Vertex Shader: Code:

```
#version 330 core

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

out vec2 TexCoord;
out vec3 ourColor;

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

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0f);
    TexCoord = aTexCoord;
    ourColor = aColor;
}
```

Fragment Shader: Code:

```
#version 330 core

in vec2 TexCoord;
in vec3 ourColor;

out vec4 FragColor;

uniform sampler2D texture1;
uniform sampler2D texture2;

void main()
{
    FragColor = texture(texture1, TexCoord);
}
```

Main File:

```
#include "Main_include.hpp"

void framebuffer_size_callback(GLFWwindow* window, int width, int height);
void mouse_callback(GLFWwindow* window, double xpos, double ypos);
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset);
void processInput(GLFWwindow *window, Camera* camera);
void RecursiveTriangle(unsigned int transformID, glm::mat4 mat, const glm::vec3& translate,
 ↪  int depth);

// settings
const unsigned int SCR_WIDTH = 800;
const unsigned int SCR_HEIGHT = 600;


int main()
{
    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

    GLFWwindow* window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT, "LearnOpenGL", NULL, NULL);
```

```
    if (window == NULL)
    {
        std::cout << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return -1;
    }
    glfwMakeContextCurrent(window);
    glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
    glfwSetCursorPosCallback(window, mouse_callback);
    glfwSetScrollCallback(window, scroll_callback);

    if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
    {
        std::cout << "Failed to initialize GLAD" << std::endl;
        return -1;
    }
    glEnable(GL_DEPTH_TEST);
    Shader t1Shader("Shaders/rotating_pyramid.glvs", "Shaders/rotating_pyramid.glfs");

    // set up vertex data (and buffer(s)) and configure vertex attributes
    // ------------------------------------------------------------------
    float vertices[] = {
         0.0f, -1.0f,  0.0f, 1.0f, 0.0f, 0.0f, 0.5f, 0.0f,
        -1.0f,  1.0f,  1.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f,
         1.0f,  1.0f,  1.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, // first triangle
         0.0f, -1.0f,  0.0f, 1.0f, 0.0f, 0.0f, 0.5f, 0.0f,
         1.0f,  1.0f,  1.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f,
         1.0f,  1.0f, -1.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, // second triangle
         0.0f, -1.0f,  0.0f, 1.0f, 0.0f, 0.0f, 0.5f, 0.0f,
         1.0f,  1.0f, -1.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f,
        -1.0f,  1.0f, -1.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, // third triangle
         0.0f, -1.0f,  0.0f, 1.0f, 0.0f, 0.0f, 0.5f, 0.0f,
        -1.0f,  1.0f, -1.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f,
        -1.0f,  1.0f,  1.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, // fourth triangle
        -1.0f,  1.0f,  1.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f,
         1.0f,  1.0f,  1.0f, 0.0f, 0.0f, 1.0f, 1.0f, 0.0f,
         1.0f,  1.0f, -1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 1.0f,
        -1.0f,  1.0f,  1.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f,
         1.0f,  1.0f, -1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 1.0f,
        -1.0f,  1.0f, -1.0f, 0.0f, 1.0f, 1.0f, 0.0f, 1.0f

    };
    unsigned int VBO, VAO;
    // unsigned int EBO;
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
    //glGenBuffers(1, &EBO);
    // bind the Vertex Array Object first, then bind and set vertex buffer(s), and then
     ↪  configure vertex attributes(s).
    glBindVertexArray(VAO);

    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

    //glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
    //glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);

    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);

```

```
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float),
    ↪ (void*)(3*sizeof(float)));
    glEnableVertexAttribArray(1);

    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(6 *
    ↪ sizeof(float)));
    glEnableVertexAttribArray(2);

    //load and create texture
    //-----------------------
    unsigned int texture1, texture2;
    glGenTextures(1, &texture1);
    glBindTexture(GL_TEXTURE_2D, texture1);

    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);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

    int width,height,nrChannels;

    stbi_set_flip_vertically_on_load(true);
    unsigned char* data = stbi_load("media/container.jpg", &width, &height, &nrChannels, 0);
    if(data)
    {
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE,
    ↪ data);
        glGenerateMipmap(GL_TEXTURE_2D);
    }
    else
    {
        std::cout<<"Failed to load data";
    }
    stbi_image_free(data);

    glGenTextures(1, &texture2);
    glBindTexture(GL_TEXTURE_2D, texture2);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_MIRRORED_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_MIRRORED_REPEAT);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

    data = stbi_load("media/awesomeface.png", &width, &height, &nrChannels, 0);
    if(data)
    {
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA,
    ↪ GL_UNSIGNED_BYTE, data);
        glGenerateMipmap(GL_TEXTURE_2D);
    }
    else
    {
        std::cout<<"Failed to load data";
    }
    stbi_image_free(data);

    Camera cam1{glm::vec3(0.0f, 0.0f, 3.0f)};
    glfwSetWindowUserPointer(window, (void*)(&cam1));

