Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

Lab Work 2

[Course Code: COMP 342]

Submitted by:
Jatin Madhikarmi
Roll No. 32

Submitted to:
Mr. Dhiraj Shrestha
Department of Computer Science and Engineering

1 Implement Digital Differential Analyzer Line drawing algorithm.

The Digital Differential Analyzer (DDA) is an incremental scan conversion algorithm used for line drawing. It performs calculations at each step by using the results from the previous step to determine the next pixel coordinates.

The DDA Algorithm

The algorithm follows these mathematical steps to draw a line between two points (x_1, y_1) and (x_2, y_2):

1. **Calculate Differences:** Compute the change in horizontal and vertical distance:

$$dx = x_2 - x_1, \quad dy = y_2 - y_1 \tag{1}$$

2. **Determine Steps:** Find the number of iterations required based on the larger magnitude:

$$steps = \max(|dx|, |dy|) \tag{2}$$

3. **Calculate Increments:** Determine the values to add to x and y at each iteration:

$$x_{inc} = \frac{dx}{steps}, \quad y_{inc} = \frac{dy}{steps} \tag{3}$$

4. **Iteration:** Initialize $(x, y) = (x_1, y_1)$. For each step from 0 to $steps$:
   - Plot the pixel at $(\text{round}(x), \text{round}(y))$.
   - Update $x = x + x_{inc}$ and $y = y + y_{inc}$.

PyOpenGL Implementation

Below is the Python implementation using the `PyOpenGL` and `GLUT` libraries.

```python
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *

def round_val(v):
    return int(v + 0.5)

def draw_dda():
    # Endpoints of the line
    x1, y1 = 50, 50
    x2, y2 = 450, 300

    dx = x2 - x1
    dy = y2 - y1

    # Find the number of steps
    steps = abs(dx) if abs(dx) > abs(dy) else abs(dy)

    # Calculate increments
    x_inc = dx / float(steps)
    y_inc = dy / float(steps)

    # Starting coordinates
    x, y = x1, y1

    glClear(GL_COLOR_BUFFER_BIT)
    glColor3f(0.0, 1.0, 1.0)  # Cyan color
    glBegin(GL_POINTS)

    for _ in range(int(steps) + 1):
        glVertex2i(round_val(x), round_val(y))
        x += x_inc
        y += y_inc
```

Figure 1: Screenshot of the DDA Algorithm Source Code

```
    for _ in range(int(steps) + 1):
        glVertex2i(round_val(x), round_val(y))
        x += x_inc
        y += y_inc

    glEnd()
    glFlush()

def main():
    glutInit()
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
    glutInitWindowSize(500, 500)
    glutInitWindowPosition(100, 100)
    glutCreateWindow(b"DDA Line Drawing Algorithm")

    # Initialize projection
    glClearColor(0.0, 0.0, 0.0, 1.0)
    gluOrtho2D(0, 500, 0, 500)

    glutDisplayFunc(draw_dda)
    glutMainLoop()

if __name__ == "__main__":
    main()
```

