Homework 2 - report

> **Name：朱驛庭**
> **Student ID：111550093**

Step 1: Setting Up Shaders

Shaders are small programs written in GLSL that run on the GPU. Typically, a vertex shader processes vertex data, while a fragment shader handles pixel data. The tasks involve:

1. Design vertex shader and fragment shader
2. Create shader object in the main function

Vertex Shader

> `vertexShader.vert`

```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()
{
    // Adjust the vertex position to create a squeeze effect
    vec3 squeezedPos = aPos;
    squeezedPos.y += squeezedPos.z * sin(squeezeFactor) / 2;
    squeezedPos.z += squeezedPos.y * sin(squeezeFactor) / 2;

    // Set gl_Position with the adjusted vertex position
    gl_Position = projection * view * model * vec4(squeezedPos, 1.0);

    // Set TexCoord to aTexCoord
    TexCoord = aTexCoord;
}
```

- Above is the GLSL vertex shader which applies a "squeeze" effect to vertex positions using a sinusoidal adjustment.
- It takes input attributes for
  1. position (`aPos`)
  2. normals (`aNormal`)
  3. texture coordinates (`aTexCoord`)

  and then adjusts the squeezed vertex position (`squeezedPos`) by modifying the `y` and `z` components based on the `squeezeFactor` uniform, which determines the intensity of the effect. The transformed position is then multiplied by the `model`, `view`, and `projection` matrices to compute the final `gl_Position`, ensuring proper 3D transformations.

- Additionally, the texture coordinates (`aTexCoord`) are passed through unchanged to the fragment shader via the `TexCoord` output.

Fragment Shader

> `fragmentShader.frag`

```
#version 330 core
out vec4 FragColor;

in vec2 TexCoord;

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

void main()
{
    // Retrieve the color from the texture at TexCoord
    vec4 color = texture(ourTexture, TexCoord);

    // Calculate the dot product and set FragColor
    if (useRainbowColor) {
        float r = dot(color.r, rainbowColor.r);
        float g = dot(color.g, rainbowColor.g);
        float b = dot(color.b, rainbowColor.b);
        FragColor = vec4(r, g, b, color.a);
    }
    else if (useHelicopter) {
        FragColor = vec4(cubeColor, 1.0);
    }
    else {
        FragColor = color;
    }
}
```

- This fragment shader dynamically sets the final color of a rendered fragment based on texture and input parameters. It samples a texture at the provided `TexCoord` to get the base color.
- If the `useRainbowColor` uniform is true, the shader modifies the texture color by computing the dot product of each RGB component with the corresponding component of the `rainbowColor` uniform, creating a color-tinted effect.
- If `useRainbowColor` is false but `useHelicopter` is true, indicating that the program is rendering helicopter now, then the shader overrides the texture color with the `cubeColor` uniform. `cubeColor` is the (R,G,B) value of given cube color.
- If neither condition is met, the shader uses the original texture color as is. This provides flexibility to adjust fragment coloring dynamically at runtime based on object-specific logic or effects.

TODO#1: `createShader`

> This function reads shader source files (e.g., `.vert` for vertex shaders and `.frag` for fragment shaders) and compiles them.

```cpp
// TODO#1: createShader
unsigned int createShader(const string &filename, const string &type) {
    // Create shader object based on type
    unsigned int shader;
    if (type == "vert")
        shader = glCreateShader(GL_VERTEX_SHADER);
    else if (type == "frag")
        shader = glCreateShader(GL_FRAGMENT_SHADER);

    // Read shader source from file
    string shaderStr;
    ifstream shaderFile(filename);
    stringstream shaderStream;
    shaderStream << shaderFile.rdbuf();
    shaderFile.close();
    shaderStr = shaderStream.str();
    const char* shaderSrc = shaderStr.c_str();

    // Compile shader
    glShaderSource(shader, 1, &shaderSrc, NULL);
    glCompileShader(shader);

    // Check for compile errors
    int success;
    char infoLog[512];
    glGetShaderiv(shader, GL_COMPILE_STATUS, &success);
    if (!success) {
        glGetShaderInfoLog(shader, 512, NULL, infoLog);
        cout << "ERROR::SHADER::COMPILATION_FAILED\n" << infoLog << endl;
    }

    return shader;
}
```

1. **Inputs**:
   - `filename`: Shader source file.
   - `type`: Shader type ("vert" for vertex, "frag" for fragment).
2. Creates a shader object based on type using `glCreateShader`.
3. Reads shader code from the file into a `const char*`.
4. Uses `glShaderSource` and `glCompileShader` to compile the shader.
5. **Error Check**:
   - Checks compilation status with `glGetShaderiv`.
   - Prints error details if compilation fails.
6. Returns the compiled shader ID.

