Compulsory 1 Report

Sebastian Zapart

January 26, 2024

Contents

1 Introduction

This report is my compulsory one assignment for Math 3 and 3DProgramming course. It encompasses 3 sub-assignments of increasing complexity. In general, I need to create a mathematical expression then program it to produce an output file of vertices. Once I have a file of vertices, I must render the graph/3Dshape using OpenGL and its visualization pipeline. The source code can be found by clicking: "HERE" at github.com or by using this URL: https://github.com/Shrimpy02/SebastianZapart_Compulsory1.git.

OpenGL Transformations In OpenGL there are several important steps that have been taken to process and visualize all the given assignments. Firstly, OpenGL only projects coordinate between -1 to 1 to the view screen. Globally, coordinates wouldn't fit within these bounds. Therefore, I have defined several transformation matrices that transform the coordinates in accordance with a view space. This allows for camera movement and most importantly perspective.

2 Assignment One

The first assignment is to create a simple function with a fitting definition range and resolution. I decided to create a third-degree polynomial function because the graph it creates is more interesting and we get to see relevant changes in the "y" direction. Additionally, the colors for rendering are based on the Δy value. A straight line for example would show little mastery of either subject since it would have no color change nor change in its shape.

2.1 Math

---

f(x) regular and derivative

$$f(x) = x^3 - 4x^2 + 2x + 2, Df = [-1, 4] \tag{1}$$

$$f'(x) = 3x^2 - 8x + 2, Df = [-1, 4] \tag{2}$$

The resolution "h" and points "n"

$$h = \frac{b-a}{n} \tag{3}$$

$$h = \frac{4-(-1)}{10} = \frac{5}{10} = \frac{1}{2} \tag{4}$$

With n = 10 we get a resolution of 0.5 based on DfMin("a") and DfMax("b") and the actual number of points; n + 1 = 11. This gives a suitable graph smoothness and range of motion to see variations in Δy.

C++ Code The code for this math uses an x value iterated by the resolution h to calculate f(x) = y. These values are used to create a vertex struct stored in a vector. In addition to positional values the vertex struct also stores color values in rgb. As mentioned, the colors are based on the Δ values of each coordinate. Since Δx = h, the color red is constant at 0.5. Δz is always 0 since the graph is only two dimensional therefore the blue value is also always 0. The green value is normalized to a value between 0 and 5 so that once processed returns a suitable color value in the range 0 − 1 for OpenGL.

$$g = \left|\frac{\Delta y}{5}\right| \tag{5}$$

An example of how I created the code to generate all vertices of any graph.

Listing 1: Generate Vertices function

```
struct Vertex
{
        float x, y, z;
        float r, g, b;
};

void GenerateVertices()
{
        for (float i = defLow; i <= defHigh; i += resolution)
        {
                funcVertices.push_back(CreateVertex(i,
                                        Function(i),
                                        0,
                                        resolution,
                                        calcG(Function(i)),
                                        0));
        }
}
```

As an example, we can see the first four points manually calculated below:

$$f(-1) = (-1)^3 - 4 * (-1)^2 + 2 * (-1) + 2 = -5, (-1, -5) \tag{6}$$

$$f(-0.5) = (-0.5)^3 - 4 * (-0.5)^2 + 2 * (-0.5) + 2 = -0.13, (-0.5, -0.13) \tag{7}$$

$$f(0) = (0)^3 - 4 * (0)^2 + 2 * (0) + 2 = 2, (0, 2) \tag{8}$$

$$f(0.5) = (0.5)^3 - 4 * (0.5)^2 + 2 * (0.5) + 2 = 2.13, (0.5, 2.13) \tag{9}$$

---

f'(x) calculated for practice, its graph is not drawn.

$$f'(-1) = 3 * (-1)^2 - 8 * (-1) + 2 = 13, (-1, 13) \tag{10}$$

$$f'(-0.5) = 3 * (-0.5)^2 - 8 * (-0.5) + 2 = 6.75, (-0.5, 6.75) \tag{11}$$

$$f'(0) = 3 * (0)^2 - 8 * (0) + 2 = 2, (0, 2) \tag{12}$$

$$f'(0.5) = 3 * (0.5)^2 - 8 * (0.5) + 2 = -1.25, (0.5, -1.25) \tag{13}$$

2.2 Opengl

Because of the transformation matrices mentioned earlier, displaying the graph is as simple as reading the file to a vector of floats. Then assigning the vector to the VBO, then with simple draw criteria's like "GL-LINE-STRIP" it draws a line between each pair of points passed into the VAO.

