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

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 $(round(x), round(y))$
    - Update $x = x + x_{inc}$ and $y = y + y_{inc}$

1.3 Source Code

```python
import glfw
from OpenGL.GL import *

def dda_line(x1, y1, x2, y2):
    points = []

    dx = x2 - x1
    dy = y2 - y1

    # Determine number of steps
    steps = max(abs(dx), abs(dy))

    if steps == 0:
        return [(x1, y1)]

    # Calculate increment for each step
    x_inc = dx / steps
    y_inc = dy / steps

    # Starting point
    x, y = x1, y1

    for _ in range(int(steps) + 1):
        points.append((round(x), round(y)))
        x += x_inc
        y += y_inc

    return points

def draw_line_points(points):
    glBegin(GL_POINTS)
    for x, y in points:
        glVertex2f(x, y)
    glEnd()

def main():
    if not glfw.init():
        return

    width, height = 800, 600
    window = glfw.create_window(width, height, "DDA Line Drawing Algorithm"
    , None, None)

    if not window:
        glfw.terminate()
        return

    glfw.make_context_current(window)

    glViewport(0, 0, width, height)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    glOrtho(0, width, height, 0, -1, 1)  # 2D orthographic projection
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()

    glPointSize(2.0)

    lines = [
        (100, 100, 700, 500),  # Diagonal line
        (400, 50, 400, 550),   # Vertical line
        (50, 300, 750, 300),   # Horizontal line
        (200, 150, 600, 450),  # Another diagonal
        (150, 500, 650, 100),  # Diagonal with negative slope
    ]

    all_points = []
    for x1, y1, x2, y2 in lines:
        points = dda_line(x1, y1, x2, y2)
        all_points.extend(points)

    while not glfw.window_should_close(window):
        glClear(GL_COLOR_BUFFER_BIT)
        glClearColor(0.0, 0.0, 0.0, 1.0)

        glColor3f(1.0, 1.0, 1.0)

        draw_line_points(all_points)

        glfw.swap_buffers(window)
        glfw.poll_events()

    glfw.terminate()

```

```
if __name__ == "__main__":
    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

```python
import glfw
from OpenGL.GL import *

def bresenham_line(x1, y1, x2, y2):
    points = []

    dx = abs(x2 - x1)
    dy = abs(y2 - y1)

    x_step = 1 if x2 > x1 else -1
    y_step = 1 if y2 > y1 else -1

    x, y = x1, y1

    # Case 1: |m| < 1
    if dx > dy:
        p = 2 * dy - dx  # Initial decision parameter

        for _ in range(dx + 1):
            points.append((x, y))

            if p >= 0:
                y += y_step
                p += 2 * (dy - dx)
            else:
                p += 2 * dy

            x += x_step

    # Case 2: |m| >= 1
    else:
        p = 2 * dx - dy  # Initial decision parameter

        for _ in range(dy + 1):
            points.append((x, y))

            if p >= 0:
                x += x_step
                p += 2 * (dx - dy)
            else:
                p += 2 * dx

            y += y_step

    return points

def draw_line_points(points, color=(1.0, 1.0, 1.0)):
    glColor3f(*color)
    glBegin(GL_POINTS)
    for x, y in points:
        glVertex2f(x, y)
    glEnd()

def main():
    if not glfw.init():
        return

    width, height = 800, 600
    window = glfw.create_window(width, height, "Bresenham Line Drawing
    Algorithm", None, None)

    if not window:
        glfw.terminate()
        return

    glfw.make_context_current(window)

    glViewport(0, 0, width, height)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    glOrtho(0, width, height, 0, -1, 1)  # 2D orthographic projection
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()

    glPointSize(3.0)

    lines = [
        #Gentle slopes (|m| < 1) - Red
        ((100, 300, 400, 350), (1.0, 0.0, 0.0)),
        ((100, 200, 400, 150), (1.0, 0.0, 0.0)),
        ((450, 300, 750, 300), (1.0, 0.0, 0.0)),

