



Computer Graphics Lab

Name: _____

Roll no. – 2K14/SE/____

Software Engineering

Section-B

EXPERIMENT 1

- **AIM :-** Write a program to implement DDA Line Algorithm.
- **Description of Aim & Related Theory:-**The digital differential analyzer(DDA) is a scan conversion line algorithm based on calculating either Δy or Δx . We sample the line at unit intervals in one coordinate and determine corresponding integer values nearest the line path for other coordinate.

If slope is less than or equal to 1, we sample at unit x intervals ($\Delta x=1$) and compute each successive y value as

$$Y_{k+1}=y_k+m$$

For lines with a positive slope greater than 1, we sample at unit y intervals ($\Delta y=1$) and calculate each succeeding x value as

$$X_{k+1}=x_k+1/m$$

By using above equations we can calculate pixels positions along line.

- **Algorithm :**

Digital Differential Analyzer Line Drawing Method

1. Read the end points of a line (x_1, y_1) & (x_2, y_2).
2. If the points are same, plot the points & exit (Enter continue with the calculation).
3. $\Delta x = |x_2 - x_1|$ & $\Delta y = |y_2 - y_1|$
4. if $\text{abs}(\Delta x) \geq \text{abs}(\Delta y)$
then Steps = Δx
Else
Steps = Δy
5. x increment = $\Delta x / \text{steps}$.
6. y increment = $\Delta y / \text{steps}$
7. Initialize (x, y) with (x_1, y_1)
 $x = x_1$
 $y = y_1$
8. Plot the rounded coordinate (x, y)
9. Initialize counter K=1
10. Start the loop
 $x = x + x \text{ increment}$

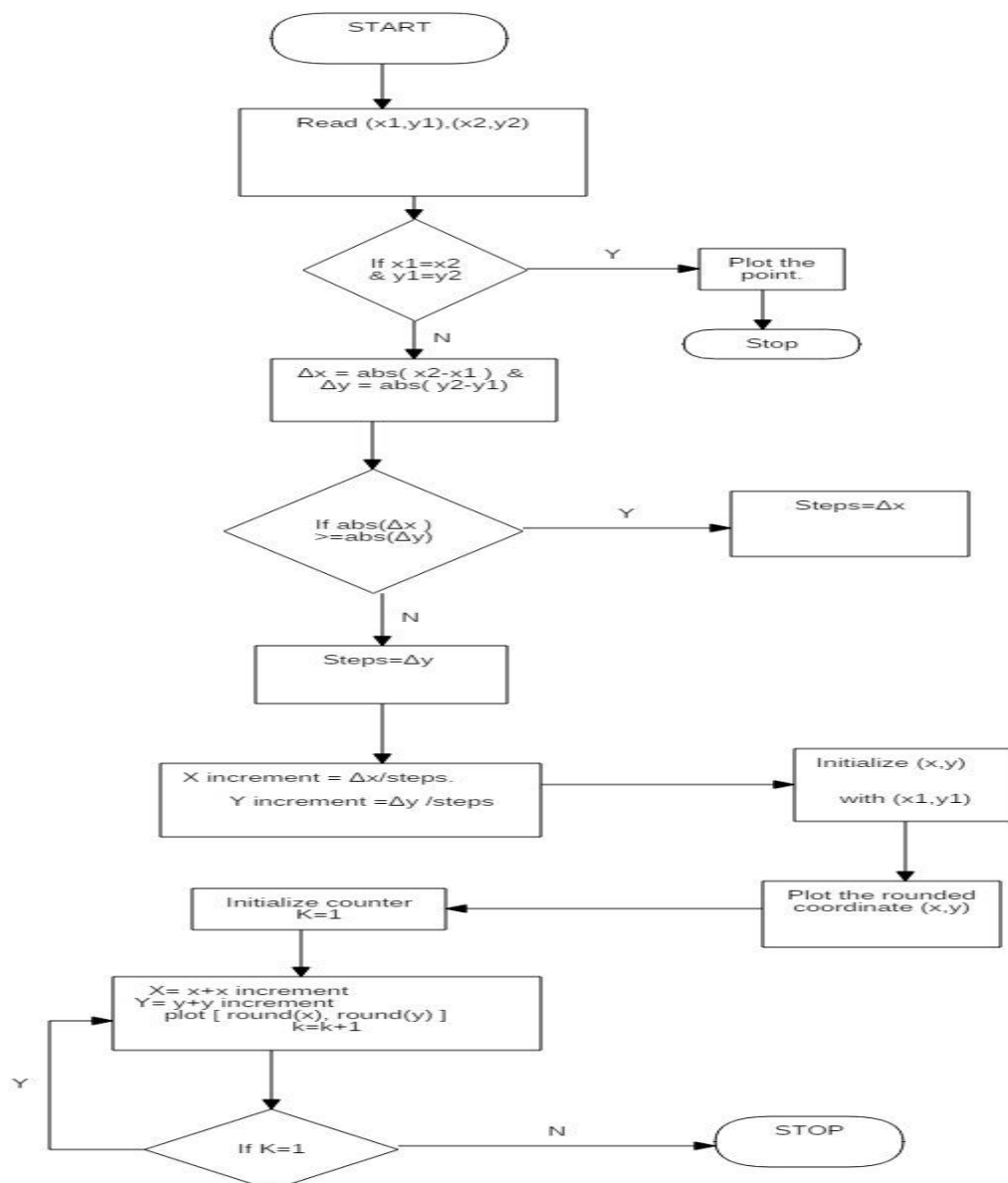
$y = y + y \text{ increment}$

Plot the rounded coordinate(x, y)

11. Continue the loop till the counter = steps

12.stop.

➤ **Flow Chart:**



➤ **Code:**

```
#include<stdlib.h>

#include<iostream.h>

#include<graphics.h>

#include<stdio.h>

#include<dos.h>

#include<conio.h>


#define round(val) (int)(val + 0.5)


int main(){

    int gd=DETECT, gm;

    initgraph(&gd, &gm, "/TURBOC3/BGI");

    void line_dda(int, int, int, int);

    int xa, xb, ya, yb;

    printf("Enter two coordinates\n");

    scanf("%d %d", &xa, &ya);

    scanf("%d %d", &xb, &yb);

    line_dda(xa, ya, xb, yb);

    getch();

    closegraph();

    return 0;

}


void line_dda(int xa, int ya, int xb, int yb)

{

    int Dx = xb - xa, Dy = yb - ya, steps, k;

    float xin, yin, X = xa, Y = ya;

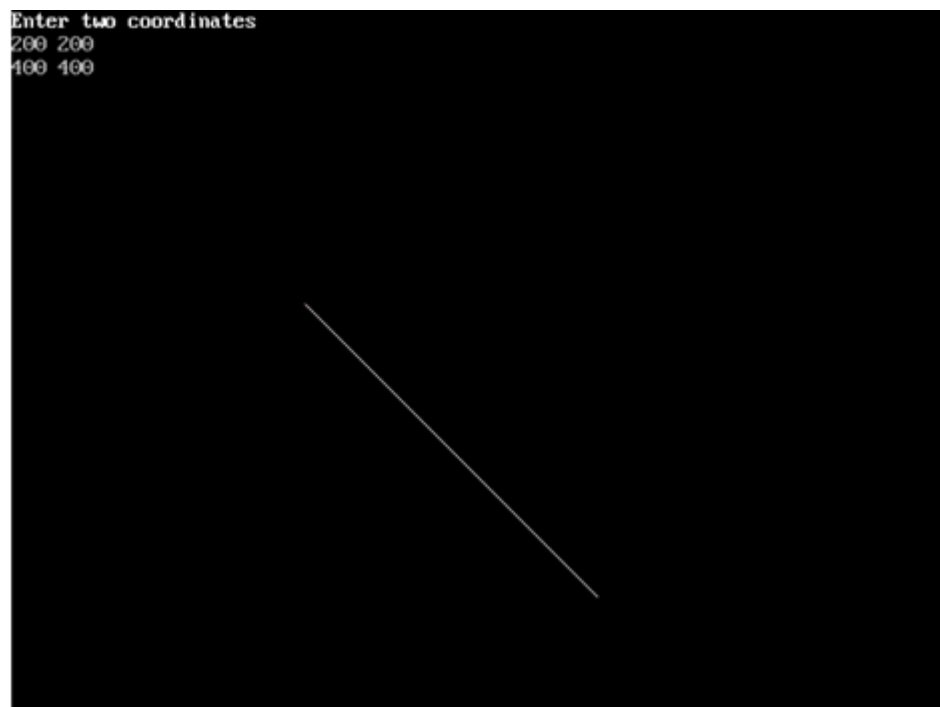
    if(abs(Dx) > abs(Dy)){

        steps = abs(Dx);

    }
```

```
Else
{
    steps = abs(Dy);
}
xin = Dx/(float)steps;
yin = Dy/(float)steps;
putpixel(round(X), round(Y), 6);
for(k = 0; k < steps; k++){
X = X + xin;
Y = Y + yin;
putpixel(round(X), round(Y), WHITE);
delay(50);
}
}
```

➤ **Result/Output:**



- **Discussion :** The DDA algorithm is a faster method for calculating pixel positions than the direct use of equation $y=mx+c$.

It eliminates the multiplication in line equation by making use of raster characteristics, so that appropriate increments are applied in the x or y direction to step to pixel positions along the line path.

- **Finding & Learning :**

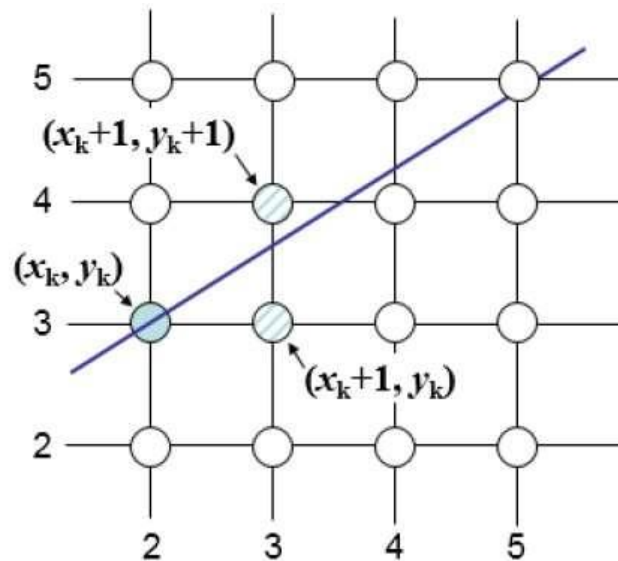
We found out that the accumulation of roundoff error in successive additions of the floating-point increment, however, can cause the calculated pixel positions to drift away from the true line path for long line segments. Furthermore, the rounding operations and floating-point arithmetic in procedure line DDA are still time-consuming. We can improve the performance of the DDA algorithm by separating the increments m and $1/m$ into integer and fractional parts so that all calculations are reduced to integer operations.

EXPERIMENT 2

➤ **Aim:** To generate a line using the Bresenham Line Algorithm.

➤ **Theory:**

A line-generating algorithm (Bresenham Line Algorithm) scan converts lines using only incremental integer calculations that can be adapted to display circles and other curves.



In the above fig. , the vertical axes show scan-line positions, and the horizontal axes identify pixel columns. So here, we need to decide which of two possible pixel positions is closer to the line path at each sample step.

Next pixel position is determined by testing the sign of an integer parameter, whose value is proportional to the difference between the separations of the two pixel positions from the actual line path. Our choices are the pixels at positions (x_k+1, y_k) and (x_k+1, y_k+1) .

➤ **Algorithm :**

1. Input the two line endpoints and store the left endpoint in (x_0, y_0) .
2. Load (x_0, y_0) into the frame buffer, that is, plot the first point.
3. Calculate constants Δx , Δy , $2\Delta y$ and $2\Delta y - 2\Delta x$, and obtain the starting value for the decision parameter as

$$p_0 = 2\Delta y - \Delta x$$

4. At each x_k , along the line, starting at $k=0$, perform the following test:

If $p_k < 0$, the next point to plot is (x_{k+1}, y_k) and

$$P_{k+1} = p_k + 2\Delta y$$

Otherwise, the next point to plot is (x_{k+1}, y_{k+1}) and

$$P_{k+1} = p_k + 2\Delta y - 2\Delta x$$

5. Repeat step 4 Δx times or till we reach the endpoint.

➤ **CODE :**

```
#include<stdio.h>
#include<stdlib.h>
#include<graphics.h>

void bresline(int x1, int y1, int x2, int y2)
{
    int twody = 2*abs(y2-y1);
    int dx = abs(x2-x1);
    int p = twody-dx;
    int x, y, xend, ystep=1;
    int twodydx= twody - 2*dx;

    if(x1>x2)
    {
        x=x2;
        y=y2;
        xend=x1;
        if(y2>y1) ystep=-1;
    }
    else
    {
        x=x1;
        y=y1;
        xend=x2;
        if(y1>y2) ystep=-1;
    }
    putpixel(x,y,15);

    while(x<xend)
```



```

    {
        x++;
        if(p<0)
        {
            p+=twody;
        }
        else
        {
            y+=ystep;
            p+=twodydx;
        }
        putpixel(x,y,15);
    }
}

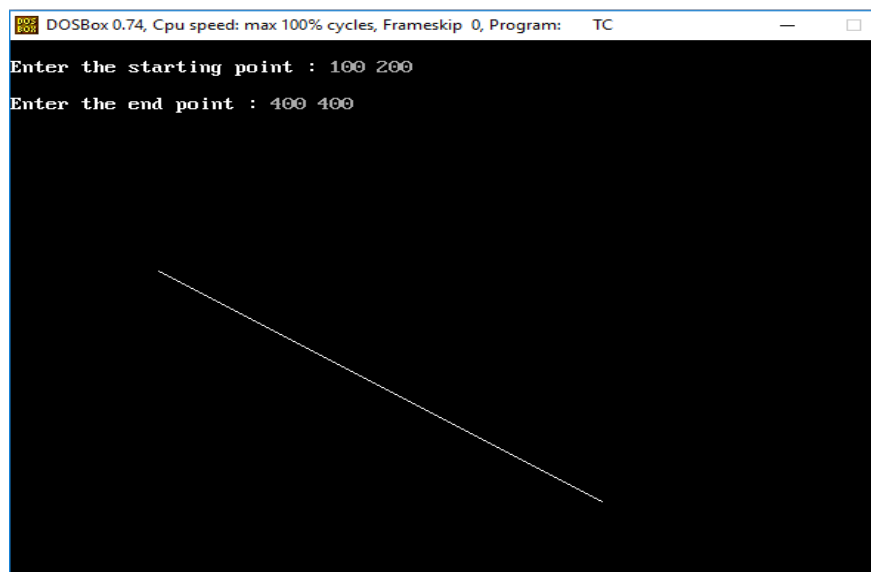
int main()
{
    int x1, x2, y1, y2;
    int gdriver = DETECT, gmode, errorcode;
    initgraph(&gdriver, &gmode, "..\\");
    errorcode = graphresult();

    if (errorcode != grOk)
    {
        printf("Graphics error: %s\n", grapherrormsg(errorcode));
        printf("Press any key to halt:");
        getch();
        exit(1);
    }
    printf("Enter start point\n");
    scanf("%d %d", &x1, &y1);
    printf("Enter end point\n");
    scanf("%d %d", &x2, &y2);
    bresline(x1, y1, x2, y2);

    getch();
    closegraph();
    return 0;
}

```

➤ **Output :**



Hence, we have the desired line as output on the screen.

➤ **Discussion :**

- Bresenham's line algorithm is a highly efficient incremental method over DDA.
- It produces mathematically accurate results using only integer addition, subtraction, and multiplication by 2, which can be accomplished by a simple arithmetic shift operation.

Learnings/Findings :

- Bresenham's line algorithm is more accurate method to produce a line, with given end points, than other line algorithms.
- Other curves circle or ellipse, also follow the same approach.

EXPERIMENT 4

- **AIM:** Write a program to implement midpoint ellipse drawing algorithm.
- **RELATED THEORY:** Mid-Point Ellipse Drawing Algorithm uses symmetry of ellipse to draw an ellipse in computer graphics. This algorithm is implemented for one quadrant only. We divide the quadrant into two regions and the boundary of two regions is the point at which the curve has a slope of -1. We process by taking unit steps in the x direction to the point P (where curve has a slope of -1), then taking unit steps in the y direction and applying midpoint algorithm at every step.

➤ **ALGORITHM:**

1. Input r_x , r_y , and ellipse center (x_c, y_c) , and obtain the first point on an ellipse centered on the origin as
$$(x_0, y_0) = (0, r_y)$$
2. Calculate the initial value of the decision parameter in region 1 as
$$p_{10} = r_y^2 - r_x^2 r_y + \frac{1}{4} r_x^2$$
3. At each x_k position in region 1, starting at $k = 0$ perform the following test: If $p_{1k} < 0$, the next point along the ellipse centered on $(0, 0)$ is $(x_k + 1, y_k)$ and
$$p_{1k+1} = p_{1k} + 2r_y^2 x_{k+1} + r_y^2$$
Otherwise, the next point along the circle is $(x_k + 1, y_k - 1)$ and
$$p_{1k+1} = p_{1k} + 2r_y^2 x_{k+1} - 2r_x^2 y_{k+1} + r_y^2$$
with
$$2r_y^2 x_{k+1} = 2r_y^2 x_k + 2r_y^2 \quad 2r_x^2 y_{k+1} = 2r_x^2 y_k - 2r_x^2$$
And continue until $2r_y^2 x \geq 2r_x^2 y$.
4. Calculate the initial value of the decision parameter in region 2 using the last point (x_0, y_0) calculated in region 1 as
$$p_{20} = r_y^2 (x_0 + \frac{1}{2})^2 + r_x^2 (y_0 - 1)^2 - r_x^2 r_y^2$$
5. At each y_k position in region 2, starting at $k = 0$, perform the following test: If $p_{2k} > 0$, the next point along the ellipse centered on $(0, 0)$ is $(x_k, y_k - 1)$ and
$$p_{2k+1} = p_{2k} - 2r_x^2 y_{k+1} + r_x^2$$
Otherwise, the next point along the circle is $(x_k + 1, y_k - 1)$ and
$$p_{2k+1} = p_{2k} + 2r_y^2 x_{k+1} - 2r_x^2 y_{k+1} + r_x^2$$
using the same incremental calculations for x and y as in region 1.
6. Determine symmetry points in the other three quadrants.
7. Move each calculated pixel position (x, y) onto the elliptical path centered on (x_c, y_c) and plot the coordinate values:
$$x = x + x_c, \quad y = y + y_c$$
8. Repeat the steps for region 1 until $2r_y^2 x \geq 2r_x^2 y$.

➤ **CODE:**

```
#include <stdio.h>
#include <conio.h>
#include <graphics.h>
#include <dos.h>

int main() {
    /* request auto detection */
    int gdriver = DETECT, gmode, err;
    long midx, midy, xradius, yradius;
    long xrad2, yrad2, twoxrad2, twoyrad2;
    long x, y, dp, dpx, dpy;

    /* initialize graphic mode */
    initgraph(&gdriver, &gmode, "C:/TURBOC3/BGI");
    err = graphresult();

    if (err != grOk) {
        /* error occurred */
        printf("Graphics Error: %s\n",
               grapherrormsg(err));
        return 0;
    }

    /* x axis radius and y axis radius of ellipse */
    xradius = 100, yradius = 50;

    /* finding the center postion to draw ellipse */
    midx = getmaxx() / 2;
    midy = getmaxy() / 2;

    xrad2 = xradius * xradius;
    yrad2 = yradius * yradius;

    twoxrad2 = 2 * xrad2;
    twoyrad2 = 2 * yrad2;
    x = dpx = 0;
    y = yradius;
    dpy = twoxrad2 * y;

    putpixel(midx + x, midy + y, WHITE);
    putpixel(midx - x, midy + y, WHITE);
    putpixel(midx + x, midy - y, WHITE);
    putpixel(midx - x, midy - y, WHITE);

    dp = (long) (0.5 + yrad2 - (xrad2 * yradius) + (0.25 * xrad2));

    while (dpx < dpy) {
        x = x + 1;
        dpx = dpx + twoyrad2;
        if (dp < 0) {
            dp = dp + yrad2 + dpx;
        } else {
            y = y - 1;
            dpy = dpy - twoxrad2;
        }
    }
}
```

```

        dp = dp + yrad2 + dpx - dpy;
    }

    /* plotting points in y-axis(top/bottom) */
    putpixel(midx + x, midy + y, WHITE);
    putpixel(midx - x, midy + y, WHITE);
    putpixel(midx + x, midy - y, WHITE);
    putpixel(midx - x, midy - y, WHITE);
    delay(100);
}

delay(500);

dp = (long)(0.5 + yrad2 * (x + 0.5) * (x + 0.5) +
           xrad2 * (y - 1) * (y - 1) - xrad2 * yrad2);

while (y > 0) {
    y = y - 1;
    dpy = dpy - twoxrad2;

    if (dp > 0) {
        dp = dp + xrad2 - dpy;
    } else {
        x = x + 1;
        dpx = dpx + twoyrad2;
        dp = dp + xrad2 - dpy + dpx;
    }

    /* plotting points at x-axis(left/right) */
    putpixel(midx + x, midy + y, WHITE);
    putpixel(midx - x, midy + y, WHITE);
    putpixel(midx + x, midy - y, WHITE);
    putpixel(midx - x, midy - y, WHITE);
    delay(100);
}

getch();

/* deallocate memory allocated for graphic screen */
closegraph();

return 0;
}

```

➤ **RESULT:**

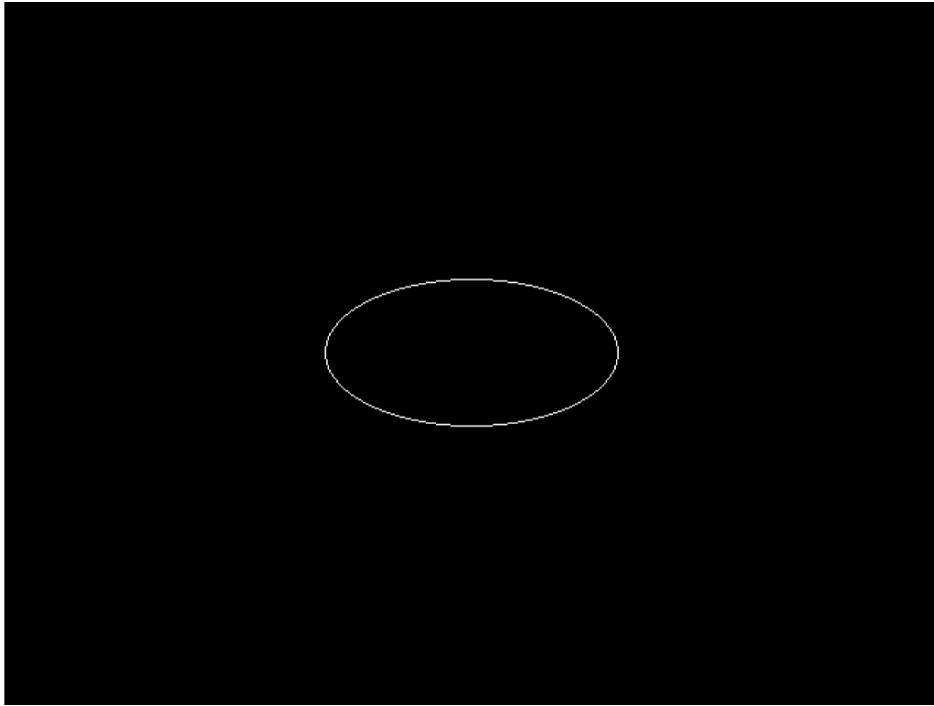


Image 1

Hence the ellipse can be drawn using this algorithm.

➤ **DISCUSSION:**

The output in image 1 shows the drawing of ellipse which is drawn by using mid-point algorithm for ellipse.

➤ **FINDING AND LEARNING:**

From the above output we observe that we can use symmetry of ellipse to draw an ellipse in computer graphics. Here, we calculated pixel points for one quadrant and used those points to draw the whole figure. That is, suppose the calculated pixel point is (x, y) then pixel point in other quadrants will be $(-x, y)$, $(x, -y)$, $(-x, -y)$.

EXPERIMENT-5

➤ **Aim:** To implement the Mid Point Circle Algorithm.

➤ **Related Theory:**

As in the raster line algorithm, we sample at unit intervals and determine the closest pixel position to the specified circle path at each step. For a given radius r and screen center position (x_c, y_c) , we can first set up our algorithm to calculate pixel positions around a circle path centered at the coordinate origin $(0, 0)$. Then each calculated position (x, y) is moved to its proper screen position by adding x to x_c and y to y_c .

Along the circle section from $x = 0$ to $x = y$ in the first quadrant, the slope of the curve varies from 0 to -1. Therefore, we can take unit steps in the positive x direction over this octant and use a decision parameter to determine which of the two possible y positions is closer to the circle path.

We use function $f(x,y)=x^2+y^2-r^2$

➤ **Algorithm:**

1. Input radius r and circle center (x_c, y_c) and obtain the first point on the circumference of a circle centered on the origin as

$$(x_0, y_0) = (0, r)$$

2. Calculate the initial value of the decision parameter as

$$p_0 = 5/4 - r$$

3. At each x_k position, starting at $k = 0$, perform the following test: If $p_k < 0$, the next point along the circle centered on $(0, 0)$ is $(x_k + 1, y_k)$ and

$$p_{k+1} = p_k + 2x_{k+1} + 1$$

Otherwise, the next point along the circle is $(x_k + 1, y_k - 1)$ and

where $2x_{k+1} = 2x_k + 2$ and $2y_{k+1} = 2y_k - 2$.

4. Determine symmetry points in the other seven octants.

5. Move each calculated pixel position (x, y) onto the circular path centered on (x_c, y_c) and plot the coordinate values:

$$y = y + y_c \text{ and } x = x + x_c$$

6. Repeat steps 3 through 5 until $x > y$.

➤ **Code:**

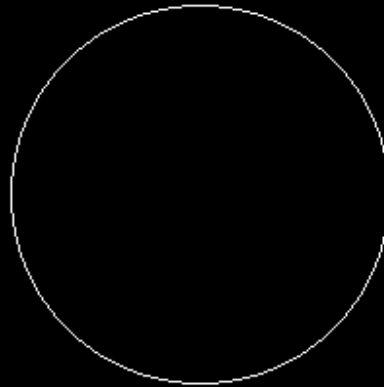
```
#include<stdio.h>
#include<stdlib.h>
#include<graphics.h>
void setpixel(int xc, int yc, int x, int y)
{  putpixel(xc+x,yc+y,15);
    putpixel(xc+x,yc-y,15);
    putpixel(xc-x,yc+y,15);
    putpixel(xc-x,yc-y,15);
    putpixel(xc+y,yc+x,15);
    putpixel(xc+y,yc-x,15);
    putpixel(xc-y,yc+x,15);
    putpixel(xc-y,yc-x,15);
}
void midptcircle(int xc, int yc, int r)
{  int p = 1 - r;
    int x= 0, y= r;
    setpixel(xc,yc,x,y);
    while(x<y)
    {  x++;
        if(p<0)
        {
            p+= 2*x +1;
        }
        else
        {  y--;
            p+= 2*(x-y) +1;
        }
        setpixel(xc,yc,x,y);
    }
}
int main()
```



```
{  
    int xc, yc, r;  
    int gdriver = DETECT, gmode, errorcode;  
    initgraph(&gdriver, &gmode, "..\\");  
    errorcode = graphresult();  
    if (errorcode != grOk)  
    {  
        printf("Graphics error: %s\\n", grapherrormsg(errorcode));  
        printf("Press any key to halt:");  
        getch();  
        exit(1);  
    }  
    printf("Enter center of circle\\n");  
    scanf("%d %d", &xc, &yc);  
    printf("Enter radius\\n");  
    scanf("%d", &r);  
    midptcircle(xc,yc,r);  
    getch();  
    closegraph();  
    return 0;  
}
```

➤ **Output:**

```
*** Mid-Point Subdivision algorithm of circle ***  
Enter the value of Xc  400  
Enter the value of Yc  140  
Enter the Radius of circle  97
```



➤ **Discussion:**

Here we can see that how we can more points while using the symmetry of circle.

➤ **Findings and Learning:**

We can use symmetry to get other points in the other octants. If the centre of the circle is not origin then we can use the concept of shifting and shift the centre to origin.

EXPERIMENT 6

- **Aim:** To write and Implement code to draw elliptical arc using parametric equation.
- **Description:**

To write and implement a program in C to plot points on an elliptical arc using parametric equation of the ellipse

The following equations define an ellipse trigonometrically:

$$x = h + a \cos \theta$$

$$y = k + b \sin \theta$$

where (x, y) = current coordinates

a = length of major axis

b = length of minor axis

θ = current angle

The value of θ is varied

The algorithm starts with some value of θ , and then loops adding an increment to θ each time until the value of θ exceeds the specified range.

➤ **Algorithm:**

1. Start
2. Initialize the graphic system.
3. Get the values of variables:
 - a. Central point of the elliptical arc (H, K).
 - b. Radius of the ellipse along X-axis and Y-axis (RX, RY).
 - c. Starting ANGLE of Arc (S, E).
4. Elliptical_Arc(H, K, RX, RY, S, E)

Elliptical_Arc(H, K, RX, RY, S, E)

IF (S <= E) THEN:

 ANGLE := S

 RANGE := E

ELSE:

 ANGLE := E

 RANGE := S

 X := (RX*cos(ANGLE))

 Y := (RY*sin(ANGLE))

 DO:

 putpixel((INT)(H+X+0.5), (INT)(K-Y+0.5))

```

    ANGLE := ANGLE+0.001
    X := (RY*cos(ANGLE))
    Y := (RY*sin(ANGLE))
    IF (RANGE > ANGLE) THEN:
        EXIT

```

➤ **Code:**

```

#include <stdlib.h>
#include <iostream.h>
#include <graphics.h>
#include <conio.h>
#include <math.h>

void Elliptical_arc( int, int, int, int, int, int );

int main( int argc, char *argv[] ) {
    int driver = DETECT;
    int mode;
    int h=0,k=0,rx=0,ry=0,s=0,e=0;
    do{
        cout<<"\nCentral point of the Elliptical Arc (h,k) :";
        cout<<"\n-----";
        cout<<"\nEnter the value of h = ";
        cin>>h;
        cout<<"Enter the value of k = ";
        cin>>k;
        cout<<"\nRadius of the Ellipse (rx,ry) :";
        cout<<"\n-----";
        cout<<"\nEnter the radius along x-axis rx = ";
        cin>>rx;
        cout<<"Enter the radius along y-axis ry = ";
        cin>>ry;
    }
}

```

```

        cout<<"\nStarting & Ending Angle of Arc (s,e) :";
        cout<<"\n-----";
        cout<<"\nEnter the value of s = ";
        cin>>s;
        cout<<"Enter the value of e = ";
        cin>>e;
        initgraph(&driver,&mode,"C:/TURBOC3/BGI");
        setcolor(15);
        Elliptical_arc(h,k,rx,ry,s,e);
        setcolor(15);
        outtextxy(110,460,"Press <Enter> to continue or any other key to exit. ");
        int key = int(getch());
        if(key!=13)
            break;
        else closegraph();
    } while(1);
    return 0;
}

```

```

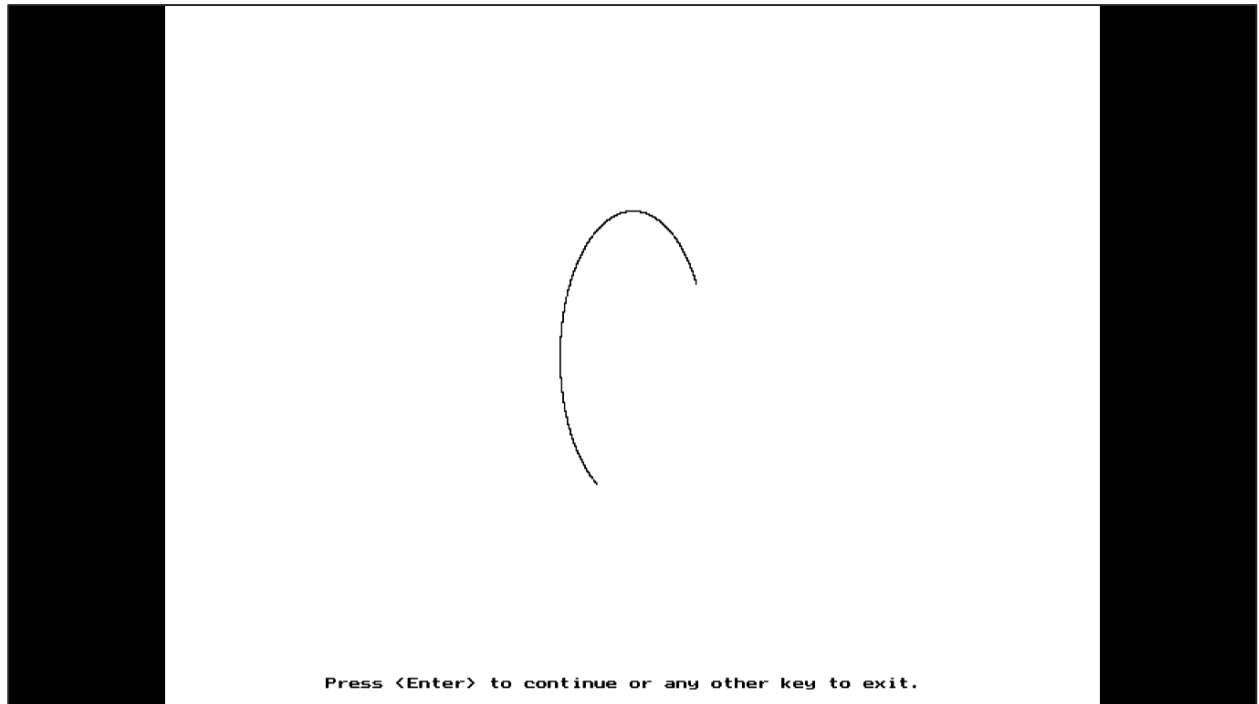
void Elliptical_arc(int h,int k,int rx, int ry, int s,int e) {
    int color = getcolor();
    float angle = (((s<=e)?s:e)*(M_PI/180));
    float range = (((e>s)?e:s)*(M_PI/180));

    float x = (rx*cos(angle));
    float y = (ry*sin(angle));

    do {
        putpixel((int)(h+x+0.5),(int)(k-y+0.5),color);
        angle+=0.001;
        x = (rx*cos(angle));
        y = (ry*sin(angle));
    } while(angle<=range);
}

```

➤ **Output:**



➤ **Discussion:**

This algorithm is based on the parametric form of the ellipse equation.

$$x = h + a \cos \theta$$

$$y = k + b \sin \theta$$

where (x, y) = current coordinates

a = length of major axis

b = length of minor axis

θ = current angle

What these equations do is generate the (x, y) coordinates of a point on the ellipse given an angle θ . The algorithm starts with some value of θ , and then loops adding an increment to θ each time until the value of θ exceeds the specified range. It draws straight line segments between these successive points on the ellipse. The ellipse is thus drawn as a series of straight lines. If the increment is small enough, the result looks like an ellipse to the eye, even though in strict mathematical terms it is not!

➤ **Learning and Findings:**

- a) We saw that using parametric equation of ellipse, it is possible to draw some points of an ellipse on a digital computer graphics system.
- b) The decision about how big to make the step size is a tradeoff. If it is very small, many lines will be drawn for a smooth circle, but there will be more computer time used to do it. If it is too large the circle will not be smooth and be visually ugly.
- c) The ellipse shows four-way symmetry. Hence, we can use the symmetry to decrease the number of steps. It will considerably increase efficiency of the algorithm.

EXPERIMENT- 7

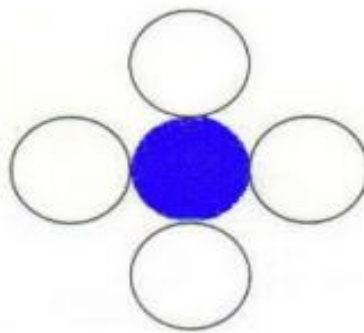
➤ **AIM : Write a program to implement Flood Fill algorithm.**

➤ **THEORY:**

Sometimes we come across an object where we want to fill the area and its boundary with different colors. We can paint such objects with a specified interior color instead of searching for particular boundary color as in boundary filling algorithm.

Instead of relying on the boundary of the object, it relies on the fill color. In other words, it replaces the interior color of the object with the fill color. When no more pixels of the original interior color exist, the algorithm is completed .

In Flood Fill algorithm we start with some seed and examine the neighboring pixels, however pixels are checked for a specified interior color instead of boundary color and is replaced by a new color. It can be done using 4 connected or 8 connected region method.



Below we use 4 connected region recursive algorithm to implement this algorithm.

➤ **ALGORITHM:**

Step 1 – Initialize the value of seed point (seedx, seedy), fcolor and dcol.

Step 2 – Define the boundary values of the polygon.

Step 3 – Check if the current seed point is of default color, then repeat the steps 4 and 5 till the boundary pixels reached.

```
If getpixel(x, y) = dcol then repeat step 4 and 5
```

Step 4 – Change the default color with the fill color at the seed point.

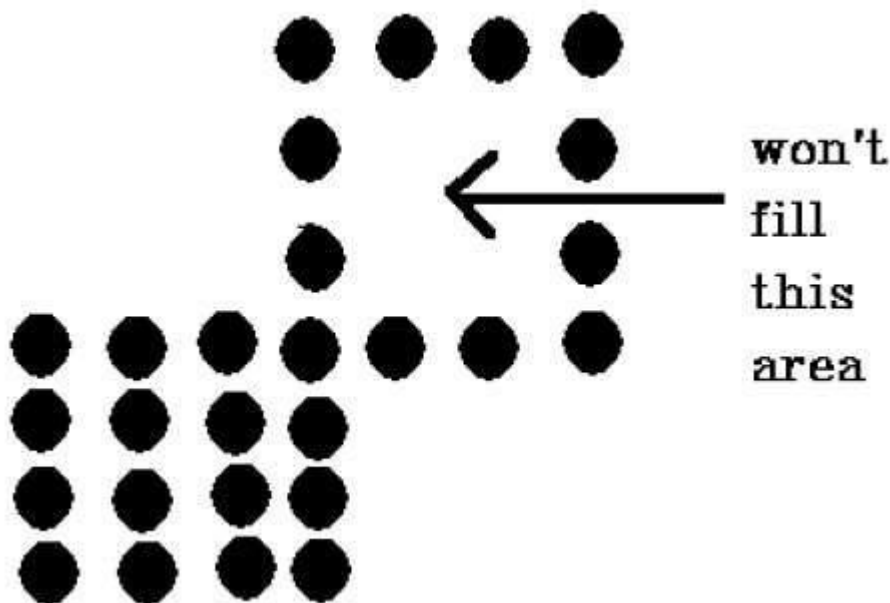
```
setPixel(seedx, seedy, fcol)
```

Step 5 – Recursively follow the procedure with four neighborhood points.

```
FloodFill (seedx - 1, seedy, fcol, dcol)
FloodFill (seedx + 1, seedy, fcol, dcol)
FloodFill (seedx, seedy - 1, fcol, dcol)
FloodFill (seedx, seedy + 1, fcol, dcol)
```

Step 6 – Exit

There is a problem with this technique. Consider the case as shown below where we tried to fill the entire region. Here, the image is filled only partially. In such cases, 4-connected pixels technique cannot be used.



➤ **CODE:**

```
#include<stdio.h>
#include<graphics.h>
#include<dos.h>

void floodFill(int x,int y,int oldcolor,int newcolor)
{
    if(getpixel(x,y) == oldcolor)
    {
        putpixel(x,y,newcolor);
        floodFill(x+1,y,oldcolor,newcolor);
        floodFill(x,y+1,oldcolor,newcolor);
        floodFill(x-1,y,oldcolor,newcolor);
        floodFill(x,y-1,oldcolor,newcolor);
    }
}
//getpixel(x,y) gives the color of specified pixel

int main()
{
    int gm,gd=DETECT,radius;
    int x,y;

    printf("Enter x and y positions for circle\n");
    scanf("%d%d",&x,&y);
    printf("Enter radius of circle\n");
    scanf("%d",&radius);

    initgraph(&gd,&gm,"c:\\turbo3\\bgi");
    circle(x,y,radius);
    floodFill(x,y,0,15);
    delay(5000);
    closegraph();

    return 0;
}
```

➤ **OUTPUT:**

```
Enter x and y positions for circle  
100 100  
Enter radius of circle  
30_
```



➤ **RESULT AND DISCUSSION:**

Flood fill algorithm has filled the desired circle completely. The flood fill algorithm takes three parameters: a start node, a target colour, and a replacement colour. The algorithm looks for all nodes which are connected to the start node by a path of the target colour, and changes them to the replacement colour.

➤ **FINDING AND LEARNING:**

We can modify findFill() method to reduce storage requirements of the stack by filling horizontal pixel spans ,i.e., we stack only beginning positions for those pixel spans having oldcolour .In this modified version, starting at the first position of each span, the pixel values are replaced until a value other than old colour is encountered. We can also show an area bordered by several different colour regions too.

EXPERIMENT 8

- **AIM :** WAP to implement Boundary Fill Algorithm
- **Description :** The boundary fill algorithm works as its name. This algorithm picks a point inside an object and starts to fill until it hits the boundary of the object. The color of the boundary and the color that we fill should be different for this algorithm to work.

In this algorithm, we assume that color of the boundary is same for the entire object. The boundary fill algorithm can be implemented by 4-connected pixels or 8-connected pixels.

➤ **Algorithm :**

4 Connected pixel :

- **Step 1** – Initialize the value of seed point (seedx, seedy), fcolor and dcol.
- **Step 2** – Define the boundary values of the polygon.
- **Step 3** – Check if the current seed point is of default color, then repeat the steps 4 and 5 till the boundary pixels reached.

If `getpixel(x , y) = dcol` then repeat step 4 and 5 .

- **Step 4** – Change the default color with the fill color at the seed point.

`setPixel(seedx , seedy , fcol)`

- **Step 5** – Recursively follow the procedure with four neighborhood points.

`FloodFill (seedx - 1 , seedy , fcol , dcol)`

`FloodFill (seedx + 1 , seedy , fcol , dcol)`

`FloodFill (seedx , seedy - 1 , fcol , dcol)`

`FloodFill (seedx -1 , seedy + 1 , fcol , dcol)`

- **Step 6** – Exit

➤ **Code :**

```
#include <conio.h>
#include <stdio.h>
#include <graphics.h>
#include <dos.h>

void fill_right(int x,int y);
void fill_left(int x,int y);

int main()
{
int gd = DETECT,gm,n,i,x,y;
initgraph(&gd,&gm,"C:/TURBOC3/BGI");
printf("**** Boundary Fill Algorithm ****\n");
line(50,50,200,50);
line(200,50,200,300);
line(200,300,50,300);
line(50,300,50,50);
x = 100;
y = 100;
fill_right(x,y);
fill_left(x-1,y);
getch();
return 0;
}

void fill_right(int x,int y)
{
if((getpixel(x,y)!=WHITE) && (getpixel(x,y)!=RED))
{
putpixel(x,y,RED);
fill_right(++x,y);
x=x-1;
fill_right(x,y-1);
fill_right(x,y+1);
}
delay(1);
}

void fill_left(int x,int y)
{
if((getpixel(x,y)!=WHITE) && (getpixel(x,y)!=RED))
{
putpixel(x,y,RED);
fill_left(--x,y);
x=x+1;
fill_left(x,y-1);
fill_left(x,y+1);
}
delay(1);}
```

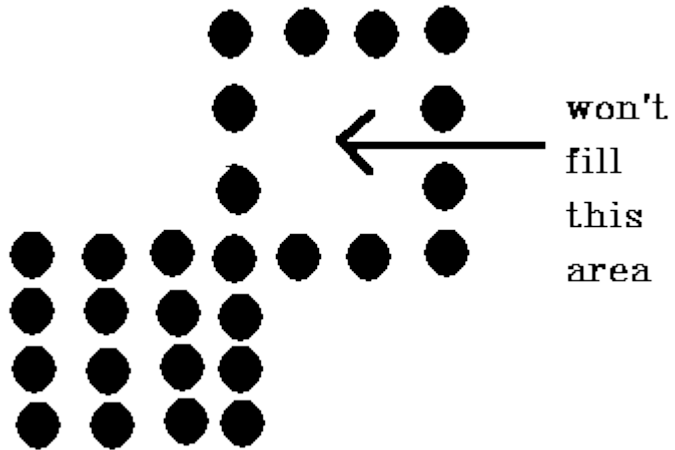
➤ **Output :**



➤ **Discussions :**

The algorithm works because we pick different colors for boundary and fill are different. The color is filled in the object until it hits the boundary. It is recursive in nature as the function returns when the pixel to be filled is the boundary color or already the fill color.

There is a problem with this technique.



Here, the image is filled only partially. In such cases, 4-connected pixels technique cannot be used.

We can use 8 connected pixel to resolve this problem.

EXPERIMENT 10

➤ **AIM:** Write a program to implement Liang-Barsky Line Clipping Algorithm.

➤ **DESCRIPTION AND RELATED THEORY:**

The Liang-Barsky algorithm is a line clipping algorithm. The idea of the Liang-Barsky clipping algorithm is to do as much testing as possible before computing line intersections.

Liang and Barsky have created an algorithm that uses floating-point arithmetic but finds the appropriate end points with at most four computations. This algorithm uses the parametric equations for a line and solves four inequalities to find the range of the parameter for which the line is in the viewport.

Consider the parametric definition of a line:

$$x = x_1 + u\Delta x$$

$$y = y_1 + u\Delta y$$

$$\Delta x = (x_2 - x_1)$$

$$\Delta y = (y_2 - y_1) \quad 0 \leq (u, v) \leq 1$$

A line is inside the clipping region for values of u such that:

$$x_{\min} \leq x_0 + u\Delta x \leq x_{\max} \quad \Delta x = x_{\text{end}} - x_0$$

$$y_{\min} \leq y_0 + u\Delta y \leq y_{\max} \quad \Delta y = y_{\text{end}} - y_0$$

This can be described as $u p_k \leq q_k$, $k = 1, 2, 3, 4$

Rearranging, we get

$$-u\Delta x \leq (x_1 - x_{\min}) \quad (\text{LEFT})$$

$$u\Delta x \leq (x_{\max} - x_1) \quad (\text{RIGHT})$$

$$-v\Delta y \leq (y_1 - y_{\min}) \quad (\text{BOTTOM})$$

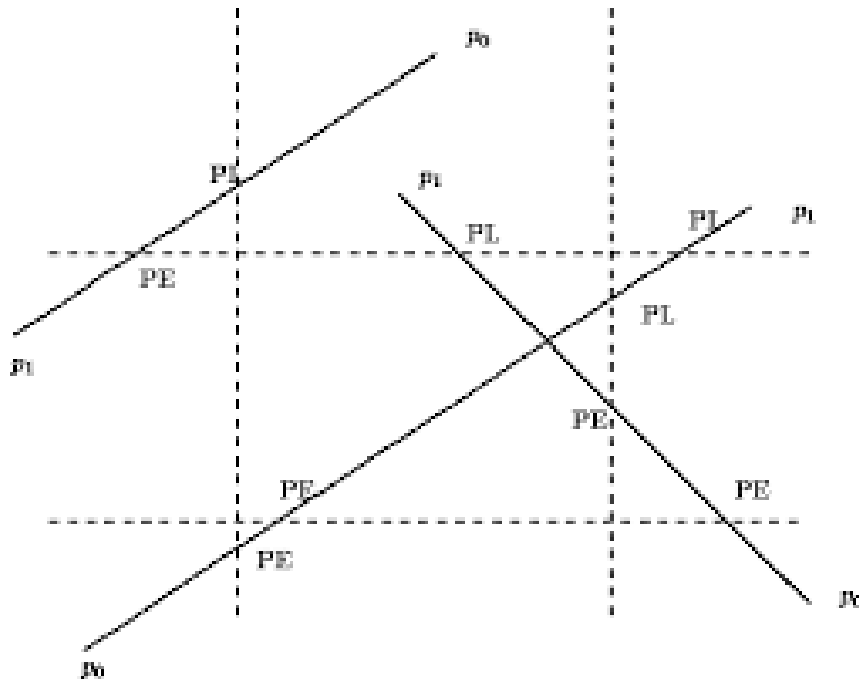
$$v\Delta y \leq (y_{\max} - y_1) \quad (\text{TOP})$$

When $p_k < 0$, as u increases - line goes from outside to inside - entering

When $p_k > 0$, - line goes from inside to outside - exiting

When $p_k = 0$, - line is parallel to an edge

If there is a segment of the line inside the clip region, a sequence of infinite line intersections must go: entering, entering, exiting, exiting.



➤ **ALGORITHM :**

- 1) Read two end points of line $P_1 (x_1, y_1)$ and $P_2 (x_2, y_2)$
- 2) Read two corner vertices, left top and right bottom of window: (X_{wMIN}, Y_{wMAX}) and (X_{wMAX}, Y_{wMIN})
- 3) Calculate values of parameters p_i and q_i for $i = 1, 2, 3, 4$ such that

$$p_1 = -\Delta x$$

$$q_1 = x_1 - X_{MIN}$$

$$p_2 = \Delta x$$

$$q_2 = X_{MAX} - x_i$$

$$p_3 = -\Delta y$$

$$q3 = y1 - Y_{\text{MIN}}$$

$$q2 = \Delta y$$

$$q4 = Y_{\text{MAX}} - y1$$

4) If $p_i = 0$ then,

{Line is parallel to i th boundary.

If $q_i < 0$ then,

{

Line is completely outside the boundary. Therefore, discard line segment and Go to Step 10.}

else

Check line is horizontal or vertical and accordingly check line end points with corresponding boundaries. If line endpoints lie within the bounded area then use them to draw line. Otherwise use boundary coordinates to draw line. Go to Step 10.

}

}

5) Initialize $t1 = 0$ and $t2 = t1$

6) Calculate values for q_i/p_i for $i = 1, 2, 3, 4$

7) Select values of q_i/p_i where $p_i < 0$ and assign maximum out of them as $t1$.

8) Select values of q_i/p_i where $p_i > 0$ and assign maximum out of them as $t2$.

9) If ($t1 < t2$)

{

Calculate endpoints of clipped line:

$$xx1 = x1 + t1 \Delta x$$

$xx2 = x1 + t2 \Delta x$

$yy1 = y1 + t1 \Delta y$

$yy2 = y1 + t2 \Delta y$

Draw line (xx1, yy1, xx2, yy2)

}

10) Stop

➤ **CODE:**

```
#include<iostream.h>
```

```
#include<conio.h>
```

```
#include<graphics.h>
```

```
void main()
```

```
{
```

```
    int gd = DETECT, gm;
```

```
    initgraph(&gd, &gm, "C:\\\\TC\\\\BGI");
```

```
    int x1, y1, x2, y2, xmax, xmin, ymax, ymin, xx1, yy1, xx2, yy2, dx, dy, i;
```

```
    int p[4], q[4];
```

```
    float t1, t2, t[4];
```

```
    cout<<"Enter the lower co-ordinates of window";
```

```
    cin>>xmin>>ymin;
```

```
    cout<<"Enter the upper co-ordinates of window";
```

```
    cin>>xmax>>ymax;
```

```
    setcolor(RED);
```

```
    rectangle(xmin, ymin, xmax, ymax);
```

```
    cout<<"Enter x1:";
```

```
    cin>>x1;
```

```
    cout<<"Enter y1:";
```

```
    cin>>y1;
```

```
    cout<<"Enter x2:";
```

```
    cin>>x2;
```

```
    cout<<"Enter y2:";
```

```
    cin>>y2;
```

```

line(x1,y1,x2,y2);
dx=x2-x1;
dy=y2-y1;
p[0]=-dx;
p[1]=dx;
p[2]=-dy;
p[3]=dy;
q[0]=x1-xmin;
q[1]=xmax-x1;
q[2]=y1-ymin;
q[3]=ymax-y1;

for(i=0;i < 4;i++)
{

    if(p[i]!=0)
    {

        t[i]=(float)q[i]/p[i];

    }
    else
        if(p[i]==0 && q[i] < 0)
            cout<<"Line completely outside the window";
        else
            if(p[i]==0 && q[i] >= 0)
                cout<<"Line completely inside the window";

}

if (t[0] > t[2]){
    t1=t[0];
}
Else{
    t1=t[2];
}
if (t[1] < t[3]){
    t2=t[1];
}

```

```

else{
            t2=t[3];
        }
        if (t1 < t2){
            xx1=x1+t1*dx;
            xx2=x1+t2*dx;
            yy1=y1+t1*dy;
            yy2=y1+t2*dy;
            cout<<"Line after clipping:";
            setcolor(WHITE);
            line(xx1,yy1,xx2,yy2);
        }
        Else
    {
            cout<<"Line lies out of the window" }

        getch();

    }

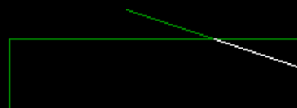
```

➤ **OUTPUT:**

```

Enter the lower co-ordinates of window200 300
Enter the upper co-ordinates of window400 350
Enter x1:280
Enter y1:280
Enter x2:400
Enter y2:320
line after clipping:

```



➤ **DISCUSSION:**

In summary : (x,y) coordinates are only computed for the two final intersection points. At most 4 parameter values are computed. This is a non-iterative algorithm and It Can be extended to 3-D.

➤ **FINDINGS AND LEARNINGS:**

Pros:

- 1) Less intersection calculations.
- 2) More efficient.
- 3) Only requires one division to update t1 and t2.
- 4) Window intersections of line are calculated just once.

Cons:

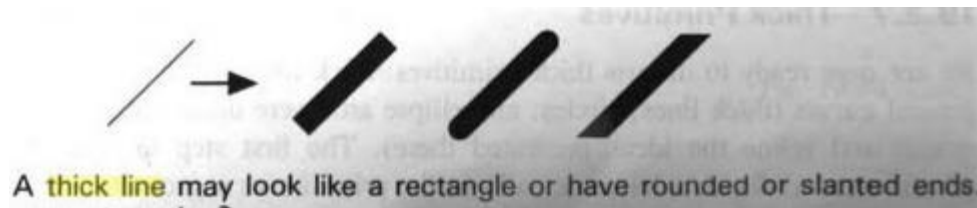
- 1)Cohen-Sutherland is much easier to understand.

EXPERIMENT 11

- **AIM:** Write a program to implement **Thick Lines**.

➤ **DESCRIPTION OF AIM AND RELATED THEORY:**

It is based on an extension to Bresenham's Line drawing algorithm given by Alan Murphy.



➤ **Advantages of the Algorithm:**

- Each pixel, which is set, is visited only ONCE - this makes the algorithm very suitable for use in XOR situations (for example, when pixels are XORed onto other images).
- The inner loop and most of the outer loop require only addition, and comparison testing. This produces an efficient algorithm.
- The line thickness can be a real number, is not necessary to be an integer.
- The perceived line thickness does not vary for lines of different orientations.
- It is possible to have a thickness which is a function of the distance from the start point - the thickness varies along the line, giving the possibility of depth-cueing (Z-axis effects).

➤ **ALGORITHM:**

In order to draw a thick line, you need to draw a number of parallel single thickness lines. There are 2 ways to do this:

1. Draw lines perpendicular to the ideal line stepping along it.
2. Draw lines parallel to the ideal line and step perpendicularly.

The document describes algorithms for each of the above situations. Both algorithms have two Bresenham loops, one inside the other. The outer loop does the 'stepping' and the inner loop draws single pixel lines.

The key aspect of the technique is that the outer loop contains just the right information for the inner loop. This is important as the inner loop must be started in the 'correct' phase of its cycle so that the diagonal moves occur in phase. The outer loop's difference terms are passed to the inner loop to accomplish this.

Another aspect of the technique is that an 'extra' inner loop cycle is required whenever the outer loop and the inner loop do a diagonal move together. In this case the outer

loop does a 'double square' move rather than a diagonal move. When the outer loop detects this condition it performs a square move, calls the inner loop, does another square move at right angles and calls the inner loop again. In this way a diagonal move is achieved but there are two calls to the inner loop.

➤ **CODE:**

```
#include<graphics.h>
#include<conio.h>
#include<math.h>
voidshow_screen();
voidThick_line(constint,constint,constint,constint,constint = 0);
int main()
{
    int driver=DETECT;
    int mode;
    initgraph(&driver,&mode,"C:/TURBOC3/BGI");
    show_screen();
    setcolor(15);
    Thick_line(150,100,540,100,0);
    Thick_line(150,175,540,175,1);
    Thick_line(150,250,540,250,2);
    Thick_line(150,325,540,325,3);
    Thick_line(150,400,540,400,4);
    char Style[5][15]={"Style #1","Style #2","Style #3","Style #4","Style #5"};
    for(int count=0;count<5;count++)
        outtextxy(60,(97+(75*count)),Style[count]);
    getch();
    return 0;
}
voidThick_line(constint x_1,const int y_1,const int x_2,const int y_2,const intline_type)
{
    int color=getcolor();
    int x1=x_1;
    int y1=y_1;
    int x2=x_2;
    int y2=y_2;
    if(x_1>x_2)
    {
        x1=x_2;
        y1=y_2;
        x2=x_1;
        y2=y_1;
    }
    int dx=abs(x2-x1);
    intdy=abs(y2-y1);
    intinc_dec=((y2>=y1)?1:-1);
    if(dx>dy)
    {
        inttwo_dy=(2*dy);
        inttwo_dy_dx=(2*(dy-dx));
        int p=((2*dy)-dx);
        int x=x1;
        int y=y1;
```

```

while(x<=x2)
{
    if(line_type==0)
        putpixel(x,y,color);
    else if(line_type==1)
    {
        putpixel(x,y,color);
        putpixel(x,(y+1),color);
    }
    else if(line_type==2)
    {
        putpixel(x,y,color);
        putpixel(x,(y+1),color);
        putpixel(x,(y+2),color);
    }
    else if(line_type==3)
    {
        putpixel(x,y,color);
        putpixel(x,(y+1),color);
        putpixel(x,(y+2),color);
        putpixel(x,(y+3),color);
    }
    else if(line_type==4)
    {
        putpixel(x,y,color);
        putpixel(x,(y+1),color);
        putpixel(x,(y+2),color);
        putpixel(x,(y+3),color);
        putpixel(x,(y+4),color);
    }
    x++;
    if(p<0)
        p+=two_dy;
    else
    {
        y+=inc_dec;
        p+=two_dy_dx;
    }
}
}
else
{
    inttwo_dx=(2*dx);
    inttwo_dx_dy=(2*(dx-dy));
    int p=((2*dx)-dy);
    int x=x1;
    int y=y1;
    while(y!=y2)
    {
        if(line_type==0)
            putpixel(x,y,color);
        else if(line_type==1)
        {
            putpixel(x,y,color);
            putpixel(x,(y+1),color);
        }
    }
}

```

```

        else if(line_type==2)
        {
            putpixel(x,y,color);
            putpixel(x,(y+1),color);
            putpixel(x,(y+2),color);
        }
        else if(line_type==3)
        {
            putpixel(x,y,color);
            putpixel(x,(y+1),color);
            putpixel(x,(y+2),color);
            putpixel(x,(y+3),color);
        }
        else if(line_type==4)
        {
            putpixel(x,y,color);
            putpixel(x,(y+1),color);
            putpixel(x,(y+2),color);
            putpixel(x,(y+3),color);
            putpixel(x,(y+4),color);
        }
        y+=inc_dec;
        if(p<0)
            p+=two_dx;
        else
        {
            x++;
            p+=two_dx_dy;
        }
    }
}

void show_screen()
{
    setfillstyle(1,1);
    bar(256,27,365,37);
    settextstyle(0,0,1);
    setcolor(15);

    outtextxy(5,5,"*****");
    outtextxy(5,17,"*-
*****");
    outtextxy(5,29,"*-----");
    outtextxy(5,41,"*-
*****");
    outtextxy(5,53,"*-
*****");
    setcolor(11);
    for(int count=0;count<=30;count++)
        outtextxy(5,(65+(count*12)),"*-**-");
}

```

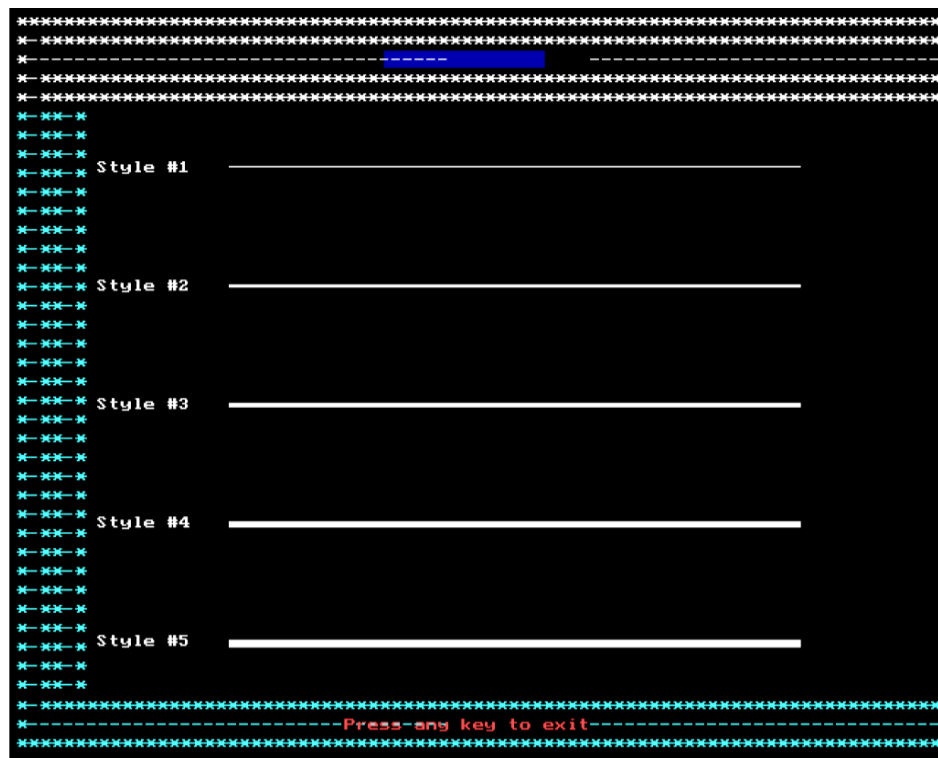
```

outtextxy(5,438,"*~
*****
*****_*");
outtextxy(5,450,"*-----
-----*");

outtextxy(5,462,"*****
*****");
setcolor(12);
outtextxy(229,450,"Press any key to exit");
}

```

➤ OUTPUT:



```

*****
*****
*****
Style #1  -----
*****
Style #2  -----
*****
Style #3  -----
*****
Style #4  -----
*****
Style #5  -----
*****
*****
*****
Press any key to exit
*****

```

➤ DISCUSSION:

The most powerful aspect of this algorithm is the fact that it can vary the thickness along the base line. This allows drawing very decorative lines. Consider the case where the width function is a sine wave etc. Another possibility is to use this feature to draw lines with perspective. Also, the left and right halves of the line are drawn independently of each other. This allows yet more opportunities to draw decorations.

With a minor changes to the algorithm, the algorithm can be made to traverse each pixel only once. The only pixels drawn twice are the ones on the base line. This could be eliminated easily. This would allow the algorithm to be used for situations where the objective is not to directly paint but execute some operation on the pixels, such as negating the background, doing an intersection etc.

Since we have a measure of the current thickness in the perpendicular function, we could draw anti-aliased lines using this algorithm. We could vary the brightness as we move away from the center of the thick line.

➤ **FINDING AND LEARNING:**

The major learning from this algorithm is to make use of simple line drawing algorithm and by using diagonal move technique or double-square technique, extend that into making thicker lines and thus even generate filled polygons. We also learned that using the above stated algorithm we can not only generate thick lines using the perpendicular drawing approach, but also we can vary the thickness of the line on the base and the ends and hence generating various shapes.

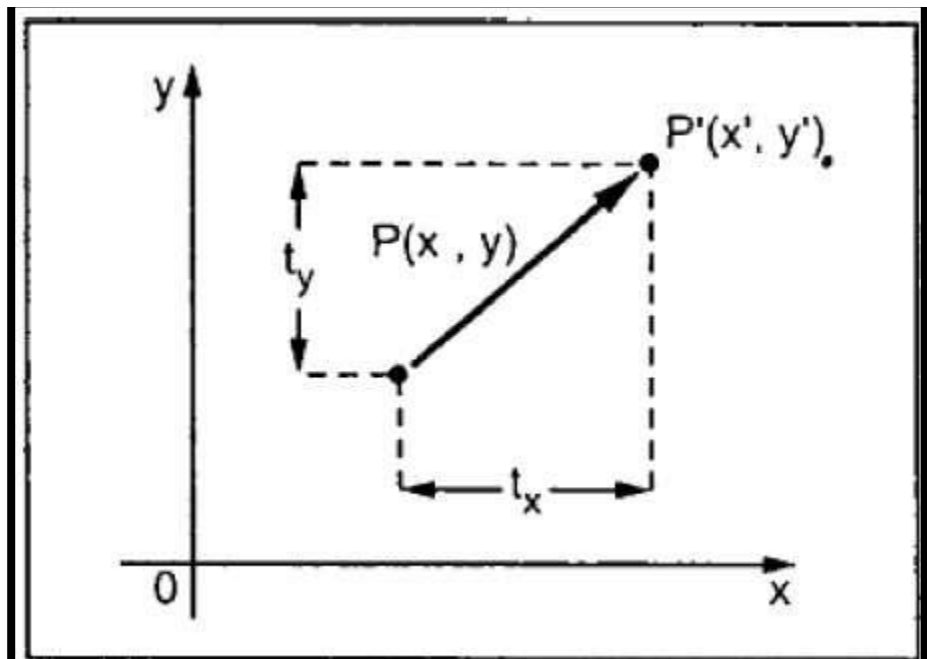
EXPERIMENT 12

➤ **AIM:**

To implement 2D translation of a Triangle

➤ **THEORY AND DESCRIPTION :**

A translation moves an object to a different position on the screen. A point in 2D can be translated by adding translation coordinate (t_x , t_y) to the original coordinate (X , Y) to get the new coordinate (X' , Y').



From the above figure,

$$X' = X + t_x$$

$$Y' = Y + t_y$$

The pair (t_x , t_y) is called the translation vector or shift vector. The above equations can also be represented using a translation matrix.

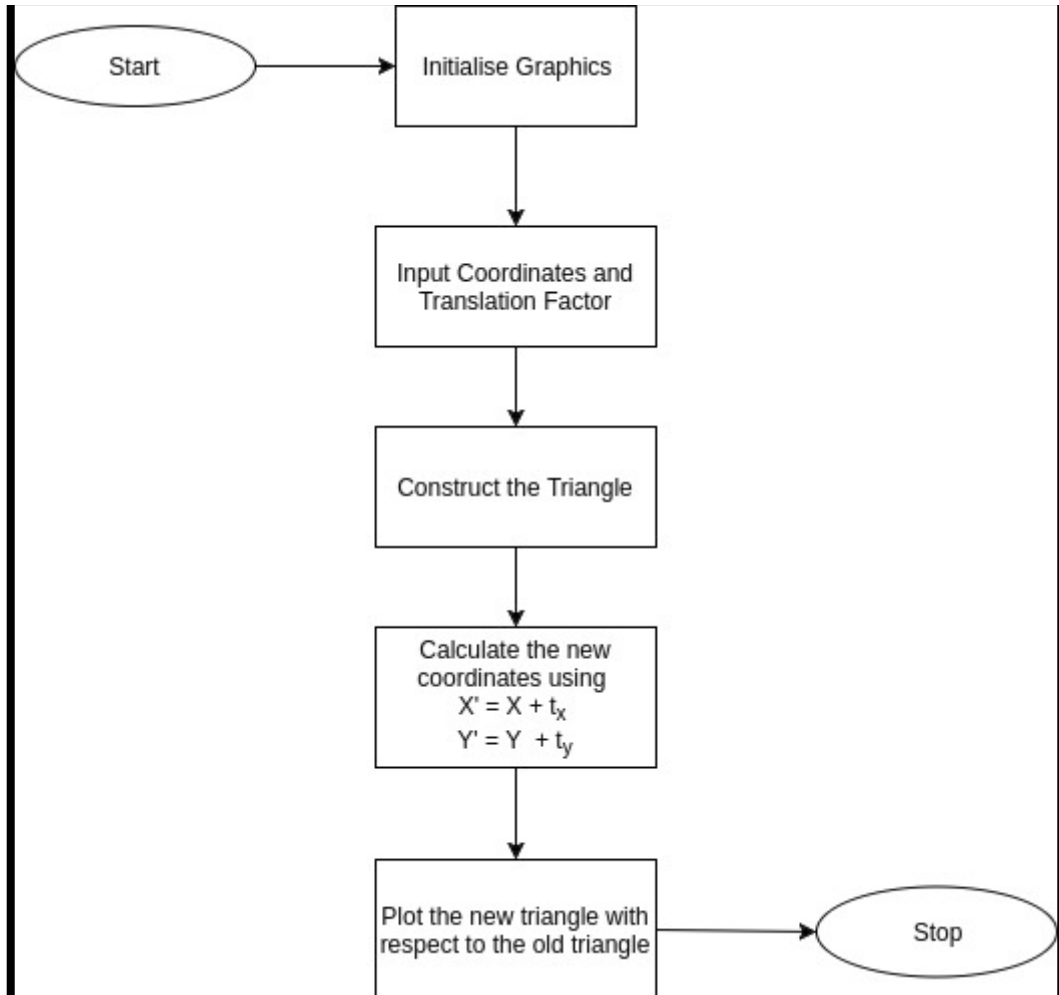
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

In this experiment, the 3 coordinates of a triangle are given along with the translation factor. We need to find the new coordinates and plot the translated triangle with respect to the original triangle.

➤ **ALGORITHM AND FLOWCHART:**

1. Start
2. Initialize the graphics mode.
3. Construct the Triangle using the three coordinates to draw three lines.
4. Translation

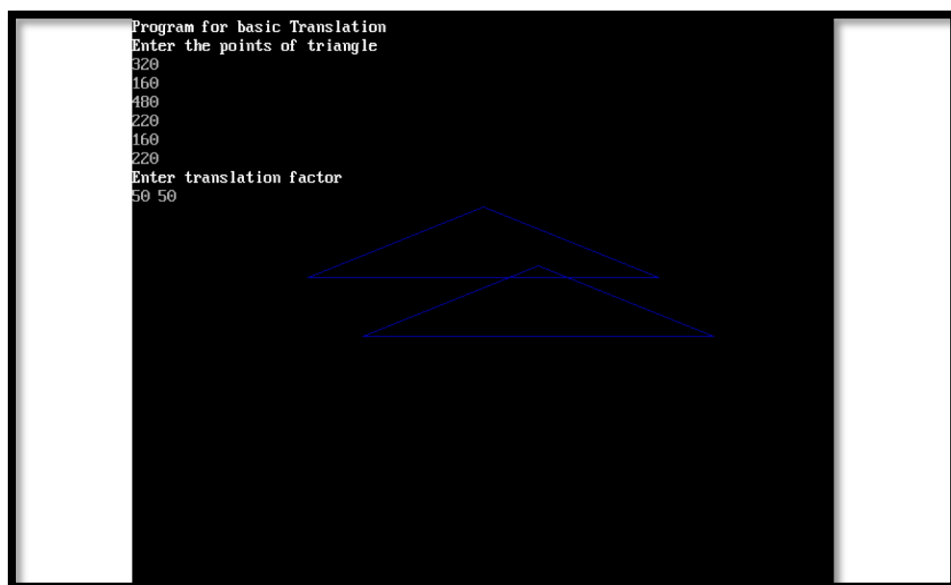
- a) Get the translation value t_x , t_y
- b) Move the 2d object with t_x , t_y ($x'=x+t_x, y'=y+t_y$ for all the three points).
- c) Plot the new three points, i.e. (x', y')



➤ **CODE:**

```
#include<graphics.h>
#include<stdlib.h>
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
    int gm , gd=DETECT;
    initgraph(&gd, &gm , "C:\\\\TurboC3\\\\BGI");
    int x1 , x2 , x3 , y1 , y2 , y3;
    printf("Program for basic Translation\\nEnter the points of triangle\\n");
    setcolor(1);
    scanf("%d%d%d%d%d%d",&x1 , &y1 , &x2 , &y2 , &x3 ,&y3);
    printf("Enter translation factor\\n");
    int tx , ty;
    scanf("%d %d" , &tx , &ty);
    line(x1 , y1 , x2, y2);
    line(x2 , y2 , x3 , y3);
    line(x1 , y1 , x3, y3);
    int nx1 , nx2, nx3 , ny1 , ny2 , ny3 ;
    nx1= x1+tx;
    nx2= x2+tx;
    nx3= x3+tx;
    ny1= y1+ty;
    ny2= y2+ty;
    ny3= y3+ty;
    line(nx1 , ny1 , nx2 , ny2);
    line(nx2 , ny2 , nx3 , ny3);
    line (nx3 , ny3 , nx1 , ny1);
    getch();
    closegraph();
}
```

➤ **OUTPUT:**



➤ **DISCUSSIONS:**

From the above results it becomes evident that when 2 dimensional objects represented in the form of a 2-D matrix can be transformed using t_x and t_y as the translation parameters. Here t_x and t_y are defined as the difference between the x coordinates and y coordinates of the original point and the point where the translation has to be done. The value of t_x and t_y is positive is the new point where the object has to be translated is in 1st coordinate and the value of point ($x' > x$ and $y' > y$). Similarly to translate back the point to the original position t_x can be substituted with $-t_x$ and t_y with $-t_y$.

➤ **FINDINGS & LEARNINGS:**

From the above experimentation and discussion we can find out that for any point **A(X,Y)** to be translated to point **B(X',Y')** we define the translation matrix as

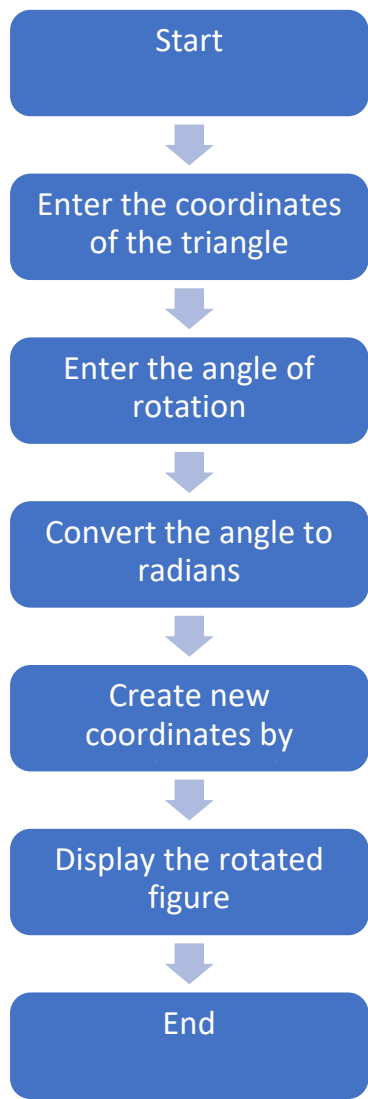
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

The following learning find a wide range of application as follows:-

- Applications is viewport transformation
- Applications in 3D animations
- Applications in designing computer games.
- Image scaling
- Pixel art
- And many more such interesting applications

EXPERIMENT 13

- **Aim:~** WAP to implement 2D Rotation
- **Objective:** To enter the coordinates of a triangle from the user, and rotate the figure by an angle entered by the user
- **Flow Chart:**



- **Code:**

```
#include<graphics.h>
#include<stdlib.h>
#include<stdio.h>
```

```

#include<conio.h>

#include<math.h>

void main()
{
    int gm , gd=DETECT;
    initgraph(&gd, &gm , "C:\\TurboC3\\BGI");
    int x1 , x2 , x3 , y1 , y2 , y3;
    printf("Program for basic Rotation\nEnter the points of triangle\n");
    setcolor(1);
    scanf("%d%d%d%d%d%d", &x1 , &y1 , &x2 , &y2 , &x3 , &y3);

    line(x1 , y1 , x2, y2);
    line(x2 , y2 , x3 , y3);
    line(x1 , y1 , x3, y3);

    printf("Enter the angle of rotation:\n");
    int rq;
    scanf("%d" , &rq);
    rq=3.14*rq/180;
    int nx1 , nx2, nx3 , ny1 , ny2 , ny3 ;

    nx1= abs(x1*cos(rq)+y1*sin(rq));
    ny1= abs(x1*sin(rq)+y1*cos(rq));
    nx3= abs(x3*cos(rq)+y3*sin(rq));
    ny2= abs(x2*sin(rq)+y2*cos(rq));
    nx2= abs(x2*cos(rq)+y2*sin(rq));
    ny3= abs(x3*sin(rq)+y3*cos(rq));

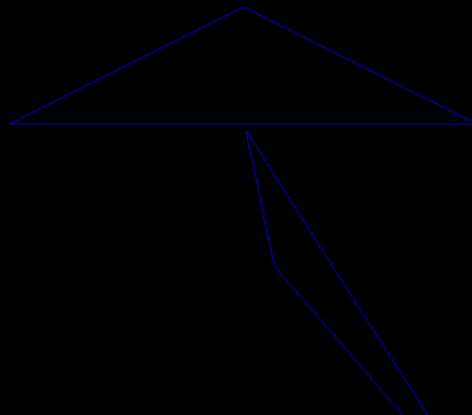
    line(nx1 , ny1 , nx2 , ny2);

```

```
    line(nx2 , ny2 , nx3 , ny3);  
    line (nx3 , ny3 , nx1 , ny1);  
    getch();  
    closegraph();  
}
```

➤ **Output:**

```
Program for basic Rotation  
Enter the points of triangle  
320 200  
480 280  
160 280  
Enter the angle of rotation:  
90
```



EXPERIMENT 14

➤ **AIM:**

WAP to implement scaling an object

➤ **DESCRIPTION OF AIM AND RELATED THEORY:**

Transformation is a process of converting the original picture coordinates into a different set of picture coordinates by applying certain rules to change their position and/or structure. When a transformation takes place on a 2D plane, it is called 2D transformation.

Scaling is a type of linear transformation which is used to change the size of an object. In the scaling process, the dimensions of the object are either expanded or compressed. Scaling can be achieved by multiplying the original coordinates of the object with the scaling factor to get the desired result. When $(S_x, S_y) < 1$, the image is compressed, else it is enlarged.

Let us assume that in a two dimensional system, the original coordinates are (X, Y) , the scaling factors are (S_x, S_y) , and the produced coordinates are (X', Y') .

This can be mathematically represented as shown below:

$$X' = X \cdot S_x \text{ and } Y' = Y \cdot S_y$$

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 –

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

OR

$$P' = P \cdot S$$

Where S is the scaling matrix. The scaling process is shown in the following figure.

Scaling is of two types:

- Uniform: when both the scaling factors are the same. It is a linear transformation that enlarges (increases) or shrinks (diminishes) objects by a scale factor that is the same in all directions

- Non-uniform: is obtained when at least one of the scaling factors is different from the others; a special case is directional scaling or stretching (in one direction). Non-uniform scaling changes the shape of the object.

➤ **ALGORITHM:**

1. Start
2. Initialize the graphics mode.
3. Plot a 2D object using given coordinates (x,y)
4. Get the scaling value Sx,Sy
5. Resize the object with Sx,Sy ($x'=x*Sx, y'=y*Sy$)
6. Plot the new object using new coordinates (x',y')

➤ **CODE:**

```
#include <iostream.h>
#include <conio.h>
#include <graphics.h>
#include <math.h>

char IncFlag;

int PolygonPoints[4][2] = {{10,10},{10,100},{100,100},{100,10}};

float Sx=0.5;
float Sy=2.0;

void PolyLine()
{
    intiCnt;
    cleardevice();
    line(0,240,640,240);
    line(320,0,320,480);
    for (iCnt=0; iCnt<4; iCnt++)
    {
        line(320+PolygonPoints[iCnt][0],240-PolygonPoints[iCnt][1],
```

```

        320+PolygonPoints[(iCnt+1)%4][0],240-PolygonPoints[(iCnt+1)%4][1]);
    }
}

void Scale()
{
    intiCnt;
    intTx,Ty;
    cout<<endl;
    for (iCnt=0; iCnt<4; iCnt++)
    {
        PolygonPoints[iCnt][0] *= Sx;
        PolygonPoints[iCnt][1] *= Sy;
    }
}

void main()
{
    intgDriver = DETECT, gMode;
    intiCnt;
    initgraph(&gDriver, &gMode, "C:\\TURBOC3\\BGI");
    PolyLine();
    getch();
    Scale();
    PolyLine();
    getch();
}

```

➤ **RESULT/OUTPUT:**

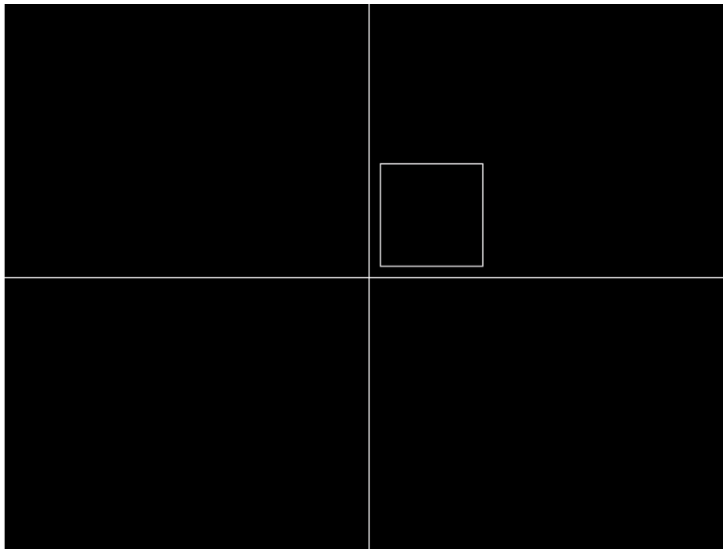


IMAGE 1: OBJECT BEFORE SCALING

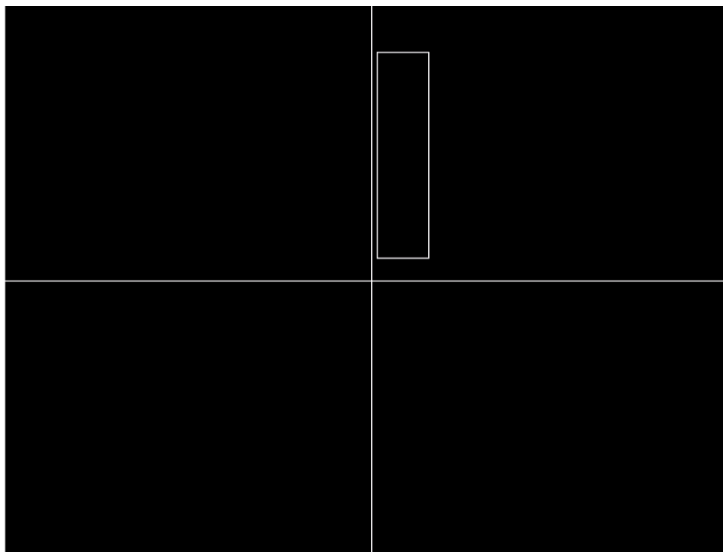


IMAGE 2: OBJECT AFTER SCALING WITH $S_x = 0.5$ AND $S_y = 2.0$

➤ **DISCUSSION:**

The output in Image 1 shows the object before scaling. After applying a scaling factor S_x of 0.5 and S_y of 2.0, the object with new vertices is produced as in Image 2.

➤ **FINDING AND LEARNING:**

From the above output we observe that when the value of S_x is 0.5, the length parallel to x-axis is reduced to half. Similarly, when the value of S_y is 2.0, the length parallel to y-axis is doubled. This causes a change in the shape of the object, and is an example of non-uniform scaling. Thus we see that scaling of an object has several applications, one of which is the window to view port transformation.

EXPERIMENT 16

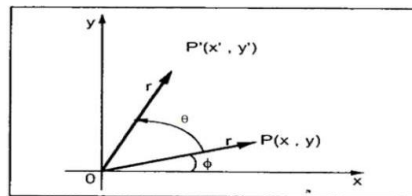
➤ **AIM:**

Write a program to implement Rotation of Object.

➤ **Theory and Description:**

In rotation, we rotate the object at particular angle θ (theta) from its origin. From the following figure, we can see that the point $P(X, Y)$ 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 $P'(X', Y')$.



Using standard trigonometric the original coordinate of point $P(X, Y)$ can be represented as –

$$X = r \cos \phi \dots (1)$$

$$Y = r \sin \phi \dots (2)$$

Same way we can represent the point $P'(X', Y')$ as –

$$x' = r \cos(\phi + \theta) = r \cos \phi \cos \theta - r \sin \phi \sin \theta \dots (3)$$

$$y' = r \sin(\phi + \theta) = r \cos \phi \sin \theta + r \sin \phi \cos \theta \dots (4)$$

Substituting equation (1) & (2) in (3) & (4) respectively, we will get

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

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

Representing the above equation in matrix form we have the Rotation matrix R as,

$$[X'Y'] = [XY] \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \text{OR}$$

$$P' = P \cdot R$$

Where R is the rotation matrix

$$R = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

➤ **Algorithm:**

1. Start
2. Initialize the graphics mode.
3. Construct a 2D object (use DrawLine()) e.g. (x , y))
4. Rotation
 - a. Get the Rotation angle
 - b. Rotate the object by the angle

$$x' = x \cos \phi - y \sin \phi$$

$$y' = x \sin \phi + y \cos \phi$$
 - c. Plot (x',y')

