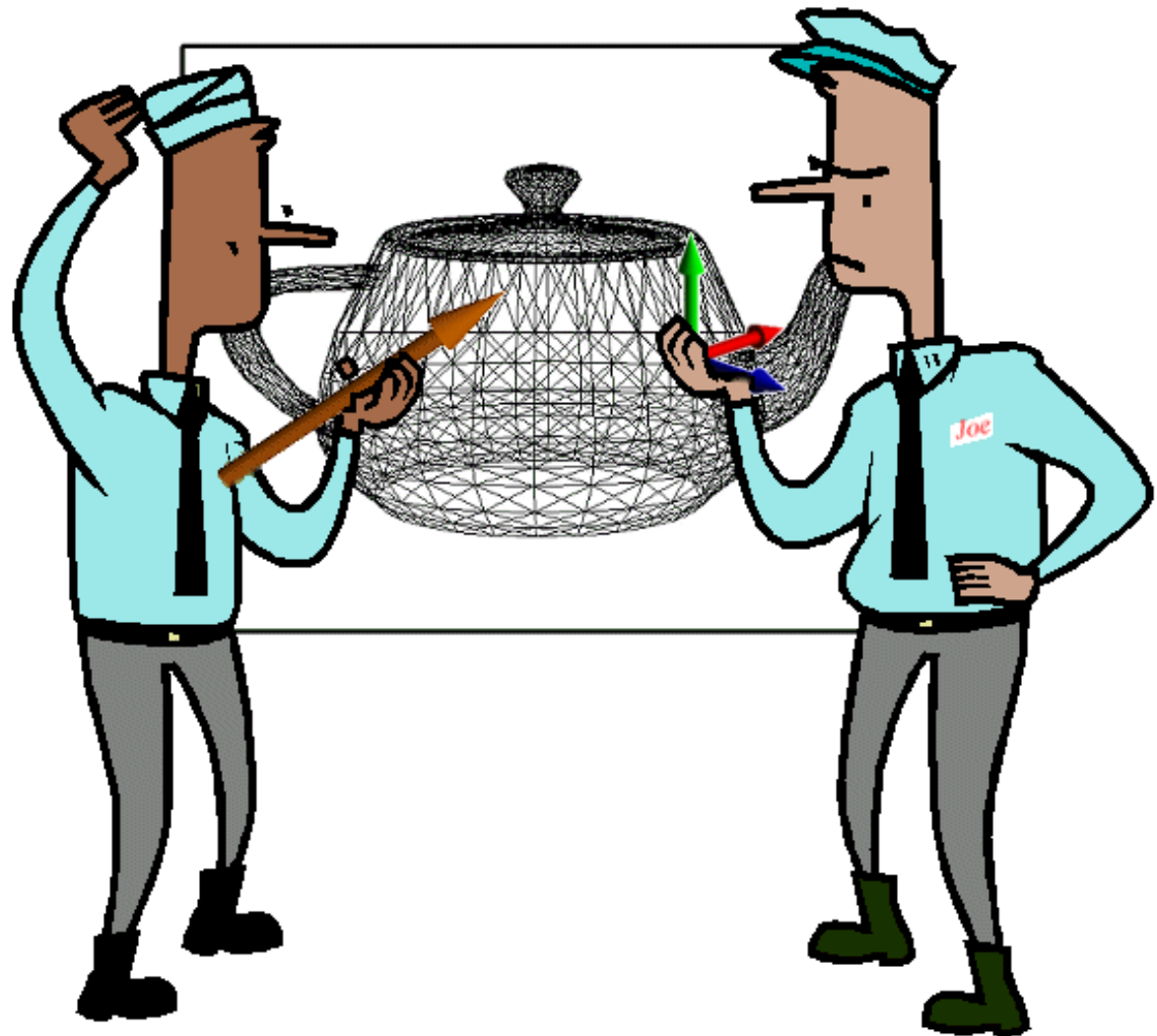


# 6.837 LECTURE 9

1. [3D Transformations - Part 2 Mechanics](#)
3. [Modeling transformations](#)
5. [Viewing Transformation](#)
7. [Rasterization and Display](#)
9. [Vector and Matrix Algebra](#)
11. [Translations](#)
13. [Decomposing Rotations](#)
- 14a. [The Geometry of a Rotation](#)
15. [The Symmetric Matrix](#)
17. [Weighting Factors](#)
19. [Some Sanity Checks](#)
21. [Quaternions](#)
23. [Rotation by Quaternion](#)
25. [Quaternion Interpolation](#)
27. [More Modeling Transforms](#)
2. [The 3-D Graphics Pipeline](#)
4. [Illumination](#)
6. [Clipping and Projection](#)
8. [Rigid-Body Transformations](#)
10. [Cross Product in Matrix Form](#)
12. [Rotations](#)
14. [The Geometry of a Rotation](#)
- 14b. [The Rodrigues Formula](#)
16. [The Skew Symmetric Matrix](#)
18. [Some Sanity Checks](#)
20. [Some Sanity Checks](#)
22. [Quaternion Facts](#)
24. [Quaternion Composition](#)
26. [Euclidean Transformations](#)
28. [Next Time](#)

# 3D Transformations - Part 2 Mechanics

- The 3-D graphics pipeline
- Rigid-body transforms



