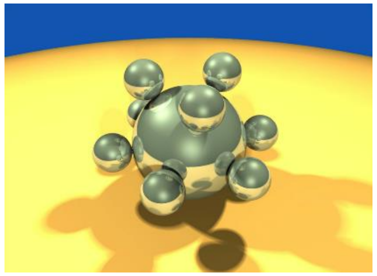
Computer Graphics using Open GL, 3rd Edition

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.

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

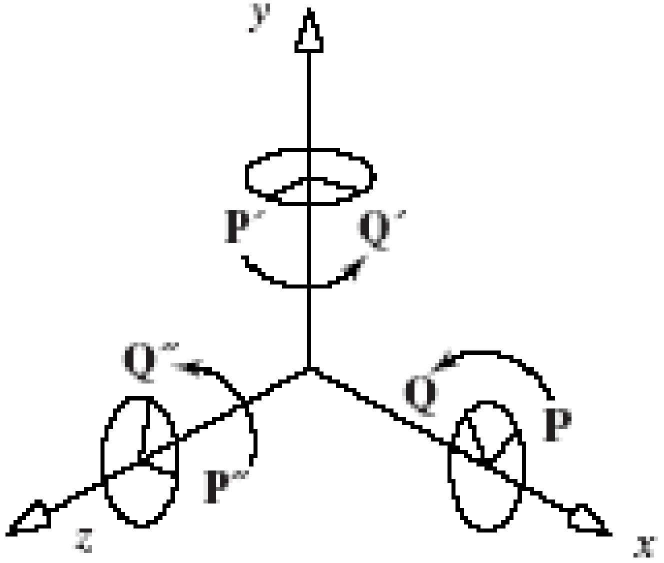
Shear

- 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.

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

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

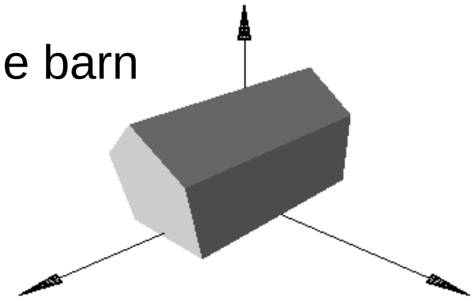
Rotations (3)

- 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.

