Scan Conversion

4.1 INTRODUCTION

The process of representing continuously graphic objects as a collection of discrete pixels is called scan conversion. In other words, we can say that scan conversion is continuous-to-discrete transformation.

FIGURE 4.1 Continuous-to-Discrete Transformation

The most used graphic objects are the lines, the sectors, the arcs, the ellipses, the rectangles, and the polygons. In this book we will discuss only simple primitives, i.e., straight lines and few simple curves such as circle, ellipse, parabola and hyperbola. The motivation of a scan conversion algorithm is not only to produce primitive but also to produce it rapidly and satisfactorily.

On the monitor screen, let origin actually be at the top left-hand corner; x coordinate increases from left-to-right and y coordinate increases in downward direction. Such a system is left-handed coordinate system, Fig. 4.2(a). However, in this chapter and onward we will consider origin at lower left-hand corner where x coordinate increases from left-to-right and y coordinate increases in upward direction. Such a system is called right-handed coordinate system, Fig. 4.2(b).

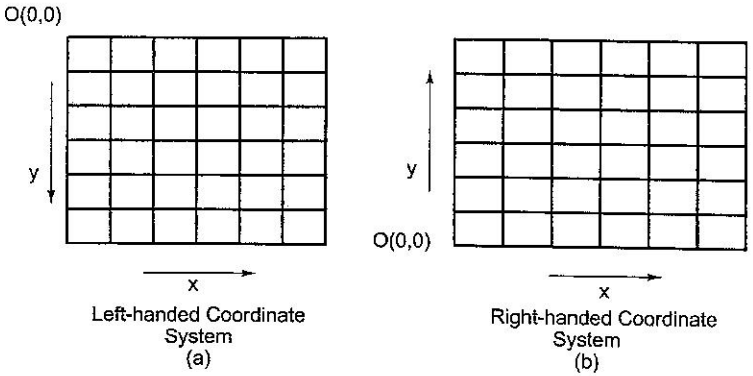


FIGURE 4.2

Lecture – 3

NOTE :- The distortion introduced by the conversion from continuous space to discrete space is referred to as the Aliasing effect.

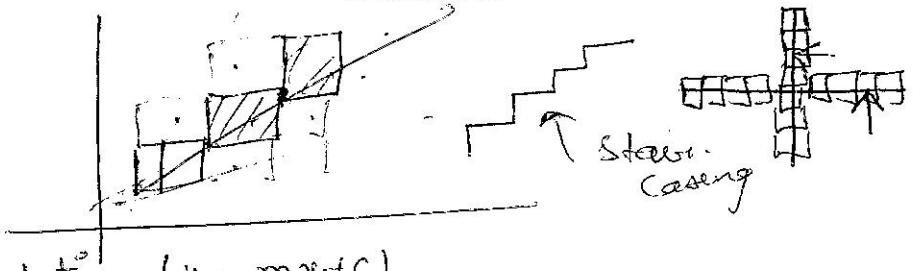
3) Output Primitives

- Basic geometric structures are referred to as Output primitives, like line, circle, and other curves.
- Scan conversion :- The area of computer graphics that is responsible for converting a continuous figure into its discrete approximation.

Scan Converting Lines :-

⇒ Given the specification of a line, we need to find the collection of addressable pixels which most closely approximate the line.

- It is easy to draw vertical and horizontal lines and lines with slope = ±1. But other lines create the problem of stair casing, also called as jaggies or aliasing.



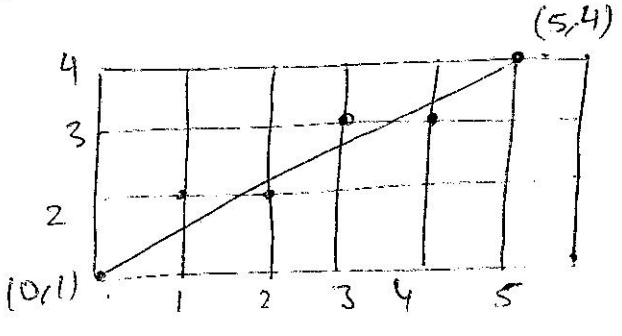
- A simple solution ($y = mx + c$)
- Digital Differential Analyzer (DDA) ($y = \frac{(y_1 - y_0)}{(x_1 - x_0)} x + b$)
- Bresenham's Line Algorithm ($d_{start} = 2\Delta x - \Delta y$)
- Mid-point Line Algorithm

Simple Solution :- (Not Suitable for high Slopes)

Ex. starting point $x_0 = 0$, end point $x_1 = 5$, Slope = 0.6, $b = 1$
Calculate y values.

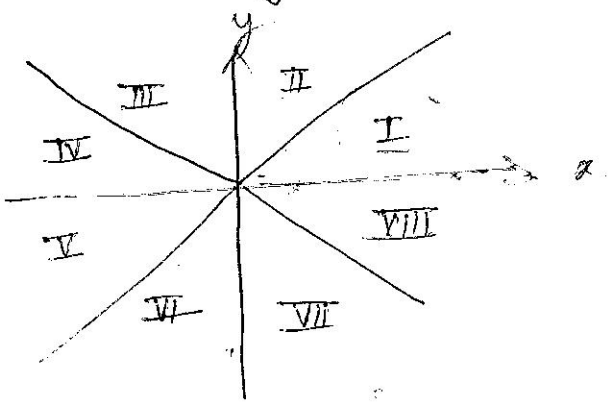
Ans

$y_0 = 1$ (0, 1)
$y_1 = 2$ (5/5) (1, 2)
$y_2 = 2$ (11/5) (2, 2)
$y_3 = 3$ (14/5) (3, 3)
$y_4 = 3$ (17/5) (4, 3)
$y_5 = 4$ (20/5) (5, 4)



DDA Algorithm :-

- Also called incremental algorithm.



Case I: $|dx| > |dy|$ $(m < 1)$

```
x = x0, y = y0
while (x < x1)
{ draw_point (x, Round(y));
  y = y + m;
  x = x + 1;
}
```

Case II: $|dy| > |dx|$ $(m > 1)$

```
x = x0, y = y0
while (y < y1)
{ draw_point (Round(x), y);
  x = x + 1/m;
  y = y + 1;
}
```

Case III: $m > 1$, negative slope

```
x = x - 1/m;
y = y - 1;
```

Case IV: $m < 1$, negative slope

```
y = y - m;
x = x - 1;
```

NOTE :- Here we have removed costly computations of "*" and "/" but Round() and floating point addition still exist.

Q Draw a line connecting (100,100) and (500,105) using DDA.

Ans $m < 1$

$y = y + m;$

$x = x + 1;$

$x_0 = 100, \quad y_0 = 100$

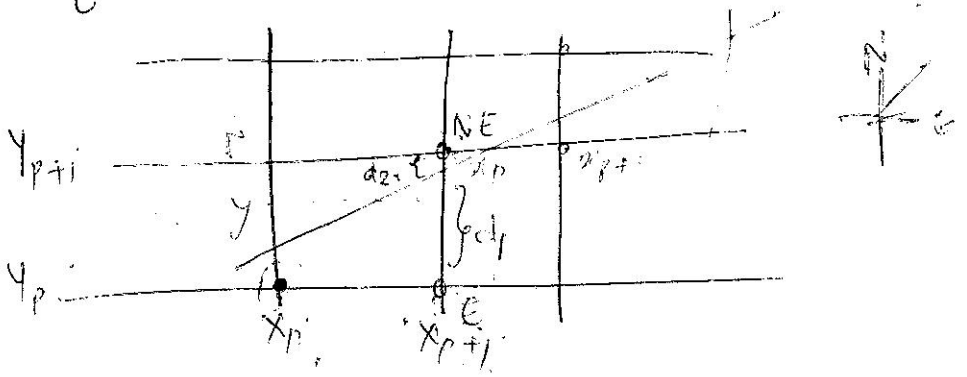
$x_m = 500, \quad y_m = 105$

$x_1 = x_0 + 1 = 100 + 1 = 101$

$y_1 = y_0 + m = 100 + \frac{5}{400} =$

Bresenham's Line Algorithm :-

⇒ Very efficient and faster line drawing Algorithms. It scan converts lines and uses only increment integer calculation. (Also used for circles and other similar curves).



