

Module II

Line Drawing Algorithms

- Digital Differential Analyzer (DDA)
- Bresenham's Algorithm

DDA

- Line drawing is accomplished by calculating intermediate positions along the line path between two specified end positions.
- A straight line may be defined by two endpoints and an equation. Consider the two endpoints are described by (x_1, y_1) and (x_2, y_2) . The equation of the line is used to describe the x, y coordinates of all the points that lie between these two endpoints.

Using the equation of a straight line, $y = mx + b$

where $m = \Delta y / \Delta x$

and

$b =$ y intercept,

we can find values of y by incrementing $x = x_1$
to $x = x_2$.

A line connects two points. It is a basic element in graphics. To draw a line, you need two points between which you can draw a line. In the following three algorithms, we refer the one point of line as X_0, Y_0 and the second point of line as X_1, Y_1 .

A line connects two points. It is a basic element in graphics. To draw a line, you need two points between which you can draw a line. In the following three algorithms, we refer the one point of line as X_0, Y_0 and the second point of line as X_1, Y_1 .

DDA Algorithm

Digital Differential Analyzer (DDA) algorithm is the simple line generation algorithm which is explained step by step here.

Step 1: Get the input of two end points (X_0, Y_0) and (X_1, Y_1).

Step 2: Calculate the difference between two end points.

$$\begin{aligned} dx &= X_1 - X_0 \\ dy &= Y_1 - Y_0 \end{aligned}$$

Step 3: Based on the calculated difference in step-2, you need to identify the number of steps to put pixel. If $dx > dy$, then you need more steps in x coordinate; otherwise in y coordinate.

```
if(absolute(dx) > absolute(dy))
    Steps = absolute(dx);
else
    Steps = absolute(dy);
```

Step 4: Calculate the increment in x coordinate and y coordinate.

```
Xincrement = dx / (float) steps;
Yincrement = dy / (float) steps;
```

Step 5: Put the pixel by successfully incrementing x and y coordinates accordingly and complete the drawing of the line.

```
For(int v=0; v < Steps; v++)
```

```
{ x = x + Xincrement;
```

```
y = y + Yincrement; putpixel(Round(x), Round(y));
```

```
}
```

DDA Algorithm

```
void lineDDA( int xa, int ya, int xb, int yb )  
{  
    int dx = xb - xa, dy = yb - ya, steps, k;
```

- float xIncrement, yIncrement, x = xa, y = ya;
- if(abs(dx) > abs(dy))
- steps = abs(dx);
- else
- steps = abs(dy);
- xIncrement = (dx)/float(steps);
- yIncrement = (dy)/float(steps);
- setPixel(round(x), round(y));
- for(k=0; k<steps; k++)
- {
- x += xIncrement;
- y += yIncrement;
- setPixel(round(x), round(y));
- }
- }

Advantages and Disadvantages

ADV:

- Simple
- Faster

Disadvantages

The rounding operations and floating point arithmetic are still time consuming.

Bresenham's Line Drawing Algorithm

Given-

- Starting coordinates = (X_0, Y_0)
- Ending coordinates = (X_n, Y_n)

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

Step-01:

Calculate ΔX and ΔY from the given input.

These parameters are calculated as-

- $\Delta X = X_n - X_0$
- $\Delta Y = Y_n - Y_0$

Step-02:

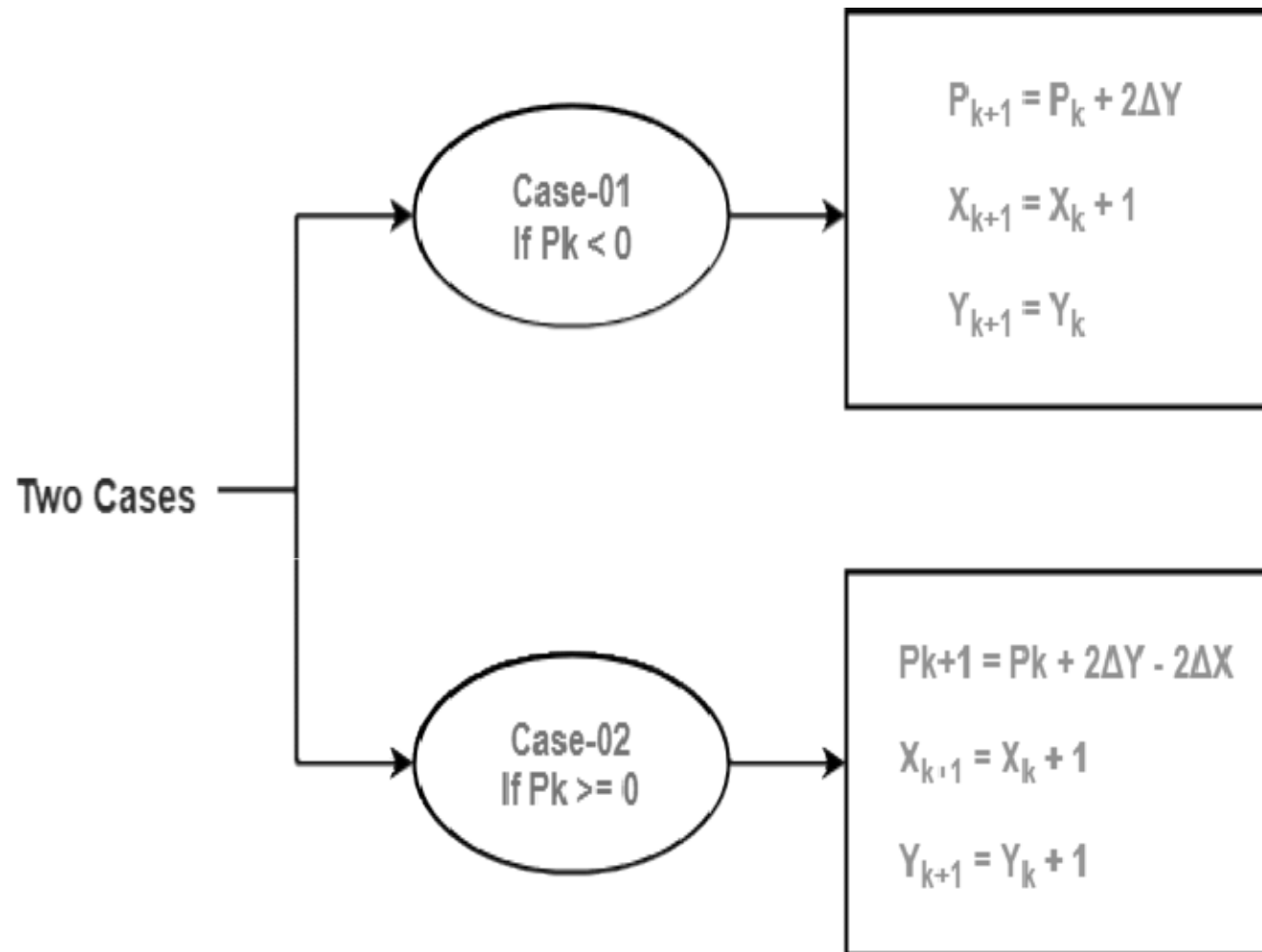
- Calculate the decision parameter P_k .

It is calculated as-

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

Step -03

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



Step-04:

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

Examples

- **Problem-01:**

Calculate the points between the starting coordinates (9, 18) and ending coordinates (14, 22).

Solution

Given-

- Starting coordinates = $(X_0, Y_0) = (9, 18)$
- Ending coordinates = $(X_n, Y_n) = (14, 22)$

Step-01:

Calculate ΔX and ΔY from the given input.

$$\Delta X = X_n - X_0 = 14 - 9 = 5$$

$$\Delta Y = Y_n - Y_0 = 22 - 18 = 4$$

Step-02:

Calculate the decision parameter.

$$\begin{aligned}P_k &= 2\Delta Y - \Delta X \\&= 2 \times 4 - 5 \\&= 3\end{aligned}$$

So, decision parameter $P_k = 3$

- As $P_k \geq 0$, so case-02 is satisfied.

