Graphics
LAB 2

謝侑哲

112550069

1. Explain in detail how to use GLSL by taking screenshots.

A. Prepare Shader

```
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 x and z coordinates.

    vec3 pos = aPos;
    float squeezeCofficient = sin(squeezeFactor) / 2.0;
    pos.x = aPos.x + aPos.z * squeezeCofficient;
    pos.z = aPos.z + aPos.x * squeezeCofficient;

    gl_Position = projection * view * model * vec4(pos, 1.0);

    TexCoord = aTexCoord;

}
```

```
void main()
{
    // TODO: Implement Breathing Light Effect
    //   1. Retrieve the color from the texture at texCoord.
    //   2. Set FragColor to be the texture color * (breathingColor * intensity)
    //   Note: Ensure FragColor is appropriately set for both breathing light and normal cases.

    vec4 textureColor = texture(ourTexture, TexCoord);

    float stripeSwitch = 1.0;
    if (useStripes == 1) {
        const float PI = 3.14159265359;
        float angle = atan(ObjPos.y, ObjPos.x);
        float norm = (angle + PI) / (2.0 * PI);
        float t = norm * stripeFrequency;
        stripeSwitch = step(0.5, fract(t));
    }
    vec3 stripeColor = mix(vec3(0.0), vec3(1.0), stripeSwitch);

    vec3 finalRgb = textureColor.rgb * stripeColor * (breathingColor * intensity);
    FragColor = vec4(finalRgb, textureColor.a);
}
```

在程式開始運作之前，要先準備好用來調整 vertex 位置、分布的 vertex shader 跟用來調整 rendered effect 的 fragment shader。

本次作業的 vertex shader 是用來調整魚的壓縮變形的，在 shader 中，會把 vertex 讀進來，然後對於其 x, z 根據 sin 時間變化，在 0.5~1.5 倍之間變化，而 y 軸不變，因為有一軸不變 (或是可以設計成變化量相反)，所以可以做到類似擠壓的效果。最後再把擠壓過後 vertex 應該在的位置傳回 GLSL 的在 normalized 相機空間的 position。

Fragment shader 則是想要做到呼吸燈的效果 (條紋效果後面再提到)，呼吸燈的效果就是讓光照會隨著時間變化。首先從 texture map 中，依據 texture coordinate 去決定物件各處的顏色(材質)，得到物體的基本 color，並且 main function 中會計算在這個時間點應該要有的呼吸燈估強度，接著對於物體原本的顏色(RGB 反照率)打上強度隨時間變化的燈光，就可以做到呼吸燈的效果。

B. Initialize

```
void init() {
#if defined(__linux__) || defined(__APPLE__)…
#else…
#endif

    // Object
    fishObject = new Object(dirAsset + "fish.obj");
    columnObject = new Object(dirAsset + "column.obj");
    groundObject = new Object(dirAsset + "cube.obj");

    // Shader
    vertexShader = createShader(dirShader + "vertexShader.vert", "vert");
    fragmentShader = createShader(dirShader + "fragmentShader.frag", "frag");
    shaderProgram = createProgram(vertexShader, fragmentShader);
    glUseProgram(shaderProgram);

    // Texture
    fishTexture = loadTexture(dirTexture + "fish.jpg");
    columnTexture = loadTexture(dirTexture + "column.jpg");
    groundTexture = loadTexture(dirTexture + "ground.jpg");

    // VAO, VBO
    fishVAO = modelVAO(*fishObject);
    columnVAO = modelVAO(*columnObject);
    groundVAO = modelVAO(*groundObject);
}
```

```
unsigned int createShader(const string &filename, const string &type) {
    ifstream shaderFile;
    shaderFile.exceptions(ifstream::failbit | ifstream::badbit);
    try { // 把shadder code讀進來
        shaderFile.open(filename);
        stringstream shaderStream;
        shaderStream << shaderFile.rdbuf();
        shaderFile.close();
        shaderCode = shaderStream.str();
    } catch (ifstream::failure &e) {
        std::cout << "[ERROR] shader file is not successfully read: " << filename << std::endl;
        return 0;
    }
    const char *shaderSource = shaderCode.c_str(); // source code

    unsigned int shader;
    GLenum shaderType;
    if (type == "vert")
        shaderType = GL_VERTEX_SHADER; // 決定shader type
    else if (type == "frag")
        shaderType = GL_FRAGMENT_SHADER;
    else {
        std::cout << "[ERROR] shader unknown type: " << type << std::endl;
        return 0;
    }
    shader = glCreateShader(shaderType); // 創建shader object

    glShaderSource(shader, 1, &shaderSource, NULL); // set source code into shader
    glCompileShader(shader); // compile with shader source code

    // 確認是否compile成功
    int success;
    char infoLog[512];
    glGetShaderiv(shader, GL_COMPILE_STATUS, &success); // get compile status information
    if (!success) {
        glGetShaderInfoLog(shader, 512, NULL, infoLog);
        std::cout << "[ERROR] shader " << type << " compilation failed\n" << infoLog << std::endl;
        return 0;
    }
    return shader;
```

