OpenGL Shading Lanuage (GLSL)

CSE 781 Winter 2010

Han-Wei Shen

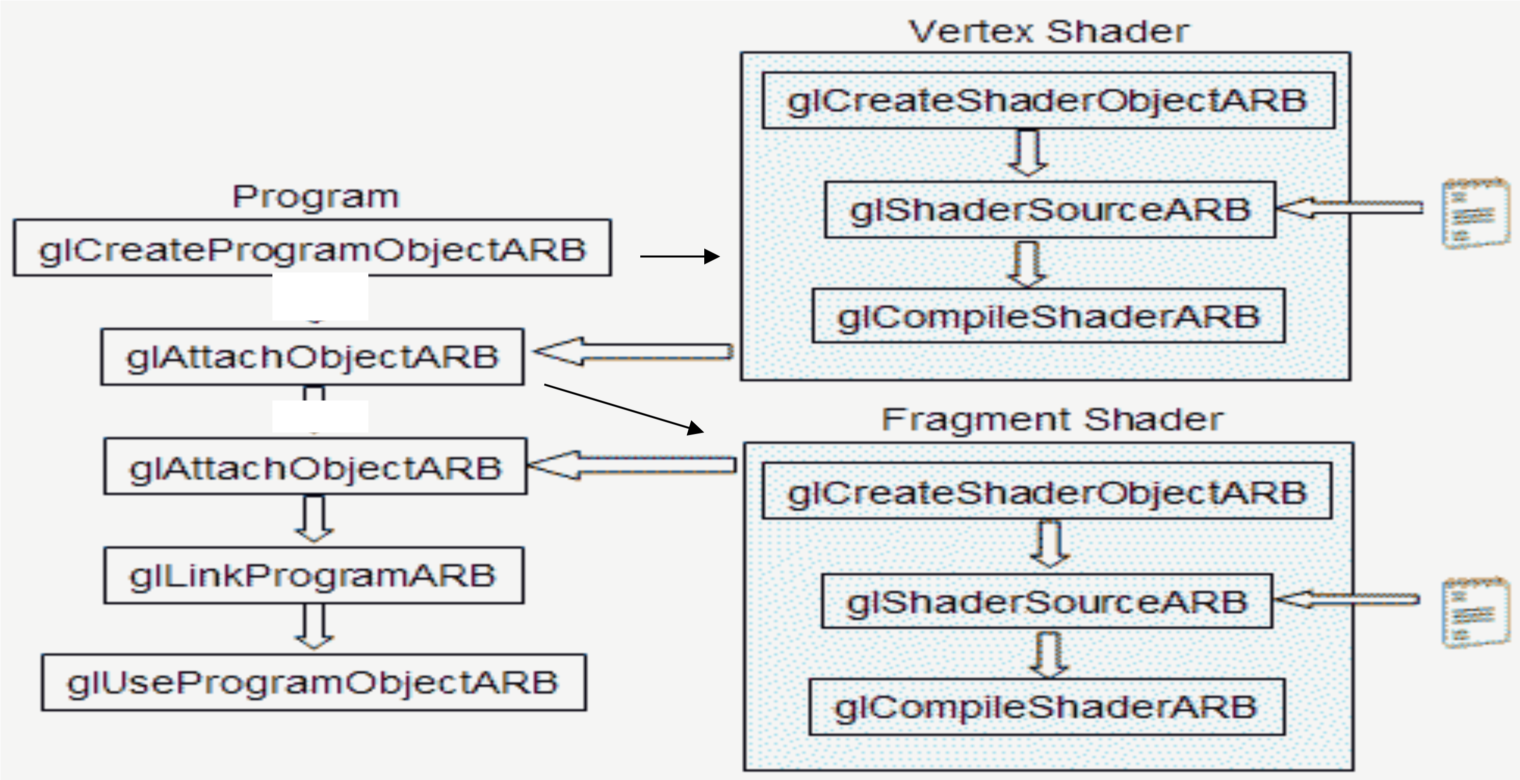
OpenGL Shading Language (GLSL)

