// Multiple lights

// see update function for keys

#pragma comment(linker, "/NODEFAULTLIB:MSVCRT")

#include <iostream>

#include <fstream>

#include <string>

#include <sstream>

#include <vector>

using namespace std;

#include <GL/glew.h>

#include <GLFW/glfw3.h>

#include <GLM/glm.hpp>

#include <GLM/gtx/transform.hpp>

#include <GLI/gli.hpp>

void errorCallbackGLFW(int error, const char\* description);

void hintsGLFW();

void endProgram();

void render(GLfloat currentTime);

void update(GLfloat currentTime);

void setupRender();

void startup();

void onResizeCallback(GLFWwindow\* window, int w, int h);

void onKeyCallback(GLFWwindow\* window, int key, int scancode, int action, int mods);

void onMouseButtonCallback(GLFWwindow\* window, int button, int action, int mods);

void onMouseMoveCallback(GLFWwindow\* window, double x, double y);

void onMouseWheelCallback(GLFWwindow\* window, double xoffset, double yoffset);

void debugGL();

static void APIENTRY openGLDebugCallback(GLenum source,

GLenum type,

GLuint id,

GLenum severity,

GLsizei length,

const GLchar\* message,

const GLvoid\* userParam);

string readShader(string name);

void checkErrorShader(GLuint shader);

void readObj(string name, struct modelObject \*obj);

void readTexture(string name, GLuint texture);

// VARIABLES

float modelAngle2[] = { 0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f };

float modelAngle3[] = { 0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f };

float modelAngle4[] = { 0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f };

double timer = 0;

bool rotate1[] = { false,false,false,false,false,false,false,false,false };

GLFWwindow\* window;

int windowWidth = 640;

int windowHeight = 480;

//int windowHeight = 480;

bool running = true;

glm::mat4 proj\_matrix;

glm::vec3 modelAngle = glm::vec3(0.0f, 0.0f, 0.0f);

glm::vec3 modelDisp = glm::vec3(0.0f, 0.0f, 0.0f);

glm::vec3 cameraPosition = glm::vec3(0.0f, 0.0f, 5.0f);

glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, -1.0f);

glm::vec3 cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);

float aspect = (float)windowWidth / (float)windowHeight;

float fovy = 45.0f;

bool keyStatus[1024];

GLfloat deltaTime = 0.0f;

GLfloat lastTime = 0.0f;

// FPS camera variables

GLfloat yaw = -90.0f; // init pointing to inside

GLfloat pitch = 0.0f; // start centered

GLfloat lastX = (GLfloat)windowWidth / 2.0f; // start middle screen

GLfloat lastY = (GLfloat)windowHeight / 2.0f; // start middle screen

bool firstMouse = true;

// OBJ Variables

struct modelObject {

std::vector < glm::vec3 > out\_vertices;

std::vector < glm::vec2 > out\_uvs;

std::vector < glm::vec3 > out\_normals;

GLuint\* texture;

GLuint program;

GLuint vao;

GLuint buffer[2];

GLint mv\_location;

GLint proj\_location;

GLint tex\_location;

// extra variables for this example

GLuint matColor\_location;

GLuint lightColor\_location;

} objectModel, lightModel;

glm::vec3 \*modelPositions;

glm::vec3 \*modelRotations;

// Light

bool movingLight = true;

int movingLightNumber = 0;

glm::vec3 \*lightDisp;

//glm::vec3 lightDisp = glm::vec3(-1.0f, -1.0f, 0.0f);

glm::vec3 ia = glm::vec3(1.0f, 1.0f, 1.0f);

GLfloat ka = 5.0;

//glm::vec3 id = glm::vec3(0.93f, 0.75f, 0.32f);

glm::vec3\* id;

GLfloat kd = 1.0;

glm::vec3 is = glm::vec3(1.0f, 1.0f, 1.0f);

GLfloat ks = 1.0;

// framebuffer

GLuint framebuffer;

GLuint framebufferTexture;

GLuint depthbuffer;

GLuint displayVao;

GLuint displayBuffer[2];

std::vector < glm::vec2 > displayVertices;

std::vector < glm::vec2 > displayUvs;

GLuint displayProgram;

int main()

{

if (!glfwInit()) { // Checking for GLFW

cout << "Could not initialise GLFW...";

return 0;

}

glfwSetErrorCallback(errorCallbackGLFW); // Setup a function to catch and display all GLFW errors.

hintsGLFW(); // Setup glfw with various hints.

// Start a window using GLFW

string title = "My OpenGL Application";

// Fullscreen

//const GLFWvidmode \* mode = glfwGetVideoMode(glfwGetPrimaryMonitor());

//windowWidth = mode->width; windowHeight = mode->height;

//window = glfwCreateWindow(windowWidth, windowHeight, title.c\_str(), glfwGetPrimaryMonitor(), NULL);

// Window

window = glfwCreateWindow(windowWidth, windowHeight, title.c\_str(), NULL, NULL);

if (!window) { // Window or OpenGL context creation failed

cout << "Could not initialise GLFW...";

endProgram();

return 0;

}

glfwMakeContextCurrent(window); // making the OpenGL context current

// Start GLEW (note: always initialise GLEW after creating your window context.)

glewExperimental = GL\_TRUE; // hack: catching them all - forcing newest debug callback (glDebugMessageCallback)

GLenum errGLEW = glewInit();

if (GLEW\_OK != errGLEW) { // Problems starting GLEW?

cout << "Could not initialise GLEW...";

endProgram();

return 0;

}

debugGL(); // Setup callback to catch openGL errors.

// Setup all the message loop callbacks.

glfwSetWindowSizeCallback(window, onResizeCallback); // Set callback for resize

glfwSetKeyCallback(window, onKeyCallback); // Set Callback for keys

glfwSetMouseButtonCallback(window, onMouseButtonCallback); // Set callback for mouse click

glfwSetCursorPosCallback(window, onMouseMoveCallback); // Set callback for mouse move

glfwSetScrollCallback(window, onMouseWheelCallback); // Set callback for mouse wheel.

//glfwSetInputMode(window, GLFW\_CURSOR, GLFW\_CURSOR\_NORMAL); // Set mouse cursor. Fullscreen

glfwSetInputMode(window, GLFW\_CURSOR, GLFW\_CURSOR\_DISABLED); // Set mouse cursor FPS.

setupRender(); // setup some render variables.

startup(); // Setup all necessary information for startup (aka. load texture, shaders, models, etc).

do { // run until the window is closed

GLfloat currentTime = (GLfloat)glfwGetTime(); // retrieve timelapse

deltaTime = currentTime - lastTime; // Calculate delta time

lastTime = currentTime; // Save for next frame calculations.

glfwPollEvents(); // poll callbacks

update(currentTime); // update (physics, animation, structures, etc)

render(currentTime); // call render function.

glfwSwapBuffers(window); // swap buffers (avoid flickering and tearing)

running &= (glfwGetKey(window, GLFW\_KEY\_ESCAPE) == GLFW\_RELEASE); // exit if escape key pressed

running &= (glfwWindowShouldClose(window) != GL\_TRUE);

} while (running);

endProgram(); // Close and clean everything up...

cout << "\nPress any key to continue...\n";

cin.ignore(); cin.get(); // delay closing console to read debugging errors.

return 0;

}

void errorCallbackGLFW(int error, const char\* description) {

cout << "Error GLFW: " << description << "\n";

}

void hintsGLFW() {

glfwWindowHint(GLFW\_OPENGL\_DEBUG\_CONTEXT, GL\_TRUE); // Create context in debug mode - for debug message callback

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 4);

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 5);

}

void endProgram() {

glfwMakeContextCurrent(window); // destroys window handler

glfwTerminate(); // destroys all windows and releases resources.

