

Kathmandu University
Department of Computer Science and Engineering
Dhulikhel, Kavre



“Lab Report - II”

[Code No: COMP 342]
Subject: Computer Graphics

Submitted by:

Aryan Koju (31)
CS-2023 (III / I)

Submitted to:

Mr. Dhiraj Shrestha
Department of Computer Science and Engineering
Kathmandu University

Submission: December 26, 2025

1. Implement Digital Differential Analyzer Line drawing algorithm.

The Digital Differential Analyzer (DDA) algorithm draws a straight line by calculating incremental steps in x and y directions. It uses floating-point arithmetic and rounds the calculated points to the nearest pixel.

Algorithm

1. Start with the two endpoints (x_0, y_0) and (x_1, y_1) .

2. Compute differences:

$$dx = x_1 - x_0$$

$$dy = y_1 - y_0$$

3. Determine number of steps:

$$\text{step size} = \max(|dx|, |dy|)$$

4. Compute the increments:

$$x_inc = dx / \text{step size}$$

$$y_inc = dy / \text{step size}$$

5. Initialize:

$$x = x_0, y = y_0$$

6. Plot initial point:

$$\text{plot}(\text{round}(x), \text{round}(y))$$

7. Repeat for steps times:

- $x = x + x_inc$

- $y = y + y_inc$

- Plot $\text{round}(x), \text{round}(y)$

Source Code

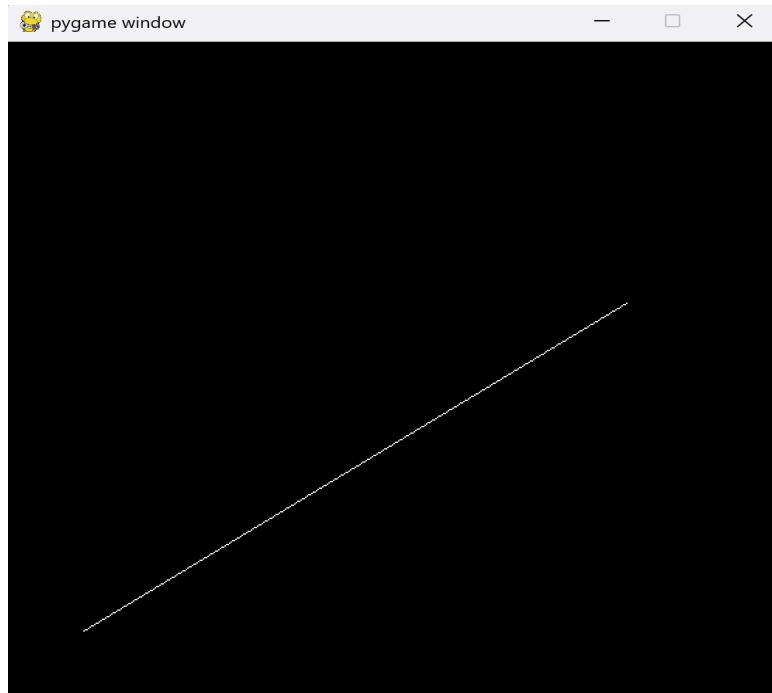
```
def dda(x0, y0, x1, y1):
    dx = x1 - x0
    dy = y1 - y0
    stepsize = int(max(abs(dx), abs(dy)))

    x_inc = dx / stepsize
    y_inc = dy / stepsize

    x, y = x0, y0
    glBegin(GL_POINTS)
    for _ in range(ssize + 1):
        glVertex2f(round(x), round(y))
        x += x_inc
        y += y_inc
    glEnd()
```

Fig: DDA Line Drawing Algorithm

Output



2. Implement Bresenham Line Drawing algorithm for both slopes($|m|<1$ and $|m|>1$).

Bresenham's algorithm is an efficient line drawing technique that uses only integer calculations. It selects the nearest pixel based on a decision parameter, making it faster than DDA. The algorithm works for all slopes.

