Unit 3

Raster Scan Graphics

Unit 3: Raster Scan Graphics

- Introduction to line drawing algorithm
- Digital Differential Analyzer (DDA)
- Bresenham's Line drawing algorithm
- Integer Bresenham's Line Drawing Algorithm
- Generalized Integer Bresenham's Line Drawing Algorithm
- Bresenham's Circle drawing algorithm
- Scan Conversion techniques: RLE, Frame Buffer
- Scan converting polygons: Edge fill and Seed fill algorithms
- Anti-aliasing

LINE DRAWING

- A CRT raster display is a matrix of discrete finite area cells (pixels), each of which can be made bright. Therefore it is not possible to draw a straight line from one point to another.
- The process of determining which pixel provide best approximation to the desired line is known as **rasterization**.
- When it combined with the process of generating the picture in scan line order, it is known as **scan conversion**.

When considering specific line drawing algorithm, goals are :

- Straight lines should appear straight.
- Lines should start and end accurately, matching endpoints with connecting lines.
- Lines should have constant brightness.
- Lines should be drawn as rapidly as possible.

But not all of them are achievable with the discrete space of a raster device.

Rasterization of Straight Line

- For horizontal, vertical and 45^0 – the brightness constant along the line.
- All other cases – the rasterization generates uneven brightness

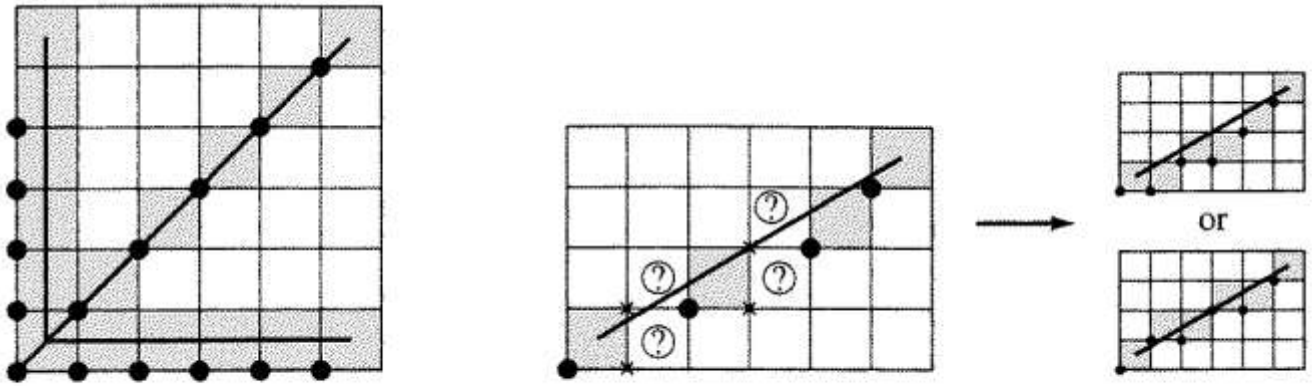


Figure 2–1 Rasterization of straight lines.

Digital Differential Analyzer (DDA)

One technique for obtaining a rasterized straight line is to solve the governing differential equation for a straight line, i.e.

$$\frac{dy}{dx} = \text{constant} \qquad \text{or} \qquad \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}$$

The solution of the finite difference approximation is

$$y_{i+1} = y_i + \Delta y$$

$$y_{i+1} = y_i + \frac{y_2 - y_1}{x_2 - x_1}\Delta x \qquad (2-1)$$

where x_1, y_1 and x_2, y_2 are the end points of the required straight line, and y_i is the initial value for any given step along the line. In fact, Eq. $(2-1)$ represents a recursion relation for successive values of y along the required line. Used to rasterize a line, it is called a digital differential analyzer (DDA).[1] For a simple DDA, either Δx or Δy, whichever is larger, is chosen as one raster unit. A simple algorithm which works in all quadrants is:

Digital Differential Analyzer (DDA)

digital differential analyzer *(DDA) routine for rasterizing a line*

the line end points are (x_1, y_1) *and* (x_2, y_2)*, assumed not equal*

Integer *is the integer function. Note: Many Integer functions are floor functions; i.e.,* $Integer(-8.5) = -9$ *rather than* -8*. The algorithm assumes this is the case.*

Sign *returns* $-1, 0, 1$ *for arguments* $< 0, = 0, > 0$*, respectively*

approximate the line length

if $\text{abs}(x_2 - x_1) \geq \text{abs}(y_2 - y_1)$ **then**
    $\text{Length} = \text{abs}(x_2 - x_1)$
else
    $\text{Length} = \text{abs}(y_2 - y_1)$
end if

select the larger of Δx *or* Δy *to be one raster unit*

$\Delta x = (x_2 - x_1)/\text{Length}$
$\Delta y = (y_2 - y_1)/\text{Length}$

Digital Differential Analyzer (DDA)

```
    round the values rather than truncate, so that center pixel addressing
       is handled correctly
    x = x1 + 0.5
    y = y1 + 0.5
    begin main loop
    i = 1
    while (i ≤ Length)
        setpixel (Integer(x), Integer(y))
        x = x + Δx
        y = y + Δy
        i = i + 1
    end while
finish
```

Digital Differential Analyzer (DDA)

Example 2–1 Simple DDA First Quadrant

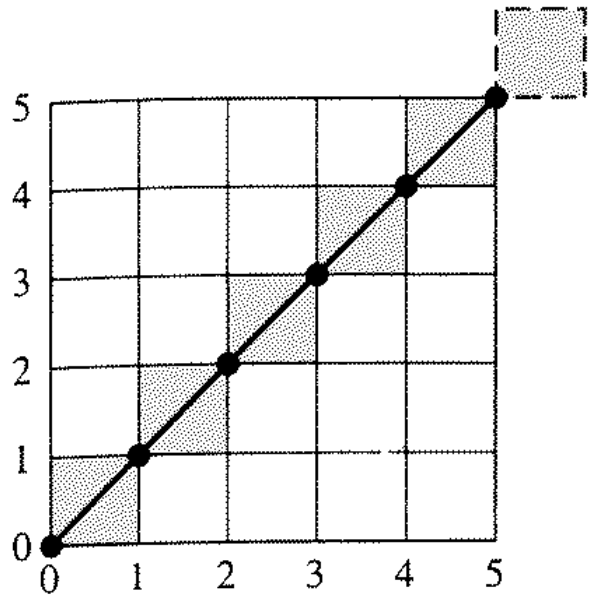
Consider the line from (0, 0) to (5, 5). Use the simple DDA to rasterize this line. Evaluating the steps in the algorithm yields

initial calculation

$x_1 = 0$
$y_1 = 0$
$x_2 = 5$
$y_2 = 5$
Length $= 5$
$\Delta x = 1$
$\Delta y = 1$
$x = 0.5$
$y = 0.5$

Incrementing through the main loop yields

i	setpixel	x	y
		0.5	0.5
1	(0, 0)		
		1.5	1.5
2	(1, 1)		
		2.5	2.5
3	(2, 2)		
		3.5	3.5
4	(3, 3)		
		4.5	4.5
5	(4, 4)		
		5.5	5.5



Digital Differential Analyzer (DDA)

Example 2–2 Simple DDA in the Third Quadrant

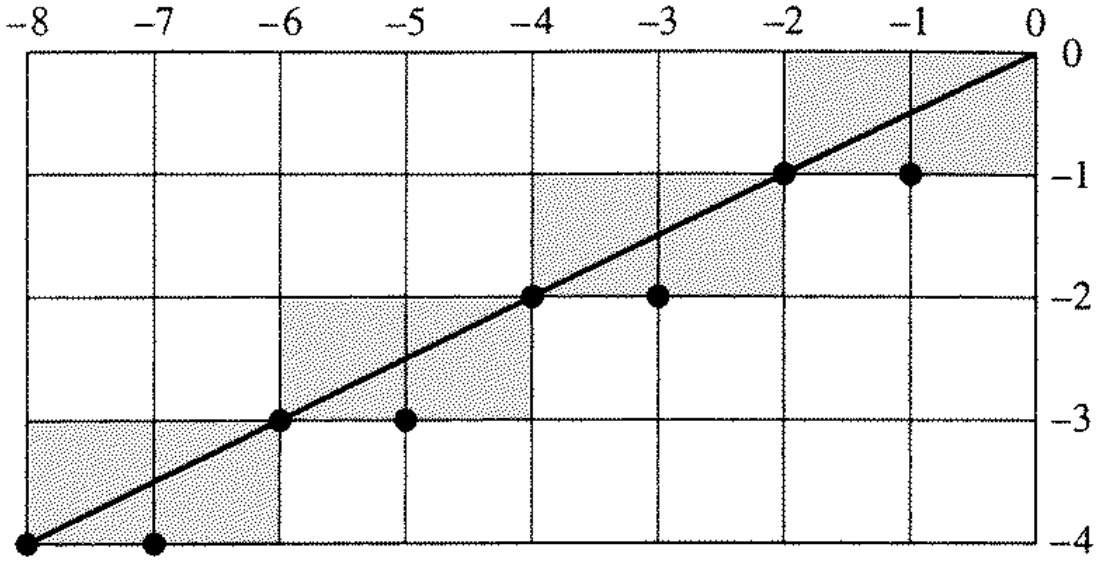
Consider the line from $(0, 0)$ to $(-8, -4)$ in the third quadrant. Evaluating the algorithm yields

initial calculations

$x_1 = 0$
$y_1 = 0$
$x_2 = -8$
$y_2 = -4$
$\text{Length} = 8$
$\Delta x = -1$
$\Delta y = -0.5$
$x = -0.5$
$y = -0.5$

Incrementing through the main loop,

i	setpixel	x	y
		0.5	0.5
1	(0, 0)		
		−0.5	0
2	(−1, 0)		
		−1.5	−0.5
3	(−2, −1)		
		−2.5	−1.0
4	(−3, −1)		
		−3.5	−1.5
5	(−4, −2)		
		−4.5	−2.0
6	(−5, −2)		
		−5.5	−2.5
7	(−6, −3)		
		−6.5	−3.0
8	(−7, −3)		
		−7.5	−3.5



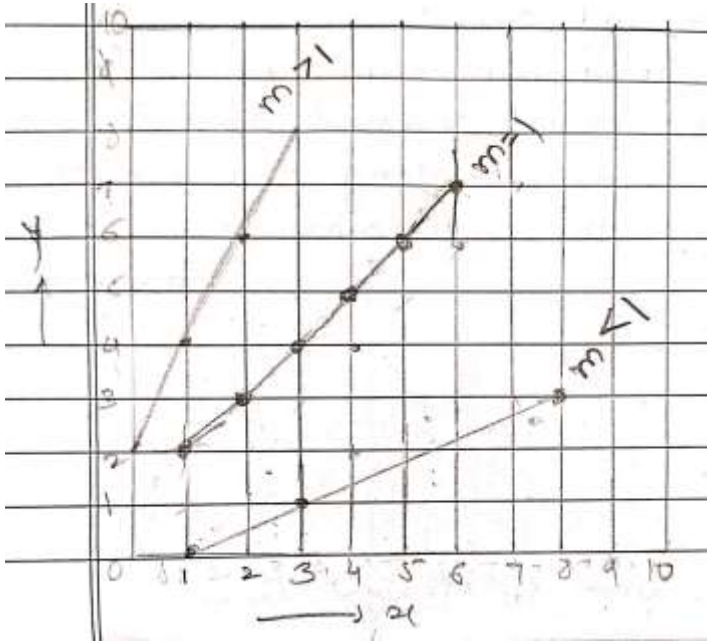
Digital Differential Analyzer (DDA)

- **Advantages of Digital Differential Analyzer**
  - It is a simple algorithm to implement.
  - It is a faster algorithm than the direct line equation.
  - We cannot use the multiplication method in Digital Differential Analyzer.
  - Digital Differential Analyzer algorithm tells us about the overflow of the point when the point changes its location.
- **Disadvantages of Digital Differential Analyzer**
  - The floating-point arithmetic implementation of the Digital Differential Analyzer is time-consuming.
  - The method of round-off is also time-consuming.
  - Sometimes the point position is not accurate.

Bresenham's Line Drawing Algorithm

- Equation of Line : $y = mx + c$

  where m = slope of line and c = y intercept
- If line is at 45^0 then $m = \Delta y/\Delta x = 1$
- If line is at $< 45^0$ then $m = \Delta y/\Delta x < 1$
- If line is at $> 45^0$ then $m = \Delta y/\Delta x > 1$



- A line equation $y = mx + c$ is called as vertex.
- This equation is not useful to draw a line on screen because screen is collection of pixel. i.e. called as raster (x1, y1) and (x2, y2).
- Converting a vector to raster is called as rasterization.

Bresenham's Line Drawing Algorithm

- If the line is at 45^0, m = 1 then both x and y value will increased.
- If the line is at < 45^0, m < 1 then the value of x will increase always and value of y will or will not be increase.
- If the line is at > 45^0, m > 1 then the value of y will increase always and value of x will or will not be increase.

  Now,

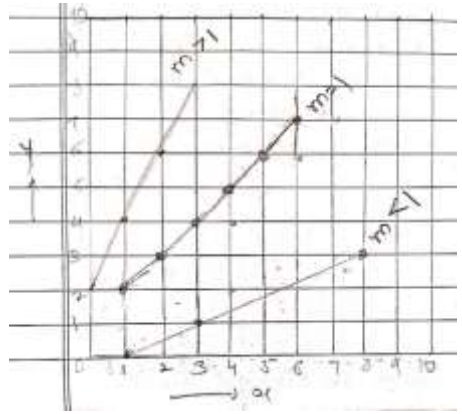
  $\Delta x = x2 - x1$

  $\Delta y = y2 - y1$

  Length = max (Δx, Δy)

  $x_{increment} = \Delta x/length$

  $y_{increment} = \Delta y/length$



- Ex : (x1, y1) = (2, 4) and (x2, y2) = (8, 6) : x will increase always and value of y will or will not be increase.

Bresenham's Line Drawing Algorithm

- This algorithm seeks to select the optimum raster locations that represent a straight line.
- To accomplish this, the algorithm always increments by one unit in either x or y, depending on the slope of line.
- The increment in other variable, either 0 or 1, is determined by examining the distance between the actual line and the nearest grid locations. This distance is called the error.

$\frac{1}{2} \le \frac{\Delta y}{\Delta x} \le 1$ (error ≥ 0)
Plot (1,1)

$0 \le \frac{\Delta y}{\Delta x} < \frac{1}{2}$ (error < 0)
Plot (1,0)

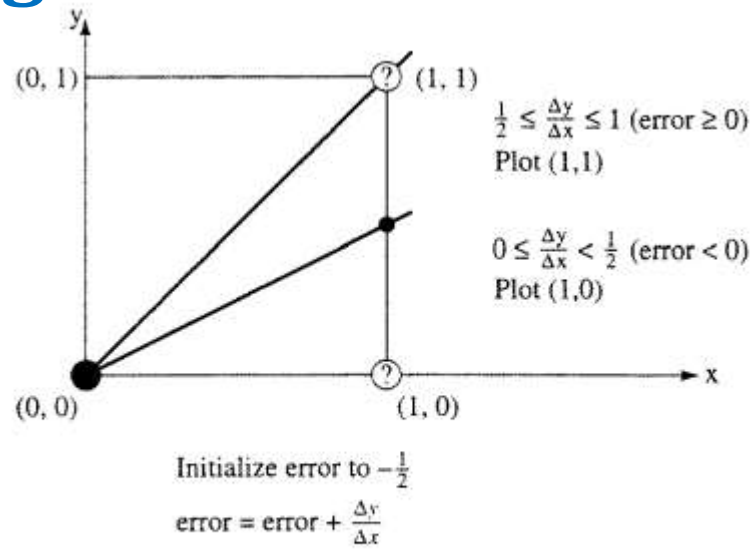
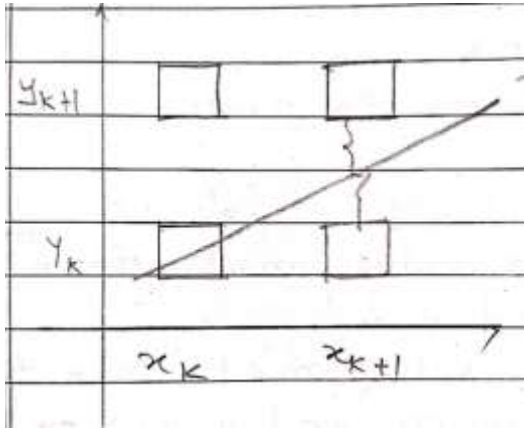


Figure 2–4 Basis of Bresenham's algorithm.



Bresenham's Line Drawing Algorithm

Bresenham's line rasterization algorithm for the first octant

the line end points are (x_1, y_1) *and* (x_2, y_2)*, assumed not equal*
Integer *is the integer function*
x, y, Δx, Δy *are assumed integer;* e *is real*

```
    initialize variables
    x = x1
    y = y1
    Δx = x2 − x1
    Δy = y2 − y1
    m = Δy/Δx
    initialize e to compensate for a nonzero intercept
    e = m − 1/2
    begin the main loop
    for i = 1 to Δx
        setpixel (x, y)
        while (e > 0)
            y = y + 1
            e = e − 1
        end while
        x = x + 1
        e = e + m
    next i
finish
```

- If $\Delta_y > \Delta_x$, then exchange values of Δ_y and Δ_x

Bresenham's Line Drawing Algorithm

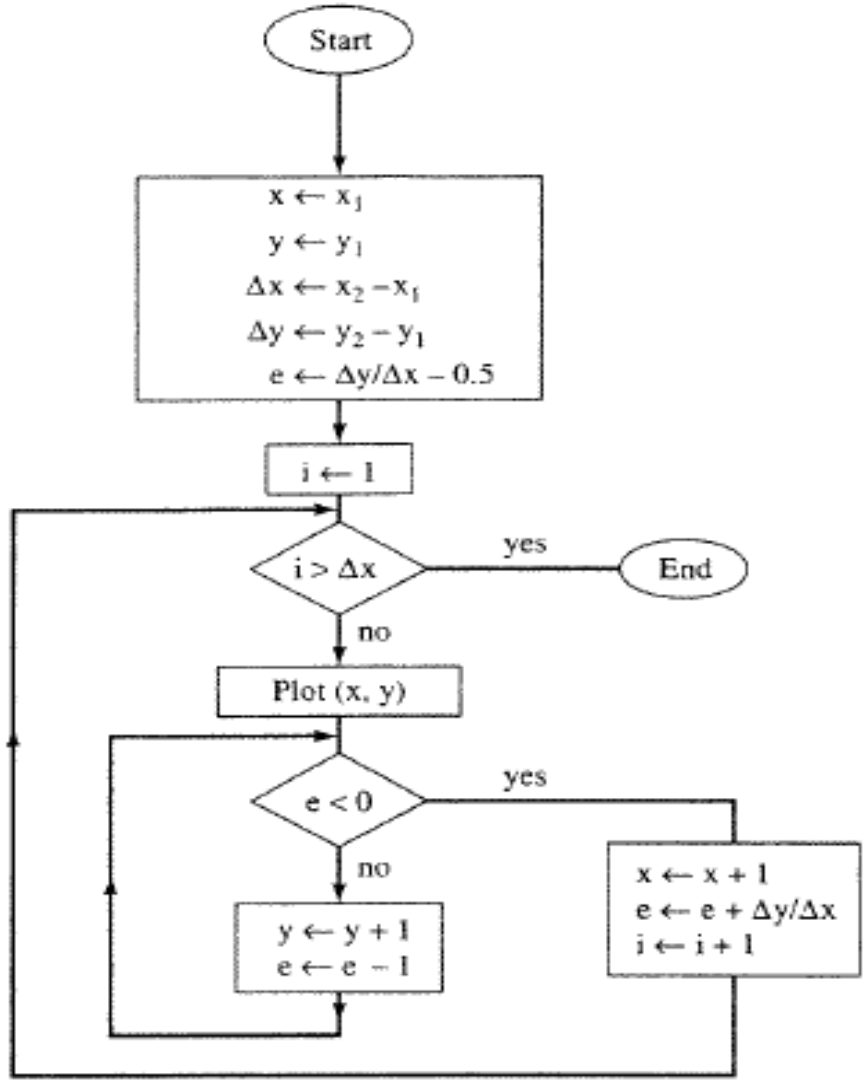


Figure 2–6 Flowchart for Bresenham's algorithm.

