2D and 3D graphics with OpenGL: 2D Geometric Transformations: Basic 2D Geometric Transformations, matrix representations and homogeneous coordinates, 2D Composite transformations, other 2D transformations, raster methods for geometric transformations, OpenGL raster transformations, OpenGL geometric transformations function

1. **Explain translation, rotation, scaling with digram and matrx form 10marks**
2. **Explain OpenGL raster transformations & OpenGL geometric transformations function 10marks**
3. **Explain Homogeneous Coordinates**

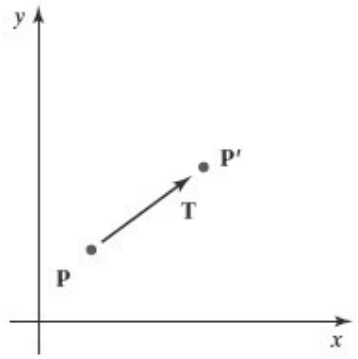
Geometric Transformations

Operations that are applied to the geometric description of an object to change its position, orientation, or size are called **geometric transformations**

Basic Two-Dimensional Geometric Transformations

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.
- We are moving the original point position along a straight-line path to its new location.
- 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



- The translation values of x' and y' is calculated as

- The $x' = x + t_x, \qquad y' = y + t_y$ translation distance pair (t_x, t_y) is called a **translation vector** or **shift vector Column vector representation is given as**

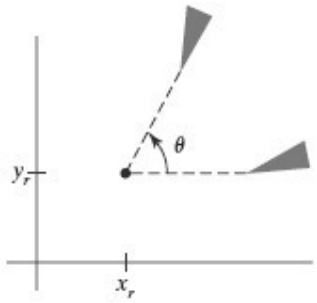
$$\mathbf{P} = \begin{bmatrix} x \\ y \end{bmatrix}, \qquad \mathbf{P}' = \begin{bmatrix} x' \\ y' \end{bmatrix}, \qquad \mathbf{T} = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

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

- Translation $\mathbf{P}' = \mathbf{P} + \mathbf{T}$ is a *rigid-body transformation* that moves objects without deformation.

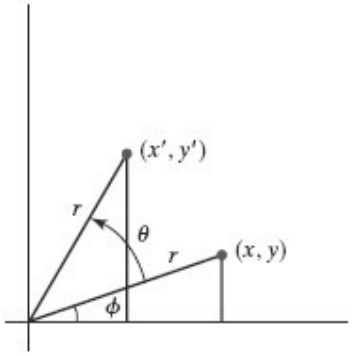
Two-Dimensional Rotation

- We generate a **rotation** transformation of an object by specifying a **rotation axis** and a **rotation angle.**
- A two-dimensional rotation of an object is obtained by repositioning the object along a circular path in the *xy* plane.
- In this case, we are rotating the object about a rotation axis that is perpendicular to the *xy* plane (parallel to the coordinate *z* axis)**.**
- 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 above Figure , and a negative value rotates objects in the clockwise direction.

- The angular and coordinate relationships of the original and transformed point positions as Shown below fig.



- 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.
- we can express the transformed coordinates in terms of angles θ and φ as

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

- The original coordinates of the point in polar coordinates are $x = r\cos\phi, \qquad y = r\sin\phi$
- Substituting expressions of x and y in the eaquations of x' and y' we get
- We $x' = x\cos\theta - y\sin\theta$, $y' = x\sin\theta + y\cos\theta$ can write the rotation equations in the matrix form
  - **P'** = **R**· **P**
    - Where the rotation matrix is,