Algorithm

1. Start with two endpoints:

$$(x_0, y_0) \text{ and } (x_1, y_1).$$

2. Compute the differences:

$$dx = |x_1 - x_0|$$

$$dy = |y_1 - y_0|$$

3. Determine the direction of movement:

- If $x_1 \geq x_0$ then $sx = 1$, else $sx = -1$
- If $y_1 \geq y_0$ then $sy = 1$ else $sy = -1$

4. Initialize the starting point:

$$x = x_0, y = y_0$$

Case 1: $|m| < 1$ (i.e., $dx > dy$)

5. Initialize the decision parameter:

$$P = 2dy - dx$$

6. For each step from 0 to dx :

- Plot the point (x, y)

7. If $p \geq 0$:

- $y = y + sy$
- $p = p + 2dy - 2dx$

8. Else:

- $p = p + 2dy$

9. Update:

- $x = x + sx$

Case 2: $|m| \geq 1$ (i.e., $dy \geq dx$)

10. Initialize the decision parameter:

$$p = 2dx - dy$$

11. For each step from 0 to dy :

- Plot the point (x,y)

12. If $p \geq 0$:

- $x = x + sx$
- $p = p + 2dx - 2dy$

13. Else:

- $p = p + 2dx$

14. Update:

- $y = y + sy$

Source Code

```
# ----- Bresenham Algorithm -----
def bresenham_line(x0, y0, x1, y1):
    dx = abs(x1 - x0)
    dy = abs(y1 - y0)

    # Direction of movement
    sx = 1 if x1 >= x0 else -1
    sy = 1 if y1 >= y0 else -1

    x, y = x0, y0

    glBegin(GL_POINTS)

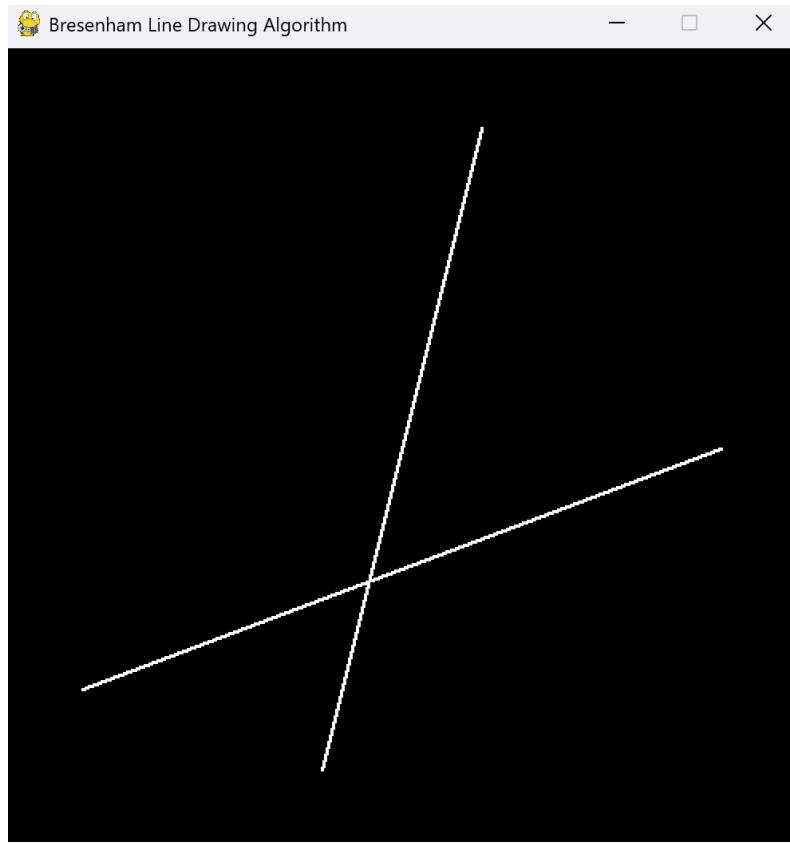
    # ----- Case 1: |m| < 1 (dx > dy) -----
    if dx > dy:
        p = 2 * dy - dx
        for _ in range(dx + 1):
            glVertex2i(x, y)
            x += sx
            if p >= 0:
                y += sy
                p += 2 * (dy - dx)
            else:
                p += 2 * dy

    # ----- Case 2: |m| >= 1 (dy >= dx) -----
    else:
        p = 2 * dx - dy
        for _ in range(dy + 1):
            glVertex2i(x, y)
            y += sy
            if p >= 0:
                x += sx
                p += 2 * (dx - dy)
            else:
                p += 2 * dx

    glEnd()
```

