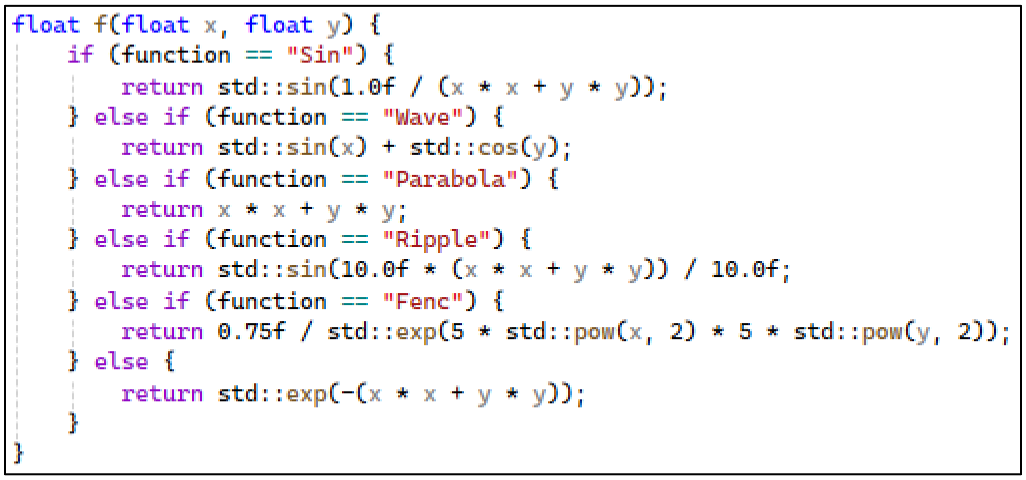
Data visualization – principles and practices, 2nd edition
Chapter 2 – project 1

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.

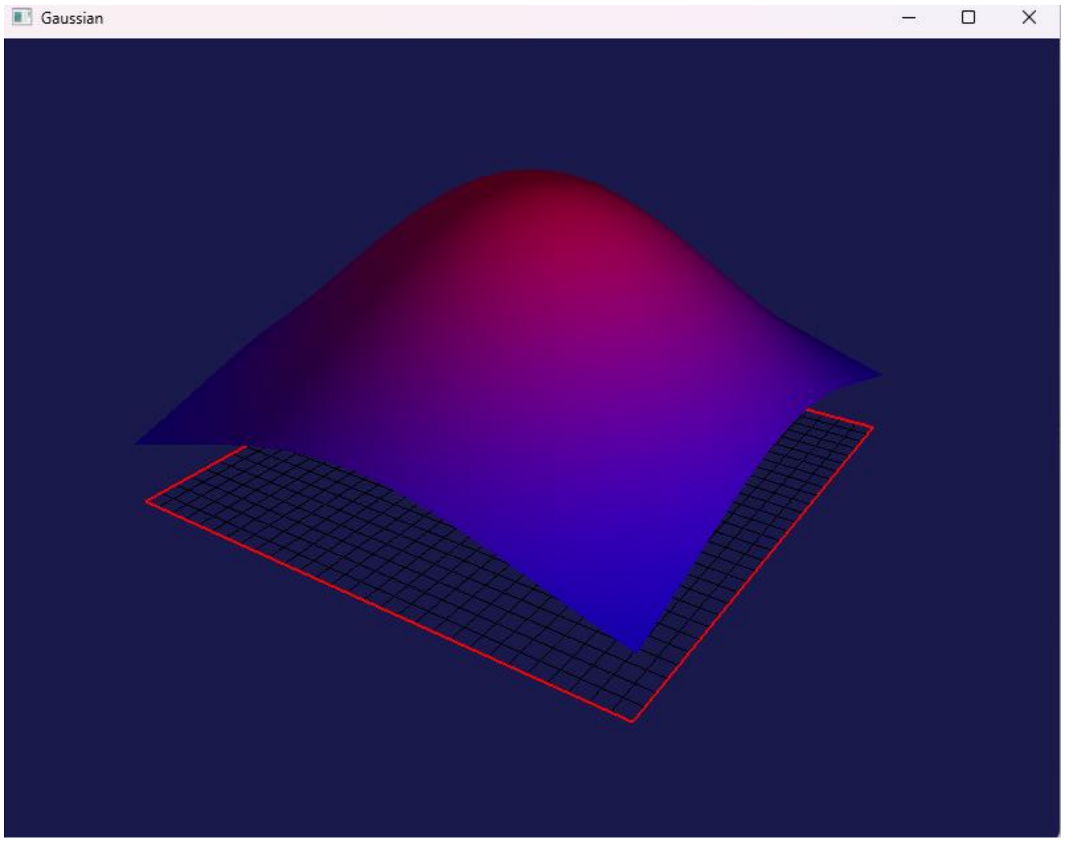
Function	Equation
Gaussian	$f(x,y) = e^{-(x^2+y^2)}$
Parabola	$f(x,y) = x^2 + y^2$
Ripple	$f(x,y) = \frac{\sin\left(10 \times (\mathrm{x}^2 + \mathrm{y}^2)\right)}{10}$
Wave	$f(x,y) = \sin(x) + \cos(y)$
Sin	$f(x,y) = \sin\left(\frac{1}{x^2 + y^2}\right)$
Fenc	$f(x,y) = \frac{0.75}{e^{5x^2 \times 5y^2}}$

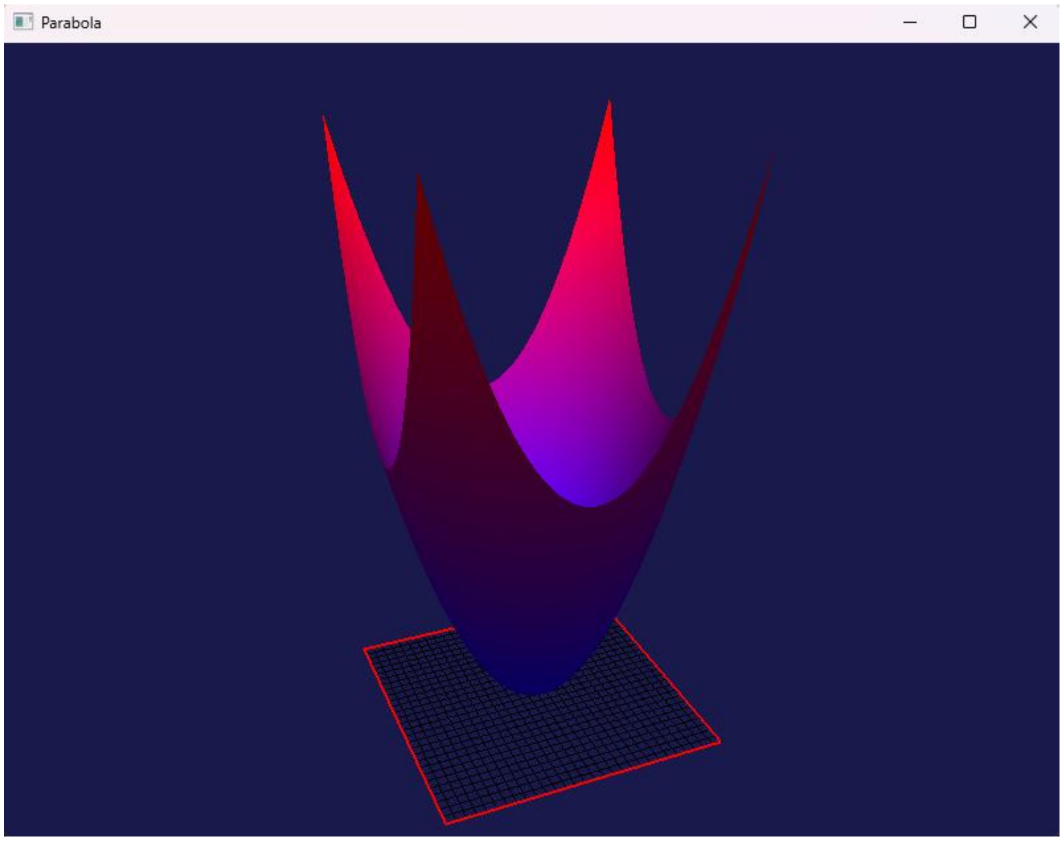
Table 1 – Function formulas.

