Motto

It hurts but it is definitively healthy

(a pediatrician)

Introduction to Mathematics for Computer Graphics …

that should be a review of very well known things

Mathematics for CG

Geometry.

Deals with the position of the objects.

So the question is "***where it is?***"

Deals with coordinates.

Topology.

Answers the question how the things are connected

From this viewpoint circle and ellipse are equal

Geometry vs Topology

- Generally it is a good idea to look for data structures that **separate the geometry from the topology**
  - **Geometry**: locations of the vertices
  - **Topology**: organization of the vertices and edges
  - Example: a polygon is an ordered list of vertices with an edge connecting successive pairs of vertices and the last to the first
  - **Topology holds even if geometry changes**

Vector Space

Def. An n-dimensional ***vector space*** consists of a set of *vectors* and two operations: **addition** and **scalar multiplication**. The vector space is **closed** under these two operations. There exists a distinguished member of the set called ***zero vector*** **O** that has the following properties:

a.**O** = **O** for all scalars a

O+**v** = **v**+**O** = **v** for all vectors **v**

Affine Space

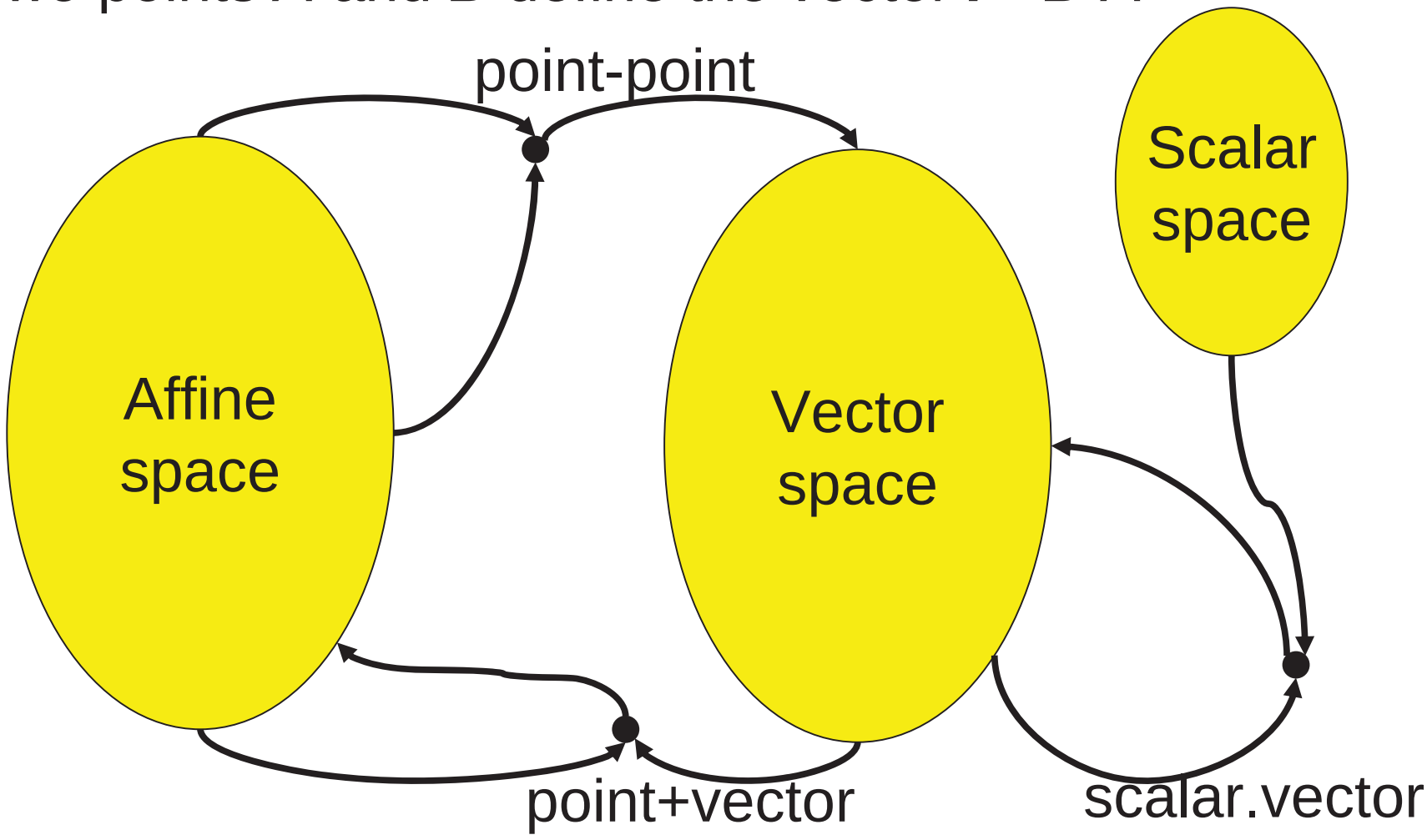
Def. An n-dimensional ***affine space*** consists of a set of *points*, an associated n-dimensional vector space, and two operations: **subtraction of two points in the set** and **addition of a point and a vector in the associated vector space**. The subtraction results in a vector whereas the addition results in a point.

✓*A vector* is expressed by its components ***P***=(x,y,z)

✓*A point* is expressed by its coordinates *A*=[x,y,z]

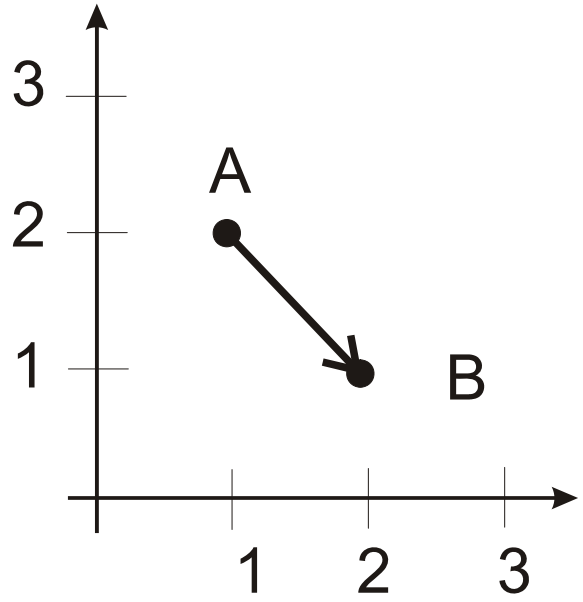
Spaces

Two points *A* and *B* define the vector ***P***=*B*-*A*



Spaces

Let's have two points *A* and *B*
The vector ***P*** is the ***P***=*B*-*A*



example:

A=[1,2,0], *B*=[2,1,0]

P= *B*-*A* = [2,1,0] - [1,2,0] = (1,-1,0)

note that the same vector can be obtained

e.g., from *A*=[0,0,0], *B*=[1,-1,0]

and infinite number of different pairs!

PURDUE UNIVERSITY

Spaces

A vector is a representative of a *set* of oriented lines

In other words a vector can be translated!

A vector added to point yields to a new point

example:

A=[1,2,0], $\boldsymbol{P}$=(1,-1,0)

$A+\boldsymbol{P}$=[1,2,0] + (1,-1,0) = [2,1,0]

Spaces

In general, the operation POINT+POINT is undefined

but there are exceptions:

C=(A+B)/2 is the mid point of a line segment

note: In general the operation $\sum a_i A_i$ is defined iff

$\sum a_i = 1$ or $\sum a_i = 0$

in the first case the result is a point,

in the second one a vector

Spaces

The operation

$\sum a_i A_i$

is called

Affine Combination (or **Affine Sum**)

iff

$\sum a_i = 1$

Affine combination results in a point

Space

Example:

Write a procedure that evaluates mid-point of a line

```
float *MidPoint(float *a, float *b, float *ret)
{
  ret[0]=(a[0]+b[0])/2.f;
  ret[1]=(a[1]+b[1])/2.f;
  ret[2]=(a[2]+b[2])/2.f;
  return ret;
}
```

note: a cycle will be slow!

Spaces

Example:

Linear interpolation

let's have two points *A* and *B*
and the parameter $0 \leq t \leq 1$

the point *Q* inside is

$$Q=(1-t)A+tB$$

note: ~ t=0, Q=A
~ t=1, Q=B
~ t=0.5; Q=mid point
~ (1-t)+t=1, Q is a point!

Space

Example:

Write a procedure of linear interpolation

```
float *Interpolate(float *a, float *b, float t, float *ret)
{
  static float t1=1-t;
  ret[0]= t1*a[0]+t*b[0]
  ret[1]= t1*a[1]+t*b[1]
  ret[2]= t1*a[2]+t*b[2]
  return ret;
}
```

note: a cycle will be slow!

Mathematics for CG

Def:
An affine space with an additional concept of *metrics* is called a **Euclidean space**

$|\mathbf{u}| = \sqrt{\mathbf{u}.\mathbf{u}}$

Example:
procedure that returns distance of two points

```
float Dist(float *a, float *b)
{return sqrt(a[0]-b[0])*(a[0]-b[0])+
            (a[1]-b[1])*(a[1]-b[1])+
            (a[2]-b[2])*(a[2]-b[2]));}
```

note:very bad…
many things repeat,
macro or inline is better

PURDUE UNIVERSITY

Size of a vector

$|\boldsymbol{P}| = \sqrt{(P_x^2 + P_y^2 + P_z^2)}$

☯ this is exactly distance of two points!

$|A,B| = \sqrt{((B_x - A_x)^2 + (B_y - A_y)^2 + (B_z - A_z)^2)}$

Normalized vector is a vector of the unit size

this can be easily achieved

$\boldsymbol{P}' = \boldsymbol{P}/|\boldsymbol{P}|$

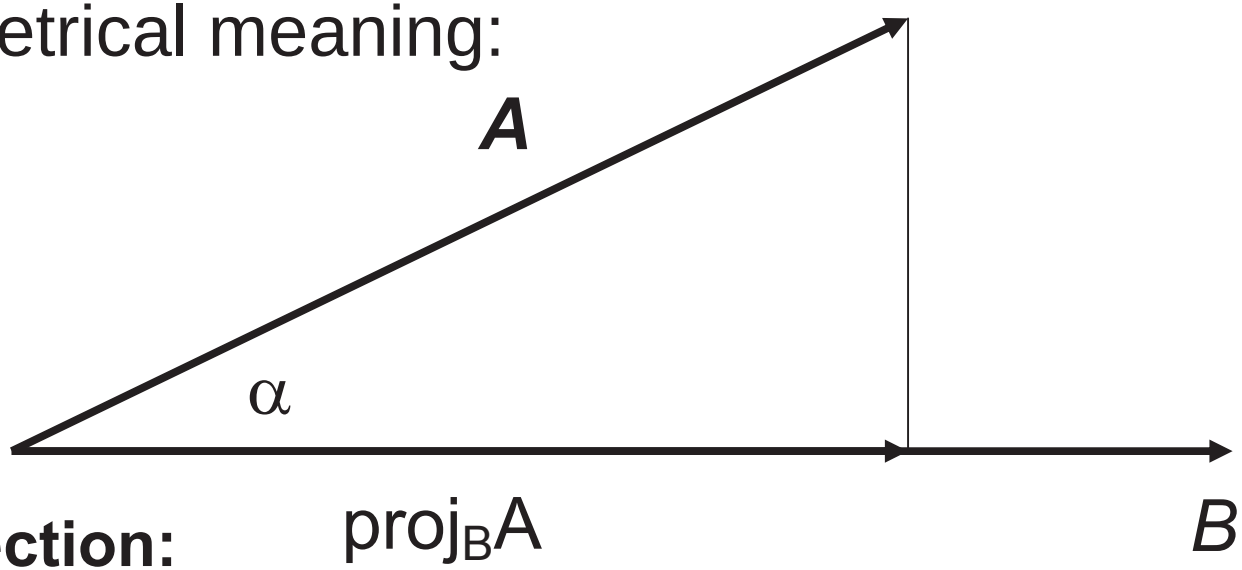
do not confuse it with *the normal vector*!

The Dot Product

☯ *The dot product* of two vectors is a number

$\boldsymbol{A}\cdot\boldsymbol{B}=|\boldsymbol{A}|\,|\boldsymbol{B}|\cos\alpha$,
where α is angle between ***A*** and ***B***

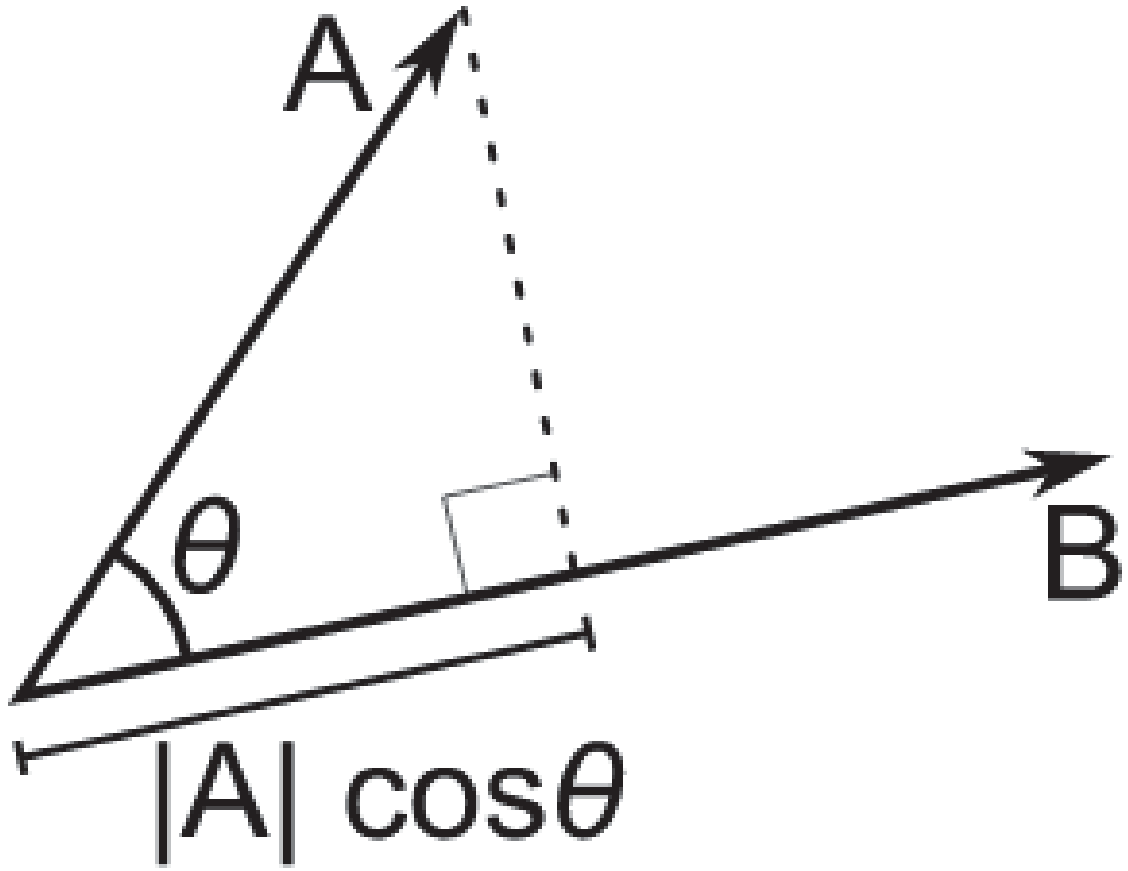
Geometrical meaning:



Vector Projection:

$\text{proj}_B\text{A} = (\boldsymbol{A}\cdot\boldsymbol{B})\boldsymbol{B}/|\boldsymbol{B}|^2$ is the perpendicular projection of the vector ***A*** onto the vector ***B***

Scalar Projection:



PURDUE UNIVERSITY

The Dot Product

having two vectors

P and ***Q***

the dot product is calculated as

$$\boldsymbol{P}\bullet\boldsymbol{Q}=(P_xQ_x+P_yQ_y+P_zQ_z)$$

or:

```
dot=P[0]Q[0]+P[1]Q[1]+P[2]Q[2];
```

The Dot Product

☯ Why is it so important?

Example:

Having three points
A=[1,0,0], B=[1,1,1], and C=[-1,1,-1].
Are the lines AB and AC perpendicular?

Yes, if the vectors $\boldsymbol{AB}$ and $\boldsymbol{AC}$ are perpendicular.

$\boldsymbol{AB}=B-A$=[1,1,1] -[1,0,0]=(0,1,1)
$\boldsymbol{AC}=C-A$=[-1,1,-1] -[1,0,0]=(-1,1,-1)

$\mathbf{AB \bullet AC}$=0(-1)+1.1+1(-1)=0

yes they are….

PURDUE UNIVERSITY

The Dot Product

☯ How can we get the angle between two lines?

1) Get the vectors ***P,Q***

2) Normalize them, ***P'***, ***Q'***

3) Get the dot product ***P'•Q'***=|1||1|cos α

4) **arccos** (***P'•Q'***) gives the angle

PURDUE UNIVERSITY

The Dot Product

Example:

take vector ***P***=(10,0,0) and ***Q***=(0,-5,0)
normalize them and calculate the dot product

P' = (1,0,0), ***Q'*** = (0,-1,0)

P' • ***Q'*** = 1*0 + (-1)*0 + 0*0 = 0

why?

The Vector Product

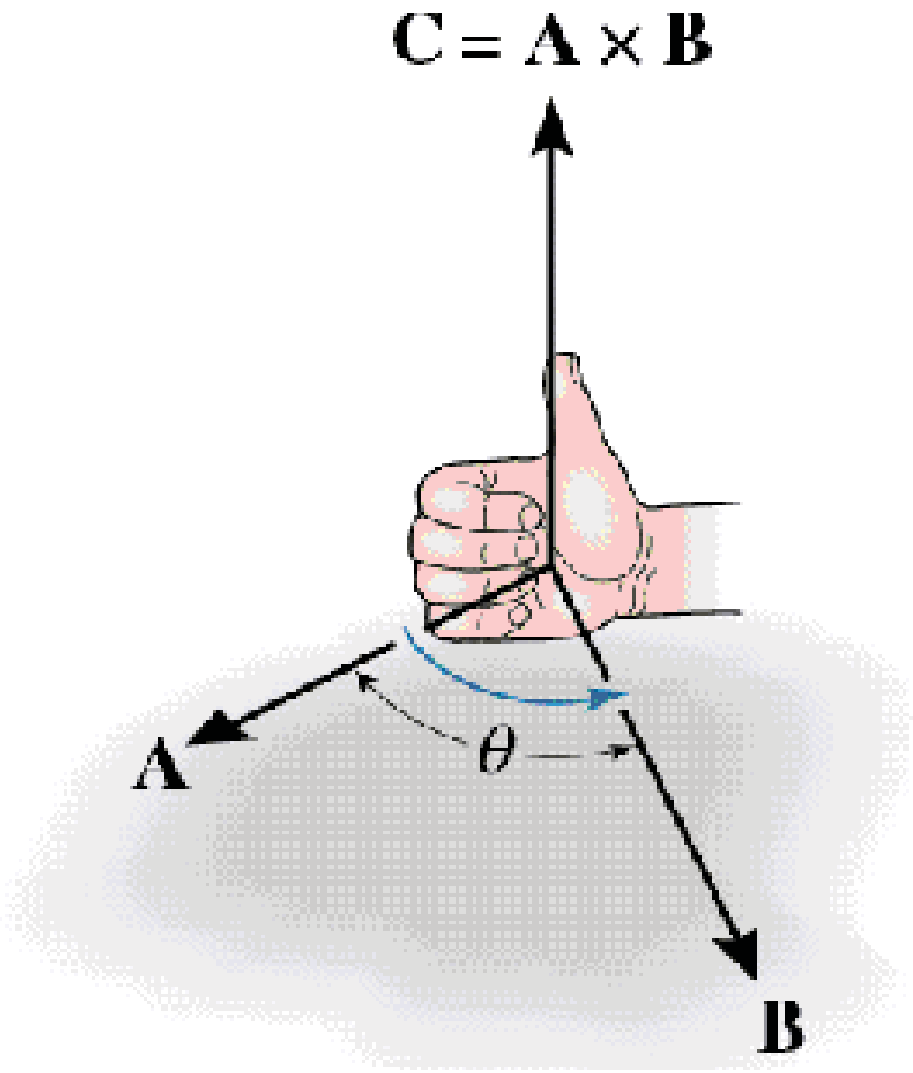
*the vector product (**cross product**)*

the vector product of two vectors P and Q is also a vector

$$\boldsymbol{R} = \boldsymbol{P} \times \boldsymbol{Q} = ((P_yQ_z - P_zQ_y), -(P_xQ_z - P_zQ_x), (P_xQ_y - P_yQ_x))$$

R is the vector perpendicular to ***P*** and ***Q***,
or zero vector if ***P*** is parallel to ***Q***
or zero vector if ***P=0*** or ***Q=0***

PQR form a basis of the coordinate system!
(The vector space)


