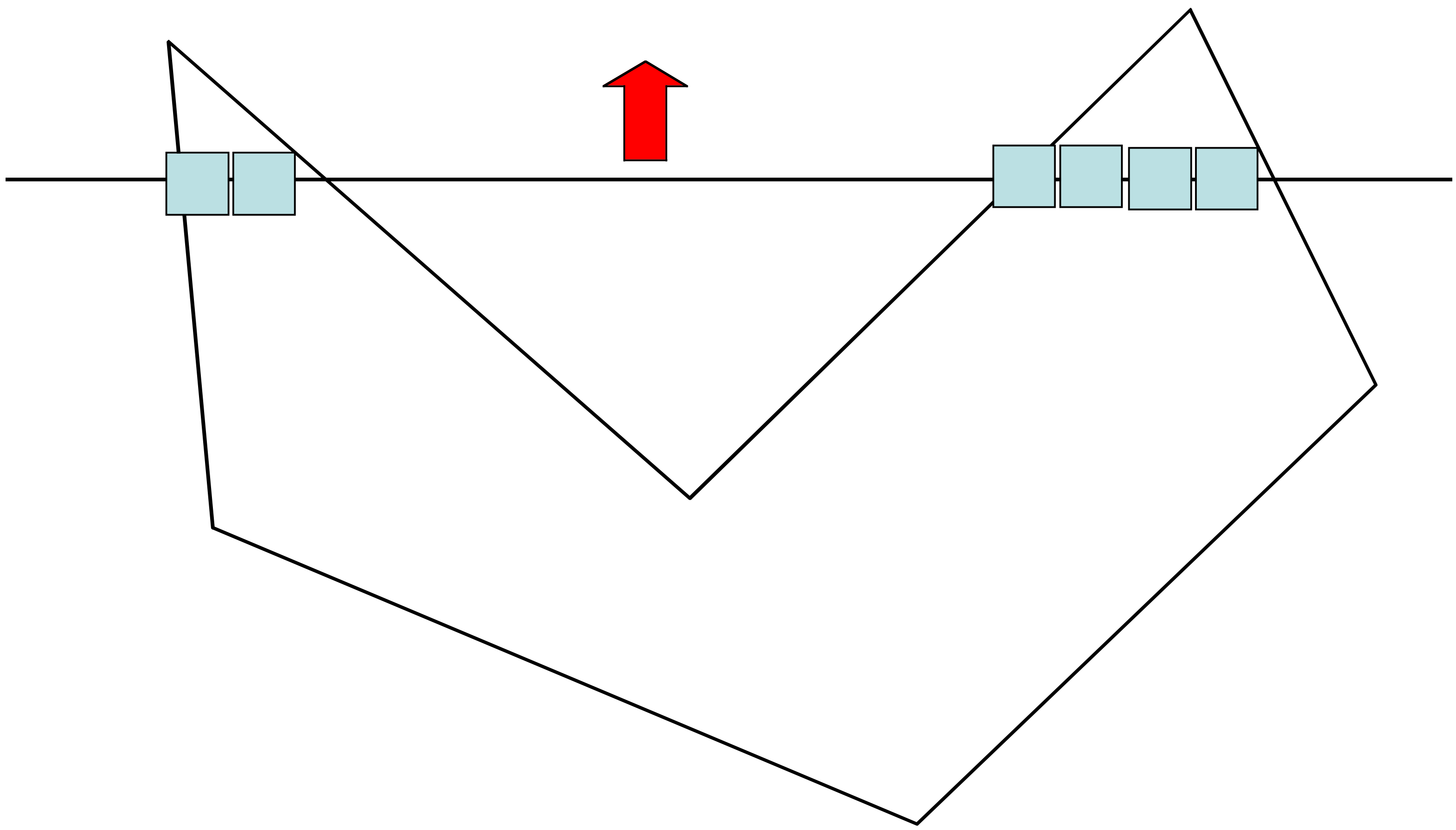
Sweep fill

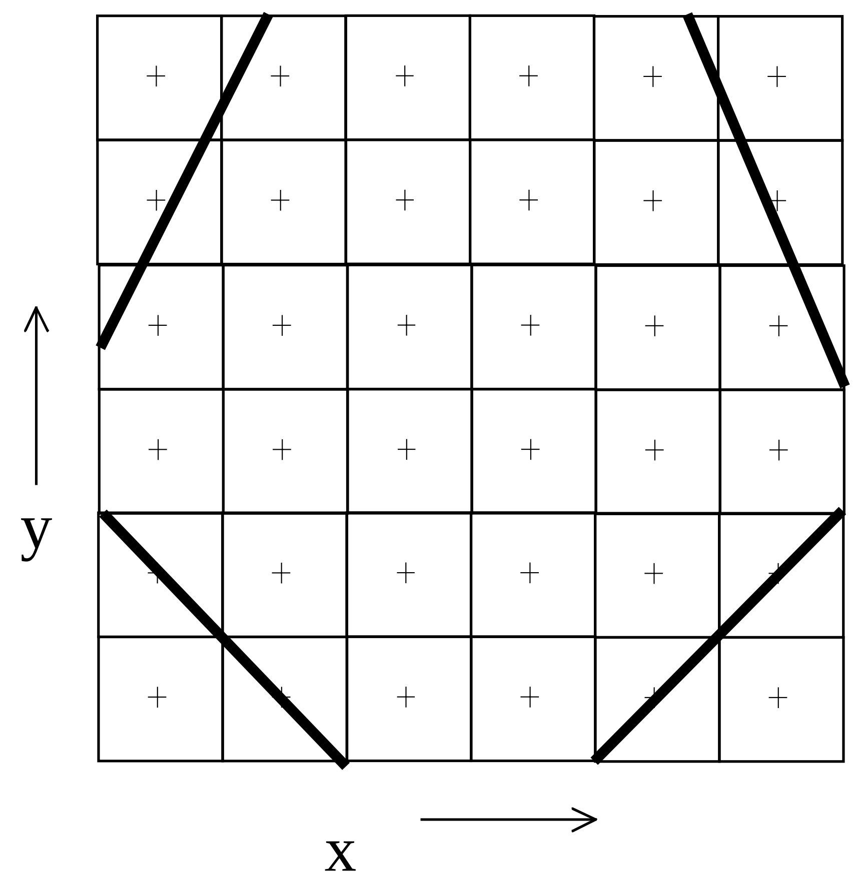


Sweep fill

- 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

- Process - fill the bottom horizontal span of pixels; move up and keep filling
- have xmin, xmax.
- define:
  - floor(x):= if x integer, x else truncate(x)
  - ceiling(x):= truncate(x)+1
- fill from ceiling(xmin) up to but not including floor(xmax)
- consistent with convention



