Chapter 4 – Geometry and Line Generation

4.1 Point and Line

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 $\mathbf{X_0, Y_0}$ and the second point of line as $\mathbf{X_1, Y_1}$.

A **line**, or **straight line**, is an (infinitely) thin, (infinitely) long, straight geometrical object, i.e. a curve that is long and straight. Given two points, in Euclidean geometry, one can always find exactly one line that passes through the two points; this line provides the shortest connection between the points and is called a straight line. Three or more points that lie on the same line are called ***collinear***. Two different lines can intersect in at most one point; whereas two different planes can intersect in at most one line. This intuitive concept of a line can be formalized in various ways.

A line may have three forms with respect to slope i.e. it may have slope = 1 as shown in following figure (a), or may have slope < 1 as shown in figure (b) or it may have a slope > 1 as shown in figure (c). Now if a line has slope = 1 it is very easy to draw the line by simply starting from one point and go on incrementing the x and y coordinates till they reach the second point. So that is a simple case but if slope < 1 or is > 1 then there will be some problem.

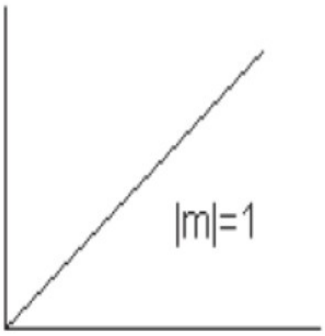


figure (a)

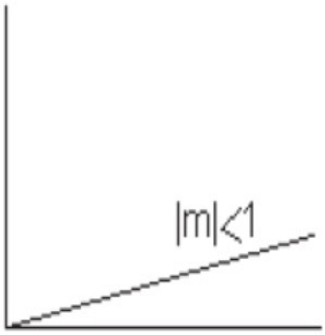


figure (b)

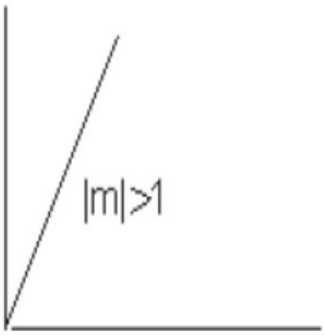


figure (c)

There are three line drawing techniques such as:

- Incremental line algorithm
- DDA line algorithm
- Bresenham line algorithm

4.1.1 Incremental Line Algorithm

This algorithm uses simple line equation **y = m x + b**

Where **m = dy / dx**

and b **= y – m x**

Now check if **|m| < 1** then starting at the first point, simply increment **x by 1** (unit increment) till it reaches ending point; whereas calculate **y point** by the equation for each **x** and conversely if **|m|>1** then increment **y by 1** till it reaches ending point; whereas calculate **x point** corresponding to each **y**, by the equation.

Now before moving ahead let us discuss why these two cases are tested. First if |m| is less than 1 then it means that for every subsequent pixel on the line there will be unit increment in x direction and there will be less than 1 increment in y direction and vice versa for slope greater than 1.

Let us clarify this with the help of an example:

Example-1: Suppose a line has two points p1 (10, 10) and p2 (20, 18)

Now difference between y coordinates that is dy = y2 – y1 = 18 – 10 = 8

Whereas difference between x coordinates is dx = x2 – x1 = 20 – 10 = 10

This means that there will be 10 pixels on the line in which for x-axis there will be distance of 1 between each pixel and for y-axis the distance will be 0.8.

Example-2: Consider case of another line with points p1 (10, 10) and p2 (16, 20)

Now the difference between y coordinates that is dy = y2 – y1 = 20 – 10 = 10

Whereas difference between x coordinates is dx = x2 – x1 = 16 – 10 = 6

This means that there will be 10 pixels on the line in which for x-axis there will be distance of 0.6 between each pixel and for y-axis the distance will be 1.

Now having discussed this concept at length let us now understand the algorithm to draw a line using above technique which is called as incremental line algorithm:

Incremental_Line (Point p1, Point p2)

dx = p2.x – p1.x

dy = p2.y – p1.y

m = dy / dx

x = p1.x

y = p1.y

b = y – m * x

if |m| < 1

for counter = p1.x to p2.x

drawPixel (x, y)

x = x + 1

y = m * x + b

else

for counter = p1.y to p2.y

drawPixel (x, y)

y = y + 1

x = (y – b) / m

Hence the above algorithm is quite simple and easy as it involves lot of mathematical calculations that is for calculating coordinate using equation each time and secondly it works only in incremental direction.

4.1.2 DDA Algorithm (Digital Differential Analyzer)

The DDA algorithm has very simple technique. Find the difference of **dx and dy** between **x coordinates and y coordinates** respectively by ending points of a line. If **|dx|** is greater than **|dy|,** then **|dx|** will be step and otherwise **|dy|** will be step.

if |dx|>|dy| then

step = |dx|

else

step = |dy|

Now it is very simple to say that step is the total number of pixel required for a line. Next step is to divide dx and dy by step to get xIncrement and yIncrement that is the increment required in each step to find next pixel value.

xIncrement = dx/step

yIncrement = dy/step

Next a loop is required that will run step times. In the loop drawPixel and add xIncrement in x1 by and yIncrement in y1.