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

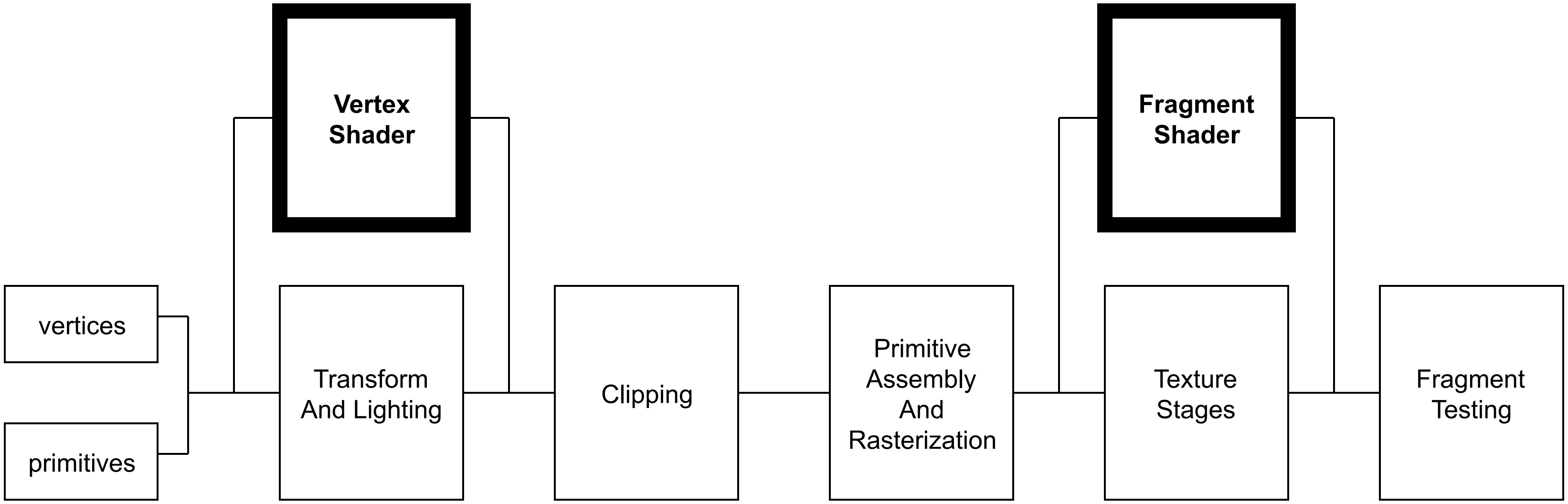
Shader Objects

- Shaders are defined as an array of strings
- 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 current state
- Goal was to mimic C/C++ source code development model

Sequence


Vertex Shader
glCreateShaderObjectARB
glShaderSourceARB
glCompileShaderARB
Program
glCreateProgramObjectARB
glAttachObjectARB
glAttachObjectARB
glLinkProgramARB
glUseProgramObjectARB
Fragment Shader
glCreateShaderObjectARB
glShaderSourceARB
glCompileShaderARB

