

Graphics PRIMITIVES

■ Chapter- 3 & Unit-2

Objectives

- Introduction to Primitives
- Points & Lines
- Line Drawing Algorithms
 - Digital Differential Analyzer (DDA)
 - Bresenham's Algorithm
 - Mid-Point Algorithm
- Circle Generating Algorithms
 - Properties of Circles
 - Bresenham's Algorithm
 - Mid-Point Algorithm
- Ellipse Generating Algorithms
 - Properties of Ellipse
 - Bresenham's Algorithm
 - Mid-Point Algorithm
- Fill Area Primitives
- Boundary Fill
- Flood Fill algorithm
- Character generation
- Antialiasing methods

Introduction

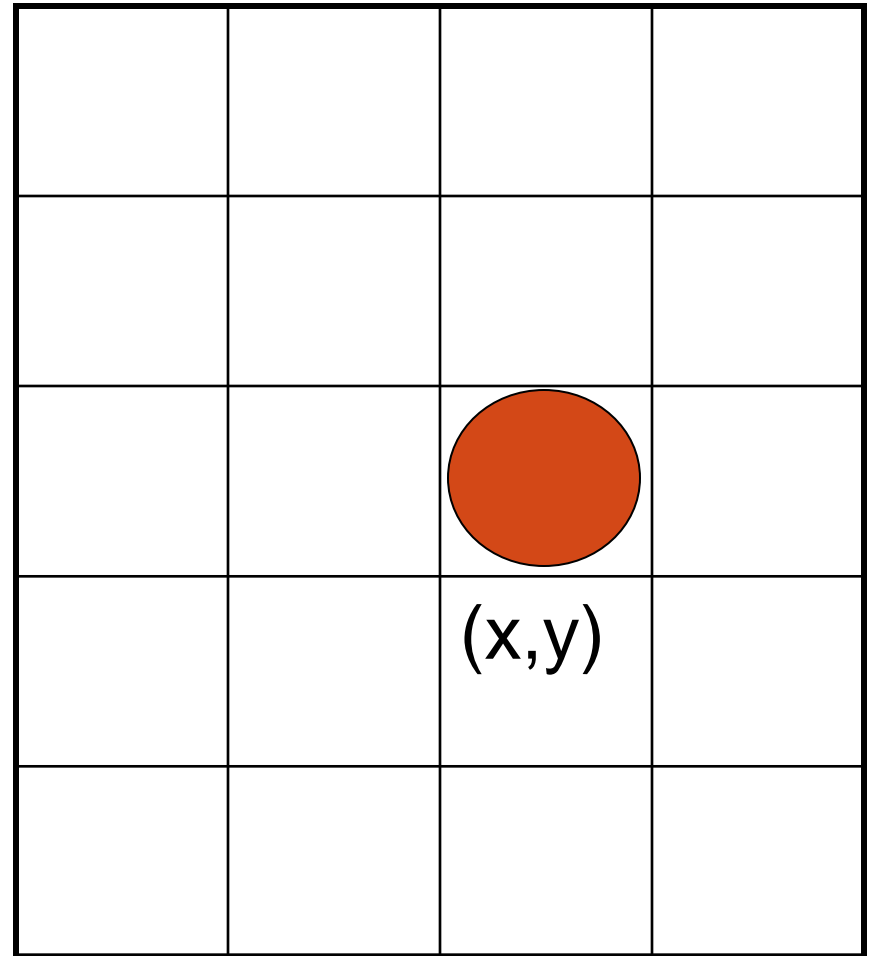
- For a raster display, a picture is completely specified by:
 - intensity and position of pixels, or/and
 - set of complex objects
- Shapes and contours can either be stored in terms of pixel patterns (bitmaps) or as a set of basic geometric structures (for example, line segments).

Introduction

- Output primitives are the basic geometric structures which facilitate or describe a scene/picture. Example of these include:
 - *points, lines, curves (circles, conics etc), surfaces, fill colour, character string etc.*

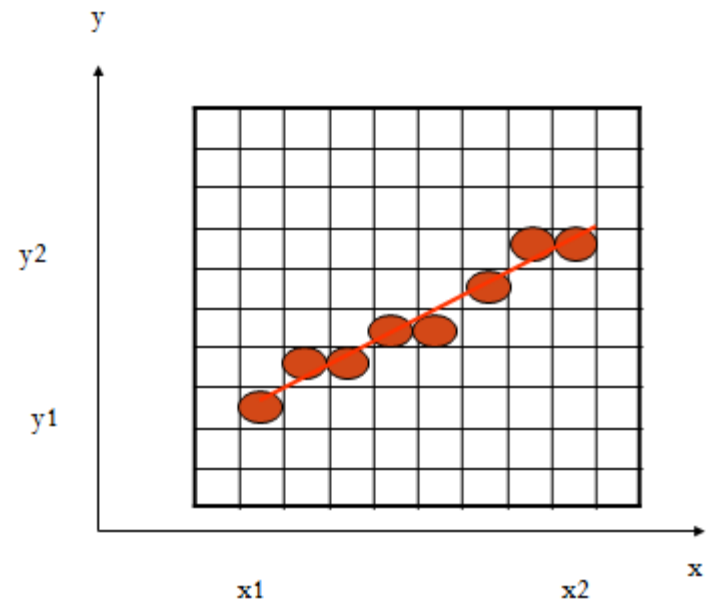
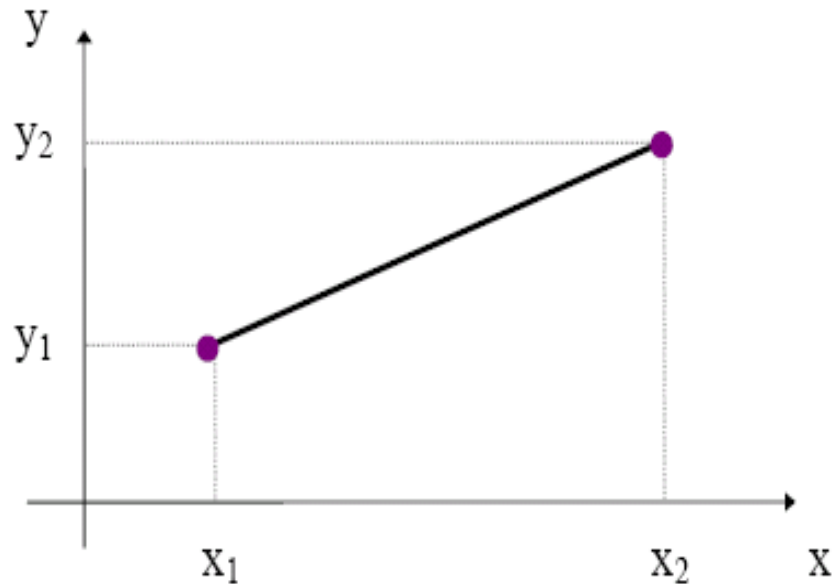
Points

- A point is shown by illuminating a pixel on the screen



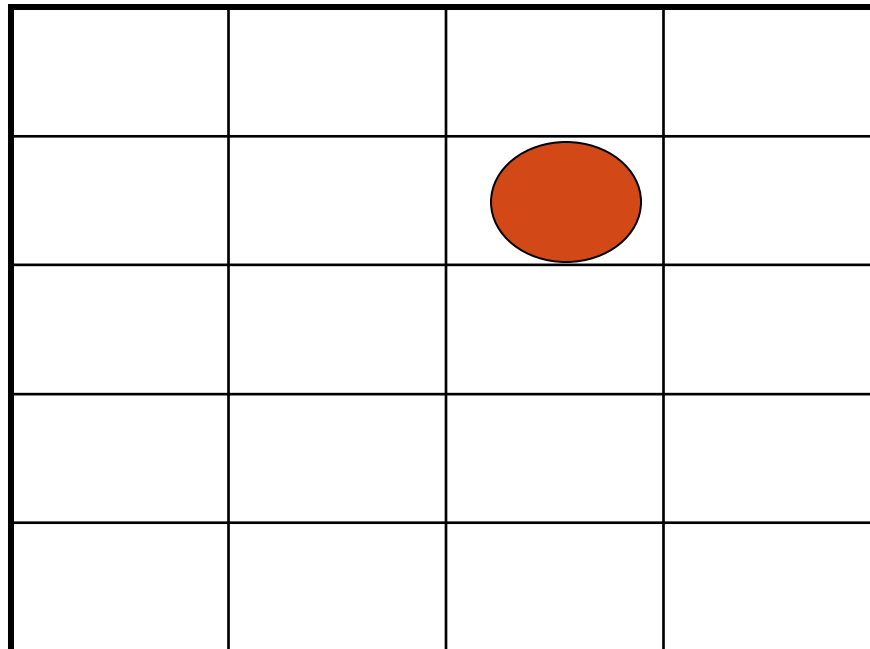
Lines

- A line segment is completely defined in terms of its two endpoints.
- A line segment is thus defined as:
$$\text{Line_Seg} = \{ (x_1, y_1), (x_2, y_2) \}$$
- A line is produced by means of illuminating a set of intermediary pixels between the two endpoints.

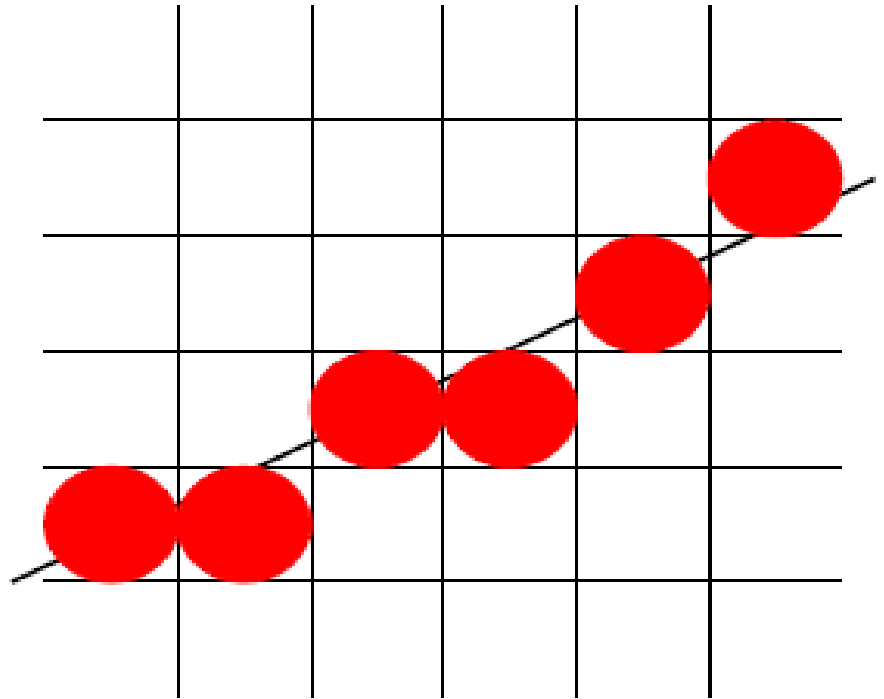


Lines

- Lines is digitized into a set of discrete integer positions that approximate the actual line path.
- Example: A computed line position of $(10.48, 20.51)$ is converted to pixel position $(10, 21)$.

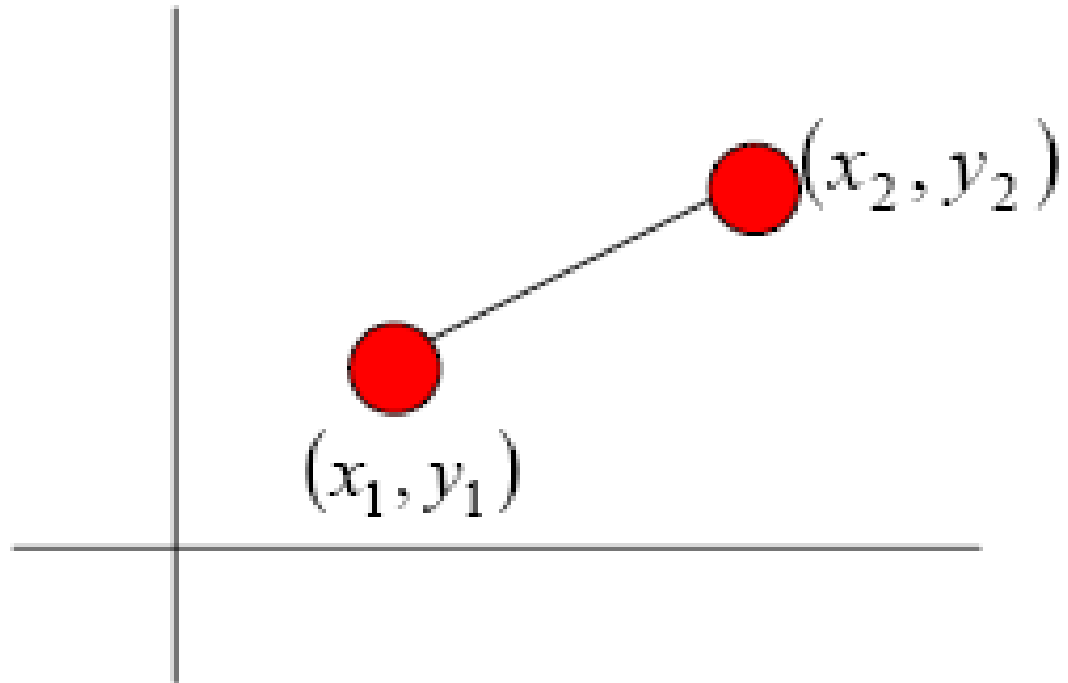


- *The rounding of coordinate values to integer causes all but horizontal and vertical lines to be displayed with a stair step appearance “the jaggies”.*



* Line Drawing Algorithms

- A straight line segment is defined by the coordinate position for the end points of the segment.
- Given Points (x_1, y_1) and (x_2, y_2)



Line

- All line drawing algorithms make use of the fundamental equations:
- Line Eqn. $y = m.x + b$
- Slope $m = y_2 - y_1 / x_2 - x_1 = \Delta y / \Delta x$
- y-intercept $b = y - m.x$
- *x-interval* $\rightarrow \Delta x = \Delta y / m$
- *y-interval* $\rightarrow \Delta y = m \Delta x$

DDA Algorithm (Digital Differential Analyzer)

- A line algorithm Based on calculating either Δy or Δx using the above equations.
- *There are two cases:*
 - *Positive slop*
 - *Negative slop*

DDA- Line with positive Slope

If $m \leq 1$ then take $\Delta_x = 1$

- Compute successive y by

$$y_{k+1} = y_k + m \quad (1)$$

- *Subscript k takes integer values starting from 1, for the first point, and increases by 1 until the final end point is reached.*
- *Since $0.0 < m \leq 1.0$, the calculated y values must be rounded to the nearest integer pixel position.*

DDA with negative slope

- If $m > 1$, reverse the role of x and y and take $\Delta y = 1$, calculate successive x from

$$x_{k+1} = x_k + 1/m \quad (2)$$

- In this case, each computed x value is rounded to the nearest integer pixel position.
- The above equations are based on the assumption that lines are to be processed from left endpoint to right endpoint.

DDA

- In case the line is processed from Right endpoint to Left endpoint, then

$$\Delta_x = -1,$$

$$y_{k+1} = y_k - m \quad \text{for } m \leq 1 \quad (3)$$

or

$$\Delta_y = -1,$$

$$x_{k+1} = x_k - 1/m \quad \text{for } m > 1 \quad (4)$$

DDA

- If $m < 1$,
 - use(1) [provided line is calculated from left to right] and
 - use(3) [provided line is calculated from right to left].
- If $m \geq 1$
 - use (2) or (4).

Merits + Demerits

- Faster than the direct use of line Eqn.
- It eliminates the multiplication in line Eqn.
- For long line segments, the true line Path may be mislead due to round off.
- Rounding operations and floating-point arithmetic are still time consuming.
- The algorithm can still be improved.
- Other algorithms, with better performance also exist.

