Tutorial 03

August 17, 2015

1 About

This code is mostly same as the code of 01_triangle.cpp (Tutorial 01). So you might see quite a lot similarities.

2 Running the Code

If you have a driver supporting OpenGL 4.1 then running 02_colorcube will do the trick for you. But if your system supports OpenGL 3.2+. Then in order to run the code the following changes need to be made in the 02_colorcube.cpp. In the line number 136 and 137,

```
 glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
 glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 1);
```

you need to change to

```
 glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
 glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
```

Also in the 02_fshader.glsl and 02_vshader.glsl you need to modify the first line to

```
#version 330
```

Once you make the above changes you can compile and run the 02_colorcube file.

3 Understanding the code

Now, expecting that you have already went thorught the Tutorial 01, I won't go through the most of the basic skeleton code that has already been explained in the Tutorial 01. For example the main() function, it is exactly the same and doesn't need any further explanation.

To start with, we have defined several variables, which I'll come to later.

```
GLuint shaderProgram;
GLuint vbo, vao;
```

Next up, we declare positions of 8 vertices and colors for each of the vertices, this is also pretty similar to the previous tutorial.

```
//6 faces, 2 triangles/face, 3 vertices/triangle
const int num_vertices = 36;

//Eight vertices in homogeneous coordinates
glm::vec4 positions[8] = {
  glm::vec4(-0.5, -0.5, 0.5, 1.0),
  glm::vec4(-0.5, 0.5, 0.5, 1.0),
  glm::vec4(0.5, 0.5, 0.5, 1.0),
  glm::vec4(0.5, -0.5, 0.5, 1.0),
  glm::vec4(-0.5, -0.5, -0.5, 1.0),
  glm::vec4(-0.5, 0.5, -0.5, 1.0),
  glm::vec4(0.5, 0.5, -0.5, 1.0),
  glm::vec4(0.5, -0.5, -0.5, 1.0)
};

//RGBA colors
glm::vec4 colors[8] = {
  glm::vec4(0.0, 0.0, 0.0, 1.0),
  glm::vec4(1.0, 0.0, 0.0, 1.0),
  glm::vec4(1.0, 1.0, 0.0, 1.0),
  glm::vec4(0.0, 1.0, 0.0, 1.0),
  glm::vec4(0.0, 0.0, 1.0, 1.0),
  glm::vec4(1.0, 0.0, 1.0, 1.0),
  glm::vec4(1.0, 1.0, 1.0, 1.0),
  glm::vec4(0.0, 1.0, 1.0, 1.0)
};
```

But here, we declare num_vertices=36, now this has a reason. Since each object is expressed as triangles, we will construct every single face of a cube using two triangles.

The next piece of code does exactly this thing. For every given value of a,b,c,d, it creates a face of a cube, by making two adjoint triangles looking as a square. Since we have specified the positions in a specific way, the calls made during the colorcube() command, create different faces of the cube, you can play around with these values to get a better understanding of this piece of code.

```
void quad(int a, int b, int c, int d)
{
  v_colors[tri_idx] = colors[a]; v_positions[tri_idx] =
     positions[a]; tri_idx++;
  v_colors[tri_idx] = colors[b]; v_positions[tri_idx] =
     positions[b]; tri_idx++;
  v_colors[tri_idx] = colors[c]; v_positions[tri_idx] =
     positions[c]; tri_idx++;
  v_colors[tri_idx] = colors[a]; v_positions[tri_idx] =
     positions[a]; tri_idx++;
  v_colors[tri_idx] = colors[c]; v_positions[tri_idx] =
     positions[c]; tri_idx++;
  v_colors[tri_idx] = colors[d]; v_positions[tri_idx] =
     positions[d]; tri_idx++;
 }

// generate 12 triangles: 36 vertices and 36 colors
void colorcube(void)
{
    quad( 1, 0, 3, 2 );
    quad( 2, 3, 7, 6 );
    quad( 3, 0, 4, 7 );
    quad( 6, 5, 1, 2 );
    quad( 4, 5, 6, 7 );
    quad( 5, 4, 0, 1 );
}
```