Bresenham's Line Drawing Algorithm-Example

Example 2–3 Bresenham's Algorithm

Consider the line from $(0, 0)$ to $(5, 5)$. Rasterizing the line with the Bresenham algorithm yields

initial calculations

$x = 0$
$y = 0$
$\Delta x = 5$
$\Delta y = 5$
$m = 1$
$e = 1 - 1/2 = 1/2$

Bresenham's Line Drawing Algorithm-Example

Incrementing through the main loop yields

i	setpixel	e	x	y
		$\frac{1}{2}$	0	0
1	(0,0)			
		$-\frac{1}{2}$	0	1
		$\frac{1}{2}$	1	1
2	(1,1)			
		$-\frac{1}{2}$	1	2
		$\frac{1}{2}$	2	2
3	(2,2)			
		$-\frac{1}{2}$	2	3
		$\frac{1}{2}$	3	3
4	(3,3)			
		$-\frac{1}{2}$	3	4
		$\frac{1}{2}$	4	4
5	(4,4)			
		$-\frac{1}{2}$	4	5
		$\frac{1}{2}$	5	5

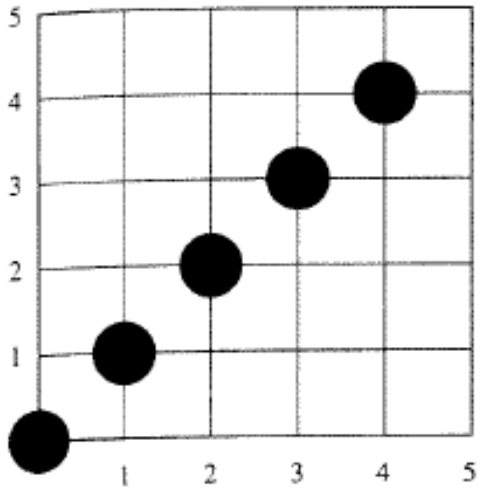


Figure 2–7 Results for the Bresenham algorithm in the first octant.

Integer Bresenham's Line Drawing Algorithm

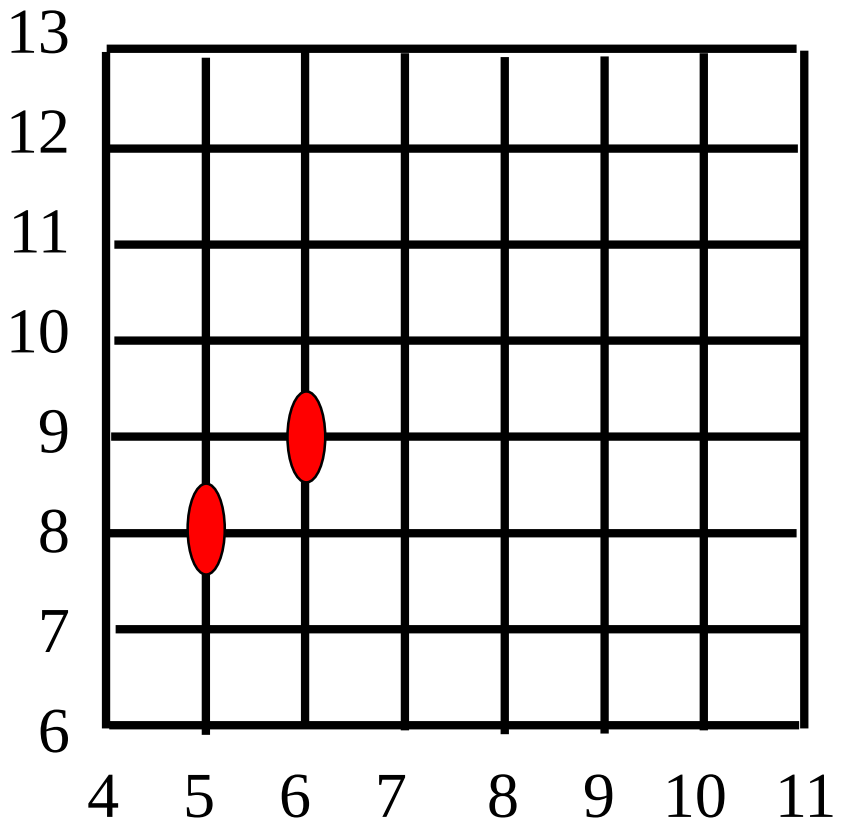
Bresenham's integer algorithm *for the first octant*

the line end points are (x_1, y_1) *and* (x_2, y_2), *assumed not equal*
all variables are assumed integer

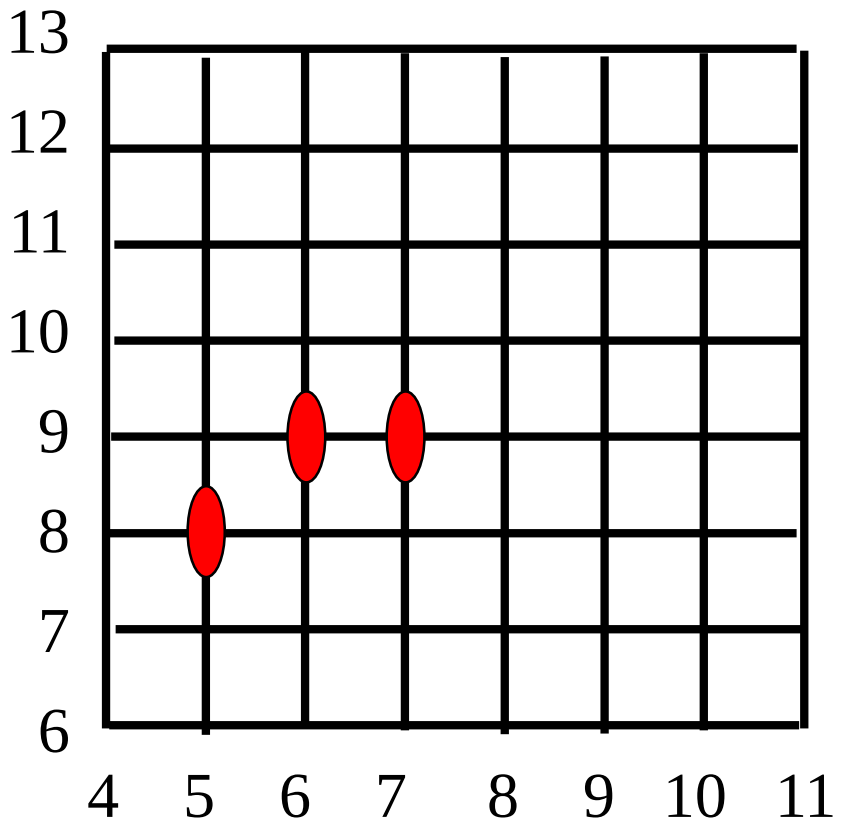
```
initialize variables
x = x1
y = y1
Δx = x2 − x1
Δy = y2 − y1
initialize ē to compensate for a nonzero intercept
ē = 2 * Δy − Δx
begin the main loop
for i = 1 to Δx
    setpixel (x, y)
    while (ē > 0)
        y = y + 1
        ē = ē − 2 * Δx
    end while
    x = x + 1
    ē = ē + 2 * Δy
next i
finish
```

- If $\Delta_y > \Delta_x$, then exchange values of Δ_y and Δ_x

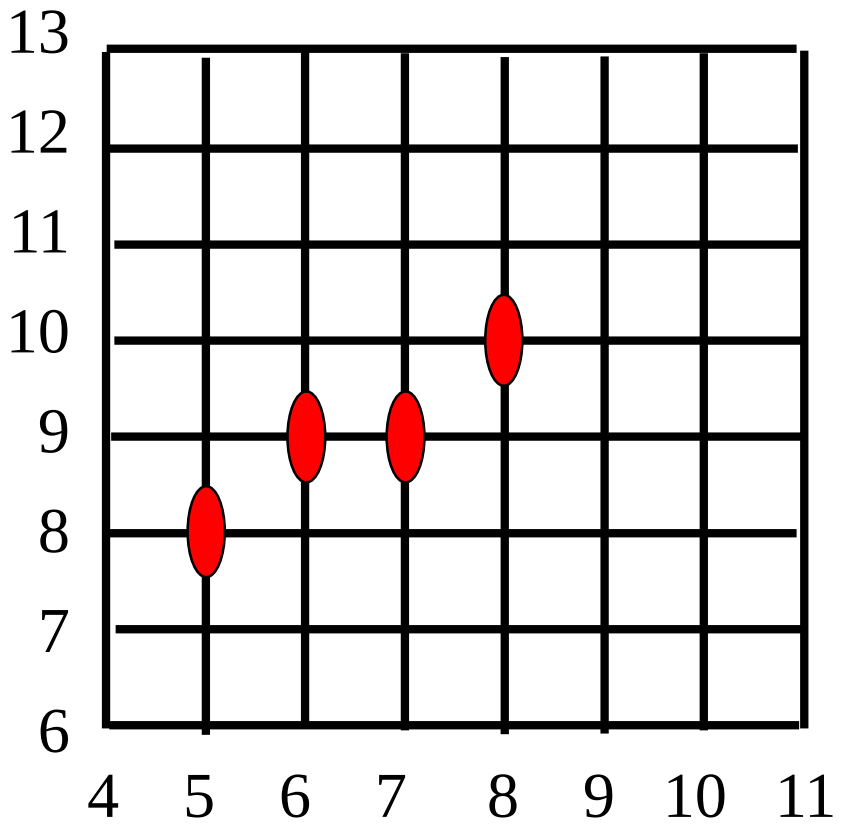
Graph



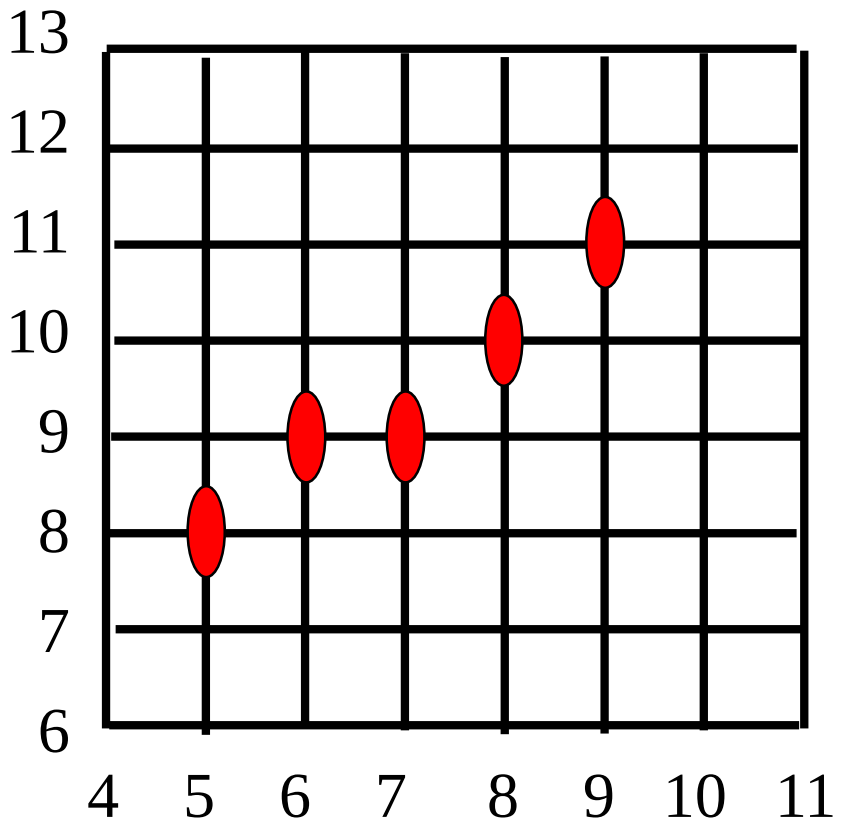
Continue the process...



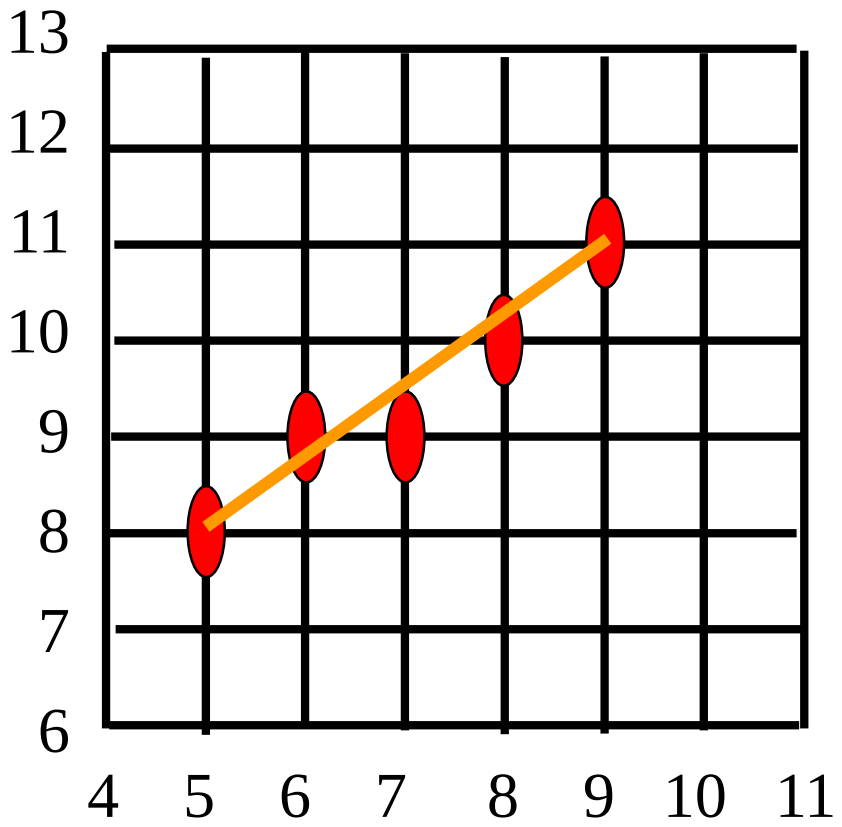
Graph



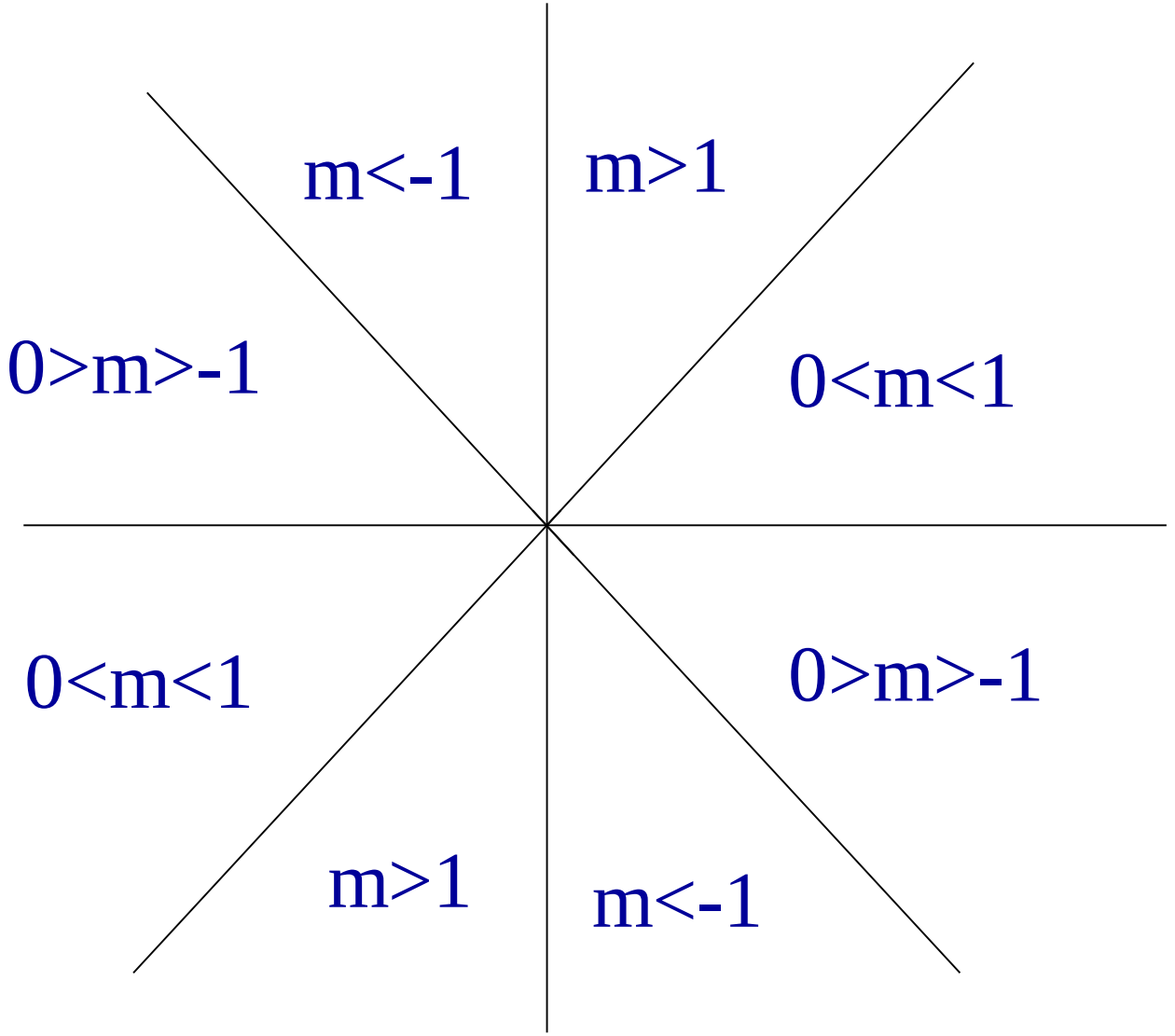
Graph



Graph



Line-drawing algorithm should work in every octant, and special cases



Generalized Integer Bresenham's Line Drawing Algorithm

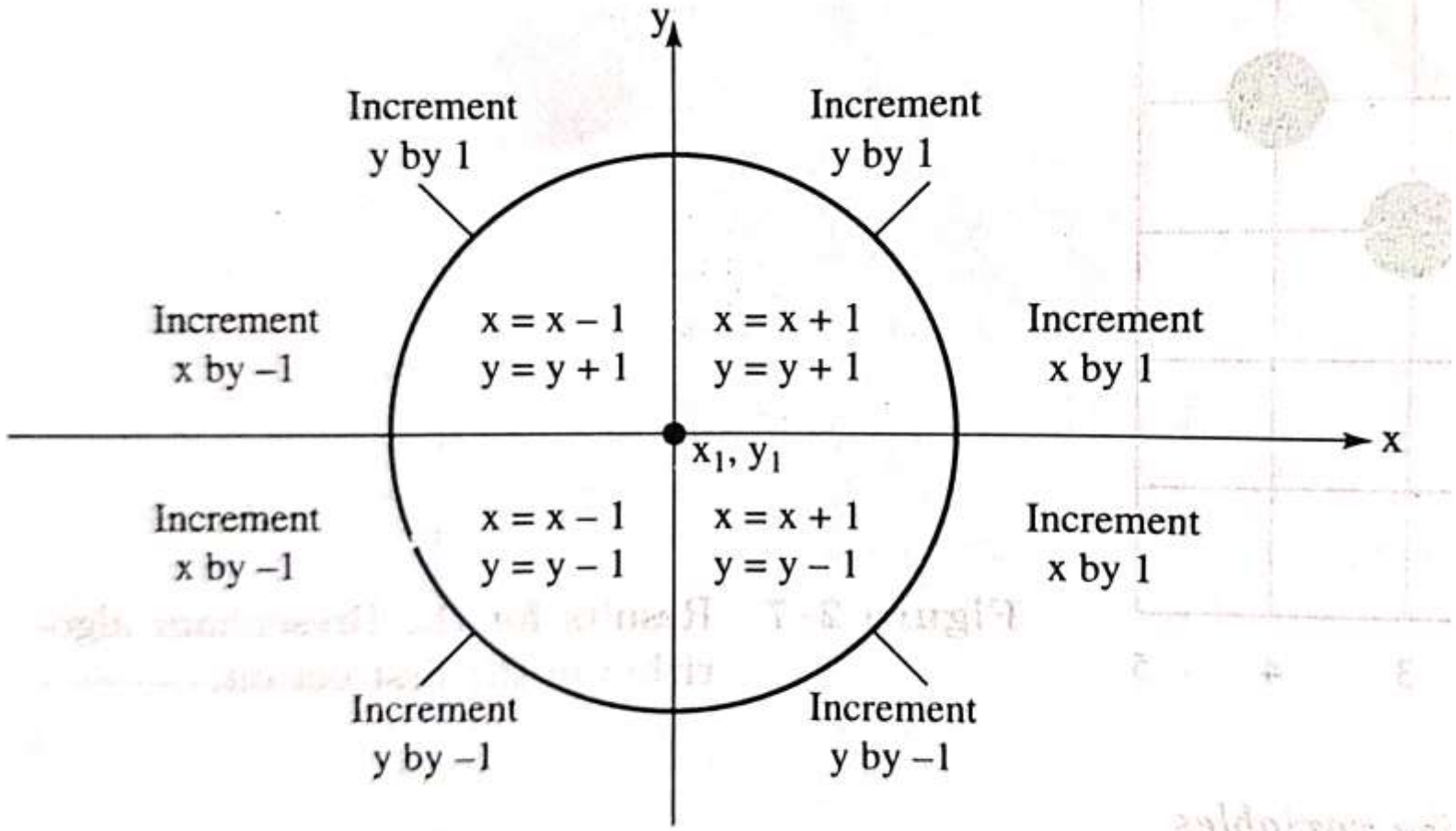


Figure 2–8 Conditions for general Bresenham's algorithm.

