Introduction to the OpenGL Shading Language (GLSL)

Mike Bailey
mjb@cs.oregonstate.edu

The Basic Computer Graphics Pipeline, OpenGL-style

The Basic Computer Graphics Pipeline, Shader-style

GLSL Variable Types

attribute These are per-vertex in variables. They are assigned per-vertex and passed into the vertex shader, usually with the intent to interpolate them through the rasterizer.

uniform These are “global” values, assigned and left alone for a group of primitives. They are read-only accessible from all of your shaders. They cannot be written to from a shader.

out / in These are passed from one shader stage to the next shader stage. In our case, out variables come from the vertex shader, are interpolated in the rasterizer, and go in to the fragment shader. Attribute variables are in variables to the vertex shader.

GLSL Shaders Are Like C With Extensions for Graphics:

- Types include int, vec2, vec3, vec4
- Types include float, vec2, vec3, vec4
- Types include mat2, mat3, mat4
- Types include bool, vec2, vec3, vec4
- Types include sampler to access textures
- Vector components are accessed with [index], .rgba, .xyzw, or.stpq
- You can ask for parallel SIMD operations (doesn’t necessarily do it in hardware):
  - vec4 a, b, c;
  - a = b + c;
- Vector components can be “swizzled” (c1.rgb = c2.abgr)
- Type qualifiers: const, attribute, uniform, in, out
- Variables can have “layout qualifiers” (more on this later)
- The discard operator is used in fragment shaders to get rid of the current fragment
The discard Operator

```cpp
if (random number < 0.5) discard;
```

GLSL Shaders Are Missing Some C-isms:

- No type casts -- use constructors instead: `int i = int(x);`
- Only some amount of automatic promotion (don’t rely on it!)
- No pointers
- No strings
- No enums
- Can only use 1-D arrays (no bounds checking)

Warning: integer division is still integer division!

```cpp
float f = float(2 / 4); // still gives 0. like C, C++, Python, Java
```

The Shaders' View of the Basic Computer Graphics Pipeline

- A missing stage is OK. The output from one stage becomes the input of the next stage that is there.
- The last stage before the fragment shader feeds its output variables into the rasterizer. The interpolated values then go to the fragment shader.

We are just going to cover these two

A GLSL Vertex Shader Replaces These Operations:

- Vertex transformations
- Normal transformations
- Normal utilization (normalization)
- Computing per-vertex lighting
- Taking per-vertex texture coordinates (s,t) and passing them through the rasterizer to the fragment shader

A GLSL Fragment Shader Replaces These Operations:

- Color computation
- Texturing
- Handling of per-fragment lighting
- Color blending
- Discarding fragments

Built-in Vertex Shader Variables You Will Use a Lot:

- `vec4 gl_Vertex`
- `vec3 gl_Normal`
- `vec4 gl_Color`
- `vec4 gl_MultiTexCoord0`
- `mat4 gl_ModelViewMatrix`
- `mat4 gl_ProjectionMatrix`
- `mat4 gl_ModelViewProjectionMatrix`
- `mat4 gl_NormalMatrix` (this is the transpose of the inverse of the MV matrix)

Built-in Fragment Shader Variables You Will Use a Lot:

- `vec4 gl_FragColor`

Note: while this all still works, OpenGL now prefers that you pass in all the above variables (except `gl_Position`) as user-defined attribute variables. We’ll talk about this later.
My Own Variable Naming Convention

With 7 different places that GLSL variables can be written from, I decided to adopt a naming convention to help me recognize what program-defined variables came from what sources:

<table>
<thead>
<tr>
<th>Beginning letter(s)</th>
<th>Means that the variable …</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Is a per-vertex attribute from the application</td>
</tr>
<tr>
<td>u</td>
<td>Is a uniform variable from the application</td>
</tr>
<tr>
<td>v</td>
<td>Came from the vertex shader</td>
</tr>
<tr>
<td>tc</td>
<td>Came from the tessellation control shader</td>
</tr>
<tr>
<td>te</td>
<td>Came from the tessellation evaluation shader</td>
</tr>
<tr>
<td>g</td>
<td>Came from the geometry shader</td>
</tr>
<tr>
<td>f</td>
<td>Came from the fragment shader</td>
</tr>
</tbody>
</table>

