University for the Common Good

Module Name: Graphics Programming

Module ID: M31625657

Module Assignment: Shader Coursework

Module Leader: Bryan Young

Academic Session: 2022-2023

Student Name: Aiden Donkin

Student ID: S2041059

I confirm that the code contained in this file (other than that provided or authorised) is all my own work and has not been submitted elsewhere in fulfilment of this or any other award.

Signature: Aiden Donkin

Word Count (Excluding Code and Formatting): 1,973.

Word Count (Including Code and Formatting): 2,292.

Table of Contents

1. Introduction 3

2. Vertex Shader 3

2.1 Input Variables 3

2.2 Output Variables 4

2.3 Functionality 4

3. Fragment Shader 6

3.1 Input Variables 6

3.2 Output Variables 7

3.3 Functionality 7

4. Project Files 8

1. Introduction

The primary goal of this project was to create three unique shader programs, each specialising in a specific graphical technique. The first of these three shaders consisted of a geometry shader which creates an effect which explodes an object into many identical pieces before reversing the process and returning the original shape. The function of the second shader was to achieve an environmental mapping effect in conjunction with a skybox created from a cube map. When applied to an object, this shader would reflect the features displayed by the current skybox (in this case, the mountains appearing on the implemented skybox) to create a semi-mirroring visual effect.

Alongside these two shaders, a third shader was created and it is this shader that is the primary focus of this report. This primary aim of the third shader was to create a visual effect that emulates the surface of the Sun when applied to an object. In the early stages of development, two effects were identified that would aid the achievement of this goal: a swirling effect on the surface of the object and a glow effect. The purpose of the swirling effect was to create a dynamic object to recreate the perceived movement of gas on the photosphere of the Sun. In an effort to imitate the light produced by the sun, a glow effect was also developed which utilised the lighting technique commonly referred to as bloom.

Bloom lighting is a shader effect which is regularly used in the gaming industry. Two examples of its popular uses include creating atmospheric environments and guiding player progression towards objects or areas of interest. The technique simulates the visual effect of light spilling out from the object it is applied to and into the surrounding area. In this project, the bloom effect is achieved by utilising a blur effect and manipulating the brightness of the texture it is applied to.

2. Vertex Shader:

```
#version 420 core

out vert_vars //Declaration of output variables that will be passed to fragment shader
{
    vec2 outputTexCoords;
    vec3 swirlPosVector; // Modified vertex position
}vert_out;

layout(location = 0) in vec3 vertPosition; //Declaration of layout variables
layout(location = 1) in vec2 vertTexCoords;

uniform mat4 model; //Declaration of uniform variables, defined in main class
uniform mat4 view;
uniform mat4 projection;
uniform float swirlSeed;
uniform float timeVar;
uniform float randomConstant;

float random(float seed) //Used to create a random element in the swirl effect
{
    return fract(sin(seed) * randomConstant ); //Used the defined randomConstant to return a semi-random number
}

void main()
{
    vec4 transformPosition = model * vec4(vertPosition, 1.0); //Determines position of shader

    vec2 swirledTexCoord = vertTexCoords; //Assigns texture coordinates

    // Creation of swirling effect
    float swirlIntensity = 2.0; // Determines the intensity of the swirl displayed
    float angle = length(transformPosition.xy) * swirlIntensity +  timeVar * random(swirlSeed); //Adds a random element to the swirl
    mat2 swirlAngleMatrix = mat2(cos(angle), sin(angle), -sin(angle), cos(angle)); //Determines angle of swirl
    vert_out.swirlPosVector = vec3(swirlAngleMatrix * transformPosition.xy, transformPosition.z); // Passes swirl data to output variable

    gl_Position = projection * view * vec4(vert_out.swirlPosVector, 1.0); //Sets current vertext position
    vert_out.outputTexCoords = swirledTexCoord; //Passes the texture coordinate data to the output variable
}
```

Figure 1: The Vertex Shader (stored as sunShader.vert in project files).

2.1 Input Variables

The vertex shader displayed in Figure 1 has several input variables and they are divided into two distinct categories: layout qualifiers and uniform variables. The layout qualifiers are defined in the

shader class (stored as shader.cpp in project files). In this case, there are two layout qualifiers: '**vertPosition**' and '**vertTexCoords**'. '**vertPosition**' stores the position vector of the shader and '**vertTextCoords**' is responsible for storing the texture coordinates of the shader. The purpose of these values is to undergo potential manipulation and then be passed onto the output variables for use in the fragment shader.