Figure 2: Screenshot of the DDA Algorithm Source Code

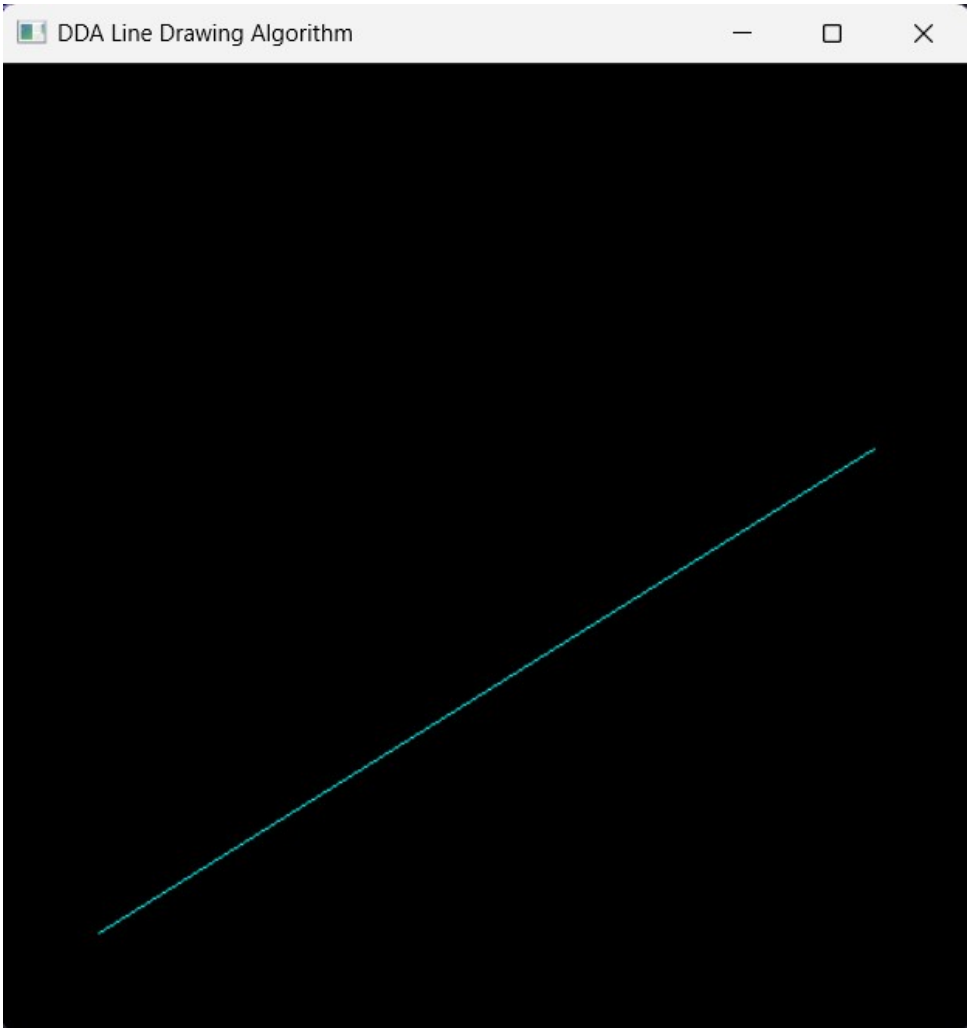

Figure 3: Output of the DDA Algorithm Source Code

Conclusion

The DDA algorithm is a simple and efficient way to generate lines. Its primary advantage is its simplicity, as it avoids complex multiplication by using incremental addition. However, because it relies on floating-point arithmetic and requires a rounding function at every step, it is generally considered slower than integer-based algorithms like Bresenham's.

2 Implement Bresenham Line Drawing Algorithm for both slopes ($|m| < 1$ and $|m| \geq 1$)

Bresenham's line algorithm is an efficient method for scan-converting lines on a grid. It uses only integer addition, subtraction, and bit-shifting, avoiding floating-point arithmetic. This makes it highly optimized for hardware and low-level graphics libraries.

The Algorithm

The algorithm differs based on the absolute value of the slope m.

Case 1: Shallow Slope ($|m| < 1$)

When the line is more horizontal than vertical, we increment x by 1 at each step and determine the y coordinate.

1. Calculate constants: $\Delta x = |x_1 - x_0|$ and $\Delta y = |y_1 - y_0|$.
2. Calculate the initial decision parameter:
$$P_0 = 2\Delta y - \Delta x$$
3. For each step k:
   - If $P_k < 0$: Next point is $(x_k + 1, y_k)$ and $P_{k+1} = P_k + 2\Delta y$.
   - If $P_k \geq 0$: Next point is $(x_k + 1, y_k + 1)$ and $P_{k+1} = P_k + 2\Delta y - 2\Delta x$.

Case 2: Steep Slope ($|m| \geq 1$)

When the line is more vertical, the roles of x and y are swapped. We increment y and calculate x.

1. Initial decision parameter:
$$P_0 = 2\Delta x - \Delta y$$
2. For each step k:
   - If $P_k < 0$: Next point is $(x_k, y_k + 1)$ and $P_{k+1} = P_k + 2\Delta x$.
   - If $P_k \geq 0$: Next point is $(x_k + 1, y_k + 1)$ and $P_{k+1} = P_k + 2\Delta x - 2\Delta y$.

PyOpenGL Implementation

The following source code provides a generalized implementation of the Bresenham algorithm using the `PyOpenGL` library.

```python
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *

def draw_bresenham_line(x0, y0, x1, y1):
    """Generalized Bresenham Line Drawing Algorithm"""
    dx = abs(x1 - x0)
    dy = abs(y1 - y0)
    sx = 1 if x0 < x1 else -1
    sy = 1 if y0 < y1 else -1
    err = dx - dy

    glBegin(GL_POINTS)
    while True:
        glVertex2i(x0, y0)
        if x0 == x1 and y0 == y1:
            break
        e2 = 2 * err
        if e2 > -dy:
            err -= dy
            x0 += sx
        if e2 < dx:
            err += dx
            y0 += sy
    glEnd()

def display():
    glClear(GL_COLOR_BUFFER_BIT)
    glColor3f(1.0, 1.0, 1.0)
    # Example Lines
    draw_bresenham_line(50, 50, 400, 200) # |m| < 1
    draw_bresenham_line(50, 50, 200, 400) # |m| > 1
    glFlush()
```

Figure 4: Screenshot of the Bresenham Algorithm Source Code

```
def init():
    glClearColor(0.0, 0.0, 0.0, 1.0)
    gluOrtho2D(0, 500, 0, 500)

def main():
    glutInit()
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
    glutInitWindowSize(500, 500)
    glutCreateWindow(b"Bresenham Line Algorithm")
    init()
    glutDisplayFunc(display)
    glutMainLoop()

if __name__ == "__main__":
    main()
```

