Today: 3-D Transforms

Last time, we developed 2-D transformations

But we're mainly interested in 3-D graphics

So today, we'll extend these tools to 3-D

An Alternative View of Transformations

Can be thought of as mapping points to new locations

- this is the basis of the presentation from last time

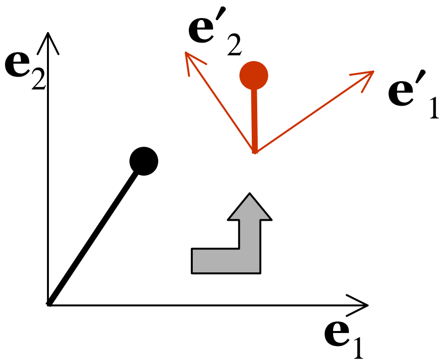
Can also be thought of as a change of coordinate system

- vectors as specified as a linear combination of basis vectors
- for instance, in 2-D:

$$\mathbf{p} = p_1\mathbf{e}_1 + p_2\mathbf{e}_2$$

- transformed vector is similar combination of transformed basis

$$\mathbf{p}' = p_1\mathbf{e}'_1 + p_2\mathbf{e}'_2$$



This is frequently a useful approach to understanding transformations

Scaling & Translation in 3-D

Looks pretty much the same as in 2-D

- just add on the z dimension to everything

Scaling

$$\mathbf{S} = \begin{bmatrix} r & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & t & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translation

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Unfortunately, rotation is not so simple …

Rotation About Coordinate Axes

Looks pretty similar to 2-D case

Specify rotation as 3 angles

- one per coordinate axis
- these are called Euler angles
- fairly widely used

Drawback 1: Result is order *dependent*

- suppose we rotate about x then y
- y rotation is about transformed axis after x rotation is performed
- this gets confusing

Drawback 2: Difficult to interpolate

- for animation want to interpolate angles
- resulting motion can be *weird*

$$\mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}_y = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}_z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation About Coordinate Axes

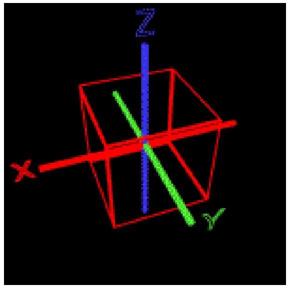
Drawback 1: Result is order *dependent*

- suppose we rotate about x then y
- y rotation is about transformed axis after x rotation is performed
- this gets confusing

Drawback 2: Difficult to interpolate

- for animation want to interpolate angles
- resulting motion can be *weird*

Can produce gimbal lock



$$\mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}_y = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}_z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Some Mathematical Definitions

The dual matrix of a vector u

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

- can write vector cross product $\mathbf{u} \times \mathbf{v}$ as matrix multiply $\mathbf{u}^* \mathbf{v}$

The outer product of a vector $\mathbf{u}$ (with itself)

$$\mathbf{u}\mathbf{u}^{\mathsf{T}} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \begin{bmatrix} x & y & z \end{bmatrix} = \begin{bmatrix} x^2 & xy & xz \\ xy & y^2 & yz \\ xz & yz & z^2 \end{bmatrix}$$

Rotation About Arbitrary Axis

Let's suppose we have a unit direction vector

$$\mathbf{u} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \text{ where } x^2 + y^2 + z^2 = 1$$

We can derive a rotation by a given angle about this axis

$$\mathbf{R}(\theta, \mathbf{u}) = \mathbf{u}\mathbf{u}^{\top} + \cos\theta(\mathbf{I} - \mathbf{u}\mathbf{u}^{\top}) + (\sin\theta)\mathbf{u}^{*}$$

This is the approach used by OpenGL — glRotatef(θ, x, y, z)

Has many of the same interpolation problems as Euler angles

Quaternions

These are essentially generalized complex numbers

- a scalar part + a vector part — 1 real and 3 imaginary parts

$$q = (s, \mathbf{v}) = s + x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$$

