Enhancing the Effects You Can Create with Computer Graphics by Using the OpenGL Shading Language (GLSL)

Mike Bailey
mjb@cs.oregonstate.edu

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License

Computer Graphics

The Basic Computer Graphics Pipeline, OpenGL-style

Model Transform

View Transform

glscalef

gluLookAt

gluPerspective

Texture Processing, Per-fragment Lighting

Fragment Shader

Vertex Shader

MC = Model Vertex Coordinates
WC = World Vertex Coordinates
EC = Eye Vertex Coordinates
NDC = Normalized Device Coordinates (-1.→1.)
SC = Screen Coordinates (pixels)

GLSL Variable Types

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.

GLSL is like C with Extensions for Graphics:

- Types include int,ivec2,ivec3,ivec4
- Types include float,vec2,vec3,vec4
- Types include bool,bvec2,bvec3,bvec4
- Vector components are accessed with .rgba, .xyzw, or.stpq
- Types include mat4
  - Computer Graphics uses 4x4 matrices to transform 3D vertices
- Types include sampler1D, sampler2D, sampler3D to access textures
- You can use parallel SIMD operations (doesn't necessarily get implemented in hardware):
  vec4 a = vec4(1, 2, 3, 4);
  vec4 b = vec4(5, 6, 7, 8);
  vec4 c = a + b;
- Vector components can be "swizzled" (to .abgr = from rgba)
- Type qualifiers: const, uniform, in, out
- Variables can have "layout qualifiers" to describe how data is stored
- The discard operator is used in fragment shaders to get rid of the current fragment

Computer Graphics uses quantities in groups of 2, 3, and 4
Computer Graphics uses 4x4 matrices to transform 3D vertices
The discard Operator Halts Production of the Current Fragment

\[
\text{if} ( \text{random number} < 0.5 ) \quad \text{discard}
\]

GLSL is Missing Some C-isms:

- No type casts — use constructors instead:
  \[
  \text{float} \ a = 3.14;
  \text{int} \ \text{i} = \text{int} \ a;
  \]
- No pointers, strings, or enums

Warning: Integer division is still integer division!

\[
\text{int} \ \text{n} = 2;
\text{int} \ \text{d} = 4;
\text{float} \ \text{f} = \text{float} (\text{n} / \text{d}); \quad \text{// still gives 0, just like C, C++, Python, and Java do}
\]

To get the correct result:

\[
\text{float} \ \text{f} = \text{float} (\text{n}) / \text{float} (\text{d});
\]

Our Shaders’ View of the Basic Computer Graphics Pipeline

There are actually four more GLSL shader types we won’t be covering here. In CS 457/557, we will cover all of them.

A GLSL Vertex Shader Takes Over These Operations:

- Vertex transformations
- Normal Vector transformations
- Computing per-vertex lighting (although, if you are using shaders anyway, per-fragment lighting looks better)
- Taking per-vertex texture coordinates \((s,t)\) and interpolating them through the rasterizer into the fragment shader

A GLSL Fragment Shader Takes Over These Operations:

- Color computation
- Texture lookup
- Blending colors with textures (like GL REPLACE and GL_MODULATE used to do)
- Discarding fragments

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

Input built-ins:

- \text{vec4} \ \text{gl\_Vertex}
- \text{vec3} \ \text{gl\_Normal}
- \text{vec4} \ \text{gl\_Color}
- \text{vec4} \ \text{gl\_MultiTexCoord0}
- \text{mat4} \ \text{gl\_ModelViewMatrix}
- \text{mat4} \ \text{gl\_ProjectionMatrix}
- \text{mat4} \ \text{gl\_ModelViewProjectionMatrix} (= \text{gl\_ModelViewMatrix} \times \text{gl\_ProjectionMatrix})
- \text{mat3} \ \text{gl\_NormalMatrix} (this is the transpose of the inverse of the MV matrix)

Output built-in:

- \text{vec4} \ \text{gl\_Position}

Note: While this all still works, OpenGL now prefers that you pass in all the above input variables as user-defined variables. We can talk about this later. For now, we are going to use the most straightforward approach possible.
Built-in Fragment Shader Variables You Will Use a Lot:

Output built-in \( \text{vec4 gl\_FragColor} = \) the RGBA being sent to the framebuffer

Note: While this all still works, OpenGL now prefers that you pass the RGBA out as a user-defined \text{out} variable. We can talk about this later. For now, we are going to use the most straightforward approach possible.

