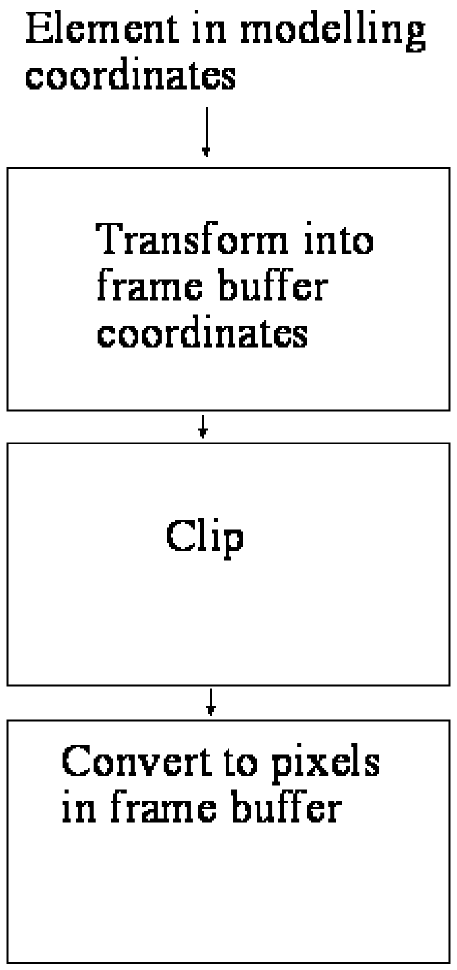
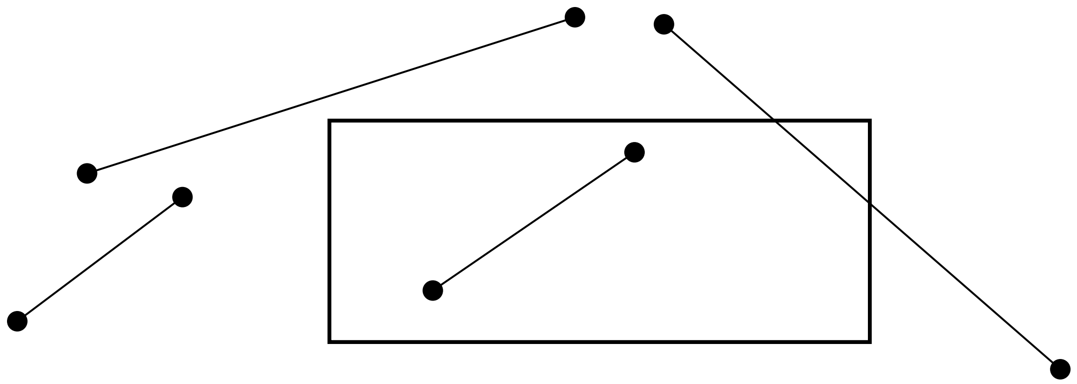
Clipping

- 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 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?

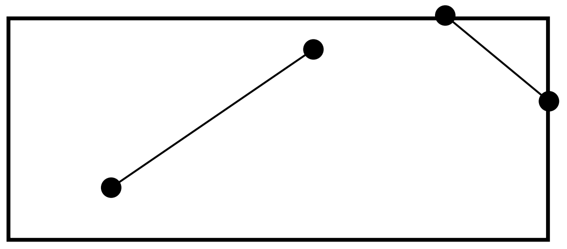

Element in modelling coordinates
Transform into frame buffer coordinates
Clip
Convert to pixels in frame buffer

