Department of Computer Application

National Institute of Technology Kurukshetra, Kurukshetra -136119

Computer Graphics & Multimedia Lab

MCA-233

(2024-27)

Submitted by:

Name: Suraj Pandey

Roll no: 524110023

Semester: 3rd

Section: Group 2

Submitted to:

Name: Dr. Suresh

Date: 17-11-2025

INDEX

Experiment 1: To implement DDA algorithm for line and circle

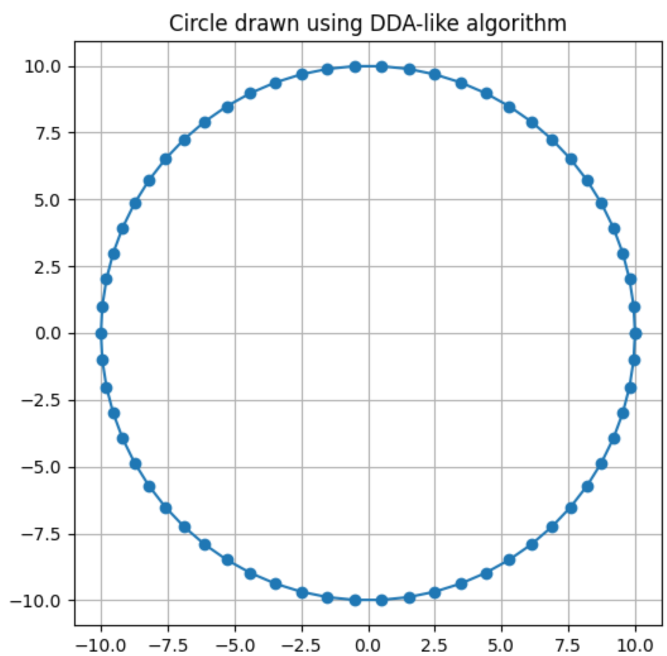
Program Code (Python):

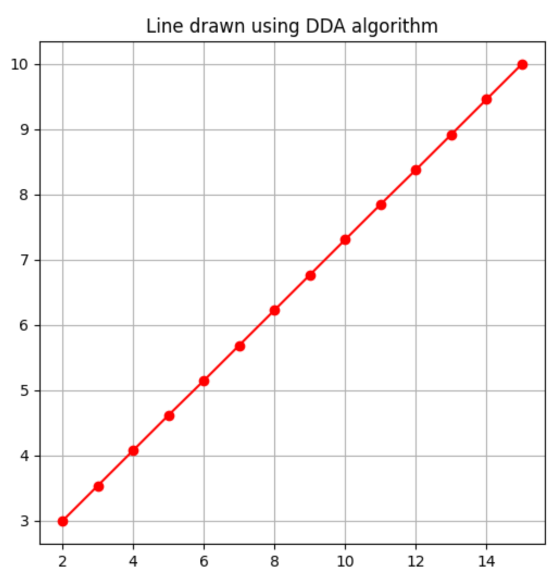
```
import math
import matplotlib.pyplot as plt
# Circle parameters
xc, yc = 0, 0
r = 10
# Number of steps to approximate the circle
steps = int(2 * math.pi * r)
x_points = []
y_points = []
theta = 0
theta_increment = (2 * math.pi) / steps
for _ in range(steps + 1):
    x = xc + r * math.cos(theta)
    y = yc + r * math.sin(theta)
    x_points.append(x)
    y_points.append(y)
    theta += theta_increment
plt.figure(figsize=(6,6))
plt.plot(x_points, y_points, marker='o')
plt.title("Circle drawn using DDA-like algorithm")
plt.axis('equal')
plt.grid(True)
plt.show()
```

Program Code (Python):

```
import matplotlib.pyplot as plt
# Line coordinates
x1, y1 = 2, 3
x2, y2 = 15, 10
# Calculate dx, dy
dx = x2 - x1
dy = y2 - y1
# Number of steps
steps = max(abs(dx), abs(dy))
# Calculate increments
x_inc = dx / steps
y_inc = dy / steps
# Generate points
x_points = []
y_points = []
x, y = x1, y1
for _ in range(steps + 1):
    x_points.append(round(x, 2))   # keep decimals for smooth plotting
    y_points.append(round(y, 2))
    x += x_inc
    y += y_inc
# Plot line
plt.figure(figsize=(6,6))
plt.plot(x_points, y_points, marker='o', color='r')
plt.title("Line drawn using DDA algorithm")
plt.grid(True)
plt.show()
```

Output:


Circle drawn using DDA-like 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


Line drawn using DDA algorithm
10
9
8
7
6
5
4
3
2
4
6
8
10
12
14

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

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

```
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()
```

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

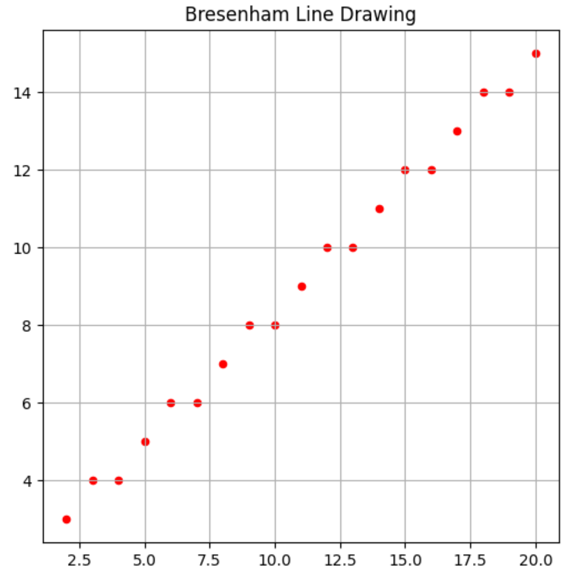
```
import matplotlib.pyplot as plt

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()
```