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

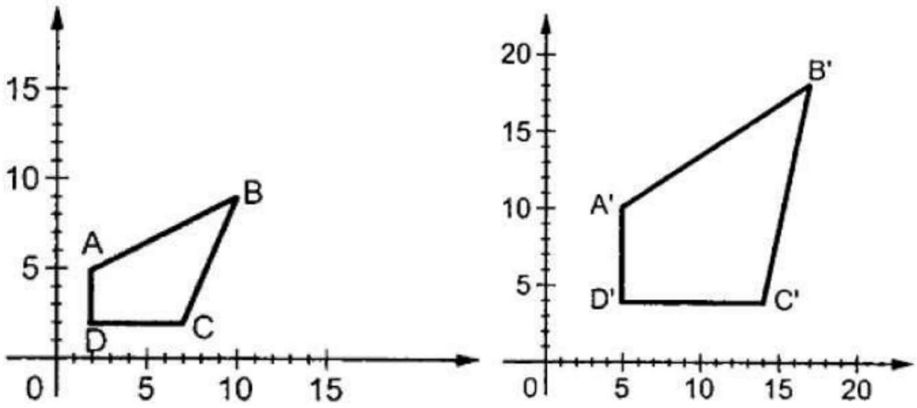
Two dimensional Scaling

To change the size of an object, scaling transformation is used. In the scaling process, you either expand or compress the dimensions of the object. Scaling can be achieved by multiplying the original coordinates of the object with the scaling factor to get the desired result.

Let us assume that the original coordinates are X,Y , the scaling factors are (SX, SY), and the produced coordinates are X′,Y′X′,Y′. This can be mathematically represented as shown below –

X' = X . SX and Y' = Y . SY

Where S is the scaling matrix. The scaling process is shown in the following figure.



If we provide values less than 1 to the scaling factor S, then we can reduce the size of the object. If we provide values greater than 1, then we can increase the size of the object.

Matrix Representations and Homogeneous Coordinates

Each of the three basic two-dimensional transformations (translation, rotation, and scaling) can be expressed using matrices. For transformations, we represent points and transformations in a way that allows easy combination and application using matrix multiplication.

1. **Translation**:
   - **Matrix Form**: $$P' = M_1 \cdot P + M_2$$

Where:

- $P = \begin{bmatrix} x \\ y \end{bmatrix}$ is the original point.
- $P' = \begin{bmatrix} x' \\ y' \end{bmatrix}$ is the transformed point.
- M_1 is a 2x2 identity matrix for translation.
- $M_2 = \begin{bmatrix} T_x \\ T_y \end{bmatrix}$ is the translation vector.

- **Translation Matrix**

$$M_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$M_2 = \begin{bmatrix} T_x \\ T_y \end{bmatrix}$$

2. **Rotation**:
   - Matrix Form:

$$P' = R(\theta) \cdot P$$

Where:

- $R(\theta)$ is the rotation matrix with angle θ.

- Rotation Matrix:

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

3. **Scaling**:
   - Matrix Form:

$$P' = S(s_x, s_y) \cdot P$$

Where:

- $S(s_x, s_y)$ is the scaling matrix with scaling factors s_x and s_y.

- Scaling Matrix:

$$S(s_x, s_y) = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

Homogeneous Coordinates

Homogeneous coordinates allow combining translation, rotation, and scaling into a single matrix operation by expanding 2D coordinates to 3D.

- **Conversion to Homogeneous Coordinates**:

- Original point: (x,y)
- Homogeneous point: (x,y,1)

- **General Homogeneous Coordinate**

$$(h \cdot x, h \cdot y, h)$$

- Common choice: $h = 1$

- **Transformation Matrices in Homogeneous Coordinates**:

1. Translation:

- Matrix:

$$T(T_x, T_y) = \begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix}$$

- Operation:

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

2. Rotation:

- Matrix:

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

- Operation:

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

3. Scaling:

- Matrix:

$$S(s_x, s_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Operation:

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

Two-Dimensional Composite Transformations

✓ Forming products of transformation matrices is often referred to as a **concatenation,** or **composition,** of matrices if we want to apply two transformations to point position **P**, the transformed location would be calculated as

$$\begin{aligned} \mathbf{P}' &= \mathbf{M}_2 \cdot \mathbf{M}_1 \cdot \mathbf{P} \\ &= \mathbf{M} \cdot \mathbf{P} \end{aligned}$$

✓ The coordinate position is transformed using the composite matrix **M**, rather than applying the individual transformations **M**1 and then**M**2.