To sum-up all above in the algorithm, we will get,

DDA_Line (Point p1, Point p2)

dx = p2.x – p1. x

```
dy = p2.y – p1. y
x1=p1.x
y1=p1.y
if |dx|>|dy| then
step = |dx|
else
step = |dy|
xIncrement = dx/step
yIncrement = dy/step
for counter = 1 to step
drawPixel (x1, y1)
x1 = x1 + xIncrement
y1 = y1 + yIncrement
```

4.1.3 Bresenhams Line Algorithm

The Bresenham algorithm is another incremental scan conversion algorithm. The big advantage of this algorithm is that, it uses only integer calculations. Moving across the x axis in unit intervals and at each step choose between two different y coordinates.

Bresenham's algorithm finds the closest integer coordinates to the actual line, using only integer math. Assuming that the slope is positive and less than 1, moving 1 step in the x direction, y either stays the same, or increases by 1. A decision function is required to resolve this choice.

If the current point is (xi, yi), the next point can be either (xi+1,yi) or (xi+1,yi+1) . The actual position on the line is (xi+1, m(xi+1)+c) . Calculating the distance between the true point, and the two alternative pixel positions available gives:

$$\mathbf{d1 = y - yi}$$
$$\mathbf{= m * (x+1)+b-yi}$$
$$\mathbf{d2 = yi + 1 - y}$$
$$\mathbf{= yi + 1 – m (xi + 1) - b}$$

Let us define a decision function p, to determine which distance is closer to the true point. By taking the difference between the distances, the decision function will be positive if d1 is larger, and negative otherwise. A positive scaling factor is added to ensure that no division is necessary and only integer math need be used.

$$pi = dx\ (d1-d2)$$

pi = dx (2m * (xi+1) + 2b – 2yi-1)

pi = 2 dy (xi+1) –2 dx yi + dx (2b-1) ---------- (i)

pi = 2 dy xi – 2 dx yi + k ---------- (ii)

where k=2 dy + dx (2b-1)

Then we can calculate pi+1 in terms of pi without any xi , yi or k .

pi+1 = 2 dy xi+1 – 2 dx yi+1 + k

pi+1 = 2 dy (xi + 1) - 2 dx yi+1 + k since xi+1= xi + 1

pi+1 = 2 dy xi + 2 dy- 2 dx yi+1 + k ---------- (iii)

Now subtracting (ii) from (iii), we get

pi+1 - pi = 2 dy - 2 dx (yi+1 - yi)

pi+1 = pi + 2 dy - 2 dx (yi+1 - yi)

If the next point is: (xi+1,yi) then

d1<d2 => d1-d2<0

=> pi<0

=> **pi+1= pi + 2 dy**

If the next point is: (xi+1,yi+1) then

d1>d2 => d1-d2>0

=> pi>0

=> **pi+1= pi + 2 dy - 2 dx**

The pi is our decision variable, and calculated using integer arithmetic from pre-computed constants and its previous value. Now a question is remaining how to calculate initial value of pi. For that use equation (i) and put values (x1, y1)

pi = 2 dy (x1+1) – 2 dx yi + dx (2b-1)

where b = y – m x implies that

pi = 2 dy x1 +2 dy – 2 dx yi + dx (2 (y1 – mx1) -1)

pi = 2 dy x1 +2 dy – 2 dx yi + 2 dx y1 – 2 dy x1 - dx

pi = 2 dy x1 +2 dy – 2 dx yi + 2 dx y1 – 2 dy x1 - dx

there are certain figures that will cancel each other shown in same different colour

pi = 2 dy - dx

Thus Bresenham's line drawing algorithm is as follows:

dx = x2-x1

dy = y2-y1

```
p = 2dy-dx
c1 = 2dy
c2 = 2(dy-dx)
x = x1
y = y1
plot (x,y,colour)
while (x < x2 )
        x++;
        if (p < 0)
        p = p + c1
        else
        p = p + c2
        y++
        plot (x,y,colour)
```

Again, this algorithm can be easily generalized to other arrangements of the end points of the line segment, and for different ranges of the slope of the line.

4.1.4 Improving Performance of Line Drawing

Several techniques can be used to improve the performance of line-drawing procedures. These are important because line drawing is one of the fundamental primitives used by most of the other rendering applications. An improvement in the speed of line-drawing will result in an overall improvement of most graphical applications. Removing procedure calls using macros or inline code can produce improvements. Unrolling loops also may produce longer pieces of code, but these may run faster. The use of separate x and y coordinates can be discarded in favour of direct frame buffer addressing. Most algorithms can be adapted to calculate only the initial frame buffer address corresponding to the starting point and to replaced:

X++ with Addr++

Y++ with Addr+=XResolution

Fixed point representation allows a method for performing calculations using only integer arithmetic, but still obtaining the accuracy of floating point values. In fixed point, the fraction part of a value is stored separately, in another integer:

M = Mint.Mfrac

Mint = Int(M)

Mfrac = Frac(M)× MaxInt

Addition in fixed point representation occurs by adding fractional and integer components separately, and only transferring any carry-over from the fractional result to the integer result. The sequence could be implemented using the following two integer additions:

ADD Yfrac,Mfrac ; ADC Yint,Mint

Improved versions of these algorithms exist. For example the following variations exist on Bresenham's original algorithm:

Symmetry (forward and backward simultaneously)

Segmentation (divide into smaller identical segments - GCD(D x,D y))

Double step, triple step, n step

4.1.5 Introduction to Pixel

A pixel is the smallest unit of a digital image or graphic that can be displayed and represented on a digital display device. A pixel is the basic logical unit in digital graphics. Pixels are combined to form a complete image, video, text or any visible thing on a computer display. A pixel is also known as a picture element.