Generalized Integer Bresenham's Line Drawing Algorithm

generalized integer Bresenham's algorithm *for all quadrants*
the line end points are (x_1, y_1) *and* (x_2, y_2), *assumed not equal*
all variables are assumed integer
the **Sign** *function returns* $-1, 0, 1$ *as its argument is* < 0, $= 0$, *or* > 0

```
    initialize variables
    x = x1
    y = y1
    Δx = abs(x2 − x1)
    Δy = abs(y2 − y1)
    s1 = Sign(x2 − x1)
    s2 = Sign(y2 − y1)
    interchange Δx and Δy, depending on the slope of the line
    if Δy > Δx then
        Temp = Δx
        Δx = Δy
        Δy = Temp
        Interchange = 1
    else
        Interchange = 0
    end if
    initialize the error term to compensate for a nonzero intercept
    ē = 2 * Δy − Δx
    main loop
    for i = 1 to Δx
        setpixel (x, y)
```

Generalized Integer Bresenham's Line Drawing Algorithm

```
        while (ē > 0)
            if Interchange = 1 then
                x = x + s1
            else
                y = y + s2
            end if
            ē = ē − 2 * Δx
        end while
        if Interchange = 1 then
            y = y + s2
        else
            x = x + s1
        end if
        ē = ē + 2 * Δy
    next i
finish
```

Generalized Integer Bresenham's Line Drawing Algorithm

Example 2–4 Generalized Bresenham's algorithm

To illustrate the general Bresenham algorithm, consider the line from $(0, 0)$ to $(-8, -4)$. This line was previously considered in Ex. 2–2 using a simple DDA algorithm.

initial calculations

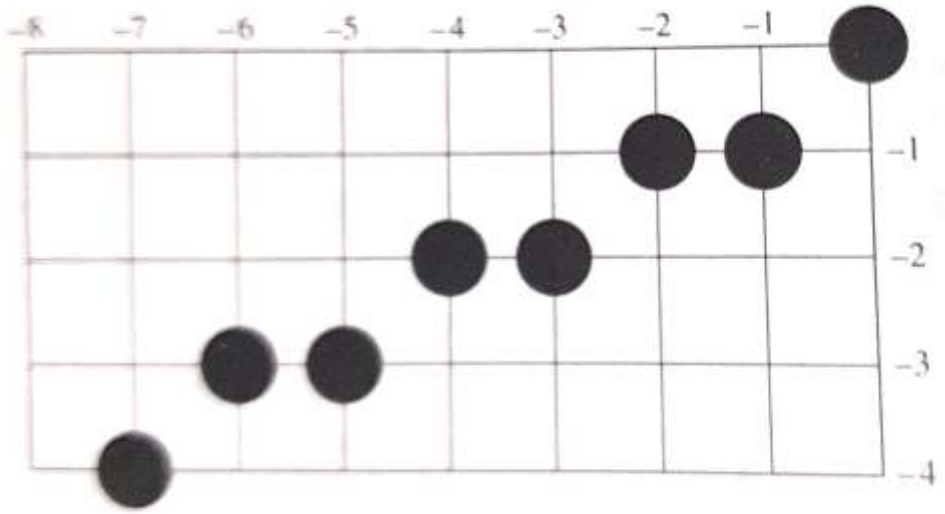
$x = 0$
$y = 0$
$\Delta x = 8$
$\Delta y = 4$
$s_1 = -1$
$s_2 = -1$
Interchange $= 0$
$\bar{e} = 0$

Incrementing through the main loop yields

i	setpixel	$\bar{e}$	x	y
		0	0	0
1	(0, 0)			
		8	−1	0
2	(−1, −1)			
		−8	−1	−1
		0	−2	−1
3	(−2, −1)			
		8	−3	−1
4	(−3, −1)			
		−8	−3	−2
		0	−4	−2

Generalized Integer Bresenham's Line Drawing Algorithm

i	setpixel	ē	x	y
5	(−4, −2)			
		8	−5	−2
6	(−5, −2)			
		−8	−5	−3
		0	−6	−3
7	(−6, −3)			
		8	−7	−3
8	(−7, −4)			
		−8	−7	−4
		0	−8	−4



ADVANTAGES

- Uses fixed points
- Easy to calculate (only addition & subtraction)
- Fast execution compare to DDA
- More accurate and efficient

DISADVANTAGES

- Drift away from actual line path
- Causes stair-case pattern

Bresenham's Circle Generation Algorithm

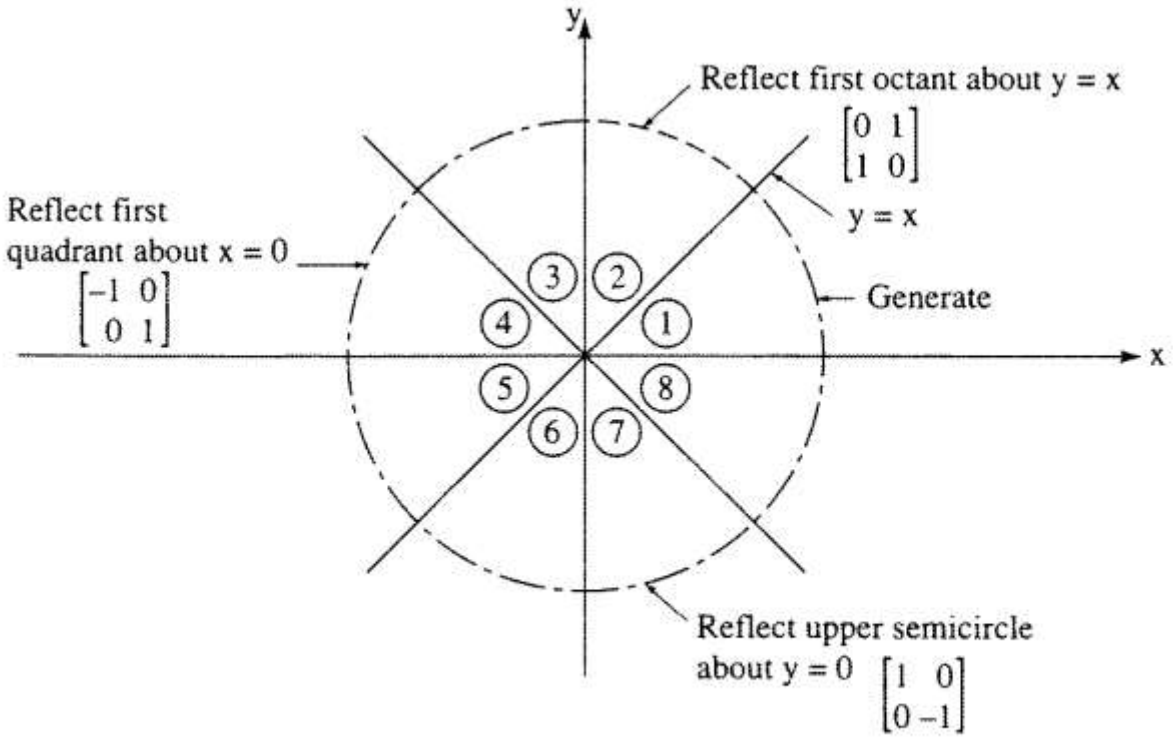


Figure 2–10 Generation of a complete circle from the first octant.

- One of the most efficient and easiest circle drawing algorithm.
- Only one octant of the circle need to be generated.
- Other parts are obtained by successive reflection.

Bresenham's Circle Generation Algorithm

To derive Bresenham's circle generation algorithm, consider 1st quadrant of an origin centered circle.

- If x = 0, y= R, then for clockwise generation of circle, y is monotonically decreasing function of x in the first quadrant.
- If y = 0, x= R, then for counter clockwise generation of circle, x is monotonically decreasing function of y in the first quadrant.

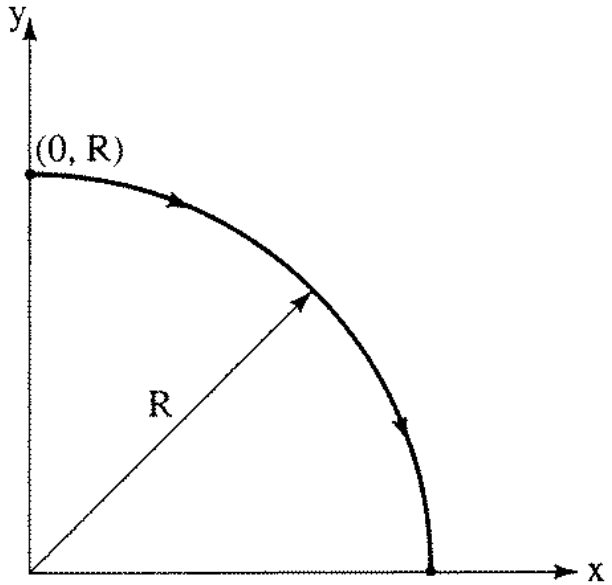


Figure 2–11 First quadrant of a circle.

Bresenham's Circle Generation Algorithm

- Assuming clockwise generation of circle, for any given point on the circle there are only 3 possible selections for next pixel
  - Horizontally to the right
  - Diagonally downward to the right
  - Vertically to the downward

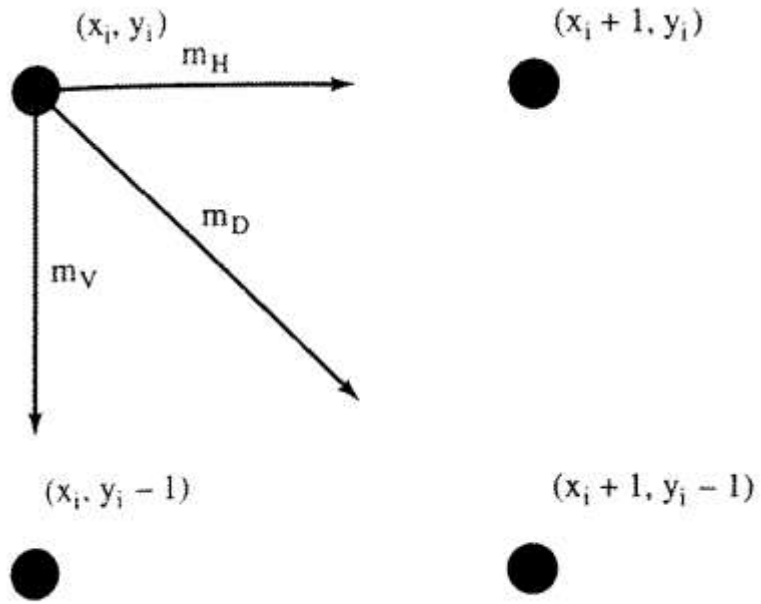


Figure 2–12 First quadrant pixel selections.

$$m_H = |(x_i + 1)^2 + (y_i)^2 - R^2|$$
$$m_D = |(x_i + 1)^2 + (y_i - 1)^2 - R^2|$$
$$m_V = |(x_i)^2 + (y_i - 1)^2 - R^2|$$

Bresenham's Circle Generation Algorithm

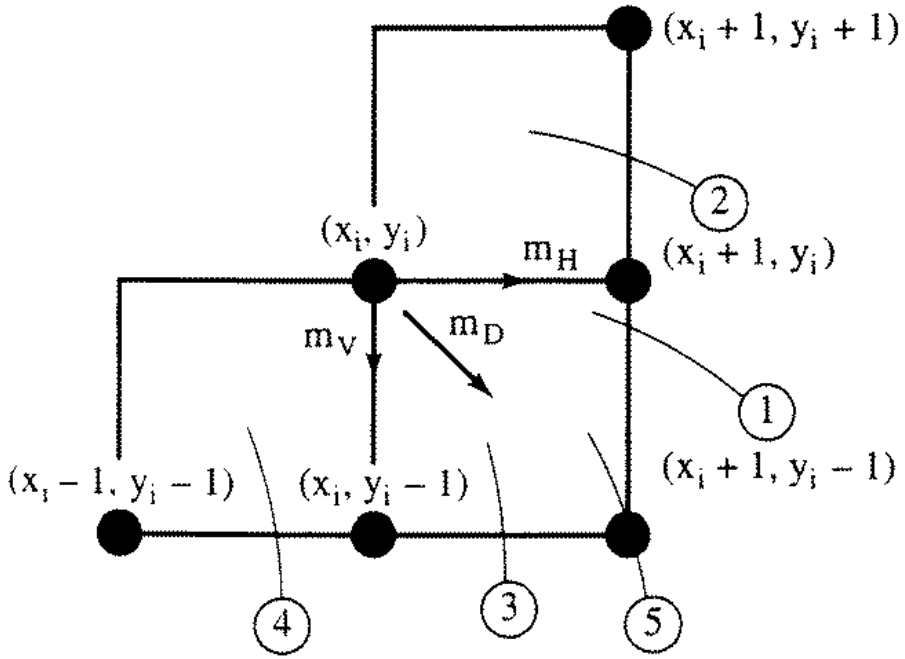


Figure 2–13 Intersection of a circle and the raster grid.

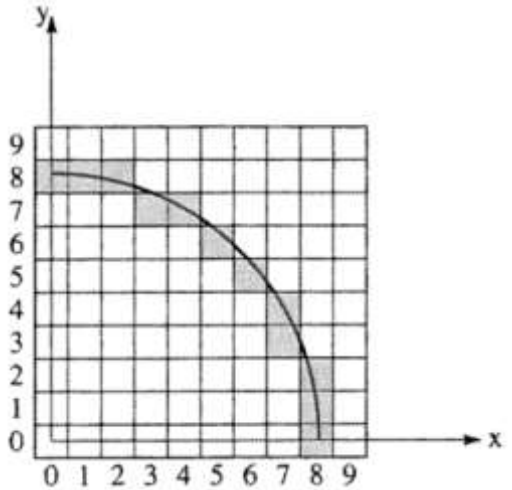
Bresenham's Circle Generation Algorithm

The difference between the square of the distance from the center of the circle to the diagonal pixel at $(x_i + 1, y_i - 1)$ and the distance to a point on the circle R^2 is

$$\Delta_i = (x_i + 1)^2 + (y_i - 1)^2 - R^2$$

If $\Delta_i < 0$, then the diagonal point $(x_i + 1, y_i - 1)$ is inside the actual circle, i.e., case 1 or 2 in Fig. 2–13. It is clear that either the pixel at $(x_i + 1, y_i)$, i.e., m_H, or that at $(x_i + 1, y_i - 1)$, i.e., m_D, must be chosen.

$$\delta = \left|(x_i + 1)^2 + (y_i)^2 - R^2\right| - \left|(x_i + 1)^2 + (y_i - 1)^2 - R^2\right|$$



If $\delta < 0$, then the distance from the actual circle to the diagonal pixel, m_D, is greater than that to the horizontal pixel, m_H. Conversely, if $\delta > 0$, then the distance to the horizontal pixel, m_H, is greater; thus, if

$\delta \leq 0$ choose m_H at $(x_i + 1, y_i)$

$\delta > 0$ choose m_D at $(x_i + 1, y_i - 1)$

Bresenham's Circle Generation Algorithm

- Thus the δ can be evaluated as:

$$\delta = (x_i + 1)^2 + (y_i)^2 - R^2 + (x_i + 1)^2 + (y_i - 1)^2 - R^2$$

Completing the square for the $(y_i)^2$ term by adding and subtracting $-2y_i + 1$ yields

$$\delta = 2[(x_i + 1)^2 + (y_i - 1)^2 - R^2] + 2y_i - 1$$

Using the definition for Δ_i gives

$$\delta = 2(\Delta_i + y_i) - 1$$

Bresenham's Circle Generation Algorithm

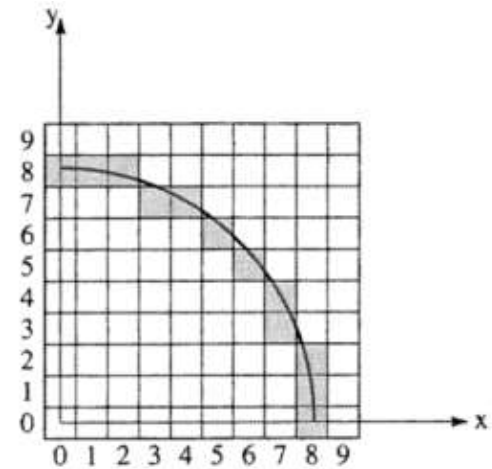
If $\Delta_i > 0$, then the diagonal point $(x_i + 1, y_i - 1)$ is outside the actual circle, i.e., case 3 or 4 in Fig. 2–13. Here, it is clear that either the pixel at $(x_i + 1, y_i - 1)$, i.e., m_D, or that at $(x_i, y_i - 1)$, i.e., m_V, must be chosen.

$$\delta' = |(x_i + 1)^2 + (y_i - 1)^2 - R^2| - |(x_i)^2 + (y_i - 1)^2 - R^2|$$

If $\delta' < 0$, then the distance from the actual circle to the vertical pixel at $(x_i, y_i - 1)$ is larger and the diagonal move m_D to the pixel at $(x_i + 1, y_i - 1)$ is chosen. Conversely, if $\delta' > 0$, then the distance from the actual circle to the diagonal pixel is greater, and the vertical move to the pixel at $(x_i, y_i - 1)$ is chosen. Thus, if

$\delta' \leq 0$ choose m_D at $(x_i + 1, y_i - 1)$

$\delta' > 0$ choose m_V at $(x_i, y_i - 1)$



Bresenham's Circle Generation Algorithm

- Thus the δ' can be evaluated as:

$$\delta' = (x_i + 1)^2 + (y_i - 1)^2 - R^2 + (x_i)^2 + (y_i - 1)^2 - R^2$$

Completing the square for the $(x_i)^2$ term by adding and subtracting $2x_i + 1$ yields

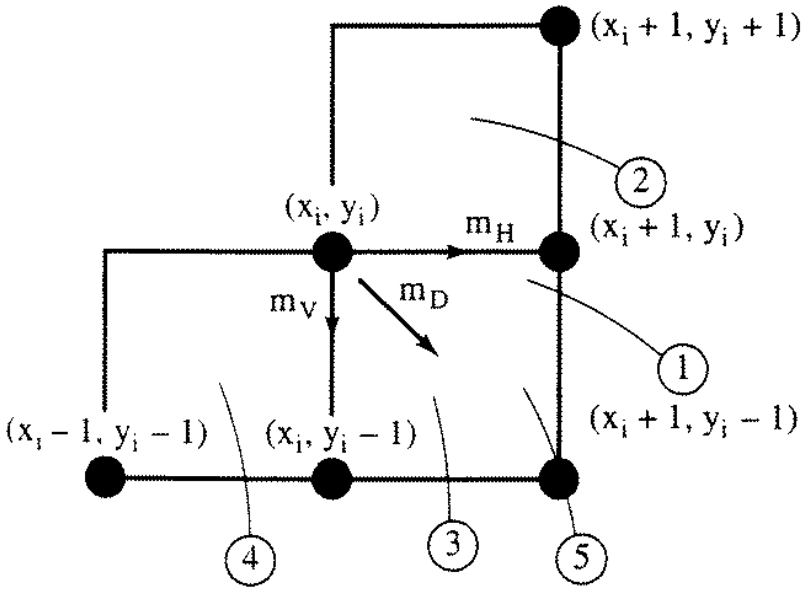
$$\delta' = 2\left[(x_i + 1)^2 + (y_i - 1)^2 - R^2\right] - 2x_i - 1$$

Using the definition of Δ_i then yields

$$\delta' = 2(\Delta_i - x_i) - 1$$

Bresenham's Circle Generation Algorithm

- Summarizing the results:



$$
\begin{array}{lll}
\Delta_i < 0 & & \\
\delta \leq 0 & \text{choose the pixel at } (x_i + 1, y_i) & \longrightarrow \quad m_H \\
\delta > 0 & \text{choose the pixel at } (x_i + 1, y_i - 1) & \longrightarrow \quad m_D \\
 & & \\
\Delta_i > 0 & & \\
\delta' \leq 0 & \text{choose the pixel at } (x_i + 1, y_i - 1) & \longrightarrow \quad m_D \\
\delta' > 0 & \text{choose the pixel at } (x_i, y_i - 1) & \longrightarrow \quad m_V \\
 & & \\
\Delta_i = 0 & \text{choose the pixel at } (x_i + 1, y_i - 1) & \longrightarrow \quad m_D
\end{array}
$$

Figure 2–13 Intersection of a circle and the raster grid.