C = A × B
A
θ
B

$$
\begin{aligned}
\vec{v} \times \vec{w} &= \begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ v_1 & v_2 & v_3 \\ w_1 & w_2 & w_3 \end{vmatrix} \\
&= \hat{i} \begin{vmatrix} v_2 & v_3 \\ w_2 & w_3 \end{vmatrix} - \hat{j} \begin{vmatrix} v_1 & v_3 \\ w_1 & w_3 \end{vmatrix} + \hat{k} \begin{vmatrix} v_1 & v_2 \\ w_1 & w_2 \end{vmatrix} \\
&= (v_2 w_3 - v_3 w_2)\hat{i} - (v_1 w_3 - v_3 w_1)\hat{j} + (v_1 w_2 - v_2 w_1)\hat{k}
\end{aligned}
$$

$$\mathbf{x}_1 \times \mathbf{x}_2 = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ 2 & -3 & 1 \\ -2 & 1 & 1 \end{vmatrix}$$

$$= \begin{vmatrix} -3 & 1 \\ 1 & 1 \end{vmatrix}\mathbf{i} - \begin{vmatrix} 2 & 1 \\ -2 & 1 \end{vmatrix}\mathbf{j} + \begin{vmatrix} 2 & -3 \\ -2 & 1 \end{vmatrix}\mathbf{k}$$

$$= [(-3)(1)-(1)(1)]\mathbf{i} - [(2)(1)-(-2)(1)]\mathbf{j} + [(2)(1)-(-2)(-3)]\mathbf{k}$$

$$= -4\mathbf{i} - 4\mathbf{j} + 8\mathbf{k}$$

The Vector Product

the vector product

easy way to remember the formula

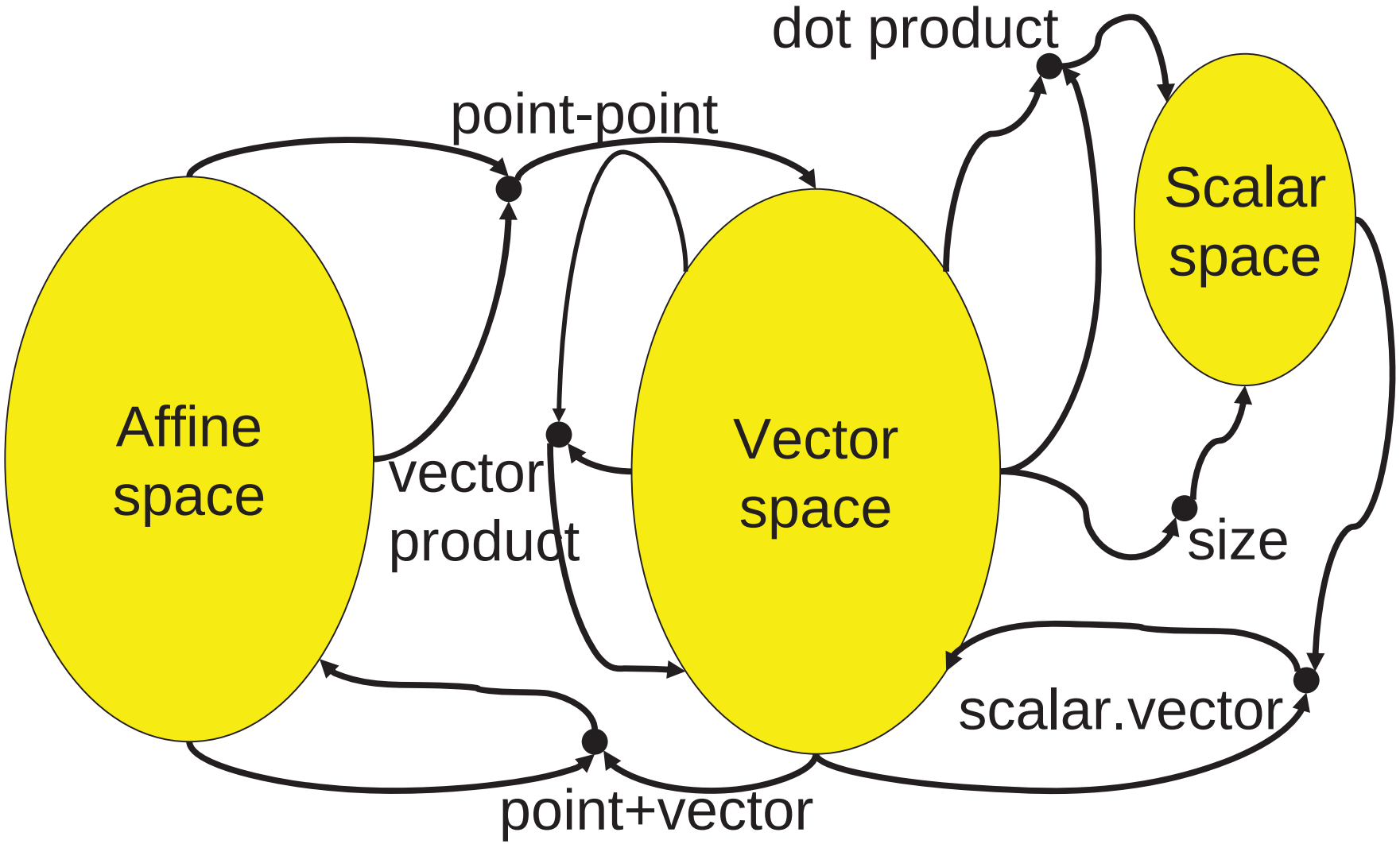
$$\boldsymbol{R} = \boldsymbol{P} \times \boldsymbol{Q} = \begin{vmatrix} R_x & R_y & R_z \\ P_x & P_y & P_z \\ Q_x & Q_y & Q_z \end{vmatrix}$$

Note: it is NOT commutative, i.e., $\boldsymbol{P}\times\boldsymbol{Q} \neq \boldsymbol{Q}\times\boldsymbol{P}$

in fact

$$-\boldsymbol{R} = \boldsymbol{Q} \times \boldsymbol{P}$$

Mathematics for CG



Line

Parametric equation of a line

Defined by two points *A* and *B*
its equation is

$$P(t) = [x(t), y(t), z(t)] = A + t(B-A); \quad -\infty < t < \infty$$

---

there are special cases:

$P(0) = A \quad P(1) = B$

Note also:

Point+tVector -> point

Line (contd.)

example:

A=[1,2,3], B=[0,1,6]

$P(t)$=[x(t), y(t), z(t)] =
= [1,2,3] + t ([0,1,6] - [1,2,3])=
= [1-t, 2-t, 3+3t]

try

t=0 $P(t)$ = [1,2,3]

t=1 $P(t)$ = [1-1,2-1,3+3] = [0,1,6]

PURDUE UNIVERSITY

Line (contd.)

Can be also thought of as a blending function

$x(t) = (1-t)*X1 + t*X2$

$y(t) = (1-t)*Y1 + t*Y2$

$z(t) = (1-t)*Z1 + t*Z2$

this ***linear blending*** is frequently used in CG:

$P(t) = A + t(B-A)$

$P(t) = (1-t)\,A + t\,B$

Note:$(1-t)+t=1$

PURDUE UNIVERSITY

Line (contd.)

*Why the explicit form is **not** good for computer graphics?*

Line is defined

$y = mx + b$

- cannot represent vertical lines
- cannot represent line segments (only infinite lines)
- cannot represent lines in 3D (!)

Plane

Parametric equation of a plane

Given by:

1) Two vectors ***P,Q*** inside the plane and a point *A*

$$R(u,v)=[x(u,v),\ y(u,v),\ z(u,v)] = A+u\boldsymbol{P}+v\boldsymbol{Q} \quad -\infty<u,v<\infty$$

Plane (contd.)

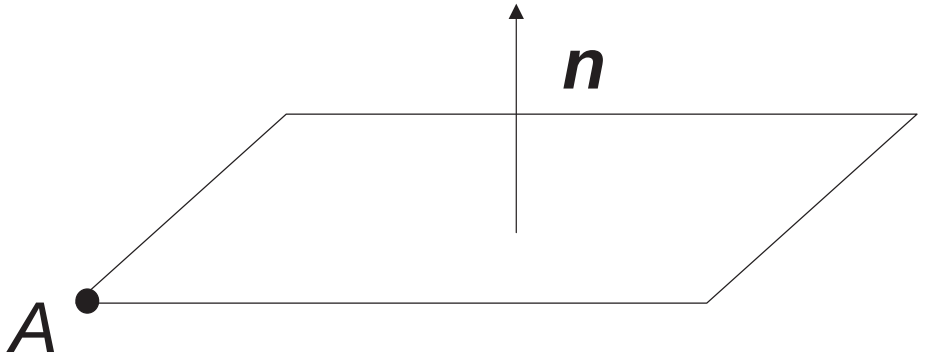
Parametric equation of a plane

Given by:

2) The normal vector ***n*** and a point *A*

any point *X*=[x,y,z] is the member of this plane iff

$(X-A)\bullet \boldsymbol{n}=0$



why?

X-*A* is a vector, A is inside the plane,

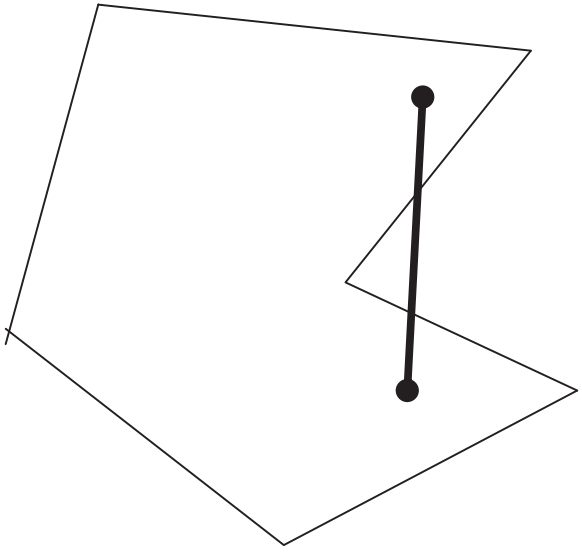
if n•(*X*-*A*)=0, they are orthogonal

Convex vs. Concave

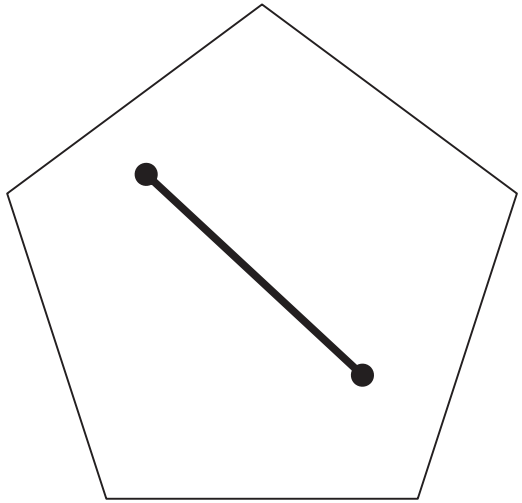
Convex vs Concave

convex set:

any line connecting two points of the set lies completely inside the set



concave



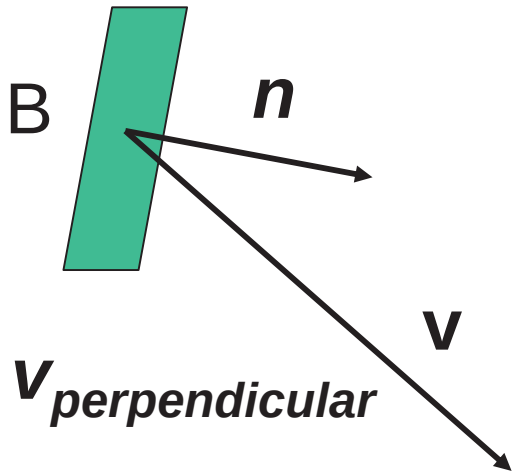
convex

Applications

Fast mirroring

Plane is defined by a point *B* and normalized normal vector ***n***

We want to mirror vector ***v*** *wrt* ***n***



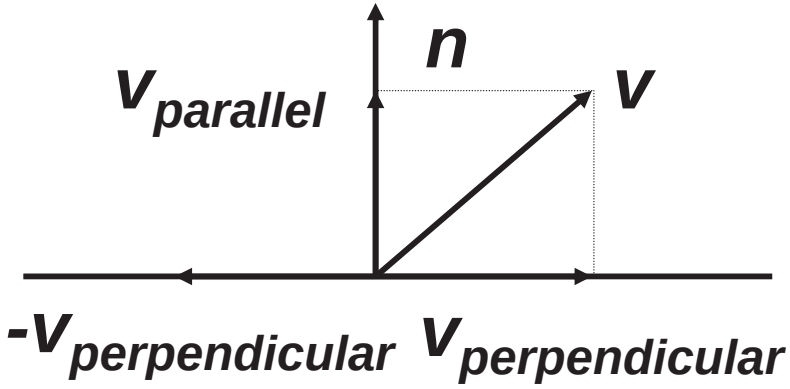
v can be expressed as $\boldsymbol{v}_{parallel}$ and $\boldsymbol{v}_{perpendicular}$

1) $\boldsymbol{v}_{parallel} = \boldsymbol{n}\ (\boldsymbol{n} \bullet \boldsymbol{v})$

2) $\boldsymbol{v}_{perpendicular} = \boldsymbol{v} - \boldsymbol{v}_{parallel}$

then the mirror reflection is

3) $\boldsymbol{v}_{mirror} = \boldsymbol{v}_{parallel} - \boldsymbol{v}_{perpendicular}$

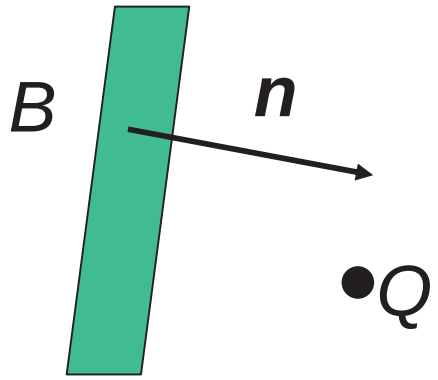


Applications

Distance of a point from a plane

Plane defined by the point *B* and
normalized normal vector ***n***
We want to compute distance of the point Q

1) Express vector ***v*** = (*Q*-*B*)
2) Evaluate projection of ***v*** onto ***n***
3) Size of this vector is the distance
dist = ***n***•(*Q*-*B*)why?



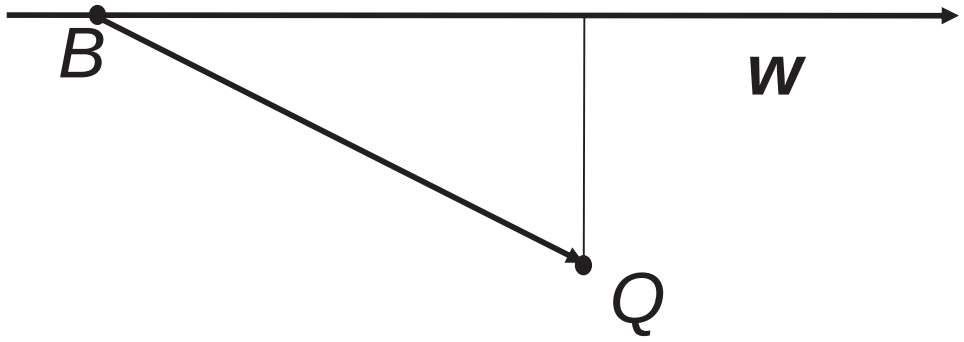
n•v = cos α . |**n**||**v**| = |**v**|cos α
i.e., projection of the **v** onto **n**

Mathematics for CG (contd.)

Distance of a point from a line

Line is defined by point *B* and normalized vector *w*
We want to compute distance of the point *Q*

1) Express vector $\boldsymbol{v}$ = (*Q*-*B*)
2) Decompose $\boldsymbol{v}$ to $\boldsymbol{v}_{parallel}$ and $\boldsymbol{v}_{perpendicular}$
3) Size of the vector $\boldsymbol{v}_{perpendicular}$ is the distance



Geometric Transformations

Why?

✓ Instancing:

an object is represented only once and its copies are expressed as its transforms this saves space!

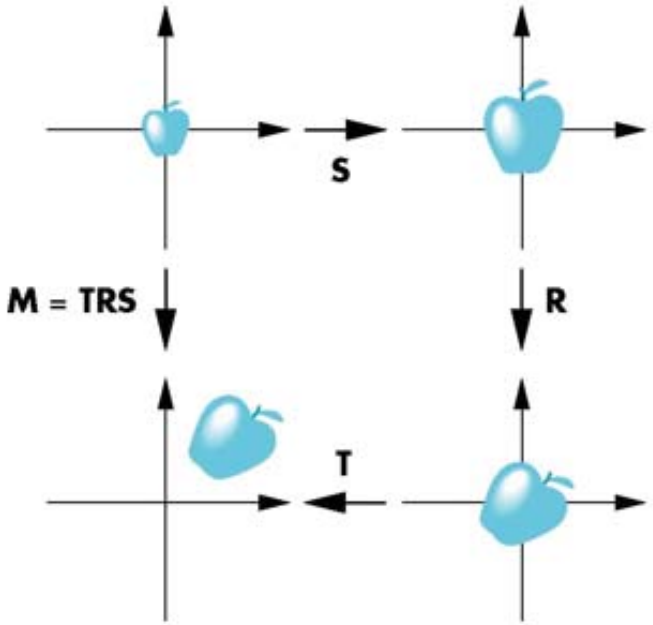
✓ Hierarchical modeling:

we can group a collection of geometry under a transform node and manipulate it all at once

Instancing

- In modeling, we often start with a simple object centered at the origin, oriented with the axis, and at a standard size
- We apply an *instance transformation* to its vertices to

1. Scale
2. Orient
3. Locate



Geometric transformations

Properties of transform:

- ✓compact representation (matrices)
- ✓fast implementation (Hardware support)
- ✓easy to invert (inverse matrix)
- ✓easy to compose (matrix multiplication)

PURDUE UNIVERSITY

Coordinates

Transformations of a single point
>> Transform of an object is a transform of all of its points(vertices).

Point is expressed by its coordinates *P*=[x,y,z]
but in CG we usually use *homogenous coordinates*

the homogenous coordinates of a point P are
P=[X,Y,Z,W],
where
x=X/W, y=Y/W, z=Z/W

we usually put W=1

Homogeneous Coordinates and Computer Graphics

- Homogeneous coordinates are key to all computer graphics systems
  - All standard transformations (rotation, translation, scaling) can be implemented with matrix multiplications using 4 x 4 matrices
  - **Hardware pipeline works with 4 dimensional representations**
  - For orthographic viewing, we can maintain $w=0$ for vectors and $w=1$ for points
  - For perspective we need a *perspective division*

Geometric transformations (contd.)

there are several advantages of this:

- we can express affine transforms (see next) as one matrix
- we can express projections as one matrix as well
- transforms can be composed by matrix multiplication
- compact representation of points and vectors

points $W \neq 0$
vectors $W = 0$

Geometric transformations (contd.)

example:
point *P* has homogenous coordinates [2, 4, 12, 2]
What are its **Cartesian coordinates**?
They are *P*=[2/2, 4/2, 12/2]=[1,2,6]

example:
Point has Cartesian coordinates [1,2,3]

What are its homogenous coordinates?

[1w,2w,3w,w] and w $\neq$0

for example [1,2,3,1], [2,4,6,2],etc.

Geometric transformations (contd.)

