Clipping

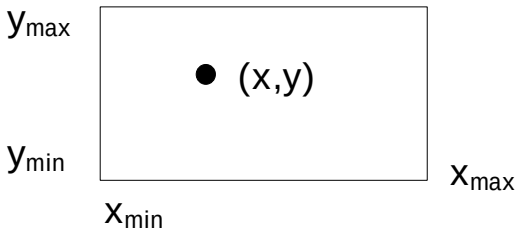
- Want to "clip" objects being drawn that protrude past the bounds of the display/window
- Why?
  - wrap-around (on older devices)
  - waste of time drawing objects that aren't going to be visible (inefficient)
- Can be accomplished
  - before scan conversion - analytically (ex: flt. Pt. Intersection)
  - during scan conversion (ex: scissoring)
  - after scan conversion - create the entire canvas in memory and display only visible part (ex: panning large static scenes)

- Scissoring
  - combining clipping and scan conversion
  - simplest, brute force technique
  - scan-convert entire primitive and write only visible pixels
  - can be fast in certain circumstances
    - if output primitive not >> than clip rectangle (faster to clip individual pixels than analytically)

Clipping Points

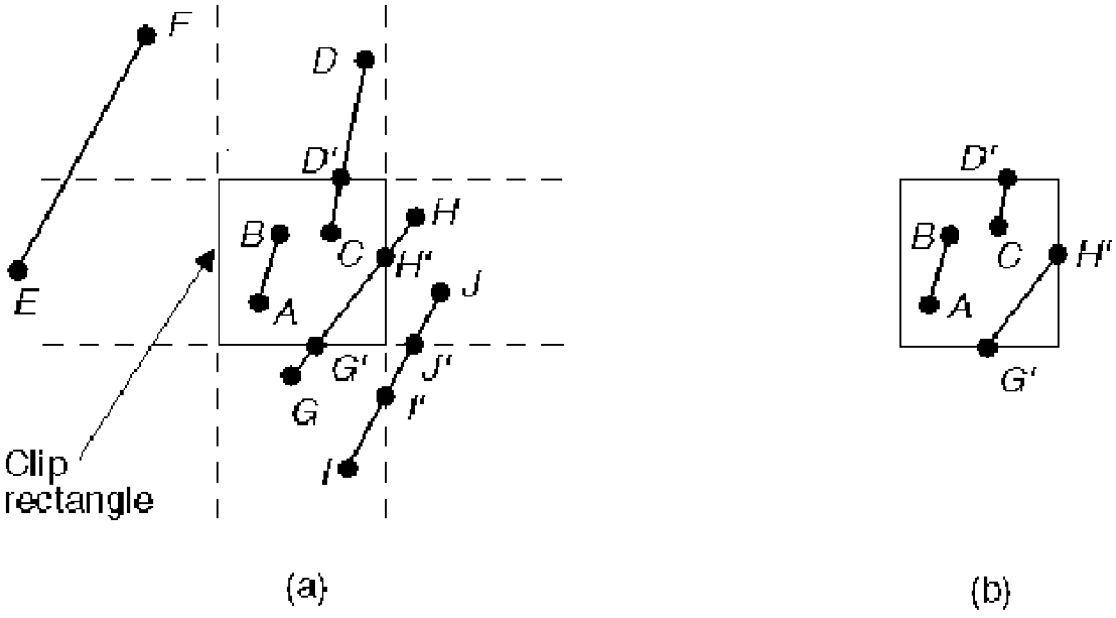
- for now, assume that clipping against rectangles.
- easy: $x_{min} \leq x \leq x_{max}$

  $y_{min} \leq y \leq y_{max}$



Clipping Lines

- lines are clipped to line segments
- can check whether a line needs to be clipped by looking at its endpoints

