Line Drawing Algorithms

1. DDA (Digital Differential Analyzer) Algorithm
2. Bresenham's Algorithm

DDA

Line drawing is accomplished by calculating intermediate positions along the line path between two specified end position

eg:- (3, 4) (6, 10)

(x_1, y_1) (x_2, y_2)

Two end points

calculate intermediate points b/w two end points.

equation of a straight line $y = mx + b$

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

b = y intercept.

DDA algorithm

1. Get the i/p of two end points (x_1, y_1) and (x_2, y_2)

2. Calculate

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

$dy (\Delta y) = y_2 - y_1$

3. if (abs(dx) > abs(dy)

steps = abs(dx)

else

steps = abs(dy)

4. calculate xincrement & yincrement

$x_{inc} = dx / (float)\ steps.$

$y_{inc} = dy / (float)\ steps.$

5. $x = x_1 \quad y = y_1$

setpixel (round(x_1), round(y))

for (k = 0; k < steps; k++)

{

x = x + xinc

y = y + yinc

setpixel (round(x), round(y))

}

Adv

Simple

Faster

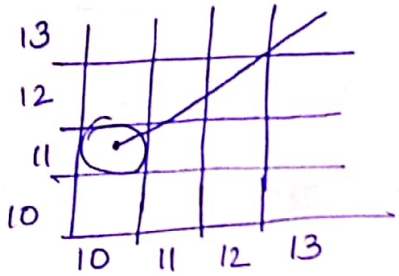
Disadvantages

rounding operations and floating arithmetic are still time consuming.

Bresenham's line drawing algorithm

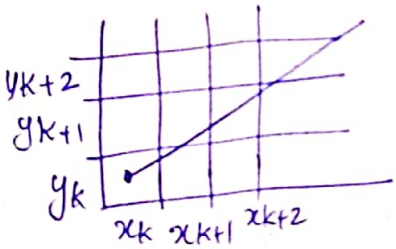
* Accurate and efficient
* uses only incremental integer calculation

The method is described for a line segment with a positive slop less than one.



Decide what is the next pixel position

(11, 11) or (11, 12)

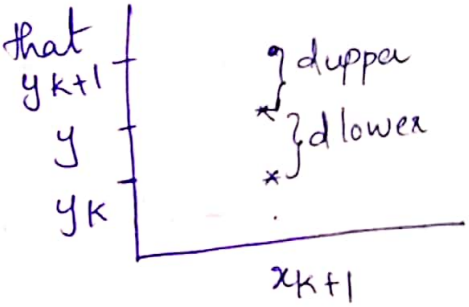


next pixel position is

(x_{k+1}, y_k) or (x_{k+1}, y_{k+1})

Determined using a decision parameter

for that



$$y = mx + b$$

$$= m(x_k + 1) + b$$

$$d_{lower} = y - y_k$$

$$= m(x_k + 1) + b - y_k$$

$d_{upper} = (y_k + 1) - y$

$= y_k + 1 - m(x_k + 1) - b$

$d_{lower} - d_{upper} = 2m(x_k + 1) + 2y_k + 2b - 1$

$m = \frac{\Delta x}{\Delta y}$

Decision parameter $P_k = \Delta x (d_{lower} - d_{upper})$

$P_k = 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c$

c is constance $c = 2\Delta y + \Delta x (2b - 1)$

* when P_k is negative, we can plot the lower pixel.
* when P_k is +ve, we can plot the upper pixel.

We can obtain the values of successive decesion parameter using incremental integer calculations.

At step $k+1$,

$P_{k+1} = 2\Delta y x_{k+1} - 2\Delta x y_{k+1} + c$

$P_{k+1} - P_k = 2\Delta y (x_{k+1} - x_k) - 2\Delta x (y_{k+1} - y_k) + c$

Take $x_{k+1} = x_k + 1$

So $P_{k+1} = P_k + 2\Delta y (x_k + 1 - x_k) - 2\Delta x (y_{k+1} - y_k) + c$

$$\boxed{P_{k+1} = P_k + 2\Delta y - 2\Delta x (y_{k+1} - y_k) + c}$$

when $P_k > 0$ we plot the upper pixel y_{k+1}

i.e $y_{k+1} = y_k + 1$

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

when $P_k < 0$ we plot the lower pixel, y_k

i.e $y_{k+1} = y_k$

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

Initial parameter $P_0 = 2\Delta y - \Delta x$

Bresenham's Line Drawing Algorithm

1. Input the two endpoints and store the left end point in (x_0, y_0)
2. Load (x_0, y_0) into frame buffer that is plot the first point
3. Calculate constants Δx, Δy, $2\Delta y$, $2\Delta y - 2\Delta x$ and find P_0

