Tutorial 6

In this tutorial we will learn about lighting and shading in OpenGL. We will do Gouraud and Flat shading with the help of in build lighting functions of OpenGL, and Per-pixel shading using OpenGL Shading Language (GLSL). We will be using Blinn-Phong Illumination Model for lighting any pixel in the scene. Note that shading and illumination are two different things, we can use any shading approach (Gouraud, Per-Pixel, etc.) with any illumination model (Phong, Blinn-Phong, etc.). The whole tutorial is divided into multiple files, tutorial_06.cpp, GLShaderUtils.cpp is a helper code for loading shader programs from files, pp_vertex_shader.glsl and pp_fragment_shader.glsl implements the Per-pixel shading.

Simple Lighting

Lighting with OpenGL internal functions is very simple. It provides at least eight lights sources to be defined form (GL_LIGHT0-7). We just need to enable GL_LIGHTING and required light sources (GL_LIGHT0) using glEnable() function. It also provides with two basic shading models with constants GL_FLAT for faceted shading and GL_SMOOTH for Gouraud shading. Shading model can be changed by passing these constants to glShadeModel() function. Once we are done with shading model, now we need to set light properties such as light position, colour, etc. This is done using glLightfv() function, as shown below.

```
GLfloat light_position[]={0.0, 10.0, 10.0, 1.0};
GLfloat light_diffuse[]={0.7, 0.7, 0.5, 1.0};
GLfloat light_specular[]={1.0, 0.0, 0.0, 1.0};

//Enable lighting
glEnable(GL_LIGHTING);

//Define shading model
glShadeModel(GL_FLAT);
```

```
//Define light properties
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
```

Next we need to define material properties, this done using glMaterialfv() function. Note that by default OpenGL doesn't uses the color set by glColor*() function when lighting is enabled. Thus either we need to set the material properties or we need to explicitly notify OpenGL to use colors set by glColor*(). This done by enabling GL_COLOR_MATERIAL using glEnable() function.

```
glEnable(GL_COLOR_MATERIAL);
glColor4fv(cyan);
OR
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, cyan);
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, cyan);
```

Note that these ligthing functions provides only basic lighting and are removed since OpenGL v3.1 core profile. Thus we need to write shaders for complex lighting.

Shading Language

To use shaders with OpenGL we need GLEW (OpenGL Extension Wrangler Library). GLEW is an extension to older OpenGL. First thing to do is to write a shader file. Shaders are written in GLSL language which is similar to C language.

```
#version 130

uniform vec4 light_color;
uniform vec4 lightPos;
varying vec4 color;
```

First line of shader is the version number. Then we declare our variables and struct declaration. Each variable has one of the either qualifier, uniform, varying, in, out, and inout. Where 'uniform' variables can only be edited from c++ program and are read only in shaders and that they are constant over a frame. Variables with 'varying' qualifier are interpolated in fragment shader, to get its value for each pixel. Variables with 'in' qualifier are passed from previous stage and are read only in this shader. Values of 'out' variables are propagated to the next stage. Variables with 'inout' qualifier are input

from previous stage and are also passed to next stage. A variable can be of type int, float, double, or a struct and it can be a vector or matrix also. Here we are using a vector of four floats for light colour and one for light position, which are passed form our program. The variable 'color' is what we will calculate and will be passed to next stage (fragment shader). It represents the colour of the vertex being processed. Then comes the main() function, which implements the main logic of our program and/or invokes other functions. Similar to other programming languages main() is the entry point for execution.

For our purpose we are going to discuss olny two types of shader viz vertex shader and fragment shader, but there are others also. Vertex shader is a program which is called once for each vertex, that is each vertex is processed once by vertex shader. While fragment shader is called for each pixel of a fragment. Thus any vertex based processing required can be implemented in vertex shader and all per pixel based computing can be implemented in fragment shader.

