

## UNIT2

### Two Dimensional Transformations

In many applications, changes in orientations, size, and shape are accomplished with geometric transformations that alter the coordinate descriptions of objects.

Basic geometric transformations are:

- Translation
- Rotation
- Scaling

Other transformations:

- Reflection
- Shear

#### **Basic Transformations**

##### Translation

We translate a 2D point by adding translation distances,  $t_x$  and  $t_y$ , to the original coordinate position  $(x, y)$ :

$$x' = x + t_x, \quad y' = y + t_y$$

Alternatively, translation can also be specified by the following transformation matrix:

$$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

Then we can rewrite the formula as:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

For example, to translate a triangle with vertices at original coordinates  $(10, 20), (10, 10), (20, 10)$  by  $t_x=5, t_y=10$ , we compute as follows:

Translation of vertex  $(10, 20)$ :

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 5 \\ 0 & 1 & 10 \end{bmatrix} \begin{bmatrix} 10 \\ 20 \\ 1 \end{bmatrix} = \begin{bmatrix} 1*10+0*20+5*1 \\ 0*10+1*20+1*1 \end{bmatrix} = \begin{bmatrix} 15 \\ 30 \end{bmatrix}$$

Translation of vertex  $(10, 10)$ :

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 5 \\ 0 & 1 & 10 \end{bmatrix} \begin{bmatrix} 10 \\ 10 \\ 1 \end{bmatrix} = \begin{bmatrix} 1*10+0*10+5*1 \\ 0*10+1*10+1*1 \end{bmatrix} = \begin{bmatrix} 15 \\ 20 \end{bmatrix}$$

Translation of vertex(20,10):

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 5 \\ 0 & 1 & 10 \end{bmatrix} \begin{bmatrix} 20 \\ 10 \\ 1 \end{bmatrix} = \begin{bmatrix} 1*20 + 0*10 + 5*1 \\ 0*20 + 1*10 + 10*1 \end{bmatrix} = \begin{bmatrix} 25 \\ 20 \\ 1 \end{bmatrix}$$

The resultant coordinates of the triangle vertices are (15,30), (15,20), and (25,20) respectively. Exercise: tr

anslate a triangle with vertices at original coordinates (10,25), (5,10), (20,10) by  $x=15, y=5$ . Roughly plot the original and resultant triangles.

### Rotation About the Origin

To rotate an object about the origin (0,0), we specify the rotation angle  $\theta$ . Positive and negative values for the rotation angle define counterclockwise and clockwise rotations respectively. The following is the computation of this rotation for a point:

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

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

Alternatively, this rotation can also be specified by the following transformation matrix:

$$\begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Then we can rewrite the formula as:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

For example, to rotate a triangle about the origin with vertices at original coordinates (10,20), (10,10), (20,10) by 30 degrees, we compute as follows:

$$\begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos 30 & -\sin 30 & 0 \\ \sin 30 & \cos 30 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0.866 & -0.5 & 0 \\ 0.5 & 0.866 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Rotation of vertex (10,20):

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 0.866 & -0.5 & 0 \\ 0.5 & 0.866 & 0 \end{bmatrix} \begin{bmatrix} 10 \\ 20 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.866*10 + (0.5)*20 + 0*1 \\ 0.5*10 + 0.866*20 + 0*1 \end{bmatrix} = \begin{bmatrix} -1.34 \\ 22.32 \\ 1 \end{bmatrix}$$

Rotation of vertex (10,10):

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 0.866 & -0.5 & 0 \\ 0.5 & 0.866 & 0 \\ 1 \end{bmatrix} \begin{bmatrix} 10 \\ 10 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.866*10 + (0.5)*10 & 0*1 \\ 0.5*10 + 0.866*10 + 0 & 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 3.66 \\ 13.66 \\ 1 \end{bmatrix}$$

Rotation of vertex (20,10):

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 0.866 & -0.5 & 0 \\ 0.5 & 0.866 & 0 \\ 1 \end{bmatrix} \begin{bmatrix} 20 \\ 10 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.866*20 + (0.5)*10 & 0*1 \\ 0.5*20 + 0.866*10 + 0 & 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 12.32 \\ 18.66 \\ 1 \end{bmatrix}$$

The resultant coordinates of the triangle vertices are (-1.34, 22.32), (3.6, 13.66), and (12.32, 18.66) respectively.

Exercise: Rotate a triangle with vertices at original coordinates (10,20), (5,10), (20,10) by 45 degrees. Roughly plot the original and resultant triangles.

