Augmented Reality
Lab 2

Gareth Young, Binh-Son Hua
Trinity College Dublin

In this lab, we will recap the graphics pipeline, reviewing its key concepts and and functionality. We will develop a simple OpenGL application to procedurally create and render a virtual city with buildings. Our objectives are:

- To recap the basic concepts of OpenGL
- To perform a rendering of a simple triangle mesh with texture
- To procedurally model a virtual city of buildings

0. Background

First of all, please read carefully the provided source code. The provided source code implements a typical OpenGL application. If you have taken computer graphics before, you should find the code familiar and easy to work with. You can work on the basics of OpenGL in the supplementary material if necessary.

In the implementation, we represent our building geometry by a canonical cube, centered at (0, 0, 0). The original model of the cube spans from -1 to 1 in each dimension, so the initial size of the cube is 2x2x2 units.
Each building has its own position and scale, which specifies its location and size in the world space. Starting with the canonical cube, we will translate and scale the cube to transform the cube into the actual building on the ground plane. We assume that the ground plane is at y = 0. We also assume that our buildings are always axis-aligned, i.e., there is no rotation of the building about any axes.

You will find the geometry and color definition of a cube already provided, organized into a separate C++ struct. By default, for debugging purposes, we colorize each face of the cube with red (front), green (left), blue (top), yellow (back), right (cyan), bottom (magenta).

However, the camera view and the projection transform have not been set, and therefore, if you compile and run the program, you will see a blank sky blue screen. But don't worry, we are very close to see some rendering. Let us proceed with adding a camera and its 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 lab2.cpp.

```
static glm::vec3 eye_center(300, 300.0f, 0);

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 properly as above.

2. Perspective projection

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

Implementation. The projection matrix can be constructed by adding this code to before the main loop (look for glm::mat4 projectionMatrix).

```
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;
for (int i = 0; i < buildings.size(); ++i)
    buildings[i].render(vp);
```

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

```
// Set model-view-projection matrix
glm::mat4 mvp = cameraMatrix * modelMatrix;
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:

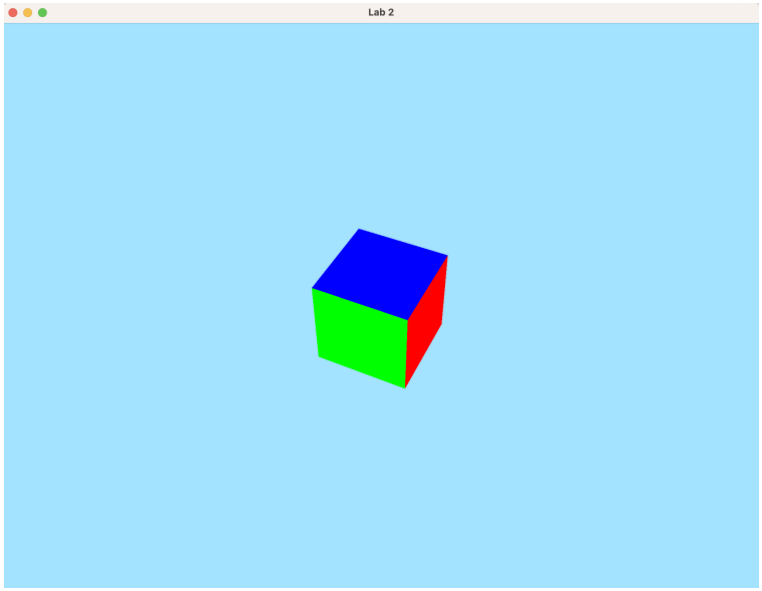
```
uniform mat4 MVP;
```

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 cube as follows. Press left/right arrow key to rotate the cube.