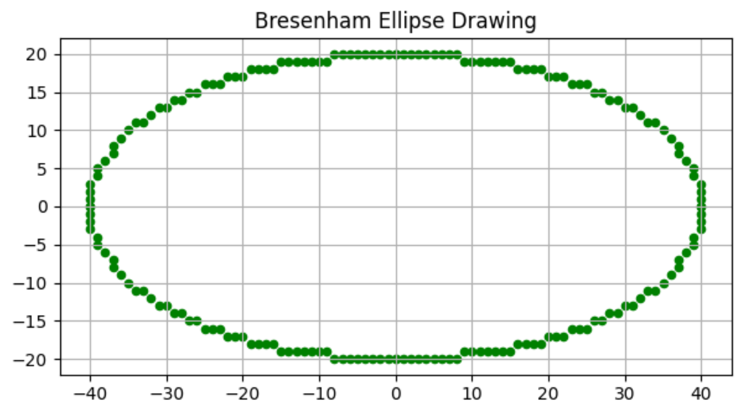
Program Code (Python): Bresenham's Ellipse Drawing Algorithm

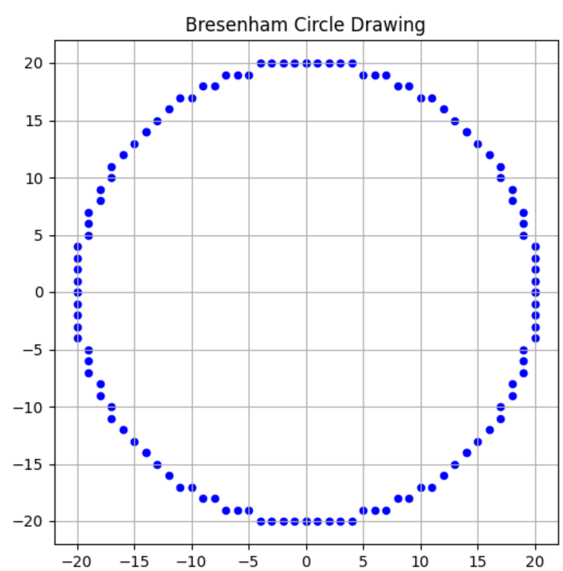
```
import matplotlib.pyplot as plt

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:


Bresenham Line Drawing
14
12
10
8
6
4
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


Bresenham Ellipse Drawing
20
15
10
5
0
−5
−10
−15
−20
−40
−30
−20
−10
0
10
20
30
40

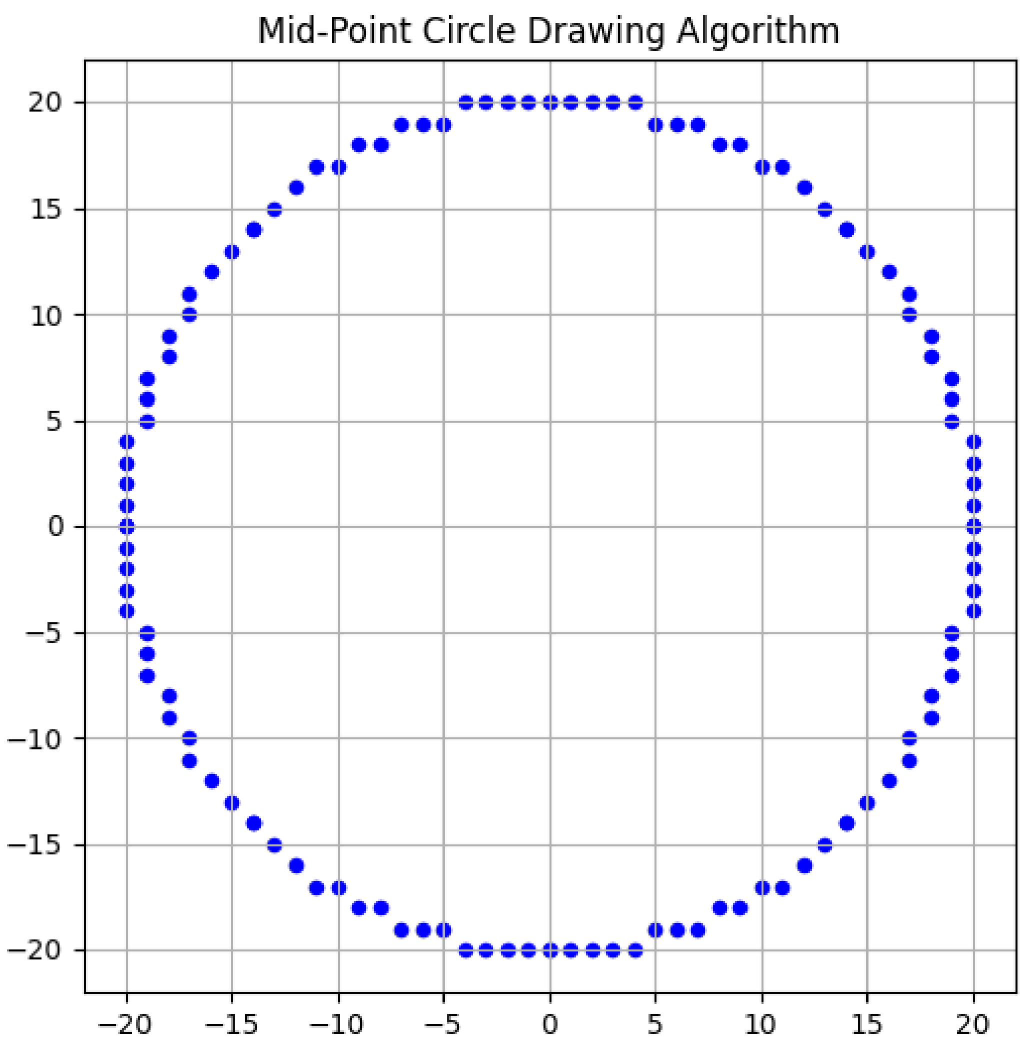

Bresenham Circle Drawing
20
15
10
5
0
−5
−10
−15
−20
−20
−15
−10
−5
0
5
10
15
20

Experiment 3: To implement Mid-Point circle algorithm

Program Code (Python):

