CS 430/536
Computer Graphics I

Line Clipping
2D Transformations
Week 2, Lecture 3

David Breen, William Regli and Maxim Peysakhov
Geometric and Intelligent Computing Laboratory
Department of Computer Science
Drexel University
http://gicl.cs.drexel.edu
Overview

• Cohen-Sutherland Line Clipping
• Parametric Line Clipping
• 2D Affine transformations
• Homogeneous coordinates
• Discussion of homework #1

Lecture Credits: Most pictures are from Foley/VanDam; Additional and extensive thanks also goes to those credited on individual slides
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 $P_0 = (x_0, y_0)$ 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 line intersects the box
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 $P_2$

<table>
<thead>
<tr>
<th>1001</th>
<th>1000</th>
<th>1010</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Window</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>0000</td>
<td>0010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>0100</td>
<td>0110</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bit 1:** 1 if point is above window, i.e. $y > WT$.
**Bit 2:** 1 if point is below window, i.e. $y < WB$.
**Bit 3:** 1 if point is right of window, i.e. $x > WR$.
**Bit 4:** 1 if point is left of window, i.e. $x < WL$.

Pics/Math courtesy of Dave Mount @ UMD-CP
Cohen-Sutherland

- Line is completely visible iff both code values of endpoints are 0, i.e. \( C_0 \lor C_1 = 0 \)

- If line segments are completely outside the window, then \( C_0 \land C_1 \neq 0 \)

\[
\begin{array}{ccc}
1001 & 1000 & 1010 \\
0001 & 0000 & 0010 \\
0101 & 0100 & 0110 \\
WL & WR & WT \\
\end{array}
\]

Pics/Math courtesy of Dave Mount @ UMD-CP
Cohen-Sutherland

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

- 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
Cohen Sutherland

Window

WT
WB
WL
WR
Cohen Sutherland
Cohen Sutherland
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 <= t <= 1 defines line segment between** \( P_0 \) **and** \( P_1 \)
- **\( P(0) = P_0 \) \( P(1) = P_1 \)**
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
Finding the Intersection Points

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

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

\( N_i \rightarrow \) Normal to edge i

\[ N_i \cdot [P(t) - P_{Ei}] = 0 \]

\[ N_i \cdot [P_0 + t(P_1 - P_0) - P_{Ei}] = 0 \]

\[ N_i \cdot [P_0 - P_{Ei}] + N_i \cdot t[P_1 - P_0] = 0 \]

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

\[ t = \frac{N_i \cdot [P_0 - P_{Ei}]}{-N_i \cdot D} \]

Make sure

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

$N_i$ 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 $-90^\circ$
  $N_x = E_y$
  $N_y = -E_x$
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 => angle $P_0 P_1$ and $N_i$ greater $90^\circ$ => $N_i \cdot D < 0$
- 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 $(x_e, y_e)$, $(x_l, y_l)$
2D Transformations
2D Affine Transformations

All represented as matrix operations on vectors!
Parallel lines preserved, angles/lengths not

- Scale
- Rotate
- Translate
- Reflect
- Shear

Pics/Math courtesy of Dave Mount @ UMD-CP
2D Affine Transformations

- **Example 1**: rotation and non uniform scale on unit cube

- **Example 2**: shear first in x, then in y

**Note:**
- Preserves parallels
- Does not preserve lengths and angles
2D Transforms: Translation

- Rigid motion of points to new locations
  \[ x' = x + d_x \]
  \[ y' = y + d_y \]

- Defined with column vectors:
  \[
  \begin{bmatrix}
  x' \\
  y'
  \end{bmatrix}
  =
  \begin{bmatrix}
  x \\
  y
  \end{bmatrix}
  +
  \begin{bmatrix}
  d_x \\
  d_y
  \end{bmatrix}
  \]
  as \( P' = P + T \)

1994 Foley/VanDam/Finer/Hugues/Phillips ICG
2D Transforms: Scale

- Stretching of points along axes:
  
  \[ x' = s_x \cdot x \]
  
  \[ y' = s_y \cdot y \]

In matrix form:

\[
\begin{bmatrix}
  x' \\
  y'
\end{bmatrix} =
\begin{bmatrix}
  s_x & 0 \\
  0 & s_y
\end{bmatrix} \cdot
\begin{bmatrix}
  x \\
  y
\end{bmatrix}
\]

or just: \( P' = S \cdot P \)
2D Transforms: Rotation

- Rotation of points about the origin
  
  \[ x' = x \cdot \cos \theta - y \cdot \sin \theta \]
  
  \[ y' = x \cdot \sin \theta + y \cdot \cos \theta \]

  Positive Angle: CCW
  Negative Angle: CW

  \[
  \begin{bmatrix}
  x' \\
  y'
  \end{bmatrix} =
  \begin{bmatrix}
  \cos \theta & -\sin \theta \\
  \sin \theta & \cos \theta
  \end{bmatrix}
  \cdot
  \begin{bmatrix}
  x \\
  y
  \end{bmatrix}
  \]

  or just: \( P' = R \cdot P \)
2D Transforms: Rotation

- Substitute the 1\textsuperscript{st} two equations into the 2\textsuperscript{nd} two to get the general equation

\[ x = r \cdot \cos \phi \]
\[ y = r \cdot \sin \phi \]
\[ x' = r \cdot \cos (\theta + \phi) = r \cdot \cos \phi \cdot \cos \theta - r \cdot \sin \phi \cdot \sin \theta \]
\[ y' = r \cdot \sin (\theta + \phi) = r \cdot \cos \phi \cdot \sin \theta + r \cdot \sin \phi \cdot \cos \theta \]
\[ x' = x \cos(\theta) - y \sin(\theta) \]
\[ y' = x \sin(\theta) + y \cos(\theta) \]
Homogeneous Coordinates