A pixel is represented by a dot or square on a computer monitor display screen. Pixels are the basic building blocks of a digital image or display and are created using geometric coordinates. Depending on the graphics card and display monitor, the quantity, size and color combination of pixels varies and is measured in terms of the display resolution.

For example, a computer with a display resolution of 1280 x 768 will produce a maximum of 98,3040 pixels on a display screen. Each pixel has a unique logical address, a size of eight bits or more and, in most high-end display devices, the ability to project millions of different colors.

The pixel resolution spread also determines the quality of display; more pixels per inch of monitor screen yields better image results.

Setting a Pixel

Initial Task: Turning on a pixel (loading the frame buffer/bit-map). Assume the simplest case, i.e., an 8-bit, non-interlaced graphics system. Then each byte in the frame buffer corresponds to a pixel in the output display.

To find the address of a particular pixel (X,Y) we use the following formula:

addr(X, Y) = addr(0,0) + Y rows * (Xm + 1) + X (all in bytes)

addr(X,Y) = the memory address of pixel (X,Y)

addr(0,0) = the memory address of the initial pixel (0,0)

Number of rows = number of raster lines.

Number of columns = number of pixels/raster line.

Example: For a system with 640 × 480 pixel resolution, find the address of pixel X = 340, Y = 150?

Solution: addr(340, 150) = addr(0,0) + 150 * 640 (bytes/row) + 340

= base + 96,340 is the byte location

Graphics system usually have a command such as set_pixel (x, y) where x, y are integers.

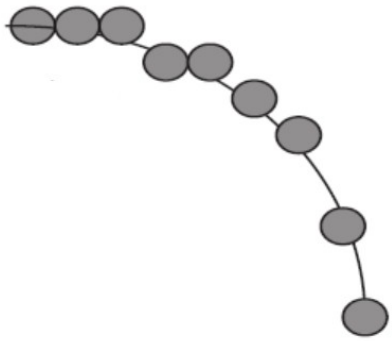
4.2 Circle Drawing Techniques

A circle is the set of points in a plane that are equidistant from a given point O. The distance r from the center is called the radius, and the point O is called the center. Twice the radius is known as the diameter **d = 2r**. The angle a circle subtends from its center is a full angle, equal to 360° or 2π radians.

A circle has the maximum possible area for a given perimeter, and the minimum possible perimeter for a given area. The perimeter C of a circle is called the circumference, and is given by **C** = 2π **r**

First of all for circle drawing we need to know what the input will be. Well the input will be one center point (x, y) and radius r. Therefore, using these two inputs there are a number of ways to draw a circle. They involve understanding level very simple to complex and reversely time complexity inefficient to efficient. We will see them one by one giving comparative study.

4.2.1 Circle Drawing Using Cartesian coordinates



This technique uses the equation for a circle on radius r centred at (0, 0) given as:

$$x^2 + y^2 = r^2,$$

an obvious choice is to plot

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

Obviously in most of the cases the circle is not centered at (0, 0), rather there is a center point (X_c, y_c); other than (0, 0). Therefore the equation of the circle having center at point (X_c, y_c).

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

this implies that,

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

Using above equation a circle can easily be drawn. The value of x varies from $\mathbf{r - x_c}$ **to** $\mathbf{r + x_c}$. Whereas y will be calculated using above formula. Using this technique we can write a simple algorithm:

Circle1 (xcenter, ycenter, radius)
for x = radius - xcenter to radius + xcenter

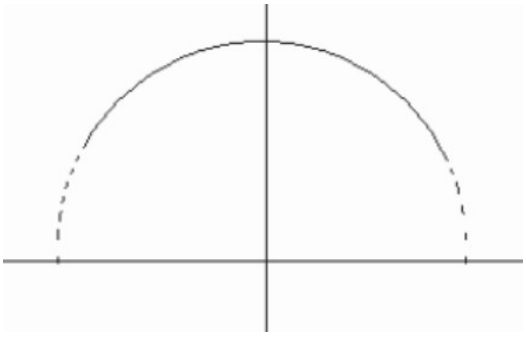
$y = x_c + \sqrt{r^2 - (x - x_c)^2}$

drawPixel (x, y)

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

drawPixel (x, y)

4.2.2 Circle Drawing using Polar Coordinates



A better approach, to eliminate unequal spacing as shown in above figure is to calculate points along with the circular boundary using polar coordinates r and θ expressing the circle equation in parametric polar form yields the pair of equations:

$$x = x_c + r \cos \theta$$
$$y = y_c + r \sin \theta$$

Using above equation circle can be plotted by calculating x and y coordinates as θ takes values from 0 to 360 degrees or 0 to . The step size chosen for θ depends on the application and the display device. Larger angular separations along the circumference can be connected with straight-line segments to approximate the circular path. For a more continuous boundary on a raster display, we can set the step size at 1/r. This plots pixel positions that are approximately one unit apart.

Now let us see how this technique can be sum up in algorithmic form.