## Scaling With Respect to the Origin

We scale a 2D object with respect to the origin by setting the scaling factors  $s_x$  and  $s_y$ , which are multiplied to the original vertex coordinate positions (x,y):

$$x' = x * s_x, y' = y * s_y$$

Alternatively, this scaling can also be specified by the following transformation matrix:

$$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Then we can rewrite the formula as:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

For example, to scale a triangle with respect to the origin, with vertices at original coordinates (10,20), (10,10), (20,10) by  $s_x=2$ ,  $s_y=1.5$ , we compute as follows:

Scaling of vertex (10,20):

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1.5 & 0 \\ 1 \end{bmatrix} \begin{bmatrix} 10 \\ 20 \\ 1 \end{bmatrix} = \begin{bmatrix} 2*10 + 0*20 + 0*1 \\ 0*10 + 1.5*20 + 0*1 \\ 1 \end{bmatrix} = \begin{bmatrix} 20 \\ 30 \\ 1 \end{bmatrix}$$

Scaling of vertex(10,10):

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 10 \\ 10 \end{bmatrix} = \begin{bmatrix} 2*10 + 0*10 + 0*1 \\ 0*10 + 1.5*10 + 1*1 \end{bmatrix} = \begin{bmatrix} 20 \\ 15 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 10 \end{bmatrix} + \begin{bmatrix} 0 \\ 10 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Scaling of vertex(20, 10):

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & 1.5 \end{bmatrix} \begin{bmatrix} 20 \\ 10 \end{bmatrix} = \begin{bmatrix} 2*20 + 0*10 + 0*1 \\ 0*20 + 1.5*10 + 1*1 \end{bmatrix} = \begin{bmatrix} 40 \\ 15 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 20 \end{bmatrix} + \begin{bmatrix} 0 \\ 20 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

The resultant coordinates of the triangle vertices are (20,30), (20,15), and (40,15) respectively.

Exercise: Scale a triangle with vertices at original coordinates (10,25), (5,10), (20,10) by  $s_x=1.5$ ,  $s_y=2$ , with respect to the origin. Roughly plot the original and resultant triangles.

## Concatenation Properties of Composite Matrix

I. Matrix multiplication is associative:

$$A \cdot B \cdot C = (A \cdot B) \cdot C = A \cdot (B \cdot C)$$

Therefore, we can evaluate matrix products using these associative groupings.

For example, we have a triangle, we want to rotate it with the matrix B, then we translate it with matrix A.

Then, for a vertex of that triangle represented as C, we compute its transformation as:

$$C' = A \cdot (B \cdot C)$$

But we can also change the computation method as:

$$C' = (A \cdot B) \cdot C$$

The advantage of computing it using  $C' = (A \cdot B) \cdot C$  instead of  $C' = A \cdot (B \cdot C)$  is that, for computing the 3 vertices of the triangle,  $C_1, C_2, C_3$ , the computation time is shortened:

Using  $C' = A \cdot (B \cdot C)$ :

1. compute  $B \cdot C_1$  and put the result into  $I_1$
2. compute  $A \cdot I_1$  and put the result into  $C_1'$
3. compute  $B \cdot C_2$  and put the result into  $I_2$
4. compute  $A \cdot I_2$  and put the result into  $C_2'$
5. compute  $B \cdot C_3$  and put the result into  $I_3$
6. compute  $A \cdot I_3$  and put the result into  $C_3'$

Using  $C' = (A \cdot B) \cdot C$ :

- compute  $A \cdot B$  and put the result into M
- compute  $M \cdot C_1$  and put the result into  $C_1'$
- compute  $M \cdot C_2$  and put the result into  $C_2'$
- compute  $M \cdot C_3$  and put the result into  $C_3'$

Example: Rotate a triangle with vertices (10,20), (10,10), (20,10) about the origin by 30 degrees and then translate it by  $x=5$ ,  $y=10$ ,

We compute the rotation matrix:

$$B = \begin{bmatrix} \cos 30 & -\sin 30 & 0 \\ \sin 30 & \cos 30 & 0 \end{bmatrix} = \begin{bmatrix} 0.866 & -0.5 & 0 \\ 0.5 & 0.866 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$$

And we compute the translation matrix:

$$A = \begin{bmatrix} 1 & 0 & 5 \\ 0 & 1 & 10 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$$

Then, we compute  $M = A \cdot B$

$$M = \begin{bmatrix} 1 & 0 & 5 \\ 0 & 1 & 10 \end{bmatrix} \begin{bmatrix} 0.866 & -0.5 & 0 \\ 0.5 & 0.866 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 * 0.866 & 0 * 0.5 & 5 * 0 \\ 1 * 0.5 & 1 * 0.866 & 10 * 0 \end{bmatrix} + \begin{bmatrix} 0 * 0.866 & 0 * 0.5 & 0 * 0 \\ 0 * 0.5 & 0 * 0.866 & 0 * 0 \end{bmatrix} = \begin{bmatrix} 0.866 & 0 & 0 \\ 0.5 & 0.866 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$$

Then, we compute the transformations of the 3 vertices:

Transformation of vertex (10,20):

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 0.866 & -0.5 & 5 \\ 0.5 & 0.866 & 10 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 10 \\ 20 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.866 * 10 + (0.5) * 20 + 5 * 1 \\ 0.5 * 10 + 0.866 * 20 + 10 * 1 \\ 0 * 10 + 0 * 20 + 1 * 1 \end{bmatrix} = \begin{bmatrix} 3.66 \\ 32.32 \\ 1 \end{bmatrix}$$

Transformation of vertex (10,10):

10,10):

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 0.866 & -0.5 & 5 \\ 0.5 & 0.866 & 10 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 10 \\ 10 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.866 * 10 + (0.5) * 10 + 5 * 1 \\ 0.5 * 10 + 0.866 * 10 + 10 * 1 \\ 0 * 10 + 0 * 10 + 1 * 1 \end{bmatrix} = \begin{bmatrix} 8.66 \\ 23.36 \\ 1 \end{bmatrix}$$

Transformation of vertex (20,10):

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 0.866 & -0.5 & 5 \\ 0.5 & 0.866 & 10 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 20 \\ 10 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.866*20 + (0.5)*10 + 5*1 \\ 0.5*20 + 0.866*10 + 10*1 \\ 0 * 20 + 0 * 10 + 1 \end{bmatrix} = \begin{bmatrix} 17.32 \\ 28.66 \\ 1 \end{bmatrix}$$

The resultant coordinates of the triangle vertices are (3.66, 32.32), (8.66, 23.66), and (17.32, 28.66) respectively.

