Algorithms: Line drawing algorithms-

DDA Line generation Algorithm

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

Examples:

Input: For line segment between (2, 2) and (6, 6) :we need (3, 3) (4, 4) and (5, 5) as our intermediatepoints.

Input: For line segment between (0, 2) and (0, 6) :we need (0, 3) (0, 4) and (0, 5) as our intermediatepoints

For using graphics functions, our system output screen is treated as a coordinate system where the coordinate of the top-left corner is (0, 0) and as we move down our y-ordinate increases and as we move right our x-ordinate increases for any point (x, y).

Now, for generating any line segment we need intermediate points and for calculating them we have can use a basic algorithm called DDA(Digital differential analyzer) line generating algorithm.

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

```
// calculate dx , dy
dx = X1 - X0;
dy = Y1 - Y0;
// Depending upon absolute value of dx & dy
// choose number of steps to put pixel as
// steps = abs(dx) > abs(dy) ? abs(dx) : abs(dy)
```

```
steps = abs(dx) > abs(dy) ? abs(dx) : abs(dy);
// calculate increment in x & y for each steps
Xinc = dx / (float) steps;
Yinc = dy / (float) steps;
// Put pixel for each step
X = X0;
Y = Y0;
for (int i = 0; i <= steps; i++)
{    putpixel (X,Y,WHITE);
   X += Xinc;    Y += Yinc;}
```

```
// C program for DDA line generation
#include<stdio.h>
#include<graphics.h>

//Function for finding absolute value
int abs (int n)
{
   return ( (n>0) ? n : ( n * (-1)));
}

//DDA Function for line generation
```

```
void DDA(int X0, int Y0, int X1, int Y1)
{
   // calculate dx & dy
   int dx = X1 - X0;
   int dy = Y1 - Y0;

   // calculate steps required for generating pixels
   int steps = abs(dx) > abs(dy) ? abs(dx) : abs(dy);

   // calculate increment in x & y for each steps
   float Xinc = dx / (float) steps;
   float Yinc = dy / (float) steps;

   // Put pixel for each step
   float X = X0;
   float Y = Y0;
   for (int i = 0; i <= steps; i++)
   {
      putpixel (X,Y,RED);  // put pixel at (X,Y)
      X += Xinc;           // increment in x at each step
      Y += Yinc;           // increment in y at each step
      delay(100);          // for visualization of line-
```

```
                        // generation step by step
    }
}

// Driver program
int main()
{
    int gd = DETECT, gm;

    // Initialize graphics function
    initgraph (&gd, &gm, "");

    int X0 = 2, Y0 = 2, X1 = 14, Y1 = 16;
    DDA(2, 2, 14, 16);
    return 0;
}
```

Bresenham's Line Generation Algorithm

Given coordinate of two points A(x1, y1) and B(x2,

y2). The task to find all the intermediate points required for drawing line AB on the computer screen of pixels. Note that every pixel has integer coordinates.

Examples:

Input : A(0,0), B(4,4)

Output : (0,0), (1,1), (2,2), (3,3), (4,4)

Input : A(0,0), B(4,2)

Output : (0,0), (1,0), (2,1), (3,1), (4,2)

Below are some assumptions to keep algorithm simple.

- We draw line from left to right.
- x1 < x2 and y1< y2
- Slope of the line is between 0 and 1. We draw a line from lower left to upper right.

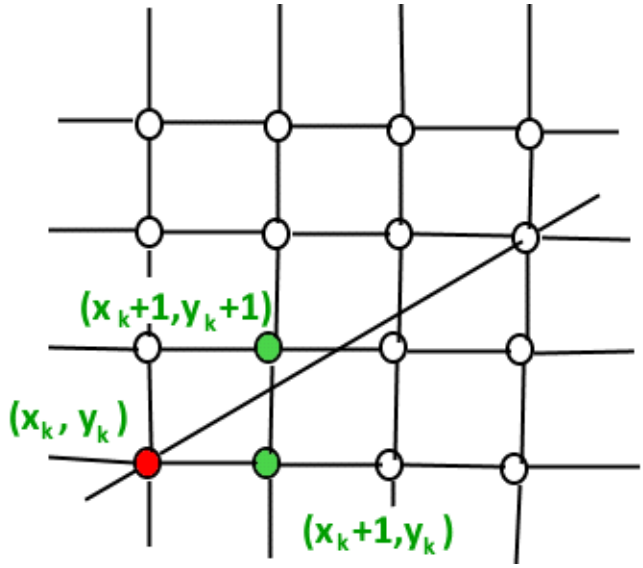
Let us understand the process by considering the naive way first.

```
// A naive way of drawing line
void naiveDrawLine(x1, x2, y1, y2)
{
m = (y2 - y1)/(x2 - x1)
for (x  = x1; x <= x2; x++)
   {
```

```
  // Assuming that the round function finds
      // closest integer to a given float.
   y = round(mx + c);
  print(x, y);
    }
}
```

