Unit 2 – Output Primitives and their Attributes

Shapes and colors of the objects can be described internally with pixel arrays or with sets of basic geometric structures, such as straight line segments and polygon color areas. The scene is then displayed either by loading the pixel arrays into the frame buffer or by scan converting the basic geometric-structure specifications into pixel patterns. Typically, graphics programming packages provide functions to describe a scene in terms of these basic geometric structures, referred to as **output** primitives, and to group sets of output primitives into more complex structures. Each output primitive is specified with input coordinate data and other information about the way that object is to be displayed. Points and straight line segments are the simplest geometric components of pictures. Additional output primitives that can be used to construct a picture include circles and other conic sections, quadric surfaces, spline curves and surfaces, polygon color areas, and character strings.

Line Drawing Algorithms:

Line drawing is achieved by calculating intermediate points along a line path between two specified end point positions. These intermediate points are calculated from the equation of the line.
The ***Slope-Intercept Equation*** of line is:

$$y = mx + b$$

where m represents the slope of the line and b represents the intercept at y-axis.If the two endpoints of the line are (x1, y1) and (x2, y2), the slope and intercept is calculated as:

$$m = (y2 - y1)/(x2 - x1)$$
$$\text{and} \quad b = y1 - m.x1$$

Algorithms for displaying straight lines are based on these equations and calculations. For any given x interval Δx, we can calculate corresponding y interval Δy as

$$\Delta y = m. \Delta x$$

Similarly, we can calculate Δx corresponding to a specified Δy as

$$\Delta x = \Delta y/m$$

For lines with slope value $|m| < 1$, x is increased and y calculated, whereas for slopes $|m| > 1$, y is increased and x calculated.

DDA(Digital Differential Analyser) Line Drawing Algorithm

The DDA algorithm is a line algorithm based on calculating either Δx or Δy.
We know the ***Slope-Intercept Equation*** of line is:

$$y = mx + c$$
$$m = \Delta y / \Delta x \qquad \text{i.e., } m = (y2 - y1)/(x2 - x1)$$

<u>Case 1:</u> Slope is positive and less than one:
Consider the line with positive slope. If the slope value is less than or equal to 1 then $\Delta x = 1$ and we have to calculate Δy value.

$$\Delta y = m.\Delta x$$
$$\therefore \Delta y = m \quad (\text{since } \Delta x = 1)$$
$$\therefore y_{k+1} - y_k = m$$
$$\therefore y_{k+1} = y_k + m \quad (\text{Eq.1})$$

Here 'k' is integer value that starts from 1 and always increases by 1. Here 'm' is float value so we have to round off the 'y' value to nearest integer value.

<u>Case 2:</u> Slope is positive and greater than one:
For slope greater than 1 then $\Delta y = 1$ and we have to calculate Δx value.

$$\Delta y = m.\Delta x$$
$$\therefore \Delta x = \Delta y / m$$
$$\therefore \Delta x = 1 / m \quad (\text{since } \Delta y = 1)$$
$$\therefore x_{k+1} - x_k = 1 / m$$
$$\therefore x_{k+1} = x_k + (1 / m) \quad (\text{Eq. 2})$$

For negative slope:
For negative slope the above same cases are possible but depends of the value of Δx or Δy we have to either increase or decrease their values.

<u>Summary:</u>

In short, for $| m | < 1$ we have to increase or decrease the Δx value by 1 depends on the value of Δx. If Δx is positive then we have to increase its and if Δx is negative then we have to decrease its value. Then we have to calculate the value of Δy based on (Eq. 1). If Δy is positive then y value is incremented by 'm' and if Δy is negative then y is decremented by 'm'.

Similar case is possible for $|m| > 1$ but here the action of x and y is interchanged.

<u>Algorithm:</u>

```
1. Read (x1, y1) and (x2, y2)(First and second coordinate of line)
2. dx = x2 – x1;
   dy = y2 – y1;
3. If absolute(dx) > absolute(dy) then
        s = absolute(dx)
   Else
        s = absolute(dy)
   End if
4. xinc = dx / s
   yinc = dy / s
5. x = x1;
   y = y1;
```

```
6. setpixel(round(x), round(y), color)
7. For i = 1 to s
     x = x + xinc
     y = y + yinc
     setpixel(round(x), round(y), color)
   End for
8. Exit
```

Disadvantages:

In DDA line drawing algorithm we have to perform rounding of the float value and this rounding may cause the line to change from original pixel position. Also this rounding requires more time. Along with the rounding operation it also involves the floating-point arithmetic operation, which also requires more time.

Bresenham's Line Drawing Method

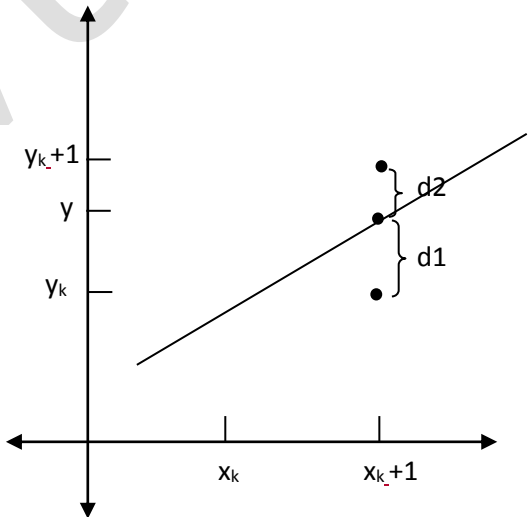
To remove the drawbacks of DDA algorithm Bresenham has given new line drawing algorithm based on integer increment/decrement in x and y value.

