

## 2-D Transformations

Transformations are functions applied to points in space

$$\mathbf{p}' = f(\mathbf{p})$$

Provide a mechanism for manipulating geometric models

Transformations are *essential* pieces of graphics systems

- OpenGL and PostScript, for instance, use them extensively

## Why Do We Need Transformations?

**Makes modeling more convenient**

- for example, often easier to generate models around origin
  - gluSphere() draws a sphere of radius  $r$  about the origin
- can then move them to final position with transformations

**Model viewing process via transformations**

- projecting 3-D to 2-D will be done this way

**Animation**

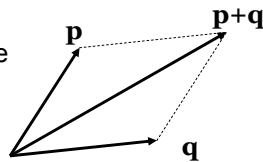
- transformations as a function of time creates motion

## Linear Algebra in 30 Seconds

We represent points as vectors  $\mathbf{p} = [x \ y \ z]$

- vectors add according to parallelogram rule
- a **linear combination** of two vectors is

$$\alpha\mathbf{p} + \beta\mathbf{q}$$



- set of vectors is **linearly independent** if none is a linear combination of the others
- a **basis** for a space is a linearly independent set of vectors whose linear combinations include all vectors in the space

$$\text{standard basis for 2-D plane: } \mathbf{e}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad \mathbf{e}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

- but there are infinitely many possible bases

## Linear & Affine Transformations

We'll be specifically interested in **linear transformations**

$$f(\alpha\mathbf{p} + \beta\mathbf{q}) = \alpha f(\mathbf{p}) + \beta f(\mathbf{q})$$

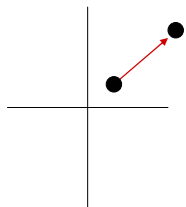
- transformation of shape determined by effect on vertices
- a crucial property that allows for efficient implementation

And the related class of **affine transformations**

$$f(\mathbf{p} + \alpha(\mathbf{q} - \mathbf{p})) = f(\mathbf{p}) + \alpha[f(\mathbf{q}) - f(\mathbf{p})]$$

- preserves affine combinations (e.g., they map lines to lines)
- another view: a linear transformation + a translation
- this is a more general class of functions

## Translation



Offset all points by constant amount

$$x' = x + \Delta x$$

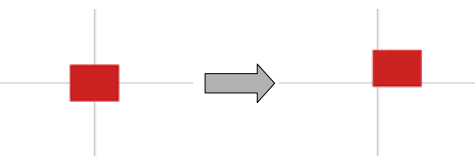
$$y' = y + \Delta y$$

Written as more concise vector equation

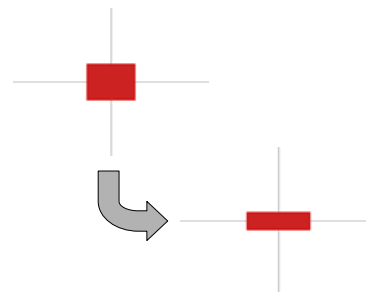
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix}$$

or

$$\mathbf{p}' = \mathbf{p} + \mathbf{d}$$



## Scaling



Scale all points by constant amount

$$x' = sx$$

$$y' = ty$$

And written as a vector equation

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s & 0 \\ 0 & t \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

or

$$\mathbf{p}' = \mathbf{S}\mathbf{p}$$

Squash/stretch along  $x$  &  $y$  axes

## Rotation of Points About Origin

First, write points in polar coordinates

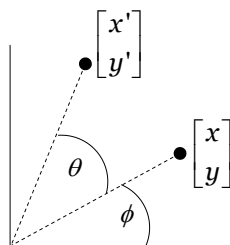
$$x = \rho \cos \phi \quad y = \rho \sin \phi$$

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

And solve for the new positions

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

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

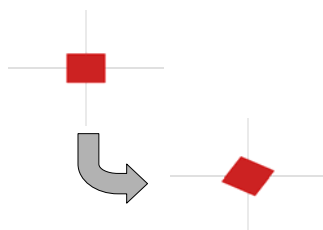


Can write this as a vector equation as well

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

or

$$\mathbf{p}' = \mathbf{R}\mathbf{p}$$



## The Three Fundamental Transformations

Translation:  $\mathbf{p}' = \mathbf{p} + \mathbf{d}$

Scaling:  $\mathbf{p}' = \mathbf{S}\mathbf{p}$

Rotation:  $\mathbf{p}' = \mathbf{R}\mathbf{p}$

We can represent any affine transformation as a sequence of these 3

Translation is only one not represented as matrix multiplication

- because it's not a linear transformation
- wouldn't it be nice if we could come up with a matrix formulation!