Figure 5: Screenshot of the Bresenham Algorithm Source Code

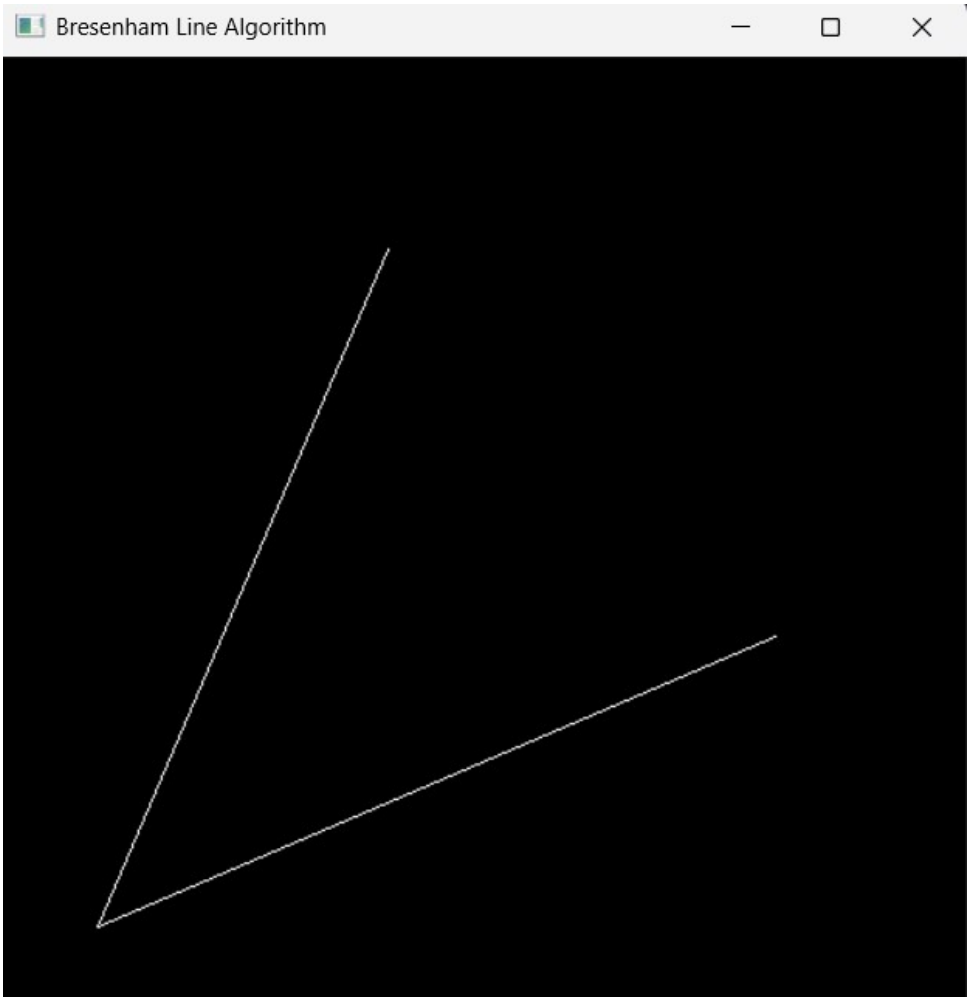

Figure 6: Output of the Bresenham Algorithm Source Code

Conclusion

In conclusion, the Bresenham Line Drawing algorithm remains a fundamental concept in computer graphics due to its efficiency and use of integer-only arithmetic. By alternating the driving axis based on the absolute value of the slope $|m|$, the algorithm ensures a continuous, high-fidelity line representation while avoiding the computational overhead of floating-point divisions or multiplications.

3 Write a Program to implement mid- point Circle Drawing Algorithm

The Mid-Point Circle algorithm determines the points needed for rasterizing a circle. It calculates the pixels for one octant (45°) and uses 8-way symmetry to complete the circle.

Step-by-Step Procedure

Given a center (x_c, y_c) and radius r:

1. **Initialization:** Start at point $(x, y) = (0, r)$.
2. **Initial Decision Parameter:** Calculate $P_0 = 1 - r$.
3. **Iteration:** While $x \leq y$, perform the following:
    - Plot the 8 symmetric points: $(x_c \pm x, y_c \pm y)$ and $(x_c \pm y, y_c \pm x)$.
    - If $P_k < 0$:
        - Next point is $(x + 1, y)$.
        - $P_{k+1} = P_k + 2x + 3$.
    - If $P_k \geq 0$:
        - Next point is $(x + 1, y - 1)$.
        - $P_{k+1} = P_k + 2x - 2y + 5$.
    - Increment $x = x + 1$.