Algorithm

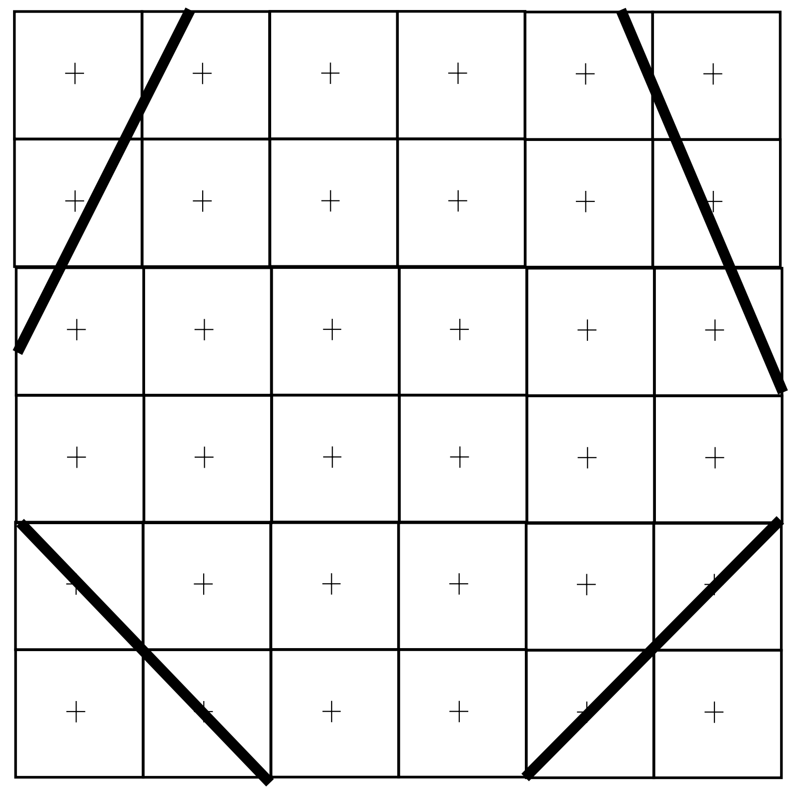
- For each row in the polygon:
  - Throw away irrelevant edges
  - Obtain newly relevant edges
  - Fill spans
  - Update spans
- Issues:
  - what aspects of edges need to be stored?
  - when is an edge relevant/irrelevant?

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:

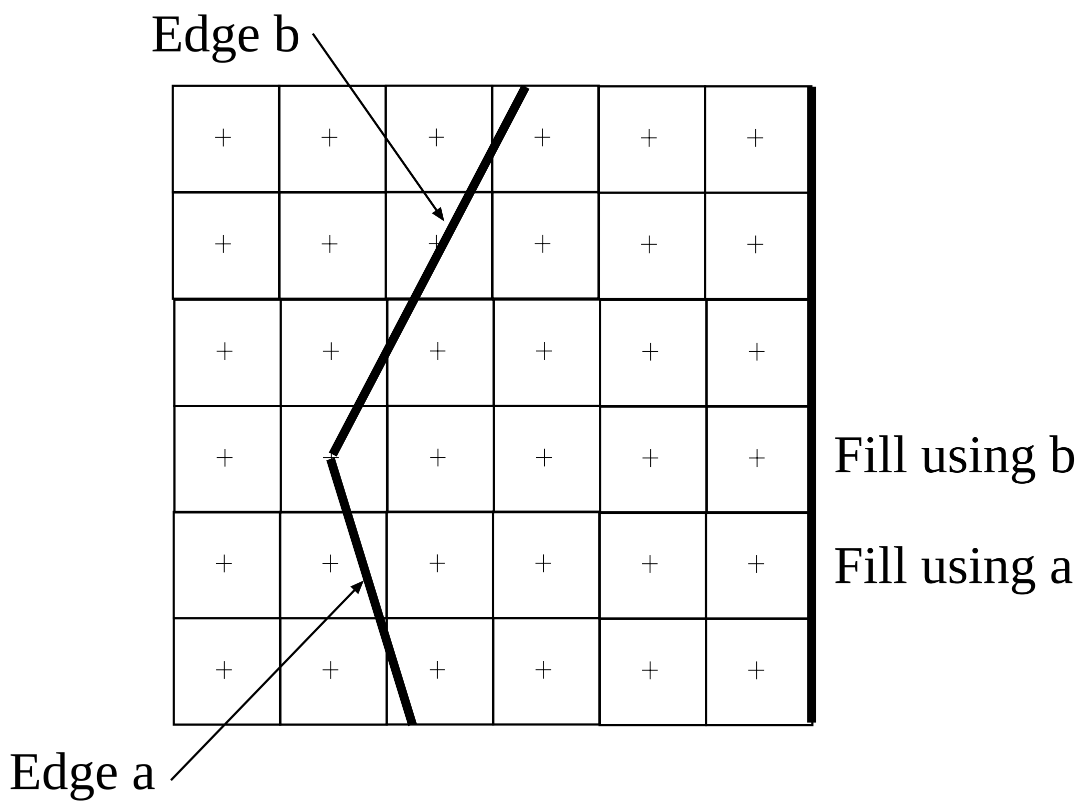
  xmax->xmax+(1/m)(xmax)

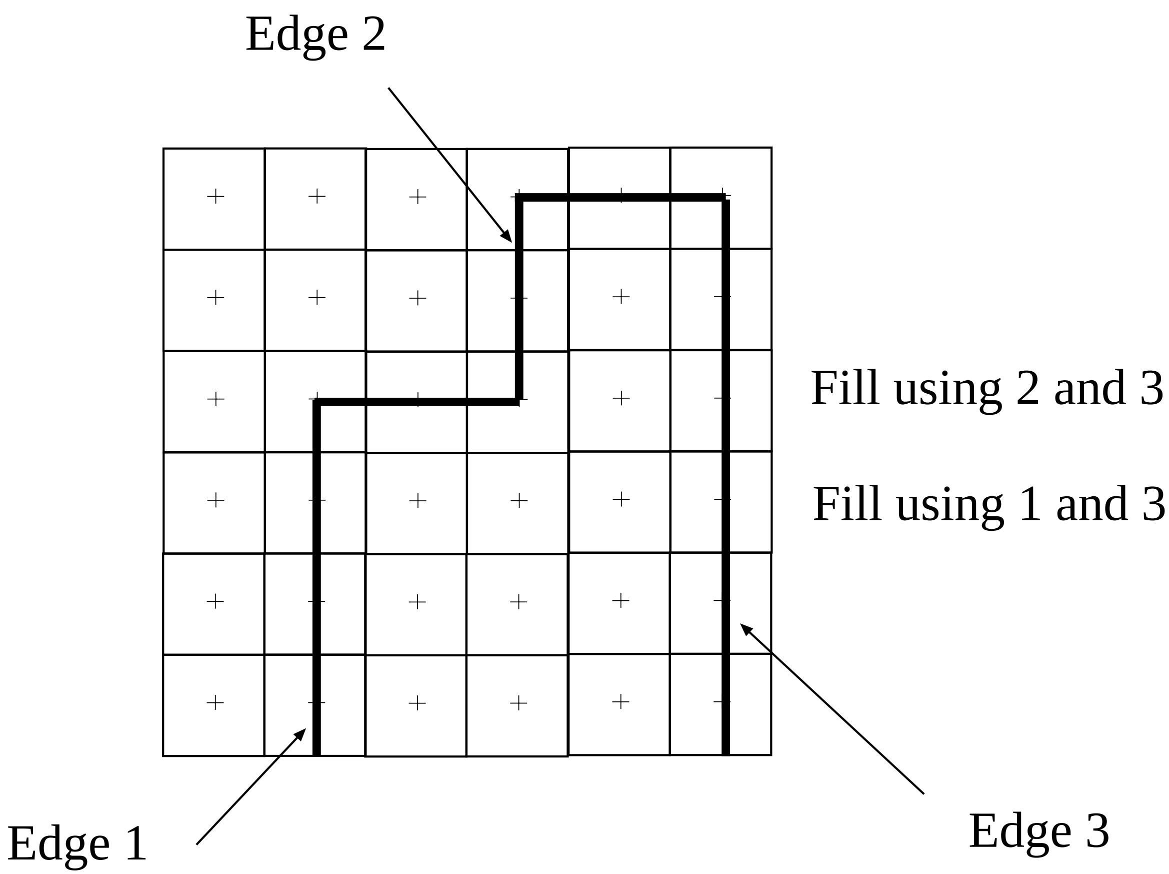
  xmin->xmin+(1/m)(xmin)



