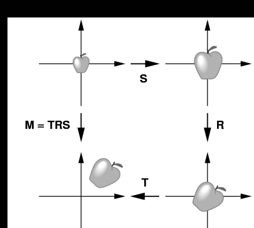
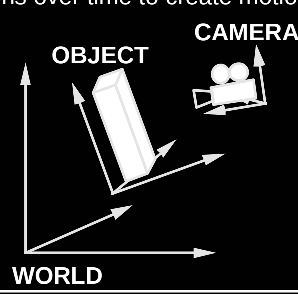
Transformations

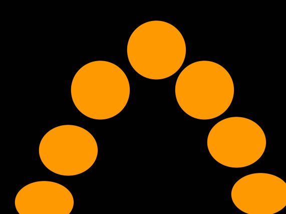
Vectors, bases, and matrices
Translation, rotation, scaling
Homogeneous coordinates
3D transformations
3D rotations
Transforming normals

Uses of Transformations

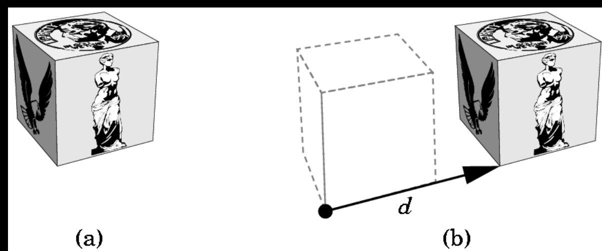
- Modeling transformations
  - build complex models by positioning simple components
  - transform from object coordinates to world coordinates
- Viewing transformations
  - placing the virtual camera in the world
  - i.e. specifying transformation from world coordinates to camera coordinates
- Animation
  - vary transformations over time to create motion

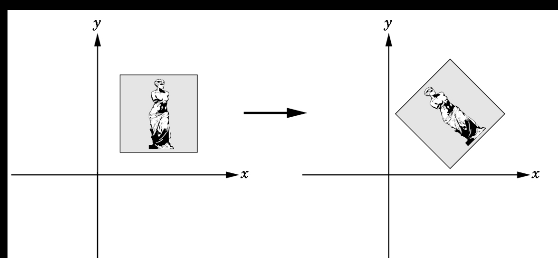


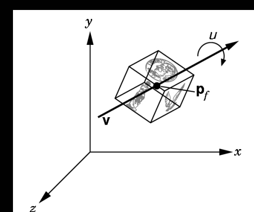




Rigid Body Transformations

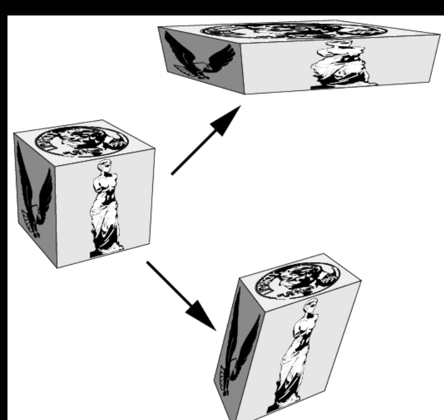


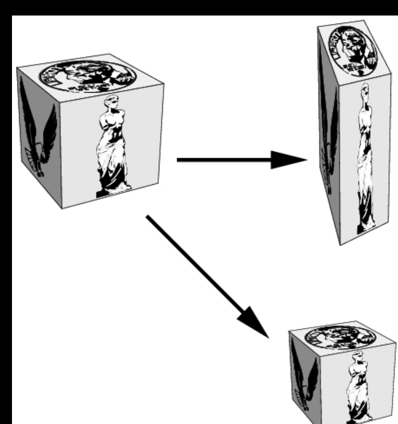




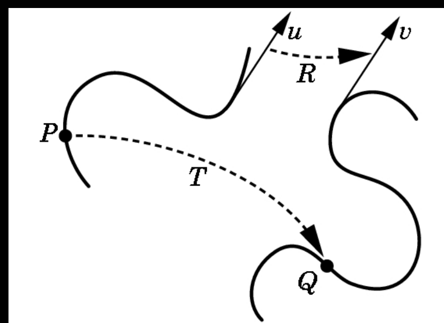
Rotation angle and line about which to rotate