Consider the line having positive slope and slope value is less than 1. First the pixel is plotted at co-ordinates (x_k, y_k). Then the x value is always increased by 1 and find out whether the y value is incremented by 1 or not. So the next pixel position is either (x_k+1, y_k) or (x_k+1, y_k+1).

From the line equation we get the original y value at $x_k + 1$ as under:

$$y = m (x_k + 1) + b \qquad (3.1)$$

As shown in figure, the distance between y_k and y is d1 and distance between $y_k + 1$ and y is d2.



Bresenham Line Drawing method

$$\therefore d1 = y - y_k$$
$$\therefore d1 = m (x_k + 1) + b - y_k \qquad \text{(From eq. (3.1))}$$

and $d2 = (y_k + 1) - y$

$\therefore d2 = (y_k + 1) - m(x_k + 1) - b$

$\therefore d1 - d2 = m(x_k + 1) + b - y_k - [y_k + 1) - m(x_k + 1) - b]$

$\therefore d1 - d2 = 2m(x_k + 1) - 2y_k + 2b - 1$

$\therefore d1 - d2 = [2\Delta y(x_k + 1)/\Delta x] - 2y_k + 2b - 1$

$\therefore \Delta x(d1-d2) = 2\Delta y(x_k + 1) - 2\Delta x\, y_k + 2\Delta x\, b - \Delta x$

$\therefore \Delta x(d1-d2) = 2\Delta y\, x_k + 2\Delta y - 2\Delta x\, y_k + 2\Delta x\, b - \Delta x$

$\therefore \Delta x(d1-d2) = 2\Delta y\, x_k - 2\Delta x\, y_k + 2\Delta x\, b - \Delta x + 2\Delta y$ (3.2)

In eq. (3.2) last three terms are constant so we can put some constant b' in place of them. Also we put $\Delta x(d1-d2) = p_k$ as decision parameter.

$\therefore p_k = 2\Delta y\, x_k - 2\Delta x\, y_k + b'$ (3.3)

As like eq. (3.3) we can find the value p_{k+1} as under:

$\therefore p_{k+1} = 2\Delta y\, x_{k+1} - 2\Delta x\, y_{k+1} + b'$ (3.4)

By subtracting eq. (3.3) from (3.4) we get:

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

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

$\therefore p_{k+1} = p_k + 2\Delta y - 2\Delta x(y_{k+1} - y_k)$ (Since $x_{k+1} = x_k+1$) (3.5)

Here from eq. (3.5) we can take the decision whether we have to increase y value or not. If decision parameter p is < 0 then d1 < d2 that means y is more closer to y_k so we have to plot (x_k, y_k) and if decision parameter p is > 0 then d1 > d2 that means y is more closer to $(y_k + 1)$ so we have to plot $(x_k, y_k + 1)$.

Here we are calculating the decision parameter value from previous value but for initial point we can determine decision parameter by:

$\therefore p_0 = 2\Delta y - \Delta x$ (3.6)

Here we have only considered the case of positive slope with less than or equal to 1. Similarly we can derive the equation for the slope greater than 1 but for that purpose the action of x and y is interchanged.

Algorithm of Bresenham for |m| < 1:

1. Read (x1, y1) and (x2, y2) (First and second coordinate of line)
2. dx = absolute(x2 – x1)
   dy = absolute(y2 – y1)
3. p = 2 * dy – dx

4. if x1 > x2 then
   x_{start} = x2
   y_{start} = y2
   x_{end} = x1
   y_{end} = y1
   else
   x_{start} = x1
   y_{start} = y1
   x_{end} = x2
   y_{end} = y2
   end if
5. setpixel(x_{start}, y_{start}, color)
6. Repeat steps 7 to 9 for i = x_{start} to x_{end}
7. x_{start} = x_{start} + 1
8. if p < 0 then
   p = p+ 2 * dy
   else
   if (((y2-y1)/(x2-x1)) > 1) then
   p = p+ 2*(dy – dx)
   y_{start} = y_{start} + 1
   else
   p = p+ 2*(dy – dx)
   y_{start} = y_{start} – 1
   end if
   endif
9. setpixel(x_{start}, y_{start}, color)
10. Exit

Circle Drawing Algorithms:

A circle is defined as the set of points that are all at a distance 'r', called the radius, from a centre position (x_c, y_c).

Midpoint Circle Drawing Method:

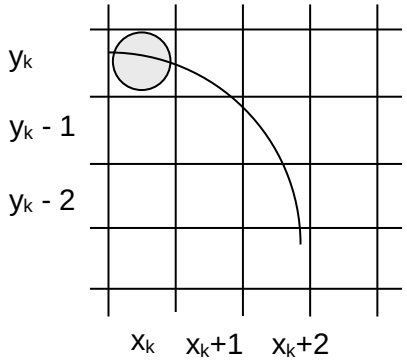
For midpoint method we are using circle function:

$$f(x,y) = x^2 + y^2 - r^2 \quad (1)$$

For any point (x,y) we have following three possibilities

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

So here we are using circle function as decision parameter.