TODO#2: `createProgram`

> This function links the vertex and fragment shaders into a single program.

```cpp
// TODO#2: createProgram
unsigned int createProgram(unsigned int vertexShader, unsigned int fragmentShader) {
    // Create program object
    GLuint program = glCreateProgram();

    // Attach shaders
    glAttachShader(program, vertexShader);
    glAttachShader(program, fragmentShader);

    // Link program
    glLinkProgram(program);

    // Check for linking errors
    int success;
    char infoLog[512];
    glGetProgramiv(program, GL_LINK_STATUS, &success);
    if (!success) {
        glGetProgramInfoLog(program, 512, NULL, infoLog);
        cout << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" << infoLog << endl;
    }

    // Detach shaders after linking
    glDetachShader(program, vertexShader);
    glDetachShader(program, fragmentShader);

    return program;
}
```

1. **Inputs**:
    - `vertexShader` and `fragmentShader`: Compiled shader IDs.
2. Generates a shader program with `glCreateProgram`.
3. Attaches the vertex and fragment shaders using `glAttachShader`.
4. Links the attached shaders using `glLinkProgram`.
5. **Error Check**:
    - Verifies the linking status with `glGetProgramiv`.
    - If linking fails, retrieves and prints error logs using `glGetProgramInfoLog`.
6. Detaches shaders from the program using `glDetachShader`.
7. Returns the linked shader program ID.

Step 2: Setting Up VAO and VBO

TODO#3: `modelVAO`

> Vertex Array Objects (VAOs) store the vertex attribute configuration, while Vertex Buffer Objects (VBOs) hold the vertex data.

```cpp
// TODO#3: modelVAO
unsigned int modelVAO(Object &model) {
    // Generate and bind VAO
    GLuint VAO;
    glGenVertexArrays(1, &VAO);
    glBindVertexArray(VAO);

    // Generate and bind VBOs
    GLuint VBO[3]; // One VBO for positions, normals, and texcoords
    glGenBuffers(3, VBO);

    // Bind and set VBO for positions
    glBindBuffer(GL_ARRAY_BUFFER, VBO[0]);
    glBufferData(GL_ARRAY_BUFFER, model.positions.size() * sizeof(float), model.positions.data(), GL_STATIC_DRAW);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0); // Position attribute
    glEnableVertexAttribArray(0);

    // Bind and set VBO for normals
    glBindBuffer(GL_ARRAY_BUFFER, VBO[1]);
    glBufferData(GL_ARRAY_BUFFER, model.normals.size() * sizeof(float), model.normals.data(), GL_STATIC_DRAW);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, (void*)0); // Normal attribute
    glEnableVertexAttribArray(1);

    // Bind and set VBO for texcoords
    glBindBuffer(GL_ARRAY_BUFFER, VBO[2]);
    glBufferData(GL_ARRAY_BUFFER, model.texcoords.size() * sizeof(float), model.texcoords.data(), GL_STATIC_DRAW);
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0, (void*)0); // Texture coordinate attribute
    glEnableVertexAttribArray(2);

    return VAO;
}
```

1. **Inputs**:
   - `model`: Object containing vertex data (`positions`, `normals`, `texcoords`).
2. **Generate and Bind VAO**:
   - Creates a Vertex Array Object (VAO) with `glGenVertexArrays`.
   - Binds the VAO using `glBindVertexArray`.
3. **Generate VBOs**:

   Creates a array of three Vertex Buffer Objects (VBOs) for **positions**, **normals**, and **texture coordinates** with `glGenBuffers`.
4. **Position Data, Normal Data, Texture Coordinate Setup**:
   - Binds the corresponding VBO.
   - Uploads vertex data to the buffer using `glBufferData`.
   - Specifies the layout of the position data using `glVertexAttribPointer`.
   - Enables the vertex attribute array.
5. Returns the VAO ID for later use.

