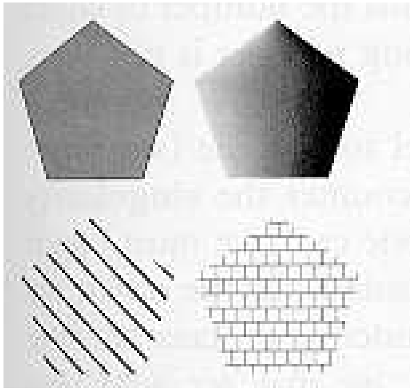
# Polygon Filling Algorithms

- **Two fundamental questions**

  - Which pixels to fill

  - With what value to fill them

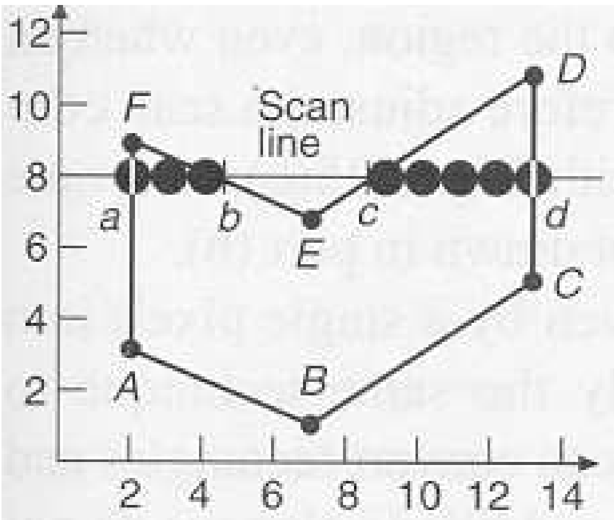


- **Two basic approaches**

  - Scan-line

    * Take successive scan lines that cross the area and fill in the spans of adjacent pixels that lie inside the area from left to right

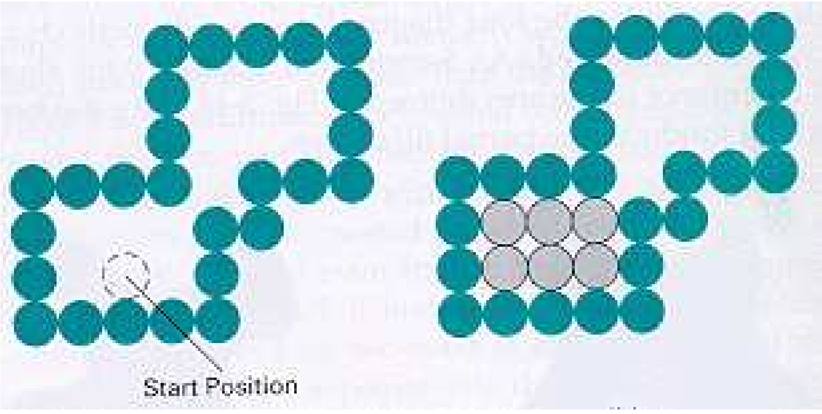
    * Typically used to fill polygons, circles, and other simple curves



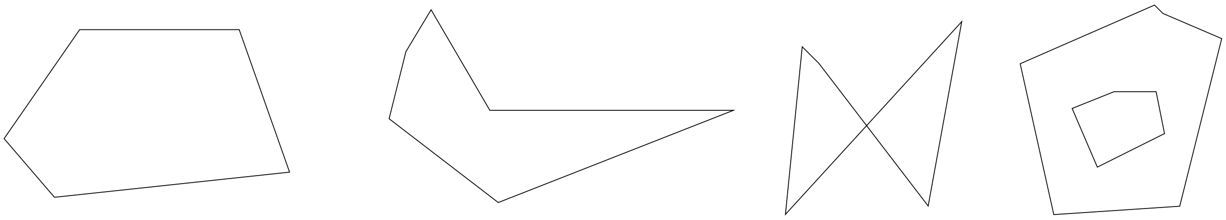
- Seed-fill (or boundary fill)

* Start from a given interior position and paint outward from this position until we encounter the boundary

