

Kathmandu University
Department of Computer Science and Engineering
Dhulikhel, Kavre



A Lab Report
on
“Computer Graphics Lab-II”

[Code No.: COMP 342]

Submitted by

Suvesh Gurung
Roll No.: 24

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

December 25, 2025

Contents

1	Lab Activity 1: Digital Differential Analyzer (DDA) Algorithm	2
1.1	Title of Lab Activity	2
1.2	Algorithm	2
1.3	Source Code	2
1.4	Output	4
2	Lab Activity 2: Bresenham's Line Drawing Algorithm	5
2.1	Title of Lab Activity	5
2.2	Algorithm	5
2.3	Source Code	5
2.4	Output	7
3	Lab Activity 3: Mid-Point Circle Drawing Algorithm	8
3.1	Title of Lab Activity	8
3.2	Algorithm	8
3.3	Source Code	8
3.4	Output	10
4	Lab Activity 4: Line Graph Generation	11
4.1	Title of Lab Activity	11
4.2	Algorithm	11
4.3	Source Code	11
4.4	Output	15
5	Lab Activity 5: Pie Chart Implementation	17
5.1	Title of Lab Activity	17
5.2	Algorithm	17
5.3	Source Code	17
5.4	Output	21
6	Lab Activity 6: Mid-Point Ellipse Drawing Algorithm	22
6.1	Title of Lab Activity	22
6.2	Algorithm	22
6.3	Source Code	22
6.4	Output	26
7	Conclusion	27
7.1	Technical Achievements	27
7.2	OpenGL Integration	27

1 Lab Activity 1: Digital Differential Analyzer (DDA) Algorithm

1.1 Title of Lab Activity

Implementation of Digital Differential Analyzer (DDA) Line Drawing Algorithm using OpenGL and GLFW.

1.2 Algorithm

The Digital Differential Analyzer (DDA) algorithm is an incremental scan conversion method for rasterizing lines. The algorithm works as follows:

Steps:

1. Calculate $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$
2. Determine the step size: $steps = \max(|\Delta x|, |\Delta y|)$
3. Calculate increments: $x_{inc} = \frac{\Delta x}{steps}$ and $y_{inc} = \frac{\Delta y}{steps}$
4. Initialize $(x, y) = (x_1, y_1)$
5. For each step:
 - Plot point at $(\text{round}(x), \text{round}(y))$
 - Update $x = x + x_{inc}$ and $y = y + y_{inc}$

1.3 Source Code

```
1 import glfw
2 from OpenGL.GL import *
3
4 def dda_line(x1, y1, x2, y2):
5     points = []
6
7     dx = x2 - x1
8     dy = y2 - y1
9
10    # Determine number of steps
11    steps = max(abs(dx), abs(dy))
12
13    if steps == 0:
14        return [(x1, y1)]
15
16    # Calculate increment for each step
17    x_inc = dx / steps
18    y_inc = dy / steps
19
20    # Starting point
21    x, y = x1, y1
22
23    for _ in range(int(steps) + 1):
24        points.append((round(x), round(y)))
25        x += x_inc
26        y += y_inc
```

```

27     return points
28
29
30 def draw_line_points(points):
31     glBegin(GL_POINTS)
32     for x, y in points:
33         glVertex2f(x, y)
34     glEnd()
35
36 def main():
37     if not glfw.init():
38         return
39
40     width, height = 800, 600
41     window = glfw.create_window(width, height, "DDA Line Drawing Algorithm"
42     , None, None)
43
44     if not window:
45         glfw.terminate()
46         return
47
48     glfw.make_context_current(window)
49
50     glViewport(0, 0, width, height)
51     glMatrixMode(GL_PROJECTION)
52     glLoadIdentity()
53     glOrtho(0, width, height, 0, -1, 1) # 2D orthographic projection
54     glMatrixMode(GL_MODELVIEW)
55     glLoadIdentity()
56
57     glPointSize(2.0)
58
59     lines = [
60         (100, 100, 700, 500), # Diagonal line
61         (400, 50, 400, 550), # Vertical line
62         (50, 300, 750, 300), # Horizontal line
63         (200, 150, 600, 450), # Another diagonal
64         (150, 500, 650, 100), # Diagonal with negative slope
65     ]
66
67     all_points = []
68     for x1, y1, x2, y2 in lines:
69         points = dda_line(x1, y1, x2, y2)
70         all_points.extend(points)
71
72     while not glfw.window_should_close(window):
73         glClear(GL_COLOR_BUFFER_BIT)
74         glClearColor(0.0, 0.0, 0.0, 1.0)
75
76         glColor3f(1.0, 1.0, 1.0)
77
78         draw_line_points(all_points)
79
80         glfw.swap_buffers(window)
81         glfw.poll_events()
82
83     glfw.terminate()

```

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

1.4 Output

The program successfully displays multiple lines drawn using the DDA algorithm. The lines include:

- Diagonal line from (100, 100) to (700, 500)
- Vertical line from (400, 50) to (400, 550)
- Horizontal line from (50, 300) to (750, 300)

The algorithm produces smooth lines with proper pixel placement using floating-point arithmetic and rounding.

2 Lab Activity 2: Bresenham's Line Drawing Algorithm

2.1 Title of Lab Activity

Implementation of Bresenham's Line Drawing Algorithm for both slopes $|m| < 1$ and $|m| \geq 1$ using OpenGL and GLFW.