Fig: Bresenham Line Drawing Algorithm

Output



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

The midpoint circle drawing algorithm is an algorithm used to determine the points needed for rasterizing a circle. We use the midpoint algorithm to calculate all the perimeter points of the circle in the first octant and then print them along with their mirror points in the other octants. This will work because a circle is symmetric about its centre.

Algorithm

1. Input center (xc , yc) and radius r .
2. Initialize $x = 0$, $y = r$.
3. Initialize decision parameter: $p = 1 - r$ (integers) and $p = 5/4 - r$ (float)
4. Plot initial points using eight symmetry:
 $(xc \pm x, yc \pm y)$ and $(xc \pm y, yc \pm x)$

5. Repeat the following steps while $x \leq y$:

6. Plot the eight symmetric points:

- $(xc + x, yc + y)$
- $(xc - x, yc + y)$
- $(xc + x, yc - y)$
- $(xc - x, yc - y)$
- $(xc + y, yc + x)$
- $(xc - y, yc + x)$
- $(xc + y, yc - x)$
- $(xc - y, yc - x)$

7. If the decision parameter $d < 0$:

$$d = d + 2x + 3$$

8. Else:

$$d = d + 2(x-y) + 5, y = y - 1$$

9. Increment x:

$$x = x + 1$$

10. Continue until all points of the circle are generated.

Source Code

```
# ----- Midpoint Circle Algorithm -----
def midpoint_circle(cx, cy, r):
    x = 0
    y = r
    d = 1 - r

    glBegin(GL_POINTS)

    while x <= y:
        # Plot eight symmetric points
        glVertex2i(cx + x, cy + y)
        glVertex2i(cx - x, cy + y)
        glVertex2i(cx + x, cy - y)
        glVertex2i(cx - x, cy - y)

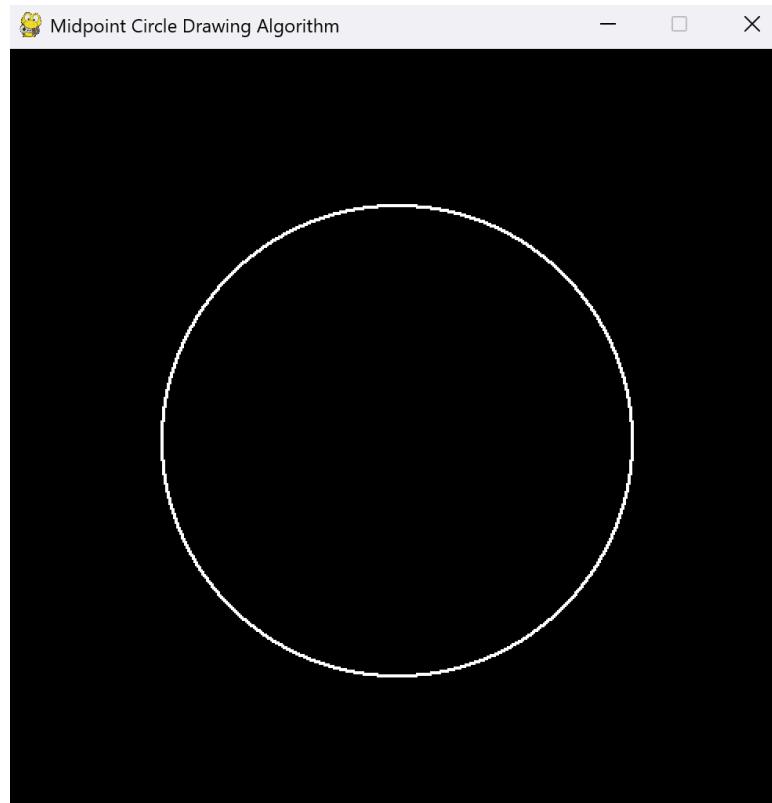
        glVertex2i(cx + y, cy + x)
        glVertex2i(cx - y, cy + x)
        glVertex2i(cx + y, cy - x)
        glVertex2i(cx - y, cy - x)

        # Update decision parameter
        if d < 0:
            d = d + 2 * x + 3
        else:
            d = d + 2 * (x - y) + 5
            y -= 1

        x += 1

    glEnd()
```

