MODULE 2
Two-Dimensional Geometric Transformations, 2D VIEWING

- Operations that are applied to the geometric description of an object to change its position, orientation, or size are called **geometric transformations.**
- Sometimes geometric transformations are also referred to as ***modeling transformations,*** *but some graphics packages make a* distinction between the two.
- **Modeling transformations** are used to **construct a scene** or to give the hierarchical description of a complex object that is composed of several parts.
  - For example, an aircraft consists of wings, tail, fuselage, engine, and other components and so on.
- **Geometric transformations,** on the other hand, can be used to **describe how objects might move around in a scene** during an animation sequence or simply to view them from another angle.

Basic Two-Dimensional Geometric Transformations

- Basic geometric transformation in graphics are: **translation, rotation & scaling.**
- Other transformation routines are: **reflection & shearing.**

Two-Dimensional Translation

- We perform a **translation on a single coordinate point by adding offsets to its** coordinates so as to **generate a new coordinate position**.
- In effect, we are moving the original point position along a straight-line path to its new location.
- If multiple coordinates are the **then add offsets to all vertices**

- To translate a two-dimensional position, we add **translation distances *tx*** and ***ty*** *to the* ***original coordinates (x, y)*** *to obtain the* ***new coordinate position (x', y')*** as shown in Figure 1.

$$x' = x + t_x, \qquad y' = y + t_y \tag{1}$$

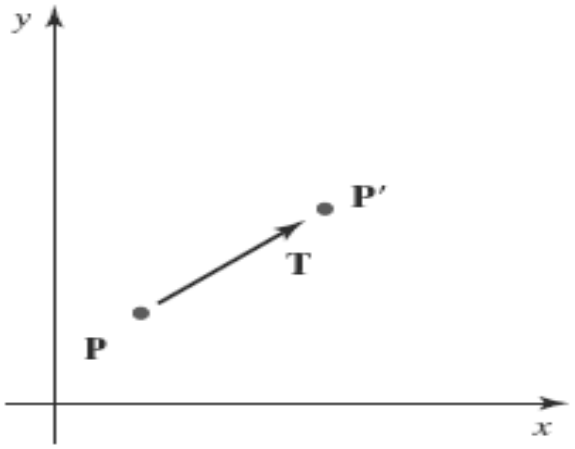


FIGURE 1
Translating a point from position **P** to position **P'** using a translation vector **T**.

- The translation distance pair (*tx, ty) is called a* ***translation vector or shift vector.***

- We can express **Equations 1** as a **single matrix equation** by using the following **column vectors** to *represent coordinate positions and the translation vector:*

$$\mathbf{P} = \begin{bmatrix} x \\ y \end{bmatrix}, \qquad \mathbf{P}' = \begin{bmatrix} x' \\ y' \end{bmatrix}, \qquad \mathbf{T} = \begin{bmatrix} t_x \\ t_y \end{bmatrix} \tag{2}$$

- This allows us to write the two-dimensional translation equations in the matrix form

$$\mathbf{P}' = \mathbf{P} + \mathbf{T} \tag{3}$$

- Translation is a *rigid-body transformation that* ***moves objects without deformation.***
- That is, every point on the object is translated by the same amount.

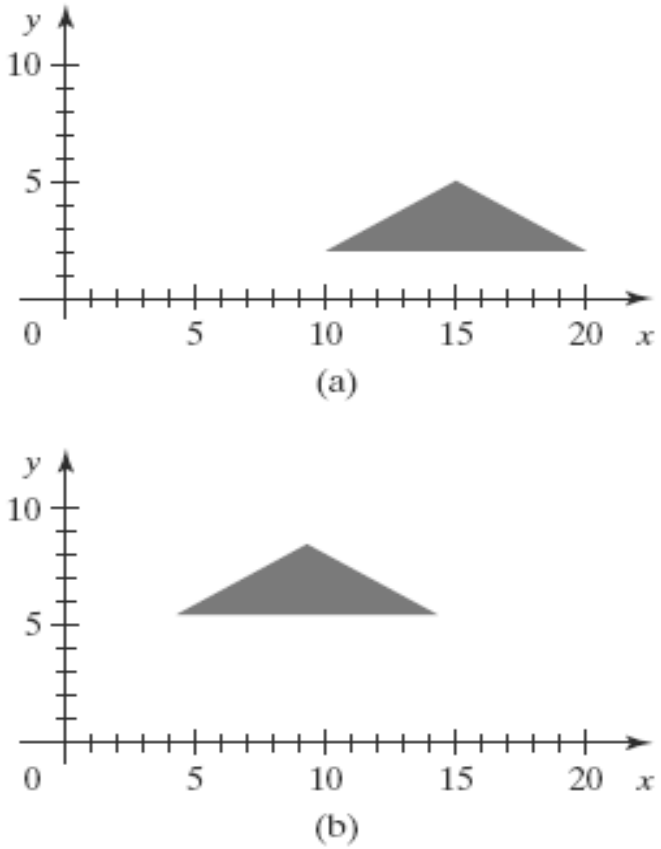


FIGURE 2
Moving a polygon from position (a) to position (b) with the translation vector (−5.50, 3.75).

- A straight-line segment is translated by applying Equation 3 to each of the two line endpoints and redrawing the line between the new endpoint positions. Similarly for the polygon as shown above.

```
class wcPt2D
{
        public:
        GLfloat x, y;
};
void translatePolygon (wcPt2D * verts, GLint nVerts, GLfloat tx, GLfloat ty)
{
        GLint k;
        for (k = 0; k < nVerts; k++)
        {
                verts [k].x = verts [k].x + tx;
                verts [k].y = verts [k].y + ty;
        }
        glBegin (GL_POLYGON);
        for (k = 0; k < nVerts; k++)
                glVertex2f (verts [k].x, verts [k].y);
        glEnd ( );
}
```

Two-Dimensional Rotation

- We generate a **rotation transformation of an object by specifying a rotation axis** and a **rotation angle.**
- **All points of the object are then transformed to new positions** by rotating the points through the specified angle about the rotation axis.
- A two-dimensional rotation of an object is obtained by **repositioning the object** along a circular path in the *xy plane*.
- **Parameters for the two-dimensional rotation** are the rotation angle θ *and* a position (xr, yr), *called the* ***rotation point (or pivot point), about which the*** object is to be rotated.

➢ A positive value for the angle ϑ *defines a* counterclockwise rotation about the pivot point, as in Figure 3, and a negative value rotates objects in the clockwise direction.

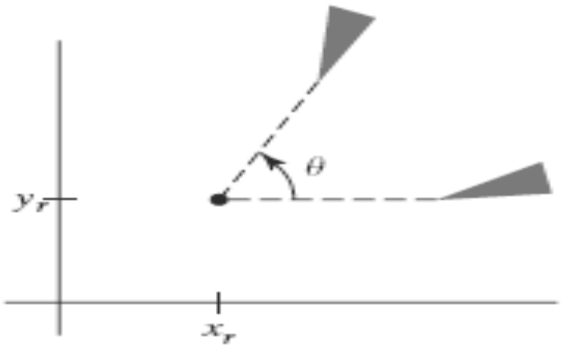


FIGURE 3
Rotation of an object through angle θ about the pivot point (x_r, y_r).

- First determine the transformation equations for rotation of a point position **P when the pivot point is** at the coordinate origin.
- The angular and coordinate relationships of the original and transformed point positions are shown in Figure 4.

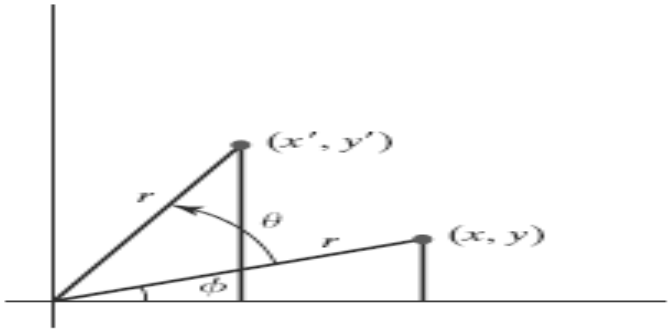


FIGURE 4
Rotation of a point from position (x, y) to position (x', y') through an angle θ relative to the coordinate origin. The original angular displacement of the point from the x axis is ϕ.

- In this figure, *r is* the constant distance of the point from the origin, angle *φ is the original angular* position of the point from the horizontal, and *θ is the rotation angle*.

- Using standard trigonometric identities, we can express the transformed coordinates in terms of angles θ *and* φ *as*

$$x' = r\cos(\phi + \theta) = r\cos\phi\cos\theta - r\sin\phi\sin\theta$$
$$y' = r\sin(\phi + \theta) = r\cos\phi\sin\theta + r\sin\phi\cos\theta \qquad (4)$$

- The original coordinates of the point in polar coordinates are

$$x = r\cos\phi, \qquad y = r\sin\phi \qquad (5)$$

- Substituting expressions 5 into 4, we obtain the transformation equations for rotating a point at position (*x, y)* *through an angle* θ *about the origin:*

$$x' = x\cos\theta - y\sin\theta$$
$$y' = x\sin\theta + y\cos\theta \qquad (6)$$

- With the column-vector representations 2 for coordinate positions, we can write the rotation equations in the matrix form

$$\mathbf{P}' = \mathbf{R} \cdot \mathbf{P} \tag{7}$$

- where the rotation matrix is

$$\mathbf{R} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{8}$$

- **Rotation of a point about an arbitrary pivot position** is illustrated in Figure 5.

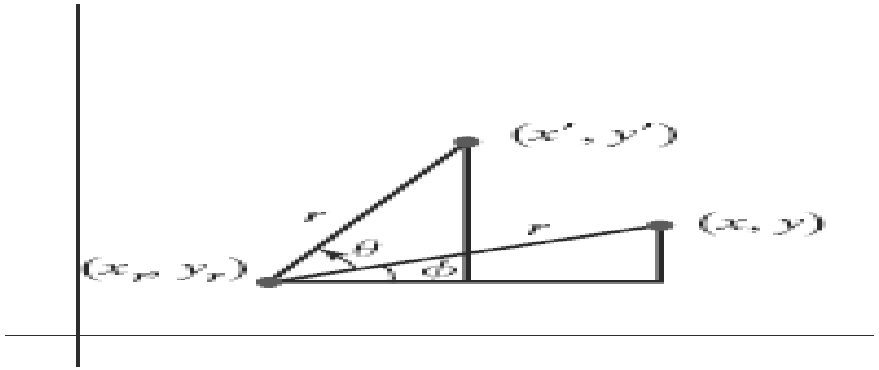


FIGURE 5
Rotating a point from position (x, y) to position (x', y') through an angle θ about rotation point (x_r, y_r).

- Using the trigonometric relationships indicated by the two right triangles in this figure, we can generalize Equations 6 to obtain the transformation equations for rotation of a point about any specified rotation position (x_r ,y_r):

$$x' = x_r + (x - x_r)\cos\theta - (y - y_r)\sin\theta$$
$$y' = y_r + (x - x_r)\sin\theta + (y - y_r)\cos\theta \qquad (9)$$

- As with translations, rotations are rigid-body transformations that **move objects without deformation.**
- Every point on an object is rotated through the same angle.
  - A straight-line segment is rotated by applying Equations 9 to each of the two line endpoints and redrawing the line between the new endpoint positions.
  - A polygon is rotated by displacing each vertex using the specified rotation angle and then regenerating the polygon using the new vertices.
- In the **following code example**, a **polygon is rotated** about a specified **world coordinate pivot point.**

```
class wcPt2D
{
    public:
        GLfloat x, y;
};
void rotatePolygon (wcPt2D * verts, GLint nVerts, wcPt2D pivPt,GLdouble theta)
{
    wcPt2D * vertsRot;
    GLint k;
    for (k = 0; k < nVerts; k++) {
     vertsRot [k].x = pivPt.x + (verts [k].x - pivPt.x) * cos (theta)- (verts [k].y - pivPt.y) *
    sin(theta);
      vertsRot [k].y = pivPt.y + (verts [k].x - pivPt.x) * sin (theta)+ (verts [k].y - pivPt.y) *
    cos (theta);
    }
    glBegin {GL_POLYGON};
    for (k = 0; k < nVerts; k++)
        glVertex2f (vertsRot [k].x, vertsRot [k].y);
    glEnd ( );
}
```

Two-Dimensional Scaling:

- To alter the size of an object, we apply a **scaling transformation.**
- **A simple two dimensional** scaling operation is performed by multiplying object positions *(x, y)* by **scaling factors *sx* and *sy* to produce the *transformed coordinates (x', y'):***

$$x' = x \cdot s_x, \qquad y' = y \cdot s_y \tag{10}$$

  - Scaling factor *sx scales an object in the x direction*, *while sy scales in the y direction.*
- The basic two-dimensional scaling equations 10 can also be written in the following matrix form:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \tag{11}$$

or

$$\mathbf{P}' = \mathbf{S} \cdot \mathbf{P} \tag{12}$$

where **S** is the 2×2 scaling matrix in Equation 11.

- Any positive values can be assigned to the scaling factors *sx and sy*.
  - ***Values* less than 1** reduce the size of objects; **values greater than 1** produce enlargements.
  - Specifying a **value of 1** for both *sx and sy* *leaves the size of objects unchanged*.
    - When *sx and sy are assigned the same value*, *a* ***uniform scaling is produced,*** which maintains relative object proportions.
    - Unequal values for *sx and sy result* in a **differential scaling that is often used in design applications, where pictures** are constructed from a few basic shapes that can be adjusted by scaling and positioning transformations (Figure 6).

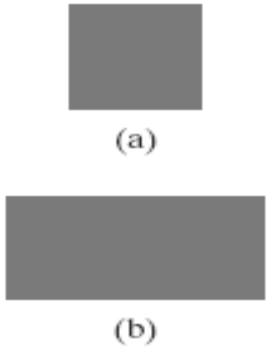


FIGURE 6
Turning a square (a) into a rectangle (b) with scaling factors $s_x = 2$ and $s_y = 1$.

- Objects transformed with Equation 11(scaling equation) are both scaled and repositioned.
- Scaling factors with absolute **values less than 1** *move objects closer to the coordinate origin*, while absolute **values greater than 1** *move coordinate positions farther from the origin.*
- Figure 7 illustrates scaling of a line by assigning the **value 0.5** to both *sx and sy in Equation 11. Both the line length and the* distance from the origin **are reduced by a factor of 1/2**.