Non-rigid Body Transformations





General Transformations

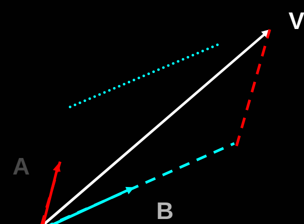


$Q = T(P)$ for points
$V = R(u)$ for vectors

Background Math: Linear Combinations of Vectors

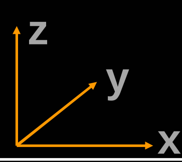
- Given two vectors, A and B, walk any distance you like in the A direction, then walk any distance you like in the B direction
- The set of all the places (vectors) you can get to this way is the set of *linear combinations* of A and B.
- A set of vectors is said to be *linearly independent* if none of them is a linear combination of the others.

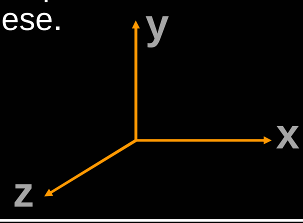
$\mathbf{V} = v_1\mathbf{A} + v_2\mathbf{B}, (v_1,v_2) \in \Re$



Bases

- A *basis* is a linearly independent set of vectors whose combinations will get you anywhere within a space, i.e. *span* the space
- *n* vectors are required to span an *n*-dimensional space
- If the basis vectors are normalized and mutually orthogonal the basis is orthonormal
- There are *lots* of possible bases for a given vector space; there's nothing special about a particular basis—but our favorite is probably one of these.





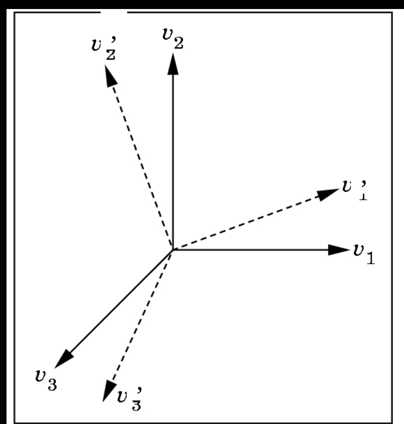
Vectors Represented in a Basis

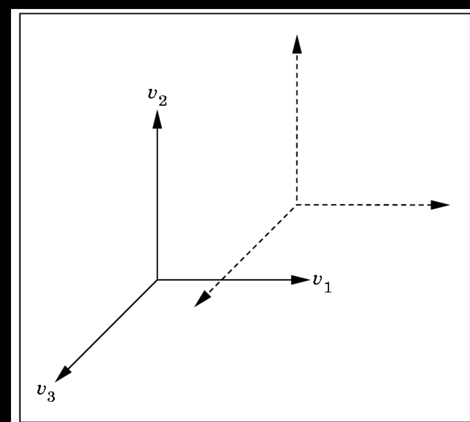
- Every vector has a unique representation in a given basis
  - the multiples of the basis vectors are the vector's *components* or *coordinates*
  - changing the basis changes the components, but not the vector
  - $V = v_1E_1 + v_2E_2 + \ldots v_nE_n$

  The vectors $\{E_1, E_2, \ldots, E_n\}$ are the *basis*

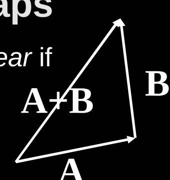
  The scalars $(v_1, v_2, \ldots, v_n)$ are the *components* of V with respect to that basis

Rotation and Translation of a Basis





Linear and Affine Maps



- A function (or map, or transformation) F is *linear* if

$$\mathbf{F}(\mathbf{A}+\mathbf{B}) = \mathbf{F}(\mathbf{A}) + \mathbf{F}(\mathbf{B})$$
$$\mathbf{F}(k\mathbf{A}) = k\,\mathbf{F}(\mathbf{A})$$

for all vectors A and B, and all scalars k.

