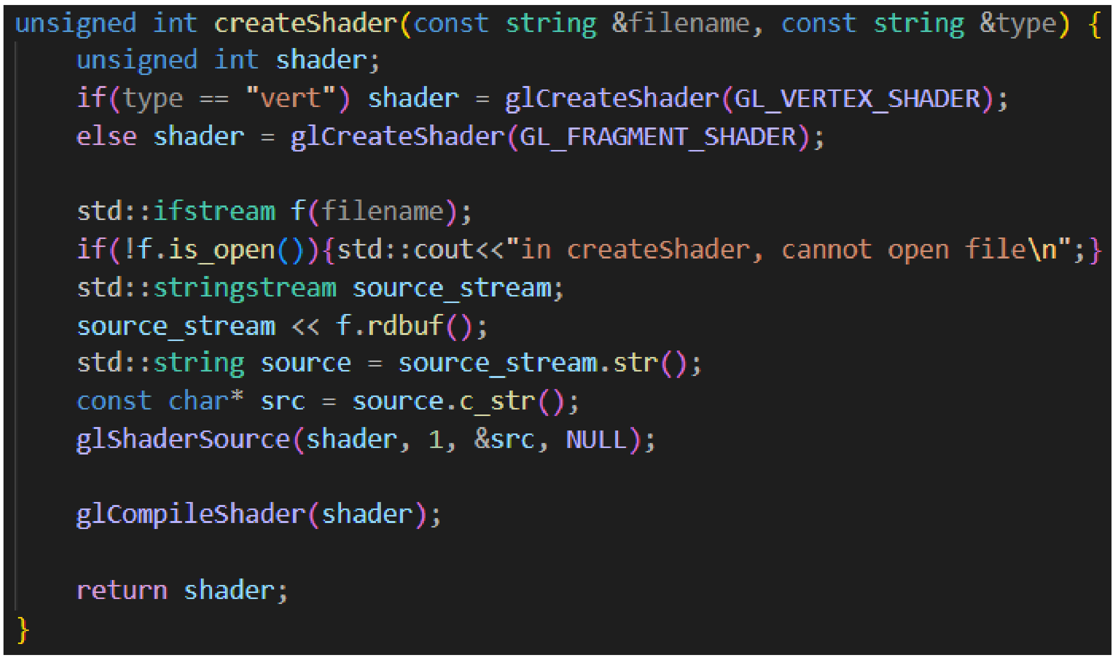
ICG HW2 Report

111550108 吳佳諭

1. Create shader and complete the shader code



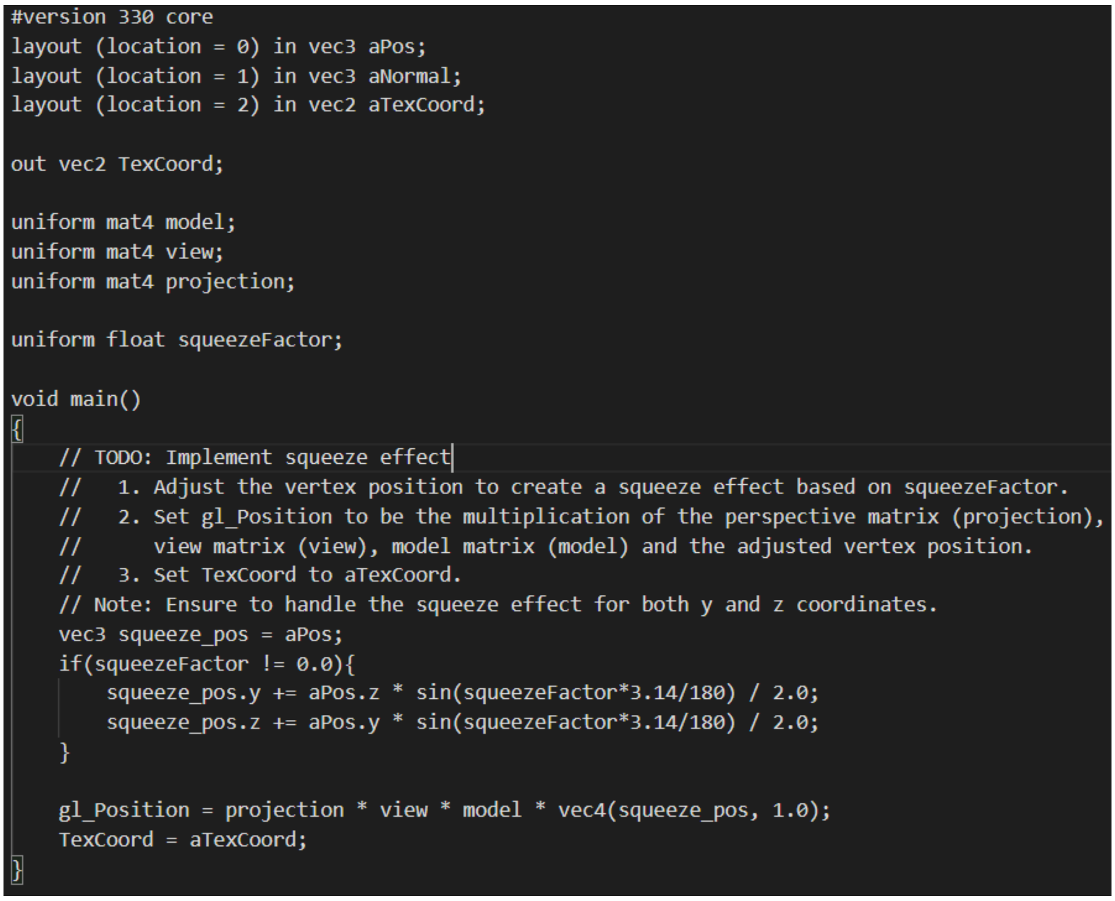
```cpp
unsigned int createShader(const string &filename, const string &type) {
    unsigned int shader;
    if(type == "vert") shader = glCreateShader(GL_VERTEX_SHADER);
    else shader = glCreateShader(GL_FRAGMENT_SHADER);

    std::ifstream f(filename);
    if(!f.is_open()){std::cout<<"in createShader, cannot open file\n";}
    std::stringstream source_stream;
    source_stream << f.rdbuf();
    std::string source = source_stream.str();
    const char* src = source.c_str();
    glShaderSource(shader, 1, &src, NULL);

    glCompileShader(shader);

    return shader;
}
```

In main.cpp, we use the function *createShader* to create shaders. First, use *glCreateShader* to create a vertex shader or a fragment shader. Then read the shader code from the file and load it into the shader by *glShaderSource*. Finally, use *glCompileShader* to compile the shader and return it.

