Computer Graphics 4052
Lab 1B

Binh-Son Hua
Trinity College Dublin

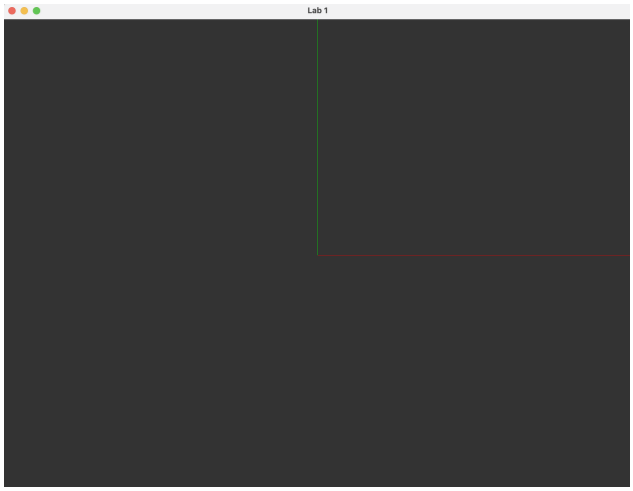
In this lab, we are going to create a simple OpenGL application to create and render a cube at the world origin. Our objectives are:

- To recap the basic concepts of OpenGL
- To visualize the 3D coordinate system for debugging and analysis purposes
- To perform rendering of a simple 3D cube

0. Background

First of all, please read carefully the provided source code in lab1_cube.cpp. The code structure follows what we have done in Lab 1A. Some shader loading code has been abstracted and stored in render/shader.h and render/shader.cpp.

Compile lab1_cube.cpp by using CMake as in Lab 1A. Run lab1_cube, if you see an OpenGL window with two axes, red and green.



In lab1_cube.cpp, there is a coordinate system with three axes (X-, Y-, and Z-axis) implemented. This is the global (world) coordinate system of our 3D scene, which we here visualize for our understanding and debugging purposes. The X-, Y-, and Z-axis are drawn at the origin, and colored with red, green, and blue, respectively.

However, the camera view and the projection transform have not been set. You will therefore have the geometry defined in the default normalized coordinates projected by an identity projection matrix, and hence the 2D appearance of the coordinate system (just X- and Y-axis shown).

To see in 3D, let us proceed with positioning our scene in a camera view and adding a perspective projection.

1. View transform

Let us first define the camera view by specifying its location (eye center), lookat position (where the camera will focus on), and the up vector. This is done at the beginning of lab1_cube.cpp.

```
// OpenGL camera view parameters
static glm::vec3 eye_center(300.0f, 300.0f, 300.0f);
static glm::vec3 lookat(0, 0, 0);
static glm::vec3 up(0, 1, 0);
```

Using these parameters, the camera space transformation matrix (a 4x4 matrix) can be constructed by

```
glm::mat4 viewMatrix = glm::lookAt(eye_center, lookat, up);
```

Applying this matrix on a 3D point in the world space will transform the point into the camera space.

Implementation. Look for glm::mat4 viewMatrix in the main loop, and you will see that currently it is simply not set. Replace it with glm::lookAt function call as above.

2. Perspective projection

Let us now define the projection model of our camera view. Here we use perspective projection and a symmetric view frustum.

Implementation. The projection matrix can be constructed by calling glm::perspective(). Below is an example code to construct a perspective projection matrix.

```
glm::float32 FoV = 45;
glm::float32 zNear = 0.1f;
glm::float32 zFar = 1000.0f;
glm::mat4 projectionMatrix = glm::perspective(glm::radians(FoV), 4.0f / 3.0f, zNear, zFar);
```

where the parameters of the perspective projection includes the field of view (FoV) along the y-axis, near plane and far plane defined as the distance from the camera center, and the aspect ratio of the image plane (4/3 in our case because our image size is 1024x768).