Thus,

$$P_{k+1} = P_k + 2\Delta Y - 2\Delta X = 3 + (2 \times 4) - (2 \times 5) = 1$$

$$X_{k+1} = X_k + 1 = 9 + 1 = 10$$

$$Y_{k+1} = Y_k + 1 = 18 + 1 = 19$$

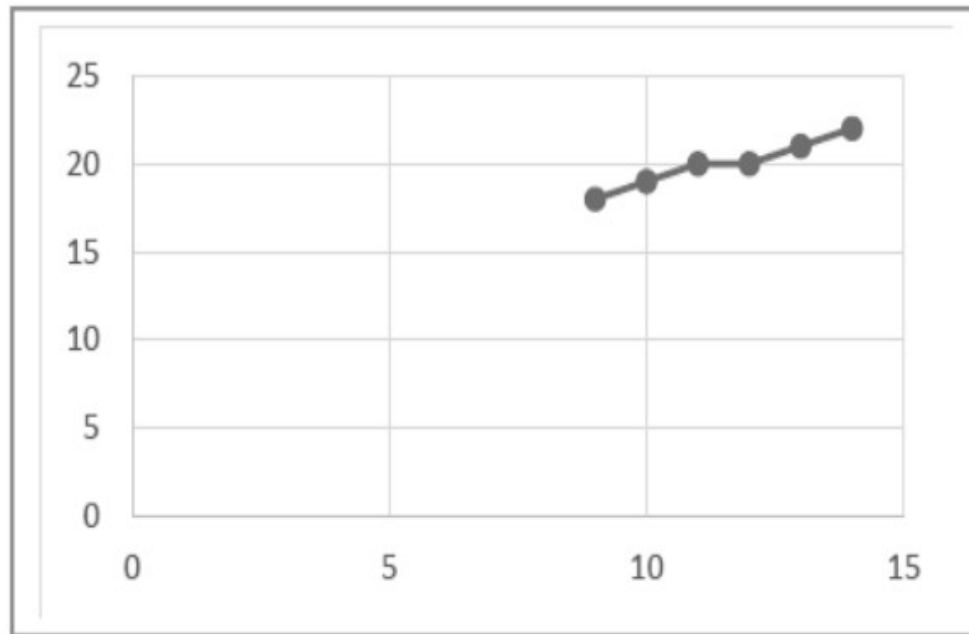
Similarly, Step-03 is executed until the end point is reached or number of iterations equals to 4 times.

$$(\text{Number of iterations} = \Delta X - 1 = 5 - 1 = 4)$$

Points to be plotted are:

| P_k | P_{k+1} | X_{k+1} | Y_{k+1} |
|-------|-----------|-----------|-----------|
| | | 9 | 18 |
| 3 | 1 | 10 | 19 |
| 1 | -1 | 11 | 20 |
| -1 | 7 | 12 | 20 |
| 7 | 5 | 13 | 21 |
| 5 | 3 | 14 | 22 |

Line plotting



Advantages of Bresenham Line Drawing Algorithm

- It is easy to implement.
- It is fast and incremental.
- It executes fast but less faster than DDA Algorithm.
- The points generated by this algorithm are more accurate than DDA Algorithm.
- It uses fixed points only.

The disadvantages of Bresenham Line Drawing Algorithm

- Though it improves the accuracy of generated points but still the resulted line is not smooth.
- This algorithm is for the basic line drawing.
- It can not handle diminishing jaggies.

MODULE II(2nd Half)

Scan Conversion-frame buffers – solid area scan conversion – polygon filling algorithms.

Scan Conversion

The process of representing continuous graphics object as a collection of discrete pixel is called scan conversion.

For example, line is defined by its two end points and the line equation, whereas the circle is defined by its radius, centre position and circle equation.

It is the responsibility of graphics system or the application program to convert each primitive from its geometric definition into a set of pixels that make up the primitive in image space. This conversion task is generally referred to as a scan conversion or rasterization.