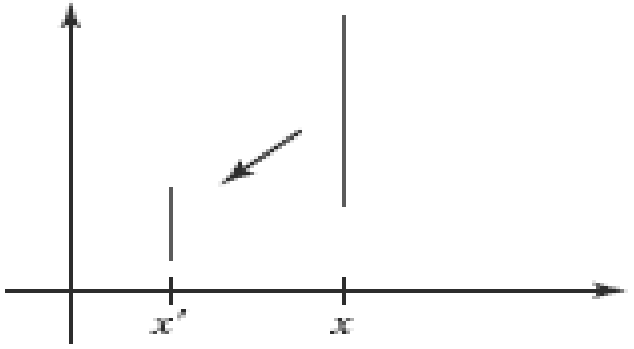


FIGURE 7
A line scaled with Equation 12 using $s_x = s_y = 0.5$ is reduced in size and moved closer to the coordinate origin.

- We can control the location of a scaled object by choosing a position, called the **fixed point**, ***that is to remain unchanged after the scaling transformation.***
- **Coordinates** for the fixed point, ***(x f , yf)***, *can be chosen anywhere.*
- Objects are now **resized by scaling the distances** between object points and the fixed point (Figure 8).
- For a coordinate position (*x*, *y*), *the scaled coordinates* ***(x', y')*** *are then calculated from the following relationships:*

$$x' - x_f = (x - x_f)s_x, \qquad y' - y_f = (y - y_f)s_y \tag{13}$$

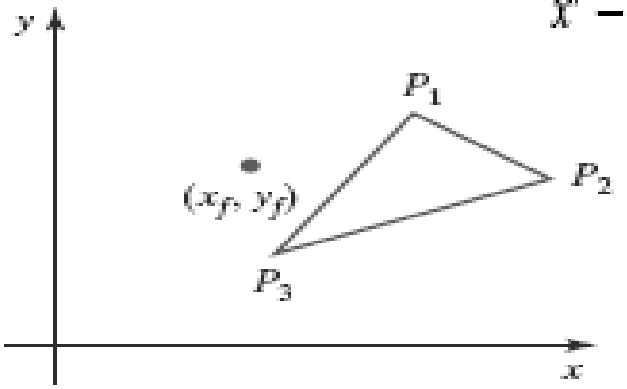


FIGURE 8
Scaling relative to a chosen fixed point (x_f, y_f). The distance from each polygon vertex to the fixed point is scaled by Equations 13.

- We can rewrite Equations 13 to separate the multiplicative and additive terms as

$$x' = x \cdot s_x + x_f(1 - s_x)$$
$$y' = y \cdot s_y + y_f(1 - s_y) \qquad (14)$$

  - where the additive terms $x_f\,(1 - sx)$ *and* $y_f\,(1 - sy)$ *are constants for all points in* the object.

- Polygons are scaled by applying transformations 14 to each vertex, then regenerating the polygon using the transformed vertices.
- For other objects, we apply the scaling transformation equations to the parameters defining the objects.

The following procedure illustrates an application of the scaling calculations for a polygon. Coordinates for the polygon vertices and for the fixed point are input parameters, along with the scaling factors. After the coordinate transformations, OpenGL routines are used to generate the scaled polygon.

```
class wcPt2D {
public:
GLfloat x, y;
};
void scalePolygon (wcPt2D * verts, GLint nVerts, wcPt2D fixedPt, GLfloat sx, GLfloat sy)
{
    wcPt2D vertsNew;
    GLint k;
    for (k = 0; k < nVerts; k++) {
        vertsNew [k].x = verts [k].x * sx + fixedPt.x * (1 - sx);
        vertsNew [k].y = verts [k].y * sy + fixedPt.y * (1 - sy);
    }
    glBegin {GL_POLYGON};
    for (k = 0; k < nVerts; k++)
        glVertex2f (vertsNew [k].x, vertsNew [k].y);
    glEnd ( );
}
```

Matrix Representations and Homogeneous Coordinates

- Many graphics applications involve sequences of geometric transformations.
  - An animation might require an *object to be translated and rotated at each increment of the motion.*
  - In design and picture construction applications, we perform translations, rotations, and scaling to *fit the picture components into their proper positions.*
- The **viewing transformations** involve sequences of translations and rotations to *take us from the original scene specification to the display on an output device.*
- Here, we consider how the matrix representations can be **reformulated** so that such transformation sequences can be ***processed efficiently.***

- The three basic two-dimensional transformations (translation, rotation, and scaling) can be expressed in the **general matrix form**

$$\mathbf{P}' = \mathbf{M}_1 \cdot \mathbf{P} + \mathbf{M}_2 \qquad (15)$$

  - Where P' is new coordinate and P is old coordinate.
  - M1(multiplicative) and M2(additive) matrix formats
  - For translation, **M1 is the identity matrix.**
  - For rotation or scaling, **M2 contains the translational terms associated with the** pivot point or scaling fixed point

- To produce a sequence of transformations with these equations, such as **scaling followed by rotation and then translation**, we could calculate the transformed coordinates one step at a time.

- First, coordinate positions are scaled,
  - then these scaled coordinates are rotated,
    - and finally, the rotated coordinates are translated.
- A more efficient approach, however, is to **combine the transformations** so that the ***final coordinate positions are obtained directly from the initial coordinates,*** without calculating intermediate coordinate values.
  - This is done by **reformulating Equation 15** to ***eliminate the matrix addition operation.***

Homogeneous Coordinates:

- **Multiplicative and translational terms** for a two-dimensional geometric transformation can be **combined into a single matrix** if we expand the *representations to 3 × 3 matrices*.
  - Then we can **use the third column of a transformation matrix** for the translation terms, and *all transformation equations can be expressed as matrix multiplications*.
    - we also need to expand the matrix representation for a two-dimensional coordinate position to a three-element column matrix.
  - A standard technique for accomplishing this is to expand each two dimensional coordinate-position representation (x, y) *to a* ***three-element representation*** (xh, yh, h), *called* ***homogeneous coordinates, where the homogeneous parameter*** h ***is a nonzero value such that***

$$x = \frac{x_h}{h}, \qquad y = \frac{y_h}{h} \tag{16}$$

- general two-dimensional homogeneous coordinate representation could also be written as ***(h·x, h·y, h)***
- A convenient choice is simply to **set *h = 1***.
  - *Each two-dimensional position is then* represented with homogeneous coordinates ***(x, y, 1).***
- **Expressing positions** in homogeneous coordinates allows us to represent all geometric transformation equations as **matrix multiplications**, which is the standard method used in graphics systems.
- **Two-dimensional coordinate positions** are represented with *three-element column vectors*, and **two-dimensional transformation operations** are expressed as *3 × 3 matrices*.

Two-Dimensional Translation Matrix

- Using a homogeneous-coordinate approach, we can represent the equations for a two-dimensional translation of a coordinate position using the following matrix multiplication:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \qquad (17)$$

- This translation operation can be written in the abbreviated form

$$\mathbf{P}' = \mathbf{T}(t_x, t_y) \cdot \mathbf{P} \qquad (18)$$

- with **T(*tx, ty) as the 3 × 3 translation matrix in Equation 17.***

Two-Dimensional Rotation Matrix

- Two-dimensional rotation transformation equations about the coordinate origin can be expressed in the matrix form

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (19)$$

or as

$$\mathbf{P}' = \mathbf{R}(\theta) \cdot \mathbf{P} \quad (20)$$

- The rotation transformation operator **R(*ϑ*) *is the 3 × 3 matrix in Equation 19***
- with rotation parameter *ϑ. We can also write this rotation matrix simply as* ***R.***
- ***By default 2d rotations are about the origin.***
- A **rotation about any other pivot point** must then be performed as a ***sequence of transformation operations.***

Two-Dimensional Scaling Matrix

- A scaling transformation relative to the coordinate origin can now be expressed as the matrix multiplication

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (21)$$

or

$$\mathbf{P}' = \mathbf{S}(s_x, s_y) \cdot \mathbf{P} \quad (22)$$

- The scaling operator **S(*sx, sy) is the 3 × 3 matrix in Equation 21 with parameters sx and sy*.** *And, in most cases, we can represent the scaling matrix simply as* ***S.***
- **Some libraries provide** a *scaling function that can generate only scaling with respect to the coordinate origin*, as in Equation 21.
- A scaling transformation relative to another reference position is handled as a **succession of transformation operations**.

Inverse Transformations

- For translation, we obtain the inverse matrix by negating the translation distances.
- Thus, if we have two-dimensional translation distances ***tx and ty***, *the inverse translation matrix is*

$$\mathbf{T}^{-1} = \begin{bmatrix} 1 & 0 & -t_x \\ 0 & 1 & -t_y \\ 0 & 0 & 1 \end{bmatrix} \qquad (23)$$

  - This produces a **translation in the opposite direction**, and the **product of a translation matrix and its inverse produces** the **identity matrix**.
  - Prove this!!!!

- An **inverse rotation** is accomplished by **replacing the rotation angle** by **its negative**.
- For example, a **two-dimensional rotation through an angle** θ *about the coordinate origin* has the **inverse transformation matrix**

$$\mathbf{R}^{-1} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{24}$$

- Negative values for rotation angles generate **rotations in a clockwise direction**, so the **identity matrix is produced** when any rotation matrix is multiplied by its inverse.
- Because only the **sine function is affected by the change in sign of the rotation angle**, the inverse matrix can also be obtained by interchanging rows and columns.
  - That is, we can calculate **the inverse of any rotation matrix R by** evaluating its transpose ($\mathbf{R}^{-1} = \mathbf{R}^{T}$)

- We form the **inverse matrix for any scaling transformation** by ***replacing the scaling parameters with their reciprocals***.
- For two-dimensional scaling with parameters ***sx and sy*** *applied relative to the coordinate origin, the inverse transformation* matrix is

$$\mathbf{S}^{-1} = \begin{bmatrix} \frac{1}{s_x} & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad (25)$$

➢ The **inverse matrix generates an opposite scaling transformation**, so the multiplication of any scaling matrix with its inverse produces **the identity matrix**.

Two-Dimensional Composite Transformations

- If a **transformation of the plane T1** is followed by a **second plane transformation T2**, then the **result itself may be represented by a single transformation T** which is the **composition of T1 and T2** taken in that order.
    - This is written as **T = T1·T2**
- **Composite transformation** can be achieved by **concatenation** of **transformation matrices** to obtain a **combined transformation matrix**.
- A combined matrix − **[T][X] = [X] [T1] [T2] [T3] [T4] …. [Tn]**
    - Where [Ti] is any combination of
        - » **Translation**
        - » **Scaling**
        - » **Shearing**
        - » **Rotation**
        - » **Reflection**

- The **change in the order of transformation** would **lead to different results**, as in **general matrix multiplication is not cumulative**, that is
  - $[A] . [B] \neq [B] . [A]$ and the **order of multiplication**.
- The **basic purpose of composing transformations** is to **gain efficiency** by ***applying a single composed transformation to a point***, rather than **applying a series of transformation, one after another**.

Composite Two-Dimensional Translations

- If two successive translation vectors ***(t1x, t1y)*** *and* ***(t2x, t2y)*** *are applied to a two dimensional* coordinate position **P, the final transformed location P' is calculated** as

$$\begin{aligned} \mathbf{P}' &= \mathbf{T}(t_{2x}, t_{2y}) \cdot \{\mathbf{T}(t_{1x}, t_{1y}) \cdot \mathbf{P}\} \\ &= \{\mathbf{T}(t_{2x}, t_{2y}) \cdot \mathbf{T}(t_{1x}, t_{1y})\} \cdot \mathbf{P} \end{aligned} \tag{27}$$

- where **P and P' are represented as three-element, homogeneous-coordinate column** vectors.
- We can verify this result by calculating the matrix product for the two associative groupings.
- The composite transformation matrix for this sequence of translations is

$$\begin{bmatrix} 1 & 0 & t_{2x} \\ 0 & 1 & t_{2y} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & t_{1x} \\ 0 & 1 & t_{1y} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_{1x} + t_{2x} \\ 0 & 1 & t_{1y} + t_{2y} \\ 0 & 0 & 1 \end{bmatrix} \tag{28}$$

or

$$\mathbf{T}(t_{2x}, t_{2y}) \cdot \mathbf{T}(t_{1x}, t_{1y}) = \mathbf{T}(t_{1x} + t_{2x}, t_{1y} + t_{2y}) \tag{29}$$

which demonstrates that two successive translations are additive.

Composite Two-Dimensional Rotations

- Two successive rotations applied to a point **P produce the transformed position**

$$\begin{aligned} \mathbf{P}' &= \mathbf{R}(\theta_2) \cdot \{\mathbf{R}(\theta_1) \cdot \mathbf{P}\} \\ &= \{\mathbf{R}(\theta_2) \cdot \mathbf{R}(\theta_1)\} \cdot \mathbf{P} \end{aligned} \tag{30}$$

- By **multiplying the two rotation matrices**, we can verify that two successive rotations are additive:

$$\mathbf{R}(\theta_2) \cdot \mathbf{R}(\theta_1) = \mathbf{R}(\theta_1 + \theta_2) \tag{31}$$

  - so that the final rotated coordinates of a point can be calculated with the composite rotation matrix as

$$\mathbf{P}' = \mathbf{R}(\theta_1 + \theta_2) \cdot \mathbf{P} \tag{32}$$

Composite Two-Dimensional Scaling's

- Concatenating transformation matrices for **two successive scaling operations** in two dimensions produces the following composite scaling matrix:

$$\begin{bmatrix} s_{2x} & 0 & 0 \\ 0 & s_{2y} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_{1x} & 0 & 0 \\ 0 & s_{1y} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_{1x} \cdot s_{2x} & 0 & 0 \\ 0 & s_{1y} \cdot s_{2y} & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (33)$$

or

$$\mathbf{S}(s_{2x}, s_{2y}) \cdot \mathbf{S}(s_{1x}, s_{1y}) = \mathbf{S}(s_{1x} \cdot s_{2x}, \quad s_{1y} \cdot s_{2y}) \quad (34)$$

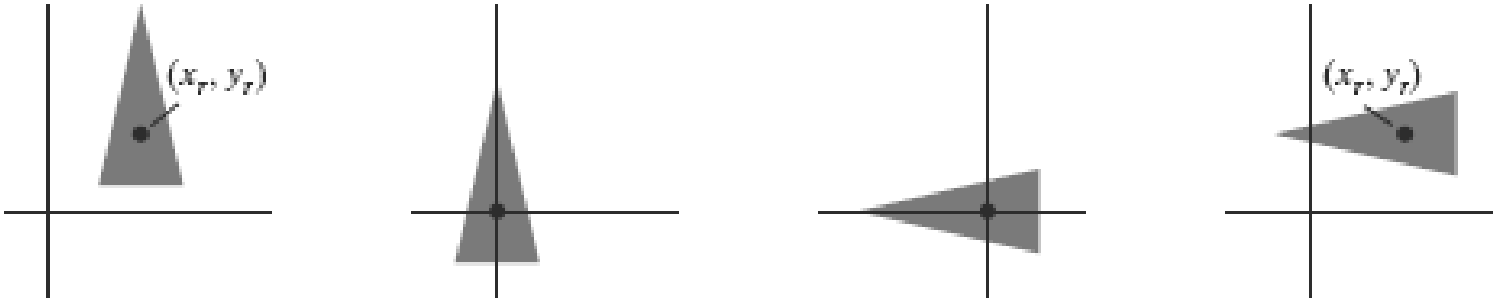
- The resulting matrix in this case indicates that **successive scaling operations** are multiplicative.
  - That is, if we were to **triple the size of an object twice in succession**, the **final size** would be nine times that of the original.

