TITLE OF THE MINI PROJECT

2D TRANSFORMATION OF OBJECTS

DEPARTMENT NAME: COMPUTER SCIENCE AND ENGINEERING

COLLEGE NAME: ST. THOMAS' COLLEGE OF ENGINEERING & TECHNOLOGY

ADVISOR/SUPERVISOR: Dr. MOUSUMI DUTT

GROUP DETAILS:

NAME	ROLL NUMBER
1) SATYAM BHARDWAJ	07
2) RAHUL KUMAR	43
3) ANISH BHANU	06
4) CHANDAN KUMAR	42
5) SHIVAM UPADHYAY	71

CONTENTS

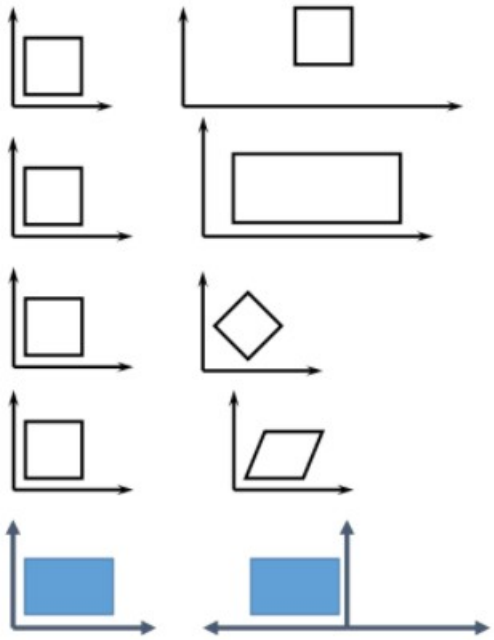
OBJECTIVES:-

- Transformation is a process of modifying and re-positioning the existing graphics. 2D Transformations take place in a two-dimensional plane.
- Transformations are helpful in changing the position, size, orientation, shape etc. of the object.
- 2D Translation is a process of moving an object from one position to another in a two-dimensional plane.

1. TRANSLATION
2. ROTATION
3. SCALING
4. REFLECTION
5. SHEARING

Elementary Transformation

- Affine transformations are usually combinations of four elementary transformations:
- 1: Translation
- 2: Scaling
- 3: Rotation
- 4: Shearing
- 5: Reflection



MOTIVATION:-

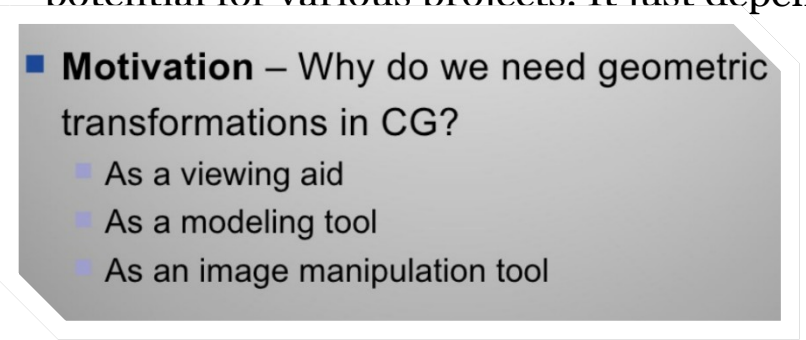
- **Sometimes objects exhibit certain symmetries, so only a part of it needs to be described, and the rest constructed by reflecting, rotating and translating the original part**
- **A designer may want to view and object from different vantage points, by rotating the object, or by moving a "synthetic camera" viewpoint.**
- **In animation, one or more objects must move relative to one another, so that their local co-ordinate systems must be shifted and rotated as the animation proceeds.**

In such cases, transformations play an important role to reposition the graphics on the screen and change their size or orientation. Transformation means changing some graphics into something else by applying rules. When a transformation takes place on a 2D plane, it is called 2D transformation. Transformation of an object in computer graphics is based

on mathematical theory and uses an important concept related to matrix theory. In general, there are two ways to represent the transformation of an object -

(i) Transformations are done by using the rules of co-ordinates axes
(ii) Transformations are done by representing in the matrix form.

2D Transformation in Computer Graphics could be applied in several projects, with a variety of goals, such as training, treatments, aid in learning, digital inclusion, simulations, among others. We realized the importance of visual programming and were motivated to do projects in this area. Computer Graphics is beyond beautiful and fun designs on screen. We can use all its potential for various projects. It just depends on our ability to create.



DDA (Digital Differential Analyzer) Algorithm

In computer graphics, the **DDA algorithm** is the simplest algorithm among all other line generation algorithms. Here, the **DDA** is an abbreviation that stands for **"Digital Differential Analyzer"**. It is an incremental method, i.e. it works by incrementing the source coordinate points according to the values of the slope generated.

Hence, we can define DDA as follows,

"DDA stands for Digital Differential Analyzer. This algorithm is incremental and is used for the rasterization of lines, triangles, and polygons."

Working of the DDA Algorithm

Suppose we have to **draw a line PQ with coordinates P (x1, y1) and Q (x2, y2)**.

1. First, Calculate **dx = (x2 - x1) and dy = (y2 - y1)**
2. Now calculate the slope **m = (dy / dx)**
3. Calculate the number of points to be plotted (i.e. **n**) by finding the maximum of **dx** and **dy**, i.e. **n = abs (max (dx , dy))**
   To draw an accurate line, more number of points are required. Therefore, the maximum of the two values is used here.
4. Now as the n is calculated, to know by how much each source point should be incremented,
   calculate $\mathbf{x_{inc}}$ and $\mathbf{y_{inc}}$ as follows: $\mathbf{x_{inc} = (dx / n)}$ **and** $\mathbf{y_{inc} = (dy / n)}$
5. Now we draw the points from **P** to **Q**. The successive points are calculated as follows: $\mathbf{(x_{next}, y_{next}) = (x + x_{inc}, y + y_{inc})}$
   Start plotting the points from **P** and stop when **Q** is reached. In case the incremented values are decimal, use the round off values.

Question: Draw a line from **A(2 , 2)** to **B(5 , 5)** using the **DDA algorithm**.

Solution:

```
Given:
x1 = 2 , y1 = 2
x2 = 5 , y2 = 6

Calculating:
dx  = (x2 - x1) = (5 - 2) = 3
dy  = (y2 - y1) = (6 - 2) = 4
n   = abs (max (dx , dy ) ) = abs (max (3 , 4) ) = 4

xinc = dx / n = 3/4 = 0.75
yinc = dy / n = 4 / 4 = 1
```

X	Y	$x = round(x + x_{inc})$	$y = y + y_{inc}$
2	2	2 + 0.75 = 2.75 = 3	2 + 1 = 3
3	3	3 + 0.75 = 3.75 = 4	3 + 1 = 4
4	4	4 + 0.75 = 4.75 = 5	4 + 1 = 5
5	5	5 + 0.75 = 5.75 = 6	5 + 1 = 6

Stop here as we have reached point **B**.

BRESENHAM'S LINE DRAWING ALGORITHM:-

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.

Given the starting and ending coordinates of a line, Bresenham Line Drawing Algorithm attempts to generate the points between the starting and ending coordinates.