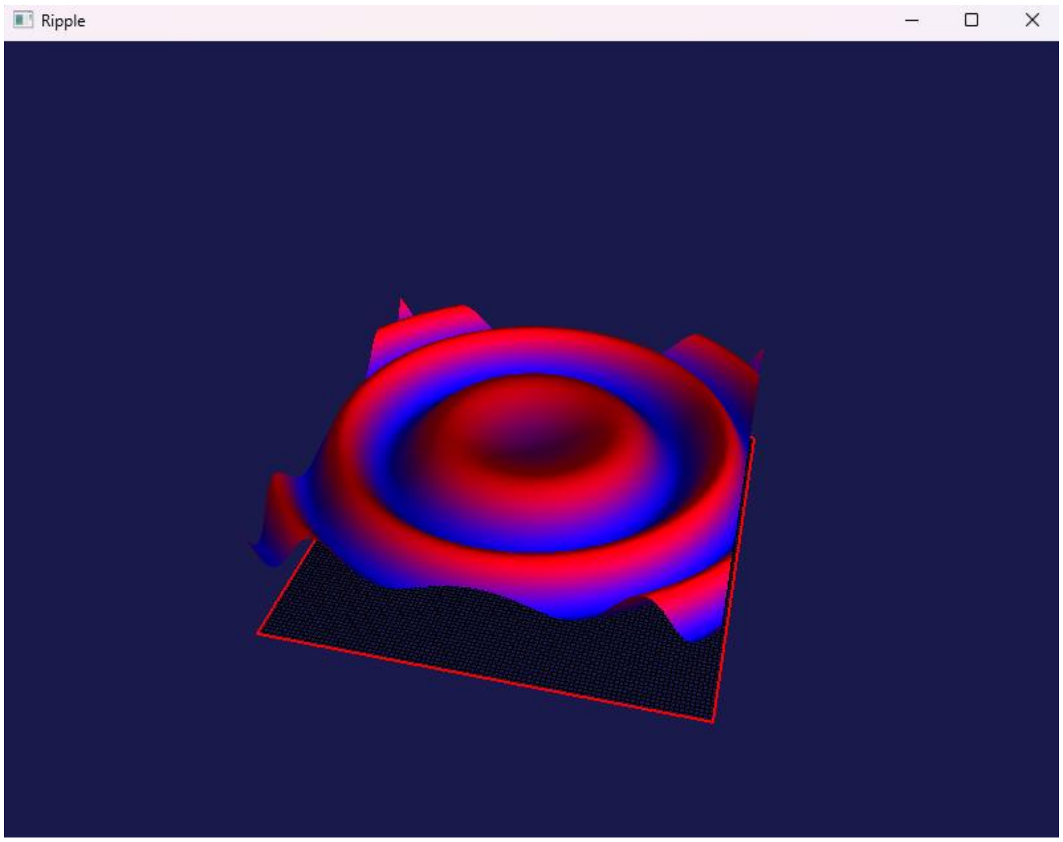


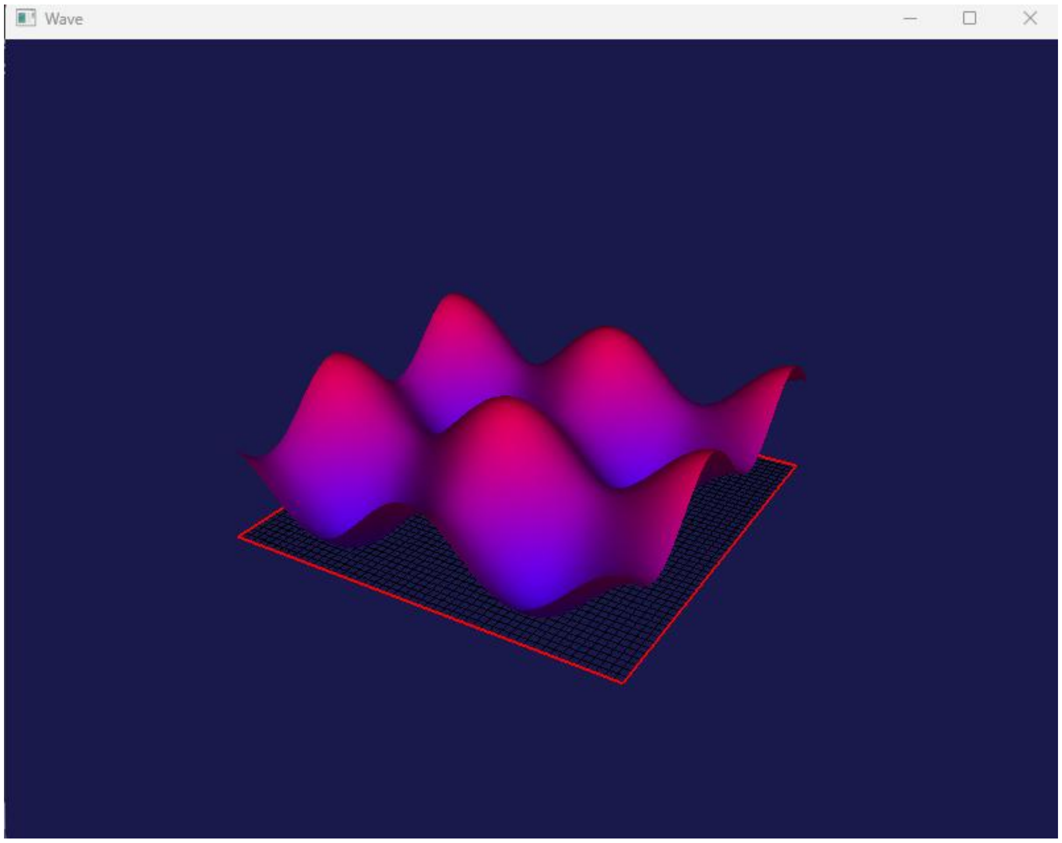
```
float f(float x, float y) {
    if (function == "Sin") {
        return std::sin(1.0f / (x * x + y * y));
    } else if (function == "Wave") {
        return std::sin(x) + std::cos(y);
    } else if (function == "Parabola") {
        return x * x + y * y;
    } else if (function == "Ripple") {
        return std::sin(10.0f * (x * x + y * y)) / 10.0f;
    } else if (function == "Fenc") {
        return 0.75f / std::exp(5 * std::pow(x, 2) * 5 * std::pow(y, 2));
    } else {
        return std::exp(-(x * x + y * y));
    }
}
```