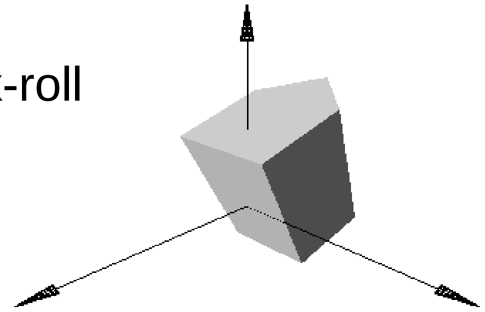
Example

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

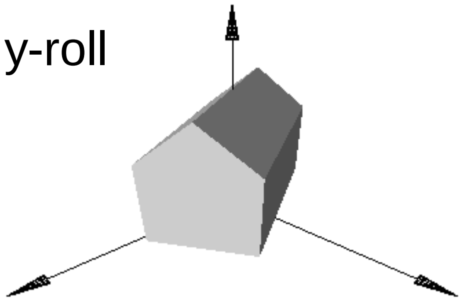
a). the barn



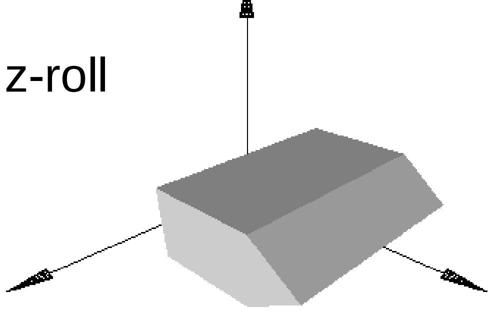
b). -70^0 x-roll



c). 30^0 y-roll



d). -90^0 z-roll

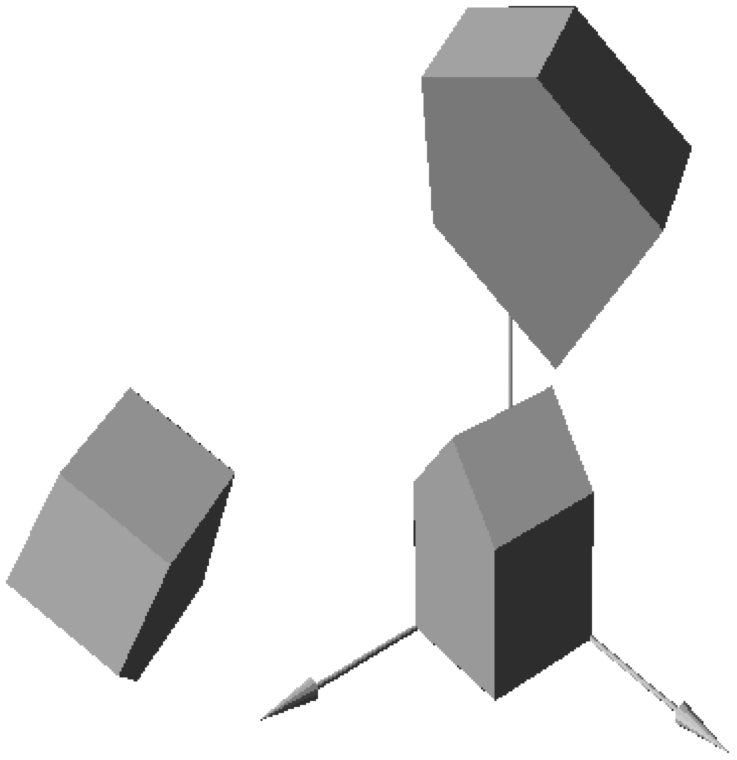


Composing 3D Affine Transformations

- 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: $\mathrm{M} = \mathrm{M}_2\mathrm{M}_1$
- 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)

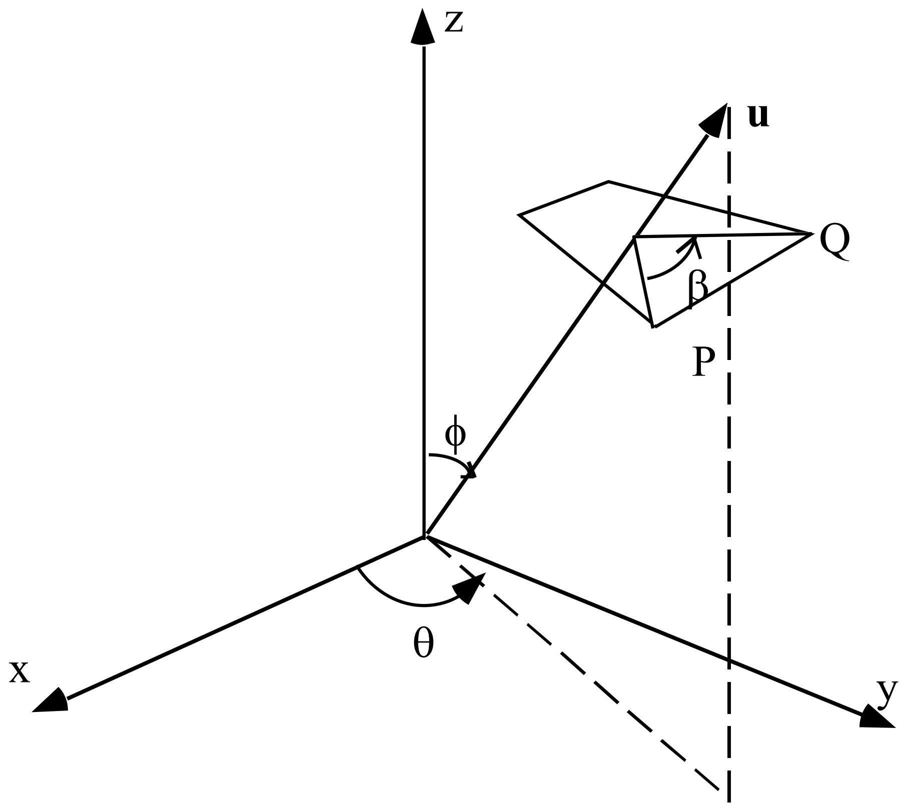
- We build a rotation in 3D by composing three elementary rotations: an *x*-roll followed by a *y*-roll, and then a *z*-roll. The overall rotation is given by $M = R_z(\beta_3)R_y(\beta_2)R_x(\beta_1)$.
- In this context the angles β_1, β_2, and β_3 are often called **Euler angles**.

