Point and Line Clipping

*Clipping:* Remove points outside a region of interest.

- Want to discard everything that’s outside of our window…

*Point clipping:* Remove points outside window.

- A point is either entirely inside the region or not.

*Line clipping:* Remove portion of line segment outside window.

- Line segments can straddle the region boundary.
- The Liang-Barsky algorithm efficiently clips line segments against a halfspace.
- Halfspaces can be combined to bound a convex region.
- Use *outcodes* to organize combination of halfspaces.
- Can use some of the ideas in Liang-Barsky to clip points.
*Polygon clipping:* Remove portion of polygon outside window.

- Polygons can straddle the region boundary.
- Concave polygons can be broken into multiple parts.
- Sutherland-Hodgeman algorithm deals with all cases efficiently.
- Built upon efficient line segment clipping.
Parametric representation of line:

\[ P(t) = (1 - t)P_0 + P_1 \]

or equivalently:

\[ P(t) = P_0 + t(P_1 - P_0) \]

- \( P_0 \) and \( P_1 \) are non-coincident points.
- For \( t \in \mathbb{R} \), \( P(t) \) defines an infinite line.
- For \( t \in [0, 1] \), \( P(t) \) defines a line segment from \( P_0 \) to \( P_1 \).
- Good for generating points on a line.
- Not so good for testing if a given point is on a line.
**Implicit representation of line:**

\[ \ell(Q) = (Q - P) \cdot \vec{n} \]

- \( P \) is a point on the line.
- \( \vec{n} \) is a vector perpendicular to the line.
- \( \ell(Q) \) gives us the signed distance from any point \( Q \) to the line.
- The sign of \( \ell(Q) \) tells us if \( Q \) is on the left or right of the line, relative to the direction of \( \vec{n} \).
- If \( \ell(Q) \) is zero, then \( Q \) is on the line.
- Use same form for the implicit representation of a halfspace.
Clipping a point to a halfspace:

- Represent a window edge implicitly...
- Use the implicit form of a line to classify a point \( Q \).
- Must choose a convention for the normal: point to the \textit{inside}.
- Check the sign of \( \ell(Q) \):
  - If \( \ell(Q) > 0 \), the \( Q \) is inside.
  - Otherwise clip (discard) \( Q \); it is on, or outside.
Clipping a line segment to a halfspace: There are three cases:

- The line segment is entirely inside.
- The line segment is entirely outside.
- The line segment is partially inside and partially outside.
Do the easy stuff first: We can devise easy (and fast!) tests for the first two cases:

- \((P_0 - P) \cdot \vec{n} < 0\) AND \((P_1 - P) \cdot \vec{n} < 0\) \(\implies\) Outside
- \((P_0 - P) \cdot \vec{n} > 0\) AND \((P_1 - P) \cdot \vec{n} > 0\) \(\implies\) Inside

We will also need to decide whether “on the boundary” is inside or outside.

Trivial tests are important in computer graphics:

- Particularly if the trivial case is the most common one.
- Particularly if we can reuse the computation for the non-trivial case.
*Do the hard stuff only if we have to:* If line segment partially inside and partially outside, need to clip it:

- Represent the segment from \( P_0 \) to \( P_1 \) in parametric form:

\[
P(t) = (1 - t)P_0 + tP_1 = P_0 + t(P_1 - P_0)
\]

- When \( t = 0 \), \( P(t) = P_0 \). When \( t = 1 \), \( P(t) = P_1 \).
- We now have the following:
• We want $t$ such that $P(t)$ is on $\ell$:

\[
(P(t) - P) \cdot \vec{n} = (P_0 + t(P_1 - P_0) - P) \cdot \vec{n} \\
= (P_0 - P) \cdot \vec{n} t(P_1 - P_0) \cdot \vec{n} \\
= 0
\]

• Solving for $t$ gives us

\[
t = \frac{(P_0 - P) \cdot \vec{n}}{(P_0 - P_1) \cdot \vec{n}}
\]

• NOTE: The values we use for our simple test can be used to compute $t$:

\[
t = \frac{(P_0 - P) \cdot \vec{n}}{(P_0 - P) \cdot \vec{n} - (P_1 - P) \cdot \vec{n}}
\]
Clipping a line segment to a window: Just clip to each of four halfspaces.

