Name: Gaurav Kishor Patil
Div: 2 Batch: C
Roll No: 54

Experiment No:1

Aim: To implement DDA algorithms for drawing a line segment between two given end points.

Objective: Draw the line using (vector) generation algorithms which determine the pixels that should be turned ON are called as digital differential analyzer (DDA).It is one of the techniques for obtaining a rasterized straight line. This algorithm can be used to draw the line in all the quadrants.

Theory:
DDA algorithm is an incremental scan conversion method. Here we perform calculations at each step using the results from the preceding step. The characteristic of the DDA algorithm is to take unit steps along one coordinate and compute the corresponding values along the other coordinate. Digital Differential Analyzer (DDA) algorithm is the simple line generation algorithm which is explained step by step here.

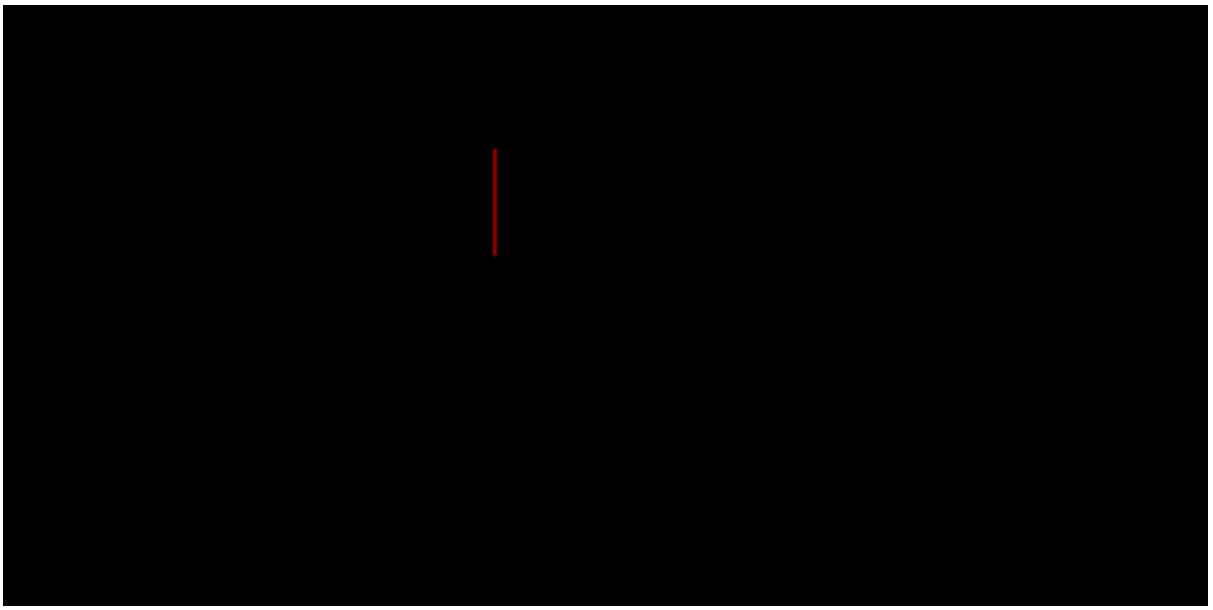
Algorithm:

Program:

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

void main() {
 int gd = DETECT, gm;
 int dx, dy, steps,k;
 float xinc, yinc, x, y;
 int x1, y1, x2, y2;
printf("Enter the coordinates of the first point (x1, y1):\n");
 scanf("%d %d", &x1, &y1);
 printf("Enter the coordinates of the second point (x2, y2):\n");
 scanf("%d %d", &x2, &y2);
 dx = x2 - x1;
 dy = y2 - y1;
 if (abs(dx) > abs(dy))
```

```
steps = abs(dx);
else
steps = abs(dy);
xinc = dx / (float) steps;
yinc = dy / (float) steps;
x = x1;
y = y1;

initgraph(&gd, &gm, "C:\\TURBOC3\\BGI");
for ( k = 1; k <= steps; k++) {
putpixel(abs(x),abs(y),RED);
x = x + xinc;
y = y + yinc;

}
getch();
getch();
closegraph();
}
```

Output:



Conclusion: Comment on -

1. Pixel
2. Equation for line
3. Need of line drawing algorithm
4. Slow or fast

Name: Gaurav Kishor Patil
Div: 2 Batch: C
Roll No: 54

Experiment No:2

Aim: To implement Bresenham's algorithms for drawing a line segment between two given end points.

Objective:
Draw a line using Bresenham's line algorithm that determines the points of an n-dimensional raster that should be selected to form a close approximation to a straight line between two points

Theory:
In Bresenham's line algorithm pixel positions along the line path are obtained by determining the pixels i.e. nearer the line path at each step.

Algorithm -
(x1,y1,x0,y0)
dx=x1-x0
dy=y1-y0
p0=2dy-dx
for k=0 to dx do
if pk<0 then
putpixel(xi+1,yi)
pn=pk+2dy

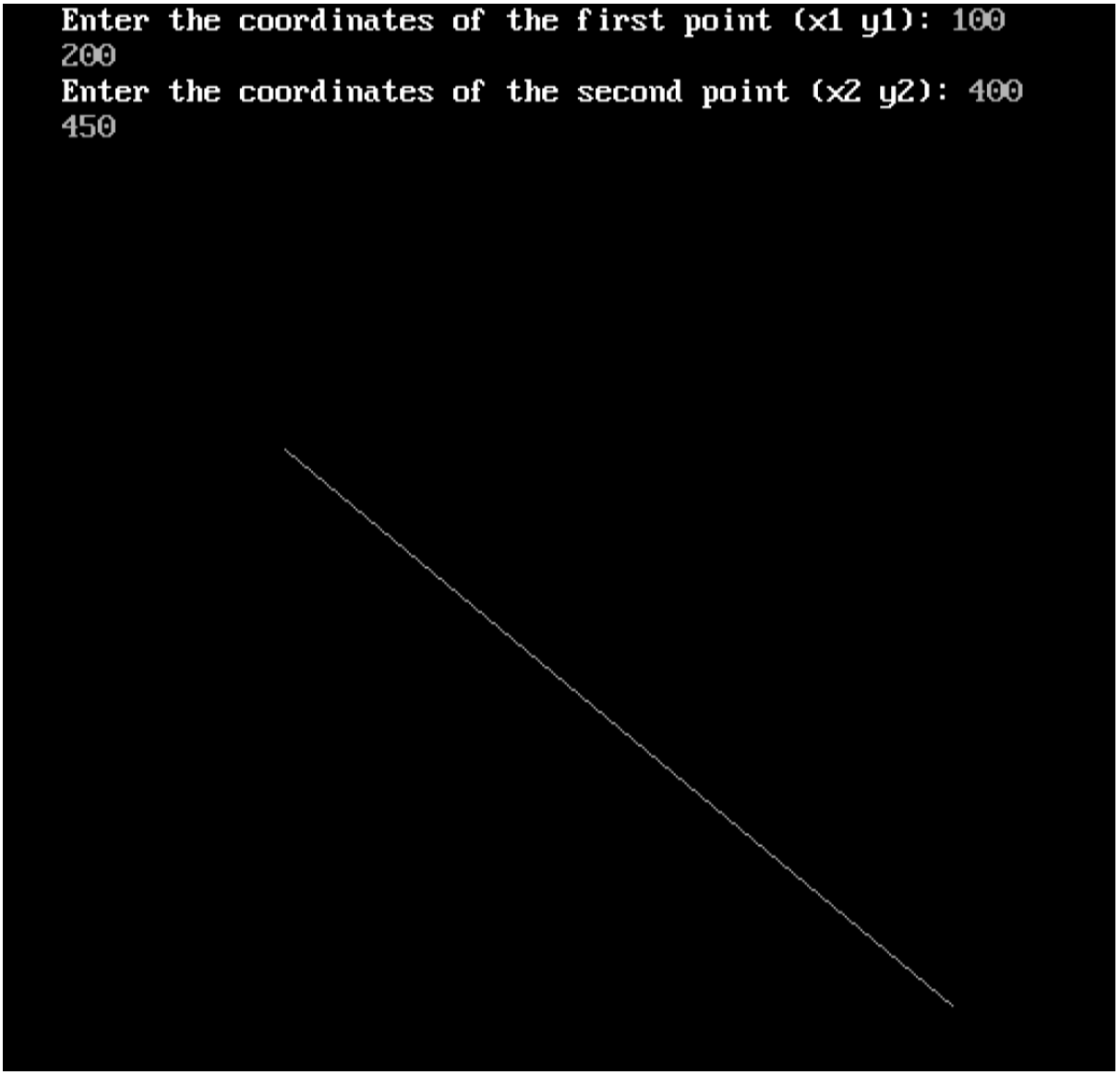
```
else
putpixel(xi+1,yi+1)
pn=pk+(2dy-2dx)
end
```

Program -

```
- #include <stdio.h>
#include <conio.h>
#include <graphics.h>
void Bresenham(int x1, int y1, int x2, int y2) {
 int dx, dy, x, y, p, end;
 dx = abs(x1 - x2);
 dy = abs(y1 - y2);
 p = 2 * dy - dx;
 if (x1 > x2) {
 x = x2;
 y = y2;
 end = x1;
 } else {
 x = x1;
 y = y1;
 end = x2;
 }
 putpixel(x, y, 7);
 while (x < end) {
 x = x + 1;
 if (p < 0) {
 p = p + 2 * dy;
 } else {
 y = y + 1;
 p = p + 2 * (dy - dx);
 }
 putpixel(x, y, 7);
 }
}
int main() {
 int gd = DETECT, gm;
 initgraph(&gd, &gm, RED);
 int x1, y1, x2, y2;
 printf("Enter the coordinates of the first point (x1 y1): ");
 scanf("%d %d", &x1, &y1);
 printf("Enter the coordinates of the second point (x2 y2): ");
 scanf("%d %d", &x2, &y2);
 Bresenham(x1, y1, x2, y2);

 getch();
 closegraph();
```

```
 return 0;
}
```

Output –



Conclusion: Comment on -

1. Pixel

2. Equation for line
3. Need of line drawing algorithm
4. Slow or fast

Name: Gaurav Kishor Patil
Div: 2 Batch: C
Roll No: 54

Experiment No:3

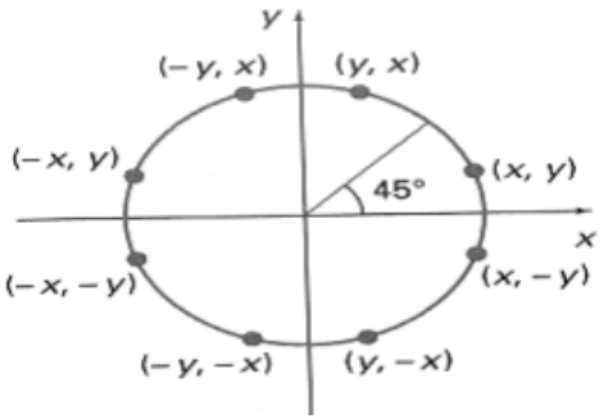
Aim: To implement midpoint circle algorithm.

Objective:

Draw a circle using mid-point circle drawing algorithm by determining the points needed for rasterizing a circle. 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.

Theory:

The shape of the circle is similar in each quadrant. We can generate the points in one section and the points in other sections can be obtained by considering the symmetry about x-axis and y-axis.

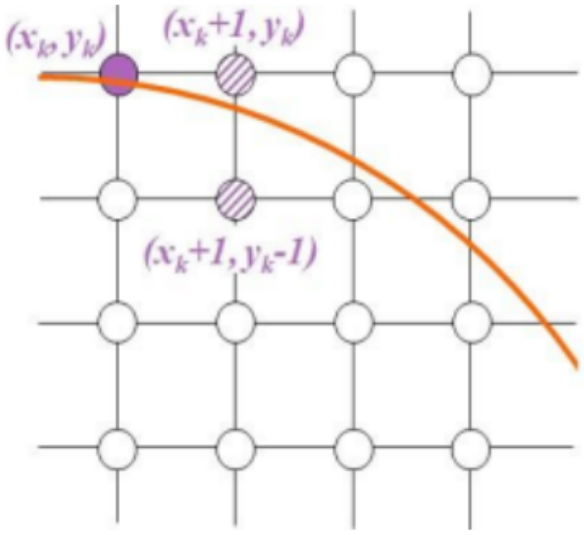


The equation of circle with center at origin is x 2 + y 2 = r 2

Let the circle function is f circle (x, y) -

- is < 0, if (x, y) is inside circle boundary,
- is = 0, if (x, y) is on circle boundary,
- is > 0, if (x, y) is outside circle boundary.

Consider the pixel at (xk, yk) is plotted,



Now the next pixel along the circumference of the circle will be either (xk + 1, yk) or (xk + 1, yk – 1) whichever is closer the circle boundary.

Let the decision parameter pk is equal to the circle function evaluate at the mid-point between two pixels.

If pk < 0, the midpoint is inside the circle and the pixel at yk is closer to the circle boundary.

Otherwise, the midpoint is outside or on the circle boundary and the pixel at yk – 1 is closer to the circle boundary.

Algorithm –

MIDPOINT_CIRCLE:

x=0

y=r

p=1-r

EightWaySummetry(x,y)

putpixel(x,y)

putpixel(x,-y)

putpixel(-x,y)

putpixel(-x,-y)

putpixel(y,x)

putpixel(y,-x)

putpixel(-y,x)

putpixel(-y,-x)

while x<y do

if p<0

p=p+2x+3

else

p=p+2x-2y+5

y=y-1

end

x=x+1

end

Program –

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
void main()
{
 int x,y,p,r,xc,yc;
 int gd=DETECT,gm;
 initgraph(&gd,&gm,"C:\\TURBOC3\\BGI");
 printf("Enter xc,yc:");
 scanf("%d%d",&xc,&yc);
 printf("Enter radius:");
 scanf("%d",&r);
 x=0;
 y=r;
 p=1-r;
 do
 {
   putpixel(xc+x,yc+y,RED);
   putpixel(xc+y,yc+x,RED);
   putpixel(xc-y,yc+x,RED);
   putpixel(xc-x,yc+y,RED);
   putpixel(xc-x,yc-y,RED);
   putpixel(xc-y,yc-x,RED);
   putpixel(xc+y,yc-x,RED);
```

```
    putpixel(xc+x,yc-y,RED);
    if(p<0)
    {
    p=p+(2*x)+3;
    }
    else
    {
    y=y-1;
    p=p+(2*x)-(2*y)+1;
    }
    x=x+1;
    }while(y>x);
    getch();
    closegraph();
}
```

output –

Conclusion: Comment on

1. Fast or slow

2. Draw one arc only and repeat the process in 8 quadrants

3. Difference with line drawing method

Name: Gaurav Kishor Patil
Div: 2 Batch: C
Roll No: 54

Experiment No:4

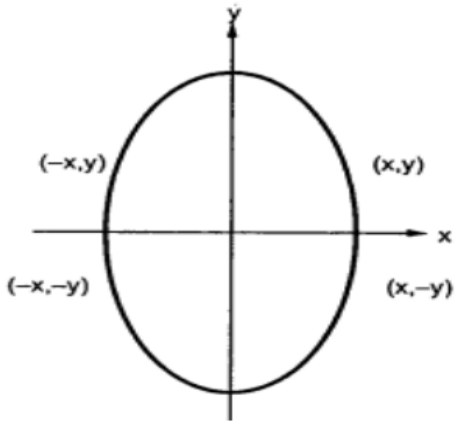
Aim-To implement midpoint Ellipse algorithm

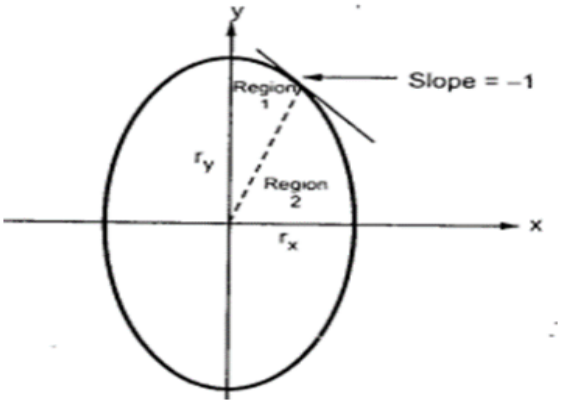
Objective:

Draw the ellipse using Mid-point Ellipse algorithm in computer graphics. Midpoint ellipse algorithm plots (finds) points of an ellipse on the first quadrant by dividing the quadrant into two regions.

Theory:

Midpoint ellipse algorithm uses four way symmetry of the ellipse to generate it. Figure shows the 4-way symmetry of the ellipse.





Here the quadrant of the ellipse is divided into two regions as shown in the fig. Fig. shows the

division of first quadrant according to the slope of an ellipse with rx < ry. As ellipse is drawn from 90 0 to 0 0 , x moves in positive direction and y moves in negative direction and ellipse passes through two regions 1 and 2.

The equation of ellipse with center at (xc, yc) is given as -

[(x – xc) / rx] 2 + [(y – yc) / ry] 2 = 1

Therefore, the equation of ellipse with center at origin is given as -

[x / rx] 2 + [y / ry] 2 = 1

i.e. x 2 ry 2 + y 2 rx 2 = rx 2 ry 2

Let, f ellipse (x, y) = x2 ry2 + y2 rx2 - rx2 ry2

Algorithm:

```
x=0,y=ry

p1=ry*ry-ry*rx*rx+rx*rx/4


dx=2*x*ry*ry

dy=2*y*rx*rx

while(dx<dy)

{

putpixel(x,y)

if(p1<0)

{

p=p1+2*ry*ry*x+ry*ry

x=x+1

}

else

{

p=p1+2*ry*ry/*x+ry*ry-2*rx*rxy
```

```
x=x+1
y=y-1
}
p2=ry*ry*(x+1/2)+rx*rx*(y-1)*(y-1)-rx*rx*ry*ry
while(y>=0)
{
putpixel(x,y)



if(p2<0)
{
p=p2+rx*rx-2rx*rx*y
y=y-1
x=x+1
}
else
{
P=p2+rx*rx-2*rx*rx*y+2*ry*ry*x
Y=y-1
}
}
```

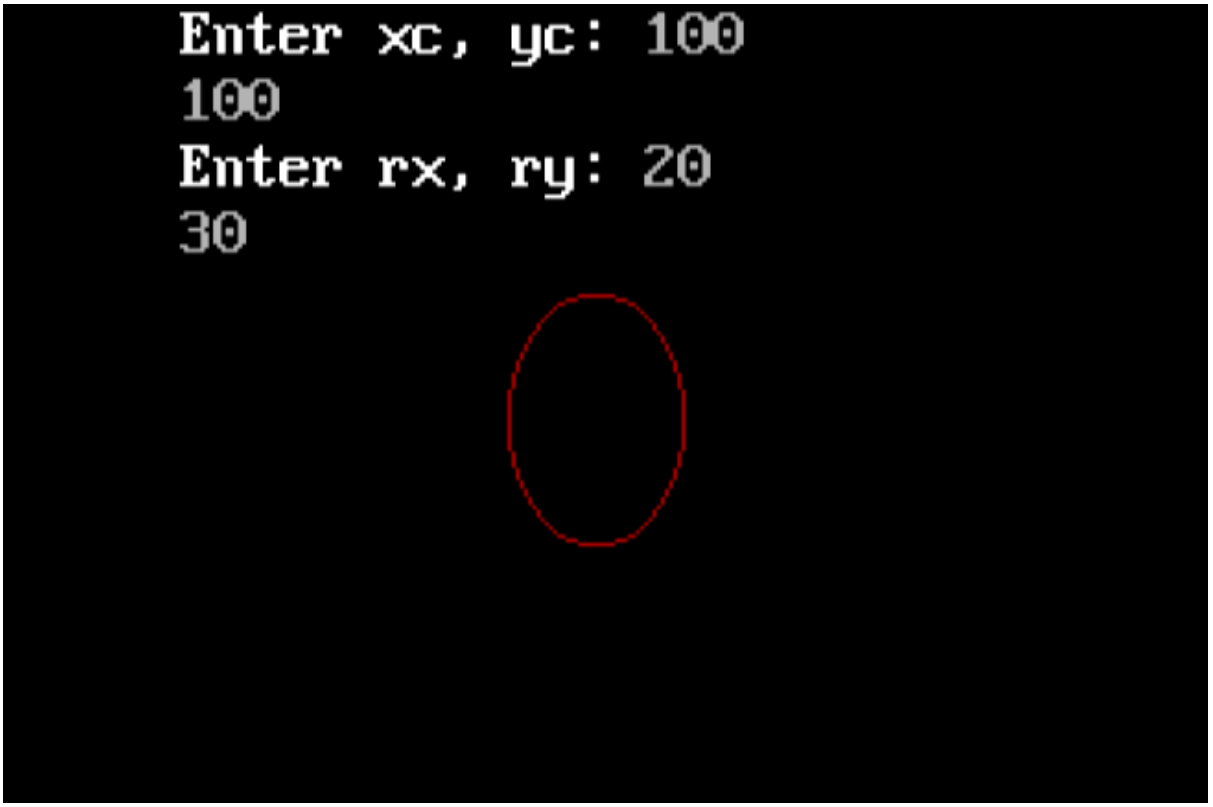
Program:

```
#inlcude <stdio.h>
#include<graphics.h>
#include<conio.h>
#include<math.h>
```

```
void main()
{
int p,rx,ry,xc,yc,x,y;
int gd= DETECT, gm;
initgraph(&gd, &gm, "C:\\TURBOC3\\BGI");
printf("Enter xc, yc");
scanf("%d %d", &xc, &yc);
printf("Enter rx, ry");
scanf("%d %d", &rx, &ry);
x=0;
y = ry;
p = (ry*ry)- (ry * rx * rx)+((rx * rx)/4);
while((2*x*ry*ry)<(2*y*rx*rx))
{
putpixel(xc + x, yc +y, RED);
putpixel(xc - x, yc +y, RED);
putpixel(xc + x, yc -y, RED);
putpixel(xc - x, yc -y, RED);
if(p<0)
{
p = p+(2*x*ry*ry) + (ry * ry);
x = x + 1;
}
else{
p = p+ (2*x*ry*ry) + (ry * ry) - (2*y*rx*rx);
x= x+1;
y= y - 1;
```

```
}
}
p = (x +0.5)* (x +0.5)*(ry * ry) *(y-1)*(y-1)-(rx * rx * ry * ry);
while(y>=0)
{
putpixel(xc + x, yc +y, RED);
putpixel(xc - x, yc +y, RED);
putpixel(xc + x, yc -y, RED);
putpixel(xc - x, yc -y, RED);
if(p<0)
{
p = p+ (rx * rx) - (2 * rx * rx * y);
y= y-1;
}
else
{
p = p + (rx * rx) - (2 * rx * rx * y) + ( 2* ry * ry * x);
y= y-1;
x = x+1;
}
getch();
closegraph();
}
```

Output:



Conclusion: Comment on

1. Slow or fast
2. Difference with circle
3. Importance of object

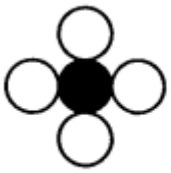
Name: Gaurav Kishor Patil
Div: 2 Batch: C
Roll No: 54

Experiment No:5

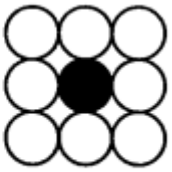
Aim: To implement Area Filling Algorithm: Boundary Fill, Flood Fill.

Objective:
Polygon is an ordered list of vertices as shown in the following figure. For filling polygons with particular colors, we need to determine the pixels falling on the border of the polygon and those which fall inside the polygon. Objective is to demonstrate the procedure for filling polygons using different techniques.

Theory:
1) Boundary Fill algorithm –
Start at a point inside a region and paint the interior outward toward the boundary. If the boundary is specified in a single color, the fill algorithm processed outward pixel by pixel until the boundary color is encountered. A boundary-fill procedure accepts as input the coordinate of the interior point (x, y), a fill color, and a boundary color.



(a) Four connected region



(b) Eight connected region

Procedure:

```
boundary_fill (x, y, f_color, b_color)
{
if (getpixel (x, y) != b_colour && getpixel (x, y) != f_colour)
        {
        putpixel (x, y, f_colour)
        boundary_fill (x + 1, y, f_colour, b_colour);
        boundary_fill (x, y + 1, f_colour, b_colour);
        boundary_fill (x - 1, y, f_colour, b_colour);
        boundary_fill (x, y - 1, f_colour, b_colour);
        }
}
```

Program:

1.
a.4-way

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

void bf(int,int,int,int);
void main()
{
int gd=DETECT,gm;
initgraph(&gd,&gm,"");
setcolor(10);
rectangle(50,50,100,100);
bf(70,70,10,15);
getch();
closegraph();

}
void bf(int x,int y,int fcolor,int bcolor)
{
int ccolor=getpixel(x,y);
if((ccolor!=fcolor) && (ccolor!=bcolor))
{
putpixel(x,y,fcolor);
```

```
delay(5);
bf(x,y-1,fcolor,bcolor);
bf(x,y+1,fcolor,bcolor);
bf(x+1,y,fcolor,bcolor);
bf(x-1,y,fcolor,bcolor);
}

}
```

2.
.8-way

```
#include<stdio.h>

#include<conio.h>
#include<graphics.h>

void bf(int,int,int,int);
void main()
{
int gd=DETECT,gm;
initgraph(&gd,&gm,"");
setcolor(10);
rectangle(50,50,100,100);
bf(70,70,10,15);
getch();
closegraph();

}
void bf(int x,int y,int fcolor,int bcolor)
{
int ccolor=getpixel(x,y);
if((ccolor!=fcolor) && (ccolor!=bcolor))
{
putpixel(x,y,fcolor);
delay(5);
bf(x,y-1,fcolor,bcolor);
bf(x,y+1,fcolor,bcolor);
bf(x+1,y,fcolor,bcolor);
bf(x-1,y,fcolor,bcolor);
bf(x+1,y-1,fcolor,bcolor);
bf(x+1,y+1,fcolor,bcolor);
bf(x-1,y-1,fcolor,bcolor);
bf(x-1,y+1,fcolor,bcolor);
}

}
```

Output:

For boundary fill

2) Flood Fill algorithm –

Sometimes we want to fill an area that is not defined within a single color boundary. We paint such areas by replacing a specified interior color instead of searching for a boundary color value. This approach is called a flood-fill algorithm.

1. We start from a specified interior pixel (x, y) and reassign all pixel values that are currently set to a given interior color with the desired fill color.
2. If the area has more than one interior color, we can first reassign pixel values so that all interior pixels have the same color.
3. Using either 4-connected or 8-connected approach, we then step through pixel positions until all interior pixels have been repainted.

(a) Four connected region (b) Eight connected region

Procedure -

```
flood_fill (x, y, old_color, new_color)
{
if (getpixel (x, y) = old_colour)
        {
        putpixel (x, y, new_colour);
        flood_fill (x + 1, y, old_colour, new_colour);
```

```
        flood_fill (x - 1, y, old_colour, new_colour);
        flood_fill (x, y + 1, old_colour, new_colour);
        flood_fill (x, y - 1, old_colour, new_colour);
        flood_fill (x + 1, y + 1, old_colour, new_colour);
        flood_fill (x - 1, y - 1, old_colour, new_colour);
        flood_fill (x + 1, y - 1, old_colour, new_colour);
        flood_fill (x - 1, y + 1, old_colour, new_colour);
        }
}
```

Program:

.

4-way

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

void ff(int,int,int,int);
void main()
{
int gd=DETECT,gm;
initgraph(&gd,&gm,"");
setcolor(10);
rectangle(50,50,100,100);
ff(70,70,0,15);
getch();
closegraph();

}
void ff(int x,int y,int ocolor,int ncolor)
{

if(getpixel(x,y)==ocolor)
{
putpixel(x,y,ncolor);
delay(5);
ff(x,y-1,ocolor,ncolor);
ff(x,y+1,ocolor,ncolor);
ff(x+1,y,ocolor,ncolor);
ff(x-1,y,ocolor,ncolor);
}

}
```

8-way

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

```
#include<graphics.h>

void ff(int,int,int,int);
void main()
{
int gd=DETECT,gm;
initgraph(&gd,&gm,"");
setcolor(10);
rectangle(50,50,100,100);
ff(70,70,0,15);
getch();
closegraph();

}
void ff(int x,int y,int ocolor,int ncolor)
{

if(getpixel(x,y)==ocolor)
{
putpixel(x,y,ncolor);
delay(5);
ff(x,y-1,ocolor,ncolor);
ff(x,y+1,ocolor,ncolor);
ff(x+1,y,ocolor,ncolor);
ff(x-1,y,ocolor,ncolor);
ff(x+1,y-1,ocolor,ncolor);
ff(x+1,y+1,ocolor,ncolor);
ff(x-1,y-1,ocolor,ncolor);
ff(x-1,y+1,ocolor,ncolor);
}

}
```

Output:

For flood Fill

Conclusion: Comment on

1. Importance of Flood fill
2. Limitation of methods
3. Usefulness of method

Name: Gaurav Kishor Patil
Div: 2 Batch: C
Roll No: 54

Experiment No:6

Aim: To implement Character Generation.

Objective:

Identify the different Methods for Character Generation and generate the character using Stroke

Theory:

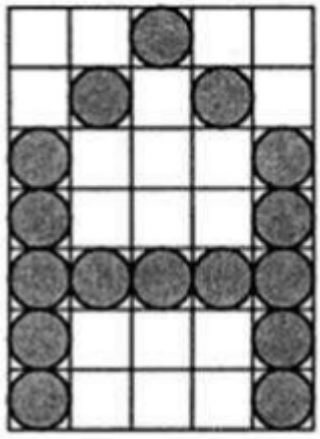
Bit map method –

Bitmap method is a called dot-matrix method as the name suggests this method use array of bits for generating a character. These dots are the points for array whose size is fixed.

· In bit matrix method when the dots are stored in the form of array the value 1 in array represent the characters i.e. where the dots appear we represent that position with numerical value 1 and the value where dots are not present is represented by 0 in array.

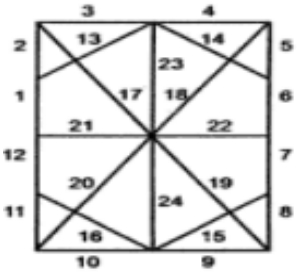
· It is also called dot matrix because in this method characters are represented by an array of dots in the matrix form. It is a two-dimensional array having columns and rows.

A 5x7 array is commonly used to represent characters. However, 7x9 and 9x13 arrays are also used. Higher resolution devices such as inkjet printer or laser printer may use character arrays that are over 100x100.

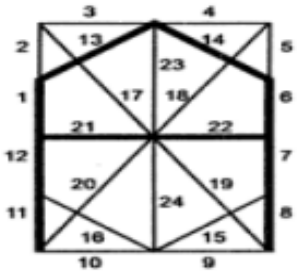


Starburst method –

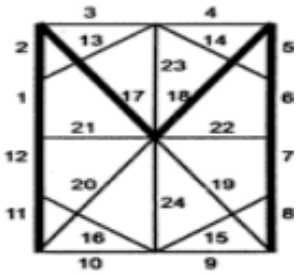
In this method a fix pattern of line segments is used to generate characters. Out of these 24-line segments, segments required to display for particular character are highlighted. This method of character generation is called starburst method because of its characteristic appearance. The starburst patterns for characters A and M. the patterns for particular characters are stored in the form of 24 bit code, each bit representing one line segment. The bit is set to one to highlight the line segment; otherwise, it is set to zero. For example, 24-bit code for Character A is 0011 0000 0011 1100 1110 0001 and for character M is 0000 0011 0000 1100 1111 0011.



a) Star burst pattern of 24 line segments



b) Star burst pattern for character A



c) Star burst pattern for character M

Program:

```
#include <stdio.h>
#include <conio.h>
#include <graphics.h>
void main(){
    int gd=DETECT,gm;
    int i,j,k;
    int a[10][10]={
              {1,1,1,1,1,1,1,1,1,1},
              {1,1,1,1,1,1,1,1,1,1},
              {1,1,0,0,0,0,0,0,0,0},
              {1,1,0,0,0,0,0,0,0,0},
              {1,1,0,0,0,1,1,1,1,1},
              {1,1,0,0,0,1,1,1,1,1},
              {1,1,0,0,0,0,0,0,1,1},
              {1,1,0,0,0,0,0,0,1,1},
              {1,1,1,1,1,1,1,1,1,1},
              {1,1,1,1,1,1,1,1,1,1}

    };
    int b[10][10]={
              {1,1,1,1,1,1,1,1,1,1},
              {1,1,1,1,1,1,1,1,1,1},
              {1,1,0,0,0,0,0,0,1,1},
              {1,1,0,0,0,0,0,0,1,1},
              {1,1,1,1,1,1,1,1,1,1},
              {1,1,1,1,1,1,1,1,1,1},
              {1,1,0,0,0,0,0,0,1,1},
              {1,1,0,0,0,0,0,0,1,1},
              {1,1,0,0,0,0,0,0,1,1},
              {1,1,0,0,0,0,0,0,1,1}
    };
    int c[10][10]={
            {1,1,0,0,0,0,0,0,1,1},
            {1,1,0,0,0,0,0,0,1,1},
            {1,1,0,0,0,0,0,0,1,1},
            {1,1,0,0,0,0,0,0,1,1},
            {1,1,0,0,0,0,0,0,1,1},
            {1,1,0,0,0,0,0,0,1,1},
            {1,1,0,0,0,0,0,0,1,1},
            {1,1,0,0,0,0,0,0,1,1},
```

```
            {1,1,1,1,1,1,1,1,1,1},
            {1,1,1,1,1,1,1,1,1,1},
    };
    int d[10][10]={
            {1,1,1,1,1,1,1,1,1,1},
            {1,1,1,1,1,1,1,1,1,1},
            {1,1,0,0,0,0,0,0,1,1},
            {1,1,0,0,0,0,0,0,1,1},
            {1,1,1,1,1,1,1,1,1,1},
            {1,1,1,1,0,0,0,0,0,0},
            {1,1,0,0,1,1,0,0,0,0},
            {1,1,0,0,0,1,1,0,0,0},
            {1,1,0,0,0,0,1,1,0,0},
            {1,1,0,0,0,0,0,0,1,1}
    };
    int e[10][10]={
            {1,1,1,1,1,1,1,1,1,1},
            {1,1,1,1,1,1,1,1,1,1},
            {1,1,0,0,0,0,0,0,1,1},
            {1,1,0,0,0,0,0,0,1,1},
            {1,1,1,1,1,1,1,1,1,1},
            {1,1,1,1,1,1,1,1,1,1},
            {1,1,0,0,0,0,0,0,1,1},
            {1,1,0,0,0,0,0,0,1,1},
            {1,1,0,0,0,0,0,0,1,1},
            {1,1,0,0,0,0,0,0,1,1}
    };
    int f[10][10]={
            {1,1,0,0,0,0,0,0,1,1},
            {1,1,0,0,0,0,0,0,1,1},
            {1,1,0,0,0,0,0,0,1,1},
            {1,1,0,0,0,0,0,0,1,1},
            {1,1,0,0,0,0,0,0,1,1},
            {1,1,0,0,0,0,0,0,1,1},
            {0,1,0,0,0,0,0,1,0,0},
            {0,0,1,0,0,0,1,0,0,0},
            {0,0,0,1,0,1,0,0,0,0},
            {0,0,0,0,1,0,0,0,0,0}
    };
    initgraph(&gd,&gm,"C:\\TurboC++\Disk\TurboC3\BGI");
    for(k=0;k<6;k++){
    for(i=0;i<10;i++){
for(j=0;j<10;j++){
    if(a[i][j]==1){
      putpixel(200+j,200+i,RED);

    }
    }
    }
    for(i=0;i<10;i++){
      for(j=0;j<10;j++){
           if(b[i][j]==1){
      putpixel(220+j,200+i,GREEN);
```

```
    }
      }
    }
       for(i=0;i<10;i++){
for(j=0;j<10;j++){
    if(c[i][j]==1){
      putpixel(240+j,200+i,RED);

    }
    }
    }
       for(i=0;i<10;i++){
for(j=0;j<10;j++){
    if(d[i][j]==1){
      putpixel(260+j,200+i,RED);

    }
    }
    }
       for(i=0;i<10;i++){
for(j=0;j<10;j++){
    if(e[i][j]==1){
      putpixel(280+j,200+i,RED);

    }
    }
    }
       for(i=0;i<10;i++){
for(j=0;j<10;j++){
    if(f[i][j]==1){
      putpixel(300+j,200+i,RED);

    }
    }
    }

    }
    getch();
    closegraph();
}
```

Output -

Conclusion: Comment on

1. different methods
2. advantage of stroke method
3. one limitation

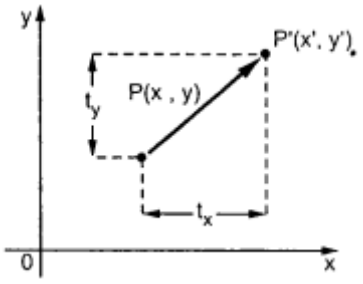
Name: Gaurav Kishor Patil
Div: 2 Batch: C
Roll No: 54

Experiment No:7

Aim: To implement 2D Transformations: Translation, Scaling, Rotation.

Objective:
To understand the concept of transformation, identify the process of transformation and application of these methods to different object and noting the difference between these transformations.

Theory:
1) Translation –
Translation is defined as moving the object from one position to another position along straight line path. We can move the objects based on translation distances along x and y axis. tx denotes translation distance along x-axis and ty denotes translation distance along y axis.



Consider (x,y) are old coordinates of a point. Then the new coordinates of that same point (x',y') can be obtained as follows:
x' = x + tx
y' = y + ty

We denote translation transformation as P. we express above equations in matrix form as:
P' = P + T , where

$$P = \begin{bmatrix} x \\ y \end{bmatrix} \quad P' = \begin{bmatrix} x' \\ y' \end{bmatrix} \quad T = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

Program:

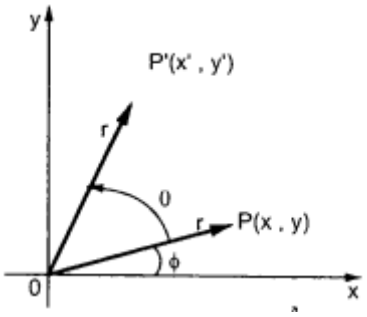
```
#include <stdio.h>
#include <conio.h>
#include <graphics.h>

int main() {
   int gd = DETECT, t, r, gm, ch, sx, sy, tx, ty, nx1, nx2, ny1, ny2;
   initgraph(&gd, &gm, "");
   line(100, 100, 200, 100);
   printf("1. translation, 2. rotation, 3. scaling:\n");
   printf("Enter your choice: ");
   scanf("%d", &ch);

   switch (ch) {
      case 1:
         printf("Enter translation factors (tx ty): ");
         scanf("%d %d", &tx, &ty);
         nx1 = 100 + tx;
         ny1 = 100 + ty;
         nx2 = 200 + tx;
         ny2 = 100 + ty;
         line(nx1, ny1, nx2, ny2);
         getch();


      case 2:
         printf("Enter angle (degrees): ");
         scanf("%d", &r);
         t = (3.14 * r) / 180;
         nx1 = (int)(100 + (100 * cos(t)));
         ny1 = (int)(100 + (100 * sin(t)));
         line(100, 100, nx1, ny1);
         getch();


      case 3:
         printf("Enter scaling factors (sx sy): ");
         scanf("%d %d", &sx, &sy);
         nx1 = 100 * sx;
```

```
        ny1 = 100 * sy;
        nx2 = 200 * sx;
        ny2 = 100 * sy;
        line(nx1, ny1, nx2, ny2);
        getch();


      }

   closegraph();
   return 0;
}
```

Output –

2) Rotation –

A rotation repositions all points in an object along a circular path in the plane centered at the pivot point. We rotate an object by an angle theta. New coordinates after rotation depend on both x and y.



