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
The Basic Computer Graphics Pipeline, *OpenGL-style*

- **Model Transform**
  - `glRotatef`
  - `glTranslatef`
  - `glScalef`

- **View Transform**
  - `gluLookAt`

- **Projection Transform**
  - `gluPerspective`

- **Fragment Processing, Texturing, Per-fragment Lighting**
  - `glColor3f`
  - `glTexture*`
  - `glMaterial*`
  - `glLight*`

- **Rasterization**

**Vertices, Normal Vectors, Texture Coordinates**
- **MC**
- **WC**
- **EC**
- **NDC**
- **SC**

**Framebuffer**

**MC** = Model Vertex Coordinates  
**WC** = World Vertex Coordinates  
**EC** = Eye Vertex Coordinates  
**NDC** = Normalized Device Coordinates (-1.→1.)  
**SC** = Screen Coordinates (pixels)
The Basic Computer Graphics Pipeline, *OpenGL-style*

**Vertex Shader**
- Model Transform
  - `glRotatef`  
  - `glTranslatef`  
  - `glScalef`
- View Transform
  - `gluLookAt`
- Projection Transform
  - `gluPerspective`

**Framebuffer**
- RGBA pixels

**Fragment Processing, Texturing, Per-fragment Lighting**
- `glColor3f`
- `glTexture*`
- `glMaterial*`
- `glLight*`

**Rasterization**
- Vertices
- Fragments
- Pixels

**Transformations**
- MC = Model Vertex Coordinates
- WC = World Vertex Coordinates
- EC = Eye Vertex Coordinates
- NDC = Normalized Device Coordinates (-1.→1.)
- SC = Screen Coordinates (pixels)
The Basic Computer Graphics Pipeline, *OpenGL*-style

Vertex Shader

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

Fragment Shader

MC → NDC → Rasterization → SC → Framebuffer

MC → Vertices

NDC → Fragments

Vertex, Normal Vectors, Texture Coordinates

Framebuffer

RGBA pixels → Pixels

OpenGL-style Rasterization

Vertex Shader

- glTranslatef
- glScalef
- glRotatef
- gluLookAt
- gluPerspective

Fragment Shader

- glColor3f
- glTexture*
- glMaterial*
- glLight*

MC → NDC → Vertices

Fragments

SC → Frames
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
- Types include sampler1D, sampler2D, sampler3D to access textures
- You can use parallel SIMD operations (doesn’t necessarily get implemented in hardware):
  
  ```cpp
  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:
  
  ```
  float x = 3.14;
  int i = int( x );
  ```

- No pointers, strings, or enums

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

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

**To get the correct result:**

```
float f = float( n ) / float( 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)`
- `mat3 gl_NormalMatrix` (this is the transpose of the inverse of the MV matrix)

**Input built-ins**

**Output built-in**

*Note: while this all still works, OpenGL now prefers that you pass in all the above input variables as user-defined *in* 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

\[
\text{vec4 gl\_FragColor} = \text{the RGBA being sent to the framebuffer}
\]

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>a</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>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” GLSL – it is just my way of handling the chaos
The Minimal Vertex and Fragment Shader

Vertex shader:

```cpp
#version 330 compatibility

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

Fragment shader:

```cpp
#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 (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.
This pattern example is defined by three uniform variables: uS0, uT0, and uD, all in texture coordinates (0.-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.
- uS0, uT0 are the center of the pattern in texture coordinates
- uD is the size of the pattern in texture coordinates
A Little More Interesting, I:
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:

**The vertex shader:**

```glsl
#version 330 compatibility

out vec2 vST;

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

The texture coordinates need to come from the *vertex shader* because they are assigned to each *vertex* to begin with.
A Little More Interesting, II:
Drawing a Pattern with the Fragment Shader

The fragment shader answers the question: “Am I (the current fragment) inside the pattern or outside it?”

The fragment shader:

```glsl
#version 330 compatibility
uniform float uS0, uT0, uD; // from your program
in vec2 vST; // from the vertex shader, interpolated through the rasterizer

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
    vec3 myColor = vec3( 1., 0.5, 0. ); // default color

    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 >>
}
```

...
A Little More Interesting, III: Drawing a Pattern with the Fragment Shader

The fragment shader answers the question: “Am I (the current fragment) inside the pattern or outside it?”

