1 Task 1 – Build the BOC(HK) Tower (40 points)

1.1 Task Description

In this task, you are going to use the 2D transformation matrix you learned during the lecture to transfer one base triangle to build the BOC(HK) tower (see Figure 1), which is a masterpiece of the world-renowned Chinese-American architect I.M. Pei. His inspiration for the building's construction was the elegant stature of bamboo, its sectioned trunk ... (find more on *this link*).

Figure 1: 2D triangle mesh of BOC(HK) tower.

Back to our assignment, the BOC tower is formed by 22 triangles. Seems to be a lot? Don't worry! There are only 4 types of them. The first 16 triangles can be obtained by purely translation and rotation. Triangles 17 to 20 require (Non-Uniform) scaling to be applied first (note that 17 and 18 are slightly wider than 19 and 20). Finally, triangles 21 and 22 are right triangles so I believe you have to also apply the shearing transform matrix.

1.2 Code Framework

1.2.1 Editor

- Feel free to use any text editor you like.
- If you don't have any favorite editor, we encourage you to use *VS Code* and **add some plugins** (e.g. *Live Server*) to help your coding.

1.2.2 JavaScript

Here's a great resource to reference: *https://www.w3schools.com/js/default.asp*

- console.log() prints out into the log in Chrome. Super useful for debugging.

- Start a variable with var keyword.
  - Ex: var x = 5;
  - You can define variables without var, but it makes the variables as global variables wherever you define.
- Functions start with the function keyword, then name and (parameters)
  - There are many ways to define functions in JavaScript. We encourage you to reference the w3schools tutorials above if you have questions or concerns.

1.2.3 Start the webpage

To start the webpage,

- First, you need to start a local web server.
  - **Option 1:** Open a terminal in the directory containing *index.html* and run `python -m http.server`. Then, open your browser and go to `http://localhost:8000`.
    (Note: you will need to restart the server each time you want to preview updates to your code.)
  - **Option 2 (*recommended*):** Install the *Live Server* extension in VS Code. This plugin launches a local web server and automatically refreshes the page whenever you save changes to your code.
- If everything goes well, you should now see three triangles shown in the Figure 2.

Figure 2: Screenshot of the task 1 start state, the rightmost triangle is the base triangle and the left two are obtained by translation and rotation of the base triangle.

Note:

- You are recommended to use Google Chrome and disable the cache feature when you are debugging. *This link* shows how to disable the cache.

- If you double-click to open *index.html* you will probably not see anything, this is because the page has to load other local files (shaders, models, etc) which are blocked by the CORS policy (see details in *here*)

1.2.4 glMatrix.js

In this assignment, we use glMatrix.js to compute the vector and matrix operations. The doc can be found here: *https://glmatrix.net/docs/index.html*. Specifically, You may find mat3 useful in this task (for 2-dimensional transformation) and mat4 useful in the next task (for 3-dimensional transformation).

1.3 Task instruction

Finally, we can start to work on building the BOC tower. In this task, we want you to focus on the 2D transformation. You only need to care about the one file, *boc-tower.js*, and nothing else needs to be modified. In *boc-tower.js*, you need to:

1. Create 22 triangles by calling function createBaseTriangleVertices();
2. Create transformation matrix for each triangle by matrix multiplication of translation, rotation, scaling and shearing matrices; See Section 1.2.4 for matrix operations.
3. Transform each triangle by using transformTriangleVertices();
4. Concatenate all the triangles into a single list by using convertToList() for WebGL to render. (already done for you)

2 Task 2 – Spinning Teapot (60 points)

When you first start the webpage in your web browser as described in Section 1.2.3, you should see one static teapot as shown in Figure 3. When you finish this task, you will get three spinning teapots as shown in Figure 6.

Figure 3: Screenshot of the task 2 start state.

2.1 Implement MVP Matrix (25 points)

In this task, you are going to implement MVP matrix manually.

2.1.1 Model Translation Matrix (5 points)

Start from render-teapot.js Line 74. Implement the translation matrix that can translate the model by `vec3(offset, 0, 0)`. See codes for detailed instructions.

2.1.2 Model Rotation Matrix (5 points)

Start from render-teapot.js Line 86. See codes for detailed instructions.