Here you can see how the vector is bound and attributes for position and color.

Listing 2: Binding Vertices

```
void bindBufferData(vector<float> _vertices)
{

        glBindBuffer(GL_ARRAY_BUFFER, VBO);
        glBufferData(GL_ARRAY_BUFFER,
                        _vertices.size() * sizeof(_vertices.data()),
                        _vertices.data(),
                         GL_STATIC_DRAW);
        }

void createAttributePointers()
{
        // position
        glVertexAttribPointer(0, 3, GL_FLOAT,
                                    GL_FALSE,
                                    6 * sizeof(float),
                                    (void*)0);
        glEnableVertexAttribArray(0);

        // color
        glVertexAttribPointer(1, 3, GL_FLOAT,
                                    GL_FALSE,
                                    6 * sizeof(float),
                                    (void*)(3 * sizeof(float)));
        glEnableVertexAttribArray(1);
}
```

In figure 1 you can see the resulting output of f(x) generated with 11 points and color based on Δy. As a control I created the same function in GeoGebra and got the same result as seen in figure 2.

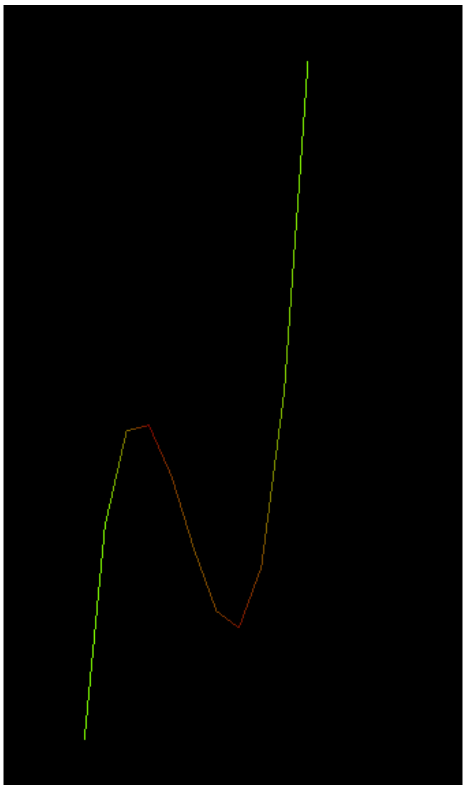
Figure 1: Graph of f(x) visualized with OpenGL.

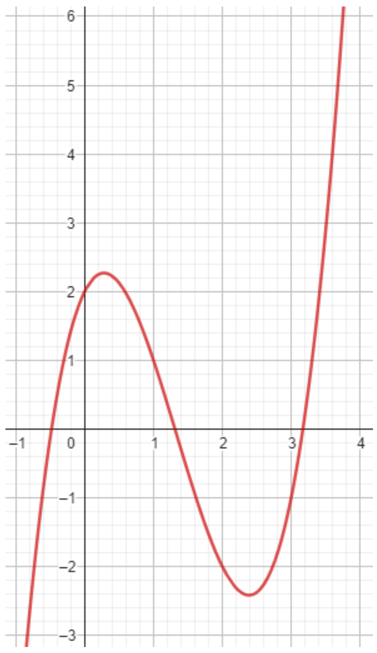

Figure 2: Graph of f(x) visualized with GeoGebra.