2.2 Algorithm

Bresenham's Line Drawing Algorithm is an efficient scan conversion algorithm that uses only integer arithmetic. It handles two cases based on slope:

Case 1: $|m| < 1$

1. Calculate $dx = |x_2 - x_1|$ and $dy = |y_2 - y_1|$
2. Initialize decision parameter: $p = 2dy - dx$
3. For each x from x_1 to x_2 :
 - Plot (x, y)
 - If $p \geq 0$: increment y , $p = p + 2(dy - dx)$
 - Else: $p = p + 2dy$

Case 2: $|m| \geq 1$

1. Calculate $dx = |x_2 - x_1|$ and $dy = |y_2 - y_1|$
2. Initialize decision parameter: $p = 2dx - dy$
3. For each y from y_1 to y_2 :
 - Plot (x, y)
 - If $p \geq 0$: increment x , $p = p + 2(dx - dy)$
 - Else: $p = p + 2dx$

2.3 Source Code

```
1 import glfw
2 from OpenGL.GL import *
3
4 def bresenham_line(x1, y1, x2, y2):
5     points = []
6
7     dx = abs(x2 - x1)
8     dy = abs(y2 - y1)
9
10    x_step = 1 if x2 > x1 else -1
11    y_step = 1 if y2 > y1 else -1
12
13    x, y = x1, y1
14
15    # Case 1: |m| < 1
```

```

16     if dx > dy:
17         p = 2 * dy - dx # Initial decision parameter
18
19         for _ in range(dx + 1):
20             points.append((x, y))
21
22             if p >= 0:
23                 y += y_step
24                 p += 2 * (dy - dx)
25             else:
26                 p += 2 * dy
27
28             x += x_step
29
30     # Case 2: |m| >= 1
31     else:
32         p = 2 * dx - dy # Initial decision parameter
33
34         for _ in range(dy + 1):
35             points.append((x, y))
36
37             if p >= 0:
38                 x += x_step
39                 p += 2 * (dx - dy)
40             else:
41                 p += 2 * dx
42
43             y += y_step
44
45     return points
46
47 def draw_line_points(points, color=(1.0, 1.0, 1.0)):
48     glColor3f(*color)
49     glBegin(GL_POINTS)
50     for x, y in points:
51         glVertex2f(x, y)
52     glEnd()
53
54 def main():
55     if not glfw.init():
56         return
57
58     width, height = 800, 600
59     window = glfw.create_window(width, height, "Bresenham Line Drawing
Algorithm", None, None)
60
61     if not window:
62         glfw.terminate()
63         return
64
65     glfw.make_context_current(window)
66
67     glViewport(0, 0, width, height)
68     glMatrixMode(GL_PROJECTION)
69     glLoadIdentity()
70     glOrtho(0, width, height, 0, -1, 1) # 2D orthographic projection
71     glMatrixMode(GL_MODELVIEW)
72     glLoadIdentity()

```

```

73     glPointSize(3.0)
74
75
76     lines = [
77         #Gentle slopes ( $|m| < 1$ ) - Red
78         ((100, 300, 400, 350), (1.0, 0.0, 0.0)),
79         ((100, 200, 400, 150), (1.0, 0.0, 0.0)),
80         ((450, 300, 750, 300), (1.0, 0.0, 0.0)),
81
82         # Steep slopes ( $|m| \geq 1$ ) - Green
83         ((200, 100, 250, 500), (0.0, 1.0, 0.0)),
84         ((350, 500, 400, 100), (0.0, 1.0, 0.0)),
85         ((600, 100, 600, 500), (0.0, 1.0, 0.0)),
86
87         # 45-degree lines ( $|m| = 1$ ) - Blue
88         ((100, 100, 300, 300), (0.0, 0.0, 1.0)),
89         ((500, 100, 700, 300), (0.0, 0.0, 1.0)),
90
91         # Mixed slopes - Yellow
92         ((50, 450, 350, 100), (1.0, 1.0, 0.0)),
93         ((450, 450, 750, 150), (1.0, 1.0, 0.0)),
94     ]
95
96     all_line_data = []
97     for (x1, y1, x2, y2), color in lines:
98         points = bresenham_line(x1, y1, x2, y2)
99         all_line_data.append((points, color))
100
101     dx = abs(x2 - x1)
102     dy = abs(y2 - y1)
103
104     while not glfw.window_should_close(window):
105         glClear(GL_COLOR_BUFFER_BIT)
106         glClearColor(0.1, 0.1, 0.1, 1.0)
107
108         for points, color in all_line_data:
109             draw_line_points(points, color)
110
111         glfw.swap_buffers(window)
112         glfw.poll_events()
113
114     glfw.terminate()
115
116 if __name__ == "__main__":
117     main()

```

2.4 Output

The program displays lines in different colors representing different slope categories:

- **Red lines:** $|m| < 1$
- **Green lines:** $|m| \geq 1$
- **Blue lines:** $|m| = 1$

The algorithm successfully handles all octants and uses only integer arithmetic for efficiency.

3 Lab Activity 3: Mid-Point Circle Drawing Algorithm

3.1 Title of Lab Activity

Implementation of Mid-Point Circle Drawing Algorithm using OpenGL and GLFW.

3.2 Algorithm

The Mid-Point Circle Algorithm (Bresenham's Circle Algorithm) uses 8-way symmetry and integer arithmetic to efficiently draw circles.

