

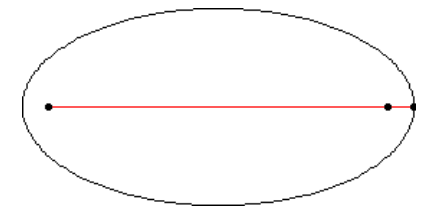
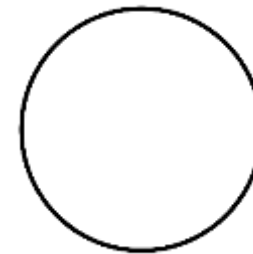
# **Computer Graphics & Multimedia Techniques**

**Unit-2**

**Graphics Primitives:**

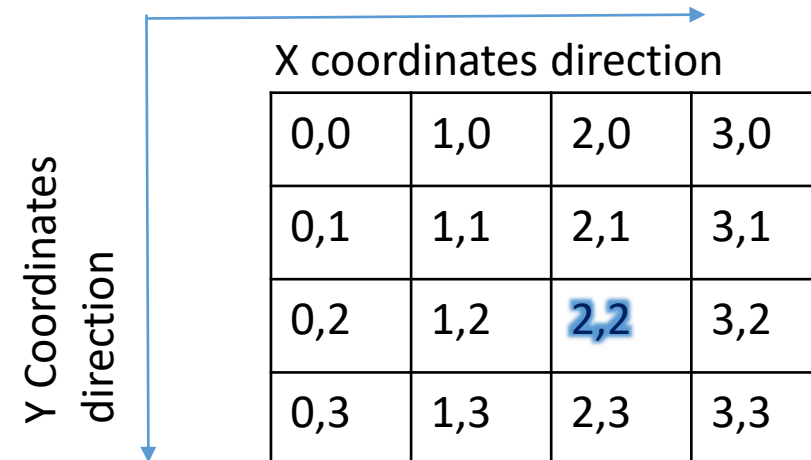
# Scan conversion:

- The process of representing continuous graphics objects as a collection of discrete pixels is called scan conversion.
- Computer graphics objects for scan conversion are followings:
  - **Point**
  - **Line**
  - **Circle**
  - **Ellipse**



# Point:

- Point:
  - Scan-Converting a point involves illuminating the pixel that contains the point.
  - A point  $p(x, y)$  is represented by the integer part of  $x$  & the integer part of  $y$  that is pixels  $[(\text{INT}(x), \text{INT}(y))]$ .
  - Ex.  $P(2, 2)$  highlighted in blue color



Y Coordinates direction

X coordinates direction

0,0	1,0	2,0	3,0
0,1	1,1	2,1	3,1
0,2	1,2	2,2	3,2
0,3	1,3	2,3	3,3

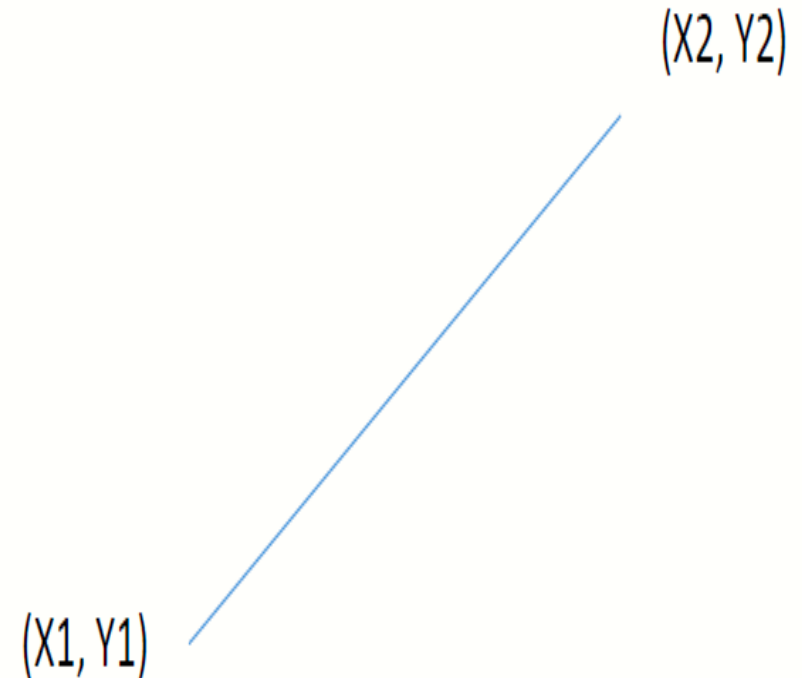
# Line:

- Two Approaches for Line

- A straight line may be defined by two endpoints & an equation. The two endpoints are described by  $(x_1, y_1)$  and  $(x_2, y_2)$ . The equation of the line is used to determine the  $x, y$  coordinates of all the points that lie between these two endpoints.

*$y = mx + b$  where  $m = (dy/dx)$  and  $b$  is constant*

- **Digital Differential Analyzer**
- **Bresenham's Line Algorithm**



# DDA (Digital Differential Analyzer)

- A linear DDA starts by calculating the smaller of  $dy$  or  $dx$  for a unit increment of the other. A line is then sampled at unit intervals in one coordinate and corresponding integer values nearest the line path are determined for the other coordinate.
- Considering a line with positive slope, if the slope is less than or equal to 1, we sample at unit  $x$  intervals ( $dx=1$ ) and compute successive  $y$  values as

When  $|M| < 1$  then (assume that  $x_1 < x_2$ )

