_FragDepth; // may be read/written
vec4 gl_FragCoord; // may be read
bool gl_FrontFacing; // may be read
```

Built-in Functions

- Angles & Trigonometry
  - radians, degrees, sin, cos, tan, asin, acos, atan
- Exponentials
  - pow, exp2, log2, sqrt, inversesqrt
- Common
  - abs, sign, floor, ceil, fract, mod, min, max, clamp

Built-in Functions

- Interpolations
  - mix(x,y,a) x*(1.0-a) + y*a
  - step(edge,x) x <= edge ? 0.0 : 1.0
  - smoothstep(edge0,edge1,x)

```
t = (x-edge0)/(edge1-edge0);
t = clamp( t, 0.0, 1.0);
return t*t*(3.0-2.0*t);
```

Built-in Functions

- Geometric
  - length, distance, cross, dot, normalize, faceForward, reflect
- Matrix
  - matrixCompMult
- Vector relational
  - lessThan, lessThanEqual, greaterThan, greaterThanEqual, equal, notEqual, any, all

Vertex Shader Input

- Vertex shader is executed once each time a vertex position is specified
  - Via glVertex or glDrawArrays or other vertex array calls
- Per-vertex input values are called "attributes"
  - Change every vertex
  - Passed through normal OpenGL mechanisms (per-vertex API or vertex arrays)
- More persistent input values are called "uniforms"
  - Can come from OpenGL state or from the application
  - Constant across at least one primitive, typically constant for many primitives
  - Passed through new OpenGL API calls

Vertex Shader Output

- Vertex shader uses input values to compute output values
- *Vertex shader must compute **gl_Position***
  - Mandatory, needed by the rasterizer
- Other output values are called "varying" variables
  - E.g., color, texture coordinates, arbitrary data
  - Will be interpolated across the primitives
  - Defined by the vertex shader
  - Can be of type float, vec2, vec3, vec4, mat2, mat3, mat4, or arrays of these
- Output of vertex processor feeds into OpenGL fixed functionality
  - If a fragment shader is active, output of vertex shader must match input of fragment shader
  - If no fragment shader is active, output of vertex shader must match the needs of fixed functionality fragment processing

Fragment Shader Input

- Output of vertex shader is the input to the fragment shader
  - Compatibility is checked when linking occurs
  - Compatibility between the two is based on **varying variables** that are defined in both shaders and that match in type and name
- Fragment shader is executed for each fragment produced by rasterization
- For each fragment, fragment shader has access to the interpolated value for each varying variable
  - Color, normal, texture coordinates, arbitrary values

Fragment Shader Input

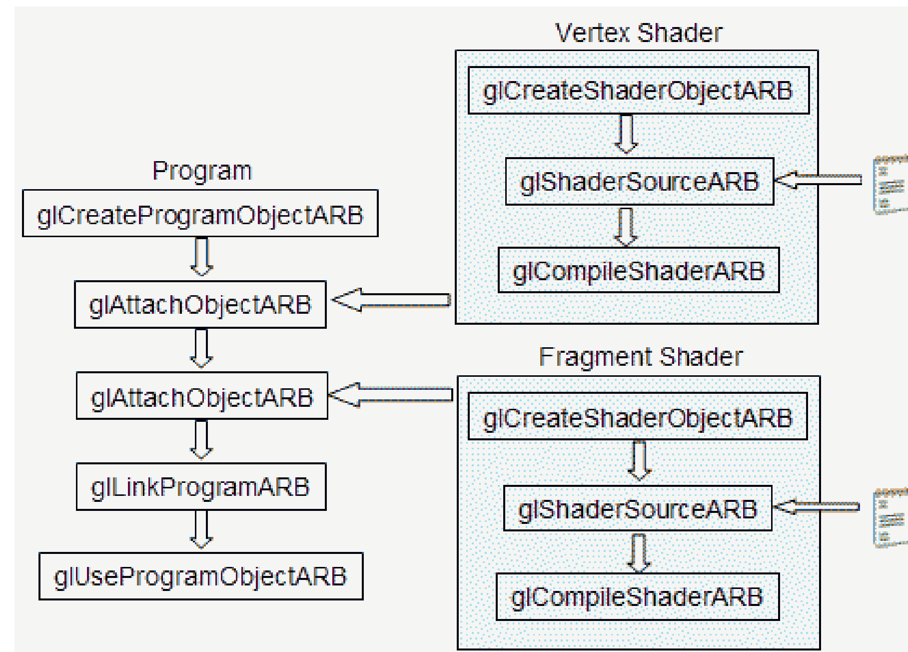
- Fragment shader may access:
  - gl_FrontFacing – contains “facingness” of primitive that produced the fragment
  - gl_FragCoord – contains computed window relative coordinates x, y, z, 1/w
- Uniform variables are also available
  - OpenGL state or supplied by the application, same as for vertex shader
- If no vertex shader is active, fragment shader get the results of OpenGL fixed functionality

Fragment Shader Output

- Output of the fragment processor goes on to the fixed function fragment operations and frame buffer operations using built-in variables
  - gl_FragColor – computed R, G, B, A for the fragment
  - gl_FragDepth – computed depth value for the fragment
  - gl_FragData[n] – arbitrary data per fragment, stored in multiple render targets
  - Values are destined for writing into the frame buffer if all back end tests (stencil, depth etc.) pass
- Clamping or format conversion to the target buffer is done automatically outside of the fragment shader

How to Use Shaders

- Four steps to using a shader
  - Send shader source to OpenGL
  - Compile the shader
  - Create an executable (i.e., link compiled shaders together)
  - Install the executable as part of the current state



Shader Wrapper Code

```
GLhandleARB programObject;
GLhandleARB vertexShaderObject;
GLhandleARB fragmentShaderObject;

unsigned char *vertexShaderSource = readShaderFile(vertexShaderFilename);
unsigned char *fragmentShaderSource = readShaderFile(fragmentShaderFilename);

programObject=glCreateProgramObjectARB();
vertexShaderObject=glCreateShaderObjectARB(GL_VERTEX_SHADER_ARB);
fragmentShaderObject=glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);

glShaderSourceARB(vertexShaderObject,1,(const char**)&vertexShaderSource,NULL);
glShaderSourceARB(fragmentShaderObject,1,(const char**)&fragmentShaderSource,NULL);

glCompileShaderARB(vertexShaderObject);
glCompileShaderARB(fragmentShaderObject);

glAttachObjectARB(programObject, vertexShaderObject);
glAttachObjectARB(programObject, fragmentShaderObject);

glLinkProgramARB(programObject);

glUseProgramObjectARB(programObject);
```

Programming Assignment 3

- PA 3 is posted on our course website
- Due on 11/9/2017, 11:59pm
  - Two weeks
- Implement shaders using a GLSL editor
  - Shader Maker

  http://cgvr.cs.uni-bremen.de/teaching/shader_maker/

Next Time …

- Texture mapping
- Reading:
  - Textbook Chapter 18