

Shader Development- By Mateusz Jadasz

by Mateusz Jadasz

Submission date: 31-Mar-2023 06:07PM (UTC+0100)

Submission ID: 201223265

File name: 17845_Mateusz_Jadasz_Shader_Development-_By_Mateusz_Jadasz_495096_1504291892.docx
(398.96K)

Word count: 3933

Character count: 28875

SHADER DEVELOPMENT

By Mateusz Jadasz (1901291)

Video Demo Link: <https://youtu.be/WPNKpstBRXk>

TABLE OF CONTENTS

Introduction	3
Literature Review	3
Emerging Trends	3
Ray Tracing	3
Graphics API's	4
Disclaimer	4
Post Processing Effects	5
Thresholding	5
Inverted Colours	6
Image Loading	6
Testing	7
User Guide	8
Bibliography	9
Appendix	10
Source.Cpp	10
fragment_shader_InverseColours.glsl	13
fragment_shader_noPost.glsl	13
fragment_shader_Thresholding.glsl	14
vertex_shader.glsl	15
MaterialClass.h	15
MeshClass.h	16
ModelClass.h	19
ObjectClass.h	19
ShaderClass.h	21
TextureClass.h	24
VertexClass.h	25

INTRODUCTION

Image processing and graphics post-processing are both some of the core concepts that make up fields like software engineering and computer science. Often utilized in creating software such as image editors, game engines, as well as 3D modelling and animating software, such as Photoshop, Unity, and Blender. Both areas deal with the analysis of graphics and images, as well as enhancement and manipulation of said graphics, and images. The key difference between image processing and graphics post-processing, is that image processing primarily focuses on working with images themselves. This includes applying filters, restoring, enhancing as well as segmenting images. Although graphics post-processing includes working with certain effects like greyscale, blur, sepia, or thresholding, which by definition could be considered filtering (which comes under image processing), every one of those things are done on 2D and 3D objects when they are rendered, using shaders.

LITERATURE REVIEW

During my research of graphics post-processing and image processing, I have come across multiple sources that have proved themselves to be very informative and generally useful in my work with shaders. One such source was “Computer Graphics: Principles and Practice” [1]. This book, written James Foley, Andries van Dam, Steven Feiner, and John Hughes, covered a wide array of topics from the 2D and 3D graphics field. Some of the topics included rendering, shaders, geometric modelling, and lighting. Similarly, the “Fundamentals of Computer Graphics” [2] by Peter Shirley and Steve Marschner, also covered topics like lighting and shading. Furthermore, the authors also wrote about other concepts like ray tracing and texture mapping which, although not as useful to my application and me personally, they were still very interesting to read about. Yet another book that proved to be very useful in my research, was “Image Processing, Analysis, and Machine Vision” [3] written by Milan Sonka, Vaclav Hlavac, and Roger Boyle. This book proved to be an interesting read when I researched thresholding, as it covered computer vision concepts like edge detection and object recognition. Alongside Sonka’s, Hlavac’s, and Boyle’s book, Richard Szeliski’s “Computer Vision: Algorithms and Applications” [4], also provided some useful insight into object recognition, as well as general image processing and segmentation. Finally, while researching concepts like compression, restoration, and enhancement, two books have proved to be very informative and useful. Those books were “The Image Processing Handbook” [5] by John Russ and Brent Neal as well as, the “Digital Image Processing” [6], a book written by Rafael Gonzalez and Richard Woods. Both books have provided a thorough introduction to the image processing concepts of compression, restoration, and enhancement.

EMERGING TRENDS

There are multiple current, new, and emerging trends in GPU and shader programming. Many of them are slowly shaping the future of high-performance computing as well as graphics in general.

RAY TRACING

One such trend is Ray tracing. Ray tracing is a technique of rendering graphics, that simulates the way that light behaves in a 3D scene. The way it works is, it traces the path of light rays as they interact with the different objects in the scene. This allows the different refractions, shadows and reflections to appear more realistic. Normally, other techniques of rendering like rasterization, render objects on to a 2D screen, which can sometimes make objects look very flat and unrealistic. Ray tracing, traces the path of light as it passes through the 2D viewing plane and onto the 3D environment. It simulates the way that light behaves as it

bounces off all the different objects in the scene and back at the light sources, which allows said objects to look more realistic with the complex lighting effects at work. However, ray tracing is a very intensive process from a computational standpoint, which means it needs quite a large amount of processing power, especially when it is done in real time, for example in games when environments and objects in scenes are rendered and de-rendered all the time.