* Typically used to fill more complex boundaries

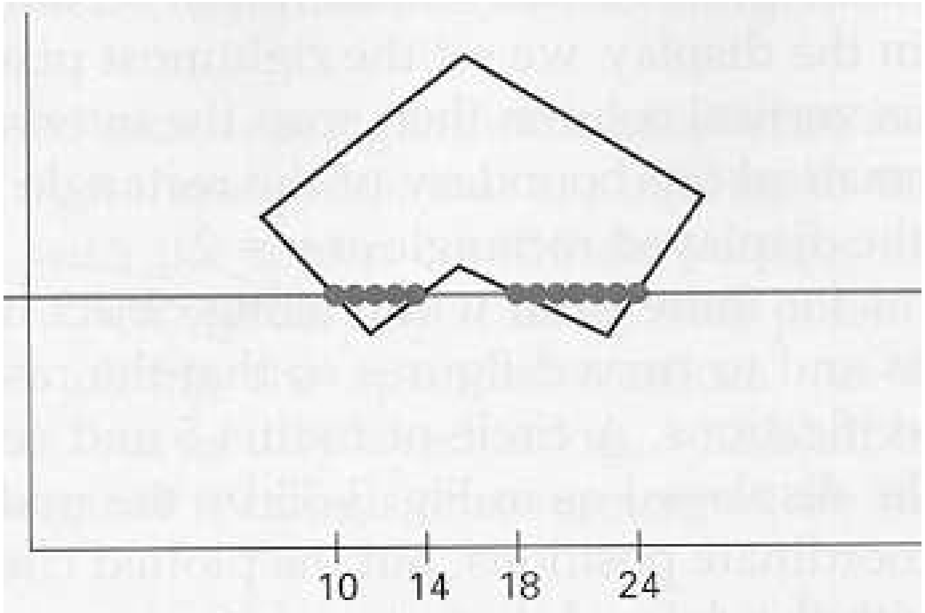



- **Different types of polygons**



simple convex    simple concave    non-simple    with holes

Scan line polygon filling




- **Algorithm**

  (i) Find the intersections of the scan line with all the edges of the polygon

  (ii) Sort the intersections by increasing x coordinate

  (iii) Fill in all the pixels between pairs of intersections that lie inside the polygon

- Odd-parity rule can be used to determine if a point is inside a region:

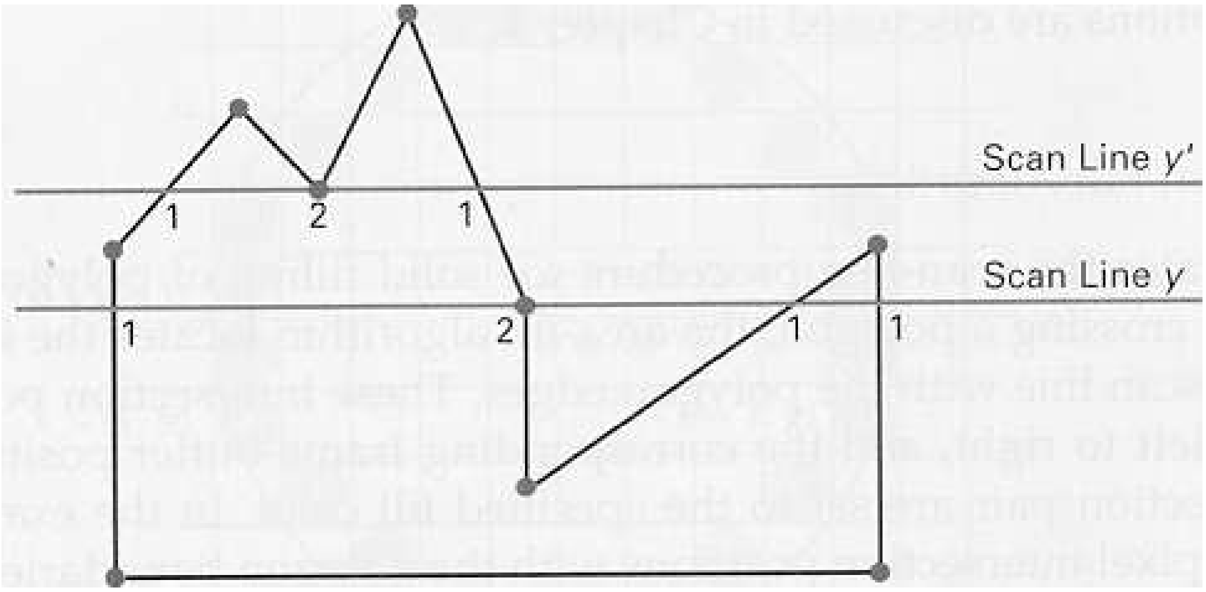
  1) parity = even (initialization)

