

3D OGL Games Programming - Mateusz Jadasz

by Mateusz Jadasz

Submission date: 09-Jan-2023 11:53PM (UTC+0000)

Submission ID: 194270898

File name: 17845_Mateusz_Jadasz_3D_OGL_Games_Programming_-_Mateusz_Jadasz_473926_107959123.docx (268.21K)

Word count: 2702

Character count: 21826

3D OGL GAMES PROGRAMMING

By Mateusz Jadasz (1901291)

TABLE OF CONTENTS

Table of Contents	2
Introduction.....	3
Application Capabilities	3
Handling Textures.....	3
Testing	4
User Guide	5
References	6
Appendix.....	7
Source.Cpp.....	7
fragment_shader.glsl	10
vertex_shader.glsl.....	11
MaterialClass.h	12
MeshClass.h	12
ModelClass.h	15
};ObjectClass.h.....	15
ShaderClass.h.....	17
TextureClass.h	20
VertexClass.h	21

INTRODUCTION

During the entire design process of the application, I have had the opportunity to use and learn about a vast variety of concepts when it came to OpenGL and well as shaders programming, and some of the functions that OpenGL can perform. These include but are not limited to implementing my own ambient, diffuse and specular lighting systems, as well as some basic vertex and fragment shaders using glsl.

APPLICATION CAPABILITIES

The first thing I did when I began writing the application code is, I created a baseline for the fragment and vertex shader files. I found myself continuously expanding and sometimes slightly changing the functionality of the glsl files, however, at the end they ended up being primarily responsible for handling the lighting as well as a setting a format for handling matrices and how some of the provided data is read. For example, which numbers in an array are used to define the locations of vertices, colours, textures and normals.

Although I was unable to get the software to serve its goal of being a memory game, it can still perform almost everything else, you could expect. The application was structured in such a way, where every important part of the algorithm, was handled independently by a different header file. This means that handling textures, meshes, shaders, 3D objects themselves, etc. was all handled using separate header files. This level of organisation allowed me to keep the main part of the code relatively short, and clean. It also meant, that if I decided to add more objects on the screen, all it took was one line of code.

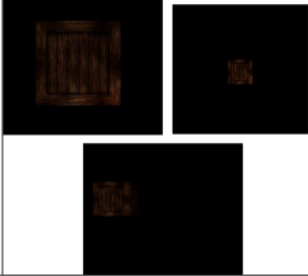
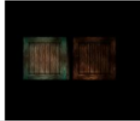
Aside from the larger, main functions of the program, although not utilised very much, there is also the possibility to scale, rotate and translate the objects in the view space.

Overall, the program can create and render 3D objects (although I limited the object options to a cube, for simplicity), it can apply multiple textures to one object, and each object is an individual entity, which means its coordinates, or textures will not interfere with other cubes on screen. The program was also written in such way, where geometric shaders could easily be added, although I never used them myself as there was no dire need. The program makes use of three different lighting types in the fragment shaders, those being ambient, diffuse, and specular. Finally, the program also makes use of face culling, and view control, which means that although there is no memory game functionality, it is still possible to move around and see the lighting effects and the objects themselves.

HANDLING TEXTURES

There were a handful of options for how I was going to handle adding textures to my objects. In the end I chose to use the SOIL2 library (Simple OpenGL Image Loader), which made handling images and adding textures incredibly easy. The library itself, is available on github [1] at: <https://github.com/SpartanJ/SOIL2>. Apart from the library itself, I also used an image [2] from a forum ran by the Blender community, which I have referenced in the code.

TESTING

Test Title	Test Description	Test Result Screenshot
Lighting	Checking the different lighting along the different faces of an object	
Creating/Removing Objects	Checking the ability to add or remove objects from the world space with ease	
Moving the View	Checking the ability to move closer, further or around an object	
Rotating the View	Checking the ability to look directly at an object, no matter where the object is being viewed from	
Layering Multiple Textures on one Object	Checking if an object can be rendered with more than one texture	
Objects Scaling with the Window Size	Checking if the objects will get bigger if the window size increases, to stay directly proportional to the display aspect ratio	

Button	Button Function
W	Move the view forward
A	Move the view back
S	Strafe the view to the left
D	Strafe the view to the right
Q	Rotate the view to the left
E	Rotate the view to the right
Esc	Exit the application entirely

REFERENCES

- [1] SpartanJ, "SOIL2," [Online]. Available: <https://github.com/SpartanJ/SOIL2>.
- [2] basil60, "Modeling a and texturing a wooden crate," May 2013. [Online]. Available: <https://blenderartists.org/t/modeling-a-and-texturing-a-wooden-crate/574576>. [Accessed 12 December 2022].