If $d_1 > d_2$, NE point is closer, Coordinate y between NE and E can be calculated using equation.

$$y = m(x_p + 1) + c$$

$$d_2 = (y_{p+1}) - y = y_p + 1 - m(x_p + 1) - c$$

$$d_1 = y - y_p = m(x_p + 1) + c - y_p$$

$$d_1 - d_2 = 2m(x_p + 1) - 2y_p + 2c - 1$$

$$d = d_1 - d_2 = 2\frac{\Delta y}{\Delta x}(x_p + 1) - 2y_p + 2c - 1$$

$$\Delta x(d_1 - d_2) = 2\Delta y\, x_p - 2\Delta x y_p + 2\Delta y + \Delta x(2c - 1)$$

$$= 2\Delta y\, x_p - 2\Delta x y_p + c'$$

$$d = \Delta x(d_1 - d_2) = 2\Delta y x_p - 2\Delta x y_p + c'$$

$\Delta x > 0$, if d is -ve, $d_1 < d_2$, we choose E.

if d is +ve, $d_1 > d_2$, we choose NE.

At each successive calculation of x values, we calculate d $d_{old} = d$

Case 1: Chosen E $\rightarrow$ $d_{new} = 2\Delta y(x_p+1) - 2\Delta x y_p + c'$

$$(\Delta d)_E = d_{new} - d_{old}$$
$$= 2\Delta y$$

Case 2: Chosen NE $\rightarrow$ $d_{new} = 2\Delta y(x_p+1) - 2\Delta x(y_p+1) + c'$

$$(\Delta d)_{NE} = d_{new} - d_{old}$$
$$= 2\Delta y - 2\Delta x$$

Now, we have to select initial decision variable d_{start}.

Assuming (0,0) as start

$$d_{start} = 2\Delta y\,0 - 2\Delta x\,0 + 2\Delta y + \Delta x(2\cdot 0 - 1)$$
$$= 2\Delta y - \Delta x$$

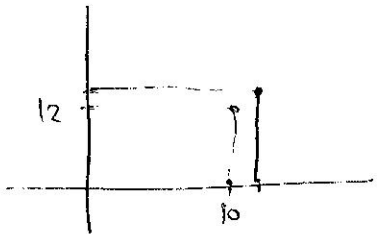
Q Draw a line from (10,12) to (20,18) using Bresenham's Algo.

$\Delta x = 10, \Delta y = 6$

$2\Delta y - 2\Delta x = -8$.

$d_{start} = 2\Delta y - \Delta x = 12 - 10 = 2$

$$m = \frac{\Delta y}{\Delta x} = 0.6 < 1$$

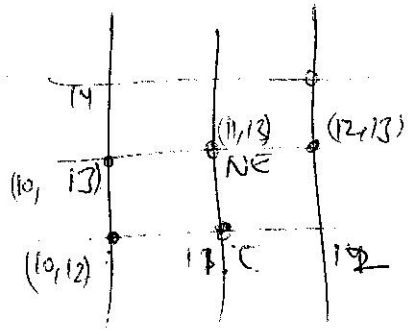


$d_{start} > 0$, so we choose NE (11,13)

$d_{new} = 2 + (-8) = -6$ ($d_{old} + 2\Delta y - 2\Delta x$)

$d_{new} < 0$, we choose E (12,13)

$d_{new} = -6 + 12 = 6$ ($d_{old} + 2\Delta y$)



SOLVED PROBLEMS

3.1 The endpoints of a given line are (0, 0) and (6, 18). Compute each value of y as x s... from 0 to 6 and plot the results.

An equation for the line was not given. Therefore, the equation of the line must be ... before proceeding. The equation of the line ($y = mx + b$) is found as follows.

First the slope m is found:

$$m = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1} = \frac{18 - 0}{6 - 0} = \frac{18}{6} = 3$$

Next, the y intercept b is found by plugging y_1 and x_1 into the equation $y = 3x + b$: $0 = ... + b$. Therefore, $b = 0$. Hence the equation for the line is $y = 3x$ (see Fig. 3.41).

3.2 Write the steps required to plot a line whose slope is between 0° and 45° using the sl... intercept equation.

1. Compute dx: $dx = x_2 - x_1$.
2. Compute dy: $dy = y_2 - y_1$.
3. Compute m: $m = dy/dx$.
4. Compute b: $b = y_1 - m \times x_1$.
5. Set (x, y) equal to the lower left-hand endpoint and set x_{end} equal to the largest ... x. If $dx < 0$, then $x = x_2$, $y = y_2$, and $x_{end} = x_1$. If $dx > 0$, then $x = x_1$, $y = y_1$, and $x_{...} = ...$
6. Test to determine whether the entire line has been drawn. If $x > x_{end}$, stop.
7. Plot a point at the current (x, y) coordinates.
8. Increment x: $x = x + 1$.
9. Compute the next value of y from the equation $y = mx + b$.
10. Go to step 6.

3.3 Use pseudo-code to describe the steps that are required to plot a line whose s... between 45° and −45° (i.e. $|m| > 1$) using the slope-intercept equation.

$y = 3x + 0$	x
0	0
3	1
6	2
9	3
12	4
15	5
18	6

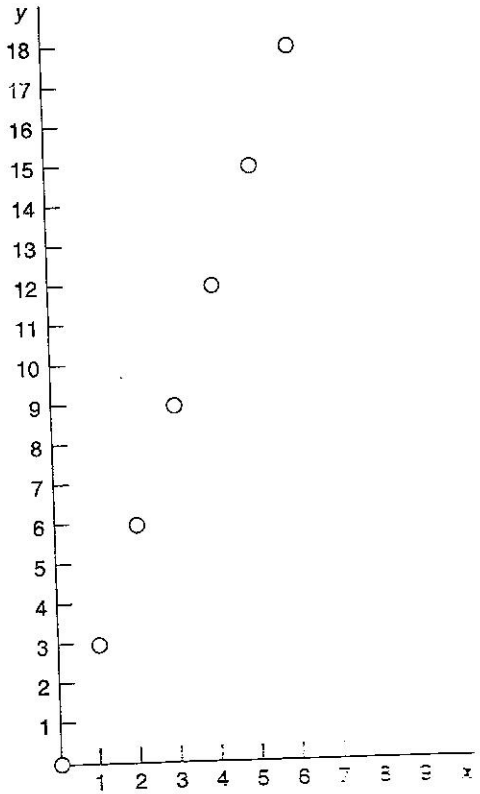


Fig. 3.41

Presume $y_1 < y_2$ for the two endpoints (x_1, y_1) and (x_2, y_2):

```
int x = x1, y = y1;
float xf, m = (y2 - y1)/(x2 - x1), b = y1 - mx1;
setPixel(x, y);
while (y < y2) {
   y ++ ;
   xf = (y - b)/m;
   x = Floor(xf + 0.5);
   setPixel(x, y);
}
```

3.4 Use pseudo-code to describe the DDA algorithm for scan-converting a line whose slope is between $-45°$ and $45°$ (i.e., $|m| \leq 1$).

Presume $x_1 < x_2$ for the two endpoints (x_1, y_1) and (x_2, y_2):

```
int x = x1, y;
float yf = y1, m = (y2 - y1)/(x2 - x1);
while (x <= x2) {
   y = Floor(yf + 0.5);
```

```
    setPixel(x, y);
    x++;
    y_f = y_f + m;
}
```

3.5 Use pseudo-code to describe the DDA algorithm for scan-converting a line whose slope is between 45° and −45° (i.e. $|m| > 1$).

Presume $y_1 < y_2$ for the two endpoints (x_1, y_1) and (x_2, y_2):

```
int x, y = y_1;
float x_f = x_1, m_inv = (x_2 - x_1)/(y_2 - y_1);
while (y <= y_2) {
    x = Floor(x_f + 0.5);
    setPixel(x, y);
    x_f = x_f + m_inv;
    y++;
}
```

3.6 What steps are required to plot a line whose slope is between 0° and 45° using Bresenham's method?

1. Compute the initial values:

$$dx = x_2 - x_1 \qquad Inc_2 = 2(dy - dx)$$
$$dy = y_2 - y_1 \qquad d = Inc_1 - dx$$
$$Inc_1 = 2dy$$

2. Set (x, y) equal to the lower left-hand endpoint and x_{end} equal to the largest value of . If $dx < 0$, then $x = x_2$, $y = y_2$, $x_{end} = x_1$. If $dx > 0$, then $x = x_1$, $y = y_1$, $x_{end} = x_2$.
3. Plot a point at the current (x, y) coordinates.
4. Test to see whether the entire line has been drawn. If $x = x_{end}$, stop.
5. Compute the location of the next pixel. If $d < 0$, then $d = d + Inc_1$. If $d \geq 0$, then $d = + Inc_2$, and then $y = y + 1$.
6. Increment x: $x = x + 1$.
7. Plot a point at the current (x, y) coordinates.
8. Go to step 4.

3.7 Indicate which raster locations would be chosen by Bresenham's algorithm when scan-converting a line from pixel coordinate (1, 1) to pixel coordinate (8, 5).