Steps:

1. Initialize $(x, y) = (0, r)$ where r is the radius
2. Calculate initial decision parameter: $p = 1 - r$
3. Plot 8 symmetric points for current (x, y)
4. While $x < y$:
 - Increment x
 - If $p < 0$: $p = p + 2x + 1$
 - Else: $y = y - 1$, $p = p + 2(x - y) + 1$
 - Plot 8 symmetric points

8-way symmetry points: $(x_c \pm x, y_c \pm y)$ and $(x_c \pm y, y_c \pm x)$

3.3 Source Code

```
1 import glfw
2 from OpenGL.GL import *
3
4 def plot_circle_points(xc, yc, x, y, points):
5     points.extend([
6         (xc + x, yc + y), # Octant 1
7         (xc - x, yc + y), # Octant 4
8         (xc + x, yc - y), # Octant 8
9         (xc - x, yc - y), # Octant 5
10        (xc + y, yc + x), # Octant 2
11        (xc - y, yc + x), # Octant 3
12        (xc + y, yc - x), # Octant 7
13        (xc - y, yc - x), # Octant 6
14    ])
15
16 def midpoint_circle(xc, yc, r):
17     points = []
18
19     x = 0
20     y = r
21
22     p = 1 - r
23
24     plot_circle_points(xc, yc, x, y, points)
```

```

25
26     # Iterate until x >= y
27     while x < y:
28         x += 1
29
30         if p < 0:
31             # Mid-point is inside the circle
32             p += 2 * x + 1
33         else:
34             # Mid-point is outside the circle
35             y -= 1
36             p += 2 * (x - y) + 1
37
38         # Plot points in all 8 octants
39         plot_circle_points(xc, yc, x, y, points)
40
41     return points
42
43 def draw_points(points, color=(1.0, 1.0, 1.0)):
44     glColor3f(*color)
45     glBegin(GL_POINTS)
46     for x, y in points:
47         glVertex2f(x, y)
48     glEnd()
49
50 def draw_filled_circle(xc, yc, r, color=(1.0, 1.0, 1.0)):
51     """
52     Optional: Draw a filled circle using the mid-point algorithm
53     by drawing horizontal lines between symmetric points
54     """
55     glColor4f(*color, 0.3)    # Semi-transparent fill
56
57     x = 0
58     y = r
59     p = 1 - r
60
61     glBegin(GL_LINES)
62     for i in range(-r, r + 1):
63         glVertex2f(xc - r, yc + i)
64         glVertex2f(xc + r, yc + i)
65     glEnd()
66
67 def main():
68     if not glfw.init():
69         return
70
71     width, height = 800, 600
72     window = glfw.create_window(width, height, "Mid-Point Circle Drawing
73     Algorithm", None, None)
74
75     if not window:
76         glfw.terminate()
77         return
78
79     glfw.make_context_current(window)
80
81     glViewport(0, 0, width, height)
82     glMatrixMode(GL_PROJECTION)

```

```

82     glLoadIdentity()
83     glOrtho(0, width, height, 0, -1, 1)    # 2D orthographic projection
84     glMatrixMode(GL_MODELVIEW)
85     glLoadIdentity()
86
87     glEnable(GL_BLEND)
88     glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
89
90     glPointSize(2.0)
91
92     # Define circles to draw
93     circles = [
94         # (center_x, center_y, radius, color)
95         (200, 150, 80, (1.0, 0.0, 0.0)),      # Red circle
96         (400, 300, 120, (0.0, 1.0, 0.0)),     # Green circle (center)
97         (600, 150, 60, (0.0, 0.0, 1.0)),      # Blue circle
98         (200, 450, 100, (1.0, 1.0, 0.0)),     # Yellow circle
99         (600, 450, 90, (1.0, 0.0, 1.0)),      # Magenta circle
100        (400, 150, 40, (0.0, 1.0, 1.0)),     # Cyan circle (small)
101    ]
102
103    # Calculate all circle points using mid-point algorithm
104    all_circle_data = []
105    for xc, yc, r, color in circles:
106        points = midpoint_circle(xc, yc, r)
107        all_circle_data.append((points, color))
108
109        print(f"Circle at ({xc},{yc}) with radius {r}: {len(points)} points
110 generated")
111
112    while not glfw.window_should_close(window):
113        glClear(GL_COLOR_BUFFER_BIT)
114        glClearColor(0.05, 0.05, 0.05, 1.0)
115
116        for points, color in all_circle_data:
117            draw_points(points, color)
118
119        glfw.swap_buffers(window)
120        glfw.poll_events()
121
122    glfw.terminate()
123
124 if __name__ == "__main__":
125     main()

```

3.4 Output

The program displays multiple circles of different sizes and colors:

- Red circle at (200, 150) with radius 80
- Green circle at (400, 300) with radius 120
- Blue circle at (600, 150) with radius 60

The algorithm efficiently generates smooth circles using only integer arithmetic and 8-way symmetry.

4 Lab Activity 4: Line Graph Generation

4.1 Title of Lab Activity

Implementation of Line Graph using DDA/Bresenham Algorithm with OpenGL and GLFW.

