- There is one more useful form of the line equation called the **parametric form**. In this case x and y-values on the line are given in terms of a parameter u. Suppose we want the line segment between (x_1, y_1) and (x_2, y_2), then we need the x co-ordinate to move uniformly from x_1 to x_2. This may be expressed by the equation,

$$x = x_1 + (x_2 - x_1)\,u$$

where $u = 0$ to 1

- When $u = 0$, $x = x_1$. As u increases to 1, x moves uniformly to x_2. Similarly we must have the y co-ordinate moving from y_1 to y_2 at the same time as x changes.

$$y = y_1 + (y_2 - y_1)\,u$$

- Similarly to find the length of line segment we can make use of Pythagorean theorem. See Fig. 2.2.3.Suppose point P_1 is (x_1, y_1) and P_2 is (x_2, y_2). Now if we want to find length (L) of line segment $P_1 P_2$ then

$$L^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2$$

$$L = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Similarly we can find midpoint of a line segment very easily.

$$(x_m, y_m) = \left[\left(\frac{x_1 + x_2}{2}\right), \left(\frac{y_1 + y_2}{2}\right)\right]$$

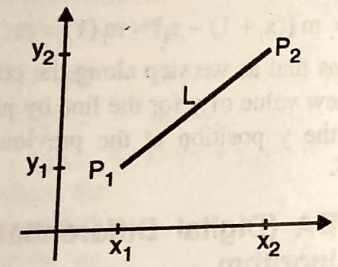


Fig. 2.2.3

Intersection of two lines

- We can determine a point where two lines can cross. By two lines crossing we mean they share some common point. That point will satisfy the equations for the both lines.
- Suppose there are two lines line 1 and line 2 and both lines are intersecting at point (x_i, y_i). Then we can represent line 1 by equation,

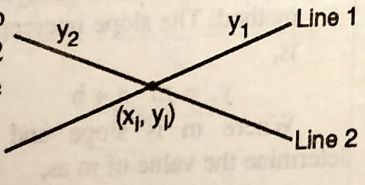


Fig. 2.2.4

$$y = m_1 x + b_1$$ and equation for line 2 as

$$y = m_2 x + b_2$$

where m_1 and m_2 are slopes of line 1 and line 2 respectively. If point (x_i, y_i) is shared by both the lines then the point (x_i, y_i) must satisfy both the line equations.

$\therefore\ y_i = m_1 x_i + b_1$ and $y_i = m_2 x_i + b_2$

$\therefore$ Equating both over y_i.

$$m_1 x_i + b_1 = m_2 x_i + b_2$$

$$m_1 x_i - m_2 x_i = b_2 - b_1$$

$$x_i = \frac{b_2 - b_1}{m_1 - m_2} \qquad \ldots(2.2.4)$$

Equating the value of x_i in any one line equation we will get y_i as,

$$y_i = m_1 x_i + b_1 = m_1\left(\frac{b_2 - b_1}{m_1 - m_2}\right) + b_1$$

$$= \frac{m_1 b_2 - m_1 b_1}{m_1 - m_2} + b_1$$

$$= \frac{m_1 b_2 - m_1 b_1 + m_1 b_1 - m_2 b_1}{m_1 - m_2}$$

$$y_i = \frac{m_1 b_2 - m_2 b_1}{m_1 - m_2}$$

$\therefore$ point $\left(\frac{b_2 - b_1}{m_1 - m_2}, \frac{m_1 b_2 - m_2 b_1}{m_1 - m_2}\right)$ is intersection point

Syllabus Topic : Scan Conversions of Point, Line, Circle and Ellipse - DDA Algorithm and Bresenham Algorithm for Line Drawing

2.3 Line Generation Algorithms

- The process of "Turning ON" the pixels for a line is called **line generation** or scan conversion of a line.
- When we want to draw a line that means we have to make the pixels on that line to ON condition i.e. we have to change the intensity of the pixels present on that line. For this task, we have two different algorithms.

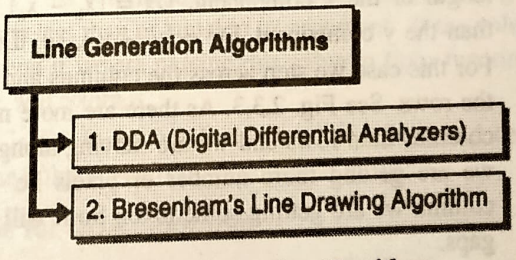


Fig. C2.1 : Line Generation Algorithms

- Both the algorithms are based on *increment method.* Let us see this increment method now.
- To draw a line we must know the two endpoints. Here we are selecting the pixels which lie near the line segment. See Fig. 2.3.1. We could try to turn on the pixels through which line is passing. But it will not be easy to find all such pixels and when we are using line drawing for big project or in animation where picture may change frequently, in that case it is very inefficient.

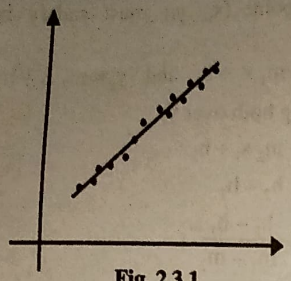
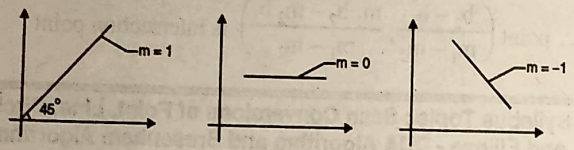
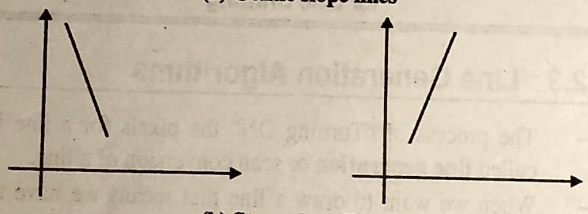


Fig. 2.3.1

- So the alternative way for this is to step along the columns of the pixels and for each column finalize which row is closest to the line. We could then turn ON the pixel in that row and column. We know how to find the row because we can place the x-value corresponding to the column into the line equation and solve for y. This will work for lines with slopes between –1 and 1 (i.e. lines which are closer to horizontal lines). Such lines are called as lines with *gentleslope*. But for *steep slope* case it will leave gaps. So we step along the rows and solve for the columns. Fig. 2.3.2 shows different lines.



(a) Gentle slope lines



(b) Steep slope lines

Fig. 2.3.2

- For *gentle slopes* ($-1 < m < 1$) there are more columns than rows. These are the line segments, where the length of the x component, $Dx = (x_b - x_a)$ is longer than the y component, $Dy = (y_b - y_a)$, i.e. $|Dx| > |Dy|$. For this case we step across the columns and solve for the rows. See Fig. 2.3.3. As there are more number of columns than rows and we are moving along columns we are getting more number of pixels i.e. on every column we are getting some pixel. So it will not leave gaps.
- For the *sharp slopes* where $|Dx| \leq |Dy|$, we step across the rows and solve for the columns. See Fig. 2.3.3.

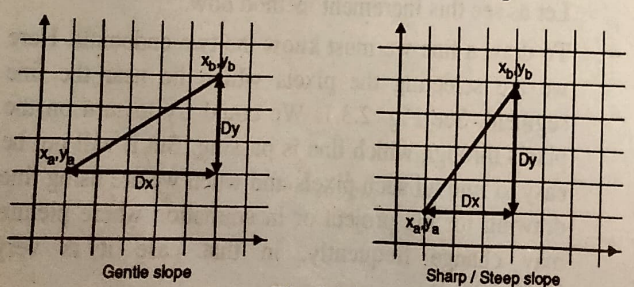


Fig. 2.3.3

- Consider the gentle slope case where we step across the columns. Each time we move from one column to the next, the value of x changes by 1. But if x always changes by exactly 1, then by what value y should change? Answer is, y will always change by exactly m (the slope). How? Let us say (x_i, y_i) is starting point. As we are considering gentle slope case means we are moving along columns i.e. x-axis, next point will be on column $(x_i + 1)$.
- If $(x_i, y_i) = (2, 3)$ then $x_i + 1 = 2 + 1$ and at that time what will be $(y_i + 1)$, that we have to find out. When we are saying that (x_i, y_i) is starting point means (x_i, y_i) satisfies equation of line i.e. $y_i = mx_i + b$. Now we know $(x_i + 1)$ and for that we have to find out $(y_i + 1)$ in such a way that $(x_i + 1, y_i + 1)$ will satisfy the equation of line i.e. $y_i + 1 = m(x_i + 1) + b$

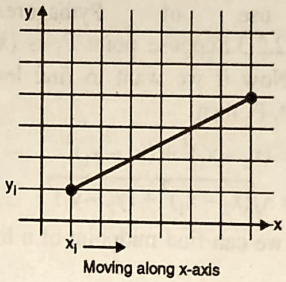


Fig. 2.3.4

$$\therefore (y_i + 1) - y_i = m(x_i + 1) + b - mx_i - b$$
$$= m[(x_i + 1) - x_i] = m(1) = m$$

- This means that as we step along the columns, we can find the new value of y for the line by just adding m or slope to the y position at the previous column. See Fig. 2.3.4.

➜ 2.3.1 DDA (Digital Differential Analyzers) Algorithm

➜ **(May 2015, May 2016, May 2017)**

Q. What are the disadvantages of DDA algorithm ?
MU - May 2015, May 2017, 5 Marks

Q. Specify the disadvantages of DDA algorithm.
MU - May 2016, 5 Marks

- Digital differential analyzer is based on incremental method. The slope intercept equation for a straight line is,

$$y = m \cdot x + b$$

Where m is slope and b is y-intercept. We can determine the value of m as,

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

$(y_2 - y_1)$ is nothing but change in y-values which is represented as Dy.

Similarly $(x_2 - x_1)$ is represented as Dx.

$$m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{Dy}{Dx}$$

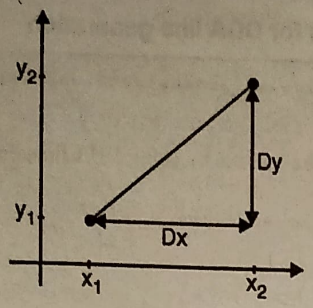


Fig. 2.3.5

Refer Fig. 2.3.5.

For lines with slope (m) is < 1 i.e. lines with positive gentle slopes, we are moving in x-direction by uniform steps of calculating the corresponding y-value by using.

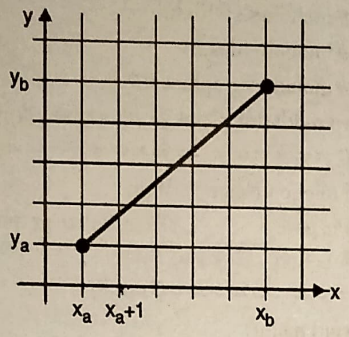


Fig. 2.3.6

$$\frac{Dy}{Dx} = m$$

$$Dy = m \cdot Dx$$

$$(y_a + 1) - (y_a) = m \cdot [(x_a + 1) - (x_a)]$$

But as we are moving in x-direction by 1 unit i.e. distance between two columns;

$(x_a + 1) - (x_a) = 1$. See Fig. 2.3.6.

$$\therefore (y_a + 1) - (y_a) = m(1)$$

$$\therefore y_a + 1 = m + y_a$$

i.e. $$y_{new} = \text{slope} + y_{old} \quad \ldots(2.3.1)$$

Similarly when the slope is > 1 i.e. lines with positive steep slopes, we are moving in y-direction by uniform steps and calculating the corresponding x-value. See Fig. 2.3.7.

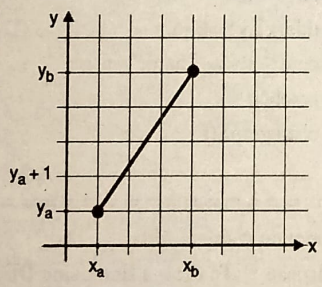


Fig. 2.3.7

$$\frac{Dy}{Dx} = m$$

i.e. $$Dx = \frac{Dy}{m}$$

$$(x_a + 1) - (x_a) = \frac{(y_a + 1) - (y_a)}{m}$$

But $[(y_a + 1) - (y_a)]$ is 1 unit i.e. distance between two rows.

$$\therefore (x_a + 1) - (x_a) = \frac{1}{m}$$

$$\therefore x_a + 1 = \frac{1}{m} + x_a$$

i.e. $$x_{new} = \frac{1}{\text{slope}} + x_{old} \quad \ldots(2.3.2)$$

When the slope (m) is = 1 then as x changes, values of y also changes. See Fig. 2.3.8.

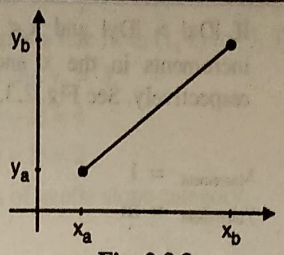


Fig. 2.3.8

i.e. $$\frac{Dy}{Dx} = m$$

but $m = 1$

$\therefore Dy = Dx$

$\therefore y_{new} = y_{old} + 1$ and $x_{new} = x_{old} + 1$

- Equation (2.3.1) and (2.3.2) are based on assumption that lines are to be processed from left end point to right side i.e. $x_{start} < x_{end}$. If this process is reversed i.e. $x_{start} > x_{end}$ then $y_{new} = y_{old} - m$ and $x_{new} = x_{old} - \frac{1}{m}$.

Since slope (m) can be any real value (floating point number) between 0 to 1, the calculated y-value for Gentle slope or calculated x-value for steep slope must be rounded to the nearest integer, because display device is having co-ordinates as integer only. See Fig. 2.3.9.

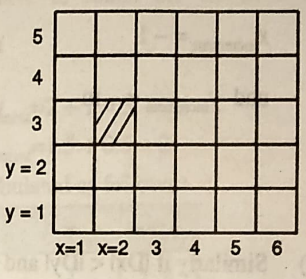


Fig. 2.3.9

- We can represent integer point as (2, 3) but we cannot represent any fractional point as (2.3, 3.1). So we need a separate function which will round that float value to integer. For this generally ceiling and flooring functions are used.
- **Ceil :** This is a function which returns smallest integer which is greater than or equal to its argument. Suppose the value is 8.6 and if we are passing this value to ceil function then that function will return 9.
- **Floor :** This is a function which returns largest integer which is less than or equal to its argument. If the value is 8.2 and if we are passing this value to floor function then it will return 8.

There is no doubt that because of this floor and ceil functions the point is diverting from actual location.

Steps for DDA algorithm

1) Accept two end points: (x_a, y_a) and (x_b, y_b)
2) Find out horizontal and vertical difference between end points.
   $Dx = x_b - x_a$ and $Dy = y_b - y_a$
3) Difference between greater magnitude determines the value of parameter steps.
4) Determine the offset needed at each step i.e. to generate the next pixel loop through this step times.
5) Before displaying every point round off the point to its nearest integer value.

6) If |Dx| > |Dy| and $x_a < x_b$, Then the values of the increments in the x and y-directions are 1 and m respectively. See Fig. 2.3.10.

$x_{increment} = 1$

$y_{increment} = m$

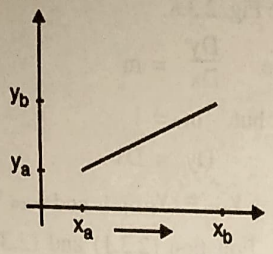


Fig. 2.3.10

7) If |Dx| > |Dy| and $x_a > x_b$, Then the decrements –1 and –m are used to generate new point. See Fig. 2.3.11.

$x_{increment} = -1$

and $y_{increment} = -m$

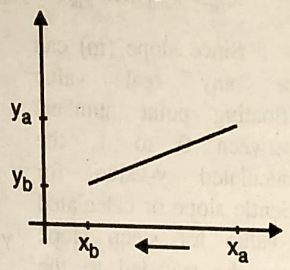
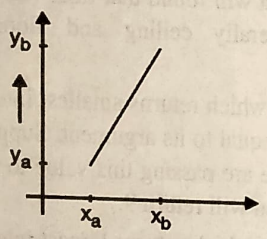


Fig. 2.3.11

8) Similarly if |Dx| < |Dy| and $y_a < y_b$

Then the values of increments in x and y-direction are 1/m and 1 respectively.

Refer Fig. 2.3.12.



$x_{increment} = \frac{1}{m}$

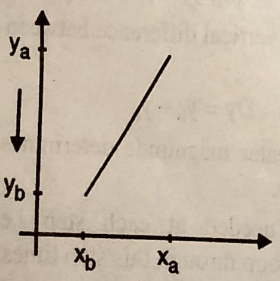
$y_{increment} = 1$

Fig. 2.3.12

9) Similarly if |Dx| < |Dy| and $y_a > y_b$

Then the values of increments in x and y-direction are $-\frac{1}{m}$ and – 1.

Refer Fig. 2.3.13.