Clipping lines

Have

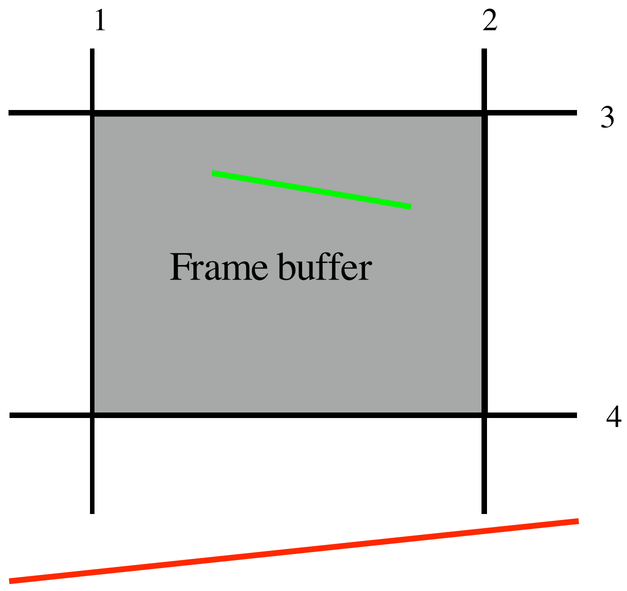


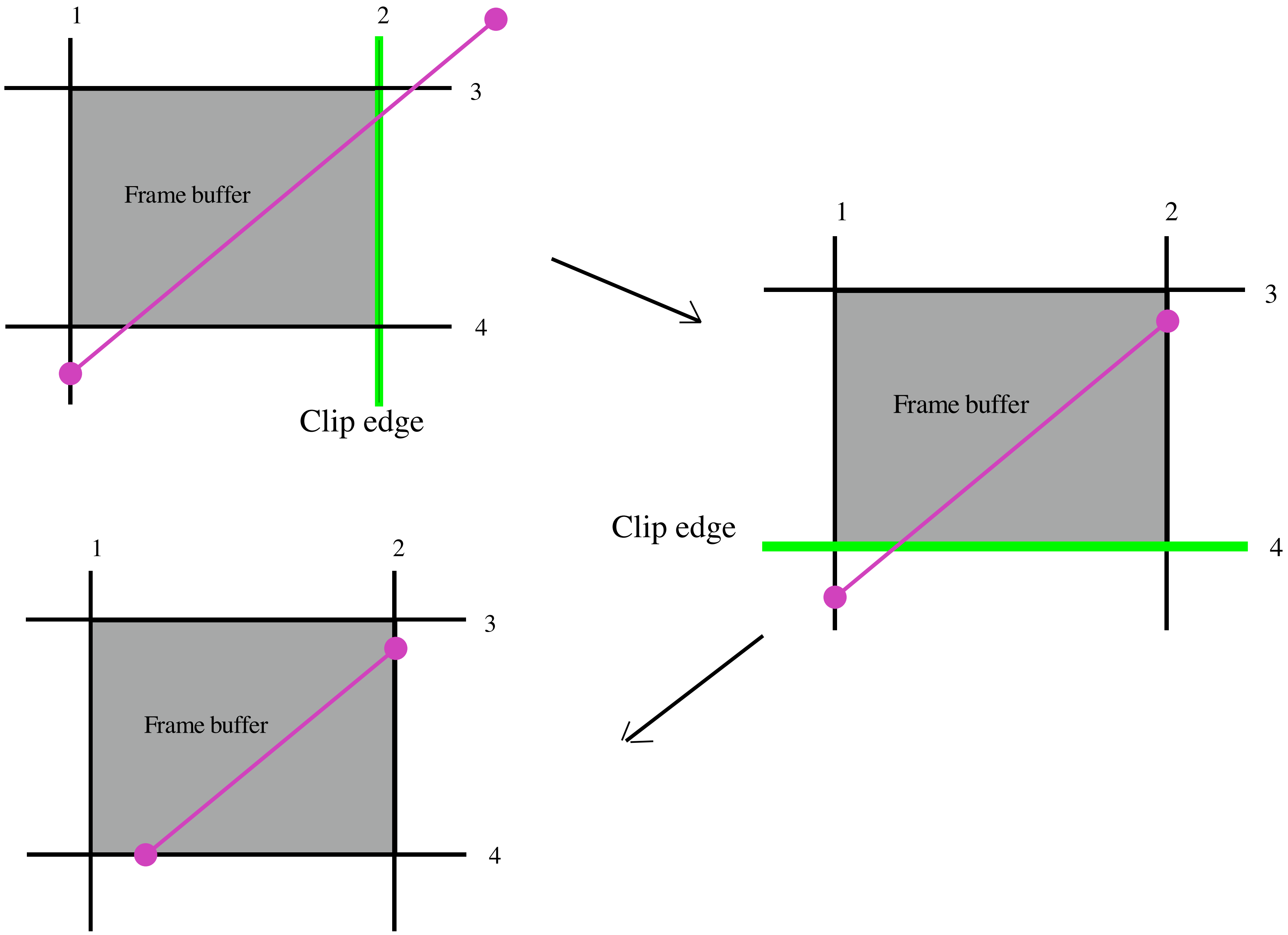
Need



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 some 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.