$$P_0 = 2\Delta y - \Delta x$$

4. At each x_k along the line, starting at $k=0$,

If $p_k < 0$ the next point to plot is

(x_{k+1}, y_k) and

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

otherwise next point to plot is

(x_{k+1}, y_{k+1}) and

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

5: Repeat step 4 $\Delta x - 1$ times

Example

Draw a line with end points $(20, 10)$ and $(30, 18)$

$\Delta x = 30 - 20 = 10$

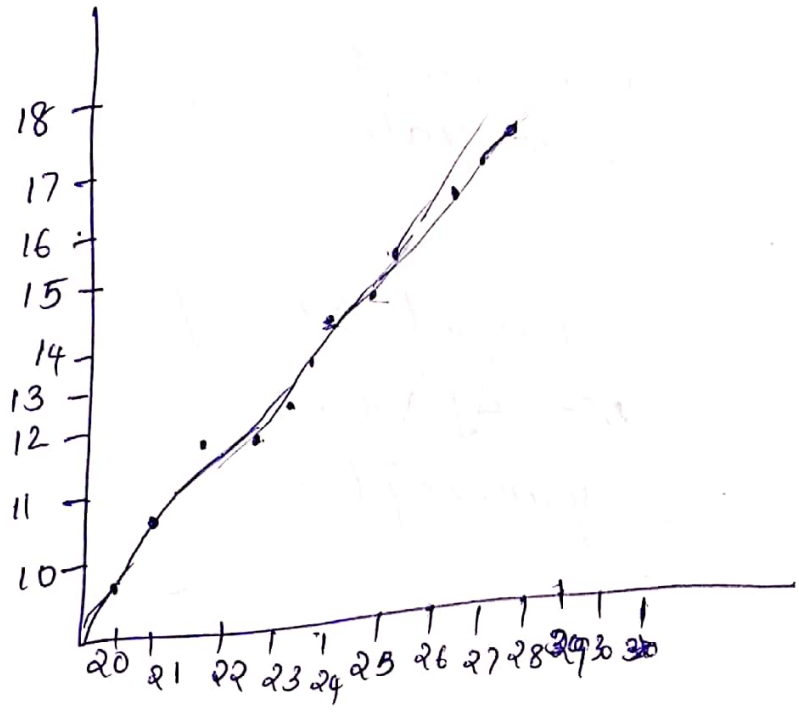
$\Delta y = 18 - 10 = 8$

$p_0 = 2\Delta y - \Delta x = 16 - 10 = 6$

$2\Delta y = 16$ & $2\Delta y - 2\Delta x = -4$

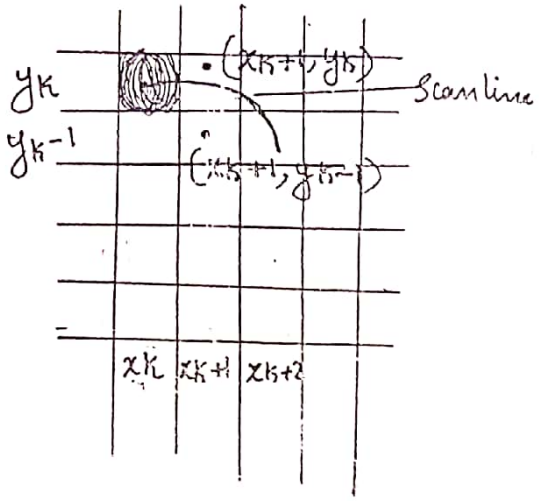
plot the initial position at $(20, 10)$ then

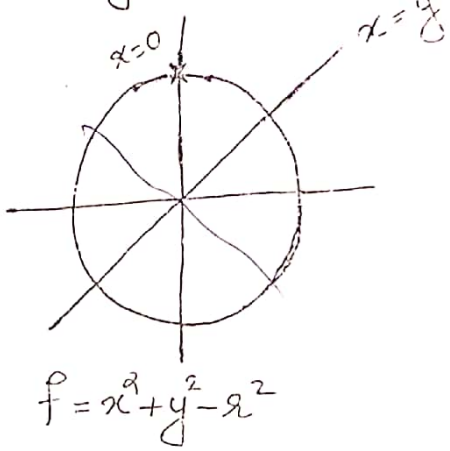
k	p_k	(x_{k+1}, y_{k+1})
0	6	(21, 11)
1	2	(22, 12)
2	-2	(23, 12)
3	14	(24, 13)
4	10	(25, 14)
5	6	(26, 15)
6	2	(27, 16)
7	-2	(28, 16)
8	14	(29, 17)
9	10	(30, 18)



Bresenhams circle drawing algorithm.