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.

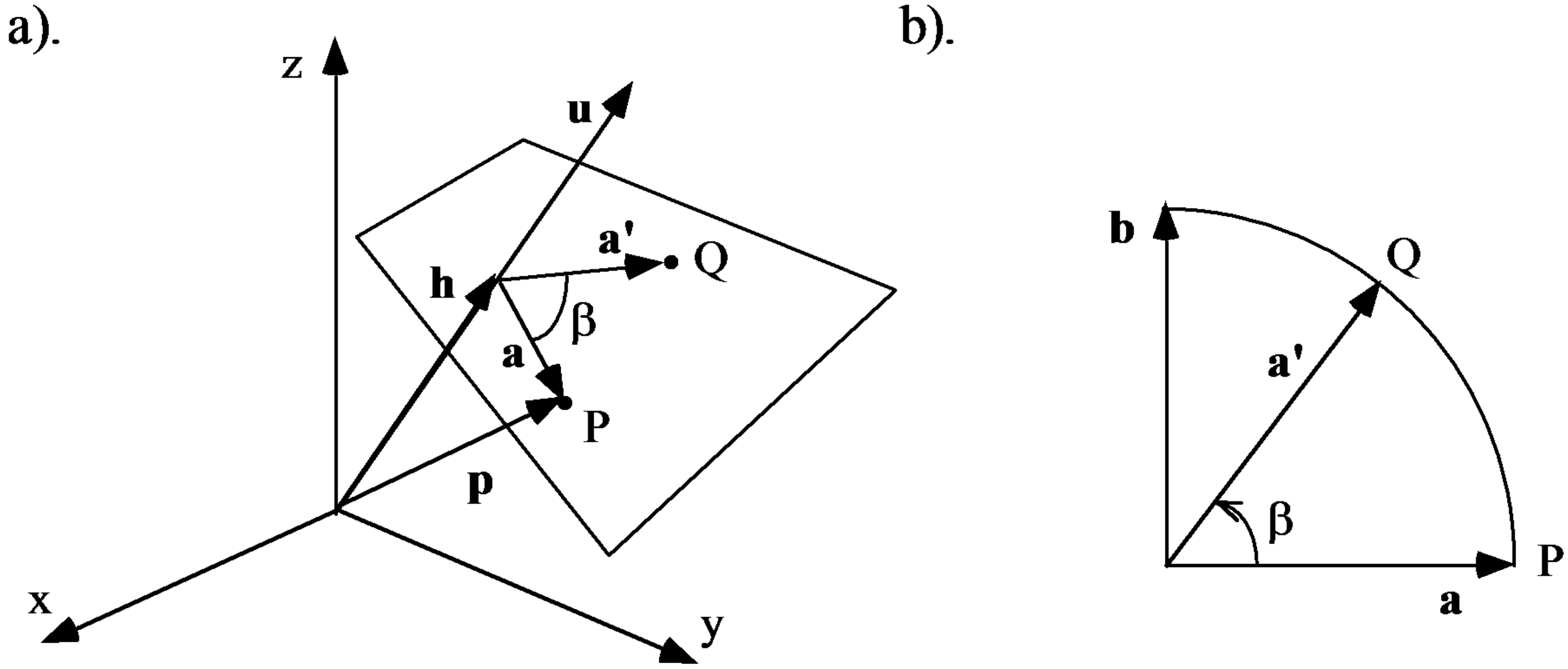
Rotating about an Arbitrary Axis (3)

- **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)