4.2 Algorithm

The line graph generation combines data normalization with line drawing algorithms:

Steps:

1. Data Normalization:

- Find min and max values: $x_{min}, x_{max}, y_{min}, y_{max}$
- Map data to pixel coordinates:

$$pixel_x = margin + \frac{x - x_{min}}{x_{max} - x_{min}} \times draw_width$$
$$pixel_y = height - margin - \frac{y - y_{min}}{y_{max} - y_{min}} \times draw_height$$

2. Line Generation:

- Connect consecutive data points using Bresenham or DDA
- Generate all intermediate pixels between points

3. Rendering:

- Draw grid and axes
- Plot the line graph
- Mark actual data points

4.3 Source Code

```
1 import glfw
2 from OpenGL.GL import *
3
4 def bresenham_line(x1, y1, x2, y2):
5     points = []
6
7     dx = abs(x2 - x1)
8     dy = abs(y2 - y1)
9
10    x_step = 1 if x2 > x1 else -1
11    y_step = 1 if y2 > y1 else -1
12
13    x, y = x1, y1
14
15    if dx > dy:
16        p = 2 * dy - dx
17        for _ in range(dx + 1):
18            points.append((x, y))
```

```

19         if p >= 0:
20             y += y_step
21             p += 2 * (dy - dx)
22         else:
23             p += 2 * dy
24             x += x_step
25     else:
26         p = 2 * dx - dy
27         for _ in range(dy + 1):
28             points.append((x, y))
29             if p >= 0:
30                 x += x_step
31                 p += 2 * (dx - dy)
32             else:
33                 p += 2 * dx
34                 y += y_step
35
36     return points
37
38 def dda_line(x1, y1, x2, y2):
39     points = []
40
41     dx = x2 - x1
42     dy = y2 - y1
43
44     steps = max(abs(dx), abs(dy))
45
46     if steps == 0:
47         return [(x1, y1)]
48
49     x_inc = dx / steps
50     y_inc = dy / steps
51
52     x, y = x1, y1
53
54     for _ in range(int(steps) + 1):
55         points.append((round(x), round(y)))
56         x += x_inc
57         y += y_inc
58
59     return points
60
61 def normalize_data(data, width, height, margin=50):
62     """
63     Normalize data to fit within the window dimensions.
64
65     Parameters:
66     data: List of (x, y) tuples representing the dataset
67     width, height: Window dimensions
68     margin: Margin from window edges
69
70     Returns: Normalized data points as pixel coordinates
71     """
72     if not data:
73         return []
74
75     # Find min and max values
76     x_values = [point[0] for point in data]

```

```

77     y_values = [point[1] for point in data]
78
79     x_min, x_max = min(x_values), max(x_values)
80     y_min, y_max = min(y_values), max(y_values)
81
82     # Avoid division by zero
83     x_range = x_max - x_min if x_max != x_min else 1
84     y_range = y_max - y_min if y_max != y_min else 1
85
86     # Available drawing area
87     draw_width = width - 2 * margin
88     draw_height = height - 2 * margin
89
90     # Normalize to pixel coordinates
91     normalized = []
92     for x, y in data:
93         # Map x from [x_min, x_max] to [margin, width-margin]
94         pixel_x = margin + ((x - x_min) / x_range) * draw_width
95         # Map y from [y_min, y_max] to [height-margin, margin] (inverted
for screen coords)
96         pixel_y = height - margin - ((y - y_min) / y_range) * draw_height
97         normalized.append((int(pixel_x), int(pixel_y)))
98
99     return normalized, (x_min, x_max, y_min, y_max)
100
101 def generate_graph_lines(data_points, algorithm='bresenham'):
102     all_points = []
103
104     line_func = bresenham_line if algorithm == 'bresenham' else dda_line
105
106     for i in range(len(data_points) - 1):
107         x1, y1 = data_points[i]
108         x2, y2 = data_points[i + 1]
109
110         line_points = line_func(x1, y1, x2, y2)
111         all_points.extend(line_points)
112
113     return all_points
114
115 def draw_points(points, color=(1.0, 1.0, 1.0)):
116     glColor3f(*color)
117     glBegin(GL_POINTS)
118     for x, y in points:
119         glVertex2f(x, y)
120     glEnd()
121
122 def draw_axes(width, height, margin=50):
123     glColor3f(0.5, 0.5, 0.5)
124     glLineWidth(1.0)
125
126     glBegin(GL_LINES)
127     # X-axis
128     glVertex2f(margin, height - margin)
129     glVertex2f(width - margin, height - margin)
130
131     # Y-axis
132     glVertex2f(margin, margin)
133     glVertex2f(margin, height - margin)

```

```

134     glEnd()
135
136 def draw_data_points_markers(points, color=(1.0, 0.0, 0.0)):
137     glColor3f(*color)
138     glPointSize(8.0)
139     glBegin(GL_POINTS)
140     for x, y in points:
141         glVertex2f(x, y)
142     glEnd()
143
144 def draw_grid(width, height, margin, divisions=10):
145     glColor3f(0.2, 0.2, 0.2)
146     glLineWidth(1.0)
147
148     draw_width = width - 2 * margin
149     draw_height = height - 2 * margin
150
151     glBegin(GL_LINES)
152     # Vertical grid lines
153     for i in range(divisions + 1):
154         x = margin + (draw_width / divisions) * i
155         glVertex2f(x, margin)
156         glVertex2f(x, height - margin)
157
158     # Horizontal grid lines
159     for i in range(divisions + 1):
160         y = margin + (draw_height / divisions) * i
161         glVertex2f(margin, y)
162         glVertex2f(width - margin, y)
163     glEnd()
164
165 def main():
166     if not glfw.init():
167         return
168
169     width, height = 1000, 700
170     window = glfw.create_window(width, height, "Line Graph - DDA/Bresenham
Algorithm", None, None)
171
172     if not window:
173         glfw.terminate()
174         return
175
176     glfw.make_context_current(window)
177
178     glViewport(0, 0, width, height)
179     glMatrixMode(GL_PROJECTION)
180     glLoadIdentity()
181     glOrtho(0, width, height, 0, -1, 1)
182     glMatrixMode(GL_MODELVIEW)
183     glLoadIdentity()
184
185     # Sample datasets to visualize
186     datasets = [
187         {
188             'name': 'Sales Data (Bresenham)',
189             'data': [(1, 20), (2, 35), (3, 30), (4, 50), (5, 45), (6, 60),
(7, 55), (8, 70), (9, 75), (10, 85)],