  2) for each intersection, invert parity

  (odd-parity -> draw, even-parity -> do not draw)

- **Scan-line intersections at polygon vertices**

  - A scan-line passing through a vertex intersects two edges at that position




- **What is the topological difference between $y$ and $y'$ ?**

  - scan-line $y$ (*case A*):

    * The two edges sharing the vertex are on opposite sides of the scan-line

    * The $y$-coordinate of $P_2$ is $y_{min}$ for $e_3$ but $y_{max}$ for $e_4$ (alternatively: $y_{max}$ and $y_{min}$)

  - scan-line $y'$ (*case B*):

    * The two edges are above the scan-line

    * The $y$-coordinate of $P_1$ is $y_{min}$ for both $e_1$ and $e_2$ (alternatively: $y_{max}$)

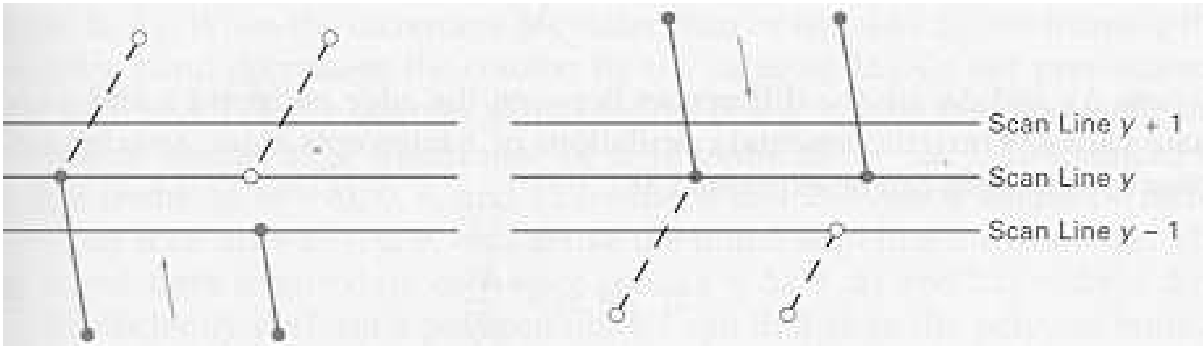
- **Rule**

  - In case A: count the vertex as a single intersection

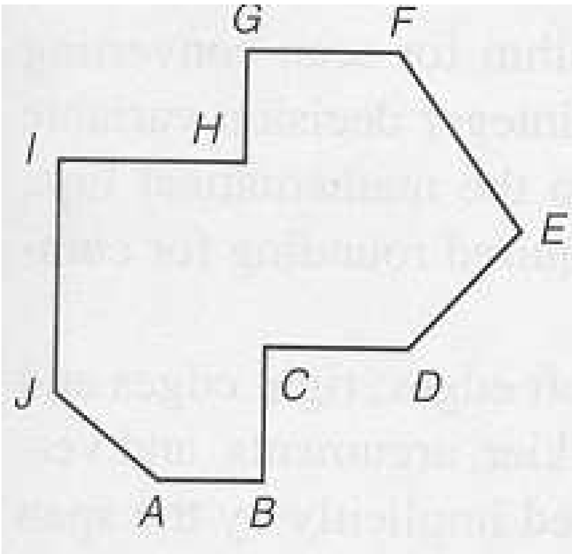
  - In case B: count the vertex as two intersections

- **Shortening an edge**

  - To avoid cases where a vertex needs to be counted as one intersection, shorten one of the two intersecting edges by one scan line