// tidy heap memory

delete[] objectModel.texture;

delete[] lightModel.texture;

delete[] modelPositions;

delete[] modelRotations;

delete[] lightDisp;

delete[] id;

}

void setupRender() {

glfwSwapInterval(1); // Ony render when synced (V SYNC)

glfwWindowHint(GLFW\_OPENGL\_PROFILE, GLFW\_OPENGL\_CORE\_PROFILE);

glfwWindowHint(GLFW\_OPENGL\_FORWARD\_COMPAT, GL\_TRUE);

glfwWindowHint(GLFW\_SAMPLES, 0);

glfwWindowHint(GLFW\_STEREO, GL\_FALSE);

}

void startup() {

// Load main object model and shaders

// --------------Main Model---------------------

objectModel.program = glCreateProgram();

string vs\_text = readShader("vs\_model.glsl"); static const char\* vs\_source = vs\_text.c\_str();

GLuint vs = glCreateShader(GL\_VERTEX\_SHADER);

glShaderSource(vs, 1, &vs\_source, NULL);

glCompileShader(vs);

checkErrorShader(vs);

glAttachShader(objectModel.program, vs);

string fs\_text = readShader("fs\_model.glsl"); static const char\* fs\_source = fs\_text.c\_str();

GLuint fs = glCreateShader(GL\_FRAGMENT\_SHADER);

glShaderSource(fs, 1, &fs\_source, NULL);

glCompileShader(fs);

checkErrorShader(fs);

glAttachShader(objectModel.program, fs);

glLinkProgram(objectModel.program);

glUseProgram(objectModel.program);

readObj("RubiksObj.obj", &objectModel);

glCreateBuffers(3, objectModel.buffer); // Create a new buffer

// Store the vertices

glNamedBufferStorage(objectModel.buffer[0], objectModel.out\_vertices.size() \* sizeof(glm::vec3), &objectModel.out\_vertices[0], GL\_DYNAMIC\_STORAGE\_BIT);

glBindBuffer(GL\_ARRAY\_BUFFER, objectModel.buffer[0]); // Bind Buffer

// Store the texture coordinates

glNamedBufferStorage(objectModel.buffer[1], objectModel.out\_uvs.size() \* sizeof(glm::vec2), &objectModel.out\_uvs[0], GL\_DYNAMIC\_STORAGE\_BIT);

glBindBuffer(GL\_ARRAY\_BUFFER, objectModel.buffer[1]); // Bind Buffer

// Store the normal Vectors

glNamedBufferStorage(objectModel.buffer[2], objectModel.out\_normals.size() \* sizeof(glm::vec3), &objectModel.out\_normals[0], GL\_DYNAMIC\_STORAGE\_BIT);

glBindBuffer(GL\_ARRAY\_BUFFER, objectModel.buffer[3]); // Bind Buffer

glCreateVertexArrays(1, &objectModel.vao); // Create Vertex Array Object

// Bind vertex position buffer to the vao and format

glVertexArrayVertexBuffer(objectModel.vao, 0, objectModel.buffer[0], 0, sizeof(GLfloat) \* 3);

glVertexArrayAttribFormat(objectModel.vao, 0, 3, GL\_FLOAT, GL\_FALSE, 0);

glEnableVertexArrayAttrib(objectModel.vao, 0); // Enable Vertex Array Attribute

// Bind texture coordinate buffer to the vao and format

glVertexArrayVertexBuffer(objectModel.vao, 1, objectModel.buffer[1], 0, sizeof(GLfloat) \* 2);

glVertexArrayAttribFormat(objectModel.vao, 1, 2, GL\_FLOAT, GL\_FALSE, 0);

glEnableVertexArrayAttrib(objectModel.vao, 1); // Enable Vertex Array Attribute

// Bind the normals buffer to the vao and format

glVertexArrayVertexBuffer(objectModel.vao, 2, objectModel.buffer[2], 0, sizeof(GLfloat) \* 3);

glVertexArrayAttribFormat(objectModel.vao, 2, 3, GL\_FLOAT, GL\_FALSE, 0);

glEnableVertexArrayAttrib(objectModel.vao, 2); // Enable Vertex Array Attribute

glBindVertexArray(objectModel.vao); // Bind VertexArrayObject

objectModel.mv\_location = glGetUniformLocation(objectModel.program, "mv\_matrix");

objectModel.proj\_location = glGetUniformLocation(objectModel.program, "proj\_matrix");

objectModel.tex\_location = glGetUniformLocation(objectModel.program, "tex");

objectModel.lightColor\_location = glGetUniformLocation(objectModel.program, "ia");

objectModel.lightColor\_location = glGetUniformLocation(objectModel.program, "ka");

//--------------Light Model--------------------------

lightModel.program = glCreateProgram();

string vs\_textLight = readShader("vs\_light.glsl"); static const char\* vs\_sourceLight = vs\_textLight.c\_str();

GLuint vsLight = glCreateShader(GL\_VERTEX\_SHADER);

glShaderSource(vsLight, 1, &vs\_sourceLight, NULL);

glCompileShader(vsLight);

checkErrorShader(vsLight);

glAttachShader(lightModel.program, vsLight);

string fs\_textLight = readShader("fs\_light.glsl"); static const char\* fs\_sourceLight = fs\_textLight.c\_str();

GLuint fsLight = glCreateShader(GL\_FRAGMENT\_SHADER);

glShaderSource(fsLight, 1, &fs\_sourceLight, NULL);

glCompileShader(fsLight);

checkErrorShader(fsLight);

glAttachShader(lightModel.program, fsLight);

glLinkProgram(lightModel.program);

readObj("sphere.obj", &lightModel);

glCreateBuffers(3, lightModel.buffer); // Create a new buffer

// Store the vertices

glNamedBufferStorage(lightModel.buffer[0], lightModel.out\_vertices.size() \* sizeof(glm::vec3), &lightModel.out\_vertices[0], GL\_DYNAMIC\_STORAGE\_BIT);

glBindBuffer(GL\_ARRAY\_BUFFER, lightModel.buffer[0]); // Bind Buffer

// Store the texture coordinates

glNamedBufferStorage(lightModel.buffer[1], lightModel.out\_uvs.size() \* sizeof(glm::vec2), &lightModel.out\_uvs[0], GL\_DYNAMIC\_STORAGE\_BIT);

glBindBuffer(GL\_ARRAY\_BUFFER, lightModel.buffer[1]); // Bind Buffer

// Store the normal Vectors

glNamedBufferStorage(lightModel.buffer[2], lightModel.out\_normals.size() \* sizeof(glm::vec3), &lightModel.out\_normals[0], GL\_DYNAMIC\_STORAGE\_BIT);

glBindBuffer(GL\_ARRAY\_BUFFER, lightModel.buffer[3]); // Bind Buffer

glCreateVertexArrays(1, &lightModel.vao); // Create Vertex Array Object

// Bind vertex position buffer to the vao and format

glVertexArrayVertexBuffer(lightModel.vao, 0, lightModel.buffer[0], 0, sizeof(GLfloat) \* 3);

