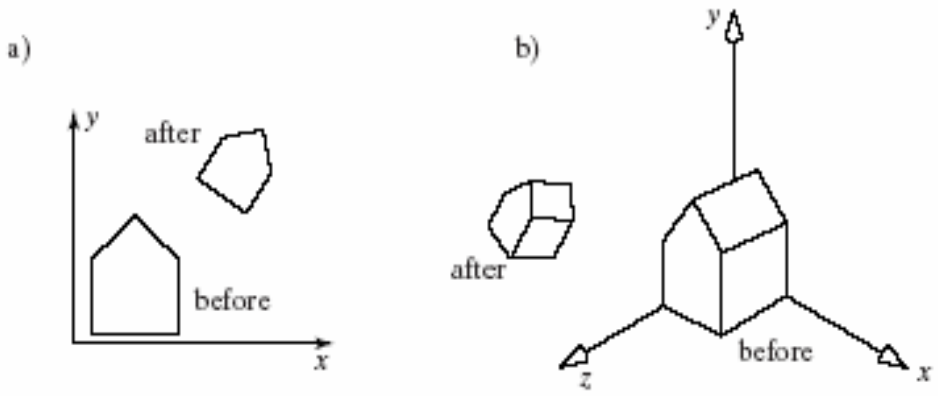
Prof Reuven Aviv Department of Computer Science
Tel Hai Academic College
Computer Graphics

Transformations

Slides adapted from F. Hill, S. Kelley Computer Graphics

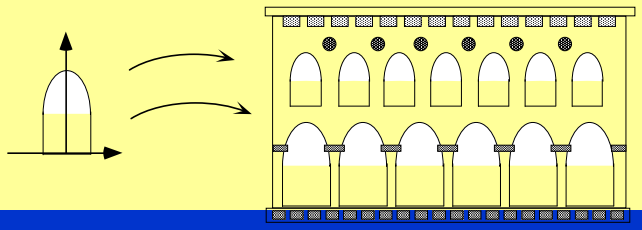
Examples of Affine Transformations

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



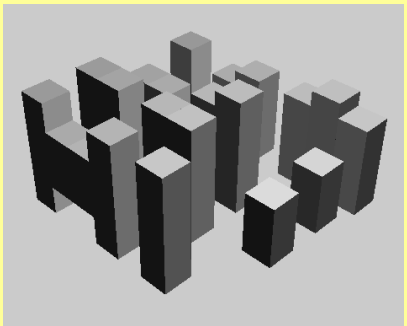
Using Transformations

- One arch is designed
- The scene is drawn by placing copies of instances of the arch at different places and with different sizes.



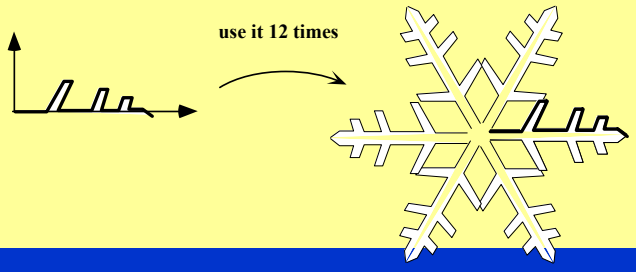
Using Transformations (2)

- In 3D, many cubes make a city.



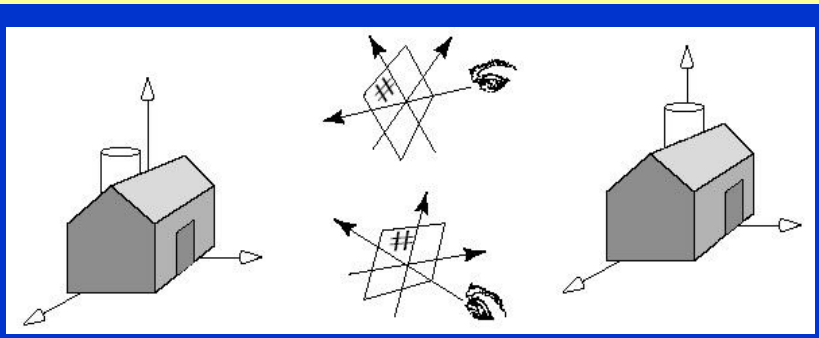
Using Transformations (3)

- 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.