The uniform variables have a pre-defined value that cannot be altered and serve the purpose of manipulating the values that will be passed further down the line onto the fragment shader. The uniform variables are assigned a value in the main class of the application (stored as main.cpp in project files), specifically within the corresponding linking function for this shader. In the case of the Sun shader there are six distinct uniform variables, each with their own purpose. The first three of these variables consist of the '**model'**, '**view**', and '**projection**' matrices. These variables are primarily responsible for mapping the chosen object from object space to screen space. Specifically, the function of the model matrix is to map the object's coordinates from local space to world space. The view matrix carries on this process from world space and transfers the data to camera space. Finally, the projection matrix moves the data from camera space to screen space. If this pipeline is correctly set up, the user can render objects on the application screen with ease.

The fourth variable, '**swirlSeed**', serves the purpose of bringing a random element into the angle of the swirl effect created by the shader. Without this variable, the swirling effect of the shader will follow a set course throughout the course of the application as the angle of the swirl has been defined at the start of the application and is never again altered. The random element allows for some dynamic swirling to occur, thus more effectively replicating the surface of the sun.

The fifth variable, '**timeVar**', is key in creating the perceived movement of the shader. This value is determined by the '**counter**' variable found in the main class. It is altered each time the '**renderGame()**' function is called and establishes a form of chronological progression within the application. In the vertex shader its primary function is found in the calculation of the swirl angle.

The final variable of this vertex shader, '**randomConstant**', is utilised in the '**random()**' function to create semi-random floats. These floats are used further down the line to help determine the angle of the swirl.

2.2 Output Variables

In the case of this vertex shader, there are only two output variables which are passed down the line to the fragment shader. These are '**outputTexCoords**' and '**swirlPosVector**'. The former is the more simplistic of the two and is determined by the texture coordinates of the object that the shader is applied to. The latter of the variables is more in depth and is directly responsible for the creation of the swirl effect. Both of these variables are output to the fragment shader through the use of the '**vert_vars**´ group.

2.3 Functionality

The primary function of the vertex shader in this instance is to create a swirl effect on the surface of the object that is influenced by a random element. This section will describe the processes undertaken to achieve this effect.

The first calculation carried out in this shader is the determination of the object's position. This is achieved through the implementation of the following code:

vec4 transformPosition = model *vec4(vertPosition, 1.0);

This utilised the layout qualifier '**vertPosition**' and the model matrix in order to achieve the storing of the transform position.

The next key step in this shader involves the determination of the variable '**swirlPosVector**'. The first stage in this process is to establish the intensity of the swirl effect that is created. In the example shown in Figure 1, the '**swirlIntensity**' variable is set to a value of 2.0. This can be easily altered to increase or decrease the magnitude of the effect. Following this, calculation of the swirl angle is required. This is determined through of the separate multiplication of the length of the transform x and y variables with the '**swirlIntensity**, as well as the '**timeVar**' variable with the value produced by the '**random()**' function where the '**swirlSeed**' variable is the only argument. This can be viewed in the following line:

float angle = length(transformPosition.xy) * swirlIntensity + timeVar * random(swirlSeed);

This step is followed by the determination of the '**swirlAngleMatrix**' variable through the application of since and cosine functions to the pre-determined '**angle**' variable. This can be viewed in the following line:

mat2 swirlAngleMatrix = mat2(cos(angle), sin(angle, -sin(angle), cos(angle));

The final step in this process is to calculate the value of '**swirlPosVector**' itself. This is achieved by creating a three-element vector variable and calculating the values from the '**swirlAngleMatrix**' and '**transformPosition**' variables. The matrix is multiplied by the x and y values of the transform to determine the first two elements of the vector, while the third element is simply the z value of the transform. The calculation is displayed below:

vert_out.swirlPosVector = vec3(swirlAngleMatrix * transformPosition.xy, transformPosition.z);

With the swirl values determined, the final processes are undertaken. Firstly, the gl_position of the vertex is determined through the multiplication of the '**projection**', '**view**', and '**swirlPosvector**' variables. This is then followed by assignment of the swirl texture coordinates to the relevant output variables:

gl_position = projection * view * vec4(vert_out.swirlPosVector, 1.0);

vert_out.outputTexCoords = swirledTextCoords;

With all of this system now complete, the application moves onto the fragment shader.