# The 3-D Graphics Pipeline



- A sneak look at where we're headed!
- Seldom are any two versions drawn the same way
- Seldom are any two versions implemented the same way
- Primitives are processed in a set series of steps
- Each stage forwards its result on to the next stage
- Hardware implementations will commonly have multiple primitives various stages

# Modeling transformations



- We start with 3-D models defined in their own *model space*

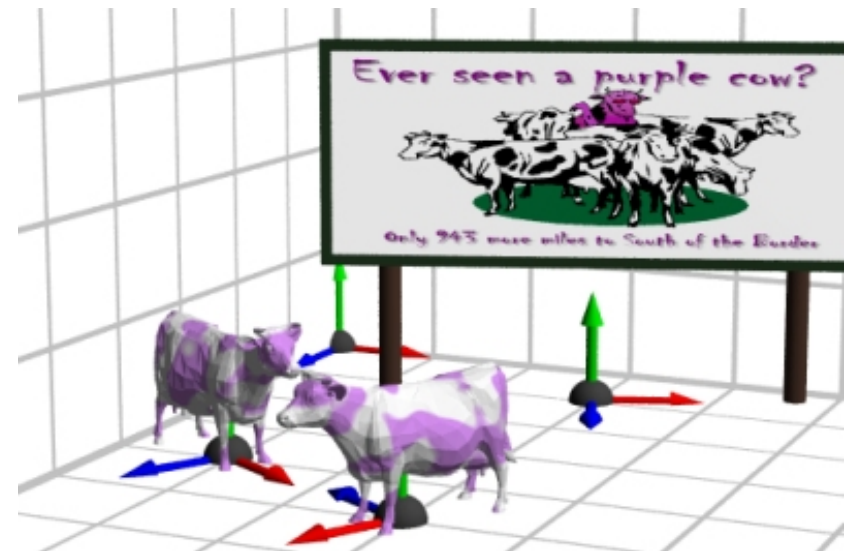
$$\vec{m}_1^t, \vec{m}_2^t \dots, \vec{m}_n^t$$

- **Modeling transformations** orient models within a common coordinate frame called *world space*

$$\vec{w}^t$$

- All objects, light sources, and the viewer live in *world space*

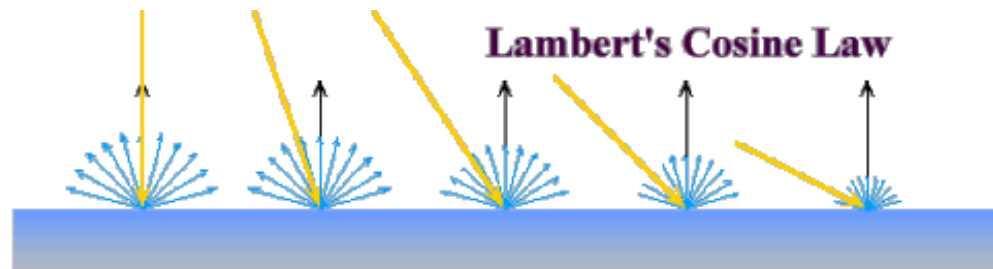
- **Trivial rejection** attempts to eliminate objects that cannot possibly be seen (an optimization)



# Illumination



- Next we **illuminate** potentially visible objects
- Object colors are determined by their material properties, and the light sources in the scene
- Illumination algorithm depends on the shading model and the surface model
- More about this in later on