Step 3: Loading Textures

TODO#4: `loadTexture`

> Textures add visual details to 3D objects. This function uses `stb_image` to load and bind textures.

```cpp
// TODO#4: loadTexture
unsigned int loadTexture(const string &filename) {
    // Enable texture 2D
    glEnable(GL_TEXTURE_2D);

    // Generate texture ID
    GLuint textureID;
    glGenTextures(1, &textureID);

    // Load image data using stb_image
    GLint width, height, nrChannels;
    unsigned char *data = stbi_load(filename.c_str(), &width, &height, &nrChannels, 0);

    if (data) {
        // Flip the image vertically
        for (int i = 0; i < height / 2; i++) {
            for (int j = 0; j < width * nrChannels; j++) {
                std::swap(data[i * width * nrChannels + j], data[(height - i - 1) * width * nrChannels + j]);
            }
        }

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

        // Bind texture
        glBindTexture(GL_TEXTURE_2D, textureID);

        // Set texture parameters
        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);

        // Generate texture and mipmap
        glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0, format, GL_UNSIGNED_BYTE, data);
        glGenerateMipmap(GL_TEXTURE_2D);

        // Free image data
        stbi_image_free(data);
    } else {
        std::cout << "Failed to load texture: " << filename << std::endl;
        stbi_image_free(data);
    }

    return textureID;
}
```

1. **Inputs**:
    - `filename`: Path to the image file for the texture.
2. **Enable and Generate Texture**:
    - Enables 2D textures using `glEnable(GL_TEXTURE_2D)`.
    - Creates a texture ID with `glGenTextures`.
3. **Load Image**

    Uses `stbi_load` to load the image file into memory and retrieve its width, height, and number of channels.
4. **Flip Image Vertically**

    If the image is successfully loaded, it flips it vertically to match OpenGL's coordinate system.
5. **Determine Image Format**

    Sets the texture format (`GL_RED`, `GL_RGB`, or `GL_RGBA`) based on the number of channels in the image.
6. **Bind and Configure Texture**
    - Binds the texture using `glBindTexture`.
    - Sets texture wrapping (`GL_REPEAT`) and filtering (`GL_LINEAR` for min/mag filters) parameters with `glTexParameteri`.
7. **Generate Texture**
    - Uploads texture data to OpenGL using `glTexImage2D`.

- Generates mipmaps with `glGenerateMipmap`.

8. **Cleanup**
   - Frees the image data memory using `stbi_image_free`.
   - Logs a failure message if the image fails to load.
9. **Return:**

   Returns the texture ID for use in rendering.

Step 4: Uniform Variable Connections

TODO#5

> Use `glGetUniformLocation` to retrieve shader uniform locations and set values.

```
// Vertex Shader: model, view, projection, squeezeFactor
GLuint modelLoc = glGetUniformLocation(shaderProgram, "model");
GLuint viewLoc = glGetUniformLocation(shaderProgram, "view");
GLuint projectionLoc = glGetUniformLocation(shaderProgram, "projection");
GLuint squeezeFactorLoc = glGetUniformLocation(shaderProgram, "squeezeFactor");

// Fragment Shader: ourTexture, rainbowColor, useRainbowColor
GLuint ourTextureLoc = glGetUniformLocation(shaderProgram, "ourTexture");
GLuint rainbowColorLoc = glGetUniformLocation(shaderProgram, "rainbowColor");
GLuint useRainbowColorLoc = glGetUniformLocation(shaderProgram, "useRainbowColor");
GLuint cubeColorLoc = glGetUniformLocation(shaderProgram, "cubeColor");
GLuint useHelicopterLoc = glGetUniformLocation(shaderProgram, "useHelicopter");
```

- **Vertex Shader Uniforms**:

  Retrieves uniform locations in the vertex shader using `glGetUniformLocation`:
  - `modelLoc`: For the "model" matrix.
  - `viewLoc`: For the "view" matrix.
  - `projectionLoc`: For the "projection" matrix.
  - `squeezeFactorLoc`: For a custom "squeezeFactor" variable.
- **Fragment Shader Uniforms**:

  Retrieves uniform locations in the fragment shader using `glGetUniformLocation`:
  - `ourTextureLoc`: For the texture sampler "ourTexture".
  - `rainbowColorLoc`: For the "rainbowColor" variable.
  - `useRainbowColorLoc`: For the boolean "useRainbowColor".
  - `cubeColorLoc`: For the "cubeColor" variable.
  - `useHelicopterLoc`: For the boolean "useHelicopter".