- Any linear map is *completely specified* by its effect on a set of basis vectors:

$$\begin{aligned}\mathbf{V} &= v_1\mathbf{E}_1 + v_2\mathbf{E}_2 + v_3\mathbf{E}_3 \\ \mathbf{F}(\mathbf{V}) &= \mathbf{F}(v_1\mathbf{E}_1 + v_2\mathbf{E}_2 + v_3\mathbf{E}_3) \\ &= \mathbf{F}(v_1\mathbf{E}_1) + \mathbf{F}(v_2\mathbf{E}_2) + \mathbf{F}(v_3\mathbf{E}_3) \\ &= v_1\mathbf{F}(\mathbf{E}_1) + v_2\mathbf{F}(\mathbf{E}_2) + v_3\mathbf{F}(\mathbf{E}_3)\end{aligned}$$

- A function F is *affine* if it is linear plus a translation
  - Thus the 1-D transformation *y=mx+b* is not linear, but affine
  - Similarly for a translation and rotation of a coordinate system
  - Affine transformations preserve lines

Transforming a Vector

- The coordinates of the transformed basis vector (in terms of the original basis vectors):

$$\mathbf{F}(E_1) = f_{11}\mathbf{E}_1 + f_{21}\mathbf{E}_2 + f_{31}\mathbf{E}_3$$
$$\mathbf{F}(E_2) = f_{12}\mathbf{E}_1 + f_{22}\mathbf{E}_2 + f_{32}\mathbf{E}_3$$
$$\mathbf{F}(E_3) = f_{13}\mathbf{E}_1 + f_{23}\mathbf{E}_2 + f_{33}\mathbf{E}_3$$

- The transformed general vector V becomes:

$$\begin{aligned}\mathbf{F}(\mathbf{V}) &= v_1\mathbf{F}(\mathbf{E}_1) + v_2\mathbf{F}(\mathbf{E}_2) + v_3\mathbf{F}(\mathbf{E}_3)\\ &= (f_{11}\mathbf{E}_1 + f_{21}\mathbf{E}_2 + f_{31}\mathbf{E}_3)v_1 + (f_{12}\mathbf{E}_1 + f_{22}\mathbf{E}_2 + f_{32}\mathbf{E}_3)v_2 + (f_{13}\mathbf{E}_1 + f_{23}\mathbf{E}_2 + f_{33}\mathbf{E}_3)v_3\\ &= (\mathbf{f}_{11}\mathbf{v}_1 + \mathbf{f}_{12}\mathbf{v}_2 + \mathbf{f}_{13}\mathbf{v}_3)\mathbf{E}_1 + (\mathbf{f}_{21}\mathbf{v}_1 + \mathbf{f}_{22}\mathbf{v}_2 + \mathbf{f}_{23}\mathbf{v}_3)\mathbf{E}_2 + (\mathbf{f}_{31}\mathbf{v}_1 + \mathbf{f}_{32}\mathbf{v}_2 + \mathbf{f}_{33}\mathbf{v}_3)\mathbf{E}_3\end{aligned}$$

and its *coordinates* (still w.r.t. E) are

$$\hat{v}_1 = (f_{11}v_1 + f_{12}v_2 + f_{13}v_3)$$
$$\hat{v}_2 = (f_{21}v_1 + f_{22}v_2 + f_{23}v_3)$$
$$\hat{v}_3 = (f_{31}v_1 + f_{32}v_2 + f_{33}v_3)$$

or just $v_i = \sum_j f_{ij} v_j$ The matrix multiplication formula!

Matrices to the Rescue

- An nxn matrix F represents a linear function in n dimensions
  - *i*-th column shows what the function does to the corresponding basis vector
- Transformation = linear combination of columns of F
  - first component of the input vector scales first column of the matrix
  - accumulate into output vector
  - repeat for each column and component
- Usually compute it a different way:
  - dot row *i* with input vector to get component *i* of output vector

$$\begin{Bmatrix} \hat{v}_1 \\ \hat{v}_2 \\ \hat{v}_3 \end{Bmatrix} = \begin{Bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{Bmatrix} \begin{Bmatrix} v_1 \\ v_2 \\ v_3 \end{Bmatrix} \qquad v_i = \sum_j f_{ij} v_j$$

Basic 2D Transformations

Translate

$x' = x + t_x$ $\quad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$ $\quad \mathbf{x'} = \mathbf{x} + \mathbf{t}$

$y' = y + t_y$

Scale

$x' = s_x x$ $\quad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$ $\quad \mathbf{x'} = \mathbf{Sx}$

$y' = s_y y$

Rotate

$x' = x \cos\theta - y \sin\theta$ $\quad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$ $\quad \mathbf{x'} = \mathbf{Rx}$

