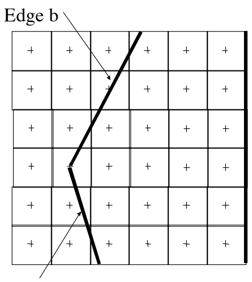
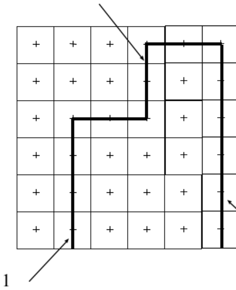
The next span - 2

- 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.

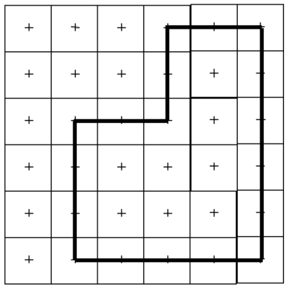


Filling in details -- 1

- 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
    - use parity rule
    - remember convention for integral x_{min} and x_{max}
    - integral top/bottom vertices have double entries
  - update each edge in AEL
    - x += (1/m)



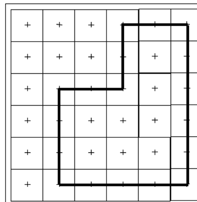
Compute the edge table (ET) to begin. Then fill polygon and update active edge list (AEL) row by row.

Format of edge entries

x	1/m	ymax

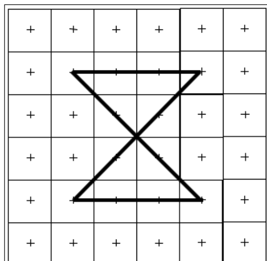
ET

ymin	
3	→ 3 | 0 | 5
2	
1	
0	→ 1 | 0 | 3 → 5 | 0 | 5



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



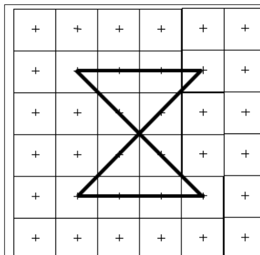
Format of edge entries

x	1/m	ymax

ET

ymin	
4	
3	
2	
1	→ 1 | 1 | 4 → 4 | -1 | 4
0	

(This is all there is in the ET---why?)



AEL just before filling

Row=4	
Row=3	2 | -1 | 4 → 3 | 1 | 4
Row=2	2 | 1 | 4 → 3 | -1 | 4
Row=1	1 | 1 | 4 → 4 | -1 | 4
Row=0	

Comments

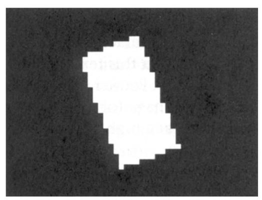
- Sort is quite fast, because AEL is usually almost in order.
- Nonetheless, OpenGL limits to convex polygons, so exactly two elements in AEL at any time, and no sorting.
- With additional logic to keep track of what color to use, can fill in many polygons at a time.
- Can be done *without* division/floating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

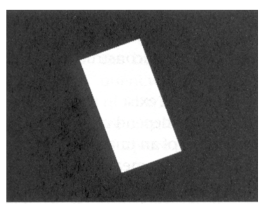
Dodging division and floating point

- 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 division/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

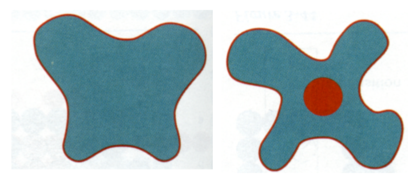
- Analogous to lines
- Anti-aliasing is done using graduated gray levels computed by smoothing and sampling
- Problem with "slivers" is really an sampling problem and is handled by filtering and sampling.


Aliasing

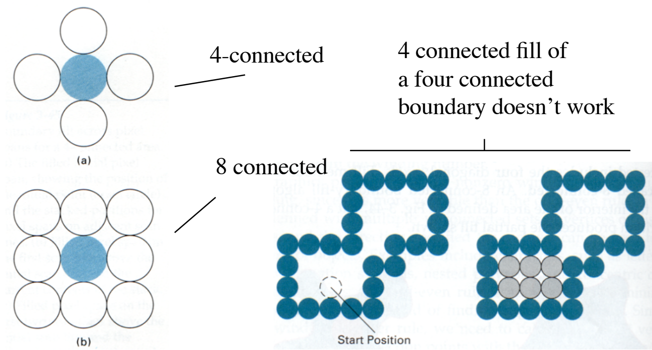

Ideal

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 recursively with all neighbours that are not boundary pixels



Choice of neighbours is important



Pattern fill

- Use coordinates as index into pattern

Clipping

- 2D elements are laid out in a convenient (often user based) coordinate system 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 want to dodge additional expensive operations on objects or parts of objects that won't be displayed.
- How do we ensure that the line/polygon lies inside a region?
- (Answer) Cut them up!