$x = x_1$ ,

$y = y_1$

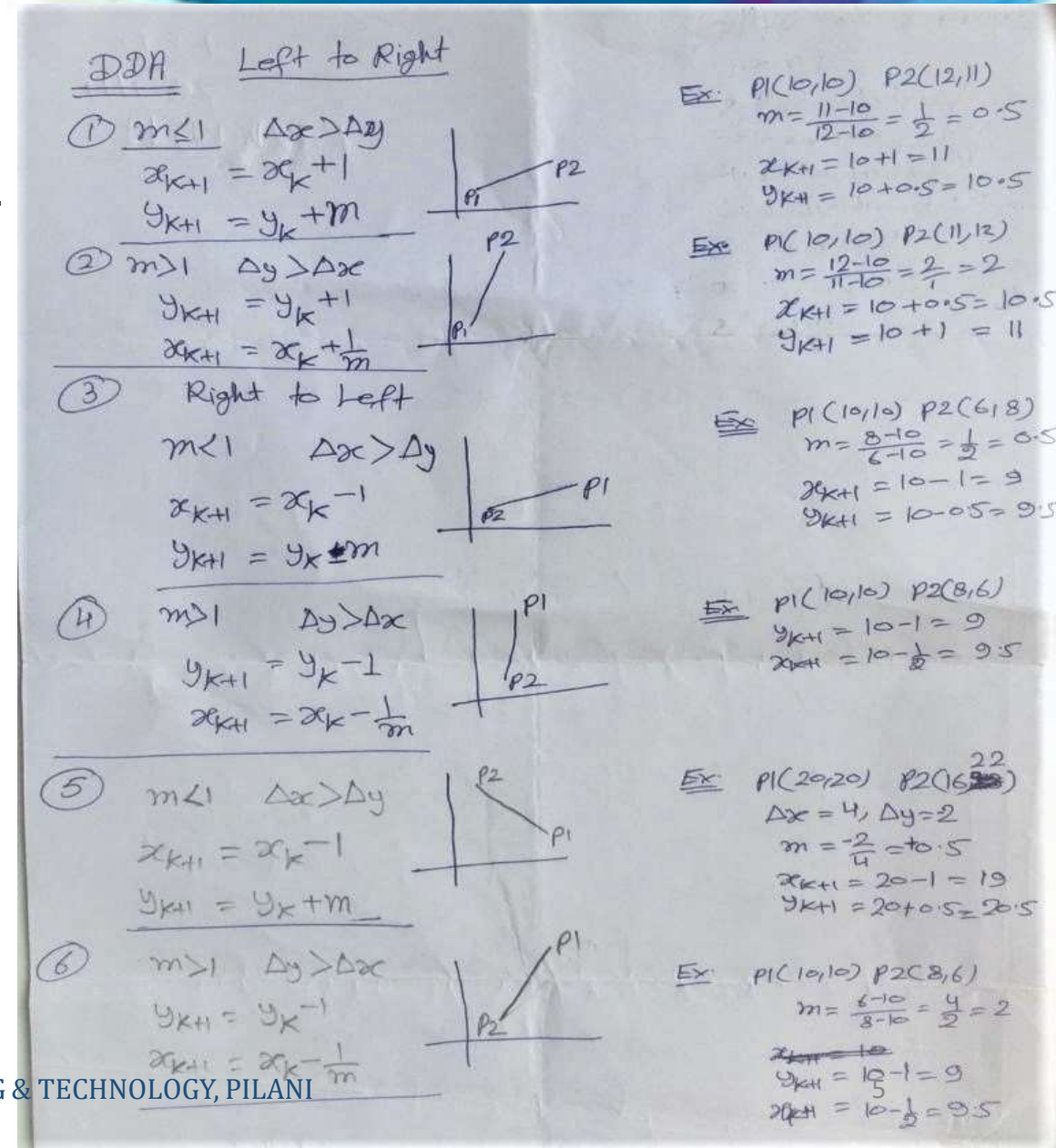
set  $\Delta x = 1$

$y_{i+1} = y + m$ ,

$x = x + 1$

Until  $x \neq x_2$

All the cases you can see in figure





# Bresenham's Line Algorithm

This algorithm is used for scan converting a line. It was developed by Bresenham. It is an efficient method because it involves only integer addition, subtractions, and multiplication operations. These operations can be performed very rapidly so lines can be generated quickly.

1. It involves only integer arithmetic, so it is simple.
2. It avoids the generation of duplicate points.
3. It can be implemented using hardware because it does not use multiplication and division.
4. It is faster as compared to DDA (Digital Differential Analyzer) because it does not involve floating point calculations like DDA Algorithm.

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

1. Input the two line endpoints and store the left endpoint in  $(x_0, y_0)$ .
2. Load  $(x_0, y_0)$  into the frame buffer; that is, plot the first point.
3. Calculate constants  $\Delta x$ ,  $\Delta y$ ,  $2\Delta y$ , 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  $p_k < 0$ , the next point to plot is  $(x_k + 1, y_k)$  and

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

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  $\Delta x$  times.

# Bresenham's Line Algorithm

## Example 3-1 Bresenham Line Drawing

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

$$\Delta x = 10, \quad \Delta y = 8$$

The initial decision parameter has the value

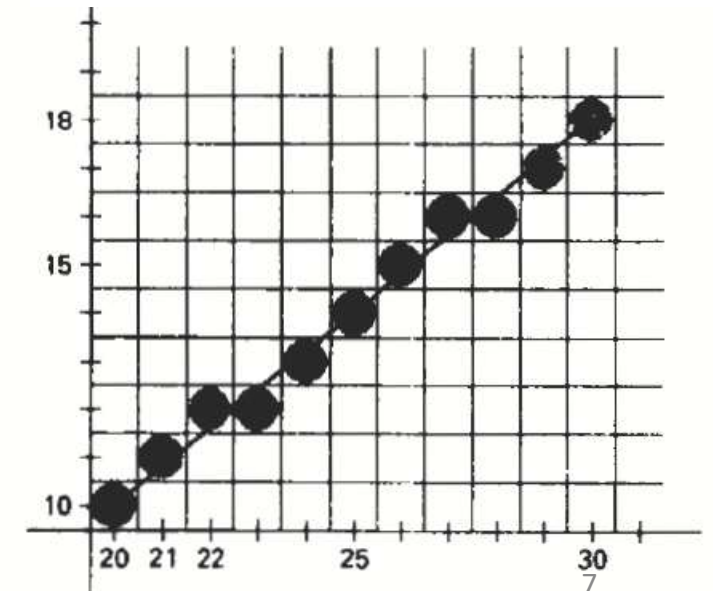
$$\begin{aligned} p_0 &= 2\Delta y - \Delta x \\ &= 6 \end{aligned}$$

and the increments for calculating successive decision parameters are

$$2\Delta y = 16, \quad 2\Delta y - 2\Delta x = -4$$

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)



# Circle: Midpoint Circle Algorithm

## Midpoint Circle Algorithm

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

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

2. Calculate the initial value of the decision parameter as

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

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  $(x_c, y_c)$  and plot the coordinate values:

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

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



# Circle: Midpoint Circle Algorithm

## Example 3-2 Midpoint Circle-Drawing

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

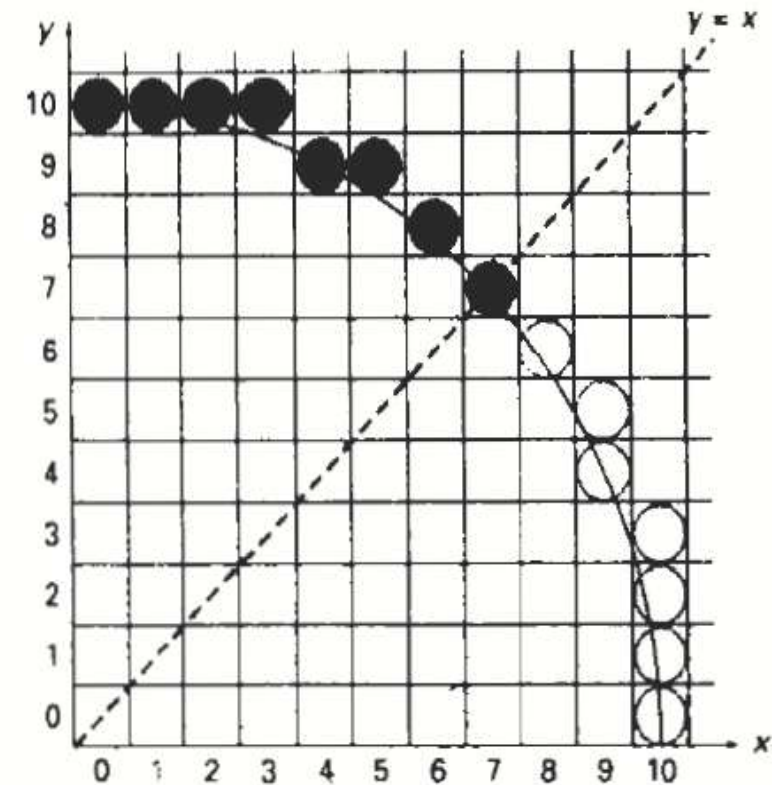
$$p_0 = 1 - r = -9$$

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

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

Successive decision parameter values and positions along the circle path are calculated using the midpoint method as