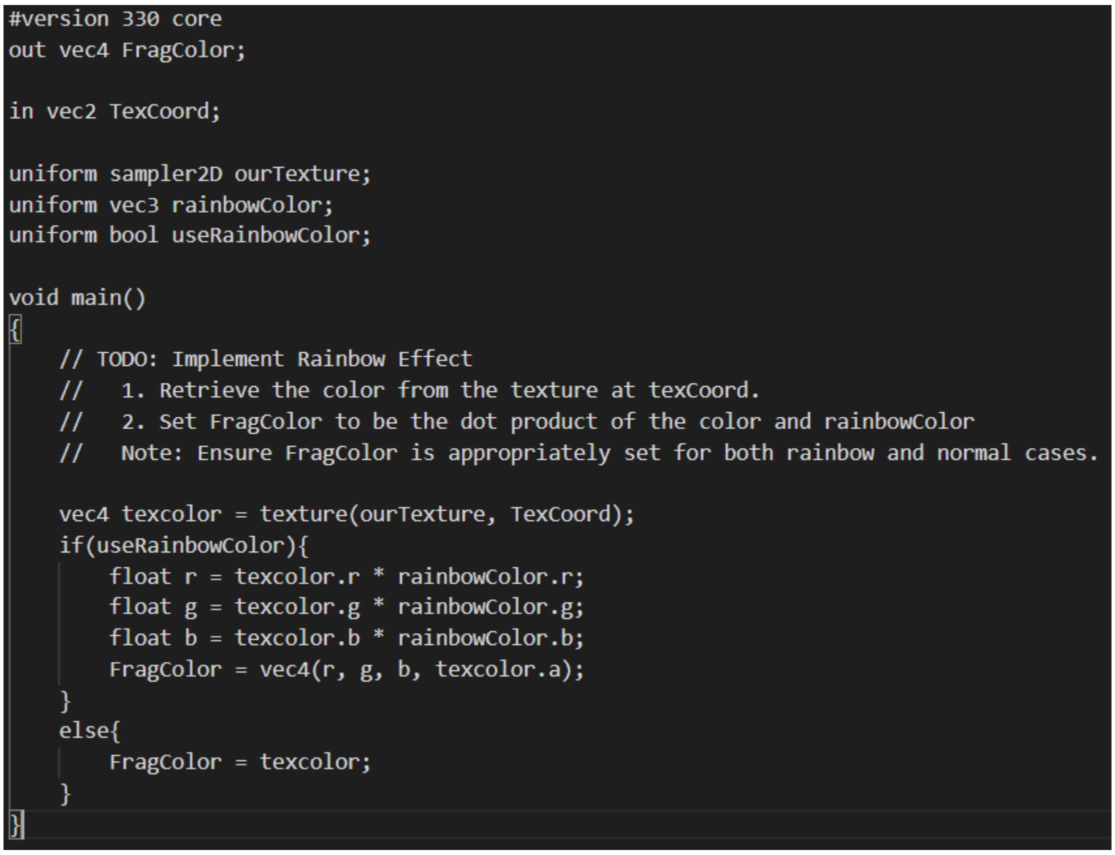
Also, we have to write our shader code. We have two shaders, vertex shader and fragment shader. We can write them in a separated file and load them to the shader later (in *createShader*).

In vertex shader, we assign the position and texture coordinate to *gl_Position* and *TexCoord*. As for squeezing, we check whether *squeezeFactor* is equal to zero. If it's not zero, then apply squeezing to the position's y and z coordinate.

In fragment shader, we pass a new uniform variable *useRainbowColor* to decide whether to apply rainbow to the object. If yes, we have fragment *color = texture color * rainbow color*. Otherwise, the fragment color is the texture color.



```glsl
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoord;

out vec2 TexCoord;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

uniform float squeezeFactor;

void main()
{
    // TODO: Implement squeeze effect
    //   1. Adjust the vertex position to create a squeeze effect based on squeezeFactor.
    //   2. Set gl_Position to be the multiplication of the perspective matrix (projection),
    //      view matrix (view), model matrix (model) and the adjusted vertex position.
    //   3. Set TexCoord to aTexCoord.
    // Note: Ensure to handle the squeeze effect for both y and z coordinates.
    vec3 squeeze_pos = aPos;
    if(squeezeFactor != 0.0){
        squeeze_pos.y += aPos.z * sin(squeezeFactor*3.14/180) / 2.0;
        squeeze_pos.z += aPos.y * sin(squeezeFactor*3.14/180) / 2.0;
    }

    gl_Position = projection * view * model * vec4(squeeze_pos, 1.0);
    TexCoord = aTexCoord;
}
```



```glsl
#version 330 core
out vec4 FragColor;

in vec2 TexCoord;

uniform sampler2D ourTexture;
uniform vec3 rainbowColor;
uniform bool useRainbowColor;

void main()
{
    // TODO: Implement Rainbow Effect
    //   1. Retrieve the color from the texture at texCoord.
    //   2. Set FragColor to be the dot product of the color and rainbowColor
    //   Note: Ensure FragColor is appropriately set for both rainbow and normal cases.

    vec4 texcolor = texture(ourTexture, TexCoord);
    if(useRainbowColor){
        float r = texcolor.r * rainbowColor.r;
        float g = texcolor.g * rainbowColor.g;
        float b = texcolor.b * rainbowColor.b;
        FragColor = vec4(r, g, b, texcolor.a);
    }
    else{
        FragColor = texcolor;
    }
}
```