GRAPHICS API'S

Another set of trends in GPU and shader programming are all the different Graphics API's, such as OpenGL, Vulkan, and DirectX. Graphics API's are sets of tools and interfaces that allow developers to program applications that can interact with GPU's in an efficient and a standardized way. There are multiple graphics API's that are used by thousands of developers every day. In the game development scene, the most commonly used API is DirectX. DirectX is technically a collection of API's put into one, which were all originally developed by Microsoft, for running software on Microsoft operating systems. Another popular API is Vulkan, which just like DirectX, also happens to be most often used for creating high performance software like games for computers, consoles, and even virtual reality. The difference between DirectX and Vulkan is that DirectX is designed to develop applications only for Microsoft based platforms. OpenGL, just like Vulkan is a cross-platform API, however the overarching differences between the three is that DirectX, although exclusive to Microsoft platforms, is the highest level of the three API's. This means that it provides a much simpler interface which is also due to the fact that compared to Vulkan and OpenGL, DirectX has much more documentation and therefore it is much easier to use overall, however this all comes at a cost of flexibility and control over hardware. OpenGL and Vulkan, although much less documented, are also much lower level than DirectX which means that developers using these APIs can count on much more control and flexibility during the development process. Furthermore, Vulkan is generally considered to be much faster than OpenGL and DirectX, and both OpenGL and Vulkan can be used to develop in a wider array of programming languages such as C++, Java, and Python, whereas DirectX, is designed to only be worked with in Microsoft's programming languages like C# and C++. However overall, DirectX is much easier to use, which means that realistically none of the API's are better than the other, it just depends on the user and the goal that the user is trying to achieve.

DISCLAIMER

In order to tackle the given task efficiently, I have decided to recycle some code that I have previously written and submitted for another module. The assignment in question was the "3D OGL Games Programming" assignment written by me, and submitted on the 9th of January 2023, for the Graphics and Games Programming module. I have decided to use the code written for this assessment partially because recycling code is generally considered good software engineering practice, and also because it would mean that during this assignment I could focus entirely on the research and implementation of the different post-processing effects as well as creating and working with different shaders. This is because using the recycled code, I could skip the process of writing and debugging systems that allowed for the basic functions of the application like the rendering and texturing of 3D objects, as well as things like camera movement and rotation. With the basic functions of the software in place I was able to fully devote myself to building upon and expanding the functionality of the existing application, rather than rewriting an already existing and well structured piece of code.

POST PROCESSING EFFECTS

THRESHOLDING

One of the post-processing effects I have decided to implement in my application was thresholding. The fragment shader responsible for this post-processing effect would essentially look at an image, convert it into grayscale by converting every fragment into an average of all of its RGB values, and then finally it would run through every fragment one more time, however this time, if the fragment value was greater than an average value of 0.5, the fragment would be coloured in as white. Otherwise, if the fragment average value was less than or equal to 0.5, the fragment would be coloured in with black. This effect has created this image:



As opposed to what the image looks like without any post processing effects:



[7]

Photograph used in this report and the application was taken by Westend61 and can be found at this.
URL: <https://www.telegraph.co.uk/news/0/do-trees-have-feelings-too-one-expert-says-they-do/>

Alternatively, the shader can be made to work with a specific range. For example, if the fragment average value was greater than 0.65 or lower than 0.35 then the pixel will be coloured in white, otherwise, if the fragment average is within the range of 0.65 and 0.35, then it will be coloured in black. This will create a very interesting looking effect on the image:



INVERTED COLOURS

Creating the shader responsible for causing the negative colour effect was a much simpler process, as all it required was to take the colour values at each texture coordinate, then overwrite it with a negative version of itself. This in turn would create an effect that made the original image look like this:

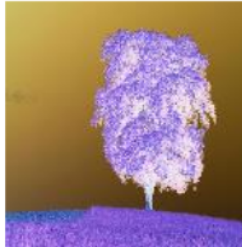
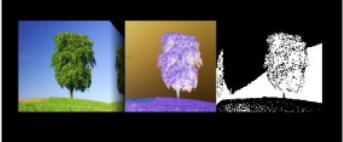
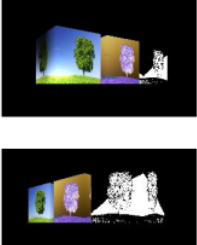


IMAGE LOADING

Image loading was a process that was already present in the original version of the software, before I began to work on expanding it. This means that although image loading isn't a native feature of OpenGL, it was handled by an external image loader, SOIL2, created by and posted on GitHub by a user known as SpartanJ. The source code and the image loader is available at this address: <https://github.com/SpartanJ/SOIL2>

Test Title	Test Description	Test Result Screenshot
Rendering Without Effects	Testing the program's capability to render an object with no post-processing effects applied to it	
Rendering With Thresholding Applied	Testing the program's capability to render an object with the appropriate thresholding fragment shader applied	
Rendering With Inverted Colours Applied	Testing the program's capability to render an object with the appropriate inverse colours fragment shader applied	
Rendering All Objects at the Same Time	Testing the program's capability to render three different cubes at the same time, all textured, and each of them being rendered using a different fragment shader	
Moving the View	Testing the ability to move the view further, closer and around the objects	
Rotating the View	Testing the ability to rotate the view and view the object from an angle, no matter where the view is positioned in relation to the objects	

--	--	--

USER GUIDE

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

BIBLIOGRAPHY

- [1] J. D. Foley, A. van Dam, S. K. Feiner and J. F. Hughes, *Computer Graphics: Principles and Practice*, Addison-Wesley Professional, 2013.
- [2] P. Shirley and S. Marschner, *Fundamentals of Computer Graphics*, A K Peters, 2015.
- [3] M. Sonka, V. Hlavac and R. Boyle, *Image Processing, Analysis, and Machine Vision*, Brooks, 2007.
- [4] R. Szeliski, *Computer Vision: Algorithms and Applications*, Springer, 2010.
- [5] J. C. Russ and F. B. Neal, *The Image Processing Handbook*, CRC Press, 2015.
- [6] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, Pearson, 2007.
- [7] Westend61 and P. Wohlleben, "Do trees have feelings too? One expert says they do," *The Telegraph*, 11 September 2016. [Online]. Available: <https://www.telegraph.co.uk/news/0/do-trees-have-feelings-too-one-expert-says-they-do/>. [Accessed 4 February 2023].

