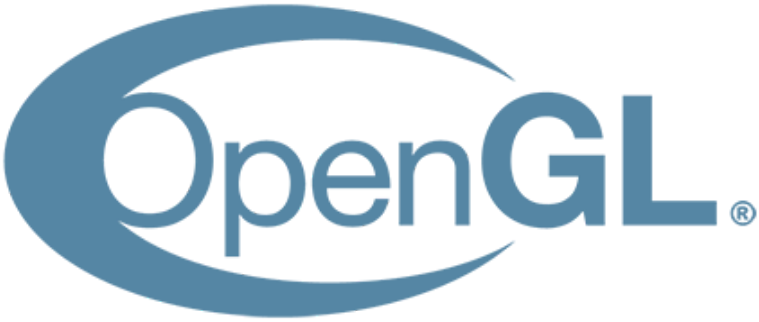
Shaders in OpenGL

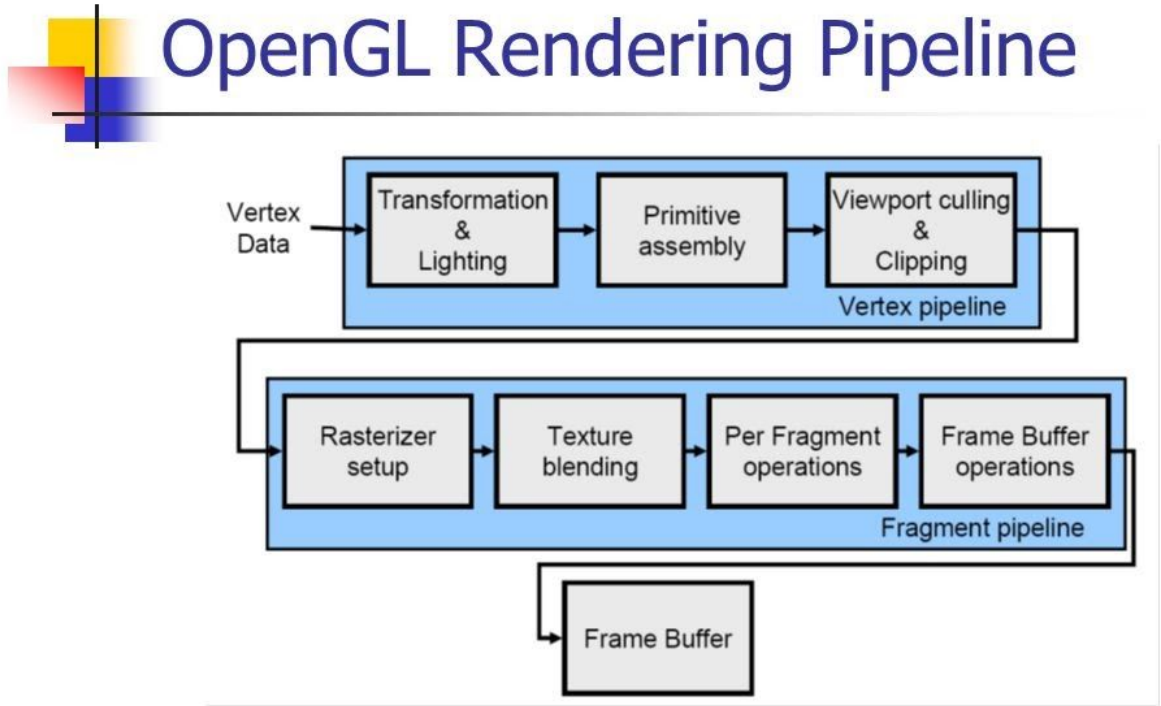


Toon Rim Shader

By Anders L. Gillies

Introduction to Shaders

Shaders are segments of code that are graphics processing unit (GPU) based. These segments of code are referred to as programs. These programs run for each point of the graphics pipeline. Shaders are just programs that handle inputs and give outputs. These programs that are GPU bound are also independent. Shaders do not share data between themselves. The only communication that they have with each other is through their inputs and outputs. Below is a picture of a very basic shader program. (learnopengl.com, n.d.)



(Figure 1) (slideplayer.com, n.d.)

