

**Kathmandu University**  
**Department of Computer Science and Engineering**  
**Dhulikhel, Kavrepalanchowk**



**A Lab Report**  
**on**  
**“Computer Graphics”**  
**[Code No: COMP342]**

**Submitted by:**

**Salina Nakarmi**

**CS III-I (60)**

**Roll-No: 34**

**Submitted to:**

**Mr. Dhiraj Shrestha**

**Department of Computer Science and Engineering**

**Submission Date:**

**2025/12/23**

## Q.No.1: Implement DDA Line Drawing

**Algorithm.** The DDA Line drawing algorithm is provided below:

1. Start with the two endpoints  $(x_1, y_1)$  and  $(x_2, y_2)$ .

2. Compute the differences:

- $dx = x_2 - x_1$
- $dy = y_2 - y_1$

3. Determine the number of steps:

- $steps = \text{MAX}(|dx|, |dy|)$

4. Compute the increments:

- $x\_inc = dx / steps$
- $y\_inc = dy / steps$

5. Initialize:

- $x = x_1$
- $y = y_1$

6. Plot the initial point:

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

7. For each step from 1 to steps:

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

The DDA algorithm applied in OpenGL through Python3 is provided below:

```
# ===== Q1: DDA Line Drawing Algorithm =====
def dda_line(x1, y1, x2, y2):
    """Digital Differential Analyzer Line Drawing Algorithm"""
    dx = x2 - x1
    dy = y2 - y1

    # Calculate steps required
    steps = abs(dx) if abs(dx) > abs(dy) else abs(dy)

    if steps == 0:
        return

    # Calculate increment in x and y
    x_increment = dx / steps
    y_increment = dy / steps

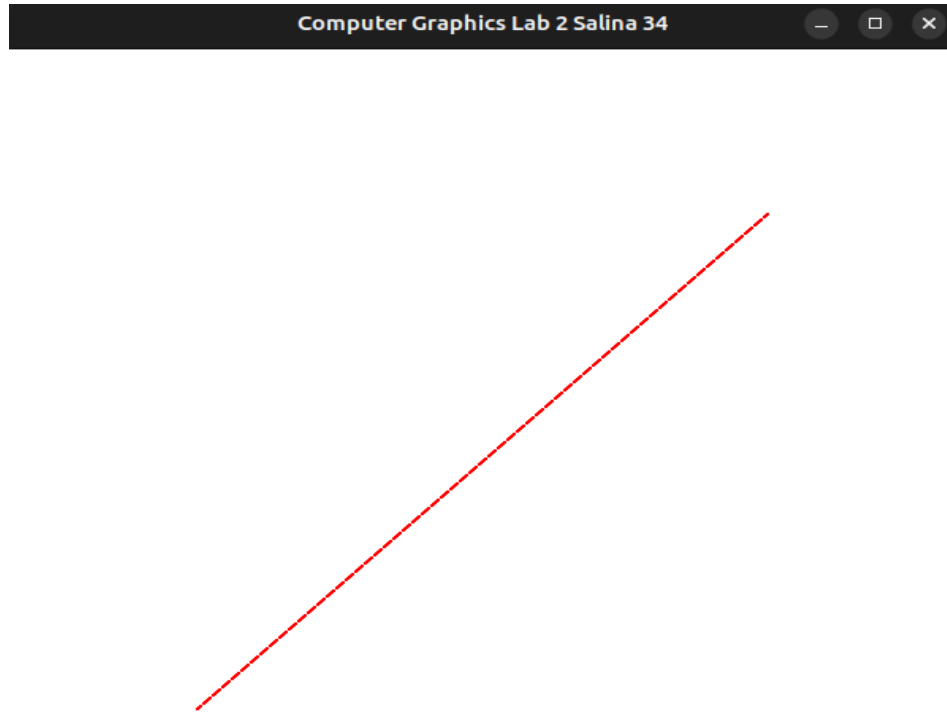
    # Initial point
    x = x1
    y = y1

    glBegin(GL_POINTS)
    for i in range(int(steps) + 1):
        glVertex2f(round(x), round(y))
        x += x_increment
        y += y_increment
    glEnd()

def display_dda():
    """Display function for DDA algorithm"""
    glClear(GL_COLOR_BUFFER_BIT)
    glClearColor(1.0, 1.0, 1.0, 1.0)
    glColor3f(1.0, 0.0, 0.0)
    glPointSize(2.0)
```