        # Steep slopes (|m| >= 1) - Green
        ((200, 100, 250, 500), (0.0, 1.0, 0.0)),
        ((350, 500, 400, 100), (0.0, 1.0, 0.0)),
        ((600, 100, 600, 500), (0.0, 1.0, 0.0)),

        # 45-degree lines (|m| = 1) - Blue
        ((100, 100, 300, 300), (0.0, 0.0, 1.0)),
        ((500, 100, 700, 300), (0.0, 0.0, 1.0)),

        # Mixed slopes - Yellow
        ((50, 450, 350, 100), (1.0, 1.0, 0.0)),
        ((450, 450, 750, 150), (1.0, 1.0, 0.0)),
    ]

    all_line_data = []
    for (x1, y1, x2, y2), color in lines:
        points = bresenham_line(x1, y1, x2, y2)
        all_line_data.append((points, color))

        dx = abs(x2 - x1)
        dy = abs(y2 - y1)

    while not glfw.window_should_close(window):
        glClear(GL_COLOR_BUFFER_BIT)
        glClearColor(0.1, 0.1, 0.1, 1.0)

        for points, color in all_line_data:
            draw_line_points(points, color)

        glfw.swap_buffers(window)
        glfw.poll_events()

    glfw.terminate()

if __name__ == "__main__":
    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

```python
import glfw
from OpenGL.GL import *

def plot_circle_points(xc, yc, x, y, points):
    points.extend([
        (xc + x, yc + y),  # Octant 1
        (xc - x, yc + y),  # Octant 4
        (xc + x, yc - y),  # Octant 8
        (xc - x, yc - y),  # Octant 5
        (xc + y, yc + x),  # Octant 2
        (xc - y, yc + x),  # Octant 3
        (xc + y, yc - x),  # Octant 7
        (xc - y, yc - x),  # Octant 6
    ])

def midpoint_circle(xc, yc, r):
    points = []

    x = 0
    y = r

    p = 1 - r

    plot_circle_points(xc, yc, x, y, points)

    # Iterate until x >= y
    while x < y:
        x += 1

        if p < 0:
            # Mid-point is inside the circle
            p += 2 * x + 1
        else:
            # Mid-point is outside the circle
            y -= 1
            p += 2 * (x - y) + 1

        # Plot points in all 8 octants
        plot_circle_points(xc, yc, x, y, points)

    return points

def draw_points(points, color=(1.0, 1.0, 1.0)):
    glColor3f(*color)
    glBegin(GL_POINTS)
    for x, y in points:
        glVertex2f(x, y)
    glEnd()

def draw_filled_circle(xc, yc, r, color=(1.0, 1.0, 1.0)):
    """
    Optional: Draw a filled circle using the mid-point algorithm
    by drawing horizontal lines between symmetric points
    """
    glColor4f(*color, 0.3)  # Semi-transparent fill

    x = 0
    y = r
    p = 1 - r

    glBegin(GL_LINES)
    for i in range(-r, r + 1):
        glVertex2f(xc - r, yc + i)
        glVertex2f(xc + r, yc + i)
    glEnd()

def main():
    if not glfw.init():
        return

    width, height = 800, 600
    window = glfw.create_window(width, height, "Mid-Point Circle Drawing
   Algorithm", None, None)

    if not window:
        glfw.terminate()
        return

    glfw.make_context_current(window)

    glViewport(0, 0, width, height)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    glOrtho(0, width, height, 0, -1, 1)  # 2D orthographic projection
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()

    glEnable(GL_BLEND)
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)

    glPointSize(2.0)

    # Define circles to draw
    circles = [
        # (center_x, center_y, radius, color)
        (200, 150, 80, (1.0, 0.0, 0.0)),     # Red circle
        (400, 300, 120, (0.0, 1.0, 0.0)),    # Green circle (center)
        (600, 150, 60, (0.0, 0.0, 1.0)),     # Blue circle
        (200, 450, 100, (1.0, 1.0, 0.0)),    # Yellow circle
        (600, 450, 90, (1.0, 0.0, 1.0)),     # Magenta circle
        (400, 150, 40, (0.0, 1.0, 1.0)),     # Cyan circle (small)
    ]

    # Calculate all circle points using mid-point algorithm
    all_circle_data = []
    for xc, yc, r, color in circles:
        points = midpoint_circle(xc, yc, r)
        all_circle_data.append((points, color))

        print(f"Circle at ({xc},{yc}) with radius {r}: {len(points)} points
    generated")

    while not glfw.window_should_close(window):
        glClear(GL_COLOR_BUFFER_BIT)
        glClearColor(0.05, 0.05, 0.05, 1.0)

        for points, color in all_circle_data:
            draw_points(points, color)

        glfw.swap_buffers(window)
        glfw.poll_events()

    glfw.terminate()

if __name__ == "__main__":
    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