$k$	$p_k$	$(x_{k+1}, y_{k+1})$	$2x_{k+1}$	$2y_{k+1}$
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



# Ellipse: Midpoint Ellipse Algorithm

## 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 circle 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 using the last point  $(x_0, y_0)$  calculated in region 1 as

4. Calculate the initial value of the decision parameter in region 2 using the last point  $(x_0, y_0)$  calculated in region 1 as

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

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 circle 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.

6. Determine symmetry points in the other three quadrants.
7. Move each calculated pixel position  $(x, y)$  onto the elliptical path centered on  $(x_c, y_c)$  and plot the coordinate values:

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

8. Repeat the steps for region 1 until  $2r_y^2 x \geq 2r_x^2 y$ .

# Ellipse: Midpoint Ellipse Algorithm

## Example 3-3 Midpoint Ellipse Drawing

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

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

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

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

Successive decision parameter values and positions along the ellipse path are calculated using the midpoint method as

$k$	$p1_k$	$(x_{k+1}, y_{k+1})$	$2r_y^2x_{k+1}$	$2r_x^2y_{k+1}$
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

We now move out of region 1, since  $2r_y^2x > 2r_x^2y$ .

We now move out of region 1, since  $2r_y^2x > 2r_x^2y$ .

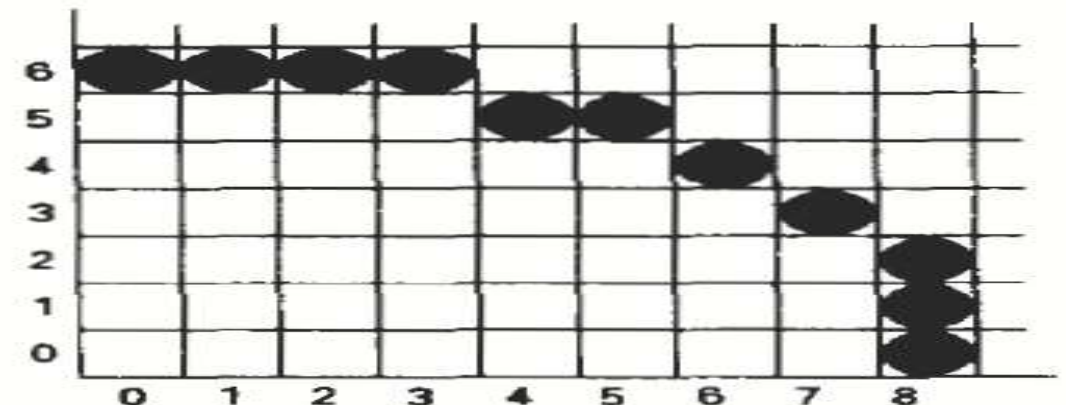
For region 2, the initial point is  $(x_0, y_0) = (7, 3)$  and the initial decision parameter is

$$p2_0 = f\left(7 + \frac{1}{2}, 2\right) = -151$$

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

$k$	$p2_k$	$(x_{k+1}, y_{k+1})$	$2r_y^2x_{k+1}$	$2r_x^2y_{k+1}$
0	-151	(8, 2)	576	256
1	233	(8, 1)	576	128
2	745	(8, 0)	—	—

A plot of the selected positions around the ellipse boundary within the first quadrant is shown in Fig. 3-23.



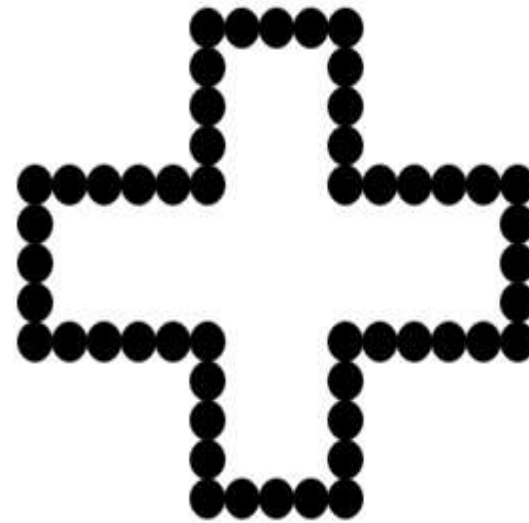


# Fill area primitives

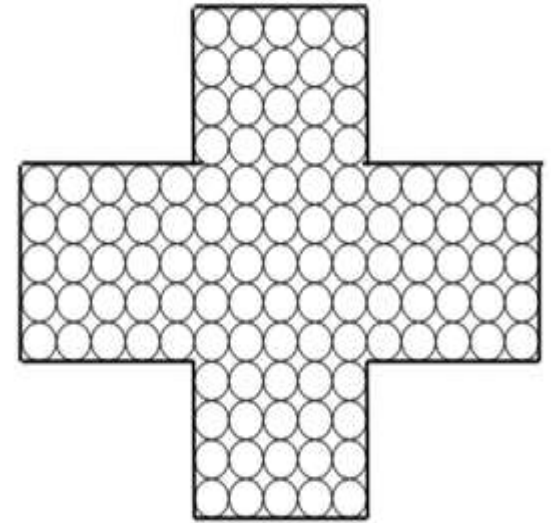
Fill area primitives/Region filling is the process of filling image or region. Filling can be of boundary or interior region as shown in figure.

Algorithms are followings:

- Scan Line Polygon fill
- Boundary fill
- Flood Fill



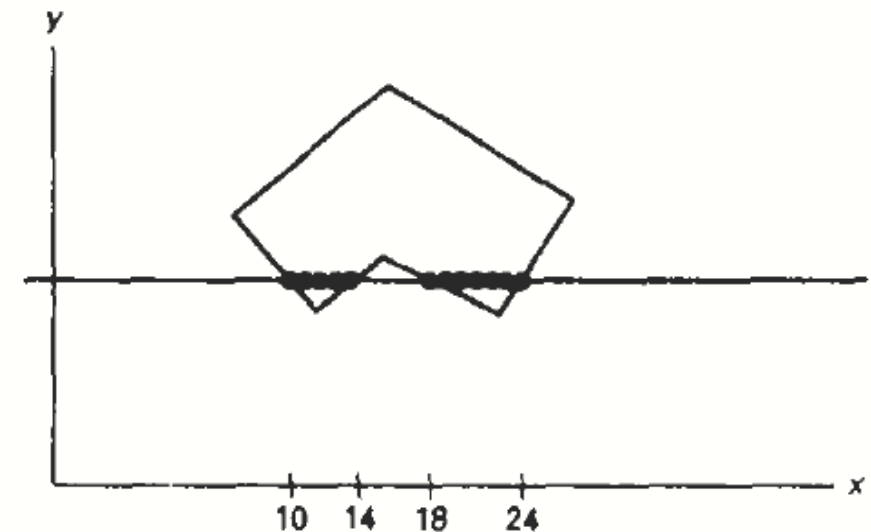
Boundary Filled Region



Interior or Flood Filled Region

# Scan Line Polygon fill

- For each scan line crossing a polygon, the area-fill algorithm locates the intersection points of the scan line with the polygon edges. These intersection points are then sorted from left to right, and the corresponding frame-buffer positions between each intersection pair are set to the specified fill color. In the example of Figure the four pixel intersection positions with the polygon boundaries define two stretches of interior pixels from  $x = 10$  to  $x = 14$  and from  $x = 18$  to  $x = 24$ .