Figure 1: DDA

The output obtained is depicted below:



*Figure 2: DDA Output*

## Q.No.2: Implement Bresenham Line Drawing

**Algorithm.** The Bresenham Line Drawing Algorithm is as follows:

1. Start with two endpoints  $(x_1, y_1)$  and  $(x_2, y_2)$ .
2. Compute:
  - $dx = |x_2 - x_1|$
  - $dy = |y_2 - y_1|$
3. Determine the direction of movement:

- $s_x = 1$  if  $x_2 \geq x_1$ , otherwise  $s_x = -1$
- $s_y = 1$  if  $y_2 \geq y_1$ , otherwise  $s_y = -1$

4. Initialize:

- $x = x_1$
- $y = y_1$

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

- Initialize decision parameter:  $p = 2dy - dx$

6. For each step from 0 to  $dx$ :

- Plot( $x, y$ )

7. If  $p \geq 0$ :

- $y = y + s_y$
- $p = p + 2(dy - dx)$

8. Else:

- $p = p + 2dy$
- $x = x + s_x$

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

- Initialize decision parameter:  $p = 2dx - dy$

10. For each step from 0 to  $dy$ :

- Plot( $x, y$ )

11. If  $p \geq 0$ :

- $x = x + s_x$
- $p = p + 2(dx - dy)$

12. Else:

- $p = p + 2dx$

- $y = y + sy$

The code implementation of BLA is:

```
# ===== Q2: Bresenham Line Drawing Algorithm =====
def bresenham_line(x1, y1, x2, y2):
    """Bresenham Line Drawing Algorithm for both slopes"""
    dx = abs(x2 - x1)
    dy = abs(y2 - y1)

    # Determine direction
    sx = 1 if x2 >= x1 else -1
    sy = 1 if y2 >= y1 else -1

    x, y = x1, y1

    glBegin(GL_POINTS)

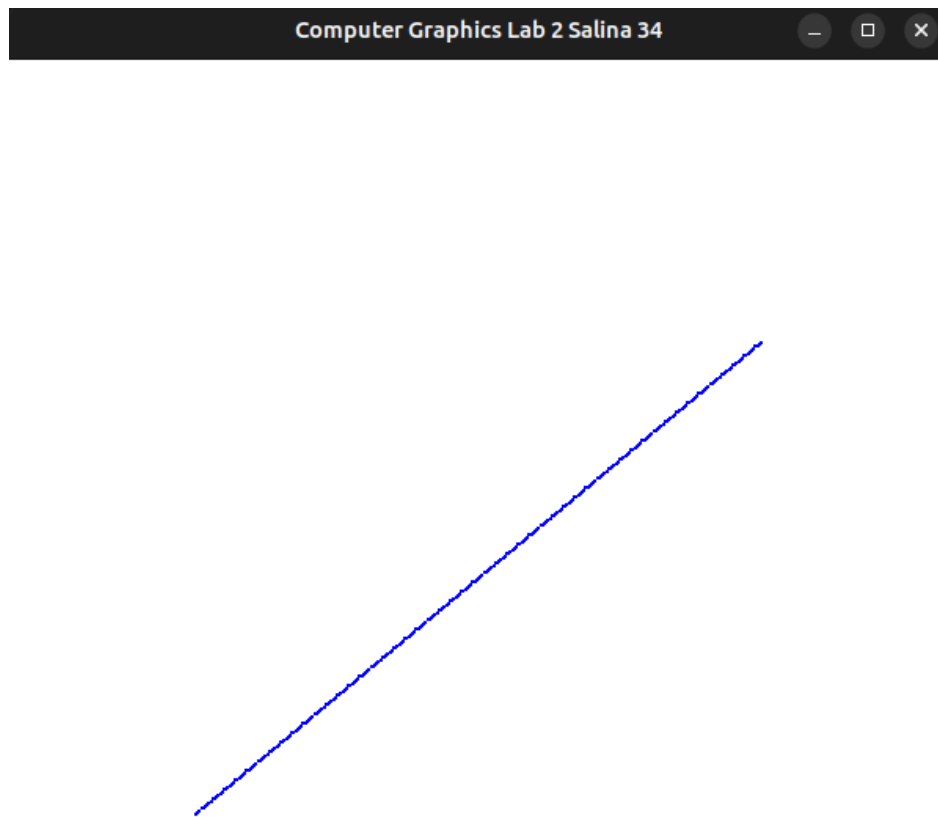
    # Case 1: |m| < 1 (dx > dy)
    if dx > dy:
        p = 2 * dy - dx
        for _ in range(dx + 1):
            glVertex2f(x, y)
            if p >= 0:
                y += sy
                p += 2 * (dy - dx)
            else:
                p += 2 * dy
            x += sx

    # Case 2: |m| >= 1 (dy >= dx)
    else:
        p = 2 * dx - dy
        for _ in range(dy + 1):
            glVertex2f(x, y)
            if p >= 0:
                x += sx
                p += 2 * (dx - dy)
            else:
                p += 2 * dx
            y += sy

    glEnd()
```