Step 5: Rendering Objects

TODO#6-1, TODO#6-2, TODO#Bonus

> These tasks involve rendering objects like airplanes and the earth. The rendering code binds VAOs and textures, sets uniforms, and issues draw calls.

Airplane Model Transformation

```
// TODO#6-1: Render Airplane
// First rotate around Y axis for orbit path (controlled by A/D keys)
airplaneModel = glm::rotate(airplaneModel, glm::radians(rotateAxisDegree), glm::vec3(0.0f,
1.0f, 0.0f));

// Translate to orbit position (radius 27)
float orbitY = 27.0f * sin(glm::radians(rotateAirplaneDegree));
float orbitZ = 27.0f * cos(glm::radians(rotateAirplaneDegree));
airplaneModel = glm::translate(airplaneModel, glm::vec3(0.0f, orbitY, orbitZ));

// Rotate around X axis for airplane rotation (controlled by rotateAirplaneSpeed)
airplaneModel = glm::rotate(airplaneModel, glm::radians(rotateAirplaneDegree - 90), glm::vec
3(-1.0f, 0.0f, 0.0f));
```

1. **Orbit Rotation (Y-axis)**:

   Rotates the airplane model around the Y-axis using `glm::rotate` based on `rotateAxisDegree` (controlled by A/D keys).

2. **Translate to Orbit Path**:

   - Computes orbit positions (`orbitY`, `orbitZ`) with a radius of 27 using sine and cosine.
   - Translates the airplane to the computed position using `glm::translate`.

3. **Airplane Rotation (X-axis)**:

   Rotates the airplane around its own X-axis using `glm::rotate` with `rotateAirplaneDegree - 90` based on `rotateAirplaneSpeed`.

Helicopter Model Transformation

```
// TODO#Bonus: Render Helicopter
//* Body
helicopterModel = glm::rotate(helicopterModel, glm::radians(rotateAxisDegree), glm::vec3(0.0f, 1.0f, 0.0f));
helicopterModel = glm::translate(helicopterModel, glm::vec3(0.0f, orbitY, orbitZ));
helicopterModel = glm::rotate(helicopterModel, glm::radians(rotateAirplaneDegree - 90), glm::vec3(-1.0f, 0.0f, 0.0f));
helicopterModel = glm::rotate(helicopterModel, glm::radians(90.0f), glm::vec3(0.0f, -1.0f, 0.0f));
glm::vec3 block1_pos = glm::vec3(0.0f, 0.0f, 0.0f);
glm::vec3 block2_pos = block1_pos + glm::vec3(8.0f, 0.0f, 0.0f);
glm::vec3 block3_pos = block2_pos + glm::vec3(4.5f, 0.0f, 0.0f);
glm::mat4 block1 = glm::translate(helicopterModel, block1_pos);
block1 = glm::scale(block1, glm::vec3(10.0f, 8.0f, 10.0f));
glm::mat4 block2 = glm::translate(helicopterModel, block2_pos);
block2 = glm::scale(block2, glm::vec3(6.0f, 7.5f, 8.0f));
glm::mat4 block3 = glm::translate(helicopterModel, block3_pos);
block3 = glm::scale(block3, glm::vec3(3.0f, 6.0f, 5.0f));

// * Connector
glm::vec3 connector_pos = block1_pos + glm::vec3(0.0f, 5.5f, 0.0f);
glm::mat4 connector = glm::translate(helicopterModel, connector_pos);
connector = glm::rotate(connector, glm::radians(rotateConnectorDegree), glm::vec3(0.0f, 1.0f, 0.0f));
connector = glm::scale(connector, glm::vec3(3.0f, 3.0f, 3.0f));

// * Rotor Blades
float bladeLength = 4.0f; // Adjust this value to control the length of the blades
float bladeHeight = 0.1f;
float bladeWidth = 0.5f;  // Adjust this value to control the width of the blades
vector<glm::vec3> blade_colors = {glm::vec3(250, 247, 240), glm::vec3(216, 210, 194), glm::vec3(177, 116, 87), glm::vec3(74, 73, 71)};
vector<glm::mat4> blades;
for (int i = 0; i < 4; ++i) {
    glm::mat4 blade = glm::rotate(connector, glm::radians(90.0f * i), glm::vec3(0.0f, 1.0f, 0.0f));
    blade = glm::translate(blade, glm::vec3(bladeLength / 2.0f, 0.0f, 0.0f)); // Translate to position
    blade = glm::scale(blade, glm::vec3(bladeLength, bladeHeight, bladeWidth)); // Scale the blade
    blades.push_back(blade);
}
```

1. **Helicopter Body**:

   - **Global Rotation**:

     Rotates the helicopter model around the Y-axis (`rotateAxisDegree`).