$x_{increment} = -\frac{1}{m}$

$y_{increment} = -1$

Fig. 2.3.13

Program for DDA line generation

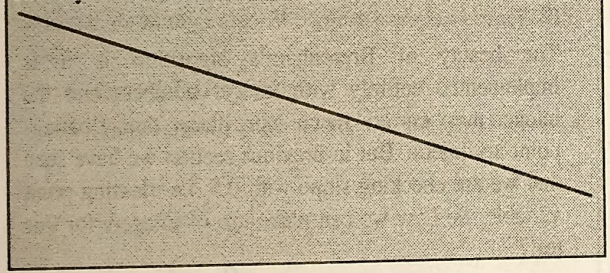
```
/*****************************************
Program :
A C program to draw a line by using DDA linegeneration
algorithm
**********************/
#include<iostream.h>
#include<conio.h>
#include<math.h>
#include<stdlib.h>
#include<dos.h>
#include<graphics.h>
void dda(int x1,int y1,int x2,int y2);
/*=========================
Function Name :  Main
Purpose       :  To initialize graphics mode and call DDA
function
============*/
void main()
{
    int gd = DETECT, gm;  /*detect the graphics drivers
automatically*/

    int x1,y1,x2,y2;
 init graph(&gd,&gm,"c:\\tcplus\\bgi");  /*Initialise to graphics
mode*/
  cleardevice();
  cout<< " DDA Line generation algorithm";
  cout<< " \n enter the starting co-ordinates for drawing
line";
  cin>>x1>>y1;
  cout<<" \n enter the ending co-ordinates";
  cin>>x2>>y2;
  dda(x1,y1,x2,y2);
  cout<< "\n Thank you";
  getch();
  closegraph();
}
/*=========================
Function Name :  dda
Purpose :  To draw a line using DDA algorithm
==========*/
 void dda(int x1,int y1,int x2,int y2)
{
    int i,dx,dy,steps;
    float x,y;
    float xinc,yinc;
    dx = (x2-x1);
    dy = (y2-y1);
if(abs(dx)>=abs(dy)) /* decide whether to move along x-
direction */
```

```
    steps = dx;       /* or y-direction */
    else
        steps = dy;
xinc =(float) dx/steps; /*calculate the increment value for x &
y*/
    yinc =(float) dy/steps;
    x = x1;
    y = y1;
putpixel (x,y,WHITE);      /* plot first point */
    for (i=1;i<steps;i++)
        {
            x = x + xinc;
            y = y + yinc;
/* convert the floating value to its nearest interger value.i.e.
same as use of floor or ceil function */
            x1 = x + 0.5 ;
            y1 = y + 0.5 ;
    putpixel (x1,y1,WHITE);  /* plot the points */
        }
}
```

Output

```
DDA Line generation algorithm
Enter the starting co-ordinates for drawing line
200
300
enter the ending co-ordinates
450
400
Thank you
```



☞ Advantages of DDA

- The DDA algorithm is a faster method for calculating pixel positions than the direct use of line equation $y = mx + b$.
- It is very easy to understand.
- It requires no special skills for implementation.

☞ Disadvantages of DDA

- Because of round off, errors are introduced and causes the calculated pixel position to drift away from the true line path.
- Because of floating point operations the algorithm is time-consuming.

☞ Example

Suppose if we want to draw a line from (1, 1) to (5, 3).

So, $x_1 = 1$, $y_1 = 1$, $x_2 = 5$ and $y_2 = 3$.

$\therefore Dx = (x_2 - x_1) = (5 - 1) = 4$

$Dy = (y_2 - y_1) = (3 - 1) = 2$

As $|Dx| > |Dy|$ the line is of gentle slope category.

$\therefore$ Steps = abs (Dx) = 4

$$x_{increment} = \frac{Dx}{steps} = \frac{4}{4} = 1$$

$$y_{increment} = \frac{Dy}{steps} = \frac{2}{4} = 0.5$$

First point we know i.e. x_1, y_1 so plot it.

$\therefore x_{new} = x_{old} + x_{increment} = 1 + 1 = 2$

$y_{new} = y_{old} + y_{increment} = 1 + 0.5 = 1.5$

But over here we have to round off 1.5 as 2 for displaying that point.

For next iteration

$x_{new} = x_{old} + x_{increment} = 2 + 1 = 3$

$y_{new} = y_{old} + y_{increment} = 1.5 + 0.5 = 2$

Now loop steps are tabulated as follows :

I	x	y	Plot
1	1	1	1, 1
2	2	1.5 ≈ 2	2, 2
3	3	2	3, 2
4	4	2.5 ≈ 3	4, 3
5	5	3	5, 3

- For calculation purpose we are using original value but for display only, we are rounding the values.
- If we do this till x_{new} and y_{new} becomes same as end point then we will get line same as shown in Fig. 2.3.14.

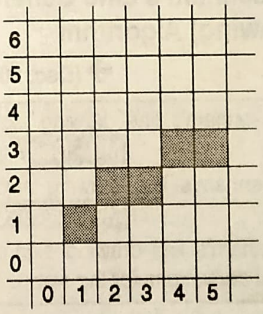


Fig. 2.3.14

Example 2.3.1

Explain DDA line drawing algorithm. Consider the line from (1,1) to (5,6). Use DDA line drawing algorithm to rasterize this line.

Solution :

Given : $x_1 = 1,\ Y_1 = 1$

$x_2 = 5,\ Y_2 = 6$

$\therefore$ Dx = $x_2 - x_1 = 5 - 1 = 4$

Dy = $y_2 - y_1 = 6 - 1 = 5$

Since |Dy| > |Dx| the line is of steep slope category

$\therefore$ Steps = abs (Dy) = 5

$$x_{increment} = \frac{Dx}{Steps} = \frac{4}{5} = 0.8$$

$$y_{increment} = \frac{Dy}{Steps} = \frac{5}{5} = 1$$

As we know 1st point, let's plot it as (1, 1)

$\therefore$ $x_{new} = x_{old} + x_{increment} = 1 + 0.8 = 1.8$

$y_{new} = y_{old} + y_{increment} = 1 + 1 = 2$

But here we need to round off 1.8 as 2 for displaying that point

$\therefore$ for next iteration

$x_{new} = x_{old} + x_{increment} = 1.8 + 0.8 = 2.6$

$y_{new} = y_{old} + y_{increment} = 2 + 1 = 3$

Like this we will calculate points till y_{new} becomes equal to y_2 i.e. 6

The following table summarizes the points.

X	Y	Plot
1	1	1, 1
1.8	2	2, 2
2.6	3	3, 3
3.4	4	3, 4
4.2	5	4, 5
5	6	5, 6

➜ 2.3.2 Bresenham's Line Generation (Drawing) Algorithm

➜ **(Dec. 2014, Dec. 2015)**

Q. Derive Bressenham's line drawing algorithm for lines with slope. **MU - Dec. 2014, 10 Marks**

Q. Derive Bresenham's line drawing algorithm for lines with slop < 1. **MU - Dec. 2015, 5 Marks**

Q. Write Bresenham's line drawing algorithm. Also write mathematical derivations for the same. **10 Marks**

- There is one more algorithm to generate lines which is called as Bresenham's line algorithm. The distance between the actual line and the nearest grid location is called error and it is denoted by G.
- Suppose line is as shown in Fig. 2.3.15. After displaying first point (0, 0) we have to select next point. Now there are two candidate pixels (1, 0) and (1, 1). Out of these two pixels we have to select one pixel.

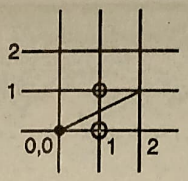


Fig. 2.3.15

- This selection of pixel will depend on the slope of the line. If the slope of line is greater than 0.5 then, we are selecting upper pixel i.e. (1, 1) and if slope is less than 0.5 then we are selecting next pixel as (1, 0).
- As we are interested in only checking whether the point is below 0.5 or above 0.5, we will put condition as if the slope of line is greater than or less than 0.5.

☞ Example

- Suppose the slope of line shown in Fig. 2.3.16 is 0.4. After displaying 1st point as (0, 0) we will add slope of line to error factor G, so that G will become 0.4.
- As we are considering gentle slope case we are moving along x-axis so next pixel will be x = 1 and at that time y will be decided by G. If G > 0.5 then y = 1 else y = 0. But here as G = 0.4 so we have to select next point as (1, 0).

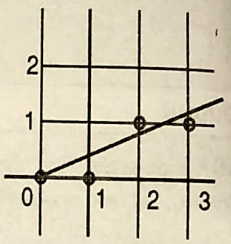


Fig.2.3.16

- Again for next point x is increased by 1 and it becomes x = 2, for that G will become G = G + m i.e. 0.8. Now this time G > 0.5 so upper pixel is more near to desired one. That is why we have to select upper point i.e. (2, 1).
- The beauty of Bresenham's algorithm is it is implemented entirely with integer numbers and the integer numbers are much more faster than floating-point arithmetic. But in previous section we have seen that we are checking slope with 0.5. i.e. floating point variable. So how we can make use of integers for this test?

☞ For Gentle slope case

- Consider P as height of pixel or error. Our condition is whether P > 0.5 or not.

If $P > 0.5$ ...(2.3.3)

- But here 0.5 is floating point so we have to convert floating point to integer. For that we will multiply both sides by 2.

$\therefore$ $2P - 1 > 0$ ...(2.3.4)

- In this equation we have removed fractional part 0.5. But still it may contain fractional part from the denominator of slope. Because every time we are updating P by,

$P = P + slope$ or $P = P + slope - 1$

i.e. here we are adding slope to P

But slope is nothing but $\frac{Dy}{Dx}$.

- So the fractional part may come due to Dx. To eliminate this we will multiply the Equation (2.3.4) by Dx.

$\therefore 2PDx - Dx > 0$

Now we will define $G = 2PDx - Dx$ ...(2.3.5)

$\therefore$ Our test will become as $G > 0$

- Now we will see how this G can be used to decide which row or column to select.

As $G = 2PDx - Dx$

$\therefore G + Dx = 2PDx$

$\therefore \frac{G + Dx}{2Dx} = P$

Now if we consider P = P + slope, then

We will get $\frac{G + Dx}{2Dx} = \frac{G + Dx}{2Dx} + \frac{Dy}{Dx}$

If we further solve this equation then we will get

$G = G + 2Dy$ ...(2.3.6)

- Similarly if we consider P=P + slope – 1, then

We will get

$G = G + 2Dy - 2Dx$ ...(2.3.7)

- To calculate this G we need only addition and subtraction and no multiplication or division. Another thing is now, this G is not containing any fractional part also.
- So, we can use test $G > 0$ to determine when a row boundary is crossed by a line instead of $P > 0.5$.
- As the initial value of P is slope, so we have to find initial value of G also. We know from Equation (2.3.5) that

$G = 2DxP - Dx$

- Put initial value of P as Dy/Dx to find initial value of G.

$\therefore G = 2Dx\left(\frac{Dy}{Dx}\right) - Dx$

$\therefore G = 2Dy - Dx$

- For each column we check G. If it is positive we move to the next row and add(2 Dy – 2 Dx) to G, because it is equivalent to P = P + slope – 1, otherwise we will keep the same row and add (2 Dy) to G.

☞ Steps for Bresenham's line drawing algorithm for Gentle slope (m < 1)

1) Accept two endpoints from user and store the left endpoint in (x_0, y_0) as starting point.

2) Plot the point (x_0, y_0)

3) Calculate all constants from two endpoints such as Dx, Dy, 2Dy, 2Dy – 2Dx and find the starting value for the G as G = 2 Dy – Dx.

4) For each column increment x and decide y-value by checking G > 0 condition. If it is true then increment y-value and add (2Dy – 2Dx) to current value of G otherwise add (2Dy) to G and don't increment y-value. Plot next point. See Fig. 2.3.17.

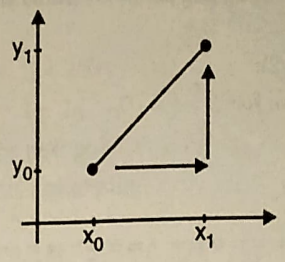


Fig. 2.3.17

5) Repeat step 4 till Dx times.

For steep slope cases just interchange the rolls of x and y i.e. we step along y-direction in unit steps and calculate successive x-values nearest the line path.

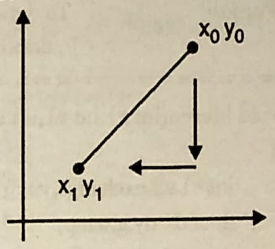


Fig. 2.3.18

If the initial position for a line with positive slope is right endpoint, as shown in Fig. 2.3.18, then we have to decrease both x and y as we step from right to left.

☞ Program for Bresenham's line generation

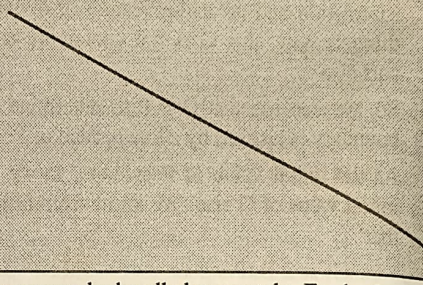
```
/*****************************************
Program :
A C program to draw a line by using Bresenham's line
generation algorithm
**********************/
#include<iostream.h>
#include<conio.h>
#include<math.h>
#include<stdlib.h>
#include<dos.h>
#include<graphics.h>
void bresen(int x1,int y1,int x2, int y2);
int sign (float arg);
/*============================
Function Name   : Main

Purpose         : To initialize graphics mode and call
                  Bresenham function
============================*/
void main()
{
int gd = DETECT, gm;
int x1,y1,x2,y2;
init graph (&gd,&gm,"c:\\tcplus\\bgi");
```

```
cleardevice();
cout<< " Bresenhams line drawing ";
cout<< "\n enter the starting point's co-ordinates for line ";
cin>>x1>>y1;
cout<<"\n enter the ending points co-ordinates";
cin>>x2>>y2;
bresen(x1,y1,x2,y2);
cout<< "\n Thank you";
getch();
closegraph();
}
/*=========================
Function Name  : bresen
Purpose        : To draw a line using bresenham's line
                 drawing algorithm
=========================*/
void bresen(int x1,int y1,int x2,int y2)
{
   int s1,s2,exchange,y,x,i;
   float dx,dy,g,temp;
   dx = abs(x2 - x1);
   dy = abs(y2 - y1);
   x = x1;
   y = y1;
   s1 = sign(x2-x1);
   s2 = sign(y2-y1); /* interchange dx & dy depending on
he slope of the line */
   if(dy>dx)
   {
      temp = dx;
      dx = dy;
      dy = temp;
      exchange = 1;
   }
  else
      exchange = 0;
 g = 2 * dy - dx;
 i = 1;
 while(i<=dx)
 {
     putpixel(x,y,WHITE);
     delay(10);
     while(g>=0)
     {
        if(exchange ==1)
              x = x + s1;
        else
              y = y + s2;
              g = g + 2dy - 2dx;
     }
        if(exchange==1)
            y = y + s2;
        else
            x = x + s1;
            g = g + 2 * dy;
            i++;
     }
}
int sign(float arg)
    {
    if (arg<0)
        return -1;
        else if(arg==0)
            return 0;
                else return 1;
    }
```

Output

```
Bresenhams line drawing,
Enter the starting point's co-ordinates for line drawing.
200
200
enter the ending points co-ordinates
250
450
Thank you
```



- Special cases can be handled separately. For horizonta lines we should not increase y. We have to jus increase/decrease x by 1 every time till endpoint.
- For vertical lines we should not increase x we have t just increase/decrease y by 1 every time till endpoint.
- Similarly for diagonal lines we will increase/decreas both x and y by 1 every time till endpoint.

☞ Example

- Plot a line by using Bresenham's line generation algorithm from (1, 1) to (5, 3). See Fig. 2.3.19.

Given :

$x_1 = 1, y_1 = 1$

$x_2 = 5, y_2 = 3$

$\therefore Dx = x_2 - x_1 = 4$

$Dy = y_2 - y_1 = 2$

$G = 2\,Dy - Dx$

$= 2\,(2) - (4)$

$G = 0$

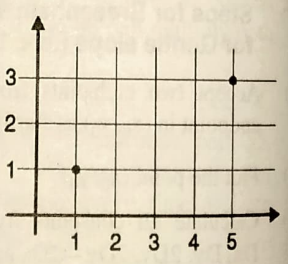


Fig. 2.3.19

- Plot 1st point (1, 1) here as |Dx| > |Dy| that means line is Gentle slope, so here we have to move on x till x_2 i..e. 5.

  After ploting 1st point as (1, 1) increase x by 1

  $\therefore x = 2$

  Here G = 0

  We have to increase y by 1 and update G as,

  $G = G + 2(Dy - Dx)$

  $= 0 + 2(2 - 4) = -4$

- So, plot next point as (2, 2) then again increase x by 1. Now it will become x = 3.

  Here G = – 4.

  So, don't increase y just update G only as

  $G = G + 2\,Dy$

  $= -4 + 2(2) = 0$

  So, plot (3, 2) go on doing this till x reaches to x_2.

  We will get points as,

X	Y
1	1
2	2
3	2
4	3
5	3

- So, the final points on line will be as shown in Fig. 2.3.20.

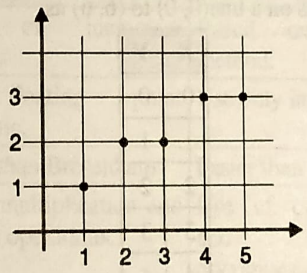


Fig. 2.3.20

- Bresenham algorithm is much more efficient than DDA, it requires only integers and requires only addition and subtraction operations.

Example 2.3.2

Consider the line from (2, 7) to (5, 5). Use Bresenham's line drawing algorithm to rasterize this line.

Solution :

Given : $x_1 = 2,\ y_1 = 7$

$x_2 = 5,\ y_2 = 5$

$\therefore Dx = |x_2 - x_1| = |5 - 2| = (3)$

$Dy = |y_2 - y_1| = |5 - 7| = (2)$

$G = 2Dy - Dx = 2(2) - 3 = 1$

Plot 1st point (2, 7)

Since |Dx| > |Dy| the line is having gentle slope.