Figure 3: BLA Algorithm

The output received is:



*Figure 4: BLA Output*

### Q.No.3: Implement Midpoint Circle.

The midpoint circle drawing algorithm is:

1. Start with center  $(cx, cy)$  and radius  $r$ .
2. Initialize:
  - $x = 0$
  - $y = r$
  - $d = 1 - r$
3. Plot the initial eight symmetric points:
  - $(cx \pm x, cy \pm y), (cx \pm y, cy \pm x)$
4. Repeat while  $x \leq y$ :
5. Plot the eight symmetric points:
  - $(cx + x, cy + y)$
  - $(cx - x, cy + y)$
  - $(cx + x, cy - y)$
  - $(cx - x, cy - y)$
  - $(cx + y, cy + x)$
  - $(cx - y, cy + x)$
  - $(cx + y, cy - x)$
  - $(cx - y, cy - x)$
6. Update the decision parameter:
  - If  $d < 0$ :
    - $d = d + 2x + 3$
  - Else:
    - $d = d + 2(x - y) + 5$
    - $y = y - 1$
7. Increment:
  - $x = x + 1$

The Code Implementation of Midpoint Circle Algorithm is:

```
# ===== Q3: Midpoint Circle Drawing Algorithm =====
def plot_circle_points(xc, yc, x, y):
    """Plot 8 symmetric points of circle"""
    glBegin(GL_POINTS)
    glVertex2f(xc + x, yc + y)
    glVertex2f(xc - x, yc + y)
    glVertex2f(xc + x, yc - y)
    glVertex2f(xc - x, yc - y)
    glVertex2f(xc + y, yc + x)
    glVertex2f(xc - y, yc + x)
    glVertex2f(xc + y, yc - x)
    glVertex2f(xc - y, yc - x)
    glEnd()

def midpoint_circle(xc, yc, r):
    """Midpoint Circle Drawing Algorithm"""
    x = 0
    y = r
    d = 1 - r

    plot_circle_points(xc, yc, x, y)

    while x < y:
        x += 1
        if d < 0:
            d += 2 * x + 3
        else:
            y -= 1
            d += 2 * (x - y) + 5
        plot_circle_points(xc, yc, x, y)

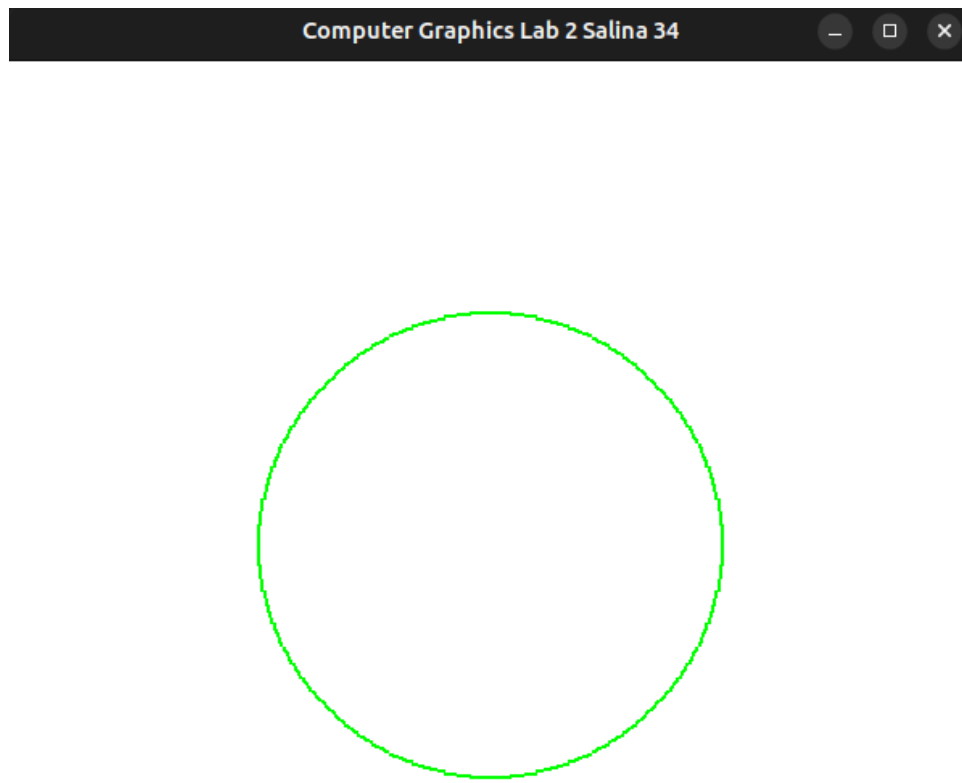
def display_circle():
    """Display function for circle"""
    glClear(GL_COLOR_BUFFER_BIT)
    glClearColor(1.0, 1.0, 1.0, 1.0)
    glColor3f(0.0, 1.0, 0.0)
    glPointSize(2.0)

    # Draw a single circle
    midpoint_circle(250, 250, 120)

    glFlush()
```