Creating Custom Shaders

Shader programs in OpenGL are completely isolated on the GPU, this means when you create a shader you need to ensure that it is loaded correctly and is binded correctly. When the 'ToonRimShader' (the custom shader created for this submission) is being loaded we need to load two different files. These files are the ToonRimShader.vert and the ToonRimShader.frag.

```
62      shaderToon.init("..\\res\\shaderToon.vert", "..\\res\\shaderToon.frag", false);
63      RimShader.init("..\\res\\RimShader.vert", "..\\res\\RimShader.frag", false);
```

(Figure 2)

The premise of the ToonRimShader was to combine the separate Rim Shader and the Toon Shader. These two effects are combined in the ToonRimShader.frag program.

(Figure 3)

Layout Qualifiers

"Layout qualifiers are sometimes used to define various options for different shader stages." (www.khronos.org, n.d.) These stage options affect the input stage or the output stage (in or out). The data can be assigned in the shader c++ program. This is done by using glBindAttribLocation, in this example it shows the binding of the "VertexPosition", "VertexTexCoord" and the "VertexNormal".

```
26      }
27
28      glBindAttribLocation(program, 0, "VertexPosition");
29      glBindAttribLocation(program, 1, "VertexTexCoord");
30      glBindAttribLocation(program, 2, "VertexNormal");
31
```

When the data has been bound it is then the "VertexPosition" and the "VertexNormal" is utilized by the vertex shader as shown below.

```
3   layout (location = 0) in vec3 VertexPosition;
4   layout (location = 2) in vec3 VertexNormal;
```

Input and Output Variables

Input and Output variables are used to send data from the vertex shader to the fragment shader. The basic structure of these is if you have a variable called “CalcuX” as an output (out) within the vertex shader then there must be a variable called the exact same in the fragment shader. However, the difference is that the variable in the fragment shader must be an input (in). For the ToonRimShader one of the values that are passed is called “Normal”. So within the vert there is an out vec3 Normal and within the fragment there is an in vec3 Normal.

```
6   out vec3 Normal;
```

```
6   in vec3 Normal;
```

Uniform Variables and Shaders

Uniform variables are important variables when programming shaders with OpenGL. Uniform variables allow data types (vec4, floats and so forth) to be carried across into your shader programs. However, uniform variables are only ever allowed to be read within your shader programs. If you wish to edit and or change their values then you need to carry that out within your C++ program (not the shader).

When changing or modifying a uniform variable we actually need to link the variable up to the program. This is done by retrieving the location of the uniform variable within a specified object, this can be done by using the glGetUniformLocation function. Within the application being used this is done in the following section.

```
void setBool(const std::string &name, bool value) const
{
    glUniform1i(glGetUniformLocation(program, name.c_str()), (int)value);

    if ((glGetUniformLocation(program, name.c_str()) == -1))
    {
        std::cerr << "Unable to load shader: " << name.c_str() << std::endl;
        __debugbreak();
    }
}
```

(Figure 4)

In figure 3 that only sets up uniform booleans, this would need to be recreated for each variable type that is needed for the application.

For the ToonRimShader there are only three uniform variables used. One for the rim shader aspect, one for the toon shader aspect then one for the matrice in the .vert file. These uniform variables are named “LightDir” which will hold the data for light direction within the application (Figure 4), then “IVD” which will hold the data of the camera’s rotation within the application (Figure 4) and “transform” which will be used to calculate the gl_Positon. These values are then used to calculate the shading required for the toon and then the rim around the mesh of the object.

```
3   //Toon
4   uniform vec3 LightDir;
5
6   in vec3 normal;
7
8   //Rim Vars
9   uniform vec3 IVD;
```

(Figure 5)

```
9   uniform mat4 transform;
```

(Figure 6)

Transform Variable

An important aspect of the entire application is the transform.h file (header file). This header file handles everything to do with the transform of the current mesh being drawn. This includes the position, rotation and scale of the object. After the mesh has been drawn the shader function is called. This function then utilizes the transform of the mesh and the camera component. This line is shown below :

```
18      void Update(const Transform& transform, const Camera& camera);
```

This is how the model view projection matrix is calculated. This is done by multiplying the view project of the camera and the transform of the model. This calculated value is then put into a uniform variable to be used by the vertex shader for future graphic calculations.