My Own Variable Naming Convention

With 7 different places that GLSL variables can be created 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>s</td>
<td>is a per-vertex \text{in} (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>t</td>
<td>came from the tessellation control shader</td>
</tr>
<tr>
<td>tc</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” GLSL — it is just my way of handling the chaos.

The Minimal Vertex and Fragment Shader

**Vertex shader:**

```glsl
#version 330 compatibility

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

**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 ...

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. Rasterizers interpolate built-in variables, such as the (x,y) position where the pixel will live and the pixel’s z-coordinate. They also interpolate the normal vector \((\text{nx,ny,nz})\) and the texture coordinates \((s,t)\). They can also interpolate user-defined variables as well.

A fragment is a “pixel-to-be”. In computer graphics, “pixel” is defined as having its full RGBA already computed and is headed to be stored in the framebuffer. A fragment does not yet have a computed RGBA, but all of the information needed to compute the RGBA is available. A fragment is turned into an RGBA pixel by the fragment processing operation.

A Little More Interesting, I: What if we Want to Color in a Pattern?

This pattern example is defined by three uniform variables: \(uS0, uT0,\) and \(uD,\) all in texture coordinates \((-1., 1.)\). \((uS0,uT0)\) are the center of the pattern. \(uD\) is the size of the pattern in texture coordinates. The \(s\) and \(t\) boundaries of the pattern are like this:

- \(s = uS0 - uD/2.\)
- \(s = uS0 + uD/2.\)
- \(t = uT0 - uD/2.\)
- \(t = uT0 + uD/2.\)

A Little More Interesting, II: Getting the Texture Coordinates from the Vertex Shader to the Fragment Shader

The vertex shader needs to pass the texture coordinates to the rasterizer so that each fragment shader gets it:

```glsl
#version 330 compatibility

out vec2 vST;

void main() {
    vST = gl_MultiTexCoord0.st;
    gl_FragColor = vec4( .5, 1., 0., 1. );
}
```

The texture coordinates need to come from the vertex shader because they are assigned to each vertex to begin with.
The fragment shader answers the question: "Am I (the current fragment) inside the pattern or outside it?"

**A Little More Interesting, II: Drawing a Pattern with the Fragment Shader**

The fragment shader:

```shading
uniform float uS0, uT0, uD; // from your program
in vec2 vST; // from the vertex shader, interpolated through the rasterizer
in vec3 vE, vL, vN; // eye, light, normal
vec4 ECposition = gl_ModelViewMatrix * gl_Vertex; // eye coordinate position
vec3 Eye       = normalize(vE); // vector from point to eye
vec3 Light     = normalize(vL); // vector from point to light
vec3 Normal = normalize(vN); // normal vector
const vec3 LIGHTPOSITION = vec3(  5., 5., 0. ); // light position
float s = vST.s; // the s coordinate of where this fragment is
float t = vST.t; // the t coordinate of where this fragment is
vec3 myColor = vec3( 1., 0.5, 0. ); // default color
vec3 mySpecularColor = vec3( 1.0, 1.0, 1.0 ); // specular highlight color
// from your program
if( uS0 - uD/2. <= s && s <= uS0 + uD/2. &&
    t <= uT0 - uD/2. && t <= uT0 + uD/2. ) {
    // this fragment is inside the pattern!
    myColor = vec3( 1., 0., 0. ); // change the pattern color if inside the pattern boundaries
}
}
```

**Drawing a Pattern on an Object**

Here's where we figure out what color this fragment will be: first before lighting:

```shading
vST = gl_MultiTexCoord0; // feature coordinates
vE = gl_ModelViewMatrix * gl_Vertex; // eye coordinate position
vL = LIGHTPOSITION - ECposition.xyz; // vector from the point to the light position
vN = normalize(gl_NormalMatrix * gl_Normal); // normal vector
vST = gl_ModelViewProjectionMatrix * gl_Vertex; // texture cords
```

Here's where we apply lighting to that color:

```shading
float d = dot(Normal,Light); // dot product of normal and light vector
vec3 diffuse = uKd * s * d; // diffuse (ambient + diffuse)
vec3 specular = uKs * s * pow( max( dot(Eye,ref),0. ), uShininess ); // specular
vec3 ambient = uKa * s; // ambient
vec3 SpecColor = mySpecularColor; // specular highlight color
vec3 ref = normalize(  reflect( -Light, Normal )  ); // reflection vector
SpecColor = SpecColor + diffuse + specular + ambient;
```

**If the Equation Defines a Square, Why Does the Pattern Look Like a Rectangle?**

In a sphere, the s coordinate encompasses twice as much angle (-180° → +180°) as the t coordinate does (-90° → +90°). So the same amount of "t" produces twice the distance as the same amount of "s". If you care, you can fix it like this:

```shading
float s = vST.s;
s = 2.*s;
```

All 4 of these must be true to conclude this fragment is inside the pattern!
Per-fragment Lighting is Good, Even Without a Pattern!

All together now!

First, follow these steps:

1. You will see two files that are already in your Sample folder: glslprogram.h and glslprogram.cpp
2. In your sample.cpp file, un-comment the line:
   #include "glslprogram.cpp"

These two files have been reduced to have just the shader features you need for Project #6.
If you are not working on Project #6, but are working on something bigger, I have more complete versions of glslprogram.h and glslprogram.cpp — just ask me.

GLSLProgram Pattern;
float                   Time;
#define MS_IN_THE_ANIMATION_CYCLE       10000
Put these in with the Global Variables:

Do this in Animate() like you've always done:

void Animate() {
    int ms = glutGet( GLUT_ELAPSED_TIME );  // milliseconds
    ms %= MS_IN_THE_ANIMATION_CYCLE;
    Time = (float)ms / (float)MS_IN_THE_ANIMATION_CYCLE; // [0., 1.)
}

Do this in InitGraphics() somewhere after where the window has been created and GLEW has been setup:

This attempts to load, compile, and link the shader program. If something goes wrong, Pattern.Create() prints error messages into the console window and returns a value of valid=false.

We cover the full GLSL API in CS 457/557.
Setting up a Shader is somewhat involved:
Here is our C++ class to simplify the Shader Setup for You

Do this in Display():

- float s0 = some function of Time
- float t0  = some function of Time
- float d   = some function of Time

Pattern.Use();  // turns the shader program on
// now no more fixed-function – the shader now handles everything
// but the shader program just sits there idling until you draw something

Pattern.SetUniformVariable( "uS0", s0 );
Pattern.SetUniformVariable( "uT0", t0 );
Pattern.SetUniformVariable( "uD",   d );

gCallList( SphereList ); // now the shader program has things to do
Pattern.UnUse();  // go back to fixed-function OpenGL

Setting Up Texturing with Shaders

Graphics chips have functionality on them called Texture Units. Each Texture Unit is identified by an integer number, typically 0-15, but oftentimes more.

To tell a shader how to get to a specific texture image, assign that texture into a specific Texture Unit number and then tell your shader what Texture Unit number to use. Your C/C++ code will look like this:

```c
glActiveTexture( GL_TEXTURE5 );  // use texture unit 5
glBindTexture( GL_TEXTURE_2D, TexName );
```

The Whole Process Looks Like This, I:

```c
// globals:
unsigned char *Texture;
GLuint TexName;
GLSLProgram Pattern;

// In InitGraphics():
glGenTextures( 1, &TexName );
int nums, numt;
Texture = BmpToTexture( "filename.bmp", &nums, &numt );
BindTexture( 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 );
gTexImage2D( GL_TEXTURE_2D, 0, 3, nums, numt, 0, 3, GL_RGB, GL_UNSIGNED_BYTE, Texture );
Pattern.Init();
bool valid = Pattern.Create( "pattern.vert", "pattern.frag" );
If( !valid ){
}
```

The Whole Process Looks Like This, II:

```c
// In Display():
Pattern.Use();
gActiveTexture( 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.UnUse();
```

2D Texturing within the Shaders

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

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

```c
#version 330 compatibility
in vec2 vST;

uniform sampler2D uTexUnit;

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

2D Texturing within the Shaders
What if You Want to Use Two Textures in a Shader?