Given-

Starting coordinates = (X_o, Y_o) Ending coordinates = (X_n, Y_n)

The points generation using Bresenham Line Drawing Algorithm involves the following steps-

Step 1:

Calculate ΔX and ΔY from the given input.
These parameters are calculated as-

$\Delta X = X_n - X_o$

$\Delta Y = Y_n - Y_o$

Step 2:

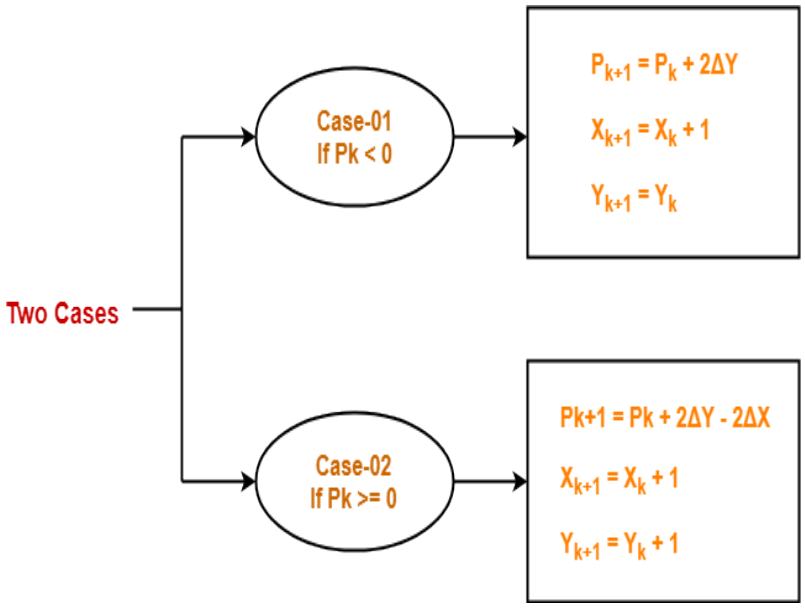
Calculate the decision parameter P_k.
It is calculated as-

$P_k = 2\Delta Y - \Delta X$

Step 3:

Suppose the current point is (X_k, Y_k) and the next point is (X_{k+1}, Y_{k+1}). Find the next point depending on the value of decision parameter Pk.

Follow the below two cases-



Step 4:

Keep repeating Step-03 until the end point is reached or number of iterations equals to $(\Delta X-1)$ times.

MIDPOINT CIRCLE DRAWING ALGORITHM:-

Given the center point and radius of circle, Mid-Point Circle Drawing Algorithm attempts to generate the points of one octant. The points for other octants are generated using the eight-symmetry property.
Given-

Centre point of Circle = (X_o, Y_o)
Radius of Circle = R
The points generation using Mid-Point Circle Drawing Algorithm involves the following steps-

Step 1:

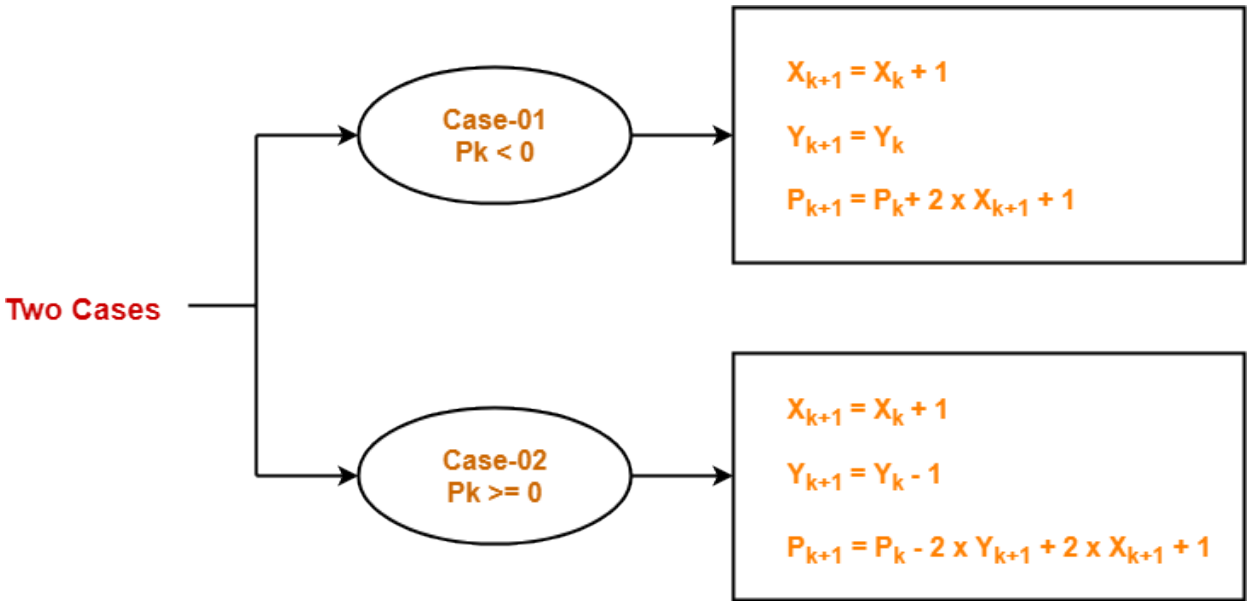
Assign the starting point coordinates (X_o, Y_o) as-
$X_o = 0$
$Y_o = R$

Step 2:

Calculate the value of initial decision parameter P_o as-
$P_o = 1 - R$

Step 3:

Suppose the current point is (X_k, Y_k) and the next point is (X_{k+1}, Y_{k+1}).
Find the next point of the first octant depending on the value of decision parameter P_k.
Follow the below two cases-



Step 4:

If the given centre point (X_o, Y_o) is not (0, 0), then do the following and plot the point-

$X_{plot} = X_c + X_o$
$Y_{plot} = Y_c + Y_o$
Here, (X_c, Y_c) denotes the current value of X and Y coordinates.

Step 5:

Keep repeating Step-03 and Step-04 until $X_{plot} >= Y_{plot}$.

Step 6:

Step 5 generates all the points for one octant.
To find the points for other seven octants, follow the eight-symmetry property of circle.