Transformations:

linear (translation, scale, rotation, shear)

non-linear

Linear are expressed as a matrix 3x3, affine by 4x4

$$\mathbf{M} = \begin{bmatrix} m11 & m12 & m13 & m14 \\ m21 & m22 & m23 & m24 \\ m31 & m32 & m33 & m34 \\ m41 & m42 & m43 & m44 \end{bmatrix}$$

P is transformed to P' by multiplication by the matrix

$P'^T = \mathbf{M}\, P^T$ i.e.,

$$P' = [X', Y', Z', W'] = [m_{11}X + m_{12}Y + m_{13}Z + m_{14}W,\ m_{21}X + m_{22}Y + m_{23}Z + m_{24}W,\ m_{31}X + m_{32}Y + m_{33}Z + m_{34}W,\ m_{41}X + m_{42}Y + m_{43}Z + m_{44}W]$$

Geometric transformations - Translation

Translation:

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & X_T \\ 0 & 1 & 0 & Y_T \\ 0 & 0 & 1 & Z_T \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

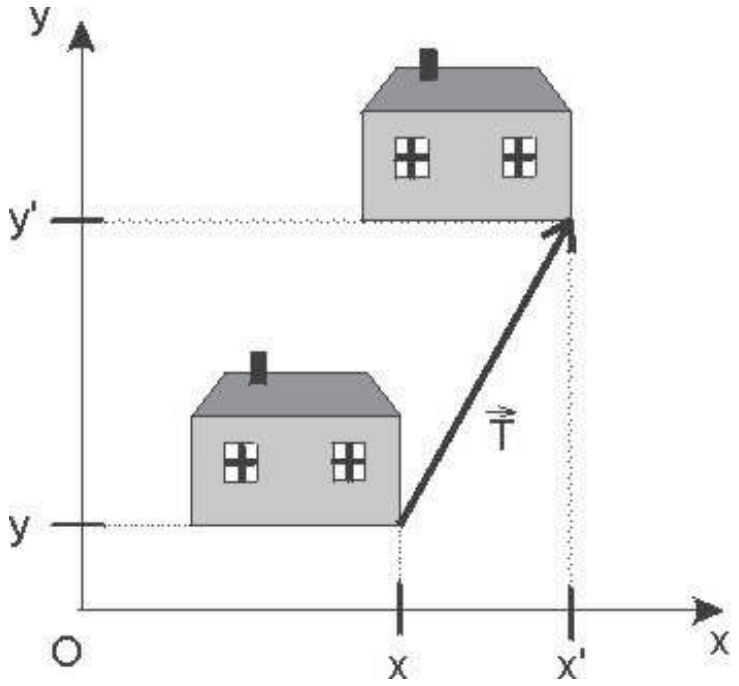
what is in fact:

$X' = X + X_T$

$Y' = Y + Y_T$

$Z' = Z + Z_T$

$W' = 1$

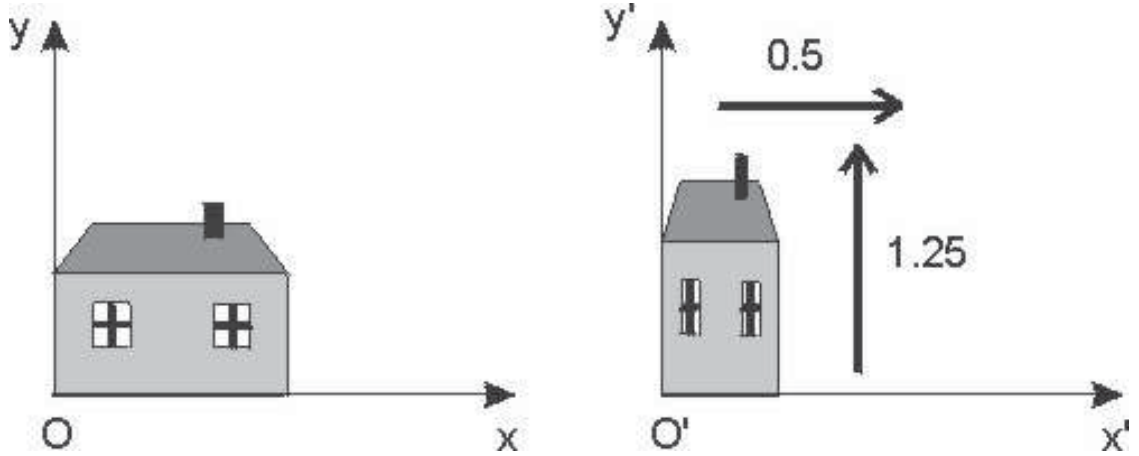


The point is moved in the direction of the vector ***T***
(Remember point+vector yields to point)

Geometric transformations - Scale

Scale:

$$\mathbf{M} = \begin{bmatrix} S_X & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



what is in fact:

$X' = S_X X$

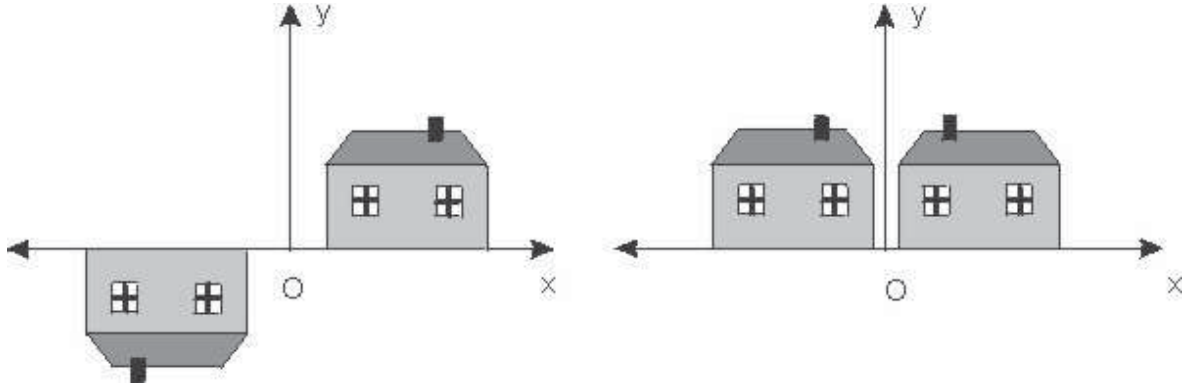
$Y' = S_Y Y$

$Z' = S_Z Z$

$W' = 1$

Geometric transformations - Symmetry

Symmetry:
is a special case of scale for
some of the coefficients
$S_X = -1$
$S_Y = -1$
$S_Z = -1$



Axis symmetry:
$S_X = -1\ X$
$S_Y = -1\ Y$
$S_Z = -1\ Z$

Geometric transformations - Symmetry

Plane symmetry:

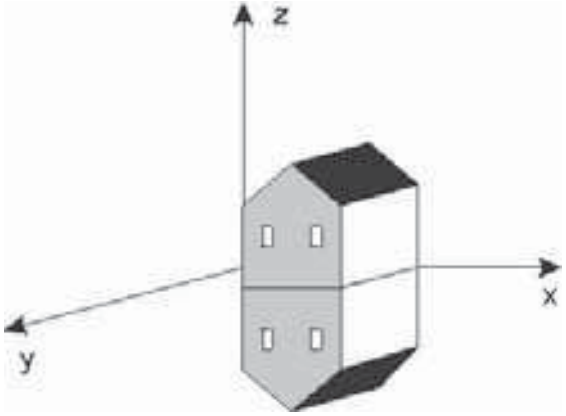
$S_X = -1$ and $S_Y = -1$ symmetry according to the plane *xy*

$S_X = -1$ and $S_Z = -1$ symmetry according to the plane *xy*

$S_Y = -1$ and $S_Z = -1$ symmetry according to the plane *yz*

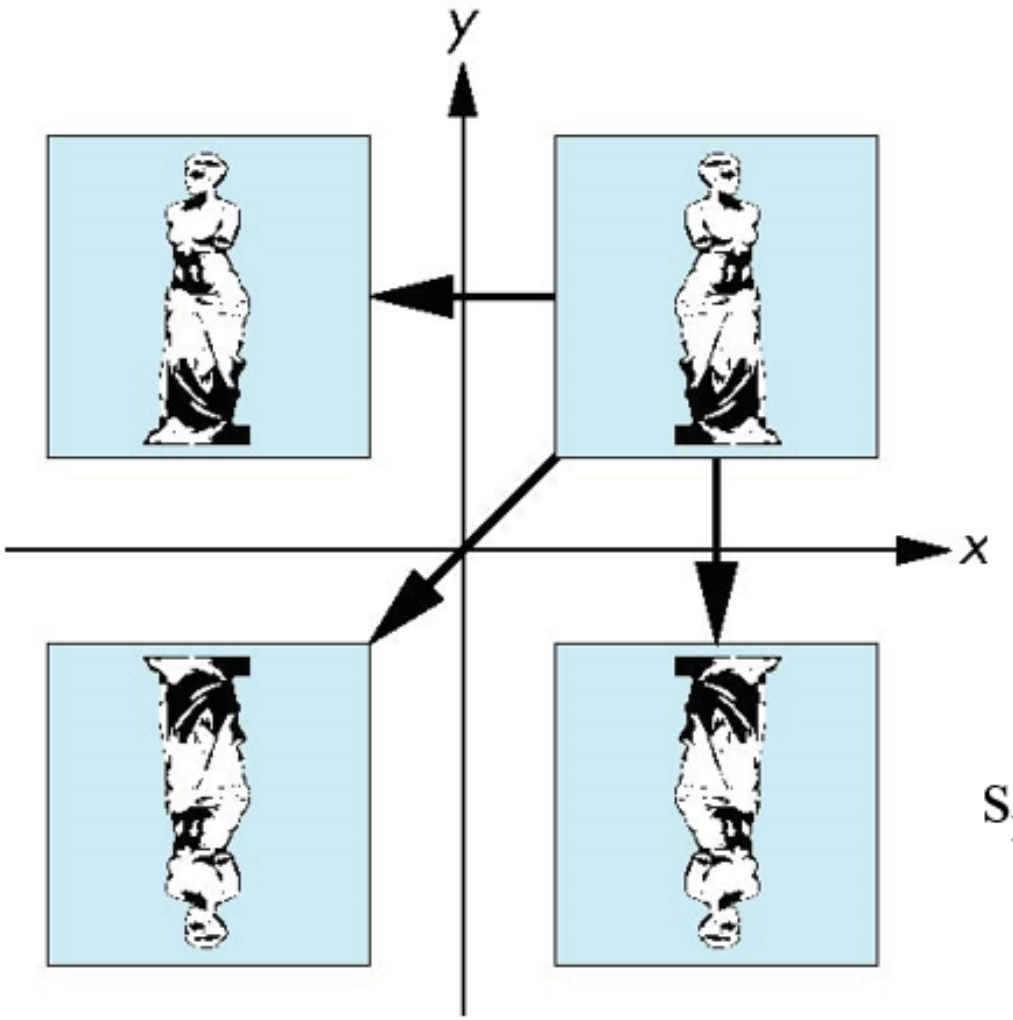
Symmetry according to the origin:

$S_X = S_Y = S_Z = -1$



Reflection

corresponds to negative scale factors



$s_x = -1\ s_y = 1$

original

$s_x = -1\ s_y = -1$

$s_x = 1\ s_y = -1$

Geometric transformations - Shear

Shear in direction

YZ

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ SH_Y & 1 & 0 & 0 \\ SH_Z & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

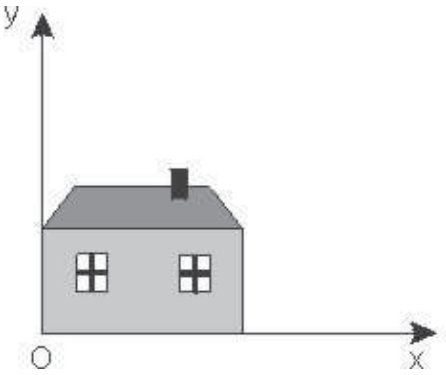
XZ

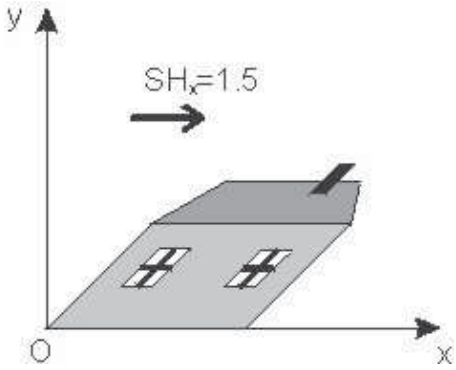
$$\mathbf{M} = \begin{bmatrix} 1 & SH_X & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & SH_Z & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

XY

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & SH_X & 0 \\ 0 & 1 & SH_Y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

two dimensional example



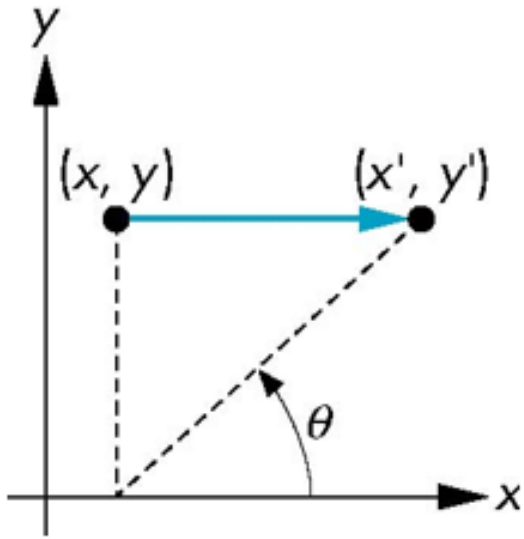


shear in the direction of the *X* axis

Shear Matrix

Consider a simple shear along x-axis

$$x' = x + y \cot \theta$$
$$y' = y$$
$$z' = z$$



$$\mathbf{H}(\theta) = \begin{bmatrix} 1 & \cot\theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Orientation

- We will define ‘orientation’ to mean an object’s instantaneous rotational configuration
- Think of it as the rotational equivalent of position

Geometric transformations - Rotation

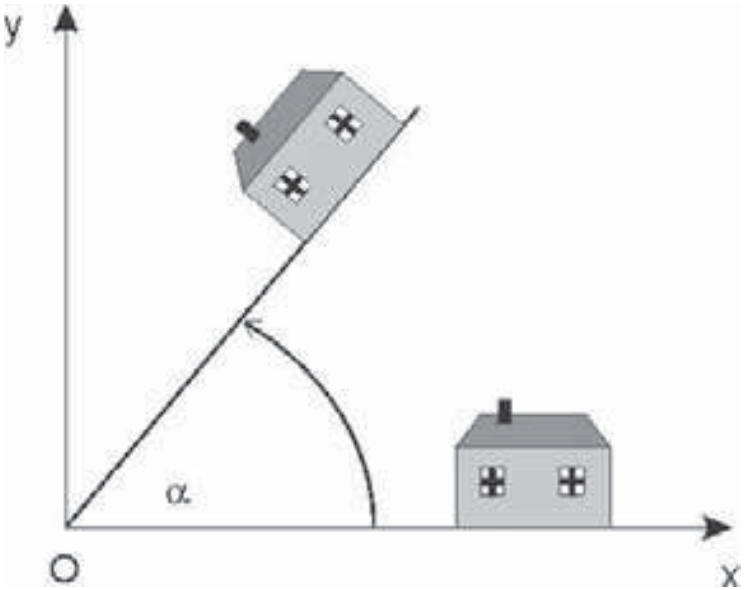
Around axis *X* of angle α

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Around axis *Y* of angle α

$$\mathbf{M} = \begin{bmatrix} \cos\alpha & 0 & -\sin\alpha & 0 \\ 0 & 0 & 0 & 0 \\ \sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

note that $\sin\alpha$ and $\cos\alpha$ can be precomputed!!



Around axis *Z* of angle α

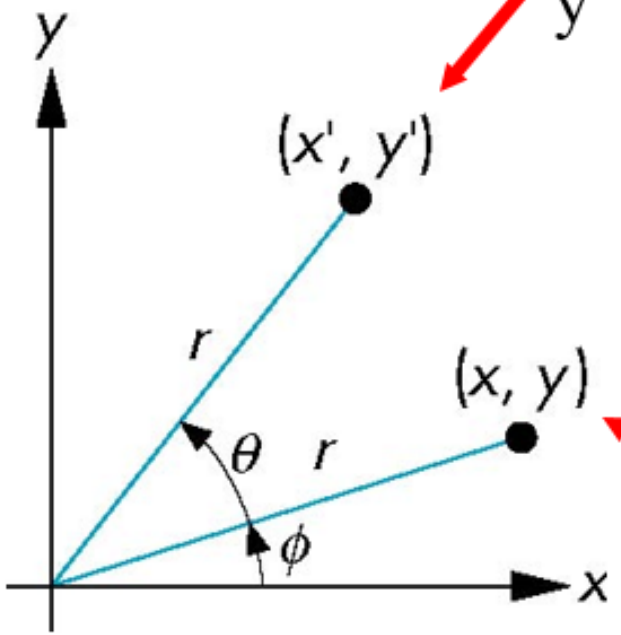
$$\mathbf{M} = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation (2D)

Consider rotation about the origin by θ degrees

- radius stays the same, angle increases by θ

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



remember

$$\sin(\alpha \pm \beta) = \sin\alpha\cos\beta \pm \cos\alpha\sin\beta$$

$$\cos(\alpha \pm \beta) = \cos\alpha\cos\beta \mp \sin\alpha\sin\beta$$

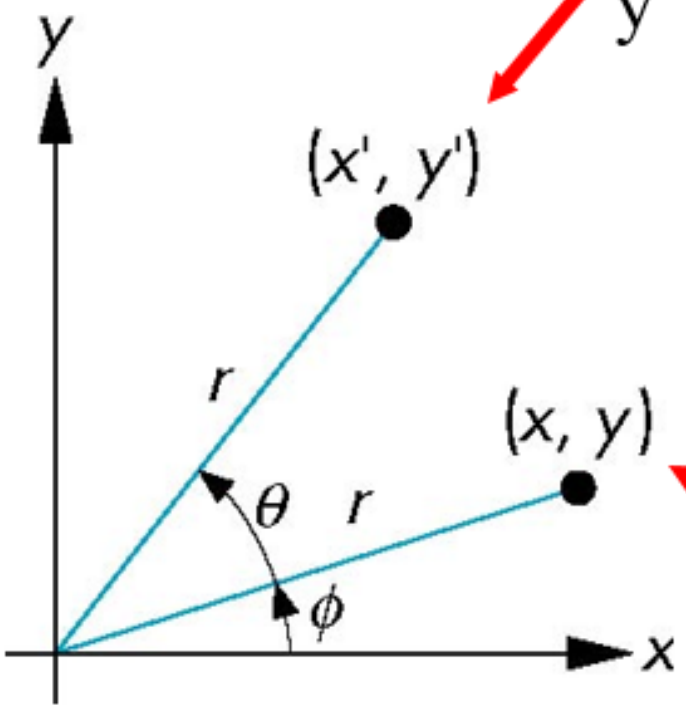
$$x = r\cos\phi$$
$$y = r\sin\phi$$

Rotation (2D)

Consider rotation about the origin by θ degrees

- radius stays the same, angle increases by θ

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



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

$$x = r\cos\phi$$
$$y = r\sin\phi$$

Rotation about the z-axis

- Rotation about z-axis in three dimensions leaves all points with the same z
  - Equivalent to rotation in two dimensions in planes of constant z

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

  - or in homogeneous coordinates

$$\mathbf{p}' = \mathbf{R}_z(\theta)\,\mathbf{p}$$

Rotation Matrix

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

Rotation about x and y axes

- Same argument as for rotation about *z*-axis
  - For rotation about x-axis >> x is unchanged
  - For rotation about y-axis >> y is unchanged

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

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

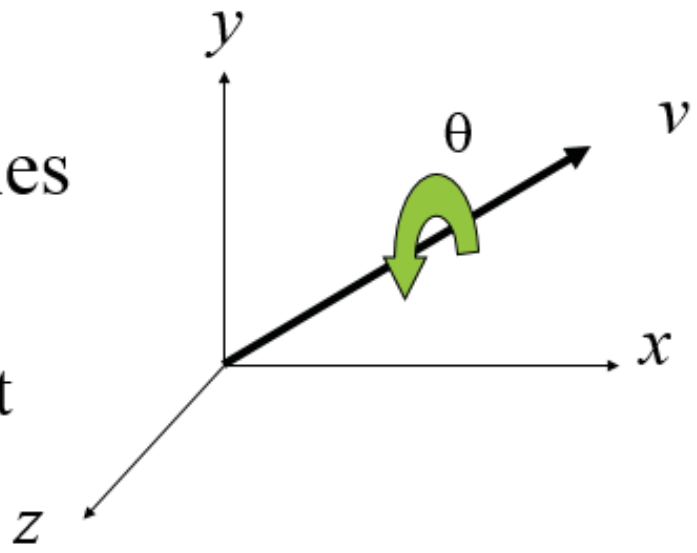
General Rotation About the Origin

A rotation by θ about an arbitrary axis can be decomposed into the concatenation of rotations about the *x*, *y*, and *z* axes

$$\mathbf{R}(\theta) = \mathbf{R}_z(\theta_z)\,\mathbf{R}_y(\theta_y)\,\mathbf{R}_x(\theta_x)$$

$\theta_x\,\theta_y\,\theta_z$ are called the Euler angles

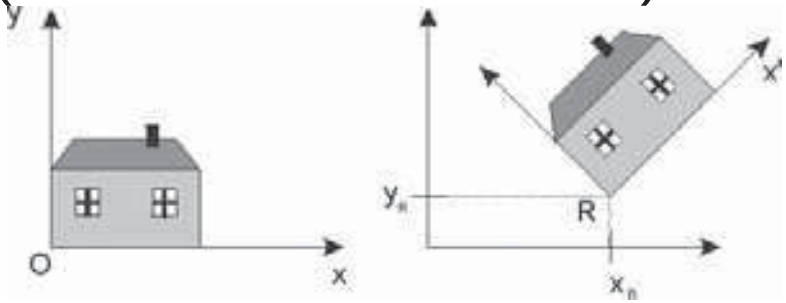
Note that rotations do not commute
We can use rotations in another order but with different angles



Geometric transformations - Example

Define matrix for rotation around line Q=[10, 20, t] and α

It is rotation about axis *Z* but moved to the point [10, 20, 0]
Its is composition of two matrices (translation and rotation)