For compactness, we pack both camera space transformation matrix and the projection matrix into a single 4x4 matrix before passing it to the rendering function.

```
glm::mat4 vp = projectionMatrix * viewMatrix;
```

In the render function of the AxisXYZ struct, we can see that the vp matrix (the camera matrix) is used as follows.

```
// Set model-view-projection matrix
```

```
glm::mat4 mvp = cameraMatrix;
glUniformMatrix4fv(mvpMatrixID, 1, GL_FALSE, &mvp[0][0]);
```

The glUniformMatrix4fv function attaches the matrix to the corresponding variable (defined by an ID) in the vertex shader, so when the vertex shader is executed, we can make use of this variable to perform vertex coordinate transformation.

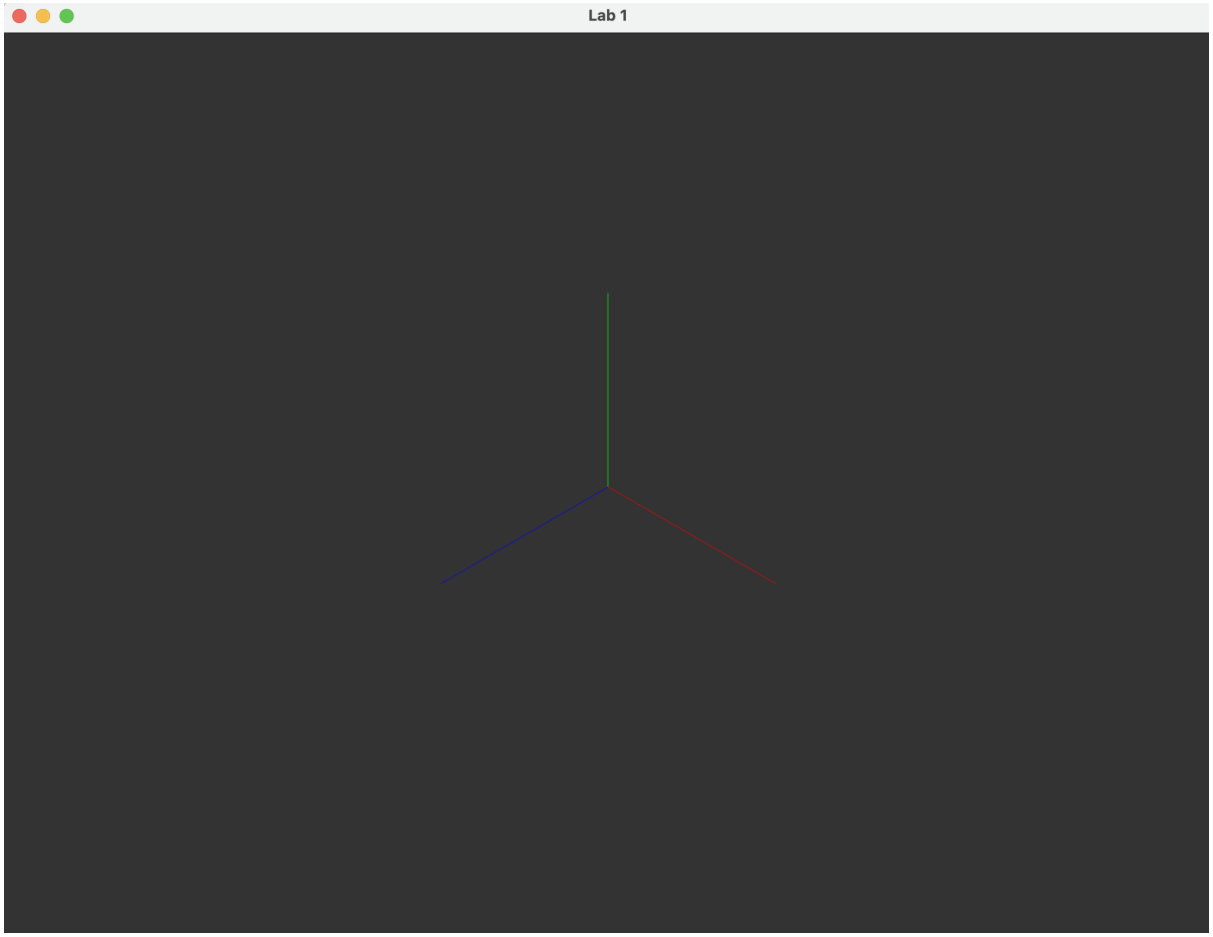
In the vertex shader, the following lines before the main function defines the MVP variable:

```
uniform mat4 MVP;
```

and in the main function, we multiple MVP matrix with the vertex position.

```
void main() {
   gl_Position =  MVP * vec4(vertexPosition, 1);
}
```