```
import matplotlib.pyplot as plt
def midpoint_circle(xc, yc, r):
    x = 0
    y = r
    p = 1 - r   # Initial decision parameter
    points = []
    while x <= y:
        # 8-way symmetry
        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)
        ])
        x += 1
        if p < 0:
            p += 2 * x + 1
        else:
            y -= 1
            p += 2 * (x - y) + 1
    return points

# Example usage
circle_points = midpoint_circle(0, 0, 20)
x, y = zip(*circle_points)

plt.figure(figsize=(6,6))
plt.scatter(x, y, color='blue', s=20)
plt.title("Mid-Point Circle Drawing Algorithm")
plt.gca().set_aspect('equal', adjustable='box')
plt.grid(True)
plt.show()
```

Output:


Mid-Point Circle Drawing Algorithm
20
15
10
5
0
−5
−10
−15
−20
−20
−15
−10
−5
0
5
10
15
20

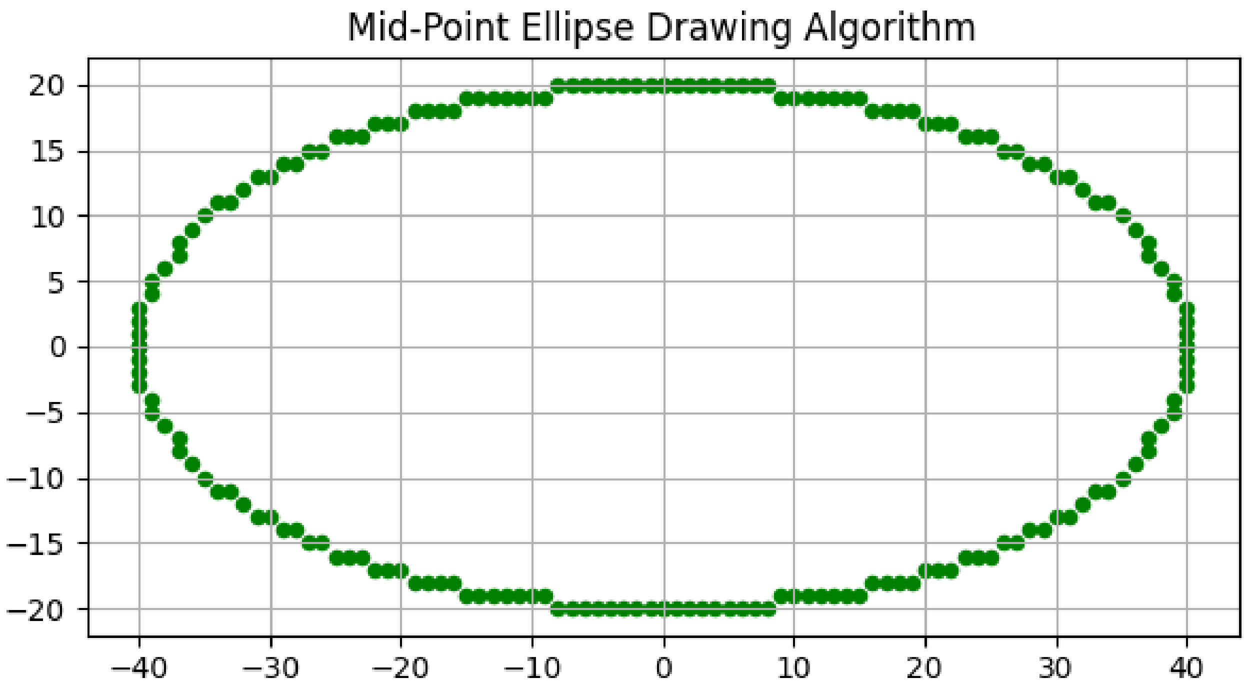
Experiment 4: To implement Mid-Point ellipse algorithm

Program Code (Python):

```
import matplotlib.pyplot as plt
def midpoint_ellipse(xc, yc, rx, ry):
    points = []
    x = 0
    y = ry

    # Region 1
    d1 = (ry**2) - (rx**2 * ry) + (0.25 * rx**2)
    dx = 2 * (ry**2) * x
    dy = 2 * (rx**2) * y

    while dx < dy:
        # 4-way symmetry
        points.extend([
            (xc + x, yc + y), (xc - x, yc + y),
            (xc + x, yc - y), (xc - x, yc - y)
        ])

        if d1 < 0:
            x += 1
            dx = dx + (2 * (ry**2))
            d1 = d1 + dx + (ry**2)
        else:
            x += 1
            y -= 1
            dx = dx + (2 * (ry**2))
            dy = dy - (2 * (rx**2))
            d1 = d1 + dx - dy + (ry**2)

    # Region 2
    d2 = ((ry**2) * ((x + 0.5) ** 2)) + ((rx**2) * ((y - 1) ** 2)) - (rx**2 * ry**2)
```

```
    while y >= 0:
        points.extend([
            (xc + x, yc + y), (xc - x, yc + y),
            (xc + x, yc - y), (xc - x, yc - y)
        ])
        if d2 > 0:
            y -= 1
            dy = dy - (2 * (rx**2))
            d2 = d2 + (rx**2) - dy
        else:
            y -= 1
            x += 1
            dx = dx + (2 * (ry**2))
            dy = dy - (2 * (rx**2))
            d2 = d2 + dx - dy + (rx**2)
    return points
# Example usage
ellipse_points = midpoint_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("Mid-Point Ellipse Drawing Algorithm")
plt.gca().set_aspect('equal', adjustable='box')
plt.grid(True)
plt.show()
```

Output:


Mid-Point Ellipse Drawing Algorithm
20
15
10
5
0
−5
−10
−15
−20
−40
−30
−20
−10
0
10
20
30
40

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

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()
```