1) Move point to the origin
2) Rotate about axis *Z* and α
3) Move back

Let **T** be the matrix for translation to the fixed point. Let **R** be the matrix for rotation around the *Z* and let $\mathbf{T}^{-1}$ be the matrix for inverse translation

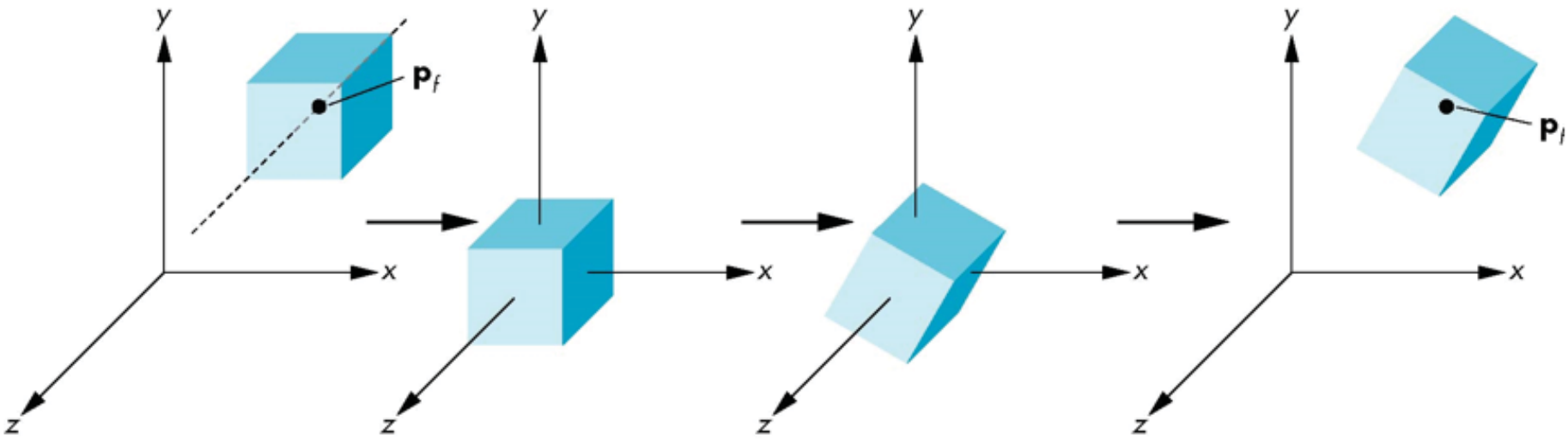
$$\mathbf{M} = \mathbf{T}\,\mathbf{R}\,\mathbf{T}^{-1}$$

We are multiplying from the right,
so the order of the matrices is exactly the same
as the order of the performed operations

Rotation About a Fixed Point other than the Origin

1. Move fixed point to origin
2. Rotate
3. Move fixed point back

$$\mathbf{M} = \mathbf{T}(p_f)\,\mathbf{R}(\theta)\,\mathbf{T}(-p_f)$$



3D Transformations

$$\begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translation Using Column Vectors

$$\begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling

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

Rotation About x axis

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

Rotation about y axis

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

Rotation about z axis

PURDUE UNIVERSITY

Projections

Projections transform point from m-dimensional space into n-dimensional space and m>n

We are interested in m=3 and n=2 i.e.,
i.e., from 3D space to the plane

we need to define two things:

center of projection (**COP**)

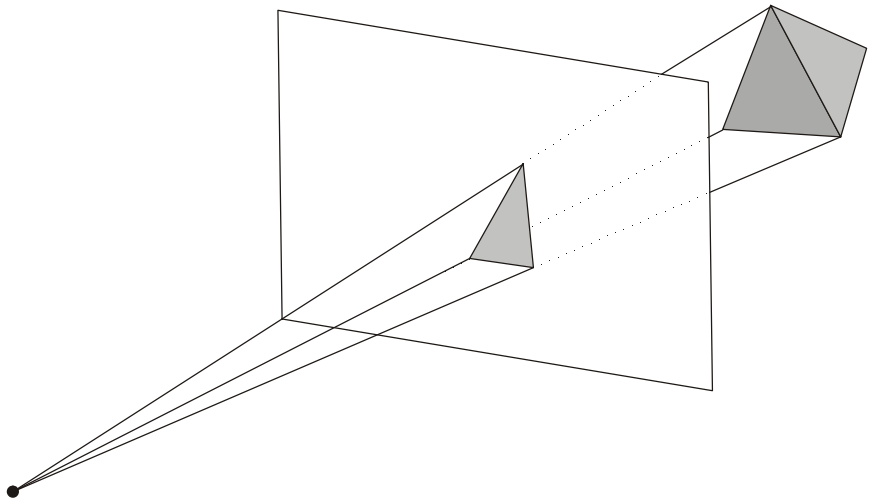
plane of projection (**PP**)

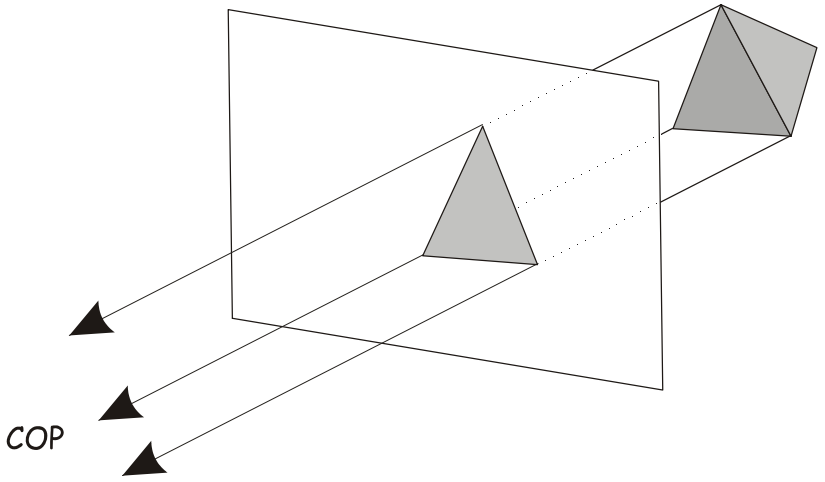
Projections

There are two basic types of projections

✓ **Perspective** projection:
COP is in finite distance from PP

✓ **Parallel** projection:
distance between COP and PP is infinite





Projections (contd.)

Properties:

Perspective projection:

~ Size of the projected objects varies with distance from PP

~ It looks realistic

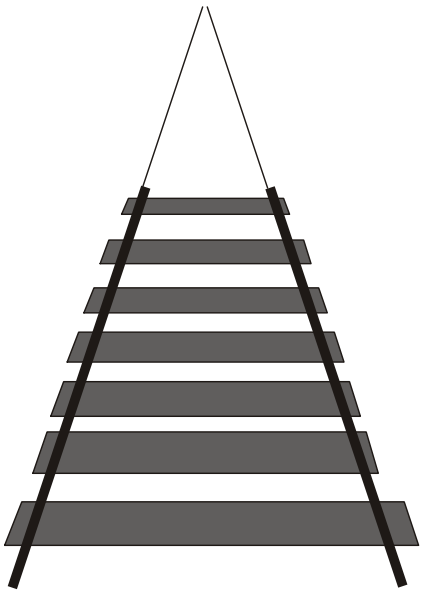
~ Distances and angles are not preserved

~ Parallel lines do not remain parallel

~ Lines that are not parallel to PP meet at **vanishing point**

~ Center of a line is not projected as a center of a projected line

~ Used in photorealistic rendering



PURDUE UNIVERSITY

Projections (contd.)

Parallel projection:

~ Size of the projected object
  does not depend on distance from PP
~ Good for exact measurements
~ Parallel lines remain parallel after projection
~ Angles are not necessarily preserved
~ Center of a line is projected as
  a center of a projected line
~ Used in CAGD and CAD

Projections (contd.)

Perspective projection example:

Projections (contd.)

Derivation of the equations:

Let *A*=[x,y,z] be the projected point and
PP be the plane z=d

We denote the projected point *A*'=[x', y', d]

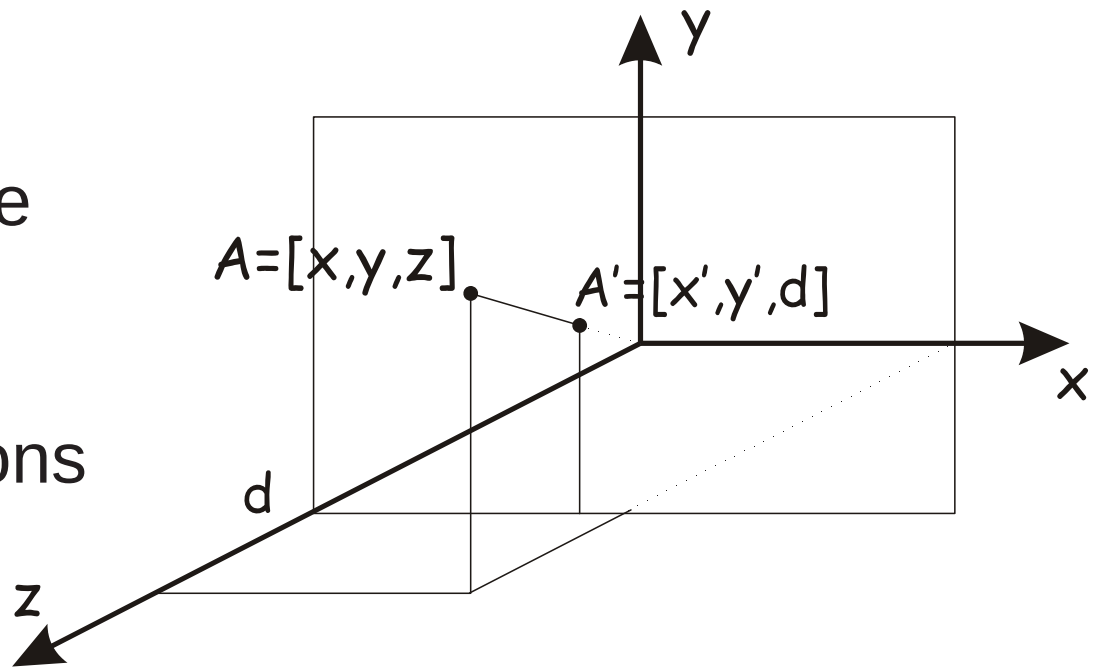
Parallel projection:

A'=[x,y,0] i.e.,
x' = x
y' = y
we simply "forget" the *z* coordinate

Projections (contd.)

Perspective projection:

assume *COP*=[0,0,0]
thus line connecting point *A* and *COP* has equation *P*(t) = COP + t(*A* - *COP*)
i.e.,
P(t) = [t x, t y, t z]
for plane *xy* we have
d= t z
i.e., t= d/z
this leads to equations
x' = x d/z
y' = y d/z



Projections (contd.)

In CG we use different spaces:
1) *Object space*
used for describing objects
The objects have position and orientation.

2) *Camera space*
Corresponds to the camera (or eye).
The camera is placed in [0,0,0] and
corresponds to COP.
Sometimes called the eye space

Projections (contd.)

3) *Image space*
result of the projection.
It is usually parallel to the plane *xy* in eye space.
Sometimes called
Normalized Device Coordinate Space (NDC)

4) *Screen space*
corresponds to final rasterized image.
It has discrete coordinates - the image resolution

PURDUE UNIVERSITY

Projections (contd.)

Camera space → Image space transformation

at the beginning, camera is usually placed at the origin [0,0,0] corresponds to COP

screen is placed at distance *d* and is parallel to *xy* corresponds to PP

the transformation can be expressed as a matrix:

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

Projections (contd.)

Camera space→screen space transformation

if we write the general point transform as
$A' = \mathbf{M}\, A^T$

thus we have:

$A' = [x', y', z', 1] = \mathbf{M}\, [x,y,z,1]^T = [x, y, z, z/d]$

and while it is transformed to
Cartesian coordinates
from homogenous ones we get:

$A' = [x', y', z'] = [x\ d/z, y\ d/z, d]$

Projections (contd.)

Camera space $\rightarrow$ Image space transformation
is there any matrix for parallel projection?Yes, this one:

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & d \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

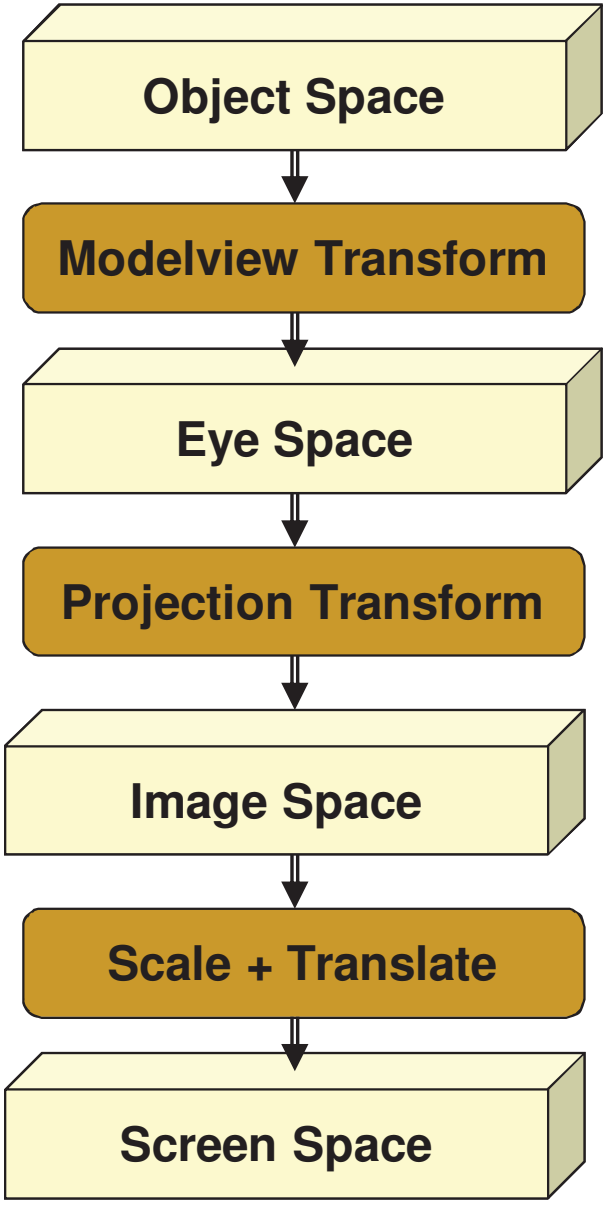
after multiplication

$$A' = [x', y', z', 1] = \mathbf{M}\,[x,y,z,1]^T = [x, y, d, 1]$$

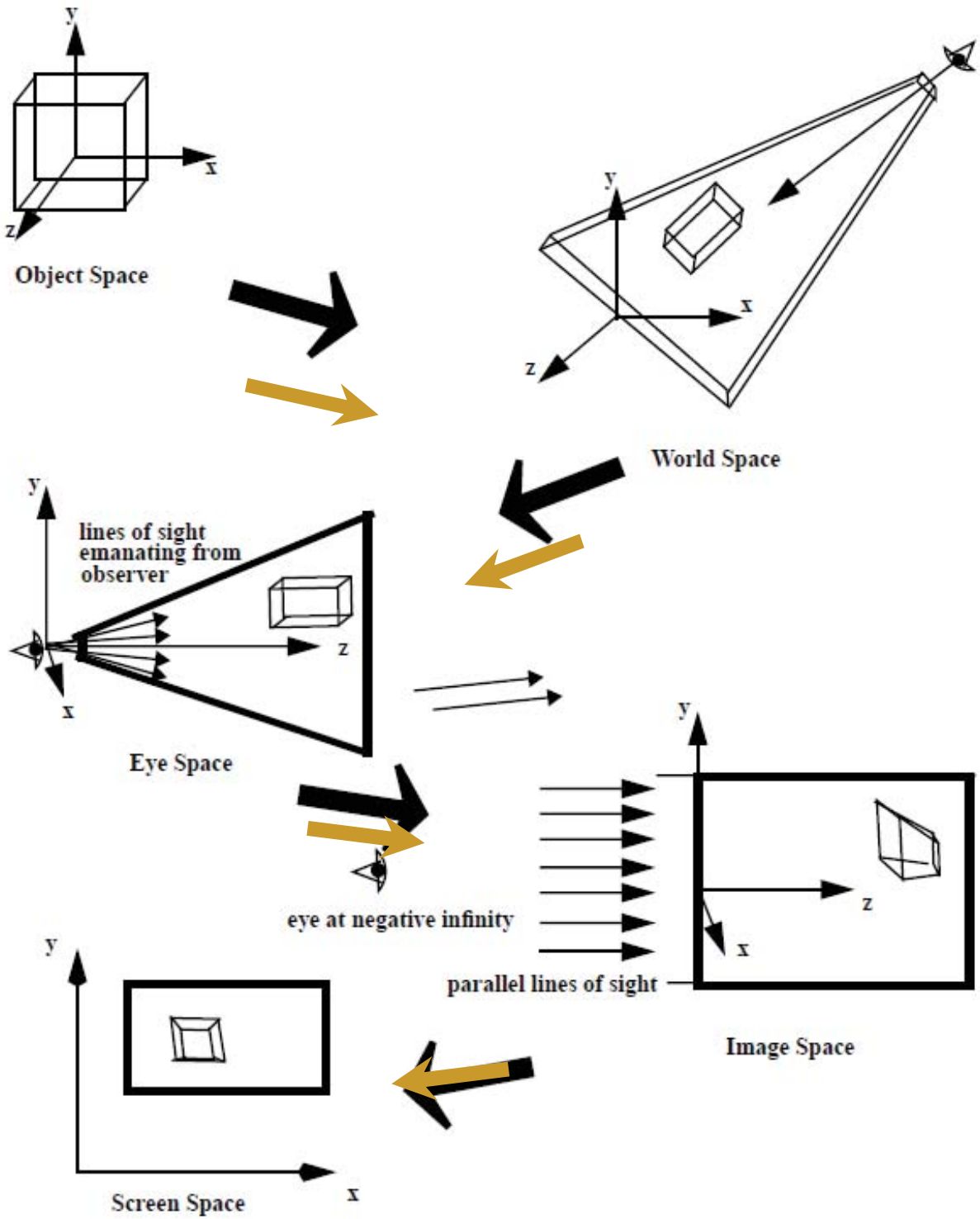
Transformations

- Object primitives transformed from an object space to screen space at each frame
- Coordinate systems can be left-handed or right-handed