# Viewing Transformation



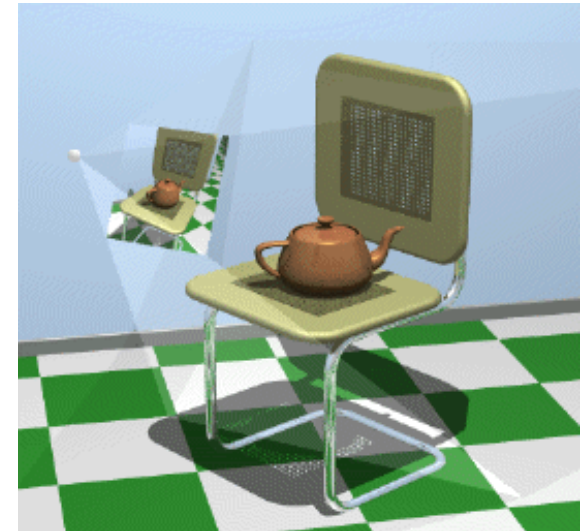
- Another change of coordinate systems
- Maps points from *world space* into *eye space*
- Viewing position is transformed to the origin
- Viewing direction is oriented along some axis
- A viewing volume is defined



# Clipping and Projection



- Next we perform clipping of the scene's objects against a three dimensional viewing volume called a *viewing frustum*
- This step totally eliminates any objects (and pieces of objects) that are not visible in the image
- A clever trick is used to straighten out the viewing frustum in to a cube
- Next the objects are projected into two-dimensions
- Transformation from eye space to *screen space*



# Rasterization and Display



- One last transformation from our screen-space coordinates into a *viewport coordinates*
- The rasterization step scan converts the object into pixels
- Involve interpolating parameters as we go
- Purely 2D operation

Almost every step in the rendering pipeline involves a change of coordinate systems. Transformations are central to understanding three-dimensional computer graphics.

# Rigid-Body Transformations

- Rigid-body, or *Euclidean transformations*
- Preserve the shape of the objects that they act on
- Includes rotations and translations (just as in two dimensions)

Recall our representation for the coordinates of 3-D points. We represent these coordinates in column vectors. A typical point with coordinates  $(x, y, z)$  is represented as:

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

This is not the only possible representation. You may encounter textbooks that consider points as row vectors. What is most important is that you use a consistent representation.

# Vector and Matrix Algebra

You've probably been exposed to vector algebra in previous courses. These include *vector addition*, the *vector dot product*, and the *vector cross product*. Let's take a minute to discuss an equivalent set of matrix operators.

We begin with the *dot product*. This operation acts on two vectors and returns a scalar. Geometrically, this operation can be described as a projection of one vector onto another. The dot product has the following matrix formulation.

$$\bar{a} \cdot \bar{b} = \begin{bmatrix} a_x & a_y & a_z \end{bmatrix} \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \alpha$$

# Cross Product in Matrix Form

The vector cross product also acts on two vectors and returns a third vector. Geometrically, this new vector is constructed such that its projection onto either of the two input vectors is zero.

$$\vec{a} \times \vec{b} = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix} \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \vec{c} \quad \begin{array}{l} \vec{a} \cdot \vec{c} = 0 \\ \vec{b} \cdot \vec{c} = 0 \end{array}$$

In order for one vector to project onto another with a length of zero, it must either have a length of zero, or be *perpendicular* to the second vector. Yet the vector generated by the cross-product operator is perpendicular to two vectors. Since the two input vectors define a plane in space, the vector that results from the cross product operation is perpendicular, or *normal* to this plane.

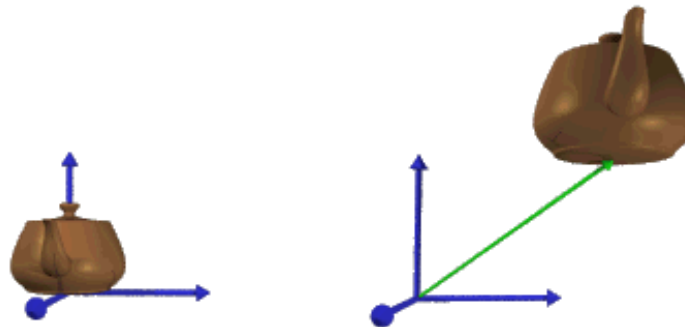
For any plane there are two possible choices of a normal vector, one on each side of a plane. The cross product is defined to be the one of these two vectors where the motion from the tip of the first input vector to the tip of the second input vector is in a counter-clockwise direction when observed from the side of the normal. This is just a restatement of the right-hand rule that you are familiar with.