#version 330 compatibility

void main( )
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

 Vertex shader:

The Minimal Vertex and Fragment Shader

void main( )
{
    gl_FragColor = vec4( .5, 1., 0., 1. );
}

Fragment shader:

A Reminder of what a Rasterizer does

There is a piece of hardware called the Rasterizer. Its job is to interpolate a line or polygon, defined by vertices, into a collection of fragments. Think of it as filling in squares on graph paper.

A fragment is a “pixel-to-be”. In computer graphics, “pixel” is defined as having its full RGBA already computed. A fragment does not yet but all of the information needed to compute the RGBA is there.

A fragment is turned into a pixel by the fragment processing operation.

Rasterizers interpolate built-in variables, such as the (x,y) position where the pixel will live and the pixel’s z-coordinate. They can also interpolate user-defined variables as well.

A Little More Interesting

vColor = gl_Vertex.xyz;

Setting rgb From xyz, I

vColor = gl_Vertex.xyz;

What's Changed About This?
What's Different About This?

Set the color from the **pre-transformed (MC) xyz**:  
```cpp
#version 330 compatibility
out vec3 vColor;
void main()
{
  vec4 pos = gl_ModelViewMatrix * gl_Vertex;
  if set rgb from xyz!
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Set the color from the **post-transformed (WC/EC) xyz**:  
```cpp
#version 330 compatibility
out vec3 vColor;
void main()
{
  vec4 pos = gl_ModelViewMatrix * gl_Vertex;
  if set rgb from xyz! why? who cares?
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Set the color from the **pre-transformed (MC) xyz**:  
```cpp
vColor = ( gl_ModelViewMatrix * gl_Vertex ).xyz;
```

**Setting rgb From xyz, II**

```cpp
vColor = gl_Vertex.xyz;
```

**Vertex shader:**

```cpp
#version 330 compatibility
uniform float uKa, uKd, uKs; // coefficients of each type of lighting
uniform vec3 uColor; // object color
uniform vec3 uSpecularColor; // light color
uniform float uShininess; // specular exponent
in vec2 vST; // texture coords
in vec3 vN; // normal vector
in vec3 vL; // vector from point to light
in vec3 vE; // vector from point to eye
void main()
{
  vec3 Normal = normalize(vN);
  vec3 Light = normalize(vL);
  vec3 Eye = normalize(vE);
  vec3 ambient = uKa * uColor;
  float d = max(dot(Normal,Light), 0.); // only do diffuse if the light can see the point
  vec3 diffuse = uKd * d * uColor;
  float s = 0.; // only do specular if the light can see the point
  if ( dot(Normal,Light) > 0. )
  {
    vec3 ref = normalize( reflect(-Light, Normal) );
    s = pow(max(dot(Eye,ref),0.), uShininess);
  }
  vec3 specular = uKs * s * uSpecularColor;
  gl_FragColor = vec4( ambient + diffuse + specular, 1.);
}
```

**Fragment shader:**

```cpp
#version 330 compatibility
uniform float uKa, uKd, uKs; // coefficients of each type of lighting
uniform vec3 uColor; // object color
uniform vec3 uSpecularColor; // light color
uniform float uShininess; // specular exponent
in vec2 vST; // texture coords
in vec3 vN; // normal vector
in vec3 vL; // vector from point to light
in vec3 vE; // vector from point to eye
void main()
{
  vec3 Normal = normalize(vN);
  vec3 Light = normalize(vL);
  vec3 Eye = normalize(vE);
  vec3 ambient = uKa * uColor;
  float d = max(dot(Normal,Light), 0.); // only do diffuse if the light can see the point
  vec3 diffuse = uKd * d * uColor;
  float s = 0.; // only do specular if the light can see the point
  if ( dot(Normal,Light) > 0. )
  {
    vec3 ref = normalize( reflect(-Light, Normal) );
    s = pow(max(dot(Eye,ref),0.), uShininess);
  }
  vec3 specular = uKs * s * uSpecularColor;
  gl_FragColor = vec4( ambient + diffuse + specular, 1.);
}
```