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);
    }

    //camera rotaion
    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 (glewInit() != 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_NoPost((char*)"vertex_shader.glsl",
(char*)"fragment_shader_noPost.glsl");
Shader shader_program_InverseColours((char*)"vertex_shader.glsl",
(char*)"fragment_shader_InverseColours.glsl");
Shader shader_program_Thresholding((char*)"vertex_shader.glsl",
(char*)"fragment_shader_Thresholding.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_NoPost.setMat4fv(modelMatrix, "modelMatrix");
Texture tree("Images/Tree.jpg", GL_TEXTURE_2D, 0); //image taken by Westend61.
Available here https://www.telegraph.co.uk/news/0/do-trees-have-feelings-too-one-expert-says-they-do/

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

    updateInput(window, &shader_program_NoPost);
    updateInput(window, &shader_program_InverseColours);
    updateInput(window, &shader_program_Thresholding);
}

```

```

        //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 in 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_NoPost.setMat4fv(projectionMatrix, "projectionMatrix");

        Model model1(glm::vec3(-0.5, 0.3, 0.0), shader_program_NoPost, &tree,
&tree, &mesh);
        Model model2(glm::vec3(0.2, 0.3, 0.0), shader_program_InverseColours,
&tree, &tree, &mesh);
        Model model3(glm::vec3(0.9, 0.3, 0.0), shader_program_Thresholding,
&tree, &tree, &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_INVERSECOLOURS.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;

void main()
{
    vec4 negativeCol = vec4(vec3(1.0 - texture(texture0,vs_texcoord)),1.0);
    fs_colour = negativeCol;
}

```

FRAGMENT_SHADER_NOPOST.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 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) * vec4(vs_colour,
1.0) * (vec4(ambientLight, 1.0) + vec4(diffuseFinal, 1.0) + vec4(specularFinal,
1.0f));
}

```

FRAGMENT_SHADER_THRESHOLDING.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;

void main()
{
    fs_colour = texture(texture0,vs_texcoord);
    float average = (fs_colour.r + fs_colour.g + fs_colour.b) / 3.0;

    if(average > 0.65 || average < 0.35)
    {
        fs_colour = vec4(1.0, 1.0, 1.0, 1.0);
    }
    else
    {
        fs_colour = vec4(0.0, 0.0, 0.0, 1.0);
    }
}

```

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:

    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& numOfVertices, GLuint*
indexArray, const unsigned& numOfIndices) {
        //set variables
        this->numOfVertices = numOfVertices;
        this->numOfIndices = numOfIndices;

        //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->numOfVertices * 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->numOfIndices *
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& numOfVertices, GLuint* indexArray,
const unsigned& numOfIndices, 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, numOfVertices, indexArray, numOfIndices);
        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(-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(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(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),

        //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(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(-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(-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)

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

    for (size_t i = 0; i < numIndices; 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;
};

```

Shader Development- By Mateusz Jadasz

GRADEMARK REPORT

FINAL GRADE

61 / 100

GENERAL COMMENTS

Instructor

The similarity score is due to reuse of boiler plate framework code and generic library function calls. No issues here.

Code compiles and executes.

Your code shows textured objects on separate cubes - ok. 2 separate effects used, thresholding and a colour transform. Good.

Code framework reuse in evidence and discussed in report. ok.

The colour inversion is straightforward - did you consider a more complex mapping perhaps, or instead, the use of more elaborate kernels and convolution for example?

A part of your review section focuses on a review of individual texts, etc. Instead you should review related work that can inform and guide your research and allow you to present to the reader an informed review of the state of the art of graphics using GPUs.

Essential ideas and concepts, nevertheless identified. Some could be more detailed. Any omissions?

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

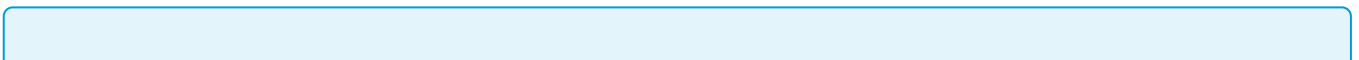
PAGE 22

PAGE 23

PAGE 24

PAGE 25

GRADING FORM: SMU GENERIC_GJ



MATEUSZ JADASZ

0

RESEARCH-INFORMED LITERATURE

KNOWLEDGE AND UNDERSTANDING

ANALYSIS

PRACTICAL APPLICATION AND DEP.

SKILLS FOR PROFESSIONAL PRAC.