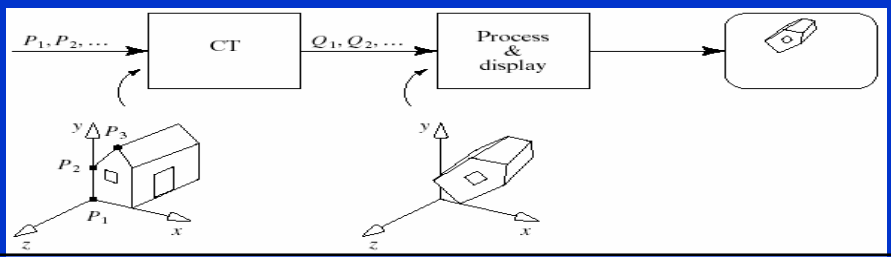
The Graphics Pipeline

- Application sends to OpenGL a series of points P_i

  glBegin(GL_LINES);

  glVertex3f(...); glVertex3f(...); ... glEnd();

- Points are changed by **the current transformation** into a different set of points, say Q_1, Q_2, Q_3.
- The new points further processed, then displayed

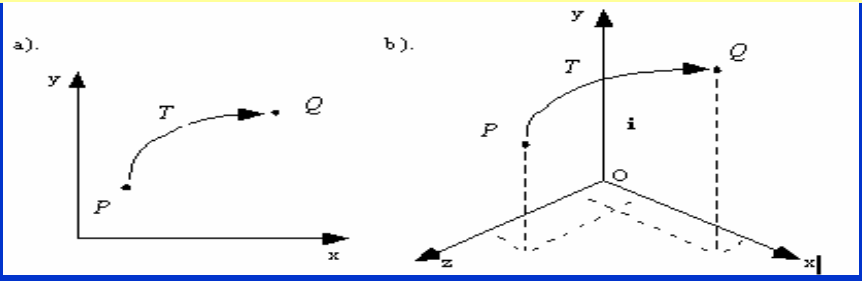


Transformations

- 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

Object Transformations

- (2D or 3D) transformation: **P** → **Q**, **Q**= *T*(**P**)
- Object Transformation: all points by same T()
- *straight line transform to a straight line?*
- In general, no
  - Affine (linear) transformation preserve lines



General 2D Transformations

- We transform points to points

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

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = T\left(\begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}\right)$$

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

Affine Transformation in the plane

- **Point & vector represented by a column matrix**
  - **Point, vector: third component always 1, 0**
- **transformation: matrix multiplication**
- **Affine matrix: third row 0, 0, 1**

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

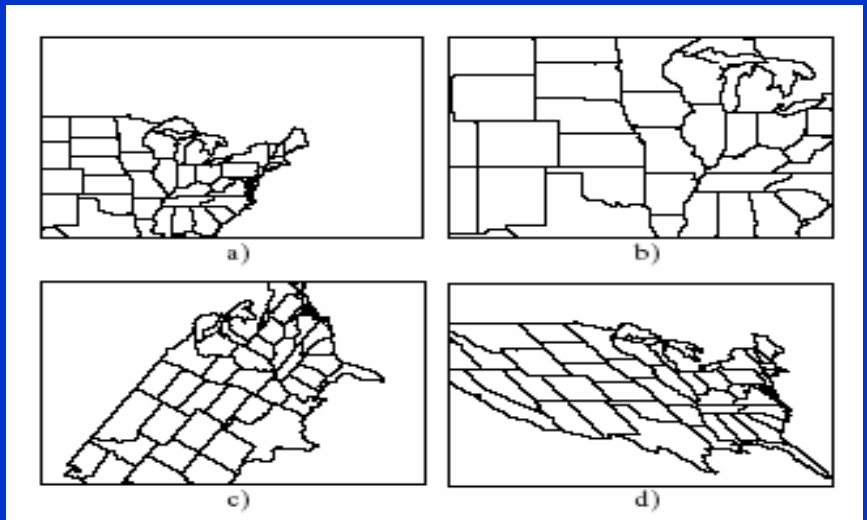
Affine Transformations (2)

