**Data visualization – principles and practices, 2nd edition  
Chapter 2 – project 1**

1. **Functions**

I experimented with the functions presented in Table 1. The implementation of these functions is presented in Listing 1. Next three pages shows the result of rendering these functions.

|  |  |
| --- | --- |
| **Function** | **Equation** |
| Gaussian |  |
| Parabola |  |
| Ripple |  |
| Wave |  |
| Sin |  |
| Fenc |  |

**Table 1 – Function formulas.**

A screen shot of a computer program

AI-generated content may be incorrect.

**Listing 1 – Function Implementation.**

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A computer screen shot of a computer screen

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

1. **Lightning and shading**

The lightning and shading depend entirely on the angle between the light source vector and the surface normal vector (N). To display realistic 3D shading, we need to calculate the surface normal vector (i.e., the normal at the four vertices of the quad). We will do that analytically using the following formula, which is implemented in Listing 2 for all functions(i.e., normalized vectors):

A screenshot of a computer program

AI-generated content may be incorrect.

**Listing 2 – Surface normal vector calculation.**

1. **Height based coloring**

To better visualize elevation changes, increase the contrast, and make data interpretation better, I implemented a height-based coloring. The scheme I follow is blue-to-red (i.e., no green component) on the z-axis. We start by normalizing the height of the vertex, then mapping the normalized height to a color using the following formulas, which are implemented in Listing 3.

A screenshot of a computer program

AI-generated content may be incorrect.

**Listing 3 – Blue-to-Red height based coloring.**

1. **Drawing the Functions**

We define quadrilaterals as surface elements. These quadrilaterals are implemented in the Quad class presented in **Listing 4**, this class encapsulates the steps necessary to render one quadrilateral element in our plots.

A computer code with text

AI-generated content may be incorrect.

**Listing 4 – Quad class.**

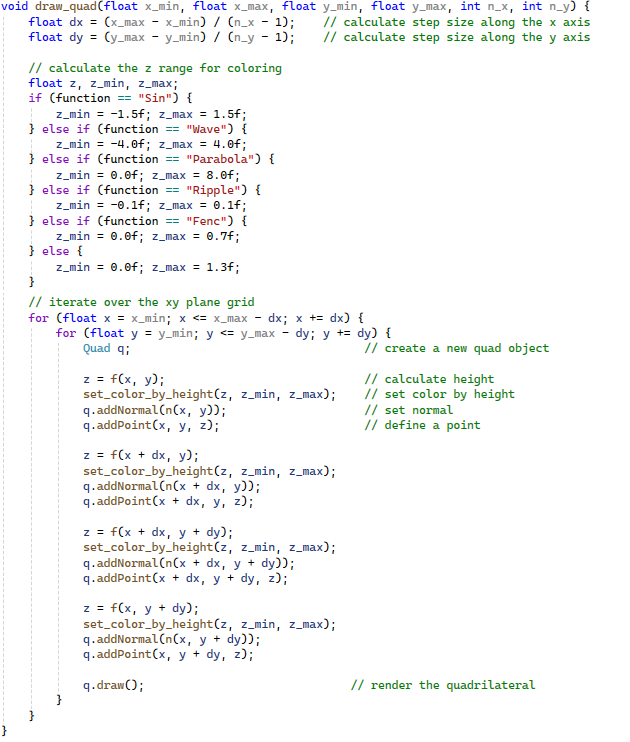
The core rendering loop is defined in the draw function explained in **Listing 5**. This function initiates the frame, chooses the domain and resolution of the function, renders the 3D surface and its base grid, and finally displays the result.

A white background with many lines and numbers

AI-generated content may be incorrect.

**Listing 5 – Rendering loop.**

To convert the mathematical function over a specified domain into a series of OpenGL quadrilateral primitives, we utilize the draw quad method presented in Listing 6. The method starts by calculating the step size along the x and y axes. It then calculates the z range for coloring. The method iterates over the xy plane grid, creates a new quad object, calculates the height of the point, its color, its normal, and adds it to the quadrilateral. After the method finishes processing the four points of the current quadrilateral, it renders it.



**Listing 6 – Draw quadrilaterals.**

1. **Viewing**

The viewing method presented in Listing 7 dictates how the 3D world is mapped to the 2D screen by setting up the camera and its lens. It also specifies which part of the window the rendered scene should occupy.