```glsl
if( uS0 - uD/2. <= s && s <= uS0 + uD/2. && uT0 - uD/2. <= t && t <= uT0 + uD/2. ) {
    myColor = vec3( 1., 0., 0. ); // change the pattern color if inside the pattern boundaries
}
```

All 4 of these must be true to conclude this fragment is inside the pattern!

- uS0, uT0 are the center of the pattern in texture coordinates
- uD is the size of the pattern in texture coordinates
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.
It's because, 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 "s" produces twice the distance as the same amount of "t". If you care, you can fix it like this:

```c
float s = vST.s;
float t = vST.t;
s = 2.*s;
```
### Applying Per-Fragment Lighting

**Vertex shader:**

```glsl
#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; // eye coordinate position
  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;
}
```

---

Oregon State University

Computer Graphics

mjb – September 20, 2023
Applying Per-Fragment Lighting

Fragment shader:

```glsl
#version 330 compatibility
uniform float uKa, uKd, uKs; // coefficients of each type of lighting
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 myColor = vec3( 1.0, 0.5, 0.0 ); // default color
    vec3 mySpecularColor = vec3( 1.0, 1.0, 1.0 ); // specular highlight color

    vec3 ambient = uKa * myColor;
    float d = 0.;
    float s = 0.
    if( dot(Normal,Light) > 0. ) // only do specular if the light can see the point
    {
        d = dot(Normal,Light);
        vec3 ref = normalize( reflect( -Light, Normal ) ); // reflection vector
        s = pow( max( dot(Eye,ref),0. ), uShininess );
    }
    vec3 diffuse = uKd * d * myColor;
    vec3 specular = uKs * s * mySpecularColor;
    gl_FragColor = vec4( ambient + diffuse + specular, 1. );
}
```

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

Here’s where we apply lighting to that color
Applying Per-Fragment Lighting
Per-fragment Lighting is Good, Even Without a Pattern!

All together now!
Setting up a Shader is somewhat Involved: 
Here is our C++ Class to Simplify the Shader Setup for You

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.
Setting up a Shader is somewhat involved: Here is our C++ Class to Simplify the Shader Setup for You

**Put these in with the Global Variables:**

```cpp
GLSLProgram Pattern;
float Time;
#define MS_IN_THE_ANIMATION_CYCLE 10000
```
Setting up a Shader is somewhat Involved:
Here is our C++ Class to Simplify the Shader Setup for You

Do this in Animate():

```c++
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. )
}
```
Setting up a Shader is somewhat Involved:  
Here is our C++ Class to Simplify the Shader Setup for You

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
Do this in Display():

```c
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
```
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 file gl.h has these lines:

```c
#define GL_TEXTURE0                             0x84C0
#define GL_TEXTURE1                             0x84C1
#define GL_TEXTURE2                             0x84C2
#define GL_TEXTURE3                             0x84C3
#define GL_TEXTURE4                             0x84C4
#define GL_TEXTURE5                             0x84C5
#define GL_TEXTURE6                             0x84C6
#define GL_TEXTURE7                             0x84C7
#define GL_TEXTURE8                             0x84C8
...```

mjb – September 20, 2023
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 );
glBindTexture( GL_TEXTURE_2D, TexName );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
glTexParameteri( 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 );

Pattern.Init( );
bool valid = Pattern.Create( "pattern.vert", "pattern.frag" );
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).

```cpp
// 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.UnUse();
```

This is the hardware Texture Unit Number. It can be 0-15 (and often higher depending on the graphics card).
2D Texturing within the Shaders

**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 sampler2D uTexUnit;

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

*texture( ) is a built-in function – it returns a vec4 (RGBA)*
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();

Fragment shader:

#pragma version 330 compatibility
in vec2 vST;
uniform sampler2D uTexUnit0;
uniform sampler2D uTexUnit1;

void main()
{
    vec3 newcolor = texture( ... 
    gl_FragColor = ... 
}
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:
Why Would You Want to Use More Than One Texture in a Shader?

Textures used here:
• Day
• Night
• Heights (bump-mapping)
• Clouds
• Specular highlights

Visualization by Nick Gebbie
**Something Goofy: Turning XYZs into RGBs in Model Coordinates**

**Vertex shader:**
```glsl
#version 330 compatibility
out vec3 vColor;

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;

void main( )
{
    gl_FragColor = vec4(vColor, 1.);
}
```
Setting rgb from the Untransformed xyz, I

vColor = gl_Vertex.xyz;
Turning XYZs into RGBs in Eye (World) Coordinates

**Vertex shader:**

```glsl
#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:**

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

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

Set the color from the **untransformed (MC) xyz**:

```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;
}
```

---

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

```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;
}
```
Setting rgb from the Transformed xyz, II

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

vColor = gl_Vertex.xyz;

vColor = ( gl_ModelViewMatrix * 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:

  ```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" );
  ```
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. (BTW, C/C++ works the same way.)

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

Trust me, you will do this sometime. It's an easy mistake to make mindlessly. I do it every so often myself.
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):
  #version 120 compatibility

- Instead of the keywords `in` and `out`, use `varying`

- Your OpenGL includes will need to look like this:
  `#include <OpenGL/gl.h>
  #include <OpenGL/glu.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`
1. Declare the GLSLProgram above the main program (i.e., as a global):

```cpp
GLSLProgram Pattern;
```

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

```cpp
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:

```cpp
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.