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

Mike Bailey
mjb@cs.oregonstate.edu

The Basic Computer Graphics Pipeline, OpenGL-style

Vertex Shader

Fragment Shader

OpenGL 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, vec2, vec3, vec4
- Types include float, vec2, vec3, vec4
- Types include bool, bvec2, bvec3, bvec4
- Vector components are accessed with .rgba, .xyzw, or stpq
- Types include mat4
- 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
The discard Operator Halts Production of the Current Fragment

```cpp
if (random_number < 0.5) {
    discard;
}
```

GLSL is Missing Some C-isms:

- No type casts -- use constructors instead:
  ```cpp
  float x = 3.14;
  int i = int(x);
  ```
- No pointers, strings, or enums

**Warning:** Integer division is still integer division!

```cpp
int n = 2;
int d = 4;
float f = float(n) / float(d); // still gives 0, just like C, C++, Python, and Java do
```

To get the correct result:

```cpp
float f = float(n / 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

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 = gl_ModelViewMatrix * gl_ProjectionMatrix` (this is the transpose of the inverse of the MV matrix)
- `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 input variables as user-defined variables. We can talk about this later. For now, we are going to use the most straightforward approach possible.

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 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 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 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 \((r=0.5, g=1., b=0., a=1.)\) 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 \((n_x, n_y, n_z)\) 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 length of each edge of the pattern. The \(s\) and \(t\) boundaries of the pattern are like this:

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

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_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    vST = gl_MultiTexCoord0.st; 
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

A Little More Interesting, I:

Getting the Texture Coordinates from the Vertex Shader to the Fragment Shader

The 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 each edge of the pattern. The \(s\) and \(t\) boundaries of the pattern are like this:

- \(s = uS0 - uD/2.\)
- \(s = uS0 + uD/2.\)
- \(t = uT0 + uD/2.\)
- \(t = uT0 - uD/2.\)
The fragment shader answers the question: "Am I (the current fragment) inside the pattern or outside it?"

```plaintext
void main()
{
  float s = vST.s; // the s coordinate of where this fragment is
  float t = vST.t; // the t coordinate of where this fragment is

  myColor = vec3(1., 0., 0.); // change the pattern color if inside the pattern boundaries

  if( uS0 - uD/2. <= s && s <= uS0 + uD/2. && uT0 - uD/2. <= t && t <= uT0 + uD/2. )
  {
    gl_FragColor = << possibly change myColor here >>
  }

  gl_FragColor = vec4(ambient + diffuse + specular, 1.);
}
```

### Applying Per-Fragment Lighting

The vertex shader sets up the necessary information for the fragment shader:

```plaintext
in vec2 uST; // from your program
in vec3 uE; // from the vertex shader, interpolated through the rasterizer
in vec3 uN; // normal vector
in vec3 uL; // vector from point to light
in vec3 uV; // vector from point to eye
in vec3 uLightPosition = vec3(5., 5., 0.);
const vec3 uSHININESS = vec3(1., 1., 1.);

void main()
{
  vST = gl_MultiTexCoord0; // feature coords
  vE = gl_NormalMatrix * uE; // eye coordinate position
  vL = uLightPosition - vE; // vector from the point to the light position
  uL = normalize(vL); // normalize vector
  vN = normalize(uN); // normal vector
  vST = gl_MultiTexCoord0; // feature coords
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

### Drawing a Pattern with the Fragment Shader

Here's where we figure out what color this fragment will be, like before:

```plaintext
float s = vST.s; // the s coordinate of where this fragment is
float t = vST.t; // the t coordinate of where this fragment is

myColor = vec3(1., 0., 0.); // change the pattern color if inside the pattern boundaries

if( uS0 - uD/2. <= s && s <= uS0 + uD/2. && uT0 - uD/2. <= t && t <= uT0 + uD/2. )
{
  myColor = vec3(1., 0., 0.); // new pattern color
}

gl_FragColor = << myColor with lighting applied >>
```

### Drawing a Pattern on an Object

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 a crisp edge. This is an advantage of procedural (equation-based) textures, as opposed to image-based textures.

```plaintext
float s = pow( max( dot(Eye, ref), 0. ), uSHININESS);
vec3 ref = normalize( reflect( -Light, Normal ) );
vec3 diffuse = uKD * d * vec3(uND, uNS, uNW); // diffuse lighting
vec3 ambient = uKa * a * vec3(uAD, uAN, uAW); // ambient lighting
vec3 specular = uKS * s * mySpecularColor; // specular lighting
vec3 myColor = vec3(1.0, 1.0, 1.0); // specular highlight color
```

### 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° to +180°) as the t coordinate does (-90° to +90°). So the same amount of "s" produces twice the distance as the same amount of "t". If you care, you can fix it like this:

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

### Applying Per-Fragment Lighting

Here's where we apply lighting to that color:

```plaintext
vec3 diffuse = uKD * d;
vec3 ambient = uKa * a;
vec3 specular = uKS * s;
vec3 mySpecularColor = vec3(1.0, 0.5, 0.0); // default color
vec3 myAmbientColor = vec3(1.0, 0.5, 0.0); // default ambient color
```
Applying Per-Fragment Lighting

Per-fragment Lighting is Good, Even Without a Pattern!

