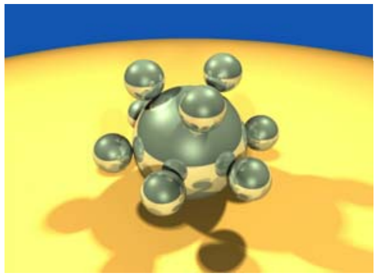
Computer Graphics using OpenGL, 3rd Edition
F. S. Hill, Jr. and S. Kelley



Chapter 5.1-2
Transformations of Objects

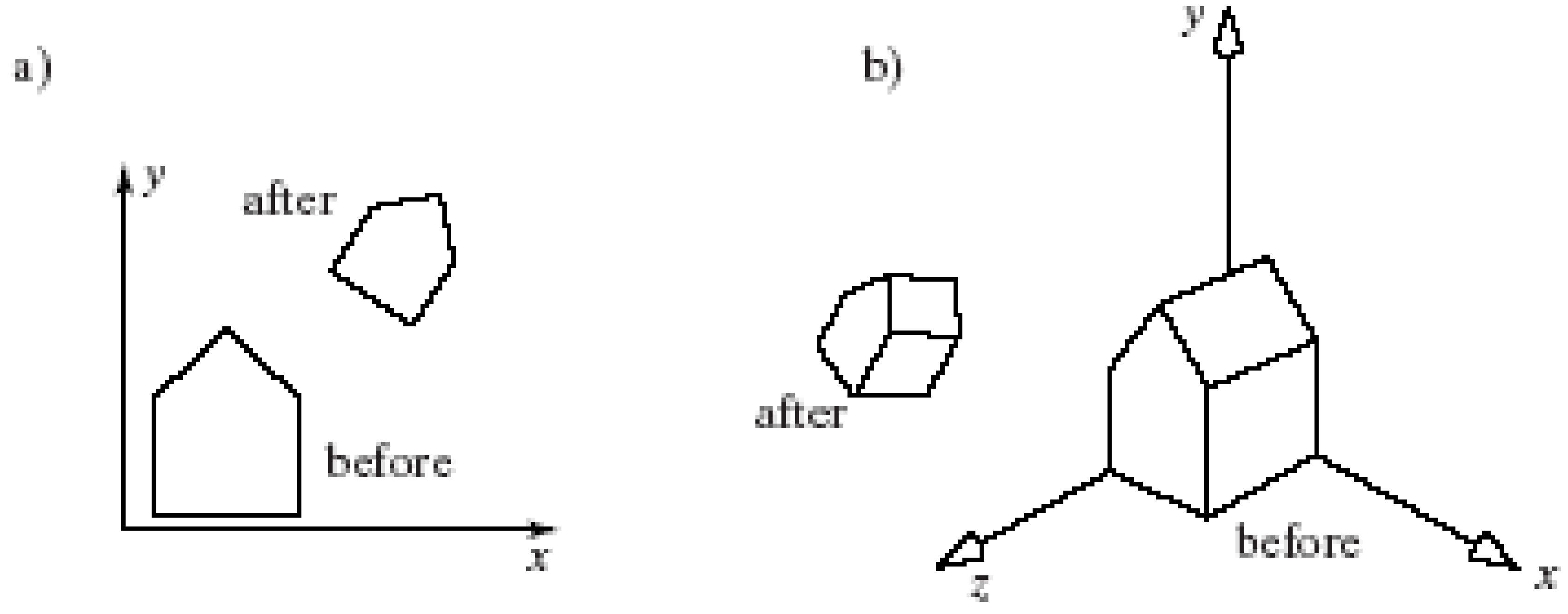
S. M. Lea
University of North Carolina at Greensboro

Transformations

- We used the window to viewport transformation to scale and translate objects in the world window to their size and position in the viewport.
- We want to build on this idea, and gain more flexible control over the size, orientation, and position of objects of interest.
- To do so, we will use the powerful **affine transformation**.

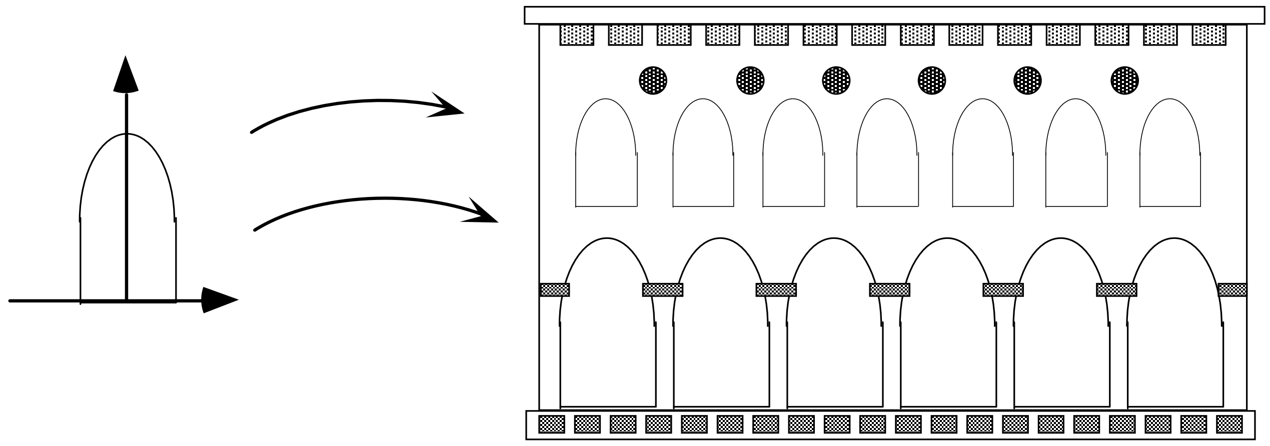
Example of Affine Transformations

- The house has been scaled, rotated and translated, in both 2D and 3D.



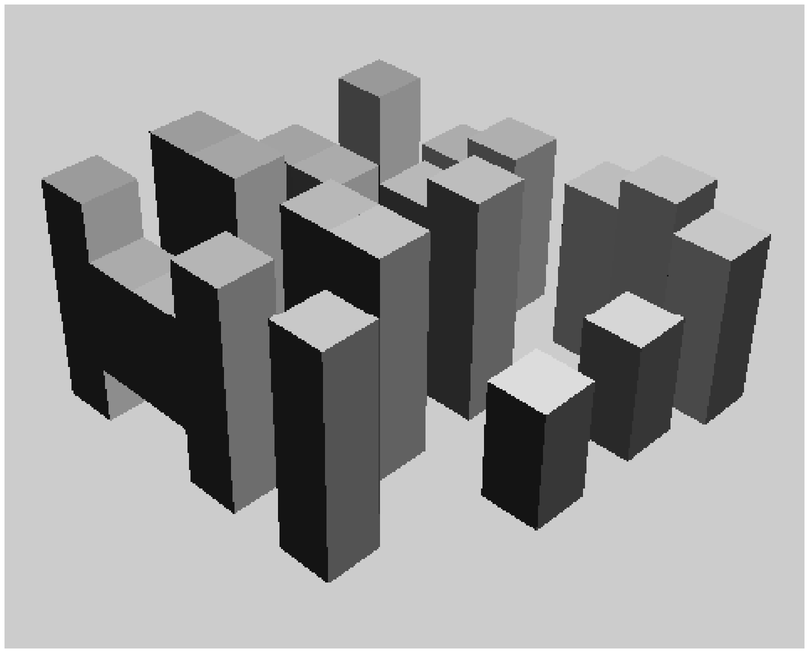
Using Transformations

- The arch is designed in its own coordinate system.
- The scene is drawn by placing a number of instances of the arch at different places and with different sizes.



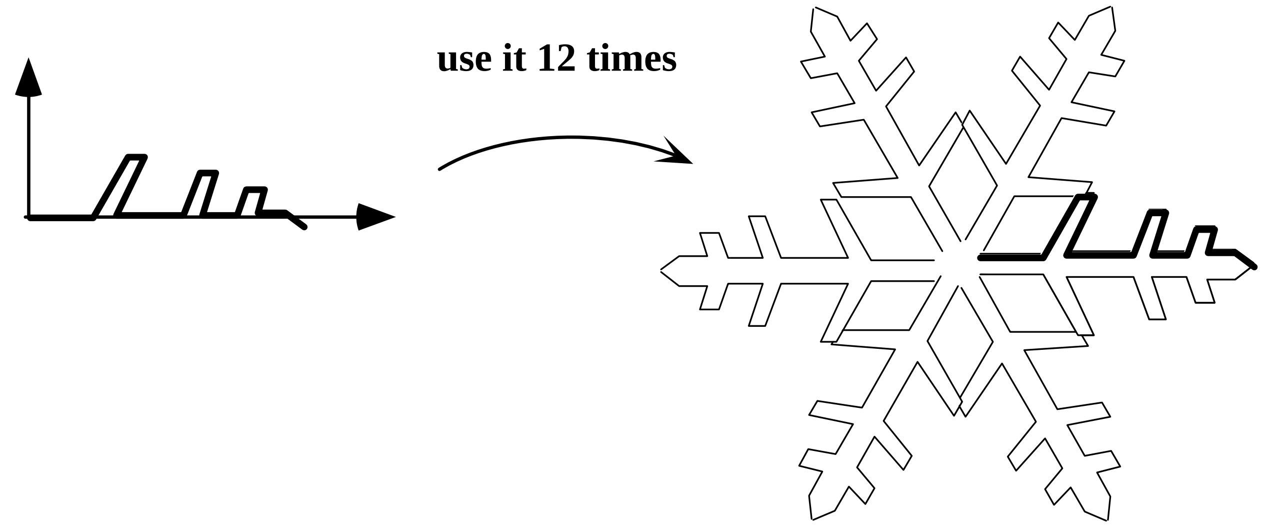
Using Transformations (2)

- In 3D, many cubes make a city.



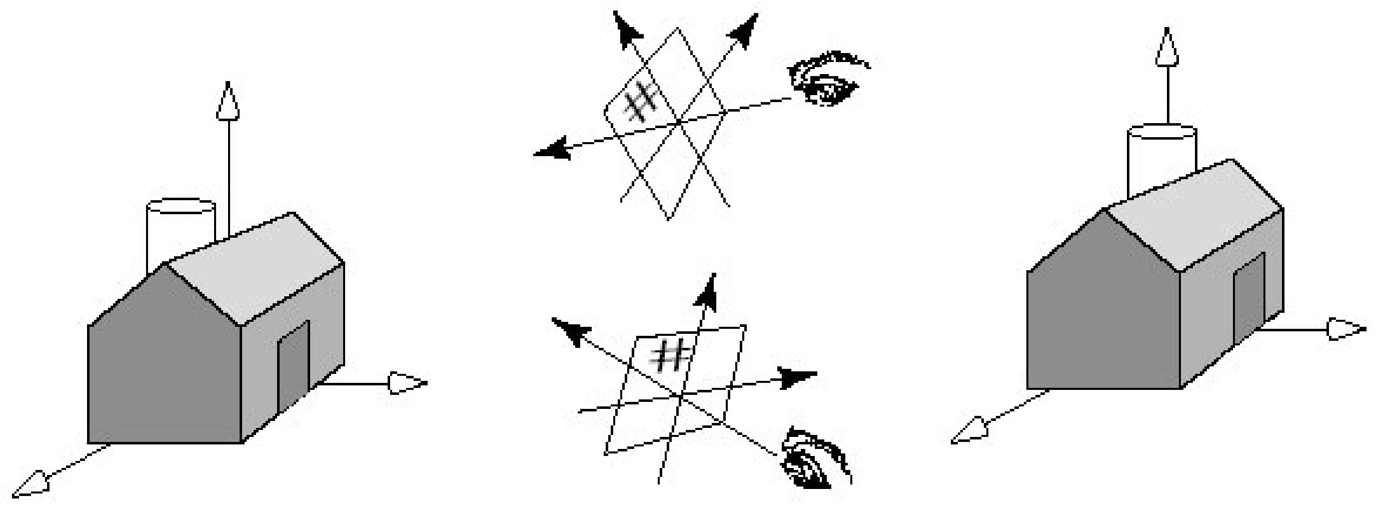
Using Transformations (3)

- The snowflake exhibits symmetries.
- We design a single **motif** and draw the whole shape using appropriate reflections, rotations, and translations of the motif.



Using Transformations (4)

- A designer may want to view an object from different vantage points.
- Positioning and reorienting a camera can be carried out through the use of 3D affine transformations.

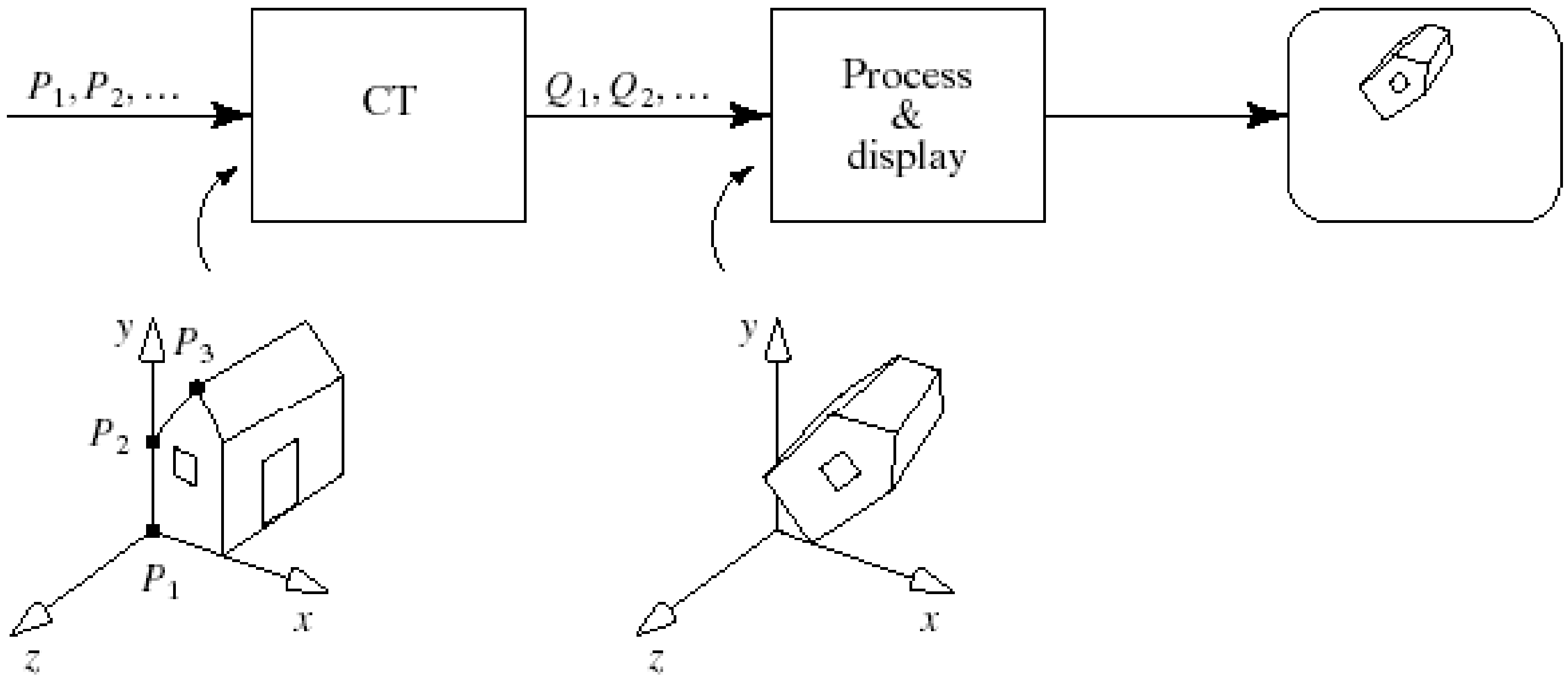


Using Transformations (5)

- In a computer animation, objects move.
- We make them move by translating and rotating their local coordinate systems as the animation proceeds.
- A number of graphics platforms, including OpenGL, provide a graphics pipeline: a sequence of operations which are applied to all points that are sent through it.
- A drawing is produced by processing each point.

The OpenGL Graphics Pipeline

- This version is simplified.



Graphics Pipeline (2)

- An application sends the pipeline a sequence of points P_1, P_2, ... using commands such as:

```
glBegin(GL_LINES);
   glVertex3f(...); // send P1 through the pipeline
   glVertex3f(...); // send P2 through the pipeline
   ...
glEnd();
```

- These points first encounter a transformation called **the current transformation** (CT), which alters their values into a different set of points, say Q_1, Q_2, Q_3.

Graphics Pipeline (3)

- Just as the original points P_i describe some geometric object, the points Q_i describe the transformed version of the same object.
- These points are then sent through additional steps, and ultimately are used to draw the final image on the display.

Graphics Pipeline (4)

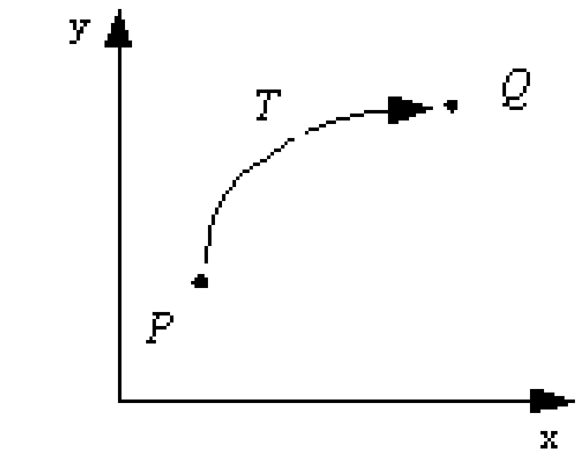
- Prior to OpenGL 2.0 the pipeline was of *fixed-functionality:* each stage had to perform a specific operation in a particular manner.
- With OpenGL 2.0 and the Shading Language (GLSL), the application programmer could not only change the order in which some operations were performed, but in addition could make the operations **programmable**.
- This allows hardware and software developers to take advantage of new algorithms and rendering techniques and still comply with OpenGL version 2.0.

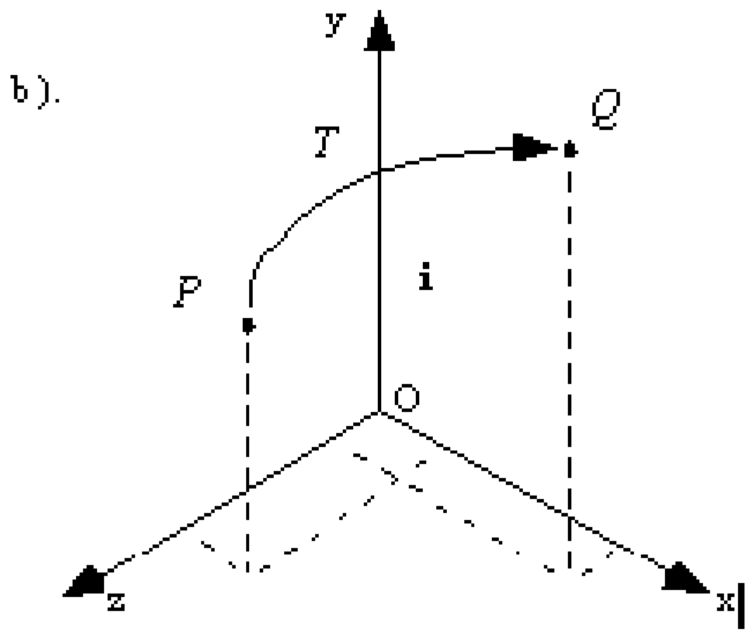
Transformations

- Transformations change 2D or 3D points and vectors, or change coordinate systems.
  - An object transformation alters the coordinates of each point on the object according to the same rule, leaving the underlying coordinate system fixed.
  - A coordinate transformation defines a new coordinate system in terms of the old one, then represents all of the object's points in this new system.
- Object transformations are easier to understand, so we will do them first.