- A circle is defined as the set of points that are all at a given distance 'r' from a center position (x_c, y_c).
- The general eqt: of a circle is $x^2+y^2 = r^2$.





$f = x^2+y^2-r^2$

- According to Bresenhams algm, we have to find the pixel position closest to the scan line.
- Here the possible pixel positions closest to the scan line are (x_k+1, y_k) and (x_k+1, y_k-1). From these, we have to find which one is more close to the scan line.

$$d_1 = (x_k+1)^2+(y_k)^2-r^2$$
$$d_2 = (x_k+1)^2+(y_k-1)^2-r^2$$

- Here the value of d_1 is always positive since the point is outside the boundary.
- The value of d_2 is negative since the point is inside the boundary.

we know that $p_k = d_1 - d_2$.

ie $P_K = d_1 - (-d_2)$

$= d_1 + d_2$.

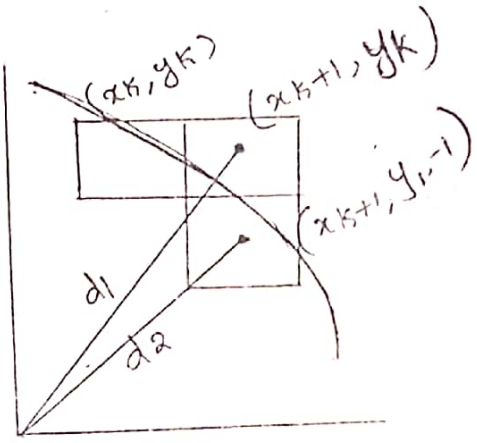
$P_K = (x_K+1)^2 + y_K^2 - r^2 + (x_K+1)^2 + (y_K-1)^2 - r^2$

$= 2(x_K+1)^2 - 2r^2 + y_K^2 + y_K^2 - 2y_K + 1$

$= 2(x_K+1)^2 + y_K^2 + (y_K-1)^2 - 2r^2$

By taking the absolute values of d_1 & d_2. P_K can be found

Here P_K is negative when $|d_1| < |d_2|$, then we plot the upper pixel (x_K+1, y_K).



P_K is positive when $|d_2| < |d_1|$, then we plot the lower pixel (x_K+1, y_K-1)

To find the next pixel

$P_{K+1} = 2(x_{K+1}+1)^2 + (y_{K+1})^2 + (y_{K+1}-1)^2 - 2r^2$

$P_{K+1} - P_K = 2(x_{K+1}+1)^2 + (y_{K+1})^2 + (y_{K+1}-1)^2 - 2r^2 - 2(x_K+1)^2 - y_K^2 - (y_K-1)^2 + 2r^2$

Take $x_{K+1} = x_K + 1$

$\therefore P_{K+1} - P_K = 2(x_K+2)^2 + (y_{K+1})^2 + (y_{K+1}-1)^2 - 2(x_K+1)^2 - y_K^2 - (y_K-1)^2$

$= 2(x_K^2 + 4x_K + 4) + (y_{K+1})^2 + (y_{K+1})^2 - 2y_K - 2(x_K^2 + 2x_K + 1) - y_K^2 - y_K^2 - 2y_K - 1$

$$= 2x_k^2 + 8x_k + 8 - 2x_k^2 - 4x_k - 2 + 2(y_{k+1})^2 - (2y_k)^2 + 2y_k - 2y_{k+1}$$

$$= 4x_k + 2(y_{k+1}^2 - y_k^2) - 2(y_{k+1} - y_k) + 6$$

$$\therefore p_{k+1} = p_k + 4x_k + 2(y_{k+1}^2 - y_k^2) - 2(y_{k+1} - y_k) + 6$$

If $p_k < 0$, (x_{k+1}, y_k), so we have to change $y_{k+1} = y_k$

$$\therefore p_{k+1} = p_k + 4x_k + 6$$

If $p_k > 0$, then plot the pixel $(x_{k+1}, y_k - 1)$, so we have to change $y_k + 1 = y_k - 1$.

$$\therefore p_{k+1} = p_k + 4x_k + 2((y_k - 1)^2 - (y_k)^2) - 2((y_k - 1) - (y_k)) + 6$$

$$= p_k + 4x_k + 2(y_k^2 - 2y_k + 1 - y_k^2) - 2(y_k - 1 - y_k) + 6$$

$$= p_k + 4x_k + -4y_k + 2 + 2 + 6$$

$$= p_k + 4(x_k - y_k) + 10$$

To find the initial decision parameter.