The next span - 2

- Horizontal edges are irrelevant
- Edge is irrelevant - when y>=ymax of edge (note appeal to convention)
- Similarly, edge is relevant when y>=ymin of edge

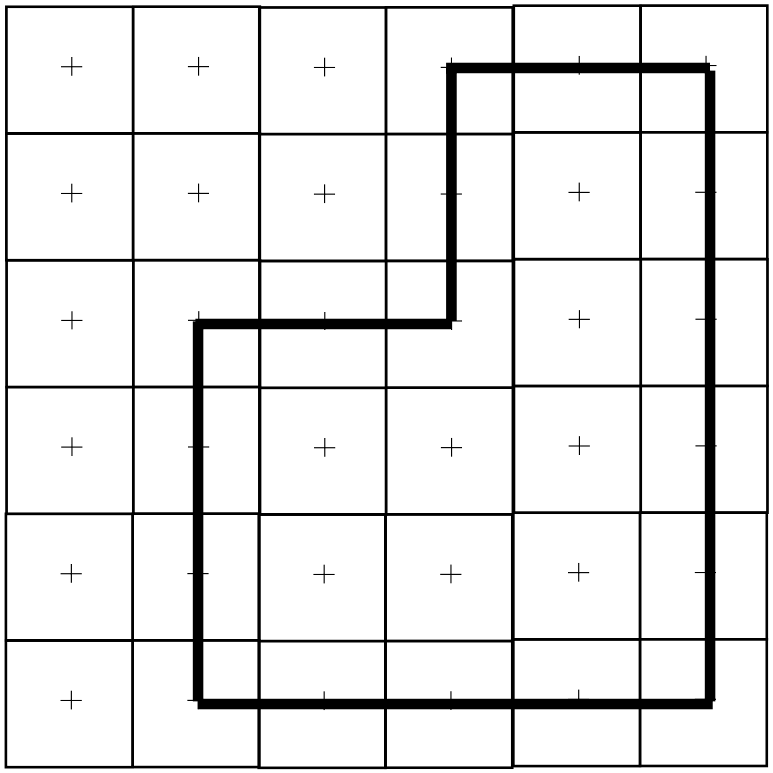



Edge 2
Fill using 2 and 3
Fill using 1 and 3
Edge 1
Edge 3

Filling in details

- maintain a list of active edges in case there are multiple spans of pixels - known as Active Edge List.
- for each edge on the list, must know: x-value, maximum y value of edge, 1/m
- Keep edges in a table, indexed by minimum y value - Edge Table

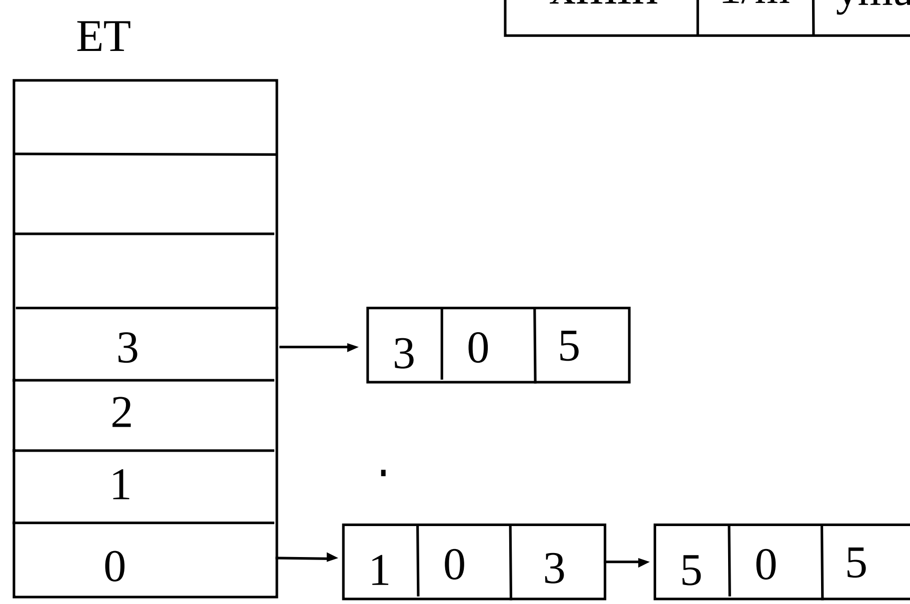
- For row = min to row=max
  - AEL=append(AEL, ET(row));
  - remove edges whose ymax=row
  - sort AEL by x-value
  - fill spans
  - update each edge in AEL

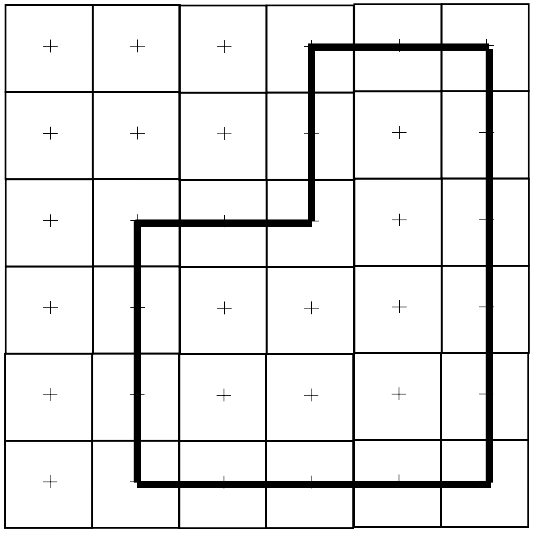


Compute the edge table (ET_ to begin. Then fill polygon and update active edgle list (AEL) row by row.

