Lab 02 Transformations

Create Context

In *createContext* we have constructed two objects, a triangle and a cube. The two objects have two attributes, the vertex coordinates (0) and the vertex color (1). A separate VAO has been created so that the two objects can be referenced easily during the rendering process.

GLM

A matrix (e.g. 4x4) is stored in memory sequentially. OpenGL is column major while C++ is row major, meaning the layout of a matrix in memory is interpreted differently.

```
GLfloat matrix[] = {
    1.0f, 0.0f, 0.0f, 0.0f, // first column
    0.0f, 1.0f, 0.0f, 0.0f, // second column
    0.0f, 0.0f, 1.0f, 0.0f, // third column
    0.5f, 0.0f, 0.0f, 1.0f  // fourth column
};
```

For example, for the above matrix the translation must be defined in the fourth row instead of the fourth column (red). Regardless of the underlying convention the homogeneous mathematical operation should be invariant under any representation. We will be using GLM library for vector matrix manipulation.

```
// Include GLM
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtx/string_cast.hpp>   // glm::to_string
using namespace glm;
```

Primitives:

```
vec3; vec4; mat3; mat4;
```

```
// identity matrix
mat4 I = mat4(1.0f);
```

Mathematical notation +, -, * is recognized.

```
// matrix vector multiplication
vec4 vec(0, 1, 4, 1);
mat4 T;
vec4 vec2 = T * vec;
```

Scaling (x2) matrix

```
mat4 triangleScaling = glm::scale(mat4(), vec3(2.0, 2.0, 2.0));
cout << glm::to_string(triangleScaling) << endl;


mat4x4(
```

```
        (2.000000, 0.000000, 0.000000, 0.000000),
        (0.000000, 2.000000, 0.000000, 0.000000),
        (0.000000, 0.000000, 2.000000, 0.000000),
        (0.000000, 0.000000, 0.000000, 1.000000))
```

Rotation of 45^o around z axis

```
mat4 triangleRotation = glm::rotate(mat4(), 3.14f / 4.0f, vec3(0.0f, 0.0f, 1.0f));
cout << glm::to_string(triangleRotation) << endl;
mat4x4(
        (0.707388, 0.706825, 0.000000, 0.000000),
        (-0.706825, 0.707388, 0.000000, 0.000000),
        (0.000000, 0.000000, 1.000000, 0.000000),
        (0.000000, 0.000000, 0.000000, 1.000000))
```

Translation $(0, 2, 1)$ matrix

```
mat4x4(
        (1.000000, 0.000000, 0.000000, 0.000000),
        (0.000000, 1.000000, 0.000000, 0.000000),
        (0.000000, 0.000000, 1.000000, 0.000000),
        (0.000000, 2.000000, 1.000000, 1.000000))
```

Note that the internal representation of GLM follows the column major order. The transpose of a transformation matrix is the row major order.

GLM provides some useful functions that evaluate the projection transformation.

```
// Projection matrix: 45° Field of View, 4:3 ratio, display range : 0.1 unit <-> 100 units
mat4 projection = perspective(radians(45.0f), 4.0f / 3.0f, 0.1f, 100.0f);
// Or, for an ortho camera:
mat4 Projection = ortho(-10.0f,10.0f, 10.0f,10.0f,0.0f,100.0f); // In world coordinates
```

Or the camera (view) matrix.

```
// Camera matrix
mat4 view = lookAt(
    vec3(5, 3, 10), // Camera is at (5, 3, 10), in World Space
    vec3(0, 0, 0), // and looks at the origin
    vec3(0, 1, 0)  // Head is up (set to 0,-1,0 to look upside-down)
);
```

Shader

The two shader programs are shown below:

```
#version 330 core

// input vertex and color data, different for all executions of this shader
layout(location = 0) in vec3 vertexPosition_modelspace;
layout(location = 1) in vec3 vertexColor;

// model view projection matrix
uniform mat4 MVP;
```

```
// output data
out vec3 color;

void main()
{
    // assign vertex position
    gl_Position = MVP * vec4(vertexPosition_modelspace, 1.0);

    // propagate color to fragment shader
    color = vertexColor;
}
```

```
#version 330 core

// input data
in vec3 color;

// output data
out vec3 fragmentColor;

void main()
{
    fragmentColor = color;
}
```

Note the use of a **uniform** variable**.** A uniform is a variable that is broadcasted from the main program to the GPU and is available to all shaders. In this case, the uniform variable represents the coordinate transformation matrix MVP (model-view-projection), which transforms each vertex coordinate of the objects.