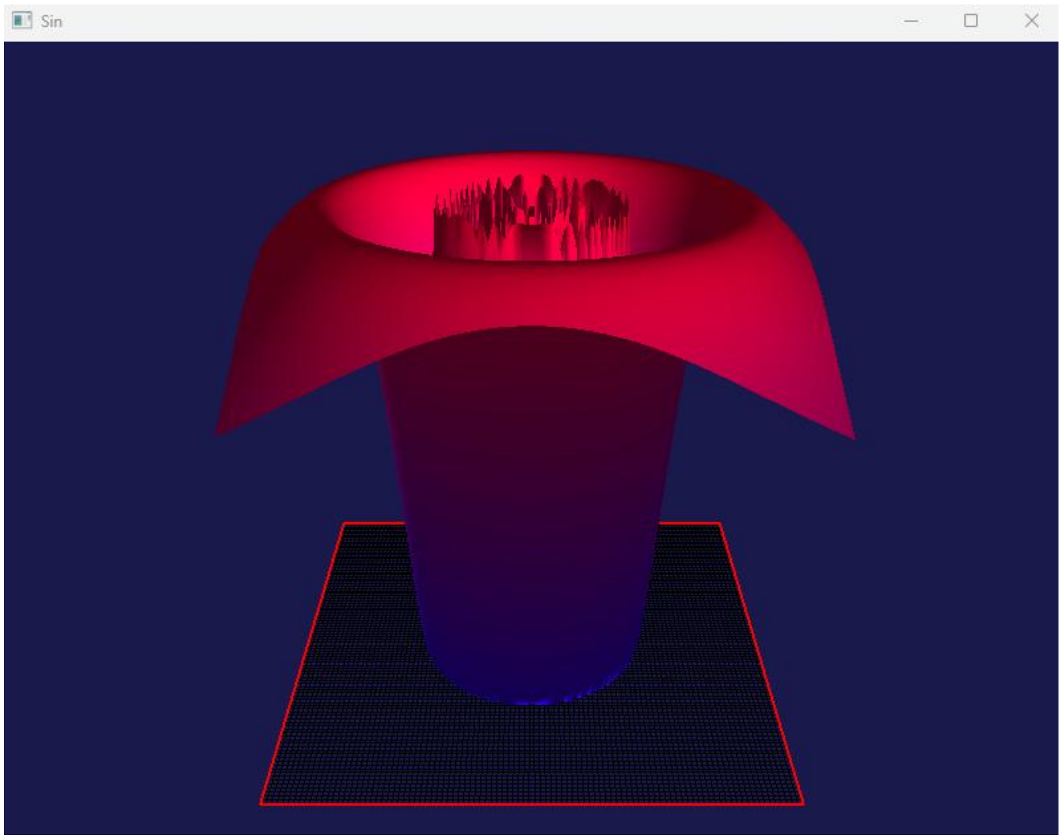
Listing 1 – Function Implementation.