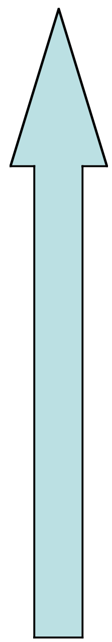
Format of AEL entries

xmin	1/m	ymax

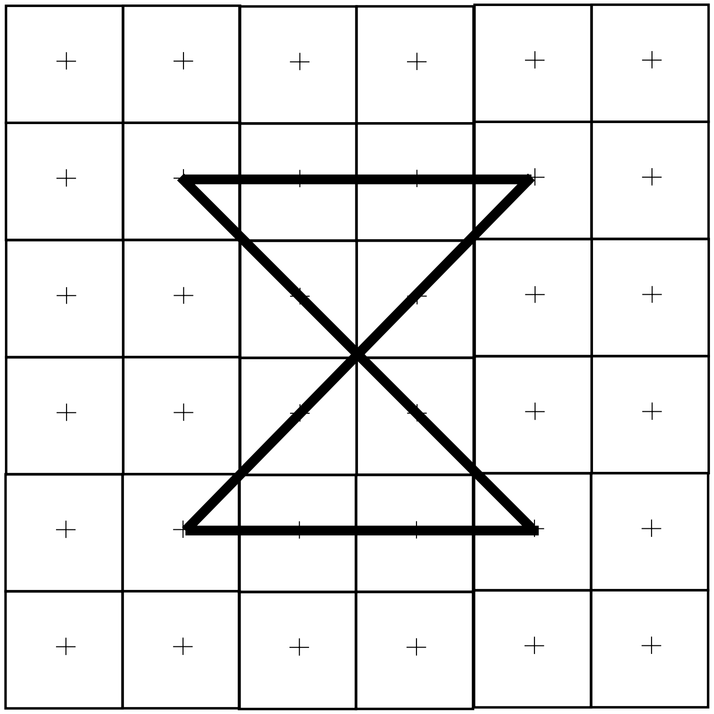




AEL just before filling



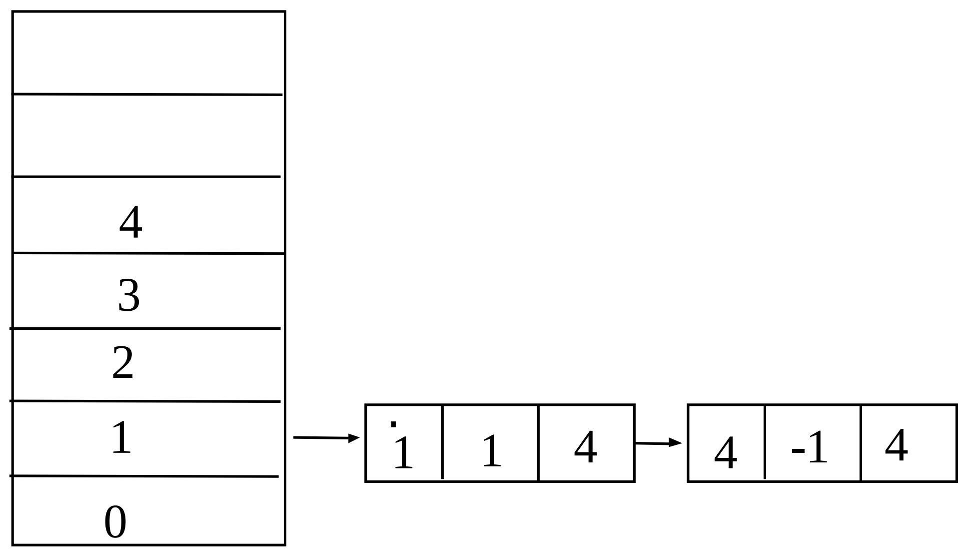
Row=5	
Row=4	[3 | 0 | 5] → [5 | 0 | 5]
Row=3	[3 | 0 | 5] → [5 | 0 | 5]
Row=2	[1 | 0 | 3] → [5 | 0 | 5]
Row=1	[1 | 0 | 3] → [5 | 0 | 5]
Row=0	[1 | 0 | 3] → [5 | 0 | 5]



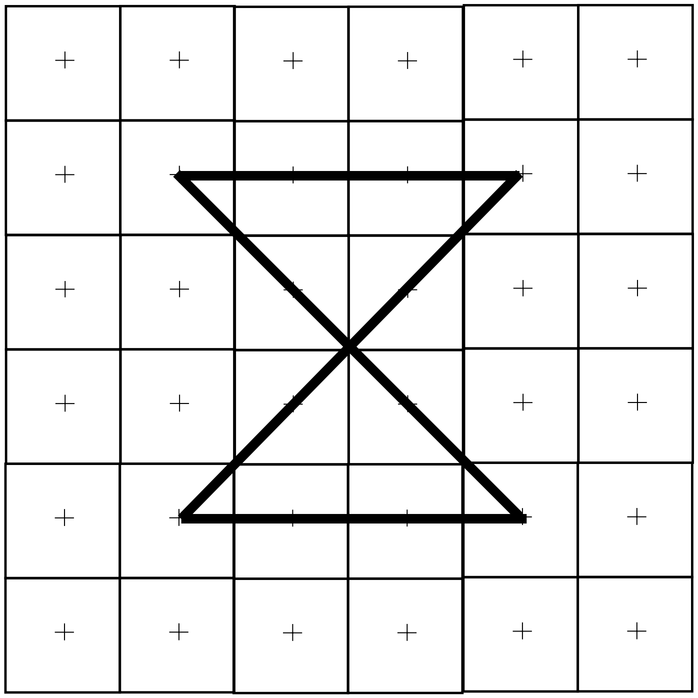
Format of AEL entries

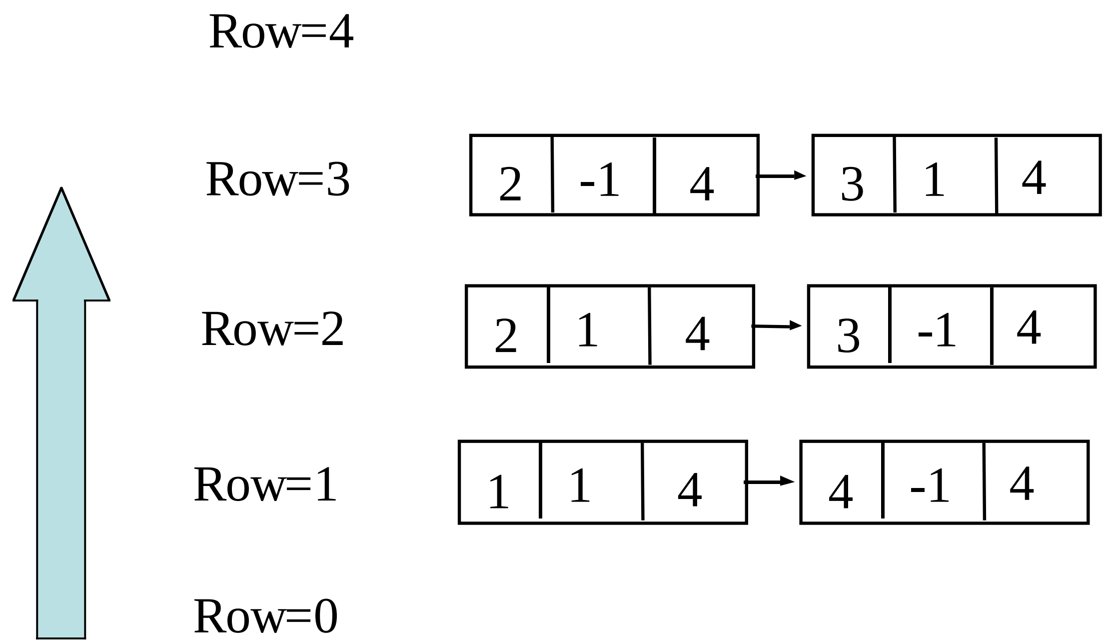
xmin	1/m	ymax

ET



AEL just before filling





Comments

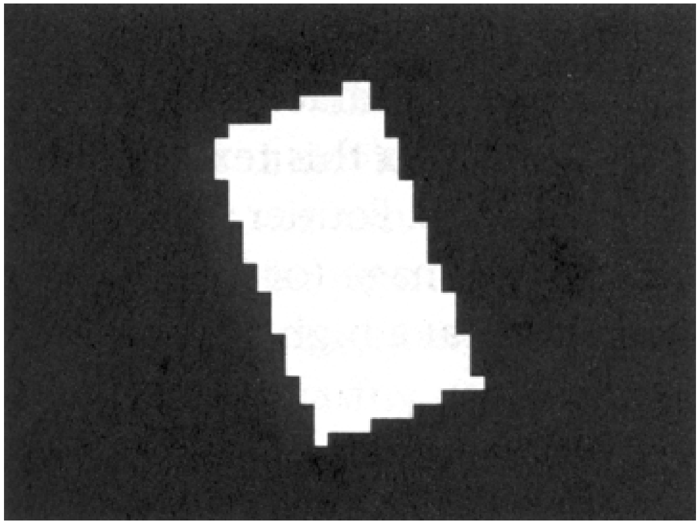
- Sort is quite fast, because AEL is usually almost in order.
- Nonetheless, OpenGL limits to convex polygons, so two and only two elements in AEL at any time, and no sorting.
- With additional logic to keep track of what color to use, can fill in many polygons at a time.
- Can be done *without* floating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