➤ **Code:**

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<graphics.h>
#include<math.h>
#define ROUND(a) ((int)(a+0.5))

void ddaline(int x1, int y1, int x2, int y2,int color)
{
    float xsteps, ysteps, x=x1, y=y1;
    int dx = x2-x1;
    int dy = y2-y1;
    int steps,k=1;
    if(abs(dx)>=abs(dy))
        steps=abs(dx);
    else steps=abs(dy);

    xsteps= dx/(float)steps;
    ysteps= dy/(float)steps;

    putpixel(ROUND(x),ROUND(y),color);

    while(k<=steps)
    {
        x+=xsteps;
        y+=ysteps;
        putpixel(ROUND(x), ROUND(y),color);
        k++;
    }
}

void rotate(int x1, int y1, int x2, int y2, float theta)
{
    int xtmp, ytmp;
    float th = (3.14 * theta )/180;
    xtmp = x1 + ROUND ((x2-x1)*cos(th) - (y2-y1)*sin(th));
    ytmp = y1 + ROUND ((x2-x1)*sin(th) + (y2-y1)*cos(th));

    ddaline(x1,y1,x2,y2,10);
    ddaline(x1,y1,xtmp,ytmp,12);
}

int main()
{
    int x1, x2, y1, y2;
    float theta;
    int gdriver = DETECT, gmode, errorcode;
    initgraph(&gdriver, &gmode, "C:\\\\TURBOC3\\\\BGI");
    errorcode = graphresult();

    if (errorcode != grOk)
    {
        printf("Graphics error: %s\\n", grapherrormsg(errorcode));
        printf("Press any key to halt:");
        getch();
    }
}
```

```

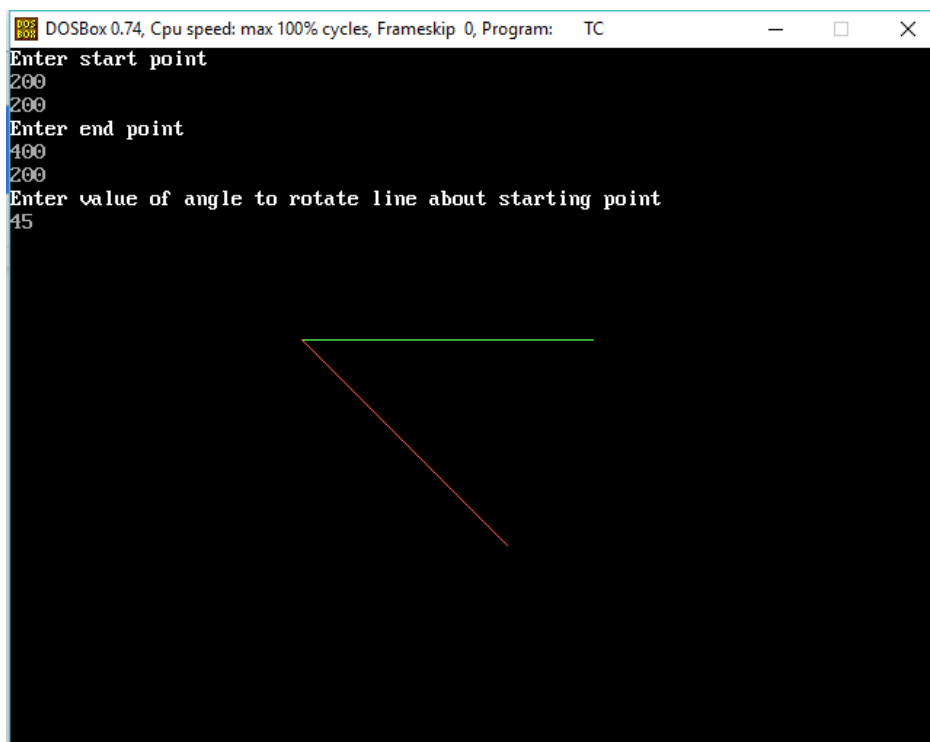
        exit(1);
    }
    printf("Enter start point\n");
    scanf("%d %d", &x1, &y1);
    printf("Enter end point\n");
    scanf("%d %d", &x2, &y2);
    printf("Enter value of angle to rotate line about starting point\n");
    scanf("%f", &theta);

    rotate(x1, y1, x2, y2, theta);

    getch();
    closegraph();
    return 0;
}

```

➤ OUTPUT:



➤ Discussion:

Green line represents initial line.

Red line represents rotated line.

The line is rotated about a fixed point which is also the point of intersections of red line and green line. The red line appears to be rotated by 45 degrees in clockwise direction because according to the screen coordinate system, the y-axis is in down direction (as the top left corner is the origin) so the screen-quadrants are the mirror reflections about horizontal axis as compared to the actual geometric quadrants.

To rotate the line in anti-clockwise direction we can simply provide negative value for the angle.

Rotation of a complex 2-D object can be visualized as the rotation of all its constituent lines about a fixed point. The figure below illustrates this

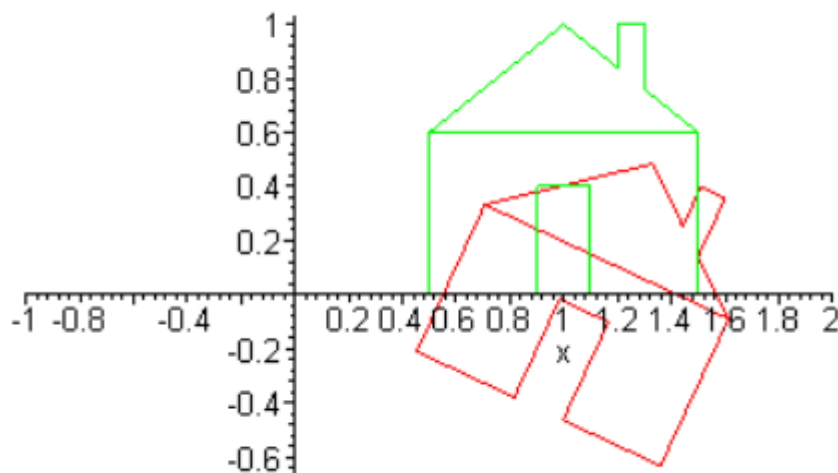


Figure: Rotation of an object by 25 degrees about origin.

➤ Findings and Learning:

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.

Rotation is one of the fundamental transformation, it plays an important role in computer graphics to change the orientation of the graphics on the screen.

A rotation matrix is a matrix that is used to perform a rotation in Euclidean space. To perform the rotation using a rotation matrix R , the position of each point must be represented by a column vector \mathbf{v} , containing the coordinates of the point. A rotated vector is obtained by using the matrix multiplication $R\mathbf{v}$.

Rotation of a complete object can be viewed as rotation of all its constituent lines about a fixed point.

Rotation matrices are square matrices, with real entries. More specifically, they can be characterized as orthogonal matrices with determinant 1.

Coordinate rotations are a natural way to express the orientation of a camera, or the attitude of a spacecraft, relative to a reference axes-set. These are also used in various image editing softwares to perform simple rotation of images.

EXPERIMENT 17

➤ **Aim:**

Write a program to implement Polygon Drawing Algorithm

➤ **Description:**

The aim of the experiment is implement drawing algorithm for polygons with given vertices.

➤ **Algorithm:**

The fundamental algorithm behind polygon drawing is as follows:

1. Traverse the list of coordinates in counter-clockwise or clockwise fashion
2. Draw the line between the current and the next point using any line drawing algorithm
3. Finally draw a line between the first and the last point to close off the polygon.

➤ **Code:**

```
#include <iostream.h>
#include <graphics.h>
#include <conio.h>
#include <math.h>

void Polygon(const int, const int []);

void Line(const int, const int, const int, const int);

int main() {
    int gd = DETECT, gm, n = 0;
    do {
        cout << "\nNumber of Points : n :";
        cout << "\nEnter the value of n (n>=3) : ";
        cin >> n;
        int *coordinates = new int[(n * 2)];

        for (int count = 0; count < n; count++) {
            cout << "Coordinates of Point-" << (count + 1) << " (x" << (count + 1) << ",y" <<
(count + 1) << ") :";
            cout << "Enter the value of x" << (count + 1) << " = ";
            cin >> coordinates[(count * 2)];
            cout << "Enter the value of y" << (count + 1) << " = ";
            cin >> coordinates[((count * 2) + 1)];
        }

        initgraph(&gd, &gm, "C:\\TURBOC3\\BGI");
        setcolor(15);
```

```

    Polygon(n, coordinates);
    delete coordinates;
    setcolor(15);
    outtextxy(110, 460, "Press <Enter> to continue or any other key to exit.");
    int key = int(getch());
    if (key != 13)
        break;
    else closegraph();
} while (1);
closegraph();
return 0;
}

```

```

void Polygon(const int n, const int coordinates[]) {
    if (n >= 2) {
        Line(coordinates[0], coordinates[1],
            coordinates[2], coordinates[3]);

        for (int count = 1; count < (n - 1); count++)
            Line(coordinates[(count * 2)], coordinates[((count * 2) + 1)],
                coordinates[((count + 1) * 2)],
                coordinates[((count + 1) * 2) + 1]);
        Line(coordinates[0], coordinates[1],
            coordinates[(count * 2)], coordinates[(count * 2) + 1]);
    }
}

```

```

void Line(const int x_1, const int y_1, const int x_2, const int y_2) {
    int color = getcolor();

    int x1 = x_1;
    int y1 = y_1;

    int x2 = x_2;
    int y2 = y_2;

    if (x_1 > x_2) {
        x1 = x_2;
        y1 = y_2;

        x2 = x_1;
        y2 = y_1;
    }

    int dx = abs(x2 - x1);
    int dy = abs(y2 - y1);
    int inc_dec = ((y2 >= y1) ? 1 : -1);

    if (dx > dy) {
        int two_dy = (2 * dy);
        int two_dy_dx = (2 * (dy - dx));
        int p = ((2 * dy) - dx);
        int x = x1;
        int y = y1;
    }
}

```

```

    putpixel(x, y, color);

    while (x < x2) {
        x++;
        if (p < 0)
            p += two_dy;
        else {
            y += inc_dec;
            p += two_dy_dx;
        }

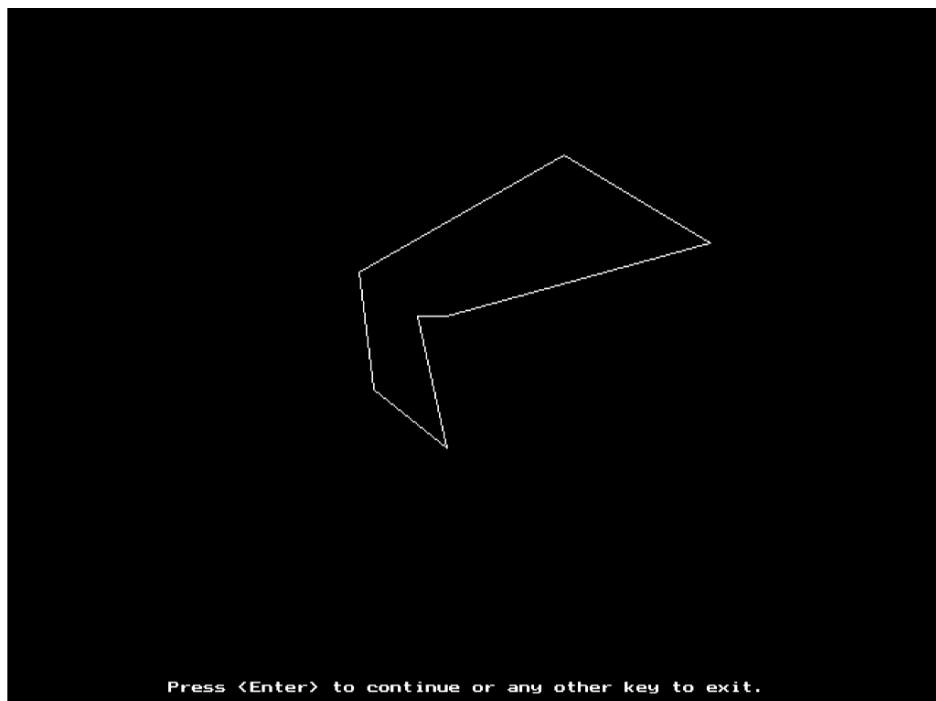
        putpixel(x, y, color);
    } else {
        int two_dx = (2 * dx);
        int two_dx_dy = (2 * (dx - dy));
        int p = ((2 * dx) - dy);
        int x = x1;
        int y = y1;

        putpixel(x, y, color);

        while (y != y2) {
            y += inc_dec;
            if (p < 0)
                p += two_dx;
            else {
                x++;
                p += two_dx_dy;
            }
            putpixel(x, y, color);
        }
    }
}

```

➤ **Result/Output :**



➤ **Discussion:**

The algorithm is relatively simple and intuitive and is similar to how humans draw polygons. We simply cycle through the coordinates in a clockwise order and draw lines between two adjacent vertices and finally between first and last vertex to close the polygon.

➤ **Finding and Learning:**

We learnt about the Polygon Drawing Algorithm and found out that other line drawing algorithms such as Bresenham's and Symmetric DDA can be used for the same. Also we learnt that polygon drawing has a lot of use cases in computer graphics including 2D maps constructions and geofence drawing.