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:
   ```cpp
   #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.

Do this in Animate():

```cpp
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 at the end of InitGraphics():

```cpp
    Pattern.Init();
    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 `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();  // no more fixed-function – shaders now handle everything
Pattern.SetUniformVariable("uS0", s0);
Pattern.SetUniformVariable("uT0", t0);
Pattern.SetUniformVariable("uD", d);

OsuSphere();

Pattern.UnUse();  // go back to fixed-function OpenGL

The Whole Process Looks Like This, I:

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

// In InitGraphics():
glGenTextures(1, &TexName);
int nums, numt;
Texture = BmpToTexture("filename.bmp", &nums, &numt);

glBindTexture(GL_TEXTURE_2D,TexName);

Pattern.Init();

if (!valid) {
...}

The Whole Process Looks Like This, II:

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

// In Display():
Pattern.Use();

// draw something

2D Texturing within the Shaders

Vertex shader:

#version 330 compatibility
out vec2 vST;

void main()

{
    vST = gl_MultiTexCoord0.st;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

Fragment shader:

#version 330 compatibility

void main()

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

The Whole Process Looks Like This, III:

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

// In Display():
Pattern.Use();

// draw something

Pattern.UnUse();

Textures 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:

gActiveTexture(GL_TEXTURE5);  // use texture unit 5

2D Texturing within the Shaders

Texture Unit (in shaders) is a built-in function — it returns a vec4 (RGBA)

Texture Unit (in shaders) is a built-in function — it returns a vec4 (RGBA)
What if You Want to Use Two Textures in a Shader?

In Display():

Pattern.Use();

glActiveTexture(GL_TEXTURE5);
glBindTexture(GL_TEXTURE_2D, TexName0);

Pattern.SetUniformVariable("uTexUnit0", 5);

Pattern.SetUniformVariable("uTexUnit1", 6);

<< draw something >>

Pattern.Use();

// In Display():

Pattern.Use();

glActiveTexture(GL_TEXTURE5);
glBindTexture(GL_TEXTURE_2D, TexName0);

Pattern.SetUniformVariable("uTexUnit0", 5);

Pattern.SetUniformVariable("uTexUnit1", 6);

<< draw something >>

Pattern.Use();

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:

Visualization by Nick Gebbie

Textures used here:

• Day
• Night
• Heights (bump-mapping)
• Clouds
• Specular highlights

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

Something Goofy: Turning XYZs into RGBs in Model Coordinates

Vertex shader:

// set rgb from xyz!

vColor = gl_Vertex.xyz;

Fragment shader:

gl_FragColor = vec4(vColor, 1.);

Setting rgb from the Untransformed xyz, f

Turning XYZs into RGBs in Eye (World) Coordinates

Vertex shader:

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

Fragment shader:

gl_FragColor = vec4(vColor, 1.);

Turning XYZs into RGBs in Eye (World) Coordinates

Vertex shader:

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

Fragment shader:

gl_FragColor = vec4(vColor, 1.);
#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;
}

What's Different About These Two?

Set the color from the untransformed (MC) xyz:

vColor = (gl_ModelViewMatrix * gl_Vertex).xyz;

Setting rgb from the transformed xyz, if...

vColor = gl_ModelViewMatrix * gl_Vertex.xyz;

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

vColor = pos.xyz; // set rgb from 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:

```c
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:

```c
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:

• Your shader version number should be 120 (at the top of the .vert and .frag files):

```c
#version 120 compatibility
```

• Instead of the keywords in and out, use varying

• Your OpenGL includes will need to look like this:

```c
#include <OpenGL/gl.h>
#include <OpenGL/glu.h>
```

• You don't need to do anything with GLEW

• Your compile sequence will look like this:

```c
g++ -framework OpenGL -framework GLUT sample.cpp -o sample -Wno-deprecated
```

A Common Error to Look Out For

Here is a piece of code:

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

It looks like our example from earlier in these notes. It compiles OK. It should work, right?

Wrong! By 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):
   ```
   GLSLProgram Pattern;
   ```

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

3. Turn on the shader program in Display(), set shader variables, draw the objects, then turn off the shader program:
   ```
   Pattern.Use();
   Pattern.SetUniformVariable( ... 
   OsuSphere();
   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!

Tips on drawing the object:

- If you want to key off of 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.