Dodging floating point

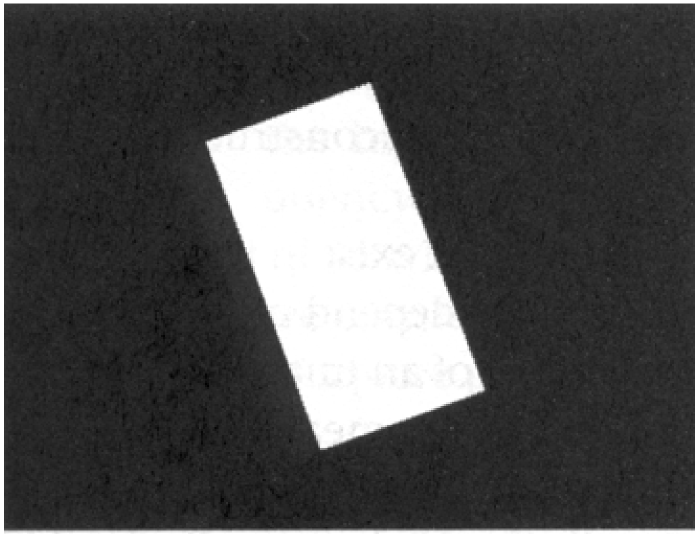
- for edge, 1/m=Dx/Dy, which is a rational number.
- store x as x_int, x_num, x_denom=Dy
- then x->x+1/m is given by:
  - x_num=x_num+Dx
  - if x_num >= x_denom
    - x_int=x_int+1
    - x_num=x_num-x_denom
- Advantages:
  - no floating point
  - can tell if x is an integer or not (check x_num=0), and 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 by smoothing and sampling
- Problem with “slivers” (page 90) is really an aliasing problem.



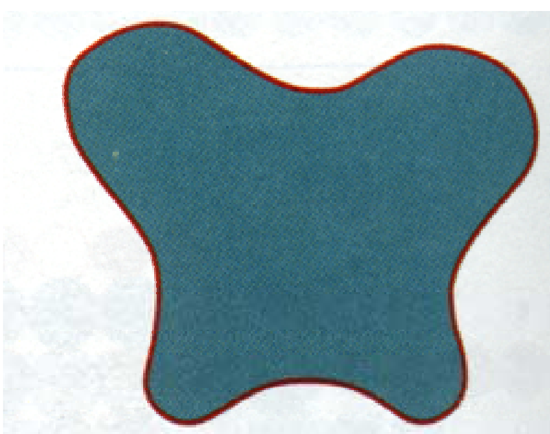
Aliasing

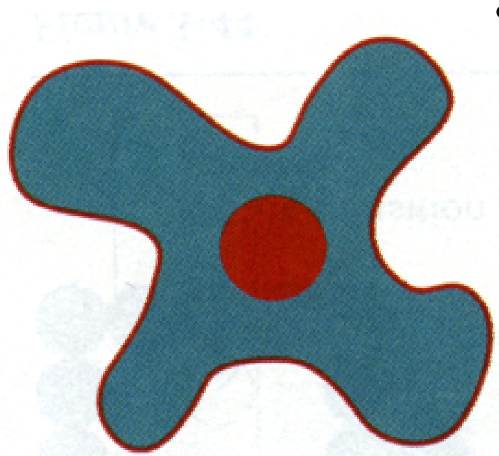


Ideal

Boundary fill

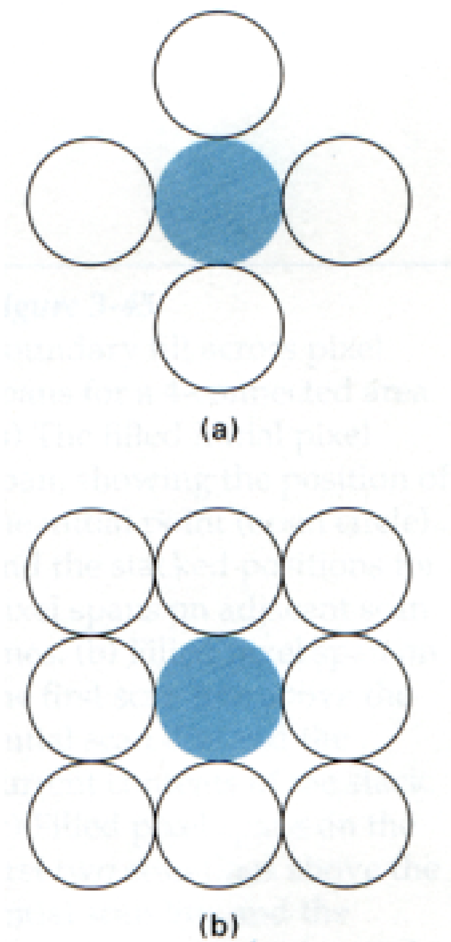
- Basic idea: fill in pixels inside a boundary