PyOpenGL Implementation

The following Python code uses the `PyOpenGL` library to implement the algorithm.

```python
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *

def plot_points(xc, yc, x, y):
    glBegin(GL_POINTS)
    glVertex2i(xc + x, yc + y)
    glVertex2i(xc - x, yc + y)
    glVertex2i(xc + x, yc - y)
    glVertex2i(xc - x, yc - y)
    glVertex2i(xc + y, yc + x)
    glVertex2i(xc - y, yc + x)
    glVertex2i(xc + y, yc - x)
    glVertex2i(xc - y, yc - x)
    glEnd()

def midpoint_circle(xc, yc, r):
    x = 0
    y = r
    p = 1 - r

    plot_points(xc, yc, x, y)

    while x < y:
        x += 1
        if p < 0:
            p = p + 2 * x + 1
        else:
            y -= 1
            p = p + 2 * (x - y) + 1
        plot_points(xc, yc, x, y)
```

Figure 7: Screenshot of the Circle Drawing Algorithm Source Code

```python
def display():
    glClear(GL_COLOR_BUFFER_BIT)
    glColor3f(1.0, 1.0, 1.0)
    # Draw a circle at center (250, 250) with radius 100
    midpoint_circle(250, 250, 100)
    glFlush()

def main():
    glutInit()
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
    glutInitWindowSize(500, 500)
    glutCreateWindow(b"Midpoint Circle Algorithm")
    gluOrtho2D(0, 500, 0, 500)
    glutDisplayFunc(display)
    glutMainLoop()

if __name__ == "__main__":
    main()
```

Figure 8: Screenshot of the Circle Drawing Source Code

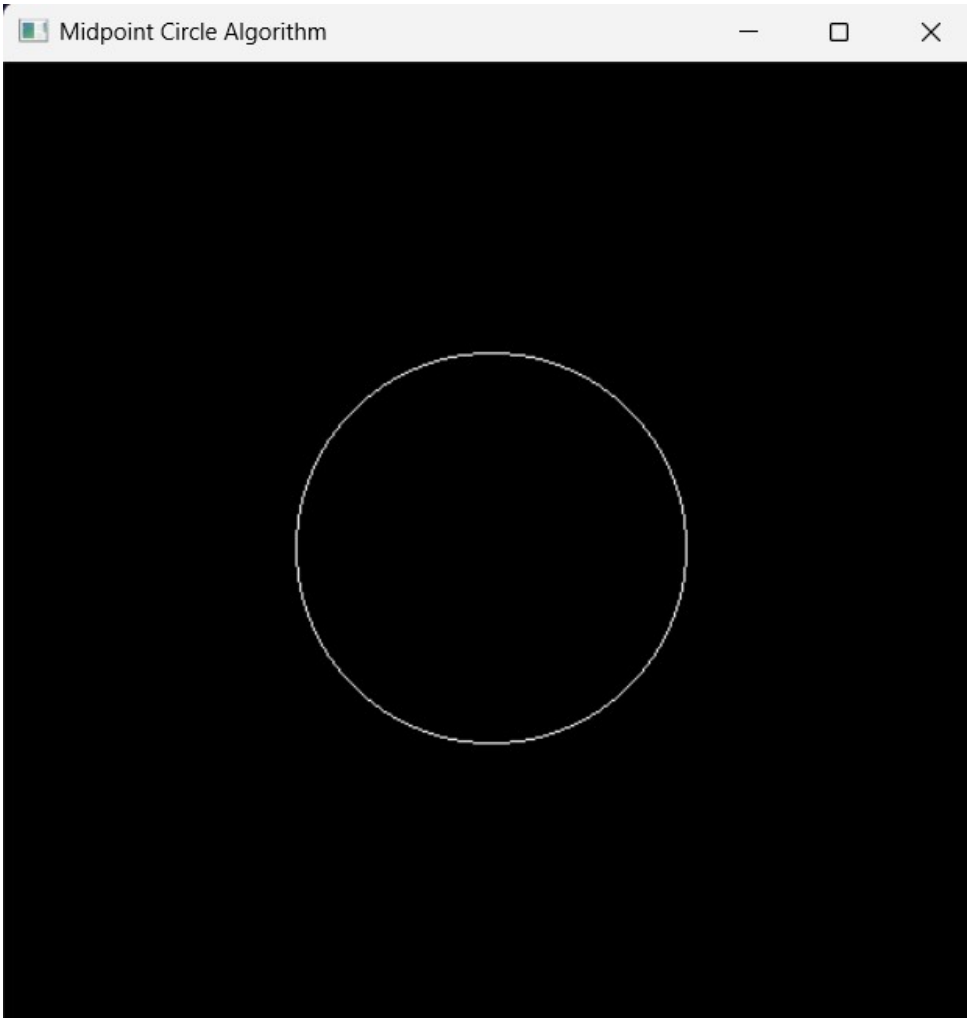

Figure 9: Output of the Circle Drawing Source Code

Conclusion

The Mid-point Circle Drawing algorithm is an efficient approach to rasterization because it exclusively uses **integer addition and subtraction**. By evaluating the midpoint between two potential pixels, it eliminates the need for complex trigonometric functions or square roots. Its use of 8-way symmetry ensures that only $1/8^{th}$ of the circle's points need to be calculated, making it ideal for performance-constrained environments.