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

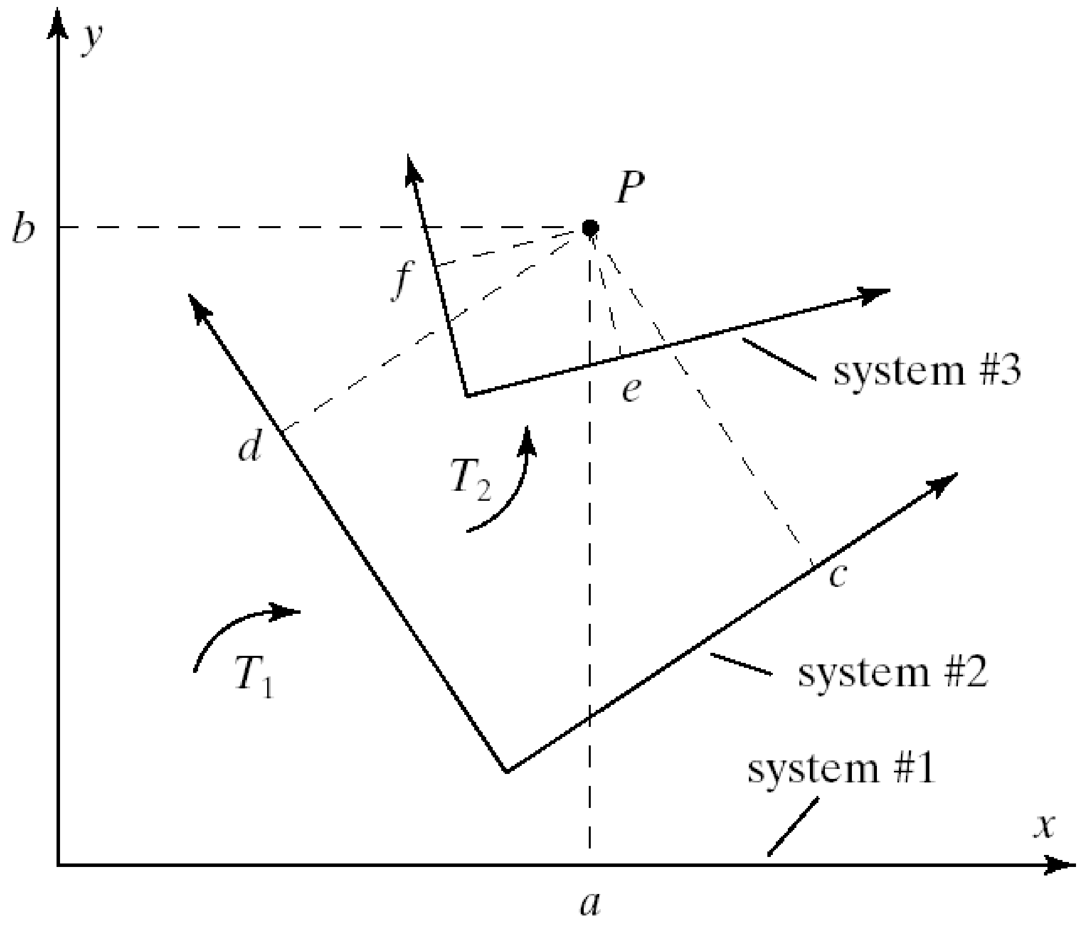
- **Affine transformations** preserve **affine combinations of points**.
- **Affine transformations preserve lines and planes.**
- **Parallelism of lines and planes is preserved.**
- **The columns of the matrix reveal the transformed coordinate frame.**
- **Relative ratios are preserved.**

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)

