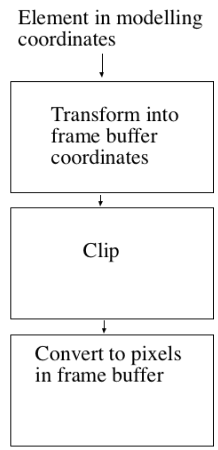
Clipping

- 2D elements are laid out in a convenient (often user based) 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 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 in the 2D pipeline

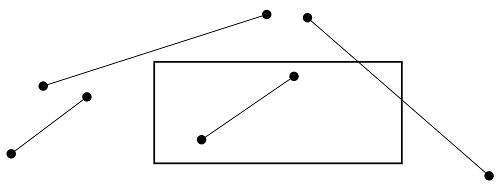


Clipping references

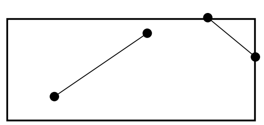
Hearn and Baker	Foley at al.
C-S (lines): p 317	C-S (lines): p 103
L-B (lines): p 322	L-B (lines): p 107
N-L (lines): p 325	N-L (lines): N.A.
S-H (poly): p 331	S-H (poly): p 112
W-A(poly): p 335	W-A(poly): N.A.

Clipping lines

Have

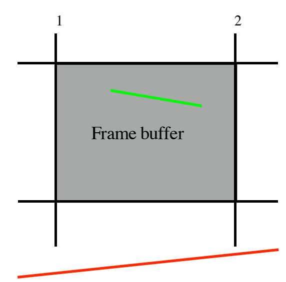


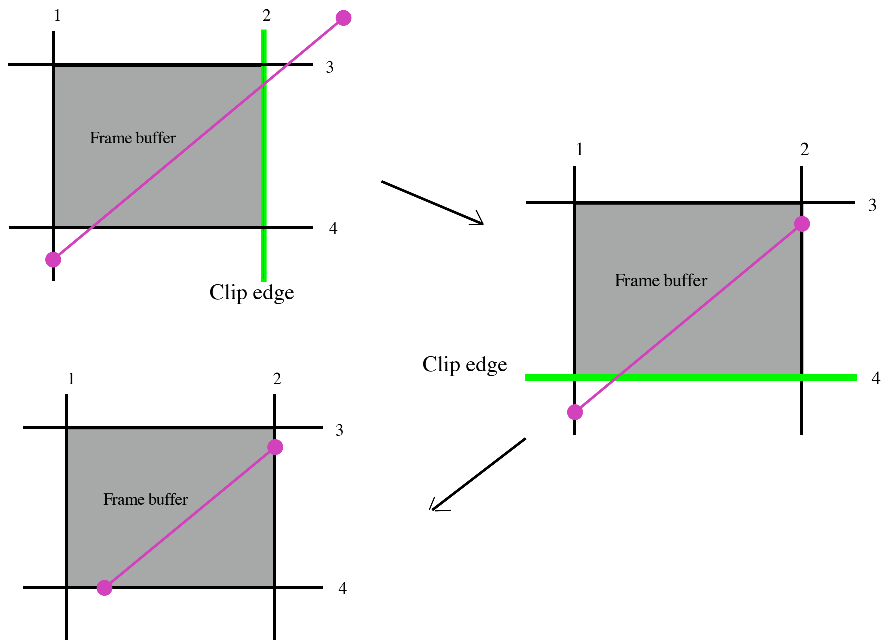
Need



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**)

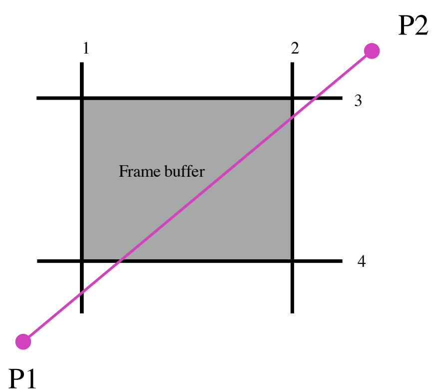




Cohen Sutherland - details

- 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?