Composite Two-Dimensional Translations

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

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

where **P** and **P'** are represented as three-element, homogeneous-coordinate column vectors

- Also, the composite transformation matrix for this sequence of translations is

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

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

Composite Two-Dimensional Rotations

- ✓ Two successive rotations applied to a point **P** produce the transformed position
- ✓ By multiplying the two rotation matrices, we can verify that two successive rotations are additive:

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

$$\mathbf{R}(\theta 2) \cdot \mathbf{R}(\theta 1) = \mathbf{R}(\theta 1 + \theta 2)$$

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

$$\mathbf{P'} = \mathbf{R}(\theta 1 + \theta 2) \cdot \mathbf{P}$$

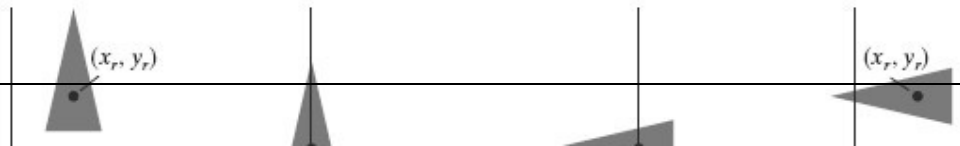
Composite Two-Dimensional Scalings

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

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

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

General Two-Dimensional Pivot-Point Rotation



- ✓ 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.

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

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

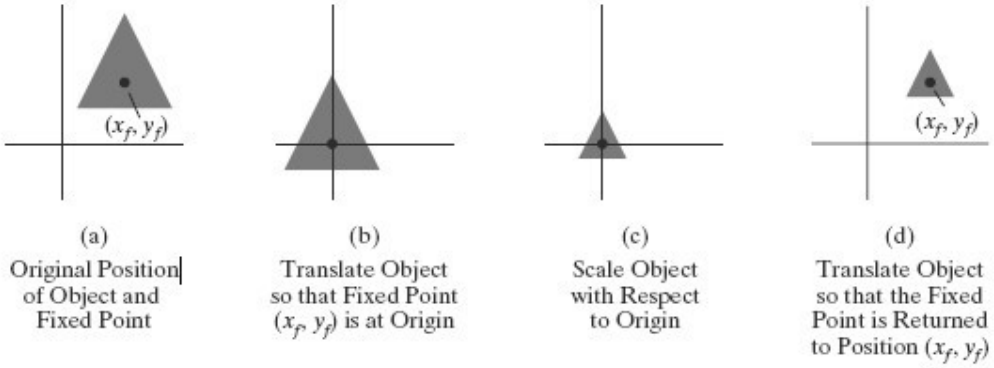
$$= \begin{bmatrix} \cos\theta & -\sin\theta & x_r(1-\cos\theta) + y_r \sin\theta \\ \sin\theta & \cos\theta & y_r(1-\cos\theta) - x_r \sin\theta \\ 0 & 0 & 1 \end{bmatrix}$$

which can be expressed in the form

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

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

General Two-Dimensional Fixed-Point Scaling



(a) Original Position of Object and Fixed Point

(b) Translate Object so that Fixed Point (x_f, y_f) is at Origin

(c) Scale Object with Respect to Origin

(d) Translate Object so that the Fixed Point is Returned to Position (x_f, y_f)

- ✓ To produce a two-dimensional scaling with respect to a selected fixed position ($x f$, yf), when 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.

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

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

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

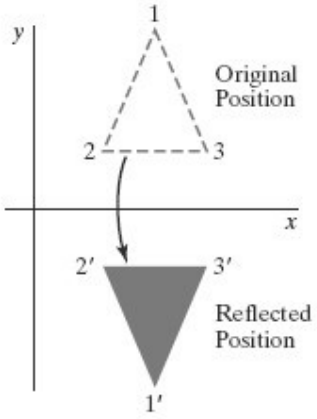
Other Two-Dimensional Transformations

Two such transformations

1. Reflection and
2. Shear.

Reflection

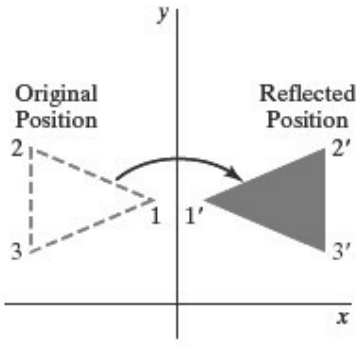
- ✓ A transformation that produces a mirror image of an object is called a **reflection.**
- ✓ For a two-dimensional reflection, this image is generated relative to an **axis of reflection** by rotating the object 180◦ about the reflection axis**.**
- ✓ Reflection about the line $y = 0$ (the x axis) is accomplished with the transformation Matrix $\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
- ✓ 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