3 Assignment Two

For the second assignment we introduce a third dimension by creating a vector function f(t) = (x(t), y(t), z(t)). By combining all the functions by axis, we can create a dynamic vector that can produce any shape we want. For this assignment I followed an example using trigonometry to create a spiral.

3.1 Math

---

f(t) = f(t) = (x(t), y(t), z(t)) for a spiral along the z = t axis.

$$x(t) = cos(\frac{\pi * t}{10}) \tag{14}$$

$$y(t) = sin(\frac{\pi * t}{10}) \tag{15}$$

$$z(t) = t \tag{16}$$

$$f(t) = (cos(\frac{\pi * t}{10}), sin(\frac{\pi * t}{10}), t) \tag{17}$$

I defined a spiral between 0 and 50. This gives 50 points for the spiral and a resolution h = 1.

c++ Code The function operates much like in the first assignment with a for loop iterating from z = 0 to z = 50 by a resolution of 1. The same is for the colors, however, this time we get to see both red and green colors since both Δx and Δy fluctuate. Blue or Δz is constant at our resolution = 1.

---

here is an example of the coded function creating these vertices.

Listing 3: Generate Vertices function

```
struct Vertex
{
        float x, y, z;
        float r, g, b;
};

void GenerateVertices()
{
        for (float i = defLow; i <= defHigh; i += resolution)
        {
                funcVertices.push_back(CreateVertex(VecFuncX(i),
                                        VecFuncY(i),
                                        i,
                                        calcR(VecFuncX(i)),
                                        calcG(VecFuncY(i)),
                                        i));
        }
}
```

3.2 Opengl

The VAO stored geometry is processed by the OpenGL shader pipeline. There are six steps in total, each with their own shader. For this project I have configured two steps to fit my use case specifically.

VertexShader The vertex shader takes the vertices given by the VAO and returns its GL-Position for drawing. This is where the transformation matrixes modify the output to a custom view space that allows perspective and movement.

FragmentShader Once GL-Position is returned for a vertex it is passed to the fragment shader to get and assigned color. OpenGL as a whole process is not this simple, this general explanation is to get a better understanding of the vertex and fragment shaders I used in this project as seen below.

Listing 4: Vertex and Fragment Shader

```
// Vertex Shader
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;

out vec3 ourColor;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{

 gl_Position = projection * view * model * vec4(aPos, 1.0);
 ourColor = aColor;

}

// Fragment Shader
#version 330 core

out vec4 FragColor;
in vec3 ourColor;

void main(){

    FragColor = vec4(ourColor, 1.0);

}
```

In figure 3 you can see the resulting output of f(t) generated with 50 points and color based on Δx and Δy. As a control I created the same function in GeoGebra and got the same result as seen in figure 4. Note the rotation of the shape stems from OpenGL using a right-handed system while GeoGebra uses a left-handed axis system.

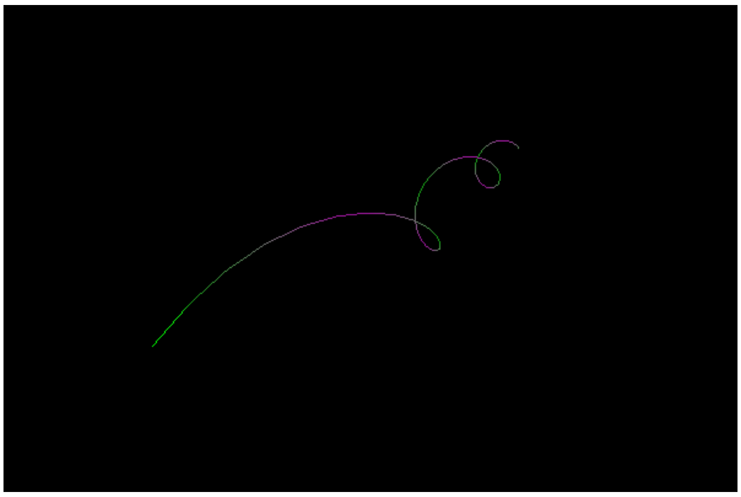

Figure 3: Spiral of f(t) visualized with OpenGL.