4 Implement the Line Function (DDA/BLA) for generating a line graph of a given set of data

Line generation algorithms are fundamental in computer graphics for rendering straight lines on raster displays. When plotting a graph of a given set of data points, straight-line segments are used to join consecutive points. Two widely used algorithms for this purpose are the Digital Differential Analyzer (DDA) and the Bresenham Line Algorithm (BLA). This document presents the algorithmic steps and implementation of both methods using PyOpenGL.

Algorithm for Line Generation

Digital Differential Analyzer (DDA) Algorithm

The DDA algorithm generates intermediate points between two given endpoints using incremental floating-point calculations.

Algorithm Steps

1. Input two endpoints (x_1, y_1) and (x_2, y_2).
2. Compute $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$.
3. Determine the number of steps:
$$steps = \max(|\Delta x|, |\Delta y|)$$
4. Compute increments:
$$x_{inc} = \frac{\Delta x}{steps}, \quad y_{inc} = \frac{\Delta y}{steps}$$
5. Initialize $(x, y) = (x_1, y_1)$.
6. For each step:
   - Plot the point $(round(x), round(y))$
   - Update $x = x + x_{inc}$ and $y = y + y_{inc}$

Bresenham Line Algorithm (BLA)

Bresenham's algorithm uses integer arithmetic and decision parameters, making it faster and more efficient.

Algorithm Steps (for $|m| < 1$)

1. Input endpoints (x_1, y_1) and (x_2, y_2).
2. Compute $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$.
3. Initialize decision parameter:
$$p = 2\Delta y - \Delta x$$
4. Set $(x, y) = (x_1, y_1)$.
5. For each x until x_2:
   - Plot (x, y)
   - If $p < 0$, update:
   $$p = p + 2\Delta y$$
   - Else update:
   $$y = y + 1, \quad p = p + 2(\Delta y - \Delta x)$$

PyOpenGL Source Code for Line Graph

The following code plots a line graph by joining a set of data points using the DDA algorithm.

```python
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *

data_points = [(50, 60), (100, 120), (150, 90), (200, 160), (250, 130)]

def dda_line(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    steps = int(max(abs(dx), abs(dy)))
    x_inc = dx / steps
    y_inc = dy / steps

    x = x1
    y = y1

    glBegin(GL_POINTS)
    for _ in range(steps + 1):
        glVertex2i(int(round(x)), int(round(y)))
        x += x_inc
        y += y_inc
    glEnd()

def display():
    glClear(GL_COLOR_BUFFER_BIT)
    glColor3f(1.0, 1.0, 1.0)

    for i in range(len(data_points) - 1):
        x1, y1 = data_points[i]
        x2, y2 = data_points[i + 1]
        dda_line(x1, y1, x2, y2)

    glFlush()
```

Figure 10: Screenshot of the Line Drawing Algorithm Source Code

```
def init():
    glClearColor(0.0, 0.0, 0.0, 1.0)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    gluOrtho2D(0, 300, 0, 200)

glutInit()
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
glutInitWindowSize(600, 400)
glutCreateWindow(b"Line Graph using DDA Algorithm")
init()
glutDisplayFunc(display)
glutMainLoop()
```