Here we simply take the model-view-projection matrix we have defined by glUniformMatrix4fv and multiply it with vertex coordinates. Note the use of homogeneous coordinates here: we append 1 to convert vertex position to a vector of 4 components.
If you compile and run the program, you will see a coordinate system in 3D as follows.



Press left/right arrow key to move the camera around the scene.
Could you explain the formula that controls the movement of the camera using the left/right arrow key?

3. Modeling a 3D box

Let us now model a 3D box centered at (0, 0, 0) to learn about how to add a basic object and its transformations. We model a cube spanning from -1 to 1 in each dimension, so the initial size of the cube is 2x2x2 units.

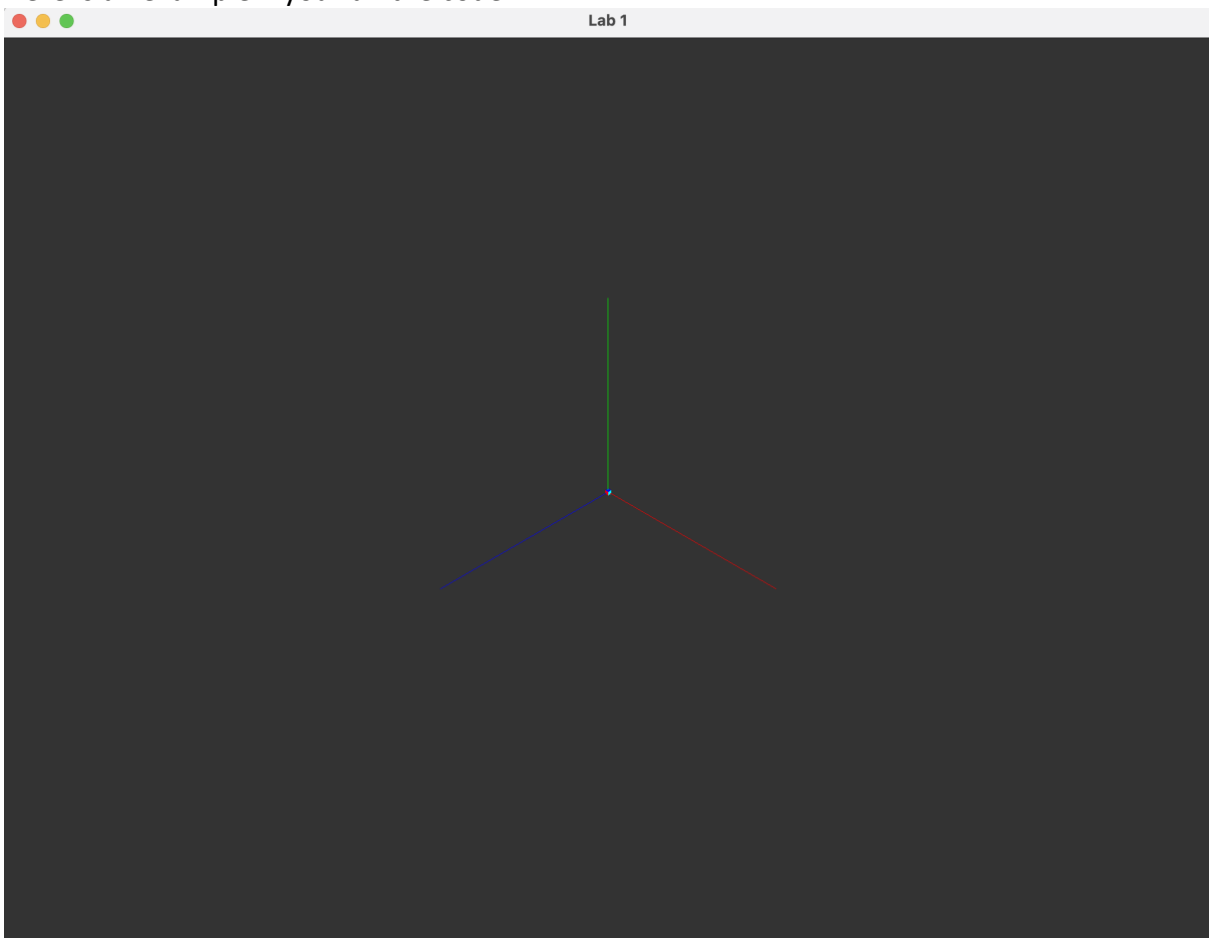
For generality, we capture the box in its own C++ struct, named Box. Each box has its own position, scale, and rotation, which specifies how to transform the box to position it in the world space. The model matrix captures the scale, rotation, and translation of the cube to transform the cube to the world space.

