openGL
Shader Designer

OpenGL Shading Language Tutorials

Written by Eirik Hoem

The OpenGL Shader Designer is distributed as freeware at www.typhoonlabs.com.

The author takes no responsibility for damage to hardware, software or other inflicted by usage of these tutorials.

The code has been tested on a GeForce FX 5600 / 256mb running the GLSL beta drivers (Detornator 57.10) from NVIDIA. Since the implementation is unfinished, I've done a few hacks to make the code work. Nothing fancy, so it should be easy to follow. Feedback will be greatly appreciated. Mail comments and suggestion to eirikhm@typhoonlabs.com

These tutorials are based on the usage of the OpenGL Shader Designer found at www.typhoonlabs.com. All shaders will be developed within that IDE, but a host application framework will also be available soon. The shader designer supports easy usage of texture units and uniform variables, but where it's appropriate I'll include sample code in C++. **Please note that this version is a WIP version of the tutorials.** That means that it's not complete and parts may be added / deleted and / or changed. Spellchecking and grammar correction will be done later, as well as proofreading and code tests.

Tutorial 0 – Getting to know the IDE

Here is a short introduction to the IDE we'll use in the following tutorials. It should be easy to follow since the editor has a light and organized GUI. To get started, open the project called sample_shader. It's located in tutorials\shaders\. Don't let the code scare you off, we'll go trough all of that later. This shader shows off an animated multi texture effect.

First let's take a look at the main window. To your right you have the code view. This is where you'll write your shader code. At the top of the code window you'll find some tabs. There is one tab for each of the files in the project. The current project has one vertex shader and one fragment shader, so there are only two tabs. To the left you see the preview window where the current shader is applied to a teapot model. Under that you'll see the uniform variables, and at the bottom the compilation output window.

The mail toolbar is located at the top of the program. It holds buttons for the most used functions and dialogs. The first ones from the left are quite obvious. It's the standard new, save and load. The next five is a bit more interesting. They are called Textures, Vertex states, Fragment states, Light & materials and Perspective.

Textures: This dialog has an overview over the texture units. That includes all bound textures, filtering and texture generation. The currently bound textures are previewed in the right side of the dialog. Cube maps can also be loaded here.

Vertex states: The vertex states dialog lets you change the vertex states such as polygon mode, back face culling and so on.

Fragment states: This one lets you change settings for the fragment processor. You'll find depth testing, blending and alpha testing among others.

Light and Materials: All material and lighting properties are located here. You'll find everything from spotlights to front and back materials. Color pickers are used to easily find the right colors. All the properties in this dialog can be accessed trough the vertex and fragment shaders. A nice feature here is saving and loading light and material presets. We will include several light and material presets.

Perspective: Changes to the Perspective settings are done here.

The next 8 buttons are file operations for the vertex and fragment shaders. After those we find five buttons for changing the preview mesh. The two last are texture and attribute plug-in managers.

Hotkeys:
(This is not the final list)

Tutorial 1 – Getting started with GLSL and the OpenGL Shader Designer

This tutorial will explain a basic shader in GLSL, consisting of a vertex shader and a pixel shader. To get started, fire up the Shader Designer and create a new project. The editor will automatically create blank source files for your shader. Save the project as "firstshader". The vertex and fragment shader files will be saved with the same name as the project.

Vertex shader:

First we are going to code the vertex shader. Select the vertex shader code window and type in the following:

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

Code explained:

#1. The start of the vertex shader, just like you'd expect in a C program.
#2. The opening bracket.
#3. Multiply the current vertex with the model view projection matrix and stores the result in gl_Position.
#5 closing bracket for the shader.

That wasn't so bad? Let's move on.

Fragment shader:
It's time to code the fragment / pixel shader. Select the tab for pixel shader file in the code window and type in the following:

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

Code explained:

#1 Indicates the start of the pixel shader, just like in the vertex shader.
#2 Opening bracket.
#3 Creates a variable of the type vec4, and fills it with the values 1.0, 0.0, 0.0, 1.0. This represents a red color.
#4 Sets gl_FragColor to the color we just selected. gl_FragColor is the output pixel.