Figure 11: Screenshot of the Line Drawing Algorithm Source Code

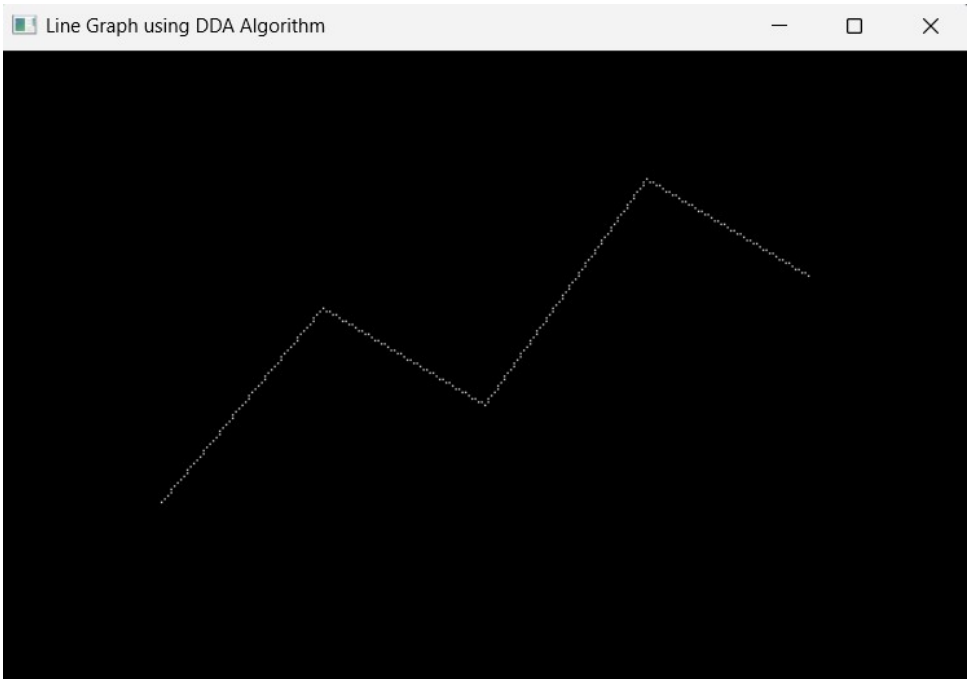

Figure 12: Output of the Line Drawing Algorithm Source Code

Conclusion

In this work, the Digital Differential Analyzer and Bresenham Line Algorithm were studied and applied to generate a line graph from a given set of data points. While the DDA algorithm is simple and easy to implement, it relies on floating-point arithmetic. Bresenham's algorithm, on the other hand, is more efficient due to its use of integer calculations. Both algorithms are fundamental to computer graphics and are widely used in rendering graphs and geometric primitives.

5 Implement the Pie chart

A pie chart is a circular statistical graphic that represents data in proportional slices. Each slice corresponds to a data value and its angular size is proportional to the value relative to the total sum of all values. In computer graphics, a pie chart can be generated by dividing a circle into sectors and rendering each sector using trigonometric functions.

Algorithm for Pie Chart Generation

The following algorithm describes the steps required to implement a pie chart.

Algorithm

1. Read the data values to be represented in the pie chart.
2. Compute the total sum of all data values.
3. For each data value, calculate its percentage contribution:
$$\text{Percentage} = \frac{\text{Value}}{\text{Total}} \times 100$$
4. Convert the percentage into an angle:
$$\text{Angle} = \frac{\text{Value}}{\text{Total}} \times 360^\circ$$
5. Initialize the starting angle to 0°.
6. For each data value:
   (a) Draw a circular sector from the starting angle to the ending angle.
   (b) Use triangle fan method with the center of the circle.
   (c) Assign a distinct color to the sector.
   (d) Update the starting angle to the ending angle.
7. Repeat until all data values are plotted.

Source Code in PyOpenGL

The following Python program uses `PyOpenGL` and `GLUT` to draw a pie chart.

```python
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *
import math

# Data for pie chart
values = [40, 25, 20, 15]
colors = [
    (1.0, 0.0, 0.0),
    (0.0, 1.0, 0.0),
    (0.0, 0.0, 1.0),
    (1.0, 1.0, 0.0)
]

def draw_pie_chart():
    glClear(GL_COLOR_BUFFER_BIT)
    glLoadIdentity()

    total = sum(values)
    start_angle = 0.0
    radius = 0.6

    for i in range(len(values)):
        angle = (values[i] / total) * 360.0
        glColor3f(*colors[i])

        glBegin(GL_TRIANGLE_FAN)
        glVertex2f(0.0, 0.0)  # center of circle

        theta = start_angle
        while theta <= start_angle + angle:
            x = radius * math.cos(math.radians(theta))
            y = radius * math.sin(math.radians(theta))
            glVertex2f(x, y)
```

Figure 13: Screenshot of the Pie Chart Algorithm Source Code

```
        glVertex2f(0.0, 0.0)  # center of circle

        theta = start_angle
        while theta <= start_angle + angle:
            x = radius * math.cos(math.radians(theta))
            y = radius * math.sin(math.radians(theta))
            glVertex2f(x, y)
            theta += 1.0

        glEnd()
        start_angle += angle

    glFlush()

def init():
    glClearColor(1.0, 1.0, 1.0, 1.0)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    gluOrtho2D(-1, 1, -1, 1)

def main():
    glutInit()
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
    glutInitWindowSize(600, 600)
    glutCreateWindow(b"Pie Chart using PyOpenGL")
    init()
    glutDisplayFunc(draw_pie_chart)
    glutMainLoop()

main()
```

