**2.2 2DGeometric Transformations:**

* + 1. **Basic 2D Geometric Transformations,**
    2. **Matrix representations and homogeneous coordinates.**
    3. **Inverse transformations,**
    4. **2DComposite transformations,**
    5. **Other 2D transformations,**
    6. **Raster methods for geometric transformations,**
    7. **OpenGL raster transformations**
    8. **OpenGL geometric transformations function,**

### Two-Dimensional Geometric Transformations

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

#### 2.2.1 those for translation, rotation, and scaling.

The geometric-transformation functions that are available in all graphics packages are

##### 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

**Code:**

**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 ( );**

**}**

* 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

are shown in Figure

# 

 The transformation equations for rotation of a point about any specified rotation position

(*xr* , *yr* ):



**Code:**

**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 ( );**

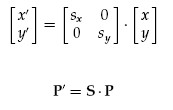
**}**

## Two-Dimensional Scaling

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

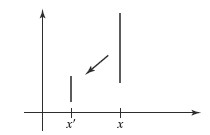
##  The basic two-dimensional scaling equations can also be written in the following matrix

form



Where **S** is the 2 × 2 scaling matrix

* 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.
* In some systems, negative values can also be specified for the scaling parameters. This not only resizes an object, it reflects it about one or more of the coordinate axes.
* Figure below illustrates scaling of a line by assigning the value 0.5 to both *sx* and *sy*



* 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* ) , are often chosen at some object position, such

as its centroid but any other spatial position can be selected.

* For a coordinate position (*x*, *y*), the scaled coordinates (*x’*, *y’*) are then calculated from

 the following relationships:

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



* Where the additive terms *x f (*1 − *sx)* and *yf (*1 − *sy)* are constants for all points in the

object.

**Code:**

**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 ( );**

**}**

### 2.2.2 Matrix Representations and Homogeneous Coordinates

* Each of the three basic two-dimensional transformations (translation, rotation, and

scaling) can be expressed in the general matrix form

 Matrix **M**1 is a 2 × 2 array containing multiplicative factors, and **M**2 is a two-element  With coordinate positions **P** and **P**’ represented as column vectors.

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.

#### 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

* 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 twodimensional coordinate-position representation (*x*, *y*) to a three-element representation (*xh*, *yh*, *h*),

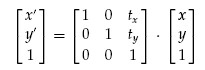
called **homogeneous coordinates,** where the **homogeneous** **parameter** *h* is a nonzero value such that



* A 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).
* The term *homogeneous coordinates* is used in mathematics to refer to the effect of this

representation on Cartesian equations.

#### Two-Dimensional Translation Matrix

* The homogeneous-coordinate for translation is given by
* This translation operation can be written in the abbreviated form

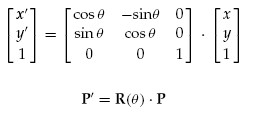


with **T**(*tx*, *ty*) as the 3 × 3 translation matrix

#### Two-Dimensional Rotation Matrix

* Two-dimensional rotation transformation equations about the coordinate origin can be

expressed in the matrix form

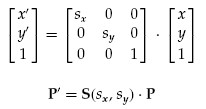


* The rotation transformation operator **R**(*θ* ) is the 3 × 3 matrix with rotation parameter *θ*.

#### Two-Dimensional Scaling Matrix

* A scaling transformation relative to the coordinate origin can now be expressed as the

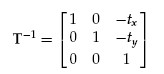
matrix multiplication

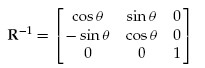


* The scaling operator S(sx, sy ) is the 3 × 3 matrix with parameters sx and sy

### 2.2.3 Inverse Transformations

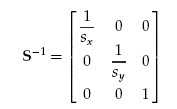
* For translation,we obtain the inverse matrix by negating the translation distances. Thus, if

we have two-dimensional translation distances *tx* and *ty*, the inverse translation matrix is 

 A two-dimensional rotation through an angle *θ* about the coordinate origin has the  An inverse rotation is accomplished by replacing the rotation angle by its negative.