Bresenham's Circle Generation Algorithm

Simple recursion relationships which yield an incremental implementation of the algorithm are easily developed. First, consider the horizontal movement, m_H, to the pixel at $(x_i + 1, y_i)$. Call this next pixel location $(i + 1)$. The coordinates of the new pixel and the value of Δ_i are then

$$
\begin{aligned}
x_{i+1} &= x_i + 1 \\
y_{i+1} &= y_i \\
\Delta_{i+1} &= (x_{i+1} + 1)^2 + (y_{i+1} - 1)^2 - R^2 \\
&= (x_{i+1})^2 + 2x_{i+1} + 1 + (y_i - 1)^2 - R^2 \\
&= (x_i + 1)^2 + (y_i - 1)^2 - R^2 + 2x_{i+1} + 1 \\
&= \Delta_i + 2x_{i+1} + 1
\end{aligned}
$$

Bresenham's Circle Generation Algorithm

Similarly, the coordinates of the new pixel and the value of Δ_i for the move m_D to $(x_i + 1, y_i - 1)$ are

$$x_{i+1} = x_i + 1$$

$$y_{i+1} = y_i - 1$$

$$\Delta_{i+1} = \Delta_i + 2x_{i+1} - 2y_{i+1} + 2$$

Those for the move m_V to $(x_i, y_i - 1)$ are

$$x_{i+1} = x_i$$

$$y_{i+1} = y_i - 1$$

$$\Delta_{i+1} = \Delta_i - 2y_{i+1} + 1$$

Bresenham's Circle Generation Algorithm

Bresenham's incremental circle algorithm *for the first quadrant*

all variables are assumed integer

```
initialize the variables
x_i = 0
y_i = R
Δ_i = 2(1 − R)
Limit = 0
while y_i ≥ Limit
    call setpixel (x_i, y_i)
    determine if case 1 or 2, 4 or 5, or 3
    if Δ_i < 0 then
        δ = 2Δ_i + 2y_i − 1
        determine whether case 1 or 2
        if δ ≤ 0 then
            call mh(x_i, y_i, Δ_i)
```

Bresenham's Circle Generation Algorithm

```
            else
                call md(x_i, y_i, Δ_i)
            end if
        else if Δ_i > 0 then
            δ' = 2Δ_i − 2x_i − 1
            determine whether case 4 or 5
            if δ' ≤ 0 then
                call md(x_i, y_i, Δ_i)
            else
                call mv(x_i, y_i, Δ_i)
            end if
        else if Δ_i = 0 then
            call md(x_i, y_i, Δ_i)
        end if
    end while
finish
```

Bresenham's Circle Generation Algorithm

move horizontally

subroutine $\mathrm{mh}(x_i, y_i, \Delta_i)$

$x_i = x_i + 1$

$\Delta_i = \Delta_i + 2x_i + 1$

end sub

move diagonally

subroutine $\mathrm{md}(x_i, y_i, \Delta_i)$

$x_i = x_i + 1$

$y_i = y_i - 1$

$\Delta_i = \Delta_i + 2x_i - 2y_i + 2$

end sub

move vertically

subroutine $\mathrm{mv}(x_i, y_i, \Delta_i)$

$y_i = y_i - 1$

$\Delta_i = \Delta_i - 2y_i + 1$

end sub

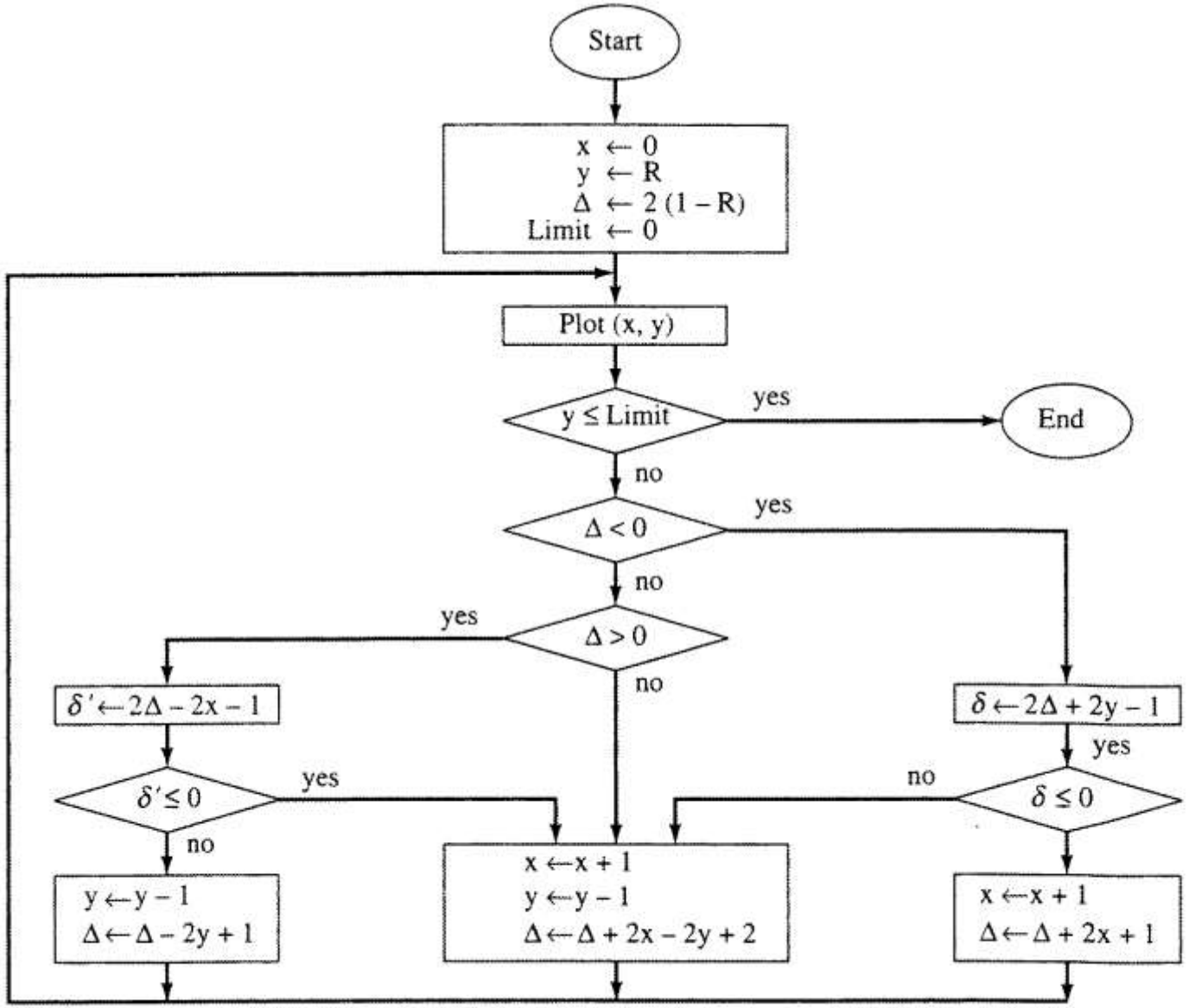


Figure 2–14 Flowchart for Bresenham's incremental circle algorithm in the first quadrant.

Bresenham's Circle Generation Algorithm - Example

Example 2–5 Bresenham's Circle Algorithm

To illustrate the circle generation algorithm, consider the origin-centered circle of radius 8. Only the first quadrant is generated.

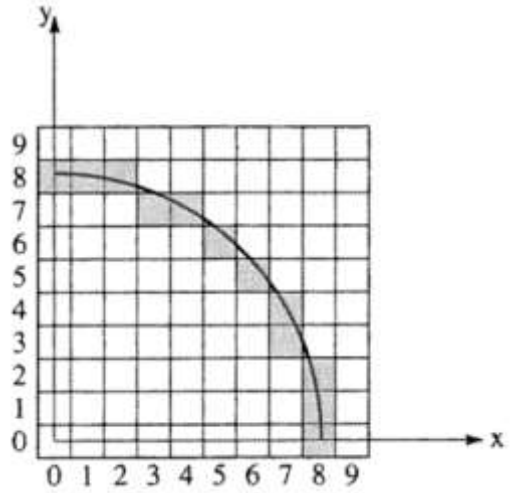
initial calculations

$x_i = 0$
$y_i = 8$
$\Delta_i = 2(1 - 8) = -14$
Limit $= 0$

Incrementing through the main loop yields

$y_i >$ Limit
continue
setpixel $(0, 8)$
$\Delta_i < 0$
$\delta = 2(-14) + 2(8) - 1 = -13$
$\delta < 0$
call mh(0,8,−14)
$x = 0 + 1 = 1$
$\Delta_i = -14 + 2(1) + 1 = -11$

$y_i >$ Limit
continue



Bresenham's Circle Generation Algorithm - Example

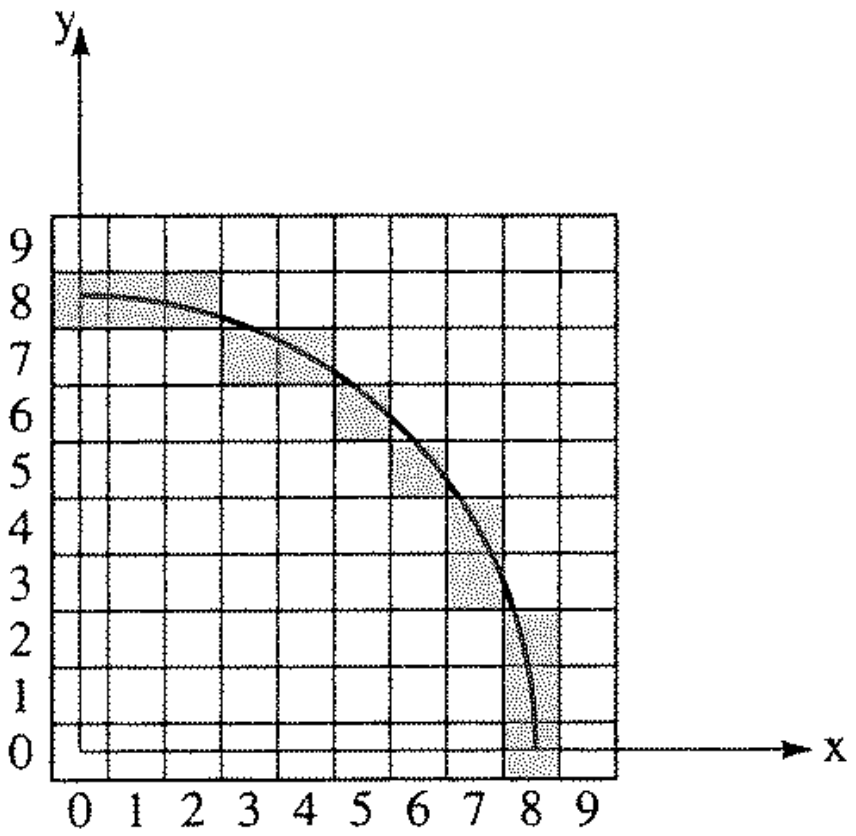
The details of each successive pass through the algorithm are summarized below. The list of pixels selected by the algorithm is $(0,8)$, $(1,8)$, $(2,8)$, $(3,7)$, $(4,7)$, $(5,6)$, $(6,5)$, $(7,4)$, $(7,3)$, $(8,2)$, $(8,1)$, $(8,0)$.

setpixel	Δ_i	δ	δ'	x	y
	-14			0	8
$(0,8)$					
	-11	-13		1	8
$(1,8)$					
	-6	-7		2	8
$(2,8)$					
	-12	3		3	7
$(3,7)$					
	-3	-11		4	7
$(4,7)$					
	-3	7		5	6
$(5,6)$					
	1	5		6	5

Bresenham's Circle Generation Algorithm - Example

$(6, 5)$				
	9	-11	7	4
$(7, 4)$				
	4	3	7	3
$(7, 3)$				
	18	-7	8	2
$(8, 2)$				
	17	19	8	1
$(8, 1)$				
	18	17	8	0
$(8, 0)$				
	complete			

Bresenham's Circle Generation Algorithm - Example



Circle radius is 8
Bright pixels

0	8
1	8
2	8
3	7
4	7
5	6
6	5
7	4
7	3
8	2
8	1
8	0

Figure 2–15 Results for Bresenham's incremental circle generation algorithm.

Scan Conversion

- In order to display the rasterized image using video techniques, it is necessary to organize the picture into precise pattern required by the graphics display. This process is called scan conversion.
- It is necessary that the information be organized and presented at appropriate frame rates in scan line order, i.e. from top to bottom and from left to right.
- Three ways:
  - **Real-time scan conversion**: most frequently used in vehicle simulation
  - **Run-length encoding**: generally used to 'store' images, or transmit images (or text)
  - **Frame buffer memory**: commonly used for image display in workstations and personal computers with graphics capability.

Run Length Encoding

- **Run length encoding** : large areas of the picture have the same intensity and colour.
- It specifies only the number of successive pixels on a given scan line with a given intensity.

Intensity	Run length

- Figure shows a simple monochrome (black and white) line drawing on a 30 × 30 raster and associated encoding for scan lines 1, 15 and 30
- Pixel-by-pixel storage requires 900 intensity values

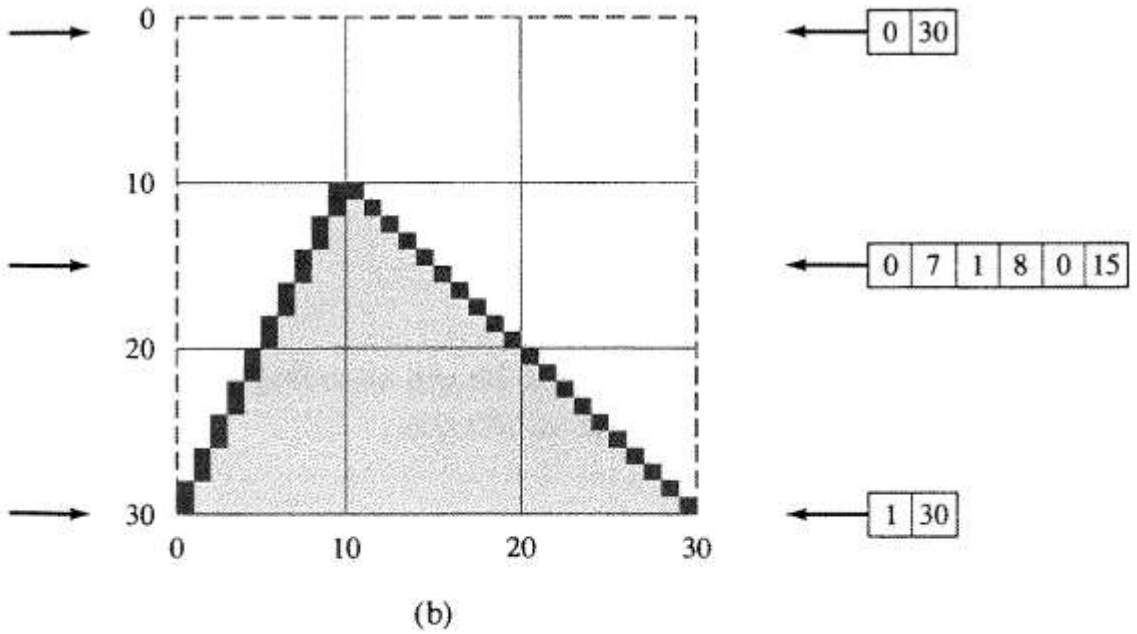


Figure 2–25 Run-length encoded image.

Run Length Encoding

- Solid figures are easily handled with run length encoding.
- It is easily extended to include colour. For colour, the intensity of each of the red, green and blue colour guns is given, preceded by the number of successive pixels for that pixel on a given scan line:

Run Length	Red Intensity	Green Intensity	Blue Intensity

- For a given colour display in which each individual colo gun is either completely off (0) or fully on (1), the encoding for scan line 15 of previous figure, with a yellow triangle on a blue background is:

15	0	0	1	7	1	1	0	8	0	0	1

Table 1–1 Simple 3-bit plane frame buffer color combinations

	Red	Green	Blue
Black	0	0	0
Red	1	0	0
Green	0	1	0
Blue	0	0	1
Yellow	1	1	0
Cyan	0	1	1
Magenta	1	0	1
White	1	1	1

Run Length Encoding

- **Disadvantages:**
  - Since the run lengths are stored sequentially, adding or deleting lines or text from the picture is difficult and time consuming.
  - There is overhead involved with both encoding and decoding the picture.
  - Storage requirement can approach twice that for pixel-by-pixel storage for short runs.

Frame Buffers

- A frame buffer is a large, continuous piece of computer memory.
- At a minimum, there is one memory bit for each pixel in the raster; this amount of memory is called a bit plane.
- A 1024 × 1024 element square requires 2^{20} or 1,048,576 memory bits in a single bit plane.
- The picture is built up in the frame buffer one bit at a time.
- Because memory bit has only two states (0 or 1), a single bit plane generates a black and white (monochrome) display.
- A DAC (digital-to-analog converter) is used to convert the bit value (0, 1) to analog signals for refreshing the screen

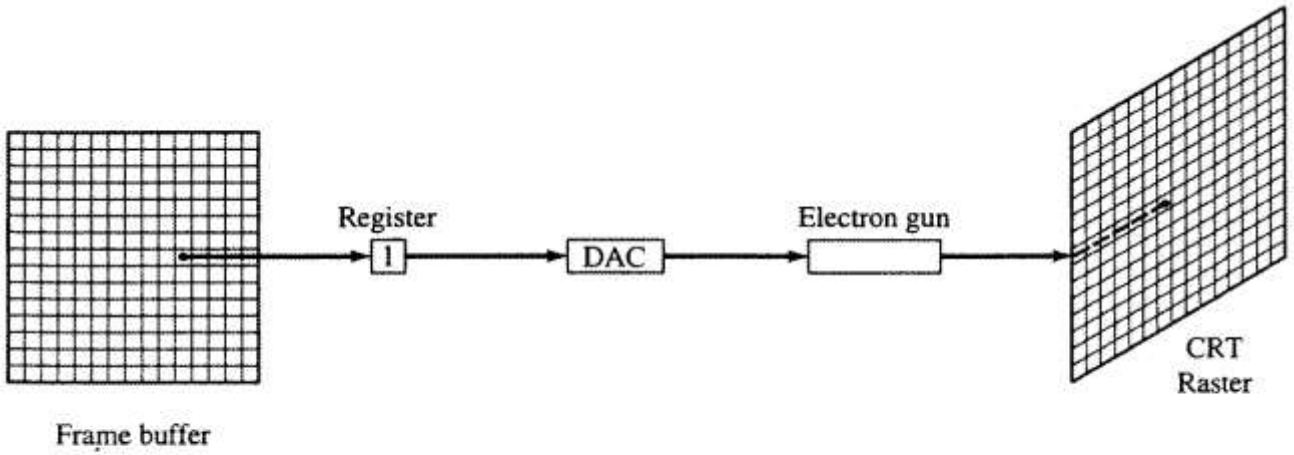
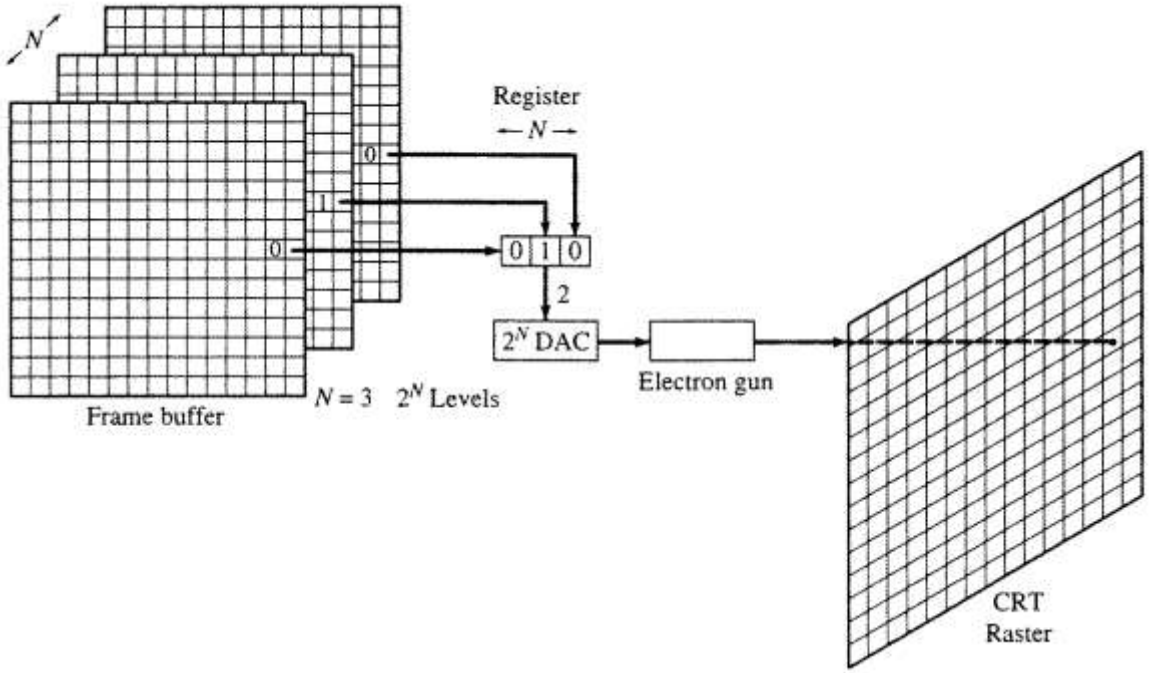


Figure 1–6 A single-bit-plane black-and-white frame buffer raster CRT graphics device.