General Two-Dimensional Pivot-Point Rotation

- By default rotation in graphics is about origin.
- we can generate a two-dimensional rotation about any **other pivot point** ***(xr, yr)*** *by performing the following sequence of translate-rotate-translate* operations:

1. **Translate the object so that the pivot-point position is moved to the coordinate** origin.
2. **Rotate the object about the coordinate origin.**
3. **Translate the object so that the pivot point is returned to its original** position.

- This transformation sequence is illustrated in Figure 9.



(a) Original Position of Object and Pivot Point

(b) Translation of Object so that Pivot Point (x_r, y_r) is at Origin

(c) Rotation about Origin

(d) Translation of Object so that the Pivot Point is Returned to Position (x_r, y_r)

- The composite transformation matrix for this sequence is obtained with the concatenation

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

$$= \begin{bmatrix} \cos\theta & -\sin\theta & x_r(1-\cos\theta) + y_r \sin\theta \\ \sin\theta & \cos\theta & y_r(1-\cos\theta) - x_r \sin\theta \\ 0 & 0 & 1 \end{bmatrix} \quad (35)$$

which can be expressed in the form

$$\mathbf{T}(x_r, y_r) \cdot \mathbf{R}(\theta) \cdot \mathbf{T}(-x_r, -y_r) = \mathbf{R}(x_r, y_r, \theta) \quad (36)$$

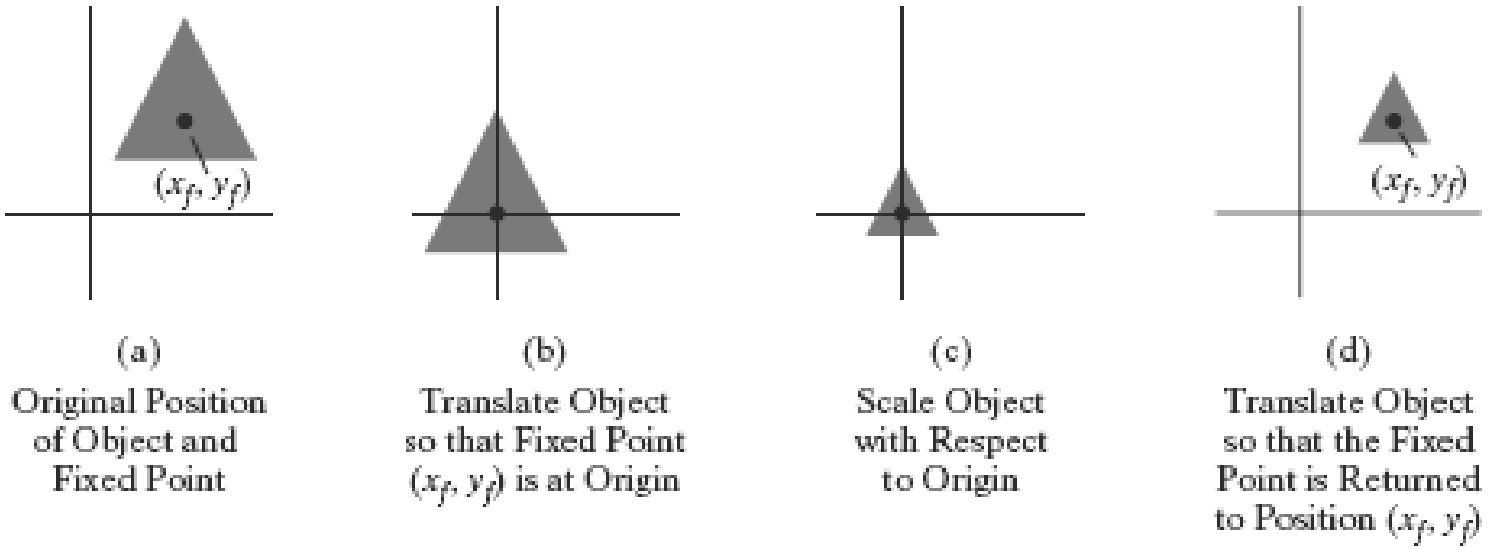
where $\mathbf{T}(-xr, -yr) = T^{-1}(xr, yr)$.

General Two-Dimensional Fixed-Point Scaling

- To produce a two-dimensional scaling with respect to a selected fixed position (x_f , y_f), *we have a function* that can scale relative to the coordinate origin only. This sequence is

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

- Figure 10 illustrates a transformation sequence



- Concatenating the matrices for these three operations produces the required scaling matrix:

$$\begin{bmatrix} 1 & 0 & x_f \\ 0 & 1 & y_f \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_f \\ 0 & 1 & -y_f \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & x_f(1-s_x) \\ 0 & s_y & y_f(1-s_y) \\ 0 & 0 & 1 \end{bmatrix} \quad (37)$$

or

$$\mathbf{T}(x_f, y_f) \cdot \mathbf{S}(s_x, s_y) \cdot \mathbf{T}(-x_f, -y_f) = \mathbf{S}(x_f, y_f, s_x, s_y) \quad (38)$$

- This transformation is generated automatically in systems that provide a scale function that accepts coordinates for the fixed point.

General Two-Dimensional Scaling Directions

- Parameters ***sx*** *and* ***sy*** *scale objects* ***along the x and y directions***.
- ***To scale*** **an object in other directions:** rotate the object to align the desired scaling directions with the **coordinate axes** before applying the scaling transformation.
- Suppose we want to **apply scaling factors** with values specified by parameters *s1 and s2 in the directions shown in Figure 11*.

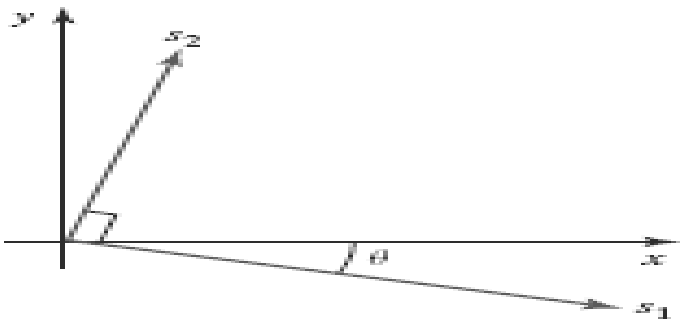


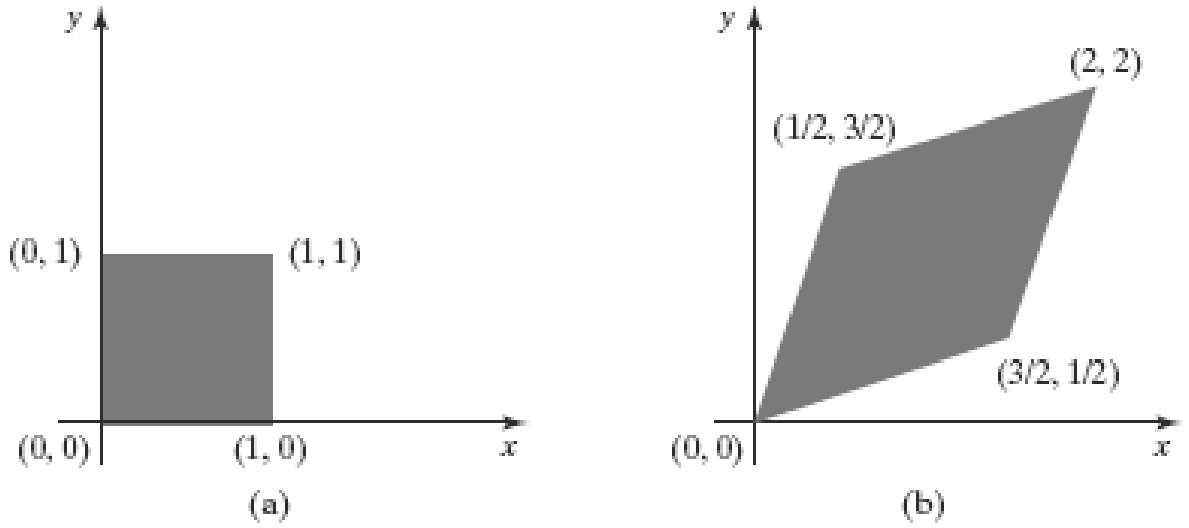
FIGURE 11
Scaling parameters s_1 and s_2 along orthogonal directions defined by the angular displacement θ.

- To accomplish the scaling without changing the orientation of the object,
    1. First perform a rotation so that the directions for *s1 and s2 coincide with the x and y axes*, *respectively. Then*
    2. *the* scaling transformation **S(*s1, s2) is applied,** *followed by*
    3. *an* ***opposite rotation to*** return points to their original orientations.
- The composite matrix resulting from the product of these three transformations is

$$\mathbf{R}^{-1}(\theta) \cdot \mathbf{S}(s_1, s_2) \cdot \mathbf{R}(\theta) = \begin{bmatrix} s_1 \cos^2\theta + s_2 \sin^2\theta & (s_2 - s_1)\cos\theta \sin\theta & 0 \\ (s_2 - s_1)\cos\theta \sin\theta & s_1 \sin^2\theta + s_2 \cos^2\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (39)$$

- As an example of this scaling transformation, we **turn a unit square** into a parallelogram (Figure 12) by **stretching it along the diagonal from (0, 0) to (1, 1).**

  1. We **first rotate** the **diagonal onto the *y axis using θ = 45*°**, *then we*
  2. ***double its length*** with the scaling values *s1 = 1 and s2 = 2, and then*
  3. *we **rotate again** to return the* diagonal to its original orientation.



Matrix Concatenation Properties

Multiplication of matrices is associative:

- For any three matrices,**M1,M2, andM3,** the matrix product **M3 · M2 · M1 can be performed by first multiplying M3 and M2 or by first multiplying M2 and M1:**

$$M_3 \cdot M_2 \cdot M_1 = (M_3 \cdot M_2) \cdot M_1 = M_3 \cdot (M_2 \cdot M_1) \quad (40)$$

- Therefore, depending upon the order in which the transformations are specified, we can construct a composite matrix either by multiplying from left to right (pre-multiplying) or by multiplying from right to left (post-multiplying).
  - **Depends on how graphics package is designed.** (either M1 is called first or M3 or vice versa)

- **Transformation** products, on the other hand, **may not be commutative**.
  - The matrix product**M2 ·M1 is not equal to M1 ·M2**
- 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

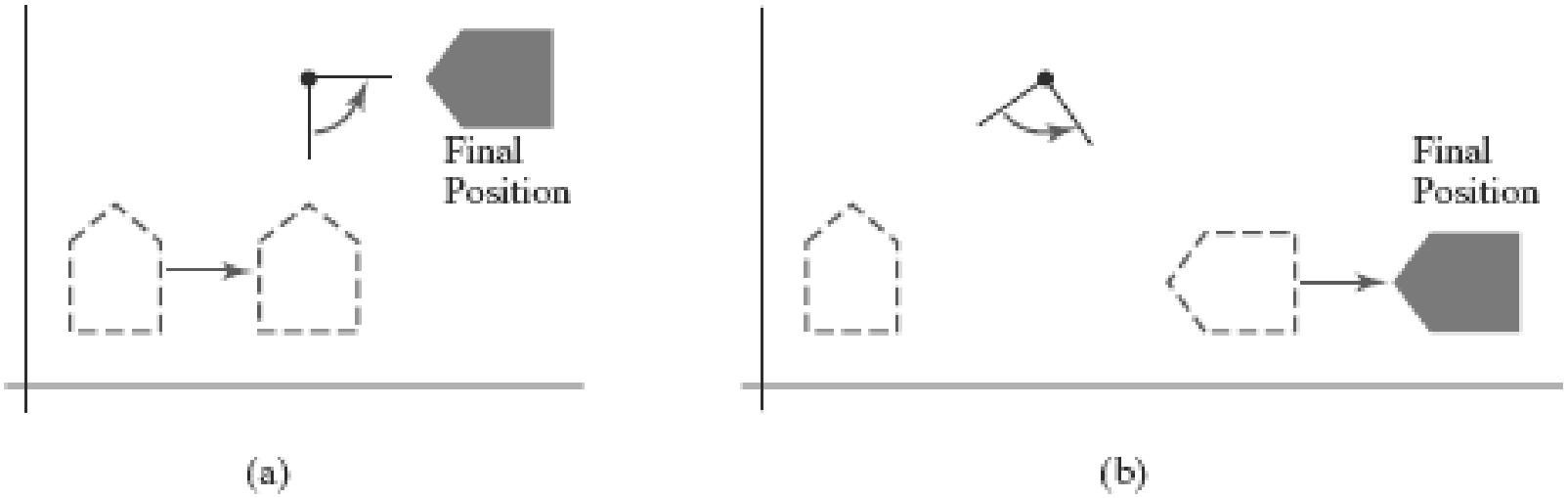


FIGURE 13

Reversing the order in which a sequence of transformations is performed may affect the transformed position of an object. In (a), an object is first translated in the x direction, then rotated counterclockwise through an angle of 45°. In (b), the object is first rotated 45° counterclockwise, then translated in the x direction.

