SCIT

School of Computing & Information Technology

CSCI336 – Interactive Computer Graphics

Viewing and Projection

In this lab, we will look at viewing and projection transformations in OpenGL using the GLM library. With viewing and projection transformations added to the rendering pipeline, we can now deal with 3D scenes in OpenGL.

Colour Cube

In the Lab5a project, a colour cube is created using the index buffer approach. A cube consists of 8 vertices as shown in Figure 1. Each vertex will have a distinct colour which will be interpolated across the triangles. The cube has 6 faces, each face consisting of two triangles, giving a total of 12 triangles.

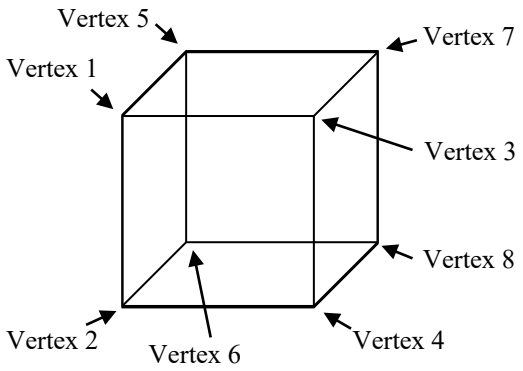

Figure 1

The following code in lab5a.cpp defines vertices of the colour cube and indices of the 12 triangles:

```
std::vector<GLfloat> vertices = {
       // colour cube
       -0.5f, 0.5f, 0.5f,   // vertex 0: position
       1.0f, 0.0f, 1.0f,    // vertex 0: colour
       ...
       ...
       0.5f, -0.5f, -0.5f,  // vertex 7: position
       0.0f, 1.0f, 0.0f,    // vertex 7: colour
};

std::vector<GLuint> indices = {
       0, 1, 2,      // triangle 1
       2, 1, 3,      // triangle 2
       ...
       7, 4, 6,      // triangle 12
};
```

The indices are buffered in an IBO and bound to a VAO. This should be familiar from the previous lab.

Since this uses an IBO, we use **glDrawElements()** to render the object. There are 6 faces which consist of 12 triangles, each triangle has 3 vertices. Hence, the number of elements is 12 x 3 = 36:

```
glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, 0);
```

Viewing and Projection Transformations

View and projection transformations are performed in the vertex shader. The vertex shader is modified slightly to have a uniform variable which stores the multiplication of the model, view and projection matrices. Note the following in the modelViewProj.vert shader:

```
uniform mat4 uModelViewProjectionMatrix;
```

The position of each vertex is transformed by this matrix:

```
gl_Position = uModelViewProjectionMatrix * vec4(aPosition, 1.0);
```

To store the viewing and projection matrices, the following are declared in lab5a.cpp:

```
glm::mat4 gViewMatrix;            // view matrix
glm::mat4 gProjectionMatrix;      // projection matrix
```

These matrices are initialised in init(), using GLM functions:

```
gViewMatrix = glm::lookAt(glm::vec3(0.0f, 0.0f, 5.0f),
       glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f, 1.0f, 0.0f));
```

This uses the **glm::lookAt()** function to create a view matrix. The first vector sets the position of the camera to be 5 units away from the origin in the +ve z-direction. The second vector sets the camera look at the world space origin. This means that the direction of the camera is in the -ve z-direction. The third vector sets the camera's up vector.

To initialise the projection matrix, we first compute the aspect ratio.

```
float aspectRatio = static_cast<float>(gWindowWidth) / gWindowHeight;
```

With this, we can use the **glm::perspective()** function to create a perspective projection matrix:

```
gProjectionMatrix = glm::perspective(glm::radians(45.0f), aspectRatio, 0.1f, 10.0f);
```

The first parameter is the field of view; the second parameter is the aspect ratio; the third and fourth parameters are the near and far clipping planes, respectively.

To render the scene, concatenate the model, view and projection matrices, then pass this to the shader:

```
glm::mat4 MVP = gProjectionMatrix * gViewMatrix * gModelMatrix;

gShader.setUniform("uModelViewProjectionMatrix", MVP);
glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, 0);
```

Compile and run the program.

3D Transformation (Add the code in this section yourself)

First, define translation and rotation sensitivities to be able to adjust the transformation speed:

```
const float gTranslateSensitivity = 1.0f;
const float gRotateSensitivity = 1.0f;
```

To move the cube around in 3D, add the following code to the update_scene() function.