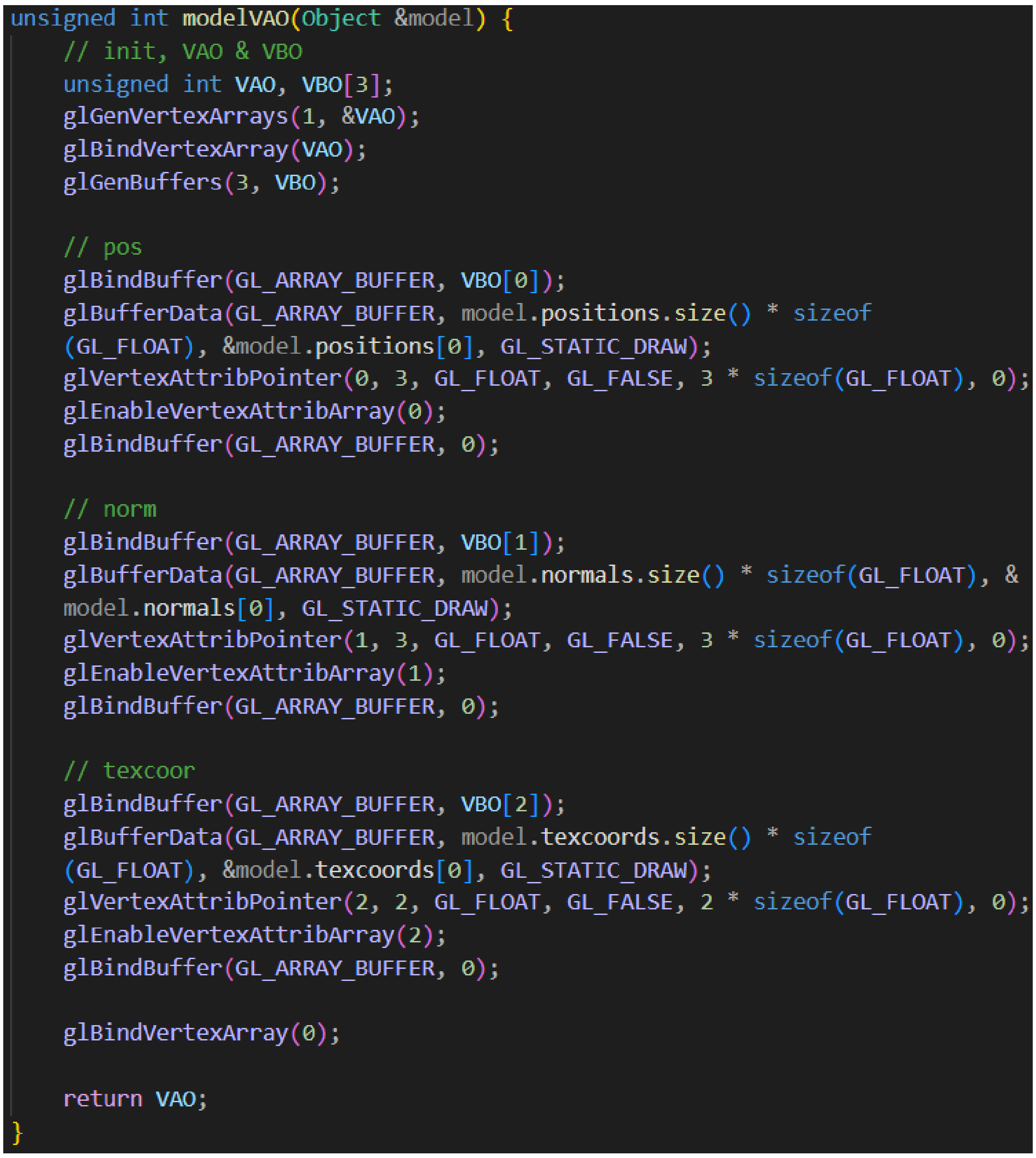
Vertex shader (up) and fragment shader (bottom)

2. Create VAO, VBO

Create the model's VAP and VBO. VBO has three elements. The first element stores the model's position. The second stores the model's normal. And the last one stores the feature.

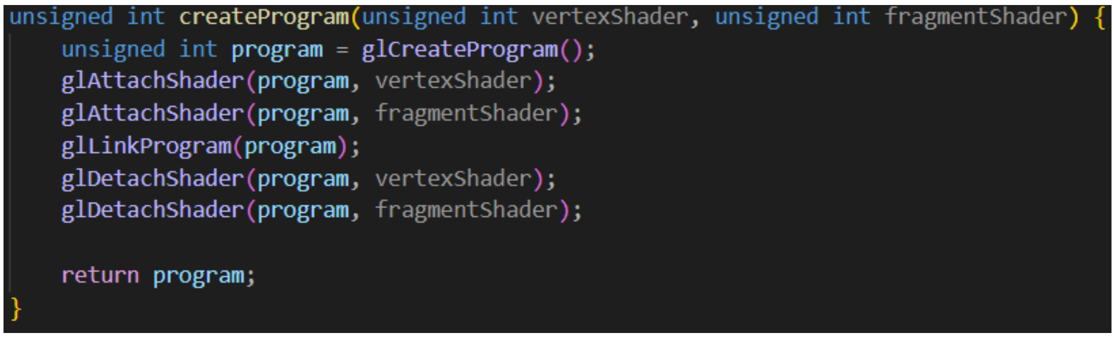
For VAO, we use *glGenVertexArrays* to generate a vertex array and use *glBindVertexArray* to bind it.

For VBO, each element (position, normal, texture) is bind to the buffer. Use *glBufferData* to tell it how's the data being stored in the array, and *glVertexAttribPointer* tells where should the data store in the array.



```
unsigned int modelVAO(Object &model) {
    // init, VAO & VBO
    unsigned int VAO, VBO[3];
    glGenVertexArrays(1, &VAO);
    glBindVertexArray(VAO);
    glGenBuffers(3, VBO);

    // pos
    glBindBuffer(GL_ARRAY_BUFFER, VBO[0]);
    glBufferData(GL_ARRAY_BUFFER, model.positions.size() * sizeof
    (GL_FLOAT), &model.positions[0], GL_STATIC_DRAW);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GL_FLOAT), 0);
    glEnableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    // norm
    glBindBuffer(GL_ARRAY_BUFFER, VBO[1]);
    glBufferData(GL_ARRAY_BUFFER, model.normals.size() * sizeof(GL_FLOAT), &
    model.normals[0], GL_STATIC_DRAW);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GL_FLOAT), 0);
    glEnableVertexAttribArray(1);
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    // texcoor
    glBindBuffer(GL_ARRAY_BUFFER, VBO[2]);
    glBufferData(GL_ARRAY_BUFFER, model.texcoords.size() * sizeof
    (GL_FLOAT), &model.texcoords[0], GL_STATIC_DRAW);
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(GL_FLOAT), 0);
    glEnableVertexAttribArray(2);
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    glBindVertexArray(0);

    return VAO;
}
```

