Computer Graphics 4052
Lab 2

Binh-Son Hua
Trinity College Dublin

In this lab, we are going to create a simple OpenGL application to learn about texture mapping in OpenGL. Our goal is to model and render a virtual city with buildings. Our objectives are:

- To recap the basic concepts of OpenGL
- To perform a rendering of a simple cube with texture
- To procedurally model a virtual city of buildings
- To learn how to render the background environment using a texture mapped skybox

1. Texture mapping

Let us start with the cube model we had in Lab 1. Recall that we model the cube centered at (0, 0, 0) with its dimensions spanning from -1 to 1, so the initial size of the cube is 2x2x2 units. Each vertex of the cube stores the 3D coordinates along with the RGB color of each vertex.

The idea is to use this cube to represent a building. To improve the photorealism of our building, we will wrap an image onto the cube surfaces to model the building facades. This process is called texture mapping. The sample code in Lab 2 packages a few textures (generated by AI) that can be used for modelling the facades. The textures are in JPG format which you can view by a generic image viewer.

To perform texture mapping, we first need to extend the cube vertices with 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, and skip the textures on the top and bottom faces.

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 = LoadTextureTileBox("../lab2/facade4.jpg");
```

Inspect LoadTextureTileBox function to learn about the implementation of texture loading.

```
// Generate an OpenGL texture and make use of it
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);

// To tile textures on a box, we set wrapping to repeat
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

// Load the image into the current OpenGL texture
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, w, h, 0, GL_RGB, GL_UNSIGNED_BYTE,
img);
```

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, and enable support for texture mapping. The changes for the vertex shader include:

```
#version 330 core

// TODO: Vertex's UV input to this vertex shader
//layout(location = 2) in vec2 vertexUV;

// TODO: UV output to fragment shader
//out vec2 uv;

void main() {
    ...

    // TODO: We simply assign the input UV to the output UV
    //uv = ...;
}
```

The changes for the fragment shader include:

```
// TODO: To receive UV input to this fragment shader
//in vec2 uv;

// TODO: To access the texture sampler
//uniform sampler2D textureSampler;

void main()
{
    // TODO: Perform texture lookup.
    // Hint: use the GLSL texture() function
    // It takes as input the texture sampler
```

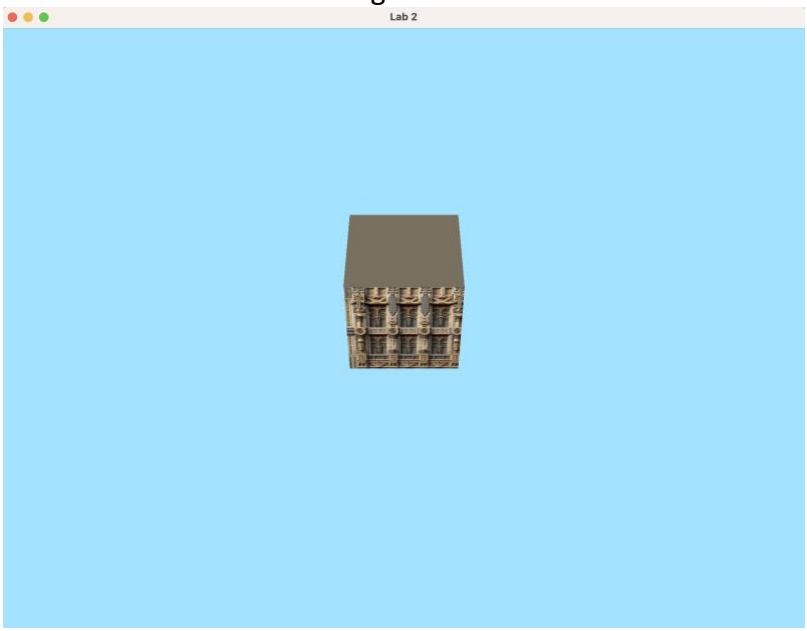
```
    // and the UV coordinate and returns an RGB color.
    // finalColor = texture(...).rgb;