- If points or vector represented by row matrices
  - $P = (P_x, P_y, 1)$
- Transformation: $Q = P\, M^T$
- $Q = (Q_x, Q_y, 1)$

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

Elementary Affine Transformations

- translation, scaling, rotation, shear



2D Translations

- *How points are translated?*
- *How vectors are translated?*
- To translation of a point **P** by a in the x direction and b in the y direction:

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

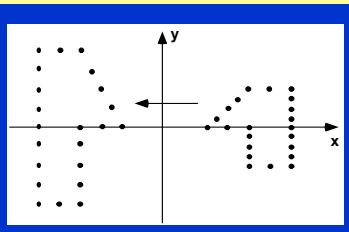
- using homogeneous coordinates allow us to include translation as an affine transformation.

2D Scaling

- *What is scaling?*
- Scaling distances from the origin, along axes
- If $S_x = S_y$ Uniform Scaling; magnification factor S_x
- If $Sx \neq Sy$: Differential Scaling. distorts the image.
- If S_x or $S_y < 0$, image reflected across the x or y axis.

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

$S(S_x, S_y)$



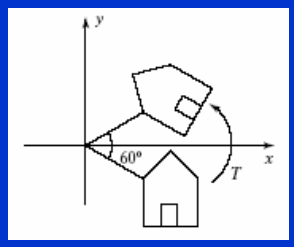
S(-1, 2)

2D Rotation

- Counterclockwise around origin by angle θ:

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

R(θ)

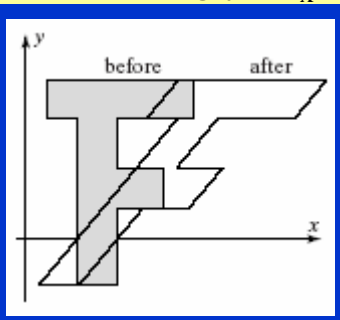


Deriving the Rotation Matrix

- Point **P** is at distance *R* from the origin, at angle Φ
  - **P:** $P_x = R\ cos(\Phi)$ $P_y = R\ sin(\Phi)$
- **Q** at the same distance as **P**, and at angle θ + Φ:
  - **Q:** $Q_x = R\ cos(\theta + \Phi)$, $Q_y = R\ sin(\theta + \Phi)$
- $cos(\theta + \Phi) = cos(\theta)\ cos(\Phi) - sin(\theta)\ sin(\Phi)$
- $sin(\theta + \Phi) = sin(\theta)\ cos(\Phi) + cos(\theta)\ sin(\Phi)$
- Eliminate the Φ terms, express (Q_x, Q_y) in terms of P_x, P_y.

Shear

- *What is shear?*
- Shear along the x direction: $Q_x = P_x + h*P_y$ $Q_y = P_y$
  - Amount of translation along x proportional to P_y
  - *Example: italic* transformation
- Shear along y: $Q_x = P_x$; $Q_y = g*P_x + P_y$



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

H(h, g)

Inverse Translation and Scaling

$Tr^{-1}(t_x, t_y) = Tr(-t_x, -t_y)$

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

$S^{-1}(s_x, s_y) = S(1/s_x, 1/s_y)$

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

Inverse Rotation and Shear

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

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

Composing 2D Transformations: Examples

- *How to rotate around a point* **V**, *by an angle* θ
- 1. Translate, so that **V** moves to the origin. **Tr(-$\underline{\mathbf{v}}$)**
- 2. rotate. R(θ)
- 3. Translate, so that **V** moves back.
  - Q = Tr($\underline{\mathbf{v}}$) R(θ) Tr(-$\underline{\mathbf{v}}$)

- *How to shear around an arbitrary point* **V?**
- Q = Tr($\underline{\mathbf{v}}$) H(f,g) Tr(-$\underline{\mathbf{v}}$)

- *How to scale about an arbitrary point* **V?**
- Q = Tr($\underline{\mathbf{v}}$)S(S_x, S_y)Tr(-$\underline{\mathbf{v}}$)

- what is the vector $\underline{\mathbf{v}}$?

