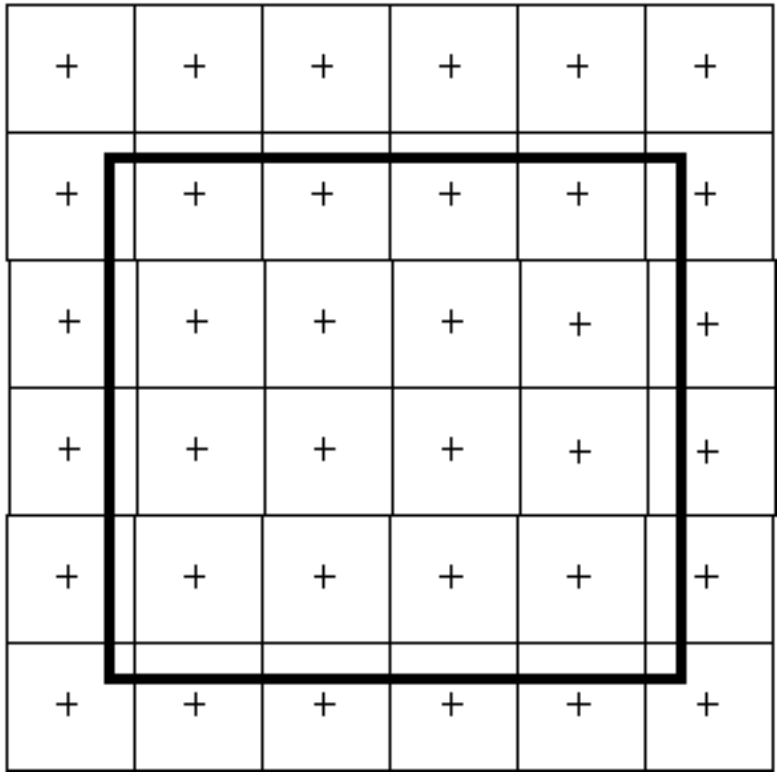
Which pixel is inside?

- Each pixel is a *sample*, at coordinates (x, y).
  - imagine a piece of paper, where coordinates are continuous
  - pixels are samples on a grid of a drawing on this piece of paper.
- If ideal point (corresponding to grid center) is inside, pixel is inside. **(Easy case)**



Which pixel is inside?

In the context of the sweep fill algorithm to come soon: Suppose we are sweeping from left to right. Then for pixels with **fractional** intersections:

1) Going from outside to inside, then take true intersection, and r**ound up** to get first interior point.
2) Going from inside to outside, then take true intersection, and **round down** to get last interior point.

Note that if we are considering an adjacent polygon, 1) and 2) are reversed, so it should be clear that for most cases, the pixels owned by each polygon is well defined (and we don't erase any when drawing the other polygon).

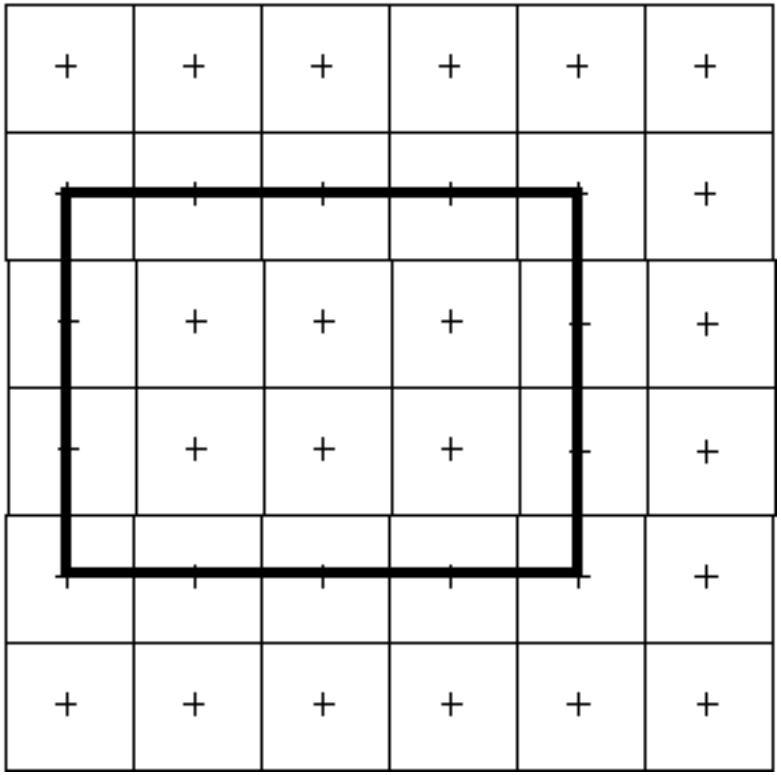
Ambiguous cases

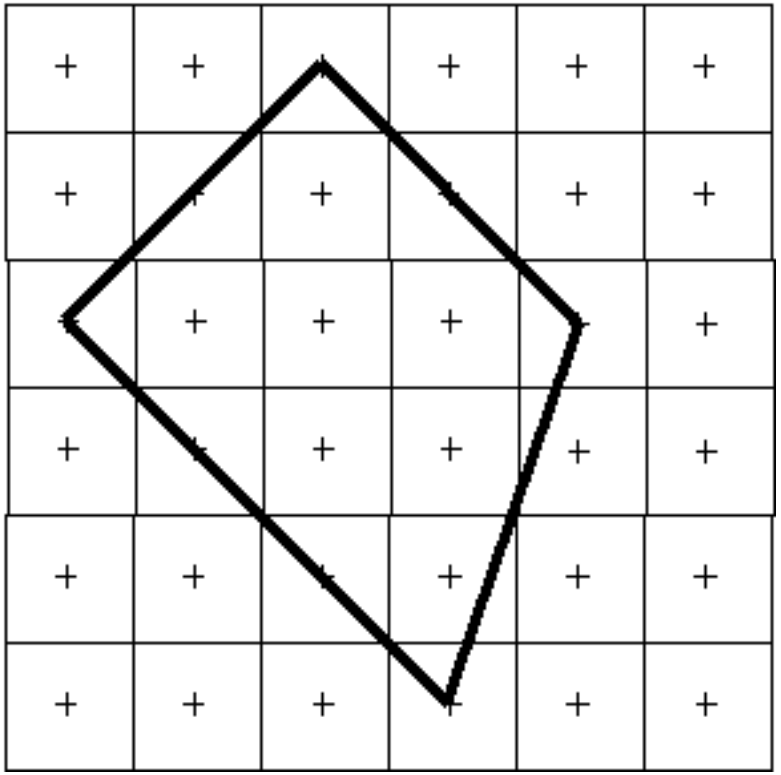
- What if a pixel is exactly on the edge? (non-fractional case)
- Polygons are usually adjacent to other polygons, so we want a rule which will give the pixel to *one* of the adjacent polygons or the *other* (as much as possible).
- Basic rule: Draw left and bottom edges
  - horizontal edge? if (x+∂, y+□) is in, pixel is in
  - otherwise if (x+∂, y) is in, pixel is in
  - in practice one implements a sweep fill procedure that is consistent with this rule (we don't test the rule explicitly)