Circle2 (xcenter, ycenter, radius)

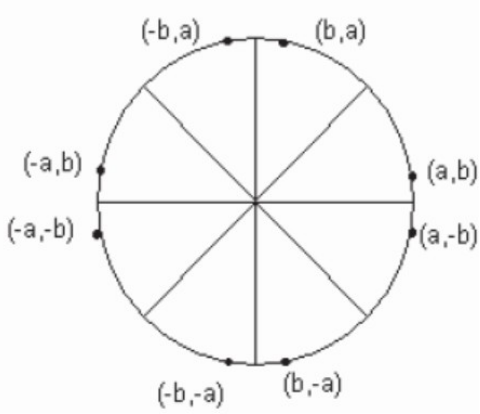
for $\theta = 0$ to 2π step $1/r$

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

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

drawPixel (x, y)

Again this is very simple technique and also solves problem of unequal space but unfortunately this technique is still inefficient in terms of calculations involves especially floating point calculations.



Calculations can be reduced by considering the symmetry of circles. The shape of circle is similar in each quadrant. We can generate the circle section in the second quadrant of the xy-plane by noting that the two circle sections are symmetric with respect to the y axis and circle sections in the third an fourth quadrants can be obtained from sections in the first and second quadrants by considering symmetry about the x axis. We can take this one step further and note that there is also symmetry between octants. Circle sections in adjacent octants within one quadrant are symmetric with respect to the 45^0 line dividing the two octants. These symmetry conditions are illustrated in above figure. Therefore above algorithm can be optimized by using symmetric octants. Let's see:

Circle2 (xcenter, ycenter, radius)

for $\theta = 0$ to $\pi / 4$ step 1/r

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

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

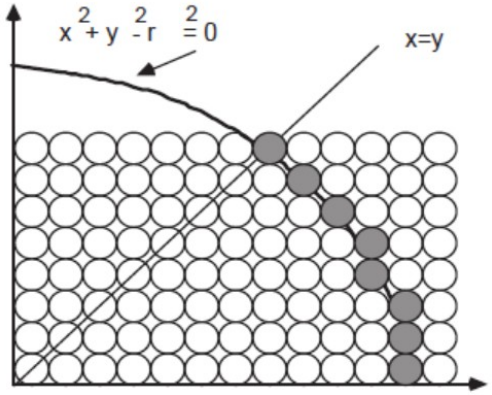
DrawSymmetricPoints (xcenter, ycenter, x, y)

DrawSymmeticPoints (xcenter, ycenter, x, y)

Plot (x + xcenter, y + ycenter)
Plot (y + xcenter, x + ycenter)
Plot (y + xcenter, -x + ycenter)
Plot (x + xcenter, -y + ycenter)
Plot (-x + xcenter, -y + ycenter)
Plot (-y + xcenter, -x + ycenter)
Plot (-y + xcenter, x + ycenter)
Plot (-x + xcenter, y + ycenter)

4.2.3 Midpoint Circle Algorithm

As in the Bresenham line drawing algorithm we derive a decision parameter that helps us to determine whether or not to increment in the y coordinate against increment of x coordinate or vice versa for slope > 1. Similarly here we will try to derive decision parameter which can give us closest pixel position.



Let us consider only the first octant of a circle of radius r centred on the origin. We begin by plotting point (r, 0) and end when x < y.

The decision at each step is whether to choose the pixel directly above the current pixel or the pixel; which is above and to the left (8-way stepping).

Assume:

$P_i = (x_i, y_i)$	is the current pixel.
$T_i = (x_i, y_i + 1)$	is the pixel directly above
$S_i = (x_i - 1, y_i + 1)$	is the pixel above and to the left.

To apply the midpoint method, we define a circle function:

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

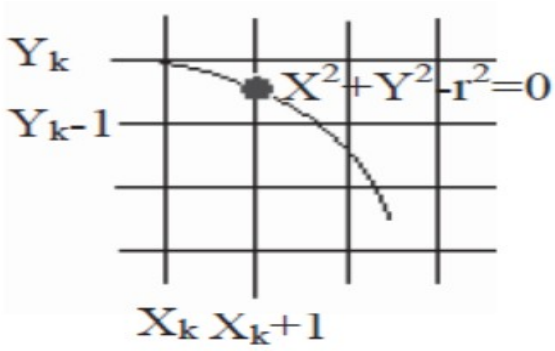
Therefore following relations can be observed:

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

The circle function tests given above are performed for the midpoints between pixels near the circle path at each sampling step. Thus, the circle function is the decision parameter in the midpoint algorithm, and we can set up incremental calculations for this function as we did in the line algorithm.

Figure below shows the midpoint between the two candidate pixels at sampling position $\mathbf{x_{k+1}}$. Assuming we have just plotted the pixel at $\mathbf{(x_k, y_k)}$, we next need to determine whether the pixel

at position (x_{k+1}, y_k), we next need to determine whether the pixel at position (x_{k+1}, y_k), or the one at position (x_{k+1}, y_{k-1}) is closer to the circle. Our decision parameter is the circle function evaluated at the midpoint between these two pixels:



$$P_k = f_{circle}(x_k + 1, y_k - \tfrac{1}{2})$$
$$P_k = (x_k + 1)^2 + (y_k - \tfrac{1}{2})^2 - r^2 \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (1)$$

If $p_{k<0}$, this midpoint is inside the circle and the pixel on scan line y_k is closer to the circle boundary. Otherwise, the mid position is outside or on the circle boundary, and we select the pixel on scan-line y_{k-1}.