*Figure 3-35*

Interior pixels along a scan line passing through a polygon area.



# Scan Line Polygon fill

- Some scan-line intersections at polygon vertices require special handling. A scan line passing through a vertex intersects two polygon edges at that position, adding two points to the list of intersections for the scan line. Figure shows two scan lines at positions  $y$  and  $y'$  that intersect edge endpoints. Scan line  $y$  intersects five polygon edges. Scan line  $y'$ , however, intersects an even number of edges although it also passes through a vertex. Intersection points along scan line  $y'$  correctly identify the interior pixel spans. But with scan line  $y$ , we need to do some additional processing to determine the correct interior points.

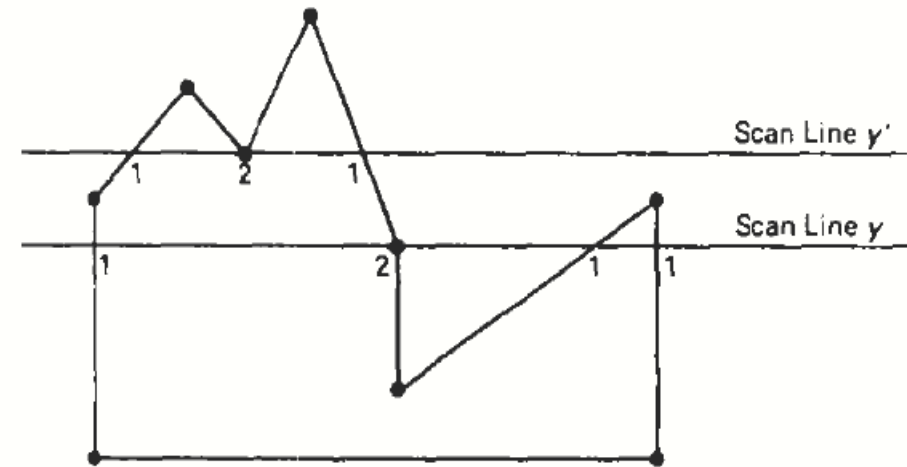


Figure 3-36

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.

# Scan Line Polygon fill

- One way to resolve the question as to whether we should count a vertex as one intersection or two is to shorten some polygon edges to split those vertices that should be counted as one intersection. We can process nonhorizontal edges around the polygon boundary in the order specified, either clockwise or counterclockwise. As we process each edge, we can check to determine whether that edge and the next nonhorizontal edge have either monotonically increasing or decreasing endpoint  $y$  values. If so, the lower edge can be shortened to ensure that only one intersection point is generated for the scan line going through the common vertex joining the two edges. Figure 3-37 illustrates shortening of an edge.

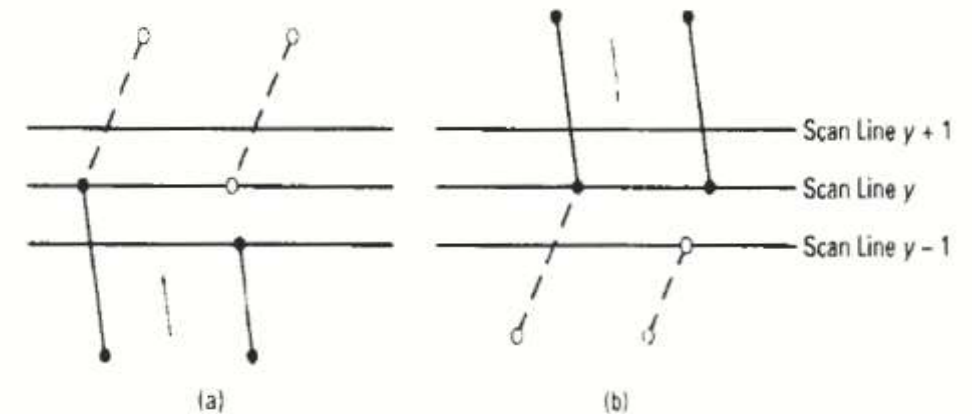


Figure 3-37

Adjusting endpoint  $y$  values for a polygon, as we process edges in order around the polygon perimeter. The edge currently being processed is indicated as a solid line. In (a), the  $y$  coordinate of the upper endpoint of the current edge is decreased by 1. In (b), the  $y$  coordinate of the upper endpoint of the next edge is decreased by 1.

# Inside-Outside test: Odd-even Rule and Nonzero winding

- **The odd-even rule:** also called the odd parity rule or the even-odd rule, by conceptually drawing a line from any position  $P$  to a distant point outside the coordinate extents of the object and counting the number of edge crossings along the line. If the number of polygon edges crossed by this line is odd, then  $P$  is an **interior** point. Otherwise,  $P$  is an **exterior** point. To obtain an accurate edge count, we must be **sure** that the line path we **choose** does not intersect any polygon vertices. Figure (a) shows the interior and exterior regions obtained from the odd-even rule for a self-intersecting set of edges. The scan-line polygon fill algorithm discussed in the previous section is an example of area filling **using** the odd-even rule.
- **Nonzero winding:** Another method for defining interior regions is the nonzero winding number rule, which counts the number of times the polygon edges wind around a particular point in the counterclockwise direction. This count is called the winding number, and the interior points of a two-dimensional object are defined to be those that have a nonzero value for the winding number.