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 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

Cohen Sutherland - details

- Trivial reject:
  -
- Trivial accept:
  -
- Clipping line against vertical/horizontal edge is easy:
  - line has endpoints (x_s, y_s) and (x_e, y_e)
  -
  -
-

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

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:

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

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

Cyrus-Beck/Liang-Barsky clipping

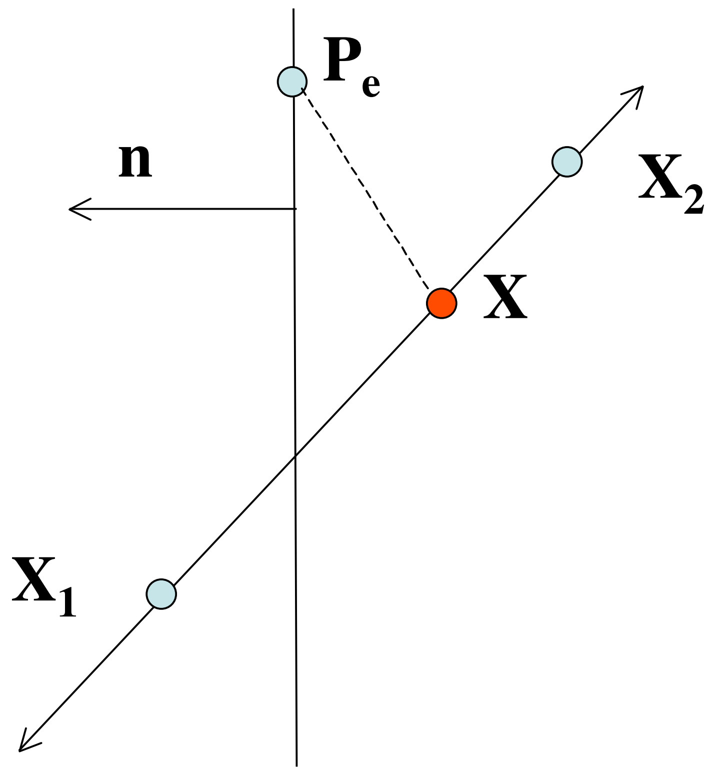
- Consider the parameter values, t, for each clip edge
- Only t inside (0,1) is relevant
- Assumptions
  - $\mathbf{X}_1$!= $\mathbf{X}_2$
  - Ignore case where line is parallel to a clip edge (has no effect, but would lead to divide by zero).
  - We have a normal, **n**, for each clip edge pointing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

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:

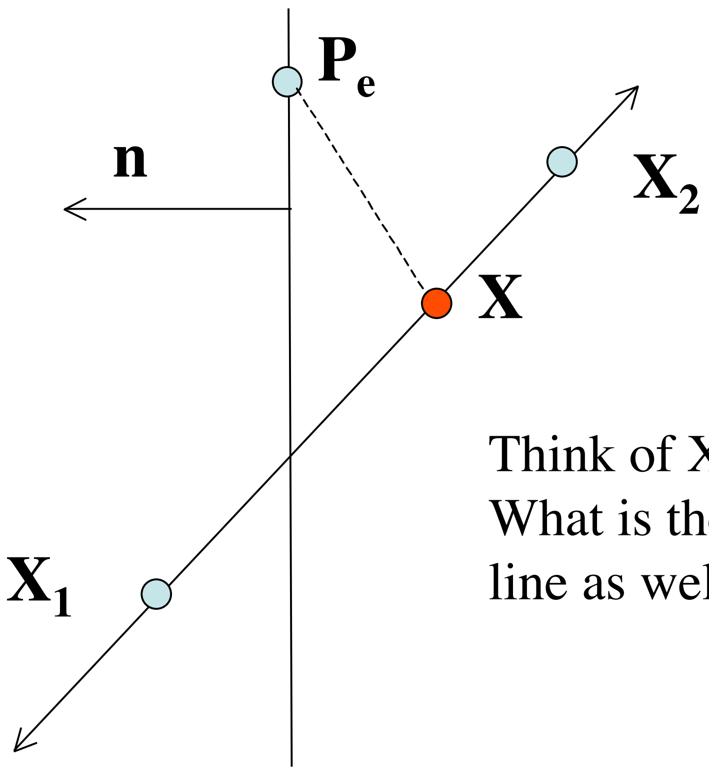
    left (-1,0) right (1,0) top (0,1) bottom (0,-1)