```

```
    t1Shader.use();
    t1Shader.setUniform("texture1", 0);
    t1Shader.setUniform("texture2", 1);
    // render loop
    // -----------
    while (!glfwWindowShouldClose(window))
    {

        // input
        // -----
        processInput(window, &cam1);

        // render
        // ------
        glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, texture1);
        glActiveTexture(GL_TEXTURE1);
        glBindTexture(GL_TEXTURE_2D, texture2);

        glm::mat4 projection{1.0f};
        projection = glm::perspective(glm::radians(cam1.GetZoom()),
                    16.0f/9.0f, 0.1f, 100.0f);

        t1Shader.setUniform("projection", projection);

        t1Shader.setUniform("view", cam1.ViewMatrix());


        RecursiveTriangle(glGetUniformLocation(t1Shader.shaderProgramID,"model"),
                          glm::mat4(1.0f), glm::vec3(0.0f,-0.5f,0.0f), 5);

        glBindVertexArray(VAO);
        glDrawArrays(GL_TRIANGLES, 0, 18);


        // glfw: swap buffers and poll IO events (keys pressed/released, mouse moved etc.)
        // -------------------------------------------------------------------------------
        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    // optional: de-allocate all resources once they've outlived their purpose:
    // ------------------------------------------------------------------------
    glDeleteVertexArrays(1, &VAO);
    glDeleteBuffers(1, &VBO);
    //glDeleteBuffers(1, &EBO);
    // glfw: terminate, clearing all previously allocated GLFW resources.
    // ------------------------------------------------------------------
    glfwTerminate();
    return 0;
}

// process all input: query GLFW whether relevant keys are pressed/released this frame and
 ↪  react accordingly
void processInput(GLFWwindow *window, Camera* camera)
{
    static float lastframe{0.0f};
    static float currentframe{0.0f};
```

```
    static float deltatime{0.0f};
    static int state{GLFW_CURSOR_NORMAL};

    currentframe = (float)glfwGetTime();
    deltatime = currentframe - lastframe;
    lastframe = currentframe;

    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
    {glfwSetWindowShouldClose(window, true);}
    if (glfwGetKey(window, GLFW_KEY_LEFT_SHIFT) == GLFW_PRESS
        && glfwGetKey(window, GLFW_KEY_Z) == GLFW_PRESS)
    {
        glfwSetInputMode(window, GLFW_CURSOR, state);
        state = (state == GLFW_CURSOR_NORMAL ? GLFW_CURSOR_DISABLED : GLFW_CURSOR_NORMAL);
    }
    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
    {camera->MoveCamera(CameraMovement::FORWARD, deltatime);}
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
    {camera->MoveCamera(CameraMovement::BACKWARD, deltatime);}
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
    {camera->MoveCamera(CameraMovement::LEFT, deltatime);}
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
    {camera->MoveCamera(CameraMovement::RIGHT, deltatime);}
}

// glfw: whenever the window size changed (by OS or user resize) this callback function
 ↪  executes
void framebuffer_size_callback(GLFWwindow* window, int width, int height)
{
    glViewport(0, 0, width, height);
}

void mouse_callback(GLFWwindow* window, double xpos, double ypos)
{
    static bool firstMouse{true};
    static float lastX{(float)SCR_WIDTH / 2.0f};
    static float lastY{(float)SCR_HEIGHT / 2.0f};
    if (firstMouse)
    {
        lastX = xpos;
        lastY = ypos;
        firstMouse = false;
    }

    float xoffset = xpos - lastX;
    float yoffset = lastY - ypos; // reversed since y-coordinates go from bottom to top
    lastX = xpos;
    lastY = ypos;

    Camera* camera = (Camera*)glfwGetWindowUserPointer(window);
    camera->RotateCamera(xoffset, yoffset);
}

void scroll_callback(GLFWwindow* window, double xoffset, double yoffset)
{
    Camera* camera = (Camera*)glfwGetWindowUserPointer(window);
    camera->ZoomCamera(yoffset);
}

void RecursiveTriangle(unsigned int transformID, glm::mat4 mat, const glm::vec3& translate,
 ↪  int depth)
```

```
{
    if (depth == 0)
    {
        return;
    }
    mat = glm::translate(mat, translate);
    mat = glm::scale(mat, glm::vec3(0.5f, 0.5f, 0.5f));
    glm::mat4 matnew = mat;
    mat = glm::rotate(mat, (float)glfwGetTime(), glm::vec3(0.0f, 1.0f, 0.0f));

    glUniformMatrix4fv(transformID, 1, GL_FALSE, &mat[0][0]);
    glDrawArrays(GL_TRIANGLES, 0, 18);
    --depth;
    RecursiveTriangle(transformID, matnew, glm::vec3(0.0f, 1.7f, 0.0f), depth);
    RecursiveTriangle(transformID, matnew, glm::vec3(-1.4f, -0.5f, 0.0f), depth);
    RecursiveTriangle(transformID, matnew, glm::vec3(1.4f,  -0.5f, 0.0f), depth);
}
```