Transformations (2)

- A (2D or 3D) transformation *T*( ) alters each point, *P* into a new point, *Q*, using a specific formula or algorithm: *Q*= *T*(*P*).





Transformations (3)

- An arbitrary point *P* in the plane is **mapped** to *Q*.
- *Q* is the **image** of *P* under the mapping *T*.
- We transform an object by transforming each of its points, using the *same* function *T*() for each point.
- The **image** of line *L* under *T*, for instance, consists of the images of *all* the individual points of L.

Transformations (4)

- Most mappings of interest are continuous, so the image of a straight line is still a connected curve of some shape, although it’s not necessarily a straight line.
- Affine transformations, however, *do* preserve lines: the image under *T* of a straight line is also a straight line.

Transformations (5)

- We use an explicit coordinate frame when performing transformations.
- A coordinate frame consists of a point $\mathcal{O}$, called the **origin**, and some mutually perpendicular vectors (called **i** and **j** in the 2D case; **i, j,** and **k** in the 3D case) that serve as the axes of the coordinate frame.
- In 2D,

$$\tilde{P} = \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}, \tilde{Q} = \begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix}$$

Transformations (6)

- Recall that this means that point $\mathscr{P}$ is at location $= \mathscr{P}_x\, \mathbf{i} + \mathscr{P}_y\, \mathbf{j} + \mathscr{O}$, and similarly for $\mathscr{Q}$.
- $\mathscr{P}_x$ and $\mathscr{P}_y$ are the coordinates of $\mathscr{P}$.
- To get from the origin to point $\mathscr{P}$, move amount $\mathscr{P}_x$ along axis **i** and amount $\mathscr{P}_y$ along axis **j**.

Transformations (7)

- Suppose that transformation *T* operates on any point $\mathscr{P}$ to produce point $\mathscr{Q}$:
- $\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = T\left(\begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}\right)$ or $\mathscr{Q} = \mathrm{T}(\mathscr{P})$.
- T may be any transformation: e.g.,

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(P_x)e^{-P_x} \\ \dfrac{\ln(P_y)}{1+P_x^2} \\ 1 \end{pmatrix}$$

Transformations (8)

- To make **affine** transformations we restrict ourselves to much simpler families of functions, those that are *linear* in P_x and P_y.
- Affine transformations make it easy to scale, rotate, and reposition figures.
- Successive affine transformations can be combined into a single overall affine transformation.

Affine Transformations

- Affine transformations have a compact matrix representation.
- The matrix associated with an affine transformation operating on 2D vectors or points must be a three-by-three matrix.
  - This is a direct consequence of representing the vectors and points in homogeneous coordinates.

Affine Transformations (2)

- Affine transformations have a simple form.
- Because the coordinates of $\mathcal{Q}$ are *linear* combinations of those of $\mathcal{P}$, the transformed point may be written in the form:

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} m_{11}P_x + m_{12}P_y + m_{13} \\ m_{21}P_x + m_{22}P_y + m_{23} \\ 1 \end{pmatrix}$$

Affine Transformations (3)

- There are six given constants: m_{11}, m_{12}, etc.
- The coordinate Q_x consists of portions of both P_x and P_y, and so does Q_y.
- This *combination* between the *x*- and *y*-components also gives rise to rotations and shears.

Affine Transformations (4)

- Matrix form of the affine transformation in 2D:

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}$$

- For a 2D affine transformation the third row of the matrix is always (0, 0, 1).

Affine Transformations (5)

- Some people prefer to use row matrices to represent points and vectors rather than column matrices: e.g., $P = (P_x, P_y, 1)$
- In this case, the *P* vector must *pre-multiply* the matrix, and the transpose of the matrix must be used: $Q = P M^T$.

$$M^T = \begin{pmatrix} m_{11} & m_{21} & 0 \\ m_{12} & m_{22} & 0 \\ m_{13} & m_{23} & 1 \end{pmatrix}$$

Affine Transformations (6)

- Vectors can be transformed as well as points.
- If a 2D vector **v** has coordinates V_x and V_y then its coordinate frame representation is a column vector with third component 0.

Affine Transformations (7)

- When vector **V** is transformed by the same affine transformation as point P, the result is $$\begin{pmatrix} W_x \\ W_y \\ 0 \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} V_x \\ V_y \\ 0 \end{pmatrix}$$
- Important: to transform a point *P* into a point *Q*, *post-multiply M* by *P*: Q = M P.

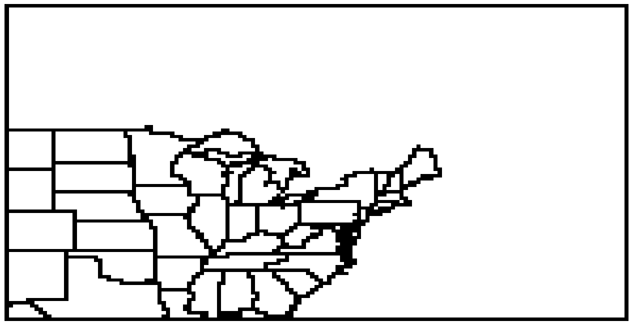
Affine Transformations (8)

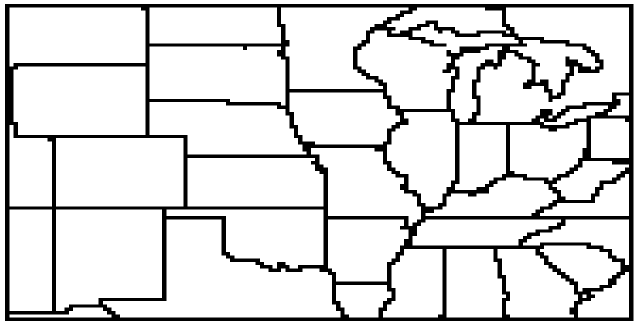
- Example: find the image Q of point P = (1, 2, 1) using the affine transformation

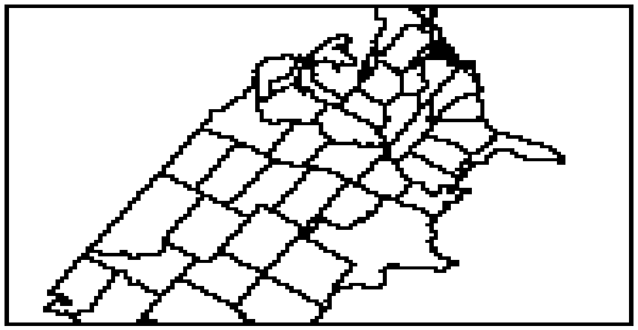
$$M = \begin{pmatrix} 3 & 0 & 5 \\ -2 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix}; Q = \begin{pmatrix} 8 \\ 2 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 & 0 & 5 \\ -2 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}$$

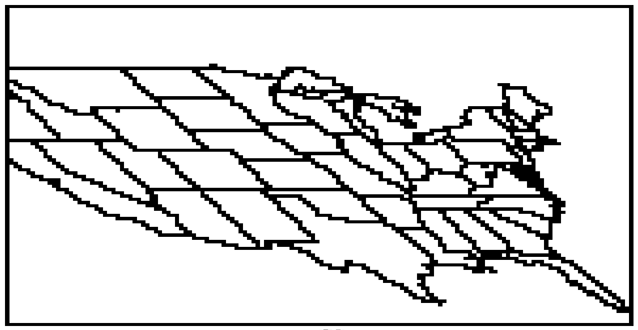
Geometric Effects of Affine Transformations

- Combinations of four elementary transformations: (a) a translation, (b) a scaling, (c) a rotation, and (d) a shear (all shown below).


a)


b)


c)


d)

Translations

- The amount *P* is translated does not depend on P's position.
- It is meaningless to translate vectors.
- To translate a point P by a in the x direction and b in the y direction use the matrix:

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix} = \begin{pmatrix} Q_x + a \\ Q_y + b \\ 1 \end{pmatrix}$$

- Only using homogeneous coordinates allow us to include translation as an affine transformation.

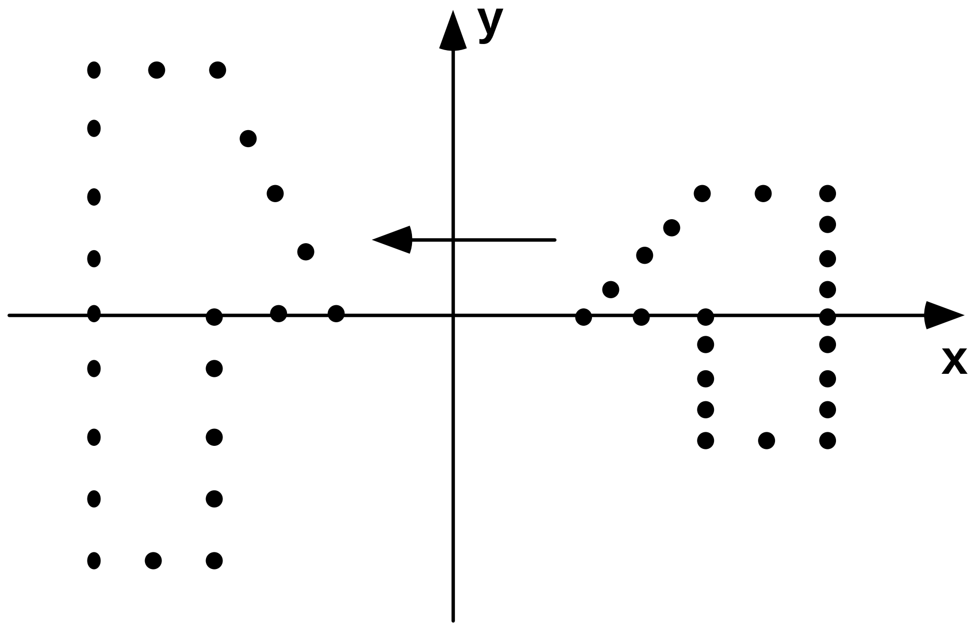
Scaling

- Scaling is about the origin. If $S_x = S_y$ the scaling is uniform; otherwise it distorts the image.
- If S_x or $S_y < 0$, the image is reflected across the x or y axis.
- The matrix form is

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}$$

Example of Scaling

- The scaling (*S*x, *S*y) = (-1, 2) is applied to a collection of points. Each point is both reflected about the *y*-axis and scaled by 2 in the *y*-direction.



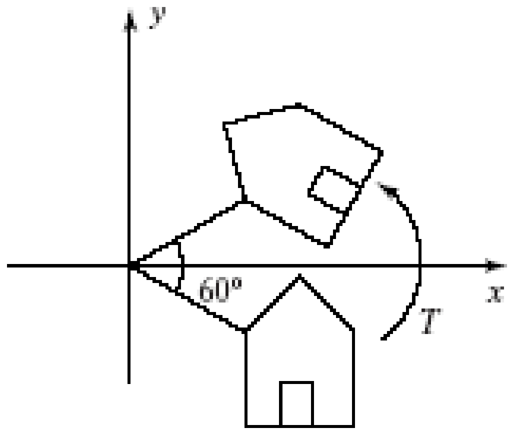
Types of Scaling

- Pure reflections, for which each of the scale factors is +1 or -1.
- A **uniform scaling**, or a magnification about the origin: $S_x = S_y$, magnification $|S|$.
  - Reflection also occurs if S_x or S_y is negative.
  - If $|S| < 1$, the points will be moved closer to the origin, producing a reduced image.
- If the scale factors are not the same, the scaling is called a **differential scaling**.

Rotation

- Counterclockwise around origin by angle θ:

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}$$



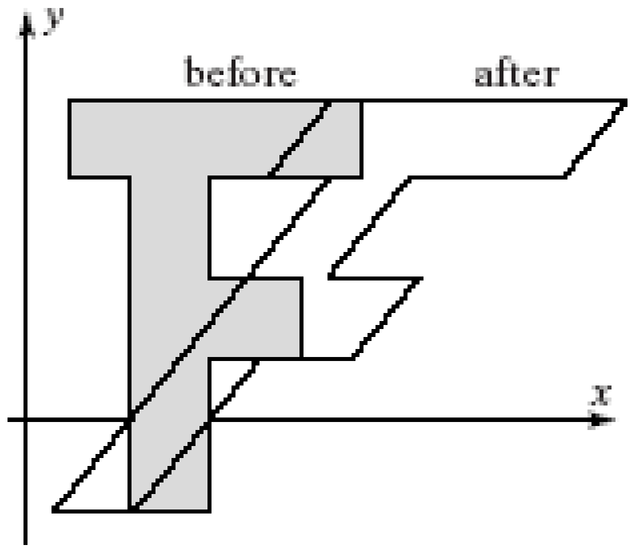
Deriving the Rotation Matrix

- P is at distance R from the origin, at angle Φ; then $P = (R\ cos(\Phi), R\ sin(\Phi))$.
- Q must be at the same distance as P, and at angle $\theta + \Phi$: $Q = (\mathrm{R}\ \mathrm{cos}(\theta + \Phi), \mathrm{R}\ \mathrm{sin}(\theta + \Phi))$.
- $cos(\theta + \Phi) = cos(\theta)\ cos(\Phi) - sin(\theta)\ sin(\Phi)$; $sin(\theta + \Phi) = sin(\theta)\ cos(\Phi) + cos(\theta)\ sin(\Phi)$.
- Use $P_x = R\ cos(\Phi)$ and $P_y = R\ sin(\Phi)$.

Shear

- Shear H about origin: x depends linearly on y in the figure.
- Shear along x: h ≠ 0, and P_x depends on P_y (for example, *italic* letters).
- Shear along y: g ≠ 0, and P_y depends on P_x.

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & h & 0 \\ g & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}$$



Inverses of Affine Transformations

- det(M) = $m_{11}*m_{22} - m_{21}*m_{12}$⑤ 0 means that the inverse of a transformation exists.
  - That is, the transformation can be "undone“.
- $M\ M^{-1} = M^{-1}M = I$, the identity matrix (ones down the major diagonal and zeroes elsewhere).

Inverse Translation and Scaling

- Inverse of translation T^{-1}:

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & -t_x \\ 0 & 1 & -t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}$$

- Inverse of scaling S^{-1}:

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1/S_x & 0 & 0 \\ 0 & 1/S_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}$$

Inverse Rotation and Shear

- Inverse of rotation $R^{-1} = R(-\theta)$:

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}$$

- Inverse of shear H^{-1}: generally h=0 or g=0.

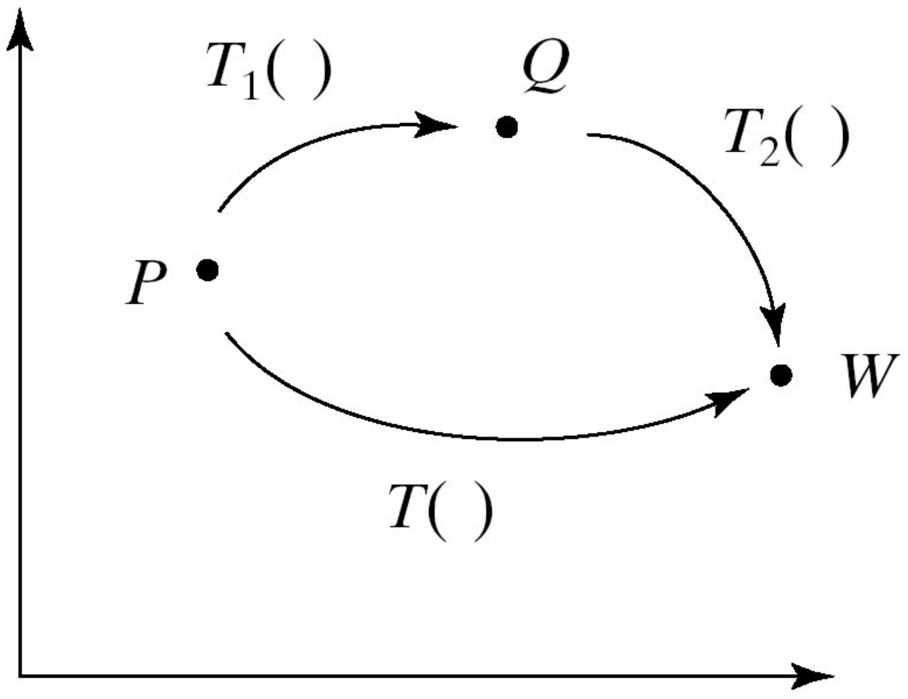
$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & -h & 0 \\ -g & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix} \frac{1}{1-gh}$$

Composing Affine Transformations

- Usually, we want to apply several affine transformations in a particular order to the figures in a scene: for example,
  - translate by (3, - 4)
  - then rotate by 30^o
  - then scale by (2, - 1) and so on.
- Applying successive affine transformations is called **composing** affine transformations.

Composing Affine Transformations (2)

- $T_1(\ )$ maps P into Q, and $T_2(\ )$ maps Q into point W. Is $W = T_2(Q) = T_2(T_1(P))$affine?
- Let $T_1=M_1$ and $T_2=M_2$, where M_1 and M_2 are the appropriate matrices.
- $W = M_2(M_1P)) = (M_2M_1)P = MP$ by associativity.
- So $M = M_2M_1$, the product of 2 matrices (in reverse order of application), which is affine.



Composing Affine Transformations: Examples

- To rotate around an arbitrary point: translate P to the origin, rotate, translate P back to original position. $Q = T_P R T_{-P} P$
- Shear around an arbitrary point: $Q = T_P H T_{-P} P$
- Scale about an arbitrary point:

  $Q = T_P S T_{-P} P$

Composing Affine Transformations (Examples)

- Reflect across an arbitrary line through the origin $\mathcal{O}$: Q = R(θ) S R(-θ) P
- The rotation transforms the axis to the x-axis, the reflection is a scaling, and the last rotation transforms back to the original axis.
- Window-viewport: Translate by -w.l, -w.b, scale by A, B, translate by v.l, v.b.

Properties of 2D and 3D Affine Transformations

- Affine transformations *preserve* affine combinations of points.
  - $W = a_1P_1 + a_2P_2$ is an affine combination.
  - $MW = a_1MP_1 + a_2MP_2$
- Affine transformations preserve lines and planes.
  - A line through A and B is $L(t) = (1-t)A + tB$, an affine combination of points.
  - A plane can also be written as an affine combination of points: $P(s, a) = sA + tB +(1 – s – t)C$.