## Homogeneous Coordinates

Let's add an extra dimension to our vectors

$$\text{position: } \begin{bmatrix} x \\ y \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \text{direction: } \begin{bmatrix} a \\ b \end{bmatrix} \rightarrow \begin{bmatrix} a \\ b \\ 0 \end{bmatrix}$$

- this added dimension is the homogeneous coordinate
- in general, we'll have coordinates  $[x \ y \ w]$
- the resulting 3-D space is a **projective space**

To convert back, divide by  $w$  and drop the last coordinate

- all vectors  $[ax \ ay \ aw]$  represent the same point
- if  $w=0$ , it represents a "point at infinity"

So what does this do to our transformation equations?

## Transforms in Homogeneous Coordinates

Scaling

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s & 0 & 0 \\ 0 & t & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

or  
 $\mathbf{p}' = \mathbf{S}\mathbf{p}$

Rotation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

or  
 $\mathbf{p}' = \mathbf{R}\mathbf{p}$

Translation

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

or  
 $\mathbf{p}' = \mathbf{T}\mathbf{p}$

## Fundamental Homogeneous Transforms

Translation:  $\mathbf{p}' = \mathbf{T}\mathbf{p}$

Scaling:  $\mathbf{p}' = \mathbf{S}\mathbf{p}$

Rotation:  $\mathbf{p}' = \mathbf{R}\mathbf{p}$

Now we can write all three transforms as matrix multiplications

In general, we'll be using some sequence of transformations

$$\mathbf{M}_1(\mathbf{M}_2(\dots\mathbf{M}_n(\mathbf{v})\dots))$$

## Composing Transformations

Matrix multiplication is associative (but *not* commutative)

- can collapse sequences of transformations into single matrix
- but must *not* reorder any of them

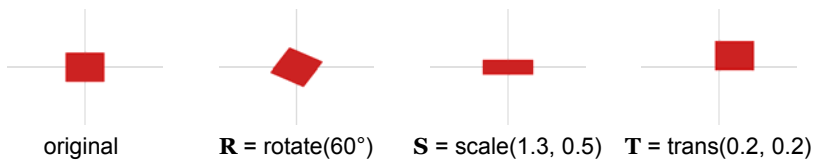
$$\begin{aligned} \mathbf{A}(\mathbf{B}(\mathbf{C}(\mathbf{D}(\mathbf{v})))) &= \mathbf{ABCD}\mathbf{v} \\ &= (\mathbf{ABCD})\mathbf{v} \end{aligned}$$

Can choose to multiply pairs at either end

- add new matrices at beginning — **pre-multiply**
- add new matrices at the end — **post-multiply**

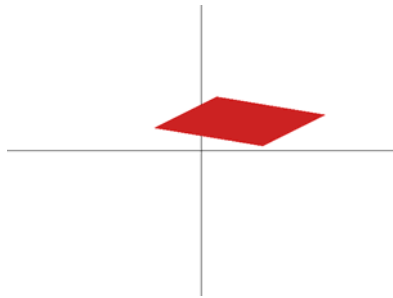
$$\mathbf{ABCD} = (((\mathbf{AB})\mathbf{C})\mathbf{D})$$

## Exercise: Composing Transformations

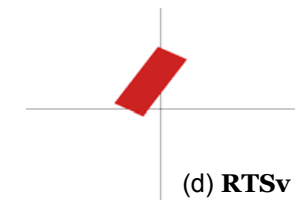
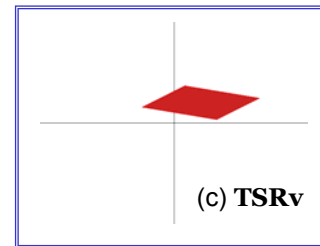
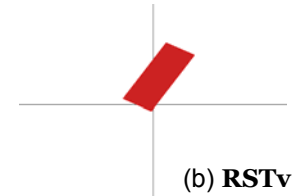
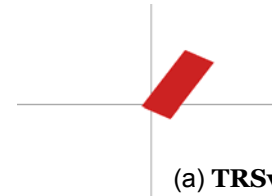


What order of **R, S, T** will produce this figure?

- (a) **TRSv**
- (b) **RSTv**
- (c) **TSRv**
- (d) **RTSv**



## Exercise: Composing Transformations



## 3-D Transforms

We developed 2-D transformations

But we're mainly interested in 3-D graphics

So we'll extend these tools to 3-D

## Scaling & Translation in 3-D

Looks pretty much the same as in 2-D

- just add on the  $z$  dimension to everything

*Scaling*

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

*Translation*

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

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

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

**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

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

**Drawback 2: Difficult to interpolate**

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

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

## Rotation About 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

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

This is the approach used by OpenGL — `glRotatef( $\theta$ ,  $x$ ,  $y$ ,  $z$ )`

Has many of the same interpolation problems as Euler angles

## Some Mathematical Definitions

The **dual matrix** of a vector  $\mathbf{u}$

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

- 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)