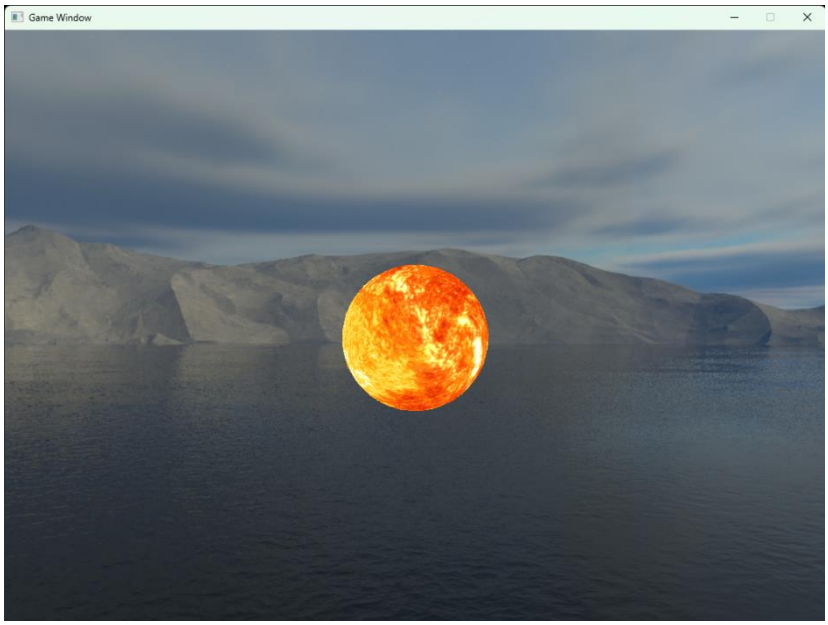

Figure 2: A screenshot of the sun shader applied to an object.

3. Fragment Shader:

```
#version 420 core

in vert_vars //Receives variables from vertex shader
{
    vec2 outputTexCoords;
    vec3 swirlPosVector;
}frag_in;

out vec4 fragColor;

uniform sampler2D textureSamplerVar; //Declaration of uniform variables, defined in main class
uniform float timeVarFrag; // Time value in seconds
uniform float minBI; // Minimum bloom intensity
uniform float maxBI; // Maximum bloom intensity
uniform float bFrequency; // Frequency of bloom intensity modulation

const int blurEffectRadius = 6; // Adjust the blur radius to control the size of the blur

void main()
{
    vec4 textureColour = texture(textureSamplerVar, frag_in.outputTexCoords); //Assigns data from texture sample

    float currentBI = mix(minBI, maxBI, 0.5 + 0.5 * sin(timeVarFrag * bFrequency)); //Determines the intensity of the bloom effect

    vec3 bloomEffectColour = vec3(0.0); //Creates a colour for the bloom effect

    for (int x = -blurEffectRadius; x <= blurEffectRadius; x++) //For loop used to create bloom colour from offset vector, texture coordinates, and texture sample
    {
        for (int y = -blurEffectRadius; y <= blurEffectRadius; y++)
        {
            vec2 offsetVector = vec2(x, y);
            vec4 bloomEffectSample = texture(textureSamplerVar, frag_in.outputTexCoords + offsetVector / textureSize(textureSamplerVar, 0));
            bloomEffectColour += bloomEffectSample.rgb;
        }
     }

    bloomEffectColour /= float((blurEffectRadius * 2 + 1) * (blurEffectRadius * 2 + 1)); //Refines bloom colour based on blur effect radius
    bloomEffectColour *= currentBI; //Applies bloom intensity variable to created colour


    vec3 outputColour = textureColour.rgb + bloomEffectColour; //Combines the texture colour with the created bloom colour
    fragColor = vec4(outputColour, 1.0); //Outputs the frag colour
}
```

Figure 3: The Fragment Shader (stored as sunShader.frag in project files).

3.1 Input Variables

The input variables of this fragment shader can be divided into three distinct categories: uniform variables, output variables of the vertex shader (prefaced by '**frag_in'**), and constants. Unlike the vertex shader, there are no layout qualifiers present in this shader.

The uniform variables are by far the most numerous in this shader; there are five in total and they are responsible for defining values that are used the creation of the bloom lighting effect. The first of these variables is the 2D sampler value '**textureSamplerVar**'. This variable provides a sample of the texture applied to the object which is directly involved in altering the colour of the texture. The second variable is similar to one seen in the vertex shader, '**timeVarFrag**'. This is, once again, involved in establishing a sense of time for the shader.