Successive decision parameters are obtained using incremental calculations. We obtain a recursive expression for the next decision parameter by evaluating the circle function at sampling position

$$x_{k+2} = x_{k+1} + 1 = x_k + 1 + 1 = x_k + 2:$$

$$P_{k+1} = f_{circle}(x_{k+1} + 1, y_{k+1} - \tfrac{1}{2})$$
$$P_{k+1} = [(x_k + 1) + 1]^2 + (y_{k+1} - \tfrac{1}{2})^2 - r^2 \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (2)$$

Subtracting (1) from (2), we get

$$P_{k+1} - P_k = [(x_k + 1) + 1]^2 + (y_{k+1} - \tfrac{1}{2})^2 - r^2 - (x_k + 1)^2 - (y_k - \tfrac{1}{2})^2 + r^2$$

or

$$P_{k+1} = P_k + 2(x_k + 1) + (y^2_{k+1} - y^2_k) - (y_{k+1} - y_k) + 1$$

Where y_{k+1} is either y_k **or** y_{k-1}, depending on the sign of P_k. Therefore, if $P_{k<0}$ or negative then y_{k+1} will be y_k and the formula to calculate P_{k+1} will be:

$$P_{k+1} = P_k + 2(x_k + 1) + (y^2_k - y^2_k) - (y_k - y_k) + 1$$
$$P_{k+1} = P_k + 2(x_k + 1) + 1$$

Otherwise, if $P_{k>0}$ or positive then y_{k+1} will be y_{k-1} and the formula to calculate P_{k+1} will be:

$$P_{k+1} = P_k + 2(x_k + 1) + [(y_k - 1)^2 - y^2_k] - (y_k - 1 - y_k) + 1$$
$$P_{k+1} = P_k + 2(x_k + 1) + (y^2_k - 2y_k + 1 - y^2_k] - (y_k - 1 - y_k) + 1$$

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

Now a similar case that we observe in line algorithm is that how would starting P_k be evaluated. For this at the start pixel position will be (0, r). Therefore, putting this value is equation, we get

$$P_0 = (0 + 1)^2 + (r - \tfrac{1}{2})^2 - r^2$$
$$P_0 = 1 + r^2 - r + \tfrac{1}{4} - r^2$$
$$P_0 = 5/4 - r$$

If radius r is specified as an integer, we can simply round p_0 to:

$$P_0 = 1 - r$$

Since all increments are integer. Finally sum up all in the algorithm:

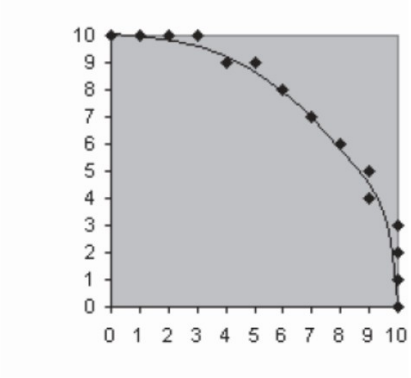
```
MidpointCircle (xcenter, ycenter, radius)
y = r;
x = 0;
p = 1 - r;
do
        DrawSymmetricPoints (xcenter,  ycenter, x, y)
        x = x + 1
        If p < 0 Then
                p = p + 2 * ( x + 1 ) + 1
        else
                y = y - 1
                p = p + 2 * ( x + 1 ) – 2 * ( y  - 1 ) + 1
while ( x  <  y )
```

Now let us consider an example to calculate first octant of the circle using above algorithm; while one quarter is displayed where you can observe that exact circle is passing between the points calculated in a raster circle.

Example:

xcenter= 0 ycenter= 0 radius= 10

p	x	Y
-9	0	10
-6	1	10
-1	2	10
6	3	10
-3	4	9
8	5	9
5	6	8
6	7	7

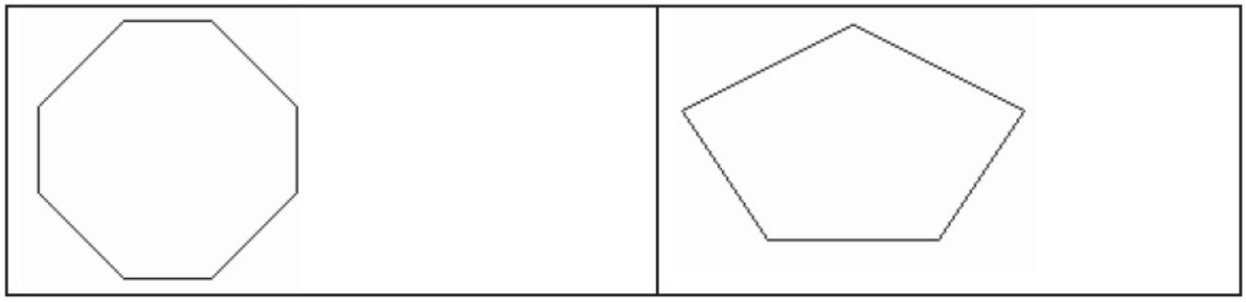


4.3 Polygon Filling Algorithm

A **polygon** can be defined as a shape that is formed by line segments that are placed end to end, creating a continuous closed path. Polygons can be divided into three basic types such as **convex**, **concave**, and **complex**.