Frame Buffers

- Color or gray levels can be achieved in the display using additional bit planes.
- The result for n bits per pixel (color depth=n) is a collection of n bit planes (2^n colors or gray shades at every pixel)
- The resulting binary number is interpreted as an intensity level between 0 (dark) and 2^N -1 (full intensity). Total 2^n intensity levels are possible.
- This converted into an analog voltage between 0 and maximum voltage of the gun by DAC.
- A 3-bit plane frame buffer for a 1024 × 1024 raster requires 3, 145,728 (3 × 1024 × 1024) memory bits.



Frame Buffers

- Using lookup table we can increase in the number of available intensity levels.
- Upon reading a bit-plane in the frame buffer, the resulting number is used as an index into the look table.
- Look table must contain 2^N entries.
- Each entry in the lookup table is W bits wide. W may greater than N.
- So, 2^W intensities are available.
- At a time 2^N intensities are available.

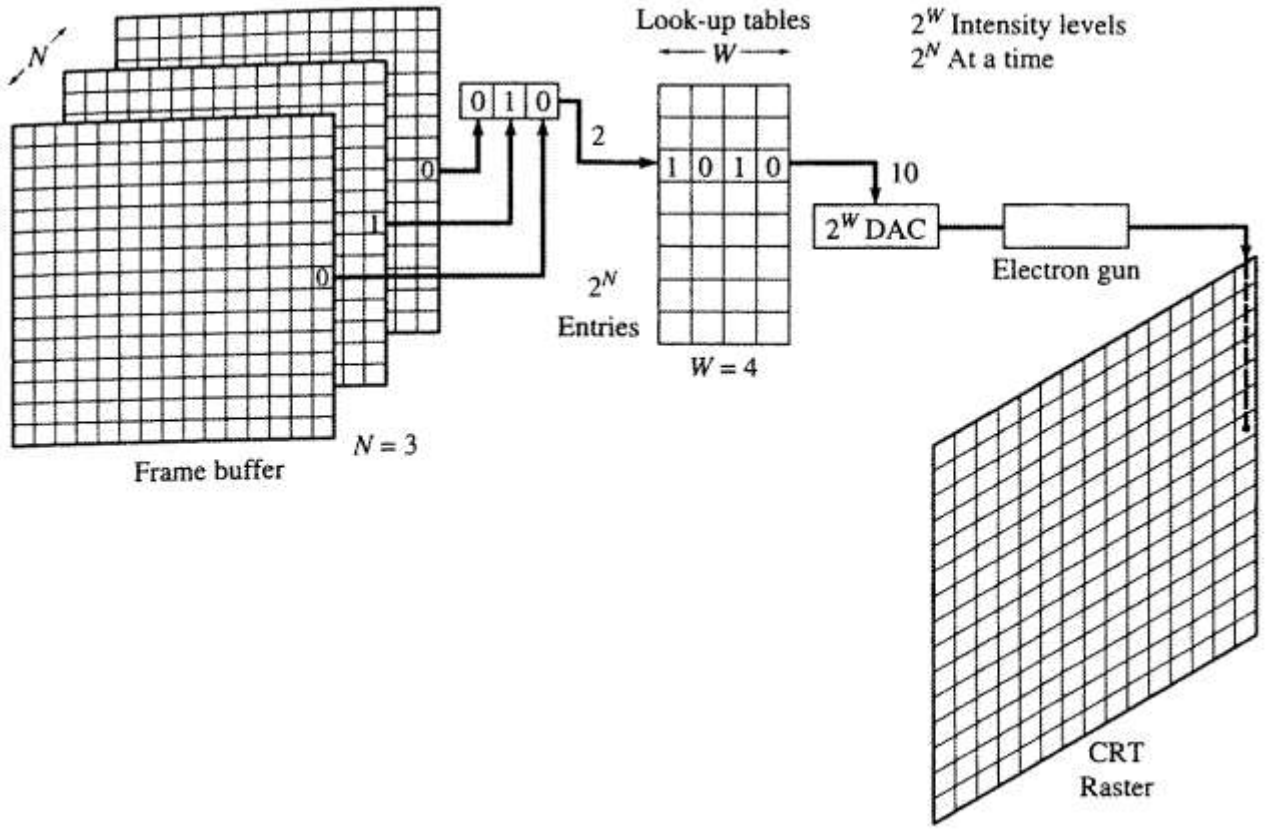


Figure 1–8 An N-bit-plane gray level frame buffer, with a W-bit-wide lookup table.

Frame Buffers

- Because there are 3 primary colours, a simple frame colour frame buffer is implemented with three bit planes, one for each primary colour.
- These 3 primaries (red, green and blue) are combined at the CRT to generate 8 colours. (Table below)

Table 1–1 Simple 3-bit plane frame buffer color combinations

	Red	Green	Blue
Black	0	0	0
Red	1	0	0
Green	0	1	0
Blue	0	0	1
Yellow	1	1	0
Cyan	0	1	1
Magenta	1	0	1
White	1	1	1

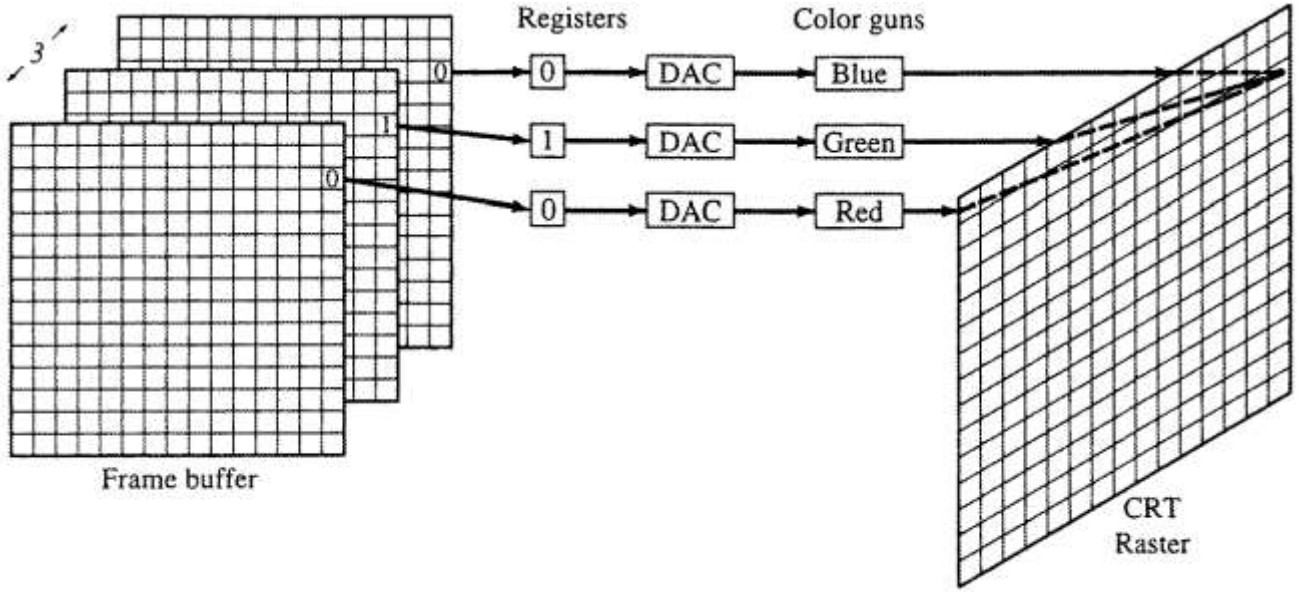


Figure 1–9 Simple color frame buffer.

Frame Buffers Scan Conversion : Video Basics

- Scanning pattern and frequency of repetition are based on human visual perception and electronic principle.
- If individual images are presented at a rate greater than the time required for the visual system to resolve individual images, one image persists while the next is being presented.
- The minimum picture presentation or update rate is 25 frames/second.
- Minimum refresh or repetition rate is twice i.e. 50 frames/second.
- In movies film picture presentation rate is 24 frames/second wile update rate is 48 frames/second.
- The same effect is achieved in video using a technique called **interlacing.**
- The American standard video system uses 525 horizontal lines with a frame or viewing aspect ratio of 4:3 (700 vertical lines).
- The repetition or frame rate is 30 frames/second.

Frame Buffers Scan Conversion : Video Basics

- Each frame is divided into two fields , interlaced or interwoven with each other.
- One field contains all odd numbered scan lines and the other contains even numbered scan lines.
- Scanning pattern:
  - Left to right and top to bottom, all odd lines first.
  - **Horizontal retrace:** requires 17% time allowed for one scan line.
  - **Vertical retrace:** 262 ½ lines scanned, requires time equivalent for 21 lines.
  - Repeat same for even scan lines
  - Thus 60 fields per second.

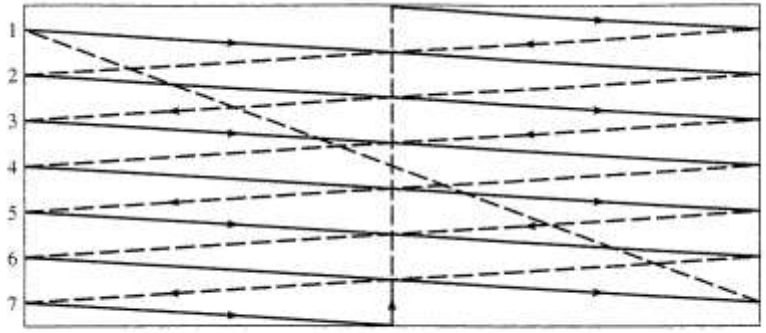


Figure 1–16 Schematic of a seven-line interlaced scan line pattern. The odd field begins with line 1. The horizontal retrace is shown dashed.

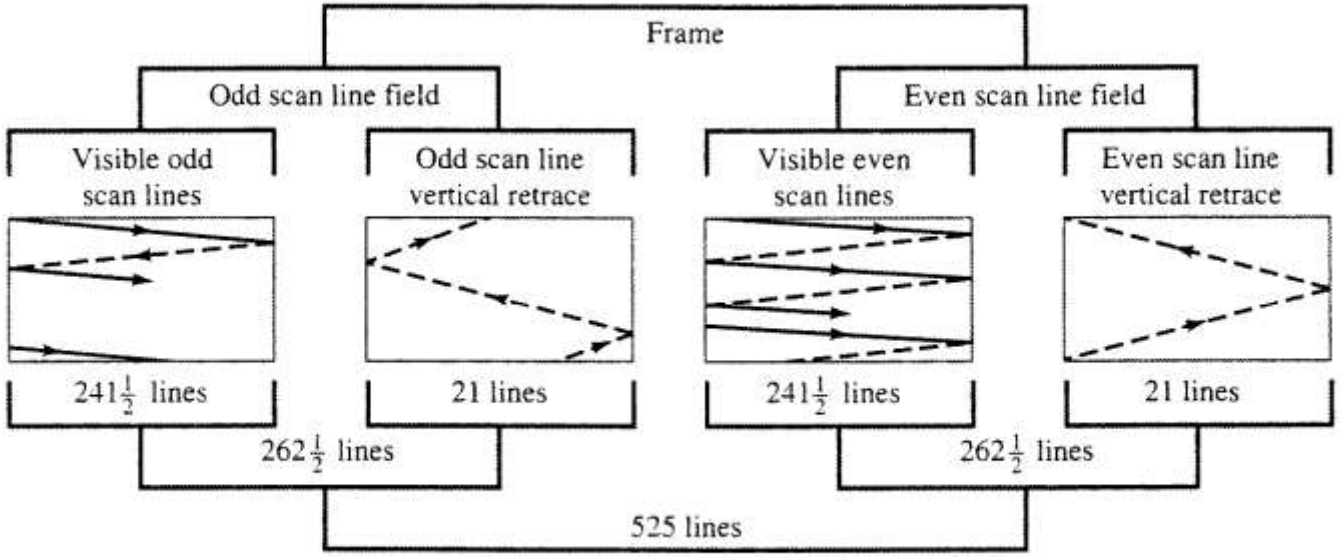
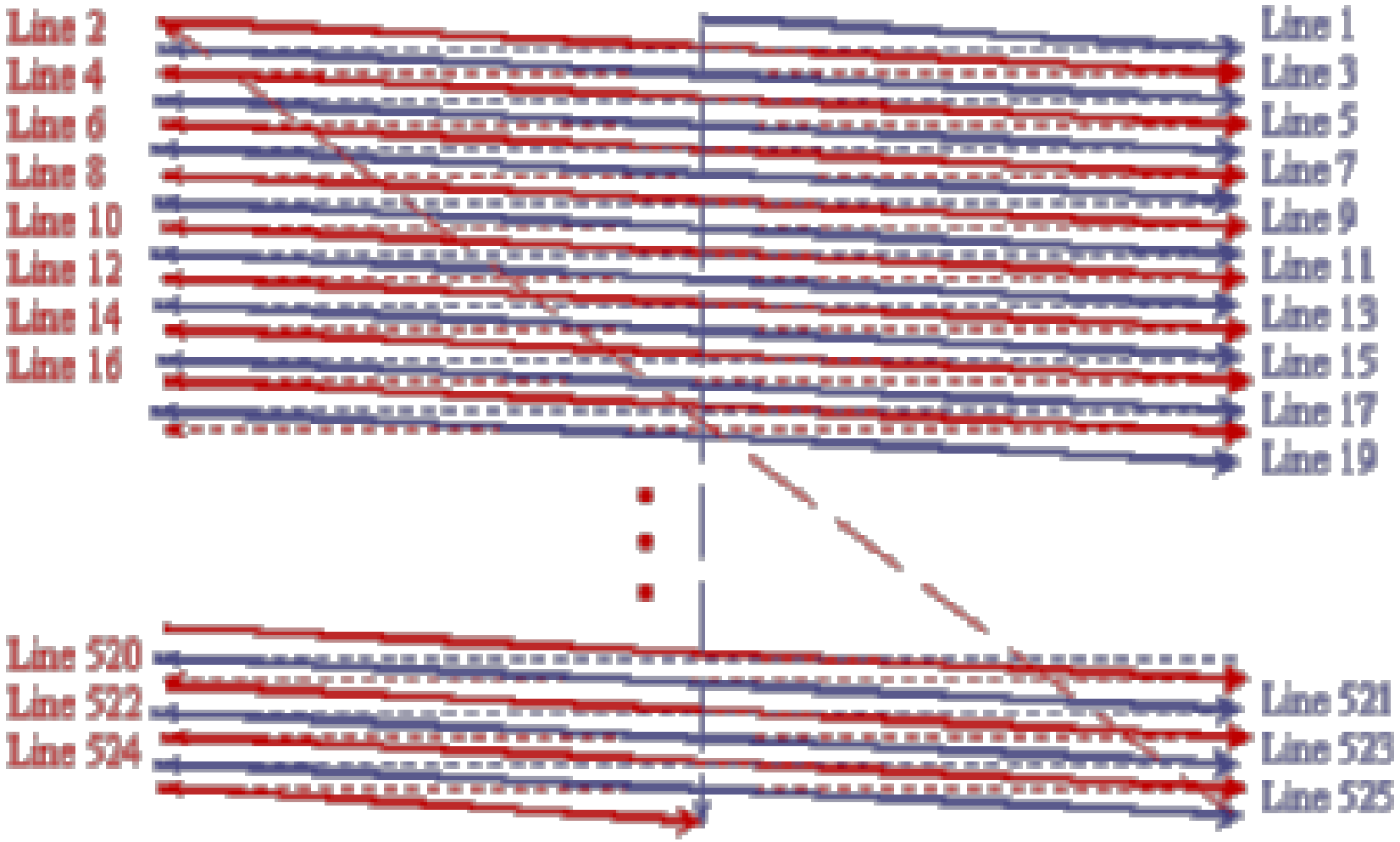


Figure 1–17 A 525-line standard frame schematic.

Frame Buffers Scan Conversion : Video Basics



Frame Buffers Scan Conversion : Video Basics

- Out of 525 only 483 lines actually visible due to vertical retrace for each field.
- During this time the electron beam is made invisible.
- The time available for each scan line is calculated for a frame repetition rate of 30 as:

  1/30 sec./frame x 1/525 frame/scan lines = 63 ½ microseconds/scan line
- Approximately 10 ½ microseconds required for horizontal retrace so, time available for each scan line is 53 microseconds.
- With video aspect ratio of 4:3 there are 644 pixels on each scan line. The time available to access and display a pixel is thus:

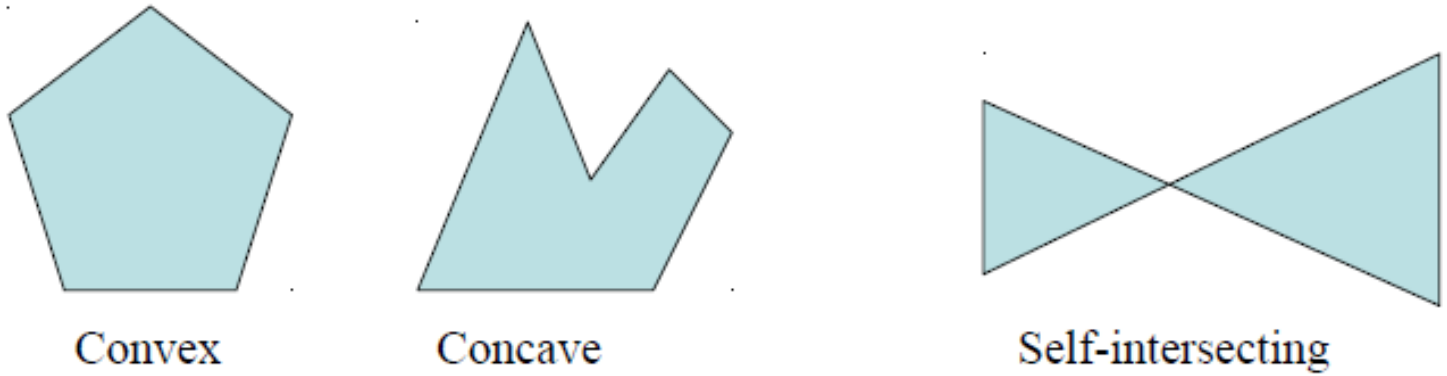
  53 microseconds/ scan line x 1/644 scan line/pixels = 82 nanoseconds

Polygon surface

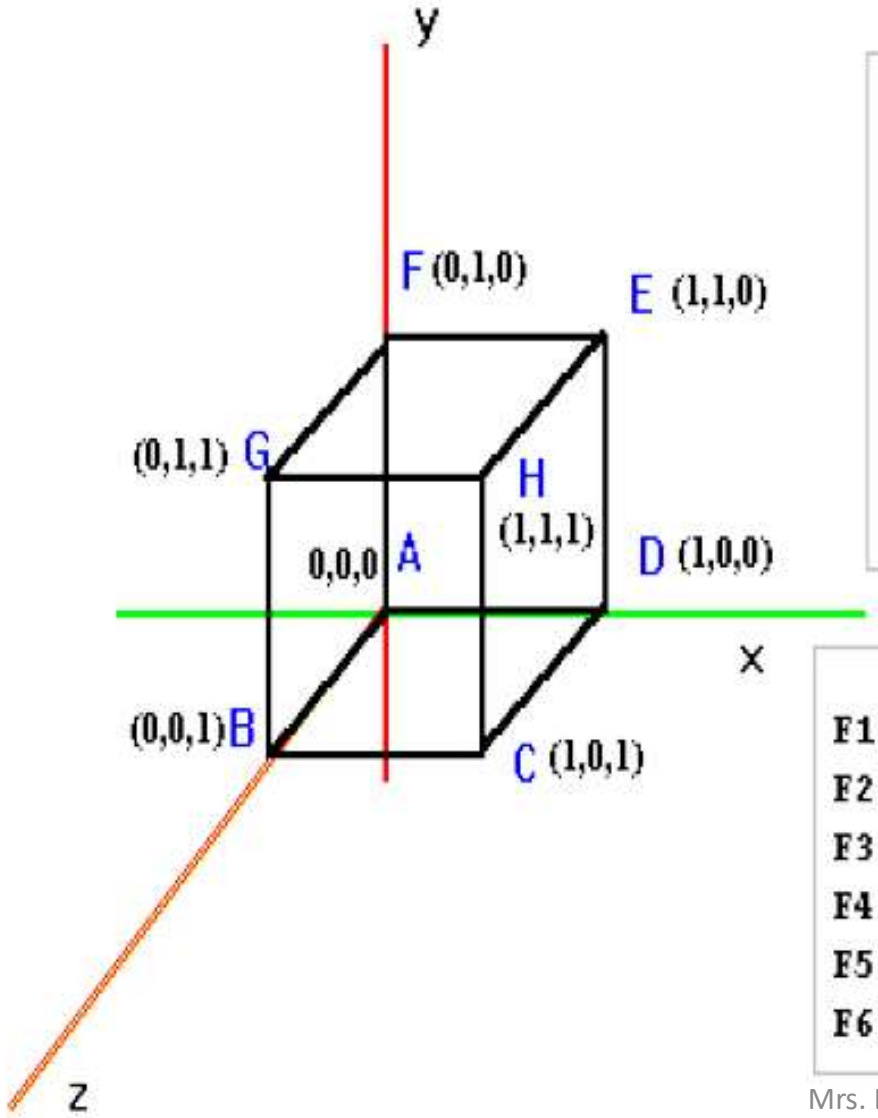
- A polygon is an important *graphics primitive.*
- A polygon is a *closed area* of image bounded by straight or curved lines and filled with one solid colour.
- Since images are two dimensional, a polygon is a closed planar figure.
- A polygon can be defined as an image which consists of a finite ordered set of straight boundaries called *edges*.
- The polygon can also be defined by an ordered sequence of *vertices*, i.e, the corners of the polygon.
- The edges of the polygon are then obtained by traversing the vertices in the given order;
- Two consecutive vertices define one edge.
- The polygon can be closed by connecting the last vertex to the first.