Display Pipeline

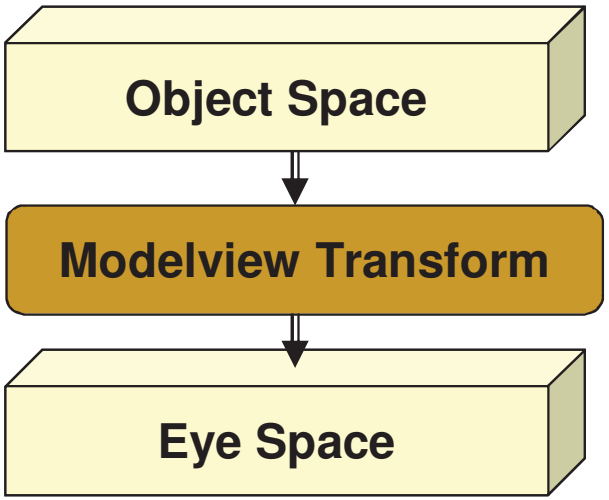


Display Pipeline

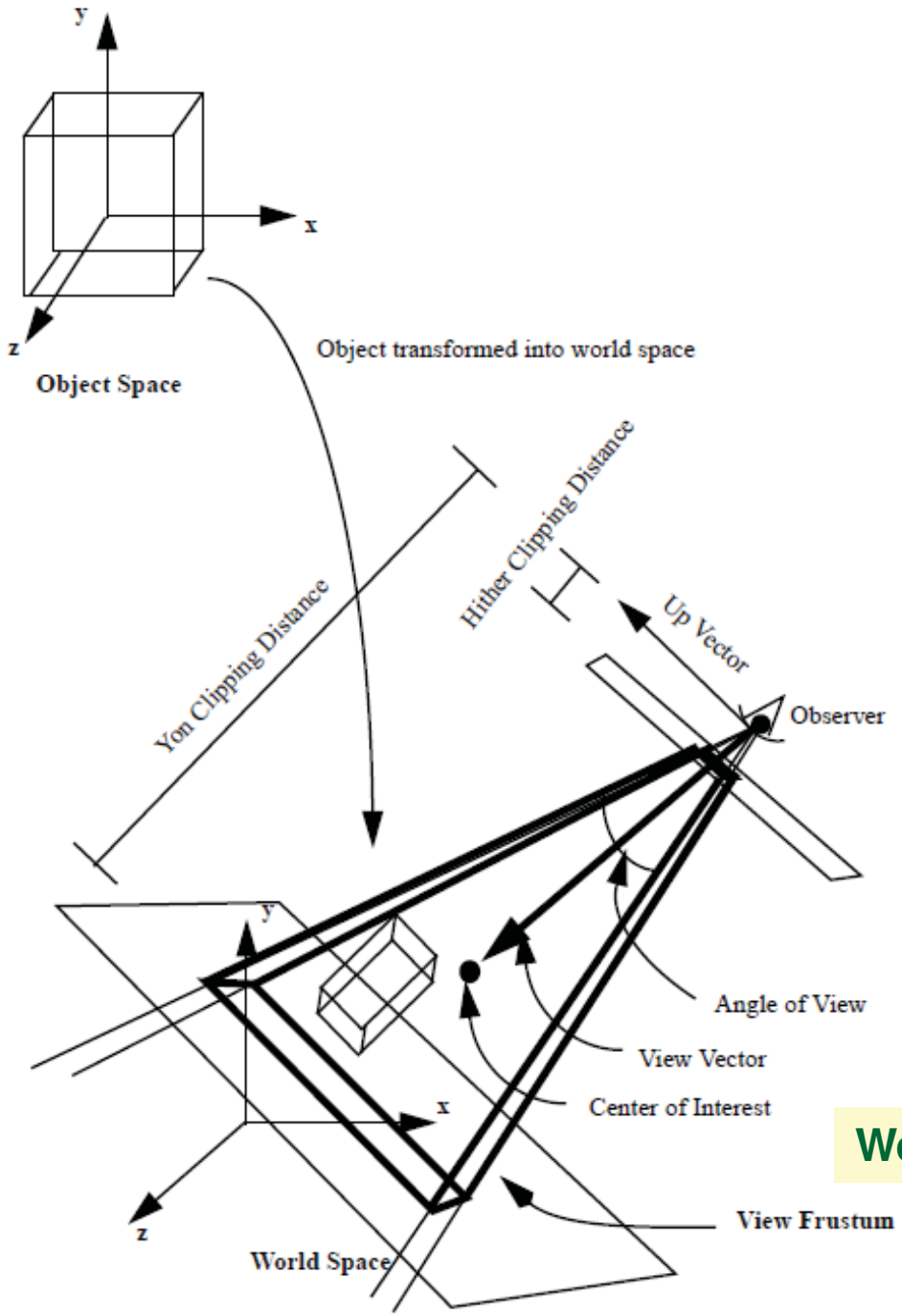


Modelview transform

- Initially, each object is defined in its *object space*
- Modelview includes both viewing and modeling transforms:
  - **Modeling transform:** assembles all objects into a *world space*
  - **View transform:** orients entire collection of objects w.r.t. camera position



Modeling transform

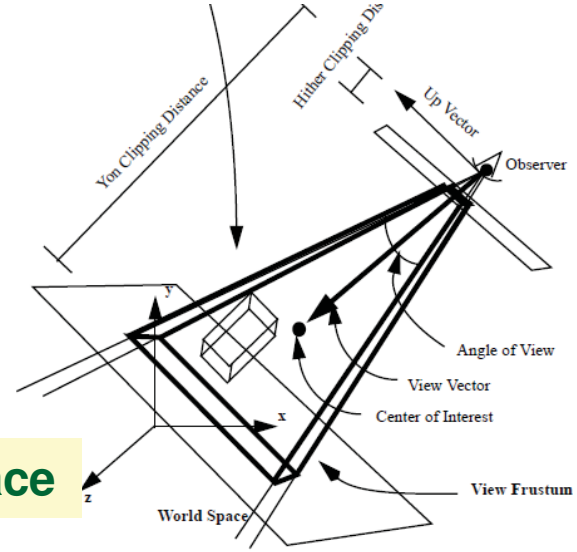


World space

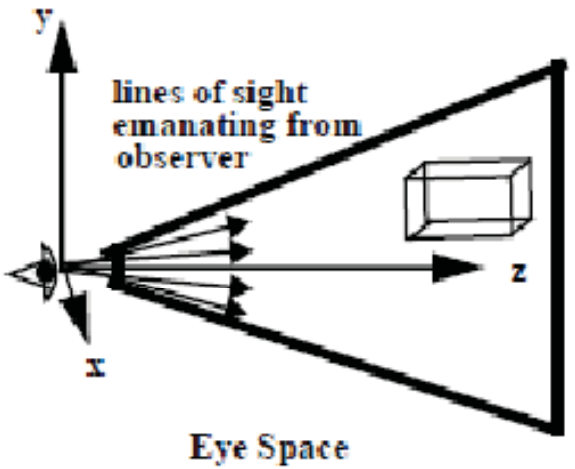
Modelview transform

- Eye space: world space oriented w.r.t. camera position
  - Viewer position at origin

- Note: APIs use different conventions
  - E.g. OpenGL: looking towards negative z axis



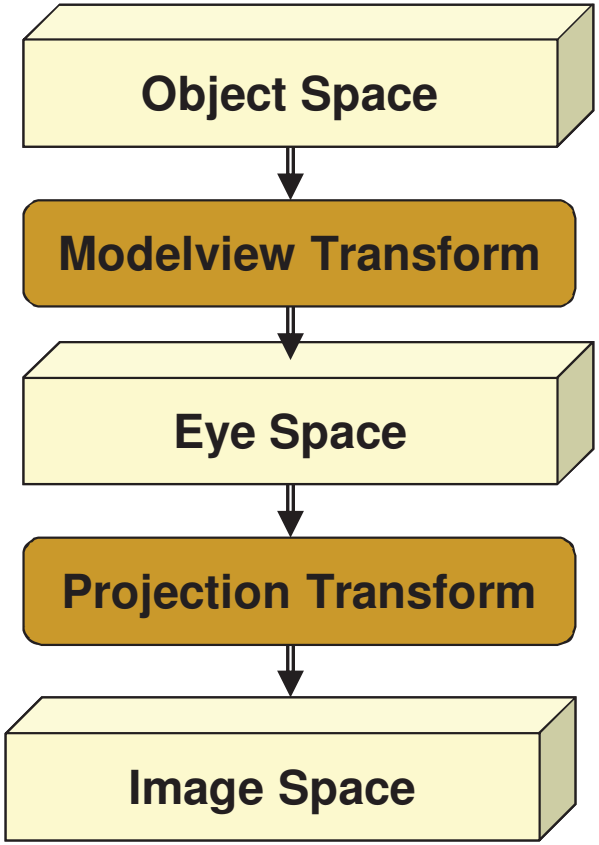
World space



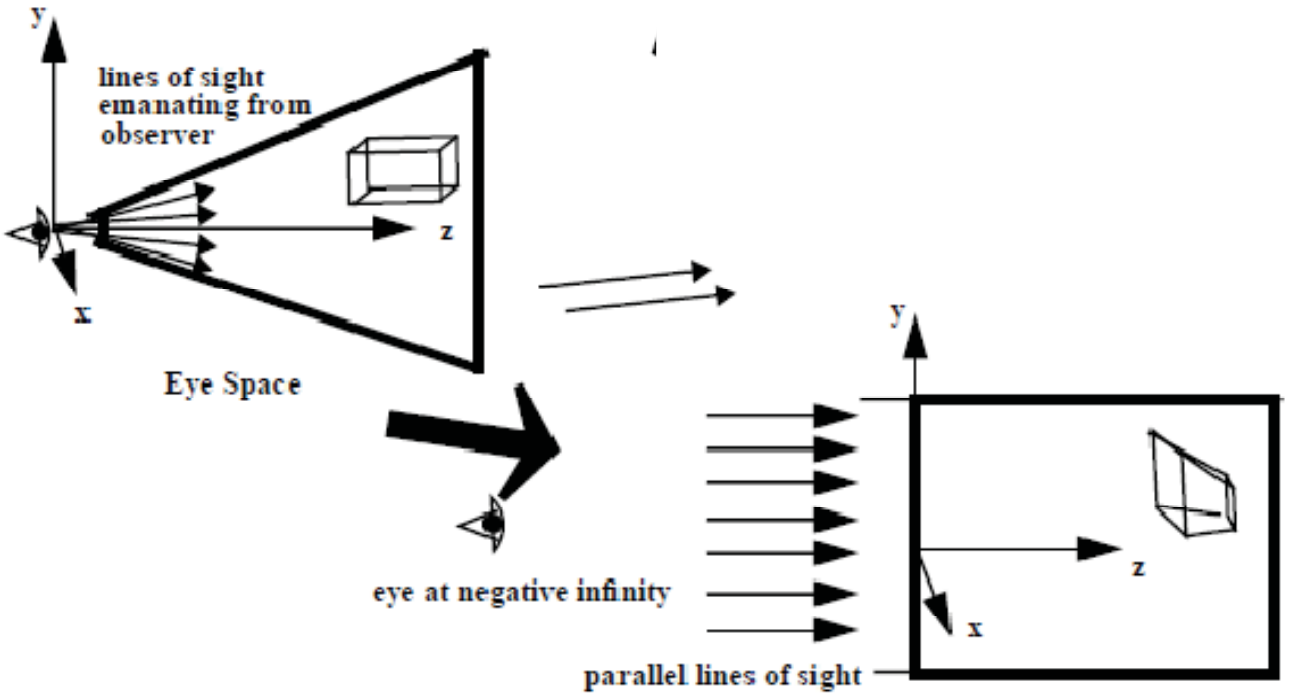
Eye space

Projection Transform

- Next step: projection transform from eye space to image space
- Viewing frustum becomes a rectangular solid (cuboid)

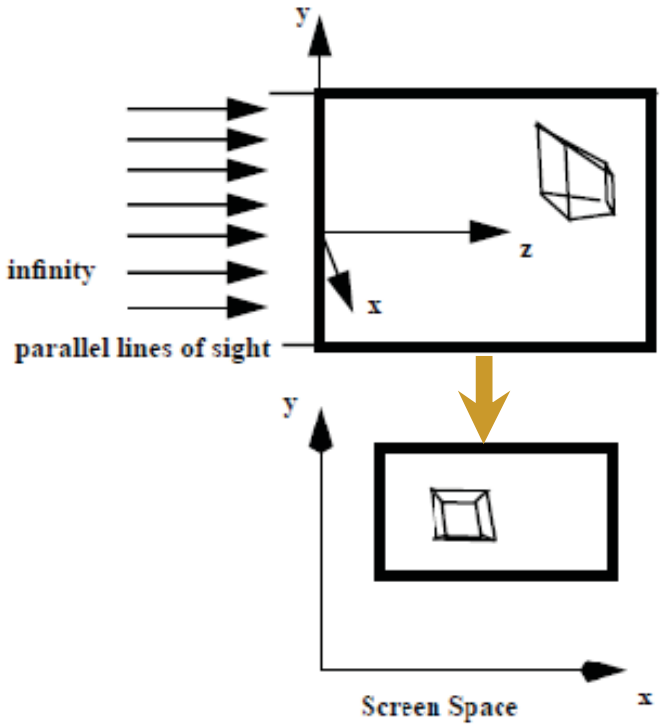


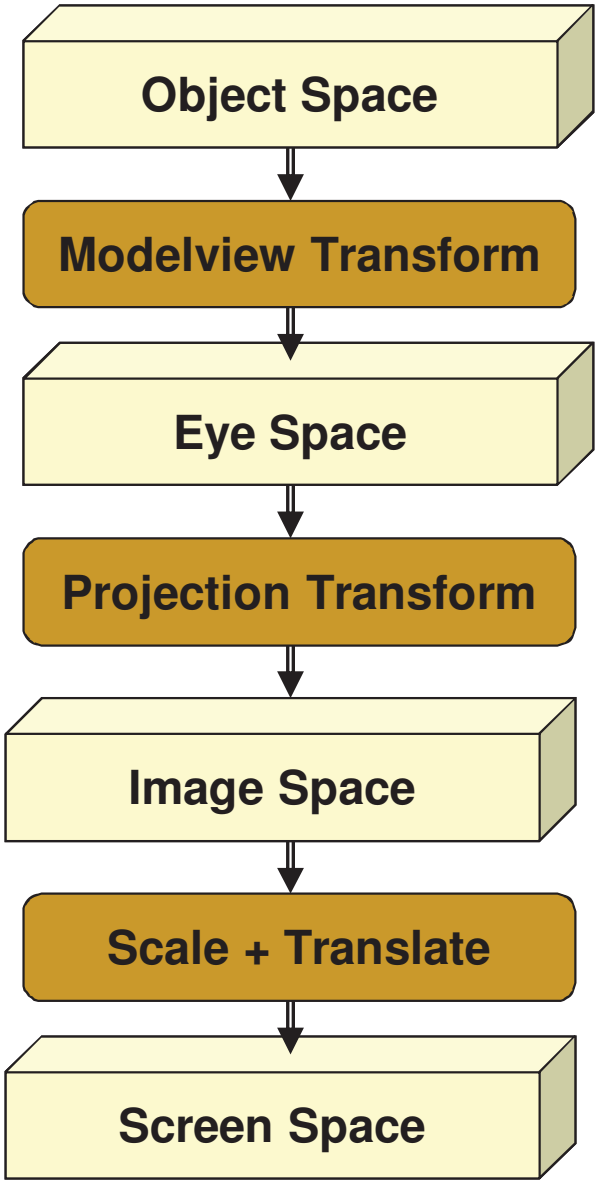
Projection Transform



Screen Space

- Last transform:
  - Image Space → Screen space





Homogeneous Coordinates & Transformation Matrix

Homogeneous Coordinates

Homogeneous Coordinates:

$$\left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right) = [x, y, z, w]$$

$$(x, y, z) = [x, y, z, 1]$$

Point represented as column vector:

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

Transformations

Basic Transformations:

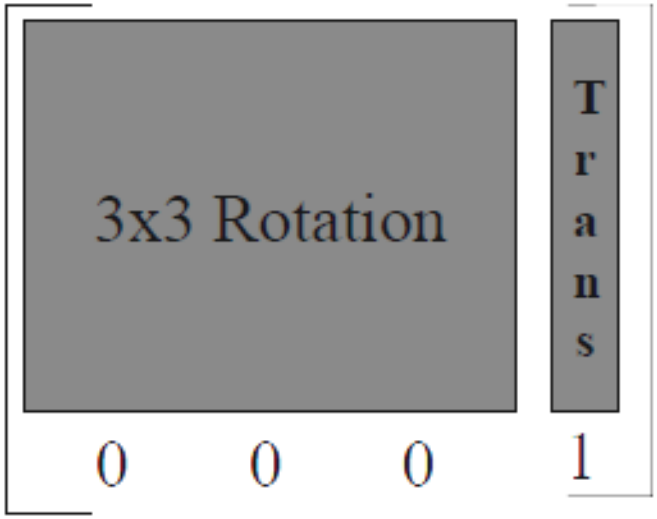
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & m \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$P' = M_1 M_2 M_3 M_4 M_5 M_6 P$$

Compounding Transformations:

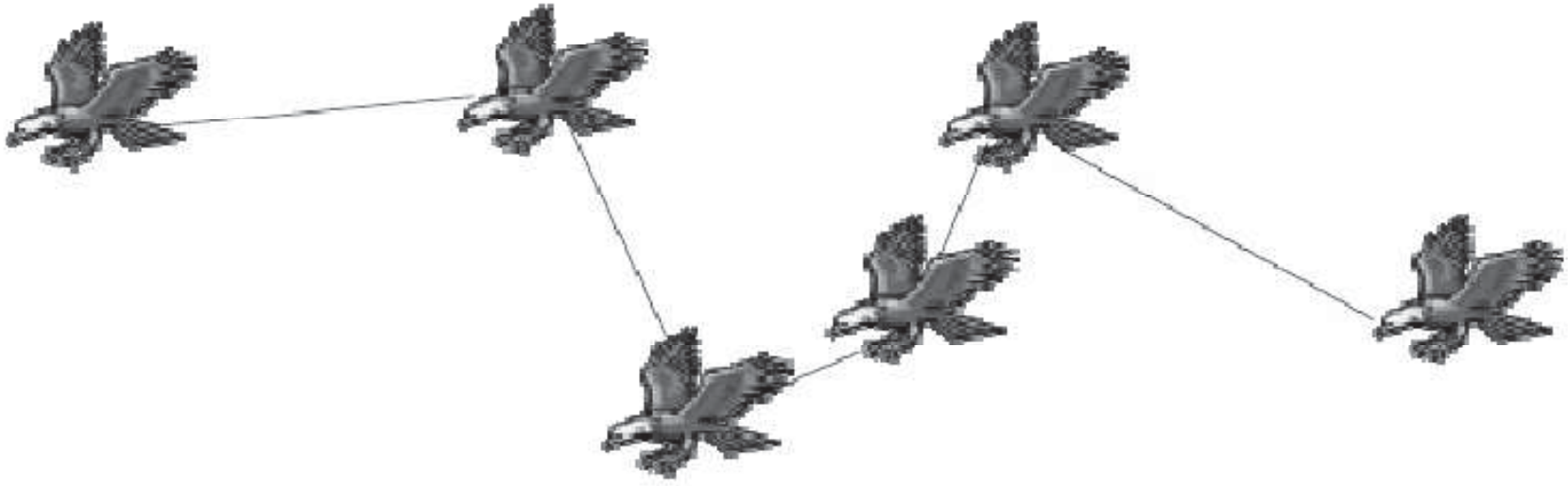
$$M = M_1 M_2 M_3 M_1 M_5 M_6$$

$$P' = MP$$



3D Transformations in Keyframing

- Animator specifies important keyframes
- Computer generates the in-between frames automatically using interpolation



Interpolating Translations is Straightforward

- Linearly interpolate t_x, t_y, t_z

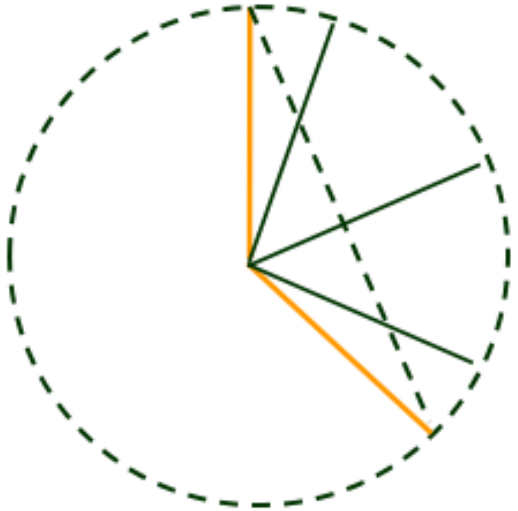
$$\begin{bmatrix} x+t_x \\ y+t_y \\ z+t_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Interpolating Rotations

- The upper left 3x3 submatrix of a transformation matrix is the rotation matrix
- Maybe we can just interpolate the entries of that matrix to get the in-between rotations?
- Problem:
  - Rows and columns are orthonormal (unit length and perpendicular to each other)
  - Linear interpolation doesn't maintain this property, leading to nonsense for the inbetween rotations

Interpolating Rotations

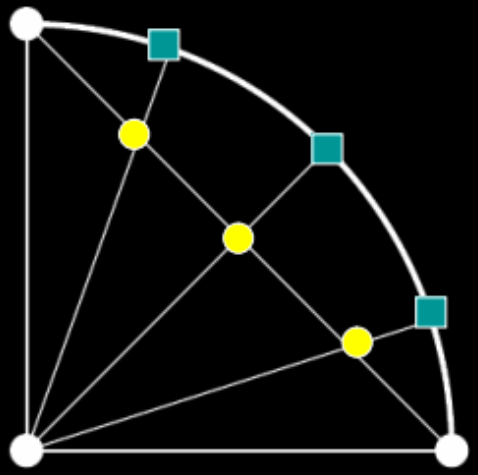
- Look at lerp diagram
- Orange vectors are basis vectors



- Get shorter in the middle!
  - Covers more arc in the middle
  - I.e. rotates slower on the edges, faster in the middle

Interpolating Rotations

Linear interpolation generates unequal spacing of points after projecting to circle



Interpolating Rotations

Example:

- Interpolate linearly from a positive 90 degree rotation about y to a negative 90 degree rotation about y

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix} \longrightarrow \begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

- Linearly interpolate each component and halfway between, you get this…

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Not a rotation matrix
Not orthonormal
No sense

Orientation Representation Methods

- Transformation matrices
- ???
- ???
- ???
- ???

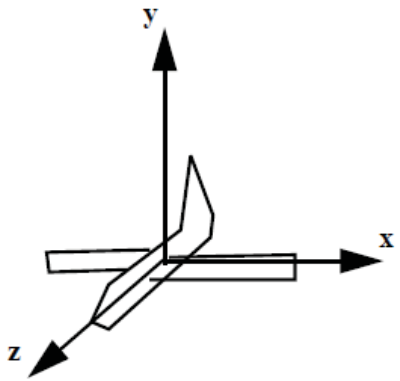
Orientation Representation Methods

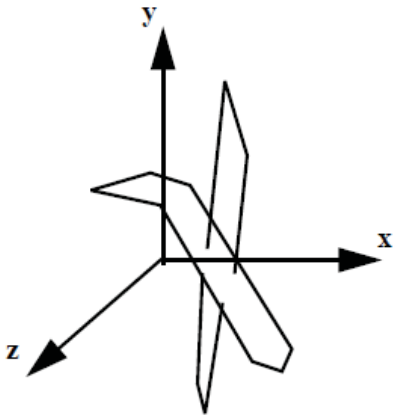
- Transformation matrices
- **Fixed angle representation**
- ???
- ???
- ???