- **Horizontal edges**




  - Do not count their vertices

  - Invert parity bit if $y_{\text{min}}$

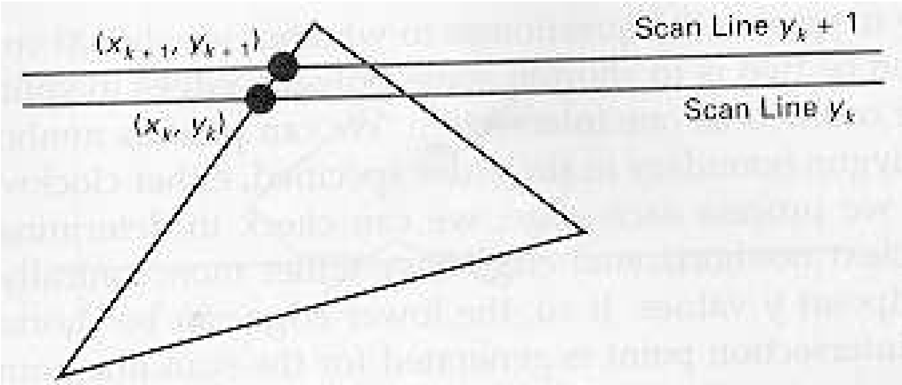
  - Do not invert parity bit if $y_{\text{max}}$

  - Pixels are ommited

- **Determining edge intersections**

  - brute force approach: test each polygon edge for intersection with each new scan line

  - efficient approach: take advantage of *edge coherence*



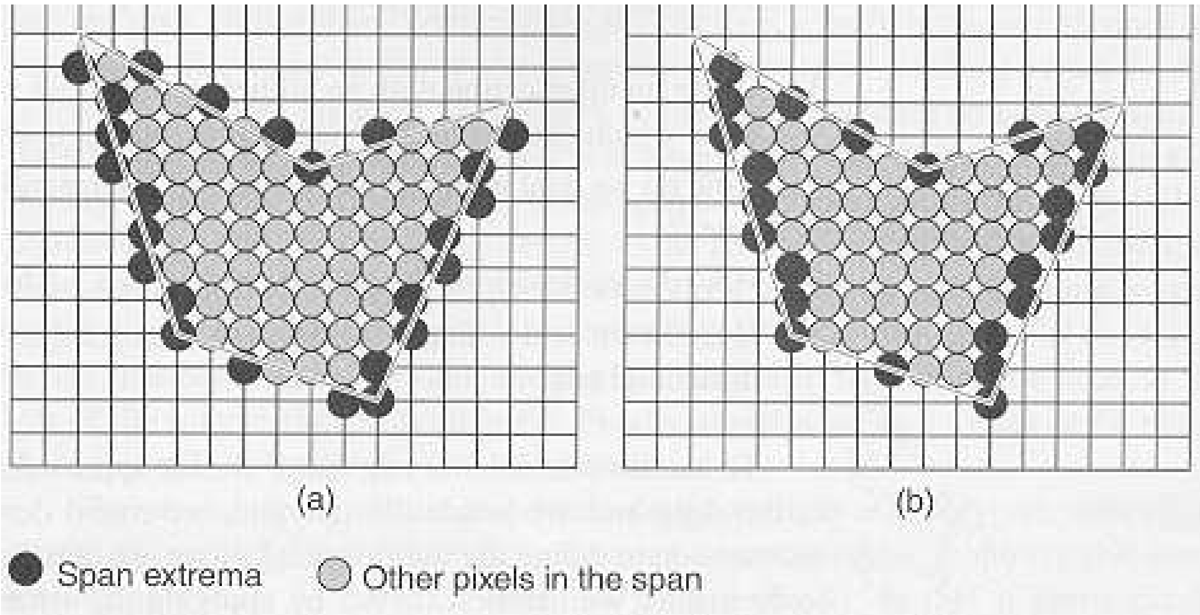

  - hint: compute $x_{k+1}$ using $x_k$

    $m = (y_{k+1} - y_k)/((x_{k+1} - x_k)$, $y_{k+1} = y_k + 1$, $x_{k+1} = x_k + 1/m$