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

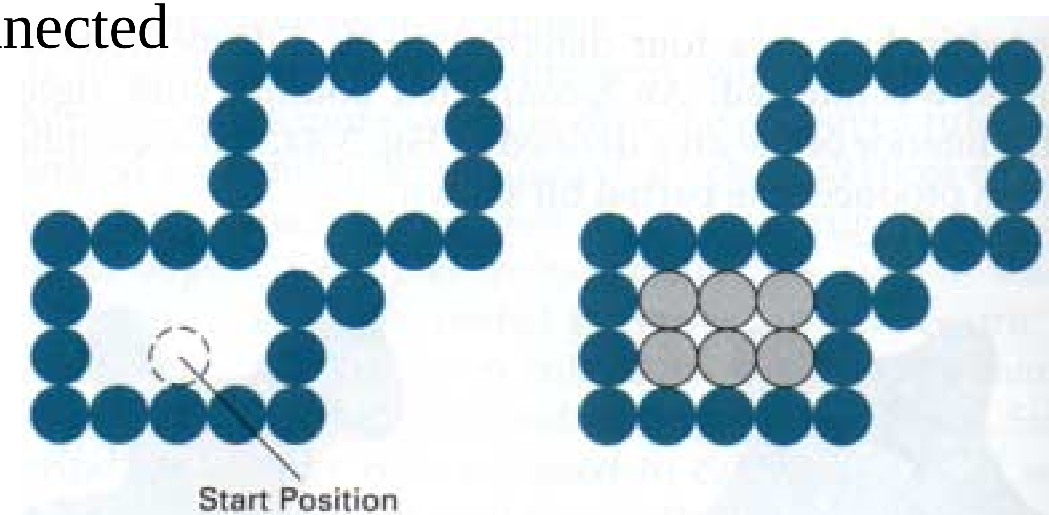
Choice of neighbours is important



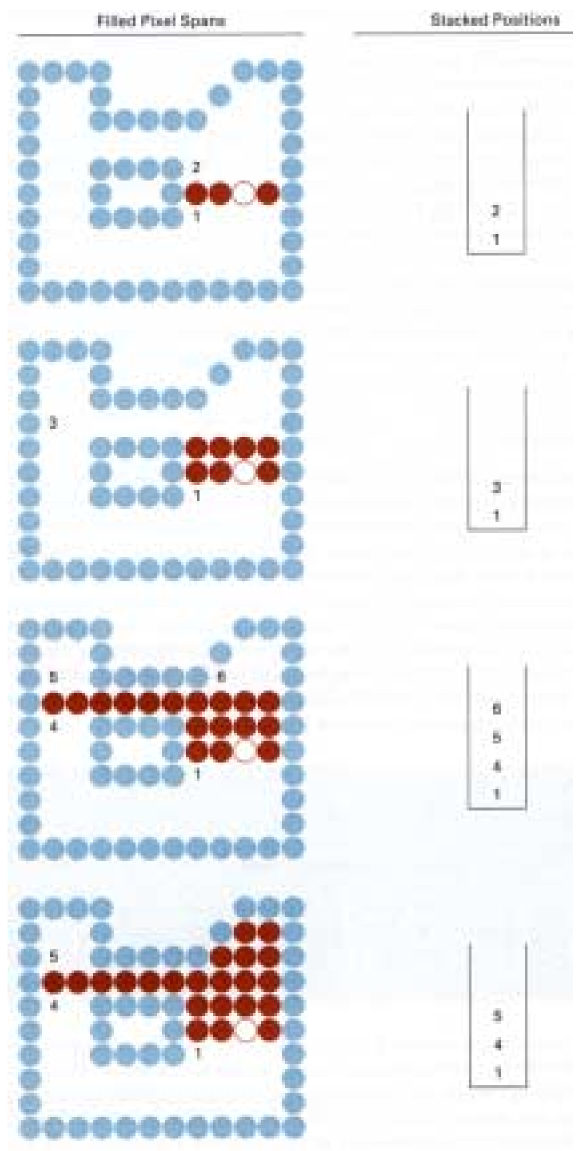
4-connected

8 connected

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



- Using spans for boundary fill means a less messy stack.

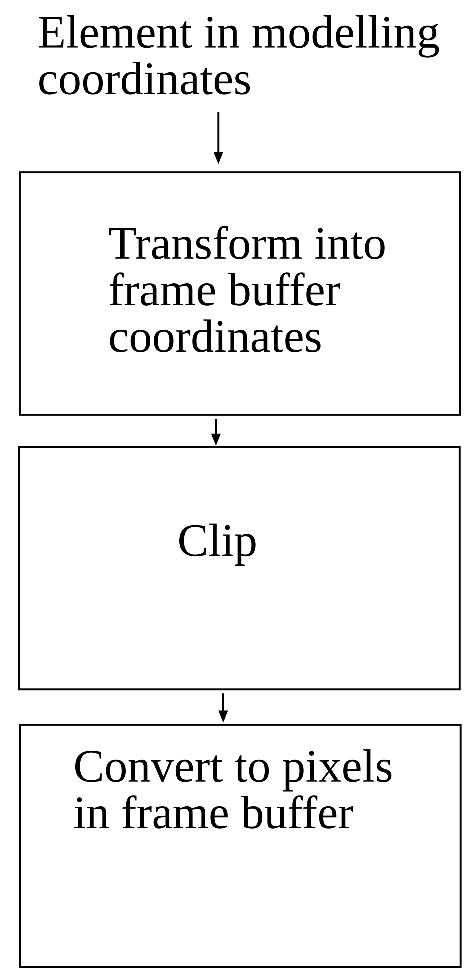


Pattern fill

- Use index into screen as index into pattern

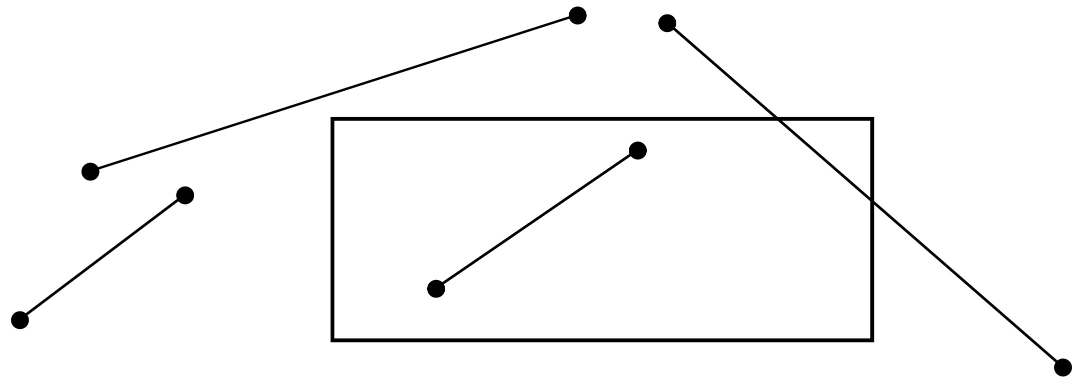
Clipping

- 2D elements are laid out in a convenient coordinate system--perhaps km for a map--and then transformed to a frame buffer coordinate system.
- Objects that are to be drawn must lie inside frame buffer, and may have to lie inside particular region - e.g. viewport.
- We may also want to dodge additional expensive operations on objects or parts of objects that won’t be displayed.
- How do we ensure line/polygon lies inside a region?

