

GPU assignment

by Tynan Kahar

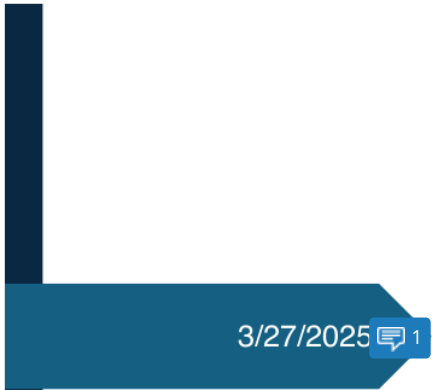
Submission date: 28-Mar-2025 08:31AM (UTC+0000)

Submission ID: 254034965

File name: 41519_Tynan_Kahar_GPU_assignment_650609_1126087450.docx (66.87K)

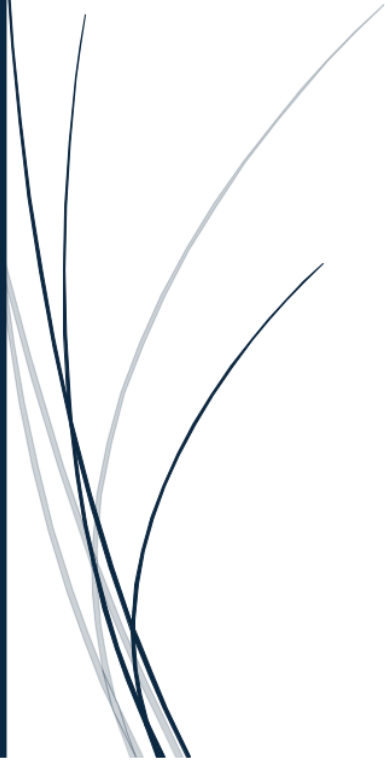
Word count: 1899

Character count: 11991

A thick dark blue vertical bar runs down the left side of the page. A horizontal blue arrow points to the right from the bar, containing the date and a comment icon.

3/27/2025  1

Shader Development

Several thin, curved lines in dark blue and light grey originate from the vertical bar and sweep upwards and to the right.

Tynan Kahar
UWTSD

Contents

Video of code	2
Introduction	2
Edge Detection	2
Colour Inversion.....	3
Practical Applications.....	3
Edge Detection.....	3
Colour Inversion	3
Review of the code	4
Renderer.cpp	4
Renderer.h	9
Vertex.shader	10
Edge_colour.shader	10
Edge detection.....	10
Colour inversion.....	11
Main.cpp.....	12
References.....	14

Video of code

<https://youtu.be/lcLP8Kr4TWY>

Introduction

Through this review, I will explore the applications of edge detection and colour inversion which are both fundamental image processing techniques, both of these are very effective in a range of different ways, they have uses in security medicine art as well as a myriad of other areas where that can be instrumental in specific processes. For this assignment, I have created some OpenGL code that loads an image with edge detection on the left and colour inversion on the right, showing how to manipulate an image dynamically using shaders. as well as looking at real-world implementations of edge detection and colour inversion in terms of image processing.

Edge Detection

Edge detection is one of the most basic uses of image processing; it identifies significant changes in pixels, which usually indicate boundaries or texture changes in an image. By detecting these edges, we can gain useful information from images while making them require less data.

The way edge detection works is by analysing the change in the intensity of localised pixels, mainly looking for where there are drastic changes which indicate an edge. It starts by making the image greyscale. The formula for greyscale intensity is $\text{intensity} = 0.299r + 0.587g + 0.114b$.

Then you need to work the gradient which controls the strength of the edges. This works using this code:

```
vec3 kernelX[3] = vec3[(vec3(-1, 0, 1),  
                        vec3(-2, 0, 2),  
                        vec3(-1, 0, 1))];
```

```
vec3 kernelY[3] = vec3[(vec3(-1, -2, -1),  
                        vec3( 0, 0, 0),  
                        vec3( 1, 2, 1))];
```

Finally, there is setting the upper and lower thresholds for what determines a strong edge over a weak one.

Colour Inversion

Colour inversion is where, using image processing, an image has the colour of its pixels reversed. The way this works is by subtracting each pixel colour value from the maximum intensity. There are a few different formulas for colour inversions; this code uses RGB colour inversion, which is done using this code:

```
vec4 color = texture(u_Texture, v_TexCoords);  
  
return vec4(1.0 - color.rgb, 1.0);
```

this code works with normalized values (0 to 1)

Practical Applications

Edge Detection

The practical applications of edge detection are in robotics, where it can work in a myriad of different ways, working on object recognition where robots can use it to identify the differences between things in their field of view; obstacle detection, where things like self-driving cars can use it in order to recognise that there are obstacles on their route; mapping, where things like drones can use edge detection to map rooms to avoid obstacles; and quality control, where it can be used to detect defects.

