Clipping

- 2D elements are laid out in some 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.
- Thus we want to compute what part of a 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:
  - If the line is all on wrong side of some edge? throw it away (trivial reject--e.g. red line with respect to bottom edge)
  - If the line is all on right side of all edges? doesn’t need clipping (trivial accept--e.g. green line).
  - If the line crosses edge? Replace endpoint on wrong side with crossing point.
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:
  \[
  \begin{align*}
  x &= x_1 + ux \\
  y &= y_1 + uy
  \end{align*}
  \]
- Clipping conditions on parameter:
  \[
  x_{\text{min}} \leq x_1 + ux \leq x_{\text{max}} \\
  y_{\text{min}} \leq y_1 + uy \leq y_{\text{max}}
  \]
Cyrus-Beck/Liang-Barsky--2

- Conditions become: $u = \frac{q_k}{p_k}$  
- 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.

\[
\begin{align*}
  p_1 &= \boxed{x} & q_1 &= x_1 \boxed{x_{\text{min}}} \\
  p_2 &= \boxed{x} & q_2 &= x_{\text{max}} \boxed{x_1} \\
  p_3 &= \boxed{y} & q_3 &= y_1 \boxed{y_{\text{min}}} \\
  p_4 &= \boxed{y} & q_4 &= y_{\text{max}} \boxed{y_1}
\end{align*}
\]
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).

\[ 0 \leq u \leq 1 \]

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: $u_{\text{small}} = \max(0, \text{incoming values})$
- parameter value for large u end of line is: $u_{\text{large}} = \min(1, \text{outgoing values})$
- if $u_{\text{small}} < u_{\text{large}}$, 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
• 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

Nicholl-Lee-Nicholl clipping
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

Polygon to clip

Clip window
Sutherland-Hodgeman polygon clip

Polygon to clip

Clip entire polygon against one edge

Clip window
Sutherland-Hodgeman polygon clip

Polygon to clip

Clip window

Then clip it against the next
Sutherland-Hodgeman polygon clip

Polygon to clip

Then the next
Sutherland-Hodgeman polygon clip

Polygon to clip

And finally, the last one.
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
    - 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
polygon edge crosses clip edge going from out to in \(\Rightarrow\) emit crossing, next vertex
polygon edge crosses clip edge going from in to out \(\Rightarrow\) emit crossing
polygon edge goes from out to out \(\Rightarrow\) emit nothing
polygon edge goes from in to in \(\Rightarrow\) emit next vertex
Now have

- Polygon edge crosses clip edge going from out to in \implies\text{emit crossing, next vertex}
- Polygon edge crosses clip edge going from in to out \implies\text{emit crossing}
- Polygon edge goes from out to out \implies\text{emit nothing}
- Polygon edge goes from in to in \implies\text{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

class Polygon

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 $\Rightarrow$ emit crossing, next vertex
- Polygon edge crosses clip edge going from in to out $\Rightarrow$ emit crossing
- Polygon edge goes from out to out $\Rightarrow$ emit nothing
- Polygon edge goes from in to in $\Rightarrow$ emit next vertex
Clipping against final(bottom) edge gives

- polygon edge crosses clip edge going from out to in $\Rightarrow$ emit crossing, next vertex
- polygon edge crosses clip edge going from in to out $\Rightarrow$ emit crossing
- polygon edge goes from out to out $\Rightarrow$ emit nothing
- polygon edge goes from in to in $\Rightarrow$ emit next vertex