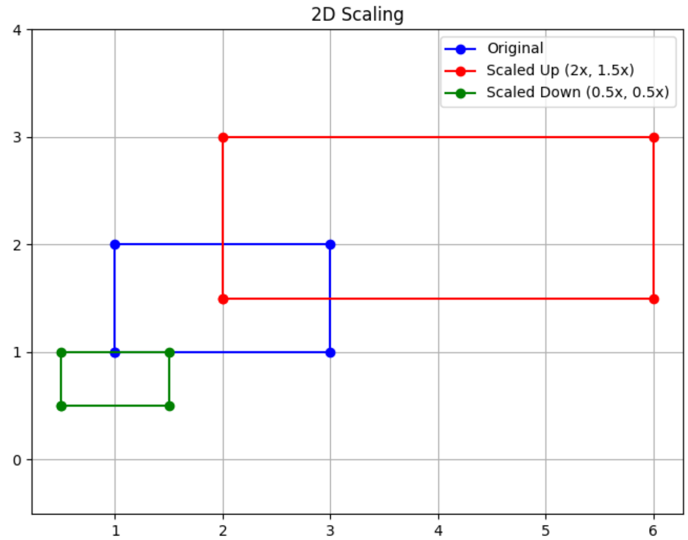
Program Code (Python): Shearing

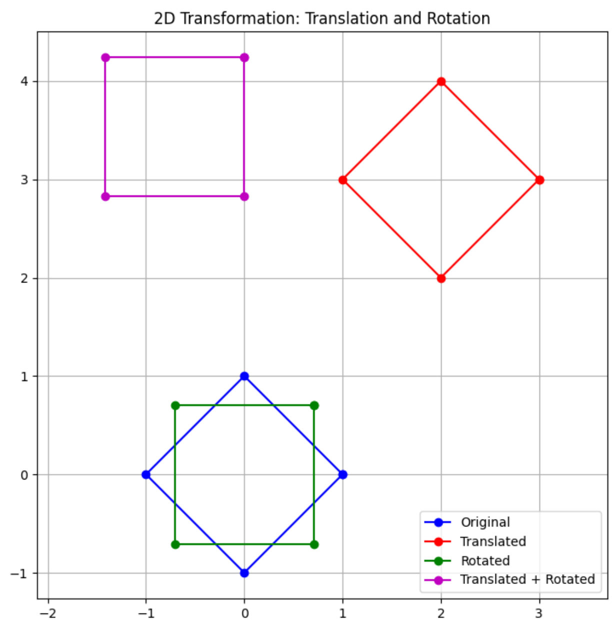
```
import numpy as np
import matplotlib.pyplot as plt
def shear_x(points, shx):
    """
    Shear points horizontally by shx.
    """
    shear_matrix = np.array([[1, shx],
                             [0, 1]])
    return points.dot(shear_matrix.T)
def shear_y(points, shy):
    """
    Shear points 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()
```

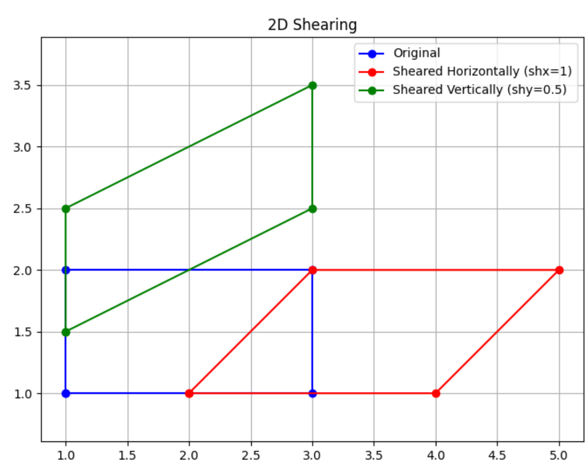
Program Code (Python): Rotation and translation

```
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:


2D Scaling
Original
Scaled Up (2x, 1.5x)
Scaled Down (0.5x, 0.5x)


2D Transformation: Translation and Rotation
Original
Translated
Rotated
Translated + Rotated


2D Shearing
Original
Sheared Horizontally (shx=1)
Sheared Vertically (shy=0.5)

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

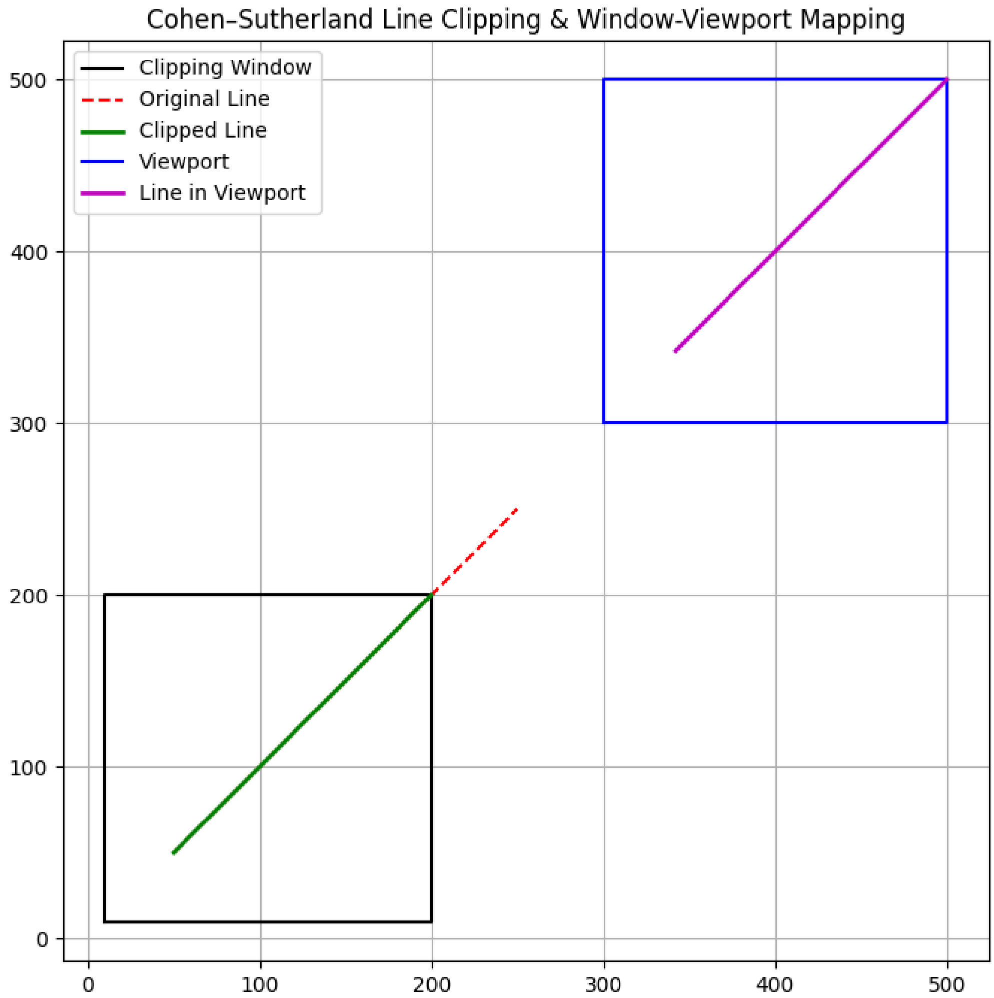
Program Code (Python):