So we have to move along x till x_2 i.e. 5; after plotting 1st point as (2,7), increase x by 1.

$\therefore x = 3$

Here G > 0

$\therefore$ We have to decrease y by 1 because $y_1 > y_2$ and update G as

$G = G + 2(Dy - Dx)$

$= 1 + 2(2 - 3) = 1 + 2(-1) = 1 - 2 = -1$

$\therefore$ Plot the next point as (3,6)

Now again increasing x by 1, x becomes 4; here G = – 1

$\therefore$ Don't modify y and update G as

$G = G + 2Dy = -1 + 2(2) = 3$

So, plot next point as (4,6). Go on doing this till x becomes x_2.

$\therefore$ Here the points we are getting will be

X	Y
2	7
3	6
4	6
5	5

Example 2.3.3

Consider the line from (4, 9) to (7, 7).

Draw a line using Bresenham's line drawing algorithm.

Solution :

Given : $x_1 = 4,\ y_1 = 9$

$x_2 = 7,\ y_2 = 7$

$\therefore Dx = |x_2 - x_1| = |7 - 4| = 3$

$Dy = |y_2 - y_1| = |7 - 9| = 2$

$G = 2Dy - Dx = 2(2) - 3 = 4 - 3 = 1$

Let's plot 1st point as (4, 9)

Since |Dx| > |Dy| the line is of Gentle slope category, so we have to move along x till x_2 reaches.

After potting 1st point as (4, 9) increase x by 1

$\therefore x = 5$

Here G > 0

So update y by – 1 because $y_1 > y_2$ and modify G as

$G = G + 2(Dy - Dx)$

$= 1 + 2(2 - 3) = 1 + 2(-1)$

$= -1$

$\therefore$ Plot next point as (5, 8)

Now again increasing x by 1, x becomes 6. Here G = – 1

$\therefore$ Don't modify y and update G as

$G = G + 2Dy = -1 + 2(2)$

$= -1 + 4 = +3$

$\therefore$ Plot next point as (6, 8)

Now again increase x by 1, x becomes 7, Here G = 3

i.e. G > 0

∴ update y by – 1 because y1 > y2 and modify G as

$$G = G + 2(Dy - Dx)$$
$$= 3 + 2(2 - 3) = 3 + 2(-1) = 1$$

∴ Plot next point as (7, 7) which is our another end point

Hence we get following points on line (4, 9) to (7, 9) as

X	Y
4	9
5	8
6	8
7	7

Example 2.3.4

Using Bresenham's line algorithm,find out which pixel would be turned on for the line with end points (4, 4) to (12, 9).

Solution :

Given : $x_1 = 4,\ y_1 = 4$

$x_2 = 12,\ y_2 = 9$

$\therefore Dx = |x_2 - x_1| = |12 - 4| = 8$

$Dy = |y_2 - y_1| = |9 - 4| = 5$

$G = 2Dy - Dx = 2(5) - 8 = 10 - 8 = 2$

Let's plot 1st point as (4,4)

Since |Dx| > |Dy| the line is of Gentle slope category

So we have to move along x till x_2 i.e. 12

After plotting 1st point as (4, 4), increase x by 1.

∴ x becomes equal to 5

Here G > 0

So update y by 1 and modify G as

$$G = G + 2(Dy - Dx) = 2 + 2(5 - 8) = 2 + 2(-3)$$
$$= 2 - 6$$
$$= -4$$

∴ Plot next point as (5, 5)

Now again increasing x by 1, x becomes 6, Here G = – 4

∴ Don't modify y and update G as

$$G = G + 2Dy = -4 + 2(5) = -4 + 10 = 6$$

Plot next point as (6, 5). Go on doing this till x becomes x_2.

Hence we get following points on a line (4, 4) to (12, 9)

X	Y
4	4
5	5
6	5
7	6
8	6
9	7
10	7
11	8
12	9

Example 2.3.5

Consider the line from (0,0) to (6,6) Bresenham's algorithm to rasterize this line.

Solution : Refer Section 2.3.2.

Given : $x_1 = 0$ $y_1 = 0$

$x_2 = 6$ $y_2 = 6$

$\therefore Dx = |x_2 - x_1| = |6 - 0| = 6$

$Dy = |y_2 - y_1| = |6 - 0| = 6$

$G = 2Dy - Dx = 2(6) - 6 = 12 - 6 = 6$

Lets plot 1st point as (0, 0)

Since Dx = Dy the line is in gentle slope category. The line is having slope as

$$\frac{Dy}{Dx} = \frac{6}{6} = 1$$

It means we have to move along x direction from x_1 to x_2 and find the corresponding y value.

After plotting 1st point as (0, 0), increase x by 1.

∴ x becomes 1

Here G > 0, so update y by 1 and modify G as

$$G = G + 2(Dy - Dx) = 6 + 2(6 - 6) = 2$$

∴ Plot next point as (1, 1)

Now again increasing x by 1, x becomes 2, Here G = 6

∴ Since G > 0, so update y by 1 and modify G as

$$G = G + 2(Dy - Dx) = 6 + 2(6 - 6) = 6$$

∴ Plot next point as (2, 2)

Go on doing this till x becomes x_2. Hence we get following points on a line(0, 0) to (6, 6) as

X	Y
0	0
1	1
2	2
3	3
4	4
5	5
6	6

Example 2.3.6

Find out which pixel would be turned on for the line with end points (2, 2) to (6, 5) using the Bresenham line algorithm.

Solution : Refer Section 2.3.2.

Given : $x_1 = 2,$ $y_1 = 2$

$x_2 = 6,$ $y_2 = 5$

$\therefore Dx = x_2 - x_1 = 4$ $Dy = y_2 - y_1 = 3$

$G = 2Dy - Dx = 2(3) - (4)$

$G = 2$

Plot 1st point (2, 2) here as |Dx| > |Dy| that means line is Gentle slope, so we have to moveon x till x_2 i..e. 6.

After plotting 1st point as (2, 2) increase x by 1

$\therefore x = 3$ Here G= 2

We have to increase y by 1 and update G as,

$G = G + 2(Dy - Dx) = 2 + 2(3 - 4) = 0$

So, plot next point as (3, 3) then again increase x by 1. Now it will become x = 4.

Here G = 0.

We have to increase y by 1 and update G as,

$G = G + 2(Dy - Dx) = 0 + 2(3 - 4) = -2$

So, plot next point as (4, 4) then again increase x by 1. Now it will become x = 5.

Here G = −2.

So, don't increase y just update G only as

$G = G + 2Dy = -2 + 2(2) = 2$

So, plot (5, 4) go on doing this till x reaches to x_2.

We will get points as,

X	Y
2	2
3	3
4	4
5	4
6	5

2.3.3 Comparison of DDA and Bresenham's Line Drawing Algorithm

Sr. No.	DDA	Bresenham
1.	Based on increment method.	Based on increment method.
2.	Use floating point arithmetic.	Use only integers.
3.	Slower than Bresenham	Faster than DDA
4.	Use of multiplication and division operations.	Use of only Addition and Subtraction operations.
5.	To display pixel we need to use either floor or ceil function.	No need of floor or ceil function for display.
6.	Because of floor and ceil function error component is introduced.	No error component is introduced.
7.	The co-ordinate location is same as that of Bresenham.	The co-ordinate location is same as that of DDA.

2.4 Thick Line Generation

→ (May 2013)

Q. Explain the method to draw a thick line using Bresenham's algorithm. **MU - May 2013, 5 Marks**

- We can produce thick lines also, whose thickness is greater than one pixel. To produce a thick line segment, we can run two line drawing algorithms in parallel to find the pixels along the line edges.
- As we are moving to next pixel of line 1 and line 2, we must also turn on all the pixels which lie between the boundaries. See Fig. 2.4.1.

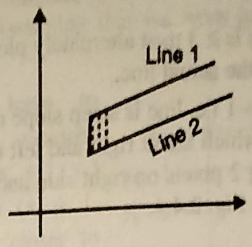


Fig. 2.4.1

- To be more specific, if we want to draw a gentle slope line from (x_a, y_a) to(x_b, y_b) with thickness w, as shown in Fig. 2.4.2., then the corner points of thick line become.

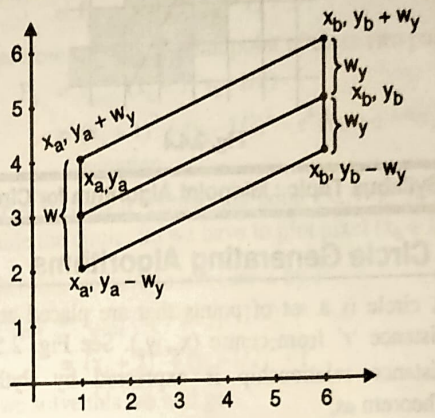


Fig. 2.4.2

$(x_a, y_a + w_y)$, $(x_b, y_b + w_y)$, $(x_a, y_a - w_y)$, $(x_b, y_b - w_y)$

Where w_y can be calculated as,

$$w_y = \frac{w-1}{2}\left(\frac{((x_b - x_a)^2 + (y_b - y_a)^2)^{1/2}}{|x_b - x_a|}\right)$$

- w_y is the amount by which the boundary line is moved from the center.
- (w − 1) factor is the desired width minus one pixel thickness, we automatically receive from drawing the boundary. We divide this by 2 because half the thickness will be used to offset the top boundary and other half to move the bottom boundary. The factor containing x and y-values is needed to find the amount to shift up and down in order to achieve proper width w.

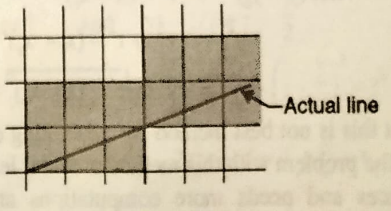


Fig. 2.4.3

- The above mentioned is one of the general method. For special thing we can have other alternatives also. Such as if thickness required is 2 and slope of line is < 1 i.e. line in gentle slope category. See Fig. 2.4.3. Then we can simply plot pixel or draw another line next to actual line.
- If thickness is ≥ 3 then alternately plot the pixels above and below the actual line.
- If slope is > 1 i.e. line is steep slope category then pick the pixels which are at right and left of the line. i.e. we are plotting 2 pixels on right side and one pixel on left of line. See Fig. 2.4.4.

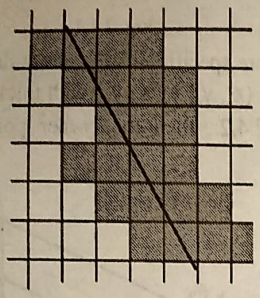


Fig. 2.4.4

Syllabus Topic : Midpoint Algorithm for Circle

2.5 Circle Generating Algorithms

- A circle is a set of points that are placed at a given distance 'r' from centre (x_c, y_c). See Fig. 2.5.1. This distance relationship is expressed by Pythagorean Theorem as,

$$(x - x_c)^2 + (y - y_c)^2 = r^2$$

- We could use this equation to calculate the position of points on circumference of circle by stepping along x-axis from $(x_c - r)$ to $(x_c + r)$ and calculating the corresponding y-values at each position by using,

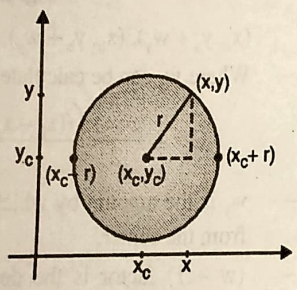


Fig. 2.5.1

$$y = y_c \pm \sqrt{r^2 - (x - x_c)^2}$$

- We derived this equation from the equation of circle only.

$$\text{i.e. } (y - y_c)^2 = r^2 - (x - x_c)^2$$

$$y^2 = y_c^2 + r^2 - (x - x_c)^2$$

$$y = y_c \pm \sqrt{r^2 - (x - x_c)^2}$$

- But this is not best method for generating circle as one of the problem with this method is that it leaves uneven spaces and needs more computations at each step. See Fig. 2.5.2.
- We can minimize this problem of spacing by interchanging the position of xand y.
- But there is another way to solve this spacing problem. And that is by using polar co-ordinates r and θ. We use *parametric polar form* to yield the pair of equation as,

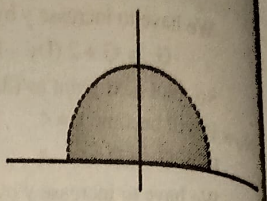


Fig. 2.5.2

$$x = x_c + r\cos\theta$$

$$y = y_c + r\sin\theta$$

- With this we are displaying points at fixed angular step size. The step size chosen for θ will depend on application. See Fig. 2.5.3.
- We can reduce the computations by considering the symmetry of circles. By using this symmetry property there is no need to find each and every point on the boundary of the circle

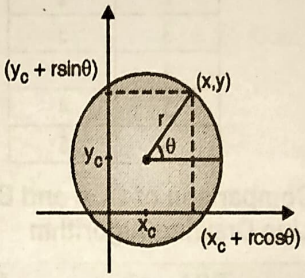


Fig. 2.5.3

- We can divide the circle in 4 quadrants or in 8 octants also. Here we will find only one octant's co-ordinates and then we will just replicate that one octant's co-ordinates to rest seven octants.
- See Fig. 2.5.4. Suppose we have calculated a point (x, y) then from this (x, y) we will plot rest seven points as (x, y), (y, x), (– x, y),(– y, x), (x, – y), (y, – x),(– x, – y), (– y, – x). So using this symmetry proper circle generation becomes fast.

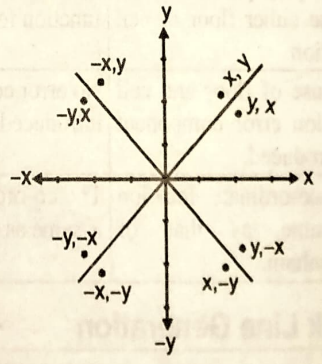


Fig. 2.5.4

- With Cartesian equation of Pythagorean we have to perform multiplication and square root operations, while in parametric equation we have to perform multiplications and trigonometric calculations.

- Both these methods require more computational time. So we have another method of *Bresenham's* which totally deals with integers and performs only addition and subtraction, so it is fast also. This method is applicable for different types of curves also. There is one more approach which is called as *midpoint circle generation*. This approach generates the same pixel positions as the Bresenham's circle algorithm. So first we will see midpoint circle generation algorithm.

2.5.1 Midpoint Circle Generation Algorithm

➔ **(May 2014, Dec. 2014, May 2015, Dec. 2015, May 2016, Dec. 2016)**

Q. Specify midpoint circle algorithm.
MU - May 2014, 5 Marks

Q. Explain midpoint circle algorithm. Use the same to plot the circle, whose radius is 10 units.
MU - Dec. 2014, Dec. 2015, May 2016, Dec. 2016, 10 Marks

Q. Explain the midpoint circle generating algorithm.
MU - May 2015, 8 Marks

Q. Explain mid-point circle algorithm. In order to support your explanation, show mathematical derivation.
(8 Marks)

Q. Derive Mid point circle algorithm. **(10 Marks)**

- Here we have to determine the closest pixel position to the specified circle's path at each step. For this we have to accept radius 'r' and center point x_c and y_c. See Fig. 2.5.5. We will first see for center point as origin (0, 0).
- Then each calculated point is to be moved to proper position by adding x_c and y_c to corresponding x and y-value. In first quadrant we are moving in x-direction with starting point as x = 0 to ending point x = y.
- To apply a midpoint method, we define a circle function.

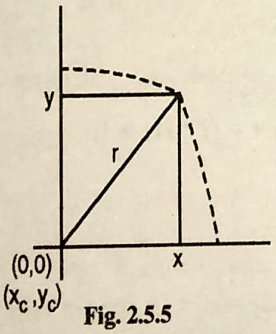


Fig. 2.5.5

$f_{circle}(x, y) = x^2 + y^2 - r^2$ which is same as $r^2 = x^2 + y^2$

Which is derived from equation of circle,

$(x - x_c)^2 + (y - y_c)^2 = r^2$. But here we are considering x_c, y_c as (0, 0) $\therefore x^2 + y^2 = r^2$.

- Any point (x, y) on the boundary of the circle with radius r satisfies the equation $f_{circle}(x, y) = 0$. If point is inside the circle, then circle function is negative and if point is outside circle then circle function is positive.

i.e. if

$f_{circle}(x, y) < 0$, then x, y is inside the circle boundary
$= 0$, then x, y is on circle boundary
> 0, then x, y is outside the circle boundary.

- Thus circle function is the decision parameter in this algorithm. Assuming that we have just plotted x_k, y_k point.
- Now we have to determine whether next point $(x_k + 1, y_k)$ is closer or $(x_k + 1, y_k - 1)$ is closer to actual circle. See Fig. 2.5.6.

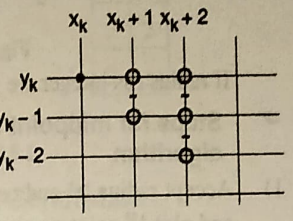


Fig. 2.5.6

- Our decision parameter (P_k) is circle function.

$\therefore P_k = f_{circle}(x_k + 1, y_k)$ or $P_k = f_{circle}(x_k + 1, y_k - 1)$

- Therefore we will take midpoint of these two points as,

$P_k = f_{circle}(x_k + 1, y_k - 1/2)$
$= (x_k + 1)^2 + (y_k - 1/2)^2 - r^2$ from circle equation

- If $P_k < 0$, then it means this point $(x_k + 1, y_k - 1/2)$ is inside the circle. So we have to plot pixel $(x_k + 1, y_k)$.

