MASTER OF COMPUTER APPLICATIONS NATIONAL INSTITUTE OF TECHNOLOGY KURUKSHETRA, HARYANA, INDIA

Computer Graphics & Multimedia Lab(MCA-233)

LAB RECORD (2025-2026)

SUBMITTED BY:
Name: Jayesh Solanki
Roll No: 524110055
Program: MCA
Semester: 3rd
Group: G-4, Serial No.05

SUBMITTED TO:
Dr. S. Suresh

Experiment 1: To implement DDA algorithm for line and circle.

The Digital Differential Analyzer (DDA) algorithm is a simple and efficient method used in computer graphics to draw lines and circles by calculating intermediate points between two endpoints or around a circular path.

1. DDA Line Drawing Algorithm

The DDA line algorithm works by incrementing one coordinate (x or y) in small equal steps and computing the other coordinate using the slope of the line.
The total number of steps is decided by the maximum difference between x and y directions.

The algorithm increases x and y by:

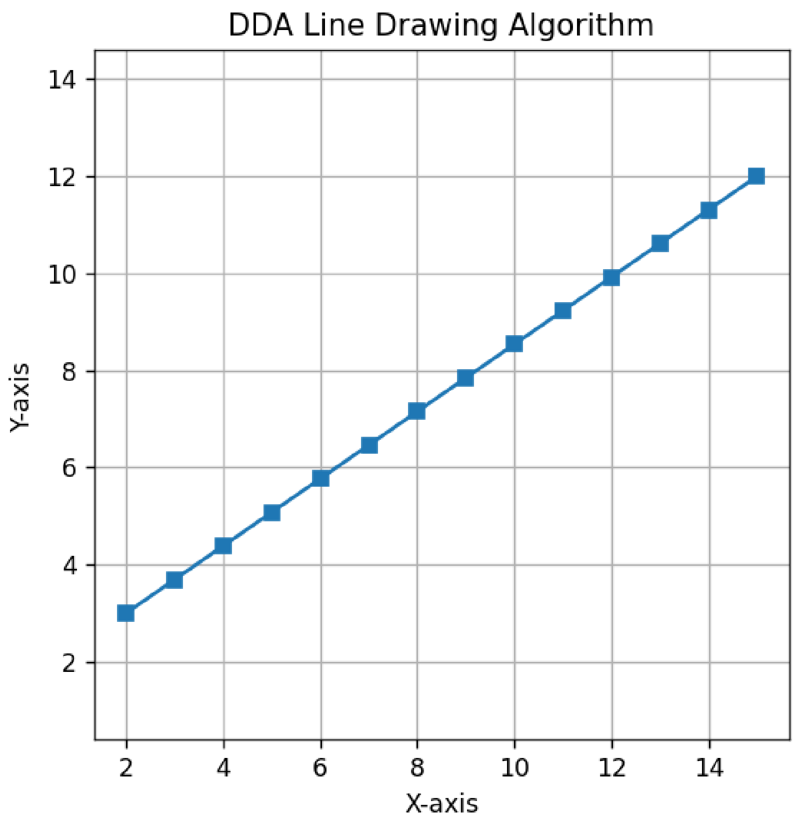
$$x_{\text{next}} = x + \Delta x/\text{steps}$$
$$y_{\text{next}} = y + \Delta y/\text{steps}$$

By plotting each calculated point, a smooth line is formed between the starting and ending points. DDA is simple to implement and provides a good approximation of a straight line.

Program Code (Python):

```
import matplotlib.pyplot as plt
def dda_line(x1, y1, x2, y2):
    x, y = x1, y1
    dx = x2 - x1
    dy = y2 - y1
    steps = int(max(abs(dx), abs(dy)))
    x_inc = dx / steps
    y_inc = dy / steps
    xs = []
    ys = []
    for _ in range(steps + 1):
        xs.append(x)
        ys.append(y)
        x += x_inc
        y += y_inc
    return xs, ys
x1, y1 = 2, 3
x2, y2 = 15, 12
xs, ys = dda_line(x1, y1, x2, y2)
plt.figure(figsize=(5, 5))
plt.plot(xs, ys, marker='s')
plt.title("DDA Line Drawing Algorithm")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.grid(True)
plt.axis("equal")
plt.show()
```

OUTPUT:


DDA Line Drawing Algorithm
Y-axis
X-axis
14
12
10
8
6
4
2
2
4
6
8
10
12
14

2. DDA Circle Drawing Algorithm

The DDA circle algorithm is based on the parametric equation of a circle. It assumes small angle increments and uses:

$$x = r\cos\ \theta, y = r\sin\ \theta$$

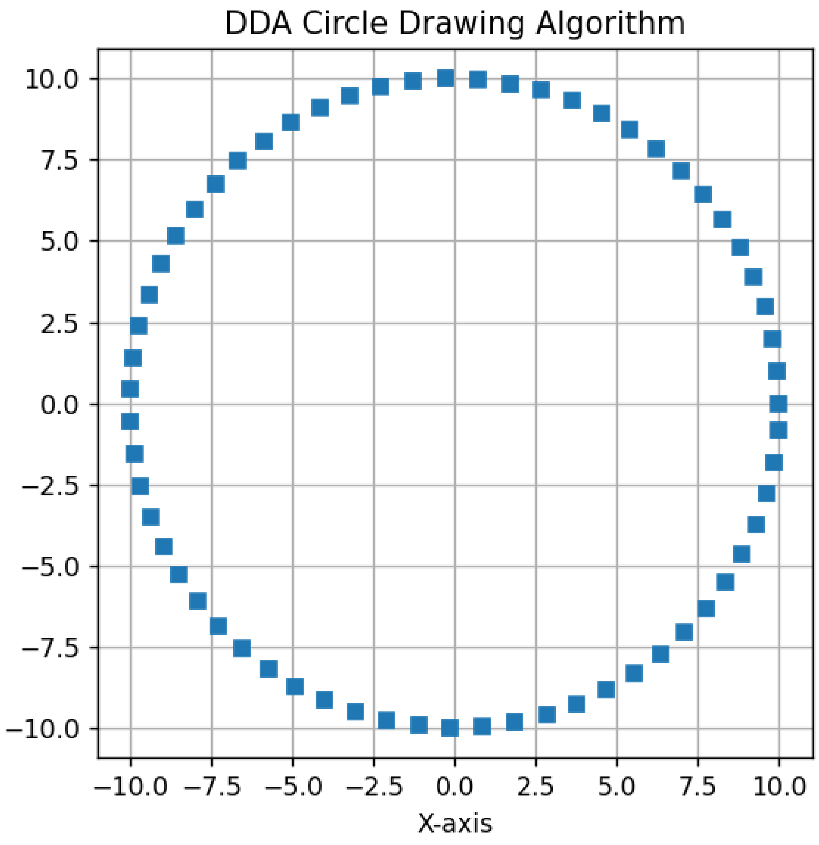
The center coordinates are added to shift the circle to the correct position.
By gradually increasing the angle θ in small fixed steps and computing the corresponding (x, y) values, the algorithm generates points along the circumference of the circle.

Plotting all these points in sequence results in a smooth circular shape.

Program Code (Python):

```
import matplotlib.pyplot as plt
import numpy as np

def dda_circle(xc, yc, r):
    xs = []
    ys = []
    # Step size = small angle increment (in radians)
    theta = 0
    step = 1 / r    # standard DDA circle step (small angle)

    while theta <= 2 * np.pi:
        x = xc + r * np.cos(theta)
        y = yc + r * np.sin(theta)
        xs.append(x)
        ys.append(y)
        theta += step

    return xs, ys

xc, yc = 0, 0   # center
r = 10          # radius

xs, ys = dda_circle(xc, yc, r)

plt.figure(figsize=(5, 5))
plt.plot(xs, ys, marker='s', linestyle="None")
plt.title("DDA Circle Drawing Algorithm")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.grid(True)
plt.axis("equal")
plt.show()
```

OUTPUT:


DDA Circle Drawing Algorithm
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
X-axis

Experiment 2: To implement Bresenham’s algorithms for line, circle and ellipse drawing

Bresenham’s algorithms are efficient raster graphics techniques used to draw lines, circles, and ellipses using only **integer arithmetic**. These algorithms avoid floating-point calculations, making them faster and ideal for computer graphics.

1. Bresenham’s Line Drawing Algorithm

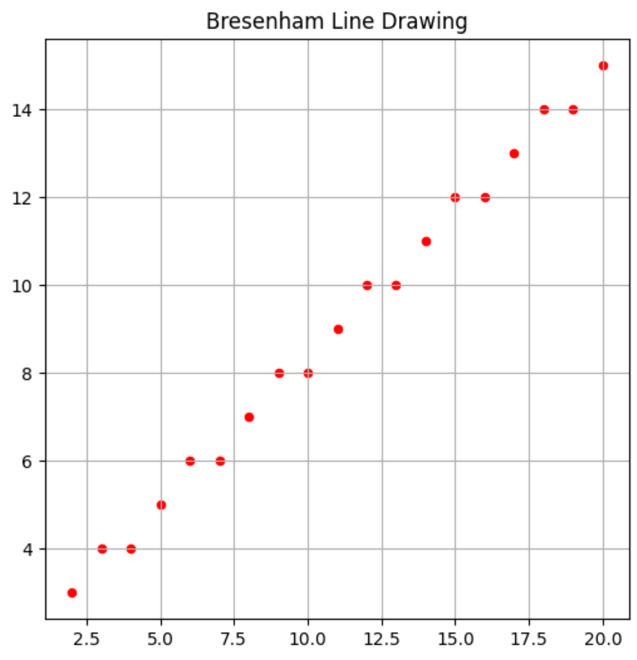
Bresenham’s line algorithm determines the closest pixel to the ideal mathematical line between two points.
It uses a **decision parameter** to choose between two possible pixels at each step.

The algorithm incrementally updates x (or y) and checks the decision variable to determine whether to move in the x-direction only or both x and y directions.
Because it uses only **addition, subtraction, and bit shifting**, it is one of the most efficient algorithms for raster line drawing.