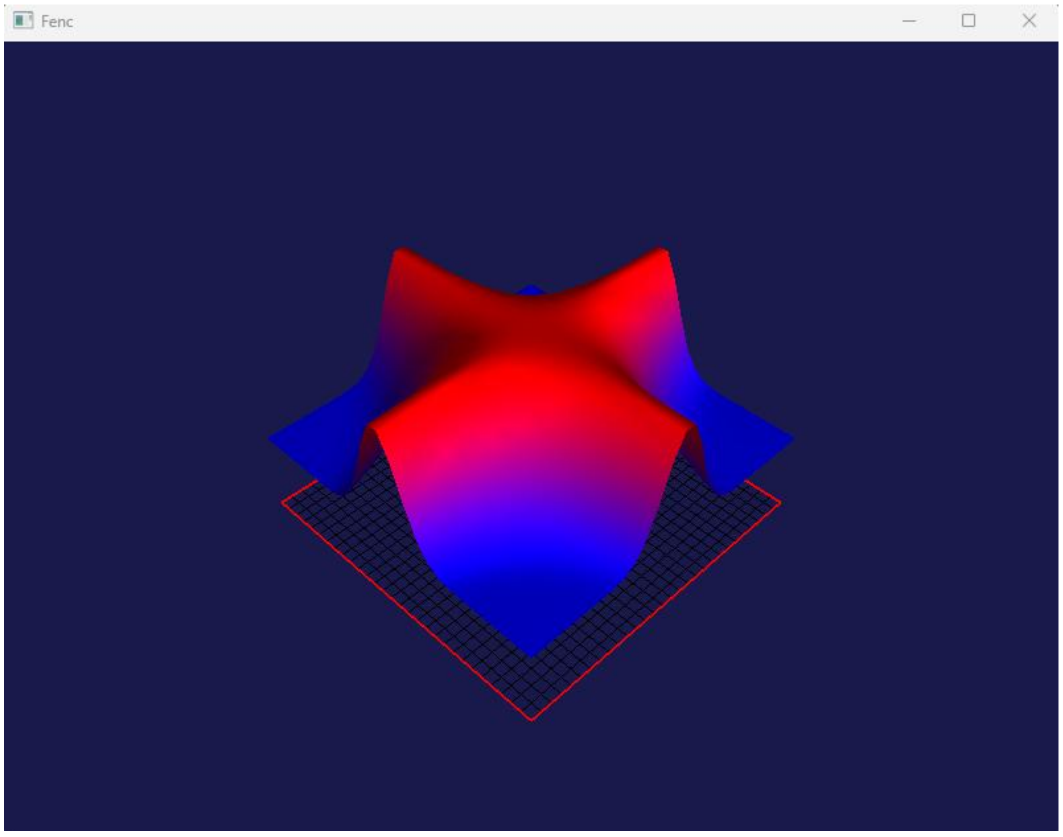

Gaussian


Parabola


Ripple


Wave

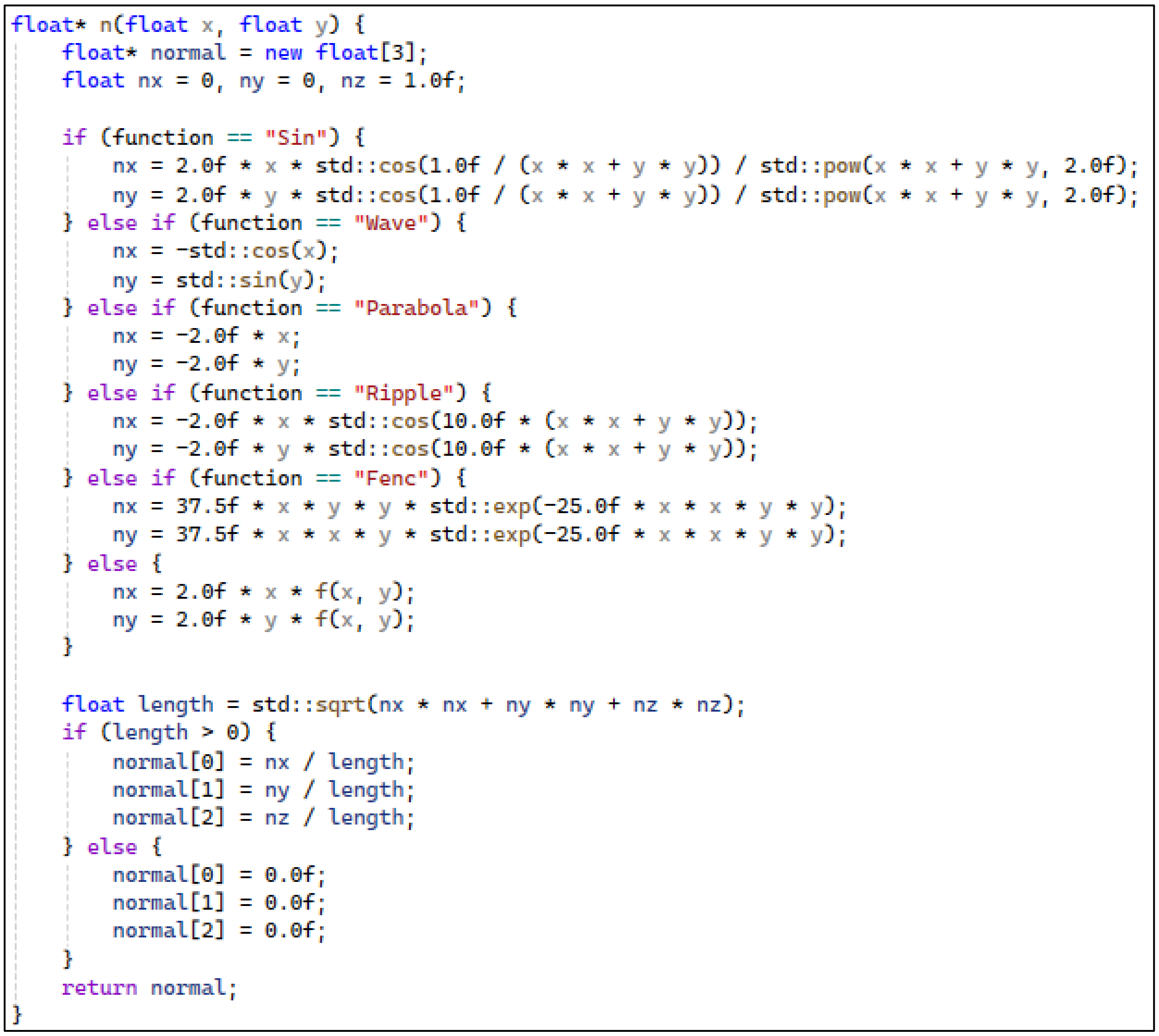

Sin


Fenc

2. 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):

$$N = \left\langle -\frac{\partial f}{\partial x}, -\frac{\partial f}{\partial y}, 1 \right\rangle$$



```cpp
float* n(float x, float y) {
    float* normal = new float[3];
    float nx = 0, ny = 0, nz = 1.0f;

    if (function == "Sin") {
        nx = 2.0f * x * std::cos(1.0f / (x * x + y * y)) / std::pow(x * x + y * y, 2.0f);
        ny = 2.0f * y * std::cos(1.0f / (x * x + y * y)) / std::pow(x * x + y * y, 2.0f);
    } else if (function == "Wave") {
        nx = -std::cos(x);
        ny = std::sin(y);
    } else if (function == "Parabola") {
        nx = -2.0f * x;
        ny = -2.0f * y;
    } else if (function == "Ripple") {
        nx = -2.0f * x * std::cos(10.0f * (x * x + y * y));
        ny = -2.0f * y * std::cos(10.0f * (x * x + y * y));
    } else if (function == "Fenc") {
        nx = 37.5f * x * y * y * std::exp(-25.0f * x * x * y * y);
        ny = 37.5f * x * x * y * std::exp(-25.0f * x * x * y * y);
    } else {
        nx = 2.0f * x * f(x, y);
        ny = 2.0f * y * f(x, y);
    }

    float length = std::sqrt(nx * nx + ny * ny + nz * nz);
    if (length > 0) {
        normal[0] = nx / length;
        normal[1] = ny / length;
        normal[2] = nz / length;
    } else {
        normal[0] = 0.0f;
        normal[1] = 0.0f;
        normal[2] = 0.0f;
    }
    return normal;
}
```

Listing 2 – Surface normal vector calculation.

3. 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.

$$R = t$$
$$G = 0.0f$$
$$B = 1.0f - t$$

```
void set_color_by_height(float z, float z_min, float z_max) {
    float t = (z - z_min) / (z_max - z_min);

    if (t < 0.0f) t = 0.0f;
    if (t > 1.0f) t = 1.0f;

    float R = t;
    float G = 0.0f;
    float B = 1.0f - t;

    glColor3f(R, G, B);
}
```