Suppose we have plot the first point at (x_k, y_k) coordinate. Now we have to decide whether the next coordinate is $(x_k + 1, y_k)$ or $(x_k + 1, y_k - 1)$. For that we are using decision parameter at midpoint of y_k & $y_k - 1$.

$$\begin{aligned} p_k &= f(x_k + 1, y_k - ½) \\ &= (x_k + 1)^2 + (y_k - ½)^2 - r^2 \\ &= (x_k + 1)^2 + y_k^2 - y_k + ¼ - r^2 \end{aligned} \quad (2)$$

If $p_k < 0$ then the midpoint is inside the circle so y_k is closer to the circle boundary so our point is $(x_k + 1, y_k)$ otherwise the point is $(x_k + 1, y_k - 1)$. Similarly we have next decision parameter is p_{k+1}.

$$\begin{aligned} p_{k+1} &= f(x_{k+1} + 1, y_{k+1} - ½) \\ &= ((x_k + 1) + 1)^2 + (y_{k+1} - ½)^2 - r^2 \\ &= (x_k + 1)^2 + 2(x_k + 1) + 1 + y_{k+1}^2 - y_{k+1} + ¼ - r^2 \end{aligned} \quad (3)$$

$$\begin{aligned} p_{k+1} - p_k &= 2(x_k + 1) + 1 + y_{k+1}^2 - y_{k+1} - (y_k^2 - y_k) \\ &= 2(x_k + 1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1 \end{aligned}$$

$$p_{k+1} = p_k + 2(x_k + 1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1 \quad (4)$$

If $p_k < 0$ then

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

$$\begin{aligned} p_{k+1} &= p_k + 2(x_k + 1) + (y_k^2 - y_k^2) - (y_k - y_k) + 1 \\ p_{k+1} &= p_k + 2x_{k+1} + 1 \end{aligned} \quad (5)$$

else

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

$$\begin{aligned} p_{k+1} &= p_k + 2(x_k + 1) + ((y_k - 1)^2 - y_k^2) - ((y_k - 1) - y_k) + 1 \\ &= p_k + 2x_{k+1} + (-2y_k + 2) + 1 \\ &= p_k + 2x_{k+1} - 2(y_k - 1) + 1 \\ p_{k+1} &= p_k + 2x_{k+1} - 2y_{k+1} + 1 \end{aligned} \quad (6)$$

Derivation of initial parameter p_0:

Here we have initial coordinate is (0, r) with respect to center coordinate (0,0). Next point is either (1, r) or (1, r-1). So decision parameter

$$\begin{aligned} p_0 &= f(1, r - ½) \\ &= 1 + (r - ½)^2 - r^2 \\ &= 1 + r^2 - r + ¼ - r^2 \\ &= 1 - r + ¼ \\ &= \mathbf{5/4 - r} \end{aligned} \quad (7)$$

Algorithm:

1. Read centre coordinate (x_c, y_c) and radius r
2. x = 0
   y = r
3. p = 1 – radius
4. Calculate the symmetry points for all the eight octants and add the centre coordinate (x_c, y_c) to each calculated point. Then plot that point.
5. Repeat steps 6 to 8 while x < y
6. x = x + 1
7. if p < 0 then
   p = p + 2 * x + 1
   else
   y = y –1
   p = p + 2 * (x – y) + 1
   end if
8. Calculate the symmetry points for all the eight octants and add the centre coordinate (x_c, y_c) to each calculated point. Then plot that point.
9. Exit

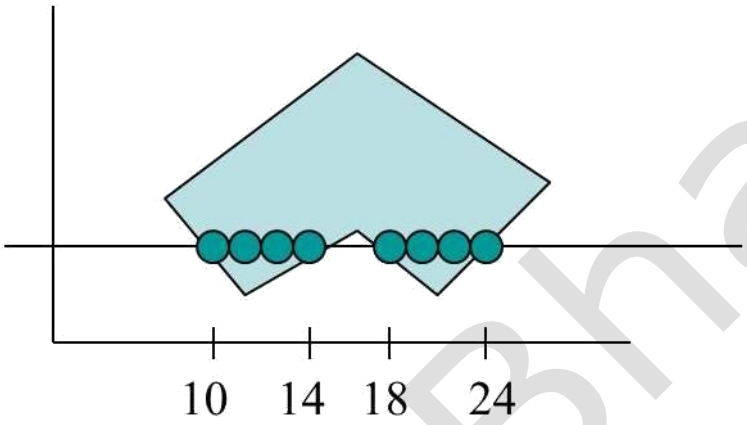
Filled Area Primitives

A standard output primitive in general graphics packages is a solid-color or patterned polygon area. Other kinds of area primitives are sometimes available, but polygons are easier to process since they have linear boundaries

There are two basic approaches to area filling on raster systems. One way to fill an area is to determine the overlap intervals for scan lines that cross the area. Another method for area filling is to start from a given interior position and paint outward from this point until we encounter the specified boundary conditions. The scan-line approach is typically used in general graphics packages to fill polygons, circles, ellipses, and other simple curves. Fill methods starting from an interior point are useful with more complex boundaries and in interactive painting systems

Scan Line Polygon Fill Algorithm

A standard output primitive in general graphics package is a solid color or patterned polygon area. There are two basic approaches to filling on raster systems. Determine overlap Intervals for scan lines that cross that area. Start from a given interior point and paint outward from this point until we encounter the boundary. The first approach is mostly used in general graphics packages, however second approach is used in applications having complex boundaries and interactive painting systems.



For each scan line crossing a polygon is then sorted from left to right, and the corresponding frame buffer positions between each intersection pair are set to the specified color. These intersection points are then sorted from left to right , and the corresponding frame buffer positions between each intersection pair are set to specified color.