Program Code (Python):

```python
import matplotlib.pyplot as plt
def bresenham_line(x1, y1, x2, y2):
    points = []
    dx = abs(x2 - x1)
    dy = abs(y2 - y1)
    sx = 1 if x1 < x2 else -1
    sy = 1 if y1 < y2 else -1
    err = dx - dy
    while True:
        points.append((x1, y1))
        if x1 == x2 and y1 == y2:
            break
        e2 = 2 * err
        if e2 > -dy:
            err -= dy
            x1 += sx
        if e2 < dx:
            err += dx
            y1 += sy
    return points
# Example usage
line_points = bresenham_line(2, 3, 20, 15)
x, y = zip(*line_points)
plt.figure(figsize=(6,6))
plt.scatter(x, y, color='red', s=20)
plt.title("Bresenham Line Drawing")
plt.grid(True)
plt.show()
```

OUTPUT:



2. Bresenham's Circle Drawing Algorithm

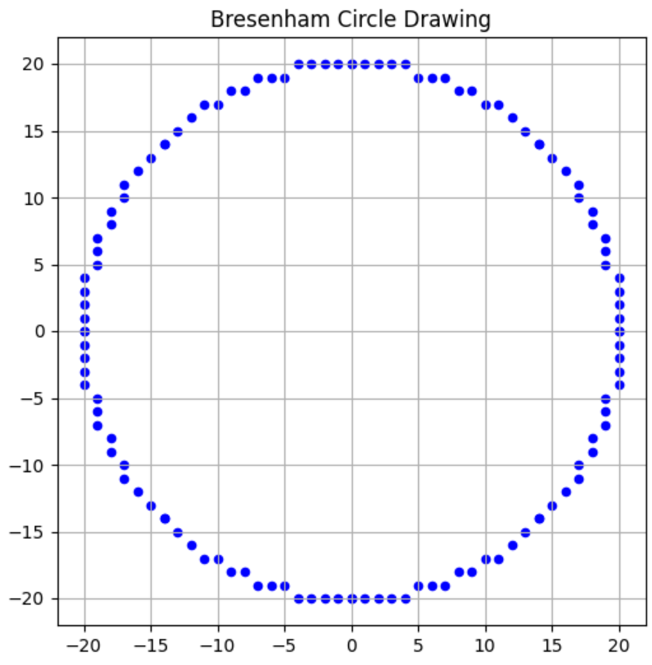
Bresenham's circle algorithm (also called Mid-point Circle Algorithm) is used to draw a circle by calculating only the first octant and then reflecting the points to the remaining seven octants using circle symmetry.

It uses a decision parameter to choose between two pixel positions at each step as the algorithm moves around the circle.
The method avoids trigonometric functions and relies only on integer addition and subtraction, making circle generation fast and accurate.

Program Code (Python):

```
import matplotlib.pyplot as plt
def bresenham_circle(xc, yc, r):
    x = 0
    y = r
    d = 3 - 2 * r
    points = []
    while x <= y:
        points.extend([
            (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)
        ])
        if d < 0:
            d += 4 * x + 6
        else:
            d += 4 * (x - y) + 10
            y -= 1
        x += 1
    return points
# Example usage
circle_points = bresenham_circle(0, 0, 20)
x, y = zip(*circle_points)
plt.figure(figsize=(6,6))
plt.scatter(x, y, color='blue', s=20)
plt.title("Bresenham Circle Drawing")
plt.gca().set_aspect('equal', adjustable='box')
plt.grid(True)
plt.show()
```

OUTPUT:



3. Bresenham's Ellipse Drawing Algorithm

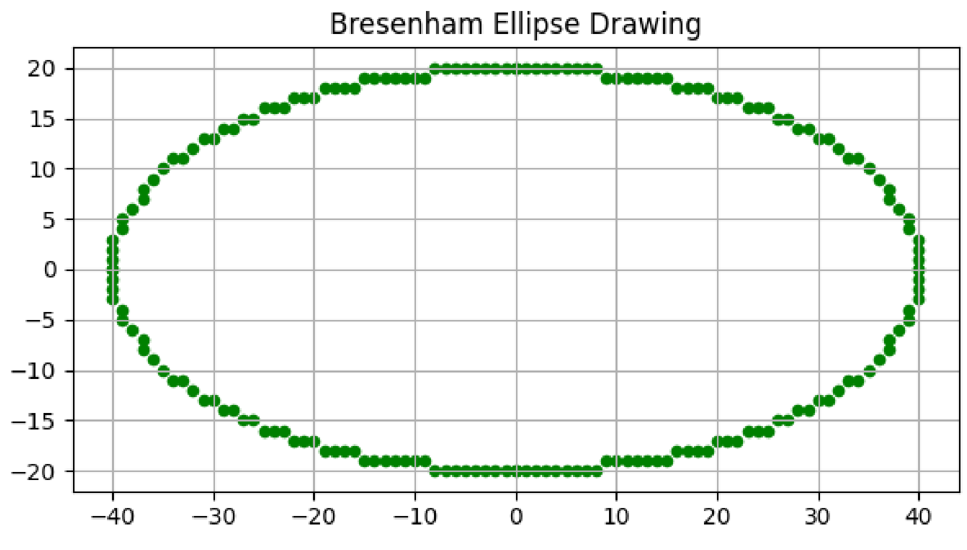
Bresenham's ellipse algorithm uses the symmetry of an ellipse to generate points in its first quadrant and then reflects them to complete the full ellipse.
The algorithm divides the ellipse into **two regions** based on slope and uses separate decision parameters to determine the next pixel in each region.

Like the line and circle algorithms, it uses only integer arithmetic and incremental error updates, providing a highly efficient method to draw ellipses in raster graphics.

Program Code (Python): Bresenham's Ellipse Drawing Algorithm

```
import matplotlib.pyplot as plt

def bresenham_ellipse(xc, yc, rx, ry):
    x = 0
    y = ry
    rx2 = rx * rx
    ry2 = ry * ry
    two_rx2 = 2 * rx2
    two_ry2 = 2 * ry2
    px = 0
    py = two_rx2 * y
    points = []

    # Region 1
    p = round(ry2 - (rx2 * ry) + (0.25 * rx2))
    while px < py:
        points.extend([
            (xc+x, yc+y), (xc-x, yc+y),
            (xc+x, yc-y), (xc-x, yc-y)
        ])
        x += 1
        px += two_ry2
        if p < 0:
            p += ry2 + px
        else:
            y -= 1
            py -= two_rx2
            p += ry2 + px - py

    # Region 2
    p = round(ry2 * (x + 0.5) ** 2 + rx2 * (y - 1) ** 2 - rx2 * ry2)
    while y >= 0:
        points.extend([
            (xc+x, yc+y), (xc-x, yc+y),
            (xc+x, yc-y), (xc-x, yc-y)
        ])
        y -= 1
        py -= two_rx2
        if p > 0:
            p += rx2 - py
        else:
            x += 1
            px += two_ry2
            p += rx2 - py + px
    return points

# Example usage
ellipse_points = bresenham_ellipse(0, 0, 40, 20)
x, y = zip(*ellipse_points)

plt.figure(figsize=(7,6))
plt.scatter(x, y, color='green', s=20)
plt.title("Bresenham Ellipse Drawing")
plt.gca().set_aspect('equal', adjustable='box')
plt.grid(True)
plt.show()
```

OUTPUT:



Experiment 3: To implement Mid-Point circle algorithm

The Mid-Point Circle Algorithm is an efficient raster graphics algorithm used to draw a circle by exploiting the **symmetry** of the circle and using **integer arithmetic only**. The algorithm calculates all the points of the circle's first octant and then mirrors these points into the remaining seven octants.

A circle with center $(x_c, y_c)$and radius $r$satisfies the equation:

$$x^2 + y^2 = r^2$$

The algorithm starts from the point $(0, r)$and moves outward by choosing between two possible next pixels at each step. A **decision parameter (d)** is used to determine which pixel is closer to the true circle.

The decision parameter is initialized as:

$$d = 1 - r$$

At each step:

- If $d < 0$, the next point is chosen horizontally.
- If $d \geq 0$, the next point is chosen diagonally, and the parameter is updated accordingly.

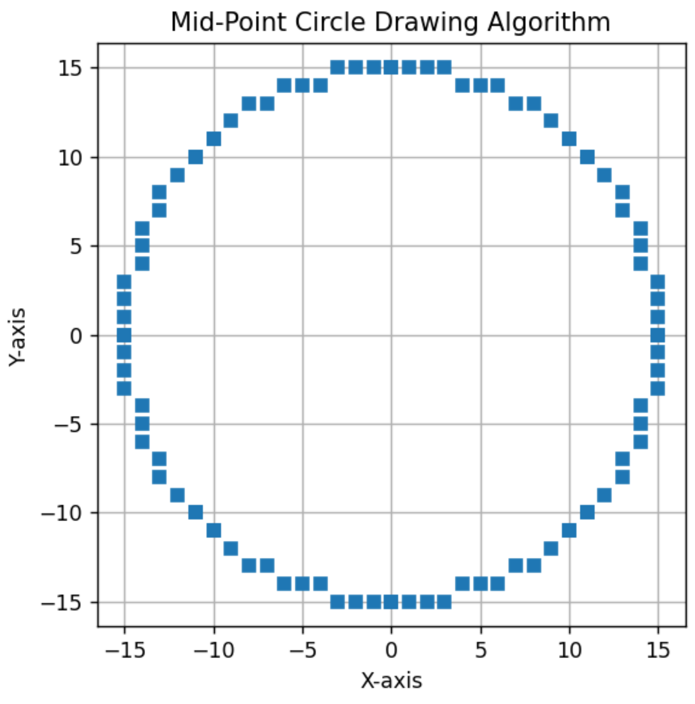
By incrementally updating x and y values and plotting symmetric points in all octants, the algorithm generates a smooth circle without using multiplication, division, or trigonometric functions.
This makes the Mid-Point Circle Algorithm fast and highly suitable for raster displays.

Program Code (Python):

```
import matplotlib.pyplot as plt

def midpoint_circle(xc, yc, r):
   x = 0
   y = r
   p = 1 - r  # initial decision parameter
   xs = []
   ys = []
   def plot_circle_points(xc, yc, x, y):
      # 8-way symmetry points
      xs.extend([xc + x, xc - x, xc + x, xc - x,
               xc + y, xc - y, xc + y, xc - y])
      ys.extend([yc + y, yc + y, yc - y, yc - y,
               yc + x, yc + x, yc - x, yc - x])
```

```
    # plot initial points
    plot_circle_points(xc, yc, x, y)

    while x < y:
        x += 1
        if p < 0:
            p = p + 2*x + 1
        else:
            y -= 1
            p = p + 2*(x - y) + 1
        plot_circle_points(xc, yc, x, y)
    return xs, ys

xc, yc = 0, 0   # center
r = 15          # radius
xs, ys = midpoint_circle(xc, yc, r)

plt.figure(figsize=(5, 5))
plt.plot(xs, ys, marker='s', linestyle='None')
plt.title("Mid-Point Circle Drawing Algorithm")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.grid(True)
plt.axis("equal")
plt.show()
```

OUTPUT:



Experiment 4: To implement Mid-Point ellipse algorithm

The Mid-Point Ellipse Algorithm is an efficient raster graphics algorithm used to draw an ellipse using **integer arithmetic** and **symmetry**. An ellipse with center (x_c, y_c), major axis a, and minor axis $b$satisfies the equation:

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

The algorithm uses the symmetry of the ellipse and calculates only the points in the **first quadrant**, then mirrors them into the remaining three quadrants.

The algorithm divides the first quadrant into **two regions**:

Region 1 (slope < –1)

- x is incremented by 1 each step.
- A **decision parameter** determines whether y decreases or remains the same.
- The point closest to the true ellipse is chosen based on the decision value.

Region 2 (slope > –1)

- y is decremented by 1 each step.
- The decision parameter decides whether x increases or stays the same.