Figure 14: Screenshot of the Pie Chart Algorithm Source Code

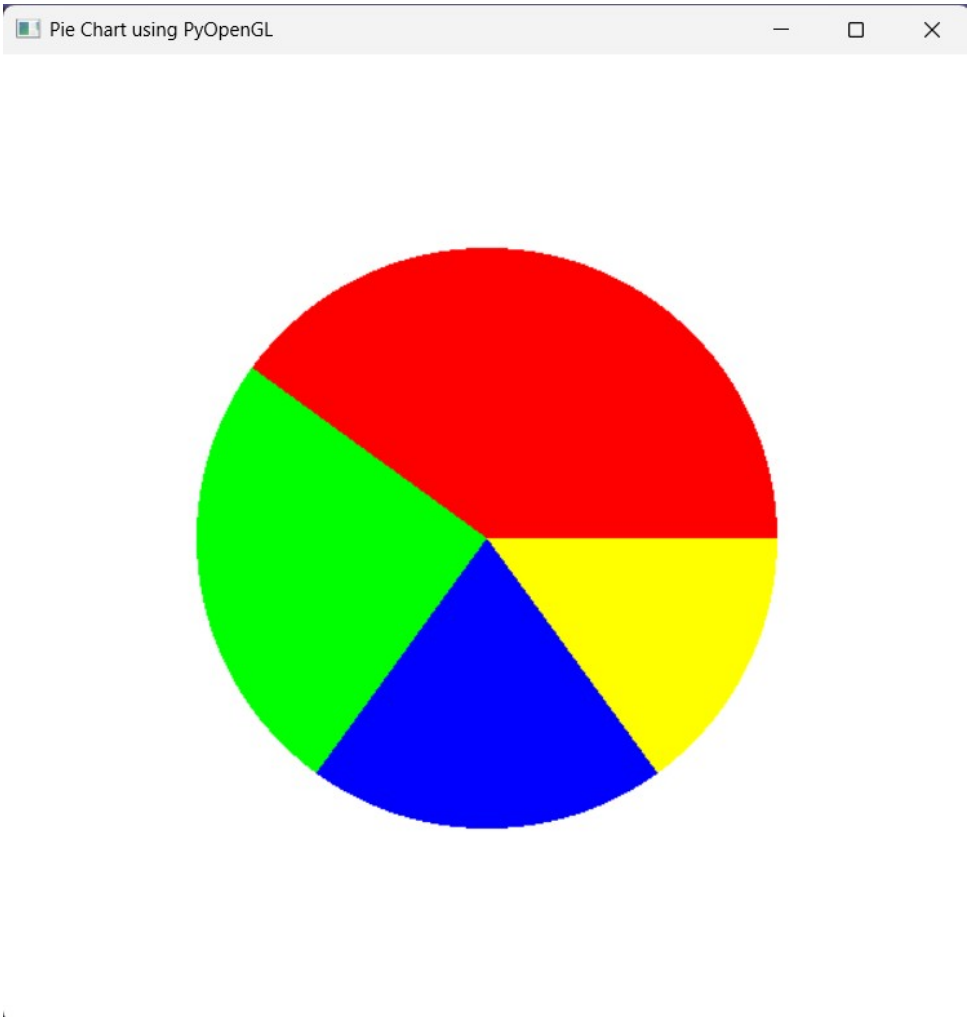

Figure 15: Output of the Pie Chart Algorithm Source Code

Conclusion

In this experiment, a pie chart was successfully implemented using PyOpenGL by dividing a circle into angular sectors proportional to the given data values. The use of trigonometric functions allows accurate plotting of each slice, while the incremental angle approach ensures smooth rendering. This method demonstrates how fundamental computer graphics concepts such as coordinate geometry and polygon filling are applied to real-world data visualization.

6 Implement midpoint Ellipse drawing Algorithm

The standard equation of an ellipse centered at the origin is:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$

where:

- a = semi-major axis (along x-axis)
- b = semi-minor axis (along y-axis)

Algorithm (Midpoint Ellipse Drawing Algorithm)

Step 1: Input

- Center of ellipse (x_c, y_c)
- Semi-major axis a
- Semi-minor axis b

Step 2: Initialize

- Start point: $(x, y) = (0, b)$
- Compute initial decision parameter for Region 1:

$$p_1 = b^2 - a^2 b + \frac{1}{4}a^2$$

Step 3: Region 1 ($|slope| < 1$)

- While $2b^2x \leq 2a^2y$:
  - Plot points using symmetry
  - If $p_1 < 0$:

$$x = x + 1$$
$$p_1 = p_1 + 2b^2x + b^2$$

  - Else:

$$x = x + 1, \quad y = y - 1$$
$$p_1 = p_1 + 2b^2x - 2a^2y + b^2$$

Step 4: Region 2 ($|slope| > 1$)

- Compute initial decision parameter:

$$p_2 = b^2(x + 0.5)^2 + a^2(y - 1)^2 - a^2b^2$$

- While $y \geq 0$:
  - Plot points using symmetry
  - If $p_2 > 0$:

$$y = y - 1$$
$$p_2 = p_2 - 2a^2y + a^2$$

  - Else:

$$x = x + 1, \quad y = y - 1$$
$$p_2 = p_2 + 2b^2x - 2a^2y + a^2$$

Step 5: Use Symmetry

Plot the four symmetric points:

$$(x_c \pm x,\ y_c \pm y)$$

Source Code (PyOpenGL Implementation)

```python
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *

xc, yc = 0, 0
a, b = 120, 80

def plot(x, y):
    glVertex2i(xc + x, yc + y)
    glVertex2i(xc - x, yc + y)
    glVertex2i(xc + x, yc - y)
    glVertex2i(xc - x, yc - y)

def midpoint_ellipse():
    x = 0
    y = b

    p1 = b*b - a*a*b + 0.25*a*a

    glBegin(GL_POINTS)
    while (2*b*b*x) <= (2*a*a*y):
        plot(x, y)
        if p1 < 0:
            x += 1
            p1 += 2*b*b*x + b*b
        else:
            x += 1
            y -= 1
            p1 += 2*b*b*x - 2*a*a*y + b*b

    p2 = b*b*(x + 0.5)**2 + a*a*(y - 1)**2 - a*a*b*b

    while y >= 0:
        plot(x, y)
```