Composing Affine Transformations (Examples)

- *How to reflect across an arbitrary line through the origin?*
- Suppose the reflection line has angle θ with X axis
- M = R(θ) S(1, -1) R(-θ)
- First rotate, so that the reflection line goes to the x-axis,
- Next, do reflection relative to X axis (scaling)
- Then, rotate, so that reflection line goes to original location

- How to perform the Window - viewport Transformation:
- Translate by -w.l, -w.b, scale by A, B, translate by v.l, v.b.

Properties of 2D and 3D Affine Transformations

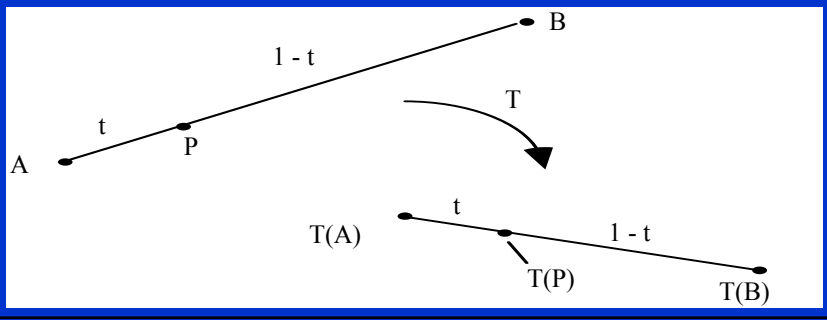
- *preserve* affine combinations of points.
  - $\mathbf{W} = a_1\mathbf{P}_1 + a_2\mathbf{P}_2$ is an affine combination
  - $M\mathbf{W} = a_1M\mathbf{P}_1 + a_2M\mathbf{P}_2$
- Result M preserve lines and planes. *Why*
- A line AB an affine combination of points (**A**, **B**)
  - $\mathbf{L}(t) = (1-t)\mathbf{A} + t\mathbf{B}$
- A plane is an affine combination of points:
  - $\mathbf{P}(s, a) = s\mathbf{A} + t\mathbf{B} + (1 - s - t)\mathbf{C}$.

Parallelism of lines and planes is preserved

- Line $A + \mathbf{b}t$ having direction **b**
- transform to $M(\mathbf{A} + \mathbf{b}t) = M\mathbf{A} + M\mathbf{b}t$, direction $M\mathbf{b}$.
  - Direction independent of **A**
- two different lines $A_1 + \mathbf{b}t$ and $A_2 + \mathbf{b}t$ will transform into two lines both having the direction, $M\mathbf{b}$.
- *parallelograms map into other parallelograms*.

Relative Ratios are preserved

- **P** lying a fraction *t* of the way between **A** and **B**
- 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**).



Transformation of area

- When the 2D transformation with matrix *M* is applied to a 2D object, its area is multiplied by the *magnitude of the determinant* of *M*:

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

- 3D: volume of a 3D object is scaled by |det *M*| when the object is transformed by the 3D transformation based on matrix *M*.

Decomposition of affine matrix

- any 3 x 3 affine matrix can be written as the product of (reading right to left)
  - a translation matrix,
  - a rotation matrix,
  - a scaling matrix,
  - and a shear matrix.
- M = (shear)(scaling)(rotation)(translation)
- This is not the only way

3D World

- Coordinate frame: origin **O** and three mutually perpendicular axes in the directions $\underline{\mathbf{i}}$, $\underline{\mathbf{j}}$, and $\underline{\mathbf{k}}$
- Point **P** in this frame is given by
  - $\mathbf{P} = \mathbf{O} + P_x\underline{\mathbf{i}} + P_y\underline{\mathbf{j}} + P_z\underline{\mathbf{k}}$,
  - a vector $\underline{\mathbf{V}}$ is given by $V_x\underline{\mathbf{i}} + V_y\underline{\mathbf{j}} + V_z\underline{\mathbf{k}}$.