Properties of Transformations (2)

- Parallelism of lines and planes is preserved.
  - Line A + $\mathbf{b}t$ having direction $\mathbf{b}$ transforms to the line given in homogeneous coordinates by M(A + **b**t) **=** MA **+** M**b**t, which has direction vector M**b***.*
  - M**b** does *not* depend on point A. Thus two different lines A_1+ $\mathbf{b}t$ and A_2 + $\mathbf{b}t$ that have the same direction will transform into two lines both having the direction, so they *are* parallel.
- An important consequence of this property is that *parallelograms map into other parallelograms*.

Properties of Transformations (3)

- The direction vectors for a plane also transform into new direction vectors independent of the location of the plane.
- As a consequence, parallelepipeds map into other parallelepipeds.

Properties of Transformations (4)

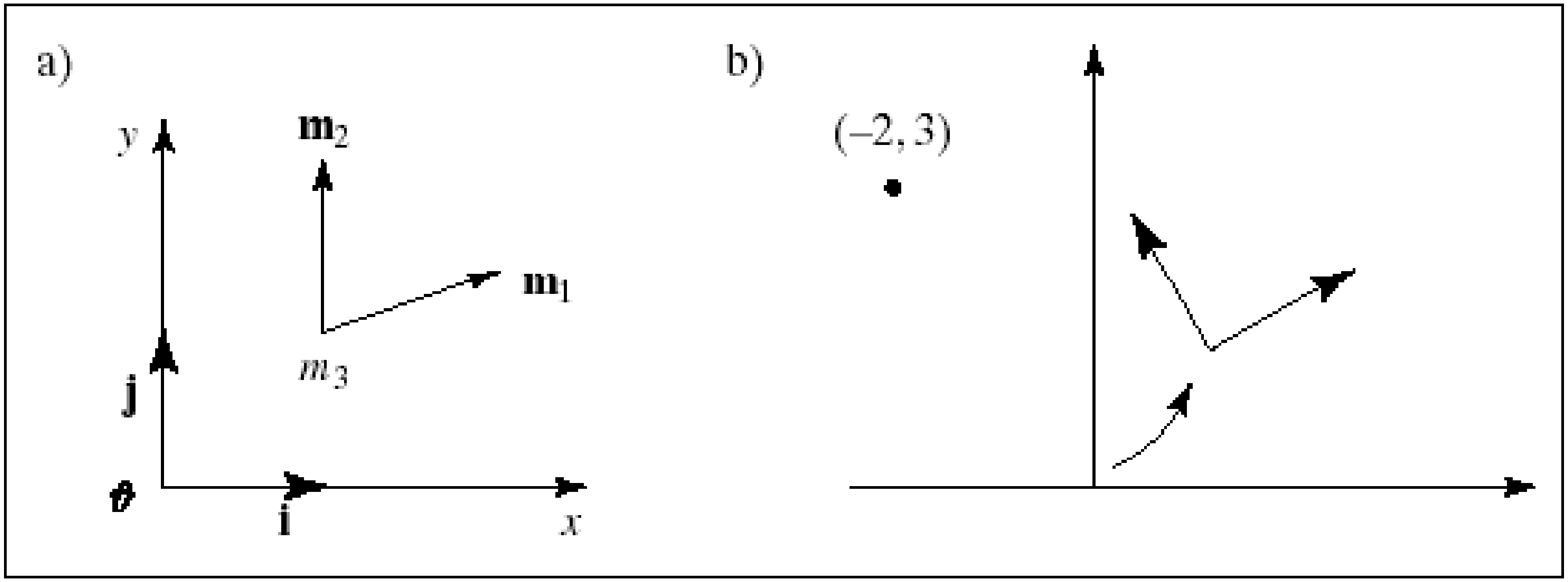
- The columns of the matrix reveal the transformed coordinate frame:
  - Vector **i** transforms into column m_1, vector **j** into column m_2, and the origin $\mathcal{O}$ into point m_3.
  - The coordinate frame (**i**, **j**, $\mathcal{O}$) transforms into the coordinate frame ($\mathbf{m}_1$, $\mathbf{m}_2$, m_3), and these new objects are precisely the columns of the matrix.

$$M = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{pmatrix} = \left(m_1 \mid m_2 \mid m_3 \right)$$

Properties of Transformations (5)

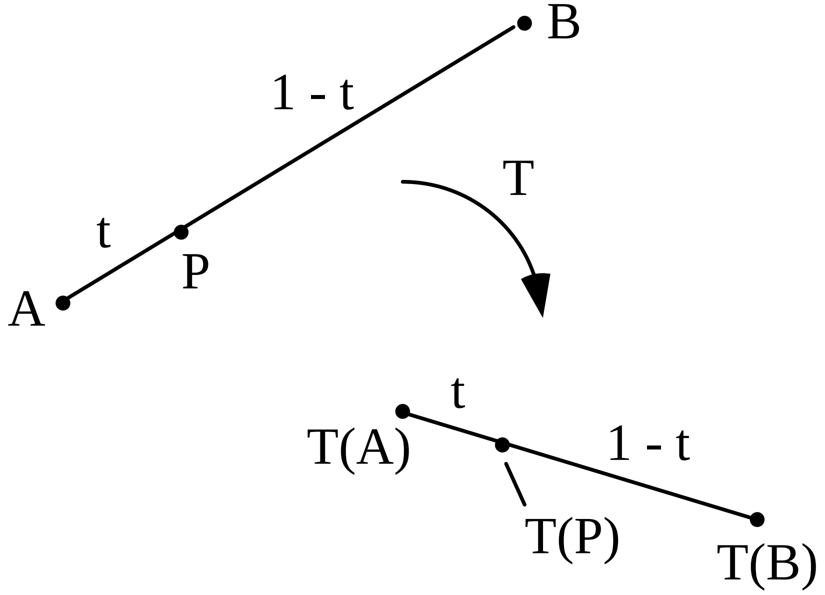
- The axes of the new coordinate frame are not necessarily perpendicular, nor must they be unit length.
  - They are still perpendicular if the transformation involves only rotations and uniform scalings.
- Any point $P = P_x\mathbf{i} + P_y\mathbf{j} + \mathcal{O}$ transforms into $Q = P_x\mathbf{m}_1 + P_y\mathbf{m}_2 + m_3$.

Properties of Transformations (6)



Properties of Transformations (7)

- Relative ratios are preserved: consider point *P* lying a fraction *t* of the way between two given points, *A* and *B* (see figure).
- Apply affine transformation *T*( ) to *A* , *B*, and *P*.
- The transformed point, *T*(*P*), lies the *same* fraction *t* of the way between images *T*(*A*) and *T*(*B*).



Properties of Transformations (8)

- How is the area of a figure affected by an affine transformation?
- It is clear that neither translations nor rotations have any effect on the area of a figure, but scalings certainly do, and shearing might.
- The result is simple: When the 2D transformation with matrix *M* is applied to an object, its area is multiplied by the *magnitude of the determinant* of *M*:

$$\frac{area\,after\,transformation}{area\,before\,transformation} = \left|\det M\right|$$

Properties of Transformations (9)

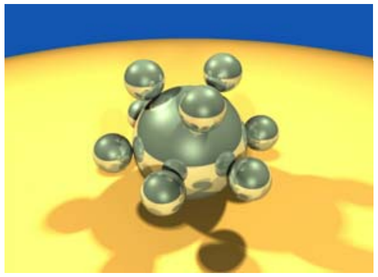
- In 2D the determinant of the matrix *M* is $(m_{11}m_{22} - m_{12}m_{21})$.
- For a pure scaling, the new area is S_xS_y times the original area, whereas for a shear along <u>one</u> axis the new area is the same as the original area.
- In 3D similar arguments apply, and we can conclude that the volume of a 3D object is scaled by $|\det M|$ when the object is transformed by the 3D transformation based on matrix *M*.

Properties of Transformations (10)

- Every affine transformation is composed of elementary operations.
- A matrix may be factored into a product of elementary matrices in various ways. One particular way of factoring the matrix associated with a 2D affine transformation yields

  M = (shear)(scaling)(rotation)(translation)
- That is, any 3 x 3 matrix that represents a 2D affine transformation can be written as the product of (reading right to left) a translation matrix, a rotation matrix, a scaling matrix, and a shear matrix.

Computer Graphics using Open GL, 3rd Edition
F. S. Hill, Jr. and S. Kelley



Chapter 5.3-5
Transformations of Objects

S. M. Lea
University of North Carolina at Greensboro

3D Affine Transformations

- Again we use coordinate frames, and suppose that we have an origin $\mathcal{O}$ and three mutually perpendicular axes in the directions **i**, **j**, and **k** (see Figure 5.8). Point *P* in this frame is given by $P = \mathcal{O} + P_x\mathbf{i} + P_y\mathbf{j} + P_z\mathbf{k}$, and vector V by $V_x\mathbf{i} + V_y\mathbf{j} + V_z\mathbf{k}$.

$$P = \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}, V = \begin{pmatrix} V_x \\ V_y \\ V_z \\ 0 \end{pmatrix}$$

3-D Affine Transformations

- The matrix representing a transformation is now 4 x 4, with Q = M P as before.

$$M = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- The fourth row of the matrix is a string of zeroes followed a lone one.

Translation and Scaling

- Translation and scaling transformation matrices are given below. The values S_x, S_y, and S_z cause scaling about the origin of the corresponding coordinates.

$$T = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}, S = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

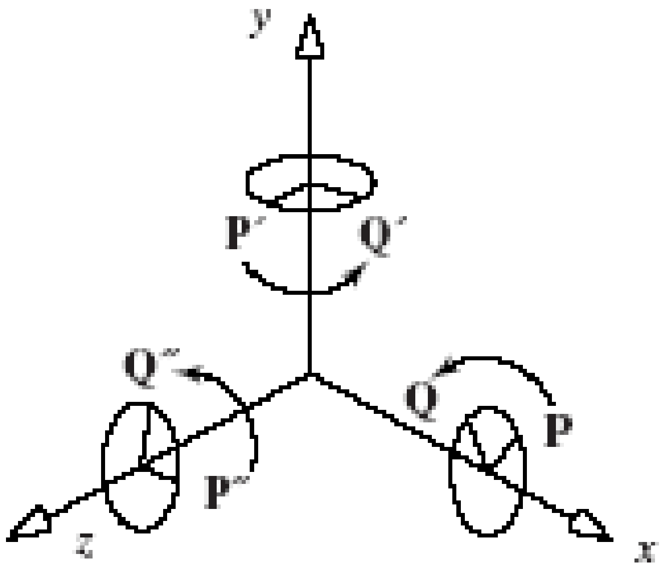
Shear

- The shear matrix is given below.
  - a: y along z; b: z along x; c: x along y; d: z along y; e: x along z; f: y along z
- Usually only one of {a,…,f} is non-zero.

$$H = \begin{pmatrix} 1 & a & b & 0 \\ c & 1 & d & 0 \\ e & f & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotations

- Rotations are more complicated. We start by defining a **roll** (rotation **counter-clockwise** around an axis looking **toward** the origin):



Rotations (2)

- *z*-roll: the *x*-axis rotates to the *y*-axis.
- *x*-roll: the *y*-axis rotates to the *z*-axis.
- *y*-roll: the *z*-axis rotates to the *x*-axis.

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\vartheta & -\sin\vartheta & 0 \\ 0 & \sin\vartheta & \cos\vartheta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, R_y = \begin{pmatrix} \cos\vartheta & 0 & \sin\vartheta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\vartheta & 0 & \cos\vartheta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

$$R_z = \begin{pmatrix} \cos\vartheta & -\sin\vartheta & 0 & 0 \\ \sin\vartheta & \cos\vartheta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

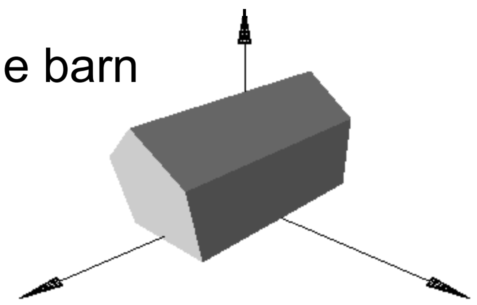
Rotations (3)

- Note that 12 of the terms in each matrix are the zeros and ones of the identity matrix.
- They occur in the row and column that correspond to the axis about which the rotation is being made (e.g., the first row and column for an *x*-roll).
- They guarantee that the corresponding coordinate of the point being transformed will not be altered.
- The *cos* and *sin* terms always appear in a rectangular pattern in the other rows and columns.

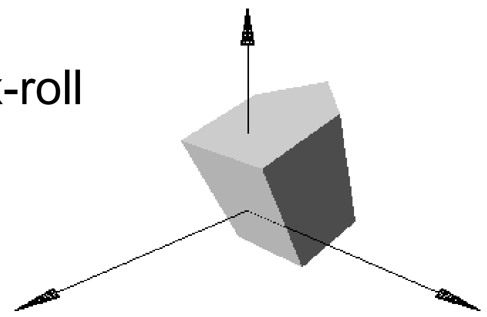
Example

- A barn in its original orientation, and after a -70° *x*-roll, a 30° *y*-roll, and a -90° *z*-roll.

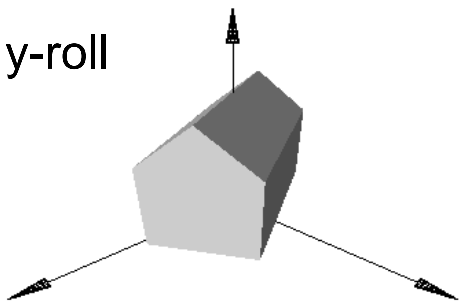
a). the barn



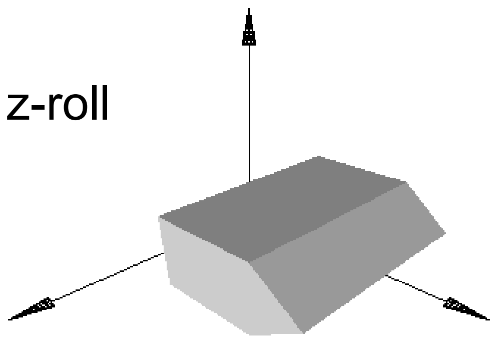
b). -70^0 x-roll



c). 30^0 y-roll



d). -90^0 z-roll

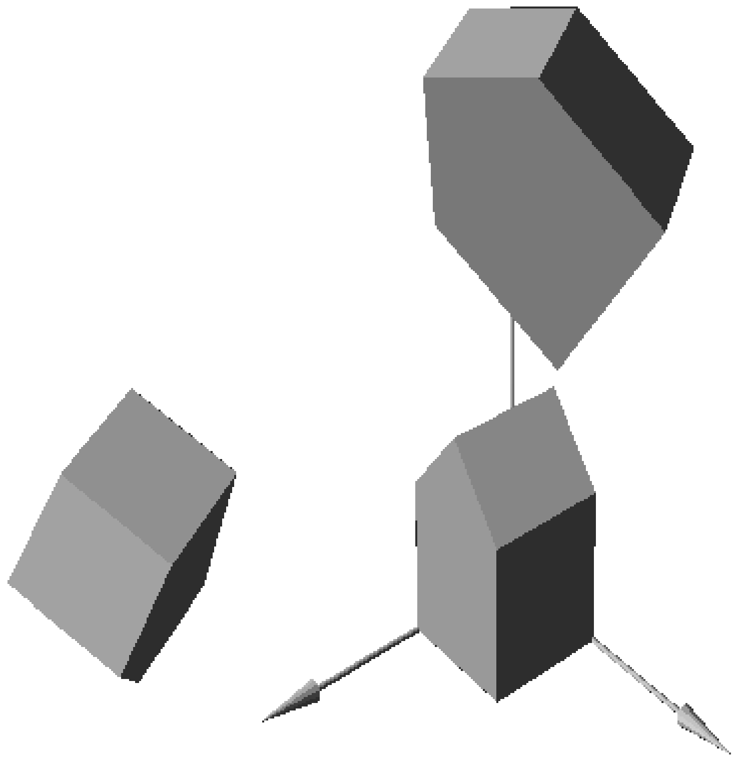


Composing 3D Affine Transformations

- 3D affine transformations can be composed, and the result is another 3D affine transformation.
- The matrix of the overall transformation is the product of the individual matrices M_1 and M_2 that perform the two transformations, with M_2 *pre-multiplying* M_1: $M = M_2 M_1$
- Any number of affine transformations can be composed in this way, and a single matrix results that represents the overall transformation.

Example

- A barn is first transformed using some M_1, and the transformed barn is again transformed using M_2. The result is the same as the barn transformed once using M_2M_1.



Building Rotations

- All 2D rotations are R_z. Two rotations combine to make a rotation given by the sum of the rotation angles, and the matrices commute.
- In 3D the situation is much more complicated, because rotations can be about different axes.
- The order in which two rotations about different axes are performed *does* matter: 3D rotation matrices do **not** commute.

Building Rotations (2)

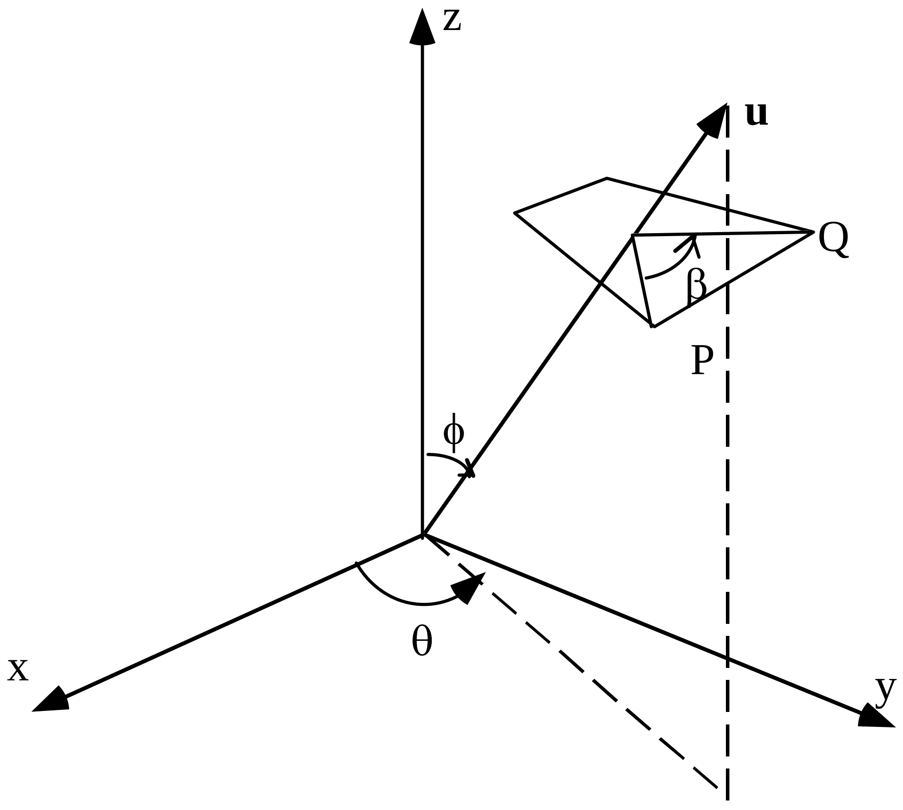
- We build a rotation in 3D by composing three elementary rotations: an *x*-roll followed by a *y*-roll, and then a *z*-roll. The overall rotation is given by $M = R_z(\beta_3)R_y(\beta_2)R_x(\beta_1)$.
- In this context the angles β_1, β_2, and β_3 are often called **Euler angles**.