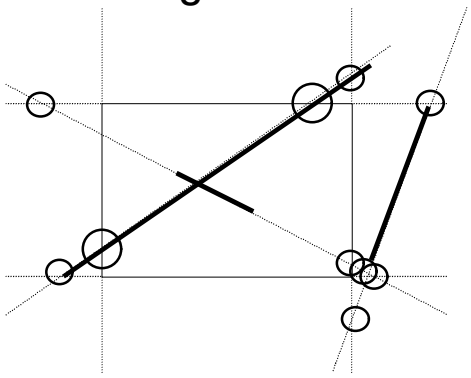


Observations

1. If the 2 endpoints of a line are within the clipping rectangle, then the line is completely inside (trivially accept) (AB)
2. If 1 endpoint is inside and the other is outside then MUST compute the pt, of intersection (CD)
3. If both endpoints are outside then the line may or may not be inside (EF, GH & IJ)

1. Solving Simultaneous Equations (brute force)

- intersect the line with each of the 4 clip edges (x_{min}, x_{max}, y_{min}, y_{max})
- test these intersection pts to see if inside the clipping rectangle
- can do this 2 ways
    1. assuming infinite lines (y=mx+b)



    2. deal with line segments - better
    - parametric formulation:

$$x = x_0 + t(x_1 - x_0) \qquad t \in [0,1]$$

$$y = y_0 + t(y_1 - y_0)$$

for endpoints (x_0, y_0), (x_1, y_1)

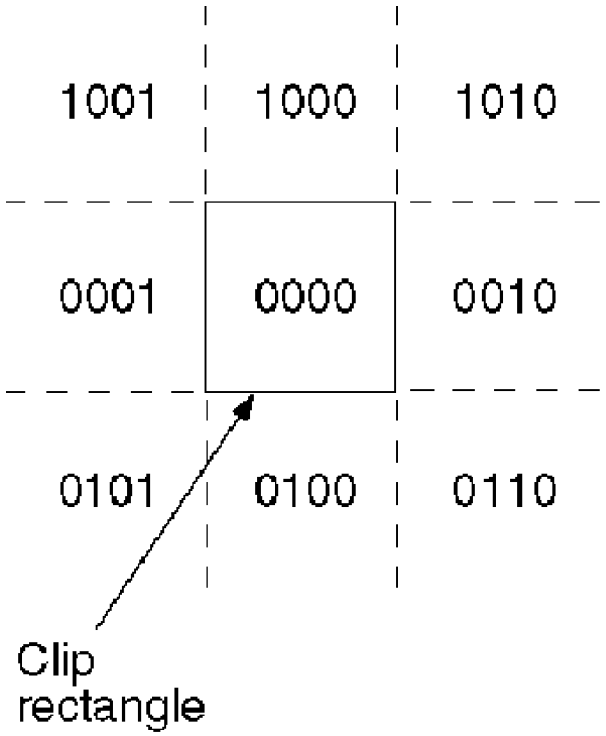
consider parametric equations for the lines (t_{line}) as well as the clipping edge (t_{edge})

- line: (x_0, y_0) to (x_1, y_1)
  $x = x_0 + t_{line}(x_1 - x_0)$
  $y = y_0 + t_{line}(y_1 - y_0)$
- clipping edge: (x_2, y_2) to (x_3, y_3)
  $x = x_2 + t_{edge}(x_3 - x_2)$
  $y = y_2 + t_{edge}(y_3 - y_2)$
- if (x,y) exists such that
  $t_{line} \in [0,1]$ and
  $t_{edg} \in [0,1]$
  then (x,y) lies on the intersection between the line and the clipping edge
- NOTE: will get many pts (x,y) if edge is a subset of the line (coincident), check for this.

BUT, both of these approaches are inefficient

Cohen-Sutherland Algorithm

- determines whether intersection calculations can be avoided by performing "region checks".
- assign a 4-bit code to the clipping rectangle AND the surrounding regions
- code ABRL (Above, Below, Right, Left)



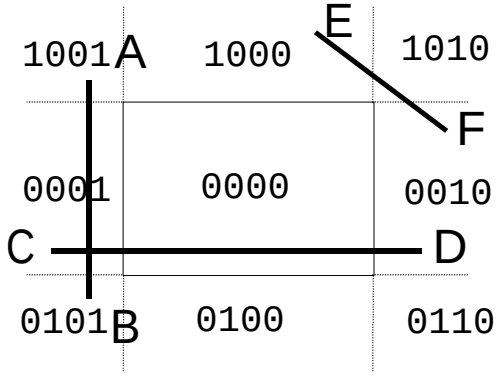
- easy to determine region of an endpoint (x,y)