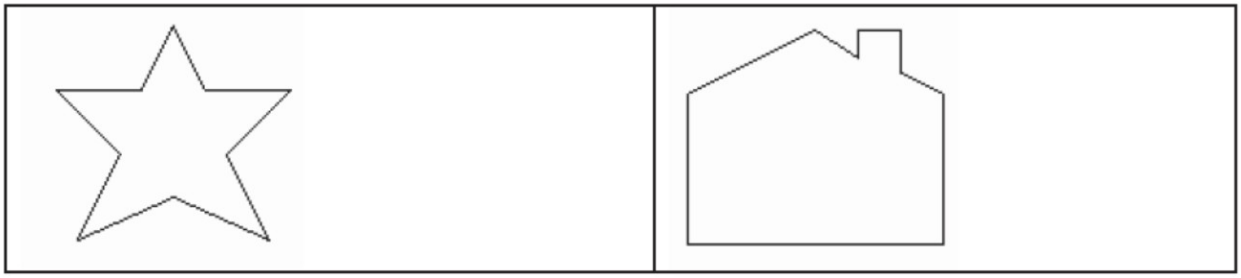
Convex: Convex polygons are the simplest type of polygon to fill. To determine whether or not a polygon is convex, ask the following question:

Does a straight line connecting any two points that are inside the polygon intersect any edges of the polygon?

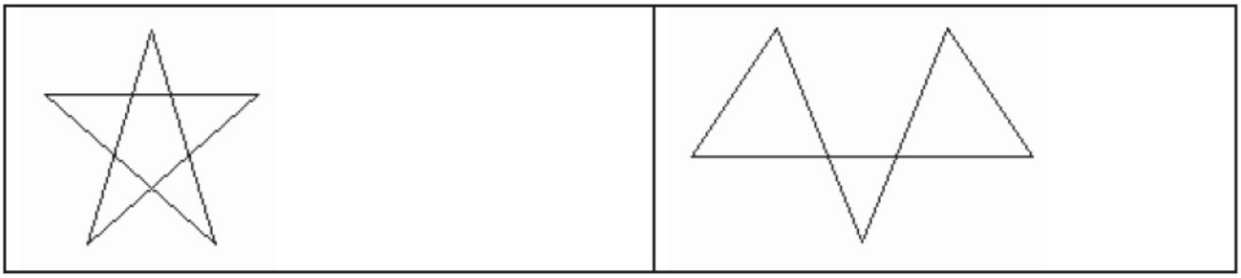


If the answer is no, the polygon is convex. This means that for any scan-line, the scanline will cross at most two polygon edges (not counting any horizontal edges). Convex polygon edges also do not intersect each other.

Concave: Concave polygons are a superset of convex polygons, having fewer restrictions than convex polygons. The line connecting any two points that lie inside the polygon may intersect more than two edges of the polygon. Thus, more than two edges may intersect any scan line that passes through the polygon. The polygon edges may also touch each other, but they may not cross one another.



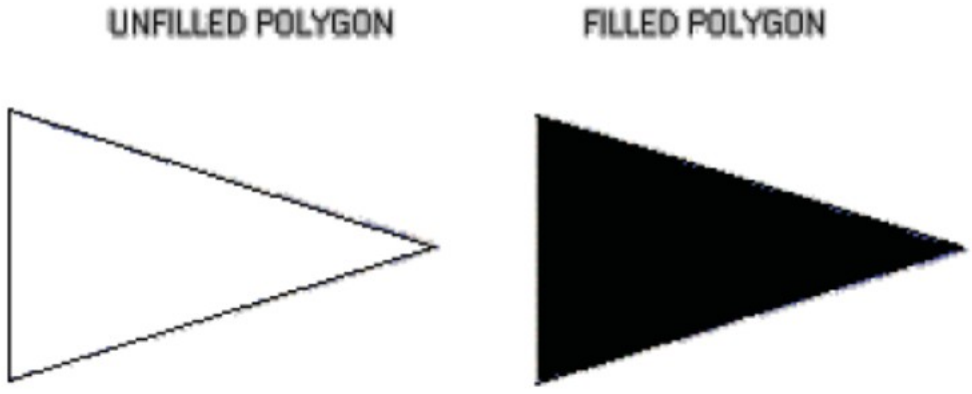
Complex: Complex polygons are just what their name suggests: complex. Complex polygons are basically concave polygons that may have self-intersecting edges. The complexity arises from distinguishing which side is inside the polygon when filling it.



Difference between Filled and Unfilled Polygon

When an unfilled polygon is rendered, only the points on the perimeter of the polygon are drawn. Examples of unfilled polygons are shown below:

However, when a polygon is filled, the interior of the polygon must be considered. All of the pixels within the boundaries of the polygon must be set to the specified color or pattern. Here, we deal only with solid colors. The following figure shows the difference between filled and unfilled polygons.