```

```

190         'algorithm': 'bresenham',
191         'color': (0.2, 0.8, 1.0)
192     },
193     {
194         'name': 'Temperature Data (DDA)',
195         'data': [(1, 15), (2, 18), (3, 22), (4, 25), (5, 28), (6, 32),
196 (7, 30), (8, 27), (9, 23), (10, 20)],
197         'algorithm': 'dda',
198         'color': (1.0, 0.5, 0.2)
199     }
200 ]
201
202 # 0 -> sales data (DDA), 1 -> temperature data (Bresenham)
203 current_dataset = 0
204 dataset = datasets[current_dataset]
205
206 margin = 80
207
208 # Normalize data to window coordinates
209 normalized_points, bounds = normalize_data(dataset['data'], width,
height, margin)
210
211 # Generate line graph using specified algorithm
212 graph_points = generate_graph_lines(normalized_points, dataset['
algorithm'])
213
214 for i, (x, y) in enumerate(dataset['data'], 1):
215     print(f" Point {i}: ({x}, {y})")
216
217 glPointSize(2.0)
218
219 while not glfw.window_should_close(window):
220     glClear(GL_COLOR_BUFFER_BIT)
221     glClearColor(0.05, 0.05, 0.1, 1.0)
222
223     draw_grid(width, height, margin, divisions=10)
224
225     draw_axes(width, height, margin)
226
227     draw_points(graph_points, dataset['color'])
228
229     draw_data_points_markers(normalized_points, (1.0, 0.0, 0.0))
230
231     glfw.swap_buffers(window)
232     glfw.poll_events()
233
234     glfw.terminate()
235
236 if __name__ == "__main__":
237     main()

```

4.4 Output

The program generates a line graph displaying data trends with:

- Normalized data points fitted to window dimensions

-
- Smooth line connecting data points using the Bresenham algorithm
 - Grid lines for reference
 - Coordinate axes
 - Data point markers

Example data: Sales data showing values from 20 to 70 over 8 time periods.

5 Lab Activity 5: Pie Chart Implementation

5.1 Title of Lab Activity

Implementation of a Pie Chart using OpenGL and GLFW.

5.2 Algorithm

The pie chart algorithm converts data values into circular sectors:

Steps:

1. Calculate Angles:

- Total = $\sum_{i=1}^n value_i$
- For each value: $angle_i = \frac{value_i}{Total} \times 2\pi$
- Start angle = $-\pi/2$ (12 o'clock position)

2. Draw Sectors:

- Use GL_TRIANGLE_FAN for filled sectors
- Center point at (x_c, y_c)
- For each segment: $x = x_c + r \cos(\theta)$, $y = y_c + r \sin(\theta)$

3. Render Components:

- Draw filled sectors with colors
- Draw sector outlines
- Draw legend with color boxes

5.3 Source Code

```
1 import glfw
2 from OpenGL.GL import *
3 import math
4
5 def draw_filled_circle_sector(cx, cy, radius, start_angle, end_angle, color,
6     , segments=100):
7     glColor3f(*color)
8     glBegin(GL_TRIANGLE_FAN)
9
10    glVertex2f(cx, cy)
11
12    # Calculate number of segments for this sector
13    angle_range = end_angle - start_angle
14    sector_segments = max(int(segments * (angle_range / (2 * math.pi))), 2)
15
16    # Draw vertices along the arc
17    for i in range(sector_segments + 1):
18        angle = start_angle + (angle_range * i / sector_segments)
19        x = cx + radius * math.cos(angle)
20        y = cy + radius * math.sin(angle)
```

```

20         glVertex2f(x, y)
21
22     glEnd()
23
24 def draw_circle_outline(cx, cy, radius, color=(1.0, 1.0, 1.0), segments
25 =100):
26     glColor3f(*color)
27     glLineWidth(2.0)
28     glBegin(GL_LINE_LOOP)
29
30     for i in range(segments):
31         angle = 2.0 * math.pi * i / segments
32         x = cx + radius * math.cos(angle)
33         y = cy + radius * math.sin(angle)
34         glVertex2f(x, y)
35
36     glEnd()
37
38 def draw_sector_outline(cx, cy, radius, start_angle, end_angle, color=(1.0,
39 1.0, 1.0), segments=100):
40     glColor3f(*color)
41     glLineWidth(2.0)
42
43     # Draw the arc
44     glBegin(GL_LINE_STRIP)
45     angle_range = end_angle - start_angle
46     sector_segments = max(int(segments * (angle_range / (2 * math.pi))), 2)
47
48     for i in range(sector_segments + 1):
49         angle = start_angle + (angle_range * i / sector_segments)
50         x = cx + radius * math.cos(angle)
51         y = cy + radius * math.sin(angle)
52         glVertex2f(x, y)
53
54     glEnd()
55
56     # Draw radial lines
57     glBegin(GL_LINES)
58     # Start radius
59     glVertex2f(cx, cy)
60     glVertex2f(cx + radius * math.cos(start_angle), cy + radius * math.sin(
61     start_angle))
62     # End radius
63     glVertex2f(cx, cy)
64     glVertex2f(cx + radius * math.cos(end_angle), cy + radius * math.sin(
65     end_angle))
66     glEnd()
67
68 def create_pie_chart(data, labels, colors, cx, cy, radius):
69     total = sum(data)
70     sectors = []
71
72     current_angle = -math.pi / 2 # Start at top (12 o'clock position)
73
74     for i, (value, label, color) in enumerate(zip(data, labels, colors)):
75         percentage = (value / total) * 100
76         angle_size = (value / total) * 2 * math.pi