Figure 5: Midpoint Circle Algorithm

The output received is as follows:



*Figure 6: Midpoint Circle Algorithm*

#### **Q.No.4: Implement Line Graph.**

The algorithm for Random Line Generation using DDA is as follows:

1. **Choose the number of points**

- Let  $n$  be the number of data points.

## 2. Compute equal horizontal spacing

- Define a left and right margin.
- Compute:  $x\_gap = (window\_width - 2 \times margin) / (n - 1)$

## 3. Generate a dataset

- For each index  $i = 0$  to  $n - 1$ :
  - Compute:  $x_i = margin + i \times x\_gap$
  - Generate a random  $y$ -value within a chosen vertical range:  $y_i = random(y\_min, y\_max)$
  - Store all generated points as:  $P = \{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$

## 4. For each consecutive pair of points, generate a line using DDA

- For each segment between  $(x_i, y_i)$  and  $(x_{i+1}, y_{i+1})$ :
  - Compute:  $dx = x_{i+1} - x_i$ ,  $dy = y_{i+1} - y_i$
  - Determine number of steps:  $steps = \max(|dx|, |dy|)$
  - Compute increments:  $x\_inc = dx/steps$ ,  $y\_inc = dy/steps$
  - Initialize:  $x = x_i$ ,  $y = y_i$
  - For each step from 0 to steps:
    - Compute the plotted point:  $(round(x), round(y))$
    - Update:  $x = x + x\_inc$ ,  $y = y + y\_inc$

## 5. Store or output all computed points

- Collect every generated point from the DDA segments to form the complete line graph.