Above algorithm works, but it is slow. The idea of Bresenham's algorithm is to avoid floating point multiplication and addition to compute mx + c, and then computing round value of (mx + c) in every step. In Bresenham's algorithm, we move across the x-axis in unit intervals.

- We always increase x by 1, and we choose about next y, whether we need to go to y+1 or remain on y. In other words, from any position (Xk, Yk) we need to choose between (Xk + 1, Yk) and (Xk + 1, Yk + 1).



- We would like to pick the y value (among Yk + 1 and Yk) corresponding to a point that is closer to the original line.

We need to a decision parameter to decide whether to pick Yk + 1 or Yk as next point. The idea is to keep track of slope error from previous increment to y. If the slope error becomes greater than 0.5, we know that the line has moved upwards one pixel, and that we must increment our y coordinate and readjust the error to represent the distance from the top of the new pixel – which is done by subtracting one from error.

// **Modifying the naive way to use a parameter**

// **to decide next y**.

```
void withDecisionParameter(x1, x2, y1, y2)
{
m = (y2 - y1)/(x2 - x1)
```

```
   slope_error = [Some Initial Value]
   for (x = x1, y = y1; x = 0.5)
    {          y++;          slope_error  -= 1.0;      }
}
```

How to avoid floating point arithmetic
The above algorithm still includes floating point arithmetic. To avoid floating point arithmetic, consider the value below value m.

m = (y2 – y1)/(x2 – x1)

We multiply both sides by (x2 – x1)

We also change slope_error to slope_error * (x2 – x1). To avoid comparison with 0.5, we further change it to slope_error * (x2 – x1) 2.

Also, it is generally preferred to compare with 0 than 1.

```
// Modifying the above algorithm to avoid floating
// point arithmetic and use comparison with 0.
void bresenham(x1, x2, y1, y2)
{
   m_new = 2 * (y2 - y1)
   slope_error_new = [Some Initial Value]
for (x = x1, y = y1; x = 0)
   {    y++;
```

```
 slope_error_new  -= 2 * (x2 - x1);
      }
}
```

The initial value of slope_error_new is 2*(y2 – y1) – (x2 – x1). Refer this for proof of this value

Below is the implementation of above algorithm.

```
// C++ program for Bresenham's Line Generation
// Assumptions :
// 1) Line is drawn from left to right.
// 2) x1 < x2 and y1 < y2
// 3) Slope of the line is between 0 and 1.
//    We draw a line from lower left to upper
//    right.
#include<bits/stdc++.h>
using namespace std;
// function for line generation
void bresenham(int x1, int y1, int x2, int y2)
{
  int m_new = 2 * (y2 - y1);
  int slope_error_new = m_new - (x2 - x1);
  for (int x = x1, y = y1; x <= x2; x++)
  {
```

```
        cout << "(" << x << "," << y << ")\n";

        // Add slope to increment angle formed
        slope_error_new += m_new;

        // Slope error reached limit, time to
        // increment y and update slope error.
        if (slope_error_new >= 0)
        {
            y++;
            slope_error_new  -= 2 * (x2 - x1);
        }
    }
}
// driver function
int main()
{
  int x1 = 3, y1 = 2, x2 = 15, y2 = 5;
  bresenham(x1, y1, x2, y2);
  return 0;
}
```

Output :(3,2)

(4,3)

(5,3)

(6,3)

(7,3)

(8,4)

(9,4)

(10,4)

(11,4)

(12,5)

(13,5)

(14,5)

(15,5)

The above explanation is to provides a rough idea behind the algorithm. For detailed explanation and proof, readers can refer below references.