# Translations

Objects are usually defined relative to their own coordinate system. We can *translate* points in space to new positions by adding offsets to their coordinates, as shown in the following vector equation.

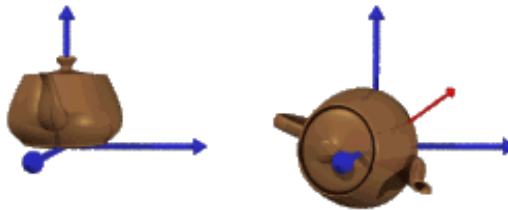
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

The following figure shows the effect of translating a teapot.



# Rotations

Rotations in three-dimensions are considerably more complicated than two-dimensional rotations. In general, rotations are specified by a rotation axis and an angle. In two-dimensions there is only one choice of a rotation axis that leaves points in the plane.



There are several different ways to present rotations. I will use a different approach than that used in most books. Typically, all possible rotations are treated as the composition of three canonical rotations, one about the x-axis, one about the y-axis and one about the z-axis. In order to use this model you need to do the following. Memorize the three canonical rotations, which aside from the signs of the sines, isn't too hard. Next you have to go through a series of rotations which move the desired rotation axis onto one of your canonical rotations, and then you have to rotate it back without introducing any extraneous twists. **This is a difficult and error-prone process. But worst of all it is ambiguous. There exist several different combinations canonical rotations that result in the same overall result.**

# Decomposing Rotations

There is another way that is both easier to understand and provides you with more insights into what rotation is really about. Instead of specifying a rotation by a series of canonical angles, we will specify an arbitrary axis of rotation and an angle. We will also first consider rotating vectors, and come back to points shortly.

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \left( \text{Symmetric} \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} (1 - \cos\theta) + \text{Skew} \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} \sin\theta + \mathbf{I} \cos\theta \right) \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

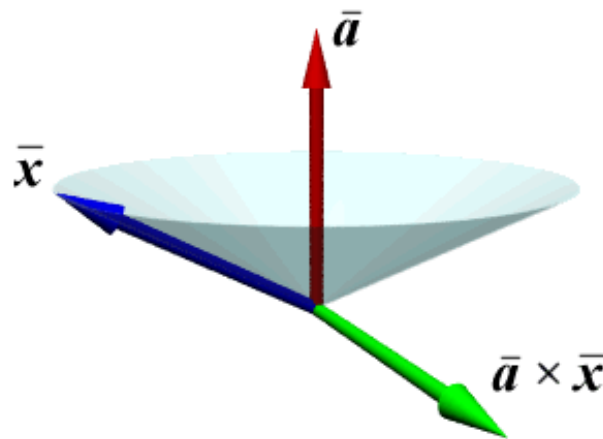
The vector  $a$  specifies the axis of rotation. *This axis vector must be normalized.* The rotation angle is given by  $\theta$ .

You might ask "How am I going to remember **this** equation?". However, once you understand the geometry of rotation, the equation will seem obvious.

The basic idea is that *any rotation can be decomposed into weighted contributions from three different vectors.*