In lab1_cube.cpp, the geometry and color definition of the cube is provided in the Box struct. For debugging purposes, we colorize each face of the cube with red (front), green (left), blue (top), yellow (back), right (cyan), bottom (magenta).

A default box is already created for you in the main function. To show the box, add the following code to the main loop. The render function draws the box using the current view and projection transformation matrix.

```
// Render the box
mybox.render(vp);
```

Here is an example if you run the code:



If you look carefully at the center of the coordinate system, you will see something! This is our box but in the current view it is just tiny!

Could you explain why the box appears so small?

4. Model transform

Let us now explore model transform to make our box bigger. So far, the cube is of size 2x2, at the origin (0, 0, 0). Let us implement a transform to bring our box to a proper scale, rotation, and position.

Implementation. Let us define our model transformation matrix, as follows.

```
glm::mat4 modelMatrix = glm::mat4();
// Translate the box to its position
modelMatrix = glm::translate(modelMatrix, position);
// Scale the box along each axis
modelMatrix = glm::scale(modelMatrix, scale);
```

The above code snippet first scales the cube along each axis, so that the cube represents its actual size. We then move the box to the expected location by translating it. Mathematically, the model transform can be written as

$$Mp = M_{translatePosition}\, M_{scale}\, p$$

where p is the homogeneous coordinates of a point, represented by column vector (x, y, z, 1), and M is final model transform matrix.

It is worth noting that the order of the matrix transform in OpenGL code should match the order of the matrix product in the math formula. In our current convention, we have to define the matrix in OpenGL following the from left to right in math formula, i.e., we define the translation to the expected position first, then the scale.

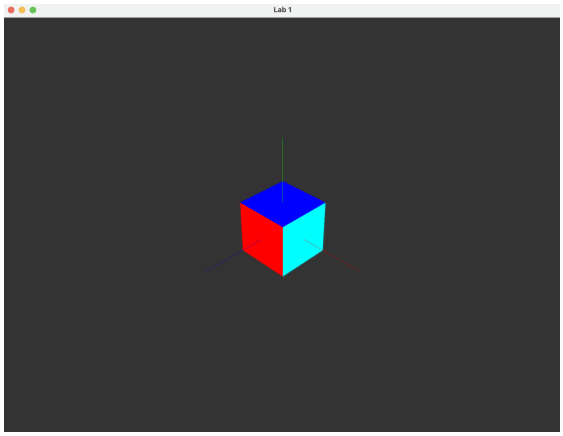
To make use of the model transform, we multiply it to the view projection matrix passed into the render function:

```
glm::mat4 mvp = cameraMatrix * modelMatrix;
```

Implementation. Try setting your box to a new location by modifying its current initialize function call.

```
// A default box
Box mybox;
mybox.initialize(glm::vec3(0, 0, 0),       // translation
                 glm::vec3(30, 30, 30)     // scale
);
```