Camera Movement Logic: Code:

```
#include "Camera.hpp"

Camera::Camera(glm::vec3 position ,
               glm::vec3 wup ,
               float yaw, float pitch ) :
               Position{position}, WorldUp{wup}, Front{0.0f, 0.0f, -1.0f}, Yaw{yaw},
 ↪  Pitch{pitch},
               MovementSpeed{SPEED}, MouseSensitivity{SENSITIVITY}, Zoom{ZOOM}
{
    UpdateCamera();
}

Camera::Camera(float posX, float posY, float posZ,
           float wupX, float wupY, float wupZ) : Camera(glm::vec3(posX,posY,posZ),
                                                        glm::vec3(wupX, wupY,wupZ))
{
}
void Camera::UpdateCamera()
{
    glm::vec3 front;
    front.x = glm::cos(glm::radians(Yaw)) * glm::cos(glm::radians(Pitch));
    front.y = glm::sin(glm::radians(Pitch));
    front.z = glm::sin(glm::radians(Yaw)) * glm::cos(glm::radians(Pitch));
    Front   = glm::normalize(front);

    Right = glm::normalize(glm::cross(Front, WorldUp));
    Up    = glm::normalize(glm::cross(Right, Front));
}

void Camera::MoveCamera(CameraMovement direction, float deltatime)
{
    float velocity = MovementSpeed * deltatime;
    if(direction == CameraMovement::FORWARD)
        Position += velocity * Front;
    if(direction == CameraMovement::BACKWARD)
        Position -= velocity * Front;
    if(direction == CameraMovement::LEFT)
        Position -= velocity * Right;
    if(direction == CameraMovement::RIGHT)
        Position += velocity * Right;
}

```

```cpp
void Camera::RotateCamera(float xoffset, float yoffset, bool constrainPitch)
{
    Yaw   += xoffset * MouseSensitivity;
    Pitch += yoffset * MouseSensitivity;

    if (constrainPitch)
    {
        if (Pitch > 89.0f)
        {
            Pitch = 89.0f;
        }
        else if (Pitch < -89.0f)
        {
            Pitch = -89.0f;
        }
    }
    UpdateCamera();
}

void Camera::ZoomCamera(float yoffset)
{
    Zoom -= (float)yoffset;
    if (Zoom < 1.0f)
        Zoom = 1.0f;
    if (Zoom > 45.0f)
        Zoom = 45.0f;
}

glm::mat4 Camera::ViewMatrix()
{
    return glm::lookAt(Position, Position + Front, WorldUp);
}

glm::mat4 Camera::lookat(const glm::vec3& eye, const glm::vec3& center, const glm::vec3& up)
{
    glm::vec3 Direction{eye - center};
    glm::vec3 Right{glm::normalize(glm::cross(up, Direction))};
    glm::vec3 Camup{glm::normalize(glm::cross(Direction, Right))};

    glm::mat4 lookat{1.0f};
    lookat[0][0] = Right.x;
    lookat[0][1] = Camup.x;
    lookat[0][2] = Direction.x;
    lookat[1][0] = Right.y;
    lookat[1][1] = Camup.y;
    lookat[1][2] = Direction.y;
    lookat[2][0] = Right.z;
    lookat[2][1] = Camup.z;
    lookat[2][2] = Direction.z;
    lookat[3][0] = - glm::dot(eye, Right);
    lookat[3][1] = - glm::dot(eye, Camup);
    lookat[3][2] = - glm::dot(eye, Direction);
    return lookat;

}


float Camera::GetZoom()
{
    return Zoom;
}
```