Cohen Sutherland - details

- Trivial reject condition?
- Trivial accept condition?
- 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?
  - 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

Cyrus-Beck/Liang-Barsky clipping

- Parametric clipping: consider line in parametric form and reason about the parameter values
- More efficient, as we don't compute the coordinate values at irrelevant vertices
- Line is:

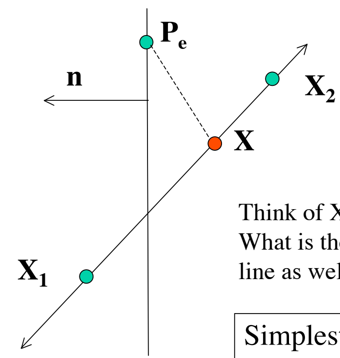
$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + t \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix}$$

$$\Delta x = x_2 - x_1$$
$$\Delta y = y_2 - y_1$$

Cyrus-Beck/Liang-Barsky clipping

- Consider the parameter values, t, for each clip edge
- Only t inside (0,1) is relevant
- Assumptions
  - $\mathbf{X}_1$!= $\mathbf{X}_2$
  - Ignore case where line is parallel to a clip edge (has no effect, but would lead to divide by zero).
  - We have a normal, **n**, for each clip edge pointing outward
  - For axis aligned rectangle (the usual case) these are?

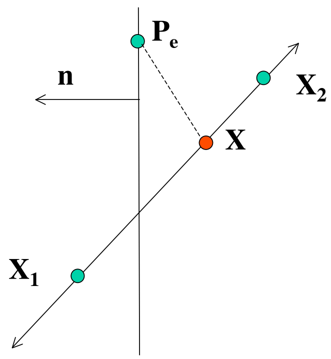
Computing t for intersection point



Think of X moving along the line shown. What is the condition that it is on the other line as well (i.e., intersects?)

Simplest to work from condition ($\mathbf{X}$(t)-$\mathbf{P}_e$)•n=0

Computing t for intersection point



Set

$$\mathbf{D} = \mathbf{X}_2 - \mathbf{X}_1$$

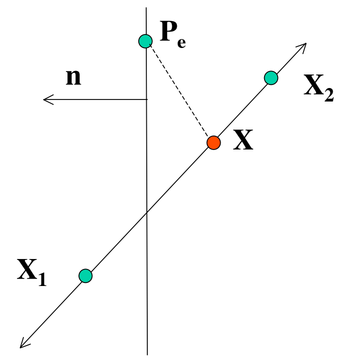
Then

$$\mathbf{X} = \mathbf{X}_1 + t\mathbf{D}$$

And condition is

$$(\mathbf{P}_e - (\mathbf{X}_1 + t\mathbf{D})) \bullet \mathbf{n} = \mathbf{0}$$

Computing t for intersection point, X



Condition

$$(\mathbf{P}_e - (\mathbf{X}_1 + tD)) \bullet \mathbf{n} = \mathbf{0}$$

Rearrange

$$(\mathbf{P}_e - \mathbf{X}_1) \bullet \mathbf{n} = t\mathbf{D} \bullet \mathbf{n}$$

And solve

$$t = \frac{(\mathbf{P}_e - \mathbf{X}_1) \bullet \mathbf{n}}{\mathbf{D} \bullet \mathbf{n}}$$

Computing t for intersection point

From previous slide $t = \frac{(\mathbf{P}_e - \mathbf{X}_1) \bullet \mathbf{n}}{\mathbf{D} \bullet \mathbf{n}}$

This simplifies greatly for axis aligned rectangles

Consider left edge. Now **n**=? and $\mathbf{P_e}$=?

And t = **?**

- All four special cases can expressed by: $t = \frac{q_k}{p_k}$
- Where

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

- Faster derivation for this special case?

- All four cases can expressed by: $t = \dfrac{q_k}{p_k}$
- Where

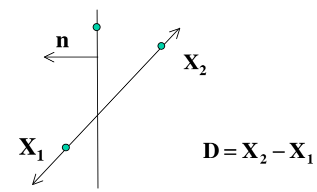
$$\begin{aligned}
p_1 &= -\Delta x & q_1 &= x_1 - x_{\min} \\
p_2 &= \Delta x & q_2 &= x_{\max} - x_1 \\
p_3 &= -\Delta y & q_3 &= y_1 - y_{\min} \\
p_4 &= \Delta y & q_4 &= y_{\max} - y_1
\end{aligned}$$

- One can also get this special case **directly** by solving:

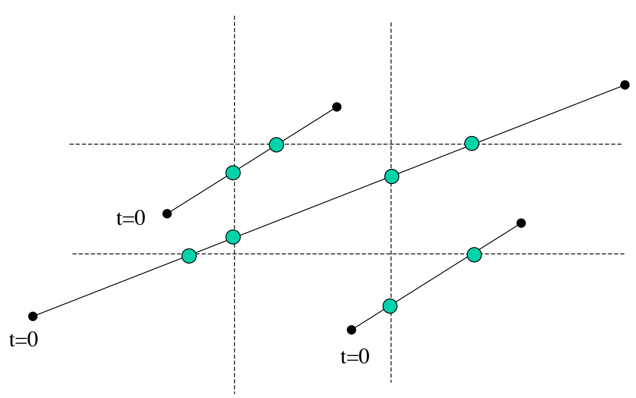
$$x_{\min} \le x_1 + t\Delta x \le x_{\max}$$
$$y_{\min} \le y_1 + t\Delta y \le y_{\max}$$

Cyrus-Beck/Liang-Barsky (cont)

- Next step: Use the t's to determine the clip points
- Recall that only t in (0,1) is relevant, but we need additional logic to determine clip endpoints from multiple t's inside (0,1).
- We imagine going from X1 to X2 and classify intersections as either potentially entering (PE) or potentially leaving (PL) if they go across a clip edge from outside in or inside out.
- This is easily determined from the sign of **D•n** which we have already computed.



PE vs PL example



Cyrus-Beck/Liang-Barsky--Algorithm

- Compute incoming (PE) *t* values, which are q_k/p_k for each $p_k<0$
- Compute outgoing (PL) *t* values, which are q_k/p_k for each $p_k>0$
- Parameter value for small *t* end of the segment is:

  t_{small}= max(0, incoming values)
- Parameter value for large *t* end of the segment is:

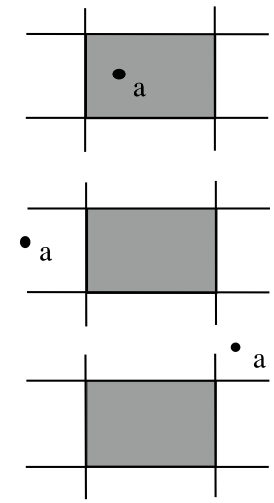
  t_{large}=min(1, outgoing values)
- If $t_{small}<t_{large}$, there is a segment portion in the clip window - compute endpoints by substituting these two *t* values (how)?
- Otherwise reject because it is outside.

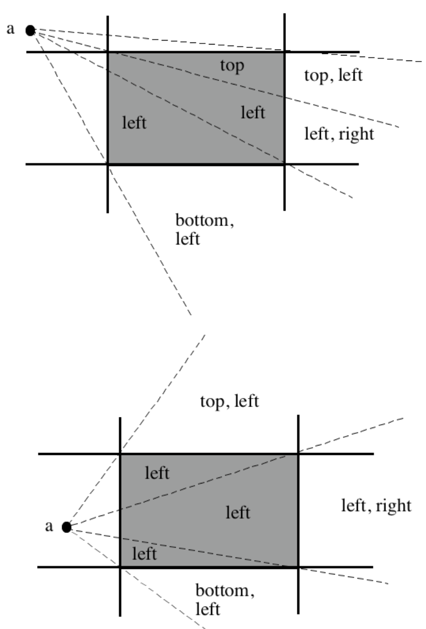
Cyrus-Beck/Liang-Barsky--Notes

- Works fine if clipping window is not an axis-aligned rectangle. Computing the *t* values is just more expensive.
- **Bibliographic note**: Original algorithm was Cyrus-Beck (close to what we have done here). A very similar algorithm was independently developed later by Liang-Barsky with some additional improvements for identifying early rejects as the *t* values are computed.