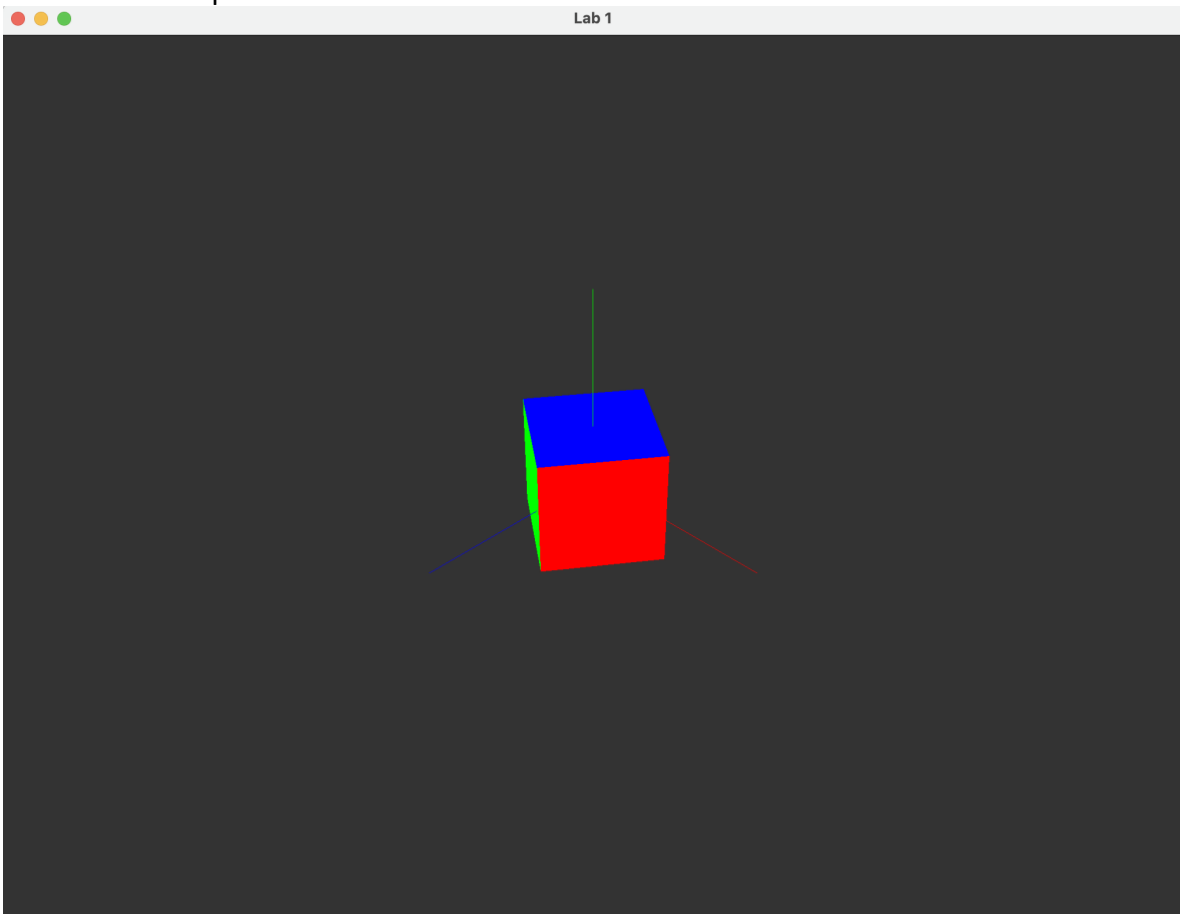
Recompile, and check if your cube moves to a new location. Below is an example result.



Task:

Implement a rotation about an arbitrary axis for the cube. Hint: use glm::rotate() function.