Building Rotations (3)

- ***Euler’s Theorem: Any rotation (or sequence of rotations) about a point is equivalent to a single rotation about some axis through that point.***
- *Any* 3D rotation around an axis (passing through the origin) can be obtained from the product of five matrices for the appropriate choice of Euler angles; we shall see a method to construct the matrices.
- This implies that three values are required (and only three) to completely specify a rotation!

Rotating about an Arbitrary Axis

- We wish to rotate around axis **u** to make P coincide with Q.
- **u** can have any direction; it appears difficult to find a matrix that represents such a rotation.
- But it can be found in two ways, a classic way and a constructive way.



Rotating about an Arbitrary Axis (2)

- **The classic way.** Decompose the required rotation into a sequence of known steps:
  - Perform two rotations so that **u** becomes aligned with the *z*-axis.
  - Do a *z*-roll through angle *β*.
  - Undo the two alignment rotations to restore **u** to its original direction.
- $R_{\mathbf{u}}(\beta) = R_z(-\theta)\, R_y(-\Phi)\, R_z(\beta)\, R_y(\Phi)\, R_z(\theta)$ is the desired rotation.

Rotating about an Arbitrary Axis (3)

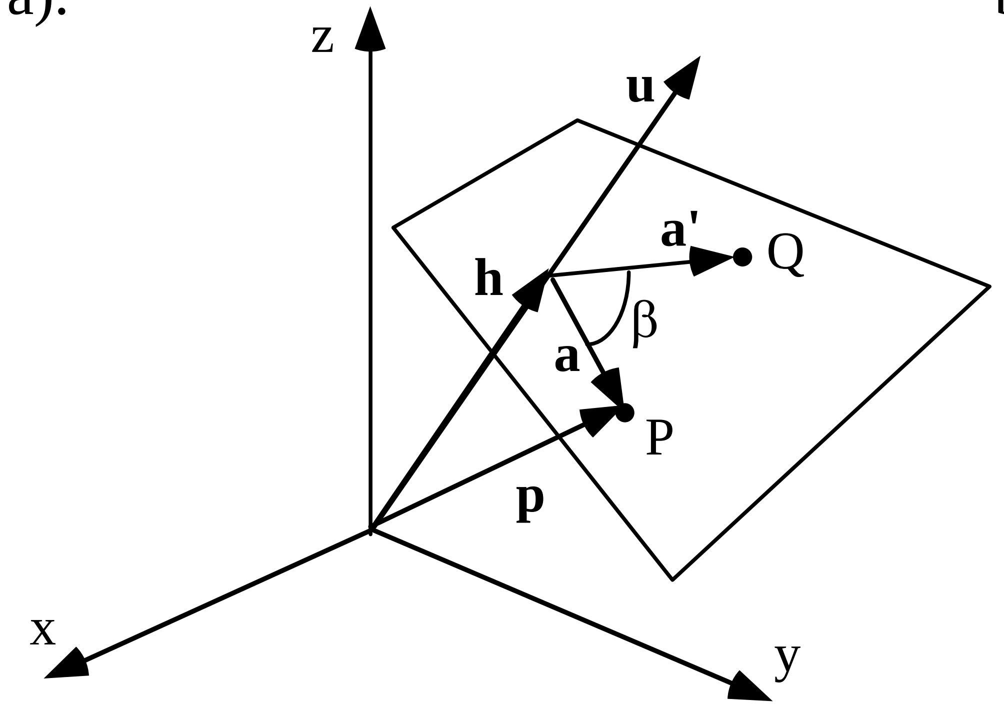
- **The constructive way.** Using some vector tools we can obtain a more revealing expression for the matrix $R_{\mathbf{u}}(b)$.
- We wish to express the operation of rotating point *P* through angle *b* into point *Q*.
- The method, given in Case Study 5.5, effectively establishes a 2D coordinate system in the plane of rotation as shown.

Rotating about an Arbitrary Axis (4)

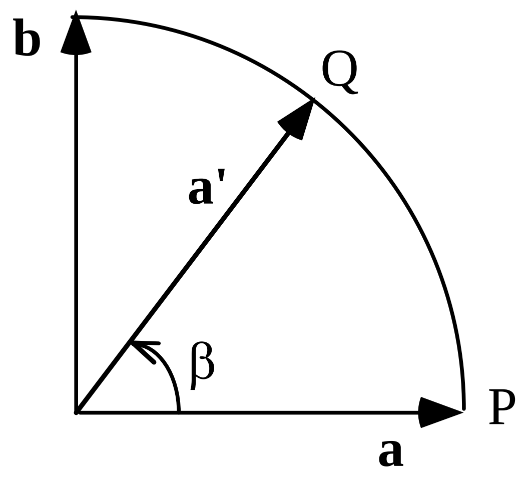
- This defines two orthogonal vectors **a** and **b** lying in the plane, and as shown in Figure 5.25b point *Q* is expressed as a linear combination of them. The expression for *Q* involves dot products and cross products of various ingredients in the problem.
- But because each of the terms is linear in the coordinates of *P*, it can be rewritten as *P* times a matrix.

Rotating about an Arbitrary Axis (5)

a).



b).



Rotating about an Arbitrary Axis (6)

- c = cos(β), s = sin(β), and u_x, u_y, u_z are the components of u.
- Then

$$R_u(\beta) = \begin{pmatrix} c+(1-c)u_x^2 & (1-c)u_y u_x - s u_z & (1-c)u_z u_x + s u_y & 0 \\ (1-c)u_x u_y + s u_z & c+(1-c)u_y^2 & (1-c)u_z u_y - s u_x & 0 \\ (1-c)u_x u_z - s u_y & (1-c)u_y u_z + s u_x & c+(1-c)u_z^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotating about an Arbitrary Axis (6)

- Open-GL provides a rotation about an arbitrary axis:

```
glRotated (beta, ux, uy, uz);
```

- beta is the angle of rotation.
- ux, uy, uz are the components of a vector u normal to the plane containing P and Q.

Summary of Properties of 3D Affine Transformations

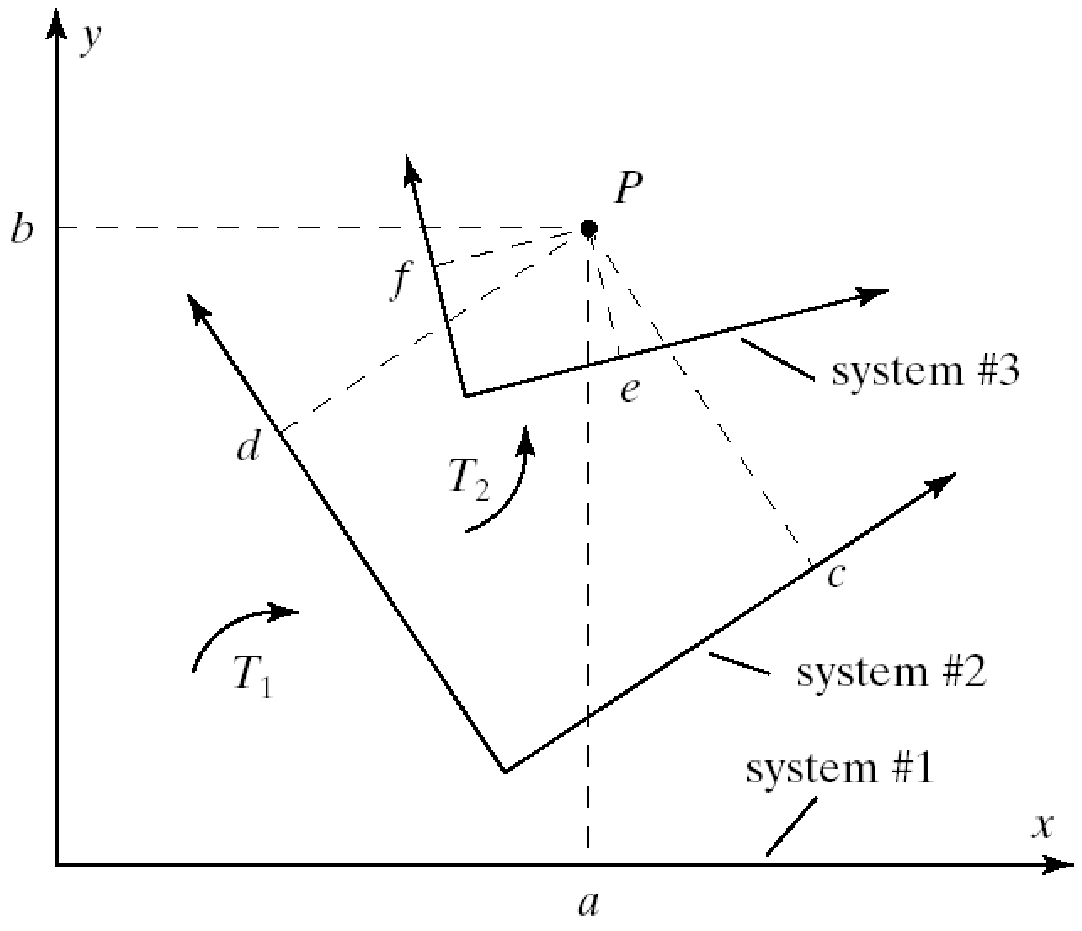
- **Affine transformations** preserve **affine combinations of points**.
- **Affine transformations preserve lines and planes.**
- **Parallelism of lines and planes is preserved.**
- **The columns of the matrix reveal the transformed coordinate frame.**
- **Relative ratios are preserved.**

Summary of Properties of 3D Affine Transformations (2)

- **The effect of transformations on the volumes of objects.** If 3D object *D* has volume *V*, then its image *T*(*D*) has volume |*det M* | *V*, where |*det M*| is the absolute value of the determinant of *M*.
- **Every affine transformation is composed of elementary operations.** A 3D affine transformation may be decomposed into a composition of elementary transformations. See Case Study 5.3.

Transforming Coordinate Systems

- We have a 2D coordinate frame #1, with origin $\mathcal{O}$ and axes **i** and **j**.
- We have an affine transformation *T*(.) with matrix *M*, where *T*(.) transforms coordinate frame #1 into coordinate frame #2, with new origin $\mathcal{O}$' = *T*($\mathcal{O}$), and new axes **i**' = *T*(**i**) and **j**' = *T*(**j**).



Transforming Coordinate Systems (2)

- Now let *P* be a point with representation $(c, d, 1)^T$ in the new system #2.
- What are the values of *a* and *b* in its representation $(a, b, 1)^T$ in the original system #1?
- The answer: simply *premultiply* $(c, d, 1)^T$ by *M*:

$$(a, b, 1)^T = M\,(c, d, 1)^T$$

Transforming Coordinate Systems (3)

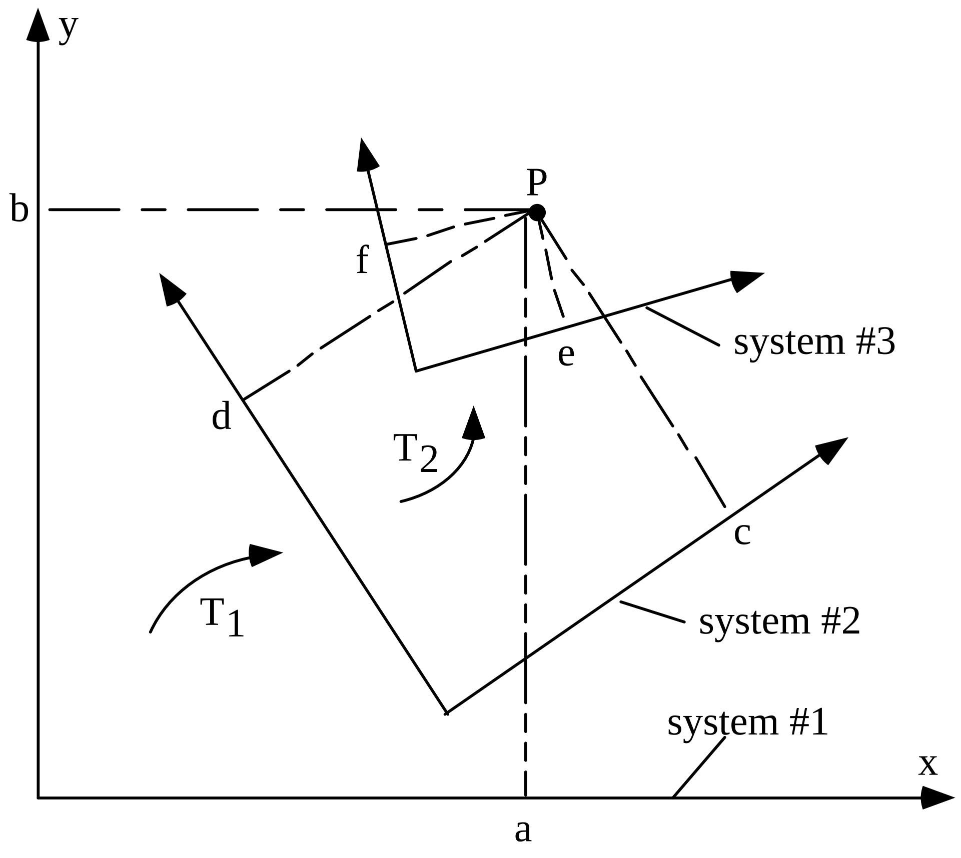
- We have the following theorem:
- Suppose coordinate system #2 is formed from coordinate system #1 by the affine transformation *M*. Further suppose that point $P = (P_x, P_y, P_z, 1)$ are the coordinates of a point *P* expressed in system #2. Then the coordinates of *P* expressed in system #1 are *MP*.

Successive Transformations

- Now consider forming a transformation by making two successive changes of the coordinate system. What is the overall effect?
- System #1 is converted to system #2 by transformation $T_1(.)$, and system #2 is then transformed to system #3 by transformation $T_2(.)$. Note that system #3 is transformed *relative* to #2.

Successive Transformations (2)

- Point *P* has representation $(e, f, 1)^T$ with respect to system #3. What are its coordinates $(a, b, 1)^T$ with respect to the original system #1?



Successive Transformations (3)

- To answer this, collect the effects of each transformation: In terms of system #2 the point P has coordinates $(c, d, 1)^T = M_2(e, f, 1)^T$. And in terms of system #1 the point $(c, d, 1)^T$ has coordinates $(a, b, 1)^T = M_1(c, d, 1)^T$. So $(\mathrm{a}, \mathrm{b}, 1)^{\mathrm{T}} = \mathrm{M}_1(\mathrm{d}, \mathrm{c}, 1)^{\mathrm{T}} = \mathrm{M}_1\mathrm{M}_2(\mathrm{e}, \mathrm{f}, 1)^{\mathrm{T}}$
- The essential point is that when determining the desired coordinates $(a, b, 1)^T$ from $(e, f, 1)^T$ we *first* apply M_2 and *then* M_1, just the *opposite* order as when applying transformations to points.

Successive Transformations (4)

- **To transform points.** To apply a sequence of transformations $T_1()$, $T_2()$, $T_3()$ (in that order) to a point P, form the matrix $M = M_3 \times M_2 \times M_1$.
- Then P is transformed to *MP; pre-multiply* by M_i.
- **To transform the coordinate system.** To apply a sequence of transformations $T_1()$, $T_2()$, $T_3()$ (in that order) to the coordinate system, form the matrix $M = M_1 \times M_2 \times M_3$.
- Then P in the transformed system has coordinates MP in the original system. To compose each additional transformation M_i you must *post-multiply* by M_i.

Open-GL Transformations

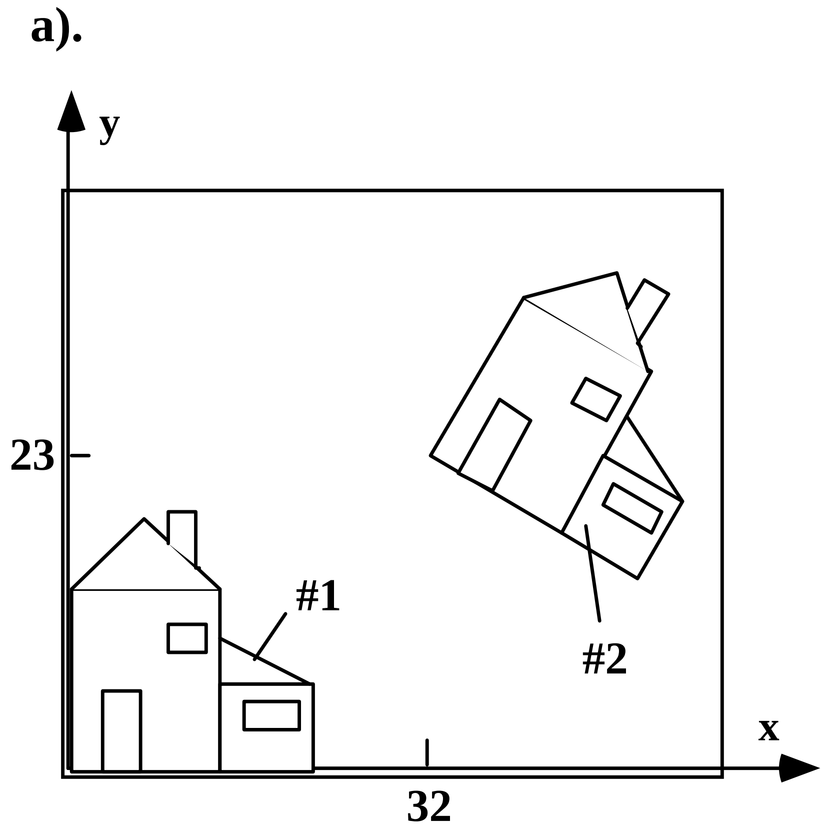
- Open-GL actually transforms coordinate systems, so <u>in your programs</u> you will have to apply the transformations in reverse order.
- E.g., if you want to translate the 3 vertices of a triangle and then rotate it, your program will have to do rotate and then translate.

Using Affine Transformations in Open-GL

- glScaled (sx, sy, sz); // 2-d: sz = 1.0
- glTranslated (tx, ty, tz); //2-d: tz = 0.0
- glRotated (angle, ux, uy, uz); // 2-d: ux = uy = 0.0; uz = 1.0
- The sequence of commands is
  - glLoadIdentity();
  - glMatrixMode (GL_MODELVIEW);
  - // transformations 1, 2, 3, ... (in reverse order)
- This method makes Open-GL do the work of transforming for you.