    Choose the closest pixel to represent $(x_{k+1}, y_{k+1})$

    Warning: pixel must be inside the polygon

    The closest interior point can be computed using only integer arithmetic



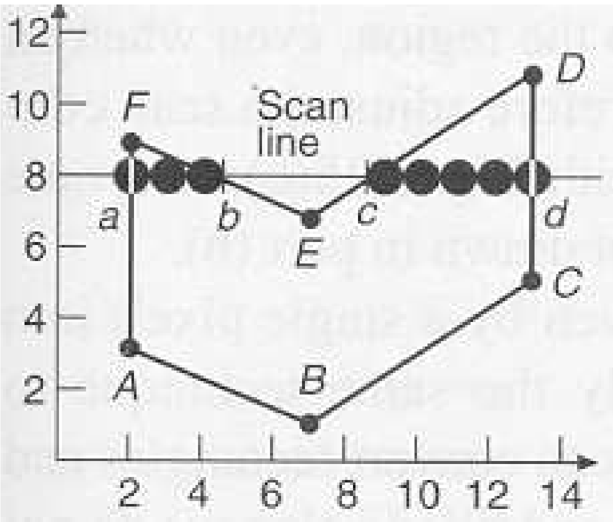
- **Efficient polygon filing**

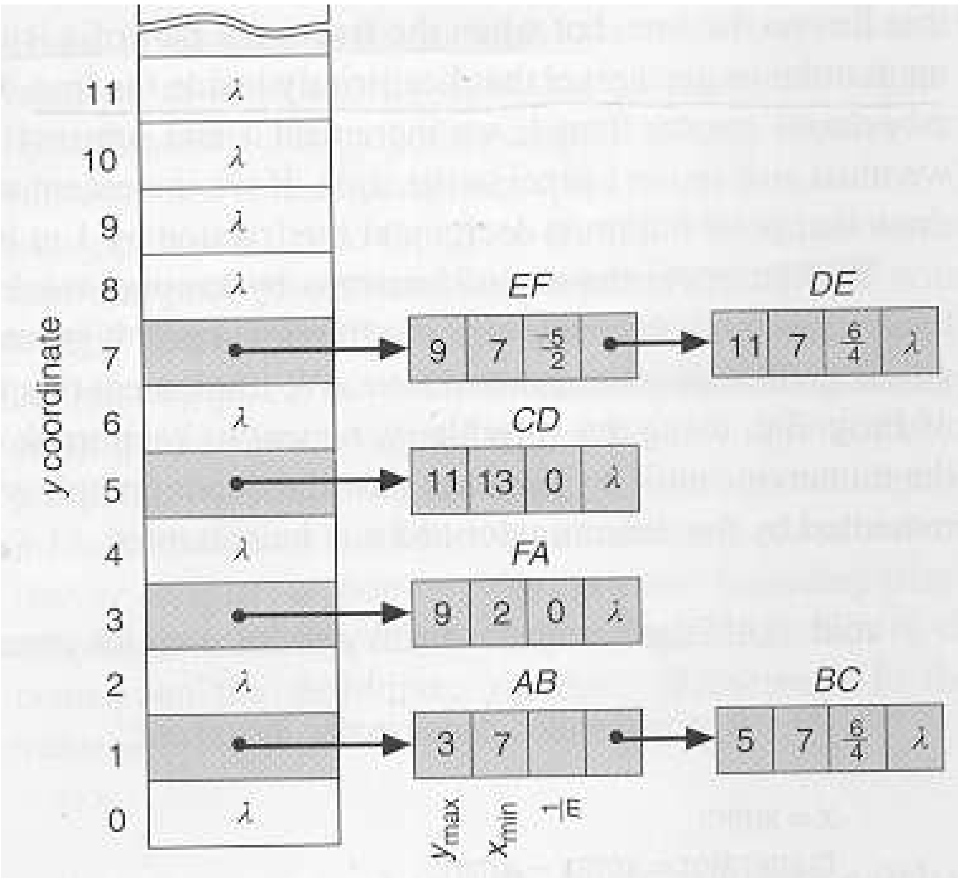
  - Store the polygon edges in a sorted edge table (ET)