glVertexArrayAttribFormat(lightModel.vao, 0, 3, GL\_FLOAT, GL\_FALSE, 0);

glEnableVertexArrayAttrib(lightModel.vao, 0); // Enable Vertex Array Attribute

// Bind texture coordinate buffer to the vao and format

glVertexArrayVertexBuffer(lightModel.vao, 1, lightModel.buffer[1], 0, sizeof(GLfloat) \* 2);

glVertexArrayAttribFormat(lightModel.vao, 1, 2, GL\_FLOAT, GL\_FALSE, 0);

glEnableVertexArrayAttrib(lightModel.vao, 1); // Enable Vertex Array Attribute

// Bind the normals buffer to the vao and format

glVertexArrayVertexBuffer(lightModel.vao, 2, lightModel.buffer[2], 0, sizeof(GLfloat) \* 3);

glVertexArrayAttribFormat(lightModel.vao, 2, 3, GL\_FLOAT, GL\_FALSE, 0);

glEnableVertexArrayAttrib(lightModel.vao, 2); // Enable Vertex Array Attribute

glBindVertexArray(lightModel.vao); // Bind VertexArrayObject

lightModel.mv\_location = glGetUniformLocation(lightModel.program, "mv\_matrix");

lightModel.proj\_location = glGetUniformLocation(lightModel.program, "proj\_matrix");

lightModel.tex\_location = glGetUniformLocation(lightModel.program, "tex");

//--------------------------------------------

modelPositions = new glm::vec3[27];

modelPositions[0] = glm::vec3(0.0f, 0.0f, 0.0f);

modelPositions[1] = glm::vec3(0.4f, 0.0f, 0.0f);

modelPositions[2] = glm::vec3(-0.4f, 0.0f, 0.0f);

modelPositions[3] = glm::vec3(0.0f, 0.0f, 0.4f);

modelPositions[4] = glm::vec3(0.4f, 0.0f, 0.4f);

modelPositions[5] = glm::vec3(-0.4f, 0.0f, 0.4f);

modelPositions[6] = glm::vec3(0.0f, 0.0f, -0.4f);

modelPositions[7] = glm::vec3(0.4f, 0.0f, -0.4f);

modelPositions[8] = glm::vec3(-0.4f, 0.0f, -0.4f);

modelPositions[9] = glm::vec3(0.0f, 0.4f, 0.0f);

modelPositions[10] = glm::vec3(0.4f, 0.4f, 0.0f);

modelPositions[11] = glm::vec3(-0.4f, 0.4f, 0.0f);

modelPositions[12] = glm::vec3(0.0f, 0.4f, 0.4f);

modelPositions[13] = glm::vec3(0.4f, 0.4f, 0.4f);

modelPositions[14] = glm::vec3(-0.4f, 0.4f, 0.4f);

modelPositions[15] = glm::vec3(0.0f, 0.4f, -0.4f);

modelPositions[16] = glm::vec3(0.4f, 0.4f, -0.4f);

modelPositions[17] = glm::vec3(-0.4f, 0.4f, -0.4f);

modelPositions[18] = glm::vec3(0.0f, -0.4f, 0.0f);

modelPositions[19] = glm::vec3(0.4f, -0.4f, 0.0f);

modelPositions[20] = glm::vec3(-0.4f, -0.4f, 0.0f);

modelPositions[21] = glm::vec3(0.0f, -0.4f, 0.4f);

modelPositions[22] = glm::vec3(0.4f, -0.4f, 0.4f);

modelPositions[23] = glm::vec3(-0.4f, -0.4f, 0.4f);

modelPositions[24] = glm::vec3(0.0f, -0.4f, -0.4f);

modelPositions[25] = glm::vec3(0.4f, -0.4f, -0.4f);

modelPositions[26] = glm::vec3(-0.4f, -0.4f, -0.4f);

modelRotations = new glm::vec3[12];

modelRotations[0] = glm::vec3(0.0f, 0.0f, 0.0f);

modelRotations[1] = glm::vec3(20.0f, 10.0f, 0.0f);

modelRotations[2] = glm::vec3(30.0f, 40.0f, 0.0f);

modelRotations[3] = glm::vec3(20.0f, 50.0f, 0.0f);

modelRotations[4] = glm::vec3(40.0f, 50.0f, 0.0f);

modelRotations[5] = glm::vec3(70.0f, 60.0f, 0.0f);

modelRotations[6] = glm::vec3(80.0f, 60.0f, 0.0f);

modelRotations[7] = glm::vec3(10.0f, 20.0f, 0.0f);

modelRotations[8] = glm::vec3(70.0f, 30.0f, 0.0f);

modelRotations[9] = glm::vec3(50.0f, 40.0f, 0.0f);

modelRotations[10] = glm::vec3(60.0f, 80.0f, 0.0f);

modelRotations[11] = glm::vec3(70.0f, 80.0f, 0.0f);

lightDisp = new glm::vec3[6];

lightDisp[0] = glm::vec3(-1.5f, 0.0f, 0.0f);

lightDisp[1] = glm::vec3(0.0f, 0.0f, 1.5f);

lightDisp[2] = glm::vec3(1.5f, 0.0f, 0.0f);

lightDisp[3] = glm::vec3(0.0f, 0.0f, -1.5f);

lightDisp[4] = glm::vec3(0.0f, 1.5f, 0.0f);

lightDisp[5] = glm::vec3(0.0f, -1.5f, 0.0f);

id = new glm::vec3[4];

id[0] = glm::vec3(1.0f, 0.0f, 0.0f);

id[1] = glm::vec3(0.0f, 1.0f, 0.0f);

id[2] = glm::vec3(0.0f, 0.0f, 1.0f);

id[3] = glm::vec3(1.0f, 1.0f, 0.0f);

glFrontFace(GL\_CCW);

glCullFace(GL\_BACK);

glEnable(GL\_CULL\_FACE);

glEnable(GL\_DEPTH\_TEST);

glDepthFunc(GL\_LEQUAL);

// Calculate proj\_matrix for the first time.

aspect = (float)windowWidth / (float)windowHeight;

proj\_matrix = glm::perspective(glm::radians(fovy), aspect, 0.1f, 1000.0f);

// ----------Start Framebuffer---------------

glGenFramebuffers(1, &framebuffer);

glBindFramebuffer(GL\_FRAMEBUFFER, framebuffer);

// Create a texture for the framebuffer

glGenTextures(1, &framebufferTexture);

// Bind this texture so we can modify it.

glBindTexture(GL\_TEXTURE\_2D, framebufferTexture);

// Start the texture and give it a size but no data- of course if you resize you need to change your texture.