Code for DDA Algorithm

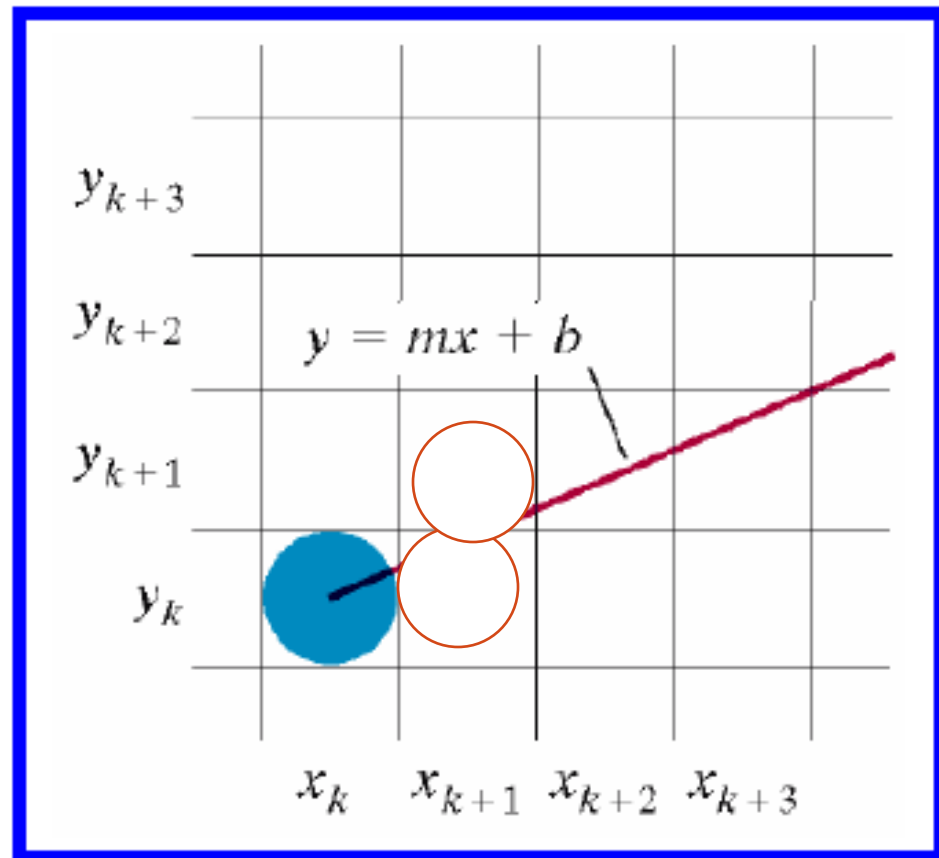
```
Procedure lineDDA(xa,ya,xb,yb:integer);
Var
    dx,dy,steps,k:integer
    xIncrement,yIncrement,x,y:real;
begin
    dx:=xb-xa;
    dy:=yb-ya;
    if abs(dx)>abs(dy) then steps:=abs(dx)
    else steps:=abs(dy);
    xIncrement:=dx/steps;
    yIncrement:=dy/steps;
    x:=xa;
    y:=ya;
    setPixel(round(x),round(y),1);
    for k:=1 to steps do
        begin
            x:=x+xIncrement;
            y:=y+yIncrement;
            setPixel(round(x),round(y),1)
        end
    end; {lineDDA}
```

Bresenham's Line Algorithm

- It is an efficient raster line generation algorithm.
- It can be adapted to display circles and other curves.
- The algorithm
 - After plotting a pixel position (x_k, y_k) , what is the next pixel to plot?
- Consider lines with positive slope.

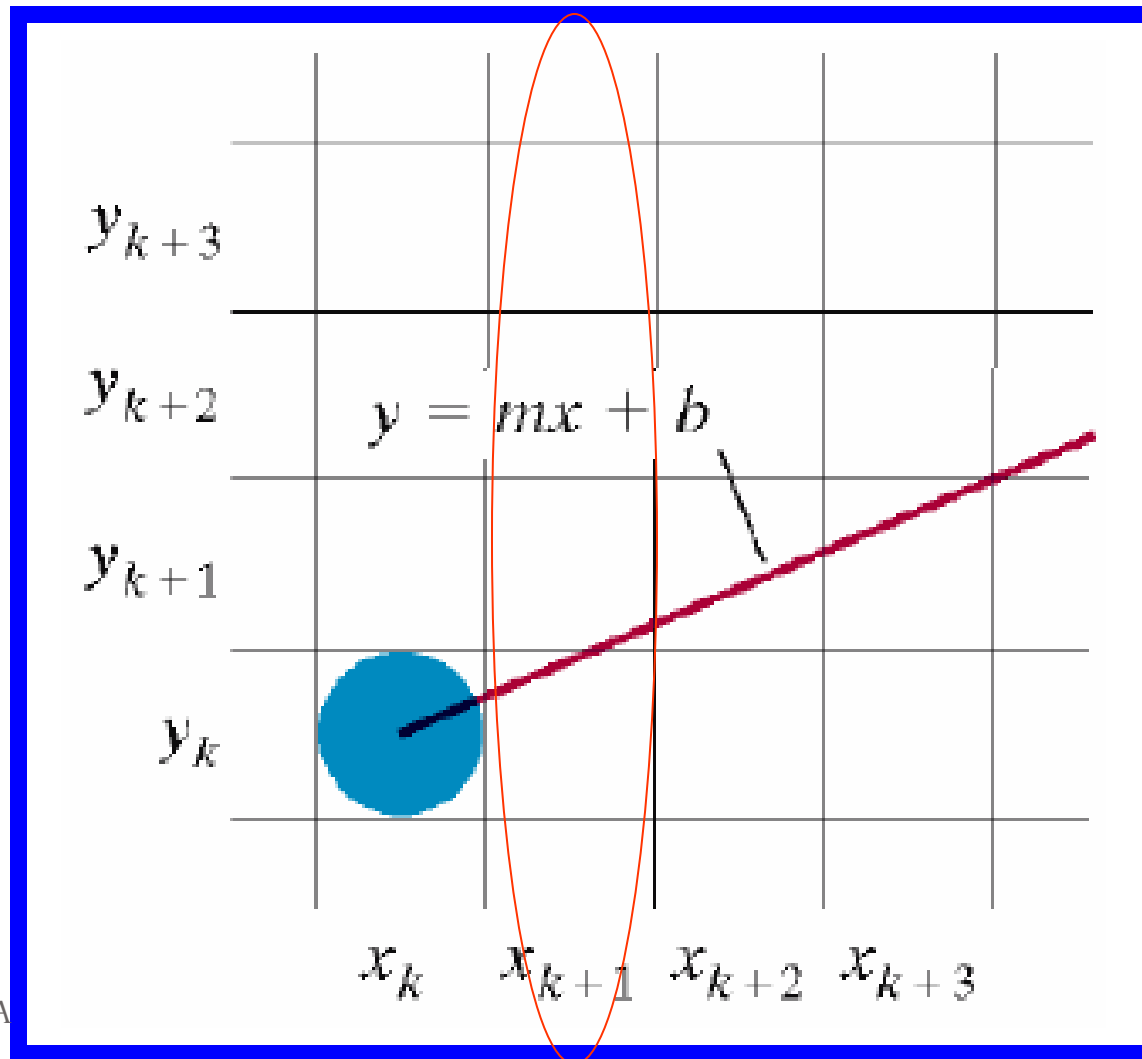
Bresenham's Line

- For a positive slope, $0 < m < 1$ and line is starting from left to right.
- After plotting a pixel position (x_k, y_k) we have two choices for next pixel:
 - $(x_k + 1, y_k)$
 - $(x_k + 1, y_k + 1)$



Bresenham's Line

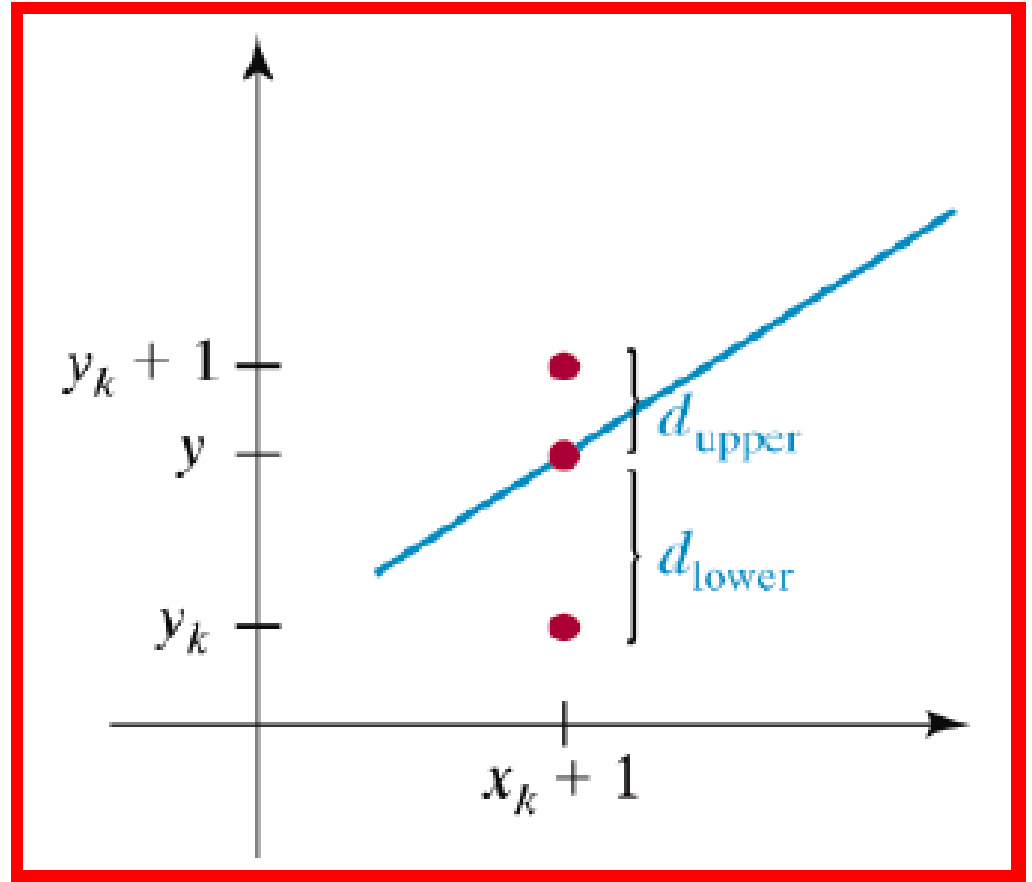
- At position $x_k + 1$, we pay attention to the intersection of the vertical pixel and the mathematical line path.



Bresenham's Line

- At position $x_k + 1$, we label vertical pixel separations from the mathematical line path as

d_{lower} , d_{upper} .



Bresenham's Line

- The y coordinate on the mathematical line at $x_k + 1$ is calculated as

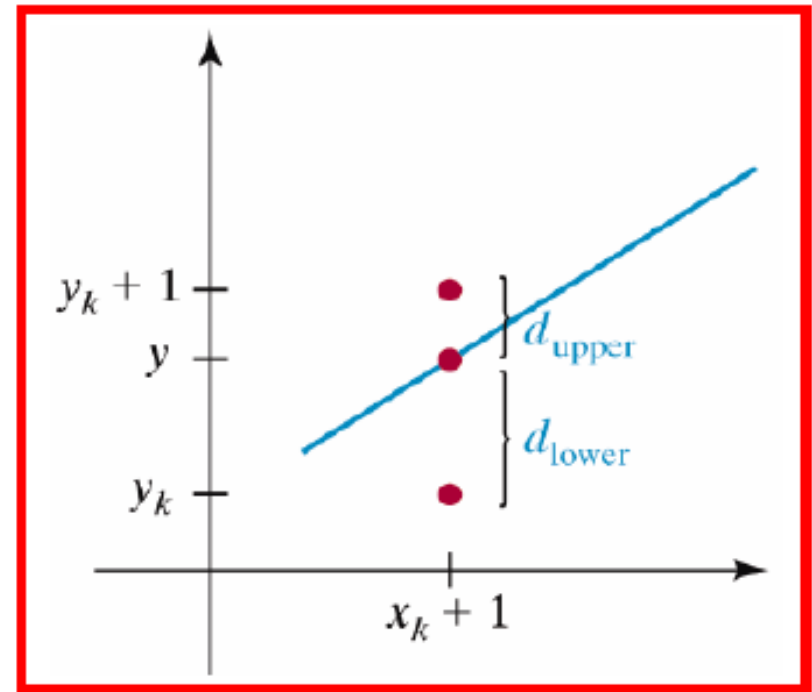
$$y = m(x_k + 1) + b$$

then

$$\begin{aligned} d_{\text{lower}} &= y - y_k \\ &= m(x_k + 1) + b - y_k \end{aligned}$$

and

$$\begin{aligned} d_{\text{upper}} &= (y_k + 1) - y \\ &= y_k + 1 - m(x_k + 1) - b \end{aligned}$$



Bresenham's Line

- To determine which of the two pixels is closest to the line path, we set an efficient test based on the difference between the two pixel separations

$$\begin{aligned}d_{\text{lower}} - d_{\text{upper}} &= 2m (x_k + 1) - 2y_k + 2b - 1 \\ &= 2 (\Delta y / \Delta x) (x_k + 1) - 2y_k + 2b - 1\end{aligned}$$

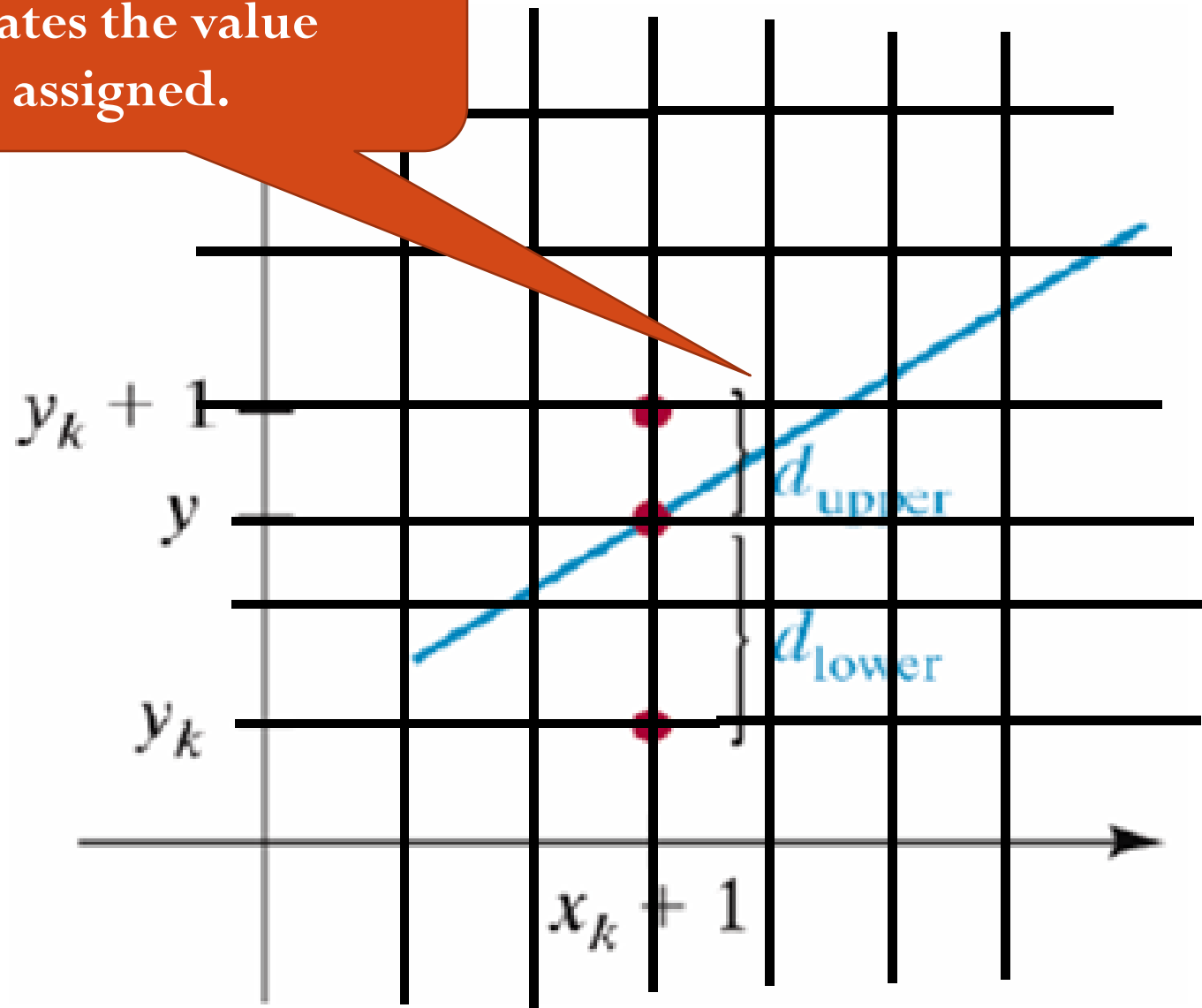
- Consider a decision parameter p_k such that

$$\begin{aligned}p_k &= \Delta x (d_{\text{lower}} - d_{\text{upper}}) \\ &= 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c\end{aligned}$$

- where

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

According to x co-ordinates the value of y is assigned.



Bresenham's Line

- Since $\Delta x > 0$, Comparing (d_{lower} and d_{upper}), would tell which pixel is closer to the line path; is it y_k or $y_k + 1$
- If ($d_{\text{lower}} < d_{\text{upper}}$)
 - Then p_k is negative
 - Hence plot lower pixel.
 - Otherwise
 - Plot the upper pixel.