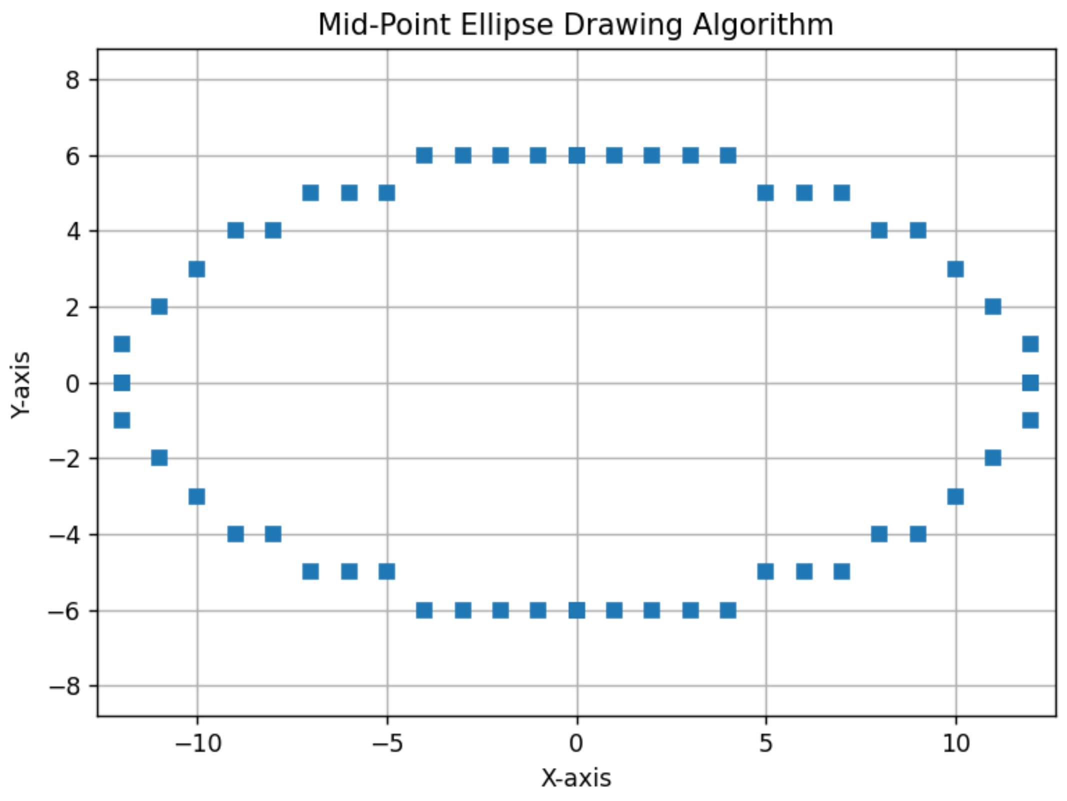
Initial decision parameters for each region are computed from the ellipse equation. At each step, the next point is selected using only addition and subtraction, making the algorithm efficient and fast.

By plotting the calculated point and reflecting it across both axes, the complete ellipse is drawn smoothly.

Program Code (Python):

```
import matplotlib.pyplot as plt

def midpoint_ellipse(cx, cy, rx, ry):
    xs = []
    ys = []
    # Initial position
    x = 0
    y = ry

    # Region 1 decision parameter
    p1 = ry**2 - (rx**2 * ry) + (0.25 * rx**2)

    dx = 2 * ry**2 * x
    dy = 2 * rx**2 * y
    while dx < dy:
        xs.extend([cx + x, cx - x, cx + x, cx - x])
        ys.extend([cy + y, cy + y, cy - y, cy - y])

        x += 1
        dx = 2 * ry**2 * x

        if p1 < 0:
            p1 = p1 + dx + ry**2
        else:
            y -= 1
            dy = 2 * rx**2 * y
            p1 = p1 + dx - dy + ry**2

    # Region 2 decision parameter
    p2 = ((ry**2) * ((x + 0.5)**2)) + ((rx**2) * ((y - 1)**2)) - (rx**2 * ry**2)
    while y >= 0:
        xs.extend([cx + x, cx - x, cx + x, cx - x])
        ys.extend([cy + y, cy + y, cy - y, cy - y])

        y -= 1
        dy = 2 * rx**2 * y

        if p2 > 0:
            p2 = p2 - dy + rx**2
        else:
            x += 1
            dx = 2 * ry**2 * x
            p2 = p2 + dx - dy + rx**2

    return xs, ys

cx, cy = 0, 0   # center of ellipse
rx, ry = 12, 6  # radii

xs, ys = midpoint_ellipse(cx, cy, rx, ry)

plt.figure(figsize=(6, 5))
plt.plot(xs, ys, marker='s', linestyle="None")
plt.title("Mid-Point Ellipse Drawing Algorithm")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.grid(True)
plt.axis("equal")
plt.show()
```

OUTPUT:



Mid-Point Ellipse Drawing Algorithm
Y-axis
X-axis
8
6
4
2
0
−2
−4
−6
−8
−10
−5
0
5
10

Experiment 5: To perform 2D Transformations such as translation, rotation, scaling, reflection and shearing

2D transformations are operations used in computer graphics to change the position, size, or orientation of objects in a two-dimensional plane. These transformations are represented using matrix operations, which make them easy to combine and apply to multiple points.

The main 2D transformations are:

1. Translation

Translation shifts an object from one position to another by adding a translation factor to its coordinates.

If the translation distances are $T_x$and T_y:

$$x' = x + T_x, y' = y + T_y$$

Translation moves the object without changing its shape or orientation.

2. Rotation

Rotation changes the orientation of an object around a fixed point, usually the origin.

For rotation by angle θ:

$$x' = x\cos\ \theta - y\sin\ \theta$$
$$y' = x\sin\ \theta + y\cos\ \theta$$

Rotation preserves the shape and size of the object.

3. Scaling

Scaling enlarges or shrinks an object by multiplying its coordinates with scaling factors $S_x$and S_y:

$$x' = x \cdot S_x, y' = y \cdot S_y$$

Uniform scaling keeps the proportions same, while non-uniform scaling distorts the object.

4. Reflection

Reflection creates a mirror image of an object about a line such as the x-axis, y-axis, or origin.

Examples:

- About x-axis: $(x', y') = (x, -y)$
- About y-axis: $(x', y') = (-x, y)$

Reflection changes orientation but keeps the shape and size unchanged.

5. Shearing

Shearing shifts one coordinate in proportion to the other, causing a slant or tilt.

For x-shear with factor Sh_x:

$$x' = x + y \cdot Sh_x, y' = y$$

For y-shear:

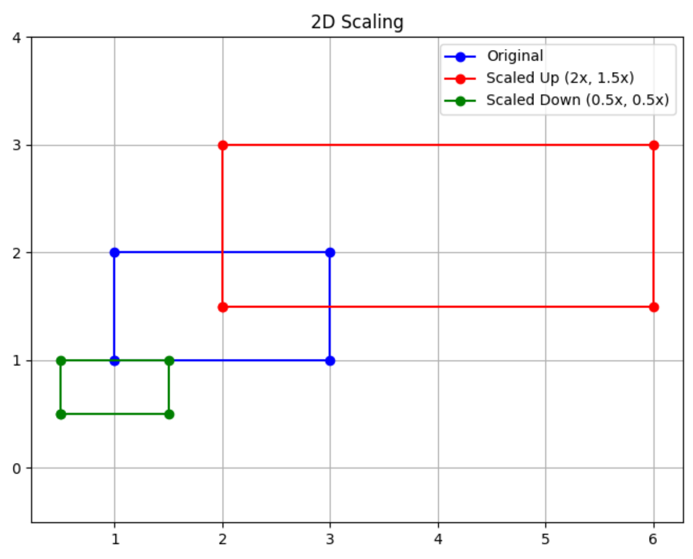
$$x' = x, y' = y + x \cdot Sh_y$$

Shearing changes the shape while keeping the area constant.

Program Code (Python):Scaling

```
import numpy as np
import matplotlib.pyplot as plt
def scale(points, sx, sy):
    """
    Scale points by sx and sy along x and y axes.
    """
    scaling_matrix = np.array([[sx, 0],
                               [0, sy]])
    return points.dot(scaling_matrix.T)

# Original points (a rectangle)
points = np.array([[1, 1], [3, 1], [3, 2], [1, 2], [1, 1]])  # closed rectangle

# Perform scaling
scaled_up = scale(points, sx=2, sy=1.5)     # scale up x by 2 and y by 1.5
scaled_down = scale(points, sx=0.5, sy=0.5) # scale down by half

# Plotting
plt.figure(figsize=(8, 6))
plt.axis('equal')

plt.plot(points[:, 0], points[:, 1], 'bo-', label='Original')
plt.plot(scaled_up[:, 0], scaled_up[:, 1], 'ro-', label='Scaled Up (2x, 1.5x)')
plt.plot(scaled_down[:, 0], scaled_down[:, 1], 'go-', label='Scaled Down (0.5x, 0.5x)')

plt.legend()
plt.title('2D Scaling')
plt.grid(True)
plt.show()
```

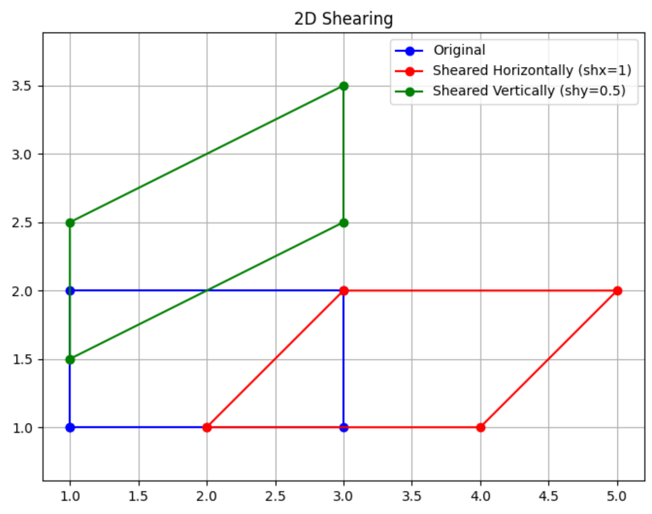
OUTPUT:



Program Code (Python): Shearing

```
import numpy as np
import matplotlib.pyplot as plt
def shear_x(points, shx):
    """
    Shear points horizontally by shx.
    """
    shear_matrix = np.array([[1, shx],
                             [0, 1]])
    return points.dot(shear_matrix.T)
def shear_y(points, shy):
    """
    Shear points vertically by shy.
    """
    shear_matrix = np.array([[1, 0],
                             [shy, 1]])
    return points.dot(shear_matrix.T)
# Original points (a rectangle)
points = np.array([[1, 1], [3, 1], [3, 2], [1, 2], [1, 1]])  # closed rectangle
# Apply shearing
sheared_x = shear_x(points, shx=1)  # shear horizontally by 1
sheared_y = shear_y(points, shy=0.5)  # shear vertically by 0.5
# Plotting
plt.figure(figsize=(8, 6))
plt.axis('equal')
plt.plot(points[:, 0], points[:, 1], 'bo-', label='Original')
plt.plot(sheared_x[:, 0], sheared_x[:, 1], 'ro-', label='Sheared Horizontally (shx=1)')
plt.plot(sheared_y[:, 0], sheared_y[:, 1], 'go-', label='Sheared Vertically (shy=0.5)')

plt.legend()
plt.title('2D Shearing')
plt.grid(True)
plt.show()
```