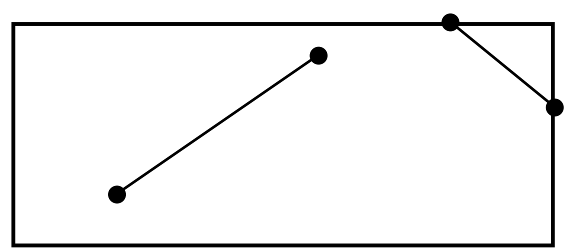


Clipping lines against rectangles

Have

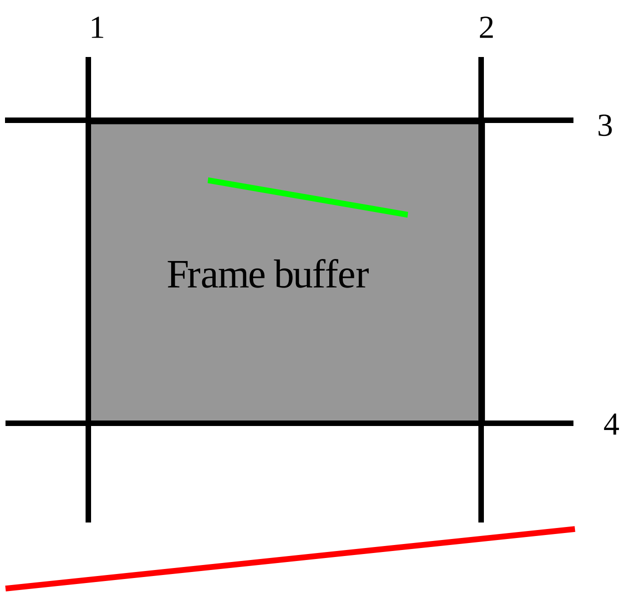


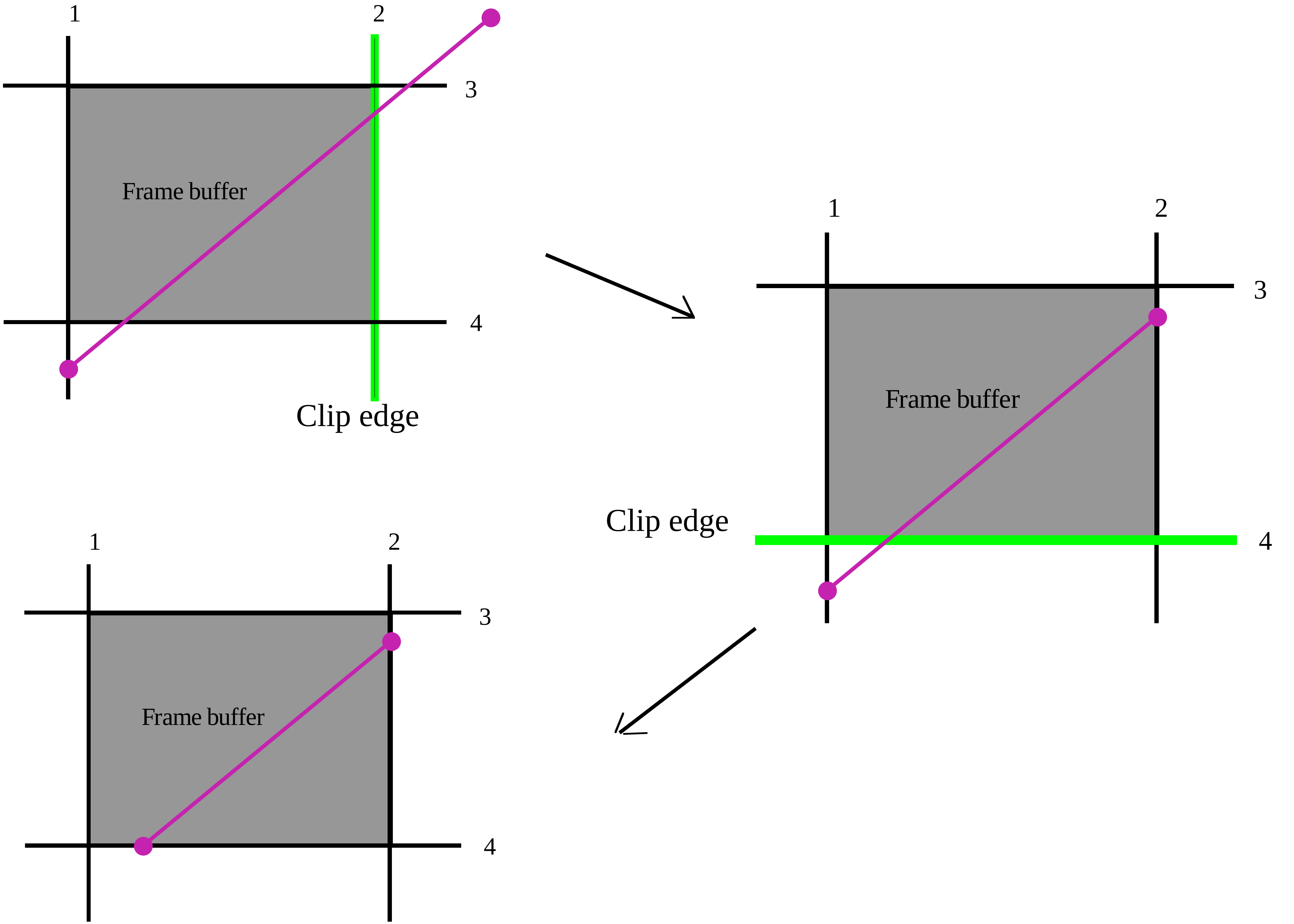
Compute



Cohen-Sutherland clipping (lines)

- Clip line against convex region.
- For each edge of the region, clip line against that edge:
  - line all on wrong side of some edge? throw it away (trivial reject--e.g. red line with respect to bottom edge)
  - line all on right side of *all* edges? doesn't need clipping (trivial accept--e.g. green line).
  - line crosses edge? replace endpoint on wrong side with crossing point.




1
2
3
4
Frame buffer
Clip edge
1
2
3
4
Frame buffer
Clip edge
1
2
3
4
Frame buffer

Cohen Sutherland - details

- Only need to clip line against edges where one endpoint is outside.
- The state (e.g., in or out) of that endpoint changes due to clipping--need to track this.
- Use “outcode” to record endpoint in/out wrt each edge. One bit per edge, 1 if out, 0 if in.

- Trivial reject:
  - outcode(p1)&outcode(p2)!=0
- Trivial accept:
  - outcode(p1)|outcode(p2)==0
- Clipping line against edge is easy: e.g line has endpoints (x_s, y_s) and (x_e, y_e), clip against x=a gives the point: $(a, y_s+(a - x_s)((y_e - y_s)/(x_e - x_s))$

Cohen Sutherland - Algorithm

- Compute outcodes for endpoints
- While not trivial accept and not trivial reject:
  - clip against a problem edge (i.e. one for which an outcode bit is not 0)
  - compute outcodes again
- Return appropriate data structure

Cyrus-Beck/Liang-Barsky clipping

- Parametric clipping - view line in parametric form and reason about the parameter values
- More efficient, as not computing the coordinate values at irrelevant vertices
- Line is: $$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + u \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix}$$
- Clipping conditions on parameter:

$$x_{\min} \le x_1 + u\Delta x \le x_{\max}$$
$$y_{\min} \le y_1 + u\Delta y \le y_{\max}$$

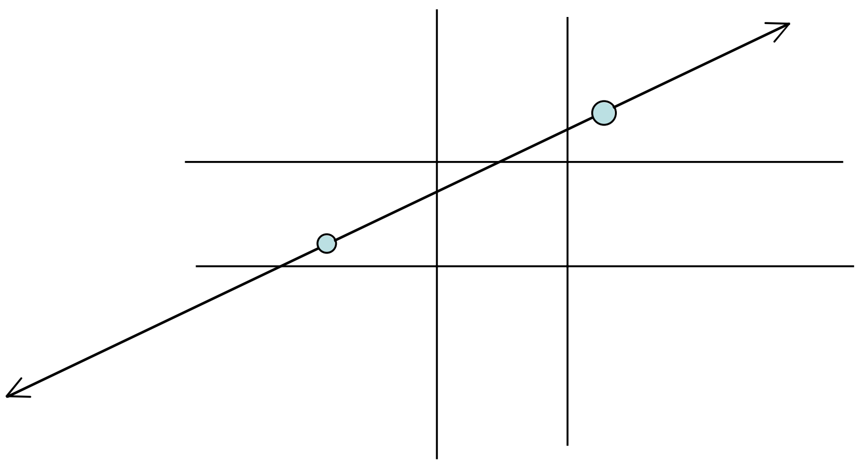
Cyrus-Beck/Liang-Barsky--2

- Conditions become: $\quad u = \dfrac{q_k}{p_k}$ where
- when $p_k<0$, as u increases line goes from outside to inside
- when $p_k>0$, line goes from inside to outside
- infinite line:
  - parallel to an edge (in which case $p_k=0$ for some k, and clipping is easy)
  - or:
    - if there is a segment, parameter goes inside-inside-outside-outside.

$$
\begin{array}{ll}
p_1 = -\Delta x & q_1 = x_1 - x_{\min} \\
p_2 = \Delta x & q_2 = x_{\max} - x_1 \\
p_3 = -\Delta y & q_3 = y_1 - y_{\min} \\
p_4 = \Delta y & q_4 = y_{\max} - y_1
\end{array}
$$

Cyrus-Beck/Liang-Barsky--3

- Consider infinite line extension of segment. There are 3 cases:
  - parallel to an edge (in which case p_k=0 for some k, and clipping is easy)
  - no intersection with the clipping rectangle
  - or, we go inside-inside-outside-outside (must get inside a corner, and leave the opposite corner).