```python
import glfw
from OpenGL.GL import *

def bresenham_line(x1, y1, x2, y2):
    points = []

    dx = abs(x2 - x1)
    dy = abs(y2 - y1)

    x_step = 1 if x2 > x1 else -1
    y_step = 1 if y2 > y1 else -1

    x, y = x1, y1

    if dx > dy:
        p = 2 * dy - dx
        for _ in range(dx + 1):
            points.append((x, y))
            if p >= 0:
                y += y_step
                p += 2 * (dy - dx)
            else:
                p += 2 * dy
            x += x_step
    else:
        p = 2 * dx - dy
        for _ in range(dy + 1):
            points.append((x, y))
            if p >= 0:
                x += x_step
                p += 2 * (dx - dy)
            else:
                p += 2 * dx
            y += y_step

    return points

def dda_line(x1, y1, x2, y2):
    points = []

    dx = x2 - x1
    dy = y2 - y1

    steps = max(abs(dx), abs(dy))

    if steps == 0:
        return [(x1, y1)]

    x_inc = dx / steps
    y_inc = dy / steps

    x, y = x1, y1

    for _ in range(int(steps) + 1):
        points.append((round(x), round(y)))
        x += x_inc
        y += y_inc

    return points

def normalize_data(data, width, height, margin=50):
    """
    Normalize data to fit within the window dimensions.

    Parameters:
    data: List of (x, y) tuples representing the dataset
    width, height: Window dimensions
    margin: Margin from window edges

    Returns: Normalized data points as pixel coordinates
    """
    if not data:
        return []

    # Find min and max values
    x_values = [point[0] for point in data]
    y_values = [point[1] for point in data]

    x_min, x_max = min(x_values), max(x_values)
    y_min, y_max = min(y_values), max(y_values)

    # Avoid division by zero
    x_range = x_max - x_min if x_max != x_min else 1
    y_range = y_max - y_min if y_max != y_min else 1

    # Available drawing area
    draw_width = width - 2 * margin
    draw_height = height - 2 * margin

    # Normalize to pixel coordinates
    normalized = []
    for x, y in data:
        # Map x from [x_min, x_max] to [margin, width-margin]
        pixel_x = margin + ((x - x_min) / x_range) * draw_width
        # Map y from [y_min, y_max] to [height-margin, margin] (inverted
   for screen coords)
        pixel_y = height - margin - ((y - y_min) / y_range) * draw_height
        normalized.append((int(pixel_x), int(pixel_y)))

    return normalized, (x_min, x_max, y_min, y_max)

def generate_graph_lines(data_points, algorithm='bresenham'):
    all_points = []

    line_func = bresenham_line if algorithm == 'bresenham' else dda_line

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

        line_points = line_func(x1, y1, x2, y2)
        all_points.extend(line_points)

    return all_points

def draw_points(points, color=(1.0, 1.0, 1.0)):
    glColor3f(*color)
    glBegin(GL_POINTS)
    for x, y in points:
        glVertex2f(x, y)
    glEnd()

def draw_axes(width, height, margin=50):
    glColor3f(0.5, 0.5, 0.5)
    glLineWidth(1.0)

    glBegin(GL_LINES)
    # X-axis
    glVertex2f(margin, height - margin)
    glVertex2f(width - margin, height - margin)

