CS 430
Computer Graphics I

Line Clipping
2D Transformations

Week 2, Lecture 3

David Breen, William Regli and Maxim Peysakhov
Department of Computer Science
Drexel University

UNIVERSITY

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.

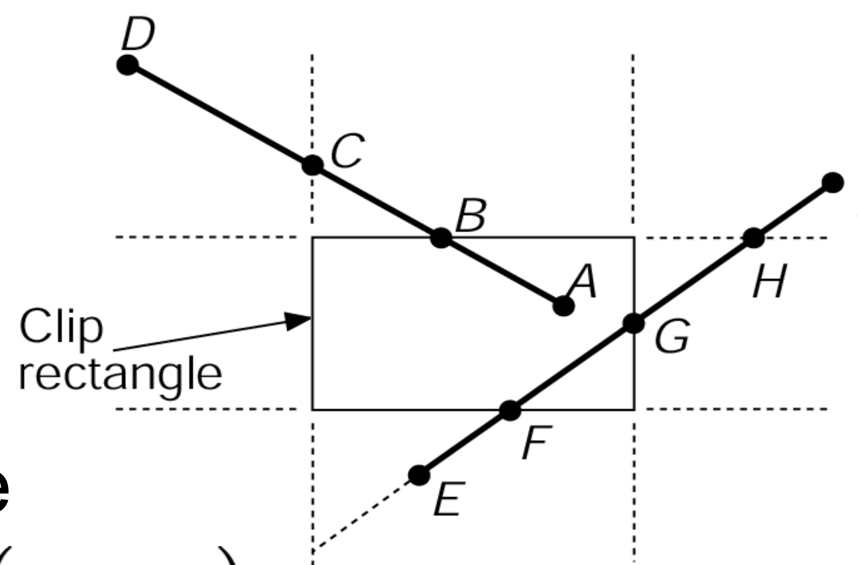
Else do nothing

- Issues with scissoring:
  - Too slow
  - Does more work then necessary
- Better to clip lines to window, than “draw” lines that are outside of window

Pics/Math courtesy of Dave Mount @ UMD-CP

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

$$P_0 = (x_0, y_0) \text{ and } P_1 = (x_1, y_1)$$

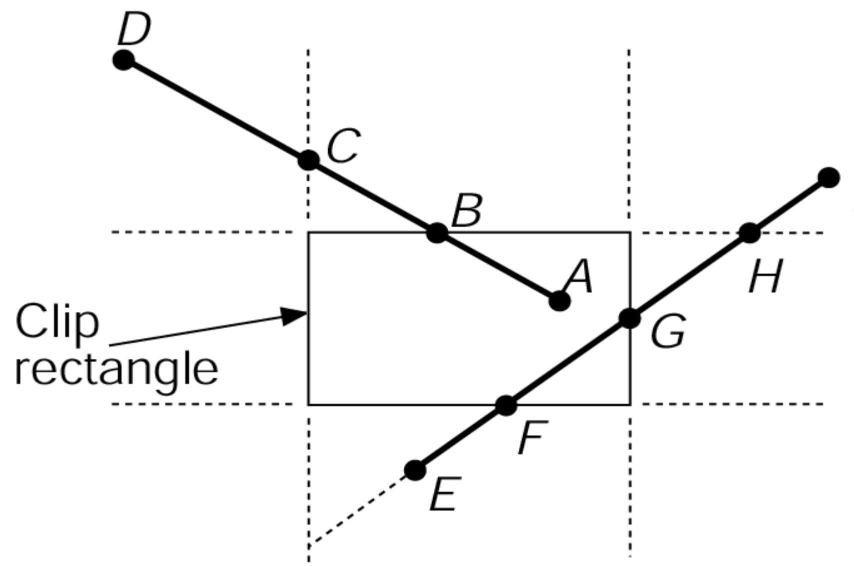
- And window: $WT,\ WB,\ WL$ and WR

Pics/Math courtesy of Dave Mount @ UMD-CP

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



Pics/Math courtesy of Dave Mount @ UMD-CP

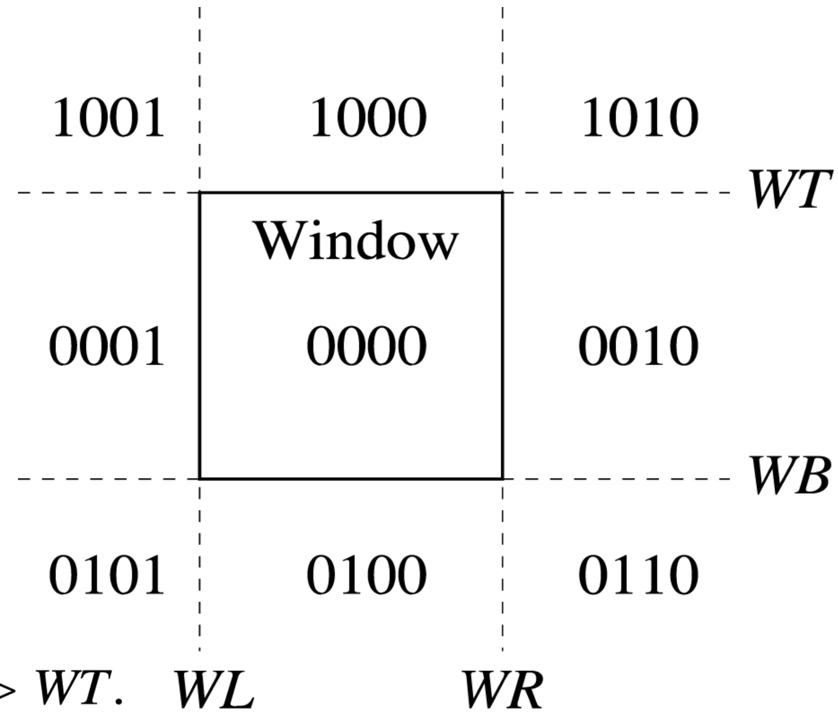
Cohen-Sutherland

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



Bit 1: 1 if point is above window, i.e. $y > WT$.

Bit 2: 1 if point is below window, i.e. $y < WB$.

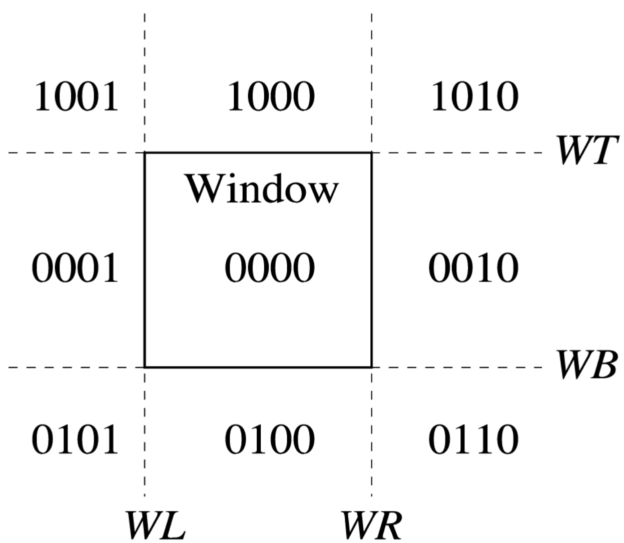
Bit 3: 1 if point is right of window, i.e. $x > WR$.