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGB, windowWidth, windowHeight, 0, GL\_RGB, GL\_UNSIGNED\_BYTE, 0);

// filtering needed - future lecture

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

// Depth buffer texture - Need to attach depth too otherwise depth testing will not be performed.

glGenRenderbuffers(1, &depthbuffer);

glBindRenderbuffer(GL\_RENDERBUFFER, depthbuffer);

glRenderbufferStorage(GL\_RENDERBUFFER, GL\_DEPTH\_COMPONENT, windowWidth, windowHeight);

glFramebufferRenderbuffer(GL\_FRAMEBUFFER, GL\_DEPTH\_ATTACHMENT, GL\_RENDERBUFFER, depthbuffer);

// Create a quad to display our framebuffer

displayVertices.push\_back(glm::vec2(-1.0f, 1.0f));

displayVertices.push\_back(glm::vec2(-1.0f, -1.0f));

displayVertices.push\_back(glm::vec2(1.0f, -1.0f));

displayVertices.push\_back(glm::vec2(-1.0f, 1.0f));

displayVertices.push\_back(glm::vec2(1.0f, -1.0f));

displayVertices.push\_back(glm::vec2(1.0f, 1.0f));

displayUvs.push\_back(glm::vec2(0.0f, 1.0f));

displayUvs.push\_back(glm::vec2(0.0f, 0.0f));

displayUvs.push\_back(glm::vec2(1.0f, 0.0f));

displayUvs.push\_back(glm::vec2(0.0f, 1.0f));

displayUvs.push\_back(glm::vec2(1.0f, 0.0f));

displayUvs.push\_back(glm::vec2(1.0f, 1.0f));

glCreateBuffers(2, displayBuffer); // Create a new buffer

// Store the vertices

glNamedBufferStorage(displayBuffer[0], displayVertices.size() \* sizeof(glm::vec2), &displayVertices[0], GL\_DYNAMIC\_STORAGE\_BIT);

glBindBuffer(GL\_ARRAY\_BUFFER, displayBuffer[0]); // Bind Buffer

// Store the texture coordinates

glNamedBufferStorage(displayBuffer[1], displayUvs.size() \* sizeof(glm::vec2), &displayUvs[0], GL\_DYNAMIC\_STORAGE\_BIT);

glBindBuffer(GL\_ARRAY\_BUFFER, displayBuffer[1]); // Bind Buffer

glCreateVertexArrays(1, &displayVao); // Create Vertex Array Object

// Bind vertex position buffer to the vao and format

glVertexArrayVertexBuffer(displayVao, 0, displayBuffer[0], 0, sizeof(GLfloat) \* 2);

glVertexArrayAttribFormat(displayVao, 0, 2, GL\_FLOAT, GL\_FALSE, 0);

glEnableVertexArrayAttrib(displayVao, 0); // Enable Vertex Array Attribute

// Bind texture coordinate buffer to the vao and format

glVertexArrayVertexBuffer(displayVao, 1, displayBuffer[1], 0, sizeof(GLfloat) \* 2);

glVertexArrayAttribFormat(displayVao, 1, 2, GL\_FLOAT, GL\_FALSE, 0);

glEnableVertexArrayAttrib(displayVao, 1); // Enable Vertex Array Attribute

glBindVertexArray(displayVao); // Bind VertexArrayObject

//load shaders

displayProgram = glCreateProgram();

string dvs\_text = readShader("vs\_display.glsl"); static const char\* dvs\_source = dvs\_text.c\_str();

GLuint dvs = glCreateShader(GL\_VERTEX\_SHADER);

glShaderSource(dvs, 1, &dvs\_source, NULL);

glCompileShader(dvs);

checkErrorShader(dvs);

glAttachShader(displayProgram, dvs);

string dfs\_text = readShader("fs\_display.glsl"); static const char\* dfs\_source = dfs\_text.c\_str();

GLuint dfs = glCreateShader(GL\_FRAGMENT\_SHADER);

glShaderSource(dfs, 1, &dfs\_source, NULL);

glCompileShader(dfs);

checkErrorShader(dfs);

glAttachShader(displayProgram, dfs);

glLinkProgram(displayProgram);

glUseProgram(displayProgram);

}

void update(GLfloat currentTime) {

// calculate movement

GLfloat cameraSpeed = 1.0f \* deltaTime;

if (keyStatus[GLFW\_KEY\_W]) cameraPosition += cameraSpeed \* cameraFront;

if (keyStatus[GLFW\_KEY\_S]) cameraPosition -= cameraSpeed \* cameraFront;

if (keyStatus[GLFW\_KEY\_A]) cameraPosition -= glm::normalize(glm::cross(cameraFront, cameraUp)) \* cameraSpeed;

if (keyStatus[GLFW\_KEY\_D]) cameraPosition += glm::normalize(glm::cross(cameraFront, cameraUp)) \* cameraSpeed;

if (keyStatus[GLFW\_KEY\_L] && (movingLight == false)) {

cout << "Change mode to moving light...\n";

movingLight = true;

}

if (keyStatus[GLFW\_KEY\_M] && (movingLight == true)) {

cout << "Change mode to moving object...\n";

movingLight = false;

}

if (keyStatus[GLFW\_KEY\_0] && (movingLight == true)) {

movingLightNumber = 0; cout << "Moving light 0...\n";

}

if (keyStatus[GLFW\_KEY\_1] && (movingLight == true)) {

movingLightNumber = 1; cout << "Moving light 1...\n";

}

if (keyStatus[GLFW\_KEY\_2] && (movingLight == true)) {

movingLightNumber = 2; cout << "Moving light 2...\n";

}

if (keyStatus[GLFW\_KEY\_3] && (movingLight == true)) {

movingLightNumber = 3; cout << "Moving light 3...\n";

}

if (keyStatus[GLFW\_KEY\_Y])

rotate1[0] = true;

if (keyStatus[GLFW\_KEY\_U])

rotate1[1] = true;

if (keyStatus[GLFW\_KEY\_I])

rotate1[2] = true;

if (keyStatus[GLFW\_KEY\_H])

rotate1[3] = true;

if (keyStatus[GLFW\_KEY\_J])

rotate1[4] = true;

if (keyStatus[GLFW\_KEY\_K])

rotate1[5] = true;

if (keyStatus[GLFW\_KEY\_B])

rotate1[6] = true;

if (keyStatus[GLFW\_KEY\_N])

rotate1[7] = true;

if (keyStatus[GLFW\_KEY\_M])

rotate1[8] = true;

if (rotate1[0] == true)

{

modelAngle2[1] += 0.1;

modelAngle2[4] += 0.1;

modelAngle2[7] += 0.1;

modelAngle2[10] += 0.1;

modelAngle2[13] += 0.1;

modelAngle2[16] += 0.1;

modelAngle2[19] += 0.1;

modelAngle2[22] += 0.1;

modelAngle2[25] += 0.1;

timer++;

if (timer >= 63 && rotate1[0] == true)

{

rotate1[0] = false;

timer = 0;

modelAngle2[1] = 0.0;

modelAngle2[4] = 0.0;

modelAngle2[7] = 0.0;

modelAngle2[10] = 0.0;

modelAngle2[13] = 0.0;

modelAngle2[16] = 0.0;

modelAngle2[19] = 0.0;

modelAngle2[22] = 0.0;

modelAngle2[25] = 0.0;

}

}

if (rotate1[1] == true)

{

modelAngle2[0] += 0.1;

modelAngle2[3] += 0.1;

modelAngle2[6] += 0.1;

modelAngle2[9] += 0.1;

modelAngle2[12] += 0.1;

modelAngle2[15] += 0.1;

modelAngle2[18] += 0.1;

modelAngle2[21] += 0.1;

modelAngle2[24] += 0.1;

timer++;

if (timer >= 63 && rotate1[1] == true)