The remaining three uniform variables are all responsible for determining different aspects of the bloom effect and are adjustable in the main class to tailor the effect to the desires of the user. The first of these three variables is '**minBI**'. The purpose of this float is to serve as the minimum boundary for the intensity of the bloom effect. The bloom effect provided by this shader changes its intensity dynamically as the application runs, increasing and decreasing in its brightness as time passes. In turn the float variable '**maxBI**' serves as the upper boundary for this effect. The '**bFrequency**' variable determines the frequency at which the bloom effect modulates. It works in conjunction with the time variable to achieve this.

The output variables received from the vertex shader include '**outputTexCoords**' and '**swirlPosVector**'. As this shader is primarily concerned with the creation of bloom lighting, the latter of these variables is not modified further. However the former variable, '**outputTexCoords**', is key in producing the desired glow effect.

The final input variable of the fragment shader is a float constant labelled '**blurEffectRadius**'. This constant determines the radius of the blur effect which is added to the shader in order to simulate a

glow effect. It is adjustable within the shader and increasing the value will have a noticeable effect on the magnitude of the object's glow.

3.2 Output Variables

There is only a singular output variable present in this shader, '**fragColor**'. This variable is responsible for sending the final value calculated in this shader further down the line in order for the user to visualise the shader effect.

3.3 Functionality

The primary function of this shader is to establish bloom lighting to effectively imitate the glow produce by the Sun. This section will describe the system created to produce this effect.

The first function carried out by this shader is the calculation of the vector variable '**textureColour**', which is used later in the shader to determine the final colour output by the shader. It is calculated through use of the **'textureFunction()**' with '**textureSamplerVar**' and '**outputTexCoords**' being its two arguments. This is displayed in the following code:

```
vec4 textureColour = texture(textureSamplerVar, frag_in.outputTexCoords);
```

The second stage of this shader is to determine the intensity of the bloom effect and to create a three-element vector variable to store the colour of the bloom effect ('**bloomEffectColour**'). The process of calculating the '**currentBI**' variable involves utilising the '**mix()**' function to produce a value between boundary variables ('**minBI**' and '**maxBI**') that is also influenced by the sine of the multiplication of the time and frequency variables ('**timeVarFrag**' and '**bFrequency**' respectively). The complete calculation is displayed below:

```
float currentBI = mix(minBI, maxBI, 0.5 + 0.5 * sin(timeVarFrag * bFrequency));

vec 3 bloomEffectColour = vec3(0.0);
```

The next stage of the fragment shader was to produce a value for the '**bloomEffectColour**' variable through the application of nesting for loops and the '**textureFunction()**'. The conditions of the for loops were integer variables that were limited by the '**blurEffectRadius**' constant. The full course of the loops produces a four-element vector variable ('**bloomEffectSample**') which possesses colour values that are subsequently assigned to the '**bloomEffectColour**' variable. This process can be viewed below:

```
for (int x = -blurEffectRadius; x <= blurEffectRadius ; x++)
{
        for (int y = -blurEffectRadius; y<= blurEffectRadius; y++)
        {
                vec2 offSetVector = vec2(x, y);
                vec4 bloomEffectSample = texture(textureSamplerVar, frag_in.outputTexCoords +
                offsetVector / textureSize(textureSamplerVar, 0));
                bloomEffectColour += bloomEffectSample.rgb;
        }
}
```

Once the loops have run their course, the '**bloomEffectColour**' variable undergoes further manipulation from the '**blurEffectRadius**' constant through utilisation of the '**float()**' function. It is at this stage that the intensity variable for the bloom effect is multiplied with the colour vector that has been created. Arbitrary values were added to this process to further refine the desired effect:

```
bloomEffectColour /=float((blurEffectRadius * 2 + 1,) * blurEffectRadius * 2 + 1));

bloomEffectColour *= currentBI;
```

The final step of the shader involves combining the two colour vectors created, and then assigning the value of the output variable '**fragColor**'. It is here that the bloom effect is applied to the texture to effectively produce the desired effect of the Sun's glow:

```
vec3 outputColour = textureColour.rgb + bloomEffectColour;

fragColor = vec4(outputColour, 1.0);
```

4. Project Files

All of the project files used in this application have been stored on GitHub. They can be accessed with the following link:

https://github.com/AidenD1799/AidenDonkinGraphicProject2023