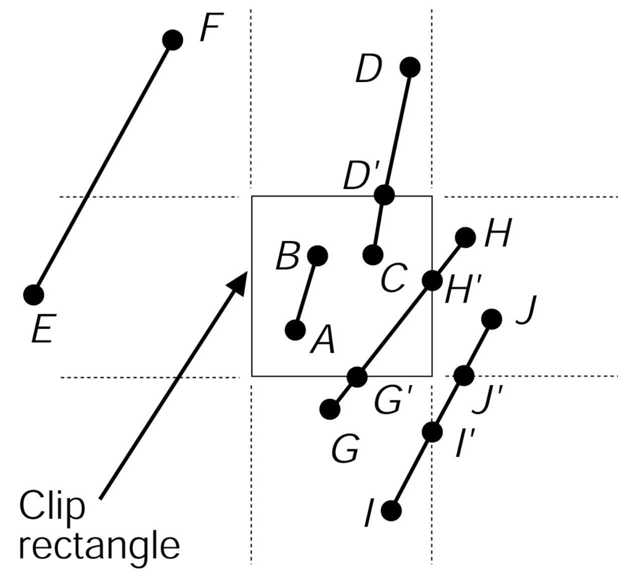
Bit 4: 1 if point is left of window, i.e. $x < WL$.

Pics/Math courtesy of Dave Mount @ UMD-CP

Cohen-Sutherland

- Line is completely visible iff both code values of endpoints are *0,* i.e. $C_0 \vee C_1 = 0$
- If line segments are completely outside the window, then $C_0 \wedge C_1 \neq 0$



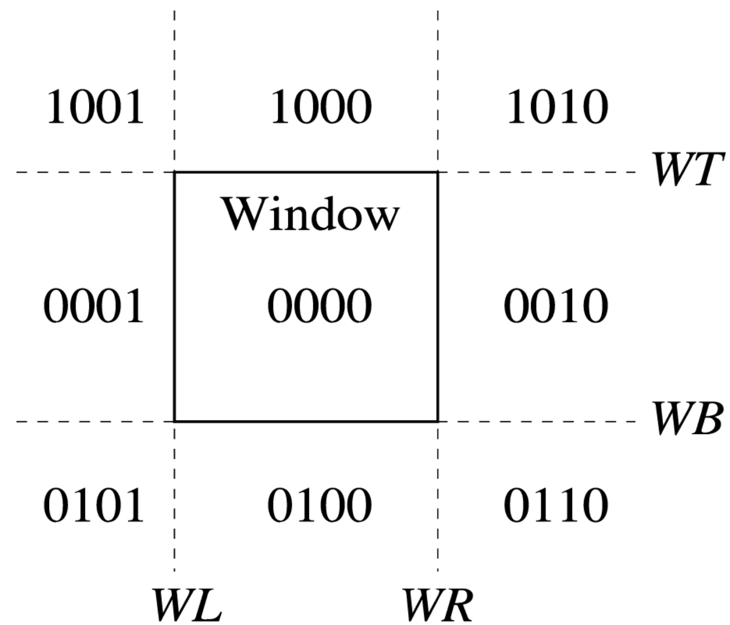


Pics/Math courtesy of Dave Mount @ UMD-CP

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:* $(\mathbf{x_c}, \mathbf{y_c})$



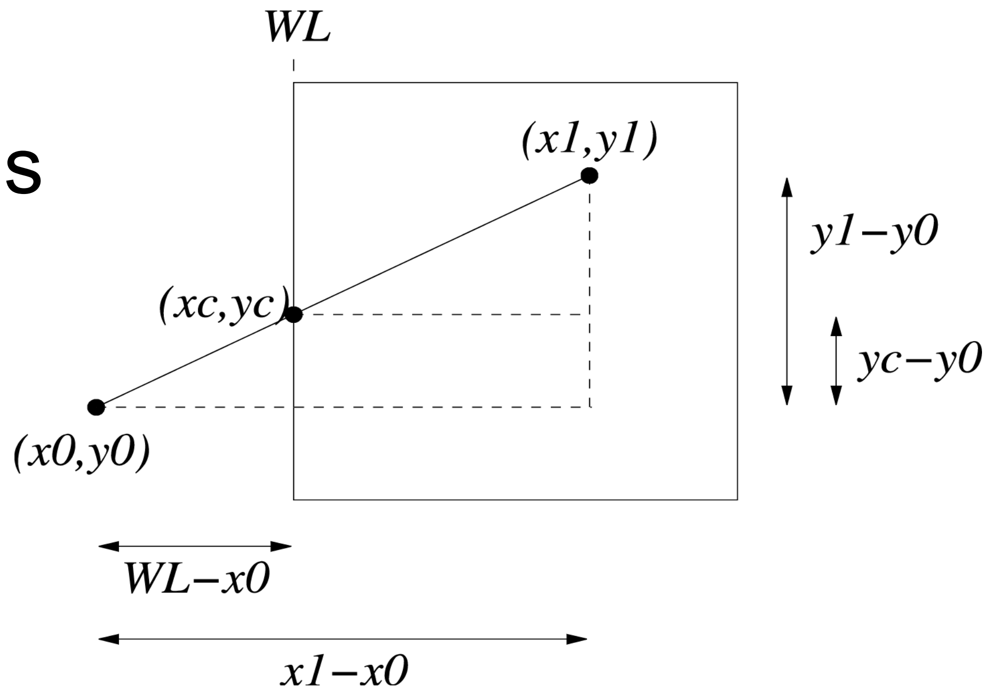
Pics/Math courtesy of Dave Mount @ UMD-CP