It can also be used in medical imaging in order to enhance the quality of images to allow doctors to more easily identify an abnormality in the body, which can include things like tumours and lesions; it can also be used to look over blood vessels and organs in order to detect if someone has specific conditions. It is also used to enhance X-rays, making them easier to analyse.

Colour Inversion

The practical applications of colour inversion are that it can be used for X-rays and scans to make the visibility of parts of the body more distinct, and it can also be used to help those with visual impairments by improving the contrast, making it easier for them to see it. There is also the artistic side, where it can be used for negative image effects. Moreover, it plays a crucial role in security and forensics, as it can be used to bring to light unseen details in a photo; for example, it can enhance low-visibility videos for surveillance footage. This is because it makes the contrast more distinguished, allowing for easier analysis of things like facial features and clothing as well as licences; it is especially effective when it comes to night vision situations, making it far easier to detect movement. It also can be used to reveal if a document is tampered with; for example, if a document has been changed digitally, inverting the colours will result in inconsistencies in brightness and textures being completely visible. It can also be used in handwriting to find if there was a forgery. It also can work for

crime scene analysis, as some evidence may be too faint to notice, so by inverting the colour, it would become more visible. This is especially prevalent when it comes to bloodstains, as it's used not just to find faint stains but also to help experts analyse them, as they're clearer.

Review of the code

Renderer.cpp

This is the compile shader function, it is used to compile a shader from OpenGL source code, subsequently creating a new shader, and using the shader which is returned you can link it to the rest of the program, it has an error check within it in order to ensure that it compiles correctly and it will alert me if not making it easier to go back and find the bug.

```
unsigned int Renderer::compileShader(const char* shaderSource, GLenum shaderType) {
    unsigned int shader = glCreateShader(shaderType);
    glShaderSource(shader, 1, &shaderSource, nullptr);
    glCompileShader(shader);

    int success;
    glGetShaderiv(shader, GL_COMPILE_STATUS, &success);
    if (!success) {
        char infoLog[512];
        glGetShaderInfoLog(shader, 512, nullptr, infoLog);
        std::cerr << "ERROR::SHADER::COMPILATION_FAILED\n" << infoLog << std::endl;
        return 0;
    }
    return shader;
}
```

This is a function that loads compiles and links the OpenGL shader program from the vertex and fragment shader source files. After creating and linking the shaders it has an error check to ensure that they function as intended.

```
unsigned int Renderer::loadShaderProgram(const std::string& vertexPath, const std::string& fragmentPath) {
```

```
    std::ifstream vertexFile(vertexPath);
```

```
    std::stringstream vertexStream;
```

```
    vertexStream << vertexFile.rdbuf();
```

```
    std::string vertexCode = vertexStream.str();
```

```
    std::ifstream fragmentFile(fragmentPath);
```

```
    std::stringstream fragmentStream;
```

```
    fragmentStream << fragmentFile.rdbuf();
```

```
    std::string fragmentCode = fragmentStream.str();
```

```
    unsigned int vertexShader = compileShader(vertexCode.c_str(), GL_VERTEX_SHADER);
```

```
    unsigned int fragmentShader = compileShader(fragmentCode.c_str(),  
GL_FRAGMENT_SHADER);
```

```
    unsigned int shaderProgram = glCreateProgram();
```

```
    glAttachShader(shaderProgram, vertexShader);
```

```
    glAttachShader(shaderProgram, fragmentShader);
```

```
    glLinkProgram(shaderProgram);
```

```
    int success;
```

```
    glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
```

```
    if (!success) {
```

```
        char infoLog[512];
```

```
        glGetProgramInfoLog(shaderProgram, 512, nullptr, infoLog);
```

```

        std::cerr << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" << infoLog <<
std::endl;

        return 0;

    }

```

```

        glDeleteShader(vertexShader);
        glDeleteShader(fragmentShader);

        return shaderProgram;
    }

```

This function loads the image as a texture and creates an object so that it can be rendered using the STB_library. Within this function it sets texture parameters, it then obviously loads the image, it then create the open gl texture, before that using a final set of error handling messages.

```

unsigned int Renderer::loadTexture(const char* texturePath) {
    unsigned int textureID;
    glGenTextures(1, &textureID);

    glBindTexture(GL_TEXTURE_2D, textureID);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

    int width, height, nrChannels;
    unsigned char* data = stbi_load(texturePath, &width, &height, &nrChannels, 0);
    if (data) {