4 Shaders

The next piece of code void initBuffersGL(void), is essentially what were the initShadersGL() and initVertexBufferGL() in the Tutorial 01.

```
void initBuffersGL(void)
{
  colorcube();
  //Ask GL for a Vertex Attribute Object (vao)
  glGenVertexArrays (1, &vao);
  //Set it as the current array to be used by binding it
  glBindVertexArray (vao);

  //Ask GL for a Vertex Buffer Object (vbo)
  glGenBuffers (1, &vbo);
  //Set it as the current buffer to be used by binding it
  glBindBuffer (GL_ARRAY_BUFFER, vbo);
  //Copy the points into the current buffer
  glBufferData (GL_ARRAY_BUFFER, sizeof (v_positions) +
     sizeof(v_colors), NULL, GL_STATIC_DRAW);
  glBufferSubData( GL_ARRAY_BUFFER, 0, sizeof(v_positions),
     v_positions );
  glBufferSubData( GL_ARRAY_BUFFER, sizeof(v_positions),
     sizeof(v_colors), v_colors );

  // Load shaders and use the resulting shader program
  std::string vertex_shader_file("02_vshader.glsl");
  std::string fragment_shader_file("02_fshader.glsl");

  std::vector<GLuint> shaderList;
  shaderList.push_back(csX75::LoadShaderGL(GL_VERTEX_SHADER,
     vertex_shader_file));
  shaderList.push_back(csX75::LoadShaderGL(GL_FRAGMENT_SHADER
     , fragment_shader_file));

  shaderProgram = csX75::CreateProgramGL(shaderList);
  glUseProgram( shaderProgram );

  // set up vertex arrays
  GLuint vPosition = glGetAttribLocation( shaderProgram, "
     vPosition" );
  glEnableVertexAttribArray( vPosition );
  glVertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE, 0,
      BUFFER_OFFSET(0) );

  GLuint vColor = glGetAttribLocation( shaderProgram, "vColor
     " );
  glEnableVertexAttribArray( vColor );
  glVertexAttribPointer( vColor, 4, GL_FLOAT, GL_FALSE, 0,
     BUFFER_OFFSET(sizeof(v_positions)) );
}
```

We initially call the colorcube() method to populate the two vec4 arrays v_positions and v_colors.

After generating vao and vbo, we create a buffer of size v_positions+v_colors. (Since our buffer is going to hold the color data as well as the position data for each vertex). In the next two lines,

```
glBufferSubData( GL_ARRAY_BUFFER, 0, sizeof(v_positions),
    v_positions );
glBufferSubData( GL_ARRAY_BUFFER, sizeof(v_positions),
    sizeof(v_colors), v_colors );
```

We fill up the buffer using the two types of sub data, position and color. The syntax and semantics of these functions are already explained in tutorial 01.

After this, we load the shaders and push them back into shaderlist vector, for the reference to the shaders.

Then we specify how position and color data are stored in VBO using the vao, now as you remember, vertex attrib pointer, VAO, describes how vertex attributes are stored in the vertex buffer object, VBO. But, in this case we have two things that are to be stored in VBO, colors and positions. So, first we get the location of the parameter(vPosition and vColor), using glGetAttribLocation(). As, you can see that we have defined the same variables in the shader as well. If you remember, we mentioned in the tutorial 0 that the declaration of vPosition and vColor, in shader code may also be done using *layout(location = 0) in vec4 vp;*, the return value of glGetAttribLocation(), is the same location value from the shader program. Now, we can use glVertexAttribPointer, using the location for color and position pointers.

5 Rotation Matrix

```
rotation_matrix = glm::rotate(glm::mat4(1.0f), xrot, glm::
    vec3(1.0f,0.0f,0.0f));
  rotation_matrix = glm::rotate(rotation_matrix, yrot, glm::
     vec3(0.0f,1.0f,0.0f));
  rotation_matrix = glm::rotate(rotation_matrix, zrot, glm::
     vec3(0.0f,0.0f,1.0f));
  ortho_matrix = glm::ortho(-2.0, 2.0, -2.0, 2.0, -2.0, 2.0);
```

In this piece of code we multiply the rotations about x,y and z axes to the rotation matrix. Initially we start with identity matrix (glm::mat4(1.0f)), then multiply the xrot, yrot and zrot one by one, the third parameter in glm::rotate() is the axis, along which we have to rotate. Also, in order to perform an orthographic projection, create a orthographic matrix with glm::ortho(), this sets up our orthographic projection matrix. The parameters given to it are our window co-ordinates in this order, left, right, bottom, top, z-near, z-far. Where, z-near and z-far are the location of near and far clipping plane. Finally, we multiply the ortho matrix to rotation matrix, doing so applies the orthographic projection to model, when we multiply it to get our final coordinates in the shader.

```
glUniformMatrix4fv(uModelViewMatrix, 1, GL_FALSE, glm::
    value_ptr(modelview_matrix));
```

The above statement modifies the value of uModelViewMatrix. The parameter '1' specifies that we are modifying one matrix and GL_FALSE specifies that there is no need to transpose the matrix and finally we provide with what value we need to modify the matrix.

6 Shaders

6.1 Vertex Shader

```
attribute vec4 vPosition;
attribute vec4 vColor;
varying vec4 color;
```

In the main() function here, instead of directly assigning the vPosition to gl_position, we multiply it with our Modelview Matrix, hence applying the required transformations. We are also providing the color using the attributes vPosition and vColor, which we defined in the initBuffersGL() call in our main code.

6.2 Fragment Shader

The fragment shader is quite similar to the code previous tutorial and you can easily see that we are assigning the color of the fragment, from our input from vertex shader.

6.3 Keyboard Input

In the key_callback() function in gl_framework.cpp we deal with various keyboard inputs. You can see that here we increment/decrement our rotation parameters whenever their respective key is pressed