2.1.3 Matrix Transformation Order Matters! (5 points)

Start from render-teapot.js Line 97. Try to change the order of the above translation and rotation. Describe your findings and explain why.

2.1.4 View Matrix (5 points)

Start from render-teapot.js Line 110. See codes for detailed instructions.

2.1.5 Perspective Matrix (5 points)

Start from render-teapot.js Line 125. Please manually set the perspective matrix as shown in Figure 4. See codes for detailed instructions.

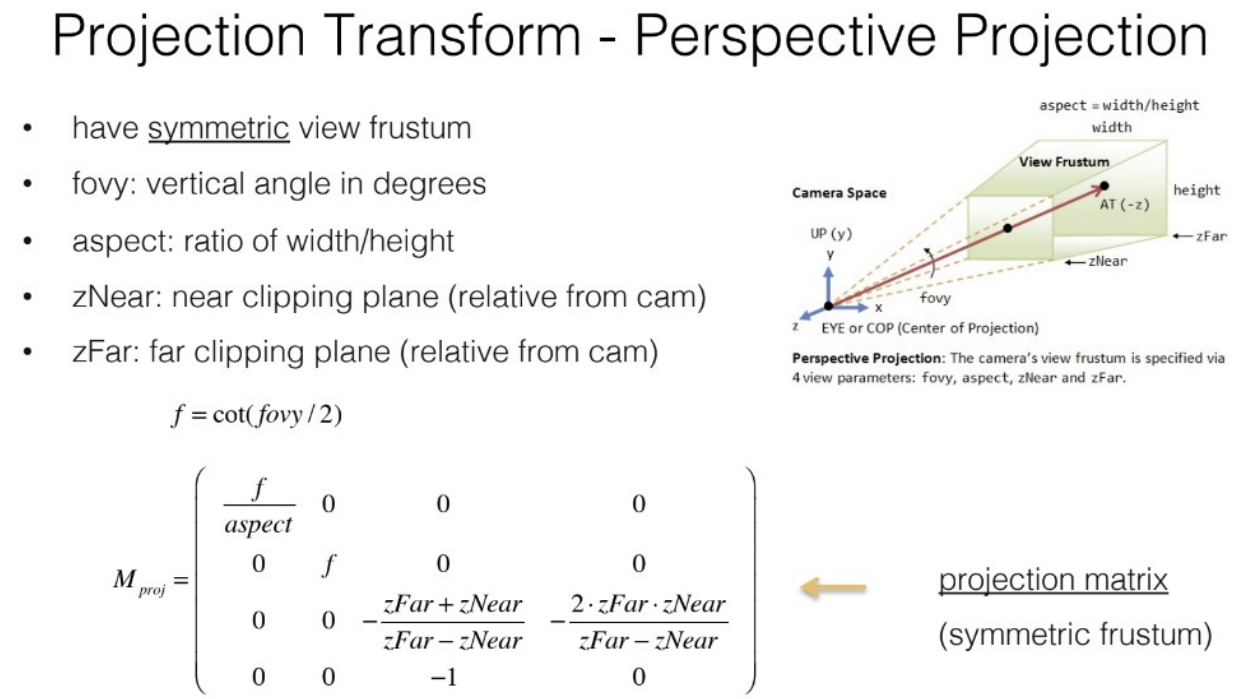

Figure 4: Perspective Matrix.

2.2 Implement Lighting Model (35 points)

In this task, you need to implement **Gouraud** and **Phong** shading in vertex and fragment shaders by GLSL, which is the OpenGL/WebGL shading language. Read through the GLSL tutorial to understand some of the nuances of the language, *https://webglfundamentals.org/webgl/lessons/webgl-shaders-and-glsl.html*. You will find this knowledge very helpful in this task. You don't need to worry about anything involving gl.insertFunctionNameHere() function calls. We handle the vertex arrays and attributes for you, so only worry about the GLSL code in this tutorial.