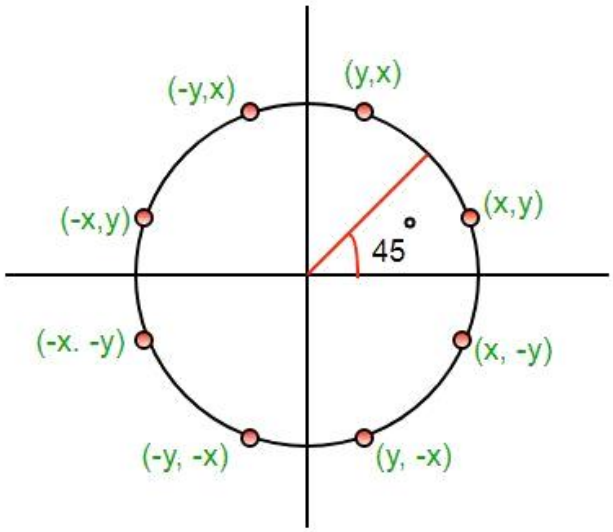
Circle:

Mid-Point Circle Drawing Algorithm

We need to plot the perimeter points of a circle whose center co-ordinates and radius are given using the Mid-Point Circle Drawing Algorithm.

We use the above algorithm to calculate all the perimeter points of the circle in the **first octant** and then print

them along with their mirror points in the other octants. This will work only because a circle is symmetric about it's centre.



The algorithm is very similar to the Mid-Point Line Generation Algorithm. Here, only the boundary condition is different.

For any given pixel (x, y), the next pixel to be plotted is either **(x, y+1)** or **(x-1, y+1)**. This can be decided by following the steps below.

- Find the mid-point **p** of the two possible pixels i.e (x-0.5, y+1)
- If **p** lies inside or on the circle perimeter, we plot the pixel (x, y+1), otherwise if it's outside we plot the pixel (x-1, y+1)

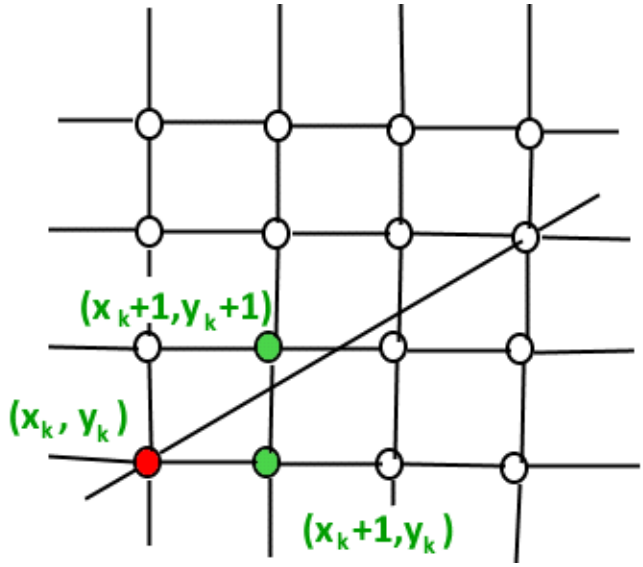
Boundary Condition : Whether the mid-point lies inside or outside the circle can be decided by using the formula:-

Given a circle centered at (0,0) and radius r and a point p(x,y)
*F(p) = x**2 + y2 – r2***

if F(p)<0, the point is inside the circle

F(p)=0, the point is on the perimeter

F(p)>0, the point is outside the circle



n our program we denote F(p) with P. The value of P is calculated at the mid-point of the two contending pixels i.e. (x-0.5, y+1). Each pixel is described with a subscript k.