```

```

        std::cout << "Texture loaded successfully: " << texturePath << std::endl;
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
GL_UNSIGNED_BYTE, data);
        glGenerateMipmap(GL_TEXTURE_2D);
    }
    else {
        std::cerr << "ERROR::TEXTURE::FAILED_TO_LOAD: " << texturePath << std::endl;
    }
    stbi_image_free(data);

    return textureID;
}

```

This is the render image function, which renders the image using open gl, it binds the textures and sets the shader program as well as drawing a rectangle using indexed rendering.

```

void Renderer::renderImage(unsigned int texture, unsigned int shaderProgram) {
    glUseProgram(shaderProgram);

    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, texture);

    glUniform2f(glGetUniformLocation(shaderProgram, "u_Texel"), 1.0f / 800.0f, 1.0f / 600.0f);
    // Adjust as per your window size

    glBindVertexArray(VAO);
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
}

```

This function sets up the rectangle for rendering in open gl, it does this by defining the vertices as well as the indices, it also configures the vertex and element buffer objects, as well as the vertex array object.

```
void Renderer::setupQuad() {
```

```
    float vertices[] = {
```

```
        -1.0f, -1.0f, 0.0f, 0.0f, 1.0f,
```

```
        1.0f, -1.0f, 0.0f, 1.0f, 1.0f,
```

```
        1.0f, 1.0f, 0.0f, 1.0f, 0.0f,
```

```
        -1.0f, 1.0f, 0.0f, 0.0f, 0.0f
```

```
    };
```

```
    unsigned int indices[] = {
```

```
        0, 1, 2,
```

```
        0, 2, 3
```

```
    };
```

```
    glGenVertexArrays(1, &VAO);
```

```
    glGenBuffers(1, &VBO);
```

```
    glGenBuffers(1, &EBO);
```

```
    glBindVertexArray(VAO);
```

```
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
```

```
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

```
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
```

```
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices,  
GL_STATIC_DRAW);
```

```
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5 * sizeof(float), (void*)0);
```

```

    glEnableVertexAttribArray(0);

    glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 5 * sizeof(float), (void*)(3 *
sizeof(float)));

    glEnableVertexAttribArray(1);


    glBindVertexArray(0);
}

```

Renderer.h

The renderer.h sets up the renderer class, to be used in the renderer.cpp file this means setting all of the functions that will be used in the cpp file, like loadshaderprogram and loadtexture as well as some others. It also creates the variables that are used.

```

class Renderer {
public:
    Renderer(); // Constructor declaration
    ~Renderer(); // Destructor declaration


    unsigned int loadShaderProgram(const std::string& vertexPath, const std::string&
fragmentPath);

    unsigned int loadTexture(const char* texturePath);

    void renderImage(unsigned int texture, unsigned int shaderProgram);


private:
    void setupQuad();

    unsigned int compileShader(const char* shaderSource, GLenum shaderType);


    unsigned int VAO, VBO, EBO;
};

```

Vertex.shader

This code is a very basic vertex shader for open gl.

```
#version 330 core
```

```
layout (location = 0) in vec3 aPos; // Position attribute
```

```
layout (location = 1) in vec2 aTexCoord; // Texture coordinate attribute
```

```
out vec2 v_TexCoords;
```

```
void main()
```

```
{
```

```
    gl_Position = vec4(aPos, 1.0);
```

```
    v_TexCoords = aTexCoord;
```

```
}
```

Edge_colour.shader

Edge detection

This is a fragment shader that has an edge detection function that works on the image in order to highlight the edges, it also makes it grey so the edges are more visible.

```
#version 330 core
```

```
out vec4 FragColor;
```

```
in vec2 v_TexCoords;
```

```
uniform sampler2D u_Texture;
```

```
uniform vec2 u_Texel; // 1 / texture size
```

```
vec4 applyEdgeDetection()
```

```
{
```

```
    vec3 kernelX[3] = vec3[(vec3(-1, 0, 1),
```

```
        vec3(-2, 0, 2),
```

```
        vec3(-1, 0, 1))];
```

```

vec3 kernelY[3] = vec3[(vec3(-1, -2, -1),
                        vec3( 0,  0,  0),
                        vec3( 1,  2,  1))];

vec3 colourX = vec3(0.0);
vec3 colourY = vec3(0.0);

for(int y = -1; y <= 1; y++)
{
    for(int x = -1; x <= 1; x++)
    {
        vec3 sample = texture(u_Texture, v_TexCoords + u_Texel * vec2(x, y)).rgb;
        colourX += kernelX[y + 1][x + 1] * sample;
        colourY += kernelY[y + 1][x + 1] * sample;
    }
}

float edgeStrength = length(colourX) + length(colourY);
return vec4(vec3(edgeStrength), 1.0);
}

```

Colour inversion

This is a colour inversion function that changes and reverses the colours on the image.