2.2.1 Gouraud Shading (15 points)

You will first implement Gouraud shading, which implements per-vertex lighting (in the vertex shader) and then lets the rasterizer interpolate the resulting colors to each fragment inside the primitives (i.e., triangles). After finishing, you should get something similar as shown in Figure 5

- **Diffuse Term** Your first task will be to extend ambient lighting by adding the diffuse term in the vertex shader file `shader_vertex_gouraud.glsl`. Your changes to these shaders will appear on the middle teapot. The light sources in the scene is defined in the lightSource structs. In the shaders, extend the ambient shading calculation by the diffuse term. Note that you will need the surface normal (in model space) at the current vertex to do this calculation, which corresponds to the normal uniform in the provided vertex shader.

All computations should be performed in the view coordinate system, such that the camera is in the origin looking into the negative z direction. That means, you need to transform the normal into the view coordinate system by multiplying it by the Model matrix.

- **Specular Term** Diffuse lighting is a step in the right direction, but it does not support any view-dependent lighting effects. Extend your diffuse vertex shader by the specular term. For this calculation you will need to compute the view vector from 3D vertex position to the camera. Because you are operating in view coordinates, the camera will be in the origin, but you need to make sure to transform your vertex into the view coordinate system first. You might find the GLSL function reflect() to be useful for calculating the perfect reflector. Pay close attention to the direction of the vector that reflect is expecting.

Figure 5: Gouraud Shading (middle teapot)

2.2.2 **Phong Shading** (15 points)

Implement Phong shading (i.e., per-fragment lighting). In the Phong shading shader files provided: `shader_vertex_phong.glsl` and `shader_fragment_phong.glsl`. We have finished the vertex shader for you that does all the 3D transformations. You will need to modify the fragment shader that does all the lighting calculations.

The fragment shader should take as input the output of the vertex shader (3D vertex position and 3D normal). Make sure the normal is normalized after being interpolated by the rasterizer! You will perform the same lighting calculations that you already implemented in the vertex shader in Gouraud shading for each fragment here. (i.e. add diffuse term and specular term) After finishing, the scene will look like Figure 6

2.2.3 **Comparison of Gouraud and Phong shading** (5 points)

Comparing the per-vertex lighting (Gouraud shading) and the per-fragment lighting (Phong shading) models, what's the difference? What are the benefits and downsides of both shaders? Specifically, comment on both quality of shading and computational load.

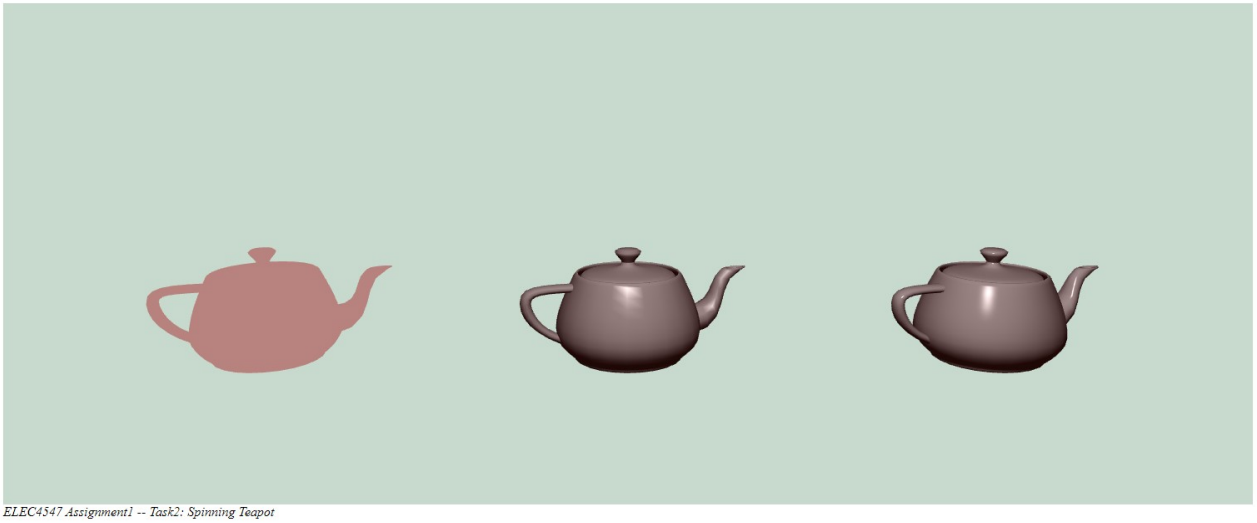

Figure 6: Phong Shading (right teapot)