ICG HW2 Report

I. Create program

(in main:)

Call the custom function to create a vertex shader and a fragment shader, and attach them to the shader program. Then actives the shader program before rendering.

```
unsigned int vertexShader, fragmentShader, shaderProgram;
vertexShader = createShader("vertexShader.vert", "vert");
fragmentShader = createShader("fragmentShader.frag", "frag");
shaderProgram = createProgram(vertexShader, fragmentShader);
glUseProgram(shaderProgram);
```

(in functions:)

The ‘createShader’ function returns a compiled shader of the specified type.

```
unsigned int createShader(const string& filename, const string& type)
{
    unsigned int shader;

    // Create either vertex shader or fragment shader
    if (type == "vert") shader = glCreateShader(GL_VERTEX_SHADER);
    else shader = glCreateShader(GL_FRAGMENT_SHADER);

    // Set the source code in the shader
    ifstream fs(filename);
    stringstream ss;
    string s;
    while (getline(fs, s))
    {
        ss << s << "\n";
    }
    string temp = ss.str();
    const char* source = temp.c_str();
    glShaderSource(shader, 1, &source, NULL);

    // Compile the shader
    glCompileShader(shader);

    return shader;
}
```

The ‘createProgram’ function returns a linked program that is attached with the specified vertex shader and fragment shader.

```
unsigned int createProgram(unsigned int vertexShader, unsigned int fragmentShader)
{
    // Create a program object
    unsigned int program = glCreateProgram();

    // Attach the shader object to the program
    glAttachShader(program, vertexShader);
    glAttachShader(program, fragmentShader);

    // Link this program
    glLinkProgram(program);

    // Detatch the shader object from the program
    glDetachShader(program, vertexShader);
    glDetachShader(program, fragmentShader);

    return program;
}
```

II. Create VAO, VBO, and bind VAO to the object

(in main:)

Call the custom function to set up VAO for penguin and board objects.

```
unsigned int penguinVAO, boardVAO;
penguinVAO = modelVAO(penguinModel);
boardVAO = modelVAO(boardModel);
```

(in function:)

The ‘modelVAO’ function returns a VAO. The processes include: Generate a vertex array object, VAO, bind it and set up its corresponding VBO. Generate three vertex buffer objects, VBOs, each for position, normal, and texture coordinates. Bind them to the target buffer, copy data, and link with the vertex shader input.

```
unsigned int modelVAO(Object& model)
{
    unsigned int VAO, VBO[3];

    // Generate vertex array object
    glGenVertexArrays(1, &VAO);
    // Bind the vertex array object
    glBindVertexArray(VAO);

    // Generate three vertex buffer objects
    glGenBuffers(3, VBO);

    // Bind the target buffer
    glBindBuffer(GL_ARRAY_BUFFER, VBO[0]);
    // Copy the data into the target buffer
    glBufferData(GL_ARRAY_BUFFER, sizeof(GL_FLOAT) * (model.positions.size()), &(model.positions[0]), GL_STATIC_DRAW);
    // Link the vertex buffer with the vertex shader input
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(GL_FLOAT) * 3, 0);
    glEnableVertexAttribArray(0);
    // Unbind the target buffer
    glBindBuffer(GL_ARRAY_BUFFER, 0);

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

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

    // Unbind the array object
    glBindVertexArray(0);

    return VAO;
```

(in GLSL:)

In vertex shader, use layout qualifiers to specify the layout of input attributes, namely position, normal, and texture coordinates.

```
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoord;
```

III. Data connection

(in main:)

Retrieve locations for uniform variables, i.e., model, view, projection matrix, and other factors like squeezeFactor, timer, etc.

```
// Retrieve locations for model, view, and projection matrices.
unsigned int mLoc, vLoc, pLoc;
mLoc = glGetUniformLocation(shaderProgram, "M");
vLoc = glGetUniformLocation(shaderProgram, "V");
pLoc = glGetUniformLocation(shaderProgram, "P");
// Obtain locations for squeezeFactor, grayscale, and texture.
unsigned int sfLoc, gsLoc, texLoc;
sfLoc = glGetUniformLocation(shaderProgram, "squeezeFactor");
gsLoc = glGetUniformLocation(shaderProgram, "useGrayscale");
texLoc = glGetUniformLocation(shaderProgram, "ourTexture");
// Obtain location for timer.
unsigned int tLoc;
tLoc = glGetUniformLocation(shaderProgram, "timer");
```