  $y > y_{max}$ A $x > x_{max}$ R

  $y < y_{min}$ B $x < x_{min}$ L

Algorithm:

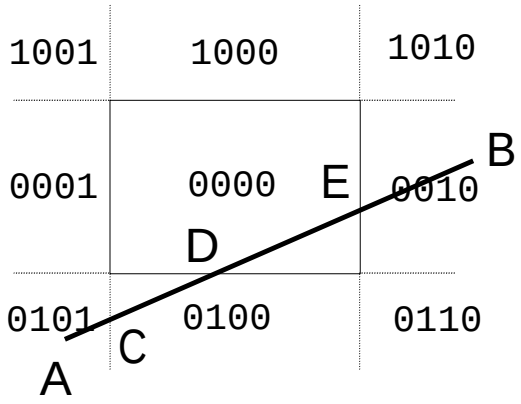
1. Calculate the code of the 2 line endpoints
2. Check for trivial acceptance (both codes = 0000)
3. Perform bitwise AND of codes
4. Trivially reject if result ≠ 0000.
5. Choose an endpoint outside the clipping rectangle. Test its code to determine which clip edge was crossed and find the intersection of the line and that clip edge (test the edges in a consistent order).
6. Replace endpoint (selected above) with intersection point
7. Repeat



A	1001	C	0001
B	0101	D	0010
	0001 Reject		0000 Intersect

E	1000
F	0010
	0000 Intersect

Intersection Example



A 0101

B 0010

0000 Intersection

- A is outside, intersection test yields C, replace A by C

C 0100

B 0010

0000 Intersection

- C replaced by D

D 0000

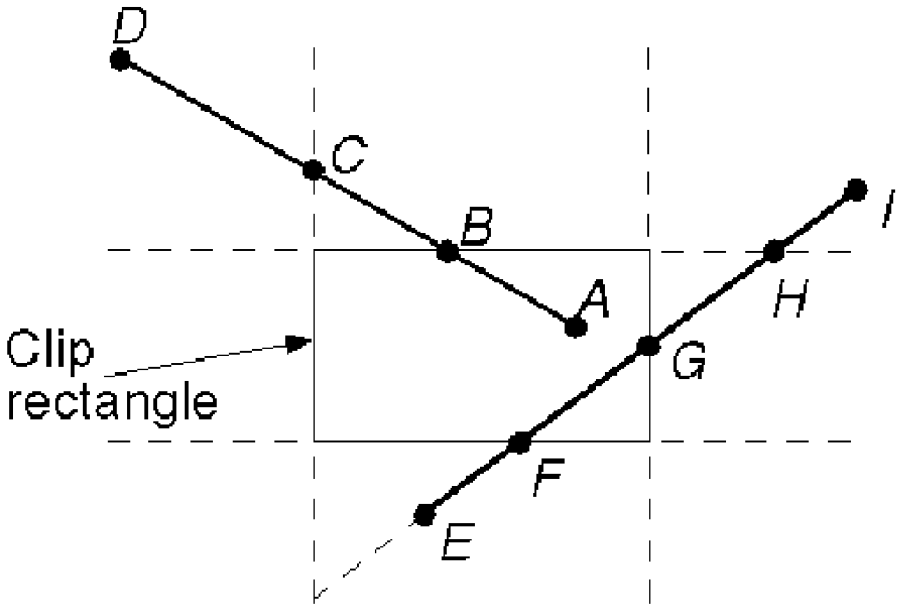
B 0010

0000 Intersection

- B replaced by E

E 0000

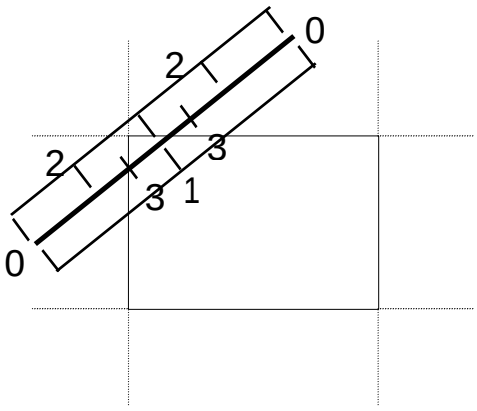
D 0000 Accept DE



- Note: order of computing intersection of line segments with clipping edges is either a predetermined consistent pattern, or dependant on code of endpoints.
- fairly efficient algorithm (widely used)
- pseudocode in text.