    # Y-axis
    glVertex2f(margin, margin)
    glVertex2f(margin, height - margin)
    glEnd()

def draw_data_points_markers(points, color=(1.0, 0.0, 0.0)):
    glColor3f(*color)
    glPointSize(8.0)
    glBegin(GL_POINTS)
    for x, y in points:
        glVertex2f(x, y)
    glEnd()

def draw_grid(width, height, margin, divisions=10):
    glColor3f(0.2, 0.2, 0.2)
    glLineWidth(1.0)

    draw_width = width - 2 * margin
    draw_height = height - 2 * margin

    glBegin(GL_LINES)
    # Vertical grid lines
    for i in range(divisions + 1):
        x = margin + (draw_width / divisions) * i
        glVertex2f(x, margin)
        glVertex2f(x, height - margin)

    # Horizontal grid lines
    for i in range(divisions + 1):
        y = margin + (draw_height / divisions) * i
        glVertex2f(margin, y)
        glVertex2f(width - margin, y)
    glEnd()

def main():
    if not glfw.init():
        return

    width, height = 1000, 700
    window = glfw.create_window(width, height, "Line Graph - DDA/Bresenham
   Algorithm", None, None)

    if not window:
        glfw.terminate()
        return

    glfw.make_context_current(window)

    glViewport(0, 0, width, height)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    glOrtho(0, width, height, 0, -1, 1)
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()

    # Sample datasets to visualize
    datasets = [
        {
            'name': 'Sales Data (Bresenham)',
            'data': [(1, 20), (2, 35), (3, 30), (4, 50), (5, 45), (6, 60),
   (7, 55), (8, 70), (9, 75), (10, 85)],
            'algorithm': 'bresenham',
            'color': (0.2, 0.8, 1.0)
        },
        {
            'name': 'Temperature Data (DDA)',
            'data': [(1, 15), (2, 18), (3, 22), (4, 25), (5, 28), (6, 32), (7, 30), (8, 27), (9, 23), (10, 20)],
            'algorithm': 'dda',
            'color': (1.0, 0.5, 0.2)
        }
    ]

    # 0 -> sales data (DDA), 1 -> temperature data (Bresenham)
    current_dataset = 0
    dataset = datasets[current_dataset]

    margin = 80

    # Normalize data to window coordinates
    normalized_points, bounds = normalize_data(dataset['data'], width, height, margin)

    # Generate line graph using specified algorithm
    graph_points = generate_graph_lines(normalized_points, dataset['algorithm'])

    for i, (x, y) in enumerate(dataset['data'], 1):
        print(f"  Point {i}: ({x}, {y})")

    glPointSize(2.0)

    while not glfw.window_should_close(window):
        glClear(GL_COLOR_BUFFER_BIT)
        glClearColor(0.05, 0.05, 0.1, 1.0)

        draw_grid(width, height, margin, divisions=10)

        draw_axes(width, height, margin)

        draw_points(graph_points, dataset['color'])

        draw_data_points_markers(normalized_points, (1.0, 0.0, 0.0))

        glfw.swap_buffers(window)
        glfw.poll_events()

    glfw.terminate()

