3D Graphics Programming

Lab 5: Matrix

Karsten Pedersen
Department of Creative Technology

October 16, 2018

In this lab we will be looking into making your project look as though it is moving. In OpenGL, we do not modify the data stored in the VBOs because doing so would be very expensive each frame. Instead we keep the data as it is and instead perform calculations on the positions as we draw the shape. We do this in the Vertex shader.
The first step is to modify the shader and add a new uniform in the same way we did when changing the color. The only difference is that instead of a color (vec4), we want to pass in a mat4 which can contain the transform. This will be the model matrix. We also want to pass in another matrix called the Projection matrix so that we can add perspective into our project. This will help allow us to see the triangle move unlike if we remain in normalised device coordinates.

```
uniform mat4 in_Projection;
uniform mat4 in_Model;
```

We now multiply the *in_Position* by the new model matrix and projection matrix. This transforms the vertex coordinates into world space and then into screen space respectively.

```
attribute vec3 in_Position;
attribute vec4 in_Color;

varying vec4 ex_Color;

void main()
{
  gl_Position = in_Projection * in_Model * vec4(in_Position, 1.0);
  ex_Color = in_Color;
}
```

Now in the C++ we need to generate our perspective projection matrix and assign it to the *in_Projection* variable. Likewise we also need to create our model matrix and assign it to *in_Model*. Finally we draw the shape.

```
float angle = 0;

// Inside the main loop

// Draw with perspective projection matrix
shader->setUniform("in_Projection", glm::perspective(glm::radians(45.0f),
 (float)WINDOW_WIDTH / (float)WINDOW_HEIGHT, 0.1f, 100.f));

glm::mat4 model(1.0f);
model = glm::translate(model, glm::vec3(0, 0, -2.5f));
model = glm::rotate(model, glm::radians(angle), glm::vec3(0, 1, 0));

shader->setUniform("in_Model", model);
shader->draw(shape);

angle += 0.1f;
```

Note:
In this lab we are using the wrapper code to cut down on the amount of code we need to write to perform this task. The main thing to note that the setUniform function uses *glGetUniformLocation* to obtain the uniform location and *glUniformMatrix4fv* to assign the matrix. The exact function call is glUniformMatrix4fv(uniformId, 1, GL_FALSE, glm::value_ptr(value)); *uniformId* is the *GLint* containing the uniform variable location and *value* is the matrix to assign. The *glm::value_ptr* is to extract the raw matrix out of the glm wrapper for use with OpenGL.

With that working, you should be able to increase *angle* per frame and see the triangle spin. Now try to also draw the same triangle but with an orthographic projection matrix. The following code demonstrates this:

```
// Draw with orthographic projection matrix
model = glm::mat4(1.0f);

shader->setUniform("in_Projection", glm::ortho(0.0f,
  (float)WINDOW_WIDTH, 0.0f, (float)WINDOW_HEIGHT, 0.0f, 1.0f));

model = glm::translate(model, glm::vec3(100, WINDOW_HEIGHT - 100, 0));
model = glm::scale(model, glm::vec3(100, 100, 1));

shader->setUniform("in_Model", model);
shader->draw(shape);
```

By the end of this lab, your project should look similar to that in **Figure 1**.

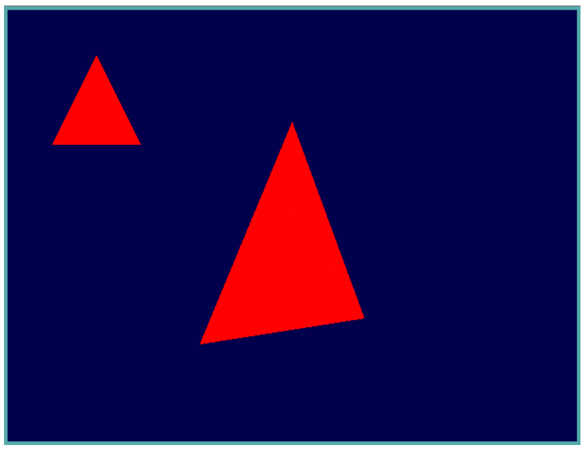

Figure 1: *The final output of the rotating triangle*