Declare and initialise the following static variables to store the object's translation and rotation:

```
static glm::vec3 translateVec(0.0f);
static glm::vec2 rotateAngle(0.0f);
```

To update the translation vector based on key presses, add the following:

```
if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
       translateVec.x -= gTranslateSensitivity * gFrameTime;
if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
       translateVec.x += gTranslateSensitivity * gFrameTime;
if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
       translateVec.y += gTranslateSensitivity * gFrameTime;
if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
       translateVec.y -= gTranslateSensitivity * gFrameTime;
if (glfwGetKey(window, GLFW_KEY_E) == GLFW_PRESS)
       translateVec.z -= gTranslateSensitivity * gFrameTime;
if (glfwGetKey(window, GLFW_KEY_C) == GLFW_PRESS)
       translateVec.z += gTranslateSensitivity * gFrameTime;
```

Add the following to update the rotation angles about the *x* and *y* axes based on key presses:

```
if (glfwGetKey(window, GLFW_KEY_J) == GLFW_PRESS)
       rotateAngle.y -= gRotateSensitivity * gFrameTime;
if (glfwGetKey(window, GLFW_KEY_L) == GLFW_PRESS)
       rotateAngle.y += gRotateSensitivity * gFrameTime;
if (glfwGetKey(window, GLFW_KEY_I) == GLFW_PRESS)
       rotateAngle.x -= gRotateSensitivity * gFrameTime;
if (glfwGetKey(window, GLFW_KEY_K) == GLFW_PRESS)
       rotateAngle.x += gRotateSensitivity * gFrameTime;
```

Finally, create the model's matrix based on the translation vector and rotation angles:

```
gModelMatrix = glm::translate(translateVec)
       * glm::rotate(rotateAngle.x, glm::vec3(1.0f, 0.0f, 0.0f))
       * glm::rotate(rotateAngle.y, glm::vec3(0.0f, 1.0f, 0.0f));
```

Compile and run the program.

Try moving and rotating the object in 3D. Note that you can move the object closer to and further away from the camera's position.

Depth Buffer Test (Add the code in this section yourself)

Notice that the object does not look right. This is more obvious when you move and rotate the colour cube.

The reason for this is because visibility determination is not performed. This means that polygons rendered later will simply overwrite anything that was rendered previously, without checking what should be in front and what should be behind. To overcome this, we need to enable the depth buffer test. Put the following in the init() function:

```
glEnable(GL_DEPTH_TEST);    // enable depth buffer test
```

In the render_scene() function, the depth buffer bits must be cleared along with the colour buffer bit:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Compile and run the program. The colour cube should now look correct.

Viewpoint and Direction (Modify the code based on the following instructions)

Modify the code by changing the viewpoint (which will affect the viewing direction), when creating the view matrix:

```
gViewMatrix = glm::lookAt(glm::vec3(-5.0f, 5.0f, 5.0f),
       glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f, 1.0f, 0.0f));
```

The viewpoint is now at coordinate (-5.0f, 5.0f, 5.0f), looking at the world space origin, i.e. (0.0f, 0.0f, 0.0f).

Compile and run the program.

Keep in mind where the camera is in world space, then try moving and rotating the object.

Perspective Projection's Field of View (Modify the code based on the following instructions)

In perspective projection, changing the field of view changes the camera's "zoom".

Try changing the projection matrix to the following:

```
gProjectionMatrix = glm::perspective(glm::radians(30.0f), aspectRatio, 0.1f, 10.0f);
```

Compile and run the program.

Now change the projection matrix to the following:

```
gProjectionMatrix = glm::perspective(glm::radians(60.0f), aspectRatio, 0.1f, 10.0f);
```

Compile and run the program.

In summary, decreasing the field of view results in the camera "zooming in", while increasing the field of view results in the camera "zooming out".

Orthographic Projection (Modify the code based on the following instructions)

Set the viewpoint back to:

```
gViewMatrix = glm::lookAt(glm::vec3(0.0f, 0.0f, 5.0f),
       glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f, 1.0f, 0.0f));
```

The code current sets the projection matrix to use perspective projection. Change the projection matrix to orthographic projection. Since the viewport is not a square, scale the left and right arguments by the aspect ratio:

```
gProjectionMatrix = glm::ortho(-2.0f * aspectRatio, 2.0f * aspectRatio, -2.0f, 2.0f, 0.1f,
10.0f);
```

Compile and run the program.

Try moving and rotating the object. Note that in orthographic projection, lines do not change with distance from the viewpoint. Hence, moving the object closer of further way from the viewport will not change the object's size.