Computing t for intersection point



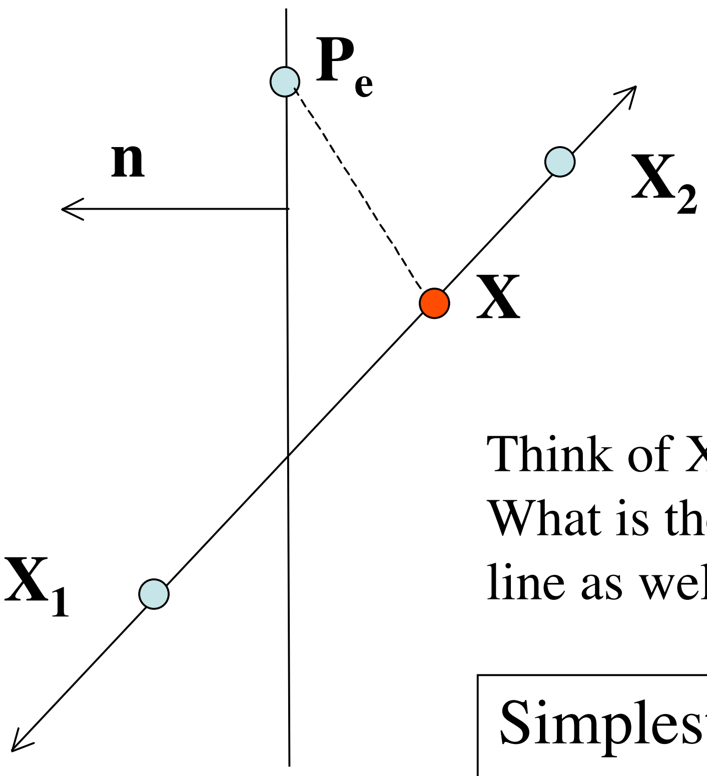
*

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?)