Example

- We have version 1 of the house defined (vertices set), but what we really want to draw is version 2.
- We could write routines to transform the coordinates – this is the hard way.
- The easy way lets GL do the transforming.

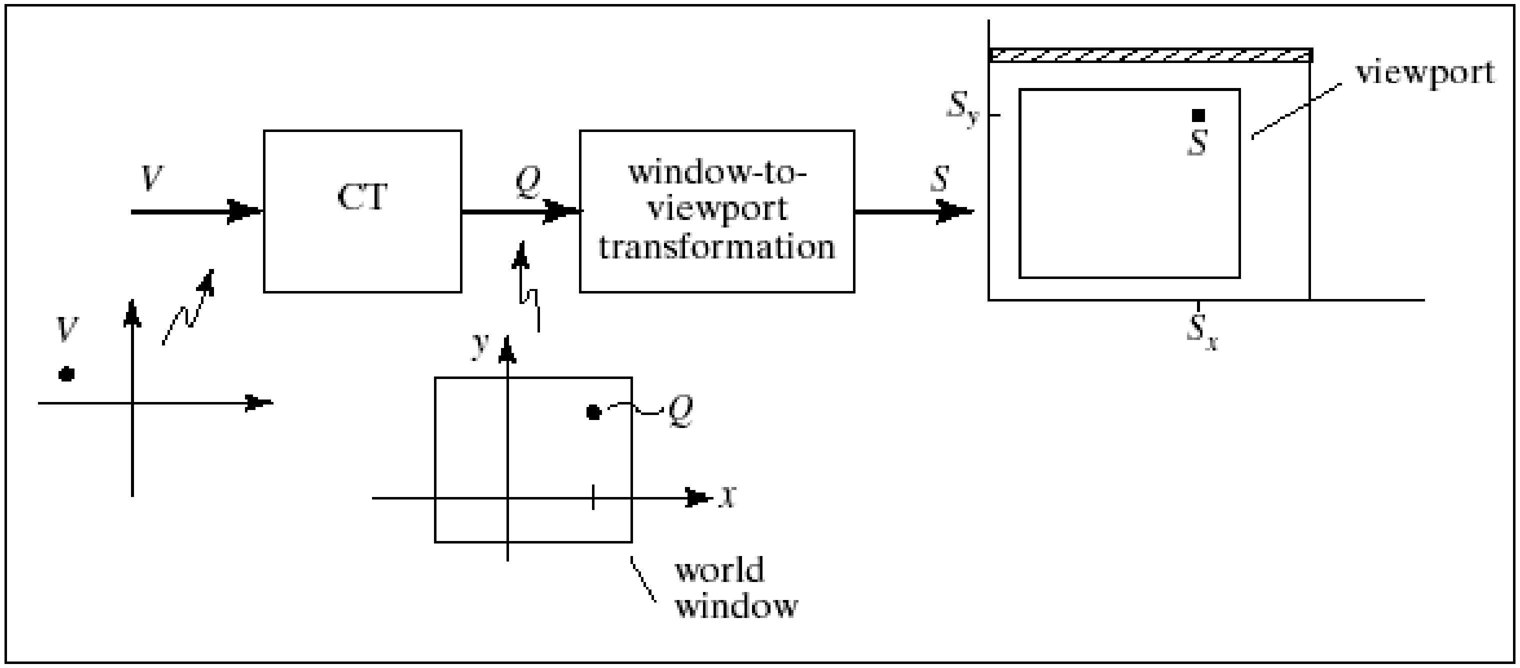


Example: the Easy Way (2)

- We cause the desired transformation to be applied automatically to each vertex. Just as we know the window to viewport mapping is quietly applied to each vertex as part of the graphics pipeline, we can have an additional transformation be applied as well.
- It is often called the **current transformation,** *CT*. We enhance moveTo() and lineTo() so that they first apply this transformation to the argument vertex, and then apply the window to viewport mapping.

Example (3)

- When glVertex2d()is called with argument V, the vertex *V* is first transformed by the *CT* to form point *Q*.
- *Q* is then passed through the window to viewport mapping to form point *S* in the screen window.



Example (4)

- How do we extend moveTo() and lineTo() so they carry out this additional mapping?
- The transform is done automatically by OpenGL! OpenGL maintains a so-called **modelview matrix,** and every vertex that is passed down the graphics pipeline is multiplied by this modelview matrix.
- We need only set up the modelview matrix once to embody the desired transformation.

Example (5)

- The principal routines for altering the modelview matrix are glRotated(), glScaled(), and glTranslated().
- These don't set the *CT* directly; instead each one *postmultiplies* the *CT* (the modelview matrix) by a particular matrix, say *M*, and puts the result back into the *CT*.
- That is, each of these routines creates a matrix *M* as required for the new transformation, and performs: CT = CT *M.

Example (6)

- glScaled (sx, sy, sz);   // 2-d: sz = 1.0
- glTranslated (tx, ty, tz); //2-d: tz = 0.0
- glRotated (angle, ux, uy, uz); // 2-d: ux = uy = 0.0; uz = 1.0
- This method makes Open-GL do the work of transforming for you.

Example (7)

- Of course, we have to start with some MODELVIEW matrix:
- The sequence of commands is
  - glMatrixMode (GL_MODELVIEW);
  - glLoadIdentity();
  - // transformations 1, 2, 3, ... (in reverse order)
- Wrapper code for routines to manipulate the CT is in Figure 5.33.

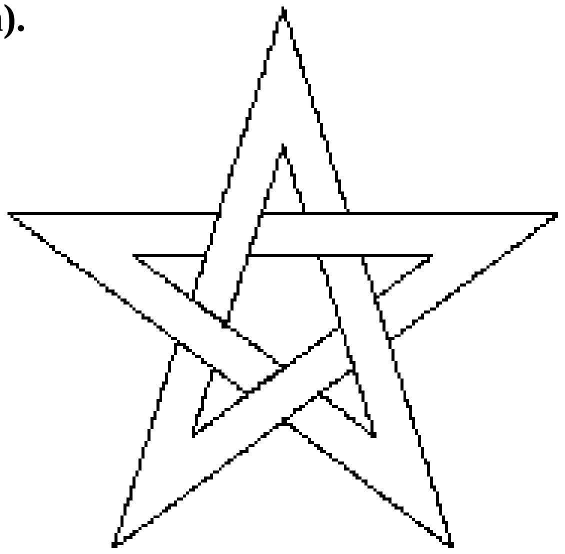
Example (8)

- Code to draw house #2: note translate is done before rotate (reverse order).
- setWindow(...);
- setViewport(..); // set window to viewport
  // mapping
- initCT(); // get started with identity
  // transformation
- translate2D(32, 25); // CT includes translation
- rotate2D(-30.0); // CT includes translation and
  // rotation
- house(); // draw the transformed house

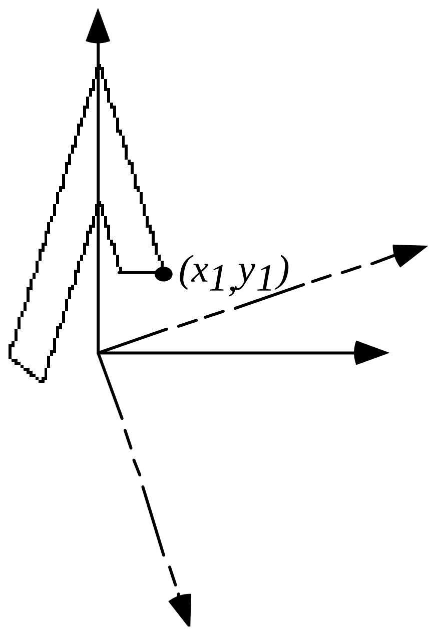
Example 2: Star

- A star made of "interlocking" stripes: starMotif() draws a part of the star, the polygon shown in part b. (Help on finding polygon's vertices in Case Study 5.1.)
- To draw the whole star we draw the motif five times, each time rotating the motif through an additional 72°.

a).



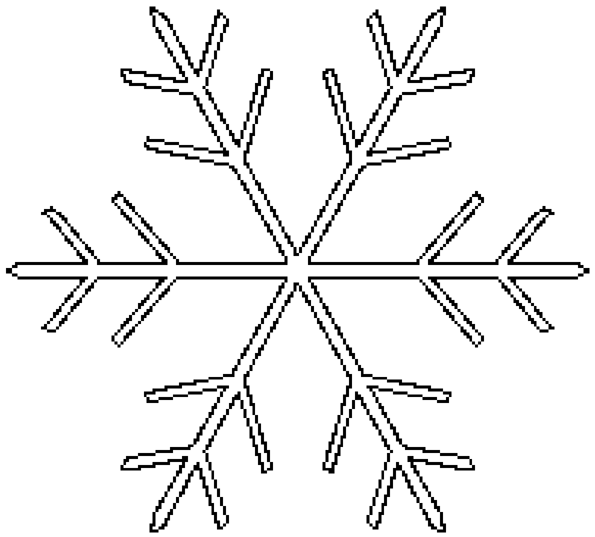
b).



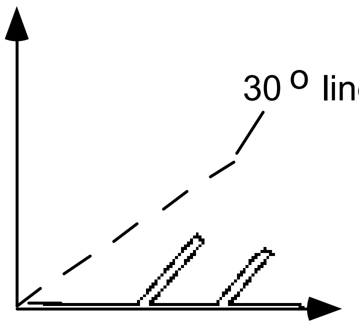
Example 3: Snowflake

- The motif and the figure are shown below. glScaled() is used to reflect the motif to get a complete branch and then to restore the original axis. Rotate by 60^o between branches.

a).

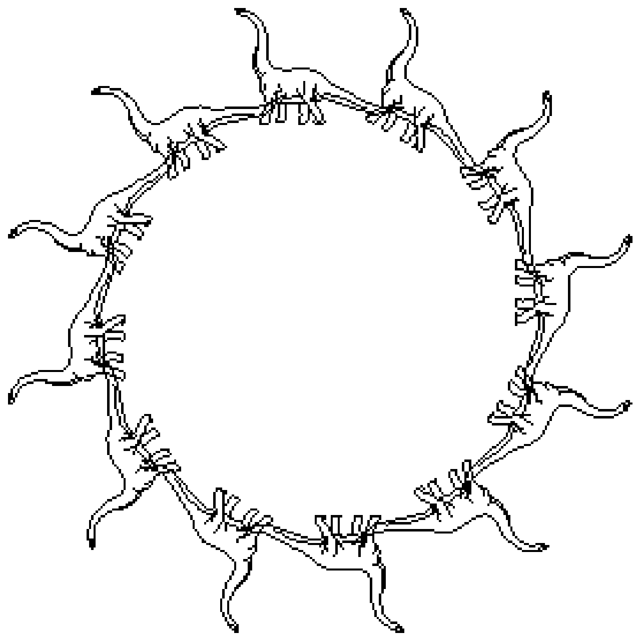
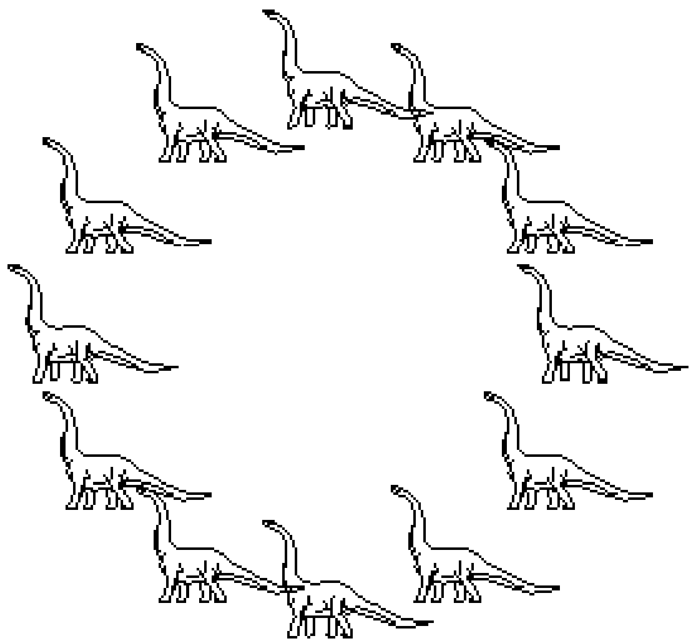


b).



Example 4: Dino Patterns

- The dinosaurs are distributed around a circle in both versions. Left: each dinosaur is rotated so that its feet point toward the origin; right: all the dinosaurs are upright.

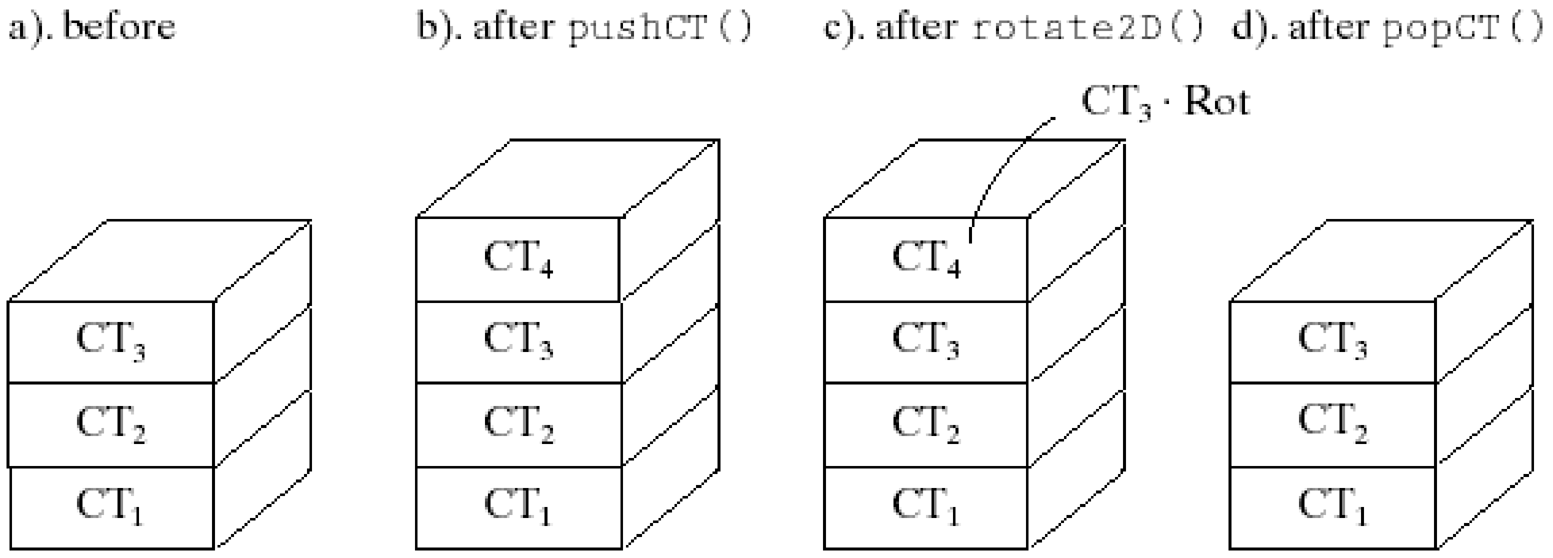
Example 4 (2)

- drawDino() draws an upright dinosaur centered at the origin.
- In a) the coordinate system for each motif is rotated about the origin through a suitable angle, and then translated along its *y*-axis by *H* units.
- Note that the *CT* is reinitialized each time through the loop so that the transformations don’t accumulate.
- An easy way to keep the motifs upright (as in part b) is to pre-rotate each motif before translating it.

Affine Transformations Stack

- It is also possible to push/pop the current transformation from a stack in OpenGL, using the commands

```
glMatrixMode (GL_MODELVIEW);
glPushMatrix(); //or glPopMatrix();
```



Affine Transformations Stack (2)

- The implementation of pushCT() and popCT() uses OpenGL routines glPushMatrix() and glPopMatrix().
- Caution: Note that each routine must inform OpenGL which matrix stack is being affected.
- In OpenGL, popping a stack that contains only one matrix is an error; test the number of matrices using OpenGL's query function glGet(G L_MODELVIEW_STACK_DEPTH).

Affine Transformations Stack (3)

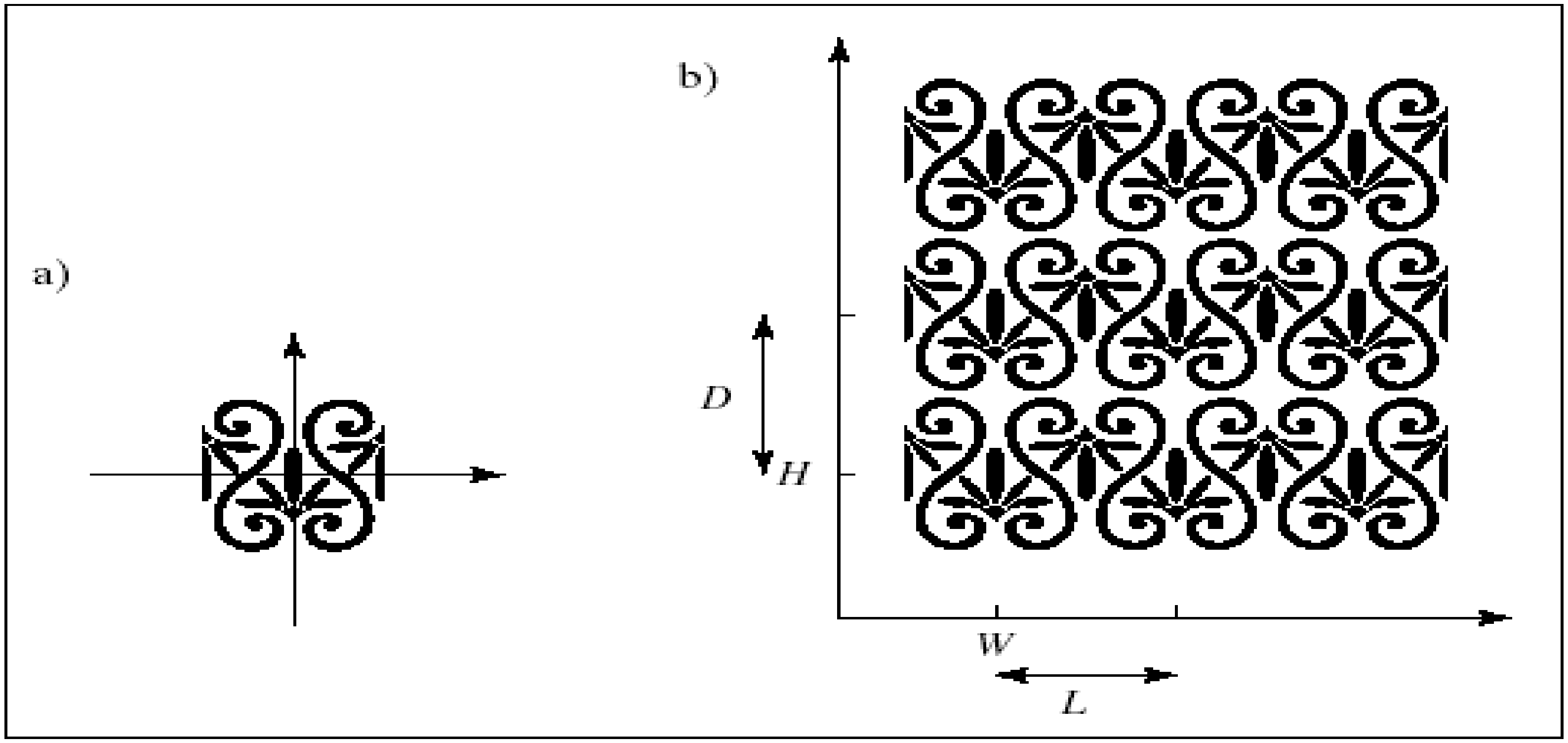
```
pushCT(void)
{   glMatrixMode(GL_MODELVIEW);
    glPushMatrix();           // push a copy of the top matrix
}
checkStack(void)
{   if (glGet (GL_MODELVIEW_STACK_DEPTH) ≤ 1) )
    // do something
    else  popCT();
}
popCT(void)
{   glMatrixMode(GL_MODELVIEW);
    glPopMatrix();            // pop the top matrix from the stack
}
```

Example 5: Motif

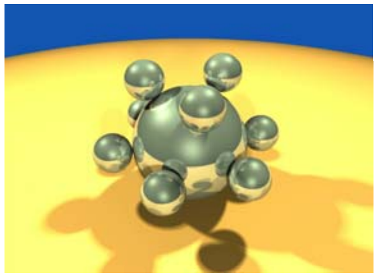
- **Tilings** are based on the repetition of a basic motif both horizontally and vertically.
- Consider tiling the window with some motif, drawn centered in its own coordinate system by routine motif().
- Copies of the motif are drawn *L* units apart in the x-direction, and *D* units apart in the *y*-direction, as shown in part b).