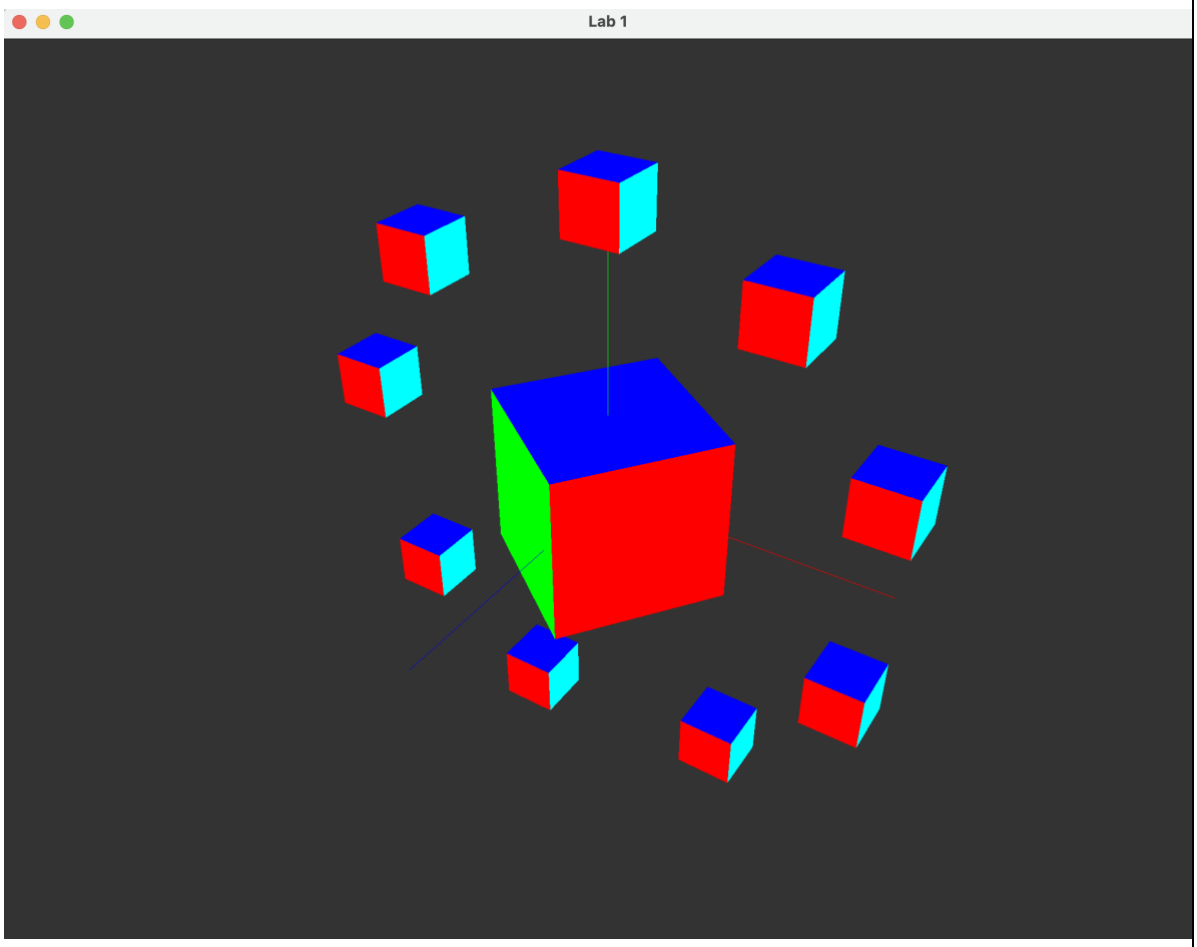
Here is an example:



Then add more cubes to the scene.

Could you position the cubes along a particular path?

Here is an example that positions the cube along a circular path:



Submission:
Package only lab1_cube.cpp, and an mp4 video that captures the rendering of your cube scene into lab1_results.zip. On Blackboard, go to Submissions -> Lab 1 and upload your zip file.
Please only pack the .cpp and .mp4. Do not upload the entire source code unless you have other dependencies.

Deadline: **Wednesday, Oct 09, 2024, at 12pm (noon).**

Marking: You will get a complete/incomplete score based on your results.