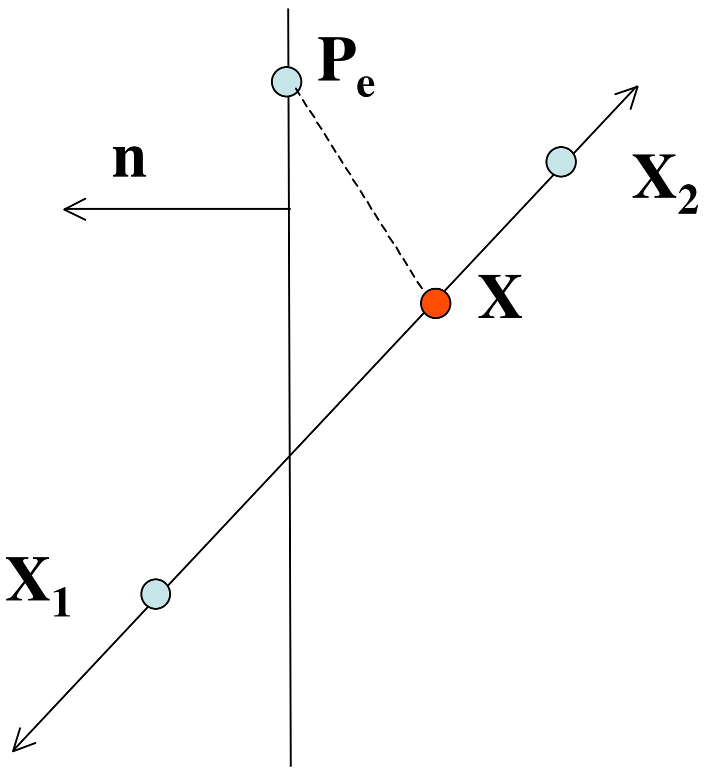
Computing t for intersection point



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

Computing t for intersection point



Set

$$\mathbf{D} = \mathbf{X_2} - \mathbf{X_1}$$

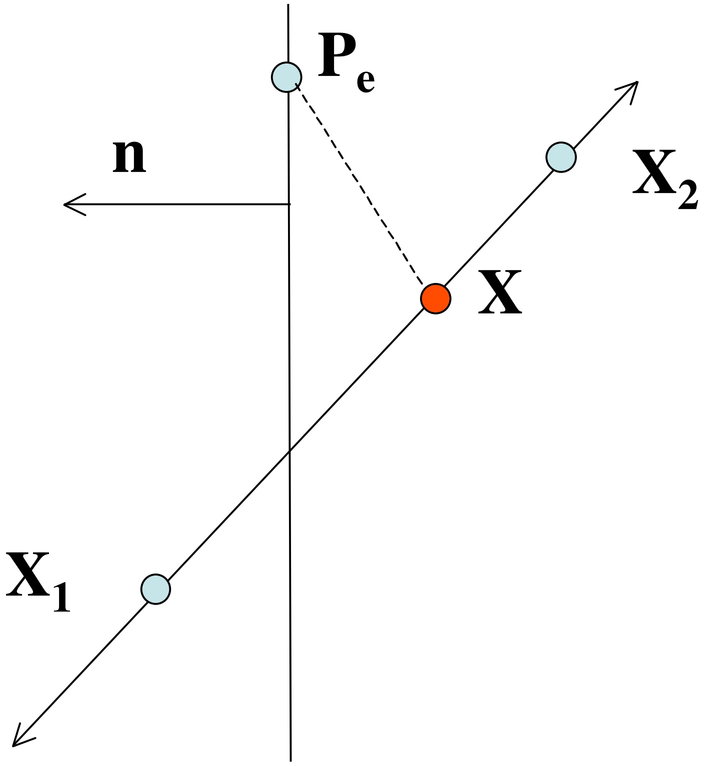
Then

$$\mathbf{X} = \mathbf{X_1} + t\mathbf{D}$$

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 $t = \dfrac{(\mathbf{P_e} - \mathbf{X_1}) \bullet \mathbf{n}}{\mathbf{D} \bullet \mathbf{n}}$

This simplifies greatly for axis aligned rectangles

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

And t = **?**

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**=(-1,0) and $\mathbf{P_e}$=(x_{min},0)

And $t = \frac{(x_1 - x_{min})}{-\Delta x}$

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

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

- Faster derivation for this special case?

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

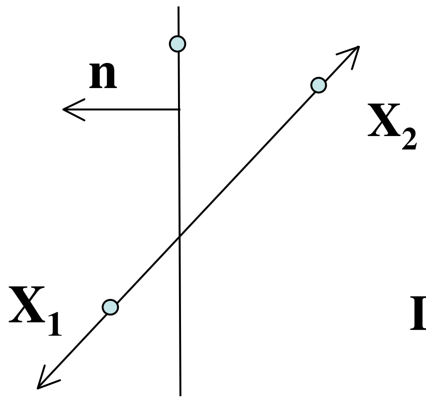
$$
\begin{aligned}
p_1 &= -\Delta x & q_1 &= x_1 - x_{\min} \\
p_2 &= \Delta x & q_2 &= x_{\max} - x_1 \\
p_3 &= -\Delta y & q_3 &= y_1 - y_{\min} \\
p_4 &= \Delta y & q_4 &= y_{\max} - y_1
\end{aligned}
$$