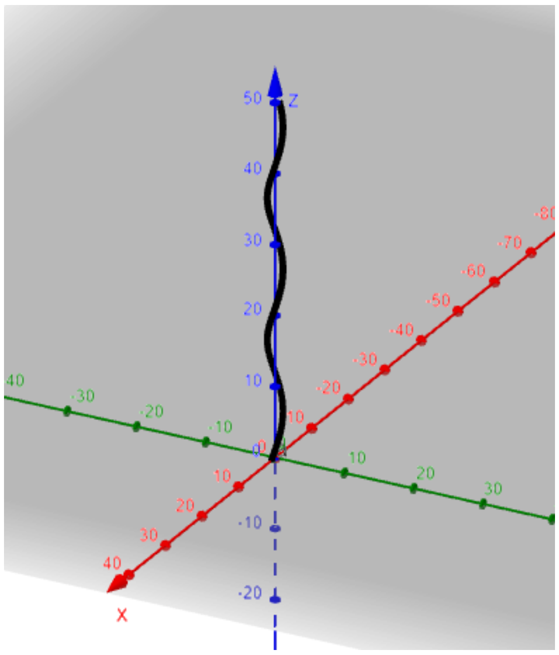

Figure 4: Spiral of f(t) visualized with GeoGebra.

4 Assignment Three

For the third assignment I was tasked to create a function of two variables and use it in addition to OpenGL to render a plane, since f(x,y) = z. I decided the function I created should be trigonometric because the fluctuating values would give an interesting shape to the plane.

4.1 Math

Function for f(x,y).

$$f(x, y) = sin(x) * cos(y) * (\frac{x + y}{10}), Dfx[0, 10] * Dfy[0, 10] \tag{18}$$

I defined a resolution h = 0.5 since and a Df range between 0 and 10 for each axis. This produces a large plane with a good definition of 9600 points. However, because of the fluctuating Δ values the colors would be fragmented throughout the plane. Therefore, I choose to rather use the positional values normalized and processed to a value between 0-1.

Color for rgb normalization. Note color blue *3 because its value is on average much lower than x or y so it wouldn't be visible.

$$r = \left|\frac{x}{10}\right| \tag{19}$$

$$g = \left|\frac{y}{10}\right| \tag{20}$$

$$b = \left|\frac{z * 3}{10}\right| \tag{21}$$

The function in code is much the same as the previous examples. Since we will render the plane using triangles, and you need 6 points per square. Additionally, we must generate in both the x and y directions. This means we need to nest two for loops for the appropriate depth as you can see in the code below.

Listing 5: Generate Vertices for plane function

```
void GenerateVertices()
{
for (float x = defXMin; x < defXMax; x += resolution)
{
        for(float y = defYMin; y < defYMax; y += resolution)
        {
        // Creates first triangle
        // Vertex 1
        float z = twoVariableFunction(x, y);
        Vertices.push_back(CreateVertex(x, y, z,
                                         calcR(x), calcG(y), calcB(z)));

        // Vertex 2
        z = twoVariableFunction(x+resolution, y);
        Vertices.push_back(CreateVertex(x + resolution, y, z,
                                        calcR(x + resolution), calcG(y),
                                        calcB(z)));

        // Vertex 3
        z = twoVariableFunction(x, y + resolution);
        Vertices.push_back(CreateVertex(x, y + resolution, z,
                                        calcR(x), calcG(y + resolution),
                                        calcB(z)));

        // Creates second triangle
        // Vertex 4
        z = twoVariableFunction(x + resolution, y);
        Vertices.push_back(CreateVertex(x + resolution, y, z,
                                         calcR(x + resolution), calcG(y),
                                         calcB(z)));

        // Vertex 5
        z = twoVariableFunction(x, y + resolution);
        Vertices.push_back(CreateVertex(x, y + resolution, z,
                                         calcR(x), calcG(y + resolution),
                                         calcB(z)));

        // Vertex 6
        z = twoVariableFunction(x + resolution, y + resolution);
        Vertices.push_back(CreateVertex(x + resolution,
                                         y + resolution, z,
                                        calcR(x + resolution),
                                        calcG(y + resolution),
                                        calcB(z)));
        }
}
}
```