First, the starting values must be found. In this case

$$dx = x_2 - x_1 = 8 - 1 = 7 \qquad dy = y_2 - y_1 = 5 - 1 = 4$$

Therefore

$$Inc_1 = 2dy = 2 \times 4 = 8$$
$$Inc_2 = 2(dy - dx) = 2 \times (4 - 7) = -6$$

Computer Graphics & Multimedia Techniques

#Lecture - 4 :-

Mid-point Line Algorithm :-

⇒ We find out the mid point M between NE and E and test if this mid point, say M lies above or below this line path.

⇒ This method is more easily applied to other conics.

$$y = \left(\frac{dy}{dx}\right) * x + B$$

$$dy = y_1 - y_0 \;,\; dx = x_1 - x_0$$

The same equation can be written as :

$$f(x,y) = a*x + b*y + C = 0$$

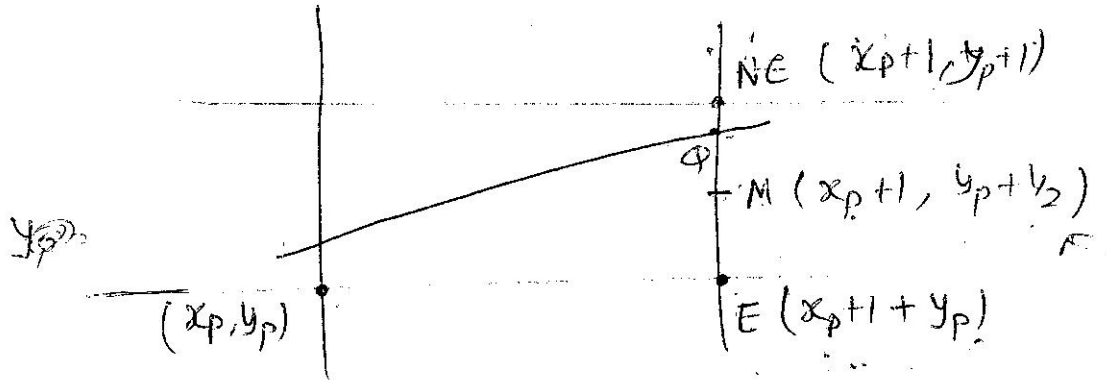
$$m = -a/b$$

$$f(x,y) = dy*x + (-dx)*y + B*dx = 0$$

$$a = dy \;,\; b = -dx \;,\; c = B*dx$$

if $f(x,y) > 0$, Midpoint M lies below the line (NE)

if $f(x,y) < 0$, Midpoint M lies above the line (E)



$$d = f(x_{p+1}, y_p + 1/2) = a(x_p + 1) + b(y_p + 1/2) + c$$

(i) if $d > 0 \rightarrow$ choose NE

$$(\Delta d)_{NE} = dy - dx$$

(ii) if $d < 0 \rightarrow$ choose E

$$(\Delta d)_E = dy$$

Now, we need initial decision variable,

$$d_{start} = f(x_0 + 1, y_0 + 1/2)$$

$$= a(x_0 + 1) + b(y_0 + 1/2) + c$$

$$= (ax_0 + by_0 + c) + a + b/2$$

$$= f(x_0, y_0) + a + b/2$$

$f(x_0, y_0) \rightarrow$ Initial point so $f(x_0, y_0) = 0$

$$d_{start} = a + \frac{b}{2}$$

$$= dy - \frac{dx}{2}$$

Multiply by 2

$$d_{start} = 2dy - dx$$

$$(\Delta d)_{NE} = 2dy - 2dx$$

$$(\Delta d)_E = 2dy$$

→ Same As in Bresenham's Algo

3.3 SCAN-CONVERTING A CIRCLE

A circle is a symmetrical figure. Any circle-generating algorithm can take advantage of the circle's symmetry to plot eight points for each value that the algorithm calculates. Eight-way symmetry is used by reflecting each calculated point around each 45° axis. For example, if point 1 in Fig. 3.4 were calculated with a circle algorithm, seven more points could be found by reflection. This reflection is accomplished by reversing the x, y coordinates as in point 2, reversing the x, y coordinates and reflecting about the y axis as in point 3, reflecting about the y axis as in point 4, switching the signs of x and y as in point 5, reversing the x, y coordinates, reflecting about the y axis and reflecting about the x axis as in point 6, reversing the x, y coordinates and reflecting about the y axis as in point 7, and reflecting about the x axis as in point 8.

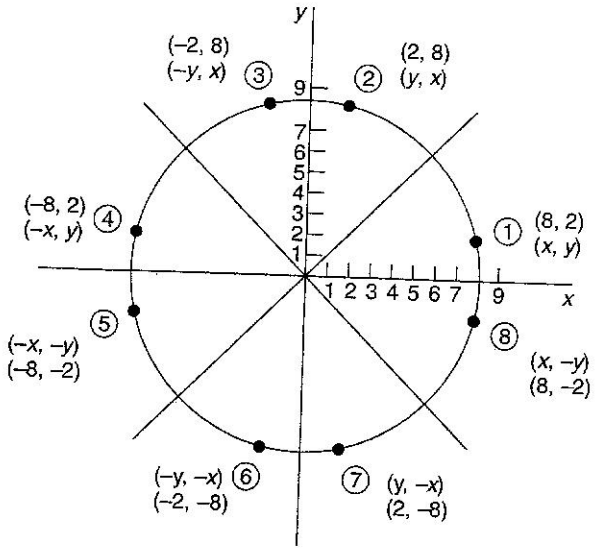


Fig. 3.4 *Eight-way Symmetry of a Circle*

To summarize:

$$P_1 = (x, y) \qquad P_5 = (-x, -y)$$
$$P_2 = (y, x) \qquad P_6 = (y, -x)$$
$$P_3 = (-y, x) \qquad P_7 = (y, -x)$$
$$P_4 = (-x, y) \qquad P_8 = (x, -y)$$

Defining a Circle

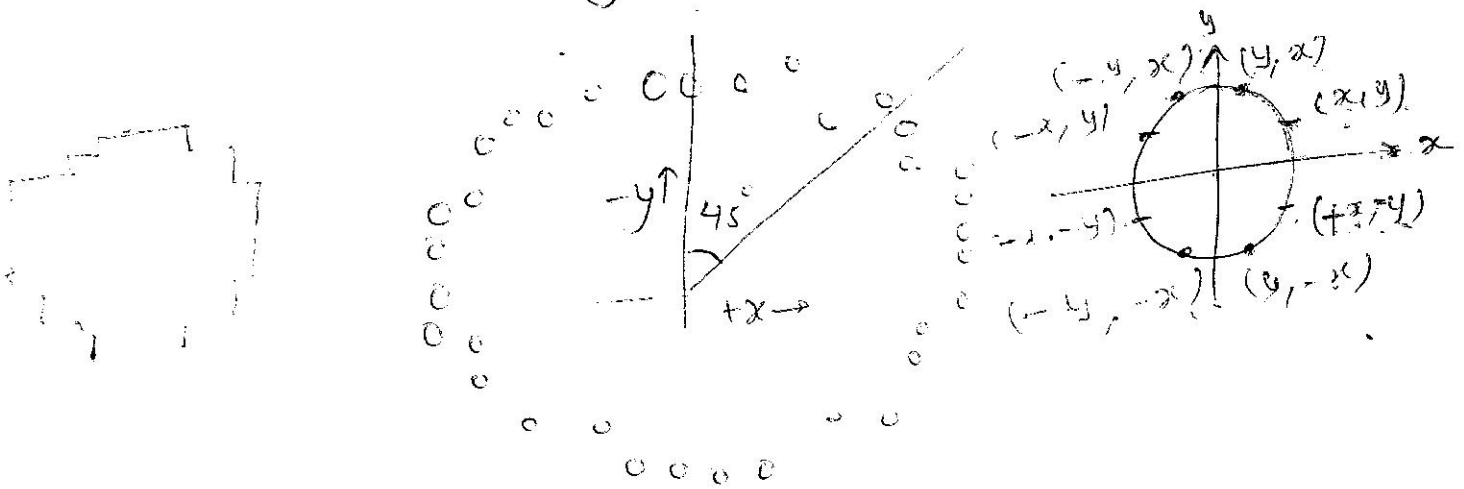
There are two standard methods of mathematically defining a circle centered at the origin. The first method defines a circle with the second-order polynomial equation (see Fig. 3.5)

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

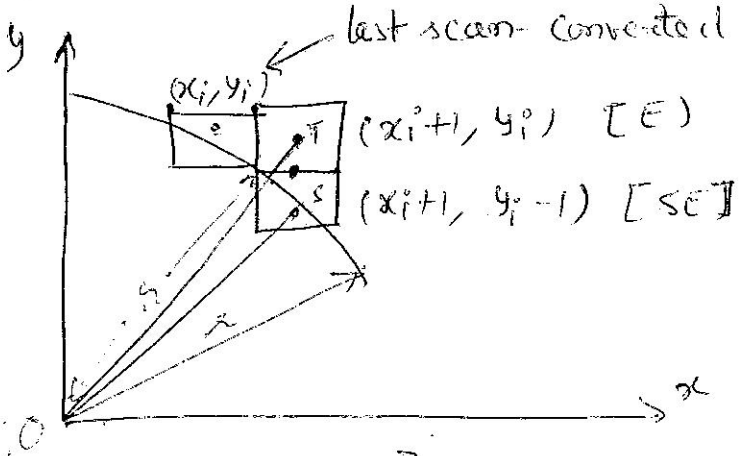
where $x = x$ coordinate
$y = y$ coordinate
r = radius of the circle

Bresenham's Circle Algorithm:

⇒ If the 8-way symmetry of a circle is used to generate a circle, points will only have to be generated through a 45° angle, And if points are generated from 90° to 45°, moves will be made only in the +x and -y directions.