Fixed Angle Representation

- Angles used to rotate about fixed axes
- Many possible orderings: x-y-z, x-y-x,y-x-z

  - as long as axis does immediately follow itself such as x-x-y
- A rotation of 10, 45, 90 would be written as
  - P’ = Rz(90) Ry(45) Rx(10) P
  - We want to first rotate about x, then y, then z.



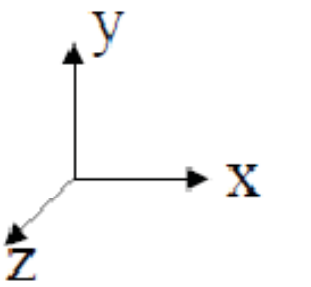


Fixed Angle Representation

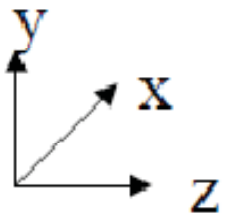
- Gimbal Lock: problem occurs when two of the axes of rotation line up.
  - A basic problem with representing 3-D rotations using Fixed angles or Euler angles

Gimbal Lock

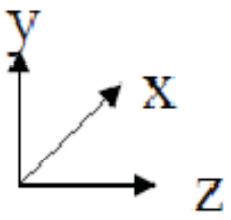
- A 90 degree rotation about the y axis aligns the first axis of rotation with the third.
- Incremental changes in x,z produce the same results → **lost a degree of freedom**



Rotx(0)



Roty(90)



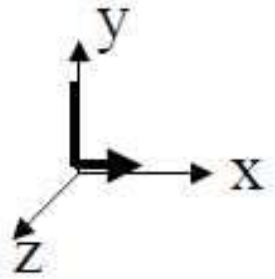
Rotz(0)

Fixed Angles in Interpolation

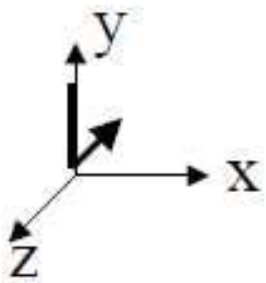
- Fixed angle also has problem while interpolating.
  - e.g. (0,90,0) → (90,45,90)
  - Both start and end rotations in yz plane

    - Halfway between (0, 90, 0) & (90, 45, 90)
    - Interpolate directly, get (45, 67.5, 45)
    - Desired result is (90, 22.5, 90)

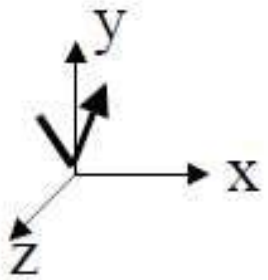
- With 45 rotation from first orientation to next, we expect (90, 22.5, 90), but get (45, 67.5, 45)
  - Object will go out of yz plane → not expected behavior



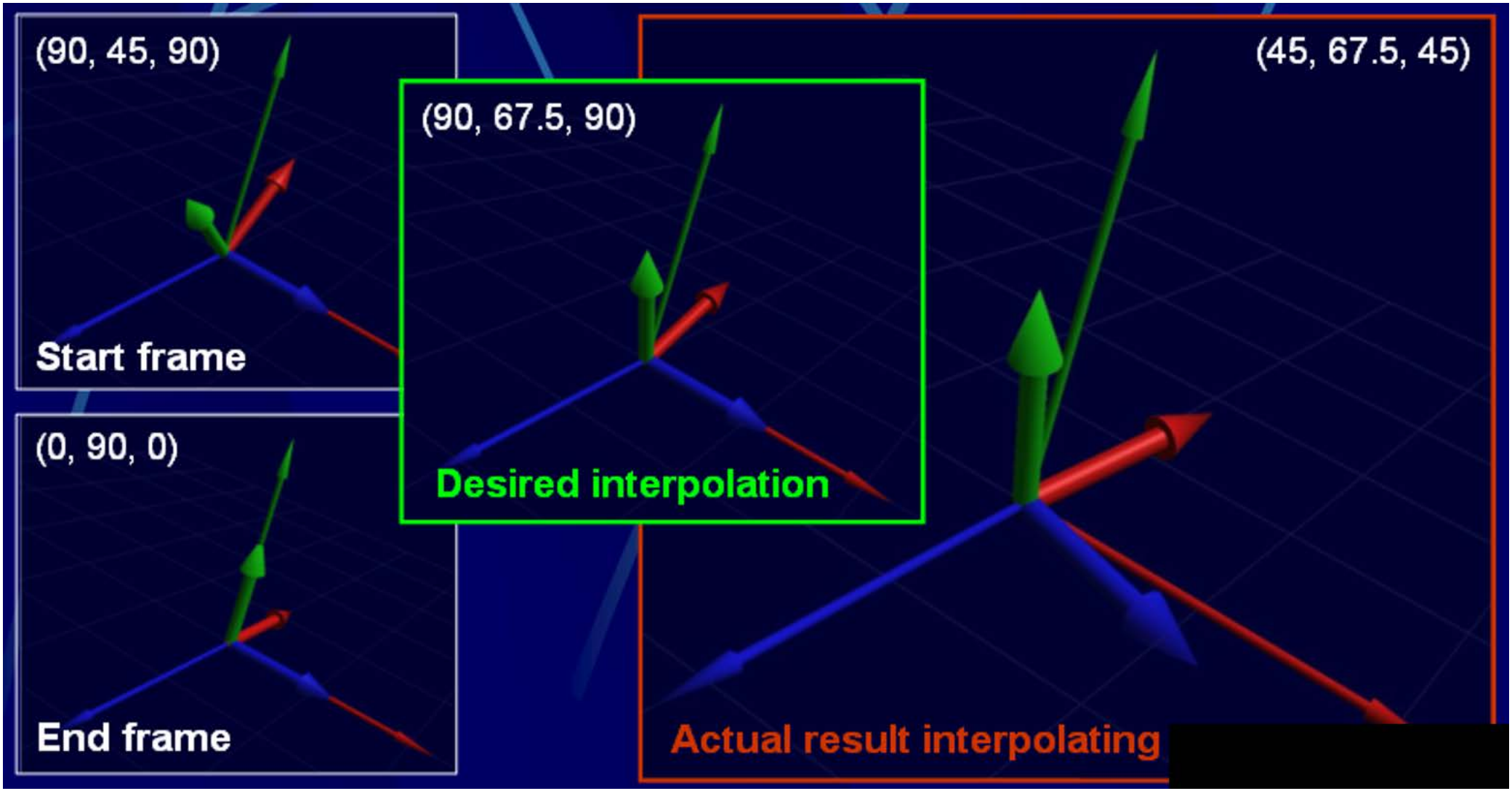
Initial Orientation
(object space)



(0,90,0)



(90, 45, 90)

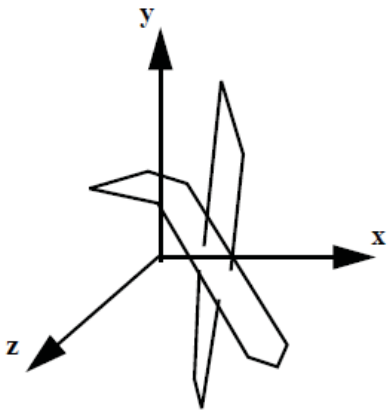

(90, 45, 90)
Start frame
(90, 67.5, 90)
Desired interpolation
(0, 90, 0)
End frame
(45, 67.5, 45)
Actual result interpolating

Orientation Representation Methods

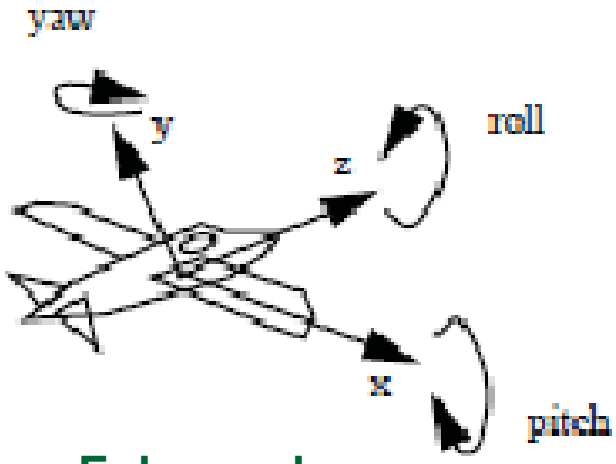
- Transformation matrices
- Fixed angle representation
- **Euler angle representation**
- ???
- ???

Euler Angle Representation

- **Euler Angle representation**: same as fixed angle, but axes of rotation move with the object
- roll, pitch, yaw of an aircraft



Fixed angle



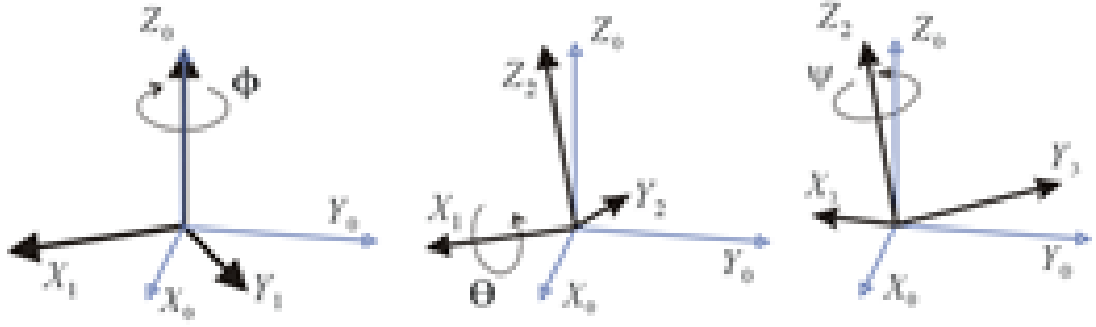
Euler angle

Euler Angles vs. Fixed Angles

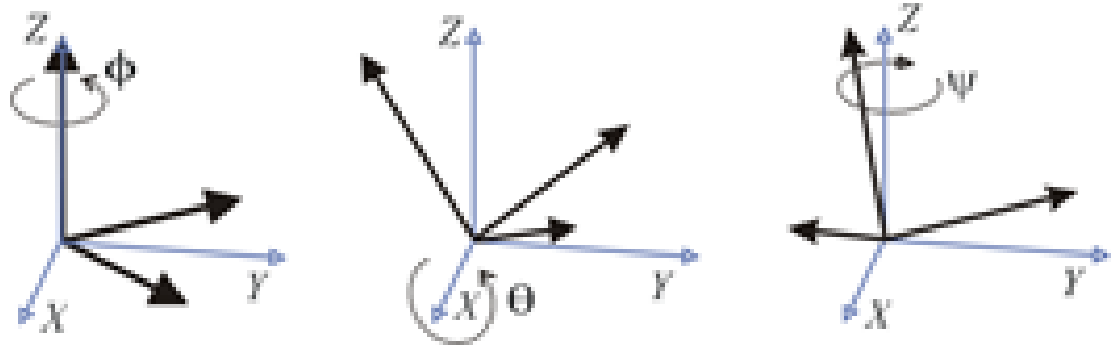
- Euler angle - rotates around local axes
- Fixed angle - rotates around world axes
- Rotations are reversed
  - x-y-z Euler angles == z-y-x fixed angles

Euler Angles vs. Fixed Angles

- *z-x-z* Euler angles: (-60,30,45)



- z-x-z fixed angles: (45,30,-60)



Euler's Theorem

- Euler's Theorem: Any two independent orthonormal coordinate frames can be related by a sequence of rotations (not more than three) about coordinate axes, where no two successive rotations may be about the same axis.
- Not to be confused with Euler angles, Euler integration, Newton-Euler dynamics, inviscid Euler equations, Euler characteristic…
- Leonard Euler (1707-1783)

Euler Angles

- This means that we can represent an orientation with 3 numbers
- A sequence of rotations around principle axes is called an *Euler Angle Sequence*
- Assuming we limit ourselves to 3 rotations without successive rotations about the same axis, we could use any of the following 12 sequences:

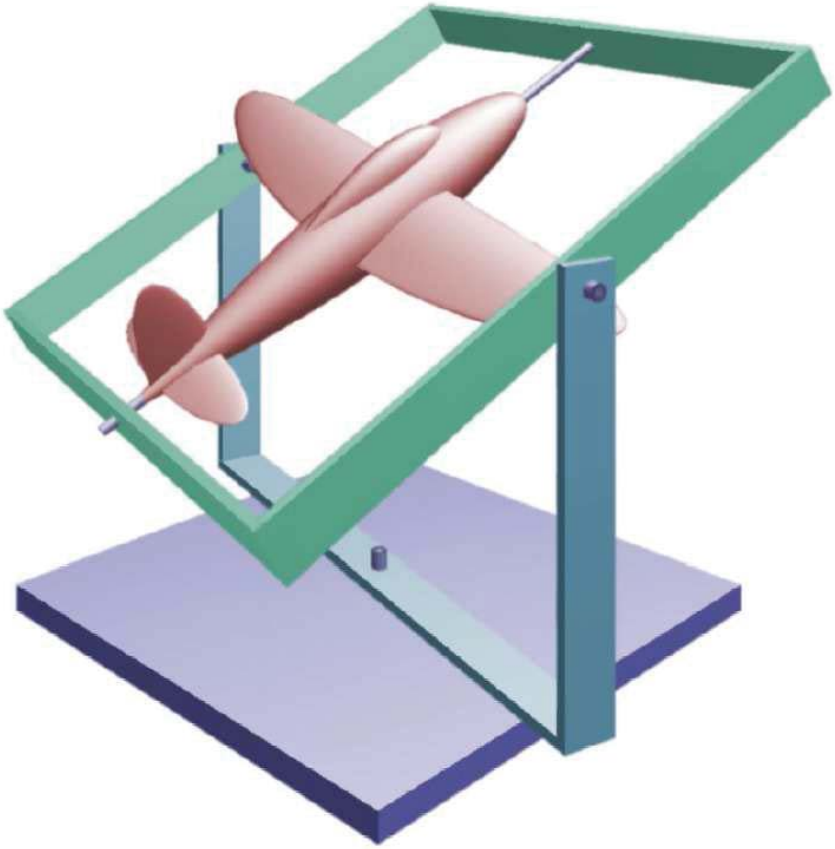
XYZ	XZY	XYX	XZX
YXZ	YZX	YXY	YZY
ZXY	ZYX	ZXZ	ZYZ

Euler Angles

- This gives us 12 redundant ways to store an orientation using Euler angles
- Different industries use different conventions for handling Euler angles (or no conventions)

Euler Angles

- An Euler angle is a rotation about a single axis.
- Any orientation can be described by composing three rotations, one around each coordinate axis.
- Roll, pitch and yaw (perfect for flight simulation)



http://www.fho-emden.de/~hoffmann/gimbal09082002.pd

Euler Angle Representation

- Euler Angle rotations about moving axes written in reverse order are the same as the fixed axis rotations.

$$R_{y'}(\beta)R_x(\alpha) = R_x(\alpha)R_y(\beta)R_x(\alpha)R_x(-\alpha) = R_x(\alpha)R_y(\beta)$$

$$R_{z''}(\gamma)R_{y'}(\beta)R_x(\alpha) = R_x(\alpha)R_y(\beta)R_z(\gamma)R_y(-\alpha)R_x(-\beta)R_y(\beta)R_x(\alpha) = R_x(\alpha)R_y(\beta)R_z(\gamma)$$

Euler **Fixed**

Euler Angle Issues

- No easy concatenation of rotations

  Our easy addition of angles goes out the window with Euler angles

- Still has interpolation problems
- Can lead to gimbal lock

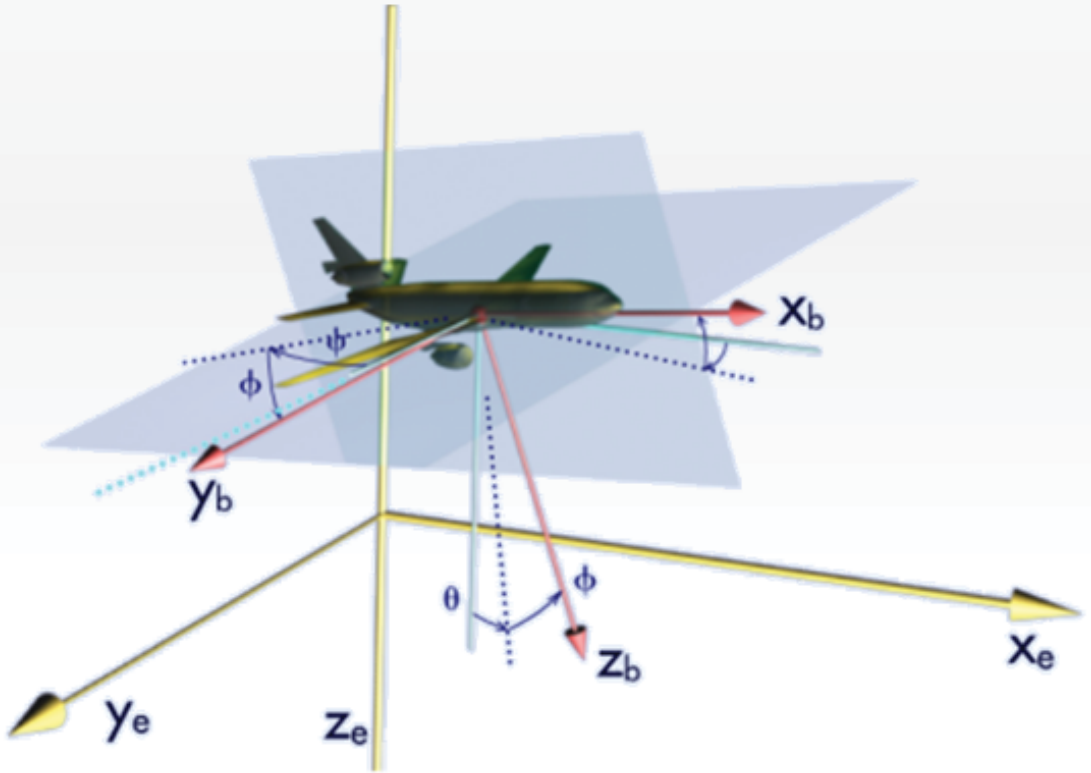
3D Orientation: Euler angles

Whilst Euler angles have been widely used in graphical applications, they present a number of problems. In particular, changing the order in which the individual axis rotations are applied results in a different outcome, i.e.:

$$r_1 r_2 \neq r_2 r_1$$

This introduces problems when combining two or more rotations.

Additionally, if the rotational axes are fixed (e.g. to the world x, y and z axes) then the issue of gimbal lock arises whereby a rotation results in one axis being aligned with another (thereby preventing any further rotation around that particular axis).