After this we have to increase x by 1.

i.e. $P_{k\,new} = f_{circle}(x_k + 2, y_k - 1/2)$
$= (x_k + 2)^2 + (y_k - 1/2)^2 - r^2$

if we solve this we will get

$P_{k\,new} = P_k + (2x_k + 1)$

i.e. when, earlier we have not changed row.

- If $P_k > 0$ then we will select $y_k - 1$ for displaying. Like this we will select the point to be displayed.

$P_{knew} = f_{circle}(x_k + 2, y_k - 3/2)$
$= (x_k + 2)^2 + (y_k - 3/2)^2 - r^2$

After solving above equation we will get,

$P_{knew} = P_k + (2x_k - 2y_k + 1)$

i.e. when, earlier we have changed row.

- Similarly the initial decision parameter is obtained by evaluating the circle function at start position $(x_0, y_0) = (0, r)$. See Fig. 2.5.7.

Now calculate the decision parameter.

$P_0 = f_{circle}(x_k + 1, y_k - 1/2)$

But $x_k = 0$ and $y_k = r$

$\therefore x_k + 1 = 1$ and $y_k - 1/2 = r - \frac{1}{2}$

$$P_0 = f_{circle}\left(1, r - \frac{1}{2}\right) = 1^2 + \left(r - \frac{1}{2}\right)^2 - r^2$$

$$= 1 + r^2 + \frac{1}{4} - 2\frac{1}{2}r - r^2 = 1 + \frac{1}{4} - r$$

$$P_0 = \frac{5}{4} - r$$

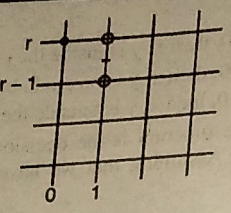


Fig. 2.5.7

If radius r is integer we will round it as $P_0 = 1 - r$.

Steps for midpoint circle generation algorithm

1) Accept radius 'r' and center of circle (x_c, y_c) from user and plot 1st point on circumference of circle.

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

2) Calculate the initial value of the decision parameter.

$$P_0 = 1 - r$$

3) If we are using octant symmetry property to plot the pixels, then until (x < y) we have to perform following steps.

If P_k is less than 0

modify P_k as $P_k = P_k + 2x + 1$ and

then increase x by 1

otherwise

modify P_k as $P_k = P_k + 2(x - y) + 1$ and

increase x by 1

and decrease y by 1

4) Determine the symmetry points in other octants also

5) Since we have derived all formulas by considering center point as origin. Move each calculated pixel position (x, y) on to the circular path centered on (x_c, y_c) and plot the co-ordinate values as

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

$$y = y + y_c$$

But actually the center point may be anything, so we have to add that center co-ordinates to midpoint x and y to plot the point.

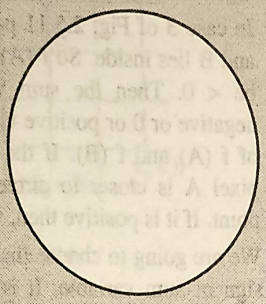
Midpoint Circle generation program

```
/*********************************************
Program :
A C program to draw a circle by using midpoint circle
generation algorithm
*********************************************/
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
#include<graphics.h>
#include<dos.h>
void circ_midpt(int x,int y,int rad);
void display(int,int,int,int);
/*=============================================
Function Name  : Main
Purpose        : To initialize graphics mode and
                 call mid point circle function
=============================================*/
void main()
{
 int gd = DETECT, gm, x,y,r;
 initgraph(&gd,&gm,"c:\\tcplus\\bgi");
 cleardevice();
 cout<<" Midpoint circle generation algorithm ";
 cout<<"\n enter the center co-ordinates for the circle";
 cin>>x>>y;
 cout<<"\n enter the radius of the circle";
cin>>r;
 circ_midpt(x,y,r);
getch();
 closegraph();
}
/*=============================================
Function Name  : circ_midpt
Purpose        : To draw a circle by mid point circle
                 generation algorithm
=============================================*/
void circ_midpt(int x,int y,int rad)
{
    int x1,y1;
float dp; //initialising the descision parameter.
    x1 = 0;   //initialising the X,Y cordinates.
    y1 = rad;
    dp = 1 - rad;
    while(x1<=y1)
    {
       if(dp<=0)
       {
          x1++;
          dp += (2 * x1) + 1;
       }
       else
       {
          x1++;
          y1--;
          dp += 2*(x1-y1)+1;
       }
       display(x1,y1,x,y);
    }
}
/*=============================================
Function  : Name:  display
```

```
Purpose     : To plot the pixels on circle by symmetry
              property.
==========================================*/
void display (int x1,int y1,int x,int y)
{
putpixel(x1+x,y1+y,WHITE);
                    //plotting the pixels by using symmetry.
  putpixel(x1+x,y-y1,WHITE);
  putpixel(x-x1,y1+y,WHITE);
  putpixel(x-x1,y-y1,WHITE);
  putpixel(x+y1,y+x1,WHITE);
  putpixel(x+y1,y-x1,WHITE);
  putpixel(x-y1,y+x1,WHITE);
  putpixel(x-y1,y-x1,WHITE);
}
```

Output

```
Midpoint circle generation algorithm
enter the center co-ordinates for the circle
320
240
enter the radius of the circle
150
```



Example 2.5.1

Plot a circle whose radius is 3 and whose center co-ordinates are (0, 0).

Solution :

Given : $r = 3, x_c = 0, y_c = 0$

Plot 1st point as (0, r) i.e. $y = r$

$\therefore$ Plot (0, 3)

Like that find out other points from (0, 3) by using symmetry property

So plot (3, 0)

plot (0, – 3)

plot (– 3, 0)

find $P = 1 - r$ $\therefore$ $P = 1 - 3 = -2$

Till $(x < y)$ we have to perform following

If $(P < 0)$

In this case $P = -2$

So we have to increase x by 1 and

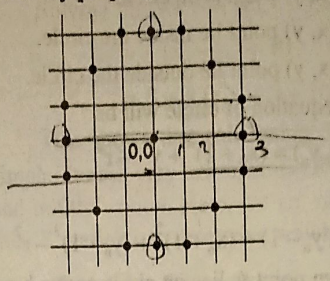
modify P as $P = P + 2x + 1$

i.e. $P = -2 + 2(0) + 1 = -2 + 1 = -1$

Here we are not increasing y.

$\therefore$ Plot $(x + 1, y)$ i.e. (1, 3)

Again continuing this process we will get following table. Then plot the co-ordinates of this table by using symmetry property as shown in Fig. P. 2.5.1.



x	y
0	3
1	3
2	2
3	1

Fig. P. 2.5.1

2.5.2 Bresenham Circle Generation

➔ **(May 2017)**

Q. Explain Bresenham's Circle drawing algorithm in detail.
MU - May 2017, 10 Marks

- This algorithm does only integer arithmetic which makes it faster than floating point. We are creating only one octant of the circle i.e. from 90° to 45°.
- See Fig. 2.5.9. We derive other seven octants of the circle by symmetry property. The circle is generated by considering center point as origin with radius 'r'. The algorithm calculates one new pixel per step. From any point (x_n, y_n) on the circle, the next point $(x_n + 1, y_n + 1)$ must be either one on right or one to the right and down.

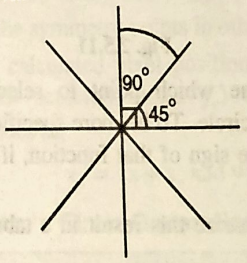


Fig. 2.5.9

- If P_n is a current point with co-ordinate (x_n, y_n) then from Fig. 2.5.10 we will come to know that next point could be either A or B.
- Point A will be having co-ordinates $(x_n + 1, y_n)$ and B will be $(x_n + 1, y_n - 1)$ so for selecting one of these two candidate pixels we have to perform some test.
- To do this, we know that a function of a circle is $f_{circle}(x, y) = x^2 + y^2 - r^2$.

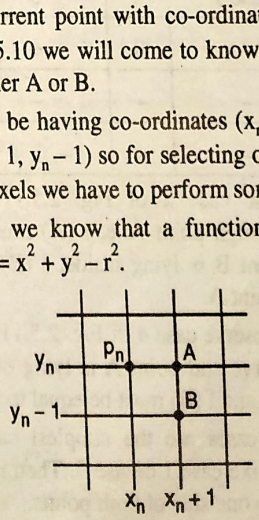


Fig. 2.5.10

- This function determines a circle with radius 'r' around the origin in the following way :

  If $f(x, y) = 0$ then (x, y) point is on circle.

  If $f(x, y) < 0$ then (x, y) point lie inside the circle.

  If $f(x, y) > 0$ then (x, y) point lie outside the circle.

  i.e. for point A the equation of circle will be

  $$f(A) = f(x_n + 1, y_n) = (x_n + 1)^2 + y_n^2 - r^2$$

  and for point B

  $$f(B) = f(x_n + 1, y_n - 1) = (x_n + 1)^2 + (y_n - 1)^2 - r^2$$

- Now if $f(A) = 0$ then point A lies on circle so we have to plot that point else we have to check, if $(f(A) < 0)$ i.e. Is point A lies inside the circle. Or if $(f(A) > 0)$, i.e. Is point A lies outside the circle and accordingly we have to select a particular point.
- To select one point from A and B we are having five different cases, Fig. 2.5.11 shows sketches of all the five cases with two candidate pixels A and B.

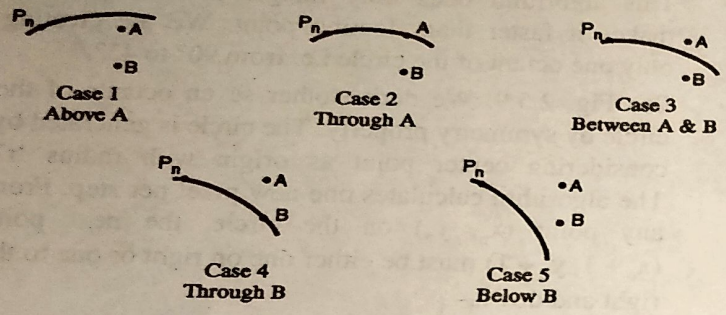


Fig. 2.5.11

- To determine which point to select we are using equation of circle. To be more specific, we are dealing with only the sign of that function, if it is negative or positive.
- Let us summarize this result in a tabular form for all the 5 cases.

	Case 1	2	3	4	5
f (A)	–	0	+	+	+
f (B)	–	–	–	0	+
Sum	–	–	–, 0, +	+	+

- Observe the case 2 of Fig. 2.5.11. Here curve is passing through point Ai.e. f (A) must be equal to 0; And as point B is lying inside, f (B) < 0. So we are selecting point A.
- Similarly observe case 4 of Fig. 2.5.11. Here point B is lying on curve and point A is lying outside. So, f (A) must be > 0 and f (B) must be equal to 0.
- These two cases are the simplest cases. But if the situation is like case 1 orcase 5. Then in both cases the curve lies on one side of both points.

Let's concentrate on case 1. Here curve is above A and B point. It means both points A and B are lying inside the curve then both f (A) and f (B) must be < 0. So which point to select? Since both points are inside only, we are introducing new term "sum" which will be summation of both f (A) and f (B) i.e. sum = f (A) + f (B). Here we are going to plot point A because it is more near to actual curve than the point B.

- Similarly in case 5, the curve is below to both points A and B i.e. both points are outside means the f (A) and f (B) must be > 0. Here also we are selecting a point which is more nearer to the curve. In this case point B is more nearer so we are selecting that point.
- Now to decide when to select point A or point B, we are using sum variable. Here we are interested in only the sign of sum variable.
- In case 1 both f (A) and f (B) are negative so the sum will be obviously negative and in case 5 both f (A) and f (B) are positive so the sum will be positive only. From this we can make a statement as, if the sign of sum variable is negative then select point A and if the sign of sum variable is positive then select point B.
- In case 3 of Fig. 2.5.11 point A lies outside the curve and B lies inside. So f (A) must be > 0 and f (B) must be < 0. Then the sum variable will contain either negative or 0 or positive value, depending on the value of f (A) and f (B). If the sum is negative or 0, then pixel A is closer to circle and we have to select A point. If it is positive then, we have to select point B.
- We are going to choose the point A or B depending on sign of sum variable. It is not necessary to calculate both formulas and the sum for every step.
- The starting pixel P_0 is taken as $(x_0, y_0) = (0, r)$. Refer Fig. 2.5.12.

Then
$$\begin{aligned} f(A) &= f(x_0 + 1, y_0) \\ &= (x_0 + 1)^2 + (y_0)^2 - r^2 \\ &= (0 + 1)^2 + (r^2) - r^2 \\ &= 1 \end{aligned}$$

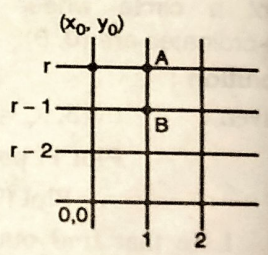


Fig. 2.5.12

$$\begin{aligned} \text{Similarly, } f(B) &= f(x_0 + 1, y_0 - 1) \\ &= f(x_0 + 1, r - 1) \\ &= (0 + 1)^2 + (r - 1)^2 - r^2 \\ &= 0 + 0 + 1 + r^2 - 2r + 1 - r^2 \\ &= 2 - 2r \end{aligned}$$

$$\therefore \quad \text{Sum} = f(A) + f(B)$$
$$= 1 + 2 - 2r$$
$$\therefore \quad \text{Sum} = 3 - 2r$$

So depending on whether sum is <, > or = 0, we will select A or B as next point. And we can find next sum going from P_1 to P_2 by considering old sum (S) value. Consider Fig. 2.5.13.

If we want to make it generalized then

$$\text{Sum} = f(A) + f(B)$$
$$= f(x_n + 1, y_n) + f(x_n + 1, y_n - 1)$$
$$\therefore S = (x_n + 1)^2 + y_n^2 - r^2 + (x_n + 1)^2 + (y_n - 1)^2 - r^2$$

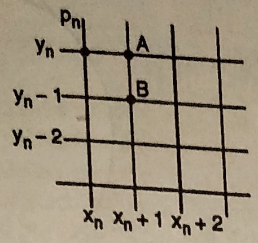


Fig. 2.5.13

Step 1 : If $S < 0$ then we will print next point as A i.e. $(x_n + 1, y_n)$. We will call this point as $(P_n + 1)$. Now in order to go for next pixel i.e. $(P_n + 2)$ we must compute sum again. We will call this as S_{new} and express it using point P_n. See Fig. 2.5.14.

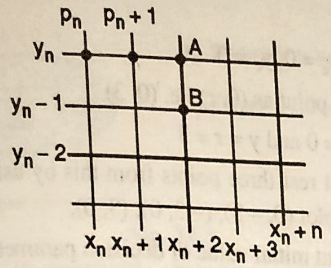


Fig. 2.5.14

$$S_{new} = f(A) + f(B)$$
$$= f(x_n + 2, y_n) + f(x_n + 2, y_n - 1)$$
$$\therefore S_{new} = (x_n + 2)^2 + y_n^2 - r^2 + (x_n + 2)^2 + (y_n - 1)^2 - r^2$$

Solving this we will get,

$$S_{new} - S = 4x_n + 6$$
$$\therefore S_{new} = S + 4x_n + 6$$

$\therefore$ For the next pixel $(P_n + 2)$ we have to select point A if $S_{new} <= 0$ else select point B.

Step 2 : But if at initial stage, $S > 0$ then we will print $(P_n + 1)$ as $(x_n + 1, y_n - 1)$. See Fig. 2.5.15.

Now in this case to find $P_n + 2$

$$S_{new} = f(A) + f(B)$$
$$= f(x_n + 2, y_n - 1) + f(x_n + 2, y_n - 2)$$

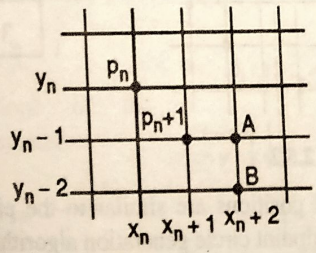


Fig. 2.5.15

$$\therefore S_{new} = (x_n + 2)^2 + (y_n - 1)^2 - r^2 + (x_n + 2)^2 + (y_n - 2)^2 - r^2$$

Solving this we will get,

$$S_{new} - S = 4(x_n - y_n) + 10$$
$$\therefore S_{new} = S + 4(x_n - y_n) + 10$$

This shows that calculating S_{new} from old 'S' is very simple. It uses very few operations like addition, subtraction and multiplication. Here, we are making use of just sign of this 'S' variable to determine which pixel to draw.