```

vec4 applyColourInversion()
{
    vec4 color = texture(u_Texture, v_TexCoords);
    return vec4(1.0 - color.rgb, 1.0);
}

```

This is used to apply the edge detection on one side and the colour inversion on the other.

```

void main()
{
    if (v_TexCoords.x < 0.5) {
        FragColor = applyEdgeDetection(); // Apply Sobel edge detection on the left half
    } else {
        FragColor = applyColourInversion(); // Apply color inversion on the right half
    }
}

```

Main.cpp

This is the main, it starts by Initialize GLFW and then running a few error checks to ensure all the things needed for the code are running correctly, it then creates the window, obviously followed by another error check to ensure the window loads. It then activates GLEW before then loading the shaders and then the image, then is the main while loop, where it renders the image.

```

int main() {

    if (!glfwInit()) {
        std::cerr << "ERROR::GLFW::INIT_FAILED" << std::endl;
        return -1;
    }

    GLFWwindow* window = glfwCreateWindow(800, 600, "Image Processing", nullptr,
    nullptr);
    if (!window) {
        std::cerr << "ERROR::GLFW::WINDOW_CREATION_FAILED" << std::endl;
        glfwTerminate();
        return -1;
    }

    glfwMakeContextCurrent(window);
    glfwSwapInterval(1);

```

```
if (glewInit() != GLEW_OK) {
    std::cerr << "ERROR::GLEW::INIT_FAILED" << std::endl;
    return -1;
}

Renderer renderer;

unsigned int shaderProgram =
renderer.loadShaderProgram("C:/Users/tynan/OneDrive/Desktop/Shader
Development/Shader Development/shader/vertex.shader",
"C:/Users/tynan/OneDrive/Desktop/Shader Development/Shader
Development/shader/Edge_colour.shader");

unsigned int texture = renderer.loadTexture("C:/Users/tynan/OneDrive/Desktop/Shader
Development/Shader Development/image/image.png");

// Main rendering loop
while (!glfwWindowShouldClose(window)) {

    // Render image with shader
    renderer.renderImage(texture, shaderProgram);

    // Swap buffers
    glfwSwapBuffers(window);
    glfwPollEvents();
}

glfwTerminate();
return 0;
}
```

References

<https://blog.roboflow.com/edge-detection/>

<https://bbau.ac.in/dept/CS/TM/Edge%20detection%20.pdf>

<https://borisfx.com/blog/how-to-invert-colors-on-a-picture/#:~:text=An%20inverted%20image%20can%20add,to%20the%20strong%20color%20contrasts.>

GPU assignment

GRADEMARK REPORT

FINAL GRADE

73/100

GENERAL COMMENTS

Nice, I like your implementation especially!
Your engagement with literature and lack of use of IEEE referencing let you down big time, you could have scored a decent chunk for that!

Your other fatal flaw was the use of subjective writing.
In academic report writing, the use of personal pronouns such as "I" or "we" is generally avoided to maintain an objective and formal tone. Academic writing aims to present information, arguments, and findings in a way that is neutral and evidence-based, rather than reflective of personal opinion or involvement. By avoiding "we", the focus remains on the subject matter rather than the author, which enhances the credibility and impartiality of the report. This convention also aligns with the broader academic goal of producing work that can be critically evaluated on the basis of logic and evidence, rather than perceived personal bias or collective authorship.

===== MARKING GRID ===== Engagement with Literature Skills [20] - Review of GPUs/shaders

5

Knowledge and Understanding Skills [10] - Understanding shader/GPU architectures - Evident in review + code **9**

Cognitive and Intellectual Skills [25] - Critical review of literature - Problem analysis & solution design **22**

Practical Application Skills [40] - Solution/testing in C++ OGL + shaders **35**

PAGE 1



Comment 1

no comment

PAGE 2

PAGE 3



Comment 2

In academic report writing, the use of personal pronouns such as "we" is generally avoided to maintain an objective and formal tone. Academic writing aims to present information, arguments, and findings in a way that is neutral and evidence-based, rather than reflective of personal opinion or involvement. By avoiding "we", the focus remains on the subject matter rather than the author, which enhances the credibility and impartiality of the report. This convention also aligns with the broader academic goal of producing work that can be critically evaluated on the basis of logic and evidence, rather than perceived personal bias or collective authorship.

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