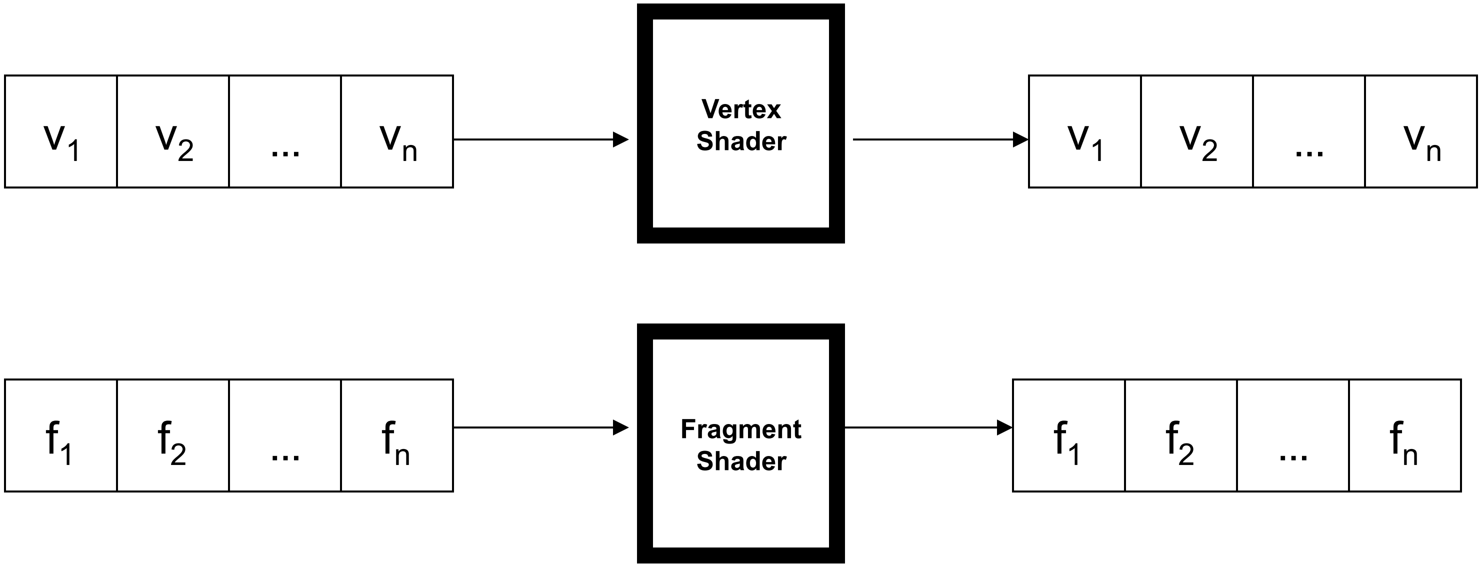
The Programmable GPU



- **GPU** = **vertex shader** (vertex program) + **fragment shader** (fragment program, pixel program)
- Vertex shader replaces per-vertex transform & lighting
- Fragment shader replaces texture stages
- Fragment testing after the fragment shader
- Flexibility to do framebuffer pixel blending

GPU programming model

- “Stream programming”
  - Process each vertex or fragment independently



The idea

- 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, which might have been modified.
- The fragment shader can now work on the interpolated values, including the user defined variables.

Vertex Program

- Replace the fixed-function operations performed by the vertex processor
- A vertex program is executed on each vertex triggered by glVertex*()
- Each vertex program 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

A very simple vertex program

```
void main(void)
{
    gl_Position = gl_ProjectionMatrix*gl_ModelViewMatrix*gl_Vertex;
}
```

- Just a passing-through shader: convert a vertex from local space to clip space
- No color is assigned here, so the fragment program will need to decide the fragment colors
- All variables starts with ‘gl_’ are part of OpenGL state so no need to declare

GLSL Data Types

- Supported data types are very similar to C/C++: float, int, bool, etc
- Additional types examples: vec2, vec3, vec4, mat2, mat3, mat4
- Can use C++ style constructor
  - vec3 a = vec3(1.0, 2.1, -1.2);
  - vec3 b = vec2(a); //conversion

GLSL Qualifiers

- Three types of variables: *Attributes, Uniform, Varying*
  - Attribute: used by vertex shaders for variables that can change once per vertex
    - Build-in attributes: gl_Vertex, gl_FrontColor
    - User-definted attributes (example): temperature, velocity
  - Uniform: variables set for the entire primitive, i.e., assigned outside glBegin()/glEnd();
    - Also include build-in and user-definted

Attributes

- Built-in

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

- User-defined (examples)

```
attribute vec3  myTangent;
attribute vec3  myBinormal;
Etc…
```

Built-in Uniforms

```
uniform   mat4  gl_ModelViewMatrix;
uniform   mat4  gl_ProjectionMatrix;
uniform   mat4  gl_ModelViewProjectionMatrix;
uniform   mat3  gl_NormalMatrix;
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];
```

GLSL Qualifiers (cont'd)

- Varying variables: the mechanism for conveying data from a vertex program to a fragment program
- Defined on a per vertex basis but interpolated over the primitive for the fragment program.
- Include build-in and user defined varying variables

Built-in Varyings

```
varying   vec4  gl_FrontColor      // vertex
varying   vec4  gl_BackColor;      // vertex
varying   vec4  gl_FrontSecColor;  // vertex
varying   vec4  gl_BackSecColor;   // vertex

varying   vec4  gl_Color;          // fragment
varying   vec4  gl_SecondaryColor; // fragment

varying   vec4  gl_TexCoord[];     // both
varying   float gl_FogFragCoord;   // both
```

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, notEqual, any, all**

Built-in functions

- Texture
  - **texture1D, texture2D, texture3D, textureCube**
  - **texture1DProj, texture2DProj, texture3DProj, textureCubeProj**
  - **shadow1D, shadow2D, shadow1DProj, shadow2Dproj**
- Vertex
  - **ftransform**

Vertex Processor 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 Processor Output

- Vertex shader uses input values to compute output values
- Vertex shader **must** compute gl_Position
  - Mandatory, needed by the rasterizer
  - Can use built-in function ftransform() to get invariance with fixed functionality

Vertex Processor Output

- Other output values are called “varying” variables
  - E.g., color, texture coordinates, arbitrary data
  - Will be interpolated in a perspective-correct fashion 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

Vertex Program Capabilities

- Vertex program can do general processing, including things like:
  - Vertex transformation
  - Normal transformation, normalization and rescaling
  - Lighting
  - Color material application
  - Clamping of colors
  - Texture coordinate generation
  - Texture coordinate transformation

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.