Other Two-Dimensional Transformations

Reflection:

- A transformation that produces a **mirror image of an object** is called a **reflection.**
- For a two-dimensional reflection, this image is generated relative to an **axis of reflection by rotating the object 180° about the reflection axis.**
- Reflection about the line ***y = 0 (the x axis)*** *is accomplished with the transformation* matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad (52)$$

- This **transformation retains *x values***, *but "flips" the y values of coordinate positions*.
- The resulting orientation of an object after it has been reflected about the *x* axis is shown in Figure

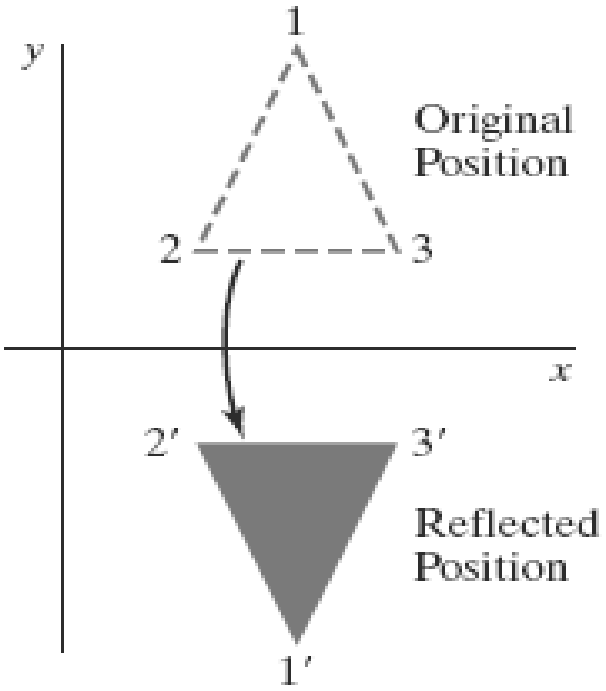
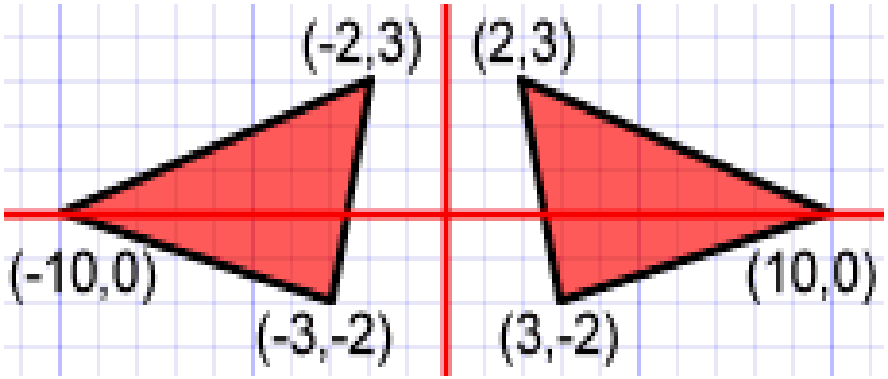


FIGURE 16
Reflection of an object about the *x* axis.

- A reflection about the **line *x* = 0 (the *y* axis)** *flips x coordinates while keeping* ***y coordinates the same****. The matrix for this transformation is*

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad (53)$$

- Figure 17 illustrates the change in position of an object that has been reflected about the line $x = 0$.



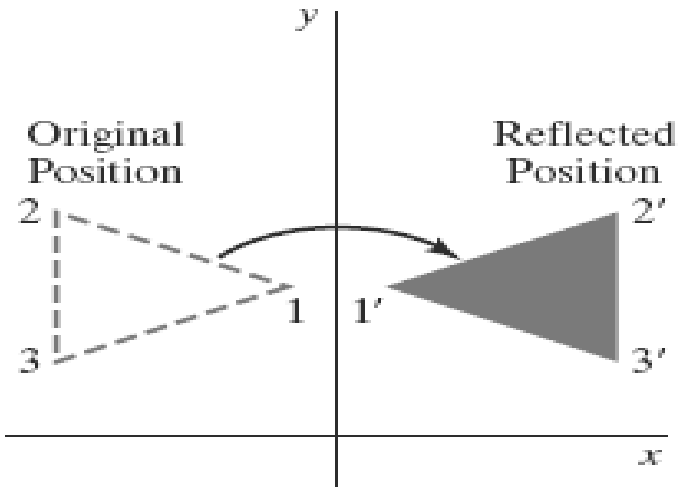


FIGURE 17
Reflection of an object about the *y* axis.

- We flip both the ***x and y coordinates*** *of a point by* ***reflecting relative to an axis*** that is perpendicular to the *xy plane* *and that passes through the coordinate origin.*
- This reflection is sometimes referred to as a **reflection relative to the coordinate origin**, and it is equivalent to reflecting with respect to both coordinate axes.
- The matrix representation for this reflection is

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad (54)$$

- The reflection matrix 54 is the same as the rotation matrix **R(*ϑ*) *with ϑ = 180*** $^\circ$
- We are simply rotating the object in the ***xy plane*** *half a revolution about the origin.*

- An example of **reflection about the origin** is shown in Figure 18.

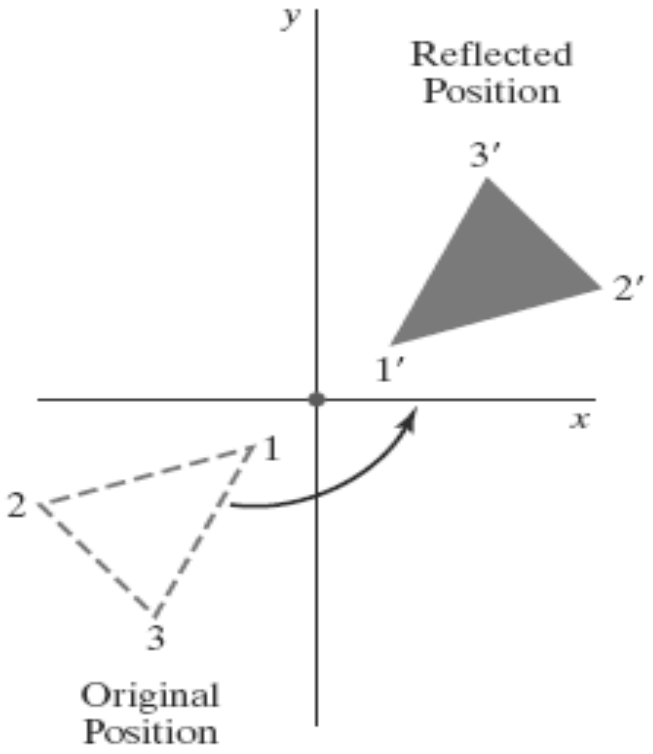


FIGURE 18
Reflection of an object relative to the coordinate origin. This transformation can be accomplished with a rotation in the *xy* plane about the coordinate origin.

- Reflection 54 can be generalized to **any reflection point in the *xy plane*** (Figure 19).
- This reflection is the same as a 180◦ rotation in the *xy plane about* the reflection point

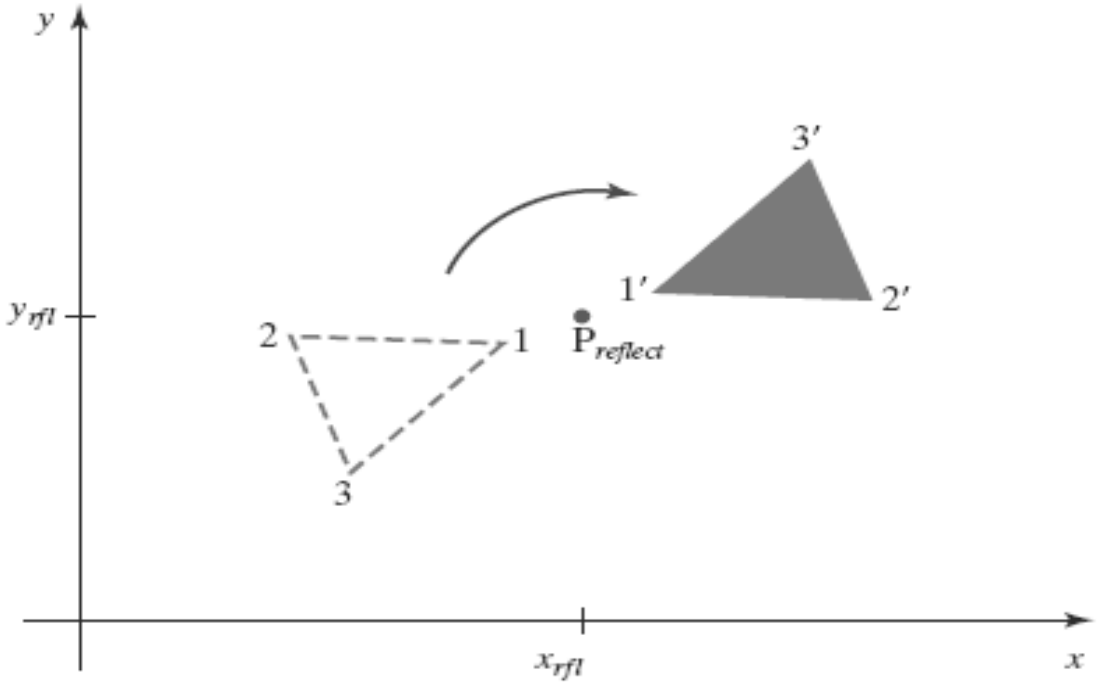


FIGURE 19
Reflection of an object relative to an axis perpendicular to the *xy* plane and passing through point $P_{reflect}$.

- If we choose the **reflection axis** as the **diagonal line** $y = x$ *(Figure 20), the* reflection matrix is

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{55}$$

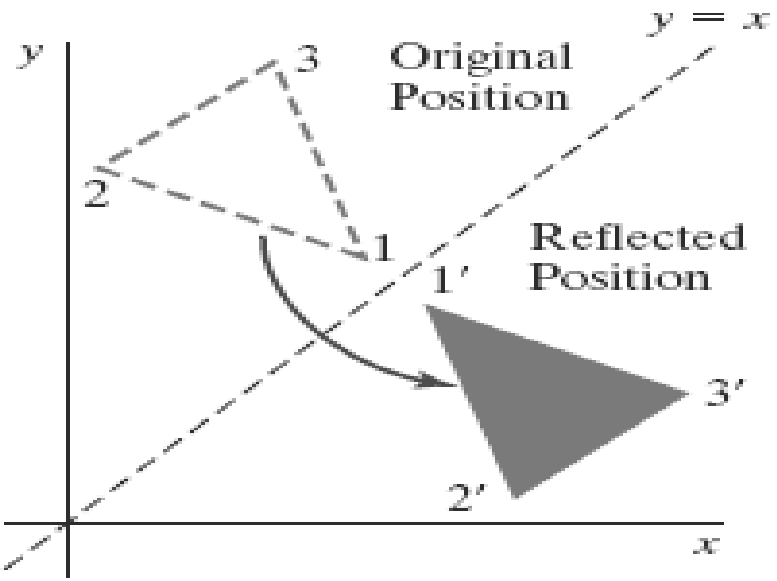


FIGURE 20
Reflection of an object with respect to the line $y = x$.

- Generally **Reflection about any line in Computer Graphics is represented by**, $y = mx + b$
- We can **derive reflection matrix** by *concatenating a sequence of rotation and coordinate axis* reflection matrices.
- One possible sequence is shown in Figure 21.

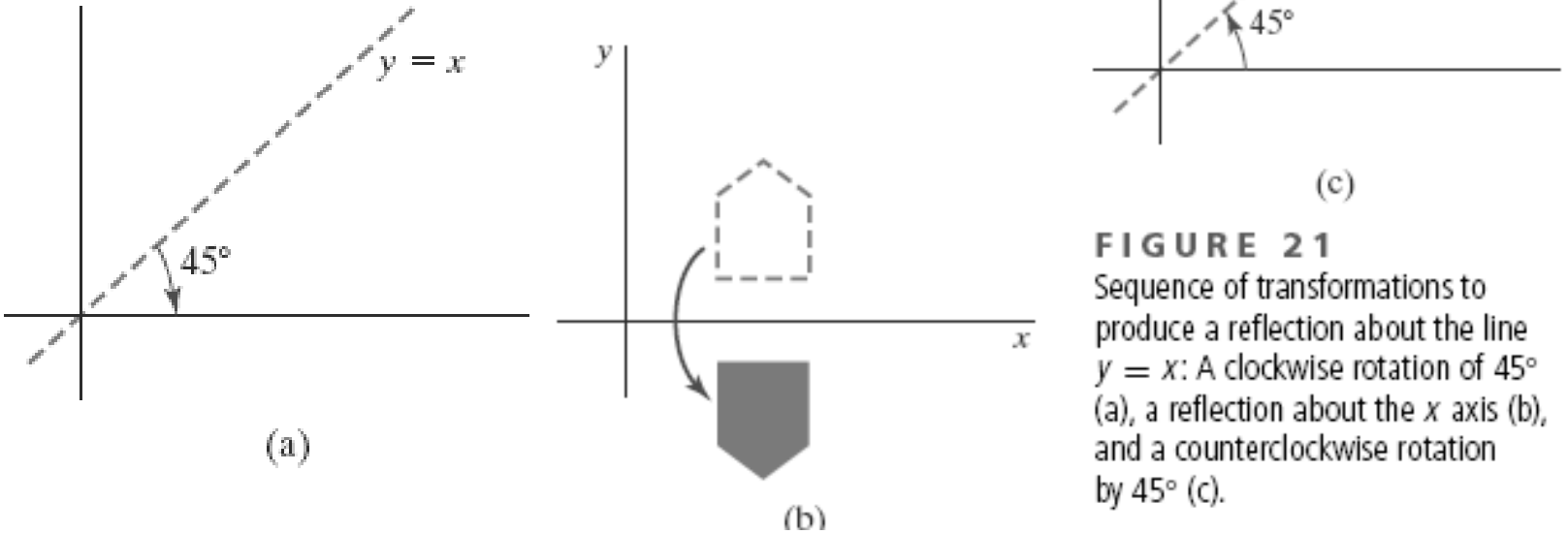


FIGURE 21
Sequence of transformations to produce a reflection about the line $y = x$: A clockwise rotation of 45° (a), a reflection about the x axis (b), and a counterclockwise rotation by 45° (c).