Pseudo-code:

given: P, n defining a window edge
for each edge (A,B) = (P0,P1)
    wecA = (A-P) . n
    wecB = (B-P) . n
    if ( wecA < 0 AND wecB < 0 ) then reject
    if ( wecA >= 0 AND wecB >= 0 ) then next
    t = wecA / (wecA - wecB)
    if (wecA < 0 ) then
        A = A + t*(B-A)
    else
        B = B + t*(B-A)
endif
endfor

NOTE:

- Liang-Barsky Algorithm lets us clip lines to arbitrary convex windows.
• Optimizations can be made for the special case of horizontal and vertical window edges.
Q: Can we short-circuit evaluation of a clip on one edge if we know line segment is out on another?

A: Yes. Use outcodes.
   - Do all trivial tests first.
   - Combine results using Boolean operations to determine
     - Trivial accept on window: all edges have trivial accept.
     - Trivial reject on window: any edge has trivial reject.
   - Do Boolean AND, OR operations on bits in a packed integer.
   - Do full clip only if no trivial accept/reject on window.
Line-clip Algorithm generalizes to 3D:

- Half-space now lies on one side of a plane.
- The implicit formula for a plane in 3D is the same as that for a line in 2D.
- The parametric formula for the line to be clipped is unchanged.
3D Clipping

- When do we clip in 3D? We should clip to the near plane before we project. Otherwise, we might map $z$ to 0 and the $x/z$ and $y/z$ are undefined.
- We could clip to all 6 sides of the truncated viewing pyramid. but the plane equations are simpler if we clip after projection, because all sides of volume are parallel to coordinate plane.
- Clipping to a plane in 3D is identical to clipping to a line in 2D.
- We can also clip in homogeneous coordinates.
Polygon Clipping

Polygon Clipping (Sutherland-Hodgeman):

- Window must be a convex polygon.
- Polygon to be clipped can be convex or not.

Approach:

- Polygon to be clipped is given as \( v_1, \ldots, v_n \)
- Each polygon edge is a pair \([v_i, v_{i+1}]\) \(i = 1, \ldots, n\)
  - Don’t forget wraparound; \([v_n, v_1]\) is also an edge
- Process all polygon edges in succession against a window edge
  - Polygon in – polygon out
  - \( v_1, \ldots, v_n \rightarrow w_1, \ldots, w_m \)
- Repeat on resulting polygon with next sequential window edge.
**Contrast with Line Clipping**

- **Line clipping:**
  - Use outcodes to check all window edges before any clip
  - Clip only against possible intersecting window edges
  - Deal with window edges in any order
  - Deal with line segment endpoints in either order

- **Polygon clipping:**
  - Each window edge must be used
  - Polygon edges must be handled in sequence
  - Polygon edge endpoints have a given order
  - Stripped-down line-segment/window-edge clip is a subtask

There are four cases to consider.
Four Cases

- $s = v_i$ is the polygon edge starting vertex
- $p = v_{i+1}$ is the polygon edge ending vertex
- $i$ is a polygon-edge/window-edge intersection point
- $w_j$ is the next polygon vertex to be output

Case 1: Polygon edge is entirely inside the window edge

- $p$ is next vertex of resulting polygon
- $p \rightarrow w_j$ and $j + 1 \rightarrow j$
inside | outside

s

p
(output)

Case 1
Case 2: Polygon edge crosses window edge going out

- Intersection point $i$ is next vertex of resulting polygon
- $i \rightarrow w_j$ and $j + 1 \rightarrow j$
Case 3: Polygon edge is entirely outside the window edge

- No output
Case 4: Polygon edge crosses window edge going in

- Intersection point \( i \) and \( p \) are next two vertices of resulting polygon
- \( i \rightarrow w_j \) and \( p \rightarrow w_{j+1} \) and \( j + 2 \rightarrow j \)
An Example with a Non-convex Polygon