Cohen-Sutherland

- Clearly: $x_c = WL$
- Using *similar* triangles

$$\frac{y_c - y_0}{y_1 - y_0} = \frac{WL - x_0}{x_1 - x_0}$$

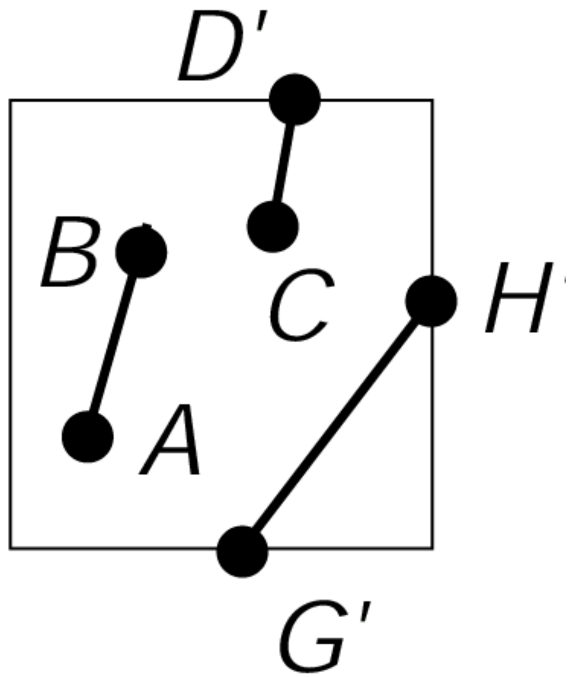
- Solving for y_c gives



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

Pics/Math courtesy of Dave Mount @ UMD-CP

Cohen-Sutherland

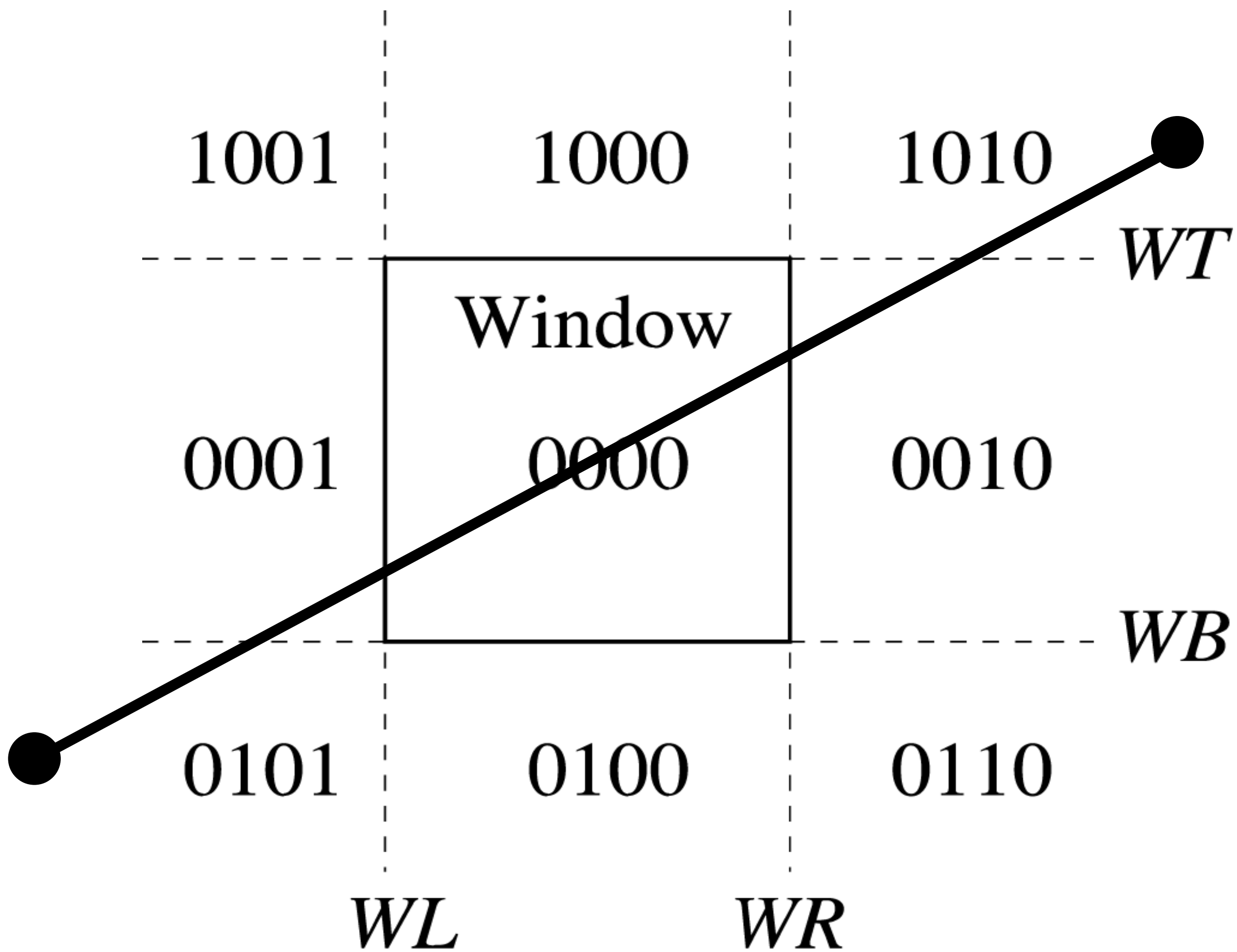


- Replace
  (x_0, y_0) with (x_c, y_c)
- Re-compute codes
- Continue until all bit flips (clip lines) are processed, i.e. all points are inside the clip window