MIDPOINT ELLIPSE DRAWING ALGORITHM:-

This is an incremental method for scan converting an ellipse that is cantered 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 of that can be used to decide if the midpoint between two candidate pixels is inside or outside the ellipse:

Input (r_x,r_y) cantered at origin and the first point is $(x_o,y_o)=(0,r_y)$

Region 1:

$P1_o=r_y^2-r_x^2r_y+1/4r_x^2$
If $p1_k<0$ then (X_{k+1},Y_k)
and $p1_{k+1}=p1_k+2r_y^2x_{k+1}+r_y^2$
Else (X_{k+1},Y_{k-1})
$P1_{k+1}=p1_k+2r_y^2x_{k+1}-2r_x^2y_{k+1}+r_y^2$
And continue till $2r_y^2x>=2r_x^2y$

Region 2:

Here (x_o,y_o) is the last point calculated in region 1 $P2_o=r_y^2(x_o+1/2)^2+r_x^2(y_o-1)^2-r_x^2r_y^2$
If $p2_k>0$ then (x_k,y_{k-1}) and
$P2_{k+1}=p2_k-2r_x^2y_{k+1}+r_x^2$
Else (x_{k+1},y_{k-1})
$P2_{k+1}=p2_k+2r_y^2x_{k+1}-2r_x^2y_{k+1}+r_x^2$
Till $y=0$, i.e., last point is $(r_x,0)$

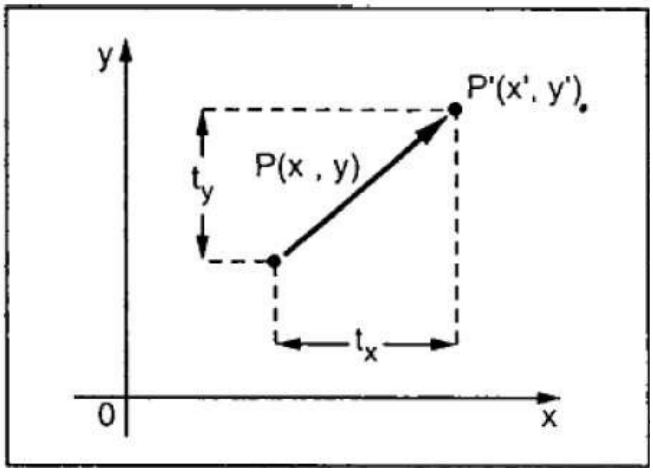
TRANSLATION:-

It is the straight-line movement of an object from one position to another is called Translation. Here the object is positioned from one coordinate location to another.
To translate a point from coordinate position (x, y) to another (x1 y1), we add algebraically the translation distances Tx and Ty to original coordinate.

$x1 = x + T_x$

$y1 = y + T_y$

The translation pair (T_x, Ty) is called as shift vector.
Translation is a movement of objects without deformation. Every position or point is translated by the same amount. When the straight line is translated, then it will be drawn using endpoints.
For translating polygon, each vertex of the polygon is converted to a new position. Similarly, curved objects are translated. To change the position of the circle or ellipse its centre coordinates are transformed, then the object is drawn using new coordinates.
Let P is a point with coordinates (x, y). It will be translated as (x1, y1).



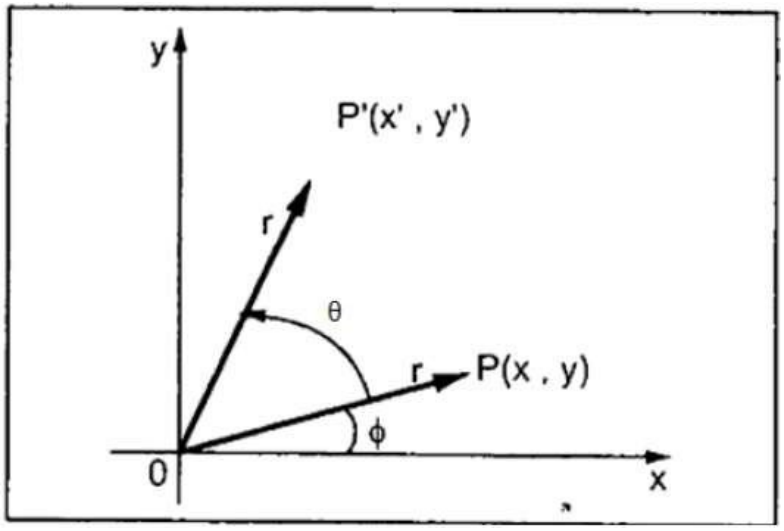
Matrix for Translation:

$$\begin{vmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{vmatrix} \quad \text{Or} \quad \begin{vmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{vmatrix}$$

ROTATION:-

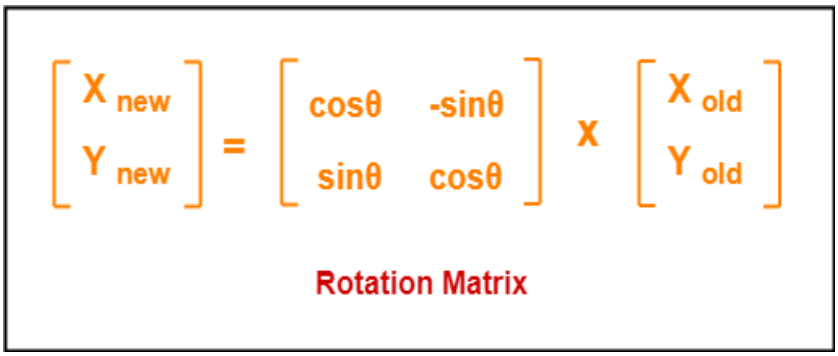
2D Rotation is a process of rotating an object with respect to an angle in a two-dimensional plane.

Consider a point object O has to be rotated from one angle to another in a 2D plane. Now let, (1) Initial coordinates of the object O = (X_{old}, Y_{old}) (2) Initial angle of the object O with respect to origin = Φ (3) Rotation angle = θ (4) New coordinates of the object O after rotation = (X_{new}, Y_{new})

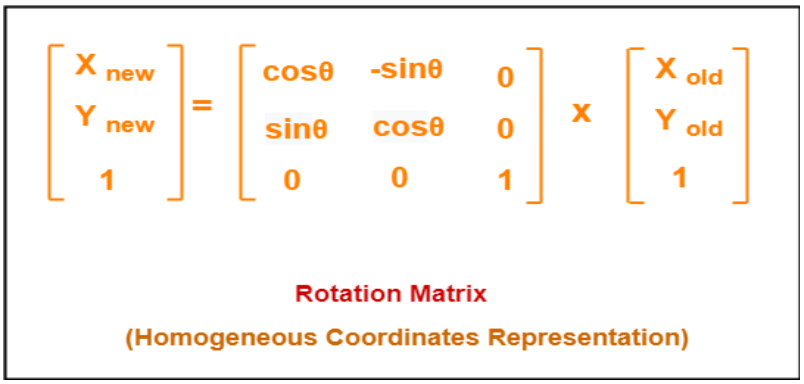


This rotation is achieved by using the following rotation equations-
$X_{new} = X_{old} \times \cos\theta - Y_{old} \times \sin\theta$, $Ynew = X_{old} \times \sin\theta + Y_{old} \times \cos\theta$

In Matrix form, the above rotation equations may be represented as-



For homogeneous coordinates, the above rotation matrix may be represented as a 3 x 3 matrix as-



Matrix for rotation:

Clockwise:

Matrix for homogeneous co-ordinate rotation

$$R = \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Anti-clockwise:

Matrix for homogeneous co-ordinate rotation

$$R = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

SCALING:-

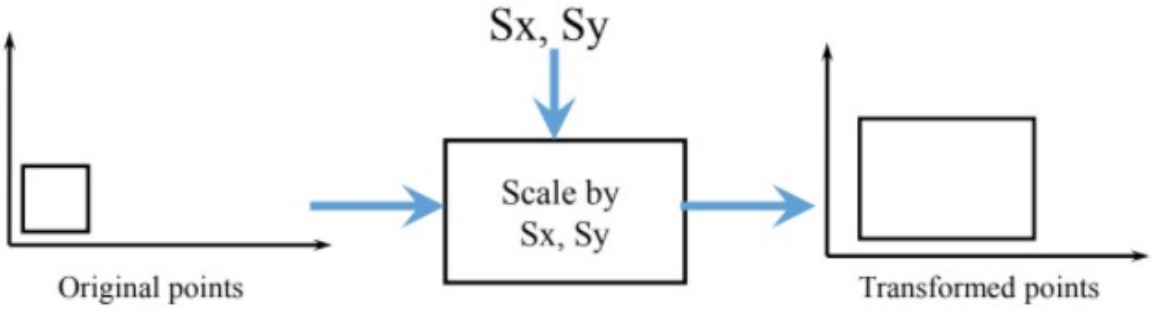
It is used to alter or change the size of objects.