Output



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

Algorithm

1. Define the dataset size
 - Let n be the number of points to plot on the graph.
2. Calculate horizontal positions
 - Set left and right margins for the plotting area.
 - Compute the horizontal gap between consecutive points:

$$\text{Gap} = \text{Window width} - 2 \times \text{margin} / n-1$$

3. Create dataset values

For each index $i = 0$ to $n-1$:

- a. Assign $x_i = \text{margin} + i \times \text{gap}$

- b. Generate a random vertical value y_i within a given range $[y_{\min}, y_{\max}]$
 - c. Store the points as pairs:

$$P = \{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$$
4. Draw lines between points using DDA
- For each consecutive pair $(x_i, y_i) \rightarrow (x_{i+1}, y_{i+1})$:
 - i. Compute differences:

$$dx = x_{i+1} - x_i, dy = y_{i+1} - y_i$$
 - ii. Determine the number of steps:

$$\text{Stepsize} = \max(|dx|, |dy|)$$
 - iii. Compute increments per step:

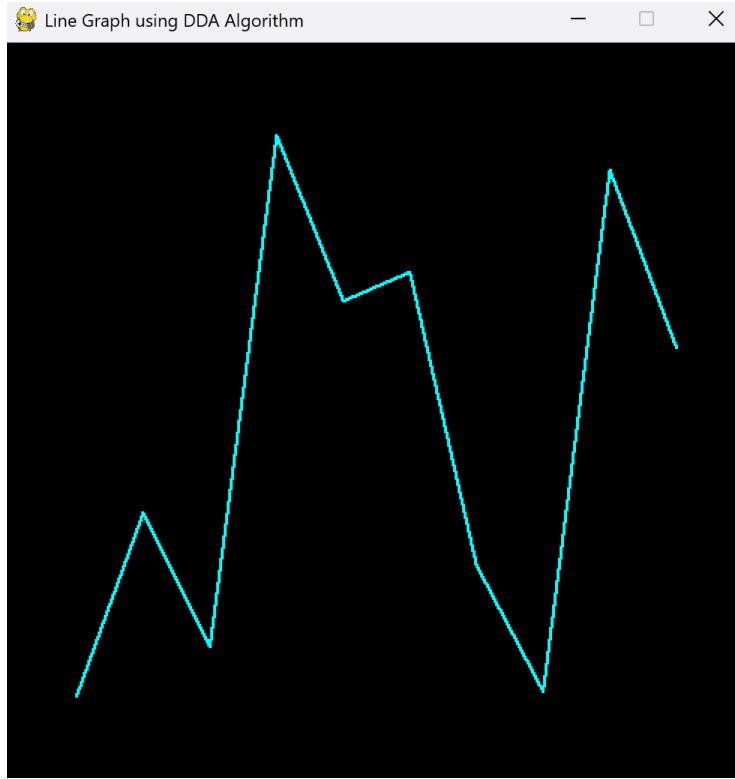
$$x\text{-inc} = dx/\text{stepsize}, y\text{-inc} = dy/\text{stepsize}$$
 - iv. Start at $(x, y) = (x_i, y_i)$ and for each step:
 - 1. Plot the point $(\text{round}(x), \text{round}(y))$
 - 2. Update: $x = x + x\text{-inc}, y = y + y\text{-inc}$
5. Collect the plotted points from all segments to form the complete line graph.

