Computer Graphics and Image Processing Laboratory
(CSPC-423)

B.Tech VIIth Semester
(August – December 2024)

Submitted by

Ashutosh Jha (21103029)
Group-G2

Submitted to

Ms. Sukhwinder Kaur

Department of Computer Science & Engineering
Dr. B.R. Ambedkar National Institute of Technology Jalandhar
-144008, Punjab, India

Table of Content

Practical#1

Objective: Implement and plot the DDA Line Drawing Algorithm.

Theory:

DDA (Digital Differential Analyzer) is a line drawing algorithm used in computer graphics to generate a line segment between two specified endpoints. It is a simple and efficient algorithm that works by using the incremental difference between the x-coordinates and y-coordinates of the two endpoints to plot the line.

The steps involved in DDA line generation algorithm are:

1. Input the two endpoints of the line segment, (x1,y1) and (x2,y2).
2. Calculate the difference between the x-coordinates and y-coordinates of the endpoints as dx and dy respectively.
3. Calculate the slope of the line as m = dy/dx.
4. Set the initial point of the line as (x1,y1).
5. Loop through the x-coordinates of the line, incrementing by one each time, and calculate the corresponding y-coordinate using the equation y = y1 + m(x – x1).
6. Plot the pixel at the calculated (x,y) coordinate.
7. Repeat steps 5 and 6 until the endpoint (x2,y2) is reached.

DDA algorithm is relatively easy to implement and is computationally efficient, making it suitable for real-time applications. However, it has some limitations, such as the inability to handle vertical lines and the need for floating-point arithmetic, which can be slow on some systems. Nonetheless, it remains a popular choice for generating lines in computer graphics.

In any 2-Dimensional plane, if we connect two points (x0, y0) and (x1, y1), we get a line segment. But in the case of computer graphics, we can not directly join any two coordinate points, for that, we should calculate intermediate points' coordinates and put a pixel for each intermediate point, of the desired color with the help of functions like putpixel(x, y, K) in C, where (x,y) is our co-ordinate and K denotes some color.

DDA Algorithm:

Consider one point of the line as (X0, Y0) and the second point of the line as (X1, Y1).

```
// calculate dx , dy
dx = X1 – X0;
dy = Y1 – Y0;
// Depending upon absolute value of dx & dy
// choose number of steps to put pixel as
// steps = abs(dx) > abs(dy) ? abs(dx) : abs(dy)
steps = abs(dx) > abs(dy) ? abs(dx) : abs(dy);
// calculate increment in x & y for each steps
Xinc = dx / (float) steps;
Yinc = dy / (float) steps;
// Put pixel for each step
X = X0;
Y = Y0;
for (int i = 0; i <= steps; i++)
{
   putpixel (round(X),round(Y),WHITE);
   X += Xinc;
```