Pk = (Xk — 0.5)2 + (yk + 1)2 – r2

Now,
xk+1 = xk or xk-1 , yk+1= yk +1

∴ Pk+1 = (xk+1 – 0.5)2 + (yk+1 +1)2 – r2
= (xk+1 – 0.5)2 + [(yk +1) + 1]2 – r2
= (xk+1 – 0.5)2 + (yk +1)2 + 2(yk + 1) + 1 – r2
= (xk+1 – 0.5)2 + [– (xk – 0.5)2 +(xk – 0.5)2] + (yk + 1)2 – r2 + (yk + 1) + 1

The first point to be plotted is (r, 0) on the x-axis. The initial value of P is calculated as follows:-

P1 = (r – 0.5)2 + (0+1)2 – r2
= 1.25 – r
= 1 -r (When rounded off)

Examples:

Input : Centre -> (0, 0), Radius -> 3

Output : (3, 0) (3, 0) (0, 3) (0, 3)
(3, 1) (-3, 1) (3, -1) (-3, -1)
(1, 3) (-1, 3) (1, -3) (-1, -3)
(2, 2) (-2, 2) (2, -2) (-2, -2)

(x1,y1) is initioally printed before the loop: (3,0) (3,0) (0,3) (0,3)

k	Pk	Xk	Yk	Pk+1	Xk+1	Yk+1	Output
1	-2	3	0	-1	3	1	(3,1) (-3,1) (3,-1) (-3,-3) (1,3) (-1,3) (1,-3) (-1,-3)
2	-1	3	1	2	2	2	(2,2) (-2,2) (2,-2) (-2,-2)
3	2	2	2				Break from loop

Input : Centre -> (4, 4), Radius -> 2

Output : (6, 4) (6, 4) (4, 6) (4, 6)
(6, 5) (2, 5) (6, 3) (2, 3)

(5, 6) (3, 6) (5, 2) (3, 2)

```
// C program for implementing
// Mid-Point Circle Drawing Algorithm
#include<stdio.h>

// Implementing Mid-Point Circle Drawing Algorithm
void midPointCircleDraw(int x_centre, int y_centre, int r)
{
    int x = r, y = 0;

    // Printing the initial point on the axes
    // after translation
    printf("(%d, %d) ", x + x_centre, y + y_centre);

    // When radius is zero only a single
    // point will be printed
    if (r > 0)
    {
        printf("(%d, %d) ", x + x_centre, -y + y_centre);
        printf("(%d, %d) ", y + x_centre, x + y_centre);
        printf("(%d, %d)\n", -y + x_centre, x + y_centre);
```

```
    }

    // Initialising the value of P
    int P = 1 - r;
    while (x > y)
    {
        y++;

        // Mid-point is inside or on the perimeter
        if (P <= 0)
            P = P + 2*y + 1;

        // Mid-point is outside the perimeter
        else
        {
            x--;
            P = P + 2*y - 2*x + 1;
        }

        // All the perimeter points have already been
printed
        if (x < y)
```

```
            break;

        // Printing the generated point and its reflection
        // in the other octants after translation
        printf("(%d, %d) ", x + x_centre, y + y_centre);
        printf("(%d, %d) ", -x + x_centre, y + y_centre);
        printf("(%d, %d) ", x + x_centre, -y + y_centre);
        printf("(%d, %d)\n", -x + x_centre, -y +
y_centre);

        // If the generated point is on the line x = y then
        // the perimeter points have already been printed
        if (x != y)
        {
            printf("(%d, %d) ", y + x_centre, x + y_centre);
            printf("(%d, %d) ", -y + x_centre, x +
y_centre);
            printf("(%d, %d) ", y + x_centre, -x +
y_centre);
            printf("(%d, %d)\n", -y + x_centre, -x +
y_centre);
        }
    }
```

```
}

// Driver code
int main()
{
    // To draw a circle of radius 3 centred at (0, 0)
    midPointCircleDraw(0, 0, 3);
    return 0;
}
```

Output:

```
(3, 0) (3, 0) (0, 3) (0, 3)
(3, 1) (-3, 1) (3, -1) (-3, -1)
(1, 3) (-1, 3) (1, -3) (-1, -3)
(2, 2) (-2, 2) (2, -2) (-2, -2)
```