⇒ The best Approximation of the true circle will be described by those pixels in the raster that fall the least distance.



$D(T) = (\text{origin to } T)^2 - (\text{origin to True circle})^2$

$D(T) = (x_i+1)^2 + y_i^2 - r^2$

$D(S) = (\text{origin to } S)^2 - (\text{origin to True circle})^2$

$= (x_i+1)^2 + (y_i-1)^2 - r^2$

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

Midpoint Circle Algorithm:-

⇒ Very similar to Bresenham's Approach.

⇒ $f(x,y) = x^2 + y^2 - r^2$

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

⇒ Now consider the coordinates of the point halfway between pixel T and pixel S $(x_i + 1, y_i - \frac{1}{2})$. This is called the midpoint and we use it to define decision parameter

$$p_i = f(x_i + 1, y_i - \tfrac{1}{2}) = (x_i + 1)^2 + (y_i - \tfrac{1}{2})^2 - r^2$$

(i) p_i is -ve, midpoint is inside the circle, we choose 'T'

(ii) p_i is zero or +ve, midpoint is outside the circle, we choose 'S'

$$p_{i+1} = \begin{cases} p_i + 2(x_i + 1) + 1 & \text{if } (p_i < 0, \text{'T'}) \\ p_i + 2(x_i + 1) + 1 - 2(y_i - 1) & \text{if } (p_i \geq 0, \text{'S'}) \end{cases}$$

$$\Rightarrow p_{i+1} = \begin{cases} p_i + 2x_i + 3 & \text{if } p_i < 0 \\ p_i + 2(x_i - y_i) + 5 & \text{if } p_i \geq 0 \end{cases}$$

for $(0, r)$

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

⇒ This is not really integer computation. However, when 'r' is an integer we can simply set $p_i = 1 - r$.

Decision Variable $d_i = D(T) + D(S)$

$$d_i = 2(x_i+1)^2 + y_i^2 + (y_i-1)^2 - 2r^2$$

when $d_i < 0$, T is chosen

when $d_i > 0$, S is chosen

for the next step.

$$d_{i+1} = 2(x_{i+1}+1)^2 + y_{i+1}^2 + (y_{i+1}-1)^2 - 2r^2$$

Hence,

$$d_{i+1} - d_i = 2(x_{i+1}+1)^2 + (y_{i+1})^2 + (y_{i+1}-1)^2 - 2(x_i+1)^2 - y_i^2 - (y_i-1)^2$$

1) If T is chosen ($d_i < 0$), then $y_{i+1} = y_i$, so

$$d_{i+1} = d_i + 4x_i + 6$$

2) If S is chosen ($d_i \geq 0$), then $y_{i+1} = y_i - 1$, so

$$d_{i+1} = d_i + 4(x_i - y_i) + 10$$

Hence

$$d_{i+1} = \begin{cases} d_i + 4x_i + 6 & \text{if } d_i < 0 \\ d_i + 4(x_i - y_i) + 10 & \text{if } d_i \geq 0 \end{cases}$$

we set $(0, r)$ as starting pixel coordinates

$$d_1 = 2(0+1)^2 + r^2 + (r-1)^2 - 2r^2 = 3 - 2r$$

Given radius = 10, draw a circle using Mid-point Algo.

$x = 0,\ y = 10,\ p_i = 1 - r = -9$

$2x = 0,\ 2y = 20$

initial $(0, r) \rightarrow (0, 10)$

$p_i < 0$ so we choose 'T' $(x_p + 1, y_p) \rightarrow (1, 10)$

and new decision variable

$$p_i = p_i + 2x + 3$$

$$= -9 + 2 \times 0 + 3$$

$$= -6$$

Again $p_i < 0$, so again $(x_{p+1}, y_p) \Rightarrow (2, 10)$

Algorithm 4.7 Plot of circle with Mid-point Method

Input (h,k) and r where (h,k): centre of circle
r : radius of circle

$x = 0$

$y = r$

$d = 1-r$

while $(x \leq y)$

plotpixel $(x + h, y + k)$ /* Plot eight points—each point
plotpixel $(-x + h, y + k)$ corresponding to one octant
plotpixel $(x + h, -y + k)$
plotpixel $(-x + h, -y + k)$
plotpixel $(y + h, x + k)$
plotpixel $(-y + h, x + k)$
plotpixel $(y + h, -x + k)$
plotpixel $(-y + h, -x + k)$

If$(d < 0)$, then

$d = d + 2*x + 3$ /* upper pixel

else

$d = d + 2*(x - y)+5$ /* lower pixel

$y = y - 1$

end if

$x = x + 1$

end while

stop

4.5 ELLIPSE

An ellipse can be divided into four equal parts. So if one part (or quadrant) can be generated then other three parts can be easily replicated. Therefore, we need to compute only one part (or quadrant) to determine the ellipse completely, as shown in Fig. 4.23 with set of four symmetrical points.

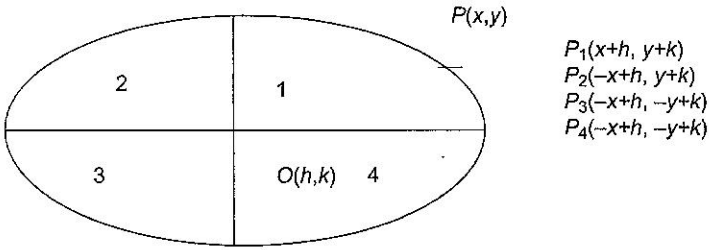


FIGURE 4.23 Four-way Symmetry of Ellipse

4.5.1 Digital Digitizer Analyzer (DDA)

There are two DDA techniques for scan conversion of ellipse, namely:

1. Direct or Polynomial Approach
2. Parametric or Trigonometric Approach

4.5.1.1 Direct or Polynomial Approach

An ellipse can be represented as second degree polynomial equation

$$\frac{(x-h)^2}{a^2} + \frac{(y-k)^2}{b^2} = 1 \tag{4.9}$$

The equation (4.9) can be simplified as:

$$y = k \pm b\sqrt{1-\frac{(x-h)^2}{b^2}}$$

To draw an ellipse value of x is incremented in units of 1 from $h–a$ to $h + a$, the equation 4.9 is used to solve the value of y. This method is time-consuming, since multiplications and square-root operations are involved at each step. However, we can improve the process by taking advantage of symmetry in an ellipse. Therefore, we need to compute only first quadrant to determine the ellipse completely, as shown in Fig. 4.23. With this method plots are generated from 0° to 45°, each x coordinate is incremented by 1 from 0 to **a** and each value of y is found by evaluating $b\sqrt{1-\frac{x*x}{a*a}}$.

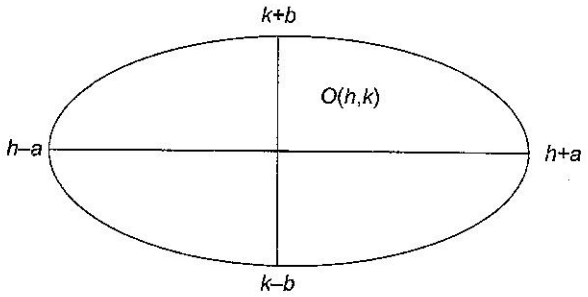


FIGURE 4.24

Algorithm 4.8 Plot of Ellipse with Polynomial Approach

Input (h,k), $r\,(h,k)$: Centre of ellipse
$2a$: Length of Major axis
$2b$: Length of Minor axis

$x$start = 0

$x$end = a

while ($x \leq$ xend)

$y = b$ * square root of $(1 - x^*x/a^*a)$

plotpixel $(x + h, y + k)$
plotpixel $(-x + h, y + k)$
plotpixel $(x + h, -y + k)$
plotpixel $(-x + h, -y + k)$

$x = x + 1$

end while
stop

4.5.1.2 *Parametric or Trigonometric Approach*

Parametric polar representation of an ellipse

$$x = h + a\cos(\theta) \qquad (4.10$$

$$y = k + b\sin(\theta) \qquad (4.11$$

where, (h, k) centre of the ellipse, $2a$ be the length of major axis and $2b$ be the length of minor axis, and θ is measured in radian 0 to 2π.