Consider (x_k, y_k) as (x_0, y_0) $(x_0, y_0) = (0, r)$

Substitute in p_k (x_0, y_0), then we get.

$$p_0 = 2(0+1)^2 + r^2 + (r-1)^2 - 2r^2$$

$$= 2 + r^2 + r^2 - 2r + 1 - 2r^2$$

$$= 3 - 2r$$

Bresenhams Circle Algorithm.

(1) Input radius 'r' and circle centre (x_c, y_c) and obtain the first point on the circumference of a circle centered on the origin as $(x_0, y_0) = (0, r)$

(2) Calculate initial value of the decision parameter

$$p_0 = 3 - 2r$$

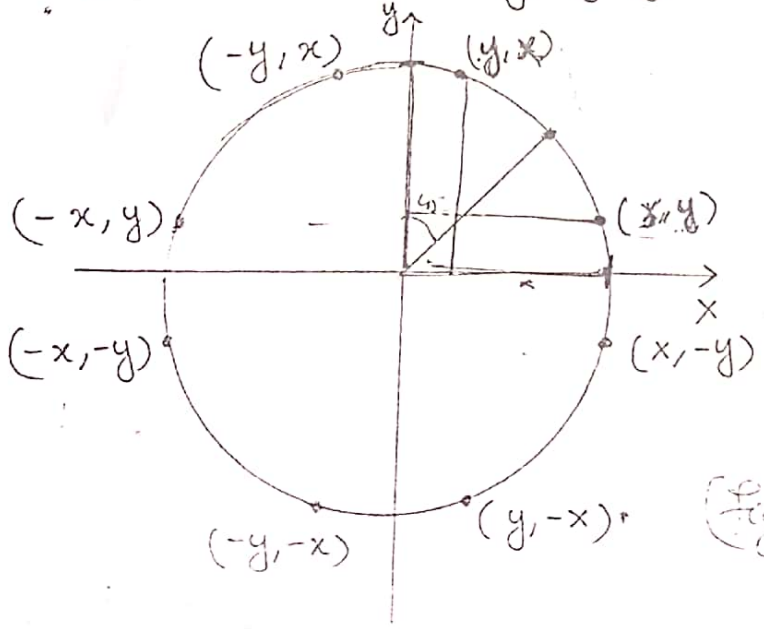
(3) At each x_k position starting at $k = 0$, perform the following test:

(a) if $p_k < 0$, the next point along the circle centered $(0,0)$ is (x_{k+1}, y_k) and $p_{k+1} = p_k + 4x_k + 6$

(b) Otherwise, the next point along the circle is (x_{k+1}, y_{k-1}) and $p_{k+1} = p_k + 4(x_k - y_k) + 10$.

(4) Determine the symmetric points in the other 7 oct

(5) Move each calculated pixel position (x, y) on to circular path centered on (x_c, y_c) and plot the co-ordinate values $x = x + x_c$ and $y = y + y_c$.

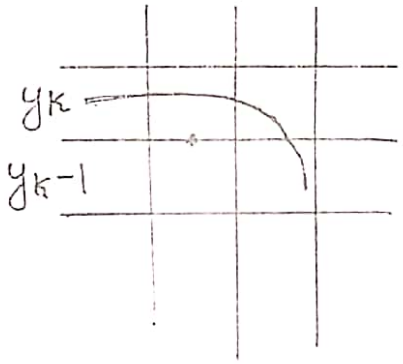


(fig a)

- The above (fig: a) shows the eight way Symmetry of a circle.
- Calculation of a circle point (x, y) in one octant yields the circle points shown for the other seven octants.
- ie In all circle algorithms, divide the circle into 8 parts and find out the pixel position of one part since it is an eight way symmetry.
- All the other positions have the same pixel positions only change is for values of coordinates in 4 quadrant.

Midpoint circle Algorithms. (Study explanation from text page no: 118)

- Here we are taking the midpoint of the pixel positions.