inverse transformation matrix

* We form the inverse matrix for any scaling transformation by replacing the scaling parameters with their reciprocals. the inverse transformation matrix is



### 2.2.4 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



* 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 (*t*1*x*, *t*1*y*) and (*t*2*x*, *t*2*y*) are applied to a twodimensional coordinate position **P**, the final transformed location **P**’ is calculated as



where **P** and **P**’ are represented as three-element, homogeneous-coordinate

column vectors

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

### 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:

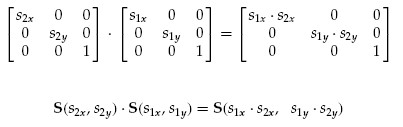
**R***(θ*2*)* · **R***(θ*1*)* = **R***(θ*1 + *θ*2*)*

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

**P’** = **R***(θ*1 + *θ*2*)* · **P**

### Composite Two-Dimensional Scalings

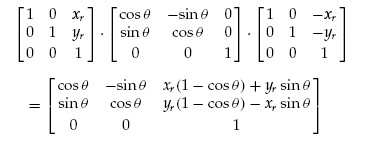
 Concatenating transformation matrices for two successive scaling operations in two dimensions produces the following composite scaling matrix



***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



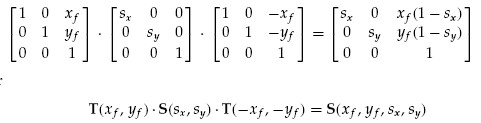
which can be expressed in the form 

where **T***(*−*xr* , −*yr )* = **T**−1*(xr* , *yr )*.

***General Two-Dimensional Fixed-Point Scaling***

# 

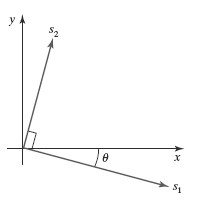
* To produce a two-dimensional scaling with respect to a selected fixed position (*x f* , *yf* ), when we have a function that can scale relative to the coordinate origin only. This sequence is
  1. Translate the object so that the fixed point coincides with the coordinate origin.
  2. Scale the object 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:

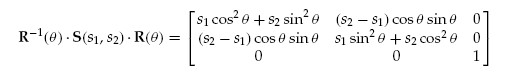


## General Two-Dimensional Scaling Directions

* Parameters *sx* and *sy* scale objects along the *x* and *y* directions.
* We can scale an object in other directions by rotating 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 *s*1 and *s*2

in the directions shown in Figure



* The composite matrix resulting from the product of these three transformations is

***Matrix Concatenation Properties Property 1:***

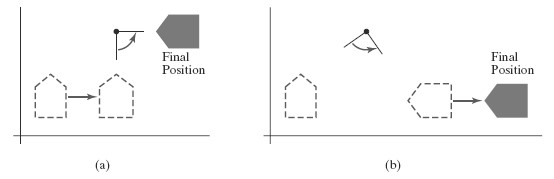
* Multiplication of matrices is associative.
* For any three matrices,**M**1,**M**2, and**M**3, the matrix product **M**3 · **M**2 · **M**1 can be performed by first multiplying **M**3 and **M**2 or by first multiplying**M**2 and **M**1:

**M**3 ·**M**2 ·**M**1 = *(***M**3 ·**M**2*)* ·**M**1 =  **M**3 · *(***M**2 ·**M**1*)*

* We can construct a composite matrix either by multiplying from left to right (premultiplying) or by multiplying from right to left (postmultiplying)

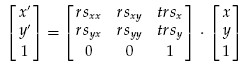
***Property 2:***

* Transformation products, on the other hand, may not be commutative. The matrix product**M**2 ·**M**1 is not equal to**M**1 ·**M**2, in general.
* 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

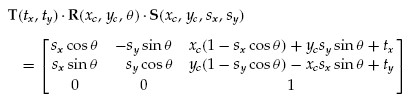


* Reversing the order in which a sequence of transformations is performed may affect the transformed position of an object. In (a), an object is first translated in the x direction, then rotated counterclockwise through an angle of 45◦. In (b), the object is first rotated 45◦ counterclockwise, then translated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

