Tutorial 04

Harshavardhan Kode

September 14, 2015

---

1 About

This tutorial an extension of the Tutorial 03. So you might see quite a lot similarities. The following things are new.

- A Plane is added underneath the cube
- A Camera is added that can be moved around the cube
- The view of the camera can be toggled between Orthographic or Perspective

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

The understanding for the code of basics, VAOs, VBOs and Shaders is documented in the previous tutorials. This tutorial would not go into much detail about those again, but explain all the new stuff.

To draw multiple objects, it is essential that you have a good way of storing the attributes (position, color etc.) of each object and be able to draw it appropriately. One way to do it is to store the attributes (vPosition, vColor here) corresponding each object in a different VAO. While drawing, we simply bind the VAO corresponding to the object, load the appropriate transformation matrix and draw it.

You will see that in this tutorial we have defined two VAOs and VBOs.

```
GLuint vbo[2], vao[2];
```

It is followed by the creation of the cube vertex and colour arrays which was discussed in the previous tutorial.

Apart from that you will see this extra piece of code.

```
glm::vec4 v_positions2[6] = {
  glm::vec4(1.0, -1.0, 1.0, 1.0),
  glm::vec4(1.0, -1.0, -1.0, 1.0),
  glm::vec4(-1.0, -1.0, -1.0, 1.0),
  glm::vec4(1.0, -1.0, 1.0, 1.0),
  glm::vec4(-1.0, -1.0, -1.0, 1.0),
  glm::vec4(-1.0, -1.0, 1.0, 1.0)
};

glm::vec4 v_colors2[6] = {
  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)
};
```

This is basically the positions of vertices and the colours to form a plane below the cube (the plane is made using two triangle primitives, as are all the faces of the cube, and hence 6 vertices)

Next, in the function initBuffersGL, we actually initialize the Shader program and the VAOs, VBOs corresponding to the two objects. I will split the function into three major parts to explain it. First part is the loading of the shaders and obtaining the attribute indices.

```
  // Load shaders and use the resulting shader program
  std::string vertex_shader_file("04_vshader.glsl");
  std::string fragment_shader_file("04_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 );

  // getting the attributes from the shader program
  GLuint vPosition = glGetAttribLocation( shaderProgram, "
     vPosition" );
  GLuint vColor = glGetAttribLocation( shaderProgram, "vColor
     " );
  uModelViewMatrix = glGetUniformLocation( shaderProgram, "
     uModelViewMatrix");
```

This is same as in the previous tutorials. Initially the shader is loaded and the attribute indices are obtained. It is done in the beginning because the plan is to use the same shader to draw all the objects and so we require the index of the attributes in the shader program. The description of each of the steps can be seen in the previous tutorials

Next we allocate the buffers

```
  //Ask GL for two Vertex Attribute Objects (vao) , one for
     the colorcube and one for the plane.
  glGenVertexArrays (2, vao);
```

```
//Ask GL for two Vertex Buffer Object (vbo)
glGenBuffers (2, vbo);
```

Two VBOs and VAOs are allocated, one each for the cube and the plane.

Next we fill up the buffer for the cube and enable the appropriate attributes and assign them appropriate locations in the VBOs.

```
//Set 0 as the current array to be used by binding it
glBindVertexArray (vao[0]);
//Set 0 as the current buffer to be used by binding it
glBindBuffer (GL_ARRAY_BUFFER, vbo[0]);

colorcube();
//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 );

// set up vertex array

glEnableVertexAttribArray( vPosition );
glVertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE, 0,
    BUFFER_OFFSET(0) );

glEnableVertexAttribArray( vColor );
glVertexAttribPointer( vColor, 4, GL_FLOAT, GL_FALSE, 0,
   BUFFER_OFFSET(sizeof(v_positions)) );
```

See that all the attribute locations are set when vao[0] is bound. so all the changes happen in that VAO. This is same as in the previous tutorial.

Next we do the same for the plane, but use the other VAO and VBO.

```
//Set 1 as the current array to be used by binding it
glBindVertexArray (vao[1]);
//Set 1 as the current buffer to be used by binding it
glBindBuffer (GL_ARRAY_BUFFER, vbo[1]);

//Copy the points into the current buffer
glBufferData (GL_ARRAY_BUFFER, sizeof (v_positions2) +
   sizeof(v_colors2), NULL, GL_STATIC_DRAW);
```

```
glBufferSubData( GL_ARRAY_BUFFER, 0, sizeof(v_positions2),
   v_positions2 );
glBufferSubData( GL_ARRAY_BUFFER, sizeof(v_positions2),
   sizeof(v_colors2), v_colors2 );

// set up vertex array

glEnableVertexAttribArray( vPosition );
glVertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE, 0,
    BUFFER_OFFSET(0) );

glEnableVertexAttribArray( vColor );
glVertexAttribPointer( vColor, 4, GL_FLOAT, GL_FALSE, 0,
   BUFFER_OFFSET(sizeof(v_positions2)) );
```

With this, our buffers for each of the object are ready! To draw, we simply enable the corresponding VAO and call glDrawArrays() with the appropriate number of vertices.

4 Camera/Eye

In this tutorial, as mentioned at the beginning, apart from rotating the cube, we also add functionality to move the camera and toggle the view between ortho or perspective. I would be using the word 'camera' in this tutorial for the eye.

In the header file, 04_camera_viewing.hpp, you will see the following lines have been added

```
// Camera position and rotation Parameters
GLfloat c_xpos = 0.0, c_ypos = 0.0, c_zpos = 2.0;
GLfloat c_up_x = 0.0, c_up_y = 1.0, c_up_z = 0.0;
GLfloat c_xrot=0.0,c_yrot=0.0,c_zrot=0.0;
```

These represent the parameters of the camera location and the up vector. c_pos represent the initial position of the camera. c_up_ represent the upvector. c_rot represent the rotation of the camera's initial position vector about the world origin.