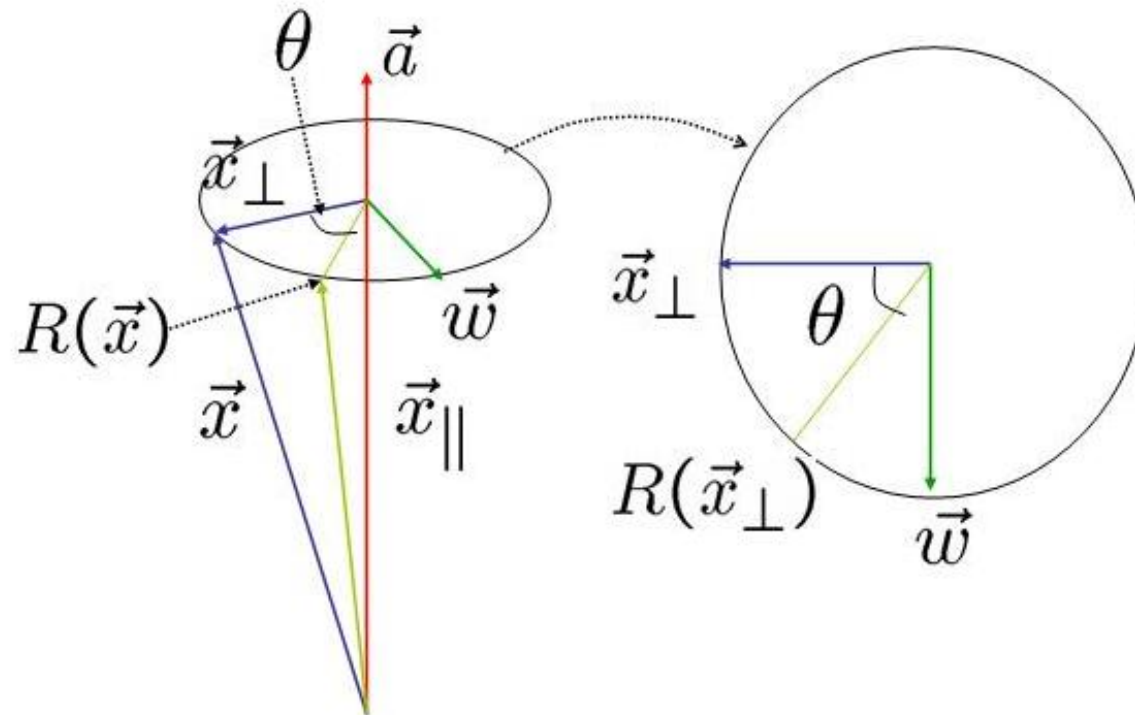
# The Geometry of a Rotation

We can actually define a *natural* basis for rotation in terms of three defining vectors. These vectors are the rotation axis, a vector perpendicular to both the rotation axis and the vector being rotated, and the vector itself. These vectors correspond to the each respective term in the expression.



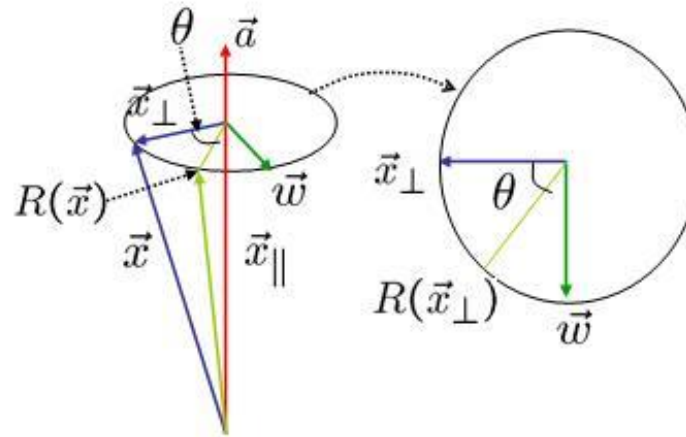
Let's look at this in greater detail

# The Geometry of a Rotation



# The Rodrigues Formula

$$\begin{aligned}
 \vec{w} &= \vec{a} \times \vec{x}_\perp \\
 &= \vec{a} \times (\vec{x} - \vec{x}_\parallel) \\
 &= (\vec{a} \times \vec{x}) - (\vec{a} \times \vec{x}_\parallel) \\
 &= \vec{a} \times \vec{x}
 \end{aligned}$$



$$R(\vec{x}_\perp) = \cos \theta \vec{x}_\perp + \sin \theta \vec{w}$$

$$\vec{x}_\parallel = (\vec{a} \cdot \vec{x}) \vec{a}$$

$$\vec{x}_\perp = \vec{x} - \vec{x}_\parallel = \vec{x} - (\vec{a} \cdot \vec{x}) \vec{a}$$

$$\begin{aligned}
 R(\vec{x}) &= R(\vec{x}_\parallel) + R(\vec{x}_\perp) \\
 &= R(\vec{x}_\parallel) + \cos \theta \vec{x}_\perp + \sin \theta \vec{w} \\
 &= (\vec{a} \cdot \vec{x}) \vec{a} + \cos \theta (\vec{x} - (\vec{a} \cdot \vec{x}) \vec{a}) + \sin \theta \vec{w} \\
 &= \cos \theta \vec{x} + (1 - \cos \theta) (\vec{a} \cdot \vec{x}) \vec{a} + \sin \theta (\vec{a} \times \vec{x})
 \end{aligned}$$