In the given example, four pixel intersections define stretches from x=10 to x=14 and x=18 to x=24

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

In the given example , scan Line ***y*** intersects five polygon edges and the scan Line ***y'*** intersects 4 edges although it also passes through a vertex ***y'*** correctly identifies internal pixel spans ,but need some extra processing

The topological difference between scan line y and scan line y' is identified by noting the position of the intersecting edges relative to the scan line. For Scan line ***y***, the two intersecting edges sharing a vertex are on opposite sides of the scan line. But for scan line ***y'***, the two intersecting edges are both above the scan line. Thus, the vertices that require additional processing are those that have connecting edges on opposite sides of scan line.

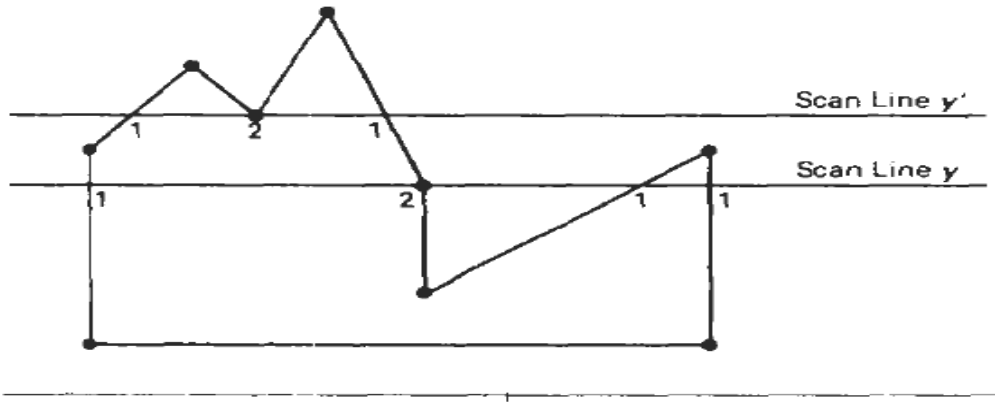


Figure 3-36
Intersection points along scan lines that intersect polygon vertices. Scan line *y* generates an odd number of intersections, but scan line *y'* generates an even number of intersections that can be paired to identify correctly the interior pixel spans.

We can identify these vertices by tracing around the polygon boundary either in clock-wise or anti-clockwise order and observing the relative changes in vertex ***y*** coordinates as we move from one edge to the next.

If the endpoint ***y*** values of two consecutive edges monotonically increase or decrease, we need to count the middle vertex as a single intersection point for any scan line passing through that vertex.

Otherwise, the shared vertex represents a local extremum (min. or max.) on the polygon boundary, and the two edge intersections with the scan line passing through that vertex can be added to the intersection list.

One way to resolve this is also to shorten some polygon edges to split those vertices that should be counted as one intersection. Non horizontal edges around the polygon boundary in the order specified, either clockwise or anti-clockwise. When the end point y coordinates of the two edges are increasing, the y value of the upper endpoint for the current edge is decreased by 1. When the endpoint y values are monotonically decreasing, we decrease the y coordinate of the upper endpoint of the edge following the current edge.

Algorithm:

1. Set y to the smallest y coordinate that has an entry in the ET; i.e, y for the first nonempty bucket.
2. Initialize the AET to be empty.
3. Repeat until the AET and ET are empty:
   3.1 Move from ET bucket y to the AET those edges whose y_min = y (entering edges).

3.2 Remove from the AET those entries for which y = y_max (edges not involved in the next scanline), the sort the AET on x (made easier because ET is presorted).

3.3 Fill in desired pixel values on scanline y by using pairs of x coordinates from AET.

3.4 Increment y by 1 (to the coordinate of the next scanline).

3.5 For each nonvertical edge remaining in the AET, update x for the new y.

4. Exit

Inside Outside Tests

Area-filling algorithms and other graphics processes need to find inside regions of objects. Graphics packages normally use either the odd-even rule or the nonzero winding number to identify interior regions of an object.

1. Odd-Even Method

In Odd-Even Method, we conceptually draw a line from a position P to a point outside the region of the object and counting the number of edges of the object crossing the line. We can find a point outside a polygon, for example, by picking a point with the x-coordinate smaller than the smallest x-coordinate of the polygon's vertices.

If the number of polygon edges crossed by this line is odd, then the point P is inside the region. Otherwise, P is an exterior point. To get an accurate edge count, the line path should not cross any endpoint of the polygon.

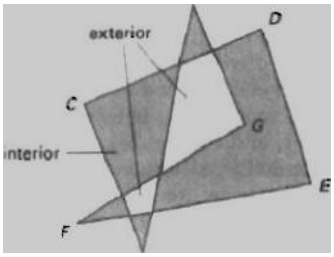
Non-zero Winding Number Method

Another method for defining interior regions is the nonzero winding number method. In this method also we start be imagining a line drawn from any position P to a point beyond the coordinate region of the object. The line we choose should not pass through any vertices (end points). We count the number of edges that cross the line in each direction. We add 1 to the winding number every time we intersect a polygon edge that crosses the line from right to left or from above the line to below the line and we subtract 1 every time we intersect an edge that crosses the line from left to right or from below the line to above the line.