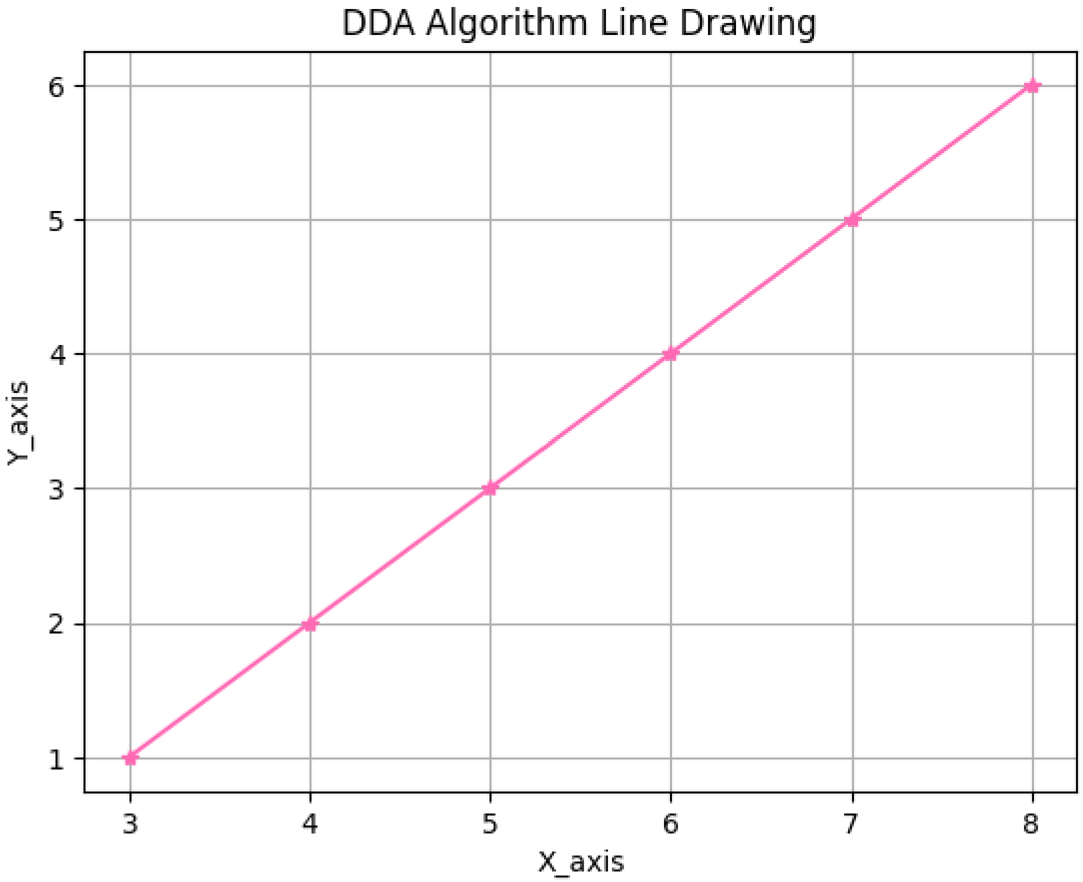
Y += Yinc;
}

Code:

```
import matplotlib.pyplot as plt

def DDA(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1


    steps = abs(dx) if abs(dx) > abs(dy) else abs(dy)
   #steps = max(abs(dx),abs(dy))
    x_increment = dx / steps
    y_increment = dy / steps

    x = x1
    y = y1

    points = []

    for i in range(steps):
        x += x_increment
        y += y_increment
        points.append((round(x), round(y)))

    return points, steps

def draw_line(points):
    x_values = [point[0] for point in points]
    y_values = [point[1] for point in points]
    plt.plot(x_values, y_values, marker='*',color='hotpink')
    plt.xlabel('X_axis')
    plt.ylabel('Y_axis')
    plt.title('DDA Algorithm Line Drawing')
    plt.grid(True)
    plt.show()

x1, y1 = 2, 0
x2, y2 =8, 6
points, steps = DDA(x1, y1, x2, y2)
print("Number of iterations:", steps)
print("Points between the two  points:", points)
draw_line(points)
```

Output:

```
Number of iterations: 6
Points between the two  points: [(3, 1), (4, 2), (5, 3), (6, 4), (7, 5), (8, 6)]
```



Practical#2

Objective:

A. Implement and plot the Bresenham's Line Generation Algorithm

Theory:

This algorithm is used for scan converting a line. It was developed by Bresenham. It is an efficient method because it involves only integer addition, subtractions, and multiplication operations. These operations can be performed very rapidly so lines can be generated quickly. In this method, next pixel selected is that one who has the least distance from true line.

Algorithm:

Step1: Start Algorithm
Step2: Declare variable x1,x2,y1,y2,d,i1,i2,dx,dy
Step3: Enter value of x1,y1,x2,y2
Where x1,y1are coordinates of starting point
And x2,y2 are coordinates of Ending point
Step4: Calculate dx = x2-x1
Calculate dy = y2-y1
Calculate i1=2*dy
Calculate i2=2*(dy-dx)
Calculate d=i1-dx
Step5: Consider (x, y) as starting point and xendas maximum possible value of x.
If dx < 0
Then x = x2
y = y2
xend=x1
If dx > 0
Then x = x1
y = y1
xend=x2

Step6: Generate point at (x,y)coordinates.
Step7: Check if whole line is generated.
If x > = xend
Stop.
Step8: Calculate co-ordinates of the next pixel
If d < 0
Then d = d + i1
If d ≥ 0

Then d = d + i2
Increment y = y + 1
Step9: Increment x = x + 1

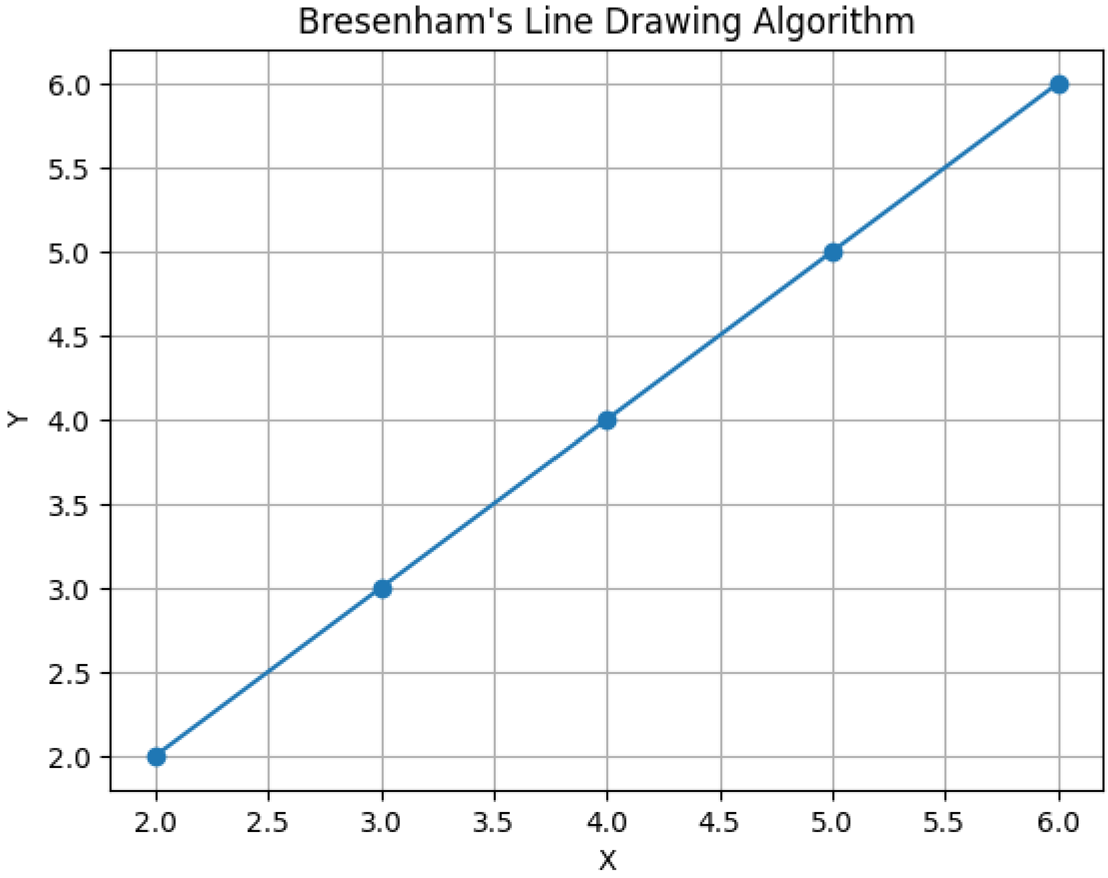
Step10: Draw a point of latest (x, y) coordinates
Step11: Go to step 7
Step12: End of Algorithm