// In Display( )
Pattern.Use();
glActiveTexture(GL_TEXTURE5);
glBindTexture(GL_TEXTURE_2D, TexName0);
glActiveTexture(GL_TEXTURE6);
glBindTexture(GL_TEXTURE_2D, TexName1);

Pattern.SetUniformVariable( "uTexUnit0", 5);
Pattern.SetUniformVariable( "uTexUnit1", 6);

<< draw something >>
Pattern.UnUse();

// In Shader:

void main( ) {
vec3 newcolor = texture( …
vec4 pos = gl_Vertex;
vec4 pos = gl_ModelViewMatrix * gl_Vertex;
vec4 pos = gl_ModelViewProjectionMatrix * gl_Vertex;
}

Why Would You Want to Use More Than One Texture in a Shader?

Once the RGBs have been read from a texture, they are just numbers. You can do any arithmetic you want with the texture RGBs, other colors, lighting, etc. Here is an example of blending two textures at once:

Something Goofy: Turning XYZs into RGBs in Model Coordinates

Setting rgb from the Untransformed xyz, I

Turning XYZs into RGBs in Eye (World) Coordinates
What's Different About These Two?

Set the color from the untransformed (MC) xyz:

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

Set the color from the transformed (WC/EC) xyz:

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

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, unless it came from Apple. In that case, who knows how much OpenGL support it has? (The most recent OpenGL level is 4.6)
- Update your graphics driver to the most recent version!
- Do the GLEW setup. It looks like this in the sample code:
  ```cpp
  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 created a graphics window. (It is this way in the sample code, but I'm saying this because I know some of you go in and "simplify" my sample code by deleting everything you don't think you need.)
- You use the GLSL C++ class you've been given only after a window has been created and GLEW has been setup. Only then can you initialize your shader program:
  ```cpp
  Pattern.Init();
  bool valid = Pattern.Create( "pattern.vert", "pattern.frag" );
  ```

Differences If You are on a Mac

- Apple froze their GLSL support at version 1.20 – here is how to adapt to that:
  ```cpp
  • Your shader version number should be 120 (at the top of the .vert and .frag files):
    #version 120 compatibility
  • Instead of the keywords in and out, use varying
  • Your OpenGL includes will need to look like this:
    #include <OpenGL/glew.h>
    #include <OpenGL/egl.h>
  • You don't need to do anything with GLEW
  • Your compile sequence will look like this:
    g++ -framework OpenGL -framework GLUT sample.cpp -o sample -Wno-deprecated
  ```

Hints when re-declaring vColor in "vec3 vColor = pos.xyz", you are making a local copy of it and writing into the local copy, not the out variable. The out version of vColor is never getting written to, and so the vColor in the fragment shader will have no value. (STW, C/C++ works the same way.)

Don't ever re-declare in, out, or uniform variables!

Abandon hope, all ye who do this
1. Declare the GLSLProgram above the main program (i.e., as a global):
   \[ \text{GLSLProgram } \text{Pattern}; \]

2. At the end of InitGraphics( ), create the shader program and setup your shaders:
   \[ \text{Pattern.Init();} \]
   \[ \text{bool valid } = \text{Pattern.Create( "pattern.vert", "pattern.frag" );} \]
   \[ \text{if( ! valid ) { . . . }} \]

3. Turn on the shader program in Display( ), set shader variables, draw the objects, then turn off the shader program:
   \[ \text{Pattern.Use();} \]
   \[ \text{Pattern.SetUniformVariables( . . . );} \]
   \[ \text{OsuSphere();} \]
   \[ \text{Pattern.UnUse(); } \text{return to the fixed function pipeline}; \]

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

---

Tips on drawing the object:

- If you want to use s and t coordinates in your shaders, the object must 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 must 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. Use it, not the GLUT sphere.