3. Model transform

Let us now explore model transform. So far, our building is represented by the canonical cube, of size 2x2 at the origin (0, 0, 0). We have a simple model transform implemented with a scale. Let us polish this part to transform our building to a proper scale 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 to make it look like a building
modelMatrix = glm::scale(modelMatrix, scale);
// Move our box up so that the bottom face is at the ground (y = 0)
modelMatrix = glm::translate(modelMatrix, glm::vec3(0, 1, 0));
```

The above code snippet first performs a translation to the canonical cube, move it up by 1 unit along the y-axis, so the cube now stays on the ground plane (y = 0). We then scale the cube along each axis, so that the cube represents an actual building size. Here we model a skyscraper, so the dimension along y-axis is the greatest of all dimensions. Finally, we move the building to the final location. Mathematically, the model transform can be written as

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

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, and finally the translation up by 1 unit.

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 building to a new location by modifying the first vec3 parameter of the initialize function call.

```
Building b;
b.initialize(glm::vec3(50, 0, 50), glm::vec3(30, 30, 30));
buildings.push_back(b);
```

Recompile, and check if your cube moves to a new location.

4. Texture mapping

To improve the photorealism of our building, a simple approach is to wrap an image onto the cube surfaces. This is called texture mapping. Lab2 package includes a few textures (generated by AI) that can be used for texture mapping. The textures are in JPG format which you can view by a general image viewer.

To perform texture mapping, we first need to add to each vertex a new attribute: UV coordinates. The UV coordinates defines how we can retrieve the color values from the texture and apply them to a surface. In our case, this is very simple. We map the entire texture to the entire face of our cube. We separately map the texture for the front, back, left, right of the cube. For simplicity, we reuse the same texture for all these four faces. We prefer a uniform color for the top and bottom of a building.

Implementation. Let us define the UV coordinates for each vertex of a cube, as follows.

```
GLfloat uv_buffer_data[48] = {
    // Front
    0.0f, 1.0f,
    1.0f, 1.0f,
    1.0f, 0.0f,
    0.0f, 0.0f,
```

```
    // Back
    0.0f, 1.0f,
    1.0f, 1.0f,
    1.0f, 0.0f,
    0.0f, 0.0f,

    // Left
    0.0f, 1.0f,
    1.0f, 1.0f,
    1.0f, 0.0f,
    0.0f, 0.0f,

    // Right
    0.0f, 1.0f,
    1.0f, 1.0f,
    1.0f, 0.0f,
    0.0f, 0.0f,

    // Top - we do not want texture the top
    0.0f, 0.0f,
    0.0f, 0.0f,
    0.0f, 0.0f,
    0.0f, 0.0f,

    // Bottom - we do not want texture the bottom
    0.0f, 0.0f,
    0.0f, 0.0f,
    0.0f, 0.0f,
    0.0f, 0.0f,
};
```

Let us now extend the initialize function of our Building struct to include

```
    // Create a vertex buffer object to store the UV data
    glGenBuffers(1, &uvBufferID);
```

```
    glBindBuffer(GL_ARRAY_BUFFER, uvBufferID);
    glBufferData(GL_ARRAY_BUFFER, sizeof(uv_buffer_data), uv_buffer_data,
GL_STATIC_DRAW);
```

We will need to load our texture into the GPU memory. This can be done by

```
textureID = LoadTexture("../lab2/facade4.jpg");
```

Inspect LoadTexture function in render/texture.cpp for the implementation of texture loading.

To perform texture mapping, we need to sample the texture in the fragment shader. In the initialize function, add the following line to retrieve the ID of the textureSampler variable in the fragment shader. Note that this line must be after the LoadShaders function are called.

```
    // Get a handle for our "textureSampler" uniform
    textureSamplerID  = glGetUniformLocation(programID, "textureSampler");
```

In the render function, we now need to activate the UV buffer, and binds the texture sampler to use the texture we have just loaded. This can be done before glDrawElements.