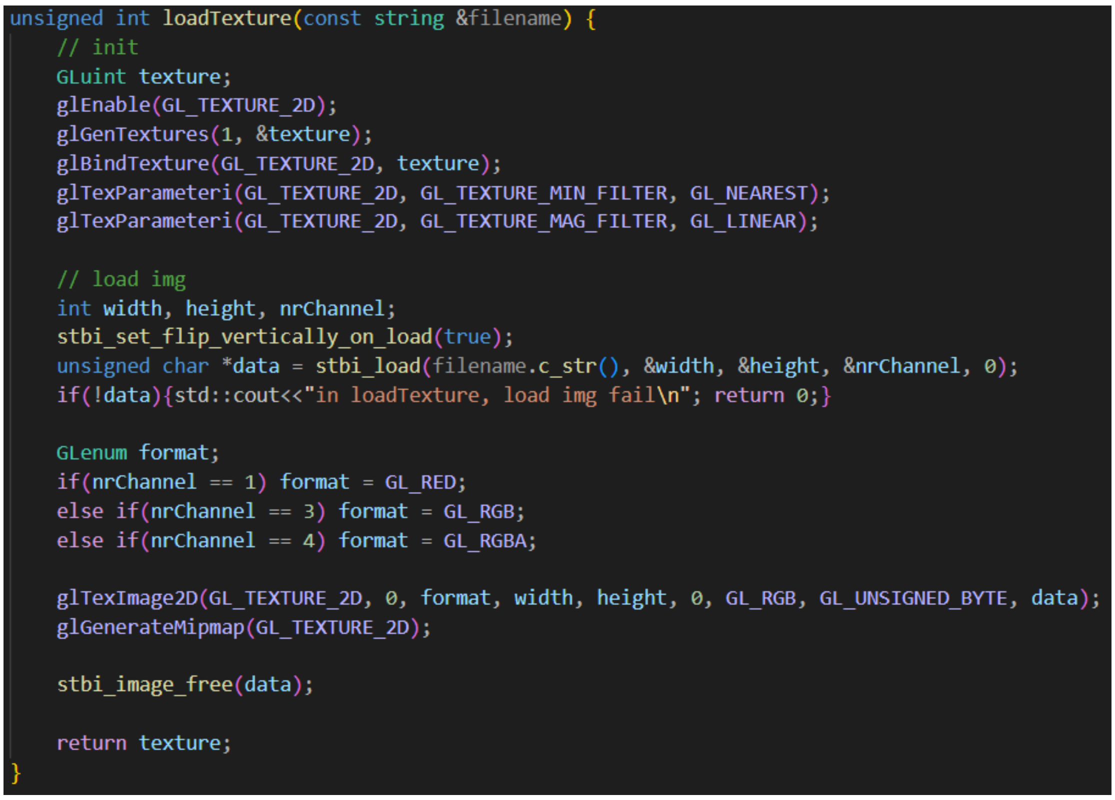
3. Create program



```cpp
unsigned int createProgram(unsigned int vertexShader, unsigned int fragmentShader) {
    unsigned int program = glCreateProgram();
    glAttachShader(program, vertexShader);
    glAttachShader(program, fragmentShader);
    glLinkProgram(program);
    glDetachShader(program, vertexShader);
    glDetachShader(program, fragmentShader);

    return program;
}
```

In *createProgram*, we combine the two shaders and return the program. First attach them to the program and link them. And detach them after we finish.

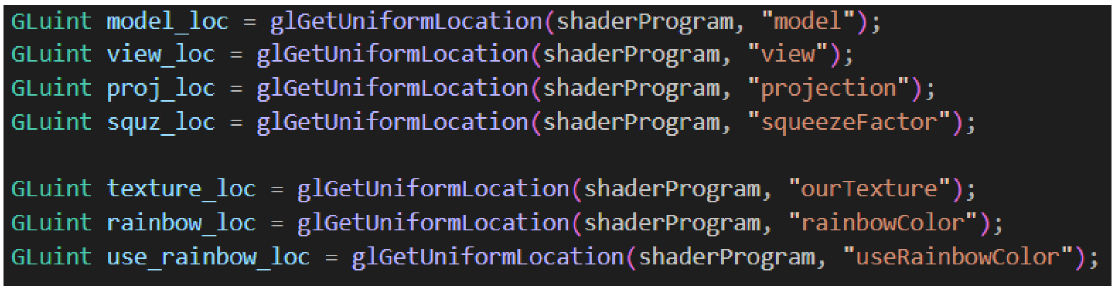
4. Load texture



```cpp
unsigned int loadTexture(const string &filename) {
    // init
    GLuint texture;
    glEnable(GL_TEXTURE_2D);
    glGenTextures(1, &texture);
    glBindTexture(GL_TEXTURE_2D, texture);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    // load img
    int width, height, nrChannel;
    stbi_set_flip_vertically_on_load(true);
    unsigned char *data = stbi_load(filename.c_str(), &width, &height, &nrChannel, 0);
    if(!data){std::cout<<"in loadTexture, load img fail\n"; return 0;}

    GLenum format;
    if(nrChannel == 1) format = GL_RED;
    else if(nrChannel == 3) format = GL_RGB;
    else if(nrChannel == 4) format = GL_RGBA;

    glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
    glGenerateMipmap(GL_TEXTURE_2D);

    stbi_image_free(data);

    return texture;
}
```

We can apply texture to the object. First, we generate a texture by *glGenTextures*. Then bind it to with its type (here is *GL_TEXTURE_2D*) using *glBindTexture.* Next, use *stbi_load* to load the image. After loading the image, we can use *glTexImage2D* to create our texture with the image.

5. Data connection



```
GLuint model_loc = glGetUniformLocation(shaderProgram, "model");
GLuint view_loc = glGetUniformLocation(shaderProgram, "view");
GLuint proj_loc = glGetUniformLocation(shaderProgram, "projection");
GLuint squz_loc = glGetUniformLocation(shaderProgram, "squeezeFactor");

GLuint texture_loc = glGetUniformLocation(shaderProgram, "ourTexture");
GLuint rainbow_loc = glGetUniformLocation(shaderProgram, "rainbowColor");
GLuint use_rainbow_loc = glGetUniformLocation(shaderProgram, "useRainbowColor");
```

After finishing all the above functions, we can start to write our main function. First, we have to know where is the position of the variable in the shader code, so we use *glGetUniformLocation* to get their position. Later when we want to pass value to the shader, we can pass them to their correct position.