Code:

```
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

def plot_line(x1, y1, x2, y2):
    points = bresenham_line(x1, y1, x2, y2)
    x_coords, y_coords = zip(*points)

    plt.plot(x_coords, y_coords, marker='o')
    plt.title('Bresenham\'s Line Drawing Algorithm')
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.grid(True)
    plt.show()

    # Print the output coordinates
    for i, (x, y) in enumerate(points):
        print(f"Point {i+1}: ({x}, {y})")

# Example usage
x1, y1 = 2, 2
x2, y2 = 6, 6

plot_line(x1, y1, x2, y2)
```

Output:

```
...  Point 1: (2, 2)
     Point 2: (3, 3)
     Point 3: (4, 4)
     Point 4: (5, 5)
     Point 5: (6, 6)
```



Objective:

B. Implementation of Mid-Point Circle drawing algorithm. Also, show all the symmetrical octant coordinates along with resultant circle

Theory:

The mid-point circle drawing algorithm is an algorithm used to determine the points needed for rasterizing a circle.
We use the mid-point algorithm to calculate all the perimeter points of the circle in the first octant and
then print them along with their mirror points in the other octants. This will work because a circle is
symmetric about its centre.
For any given pixel (x, y), the next pixel to be plotted is either (x, y+1) or (x-1, y+1). This can be decided
by following the steps below.
1. Find the mid-point p of the two possible pixels i.e (x-0.5, y+1)

2. If p lies inside or on the circle perimeter, we plot the pixel (x, y+1), otherwise if it’s outside we
plot the pixel (x-1, y+1)

Algorithm:

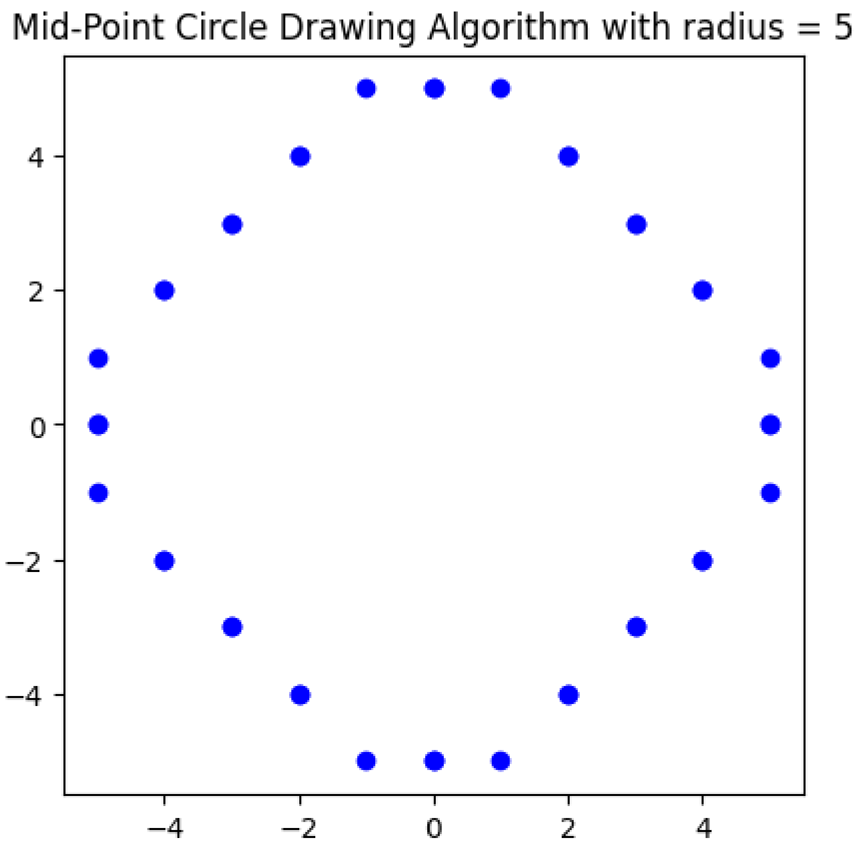
Step1: Put x =0, y =r in equation 2
We have p=1-r
Step2: Repeat steps while $x \leq y$
Plot (x, y)
If (p<0)
Then set p = p + 2x + 3
Else
p = p + 2(x-y)+5
y =y - 1 (end if)
x =x+1 (end loop)
Step3: End

Code:

```
import matplotlib.pyplot as plt

def plot_circle_points(x, y):
    points = [
        (x, y), (y, x), (-y, x), (-x, y),
        (-x, -y), (-y, -x), (y, -x), (x, -y)
    ]
    for point in points:
        plt.plot(*point, 'bo')
    return points

def draw_circle(radius):
    x = 0
    y = radius
    p = 1 - radius  # Initial decision parameter

    all_points = []
    all_points.extend(plot_circle_points(x, y))  # Plot initial
points

    while x <= y:
        x += 1
        if p < 0:
            p = p + 2 * x + 3
        else:
            y -= 1
            p = p + 2 * (x - y) + 5
        all_points.extend(plot_circle_points(x, y))  # Plot
symmetrical points
```

```
    plt.gca().set_aspect('equal', adjustable='box')
    plt.title(f"Mid-Point Circle Drawing Algorithm with radius = {radius}")
    plt.show()

    # Remove duplicates from the list of points and print them
    unique_points = list(set(all_points))
    unique_points.sort()
    print("Symmetrical Octant Coordinates:")
    for i, (x, y) in enumerate(unique_points):
        print(f"Point {i+1}: ({x}, {y})")

if __name__ == "__main__":
    radius = int(input("Enter the radius of the circle: "))
    draw_circle(radius)
```

Output:



Practical#3

Objective: Implement and plot the Bresenham's Circle Drawing Algorithm.