## II. Matrix multiplication may not be commutative:

$$A \cdot B \text{ may not equal } B \cdot A$$

This means that if we want to translate and rotate an object, we must be careful about the order in which the composite matrix is evaluated. Using the previous example, if you compute  $C' = (A \cdot B) \cdot C$ , you are rotating the triangle with B first, then translate it with A, but if you compute  $C' = (B \cdot A) \cdot C$ , you are translating it with A first, then rotate it with B. The result is different.

Exercise: Translate a triangle with vertices (10,20), (10,10), (20,10) by  $t_x=5$ ,  $t_y=10$  and then rotate it about the origin by 30 degrees. Compare the result with the one obtained previously: (3.66,32.32), (8.66,23.66), and (17.32,28.66) by plotting the original triangle together with these 2 results.

## Composite Transformation Matrix

### Translations

By common sense, if we translate a shape with 2 successive translation vectors:  $(t_{x1}, t_{y1})$  and  $(t_{x2}, t_{y2})$ , it is equal to a single translation of  $(t_{x1}+t_{x2}, t_{y1}+t_{y2})$ .

This additive property can be demonstrated by composite transformation matrix:

$$\begin{aligned} & \begin{bmatrix} 1 & 0t_{x1} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 10t_{x2} \\ 1 \end{bmatrix} = \begin{bmatrix} 1*1 + 0*0 & t_{x1}*01*0 \\ 0*t_{x2} & 1 \end{bmatrix} = \begin{bmatrix} 1 & t_{x1} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 10 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 10 + t_{x1} & 0 \\ 0 & 1 \end{bmatrix} \\ & = \begin{bmatrix} 1 & t_{x1} + t_{x2} \\ 0 & 1 \end{bmatrix} \end{aligned}$$

This demonstrates that 2 successive translations are additive.

## Rotations

By common sense, if we rotate a shape with 2 successive rotation angles:  $\theta$  and  $\alpha$ , about the origin, it is equal to rotating the shape once by an angle  $\theta + \alpha$  about the origin.

Similarly, this additive property can be demonstrated by composite transformation matrix:

$$\begin{aligned}
 & \left[ \begin{array}{ccc} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{array} \right] \left[ \begin{array}{ccc} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{array} \right] \\
 & = \left[ \begin{array}{ccc} \cos \theta \cos \alpha + (\sin \theta) * \sin \alpha & \cos \theta * (\sin \alpha) + (\sin \theta) * \cos \alpha & 0 \\ \sin \theta \cos \alpha + \cos \theta * \sin \alpha & \sin \theta * (\sin \alpha) + \cos \theta * \cos \alpha & 0 \\ 0 * \cos \alpha + 0 * \sin \alpha + 1 * 0 & 0 * (\sin \alpha) + 0 * \cos \alpha + 1 * 0 & 0 * 0 + 0 * 0 + 1 * 1 \end{array} \right] \\
 & = \left[ \begin{array}{ccc} \cos(\theta + \alpha) & -\sin(\theta + \alpha) & 0 \\ \sin(\theta + \alpha) & \cos(\theta + \alpha) & 0 \\ 0 & 0 & 1 \end{array} \right]
 \end{aligned}$$

This demonstrates that 2 successive rotations are additive.

## Scalings With Respect to the Origin

By common sense, if we scale a shape with 2 successive scaling factor:  $(s_{x1}, s_{y1})$  and  $(s_{x2}, s_{y2})$ , with respect to the origin, it is equal to a single scaling of  $(s_{x1} * s_{x2}, s_{y1} * s_{y2})$  with respect to the origin. This multiplicative property can be demonstrated by composite transformation matrix:

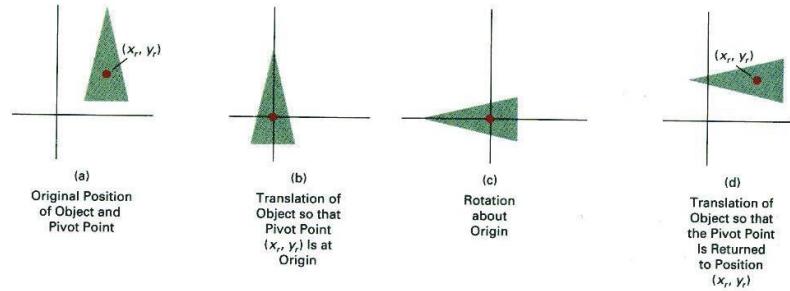
$$\begin{aligned}
 & \left[ \begin{array}{ccc} s_{x1} & 0 & 0 \\ 0 & s_{y1} & 0 \\ 0 & 0 & 1 \end{array} \right] \left[ \begin{array}{ccc} s_{x2} & 0 & 0 \\ 0 & s_{y2} & 0 \\ 0 & 0 & 1 \end{array} \right] \\
 & = \left[ \begin{array}{ccc} s_{x1} * s_{x2} & 0 * 0 + 0 * 0 * s_{x1} * 0 & 0 * s_{y2} + 0 * 0 * s_{x1} * 0 & 0 * 0 + 0 * 1 \\ 0 * s_{x2} + s_{y1} * 0 & 0 * 0 & 0 * 0 & 0 * 0 \\ 0 * s_{y2} + 0 * 0 * s_{x1} * 0 & 0 * 0 + 0 * s_{y2} & 0 * 0 + 1 * 0 & 0 * 0 \\ x2 & y2 & 1 & 1 \end{array} \right] \\
 & = \left[ \begin{array}{ccc} s_{x1} * s_{x2} & 0 & 0 \\ 0 & s_{y1} * s_{y2} & 0 \\ 0 & 0 & 1 \end{array} \right]
 \end{aligned}$$

This demonstrates that 2 successive scalings with respect to the origin are multiplicative.

## General Pivot-Point Rotation

Rotation about an arbitrary pivot point is not as simple as rotation about the origin. The procedure of rotation about an arbitrary pivot point is:

- Translate the object so that the pivot-point position is moved to the origin.
- Rotate the object about the origin.
- Translate the object so that the pivot point is returned to its original position.



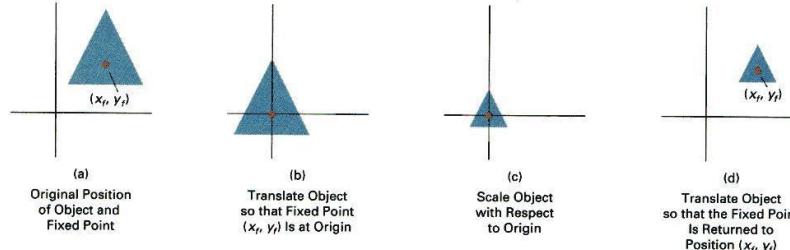
The corresponding composite transformation matrix is:

$$\begin{aligned}
 & \begin{bmatrix} 1 & 0 & x_r \\ 0 & 1 & y_r \end{bmatrix} \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ \sin\theta & -\cos\theta & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_r \\ 0 & 1 & -y_r \end{bmatrix} \\
 & = \begin{bmatrix} 0 & 0 & 1 \\ \cos\theta & -\sin\theta & x_r \\ \sin\theta & \cos\theta & y_r \end{bmatrix} \begin{bmatrix} 1 & 0 & \theta \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
 & = \begin{bmatrix} 0 & 0 & 1 \\ \cos\theta & -\sin\theta & -x_r \cos\theta + y_r \sin\theta + x_r \\ \sin\theta & \cos\theta & -x_r \sin\theta - y_r \cos\theta + y_r \end{bmatrix}
 \end{aligned}$$

## General Fixed-Point Scaling

Scaling with respect to an arbitrary fixed point is not as simple as scaling with respect to the origin. The procedure of scaling with respect to an arbitrary fixed point is:

1. Translate the object so that the fixed point coincides with the origin.
2. Scale the object with respect to the origin.
3. Use the inverse translation of step 1 to return the object to its original position.



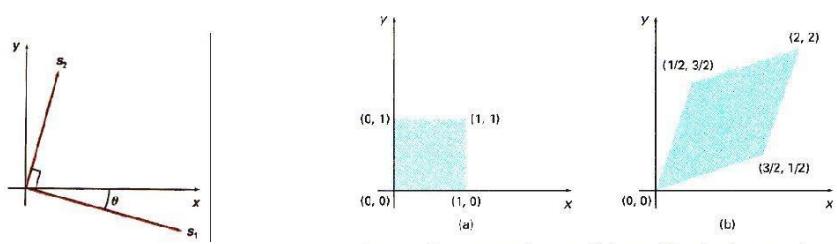
The corresponding composite transformation matrix is:

$$\begin{bmatrix} 1 & 0 & x_f \\ 0 & 1 & y_f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_f \\ 0 & 1 & -y_f \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & x_f(1-s_x) \\ 0 & s_y & y_f(1-s_y) \\ 0 & 0 & 1 \end{bmatrix}$$

### General Scaling Direction

Scaling along an arbitrary direction is not as simple as scaling along the x-y axis. The procedure of scaling along and normal to an arbitrary direction ( $s_1$  and  $s_2$ ), with respect to the origin, is:

1. Rotate the object so that the directions for  $s_1$  and  $s_2$  coincide with the x and y axes respectively.
2. Scale the object with respect to the origin using  $(s_1, s_2)$ .
3. Use an opposite rotation to return points to their original orientation.



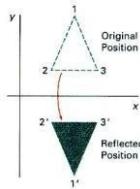
A square (a) is converted to a parallelogram (b) using the composite transformation matrix 5-35, with  $s_1 = 1$ ,  $s_2 = 2$ , and  $\theta = 45^\circ$ .