   - **Translation**:

     Translates the helicopter to an orbit path determined by `orbitY` and `orbitZ`.

   - **Local Rotations**:

     Applies X-axis rotations for the airplane-like motion.

   - **Translating and Scaling**:

     Translates and scales individual blocks to define the shape of the helicopter.

2. **Connector**:
    - **Connector Placement**:

      Positions the connector relative to `block1`.

    - **Rotation**:

      Rotates the connector around its local axis (`rotateConnectorDegree`).

    - **Scaling**:

      Scales the connector for its appearance.

3. **Rotor Blades**:
    - **Blade Dimensions**:

      Defines blade length, width, and height for customization.

    - **Color Assignment**:

      Prepares different colors for the rotor blades.

    - **Blade Placement**:

      Iterates to create four rotor blades:

        - Rotates each blade around the connector.
        - Translates and scales each blade to its appropriate position.
        - Adds the blade to the `blades` list for rendering.

Conditional Rendering

```
if (useHelicopter) {
    drawModel("cube", block1, view, projection, 255, 241, 0);
    drawModel("cube", block2, view, projection, 0, 0, 0);
    drawModel("cube", block3, view, projection, 177, 116, 87);
    drawModel("cube", connector, view, projection, 255, 255, 255);
    drawModel("cube", connector, view, projection, 255, 255, 255);
    for (int i = 0; i < 4; ++i) {
        drawModel("cube", blades[i], view, projection, blade_colors[i][0], blade_colors[i][1], blade_colors[i][2]);
    }
}
else {
    // Draw the airplane
    drawModel("airplane", airplaneModel, view, projection, 1.0f, 1.0f, 1.0f);
}
```

1. **Render Helicopter** (if `useHelicopter` is true):
    - Draws the main helicopter components (`block1` , `block2` , `block3` , and `connector`) using `drawModel` with specific colors for each block.
    - Loops through the rotor blades (`blades`) to render each one with its corresponding color from `blade_colors` .
2. **Render Airplane** (if `useHelicopter` is false):
    - Draws the airplane model with pre-loaded texture.

Earth Model Transformation & Render

```
// TODO#6-2: Render Earth
// Set up earth model matrix
earthModel = glm::scale(earthModel, glm::vec3(10.0f));  // Scale 10x
earthModel = glm::rotate(earthModel, glm::radians(rotateEarthDegree), glm::vec3(0.0f, 1.0f, 0.0f));  // Rotate around Y axis

// Draw the earth
drawModel("earth", earthModel, view, projection, 1.0f, 1.0f, 1.0f);
```

1. **Earth Model Matrix Setup**:
    - **Scaling**:

      Scales the earth model by a factor of 10 using `glm::scale` for larger rendering.

- **Rotation**:

  Rotates the earth around the Y-axis (`rotateEarthDegree`) using `glm::rotate`.

2. **Rendering the Earth**:

   Draws the earth model with the calculated transformation matrix (`earthModel`) and applies `view` and `projection` matrices.