Bresenham's Line

- We can obtain the values of successive decision parameter as follows:

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

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

- Subtracting these two equations

$$p_{k+1} - p_k = 2\Delta y (x_{k+1} - x_k) - 2\Delta x (y_{k+1} - y_k)$$

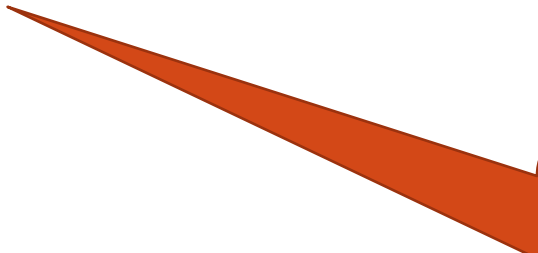
- But $x_{k+1} - x_k = 1$, Therefore

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x (y_{k+1} - y_k)$$

Bresenham's Line

- $(y_{k+1} - y_k)$ is either 0 or 1, depending on the sign of p_k (plotting lower or upper pixel).
- The recursive calculation of p_k is performed at integer x position, starting at the left endpoint.
- p_0 can be evaluated as:

$$p_0 = 2\Delta y - \Delta x$$



(Final result) it is the initial decision parameter.

Bresenham's Line-Drawing Algorithm for $m < 1$

1. Input the two line end points and store the left end point in (x_0, y_0) .
2. Load (x_0, y_0) into the frame buffer; that is, plot the first point.
3. Calculate the constants Δx , Δy , $2\Delta y$, and $2\Delta y - 2\Delta x$, and obtain the starting value for the decision parameter as

$$p_0 = 2\Delta y - \Delta x$$

4. At each x_k along the line, starting at $k = 0$, perform the following test: If $p_k < 0$, next point to plot is (x_{k+1}, y_k)

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

Otherwise, the 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

- To illustrate the algorithm, we digitize the line with endpoints (20,10) and (30,18). This line has slope of 0.8, with

$$\Delta_x = 10$$

$$\Delta_y = 8$$

- The initial decision parameter has the value

$$p_0 = 2\Delta_y - \Delta_x = 6$$

- and the increments for calculating successive decision parameters are

$$2 \Delta_y = 16$$

$$2 \Delta_y - 2 \Delta_x = -4$$

Example

- We plot the initial point $(x_0, y_0) = (20, 10)$ and determine successive pixel positions along the line path from the decision parameter as

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

Example

A plot of the pixels generated along this line path is shown in Fig.

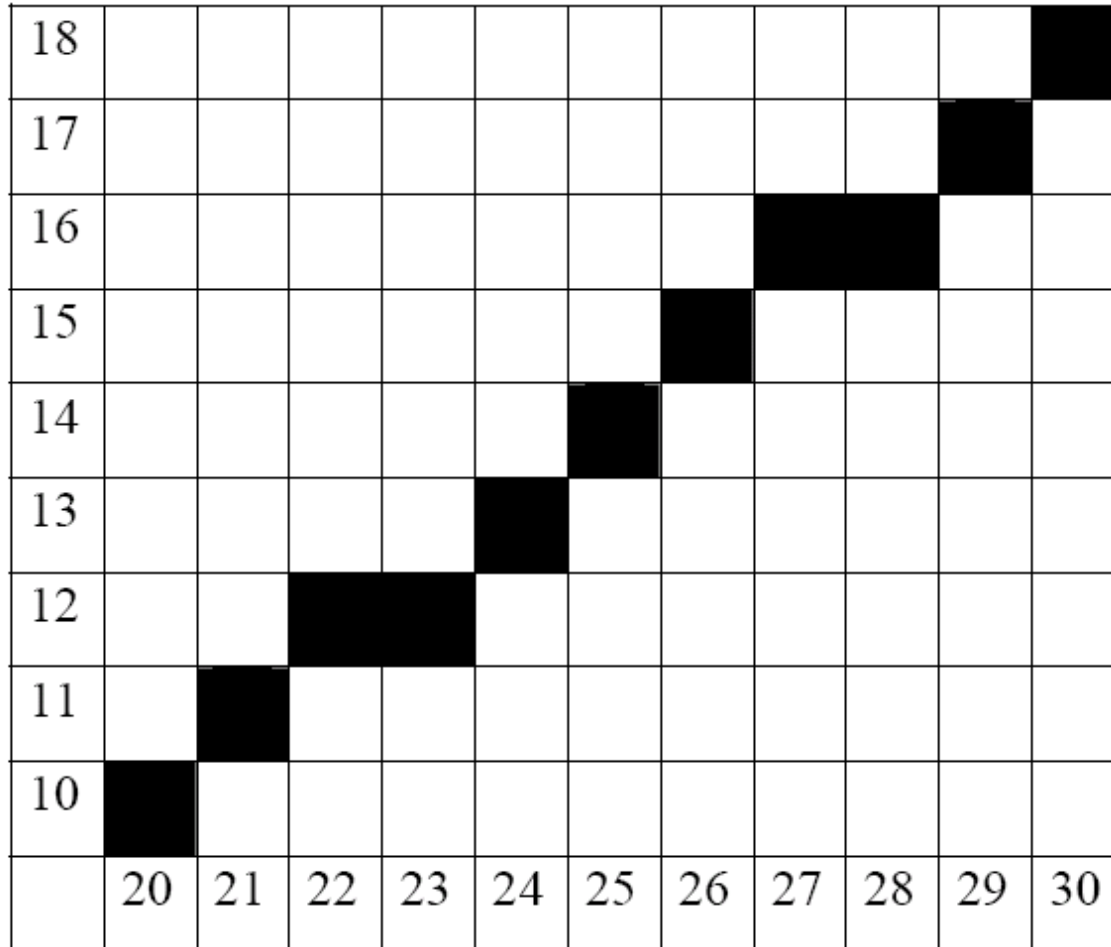
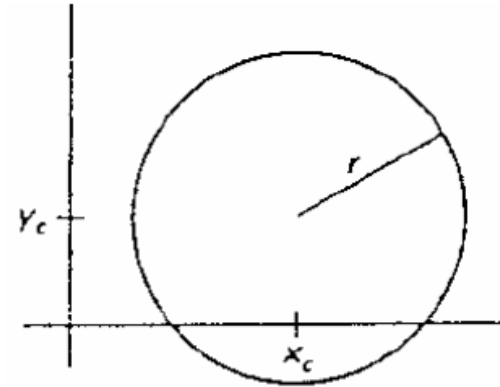


Figure: The Bresenham line from point (20,10) to point (30,18)

Circle Generating Algorithms

- A circle is defined as the set of points that are all at a given distance r from a center point (x_c, y_c) .

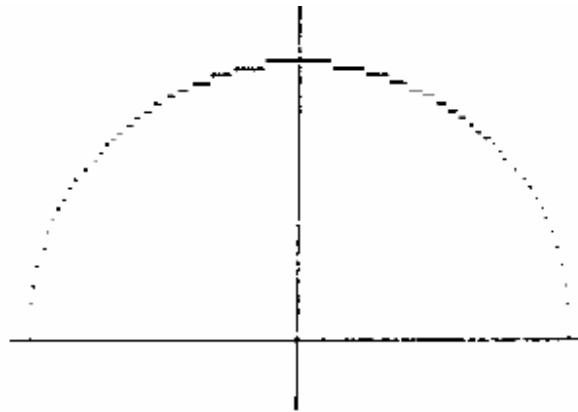


- For any circle point (x, y) , this distance is expressed by the Equation
$$(x - x_c)^2 + (y - y_c)^2 = r^2$$
- We calculate the points by stepping along the x-axis in unit steps from $x_c - r$ to $x_c + r$ and calculate y values as

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

Circle Generating Algorithms

- **There are some problems with this approach:**
 1. Considerable computation at each step.
 2. Non-uniform spacing between plotted pixels as in this Figure.



Circle Generating Algorithms

- Problem 2 can be removed using the polar form:

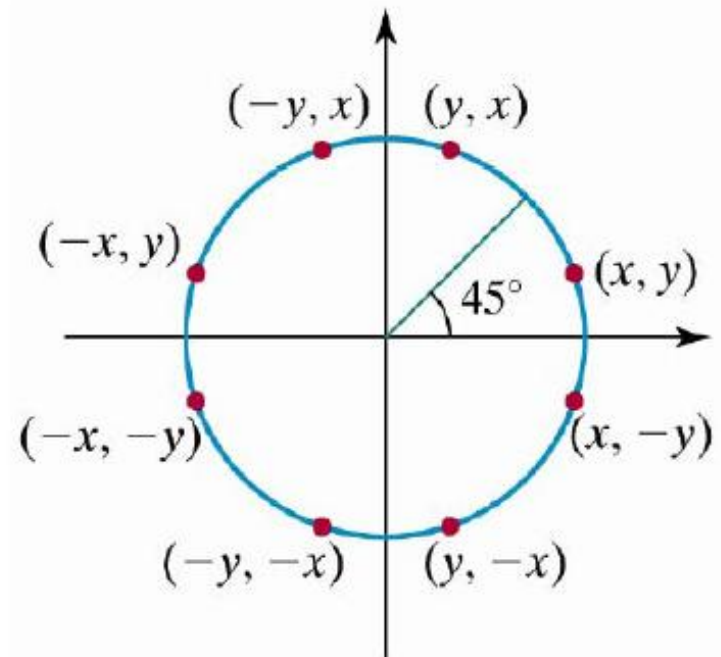
$$x = x_c + r \cos \theta$$

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

- using a fixed angular step size, a circle is plotted with equally spaced points along the circumference.

Circle Generating Algorithms

- Problem 1 can be overcome by considering the symmetry of circles as in Figure 3.
- But it still requires a good deal of computation time.



- **Efficient Solutions**
 - Midpoint Circle Algorithm

Mid point Circle Algorithm

- To apply the midpoint method, we define a circle function:
- Any point (x,y) on the boundary of the circle with radius r satisfies the equation $f_{circle}(x,y) = 0$.

$$f_{circle}(x,y) = x^2 + y^2 - r^2 = 0 \quad (2)$$

- *If the points is in the interior of the circle, the circle function is negative.*
- *If the point is outside the circle, the circle function is positive.*
- *To summarize, the relative position of any point (x,y) can be determined by checking the sign of the circle function:*

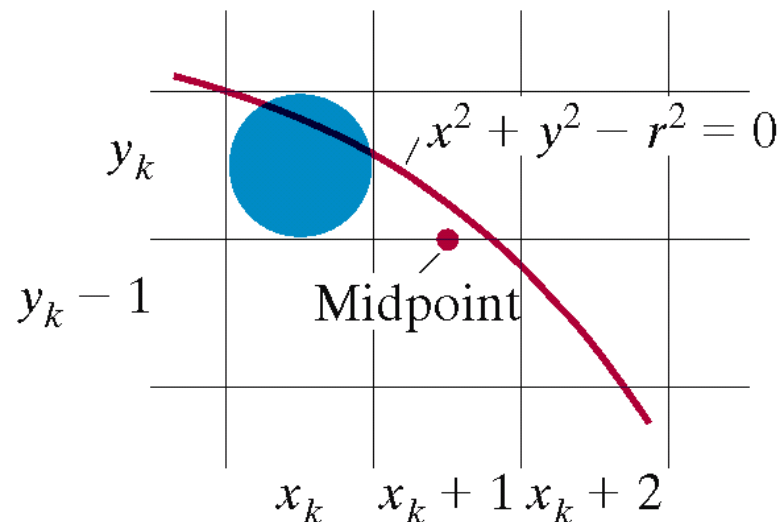
Mid point Circle Algorithm

$$f_{circle}(x, y) = \begin{cases} < 0 & \text{if } (x, y) \text{ is inside the circle boundary} \\ 0 & \text{if } (x, y) \text{ is on the circle boundary} \\ > 0 & \text{if } (x, y) \text{ is outside the circle boundary} \end{cases} \quad (3)$$

- The circle function tests in (3) are performed for the mid positions 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.

Mid point Circle Algorithm

- Figure shows the midpoint between the two candidate pixels at sampling position $x_k + 1$. Assuming we have just plotted the pixel at (x_k, 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.



Mid point Circle Algorithm

- Our decision parameter is the circle function (2) evaluated at the midpoint between these two pixels:

$$\begin{aligned} P_k &= f_{circle}\left(x_k + 1, y_k - \frac{1}{2}\right) \\ &= (x_k + 1)^2 + \left(y_k - \frac{1}{2}\right)^2 - r^2 \end{aligned} \tag{4}$$

Mid point Circle Algorithm

- 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 midpoint 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.

Mid point Circle Algorithm

- We obtain a recursive expression for the next decision parameter by evaluating the circle function at sampling position $x_{k+1} + 1 = x_k + 2$

$$\begin{aligned} p_{k+1} &= f_{circle}\left(x_{k+1} + 1, y_{k+1} - \frac{1}{2}\right) \\ &= [(x_k + 1) + 1]^2 + \left(y_{k+1} - \frac{1}{2}\right)^2 - r^2 \end{aligned}$$

or

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

- where y_{k+1} is either y_k or y_{k-1} , depending on the sign of p_k .

Mid point Circle Algorithm

- Increments for obtaining p_{k+1} are either
 - $2x_{k+1} + 1$ (if p_k is negative) or
 - $2x_{k+1} + 1 - 2y_{k+1}$ (if p_k is positive)
- Evaluation of the terms $2x_{k+1}$ and $2y_{k+1}$ can also be done incrementally as:

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

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

Mid point Circle Algorithm

- At the start position $(0, r)$, these two terms $(2x, 2y)$ have the values 0 and $2r$, respectively.
- Each successive value is obtained by adding 2 to the previous value of $2x$ and subtracting 2 from the previous value of $2y$.

Mid point Circle Algorithm

- The initial decision parameter is obtained by evaluating the circle function at the start position $(x_0, y_0) = (0, r)$:

$$\begin{aligned} p_0 &= f_{circle}\left(1, r - \frac{1}{2}\right) \\ &= 1 + \left(r - \frac{1}{2}\right)^2 - r^2 \end{aligned}$$

or

$$p_0 = \frac{5}{4} - r$$

Mid point Circle Algorithm

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

$$p_0 = 1 - r \quad (\text{for } r \text{ an integer})$$

- since all increments are integers.

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

$$p_0 = \frac{5}{4} - r$$

3. At each x_k position, starting at $k = 0$, perform the following test:
If $p_k < 0$, the next point along the circle centered on $(0,0)$ is (x_{k+1}, y_k) and

$$p_{k+1} = p_k + 2x_{k+1} + 1$$

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) onto the circular path centered on (x_0, y_0) and plot the coordinate values:

$$x = x + x_c, y = y + y_c$$

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

Example

- Given a circle radius $r = 10$, we demonstrate the midpoint circle algorithm by determining positions along the circle octant in the first quadrant from $x = 0$ to $x = y$. The initial value of the decision parameter is

$$p_0 = 1 - r = -9$$

Example

- For the circle centered on the coordinate origin, the initial point is $(x_0, y_0) = (0, 10)$, and initial increment terms for calculating the decision parameters are

$$2x_0 = 0, 2y_0 = 20$$

- Successive decision parameter values and positions along the circle path are calculated using the midpoint method as shown in the table.

Example

k	p_k	(x_{k+1}, y_{k+1})	$2x_{k+1}$	$2y_{k+1}$
0	-9	(1,10)	2	20
1	-6	(2,10)	4	20
2	-1	(3,10)	6	20
3	6	(4,9)	8	18
4	-3	(5,9)	10	18
5	8	(6,8)	12	16
6	5	(7,7)	14	14

Example

- A plot of the generated pixel positions in the first quadrant is shown in Figure 5.

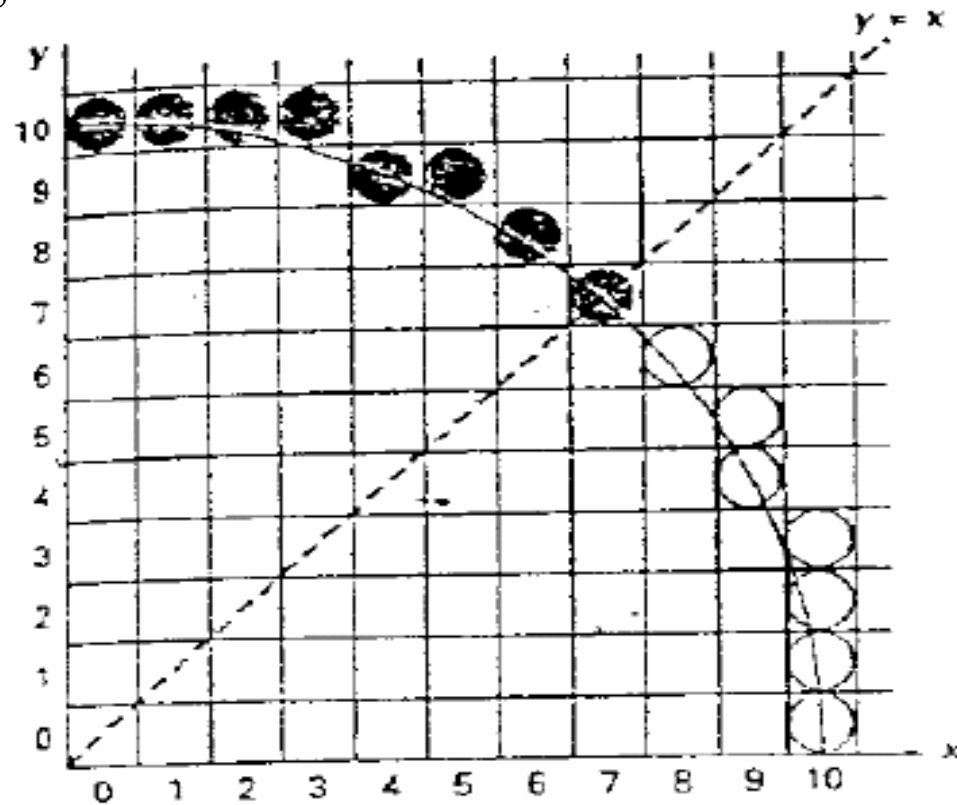


Figure 5

Midpoint Ellipse Algorithm

- Ellipse equations are greatly simplified if the major and minor axes are oriented to align with the coordinate axes.
- In Fig. , we show an ellipse in “standard position” with major and minor axes oriented parallel to the x and y axes.
- Parameter r_x for this example labels the semi-major axis, and parameter r_y labels the semi-minor axis.

