Line Drawing and Clipping

Outline

- Line drawing
- Digital differential analyzer
- Bresenham’s algorithm
- Cohen-Sutherland line clipping
- Parametric line clipping

Scan-Conversion Algorithms

- Scan-Conversion: Computing pixel coordinates for ideal line on 2D raster grid
- Pixels best visualized as circles/dots
  - Why? Monitor hardware

Digital Differential Analyzer (DDA)

- If slope is less then 1
  - $\Delta x = 1$
  - else $\Delta y = 1$
- Compute corresponding $\Delta y (\Delta x) : m (1/m)$
- $x_{k+1} = x_k + \Delta x$
- $y_{k+1} = y_k + \Delta y$
- Issues:
  - Avoiding floating point ops, multiplications, and real numbers

Bresenham’s Algorithm

- 1965 @ IBM
- Basic Idea:
  - Only integer arithmetic
  - Incremental
- Consider the implicit equation for a line: $f(x,y) = ax + by + c = 0$
Bresenham's Algorithm

• Suppose we just finished \((p_x, p_y)\)
  - (assume 0 ≤ slope ≤ 1)
  - other cases symmetric
• Which pixel next?
  - \(E\) or \(NE\)

  East \((E = (p_x + 1, p_y))\)
  NorthEast \((NE = (p_x + 1, p_y + 1))\)

Bresenham's Algorithm

Assume:

• \(q\) exact value at \(x = p_x + 1\)
• \(y\) midway between \(E\) and \(NE\): \(m = p_y + 1/2\)

Observe:

If \(q < m\), then pick \(E\)
Else pick \(NE\)
If \(q = m\)
  it doesn’t matter

Bresenham's Algorithm

• Create "modified" implicit function (2x)
  \(f(x, y) = 2ax + 2by + 2c = 0\)
• Create a decision variable \(D\) to select,
  where \(D\) is the value of \(f\) at the midpoint:
  \[ D = f(p_x + 1, p_y + (1/2)) \]
  \[ = 2a(p_x + 1) + 2b(p_y + \frac{1}{2}) + 2c \]
  \[ = 2ap_x + 2bp_y + (2a + b + 2c) \]
• If \(D > 0\) then \(m\) is below the line \(f(x, y)\)
  – \(NE\) is the closest pixel
• If \(D ≤ 0\) then \(m\) is above the line \(f(x, y)\)
  – \(E\) is the closest pixel

• Note: because we multiplied by \(2x\), \(D\) is
  now an integer---which is very good news
• How do we make this incremental??

Case I: When \(E\) is next

• What increment for computing a new \(D\)?
  \[ D_{new} = f(p_x + 2, p_y + (1/2)) \]
  \[ = 2a(p_x + 2) + 2b(p_y + \frac{1}{2}) + 2c \]
  \[ = 2ap_x + 2bp_y + (4a + b + 2c) \]
  \[ = 2ap_x + 2bp_y + (2a + b + 2c) + 2a \]
  \[ = D + 2a = D + 2d_x \]
• Hence, increment by: \(2d_x\)

Case II: When \(NE\) is next

• What increment for computing a new \(D\)?
  \[ D_{new} = f(p_x + 2, p_y + 1 + (1/2)) \]
  \[ = 2a(p_x + 2) + 2b(p_y + 1/2) \]
  \[ = 2ap_x + 2bp_y + (4a + 3b + 2c) \]
  \[ = 2ap_x + 2bp_y + (2a + b + 2c) + (2a + 2b) \]
  \[ = D + 2(a + b) = D + 2(d_x + d_y) \]
• Hence, increment by: \(2(d_y - d_x)\)
How to get an initial value for \( D \)?

- Suppose we start at: \((q_x, q_y)\)
- Initial midpoint is: \((q_x + 1, q_y + 1/2)\)

Then:

\[
D_{init} = f(q_x + 1, q_y + 1/2) \\
= 2a(q_x + 1) + 2b \left( q_y + \frac{1}{2} \right) + 2c \\
= (2aq_x + 2bq_y + 2c) + (2a + b) \\
= 0 + 2a + b \\
= 2d_y - d_x
\]

