Overview

- Cohen-Sutherland Line Clipping
- Parametric Line Clipping

Motivation

- Only draw lines inside window
- Clip lines to window boundary

Scissoring Clipping

Performed during scan conversion of the line (circle, polygon)

Compute the next point \((x, y)\)

If \(x_{\min} \leq x \leq x_{\max}\) and \(y_{\min} \leq y \leq y_{\max}\)

Then output the point.

Else do nothing

- Issues with scissoring:
  - Too slow
  - Does more work than necessary
  - Better to clip lines to window, than “draw” lines that are outside of window
The Cohen-Sutherland Line Clipping Algorithm

- How to clip lines to fit in windows?
  - easy to tell if whole line falls w/in window
  - harder to tell what part falls inside
- Consider a straight line
- And window: WT, WB, WL and WR

Cohen-Sutherland

Perform trivial accept and reject
- Assign each end point a location code
- Perform bitwise logical operations on a line’s location codes
- Accept or reject based on result
- Frequently provides no information
  - Then perform more complex line intersection

Cohen-Sutherland

The Easy Test:
- Compute 4-bit code based on endpoints
  $P_1$ and $P_2$

Otherwise, we clip the lines:
- We know that there is a bit flip, w.o.l.g.
  assume its $(x_0, x_1)$
- Which bit? Try ’em all!
  - suppose it’s bit 4
  - Then $x_0 < WL$ and we know that $x_1 < WL$
  - We need to find the point: $(x_c, y_c)$
Cohen-Sutherland

- Clearly: \( x_c = WL \)
- Using similar triangles
  \[
  \frac{y_c - y_0}{y_1 - y_0} = \frac{WL - x_0}{x_1 - x_0}
  \]
- Solving for \( y_c \) gives
  \[
  y_c = \frac{WL - x_0}{x_1 - x_0} \left( \frac{y_1 - y_0}{x_1 - x_0} \right) + y_0
  \]
- Replace \((x_0, y_0)\) with \((x_c, y_c)\)
- Re-compute codes
- Continue until all bit
flips (clip lines) are
processed, i.e. all
points are inside the
clip window
Cohen Sutherland

Parametric Line Clipping
- Developed by Cyrus and Beck in 1978
- Used to clip 2D/3D lines against convex polygon/polyhedron
- Liang and Barsky (1984) algorithm efficient in clipping upright 2D/3D clipping regions
- Cyrus-Beck may be reduced to more efficient Liang-Barsky case
- Based on parametric form of a line
  - Line: \( P(t) = P_0 + t(P_1 - P_0) \)

Parametric Line Equation
- Line: \( P(t) = P_0 + t(P_1 - P_0) \)
  - \( t \) value defines a point on the line going through \( P_0 \) and \( P_1 \)
  - \( 0 \leq t \leq 1 \) defines line segment between \( P_0 \) and \( P_1 \)
  - \( P(0) = P_0 \quad P(1) = P_1 \)

The Cyrus-Beck Technique
- Cohen-Sutherland algorithm computes \((x,y)\) intersections of the line and clipping edge
- Cyrus-Beck finds a value of parameter \( t \) for intersections of the line and clipping edges
- Simple comparisons used to find actual intersection points
- Liang-Barsky optimizes it by examining \( t \) values as they are generated to reject some line segments immediately

Calculating \( N_i \)
- \( N \) for window edges
  - WT: (0,1)  WB: (0,-1)  WL: (-1,0)  WR: (1,0)
- \( N_i \) for arbitrary edges
  - Calculate edge direction
    - \( E = (V_1 - V_0) / |V_1 - V_0| \)
    - Be sure to process edges in CCW order
  - Rotate direction vector -90°
    - \( N_i = E_i \)
    - \( N_i = -E_i \)

Finding the Intersection Points
- Line \( P(t) = P_0 + t(P_1 - P_0) \)
- Point on the edge \( P_{ei} \)
- \( N \) → Normal to edge \( i \)
  - \( N_i \) for \( P_1 - P_0 \)
  - Compute direction vector
    - \( E = (V_1 - V_0) / |V_1 - V_0| \)
    - Be sure to process edges in CCW order
  - Rotate direction vector -90°
    - \( N_i = E_i \)
  - Make sure
    - 1. \( D \neq 0 \), or \( P_1 \neq P_2 \)
    - 2. \( N_i \cdot D = 0 \), lines are not parallel
Finding the Line Segment

- Calculate intersection points between line and every window line
- Classify points as potentially entering (PE) or leaving (PL)
- PE if crosses edge into inside half plane \( \Rightarrow \) angle \( P_0 P_1 N_i \) greater than 90° \( \Rightarrow N_i \cdot D < 0 \)
- PL otherwise.
- Find \( T_e = \max(t_e) \)
- Find \( T_l = \min(t_l) \)
- Discard if \( T_e > T_l \)
- If \( T_e < 0 \), \( T_e = 0 \)
- If \( T_l > 1 \), \( T_l = 1 \)
- Use \( T_e, T_l \) to compute intersection coordinates \((x_e, y_e), (x_l, y_l)\)