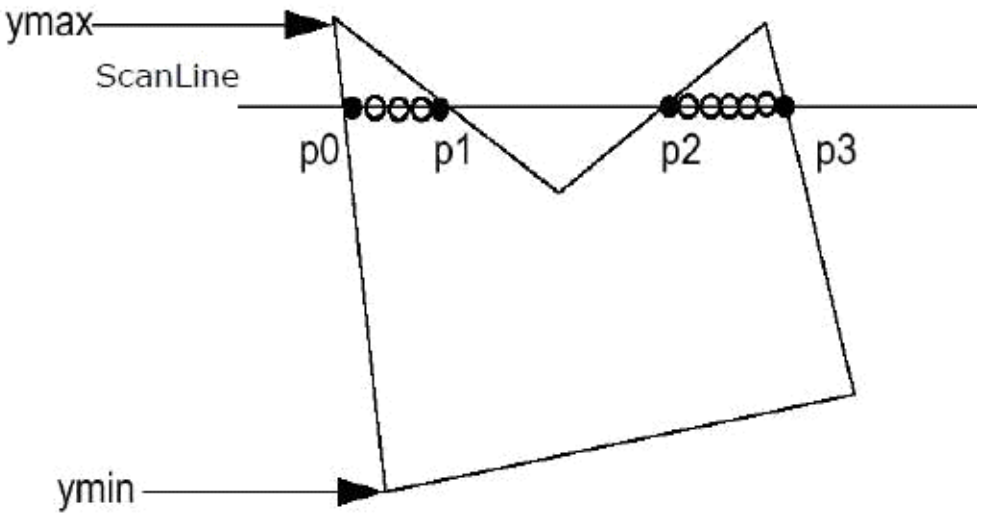
Polygon Fill Algorithum

Polygon is an ordered list of vertices as shown in the following figure. For filling polygons with particular colors, you need to determine the pixels falling on the border of the polygon and those which fall inside the polygon. In this chapter, we will see how we can fill polygons using different techniques.

Scan Line Algorithm

This algorithm works by intersecting scanline with polygon edges and fills the polygon between pairs of intersections. The following steps depict how this algorithm works.

Step 1 – Find out the Ymin and Ymax from the given polygon.



Step 2 – ScanLine intersects with each edge of the polygon from Ymin to Ymax. Name each intersection point of the polygon. As per the figure shown above, they are named as p0, p1, p2, p3.

Step 3 – Sort the intersection point in the increasing order of X coordinate i.e. p0,p1*p0,p1*, p1,p2*p1,p2*, and p2,p3*p2,p3*.

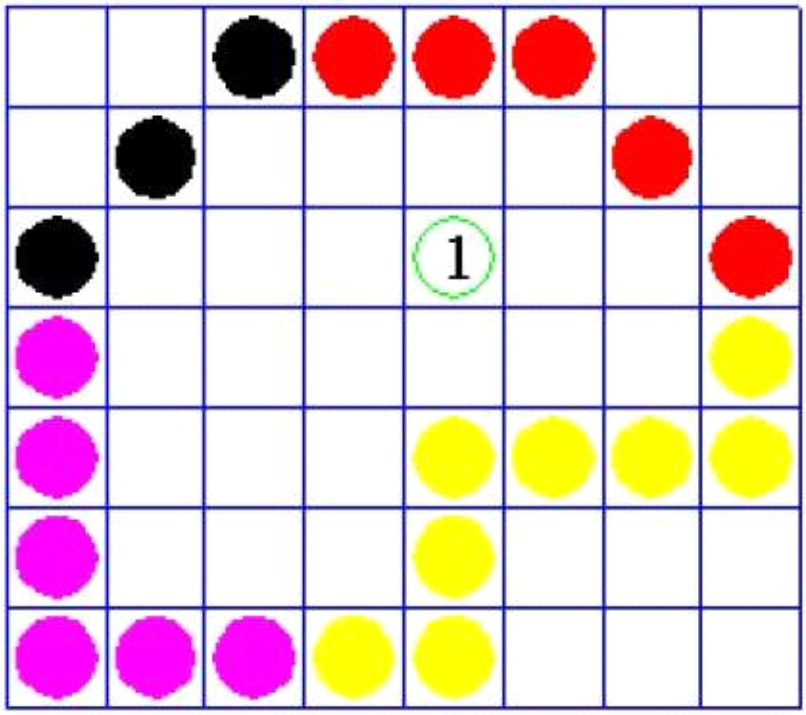
Step 4 – Fill all those pair of coordinates that are inside polygons and ignore the alternate pairs.