Lets look at the shader code for Gouraud shading :

```
//----Vertex Shader----
void main (void)
{

  // Internal variables
  // gl_Position : location of the vertex on the screen
  // gl_ModelViewProjectionMatrix : stores the product of
  //                    projection * view * model matrices
  // gl_Vertex : location of vertex passed by glVertex***()
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

  //
  // Normals = Orientation * NormalInWCS
  // Normal matrix = transpose(inverse(mat3(ModelView matrix)));
  glNor = normalize(gl_NormalMatrix * gl_Normal);

  // Vertex in view co-ordinate system
  // (for use in fragment shader)
  glVer = gl_ModelViewMatrix * gl_Vertex;

  //Just pass the colour as it is
  glCol = gl_Color;
```

```
}


//----Fragment Shader----

// Variables set from C++ code
uniform vec4 lightPos;
uniform vec4 light_color;

//Variables passed from vertex shader
varying vec4 glCol;
varying vec4 glVer;
varying vec3 glNor;

void main (void)
{
  // Defining Materials
  // (try passing material properties from C++ code)
  vec4 diffuse = vec4(1.0, 1.0, 1.0, 1.0);
  vec4 ambient = vec4(0.05, 0.05, 0.05, 1.0);
  vec4 specular = vec4(0.1, 0.1, 0.1, 1.0);
  float shininess = 0.5;
  vec4 spec = vec4(0.0);

  // Defining Light
  // LightDirection (L) = LightPositionInVCS - VertexPositionInVCS
  vec3 lightDir = normalize(vec3(lightPos - glVer));

  //Calculate Diffuse Component
  vec3 n = glNor;
  float dotProduct = dot(n, lightDir);
  float intensity =  max( dotProduct, 0.0);

  // Compute specular component only if light falls on vertex
  if(intensity > 0.0)
  {
        vec4 eye = glVer;
        vec3 v = vec3(-eye);//View vector (opposite direction of eye)
        vec3 h = normalize(lightDir + v ); //Half vector
        float intSpec = max(dot(h,n), 0.0);
```

```
        spec = specular * pow(intSpec, shininess);
  }

  //Final color
  gl_FragColor = (intensity * diffuse  + spec + ambient) * glCol * ligh

//-------No Lights-----------
//gl_FragColor = glCol;
}
```

Once we are done with writing shaders, now we need to load the shader from files (.glsl) to our C++ code and compile them. This done using glShaderSource() and glCompileShader(), which are used by helper function LoadShader(). LoadShader takes in two arguments viz. path to vertex shader file and path to fragment shader file, and returns program identifier. The syntax for LoadShader() function is as follows:

GLuint LoadShader(string vshader, string fshader);
Parameters:

- vshader - path to vertex shader file.
- fshader - path to fragment shader file.

```
void init(){
 ...
  pps_program = LoadShader("pp_vertex_shader.glsl",
                           "pp_fragment_shader.glsl");
  l1c_pps = glGetUniformLocation(pps_program,"light_color");
  l1p_pps = glGetUniformLocation(pps_program,"lightPos");
 ...
}
```

Once the shaders are compiled and loaded, now we can get location identifiers for the shader variables. This identifiers are then used to pass the values from our C++ program. In the above code init() function loads the required shaders using LoadShader() function call and also gets the identifier for shader variables using glGetUniformLocation(). The syntax for glGetUniformLocation() function is as follows:
GLuint glGetUniformLocation(GLuint pg_identifier, const GLchar* var_name);
Parameters:

- pg_identifier - program identifier.
- var_name - variable name as declared in shader.

You might also want to look at glGetAttribLocation() function for loading non uniform variables. Now that we are done loading our shaders, the last step remaining is that, we need to notify OpenGL of which shaders are to be used and what are the values of our variables. This is done using functions glUseProgram() and glUniform4fv(),glUniformMatrix4fv,etc. Function glUseProgram() specifies the program to be used and it takes in program identifier as it's parameter. Functions like glUniform4fv(), glUniformMatrix4fv(),etc are used to pass the values to our variable in shaders.

```
...
glUseProgram(pps_program);
glUniform4fv(l1c_pps,1,l1_col);
glUniform4fv(l1p_pps,1,l1_pos);
...
```

The code snippet above shows an example of using the PerPixel shader and loading values for light colour and light position. You can also pass other values to shader, such as material properties of a vertex or transformation matrices, etc. with a little change in a code.