Euler Angle Concatenation

- Can't just add or multiply components
- Best way:
  - Convert to matrices
  - Multiply matrices
  - Extract euler angles from resulting matrix
- Not cheap

Euler Angles to Matrix Conversion

- To build a matrix from a set of Euler angles, we multiply a sequence of rotation matrices together:

$$\mathbf{R}_x \cdot \mathbf{R}_y \cdot \mathbf{R}_z = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_x & s_x \\ 0 & -s_x & c_x \end{bmatrix} \cdot \begin{bmatrix} c_y & 0 & -s_y \\ 0 & 1 & 0 \\ s_y & 0 & c_y \end{bmatrix} \cdot \begin{bmatrix} c_z & s_z & 0 \\ -s_z & c_z & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

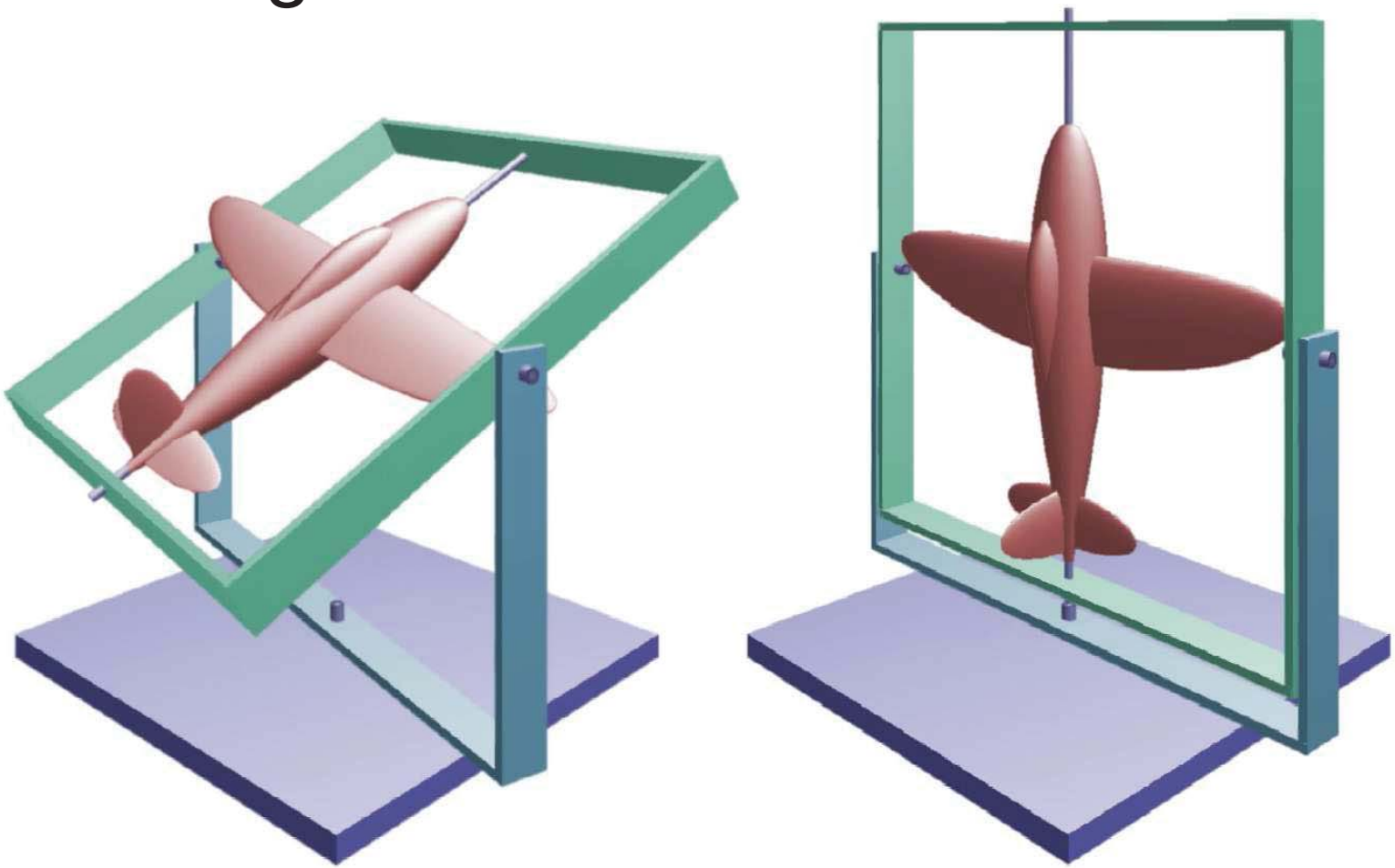
$$= \begin{bmatrix} c_y c_z & c_y s_z & -s_y \\ s_x s_y c_z - c_x s_z & s_x s_y s_z + c_x c_z & s_x c_y \\ c_x s_y c_z + s_x s_z & c_x s_y s_z - s_x c_z & c_x c_y \end{bmatrix}$$

Gimbal Lock

- One potential problem that they can suffer from is ‘gimbal lock’
- This results when two axes effectively line up, resulting in a temporary loss of a degree of freedom
- This is related to the singularities in longitude that you get at the north and south poles

Gimbal Lock

- Two or more axis align resulting in a loss of rotation degrees of freedom.



http://www.fho-emden.de/~hoffmann/gimbal09082002.pdf

GIMBAL LOCK EXPLAINED

Euler Angles in the Real World

- Apollo inertial measurement unit
- To “prevent” lock, they added a fourth Gimbal!

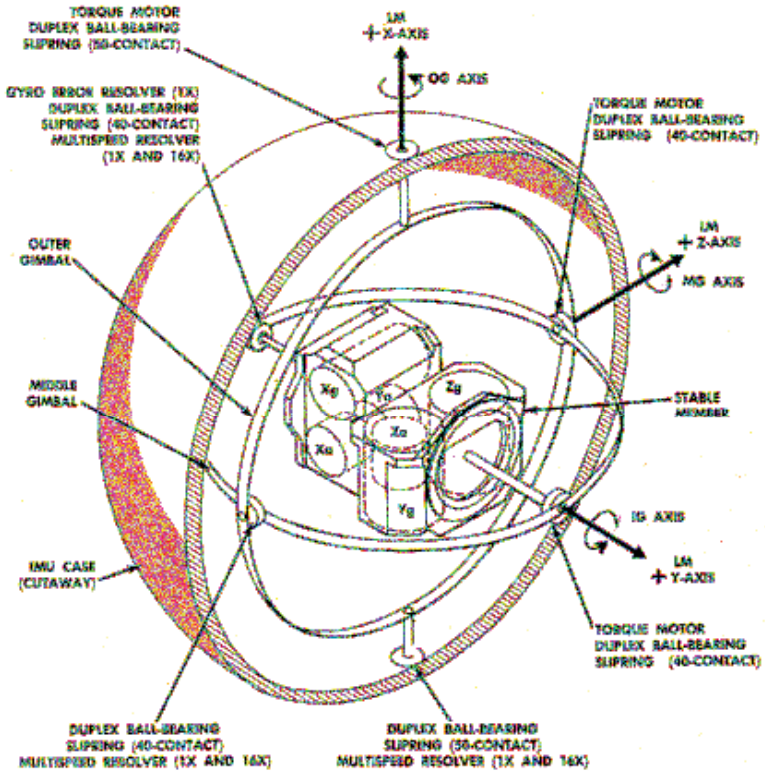
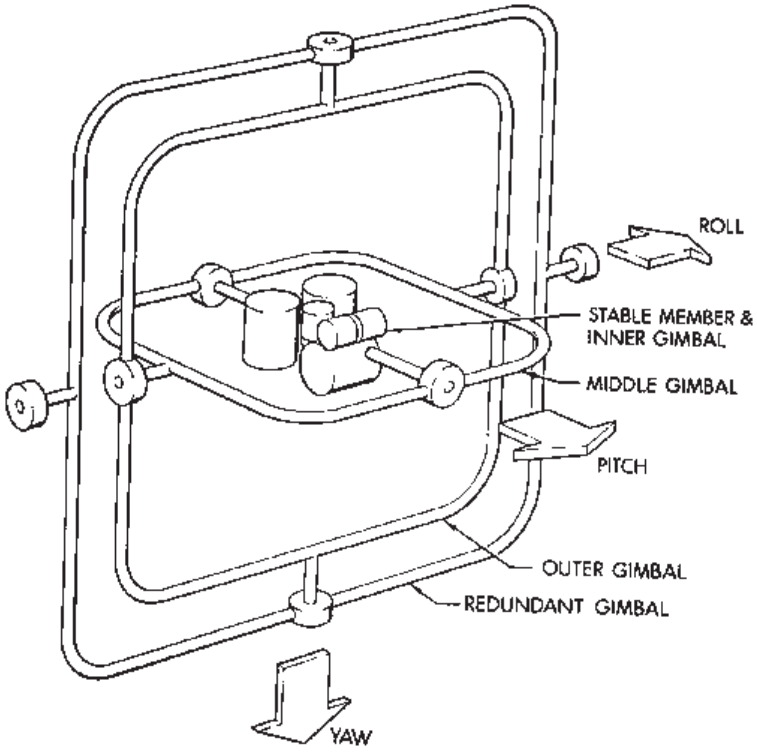


Figure 2.1-24. IMU Gimbal Assembly



http://www.hq.nasa.gov/office/pao/History/alsj/gimbals.html

Orientation Representation Methods

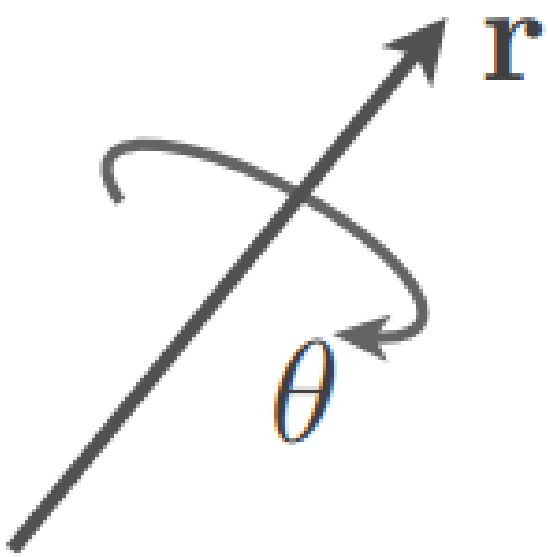
- Transformation matrices
- Fixed angle representation
- Euler angle representation
- **Axis-angle representation**
- ???

3D Orientation: Axis-angle Representation

Any rotation (and hence any combination of rotations) in 3D can be represented as a single rotation about a fixed axis (i.e. specified using an axis and an angle).

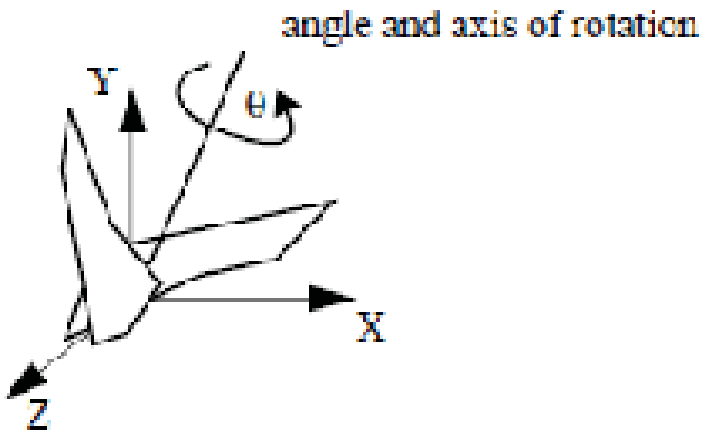
The scaled axis representation combines the axis and angle into a single vector, where the direction of the vector gives the axis and the magnitude of the vector gives the angle in the range [0, π] (a negative angle is equivalent to a positive rotation in the opposite direction).

This is a compact orientation representation but when representing rotations the mathematics involved in combining two rotations is not straightforward.



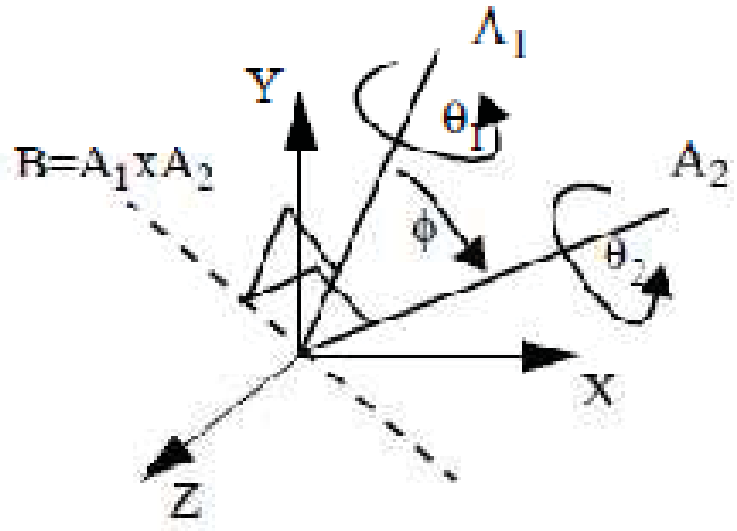
Axis Angle Representation

- Euler's Rotation Theorem:
- Any orientation can be represented by a 4-tuple
  - angle, vector(x,y,z)
- where the angle is the amount to rotate by and the vector is the axis to rotate about



Interpolating Axis Angle Representations

- Can interpolate the angle and axis separately
- No gimbal lock
- But, can't efficiently compose rotations…must convert to matrices first



$$B = A_1 \times A_2$$

$$\phi = \cos^{-1}\left(\frac{A_1 \bullet A_2}{|A_1||A_2|}\right)$$

$$A_k = R_B(k \cdot \phi) A_1$$

$$\theta_k = (1-k) \cdot \theta_1 + k \cdot \theta_2$$

Orientation Representation Methods

- Transformation matrices
- Fixed angle representation
- Euler angle representation
- Axis-angle representation
- **Quaternions**

Quaternion Representation

- All prior representations have drawbacks
- Quaternions:
  - Good for interpolation
  - Can be multiplied (composed)
  - No gimbal lock

Quaternions

- Quaternions are an interesting mathematical concept with a deep relationship with the foundations of algebra and number theory
- Invented by W.R.Hamilton in 1843
- In practice, they are most useful to us as a means of representing orientations
- A quaternion has 4 components

$$\mathbf{q} = \begin{bmatrix} q_0 & q_1 & q_2 & q_3 \end{bmatrix}$$

Quaternions (Imaginary Space)

- Quaternions are actually an extension to complex numbers
- Of the 4 components, one is a 'real' scalar number, and the other 3 form a vector in imaginary ijk space!

$$\mathbf{q} = q_0 + iq_1 + jq_2 + kq_3$$

$$i^2 = j^2 = k^2 = ijk = -1$$

$$i = jk = -kj$$

$$j = ki = -ik$$

$$k = ij = -ji$$

Quaternions (Scalar/Vector)

- Sometimes, they are written as the combination of a scalar value s and a vector value **v**

$$\mathbf{q} = \langle s, \mathbf{v} \rangle$$

where

$$s = q_0$$

$$\mathbf{v} = \begin{bmatrix} q_1 & q_2 & q_3 \end{bmatrix}$$

Unit Quaternions

$$|\mathbf{q}| = \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2} = 1$$

- These correspond to the set of vectors that form the ‘surface’ of a 4D hypersphere of radius 1
- The ‘surface’ is actually a 3D volume in 4D space, but it can sometimes be visualized as an extension to the concept of a 2D surface on a 3D sphere

Quaternions

- 4-tuple of real numbers
  - s,x,y,z or [s,v]
  - s is a scalar
  - v is a vector
- Same information as axis/angle but in a different form

$$q = Rot_{\theta,(x,y,z)} = [\cos(\theta/2), \sin(\theta/2) \cdot (x,y,z)]$$

Quaternions as Rotations

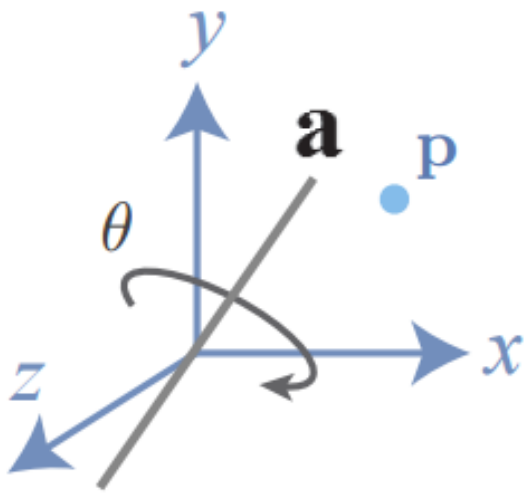
- A quaternion can represent a rotation by an angle θ around a unit axis **a**:

$$\mathbf{q} = \left[\cos\frac{\theta}{2}, \quad a_x \sin\frac{\theta}{2}, \quad a_y \sin\frac{\theta}{2}, \quad a_z \sin\frac{\theta}{2} \right]$$

or

$$\mathbf{q} = \left\langle \cos\frac{\theta}{2}, \mathbf{a}\sin\frac{\theta}{2} \right\rangle$$

- If **a** is unit length, then **q** will be also



$$\mathbf{q}_p = \begin{bmatrix} 0 \\ \mathbf{p} \end{bmatrix} \qquad \mathbf{q} = \begin{bmatrix} \cos(\theta/2) \\ \sin(\theta/2)\mathbf{a} \end{bmatrix}$$

Quaternions as Rotations

- If **a** is unit length, then **q** will be also

$$|\mathbf{q}| = \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2}$$

$$= \sqrt{\cos^2\frac{\theta}{2} + a_x^2\sin^2\frac{\theta}{2} + a_y^2\sin^2\frac{\theta}{2} + a_z^2\sin^2\frac{\theta}{2}}$$

$$= \sqrt{\cos^2\frac{\theta}{2} + \sin^2\frac{\theta}{2}\left(a_x^2 + a_y^2 + a_z^2\right)}$$

$$= \sqrt{\cos^2\frac{\theta}{2} + \sin^2\frac{\theta}{2}|\mathbf{a}|^2} = \sqrt{\cos^2\frac{\theta}{2} + \sin^2\frac{\theta}{2}}$$

$$= \sqrt{1} = 1$$

Quaternion to Matrix

- To convert a quaternion to a rotation matrix:

$$\begin{bmatrix} 1-2q_2^2-2q_3^2 & 2q_1q_2+2q_0q_3 & 2q_1q_3-2q_0q_2 \\ 2q_1q_2-2q_0q_3 & 1-2q_1^2-2q_3^2 & 2q_2q_3+2q_0q_1 \\ 2q_1q_3+2q_0q_2 & 2q_2q_3-2q_0q_1 & 1-2q_1^2-2q_2^2 \end{bmatrix}$$

Quaternion Math

- Addition
  - $[s_1, v_1] + [s_2, v_2] = [s_1 + s_2, v_1 + v_2]$
- Multiplication
  - $[s_1, v_1] * [s_2, v_2] = [\ s_1 s_2 - v_1 v_2\ ,\ s_1 v_2 + s_2 v_1 + v_1 \times v_2\ ]$
- Multiplication is not commutative but is associative (just like transformation matrices, as expected)
  - $q_1 q_2 \neq q_2 q_1$
  - $(q_1 q_2) q_3 = q_1 (q_2 q_3)$

Quaternion Math

- A point in space is represented as: $[0, v_1]$
- Multiplicative identity $[1,(0,0,0)]$
- Inverse of a quaternion

$$q^{-1} = (1/\|q\|)^2 \cdot [s,-v]$$

where

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

$$\mathbf{q}\mathbf{q}^{-1} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$q = a + bi + cj + dk$$

Then the conjugate is

$$q^* = a - bi - cj - dk$$

and the inverse is

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

Rotating Vectors Using Quaternions

- To rotate a vector v using quaternions
  - represent the vector as [0,v]
  - represent the rotation as a quaternion, *q*

$$v' = Rot(v) = q \cdot v \cdot q^{-1}$$

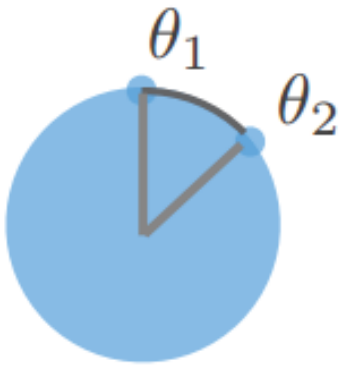
$$q = Rot_{\theta,(x,y,z)} = [\cos(\theta/2), \sin(\theta/2) \cdot (x,y,z)]$$

- Easy to compose quaternions as well

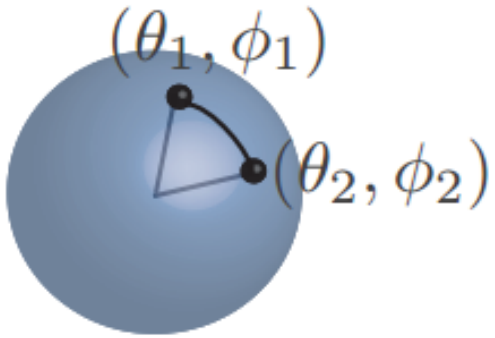
$$
\begin{aligned}
\mathbf{q}\mathbf{q}_p\mathbf{q}^{-1} &= \begin{bmatrix} w \\ \mathbf{v} \end{bmatrix} \begin{bmatrix} 0 \\ \mathbf{p} \end{bmatrix} \begin{bmatrix} w \\ -\mathbf{v} \end{bmatrix} \\
&= \begin{bmatrix} w \\ \mathbf{v} \end{bmatrix} \begin{bmatrix} \mathbf{p} \cdot \mathbf{v} \\ w\mathbf{p} - \mathbf{p} \times \mathbf{v} \end{bmatrix} \\
&= \begin{bmatrix} w\mathbf{p} \cdot \mathbf{v} - \mathbf{v} \cdot w\mathbf{p} + \mathbf{v} \cdot \mathbf{p} \times \mathbf{v} \ = 0 \\ w(w\mathbf{p} - \mathbf{p} \times \mathbf{v}) + (\mathbf{p} \cdot \mathbf{v})\mathbf{v} + \mathbf{v} \times (w\mathbf{p} - \mathbf{p} \times \mathbf{v}) \end{bmatrix}
\end{aligned}
$$

Quaternion Interpolation

- We can think of rotations as lying on an n-D unit sphere
- Interpolating rotations means moving on n-D sphere
  - Can encode position on sphere by unit vector