# The Symmetric Matrix

First we'll see why the symmetric matrix of a vector generates a vector in the the direction of the axis.

$$\text{Symmetric} \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} \begin{bmatrix} a_x & a_y & a_z \end{bmatrix} = \begin{bmatrix} a_x^2 & a_x a_y & a_x a_z \\ a_x a_y & a_y^2 & a_y a_z \\ a_x a_z & a_y a_z & a_z^2 \end{bmatrix}$$

The symmetric matrix is composed of the outer product of a row vector and an column vector of the same value.

$$\text{Symmetric} \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \bar{a}(\bar{a} \cdot \bar{x})$$

# The Skew Symmetric Matrix

Next, consider how the skew symmetric matrix of a vector generates a vector that is perpendicular to both the axis and it's input vector.

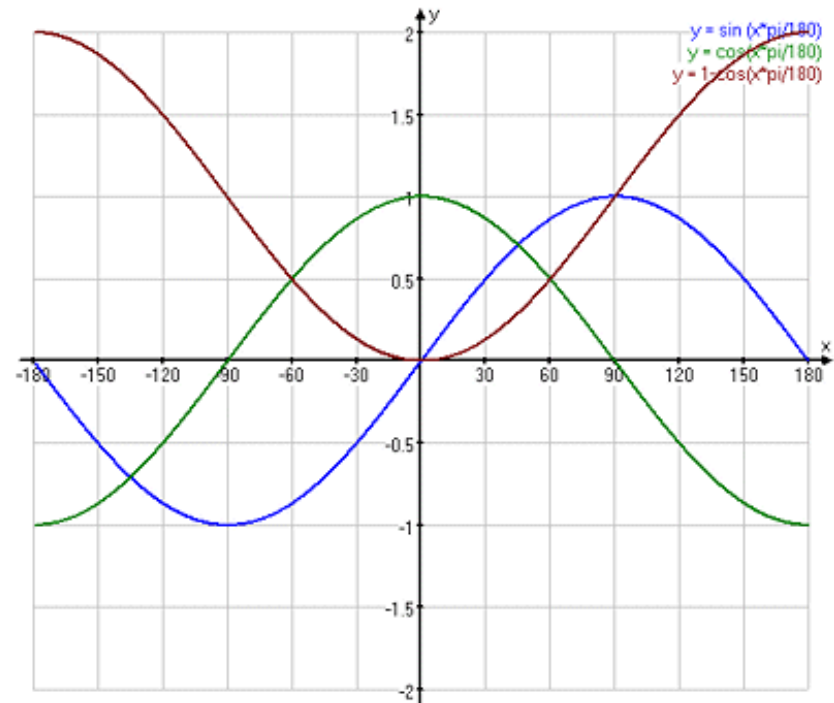
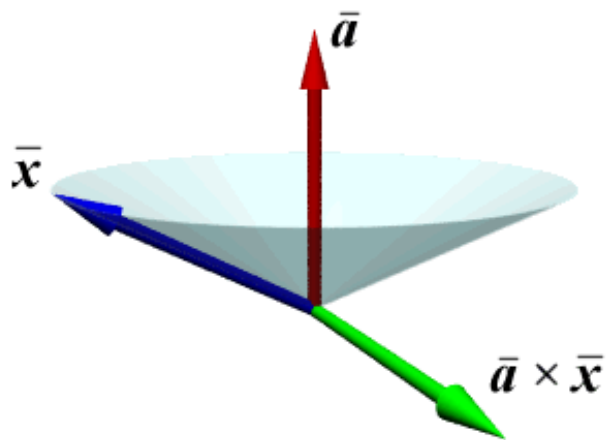
$$\text{Skew} \begin{pmatrix} \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} \end{pmatrix} = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix}$$

$$\text{Skew}(\bar{a})\bar{x} = \bar{a} \times \bar{x}$$

From the identity matrix in the third term it is easy to see how it will generate a vector in the same direction as the input vector.