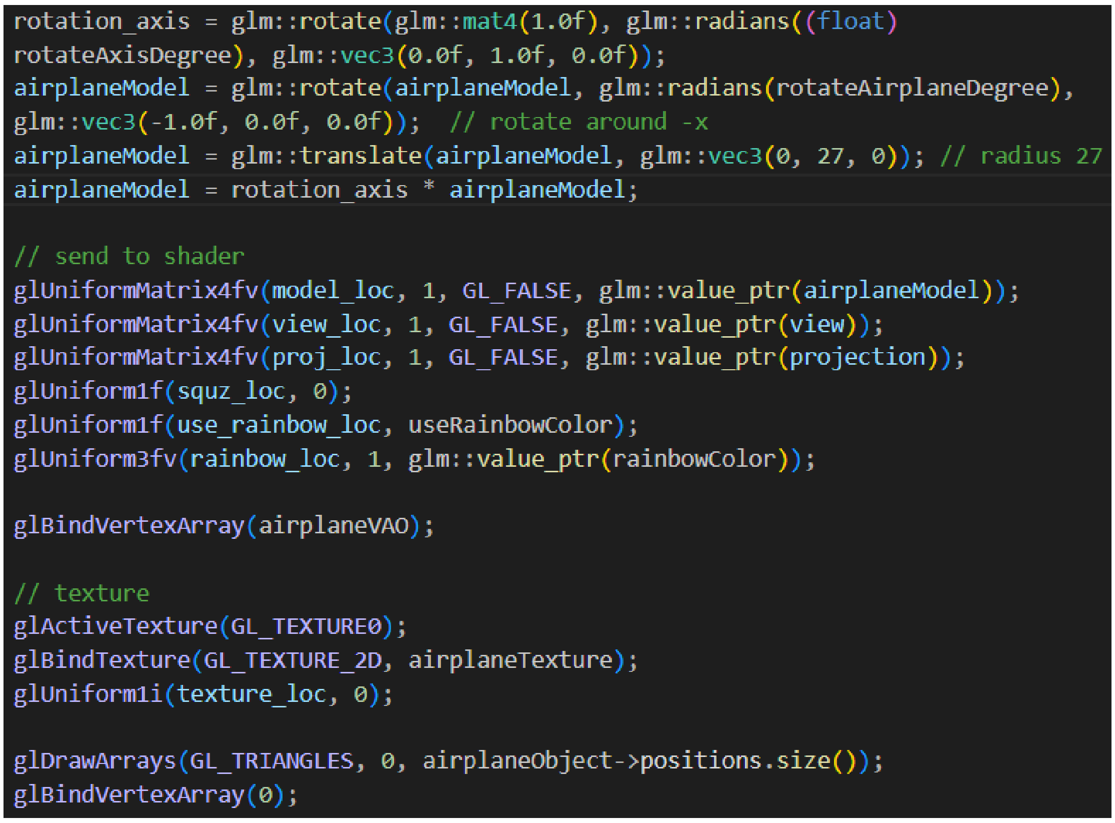
6. Render object

When rendering the object, we calculate its transform matrix and send the variables to the shader.

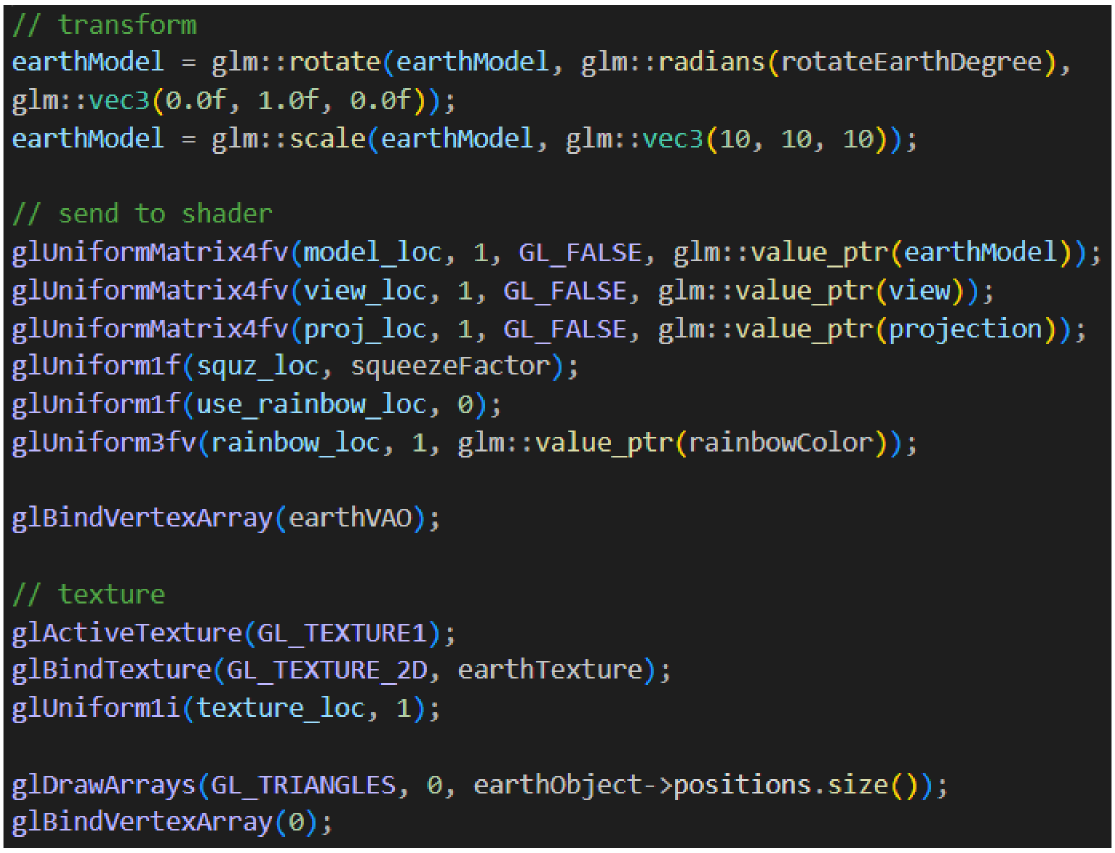
Notice that even if we don't need the variable for the object (but the variable is in shader code), we still have to pass something to it. For example, the *squeezeFactor* is for the earth. The airplane doesn't need to squeeze, but we still pass it to the shader (just give it zero), and the shader code will determine how to use this variable. For example, if the *squeezeFactor* is zero, the vertex shader won't squeeze the object.

When passing values to the shader, different data type has its own function. for example, *glUniformMatrix4fv* is for mat4, *glUniform1f* is for a floating point, and *glUniform3fv* is for vec3. If we use a wrong function, the compiler won't give an error and the program can still run, but that variable cannot be passed to the shader, so it will look strange.

For the transformation matrix in airplane. We define two matrix, *airplaneModel* and *rotation_axis*. *airplaneModel* is for the airplane's rotation and translation, which make the airplane rotate around -x axis and radius is 27. The *rotation_axis* is for the airplane's rotation axis. When we press key 'D' and key 'A', we can rotate the rotation axis of the Airplane +1/-1 degree around Y axis. The final airplane's transformation matrix is *rotation_axis * airplaneModel*.