- **first** perform a clockwise rotation with respect to the origin through a 45◦ angle, which rotates the line $y = x$ *onto the x axis*.
- *Next, we* ***perform a reflection with*** **respect to the *x axis***.
- *The* ***final step*** *is to rotate the line* $y = x$ *back to its original* position with a **counterclockwise rotation through 45◦.**

- To obtain **a transformation matrix for reflection about the diagonal $y = -x$**, we could concatenate matrices for the transformation sequence:
  - (1) clockwise rotation by 45◦,
  - (2) reflection about the *y axis, and*
  - *(3) counterclockwise rotation* by 45◦. The resulting transformation matrix is

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{56}$$

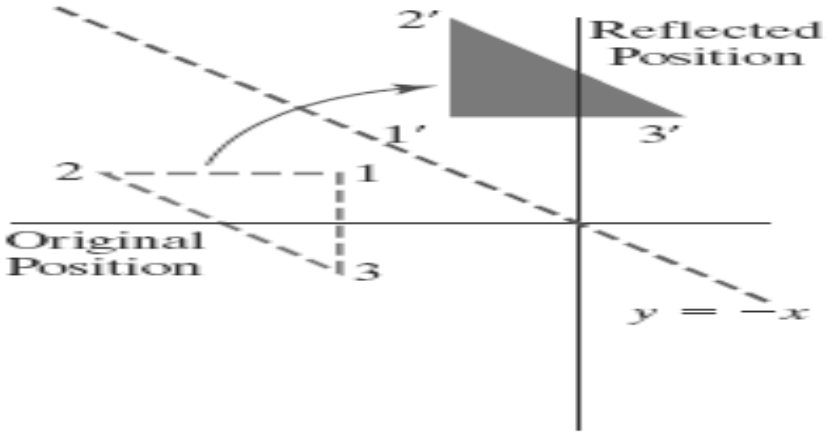


FIGURE 22
Reflection with respect to the line $y = -x$.

- **Reflections about any line** $y = mx + b$ *in the* ***xy plane*** *can be accomplished* with a **combination of translate-rotate-reflect transformations**.
  - In general, we **first translate** the line so that it passes through the origin.
  - Then we can rotate the line onto one of the coordinate axes and reflect about that axis.
  - Finally, we restore the line to its original position with the inverse rotation and translation transformations.

Shear:

- A transformation that **distorts or slants the shape of an object** is called a **shear.**
- **Two common shearing** transformations are: those that shift coordinate *x values* and *those that shift y* values.
  - An *x-direction shear relative to the x axis is produced with the transformation* matrix

$$\begin{bmatrix} 1 & sh_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{57}$$

which transforms coordinate positions as

$$x' = x + sh_x \cdot y, \qquad y' = y \tag{58}$$

➢ However; in both the cases **only one coordinate changes its coordinates** and other preserves its values. Shearing is also termed as **Skewing**.

- Any **real number can be assigned** to the **shear parameter** s_{hx}
- Setting **parameter** s_{hx} *to the* ***value 2,*** *for example,* changes the square in Figure 23 into a parallelogram.
- **Negative values for** s_{hx} *shift coordinate positions to the left.*

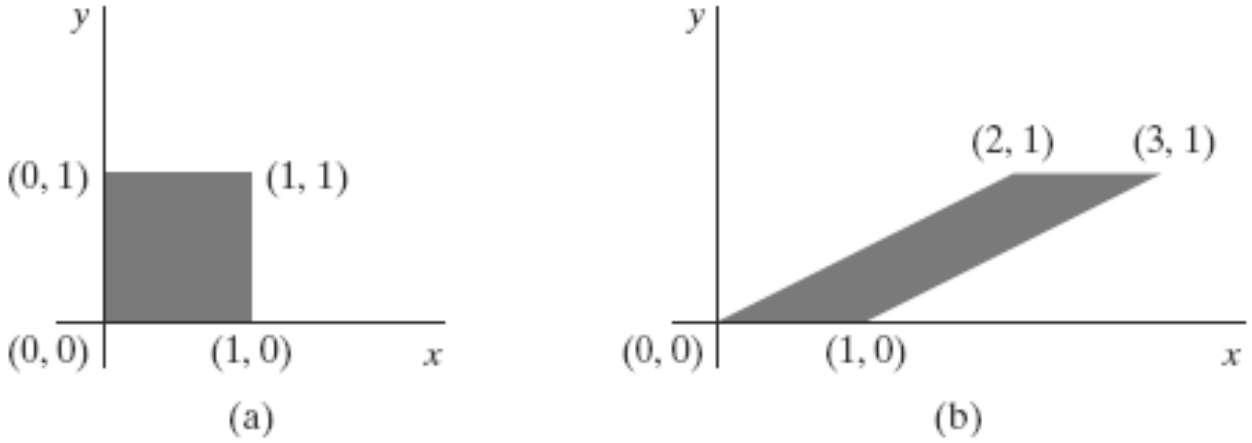


FIGURE 23
A unit square (a) is converted to a parallelogram (b) using the x-direction shear matrix 57 with $sh_x = 2$.

- We can generate *x-direction shears relative to other reference lines* *with*

$$\begin{bmatrix} 1 & sh_x & -sh_x \cdot y_{ref} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{59}$$

Now, coordinate positions are transformed as

$$x' = x + sh_x(y - y_{ref}), \qquad y' = y \tag{60}$$

- Figure 24 for a shear parameter value of 1/2 relative to the line ***yref* = −1**

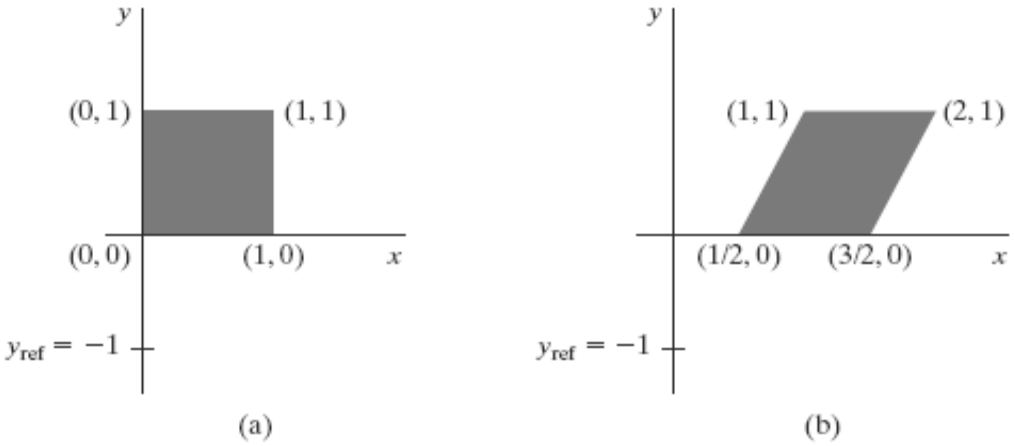


FIGURE 24
A unit square (a) is transformed to a shifted parallelogram (b) with $sh_x = 0.5$ and $y_{ref} = -1$ in the shear matrix 59.

- A ***y-direction shear*** *relative to the line* $x = x_{ref}$ *is generated with the transformation* matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ sh_y & 1 & -sh_y \cdot x_{\text{ref}} \\ 0 & 0 & 1 \end{bmatrix} \tag{61}$$

which generates the transformed coordinate values

$$x' = x, \qquad y' = y + sh_y(x - x_{\text{ref}}) \tag{62}$$

- Figure 25 illustrates the conversion of a square into a parallelogram with *shy = 0.5 and xref = −1*

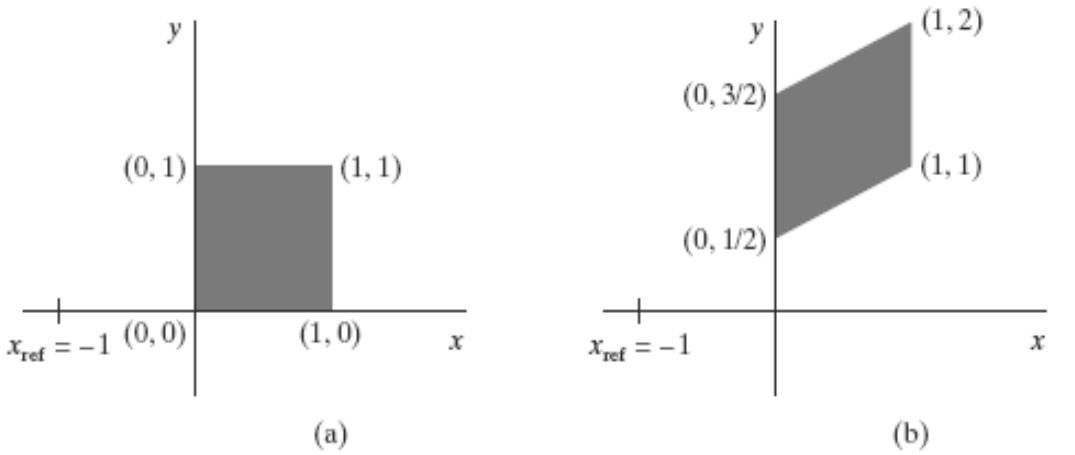
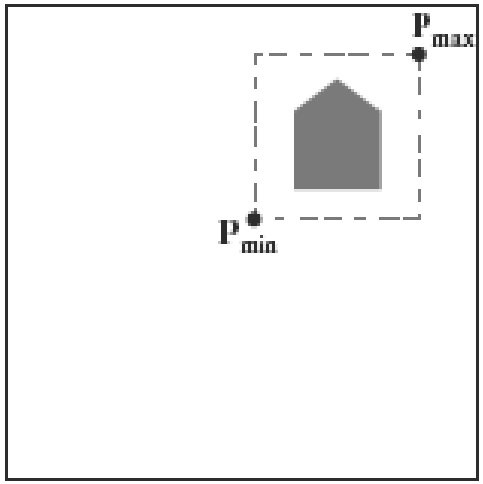


FIGURE 25
A unit square (a) is turned into a shifted parallelogram (b) with parameter values $sh_y = 0.5$ and $x_{\text{ref}} = -1$ in the y-direction shearing transformation 61.

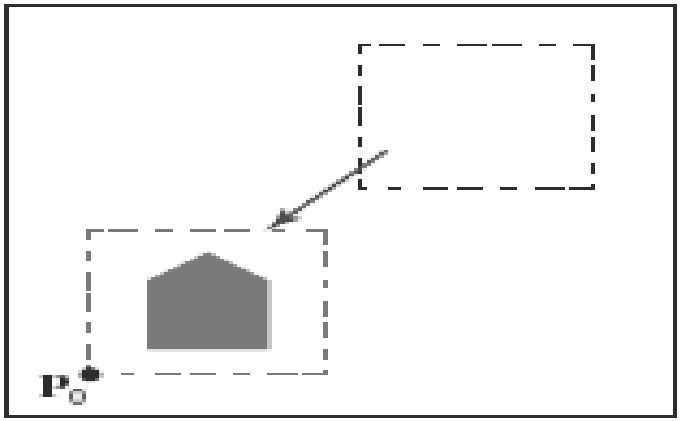
Raster Methods for Geometric Transformations

- The characteristics of raster systems suggest an alternate method for performing certain two-dimensional transformations.
- Raster systems store picture information as color patterns in the frame buffer.
    - Therefore, some simple object transformations can be carried out rapidly by manipulating an array of pixel values using few arithmetic operations.
- Functions that **manipulate rectangular pixel arrays** are called raster operations *and* **moving a block of pixel values** from one position to another is termed a ***block transfer, a bitblt, or a pixblt***.

- Figure 26 illustrates a two-dimensional translation implemented as a block transfer of a refresh-buffer area.
- **All bit settings in the rectangular area** shown are copied as a block into another part of the frame buffer. [We can erase the pattern at the original location by assigning the background color to all pixels within that block]



(a)



(b)

FIGURE 26
Translating an object from screen position (a) to the destination position shown in (b) by moving a rectangular block of pixel values. Coordinate positions P_{min} and P_{max} specify the limits of the rectangular block to be moved, and P_0 is the destination reference position.

- Rotations in 90-degree increments are accomplished easily by rearranging the elements of a pixel array.
- We can rotate a two-dimensional object or pattern 90◦ counterclockwise by **reversing the pixel values in each row of the array**, then interchanging rows and columns.
- A **180◦ rotation** is obtained by **reversing the order of the elements in each row of the array**, then **reversing the order of the rows**. Figure 27 demonstrates the array manipulations that can be used to rotate a pixel block by 90◦ and by 180

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

(a)

$$\begin{bmatrix} 3 & 6 & 9 & 12 \\ 2 & 5 & 8 & 11 \\ 1 & 4 & 7 & 10 \end{bmatrix}$$

(b)

$$\begin{bmatrix} 12 & 11 & 10 \\ 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

(c)

FIGURE 27
Rotating an array of pixel values. The original array is shown in (a), the positions of the array elements after a 90° counterclockwise rotation are shown in (b), and the positions of the array elements after a 180° rotation are shown in (c).

- We can use similar methods to **scale a block of pixels**.
  - Pixel areas in the original block are scaled, using specified values for ***sx and sy***, *and* ***then mapped onto a set of*** **destination pixels**.
  - The color of each destination pixel is then assigned according to its area of overlap with the scaled pixel areas (Figure 29).
- An **object can be reflected** using raster transformations that reverse row or column values in a pixel block, combined with translations.
- **Shears** are produced with shifts in the positions of array values along rows or columns.

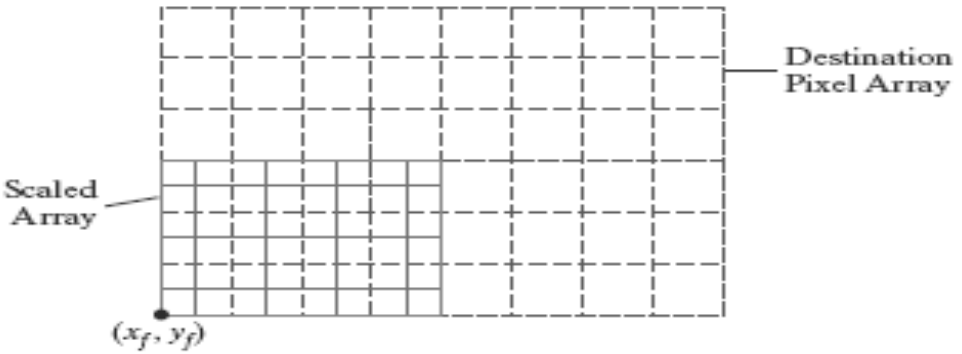


FIGURE 29
Mapping destination pixel areas onto a scaled array of pixel values. Scaling factors $s_x = s_y = 0.5$ are applied relative to fixed point (x_f, y_f).

OpenGL Raster Transformations

- A **translation of a rectangular array of pixel-color values from one buffer area to another** can be accomplished in OpenGL as the following copy operation:

```
glCopyPixels (xmin, ymin, width, height, GL_COLOR);
```

  - The first four parameters in this function give the location and dimensions of the pixel block; and the OpenGL symbolic constant **GL COLOR specifies that it is color** values are to be copied.
  - This array of pixels is to be copied to a rectangular area of a **refresh buffer** whose lower-left corner is at the location specified by the current raster position.

  - Pixel-color values are copied as either RGBA values or **color-table indices**, depending on the current setting for the color mode.
  - **Both** the region to be copied (the source) and the destination area **should lie** completely within the bounds of the screen coordinates

- This translation can be carried out on any of the OpenGL buffers used for refreshing, or even between different buffers.
- A source buffer for the **glCopyPixels function** is chosen with the **glReadBuffer** routine, and a destination buffer is selected with the **glDrawBuffer** routine.