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

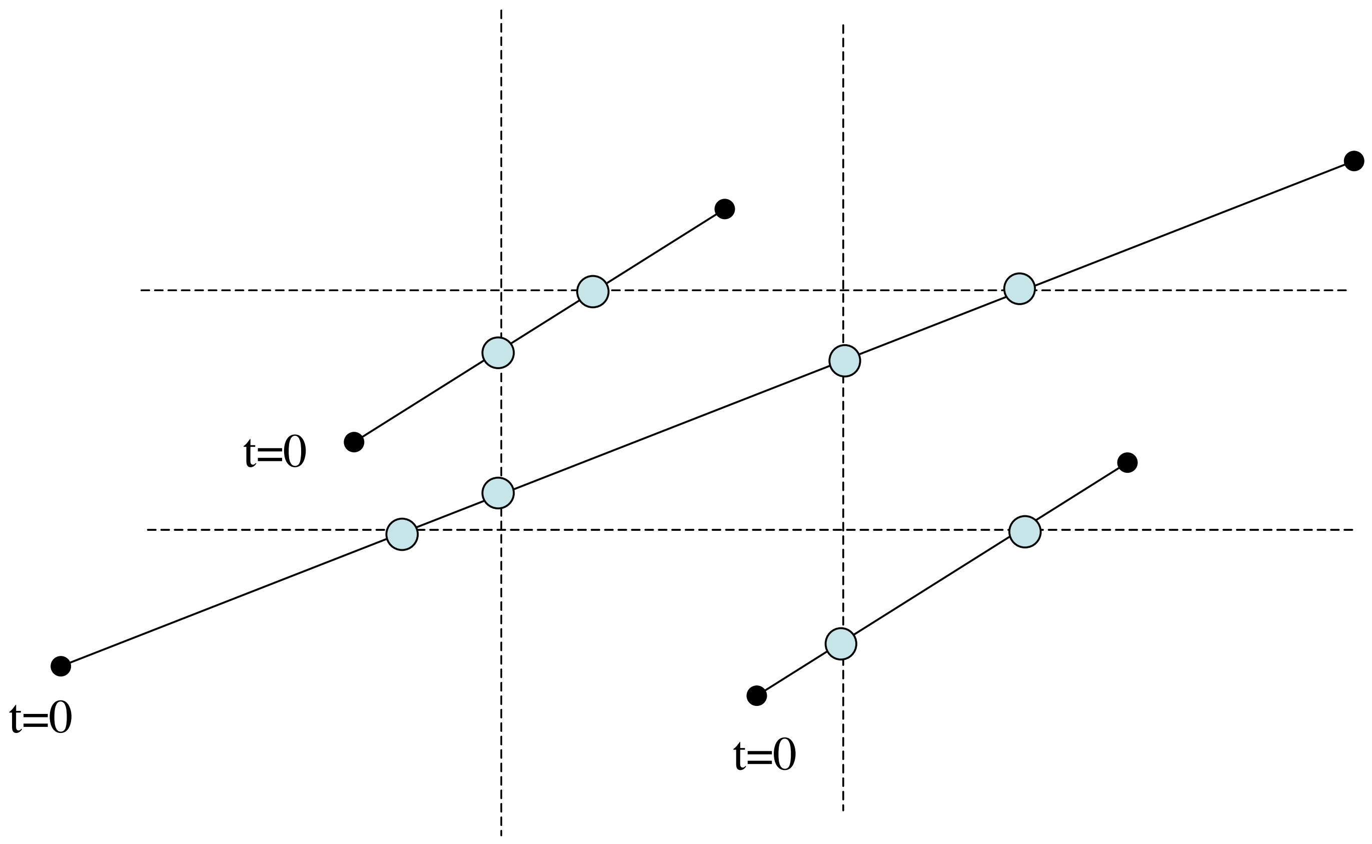
$$
\begin{aligned}
x_{\min} &\le x_1 + t\Delta x \le x_{\max} \\
y_{\min} &\le y_1 + t\Delta y \le y_{\max}
\end{aligned}
$$