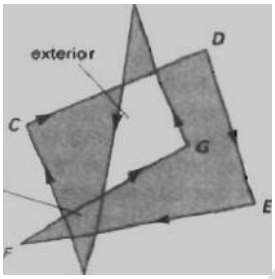
The final value of the winding number, after the entire edge crossings have been counted, determines the position of P. If the winding number

is nonzero then the point is inside the region. Otherwise, if winding number is zero, the point is outside the region.

For standard polygons and other simple shapes, the nonzero winding number rule and odd-even rule give the same results. But for complicate set-intersecting polygons, the two methods may give different results, as given below.



Odd even rule



Non zero winding number rule

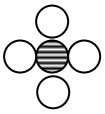
Boundary Fill Algorithm

One approach to filling an area is to start at a point inside a region and fill interior outwards pixel by pixel towards the boundary until boundary colour is encountered. This method is known as the ***Boundary Fill Method***.

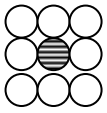
This method is generally used in interactive painting packages, where interior points are easily selected. A boundary fill procedure accepts three parameters - a point inside the region (x, y), the boundary colour and the filling colour. Starting from (x, y), its neighbouring positions are tested to determine whether they are of the boundary colour. If not, they are painted with fill colour, and their neighbouring points are tested. This process continues until all pixels up to the boundary colour for the area have been tested.

There are two methods for going to neighbouring pixels from the current position. For the given point, its neighbouring four points, that is, left, right, top and bottom, are tested. This method is known as ***4-connected***.

Another method to test neighbouring pixel is ***8-connected***. In this the four neighbouring points as well as the diagonal points are tested.



4-connected



8-connected

An 8-connected boundary-fill algorithm can fill more complex regions as compared to 4-connected boundary-fill algorithm.

Algorithm:

The algorithm for boundary fill using recursion is given by 4-connected method is given. (x,y) is the point inside the polygon. 'fillcolour' is the colour by which the polygon is to be filled. 'bordercolour' is the colour of the border.

BoundaryFill (x, y, fillcolour, bordercolour)

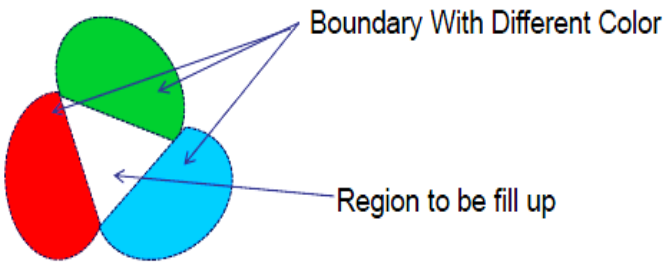
1. colour = getpixelcolour(x,y)
2. if (colour != bordercolour and colour != fillcolour), then
   (a) setpixelcolour (x, y, fillcolour)
   (b) BoundaryFill(x+1, y, fillcolour, bordercolour)
   (c) BoundaryFill(x–1 , y, fillcolour, bordercolour)
   (d) BoundaryFill(x, y+1, fillcolour, bordercolour)
   (e) BoundaryFill(x, y–1 , fillcolour, bordercolour)
3. Return

For 8-connected, call the BoundaryFill() function four more times with the coordinate values of the diagonal points, that is, the right-top (x+1, y+1), the right-bottom (x+1, y-1), the left-top (x-1, y+1) and the left-bottom (x-1, y-1).

Recursive boundary-fill algorithm may not fill regions correctly if some interior pixels are already displayed in fill colour. This occurs because the algorithm checks neighbouring pixels for boundary colour and for fill colour. If there is already a pixel with fill colour, the recursive loop will stop and may leave other interior pixels unfilled.

Flood Fill Method

If we want to fill an area whose boundaries have many different colours, then another approach is followed for filling the polygon. We paint by replacing a specific interior colour with the fill colour. This method is known as ***Flood Fill Method***.



We start from a specific interior point (x, y) and reassign the pixels with current interior colour with the specified fill colour. Using either 4-connected or 8-connected approach, we pass through pixel positions until all interior points have been repainted.

Algorithm:

The algorithm for flood fill using recursion is given by 4-connected method is given. (x,y) is the point inside the polygon. 'fillcolour' is the colour by which the polygon is to be filled. 'oldcolour' is the existing interior colour.

FloodFill (x, y, fillcolour, oldcolour)

1. colour = getpixelcolour(x,y)
2. if (colour == oldcolour), then
   (a) setpixelcolour (x, y, fillcolour)
   (b) FloodFill(x+1, y, fillcolour, oldcolour)
   (c) FloodFill(x–1 , y, fillcolour, oldcolour)
   (d) FloodFill(x, y+1, fillcolour, oldcolour)
   (e) FloodFill(x, y–1 , fillcolour, oldcolour)
3. Return

Character Generation

Letters, numbers and other characters can be displayed in a variety of sizes and styles. The design style for a set of characters is called a ***Typeface*** or ***Font***.

Typefaces or Fonts can be divided into two broad groups: ***Serif*** and ***Sans Serif***. Serif type has small lines at the ends of the main character strokes (called accents), while Sans Serif does not have accents. Serif type is generally more readable, that is, it is easier to read in large amount of texts. On the other hand, the individual characters in sans-serif type are easier to identify. So, Sans-Serif type is said to be more legible and is good for labelling and short heading.

Two different methods are used for storing computer fonts:

Bitmap Method or Dot-Matrix Method: In this, the character shapes in a particular font are represented in form of rectangular grid patterns. The set of characters are then referred to as a Bitmap Font.