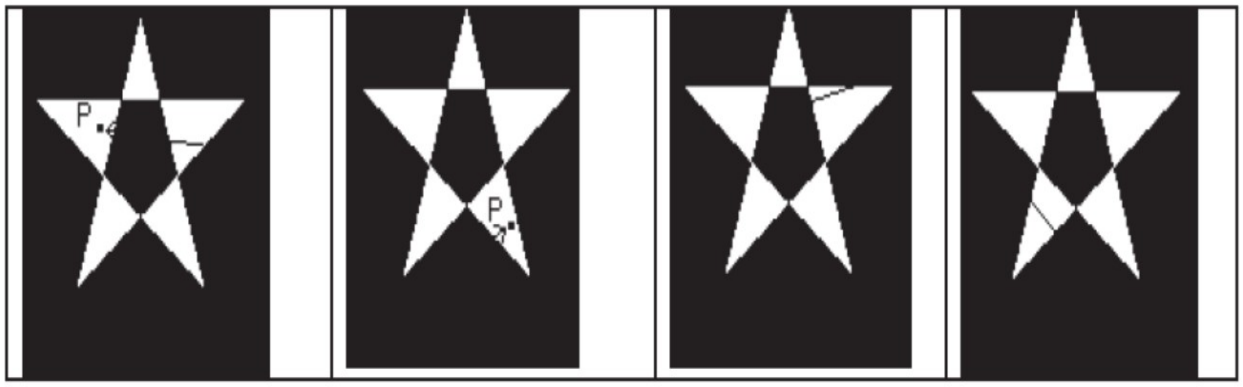


In order to determine which pixels are inside the polygon, the odd-parity rule is used within the scan-line polygon fill algorithm.

4.3.1 Parity

Parity is a concept used to determine which pixels lie within a polygon, i.e. which pixels should be filled for a given polygon.

The Underlying Principle: Conceptually, the odd parity test entails drawing a line segment from any point that lies outside the polygon to a point P, which we wish to determine whether it is inside or outside of the polygon. Count the number of edges that the line crosses. If the number of polygon edges crossed is **odd**, then P lies within the polygon. Similarly, if the number of edges is even, then P lies outside of the polygon. There are special ways of counting the edges when the line crosses a vertex. Examples of counting parity can be seen in the following demonstration.



4.3.2 Scan Line Algorithm

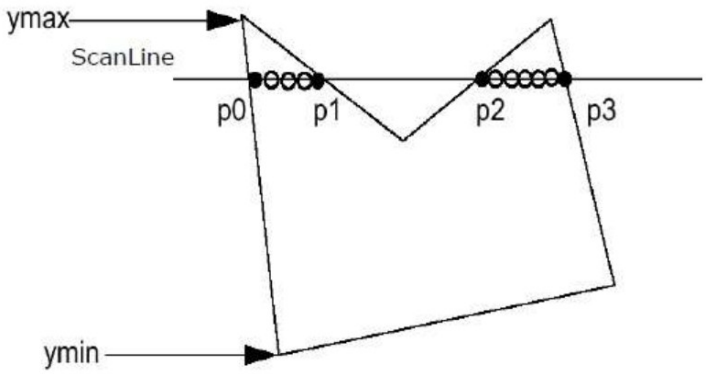
In order to fill a polygon, we do not want to have to determine the type of polygon that we are filling. The easiest way to avoid this situation is to use an algorithm that works for all three types of polygons. Since both convex and concave polygons are subsets of the complex type, using an algorithm that will work for complex polygon filling should be sufficient for all three types. The scan-line polygon fill algorithm, which employs the odd/even parity concept works for complex polygon filling.

Reminder: The basic concept of the scan-line algorithm is to draw points from edges of odd parity to even parity on each scan-line.

A scan-line is a line of constant y value, i.e., y = c, where c lies within our drawing region, e.g., the window on our computer screen.

The following steps show how this algorithm works.

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



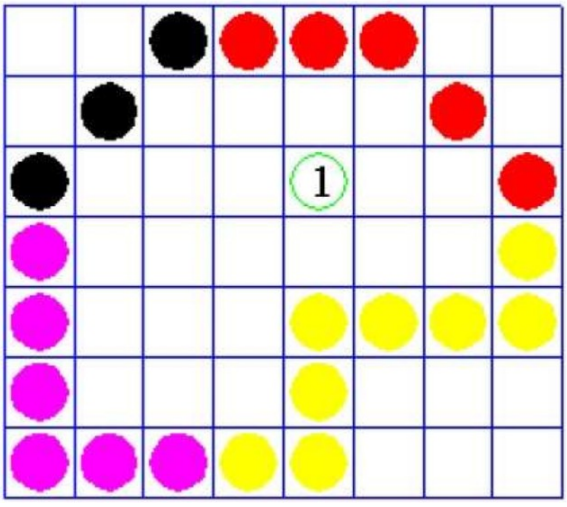
Step 2 − ScanLine intersects with each edge of the polygon from Ymin to Ymax. Name each intersection point of the polygon. As per the figure shown above, they are named as p0, p1, p2, p3.

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

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

4.3.2 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.



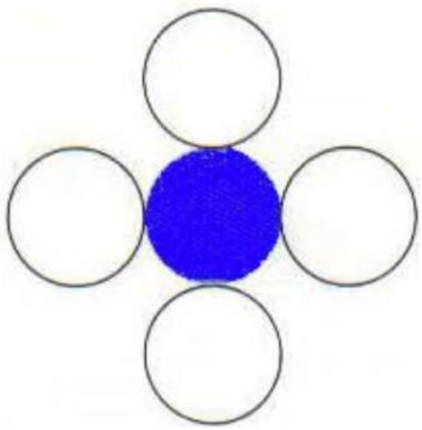
4.3.3 Boundary Fill Algorithm

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.