.Vert and .Frag ToonRimShader

.Vert

The ToonRimShader.vert file takes in a mat4 variable called transform. This is then used to calculate the gl_Position by multiplying by the vect4(VertexPosition, 1.0f). However, all the 'out' values that are used in the fragment shader are calculated in the main body of the .vert program. The variable 'Normal' is then used to calculate the light intensity within the fragment program.

```
1   #version 400
2
3   layout (location = 0) in vec3 VertexPosition;
4   layout (location = 2) in vec3 VertexNormal;
5
6   out vec3 Normal;
7   out vec4 v_pos;
8
9   uniform mat4 transform;
10
11  void main()
12  {
13      Normal = VertexNormal;
14      gl_Position = transform * vec4(VertexPosition, 1.0);
15  }
```

.Frag

The ToonRimShader.frag purpose is to calculate all the effects for the shader using certain values.

```
#version 400

//Toon
uniform vec3 LightDir;

in vec3 Normal;

//Rim Vars
uniform vec3 IVD;

//out vec4 FragColor;

vec4 Toon();
vec4 Rim();

void main()
{
    gl_FragColor = Toon() * Rim();
}

vec4 Toon()
{
    float Intensity;
    vec3 Colour;

    Intensity = dot(LightDir, Normal);
    if(Intensity > 0.9)
    {
        Colour = vec3(1, 0, 0);
    }
    else if (Intensity > 0.7)
    {
        Colour = vec3(0.7, 0, 0);
    }
    else if (Intensity > 0.5)
    {
        Colour = vec3(0.5, 0, 0);
    }
    else
    {
        Colour = vec3(0.3, 0, 0);
    }

    return vec4(Colour, 1.0); //Returning vec4 cause (R,G,B,Alpha)
}

vec4 Rim()
{
    float RimShader = 1 - max(dot(Normal, IVD), 0.0);
    vec3 Col = vec3(smoothstep(0.0, 1.0, RimShader));
    return vec4(Col, 1.0);
}
```

(Figure 7)

Within Figure 7 you will see a Toon() function, this is where the toon shading effect is calculated. Within this example we will see four different layers. These layers are defined by the intensity at that point in the mesh. The intensity layers are 1, 0.7, 0.5 and

0.3. With the decreasing intensity the colour is also reduced giving that blending effect. This can be built upon to create interesting effects with the toon shader. By dividing the intensity sections even further we could generate ten different layers with each layer having a different colour assigned to it. This will result in a shader that looks similar to infrared filters in video games. The images below show the modified code and the result.

```
{
    float Intensity;
    vec3 Colour;

    Intensity = dot(LightDir, Normal);
    if(Intensity > 0.9)
    {
        Colour = vec3(1,0, 0);
    }
    else if (Intensity > 0.8 && Intensity < 1)
    {
        Colour = vec3(0.9, 0, 0);
    }
    else if (Intensity > 0.8)
    {
        Colour = vec3(0.8, 0, 0);
    }
    else if (Intensity > 0.7)
    {
        Colour = vec3(0, 0.7, 0);
    }
        else if (Intensity > 0.6)
    {
        Colour = vec3(0, 0.6, 0);
    }
    else if (Intensity > 0.5)
    {
        Colour = vec3(0, 0, 0.5);
    }
    else if (Intensity > 0.4)
    {
        Colour = vec3(0, 0.4, 0);
    }
    else if (Intensity > 0.3)
    {
        Colour = vec3(0.3, 0, 0);
    }
    else if (Intensity > 0.2)
    {
        Colour = vec3(0, 0.2, 0);
    }
    else
    {
        Colour = vec3(0, 0, 0.1);
    }

    return vec4(Colour, 1.0); //Returning vec4 cause (R,G,B,Alpha)
```

(Figure 8)