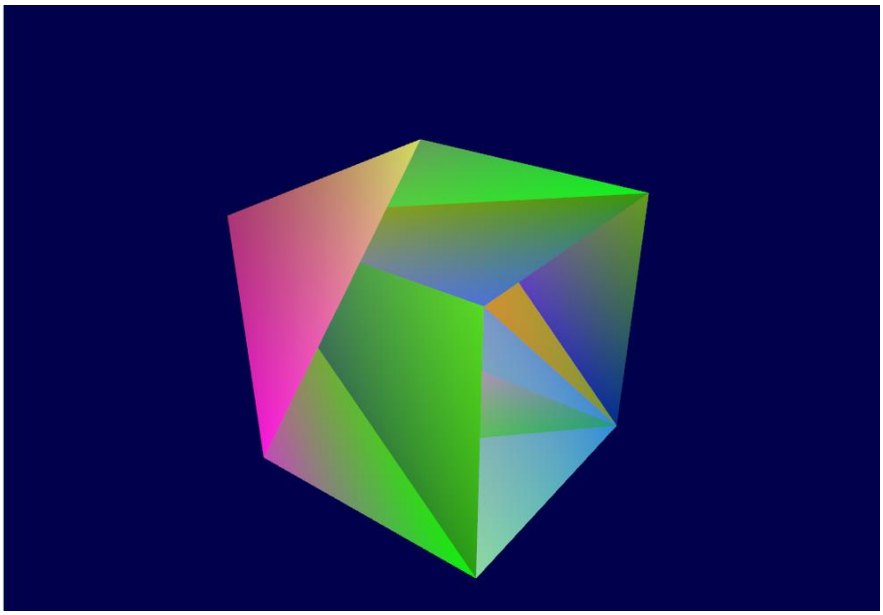
Accessing the uniform in the main program is initiated by a reference pointer.

```
GLuint MVPLocation;
//…
// Get a pointer location to model matrix in the vertex shader
MVPLocation = glGetUniformLocation(shaderProgram, "MVP");
```

The MVP matrix is transmitted to the shader program by the following command

```
mat4 triangleMVP = projection * view * triangleModel;
// Transmit the mvp matrix to shader
// (&triangleMVP[0][0] -> first component of the sequential array)
glUniformMatrix4fv(MVPLocation, 1, GL_FALSE, &triangleMVP[0][0]);
```

Z-Buffer



Sometimes we can observe the above result. This is caused when a triangle that is farther away from the camera is drawn after a triangle that is closer. To solve these issues, we use the Z-Buffer. This buffer that sorts the fragments according to their depth so that the GPU is able to draws them in the correct order.

```
// Enable depth test
glEnable(GL_DEPTH_TEST);
// Accept fragment if it closer to the camera than the former one
glDepthFunc(GL_LESS);
```

```
// Clear the screen (color and depth)
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Tasks

1) Rotate and translate the cube and/or the triangle as you wish in order to get used to the different transformations. What happens when you change the order the translation (translation, rotation and scaling)?
2) Stretch the cube by a factor of 2 along the x-axis.
3) Construct a roof on top of the cube.
4) Make the cube rotate periodically around an arbitrary axis with a speed of 3 cycles/second.
5) Make the cube contract and expand periodically.
6) Make the camera move around the house periodically.

Reading

Anton's OpenGL 4 Tutorials - Anton Gerdelan pp. 85-110