- Observe: *translation* is treated differently from *scaling* and *rotation*

- **Homogeneous coordinates**: allows all transformations to be treated as matrix multiplications

\[
P' = P + T \\
P' = S \cdot P \\
P' = R \cdot P
\]

Example: A 2D point \((x,y)\) is the line \((x,y,w)\), where \(w\) is any real #, in 3D homogenous coordinates.

To get the point, *homogenize* by dividing by \(w\) (i.e. \(w=1\))
Recall our Affine Transformations

Rotation  Translation  Uniform Scaling  Nonuniform Scaling  Reflection  Shearing
Matrix Representation of 2D Affine Transformations

- Translation:
  \[
  \begin{bmatrix}
  x' \\ y' \\ 1
  \end{bmatrix} =
  \begin{bmatrix}
  1 & 0 & d_x \\
  0 & 1 & d_y \\
  0 & 0 & 1
  \end{bmatrix} \cdot
  \begin{bmatrix}
  x \\ y \\ 1
  \end{bmatrix}
  \]

- Scale:
  \[
  \begin{bmatrix}
  x' \\ y' \\ 1
  \end{bmatrix} =
  \begin{bmatrix}
  s_x & 0 & 0 \\
  0 & s_y & 0 \\
  0 & 0 & 1
  \end{bmatrix} \cdot
  \begin{bmatrix}
  x \\ y \\ 1
  \end{bmatrix}
  \]

- Rotation:
  \[
  \begin{bmatrix}
  x' \\ y' \\ 1
  \end{bmatrix} =
  \begin{bmatrix}
  \cos \theta & -\sin \theta & 0 \\
  \sin \theta & \cos \theta & 0 \\
  0 & 0 & 1
  \end{bmatrix} \cdot
  \begin{bmatrix}
  x \\ y \\ 1
  \end{bmatrix}
  \]

- Shear:
  \[
  SH_x =
  \begin{bmatrix}
  1 & a & 0 \\
  0 & 1 & 0 \\
  0 & 0 & 1
  \end{bmatrix}
  \]

- Reflection:
  \[
  F_y =
  \begin{bmatrix}
  1 & 0 & 0 \\
  0 & -1 & 0 \\
  0 & 0 & 1
  \end{bmatrix}
  \]
Composition of 2D Transforms

- Rotate about a point \( P1 \)
  - Translate \( P1 \) to origin
  - Rotate
  - Translate back to \( P1 \)

\[
\begin{align*}
\mathbf{P}' &= \mathbf{T} \ast \mathbf{P} \\
\mathbf{T} &= \begin{bmatrix}
\cos \theta & -\sin \theta & 0 \\
\sin \theta & \cos \theta & 0 \\
0 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
1 & 0 & -x_1 \\
0 & 1 & -y_1 \\
0 & 0 & 1
\end{bmatrix}
\end{align*}
\]
Composition of 2D Transforms

- Scale object around point \( P1 \)
  - \( P1 \) to origin
  - Scale
  - Translate back to \( P1 \)
  - Compose into \( \mathcal{T} \)

\[
T(x_1, y_1) \cdot S(S_x, S_y) \cdot T(-x_1, -y_1)
\]

\[
\begin{bmatrix}
1 & 0 & x_1 \\
0 & 1 & y_1 \\
0 & 0 & 1
\end{bmatrix} \cdot \begin{bmatrix}
S_x & 0 & 0 \\
0 & S_y & 0 \\
0 & 0 & 1
\end{bmatrix} \cdot \begin{bmatrix}
1 & 0 & -x_1 \\
0 & 1 & -y_1 \\
0 & 0 & 1
\end{bmatrix}
\]

\[
= \begin{bmatrix}
S_x & 0 & x_1(1 - S_x) \\
0 & S_y & y_1(1 - S_y) \\
0 & 0 & 1
\end{bmatrix}
\]

\[
P' = \mathcal{T} \cdot P
\]
Composition of 2D Transforms

- Scale + rotate object around point $P1$ and move to $P2$
  - $P1$ to origin
  - Scale
  - Rotate
  - Translate to $P2$

$$P' = T * P$$

$T(x_2, y_2) \cdot R(\theta) \cdot S(s_x, s_y) \cdot T(-x_1, -y_1)$

Original house  →  Translate $P_1$ to origin  →  Scale  →  Rotate  →  Translate to final position $P_2$
Composition of 2D Transforms

• Be sure to multiple transformations in proper order!

\[ P' = (T \star (R \star (S \star (T \star P)))) \]

\[ P' = ((T \star (R \star (S \star T))) \star P) \]

\[ P' = \mathcal{T} \star P \]
Programming assignment 1

• Implement Simplified Postscript reader
• Implement 2D transformations
• Implement Cohen-Sutherland clipping
  – Generalize edge intersection formula
• Generalize DDA or Bresenham algorithm
• Implement XPM image writer