Ambiguous cases

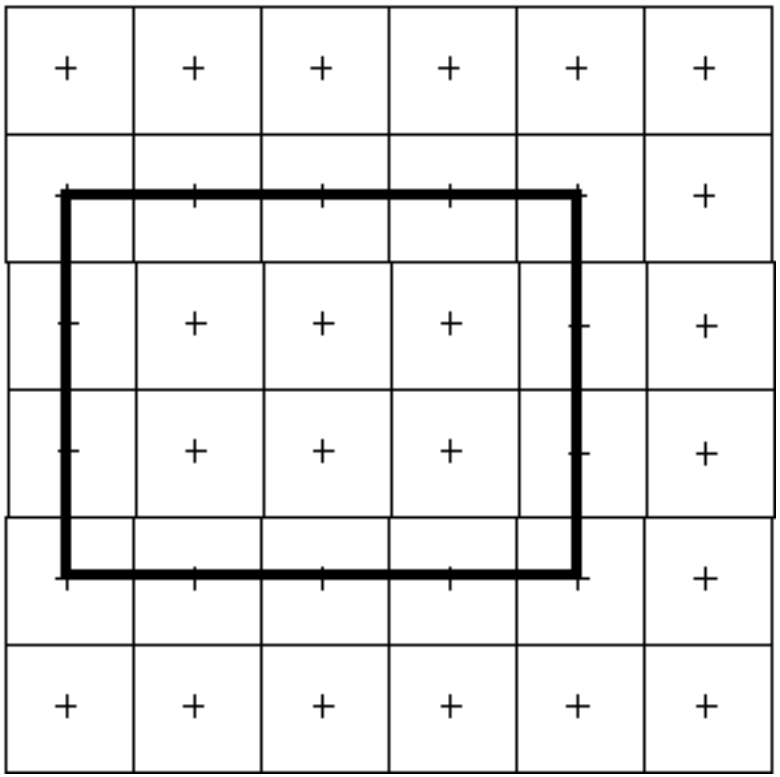
- What if it is a vertex between the two cases?
  - In this case we essentially draw left vertices and bottom vertices, but details get absorbed into scan conversion (sweep fill). Note that the algorithm in Foley et al. solves this problem by making a special case for parity calculation (y_{min} vertices are counted for parity calculation, but y_{max} are not)
  - As mentioned on the bottom of page 86 of Foley et al., there is no perfect solution to the problem. There will be edges that could be closed, but are left open in case another polygon comes by that would compete for pixels, and, on occasion, there is a “hole” (preferred compared to rewriting pixels).

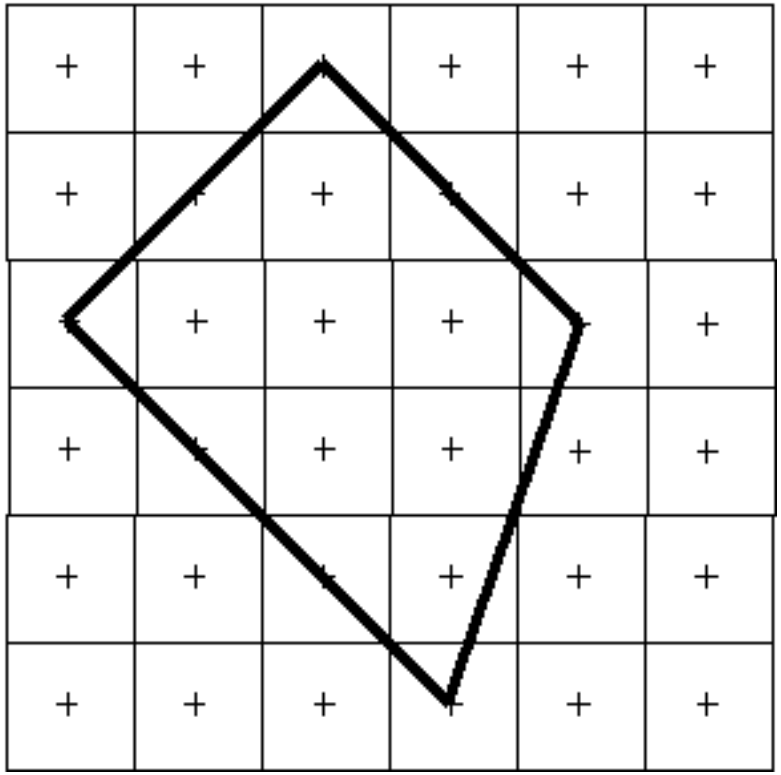
Ambiguous inside cases (?)