Status Update and Animation Logic

```
// Status update
currentTime = glfwGetTime();
dt = currentTime - lastTime;
lastTime = currentTime;

/* TODO#7: Update "rotateEarthDegree", "rotateAirplaneDegree", "rotateAxisDegree",
    *          "squeezeFactor", "rainbowColor"
    */

// Update rotation angles based on elapsed time (dt is in seconds)
rotateEarthDegree = fmod(rotateEarthDegree + rotateEarthSpeed * dt, 360.0f);  // 30 degrees/sec
rotateAirplaneDegree = fmod(rotateAirplaneDegree + rotateAirplaneSpeed * dt, 360.0f);  // 90 degrees/sec
rotateConnectorDegree = fmod(rotateConnectorDegree + rotateConnectorSpeed, 360.0f);  // 90 degrees/sec

// Update squeeze factor if squeezing is enabled
if (useSqueeze) {
    squeezeFactor = fmod(squeezeFactor + 90 * dt, 360.0f);  // 90 degrees/sec
}

// Update rainbow color if rainbow mode is enabled
if (useRainbowColor) {
    rainbowDegree = fmod(rainbowDegree + rainbowSpeed * dt, 360.0f);  // 72 degrees/sec

    // Convert HSV to RGB
    float hue = rainbowDegree;
    float saturation = 1.0f;
    float value = 1.0f;

    float c = value * saturation;
    float x = c * (1 - abs(fmod(hue / 60.0f, 2.0f) - 1));
    float m = value - c;

    if (hue < 60) rainbowColor = glm::vec3(c, x, 0);
    else if (hue < 120) rainbowColor = glm::vec3(x, c, 0);
    else if (hue < 180) rainbowColor = glm::vec3(0, c, x);
    else if (hue < 240) rainbowColor = glm::vec3(0, x, c);
    else if (hue < 300) rainbowColor = glm::vec3(x, 0, c);
    else rainbowColor = glm::vec3(c, 0, x);

    rainbowColor += glm::vec3(m);
}
```

1. Time Update:

Tracks time using `glfwGetTime`:

- `currentTime` is the current frame's time.
- `dt` (delta time) is the time difference between frames for smooth animation.
- Updates `lastTime` for the next frame.

2. Rotation Updates:

- **Earth Rotation (`rotateEarthDegree`)**:
  - Updates the Earth's rotation at a rate of 30 degrees per second.
- **Airplane Rotation (`rotateAirplaneDegree`)**:
  - Updates airplane's rotation around its axis at 90 degrees per second.
- **Connector Rotation (`rotateConnectorDegree`)**:
  - Updates the connector's rotation at 90 degrees per second.
- Uses `fmod` to wrap rotation angles to a 0–360 range.

3. Squeeze Effect:

If `useSqueeze` is enabled:

- Updates the `squeezeFactor` to oscillate smoothly at 90 degrees per second.

4. Rainbow Color Update:

- If `useRainbowColor` is enabled:
  - Updates the `rainbowDegree` hue at 72 degrees per second.
- Converts `rainbowDegree` from HSV to RGB:
  - Uses the HSV to RGB formula to determine the color based on hue, saturation, and value.
  - Computes RGB values in segments (0–60, 60–120, etc.) for smooth transitions.

Helper function: `drawModel()`

```
// Send transformation matrices to vertex shader
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::value_ptr(view));
glUniformMatrix4fv(projectionLoc, 1, GL_FALSE, glm::value_ptr(projection));
if (modelName == "earth") {
    glUniform1f(squeezeFactorLoc, glm::radians(squeezeFactor));
}
else {
    glUniform1f(squeezeFactorLoc, 0.0f);
}

// Send uniforms to fragment shader
if (modelName == "airplane") {
    glUniform1i(ourTextureLoc, 0);
    glUniform3fv(rainbowColorLoc, 1, glm::value_ptr(rainbowColor));
    glUniform1i(useRainbowColorLoc, useRainbowColor);
    glUniform3fv(cubeColorLoc, 1, glm::value_ptr(glm::vec3(0.0f, 0.0f, 0.0f))); // No color for airplane
    glUniform1i(useHelicopterLoc, useHelicopter);
}
else if (modelName == "earth") {
    glUniform1i(ourTextureLoc, 0);
    glUniform3fv(rainbowColorLoc, 1, glm::value_ptr(glm::vec3(1.0f)));  // No rainbow effect for earth
    glUniform1i(useRainbowColorLoc, false);  // Disable rainbow effect
    glUniform3fv(cubeColorLoc, 1, glm::value_ptr(glm::vec3(0.0f, 0.0f, 0.0f)));
    glUniform1i(useHelicopterLoc, false); // Disable helicopter effect
}
else if (modelName == "cube") {
    glUniform1i(ourTextureLoc, 0);
    glUniform3fv(rainbowColorLoc, 1, glm::value_ptr(glm::vec3(1.0f)));  // No rainbow effect for cube
    glUniform1i(useRainbowColorLoc, false);  // Disable rainbow effect
    glUniform3fv(cubeColorLoc, 1, glm::value_ptr(glm::vec3(r / 255.0f, g / 255.0f, b / 255.0f)));
    glUniform1i(useHelicopterLoc, useHelicopter);
}

// Bind texture and VAO and draw
if (modelName == "airplane") {
    glActiveTexture(GL_TEXTURE0); // Activate texture unit 0
    glBindTexture(GL_TEXTURE_2D, airplaneTexture); // Bind the texture to unit 0
    glBindVertexArray(airplaneVAO);
    glDrawArrays(GL_TRIANGLES, 0, airplaneObject->positions.size() / 3);
}
else if (modelName == "earth") {
    glActiveTexture(GL_TEXTURE0); // Activate texture unit 0
    glBindTexture(GL_TEXTURE_2D, earthTexture); // Bind the texture to unit 0
    glBindVertexArray(earthVAO);
    glDrawArrays(GL_TRIANGLES, 0, earthObject->positions.size() / 3);
}
else if (modelName == "cube") {
    glBindVertexArray(cubeVAO);
    glDrawArrays(GL_TRIANGLES, 0, cubeObject->positions.size() / 3);
}
```