Midpoint Ellipse Algorithm

- The equation for the ellipse shown in Fig. 3-22 can be written in terms of the ellipse center coordinates and parameters r_x and r_y as

$$\left(\frac{x - x_c}{r_x}\right)^2 + \left(\frac{y - y_c}{r_y}\right)^2 = 1 \quad (3-37)$$

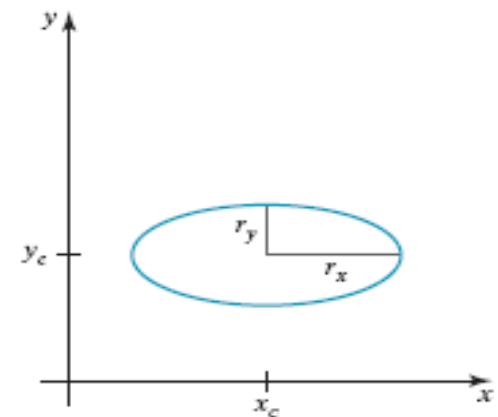


FIGURE 3-22 Ellipse centered at (x_c, y_c) with semimajor axis r_x and semiminor axis r_y .

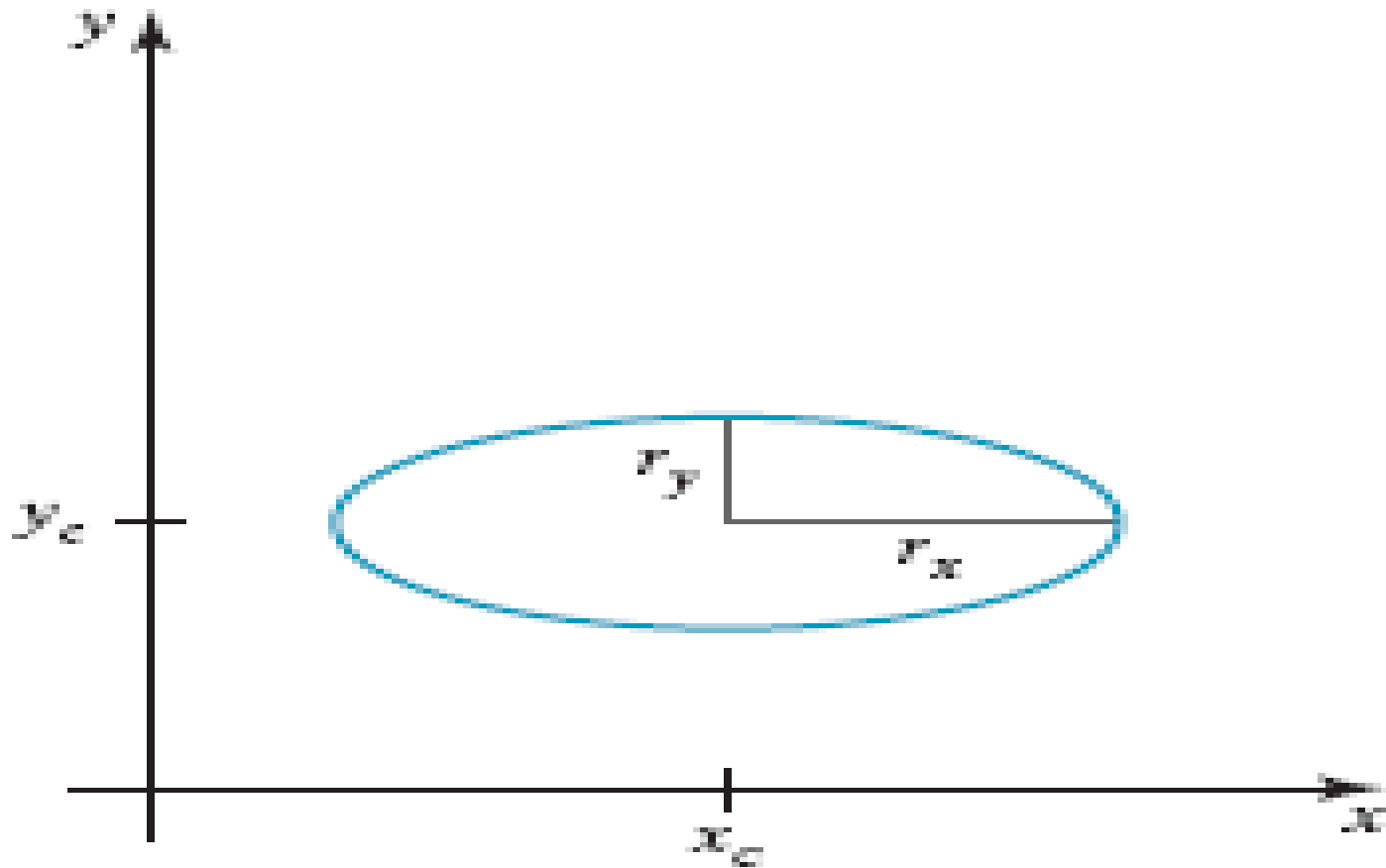


FIGURE 3-22 Ellipse centered at (x_c, y_c) with semimajor axis r_x and semiminor axis r_y .

Midpoint Ellipse Algorithm

- Using polar coordinates r and θ , we can also describe the ellipse in standard position with the parametric equations

$$\begin{aligned}x &= x_c + r_x \cos \theta \\y &= y_c + r_y \sin \theta\end{aligned}\tag{3-38}$$

Midpoint Ellipse Algorithm

- The midpoint ellipse method is applied throughout the first quadrant in two parts.
- Figure 3-25 shows the division of the first quadrant according to the slope of an ellipse with $r_x < r_y$.

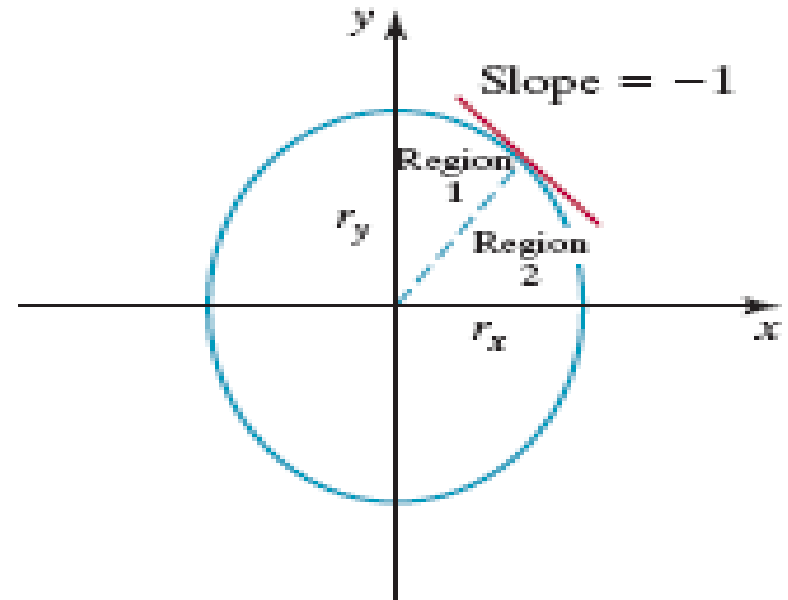


FIGURE 3-25 Ellipse processing regions. Over region 1, the magnitude of the ellipse slope is less than 1.0; over region 2, the magnitude of the slope is greater than 1.0.

Midpoint Ellipse Algorithm

- Regions 1 and 2 (Fig. 3-25) can be processed in various ways.
- We can start at position $(0, r_y)$ and step clockwise along the elliptical path in the first quadrant, shifting from unit steps in x to unit steps in y when the slope becomes less than -1.0 .
- Alternatively, we could start at $(r_x, 0)$ and select points in a counterclockwise order, shifting from unit steps in y to unit steps in x when the slope becomes greater than -1.0 .

Midpoint Ellipse Algorithm

- We define an ellipse function from Eq. 3-37 with $(x_c, y_c) = (0, 0)$ as

$$f_{\text{ellipse}}(x, y) = r_y^2 x^2 + r_x^2 y^2 - r_x^2 r_y^2 \quad (3-39)$$

- which has the following properties:

$$f_{\text{ellipse}}(x, y) \begin{cases} < 0, & \text{if } (x, y) \text{ is inside the ellipse boundary} \\ = 0, & \text{if } (x, y) \text{ is on the ellipse boundary} \\ > 0, & \text{if } (x, y) \text{ is outside the ellipse boundary} \end{cases} \quad (3-40)$$

Midpoint Ellipse Algorithm

- Starting at $(0, r_y)$, we take unit steps in the x direction until we reach the boundary between region 1 and region 2 (Fig. 3-25).
- Then we switch to unit steps in the y direction over the remainder of the curve in the first quadrant.
- At each step we need to test the value of the slope of the curve.

Midpoint Ellipse Algorithm

- The ellipse slope is calculated from Eq. 3-39 as

$$\frac{dy}{dx} = -\frac{2r_y^2x}{2r_x^2y} \quad (3-41)$$

- At the boundary between region 1 and region 2, $dy/dx = -1.0$ and

$$2r_y^2x = 2r_x^2y$$

- Therefore, we move out of region 1 whenever

$$2r_y^2x \geq 2r_x^2y \quad (3-42)$$

Midpoint Ellipse Algorithm

- Figure 3-26 shows the midpoint between the two candidate pixels at sampling position $x_k + 1$ in the first region.
- Assuming position (x_k, y_k) has been selected in the previous step, we determine the next position along the ellipse path by evaluating the decision parameter (that is, the ellipse function 3-39) at this midpoint:

$$\begin{aligned} p1_k &= f_{\text{ellipse}}\left(x_k + 1, y_k - \frac{1}{2}\right) \\ &= r_y^2(x_k + 1)^2 + r_x^2\left(y_k - \frac{1}{2}\right)^2 - r_x^2 r_y^2 \end{aligned} \quad (3-43)$$

Midpoint Ellipse Algorithm

- If $p1_k < 0$, the midpoint is inside the ellipse and the pixel on scan line y_k is closer to the ellipse boundary.
- Otherwise, the midposition is outside or on the ellipse boundary, and we select the pixel on scan line $y_k - 1$.

Midpoint Ellipse Algorithm

- At the next sampling position ($x_{k+1} + 1 = x_k + 2$), the decision parameter for region 1 is evaluated as

$$\begin{aligned} p1_{k+1} &= f_{\text{ellipse}}\left(x_{k+1} + 1, y_{k+1} - \frac{1}{2}\right) \\ &= r_y^2[(x_k + 1) + 1]^2 + r_x^2\left(y_{k+1} - \frac{1}{2}\right)^2 - r_x^2 r_y^2 \end{aligned}$$

or

$$p1_{k+1} = p1_k + 2r_y^2(x_k + 1) + r_y^2 + r_x^2 \left[\left(y_{k+1} - \frac{1}{2}\right)^2 - \left(y_k - \frac{1}{2}\right)^2 \right] \quad (3-44)$$

where y_{k+1} is either y_k or $y_k - 1$, depending on the sign of $p1_k$.

Midpoint Ellipse Algorithm

- Decision parameters are incremented by the following amounts:

$$\text{increment} = \begin{cases} 2r_y^2 x_{k+1} + r_y^2, & \text{if } p1_k < 0 \\ 2r_y^2 x_{k+1} + r_y^2 - 2r_x^2 y_{k+1}, & \text{if } p1_k \geq 0 \end{cases}$$

Midpoint Ellipse Algorithm

- At the initial position $(0, r_y)$, these two terms evaluate to

$$2r_y^2x = 0 \quad (3-45)$$

$$2r_x^2y = 2r_x^2r_y \quad (3-46)$$

- As x and y are incremented, updated values are obtained by adding $2r_y^2$ to the current value of the increment term in Eq. 3-45 and subtracting $2r_x^2$ from the current value of the increment term in Eq. 3-46.
- The updated increment values are compared at each step, and we move from region 1 to region 2 when condition 3-42 is satisfied.

Midpoint Ellipse Algorithm

- In region 1, the initial value of the decision parameter is obtained by evaluating the ellipse function at the start position $(x_0, y_0) = (0, r_y)$:

$$\begin{aligned} p1_0 &= f_{\text{ellipse}}\left(1, r_y - \frac{1}{2}\right) \\ &= r_y^2 + r_x^2\left(r_y - \frac{1}{2}\right)^2 - r_x^2 r_y^2 \end{aligned}$$

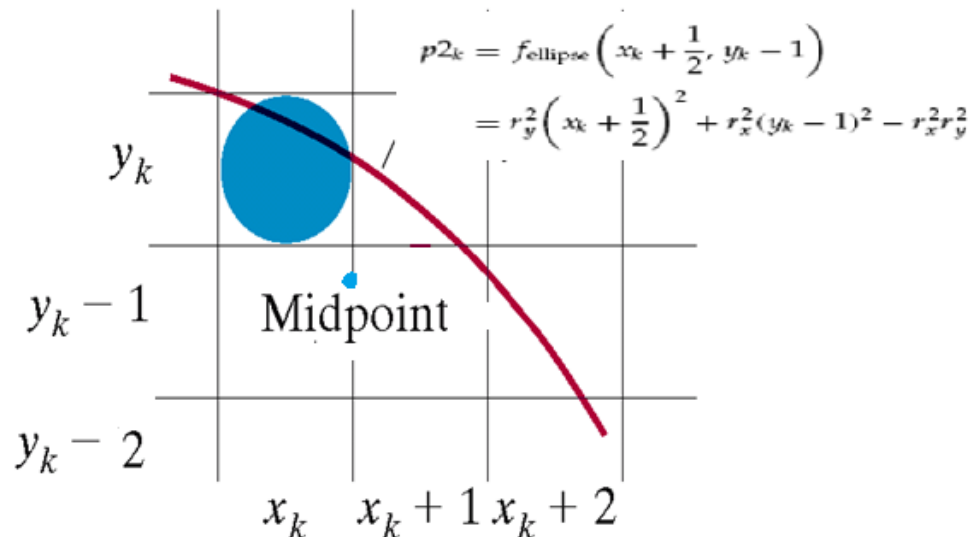
or

$$p1_0 = r_y^2 - r_x^2 r_y + \frac{1}{4} r_x^2 \quad (3-47)$$

Midpoint Ellipse Algorithm

- Over region 2, we sample at unit intervals in the negative y direction, and the midpoint is now taken between horizontal pixels at each step (Fig. 3-27).

$$\begin{aligned}
 p2_k &= f_{\text{ellipse}}\left(x_k + \frac{1}{2}, y_k - 1\right) \\
 &= r_y^2 \left(x_k + \frac{1}{2}\right)^2 + r_x^2 (y_k - 1)^2 - r_x^2 r_y^2
 \end{aligned} \tag{3-48}$$



Midpoint Ellipse Algorithm

- If $p2_k > 0$, the midpoint is outside the ellipse boundary, and we select the pixel at x_k .
- If $p2_k \leq 0$, the midpoint is inside or on the ellipse boundary, and we select
- pixel position x_{k+1} .

Midpoint Ellipse Algorithm

- To determine the relationship between successive decision parameters in region 2, we evaluate the ellipse function at the next sampling step $y_{k+1} - 1 = y_k - 2$:

$$\begin{aligned} p2_{k+1} &= f_{\text{ellipse}}\left(x_{k+1} + \frac{1}{2}, y_{k+1} - 1\right) \\ &= r_y^2 \left(x_{k+1} + \frac{1}{2}\right)^2 + r_x^2 [(y_k - 1) - 1]^2 - r_x^2 r_y^2 \end{aligned} \quad (3-49)$$

or

$$p2_{k+1} = p2_k - 2r_x^2(y_k - 1) + r_x^2 + r_y^2 \left[\left(x_{k+1} + \frac{1}{2}\right)^2 - \left(x_k + \frac{1}{2}\right)^2 \right] \quad (3-50)$$

with x_{k+1} set either to x_k or to $x_k + 1$, depending on the sign of $p2_k$.