$y' = x \sin\theta + y \cos\theta$

Parameters t, *s*, and θ are the "control knobs"

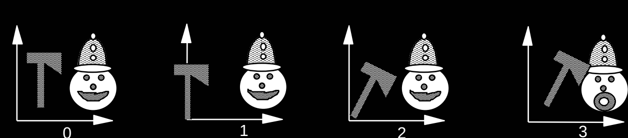
Compound Transformations

- Build *compound* transformations by stringing basic ones together, e.g.
  - *"translate p to the origin, rotate, then translate back"*

  can also be described as a rotation about p
- Any sequence of linear transformations can be collapsed into a single matrix formed by multiplying the individual matrices together

$$v_i = \sum_j f_{ij} \left(\sum_k g_{jk} v_k \right)$$
$$= \sum_k \left(\sum_j f_{ij} g_{jk} \right) v_k$$
$$m_{ij} = \sum_j f_{ij} g_{jk}$$

- This is good: can apply a whole sequence of transformation at once



Translate to the origin, rotate, then translate back.

Homogeneous Coordinates

- Translation is not linear--how to represent as a matrix?
- Trick: add extra coordinate to each vector

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- This extra coordinate is the *homogeneous* coordinate, or *w*
- When extra coordinate is used, vector is said to be represented in *homogeneous coordinates*
- Drop extra coordinate after transformation (project to w=1)
- We call these matrices *Homogeneous* Transformations

W!? Where did that come from?

- Practical answer:
  - W is a clever algebraic trick.
  - Don't worry about it too much. The w value will be 1.0 for the time being.
  - If w is not 1.0, divide all coordinates by w to make it so.
- Clever Academic Answer:
  - (x,y,w) coordinates form a 3D *projective space.*
  - All nonzero scalar multiples of (x,y,1) form an equivalence class of points that project to the same 2D Cartesian point (x,y).
  - For 3-D graphics, the 4D projective space point (x,y,z,w) maps to the 3D point (x,y,z) in the same way.

Homogeneous 2D Transformations

The basic 2D transformations become

Translate: *Scale:* *Rotate:*

$$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Any affine transformation can be expressed as a combination of these.

We can combine homogeneous transforms by multiplication.

Now *any* sequence of translate/scale/rotate operations can be collapsed into a single homogeneous matrix!

3D Transformations

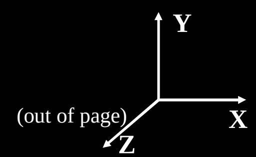
- 3-D transformations are very similar to the 2-D case
- Homogeneous coordinate transforms require 4x4 matrices
- Scaling and translation matrices are simply:

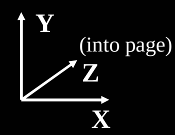
$$\mathbf{S} = \begin{bmatrix} \mathbf{s}_0 & 0 & 0 & 0 \\ 0 & \mathbf{s}_1 & 0 & 0 \\ 0 & 0 & \mathbf{s}_2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & \mathbf{t}_0 \\ 0 & 1 & 0 & \mathbf{t}_1 \\ 0 & 0 & 1 & \mathbf{t}_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Rotation is a bit more complicated in 3-D
  - left- or right-handedness of coordinate system affects direction of rotation
  - different rotation axes

3-D Coordinate Systems

- Right-handed vs. left-handed



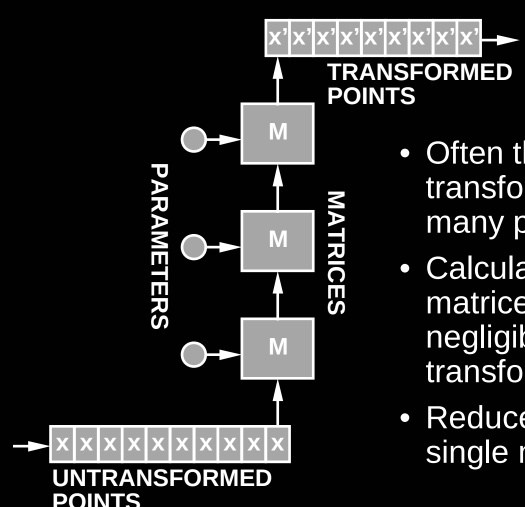


- Z-axis determined from X and Y by cross product: Z=X×Y

$$\mathbf{Z} = \mathbf{X} \times \mathbf{Y} = \begin{bmatrix} X_2 Y_3 - X_3 Y_2 \\ X_3 Y_1 - X_1 Y_3 \\ X_1 Y_2 - X_2 Y_1 \end{bmatrix}$$

- Cross product follows right-hand rule in a right-handed coordinate system, and left-hand rule in left-handed system.