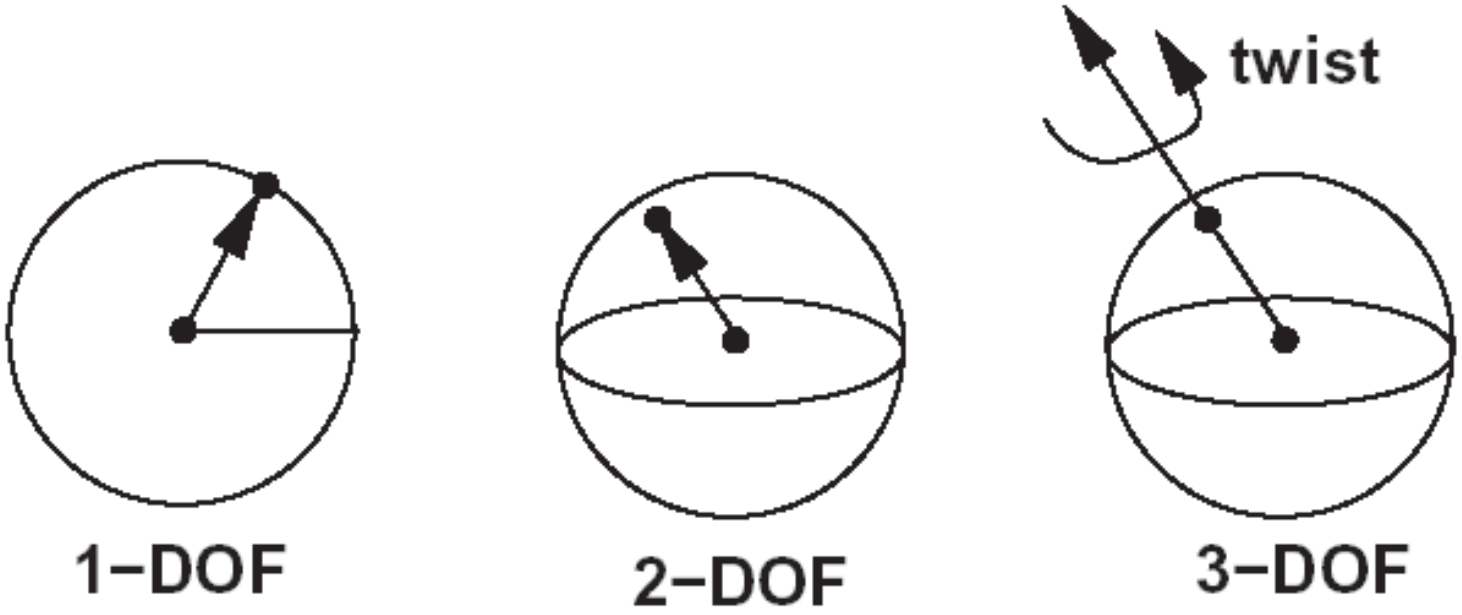
1-angle rotation can be represented by a unit circle



2-angle rotation can be represented by a unit sphere

Solution: Quaternion Interpolation

- Interpolate orientation on the unit sphere in 4D
  - Logical and easy, isn't it?
- By analogy:
  1-, 2-, 3-DOF rotations as constrained points on 1, 2, 3-spheres in 2D, 3D and 4D



$q = s + xi + yj + zk$ where i, j, and k are imaginary numbers

Rotation	Quaternions
Identity (no rotation)	1, -1
180 degrees about x-axis	i, -i
180 degrees about y-axis	j, -j
180 degrees about z-axis	k, -k
angle θ, axis (unit vector)	$\pm[\cos(\theta/2) + \vec{n}\sin(\theta/2)]$

Quaternion Interpolation

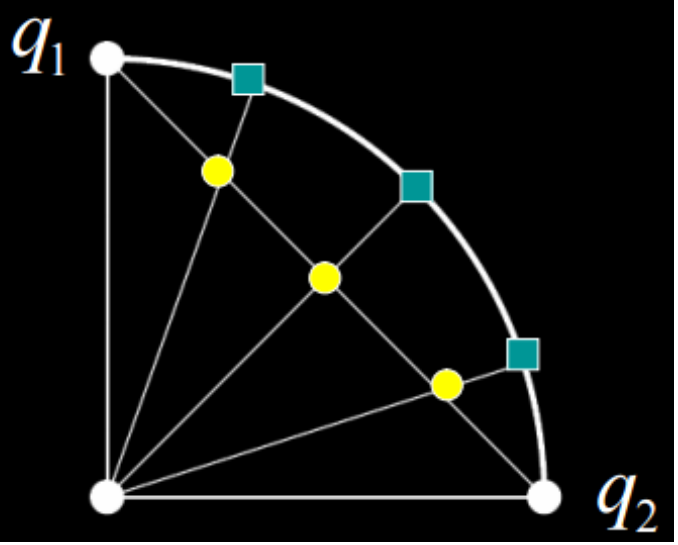
- Interpolating quaternions produces better results than Euler angles
- A quaternion is a point on the 4-D unit sphere
  - interpolating rotations requires a unit quaternion at each step - another point on the 4-D sphere
  - move with constant angular velocity along the great circle between the two points
  - **S**pherical **L**inear int**ERP**olation (**SLERP**ing)
- Any rotation is given by 2 quaternions, so pick the shortest SLERP

Spherical Linear Interpolation (slerp)

- Want equal increment along arc connecting two quaternions on the spherical surface

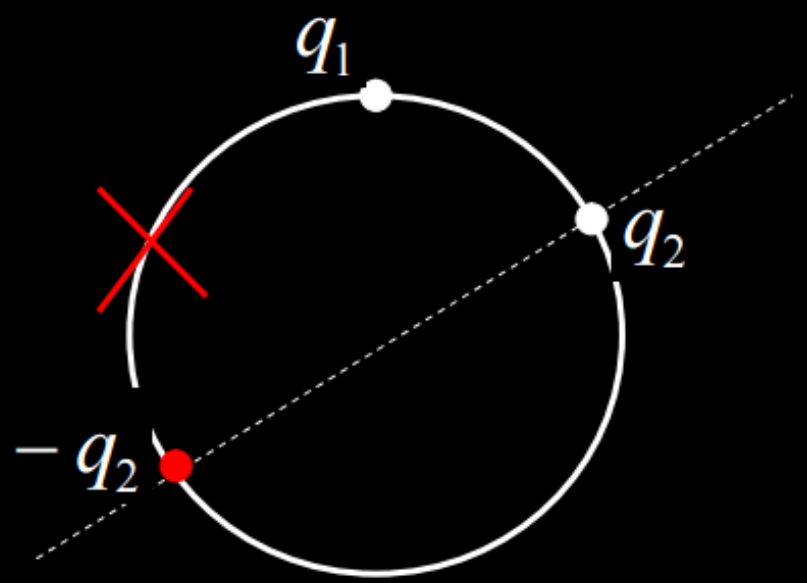
$$slerp(q_1, q_2, u) = \frac{\sin(1-u)\theta}{\sin\theta} q_1 + \frac{\sin u\theta}{\sin\theta} q_2$$

- Normalize to regain unit quaternion



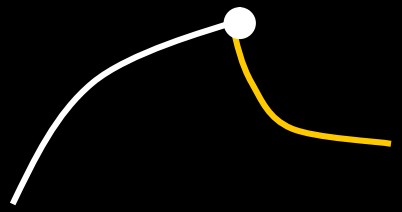
Spherical Linear Interpolation (slerp)

- Recall that q and $-q$ represent same rotation
- Slerp can go the LONG way!
- Have to go the short way $\quad q_1 \cdot q_2 > 0$



What if there are multiple segments?

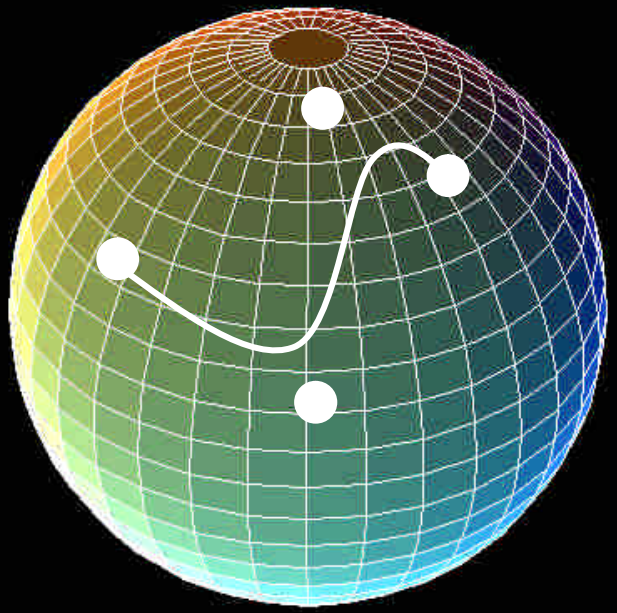
- As linear interpolation in Euclidean space, we can have first order discontinuity



- Need a cubic curve interpolation to maintain first order continuity

Bezier Interpolation on 4D Sphere

- Have to perform interpolation on 4D sphere
- Construct Bezier curve by iteratively applying slerp

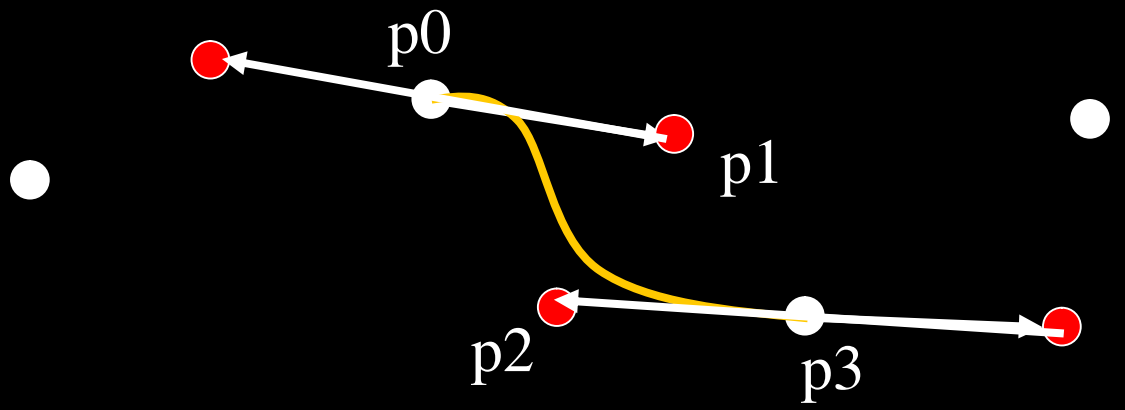


Bezier Interpolation in Euclidean Space

$$P(u)=[u3\ u2\ u1]\begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 3 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}\begin{pmatrix} p0 \\ p1 \\ p2 \\ p3 \end{pmatrix}$$

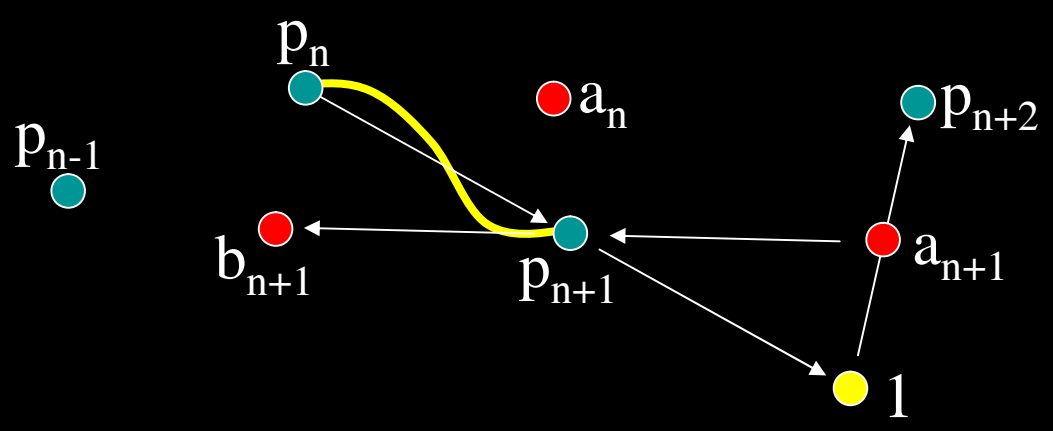
Colinearity of the control points at either side of an endpoint guarantees the 1st order continuity

$$P'(0) = 3(p1-p0),\ P'(1)=3(p3-p2)$$



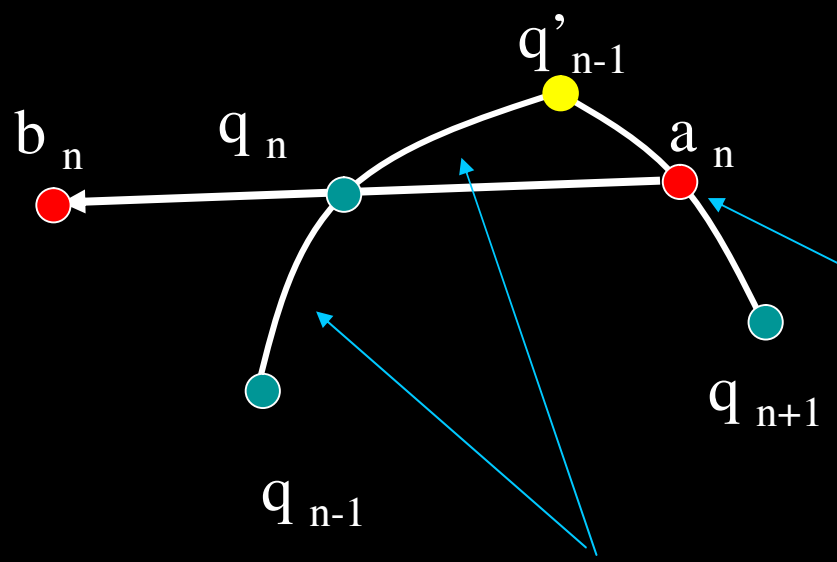
Bezier Interpolation in Euclidean Space

- Automatically generate control points



Bezier Interpolation on 4D sphere

- Automatically generating interior (spherical) control point

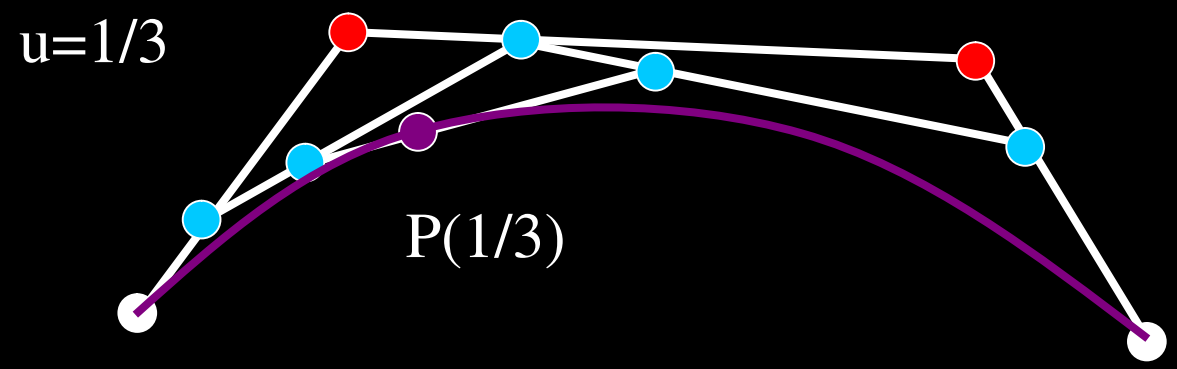


$$Bisect(q'_{n-1}, q_{n+1}) = \frac{q'_{n-1} + q_{n+1}}{\left\| q'_{n-1} + q_{n+1} \right\|}$$

$$double(q_{n-1}, q_n) = 2(q_{n-1} \cdot q_n)q_n - q_{n-1}$$

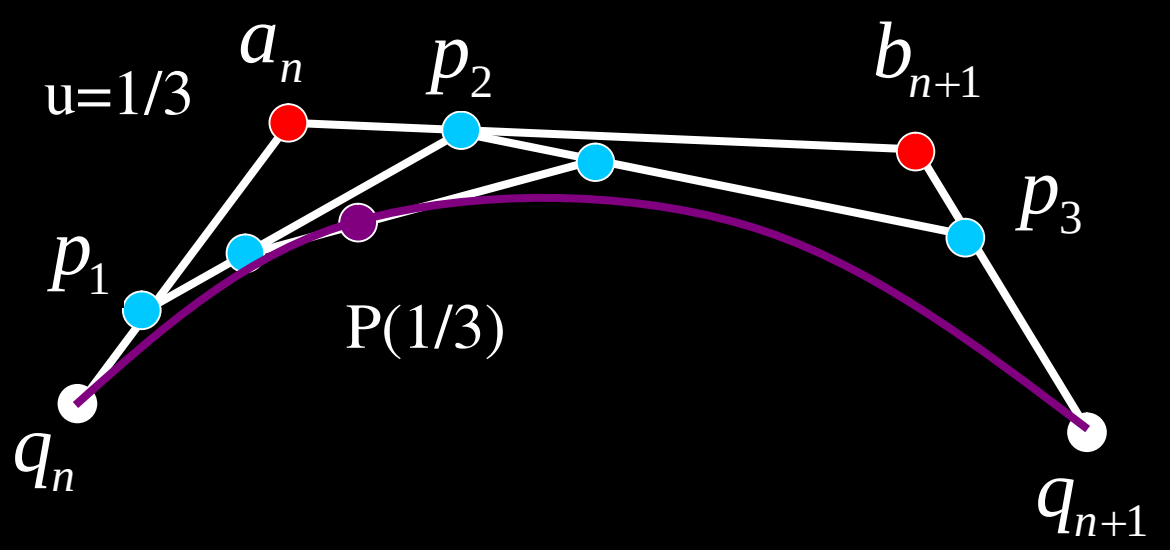
De Casteljau Construction of Bezier Curve

- Constructing Bezier curve by multiple linear interpolation



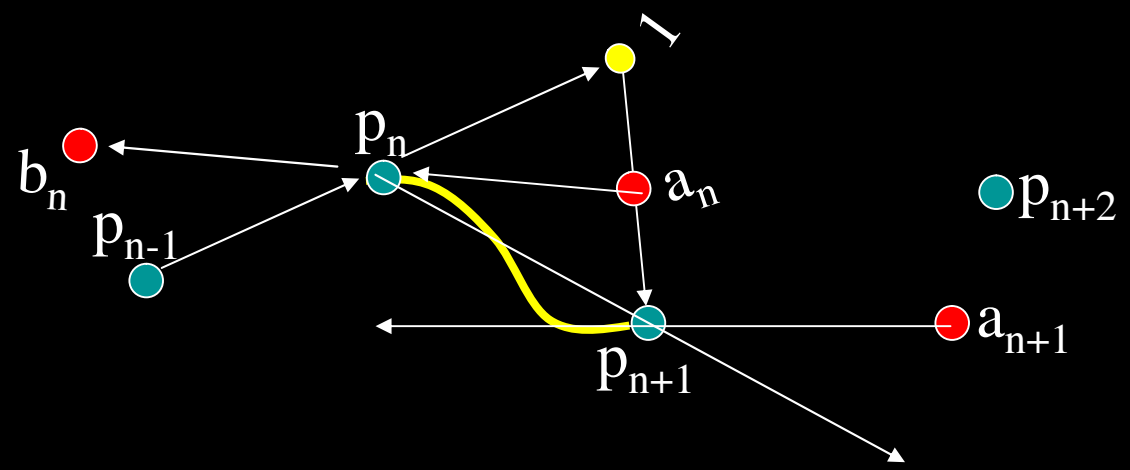
De Casteljau Construction on 4D Sphere

$$p_1 = slerp(q_n, a_n, \tfrac{1}{3})$$
$$p_2 = slerp(a_n, b_{n+1}, \tfrac{1}{3})$$
$$p_3 = slerp(b_{n+1}, q_{n+1}, \tfrac{1}{3})$$
$$p_{12} = slerp(p_1, p_2, \tfrac{1}{3})$$
$$p_{23} = slerp(p_2, p_3, \tfrac{1}{3})$$
$$p = slerp(p_{12}, p_{23}, \tfrac{1}{3})$$



Bezier Interpolation in Euclidean Space

- Automatically generate control points



Best Rotation Representation

We can convert to/from any of these representations

- but the mapping is not one-to-one

Choose the best representation for the task

- Input – ???
- Interpolation – ???
- Composing rotations – ???
- Drawing – ???

Best Rotation Representation

We can convert to/from any of these representations

- but the mapping is not one-to-one

Choose the best representation for the task

- Input – Euler, axis-angle(?)
- Interpolation – quaternions
- Composing rotations – quaternions, orientation matrix
- Drawing – orientation matrix

Summary

- Linear transformations represented by 4x4 matrices are a fundamental operation in computer graphics & animation.
- There are several orientation representations to choose from.
- Matrix, fixed axis, Euler, axis-angle are intuitive, simple to implement, and often sufficient. But they each have disadvantages.
- Quaternions are more robust, but somewhat harder to manipulate.