Clipping in the 2D pipeline

Element in modelling coordinates

↓

Transform into frame buffer coordinates

↓

Clip

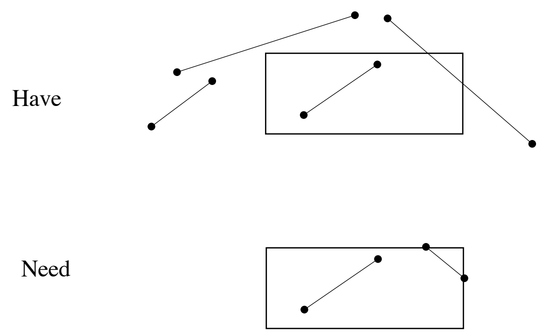
↓

Convert to pixels in frame buffer

Clipping references

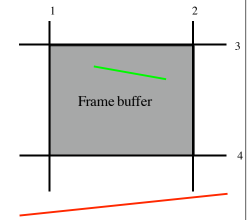
Hearn and Baker	Foley at al.
C-S (lines): p 317	C-S (lines): p 103
L-B (lines): p 322	L-B (lines): p 107
N-L (lines): p 325	N-L (lines): N.A.
S-H (poly): p 331	S-H (poly): p 112
W-A(poly): p 335	W-A(poly): N.A.

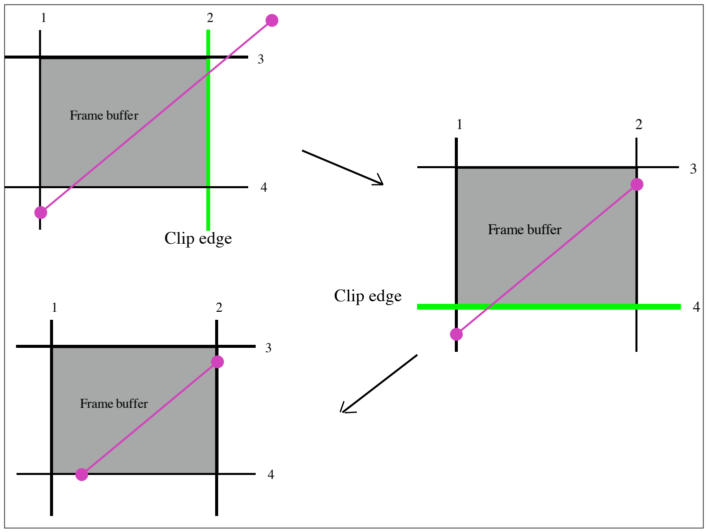
Clipping lines



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 any edge? throw it away (**trivial reject**--e.g. red line with respect to bottom edge)
  - line all on correct 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 (**clip**)

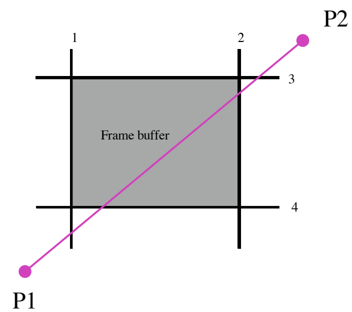




Cohen Sutherland - details

- Only need to clip line against edges where one endpoint is inside and one is outside.
- The state of the *outside* endpoint (e.g., in or out, w.r.t. a given edge) changes due to clipping as we proceed--need to track this.
- Use “outcode” to record endpoint in/out wrt each edge. One bit per clipping edge, 1 if out, 0 if in.

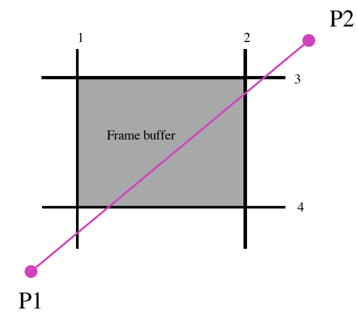
Outcode example



Outcode for P1?

Outcode for P2?

Outcode example



Outcode for P1?
[1 0 0 1]

Outcode for P2?
[0 1 1 0]

Note: As we process the four edges, the outcodes change

Cohen Sutherland - details

- Trivial reject
  - outcode(p1) & outcode(p2) != 0
- Trivial accept:
  - outcode(p1) | outcode(p2) == 0
- Clipping line against vertical/horizontal edge is easy:
  - line has endpoints (x_s, y_s) and (x_e, y_e)
  - e.g. (vertical case) clip against x=a gives the point
    $(a,\ y_s+(a - x_s)((y_e - y_s)/(x_e - x_s)))$
  - new point replaces the point for which outcode() is true
- Algorithm is valid for any convex clipping region (intersections are slightly more difficult)

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 1)
  - compute outcodes again
- Return appropriate data structure