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

3-D Affine Transformations

- The matrix representing a transformation is 4 x 4
- fourth row 0, 0, 0, 1

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

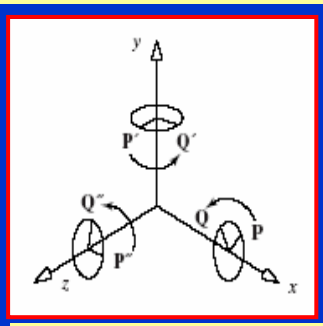
Translation, Scaling and Shearing

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

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

x-roll, y-roll, z-roll

- rotation **counter-clockwise** around an axis looking **toward** the origin



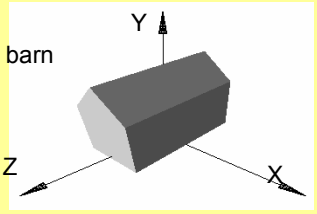
$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\vartheta & -\sin\vartheta & 0 \\ 0 & \sin\vartheta & \cos\vartheta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, R_y = \begin{pmatrix} \cos\vartheta & 0 & \sin\vartheta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\vartheta & 0 & \cos\vartheta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

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

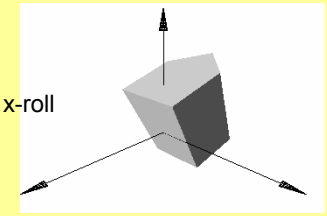
- 1's and 0's row & column of rotation axis
- *cos* and *sin* in a rectangular pattern
  - in the other rows and columns

Examples

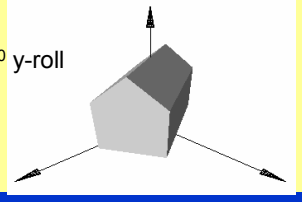
a). the barn



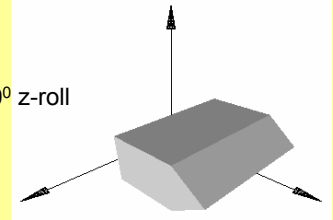
b). -70^0 x-roll



c). 30^0 y-roll

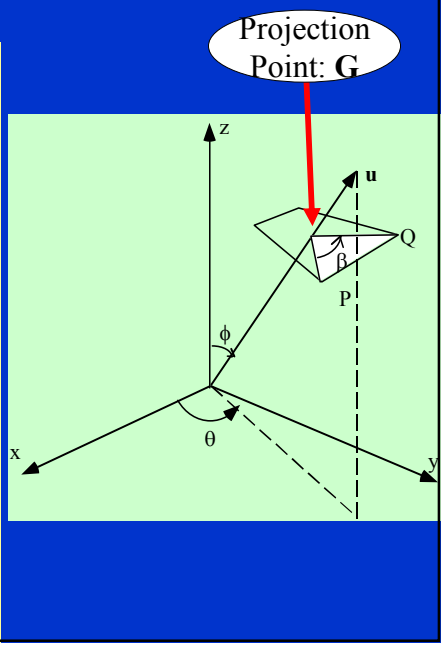


d). -90^0 z-roll



Rotating about an Arbitrary Axis

- Goal: rotate around axis **u**
- make **P** coincide with **Q**
  - Assume β is given
  - *How to calculate β?*
- **u** can have any direction
- Direction: polar angles
  - Φ is given
  - Θ is given
- *By 5 rolls – how?*



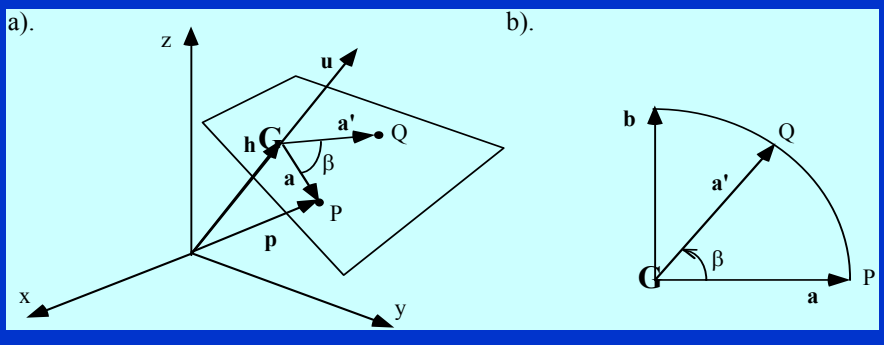
The Classic Construction of rotation around arbitrary axis **u**, by angle β. 5 rolls