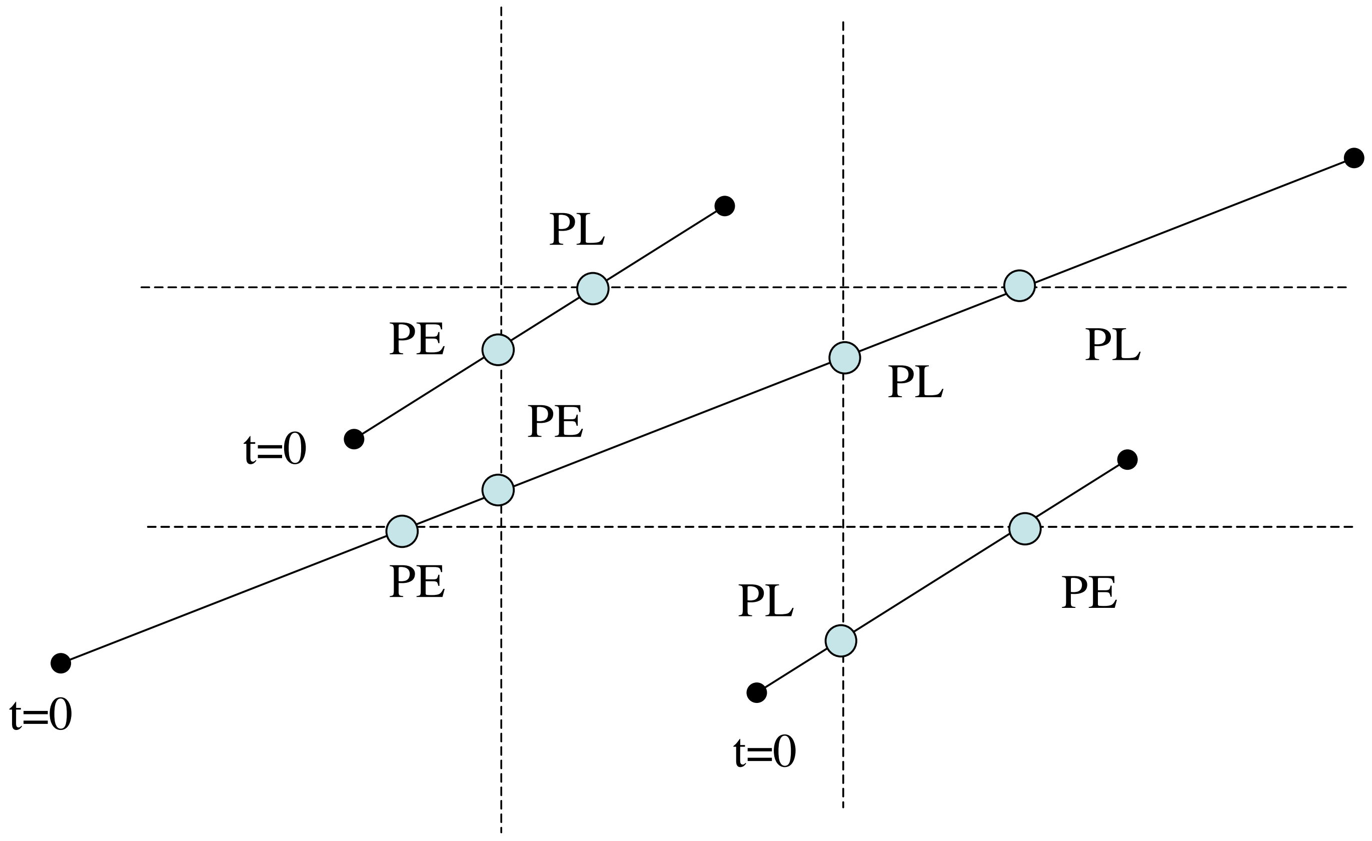
Cyrus-Beck/Liang-Barsky (cont)

- 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.
- Whether an edge is PE or PL is easily determined from the sign of **D•n** which we have already computed.



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


t=0
t=0
t=0


PL
PE
PL
PE
PL
t=0
PE
PL
PE
t=0
t=0

Cyrus-Beck/Liang-Barsky--Algorithm

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 t values (otherwise reject as it is outside).