***General Two-Dimensional Composite Transformations and Computational Efficiency***  A two-dimensional transformation, representing any combination of translations,

 rotations, and scalings, can be expressed as

* The four elements *rsjk* are the multiplicative rotation-scaling terms in the transformation, which involve only rotation angles and scaling factors if an object is to be scaled and rotated about its centroid coordinates (*xc* , *yc* ) and then translated, the values for the elements of the composite transformation matrix are



* Although the above matrix requires nine multiplications and six additions, the explicit calculations for the transformed coordinates are



* We need actually perform only four multiplications and four additions to transform coordinate positions.
* Because rotation calculations require trigonometric evaluations and several multiplications for each transformed point, computational efficiency can become an important consideration in rotation transformations
* If we are rotating in small angular steps about the origin, for instance, we can set cos *θ* to 1.0 and reduce transformation calculations at each step to two multiplications and two

additions for each set of coordinates to be rotated.

* These rotation calculations are

*x’*= *x* − *y* sin *θ*, *y*’ = *x* sin *θ* + *y*

## Two-Dimensional Rigid-Body Transformation

* If a transformation matrix includes only translation and rotation parameters, it is a **rigid-**

**body transformation matrix.**

* The general form for a two-dimensional rigid-body transformation matrix is

# 

where the four elements *r jk* are the multiplicative rotation terms, and the elements *trx*

and *try* are the translational terms

* A rigid-body change in coordinate position is also sometimes referred to as a **rigidmotion** transformation.
* In addition, the above matrix has the property that its upper-left 2 × 2 submatrix is an

*orthogonal matrix*.

* If we consider each row (or each column) of the submatrix as a vector, then the two row vectors (*rxx*, *rxy*) and (*ryx*, *ryy*) (or the two column vectors) form an orthogonal set of unit vectors.

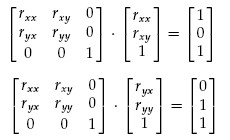
* Such a set of vectors is also referred to as an *orthonormal* vector set. Each vector has unit length as follows



and the vectors are perpendicular (their dot product is 0):

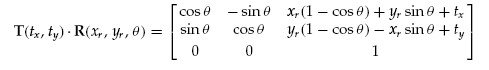


* Therefore, if these unit vectors are transformed by the rotation submatrix, then the vector (*rxx*, *rxy*) is converted to a unit vector along the *x* axis and the vector (*ryx*, *ryy*) is transformed into a unit vector along the *y*  axis of the coordinate system

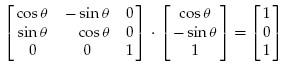


* For example, the following rigid-body transformation first rotates an object through an

angle *θ* about a pivot point (*xr* , *yr* ) and then translates the object



* Here, orthogonal unit vectors in the upper-left 2×2 submatrix are (cos *θ*, −sin *θ*) and (sin

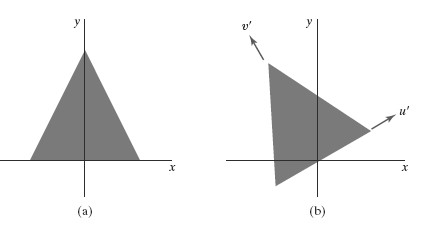
 *θ*, cos *θ*).

## Constructing Two-Dimensional Rotation Matrices

* The orthogonal property of rotation matrices is useful for constructing the matrix when we know the final orientation of an object, rather than the amount of angular rotation necessary to put the object into that position.
* We might want to rotate an object to align its axis of symmetry with the viewing (camera) direction, or we might want to rotate one object so that it is above another

object.

* Figure shows an object that is to be aligned with the unit direction vectors **u**\_ and **v**



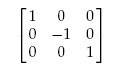
The rotation matrix for revolving an object from position (a) to position (b) can be constructed with the values of the unit orientation vectors u’ and v ’ relative to the original orientation.

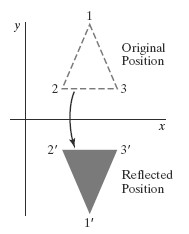
## 2.2.5 Other Two-Dimensional Transformations

Two such transformations

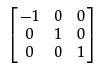
1. Reflection and
2. Shear.