In case of line, it can be possible that any pixel may have any floating value like (2.7,5) which will not be considered by the system.

Therefore, the process of making these coordinate according to system's assumption i.e (3,5) to plot the pixel is scan conversion.

Frame Buffer

A frame buffer (frame buffer, or sometimes frame store) is a portion of RAM containing a bitmap that drives a video display. It is a buffer containing a complete frame of data. Modern video cards contain frame buffer circuitry in their cores. This circuitry converts an in-memory bitmap into a video signal that can be displayed on a computer monitor.

The information in the buffer typically consists of color values for every pixel to be shown on the display. Color values are commonly stored in 1-bit binary (monochrome), 4-bit palettized, 8-bit palettized, 16-bit high color and 24-bit true color formats. An additional alpha channel is sometimes used to retain information about pixel transparency. The total amount of memory required for the frame buffer depends on the resolution of the output signal, and on the color depth or palette size.

Solid Area Scan Conversion

Solid area may represent lines of various thicknesses, colored geometric shapes appearing in diagrams, or facets of 3D objects. Generating a display of a solid area requires determining 3 properties of the area, *its mask, its shading rule, and its priority*.

The mask of an area is a representation that defines which pixels lie within the solid area. A mask values represented in the form of matrix, 0 indicates a pixel outside the area and 1 indicates a pixel within the area. *A shading rule* specifies how to compute the intensity of each pixel within the area. If an image should contain more than one solid area, *the priority* of each area determines which area is displayed when two or more areas overlap.

Geometric Representation of Areas

A solid area representation, it must be possible to determine the three essential attributes of area, its mask, priority, and shading. Mask representation is more complex because each representation involving a different scan conversion algorithm. A simple representation is a matrix of binary values. Some geometric figures have very simple representations and need only simple scan – conversion algorithms. Example scan converted with DDA. Another one is rectangle aligned with coordinate axes.

```
Procedure WriteRectangle (x, y, width, height, intensity: integer);
```

```
    Var i, j: integer;
```

```
Begin
```

```
    For j= y to y + height –1 do
```

```
        For i= x to x + width –1 do
```

```
            Setpixel (frame buffer, i, j, intensity);
```

```
End
```

Polygons have many shapes so we couldn't provide generalized procedure like this. All geometric representations describe objects by boundaries or outlines of the solid area. Scan conversion algorithms require a precise representation of the boundary in order to compute the area's mask. For polygon, we can represent the outline by an ordered list of vertices so that adjacent vertices in the list represent edge of the polygon.

Polygon Filling Algorithm

An ordered list of vertices forms a polygon. The pixels that fall on the border of the polygon are determined and the pixels that fall inside are determined in order to color the polygon.

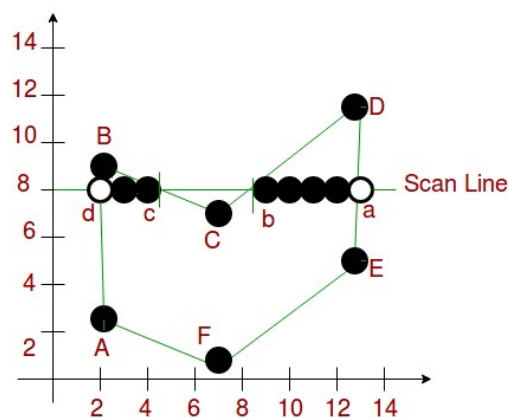
Scan-line Polygon

Figures on a computer screen can be drawn using polygons. To fill those figures with color, we need to develop some algorithm. There are two famous algorithms for this purpose: Boundary fill and Scan line fill algorithms.

Boundary filling requires a lot of processing and thus encounters few problems in real time. Thus the viable alternative is scan line filling as it is very robust in nature. This article discusses how to use Scan line filling algorithm to fill colors in an image.

Scanline Polygon filling Algorithm

Scan line filling is basically filling up of polygons using horizontal lines or scan lines. The purpose of the SLPF algorithm is to fill (color) the interior pixels of a polygon given only the vertices of the figure. To understand Scan line, think of the image being drawn by a single pen starting from bottom left, continuing to the right, plotting only points where there is a point present in the image, and when the line is complete, start from the next line and continue. This algorithm works by intersecting scan line with polygon edges and fills the polygon between pairs of intersections.