The change is done using scaling factors.
There are two scaling factors, i.e., Sx in x direction Sy in y-direction.
If the original position is x and y, scaling factors are Sx and Sy then the value of coordinates after scaling will be x1 and y1.
If the picture to be enlarged to twice its original size then Sx = Sy = 2. If Sx and Sy are not equal then scaling will occur but it will elongate or distort the picture.
If scaling factors are less than one, then the size of the object will be reduced.
If scaling factors are higher than one, then the size of the object will be enlarged.
If Sx and Sy are equal it is also called as Uniform Scaling. If not equal then called as Differential Scaling. If scaling factors with values less than one will move the object closer to coordinate origin, while a value higher than one will move coordinate position farther from origin.

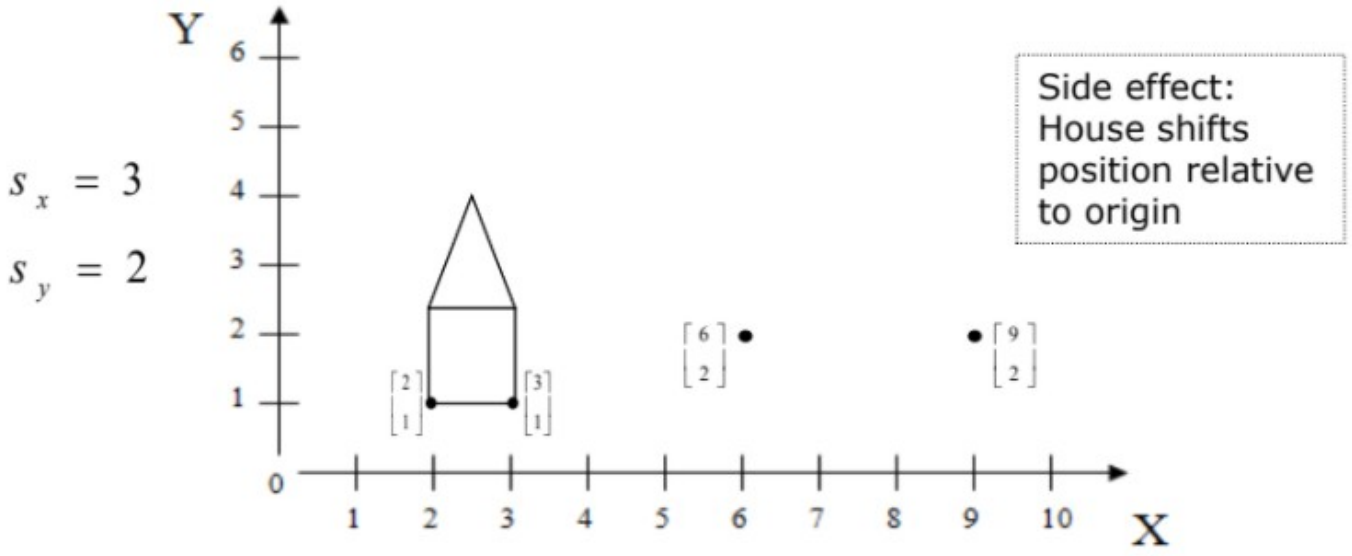
Enlargement: If $T1 = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$, If (x1 y1) is original position and T1

is translation vector then (x2 y2) are coordinated after scaling

$$[x_2 y_2] = [\ x_1 y_1] \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} = [2x_1\ 2y_1]$$



1.

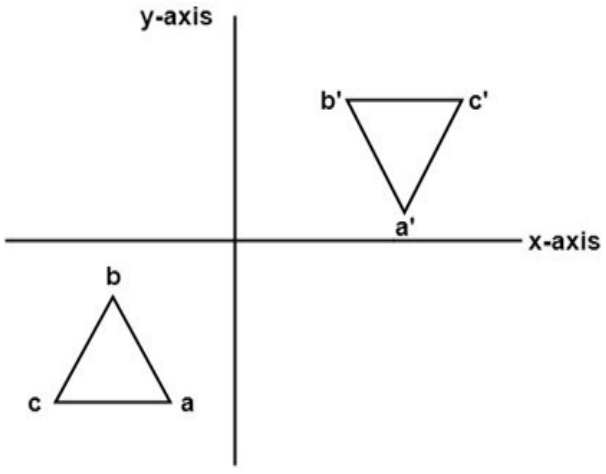


R

eflection about an axis perpendicular to xy plane and passing through origin:

The matrix of this transformation is given below -

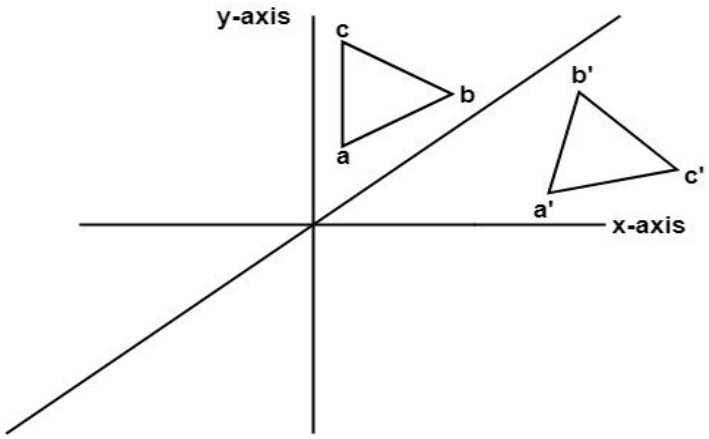
$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



In this value of x and y both will be reversed. This is also called as half revolution about the origin.