The code implementation of the algorithm is as follows:

```
# ===== Q4: Line Graph using DDA =====
def display_line_graph():
    """Display function for line graph using random data"""
    glClear(GL_COLOR_BUFFER_BIT)
    glClearColor(1.0, 1.0, 1.0, 1.0)

    # Generate random data points
    n = 8 # Number of points
    margin = 50
    window_width = 500
    x_gap = (window_width - 2 * margin) / (n - 1)

    # Generate random y values
    random.seed(42) # For consistent output
    data = []
    for i in range(n):
        x = margin + i * x_gap
        y = random.randint(100, 400)
        data.append((x, y))

    # Draw axes
    glColor3f(0.0, 0.0, 0.0)
    glPointSize(1.0)
    dda_line(40, 50, 40, 450) # Y-axis
    dda_line(40, 50, 500, 50) # X-axis

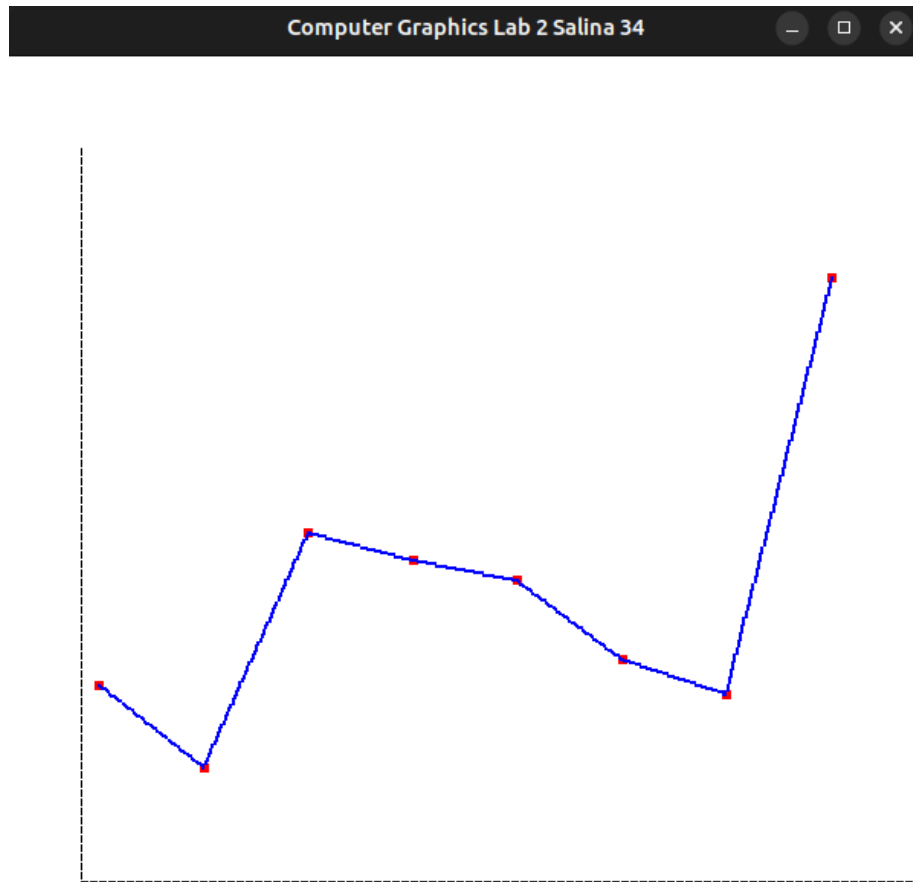
    # Draw data points
    glColor3f(1.0, 0.0, 0.0)
    glPointSize(6.0)
    glBegin(GL_POINTS)
    for point in data:
        glVertex2f(point[0], point[1])
    glEnd()

    # Connect points with lines using DDA
    glColor3f(0.0, 0.0, 1.0)
    glPointSize(2.0)
    for i in range(len(data) - 1):
        dda_line(int(data[i][0]), int(data[i][1]),
                 int(data[i+1][0]), int(data[i+1][1]))

    glFlush()
```

Figure 7: Random Line Graph

The output received is shown below:



*Figure 8: Line Graph Output*

### **Q.No.5: Implement Pie Chart.**

The pie-chart algorithm is as follows:

- A. Number of slices  $n = 6$
- B. Random integer values  $d_i$  for each slice
- C. Center coordinates  $(cx, cy)$
- D. Radius  $r$
- E. Steps per slice = 100

### 1. Generate Random Data

- For each slice  $i$  from 1 to 6:
  - Generate a random integer value:  $d_i = \text{Random}(5, 50)$
- Compute the total:  $T = \sum d_i$
- Convert each data value to an angle proportional to the total:  $\theta_i = (360 \times d_i) / T$

### 2. Compute Start and End Angles for Each Slice

- Set  $\text{start\_angle} = 0$
- For each slice  $i$ :
  - Compute  $\text{end\_angle} = \text{start\_angle} + \theta_i$
  - Store the pair  $(\text{start\_angle}, \text{end\_angle})$
  - Update:  $\text{start\_angle} = \text{end\_angle}$

### 3. Incremental Slice Boundary Generation

- For each slice  $i$ :
  - Initialize an empty list  $\text{points}$
  - Add the center point  $(cx, cy)$  as the first entry
  - For each step  $k$  from 0 to  $\text{steps}$ :
    - Compute the interpolated angle:  $\alpha = \text{start\_angle}_i + (k/\text{steps})(\text{end\_angle}_i - \text{start\_angle}_i)$
    - Convert to radians:  $\text{rad} = \alpha \times (\pi/180)$
    - Compute boundary point coordinates:
      - $x = cx + r \times \cos(\text{rad})$
      - $y = cy + r \times \sin(\text{rad})$
    - Round and append point:  $\text{points} \leftarrow \text{points} \cup \{(\text{round}(x), \text{round}(y))\}$
    -