Flood Fill Algorithm

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

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

Once again, this algorithm relies on the Four-connect or Eight-connect method of filling in the pixels. But instead of looking for the boundary color, it is looking for all adjacent pixels that are a part of the interior.



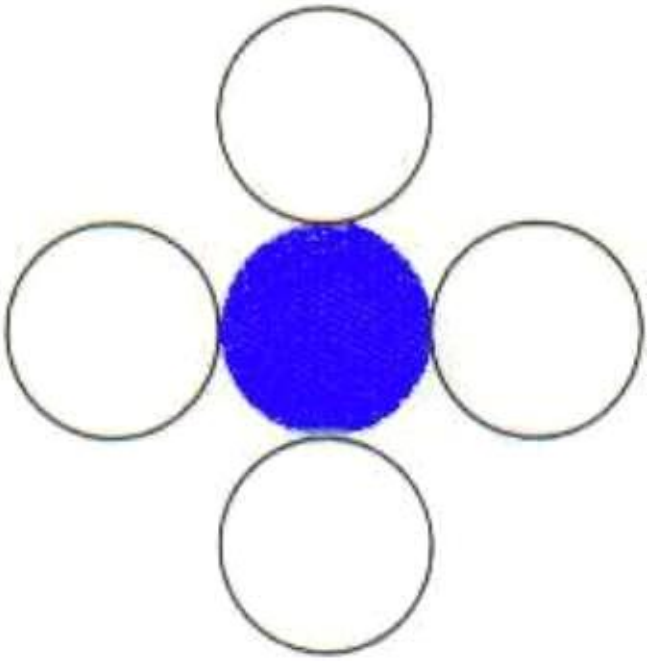
Boundary Fill Algorithm

The boundary fill algorithm works as its name. This algorithm picks a point inside an object and starts to fill until it hits the boundary of the object. The color of the boundary and the color that we fill should be different for this algorithm to work.

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

4-Connected Polygon

In this technique 4-connected pixels are used as shown in the figure. We are putting the pixels above, below, to the right, and to the left side of the current pixels and this process will continue until we find a boundary with different color.



Algorithm

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

Step 2 – Define the boundary values of the polygon.

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

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

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

setPixel(seedx, seedy, fcol)

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

FloodFill (seedx – 1, seedy, fcol, dcol)

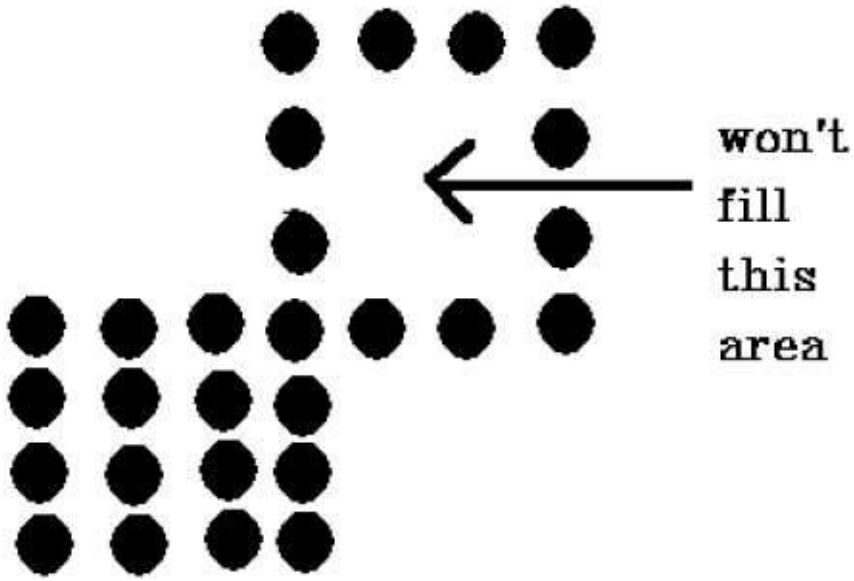
FloodFill (seedx + 1, seedy, fcol, dcol)