Theory:

Scan-Converting a circle using Bresenham's algorithm works as follows: Points are generated from 90° to 45°, moves will be made only in the +x & -y directions. The best approximation of the true circle will be described by those pixels in the raster that falls the least distance from the true circle. We want to generate the points from 90° to 45°. Assume that the last scan-converted pixel is P1 as shown in fig. Each new point closest to the true circle can be found by taking either of two actions.

1. Move in the x-direction one unit or
2. Move in the x- direction one unit & move in the negative y-direction one unit.

Algorithm:

Step1: Start Algorithm
Step2: Declare p, q, x, y, r, d variables
p, q are coordinates of the center of the circle
r is the radius of the circle
Step3: Enter the value of r
Step4: Calculate d = 3 - 2r
Step5: Initialize x=0
&nbsy= r
Step6: Check if the whole circle is scan converted
If x > = y
Stop
Step7: Plot eight points by using concepts of eight-way symmetry. The center is at (p, q).
Current active
pixel is (x, y).
putpixel (x+p, y+q)
putpixel (y+p, x+q)
putpixel (-y+p, x+q)
putpixel (-x+p, y+q)
putpixel (-x+p, -y+q)
putpixel (-y+p, -x+q)
putpixel (y+p, -x+q)
putpixel (x+p, -y-q)

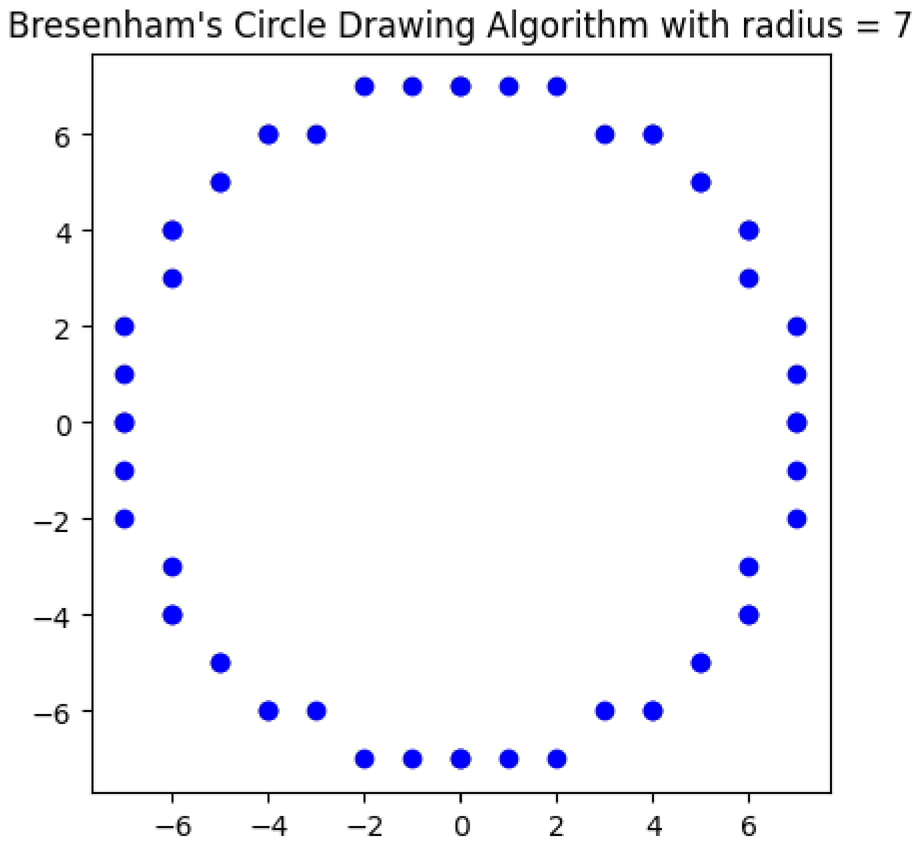
Step8: Find location of next pixels to be scanned
If d < 0
then d = d + 4x + 6
increment x = x + 1
If d ≥ 0

then d = d + 4 (x - y) + 10
increment x = x + 1
decrement y = y - 1
Step9: Go to step 6
Step10: Stop AlgorithmCode:

Code:

```
import matplotlib.pyplot as plt

def plot_circle_points(x, y):
    points = [
        (x, y), (y, x), (-y, x), (-x, y),
        (-x, -y), (-y, -x), (y, -x), (x, -y)
    ]
    for point in points:
        plt.plot(*point, 'bo')
    return points

def draw_circle(radius):
    x = 0
    y = radius
    d = 3 - 2 * radius
    all_points = []
    all_points.extend(plot_circle_points(x, y))

    while x <= y:
        if d < 0:
            d = d + 4 * x + 6
        else:
            d = d + 4 * (x - y) + 10
            y -= 1
        x += 1
        all_points.extend(plot_circle_points(x, y))

    plt.gca().set_aspect('equal', adjustable='box')
    plt.title(f"Bresenham's Circle Drawing Algorithm with radius
= {radius}")
    plt.show()

    unique_points = list(set(all_points))
    for i, (x, y) in enumerate(sorted(unique_points)):
        print(f"Point {i+1}: ({x}, {y})")

if __name__ == "__main__":
    radius = int(input("Enter the radius of the circle: "))
    draw_circle(radius)
```

Output:


Bresenham's Circle Drawing Algorithm with radius = 7
6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

Practical#4

Objective: Implement Mid Point Ellipse drawing algorithm. Also, print the output coordinates and display resultant ellipse.