That's all there is to it. Not too hard eh? This shader just transforms the position to world space and set the resulting fragment color to red. Just hit F4 to compile and link the shaders. You should now see a red sphere in the preview window located in the upper left corner of the IDE.

Tutorial 2 – Texturing

This tutorial will show you how to do simple texturing and multi texturing in GLSL. First create a new project called “textures”. Then go to the Project menu and select Textures. Add a texture for unit1 and press accept. (To check if it’s loaded, press refresh).

Vertex Shader:

Type in the following vertex program.

```
void main(void)
{
            gl_TexCoord[0] = gl_MultiTexCoord0;
            gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

The code here is equal to the code in tutorial 1, so I won’t explain it further.

Fragment Shader:

Type in the following pixel shader.

```
uniform sampler2D tex;

void main(void)
{
            vec4 value = texture2D(tex, vec2(gl_TexCoord[0]));
            gl_FragColor = value;
}
```

Code explained:
#1 creates a texture sampler for a 2 dimensional texture. Uniform is a keyword that tells the shader that the variable gets its value from the host application. We’ll take a closer look at uniforms later. #5 creates a vec4 variable that holds the color value for the current fragment. The sampler2D() operation gets the value at gl_Texcoord[0] in the sampler2d called tex.
#6 sets the current fragment color to the color we got from the texture.

That’s all that’s needed for basic texturing. We just get the fragment color from the texture and set it in the shader. I decided to include the code for multi texturing in the same tutorial. It’s not much different from normal texturing. First add a texture for texture unit two, just as you did at the start of this tutorial.

Then add the following line to the vertex shader:

```
gl_TexCoord[1] = gl_MultiTexCoord1;
```

That makes sure that we get the texture coordinates for texture unit 1 as well.

After that do the following changes in the pixel shader:

```
uniform sampler2D tex1;
uniform sampler2D tex2;

void main(void)
{
            vec4 value1 = texture2D(tex1, vec2(gl_TexCoord[0]));
            vec4 value2 = texture2D(tex2, vec2(gl_TexCoord[1]));

            gl_FragColor = (value1+value2) * 0.5;
}
```

The difference in the pixel shader is that we now get two values from two texture units. When we're setting the fragment color value we add these two values together and multiply by 0.5. By doing this we end up with the average color of the two. Compile and link by pressing F4.

If you want more textures, just extend the shader the same way we just did.

Tutorial 3 – Color keying

This tutorial will show the usage of an operation called discard. It lets you discard (not draw) fragments. We'll code a shader that simply discards all red fragments. Create a new project called colorkey, and add a texture to texture unit 1. I've used a 256x256 texture with one red, one green, one blue and one white square.

Vertex shader:
The vertex shader is just like in the texture tutorial.

Fragment shader:

```
void main(void)
{
        vec4 value = sampler2D(tex, vec4(gl_Texcoord[0]));
        vec4 color = { 0.95, 0.0, 0.0, 1.0};

        if ((value.x >= color.x) && (value.y == color.y) && (value.z == color.z))
                discard;

        gl_FragColor = vec4(value.r, value.g, value.b, 1);}
}
```

Code explained:
#6-7 compares the color value from the texture with the color value we created. If they are equal, the current fragment gets discarded.

As you've probably noticed, the color we test against isn't pure red.

Tutorial 4 – Varying variables

You'll now learn a way to share variables between vertex and pixel shaders. This is done by using so-called varying variables. They are given values in the vertex shader and interpolated when accessed from the fragment shader. Create a new project and call it "varying".

Vertex shader:

Type in the following vertex shader:

```
varying vec4 color;

void main(void)
{
        color = vec4(1.0, 0.0, 0.0, 1.0 );
        gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Code Explained:
#1 creates a varying variable that can be accessed by the fragment shader.

#4 sets the value of the varying variable

Fragment shader:

```
varying vec4 color;

void main(void)
{
	gl_FragColor = color;
}
```

Code explained:
#1 declared the varying variable we created in the vertex shader. This variable already has a value, so we just use it without further hassle.

That's wasn't so bad, was it? In the next tutorial we are going to look at uniform variables.

Tutorial 5 – Uniform variables

What good are shaders if you can't pass values from your host application? Uniform variables are variables that are passed from the host application. In the Shader Designer ID you've got a list of uniforms at the left side of the main window. We have already used uniforms to access texture units. Now you'll hopefully understand more of what's going on. Remember that uniform variables are read-only.