- We can rotate a block of pixel-color values in 90-degree increments by first saving the block in an array, ***then rearranging the elements of the array and placing it back in the refresh buffer.*** A block of RGB color values in a buffer can be saved in an array with the function

```
glReadPixels (xmin, ymin, width, height, GL_RGB,
                    GL_UNSIGNED_BYTE, colorArray);
```

If color-table indices are stored at the pixel positions, we replace the constant **GL_RGB with GL _COLOR _INDEX.**

- we put the rotated array back in the buffer with

```
glDrawPixels (width, height, GL_RGB, GL_UNSIGNED_BYTE,
                    colorArray);
```

- A two-dimensional scaling transformation can be performed as a raster operation in OpenGL by specifying scaling factors and then invoking either **glCopyPixels or glDrawPixels.**
- **For the raster operations, we set the scaling** factors with

```
glPixelZoom (sx, sy);
```

  - where parameters **sx and sy can be assigned any nonzero floating-point values.**
  - Positive values greater than 1.0 increase the size of an element in the source array, and positive values less than 1.0 decrease element size.
  - A negative value for **sx or sy, or both, produces a reflection and scales the array elements.**

Transformations between Two-Dimensional Coordinate Systems

- Computer-graphics applications involve **coordinate transformations from one reference frame to another** during various stages of scene processing.
- The **viewing routines** transform object descriptions from world coordinates to device coordinates.
- For modeling and design applications, individual objects are typically defined in their own **local Cartesian references**.
  - These local-coordinate descriptions must then be transformed into positions and orientations within the overall scene coordinate system.

- Also, scenes are sometimes described in non-Cartesian reference frames that take advantage of object symmetries. **Coordinate descriptions in these systems must be converted to Cartesian world coordinates for processing.**

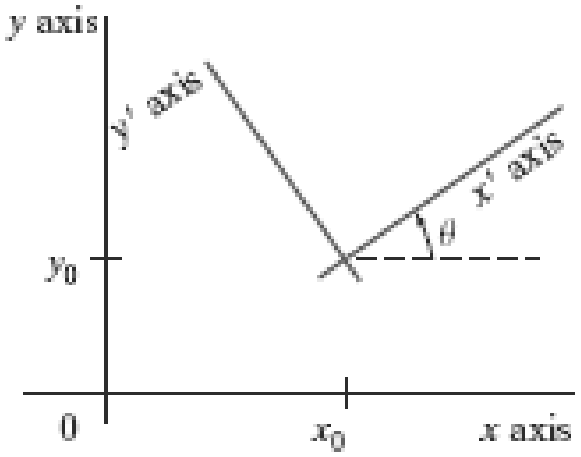


FIGURE 30
A Cartesian $x'y'$ system positioned at (x_0, y_0) with orientation θ in an xy Cartesian system.

- Figure 30 shows a Cartesian *x' y' system specified with coordinate origin (x0, y0) and orientation angle ϑ in a Cartesian xy reference frame.*
- *To transform* object descriptions from *xy coordinates to x'y'coordinates, we set up a transformation* that superimposes the *x'y' axes onto the xy axes.*

- This is done in two steps:

  1. **Translate so that the origin (*x0, y0) of the x'y'system is moved to the origin*** (0, 0) of the *xy system*
  2. **Rotate the *x' axis onto the x axis.***

- Translation of the coordinate origin is accomplished with the matrix transformation

$$\mathbf{T}(-x_0, -y_0) = \begin{bmatrix} 1 & 0 & -x_0 \\ 0 & 1 & -y_0 \\ 0 & 0 & 1 \end{bmatrix} \qquad (63)$$

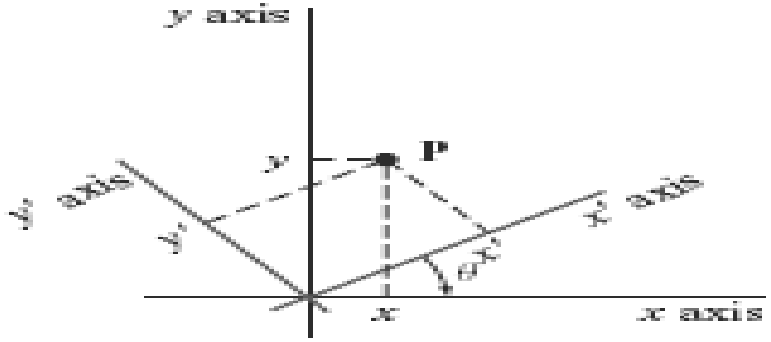


FIGURE 3 1
Position of the reference frames shown in Figure 30 after translating the origin of the $x'y'$ system to the coordinate origin of the xy system.

- To get the axes of the two systems into coincidence, we then perform the clockwise rotation

$$\mathbf{R}(-\theta) = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (64)$$

- Concatenating these two transformation matrices gives us the complete composite matrix for transforming object descriptions from the ***xysystem*** *to the* ***x'y'*** *system:*

$$\mathbf{M}_{xy,x'y'} = \mathbf{R}(-\theta) \cdot \mathbf{T}(-x_0, -y_0) \quad (65)$$

OpenGL Functions for Two-Dimensional Geometric Transformations

- In the core library of OpenGL, a separate function is available for each of the basic geometric transformations.
- Because OpenGL is designed as a three-dimensional graphics application programming interface (API), all transformations are specified in **three dimensions**.
- **Internally**, **all coordinate positions are represented** as **four- element column vectors**, and all transformations are represented using 4×4 matrices.

Basic OpenGL Geometric Transformations

- A 4× 4 translation matrix is constructed with the following routine:

```
glTranslate* (tx, ty, tz);
```

  - Translation parameters **tx, ty, and tz can be assigned any real-number values,** and the single suffix code to be affixed to this function is either **f (float) or d (double).**
  - For two-dimensional applications, we set **tz = 0.0; and a two-dimensional** position is represented as a four-element column matrix with the *z component* equal to 0.0.
  - For example, we translate subsequently defined coordinate positions 25 units in the *x direction* and *−10* units in the *y direction* with the *statement*

glTranslatef (25.0, -10.0, 0.0);

- Similarly, a **4 × 4 rotation matrix** is generated with

  glRotate* (theta, vx, vy, vz);

  - where the vector **v = (vx, vy, vz) can have any floating-point values for its components.**
  - This vector defines the **orientation for a rotation axis** that passes through the coordinate origin.

- **Rotation in two-dimensional systems** is rotation about the *z axis*, *specified* as a unit vector with *x and y components of zero*, *and a z component of 1.0.*
- *For* example, the statement **glRotatef (90.0, 0.0, 0.0, 1.0);** sets up the matrix for a 90◦ rotation about the *z axis*.

- We obtain a **4 × 4 scaling matrix** with respect to the coordinate origin with the following routine:

glScale* (sx, sy, sz);

- Scaling in a two-dimensional system involves changes in the *x and y dimensions, so a typical two-dimensional scaling operation has a* ***z scaling factor of 1.0***
- ***Negative values will generate reflections.***
- For example, the following statement produces a matrix that scales by a factor of **2 in the *x direction***, ***scales by a factor of 3 in the y direction***, *and reflects* with respect to the *x axis:*

glScalef (2.0, -3.0, 1.0);

Two-Dimensional Viewing

The Two-Dimensional Viewing Pipeline:

- A section of a two-dimensional scene that is selected for display is called a **clipping window** ***because all parts of the scene outside the selected section are "clipped"*** off.
- The only part of the scene that shows up on the screen is what is inside the **clipping window**.
  - Also called as ***world window*** *or* ***the viewing window***.
- *Clipping window can also refer to a selected section of* a scene that is ***eventually converted to pixel patterns within a display window*** on the video monitor.

- Graphics packages allow us also to control the placement within the display window using **another "window"** called the **viewport.**
  - **Objects *inside the clipping window are mapped to the viewport***, and it is the viewport that is then positioned within the display window.
  - The **clipping window selects *what we want to see***; *the* ***viewport indicates where it is to be viewed on the output*** **device**.
  - By changing the position of a viewport, we can *view objects at different positions* on the display area of an output device.
  - **Multiple viewports** can be used to display different sections of a scene at different screen positions.
  - by varying the size of viewports→change the size and proportions of displayed objects.
  - achieve zooming effects by successively mapping different-sized clipping windows onto a fixed-size viewport.

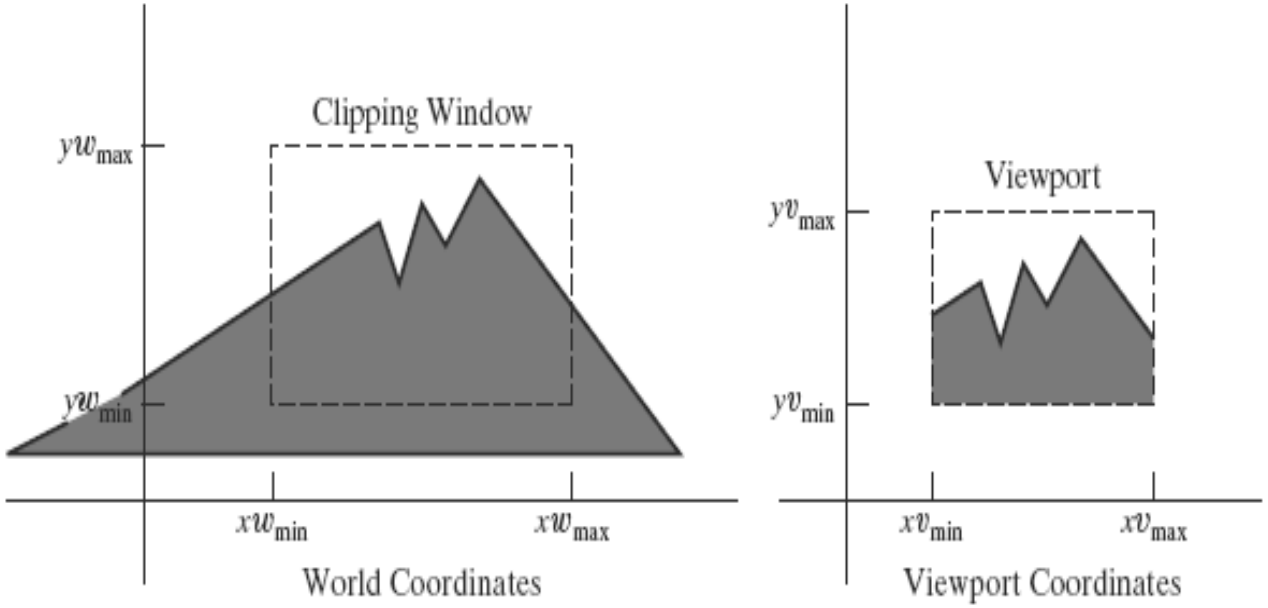


FIGURE 1
A clipping window and associated viewport, specified as rectangles aligned with the coordinate axes.

- The **mapping of a two-dimensional**, **world-coordinate scene description** to **device coordinates** is called a **two-dimensional viewing transformation.** **Sometimes** this transformation is simply referred to as the ***window-to-viewport transformation*** or **the *windowing transformation***.

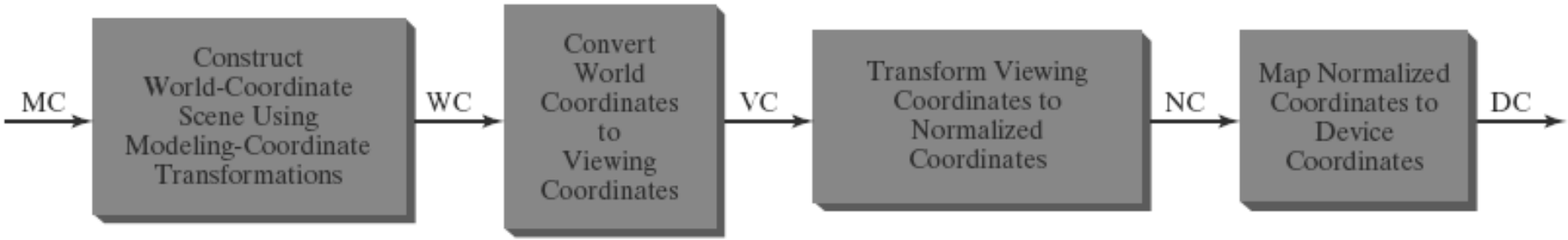


FIGURE 2
Two-dimensional viewing-transformation pipeline.

- Once a world-coordinate scene has been constructed, we could set up a separate two-dimensional, **viewing coordinate reference frame for specifying the clipping window.**
- **But the clipping window** is often just defined in world coordinates, so viewing coordinates for two-dimensional applications are the same as world coordinates.

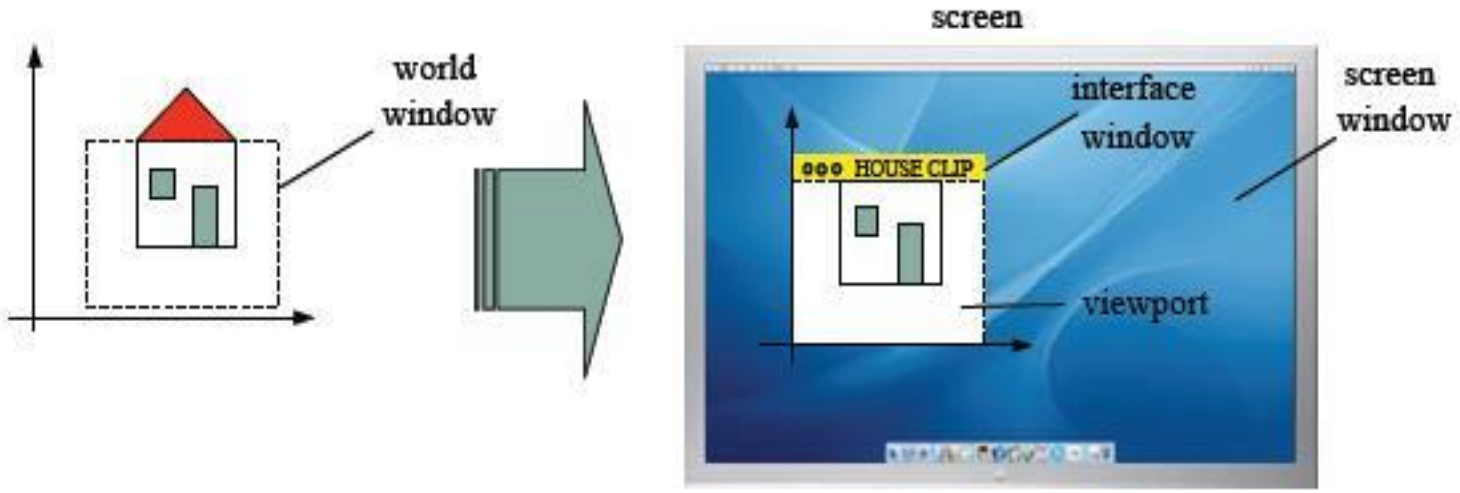
- To make the **viewing process independent of the requirements of any output device**, *graphics systems convert object descriptions to normalized coordinates and apply the clipping routines*.
  - Some systems use normalized coordinates in the range from 0 to 1, and others use a normalized range from −1 to 1
- Depending upon the **graphics library in use**, the **viewport is defined** either in *normalized coordinates or in screen coordinates after the normalization process.*
- At the **final step of the viewing transformation**, the *contents of the viewport are transferred to positions within the display window*.