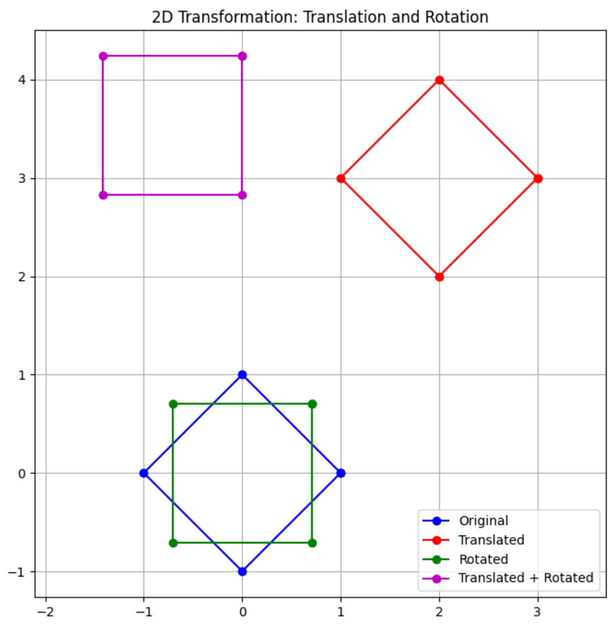
OUTPUT:



Program Code (Python): Rotation and translation

```
import numpy as np
import matplotlib.pyplot as plt
def translate(points, tx, ty):
    translation_vector = np.array([tx, ty])
    return points + translation_vector
def rotate(points, theta):
    rotation_matrix = np.array([
        [np.cos(theta), -np.sin(theta)],
        [np.sin(theta),  np.cos(theta)]
    ])
    return points.dot(rotation_matrix.T)
# Original points (a square around origin)
points = np.array([[1, 0], [0, 1], [-1, 0], [0, -1], [1, 0]])  # Close the shape by repeating first
point
# Perform transformations
translated_points = translate(points, tx=2, ty=3)
rotated_points = rotate(points, theta=np.pi/4)
translated_then_rotated = rotate(translated_points, theta=np.pi/4)
# Plotting
plt.figure(figsize=(8, 8))
plt.axis('equal')
# Original points
plt.plot(points[:, 0], points[:, 1], 'bo-', label='Original')
# Translated points
plt.plot(translated_points[:, 0], translated_points[:, 1], 'ro-', label='Translated')
# Rotated points
plt.plot(rotated_points[:, 0], rotated_points[:, 1], 'go-', label='Rotated')
# Translated then rotated points
plt.plot(translated_then_rotated[:, 0], translated_then_rotated[:, 1], 'mo-', label='Translated +
Rotated')
plt.legend()
plt.title('2D Transformation: Translation and Rotation')
plt.grid(True)plt.show()
```

OUTPUT:



Experiment 6: To implement Cohen–Sutherland 2D clipping and window–viewport mapping

1. Cohen–Sutherland Line Clipping Algorithm

The **Cohen–Sutherland algorithm** is a widely used line clipping method in computer graphics.
It determines whether a line segment is completely inside the clipping window, completely outside, or partially inside.

Each endpoint of the line segment is assigned a **4-bit region code** based on its position relative to the clipping window:

- Left
- Right
- Top
- Bottom

These bits indicate the point's location.
The algorithm works by:

1. **Trivial Acceptance**:
   If both endpoints have region code 0000, the line lies completely inside the window.
2. **Trivial Rejection**:
   If the logical AND of region codes ≠ 0, the line lies completely outside.
3. **Partial Clipping**:
   If neither accepted nor rejected, the line intersects the boundary.
   The algorithm finds intersection points with the window edges, replaces the outside endpoint with the intersection, and repeats the test.

This continues until the line becomes accepted or rejected.
The method is efficient because it uses bitwise operations.

2. Window–Viewport Mapping

Window–viewport mapping transforms coordinates from the **world window** (user-defined area) to the **viewport** (display area on screen).

The window is defined in world coordinate system:

$$(x_{w_{min}}, y_{w_{min}})\text{to}(x_{w_{max}}, y_{w_{max}})$$

The viewport is defined in device coordinate system:

$$(x_{v_{min}}, y_{v_{min}})\text{to}(x_{v_{max}}, y_{v_{max}})$$

The mapping uses scaling and translation to convert window coordinates (xw, yw) to viewport coordinates (xv, yv):

$$x_v = x_{v_{min}} + \frac{(x_w - x_{w_{min}})(x_{v_{max}} - x_{v_{min}})}{(x_{w_{max}} - x_{w_{min}})}$$
$$y_v = y_{v_{min}} + \frac{(y_w - y_{w_{min}})(y_{v_{max}} - y_{v_{min}})}{(y_{w_{max}} - y_{w_{min}})}$$

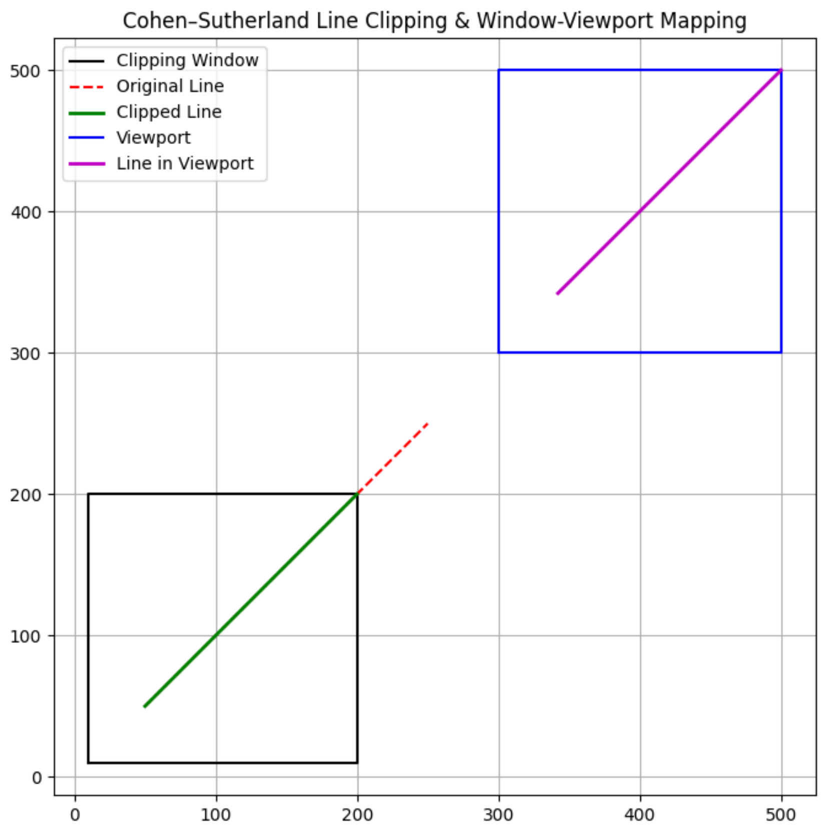
This ensures that shapes drawn in the window appear correctly scaled and positioned within the viewport.

Program Code (Python):

```
import matplotlib.pyplot as plt
# Region codes
INSIDE, LEFT, RIGHT, BOTTOM, TOP = 0, 1, 2, 4, 8
# Function to compute region code
def compute_code(x, y, x_min, y_min, x_max, y_max):
    code = INSIDE
    if x < x_min:  # to the left
        code |= LEFT
    elif x > x_max:  # to the right
        code |= RIGHT
    if y < y_min:  # below
        code |= BOTTOM
    elif y > y_max:  # above
        code |= TOP
    return code
# Cohen–Sutherland Line Clipping Algorithm
def cohen_sutherland_clip(x1, y1, x2, y2, x_min, y_min, x_max, y_max):
    code1 = compute_code(x1, y1, x_min, y_min, x_max, y_max)
    code2 = compute_code(x2, y2, x_min, y_min, x_max, y_max)
    accept = False
    while True:
        if code1 == 0 and code2 == 0:  # Trivially accepted
            accept = True
            break
        elif (code1 & code2) != 0:  # Trivially rejected
            break
        else:  # Needs clipping
            if code1 != 0:
                code_out = code1
            else:
                code_out = code2

            if code_out & TOP:
                x = x1 + (x2 - x1) * (y_max - y1) / (y2 - y1)
                y = y_max
            elif code_out & BOTTOM:
                x = x1 + (x2 - x1) * (y_min - y1) / (y2 - y1)
                y = y_min
            elif code_out & RIGHT:
                y = y1 + (y2 - y1) * (x_max - x1) / (x2 - x1)
                x = x_max
            elif code_out & LEFT:
                y = y1 + (y2 - y1) * (x_min - x1) / (x2 - x1)
                x = x_min

            if code_out == code1:
                x1, y1 = x, y
                code1 = compute_code(x1, y1, x_min, y_min, x_max, y_max)
            else:
                x2, y2 = x, y
                code2 = compute_code(x2, y2, x_min, y_min, x_max, y_max)

    if accept:
        return (x1, y1, x2, y2)
    else:
        return None
# Window to viewport mapping
def window_to_viewport(x, y, xw_min, yw_min, xw_max, yw_max, xv_min, yv_min,
xv_max, yv_max):
    sx = (xv_max - xv_min) / (xw_max - xw_min)
    sy = (yv_max - yv_min) / (yw_max - yw_min)

    xv = xv_min + (x - xw_min) * sx
    yv = yv_min + (y - yw_min) * sy
    return xv, yv
# Example: window and viewport
xw_min, yw_min, xw_max, yw_max = 10, 10, 200, 200   # clipping window
xv_min, yv_min, xv_max, yv_max = 300, 300, 500, 500 # viewport
# Original line
x1, y1, x2, y2 = 50, 50, 250, 250
# Perform clipping
clipped_line = cohen_sutherland_clip(x1, y1, x2, y2, xw_min, yw_min, xw_max, yw_max)

plt.figure(figsize=(8,8))
plt.title("Cohen–Sutherland Line Clipping & Window-Viewport Mapping")
plt.axis("equal")
plt.grid(True)
# Draw clipping window
plt.plot([xw_min, xw_max, xw_max, xw_min, xw_min],
        [yw_min, yw_min, yw_max, yw_max, yw_min],
        'k-', label="Clipping Window")
# Original line
plt.plot([x1, x2], [y1, y2], 'r--', label="Original Line")
```

```
# Clipped line inside window
if clipped_line:
    cx1, cy1, cx2, cy2 = clipped_line
    plt.plot([cx1, cx2], [cy1, cy2], 'g-', linewidth=2, label="Clipped Line")

    # Map clipped line to viewport
    vx1, vy1 = window_to_viewport(cx1, cy1, xw_min, yw_min, xw_max, yw_max, xv_min,
yv_min, xv_max, yv_max)
    vx2, vy2 = window_to_viewport(cx2, cy2, xw_min, yw_min, xw_max, yw_max, xv_min,
yv_min, xv_max, yv_max)
    # Draw viewport
    plt.plot([xv_min, xv_max, xv_max, xv_min, xv_min],
             [yv_min, yv_min, yv_max, yv_max, yv_min],
             'b-', label="Viewport")
    # Line in viewport
    plt.plot([vx1, vx2], [vy1, vy2], 'm-', linewidth=2, label="Line in Viewport")
plt.legend()
plt.show()
```

Output:



Experiment 7: To implement Liang Barsky line clipping algorithm

The **Liang–Barsky algorithm** is an efficient line clipping method used in computer graphics to clip a line segment against a rectangular clipping window. It is based on the **parametric equation of a line** and uses inequalities to determine the visible portion of the line.