The corresponding composite transformation matrix is:

$$\begin{bmatrix} \cos(-\theta) & -\sin(-\theta) & 0 \\ \sin(-\theta) & \cos(-\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_1 & 0 & 0 \\ 0 & s_2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

### Other Transformations Reflec

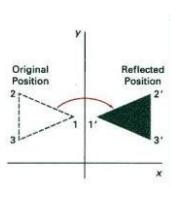
#### tion



Reflection about the x-axis:

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

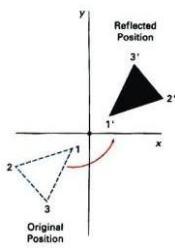
i.e.  $x' = x$ ;  $y' = -y$



Reflection about the y-axis:

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

i.e.  $x' = -x$ ;  $y' = y$

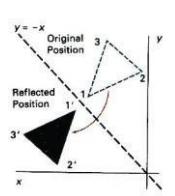


Flipping both x and y coordinates of a point relative to the origin:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -10 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

i.e.  $x' = -x$ ;  $y' = -y$

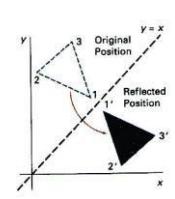


Reflection about the diagonal line  $y = x$ :

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

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

i.e.  $x' = y$ ;  $y' = x$



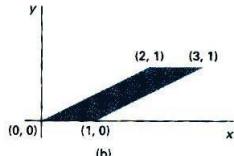
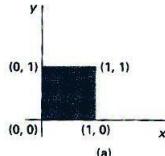
Reflection about the diagonal line  $y = -x$ :

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 \\ -10 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

i.e.  $x' = -y$ ;  $y' = -x$

## Shear



X-direction shear, with a shearing parameter  $sh_x$ , relative to the x-axis:

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

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

i.e.  $x' = x + y * sh_x$ ;  $y' = -y$

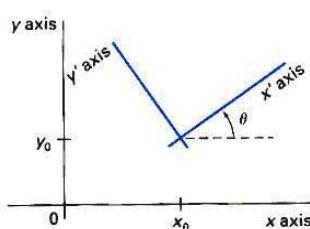
Exercise: Think of a y-direction shear, with a shearing parameter  $sh_y$ , relative to the y-axis.

## Transformation Between 2 Cartesian Systems

For modelling and design applications, individual objects may be defined in their own local Cartesian References. The local coordinates must then be transformed to position the objects within the overall scene coordinate system.

Suppose we want to transform object descriptions from the  $xy$  system to the  $x'y'$  system:

The composite transformation is:



$$\begin{bmatrix} \cos(-\theta) & -\sin(-\theta) & x_r \\ \sin(-\theta) & \cos(-\theta) & y_r \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_0 \\ 0 & 1 & -y_0 \\ 0 & 0 & 1 \end{bmatrix}$$

## 2-Dimensional viewing

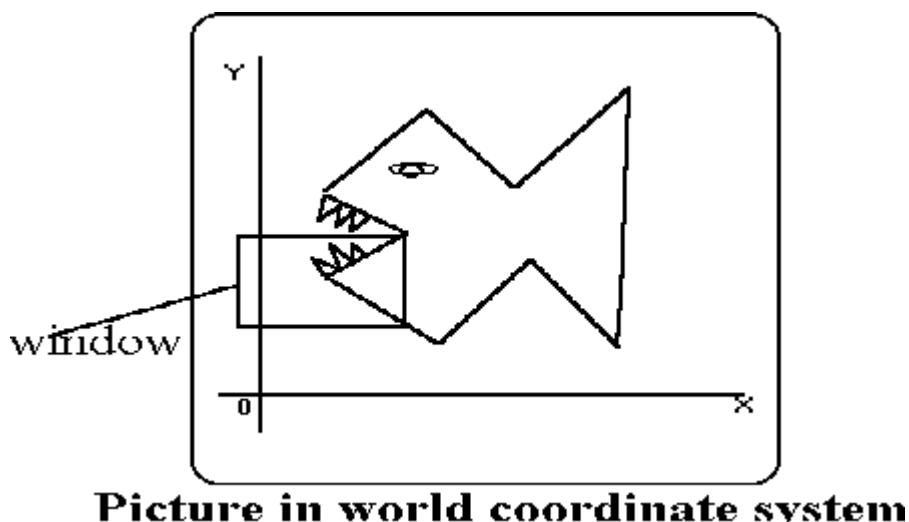
### Images on the Screen

All objects in the real world have size. We use a unit of measure to describe both the size of an object as well as the location of the object in the real world. For example, meters can be used to specify both size and distance. When showing an image of an object on the screen, we use a screen coordinate system that defines the location of the object in the same relative position as in the real world. After we select the screen coordinate system, we change the picture to display in the screen that means change it into screen coordinate system.