Sequences of Transformations



- Often the same transformations are applied to many points
- Calculation time for the matrices and combination is negligible compared to that of transforming the points
- Reduce the sequence to a single matrix, then transform

Collapsing a Chain of Matrices.

- Consider the composite function ABCD,   i.e. p' = ABCDp
- Matrix multiplication isn't commutative - the order is important
- But matrix multiplication is associative, so can calculate from right to left or left to right: ABCD = (((AB) C) D) = (A (B (CD))).
- Iteratively replace *either* the leading or the trailing pair by its product

Premultiply		Postmultiply	
$M \leftarrow D$	or	$M \leftarrow A$	both give the same result.
$M \leftarrow CM$		$M \leftarrow MB$	
$M \leftarrow BM$		$M \leftarrow MC$	
$M \leftarrow AM$		$M \leftarrow MD$	

- ***Postmultiply:*** **left-to-right (reverse of function application.)**
- ***Premultiply: right-to-left*** **(same as function application.)**

Implementing Transformation Sequences

- Calculate the matrices and cumulatively multiply them into a global *Current Transformation Matrix*
- Postmultiplication is more convenient in hierarchies -- multiplication is computed in the opposite order of function application
- The calculation of the transformation matrix, M,
  - initialize M to the identity
  - in reverse order compute a basic transformation matrix, T
  - post-multiply T into the global matrix M, $M \leftarrow MT$
- Example - to rotate by θ around [x,y]:

```
glMatrixMode(GL_MODELVIEW)/* transform objects in scene */
glLoadIdentity()          /* initialize M to identity mat.*/
glTranslatef(x, y, 0)     /* LAST:  undo translation */
glRotatef(theta,0,0,1)    /* rotate about z axis */
glTranslatef(-x, -y, 0)   /* FIRST: move [x,y] to origin. */
```

- Remember the last T calculated is the first applied to the points
  - calculate the matrices in reverse order

Column Vector Convention

- The convention in the previous slides
  - transformation is by matrix times vector, Mv
  - textbook uses this convention, 90% of the world too

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- The composite function A(B(C(D(x)))) is the matrix-vector product ABCDx

Beware: Row Vector Convention

- The transpose is also possible

$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} m_{11} & m_{21} & m_{31} \\ m_{12} & m_{22} & m_{32} \\ m_{13} & m_{23} & m_{33} \end{bmatrix}$$

- How does this change things?
  - all transformation matrices must be transposed
  - ABCDx transposed is $x^T D^T C^T B^T A^T$
  - pre- and post-multiply are reversed
- OpenGL uses transposed matrices!
  - You only notice this if you pass matrices as arguments to OpenGL subroutines, e.g. glLoadMatrix.
  - Most routines take only scalars or vectors as arguments.

What is a Normal?

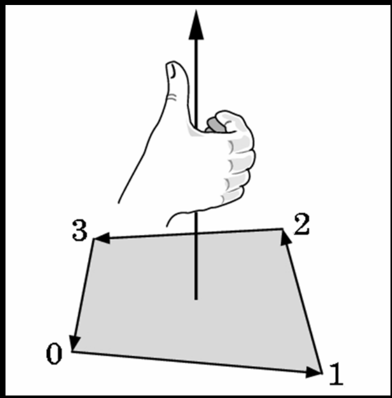
Indication of outward facing direction for lighting and shading

Order of definition of vertices in OpenGL

Right hand rule

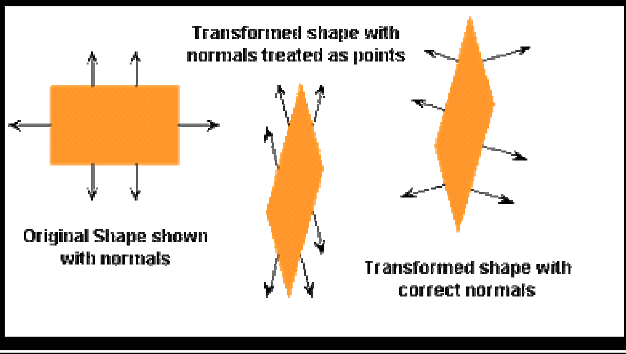
Note: GL conventions…

```
glFrontFace(GL_CCW)
glFrontFace(GL_CW)
```



Transforming Normals

- It's tempting to think of normal vectors as being like porcupine quills, so they would transform like points
- Alas, it's not so, consider the 2D affine transformation below.
- We need a different rule to transform normals.