A line segment between points $(x_1, y_1)$and (x_2, y_2)can be represented parametrically as:

$$x = x_1 + u(x_2 - x_1)$$
$$y = y_1 + u(y_2 - y_1)$$

where $u$varies from 0 to 1.

The clipping window is defined by:

- Left: x_{min}
- Right: x_{max}
- Bottom: y_{min}
- Top: y_{max}

The algorithm expresses the clipping conditions as:

$$p_i u \leq q_i$$

where

- p_i= direction of the line relative to the boundary,
- q_i= distance from the point to the boundary.

For each of the four boundaries:

Boundary	p	q
Left	$-dx$	$x_1 - x_{min}$
Right	dx	$x_{max} - x_1$
Bottom	$-dy$	$y_1 - y_{min}$
Top	dy	$y_{max} - y_1$

The algorithm calculates **u** enter and **u** leave:

- If $p_i < 0$: update **entering** value
- If $p_i > 0$: update **leaving** value

- If $p_i = 0$and $q_i < 0$: line is completely outside

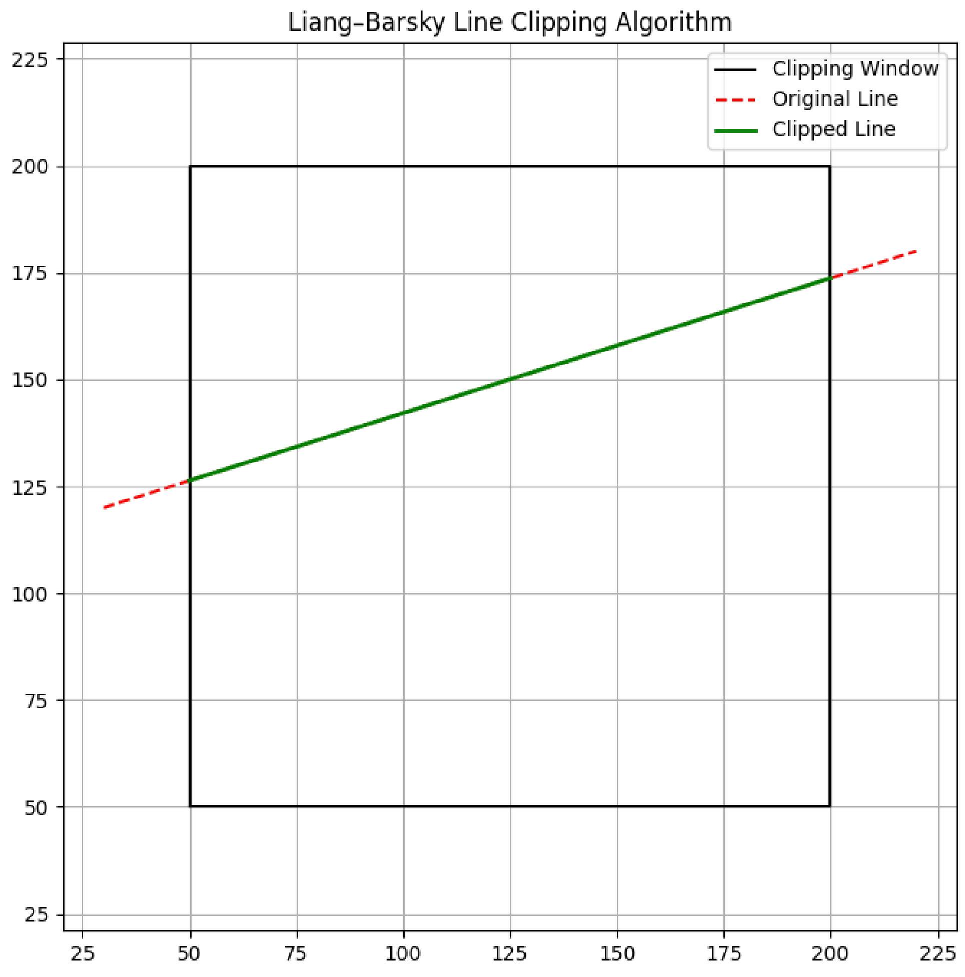
If $u_{\text{enter}} \leq u_{\text{leave}}$, the line segment between these parametric values is inside the window and is drawn.

The Liang–Barsky algorithm is more efficient than Cohen–Sutherland because it uses fewer intersection calculations and avoids repeated clipping.

Program Code (Python):

```
import matplotlib.pyplot as plt
def liang_barsky(x1, y1, x2, y2, x_min, y_min, x_max, y_max):
    dx = x2 - x1
    dy = y2 - y1
    p = [-dx, dx, -dy, dy]
    q = [x1 - x_min, x_max - x1, y1 - y_min, y_max - y1]
    u1, u2 = 0.0, 1.0
    for i in range(4):
        if p[i] == 0:  # Line is parallel
            if q[i] < 0:
                return None  # Line is outside
        else:
            u = q[i] / p[i]
            if p[i] < 0:
                u1 = max(u1, u)  # entering
            else:
                u2 = min(u2, u)  # leaving

    if u1 > u2:
        return None
    cx1 = x1 + u1 * dx
    cy1 = y1 + u1 * dy
    cx2 = x1 + u2 * dx
    cy2 = y1 + u2 * dy
    return (cx1, cy1, cx2, cy2)
# Example: clipping window
x_min, y_min, x_max, y_max = 50, 50, 200, 200
# Line to be clipped
x1, y1, x2, y2 = 30, 120, 220, 180

# Perform Liang–Barsky clipping
clipped_line = liang_barsky(x1, y1, x2, y2, x_min, y_min, x_max, y_max)

plt.figure(figsize=(8,8))
plt.title("Liang–Barsky Line Clipping Algorithm")
plt.axis("equal")
plt.grid(True)
# Draw clipping window
plt.plot([x_min, x_max, x_max, x_min, x_min],
          [y_min, y_min, y_max, y_max, y_min],
          'k-', label="Clipping Window")
# Draw original line
plt.plot([x1, x2], [y1, y2], 'r--', label="Original Line")
# Draw clipped line
if clipped_line:
    cx1, cy1, cx2, cy2 = clipped_line
    plt.plot([cx1, cx2], [cy1, cy2], 'g-', linewidth=2, label="Clipped Line")
plt.legend()
plt.show()
```

OUTPUT:



Experiment 8: To perform 3D Transformations such as translation, rotation and scaling

3D transformations are used in computer graphics to change the position, size, and orientation of objects in a three-dimensional space. These transformations are represented using **4×4 homogeneous transformation matrices**, which allow multiple transformations to be easily combined.

The main 3D transformations are:

1. Translation

Translation moves an object from one location to another along the x, y, and z axes.

If the translation distances are T_x, T_y, and T_z, the new coordinates are:

$$x' = x + T_x, y' = y + T_y, z' = z + T_z$$

Matrix form:

$$\begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Translation preserves the object's shape and orientation.

2. Rotation

Rotation changes the orientation of an object about one of the coordinate axes (x, y, or z). For an angle θ:

Rotation about X-axis:

$$\begin{aligned} x' &= x \\ y' &= y\cos\,\theta - z\sin\,\theta \\ z' &= y\sin\,\theta + z\cos\,\theta \end{aligned}$$

Rotation about Y-axis:

$$\begin{aligned} x' &= x\cos\,\theta + z\sin\,\theta \\ y' &= y \\ z' &= -x\sin\,\theta + z\cos\,\theta \end{aligned}$$

Rotation about Z-axis:

$$\begin{aligned} x' &= x\cos\,\theta - y\sin\,\theta \\ y' &= x\sin\,\theta + y\cos\,\theta \\ z' &= z \end{aligned}$$

Rotation preserves shape but changes orientation.

3. Scaling

Scaling enlarges or shrinks an object along the x, y, and z directions.

If the scaling factors are S_x, S_y, and S_z:

$$x' = x \cdot S_x, y' = y \cdot S_y, z' = z \cdot S_z$$

Matrix form:

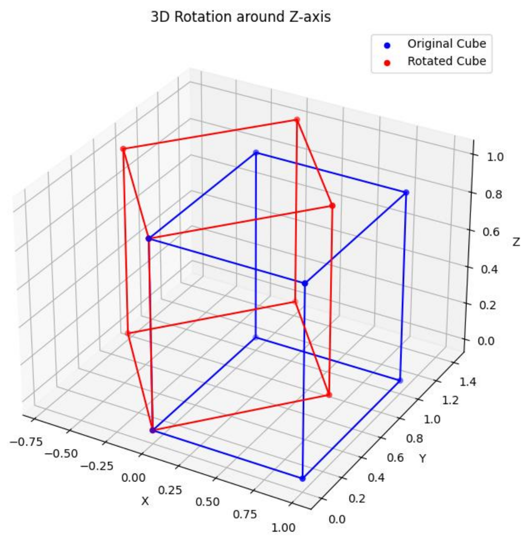
$$\begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Uniform scaling preserves proportions; non-uniform scaling distorts the object.