    * Edges are sorted by their smaller y-coordinate

    * Edges in the same bucket are sorted by the x-coordinate of the lower point

    * Horizontal edges are not stored

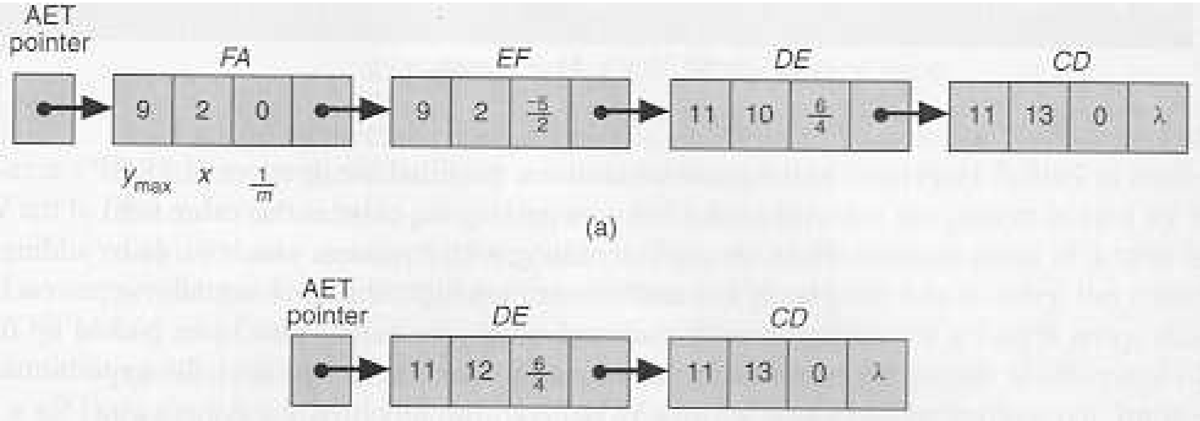


- Use an active-edge list (AEL) to keep track of the edges that the current scan-line intersects

   * The AEL contains all edges crossed by the current scan-line

   * The edges in the AEL are sorted in an increasing order, using the x intersection values




- **Algorithm (using ET and AEL)**

- Process the scan lines from the bottom of the polygon to its top

   1. Create a list of active edges for the first scan-line

   2. Find intersection points (i.e., $x_{k+1} = x_k + 1/m$, $x_0 = x_{min}$), sort them, and fill the spans defined by the pairs

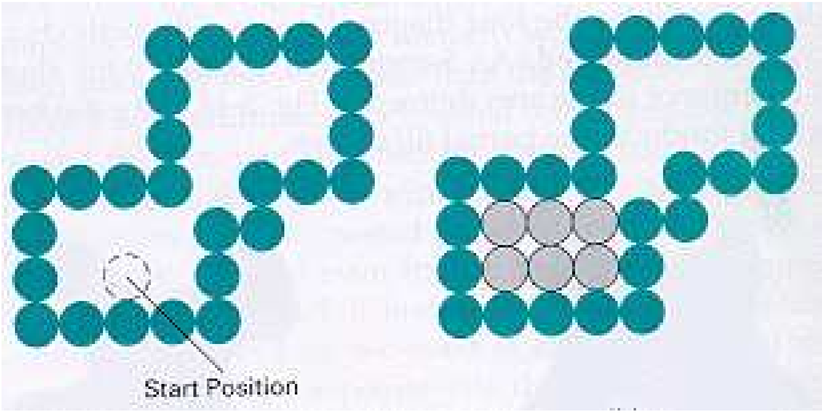
   3. Move to the next scan-line and update AEL

      3.1 Delete edges which are currently in the AEL but not intersected by the next scan-line (use $y_{max}$ to find if next scan-line intersects an edge)

      3.2 Add new edges from ET which are intersected by the next scan-line

   4. if AEL is not empty, goto step 2

Seed polygon filling



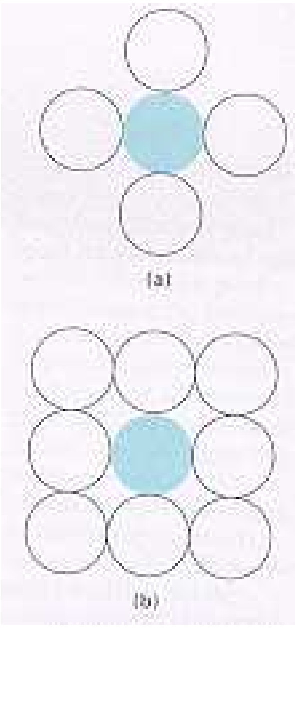

- **Algorithm (seed fill)**

    1. Start from a point inside the region

    2. Test neighboring pixels to determine whether they correspond to boundary pixels

    3. If not, paint them with the fill color and test their neighboring pixels (store neighbors in a stack)

    4. Continue until all pixels in the region have been tested



```
void boundaryFill4 (int x, int y, int fill, int boundary)
{
  int current;

  current = getPixel (x, y);
  if ((current != boundary) && (current != fill)) {
    setColor (fill);
    setPixel (x, y);
    boundaryFill4 (x+1, y, fill, boundary);
    boundaryFill4 (x-1, y, fill, boundary);
    boundaryFill4 (x, y+1, fill, boundary);
    boundaryFill4 (x, y-1, fill, boundary);
  }
}
```