In this method value of θ is incremented from 0 to 2π, then values of x and y are calculated from equations 4.10 and 4.11, respectively. This method is time-consuming since multiplications and trigonometric functions are involved at each step. However, we can improve the process by taking advantage of symmetry in an ellipse. Therefore, we need to compute only first quadrant to determine the ellipse completely, as shown in Fig. 4.23. With this method plots are generated from 0° to 45°, each x and y value is computed by evaluated equations 4.10 and 4.11, respectively with theta *incerment thetainc* (say).

theta = theta + thetainc
end while
stop

Note: If the increment is in clockwise direction, thetainc is taken to be *negat*

4.5.2 Mid-point Method

Ellipse is described by the equation:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$

where, $2a$ = length of major ax
$2b$ = length of minor ax

The mid-point technique applied to line and circle can also be applied to ellipse. To simplify the analysis of the algorithm, we consider only arc of ellipse with centre at O(0,0) that lies in first quadrant, since the other three quadrants can be drawn by symmetry.

From coordinate geometry, we get

$$F(x,y) = b^2x^2 + a^2y^2 - a^2b^2 \begin{cases} < 0 \text{ point } (x, y) \text{ lies inside the ellipse.} \\ = 0 \text{ point } (x, y) \text{ lies on the ellipse.} \\ > 0 \text{ point } (x, y) \text{ lies outside the ellipse.} \end{cases}$$

Working of the Method

First quadrant is temporarily divided into two regions:

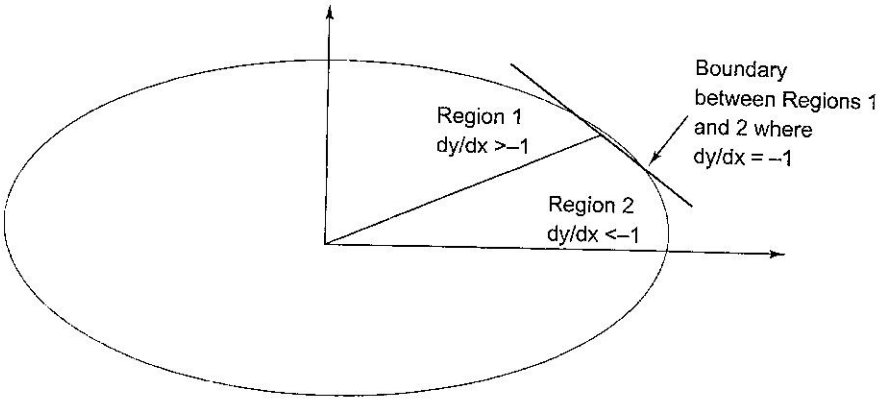


FIGURE 4.26 Representation of Two Regions in First Quadrant

Region 1

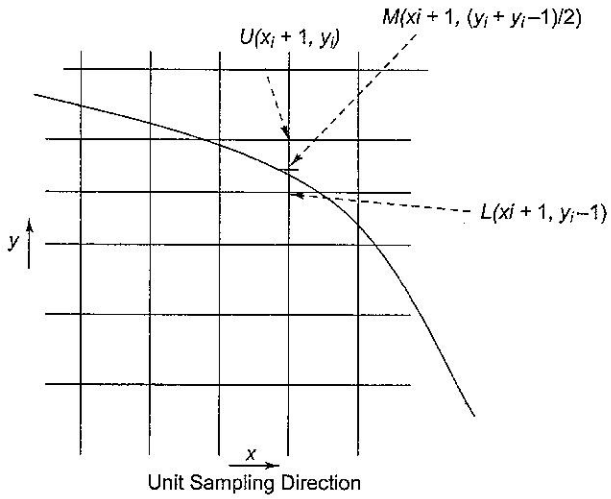


FIGURE 4.27 Choosing Pixels in Region 1 with Mid-Point Method

Region 2

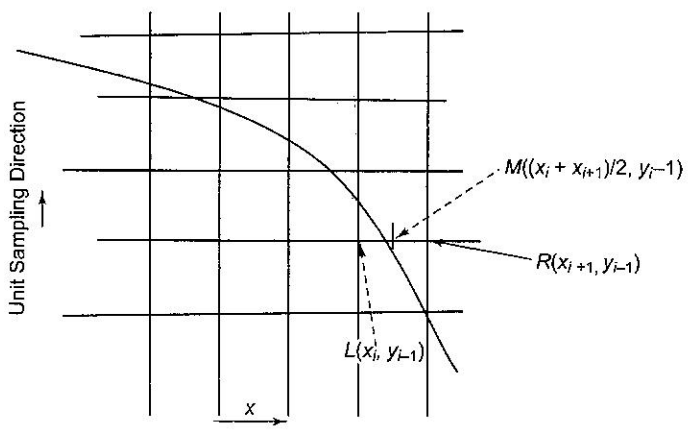


FIGURE 4.28 Choosing Pixels in Region 2 with Mid-Point Method

The checking is done by evaluating $f(x_i + 1, y_i - 1/2)$ and checking the sign.

Let, $$di = f\left(x_i + 1, y_i - \frac{1}{2}\right) = b^2(x_i + 1)^2 + a^2\left(y_i - \frac{1}{2}\right) - a^2b^2$$

Similarly, $$d_{i+1} = f\left(x_{i-1} + 1, y_{i+1} - \frac{1}{2}\right) = b^2(x_{i+1} + 1)^2 + a^2\left(y_{i+1} - \frac{1}{2}\right)^2 - a^2b^2$$

$$\Rightarrow \quad d_{i+1} - d_i = b^2\left[(x_{i+1} + 1)^2 - (x_i + 1)^2 - (x_i + 1)^2\right] + a^2\left[\left(y_{i+1} - \frac{1}{2}\right)^2 - \left(y_i - \frac{1}{2}\right)^2\right]$$

Sectors

A sector is scan-converted by using any of the methods of scan-converting an arc and then scan-converting two lines from the center of the arc to the endpoints of the arc.

For example, assume that a sector whose center is at point (h, k) is to be scan-converted. First, scan-convert an arc from θ_1 to θ_2. Next, a line would be scan-converted from (h, k) to $(r \cos \theta_1 + h, r \sin \theta_1 + k)$. A second line would be scan-converted from (h, k) to $(r \cos \theta_2 + h, r \sin \theta_2 + k)$.

3.6 SCAN-CONVERTING A POLYGON

Generally closed contours are represented by a cluster of polygons. Thus, to fill or to draw a contour it is necessary to have methods for filling polygons. There are many algorithms for filling polygons. It may be observed that adjacent pixels are likely to have the same characteristics in polygons. This property is called *spatial coherence*. Adjacent pixels on a scan line are likely to have the same characteristics. This is called *scan line coherence*. These two properties are useful in scan converting polygons.

Parity Scan Conversion Algorithm

It may be observed that the scan lines intersect polygons in pairs (see Fig. 3.16). This can be used to set a flag called a *parity bit* to determine whether a particular pixel on a scan line is inside or outside of the polygon. Initially the parity bit is set to zero to indicate that the scan line is outside the polygon. When the scan line intersects the polygon the parity bit is set to 1. This indicates that the scan line is now inside the polygon. At the next intersection the parity bit is set to 0 indicating that the scan line is out of the polygon. When the parity bit is 1 the pixels are set to the polygon colour. When the parity bit is 0 the pixels are set to background colour. (see Fig. 3.17)

Ordered Edge List Algorithm for Polygon Filling

There are many algorithms for polygon filling and solid area scan converting the polygons. We mention here an ordered edge list algorithm which depends on sorting the intersecting points of the polygon edge with the scan line in the scan line order. Here we use half interval scan lines. The algorithm is as given on the next page.