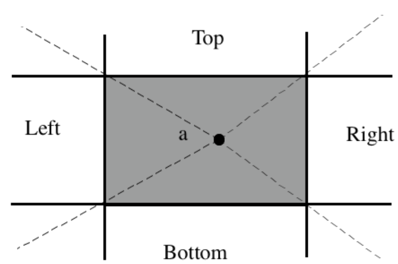
Nicholl-Lee-Nicholl clipping

- Fast specialized method
- We will just outline the basic idea
- Consider segment with endpoints: a, b
- Cases:
  - a inside
  - a in edge region
  - a in corner region
- For each case, we generate specialized test regions for b
- Which region b is in is determined by simple“which-side” tests.
- The region b is in determines which edges need to be clipped against.
- Speed is enhanced by good ordering of tests, and caching intermediate results

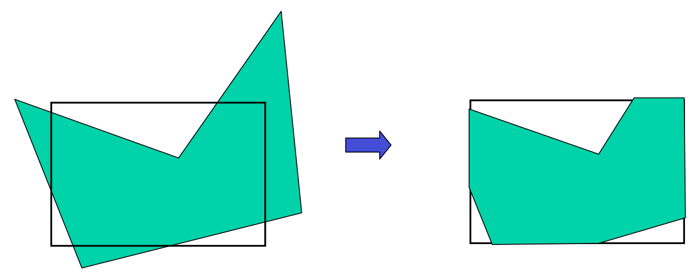




NLN clipping: Compute the area that b is in, and clip the segment ab against the edges specified.



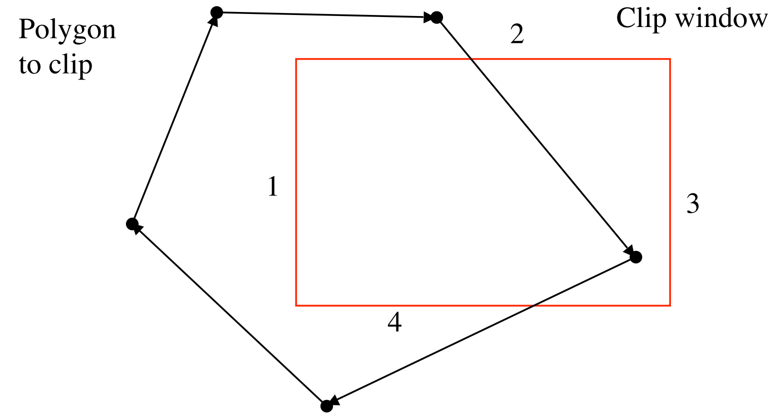
Polygon clip (against convex polygon)



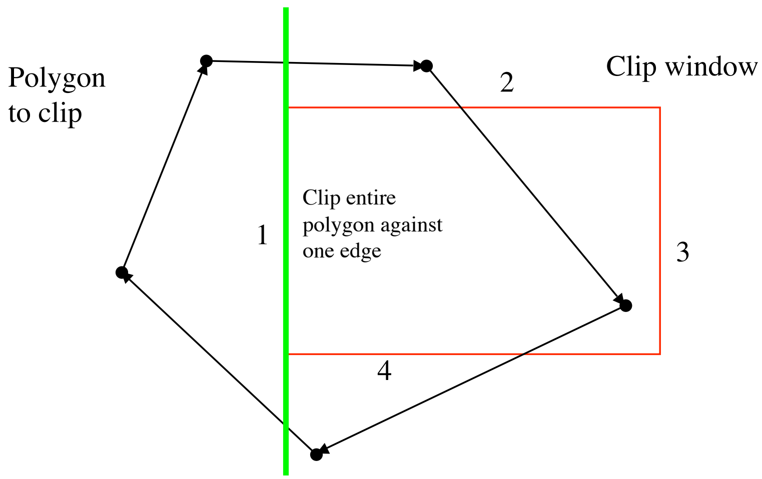
Sutherland-Hodgeman polygon clip

- Recall: polygon is convex if any line joining two points inside the polygon, also lies inside the polygon; implies that a point is inside if it is on the right side of each edge.
- Clipping each edge of a given polygon doesn't make sense - how do we reassemble the pieces? We want to arrange doing so on the fly.
- Clipping the polygon against each edge of the clip window in *sequence* works if the clip window is *convex*.
- (Note similarity to Sutherland-Cohen line clipping)