Polygon surface

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



Polygon surface



Vertex List

A	= V1 =	0	0	0
B	= V2 =	0	0	1
C	= V3 =	1	0	1
D	= V4 =	1	0	1
E	= V5 =	1	1	0
F	= V6 =	0	1	0
G	= V7 =	0	1	1
H	= V8 =	1	1	1

Edge List

	line	vertex
E1	AB	V1, V2
E2	BC	V2, V3
E3	CD	V3, V4
E4	DA	V4, V1
E5	DE	V4, V5
E6	EH	V5, V8
E7	HC	V8, V3
E8	EF	V5, V6
E9	FG	V6, V7
E10	GH	V7, V8
E11	GB	V7, V2
E12	FA	V6, V1

Face List

	edges
F1	E1, E2, E3, E4
F2	E3, E5, E6, E7
F3	E8, E9, E10, E6
F4	E9, E11, E1, E12
F5	E2, E7, E10, E11
F6	E4, E5, E8, E12

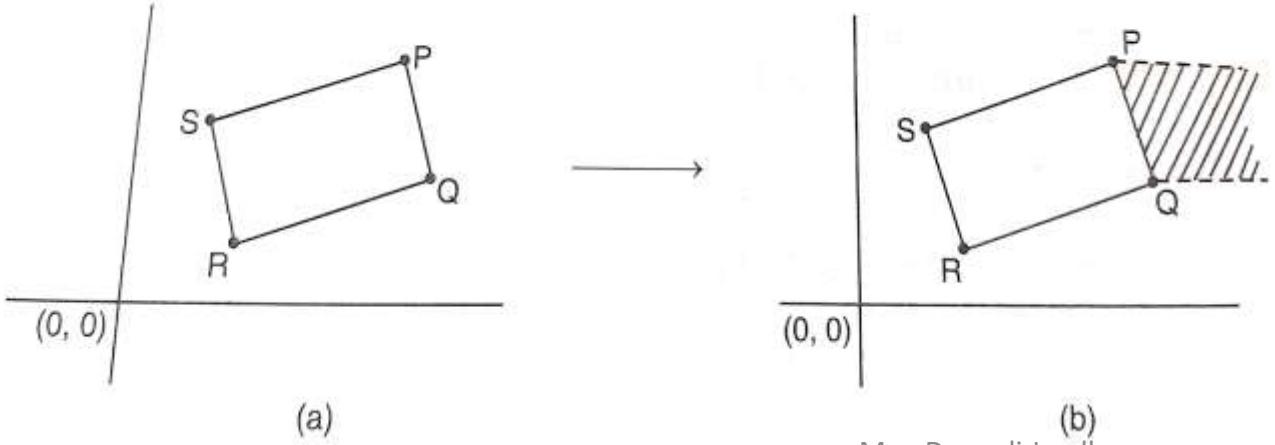
Scan Converting Polygon-Edge Fill Algorithm

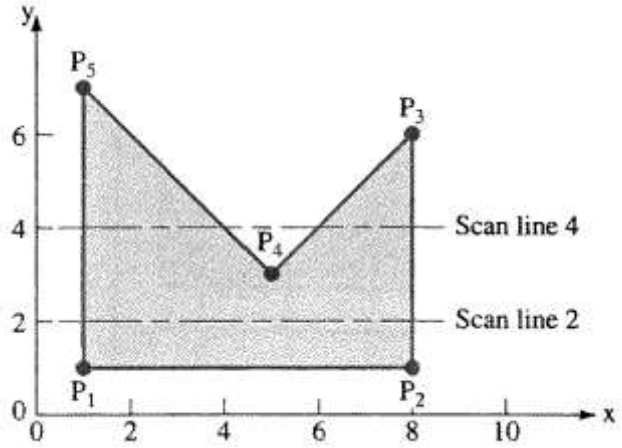
Edge fill algorithm

For each scan line intersecting a polygon edge at (x_1, y_1) complement all pixels whose midpoints lie to the right of (x_1, y_1), i.e., for (x, y_1), $x + 1/2 > x_1$.

- Means if the pixel is having back ground colour then we have to make the colour of the pixel as fill colour and it earlier it is fill colour then make it as a background colour.
- The algorithm is applied to each polygon edge individually
- The order in which edges are considered is not important.

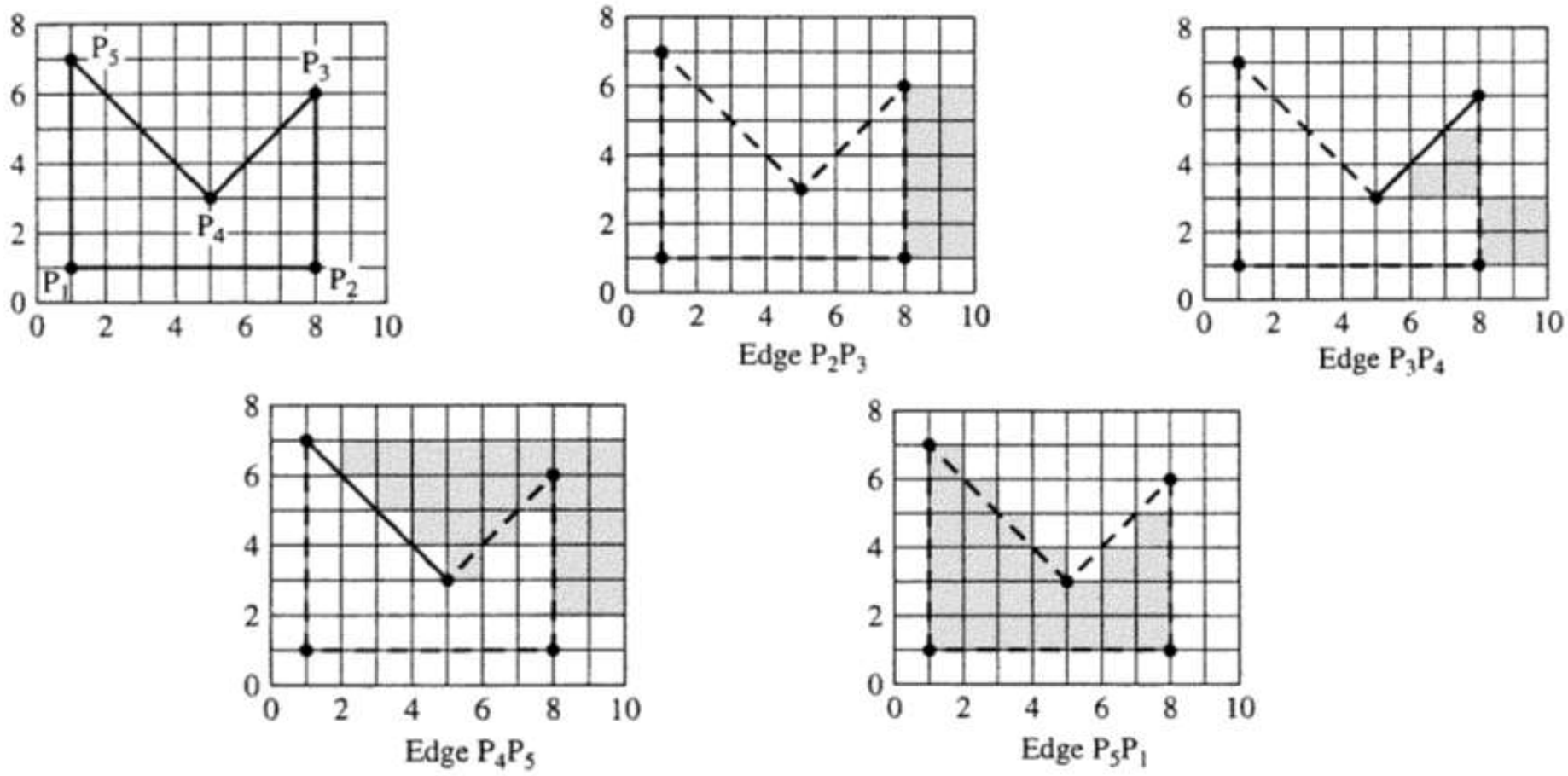
For example, consider a polygon PQRS as shown in Fig. 4.10 which is to be filled.





Scan Converting Polygon-Edge Fill Algorithm

- Pixels that are half inside and half outside the polygon : the edge fill algorithm activates them only if the inside of polygon lies to the left of the center of the pixel.



Edge fill algorithm

Scan Converting Polygon-Edge Fill Algorithm

- Disadvantages:
  - For complex pictures each individual pixel is addressed many times, so requires more time.
  - Hence algorithm is limited by input/output considerations.
  - We complement all pixels whose mid point lie on the RHS of edge. This means that even unwanted pixels are being visited.

Scan Converting Polygon-Fence Fill Algorithm

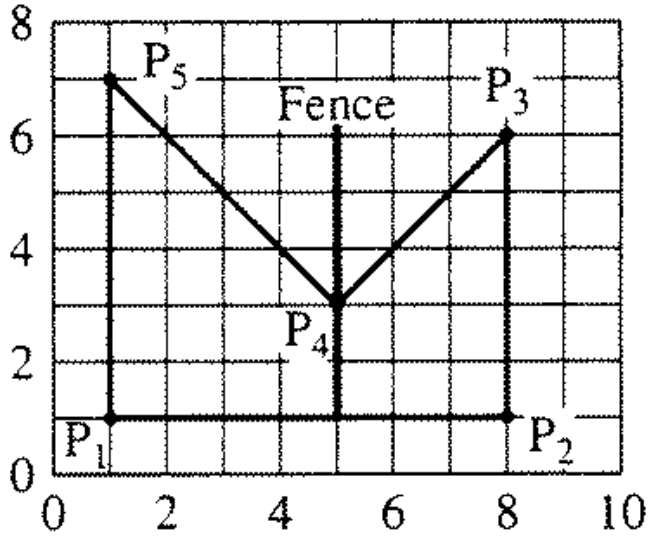
- The number of pixels addressed by the edge fill algorithm is reduced by a fence fill algorithm.

The fence fill algorithm:

For each scan line intersecting a polygon edge:

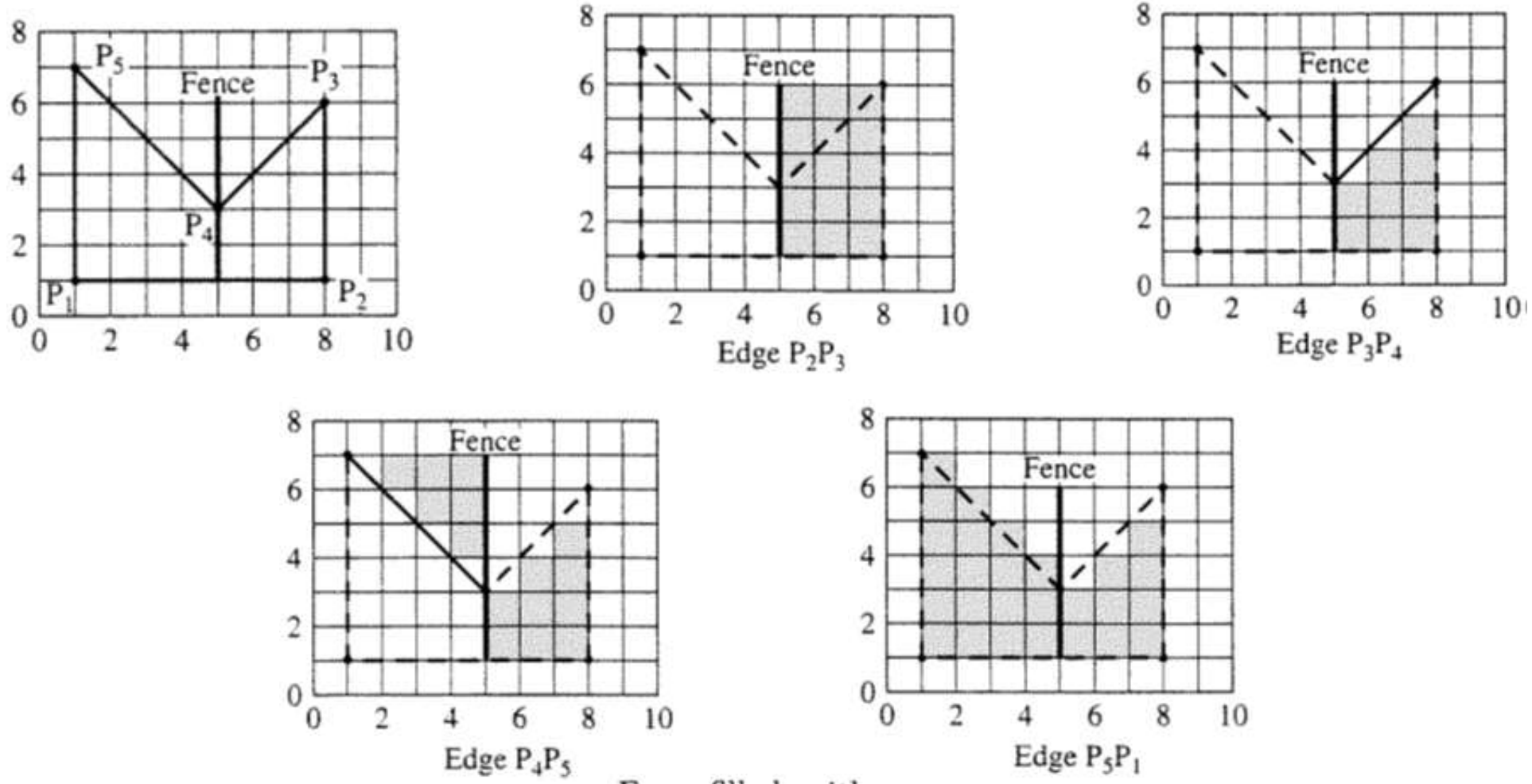
If the intersection is to the left of the fence, complement all pixels having a midpoint to the right of the intersection of the scan line and the edge, and to the left of the fence.

If the intersection is to the right of the fence, complement all pixels having a midpoint to the left of or on the intersection of the scan line and the edge, and to the right of the fence.



Scan Converting Polygon-Fence Fill Algorithm

- Pixels that are half inside and half outside the polygon : the edge fill algorithm activates them only if the inside of polygon lies to the right of the center of the pixel.



Fence fill algorithm.

Scan Converting Polygon-Edge Flag Algorithm

- Disadvantages of both edge fill and fence fill algorithm is the number of pixels addressed more than once.
- This advantage is eliminated by a modification called edge flag algorithm.
- Two steps:
  1. Outline the contour.
  2. Fill between bounding pixels.

The edge flag algorithm:

Contour outline:

Using the half scan line convention for each edge intersecting the scan line, set the leftmost pixel whose midpoint lies to the right of the intersection, i.e., for $x + 1/2 > x_{intersection}$, to the boundary value.

Fill:

```
For each scan line intersecting the polygon
    Inside = FALSE
    for x = 0 (left) to x = xmax (right)
        if the pixel at x is set to the boundary value then
            negate Inside
        end if
        if Inside = TRUE then
            set the pixel at x to the polygon value
        else
            reset the pixel at x to the background value
        end if
    next x
```

Scan Converting Polygon-Edge Flag Algorithm

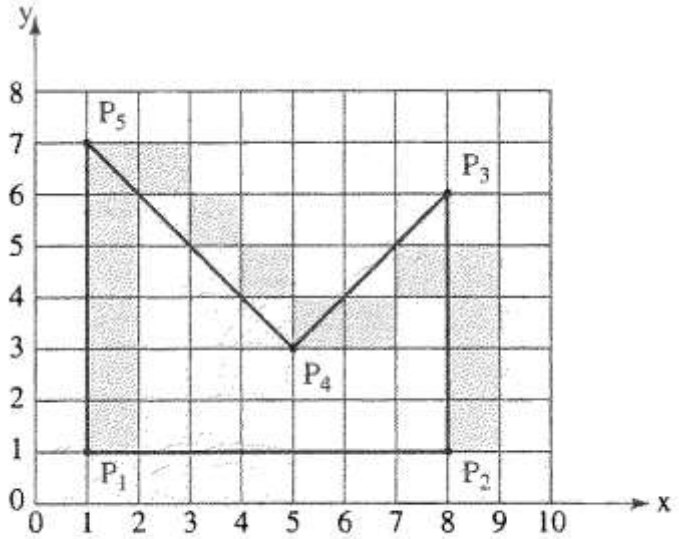
Consider the application of the edge flag algorithm to the example polygon of Fig. 2–34. First the contour is outlined. The result is shown in Fig. 2–42a. Pixels at $(1,1)$, $(1,2)$, $(1,3)$, $(1,4)$, $(1,5)$, $(1,6)$, $(2,6)$, $(3,5)$, $(4,4)$, $(5,3)$, $(6,3)$, $(7,4)$, $(8,4)$, $(8,3)$, $(8,2)$, $(8,1)$ are activated.

The polygon is then filled. To illustrate this the scan line at 3 is extracted and shown in Fig. 2–47b. Pixels at $x = 1$, 5, 6, and 8 on this scan line are activated to outline the contour. Applying the fill algorithm yields

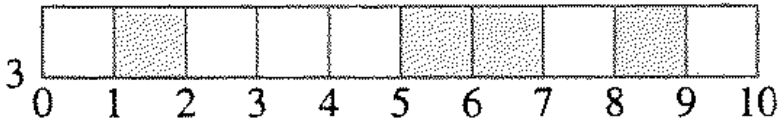
Initially

Inside = FALSE

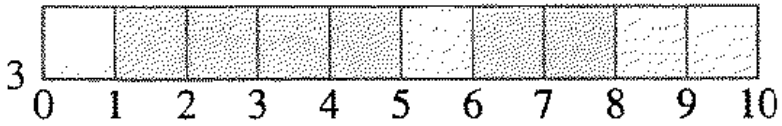
For $x = 0$	The pixel is not set to the boundary value and Inside = FALSE. Therefore, no action is taken.



(a)



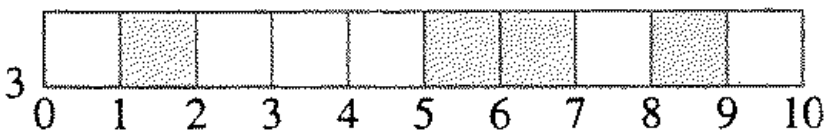
(b)



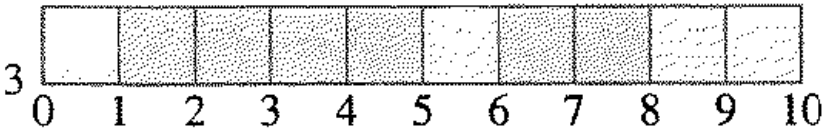
(c)

Scan Converting Polygon-Edge Flag Algorithm

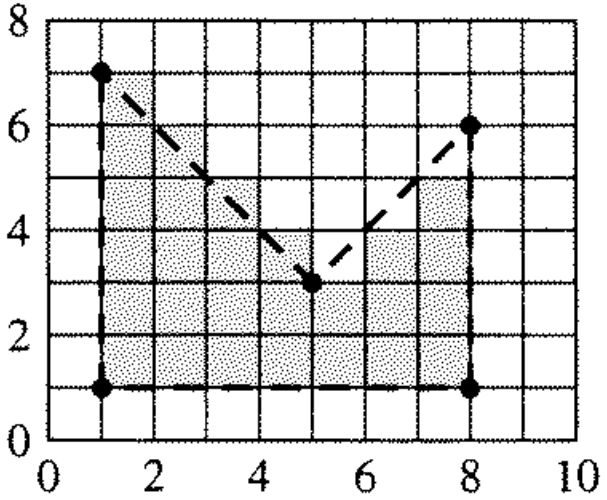
For $x = 1$	The pixel is set to the boundary value, Inside is negated to TRUE. Inside = TRUE, so the pixel is set to the polygon value.
For $x = 2,\ 3,\ 4$	The pixel is not set to the boundary value. Inside = TRUE, so the pixel is set to the polygon value.
For $x = 5$	The pixel is set to the boundary value, Inside is negated to FALSE. Inside = FALSE, so the pixel is set to the background value.
For $x = 6$	The pixel is set to the boundary value, Inside is negated to TRUE. Inside = TRUE, so the pixel is set to the polygon value.
For $x = 7$	The pixel is not set to the boundary value. Inside = TRUE, so the pixel is set to the polygon value.
For $x = 8$	The pixel is set to the boundary value, Inside is negated to FALSE. Inside = FALSE, so the pixel is set to the background.