Source Code

```
# ----- Generate Line Graph -----
def generate_line_graph(n, x_margin, y_min, y_max, width):
    points = []
    gap = (width - 2 * x_margin) / (n - 1)
    for i in range(n):
        x = x_margin + i * gap
        y = random.randint(y_min, y_max)
        points.append((x, y))
    return points
```

(Here, `random.randint()` creates random data points for demonstrating a line graph.)

Output



5. Implement the Pie chart.

Algorithm

1. Initialize the parameters:

- Number of slices $n = 6$
- Center of the pie chart: (Cx, Cy)
- Radius of the pie chart: r
- Steps per slice (for smooth boundaries): $s = 100$

2. Generate data values for slices

- For each slice $i = 1$ to n :
 - Generate a random integer value d_i within a range (e.g., 5–50)
- Compute the total sum of all slice values:
$$T = \sum_{i=1}^n d_i$$
- Convert each slice value to an angle proportional to the total:
$$\text{angle}_i = 360 \times d_i / T$$

3. Compute start and end angles for each slice

- Initialize start_angle = 0
- For each slice i:
 - i. end_angle = start_angle + angle_i
 - ii. Store (start_angle, end_angle) for the slice
 - iii. Update: start_angle = end_angle

4. Generate points for each slice

For each slice:

- a. Initialize a list of points for the slice, starting with the center (C_x, C_y)
- b. For each step k = 0 to s:
 - i. Compute the interpolated angle:

$$\Theta = \text{start_angle} + k \times (\text{end_angle} - \text{start_angle})/s$$

- ii. Compute the boundary point on the circumference:
 $x = C_x + r \cdot \cos(\theta), y = C_y + r \cdot \sin(\theta)$
- iii. Add the point (x, y) to the slice list