TakeOver

- When the vertex processor is active, the following fixed functionality is **disabled**:
  - The modelview matrix is not applied to vertex coordinates
  - The projection matrix is not applied to vertex coordinates
  - The texture matrices are not applied to texture coordinates
  - Normals are not transformed to eye coordinates
  - Normals are not rescaled or normalized
  - Texture coordinates are not generated automatically
  - Per vertex lighting is not performed
  - Color material computations are not performed
  - Etc.

Intervening Fixed Functionality

- Results from vertex processing undergo:
  - Color clamping or masking (for built-in varying variables that deal with color, but not user-defined varying variables)
  - Perspective division on clip coordinates
  - Viewport mapping
  - Depth range
  - Clipping, including user clipping
  - Front face determination
  - Clipping of color, texture coordinate, fog, point-size and user-defined varying
  - Etc.

Fragment Program

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

A very simple fragment program

```
void main(void)
{
    gl_FragColor = gl_FrontColor;
}
```

- Just a passing-through fragment shader

Fragment Program 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 Processor 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 Processor 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

Fragment Program Capabilities

Fragment shader can do general processing, like:

- Operations on interpolated values
- Texture access
- Texture application
- Fog
- Color sum
- Color matrix
- Discard fragment
- etc

Fragment Program Capabilities

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

TakeOver

- When the fragment processor is active, the following fixed functionality is disabled:
  - The texture environments and texture functions are not applied
  - Texture application is not applied
  - Color sum is not applied
  - Fog is not applied

Example: Vertex Shader

```
varying vec4 diffuseColor;
varying vec3 fragNormal;
varying vec3 lightVector;

uniform vec3 eyeSpaceLightVector;

void main(){

   vec3 eyeSpaceVertex= vec3(gl_ModelViewMatrix *
   gl_Vertex);
   lightVector= vec3(normalize(eyeSpaceLightVector -
   eyeSpaceVertex));
   fragNormal = normalize(gl_NormalMatrix * gl_Normal);

   diffuseColor = gl_Color;
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

}
```

Example: Fragment Shader

```
varying vec4 diffuseColor;
varying vec3 lightVector;
varying vec3 fragNormal;

void main(){

   float perFragmentLighting=max(dot
   (lightVector,fragNormal),0.0);

   gl_FragColor = diffuseColor * lightingFactor;

}
```

Toon Shading Example

- Toon Shading
  - Characterized by abrupt change of colors
  - Vertex Shader computes the vertex intensity (declared as varying)
  - Fragment Shader computes colors for the fragment based on the interpolated intensity

Vertex Shader

```
uniform vec3 lightDir;
varying float intensity;
void main() {
   vec3 ld;
   intensity = dot(lightDir,gl_Normal);
   gl_Position = ftransform();
}
```