2. Reflection about line y = x: The object may be reflected about line y = x with the help of following transformation matrix -

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



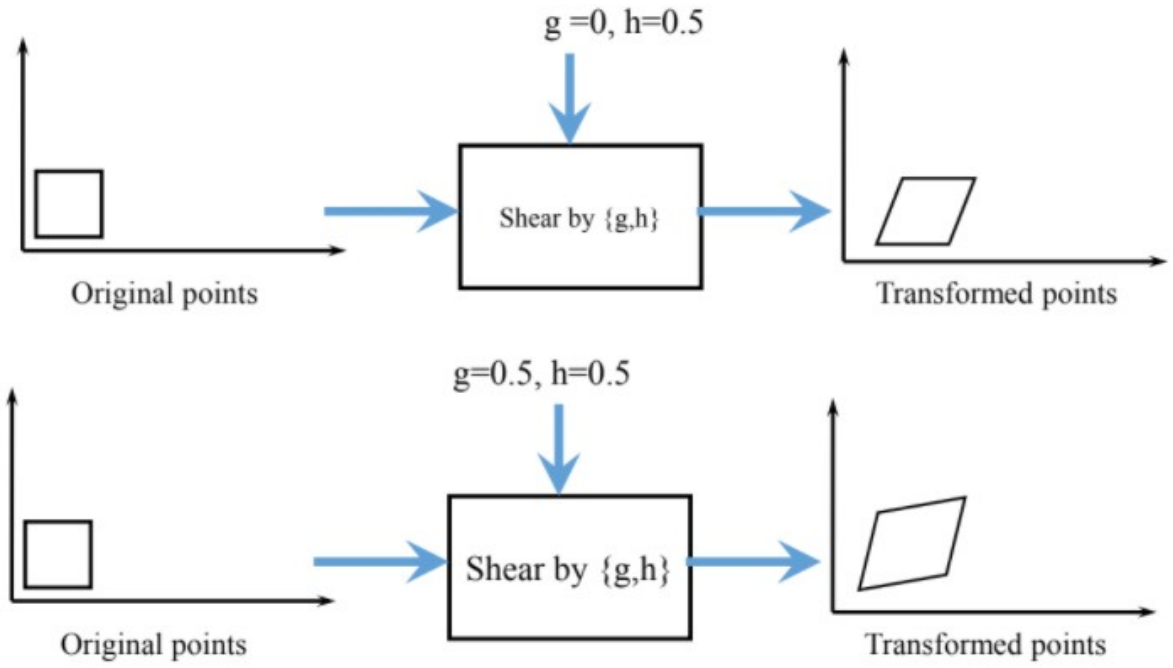
First of all, the object is rotated at 45°. The direction of rotation is clockwise. After its reflection is done concerning x-axis. The last step is the rotation of y=x back to its original position that is counterclockwise at 45°.

SHEARING:-

It is change in the shape of the object. It is also called as deformation. Change can be in the x-direction or y-direction or both directions in case of 2D. If shear occurs in both directions, the object will be distorted. But in 3D shear can occur in three directions.

Matrix for shear:

$$\begin{pmatrix} 1 & 0 & a & 0 \\ 0 & 1 & b & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



SOU

RCE CODE:-

MIDPOINT CIRCLE DRAWING ALGORITHM:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
def solver(center, r):
    x = 0
    y = r
    p = 1 - r
    points = []
    points.extend(
        [
            (center.x, y + center.y),
            (center.x, -y + center.y),
            (y + center.x, x + center.y),
            (-y + center.x, x + center.y),
        ]
    )
    while x <= y:
        x += 1
        if p < 0:
            p += 2 * x + 1
        else:
            y -= 1
            p += 2 * x - 2 * y + 1
```

```
        if x > y:
            break
        points.extend(
            [
                (x + center.x, y + center.y),
                (-x + center.x, y + center.y),
                (x + center.x, -y + center.y),
                (-x + center.x, -y + center.y),
            ]
        )
        if x != y:
            points.extend(
                [
                    (y + center.y, x + center.x),
                    (-y + center.y, x + center.x),
                    (y + center.y, -x + center.x),
                    (-y + center.y, -x + center.x),
                ]
            )
    # points.sort(key=lambda x: x[0])
    points.sort(key=lambda i: 0 if i[1] == 0 else -1 / i[1])
    return points

def circle(c1, c2, r):
    x = list(map(lambda y: y[0], solver(Point(c1, c2), r)))
    y = list(map(lambda x: x[1], solver(Point(c1, c2), r)))
    plt.plot(x, y)

    # naming the x axis
    plt.xlabel("x - axis")
    # naming the y axis
    plt.ylabel("y - axis")

    # giving a title to my graph
    plt.title("CIRCLE ")

    # function to show the plot
    # plt.show()
    plt.plot(
        x,
        y,
        color="green",
        linestyle="dashed",
        linewidth=2,
        marker="o",
        markerfacecolor="blue",
        markersize=8,
    )

    # setting x and y axis range
    plt.ylim(-61, 61)
```

```
plt.xlim(-61, 61)

# naming the x axis
plt.xlabel("x - axis")
# naming the y axis
plt.ylabel("y - axis")

# giving a title to my graph
plt.title("Circle ")

# function to show the plot
plt.show()
return x, y
```

MIDPOINT ELLIPSE DRAWING ALGORITHM:

```
def ellipse(rx, ry, xc, yc):
    x = 0
    y = ry

    # Initial decision parameter of region 1
    d1 = (ry * ry) - (rx * rx * ry) + (0.25 * rx * rx)
    dx = 2 * ry * ry * x
    dy = 2 * rx * rx * y

    # For region 1
    while dx < dy:

        # Print points based on 4-way symmetry
        print("(", x + xc, ",", y + yc, ")")
        print("(", -x + xc, ",", y + yc, ")")
        print("(", x + xc, ",", -y + yc, ")")
        print("(", -x + xc, ",", -y + yc, ")")

        # Checking and updating value of
        # decision parameter based on algorithm
        if d1 < 0:
            x += 1
            dx = dx + (2 * ry * ry)
            d1 = d1 + dx + (ry * ry)
        else:
            x += 1
            y -= 1
            dx = dx + (2 * ry * ry)
            dy = dy - (2 * rx * rx)
            d1 = d1 + dx - dy + (ry * ry)

    # Decision parameter of region 2
    d2 = (
        ((ry * ry) * ((x + 0.5) * (x + 0.5)))
        + ((rx * rx) * ((y - 1) * (y - 1)))
        - (rx * rx * ry * ry)
    )

    # Plotting points of region 2
    while y >= 0:

        # printing points based on 4-way symmetry
        print("(", x + xc, ",", y + yc, ")")
        print("(", -x + xc, ",", y + yc, ")")
        print("(", x + xc, ",", -y + yc, ")")
```

```
print("(", -x + xc, ",", -y + yc, ")")

# Checking and updating parameter
# value based on algorithm
if d2 > 0:
    y -= 1
    dy = dy - (2 * rx * rx)
    d2 = d2 + (rx * rx) - dy
else:
    y -= 1
    x += 1
    dx = dx + (2 * ry * ry)
    dy = dy - (2 * rx * rx)
    d2 = d2 + dx - dy + (rx * rx)
```

TRANSLATION:

```
def translate(original, Tx, Ty):
    TranMatrix = np.zeros((3,3))
    TranMatrix[0][0]=1
    TranMatrix[0][2]=Tx
    TranMatrix[1][1]=1
    TranMatrix[1][2]=Ty
    TranMatrix[2][2]=1

    translated=matrix_mul(TranMatrix, original)
    l = []
    for i in range(len(translated)):
        l.append([])
        for j in range(len(translated[0])-1):
            l[i].append(translated[i][j])
    print(l)
    X = np.array(original + l)