- ✓ 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

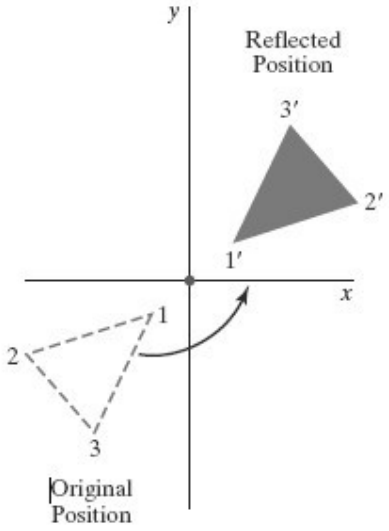
$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

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



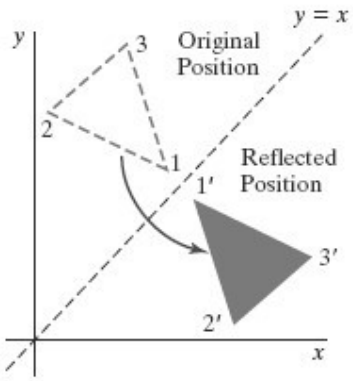
- ✓ We flip both the *x* and *y* coordinates of a point by reflecting relative to an axis that is perpendicular to the *xy* plane and that passes through the coordinate origin the matrix representation for this reflection is $\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

- ✓ An example of reflection about the origin is shown in Figure



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

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



- ✓ 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 ***Shear***

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

- ✓ A transformation that distorts the shape of an object such that the transformed shape appears as if the object were composed of internal layers that had been caused to slide over each other is called a **shear.**

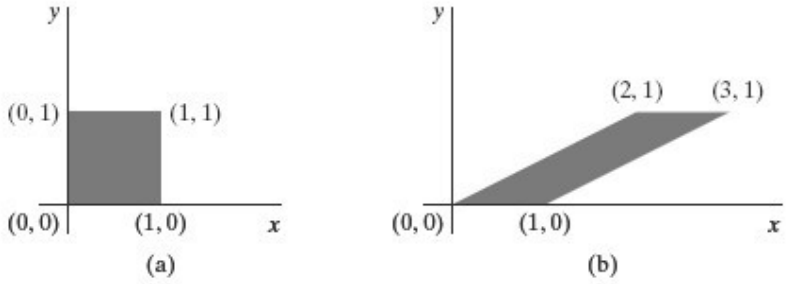
- ✓ Two common shearing transformations are those that shift coordinate *x* values and those that shift *y* values. An *x*-direction shear relative to the *x* axis is produced with the transformation Matrix

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

  which transforms coordinate positions as

$$x' = x + sh_x \cdot y, \qquad y' = y$$

- ✓ Any real number can be assigned to the shear parameter *shx* Setting parameter *shx* to the value 2, for example, changes the square into a parallelogram is shown below. Negative values for *shx* shift coordinate positions to the left.



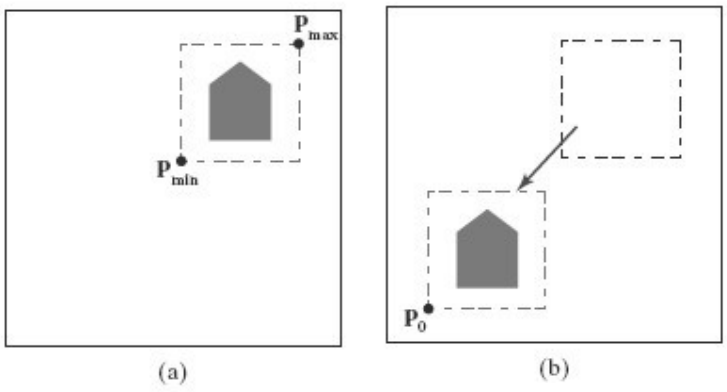
Raster Methods for Geometric Transformations

- ✓ 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
- ✓ Few arithmetic operations are needed, so the pixel transformations are particularly efficient.
- ✓ 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 below illustrates a two-dimensional translation implemented as a block transfer of

a refresh-buffer area



Translating an object from screen position (a) to the destination position shown in (b) by moving a rectangular block of pixel values. Coordinate positions Pmin and Pmax specify the limits of the rectangular block to be moved, and P0 is the destination reference position.