```
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:


Cohen–Sutherland Line Clipping & Window-Viewport Mapping
Clipping Window
Original Line
Clipped Line
Viewport
Line in Viewport
0
100
200
300
400
500
0
100
200
300
400
500

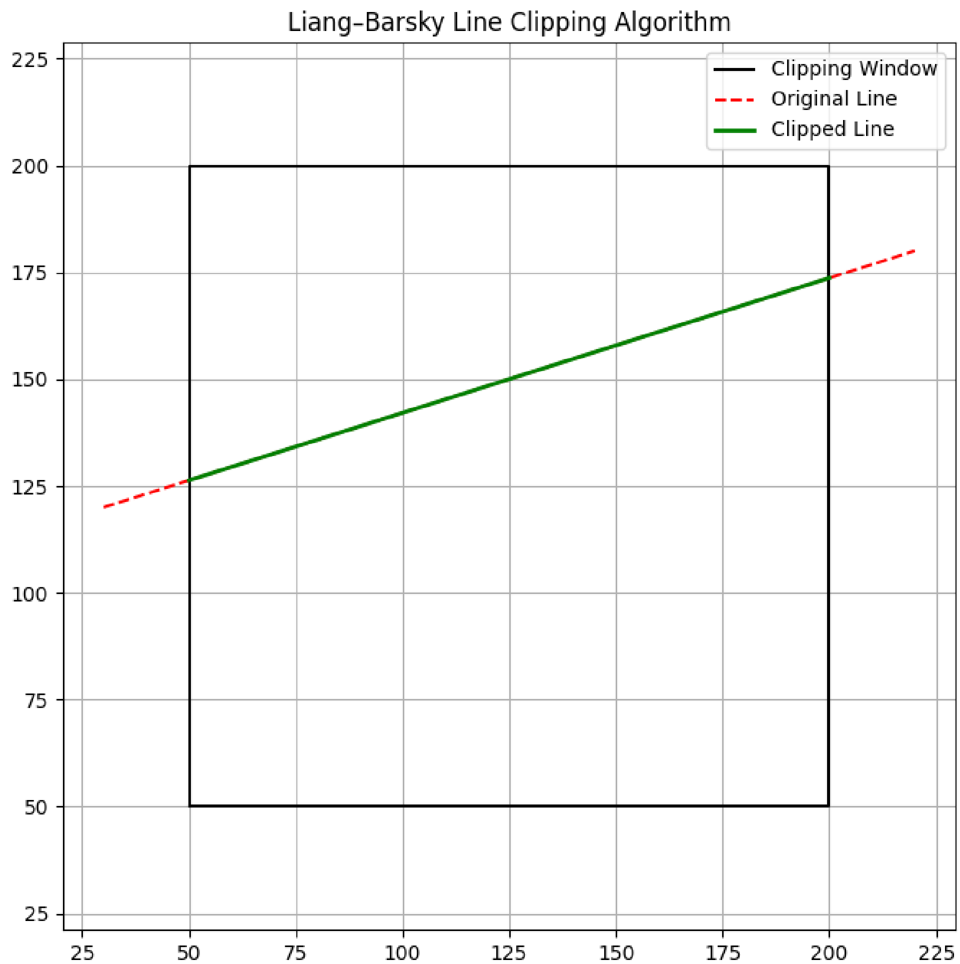
Experiment 7: To implement Liang Barsky line clipping algorithm

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:



Liang-Barsky Line Clipping Algorithm
Clipping Window
Original Line
Clipped Line
225
200
175
150
125
100
75
50
25
25
50
75
100
125
150
175
200
225

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

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()
```

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()
```