☞ Steps for Bresenham's circle generation algorithm

1) Accept radius and center co-ordinates from user and plot first point on circumference of circle.
$$(x, y) = (0, r)$$
2) Calculate the initial value of decision parameter.
$$S = 3 - 2r$$
3) If we are using octant symmetry property to plot the pixels then until $(x < y)$ we have to perform following steps :

   If $(S <= 0)$

   Update S by $S = S + 4x + 6$ and increase x by 1.

   else

   Update S by $S = S + 4(x - y) + 10$

   and increase x by 1 and decrease y by 1.
4) Determine the symmetry points in other octants also.
5) Move each calculated pixel position (x, y) on to the circular path centered on(x_c, y_c) and plot the co-ordinate values as,
$$x = x + x_c \text{ and } y = y + y_c$$

☞ Bresenham's Circle Generation program

```
/*********************************************
Program :
A C program to draw a circle by using Bresenham's circle
generation algorithm
*********************************************/
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
#include<graphics.h>
#include<dos.h>
void circ_bre(int x,int y,int rad);
void display(int,int,int,int);
void main()
{
    int gd = DETECT, gm, x,y,r;
    initgraph(&gd,&gm,"c:\\tcplus\\bgi");
    cleardevice();
```

Syllabus Topic : Midpoint Algorithm for Ellipse Drawing

2.6 Ellipse Generation

→ **(May 2013)**

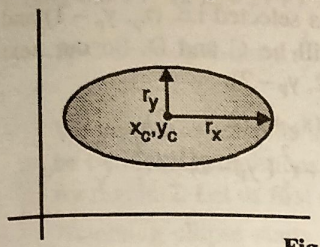
Q. Derive the midpoint algorithm for ellipse generation. **MU - May 2013, 10 Marks**

Q. Explain Midpoint ellipse algorithm with all required mathematical representation. **(12 Marks)**

Q. Derive decision parameters for the midpoint ellipse algorithm (Region 1), assuming the starting position is $(0, r_y)$ and points are to be generated along the curve path in clockwise order. **(10 Marks)**

Ellipse generating algorithms

- Ellipse is nothing but elongated circle. Therefore it can be generated by modifying circle drawing procedures.
- Ellipse has major axis and minor axis. Refer Fig. 2.6.1.



r_x Represents semimajor axis

r_y Represents semiminor axis

Fig. 2.6.1

- Ellipse is symmetric between quadrants, but not symmetric between the two octants of a quadrant.
- Thus we have to calculate pixel positions along the elliptical arc throughout one quadrant, then we obtain positions in the remaining 3 quadrants by using symmetry property. See Fig. 2.6.2.

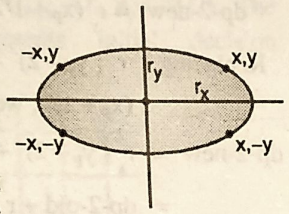


Fig. 2.6.2

Midpoint ellipse generation method

- The *midpoint ellipse* method is applied throughout the first quadrant in two parts i.e. region-1 and region-2. We are forming this regions by considering the slope of the curve. If the slope of the curve is less than –1 then we are in region-1 and when the slope becomes greater than – 1 then in region-2. See Fig. 2.6.3.

i.e. At the boundary between region-1 and 2

$$\frac{dy}{dx} = -1.$$

The slope of the ellipse is calculated as

$$\frac{dy}{dx} = -\frac{2r_y^2 x}{2r_x^2 y}$$

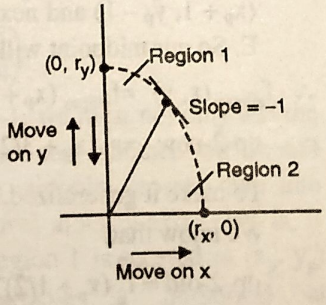


Fig. 2.6.3

At the boundary between region-1 & 2,

$$\frac{dy}{dx} = -1 \quad \text{and} \quad 2r_y^2 x = 2r_x^2 y$$

So, we move out of region-1 when

$$2r_y^2 x \geq 2r_x^2 y$$

- For region-1, we can start at $(0, r_y)$ and step clockwise along the elliptical path in the 1st quadrant shifting from unit steps x to unit steps in y, till the slope becomes less than – 1 **OR** we can start at $(r_x, 0)$ and select points in anticlockwise shifting from unit steps in y to unit steps in x, till the slope becomes greater than – 1.

Equation of ellipse is,

$$f_{ellipse}(x, y) = r_y^2 x^2 + r_x^2 y^2 - r_x^2 r_y^2$$

- Here we are making use of this function to decide whether the point is inside or outside the elliptical curve. Same as in case of circle.

If $f_{ellipse}(x, y) < 0$ then (x, y) is inside the ellipse boundary.

$= 0$ then (x, y) is on boundary.

> 0 then (x, y) is outside the ellipse boundary

- Thus the $f_{ellipse}(x, y)$ acts as a decision parameter. So, we are going to select the next pixel along the path of ellipse, according to the sign of the function.
- We are starting at position $(0, r_y)$ and take unit steps in the x-direction until we reach the boundary between region-1 and 2. Once we reach to the boundary, we switch to unit steps in the y-direction for the remainder of the curve.

Here we have to calculate the slope at each step.

$$\frac{dy}{dx} = -\frac{2r_y^2 x}{2r_x^2 y}$$

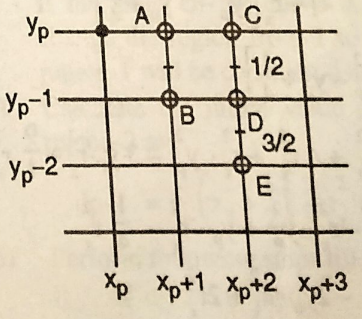
We move to region-2 from region-1 whenever

$$2r_y^2 x \geq 2r_x^2 y$$

In region-1 if the current pixel is located at (x_p, y_p). Then next candidate pixels are A & B, So the decision variable for region-1 will be $(x_p + 1, y_p - 1/2)$, as we are using midpoint algorithm. See Fig. 2.6.4.

$$\therefore \quad f_{ellipse}(x, y) = f_{ellipse}(x_p + 1, y_p - 1/2)$$

$$\text{dp-1-old} = r_y^2 (x_p + 1)^2 + r_x^2 (y_p - 1/2)^2 - r_x^2 r_y^2$$



Changing x uniformly and deciding which row is nearer

Fig. 2.6.4

1) If $f_{ellipse}(x_p + 1, y_p - 1/2) <= 0$ then print 'A'.

And for next point our candidate pixels will be C and D. So our next midpoint will be $(x_p + 2, y_p - 1/2)$.

$$\therefore f_{ellipse}(x, y) = f_{ellipse}(x_p + 2, y_p - 1/2)$$

$$\text{dp-1-new} = r_y^2 (x_p + 2)^2 + r_x^2 (y_p - 1/2)^2 - r_x^2 r_y^2$$

To make it generalized.

We know that,

$$\text{dp-1-old} = r_y^2 (x_p + 1)^2 + r_x^2 (y_p - 1/2)^2 - r_x^2 \cdot r_y^2$$

$$\text{and dp-1-new} = r_y^2 (x_p + 2)^2 + r_x^2 (y_p - 1/2)^2 - r_x^2 \cdot r_y^2$$

Now replace $r_x^2 (y_p - 1/2)^2 - r_x^2 \cdot r_y^2$ as $\text{dp-1-old} - ry^2 (x_p + 1)^2$

$$= r_y^2 (x_p + 2)^2 + \text{dp-1-old} - r_y^2 (x_p + 1)^2$$

$$= \text{dp-1-old} + r_y^2 (x_p^2 + 4x_p + 4) - r_y^2 (x_p^2 + 2x_p + 1)$$

$$= \text{dp-1-old} + r_y^2 \cdot x_p^2 + 4x_p \cdot r_y^2 + 4r_y^2 - r_y^2 x_p^2 - 2x_p r_y^2 - r_y^2$$

$$= \text{dp-1-old} + 2x_p \cdot r_y^2 + 3r_y^2$$

$$= \text{dp-1-old} + r_y^2 (2x_p + 3)$$

$$\therefore \text{dp-1-new} = \text{dp-1-old} + r_y^2 (2x_p + 3)$$

(i.e. if y is not changed)

2) But if $f_{ellipse}(x_p + 1, y_p - 1/2) > 0$ then print point 'B' and for next point our candidate pixels will be D and E. So our next midpoint will be $(x_p + 2, y_p - 3/2)$.

$$\therefore f_{ellipse}(x, y) = f_{ellipse}(x_p + 2, y_p - 3/2)$$

$$\therefore \text{dp-1-new} = r_y^2 (x_p + 2)^2 + r_x^2 (y_p - 3/2)^2 - r_x^2 \cdot r_y^2$$

If we represent this in terms of dp-1-old i.e. with respect to (x_p, y_p).

Then dp-1-new will be as follows :

As we know that,

$$\text{dp-1-old} = r_y^2 (x_p + 1)^2 + r_x^2 (y_p - 1/2)^2 - r_x^2 \cdot r_y^2$$

$$\text{and dp-1-new} = r_y^2 (x_p + 2)^2 + r_x^2 (y_p - 3/2)^2 - r_x^2 \cdot r_y^2$$

Replacing $r_y^2 \cdot r_x^2$ as $\text{dp-1-old} - r_y^2 (x_p + 1)^2 - r_x^2 (y_p - 1/2)^2$ we get

$$\text{dp-1-new} = r_y^2 (x_p + 2)^2 + r_x^2 (y_p - 3/2)^2 + \text{dp-1-old} - r_y^2 (x_p + 1)^2 - r_x^2 (y_p - 1/2)^2$$

$$= \text{dp-1-old} + r_y^2 [x_p^2 + 4x_p + 4] + r_x^2 \left[y_p^2 - 3y_p + \frac{9}{4}\right] - r_y^2 [x_p^2 + 2x_p + 1] - r_x^2 \left[y_p^2 - y_p + \frac{1}{4}\right]$$

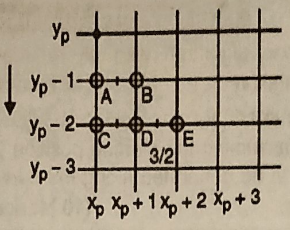
$$= \text{dp-1-old} + r_y^2 x_p^2 + 4x_p r_y^2 + 4r_y^2 + r_x^2 \cdot y_p^2 - 3y_p \cdot r_x^2 + \frac{9}{4} r_x^2 - r_y^2 x_p^2 - 2x_p \cdot r_y^2 - r_y^2 - r_x^2 \cdot y_p^2 + y_p \cdot r_x^2 - \frac{1}{4} r_x^2$$

$$= \text{dp-1-old} + 3r_y^2 + 2x_p \cdot r_y^2 - 2y_p \cdot r_x^2 + 2r_x^2$$

$$= \text{dp-1-old} + r_y^2 (3 + 2x_p) + r_x^2 (2 - 2y_p)$$

$$\text{dp-1-new} = \text{dp-1-old} + r_y^2 (2x_p + 3) + r_x^2 (-2y_p + 2)$$ (i.e. if y is changed)

Similarly, in region-2, if current pixel is at (x_p, y_p) then candidate pixels are A and B, so midpoint will be $(x_p + 1/2, y_p - 1)$. See Fig. 2.6.5.



Changing y uniformly and deciding which column is closer

Fig. 2.6.5

$$\therefore f_{ellipse}(x, y) = f_{ellipse}(x_p + 1/2, y_p - 1)$$

$$\therefore \text{dp-2-old} = f_{ellipse}(x_p + 1/2, y_p - 1)$$

$$\therefore \text{dp-2-old} = r_y^2 (x_p + 1/2)^2 + r_x^2 (y_p - 1)^2 - r_x^2 \cdot r_y^2$$

1) If dp-2-old > 0 then A is selected i.e. $(x_p, y_p - 1)$ and next candidate pixels will be C and D. So our next midpoint will be $(x_p + 1/2, y_p - 2)$.

$$\therefore f_{ellipse}(x, y) = f_{ellipse}(x_p + 1/2, y_p - 2)$$

$$\text{dp-2-new} = r_y^2 (x_p + 1/2)^2 + r_x^2 (y_p - 2)^2 - r_x^2 \cdot r_y^2$$

To make it generalized.

We know that,

$$\text{dp-2-old} = r_y^2 (x_p + 1/2)^2 + r_x^2 (y_p - 1)^2 - r_x^2 \cdot r_y^2$$

$$\text{dp-2-new} = r_y^2 (x_p + 1/2)^2 + r_x^2 (y_p - 2)^2 - r_x^2 \cdot r_y^2$$

Replacing $r_x^2 (y_p - 2)^2 - r_x^2 \cdot r_y^2$ as $\text{dp-2-old} - r_y^2 (x_p + 1)^2$, we get

$$\text{dp-2-new} = r_x^2 (y_p - 2)^2 + \text{dp-2-old} - r_x^2 (y_p - 1)^2$$

$$= \text{dp-2-old} + r_x^2 (y_p^2 - 4y_p + 4) - r_x^2 (y_p^2 - 2y_p + 1)$$

$$= \text{dp-2-old} + r_x^2 \cdot y_p^2 - 4y_p r_x^2 + 4r_x^2 - r_x^2 \cdot y_p^2 + 2y_p \cdot r_x^2 - r_x^2$$

$$= \text{dp-2-old} + 3r_x^2 - 2y_p \cdot r_x^2$$

$$= \text{dp-2-old} + r_x^2 (-2y_p + 3)$$

$$\therefore \text{dp-2-new} = \text{dp-2-old} + r_x^2 (-2y_p + 3)$$

...(i.e. if x is not changed)

2) But If dp-2-old < = 0 then point 'B' is selected i.e. $(x_p + 1, y_p - 1)$ and next candidate pixels will be D and E. So our midpoint will be $(x_p + 3/2, y_p - 2)$

$$\therefore f_{ellipse}(x, y) = f_{ellipse}(x_p + 3/2, y_p - 2)$$

$$\therefore \text{dp-2-new} = r_y^2 (x_p + 3/2)^2 + r_x^2 (y_p - 2)^2 - r_x^2 \cdot r_y^2$$

To make it generalized.

We know that,

$$\text{dp-2-old} = r_y^2 (x_p + 1/2)^2 + r_x^2 (y_p - 1)^2 - r_x^2 \cdot r_y^2$$

$$\text{and dp-2-new} = r_y^2 (x_p + 3/2)^2 + r_x^2 (y_p - 2)^2 - r_x^2 \cdot r_y^2$$

Replacing

$r_x^2 \cdot r_y^2$ as dp-2-old $- r_y^2 (x_p + 1/2)^2 - r_x^2 (y_p - 1)^2$

we get,

$$\text{dp-2-new} = r_y^2 (x_p + 3/2)^2 + r_x^2 (y_p - 2)^2 + \text{dp-2-old} - r_y^2 (x_p + 1/2)^2 - r_x^2 (y_p - 1)^2$$

$$= \text{dp-2-old} + r_y^2 \left[x_p^2 + 3x_p + \frac{9}{4}\right] + r_x^2 [y_p^2 - 4y_p + 4] - r_y^2 \left[x_p^2 + x_p + \frac{1}{4}\right] - r_x^2 [y_p^2 - 2y_p + 1]$$

$$= \text{dp-2-old} + r_y^2 x_p^2 + 3x_p \cdot r_y^2 + \frac{9}{4} r_y^2 + r_x^2 y_p^2 - 4y_p \cdot r_x^2 + 4r_x^2 - r_y^2 \cdot x_p^2 - x_p \cdot r_y^2 - \frac{1}{4} r_y^2 - r_x^2 \cdot y_p^2 + 2y_p \cdot r_x^2 - r_x^2$$

$$= \text{dp-2-old} + 2x_p \cdot r_y^2 + 2r_y^2 - 2y_p \cdot r_x^2 + 3r_x^2$$

$$= \text{dp-2-old} + r_y^2 (2x_p + 2) + r_x^2 (3 - 2y_p)$$

$\therefore$ dp-2-new = dp-2-old $+ r_y^2 (2x_p + 2) + r_x^2 (3 - 2y_p)$

...(i.e. if x is changed)

- We must also compute the initial condition. There will be two initial conditions, one for region-1 and another for region-2. Let us first see for region-1.
- Assuming ellipse starts at $(0, r_y)$. So candidate pixels are A and B. See Fig. 2.6.6.
- $\therefore$ Midpoint will be $(1, r_y - 1/2)$ and the initial value of decision parameter will be $f_{ellipse}(1, r_y - 1/2)$. The variable dp-1-initial represents initial decision parameter for region-1.

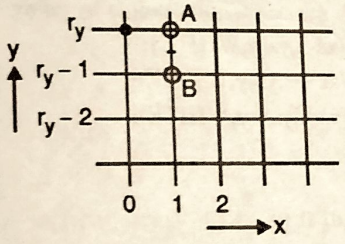


Fig. 2.6.6

$$\text{dp-1-initial} = r_y^2 + (r_y - 1/2)^2 \cdot r_x^2 - r_x^2 \cdot r_y^2$$

$$= r_y^2 + \left[r_y^2 - r_y + \frac{1}{4}\right] r_x^2 - r_x^2 \cdot r_y^2$$

$$= r_y^2 + r_x^2 \cdot r_y^2 - r_x^2 \cdot r_y + \frac{1}{4} r_x^2 - r_x^2 \cdot r_y^2$$

$$= r_y^2 + r_x^2 \left(-r_y + \frac{1}{4}\right)$$

- At each iteration in region-1, we must not only test the decision variable but also see whether we should switch regions. When the midpoint crosses over into region-2 we change to travel along y-axis. That is, if the last pixel chosen in region-1 is located at (x_p, y_p) then the decision variable is initialized at $(x_p + 1/2, y_p - 1)$. We stop drawing pixels in region-2 when the y-value of the pixel is equal to zero. See Fig. 2.6.7.