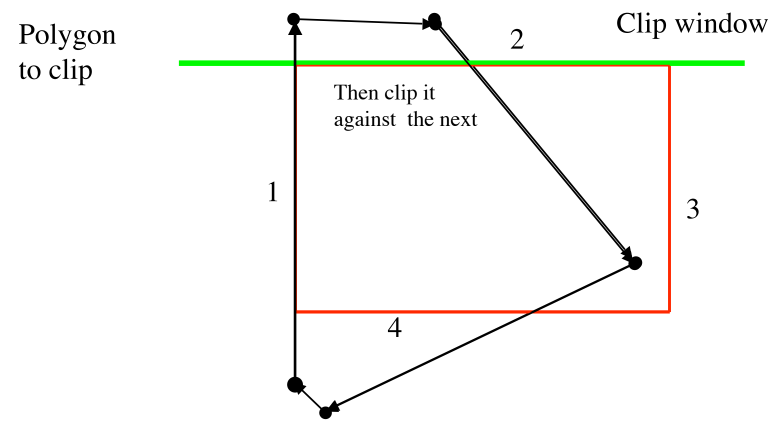
Sutherland-Hodgeman polygon clip



Sutherland-Hodgeman polygon clip

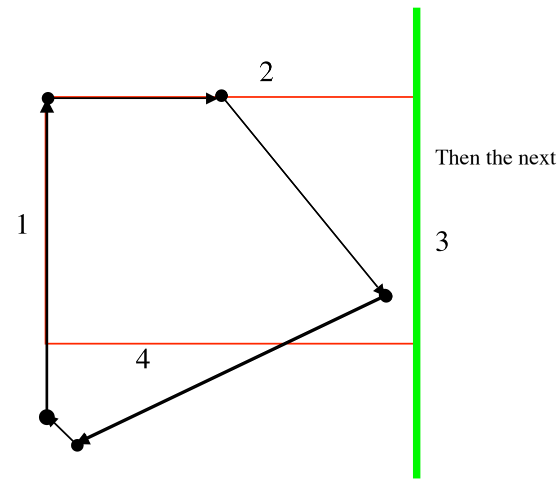


Sutherland-Hodgeman polygon clip



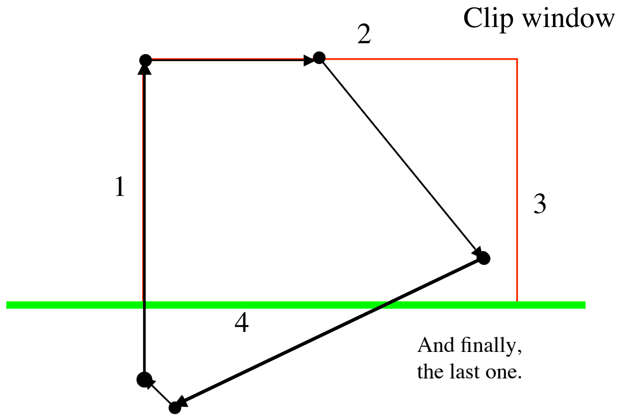
Sutherland-Hodgeman polygon clip

Polygon
to clip



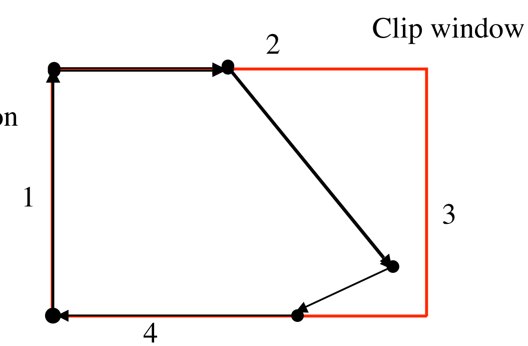
Sutherland-Hodgeman polygon clip

Polygon
to clip



Sutherland-Hodgeman polygon clip

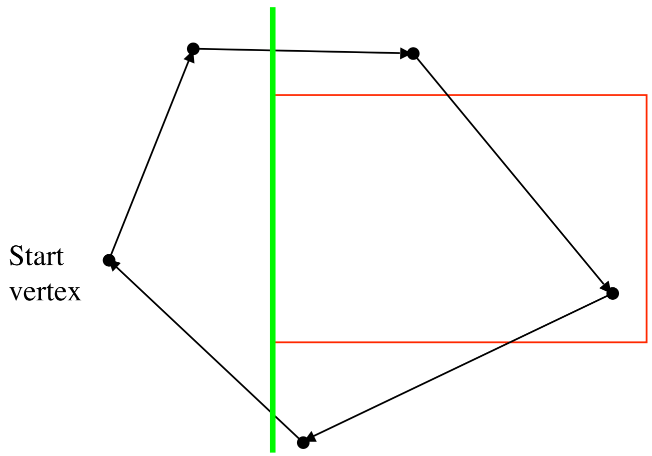
Clipped polygon



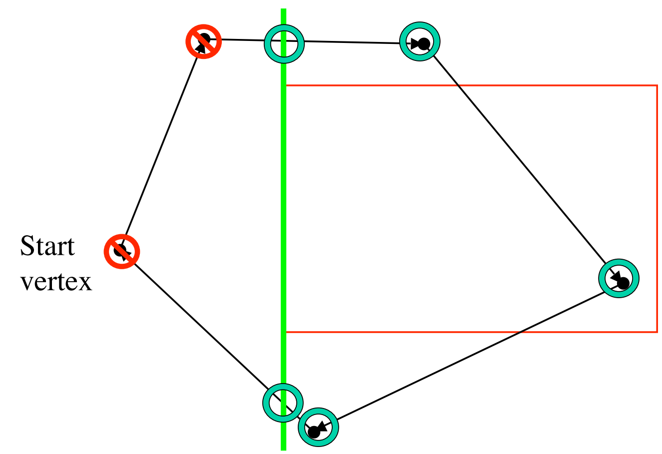
Clipping against current clip edge

- Polygon is a list of vertices
- Think of process as rewriting polygon, vertex by vertex
- Check start vertex
  - in - emit it
  - out - ignore it
- Walk along vertices and for each edge consider four cases and apply corresponding action.

- Four cases:
  - polygon edge crosses clip edge going from out to in
    -
  - polygon edge crosses clip edge going from in to out
    -
  - polygon edge goes from out to out
    -
  - polygon edge goes from in to in
    -

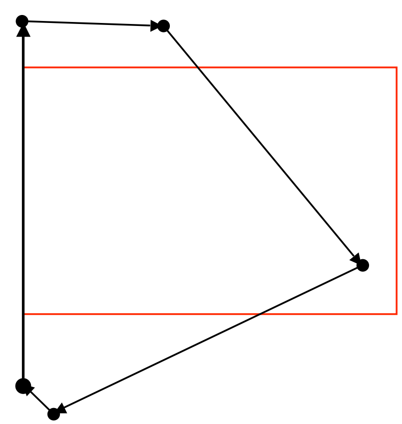


polygon edge crosses clip edge going from out to in ==> emit crossing, next vertex
polygon edge crosses clip edge going from in to out ==> emit crossing
polygon edge goes from out to out ==> emit nothing
polygon edge goes from in to in ==> emit next vertex



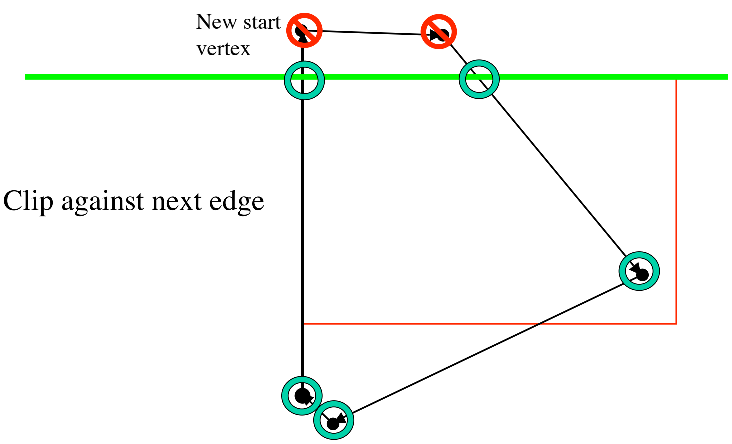
polygon edge crosses clip edge going from out to in ==> emit crossing, next vertex
polygon edge crosses clip edge going from in to out ==> emit crossing
polygon edge goes from out to out ==> emit nothing
polygon edge goes from in to in ==> emit next vertex