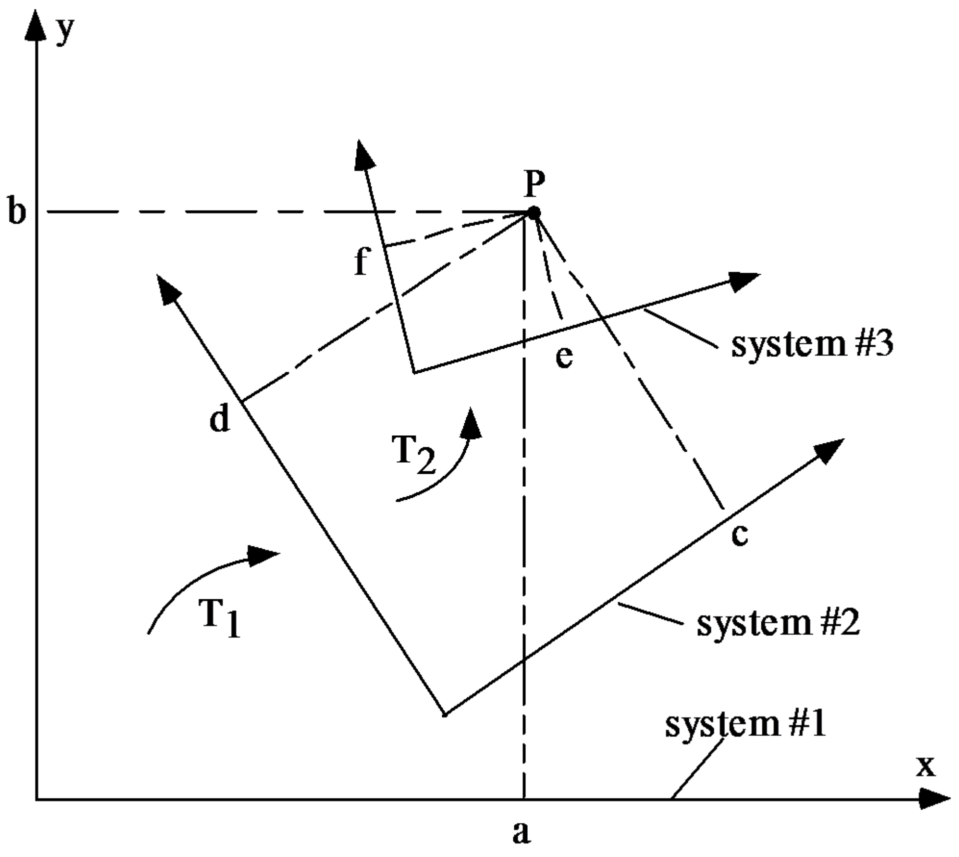
- We have the following theorem:
- Suppose coordinate system #2 is formed from coordinate system #1 by the affine transformation *M*. Further suppose that point $P = (P_x, P_y, P_z, 1)$ are the coordinates of a point *P* expressed in system #2. Then the coordinates of *P* expressed in system #1 are *MP*.

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 $(\mathrm{a}, \mathrm{b}, 1)^T = \mathrm{M}_1(\mathrm{d}, \mathrm{c}, 1)^T = \mathrm{M}_1\mathrm{M}_2(\mathrm{e}, \mathrm{f}, 1)^T$
- 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 x M_2 x 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 x M_2 x 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

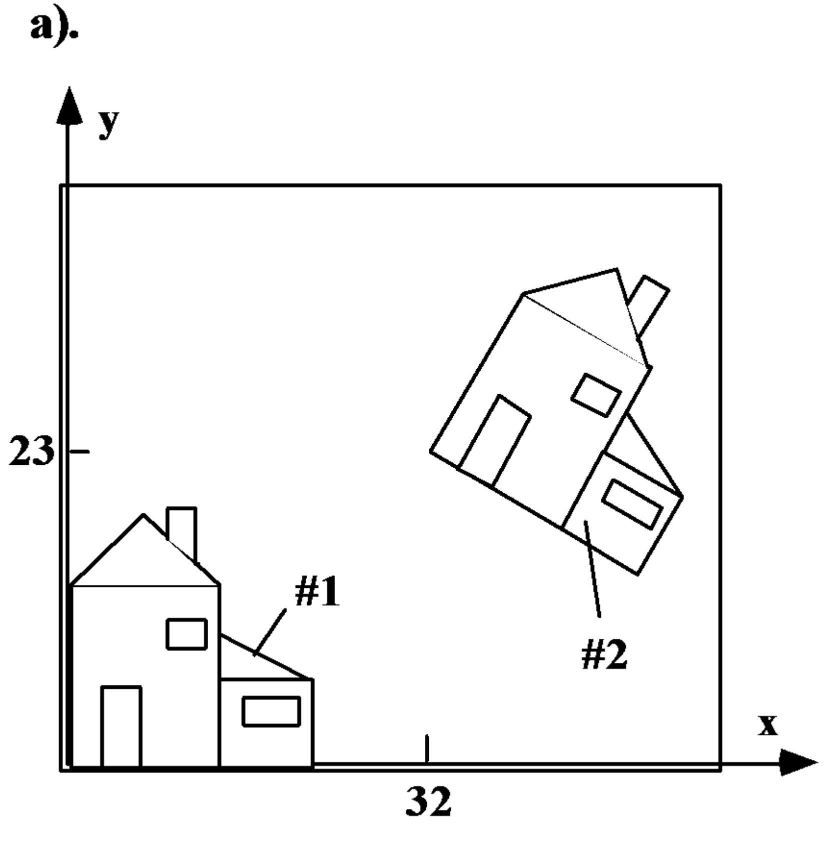
- Open-GL actually transforms coordinate systems, so in your programs you will have to apply the transformations in reverse order.
- E.g., if you want to translate the 3 vertices of a triangle and then rotate it, your program will have to do rotate and then translate.

