FILLED-AREA PRIMITIVES

Objects can be filled with solid-color or patterned polygon area in some graphical packages.

Standard output primitive in graphics packages is solid color ,patterned polygon area

Polygons are easier to process due to linear boundaries

Two basic approaches to area filling on a raster system

Determine the overlap intervals **for scan lines that cross the area. Typically useful for filling polygons, circles, ellipses**

Start from a given interior position **and paint outwards from this point until we encounter the specified boundary conditions useful for filling more complex boundaries, interactive painting system.**

Two things to consider

i. which pixels **to fill** **ii. with** what value **to fill**

Move along scan line (from left to right) that intersect the primitive and fill in pixels that lay inside

To fill rectangle with solid color

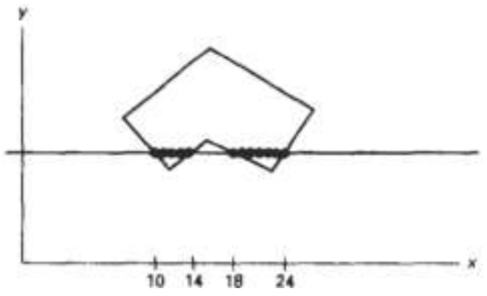
Set each pixel lying on scan line running from left edge to right with same pixel value, each span from x $_{max}$ to x $_{min}$

```
for( y  from y min    to y max  of rectangle)                 /*scan line*/
    for( x  from x min    to x max of rectangle)             /*by pixel*/
        writePixel(x, y, value);
```

Scan line Polygon Fill Algorithm

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.

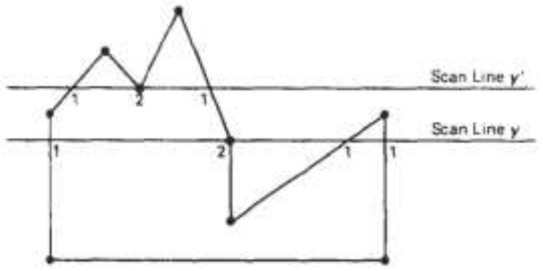


Four pixel intersection positions with the polygon boundaries define two stretches of interior pixels from x = 10 to x = 14 and from x = 18 to x = 24.

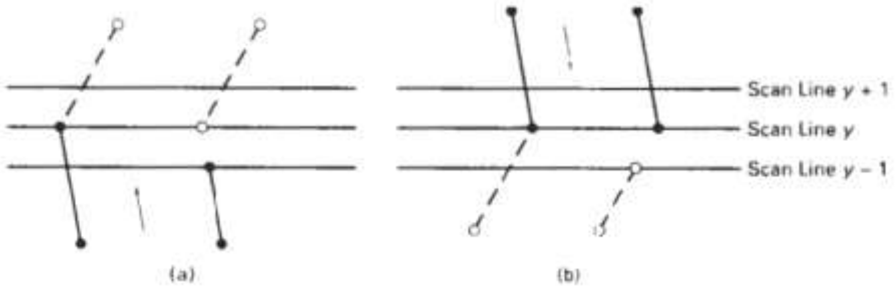
For a scan line passing through a vertex with two edges intersecting at that position, adding two points to the list of intersections for the scan line gives a solution.

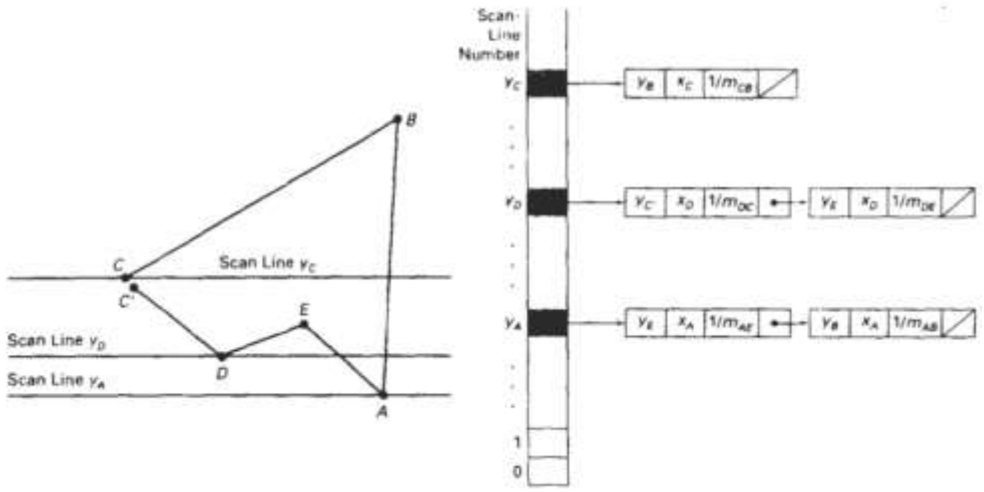
Scan line y intersects five polygon edges. Scan line y', however, intersects an even number of edges although it also passes through a vertex.