- To apply the midpoint method, we define a circle function as,

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

- Any point (x, y) on the boundary of the circle with radius 'r' satisfies the equation $f_{circle}(x, y) = 0$

$$f_{circle}(x,y) \begin{cases} <0, & \text{if } (x,y) \text{ is inside the circle boundary} \\ =0, & \text{if } (x,y) \text{ is on the circle boundary} \\ >0, & \text{if } (x,y) \text{ is outside the circle boundary.} \end{cases}$$

Midpoint circle algorithm.

(1) Input radius r and circle center (x_c, y_c) and obtain the first point on the circumference of a circle centered on the origin as, $(x_0, y_0) = (0, r)$

2. Calculate the initial value of the decision parameter as $p_0 = 5/4 - r$

3. At each x_k position, starting at $k=0$, perform the following test.

(a) If $p_k < 0$, the next point along the circle centered on $(0,0)$ is (x_{k+1}, y_k) and $p_{k+1} = p_k + 2x_{k+1} + 1$

(b) Otherwise, the next point along the circle is (x_k+1, y_k-1) and $p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$

where $2x_{k+1} = 2x_k + 2$ and

$2y_{k+1} = 2y_k - 2$

4. Determine Symmetry points in the other seven octants

5. Move each calculated pixel position (x, y) on the circular path centered on (x_c, y_c) and plot the co-ordinate values: $x = x + x_c$ and $y = y + y_c$.

6. Repeat Steps 3 through 5 until $x \geq y$.

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 aliened 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. Here we are discussing how to use Scan line filling algorithm to fill colors in an image.

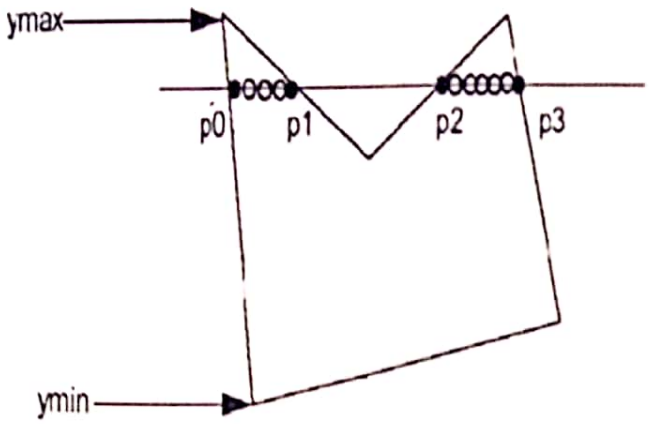
Scanline Polygon filling Algorithm

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.

The following steps depict how this algorithm works.

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

Step 2: Scan Line intersects with each edge of the polygon from Ymin to Ymax. Name each intersection point of the polygon. As per the figure shown below, 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), (p1, p2), and (p2, p3).

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

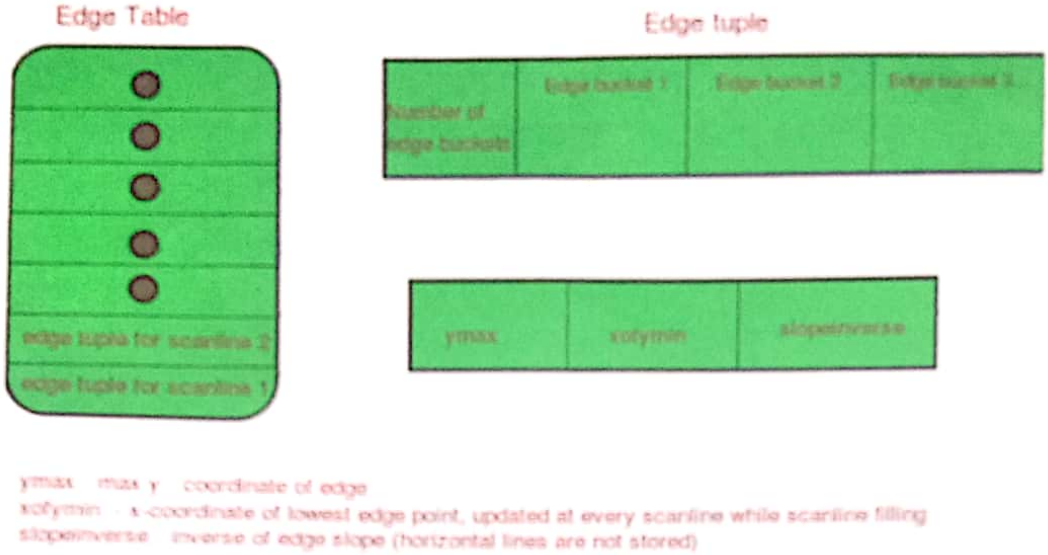
Special cases of polygon vertices:

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.