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

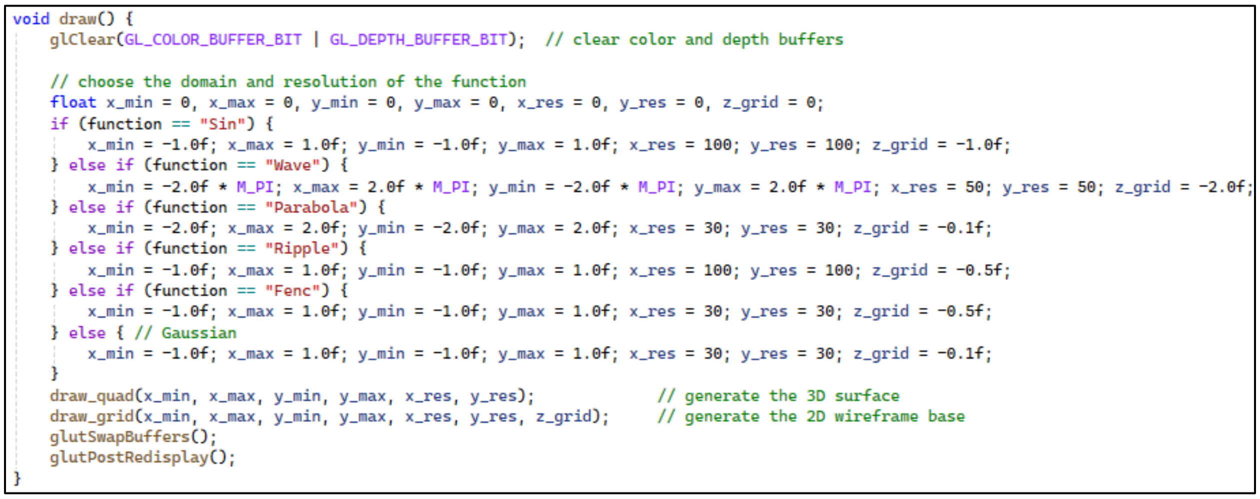
4. Drawing the Functions

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.

```
class Quad {
public:Quad()
{
    glBegin(GL_QUADS);         // start the drawing process
}
      void addPoint(float x, float y, float z) {
          glVertex3f(x, y, z);        // define one corner of the quadrilateral
      }

      void addNormal(float* n) {
          glNormal3f(n[0], n[1], n[2]);     // set the normal for ligtning and shading
      }

      void draw() {
          glEnd();    // end the definition of the quadrilateral
      }
};
```

Listing 4 – Quad class.

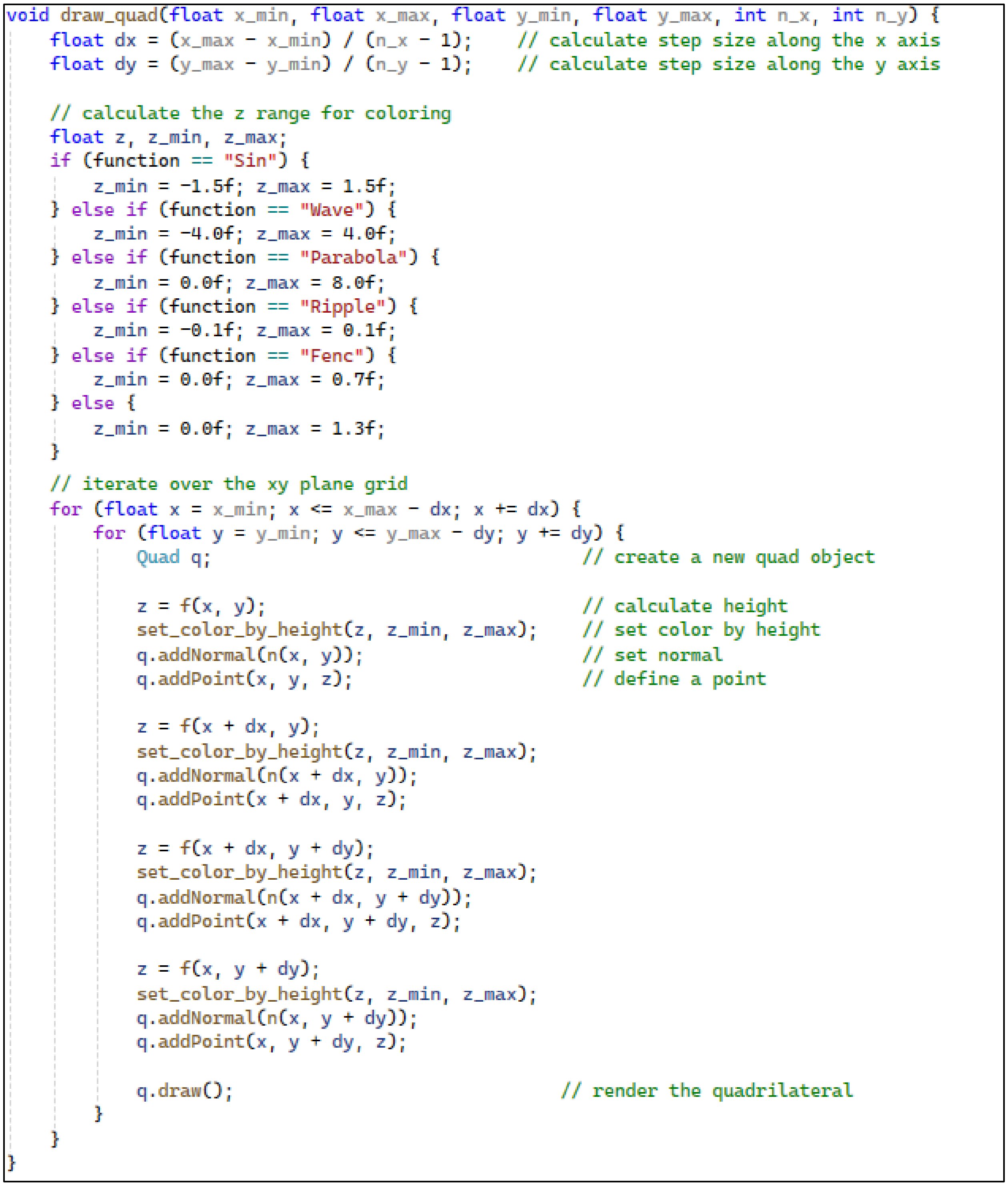
The core rendering loop is defined in the draw function explained in **Listing 5**. This function initiates the frame, chooses the domain and resolution of the function, renders the 3D surface and its base grid, and finally displays the result.



```
void draw() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  // clear color and depth buffers

    // choose the domain and resolution of the function
    float x_min = 0, x_max = 0, y_min = 0, y_max = 0, x_res = 0, y_res = 0, z_grid = 0;
    if (function == "Sin") {
        x_min = -1.0f; x_max = 1.0f; y_min = -1.0f; y_max = 1.0f; x_res = 100; y_res = 100; z_grid = -1.0f;
    } else if (function == "Wave") {
        x_min = -2.0f * M_PI; x_max = 2.0f * M_PI; y_min = -2.0f * M_PI; y_max = 2.0f * M_PI; x_res = 50; y_res = 50; z_grid = -2.0f;
    } else if (function == "Parabola") {
        x_min = -2.0f; x_max = 2.0f; y_min = -2.0f; y_max = 2.0f; x_res = 30; y_res = 30; z_grid = -0.1f;
    } else if (function == "Ripple") {
        x_min = -1.0f; x_max = 1.0f; y_min = -1.0f; y_max = 1.0f; x_res = 100; y_res = 100; z_grid = -0.5f;
    } else if (function == "Fenc") {
        x_min = -1.0f; x_max = 1.0f; y_min = -1.0f; y_max = 1.0f; x_res = 30; y_res = 30; z_grid = -0.5f;
    } else { // Gaussian
        x_min = -1.0f; x_max = 1.0f; y_min = -1.0f; y_max = 1.0f; x_res = 30; y_res = 30; z_grid = -0.1f;
    }
    draw_quad(x_min, x_max, y_min, y_max, x_res, y_res);            // generate the 3D surface
    draw_grid(x_min, x_max, y_min, y_max, x_res, y_res, z_grid);    // generate the 2D wireframe base
    glutSwapBuffers();
    glutPostRedisplay();
}
```

Listing 5 – Rendering loop.