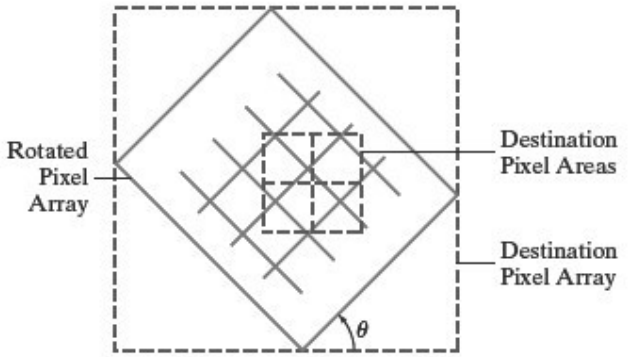
- ✓ Rotations in 90-degree increments are accomplished easily by rearranging the elements of a pixel array.
- ✓ We can rotate a two-dimensional object or pattern 90◦ counterclockwise by reversing the pixel values in each row of the array, then interchanging rows and columns.

- ✓ 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 below demonstrates the array manipulations that can be used to rotate a pixel block by 90◦ and by 180◦.
- ✓ For array rotations that are not multiples of 90◦, we need to do some extra processing.

- ✓ The general procedure is illustrated in Figure below.

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

(a) (b) (c)



- ✓ Each destination pixel area is mapped onto the rotated array and the amount of overlap with the rotated pixel areas is calculated.
- ✓ A color for a destination pixel can then be computed by averaging the colors of the overlapped source pixels, weighted by their percentage of area overlap.
- ✓ 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

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.

❖ 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.

❖ To rotate the color values, we rearrange the rows and columns of the color array, as described in the previous section. Then 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);**
- We can also combine raster transformations with logical operations to produce various effects with the *exclusive or* operator

OpenGL Functions for Two-Dimensional Geometric Transformations

- ✓ To perform a translation, we invoke the translation routine and set the components for the three-dimensional translation vector.
- ✓ In the rotation function, we specify the angle and the orientation for a rotation axis that intersects the coordinate origin.
- ✓ In addition, a scaling function is used to set the three coordinate scaling factors relative to the coordinate origin. In each case, the transformation routine sets up a 4 × 4 matrix that

  is applied to the coordinates of objects that are referenced after the transformation call

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.
- ✓ example: **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.
- ✓ If **v** is not specified as a unit vector, then it is normalized automatically before the elements of the rotation matrix are computed.
- ✓ The suffix code can be either **f** or **d**, and parameter **theta** is to be assigned a rotation angle in degree.
- ✓ For example, the statement: **glRotatef (90.0, 0.0, 0.0, 1.0);**

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

glScale* (sx, sy, sz);

- ✓ The suffix code is again either **f** or **d**, and the scaling parameters can be assigned any real-number values.
- ✓ 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 ✓ **Example: glScalef (2.0, -3.0, 1.0);**

OpenGL Matrix Operations

- ✓ The glMatrixMode routine is used to set the *projection mode which designates the matrix*

  that is to be used for the projection transformation.
- ✓ We specify the *modelview mode* with the statement

 which designates the 4×4 modelview matrix as the **current matrix**

glMatrixMode (GL_MODELVIEW);

- Two other modes that we can set with the **glMatrixMode** function are the *texture*

  mode and the *color mode*.

- The texture matrix is used for mapping texture patterns to surfaces, and the color matrix is used to convert from one color model to another.
- The default argument for the **glMatrixMode** function is **GL_MODELVIEW**.

✓ With the following function, we assign the identity matrix to the current matrix:

glLoadIdentity ( );

✓ Alternatively, we can assign other values to the elements of the current matrix using **glLoadMatrix* (elements16);**

✓ A single-subscripted, 16-element array of floating-point values is specified with parameter **elements16**, and a suffix code of either **f** or **d** is used to designate the data type

✓ The elements in this array must be specified in *column-major* order

✓ To illustrate this ordering, we initialize the modelview matrix with the following code: **glMatrixMode (GL_MODELVIEW);**

```
GLfloat elems [16];
GLint k;
for (k = 0; k < 16; k++) elems
        [k] = float (k);

glLoadMatrixf (elems);
```