Pics/Math courtesy of Dave Mount @ UMD-CP

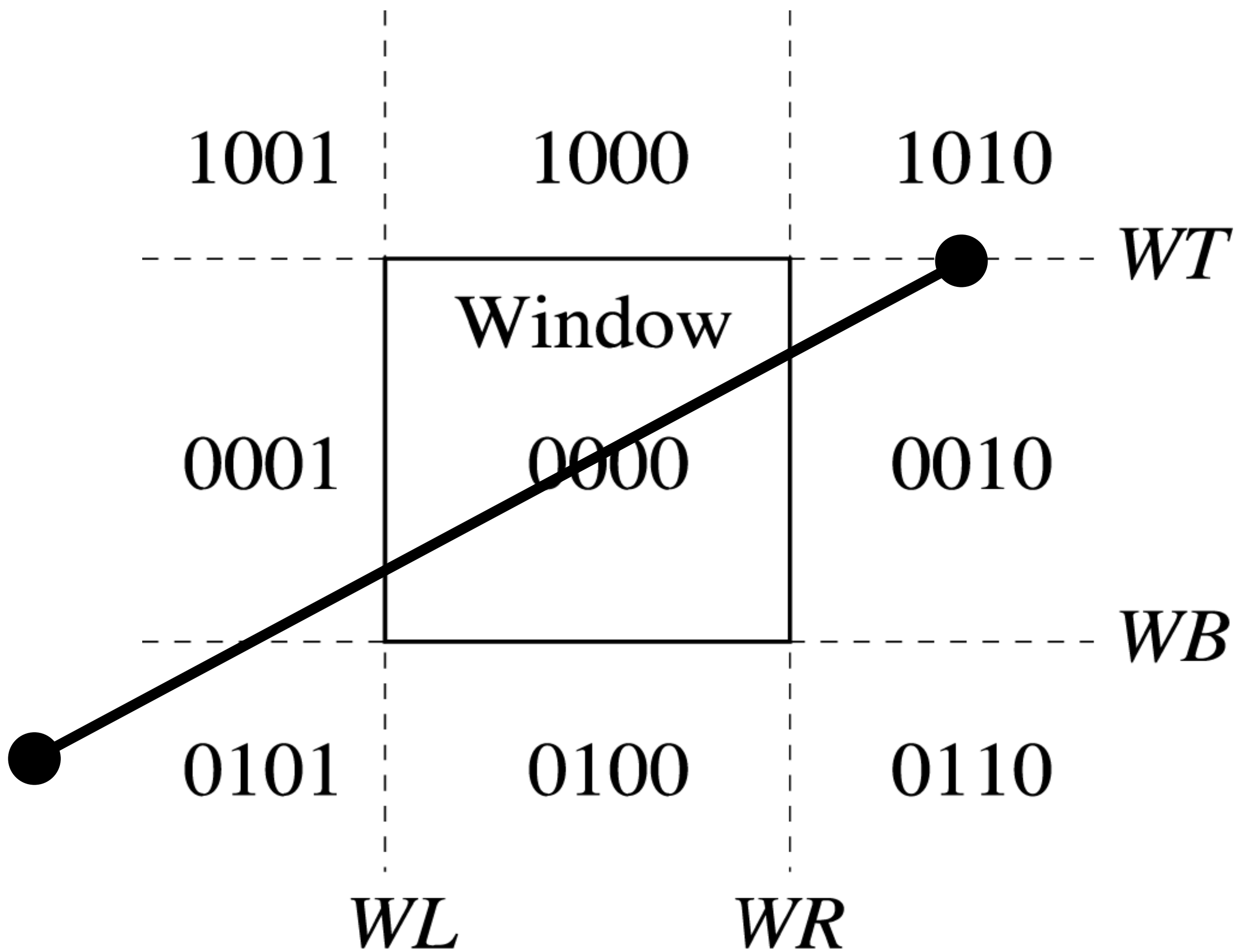
Cohen Sutherland

0101
1010



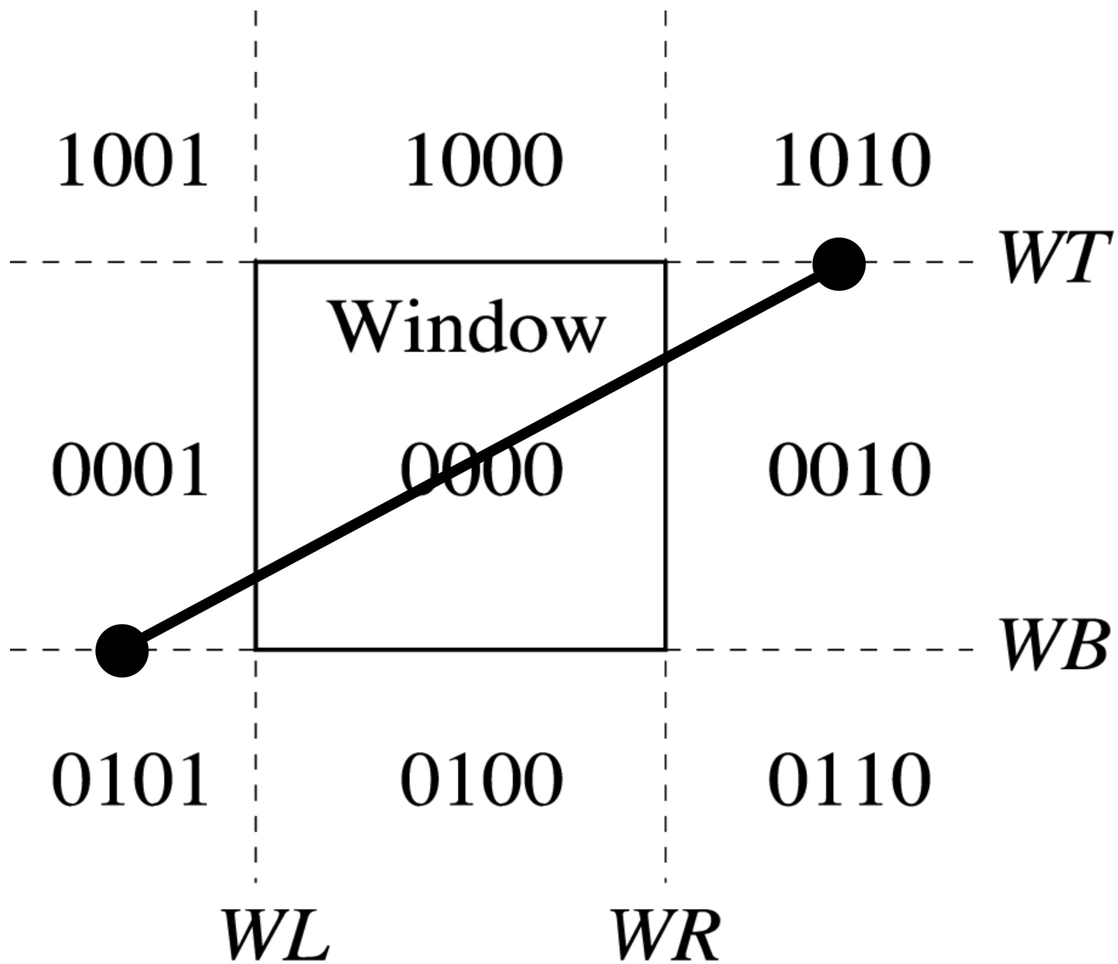
Cohen Sutherland

0101
0010



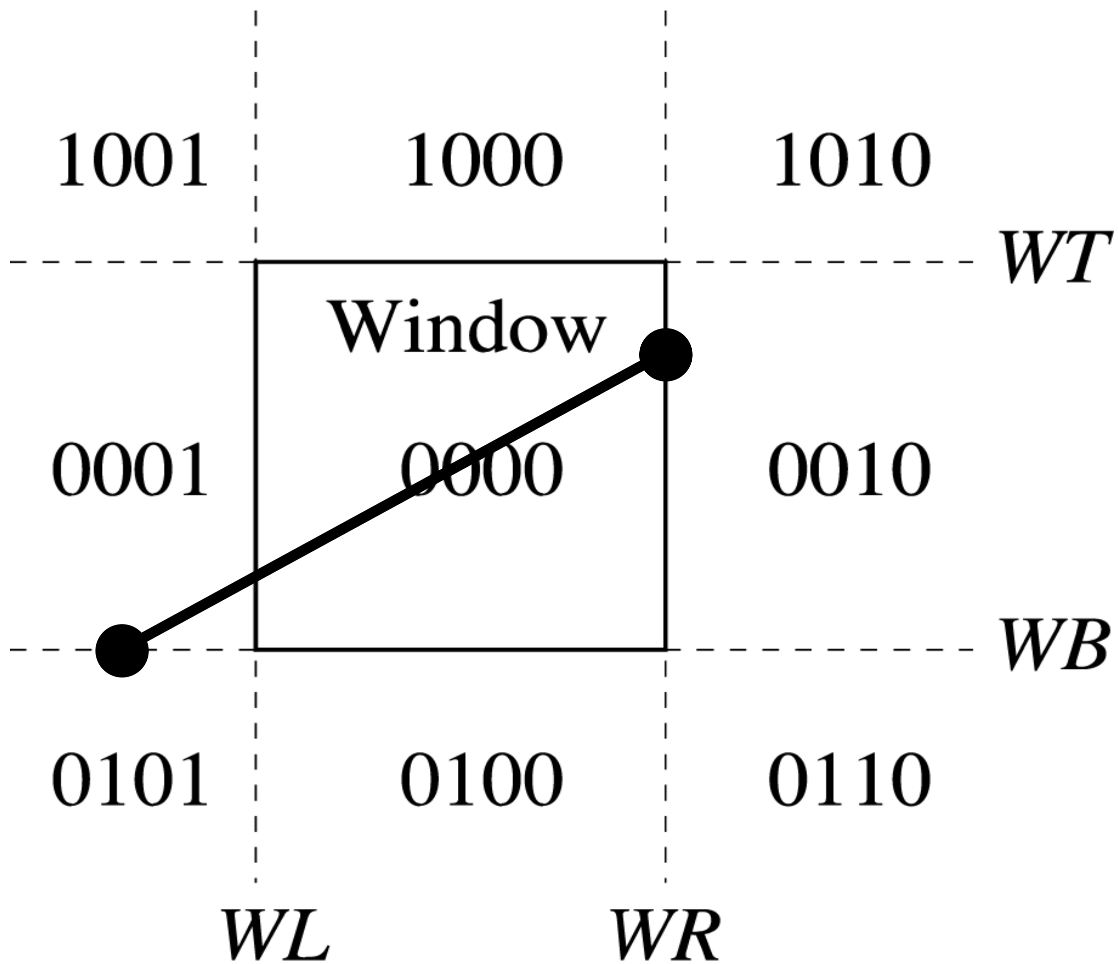
Cohen Sutherland

0001
0010



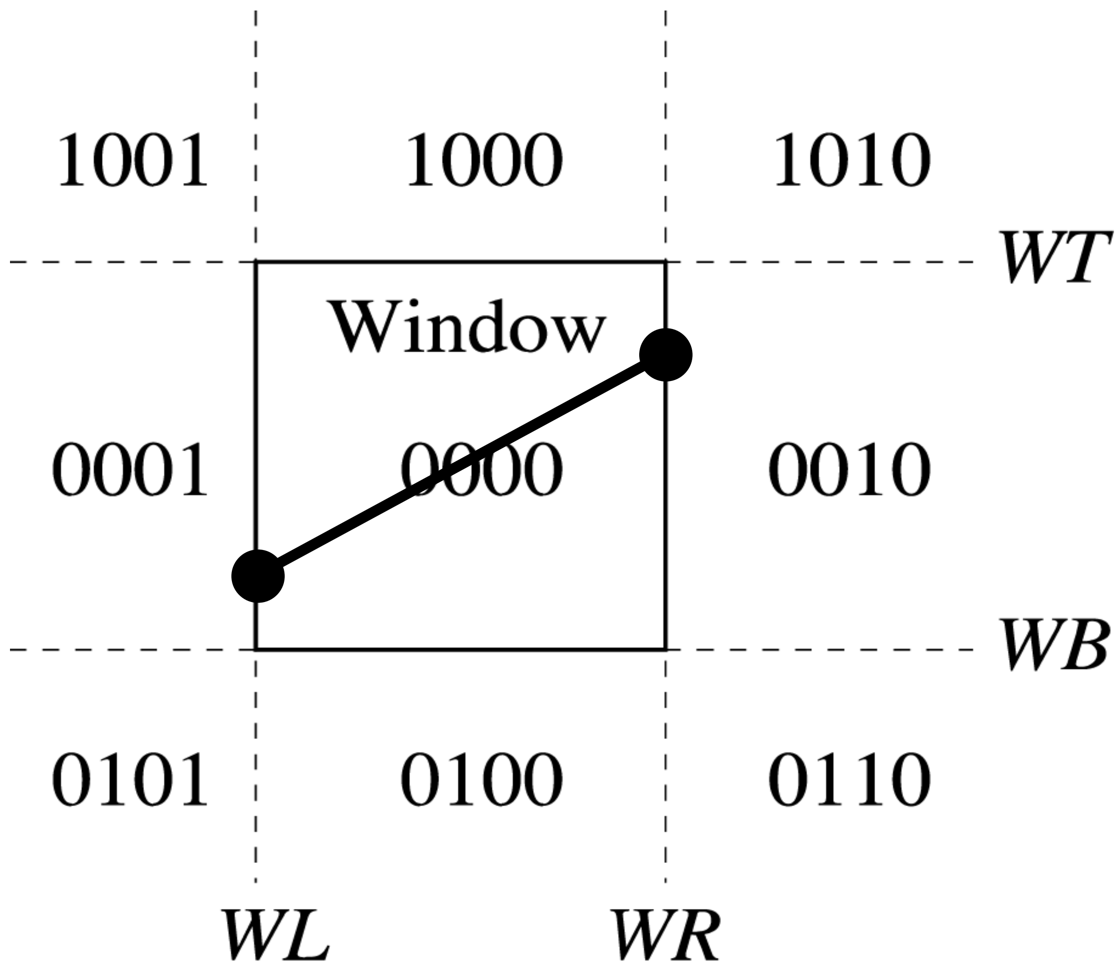
Cohen Sutherland

0001
0000



Cohen Sutherland

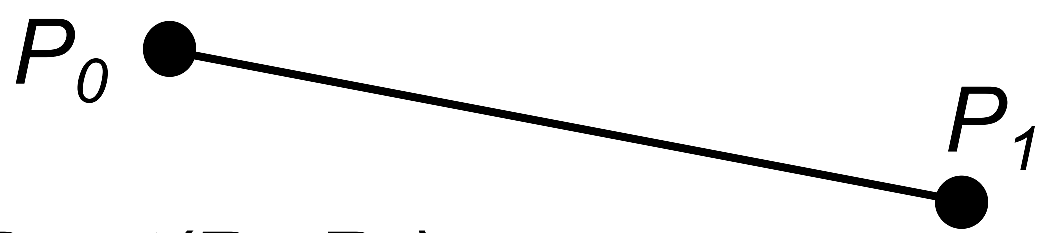
0000
0000



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*
- $P(0) = P_0$ $\qquad$ $P(1) = P_1$

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 ➔ Normal to edge i

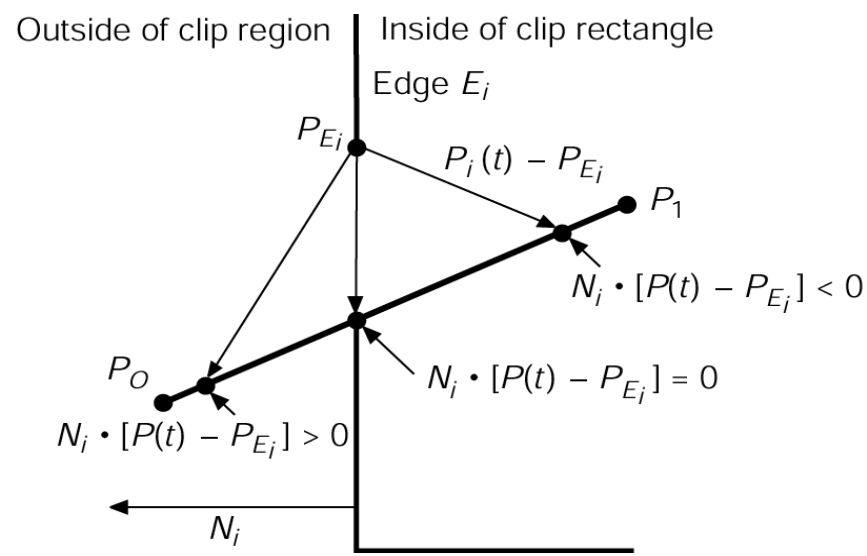
$$N_i \bullet [P(t) - P_{Ei}] = 0$$