Bitmap fonts are simplest to define and display. The character grid only needs to be mapped to a frame-buffer position. But bitmap fonts require more space. It is possible to generate different sizes and other variations, such as bold, italics, etc., but usually the results are not good.

Outline Method or Vector Method or Stroke Method: In this method, the characters are described using straight lines and curve sections. The set of characters is called an Outline Font.

Outline fonts require less storage. Different variation or sizes can be produced by manipulating the curve definitions for character outlines and the results are also good. But it takes more time to process the outline fonts.

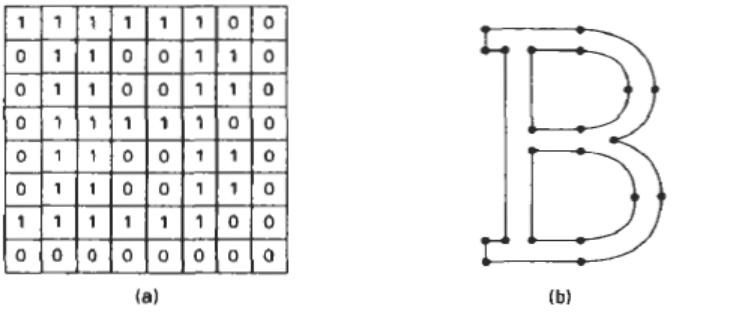


Figure 3-48
The letter B represented in (a) with an 8 by 8 bilevel bitmap pattern and in (b) with an outline shape defined with straight-line and curve segments.

Attributes

The features or characteristics of an output primitive are known as ***Attribute***. In other words, any parameter that affects the way a primitive is to be displayed is known as ***Attribute***. Some attributes, such as colour and size, are basic characteristics of primitive. Some attributes control the basic display properties of primitives. For example, lines can be dotted or dashed, thin or thick. Areas can be filled with one colour or with multiple colours pattern. Text can appear from left to right, slanted or vertical.

Line Attributes:

Basic attributes of a straight line are its type, its width, and its colour. In some graphics packages, line can also be displayed using selected pen or brush options.

1. Line Type: The line type attribute includes solid lines, dashed lines, and dotted lines. We modify the line drawing algorithm to generate such lines by setting the length and space. A dashed line could be displayed by generating spaces that is equal to length of solid part. A dotted line can be displayed by generating very short dashes with spacing equal to or greater than the dash size. Similar methods are used to produce other line-type variations.

Raster line-algorithms displays line type attribute by plotting pixels. For various dashed, dotted patterns, the line-drawing algorithms outputs part of pixels followed by spaces. Plotting dashes with a fixed number of pixels results in unequal-length dashes for different line angles. For example, the length of dash diagonally is more than horizontal dash for same number of pixels. For precision drawings, dash length should

remain approximately same for any line angle. For this, we can adjust the pixel number according to line slope.

2. Line Width: A line with more width can be displayed as parallel lines on a video monitor. In raster lines, a standard width line is generated with single pixels at each point. Width lines are displayed by plotting additional pixels along next parallel line paths. For lines with slope less than 1, we can display thick lines by plotting a vertical length of pixels at each x position along the line. Similarly, for lines with slope greater than 1, we can plot thick lines with horizontal widths for each point.

The problem with implementing width options using horizontal and vertical pixel widths is that the width of line is dependent on the slope. A 45-degree line will be displayed thinner as compared to vertical or horizontal line plotted with same number of pixel widths.

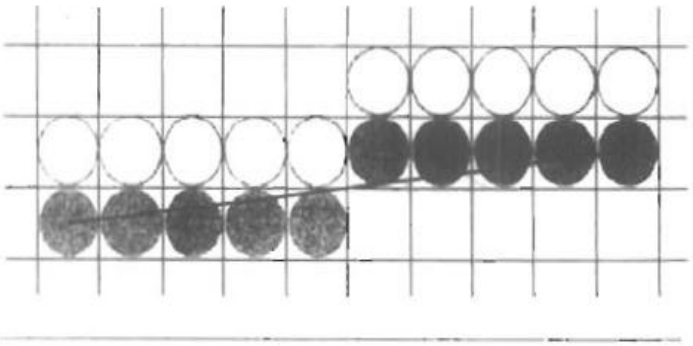


Figure 4-3
Double-wide raster line with slope $|m| < 1$ generated with vertical pixel spans.

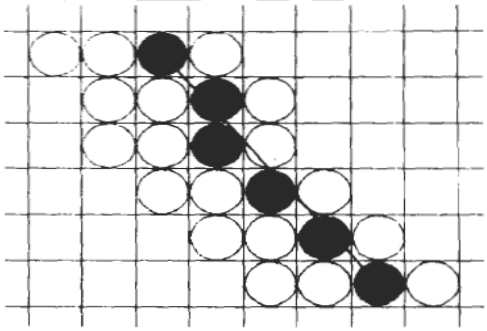
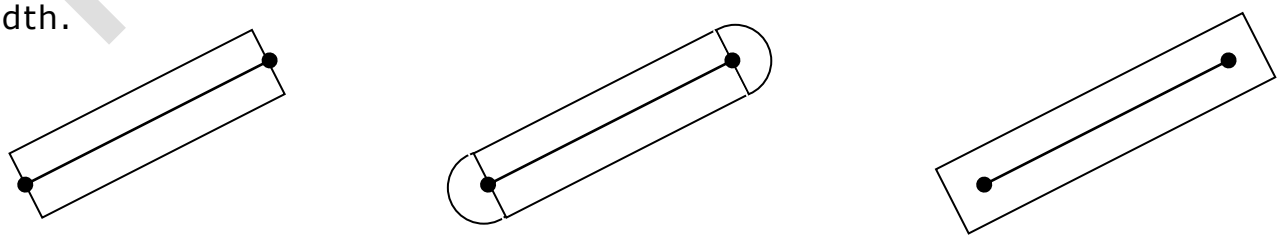


Figure 4-4
Raster line with slope $|m| > 1$ and line-width parameter $lw = 4$ plotted with horizontal pixel spans.