```

```

74     sector_info = {
75         'value': value,
76         'percentage': percentage,
77         'label': label,
78         'color': color,
79         'start_angle': current_angle,
80         'end_angle': current_angle + angle_size,
81         'mid_angle': current_angle + angle_size / 2
82     }
83
84     sectors.append(sector_info)
85     current_angle += angle_size
86
87     return sectors
88
89 def draw_pie_chart(sectors, cx, cy, radius):
90     for sector in sectors:
91         draw_filled_circle_sector(
92             cx, cy, radius,
93             sector['start_angle'],
94             sector['end_angle'],
95             sector['color']
96         )
97
98         # Draw outline for each sector
99         draw_sector_outline(
100             cx, cy, radius,
101             sector['start_angle'],
102             sector['end_angle'],
103             (0.2, 0.2, 0.2)
104         )
105
106 def draw_legend(sectors, x, y, box_size=20, spacing=30):
107     for i, sector in enumerate(sectors):
108         y_pos = y + i * spacing
109
110         # Draw colored box
111         glColor3f(*sector['color'])
112         glBegin(GL_QUADS)
113         glVertex2f(x, y_pos)
114         glVertex2f(x + box_size, y_pos)
115         glVertex2f(x + box_size, y_pos + box_size)
116         glVertex2f(x, y_pos + box_size)
117         glEnd()
118
119         # Draw box outline
120         glColor3f(1.0, 1.0, 1.0)
121         glLineWidth(1.0)
122         glBegin(GL_LINE_LOOP)
123         glVertex2f(x, y_pos)
124         glVertex2f(x + box_size, y_pos)
125         glVertex2f(x + box_size, y_pos + box_size)
126         glVertex2f(x, y_pos + box_size)
127         glEnd()
128
129 def draw_labels_on_chart(sectors, cx, cy, radius):
130     for sector in sectors:
131         # Calculate position for label

```

```

132     label_radius = radius * 0.6
133     mid_angle = sector['mid_angle']
134
135     label_x = cx + label_radius * math.cos(mid_angle)
136     label_y = cy + label_radius * math.sin(mid_angle)
137
138     # Draw a small circle at label position to show percentage location
139     glColor3f(1.0, 1.0, 1.0)
140     glPointSize(8.0)
141     glBegin(GL_POINTS)
142     glVertex2f(label_x, label_y)
143     glEnd()
144
145 def main():
146     if not glfw.init():
147         return
148
149     width, height = 1000, 700
150     window = glfw.create_window(width, height, "Pie Chart - OpenGL
Implementation", None, None)
151
152     if not window:
153         glfw.terminate()
154         return
155
156     glfw.make_context_current(window)
157
158     glViewport(0, 0, width, height)
159     glMatrixMode(GL_PROJECTION)
160     glLoadIdentity()
161     glOrtho(0, width, height, 0, -1, 1)
162     glMatrixMode(GL_MODELVIEW)
163     glLoadIdentity()
164
165     glEnable(GL_LINE_SMOOTH)
166     glEnable(GL_BLEND)
167     glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
168     glHint(GL_LINE_SMOOTH_HINT, GL_NICEST)
169
170     data = [30, 45, 15, 60, 25]
171     labels = ['Product A', 'Product B', 'Product C', 'Product D', 'Product
E']
172     colors = [
173         (1.0, 0.2, 0.2),    # Red
174         (0.2, 0.8, 0.2),    # Green
175         (0.2, 0.4, 1.0),    # Blue
176         (1.0, 0.8, 0.2),    # Yellow
177         (1.0, 0.4, 0.8),    # Pink
178     ]
179
180     center_x = width // 2 - 100
181     center_y = height // 2
182     radius = 200
183
184     sectors = create_pie_chart(data, labels, colors, center_x, center_y,
radius)
185
186     print("Pie Chart Data:")