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

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

Step 6 – Exit

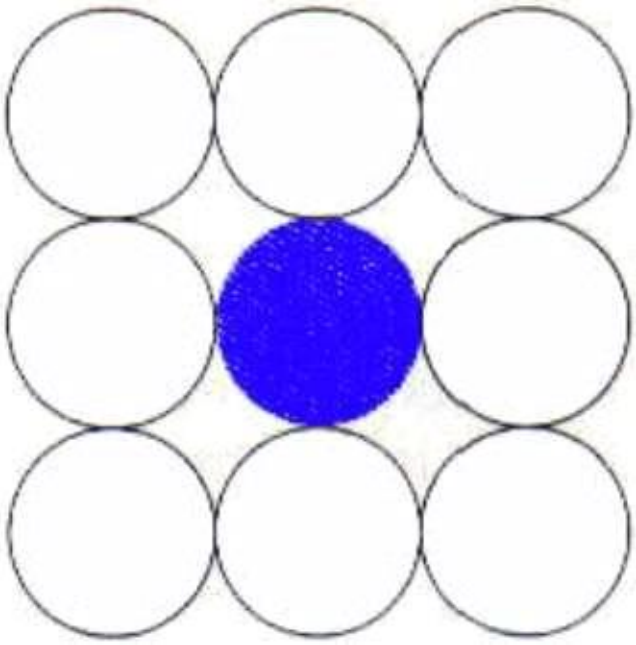
There is a problem with this technique. Consider the case as shown below where we tried to fill the entire region. Here, the image is filled only partially. In such cases, 4-connected pixels technique cannot be used.



8-Connected Polygon

In this technique 8-connected pixels are used as shown in the figure. We are putting pixels above, below, right and left side of the current pixels as we were doing in 4-connected technique.

In addition to this, we are also putting pixels in diagonals so that entire area of the current pixel is covered. This process will continue until we find a boundary with different color.



Algorithm

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

Step 2 – Define the boundary values of the polygon.

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

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

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

setPixel(seedx, seedy, fcol)

Step 5 – Recursively follow the procedure with four neighbourhood points

FloodFill (seedx – 1, seedy, fcol, dcol)

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

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

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

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

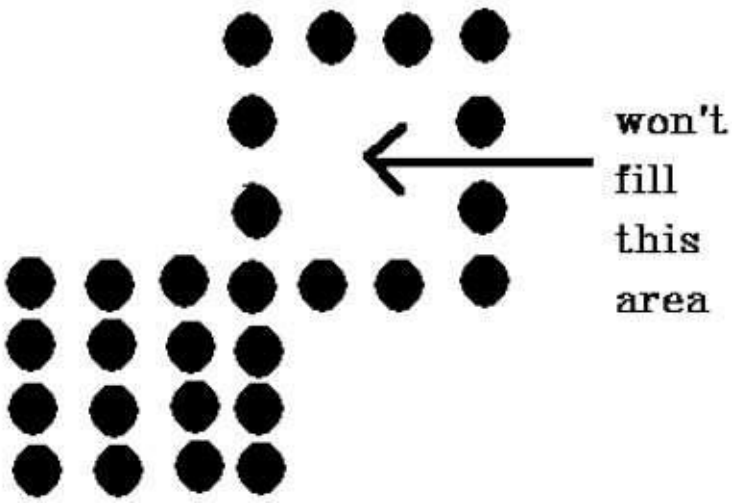
FloodFill (seedx + 1, seedy + 1, fcol, dcol)

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

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

Step 6 – Exit

The 4-connected pixel technique failed to fill the area as marked in the following figure which won't happen with the 8-connected technique.