Connect the retrieved locations by setting the values of uniform variables.

```
// Connect locations for board, view, and projection matrices
glUniformMatrix4fv(mLoc, 1, GL_FALSE, value_ptr(board));
glUniformMatrix4fv(vLoc, 1, GL_FALSE, value_ptr(view));
glUniformMatrix4fv(pLoc, 1, GL_FALSE, value_ptr(perspective));
// Connect locations for squeezeFactor, grayscale, and texture
glUniform1f(sfLoc, 0);
glUniform1i(gsLoc, useGrayscale);
glUniform1i(texLoc, 0);
// Connect location for timer
glUniform1f(tLoc, flashFactor);
```

(in GLSL:)

There are four uniform variables in the vertex shader, which are model, view, projection matrix, and squeezeFactor.

```
uniform mat4 M;
uniform mat4 V;
uniform mat4 P;

uniform float squeezeFactor;
```

There are three uniform variables in the fragment shader, which are texture, grayscale, and timer.

```
uniform sampler2D ourTexture;
//uniform float lighting;
uniform bool useGrayscale;
uniform float timer; // 添加時間變量
```

IV. Rendering

(in openGL:)

Bind the object to its VAO and render the primitives.

```
// Bind boardVAO and render primitives
glBindVertexArray(boardVAO);
glDrawArrays(GL_TRIANGLES, 0, boardModel.positions.size());
glBindVertexArray(0);
```

(in GLSL:)

Vertex shader is responsible for transforming the position of each vertex. In the main function, adjust the vertex position and update the normal. Finally, pass the normal and texture coordinates to fragment shader.

```
out vec2 texCoord;
out vec3 normal;

vec4 worldPos;

void main()
{
    // TODO: Implement squeeze effect and update normal transformation

    vec3 squeezedPos = aPos;
    // Adjust the vertex position to create a squeeze effect
    squeezedPos.y += aPos.z * sin(squeezeFactor) / 2.0;
    squeezedPos.z += aPos.y * sin(squeezeFactor) / 2.0;

    // Update worldPos
    worldPos = M * vec4(squeezedPos, 1.0);

    // Calculate the final gl_Position
    gl_Position = P * V * worldPos;

    // Update the normal transformation
    normal = mat3(transpose(inverse(M))) * aNormal;

    texCoord = aTexCoord;
}
```

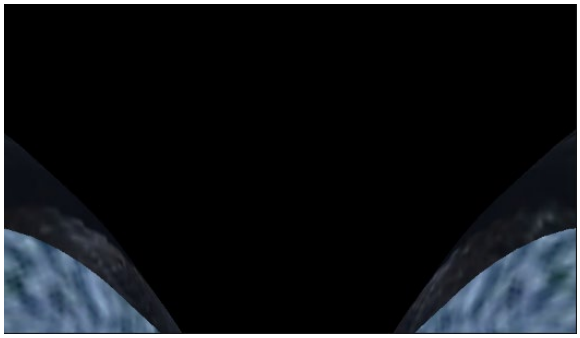
Fragment shader is responsible for determining the color of each potential pixel. It takes in normal and texture coordinates from vertex shader, computes the final color and returns it.

```
in vec2 texCoord;
in vec3 normal;

out vec4 FragColor;

void main()
{
    // TODO: Implement Grayscale Effect
    //   1. Retrieve the color from the texture at texCoord.
    //   2. If useGrayscale is true,
    //       a. Calculate the grayscale value using the dot product with the luminance weights(0.299, 0.587, 0.114).
    //       b. Set FragColor to a grayscale version of the color.
    //   Note: Ensure FragColor is appropriately set for both grayscale and color cases.

    // Retrieve the color from the texture at texCoord.
    vec4 color = texture2D(ourTexture, texCoord);

    if(useGrayscale)
    {
        // Calculate the grayscale value and set FragColor to the grayscale version
        float grayscale = dot(color.rgb, vec3(0.299, 0.587, 0.114));
        FragColor = vec4(grayscale, grayscale, grayscale, color.a);
    }
    else
    {
        // Compute the color variation based on time
        vec3 timeColor = vec3(sin(timer)+1.0, 1.0, cos(timer));

        // Apply the time-based color variation
        FragColor = vec4(color.rgb * timeColor, color.a);
    }
}
```

Problems & Solution

1. Wrong model location

I made a stupid mistake. At first I passed the argument ‘model’ into glUniformMatrix4fv(mLoc, ***), not the model names (‘board’ or ‘penguin’). As a result, the camera perspective looked wired, as shown in the following picture. After double-checking my code, I corrected the argument, and the window finally looks perfect.



2. Wired penguin texture

I found that another classmate has encountered the same problem on E3. Following the TA’s instructions, I added ‘stbi_set_flip_vertically_on_load(true)’ before ‘stbi_load’, and the penguin looks smooth and correct. The additional instruction helps to regulate the picture’s orientation according to the program’s coordinate system.