- Do a Z roll so that **u** lies in the XZ plane: $R_z(\Phi)$
- Do a Y roll so that **u** coincides with Z axis: $R_y(\theta)$
- *Where is now the plane via* **P, Q, G** *?*
- Do a *z*-roll through angle β: $R_z(\beta)$
- Do a Y roll, then Z role to bring the **u** vector to its original direction
- $R_{\mathbf{u}}(\beta) = R_z(-\Phi)\, R_y(-\theta)\, R_z(\beta)\, R_y(\theta)\, R_z(\Phi)$

The Constructive way (highlights)

- What we need is the matrix M such that **Q** = M **P**
- *rotation plane:* the plane via **P**, **Q**, **G**
  - perpendicular to the vector **u**.
- Construct a 2D coordinate system in the rotation plane
  - Origin **G**
  - **a** vector with direction from **G** towards **P:** **a** = **P** - **G**
  - **b** is orthogonal to **a**
- Write **Q, P** as linear combinations of point **G**, **a** and **b**
- Rewrite $\mathbf{Q}_x$, $\mathbf{Q}_y$, $\mathbf{Q}_z$ as linear combinations of **Px, Py, Pz**
- All coefficients are dot and cross products of **a**, **b**, **u**

Constructive Way: Result

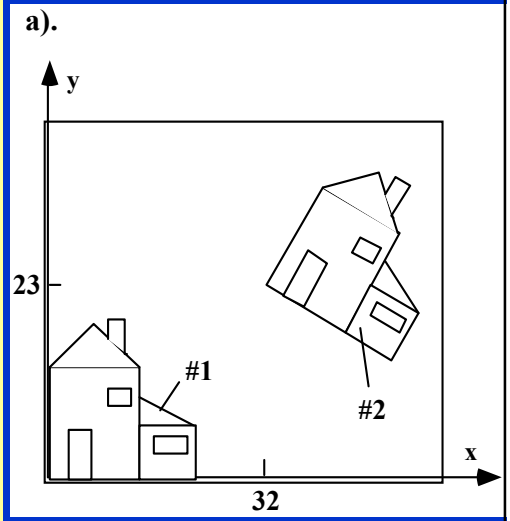


$$R_u(\beta) = \begin{pmatrix} c+(1-c)u_x^2 & (1-c)u_yu_x - su_z & (1-c)u_zu_x + su_y & 0 \\ (1-c)u_xu_y + su_z & c+(1-c)u_y^2 & (1-c)u_zu_y - su_x & 0 \\ (1-c)u_xu_z - su_y & (1-c)u_yu_z + su_x & c+(1-c)u_z^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- c = cos(β), s = sin(β), u_i components of **u**.
- Note: If matrix *R* is given, you can find **u** and β!

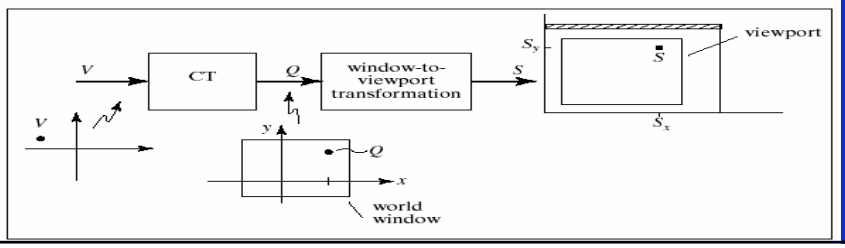
What to do in a program?

- We have house #1, we want to draw house #2.
- Need to Rotate the house, then translate
- 1. Comand OpenGL to translate
- 2. command to rotate
- Note: Reverse order!



OpenGL: The Current Transformation Matrices

- OpenGL uses **current transformation matrices** *CT*
- Here we use the *ModelView* CT, called *M*
  - M is a product of affine transformations
- OpenGL applies *M* to all subsequent primitives
  - as a part of the *graphics pipeline*
- and then applies the Window to Viewport mapping.