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

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

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

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



Make sure
1. $D \neq 0$, or $P_1 \neq P_0$
2. $N_i \bullet D \neq 0$, 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

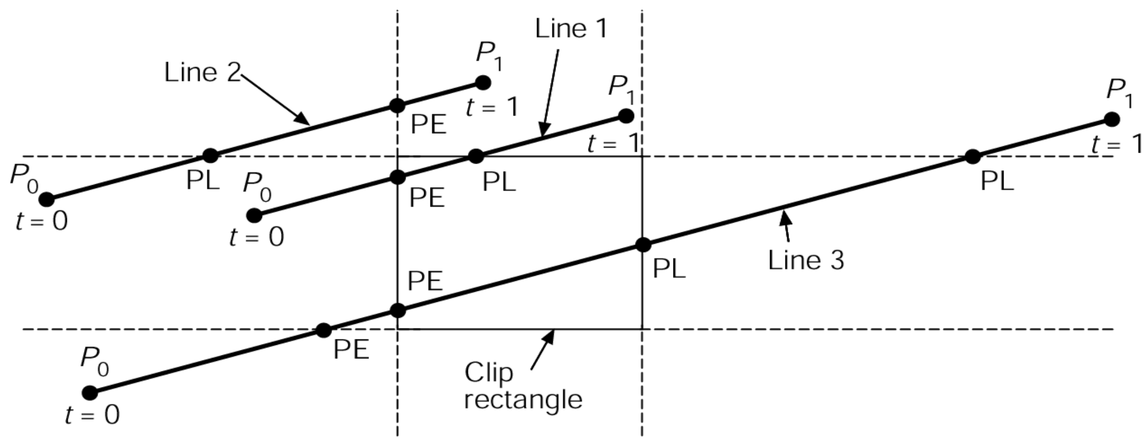
- Calculate edge direction
  - $E = (V_1 - V_0) / |V_1 - V_0|$
  - Be sure to process edges in CCW order
- Rotate direction vector -90°

  $N_x = E_y$

  $N_y = -E_x$

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° => $N_i \bullet D < 0$
- 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)