Now, coming back to the cpp file, the function renderGL() is actually where the drawing happens.

```
rotation_matrix = glm::rotate(glm::mat4(1.0f), glm::radians
    (xrot), glm::vec3(1.0f,0.0f,0.0f));
rotation_matrix = glm::rotate(rotation_matrix, glm::radians
    (yrot), glm::vec3(0.0f,1.0f,0.0f));
rotation_matrix = glm::rotate(rotation_matrix, glm::radians
    (zrot), glm::vec3(0.0f,0.0f,1.0f));
model_matrix = rotation_matrix;
```

These are the modelling transforms applied to the cube and are same as in the last tutorial, except you will notice the use of glm::radians() function. The glm library from version 0.9.7 treats all parameter angles as radians. So if you want to use degrees, you have to convert it to radians using that function

Next is creating the lookat matrix the transforms the vertices from the WCS to the VCS

```
c_rotation_matrix = glm::rotate(glm::mat4(1.0f), glm::
    radians(c_xrot), glm::vec3(1.0f,0.0f,0.0f));
c_rotation_matrix = glm::rotate(c_rotation_matrix, glm::
    radians(c_yrot), glm::vec3(0.0f,1.0f,0.0f));
c_rotation_matrix = glm::rotate(c_rotation_matrix, glm::
    radians(c_zrot), glm::vec3(0.0f,0.0f,1.0f));

glm::vec4 c_pos = glm::vec4(c_xpos,c_ypos,c_zpos, 1.0)*
    c_rotation_matrix;
glm::vec4 c_up = glm::vec4(c_up_x,c_up_y,c_up_z, 1.0)*
    c_rotation_matrix;
//Creating the lookat matrix
lookat_matrix = glm::lookAt(glm::vec3(c_pos),glm::vec3(0.0)
    ,glm::vec3(c_up));
```

In this piece of code we multiply the rotations about x,y and z axes to the camera rotation matrix and then multiply this matrix to the initial position of the camera to get its final location. This matrix is also multiplied to the initial up vector to rotate it appropriately. Note that the camera rotation matrix is just used to calculate the position of the camera, and not a part of the modelling-viewing matrix. Finally, the lookat matrix is created by using the function glm::lookAt, which takes as parameters the position of the camera, the look-at vector and the up vector and outputs the appropriate transformation from the WCS to the VCS.

Next, we create the projection matrices based on the current mode

```
//creating the projection matrix
if(enable_perspective)
  projection_matrix = glm::frustum(-1.0, 1.0, -1.0, 1.0,
     1.0, 5.0);
  //projection_matrix = glm::perspective(glm::radians(90.0)
     ,1.0,0.1,5.0);
else
  projection_matrix = glm::ortho(-2.0, 2.0, -2.0, 2.0,
     -5.0, 5.0);

view_matrix = projection_matrix*lookat_matrix;
```

glm::ortho returns the orthographic projection matrix and was discussed in the last tutorial. glm::frustum converts the frustum specified by the function to a unit cube as discussed in class, and can be directly used to obtain the perspective projection. The arguments to the function are the left, right, top and bottom at the near plane, the distance of the near plane and the far plane respectively. Alternatively, glm::perspective function can also be used. It does the same thing as glm::frustum but has different type of arguments viz. the field of view in the y direction, the y:x aspect ratio, the near plane and the far plane respectively. To enable or disable the perspective view, a boolean is modified based on a key press event, written in the key callback function and the appropriate matrix is chosen.

With this we get the projection matrix, we multiply it the the lookat matrix to get the whole viewing transformation.

Finally, we draw the objects with the appropriate transfromation matrices

```
// Draw cube
modelview_matrix = view_matrix*model_matrix;
glUniformMatrix4fv(uModelViewMatrix, 1, GL_FALSE, glm::
   value_ptr(modelview_matrix));
glBindVertexArray (vao[0]);
glDrawArrays(GL_TRIANGLES, 0, num_vertices);

// Draw plane
modelview_matrix = view_matrix;
glUniformMatrix4fv(uModelViewMatrix, 1, GL_FALSE, glm::
   value_ptr(modelview_matrix));
glBindVertexArray (vao[1]);
glDrawArrays(GL_TRIANGLES, 0, 6);
```

For drawing each of the objects, the approptiate VAO is being bound and

glDrawArrays is called with the appropriate number of vertices in the object. For the cube, the entire modelling transformations are to be applied, while the plane has to remain static. Hence, before calling glDrawArrays for the plane, the uniform value that is passed to the shader is changed to just the viewing matrix.

5 Shaders

5.1 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.

5.2 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.

5.3 Keyboard Input

In the key_callback() function in gl_framework.cpp we deal with various keyboard inputs. Various additional inputs have been added to handle the movement of the camera and to toggle between the perspective and orthographic views.

```
    else if (key == GLFW_KEY_P && action == GLFW_PRESS)
      enable_perspective = !enable_perspective;
    else if (key == GLFW_KEY_A  && action == GLFW_PRESS)
      c_yrot -= 1.0;
    else if (key == GLFW_KEY_D  && action == GLFW_PRESS)
      c_yrot += 1.0;
    else if (key == GLFW_KEY_W  && action == GLFW_PRESS)
```

```
    c_xrot -= 1.0;
  else if (key == GLFW_KEY_S  && action == GLFW_PRESS)
    c_xrot += 1.0;
  else if (key == GLFW_KEY_Q  && action == GLFW_PRESS)
    c_zrot -= 1.0;
  else if (key == GLFW_KEY_E  && action == GLFW_PRESS)
    c_zrot += 1.0;
```