    // For the existing vertex color variable, you can remove it, or
    // multiply its value with the texture, e.g.,
    // finalColor = color * texture(...).rgb;
}
```

If you do not need per-vertex color for now, you can temporarily set the color values to 1 in the box initialize function.

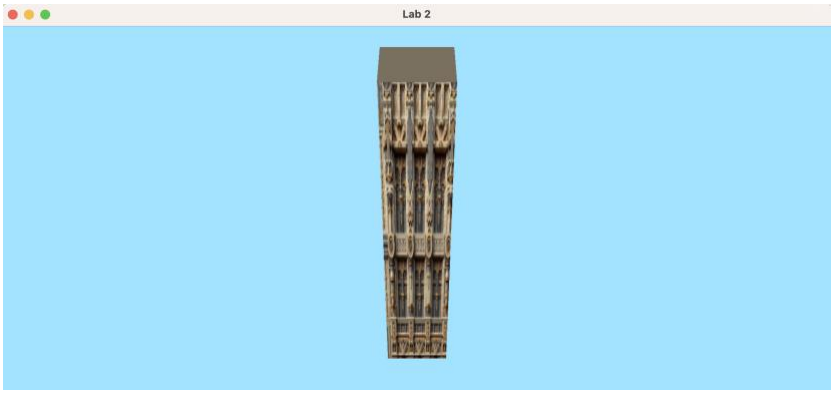
```
for (int i = 0; i < 72; ++i) color_buffer_data[i] = 1.0f;
```

Recompile and re-run. You will see a building like this screenshot.



2. Building modeling

The initial textured box is cubical and does not look very much like a high rise. Let us modify the building to have realistic dimension ratios. Could you modify the box transforms to make a tall building where its height is 5x its width and depth, as in this screenshot?
Hint: use x-dimension = 16.



Now one particular problem is that after scaling up the building, its texture is stretched and becomes distorted. This is because the current UV coordinates is in [0, 1], and the entire image is stretched to map to this range.

Let us now think about how to tile our texture. Recall that in our texture loading function, the texture parameters have been configured such that if UV is greater than 1, the texture will be automatically repeated. See the LoadTexture function:

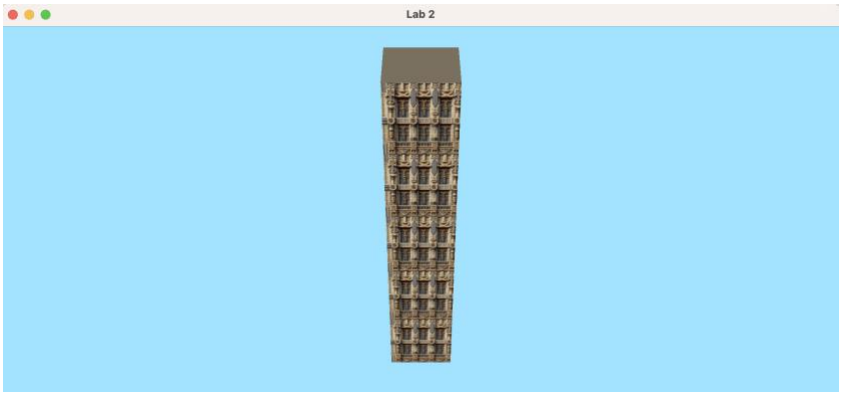
```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

where GL_REPEAT is set of GL_TEXTURE_WRAP_S and GL_TEXTURE_WRAP_T.