Which produces the matrix

$$\mathbf{M} = \begin{bmatrix} 0.0 & 4.0 & 8.0 & 12.0 \\ 1.0 & 5.0 & 9.0 & 13.0 \\ 2.0 & 6.0 & 10.0 & 14.0 \\ 3.0 & 7.0 & 11.0 & 15.0 \end{bmatrix}$$

✓ We can also concatenate a specified matrix with the current matrix as follows:

glMultMatrix* (otherElements16);

- ✓ Again, the suffix code is either **f** or **d**, and parameter **otherElements16** is a 16-element, single-subscripted array that lists the elements of some other matrix in column-major order.
- ✓ Thus, assuming that the current matrix is the modelview matrix, which we designate as

M, then the updated modelview matrix is computed as

$$\mathbf{M} = \mathbf{M} \cdot \mathbf{M}'$$

- ✓ The **glMultMatrix** function can also be used to set up any transformation sequence with individually defined matrices.
- ✓ For example,

```
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ( ); // Set current matrix to the identity.
glMultMatrixf (elemsM2); // Postmultiply identity with matrix M2.
glMultMatrixf (elemsM1); // Postmultiply M2 with matrix M1.
```

produces the following current modelview

$$\mathbf{M} = \mathbf{M}2 \cdot \mathbf{M}1$$

3D Geometric Transformations: Translation, rotation, scaling, composite 3D transformations, other 3D transformations, OpenGL geometric transformations functions

1. Translation

Translation involves moving an object from one location to another in a 3D space. This is achieved by adding a translation vector (Tx,Ty,Tz) to the coordinates of the object.

- **Translation Matrix**:

$$T = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- **Translation Operation**:

$$P' = T \cdot P$$

where $P = (x, y, z, 1)$ is the original point and $P' = (x', y', z', 1)$ is the translated point.

2. Rotation

Rotation involves rotating an object about an axis (X, Y, or Z).

- **Rotation about X-axis**:

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

- **Rotation about Y-axis**:

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

- **Rotation about Z-axis**:

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

3. Scaling

Scaling changes the size of an object. Scaling can be uniform (same factor for all axes) or non-uniform (different factors for different axes).

- **Scaling Matrix**:

$$S = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- **Scaling Operation**:

$$P' = S \cdot P$$

Composite 3D Transformations

Composite transformations combine multiple transformations into one by multiplying their matrices in the specific order of application. This allows complex transformations to be performed efficiently.

- **Order of Multiplication**: The order in which transformations are applied is crucial. For example, if you want to scale an object and then rotate it, the transformation matrix would be

$$M = R \cdot S$$

If you apply scaling first and then rotation, this would yield a different result than rotating first and then scaling.

- **Example**: To perform a translation followed by a rotation and then a scaling, the composite transformation matrix would be:

$$M = T \cdot R \cdot S$$

Other 3D Transformations

- **Shearing**: Shearing distorts the shape of an object such that the transformed shape appears as if the object is skewed. The general shear matrix in 3D can be represented as

$$Sh = \begin{bmatrix} 1 & Sh_{xy} & Sh_{xz} & 0 \\ Sh_{yx} & 1 & Sh_{yz} & 0 \\ Sh_{zx} & Sh_{zy} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Where Sh_{xy} represents the shear factor in the xy-plane, and similarly for the others.

Reflection: Reflection flips an object over a plane. For example, reflection over the xy-plane can be represented as:

$$Ref_{xy} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

OpenGL Geometric Transformation Functions

OpenGL provides various functions to apply these transformations:

- **Translation**:

$$\text{glTranslatef}(T_x, T_y, T_z)$$

This function multiplies the current matrix by a translation matrix.

- **Rotation**:

$$\text{glRotatef}(\theta, x, y, z)$$

Rotates the object by θ degrees around the axis defined by the vector (x, y, z).

- **Scaling**:

$$\text{glScalef}(S_x, S_y, S_z)$$

Scales the object by S_x, S_y, and S_z along the x, y, and z axes respectively.

- **Matrix Operations**:
  - **Load Identity Matrix**

$$\text{glLoadIdentity}()$$

Resets the current matrix to the identity matrix.

  - **Multiply Current Matrix**:

$$\text{glMultMatrixf}(M)$$

Multiplies the current matrix by the matrix M.

- **Push and Pop Matrices**:

glPushMatrix()

Saves the current matrix on the stack.

glPopMatrix()

Restores the last saved matrix from the stack.