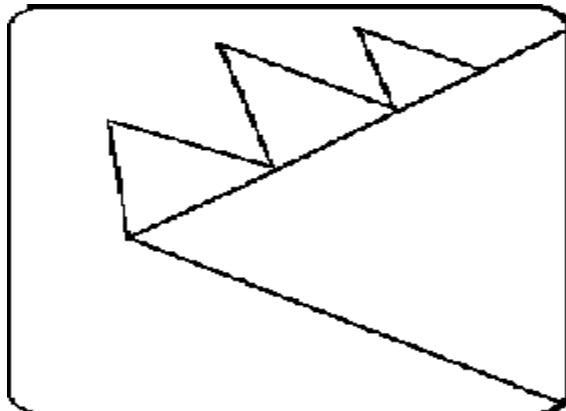
### Windows and Clipping

The world coordinate system is used to define the position of objects in the natural world. This system does not depend on the screen coordinate system, so the interval of numbers can be anything (positive, negative or decimal). Sometimes the complete picture of an object in the world coordinate system is too large and complicated to clearly show on the screen, and we need to show only some part of the object. The capability that shows some part of the object internal to a specified window is called windowing and a rectangular region in a world coordinate system is called a window. Before going into clipping, you should understand the differences between **window** and **viewport**.

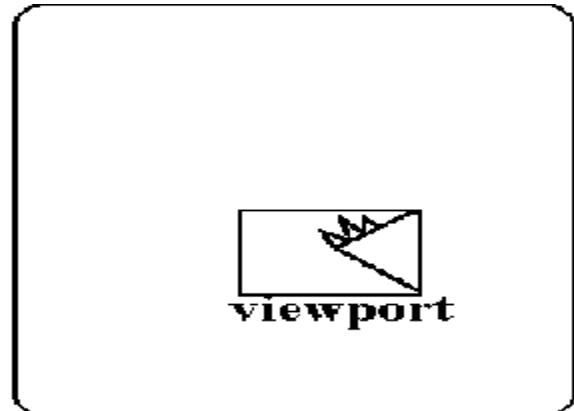
A **Window** is a rectangular region in the **world coordinate system**. This is the coordinate system used to locate an object in the natural world. The world coordinate system does not depend on a display device, so the units of measure can be positive, negative or decimal numbers.



A **Viewport** is the section of the screen where the images encompassed by the window on the world coordinate system will be drawn. A coordinate transformation is required to display the image, encompassed by the window, in the viewport. The viewport uses the **screen coordinate system** so this transformation is from the world coordinate system to the screen coordinate system.



**Screen**



**Screen**

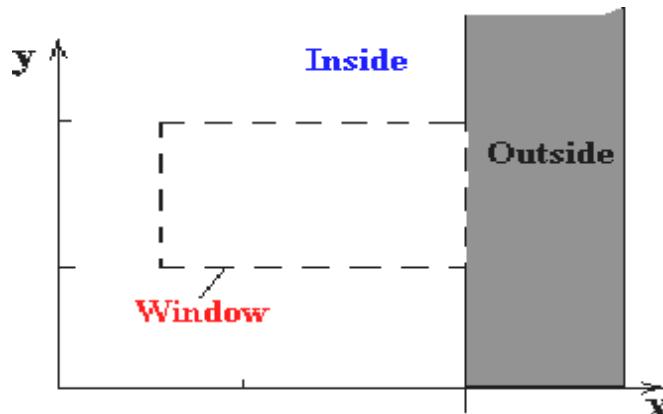
When a window is "placed" on the world, only certain objects and parts of objects can be seen. Points and lines which are outside the window are "cut off" from view. This process of "cutting off" parts of the image of the world is called **Clipping**. In clipping, we examine each line to determine whether or not it is completely inside the window, completely outside the window, or crosses a window boundary. If inside the window, the line is displayed. If outside the window, the lines and points are not displayed. If a line crosses the boundary, we must determine the point of intersection and display only the part which lies inside the window.

### Cohen-Sutherland Line Clipping

The Cohen-Sutherland line clipping algorithm quickly detects and dispenses with two common and trivial cases. To clip a line, we need to consider only its endpoints. If both endpoints of a line lie inside the window, the entire line lies inside the window. It is trivially accepted and needs no clipping. On the other hand, if both endpoints of a line lie entirely to one side of the window, the line must lie entirely outside of the window. It is trivially rejected and needs to be neither clipped nor displayed.

#### Inside-Outside Window Codes

To determine whether endpoints are inside or outside a window, the algorithm sets up a **half-space code** for each endpoint. Each edge of the window defines an infinite line that divides the whole space into two half-spaces, the **inside half-space** and the **outside half-space**, as shown below.



As you proceed around the window, extending each edge and defining an inside half-space and an outside half-space, nine regions are created - the eight "outside" regions and the one "inside" region. Each of the nine regions associated with the window is assigned a 4-bit code to identify the region. Each bit in the code is set to either a **1**(true) or a **0**(false). If the region is to the **left** of the window, the **first** bit of the code is set to 1. If the region is to the **top** of the window, the **second** bit of the code is set to 1. If to the **right**, the **third** bit is set, and if to the **bottom**, the **fourth** bit is set. The 4 bits in the code then identify each of the nine regions as shown below.