We scaled the left and right arguments because the viewport was not a square. Change the code to the following to see what happens if we do not perform scaling:

```
gProjectionMatrix = glm::ortho(-2.0f, 2.0f, -2.0f, 2.0f, 0.1f, 10.0f);
```

To change the "zoom" in orthographic projection change the left, right, bottom and top arguments, e.g.:

```
gProjectionMatrix = glm::ortho(-1.5f * aspectRatio, 1.5f * aspectRatio, -1.5f, 1.5f, 0.1f,
10.0f);
```

Now set the viewpoint to:

```
gViewMatrix = glm::lookAt(glm::vec3(-5.0f, 5.0f, 5.0f),
       glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f, 1.0f, 0.0f));
```

Repeat the changes to the orthographic projection matrix to observe the effect from this camera viewpoint and direction.

Camera Class

For this section, open the project in Lab5b. This project contains two files: Camera.h and Camera.cpp, which define a basic Camera class. The class contains member functions to update the camera's position and orientation, set the view and projection matrices, and to get the value of several camera member variables.

The program shows you how to use the Camera class to create a free-roaming camera. Important parts of the code are described here.

First, the movement and rotation sensitivities are defined, followed by a declaration of a Camera object.

```
const float gCamMoveSensitivity = 1.0f;
const float gCamRotateSensitivity = 0.1f;
...
Camera gCamera;
```

The camera's view and projection matrices are initialised as follows:

```
gCamera.setViewMatrix(glm::vec3(0.0f, 2.0f, 4.0f),
	glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f, 1.0f, 0.0f));

gCamera.setProjMatrix(glm::perspective(glm::radians(45.0f),
	static_cast<float>(gWindowWidth)/gWindowHeight, 0.1f, 10.0f));
```

Keyboard input to move the camera is handled in the update_scene() function. In this function, you can see that keyboard input is used to update variables that store forward/backward and left/right movements. These are then passed to the camera's update() member function, which updates the position and direction of the camera based on the yaw and pitch values.

Camera rotation based on mouse movement is handled in the cursor_position_callback() function. To avoid conflict between using the mouse for other interaction and for rotating the camera, camera rotation is only enabled when the right mouse button is held. The difference between the current cursor position and the previous cursor position is calculated, then passed to the camera's updateRotation() member function.

Before rendering objects in the scene, we need to pass the model-view-projection matrix to the shader. To do this, we retrieve the view and projection matrices from the camera:

```
glm::mat4 MVP = gCamera.getProjMatrix() * gCamera.getViewMatrix() * gModelMatrix;
gShader.setUniform("uModelViewProjectionMatrix", MVP);
```

Compile and run the program.

Use the WASD keys to move the camera. For camera rotation, hold the right mouse button then move the mouse.

Additional Features (Add the code in this section yourself)

When rotating the camera, the cursor is visible and difficult to use. To solve this, add the following to the mouse_button_callback() function to hide the cursor when the right mouse button is pressed, and to display the cursor (at the position where it was previously hidden) when the right mouse button is released:

```
if (button == GLFW_MOUSE_BUTTON_RIGHT && action == GLFW_PRESS)
{
	// hide cursor
	glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
}
else if (button == GLFW_MOUSE_BUTTON_RIGHT && action == GLFW_RELEASE)
{
	// display cursor
	glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_NORMAL);
}
```

Add the following to the key_callback() function to create two view presets:

```
if (key == GLFW_KEY_1 && action == GLFW_PRESS)
{
	gCamera.setViewMatrix(glm::vec3(0.0f, 2.0f, 4.0f),
		glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f, 1.0f, 0.0f));
	return;
}
if (key == GLFW_KEY_2 && action == GLFW_PRESS)
{
	// top-down view
	gCamera.setViewMatrix(glm::vec3(0.0f, 5.0f, 0.01f),
		glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f, 0.0f, -1.0f));
	return;
}
```

Pressing '1' resets the camera to the starting view, while pressing '2' creates a top-down view. Note that for a top-down view, the z value of the viewpoint is set to a small value, i.e. 0.01f. Do **NOT** make this value 0.0f.

Exercises

By now, you should understand viewing and projection transformations. Try modifying the program to do the following:

- Reset the object's position and orientation to the original starting position and orientation when the user presses 'r' (Lab5a)
- Zoom in and out using keyboard input for perspective and orthographic projections, respectively
- Use number keys to preset several camera viewpoints and directions, and projections

References

Among others, much of the material in this lab was sourced from:

- https://www.khronos.org/opengl/wiki/
- https://learnopengl.com/