if __name__ == "__main__":
    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

```python
import glfw
from OpenGL.GL import *
import math

def draw_filled_circle_sector(cx, cy, radius, start_angle, end_angle, color
    , segments=100):
    glColor3f(*color)
    glBegin(GL_TRIANGLE_FAN)

    glVertex2f(cx, cy)

    # Calculate number of segments for this sector
    angle_range = end_angle - start_angle
    sector_segments = max(int(segments * (angle_range / (2 * math.pi))), 2)

    # Draw vertices along the arc
    for i in range(sector_segments + 1):
        angle = start_angle + (angle_range * i / sector_segments)
        x = cx + radius * math.cos(angle)
        y = cy + radius * math.sin(angle)
        glVertex2f(x, y)

    glEnd()

def draw_circle_outline(cx, cy, radius, color=(1.0, 1.0, 1.0), segments
    =100):
    glColor3f(*color)
    glLineWidth(2.0)
    glBegin(GL_LINE_LOOP)

    for i in range(segments):
        angle = 2.0 * math.pi * i / segments
        x = cx + radius * math.cos(angle)
        y = cy + radius * math.sin(angle)
        glVertex2f(x, y)

    glEnd()

def draw_sector_outline(cx, cy, radius, start_angle, end_angle, color=(1.0,
     1.0, 1.0), segments=100):
    glColor3f(*color)
    glLineWidth(2.0)

    # Draw the arc
    glBegin(GL_LINE_STRIP)
    angle_range = end_angle - start_angle
    sector_segments = max(int(segments * (angle_range / (2 * math.pi))), 2)

    for i in range(sector_segments + 1):
        angle = start_angle + (angle_range * i / sector_segments)
        x = cx + radius * math.cos(angle)
        y = cy + radius * math.sin(angle)
        glVertex2f(x, y)

    glEnd()

    # Draw radial lines
    glBegin(GL_LINES)
    # Start radius
    glVertex2f(cx, cy)
    glVertex2f(cx + radius * math.cos(start_angle), cy + radius * math.sin(
    start_angle))
    # End radius
    glVertex2f(cx, cy)
    glVertex2f(cx + radius * math.cos(end_angle), cy + radius * math.sin(
    end_angle))
    glEnd()

def create_pie_chart(data, labels, colors, cx, cy, radius):
    total = sum(data)
    sectors = []

    current_angle = -math.pi / 2  # Start at top (12 o'clock position)

    for i, (value, label, color) in enumerate(zip(data, labels, colors)):
        percentage = (value / total) * 100
        angle_size = (value / total) * 2 * math.pi