Midpoint Ellipse Algorithm

- When we enter region 2, the initial position (x_0, y_0) is taken as the last position selected in region 1 and the initial decision parameter in region 2 is then

$$\begin{aligned} p_{20} &= f_{\text{ellipse}}\left(x_0 + \frac{1}{2}, y_0 - 1\right) \\ &= r_y^2 \left(x_0 + \frac{1}{2}\right)^2 + r_x^2 (y_0 - 1)^2 - r_x^2 r_y^2 \end{aligned} \quad (3-51)$$

Midpoint Ellipse Algorithm

1. Input r_x , r_y , and ellipse center (x_c, y_c) , and obtain the first point on an ellipse centered on the origin as

$$(x_0, y_0) = (0, r_y)$$

2. Calculate the initial value of the decision parameter in region 1 as

$$p1_0 = r_y^2 - r_x^2 r_y + \frac{1}{4} r_x^2$$

3. At each x_k position in region 1, starting at $k = 0$, perform the following test. If $p1_k < 0$, the next point along the ellipse centered on $(0, 0)$ is (x_{k+1}, y_k) and

$$p1_{k+1} = p1_k + 2r_y^2 x_{k+1} + r_y^2$$

Otherwise, the next point along the ellipse is $(x_k + 1, y_k - 1)$ and

$$p1_{k+1} = p1_k + 2r_y^2 x_{k+1} - 2r_x^2 y_{k+1} + r_y^2$$

with

$$2r_y^2 x_{k+1} = 2r_y^2 x_k + 2r_y^2, \quad 2r_x^2 y_{k+1} = 2r_x^2 y_k - 2r_x^2$$

and continue until $2r_y^2 x \geq 2r_x^2 y$.

4. Calculate the initial value of the decision parameter in region 2 as

$$p2_0 = r_y^2 \left(x_0 + \frac{1}{2} \right)^2 + r_x^2 (y_0 - 1)^2 - r_x^2 r_y^2$$

where (x_0, y_0) is the last position calculated in region 1.

5. At each y_k position in region 2, starting at $k = 0$, perform the following test. If $p2_k > 0$, the next point along the ellipse centered on $(0, 0)$ is (x_k, y_{k+1}) and

$$p2_{k+1} = p2_k - 2r_x^2 y_{k+1} + r_x^2$$

Otherwise, the next point along the ellipse is $(x_k + 1, y_k - 1)$ and

$$p2_{k+1} = p2_k + 2r_y^2 x_{k+1} - 2r_x^2 y_{k+1} + r_x^2$$

using the same incremental calculations for x and y as in region 1. Continue until $y = 0$.

6. For both regions, determine symmetry points in the other three quadrants.

Move each calculated pixel position (x, y) onto the elliptical path centered on (x_c, y_c) and plot the coordinate values:

$$x = x + x_c, \quad y = y + y_c$$

Example

- Given input ellipse parameters $r_x = 8$ and $r_y = 6$, we illustrate the steps in the midpoint ellipse algorithm by determining raster positions along the ellipse path in the first quadrant.
- Initial values and increments for the decision parameter calculations are

$$\begin{aligned} 2r_y^2x &= 0 && \text{(with increment } 2r_y^2 = 72) \\ 2r_x^2y &= 2r_x^2r_y && \text{(with increment } -2r_x^2 = -128) \end{aligned}$$

Example

- For region 1, the initial point for the ellipse centered on the origin is $(x_0, y_0) = (0, 6)$, and the initial decision parameter value is

$$p1_0 = r_y^2 - r_x^2 r_y + \frac{1}{4} r_x^2 = -332$$

- Successive midpoint decision parameter values and the pixel positions along the ellipse are listed in the following table.

Example

k	$p1_k$	(x_{k+1}, y_{k+1})	$2r_y^2 x_{k+1}$	$2r_x^2 y_{k+1}$
0	-332	(1, 6)	72	768
1	-224	(2, 6)	144	768
2	-44	(3, 6)	216	768
3	208	(4, 5)	288	640
4	-108	(5, 5)	360	640
5	288	(6, 4)	432	512
6	244	(7, 3)	504	384

Example

- We now move out of region 1, since

$$2r^2_y x > 2r^2_x y.$$

- For region 2, the initial point is

$$(x_0, y_0) = (7, 3)$$

- and the initial decision parameter is

$$p_{20} = f_{\text{ellipse}}\left(7 + \frac{1}{2}, 2\right) = -151$$

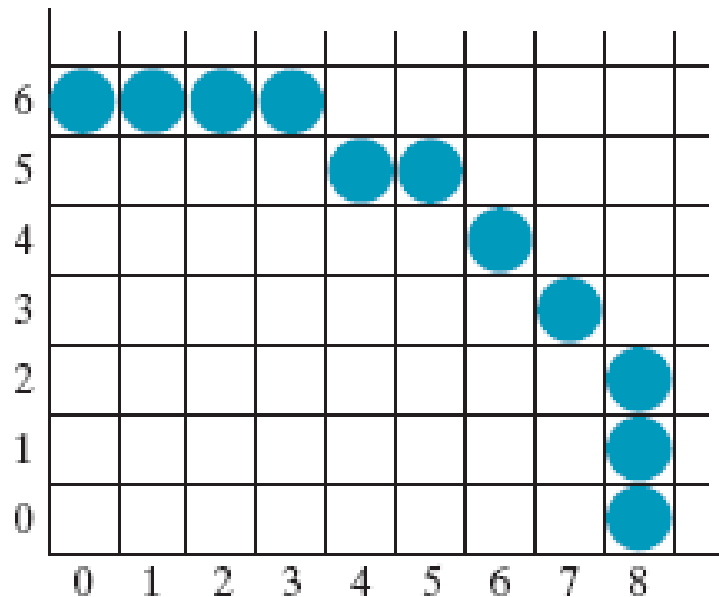
Example

- The remaining positions along the ellipse path in the first quadrant are then calculated as

k	$p1_k$	(x_{k+1}, y_{k+1})	$2r_y^2 x_{k+1}$	$2r_x^2 y_{k+1}$
0	-151	(8, 2)	576	256
1	233	(8, 1)	576	128
2	745	(8, 0)	—	—

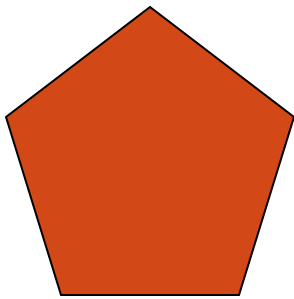
Example

- A plot of the calculated positions for the ellipse within the first quadrant is shown bellow:

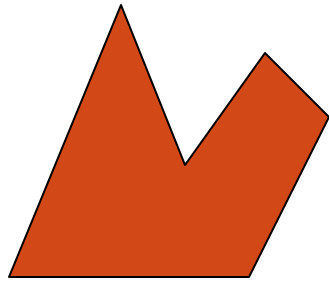


Polygon Fill Algorithm

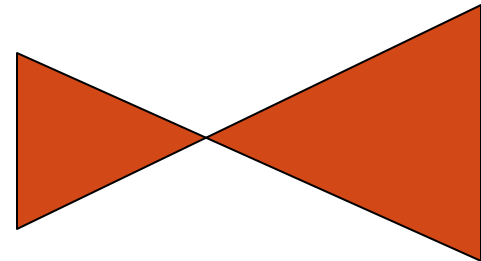
- *Different types of Polygons*
 - Simple Convex
 - Simple Concave
 - Non-simple : self-intersecting



Convex



Concave



Self-intersecting

Polygon Fill Algorithm

- A scan-line fill algorithm of a region is performed as follows:
 1. Determining the intersection positions of the boundaries of the fill region with the screen scan lines.
 2. Then the fill colors are applied to each section of a scan line that lies within the interior of the fill region.
- The simplest area to fill is a polygon, because each scan-line intersection point with a polygon boundary is obtained by solving a pair of simultaneous linear equations, where the equation for the scan line is simply $y = \text{constant}$.

Example

- Consider the following polygon:

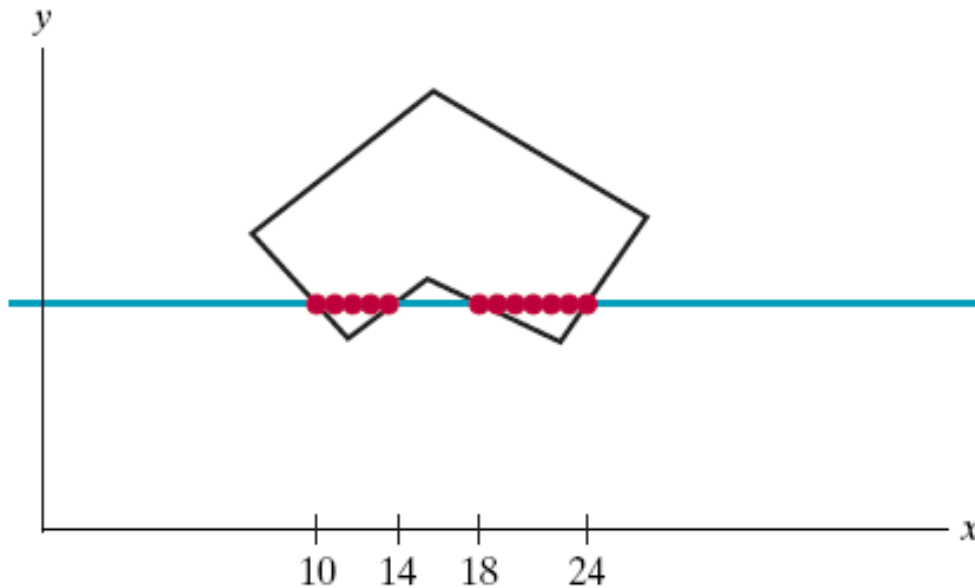


FIGURE 4-20 Interior pixels along a scan line passing through a polygon fill area.

Example

- For each scan line that crosses the polygon, the edge intersections are sorted from left to right, and then the pixel positions between, and including, each intersection pair are set to the specified fill color.
- In the previous Figure, the four pixel intersection positions with the polygon boundaries define two stretches of interior pixels.

Example

- The fill color is applied to the five pixels:
 - from $x = 10$ to $x = 14$

and

- To the seven pixels
 - from $x = 18$ to $x = 24$.

Polygon Fill Algorithm

- However, the scan-line fill algorithm for a polygon is not quite as simple
- Whenever a scan line passes through a vertex, it intersects two polygon edges at that point.
- In some cases, this can result in an odd number of boundary intersections for a scan line.

Polygon Fill Algorithm

- Consider the next Figure.
- It shows two scan lines that cross a polygon fill area and intersect a vertex.
- Scan line y' intersects an even number of edges, and the two pairs of intersection points along this scan line correctly identify the interior pixel spans.
- But scan line y intersects five polygon edges.

Polygon Fill Algorithm

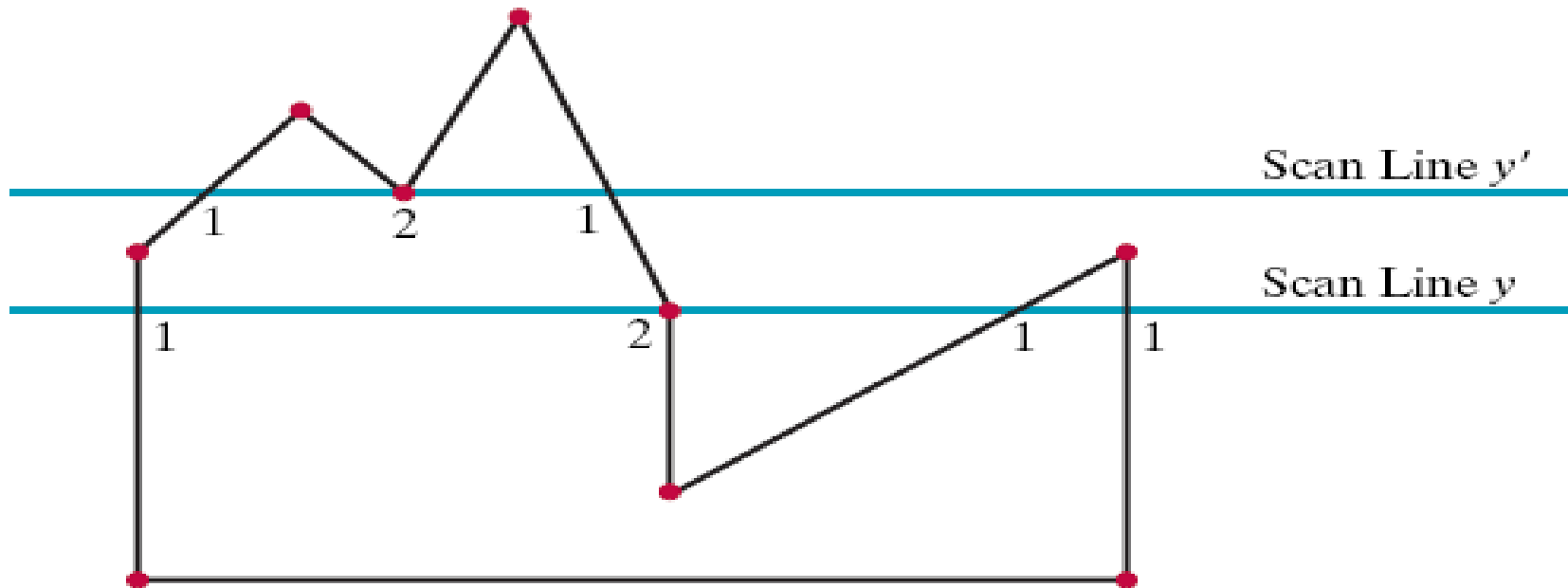


FIGURE 4-21 Intersection points along scan lines that intersect polygon vertices. Scan line y generates an odd number of intersections, but scan line y' generates an even number of intersections that can be paired to identify correctly the interior pixel spans.

Polygon Fill Algorithm

- To identify the interior pixels for scan line y , we must count the vertex intersection as only one point.
- Thus, as we process scan lines, we need to distinguish between these two cases.

Polygon Fill Algorithm

- We can detect the difference between the two cases by noting the position of the intersecting edges relative to the scan line.
- For scan line y , the two edges sharing an intersection vertex are on opposite sides of the scan line.
- But for scan line y' , the two intersecting edges are both above the scan line.

Polygon Fill Algorithm

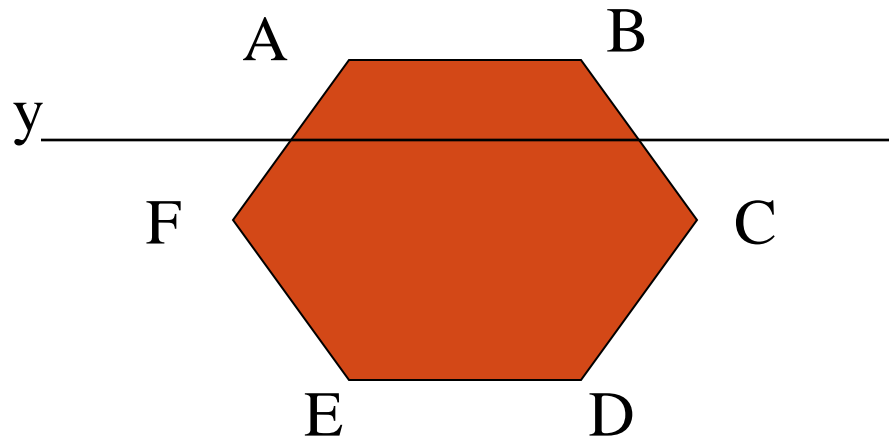
- A vertex that has adjoining edges on opposite sides of an intersecting scan line should be counted as just one boundary intersection point.
- We can identify these vertices by tracing around the polygon boundary in either clockwise or counterclockwise order and observing the relative changes in vertex y coordinates as we move from one edge to the next.

Polygon Fill Algorithm

- If the three endpoint y values of two consecutive edges increase or decrease, we need to count the shared (middle) vertex as a single intersection point for the scan line passing through that vertex.
- Otherwise, the shared vertex represents a local extremum (minimum or maximum) on the polygon boundary, and the two edge intersections with the scan line passing through that vertex can be added to the intersection list.

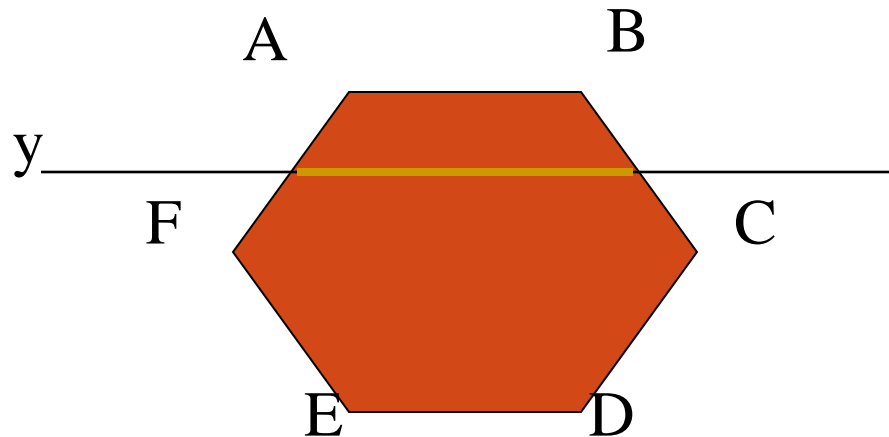
Scan-line polygon-fill algorithm

- For convex polygons.
 - Determine the intersection positions of the boundaries of the fill region with the screen scan lines.