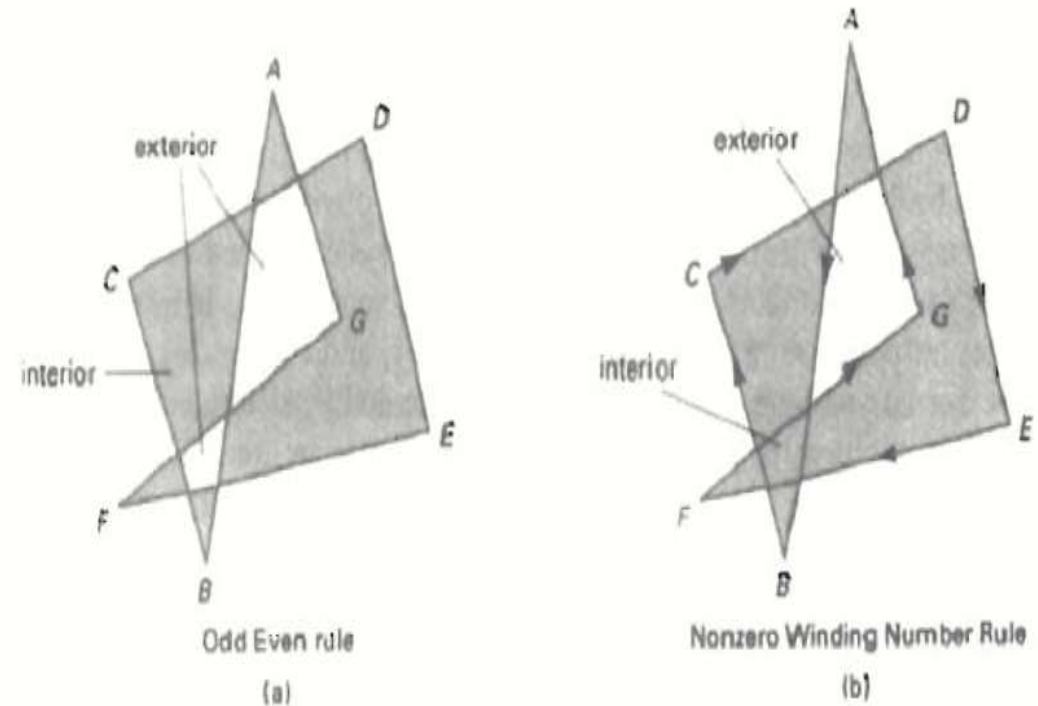


Figure 3-40  
Identifying interior and exterior regions for a self-intersecting polygon.

# Boundary fill

- Another approach to area filling is to start at a point inside a region and paint the interior outward toward the boundary. If the boundary is specified in a single color, the fill algorithm proceeds outward pixel by pixel until the boundary color is encountered. This method, called the boundary-till algorithm, is particularly useful in interactive painting packages, where interior points *are* easily selected. A boundary-fill procedure accepts as input the coordinates of an interior point  $(x, y)$ , a fill color, and a boundary color. Starting from  $(x, y)$ , the procedure tests neighboring positions to determine whether they are of the boundary color. If not, they are painted with the fill color, and their neighbors are tested. This process continues until all pixels up to the boundary color for the area have been tested. Both inner and outer boundaries can be set up to specify an area, and some examples of defining regions for boundary fill are shown in Figure.

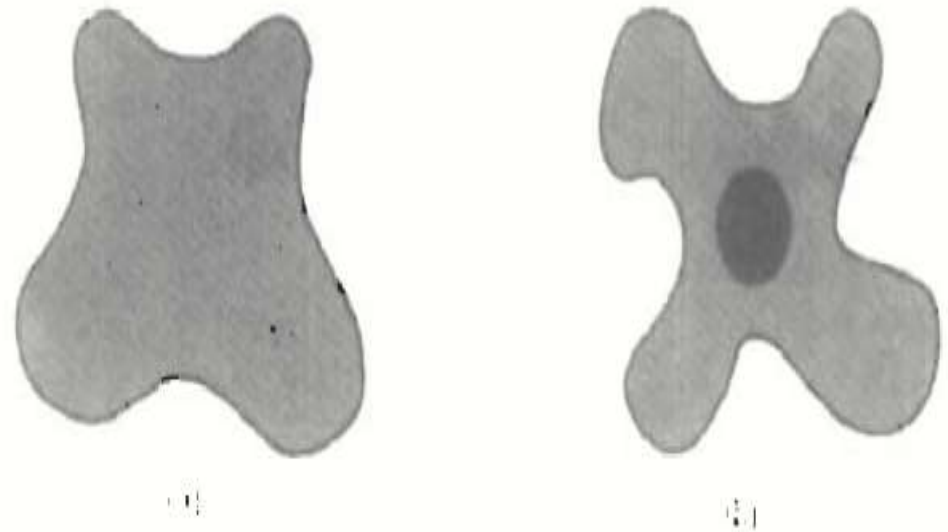


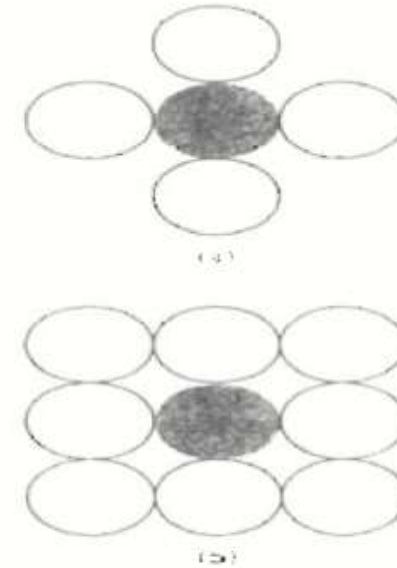
Figure 3-42

Example color boundaries for a boundary-fill procedure.



# Boundary fill: 4-connected and 8-connected

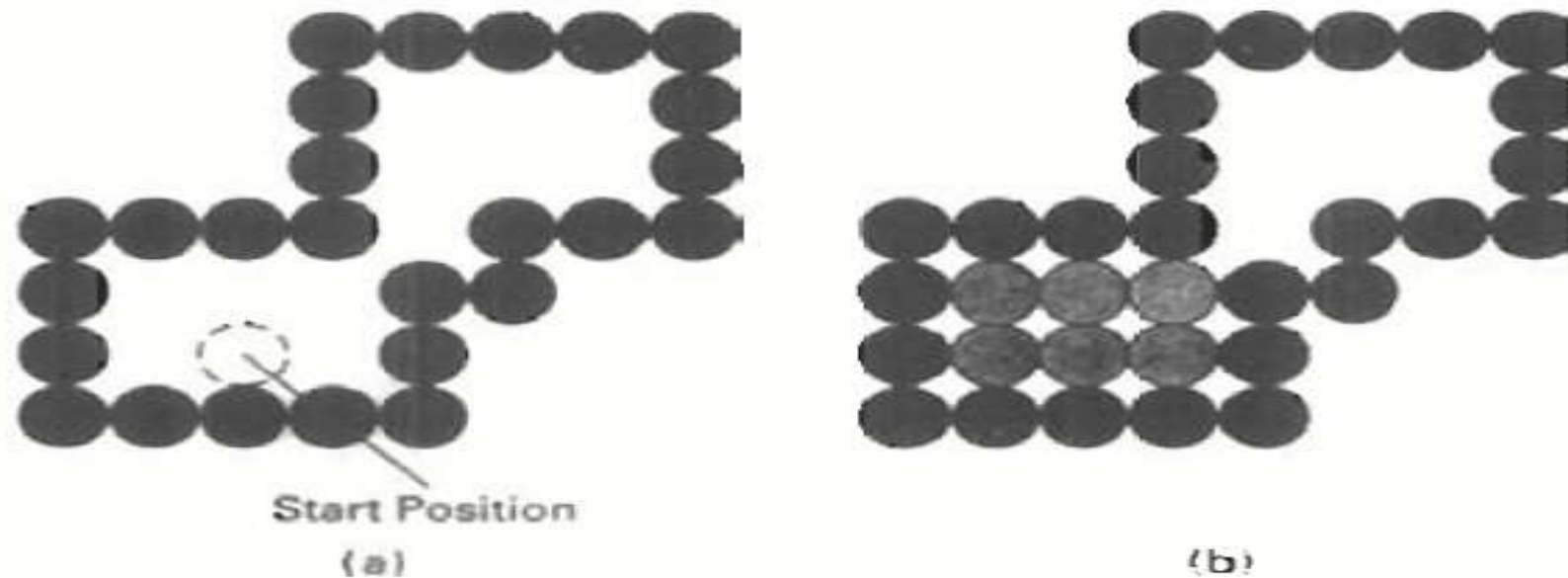
- Figure shows two methods for proceeding to neighboring pixels from the current test position. In Figure(a), four neighboring points are tested. These are the pixel positions that are right, left, above, and below the current pixel. Areas filled by this method are called **4-connected**. The second method, shown in Figure(b), is used to fill more complex figures. Here the set of neighboring positions to be tested includes the four diagonal pixels. Fill methods using this approach are called **8-connected**.