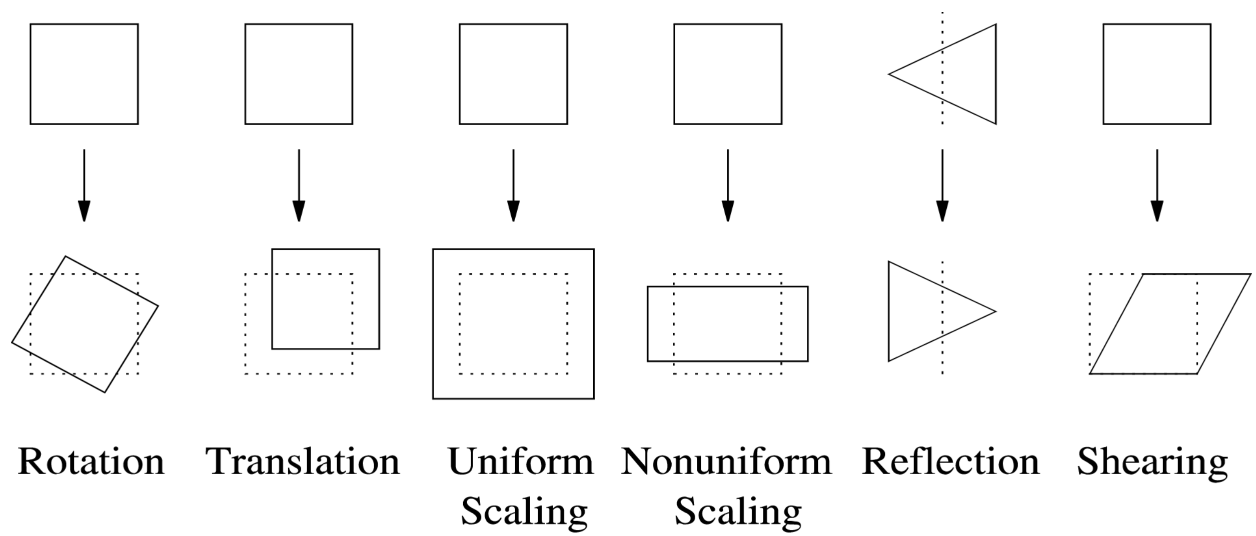
2D Transformations

2D Affine Transformations

All represented as matrix operations on vectors!

Parallel lines preserved, angles/lengths not

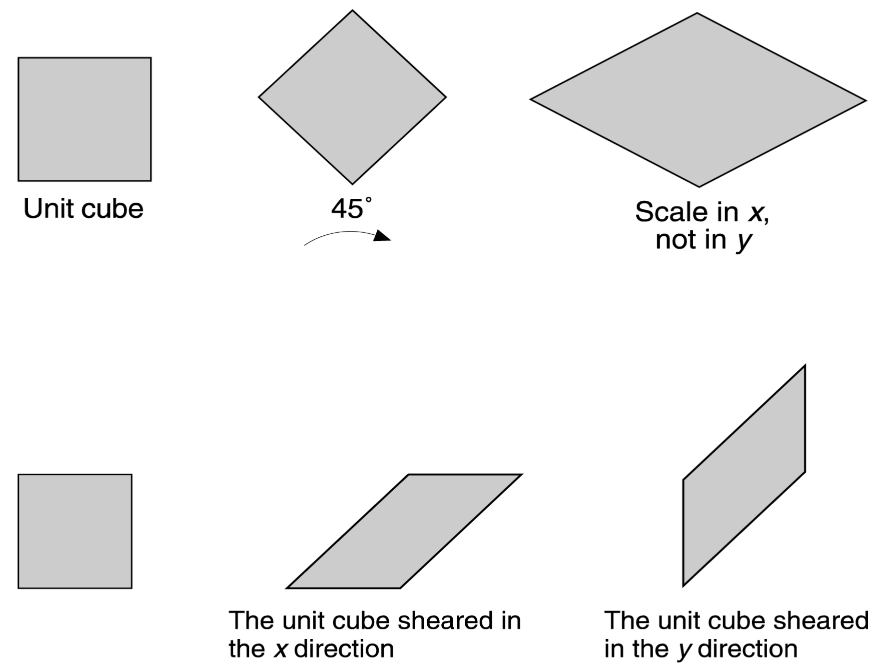
- Scale
- Rotate
- Translate
- Reflect
- Shear



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:

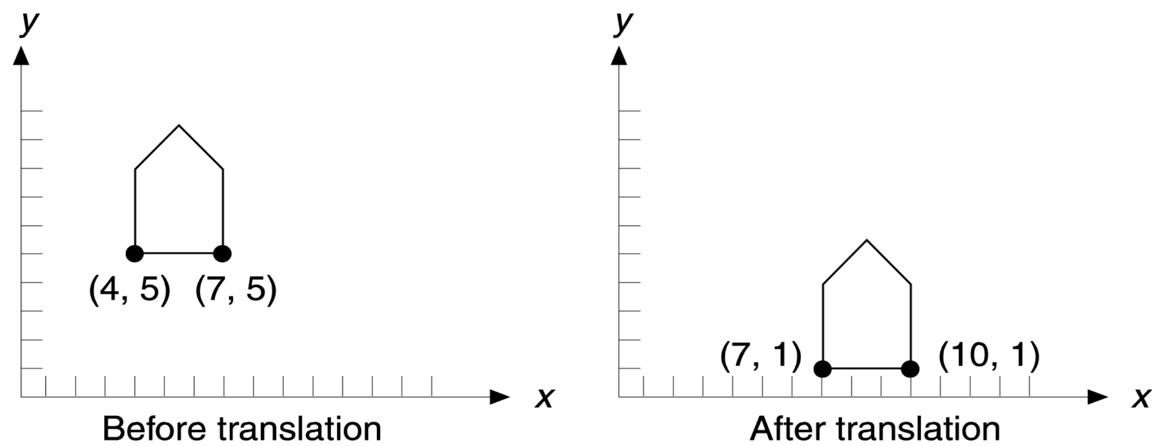
- Preserves parallels
- Does not preserve lengths and angles

2D Transforms: Translation

- Rigid motion of points to new locations

$$x' = x + d_x$$

$$y' = y + d_y$$



- Defined with column vectors:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} d_x \\ d_y \end{bmatrix}$$