Now have



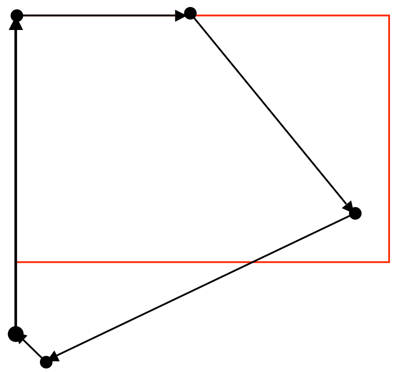
polygon edge crosses clip edge going from out to in ==> emit crossing, next vertex
polygon edge crosses clip edge going from in to out ==> emit crossing
polygon edge goes from out to out ==> emit nothing
polygon edge goes from in to in ==> emit next vertex

Clip against next edge



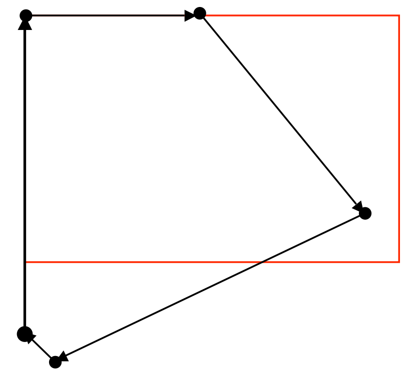
polygon edge crosses clip edge going from out to in ==> emit crossing, next vertex
polygon edge crosses clip edge going from in to out ==> emit crossing
polygon edge goes from out to out ==> emit nothing
polygon edge goes from in to in ==> emit next vertex

Now have



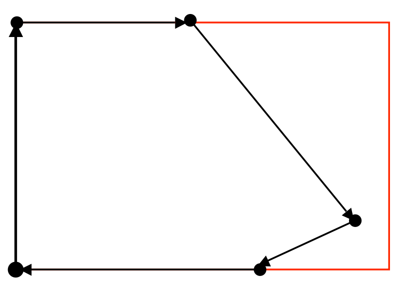
polygon edge crosses clip edge going from out to in ==> emit crossing, next vertex
polygon edge crosses clip edge going from in to out ==> emit crossing
polygon edge goes from out to out ==> emit nothing
polygon edge goes from in to in ==> emit next vertex

Clipping against next edge (right) gives



polygon edge crosses clip edge going from out to in ==> emit crossing, next vertex
polygon edge crosses clip edge going from in to out ==> emit crossing
polygon edge goes from out to out ==> emit nothing
polygon edge goes from in to in ==> emit next vertex

Clipping against final(bottom) edge gives



polygon edge crosses clip edge going from out to in ==> emit crossing, next vertex
polygon edge crosses clip edge going from in to out ==> emit crossing
polygon edge goes from out to out ==> emit nothing
polygon edge goes from in to in ==> emit next vertex

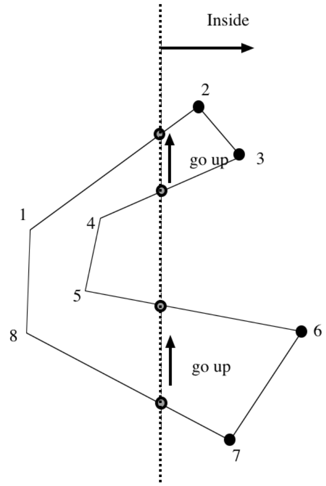
More Polygon clipping

- Notice that we can have a pipeline of clipping processes, one against each edge, each operating on the output of the previous clipper -- substantial advantage.
- Unpleasantness can result from concave polygons; in particular, polygons with empty interior.
- Can modify algorithm for concave polygons (e.g. Weiler Atherton)

Weiler Atherton

For clockwise polygon (starting outside):

- For out-to-in pair, follow usual rule
- For in-to-out pair, follow clip edge (clockwise) and then jump to next vertex (which is on the outside) and start again
- Only get a second piece if polygon is convex



Additional remarks on clipping

- Although everything discussed so far has been in terms of polygons/lines clipped against lines in 2D, all - except Nicholl-Lee-Nicholl - will work in 3D against convex regions without much change.
- This is because the central issue in each algorithm is the inside outside decision as a convex region is the intersection of half spaces.
- Inside-outside decisions can be made for lines in 2D, planes in 3D. e.g testing dx•n>=0
- Hence, all (except N-L-N) can be used to clip:
  - Lines against 3D convex regions (e.g. cubes)
  - Polygons against 3D convex regions (e.g. cubes)
- NLN could work in 3D, but the number of cases increases too much to be practical.