```
    glEnableVertexAttribArray(2);
    glBindBuffer(GL_ARRAY_BUFFER, uvBufferID);
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0, 0);

    // Set textureSampler to use texture unit 0
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, textureID);
    glUniform1i(textureSamplerID, 0);
```

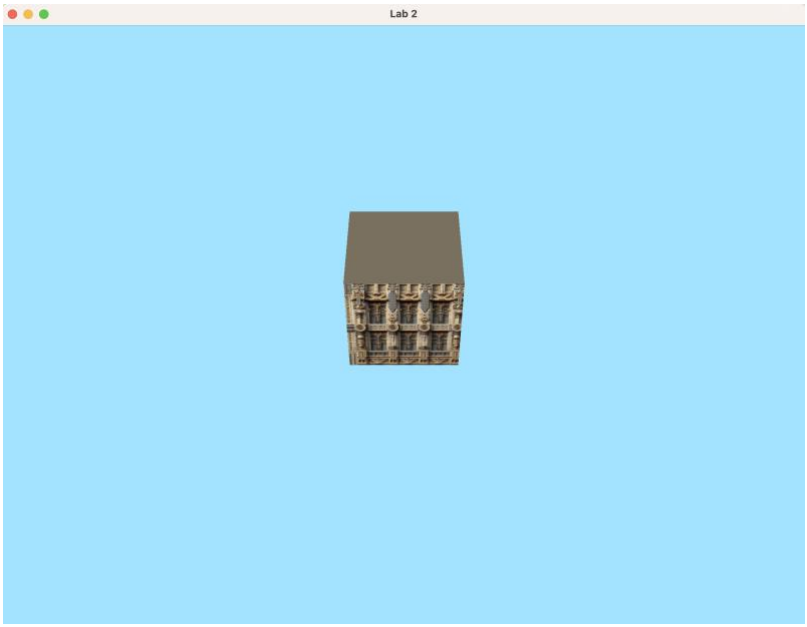
Finally, go to the vertex shader and the fragment shader, enable support for texture mapping by following the comments there.

Recompile and run the program. You will see a building like this screenshot.

Here we see that the color value now modulates to the texture value. As we do not need per-vertex color for now, let us temporarily set the color values to 1 in the initialize function.

```
for (int i = 0; i < 72; ++i) color_buffer_data[i] = 1.0f;
```

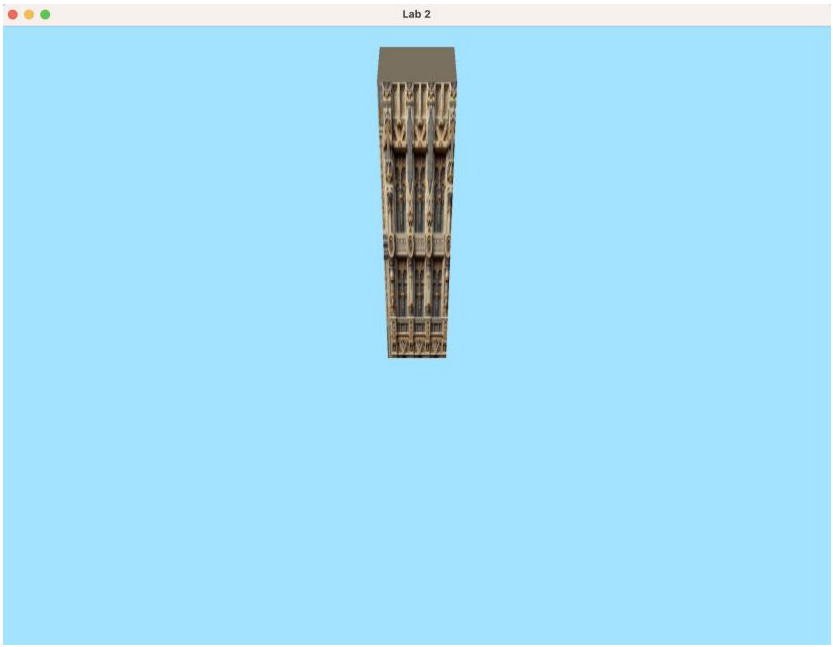
Recompile and re-run. You will see the following result.



Set your building to have more realistic dimensions. For example, use