```

```python
        sector_info = {
            'value': value,
            'percentage': percentage,
            'label': label,
            'color': color,
            'start_angle': current_angle,
            'end_angle': current_angle + angle_size,
            'mid_angle': current_angle + angle_size / 2
        }

        sectors.append(sector_info)
        current_angle += angle_size

    return sectors

def draw_pie_chart(sectors, cx, cy, radius):
    for sector in sectors:
        draw_filled_circle_sector(
            cx, cy, radius,
            sector['start_angle'],
            sector['end_angle'],
            sector['color']
        )

        # Draw outline for each sector
        draw_sector_outline(
            cx, cy, radius,
            sector['start_angle'],
            sector['end_angle'],
            (0.2, 0.2, 0.2)
        )

def draw_legend(sectors, x, y, box_size=20, spacing=30):
    for i, sector in enumerate(sectors):
        y_pos = y + i * spacing

        # Draw colored box
        glColor3f(*sector['color'])
        glBegin(GL_QUADS)
        glVertex2f(x, y_pos)
        glVertex2f(x + box_size, y_pos)
        glVertex2f(x + box_size, y_pos + box_size)
        glVertex2f(x, y_pos + box_size)
        glEnd()

        # Draw box outline
        glColor3f(1.0, 1.0, 1.0)
        glLineWidth(1.0)
        glBegin(GL_LINE_LOOP)
        glVertex2f(x, y_pos)
        glVertex2f(x + box_size, y_pos)
        glVertex2f(x + box_size, y_pos + box_size)
        glVertex2f(x, y_pos + box_size)
        glEnd()

def draw_labels_on_chart(sectors, cx, cy, radius):
    for sector in sectors:
        # Calculate position for label
        label_radius = radius * 0.6
        mid_angle = sector['mid_angle']

        label_x = cx + label_radius * math.cos(mid_angle)
        label_y = cy + label_radius * math.sin(mid_angle)

        # Draw a small circle at label position to show percentage location
        glColor3f(1.0, 1.0, 1.0)
        glPointSize(8.0)
        glBegin(GL_POINTS)
        glVertex2f(label_x, label_y)
        glEnd()

def main():
    if not glfw.init():
        return

    width, height = 1000, 700
    window = glfw.create_window(width, height, "Pie Chart - OpenGL Implementation", None, None)

    if not window:
        glfw.terminate()
        return

    glfw.make_context_current(window)

    glViewport(0, 0, width, height)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    glOrtho(0, width, height, 0, -1, 1)
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()

    glEnable(GL_LINE_SMOOTH)
    glEnable(GL_BLEND)
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
    glHint(GL_LINE_SMOOTH_HINT, GL_NICEST)

    data = [30, 45, 15, 60, 25]
    labels = ['Product A', 'Product B', 'Product C', 'Product D', 'Product E']
    colors = [
        (1.0, 0.2, 0.2),    # Red
        (0.2, 0.8, 0.2),    # Green
        (0.2, 0.4, 1.0),    # Blue
        (1.0, 0.8, 0.2),    # Yellow
        (1.0, 0.4, 0.8),    # Pink
    ]

    center_x = width // 2 - 100
    center_y = height // 2
    radius = 200

    sectors = create_pie_chart(data, labels, colors, center_x, center_y, radius)

    print("Pie Chart Data:")
    print("-" * 60)
    total = sum(data)
    for sector in sectors:
        print(f"{sector['label']:12s}: {sector['value']:6.1f} ({sector['
   percentage']:5.1f}%)")
    print("-" * 60)
    print(f"{'Total':12s}: {total:6.1f} (100.0%)")
    print("\nColors:")
    for sector in sectors:
        print(f"{sector['label']:12s}: RGB{sector['color']}")

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

        draw_pie_chart(sectors, center_x, center_y, radius)

        draw_labels_on_chart(sectors, center_x, center_y, radius)

        draw_legend(sectors, width - 250, 100)

        glColor3f(0.7, 0.7, 0.7)
        glLineWidth(2.0)
        glBegin(GL_LINES)
        glVertex2f(width // 2 - 100, 30)
        glVertex2f(width // 2 + 100, 30)
        glEnd()

        glfw.swap_buffers(window)
        glfw.poll_events()

    glfw.terminate()

if __name__ == "__main__":
    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 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 slope ≥ -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

```python
import glfw
from OpenGL.GL import *
import math

def plot_ellipse_points(xc, yc, x, y, points):
    points.extend([
        (xc + x, yc + y),   # Quadrant 1
        (xc - x, yc + y),   # Quadrant 2
        (xc - x, yc - y),   # Quadrant 3
        (xc + x, yc - y),   # Quadrant 4
    ])

def midpoint_ellipse(xc, yc, rx, ry):
    points = []

    # Region 1: Slope < -1
    x = 0
    y = ry

    # Initial decision parameters
    rx_sq = rx * rx
    ry_sq = ry * ry
    two_rx_sq = 2 * rx_sq
    two_ry_sq = 2 * ry_sq

    # Region 1 decision parameter
    p1 = ry_sq - (rx_sq * ry) + (0.25 * rx_sq)

    dx = two_ry_sq * x
    dy = two_rx_sq * y

    # Region 1: Continue while slope < -1
    while dx < dy:
        plot_ellipse_points(xc, yc, x, y, points)

        x += 1
        dx += two_ry_sq

        if p1 < 0:
            p1 += dx + ry_sq
        else:
            y -= 1
            dy -= two_rx_sq
            p1 += dx - dy + ry_sq

    # Region 2: Slope >= -1
    # Recalculate decision parameter for region 2
    p2 = ry_sq * (x + 0.5) * (x + 0.5) + rx_sq * (y - 1) * (y - 1) - rx_sq
   * ry_sq

    while y >= 0:
        plot_ellipse_points(xc, yc, x, y, points)

        y -= 1
        dy -= two_rx_sq

        if p2 > 0:
            p2 += rx_sq - dy
        else:
            x += 1
            dx += two_ry_sq
            p2 += dx - dy + rx_sq

    return points

def draw_points(points, color=(1.0, 1.0, 1.0)):
    glColor3f(*color)
    glBegin(GL_POINTS)
    for x, y in points:
        glVertex2f(x, y)
    glEnd()

```