Program Code (Python):Rotation

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D  # for 3D plotting
def rotate_3d_z(points, theta):
    """
    Rotate 3D points around Z-axis by angle theta (radians).
    points: numpy array of shape (n_points, 3)
    theta: rotation angle in radians
    """
    rotation_matrix = np.array([
        [np.cos(theta), -np.sin(theta), 0],
        [np.sin(theta),  np.cos(theta), 0],
        [0,              0,             1]
    ])
    return points.dot(rotation_matrix.T)
# Cube corner points
points = np.array([
    [0, 0, 0],
    [1, 0, 0],
    [1, 1, 0],
    [0, 1, 0],
    [0, 0, 1],
    [1, 0, 1],
    [1, 1, 1],
    [0, 1, 1]
])
# Rotate cube 45 degrees around Z-axis
theta = np.pi / 4  # 45 degrees in radians
rotated_points = rotate_3d_z(points, theta)
# Plotting
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

# Function to draw cube edges
def draw_cube(ax, pts, color, label):
    edges = [
        (0,1),(1,2),(2,3),(3,0),  # bottom face
        (4,5),(5,6),(6,7),(7,4),  # top face
        (0,4),(1,5),(2,6),(3,7)   # vertical edges
    ]
    for edge in edges:
        p1, p2 = pts[edge[0]], pts[edge[1]]
        ax.plot([p1[0], p2[0]], [p1[1], p2[1]], [p1[2], p2[2]], color=color)
    ax.scatter(pts[:,0], pts[:,1], pts[:,2], color=color, label=label)

# Draw original and rotated cubes
draw_cube(ax, points, 'blue', 'Original Cube')
draw_cube(ax, rotated_points, 'red', 'Rotated Cube')

ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_title('3D Rotation around Z-axis')
ax.legend()
plt.show()
```

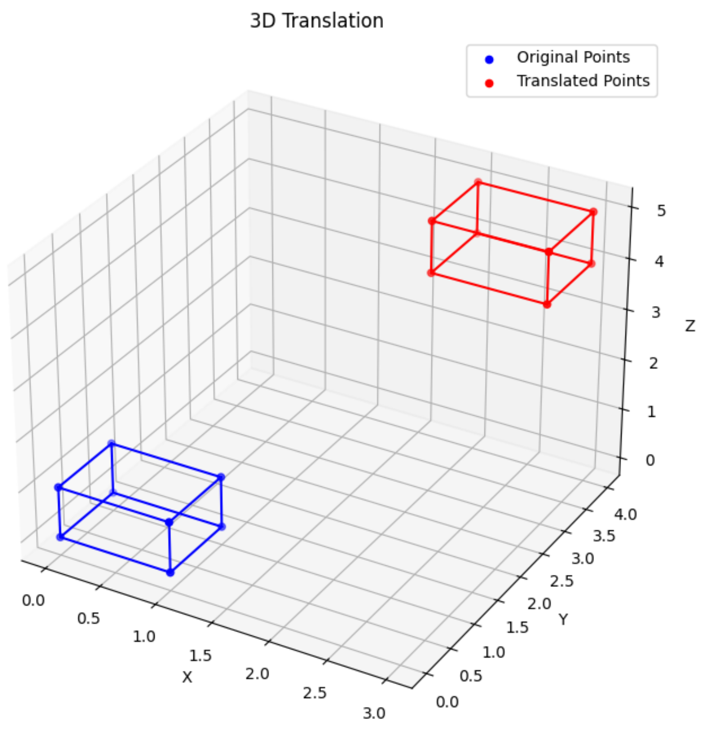
OUTPUT:



Program Code (Python):translation

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D  # needed for 3D plotting
def translate_3d(points, tx, ty, tz):
    """
    Translate 3D points by (tx, ty, tz).
    points: numpy array of shape (n_points, 3)
    tx, ty, tz: translation distances
    """
    translation_vector = np.array([tx, ty, tz])
    return points + translation_vector
# Original 3D points (a cube corners)
points = np.array([
    [0, 0, 0],
    [1, 0, 0],
    [1, 1, 0],
    [0, 1, 0],
    [0, 0, 1],
    [1, 0, 1],
    [1, 1, 1],
    [0, 1, 1]
])
# Translate by (2, 3, 4)
translated_points = translate_3d(points, tx=2, ty=3, tz=4)
# Plotting
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
# Plot original points
ax.scatter(points[:, 0], points[:, 1], points[:, 2], color='blue', label='Original Points')

# Plot translated points
ax.scatter(translated_points[:, 0], translated_points[:, 1], translated_points[:, 2], color='red',
label='Translated Points')

# Optionally connect points to form cube edges
def draw_cube(ax, pts, color):
    edges = [
        (0,1),(1,2),(2,3),(3,0),  # bottom face
        (4,5),(5,6),(6,7),(7,4),  # top face
        (0,4),(1,5),(2,6),(3,7)   # vertical edges
    ]
    for edge in edges:
        p1, p2 = pts[edge[0]], pts[edge[1]]
        ax.plot([p1[0], p2[0]], [p1[1], p2[1]], [p1[2], p2[2]], color=color)
draw_cube(ax, points, 'blue')
draw_cube(ax, translated_points, 'red')
ax.legend()
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_title('3D Translation')
plt.show()
```

OUTPUT:

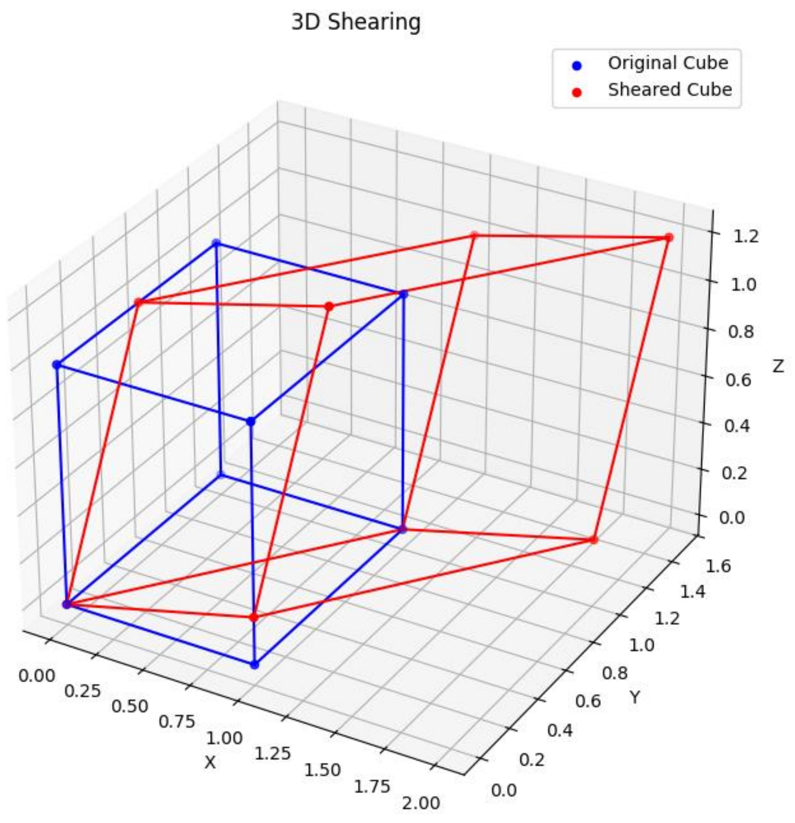


Program Code (Python): Shearing

```
import numpy as np
import matplotlib.pyplot as plt

def shear_3d(points, shxy=0, shxz=0, shyx=0, shyz=0, shzx=0, shzy=0):
    """
    Shear 3D points with given shear factors.

    Parameters:
    - shxy: shear of x relative to y
    - shxz: shear of x relative to z
    - shyx: shear of y relative to x
    - shyz: shear of y relative to z
    - shzx: shear of z relative to x
    - shzy: shear of z relative to y
    """
    shear_matrix = np.array([
        [1,   shxy, shxz],
        [shyx, 1,   shyz],
        [shzx, shzy, 1  ]
    ])
    return points.dot(shear_matrix.T)

# Cube corner points
points = np.array([
    [0, 0, 0],
    [1, 0, 0],
    [1, 1, 0],
    [0, 1, 0],
    [0, 0, 1],
    [1, 0, 1],
    [1, 1, 1],
    [0, 1, 1]
])
# Apply 3D shearing: example shear factors
sheared_points = shear_3d(points, shxy=1.0, shyz=0.5, shzx=0.2)
# Plotting
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
def draw_cube(ax, pts, color, label):
    edges = [
        (0,1),(1,2),(2,3),(3,0),  # bottom face
        (4,5),(5,6),(6,7),(7,4),  # top face
        (0,4),(1,5),(2,6),(3,7)   # vertical edges
    ]
    for edge in edges:
        p1, p2 = pts[edge[0]], pts[edge[1]]
        ax.plot([p1[0], p2[0]], [p1[1], p2[1]], [p1[2], p2[2]], color=color)
    ax.scatter(pts[:,0], pts[:,1], pts[:,2], color=color, label=label)
```

```
# Draw original and sheared cubes
draw_cube(ax, points, 'blue', 'Original Cube')
draw_cube(ax, sheared_points, 'red', 'Sheared Cube')

ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_title('3D Shearing')
ax.legend()
plt.show()
```

OUTPUT:

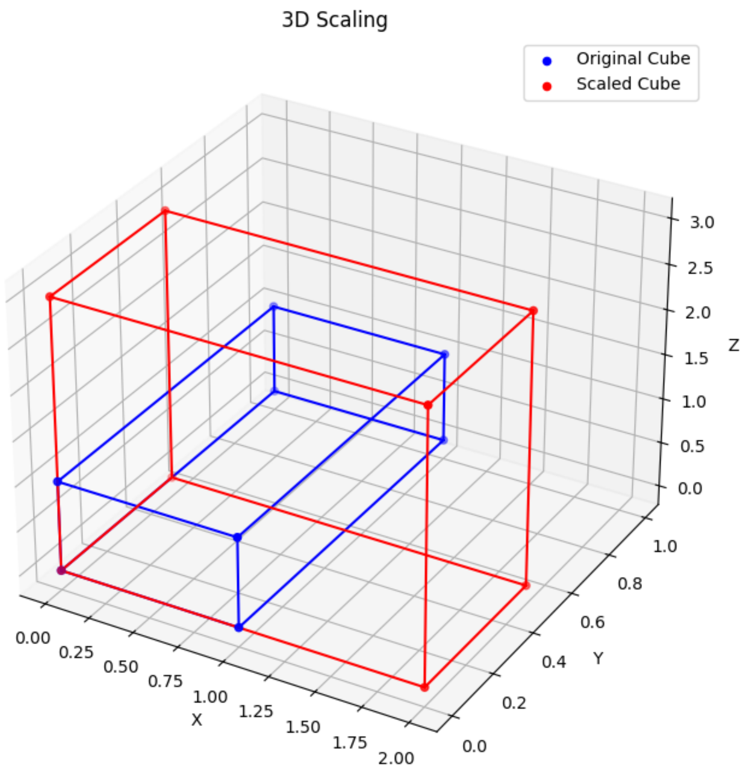


Program Code (Python): Scaling

```
import numpy as np
import matplotlib.pyplot as plt

def scale_3d(points, sx, sy, sz):
    """
    Scale 3D points by sx, sy, sz along each axis.

    points: numpy array of shape (n_points, 3)
    sx, sy, sz: scaling factors
    """
    scaling_matrix = np.array([
        [sx, 0, 0],
        [0, sy, 0],
        [0, 0, sz]
    ])
    return points.dot(scaling_matrix.T)

# Cube corner points
points = np.array([
    [0, 0, 0],
    [1, 0, 0],
    [1, 1, 0],
    [0, 1, 0],
    [0, 0, 1],
    [1, 0, 1],
    [1, 1, 1],
    [0, 1, 1]
])

# Scale factors (e.g., double x, half y, triple z)
scaled_points = scale_3d(points, sx=2, sy=0.5, sz=3)

# Plotting
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

def draw_cube(ax, pts, color, label):
    edges = [
        (0,1),(1,2),(2,3),(3,0),  # bottom face
        (4,5),(5,6),(6,7),(7,4),  # top face
        (0,4),(1,5),(2,6),(3,7)   # vertical edges
    ]
    for edge in edges:
        p1, p2 = pts[edge[0]], pts[edge[1]]
        ax.plot([p1[0], p2[0]], [p1[1], p2[1]], [p1[2], p2[2]], color=color)
    ax.scatter(pts[:,0], pts[:,1], pts[:,2], color=color, label=label)

# Draw original and scaled cubes
```

```
draw_cube(ax, points, 'blue', 'Original Cube')
draw_cube(ax, scaled_points, 'red', 'Scaled Cube')

ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_title('3D Scaling')
ax.legend()
plt.show()
```

Output:



Experiment 9: To draw different shapes such as hut, face, kite, fish etc.