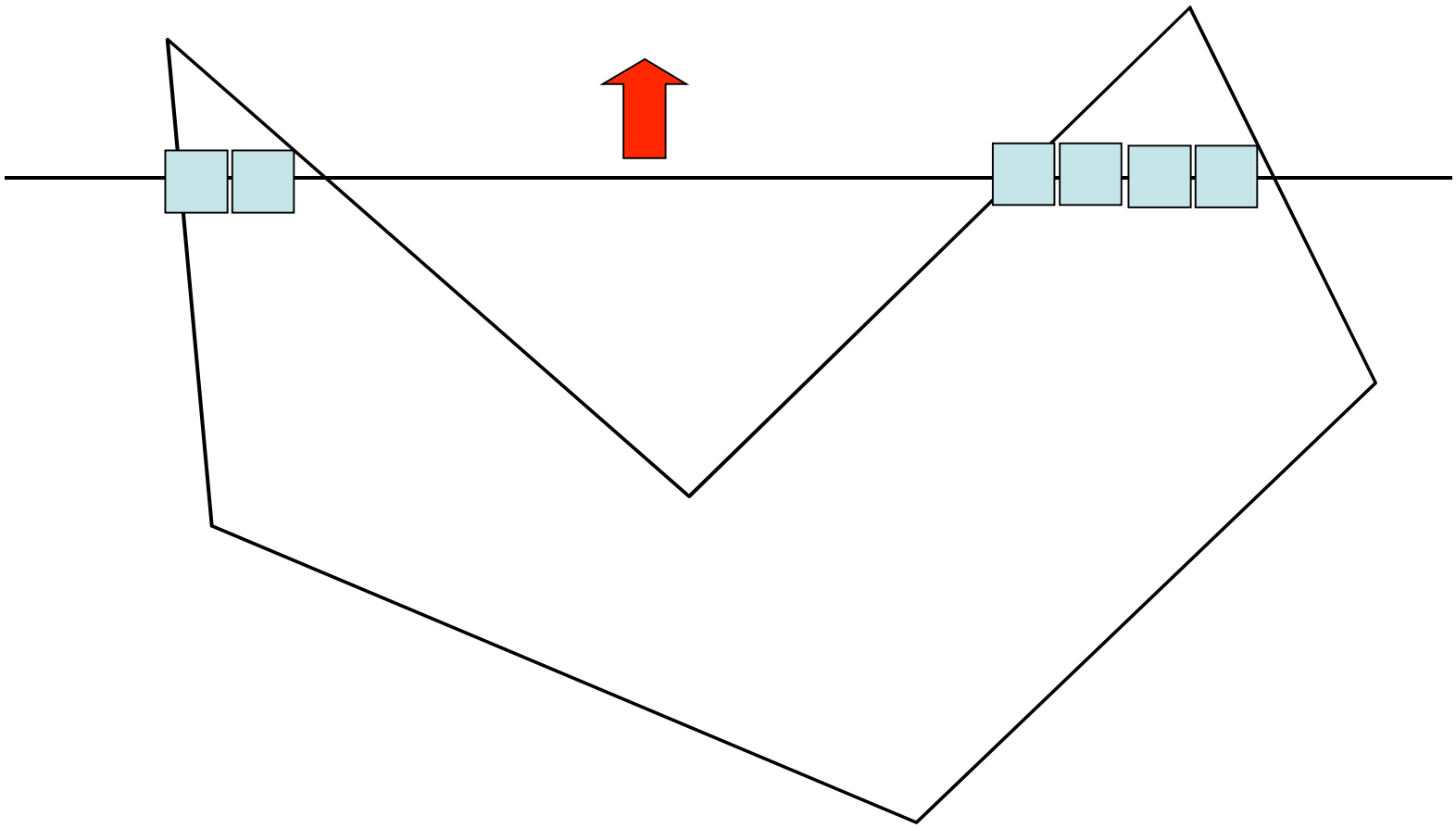


Ambiguous inside cases (answer)





Sweep fill

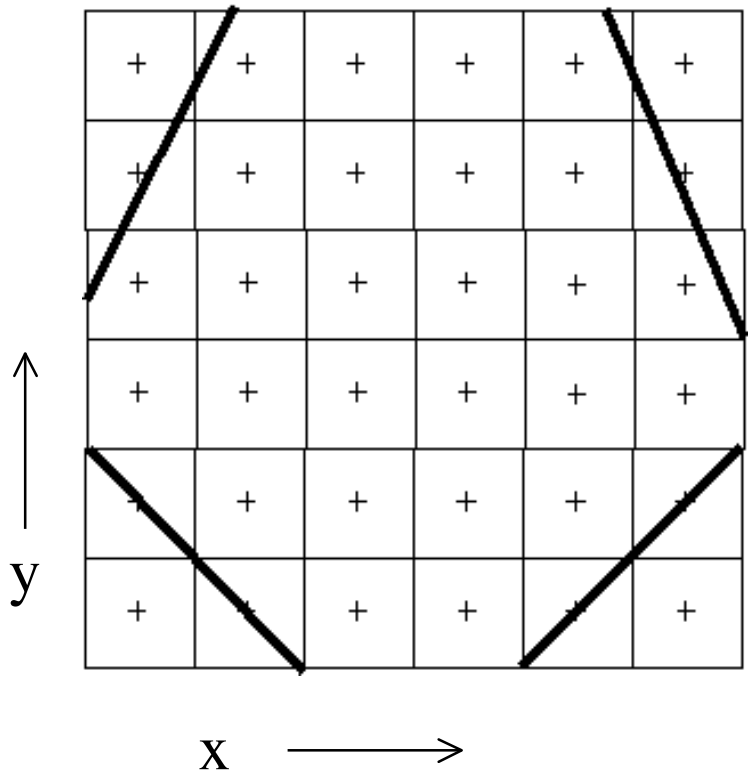


Sweep fill

- Reduces to filling many spans
- Inside/outside parity is relatively straightforward
- Need to compute the spans, then fill
- Need to update the spans for each scan
- Need to implement “inside” rule for ambiguous cases.

Spans

- Fill the bottom horizontal span of pixels; move up and keep filling
- Assume we have xmin, xmax.
- Recall--for non integral xmin (going from outside to inside), r**ound up** to get first interior point, for non integral xmax (going from inside to outside), **round down** to get last interior point
- Recall--convention for integral points gives a span closed on the left and open on the right
- **Thus**: fill from ceiling(xmin) up to but not including ceiling(xmax)