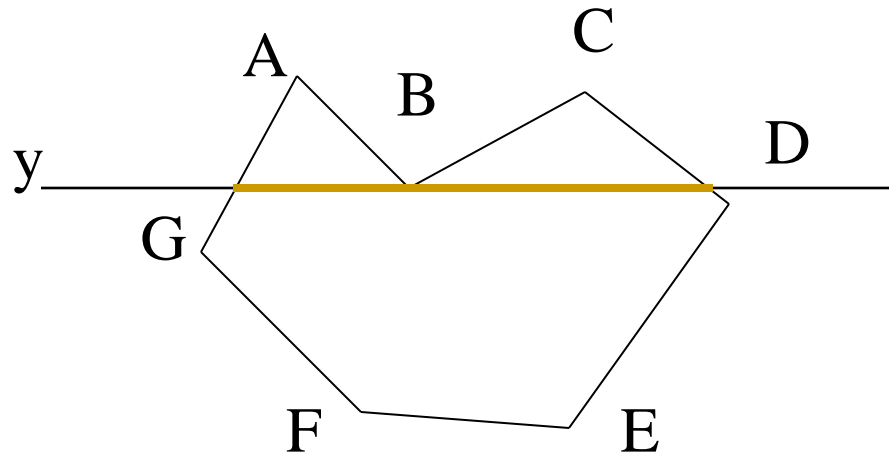
Scan-line polygon-fill algorithm

- For convex polygons.
- Pixel positions between pairs of intersections between scan line and edges are filled with color, including the intersection pixels.



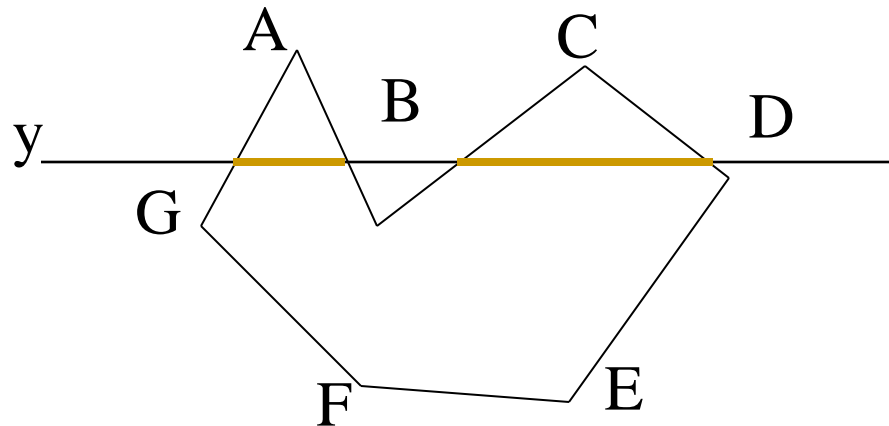
Scan-line polygon-fill algorithm

- For concave polygons.
 - Scan line may intersect more than once:
 - Intersects an even number of edges
 - Even number of intersection vertices yields to pairs of intersections, which can be filled as previously



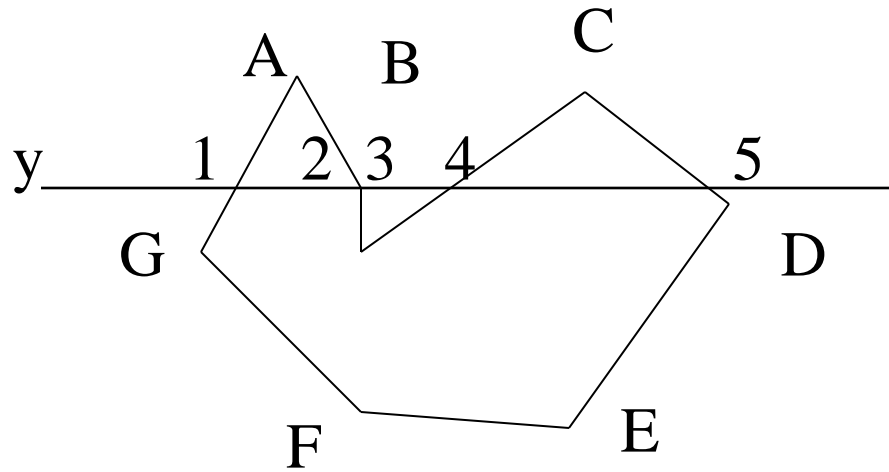
Scan-line polygon-fill algorithm

- For concave polygons.
 - Scan line may intersect more than once:
 - Intersects an even number of edges
 - Even number of intersection vertices yields to pairs of intersections, which can be filled as previously



Scan-line polygon-fill algorithm

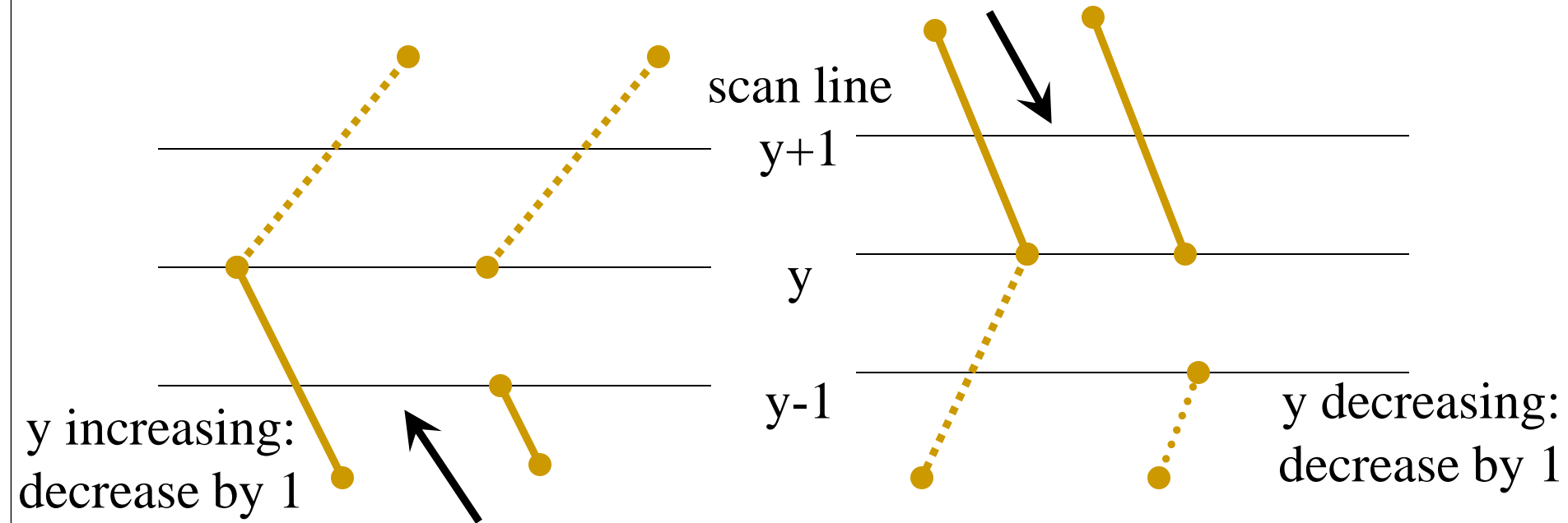
- For concave polygons.
 - Scan line may intersect more than once:
 - Intersects an **odd** number of edges
 - Not all pairs are interior: (3,4) is not interior



Scan-line polygon-fill algorithm

- For concave polygons.
 - Generate 2 intersections when at a local minimum, else generate only one intersection.
 - Algorithm to determine whether to generate one intersection or 2 intersections.
 - If the y-coordinate is monotonically increasing or decreasing, decrease the number of vertices by shortening the edge.
 - If it is not monotonically increasing or decreasing, leave the number of vertices as it is.

Scan-line polygon-fill algorithm



The y -coordinate of the upper endpoint of the current edge is decreased by 1.

The y -coordinate of the upper endpoint of the next edge is decreased by 1.

Area Fill Algorithm

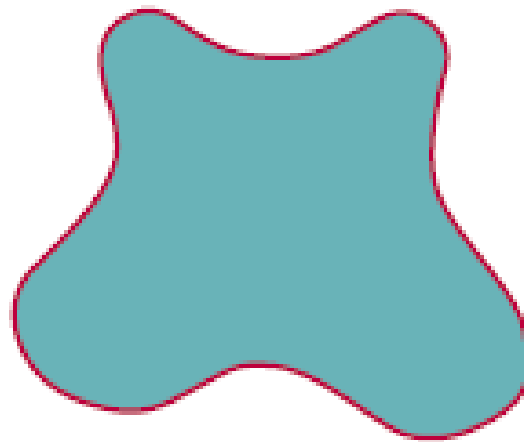
- An alternative approach for filling an area is to start at a point inside the area and “paint” the interior, point by point, out to the boundary.
- This is a particularly useful technique for filling areas with irregular borders, such as a design created with a paint program.
- The algorithm makes the following assumptions
 - one interior pixel is known, and
 - pixels in boundary are known.

Area Fill Algorithm

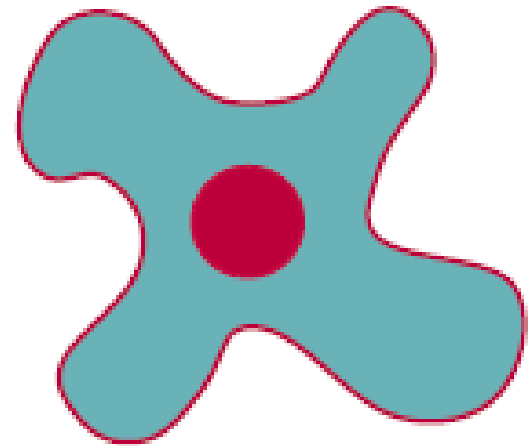
- If the boundary of some region is specified in a single color, we can fill the interior of this region, pixel by pixel, until the boundary color is encountered.
- This method, called the **boundary-fill algorithm**, is employed in interactive painting packages, where interior points are easily selected.

Example

- One can sketch a figure outline, and pick an interior point.
- The figure interior is then painted in the fill color as shown in these Figures



(a)



(b)

FIGURE 4-26 Example color boundaries for a boundary-fill procedure.

Area Fill Algorithm

- Basically, a boundary-fill algorithm starts from an interior point (x, y) and sets the neighboring points to the desired color.
- This procedure continues until all pixels are processed up to the designated boundary for the area.

Area Fill Algorithm

- There are two methods for processing neighboring pixels from a current point.
 1. Four neighboring points.
 - These are the pixel positions that are right, left, above, and below the current pixel.
 - Areas filled by this method are called **4-connected**.

Area Fill Algorithm

2. Eight neighboring points.

- This method is used to fill more complex figures.
- Here the set of neighboring points to be set includes the four diagonal pixels, in addition to the four points in the first method.
- Fill methods using this approach are called **8-connected**.

Area Fill Algorithm

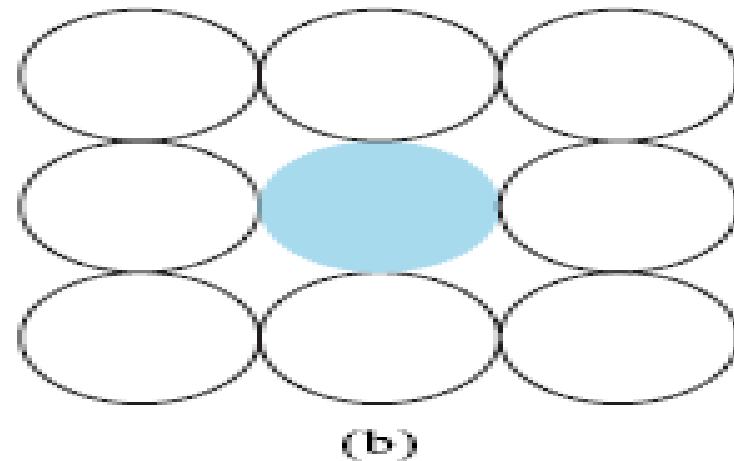
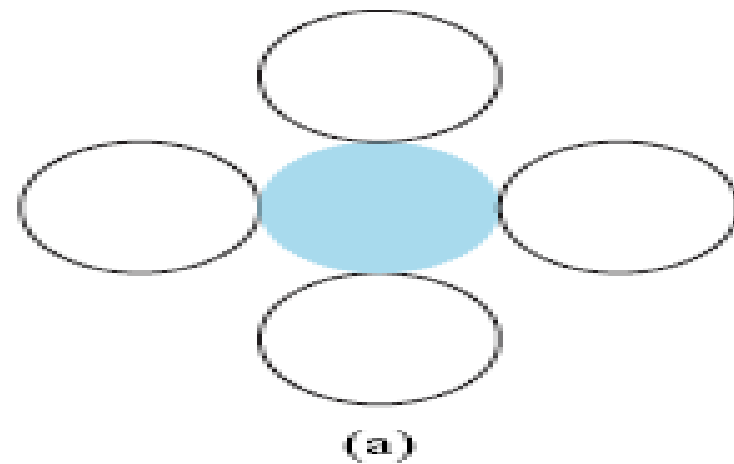


FIGURE 4-27 Fill methods applied to a 4-connected area (a) and to an 8-connected area (b). Hollow circles represent pixels to be tested from the current test position, shown as a solid color.

Area Fill Algorithm

- Consider the Figure in the next slide.
- An 8-connected boundary-fill algorithm would correctly fill the interior of the area defined in the Figure.
- But a 4-connected boundary-fill algorithm would only fill part of that region.

Area Fill Algorithm

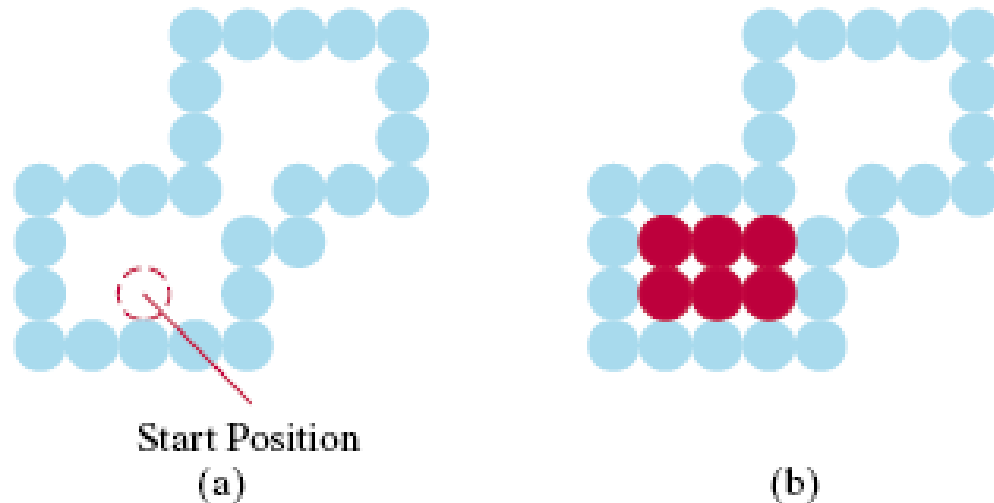


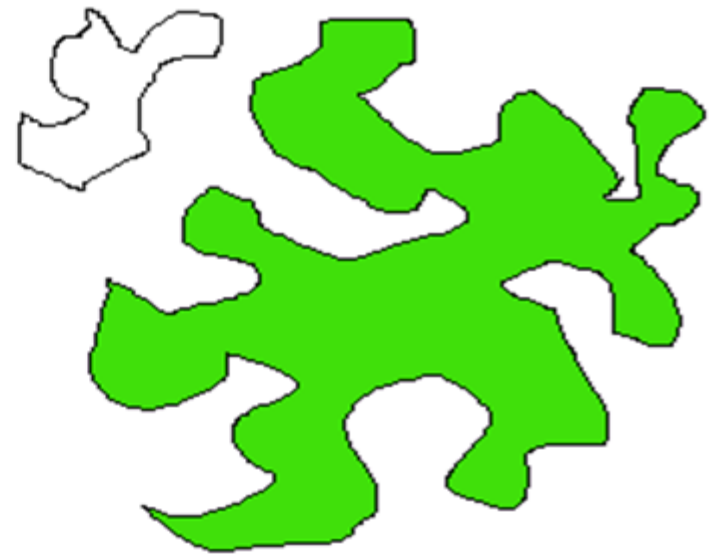
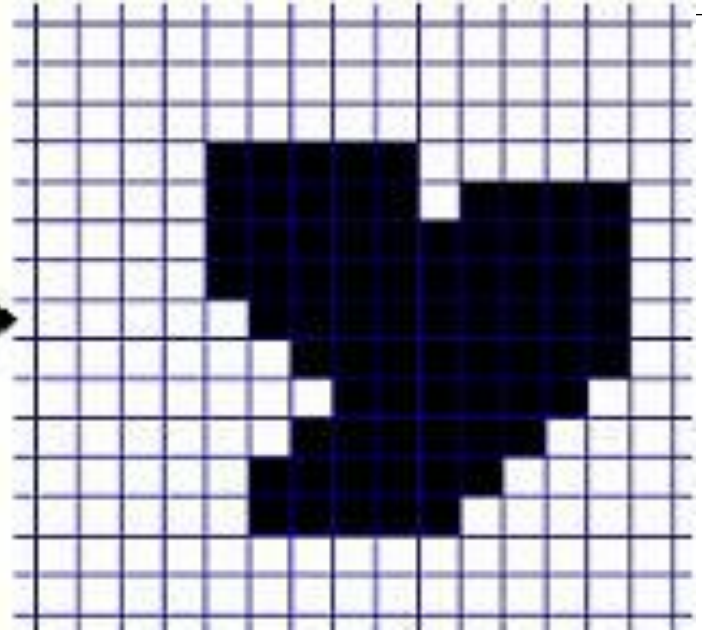
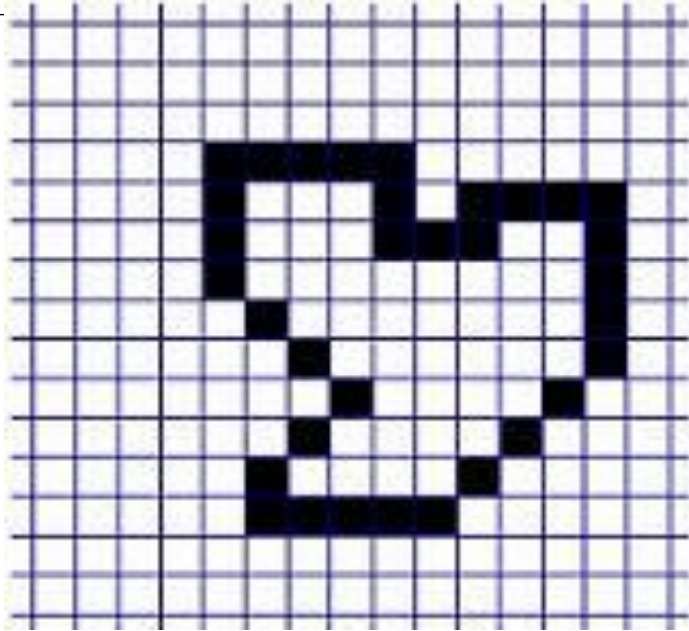
FIGURE 4-28 The area defined within the color boundary (a) is only partially filled in (b) using a 4-connected boundary-fill algorithm.