- Disadvantage: considerable stacking is needed

- **A more efficient algorithm (scan-line seed fill)**

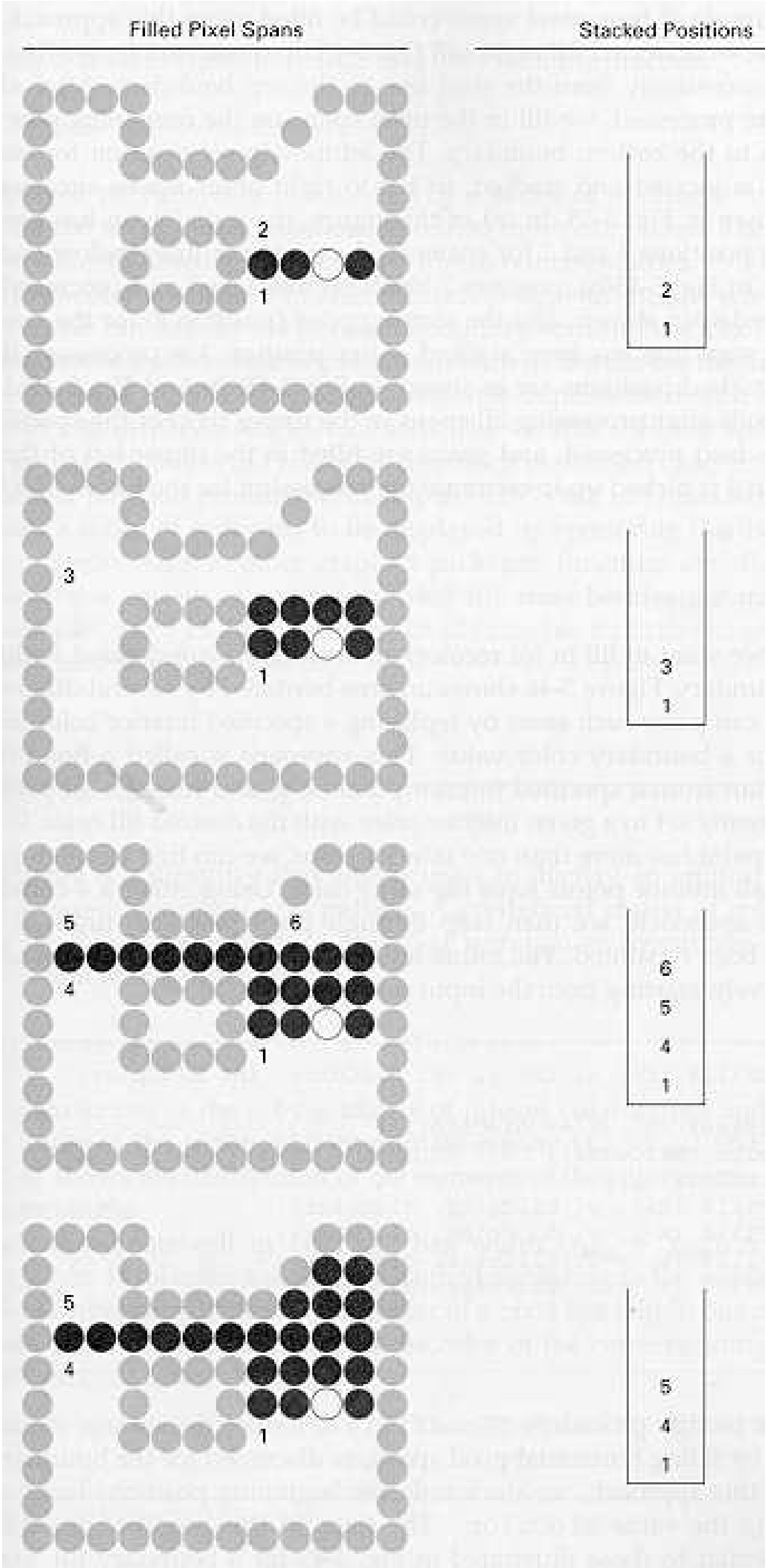
- Do not proceed to 4-connected or 8-connected pixels

- Fill horizontal pixel spans across scan-lines

  1. The span containing the seed pixel is filled to the right and left of the seed pixel along a scan-line until a boundary is found

  2. Locate and stack starting positions for spans on adjacent scan-lines

  3. Unstack the next start position and repeat the process


Filled Pixel Spans
Stacked Positions