Normals Do Not Transform Like Points

- If M is a 4x4 transformation matrix, then
  - To transform points, use p'=Mp, where p=[x y z 1]^T
  - *So to transform normals, n'=Mn, where n=[a b c 1]^T right?*
  - Wrong! This formula doesn't work for general M.

Normals Transform Like Planes

A plane $ax + by + cz + d = 0$ can be written

$\mathbf{n} \cdot \mathbf{p} = \mathbf{n}^T \mathbf{p} = 0,$ where $\mathbf{n} = [a \quad b \quad c \quad d]^T$, $\mathbf{p} = [x \quad y \quad z \quad 1]^T$

(a,b,c) is the plane normal, d is the offset.

If $\mathbf{p}$ is transformed, how should $\mathbf{n}$ transform?

To find the answer, do some magic:

$$\begin{aligned} 0 &= \mathbf{n}^T \mathbf{I} \mathbf{p} \quad \textit{equation for point on plane in original space} \\ &= \mathbf{n}^T (\mathbf{M}^{-1} \mathbf{M}) \mathbf{p} \\ &= (\mathbf{n}^T \mathbf{M}^{-1})(\mathbf{M} \mathbf{p}) \\ &= \mathbf{n}'^T \mathbf{p}' \quad \textit{equation for point on plane in transformed space} \end{aligned}$$

$\mathbf{p}' = \mathbf{M} \mathbf{p}$ *to transform point*

$\mathbf{n}' = (\mathbf{n}^T \mathbf{M}^{-1})^T = \boxed{\mathbf{M}^{-1^T} \mathbf{n} \quad \textit{to transform plane}}$

Transforming Normals - Cases

- For general transformations M that include perspective, use full formula (M inverse transpose), use the right *d*
  - *d* matters, because parallel planes do not transform to parallel planes in this case
- For affine transformations, *d* is irrelevant, can use *d*=0.
- For rotations only, M inverse transpose = M, can transform normals and points with same formula.

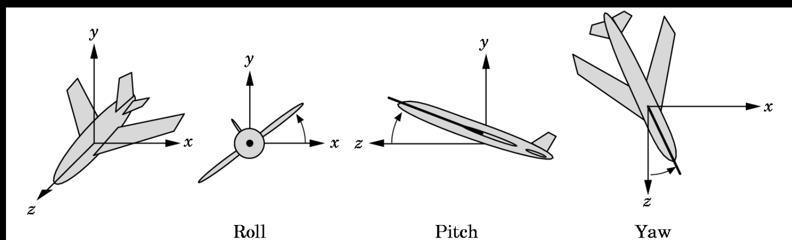
Euler Angles for 3-D Rotations

- Euler angles - 3 rotations about each coordinate axis, however
  - rotations are order-dependent, and there are no conventions about the order to use
  - angle interpolation for animation generates bizarre motions
- Widely used anyway, because they're “simple”
- Coordinate axis rotations (right-handed coordinates):

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

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

Euler Angles for 3-D Rotations



Axis-angle rotation

The matrix R rotates by α about axis (unit) v:

$$\mathbf{R} = \mathbf{v}\mathbf{v}^T + \cos\alpha(\mathbf{I} - \mathbf{v}\mathbf{v}^T) + \sin\alpha\mathbf{v}^*$$

$\mathbf{v}\mathbf{v}^T$	Project onto **v**
$\mathbf{I} - \mathbf{v}\mathbf{v}^T$	Project onto **v**'s normal plane
$\mathbf{v}^*$	Dual matrix. Project onto normal plane, flip by 90°
$\cos\alpha, \sin\alpha$	Rotate by α in normal plane

(assumes **v** is unit.)