Theory:

This is an incremental method for scan converting an ellipse that is centered at the origin in standard position i.e., with the major and minor axis parallel to coordinate system axis. It is very similar to the midpoint circle algorithm. Because of the four-way symmetry property we need to consider the entire elliptical curve in the first quadrant.
Let's first rewrite the ellipse equation and define the function f that can be used to decide if the midpoint between two candidate pixels is inside or outside the ellipse:

$$f(x, y) = b^2x^2 + a^2y^2 - a^2b^2 = \begin{bmatrix} < 0 (x, y) \text{inside} \\ o\ (x, y) \text{on} \\ > 0\ (x, y) \text{outside} \end{bmatrix}$$

Algorithm:

```
int x=0, y=b; [starting point]
int fx=0, fy=2a2 b [initial partial derivatives]
int p = b2-a2 b+a2/4
while (fx<="" 1="" {="" set="" pixel="" (x,="" y)="" x++;="" fx="fx" +="" 2b2;
        if (p<0)
        p = p + fx +b2;
        else
        {
                y--;
                fy=fy-2a2
                p = p + fx +b2-fy;
        }
}
Setpixel (x, y);
p=b2(x+0.5)2+ a2 (y-1)2- a2 b2
while (y>0)
{
        y--;
        fy=fy-2a2;
        if (p>=0)
        p=p-fy+a2
        else
        {
                x++;
                fx=fx+2b2
                p=p+fx-fy+a2;
        }
```

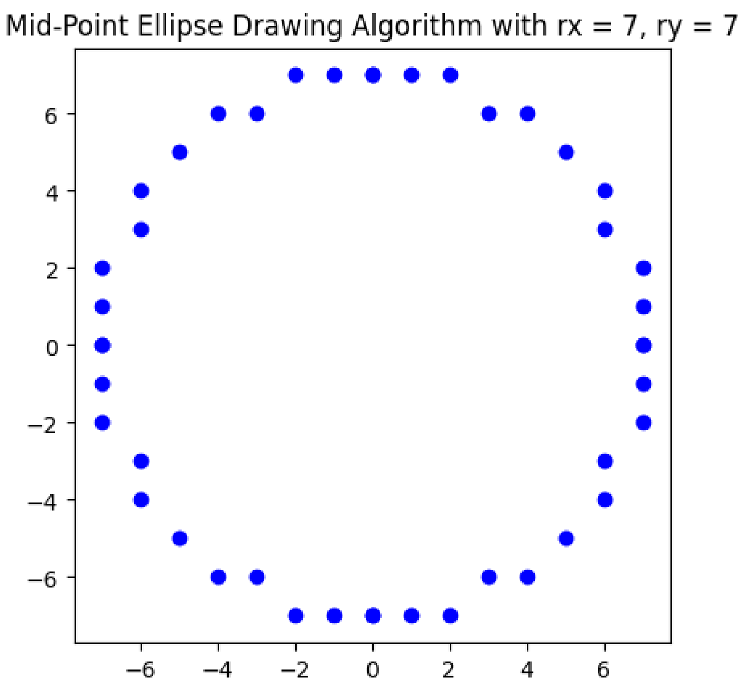
Setpixel (x,y);
}

Code:

```
import matplotlib.pyplot as plt

def plot_ellipse_points(x_center, y_center, x, y):
    points = [
        (x_center + x, y_center + y),
        (x_center - x, y_center + y),
        (x_center + x, y_center - y),
        (x_center - x, y_center - y)
    ]
    for point in points:
        plt.plot(*point, 'bo')
    return points

def draw_ellipse(rx, ry, x_center=0, y_center=0):
    x = 0
    y = ry
    rx2 = rx * rx
    ry2 = ry * ry
    tworx2 = 2 * rx2
    twory2 = 2 * ry2
    p1 = ry2 - (rx2 * ry) + (0.25 * rx2)
    px = 0
    py = tworx2 * y

    all_points = []
    all_points.extend(plot_ellipse_points(x_center, y_center, x,
y))

    while px < py:
        x += 1
        px += twory2
        if p1 < 0:
            p1 += ry2 + px
        else:
            y -= 1
            py -= tworx2
            p1 += ry2 + px - py
        all_points.extend(plot_ellipse_points(x_center, y_center,
x, y))

    p2 = (ry2 * (x + 0.5) * (x + 0.5)) + (rx2 * (y - 1) * (y -
1)) - (rx2 * ry2)
```

```
    while y > 0:
        y -= 1
        py -= tworx2
        if p2 > 0:
            p2 += rx2 - py
        else:
            x += 1
            px += twory2
            p2 += rx2 - py + px
        all_points.extend(plot_ellipse_points(x_center, y_center,
x, y))

    plt.gca().set_aspect('equal', adjustable='box')
    plt.title(f"Mid-Point Ellipse Drawing Algorithm with rx =
{rx}, ry = {ry}")
    plt.show()

    unique_points = list(set(all_points))
    for i, (x, y) in enumerate(sorted(unique_points)):
        print(f"Point {i+1}: ({x}, {y})")

if __name__ == "__main__":
    rx = int(input("Enter the x-radius of the ellipse: "))
    ry = int(input("Enter the y-radius of the ellipse: "))
    draw_ellipse(rx, ry)
```

Output:

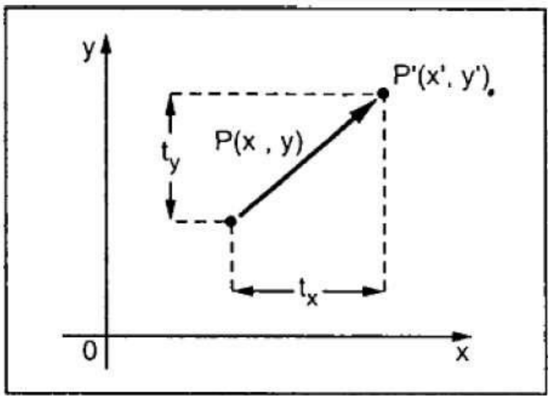


Practical#5

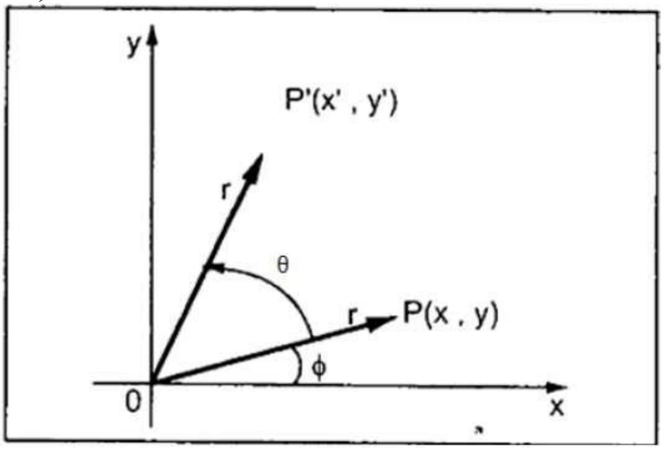
Objective: Implement 2D transformations of a rectangle.
Step By Step Procedural Algorithm
1. Enter the choice for transformation.
2. Perform the translation, rotation and scaling of 2D object.
3. Get the needed parameters for the transformation from the user.
4. Incase of rotation, object can be rotated about x or y axis.
5. Display the transmitted object in the screen along with new generated coordinates.

Theory:

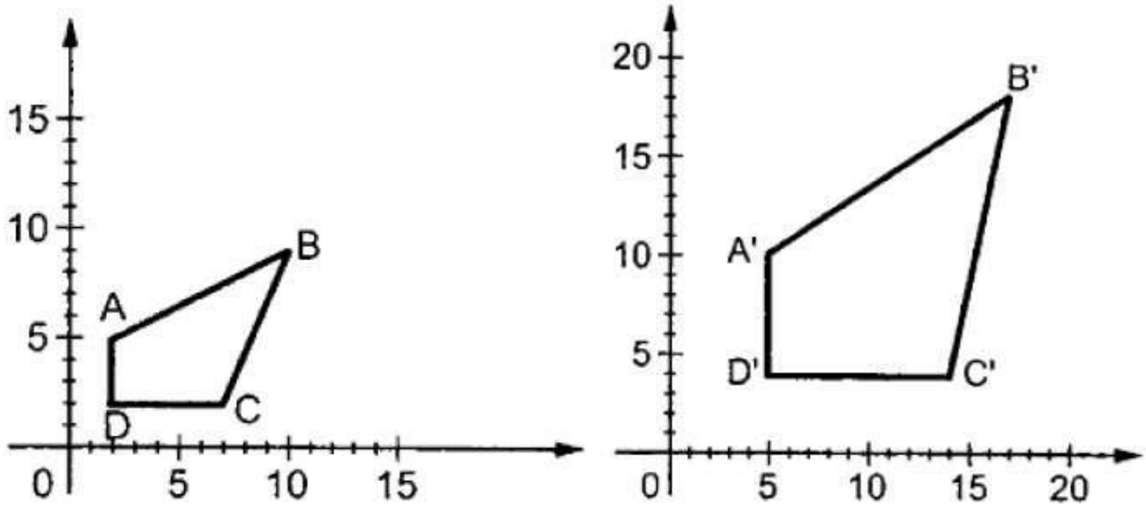
Translation
A translation moves an object to a different position on the screen. You can translate a point in 2D by adding translation coordinate (t_x, t_y) to the original coordinate X,YX,Y to get the new coordinate X′,Y′X′,Y′.



In rotation, we rotate the object at particular angle θ thetatheta from its origin. From the following figure, we can see that the point PX,YX,Y is located at angle φ from the horizontal X coordinate with distance r from the origin.
Let us suppose you want to rotate it at the angle θ. After rotating it to a new location, you will get a new point P’ X′,Y′X′,Y′.



o change the size of an object, scaling transformation is used. In the scaling process, you either expand or compress the dimensions of the object. Scaling can be achieved by multiplying the original coordinates of the object with the scaling factor to get the desired result.
Let us assume that the original coordinates are X,YX,Y, the scaling factors are (S_X, S_Y), and the produced coordinates are X′,Y′X′,Y′. This can be mathematically represented as shown below −
X' = X . S_X and Y' = Y . S_Y
The scaling factor S_X, S_Y scales the object in X and Y direction respectively. The above equations can also be represented in matrix form as below −
(X′Y′)=(XY)[Sx00Sy](X′Y′)=(XY)[Sx00Sy]
OR
P’ = P . S
Where S is the scaling matrix. The scaling process is shown in the following figure.



If we provide values less than 1 to the scaling factor S, then we can reduce the size of the object. If we provide values greater than 1, then we can increase the size of the object.