The Algorithm

```c
void bresenham(int x0, y0, x1, y1) { 
  int dx = x1 - x0; 
  int dy = y1 - y0; 
  int D = 2 * dy - dx; 
  int x = x0, y = y0; 
  if (dx < dy) { // below midpoint - go to x 
    for (x = x0; x <= x1; x++) { 
      y++; 
      if (D < 0) D += 2 * dy; 
      else { // above midpoint - go to y 
        D += 2 * (dy - dx); 
        y--; 
      } 
    } 
  } else { // above midpoint - go to y 
    for (y = y0; y <= y1; y++) { 
      x++; 
      if (D < 0) D += 2 * dy; 
      else { // below midpoint - go to x 
        D += 2 * (dy - dx); 
        x--; 
      } 
    } 
  }
}
```

Assumptions: \( q_x < r_x \)

Pre-computed: \( 2d_y, 2(d_y - d_x) \)
Bresenham's Algorithm: Example

Some issues with Bresenham's Algorithms
- Pixel 'density' varies based on slope
  - straight lines look darker, more pixels per CM
- Endpoint order
  - Line from P1 to P2 should match P2 to P1
  - Always choose E when hitting M, regardless of direction
Some issues with Bresenham’s Algorithms

- How to handle the line when it hits the clip window?
- Vertical intersections
  - Could change line slope
  - Need to change init cond.
- Horizontal intersections
  - Again, changes in the boundary conditions
  - Can’t just intersect the line w/ the box

Scissoring Clipping

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

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

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

Then output the point.
Else do nothing

- Issues with scissoring:
  - Too slow
  - Does more work than necessary

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 fails inside
- Consider a straight line \(P_0 = (x_0, y_0)\) and \(P_1 = (x_1, y_1)\)
- And window: \(WT, WB, WL\) and \(WR\)

Cohen-Sutherland

Basic Idea:
- First, do easy test
  - completely inside or outside the box?
- If no, we need a more complex test
- Note: we will also need to figure out how scan line meets the box

Cohen-Sutherland

- The Easy Test:
- Compute 4-bit code based on endpoints \(P_1\) and \(P_2\)

Cohen-Sutherland

- Line is completely visible if both code values of endpoints are 0, i.e., \(C_0 \lor C_1 = 0\)
- If line segments are completely outside the window, then \(C_0 \land C_1 \neq 0\)
Otherwise, we clip the lines:

- We know that there is a bit flip, w.o.i.g. assume its $\langle x_0, x_1 \rangle$
- Which bit? Try ‘em all!
  - suppose its bit 4
  - Then $x_0 < WL$ and we know that $x_1 \geq WL$.
  - We need to find the point: $(x_c, y_c)$

\[
\begin{align*}
\frac{WL-x_0}{x_1-x_0} &= \frac{y_c-y_0}{y_1-y_0} \\
y_c &= \frac{WL-x_0}{x_1-x_0} (y_1-y_0) + y_0
\end{align*}
\]

- 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
  \[
  y_c = \frac{WL-x_0}{x_1-x_0} (y_1-y_0) + y_0
  \]

- Replace $(x_0, y_0)$ with $(x_c, y_c)$
- Re-compute values
- Continue until all points are inside the clip window

QED

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

The Cyrus-Beck Technique

- Cohen-Sutherland algorithm computes $\langle x, y \rangle$ 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
Finding the Intersection Points

Line \( P(t) = P_0 + t(P_1 - P_0) \)

Point on the edge \( P_{E_i} \)

1. \( D \neq 0 \), or \( P_1 \neq P_0 \)
2. \( N_i \cdot D \neq 0 \) lines are not parallel

Let \( D = (P_1 - P_0) \)

\[
1 = \frac{N_i \cdot [P_i - P_0]}{N_i \cdot D}
\]

Make sure

1. \( D = 0 \), or \( P_1 = P_0 \)
2. \( N_i \cdot D = 0 \) lines are not parallel

Finding the Line Segment

Classify point as potentially entering (PE) or leaving (PL)

1. PE if crosses edge into inside half plane => angle \( P_0 P_1 \)
2. PL otherwise.

- Find \( T_a = \max(t_a) \)
- Find \( T_i = \min(t_i) \)
- Discard if \( T_a > T_i \)
- If \( T_a < 0 \), \( T_a = 0 \)
- If \( T_i > 1 \), \( T_i = 1 \)
- Use \( T_a, T_i \) to compute intersection coordinates \( (x_e, y_e), (x_l, y_l) \)

Arrangements for Presentations

- Pick dates
- Priority - reverse alphabetical order

- Go to assignment 1