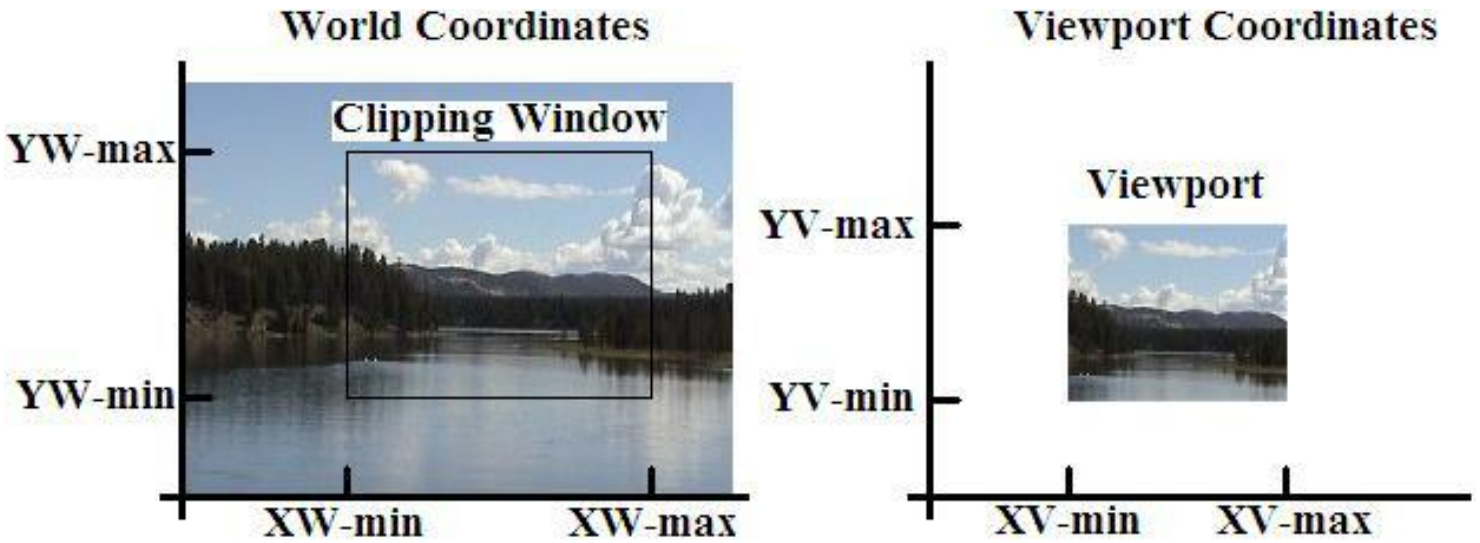
- Clipping ***is usually performed in normalized coordinates***.
  - This allows us to **reduce computations by first concatenating the various transformation matrices.**

Normalization and Viewport Transformations:

- There are many devices out there which use different screens altogether and different screen sizes and having different resolutions.
- Pixels depend on screen resolution. So, an image will look differently in different devices i.e. it will be **device dependent**.
- Hence, **normalized device coordinates is used** that makes an image device independent
- This **transformation of world coordinates to normalized device coordinates is called Normalization transformation**

Mapping the Clipping Window into a Normalized Viewport:





- First consider a **viewport** defined with normalized coordinate values between **0 and 1**.
- **Object descriptions are transferred** to this normalized space using a transformation that **maintains the same relative placement** of a point in the **viewport as it had in the clipping window**.
  - **For example:** If a coordinate position is at the center of the clipping window, for instance, it would be mapped to the center of the viewport.
- Figure 6 illustrates this window-to-viewport mapping. Position (*xw, yw) in the clipping window is mapped to position (xv, yv) in the associated* viewport.

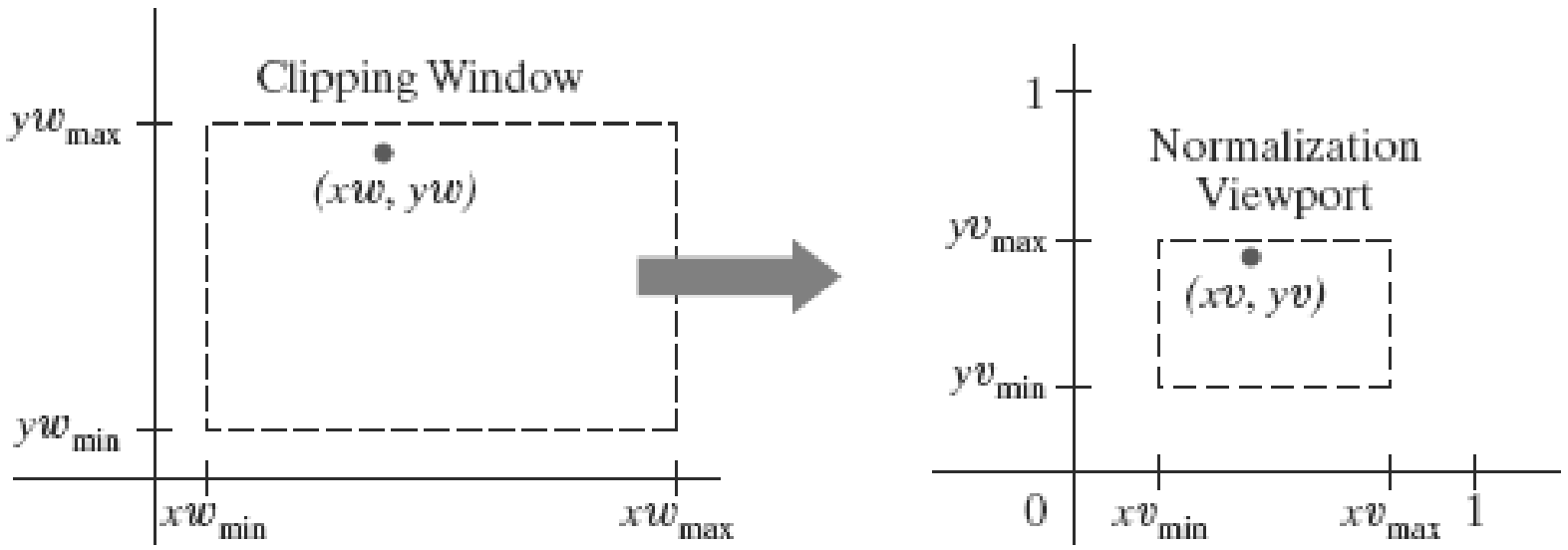
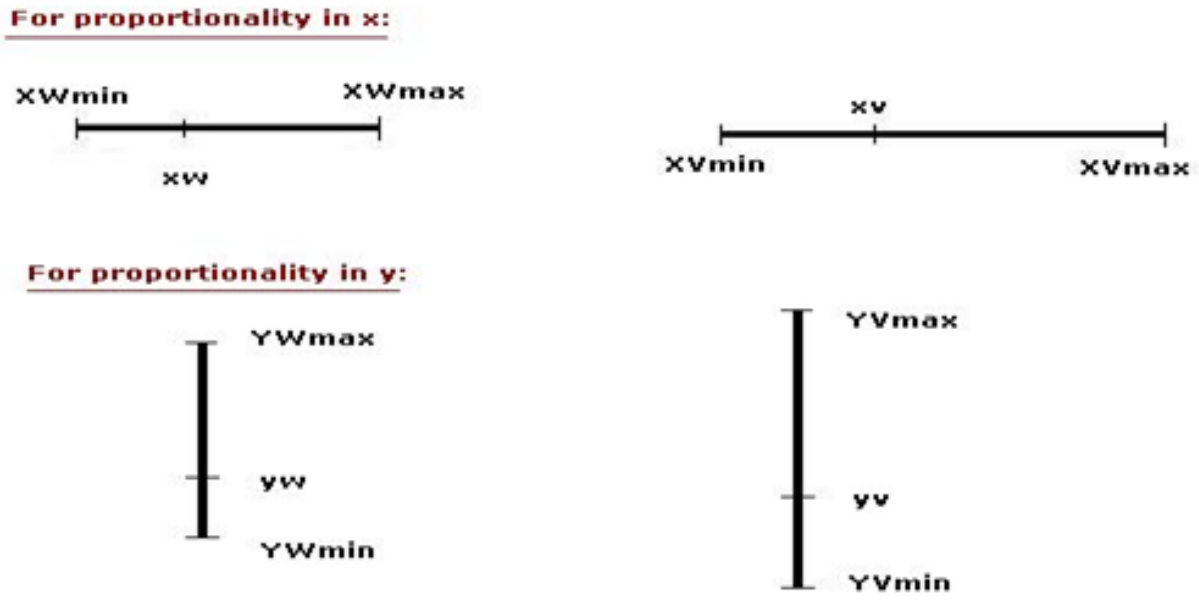


FIGURE 6

A point (xw, yw) in a world-coordinate clipping window is mapped to viewport coordinates (xv, yv), within a unit square, so that the relative positions of the two points in their respective rectangles are the same.

- Mapping should be "proportional" in the sense that **if xw is 30% of the way from the left edge of the world window**, then xv is 30% of the way from the left edge of the viewport.
- Similarly, **if yw is 30% of the way from the bottom edge of the world window**, then yv is 30% of the way from the bottom edge of the viewport. The picture below shows this proportionality.



- To transform the world-coordinate point into the same relative position within the viewport, we require that

$$\frac{xv - xv_{min}}{xv_{max} - xv_{min}} = \frac{xw - xw_{min}}{xw_{max} - xw_{min}}$$

$$\frac{yv - yv_{min}}{yv_{max} - yv_{min}} = \frac{yw - yw_{min}}{yw_{max} - yw_{min}}$$

- Solving these expressions for the viewport position (*xv, yv), we have*

$$xv = s_x xw + t_x$$

$$yv = s_y yw + t_y$$

- Where the scaling factors are

$$s_x = \frac{xv_{max} - xv_{min}}{xw_{max} - xw_{min}}$$

$$s_y = \frac{yv_{max} - yv_{min}}{yw_{max} - yw_{min}}$$

- and the translation factors are

$$t_x = \frac{xw_{max}\, xv_{min} - xw_{min}\, xv_{max}}{xw_{max} - xw_{min}}$$

$$t_y = \frac{yw_{max}\, yv_{min} - yw_{min}\, yv_{max}}{yw_{max} - yw_{min}}$$

- We could also obtain the transformation from world coordinates to viewport coordinates with the following sequence:

  1. **Scale the clipping window to the size of the viewport using a fixed-point** position of (*xwmin, ywmin).*
  2. **Translate (*xwmin, ywmin) to (xvmin, yvmin).***

- The scaling transformation in step (1) can be represented with the two dimensional matrix

$$\mathbf{S} = \begin{bmatrix} s_x & 0 & xw_{\min}(1-s_x) \\ 0 & s_y & yw_{\min}(1-s_y) \\ 0 & 0 & 1 \end{bmatrix}$$

- The two-dimensional matrix representation for the translation of the lower-left corner of the clipping window to the lower-left viewport corner is

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & xv_{\min} - xw_{\min} \\ 0 & 1 & yv_{\min} - yw_{\min} \\ 0 & 0 & 1 \end{bmatrix}$$

- And the **composite matrix representation** for the transformation to the normalized viewport is

$$\mathbf{M}_{\text{window, normviewp}} = \mathbf{T} \cdot \mathbf{S} = \begin{bmatrix} s_x & 0 & t_x \\ 0 & s_y & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

  - The window-to-viewport transformation maintains the relative placement of object descriptions.
  - An object inside the clipping window is mapped to a corresponding position inside the viewport.
  - Similarly, an object outside the clipping window is outside the viewport.
  - Relative proportions of objects, on the other hand, are **maintained only if** the aspect ratio of the viewport is the same as the aspect ratio of the clipping window.
    - *Otherwise,* world objects will be stretched or contracted *in either* the *x or y directions (or both) when displayed on the output device.*

Mapping the Clipping Window into a Normalized Square:

- Another approach to two-dimensional viewing is to **transform the clipping window into a normalized square**, clip in normalized coordinates, and **then transfer the scene description to a viewport specified in screen coordinates.**

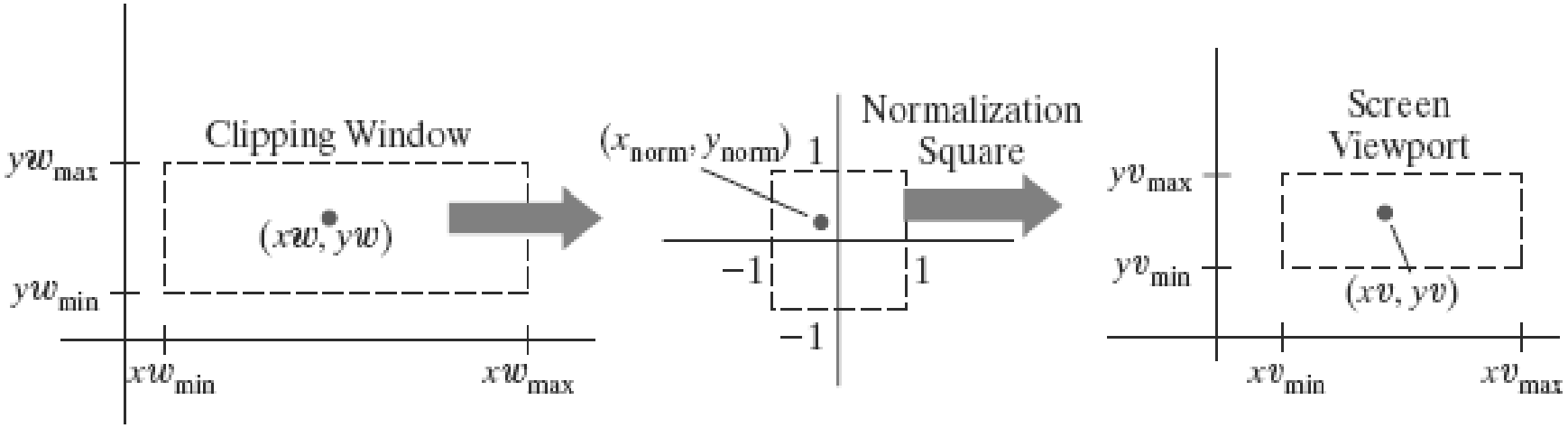


FIGURE 7

A point (xw, yw) in a clipping window is mapped to a normalized coordinate position (x_{norm}, y_{norm}), then to a screen-coordinate position (xv, yv) in a viewport. Objects are clipped against the normalization square before the transformation to viewport coordinates occurs.