Therefore, to repeat the texture, we can simply scale up the UV coordinates to a range greater than 1. 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;
```

Could you explain which texture coordinates are modified in the above for-loop? Why?
The screenshot of the building is as follows.



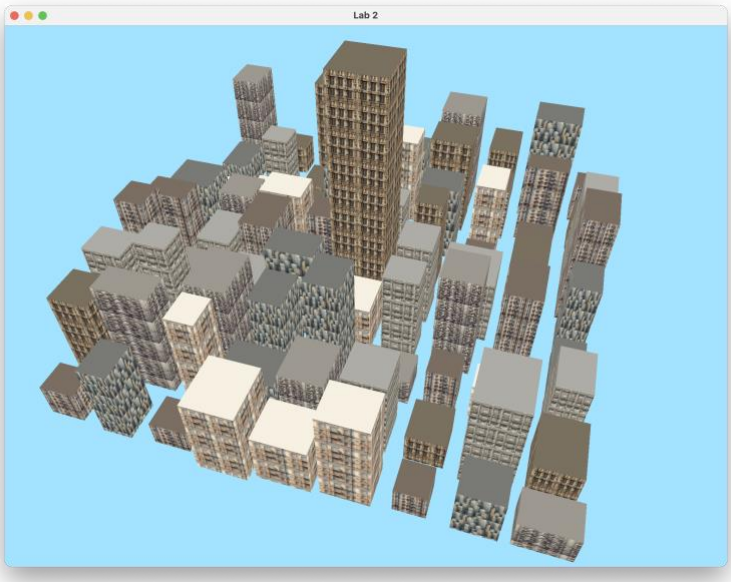
3. Procedural generation of a virtual city

Now let us scale up by adding more buildings! Define a C++ vector that stores multiple buildings using the following example:

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

To generate a block of buildings, we simply need to generate its locations, scales, texture types, and add them to the building vector.

An example of a virtual city is as follows.



Task:
Model your own virtual city. Feel free to think of a creative layout of your city with diverse building façades and roads. Use randomizations if needed.

4. Environment modeling

Have you noticed that so far in our labs, our background in OpenGL is just always a plain color?

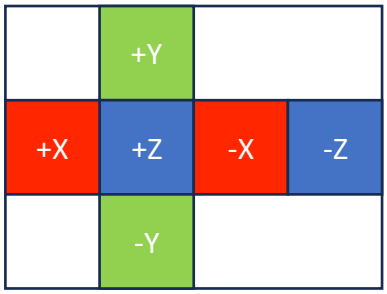
In this section, we will learn how to use our simple cube and texture mapping to model the background environment of our scene!

To do so, let us recall how we have been modelling and using our cube so far: we model a cube such that its triangles face outwards, and we view the outer surfaces of the cube from a distance.

Now let's think about the opposite: can we scale up our cube, and imagine that we stay inside the cube and then view its inner surfaces?

This will provide us a convenient way to model the environment: we can capture an environment using six images of front, back, left, right, top, and bottom view, each with a 90-degree field of view, and then map each of these images to each inner surface of our cube. When we view the inner surfaces of the cube, the textures will allow us to reproduce the perception of being in the captured environment.

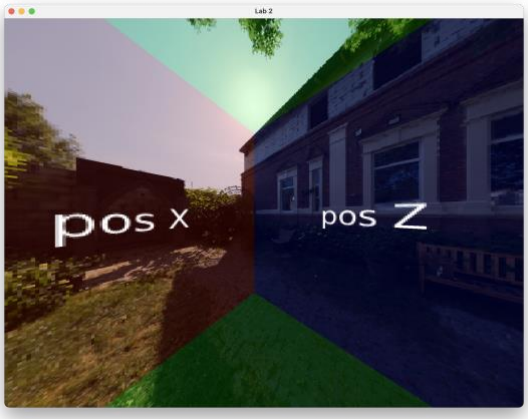
The cube with its inner surfaces textured for environment modelling is also known as the **skybox**. For learning purposes, we pack all six textures for the skybox into a single image, with the following layout. Each texture is of size 256x256, so the entire image is 1024x768.

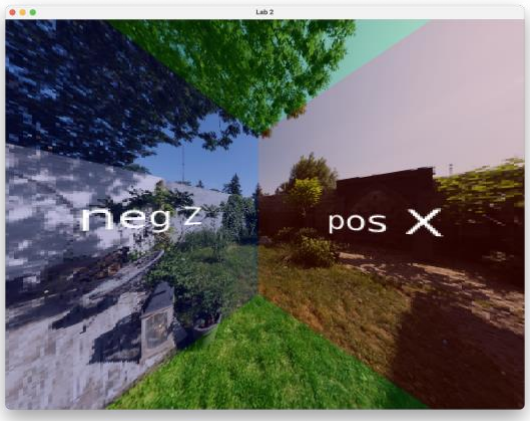


Note that in this convention, we assume when we look along +Z axis to see the +Z face when inside the box, we apply the right-handed rule so that the +X axis is at our left. Two examples of the skybox texture and its debug version are included in Lab 2 package for your references.

An example of the textures and the rendering of the skybox are shown below.







Task:
Implement the skybox rendering into a new lab2_skybox.cpp.

You can base on your code in lab2_building.cpp. Also learn how to modify lab2/CMakeLists.txt to include a new cpp file for compilation by following the example of lab2_building.cpp there.

Note that you have to implement the texture mapping of the skybox from scratch. Do not use the cubemap extensions in OpenGL, which is not the goal of this lab.

Hint: Copy below if you need it.

- Set the camera view to be inside the cube.
- Modify the cube buffers so that the inner surfaces of the cube becomes front facing in counter-clockwise order.
- Set the UV coordinates properly for each face to map to the +X, -X, +Z, -Z, +Y, -Y region in the texture. Make use of the debug textures.
- Set the texture parameters properly to remove any seams appeared in the rendering. Consider setting GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T, GL_TEXTURE_MIN_FILTER, and GL_TEXTURE_MAG_FILTER.

Submission:
Package only lab2_building.cpp, and lab2_skybox.cpp, and two mp4 videos that captures the rendering of your virtual city and skybox in different viewpoints, into lab2_results.zip.

You can also consider submitting a single lab2.cpp that renders both the virtual city in a skybox background and a single mp4 that demonstrates these two implementations.

You can find or make your own building textures and skybox textures as well.

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: **Wednesday, 30 October, 2024, at 12pm (noon).**

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

References

1) Public 3D assets: https://polyhaven.com/

2) HDRI to Cubemap: https://matheowis.github.io/HDRI-to-CubeMap/