Example 5 (2)

- The motif is translated horizontally and vertically to achieve the tiling.



Computer Graphics using OpenGL, 3rd Edition
F. S. Hill, Jr. and S. Kelley



Chapter 5.6 Transformations of Objects

S. M. Lea
University of North Carolina at Greensboro

Drawing 3D Scenes in OpenGL

- We want to transform objects in order to orient and position them as desired in a 3D scene.
- OpenGL provides the necessary functions to build and use the required matrices.
- The matrix stacks maintained by OpenGL make it easy to set up a transformation for one object, and then return to a previous transformation, in preparation for transforming another object.

The Camera in OpenGL

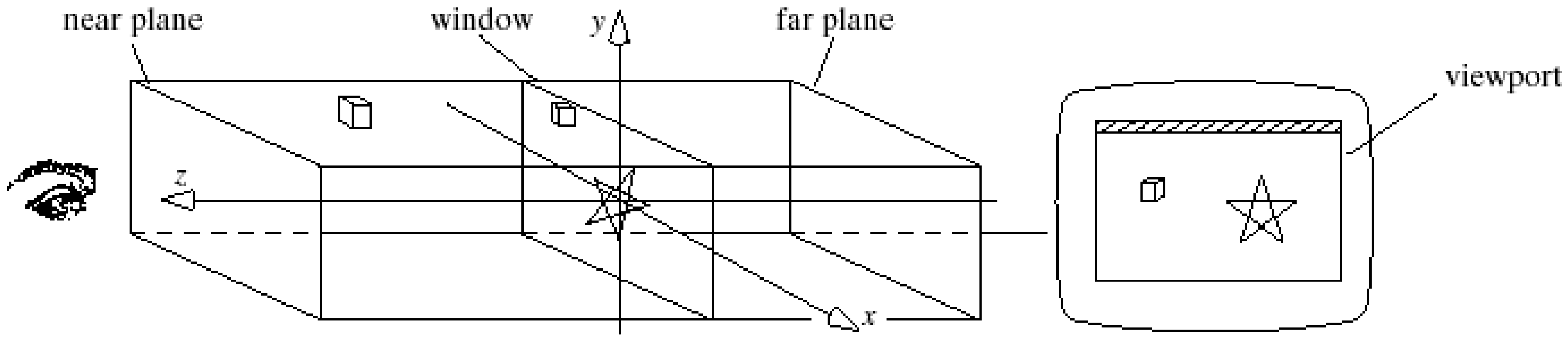
- The camera is created with a matrix.
  - We will study the details of how this is done in Chapter 7.
- For now, we just use an OpenGL tool to set up a reasonable camera so that we may pay attention primarily to transforming objects.

Interactive Programs

- In addition, we show how to make these **programs interactive** so that *at run time* the user can alter key properties of the scene and its objects.
- The camera can be altered using the mouse and keyboard so that the display can be made to change dramatically in real time. (Case Study 5.3.)

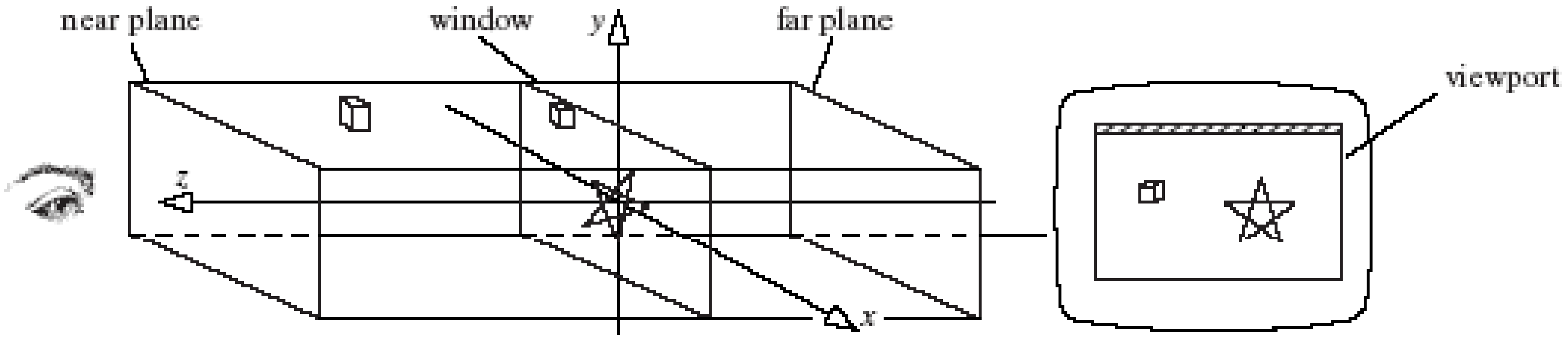
The Viewing Process and the Graphics Pipeline

- The 2D drawing so far is a special case of 3D viewing, based on a simple parallel projection.
- The eye is looking along the *z*-axis at the world window, a rectangle in the *xy*-plane.



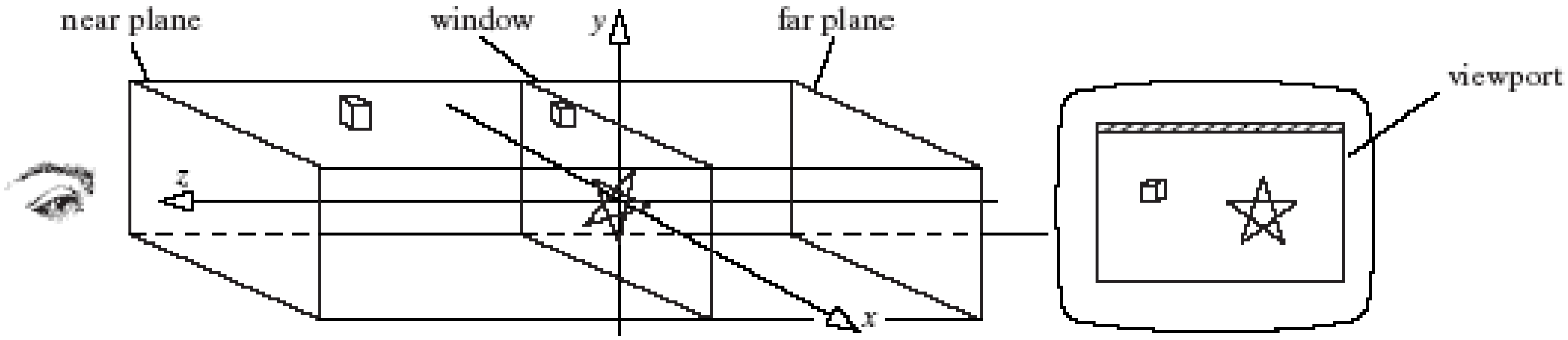
The Viewing Process and the Graphics Pipeline (2)

- *Eye* is simply a point in 3D space.
- The “orientation” of the eye ensures that the view volume is in front of the eye.
- Objects closer than *near* or farther than *far* are too blurred to see.



The Viewing Process and the Graphics Pipeline (3)

- The **view volume** of the camera is a rectangular parallelepiped.
- Its side walls are fixed by the window edges; its other two walls are fixed by a **near plane** and a **far plane**.

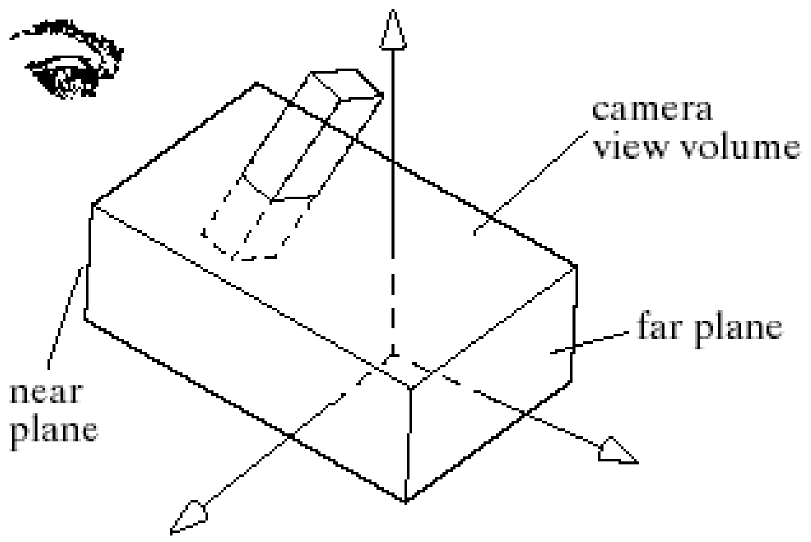


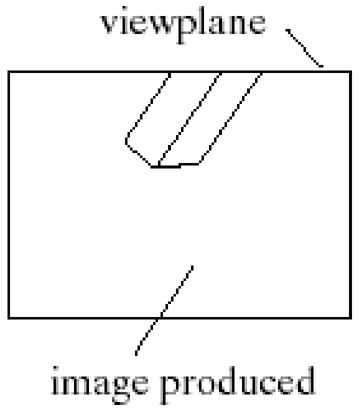
The Viewing Process and the Graphics Pipeline (4)

- Points inside the view volume are projected onto the window along lines parallel to the z-axis.
- We ignore their z-component, so that the 3D point (x_1 y_1, z_1) projects to (x_1, y_1, 0).
- Points lying outside the view volume are clipped off.
- A separate **viewport transformation** maps the projected points from the window to the viewport on the display device.

The Viewing Process and the Graphics Pipeline (5)

- In 3D, the only change we make is to allow the camera (eye) to have a more general position and orientation in the scene in order to produce better views of the scene.



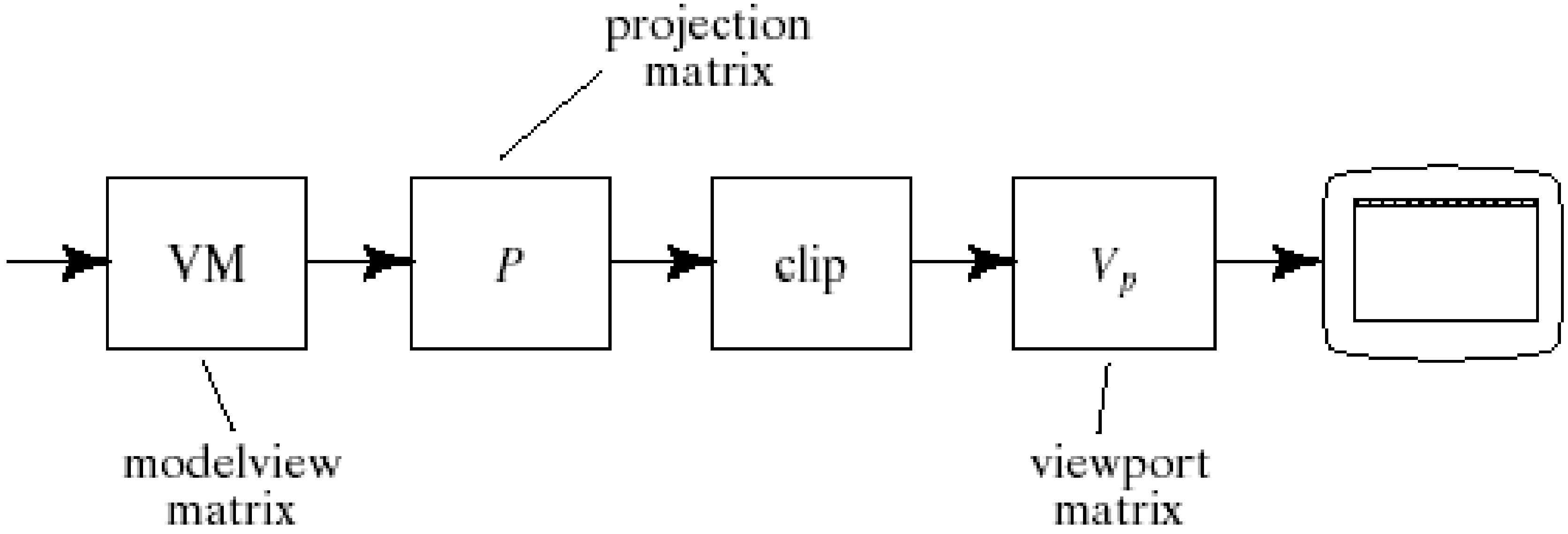


The Viewing Process and the Graphics Pipeline (6)

- The z axis points *toward* the eye. X and y point to the viewer’s right and up, respectively.
- Everything outside the view volume is clipped.
- Everything inside it is projected along lines parallel to the axes onto the window plane (parallel projection).

The Viewing Process and the Graphics Pipeline (7)

- OpenGL provides functions for defining the view volume and its position in the scene, using matrices in the graphics pipeline.



The Viewing Process and the Graphics Pipeline (8)

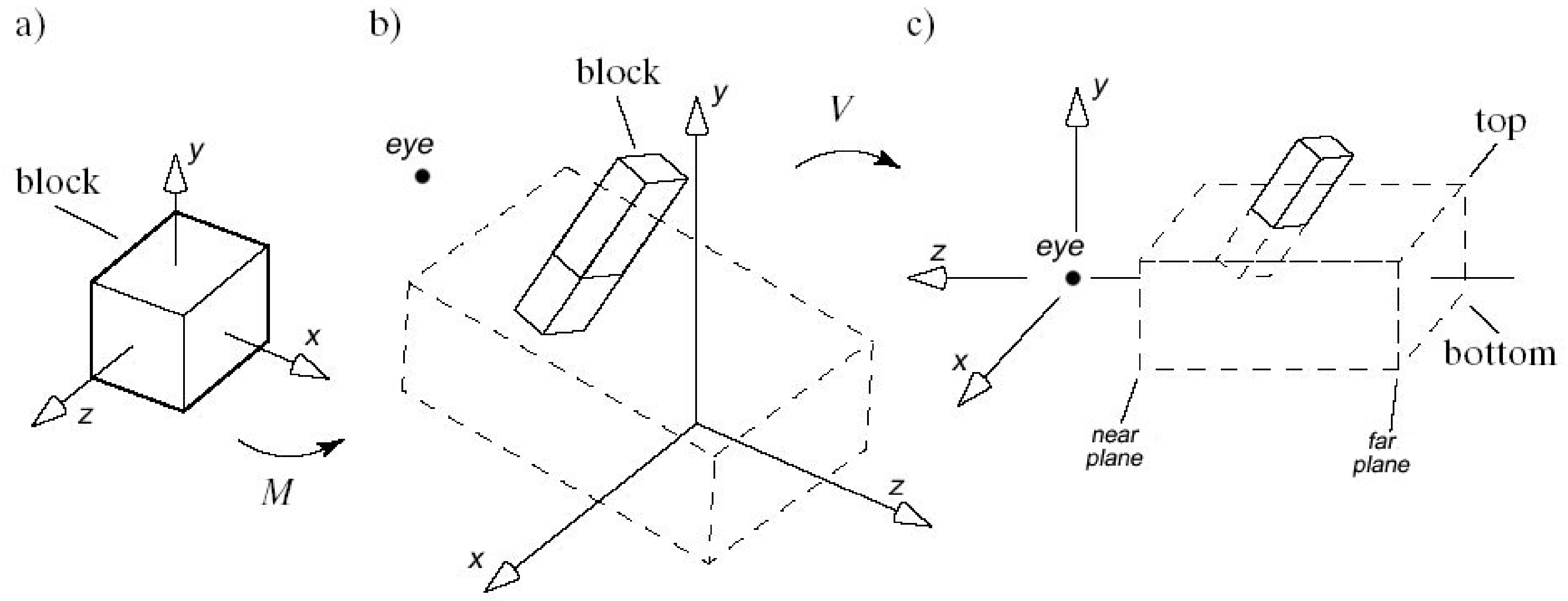
- Each vertex of an object is passed through this pipeline using glVertex3d(x, y, z).
- The vertex is multiplied by the various matrices, clipped if necessary, and if it survives, it is mapped onto the viewport.
- Each vertex encounters three matrices:
  - The **modelview matrix;**
  - The **projection matrix;**
  - The **viewport matrix;**

The Modelview Matrix

- The **modelview matrix** is the *CT* (current transformation).
- It combines modeling transformations on objects and the transformation that orients and positions the camera in space (hence *modelview*).
- It is a single matrix in the actual pipeline.
  – For ease of use, we will think of it as the product of two matrices: a modeling matrix *M*, and a viewing matrix *V*. The modeling matrix is applied first, and then the viewing matrix, so the modelview matrix is in fact the product *VM.*

The Modelview Matrix (M)

- A modeling transformation *M* scales, rotates, and translates the cube into the block.



The Modelview Matrix (V)

- The *V* matrix rotates and translates the block into a new position.
- The camera moves from its position in the scene to its generic position (eye at the origin and the view volume aligned with the *z*-axis).
- The coordinates of the block’s vertices are changed so that projecting them onto a plane (e.g., the near plane) displays the projected image properly.