**Figure 3-43**  
Fill methods applied to a 4-connected area (a) and to an 8-connected area (b). Open circles represent pixels to be tested from the current test position, shown as a solid color



# Boundary fill: 4-connected Issue



**Figure 3-44**

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

# Flood fill (Recoloring):

- Sometimes we want to fill in (or recolor) an area that is not defined within a single color boundary. Figure shows an area bordered by several different color regions. We can paint such areas by replacing a specified interior color instead of searching for a boundary color value. This approach is called a flood-fill algorithm. We start from a specified interior point  $(x, y)$  and reassign 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 reassign pixel values so that all interior points have the same color. Using either a 4-connected or 8-connected approach, we then step through pixel positions until all interior points have been repainted. The following procedure flood fills a 4-connected region recursively, starting from the input position.

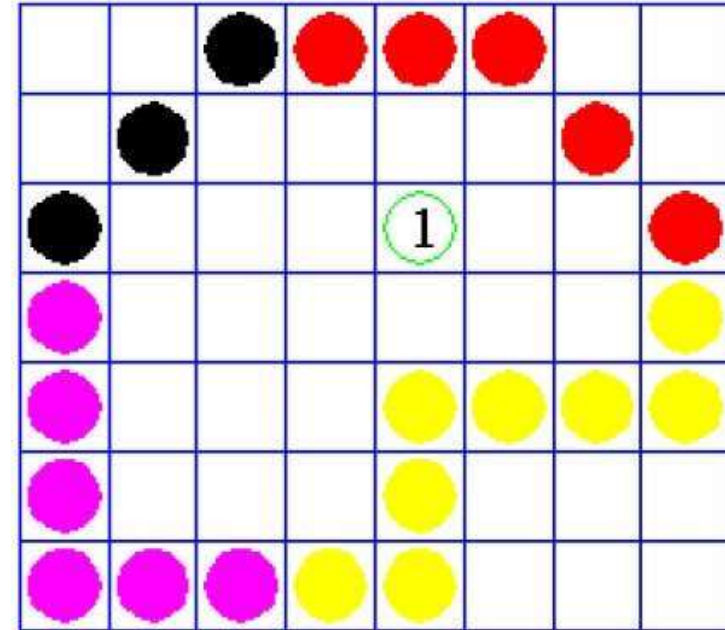


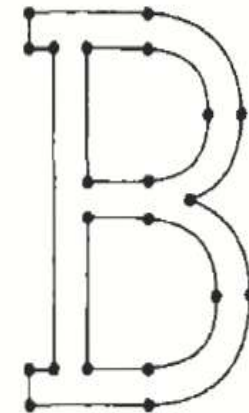
Figure: An area defined within multiple color boundaries.

# Character generation:

- Two different representations are used for storing computer fonts.
- **Bitmap:** A simple method for representing the character shapes in a particular typeface is to use rectangular grid patterns. The set of characters are then referred to as a bitmap font (or bitmapped font). Bitmap fonts are the simplest to define and display: The character grid only needs to be mapped to a frame-buffer position. In general, however, bitmap fonts require more space, because each variation (size and format) must be stored in a *font* cache. It is possible to generate different sizes and other variations, such as bold and italic, from one set, but this usually does not produce good results.
- **Outline:** Another, more flexible, scheme is to describe character shapes using straight-line and curve sections, as in PostScript, for example. In this case, the set of characters is called an outline font. In contrast to bitmap fonts, outline fonts require less storage since each variation does not require a distinct font cache. We can produce boldface, italic, or different sizes by manipulating the curve definitions for the character outlines. But it does take more time to process the outline fonts, because they must be scan converted into the frame buffer.

1	1	1	1	1	1	0	0
0	1	1	0	0	1	1	0
0	1	1	0	0	1	1	0
0	1	1	1	1	1	0	0
0	1	1	0	0	1	1	0
0	1	1	0	0	1	1	0
1	1	1	1	1	1	0	0
0	0	0	0	0	0	0	0

(a)



(b)

Figure 3-48

The letter B represented in (a) with an 8 by 8 bilevel bitmap pattern and in (b) with an outline shape defined with straight-line and curve segments.

# Line attributes:

- Basic attributes of a straight line segment are its type, its width, and its color.
- **Line Type:** Possible selections for the line-type attribute include solid lines, dashed lines, and dotted lines.
- **Line Width:** Line-width parameter is assigned a positive number to indicate the relative width of the line to be displayed. A value of 1 specifies a standard-width line. On pen plotter, for instance, a user could set lw to a value of 0.5 to plot a line whose width is half that of the standard line.

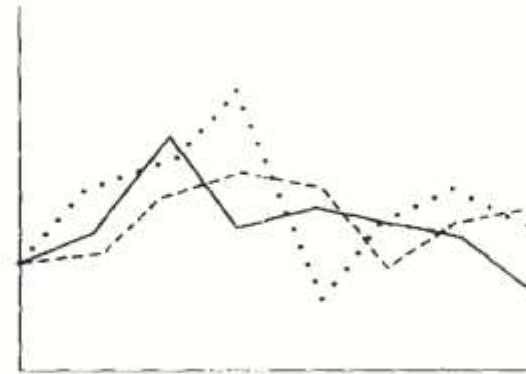


Figure 4-1  
Plotting three data sets with three different line types, as output by the chartData procedure.