Area Fill Algorithm

- The following procedure illustrates a recursive method for painting a 4-connected area with a solid color, specified in parameter **fillColor**, up to a boundary color specified with parameter **borderColor**.
- We can extend this procedure to fill an 8-connected region by including four additional statements to test the diagonal positions $(x \pm 1, y \pm 1)$.

Area Fill Algorithm

```
void boundaryFill4 (int x, int y, int fillColor, int borderColor)
{
    int interiorColor;
    /* Set current color to fillColor, then perform following oprations. */
    getPixel (x, y, interiorColor);
    if ((interiorColor != borderColor) && (interiorColor != fillColor))
    {
        setPixel (x, y); // Set color of pixel to fillColor.
        boundaryFill4 (x + 1, y , fillColor, borderColor);
        boundaryFill4 (x - 1, y, fillColor, borderColor);
        boundaryFill4 (x , y + 1, fillColor, borderColor);
        boundaryFill4 (x , y - 1, fillColor, borderColor)
    }
}
```



Area Fill Algorithm

- Some times we want to fill in (or recolor) an area that is not defined within a single color boundary.
- Consider the following Figure.



FIGURE 4-30 An area defined within multiple color boundaries.

Area Fill Algorithm

- We can paint such areas by replacing a specified interior color instead of searching for a particular boundary color.
- This fill procedure is called a **flood-fill algorithm**.

Area Fill Algorithm

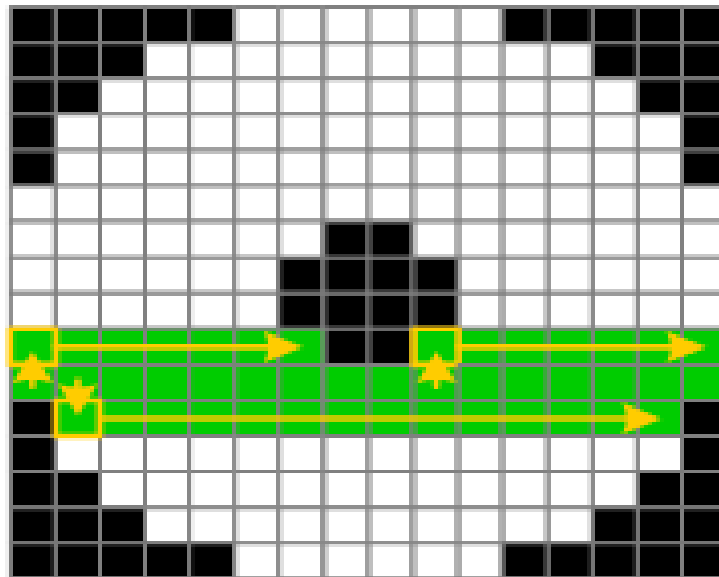
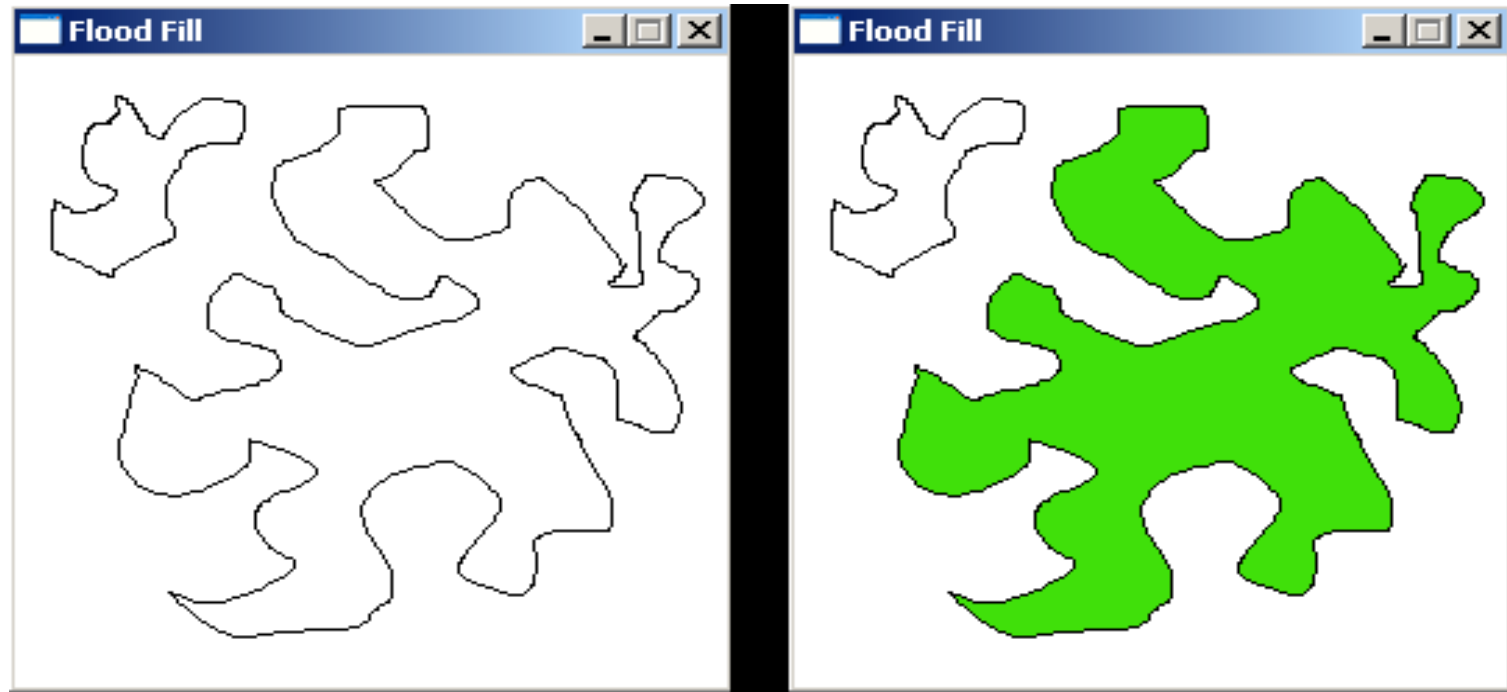
- We start from a specified interior point (x, y) and re-assign all pixel values that are currently set to a given interior color with the desired fill color.
- If the area we want to paint has more than one interior color, we can first re-assign pixel values so that all interior points have the same color.

Area Fill Algorithm

- Using either a 4-connected or 8-connected approach, we then step through pixel positions until all interior points have been repainted.
- It is particularly useful where the region to be filled has no uniform boundary.
- The following procedure flood fills a 4-connected region recursively, starting from the input position.

Area Fill Algorithm

```
void floodFill4 (int x, int y, int fillColor, int interiorColor)
{
    int color;
    /* Set current color to fillColor, then perform following operations. */
    getPixel (x, y, color);
    if (color == interiorColor) {
        setPixel (x, y);    // Set color of pixel to fillColor.
        floodFill4 (x + 1, y, fillColor, interiorColor);
        floodFill4 (x - 1, y, fillColor, interiorColor);
        floodFill4 (x, y + 1, fillColor, interiorColor);
        floodFill4 (x, y - 1, fillColor, interiorColor)
    }
}
```

Problems with Fill Algorithm (1)

- Recursive boundary-fill algorithms may not fill regions correctly if some interior pixels are already displayed in the fill color.
- This occurs because the algorithm checks next pixels both for boundary color and for fill color.

Problems with Fill Algorithm

- To avoid this, we can first change the color of any interior pixels that are initially set to the fill color before applying the boundary-fill procedure.
- Encountering a pixel with the fill color can cause a recursive branch to terminate, leaving other interior pixels unfilled.

Problems with Fill Algorithm (2)

- This procedure requires considerable stacking of neighboring points, more efficient methods are generally employed.
- These methods fill horizontal pixel spans across scan lines, instead of proceeding to 4-connected or 8-connected neighboring points.

Problems with Fill Algorithm (2)

- Then we need only stack a beginning position for each horizontal pixel span, instead of stacking all unprocessed neighboring positions around the current position.
- Starting from the initial interior point with this method, we first fill in the contiguous span of pixels on this starting scan line.

Problems with Fill Algorithm (2)

- Then we locate and stack starting positions for spans on the adjacent scan lines, where spans are defined as the contiguous horizontal string of positions bounded by pixels displayed in the border color.
- At each subsequent step, we retrieve the next start position from the top of the stack and repeat the process.

Area Fill Algorithm

The algorithm can be summarized as follows:

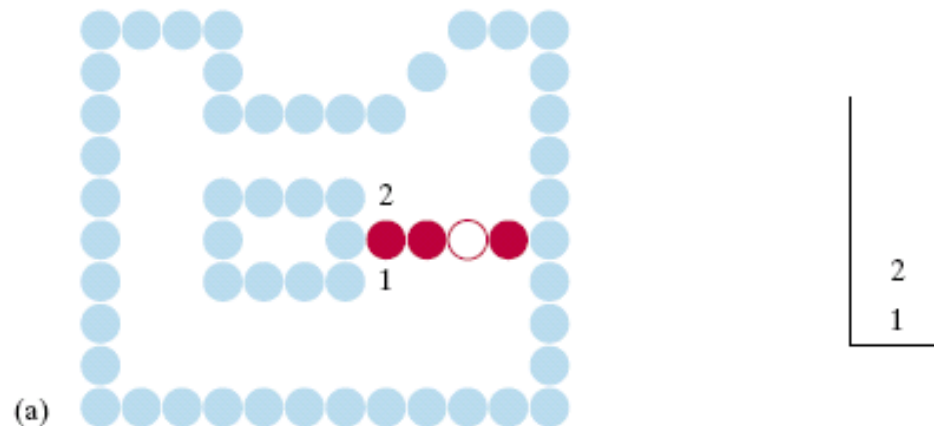
1. define seed point,
2. fill scan line containing seed point,
3. for scan lines above and below, define new seed points as:
 - i) first point inside left boundary,
 - ii) subsequent points within boundary whose left neighbor is outside,
4. d) repeat algorithm with the new set of seed points.

Example

- In this example, we first process scan lines successively from the start line to the top boundary.
- After all upper scan lines are processed, we fill in the pixel spans on the remaining scan lines in order down to the bottom boundary.
- The leftmost pixel position for each horizontal span is located and stacked, in left to right order across successive scan lines.

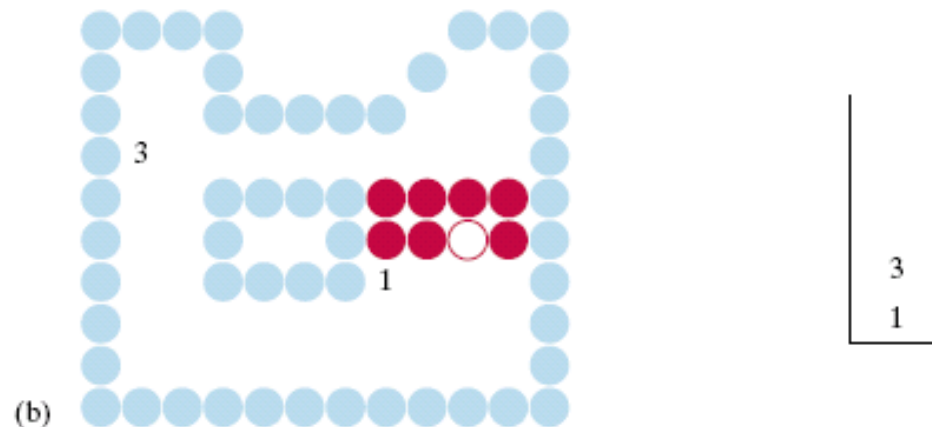
Example

- In (a) of this figure, the initial span has been filled, and starting positions 1 and 2 for spans on the next scan lines (below and above) are stacked.



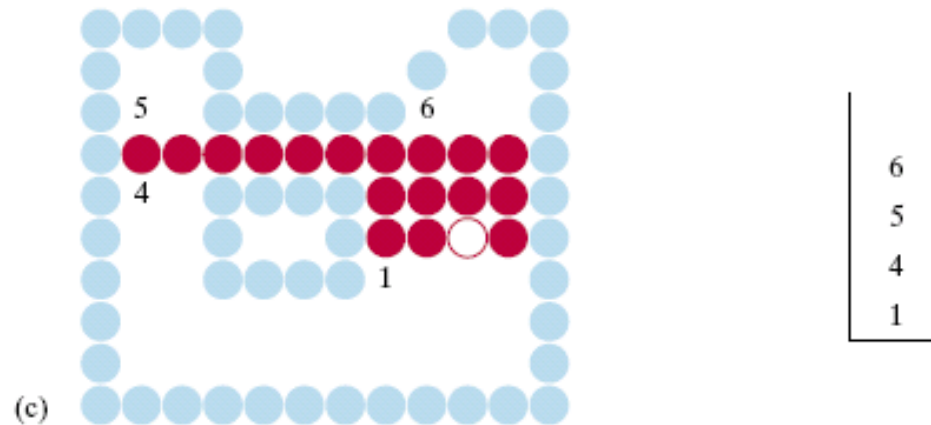
Example

- In Fig.(b), position 2 has been unstacked and processed to produce the filled span shown, and the starting pixel (position 3) for the single span on the next scan line has been stacked.



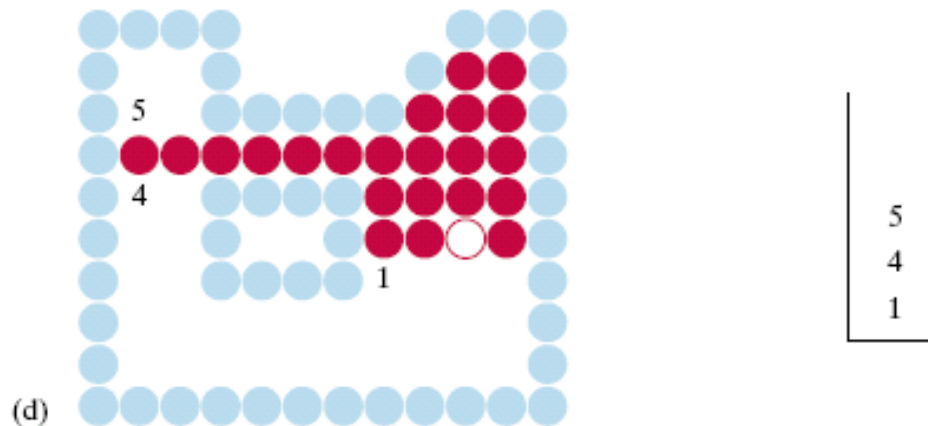
Example

- After position 3 is processed, the filled spans and stacked positions are as shown in Fig. (c).