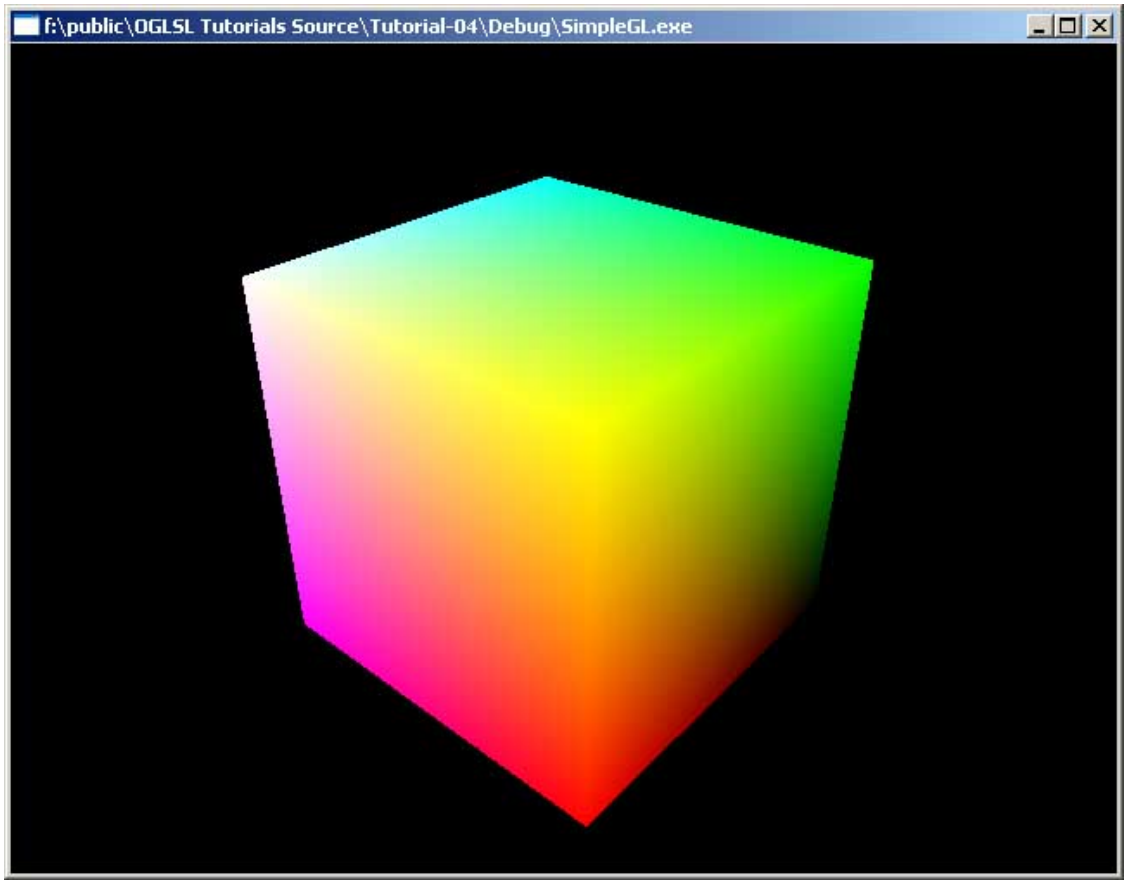
Fragment Shader

```
varying float intensity;

void main() {
    vec4 color;
    if (intensity > 0.95) color = vec4(1.0,0.5,0.5,1.0);
    else if (intensity > 0.5) color = vec4(0.6,0.3,0.3,1.0);
    else if (intensity > 0.25) color = vec4(0.4,0.2,0.2,1.0);
    else color = vec4(0.2,0.1,0.1,1.0);
    gl_FragColor = color;
 }
```

Varying Variable Example

Determine color based on x y z coordinates



Vertex Shader

```
varying float xpos;
varying float ypos;
varying float zpos;

void main(void) {
    xpos = clamp(gl_Vertex.x,0.0,1.0);
    ypos = clamp(gl_Vertex.y,0.0,1.0);
    zpos = clamp(gl_Vertex.z,0.0,1.0);
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

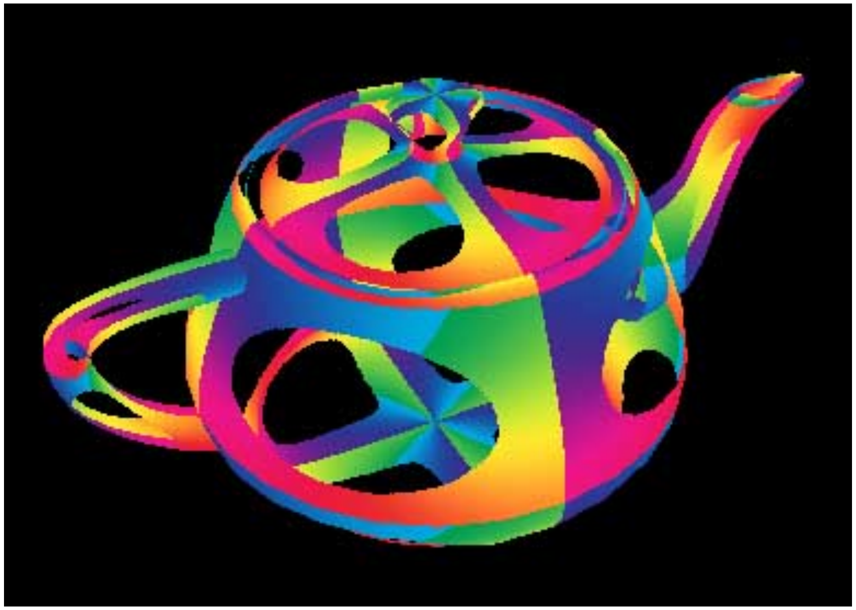
Fragment Shader

```
varying float xpos;
varying float ypos;
varying float zpos;

void main (void) {
	gl_FragColor = vec4 (xpos, ypos, zpos, 1.0);
}
```

Color Key Example

- Set a certain color (say FF00FF as transparent



Vertex Shader

```
void main(void) {

    gl_TexCoord[0] = gl_MultiTexCoord0;

    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

}
```

Fragment Shader

```
uniform sampler2D myTexture;