4.2 Opengl

Rending in OpenGL is easy once everything is set up. The render loop runs while the glfwWindowShouldClose bool is true. All that's left, more or less, is to call the OpenGLDraw command. The prosses can be optimized by using an attached EBO and indices vector. This prevents vertices from separate triangles from being drawn on top of each other, preferring rather to share vertices. I have not implemented a functional indices vector for this assignment, although the EBO system is functional in the project file.

Listing 6: Generate Vertices for plane function

```
void drawVertexGeometry(bool _useIndices)
{
        glBindVertexArray(VAO);

        if (_useIndices)
                glDrawElements(GL_TRIANGLES,
                                        numVertices,
                                        GL_UNSIGNED_INT, 0);
        else
                glDrawArrays(GL_TRIANGLES,
                                     0, numVertices);

        glBindVertexArray(0);
}
```

Listing 7: OpenGL Render loop

```
while (!glfwWindowShouldClose(window))
{
        // ----------- Rendering options -----------
        updateDeltaTime();
        processInput(window);

        glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        // ----------- Drawing -----------

        // Transformations
        myShader.processTransformations(myCamera, SCR_WIDTH, SCR_HEIGHT);

        // Rendering
        myGeometry.drawVertexGeometry(false);

        // GLFW: swap buffers and process events
        // ------------------------------------------------
        glfwSwapBuffers(window);
        glfwPollEvents();
}
```

In figure 5, 6 and 7 you can see the resulting output of f(x,y) generated with 9600 points and color based on vertex position at different angels. As a control I created the same function in GeoGebra and got the same result as seen in figure 8.

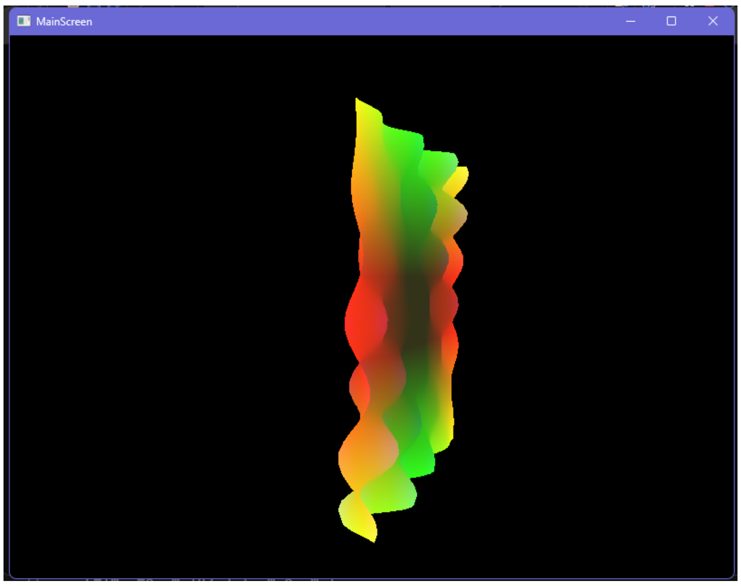

Figure 5: Plane f(x,y) left view visualized with OpenGL.