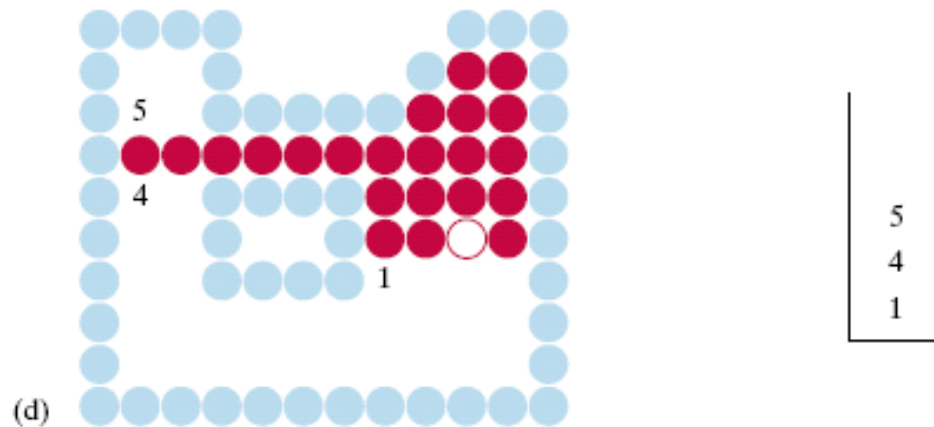
Example

- And Fig.(d) shows the filled pixels after processing all spans in the upper right of the specified area.



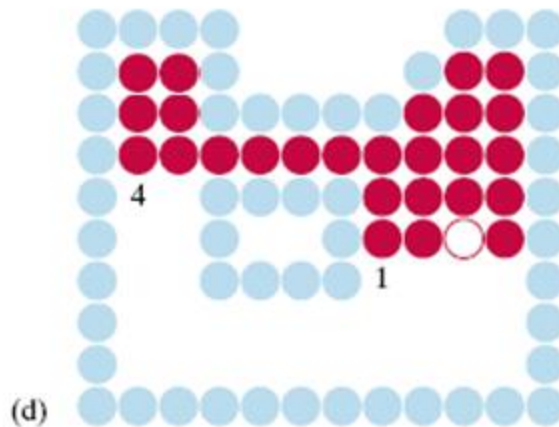
Example

- Position 5 is next processed, and spans are filled in the upper left of the region; then position 4 is picked up to continue the processing for the lower scan lines.



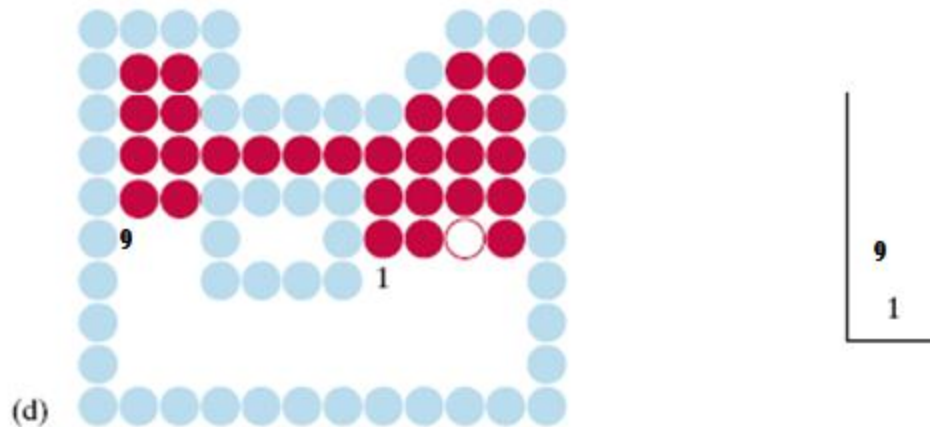
Example

- Finish up the upper scan lines.



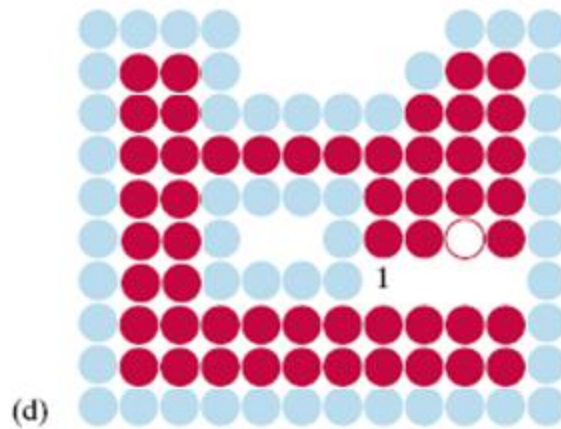
Example

- Start the bottom scan lines.



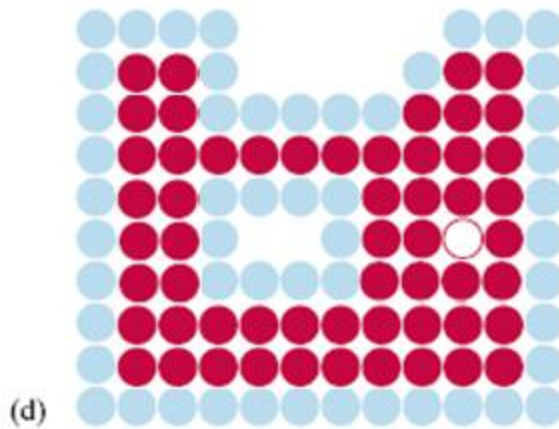
Example

- Finish up the bottom scan lines.



Example

- Finish up the bottom scan lines.



Aliasing

- Aliasing definition: Distortion of information due to under-sampling.
- In digital signal processing, **anti-aliasing** is the technique of minimizing aliasing (jagged or blocky patterns) when representing a high-resolution signal at a lower resolution.

Anti-Aliasing

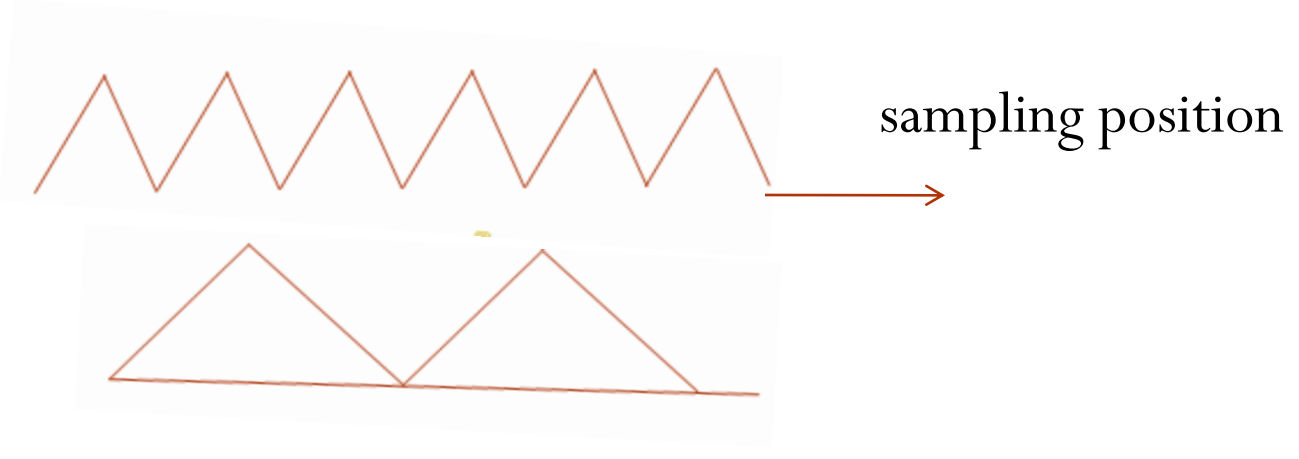
- Since we're representing real-world objects with a finite number of pixels, aliasing occurs frequently.
- Therefore, we need to implement techniques to cancel the undesirable effects of aliasing.
- These techniques are called *anti-aliasing* techniques.
- One common anti-aliasing method is super-sampling

Hello World

Hello World

A demonstration

- An example of the effects of under sampling is shown in this figure :



- We need to sample frequency to at least that of highest frequency occurring in the object, referred as **Nyquist sampling frequency**.

$$f_s = 2f_{\max}$$

- Another way is to state this is that the sampling interval should be no larger than one half the cycle interval called as the **Nyquist sampling interval**(Δx_s).

$$\Delta x_s = \Delta x_{\text{cycle}} / 2$$

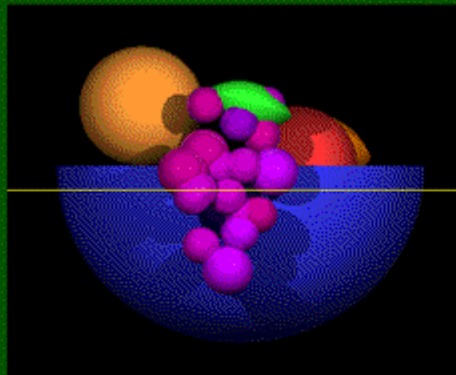
$$\text{Where } \Delta x_{\text{cycle}} = 1 / f_{\text{max}}$$

Types of anti-aliasing techniques

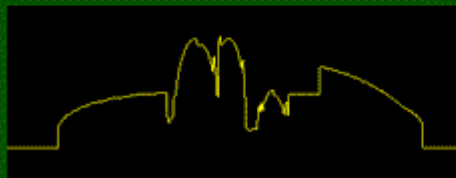
- **Pre filtering** : based on amount of pixels covered by an object.
- **Supersampling** : it tries to reduce the effect of aliasing by taking more than one sample per pixel.
- **Post filtering** : post filtering and supersampling is almost same. Giving weightage of 50% to center point.
- **Pixel Phasing** : it's a hardware based anti-aliasing techniques based on shifting of pixel position by a fraction.
- **Grey Level** : by combining black & white grey level is used to show the image.

Example

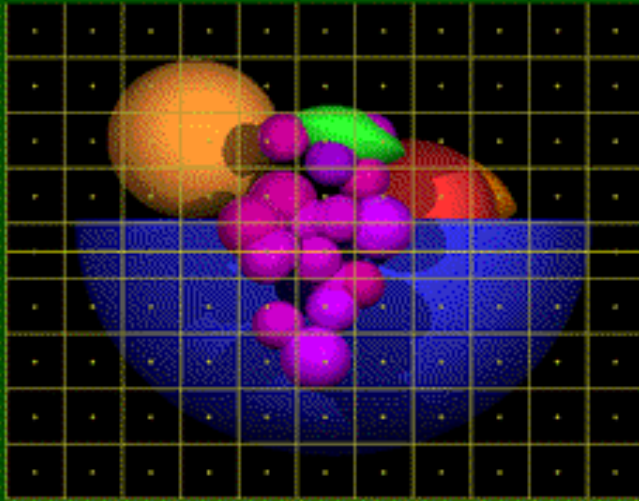
- The bowl of fruit was modeled using constructive solid geometry.



**Original
scene**



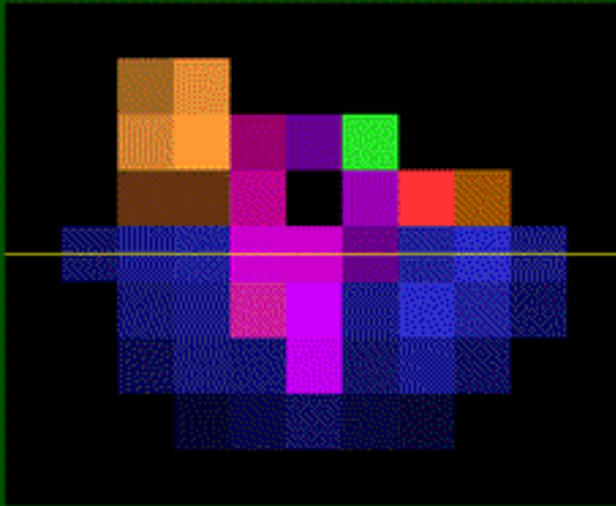
**Luminosity
signal**



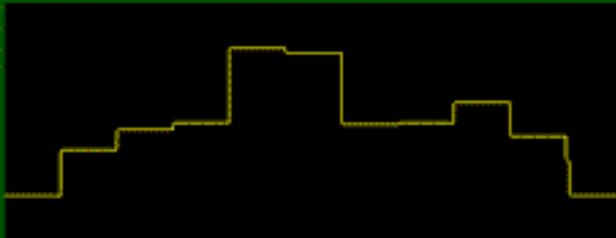
**Sampling at
pixel centers**



**Sampled
signal**



**Rendered
image**



**Luminosity
signal**

Character Generation

Character Generation

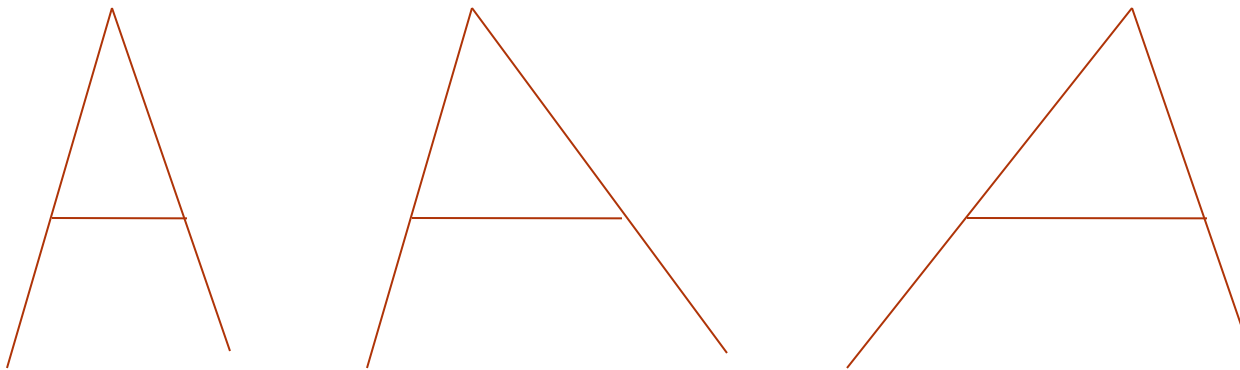
Letters, numbers, and other character can be displayed in a variety of size and styles.

Character Generation

- Two different representation are used for storing computer fonts:
 1. Outline font : Stroke method/vector character generation method
 2. Bitmap font (or bitmapped font) :

Stroke method/vector character generation method

- This method uses line segment to created characters.
- In each pixel square co-ordinates (x,y) are defined to draw character.
- By using this we can change the scale of characters.



Outline Font

- Graphic primitives such as lines and arcs are used to define the outline of each character.
- Require less storage since variation does not require a distinct



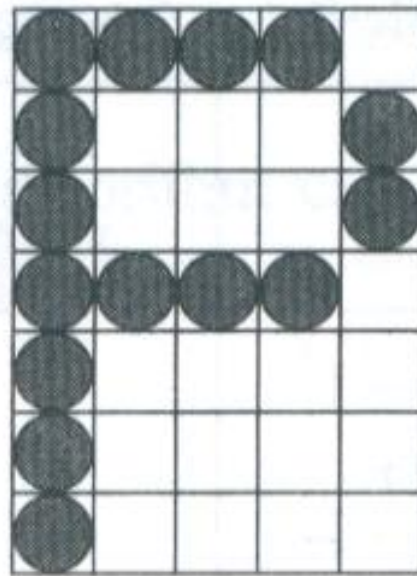
Outline Font

- We can produce boldface, italic, or different size by manipulating the curve definition for the character outlines.
- It does take more time to process the outline fonts, because they must be scan converted into frame buffer.

Outline font

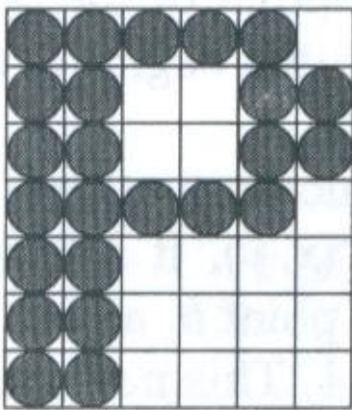
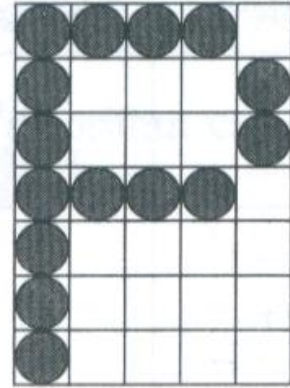
Bitmap font

- **Bitmap font (or bitmapped font):** A simple method for representing the character shapes in a particular typeface is to use rectangular grid pattern.

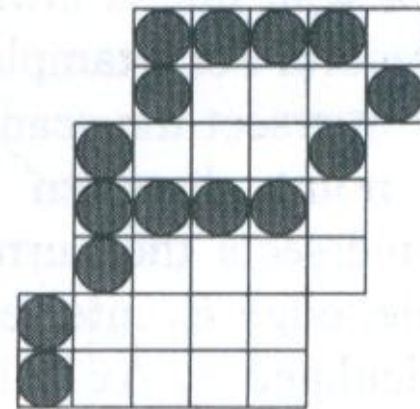


Bitmap font

- The character grid only need to be mapped to a frame buffer position.
- Bitmap fonts required more space, because each variation (size and format) must be stored in a *font cash*.



Bold



Italic