首先把給定由 vertex, normal, texture coordinate, face 組成的 object。

接著把寫好的 shader 利用 createShader 讀入，首先用 file stream 讀入 shader 的文字，接著用 glCreateShader 創建 vertex, fragment shader object，然後把讀進來的 shader code 送進 shader object，去做編譯。就可以得到有實際 shading 效果的 shader object。

```cpp
unsigned int createProgram(unsigned int vertexShader, unsigned int fragmentShader) {
    // 把不同的shader串成一個pipeline program
    unsigned int program = glCreateProgram();

    glAttachShader(program, vertexShader);
    glAttachShader(program, fragmentShader);

    glLinkProgram(program); // 串起來檢查in, out並輸出儲存執行檔

    int success;
    char infoLog[512];
    glGetProgramiv(program, GL_LINK_STATUS, &success);
    if (!success) {
        glGetProgramInfoLog(program, 512, NULL, infoLog);
        std::cout << "[ERROR] program linking failed\n" << infoLog << std::endl;
        return 0;
    }

    // link完，program object已經儲存linked後的執行檔，可以把shader detach掉，節省記憶體
    glDetachShader(program, vertexShader);
    glDetachShader(program, fragmentShader);

    return program;
}
```

接著跟把多檔 C++串起來的方式一樣，把帶有 compiled code 的 shader，放進一個完整的 shader program，是包含了怎麼從 vertex 一路到 color 的 pipeline，接著把不同 shader 的 compiled code 用 linker 串起來，就可以得到整個 program (pipeline)的可執行程式。在之後，這個 shader program 就可以直接做 shading。

```cpp
unsigned int loadTexture(const string &filename) {
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_2D, textureID); // 跟Bind VBO的概念一樣

    // 設定texture超出給定u,v範圍時，變成重複環繞
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

    // 處理texture的放大縮小
    // ex: 在相機靠近物體時，texture會被放大
    // 這裡用線性插值來處理
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    int width, height, nrChannels;
    stbi_set_flip_vertically_on_load(true); // 把jpg的座標改成OpenGL的座標（原點從左上改到左下）
    unsigned char *data = stbi_load(filename.c_str(), &width, &height, &nrChannels, 0); // load tex

    if (data) {
        GLenum format;
        if (nrChannels == 1)
            format = GL_RED;
        else if (nrChannels == 3)
            format = GL_RGB;
        else if (nrChannels == 4)
            format = GL_RGBA;
        else {
            std::cout << "[ERROR] texture unsupported format: " << filename << std::endl;
            stbi_image_free(data);
            return 0;
        }
        glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0, format, GL_UNSIGNED_BYTE, data);
    } else {
        std::cout << "Failed to load texture: " << filename << std::endl;
    }
    // free image memory
    stbi_image_free(data);

    return textureID;
}
```