(b)



(c)



The result is shown in Fig. 2–47c. The final result for the complete polygon is the same as for the edge fill algorithm as it is shown in Fig. 2–45.

Scan Converting Polygon-Edge Flag Algorithm

- EX:

Scan Converting Polygon- Seed Fill Algorithm

- Edge fill algorithm fill the polygon in scan line order.
- The seed fill algorithm assumes that at least one pixel interior to polygon or region is known.
- Algorithm attempts to find and fill all other pixels interior to the region.
- Two Methods:
  - Interior
  - Boundary defined

Scan Converting Polygon- Seed Fill Algorithm

Interior-defined region

- All pixels in the interior of the region are one colour or value and all pixels exterior to the region are another colour.
- Algorithm that fill interior-defined regions are known as flood fill algorithms.
- It is time Consuming.

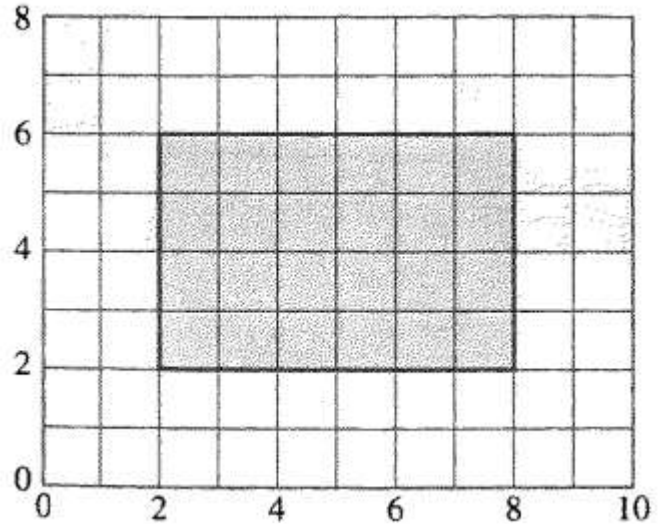


Figure 2–48 Interior-defined region.

Boundary-defined region

- All the pixels on the region boundary are a unique value or colour.
- Algorithm that fill boundary-defined regions are known as boundary fill algorithms.
- It is less time Consuming

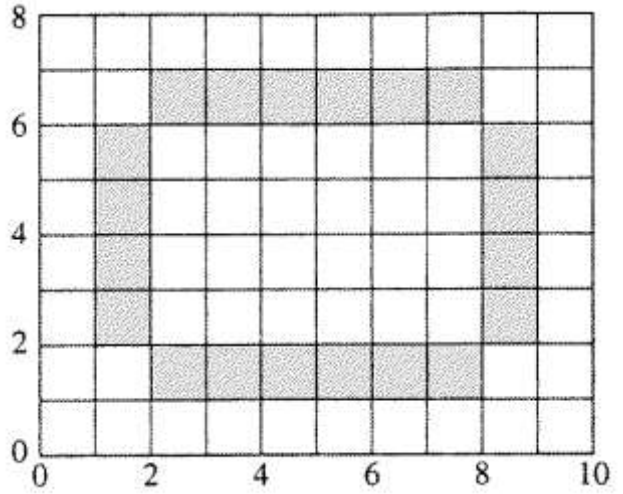


Figure 2–49 Boundary-defined region.

Scan Converting Polygon- Seed Fill Algorithm

- Interior or boundary defined regions are:
- **4 connected** –
  - These are the pixel positions that are right, left, above, and below the current pixel.

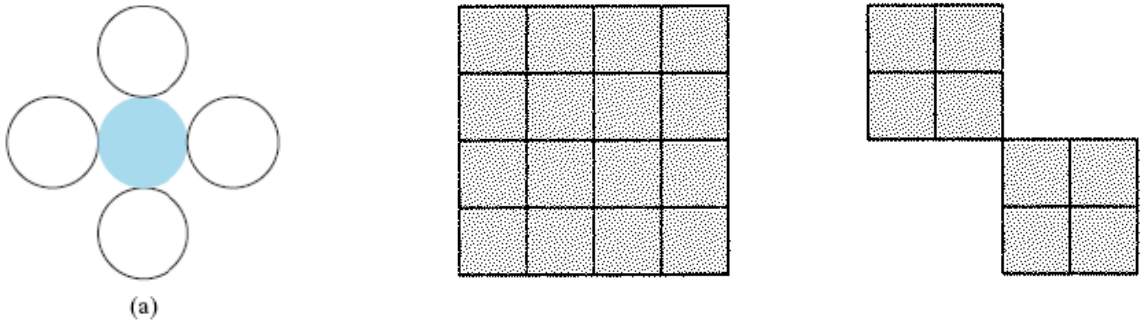


Figure 2–50 Four- and 8-connected interior-defined regions. (a) 4-connected; (b) 8-connected.

- **8 connected** –This method is used to fill more complex figures.
  - Here the set of neighbouring points to be set includes the four diagonal pixels, in addition to the right, left, above, and below the current pixel.

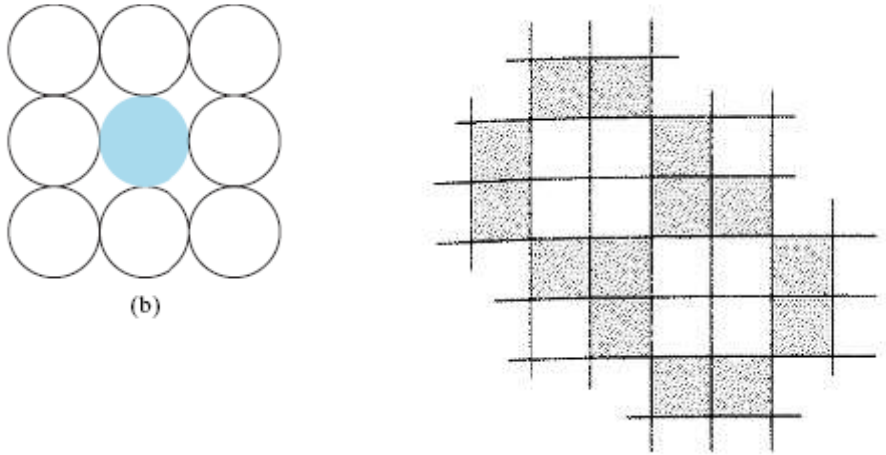


Figure 2–51 Four- and 8-connected boundary-defined regions.

Simple Seed Fill Algorithm

- A simple seed fill algorithm for a boundary-defined region can be developed using a stack.
- The algorithm examines the 4-connected pixels and pushes them on to the stack counter clockwise beginning with the pixel to the right of the current pixel.

Simple seed fill algorithm using a stack:

Push the seed pixel onto the stack

while the stack is not empty

Pop a pixel from the stack

Set the pixel to the required value

For each of the 4-connected pixels adjacent to the current pixel, check if it is a boundary pixel, or if it has already been set to the required value. In either case, ignore it. Otherwise, push it onto the stack.

Simple Seed Fill Algorithm

simple seed fill algorithm *for 4-connected boundary-defined regions*

Seed(x, y) *is the seed pixel*
Push *is a function for placing a pixel on the stack*
Pop *is a function for removing a pixel from the stack*
Pixel(x, y) = Seed(x, y)
initialize stack
Push Pixel(x, y)
while (stack not empty)
    get a pixel from the stack
    Pop Pixel(x, y)
    if Pixel(x, y) < > New value **then**
        Pixel(x, y) = New value
    end if

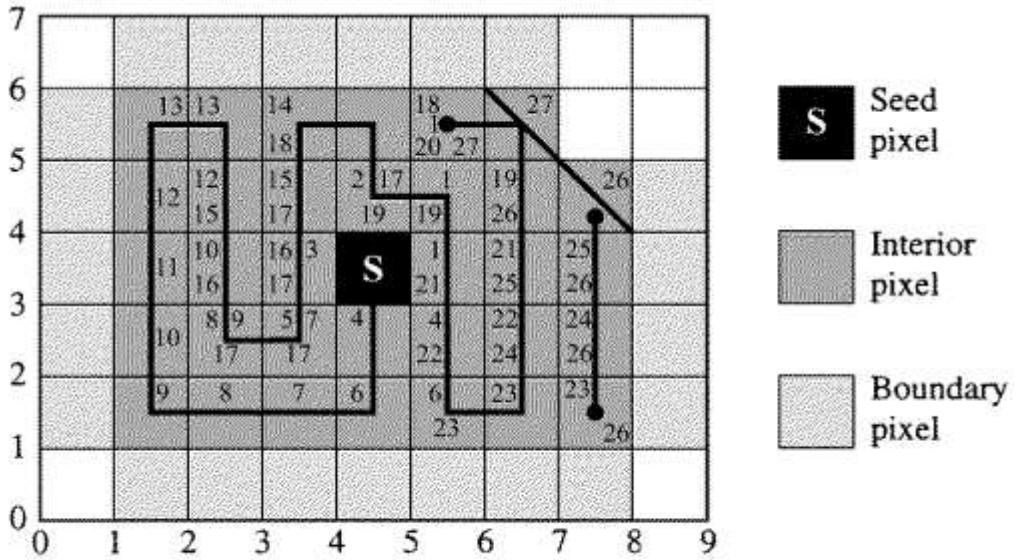


Figure 2–52 Seed fill using a simple stack algorithm.

Simple Seed Fill Algorithm (Cont...)

```
examine the surrounding pixels to see if they should be placed
   onto the stack
if (Pixel(x + 1, y) < > New value and
     Pixel(x + 1, y) < > Boundary value) then
     Push Pixel(x + 1, y)
end if
if (Pixel(x, y + 1) < > New value and
     Pixel(x, y + 1) < > Boundary value) then
     Push Pixel(x, y + 1)
end if
if (Pixel(x − 1, y) < > New value and
     Pixel(x − 1, y) < > Boundary value) then
     Push Pixel(x − 1, y)
 end if
 if (Pixel(x, y − 1) < > New value and
     Pixel(x, y − 1) < > Boundary value) then
     Push Pixel(x, y − 1)
 end if
end while
```

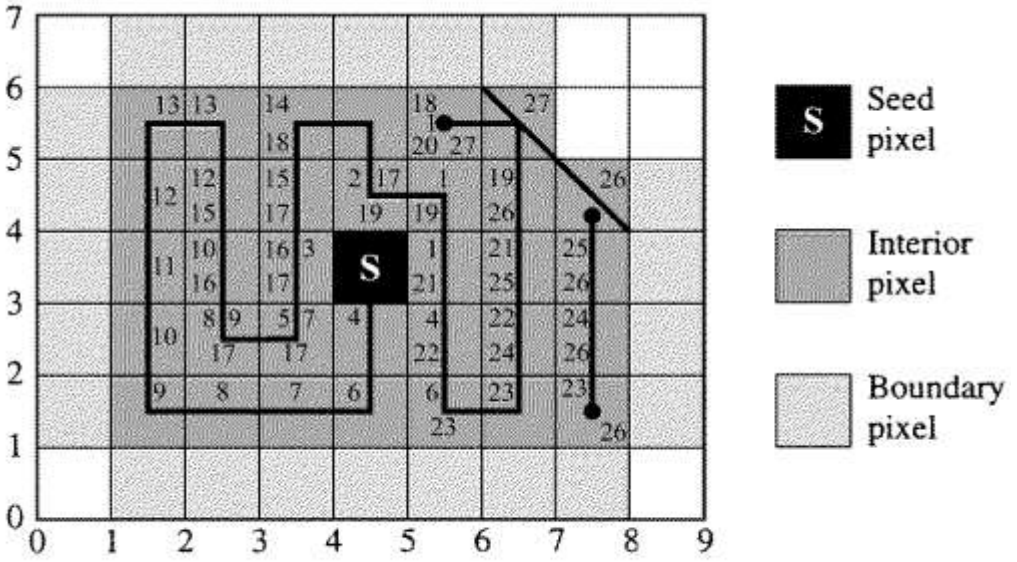


Figure 2–52 Seed fill using a simple stack algorithm.

Simple Seed Fill Algorithm (Cont...)

Example 2–12 Simple Seed Fill Algorithm

As an example of the application of the algorithm consider the boundary-defined polygonal region defined by the vertices $(1,1)$, $(8,1)$ $(8,4)$, $(6,6)$, and $(1,6)$ as shown in Fig. 2–52. The seed pixel is at $(4,3)$. The algorithm fills the polygon pixel by pixel as shown by the line in Fig. 2–52 with the arrows. The ceeds. Notice that some pixels contain more than one number. This indicates that the pixel is pushed onto the stack more than once. When the algorithm reaches pixel $(5,5)$ the stack is 23 levels deep and contains the pixels $(7,4)$, $(7,3)$, $(7,2)$, $(7,1)$, $(6,2)$, $(6,3)$, $(5,6)$, $(6,4)$, $(5,5)$, $(4,4)$, $(3,4)$, $(3,5)$, $(2,4)$, $(2,3)$, $(2,2)$, $(3,2)$, $(5,1)$, $(3,2)$, $(5,2)$, $(3,3)$, $(4,4)$, $(5,3)$.

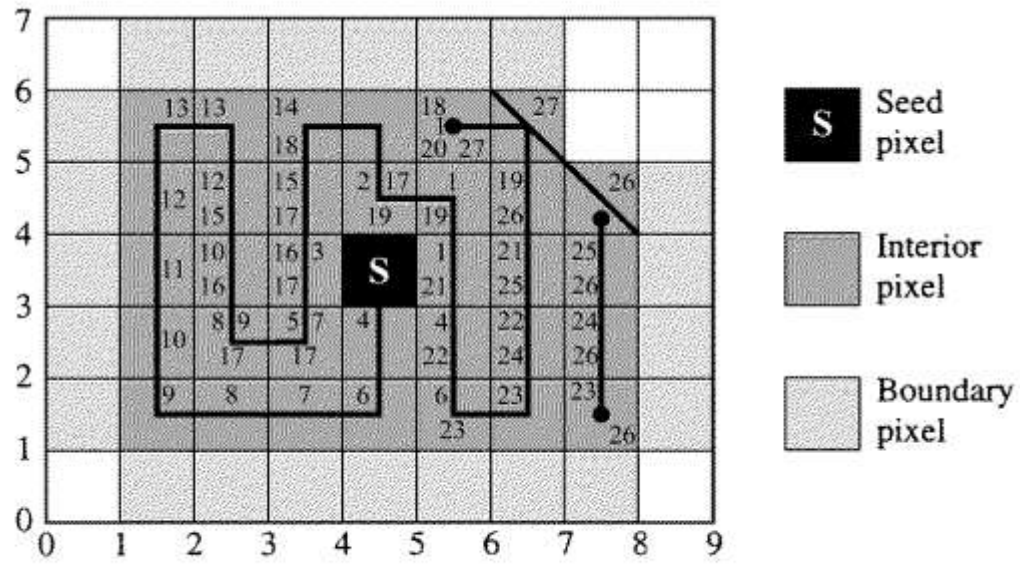
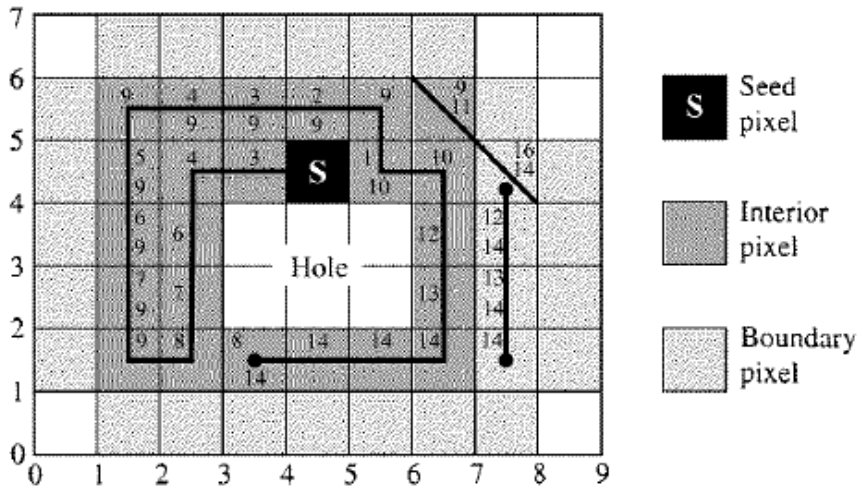


Figure 2–52 Seed fill using a simple stack algorithm.

Simple Seed Fill Algorithm with a Hole

As an example of the application of the algorithm to a polygonal boundary-defined region containing a hole, consider Fig. 2–53. Here, the exterior polygon vertices are the same as in the previous example, i.e., $(1,1)$, $(8,1)$, $(8,4)$, $(6,6)$, and $(1,6)$. The interior hole is defined by $(3,2)$, $(6,2)$, $(6,4)$, $(3,4)$. The seed pixel is at $(4,4)$. Because of the interior hole, the algorithm fills the polygon along a quite different path than in Ex. 2–12. This new path is shown by the arrowed line in Fig. 2–53. Again, the numbers in each pixel give the stack location as the algorithm proceeds. When the algorithm reaches pixel $(3,1)$ all the 4-connected surrounding pixels either contain the new value or are boundary pixels. Hence, no pixels are pushed onto the stack. At this point the stack is 14 levels deep. It contains the pixels $(7,1)$, $(7,2)$, $(7,3)$, $(6,5)$, $(7,4)$, $(6,5)$, $(3,1)$, $(1,2)$, $(1,3)$, $(1,4)$, $(2,5)$, $(3,5)$, $(4,5)$, $(5,4)$.

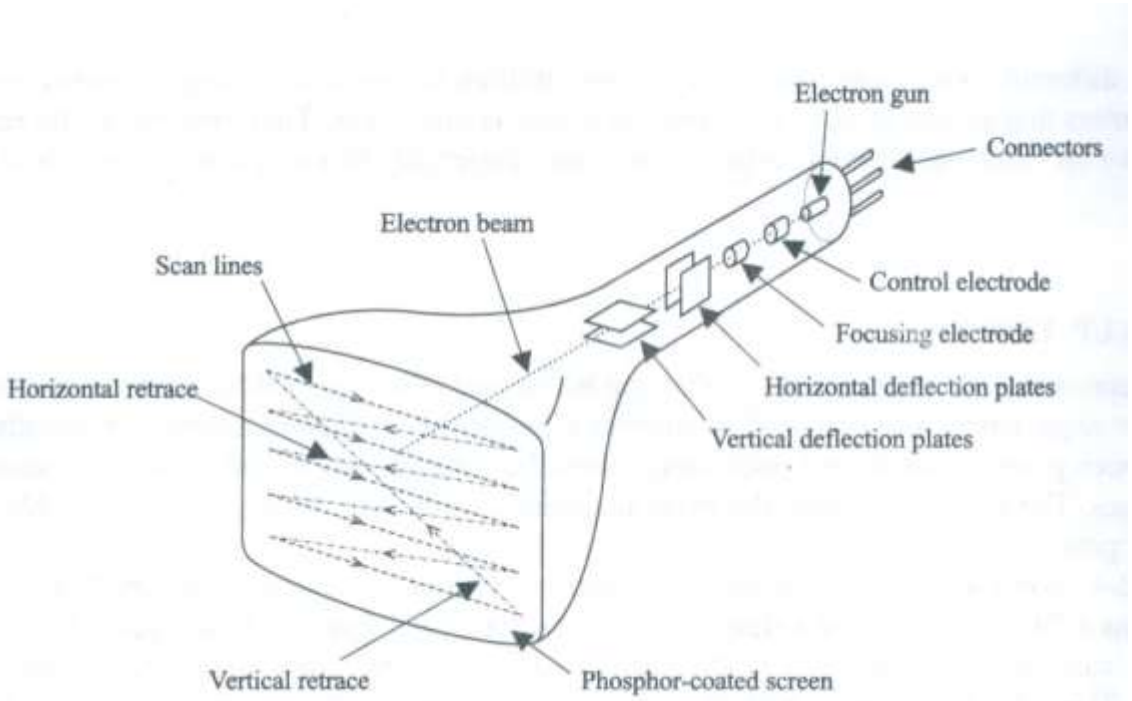
After popping the pixel $(7,1)$ from the stack, the algorithm fills the column $(7,1)$, $(7,2)$, $(7,3)$, $(7,4)$ without pushing any additional pixels onto the stack. At pixel $(7,4)$, again all 4-connected surrounding pixels contain either the new value or are boundary pixels. Returning to the stack, the algorithm finds no new pixels until that at $(6,5)$. Filling the pixel at $(6,5)$ completes the polygon fill. The algorithm completes processing of the stack without further filling. When the stack is empty, the algorithm is complete.