# ***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 transformationMatrix
* 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



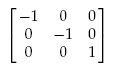
* Figure below illustrates the change in position of an object that has been reflected about

the line *x* = 0.

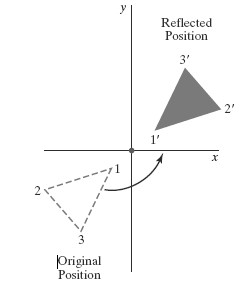
# 

* 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



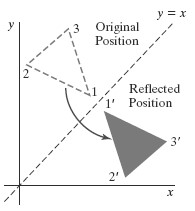
* An example of reflection about the origin is shown in Figure



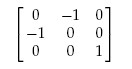
* If we choose the reflection axis as the diagonal line *y* = *x* (Figure below), the reflection

matrix is

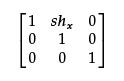
# 



* 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◦,

* 1. reflection about the *y* axis, and
  2. counterclockwise rotation by 45◦. The resulting transformation matrix is ***Shear***
* 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



which transforms coordinate positions as

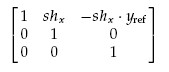


* Any real number can be assigned to the shear parameter *shx* Setting parameter *shx* to the value 2, for example,changes the square into a parallelogram is shown below. Negative values for *shx* shift coordinate positions to the left.

# 

A unit square (a) is converted to a parallelogram (b) using the x -direction shear with shx = 2.

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

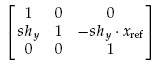


Now, coordinate positions are transformed as



* A *y*-direction shear relative to the line *x* = *x*ref is generated with the transformation

Matrix



which generates the transformed coordinate values



## 2.2.6 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

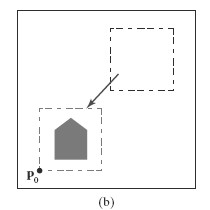
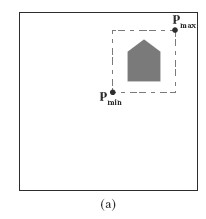
efficient.

* 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

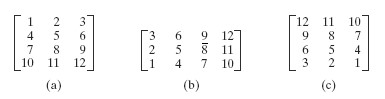
*bitblt,* or a *pixblt*.

* 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.

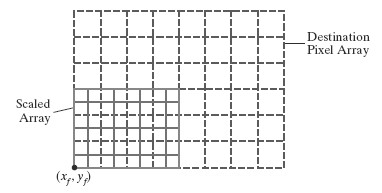
* Rotations in 90-degree increments are accomplished easily by rearranging the elements of a pixel array.
* We can rotate a two-dimensional object or pattern 90◦ counterclockwise by reversing the pixel values in each row of the array, then interchanging rows and columns.
* A 180◦ rotation is obtained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

# 

* 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



## 2.2.7 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

### 2.2.8 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** functionare the *texture*

*mode* and the *color mode*.

* + The texture matrix is used for mappingtexture patterns to surfaces, and the color matrix is used to convert from one color model to another.
  + The default argument for the **glMatrixMode** functionis **GL\_MODELVIEW**.
* With the following function, we assign the identity matrix to the current matrix:

**glLoadIdentity ( );**

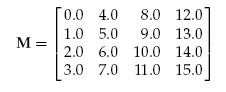
* Alternatively, we can assign other values to the elements of the current matrix using **glLoadMatrix\* (elements16);**
* A single-subscripted, 16-element array of floating-point values is specified with parameter **elements16**, and a suffix code of either **f** or **d** is used to designate the data 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