To convert the mathematical function $z = f(x, y)$ over a specified domain $(x_{min}\ to\ y_{max})$ 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.

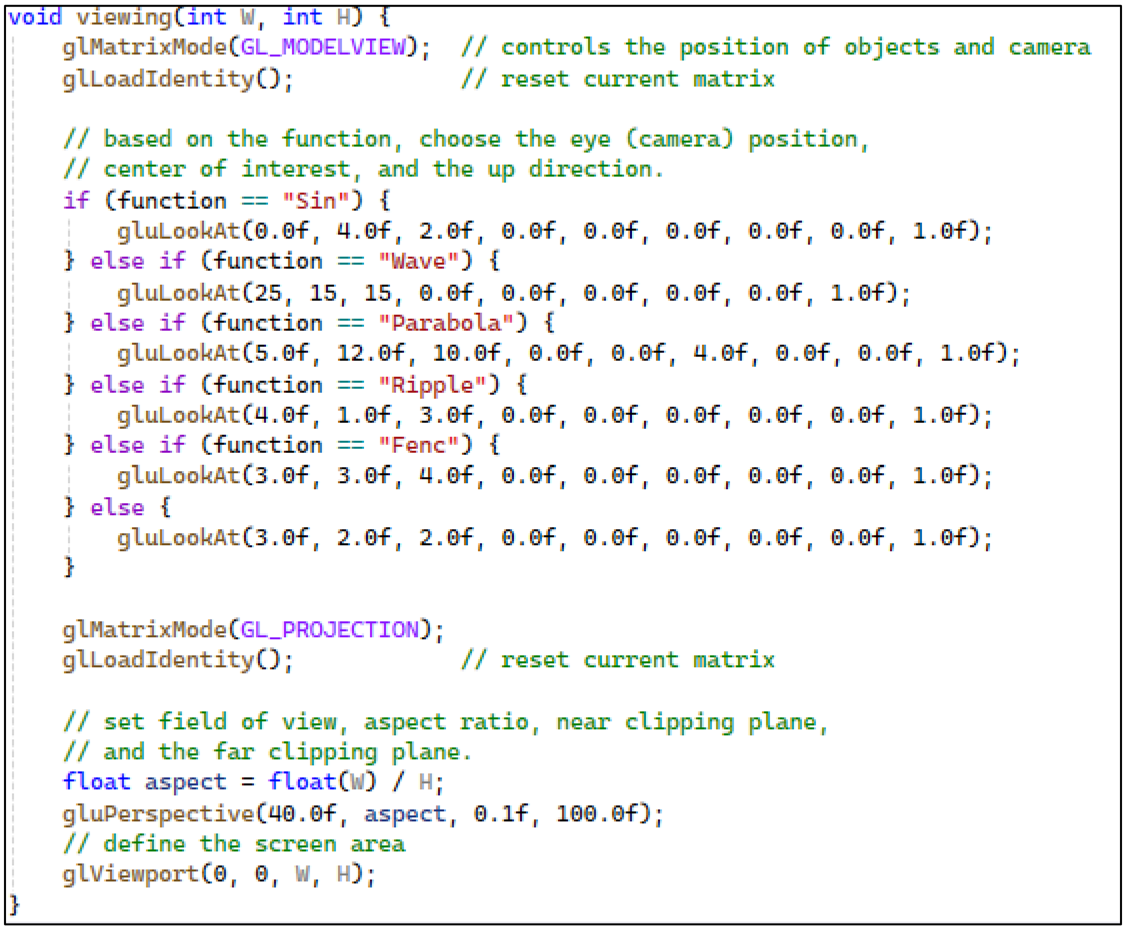


```
void draw_quad(float x_min, float x_max, float y_min, float y_max, int n_x, int n_y) {
    float dx = (x_max - x_min) / (n_x - 1);    // calculate step size along the x axis
    float dy = (y_max - y_min) / (n_y - 1);    // calculate step size along the y axis

    // calculate the z range for coloring
    float z, z_min, z_max;
    if (function == "Sin") {
        z_min = -1.5f; z_max = 1.5f;
    } else if (function == "Wave") {
        z_min = -4.0f; z_max = 4.0f;
    } else if (function == "Parabola") {
        z_min = 0.0f; z_max = 8.0f;
    } else if (function == "Ripple") {
        z_min = -0.1f; z_max = 0.1f;
    } else if (function == "Fenc") {
        z_min = 0.0f; z_max = 0.7f;
    } else {
        z_min = 0.0f; z_max = 1.3f;
    }

    // iterate over the xy plane grid
    for (float x = x_min; x <= x_max - dx; x += dx) {
        for (float y = y_min; y <= y_max - dy; y += dy) {
            Quad q;                                  // create a new quad object

            z = f(x, y);                             // calculate height
            set_color_by_height(z, z_min, z_max);    // set color by height
            q.addNormal(n(x, y));                    // set normal
            q.addPoint(x, y, z);                     // define a point

            z = f(x + dx, y);
            set_color_by_height(z, z_min, z_max);
            q.addNormal(n(x + dx, y));
            q.addPoint(x + dx, y, z);

            z = f(x + dx, y + dy);
            set_color_by_height(z, z_min, z_max);
            q.addNormal(n(x + dx, y + dy));
            q.addPoint(x + dx, y + dy, z);

            z = f(x, y + dy);
            set_color_by_height(z, z_min, z_max);
            q.addNormal(n(x, y + dy));
            q.addPoint(x, y + dy, z);

            q.draw();                              // render the quadrilateral
        }
    }
}
```

Listing 6 – Draw quadrilaterals.

5. 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.