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

The above equations can be represented in the matrix form as given below

$$[x'\ y'] = [x\ \ y]\begin{bmatrix}\cos\theta & \sin\theta \\ -\sin\theta & \cos\theta\end{bmatrix}$$

$$P' = P \cdot R$$

where R is the rotation matrix and it is given as

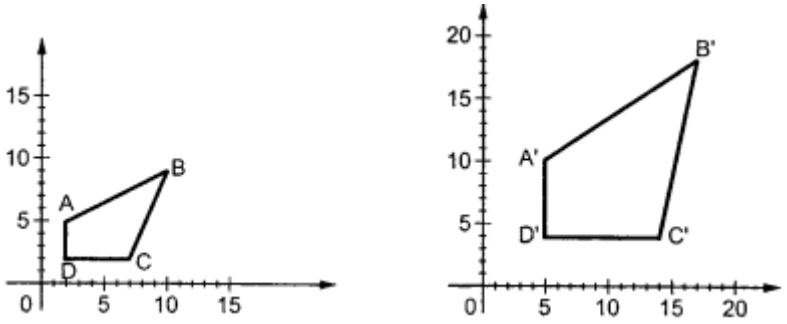
$$R = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

Program:

Output:

3) Scaling -

scaling refers to changing the size of the object either by increasing or decreasing. We will increase or decrease the size of the object based on scaling factors along x and y-axis.



If (x, y) are old coordinates of object, then new coordinates of object after applying scaling transformation are obtained as:

x' = x * Sx

y' = y * Sy

Sx and Sy are scaling factors along x-axis and y-axis. we express the above equations in matrix form as:

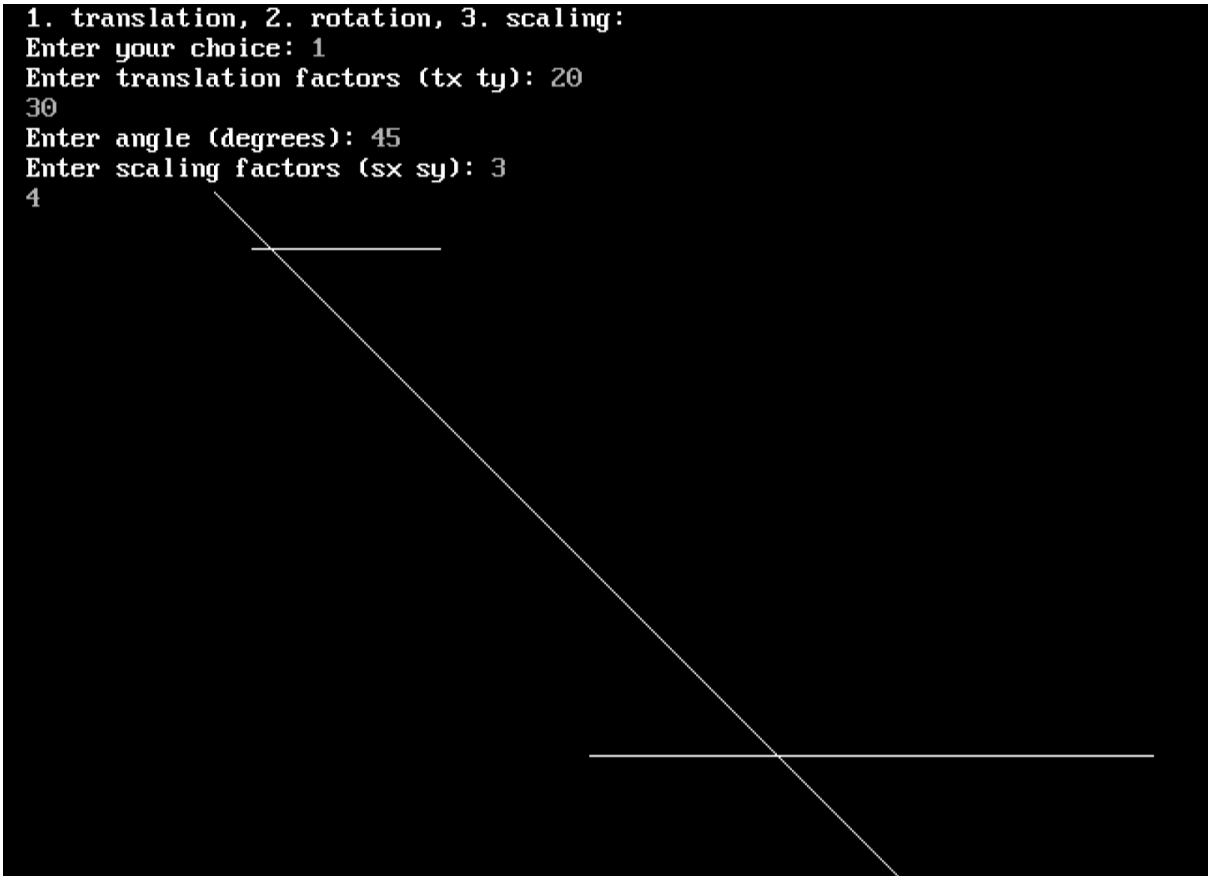
$$[x' \; y'] = [x \; y] \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix}$$
$$= [x \cdot S_x \quad y \cdot Sy]$$
$$= P \cdot S$$

Program:

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
void main()
{
int gd=DETECT,t,r,gm,ch,sx,sy,tx,ty,nx1,nx2,ny1,ny2;
initgraph(&gd,&gm,"");
line(100,100,200,100);
printf("1.translation,2.rotation,3.scalling:");
printf("enter your ch :");
scanf ("%d",&ch);
switch(ch)
{
case 1:printf("enter transition factor :");
scanf("%d %d",&tx,&ty);
nx1=100+tx;
nx1=100+ty;
nx2=200+tx;
nx2=100+ty;
line(nx1,ny1,nx2,ny2);
getch()
case 2:printf("enter angle:");
scanf("%lf",r);
t=(3.14*r)/180;
nx1=(int)(100+(100*cos(t)-(0));
ny1=(int)(100+(100*sin(t)-(0));
line(100,100,nx1,ny1);
getch();
case 3:printf("enter scalling factor :");
scanf("%d %d",&sx,&sy);
nx1=100*sx;
nx1=100*sy;
nx2=200*sx;
nx2=100*sy;
line(nx1,ny1,nx2,ny2);
getch();
default:printf("invalid");
}
getch();
closegraph();
```

}

Output -



Conclusion: Comment on :

1. Application of transformation
2. Difference noted between methods
3. Application t different object