Another problem is that it produces lines whose ends are either horizontal or vertical. We can adjust the shape of the line ends by adding ***Line Caps***.

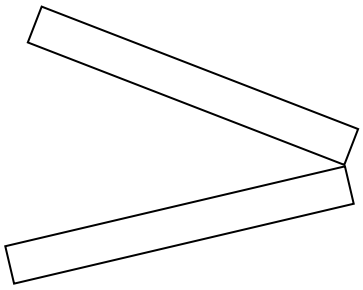
One kind of line cap is the ***Butt Cap.*** It is obtained by adjusting the end positions of lines so that the thick line is displayed with square ends that are perpendicular to the line. Another line cap is the ***Round Cap*** obtained by adding a filled semicircle to each butt cap. The semicircle has diameter equal to thickness of line. The third type of line cap is the ***Projecting Square Cap***. Here, the butt cap is extended to half of line width.



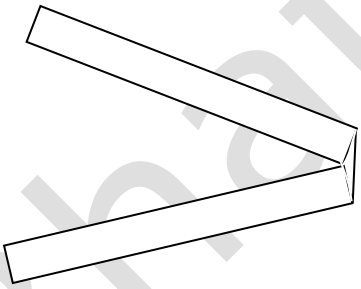
(a) (b) (c)

Thick Lines drawn with (a) Butt Cap (b) Round Cap and (c) Projecting Square Cap

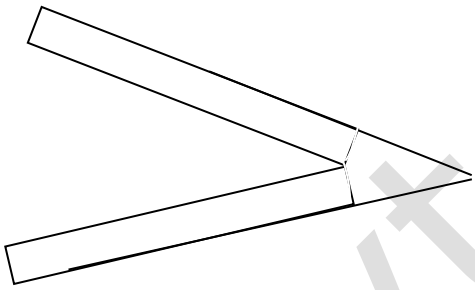
Generating thick connected line segments require other considerations. The methods that we have considered for displaying thick lines will not produce smoothly connected line segments. It leaves gaps at the boundaries between lines of different slope. There are three possible methods for smoothly joining two line segments. A ***Miter Join*** is obtained by extending the outer boundaries of each of the two lines until they meet. A ***Round Join*** is produced by covering the connection between the two segments with a circular boundary whose diameter is equal to the line width. A ***Bevel Join*** is generated by displaying the line with butt caps and filling in the triangular gap where the segments meet.



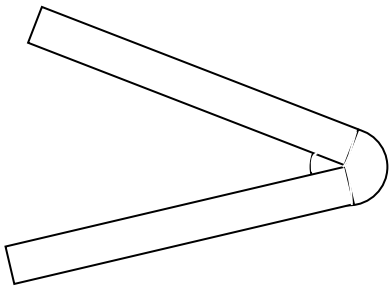
Line without Joins



Bevel Join



Miter Join



Round Join

3. Pen and Brush Options: In some graphic packages, lines can be displayed with pen or brush selection. Options in this category include shape, size and pattern. These shapes are stored in a ***Pixel Mask*** that identifies the pixel positions that are to be set along the line path. Lines generated with pen or brush shaped can be displayed in various widths by changing the size of the mask. Lines can also be displayed with selected patterns.

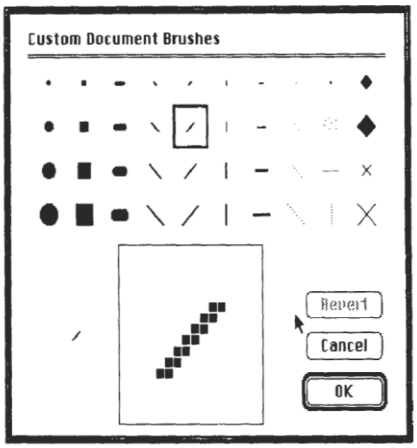


Figure 4-7
Pen and brush shapes for line display.

Figure 4-10
Curved lines drawn with a paint program using various shapes and patterns. From left to right, the brush shapes are square, round, diagonal line, dot pattern, and faded airbrush.

4. Line Colour: A system displays a line in the current colour by setting the colour value in the frame buffer at pixel locations. The number of colour choices depends on the number of bits available per pixel in the frame buffer. A line drawn in the background colour is invisible.

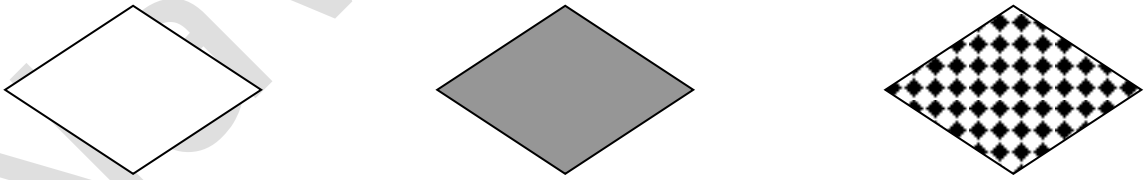
Area-Fill Attributes:

Options for filling a region include a choice between a solid colour and a patterned fill. These options can be applied to polygon regions or regions with curved boundaries.