The *Dual* Matrix

•If v=[*x,y,z*] is a vector, the skew-symmetric matrix

$$\mathbf{v}^* = \begin{bmatrix} 0 & -z & y \\ z & 0 & -x \\ -y & x & 0 \end{bmatrix}$$

is the *dual matrix* of v

•Cross-product as a matrix multiply: $v^*a = v \times a$

- •helps define rotation about an arbitrary axis
- •angular velocity and rotation matrix time derivatives

•Geometric interpretation of v^*a

- •project a onto the plane normal to v
- •rotate a by 90° about v
- •resulting vector is perpendicular to v and a

Quaternions

- Complex numbers can represent 2-D rotations
- Quaternions, a generalization of complex numbers, can represent 3-D rotations
- Quaternions represent 3-D rotations with 4 numbers:
  - 3 give the rotation axis - magnitude is sin $\alpha/2$
  - 1 gives cos $\alpha/2$
  - unit magnitude - points on a 4-D unit sphere
- Advantages:
  - no trigonometry required
  - multiplying quaternions gives another rotation (quaternion)
  - rotation matrices can be calculated from them
  - direct rotation (with no matrix)
  - no favored direction or axis

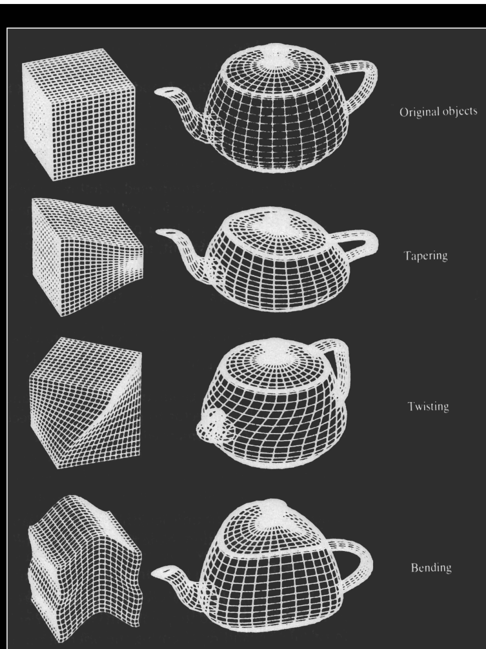
Spatial Deformations

- Linear transformations
  - take any point (x,y,z) to a new point (x',y',z')
  - Non-rigid transformations such as shear are "deformations"
- Linear transformations aren't the only types!
- A transformation is any rule for computing (x',y',z') as a function of (x,y,z).
- Nonlinear transformations would enrich our modeling capabilities.
- Start with a simple object and deform it into a more complex one.

Bendy Twisties

- Method:
  - define a few simple shapes
  - define a few simple non-linear transformations (deformations e.g. bend/twist, taper)
  - make complex objects by applying a sequence of deformations to the basic objects
- Problem:
  - a sequence of non-linear transformations can not be collapsed to a single function
  - every point must be transformed by every transformation

Bendy Twisties



Example: Z-Taper

- Method:
  - align the simple object with the z-axis
  - apply the non-linear taper (scaling) function to alter its size as some function of the z-position
- Example:
  - applying a linear taper to a cylinder generates a cone

"Linear" taper:

$$x' = (k_1 z + k_2)x$$
$$y' = (k_1 z + k_2)y$$
$$z' = z$$

General taper (*f* is any function you want):

$$x' = f(z)x$$
$$y' = f(z)y$$
$$z' = z$$

Example: Z-twist

- Method:
  - align simple object with the z-axis
  - rotate the object about the z-axis as a function of z
- Define angle, θ, to be an arbitrary function *f* (z)
- Rotate the points at z by θ = *f* (z)

"Linear" version: $f(z) = kz$

$$\theta = f(z)$$

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

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

$$z' = z$$

Extensions

- Incorporating deformations into a modeling system
  - How to handle UI issues?
- "Free-form deformations" for arbitrary warping of space
  - Use a 3-D lattice of control points to define Bezier cubics: (x',y',z') are piecewise cubic functions of (x,y,z)
  - Widely used in commercial animation systems
- Physically based deformations
  - Based on material properties
  - Reminiscent of finite element analysis