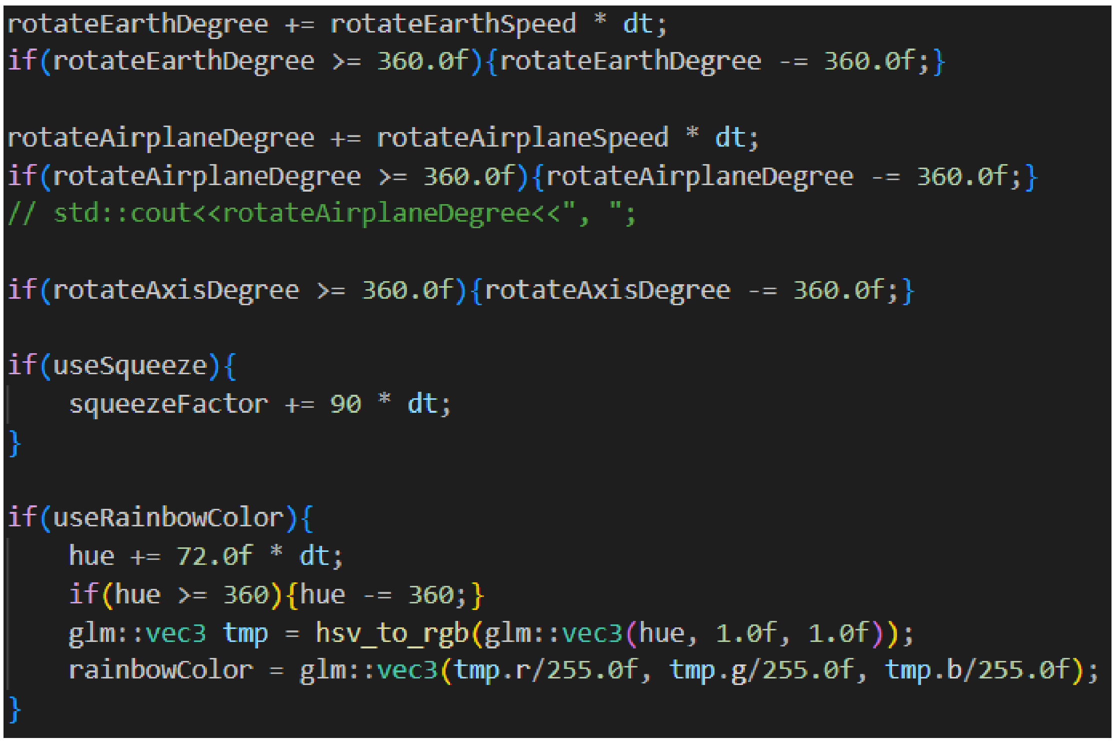
```
rotation_axis = glm::rotate(glm::mat4(1.0f), glm::radians((float)
rotateAxisDegree), glm::vec3(0.0f, 1.0f, 0.0f));
airplaneModel = glm::rotate(airplaneModel, glm::radians(rotateAirplaneDegree),
glm::vec3(-1.0f, 0.0f, 0.0f));  // rotate around -x
airplaneModel = glm::translate(airplaneModel, glm::vec3(0, 27, 0)); // radius 27
airplaneModel = rotation_axis * airplaneModel;

// send to shader
glUniformMatrix4fv(model_loc, 1, GL_FALSE, glm::value_ptr(airplaneModel));
glUniformMatrix4fv(view_loc, 1, GL_FALSE, glm::value_ptr(view));
glUniformMatrix4fv(proj_loc, 1, GL_FALSE, glm::value_ptr(projection));
glUniform1f(squz_loc, 0);
glUniform1f(use_rainbow_loc, useRainbowColor);
glUniform3fv(rainbow_loc, 1, glm::value_ptr(rainbowColor));

glBindVertexArray(airplaneVAO);

// texture
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, airplaneTexture);
glUniform1i(texture_loc, 0);

glDrawArrays(GL_TRIANGLES, 0, airplaneObject->positions.size());
glBindVertexArray(0);
```



```
// transform
earthModel = glm::rotate(earthModel, glm::radians(rotateEarthDegree),
glm::vec3(0.0f, 1.0f, 0.0f));
earthModel = glm::scale(earthModel, glm::vec3(10, 10, 10));

// send to shader
glUniformMatrix4fv(model_loc, 1, GL_FALSE, glm::value_ptr(earthModel));
glUniformMatrix4fv(view_loc, 1, GL_FALSE, glm::value_ptr(view));
glUniformMatrix4fv(proj_loc, 1, GL_FALSE, glm::value_ptr(projection));
glUniform1f(squz_loc, squeezeFactor);
glUniform1f(use_rainbow_loc, 0);
glUniform3fv(rainbow_loc, 1, glm::value_ptr(rainbowColor));

glBindVertexArray(earthVAO);

// texture
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, earthTexture);
glUniform1i(texture_loc, 1);

glDrawArrays(GL_TRIANGLES, 0, earthObject->positions.size());
glBindVertexArray(0);
```

Render airplane (up) and render earth (bottom)

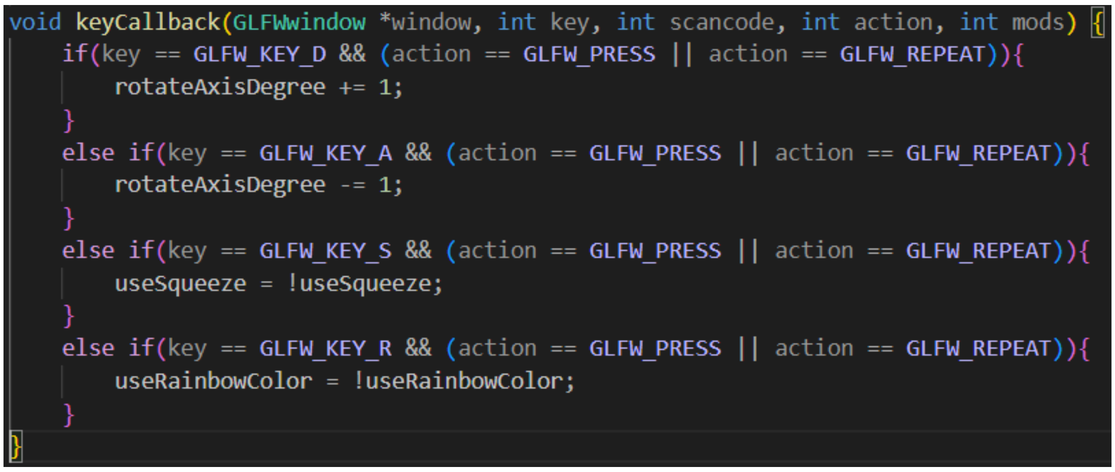
7. Update variables



```
rotateEarthDegree += rotateEarthSpeed * dt;
if(rotateEarthDegree >= 360.0f){rotateEarthDegree -= 360.0f;}

rotateAirplaneDegree += rotateAirplaneSpeed * dt;
if(rotateAirplaneDegree >= 360.0f){rotateAirplaneDegree -= 360.0f;}
// std::cout<<rotateAirplaneDegree<<", ";

if(rotateAxisDegree >= 360.0f){rotateAxisDegree -= 360.0f;}

if(useSqueeze){
    squeezeFactor += 90 * dt;
}

if(useRainbowColor){
    hue += 72.0f * dt;
    if(hue >= 360){hue -= 360;}
    glm::vec3 tmp = hsv_to_rgb(glm::vec3(hue, 1.0f, 1.0f));
    rainbowColor = glm::vec3(tmp.r/255.0f, tmp.g/255.0f, tmp.b/255.0f);
}
```

In the end of the render loop, we have to update the variables. We increase *rotateEarthDegree*, *rotateAirplaneDegree,* and *squeezeFactor* with speed*dt. Check whether rotateAxisDegree is larger than 360 and maintain its value. And calculate the rainbow color. *hsv_to_rgb* is a util function to convert color in HSV to RGB.