{

rotate1[1] = false;

timer = 0;

modelAngle2[0] = 0.0;

modelAngle2[3] = 0.0;

modelAngle2[6] = 0.0;

modelAngle2[9] = 0.0;

modelAngle2[12] = 0.0;

modelAngle2[15] = 0.0;

modelAngle2[18] = 0.0;

modelAngle2[21] = 0.0;

modelAngle2[24] = 0.0;

}

}

if (rotate1[2] == true)

{

modelAngle2[2] += 0.1;

modelAngle2[5] += 0.1;

modelAngle2[8] += 0.1;

modelAngle2[11] += 0.1;

modelAngle2[14] += 0.1;

modelAngle2[17] += 0.1;

modelAngle2[20] += 0.1;

modelAngle2[23] += 0.1;

modelAngle2[26] += 0.1;

timer++;

if (timer >= 63 && rotate1[2] == true)

{

rotate1[2] = false;

timer = 0;

modelAngle2[2] = 0.0;

modelAngle2[5] = 0.0;

modelAngle2[8] = 0.0;

modelAngle2[11] = 0.0;

modelAngle2[14] = 0.0;

modelAngle2[17] = 0.0;

modelAngle2[20] = 0.0;

modelAngle2[23] = 0.0;

modelAngle2[26] = 0.0;

}

}

if (rotate1[3] == true)

{

modelAngle3[18] += 0.1;

modelAngle3[19] += 0.1;

modelAngle3[20] += 0.1;

modelAngle3[21] += 0.1;

modelAngle3[22] += 0.1;

modelAngle3[23] += 0.1;

modelAngle3[24] += 0.1;

modelAngle3[25] += 0.1;

modelAngle3[26] += 0.1;

timer++;

if (timer >= 63 && rotate1[3] == true)

{

rotate1[3] = false;

timer = 0;

modelAngle3[18] = 0.0;

modelAngle3[19] = 0.0;

modelAngle3[20] = 0.0;

modelAngle3[21] = 0.0;

modelAngle3[22] = 0.0;

modelAngle3[23] = 0.0;

modelAngle3[24] = 0.0;

modelAngle3[25] = 0.0;

modelAngle3[26] = 0.0;

}

}

if (rotate1[4] == true)

{

modelAngle3[0] += 0.1;

modelAngle3[1] += 0.1;

modelAngle3[2] += 0.1;

modelAngle3[3] += 0.1;

modelAngle3[4] += 0.1;

modelAngle3[5] += 0.1;

modelAngle3[6] += 0.1;

modelAngle3[7] += 0.1;

modelAngle3[8] += 0.1;

timer++;

if (timer >= 63 && rotate1[4] == true)

{

rotate1[4] = false;

timer = 0;

modelAngle3[0] = 0.0;

modelAngle3[1] = 0.0;

modelAngle3[2] = 0.0;

modelAngle3[3] = 0.0;

modelAngle3[4] = 0.0;

modelAngle3[5] = 0.0;

modelAngle3[6] = 0.0;

modelAngle3[7] = 0.0;

modelAngle3[8] = 0.0;

}

}

if (rotate1[5] == true)

{

modelAngle3[9] += 0.1;

modelAngle3[10] += 0.1;

modelAngle3[11] += 0.1;

modelAngle3[12] += 0.1;

modelAngle3[13] += 0.1;

modelAngle3[14] += 0.1;

modelAngle3[15] += 0.1;

modelAngle3[16] += 0.1;

modelAngle3[17] += 0.1;

timer++;

if (timer >= 63 && rotate1[5] == true)

{

rotate1[5] = false;

timer = 0;

modelAngle3[9] = 0.0;

modelAngle3[10] = 0.0;

modelAngle3[11] = 0.0;

modelAngle3[12] = 0.0;

modelAngle3[13] = 0.0;

modelAngle3[14] = 0.0;

modelAngle3[15] = 0.0;

modelAngle3[16] = 0.0;

modelAngle3[17] = 0.0;

}

}

if (rotate1[6] == true)

{

modelAngle4[3] += 0.1;

modelAngle4[4] += 0.1;

modelAngle4[5] += 0.1;

modelAngle4[12] += 0.1;

modelAngle4[13] += 0.1;

modelAngle4[14] += 0.1;

modelAngle4[21] += 0.1;

modelAngle4[22] += 0.1;

modelAngle4[23] += 0.1;

timer++;

if (timer >= 63 && rotate1[6] == true)

{

rotate1[6] = false;

timer = 0;

modelAngle4[3] = 0.0;

modelAngle4[4] = 0.0;

modelAngle4[5] = 0.0;

modelAngle4[12] = 0.0;

modelAngle4[13] = 0.0;

modelAngle4[14] = 0.0;

modelAngle4[21] = 0.0;

modelAngle4[22] = 0.0;

modelAngle4[23] = 0.0;

}

}

if (rotate1[7] == true)

{

modelAngle4[0] += 0.1;

modelAngle4[1] += 0.1;

modelAngle4[2] += 0.1;

modelAngle4[9] += 0.1;

modelAngle4[10] += 0.1;

modelAngle4[11] += 0.1;

modelAngle4[18] += 0.1;

modelAngle4[19] += 0.1;

modelAngle4[20] += 0.1;

timer++;

if (timer >= 63 && rotate1[7] == true)

{

rotate1[7] = false;

timer = 0;

modelAngle4[0] = 0.0;

modelAngle4[1] = 0.0;

modelAngle4[2] = 0.0;

modelAngle4[9] = 0.0;

modelAngle4[10] = 0.0;

modelAngle4[11] = 0.0;

modelAngle4[18] = 0.0;

modelAngle4[19] = 0.0;

modelAngle4[20] = 0.0;

}

}

if (rotate1[8] == true)

{

modelAngle4[6] += 0.1;

modelAngle4[7] += 0.1;

modelAngle4[8] += 0.1;

modelAngle4[15] += 0.1;

modelAngle4[16] += 0.1;

modelAngle4[17] += 0.1;

modelAngle4[24] += 0.1;

modelAngle4[25] += 0.1;

modelAngle4[26] += 0.1;

timer++;

if (timer >= 63 && rotate1[8] == true)

{

rotate1[8] = false;

timer = 0;

modelAngle4[6] = 0.0;

modelAngle4[7] = 0.0;

modelAngle4[8] = 0.0;

modelAngle4[15] = 0.0;

modelAngle4[16] = 0.0;

modelAngle4[17] = 0.0;

modelAngle4[24] = 0.0;

modelAngle4[25] = 0.0;

modelAngle4[26] = 0.0;

}

}

if (movingLight == false) { // moving object rotation and z displacement

if (keyStatus[GLFW\_KEY\_LEFT]) modelAngle.y += 0.05f;

if (keyStatus[GLFW\_KEY\_RIGHT]) modelAngle.y -= 0.05f;

if (keyStatus[GLFW\_KEY\_UP]) modelAngle.x += 0.05f;

if (keyStatus[GLFW\_KEY\_DOWN]) modelAngle.x -= 0.05f;

if (keyStatus[GLFW\_KEY\_KP\_ADD]) modelDisp.z += 0.10f;

if (keyStatus[GLFW\_KEY\_KP\_SUBTRACT]) modelDisp.z -= 0.10f;