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

## Writing Transformations in OpenGL

OpenGL maintains a current transformation matrix  $\mathbf{M}$

- issue commands to post-multiply matrices into  $\mathbf{M}$
- so, commands are listed in *reverse order of application*
- `glLoadIdentity()` — set  $\mathbf{M}$  to identity matrix ( $\mathbf{M} \leftarrow \mathbf{I}$ )
- `glTranslatef( $\Delta x$ ,  $\Delta y$ ,  $\Delta z$ )` — translation ( $\mathbf{M} \leftarrow \mathbf{M}\mathbf{T}$ )
- `glRotatef( $\theta$ ,  $x$ ,  $y$ ,  $z$ )` — rotate about given axis ( $\mathbf{M} \leftarrow \mathbf{M}\mathbf{R}$ )
- `glScalef( $r$ ,  $s$ ,  $t$ )` — scale by given factors ( $\mathbf{M} \leftarrow \mathbf{M}\mathbf{S}$ )

**Example: to rotate about an arbitrary point  $\mathbf{p} = [x \ y \ z]$**

```
glTranslatef(x, y, z, 0); // (3) move p back
glRotatef(theta, 0, 0, 1); // (2) rotate around z axis
glTranslatef(-x, -y, -z, 0); // (1) move p to origin
DrawSomething();
```

## A Word of Caution on Notation

### We've consistently written points as column vectors

- virtually everyone does this
- typically represent matrices in row-major order  $\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$   
 $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$

### Some in graphics have traditionally used row vectors

- convert by transposing everything:  
 $\mathbf{ABv} \rightarrow (\mathbf{ABv})^T = \mathbf{v}^T \mathbf{B}^T \mathbf{A}^T$   $\begin{bmatrix} x' & y' \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} a & c \\ b & d \end{bmatrix}$
- note that order is reversed as well
- typically represent matrices in column-major order  
 $\begin{bmatrix} a & c \\ b & d \end{bmatrix}$
- OpenGL actually does it this way
- but you'll probably never notice

## Quaternions (continued)

### Conjugate

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

### Inversion

$$q^{-1} = \frac{\bar{q}}{\|q\|^2}$$

### 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 space
- can be used to represent the set of rotations
- forms a group with multiplication as the operation

## Quaternions

Complex numbers  $c = a + b\mathbf{i}$ ,  $\mathbf{i}^2 = -1$

### Quaternions are essentially generalized complex numbers

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

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

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

- norm

$$\|q\| = \sqrt{w^2 + x^2 + y^2 + z^2}$$

- basic quaternion operation is multiplication

$$\begin{aligned} qq' &= (s + x\mathbf{i} + y\mathbf{j} + z\mathbf{k})(s' + x'\mathbf{i} + y'\mathbf{j} + z'\mathbf{k}) \\ &= (ss' - \mathbf{v} \cdot \mathbf{v}', \quad \mathbf{v} \times \mathbf{v}' + s\mathbf{v}' + s'\mathbf{v}) \end{aligned}$$

## Rotations With Quaternions

### Rotations in a 2D plane using complex numbers

$$c = [a \ b] = a + b\mathbf{i} = re^{i\phi}, \quad d = \cos\theta + \sin\theta\mathbf{i} = e^{i\theta}$$

$$\begin{aligned} c' = cd &= re^{i\phi} e^{i\theta} = re^{i(\phi+\theta)} = r \cos(\phi+\theta) + r \sin(\phi+\theta)\mathbf{i} \\ &= [r \cos(\phi+\theta) \quad r \sin(\phi+\theta)] \end{aligned}$$

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

- construct the unit quaternion  $q = (\cos\frac{\theta}{2}, \sin\frac{\theta}{2}\mathbf{u})$

- compute the product  $(0, \mathbf{p}') = q (0, \mathbf{p}) q^{-1}$

- the resulting point  $\mathbf{p}'$  is  $\mathbf{p}$  rotated by  $\theta$  about  $\mathbf{u}$

## Quaternion-Matrix Conversion

Quaternion can also be converted to equivalent rotation matrix

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

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

## 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 3-D sphere

$$\text{Slerp}(q_1, q_2; \alpha) = q_1 (q_1^{-1} q_2)^\alpha, \quad 0 \leq \alpha \leq 1$$

$$\text{where } (\cos \theta, \sin \theta \mathbf{u})^\alpha = (\cos \alpha \theta, \sin \alpha \theta \mathbf{u})$$

But interpolating multiple rotations is more complicated

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  $\mathbf{n} \rightarrow \mathbf{M}\mathbf{n}$
- 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

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

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

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

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

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

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

This gives us the transformation rule for normal vectors

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

## 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 (Rigid-Body) Transformations

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

## 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 are 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