8. Key action



```
void keyCallback(GLFWwindow *window, int key, int scancode, int action, int mods) {
    if(key == GLFW_KEY_D && (action == GLFW_PRESS || action == GLFW_REPEAT)){
        rotateAxisDegree += 1;
    }
    else if(key == GLFW_KEY_A && (action == GLFW_PRESS || action == GLFW_REPEAT)){
        rotateAxisDegree -= 1;
    }
    else if(key == GLFW_KEY_S && (action == GLFW_PRESS || action == GLFW_REPEAT)){
        useSqueeze = !useSqueeze;
    }
    else if(key == GLFW_KEY_R && (action == GLFW_PRESS || action == GLFW_REPEAT)){
        useRainbowColor = !useRainbowColor;
    }
}
```

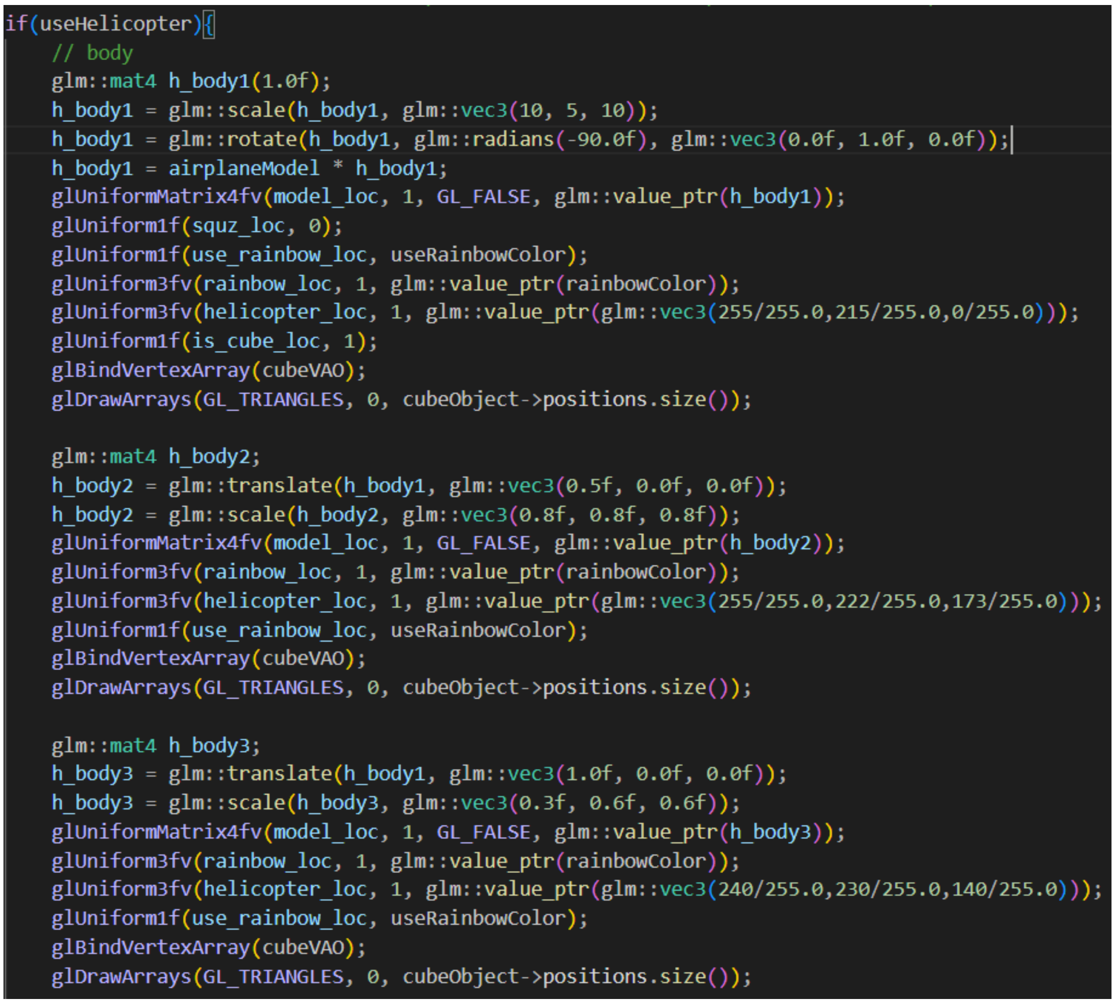
In keyCallback function, we detect the key's action and react. If click on 'D' or 'A', we increase/decrease *rotateAxisDegree* by one. If press 'S' or 'R', switch *useSqueeze* and *useRainbowColor*.

9. Bonus

To replace the helicopter with the airplane, I do the following changes.

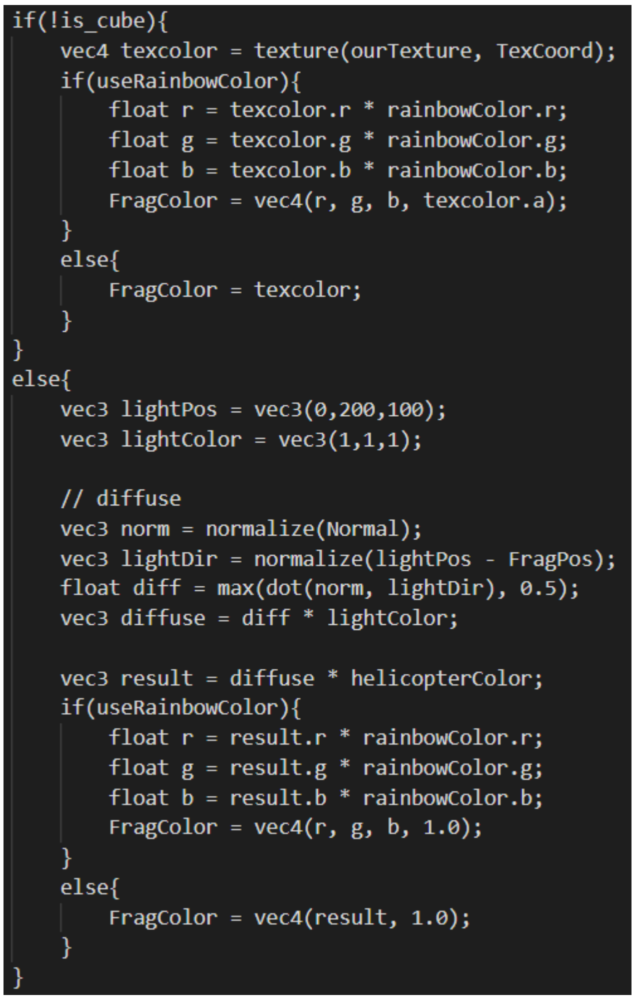
In vertex shader, pass normal, Fragpos, TexCoord to the fragment shader. In fragment shader, I add two uniform variables, helicopterColor and is_cube to handle the helicopter. And then use the normal and Fragpos to calculate the helicopter's color. The color calculation is copy from the shader code of homework 1, and modify *diff* such that the dark side of the helicopter won't be all black and disappear in the background.