3. Midpoint Subdivision Variation

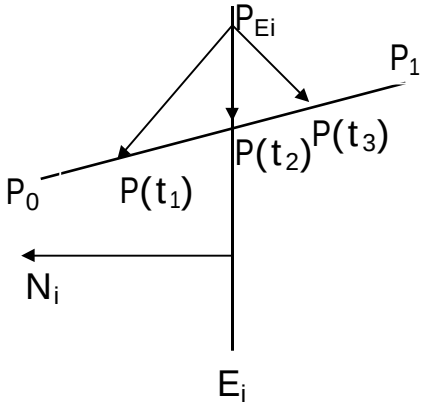
- intersection found by computing successive midpoints and testing resulting segments.



- test outcode of (00), then (01), (10)
  then (02), (21), (12), (20)
  then (03), (32), (23), …
- although seems inefficient, not bad since limited to screen resolution

Parametric Line-Clipping (Cyrus-Beck Algorithm)

- makes use of parametric representations
- intersection of line with clipping edge



line: $P(t)=P_0 + t(P_1-P_0)$
$t \in [0,1]$

- can distinguish region a point lies in by looking at dot product
  - $N_i \bullet [P(t_1) - P_{Ei}] > 0$ outside
  - $N_i \bullet [P(t_2) - P_{Ei}] = 0$ on edge
  - $N_i \bullet [P(t_3) - P_{Ei}] < 0$ inside

- to find the t value that corresponds to the intersection point $P(t_2)$ solve:

  $N_i \bullet [P(t) - P_{Ei}] = 0$

  $N_i \bullet [P_0 + t(P_1 - P_0) - P_{Ei}] = 0$

  $N_i \bullet [P_0 - P_{Ei}] + N_i \bullet [P_1 - P_0]t = 0$

  $t = (N_i \bullet [P_0 - P_{Ei}]) / (-N_i \bullet [P_1 - P_0])$

- can do this for each clip edge. (for rectangle you'll get 4 values of t)
- can trivially reject those $t \notin [0,1]$, since outside line P_0P_1
- but how to handle the rest?

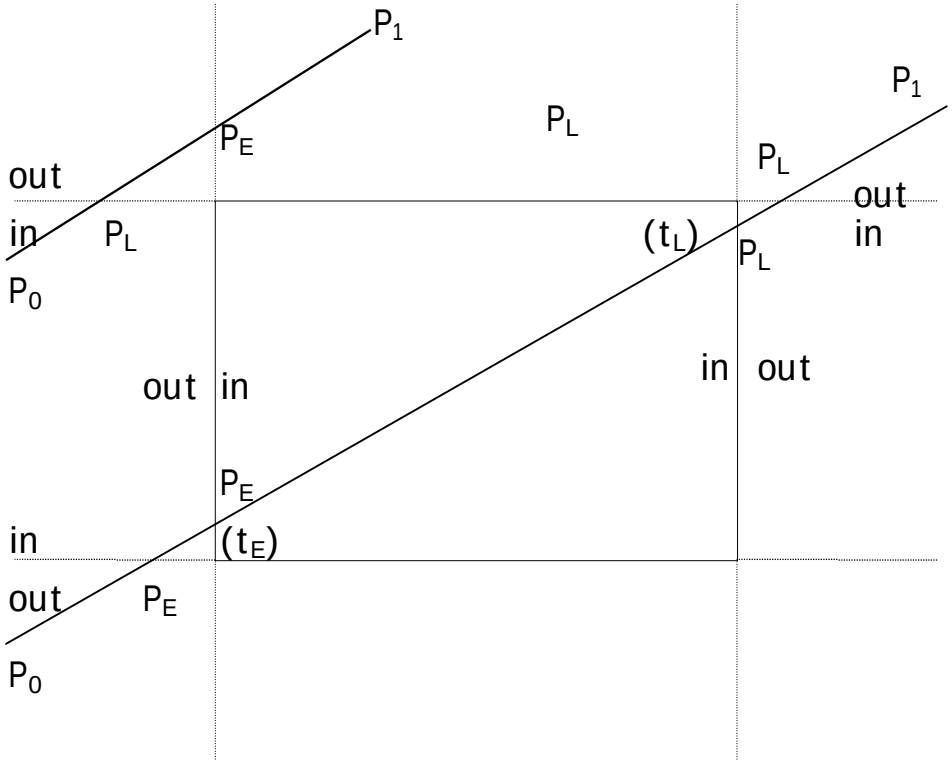
- characterize the intersections as:

  PE potentially entering (outside to inside)

  PL potentially leaving (inside to outside)

- intersecting segments will be composed of a PE-PL pair

  - idea
  - want PE with largest t (t_E)
  - PL with smallest t (t_L)
- intersecting segment in clipping plane is P(t), $t \in [t_E, t_L]$ (of course t_E & $t_L \in [0,1]$
- NOTE, if $t_E > t_L$, not inside the clip region
- pseudocode is in the text…



5. Liang-Barsky

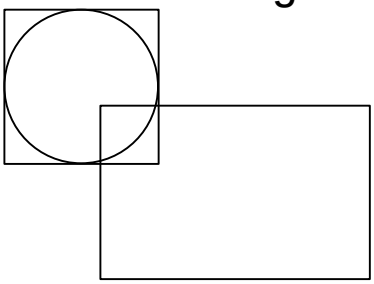
Cyrus-Beck and additional tests for rejection

Clipping Circles & Ellipses

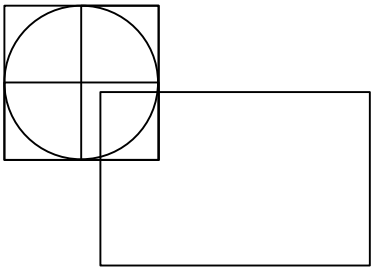
Circle

- trivial accept/reject test (quadtree bound aproach)

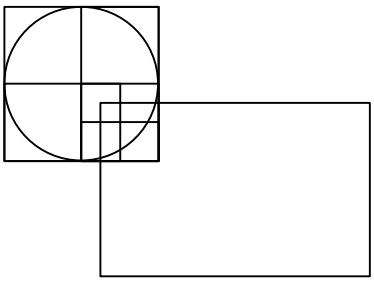
a./ test bounding box with clip region



b./ if intersects then check quadrants



c./ if intersects then check octants



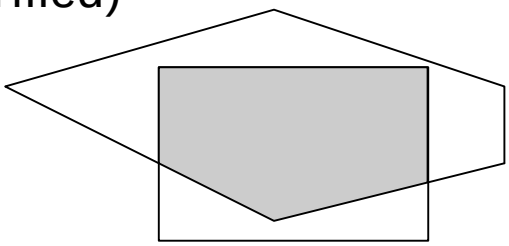
d./ compute the intersection analytically

e./ scan convert the resulting arcs (could scissor on a pixel by pixel basis).

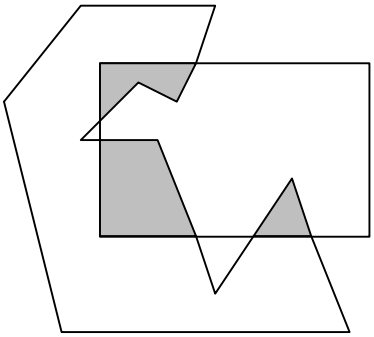
Ellipses: like circles but only down to quadrants, then scan convert

Clipping Polygons

- must test intersections of potentially many edges.
- want result to be a polygon(s) (since may be filled)