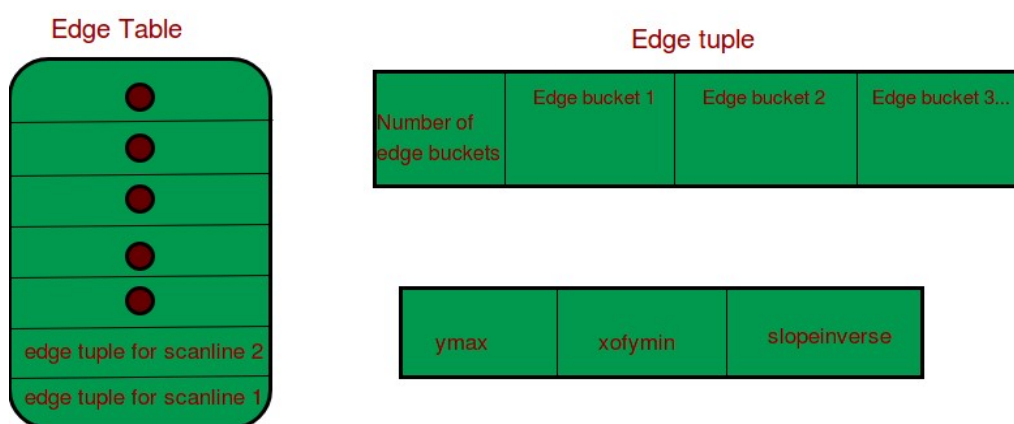


Special cases of polygon vertices:

1. If both lines intersecting at the vertex are on the same side of the scan line, consider it as two points.
2. If lines intersecting at the vertex are at opposite sides of the scan line, consider it as only one point.

Components of Polygon fill:

1. **Edge Buckets:** It contains an edge's information. The entries of edge bucket vary according to data structure you have used. In the example we are taking below, there are three edge buckets namely: ymax, xofymin, slopeinverse.
2. **Edge Table:** It consists of several edge lists -> holds all of the edges that compose the figure. When creating edges, the vertices of the edge need to be ordered from left to right and the edges are maintained in increasing ymin order. Filling is complete once all of the edges are removed from the ET.
3. **Active List:** It maintains the current edges being used to fill in the polygon. Edges are pushed into the AL from the Edge Table when an edge's yMin is equal to the current scan line being processed.
4. The Active List will be re-sorted after every pass.



ymax - max y - coordinate of edge

xofymin - x-coordinate of lowest edge point, updated at every scanline while scanline filling

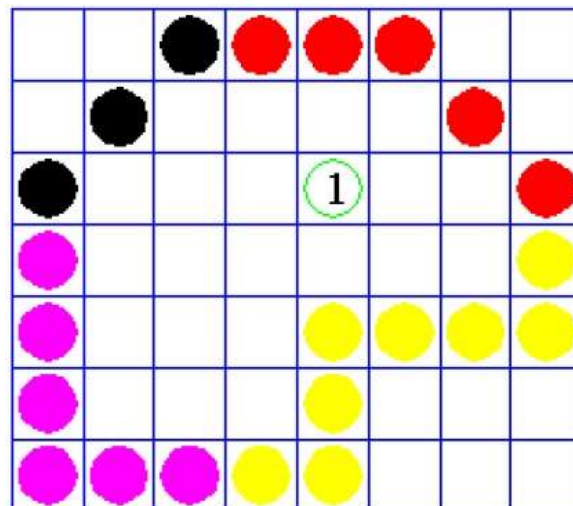
slopeinverse - inverse of edge slope (horizontal lines are not stored)

Flood Fill Algorithm

For some of the polygons, the area and boundary is filled by using different colours. A specific interior colour is used in such cases.

Fill colour option is used rather than on the object boundary. Fill colour replaces the interior colour. The algorithm is said to be complete when there are no left out pixels of the original interior colour.