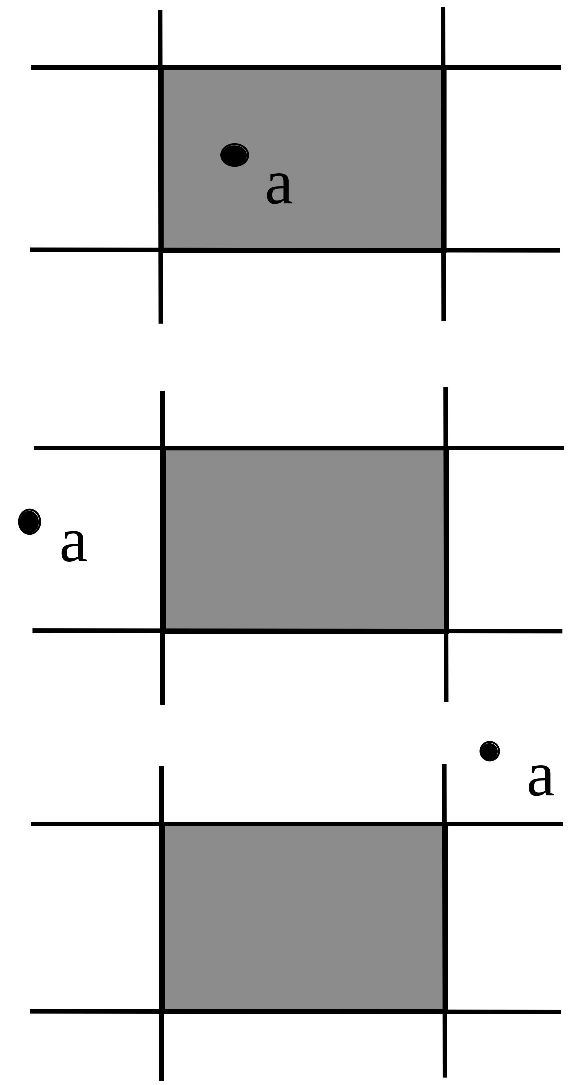
  - but to be on the *segment*, we need $0 \leq u \leq 1$

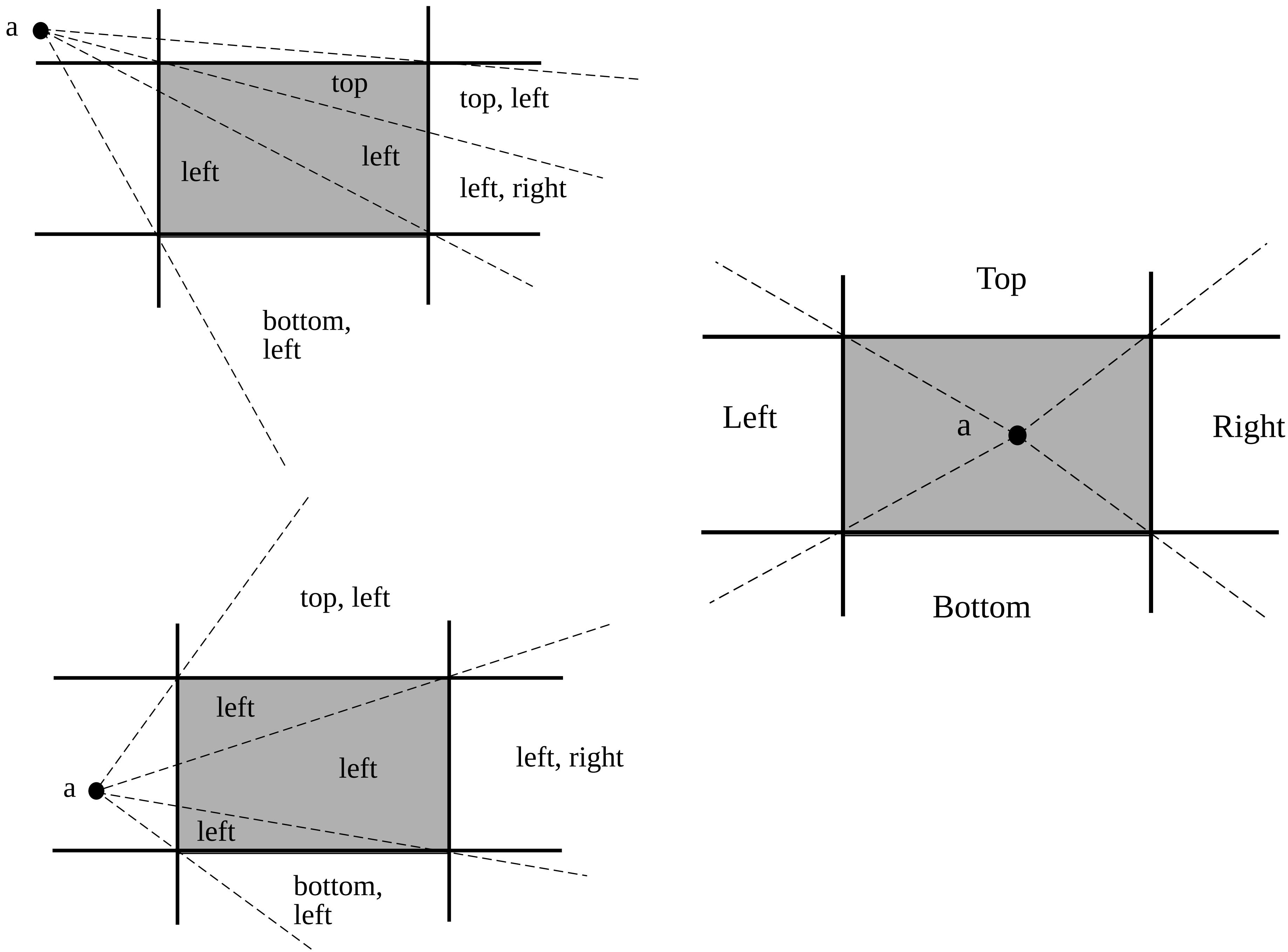
Cyrus-Beck/Liang-Barsky--Algorithm

- compute incoming u values, which are q_k/p_k for each $p_k<0$
- compute outgoing u values, which are q_k/p_k for each $p_k>0$
- parameter value for small u end of line is:$u_{small}= \max(0,$ incoming values)
- parameter value for large u end of line is: $u_{large}=\min(1,$ outgoing values)
- if $u_{small}<u_{large}$, there is a line segment - compute endpoints by substituting u values.
- Improvement (Liang-Barsky):
  - identify some rejects early as u's are computed for each edge in turn

Nicholl-Lee-Nicholl clipping

- Some edges are irrelevant to clipping, particularly if one vertex lies inside region.
- Endpoints are: a, b
- Cases:
  - a inside
  - a in edge region
  - a in corner region
- For each case, we generate specialised test regions for b, which use simple tests (slope, >, <), and tell which edges to clip against.
- Fast, but specialized




a
top
top, left
left
left
left, right
bottom,
left
Top
Left
a
Right
Bottom
top, left
left
left
left, right
a
left
bottom,
left