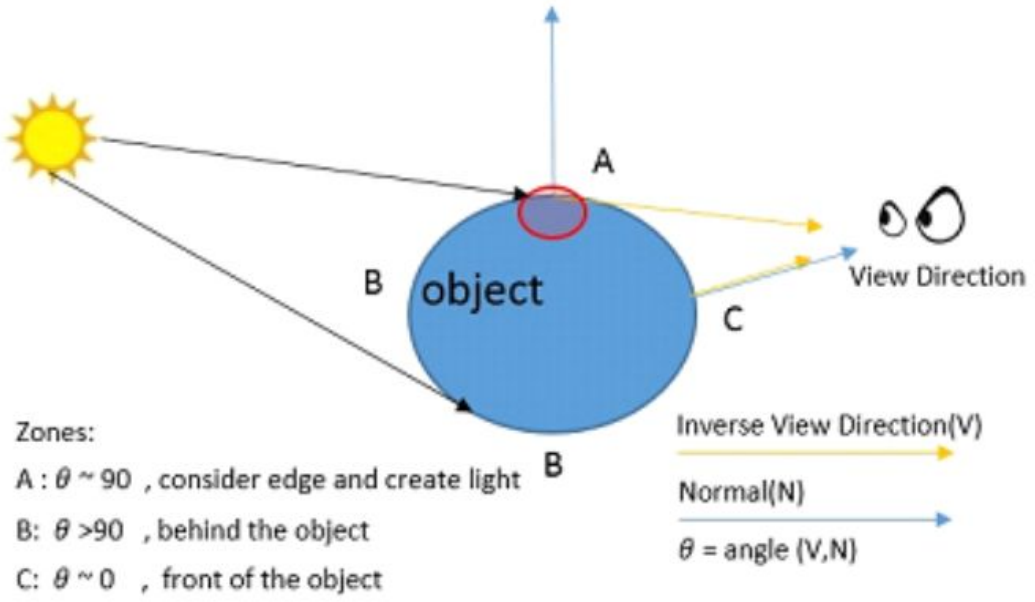
As well as the Toon() function within the .frag program there is also a Rim() function (Figure 7). This function is designed to create the shadows around the mesh. It calculates float value called RimShader by finding the dot product of the two vectors which are the Normal and the IVD (Inverse View Direction), the IVD is calculated by calculating the dot product max dot product between the Normal and the Inverse

Direction View and then subtracting it from 1. After the float value which is called RimShader is calculated from that it is then used to work out the colour vec3 variable which will be applied. This is done through the smoothstep function and feeding (0,1, RimShader) into the vec3. This is shown below :

```
71   vec4 Rim()
72   {
73       float RimShader = 1 - max(dot(Normal, IVD), 0.0);
74       vec3 Colour = vec3(smoothstep(0.0, 1.0, RimShader));
75       return vec4(Colour, 1.0);
76   }
```

(Figure 11)

Below is a diagram outlining the process of calculating the inverse view direction :



(Figure 12)

Combining Rim and Toon Shader

Combining the rim and toon shader takes place within the main void in the ToonRimShader.frag program. This is carried out by multiplying the two vec4 Toon() and Rim() together once they have both been calculated. This is then placed into the gl_FragColour which in turn is then applied to the mesh. Which will display the shader in the game view.

```
16  void main()
17  {
18      gl_FragColor = Toon() * Rim();
19  }
```

Practical Implementation of Shader

With this new version of the toon shader, there are now a vast amount of options that can be presented. The shader logic statement could be condensed into a more reusable loop. This loop could also apply a random colour giving the option to completely change visuals with every draw. This could be useful in a game about mental health and so forth. However, the current layout of the shader resembles that of an infrared weapon scope visual. The shader could be easily modified to account for the material/texture applied to the mesh. Then mixing that with the intensity variable could produce an accurate heat visual for an enemy character and so forth, non mixed version of the more complex toon rim shader shown below.

(Figure 9)

Sources

learnopengl.com. (n.d.). *LearnOpenGL - Shaders*. [online] Available at: https://learnopengl.com/Getting-started/Shaders [Accessed 1 Aug. 2020].

slideplayer.com. (n.d.). *OpenGL Shading Language (GLSL) - ppt video online download*. [online] Available at: https://slideplayer.com/slide/8927199/27 [Accessed 1 Aug. 2020].

www.opengl.org. (n.d.). *Tutorials*. [online] Available at: https://www.opengl.org/sdk/docs/tutorials/ClockworkCoders/uniform.php [Accessed 2 Aug. 2020].

www.khronos.org. (n.d.). *Layout Qualifier (GLSL) - OpenGL Wiki*. [online] Available at: https://www.khronos.org/opengl/wiki/Layout_Qualifier_(GLSL) [Accessed 3 Aug. 2020].