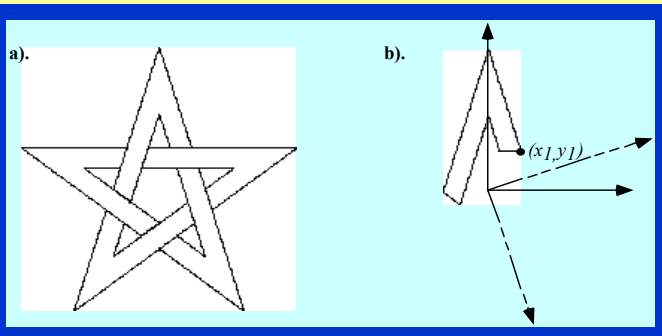


Order of Transformations

- *First Rotate, then translate: what is M?*
- *M* = *T***R*.
- This will be applied to subsequent primitives
- OpenGL always does post-multiplication on *M*
  - M can be multiplied only on the right
- Hence we specify the transformations in the reverse order of their application by OpenGL
  - Initialize M by *glLoadIdentity()* → M = I
  - call glTranslate → M = I*T
  - Then call glRotate → M = I*T*R

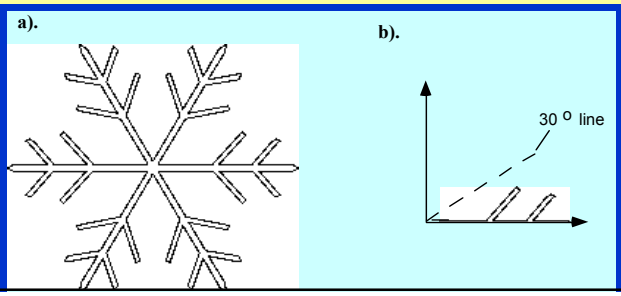
Example : Star

- Assume starMotif() draws the "motif" on the right
- For (int count = 0; count < 5; count++) {
  starMotif();
  glMatrixMode(GL_MODELVIEW) // use *ModelView* CT
  glRotated(72.0, 0.0, 0.0, 1.0)} //*rotate by 72 deg



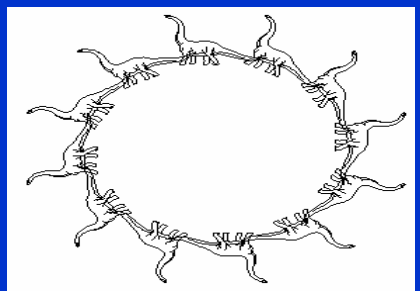
Example: Snowflake

- Motif (on the right) is given
- *How can we model a branch?*
- original motifs, plus a copy reflected across x axis
- *How can we model the snowflake on the left?*
- collection branches, created by consecutive operations of rotation by 60^o



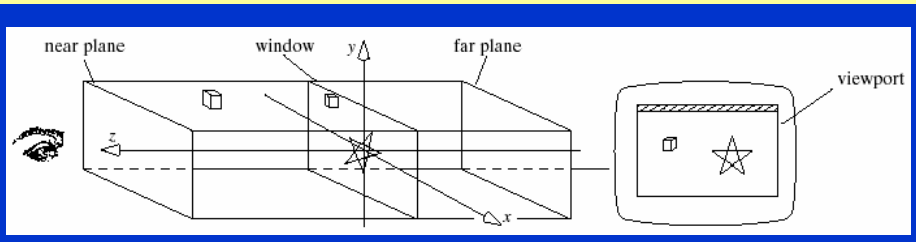
Example : Dino Patterns

- *Dino* motif is upright dinosaur centered at the origin
- *How can we models the required copies?*
- Rotate *Dino* through a suitable angle
- *translate* along *y*-axis by the length of the radius
- Then rotate again



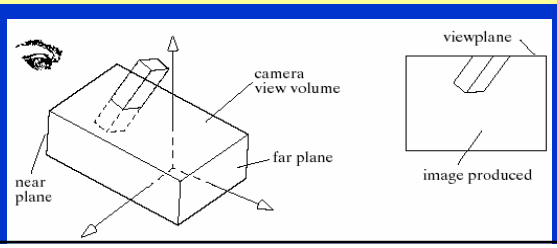
Eye (camera) and View Volume

- Eye *by default* looks along –z axis at the *world window*
  - a rectangle in the *xy*-plane.
- **view volume** of eye is a rectangular parallelepiped.
- side walls fixed by window edges; others by near/far planes
- everything outside view volume is clipped
- Everything inside projected to window plane (orth project.)