* 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

**M** = **M**·**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 matrix:

**M** = **M**2 ·**M**1

### 2.3 Two Dimensional Viewing

2.3.1 2D viewing pipeline

2.3.1 OpenGL 2D viewing functions.

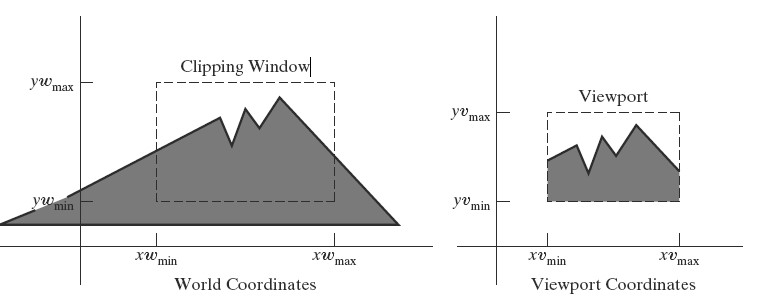
#### 2.3.1 The Two-Dimensional Viewing Pipeline

* A section of a two-dimensional scene that is selected for display is called a clipping Window.
* Sometimes the clipping window is alluded to as the *world* *window* or the *viewing window*
* Graphics packages allow us also to control the placement within the display window using another “window” called the **viewport**.
* The clipping window selects *what* we want to see; the viewport indicates *where* it is to be

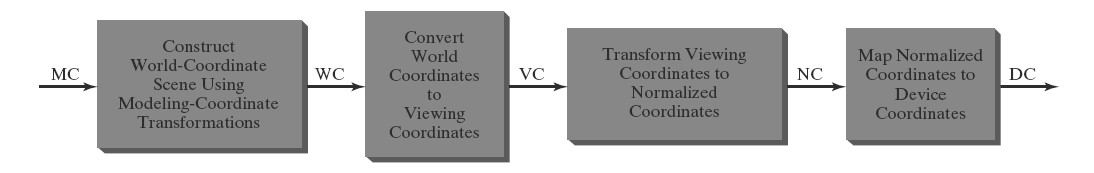
viewed on the output device.

* By changing the position of a viewport, we can view objects at different positions on the display area of an output device
* Usually, clipping windows and viewports are rectangles in standard position, with the rectangle edges parallel to the coordinate axes.
* We first consider only rectangular viewports and clipping windows, as illustrated in

Figure



##### Viewing Pipeline

* The mapping of a two-dimensional, world-coordinate scene description to device coordinates is called a **two-dimensional viewing transformation**.
* This transformation is simply referred to as the *window-to-viewport transformation* or the *windowing transformation*
* We can describe the steps for two-dimensional viewing as indicated in Figure
* Once aworld-coordinate scene has been constructed,wecould set up a separate two-

dimensional, **viewingcoordinate** **reference frame** for specifying the clipping window.

* 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.
* Systems use normalized coordinates in the range from 0 to 1, and others use a normalized

 At the final step of the viewing transformation, the contents of the viewport are range from −1 to 1. 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

#### 2.3.2 OpenGL Two-Dimensional Viewing Functions

 The GLU library provides a function for specifying a two-dimensional clipping window, and we have GLUT library functions for handling display windows.

##### 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.
* We must set the parameters for the clipping window as part of the projection transformation.
* Function:

**glMatrixMode (GL\_PROJECTION);**

* We can also set the initialization as

**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

##### 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 the

orthogonal projection has no effect on our two-dimensional scene other than to convert object positions to normalized coordinates.

 Normalized coordinates in the range from −1 to 1 are used in the OpenGL clipping

routines.

* Objects outside the normalized square (and outside the clipping window) are eliminated from the scene to be displayed.
* If we do not specify a clipping window in an application program, the default coordinates are (*xw*min, *yw*min*)* = *(*−1*.*0, −1*.*0*)* and (*xw* max, *yw*max*)* = *(*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.