```

```

187 print("-" * 60)
188 total = sum(data)
189 for sector in sectors:
190     print(f"{sector['label']}: {sector['value']:.1f} ({sector[''
percentage']:.5.1f}%)")
191     print("-" * 60)
192 print(f"{'Total':12s}: {total:.1f} (100.0%)")
193 print("\nColors:")
194 for sector in sectors:
195     print(f"{sector['label']}: {RGB{sector['color']}")

196
197 while not glfw.window_should_close(window):
198     glClear(GL_COLOR_BUFFER_BIT)
199     glClearColor(0.1, 0.1, 0.15, 1.0)

200     draw_pie_chart(sectors, center_x, center_y, radius)

201     draw_labels_on_chart(sectors, center_x, center_y, radius)

202     draw_legend(sectors, width - 250, 100)

203
204     glColor3f(0.7, 0.7, 0.7)
205     glLineWidth(2.0)
206     glBegin(GL_LINES)
207     glVertex2f(width // 2 - 100, 30)
208     glVertex2f(width // 2 + 100, 30)
209     glEnd()

210
211     glfw.swap_buffers(window)
212     glfw.poll_events()

213
214     glfw.terminate()

215 if __name__ == "__main__":
216     main()

```

5.4 Output

The program displays a colorful pie chart with:

- 5 sectors representing different products
- Each sector colored differently for distinction
- Proportional representation based on values
- Legend showing color-label mapping

Example data:

- Product A: 30 (17.1%)
- Product B: 45 (25.7%)
- Product C: 15 (8.6%)
- Product D: 60 (34.3%)
- Product E: 25 (14.3%)

6 Lab Activity 6: Mid-Point Ellipse Drawing Algorithm

6.1 Title of Lab Activity

Implementation of the Mid-Point Ellipse Drawing Algorithm using OpenGL and GLFW.

6.2 Algorithm

The Mid-Point Ellipse Algorithm uses 4-way symmetry and processes two regions:

Region 1 (where $\text{slope} < -1$):

1. Initialize: $x = 0, y = r_y$
2. Decision parameter: $p_1 = r_y^2 - r_x^2 \cdot r_y + 0.25 \cdot r_x^2$
3. While $2 \cdot r_y^2 \cdot x < 2 \cdot r_x^2 \cdot y$:
 - Plot 4 symmetric points
 - $x = x + 1$
 - If $p_1 < 0$: $p_1 = p_1 + 2r_y^2 \cdot x + r_y^2$
 - Else: $y = y - 1, p_1 = p_1 + 2r_y^2 \cdot x - 2r_x^2 \cdot y + r_y^2$

Region 2 (where $\text{slope} \geq -1$):

1. Recalculate: $p_2 = r_y^2(x + 0.5)^2 + r_x^2(y - 1)^2 - r_x^2 \cdot r_y^2$
2. While $y \geq 0$:
 - Plot 4 symmetric points
 - $y = y - 1$
 - If $p_2 > 0$: $p_2 = p_2 - 2r_x^2 \cdot y + r_x^2$
 - Else: $x = x + 1, p_2 = p_2 + 2r_y^2 \cdot x - 2r_x^2 \cdot y + r_x^2$

4-way symmetry points: $(x_c \pm x, y_c \pm y)$

6.3 Source Code

```
1 import glfw
2 from OpenGL.GL import *
3 import math
4
5 def plot_ellipse_points(xc, yc, x, y, points):
6     points.extend([
7         (xc + x, yc + y), # Quadrant 1
8         (xc - x, yc + y), # Quadrant 2
9         (xc - x, yc - y), # Quadrant 3
10        (xc + x, yc - y), # Quadrant 4
11    ])
12
13 def midpoint_ellipse(xc, yc, rx, ry):
14     points = []
```

```

15
16     # Region 1: Slope < -1
17     x = 0
18     y = ry
19
20     # Initial decision parameters
21     rx_sq = rx * rx
22     ry_sq = ry * ry
23     two_rx_sq = 2 * rx_sq
24     two_ry_sq = 2 * ry_sq
25
26     # Region 1 decision parameter
27     p1 = ry_sq - (rx_sq * ry) + (0.25 * rx_sq)
28
29     dx = two_ry_sq * x
30     dy = two_rx_sq * y
31
32     # Region 1: Continue while slope < -1
33     while dx < dy:
34         plot_ellipse_points(xc, yc, x, y, points)
35
36         x += 1
37         dx += two_ry_sq
38
39         if p1 < 0:
40             p1 += dx + ry_sq
41         else:
42             y -= 1
43             dy -= two_rx_sq
44             p1 += dx - dy + ry_sq
45
46     # Region 2: Slope >= -1
47     # Recalculate decision parameter for region 2
48     p2 = ry_sq * (x + 0.5) * (x + 0.5) + rx_sq * (y - 1) * (y - 1) - rx_sq
49     * ry_sq
50
51     while y >= 0:
52         plot_ellipse_points(xc, yc, x, y, points)
53
54         y -= 1
55         dy -= two_rx_sq
56
57         if p2 > 0:
58             p2 += rx_sq - dy
59         else:
60             x += 1
61             dx += two_ry_sq
62             p2 += dx - dy + rx_sq
63
64     return points
65
66 def draw_points(points, color=(1.0, 1.0, 1.0)):
67     glColor3f(*color)
68     glBegin(GL_POINTS)
69     for x, y in points:
70         glVertex2f(x, y)
71     glEnd()