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.



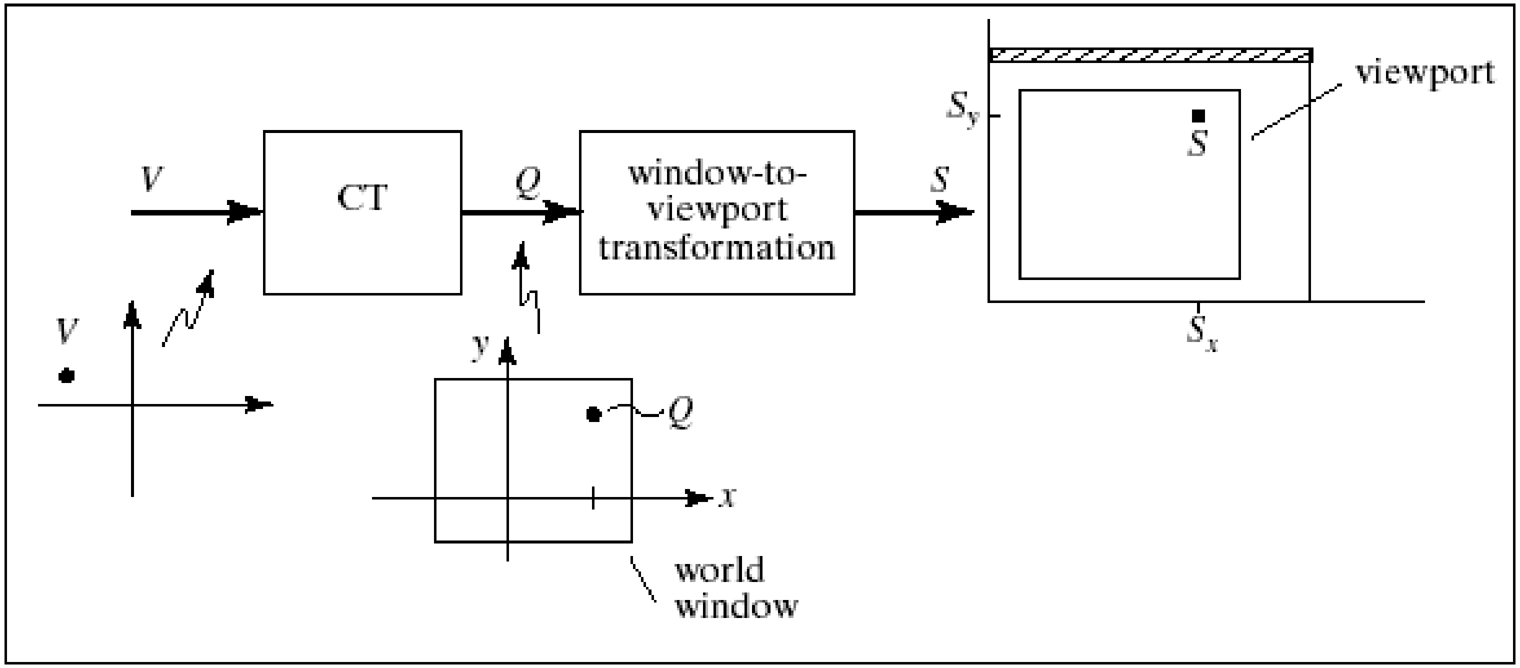
Example: the Easy Way (2)

- 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.

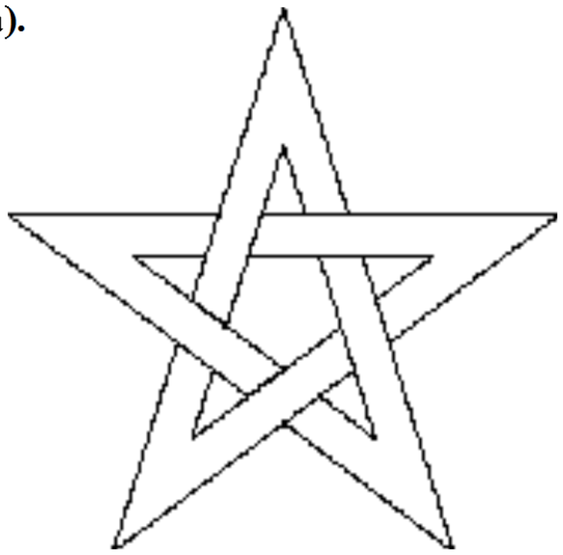
Example (8)