Go ahead and create a new project called uniforms, and then add a uniform float variable called opacity. Set the value to 0.5.

Vertex shader is the standard "get texcoord and set position".
Fragment shader:

```
uniform float opacity;
uniform sampler2D tex;

void main(void)
{
	vec4 value = texture2D(tex, vec2(gl_TexCoord[0]));
	gl_FragColor = vec4(value.x, value.y, value.z, (value.w * opacity));
}
```

Code explained:
#1 declares a float uniform variable called opacity. This variable gets it value from the host application
#7 sets the color of the current fragment. The opacity is changed by multiplying the last component of value by the uniform float variable called opacity.

Tutorial 6 – Common GLSL functions and operations

This tutorial aims to explain and show some of the most common functions and operations in GLSL. No working shader code in this tutorial, just an overview.

Note : 3x float means float, float, float and not vec3.

normalize()

Returns: vector
Arguments: vector
Explanation: Returns the normalized vector

pow()

Returns: float
Arguments 2x float
Explanation: Arg1 ^ Arg2

abs()
Returns: float
Arguments float
Explanation: returns the absolute value for the passed value

fract()
Returns: float
Arguments: float
Explanation: simulates a noise effect based on the argument passed.

max()
Returns: float?
Arguments: 2x float
Explanation: returns the highest value of the two passed.

dot()
Returns: float
Arguments: 2xvector
Explanation: returns the dot product of the two passed vectors

smoothstep()
Returns: float
Arguments: 3x float
Explanation: Interpolates?

reflect()
Returns: vector
Arguments: 2x vector
Explanation: returns the reflected arg1 around arg2.

clamp()
Returns: 3xfloat
Arguments: float
Explanation: returns arg1 clamped between arg2 and arg3

sqrt()
Returns: float
Arguments: float
Explanation: returns the square root of the passed float

inversesqrt()
Returns: float
Arguments: float
Explanation: returns the inverse square root of the passed float

saturate()
Returns: float
Arguments: float
Explanation: saturates the passed argument and returns the new value

texture2D()
Returns: vec4
Arguments: sampler2D, vec2
Explanation: returns the fragment value at the position specified by the second argument.

Tutorial 7 – Lighting (wip)

Now it's time to see how we can use GLSL to make some cool effects. First out is lighting. It's not as hard as it may sound. This method uses the ordinary openGL lighting and material states. By doing that, it's very easy to disable the shader if it's not supported by the graphics card. Set the following states (screenshot) in the lighting and material settings.

Vertex shader:

```
varying vec3 v;
varying vec3 N;

void main(void)
{
        v = vec3(gl_ModelViewMatrix * gl_Vertex);
        N = normalize(gl_NormalMatrix * gl_Normal);

        gl_TexCoord[0] = gl_MultiTexCoord0;
        gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Code explained:
#6 computed vertex, worldspace?
#7 computed normal, worldspace?

Fragment shader:

```
varying vec3 v;
varying vec3 N;
uniform sampler2D TextureUnit0;

void main(void)
{
        vec3 L = normalize(gl_LightSource[0].position.xyz - v);
        vec3 E = normalize(-v);
        vec3 R = normalize(-reflect(L,N));
        vec4 amb, spec, diff;
        vec4 basecolor = texture2D(TextureUnit0, vec2(gl_TexCoord[0]));

        amb = gl_FrontLightProduct[0].ambient * gl_FrontMaterial.ambient;

        spec = gl_FrontLightProduct[0].specular
        * pow(max(dot(R,E),0.0),0.1);

        diff = gl_FrontLightProduct[0].diffuse * max(dot(N,L), 0.0);