Inside-outside Test

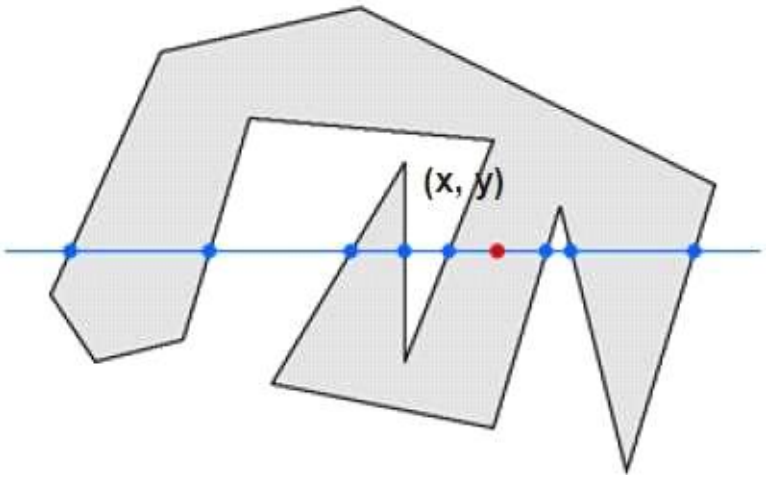
This method is also known as **counting number method**. While filling an object, we often need to identify whether particular point is inside the object or outside it. There are two methods by which we can identify whether particular point is inside an object or outside.

- Odd-Even Rule

- Nonzero winding number rule

Odd-Even Rule

In this technique, we will count the edge crossing along the line from any point x,y*x,y* to infinity. If the number of interactions is odd, then the point x,y*x,y* is an interior point; and if the number of interactions is even, then the point x,y*x,y* is an exterior point. The following example depicts this concept.

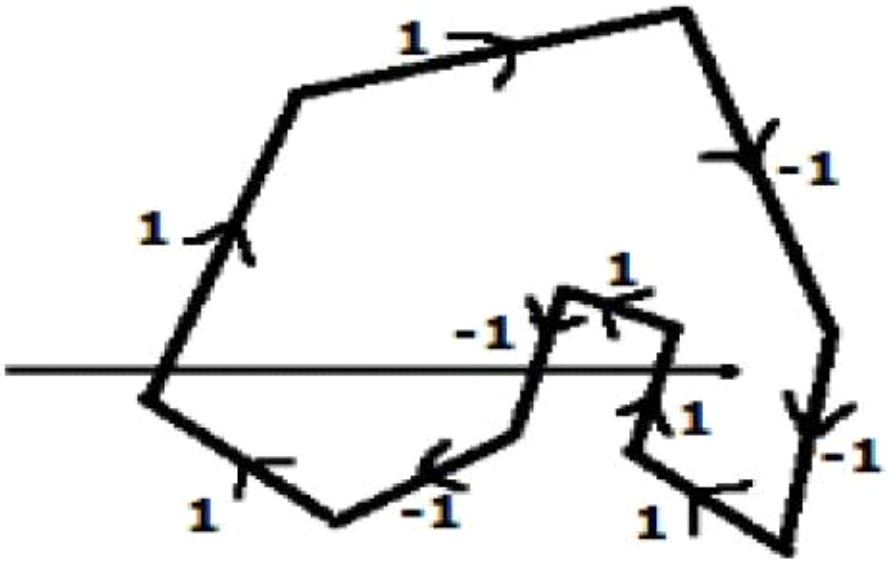


From the above figure, we can see that from the point x,y*x,y*, the number of interactions point on the left side is 5 and on the right side is 3. From both ends, the number of interaction points is odd, so the point is considered within the object.

Nonzero Winding Number Rule

This method is also used with the simple polygons to test the given point is interior or not. It can be simply understood with the help of a pin and a rubber band. Fix up the pin on one of the edge of the polygon and tie-up the rubber band in it and then stretch the rubber band along the edges of the polygon.

When all the edges of the polygon are covered by the rubber band, check out the pin which has been fixed up at the point to be test. If we find at least one wind at the point consider it within the polygon, else we can say that the point is not inside the polygon.



In another alternative method, give directions to all the edges of the polygon. Draw a scan line from the point to be test towards the left most of X direction.

- Give the value 1 to all the edges which are going to upward direction and all other -1 as direction values.
- Check the edge direction values from which the scan line is passing and sum up them.
- If the total sum of this direction value is non-zero, then this point to be tested is an **interior point,** otherwise it is an **exterior point**.
- In the above figure, we sum up the direction values from which the scan line is passing then the total is 1

– 1 + 1 = 1; which is non-zero. So the point is said to be an interior point.