}

else { // moving light displacement x y z

if (keyStatus[GLFW\_KEY\_LEFT]) lightDisp[movingLightNumber].x -= 0.05f;

if (keyStatus[GLFW\_KEY\_RIGHT]) lightDisp[movingLightNumber].x += 0.05f;

if (keyStatus[GLFW\_KEY\_UP]) lightDisp[movingLightNumber].y += 0.05f;

if (keyStatus[GLFW\_KEY\_DOWN]) lightDisp[movingLightNumber].y -= 0.05f;

if (keyStatus[GLFW\_KEY\_KP\_ADD]) lightDisp[movingLightNumber].z += 0.05f;

if (keyStatus[GLFW\_KEY\_KP\_SUBTRACT]) lightDisp[movingLightNumber].z -= 0.05f;

}

}

void render(GLfloat currentTime) {

//==============First Pass====================

//---Render framebuffer to texture

glBindFramebuffer(GL\_FRAMEBUFFER, framebuffer);

glFramebufferTexture2D(GL\_FRAMEBUFFER, GL\_COLOR\_ATTACHMENT0, GL\_TEXTURE\_2D, framebufferTexture, 0);

glViewport(0, 0, windowWidth, windowHeight);

// Clear colour buffer

glm::vec4 backgroundColor = glm::vec4(0.2f, 0.2f, 0.2f, 0.2f); glClearBufferfv(GL\_COLOR, 0, &backgroundColor[0]);

// Clear Deep buffer

static const GLfloat one = 1.0f; glClearBufferfv(GL\_DEPTH, 0, &one);

// Enable blend

glEnable(GL\_BLEND);

glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);

// ----------draw main model------------

glUseProgram(objectModel.program);

glBindVertexArray(objectModel.vao);

glUniformMatrix4fv(objectModel.proj\_location, 1, GL\_FALSE, &proj\_matrix[0][0]);

glUniform4f(glGetUniformLocation(objectModel.program, "viewPosition"), cameraPosition.x, cameraPosition.y, cameraPosition.z, 1.0);

glUniform4f(glGetUniformLocation(objectModel.program, "ia"), ia.r, ia.g, ia.b, 1.0);

glUniform1f(glGetUniformLocation(objectModel.program, "ka"), ka);

glUniform1f(glGetUniformLocation(objectModel.program, "kd"), 1.0f);

glUniform4f(glGetUniformLocation(objectModel.program, "is"), is.r, is.g, is.b, 1.0);

glUniform1f(glGetUniformLocation(objectModel.program, "ks"), 1.0f);

glUniform1f(glGetUniformLocation(objectModel.program, "shininess"), 32.0f);

glUniform1f(glGetUniformLocation(objectModel.program, "lightConstant"), 1.0f);

glUniform1f(glGetUniformLocation(objectModel.program, "lightLinear"), 0.7f);

glUniform1f(glGetUniformLocation(objectModel.program, "lightQuadratic"), 1.8f);

for (int i = 0; i < 4; i++) {

glUniform4f(glGetUniformLocation(objectModel.program, ("lights[" + to\_string(i) + "].lightPosition").c\_str()), lightDisp[i].x, lightDisp[i].y, lightDisp[i].z, 1.0f);

glUniform4f(glGetUniformLocation(objectModel.program, ("lights[" + to\_string(i) + "].id").c\_str()), id[i].r, id[i].g, id[i].b, 1.0f);

}

//glUniform4f(glGetUniformLocation(objectModel.program, "lightPosition"), cameraPosition.x, cameraPosition.y, cameraPosition.z, 1.0);

//glUniform4f(glGetUniformLocation(objectModel.program, "lightSpotDirection"), cameraFront.x, cameraFront.y, cameraFront.z, 0.0);

//glUniform1f(glGetUniformLocation(objectModel.program, "lightSpotCutOff"), glm::cos(glm::radians(12.5f)));

//glUniform1f(glGetUniformLocation(objectModel.program, "lightSpotOuterCutOff"), glm::cos(glm::radians(15.0f)));

// Bind textures and samplers - using 0 as I know there is only one texture - need to extend.

glActiveTexture(GL\_TEXTURE0);

glBindTexture(GL\_TEXTURE\_2D, objectModel.texture[0]);

glUniform1i(objectModel.tex\_location, 0);

glm::mat4 viewMatrix = glm::lookAt(cameraPosition, // eye

cameraPosition + cameraFront, // centre

cameraUp); // up

for (int i = 0; i < 27; i++) {

glm::mat4 modelMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f, 0.0f));// modelDisp.z));

modelMatrix = glm::rotate(modelMatrix, modelAngle2[i], glm::vec3(1.0f, 0.0f, 0.0f));

modelMatrix = glm::rotate(modelMatrix, modelAngle3[i], glm::vec3(0.0f, 1.0f, 0.0f));

modelMatrix = glm::rotate(modelMatrix, modelAngle4[i], glm::vec3(0.0f, 0.0f, 1.0f));

modelMatrix = glm::translate(modelMatrix, modelPositions[i]);

modelMatrix = glm::rotate(modelMatrix, modelAngle.x + modelRotations[0].x, glm::vec3(1.0f, 0.0f, 0.0f));

modelMatrix = glm::rotate(modelMatrix, modelAngle.y + modelRotations[0].y, glm::vec3(0.0f, 1.0f, 0.0f));

modelMatrix = glm::scale(modelMatrix, glm::vec3(0.2f, 0.2f, 0.2f));

glm::mat4 mv\_matrix = viewMatrix \* modelMatrix;

glUniformMatrix4fv(glGetUniformLocation(objectModel.program, "model\_matrix"), 1, GL\_FALSE, &modelMatrix[0][0]);

glUniformMatrix4fv(glGetUniformLocation(objectModel.program, "view\_matrix"), 1, GL\_FALSE, &viewMatrix[0][0]);

glDrawArrays(GL\_TRIANGLES, 0, objectModel.out\_vertices.size());

}

// ----------draw light------------

glUseProgram(lightModel.program);

glBindVertexArray(lightModel.vao);

glUniformMatrix4fv(lightModel.proj\_location, 1, GL\_FALSE, &proj\_matrix[0][0]);

// Bind textures and samplers - using 0 as I know there is only one texture - need to extend.

glActiveTexture(GL\_TEXTURE0);

glBindTexture(GL\_TEXTURE\_2D, lightModel.texture[0]);

glUniform1i(lightModel.tex\_location, 0);

for (int i = 0; i < 6; i++) {

glm::mat4 modelMatrixLight = glm::translate(glm::mat4(1.0f), glm::vec3(lightDisp[i].x, lightDisp[i].y, lightDisp[i].z));

modelMatrixLight = glm::scale(modelMatrixLight, glm::vec3(0.2f, 0.2f, 0.2f));

glm::mat4 mv\_matrixLight = viewMatrix \* modelMatrixLight;

glUniformMatrix4fv(lightModel.mv\_location, 1, GL\_FALSE, &mv\_matrixLight[0][0]);

glDrawArrays(GL\_TRIANGLES, 0, lightModel.out\_vertices.size());

}

//==============Second Pass===================

glBindFramebuffer(GL\_FRAMEBUFFER, 0); // Disable rendering framebuffer to texture - aka render normally.