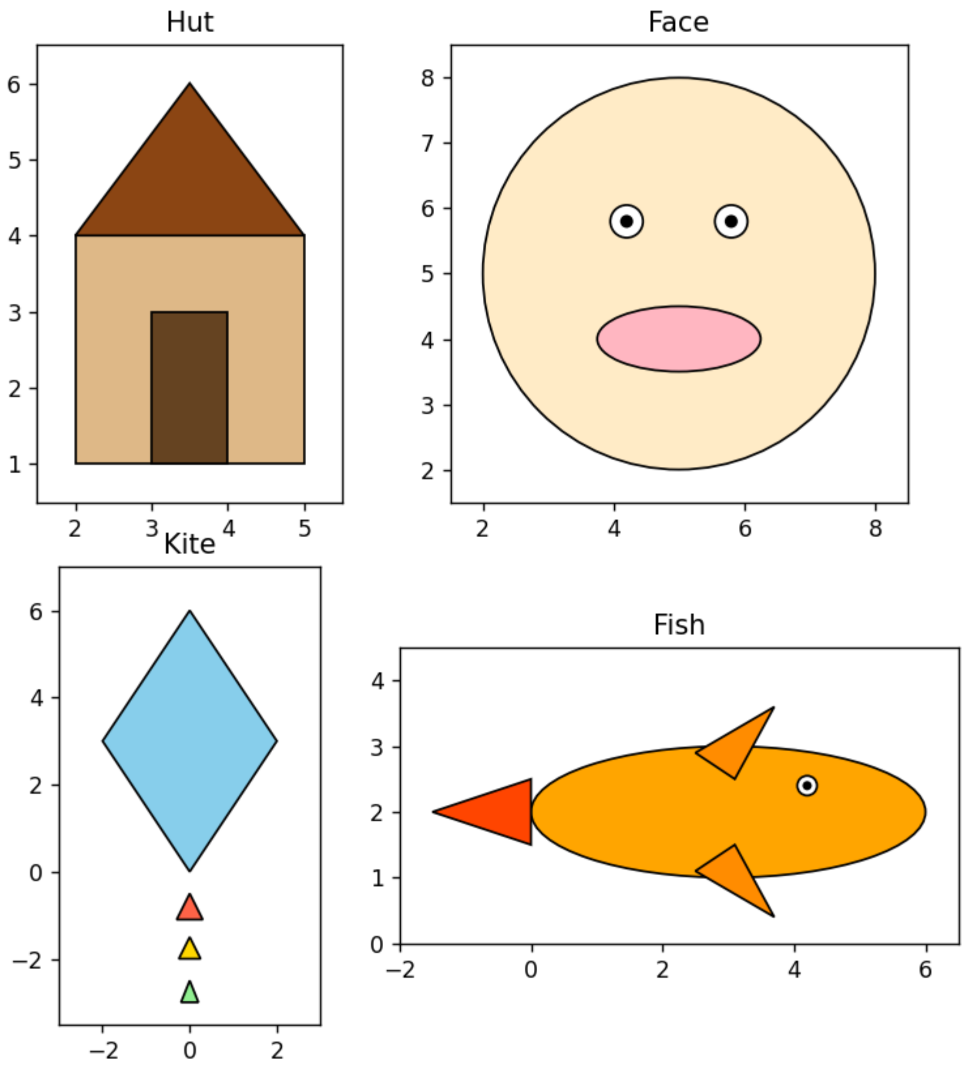
To draw different shapes such as a hut, face, kite, fish, and other objects using computer graphics, we use the basic **primitive drawing functions** provided in graphics libraries. These primitives include points, lines, circles, rectangles, arcs, and polygons. Complex figures are created by combining and arranging these basic shapes.

Program Code (Python):

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.patches import Polygon, Circle, Ellipse

# 1. HUT
def draw_hut(ax):
   walls = Polygon([[2,1],[2,4],[5,4],[5,1]], closed=True, facecolor="#deb887",
edgecolor="black")
   ax.add_patch(walls)

   roof = Polygon([[2,4],[3.5,6],[5,4]], closed=True, facecolor="#8b4513",
edgecolor="black")
   ax.add_patch(roof)

   door = Polygon([[3,1],[3,3],[4,3],[4,1]], closed=True, facecolor="#654321",
edgecolor="black")
   ax.add_patch(door)

   ax.set_title("Hut")
   ax.set_aspect('equal')
   ax.set_xlim(1.5, 5.5)
   ax.set_ylim(0.5, 6.5)

# 2. FACE
def draw_face(ax):
   face = Circle((5,5), 3, facecolor="#ffebc6", edgecolor="black")
   ax.add_patch(face)

   eye1 = Circle((4.2,5.8), 0.25, facecolor="white", edgecolor="black")
   eye2 = Circle((5.8,5.8), 0.25, facecolor="white", edgecolor="black")
   ax.add_patch(eye1); ax.add_patch(eye2)

   pupil1 = Circle((4.2,5.8), 0.1, facecolor="black")
   pupil2 = Circle((5.8,5.8), 0.1, facecolor="black")
   ax.add_patch(pupil1); ax.add_patch(pupil2)

   mouth = Ellipse((5,4), 2.5, 1, angle=0, facecolor="#ffb6c1", edgecolor="black")
```

```
    ax.add_patch(mouth)

    ax.set_title("Face")
    ax.set_aspect('equal')
    ax.set_xlim(1.5, 8.5)
    ax.set_ylim(1.5, 8.5)

# 3. KITE
def draw_kite(ax):
    kite = Polygon([[0,0],[2,3],[0,6],[-2,3]], closed=True,
                   facecolor="#87ceeb", edgecolor="black")
    ax.add_patch(kite)

    tail_segments = [
        Polygon([[0,-0.5],[0.3,-1.1],[-0.3,-1.1]], closed=True, facecolor="#ff6347",
edgecolor="black"),
        Polygon([[0,-1.5],[0.25,-2.0],[-0.25,-2.0]], closed=True, facecolor="#ffd700",
edgecolor="black"),
        Polygon([[0,-2.5],[0.2,-3.0],[-0.2,-3.0]], closed=True, facecolor="#90ee90",
edgecolor="black")
    ]
    for seg in tail_segments:
        ax.add_patch(seg)

    ax.set_title("Kite")
    ax.set_aspect('equal')
    ax.set_xlim(-3, 3)
    ax.set_ylim(-3.5, 7)

# 4. FISH
def draw_fish(ax):
    body = Ellipse((3,2), 6, 2, facecolor="#ffa500", edgecolor="black")
    ax.add_patch(body)

    tail = Polygon([[0,2.5],[-1.5,2],[0,1.5]], closed=True,
                   facecolor="#ff4500", edgecolor="black")
    ax.add_patch(tail)

    fin_top = Polygon([[2.5,2.9],[3.7,3.6],[3.1,2.5]], closed=True, facecolor="#ff8c00",
edgecolor="black")
    fin_bottom = Polygon([[2.5,1.1],[3.7,0.4],[3.1,1.5]], closed=True, facecolor="#ff8c00",
edgecolor="black")
    ax.add_patch(fin_top); ax.add_patch(fin_bottom)

    eye = Circle((4.2,2.4), 0.15, facecolor="white", edgecolor="black")
    pupil = Circle((4.2,2.4), 0.07, facecolor="black")
    ax.add_patch(eye); ax.add_patch(pupil)

    ax.set_title("Fish")
    ax.set_aspect('equal')
    ax.set_xlim(-2, 6.5)
    ax.set_ylim(0, 4.5)

# PLOT ALL SHAPES IN ONE FRAME
fig, axes = plt.subplots(2, 2, figsize=(12,10))

draw_hut(axes[0][0])
draw_face(axes[0][1])
draw_kite(axes[1][0])
draw_fish(axes[1][1])

plt.tight_layout()
plt.show()
```

Output:



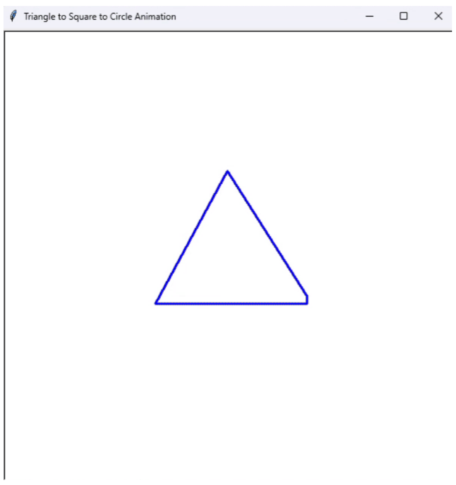
Experiment 10: To produce animation effect of triangle transform into square and then circle.

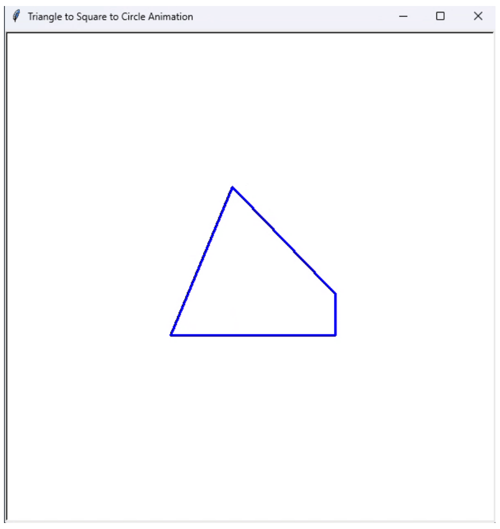
To produce the animation effect where a triangle gradually transforms into a square and then into a circle, we use the concept of **morphing** in computer graphics. Morphing is the smooth transformation of one shape into another by gradually altering the positions of their vertices.

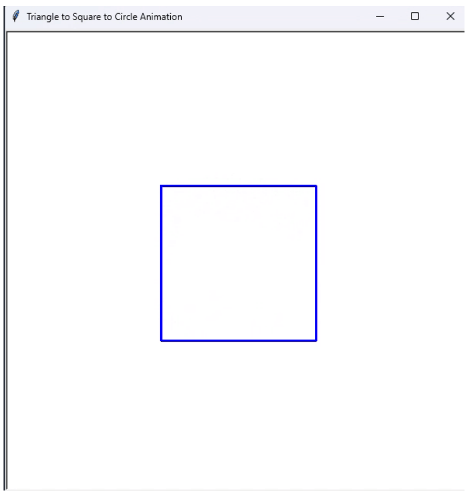
Program Code (Python):

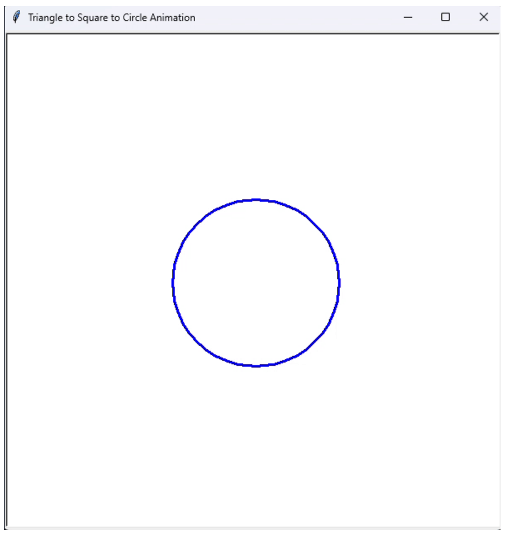
```
import turtle
import math
import time