Figure 16: Screenshot of the Ellipse Drawing Algorithm Source Code

```
        plot(x, y)
        if p2 > 0:
            y -= 1
            p2 += a*a - 2*a*a*y
        else:
            x += 1
            y -= 1
            p2 += 2*b*b*x - 2*a*a*y + a*a
    glEnd()
    glFlush()

def display():
    glClear(GL_COLOR_BUFFER_BIT)
    glColor3f(1.0, 1.0, 1.0)
    midpoint_ellipse()

def init():
    glClearColor(0, 0, 0, 1)
    gluOrtho2D(-200, 200, -200, 200)

glutInit()
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
glutInitWindowSize(600, 600)
glutCreateWindow(b"Midpoint Ellipse Drawing Algorithm")
init()
glutDisplayFunc(display)
glutMainLoop()
```

Figure 17: Screenshot of the Ellipse Drawing Algorithm Source Code

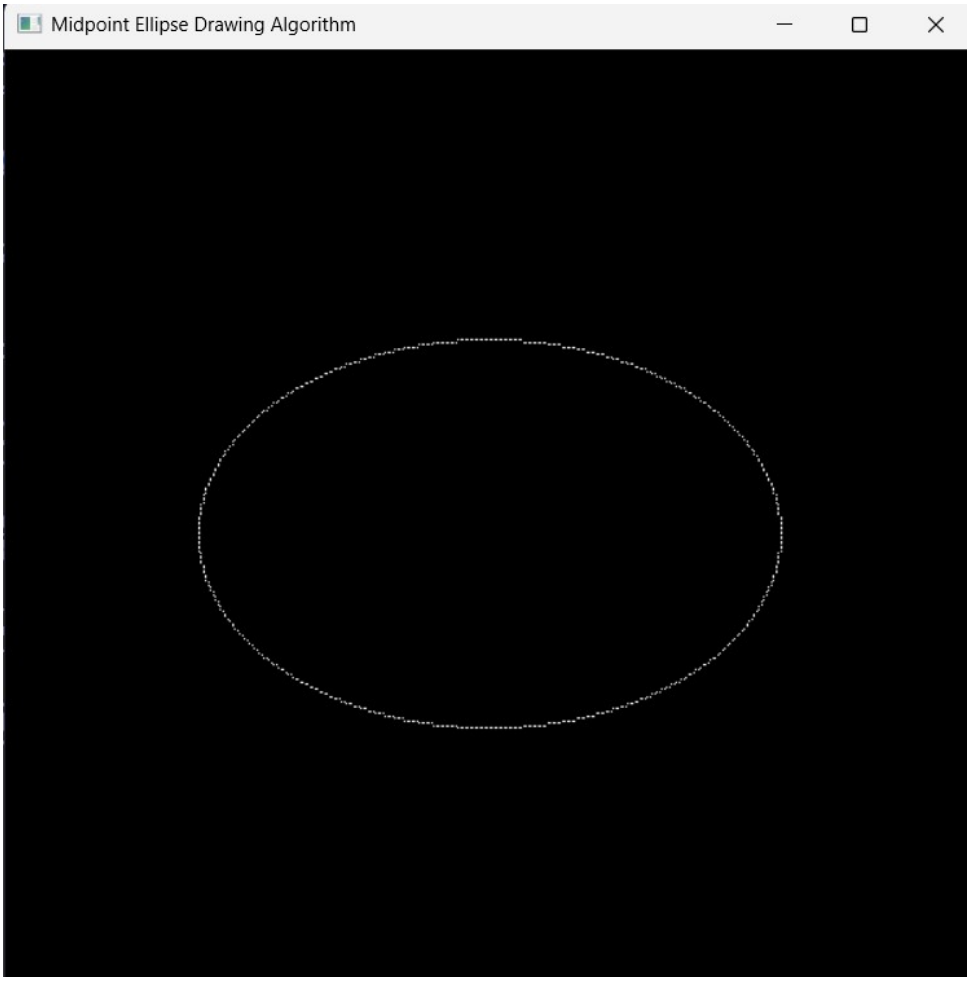

Figure 18: Output of the Ellipse Drawing Algorithm Source Code

Conclusion

The Midpoint Ellipse Drawing Algorithm is an efficient and accurate method for rasterizing an ellipse. By dividing the ellipse into two regions based on slope and using incremental decision parameters, the algorithm avoids costly floating-point calculations. Its use of symmetry further reduces computation, making it highly suitable for real-time graphics applications and foundational computer graphics education.