```
b.initialize(glm::vec3(0, 0, 0), glm::vec3(16, 80, 16));
```

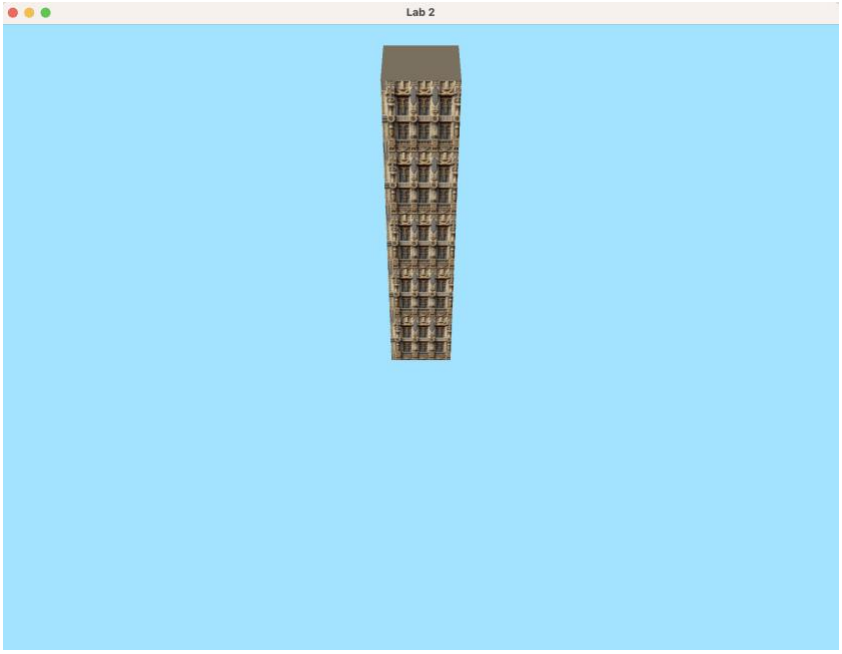
to make a tall building where its height is 5x its width and depth.



One particular problem is that after scaling up the building along the y-axis, its texture is strength unevenly. To deal with this problem, we can also scale up the UV coordinates (currently in the range of [0, 1]). Our texture loading function has been configured such that if UV is greater than 1, the texture will be automatically repeated. In the initialize function, add the following code before UV buffer is created to see the texture tiling effect.

```
for (int i = 0; i < 24; ++i) uv_buffer_data[2*i+1] *= 5;
```

This scales V coordinates to 5x, which means we will tile the texture vertically 5 times.



5. Procedural generation of buildings

Now let us scale up by adding more buildings! So far, we generate a single building:

```
std::vector<Building> buildings;
Building b;
b.initialize(glm::vec3(0, 0, 0), glm::vec3(16, 80, 16));
buildings.push_back(b);
```

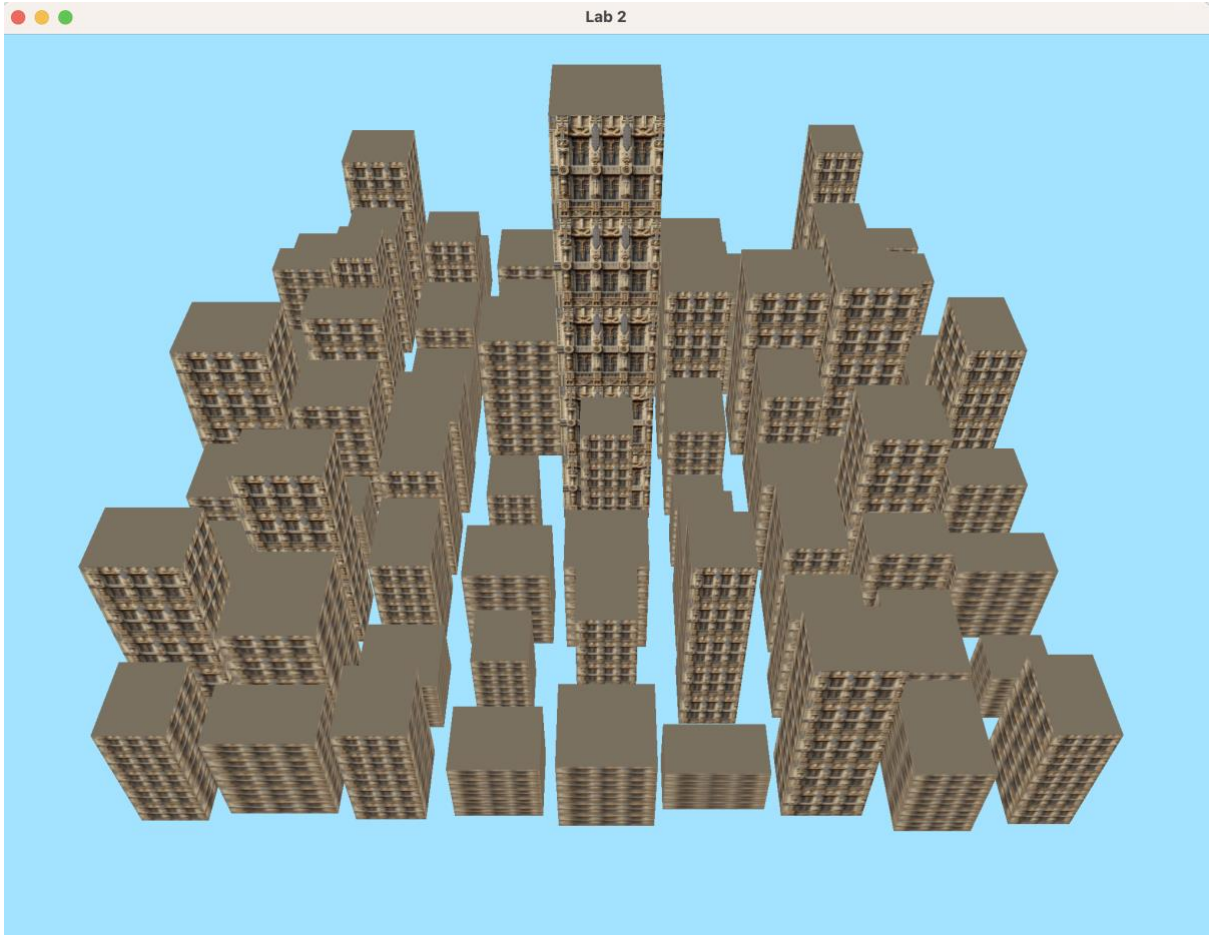
Note that we have already had a std::vector that stores an array of buildings. Therefore, to generate a block of buildings, we simply need to generate its locations, scales, texture types, and add them to the building vector.

Implementation. A very simple generation can be like this. We use a 9x9 layout, with a tallest building at the center, and surrounding buildings to be lower with some randomizations.

```
std::vector<Building> buildings;
int k = 0;
for (int z = -4; z <= 4; ++z)
{
    for (int x = -4; x <= 4; ++x)
    {
        Building b;
        if (x == 0 && z == 0) {
            // Tallest building
```

```
            b.initialize(glm::vec3(x * 32, 0, z * 32), glm::vec3(16, 80, 16));
        } else {
            if (x == 0 && z == 1) continue;     // Leave this block out to get some space for
visibility
            // Other buildings are lower
            float scaleX = 8 + 8 * randomFloat();
            float scaleY = 8 + 32 * randomFloat();
            float scaleZ = 8 + 8 * randomFloat();
            b.initialize(glm::vec3(x * 32, 0, z * 32), glm::vec3(scaleX, scaleY, scaleZ));
        }
        buildings.push_back(b);
        k++;
    }
  }
```

The result of this procedural generation is as follows.



Task:
Generate your own virtual city. You can use your own way to model the building, e.g., load the building from a 3D model file, more complex and realistic geometry and texture, or make the current building layout more realistic, adding roads, trees, etc.
Hint: Press space bar to activate automatic rotation of the virtual city and capture a video screenshot of the window.

Submission:
Package only lab2.cpp, and an mp4 video that captures the rendering of your virtual city, into lab2_results.zip. On Blackboard, go to Submissions -> Lab 2 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: Feb 5, 2025, at 12pm (noon).

Marking:
You will get 1% from this lab if you achieve comparable results as shown in the screenshots.