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],
```

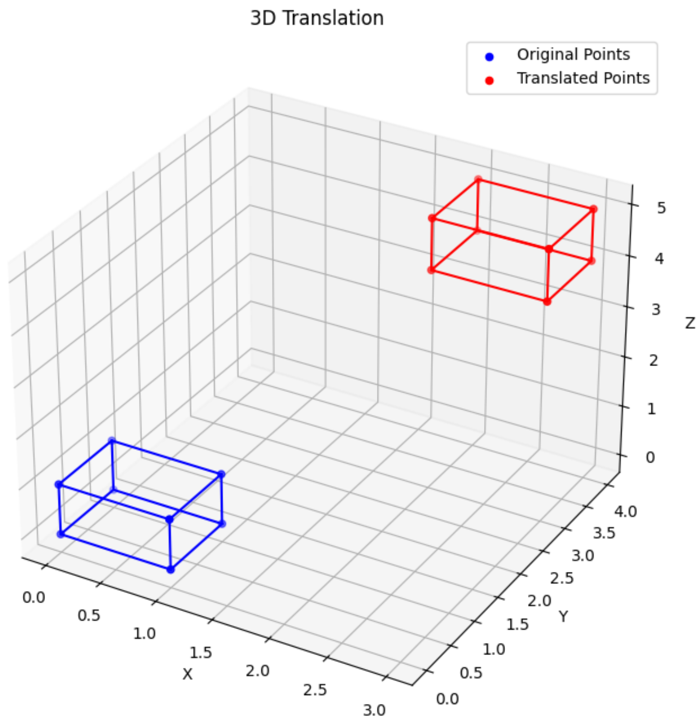
```
    [1, 1, 1],
    [0, 1, 1]
])

# 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()
```

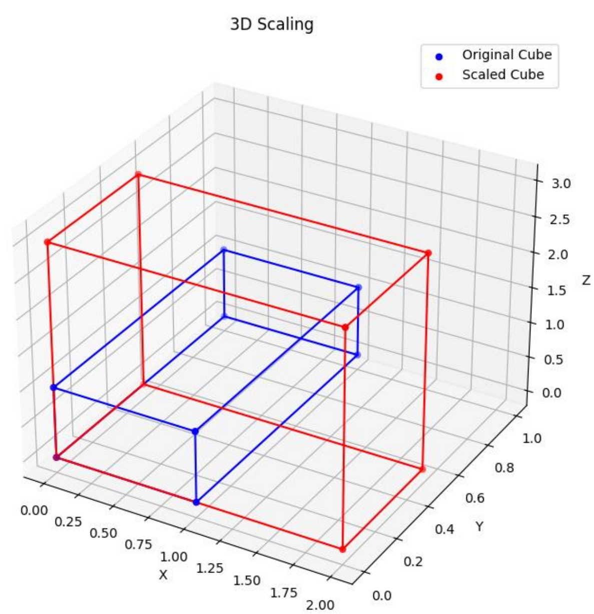
Program Code (Python): Scaling

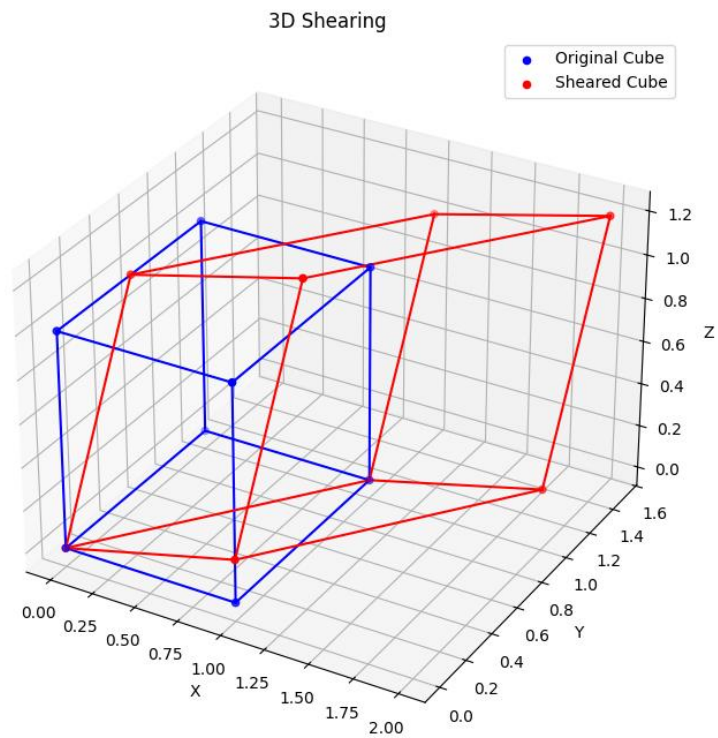
```
import numpy as np
import matplotlib.pyplot as plt

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)
```

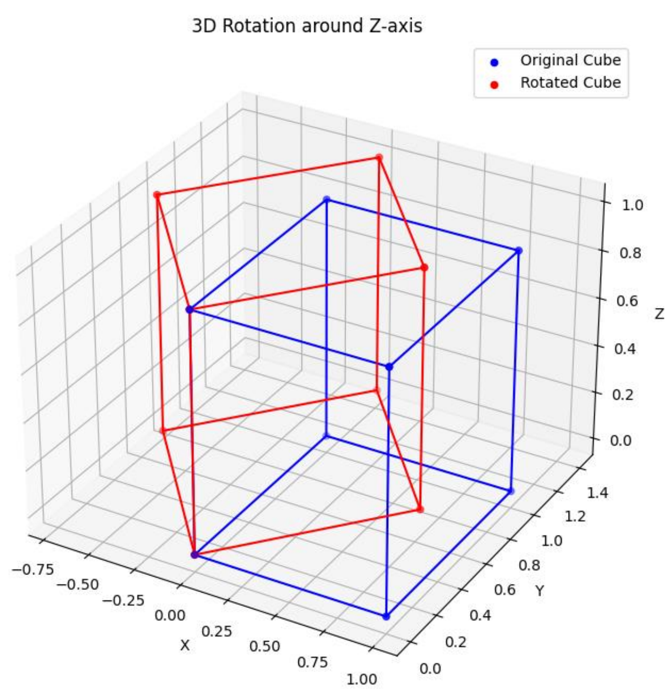
```
scaled_points = scale_3d(points, sx=2, sy=0.5, sz=3)