- 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

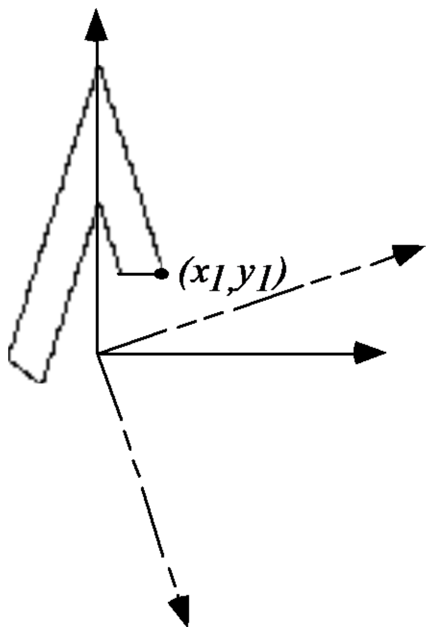
Example 2: Star

- 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).



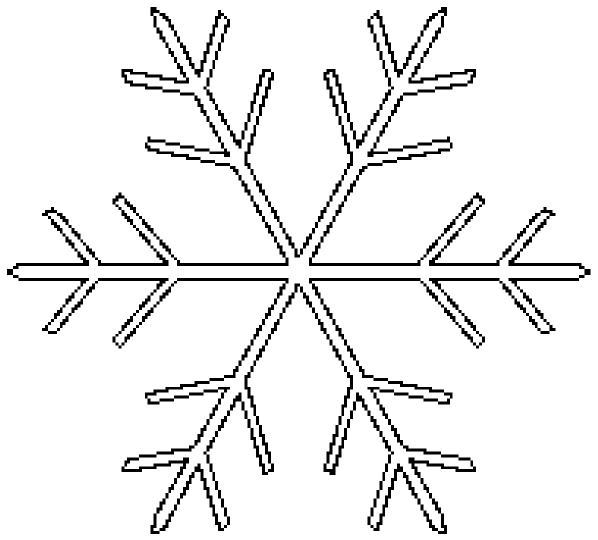
b).



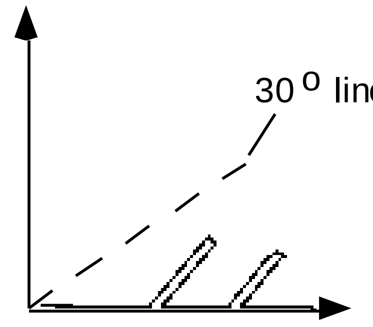
Example 3: Snowflake

- 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).