Conjugate: $\bar{q} = (s, -\mathbf{v})$

- basic quaternion operation is multiplication

$$qq' = (ss' - \mathbf{v} \cdot \mathbf{v}', \mathbf{v} \times \mathbf{v}' + s\mathbf{v}' + s'\mathbf{v})$$

$$q^{-1} = \frac{\bar{q}}{\|q\|}$$

We're interested in the class of unit quaternions

$$\|q\|^2 = q\bar{q} = s^2 + \|\mathbf{v}\|^2 = 1$$

- forms a unit sphere in 4-D sphere
- can be used to represent the set of rotations

Rotations With Quaternions

Given a point $\mathbf{p}$ and an axis $\mathbf{u}$

- construct the quaternion $q = (\cos\theta, \sin\theta\,\mathbf{u})$
- compute the product $q(0,\mathbf{p})q^{-1}$
- the resulting point $\mathbf{p}'$ is $\mathbf{p}$ rotated by 2θ about $\mathbf{u}$

Quaternion can also be converted to equivalent rotation matrix

$$q = (w, [x \quad y \quad z])$$

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

Looking At Quaternions

Quaternions have a big advantage over Euler angles

- can interpolate between rotations much more nicely
- using scheme called Spherical Linear Interpolation (SLERP)
  - walk along great circle connecting two points on 4-D sphere

But interpolating multiple rotations is still ugly

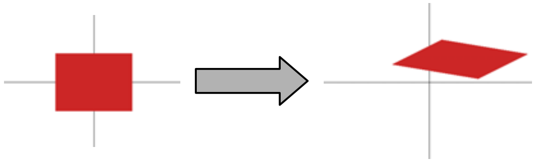
Quaternions have some other nice advantages too

- more compact than rotation matrices
- can compose rotations by quaternion multiplication
- but they can be easily converted to matrices if needed

Transformation of Normal Vectors

Affine transformations map parallel lines to parallel lines

- but the same does *not* hold for perpendicular lines



Transform $\mathbf{M}$ will not map normal vectors to normal vectors

- first guess would be to map normals as $\mathbf{n} \to \mathbf{Mn}$
- after transform, may or may not be perpendicular to surface

Normal vectors are defined by surface tangent planes

- so let's consider how planes are transformed

Transformation of Normal Vectors

A plane in 3-D space is described by the homogeneous vector

$\mathbf{n} = (a, b, c, d)$ where $ax + by + cz + d = 0$ is the plane equation

- thus any point $\mathbf{v}$ on the plane satisfies the equation

$$\mathbf{n}^{\mathsf{T}}\mathbf{v} = 0$$

For any 4x4 matrix whose inverse exists, this is equivalent to

$$\mathbf{n}^{\mathsf{T}}\mathbf{M}^{-1}\mathbf{M}\mathbf{v} = 0$$

- thus the transformed point $\mathbf{Mv}$ lies on the plane $\mathbf{n}^{\mathsf{T}}\mathbf{M}^{-1}$
- it's plane vector is $(\mathbf{n}^{\mathsf{T}}\mathbf{M}^{-1})^{\mathsf{T}}$ or $(\mathbf{M}^{-1})^{\mathsf{T}}\mathbf{n}$

This gives us the transformation rule for normal vectors

$$\mathbf{n} \rightarrow (\mathbf{M}^{-1})^{\mathsf{T}}\mathbf{n}$$

Transformation of Normal Vectors

Must in general compute actual local plane

$\mathbf{n} = (a, b, c, d)$ where $ax + by + cz + d = 0$ is the plane equation

- however, there are some simpler cases

Simplified case #1: Affine Transformations

- map parallel planes to parallel planes
- thus, can pick any value of d — might as well be 0

Simplified case #2: Orthogonal Transformations

- in this case (e.g., rotation) $\mathbf{M}^{-1} = \mathbf{M}^{\mathsf{T}}$
- thus the normal transformation rule becomes $\mathbf{n} \rightarrow \mathbf{M}\mathbf{n}$

Beyond Linear Transformations

There are of course more general kinds of transformations

- in general, any function mapping points to new locations
- for instance, might want to twist an object
- the downside: must transform all points individually

Free-form deformations common in production software

- define a 3-D grid of control points
- use grid points to control Bézier cubic splines within cells
- obviously much more complex than single matrices

For us, affine transforms are (generally) good enough

Next Time: Polygonal Modeling