接著要把給定的 texture 從 jpg 檔載入進程式，首先生成一個 texture object，然後把這個 texture 綁定到 GLSL 的變數 GL_TEXTURE_2D 上，在 GLSL 處理 texture 的時候，就會使用這個 texture object，接著設定一些 texture 如何被使用，像是如果超出界線就循環 texture、如果有對 texture 有縮放就用線性插值插出對應大小的 texture。最後讀入 texture 的 jpg 檔到 GLSL 的 GL_TEXTURE_2D 上 (該 texture object)，把他從 CPU 丟到 GPU 區塊，方便後面做 fragment 的時候可以平行快速取用。

```
unsigned int modelVAO(Object &model) {
    // 把vertex丟到GPU，就可以不用一個一個跑
    // 利用 Vertex Buffer Objects 管理
    unsigned int VAO;
    unsigned int VBO[2];  // 需要兩個buffer，儲存position and texcoords

    glGenVertexArrays(1, &VAO); // 生成Vertex Array Object，用來指向VBO的pointer，用來管理大量物件
    glBindVertexArray(VAO);

    glGenBuffers(2, VBO); // 生成 2 個 VBO，把id存在VBO[]裡

    glBindBuffer(GL_ARRAY_BUFFER, VBO[0]); // 把 VBO[0] bind 到 GL_ARRAY_BUFFER status，用來標示OpenGL struct中儲存ver
    // 把position資料丟到GPU
    // GL_STATIC_DRAW表示CPU只會設定一次資料位置(static)，之後只會被GPU頻繁讀取(draw)，讓GPU知道應該一次就放在一個比較快速能
    glBufferData(GL_ARRAY_BUFFER, sizeof(float) * model.positions.size(), &model.positions[0], GL_STATIC_DRAW);
    // 告訴OpenGL怎麼解讀buffer裡的資料
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0); // 設定location為0的讀取規則，每次讀3個float (xyz)，
    glEnableVertexAttribArray(0);

    // 跟前面的position類似
    if (!model.texcoords.empty()) {
        glBindBuffer(GL_ARRAY_BUFFER, VBO[1]);
        glBufferData(GL_ARRAY_BUFFER, sizeof(float) * model.texcoords.size(), &model.texcoords[0], GL_STATIC_DRAW);
        glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 0, (void*)0); // (uv)2個一組
        glEnableVertexAttribArray(1);
    }

    // 解綁
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindVertexArray(0);

    return VAO;

}
```

處理完 shader 跟 texture 之後，最後要初始化我們之後要把 vertex 丟到 GPU 時的 buffer 運作方式，這裡使用 Vertex Buffer Object (VBO)管理，主要管理 vertex 的 position 跟 texture coordinate，首先創建一個 Vertex Array Object (VAO)來記錄 VBO 的設定，接著開始設定怎麼把 vertex 從 CPU 放到 GPU，首先創建從 CPU 到 GPU 的 buffer (VBO)，綁定到 GLSL 的變數 GL_ARRAY_BUFFER 上，讓 GLSL 知道應該要把 Buffer 放在甚麼位置，接著用 glBufferData 讓 GPU 去設定一塊 read time optimized 區域，再設定讀取的規則 (每次跳多遠讀取)，設定好規則後套用。最後把 VBO 解綁，釋放記憶體。

完成以上動作，就完成了初始化，把 shader, texture, VBO 都初始化好，方便後面使用。

C. GLSL Location Setting

```
/* TODO#5: Data connection - Retrieve uniform variable locations
 *    1. Retrieve locations for model, view, and projection matrices.
 *    2. Retrieve locations for squeezeFactor, breathingColor, intensity, and other parameters.
 * Hint:
 *    glGetUniformLocation
 */

// 取得GLSL linked program中各個變數的位置 (link完之後，位置有各種不可能，要去實際access才知道)
GLint modelLoc, viewLoc, projLoc;
GLint squeezeFactorLoc, breathingColorLoc, intensityLoc, texLoc;
modelLoc = glGetUniformLocation(shaderProgram, "model");
viewLoc = glGetUniformLocation(shaderProgram, "view");
projLoc = glGetUniformLocation(shaderProgram, "projection");
squeezeFactorLoc = glGetUniformLocation(shaderProgram, "squeezeFactor");
breathingColorLoc = glGetUniformLocation(shaderProgram, "breathingColor");
intensityLoc = glGetUniformLocation(shaderProgram, "intensity");
texLoc = glGetUniformLocation(shaderProgram, "ourTexture");
```

因為 main.cpp 在 CPU 運行，所以我們依據 CPU 時間計算的各種變數或者是計算的 transform matrix 不能直接被 GPU 上的 shader 取用，所以要先取得 shader program 中，各個變數的 location，以便之後從 CPU 把對應變數送到 GPU 的對應位置。