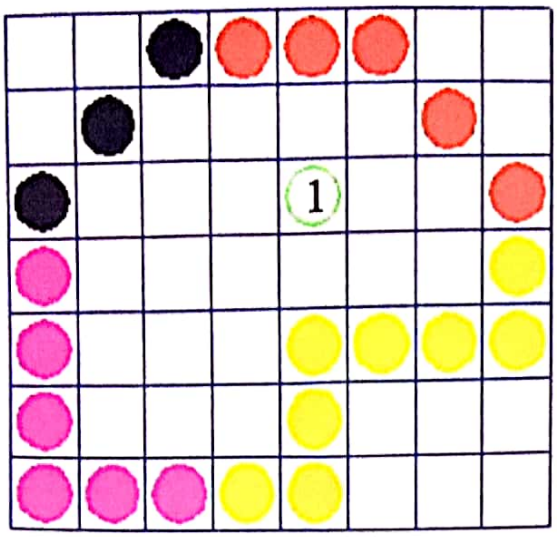


Flood Fill Algorithm

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

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

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



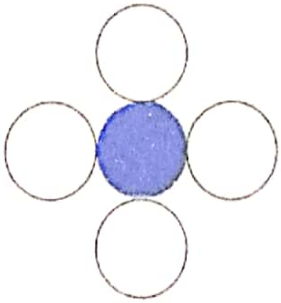
Boundary Fill Algorithm

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 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)

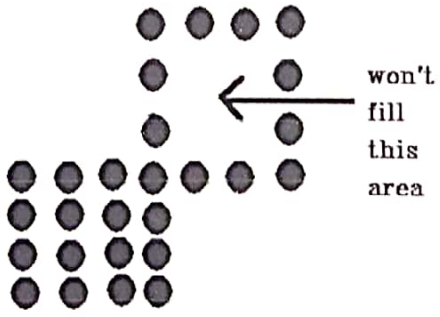
FloodFill (seedx + 1, seedy, fcol, dcol)

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

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

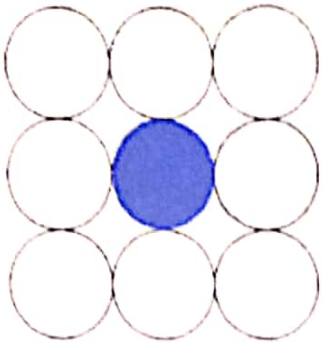
Step 6 – Exit

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

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 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 – With the four neighbourhood points, follow the procedure:

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

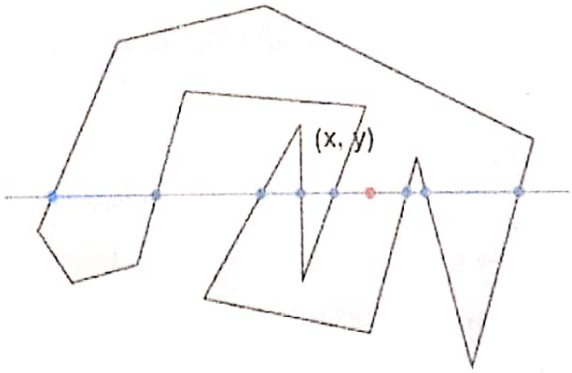
Inside-outside Test

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

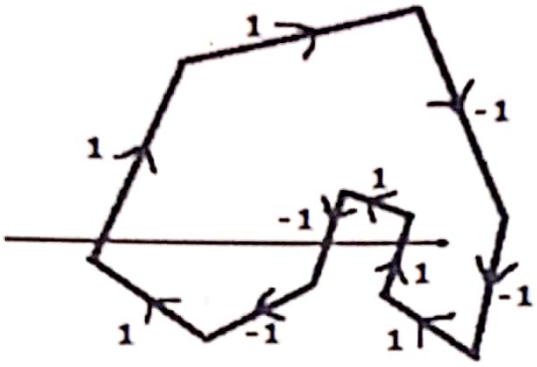
The edge that crosses along the line from the point (x,y) to infinity is counted. In case of odd interactions, the point (x,y) is an interior point and in case of even interactions, the point (x,y) is an exterior point. This concept is depicted by the following example:



It is observed from the above point (x,y), that the number of interaction 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 interaction 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.