Anti-Aliasing

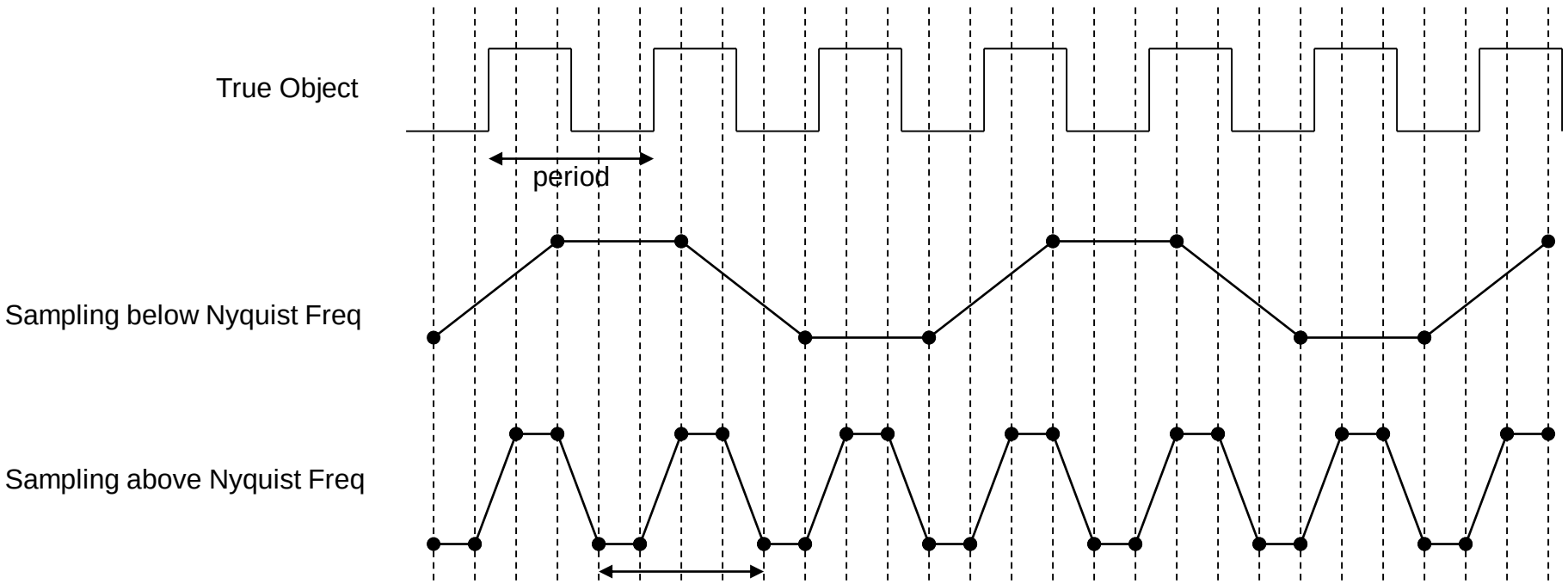
Aliasing

- Aliasing occurs due to the fact that lines, polygon edges etc. are continuous and raster device is discrete.
- To present the line or polygon edge on the raster display device, signals (electron beam) must be sampled at discrete locations.



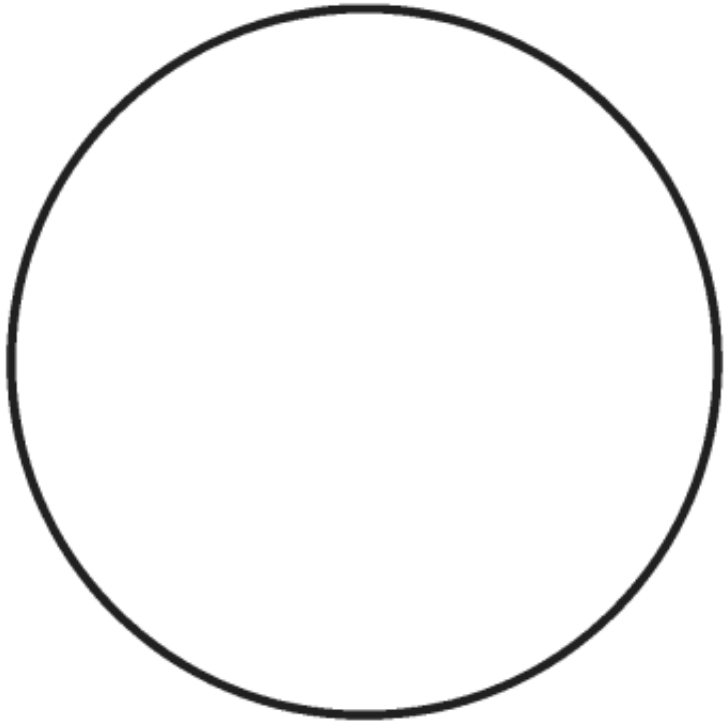
Aliasing

- To avoid aliasing, sampling frequency has to be greater than twice the highest frequency in the signal.
- This minimum sampling frequency to avoid aliasing is also called the Nyquist Sampling Frequency.
- In other words, sample at least twice every period.

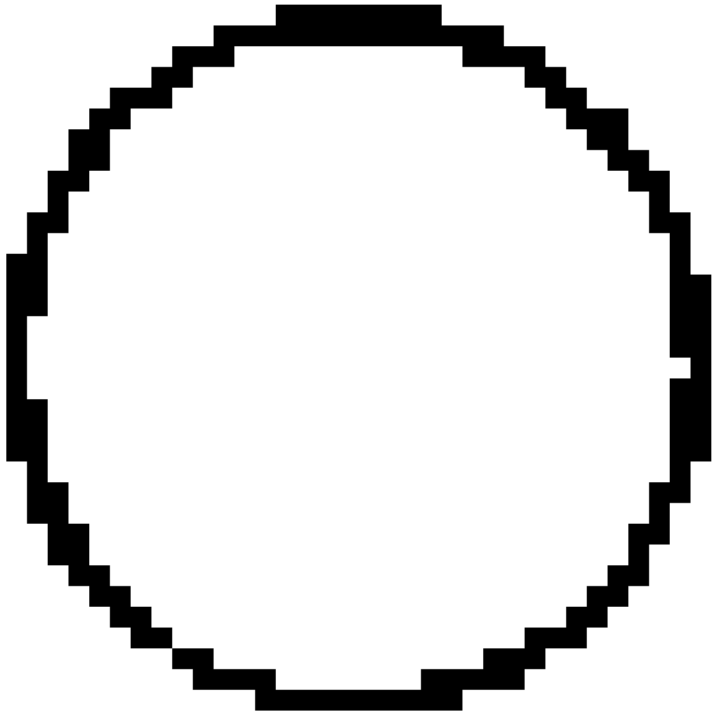


Effects of Aliasing in Graphics

- Jagged effect in rasterised graphics:



Vector representation of a circle



Jagged edges due to aliasing during the rasterisation process

Effects of Aliasing in Graphics

- Poor representation of fine detail:

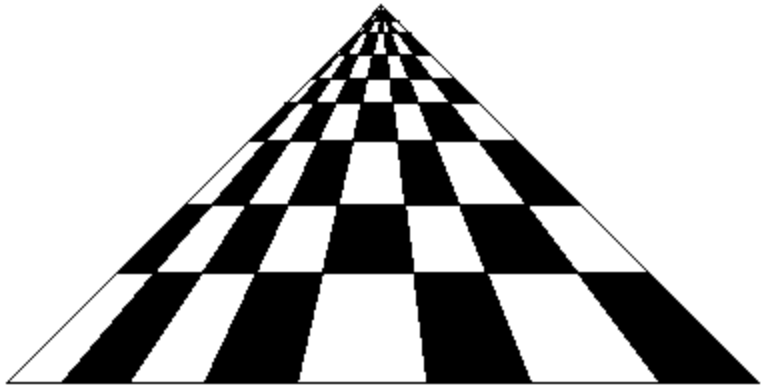
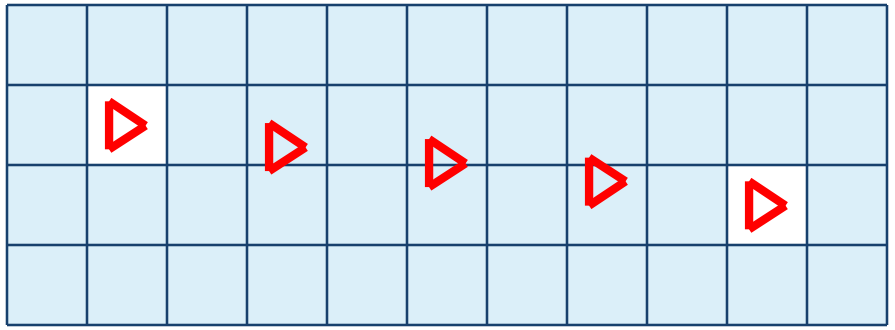


Image representing a chessboard in 3D

Closeup of most distant squares

Effects of Aliasing in Graphics

- For very small objects, if the object do not cover the point within a pixel at which the pixel attributes are evaluated, it will not be included in the resulting picture.

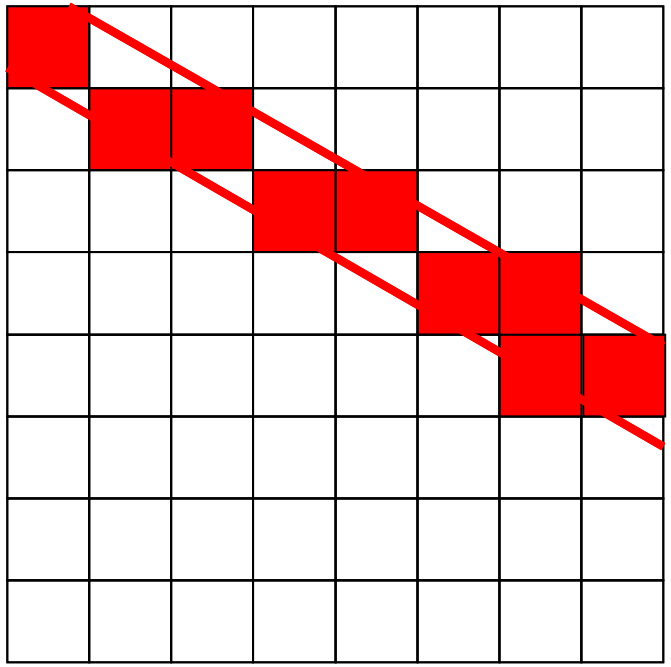


Anti-Aliasing: super-sampling

- 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

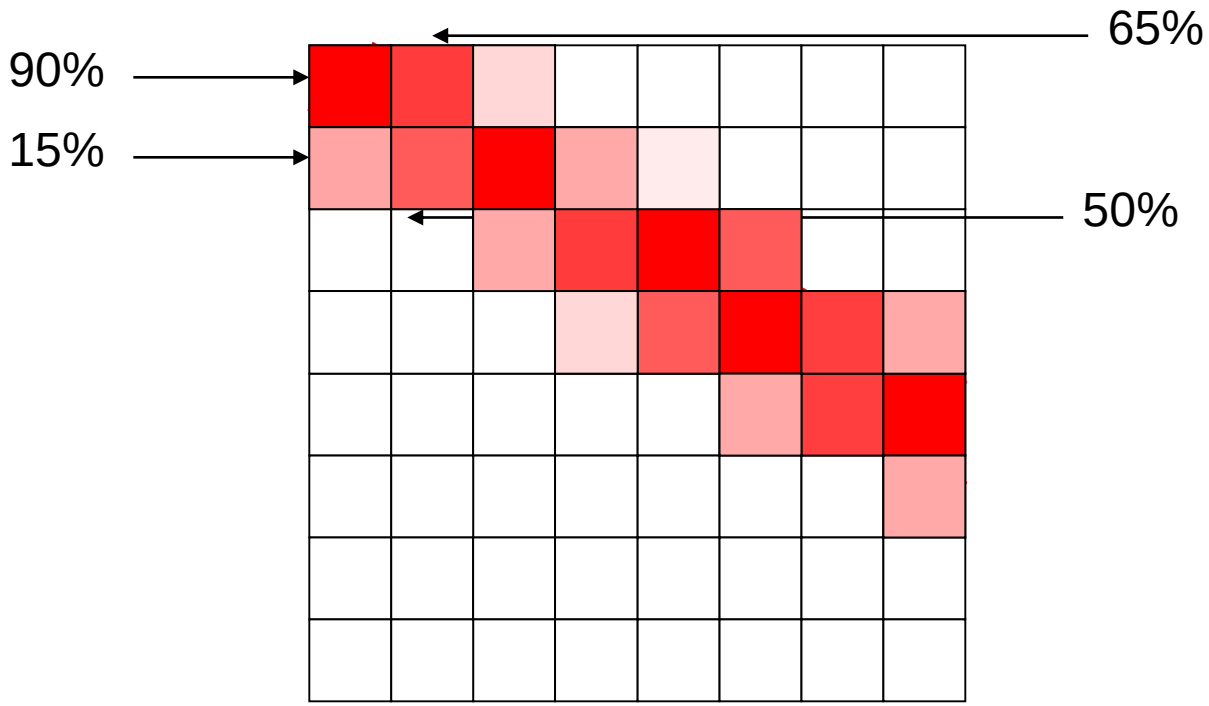
Anti-Aliasing: super-sampling

- The simplest way to rasterise would be to fill only those pixels with >50% coverage.



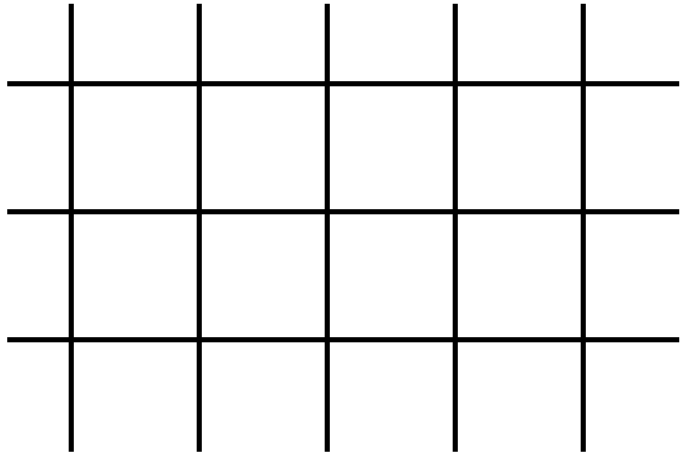
Anti-Aliasing Techniques :super-sampling

- A better method would be to determine the strength of the colour used to fill each individual pixel according to the percentage covered by the line.

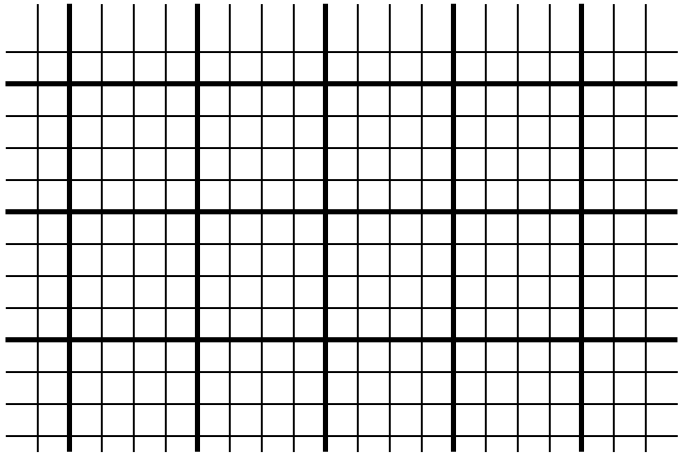


Super-Sampling

- Split single pixel into sub-pixels.
- Pixel's final color is a mixture of sub-pixels' colors. Simple method: Sample at the middle of each sub-pixel. Then, pixel's color is the average of the sub-pixels' color.

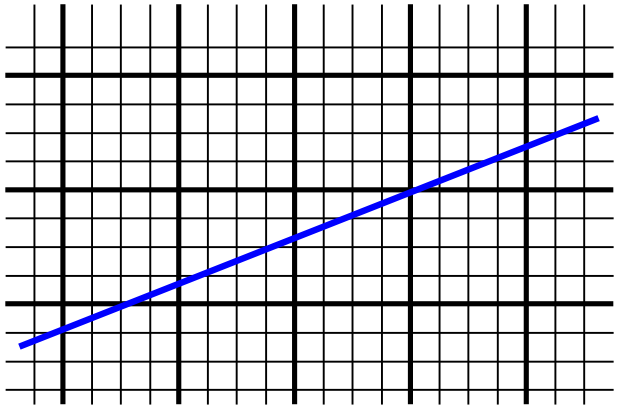


Pixels



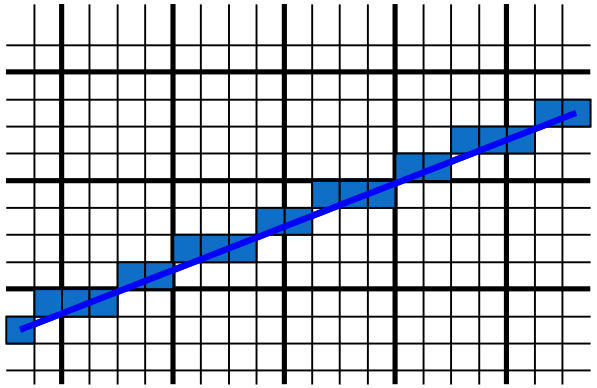
Sub-divide into sub-pixels

Super-Sampling a Zero-Width Line



Sub-divide each pixel into sub-pixels, for example 4x4 sub-pixels

Apply Bresenham's algorithm at sub-pixel level



Each pixel can have a maximum of 4 colored sub-pixels (n)

0.0	0.0	0.0	1.0	0.75
0.5	1.0	1.0	0.0	
0.5	0.0	0.0	0.0	

Fraction of pixel's color to be line's color

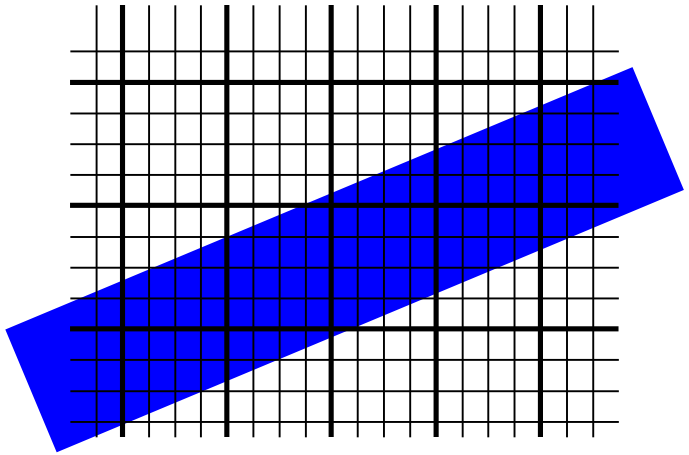
$I_{line} = \frac{s}{n}$

Assign color

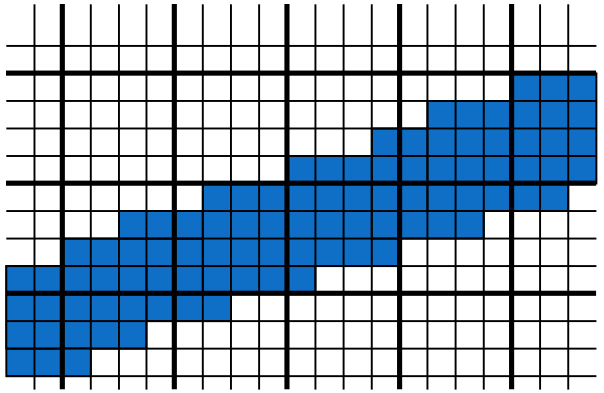
0	0	0	4	**3**
2	4	4	0	
2	0	0	0	

How many sub-pixels are colored/activated?(s)

Super-Sampling a Line with Non-Zero Width



A line that is one-pixel wide.
For every pixel: Maximum number of sub-pixels inside line = 16



A sub-pixel is considered in if its lower-left corner is inside the line

0	0	5/16	11/16
10/16	15/16	13/16	7/16
8/16	2/16	0	0

Fraction of sub-pixels are in = fraction of color of the pixel should be line color

https://youtu.be/R3AgvpOA2qw

END OF UNIT 3