In main init(), add cube object and cube VAO because the helicopter is made from the cubes. In the render loop, the transformation matrixes are copied from homework 1. And I did addition rotation to make the helicopter fly in a better way (originally it flies toward its bottom, after rotation, it can fly toward its head). Remove the drawModel function in homework 1 and pass the values to the shader.



```
if(useHelicopter){
    // body
    glm::mat4 h_body1(1.0f);
    h_body1 = glm::scale(h_body1, glm::vec3(10, 5, 10));
    h_body1 = glm::rotate(h_body1, glm::radians(-90.0f), glm::vec3(0.0f, 1.0f, 0.0f));
    h_body1 = airplaneModel * h_body1;
    glUniformMatrix4fv(model_loc, 1, GL_FALSE, glm::value_ptr(h_body1));
    glUniform1f(squz_loc, 0);
    glUniform1f(use_rainbow_loc, useRainbowColor);
    glUniform3fv(rainbow_loc, 1, glm::value_ptr(rainbowColor));
    glUniform3fv(helicopter_loc, 1, glm::value_ptr(glm::vec3(255/255.0,215/255.0,0/255.0)));
    glUniform1f(is_cube_loc, 1);
    glBindVertexArray(cubeVAO);
    glDrawArrays(GL_TRIANGLES, 0, cubeObject->positions.size());

    glm::mat4 h_body2;
    h_body2 = glm::translate(h_body1, glm::vec3(0.5f, 0.0f, 0.0f));
    h_body2 = glm::scale(h_body2, glm::vec3(0.8f, 0.8f, 0.8f));
    glUniformMatrix4fv(model_loc, 1, GL_FALSE, glm::value_ptr(h_body2));
    glUniform3fv(rainbow_loc, 1, glm::value_ptr(rainbowColor));
    glUniform3fv(helicopter_loc, 1, glm::value_ptr(glm::vec3(255/255.0,222/255.0,173/255.0)));
    glUniform1f(use_rainbow_loc, useRainbowColor);
    glBindVertexArray(cubeVAO);
    glDrawArrays(GL_TRIANGLES, 0, cubeObject->positions.size());

    glm::mat4 h_body3;
    h_body3 = glm::translate(h_body1, glm::vec3(1.0f, 0.0f, 0.0f));
    h_body3 = glm::scale(h_body3, glm::vec3(0.3f, 0.6f, 0.6f));
    glUniformMatrix4fv(model_loc, 1, GL_FALSE, glm::value_ptr(h_body3));
    glUniform3fv(rainbow_loc, 1, glm::value_ptr(rainbowColor));
    glUniform3fv(helicopter_loc, 1, glm::value_ptr(glm::vec3(240/255.0,230/255.0,140/255.0)));
    glUniform1f(use_rainbow_loc, useRainbowColor);
    glBindVertexArray(cubeVAO);
    glDrawArrays(GL_TRIANGLES, 0, cubeObject->positions.size());
```

Render loop of (part of) helicopter in main function. The whole helicopter include h_body 1-3, connection, blade 1-4.



```
if(!is_cube){
    vec4 texcolor = texture(ourTexture, TexCoord);
    if(useRainbowColor){
        float r = texcolor.r * rainbowColor.r;
        float g = texcolor.g * rainbowColor.g;
        float b = texcolor.b * rainbowColor.b;
        FragColor = vec4(r, g, b, texcolor.a);
    }
    else{
        FragColor = texcolor;
    }
}
else{
    vec3 lightPos = vec3(0,200,100);
    vec3 lightColor = vec3(1,1,1);

    // diffuse
    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(lightPos - FragPos);
    float diff = max(dot(norm, lightDir), 0.5);
    vec3 diffuse = diff * lightColor;

    vec3 result = diffuse * helicopterColor;
    if(useRainbowColor){
        float r = result.r * rainbowColor.r;
        float g = result.g * rainbowColor.g;
        float b = result.b * rainbowColor.b;
        FragColor = vec4(r, g, b, 1.0);
    }
    else{
        FragColor = vec4(result, 1.0);
    }
}
```

fragment shader main

The upper part in if (!is_cube) is for the airplane and te earth (original fragment shader code for homework 2). The lower part in else is for the cube for helicopter.

10. Problem I met

(1) Not familiar with openGL

I knew the basic concept to use the shaders and VAO, VBO, but I was not familiar with the functions in openGL, especially the parts for texture because there is no sample code in the PPT. So I had to spend some times searching how to achieve my goal. But this also make me become more familiar with those functions and understand what they are doing.

(2) The window showed nothing!

After I finished all the code, I ran the program, but the window showed nothing. It is all black. I checked my code but I couldn't find where I was wrong. I had no idea what I could do or how to debug. I don't even know which part of my code was wrong. I spent two days looking at the black screen and I thought I was really going to go crazy.

I added the error handling in create shader, create program, and open file, but they were all correct. I thought it was the shader code that had something wrong, but we cannot print things in shader code, so I cannot check whether the variables were correctly passed into the shader.

Then I changed the background color. To my surprise, the object had passed to the shader correctly. It was the texture that didn't pass to the shader, so I checked the parts of handling texture, and finally found that it was because I used a wrong function to pass the rainbow color. Instead of using *glUniform3fv* to pass the rainbow color, I used *glUniformMatrix3fv*, no wonder it couldn't pass to the shader. Since the rainbow color wasn't passed to the shader, it had a value 0. When we did $float\, r = texcolor.r * rainbowColor.r$, it will result in 0, which is black.

Why does using a wrong function still allow the program to run??? It doesn't give any error massages!!! I really spent a long time on it :(

(3) Transformation matrix of the airplane

At first, I use sin and cos to calculate the airplane's position after rotation. The airplane is first rotate and then translate to the position calculated by sin and cos. However, this makes the airplane looks strange. After thinking, I realized that its position should not be calculated by sin and cos, it should directly translate to its position (radius 27).

Later when I was doing the key function (the airplane's rotation axis will rotate around y axis when pressing 'A' or 'D'), I didn't know how to do that. My first idea is to use the formula of "rotate around arbitrary axis" in chapter 2, so I read the PPT again to think how to implement it to the airplane. But later, I thought I could use two matrixes to build the transformation matrix for the airplane. One is the original transformation matrix, including rotation and translation. The other one is for the rotation axis. When we press keys to rotate the rotation axis, we apply *glm::rotate* to the rotation axis matrix. And the final airplane transformation would be rotaton_axis_matrix * airplane_transformation_matrix.