# Weighting Factors

Even the weighting factors can be easily explained in terms of the component three vectors.



- When  $\theta = 0$  then the sum should reduce to the identity matrix
- The resulting vector after the rotation will always be in the same direction as the axis. Thus, its weight is always positive ( $1 - \cos(\theta)$ )

# Some Sanity Checks

We can now test our new found knowledge. First, consider a rotation by 0.

$$\text{Rotate} \left( \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix}, 0 \right) = \begin{bmatrix} a_x^2 & a_x a_y & a_x a_z \\ a_x a_y & a_y^2 & a_y a_z \\ a_x a_z & a_y a_z & a_z^2 \end{bmatrix} (1-1) + \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix} 0 + \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} 1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

You can also derive the rotation matrices given in the book. For instance, a rotation about the x-axis:

$$\text{Rotate} \left( \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \theta \right) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} (1 - \cos \theta) + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \sin \theta + \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cos \theta$$

$$\text{Rotate} \left( \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \theta \right) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \\ 0 & \sin \theta & \cos \theta \end{bmatrix}$$

# More Sanity Checks

A rotation about the y-axis.

$$\text{Rotate} \left( \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \theta \right) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} (1 - \cos \theta) + \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix} \sin \theta + \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cos \theta$$

$$\text{Rotate} \left( \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \theta \right) = \begin{bmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{bmatrix}$$

This approach to rotations is completely general and unambiguous.

# More Sanity Checks

Rotations about the z-axis.

$$\text{Rotate} \left( \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \theta \right) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} (1 - \cos \theta) + \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \sin \theta + \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cos \theta$$

$$\text{Rotate} \left( \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \theta \right) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

If you grasp this simple concept and understand the geometry, then don't be surprised if, in a few years, everyone is coming to you to compute their rotation matrices. They will also drop their jaws in amazement when you just write down the matrix.

# Quaternions

Matrices are not the only (or best) way of representing rotations. For one thing, they are redundant (9 numbers instead of 3) and, for another, they are difficult to interpolate.

An alternative representation was developed by Hamilton in the early 19th century (and forgotten until relatively recently). The **quaternion** is a 4-tuple of reals with the operations of addition and multiplication defined. Just as complex numbers allow us to multiply and divide two-dimensional vectors, quaternions enable us to multiply and divide four dimensional vectors.

$$q = q_0 + q_1i + q_2j + q_3k$$

$$i^2 = j^2 = k^2 = -1 \quad ij = k, jk = i, ki = j$$

A quaternion can also be interpreted as having a *scalar* part and a *vector* part. This will give us a more convenient notation.

$$\mathbf{q} = (s, \vec{a}) \quad \text{pure quaternion : } \mathbf{p} = (0, \vec{x})$$

Quaternion addition is just the usual vector addition, the quaternion product is defined as:

$$\mathbf{q}_1 \mathbf{q}_2 = (s_1 s_2 - (\vec{a}_1 \cdot \vec{a}_2), s_1 \vec{a}_2 + s_2 \vec{a}_1 + \vec{a}_1 \times \vec{a}_2)$$

# Quaternion Facts

conjugate :  $q^* = (s, -\vec{a})$

magnitude :  $|q| = \sqrt{qq^*} = \sqrt{s^2 + \vec{a} \cdot \vec{a}}$

unit quaternion :  $|q| = 1$

inverse :  $q^{-1} = \frac{1}{|q|^2} q^*$

$q^{-1} = q^*$ , for unit quaternions

It turns out that we will be able to represent rotations with a unit quaternion. Before looking at why this is so, there are a few important properties to keep in mind:

- The unit quaternions form a three-dimensional sphere in the 4-dimensional space of quaternions.
- Any quaternion can be interpreted as a rotation simply by normalizing it (dividing it by its length).
- Both  $q$  and  $-q$  represent the same rotation (corresponding to angles of  $\theta$  and  $2\pi - \theta$ )