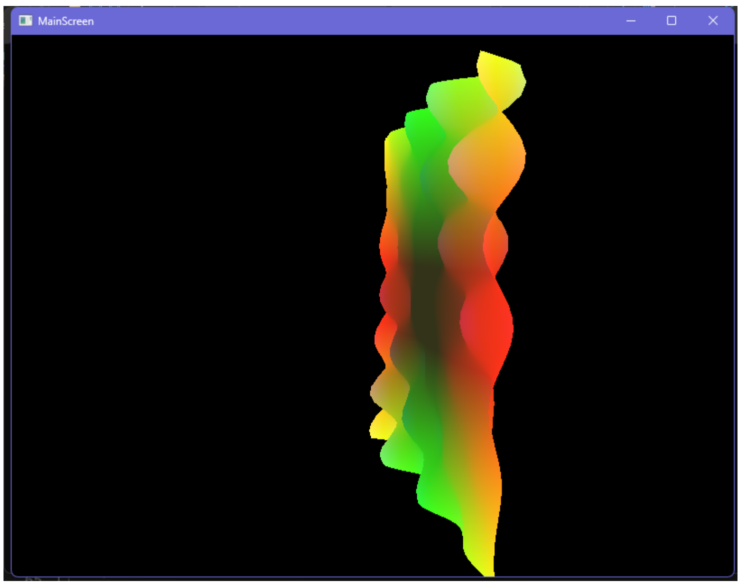

Figure 6: Plane f(x,y) right view visualized with OpenGL.

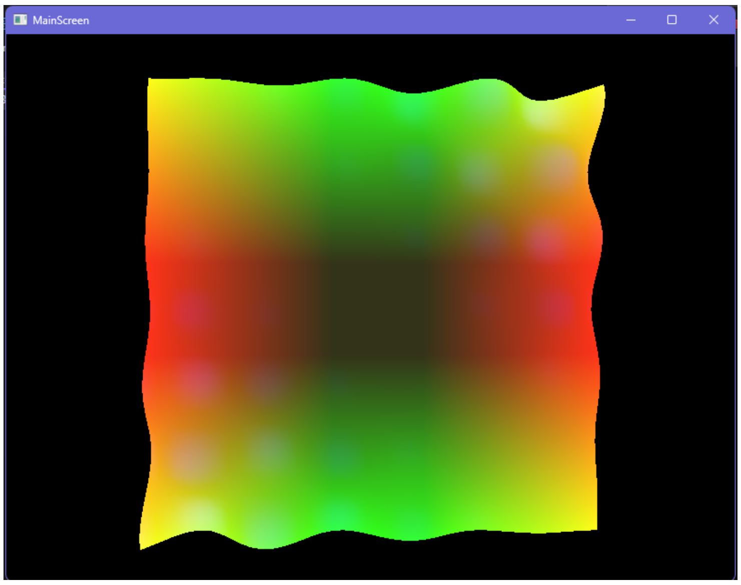

Figure 7: Plane f(x,y) front view visualized with OpenGL.

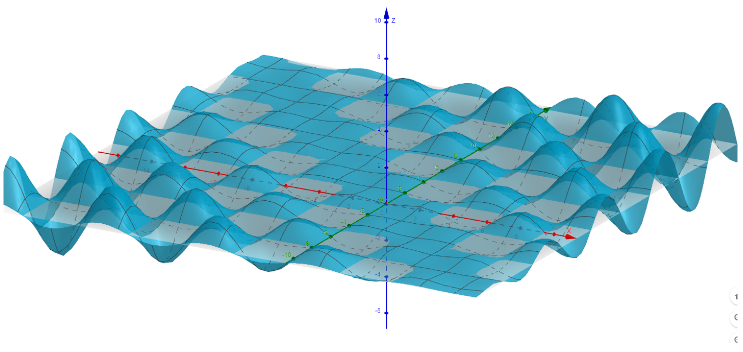

Figure 8: Plane f(x,y) visualized with GeoGbra.

5 Discussion

For this compulsory assignment I have done a lot of preparation in OpenGL for a smooth programming experience once I started. This means the assignment was verry enjoyable, especially since the results were correct in accordance with the math done beforehand. However, there were some shortcomings.

I could have implemented an EBO indices system for optimizing the geometry for the third assignment. Additionally, the school tempo was too slow for this assignment and therefore I had to learn from outside sources to complete the assignment that's supposed to be within school parameters.

All in all I enjoyed the experience because I learned a lot about renting from gpu and the GL pipeline in general. Next assignment I will focus more next time on preparation and error handling.