    # X = np.array([[1,1], [2,2.5], [3, 1], [8, 7.5], [7, 9],
[9, 9]])
    r = 'red'
    b = 'blue'
    Y = []
    Y += len(original)*[r]
    Y += len(original)*[b]

    plt.figure()
    plt.scatter(X[:, 0], X[:, 1], s = 170, color = Y[:])

    t1 = plt.Polygon(X[:3,:], color=Y[0])
    plt.gca().add_patch(t1)

    t2 = plt.Polygon(X[3:6,:], color=Y[len(original)+1])
    plt.gca().add_patch(t2)

   plt.show()
```

BRESENHAM'S LINE DRAWING ALGORITHM:

```
def brnhms(x1, y1, x2, y2):
    x_data = []
    y_data = []
    x_data.append(x1)
    y_data.append(y1)
    dx = abs(x2 - x1)
    dy = abs(y2 - y1)
    if (x2 > x1):
        xs = 1
    else:
        xs = -1
    if (y2 > y1):
        ys = 1
    else:
        ys = -1

    if (dx > dy):
        p1 = 2 * dy - dx
        while (x1 != x2):
            x1 += xs
            if (p1 >= 0):
                y1 += ys
                p1 -= 2 * dx
            p1 += 2 * dy
            x_data.append(x1)
            y_data.append(y1)

    elif (dy >= dx):
        p1 = 2 * dx - dy
        while (y1 != y2):
            y1 += ys
```

```
            if (p1 >= 0):
                x1 += xs
                p1 -= 2 * dy
            p1 += 2 * dx
            x_data.append(x1)
            y_data.append(y1)
    return (x_data,y_data)

def bres(a, b, c, d):
    # from dda_line_drawing import *
    # x, y = dda(2, 3, 10, 11)
    x, y = brnhms(a, b, c, d)
    for i in range( len(x)):
        print(f"x = {x[i]}  y = {y[i]}")
    plt.plot(
        x,
        y,
        color="green",
        linestyle="dashed",
        linewidth=2,
        marker="o",
        markerfacecolor="blue",
        markersize=8,
    )

    # setting x and y axis range
    plt.ylim(0, 50)
    plt.xlim(0, 48)

    # naming the x axis
    plt.xlabel("x - axis")
    # naming the y axis
```

```
    plt.ylabel("y - axis")

    # giving a title to my graph
    plt.title("Bresenham's Visualizer")

    # function to show the plot
    plt.show()
   return x, y
```

ROTATION:

```
def rotate_around_point_highperfClock(xy, radians, origin=(0, 0)):
    """Rotate a point around a given point.
    I call this the "high performance" version since we're caching some
    values that are needed >1 time. It's less readable than the previous
    function but it's faster.
    """
    x, y = xy
    offset_x, offset_y = origin
    adjusted_x = (x - offset_x)
    adjusted_y = (y - offset_y)
    cos_rad = math.cos(radians)
    sin_rad = math.sin(radians)
    qx = offset_x + cos_rad * adjusted_x + sin_rad * adjusted_y
    qy = offset_y + -sin_rad * adjusted_x + cos_rad * adjusted_y
    return qx, qy

def rotateAntiClock(point, angle, origin = (0,0)):
    """
    Rotate a point counterclockwise by a given angle around a given origin.

    The angle should be given in radians.
    """
    ox, oy = origin
    px, py = point

    qx = ox + math.cos(angle) * (px - ox) - math.sin(angle) * (py - oy)
    qy = oy + math.sin(angle) * (px - ox) + math.cos(angle) * (py - oy)
    return qx, qy
#dir = clockwise or anticlockwise

def rotate(x, y, radians, dir, origin=(0,0)):
    or1 = x
    or2 = y
    l1 = []
```

```
    l2 = []
    if dir == "c":
        for i in range(len(x)):
            a,b = rotate_around_point_highperfClock((x[i], y[i]),
math.radians(radians), origin)
            l1.append(a)
            l2.append(b)
    else:
        for i in range(len(x)):
            a,b = rotateAntiClock((x[i], y[i]), math.radians(radians),
origin)
            l1.append(a)
            l2.append(b)