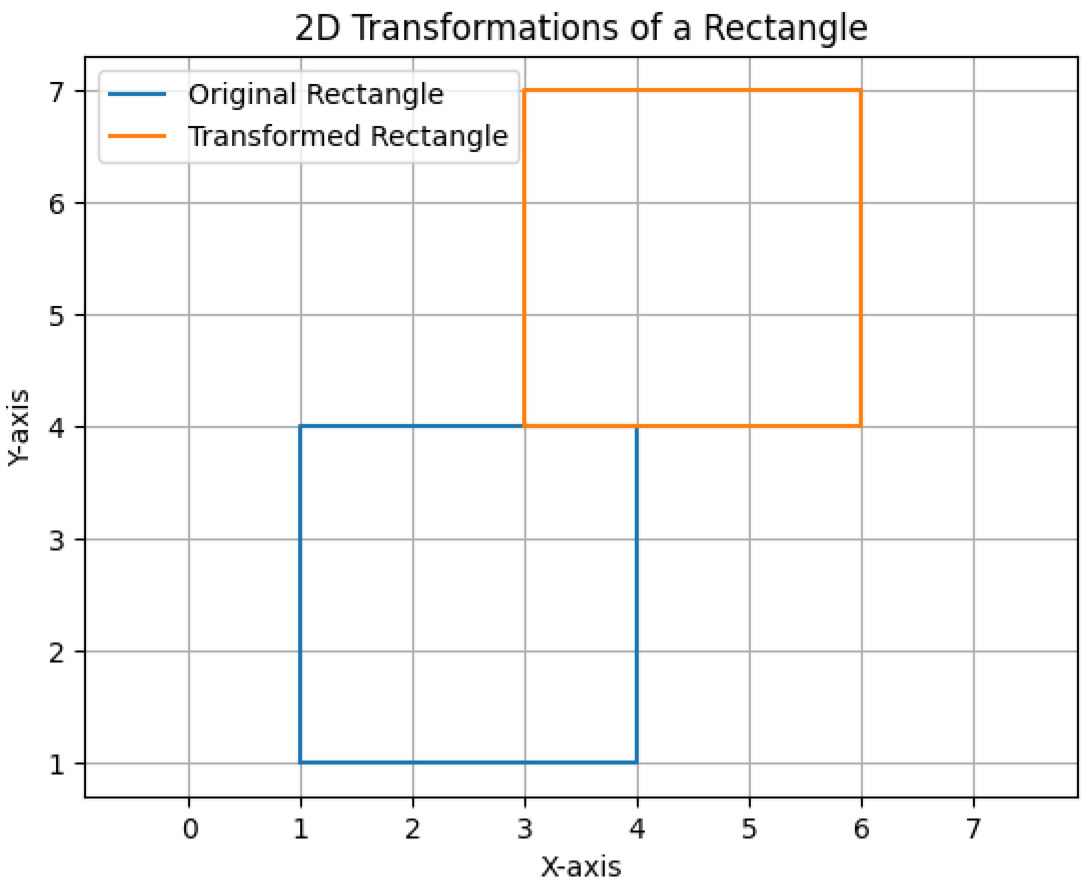
Code:

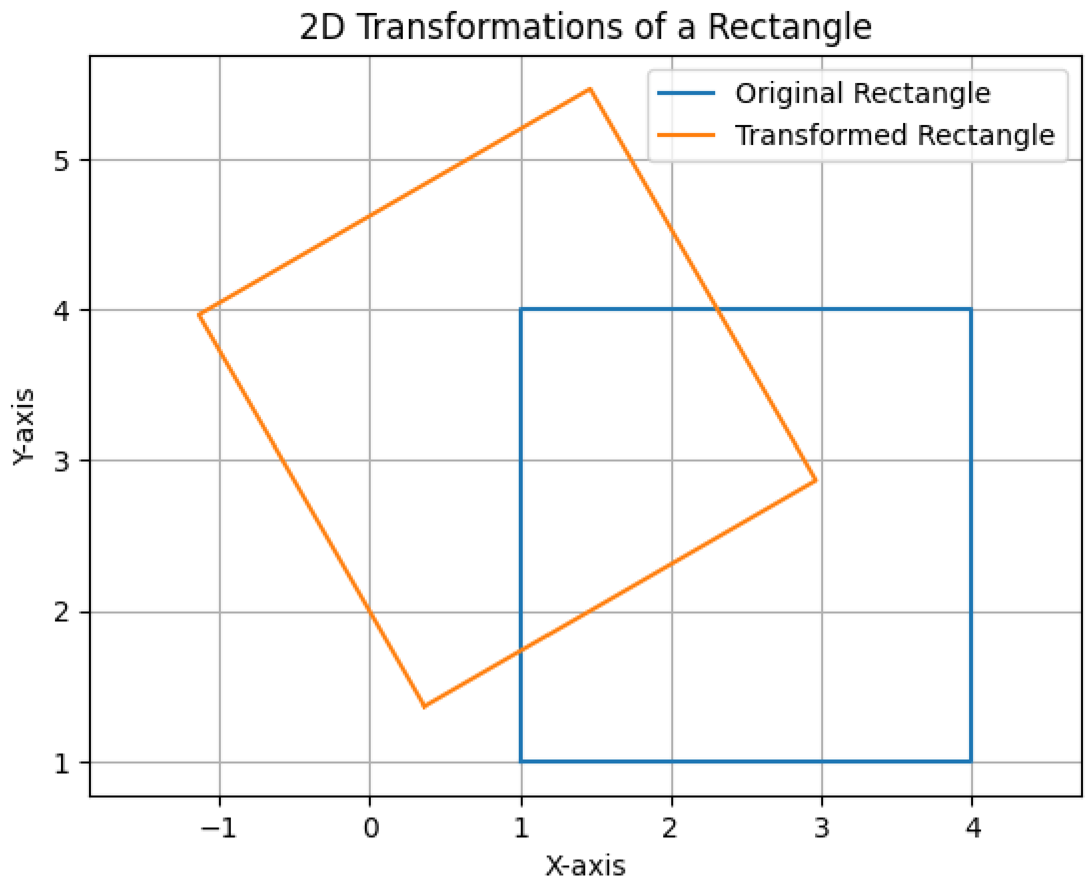
```
import matplotlib.pyplot as plt
import numpy as np

def get_transformation_choice():
    print("Choose transformation:")
    print("1. Translation")
    print("2. Rotation")
    print("3. Scaling")
    while True:
        try:
            choice = int(input("Enter choice (1/2/3): "))
            if choice in [1, 2, 3]:
                return choice
            else:
                print("Invalid choice. Please enter 1, 2, or 3.")
        except ValueError:
            print("Invalid input. Please enter a number
(1/2/3).")

def get_parameters(choice):
```