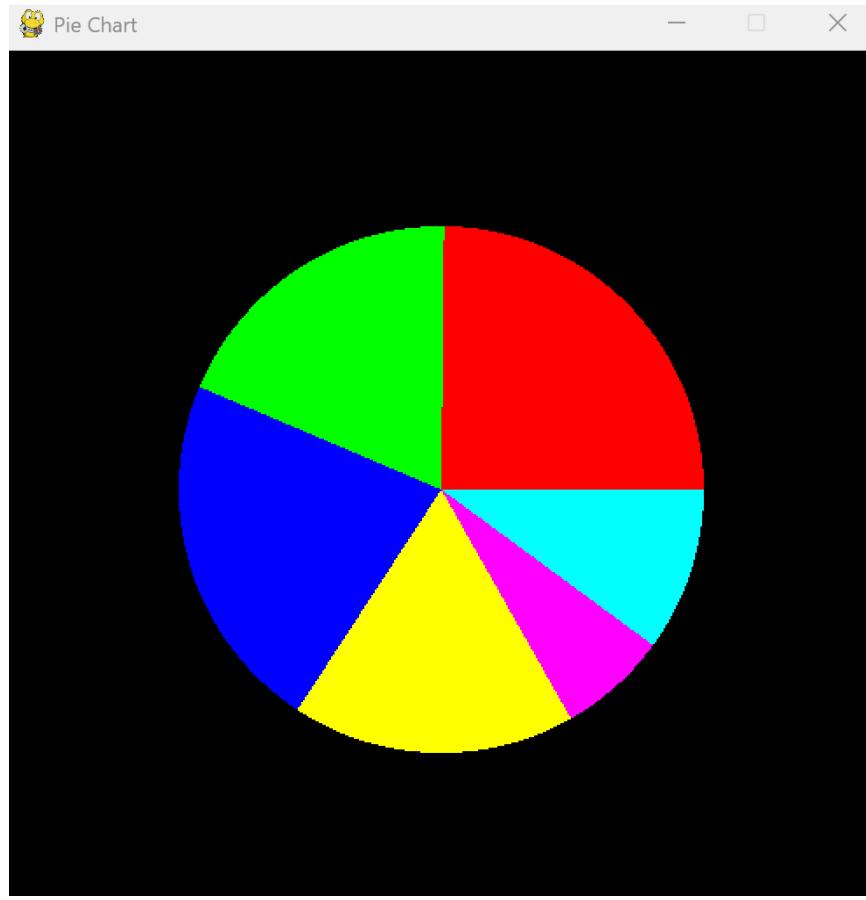
5. Draw slices

- Connect all points in the slice sequentially to form a filled wedge of the pie chart
- Repeat for all slices

Source Code

```
# ----- Pie Chart Drawing -----
def draw_pie_chart(cx, cy, r, angles, slice_colors, steps=100):
    # Draw each slice using precomputed angles and fixed colors
    for i, (start, end) in enumerate(angles):
        glBegin(GL_TRIANGLE_FAN)
        glColor3f(*slice_colors[i]) # Fixed color for this slice
        glVertex2f(cx, cy)         # Center point
        for k in range(steps + 1):
            theta = math.radians(start + k * (end - start) / steps)
            x = cx + r * math.cos(theta)
            y = cy + r * math.sin(theta)
            glVertex2f(x, y)
        glEnd()
```

Output



6. Implement a midpoint Ellipse drawing Algorithm.

The midpoint ellipse algorithm draws an ellipse using symmetry and region-based decision parameters. It efficiently determines pixel positions without floating-point operations.

Algorithm

1. Input parameters : center (xc , yc) and ellipse radii rx , ry .
2. Initialize starting point
 - Set $x = 0$, $y = ry$
3. Initialize decision parameter for Region 1
$$p_1 = ry^2 - rx^2 \cdot ry + 1/4 * rx^2$$
4. Region 1 processing (slope < 1)

- While $2ry^2x < 2rx^2y$:
 - Plot the four symmetric points
 - If $p1 < 0$:
 - $x = x + 1$
 - $p1 = p1 + 2ry^2x + ry^2$
 - Else:
 - $x = x + 1, y = y - 1$
 - $p1 = p1 + 2ry^2x - 2rx^2y + ry^2$

5. Initialize decision parameter for Region 2

$$P_2 = ry^2(x+0.5)^2 + rx^2(y-1)^2 - rx^2ry^2$$

6. Region 2 processing (slope ≥ 1)

- While $y \geq 0$:
 - Plot the four symmetric points
 - If $p2 > 0$:
 - $y = y - 1$
 - $p2 = p2 - 2rx^2y + rx^2$
 - Else:
 - $x = x + 1, y = y - 1$
 - $p2 = p2 + 2ry^2x - 2rx^2y + rx^2$