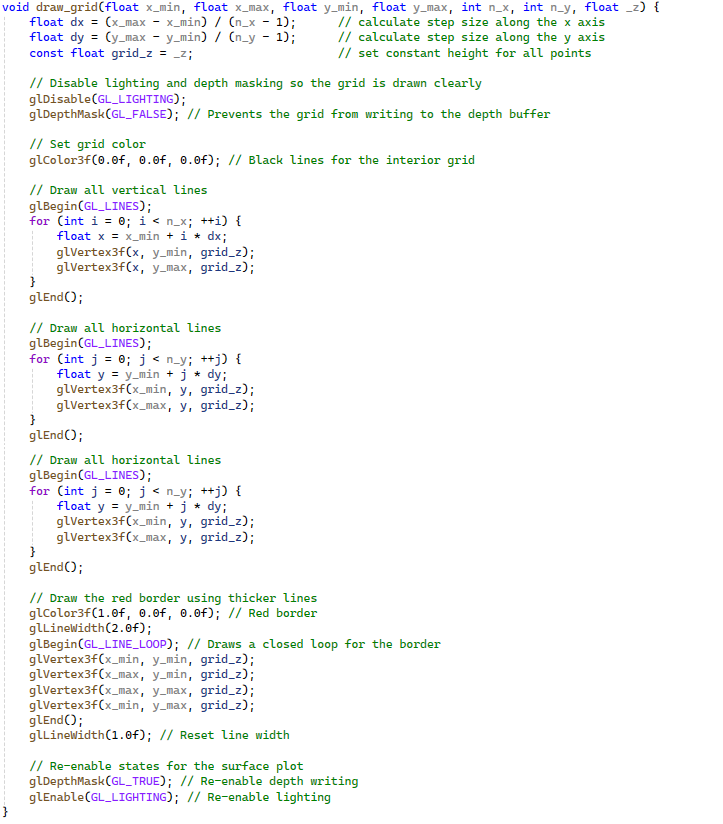
A screenshot of a computer code

AI-generated content may be incorrect.

**Listing 7 – Viewing method.**

1. **Grid drawing**

To draw a grid underneath the shape, we use the draw grid function presented in Listing 8. This function starts by calculating the step size along the x and y axes, setting the same height for all grid points. The function then disables the lighting and depth masking to color grid lines properly and prevent the grid from interleaving the rendered shape. The function proceeds to drawing the grid inside the for loops. It then draws a thicker red boundary and ends with re enabling the lightning and depth masking.



**Listing 8 – Grid drawing method.**

1. **Controlling the shading and light position**

To control the shading and light position we will use keyboard keys. The keyboard function listed in Listing 9 defines handlers for the following keys. It also calls the function update light position presented in Listing 10 to update the light position.

's': toggle shading mode.

'x' | 'X': move light along the x axis.

'y' | 'Y': move light along the y axis.

'z' | 'Z': move light along the Z axis.

'Esc': to end the execution.

A screenshot of a computer code

AI-generated content may be incorrect.

**Listing 9 – Keyboard press handler.**

A computer code with text

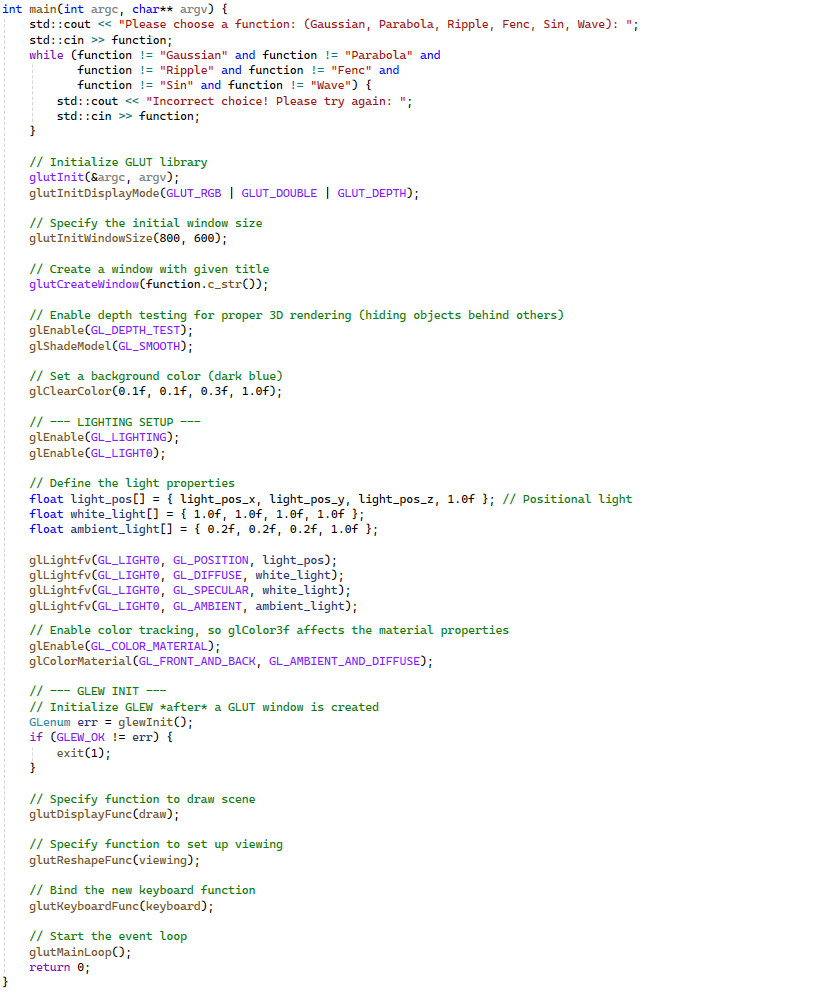
AI-generated content may be incorrect.

**Listing 10 – Update light position method.**

1. **Entry point and initializer**

The main function is the entry point and initializer of our entire C++/OpenGL program. It performs the following:

1. Request and receive user's function choice.
2. Initialize the glut library.
3. Specify the initial window size.
4. Create a window with given title.
5. Enable depth testing for proper 3D rendering (hiding objects behind others).
6. Set a background color (dark blue).
7. Set up the lightning.
8. Glue initialization.
9. Registers the draw() function as the primary rendering routine.
10. Registers the viewing() function to be called whenever the window is resized or initially created.
11. Registers the function to handle standard key presses.
12. Transfers control to the GLUT event processing loop.



**Listing 11 – Main function (entry point).**