# 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:


3D Translation
Original Points
Translated Points
X
Y
Z
0.0
0.5
1.0
1.5
2.0
2.5
3.0
4.0
3.5
0
1
2
3
4
5


3D Scaling
Original Cube
Scaled Cube
X
Y
Z
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
0.2
0.4
0.6
0.8
0.0
0.5
1.0
1.5
2.0
2.5
3.0


3D Shearing
Original Cube
Sheared Cube
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
X
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
Y
0.0
0.2
0.4
0.6
0.8
1.0
1.2
Z


3D Rotation around Z-axis
Original Cube
Rotated Cube
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
X
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
Y
0.0
0.2
0.4
0.6
0.8
1.0
Z

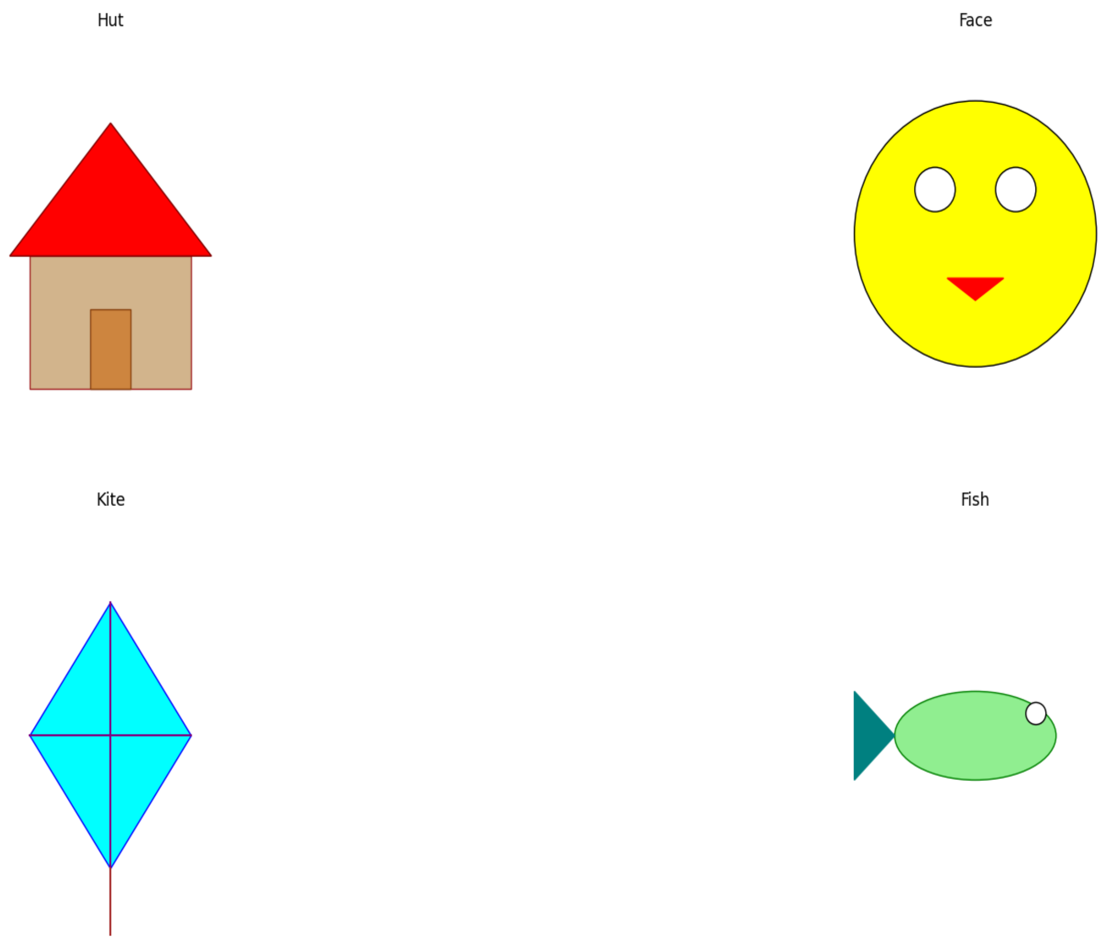
Experiment 9: To draw different shapes such as hut, face, kite, fish etc.