D. Render Ground, Column, Fish and send to GLSL

接下來要 Render 場景中的物件，並把物件根據上一步拿到的 location 送到 GPU 中。

```
/* TODO#6-1: Render Ground
 *    1. Set up ground model matrix.
 *    2. Send model, view, and projection matrices to the program.
 *    3. Send squeezeFactor, breathingColor, intensity, or other parameters to the program.
 *    4. Apply the texture, and render the ground.
 * Hint:
 *    rotate, translate, scale
 *    glUniformMatrix4fv, glUniform1f, glUniform3fv
 *    glActiveTexture, glBindTexture, glBindVertexArray, glDrawArrays
 */

// 把view, projection這些所有物件共通的矩陣送到shader program的對應位置，用來後面render
glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::value_ptr(view));
glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(projection));

groundModel = glm::mat4(1.0f);
groundModel = glm::translate(groundModel, glm::vec3(0.0f, -5.0f, 8.0f));
groundModel = glm::scale(groundModel, glm::vec3(35.0f, 1.0f, 25.0f));
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(groundModel)); // 送進shader program的

// 上傳其他的地面的GLSL變數，用來讓shder program知道該怎麼去render
glUniform1f(squeezeFactorLoc, 0.0f); // 不使用squeeze effect
glUniform3f(breathingColorLoc, 1.0f, 1.0f, 1.0f); // 呼吸燈的顏色固定白色 (保持原色)
glUniform1f(intensityLoc, 1.0f); // 不使用呼吸燈 (強度設為time-independent的1.0)

glActiveTexture(GL_TEXTURE0); // 啟用texture 0 status unit
glBindTexture(GL_TEXTURE_2D, groundTexture); // 把地面的texture綁到目前的texture unit 0
glUniform1i(texLoc, 0); // 告訴shader ground texture在texture unit 0 (texture location的第0個)

glBindVertexArray(groundVAO); // bind vao, 載入vbo設定，就不用重新設定vbo的load
glDrawArrays(GL_TRIANGLES, 0, static_cast<GLsizei>(groundObject->positions.size() / 3)); // 把
```

第一個 render 的是地板，render 的方法細節跟 Lab 1 一樣，這裡就不贅述，render 流程是: 先把地板縮放到河是大小，再移動到合適位置。在得到 transform model 之後，就根據 model location 送進 GLSL，以便在前面提到的 fragment shader 中使用。另外因為地板不需要擠壓跟呼吸燈效果，所以把 squeezeFactor, breathColor, Intensity 都寫成 identity 的數值傳入 GLSL。之後把 texture 也送入 GLSL 讓 Fragment shader 去取得 polygon color。最後載入前面做好的 VAO，讓繪圖可以在 GPU 上去做。

```
/* TODO#6-2: Render Column
 *    1. Set up column model matrix.
 *    2. Send model, view, and projection matrices to the program.
 *    3. Send squeezeFactor, breathingColor, intensity, or other parameters to the program.
 *    4. Apply the texture, and render the column.
 * Hint:
 *    rotate, translate, scale
 *    glUniformMatrix4fv, glUniform1f, glUniform3fv
 *    glActiveTexture, glBindTexture, glBindVertexArray, glDrawArrays
 */

columnModel = glm::mat4(1.0f);
columnModel = glm::translate(columnModel, glm::vec3(0.0f, -4.0f, 0.0f));
columnModel = glm::rotate(columnModel, radians(rotateColumnDegree), glm::vec3(0.0f, 1.0f, 0.0f));
columnModel = glm::rotate(columnModel, radians(-90.0f), glm::vec3(1.0f, 0.0f, 0.0f));
columnModel = glm::scale(columnModel, glm::vec3(0.05f, 0.05f, 0.05f));
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(columnModel));

glUniform1f(squeezeFactorLoc, 0.0f);
glUniform3f(breathingColorLoc, 1.0f, 1.0f, 1.0f);
glUniform1f(intensityLoc, 1.0f);

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, columnTexture);
glUniform1i(texLoc, 0);

glBindVertexArray(columnVAO);
glDrawArrays(GL_TRIANGLES, 0, static_cast<GLsizei>(columnObject->positions.size() / 3));
```

```
/* TODO#6-3: Render Fish
 *    1. Set up fish model matrix.
 *    2. Send model, view, and projection matrices to the program.
 *    3. Send squeezeFactor, breathingColor, intensity, or other parameters to the program.
 *    4. Apply the texture, and render the fish.
 * Hint:
 *    rotate, translate, scale
 *    glUniformMatrix4fv, glUniform1f, glUniform3fv
 *    glActiveTexture, glBindTexture, glBindVertexArray, glDrawArrays
 */

fishModel = glm::mat4(1.0f);
fishModel = glm::rotate(fishModel, radians(revolveFishDegree), glm::vec3(0.0f, 1.0f, 0.0f));
fishModel = glm::translate(fishModel, glm::vec3(0.0f, fishHeight, fishColumnDist));
fishModel = glm::rotate(fishModel, radians(rotateFishDegree), glm::vec3(-1.0f, 0.0f, 0.0f));
fishModel = glm::rotate(fishModel, radians(-90.0f), glm::vec3(1.0f, 0.0f, 0.0f));
fishModel = glm::scale(fishModel, glm::vec3(0.05f, 0.05f, 0.05f));
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(fishModel));

glUniform1f(squeezeFactorLoc, radians(squeezeFactor)); // 隨時間擠壓
glUniform3fv(breathingColorLoc, 1, glm::value_ptr(breathingColor)); // 呼吸燈顏色，把breathin
glUniform1f(intensityLoc, intensity); // 隨時間改變呼吸燈強度

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, fishTexture);
glUniform1i(texLoc, 0);

glBindVertexArray(fishVAO);
glDrawArrays(GL_TRIANGLES, 0, static_cast<GLsizei>(fishObject->positions.size() / 3));
```

接下來要 render column, fish，因為整個流程跟 render ground 一樣，所以這裡只講怎麼 transform 這兩個 object 跟一些變數的設定。Column 的 transform 是先縮放到對應的大小，然後旋轉到要求的面向，之後因為 column 會自轉，所以用一個會隨時間更新的 rotateColumnDegree 來旋轉 column，最後移動到對應的位置。在 shading 變數上，跟 ground 一樣，不需要呼吸燈跟擠壓效果，所以這兩個都設成 identity 的數值。Fish 的 transform 則是先縮放到對應大小、轉到要求方向，接著因為魚會隨著按鍵自轉，所以用一個會根據按鍵觸發跟時間更新的變數 rotateFishDegree 去旋轉這隻魚，最後因為魚要繞著柱子公轉，所以先把魚按轉軸移動到需要的半徑，再去依照一個隨時間變化的變數 revolveFishDegree 繞著這個半徑旋轉。對於 shading 變數，因為魚要用黃色的呼吸燈，所以用一個會隨著時間變化的 intensity 跟黃色的 breathingColor，而魚需要在按下按鍵後擠壓，所以傳入一個會因為按按鍵及時間而改變的 squeezeFactor。

D. Variable Update

```
/* TODO#7: Update "revolveFishDegree", "rotateColumnDegree", "rotateFishDegree",
 *          "fishHeight", "heightDir",
 *          "squeezeFactor", "breathingColor", "intensity"
 */


rotateColumnDegree += rotateColumnSpeed * dt;
revolveFishDegree += -revolveFishSpeed * dt;

if (useSelfRotation) {
    rotateFishDegree += rotateFishSpeed * dt;
    if (rotateFishDegree >= 360.0f) {
        rotateFishDegree = 0.0f;
        useSelfRotation = false; // 停止自轉（只在這裡觸發，避免在其他情況被中斷自轉）
    }
}

// 慢慢上下游
fishHeight += heightDir * heightSpeed * dt;
if (fishHeight > 0.5f || fishHeight < -0.5f) {
    heightDir *= -1; // 換方向
    fishHeight = clamp(fishHeight, -0.5f, 0.5f);
}

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

if (useBreathing) {
    breathingColor = glm::vec3(1.0f, 1.0f, 0.0f);
    intensity = (sin(currentTime * 3.0f) * 0.25f) + 0.75f; // 用sin來跑呼吸燈強度變化
} else {
    breathingColor = glm::vec3(1.0f, 1.0f, 1.0f); // 不使用呼吸燈：用白燈 保持顏色不變
    intensity = 1.0f;
}
```

```
void keyCallback(GLFWwindow *window, int key, int scancod
    */
    if (action == GLFW_PRESS) {

        if (key == GLFW_KEY_R) {
            if (!useSelfRotation) { // 避免重置進行中的
                useSelfRotation = true;
                rotateFishDegree = 0.0f;
            }
        }
        else if (key == GLFW_KEY_S) {
            useSqueeze = !useSqueeze;
        }
        else if (key == GLFW_KEY_B) {
            useBreathing = !useBreathing;
        }
        else if (key == GLFW_KEY_ESCAPE) {
            glfwSetWindowShouldClose(window, true);
        }
    }
```

在前一步有提到，render 時會用到很多會依據時間、按鍵變化的變數，這一步就是要講到變數是如何被更新的。

對於柱子自轉跟魚的公轉，都是隨時間變化的。而魚的自轉是在使用者按下 R 之後，使 UseSelfRotate 變成 True，才會開始做旋轉的動作，並且這個動作只在結束(角度達 360 度)之後才會變 False，做到了不可中斷性。魚的擠壓則是在使用者按下 S 之後，會觸發擠壓，然後讓 squeezeFactor 在按下 S 之後才會隨時間變化。魚的呼吸燈效果概念一樣，當使用者按下 B 之後會觸發，呼吸燈的顏色被啟動 (原本是不影響物件顏色的白光)，且 intensity 就會隨時間變化，並且是隨 sin function 變化，呈現非線性類似呼吸的效果。另外，因為還要讓魚上下游動，因此我們會讓魚隨著時間在一定的 height 範圍往上游到上界，再往下游到下界，循環往復。

2. Describe the problems you met and how you solved them.

A. Difficulty to Debug

因為要串起 shader 的整個 pipeline 還挺複雜的，中間只要有一個小部分寫錯，就會導致記憶體出錯，最後出來的視窗會直接變白畫面死掉。而完全的白畫面就會導致很難知道是出了甚麼 bug，在 GPU 上的 bug 也很難設 Breakpoint 來解掉。

我後來的解決方法是，跟著整個運行的流程，一步步的輸出所有的資訊，再嘗試從資訊中找到有沒有跟預期數值不符合的數，最後才慢慢找到了 bug 出在哪裡並解掉。

B. Frame, Time Confusion

Rotate and revolve :

- The column need to self rotate at 10 degrees per frame around y axis
- The fish need to revolve the column at -20 degrees per frame, and swim up and down slowly. You need to make sure the fish is still visible in the window.

在 requirement 中有提到，要讓柱子跟魚 per frame 去做旋轉，但是這樣的話旋轉速度會太快，導致柱子跟魚變得像陀螺一樣。而我後來選擇的解法是遵照 demo 影片的方式，改成用 per second 去旋轉，才做出比較合理的結果。

C. Texture UV Flipping Issue

在實作 Texture 時，我發現物件拿到的 texture 整個亂掉，雖然該有的 texture 有被貼到物件上，但看起來有種位置錯亂的感覺，一開始完全不知道是為甚麼會這樣炸掉，後來是在跟 LLM 對話之後，才發現原來 jpg 跟 OpenGL 的坐標系是不同的，最後用翻轉座標的方式才解決這個問題。

3. Explain how to use your design of advanced part

A. Stripes on cloumn

```
void main()
{
    // TODO: Implement Breathing Light Effect
    //   1. Retrieve the color from the texture at texCoord.
    //   2. Set FragColor to be the texture color * (breathingColor * intensity)
    //   Note: Ensure FragColor is appropriately set for both breathing light and no

    vec4 textureColor = texture(ourTexture, TexCoord);

    float stripeSwitch = 1.0;
    if (useStripes == 1) {
        const float PI = 3.14159265359;
        float angle = atan(ObjPos.y, ObjPos.x);
        float norm = (angle + PI) / (2.0 * PI);
        float t = norm * stripeFrequency;
        stripeSwitch = step(0.5, fract(t));
    }

    vec3 finalRgb = textureColor.rgb * stripeSwitch * (breathingColor * intensity);
    FragColor = vec4(finalRgb, textureColor.a);
}
```

在 fragment shader 的部分，我有額外去設計了在柱子上的 stripe，要設計出 stripe 的核心概念就是把一個物體的某個角度範圍的顏色全部變成那個 stripe 的顏色。所以第一個要做的是透過 atan 去取得每個點相對於中心點的角度，接著把角度 normalized 到 0~1 之間，代表一圈，然後根據從 CPU 傳入的 stripeFrequency 去決定要在這個物體上有幾條條紋，用 norm 呈上頻率的意義是，會讓數值範圍從 0~1 變成 0~stripeFrequency，然後我們就可以切分成每隔數值 1 的角度就出現一條條紋。之後把 stripeSwitch 當作 color 的乘數，當 stripeSwitch=0 時，相當於打上黑光，讓物體呈現黑色條紋，stripeSwitch=1 時，相當於打上白光，讓物體呈現原本的顏色。

4. Demo YouTube Link

https://youtu.be/R5tjYABccIE