The Modelview Matrix (V)

- The matrix *V* changes the coordinates of the scene vertices into the **camera's coordinate system,** or into **eye coordinates**.
- To inform OpenGL that we wish it to operate on the modelview matrix we call glMatrixMode(GL_MODELVIEW);

The Projection Matrix

- The **projection matrix** scales and translates each vertex so that those inside the view volume will be inside a *standard cube* that extends from -1 to 1 in each dimension (Normalized Device Coordinates).
- This cube is a particularly efficient boundary against which to clip objects.
- The image is distorted, but the viewport transformation will remove the distortion.
- The projection matrix also reverses the sense of the z-axis; increasing values of *z* now represent increasing values of depth from the eye.

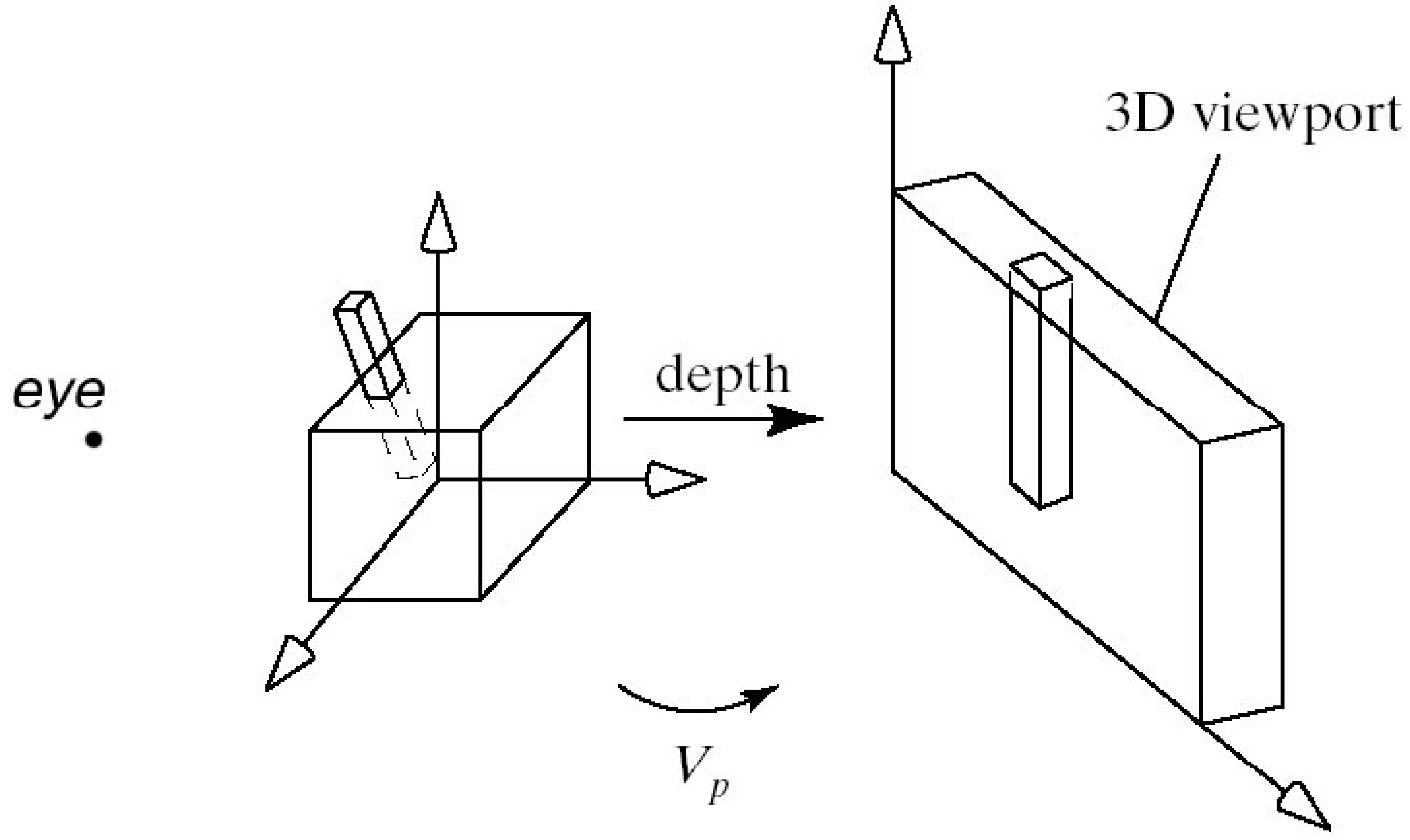
The Projection Matrix (2)

- Setting the Projection Matrix:
  - glMatrixMode(GL_PROJECTION);
  - glLoadIdentity (); // initialize projection matrix
  - glOrtho (left, right, bottom, top, near, far); // sets the view volume parellelpiped. (All arguments are glDouble ≥ 0.0.)
- left ≤ vv.x ≤ right, bottom ≤ vv.y ≤ top, and -near ≤ vv.z ≤ -far (camera at the origin looking along -z).

The Viewport Matrix

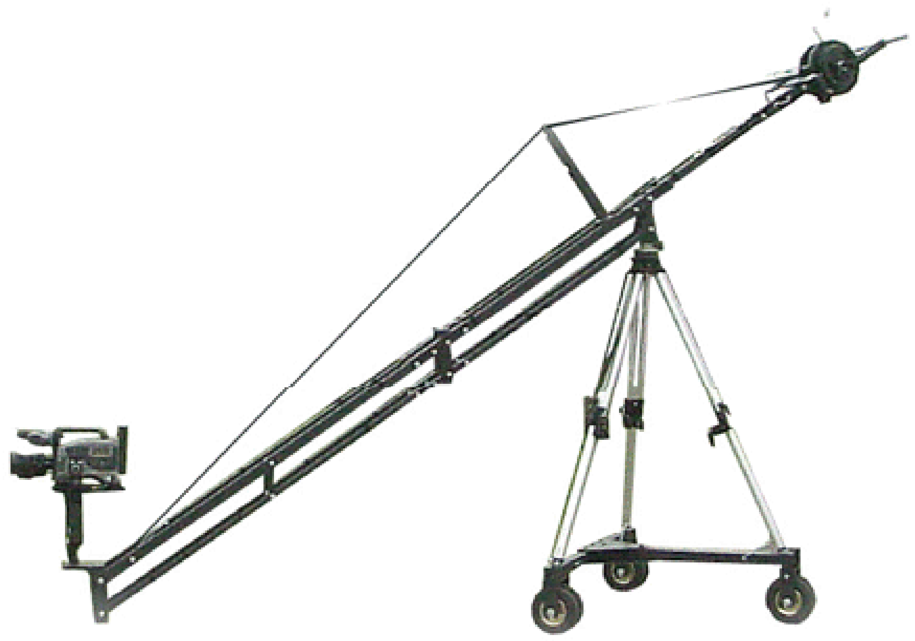
- The **viewport matrix** maps the standard cube into a 3D viewport whose *x* and *y* values extend across the viewport (in screen coordinates), and whose *z*-component extends from 0 to 1 (a measure of the depth of each point).
- This measure of depth makes hidden surface removal (do not draw surfaces hidden by objects closer to the eye) particularly efficient.

The Viewport Matrix (2)



Setting Up the Camera

- We shall use a **jib camera**.
- The photographer rides at the top of the tripod.
- The camera moves through the scene bobbing up and down to get the desired shots.



Setting Up the Scene (2)

glMatrixMode (GL_MODELVIEW);

// set up the modelview matrix

glLoadIdentity ();

// initialize modelview matrix

// set up the view part of the matrix

// do any modeling transformations on the scene

Setting Up the Projection

```
glMatrixMode(GL_PROJECTION);
   // make the projection matrix current
glLoadIdentity();
   // set it to the identity matrix
glOrtho(left, right, bottom, top, near, far);
   // multiply it by the new matrix
```

- Using 2 for *near* places the near plane at z = -2, that is, 2 units in front of the eye.
- Using 20 for *far* places the far plane at -20, 20 units in front of the eye.

Setting Up the Camera (View Matrix)

```
glMatrixMode (GL_MODELVIEW);
   // make the modelview matrix current
glLoadIdentity();
   // start with identity matrix
   // position and aim the camera
gluLookAt (eye.x, eye.y, eye.z,    // eye position
    look.x, look.y, look.z,       // the “look at” point
     0, 1, 0)   // approximation to true up direction
   // Now do the modeling transformations
```

Setting Up the Camera (2)

- What gluLookAt does is create a camera coordinate system of three mutually orthogonal unit vectors: **u**, **v**, and **n.**
- **n** = eye - look; **u** = **up** x **n**; **v** = **n** x **u**
- Normalize **n**, **u**, **v** (in the camera system) and let **e** = eye - $\mathcal{O}$ in the camera system, where $\mathcal{O}$ is the origin.

Setting Up the Camera (3)

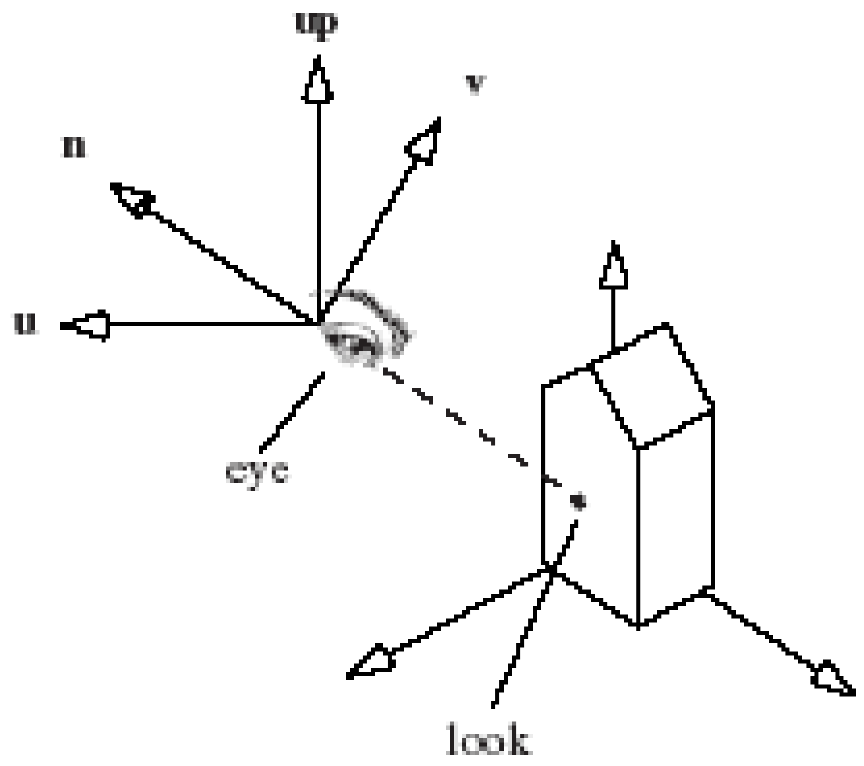
- Then gluLookAt () sets up the view matrix

$$V = \begin{pmatrix} u_x & u_y & u_z & d_x \\ v_x & v_y & v_z & d_y \\ n_x & n_y & n_z & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

  where **d** = (**-e·u**, **-e·v**, **-e·n**)
- **up** is usually (0, 1, 0) (along the y-axis), *look* is frequently the middle of the window, and *eye* frequently looks down on the scene.

The gluLookAt Coordinate System

- Camera in world coordinates:

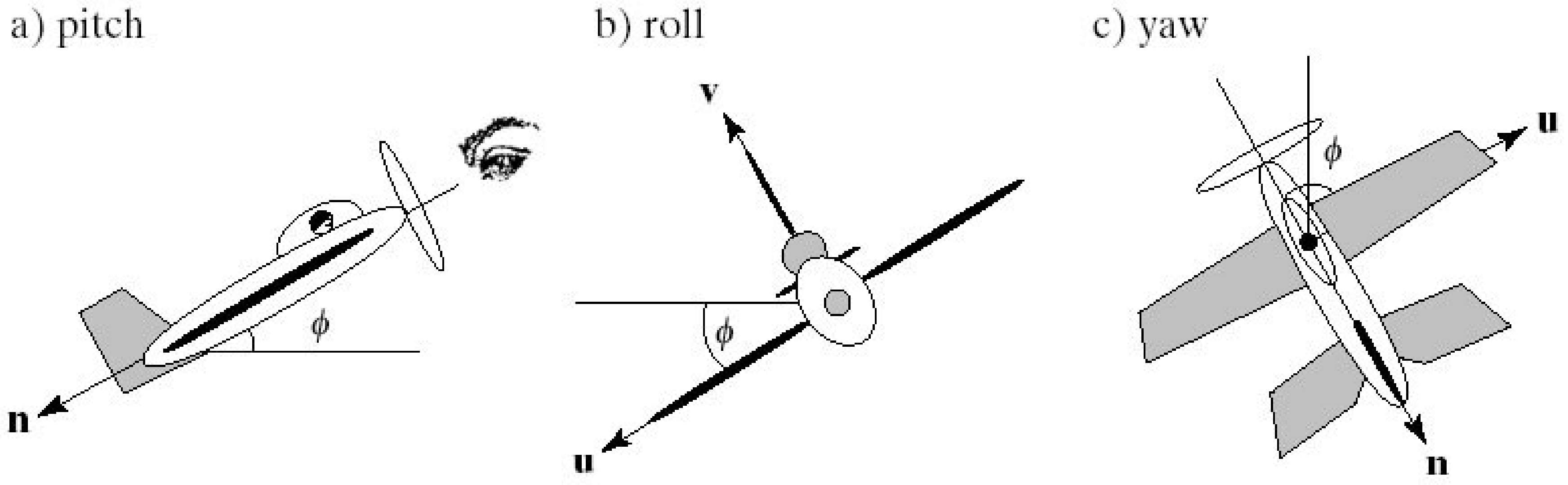


Example

```
glMatrixMode (GL_PROJECTION);
   // set the view volume (world coordinates)
glLoadIdentity();
glOrtho (-3.2, 3.2, -2.4, 2.4, 1, 50);
glMatrixMode (GL_MODELVIEW);
   // place and aim the camera
glLoadIdentity ();
gluLookAt (4, 4, 4, 0, 1, 0, 0, 1, 0);
   // modeling transformations go here
```

Changing Camera Orientation

- We can think of the jib camera as behaving like an airplane.
  - It can pitch, roll, or yaw from its position.



Changing Camera Orientation (2)

- **Pitch** – the angle between the longitudinal axis and world horizontal.
- **Roll –** the angle between the transverse axis and the world.
- **Yaw –** motion of the longitudinal axis causing a change in the direction of the plane’s flight.

Drawing 3D Shapes in OpenGL

- GLUT provides several 3D objects: a sphere, a cone, a torus, the five Platonic solids, and the teapot.
- Each is available as a **wireframe** model (one appearing as a collection of wires connected end to end) and as a solid model with faces that can be shaded.
- All are drawn by default centered at the origin.
- To use the solid version, replace Wire by Solid in the functions.

Drawing 3D Shapes in OpenGL (2)

- **cube:** glutWireCube (GLdouble size);
  - Each side is of length size.
- **sphere:** glutWireSphere (GLdouble radius, GLint nSlices, GLint nStacks);
  - nSlices is the number of "orange sections" and nStacks is the number of disks.
  - Alternately, nSlices boundaries are longitude lines and nStacks boundaries are latitude lines.

Drawing 3D Shapes in OpenGL (3)

- **torus:** glutWireTorus (GLdouble inRad, GLdouble outRad, GLint nSlices, GLint nStacks);
- **teapot:** glutWireTeapot (GLdouble size);
  - Why teapots? A standard graphics challenge for a long time was both making a teapot look realistic and drawing it quickly.

Drawing 3D Shapes in OpenGL (4)

- **tetrahedron:** glutWireTetrahedron ();
- **octahedron:** glutWireOctahedron ();
- **dodecahedron:** glutWireDodecahedron ();
- **icosahedron:** glutWireIcosahedron ();
- **cone:** glutWireCone (GLdouble baseRad, GLdouble height, GLint nSlices, GLint nStacks);

Drawing 3D Shapes in OpenGL (5)

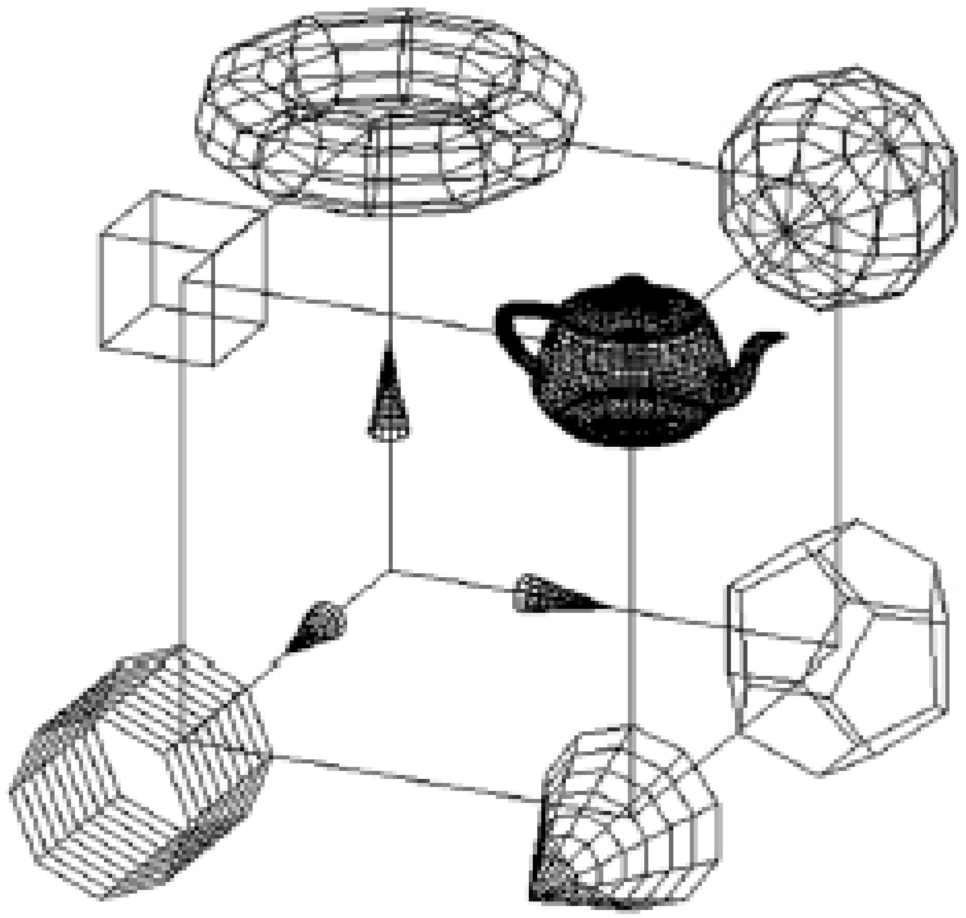
- **tapered cylinder:** gluCylinder (GLUquadricObj * qobj, GLdouble baseRad, GLdouble topRad, GLdouble height, GLint nSlices, GLint nStacks);
- The **tapered cylinder** is actually a *family* of shapes, distinguished by the value of topRad.
  - When topRad is 1, there is no taper; this is the classic **cylinder**.
  - When topRad is 0, the tapered cylinder is identical to the **cone**.