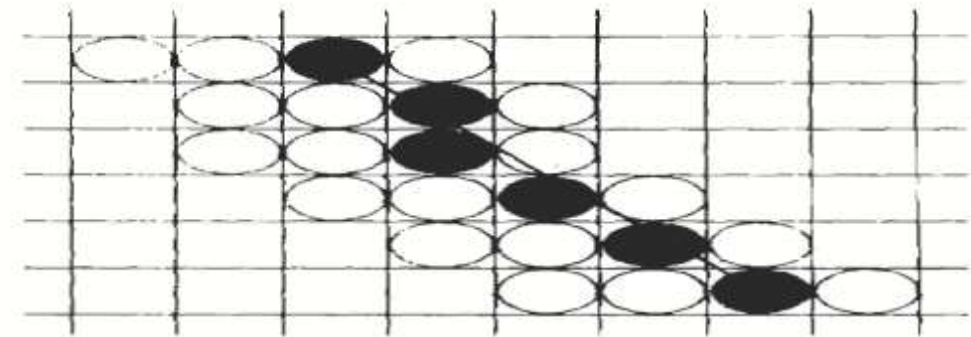
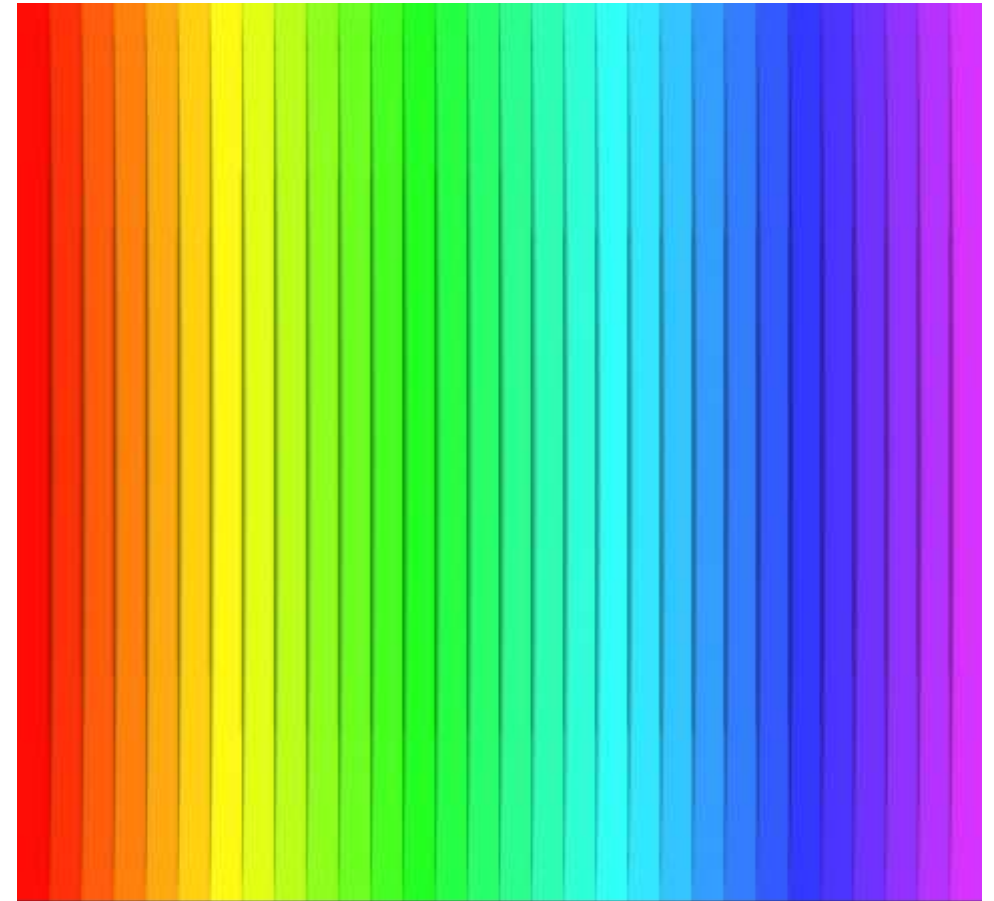


Figure 4-4  
Raster line with slope  $|m| > 1$  and line-width parameter  $lw = 4$  plotted with horizontal pixel spans.



# Line attributes:

- **Line Color:** When a system provides color (or intensity) options, a parameter giving the current color index is included in the list of system-attribute values. A polyline routine displays a line in the current color by setting this color value in the frame buffer at pixel locations along the line path using the **setpixel** procedure. The number of color choices depends on the number of bits available per pixel in the frame buffer.





# Area-fill attributes

- Options for filling a defined region include a choice between a solid color or a patterned fill and choices for the particular colors and patterns. These fill options can be applied to polygon regions or to areas defined with curved boundaries, depending on the capabilities of the available package. In addition, areas can be painted using various brush styles, colors, and transparency parameters.
- **Fill Styles**
- Areas are displayed with three basic fill styles: hollow with a color border, filled with a solid color, or filled with a specified pattern or design.

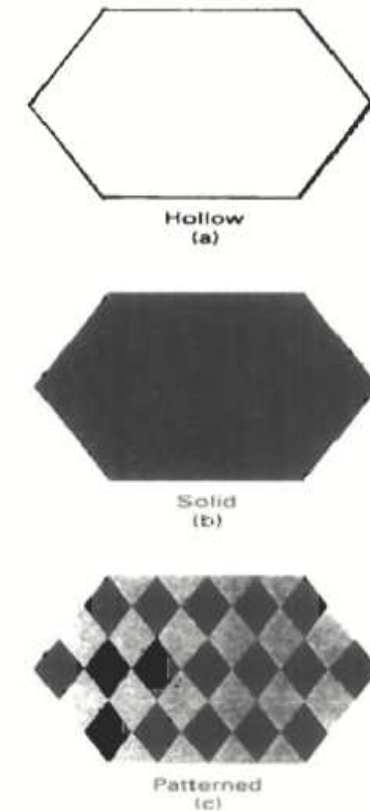
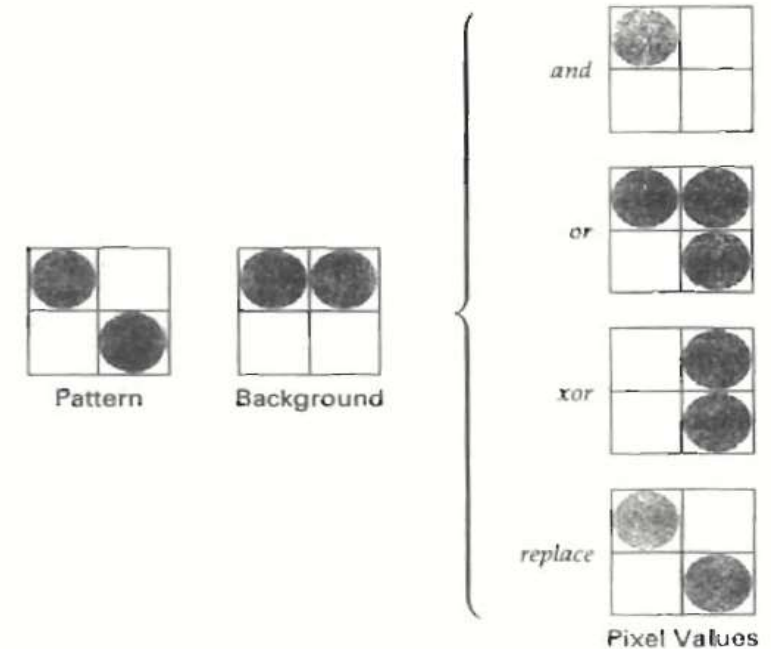


Figure 4-18  
Polygon fill styles.