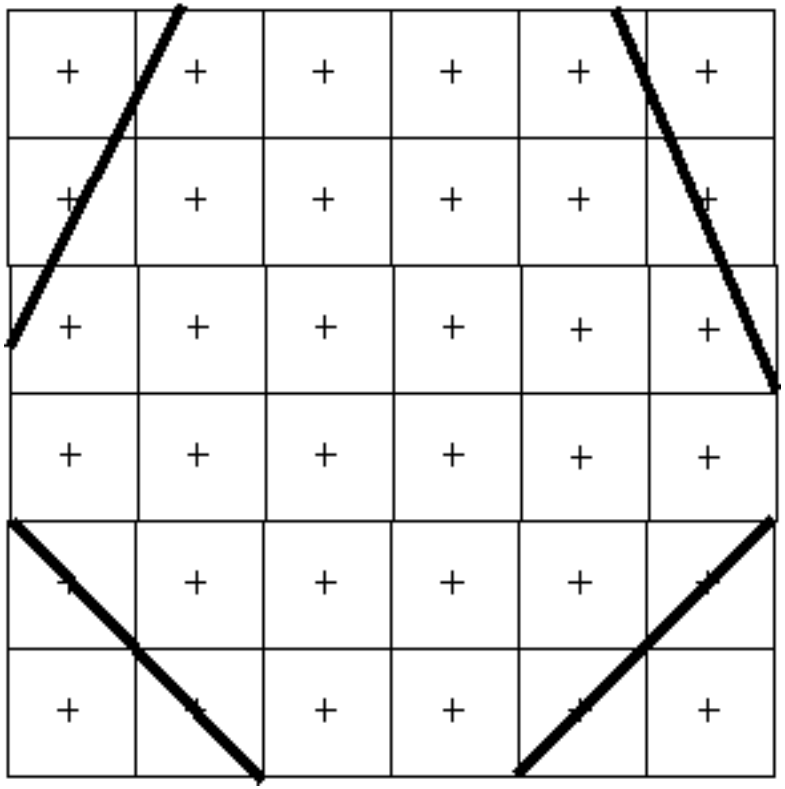
Algorithm

- For each row in the polygon:
  - Throw away irrelevant edges (horizontal ones, ones that we are done with)
  - Obtain newly relevant edges (ones that are starting)
  - Fill spans
  - Update spans

The next span - 1

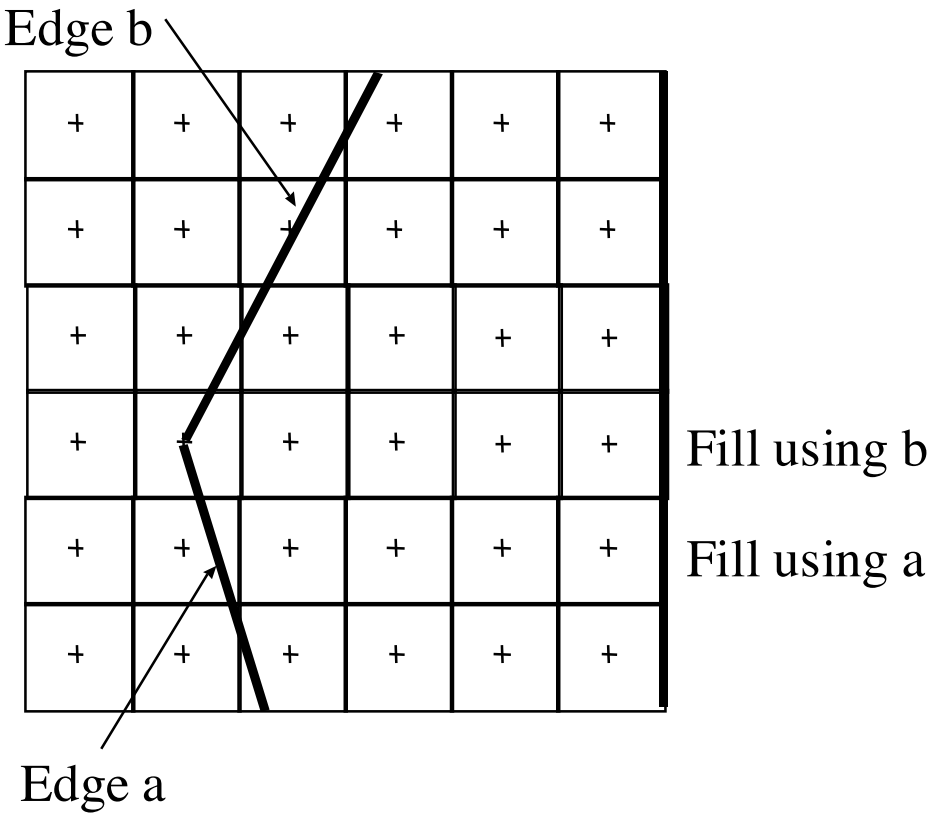
- for an edge, have $y=mx+c$
- hence, if $y_n=m\ x_n+c$, then $y_{n+1}=y_n+1=m\ (x_n+1/m)+c$
- hence, *if there is no change in the edges*, have:

$$x \mathrel{+}= (1/m)$$

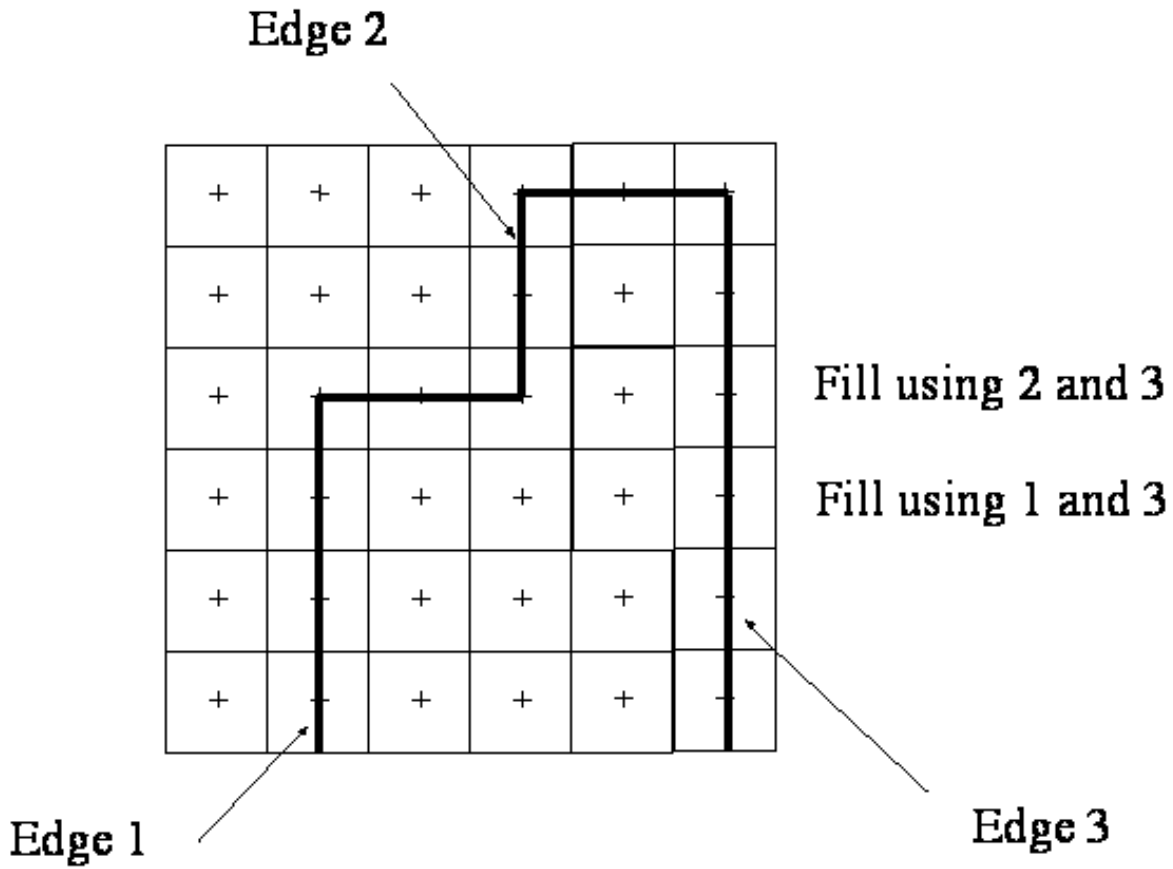


The next span - 2

- Horizontal edges are irrelevant (typically would be pruned at the outset)
- Edge becomes relevant when y>=ymin of edge (note appeal to convention)*
- Edge becomes irrelevant - when y>=ymax of edge (note appeal to convention)*



*Because we add edges and check for irrelevant edges *before* drawing, bottom horizontal edges are drawn, but top ones are not.

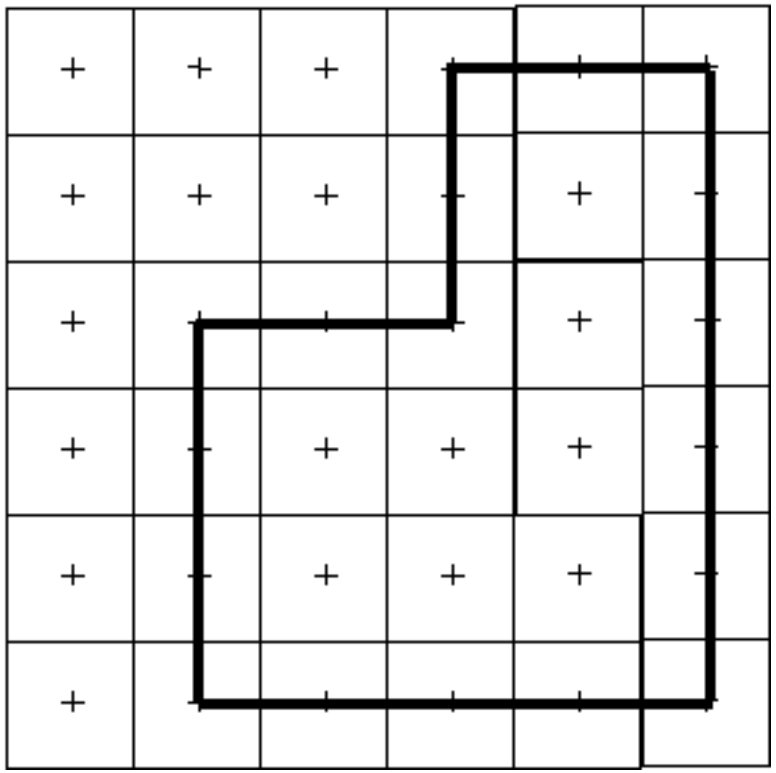

Edge 2
Fill using 2 and 3
Fill using 1 and 3
Edge 1
Edge 3

Filling in details -- 1

- For each edge store: x-value, maximum y value of edge, 1/m
  - x-value starts out as x value for y_{min}
  - m is never 0 because we ignore horizontal ones
- Keep edges in a table, indexed by minimum y value (Edge Table==ET)
- Maintain a list of active edges (Active Edge List==AEL).

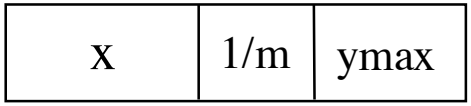
Filling in details -- 2

- For row = min to row=max
  - AEL=append(AEL, ET(row)); (add edges starting at the current row)
  - remove edges whose ymax=row
    - OK since we are assuming integral coordinates; otherwise one would use ceil(ymax)
  - sort AEL by x-value
  - fill spans
    - parity rule
    - convention for integral x_{min} and x_{max}
  - update each edge in AEL
    - x += (1/m)