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



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




Scan-Line Number
y_C
y_D
y_A
1
0
y_B | x_C | $1/m_{CB}$
y_C | x_D | $1/m_{DC}$
y_E | x_D | $1/m_{DE}$
y_E | x_A | $1/m_{AE}$
y_B | x_A | $1/m_{AB}$
B
C
C'
E
D
A
Scan Line y_C
Scan Line y_D
Scan Line y_A

Boundary-Fill Algorithm

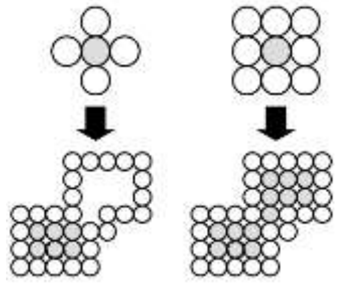
Starts at a point inside a region and paint the interior outward toward the boundary.

If the **boundary is specified in a single color**, the fill algorithm proceeds outward pixel by pixel until the boundary color is encountered.

This approach requires boundary to be specified, but interior pixels can be of multiple colors

Four connected approach considers neighboring pixels as the ones that are to the right, left, bottom and top of the seed pixel

Eight connected approach also considers diagonal pixels as neighboring pixels so it includes pixels to the right, left, bottom , top then four more diagonal ones right-top, right-bottom, left-top , left-bottom



Algorithm:

```
Step 1. Start at a point inside a region and paint the interior outward toward the
        boundary. Accepts as input coordinates of an interior point (x, y) a fill color
        and boundary color.
Step 2. Starting from (x, y) the procedure tests neighboring position to determine
        whether they are of the boundary color and if they are already filled as well
Step 3. If not they are painted with fill color and their neighbors are tested
        Recursively using 4 Connected or 8 Connected approach
Step 4. Process continues until all pixels up to boundary color for the area have been
        tested
```

Pseudocode

```
void boundaryFill (int x, int y, int fill, int boundary) (
int  current = getpixel (x, y);
       if ((current != boundary) && (current != fill)) {
              setcolor (fill) ;
              setpixel (x, y);
              boundaryfill (x+1, y, fill, boundary);
              boundaryFill (x-1, y, fill, boundary);
              boundaryFill (x, y+1, fill, boundary);
              boundaryFill (x, y-1, fill, boundary) ;
       }
}
```

Flood-Fill Algorithm

Used for filling an area that is not defined within a single color boundary.

Such areas can be filled by replacing a specified interior color instead of searching for a boundary color value.

This approach does not require boundary to be specified, but interior pixels must be of the same color

Algorithm:

```
Step 1. Start from specified interior point (x,y) and reassign all pixel values that are
        currently set  to a given interior color with the desired color.

Step 2. If the area we want to paint has more than 1 interior color, first assign pixel
        values so that all interior points have same color.

Step 3. We can then use 8 or 4 connected approach to move on recursively until all
        interior points have been repainted.
```

Pseudocode

```
void floodFill (int x, int y, int fillcolor, int oldcolor) {
       if (getpixel (x, y) == oldcolor) {
              setcolor (fillcolor);
              setpixel (x, y);
              floodFill (x+1, y, fillColor, oldColor);
              floodfill (x-1, y, fillcolor, oldcolor);
              floodFill (x, y+1, fillcolor, oldcolor);
              floodFill (x, y-1, fillColor, oldcolor);
       }
}
```