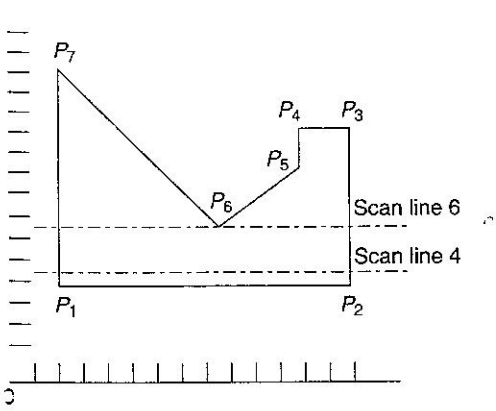


Fig. 3.16

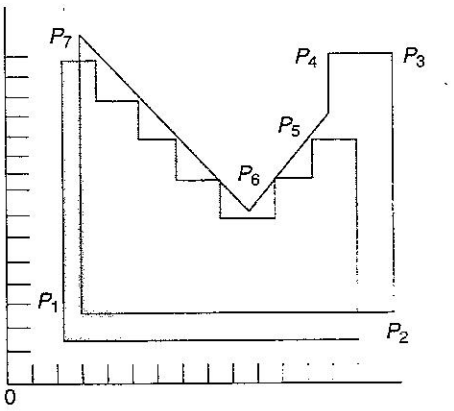


Fig. 3.17

- 1: Find the intersections of the half interval scan lines with each polygon edge using any line drawing algorithm.
- 2: Collect each intersection point $(x, y + 1/2)$ in a list.
- 3: Sort the list by scan line and increasing order of x value on the scan line.
- 4: Choose pairs of elements (x_1, y_1) and (x_2, y_2) from the sorted list.
- 5: Activate pixels on the scan line Y for integer values of x.
  Such that $x_1 \leq x + 1/2 \leq x_2$.

Example 3.1

Consider a polygon whose vertices are P_1 (2, 2), P_2 (8, 2), P_3 (8, 6), P_4 (5, 3), P_5 (1, 7). Intersections with the half interval scan lines are

Scan line 2.5: (8, 2.5), (1, 2.5)
Scan line 3.5: (8, 3.5), (1, 3.5), (5.5, 3.5), (4.5, 3.5)
Scan line 4.5: (8, 4.5), (1, 4.5), (6.5, 4.5), (3.5, 4.5)
Scan line 5.5: (8, 5.5), (1, 5.5), (7.5, 5.5), (2.5, 5.5)
Scan line 6.5: (1.5, 6.5), (1, 6.5)
Scan line 7.5: None

The complete list sorted in scan line order from the top to the bottom (i.e. from scan line 7.5 to 2.5) and then from left to right is

(1, 6.5), (1.5, 6.5), (1, 5.5), (2.5, 5.5), (7.5, 5.5), (8, 5.5), (1, 4.5), (3.5, 4.5)
(6.5, 4.5), (8, 4.5), (1, 3.5), (4.5, 3.5), (5.5, 3.5), (8, 3.5), (1, 2.5), (8, 2.5).

Extracting pair of intersections from the list and applying the algorithm give above yields the pixel activation list

(1, 6)
(1, 5), (2, 5), (7, 5)
(1, 4), (2, 4), (3, 4), (6, 4), (7, 4)
(1, 3), (2, 3), (3, 3), (4, 3), (5, 3), (6, 3), (7, 3)
(1, 2), (2, 2), (3, 2), (4, 2), (5, 2), (6, 2), (7, 2)

The results are shown in Fig. 3.18

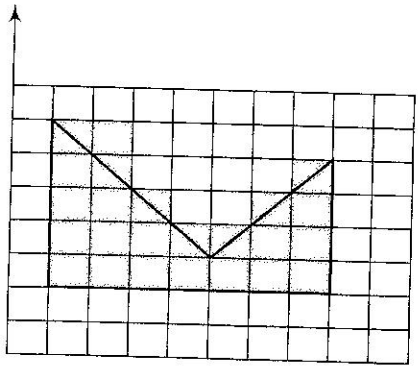


Fig. 3.18

Polygon Inside Test

Filling polygons can be done by setting the pixels inside the polygon as well as those on the boundary. This requires us to determine whether or not a given point is inside the polygon. There are two methods for doing this, namely, the even-odd method and the winding number method.

Even-odd Method

Construct a line segment between a point in the question and any point outside the polygon. Count the number of intersections of this line segment with each side of the polygon. If the number of intersections is even then the point is outside and if it is odd then the point lies inside (see Fig. 3.19).

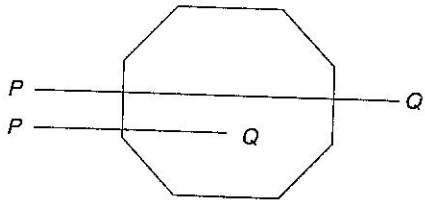


Fig. 3.19

However if the point of intersection is also a vertex point then counting the number of intersections requires some caution. The following rule may be followed in such cases. Count the number intersections as an even number if these two points lie on the same side of the constructed line otherwise count it as single intersection (see Fig. 3.20).

Now it is required to know how to find a point outside the polygon. Take any point $Q(x, y)$ with x coordinate less than minimum of all the x-coordinates vertices of the polygon or greater than the maximum of all the x-coordinates of the vertices of the polygon or any point with y-coordinate with less than the minimum of all the y-coordinates of the vertices of the polygon or greater than the maximum of all the y-coordinates of all vertices of the polygon. It is easy to observe that (x, y) lies outside the polygon.

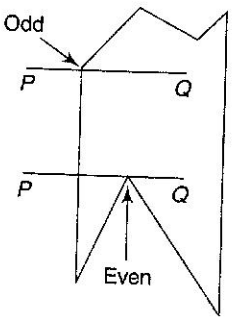


Fig. 3.20

Winding Number Method

This is an alternative method for finding the interior points of any polygon. Instead of just counting the number of intersections as in the case of even-odd method, we give each side of the polygon a number called *winding number*. The total of this winding number is called the net winding. If the net winding is zero then the point is outside otherwise it is inside.

Construct a line segment between the point in question and any point outside the polygon. Find all the sides which cross this line segment. There are two ways for this to happen, start below the line, cross it, and end above the line or start above the line, cross it, and end below the line (i.e. the first y-coordinate value is less than the second y-coordinate value or in the second case the first y-coordinate value is greater than the second y-coordinate value). In the first case the winding number of the side is given as -1 while in the second case the winding number is 1 as shown Fig. 3.21. Any other sides which do not cross this line have their winding numbers zero. The sum of the winding numbers of all the sides is the net winding. If the net winding is zero the point is outside, otherwise it is inside the polygen. These two methods give rise to different results in case of self intersecting polygons as shown in Fig. 3.22.

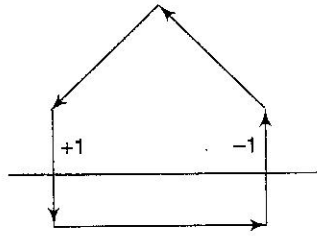


Fig. 3.21

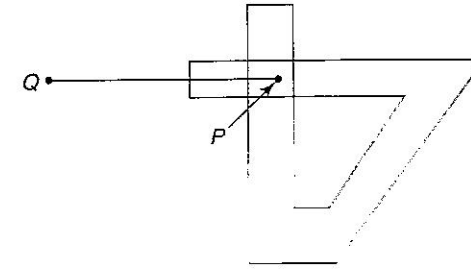


Fig. 3.22

Scan-converting a Rectangle

A rectangle whose sides are parallel to the coordinate axes may be constructed if the locations of two vertices are known [see Fig. 3.23(a)]. The remaining corner points are then derived [see Fig. 3.23(b)]. Once the vertices are known, the four sets of coordinates are sent to the line routine and the rectangle is scan-converted. In the case of the rectangle shown in Figs 3.23(a) and 3.23(b),

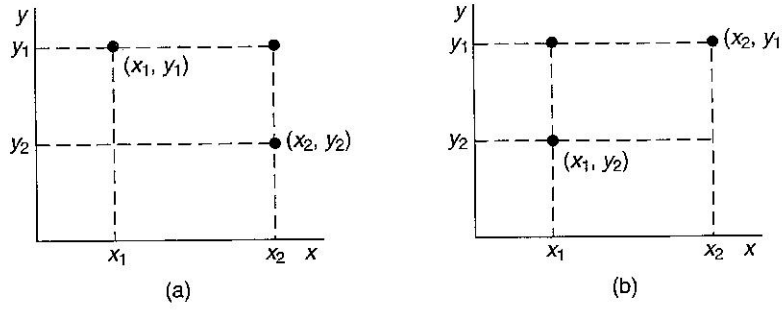


Fig. 3.23

lines would be drawn as follows: line (x_1, y_1) to (x_1, y_2); line (x_1, y_2) to (x_2, y_2); line (x_2, y_2) to (x_2, y_1); and line (x_2, y_1) to (x_1, y_1).