Areas can be displayed with various fill styles: hollow, solid, pattern and hatch.

Hollow areas are displayed using only the boundary outline, with interior colour the same as the background colour.

A Solid fill is displayed in a single colour including the borders. Fill style can also be with a specific pattern or design. The Hatch fill is used to fill area with hatching pattern.



Hollow Solid Pattern

Polygon fill styles

Diagonal hatch Diagonal Cross-hatch

Polygon fill using hatch patterns

Other fill options include specifications for the edge type, edge width and edge colour. These attributes are same as the line attributes. That is, we can display polygon edges as dotted, dashed, thick or of different colours, etc.

Soft Fill

We may want to fill area again due to 2 reasons:

- It is blurred (unclear) when painted first time, or
- It is repainting of a color area that was originally filled with semitransparent brush, where current color is then mixture of the brush color and the background color " behind" the area.

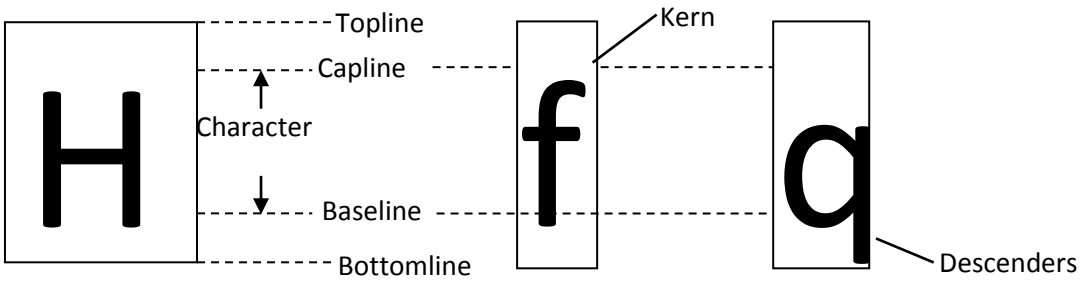
So that the fill color is combined with the background colors are referred to as Soft-fill.

Character Attributes:

The appearance of displayed characters is controlled by attributes such as font, size, colour and orientation. Attributes can be set both for entire character strings and for individual characters, known as Marker symbols.

1. Text Attributes: There are many text options available, such as font, colour, size, spacing, and orientation.

- **Text Style:** The characters in a selected font can also be displayed in various underlining styles (solid, dotted, dashed, double), in **bold**, in *italics*, shadow style, etc. Font options can be made available as predefined sets of grid patterns or as character sets designed with lines and curves.
- **Text Colour:** Colour settings for displayed text are stored in the system attribute list and transferred to the frame buffer by character loading functions. When a character string is displayed, the current colour is used to set pixel values in the frame buffer corresponding to the character shapes and position.
- **Text Size:** We can adjust text size by changing the overall dimensions, i.e., width and height, of characters or by changing only the width. Character size is specified in ***Points***, where 1 point is 0.013837 inch or approximately 1/72 inch. Point measurements specify the size of the ***Character Body***. Different fonts with the same point specifications can have different character sizes depending upon the design of the font.



Character Body

The distance between ***Topline*** and ***Bottomline*** is same for all characters in a particular size and font, but the width may be different. A smaller body width is assigned to narrow characters such as i, j, l, etc. compared to broad characters such as W or M. The ***Character Height*** is the distance between the ***Baseline*** and ***Capline*** of characters. Kerned characters, such as f and j, extend beyond the character-width limits. And letters with descenders, such as g, j, p, q, extend below the baseline.

The size can be changed in such a way so that the width and spacing of characters is adjusted to maintain the same text proportions. For example, doubling the height also doubles the character width and the spacing between characters. Also, only the width of the characters can be changes without affecting its height. Similarly, spacing between characters can be increased without changing height or width of individual characters.

The effect of different character-height setting, character-width setting, and character spacing on a text is shown below.

HEIGHT1	WIDTH1	SPACING1
HEIGHT2	**WIDTH2**	**SPACING2**
HEIGHT3	***WIDTH3***	***SPACING3***

Effect of changing Height, Width and Spacing

- **Text Orientation:** The text can be displayed at various angles, known as orientation. A procedure for orienting text rotates characters so that the sides of character bodies, from baseline to topline at aligned at some angle. Character strings can be arranged vertically or horizontally.

A text orientated by 45 degrees in anticlockwise and clockwise direction

- **Text Path:** In some applications the character strings are arranged vertically or horizontally. This is known as Text Path. Text path can be right, left, up or down.

g
n
i
r
t
s
gnirts String
s
t
r
i
n
g

A text displayed with the four text-path options

- **Text Alignment:** Another attribute for character strings is alignment. This attribute specifies how text is to be positioned with respect to the start coordinates. Vertical alignment can be top, cap, half, base and bottom. Similarly, horizontal alignment can be left, centre and right.

Alignment values for a string

Compiled By: Mr. Navtej Bhatt, Assistant Professor,
BCA Department, V.P. & R.P.T.P. Science College, VV Nagar
Class: BCA SEM V
Subject: US05CBCA02 – Computer Graphics

Declaration: This material is developed only for the reference for lectures in the classrooms. Students are required library reading for more study. This study material compiled from Book "Computer Graphics by Donald Hearn & M. Pauline Baker, PHI, 1995