Program Code (Python):

```
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle, Circle, Polygon
from matplotlib.patches import Ellipse
fig, axes = plt.subplots(2, 2, figsize=(10, 10))
# ---- Hut ----
ax = axes[0, 0]
# base
ax.add_patch(Rectangle((0.3, 0.2), 0.4, 0.3, edgecolor='brown',
facecolor='tan'))
# roof
ax.add_patch(Polygon([[0.25, 0.5], [0.75, 0.5], [0.5, 0.8]],
edgecolor='maroon', facecolor='red'))
# door
ax.add_patch(Rectangle((0.45, 0.2), 0.1, 0.18, edgecolor='saddlebrown',
facecolor='peru'))
ax.set_title("Hut")
ax.set_xlim(0, 1)
ax.set_ylim(0, 1)
ax.set_aspect('equal')
ax.axis('off')
# ---- Face ----
ax = axes[0, 1]
# face outer circle
ax.add_patch(Circle((0.5, 0.55), 0.3, edgecolor='black',
facecolor='yellow'))
# eyes
ax.add_patch(Circle((0.4, 0.65), 0.05, edgecolor='black',
facecolor='white'))
ax.add_patch(Circle((0.6, 0.65), 0.05, edgecolor='black',
facecolor='white'))
# mouth
ax.add_patch(Polygon([[0.43, 0.45], [0.57, 0.45], [0.5, 0.4]],
edgecolor='red', facecolor='red'))
ax.set_title("Face")
ax.set_xlim(0, 1)
ax.set_ylim(0, 1)
ax.set_aspect('equal')
ax.axis('off')
# ---- Kite ----
ax = axes[1, 0]
# kite body
```

```
ax.add_patch(Polygon([[0.5, 0.8], [0.7, 0.5], [0.5, 0.2], [0.3, 0.5]],
edgecolor='blue', facecolor='cyan'))
# cross lines
ax.plot([0.5, 0.5], [0.8, 0.2], color='purple')
ax.plot([0.3, 0.7], [0.5, 0.5], color='purple')
# tail
ax.plot([0.5, 0.5], [0.2, 0.05], color='brown')
ax.set_title("Kite")
ax.set_xlim(0, 1)
ax.set_ylim(0, 1)
ax.set_aspect('equal')
ax.axis('off')
# ---- Fish ----
ax = axes[1, 1]
# body
ax.add_patch(Ellipse((0.5, 0.5), 0.4, 0.2, edgecolor='green',
facecolor='lightgreen'))
# tail
ax.add_patch(Polygon([[0.3, 0.5], [0.2, 0.6], [0.2, 0.4]],
edgecolor='teal', facecolor='teal'))
# eye
ax.add_patch(Circle((0.65, 0.55), 0.025, edgecolor='black',
facecolor='white'))
ax.set_title("Fish")
ax.set_xlim(0, 1)
ax.set_ylim(0, 1)
ax.set_aspect('equal')
ax.axis('off')
plt.tight_layout()
plt.show()
```

Output:



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

Program Code (Python):

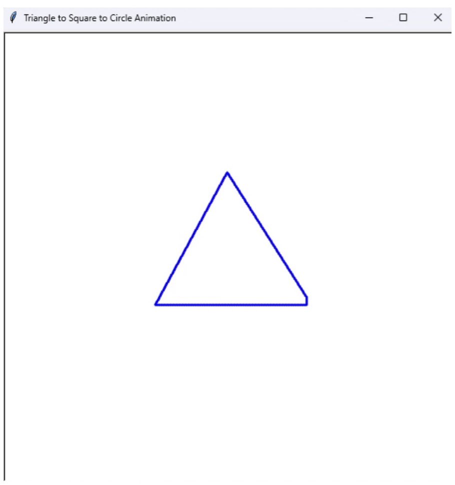
```
import turtle
import math
import time

# 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)
```

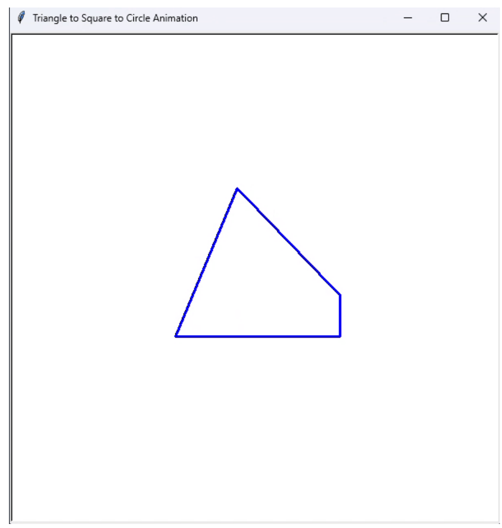
```
            for (sx, sy), (ex, ey) in zip(points_start, points_end)]

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
```

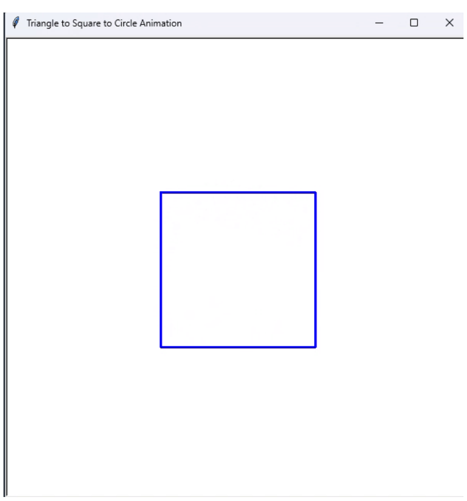
```
def main():
    turtle.tracer(0, 0)  # Turn off automatic screen updates for smoother
animation

    # 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)
```

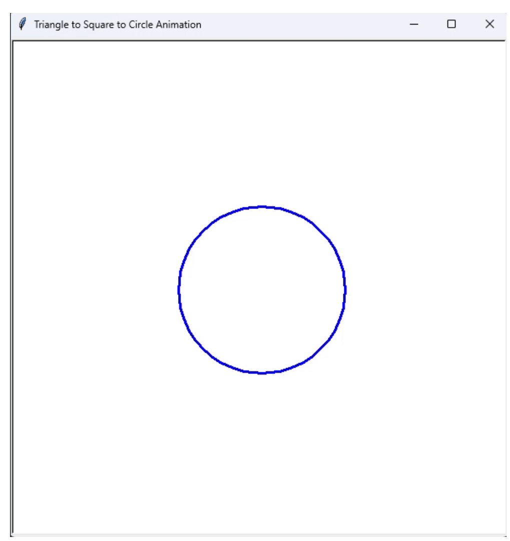
```
    time.sleep(1)  # Pause at final circle shape

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

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

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

Output:









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

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: