CSE 411

Computer Graphics

Lecture #6 2D Geometric Transformations

Prepared & Presented by Asst. Prof. Dr. Samsun M. BAŞARICI

Objectives

- HB Ch. 7, GVS Ch. 3
- Basic 2D Transformations (rigid-body transformations):
  - Translation
  - Rotation
  - Scaling
- Homogenenous Representations and Coordinates
- 2D Composite Transformations

Objectives (cont.)

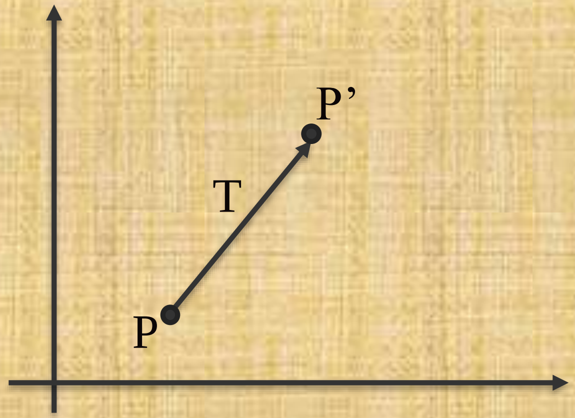
- Other Transformations:
  - Reflection
  - Shearing
- Raster Methods for Transformations and OpenGL
- Transformations between 2D Coordinate Systems and OpenGL

Geometric Transformations

- Sometimes also called modeling transformations
  - Geometric transformations: Changing an object's position (translation), orientation (rotation) or size (scaling)
  - Modeling transformations: Constructing a scene or hierarchical description of a complex object
- Others transformations: reflection and shearing operations

Basic 2D Geometric Transformations

- 2D Translation



  - $x' = x + t_x$, $y' = y + t_y$

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

  - P'=P+T
  - Translation moves the object without deformation (rigid-body transformation)

Basic 2D Geometric Transformations (cont.)

- 2D Translation
  - To move a line segment, apply the transformation equation to each of the two line endpoints and redraw the line between new endpoints
  - To move a polygon, apply the transformation equation to coordinates of each vertex and regenerate the polygon using the new set of vertex coordinates

2D Translation Routine

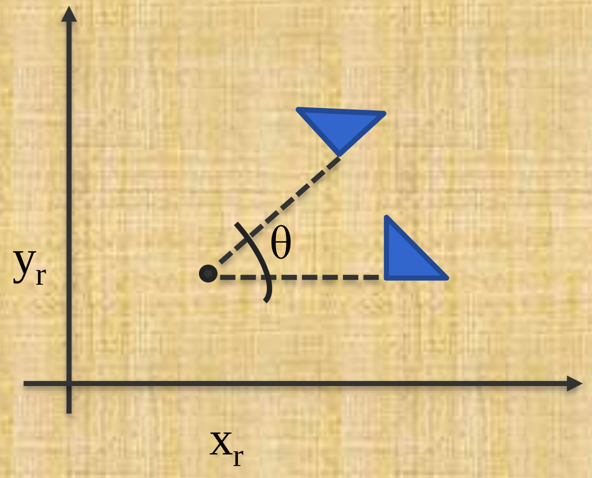
```
class wcPt2D {
    public:
       GLfloat x, y;
  };

  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 ( );
  }
```

Basic 2D Geometric Transformations (cont.)

- 2D Rotation
  - Rotation axis
  - Rotation angle
  - rotation point or pivot point (x_r, y_r)

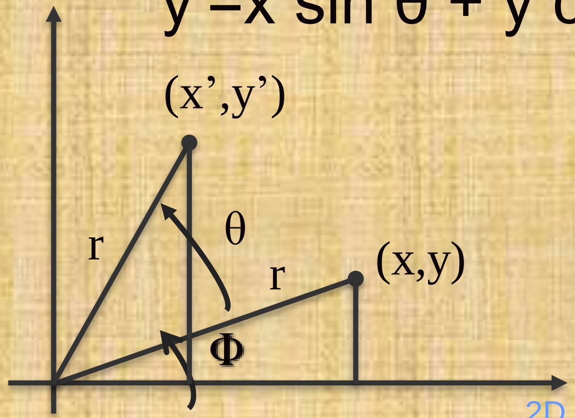


Basic 2D Geometric Transformations (cont.)

- 2D Rotation
  - If θ is positive → counterclockwise rotation
  - If θ is negative → clockwise rotation
  - Remember:
    - *cos(a + b) = cos a cos b - sin a sin b*
    - *cos(a - b) = cos a sin b + sin a cos b*

Basic 2D Geometric Transformations (cont.)

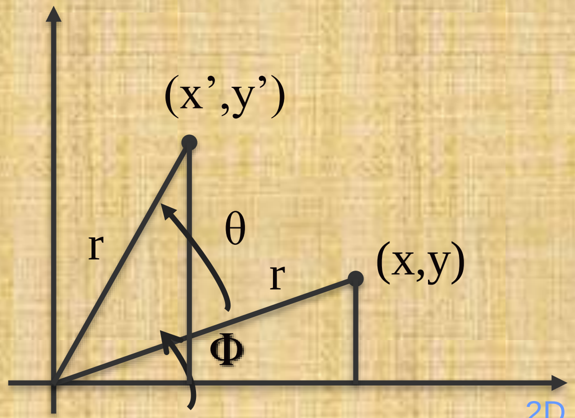
- 2D Rotation
  - At first, suppose the pivot point is at the origin
  - $x' = r\cos(\theta+\Phi) = r\cos\theta\cos\Phi - r\sin\theta\sin\Phi$
    $y' = r\sin(\theta+\Phi) = r\cos\theta\sin\Phi + r\sin\theta\cos\Phi$
  - $x = r\cos\Phi,\ y = r\sin\Phi$
  - $x' = x\cos\theta - y\sin\theta$
    $y' = x\sin\theta + y\cos\theta$



Basic 2D Geometric Transformations

- 2D Rotation
  - P'=R·P

$$R = \begin{bmatrix} \cos\Theta & -\sin\Theta \\ \sin\Theta & \cos\Theta \end{bmatrix}$$



Basic 2D Geometric Transformations (cont.)

- 2D Rotation
  - Rotation of a point about any specified position (x_r,y_r)

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

    $y' = y_r + (x - x_r)\sin\theta + (y - y_r)\cos\theta$
  - Rotations also move objects without deformation
  - A line is rotated by applying the rotation formula to each of the endpoints and redrawing the line between the new end points
  - A polygon is rotated by applying the rotation formula to each of the vertices and redrawing the polygon using new vertex coordinates

2D Rotation Routine

```
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 ( );
 }
```

Basic 2D Geometric Transformations (cont.)

- 2D Scaling
  - Scaling is used to alter the size of an object
  - Simple 2D scaling is performed by multiplying object positions (x, y) by scaling factors s_x and s_y

  $x' = x \cdot s_x$

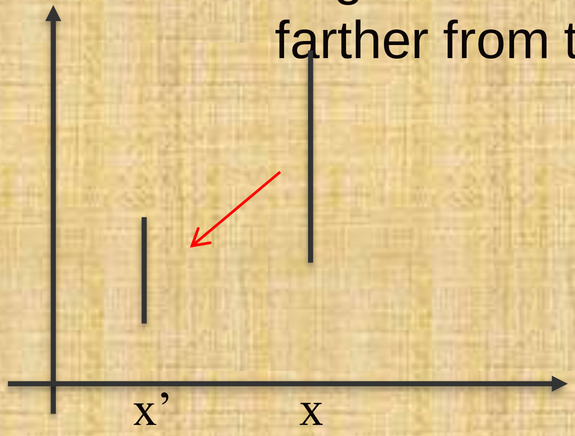
  $y' = y \cdot s_x$

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

  or $P' = S \cdot P$

Basic 2D Geometric Transformations (cont.)

- 2D Scaling
  - Any positive value can be used as scaling factor
    - Values less than 1 reduce the size of the object
    - Values greater than 1 enlarge the object
    - If scaling factor is 1 then the object stays unchanged
    - If $s_x = s_y$, we call it uniform scaling
    - If scaling factor <1, then the object moves closer to the origin and If scaling factor >1, then the object moves farther from the origin



Basic 2D Geometric Transformations (cont.)

- 2D Scaling
  - Why does scaling also reposition object?
  - Answer: See the matrix (multiplication)
  - Still no clue?
  - $\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x * s_x + y * 0 \\ x * 0 + y * s_y \end{bmatrix}$

Basic 2D Geometric Transformations (cont.)

- 2D Scaling
  - We can control the location of the scaled object by choosing a position called the **fixed point** (x_f, y_f)

    $x' - x_f = (x - x_f)\, s_x \qquad y' - y_f = (y - y_f)\, s_y$

    $x' = x \cdot s_x + x_f (1 - s_x)$

    $y' = y \cdot s_y + y_f (1 - s_y)$
  - Polygons are scaled by applying the above formula to each vertex, then regenerating the polygon using the transformed vertices

2D Scaling Routine

```
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 < n; 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 < n; k++)
        glVertex2v (vertsNew [k].x, vertsNew [k].y);
    glEnd ( );
  }
```

Matrix Representations and Homogeneous Coordinates

- Many graphics applications involve sequences of geometric transformations
  - Animations
  - Design and picture construction applications
- We will now consider matrix representations of these operations
  - Sequences of transformations can be efficiently processed using matrices

Matrix Representations and Homogeneous Coordinates (cont.)

- $P' = M_1 \cdot P + M_2$
  - P and P' are column vectors
  - M_1 is a 2 by 2 array containing multiplicative factors
  - M_2 is a 2 element column matrix containing translational terms
  - For translation M_1 is the identity matrix
  - For rotation or scaling, M_2 contains the translational terms associated with the pivot point or scaling fixed point

Matrix Representations and Homogeneous Coordinates (cont.)

- To produce a sequence of operations, such as scaling followed by rotation then translation, we could calculate the transformed coordinates one step at a time
- A more efficient approach is to combine transformations, without calculating intermediate coordinate values

Matrix Representations and Homogeneous Coordinates (cont.)

- Multiplicative and translational terms for a 2D geometric transformation can be combined into a single matrix if we expand the representations to 3 by 3 matrices
  - We can use the third column for translation terms, and all transformation equations can be expressed as matrix multiplications

Matrix Representations and Homogeneous Coordinates (cont.)

- Expand each 2D coordinate (x,y) to three element representation (x_h,y_h,h) called **homogeneous coordinates**
- h is the **homogeneous parameter** such that

  $x = x_h/h, \quad y = y_h/h,$
- → infinite homogeneous representations for a point
- A convenient choice is to choose h = 1

Matrix Representations and Homogeneous Coordinates (cont.)

- 2D Translation Matrix

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

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

Matrix Representations and Homogeneous Coordinates (cont.)

- 2D Rotation Matrix

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

or, **P’** = **R**(θ)·**P**

Matrix Representations and Homogeneous Coordinates (cont.)

- 2D Scaling Matrix

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

or, $\boldsymbol{P'} = \boldsymbol{S}(s_x, s_y) \cdot \boldsymbol{P}$

Inverse Transformations

- 2D Inverse Translation Matrix

$$T^{-1} = \begin{bmatrix} 1 & 0 & -t_x \\ 0 & 1 & -t_y \\ 0 & 0 & 1 \end{bmatrix}$$

- By the way:

$$T^{-1} * T = I$$

Inverse Transformations (cont.)

- 2D Inverse Rotation Matrix

$$R^{-1} = \begin{bmatrix} \cos\Theta & \sin\Theta & 0 \\ -\sin\Theta & \cos\Theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- And also:

$$R^{-1} * R = I$$

Inverse Transformations (cont.)

- 2D Inverse Rotation Matrix:
  - If θ is negative → clockwise
  - In

    $$R^{-1} * R = I$$

  - Only sine function is affected
  - Therefore we can say

    $$R^{-1} = R^{T}$$

  - Is that true?
  - Proof: It’s up to you ☺

Inverse Transformations (cont.)

- 2D Inverse Scaling Matrix

$$S^{-1} = \begin{bmatrix} \frac{1}{s_x} & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Of course:

$$S^{-1} * S = I$$

2D Composite Transformations

- We can setup a sequence of transformations as a **composite transformation matrix** by calculating the product of the individual transformations
- $P' = M_2 \cdot M_1 \cdot P$

  $= M \cdot P$

2D Composite Transformations (cont.)

- Composite 2D Translations
  - If two successive translation are applied to a point P, then the final transformed location P' is calculated as

$$\mathbf{P}' = \mathbf{T}(t_{x_2}, t_{y_2}) \cdot \mathbf{T}(t_{x_1}, t_{y_1}) \cdot \mathbf{P} = \mathbf{T}(t_{x_1} + t_{x_2}, t_{y_1} + t_{y_2}) \cdot \mathbf{P}$$

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

2D Composite Transformations (cont.)

- Composite 2D Rotations

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

$$\begin{bmatrix} \cos\Theta_2 & -\sin\Theta_2 & 0 \\ \sin\Theta_2 & \cos\Theta_2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\Theta_1 & -\sin\Theta_1 & 0 \\ \sin\Theta_1 & \cos\Theta_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos(\Theta_1 + \Theta_2) & -\sin(\Theta_1 + \Theta_2) & 0 \\ \sin(\Theta_1 + \Theta_2) & \cos(\Theta_1 + \Theta_2) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

2D Composite Transformations (cont.)

- Composite 2D Scaling

$$\mathbf{S}\left(s_{x_2}, s_{y_2}\right) \cdot \mathbf{S}\left(s_{x_1}, s_{y_1}\right) = \mathbf{S}\left(s_{x_1} \cdot s_{x_2}, s_{y_1} \cdot s_{y_2}\right)$$

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

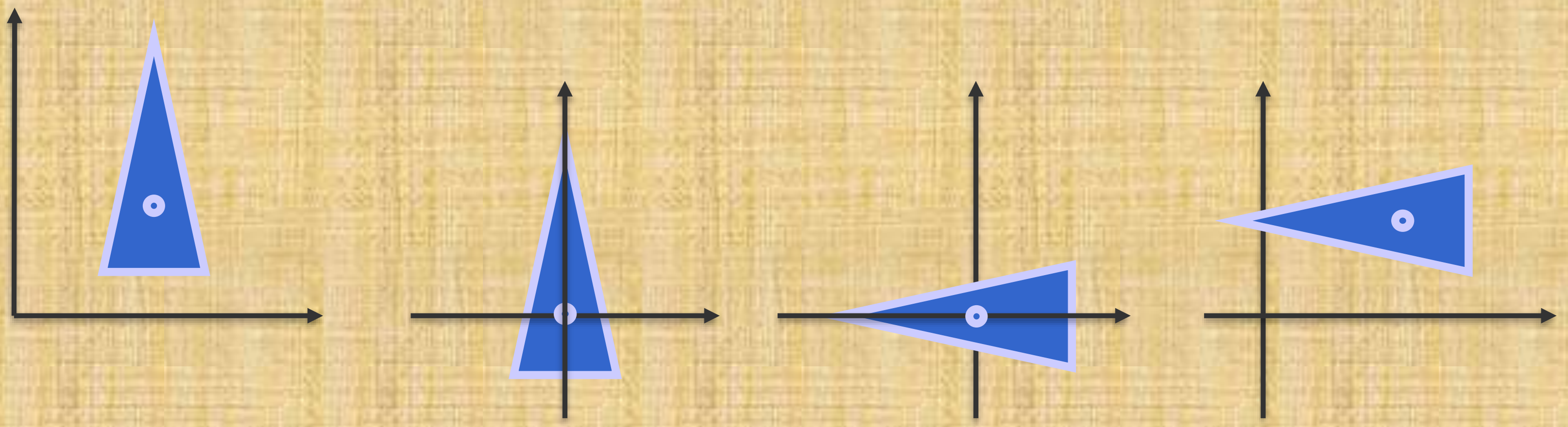
2D Composite Transformations (cont.)

- Don’t forget:
- Successive translations are additive
- Successive scalings are multiplicative
  - For example: If we triple the size of an object twice, the final size is nine (9) times the original
  - 9 times?
  - Why?
  - Proof: Again up to you ☺

General Pivot Point Rotation

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

General Pivot Point Rotation



2D Composite Transformations (cont.)

- General 2D Pivot-Point Rotation

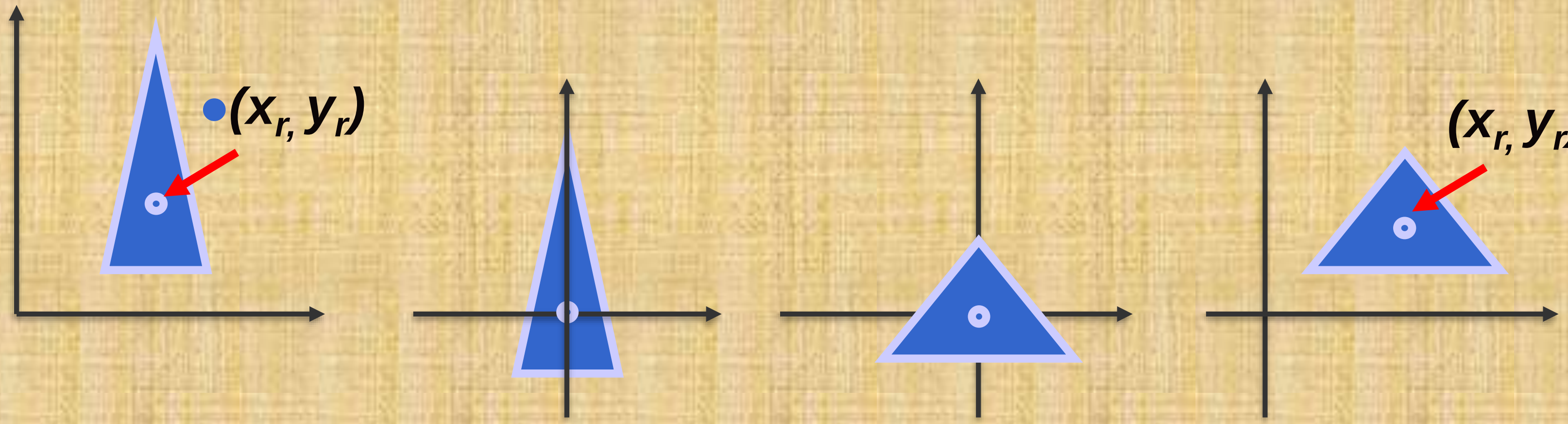
$$\begin{bmatrix} 1 & 0 & x_r \\ 0 & 1 & y_r \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\Theta & -\sin\Theta & 0 \\ \sin\Theta & \cos\Theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_r \\ 0 & 1 & -y_r \\ 0 & 0 & 1 \end{bmatrix}$$

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

General Fixed Point Scaling

- Steps:
  1. Translate the object so that the fixed point coincides with the coordinate origin.
  2. Scale the object about the origin.
  3. Translate the object so that the pivot point is returned to its original position.

General Fixed Point Scaling (cont.)



General Fixed Point Scaling (cont.)

- General 2D Fixed-Point Scaling:

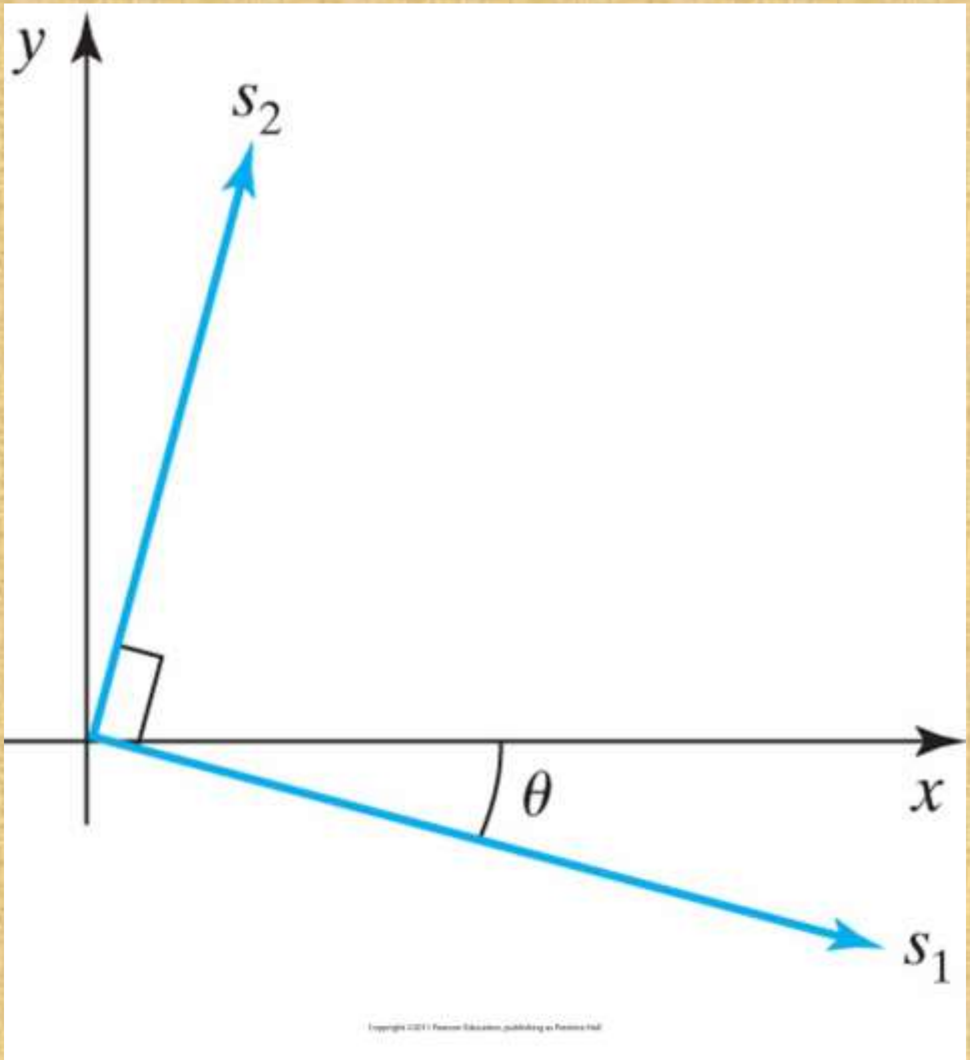
$$\begin{bmatrix} 1 & 0 & x_f \\ 0 & 1 & y_f \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 1 & 0 & -x_f \\ 0 & 1 & -y_f \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & x_f(1-s_x) \\ 0 & s_y & y_f(1-s_y) \\ 0 & 0 & 1 \end{bmatrix}$$

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

2D Composite Transformations (cont.)

- General 2D scaling directions:
  - Above: scaling parameters were along *x* and *y* directions
  - What about arbitrary directions?
  - Answer: See next slides

General 2D Scaling Directions



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

General 2D Scaling Directions (cont.)

- General procedure:
  1. Rotate so that directions coincides with *x* and *y* axes
  2. Apply scaling transformation $S(s_1, s_2)$
  3. Rotate back
- The composite matrix:

$$R^{-1}(\Theta) * S(s_1, s_2) * R(\Theta) = \begin{bmatrix} s_1 \cos^2 \Theta + s_2 \sin^2 \Theta & (s_2 - s_1) \cos \Theta \sin \Theta & 0 \\ (s_2 - s_1) \cos \Theta \sin \Theta & s_1 \sin^2 \Theta + s_2 \cos^2 \Theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

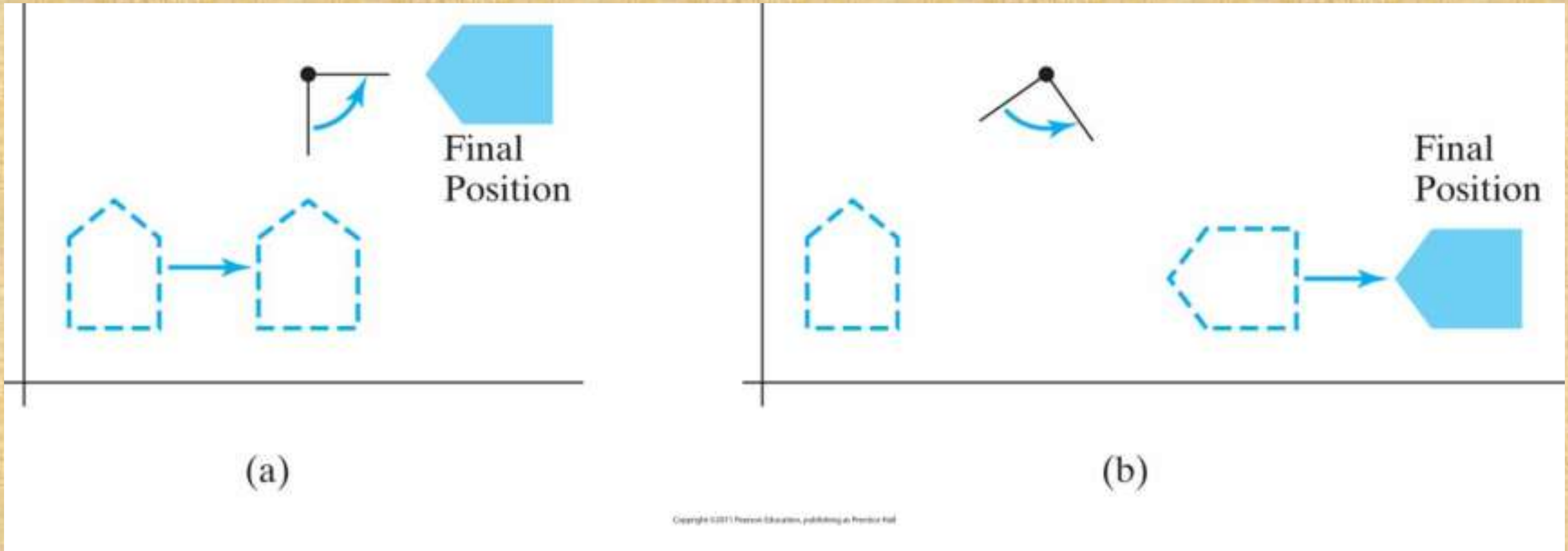
2D Composite Transformations (cont.)

- Matrix Concatenation Properties:
  - Matrix multiplication is associative !
    - $M_3 \cdot M_2 \cdot M_1 = (M_3 \cdot M_2) \cdot M_1 = M_3 \cdot (M_2 \cdot M_1)$
    - A composite matrix can be created by multiplicating left-to-right (premultiplication) or right-to-left (postmultiplication)
  - Matrix multiplication is ***not*** commutative !
    - $M_2 \cdot M_1 \neq M_1 \cdot M_2$

2D Composite Transformations (cont.)

- Matrix Concatenation Properties:
  - But:
    - Two successive rotations
    - Two successive translations
    - Two successive scalings
  - ***are*** commutative!
- Why?
- Proof: You got it: Up to you ☺ ☺

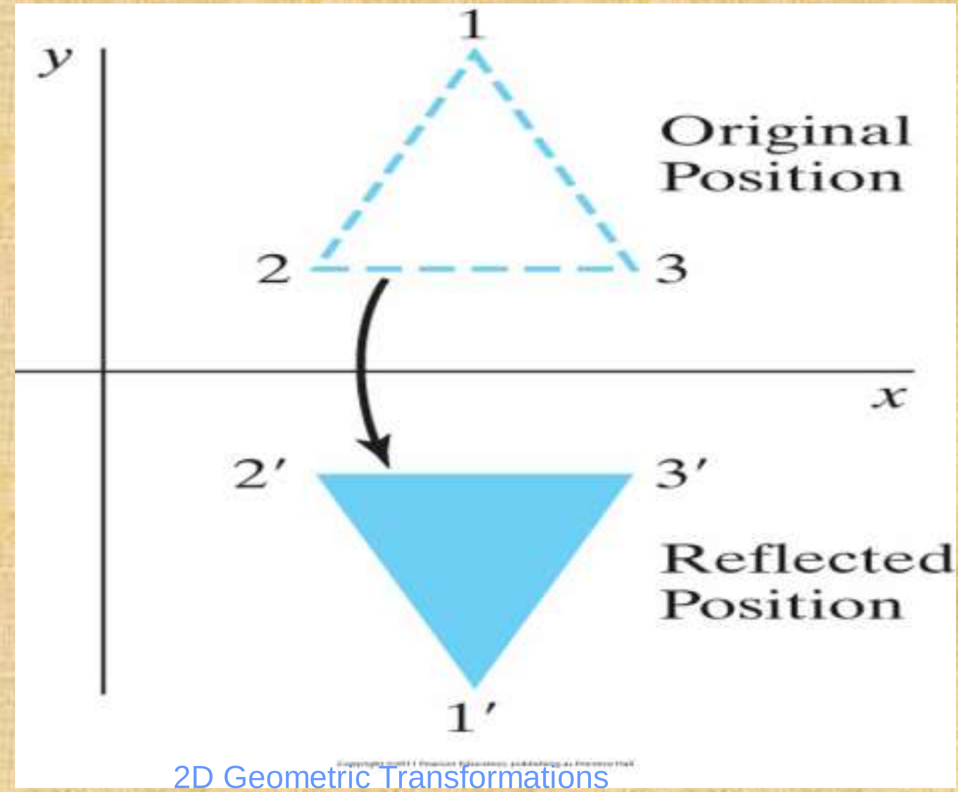
Reversing the order



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 2D Transformations

- Reflection
  - Transformation that produces a mirror image of an object



Other 2D Transformations (cont.)

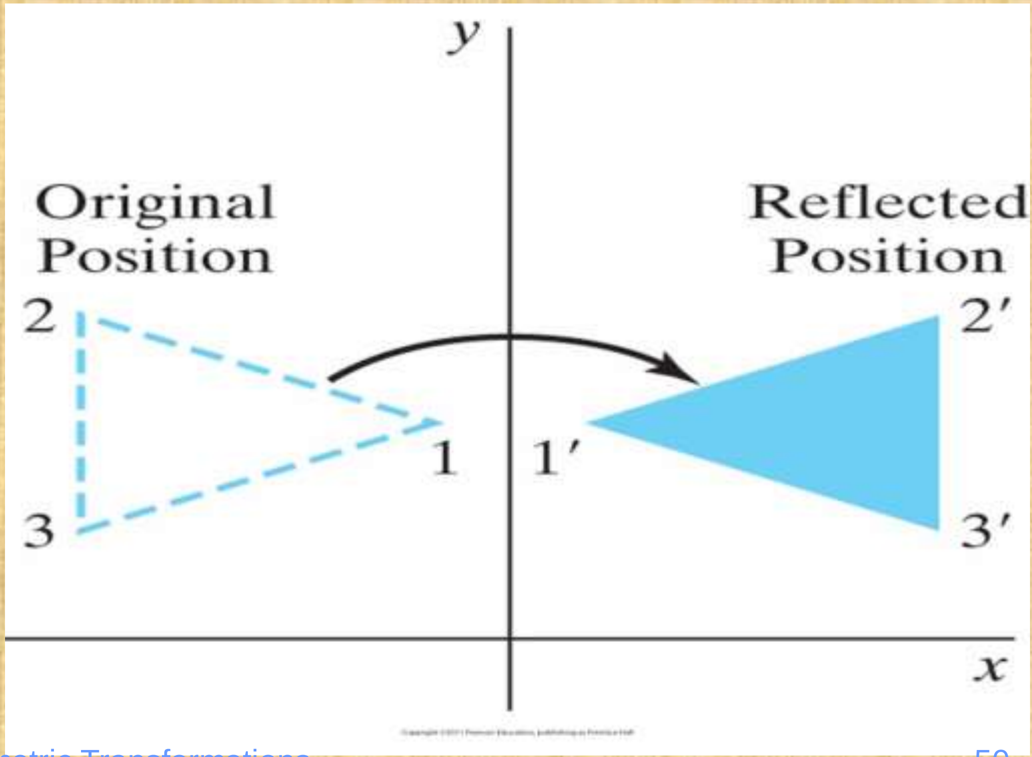
- Reflection
  - Image is generated relative to an axis of reflection by rotating the object 180° about the reflection axis
  - Reflection about the line y=0 (the x axis) (previous slide)

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

Other 2D Transformations (cont.)

- Reflection
  - Reflection about the line x=0 (the y axis)

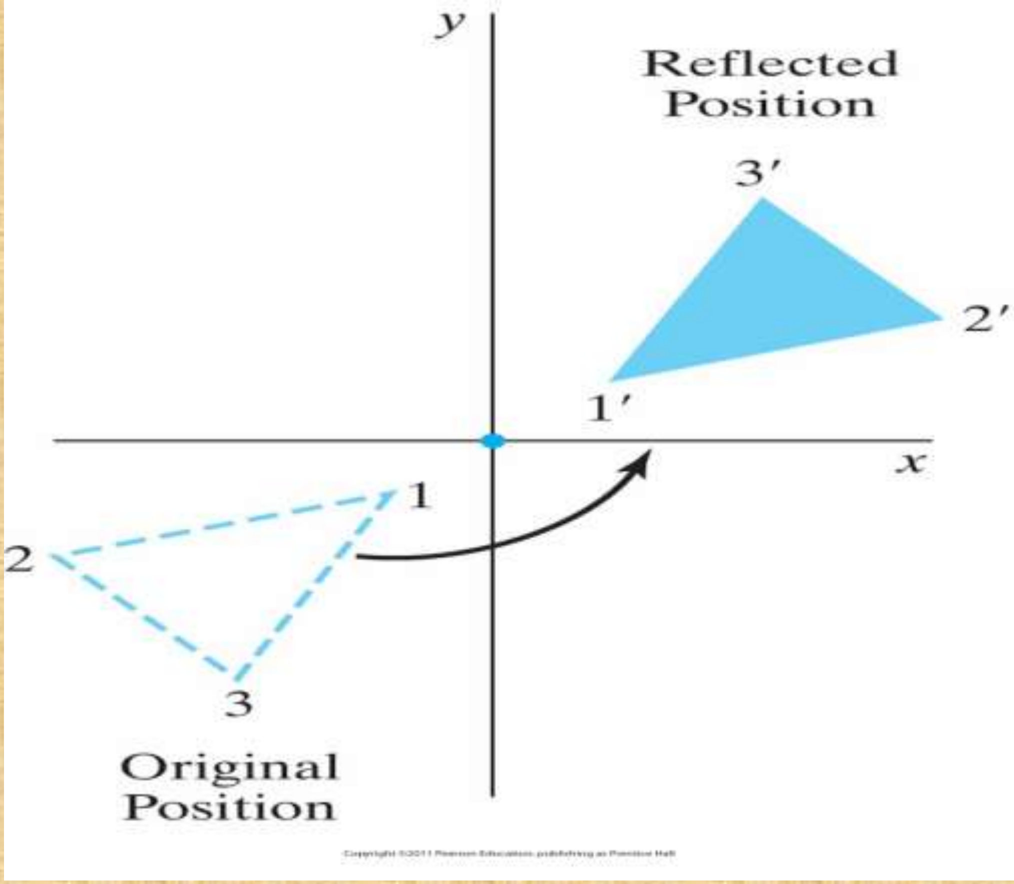
$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



Other 2D Transformations (cont.)

- Reflection about the origin

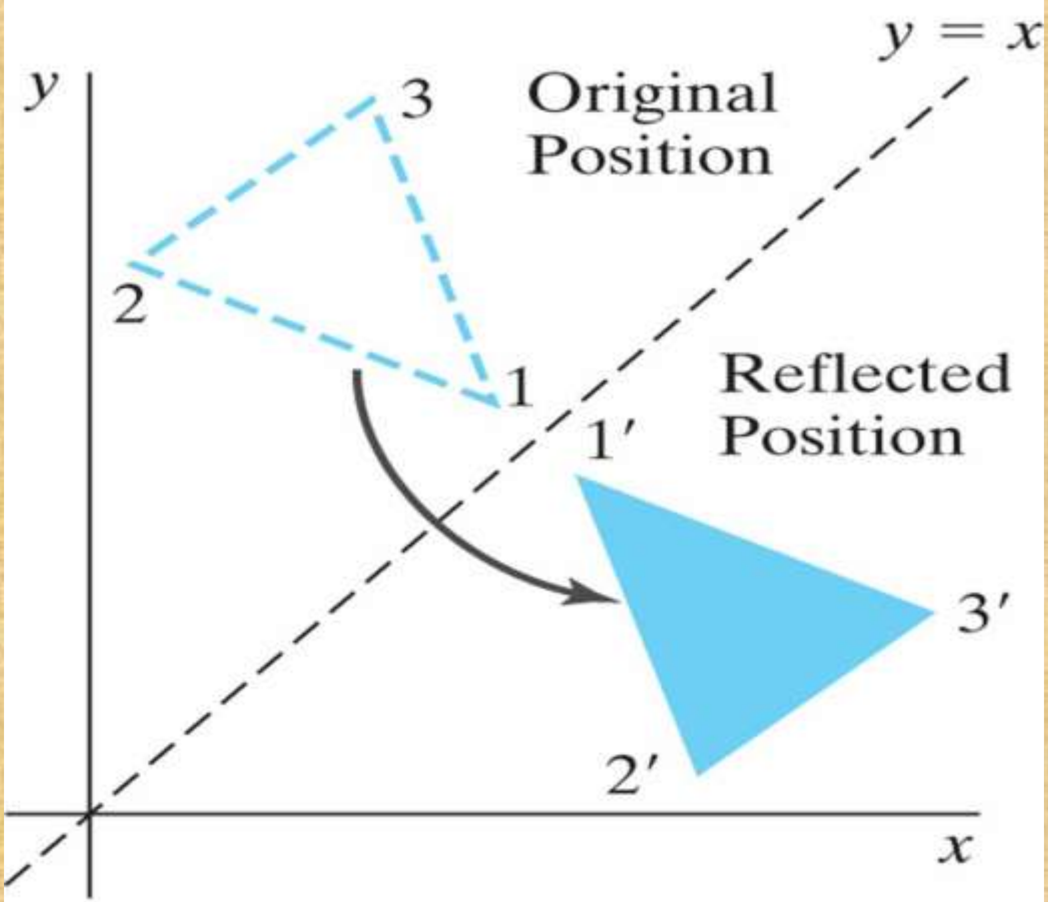
$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



Other 2D Transformations (cont.)

- Reflection about the line *y*=*x*

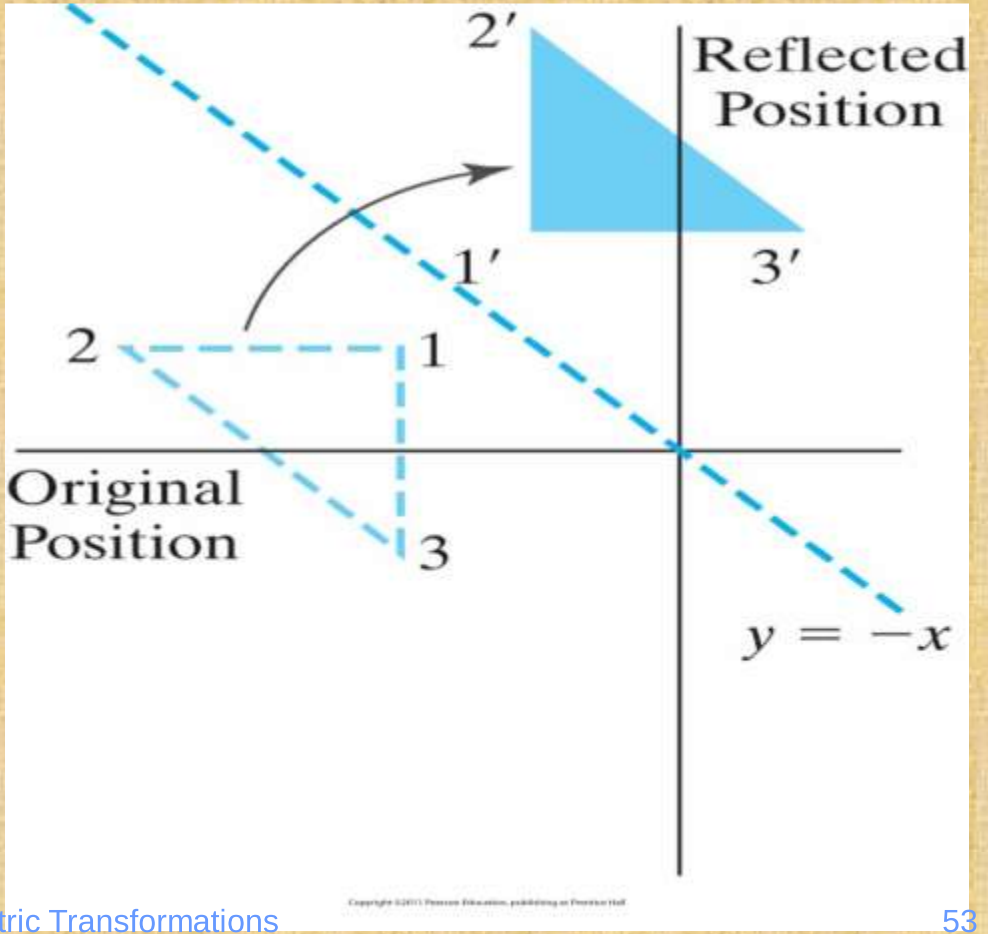
$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



Other 2D Transformations (cont.)

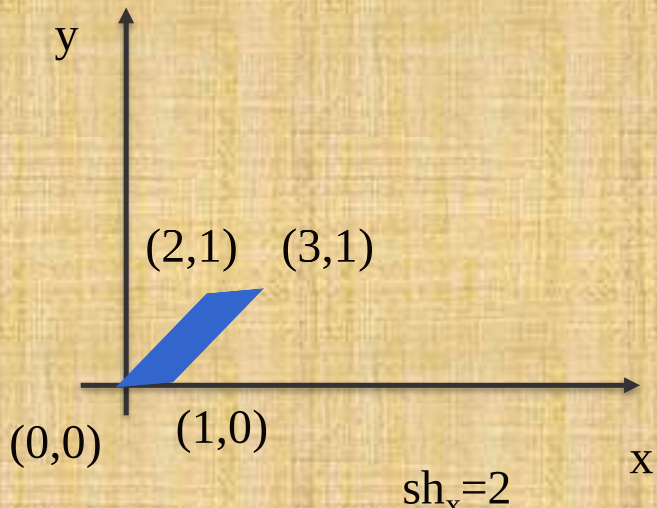
- Reflection about the line *y*=-*x*

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



Other 2D Transformations (cont.)

- Shear
  - Transformation that distorts the shape of an object such that the transformed shape appears as the object was composed of internal layers that had been caused to slide over each other



Other 2D Transformations (cont.)

- Shear
  - An x-direction shear relative to the x axis

$$\begin{bmatrix} 1 & sh_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \begin{aligned} x' &= x + sh_x \cdot y \\ y' &= y \end{aligned}$$

  - An y-direction shear relative to the y axis

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

Other 2D Transformations (cont.)

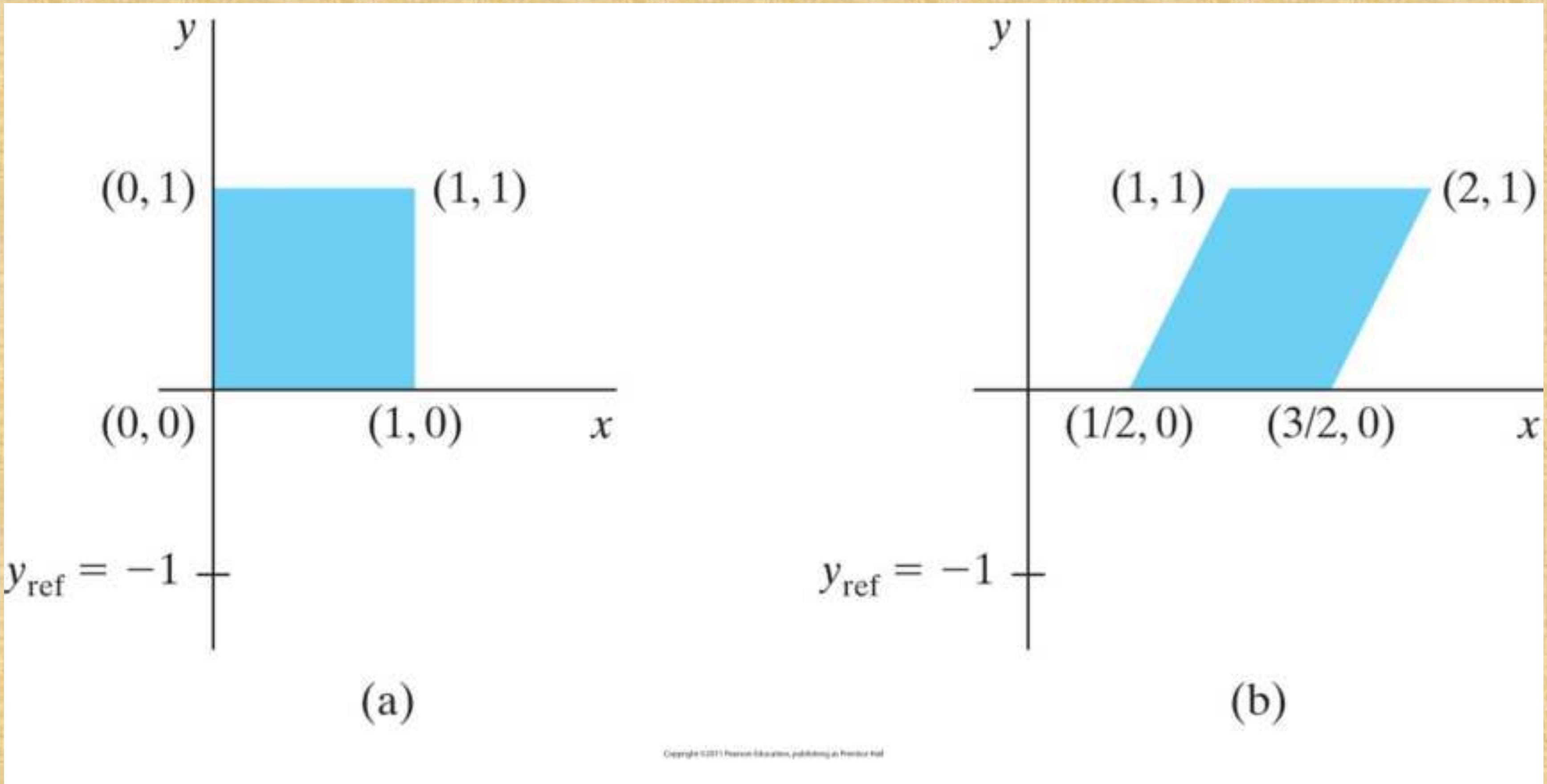
- Shear
  - x-direction shear relative to other reference lines

$$\begin{bmatrix} 1 & sh_x & -sh_x * y_{ref} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$x' = x + sh_x * \left(y - y_{ref}\right)$$

$$y' = y$$

Example



A unit square (a) is transformed to a shifted parallelogram (b) with $sh_x = 0.5$ and $y_{ref} = -1$ in the shear matrix from Slide 56

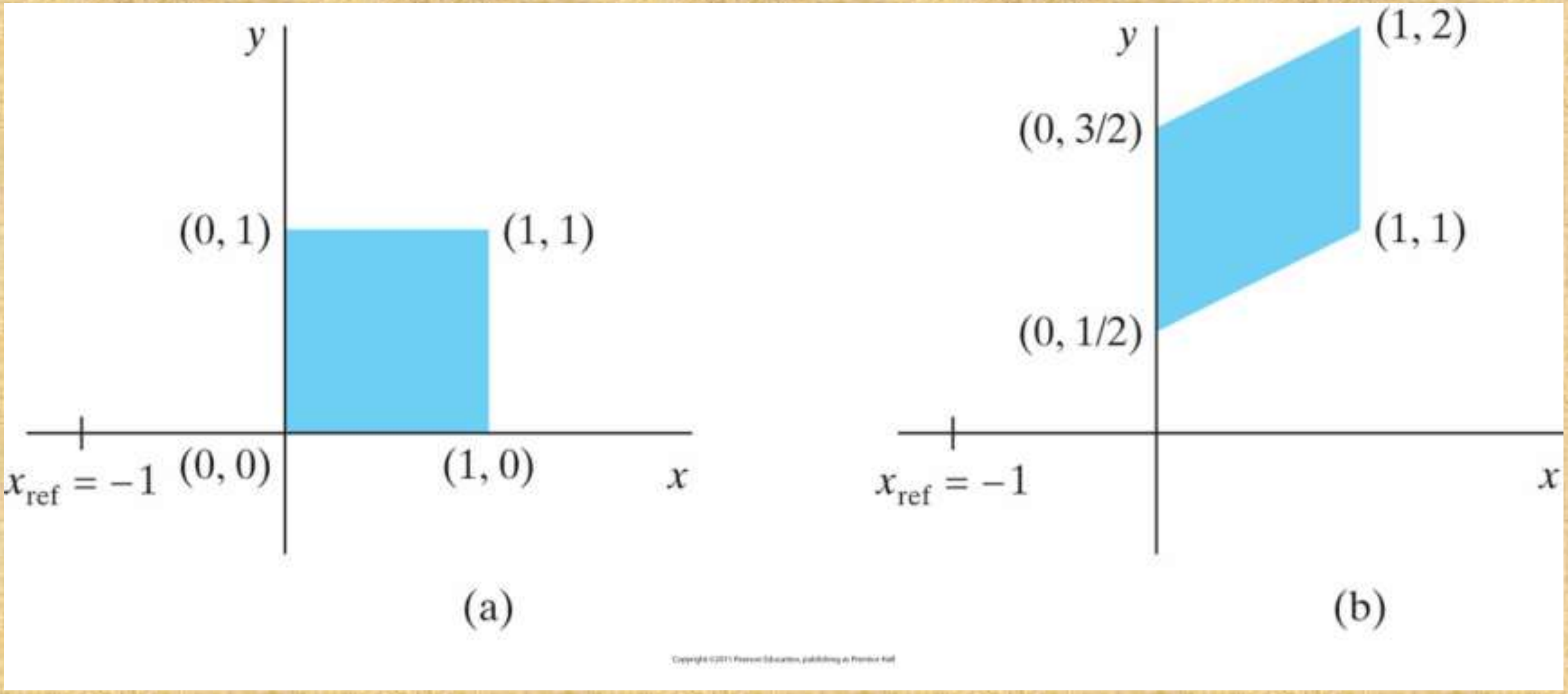
Other 2D Transformations (cont.)

- Shear
  - y-direction shear relative to the line $x = x_{ref}$

$$\begin{bmatrix} 1 & 0 & 0 \\ sh_y & 1 & -sh_y * x_{ref} \\ 0 & 0 & 1 \end{bmatrix}$$

$$x' = x$$
$$y' = x + sh_y * \left(x - x_{ref}\right)$$

Example



A unit square (a) is turned into a shifted parallelogram (b) with parameter values $sh_y = 0.5$ and $x_{ref} = -1$ in the y -direction shearing transformation from Slide 58

Raster Methods for Transformations and OpenGL

- This slide is intentionally left blank
- Your responsibility to fill it ☺

Transformation Between Coordinate Systems

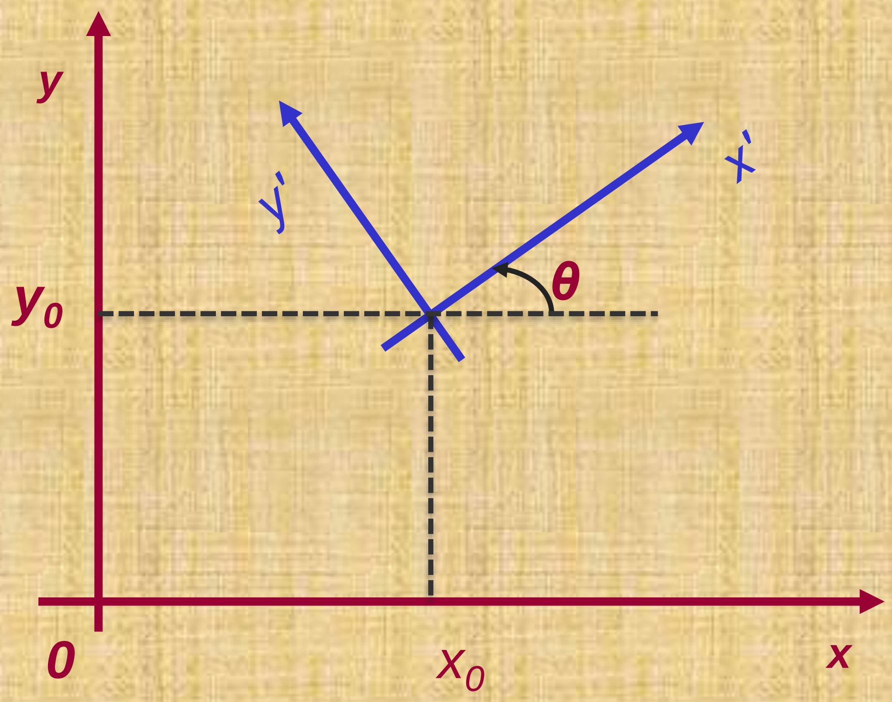
- Individual objects may be defined in their local cartesian reference system.
- The local coordinates must be transformed to position the objects within the scene coordinate system.

Transformation Between Coordinate Systems

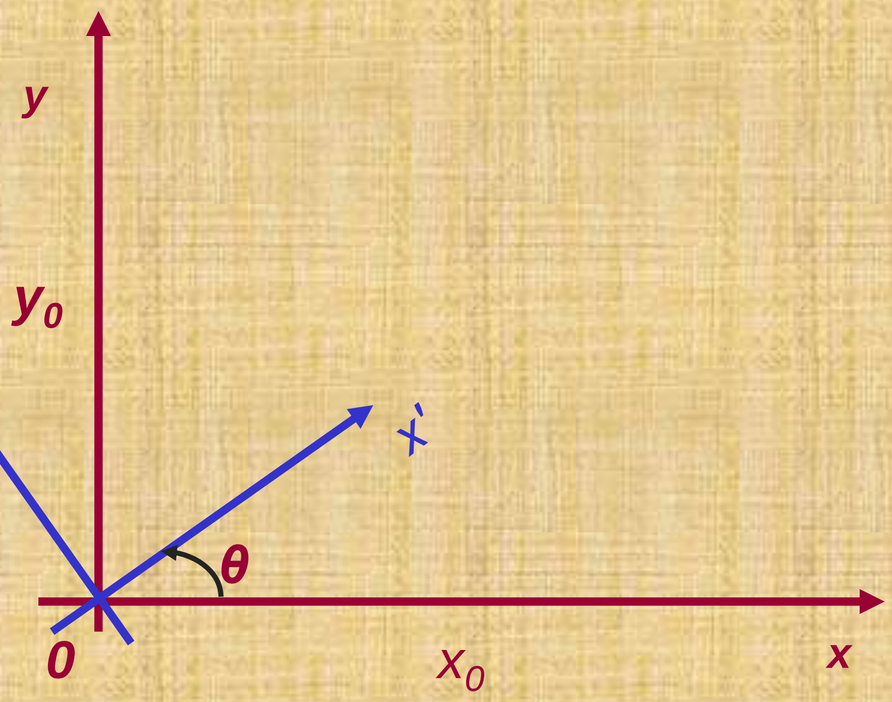
Steps for coordinate transformation

1. Translate so that the origin (x_0, y_0) of the *x′*-*y′* system is moved to the origin of the *x*-*y* system.
2. Rotate the *x′* axis on to the axis *x.*

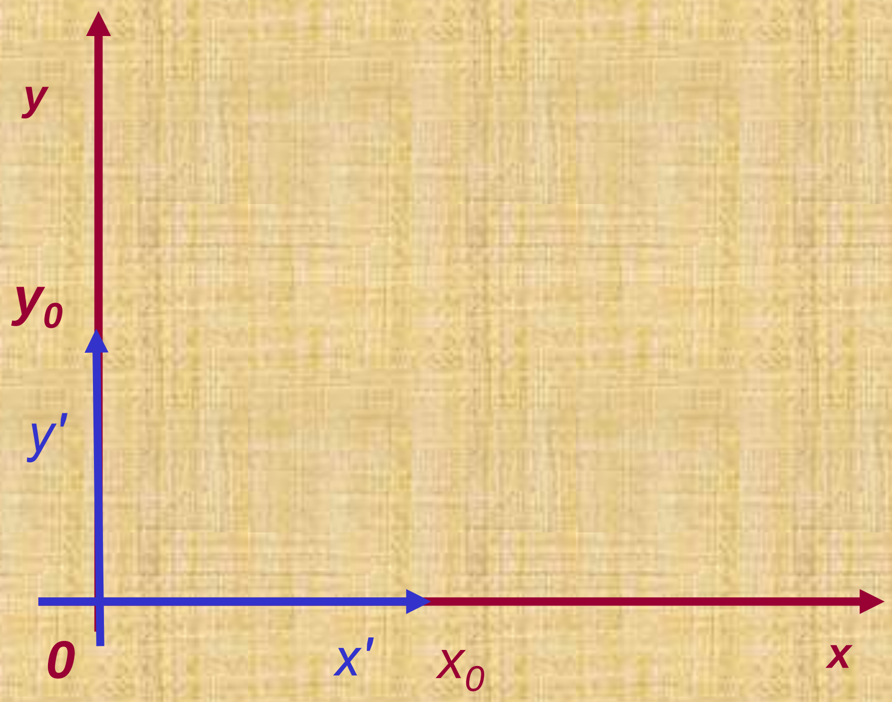
Transformation Between Coordinate Systems (cont.)



Transformation Between Coordinate Systems (cont.)



Transformation Between Coordinate Systems (cont.)



Transformation Between Coordinate Systems (cont.)

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

$$\mathbf{R}(-\theta) = \begin{bmatrix} Cos\theta & Sin\theta & 0 \\ -Sin\theta & Cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{M}_{xy,x\prime y\prime} = \mathbf{R}(-\theta) \cdot \mathbf{T}(-x_0, -y_0)$$

Transformation Between Coordinate Systems (cont.)

An alternative method:

-Specify a vector **V** that indicates the direction for the positive y′ axis. Let

$$\mathbf{v} = \frac{\mathbf{V}}{|\mathbf{V}|} = (v_x, v_y)$$

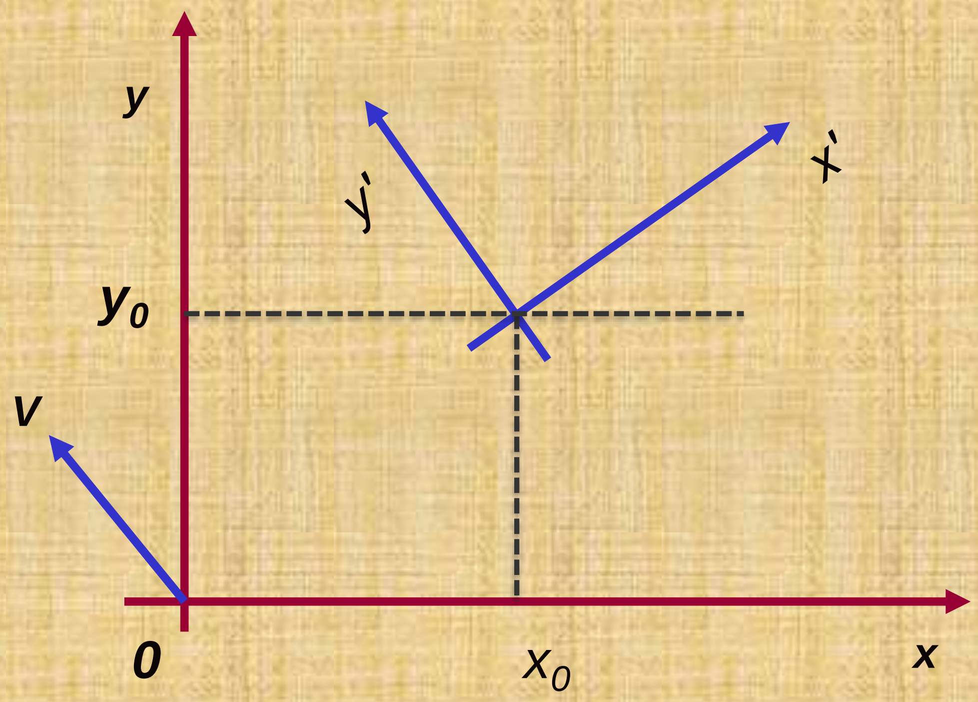
-Obtain the unit vector **u**=(u_x ,u_y) along the x′ axis by rotating **v** 90^0 clockwise.

Transformation Between Coordinate Systems (cont.)

- Elements of any rotation matrix can be expressed as elements of orthogonal unit vectors. That is, the rotation matrix can be written as

$$\mathbf{R} = \begin{bmatrix} u_x & u_y & 0 \\ v_x & v_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Transformation Between Coordinate Systems (cont.)



OpenGL Geometric Transformation Functions

- A separate function is available for each of the basic geometric transformations

AND

- All transformations are specified in **three** dimensions
- Why?
- Answer: Remember; OpenGL was developed as 3D library
- But how to perform 2D transformations?
- Answer: Set $z = 0$

Basic OpenGL Geometric Transformations

- Translation
  - glTranslate* (tx, ty, tz);
    - * is either f or d
    - tx, ty and tz are any real number
    - For 2D, set tz=0.0
- Rotation
  - glRotate* (theta, vx, vy, vz);
    - * is either f or d
    - theta is rotation angle in degrees (internally converted to radian)
    - Vector v=(vx, vy, vz) defines the orientation for a rotation axis that passes through the coordinate origin
    - For 2D, set vz=1.0 and vx=vy=0.0

Basic OpenGL Geometric Transformations (cont.)

- Scaling
  - glScale* (sx, sy, sz);
    - * is either f or d
    - sx, sy and sz are any real number
    - Negative values generate **reflection**
    - Zero values can cause error because inverse matrix cannot be calculated
- All routines construct a 4x4 transformation matrix
- OpenGL uses composite matrices
  - Be careful with the order

OpenGL Matrix Operations

- glMatrixMode(.);
  - Projection Mode: Determines how the scene is projected onto the screen
  - Modelview Mode: Used for storing and combining geometric transformations
  - Texture Mode: Used for mapping texture patterns to surfaces
  - Color Mode: Used to convert from one color mode to another

OpenGL Matrix Operations

- Modelview matrix, used to store and combine geometric transformations
  - glMatrixMode(GL_MODELVIEW);
- A call to a transformation routine generates a matrix that is multiplied by the current matrix
- To assign the identity matrix to the current matrix
  - glLoadIdentity();

OpenGL Matrix Operations (cont.)

- Alternatively:
  - glLoadMatrix* (elements16);
  - To assign other values to the elements of the current matrix
  - In column-major order:
    - First four elements in first column
    - Second four elements in second column
    - Third four elements in third column
    - Fourth four elements in fourth column

OpenGL Matrix Operations (cont.)

- Concatenating a specified matrix with current matrix:
  - glMultMatrix* (otherElements16);
  - Current matrix is postmultiplied (right-to-left) by the specified matrix
- Warning:
- Matrix notation m_{jk} means:
  - In OpenGL: j → column, k → row
  - In mathematics: j → row, k → column

OpenGL Matrix Stacks

- OpenGL maintains a matrix stack for transformations
- Initially the modelview stack contains only the identity matrix
- More about it:
  - Coming soon

OpenGL Transformation Routines

- For example, assume we want to do in the following order:
  - translate by +2, -3, +4,
  - rotate by 45^0 around axis formed between origin and 1, 1, 1
  - scale with respect to the origin by 2 in each direction.
- Our code would be

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();                    //start with identity
glScalef(2.0,2.0,2.0);   //Note: Start with the LAST operation
glRotatef(45.0,1.0,1.0,1.0);
glTranslatef(2.0,-3.0, 4.0); //End with the FIRST operation
```

OpenGL Transformation Functions

TABLE 7-1

Summary of OpenGL Geometric Transformation Functions

Function	Description
glTranslate*	Specifies translation parameters.
glRotate*	Specifies parameters for rotation about any axis through the origin.
glScale*	Specifies scaling parameters with respect to coordinate origin.
glMatrixMode	Specifies current matrix for geometric-viewing transformations, projection transformations, texture transformations, or color transformations.
glLoadIdentity	Sets current matrix to identity.
glLoadMatrix* (elems);	Sets elements of current matrix.
glMultMatrix* (elems);	Postmultiplies the current matrix by the specified matrix.
glPixelZoom	Specifies two-dimensional scaling parameters for raster operations.

Next Lecture

3D Geometric Transformations

References

- Donald Hearn, M. Pauline Baker, Warren R. Carithers, “Computer Graphics with OpenGL, 4th Edition”; Pearson, 2011
- Sumanta Guha, “Computer Graphics Through OpenGL: From Theory to Experiments”, CRC Press, 2010
- Edward Angel, “Interactive Computer Graphics. A Top-Down Approach Using OpenGL”, Addison-Wesley, 2005