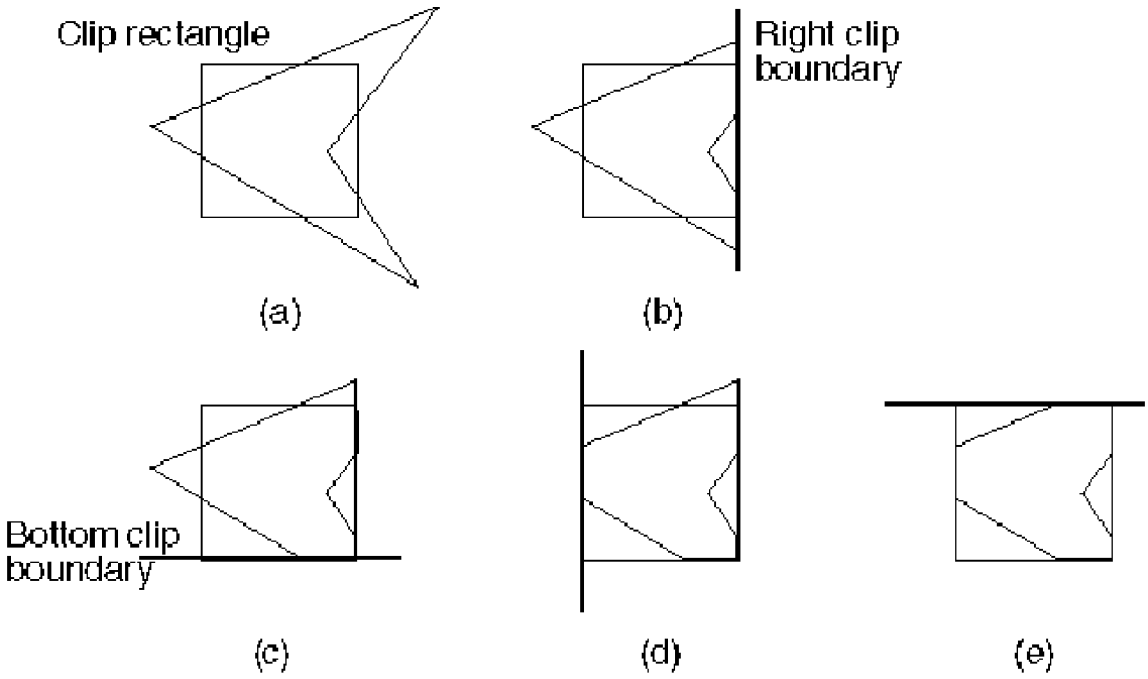
Clip of convex results in 1 polygon



Clip of concave polygon may result in several polygons

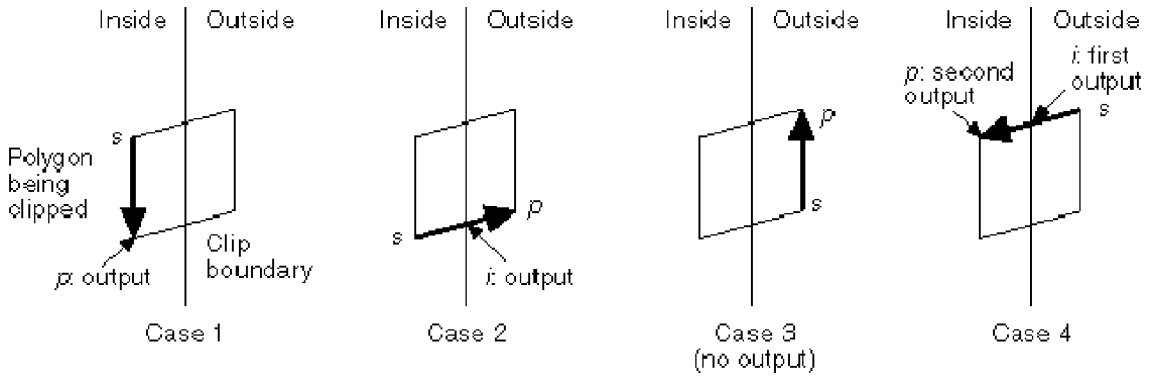
Sutherlan-Hodgman Algorithm

- divide & conquer strategy
  - subproblem: clip polygon against 1 clip edge
  - at each step, the partially clipped polygon is clipped against the next edge and so on…



- for each clipping edge
  - for each polygon edge
    - clip polygon edge to clip edge
    - store vertices (new) to new polygon

- 4 possible cases



- be careful of extra edges
- pseudocode in the text…