# Rotation by Quaternion

$$R_q(p) = qpq^{-1}$$

$$q = (\cos(\theta/2), \sin(\theta/2)\vec{a}), \quad \text{where } |\vec{a}| = 1$$

$$R_q(p) = (0, \quad (s^2 - \vec{a} \cdot \vec{a})\vec{x} \\ + 2\vec{a}(\vec{a} \cdot \vec{x}) \\ + 2s(\vec{a} \times \vec{x}) \quad )$$

$$R_q(p) = (0, \quad (\cos^2(\theta/2) - \sin^2(\theta/2))\vec{x} \\ + (2\sin^2(\theta/2))\vec{a}(\vec{a} \cdot \vec{x}) \\ + (2\cos(\theta/2)\sin(\theta/2))(\vec{a} \times \vec{x}) \quad )$$

$$R_q(p) = (0, \quad (\cos \theta)\vec{x} \\ + (1 - \cos \theta)\vec{a}(\vec{a} \cdot \vec{x}) \\ + (\sin \theta)(\vec{a} \times \vec{x}) \quad )$$

Recognize this? It is the Rodrigues formula!

# Quaternion Composition

Since a quaternion basically stores the axis vector and angle of rotation, it is not surprising that we can write the components of a rotation matrix given the quaternion components.

$$\mathbf{q} = (\cos(\theta/2), \sin(\theta/2)\vec{a}) = (w, (x, y, z))$$

$$R_{\mathbf{q}} = \begin{pmatrix} 1 - 2y^2 - 2z^2 & 2xy - 2wz & 2xz + 2wy & 0 \\ 2xy + 2wz & 1 - 2x^2 - 2z^2 & 2yz - 2wx & 0 \\ 2xz - 2wy & 2yz + 2wx & 1 - 2x^2 - 2y^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

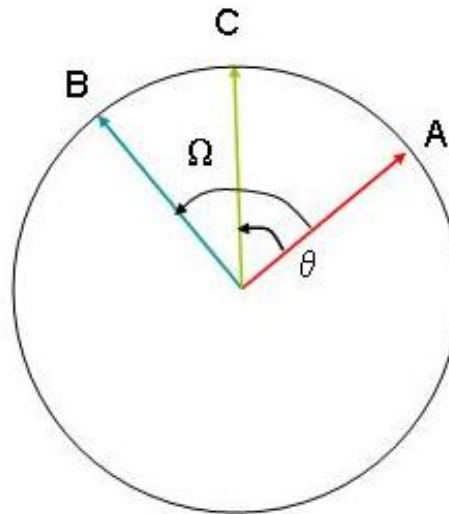
Crucially, the composition of two rotations given by quaternions is simply their quaternion product.

$$R_{\mathbf{q}'}(R_{\mathbf{q}}(\mathbf{p})) = R_{\mathbf{q}''}(\mathbf{p}) \quad \text{where } \mathbf{q}'' = \mathbf{q}'\mathbf{q}$$

- Note that the product of two unit quaternions is another unit quaternion.
- Note that quaternion multiplication, like matrix multiplication, is not commutative.

# Quaternion Interpolation

One of the main motivations for using quaternions in Graphics is the ease with which we can define interpolation between two orientations. Think, for example, about moving a camera smoothly between two views.



$$\cos \Omega = A \cdot B$$

$$\begin{aligned} C(t) &= \text{slerp}(A, B, t) \\ &= A \frac{\sin(\Omega(1-t))}{\sin \Omega} + B \frac{\sin(\Omega t)}{\sin \Omega} \end{aligned}$$

# Euclidean Transformations

For points we can combine the actions of rotation about an arbitrary axis followed by a translation into a single 6-parameter transformation.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_x \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

This set of transformations forms an algebraic group, and they are a subset of the Affine transformations we discussed last time.

# More Modeling Transforms

It is possible to expand our repertoire of modeling transformations in a way parallel to our development of 2-D transformations (We can add Similitudes, Affines, and Projective groups of operators). However, most of these are either not that useful in everyday practice, or they are commonly reduced to one or two special cases, which generalize via a series of more typical operations.

Among, those transforms remaining scales and shears are the most common:

$$Scale(\sigma_x, \sigma_y, \sigma_z) = \begin{bmatrix} \sigma_x & 0 & 0 & 0 \\ 0 & \sigma_y & 0 & 0 \\ 0 & 0 & \sigma_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$Shear(\kappa_{xy}, \kappa_{xz}, \kappa_{yz}) = \begin{bmatrix} 1 & \kappa_{xy} & \kappa_{xz} & 0 \\ 0 & 1 & \kappa_{yz} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We can generate any 3-D affine transformation using a combination of a rotation, a scale, a shear, and a translation.

# Next Time