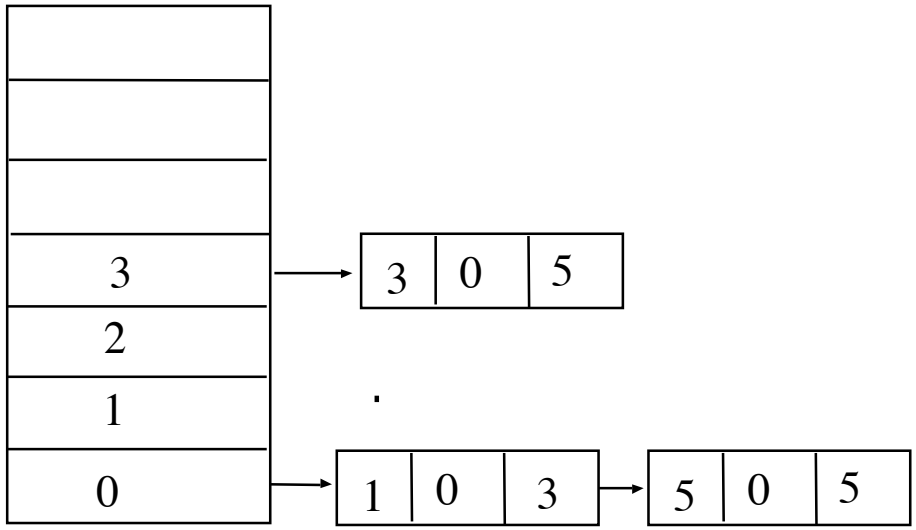


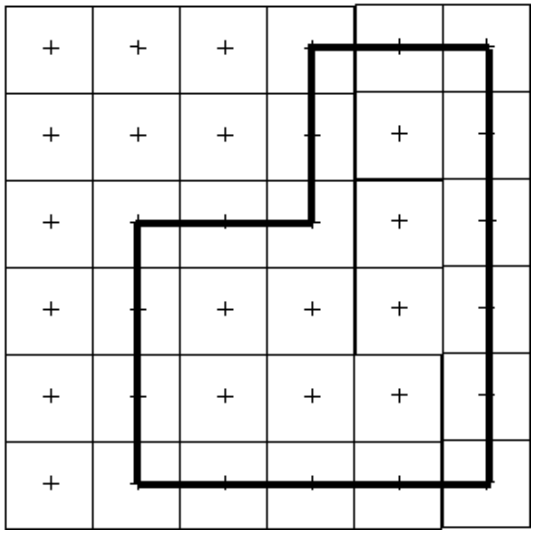
Compute the edge table (ET) to begin. Then fill polygon and update active edge list (AEL) row by row.

