Introduction to using the OpenGL Shading Language (GLSL)

Mike Bailey
mjb@cs.oregonstate.edu

The Basic Computer Graphics Pipeline, OpenGL-style

Model Transform
View Transform
Projection Transform

Fragment Processing,
Texturing,
Per-fragment Lighting

Rasterization

MC = Model Vertex Coordinates
WC = World Vertex Coordinates
EC = Eye Vertex Coordinates

Per-vertex in
variables

Uniform Variables

Per-vertex out
variables

Per-fragment in
variables

Uniform Variables

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, bvec2, bvec3, bvec4
- Types include sampler to access textures
- 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" (c.rgba = a.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 Halts Production of the Current Fragment

if (random_number < 0.5)
    discard;

GLSL Shaders Are Missing Some C-isms:

- No type casts – use constructors instead: int i = int(x);
- There is 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!

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 unitization (normalization)
- Computing per-vertex lighting
- Taking per-vertex texture coordinates (s,t) and interpolating them through the rasterizer to the fragment shader

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
- mat3 gl_NormalMatrix (this is the transpose of the inverse of the MV matrix)
- vec4 gl_Position

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. For now, we are going to use the most straightforward way possible.

A GLSL Fragment Shader Replaces These Operations:

- Color computation
- Texturing
- Color blending
- Discarding fragments

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 out the fragment color variable as a user-defined output variable. We’ll talk about this later. For now, we are going to use the most straightforward way possible.
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><code>a</code></td>
<td>is a per-vertex attribute from the application</td>
</tr>
<tr>
<td><code>u</code></td>
<td>is a uniform variable from the application</td>
</tr>
<tr>
<td><code>v</code></td>
<td>came from the vertex shader</td>
</tr>
<tr>
<td><code>tC</code></td>
<td>came from the tessellation control shader</td>
</tr>
<tr>
<td><code>te</code></td>
<td>came from the tessellation evaluation shader</td>
</tr>
<tr>
<td><code>g</code></td>
<td>came from the geometry shader</td>
</tr>
<tr>
<td><code>f</code></td>
<td>came from the fragment shader</td>
</tr>
</tbody>
</table>

This isn’t part of “official” OpenGL/GLSL – it is my way of handling the confusion.

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 have all of the information needed to compute the RGBA. It is an intermediate result of the rasterization process.

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.

Setting rgb from the Untransformed xyz, I

```
out vec3 vColor;
void main()
{
    vec4 pos = gl_Vertex;
    vColor = pos.xyz; // set rgb from xyz!
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

```
in vec3 vColor;
void main()
{
    gl_FragColor = vec4(vColor, 1.);
}
```

Something Has Changed: Setting rgb from the Transformed xyz, II

```
out vec3 vColor;
void main()
{
    vec4 pos = gl_ModelViewMatrix * gl_Vertex;
    vColor = pos.xyz; // set rgb from xyz!
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

```
in vec3 vColor;
void main()
{
    gl_FragColor = vec4(vColor, 1.);
}
```

The Minimal Vertex and Fragment Shader

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

```
void main()
{
    gl_FragColor = vec4(0.5, 1.0, 0.0, 1.0);
}
```

A Little More Interesting: Setting rgb from the Untransformed xyz, I

```
out vec3 vColor;
void main()
{
    vec4 pos = gl_Vertex;
    vColor = pos.xyz; // set rgb from xyz!
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

```
in vec3 vColor;
void main()
{
    gl_FragColor = vec4(vColor, 1.);
}
```

A Little More Interesting: Setting rgb from the Transformed xyz, II

```
out vec3 vColor;
void main()
{
    vec4 pos = gl_ModelViewMatrix * gl_Vertex;
    vColor = pos.xyz; // set rgb from xyz!
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

```
in vec3 vColor;
void main()
{
    gl_FragColor = vec4(vColor, 1.);
}
```
**What's Different About This?**

- Set the color from **untransformed** (MC) xyz
  - `vColor = gl_ModelViewMatrix * gl_Vertex`. xyz!
- Set the color from **transformed** (WC/EC) xyz
  - `vColor = ( gl_ModelViewMatrix * gl_Vertex ).xyz;`

**Setting rgb from the transformed xyz, II**

- `vColor = ( gl_ModelViewMatrix * gl_Vertex ).xyz;`

**Vertex shader:**

- Per-fragment Lighting
  - `vColor = gl_Vertex.xyz;`

**Fragment shader:**

- Per-fragment Lighting
  - `vColor = gl_Vertex.xyz;`
Within the fragment shader, find out if the current fragment is within a particular rectangle:

```glsl```
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;
```

Drawing a Pattern on an Object

Within the fragment shader, find out if the current fragment is within a particular rectangle:

```glsl```
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;
```
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. (The most recent OpenGL level is 4.6)

• Update your graphics drivers to the most recent!

• You must do the GLEW setup. It looks like this in the sample code:

```
GLenum err = glewInit();
if( err != GLEW_OK )
{
    fprintf(stderr, "glewInit Error\n");
}
else
    fprintf(stderr, "GLEW initialized OK\n");
```

This must come after you’ve opened a window. (It is this way in the sample code, but I’m saying this because I know some of you go in and “simplify” the sample code by deleting everything you don’t think you need.)

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

```
bool valid = Pattern->Create( "pattern.vert", "pattern.frag" );
```

Hints on Running Shaders on Your Own System

Guide to Where to Put Pieces of Your Shader Code, I

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

```
GLSLProgram * Pattern;
```

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

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

3. Turn on the shader program in Display(), set shader variables, draw the object, then turn off the shader program:

```
Pattern->Use( );
Pattern->SetUniformVariable( ...
Sphere( );
Pattern->UnUse( ); // return to the fixed function pipeline
```

4. When you run your program, be sure to check the console window for shader compilation errors!

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 (vt) assigned to its vertices – not all OBJ files do!

• If you want to use surface normals in your shaders, the object had better have surface normals (vn) assigned to its vertices – not all OBJ files 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 OsuSphere( ) function from the texturing project will give you a very good sphere