3.7 REGION FILLING

Region filling is the process of "coloring in" a definite image area or region. Regions may be defined at the pixel or geometric level. At the pixel level, we describe a region either in terms of the bounding pixels that outline it or as the totality of pixels that comprise it (see Fig. 3.24). In the first case the region is called boundary-defined and the collection of algorithms used for filling such a region are collectively called *boundary-fill algorithms*. The other type of region is called an interior-defined region and the accompanying algorithms are called *flood-fill algorithms*. At the geometric level a region is defined or enclosed by such abstract contouring elements as connected lines and curves. For example, a polygonal region, or a filled polygon, is defined by a closed polyline, which is a polyline (i.e., a series of sequentially connected lines) that has the end of the last line connected to the beginning of the first line.

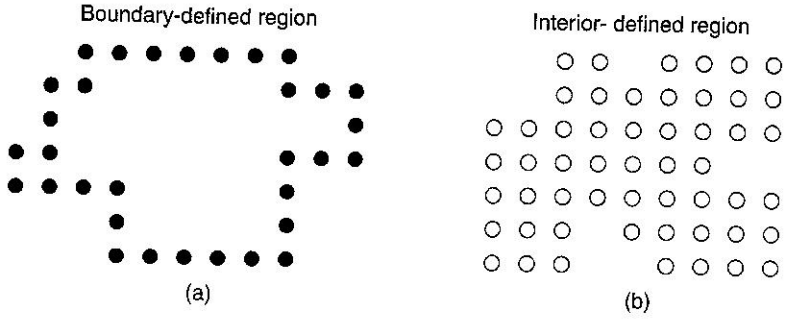


Fig. 3.24

4-Connected versus 8-Connected

An interesting point here is that, while a geometrically defined contour clearly separates the interior of a region from the exterior, ambiguity may arise when an outline consists of discrete pixels in the image space. There are two ways in which pixels are considered connected to each other to form a "continuous" boundary. One method is called 4-connected, where a pixel may have up to four neighbors [see Fig. 3.25(a)]; the other is called 8-connected, where a pixel may have up

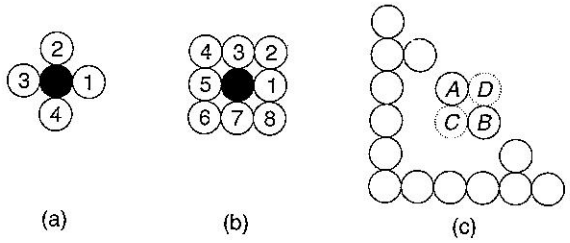


Fig. 3.25 *4-connected vs. 8-connected Pixels*

eight neighbors [see Fig. 3.25(b)]. Using the 4-connected approach, the pixels in Fig. 3.25(c) do not define a region since several pixels such as A and B are not connected. However using the 8-connected definition we identify a triangular region.

We can further apply the concept of connected pixels to decide if a region is connected to another region. For example, using the 8-connected approach, we do not have an enclosed region in Fig. 3.25(c) since "interior" pixel C is connected to "exterior" pixel D. On the other hand, if we use the 4-connected definition we have a triangular region since no interior pixel is connected to the outside.

Note that it is not a mere coincidence that the figure in Fig. 3.25(c) is a boundary-defined region when we use the 8-connected definition for the boundary pixels and the 4-connected definition for the interior pixels. In fact, using the same definition for both boundary and interior pixels would simply result in contradiction. For example, if we use the 8-connected approach we would have pixel A connected to pixel B (continuous boundary) and at the same time pixel C connected to pixel D (discontinuous boundary). On the other hand, if we use the 4-connectd definition we would have pixel A disconnected from pixel B (discontinuous boundary) and at the same time pixel C disconnected from pixel D (continuous boundary).

Boundary-fill Algorithm

This is a recursive algorithm that begins with a starting pixel, called a seed, inside the region. The algorithm checks to see if this pixel is a boundary pixel or has already been filled. If the answer is no, it fills the pixel and makes a recursive call to itself using each and every neighboring pixel as a new seed. If the answer is yes, the algorithm simply returns to its caller.

This algorithm works elegantly on an arbitrarily shaped region by chasing and filling all non-boundary pixels that are connected to the seed, either directly or indirectly through a chain of neighboring relations. However, a straightforward implementation can take time and memory to execute due to the potentially high number of recursive calls, especially when the size of the region is relatively large.

Variations can be made to limit the number of recursive calls by structuring the order in which neighboring pixels are processed. For example, we can first fill pixels to the left and right of the seed on the same scan line until boundary pixels are hit (something that can be done using a loop control structure). We then inspect each pixel above and below the line just drawn (which can also be done with a loop) to see if it can be used as a new seed for the next horizontal line to fill. This way the number of recursive calls at any particular time is merely N when the current line is N scan lines away from the initial seed.

Flood-fill Algorithm

This algorithm also begins with a seed (starting pixel) inside the region. It checks to see if the pixel has the region's original color. If the answer is yes, it fills the pixel with a new color and uses each of the pixel's neighbors as a new seed in a recursive call. If the answer is no, it returns to the caller.

This method shares great similarities in its operating principle with the boundary-fill algorithm. It is particularly useful when the region to be filled has no uniformly colored boundary. On the

3.9 ALIASING EFFECTS

Scan conversion is essentially a systematic approach to mapping objects that are defined in continuous space to their discrete approximation. The various forms of distortion that result from this operation are collectively referred to as the aliasing effects of scan conversion.

Staircase

A common example of aliasing effects is the staircase or jagged appearance we see when scan-converting a primitive such as a line or a circle. We also see the stair steps or "jaggies" along the border of a filled region.

Unequal Brightness

Another artifact that is less noticeable is the unequal brightness of lines of different orientation. A slanted line appears dimmer than a horizontal or vertical line, although all are presented at the same intensity level. The reason for this problem can be explained using Fig. 3.31, where the pixels on the horizontal line are placed one unit apart, whereas those on the diagonal line are approximately 1.414 units apart. This difference in density produces the perceived difference in brightness.

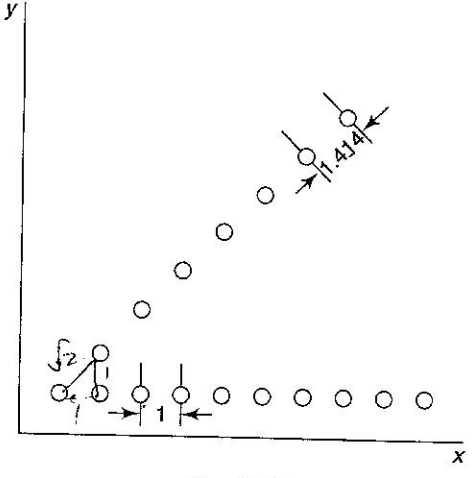


Fig. 3.31

The Picket Fence Problem

The picket fence problem occurs when an object is not aligned with, or does not fit into, the pixel grid properly.

Figure 3.32(a) shows a picket fence where the distance between two adjacent pickets is not a multiple of the unit distance between pixels. Scan-converting it normally into the image space will result in uneven distances between pickets since the endpoints of the pickets will have to be snapped to pixel coordinates [see Fig. 3.32(b)]. This is sometimes called *global aliasing*, as the overall length of the picket fence is approximately correct. On the other hand, an attempt to maintain equal spacing will greatly distort the overall length of the fence [see Fig. 3.32(c)]. This is sometimes called *local aliasing*, as the distances between pickets are kept close to their true distances.

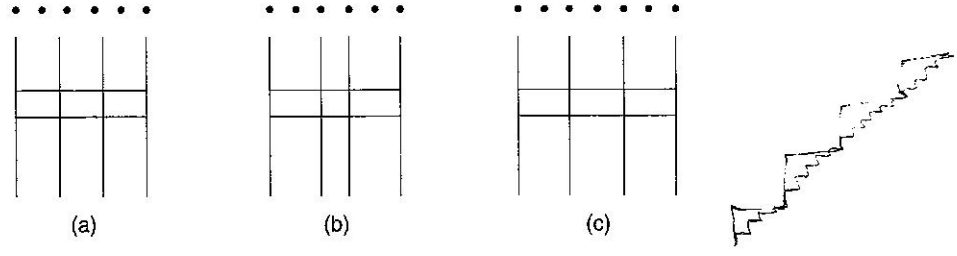


Fig. 3.32 *The Picket Fence Problem*

Another example of such a problem arises with the outline font. Suppose we want to scan-convert the uppercase character "E" in Fig. 3.33(a) from its outline description to a bitmap consisting of pixels inside the region defined by the outline. The result in Fig. 3.33(b) exhibits both asymmetry (the upper arm of the character is twice as thick as the other parts) and dropout (the middle arm is absent). A slight adjustment and/or realignment of the outline can lead to a reasonable outcome [see Fig. 3.33(c)].