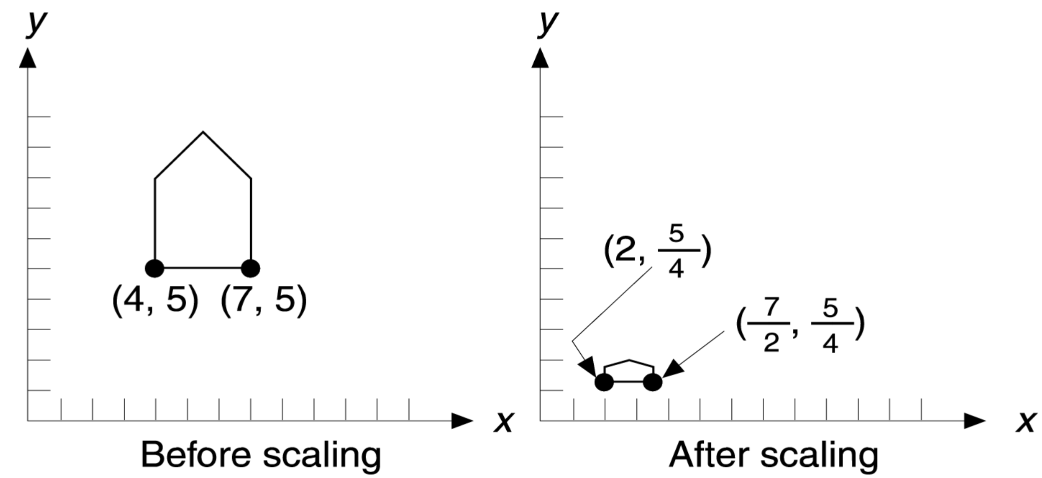
as $P' = P + T$

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: $P' = S \cdot P$

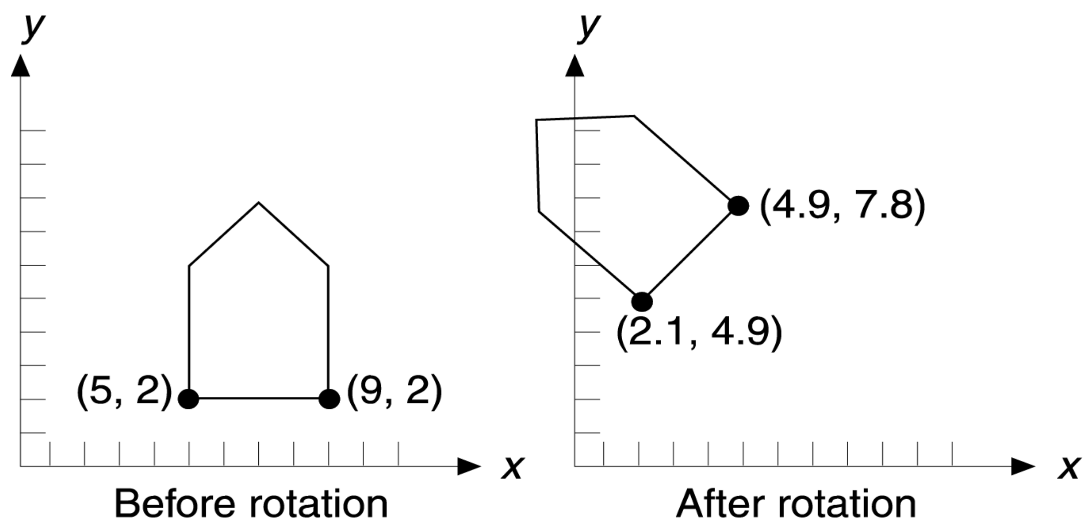
2D Transforms: Rotation

- 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

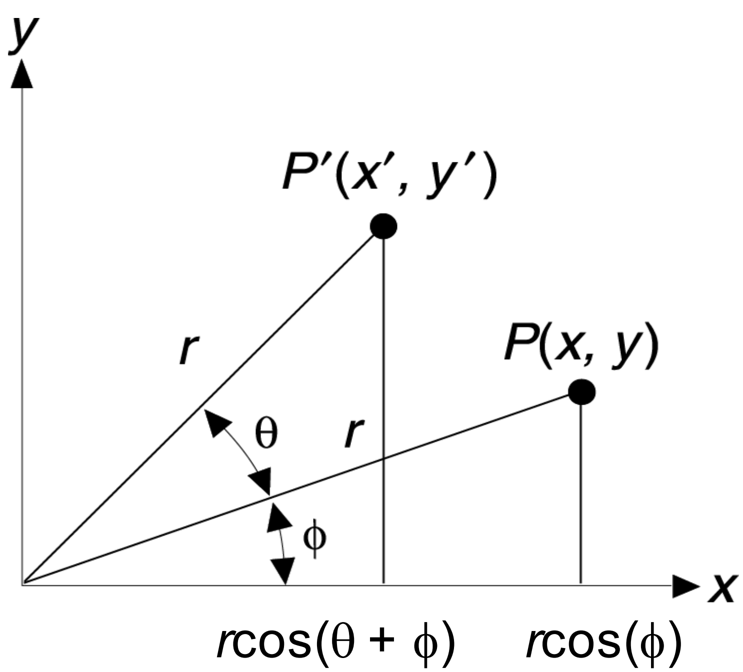


Matrix form: $\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} cos\ \theta & -sin\ \theta \\ sin\ \theta & cos\ \theta \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$

or just: $P' = R \cdot P$

2D Transforms: Rotation

- Substitute the 1st two equations into the 2nd two to get the general equation



$$x = r \cdot cos\ \phi$$

$$y = r \cdot sin\ \phi$$

$$x' = r \cdot cos\ (\theta + \phi) = r \cdot cos\ \phi \cdot cos\ \theta - r \cdot sin\ \phi \cdot sin\ \theta$$

$$y' = r \cdot sin\ (\theta + \phi) = r \cdot cos\ \phi \cdot sin\ \theta + r \cdot sin\ \phi \cdot cos\ \theta$$

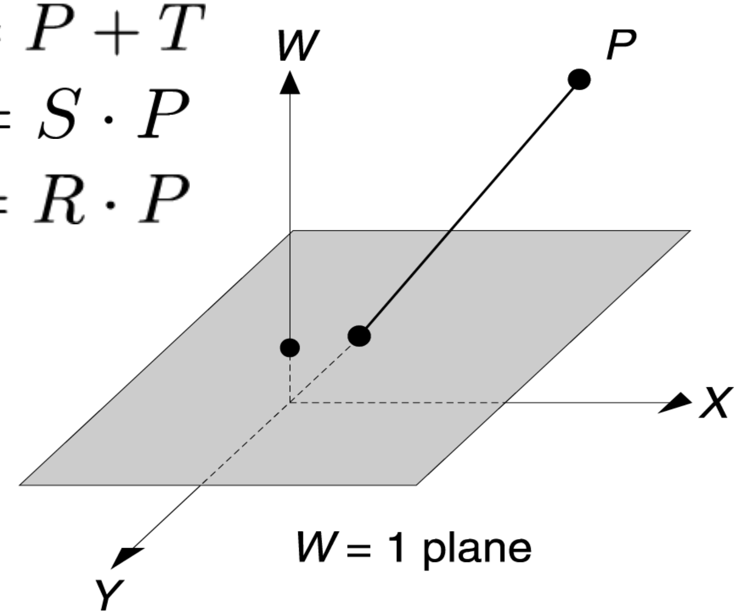
$$x' = x\cos(\theta) - y\sin(\theta)$$

$$y' = x\sin(\theta) + y\cos(\theta)$$

Homogeneous Coordinates

- Observe: *translation* is treated differently from *scaling* and *rotation*
- ***Homogeneous coordinates***: allows *all* transformations to be treated as matrix multiplications