        gl_FragColor = (basecolor) + amb + spec + diff;
}
```

Code explained:

Pheeew! That was a bit harder than the previous tutorials. This one is a bit heavier on the math, but I'll try to give a brief explanation it. If you are interested in the topic, check you the book called Real-Time Rendering (www.realtimerendering.com) and Mathematics for #D Game Programming & Computer Graphics. They are both great!

First we go trough the vertex program. We compute the normal for the current vertex and store in N, and then we compute the position for the current vertex in world space and store it in v. The rest is just like the earlier tutorials.

Then we take a look at the pixel shader. First of we find the vector from the current vertex to the light source. That's done by subtracting the vertex position from the lighthouse position. The result is stored in ***L***. After that we compute the eye vector by normalizing the inversed vector from the vertex to the light. This vector is called the Eye Vector. The result is stored in ***E***. Then we compute the reflected vector from the light vector and the normal. We store it in ***R***.

Now that we got all the vectors ready it's time to get some material properties. These properties are the same ones that you set earlier. First out is ambient. We simply grab it from gl_FrontLightProduct[0].ambient and store it in a vec4 called amb. Next out is the specular property. It's a bit trickier. First we grab the base property from gl_FrontLightProduct[0].specular. That's the easy part. Then we compute the dot product of R and E, and find the max value of the dot product and 0.0. Then we pow the value we get with (0.3 * gl_FrontMaterial.shininess). Looks a bit messy, but if you brake it up it's quite simple. I've added some illustrations to help explain. Let's go for the diffuse. First we grab it from gl_FrontLightProduct[0].diffuse, just like the rest, then we compute the dot product of N and L, and fix max max value of it and 0.0. The value we end up with gets multiplied with the diffuse.

Are you still with me? It's time to set the color of the fragment. We add amb, spec and diff, together with the base color.

That's all there is to it. If you have problems understanding this, it might help to sketch up the vectors and objects on paper.

Tutorial 7 – Per Pixel Lighting (Bump mapping (wip)

It's time to look at some per pixel lighting. Some of you know the result as bump mapping. We'll use the previous tutorial as a base for this one. For generating normal maps, I suggest you get NVIDIA's normal map generator plug-in for Adobe Photoshop. You" find it from the following site http://developer.nvidia.com/object/nv_texture_tools.html

The concept of normal maps is quite easy. A normal map holds one normal per pixel. The three components of the normal are encoded in the R, G and B values of the pixel from the normal map. When we want to do some fancy bump mapping we just extract the normal from the normal map and use it in our lighting equations.

The vertex shader is the basic one. First texture holds the base texture; the second holds the normal map.

The code is not much different. The only think that really changes is the way we calculate the normal. In the previous tutorial we used normals per vertex, but now we are using them per pixel. It's done by the following code:

```
vec3 N = texture2D(TextureUnit1, vec2(gl_TexCoord[1]));
```

Compile and see the results.

Tutorial 8 – Grayscale

This will be a short tutorial. Converting to grayscale is not much work. Some very basic math is all that's needed. We have three constant factors (R,G,B) that we multiply with the value of each of the components for fragment being processed (frag.r *R, frag.g * G, frag.b * B). Then we sum up the three values we get and use them as out values for the current fragment. This sample uses a texture so the effect is easy to see. The vertex shader is the same basic one with setup for one texture unit.

Fragment shader:

```
uniform sampler2D tex0;

const vec3 fact = {0.30, 0.59,0.11};

void main(void)
{
        vec4 value = texture2D(tex0, vec2(gl_TexCoord[0]));
        vec3 new;
        float result;

        new = value * fact;

        result = (new.x + new.y + new.z);

        gl_FragColor = vec4(result,result,result,1);
}       gl_FragColor = vec4(result,result,result,1);
```

Code explained:
We first set up the three constant factors (these could easily be changed to uniforms so we get control over the result from the host application). Then we get the color of the current fragment and multiply the components with the factors. We sum the results and store them in a vec4 called result. Note that the components in the vec4 result all have the same value. This is the “secret” behind grayscale. And that’s all for now.

Tutorial 8 – Clouds (wip)

Back in black! This tutorial is the first in a series of procedural texture generation. First out is clouds. We’ll be using Perlin Noise for this shader. The math isn’t that hard, and I’ll try to explain everything at a basic level. The noise in this tutorial will be generated in the fragment shader. Expect some cool displacement mapping in vertex shader once I get my NV40.

Stay tuned for the next release of the OpenGL Shader Designer, with more tutorials, more features and more presets.

Visit www.typhoonlabs.com for the latest updates!

Eirik Hoem
Oslo 6th of May 2004