<b>1001</b>	<b>0001</b>	<b>0101</b>
<b>1000</b>	<b>0000</b>	<b>0100</b>
<b>Window</b>		
<b>1010</b>	<b>0010</b>	<b>0110</b>

For any endpoint ( $x, y$ ) of a line, the code can be determined that identifies which region the endpoint lies. The code's bits are set according to the following conditions:

- First bit set **1** : Point lies to **left** of window  $x < x_{\min}$
- Second bit set **1** : Point lies to **right** of window  $x > x_{\max}$
- Third bit set **1** : Point lies below(**bottom**) window  $y < y_{\min}$
- fourth bit set **1** : Point lies above(**top**) window  $y > y_{\max}$

The sequence for reading the codes' bits is **LRBT**(Left, Right, Bottom, Top).

Once the codes for each endpoint of a line are determined, the logical **AND** operation of the codes determines if the line is completely outside of the window. If the logical AND of the endpoint codes is **not zero**, the line can be trivially rejected. For example, if an endpoint had a code of 1001 while the other endpoint had a code of 1010, the logical AND would be 1000 which indicates the line segment lies outside of the window. On the other hand, if the endpoint had codes of 1001 and 0110, the logical AND would be 0000, and the line could not be trivially rejected.

The logical **OR** of the endpoint codes determines if the line is completely inside the window. If the logical OR is **zero**, the line can be trivially accepted. For example, if the endpoint codes are 0000 and 0000, the logical OR is 0000 - the line can be trivially accepted. If the endpoint codes are 0000 and 0110, the logical OR is 0110 and the line cannot be trivially accepted.

## Algorithm

The Cohen-Sutherland algorithm uses a divide-and-conquer strategy. The line segment's endpoints are tested to see if the line can be trivially accepted or rejected. If the line cannot be trivially accepted or rejected, an intersection of the line with a window edge is determined and the trivial reject/accept test is repeated. This process is continued until the line is accepted.

To perform the trivial acceptance and rejection tests, we extend the edges of the window to divide the plane of the window into seven regions. Each end point of the line segment is then assigned the code of the region in which it lies.

1. Given a line segment with endpoint  $P_1 = (x_1, y_1)$  and  $P_2 = (x_2, y_2)$
2. Compute the 4-bit codes for each endpoint.

If both codes are **0000**, (bitwise OR of the codes yields 0000) line lies completely **inside** the window: pass the endpoints to the draw routine.

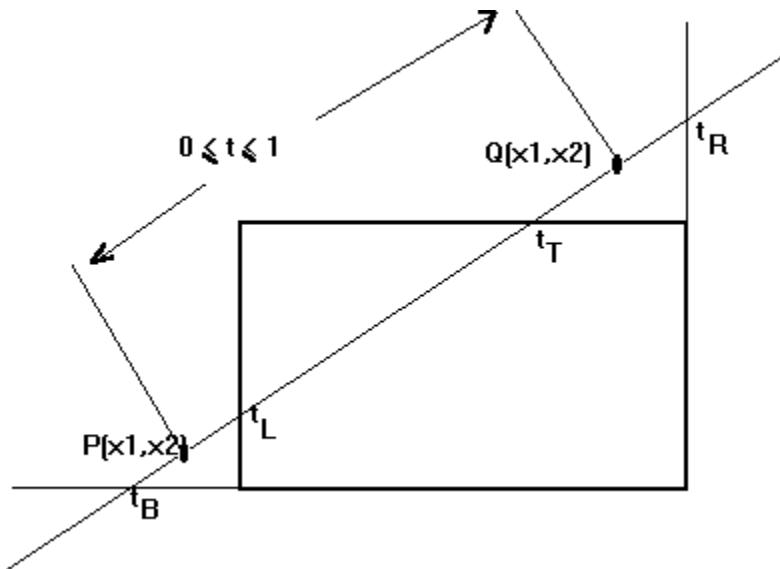
If both codes have a 1 in the same bit position (bitwise AND of the codes is **not** 0000), the line lies **outside** the window. It can be trivially rejected.

3. If a line cannot be trivially accepted or rejected, at least one of the two endpoints must lie outside the window and the line segment crosses a window edge. This line must be **clipped** at the window edge before being passed to the drawing routine.
4. Examine one of the endpoints, say  $P_1 = (x_1, y_1)$ . Read  $P_1$ 's 4-bit code in order: **Left-to-Right, Bottom-to-Top**.
5. When a set bit (1) is found, compute the intersection  $I$  of the corresponding window edge with the line from  $P_1$  to  $P_2$ . Replace  $P_1$  with  $I$  and repeat the algorithm.

## Liang-Barsky Line Clipping

The ideas for clipping line of Liang-Barsky and Cyrus-Beck are the same. The only difference is Liang-Barsky algorithm has been optimized for an upright rectangular clip window. So we will study only the idea of **Liang-Barsky**.

Liang and Barsky have created an algorithm that uses floating-point arithmetic but finds the appropriate end points with at most four computations. This algorithm uses the parametric equations for a line and solves four inequalities to find the range of the parameter for which the line is in the viewport.



Let  $P(x_1, y_1)$ ,  $Q(x_2, y_2)$  be the line which we want to study. The **parametric equation of the line segment** from gives x-values and y-values for every point in terms of a **parameter** that ranges from 0 to 1. The equations are

$$x = x_1 + (x_2 - x_1)*t = x_1 + dx*t \quad \text{and} \quad y = y_1 + (y_2 - y_1)*t = y_1 + dy*t$$

We can see that when  $t = 0$ , the point computed is  $P(x_1, y_1)$ ; and when  $t = 1$ , the point computed is  $Q(x_2, y_2)$ .