# Area-fill attributes

- **Soft Fill:**
- Modified boundary-fill and flood-fill procedures that are applied to repaint areas so that the fill color is combined with the background colors are referred to as soft-till or tint fill algorithms. One use for these fill methods is to soften the fill colors at object borders that have been blurred to antialias the edges.



**Figure 4-24**  
Combining a fill pattern with a background pattern using Boolean operations, *and*, *or*, and *xor* (exclusive or), and using simple replacement.

# Character attributers:

- The appearance of displayed characters is controlled by attributes such as font, size, color, and orientation. Attributes can be set both for entire character strings (text) and for individual characters defined as marker symbols.
- **Text Attributes:**
- There are a great many text options that can be made available to graphics programmers. First of all, there is the choice of font (or typeface), which is a set of characters with a particular design style such as New York, Courier, Helvetica, London, 'Times Roman, and various special symbol groups. The characters in a selected font can also be displayed with assorted underlining styles (solid, dotted, double), in boldface, in *italics*. and in outline or shadow styles.

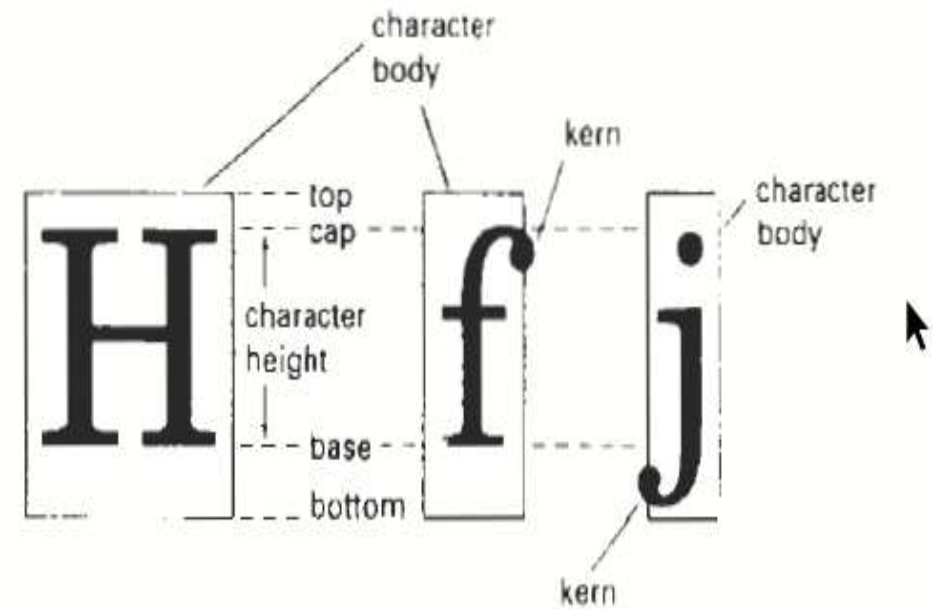


Figure 4-2j  
Character body.

# Character attributers:

Height 1

Height 2

Height 3

*Figure 4-26*

The effect of different character-height settings on displayed text.

width 0.5

width 1.0

**width 2.0**

*Figure 4-27*

The effect of different character-width settings on displayed text.

Spacing 0.0

Spacing 0.5

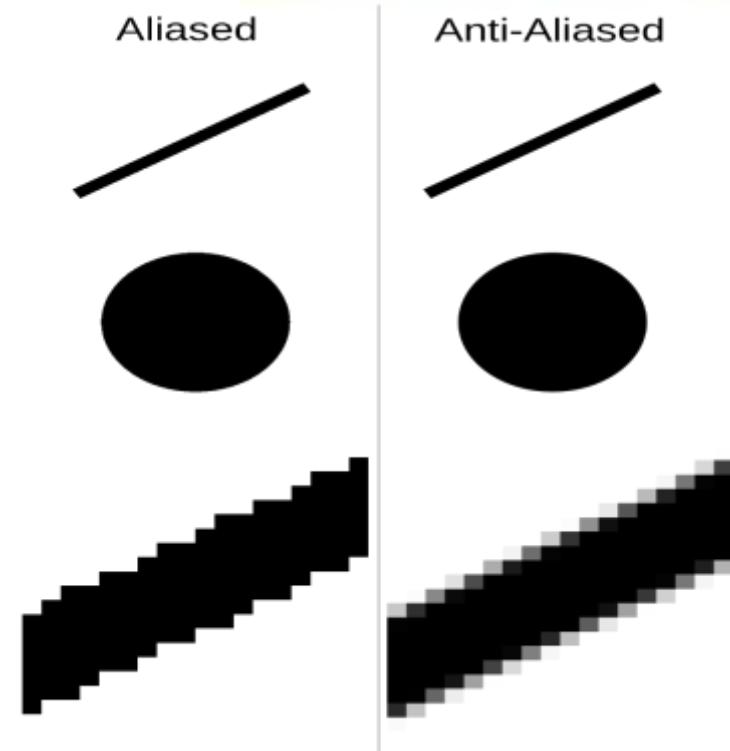
S p a c i n g 1 . 0

*Figure 4-28*

The effect of different character spacings on displayed text.

# Aliasing and Antialiasing

- Displayed primitives generated by the raster algorithms have a jagged, or staircase, appearance because the sampling process digitizes coordinate points on an object to discrete integer pixel positions. This distortion of information due to low-frequency sampling (undersampling) is called aliasing.
- **Antialiasing** is a technique used in computer graphics to remove the aliasing effect. The aliasing effect is the appearance of jagged edges or “jaggies” in a rasterized image.
- Antialiasing methods are followings
  - Using high-resolution display
  - Post filtering (Supersampling)
  - Pre-filtering (Area Sampling)
  - Pixel phasing





# Antialiasing:

- **Using high-resolution display:** One way to reduce aliasing effect and increase sampling rate is to simply display objects at a higher resolution. Using high resolution, the jaggies become so small that they become indistinguishable by the human eye. Hence, jagged edges get blurred out and edges appear smooth.

**Post filtering (Supersampling):** In this method, we are increasing the sampling resolution by treating the screen as if it's made of a much more fine grid, due to which the effective pixel size is reduced. But the screen resolution remains the same. Now, intensity from each subpixel is calculated and average intensity of the pixel is found from the average of intensities of subpixels.

# Antialiasing:

- **Pre-filtering (Area Sampling)**
- In area sampling, pixel intensities are calculated proportional to areas of overlap of each pixel with objects to be displayed. Here pixel color is computed based on the overlap of scene's objects with a pixel area.
- **Pixel phasing**
- It's a technique to remove aliasing. Here pixel positions are shifted to nearly approximate positions near object geometry. Some systems allow the size of individual pixels to be adjusted for distributing intensities which is helpful in pixel phasing.



# Thanks!