### 4. Output the final list of points for slice $i$

The code implementation of the algorithm is:

```

# ===== Q5: Pie Chart =====
def display_pie_chart():
    """Display function for pie chart with random slices"""
    glClear(GL_COLOR_BUFFER_BIT)
    glClearColor(1.0, 1.0, 1.0, 1.0)

    cx, cy = 250, 250 # Center coordinates
    radius = 150
    n = 6 # Number of slices
    steps_per_slice = 100

    # Generate random data values
    random.seed(42)
    data = [random.randint(5, 50) for _ in range(n)]
    total = sum(data)

    # Convert to angles
    angles = [(360.0 * d / total) for d in data]

    # Define colors for each slice
    colors = [
        (1.0, 0.0, 0.0), # Red
        (0.0, 1.0, 0.0), # Green
        (0.0, 0.0, 1.0), # Blue
        (1.0, 1.0, 0.0), # Yellow
        (1.0, 0.0, 1.0), # Magenta
        (0.0, 1.0, 1.0) # Cyan
    ]

    start_angle = 0

    for i in range(n):
        end_angle = start_angle + angles[i]
        glColor3f(*colors[i])

        # Generate points for this slice
        points = [(cx, cy)] # Start with center

        for k in range(steps_per_slice + 1):
            alpha = start_angle + (k / steps_per_slice) * (end_angle - start_angle)
            rad = math.radians(alpha)
            x = cx + radius * math.cos(rad)
            y = cy + radius * math.sin(rad)
            points.append((round(x), round(y)))

        # Draw the slice using GL_TRIANGLE_FAN
        glBegin(GL_TRIANGLE_FAN)
        for point in points:
            glVertex2f(point[0], point[1])
        glEnd()

        start_angle = end_angle

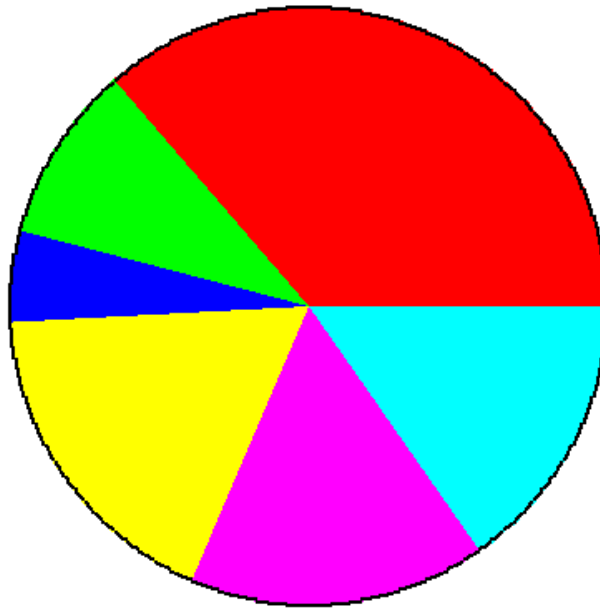
    # Draw border circle
    glColor3f(0.0, 0.0, 0.0)
    glPointSize(2.0)
    midpoint_circle(cx, cy, radius)

    glFlush()

```

Figure 9: Pie Chart Code

The output received is as follows:



*Figure 10: Pie-Chart Output*

### **Q.No.6: Implement Midpoint Ellipse Drawing Algorithm.**

The Midpoint Ellipse Drawing Algorithm is as follows:

1. Start with center  $(cx, cy)$  and radii  $rx$  (horizontal) and  $ry$  (vertical).
2. Initialize:
  - $x = 0$
  - $y = ry$
  - $rx^2 = rx \times rx$
  - $ry^2 = ry \times ry$
3. Region 1:
  - Initialize decision parameter:  $d1 = ry^2 - (rx^2 \times ry) + (0.25 \times rx^2)$
  - $dx = 2 \times ry^2 \times x$
  - $dy = 2 \times rx^2 \times y$
4. While  $dx < dy$ :
  - Plot four symmetric points:  $(cx \pm x, cy \pm y)$
  - Increment:  $x = x + 1$
  - Update:  $dx = dx + 2 \times ry^2$
  - If  $d1 < 0$ :