Format of edge entries

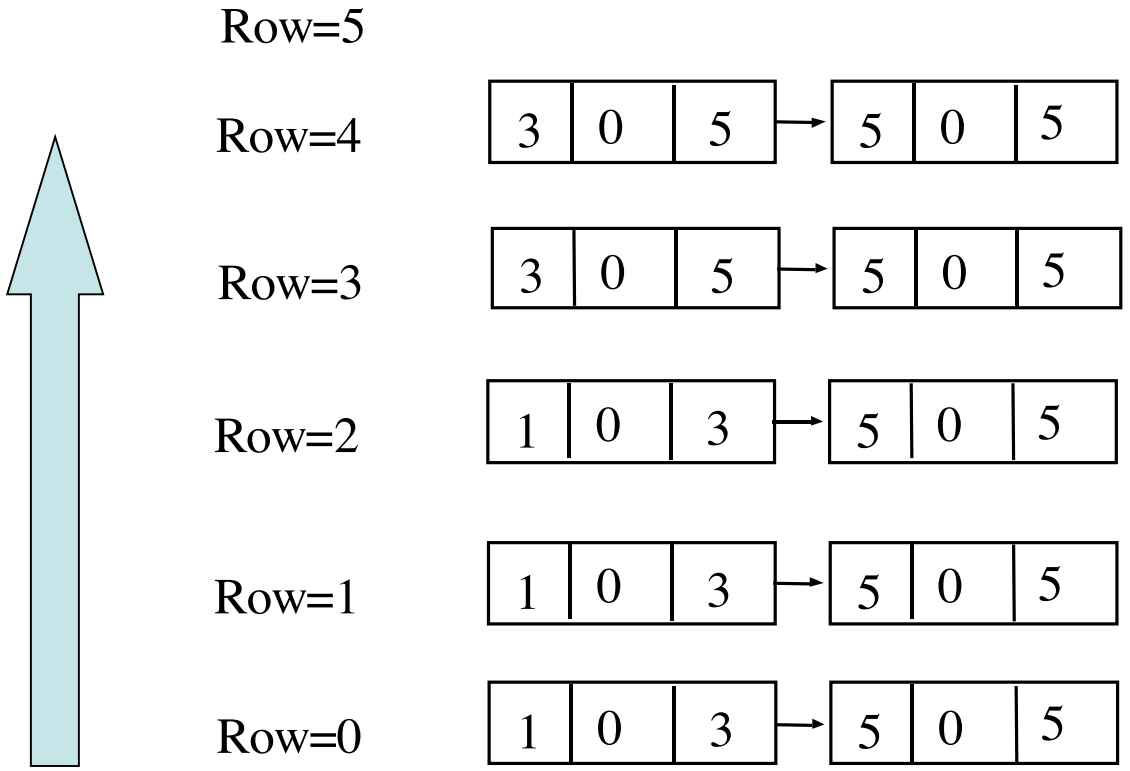


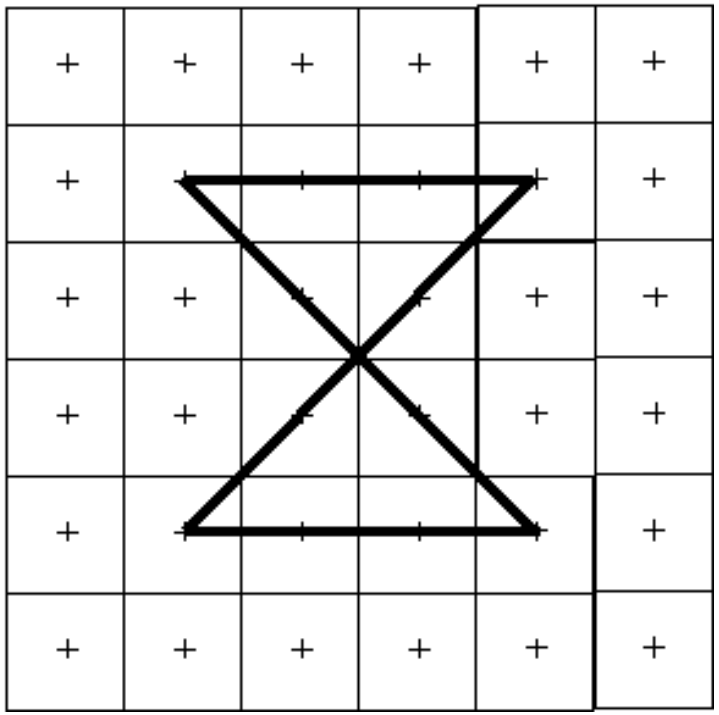
ET





AEL just before filling

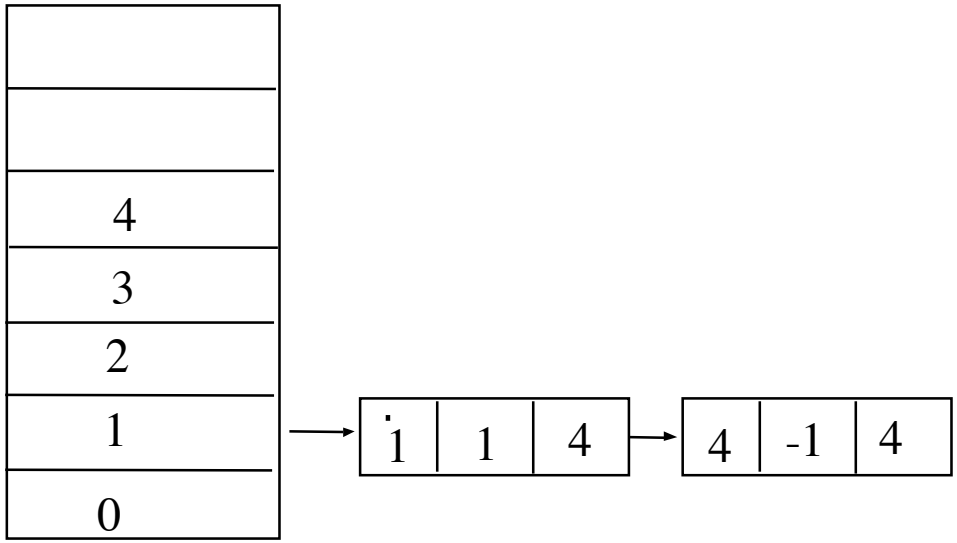


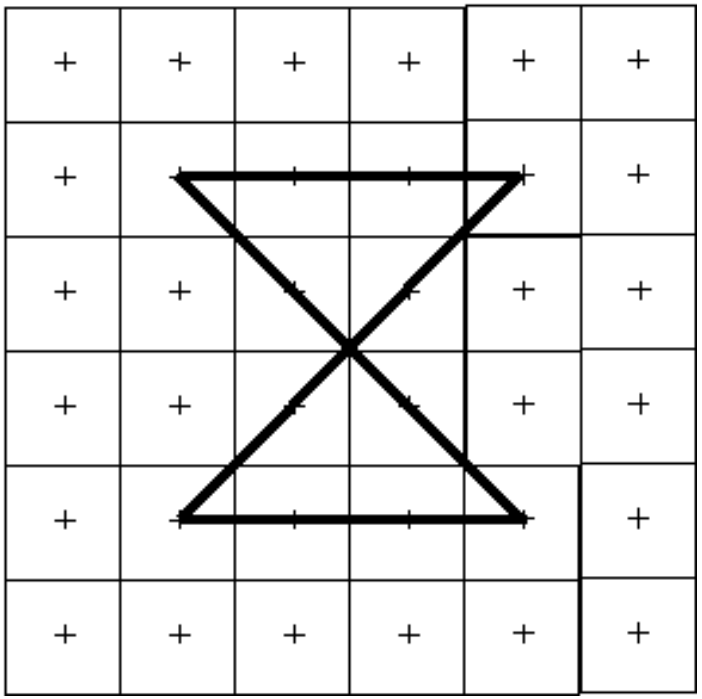


Format of edge entries

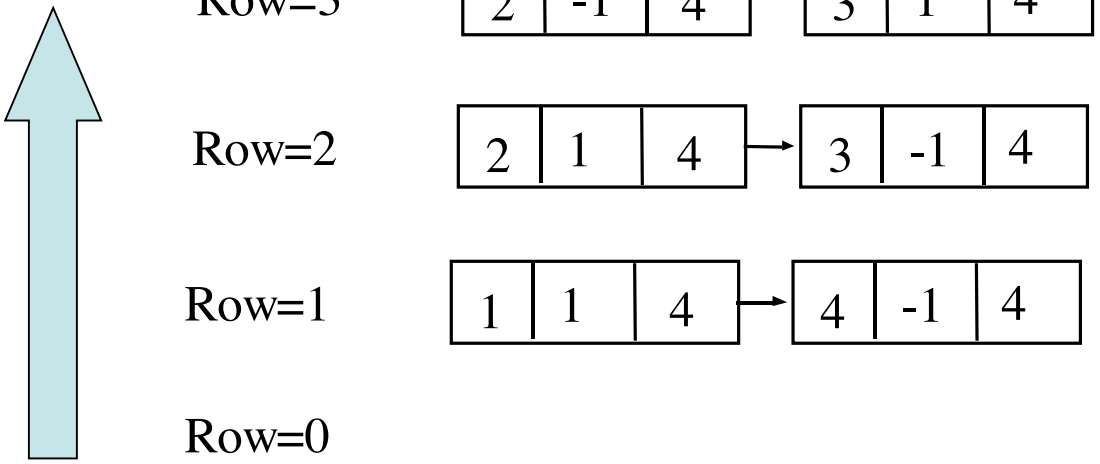
x	1/m	ymax

ET





AEL just before filling



Comments

- Sort is quite fast, because AEL is usually almost in order.
- Nonetheless, OpenGL limits to convex polygons, so two and only two elements in AEL at any time, and no sorting.
- With additional logic to keep track of what color to use, can fill in many polygons at a time.
- Can be done *without* floating point

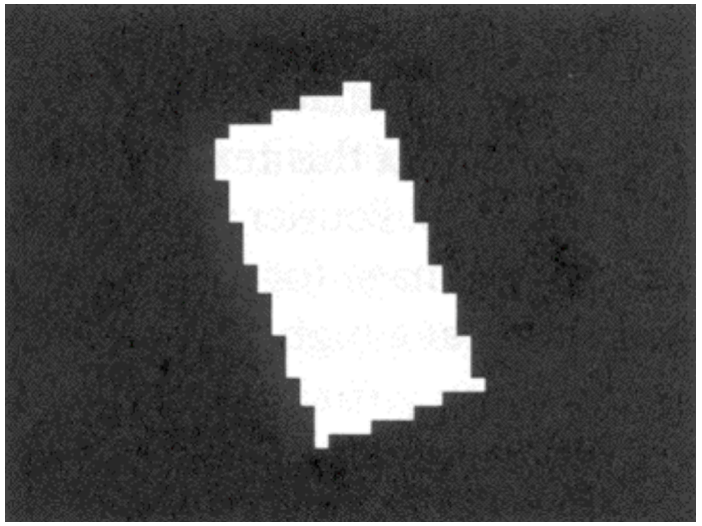
Dodging floating point

- 1/m=Dx/Dy, which is a rational number.
- x = x_int + x_num/Dy
- store x as (x_int, x_num),
- then x->x+1/m is given by:
  - x_num=x_num+Dx
  - if x_num >= x_denom
    - x_int=x_int+1
    - x_num=x_num-x_denom