#define epsilon 0.0001

void main (void) {
        vec4 value = texture2D(myTexture, vec2
   (gl_TexCoord[0]));
        if (value[0] > 1.0-epsilon) && (value[2] > 1.0-epsilon))
                discard;
        gl_FragColor = value;
}
```

Color Map Example

- Suppose you want to render an object such that its surface is colored by the temperature.
    - You have the temperatures at the vertices.
    - You want the color to be interpolated between the coolest and the hottest colors.
- Previously, you would calculate the colors of the vertices in your program, and say glColor ().
- Now, lets do it in the vertex and pixel shaders…

Vertex shader

```
// uniform qualified variables are changed at most once
// per primitive
uniform float CoolestTemp;
uniform float TempRange;

// attribute qualified variables are typically changed per vertex
attribute float VertexTemp;

// varying qualified variables communicate from the vertex
// shader to the fragment shader
varying float Temperature;
```

Vertex shader

```
void main()
{
    // compute a temperature to be interpolated per fragment,
    // in the range [0.0, 1.0]
    Temperature = (VertexTemp - CoolestTemp) / TempRange;
    /*
    The vertex position written in the application using glVertex() can
    be read from the built-in variable gl_Vertex. Use this value and
    the current model view transformation matrix to tell the rasterizer
    where this vertex is. Could use ftransform(). */
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Fragment Shader

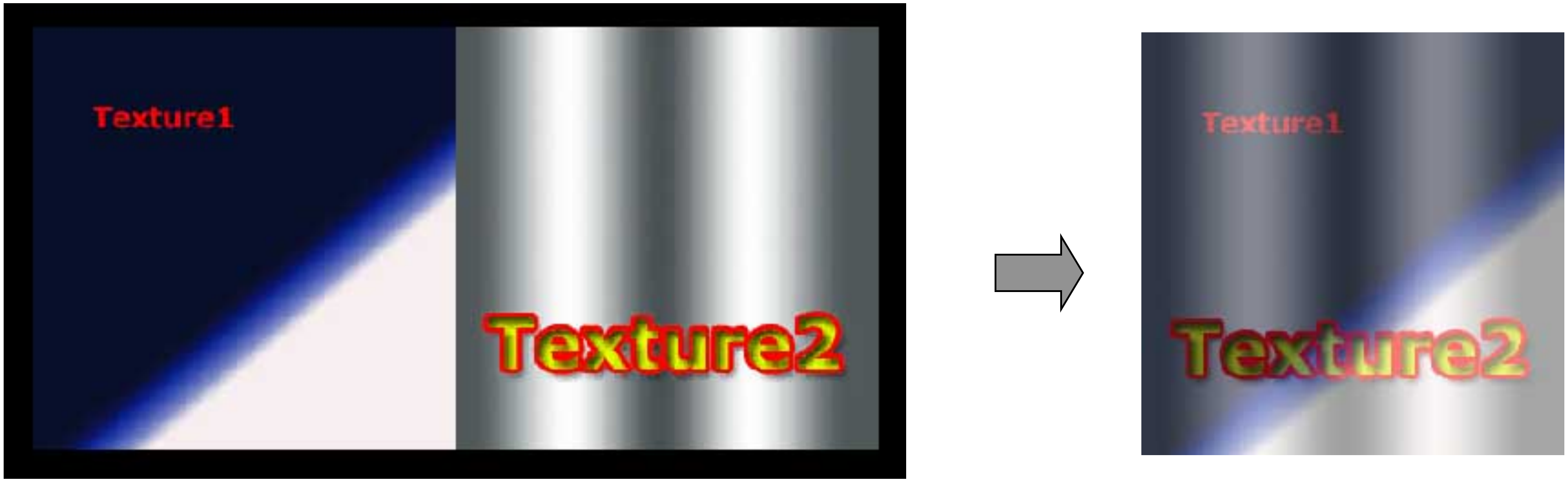
```
// uniform qualified variables are changed at most
// once per primitive by the application, and vec3
// declares a vector of three floating-point numbers
uniform vec3 CoolestColor;
uniform vec3 HottestColor;

// Temperature contains the now interpolated
// per-fragment value of temperature set by the
// vertex shader
varying float Temperature;
```

Fragment Shader

```
void main()
{
    // get a color between coolest and hottest colors, using
    // the mix() built-in function
    vec3 color = mix(CoolestColor, HottestColor, Temperature);
    // make a vector of 4 floating-point numbers by appending an
    // alpha of 1.0, and set this fragment’s color
    gl_FragColor = vec4(color, 1.0);
}
```