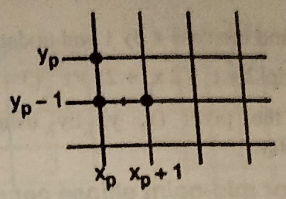


Fig. 2.6.7

The last point of region-1 will be acting as starting point of region-2.

Initial value of region-2 will be $f_{ellipse}(x_p + 1/2, y_p - 1)$.

$$\text{dp-2-initial} = r_y^2 (x_p + 1/2)^2 + r_x^2 (y_p - 1)^2 - r_x^2 \cdot r_y^2$$

$$= r_y^2 \left[x_p^2 + x_p + \frac{1}{4}\right] + r_x^2 [y_p^2 - 2y_p + 1] - r_x^2 \cdot r_y^2$$

$$= r_y^2 x_p^2 + r_y^2 \cdot x_p + \frac{1}{4} r_y^2 + r_x^2 \cdot y_p^2 - 2y_p \cdot r_x^2 + r_x^2 - r_x^2 \cdot r_y^2$$

$$\text{dp-2-initial} = r_y^2 \left[x_p^2 + x_p + \frac{1}{4}\right] + r_x^2 (1 - 2y_p)$$

Steps for midpoint ellipse generation

1) Accept r_x, r_y and ellipse center co-ordinates (x_c, y_c) and plot the first point on an ellipse centered on the origin as,

$$(x, y) = (0, r_y)$$

2) Calculate the initial value of decision parameter for region-1 as,

$$\text{dp-1} = r_y^2 + r_x^2 \left(-r_y + \frac{1}{4}\right)$$

3) We have to check whether the slope of the curve is < -1, if yes, then we are in region-1 and perform following steps for region-1.

   (i) If dp-1 is less than 0 then step along x-axis and modify decision parameter as,

$$\text{dp-1} = \text{dp-1} + r_y^2 (2x_p + 3)$$

Otherwise

Step along x-axis and decrease y by 1 and modify decision parameter as,

$$\text{dp-1} = \text{dp-1} + r_y^2 (2x_p + 3) + r_x^2 (-2y_p + 2)$$

   (ii) Draw the pixel (x, y) by using four point symmetry.

4) If the slope of the curve is > -1 then we have to change the region from 1 to 2. Now the last point of region-1 will be considered as first point of region-2.

5) Calculate the initial value of decision parameter for region-2 as,

$$\text{dp-2} = r_y^2 \left(x_p + \frac{1}{2}\right)^2 + r_x^2 (y_p - 1)^2 - r_x^2 \cdot r_y^2$$

6) Perform following steps till $(y > 0)$

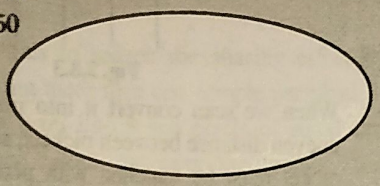
   i) If dp-2 > 0 then

   decrease y by 1 and update dp-2.

```
    ellipse1(xs1,ys1,rx1,ry1);
    getch();
    closegraph();
}
```

Output

```
Midpoint Ellipse Drawing Algorithm
Enter the Center Co-ordinates
xc = 250
yc = 200
Enter the X Radius 100
Enter the Y Radius 50
```



Example 2.6.1

Plot an ellipse having $r_y = 4$, $r_x = 3$ and whose center co-ordinates are $(x_c, y_c) = (0, 0)$.

Solution :

Given : $r_y = 4, r_x = 3$

Plot 1st point as (x, y) where x = 0 and y = ry i.e. (x, y) = (0, 4)

Then initial decision parameter is calculated by,

$$dp\text{-}1 = r_y^2 - r_x^2\left(-r_y + \frac{1}{4}\right) = (4)^2 - (3)^2\left(-(4) + \frac{1}{4}\right)$$

$$dp\text{-}1 = -\frac{71}{4}$$

Then we have to check whether we have crossed region-1 or not.

$$\therefore \quad 2r_y^2 x < 2r_x^2 y$$

$$2 \cdot (4)^2 \cdot (0) < 2 \cdot (3)^2 \cdot (4)$$

$$0 < 72$$

As $(2r_y^2 x)$ is smaller than $(2r_x^2 y)$ it means we are in region-1.

∴check for decision parameter, if it is negative then increase x only and update decision parameter.

Here $dp\text{-}1 = -\frac{71}{4}$.

∴ Increase x by 1, But don't change y and updating dp-1 gives.

$$dp\text{-}1 = dp\text{-}1 + r_y^2 (2x_p + 3) = \frac{249}{4}$$

So plot new (x, y) as (x = 1, y = 4)

Every time we have to check whether we have crossed the region or not. If yes, then find out the new initial value of region-2 and proceed along y-axis till y becomes zero.

At the end, i.e. when y become 0 we will get following table of x and yco-ordinates.

At last replicate these points to rest of the quadrants to get full ellipse as shown in Fig.P.2.6.1.

x	y
0	4
1	4
2	3
3	2
3	1
3	0

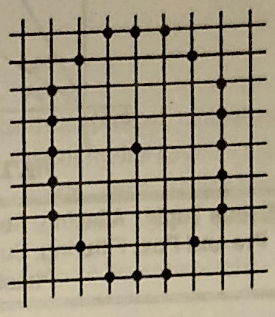


Fig. P. 2.6.1

2.7 Arcs and Sectors

☞ Arc

- An arc may be generated by using either the polynomial or trigonometric method. When *polynomial method* is used, the value of x is varied from x_1 to x_2, and the values of y are found by evaluating the expression $y = \sqrt{r^2 - x^2}$.
- When *trigonometric method* is used, we have to provide starting angle θ_1 and ending angle θ_2. The rest of the steps are similar to circle generation except symmetry. As we want to draw an arc we should not replicate the point in other quadrants. See Fig. 2.7.1.
- Arcs are nothing but a portion of circles however problem occurs only in Bresenham's circle algorithm, because we must provide starting and ending points also for that. When you are calculating next point every time you have to compare that new point with end point and if end point is in between step, then your program goes in infinite loop.

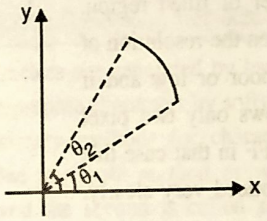


Fig. 2.7.1

☞ Sectors

- Sector is scan converted by using any method of scan-converting an arc and then scan converting two lines from the center of the arc to the endpoints of the arc. Refer Fig. 2.7.2. e.g. say center is (h, k) and we want to scan convert an arc from θ_1 to θ_2.
- Then a line would be scan-converted from (h, k) to (r cos (θ_1) + h, r sin (θ_1) + k). A second line would be scan-converted from :

  (h, k) to (r cos (θ_2) + h, r sin (θ_2) + k).

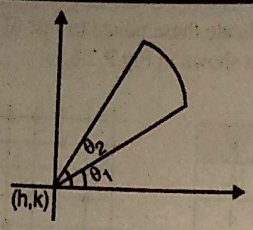


Fig. 2.7.2

Syllabus Topic : Aliasing, Anti-aliasing Techniques like Pre and Post Filtering, Super Sampling and Pixel Phasing

2.8 Aliasing and Anti-aliasing

➔ **(May 2013, May 2015, Dec. 2016)**

Q. What is aliasing ? Explain some antialiasing techniques. **MU - May 2013, 5 Marks**

Q. What are aliasing and antialiasing ? Explain any one antialiasing method. **MU - May 2015, 5 Marks**

Q. What is aliasing ? Explain any two antialiasing techniques. **MU - Dec. 2016, 5 Marks**

Q. Explain Antialiasing Techniques. **(5 Marks)**

- The various forms of distortion that results from the scan conversion operations are collectively called as **aliasing**.

☞ Staircase

A common example of aliasing effects is the staircase or jagged appearance. We see this effect when scan converting a primitive such as line or circle. We also see the stair steps or jaggies along the border of filled region. Specifically when the resolution of the display is poor or low and if the display allows only two pixel states ON or OFF in that case this effect can be noticed very clearly. See Fig. 2.8.1.

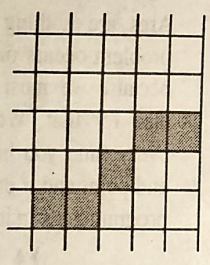


Fig. 2.8.1

☞ Unequal brightness

In this case a slanted line appears dimmer than a horizontal or vertical line, although all are presented at the same intensity level. The reason for this problem is the pixels on the horizontal line are placed one unit apart where as those on diagonal line are approximately 1.414 units apart.

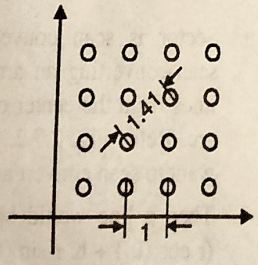


Fig. 2.8.2

This difference in distance produces the difference in brightness. Refer Fig. 2.8.2.

☞ Picket fence problem

- Picket fence problem occurs when an object is not aligned with the pixel grid properly. See Fig. 2.8.3.
- Here the distance between two adjacent pickets is not a multiple of unit distance between pixels.

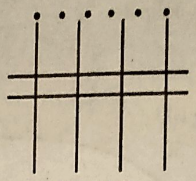


Fig. 2.8.3

- When we scan convert it into image it will result in uneven distance between pickets; since the end point of picket must be matched with pixel co-ordinates. See Fig. 2.8.4. This is sometimes called *global aliasing*, as the overall length of the picket is approximately correct.

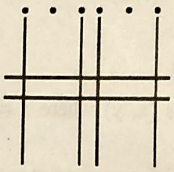


Fig. 2.8.4

- And to maintain the proper/equal length spacing sometimes it will distort the overall length of fence. This is sometimes called as *local aliasing*. See Fig. 2.8.5.

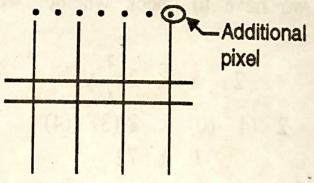


Fig. 2.8.5

☞ Antialiasing

- Most aliasing effects, occurs in static images at a moderate resolution, are often tolerable or negligible. But when the image is moving or in animation this drawback becomes bold. We can come out of this drawback by straight way increasing the image resolution.
- But for that we have to pay extra money and still the results are not always satisfactorily. There are techniques that can greatly reduce this effect and improve the images. Such techniques are collectively known as antialiasing.
- When we have displays that allow *more than two colors*, we can use the following techniques to soften the jaggies.

1. Prefiltering

- This is a technique that determines pixel intensity based on the amount of that particular pixels coverage by the object in the scene i.e. It computes pixel colors depending on an objects coverage. It means how much part or fraction of that pixel is covered by the object and depending on that, it sets the value of that pixel. It requires large number of calculations and approximations. Prefiltering generates more accurate antialiasing effect. But due to its high complexity of calculations it is not used.

2. Supersampling

- Supersampling tries to reduce the aliasing effect by sampling or taking more than one sample per pixel. It subdivides each pixel into 9 sub pixels of equal size which is often called as sampling points. See Fig. 2.8.6. Some of these 9 sub pixels may get color of background i.e. a line may not pass through them and rest of sub pixels may get color of object.

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

Fig. 2.8.6

- Suppose 3 samples or sub pixels get background color and 6 gets foreground color then the color of the pixel will be the sum of $\left(\frac{1}{3}\right)$ of background color and $\left(\frac{2}{3}\right)$ of object color. So ultimately the pixel value becomes the average of several samples.

3. Postfiltering

1/8	1/8	1/8
1/8	50%	1/8
1/8	1/8	1/8

Fig. 2.8.7

- Postfiltering and supersampling are almost same. If we consider 9 sampling points as in supersampling, we may give a lot more weightage to center point.We could give the center sample as weightage of 50% or 1/2; and remaining 50% is distributed among the rest of 8 samples. See Fig. 2.8.7.
- So we can treat supersampling as the special case of postfiltering in which each sampling point has an equal weightage $\left(\frac{1}{9}\right)$.

4. Pixel phasing

- Pixel phasing is a hardware based anti-aliasing technique. The graphics system in this case is capable of shifting individual pixels from their normal positions in the pixel grid by a fraction (typically $\frac{1}{4}$ or $\frac{1}{2}$) of unit distance between pixels.
- By moving pixels closer to the true line, this technique reduces the aliasing effect.

5. Gray level

- Many displays allow only two pixel states, ON and OFF. In that case we observe stair step aliasing effect. To reduce this effect we can make use of gray level technique.
- In this technique display allow setting pixels to gray levels between black and white to reduce the aliasing effect.
- See Fig. 2.8.8. It uses the gray levels to gradually turn off the pixels in one row as it gradually turns on the pixels in the next.

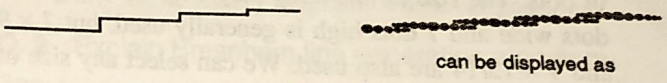


Fig. 2.8.8

2.9 Character Generation

Q. Explain Bit-Map character generation method. **(5 Marks)**

Q. Explain character Generation methods. **(5 Marks)**

- Usually characters are generated by hardware. But we can generate/prepare characters by software also. There are three primary methods for character generation. First is called as *stroke method* or *vector character* generation and the second is called as dot-matrix or *bitmap* method and third is called as *starbust method.*

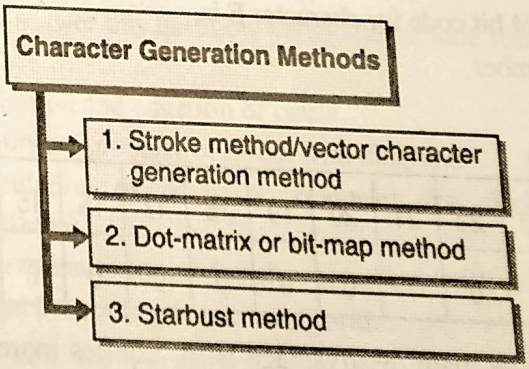


Fig. C2.2 : Character Generation

➜ **(1) Stroke method/vector character generation method**

- This method creates characters by using a set of line segments.
- We could build our own stroke method by vector generation algorithm or by using any line generation method.
- To produce a character we will give a sequence of commands that defines the start point and end points of the straight lines.

By using this we can change the scale of the characters. We can make a character twice as large as its original size. Similarly we can get characters slanted also. By using this method we can change the style of characters also. Different character styles are shown in Fig. 2.9.1.

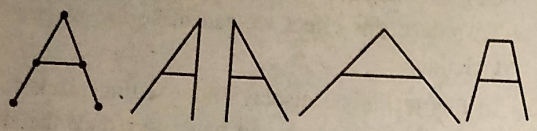


Fig. 2.9.1

➜ **(2) Dot-matrix or bit-map method**

- In this scheme, characters are represented by an array of dots. The size of this array may vary. An array of 5 dots wide and 7 dots high is generally used, but 7×9 and $9 \times 13/14$ are also used. We can select any size of array.

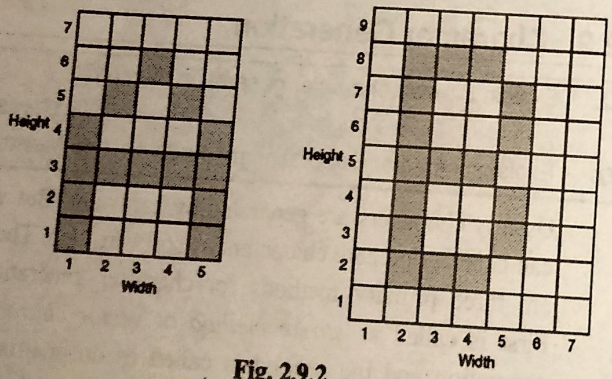


Fig. 2.9.2

- This array is like a small buffer, just big enough to hold a single character.
- The dots are the pixels of the small array.
- Placing the character on the screen then becomes a matter of copying pixel values from small character array into some portion of the screens frame buffer. See Fig. 2.9.2.
- A bitmap font uses a rectangular pattern of pixels to define each character.
- The entire font can be loaded into an area of memory called font cachedisplaying character means then copying characters image from font cache into the frame buffer at the desired position.
- Bitmap fonts require more space, because each variation (size or format) must be stored.

➜ **(3) Starbust method**

- In this method a fix pattern of line segments are used to generate characters. There are 24 line segments and out of these 24 line segments, segments required to display for particular character are highlighted.
- This method of character generation is called Starbust method because of its characteristic appearance.
- The pattern for particular character is 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.

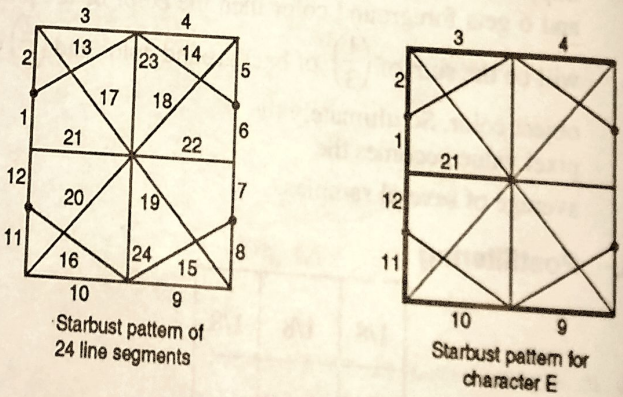


Fig. 2.9.3

24 bit code for character E is,

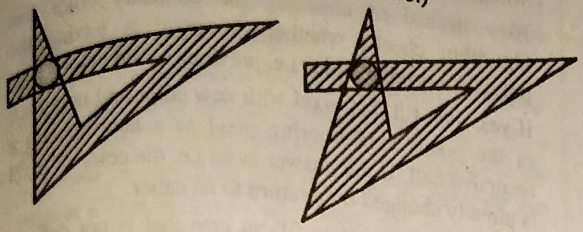
Bit number

24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
0	0	0	1	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1

But its main disadvantage is, it requires more memory and requires additional code conversion programs to display characters from the 24 bit code.

be + 1 and for D will be zero. Therefore sum will be + 1 + 0 = + 1 i.e. as the result is nonzero, the point A lies inside the polygon.