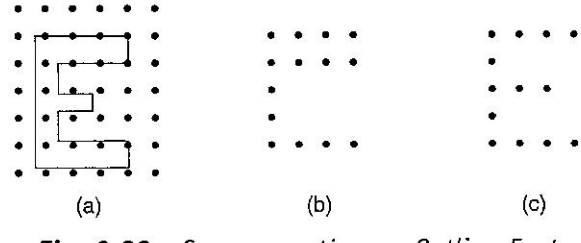


Fig. 3.33 *Scan-converting an Outline Font*

3.10 ANTI-ALIASING

Most aliasing artifacts, when appear in a static image at a moderate resolution, are often tolerable, and in many cases, negligible. However, they can have a significant impact on our viewing experience when left untreated in a series of images that animate moving objects. For example, a line being rotated around one of its endpoints becomes a rotating escalator with length-altering steps. A moving object with small parts or surface details may have some of those features intermittently change shape or even disappear.

Although increasing image resolution is a straightforward way to decrease the size of many aliasing artifacts and alleviate their negative impact, we pay a heavy price in terms of system resource (going from $W \times H$ to $2W \times 2H$ means quadrupling the number of pixels) and the results are not always satisfactory. On the other hand, there are techniques that can greatly reduce aliasing artifacts and improve the appearance of images without increasing their resolution. These techniques are collectively referred to as anti-aliasing techniques.

Some anti-aliasing techniques are designed to treat a particular type of artifact. For instance, an outline font can be associated with a set of rules or hints to guide the adjustment and realignment that is necessary for its conversion into bitmaps of relatively low resolution. An example of such approach is called the TrueType font.

Pre-filtering and Post-filtering

Pre-filtering and post-filtering are two types of general-purpose anti-aliasing techniques. The concept of filtering originates from the field of signal processing, where true intensity values are continuous signals that consist of elements of various frequencies. Constant intensity values that correspond to a uniform region are at the low end of the frequency range. Intensity values that change abruptly and correspond to a sharp edge are at the high end of the spectrum. In order to lessen the jagged appearance of lines and other contours in the image space, we seek to smooth out sudden intensity changes, or in signal-processing terms, to filter out the high frequency components. A pre-filtering technique works on the true signal in the continuous space to derive proper values for individual pixels (filtering before sampling), whereas a post-filtering technique takes discrete samples of the continuous signal and uses the samples to compute pixel values (sampling before filtering).

Area Sampling

Area sampling is a pre-filtering technique in which we superimpose a pixel grid pattern onto the continuous object definition. For each pixel area that intersects the object, we calculate the percentage of overlap by the object. This percentage determines the proportion of the overall intensity value of the corresponding pixel that is due to the object's contribution. In other words, the higher the percentage of overlap, the greater influence the object has on the pixel's overall intensity value.

In Fig. 3.34(a) a mathematical line shown in dotted form is represented by a rectangular region that is one pixel wide. The percentage of overlap between the rectangle and each intersecting pixel is calculated analytically. Assuming that the background is black and the line is white, the percentage values can be used directly to set the intensity of the pixels [see Fig. 3.34(b)]. On the other hand,

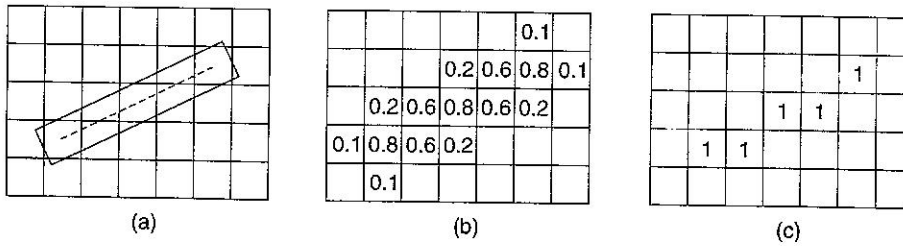


Fig. 3.34 *Area Sampling*

had the background been gray (0.5, 0.5, 0.5) and the line green (0, 1, 0), each blank pixel in the grid would have had the background gray value and each pixel filled with a fractional number f would have been assigned a value of $[0.5(1-f), 0.5(1-f)+f, 0.5(1-f)]$—a proportional blending of the background and object colors.

Although the resultant discrete approximation of the line in Fig. 3.34(b) takes on a blurry appearance, it no longer exhibits the sudden transition from an on pixel to an off pixel and vice versa, which is what we would get with an ordinary scan-conversion method [see Fig. 3.34(c)]. This tradeoff is characteristic of an anti-aliasing technique based on filtering.

Super Sampling

In this approach we subdivide each pixel into subpixels and check the position of each subpixel in relation to the object to be scan-converted. The object's contribution to a pixel's overall intensity value is proportional to the number of subpixels that are inside the area occupied by the object. Figure 3.35(a) shows an example where we have a white object that is bounded by two slanted lines on a black background. We subdivide each pixel into nine (3×3) subpixels. The scene is mapped to the pixel values in Fig. 3.35(b). The pixel at the upper right corner, for instance, is assigned $\frac{7}{9}$ since seven of its nine subpixels are inside the object area. Had the object been red (1, 0, 0) and the background light yellow (0.5, 0.5, 0), the pixel would have been assigned $\left(1\times\frac{7}{9}+0.5\times\frac{2}{9}, 0.5\times\frac{2}{9}, 0\right)$, which is $\left(\frac{8}{9}, \frac{1}{9}, 0\right)$.

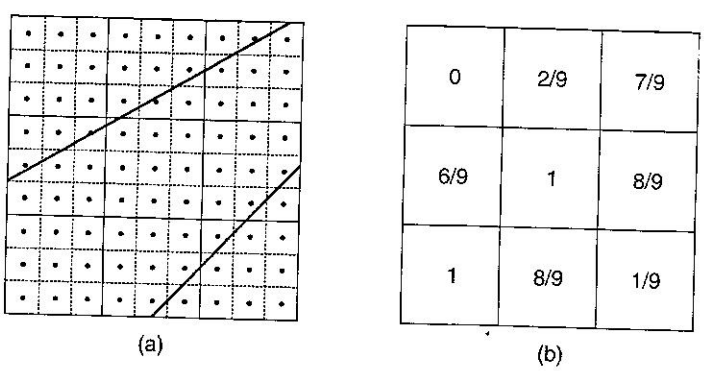


Fig. 3.35 *Super Sampling*

Super sampling is often regarded as a post-filtering technique since discrete samples are first taken and then used to calculate pixel values. On the other hand, it can be viewed as an approximation to the area sampling method since we are simply using a finite number of values in each pixel area to approximate the accurate analytical result.

Low Pass Filtering

This is a post-filtering technique in which we reassign each pixel a new value that is a weighted

average of its original value and the original values of its neighbors. A low pass filter in the form of a $(2n + 1) \times (2n + 1)$ grid, where $n \geq 1$, holds the weights for the computation. All weight values in a filter should sum to one. An example of a 3×3 filter is given in Fig. 3.36(a).

0	1/8	0
1/8	1/2	1/8
0	1/8	0

(a)

0	0	0	0	0	1/8	0
0	0	0	1/8	1/4	1/2	1/8
0	1/8	1/4	5/8	5/8	1/4	0
1/8	5/8	5/8	1/4	1/8	0	0
0	1/8	1/8	0	0	0	0

(b)

Fig. 3.36 *Low Pass Filtering*

To compute a new value for a pixel, we align the filter with the pixel grid and center it at the pixel. The weighted average is simply the sum of products of each weight in the filter times the corresponding pixel's original value. The filter shown in Fig. 3.36(a) means that half of each pixel's original value is retained in its new value, while each of the pixel's four immediate neighbors contributes one eighth of its original value to the pixel's new value. The result of applying this filter to the pixel values in Fig. 3.6(a) is shown in Fig. 3.36(b).

A low pass filter with equal weights, sometimes referred to as a box filter, is said to be doing neighborhood averaging. On the other hand, a filter with its weight values conforming to a two-dimensional Gaussian distribution is called a Gaussian filter.

Pixel Phasing

Pixel phasing is a hardware-based anti-aliasing technique. The graphics system in this case is capable of shifting individual pixels from their normal positions in the pixel grid by a fraction (typically 1/4 and 1/2) of the unit distance between pixels. By moving pixels closer to the true line or other contour, this technique is very effective in smoothing out the stair steps without reducing the sharpness of the edges.