```

```

72 def draw_axes(xc, yc, rx, ry):
73     glColor3f(0.4, 0.4, 0.4)
74     glLineWidth(1.0)
75
76     glBegin(GL_LINES)
77     # Major axis (horizontal)
78     glVertex2f(xc - rx - 20, yc)
79     glVertex2f(xc + rx + 20, yc)
80
81     # Minor axis (vertical)
82     glVertex2f(xc, yc - ry - 20)
83     glVertex2f(xc, yc + ry + 20)
84     glEnd()
85
86     # Draw center point
87     glColor3f(1.0, 1.0, 0.0)
88     glPointSize(6.0)
89     glBegin(GL_POINTS)
90     glVertex2f(xc, yc)
91     glEnd()
92
93 def draw_grid(width, height, spacing=50):
94     glColor3f(0.15, 0.15, 0.15)
95     glLineWidth(1.0)
96
97     glBegin(GL_LINES)
98     # Vertical lines
99     for x in range(0, width, spacing):
100         glVertex2f(x, 0)
101         glVertex2f(x, height)
102
103     # Horizontal lines
104     for y in range(0, height, spacing):
105         glVertex2f(0, y)
106         glVertex2f(width, y)
107     glEnd()
108
109 def draw_bounding_box(xc, yc, rx, ry):
110     glColor3f(0.3, 0.3, 0.5)
111     glLineWidth(1.0)
112
113     glBegin(GL_LINE_LOOP)
114     glVertex2f(xc - rx, yc - ry)
115     glVertex2f(xc + rx, yc - ry)
116     glVertex2f(xc + rx, yc + ry)
117     glVertex2f(xc - rx, yc + ry)
118     glEnd()
119
120 def main():
121     if not glfw.init():
122         return
123
124     width, height = 1000, 800
125     window = glfw.create_window(width, height, "Mid-Point Ellipse Drawing
126     Algorithm", None, None)
127
128     if not window:
129         glfw.terminate()

```

```

129     return
130
131     glfw.make_context_current(window)
132
133     glViewport(0, 0, width, height)
134     glMatrixMode(GL_PROJECTION)
135     glLoadIdentity()
136     glOrtho(0, width, height, 0, -1, 1)
137     glMatrixMode(GL_MODELVIEW)
138     glLoadIdentity()
139
140     glEnable(GL_LINE_SMOOTH)
141     glEnable(GL_POINT_SMOOTH)
142     glEnable(GL_BLEND)
143     glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
144     glHint(GL_LINE_SMOOTH_HINT, GL_NICEST)
145     glHint(GL_POINT_SMOOTH_HINT, GL_NICEST)
146
147     glPointSize(2.0)
148
149     ellipses = [
150         # (center_x, center_y, radius_x, radius_y, color, name)
151         (250, 200, 150, 100, (1.0, 0.3, 0.3), "Horizontal Ellipse"),
152         (650, 200, 100, 150, (0.3, 1.0, 0.3), "Vertical Ellipse"),
153         (250, 550, 180, 120, (0.3, 0.5, 1.0), "Wide Ellipse"),
154         (650, 550, 120, 80, (1.0, 0.8, 0.2), "Narrow Ellipse"),
155         (500, 400, 100, 100, (1.0, 0.4, 0.8), "Circle (rx=ry)"),
156     ]
157
158     all_ellipse_data = []
159     for xc, yc, rx, ry, color, name in ellipses:
160         points = midpoint_ellipse(xc, yc, rx, ry)
161         all_ellipse_data.append({
162             'points': points,
163             'center': (xc, yc),
164             'radii': (rx, ry),
165             'color': color,
166             'name': name
167         })
168
169     while not glfw.window_should_close(window):
170         glClear(GL_COLOR_BUFFER_BIT)
171         glClearColor(0.05, 0.05, 0.1, 1.0)
172
173         draw_grid(width, height, spacing=50)
174
175         for ellipse in all_ellipse_data:
176             xc, yc = ellipse['center']
177             rx, ry = ellipse['radii']
178
179             draw_bounding_box(xc, yc, rx, ry)
180
181             draw_axes(xc, yc, rx, ry)
182
183             draw_points(ellipse['points'], ellipse['color'])
184
185         glfw.swap_buffers(window)
186         glfw.poll_events()

```

```
187
188     glfw.terminate()
189
190 if __name__ == "__main__":
191     main()
```

6.4 Output

The program displays multiple ellipses with different orientations:

- Red horizontal ellipse at (250, 200) with $r_x = 150, r_y = 100$
- Green vertical ellipse at (650, 200) with $r_x = 100, r_y = 150$
- Blue wide ellipse at (500, 550) with $r_x = 180, r_y = 120$

The algorithm efficiently handles both horizontal and vertical ellipses using two-region processing and 4-way symmetry.

7 Conclusion

This laboratory work successfully implemented six fundamental computer graphics algorithms using OpenGL and GLFW in Python. The following key learnings and observations were made:

7.1 Technical Achievements

1. **Line Drawing Algorithms:**
 - DDA algorithm provides simple implementation using floating-point arithmetic
 - Bresenham's algorithm offers superior efficiency with integer-only operations
 - Both algorithms successfully handle lines in all octants
2. **Circle and Ellipse Algorithms:**
 - Mid-point circle algorithm effectively uses 8-way symmetry
 - Mid-point ellipse algorithm handles two regions with different decision parameters
 - Both algorithms avoid expensive trigonometric calculations
3. **Data Visualization:**
 - Line graphs successfully normalize and display arbitrary datasets
 - Pie charts effectively represent proportional data distribution
 - Both visualizations demonstrate practical applications of basic algorithms

7.2 OpenGL Integration

The use of OpenGL and GLFW provided:

- Hardware-accelerated rendering capabilities
- Cross-platform compatibility
- Simple API for 2D graphics primitives
- Real-time visualization and interaction