Introduction to the OpenGL Shading Language (GLSL)

Mike Bailey
mjb@cs.oregonstate.edu

The Basic Computer Graphics Pipeline, OpenGL-style

Vertex, Normal, Color

Vertex, Normal, Color

ModelViewMatrix, ProjectionMatrix, ModelViewProjectionMatrix

Model Transformer

View Transform

Per-vertex Lighting

Projection Transform

Fragment Processing, Texturing, Per-fragment Lighting

Rasterization

Framebuffer

MC = Model Vertex Coordinates
WC = World Vertex Coordinates
EC = Eye Vertex Coordinates
The Basic Computer Graphics Pipeline, Shader-style

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

Vertex Shader

Fragment Shader

Framebuffer

gl_Vertex, gl_Normal, gl_Color  
Per-vertex in variables  
Uniform Variables

gl_ModelViewMatrix, gl_ProjectionMatrix, gl_ModelViewProjectionMatrix  
Per-vertex out variables

gl_Position, Per-vertex out variables

Model Transform  
Per-vertex Lighting

View Transform  
Projection Transform

Fragment Processing, Texturing, Per-fragment Lighting

Framebuffer  
gl_FragColor

gl_ModelViewMatrix, gl_ProjectionMatrix, gl_ModelViewProjectionMatrix  
Per-fragment in variables  
Uniform Variables

Per-fragment out variables

Rasterization

gl_ModelViewMatrix, gl_ProjectionMatrix, gl_ModelViewProjectionMatrix  
Per-fragment in variables  
Uniform Variables
**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, ivec2, ivec3, ivec4
- Types include float, vec2, vec3, vec4
- Types include mat2, mat3, mat4
- Types include bool, bvec2, bvec3, bvec4
- 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.rgba = 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 Halts Production of the Current Fragment

```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

![Diagram of the Basic Computer Graphics Pipeline]

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`
- `mat4 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 easiest way possible.

---

A GLSL Fragment Shader Replaces These Operations:

- Color computation
- Texturing
- Handling of per-fragment lighting
- 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 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 easiest 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>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>

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

The Minimal Vertex and Fragment Shader

Vertex shader:

```glsl
#version 330 compatibility

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

This makes sure that each vertex gets transformed.

Fragment shader:

```glsl
#version 330 compatibility

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

This assigns a fixed color (r=0.5, g=1., b=0.) and alpha (=1.) to each fragment drawn.

Not terribly useful …
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:
Setting rgb From xyz, I

**Vertex shader:**

```glsl
#version 330 compatibility
out vec3 vColor;  // per-vertex
void main( )
{
    vec4 pos = gl_Vertex;
    vColor = pos.xyz; // set rgb from xyz!
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

**Fragment shader:**

```glsl
#version 330 compatibility
in vec3 vColor;  // per-fragment
void main( )
{
    gl_FragColor = vec4( vColor, 1. );
}
```
Setting rgb From xyz, I

vColor = gl_Vertex.xyz;

Setting rgb From xyz, II

Vertex shader:

```cpp
#version 330 compatibility
out vec3 vColor;

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

Fragment shader:

```cpp
#version 330 compatibility
in vec3 vColor;

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

Set the color from the **pre-transformed (MC)** \( xyz \):

```c
#version 330 compatibility
out vec3 vColor;

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

Set the color from the **post-transformed (WC/EC)** \( xyz \):

```c
#version 330 compatibility
out vec3 vColor;

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

Setting rgb From \( xyz \), II

\[ vColor = (\text{gl}_\text{ModelViewMatrix} \ast \text{gl}_\text{Vertex}).xyz; \]
**Setting rgb From xyz**

```cpp
vColor = gl_Vertex.xyz;

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

**Per-fragment Lighting**