- Use of even-odd or winding number method will depend on application. But the even-odd and winding number method gives different results for the overlapping polygons. (See Fig. 3.2.10.)



Even odd Winding number method

Fig. 3.2.10

Syllabus Topic : Filled Area Primitive - Scan Line Polygon Fill Algorithm, Boundary Fill and Flood Fill Algorithm

3.3 Polygon Filling

Q. Explain 4-connected and 8-connected methods. **(5 Marks)**

- `Filling is the process of "coloring in " a fixed area or region. Regions may be defined at pixel level or geometric levels. When the regions are defined at pixel level, we are having different algorithms like :

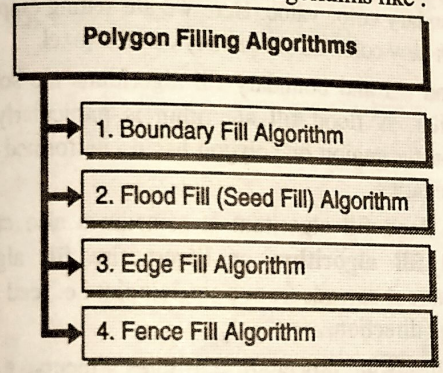


Fig. C3.3 : Polygon filling algorithms

- In case of geometric level we are having scan line fill algorithm.
- We will first see pixel level polygon filling. But before going to actual algorithm, we will see 4-connected and 8-connected pixel concept. These two techniques are the ways in which pixels are considered as connected to each other.
- In 4-connected method the pixels may have upto 4 neighbouring pixels. (See Fig. 3.3.1.). Let's assume that the current pixel is (x, y). From this pixel we can find four neighboring pixels as right, above, left and below of the current pixel.

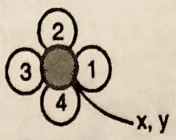


Fig. 3.3.1

- Similarly in 8-connected method the pixels may have upto 8 neighbouring pixels. (See Fig. 3.3.2.) Here from one pixel location we are finding the neighbouring 8 pixels position.

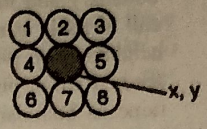


Fig. 3.3.2

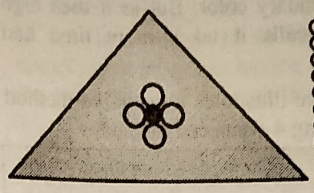


Fig. 3.3.3

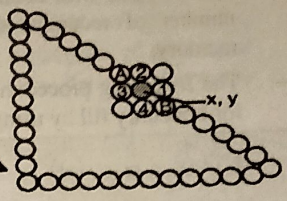


Fig. 3.3.4

- By using any one of these technique we can fill the interior of the polygon. (See Fig. 3.3.3.) But there are some cases where the use of 4-connected method is not efficient.
- Suppose we want to fill a triangle, so we are using 4-connected method. It will work fine but at boundary this method is not efficient. (See Fig. 3.3.4.). Suppose we have filled point (x, y). Now from that point we have to select either point A or B which is not possible by using 4-connected method because in 4-connected method we can go from (x, y) to either 1, 2, 3 or 4 and not A and B. So in this case we have to use 8-connected method where we can choose any one of the neighbouring pixel.

➔ 3.3.1 Boundary Fill Algorithm

➔ **(May 2017)**

Q. Write a boundary fill procedure to fill 8-connected region. **MU - May 2017, 5 Marks**

Q. Illustrate one example of polygon filling with diagram where 4-connected approach fails, while 8 connected approaches succeeds. **(7 Marks)**

Q. Write pseudo codes for boundary fill procedure. **(5 Marks)**

Q. Explain boundary fill algorithms. **(5 Marks)**

Q. Write a pseudo-code to implement boundary fill and flood fill algorithm using 4 connected method. **(10 Marks)**

- This algorithm is very simple. It needs one point which is surely inside the polygon. This point is called as "Seed point" which is nothing but a point from which we are starting the filling process. This is a recursive method. The algorithm checks to see if the seed pixel has a boundary pixels color or not.

- If the answer is no, then fill that pixel with color of boundary and make recursive call to itself using each of its neighbouring pixels as new seed. If the pixel color is same as boundary color then return to its caller. (See Fig. 3.3.5.)

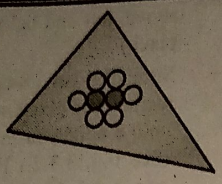


Fig. 3.3.5

- This algorithm works for any shaped polygon and fills that polygon with boundary color. But as it uses high number of recursive calls it takes more time and memory.
- The following procedure illustrates a recursive method for boundary fill by using 4-conneced method.

```
bfill (x, y, newcolor)
{
    current = getpixel (x, y) ;
    if (current != newcolor) && (current != boundary
                                    color)
    {
        putpixel (x, y, newcolor);
        bfill (x + 1, y, newcolor);
        bfill (x – 1, y, newcolor);
        bfill(x, y + 1, newcolor);
        bfill(x, y – 1, newcolor);
    }
}
```

- As we are using recursive function there is a chance that system stack may become full. So we have to develop our own logic to solve the stack problem. For this there are many solutions. One solution is instead of using system stack we can use our own stack and write our own PUSH and POP functions for that stack.
- Second solution could be use of link list i.e. dynamic memory allocation, for implementation of stack. Another solution could be develop some logic while pushing the pixels on stack i.e. before pushing the pixel on stack check whether that pixel is already visited or not.
- If it is already visited there is no point to store that pixel again. Another solution could be instead of using stack we can make use of queue also.
- But the boundary fill algorithm is having limitation. It fills the polygon having unique boundary color. If the polygon is having boundaries with different colours then this algorithm fails.

3.3.2 Flood Fill (Seed Fill) Algorithm

(Dec. 2014, May 2015, Dec. 2016)

Q. Explain 4 connected flood fill algorithm in detail.
MU - Dec. 2014, 5 Marks

Q. Write short note on : Flood fill algorithm.
MU - May 2015, Dec. 2016, 5 Marks

Q. Write pseudo codes for flood fill procedure. **(5 Marks)**

Q. Explain flood fill algorithms. **(5 Marks)**

- The limitations of boundary fill algorithms are overcome in flood fill algorithm. Like boundary fill algorithm this algorithm also begins with seed point which must be surely inside the polygon.
- Now instead of checking the boundary color this algorithm checks whether the pixel is having the polygon's original color i.e. previous or old color.
- If yes, then fill that pixel with new color and uses each of the pixels neighbouring pixel as a new seed in a recursive call. If the answer is no i.e. the color of pixel is already changed then return to its caller.
- Sometimes we want to fill an area that is not defined within a single color boundary. (See Fig. 3.3.6.). Here edge AB, BC, CD and DA are having red, blue, green and pink color, respectively.

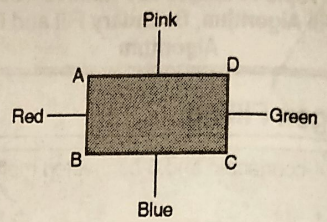


Fig. 3.3.6

- This Fig. 3.3.6 shows an area bordered by several color regions. We can paint such areas by replacing a specified interior color instead of searching for a boundary color value. Here we are setting empty pixel with new color till we get any colored pixel.
- Flood fill and boundary fill algorithms are somewhat similar. A flood fill algorithm is particularly useful when the region or polygon has no uniformed colored boundaries.
- The flood fill algorithm is sometimes also called as **seed fill algorithm** or Forest fire fill algorithm. Because it spreads from a single point i.e. seed point in all the direction.
- The following procedure illustrates a recursive method for flood fill by using 4-connected method.

```
f-fill (x, y, newcolor)
{   current = getpixel (x, y);
    if (current != newcolor)
    {   putpixel (x, y, newcolor);
        f-fill (x – 1, y, newcolor);
        f-fill ( x + 1, y, newcolor);
        f-fill (x, y – 1, newcolor);
        f-fill (x, y + 1, newcolor);
    }
}
```

- The efficiency of this algorithm can be increased in the same way as discussed in boundary fill algorithm.

3.3.3 Edge Fill Algorithm

- In edge fill algorithm the polygon is filled by selecting each edge of the polygon. This algorithm is very simple. The statement of this algorithm is, "for each scan line intersecting a polygon edge at (x_1, y_1) complement all pixels whose mid points lie to the right of (x_1, y_1)". Here the order in which the polygon edges are considered is unimportant.
- Let us try to understand this algorithm by considering an example. Refer Fig. 3.3.7. Suppose we want to fill a polygon ABCD which is shown in Fig. 3.3.7(a). As the statement of algorithm says we have to consider one edge at a time. So let us select edge AB. Now we have to complement all the pixels whose midpoints are lying on right hand side of the edge AB. After performing that task we will get the polygon as shown in Fig. 3.3.7(b). Now we have to select another edge say AD. Here the order of selection of edges is not important. Again complement all the pixels whose midpoints are lying on right hand side of selected edge. We will get the polygon as shown in Fig. 3.3.7(c). Complement means if earlier the pixel is having background color, now we have to make color of that pixel as fill color and if earlier it is fill color, then now we have to make it as background color. Like that we are selecting each and every edge of the polygon and complementing all the pixels which are on right hand side of selected edge till end of the screen.

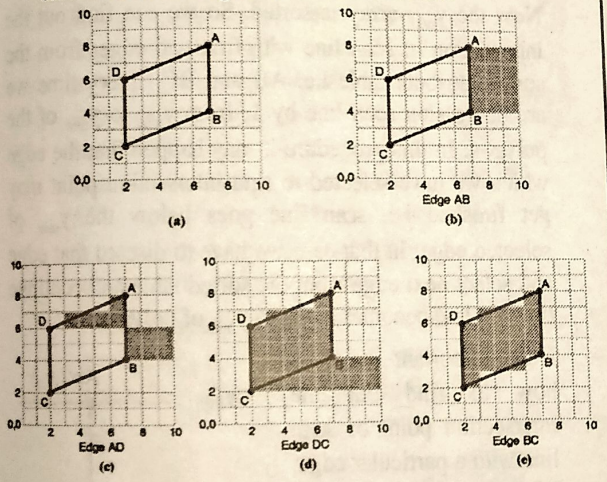


Fig. 3.3.7

- But the main drawback of this algorithm is, for complex pictures each individual pixel is addressed many times. So the algorithm requires more time. One more drawback of this algorithm is even though the polygon is placed in upper left hand corner of the screen, every time we are complementing all the pixels which are on right hand side of the selected edge till the end of the screen. So unnecessarily we are visiting pixels which we are not considering at all.

3.3.4 Fence Fill Algorithm

- In order to reduce the problems of edge fill algorithm, we are using Fence fill algorithm. The word fence means border. The algorithm is somewhat similar to edge fill algorithm. The fence fill algorithm says that for each scan line intersecting a polygon edge,
- If the intersection is to the left of the fence, complement all pixels having a midpoint to the right of the intersection of the scan line and the edge and to the left of the fence.
- If the intersection is to the right of the fence, complement all pixels having a midpoint to the left of the intersection of the scan line and the edge and to the right of the fence.
- Here in this algorithm we are using half scan line; either from edge to fence or from fence to edge only. The location of the fence can be selected anywhere in the polygon. Usually fence location is one of the polygon vertices. Let us consider an example. Refer Fig. 3.3.8.

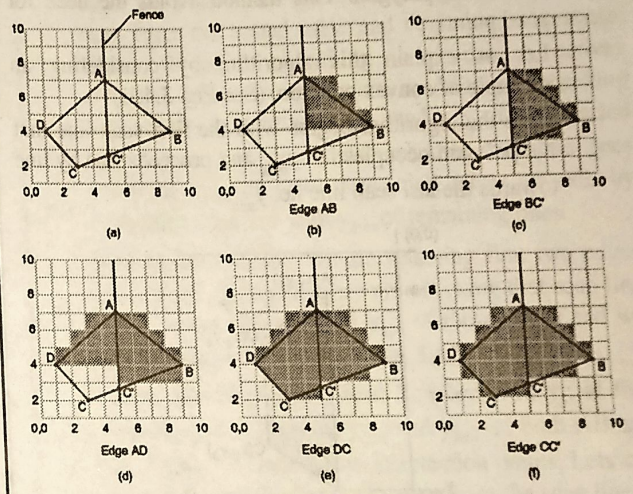


Fig. 3.3.8

- Suppose we have to fill polygon ABCD, so we will take a fence which will pass through any vertex, let us say the fence is passing from A and it cuts edge CB at C′. So our fence will be AC′. Now apply the algorithm to each edge. Let us consider edge AB. Edge AB is on right side of fence AC′ so complement all the pixels which are to the right of fence and to the left of edge AB. See Fig. 3.3.8(b).
- Then select edge BC′. Again this edge BC′ is at right side of fence so complement the pixels which are to the right side of fence and to the left of edge BC′. See Fig. 3.3.8(c). Now select edge AD. This edge is to the left of fence. So we have to complement all the pixels which are to the right of edge AD and to the left of fence. See Fig. 3.3.8(d). Like that we have to do for all the edges of the polygon. At last we will get a filled polygon which is shown in Fig.3.3.8(f).

- This algorithm is faster than the edge fill. But still the disadvantage of both edge fill and fence fill algorithms is, the number of pixels are addressed more than once.

3.4 Scan Line Filling

➔ **(May 2013, May 2014, May 2016)**

Q. Explain with example Scan line fill algorithm.
MU - May 2013, May 2014, 10 Marks

Q. Compare Boundary fill and flood fill algorithm.
MU - May 2016, 10 Marks

Q. Explain scan line fill area conversion algorithm with suitable example. **(8 Marks)**

- In contrast to boundary fill and flood fill algorithm at pixel level, this algorithm is defined at geometric level i.e. co-ordinates, edges, vertices etc. This algorithm starts with first scan line and proceeds line by line toward the last scan line and checks whether every pixel on that scan line satisfies our inside point test or not. i.e. it checks which points on that scan line are inside the polygon. This method avoids the need for seed point.
- Let us explain this algorithm by considering an example of convex polygon. (See Fig. 3.4.1.)
- Here the algorithm begins with the first scan line that the polygon occupies i.e. y_{max} and proceeds line by line towards the last scan line i.e. y_{min}.

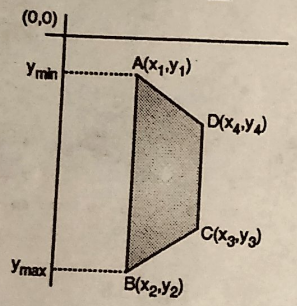


Fig. 3.4.1

Here we are considering first and last scan line of polygon not individual edges. For each edge of the polygon we are storing five attributes. As we know that to draw any edge or line we need two points. So we are going to store x_{max}, x_{min}, y_{max}, and y_{min} of the edges of the polygon along with their slopes. For edge AB we are going to store

$$x_{max} = x_2$$
$$x_{min} = x_1$$
$$y_{max} = y_2$$
$$y_{min} = y_1$$

Here x_{max} and x_{min} has as such no meaning, whereas y_{max} is the maximum y value for a particular edge and y_{min} is the minimum y value for that edge. x_{max} and x_{min} are the corresponding x component of y_{max} and y_{min}. Along with this we are going to store the slope of the dge also

Let us tabularize this

Edge	y_{max}	x_{max}	y_{min}	x_{min}	Slope
AB	y_2	x_2	y_1	x_1	m_1
AD	y_4	x_4	y_1	x_1	m_2
CD	y_3	x_3	y_4	x_4	m_3
BC	y_2	x_2	y_3	x_3	m_4

- For edge AB which is formed by (x_2,y_2) and (x_1,y_1), y_2 is greater than y_1. Therefore $y_{max} = y_2$ and $y_{min} = y_1$ and so $x_{max} = x_2$ and $x_{min} = x_1$.
- As we are filling the polygon line by line, at any particular time we are not considering all the edges. So there is no point to find whether a particular scan line intersects with each edge or not. In fact we have to select only those edges which are getting intersected by the scan line. To decide which edges are getting intersected by scan line we are making use of y_{max} of particular edge. One thing is sure that we are storing all the attributes of all edges. Out of those attributes we are selecting y_{max} and then we are sorting this y_{max} array. If some swapping is required in this y_{max} then we are going to swap whole edge. After sorting y_{max} array we will get following attribute table.

Edge	y_{max}	x_{max}	y_{min}	x_{min}	Slope
AB	y_2	x_2	y_1	x_1	m_1
BC	y_2	x_2	y_3	x_3	m_4
CD	y_3	x_3	y_4	x_4	m_3
AD	y_4	x_4	y_1	x_1	m_2

- Now the y_{max} array is sorted. So we can find out the intersection of scan line with first two edges from the sorted attribute table i.e. AB and BC. Every time we are decreasing scan line by 1, from y_{max} to y_{min} of the polygon. In this procedure it may happen that the edge which we have selected to find intersection point may get finished i.e. scan line goes below the y_{min} of selected edge. In that case we have to discard that edge and select next edge from the sorted table and continue till scan line becomes equal to y_{min} of polygon.
- Here important point is how to find out the intersection point of scan line with a particular edge. While at the time of storing attributes of edges we have stored the slope of that edge. We can make use of this slope to find intersection point of scan line with edge. (See Fig. 3.4.2.)