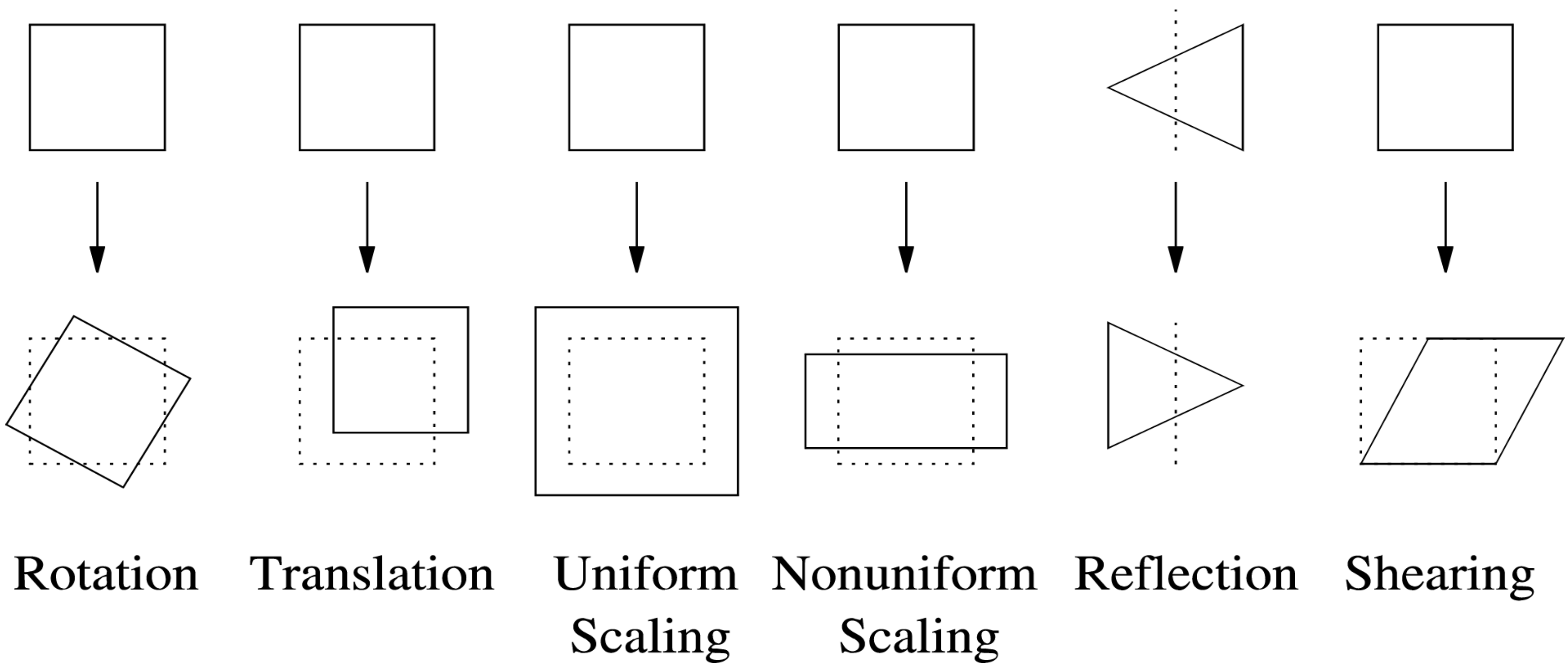
$$P' = P + T$$
$$P' = S \cdot P$$
$$P' = R \cdot P$$



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



Pics/Math courtesy of Dave Mount @ UMD-CP

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: $$F_y = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Translation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\mathbf{x' = x + d_x}$$
$$\mathbf{y' = y + d_y}$$

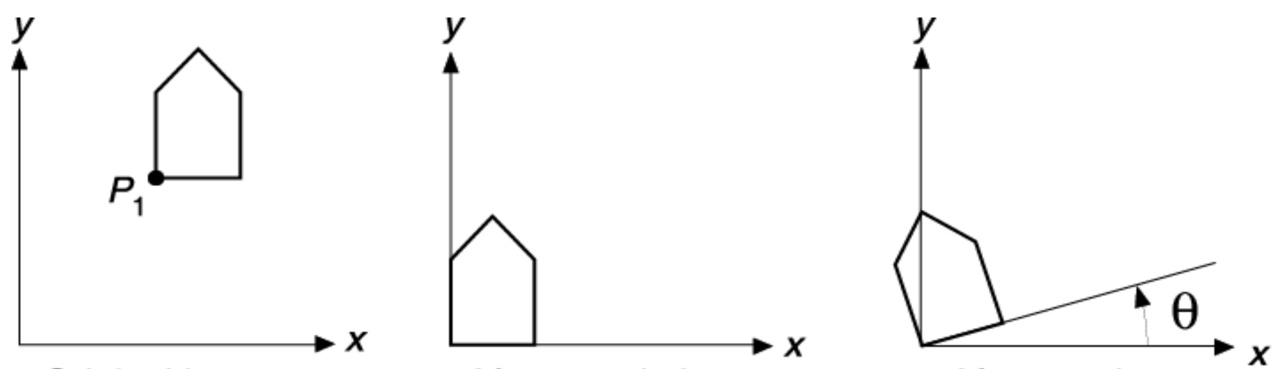
Composition of 2D Transforms

- Rotate about a point *P1*
  - Translate *P1* to origin
  - Rotate
  - Translate back to *P1*

$$T(x_1, y_1) \cdot R(\theta) \cdot T(-x_1, -y_1)$$

$$= \begin{bmatrix} 1 & 0 & x_1 \\ 0 & 1 & y_1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} cos\ \theta & -sin\ \theta & 0 \\ sin\ \theta & cos\ \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_1 \\ 0 & 1 & -y_1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$P' = \mathcal{T} * P \qquad \mathcal{T} = \begin{bmatrix} cos\ \theta & -sin\ \theta & x_1(1 - cos\ \theta) + y_1 sin\ \theta \\ sin\ \theta & cos\ \theta & y_1(1 - cos\ \theta) - x_1 sin\ \theta \\ 0 & 0 & 1 \end{bmatrix}$$



Original house

After translation of P_1 to origin

After rotation

After translation to original P_1

1994 Foley/VanDam/Finer/Huges/Phillips ICG

Composition of 2D Transforms

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

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

$$= \begin{bmatrix} 1 & 0 & x_1 \\ 0 & 1 & y_1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_1 \\ 0 & 1 & -y_1 \\ 0 & 0 & 1 \end{bmatrix}$$

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

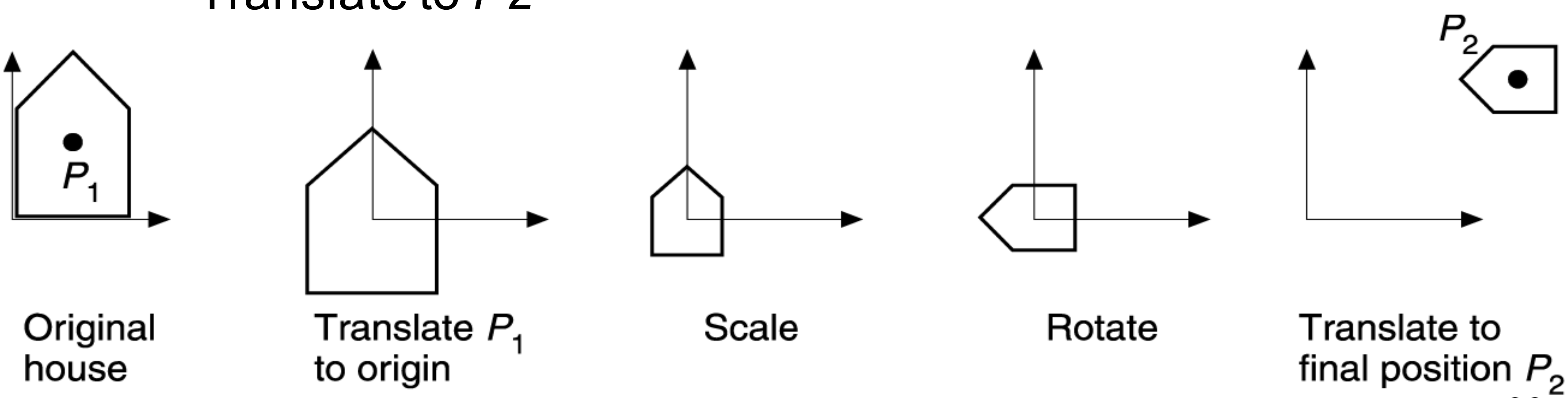
$$P' = \mathcal{T} * P$$

Composition of 2D Transforms

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

$$P' = \mathcal{T} * P$$

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



Composition of 2D Transforms

- Be sure to multiple transformations in proper order!

$$P' = (T * (R * (S * (T * P))))$$

$$P' = ((T * (R * (S * T))) * P)$$

$$P' = \mathcal{T} * P$$

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 or PBM image writer

HW1 Steps

- Read in line segments from file
- Apply transformations
- Clip against world window
- Translate lines into screen/image coordinates
- Draw clipped lines into software frame buffer
- Output frame buffer pixels to standard out either in XPM or PBM format