APPENDIX

SOURCE.CPP

```
#include "GL/glew.h"
#include "GLFW/glfw3.h"
#include "glm/glm.hpp"
#include "glm/gtc/matrix_transform.hpp"
#include "glm/ext.hpp"

#include "SOIL2/SOIL2.h"

#include "ShaderClass.h"
#include "TextureClass.h"
#include "MaterialClass.h"
#include "VertexClass.h"
#include "MeshClass.h"
#include "ObjectClass.h"
#include "ModelClass.h"

#include<fstream>
#include<iostream>
#include<string>
#include<vector>
#include<list>

using namespace std;

glm::mat4 viewMatrix(1.0);
glm::vec3 camPos = glm::vec3(0, 0, 2);
glm::vec3 camAngle = glm::vec3(0, 0, -1);
glm::vec3 camPosChange = glm::vec3(0, 0, 0);

//glfw window parameters
const int windowHeight = 640;
const int windowHeight = 480;
int framebufferWidth = 0;
int framebufferHeight = 0;

float fov = 90.0;
float nearPlane = 0.1;
float farPlane = 1000.0;
float angle = 0;

void updateInput(GLFWwindow* window, Shader* shader)
{
    //close program by pressing escape
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
    {
        glfwSetWindowShouldClose(window, GLFW_TRUE);
    }

    //camera movement
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
    {
        camPos.x = camPos.x + 0.05 * camAngle.z;
        camPos.z = camPos.z - 0.05 * camAngle.x;
    }
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
    {

```

```

        camPos.x = camPos.x - 0.05 * camAngle.z;
        camPos.z = camPos.z + 0.05 * camAngle.x;
    }
    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
    {
        camPos.x = camPos.x + 0.05 * camAngle.x;
        camPos.z = camPos.z + 0.05 * camAngle.z;
    }
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
    {
        camPos.x = camPos.x - 0.05 * camAngle.x;
        camPos.z = camPos.z - 0.05 * camAngle.z;
    }
    if (glfwGetKey(window, GLFW_KEY_Q) == GLFW_PRESS)
    {
        angle = angle - 5;
        camAngle.x = sin(angle * 0.01745);
        camAngle.z = -cos(angle * 0.01745);
    }

    if (glfwGetKey(window, GLFW_KEY_E) == GLFW_PRESS)
    {
        angle = angle + 5;
        camAngle.x = sin(angle * 0.01745);
        camAngle.z = -cos(angle * 0.01745);
    }

    viewMatrix = glm::lookAt(glm::vec3(camPos.x, camPos.y, camPos.z),
glm::vec3(camPos.x + camAngle.x, camPos.y + camAngle.y, camPos.z + camAngle.z),
glm::vec3(0.0, 1.0, 0.0));

    glm::mat4 projectionMatrix = glm::mat4(1.0);
    projectionMatrix = glm::perspective(glm::radians(fov),
static_cast<float>(frameBufferWidth) / frameBufferHeight, nearPlane, farPlane);

    //lights
    glm::vec3 lightPos0(0.0, 0.0, 2.0);

    //update uniforms
    shader->setMat4fv(viewMatrix, "viewMatrix");
    shader->setMat4fv(projectionMatrix, "projectionMatrix");
    shader->setVec3f(lightPos0, "lightPos0");
    shader->setVec3f(camPos, "camPos");
}

void framebufferResizeCallback(GLFWwindow* window, int fbW, int fbH)
{
    glViewport(0, 0, fbW, fbH);
}

void main()
{
    //initialise glfw
    glfwInit();

    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

    //openGL version 4.6
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 6);

    glfwWindowHint(GLFW_RESIZABLE, GL_TRUE);

```

```

GLFWwindow* window = glfwCreateWindow(windowWidth, windowHeight, "Matching
Game", NULL, NULL);

glfwGetFramebufferSize(window, &frameBufferWidth, &frameBufferHeight);
glfwSetFramebufferSizeCallback(window, framebufferResizeCallback);

glfwMakeContextCurrent(window);

glewExperimental = GL_TRUE;

if (glfwInit() != GLEW_OK) {
    std::cout << "ERROR: :MAIN.CPP::GLEW_INIT_FAILED" << "\n";
    glfwTerminate();
}

//Face culling
glEnable(GL_DEPTH_TEST);
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
glFrontFace(GL_CCW);

glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);

//Shaders
Shader shader_program((char*)"vertex_shader.glsl",
(char*)"fragment_shader.glsl");

//Mesh
Mesh mesh(Shape().GetVertices(), Shape().GetNumOfVertices(),
Shape().GetIndices(), Shape().GetNumOfIndices(), glm::vec3(0.0), glm::vec3(0.0),
glm::vec3(1.0));

//Textures
glm::mat4 modelMatrix(1.0);
shader_program.setMat4fv(modelMatrix, "modelMatrix");
Texture crate("Images/crate.png", GL_TEXTURE_2D, 0); //image made by Paul, also
known as basil60. Available here https://blenderartists.org/t/modeling-aand-texturing-a-wooden-crate/574576
Texture highlight("Images/crate_highlighted.png", GL_TEXTURE_2D, 1);

//main loop
while (!glfwWindowShouldClose(window))
{
    glfwPollEvents();

    updateInput(window, &shader_program);

    //clear
    glClearColor(0.0, 0.0, 0.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
GL_STENCIL_BUFFER_BIT);

    //update projection matrix on each frame to maintain the same aspect
ratio n each object when resizing window

```

```

        glfwGetFramebufferSize(window, &frameBufferWidth, &frameBufferHeight);
        glm::mat4 projectionMatrix = glm::mat4(1.0);
        projectionMatrix = glm::perspective(glm::radians(fov),
static_cast<float>(frameBufferWidth) / frameBufferHeight, nearPlane, farPlane);
        shader_program.setMat4fv(projectionMatrix, "projectionMatrix");

        Model model1(glm::vec3(-0.5, 0.3, 0.0), shader_program, &crate, &crate,
&mesh);
        Model model2(glm::vec3(0.2, 0.3, 0.0), shader_program, &crate, &crate,
&mesh);
        Model model3(glm::vec3(0.9, 0.3, 0.0), shader_program, &crate, &crate,
&mesh);
        Model model4(glm::vec3(-0.5, -0.4, 0.0), shader_program, &crate, &crate,
&mesh);
        Model model5(glm::vec3(0.2, -0.4, 0.0), shader_program, &crate, &crate,
&mesh);
        Model model6(glm::vec3(0.9, -0.4, 0.0), shader_program, &crate, &crate,
&mesh);

        //end draw
        glfwSwapBuffers(window);
        glFlush();

        glBindVertexArray(0);
        glUseProgram(0);
        glActiveTexture(0);
        glBindTexture(GL_TEXTURE_2D, 0);
    }

    //end of program
    glfwDestroyWindow(window);
    glfwTerminate();
}

```

FRAGMENT_SHADER.GLSL

```
#version 460
```

```

in vec3 vs_position;
in vec3 vs_colour;
in vec2 vs_texcoord;
in vec3 vs_normal;

out vec4 fs_colour;

uniform sampler2D texture0;
uniform sampler2D texture1;

uniform vec3 lightPos0;
uniform vec3 camPos;

struct Material
{
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    sampler2D diffuseTexture;
    sampler2D specularTexture;
};

```

```

uniform Material material;

void main()
{
    //ambient light
    vec3 ambientLight = material.ambient;

    //diffuse light
    vec3 posToLightDirVec = normalize(lightPos0 - vs_position);
    vec3 diffuseColour = vec3(1.0, 1.0, 1.0);
    float diffuse = clamp(dot(posToLightDirVec, vs_normal), 0, 1);
    vec3 diffuseFinal = material.diffuse * diffuse;

    //specular light
    vec3 lightToPosDirVec = normalize(vs_position - lightPos0);
    vec3 reflectDirVec = normalize(reflect(lightToPosDirVec,
normalize(vs_normal)));
    vec3 posToViewDirVec = normalize(camPos - vs_position);
    float specularConst = pow(max(dot(posToViewDirVec, reflectDirVec), 0), 30);
    //the higher the power, the more direct the shine will be
    vec3 specularFinal = material.specular * specularConst;

    //attenuation

    fs_colour = texture(material.diffuseTexture, vs_texcoord) * texture(texture1,
vs_texcoord) * vec4(vs_colour, 1.0) * (vec4(ambientLight, 1.0) + vec4(diffuseFinal,
1.0) + vec4(specularFinal, 1.0f));
}

```

VERTEX_SHADER.GLSL

```

#version 460

layout (location = 0) in vec3 vertex_position;
layout (location = 1) in vec3 vertex_colour;
layout (location = 2) in vec2 vertex_texcoord;
layout (location = 3) in vec3 vertex_normal;

out vec3 vs_position;
out vec3 vs_colour;
out vec2 vs_texcoord;
out vec3 vs_normal;

uniform mat4 modelMatrix;
uniform mat4 viewMatrix;
uniform mat4 projectionMatrix;

void main()
{
    vs_position = vec4(modelMatrix * vec4(vertex_position, 1.0)).xyz;
    vs_colour = vertex_colour;
    vs_texcoord = vec2(vertex_texcoord.x, vertex_texcoord.y * -1.0f);
    vs_normal = mat3(modelMatrix) * vertex_normal;

    gl_Position = projectionMatrix * viewMatrix * modelMatrix *
vec4(vertex_position, 1.0f);
}

```

MATERIALCLASS.H

```
#include "GL/glew.h"
#include "GLFW/glfw3.h"
#include "glm/glm.hpp"

class Material {
private:
    glm::vec3 ambient;
    glm::vec3 diffuse;
    glm::vec3 specular;
    GLint diffuseTexture;
    GLint specularTexture;
public:
    //constructor
    Material(glm::vec3 ambient, glm::vec3 diffuse, glm::vec3 specular, GLint
diffuseTexture, GLint specularTexture) {
        this->ambient = ambient;
        this->diffuse = diffuse;
        this->specular = specular;
        this->diffuseTexture = diffuseTexture;
        this->specularTexture = specularTexture;
    }

    //destructor
    ~Material() {

    }

    void sendToShader(Shader& program) {
        program.setVec3f(this->ambient, "material.ambient");
        program.setVec3f(this->diffuse, "material.diffuse");
        program.setVec3f(this->specular, "material.specular");
        program.set1i(this->diffuseTexture, "material.diffuseTexture");
        program.set1i(this->specularTexture, "material.specularTexture");
    }
};
```

MESHCLASS.H

```
#include "GL/glew.h"
#include "GLFW/glfw3.h"
#include "glm/glm.hpp"
#include "glm/ext.hpp"

#include<iostream>
#include<vector>

class Mesh {
private:
    /*std::vector<Vertex> vertices;
    std::vector<GLuint> indices;*/

    unsigned numOfVertices;
    unsigned numOfIndices;

    GLuint VAO;
    GLuint VBO;
    GLuint EBO;

    glm::vec3 position;
    glm::vec3 rotation;
```

```

    glm::vec3 scale;
    glm::vec3 origin;

    glm::mat4 modelMatrix;

    void InitVAO(Vertex* vertexArray, const unsigned& numVertices, GLuint*
indexArray, const unsigned& numIndices) {
        //set variables
        this->numVertices = numVertices;
        this->numIndices = numIndices;

        //generate and bind a vertex array object
        glCreateVertexArrays(1, &this->VAO);
        glBindVertexArray(this->VAO);

        //generate, bind and send vertex buffer object data
        glGenBuffers(1, &this->VBO);
        glBindBuffer(GL_ARRAY_BUFFER, this->VBO);
        glBufferData(GL_ARRAY_BUFFER, this->numVertices * sizeof(Vertex),
vertexArray, GL_STATIC_DRAW);

        //generate, bind and send element buffer object data
        glGenBuffers(1, &this->EBO);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, this->EBO);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, this->numIndices *
sizeof(GLuint), indexArray, GL_STATIC_DRAW);

        //INPUT ASSEMBLY
        //position
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(GLvoid*)offsetof(Vertex, position));
        glEnableVertexAttribArray(0);
        //colour
        glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(GLvoid*)offsetof(Vertex, colour));
        glEnableVertexAttribArray(1);
        //texcoord
        glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(GLvoid*)offsetof(Vertex, texcoord));
        glEnableVertexAttribArray(2);
        //normal
        glVertexAttribPointer(3, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(GLvoid*)offsetof(Vertex, normal));
        glEnableVertexAttribArray(3);

        //unbind array
        glBindVertexArray(0);
    }

    void UpdateUniforms(Shader* shader) {
        shader->setMat4fv(this->modelMatrix, "modelMatrix");
    }

    void updateModelMatrix() {
        this->modelMatrix = glm::mat4(1.0);
        this->modelMatrix = glm::translate(this->modelMatrix, this->position);
        this->modelMatrix = glm::rotate(this->modelMatrix, glm::radians(this-
>rotation.x), glm::vec3(1.0, 0.0, 0.0)); //rotate along x axis
        this->modelMatrix = glm::rotate(this->modelMatrix, glm::radians(this-
>rotation.y), glm::vec3(0.0, 1.0, 0.0)); //rotate along y axis
        this->modelMatrix = glm::rotate(this->modelMatrix, glm::radians(this-
>rotation.z), glm::vec3(0.0, 0.0, 1.0)); //rotate along z axis
    }

```

```

        this->modelMatrix = glm::scale(this->modelMatrix, this->scale);
    }

public:
    //constructors
    Mesh(Vertex* vertexArray, const unsigned& numVertices, GLuint* indexArray,
const unsigned& numIndices, glm::vec3 position = glm::vec3(0.0), glm::vec3 rotation
= glm::vec3(0.0), glm::vec3 scale = glm::vec3(1.0), glm::vec3 origin = glm::vec3(0.f))
    {

        this->position = position;
        this->rotation = rotation;
        this->scale = scale;
        this->origin = origin;

        this->InitVAO(vertexArray, numVertices, indexArray, numIndices);
        this->updateModelMatrix();
    }

    //destructor
    ~Mesh() {
        glDeleteVertexArrays(1, &this->VAO);
        glDeleteBuffers(1, &this->VBO);
        glDeleteBuffers(1, &this->EBO);
    }

    void SetPosition(glm::vec3 position) {
        this->position = position;
    }

    void SetRotation(glm::vec3 scale) {
        this->scale = scale;
    }

    void SetScale(glm::vec3 position) {
        this->position = position;
    }

    void Move(glm::vec3 position) {
        this->position += position;
    }

    void Rotate(glm::vec3 rotation) {
        this->rotation += rotation;
    }

    void Scale(glm::vec3 scale) {
        this->scale += scale;
    }

    void SetOrigin(const glm::vec3 origin)
    {
        this->origin = origin;
    }

    void Update() {

    }

    void Render(Shader* shader) {
        //update uniforms
        this->updateModelMatrix();
    }

```

```

        this->UpdateUniforms(shader);

        shader->Use();

        //bind vao
        glBindVertexArray(this->VAO);

        //render
        glDrawElements(GL_TRIANGLES, this->numOfIndices, GL_UNSIGNED_INT, 0);

        glBindVertexArray(0);
        glUseProgram(0);
        glActiveTexture(0);
        glBindTexture(GL_TEXTURE_2D, 0);
    }
};

```

MODELCLASS.H

```

#include "glm/glm.hpp"

class Model {
private:
    Texture* overrideTextureDiffuse;
    Texture* overrideTextureSpecular;
    Mesh* mesh;
    glm::vec3 position;

public:
    Model(glm::vec3 position, Shader &shader, Texture* overrideTextureDiffuse,
Texture* overrideTextureSpecular, Mesh* mesh)
    {
        this->overrideTextureDiffuse = overrideTextureDiffuse;
        this->overrideTextureSpecular = overrideTextureSpecular;
        this->mesh = mesh;
        this->position = position;

        Material material(glm::vec3(0.1), glm::vec3(1.0), glm::vec3(1.0),
overrideTextureDiffuse->getTextureUnit(), overrideTextureSpecular->getTextureUnit());

        material.sendToShader(shader);

        shader.Use();

        //activate texture
        overrideTextureDiffuse->Bind();
        overrideTextureSpecular->Bind();

        mesh->SetPosition(position);

        Shader* program = &shader;
        mesh->Render(program);
    }
}

```

}; OBJECTCLASS.H

```

#include "GL/glew.h"
#include "GLFW/glfw3.h"

```

```

#include<vector>

class Shape {
private:
    std::vector<Vertex> vertices;
    std::vector<GLuint> indices;

public:
    Shape() {
        Vertex vertices[] =
        {
            //position          //texcoord          //colour
            //normals
            //front face
            glm::vec3(-0.3, 0.3, 0.0),          glm::vec3(1.0, 1.0, 1.0),
            glm::vec2(0.0, 1.0), glm::vec3(0.0, 0.0, 1.0),          glm::vec3(1.0, 1.0, 1.0),
            glm::vec2(0.0, 0.0), glm::vec3(-0.3, -0.3, 0.0),          glm::vec3(1.0, 1.0, 1.0),
            glm::vec2(0.0, 0.0), glm::vec3(0.0, 0.0, 1.0),          glm::vec3(1.0, 1.0, 1.0),
            glm::vec2(1.0, 0.0), glm::vec3(0.3, -0.3, 0.0),          glm::vec3(1.0, 1.0, 1.0),
            glm::vec2(1.0, 0.0), glm::vec3(0.0, 0.0, 1.0),          glm::vec3(1.0, 1.0, 1.0),
            glm::vec2(1.0, 1.0), glm::vec3(0.3, 0.3, 0.0),          glm::vec3(1.0, 1.0, 1.0),
            glm::vec2(1.0, 1.0), glm::vec3(0.0, 0.0, 1.0),          glm::vec3(1.0, 1.0, 1.0),

            //back face
            glm::vec3(0.3, 0.3, -0.6),          glm::vec3(1.0, 1.0, 1.0),
            glm::vec2(0.0, 1.0), glm::vec3(0.0, 0.0, -1.0),          glm::vec3(1.0, 1.0, 1.0),
            glm::vec2(0.0, 0.0), glm::vec3(0.3, -0.3, -0.6),          glm::vec3(1.0, 1.0, 1.0),
            glm::vec2(0.0, 0.0), glm::vec3(0.0, 0.0, -1.0),          glm::vec3(1.0, 1.0, 1.0),
            glm::vec2(1.0, 0.0), glm::vec3(-0.3, -0.3, -0.6),          glm::vec3(1.0, 1.0, 1.0),
            glm::vec2(1.0, 0.0), glm::vec3(0.0, 0.0, -1.0),          glm::vec3(1.0, 1.0, 1.0),
            glm::vec2(1.0, 1.0), glm::vec3(-0.3, 0.3, -0.6),          glm::vec3(1.0, 1.0, 1.0),
            glm::vec2(1.0, 1.0), glm::vec3(0.0, 0.0, -1.0),          glm::vec3(1.0, 1.0, 1.0),

            //front
            0, 1, 2,
            0, 2, 3,
            //back
            4, 5, 6,
            4, 6, 7,
            //left
            7, 6, 1,
            7, 1, 0,
            //right
            3, 2, 5,
            3, 5, 4,
            //top
            7, 0, 3,
            7, 3, 4,
            //bottom
            1, 6, 5,
            1, 5, 2
        };
        unsigned numVertices = sizeof(vertices) / sizeof(Vertex);

        GLuint indices[] =
        {
            //front
            0, 1, 2,
            0, 2, 3,
            //back
            4, 5, 6,
            4, 6, 7,
            //left
            7, 6, 1,
            7, 1, 0,
            //right
            3, 2, 5,
            3, 5, 4,
            //top
            7, 0, 3,
            7, 3, 4,
            //bottom
            1, 6, 5,
            1, 5, 2
        };
        unsigned numIndices = sizeof(indices) / sizeof(GLuint);

        this->Set(vertices, numVertices, indices, numIndices);
    }
}

```

```

void Set(const Vertex* vertices,
        const unsigned numOfVertices,
        const GLuint* indices,
        const unsigned numOfIndices)
{
    for (size_t i = 0; i < numOfVertices; i++)
    {
        this->vertices.push_back(vertices[i]);
    }

    for (size_t i = 0; i < numOfIndices; i++)
    {
        this->indices.push_back(indices[i]);
    }
}

inline Vertex* GetVertices() { return this->vertices.data(); }
inline GLuint* GetIndices() { return this->indices.data(); }
inline const unsigned GetNumOfVertices() { return this->vertices.size(); }
inline const unsigned GetNumOfIndices() { return this->indices.size(); }
};

```

SHADERCLASS.H

```

#include "GL/glew.h"
#include "GLFW/glfw3.h"
#include "glm/glm.hpp"

#include<fstream>
#include<iostream>
#include<string>

class Shader
{
private:
    GLuint id;

    std::string LoadShaderSource(char* filename) {
        std::string temp = "";
        std::string source = "";

        std::ifstream in_file;

        //load vertex shader
        in_file.open(filename);

        if (in_file.is_open())
        {
            while (std::getline(in_file, temp))
            {
                source += temp + "\n";
            }
        }
        else
        {
            std::cout << "SHADER_ERROR::COULD_NOT_OPEN_FILE: " << filename <<
"\n";
        }

        in_file.close();
    }

```

```

        return source;
    }

    GLuint LoadShader(GLenum type, char* filename) {
        char infoLog[512];
        GLint success;

        GLuint shader = glCreateShader(type);
        std::string stringSource = this->LoadShaderSource(filename).c_str();
        const GLchar* source = stringSource.c_str();
        glShaderSource(shader, 1, &source, NULL);
        glCompileShader(shader);

        //error checking the compilation
        glGetShaderiv(shader, GL_COMPILE_STATUS, &success);
        if (!success)
        {
            glGetShaderInfoLog(shader, 512, NULL, infoLog);
            std::cout << "SHADER_ERROR::COULD_NOT_COMPILE_SHADER: " <<
filename << "\n";
            std::cout << infoLog << "\n";
        }

        return shader;
    }

    void LinkProgram(GLuint vertexShader, GLuint fragmentShader) {
        char infoLog[512];
        GLint success;

        this->id = glCreateProgram();

        glAttachShader(this->id, vertexShader);

        glAttachShader(this->id, fragmentShader);

        glLinkProgram(this->id);

        //error checking the program
        glGetProgramiv(this->id, GL_LINK_STATUS, &success);
        if (!success)
        {
            glGetProgramInfoLog(this->id, 512, NULL, infoLog);
            std::cout << "SHADER_ERROR::COULD_NOT_LINK_PROGRAM" << "\n";
            std::cout << infoLog << "\n";
        }
        glUseProgram(0);
    }

public:
    //constructor
    Shader(char* vertexFile, char* fragmentFile)
    {
        GLuint vertexShader = 0;
        GLuint fragmentShader = 0;

        vertexShader = LoadShader(GL_VERTEX_SHADER, vertexFile);

        fragmentShader = LoadShader(GL_FRAGMENT_SHADER, fragmentFile);

        this->LinkProgram(vertexShader, fragmentShader);
    }

```

```

        glDeleteShader(vertexShader);
        glDeleteShader(fragmentShader);
    }

    //destructor
    ~Shader() {
        glDeleteProgram(this->id);
    }

    void Use() {
        glUseProgram(this->id);
    }

    void Unuse() {
        glUseProgram(0);
    }

    void set1i(GLint value, const GLchar* name) {
        this->Use();

        glUniform1i(glGetUniformLocation(this->id, name), value);

        this->Unuse();
    }

    void set1f(GLfloat value, const GLchar* name) {
        this->Use();

        glUniform1f(glGetUniformLocation(this->id, name), value);

        this->Unuse();
    }

    void setVec2f(glm::fvec2 value, const GLchar* name) {
        this->Use();

        glUniform2fv(glGetUniformLocation(this->id, name), 1,
glm::value_ptr(value));

        this->Unuse();
    }

    void setVec3f(glm::fvec3 value, const GLchar* name) {
        this->Use();

        glUniform3fv(glGetUniformLocation(this->id, name), 1,
glm::value_ptr(value));

        this->Unuse();
    }

    void setVec4f(glm::fvec4 value, const GLchar* name) {
        this->Use();

        glUniform4fv(glGetUniformLocation(this->id, name), 1,
glm::value_ptr(value));

        this->Unuse();
    }
}