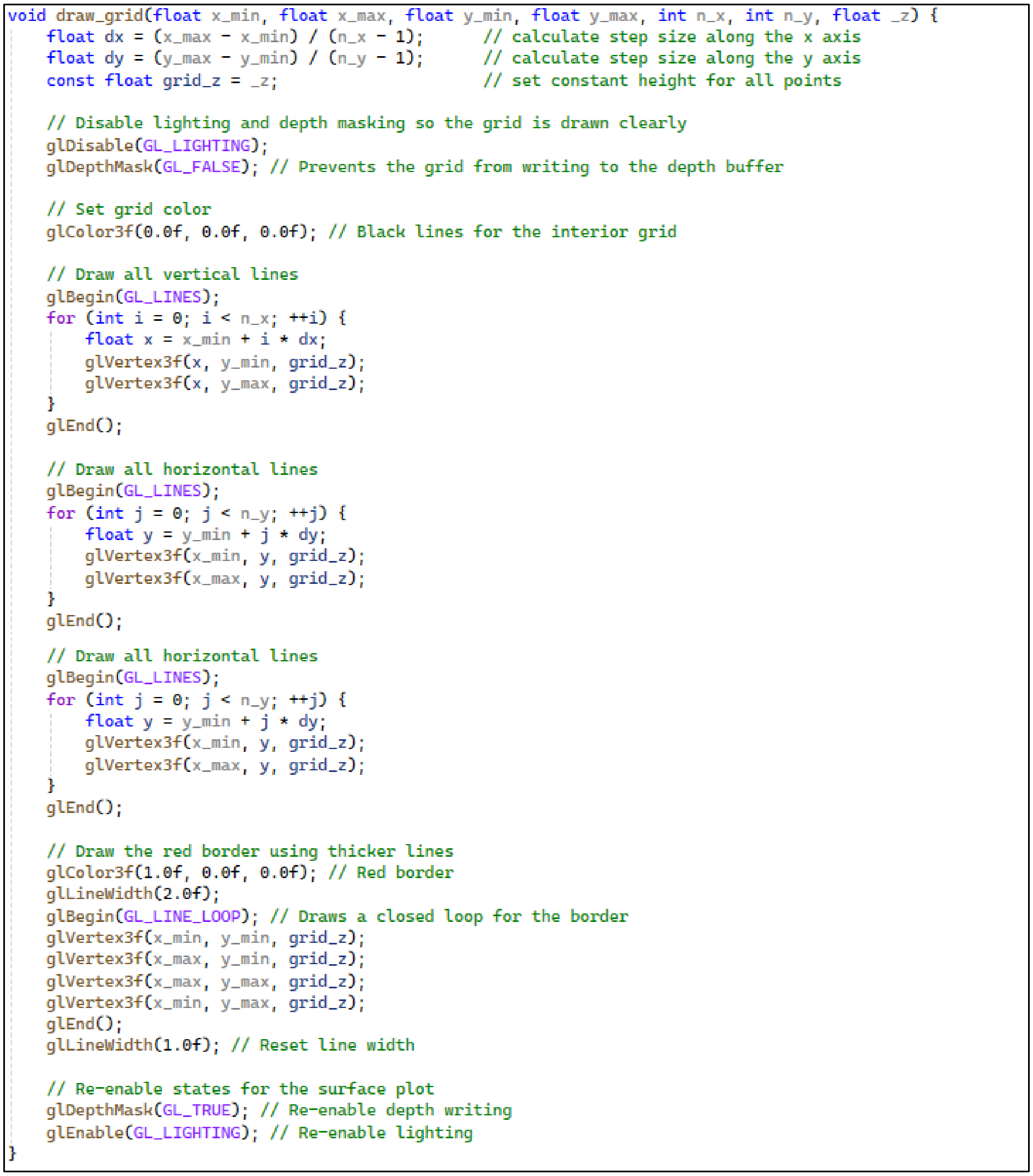


```
void viewing(int W, int H) {
    glMatrixMode(GL_MODELVIEW);  // controls the position of objects and camera
    glLoadIdentity();            // reset current matrix

    // based on the function, choose the eye (camera) position,
    // center of interest, and the up direction.
    if (function == "Sin") {
        gluLookAt(0.0f, 4.0f, 2.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f);
    } else if (function == "Wave") {
        gluLookAt(25, 15, 15, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f);
    } else if (function == "Parabola") {
        gluLookAt(5.0f, 12.0f, 10.0f, 0.0f, 0.0f, 4.0f, 0.0f, 0.0f, 1.0f);
    } else if (function == "Ripple") {
        gluLookAt(4.0f, 1.0f, 3.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f);
    } else if (function == "Fenc") {
        gluLookAt(3.0f, 3.0f, 4.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f);
    } else {
        gluLookAt(3.0f, 2.0f, 2.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f);
    }

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();            // reset current matrix

    // set field of view, aspect ratio, near clipping plane,
    // and the far clipping plane.
    float aspect = float(W) / H;
    gluPerspective(40.0f, aspect, 0.1f, 100.0f);
    // define the screen area
    glViewport(0, 0, W, H);
}
```

Listing 7 – Viewing method.

6. 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.



```
void draw_grid(float x_min, float x_max, float y_min, float y_max, int n_x, int n_y, float _z) {
    float dx = (x_max - x_min) / (n_x - 1);      // calculate step size along the x axis
    float dy = (y_max - y_min) / (n_y - 1);      // calculate step size along the y axis
    const float grid_z = _z;                     // set constant height for all points

    // Disable lighting and depth masking so the grid is drawn clearly
    glDisable(GL_LIGHTING);
    glDepthMask(GL_FALSE); // Prevents the grid from writing to the depth buffer

    // Set grid color
    glColor3f(0.0f, 0.0f, 0.0f); // Black lines for the interior grid

    // Draw all vertical lines
    glBegin(GL_LINES);
    for (int i = 0; i < n_x; ++i) {
        float x = x_min + i * dx;
        glVertex3f(x, y_min, grid_z);
        glVertex3f(x, y_max, grid_z);
    }
    glEnd();

    // Draw all horizontal lines
    glBegin(GL_LINES);
    for (int j = 0; j < n_y; ++j) {
        float y = y_min + j * dy;
        glVertex3f(x_min, y, grid_z);
        glVertex3f(x_max, y, grid_z);
    }
    glEnd();

    // Draw all horizontal lines
    glBegin(GL_LINES);
    for (int j = 0; j < n_y; ++j) {
        float y = y_min + j * dy;
        glVertex3f(x_min, y, grid_z);
        glVertex3f(x_max, y, grid_z);
    }
    glEnd();

    // Draw the red border using thicker lines
    glColor3f(1.0f, 0.0f, 0.0f); // Red border
    glLineWidth(2.0f);
    glBegin(GL_LINE_LOOP); // Draws a closed loop for the border
    glVertex3f(x_min, y_min, grid_z);
    glVertex3f(x_max, y_min, grid_z);
    glVertex3f(x_max, y_max, grid_z);
    glVertex3f(x_min, y_max, grid_z);
    glEnd();
    glLineWidth(1.0f); // Reset line width

    // Re-enable states for the surface plot
    glDepthMask(GL_TRUE); // Re-enable depth writing
    glEnable(GL_LIGHTING); // Re-enable lighting
}
```

Listing 8 – Grid drawing method.

7. 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.