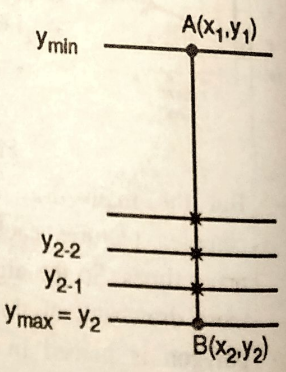


Fig. 3.4.2

- When we are moving from y_{max} to y_{min} every time we are decreasing y by 1, as the distance between two scan line is 1. So we know new points y value, which will be

y_{max}-1. Now we have to find, what will be its corresponding x value for the intersection point.

We know that

$$\text{Slope (m)} = \frac{\Delta y}{\Delta x} = \frac{\text{change in y}}{\text{change in x}} = \frac{y_{new} - y_{old}}{x_{new} - x_{old}}$$

$$m = \frac{1}{x_{new} - x_{old}}$$

$$x_{new} - x_{old} = \frac{1}{m}$$

$\therefore x_{new} = x_{old} + \frac{1}{m}$ Where m, we know from attribute table.

- x_{old} will be x-coordinate of previous lines intersection point.
- Thus by knowing the slope of every edges we can find the intersection point by decrementing 1 every time from y_{max} to y_{min} and calculating corresponding x by

$$x_{old} + \frac{1}{m}$$

- Like this we are finding intersection of scan line with every edge of polygon. Once we have found the intersection points with both edges i.e. selected edges like AB and BC, then join these two intersection points by solid line and continue. See Fig. 3.4.3.
- Again decrease y by 1 and find new intersection points for those edges till one of the edge gets over, i.e. the scan line goes below y_{min} of one edge. In this case edge BC gets over earlier, so discard edge BC and select next edge from sorted table, which is CD and continue the process. Now we have to find intersection points of scan line with edge AB and edge CD. And then join them. This process continues till scan line reaches to y_{min} of the polygon.

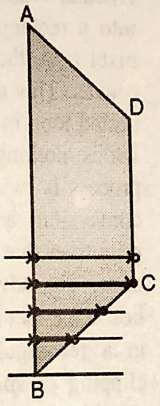


Fig. 3.4.3

- This will work very fine for convex polygons. But in some cases of concave polygons it will introduce errors. See the example given in Fig. 3.4.4.

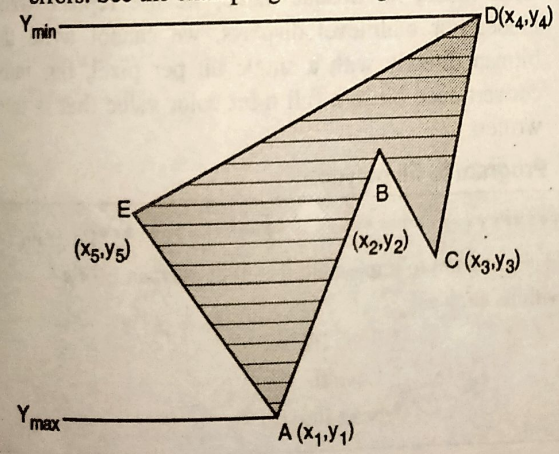


Fig. 3.4.4

- So to handle this situation we have to slightly modify the algorithm.
- Here also we are storing edges of polygon with their attributes. Then we are sorting the y_{max} array also. After sorting y_{max} array for polygon shown in Fig. 3.4.4, we will get the sorted table as follows :

edge	y_{max}	x_{max}	y_{min}	x_{min}	Slope
→ AB	y_1	x_1	y_2	x_2	m_1
→ AE	y_1	x_1	y_5	x_5	m_2
→ BC	y_3	x_3	y_2	x_2	m_3
→ CD	y_3	x_3	y_4	x_4	m_4
DE	y_5	x_5	y_4	x_4	m_5

- Now we are selecting 1st two edges i.e. AB and AE. We are starting our scan lines from y_{max} to y_{min}. If scan line is in between y_{max} and y_{min} of edge AB. Find out intersection point with that edge. Similarly for edge AE also. After finding intersection points join them by solid line. If scan line goes below y_{min} of any one of these two edges then discard that edge and select next edge from the sorted table and continue the process. Upto this point everything is similar to convex polygons. In concave polygons, in addition to checking the scan line is in between y_{max} and y_{min} of the selected lines we have to check also whether scan line goes below to y_{max} of other remaining edges or not, by comparing scan line with y_{max} of remaining lines.
- Now in our example we are selecting first two edges AB and AE. Every time we are checking that scan line is in between the y_{max} and y_{min} of these edges and we are drawing solid lines. See Fig. 3.4.5.
- But at one stage when scan line becomes y_p; at that time as y_p is in between y_{max} and y_{min} of both AB and AE. So we are finding two intersection points. Lets call them as P_1 and P_2. But at the same time the scan line y_p is below the y_{max} of next two edges of the sorted table, i.e. y_p becomes smaller than y_{max} of edge BC and CD.
- So we have to activate these two edges also and find out the intersection of scan line with these edges also. Lets assume that these points are P_3 and P_5. So join P_3 and P_4 and continue. Here we have to draw solid lines from P_1 to P_2 and P_3 to P_4. But not from P_2 to P_3.

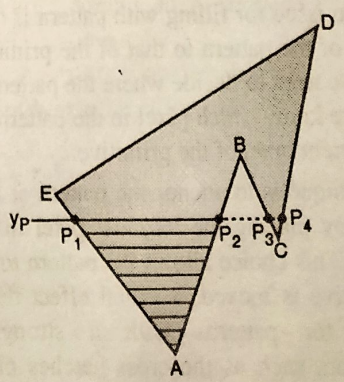


Fig. 3.4.5

- Similarly when the situation is as shown in Fig. 3.4.6. at that time we are going to using the vertex point twice i.e. we are drawing line P_1 to P_2 and P_2 to P_3.

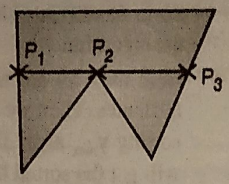


Fig. 3.4.6

- Table 3.4.1 represents the comparison of different polygon filling algorithms.

Table 3.4.1

Flood fill	Boundary fill	Edge fill	Fence fill	Scan line fill
Needs Seed point to start	Needs Seed point to start	Based on complimenting the pixels	Based on complimenting the pixels	Based on line drawing, polygon is filled
Current pixels color is compared with new pixel color	Current pixels color is compared with new pixel color and boundary color	Pixels which are on right side of an edge are getting complemented	Pixels which are on right side of an edge and to the left of fence as well as left side of edge and right side of fence are getting complemented	Intersection of polygon edges are found with the scan lines and then the solid lines are drawn between two such intersection points.
Useful for polygons having single color boundary	Useful for polygons having multi color boundary	More number of pixels are unnecessarily accessed	Less number of pixels are accessed as compared to Edge fill	Need separate attention to handle concave polygons

3.4.1 Pattern Fill Algorithm

- In this section we consider, filling with pattern, which we do by adding extra control to the part of the scan conversion algorithm that actually writes each pixel. The main issue for filling with pattern is the relation of the area of the pattern to that of the primitive. In other words, we need to decide where the pattern is anchored so that we know which pixel in the pattern corresponds to the current pixel of the primitive.
- One technique is to anchor the pattern at a vertex of a polygon by placing the leftmost pixel in the patterns first row. This choice allows the pattern to move when the primitive is moved, a visual effect that would be expected for patterns with a strong geometric organization, such as the cross hatches often used in drafting applications. But there is no distinguished point on a polygon that is obviously right for such a relative anchor, and no distinguished points at all on smoothly varying primitives such as circles and ellipses. Therefore, the programmer must specify the anchor point as appoint on or within the primitive. In some systems, the anchor point may even be applied to a group of primitives.
- Another technique is to consider the entire screen as being tiled with the pattern and to think of the primitive as consisting of an outline or filled area of transparent bits that let the pattern show through. To apply the pattern to the primitive, we index it with the current pixels (x,y) coordinates. Since patterns are defined as small P by Q bitmaps, we use modular arithmetic to make the pattern repeat. The pattern[0,0] pixel is considered coincident with the screen origin, and we can write, a bitmap pattern in transparent mode with the statement

```
If(pattern[x%P][y%Q])
 Writepixel (x,y,value);
```

- If we are filling an entire span in replace write mode, we can copy a whole of the pattern at once, assuming a low-level version of a copy pixel facility is available to write multiple pixels.
- Another technique is to scan convert a primitive first into a rectangular work area, and then to write each pixel from that bitmap to the appropriate place in the canvas. This rectangle write to the canvas is simply a nested loop in which a 1 writes the current color and 0 writes nothing or background color. This two step process is twice as much work as filling during scan conversion and therefore is not worthwhile for primitives that are encountered and scan converted only once. The advantage of a pre-scan converted bitmap lies in the fact that it is clearly faster to write each pixel in a rectangular region, without having to do any clipping or span arithmetic, than to scan convert the primitive each time from scratch while doing such clipping.
- For bi-level displays, writing current color 1, copy pixel works fine : For transparent mode, we use or write mode; for opaque mode, we use replace write mode. For multilevel displays, we cannot write the bitmap directly with a single bit per pixel, but must convert each bit to a full n-bit color value that is then written.

Program to fill polygon

```
/**************************************Program :
A C++ program to implement different polygon filling
algorithms such as
                    * flood fill
                    * Edge fill
                    * Scan line fill
```

```
by making use of mouse to accept the polygon vertices
*******************************************/
#include<iostream.h>
#include<conio.h>
#include<dos.h>
#include<process.h>
#include<graphics.h>
#include<math.h>
#include<malloc.h>

struct node
{
    int x,y;
    node * next;
};

union REGS i,o;
struct node * home = NULL;
/*====================Class Definition.
============================*/
class polyfill
{
    int sx,sy,xx[10],yy[10];
    public:
        polyfill() { }
        initmouse();
        void showmouse();
        void push(int,int);
        void getmousepos(int*,int*,int*);
        void flood(int,int,int,int);
        void scanline(int[],int[],int,int);
        void edge(int[],int[],int,int);
};
/*============================
Member Name     :   initmouse().
Purpose         :   To initialize the mouse.
============================*/
polyfill :: initmouse()
{
    i.x.ax=0;
    int86(0x33,&i,&o);
    return(o.x.ax);
}
/*============================
Member Name     :   showmouse().
Purpose         :   To show the mouse.
============================*/
void polyfill :: showmouse()
{
    i.x.ax=1;
    int86(0x33,&i,&o);
}
/*============================
Member Name     : getmousepos().
Purpose         : To get the pixel clicked by the mouse.
============================*/
void polyfill :: getmousepos(int *button,int *x,int *y)
{
    i.x.ax=3;
    int86(0x33,&i,&o);
    *button=o.x.bx;
    *x=o.x.cx;
    *y=o.x.dx;
}
/*============================
Member Name  :  push().
Purpose      :  To maintain the stack.
============================*/
void polyfill :: push(int x,int y)
{
    struct node * newn;
    if(home==NULL)
    {
        newn=(struct node *)malloc(sizeof(struct node));
        newn->x=x;
        newn->y=y;
        newn->next=NULL;
        home=newn;
    }
    else
    {
        newn=(struct node *)malloc(sizeof(struct node));
        newn->x=x;
        newn->y=y;
        home->next=newn;
        home=newn;
    }
}
/*============================
    point, background color and new color
    Member Name  :  flood().
    Purpose      :  To implement the flood fill algorithm.
    parameters are :  seed point, background color and
                      new color
============================*/
void polyfill :: flood(int sx,int sy,int b,int col)
{
    int x=sx,y=sy;
    struct node * temp;
    push(x,y);
    temp=home;
    while(temp)
    {
```

```
        x=temp->x;
        y=temp->y;
        putpixel(x,y,col);
if(getpixel(x,y+1)!=15 && getpixel(x,y+1)!=col)
        {
            putpixel(x,y+1,col);
            push(x,y+1);
        }
if(getpixel(x,y-1)!=15 && getpixel(x,y-1)!=col)
        {
            putpixel(x,y-1,col);
            push(x,y-1);
        }
if(getpixel(x+1,y)!=15 && getpixel(x+1,y)!=col)
        {
            putpixel(x+1,y,col);
            push(x+1,y);
        }
if(getpixel(x-1,y)!=15 && getpixel(x-1,y)!=col)
        {
            putpixel(x-1,y,col);
            push(x-1,y);
        }
        struct node * temp1=temp;
        temp=temp->next;
        free(temp1);
        delay(1);
    }
    getch();
}
/* Like flood fill algorithm we can implement Boundary fill
algorithm also. Only difference is we have to check every
pixels color with boundary color, instead of background color.
Students are advised to implement boundary function by their
own */
/*============================
    Member Name     :   edge().
    Purpose         :   To implement the edge fill
                        algorithm.
    Parameters are  :   vertices array of polygon, new
                        color, no. of vertices
============================*/
void polyfill :: edge(int xx[10],int yy[10],int col,int n)
{
    float x1,x2,y1,y2,m,x;
    int i=0;
    while(i<n)
    {
        if(yy[i]<yy[i+1])
        {
            y1 = yy[i];
            x1 = xx[i];
            y2 = yy[i+1];
            x2 = xx[i+1];
        }
        else
        {
            y1=yy[i+1];
            x1=xx[i+1];
            y2=yy[i];
            x2=xx[i];
        }
        m=(float)(x2-x1)/(y2-y1);
        while(y1<y2)
        {
            y1++;
            x1=x1+m;
            x=x1;
            while(x<=640)
            {
                if(getpixel(x,y1)==0)
                    putpixel(x,y1,col);
                else
                    putpixel(x,y1,0);
                x++;
            }
            delay(10);
        }
        i++;
    }
    getch();
}
/* Like Edge fill algorithm we can implement fence fill
algorithm also. Students are advised to implement fence
function by their own. */
/*============================
Member Name     :   scanline().
Purpose         :   To implement the scanline algorithm.
Parameters are  :   vertices array of polygon, new color,
                    no. of vertices
============================*/
void
 polyfill :: scanline(int xx[10],int yy[10],int col,int n)
{
    int i,k,inter_x[50],temp,y,ymax=0,ymin=480;
    float m[50],dx,dy;
    for(i=0;i<n;i++)
    {
        if(yy[i]>=ymax) ymax=yy[i];
        if(yy[i]<=ymin) ymin=yy[i];

        dx=xx[i+1]-xx[i];
```

```
        dy=yy[i+1]-yy[i];
        if(dx==0) m[i]=0;
        if(dy==0) m[i]=1;

        if(dx!=0 && dy!=0)
        m[i]=(float)dx/dy;
    }
    int cnt;
    setcolor(col);

    for(y=ymax;y>=ymin;y--)
    {
        cnt=0;
        for(i=0;i<n;i++)
        {
if((yy[i]>y&&yy[i+1]<=y) || (yy[i]<=y&&yy[i+1]>y))
            {
                inter_x[cnt]=(xx[i]+(m[i]*(y-yy[i])));
                cnt++;
            }
        }
        for(k=0;k<cnt-1;k++)
        {
            for(i=0;i<cnt-1;i++)
            {
                if(inter_x[i]>inter_x[i+1])
                {
                    temp=inter_x[i];
                    inter_x[i]=inter_x[i+1];
                    inter_x[i+1]=temp;
                }
            }
        }
        for(i=0;i<cnt-1;i+=2)
        {
            line(inter_x[i],y,inter_x[i+1]+1,y);
            delay(10);
        }
    }
    getch();
}

/*============================
    .Main Function Definition.
============================*/
void main()
{
    polyfill p;
    int
col,n,xx[10],yy[10],i=0,x,y,button,gm,gd=DETECT,sy,sx,ch;
    cout << "Enter the algorith to follow : \n"
        << "\t1.Flood fill.\n\t2.Edge fill.\n\t3.Scan line.\n"
        << "Entered choice is : ";
    cin >> ch;
    cout << "\nEnter the color number : ";
    cin >> col;
    initgraph(&gd,&gm,"c:\\tcplus\\bgi");
    p.initmouse();
p.showmouse();
    do              // accept polygons vertices
    {               // till keyboard gets hit
        p.getmousepos(&button,&x,&y);
        if((button&1)==1)
        {
            delay(500);
            xx[i] = x;
            yy[i] = y;
            i++;
        }
    } while(!kbhit());
    xx[i]=xx[0];
    yy[i]=yy[0];
    n=i;                //n will be no. of vertices
    getch();
    for(int j=0;j<i;j++)
    {
line(xx[j],yy[j],xx[j+1],yy[j+1]);  // draw a polygon
    }
    do
    {
        p.getmousepos(&button,&x,&y);
        if((button&1)==1)
{           // accepting seed point for flood fill
            delay(500);
            sx = x;
            sy = y;
            break;
        }
    }while(1);
    int b = getbkcolor();
    if(ch==1)
        p.flood(sx,sy,b,col);
    else if(ch==2)
        p.edge(xx,yy,col,n);
    else if(ch==3)
        p.scanline(xx,yy,col,n);
    getch();
    closegraph();
}
/*============================*/
```