```

```

        void setMat3fv(glm::mat3 value, const GLchar* name, GLboolean transpose =
GL_FALSE) {
            this->Use();

            glUniformMatrix3fv(glGetUniformLocation(this->id, name), 1, transpose,
glm::value_ptr(value));

            this->Unuse();
        }

        void setMat4fv(glm::mat4 value, const GLchar* name, GLboolean transpose =
GL_FALSE) {
            this->Use();

            glUniformMatrix4fv(glGetUniformLocation(this->id, name), 1, transpose,
glm::value_ptr(value));

            this->Unuse();
        }
    };

```

TEXTURECLASS.H

```

#include "GL/glew.h"
#include "GLFW/glfw3.h"
#include "glm/glm.hpp"

#include "SOIL2/SOIL2.h"

#include<iostream>
#include<string>

class Texture {
private:
    GLuint id;
    int height;
    int width;
    unsigned int type;
    GLint textureUnit;

public:
    //constructor
    Texture(const char* filename, GLenum type, GLint textureUnit) {
        this->type = type;
        this->textureUnit = textureUnit;

        unsigned char* image = SOIL_load_image(filename, &this->width, &this->
height, NULL, SOIL_LOAD_RGBA);

        glGenTextures(1, &this->id);
        glBindTexture(type, this->id);

        glTexParameteri(type, GL_TEXTURE_WRAP_S, GL_REPEAT);
        glTexParameteri(type, GL_TEXTURE_WRAP_T, GL_REPEAT);
        glTexParameteri(type, GL_TEXTURE_MAG_FILTER, GL_LINEAR_MIPMAP_LINEAR);
        glTexParameteri(type, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

        if (image)
        {
            glTexImage2D(type, 0, GL_RGBA, this->width, this->height, 0,
GL_RGBA, GL_UNSIGNED_BYTE, image);

```

```

        glGenerateMipmap(type);
    }
    else
    {
        std::cout << "TEXTURE_ERROR::TEXTURE_LOADING_FAILED: " <<
filename << "\n";
    }

    glActiveTexture(0);
    glBindTexture(type, 0);
    SOIL_free_image_data(image);

}

//destructor
~Texture() {
    glDeleteTextures(1, &this->id);
}

inline GLuint getID() const { return this->id; }

void Bind() {
    glActiveTexture(GL_TEXTURE0 + this->textureUnit);
    glBindTexture(this->type, this->id);
}

void Bind(GLint textureUnit) {
    glActiveTexture(GL_TEXTURE0 + textureUnit);
    glBindTexture(this->type, this->id);
}

void Unbind() {
    glActiveTexture(0);
    glBindTexture(this->type, 0);
}

inline GLint getTextureUnit() const { return this->textureUnit; }
};

```

VERTEXCLASS.H

```

#include "glm/glm.hpp"

struct Vertex
{
    glm::vec3 position;
    glm::vec3 colour;
    glm::vec2 texcoord;
    glm::vec3 normal;
};

```

FINAL GRADE

60/100

GENERAL COMMENTS

Instructor

Your lighting and background could be easier on the eye. The dark atmosphere you used may be better replaced with a lighter colour scheme/background. Increase diffuse and change background colour.

Multiple textures applied. Camera space applied correctly and keyboard interaction developed that moves the camera. Good. No real game play logic or interaction developed. From a SE perspective, multiple objects developed and code split into multiple files with good use of third party libraries. Well done. Your mark reflects this.

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

GRADING FORM: SMU GENERIC_GJ

MATEUSZ JADASZ

0

RESEARCH-INFORMED LITERATURE

KNOWLEDGE AND UNDERSTANDING

ANALYSIS

PRACTICAL APPLICATION AND DEP.

SKILLS FOR PROFESSIONAL PRAC.