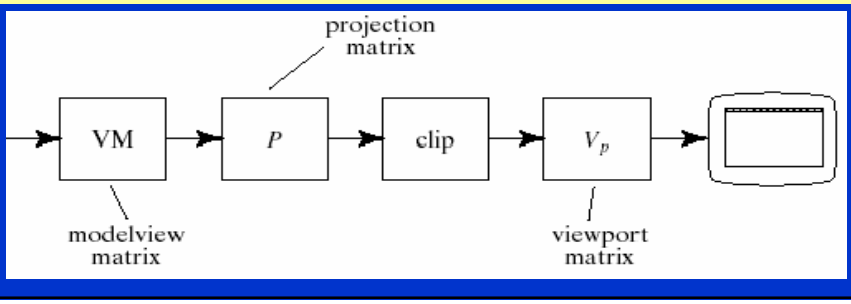
Repositioning the eye and the View Matrix V

- eye (and its view volume) can be positioned somwhre in the World. (OpenGL: use *glLookat())*
- *What happens to the View Volume*
- View Volume is accordingly positioned
- all objects and eye are then transformed by V
  - So that eye repositioned at the standard location
- The ModelView matrix multiplied by V: M → VM



Graphics Pipeline Revisited (1)

- VM includes all objects, eye, view volume transformations
- P is projection onto the World Window
  - Here we assume orthogonal transformation
    - (ignore Z coordinates)
- Vp transform what's in the Window to a viewport that will be drawn on the screen

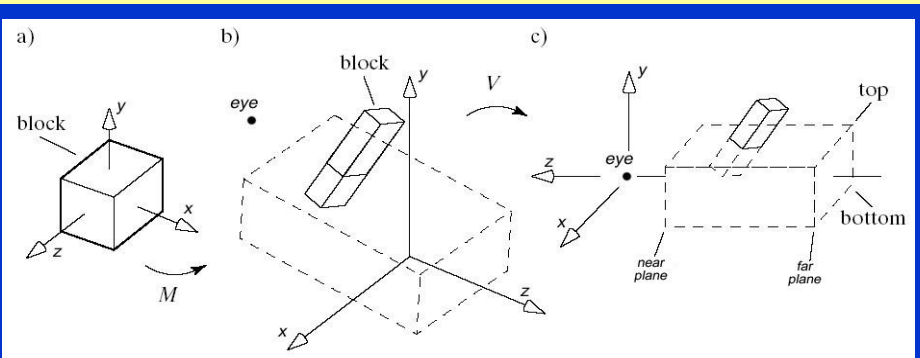


Graphics Pipeline Revisited (2)

- Input vertices in *World Coordinates*
  - using glVertex3d(x, y, z)
- Each 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 (orthogonal or perspective)
  - The *viewport* matrix

Operation of the *modelview* matrix VM

- *M* scales, rotates, and translates the cube into the block
- eye with its view volume is positioned somewhere
- *V* rotates and translates the block into a new position
  - so that Eye and view volume in the standard position
- All objects' coordinates are now called *eye coordinates*



Operation of the *Projection* Matrix, P

- P translates & scales all vertices & view volume
  - so that the new view volume is the *standard cube*
  - extends from -1 to 1 in each dimension
- clipping is done now (relatively easy. *Why?*)
- Coordinates now called *Normalized Device Coords*
- P also reverses the sense of the z coordinates
  - increasing values of *z* now represent relatively farther away points
- *Why do we need these Z values?*

The Viewport Matrix, Vp (1)

- Vp Transforms all (now clipped) objects once more!
- So that standard view volume is the 3D viewport:
- x, y coordinate values extend across the viewport
  - A rectangular area that we will see later on the screen
  - Coordinates are now called *screen coordinates*
- z-component extends from 0 to 1
- a measure of the relative depth of each point
- Helps easy identification of hidden surfaces and lines

The viewport Matrix, Vp (2)