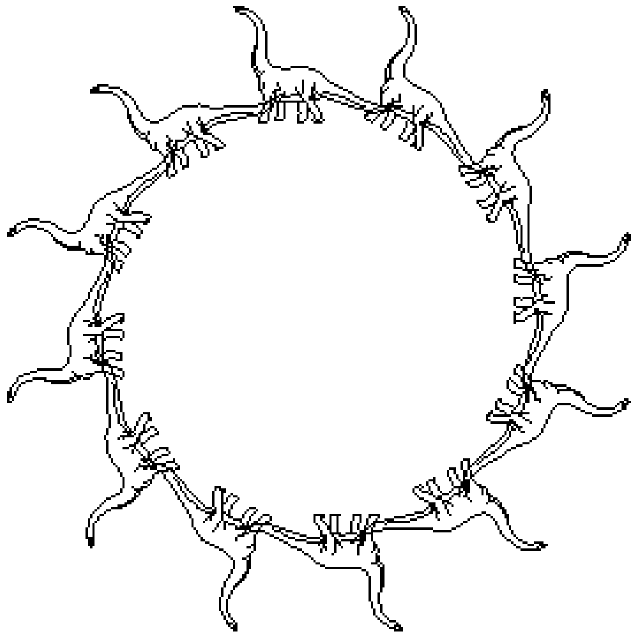


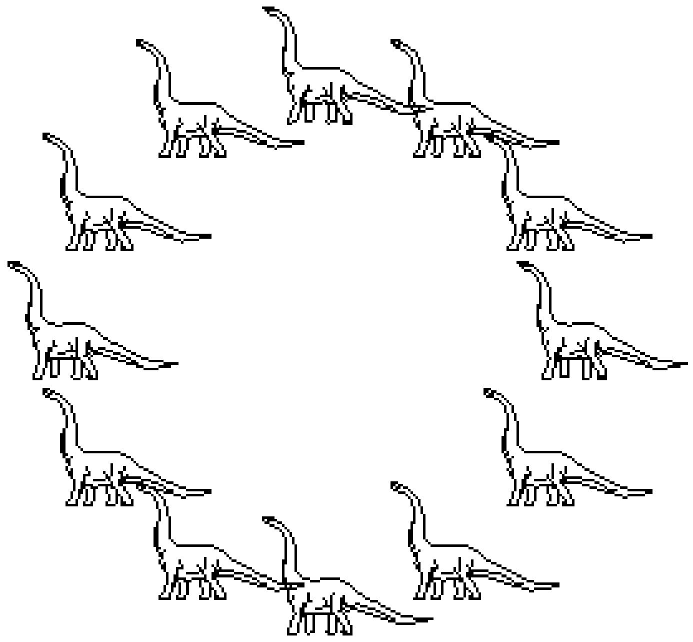
b).



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.





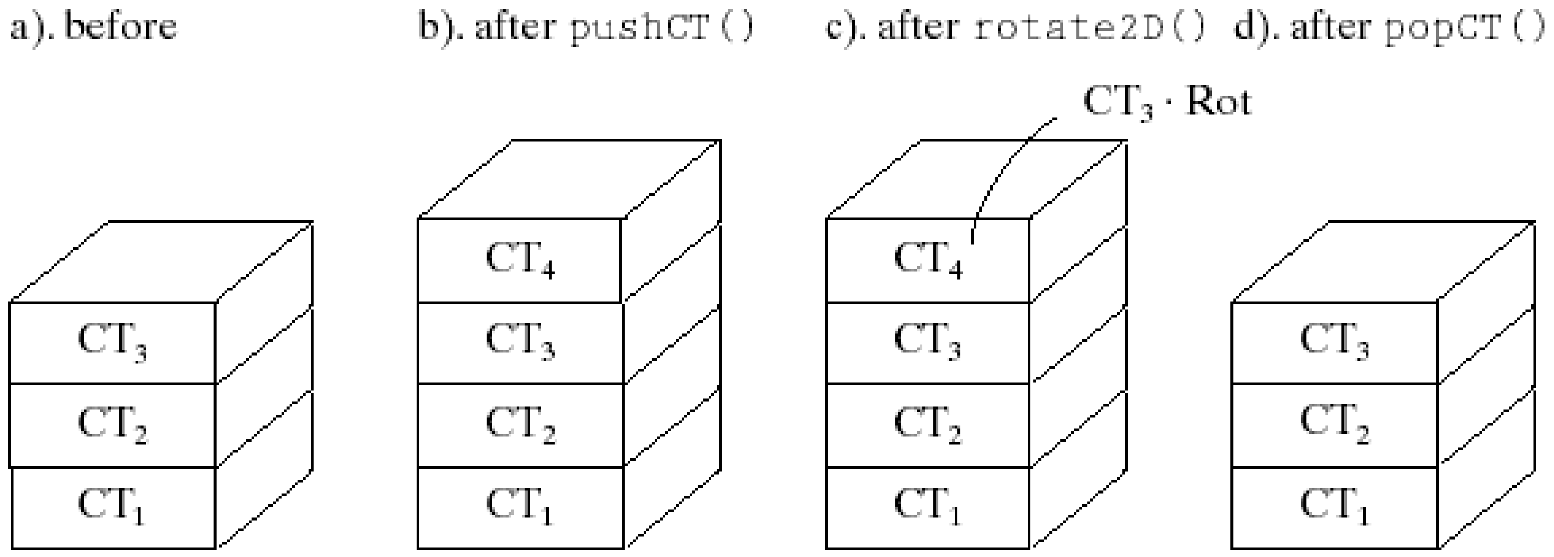
Example 4 (2)

- 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.