Multi-texturing Example



OpenGL Setup

```
glActiveTextureARB(GL_TEXTURE0_ARB); glBindTexture
(GL_TEXTURE_2D, texture1); glEnable(GL_TEXTURE_2D);
glTexEnvf (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                GL_COMBINE_EXT);
glTexEnvf (GL_TEXTURE_ENV, GL_COMBINE_RGB_EXT,
                GL_REPLACE);

glActiveTextureARB(GL_TEXTURE1_ARB); glBindTexture
(GL_TEXTURE_2D, texture2); glEnable(GL_TEXTURE_2D);
glTexEnvf (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                GL_COMBINE_EXT);
glTexEnvf (GL_TEXTURE_ENV, GL_COMBINE_RGB_EXT,
                GL_INCR);
```

OpenGL Setup (II)

```
void drawBox(float size) {
glBegin(GL_QUADS);
    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0, 1.0);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0.0, 1.0);
    glVertex3f(0.0, 0.0, 0.0);

    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0, 0.0);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0.0, 0.0);
    glVertex3f(0.0, size*1.0, 0.0);

    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0, 0.0);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1.0, 0.0);
    glVertex3f(size*1.0, size*1.0, 0.0);

    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0, 1.0);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1.0, 1.0);
    glVertex3f(size*1.0, 0.0, 0.0);
glEnd();
}
```

Vertex Shader

```
void main(void) {
    gl_TexCoord[0] = gl_MultiTexCoord0;
    gl_TexCoord[1] = gl_MultiTexCoord1;

    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Fragment Shader

```
uniform sampler2D myTexture1;
uniform sampler2D myTexture2;