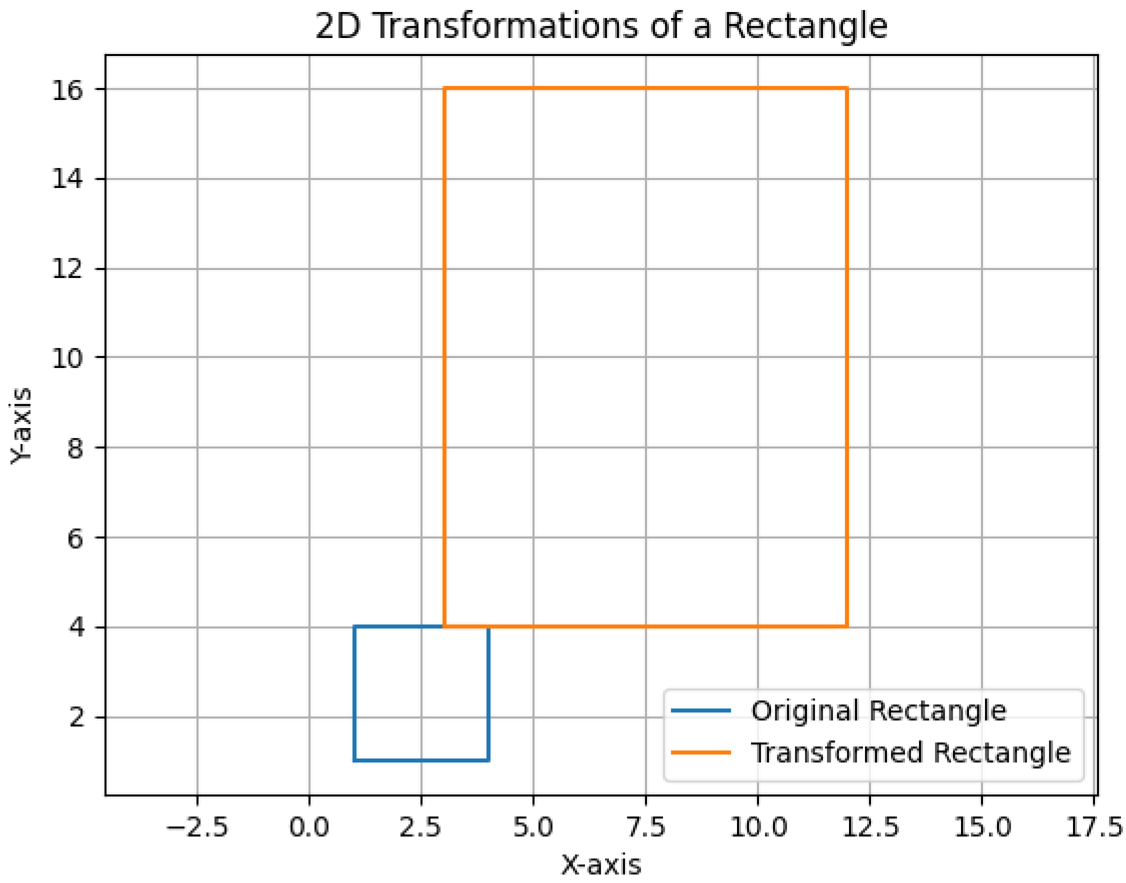
```
    if choice == 1:
        tx = float(input("Enter translation in x: "))
        ty = float(input("Enter translation in y: "))
        return (tx, ty)
    elif choice == 2:
        angle = float(input("Enter rotation angle in degrees: "))
        return (angle,)
    elif choice == 3:
        sx = float(input("Enter scaling factor in x: "))
        sy = float(input("Enter scaling factor in y: "))
        return (sx, sy)

def translate(rect, tx, ty):
    translation_matrix = np.array([[1, 0, tx],
                                   [0, 1, ty],
                                   [0, 0, 1]])
    return apply_transformation(rect, translation_matrix)

def rotate(rect, angle):
    rad = np.deg2rad(angle)
    rotation_matrix = np.array([[np.cos(rad), -np.sin(rad), 0],
                                [np.sin(rad),  np.cos(rad), 0],
                                [0,            0,           1]])
    return apply_transformation(rect, rotation_matrix)

def scale(rect, sx, sy):
    scaling_matrix = np.array([[sx, 0, 0],
                               [0, sy, 0],
                               [0, 0, 1]])
    return apply_transformation(rect, scaling_matrix)

def apply_transformation(rect, matrix):
    rect_homogeneous = np.hstack((rect, np.ones((rect.shape[0],
1))))
    transformed_rect = rect_homogeneous.dot(matrix.T)
    return transformed_rect[:, :2]

def display_rectangle(rect, transformed_rect):
    plt.figure()
    plt.plot(*zip(*np.vstack((rect, rect[0]))), label='Original
Rectangle')
    plt.plot(*zip(*np.vstack((transformed_rect,
transformed_rect[0]))), label='Transformed Rectangle')
    plt.legend()
    plt.xlabel('X-axis')
    plt.ylabel('Y-axis')
    plt.title('2D Transformations of a Rectangle')
    plt.grid(True)
    plt.axis('equal')
```

```
    plt.show()

def main():
    rect = np.array([[1, 1], [1, 4], [4, 4], [4, 1]])
    choice = get_transformation_choice()
    params = get_parameters(choice)

    if choice == 1:
        transformed_rect = translate(rect, *params)
    elif choice == 2:
        transformed_rect = rotate(rect, *params)
    elif choice == 3:
        transformed_rect = scale(rect, *params)

    display_rectangle(rect, transformed_rect)

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

Output:




2D Transformations of a Rectangle
Original Rectangle
Transformed Rectangle
Y-axis
X-axis
5
4
3
2
1
−1
0
1
2
3
4


2D Transformations of a Rectangle
Original Rectangle
Transformed Rectangle
Y-axis
X-axis
16
14
12
10
8
6
4
2
−2.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5