glClearColor(1.0f, 1.0f, 1.0f, 1.0f);

glClear(GL\_COLOR\_BUFFER\_BIT);

glDisable(GL\_DEPTH\_TEST); //not needed as we are just displaying a single quad

glUseProgram(displayProgram);

glBindVertexArray(displayVao);

glActiveTexture(GL\_TEXTURE0);

glBindTexture(GL\_TEXTURE\_2D, framebufferTexture);

glDrawArrays(GL\_TRIANGLES, 0, 6);

glBindVertexArray(0);

}

void onResizeCallback(GLFWwindow\* window, int w, int h) {

windowWidth = w;

windowHeight = h;

aspect = (float)w / (float)h;

proj\_matrix = glm::perspective(glm::radians(fovy), aspect, 0.1f, 1000.0f);

}

void onKeyCallback(GLFWwindow\* window, int key, int scancode, int action, int mods) {

if (action == GLFW\_PRESS) keyStatus[key] = true;

else if (action == GLFW\_RELEASE) keyStatus[key] = false;

if (key == GLFW\_KEY\_ESCAPE && action == GLFW\_PRESS)

glfwSetWindowShouldClose(window, GLFW\_TRUE);

}

void onMouseButtonCallback(GLFWwindow\* window, int button, int action, int mods) {

}

void onMouseMoveCallback(GLFWwindow\* window, double x, double y) {

int mouseX = static\_cast<int>(x);

int mouseY = static\_cast<int>(y);

if (firstMouse) {

lastX = (GLfloat)mouseX; lastY = (GLfloat)mouseY; firstMouse = false;

}

GLfloat xoffset = mouseX - lastX;

GLfloat yoffset = lastY - mouseY; // Reversed

lastX = (GLfloat)mouseX; lastY = (GLfloat)mouseY;

GLfloat sensitivity = 0.05f;

xoffset \*= sensitivity; yoffset \*= sensitivity;

yaw += xoffset; pitch += yoffset;

// check for pitch out of bounds otherwise screen gets flipped

if (pitch > 89.0f) pitch = 89.0f;

if (pitch < -89.0f) pitch = -89.0f;

glm::vec3 front;

front.x = cos(glm::radians(yaw)) \* cos(glm::radians(pitch));

front.y = sin(glm::radians(pitch));

front.z = sin(glm::radians(yaw)) \* cos(glm::radians(pitch));

cameraFront = glm::normalize(front);

}

static void onMouseWheelCallback(GLFWwindow\* window, double xoffset, double yoffset) {

int yoffsetInt = static\_cast<int>(yoffset);

fovy += yoffsetInt;

if (fovy >= 1.0f && fovy <= 45.0f) fovy -= (GLfloat)yoffset;

if (fovy <= 1.0f) fovy = 1.0f;

if (fovy >= 45.0f) fovy = 45.0f;

proj\_matrix = glm::perspective(glm::radians(fovy), aspect, 0.1f, 1000.0f);

}

void debugGL() {

//Output some debugging information

cout << "VENDOR: " << (char \*)glGetString(GL\_VENDOR) << endl;

cout << "VERSION: " << (char \*)glGetString(GL\_VERSION) << endl;

cout << "RENDERER: " << (char \*)glGetString(GL\_RENDERER) << endl;

// Enable Opengl Debug

//glEnable(GL\_DEBUG\_OUTPUT);

glEnable(GL\_DEBUG\_OUTPUT\_SYNCHRONOUS);

glDebugMessageCallback((GLDEBUGPROC)openGLDebugCallback, nullptr);

glDebugMessageControl(GL\_DONT\_CARE, GL\_DONT\_CARE, GL\_DONT\_CARE, 0, NULL, true);

}

static void APIENTRY openGLDebugCallback(GLenum source,

GLenum type,

GLuint id,

GLenum severity,

GLsizei length,

const GLchar\* message,

const GLvoid\* userParam) {

cout << "---------------------opengl-callback------------" << endl;

cout << "Message: " << message << endl;

cout << "type: ";

switch (type) {

case GL\_DEBUG\_TYPE\_ERROR:

cout << "ERROR";

break;

case GL\_DEBUG\_TYPE\_DEPRECATED\_BEHAVIOR:

cout << "DEPRECATED\_BEHAVIOR";

break;

case GL\_DEBUG\_TYPE\_UNDEFINED\_BEHAVIOR:

cout << "UNDEFINED\_BEHAVIOR";

break;

case GL\_DEBUG\_TYPE\_PORTABILITY:

cout << "PORTABILITY";

break;

case GL\_DEBUG\_TYPE\_PERFORMANCE:

cout << "PERFORMANCE";

break;

case GL\_DEBUG\_TYPE\_OTHER:

cout << "OTHER";

break;

}

cout << " --- ";

cout << "id: " << id << " --- ";

cout << "severity: ";

switch (severity) {

case GL\_DEBUG\_SEVERITY\_LOW:

cout << "LOW";

break;

case GL\_DEBUG\_SEVERITY\_MEDIUM:

cout << "MEDIUM";

break;

case GL\_DEBUG\_SEVERITY\_HIGH:

cout << "HIGH";

break;

case GL\_DEBUG\_SEVERITY\_NOTIFICATION:

cout << "NOTIFICATION";

}

cout << endl;

cout << "-----------------------------------------" << endl;

}

string readShader(string name) {

string vs\_text;

std::ifstream vs\_file(name);

string vs\_line;

if (vs\_file.is\_open()) {

while (getline(vs\_file, vs\_line)) {

vs\_text += vs\_line;

vs\_text += '\n';

}

vs\_file.close();

}

return vs\_text;

}

void checkErrorShader(GLuint shader) {

// Get log lenght

GLint maxLength;

glGetShaderiv(shader, GL\_INFO\_LOG\_LENGTH, &maxLength);

// Init a string for it

std::vector<GLchar> errorLog(maxLength);

if (maxLength > 0) {

// Get the log file

glGetShaderInfoLog(shader, maxLength, &maxLength, &errorLog[0]);

cout << "--------------Shader compilation error-------------\n";

cout << errorLog.data();

}

}

void readObj(string name, struct modelObject \*obj) {

cout << "Loading model " << name << "\n";

std::vector< unsigned int > vertexIndices, uvIndices, normalIndices;

std::vector< string > materials, textures;

std::vector< glm::vec3 > obj\_vertices;

std::vector< glm::vec2 > obj\_uvs;

std::vector< glm::vec3 > obj\_normals;

std::ifstream dataFile(name);

string rawData; // store the raw data.

int countLines = 0;

if (dataFile.is\_open()) {

string dataLineRaw;

while (getline(dataFile, dataLineRaw)) {

rawData += dataLineRaw;

rawData += "\n";

countLines++;

}

dataFile.close();

}

cout << "Finished reading model file " << name << "\n";

istringstream rawDataStream(rawData);

string dataLine;

int linesDone = 0;

while (std::getline(rawDataStream, dataLine)) {

if (dataLine.find("v ") != string::npos) { // does this line have a vector?

glm::vec3 vertex;

int foundStart = dataLine.find(" "); int foundEnd = dataLine.find(" ", foundStart + 1);

vertex.x = stof(dataLine.substr(foundStart, foundEnd - foundStart));

foundStart = foundEnd; foundEnd = dataLine.find(" ", foundStart + 1);

vertex.y = stof(dataLine.substr(foundStart, foundEnd - foundStart));

foundStart = foundEnd; foundEnd = dataLine.find(" ", foundStart + 1);

vertex.z = stof(dataLine.substr(foundStart, foundEnd - foundStart));

obj\_vertices.push\_back(vertex);