## Algorithm

1. Set  $t_{\min} = 0$  and  $t_{\max} = 1$
2. Calculate the values of  $t_L, t_R, t_T$ , and  $t_B$  (values).
  - o if  $t < t_{\min}$  or  $t > t_{\max}$  ignore it and go to the next edge
  - o otherwise classify the  $t$  value as entering or exiting value (using inner product to classify)
    - o if it is entering value set  $t_{\min} = t$ ; if it is exiting value set  $t_{\max} = t$
3. If  $t_{\min} < t_{\max}$  then draw a line from  $(x_1 + dx*t_{\min}, y_1 + dy*t_{\min})$  to  $(x_1 + dx*t_{\max}, y_1 + dy*t_{\max})$
4. If the line crosses over the window, you will see  $(x_1 + dx*t_{\min}, y_1 + dy*t_{\min})$  and  $(x_1 + dx*t_{\max}, y_1 + dy*t_{\max})$  are intersection between line and edge.

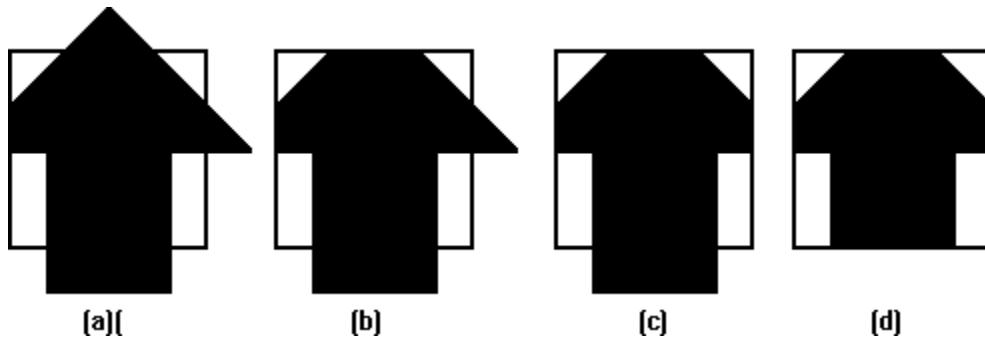
## Sutherland-Hodgman Polygon Clipping

The Sutherland - Hodgman algorithm performs a clipping of a polygon against each window edge in turn. It accepts an ordered sequence of vertices  $v_1, v_2, v_3, \dots, v_n$  and puts out a set of vertices defining the clipped polygon.



**Before clipping** This figure represents a polygon (the large, solid, upward-pointing arrow) before clipping has occurred.

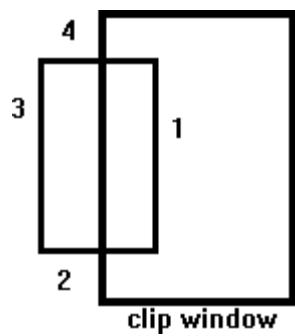
The following figures show how this algorithm works at each edge, clipping the polygon.



- a. Clipping against the left side of the clip window.
- b. Clipping against the top side of the clip window.
- c. Clipping against the right side of the clip window.
- d. Clipping against the bottom side of the clip window.

### Four Types of Edges

As the algorithm goes around the edges of the window, clipping the polygon, it encounters four types of edges. All four edge types are illustrated by the polygon in the following figure. For each edge type, zero, one, or two vertices are added to the output list of vertices that define the clipped polygon.



The four types of edges are:

1. Edges that are totally inside the clip window. - add the second inside vertex point
2. Edges that are leaving the clip window. - add the intersection point as a vertex
3. Edges that are entirely outside the clip window. - add nothing to the vertex output list
4. Edges that are entering the clip window. - save the intersection and inside points as vertices

### How To Calculate Intersections

Assume that we're clipping a polygon's edge with vertices at  $(x_1, y_1)$  and  $(x_2, y_2)$  against a clip window with vertices at  $(x_{min}, y_{min})$  and  $(x_{max}, y_{max})$ .

The location  $(IX, IY)$  of the intersection of the edge with the left side of the window is:

- i.  $IX = x_{min}$
- ii.  $IY = \text{slope} * (x_{min} - x_1) + y_1$ , where the slope  $= (y_2 - y_1) / (x_2 - x_1)$

The location of the intersection of the edge with the right side of the window is:

- i.  $IX = x_{max}$
- ii.  $IY = \text{slope} * (x_{max} - x_1) + y_1$ , where the slope  $= (y_2 - y_1) / (x_2 - x_1)$

The intersection of the polygon's edge with the top side of the window is:

- i.  $IX = x_1 + (y_{max} - y_1) / \text{slope}$
- ii.  $IY = y_{max}$

Finally, the intersection of the edge with the bottom side of the window is:

- i.  $IX = x_1 + (y_{min} - y_1) / \text{slope}$
- ii.  $IY = y_{min}$

### **SomeProblemsWithThisAlgorithm**

1. This algorithm does not work if the clip window is not convex.
2. If the polygon is not also convex, there may be some dangling edges.