```python
def draw_axes(xc, yc, rx, ry):
    glColor3f(0.4, 0.4, 0.4)
    glLineWidth(1.0)

    glBegin(GL_LINES)
    # Major axis (horizontal)
    glVertex2f(xc - rx - 20, yc)
    glVertex2f(xc + rx + 20, yc)

    # Minor axis (vertical)
    glVertex2f(xc, yc - ry - 20)
    glVertex2f(xc, yc + ry + 20)
    glEnd()

    # Draw center point
    glColor3f(1.0, 1.0, 0.0)
    glPointSize(6.0)
    glBegin(GL_POINTS)
    glVertex2f(xc, yc)
    glEnd()

def draw_grid(width, height, spacing=50):
    glColor3f(0.15, 0.15, 0.15)
    glLineWidth(1.0)

    glBegin(GL_LINES)
    # Vertical lines
    for x in range(0, width, spacing):
        glVertex2f(x, 0)
        glVertex2f(x, height)

    # Horizontal lines
    for y in range(0, height, spacing):
        glVertex2f(0, y)
        glVertex2f(width, y)
    glEnd()

def draw_bounding_box(xc, yc, rx, ry):
    glColor3f(0.3, 0.3, 0.5)
    glLineWidth(1.0)

    glBegin(GL_LINE_LOOP)
    glVertex2f(xc - rx, yc - ry)
    glVertex2f(xc + rx, yc - ry)
    glVertex2f(xc + rx, yc + ry)
    glVertex2f(xc - rx, yc + ry)
    glEnd()

def main():
    if not glfw.init():
        return

    width, height = 1000, 800
    window = glfw.create_window(width, height, "Mid-Point Ellipse Drawing
    Algorithm", None, None)

    if not window:
        glfw.terminate()
        return

    glfw.make_context_current(window)

    glViewport(0, 0, width, height)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    glOrtho(0, width, height, 0, -1, 1)
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()

    glEnable(GL_LINE_SMOOTH)
    glEnable(GL_POINT_SMOOTH)
    glEnable(GL_BLEND)
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
    glHint(GL_LINE_SMOOTH_HINT, GL_NICEST)
    glHint(GL_POINT_SMOOTH_HINT, GL_NICEST)

    glPointSize(2.0)

    ellipses = [
        # (center_x, center_y, radius_x, radius_y, color, name)
        (250, 200, 150, 100, (1.0, 0.3, 0.3), "Horizontal Ellipse"),
        (650, 200, 100, 150, (0.3, 1.0, 0.3), "Vertical Ellipse"),
        (250, 550, 180, 120, (0.3, 0.5, 1.0), "Wide Ellipse"),
        (650, 550, 120, 80, (1.0, 0.8, 0.2), "Narrow Ellipse"),
        (500, 400, 100, 100, (1.0, 0.4, 0.8), "Circle (rx=ry)"),
    ]

    all_ellipse_data = []
    for xc, yc, rx, ry, color, name in ellipses:
        points = midpoint_ellipse(xc, yc, rx, ry)
        all_ellipse_data.append({
            'points': points,
            'center': (xc, yc),
            'radii': (rx, ry),
            'color': color,
            'name': name
        })

    while not glfw.window_should_close(window):
        glClear(GL_COLOR_BUFFER_BIT)
        glClearColor(0.05, 0.05, 0.1, 1.0)

        draw_grid(width, height, spacing=50)

        for ellipse in all_ellipse_data:
            xc, yc = ellipse['center']
            rx, ry = ellipse['radii']

            draw_bounding_box(xc, yc, rx, ry)

            draw_axes(xc, yc, rx, ry)

            draw_points(ellipse['points'], ellipse['color'])

        glfw.swap_buffers(window)
        glfw.poll_events()

    glfw.terminate()

if __name__ == "__main__":
    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