- Objects outside the boundaries ***x = ±1 and y = ±1*** *are detected* and **removed from the scene description**.
- As the **final step of the viewing transformation**, the objects in the viewport are positioned within the display window.
- The **matrix for the normalization transformation is obtained** from following Equation 1 by substituting −1 for ***xvmin and yvmin*** *and substituting +1 for* ***xvmax and yvmax***.

$$\mathbf{M}_{\text{window, normviewp}} = \mathbf{T} \cdot \mathbf{S} = \begin{bmatrix} s_x & 0 & t_x \\ 0 & s_y & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

1

- Making these substitutions in the expressions for *tx*, *ty*, *sx*, *and sy*, *we have*

$$\mathbf{M}_{\text{window, normsquare}} = \begin{bmatrix} \frac{2}{xw_{\max} - xw_{\min}} & 0 & -\frac{xw_{\max} + xw_{\min}}{xw_{\max} - xw_{\min}} \\ 0 & \frac{2}{yw_{\max} - yw_{\min}} & -\frac{yw_{\max} + yw_{\min}}{yw_{\max} - yw_{\min}} \\ 0 & 0 & 1 \end{bmatrix}$$

- Similarly, after the clipping algorithms have been applied, the normalized square with edge length equal to 2 is transformed into a specified viewport.
- This time, we get the **transformation matrix** from Equation 1 by substituting −1 for *xwmin and ywmin and substituting +1 for xwmax and ywmax:*

$$\mathbf{M}_{\text{normsquare, viewport}} = \begin{bmatrix} \frac{xv_{\max} - xv_{\min}}{2} & 0 & \frac{xv_{\max} + xv_{\min}}{2} \\ 0 & \frac{yv_{\max} - yv_{\min}}{2} & \frac{yv_{\max} + yv_{\min}}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

- The **last step in the viewing process** is to position the viewport area in the display window.
- Typically, the lower-left corner of the viewport is placed at a coordinate position specified relative to the lower-left corner of the display window.

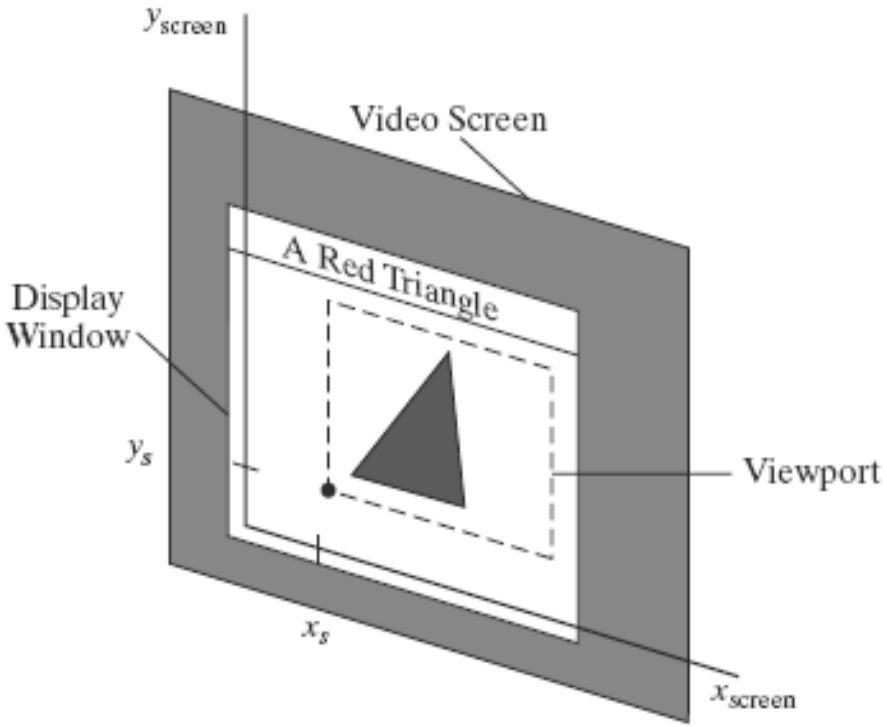


FIGURE 8
A viewport at coordinate position (x_s, y_s) within a display window.

OpenGL Two-Dimensional Viewing Functions

1. OpenGL Projection Mode:

- Before we select a clipping window and a viewport in OpenGL, we need to establish the appropriate mode for constructing the matrix to transform from world coordinates to screen coordinates.
- Therefore, we must first select the projection mode.

glMatrixMode (GL_PROJECTION);

- This designates the projection matrix as the current matrix, which is **originally set to the identity matrix**.

glLoadIdentity ( );

- This ensures that **each time we enter the projection mode**, the **matrix will be reset to the identity matrix** so that the new viewing parameters are not combined with the previous ones.

2. GLU Clipping-Window Function:

- To define a two-dimensional clipping window, we can use the GLU function:

  gluOrtho2D (xwmin, xwmax, ywmin, ywmax);

- This function specifies an orthogonal projection for **mapping the scene to the screen.**
- Normalized coordinates in the range from −1 to 1 are used in the OpenGL clipping routines.
- If we **do not specify a clipping window** in an application program, the default coordinates are *(xwmin, ywmin) = (−1.0, −1.0) and (xwmax, ywmax) = (1.0, 1.0).*
- Thus the default clipping window is the normalized square centered on the coordinate origin with a **side length of 2.0**.

3. OpenGL Viewport Function:

- We specify the viewport parameters with the OpenGL function

  glViewport (xvmin, yvmin, vpWidth, vpHeight);

- where all parameter values are given in ***integer screen coordinates relative to the display window.***
- Parameters **xvmin and yvmin specify the position of the lowerleft corner of the viewport relative to the lower-left corner of the display window**
- vpwidth and vpheight are **pixel width and height of the viewport**.
- If we do not invoke the **glViewport function in a program, the** default viewport size and position are the same as the size and position of the display window.

- **After the clipping routines have been applied**, positions within the normalized square are transformed into the viewport rectangle.
- **Coordinates for the upper-right corner of the viewport** are calculated for this transformation matrix in terms of the **viewport width and height:**

$$xvmax = xvmin + vpWidth$$

$$yvmax = yvmin + vpHeight$$

- For the final transformation, pixel colors for the primitives within the viewport are loaded into the refresh buffer at the specified screen locations.

4. Creating a GLUT Display Window:

- To access these GLUT routines for window-management system, we first need to initialize GLUT with the following function:

glutInit (&argc, argv);

- We have three functions in GLUT for defining a display window and choosing its dimensions and position:

glutInitWindowPosition (xTopLeft, yTopLeft);

glutInitWindowSize (dwWidth, dwHeight);

glutCreateWindow ("Title of Display Window");

5. Setting the GLUT Display-Window Mode and Color:

- Various display-window parameters are selected with the GLUT function

  glutInitDisplayMode (mode);

- choose a color mode (RGB or index) and different buffer combinations, and the selected parameters are combined with the logical **or** operation.
- The default mode is single buffering and the RGB (or RGBA) color mode, which is the same as setting this mode with the statement

  glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB)

- A background color for the display window is chosen in RGB mode with the OpenGL routine

  glClearColor (red, green, blue, alpha);

- In color-index mode, we set the display-window color with

  glClearIndex (index);

6. GLUT Display-Window Identifier

- Multiple display windows can be created for an application, and each is assigned a positive-integer **display-window identifier, starting with the value 1 for the** first window that is created.
- At the time that we initiate a display window, we can record its identifier with the statement

  windowID = glutCreateWindow ("A Display Window");

- Once we have saved the integer display-window identifier in variable name **windowID, we can use the identifier number to change display parameters or** to delete the display window.

7. Deleting a GLUT Display Window:

- The GLUT library also includes a function for deleting a display window that we have created.
- If we know the display window's identifier, we can eliminate it with the statement

glutDestroyWindow (windowID);

8. Current GLUT Display Window:

- When we specify any display-window operation, it is applied to the **current display window, which is either the last display window that we created or the one** we select with the following command:

glutSetWindow (windowID);

- In addition, at any time, we can **query the system to determine** which window is the current display window:

  currentWindowID = glutGetWindow ( );

- A value of 0 is returned by this function if there are **no display windows** or **if the current display window was destroyed**.

9. Relocating and Resizing a GLUT Display Window:

- We can ***reset the screen location*** for the current display window with

  glutPositionWindow (xNewTopLeft, yNewTopLeft);

- where the coordinates specify the new position for the upper-left display-window corner, relative to the upper-left corner of the screen.
- Similarly, the following function ***resets the size of the current display window:***

  glutReshapeWindow (dwNewWidth, dwNewHeight);

- To expand the current display window to fill the screen:

  glutFullScreen ( );

- The exact size of the display window after execution of this routine depends on the window-management system.
- Whenever the **size of a display window is changed**, its ***aspect ratio may change and objects may be distorted*** from their original shapes. We can adjust for a change in display-window dimensions using the statement

  glutReshapeFunc (winReshapeFcn);

- This GLUT routine is activated when the size of a display window is changed, and the new width and height are passed to its argument: the function **winReshapeFcn, in this example.**

10. Managing Multiple GLUT Display Windows:

- The GLUT library also has a number of routines for ***manipulating a display window in various ways.***
- These routines are particularly useful when we have *multiple display windows on the screen* and **we want to rearrange them or locate a particular display window.**
- **to convert** the ***current display window to an icon*** in the ***form of a small picture or symbol*** representing the window:

  glutIconifyWindow ( );

- The **label on this icon** will be the same name that we assigned to the window, but we **can change this** with the following command:

  glutSetIconTitle ("Icon Name");

- We also can ***change the name of the display window*** with a similar command:

  glutSetWindowTitle ("New Window Name");

- With **multiple display windows open on the screen**, ***some windows may overlap or totally obscure other display windows***.

- choose any **display window to be in front of all other windows** by first designating it as the current window, and then issuing the **"pop-window"** command:

  glutSetWindow (windowID);

  glutPopWindow ( );

- we can **“push” the current display window to the back** so that it is behind all other display windows.
- This sequence of operations is

  glutSetWindow (windowID);

  glutPushWindow ( );
- We can also take the **current window off the screen** with

  glutHideWindow ( );
- we can **return a “hidden” display window**, or one that has been converted to an icon, by designating it as the current display window and then invoking the function

  glutShowWindow ( );

11. GLUT Subwindows:

- Within a selected display window, we can set up any number of second-level display windows, which are called *subwindows*.
- *This provides a **means for partitioning** **display windows** **into different display sections.*** We create a subwindow with the following function:

glutCreateSubWindow (windowID, xBottomLeft, yBottomLeft, width, height);

- Parameter **windowID** *identifies the display window in which we want to set up the subwindow*.
  - All operations that is applied for first level windows are applied for subwindows
  - But we **cannot convert a GLUT subwindow to an icon**.

12. Selecting a Display-Window Screen-Cursor Shape:

- We can use the following GLUT routine to request a **shape for the screen cursor** that is to be used with the current window:

  glutSetCursor (shape);

- The **possible cursor shapes** that we can select are an **arrow pointing in a chosen direction**, **a bidirectional arrow**, **a rotating arrow**, **a crosshair**, **a wristwatch**, **a question mark**, or **even a skull** and **crossbones**

  GLUT_CURSOR_UP_DOWN → up-down arrow

  GLUT_CURSOR_CYCLE →rotating arrow

  GLUT_CURSOR_WAIT →wristwatch

  GLUT_CURSOR_DESTROY →skull and crossbones

- the exact shapes that we can use are system dependent.

13. Viewing Graphics Objects in a GLUT Display Window:

- After we have created a display window and selected its position, size, color, and other characteristics, we indicate what is to be shown in that window.
- Then we invoke the following function to assign something to that window:

  glutDisplayFunc (pictureDescrip);

- The argument is a routine that describes what is to be displayed in the current window called **pictureDescrip for this example, is referred** to as a *callback function because it is the routine that is to be executed whenever* GLUT determines that the display-window contents should be renewed.

– The following function is used to indicate that the contents of the current display window should be renewed:

glutPostRedisplay ( );

– This routine is also used when an additional object, such as a pop-up menu, is to be shown in a display window.

14. Executing the Application Program:

– When the program setup is complete and the display windows have been created and initialized, we need to issue the final GLUT command that signals execution of the program:

glutMainLoop ( );

- At this time, display windows and their graphic contents are sent to the screen.
- The program also enters the **GLUT processing loop that continually checks for** new "events," such as interactive input from a mouse or a graphics tablet.

15. Other GLUT Functions:

- The GLUT library provides a wide variety of routines to handle processes that are system dependent and to add features to the basic OpenGL library.
  - functions for generating bitmap and outline characters and it provides functions for loading values into a color table.
  - for displaying three-dimensional objects, either as solids or in a wireframe representation. These objects include a sphere, a torus, and the five regular polyhedra (cube, tetrahedron, octahedron, dodecahedron, and icosahedron).

- function that is to be **executed when there are no other events** for the system to process. We can do that with

  glutIdleFunc (function);

- The parameter for this GLUT routine could reference a background function or a procedure to update parameters for an animation when no other processes are taking place
- GLUT functions for obtaining and processing interactive input and for creating and managing menus. Individual routines are provided by GLUT for input devices such as a mouse, keyboard, graphics tablet, and spaceball.

- Finally, we can use the following function to query the system about some of the current state parameters:

glutGet (stateParam);

- This function returns an integer value corresponding to the symbolic constant we select for its argument.
- For example, we can obtain the *x-coordinate position for* the top-left corner of the current display window, relative to the top-left corner of the screen, with the constant **GLUT _WINDOW_ X;**
- **we can retrieve the current** display-window width or the screen width with **GLUT_ WINDOW _WIDTH or GLUT_ SCREEN_ WIDTH.**