**Per-fragment Lighting**

**Ambient**

**Diffuse**

**Specular**

**All together now!**
Within the fragment shader:

```c++
vec3 myColor = uColor;
if( uS0-uSize/2. <= vST.s && vST.s <= uS0+uSize/2. && uT0-uSize/2. <= vST.t && vST.t <= uT0+uSize/2. ) {
    myColor = vec3(1., 0., 0.);
}
vec3 ambient = uKa * myColor;
```

Here's the cool part: It doesn't matter (up to the limits of 32-bit floating-point precision) how far you zoom in. You still get an exact crisp edge. This is an advantage of procedural (equation-based) textures, as opposed to texel-based textures.

**A C++ Class to Handle the Shaders**

Setup:

```c++
GLSLProgram *Pattern;

Pattern = new GLSLProgram();
bool valid = Pattern->Create( "pattern.vert", "pattern.frag" );
if( ! valid ) {
    ...  
}
```

This loads, compiles, and links the shader. If something went wrong, it prints error messages and returns a value of false.

**Setting Up Texturing in Your C/C++ Program**

You do all the texture things you did before, but add this:

```c++
Pattern->Use();
glActiveTexture( GL_TEXTURE0 );
Pattern->SetUniformVariable( "uTexUnit", 0 );
```

This is the Texture Unit Number. It can be 0-15 (and often a lot higher depending on the graphics card).

```c++
void main( ) {
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```
HINTS ON RUNNING SHADERS ON YOUR OWN SYSTEM

- You need a graphics system that is OpenGL 2.0 or later. Basically, if you got your graphics system in the last 5 years, you should be OK. If you don't have access to such a system, use the CGEL. (The most recent OpenGL level there is 4.6)
- Update your graphics drivers to the most recent level!
- If you are on Windows, you must do the GLEW setup. It looks like this in the sample code:

```c
GLenum err = glewInit();
if (err != GLEW_OK) {
    fprintf(stderr, "glewInit Error
    
```
- And, this must come after you've opened a window. (It is this way in the code, but I'm saying this because I know some of you went in and "simplified" the sample code by deleting everything you didn't think you needed.)

- You can use the GLSL C++ class you've been given only after GLEW has been setup. So, initialize your shader program:

```c
bool valid = Pattern->Create( "pattern.vert", "pattern.frag" );
```
- Use the Shader Program in Display:

```c
Pattern->Use();
Pattern->SetUniformVariable( ... 
```
- Draw the object here

```c
Pattern->Use( 0 );                    // return to fixed functionality
```

GUIDE TO WHERE TO PUT PIECES OF YOUR SHADER CODE, I

Declare the GLSLProgram above the main program (as a global):

```c
GLSLProgram *Pattern;
```

At the end of InitGraphics(), create the shader program and setup your shaders:

```c
Pattern = new GLSLProgram();
bool valid = Pattern->Create("proj05.vert", "proj05.frag");
if ( ! valid ) { ... }
```

Use the Shader Program in Display:

```c
Pattern->Use();
Pattern->SetUniformVariable( ... 
```
- Draw the object here

```c
Pattern->Use( 0 );                    // return to fixed functionality
```

GUIDE TO WHERE TO PUT PIECES OF YOUR SHADER CODE, II

TIPS ON DRAWING THE OBJECT:

- If you want to key off of s and t coordinates in your shaders, the object had better have s and t coordinates assigned to its vertices – not all do!
- If you want to use surface normals in your shaders, the object had better have surface normals assigned to its vertices – not all do!
- Be sure you explicitly assign all of your uniform variables – no error messages occur if you forget to do this – it just quietly screws up.
- The glutSolidTeapot has been textured in patches, like a quilt – cute, but weird
- The MjbSphere() function from the texturing project will give you a very good sphere