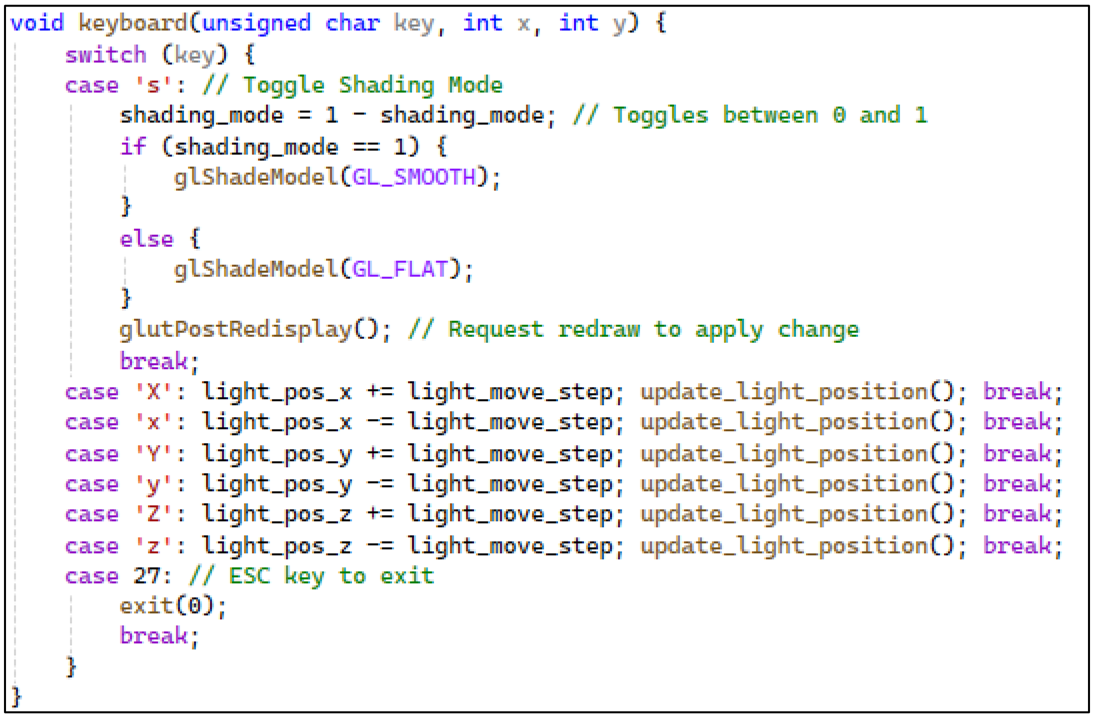
's': toggle shading mode.

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

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

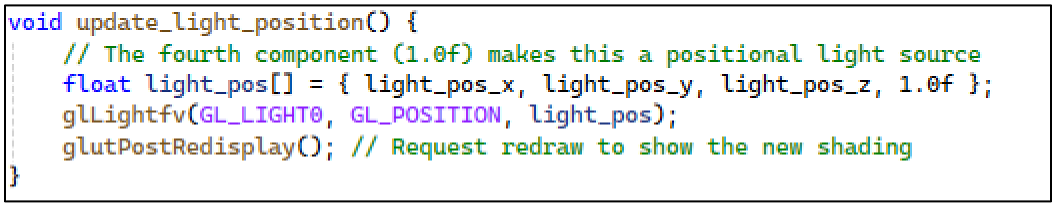
'z' | 'Z': move light along the Z axis.

'Esc': to end the execution.



```
void keyboard(unsigned char key, int x, int y) {
    switch (key) {
    case 's': // Toggle Shading Mode
        shading_mode = 1 - shading_mode; // Toggles between 0 and 1
        if (shading_mode == 1) {
            glShadeModel(GL_SMOOTH);
        }
        else {
            glShadeModel(GL_FLAT);
        }
        glutPostRedisplay(); // Request redraw to apply change
        break;
    case 'X': light_pos_x += light_move_step; update_light_position(); break;
    case 'x': light_pos_x -= light_move_step; update_light_position(); break;
    case 'Y': light_pos_y += light_move_step; update_light_position(); break;
    case 'y': light_pos_y -= light_move_step; update_light_position(); break;
    case 'Z': light_pos_z += light_move_step; update_light_position(); break;
    case 'z': light_pos_z -= light_move_step; update_light_position(); break;
    case 27: // ESC key to exit
        exit(0);
        break;
    }
}
```

Listing 9 – Keyboard press handler.



```
void update_light_position() {
    // The fourth component (1.0f) makes this a positional light source
    float light_pos[] = { light_pos_x, light_pos_y, light_pos_z, 1.0f };
    glLightfv(GL_LIGHT0, GL_POSITION, light_pos);
    glutPostRedisplay(); // Request redraw to show the new shading
}
```

Listing 10 – Update light position method.

8. 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.

```cpp
int main(int argc, char** argv) {
    std::cout << "Please choose a function: (Gaussian, Parabola, Ripple, Fenc, Sin, Wave): ";
    std::cin >> function;
    while (function != "Gaussian" and function != "Parabola" and
           function != "Ripple" and function != "Fenc" and
           function != "Sin" and function != "Wave") {
        std::cout << "Incorrect choice! Please try again: ";
        std::cin >> function;
    }

    // Initialize GLUT library
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);

    // Specify the initial window size
    glutInitWindowSize(800, 600);

    // Create a window with given title
    glutCreateWindow(function.c_str());

    // Enable depth testing for proper 3D rendering (hiding objects behind others)
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_SMOOTH);

    // Set a background color (dark blue)
    glClearColor(0.1f, 0.1f, 0.3f, 1.0f);

    // --- LIGHTING SETUP ---
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    // Define the light properties
    float light_pos[] = { light_pos_x, light_pos_y, light_pos_z, 1.0f }; // Positional light
    float white_light[] = { 1.0f, 1.0f, 1.0f, 1.0f };
    float ambient_light[] = { 0.2f, 0.2f, 0.2f, 1.0f };

    glLightfv(GL_LIGHT0, GL_POSITION, light_pos);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, white_light);
    glLightfv(GL_LIGHT0, GL_SPECULAR, white_light);
    glLightfv(GL_LIGHT0, GL_AMBIENT, ambient_light);
    // Enable color tracking, so glColor3f affects the material properties
    glEnable(GL_COLOR_MATERIAL);
    glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE);

    // --- GLEW INIT ---
    // Initialize GLEW *after* a GLUT window is created
    GLenum err = glewInit();
    if (GLEW_OK != err) {
        exit(1);
    }

    // Specify function to draw scene
    glutDisplayFunc(draw);

    // Specify function to set up viewing
    glutReshapeFunc(viewing);

    // Bind the new keyboard function
    glutKeyboardFunc(keyboard);

    // Start the event loop
    glutMainLoop();
    return 0;
}
```

Listing 11 – Main function (entry point).