- $d1 = d1 + dx + ry^2$
  - Else:
    - $y = y - 1$
    - $dy = dy - 2 \times rx^2$
    - $d1 = d1 + dx - dy + ry^2$
- 5. Region 2:
  - Initialize decision parameter:  $d2 = ry^2 \times (x + 0.5)^2 + rx^2 \times (y - 1)^2 - rx^2 \times ry^2$
- 6. While  $y \geq 0$ :
  - Plot four symmetric points:  $(cx \pm x, cy \pm y)$
  - Decrement:  $y = y - 1$
  - Update:  $dy = dy - 2 \times rx^2$
  - If  $d2 > 0$ :
    - $d2 = d2 + rx^2 - dy$
  - Else:
    - $x = x + 1$
    - $dx = dx + 2 \times ry^2$
    - $d2 = d2 + dx - dy + rx^2$

The code implementation of the algorithm is:

```

# ===== Q6: Midpoint Ellipse Drawing Algorithm =====
def plot_ellipse_points(xc, yc, x, y):
    """Plot 4 symmetric points of ellipse"""
    glBegin(GL_POINTS)
    glVertex2f(xc + x, yc + y)
    glVertex2f(xc - x, yc + y)
    glVertex2f(xc + x, yc - y)
    glVertex2f(xc - x, yc - y)
    glEnd()

def midpoint_ellipse(xc, yc, rx, ry):
    """Midpoint Ellipse Drawing Algorithm"""
    x = 0
    y = ry

    rx2 = rx * rx
    ry2 = ry * ry
    two_rx2 = 2 * rx2
    two_ry2 = 2 * ry2

    # Region 1
    d1 = ry2 - (rx2 * ry) + (0.25 * rx2)
    dx = two_ry2 * x
    dy = two_rx2 * y

    plot_ellipse_points(xc, yc, x, y)

    while dx < dy:
        x += 1
        dx += two_ry2
        if d1 < 0:
            d1 += dx + ry2
        else:
            y -= 1
            dy -= two_rx2
            d1 += dx - dy + ry2
        plot_ellipse_points(xc, yc, x, y)

    # Region 2
    d2 = ry2 * (x + 0.5) * (x + 0.5) + rx2 * (y - 1) * (y - 1) - rx2 * ry2

    while y >= 0:
        y -= 1
        dy -= two_rx2
        if d2 > 0:
            d2 += rx2 - dy
        else:
            x += 1
            dx += two_ry2
            d2 += dx - dy + rx2
        plot_ellipse_points(xc, yc, x, y)

def display_ellipse():
    """Display function for ellipse"""
    glClear(GL_COLOR_BUFFER_BIT)
    glClearColor(1.0, 1.0, 1.0, 1.0)
    glColor3f(0.5, 0.0, 0.5)
    glPointSize(2.0)

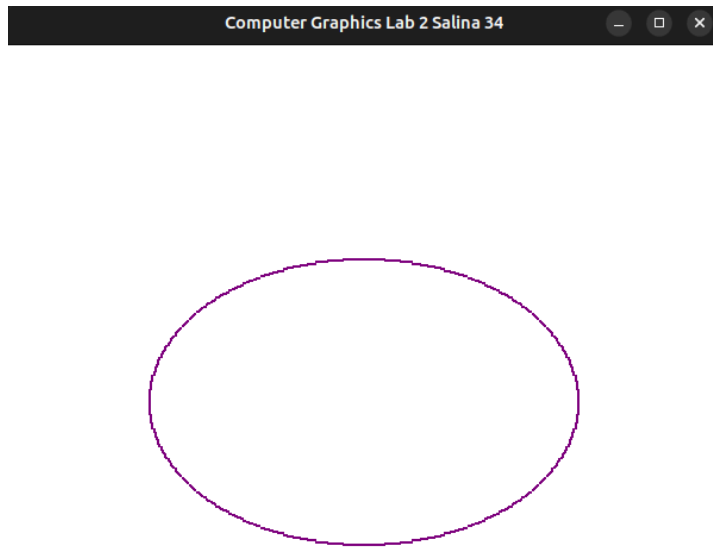
    # Draw a single ellipse
    midpoint_ellipse(250, 250, 150, 100)

    glFlush()

```

Figure 11: Midpoint Ellipse Code

The output received is as follows:



*Figure 12: Midpoint Ellipse Output*

## **Conclusion:**

In this way, we have implemented the different drawing algorithms discussed in the classroom.