}

else if (dataLine.find("vt ") != string::npos) { // does this line have a uv coordinates?

glm::vec2 uv;

int foundStart = dataLine.find(" "); int foundEnd = dataLine.find(" ", foundStart + 1);

uv.x = stof(dataLine.substr(foundStart, foundEnd - foundStart));

foundStart = foundEnd; foundEnd = dataLine.find(" ", foundStart + 1);

uv.y = stof(dataLine.substr(foundStart, foundEnd - foundStart));

obj\_uvs.push\_back(uv);

}

else if (dataLine.find("vn ") != string::npos) { // does this line have a normal coordinates?

glm::vec3 normal;

int foundStart = dataLine.find(" "); int foundEnd = dataLine.find(" ", foundStart + 1);

normal.x = stof(dataLine.substr(foundStart, foundEnd - foundStart));

foundStart = foundEnd; foundEnd = dataLine.find(" ", foundStart + 1);

normal.y = stof(dataLine.substr(foundStart, foundEnd - foundStart));

foundStart = foundEnd; foundEnd = dataLine.find(" ", foundStart + 1);

normal.z = stof(dataLine.substr(foundStart, foundEnd - foundStart));

obj\_normals.push\_back(normal);

}

else if (dataLine.find("f ") != string::npos) { // does this line defines a triangle face?

string parts[3];

int foundStart = dataLine.find(" "); int foundEnd = dataLine.find(" ", foundStart + 1);

parts[0] = dataLine.substr(foundStart + 1, foundEnd - foundStart - 1);

foundStart = foundEnd; foundEnd = dataLine.find(" ", foundStart + 1);

parts[1] = dataLine.substr(foundStart + 1, foundEnd - foundStart - 1);

foundStart = foundEnd; foundEnd = dataLine.find(" ", foundStart + 1);

parts[2] = dataLine.substr(foundStart + 1, foundEnd - foundStart - 1);

for (int i = 0; i < 3; i++) { // for each part

vertexIndices.push\_back(stoul(parts[i].substr(0, parts[i].find("/"))));

int firstSlash = parts[i].find("/"); int secondSlash = parts[i].find("/", firstSlash + 1);

if ((firstSlash + 1) != (secondSlash)) { // there are texture coordinates.

uvIndices.push\_back(stoul(parts[i].substr(firstSlash + 1, secondSlash - firstSlash + 1)));

}

normalIndices.push\_back(stoul(parts[i].substr(secondSlash + 1)));

}

}

else if (dataLine.find("mtllib ") != string::npos) { // does this object have a material?

materials.push\_back(dataLine.substr(dataLine.find(" ") + 1));

}

linesDone++;

if (linesDone % 50000 == 0) {

cout << "Parsed " << linesDone << " of " << countLines << " from model...\n";

}

}

// Double check here in which coordinate system you exported your models - and flip or not the vertices...

/\*for (unsigned int i = 0; i < vertexIndices.size(); i += 3) {

(\*obj).out\_vertices.push\_back(obj\_vertices[vertexIndices[i+2] - 1]);

(\*obj).out\_normals.push\_back(obj\_normals[normalIndices[i+2] - 1]);

(\*obj).out\_uvs.push\_back(obj\_uvs[uvIndices[i+2] - 1]);

(\*obj).out\_vertices.push\_back(obj\_vertices[vertexIndices[i+1] - 1]);

(\*obj).out\_normals.push\_back(obj\_normals[normalIndices[i+1] - 1]);

(\*obj).out\_uvs.push\_back(obj\_uvs[uvIndices[i + 1] - 1]);

(\*obj).out\_vertices.push\_back(obj\_vertices[vertexIndices[i] - 1]);

(\*obj).out\_normals.push\_back(obj\_normals[normalIndices[i] - 1]);

(\*obj).out\_uvs.push\_back(obj\_uvs[uvIndices[i + 0] - 1]);

}\*/

for (unsigned int i = 0; i < vertexIndices.size(); i++) {

(\*obj).out\_vertices.push\_back(obj\_vertices[vertexIndices[i] - 1]);

(\*obj).out\_normals.push\_back(obj\_normals[normalIndices[i] - 1]);

(\*obj).out\_uvs.push\_back(obj\_uvs[uvIndices[i] - 1]);

}

// Load Materials

for (unsigned int i = 0; i < materials.size(); i++) {

std::ifstream dataFileMat(materials[i]);

string rawDataMat; // store the raw data.

int countLinesMat = 0;

if (dataFileMat.is\_open()) {

string dataLineRawMat;

while (getline(dataFileMat, dataLineRawMat)) {

rawDataMat += dataLineRawMat;

rawDataMat += "\n";

}

dataFileMat.close();

}

istringstream rawDataStreamMat(rawDataMat);

string dataLineMat;

while (std::getline(rawDataStreamMat, dataLineMat)) {

if (dataLineMat.find("map\_Kd ") != string::npos) { // does this line have a texture map?

textures.push\_back(dataLineMat.substr(dataLineMat.find(" ") + 1));

}

}

}

(\*obj).texture = new GLuint[textures.size()]; // Warning possible memory leak here - there is a new here, where is your delete?

glCreateTextures(GL\_TEXTURE\_2D, textures.size(), (\*obj).texture);

for (unsigned int i = 0; i < textures.size(); i++) {

readTexture(textures[i], (\*obj).texture[i]);

}

cout << "done";

}

void readTexture(string name, GLuint textureName) {

gli::texture tex = gli::load(name);

if (tex.empty()) {

cout << "Unable to load file " << name;

}

gli::gl texGl(gli::gl::PROFILE\_GL33);

gli::gl::format const texFormat = texGl.translate(tex.format(), tex.swizzles());

//GLenum texTarget = texGl.translate(tex.target());

// Load and create a texture

glBindTexture(GL\_TEXTURE\_2D, textureName); // All upcoming operations now have effect on this texture object

glm::tvec3<GLsizei> const texExtent(tex.extent());

GLsizei const texFaceTotal = static\_cast<GLsizei>(tex.layers() \* tex.faces());

// Note: This only loads GL\_TEXTURE\_2D - for the complete code please visit >> http://gli.g-truc.net/

glTexStorage2D(GL\_TEXTURE\_2D, static\_cast<GLint>(tex.levels()), texFormat.Internal, texExtent.x, texExtent.y);

for (std::size\_t Layer = 0; Layer < tex.layers(); ++Layer) {

for (std::size\_t Face = 0; Face < tex.faces(); ++Face) {

for (std::size\_t Level = 0; Level < tex.levels(); ++Level) {

glTextureSubImage2D(textureName, static\_cast<GLint>(Level),

0, 0,

texExtent.x, texExtent.y,

texFormat.External, texFormat.Type, tex.data(Layer, Face, Level));

}

}

}

// This only works for 2D Textures...

// Set the texture wrapping parameters

glTextureParameteri(textureName, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);

glTextureParameteri(textureName, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);

// Set texture filtering parameters

glTextureParameteri(textureName, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTextureParameteri(textureName, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glGenerateMipmap(GL\_TEXTURE\_2D);

glBindTexture(GL\_TEXTURE\_2D, 0);// Unbind texture when done, so we won't accidentily mess up our texture.

}