    x, y = l1, l2
    for i in range( len(x)):
        print(f"x = {x[i]}  y = {y[i]}")
    plt.rc('grid', linestyle="-", color='black')
    plt.scatter(x, y, c ="blue")
    plt.scatter(or1, or2, c ="green")
    plt.xlabel(" X axis --->")
    plt.ylabel(" Y axis --->")
    plt.grid()
    plt.show()
   return x, y
```

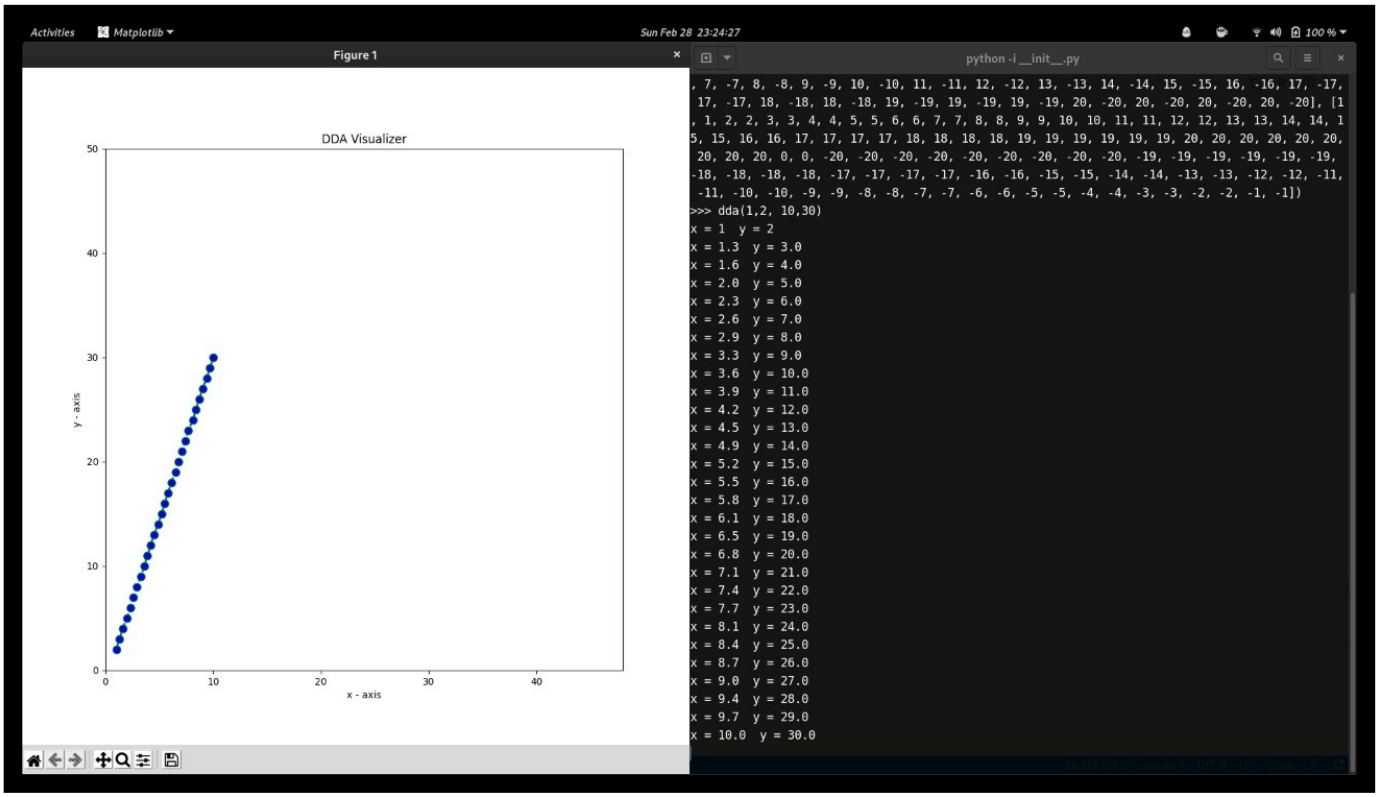
SCALING:

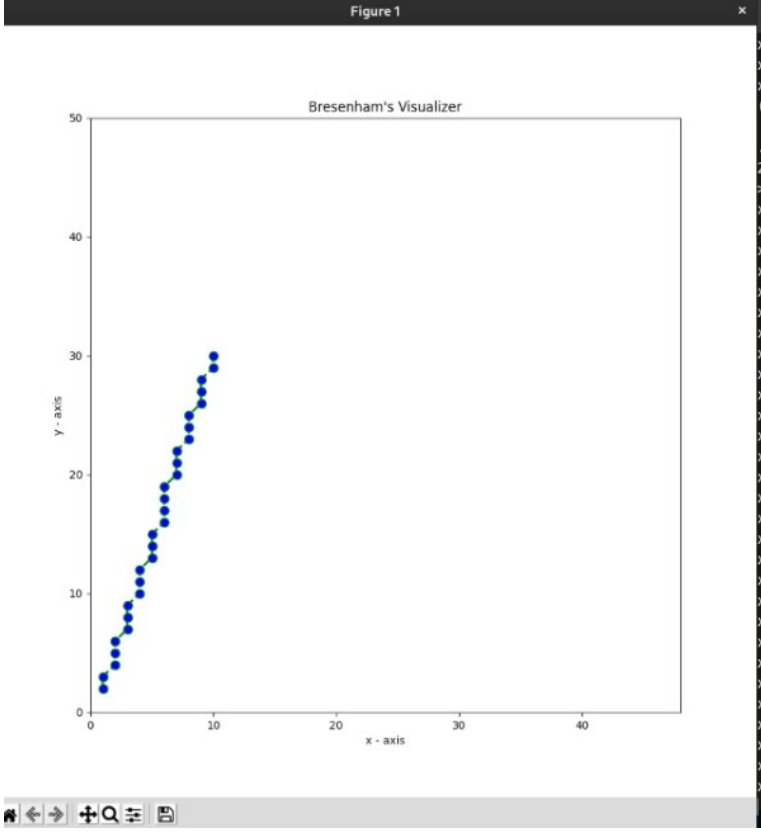
```
def scaling(original, scaleFactor):
    scaleFactor = 2
    scaled = copy.deepcopy(original)
    for i in range(len(scaled[0])):
        scaled[0][i]=scaled[0][i]*scaleFactor
        scaled[1][i]=scaled[1][i]*scaleFactor
    # X = np.array(original)
    X = np.array(original + scaled)

    # X = np.array([[1,1], [2,2.5], [3, 1], [8, 7.5], [7, 9], [9, 9]])
    r = 'red'
    b = 'blue'
```

```
Y = []
Y += len(original)*[r]
Y += len(original)*[b]

#red is original colour
#blue is scaled colour
plt.figure()
plt.scatter(X[:, 0], X[:, 1], s = 170, color = Y[:])

t1 = plt.Polygon(X[:3,:], color=Y[0])
plt.gca().add_patch(t1)

t2 = plt.Polygon(X[3:6,:], color=Y[3])
plt.gca().add_patch(t2)

plt.show()
```

OUTPUT-1

DDA LINE DRAWING :