4.3.4 The 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), 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.

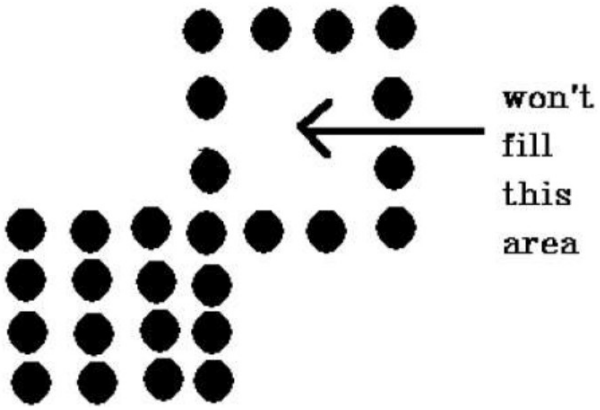
setPixel(seedx, seedy, fcol)

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

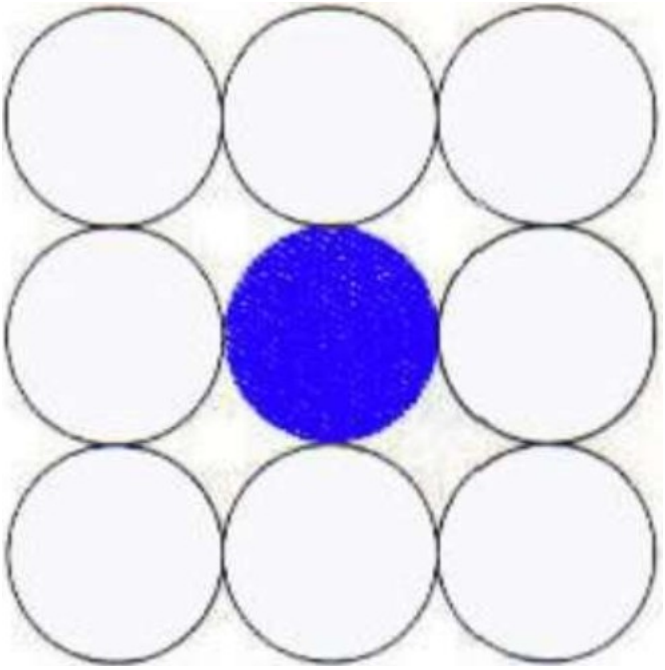
There is a problem with this technique. Consider the case as shown below where we tried to fill the entire region. Here, the image is filled only partially. In such cases, 4-connected pixels technique cannot be used.



4.3.5 The 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), 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, seedy + 1, fcol, dcol)

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

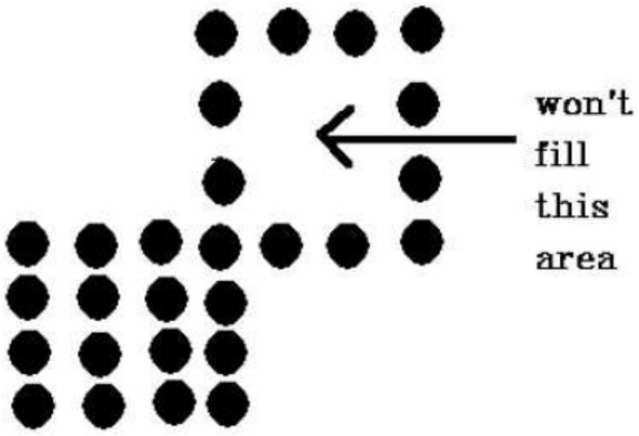
FloodFill (seedx + 1, seedy + 1, fcol, dcol)

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.

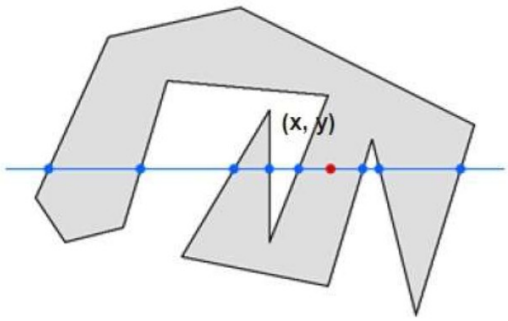


4.3.6 Inside-outside Test

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 such as Odd-Even Rule and Nonzero Winding Number Rule by which we can identify whether particular point is inside an object or outside.

Odd-Even Rule

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

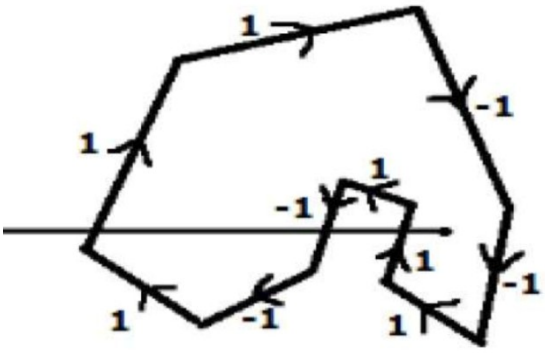


From the above figure, we can see that from the point (x,y), the number of interactions point on the left side is 5 and on the right side is 3. From both ends, the number of interaction points is odd, so the point is considered within the object.

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.