1. **Vertex Shader Uniform Setup**
   - Sends transformation matrices (`model`, `view`, `projection`) to the vertex shader using `glUniformMatrix4fv`.
   - Handles the `squeezeFactor` for scaling:
     - Enables or disables it based on the `modelName`.

2. Fragment Shader Uniform Setup

Configures fragment shader uniforms based on `modelName`:

- **"airplane"**:

  - Sets up uniforms like `ourTextureLoc`, `rainbowColor`, and `useHelicopter`.
- **"earth"**:
  - Disables effects like `rainbowColor` and `useHelicopter`.
  - Assigns default color values.
- **"cube"**:
  - Sets no texture (`ourTextureLoc = 0`) and disables effects (`rainbowColor` and `useHelicopter`).
  - Uses a custom color for the cube.

3. Binding and Drawing

- Depending on `modelName`, binds the appropriate VAO and texture:
  - Activates texture unit 0 for textured models (`airplane` and `earth`).
- Draws the model using `glDrawArrays`:
  - Mode: `GL_TRIANGLES`.
  - Vertex count: Derived from the number of positions.

Step 6: Interaction

TODO#8

> Implement key callbacks for dynamic interactions like rotation or camera control.

```cpp
// TODO#8: Key callback
void keyCallback(GLFWwindow *window, int key, int scancode, int action, int mods) {
    // Only handle GLFW_PRESS and GLFW_REPEAT actions
    if (action != GLFW_PRESS && action != GLFW_REPEAT)
        return;

    switch (key) {
        case GLFW_KEY_D:
            // Increase rotation axis by 1 degree
            rotateAxisDegree += 1.0f;
            break;

        case GLFW_KEY_A:
            // Decrease rotation axis by 1 degree
            rotateAxisDegree -= 1.0f;
            break;

        case GLFW_KEY_S:
            // Toggle squeeze effect
            useSqueeze = !useSqueeze;
            break;

        case GLFW_KEY_H:
            // Toggle helicopter effect
            useHelicopter = !useHelicopter;
            break;

        case GLFW_KEY_R:
            // Toggle rainbow color mode
            useRainbowColor = !useRainbowColor;
            if (!useRainbowColor) {
                // Reset rainbow color when turning off
                rainbowColor = glm::vec3(1.0f);
                rainbowDegree = 0.0f;
            }
            break;

        case GLFW_KEY_ESCAPE:
            // Close the window when ESC is pressed
            glfwSetWindowShouldClose(window, true);
            break;
    }
}
```

- `GLFW_KEY_D`:
  - Increases `rotateAxisDegree` by 1.0f, rotating the axis clockwise.
- `GLFW_KEY_A`:
  - Decreases `rotateAxisDegree` by 1.0f, rotating the axis counterclockwise.
- `GLFW_KEY_S`:
  - Toggles the `useSqueeze` effect on or off.
- `GLFW_KEY_H`:
  - Toggles the helicopter effect (`useHelicopter`).
- `GLFW_KEY_R`:
  - Toggles the rainbow color mode (`useRainbowColor`):
    - Resets the rainbow color and degree when turned off.
- `GLFW_KEY_ESCAPE`:
  - Closes the window by calling `glfwSetWindowShouldClose`.