# Setup turtle screen
screen = turtle.Screen()
screen.title("Triangle to Square to Circle Animation")
screen.bgcolor("white")
screen.setup(width=600, height=600)

# Create the turtle pen
pen = turtle.Turtle()
pen.speed(10)      # Set drawing speed to fastest
pen.pensize(3)    # Set pen thickness
pen.color("blue") # Pen color
pen.hideturtle()  # Hide the turtle cursor

def draw_shape(points):
    """
    Draws a closed shape connecting all points in the list.
    """
    pen.clear()           # Clear previous drawing
    pen.penup()           # Lift pen to move without drawing
    pen.goto(points[0])   # Move to the first point
    pen.pendown()         # Put pen down to start drawing

    # Draw lines to each subsequent point
    for pt in points[1:]:
        pen.goto(pt)

    pen.goto(points[0])   # Close the shape by returning to the first point

def interpolate_points(points_start, points_end, t):
    """
    Linearly interpolates between two lists of points.
    t is interpolation parameter between 0 and 1.
    Returns a new list of interpolated points.
    """
    return [((1 - t) * sx + t * ex, (1 - t) * sy + t * ey)
            for (sx, sy), (ex, ey) in zip(points_start, points_end)]

def morph_points(points_start, points_end, steps=60):
    """
    Morphs shape from points_start to points_end over a number of steps.
    For each step, interpolates the points and redraws the shape.
    """
    for i in range(steps + 1):
        t = i / steps  # interpolation fraction from 0 to 1
        points = interpolate_points(points_start, points_end, t)  # get intermediate points
        draw_shape(points)     # draw the intermediate shape
        screen.update()        # update the screen with new drawing
        time.sleep(0.034)      # pause to create animation effect

def duplicate_points(points, target_len):
    """
    Given a list of points, returns a new list with target_len points.
    This is done by interpolating points along the perimeter to match the desired length.
    This is useful to have same number of points for morphing.
    """
    n = len(points)   # original number of points
    result = []

    for i in range(target_len):
        # Map i to position along the points perimeter (fractional index)
        pos = i * n / target_len

        # Index of current segment start point
        j = int(pos) % n

        # Index of next point (wrap around)
        next_j = (j + 1) % n

        # Fractional distance between points[j] and points[next_j]
        frac = pos - j

        # Linear interpolation between points[j] and points[next_j]
        x = (1 - frac) * points[j][0] + frac * points[next_j][0]
        y = (1 - frac) * points[j][1] + frac * points[next_j][1]

        result.append((x, y))
    return result

def main():
    turtle.tracer(0, 0)  # Turn off automatic screen updates for smoother animation

    # Define triangle points (equilateral triangle)
    tri_size = 200
    tri_pts = [
        (0, tri_size / math.sqrt(3)),                    # Top vertex
        (-tri_size/2, -tri_size / (2*math.sqrt(3))),      # Bottom-left vertex
        (tri_size/2, -tri_size / (2*math.sqrt(3)))        # Bottom-right vertex
    ]

    # Define square points (4 vertices)
    sq_size = 200
    square_pts = [
        (-sq_size/2, sq_size/2),    # Top-left
        (-sq_size/2, -sq_size/2),   # Bottom-left
        (sq_size/2, -sq_size/2),    # Bottom-right
        (sq_size/2, sq_size/2)      # Top-right
    ]

    # Define circle points - 60 points around a circle
    circle_points_count = 60
    r = sq_size / 2
    circle_pts = []
    for i in range(circle_points_count):
        angle = 2 * math.pi * i / circle_points_count
        x = r * math.cos(angle)
        y = r * math.sin(angle)
        circle_pts.append((x, y))

    # Duplicate last point of triangle to match square's 4 points
    tri_pts_dup = tri_pts + [tri_pts[-1]]

    # Morph from triangle to square
    morph_points(tri_pts_dup, square_pts, steps=60)
    time.sleep(2)  # Pause before next morph

    # Duplicate square points to match circle points count (60 points)
    square_pts_dup = duplicate_points(square_pts, circle_points_count)

    # Morph from square to circle
    morph_points(square_pts_dup, circle_pts, steps=80)
    time.sleep(1)  # Pause at final circle shape

    # Draw final circle shape
    draw_shape(circle_pts)
    screen.update()

    turtle.done()  # Wait for user to close window

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

OUTPUT:


Triangle to Square to Circle Animation


Triangle to Square to Circle Animation


Triangle to Square to Circle Animation


Triangle to Square to Circle Animation

Experiment 12: Create an animation of an arrow embedded into a circle revolving around its center.

To animate an arrow inside a circle such that it appears to rotate around the circle's center, we use the concept of **2D rotation transformation**.

A circle is drawn with a fixed center point (x_c, y_c). The arrow is drawn inside the circle using multiple coordinate points. To rotate the arrow, every point of the arrow is rotated around the center of the circle by a small angle θ. This rotation is repeated continuously, which creates the effect of animation.

The rotation of any point (x, y)around the center $(x_c, y_c)$is done using the **rotation formula**:

$$x' = x_c + (x - x_c)\cos\,\theta - (y - y_c)\sin\,\theta$$
$$y' = y_c + (x - x_c)\sin\,\theta + (y - y_c)\cos\,\theta$$

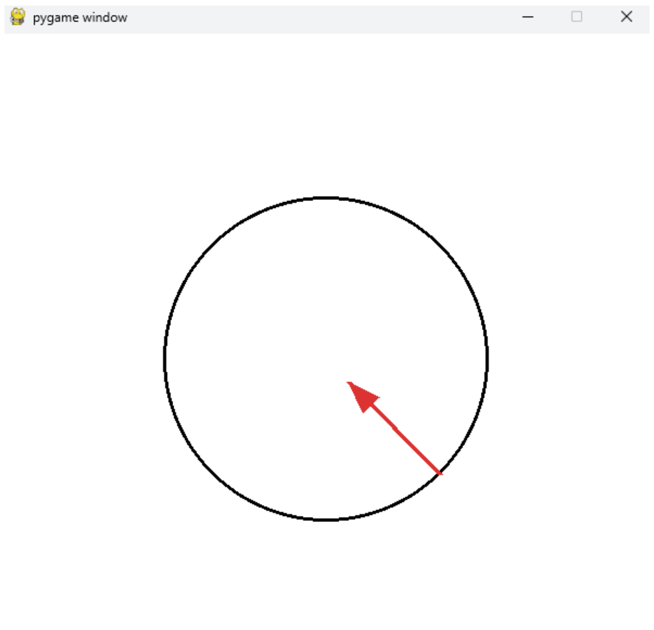
These new coordinates (x', y')give the rotated position of each point of the arrow.
By repeatedly increasing the angle (for example, $\theta = \theta + 1°$) and redrawing the arrow, a smooth rotation animation is obtained.

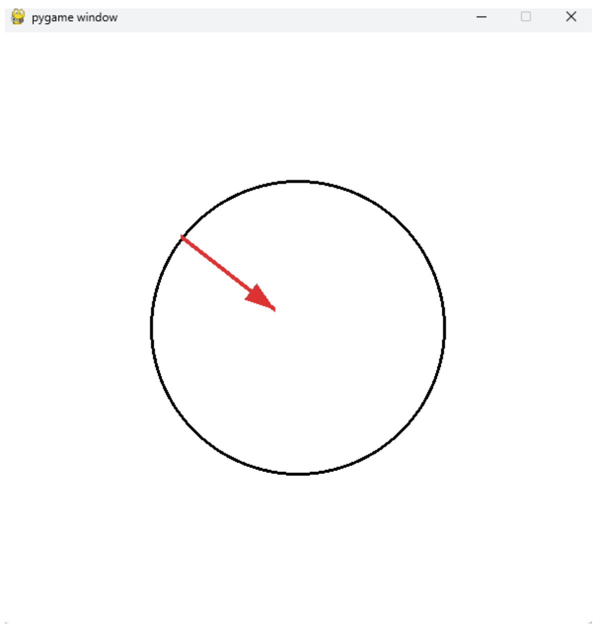
This technique is commonly used in computer graphics to animate objects such as clock hands, wheels, and rotating logos.

Program Code (Python):

```
import pygame
import math
import sys

pygame.init()

# Window dimensions
W, H = 600, 600
screen = pygame.display.set_mode((W, H))
clock = pygame.time.Clock()

center = (W // 2, H // 2)
radius = 150
angle = 0

def draw_arrow(surf, c, pos, ang, length=120, width=10):
    """
    Draw an arrow on surface 'surf' with color 'c', starting at 'pos',
    pointing in direction 'ang' (radians), with specified length and width.
    """
    end = (pos[0] + length * math.cos(ang), pos[1] + length * math.sin(ang))
    pygame.draw.line(surf, c, pos, end, width // 2)
    hl, hw = width * 3, width * 2
    left = (end[0] - hl * math.cos(ang) + hw * math.sin(ang) / 2,
            end[1] - hl * math.sin(ang) - hw * math.cos(ang) / 2)
    right = (end[0] - hl * math.cos(ang) - hw * math.sin(ang) / 2,
             end[1] - hl * math.sin(ang) + hw * math.cos(ang) / 2)
    pygame.draw.polygon(surf, c, [end, left, right])
while True:
    for e in pygame.event.get():
        if e.type == pygame.QUIT:
            pygame.quit()
            sys.exit()
    screen.fill((255, 255, 255))
    pygame.draw.circle(screen, (0, 0, 0), center, radius, 3)

    # Calculate arrow start position on circle edge
    arrow_start = (center[0] + radius * math.cos(angle), center[1] + radius * math.sin(angle))
    # Draw arrow pointing inward toward center
    draw_arrow(screen, (220, 50, 50), arrow_start, angle + math.pi)
    angle = (angle + 0.02) % (2 * math.pi)
    pygame.display.flip()
    clock.tick(60)
```

Output:


pygame window


pygame window