7. Apply symmetry and translation

- For every calculated (x, y) , plot points at:
 - $(\pm x + xc, \pm y + yc)$

Source Code

```
# ----- Midpoint Ellipse Algorithm -----
def midpoint_ellipse(xc, yc, rx, ry):
    x = 0
    y = ry

    rx2 = rx * rx
    ry2 = ry * ry

    # Region 1 decision parameter
    p1 = ry2 - rx2 * ry + 0.25 * rx2

    glBegin(GL_POINTS)

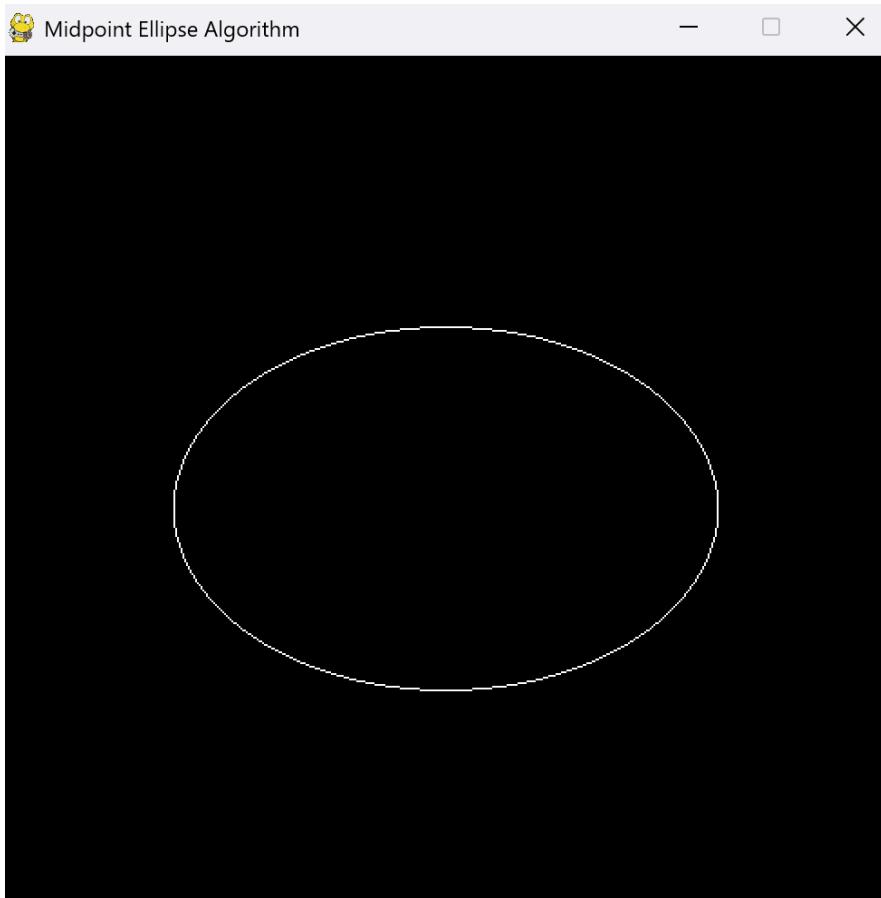
    # ----- Region 1 -----
    while 2 * ry2 * x < 2 * rx2 * y:
        plot_ellipse_points(xc, yc, x, y)
        if p1 < 0:
            x += 1
            p1 += 2 * ry2 * x + ry2
        else:
            x += 1
            y -= 1
            p1 += 2 * ry2 * x - 2 * rx2 * y + ry2

    # Region 2 decision parameter
    p2 = ry2 * (x + 0.5) ** 2 + rx2 * (y - 1) ** 2 - rx2 * ry2

    # ----- Region 2 -----
    while y >= 0:
        plot_ellipse_points(xc, yc, x, y)
        if p2 > 0:
            y -= 1
            p2 -= 2 * rx2 * y + rx2
        else:
            x += 1
            y -= 1
            p2 += 2 * ry2 * x - 2 * rx2 * y + rx2

    glEnd()
```

Output



Conclusion

In this lab, fundamental computer graphics algorithms were implemented using OpenGL and Python. Line drawing algorithms such as DDA and Bresenham were studied for efficient rasterization of straight lines, while midpoint algorithms were used to draw circles and ellipses using symmetry and decision parameters. Additionally, graphical representations like line graphs and pie charts were generated from data sets. These implementations helped in understanding how mathematical models are converted into pixel-based graphics and provided practical exposure to basic 2D graphics rendering techniques.