##### OpenGL Viewport Function

* We specify the viewport parameters with the OpenGL function

**glViewport (xvmin, yvmin, vpWidth, vpHeight);**

Where,

* + **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
* Coordinates for the upper-right corner of the viewport are calculated for this transformation matrix in terms of the viewport width and height:



* Multiple viewports can be created in OpenGL for a variety of applications.
* We can obtain the parameters for the currently active viewport using the query function

**glGetIntegerv (GL\_VIEWPORT, vpArray);**  where,  **vpArray** is a single-subscript, four-element array.

##### Creating a GLUT Display Window

* The GLUT library interfaces with any window-management system, we use the GLUT routines for creating and manipulating display windows so that our example programs will be independent of any specific machine.
* We first need to initialize GLUT with the following function:

**glutInit (&argc, argv);**

##  We have three functions inGLUTfor defining a display window and choosing its

dimensions and position:

**1. glutInitWindowPosition (xTopLeft, yTopLeft);**

gives the integer, screen-coordinate position for the top-left corner of the display window, relative to the top-left corner of the screen

**2. glutInitWindowSize (dwWidth, dwHeight);**

* we choose a width and height for the display window in positive integer pixel

dimensions.

* If we do not use these two functions to specify a size and position, the default size is 300 by 300 and the default position is *(*−1, −1*)*, which leaves the positioning of the display window to the window-management system

**3. glutCreateWindow ("Title of Display Window");**

creates the display window, with the specified size and position, and assigns a title, although the use of the title also depends on the windowing system

### Setting the GLUT Display-Window Mode and Color

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

1. **glutInitDisplayMode (mode);** 
   * We use this function to choose a color mode (RGB or index) and different buffer combinations, and the selected parameters are combined with the logical **or** operation.

1. **glutInitDisplayMode (GLUT\_SINGLE | GLUT\_RGB);** 
   * The color mode specification **GLUT\_RGB**  is equivalent to **GLUT\_RGBA.**

1. **glClearColor (red, green, blue, alpha);** 
   * A backgroundcolor for the display window is chosen in RGB mode with the OpenGL

# routine

**4. glClearIndex (index);**

* This function sets the display window color using color-index mode,
* Where parameter **index** is assigned an integer value corresponding to a position within the color table.

## 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.
* Function:

**windowID = glutCreateWindow ("A Display Window");**

**Deleting a GLUT Display Window**

* If we know the display window’s identifier, we can eliminate it with the statement

**glutDestroyWindow (windowID);**

**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);**

* 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

## Relocating and Resizing a GLUT Display Window

* We can reset the screen location for the current display window with the function  Similarly, the following function resets the size of the current display window:

**glutPositionWindow (xNewTopLeft, yNewTopLeft);**

**glutReshapeWindow (dwNewWidth, dwNewHeight);**

* With the following command, we can expand the current display window to fill the

screen:

**glutFullScreen ( );**

* 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);**

## Managing Multiple GLUT Display Windows

* The GLUT library also has a number of routines for manipulating a display window in various ways.
* We use the following routine 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");**

* We can 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 ( );**

* In a similar way, 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 ( );**

* In addition, 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 ( );**

## GLUT Subwindows

* Within a selected display window, we can set up any number of second-level display windows, which are called *subwindows.*
* 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.
* Subwindows are assigned a positive integer identifier in the same way that first-level display windows are numbered, and we can place a subwindow inside another

subwindow.

* Each subwindow can be assigned an individual display mode and other parameters. We can even reshape, reposition, push, pop, hide, and show subwindows

## 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);** where, shape can be

* **GLUT\_CURSOR\_UP\_DOWN :** an up-down arrow.  **GLUT\_CURSOR\_CYCLE:** A rotating arrow is chosen  **GLUT\_CURSOR\_WAIT:** a wristwatch shape.
* **GLUT\_CURSOR\_DESTROY:** a skull and crossbones

***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);**

* This routine, 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.
* We may need to call **glutDisplayFunc** after the  **glutPopWindow** command if the display window has been damaged during the process of redisplaying the windows.
* In this case, the following function is used to indicate that the contents of the current display window should be renewed:

**glutPostRedisplay ( );**

### 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 ( );**

### Other GLUT Functions

* Sometimes it is convenient to designate a function that is to be executed when there are no other events for the system to process. We can do that with

**glutIdleFunc (function);**

* 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, for the stateParam we can have the values

 **GLUT\_WINDOW\_X**: obtains the *x*-coordinate position for the top-left corner of the

current display window

 **GLUT\_WINDOW\_WIDTH** or **GLUT\_SCREEN\_WIDTH** : retrieve the current

display-window width or the screen width with**.**