```cpp
#version 330 compatibility

out vec2 vST; // texture coords
out vec3 vN; // normal vector
out vec3 vL; // vector from point to light
out vec3 vE; // vector from point to eye

const vec3 LIGHTPOSITION = vec3( 5., 5., 0.);

void main( )
{
    vST = gl_MultiTexCoord0.st;
    vec4 ECposition = gl_ModelViewMatrix * gl_Vertex;
    vN = normalize( gl_NormalMatrix * gl_Normal ); // normal vector
    vL = LIGHTPOSITION - ECposition.xyz; // vector from the point to the light position
    vE = vec3( 0., 0., 0. ) - ECposition.xyz; // vector from the point to the eye position
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```
Fragment shader: Per-fragment Lighting

```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 cords
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.;
    if(dot(Normal,Light) > 0.) // only do specular if the light can see the point
    {
        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, find out if the current fragment is within a particular rectangle:

```glsl
cvec3 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.
Setting up a Shader is somewhat Involved:
Here is a C++ Class to Handle the Shader Setup for You

**Setup:**

```cpp
GLSLProgram *Pattern;

// do this setup in InitGraphics()
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 into the console window and returns a value of false.

---

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

**Use this in Display():**

```cpp
float S0, T0;
float Ds, Dt;
float V0, V1, V2;
float ColorR, ColorG, ColorB;

Pattern->Use();
Pattern->SetUniformVariable("uS0", S0);
Pattern->SetUniformVariable("uT0", T0);
Pattern->SetUniformVariable("uDs", Ds);
Pattern->SetUniformVariable("uDt", Dt);
Pattern->SetUniformVariable("uColor", ColorR, ColorG, ColorB);

glBegin( GL_TRIANGLES );
    Pattern->SetAttributeVariable("aV0", V0 ); // don’t need for Project #5
    glVertex3f( x0, y0, z0 );
    Pattern->SetAttributeVariable("aV1", V1 ); // don’t need for Project #5
    glVertex3f( x1, y1, z1 );
    Pattern->SetAttributeVariable("aV2", V2 ); // don’t need for Project #5
    glVertex3f( x2, y2, z2 );
    glEnd();
Pattern->Use( 0 ); // go back to fixed-function OpenGL
```
2D Texturing

Vertex shader:

```cpp
#version 330 compatibility
out vec2 vST;

void main()
{
    vST = gl_MultiTexCoord0.st;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Fragment shader:

```cpp
#version 330 compatibility
in vec2 vST;
uniform sampler uTexUnit;

void main()
{
    vec3 newcolor = texture( uTexUnit, vST ).rgb;
    gl_FragColor = vec4( newcolor, 1.);
}
```

- **Setting Up Texturing in Your C/C++ Program**
  - This is the hardware Texture Unit Number. It can be 0-15 (and often higher depending on the graphics card).

```cpp
// globals:
unsigned char * Texture;
GLuint TexName;

// In InitGraphics():
 glGenTextures( 1, &TexName );
 int nums, numt;
 Texture = BmpToTexture( "filename.bmp", &nums, &numt );
 glBindTexture( GL_TEXTURE_2D, TexName );
 glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
 glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );
 glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
 glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
 glTexImage2D( GL_TEXTURE_2D, 0, 3, nums, numt, 0, 3, GL_RGB, GL_UNSIGNED_BYTE, Texture );

// In Display():
 Pattern->Use();
 glActiveTexture( GL_TEXTURE5 );  // use texture unit 5
 glBindTexture( GL_TEXTURE_2D, TexName );
 Pattern->SetUniformVariable( "uTexUnit", 5 );  // tell your shader program you are using texture unit 5
 << draw something >>
 Pattern->Use( 0 );
```
2D Texturing

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. (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:
  ```c
  GLenum err = glewInit();
  if (err != GLEW_OK )
  {
    fprintf( stderr, "glewInit Error\n" );
  }
  else
    fprintf( stderr, "GLEW initialized OK\n" );
  ```
  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" );
  ```
  after successfully initializing GLEW.
Declare the GLSLProgram above the main program (as a global):

GLSLProgram * Pattern;

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

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

// Use the Shader Program in Display():

Pattern->Use( );
Pattern->SetUniformVariable( ...)

// Draw the object here:

Sphere( );

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 OsuSphere( ) function from the texturing project will give you a very good sphere