void main (void) {
    vec4 texval1 = texture2D(myTexture, vec2
    (gl_TexCoord[0]));
    vec4 texval2 = texture2D(myTexture2, vec2
    (gl_TexCoord[1]));
    gl_FragColor = 0.5*(texval1 + texval2);
}
```

Syntax

- Based on syntax of ANSI C
- Some additions to support graphics functionality
- Some additions from C++
- Some differences for a cleaner language design

Special additions

- Vector types are supported for floats, integers, and booleans
  - Can be 2-, 3-, or 4- components
- Floating point matrix types are supported
  - 2x2, 3x3, or 4x4
- Type qualifiers “attribute”, “uniform”, and “varying”
- Built-in names for accessing OpenGL state and for communicating with OpenGL fixed functionality

Special additions

- A variety of built-in functions are included for common graphics operations
  - Square root, trig functions, geometric functions, texture lookups, etc.
- Keyword “discard” to cease processing of a fragment
- Vector components are named (.rgba, .xyzw, .stpq) and can be swizzled
  - The component naming is only for readability
- “Sampler” data type is added for texture access

Types

- Basic
  - float, vec2, vec3, vec4
  - int, ivec2, ivec3, ivec4
  - bool, bvec2, bvec3, bvec4
  - No string, no char/byte
  - mat2, mat3, mat4 (all floats)
  - void
  - sampler1D, sampler2D, sampler3D
- Others
  - Array (Only 1D)
  - Structures

Type Qualifiers

- const
  - variable is a constant and can only be written during its declaration
- attribute
  - per-vertex data values provided to the vertex shader
- uniform
  - – (relatively) constant data provided by the application or by OpenGL for use in the shader
- varying
  - a perspective-correct interpolated value
  - output for vertex shader
  - input for fragment shader

Type Qualifiers

- in
  - for function parameters copied into a function, but not copied out
- out
  - for function parameters copied out of a function, but not copied in
- inout
  - for function parameters copied into and out of a function

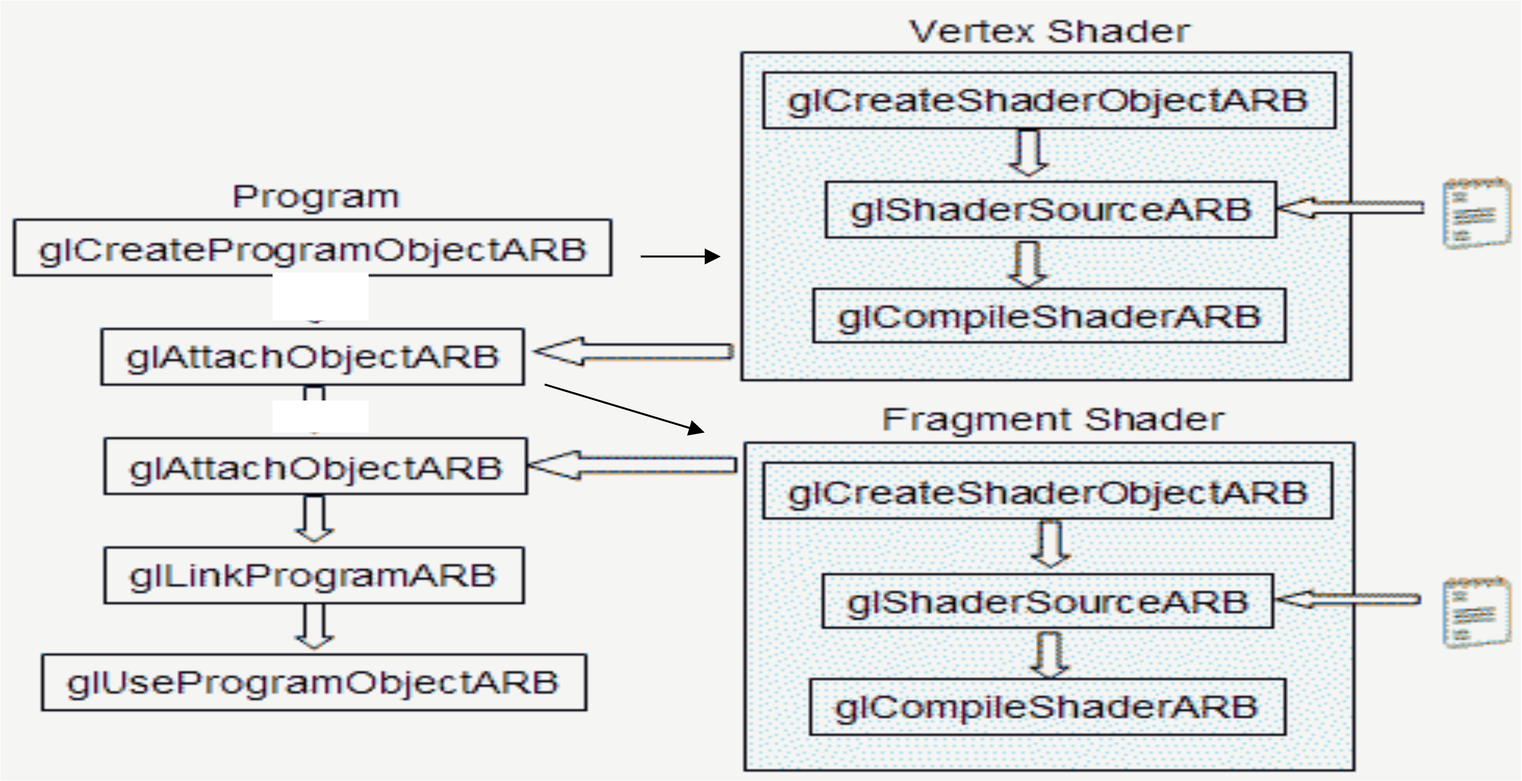
Built-in Functions

- Trigonometry/angle
  - radians, degrees, sin, cos, tan, asin, acos, atan
- Exponential
  - pow, exp2, log2, sqrt, inversesqrt
- Common
  - abs, sign, floor, ceil, fract, mod, min, max, clamp, mix, step, smoothstep
- Geometric and matrix
  - length, distance, dot, cross, normalize, ftransform, faceforward, reflect, matrixCompMult

Shader Objects

- Shaders are defined as an array of strings
- 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 current state
- Goal was to mimic C/C++ source code development model

Sequence


Vertex Shader
glCreateShaderObjectARB
glShaderSourceARB
glCompileShaderARB
Program
glCreateProgramObjectARB
glAttachObjectARB
glAttachObjectARB
glLinkProgramARB
glUseProgramObjectARB
Fragment Shader
glCreateShaderObjectARB
glShaderSourceARB
glCompileShaderARB

Creating objects

```
GLhandleARB glCreateProgramObjectARB();

GLhandleARB glCreateShaderObjectARB
   (GL_VERTEX_SHADER_ARB);

GLhandleARB glCreateShaderObjectARB
   (GL_FRAGMENT_SHADER_ARB);
```

Compiling

```
void glShaderSourceARB(GLhandleARB shader, GLsizei
  nstrings, const GLcharARB **strings, const GLint
  *lengths)
```

//if lengths==NULL, assumed to be null-terminated

```
void glCompileShaderARB(GLhandleARB shader);
```

Attaching & Linking

```
void glAttachObjectARB(GLhandleARB program,
  GLhandleARB shader);
   //twice, once for vertex shader & once for fragment shader

void glLinkProgramARB(GLhandleARB program);
  //program now ready to use

void glUseProgramObjectARB(GLhandleARB program);
  //switches on shader, bypasses FFP
  //if program==0, shaders turned off, returns to FFP
```

In short…

```
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);
```