The pixels are filled by using either Four-connect or Eight-connect method. The adjacent pixels are considered.



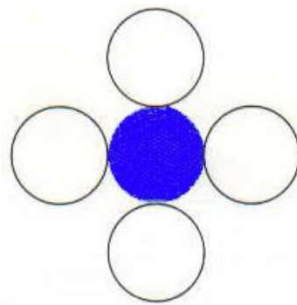
Boundary Fill Algorithm

As the name implies, this algorithm works. A point inside an object is picked and is filled until the boundary is hit by the object. This algorithm works only when the color of the boundary is different from the color that is used for filling.

For the complete object, the boundary color is assumed to be the same. Either 4-connected pixels or 8-connected pixels are used by this algorithm.

4-Connected Polygon

The pixels are used by placing them on four different sides of the current pixel till a different color boundary is identified in 4-connected polygon.



Algorithm

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

Step 2 – Define the boundary values of the polygon.

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

If $\text{getPixel}(x, y) = \text{dcol}$ then repeat step 4 and 5

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

$\text{setPixel}(\text{seedx}, \text{seedy}, \text{fcol})$

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

$\text{FloodFill}(\text{seedx} - 1, \text{seedy}, \text{fcol}, \text{dcol})$

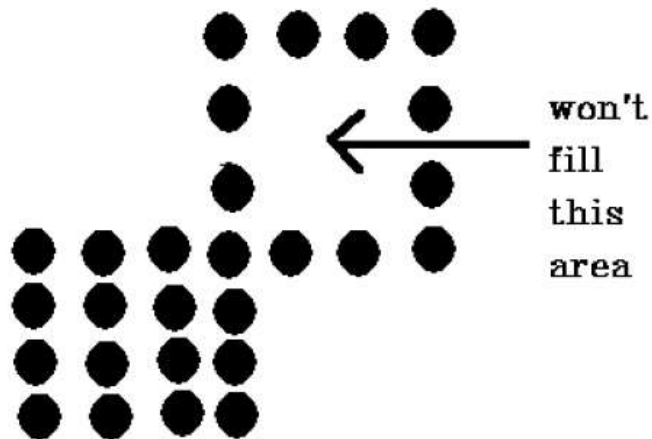
$\text{FloodFill}(\text{seedx} + 1, \text{seedy}, \text{fcol}, \text{dcol})$

$\text{FloodFill}(\text{seedx}, \text{seedy} - 1, \text{fcol}, \text{dcol})$

$\text{FloodFill}(\text{seedx}, \text{seedy} + 1, \text{fcol}, \text{dcol})$

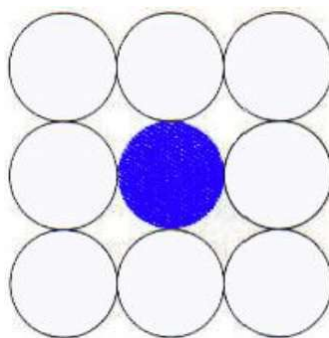
Step 6 – Exit

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

8-connected pixels are used in this technique by adding the pixels above, below, right and left side of the current pixels. In addition to this, the pixels are fixed diagonally such that the complete area of the current pixel is covered. The process is continued till a boundary is found with different color.



Algorithm

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

Step 2 – Define the boundary values of the polygon.

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

If $\text{getPixel}(x,y) = \text{dcol}$ then repeat step 4 and 5

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

$\text{setPixel}(\text{seedx}, \text{seedy}, \text{fcol})$

Step 5 – With the four neighbourhood points, follow the procedure:

$\text{FloodFill}(\text{seedx} - 1, \text{seedy}, \text{fcol}, \text{dcol})$

$\text{FloodFill}(\text{seedx} + 1, \text{seedy}, \text{fcol}, \text{dcol})$

$\text{FloodFill}(\text{seedx}, \text{seedy} - 1, \text{fcol}, \text{dcol})$

$\text{FloodFill}(\text{seedx}, \text{seedy} + 1, \text{fcol}, \text{dcol})$

$\text{FloodFill}(\text{seedx} - 1, \text{seedy} + 1, \text{fcol}, \text{dcol})$

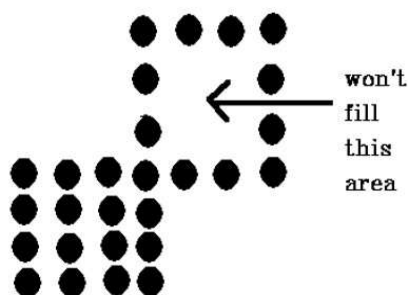
$\text{FloodFill}(\text{seedx} + 1, \text{seedy} + 1, \text{fcol}, \text{dcol})$

$\text{FloodFill}(\text{seedx} + 1, \text{seedy} - 1, \text{fcol}, \text{dcol})$

$\text{FloodFill}(\text{seedx} - 1, \text{seedy} - 1, \text{fcol}, \text{dcol})$

Step 6 – Exit

The area marked in the figure, which 4-connected pixel technique failed to fill is filled by the 8-connected technique.



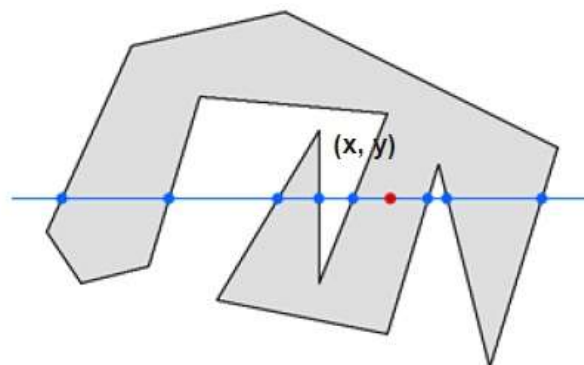
Inside-outside Test

When an object is filled, it is essential to identify whether the specific point is inside the object or outside the object. This is done by Inside-outside test also known as counting number method. An object can be identified whether inside or outside by two methods:

- Odd-Even Rule
- Nonzero winding number rule

Odd-Even Rule

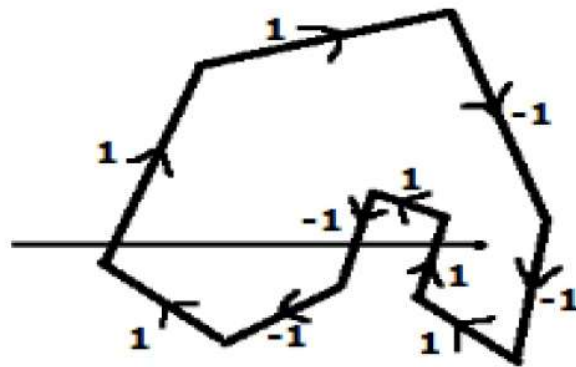
The edge that crosses along the line from the point (x,y) to infinity is counted. In case of odd interactions, the point (x,y) is an interior point and in case of even interactions, the point (x,y) is an exterior point. This concept is depicted by the following example:



It is observed from the above point (x,y) , that the number of intersection point on the left side is 5 and on the right side is 3. The point is considered within the object as the number of intersection points is considered as odd.

Nonzero Winding Number Rule

In order to test whether the point is interior or not, simple polygons are used which can be understood by using a pin and a rubber band. Pin is fixed on one of the edges of polygon and the rubber band is tied to it, and by stretching the rubber band along with the edges of the polygon.



Alternatively, directions can be given to all the edges of the polygon. A scan line can be drawn from the point to be tested towards the left most direction of X direction.

- To all the edges going to upward direction the value of 1 is given and for other -1 is assigned as direction values.
- The edge direction values are checked from which the scan line is passing and sum up them.
- The point that is to be tested is an interior point, if the total sum of the direction value is non-zero, or else it is an exterior point.
- The direction values from which the scan line is passed is summed up in the figure above, and then the total would be $1 - 1 + 1 = 1$, which is non-zero and hence the point is an interior point.