```
python -i __init__.py
x = 9.4  y = 28.0
x = 9.7  y = 29.0
x = 10.0  y = 30.0
([1, 1.3, 1.6, 2.0, 2.3, 2.6, 2.9, 3.3, 3.6, 3.9, 4.2, 4.5, 4.9, 5.2, 5.5, 5.8, 6.1, 6.5, 6.8,
 7.1, 7.4, 7.7, 8.1, 8.4, 8.7, 9.0, 9.4, 9.7, 10.0], [2, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10
.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0, 19.0, 20.0, 21.0, 22.0, 23.0, 24.0, 25.0,
26.0, 27.0, 28.0, 29.0, 30.0])
>>> bres(1,2, 10,30)
x = 1  y = 2
x = 1  y = 3
x = 2  y = 4
x = 2  y = 5
x = 2  y = 6
x = 3  y = 7
x = 3  y = 8
x = 3  y = 9
x = 4  y = 10
x = 4  y = 11
x = 4  y = 12
x = 5  y = 13
x = 5  y = 14
x = 5  y = 15
x = 6  y = 16
x = 6  y = 17
x = 6  y = 18
x = 6  y = 19
x = 7  y = 20
x = 7  y = 21
x = 7  y = 22
x = 8  y = 23
x = 8  y = 24
x = 8  y = 25
x = 9  y = 26
x = 9  y = 27
x = 9  y = 28
x = 10  y = 29
x = 10  y = 30
```

OUTPUT-2

Bresenham's Line Drawing

OUTPUT-3

Rotation of a line about Origin (0,0) 90 degrees in clockwise direction.

OUTPUT-4

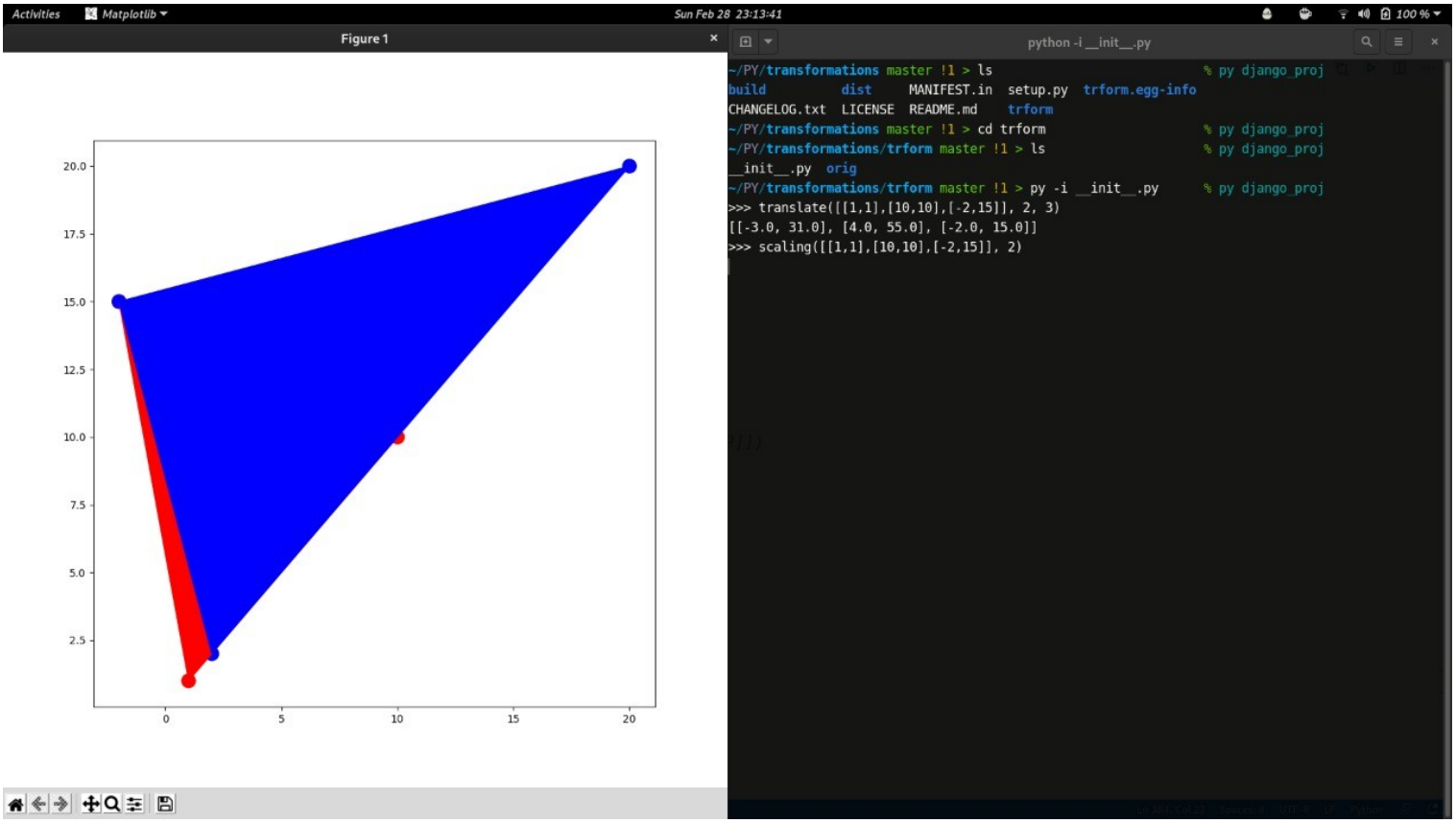
Rotation of a line about origin (0,0) 90 degrees in anticlockwise direction.

OUTPUT-5

Drawing a circle with centre (0,0) and radius 20 :

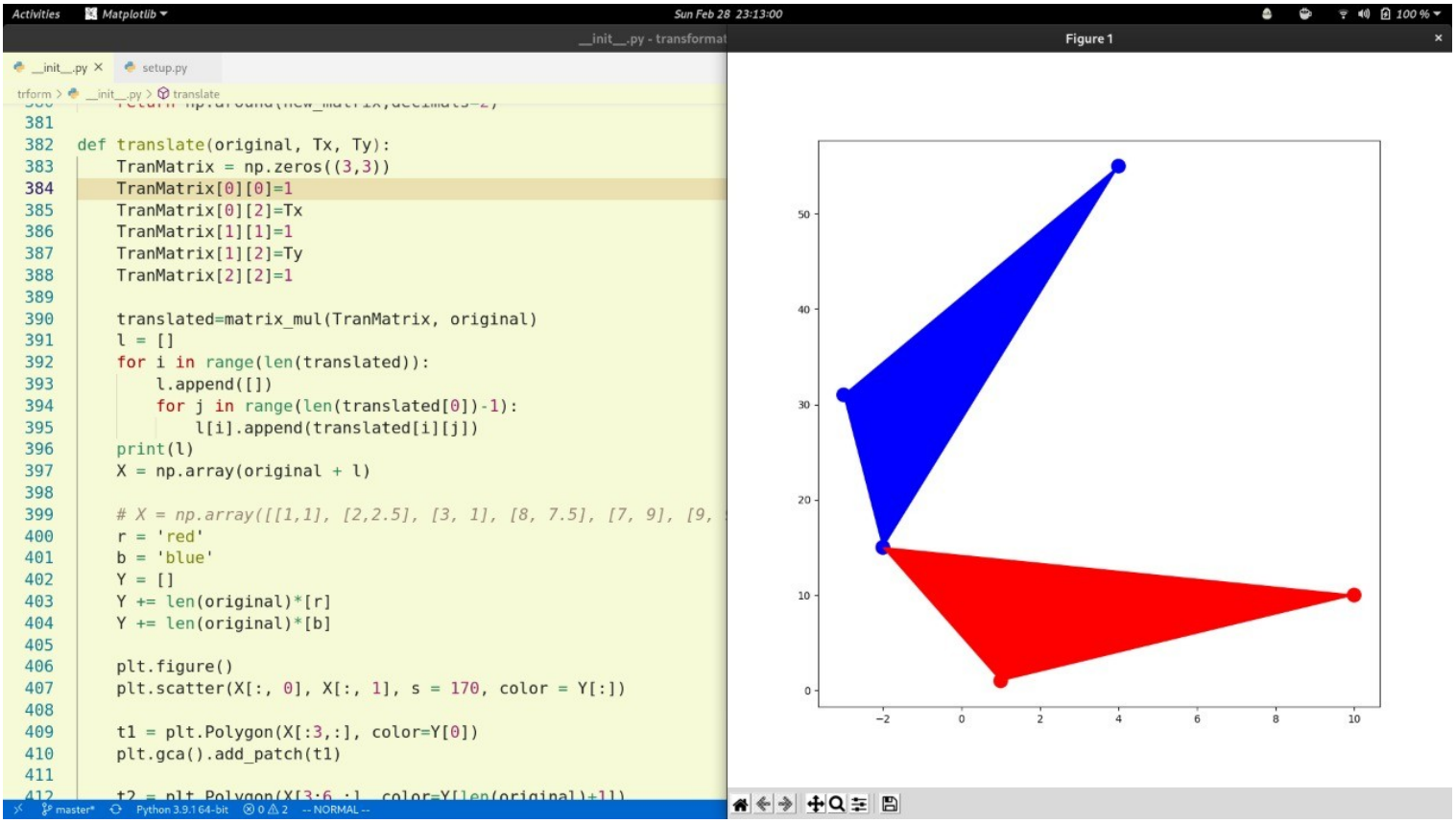
OUTPUT-6

SCALING :



OUTPUT – 7

Translate a polygon:



Bibliography:

References:

1) Hearn, Baker – “Computer Graphics (C version 2nd Ed.)” – Pearson education
2) https://www.javatpoint.com/computer-graphics-programs
3) https://www.gatevidyalay.com/computer-graphics/
4) **Computer Graphics PPT by Dr. Mosumi Dutt**
5) **www.stackoverflow.com**

GitHub link:

https://github.com/RAMESSESII2/transformations/tree/master

https://test.pypi.org/project/transform/0.2/