Drawing 3D Shapes in OpenGL (6)

- To draw the tapered cylinder in OpenGL, you must 1) define a new quadric object, 2) set the drawing style (GLU_LINE: wireframe, GLU_FILL: solid), and 3) draw the object:

```
GLUquadricObj * qobj = gluNewQuadric ();
    // make a quadric object
gluQuadricDrawStyle (qobj,GLU_LINE);
    // set style to wireframe
gluCylinder (qobj, baseRad, topRad, nSlices,
    nStacks);    // draw the cylinder
```

Example



Code for Example (Fig. 5.57)

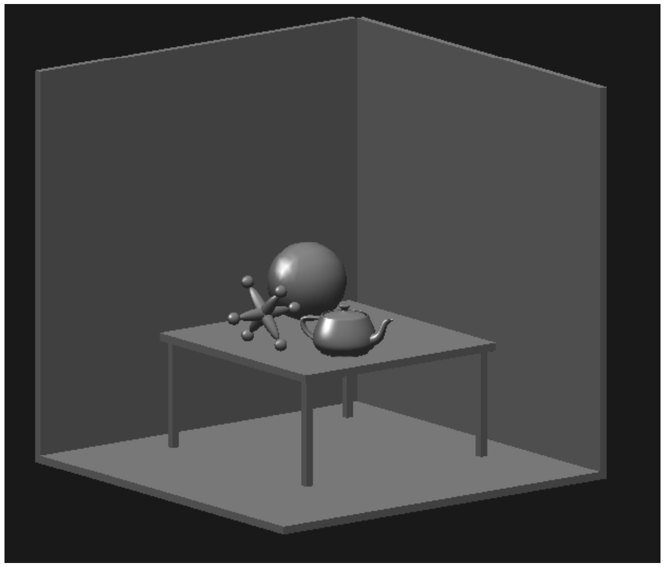
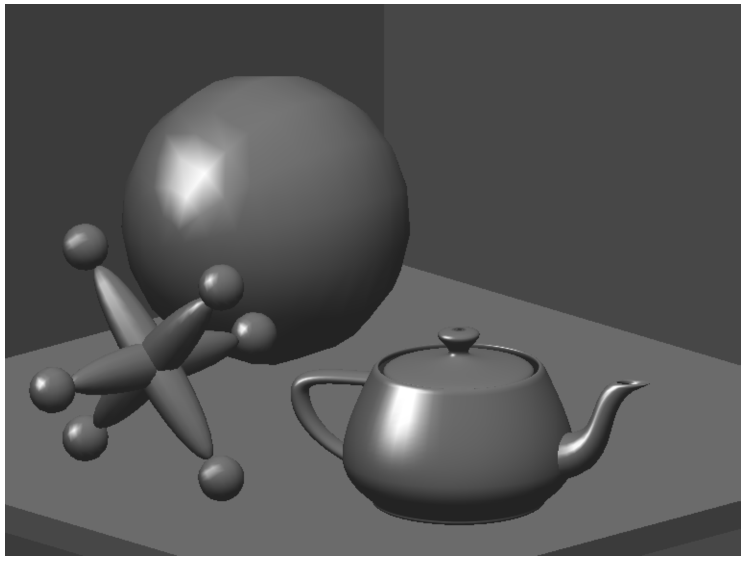
- The main() routine initializes a 640 by 480 pixel screen window, sets the viewport and background color, and specifies the drawing function as displayWire().
- In displayWire() the camera shape and position are established and each object is drawn using its own modeling matrix.
- Before each modeling transformation, a glPushMatrix() is used to remember the current transformation, and after the object has been drawn, this prior current transformation is restored with a glPopMatrix().

Code for Example (2)

- Thus the code to draw each object is imbedded in a glPushMatrix(), glPopMatrix() pair.
- To draw the *x*-axis, the *z*-axis is rotated 90^o about the *y*-axis to form a rotated system, and the axis is redrawn in its new orientation.
- This axis is drawn without immersing it in a glPushMatrix(), glPopMatrix() pair, so the rotation to produce the *y*-axis takes place in the already rotated coordinate system.

Solid 3D Drawing in OpenGL

- A solid object scene is rendered with shading. The light produces highlights on the sphere, teapot, and jack.

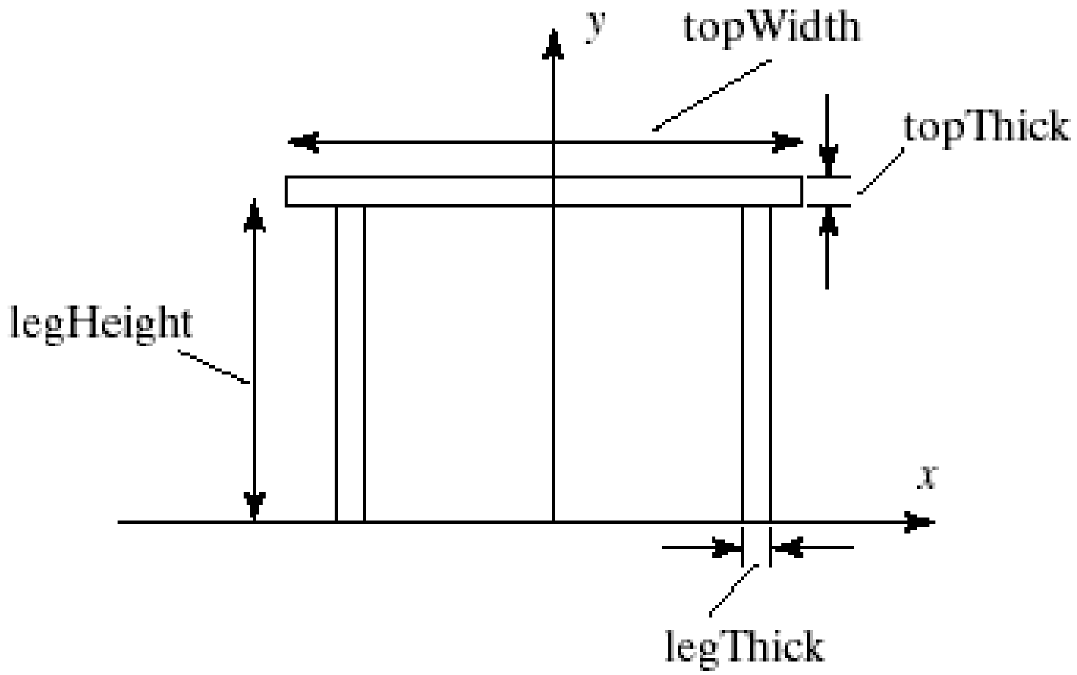
Solid 3D Drawing in OpenGL (2)

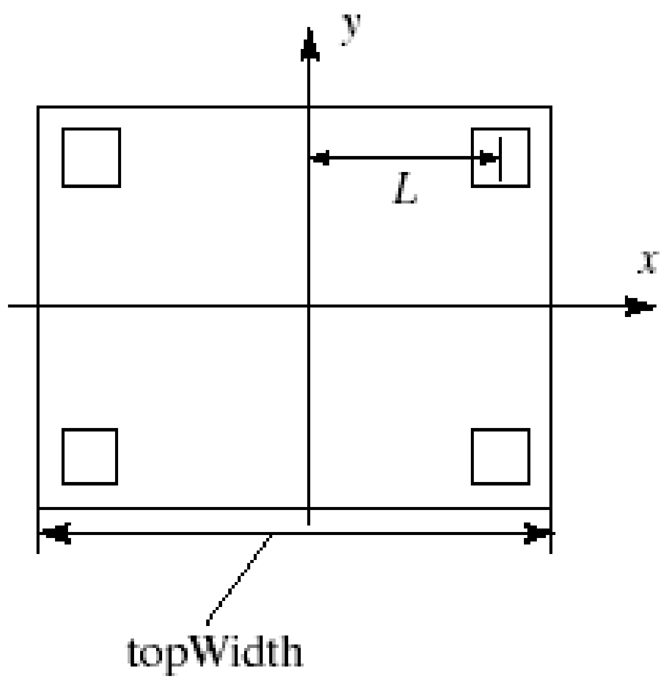
- The scene contains three objects resting on a table in the corner of a room.
- The three walls are made by flattening a cube into a thin sheet and moving it into position.
- The jack is composed of three stretched spheres oriented at right angles plus six small spheres at their ends.

Solid 3D Drawing in OpenGL (3)

- The table consists of a table top and four legs.
- Each of the table’s five pieces is a cube that has been scaled to the desired size and shape (next slide).
- The table is based on four parameters that characterize the size of its parts: topWidth, topThick, legLen, and legThick.

Table Construction





Solid 3D Drawing in OpenGL (4)

- A routine tableLeg() draws each leg and is called four times within the routine table() to draw the legs in the four different locations.
- The different parameters used produce different modeling transformations within tableLeg(). As always, a glPushMatrix(), glPopMatrix() pair surrounds the modeling functions to isolate their effect.

Code for the Solid Example (Fig. 5.60)

- The solid version of each shape, such as glutSolidSphere(), is used.
- To create shaded images, the position and properties of a light source and certain properties of the objects’ surfaces must be specified, in order to describe how they reflect light (Ch. 8).
- We just present the various function calls here; using them as shown will generate shading.

Scene Description Language (SDL)

- Previous scenes were described through specific OpenGL calls that transform and draw each object, as in the following code:

```
glTranslated (0.25, 0.42, 0.35);
glutSolidSphere (0.1, 15, 15); // draw a
   sphere
```

- The objects were “hard-wired” into the program. This method is cumbersome and error-prone.

SDL (2)

- We want the designer to be able to specify the objects in a scene using a simple language and place the description in a file.
- The drawing program becomes a general-purpose program:
  - It reads a scene file at run-time and draws whatever objects are encountered in the file.

SDL (3)

- The **Scene Description Language (SDL)**, described in Appendix 3, provides a Scene class, also described in Appendix 3 and on the book’s web site, that supports the reading of an SDL file and the drawing of the objects described in the file.

Using SDL

- A <u>global</u> Scene object is created:

  Scene scn; // create a scene object
- Read in a scene file using the read method of the class:

  scn.read("example.dat"); // read the scene file & build an object list

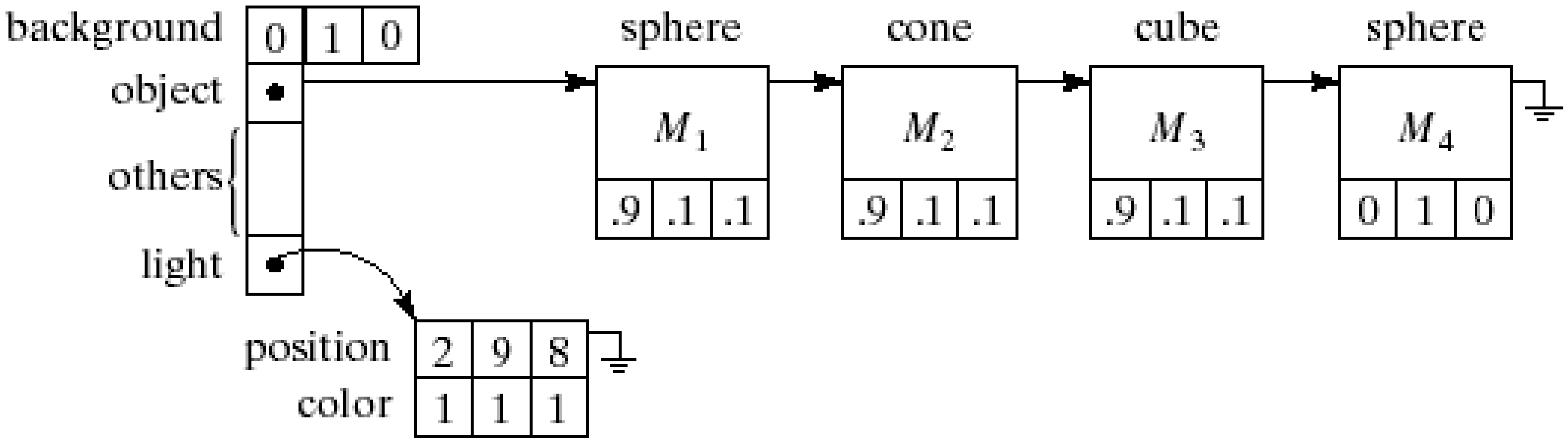
Example SDL Scene

```
! example.dat: simple scene: 1 light and 4 shapes
! beginning ! is a comment; extends to end of line
background 0 0 1          ! create a blue background
light 2 9 8 1 1 1          ! put a white light at (2, 9, 8)
diffuse .9 .1 .1      ! make following objects reddish
translate 3 5 –2   sphere    ! put a sphere at 3 5 –2
translate –4 –6 8  cone    ! put a cone in the scene
translate  1 1 1 cube                        ! add a cube
diffuse 0 1 0           ! make following objects green
translate 40 5 2 scale .2 .2 .2 sphere   ! tiny sphere
```

The SDL Scene

- The scene has a bright blue background color (*red*, *green*, *blue*) = (0, 0, 1), a bright white (1, 1, 1) light situated at (2, 9, 8), and four objects: two spheres, a cone and a cube.
- The light field points to the list of light sources, and the obj field points to the object list.
- Each shape object has its own affine transformation *M* that describes how it is scaled, rotated, and positioned in the scene. It also contains various data fields that specify its material properties. Only the diffuse field is shown in the example.

SDL Data Structure



The SDL Scene (2)

- Once the light list and object list have been built, the application can render the scene:

```
scn.makeLightsOpenGL(),
scn.drawSceneOpenGL();  // render scene with OpenGL
```

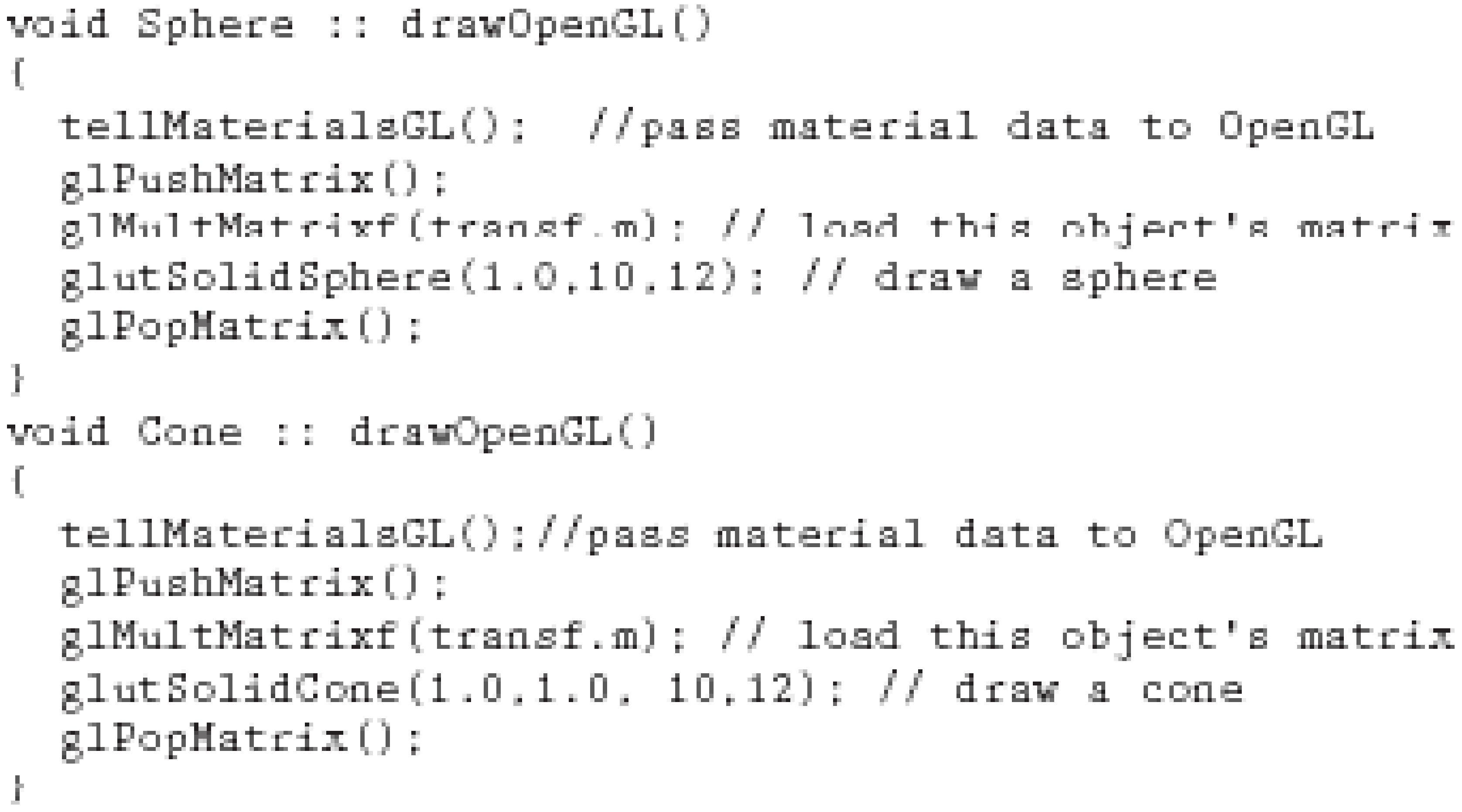
- The first instruction passes a description of the light sources to OpenGL. The second uses the method drawSceneOpenGL() to draw each object in the object list.
- The code for this method is very simple:

```
void Scene :: drawSceneOpenGL()
{     for(GeomObj* p = obj; p ; p = p->next)
          p->drawOpenGL(); // draw it
}
```

The SDL Scene (3)

- The function moves a pointer through the object list, calling drawOpenGL() for each object in turn.
- Each different shape can draw itself; it has a method drawOpenGL() that calls the appropriate routine for that shape (next slide).
- Each first passes the object’s material properties to OpenGL, then updates the modelview matrix with the object’s specific affine transformation.
- The original modelview matrix is pushed and later restored to protect it from being affected after this object has been drawn.

Examples of Objects which can Draw Themselves



```
void Sphere :: drawOpenGL()
{
  tellMaterialsGL();  //pass material data to OpenGL
  glPushMatrix();
  glMultMatrixf(transf.m); // load this object's matrix
  glutSolidSphere(1.0,10,12); // draw a sphere
  glPopMatrix();
}
void Cone :: drawOpenGL()
{
  tellMaterialsGL();//pass material data to OpenGL
  glPushMatrix();
  glMultMatrixf(transf.m); // load this object's matrix
  glutSolidCone(1.0,1.0, 10,12); // draw a cone
  glPopMatrix();
}
```

Using the SDL

- Fig. 5.63 shows the code to read in an SDL file and draw it.
- Fig. 5.64 shows the SDL file necessary to draw the solid objects picture.
- It is substantially more compact than the corresponding OpenGL code file.
  - Note also that some functions in the SDL may have to be implemented by you!