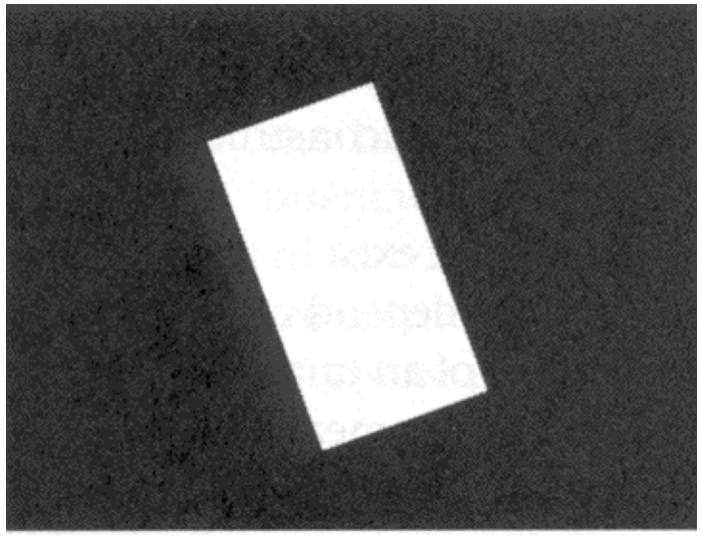
- Advantages:
  - no floating point
  - can tell if x is an integer or not (check x_num=0), and get truncate(x) easily, for the span endpoints.

Aliasing/Anti-Aliasing

- Analogous to the case of lines
- Anti-aliasing is done using graduated gray levels computed by smoothing and sampling



Aliasing



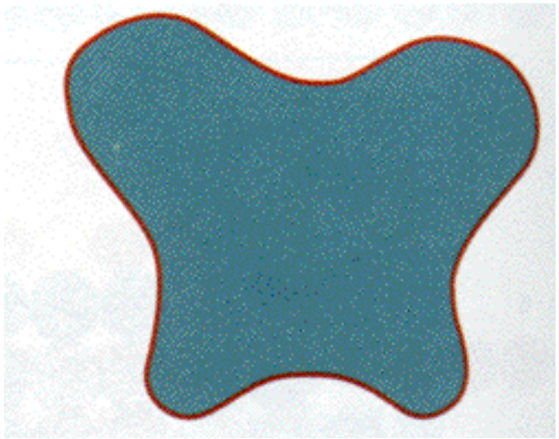
Ideal

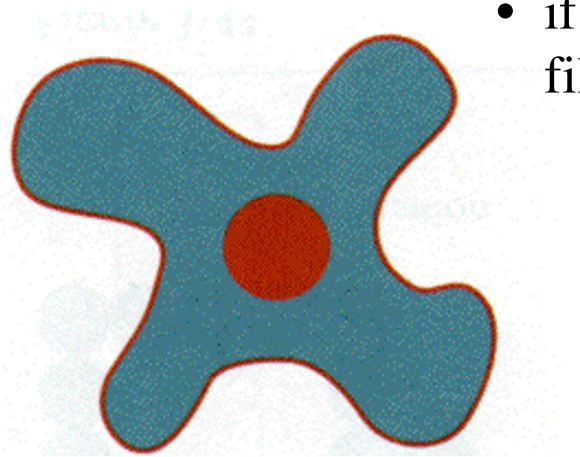
Aliasing/Anti-Aliasing

- Some anti-aliasing approaches implicitly deal with boundary ambiguity
- Problem with “slivers” is really an aliasing problem.

Boundary fill

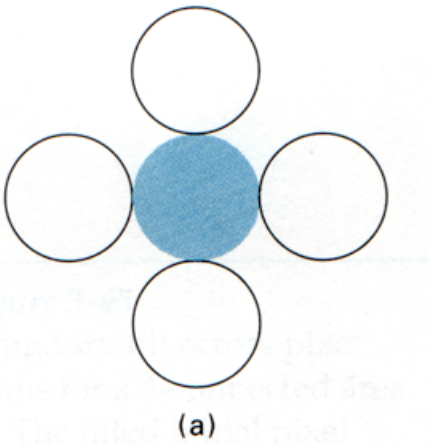
- Basic idea: fill in pixels inside a boundary





- Recursive formulation:
  - to fill starting from an inside point
    - if point has not been filled,
      - fill
      - call recursively with all neighbours that are not boundary pixels

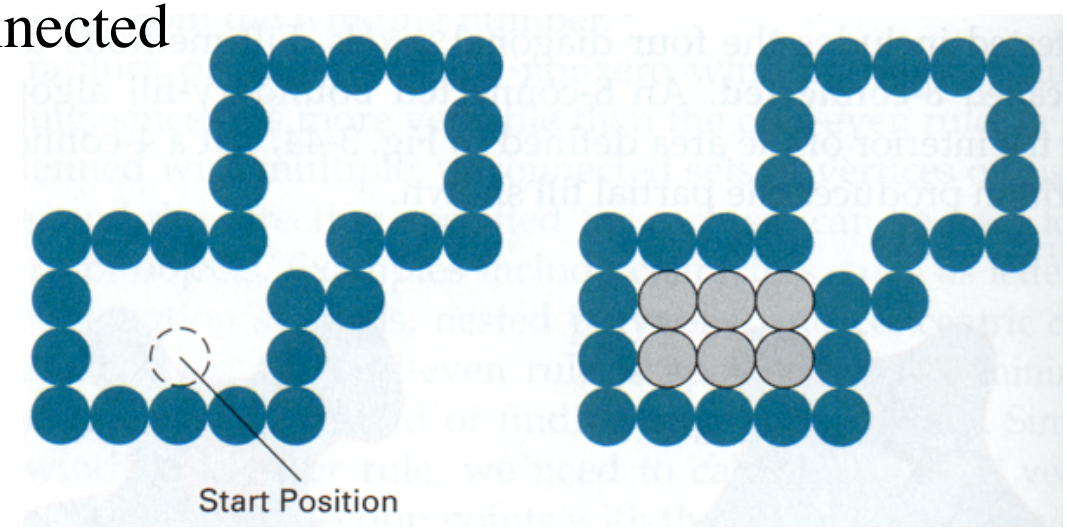
Choice of neighbours is important



4-connected

8 connected

4 connected fill of
a four connected
boundary doesn't work



Pattern fill

- Use coordinates as index into pattern