

Affine Transformation

Affine/Linear Transformation

Most used deformations $p' = f(p)$

can be represented by a matrix $\rightarrow$ can be sent as uniform parameter to the shader.

- In standard 3D coordinates $p = (x, y, z)$

$$p' = L p + t$$

L : linear component (3×3 matrix)

t : translation

- In homogeneous coordinates $p = (x, y, z, 1)$

$$p' = A p$$

A : 4×4 matrix

$$A = \left(\begin{array}{ccc|c} L & t \\ \hline 0 & 1 \end{array} \right) = \left(\begin{array}{ccc|c} L_{xx} & L_{xy} & L_{xz} & t_x \\ L_{yx} & L_{yy} & L_{yz} & t_y \\ L_{zx} & L_{zy} & L_{zz} & t_z \\ \hline 0 & 0 & 0 & 1 \end{array} \right)$$

Properties:

- $\det(L)$: Change of volume when applying L to a shape

- $L^T L = 1 \Rightarrow$ Isometry

Affine/Linear Deformations

Translation

$$t(p) = (x + t_x, y + t_y, z + t_z)$$

$$T = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Scaling

$$s(p) = (s_x x, s_y y, s_z z)$$

$$S = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation

Several possible representations
(see later)

Note: Isometry

Shearing

$$sh_{xy}(p) = (x + \lambda y, y, z)$$

$$sh_{xz}(p) = (x + \lambda z, y, z)$$

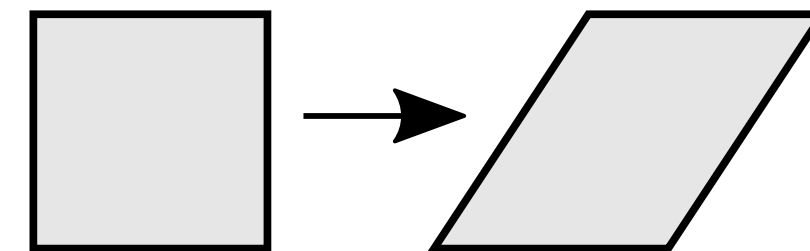
$$sh_{yx}(p) = (x, y + \lambda x, z)$$

...

$$Sh_{xy} = \begin{pmatrix} 1 & \lambda & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad Sh_{xz} = \begin{pmatrix} 1 & 0 & \lambda & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad Sh_{yx} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ \lambda & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \dots$$

Usually avoided in Graphics

Note: $\det(Sh) = 1 \rightarrow$ constant volume (but no isometry)



Rotations

3D rotations have 3 dof, No unique representation

Matrix

$$R = \begin{pmatrix} R_{xx} & R_{xy} & R_{xz} \\ R_{yx} & R_{yy} & R_{yz} \\ R_{zx} & R_{zy} & R_{zz} \end{pmatrix}$$

$$R^T R = I$$

$$\det(R) = 1$$

(+) Computationally convenient

(-) Non-explicit dof, redundancies

Euler Angles

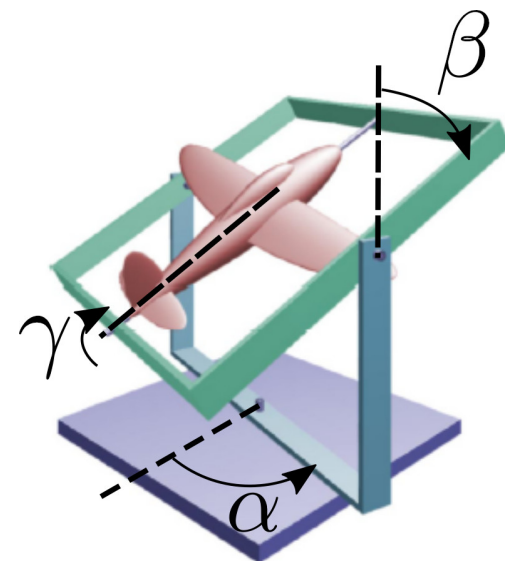
3 angles: (α, β, γ)

Composition of rotation around basic axes

Not unique (x-y-z, y-z-x, x-y-x', x-z-x', ...)

(+) Meaningfull parameters

(-) Gimbal-lock

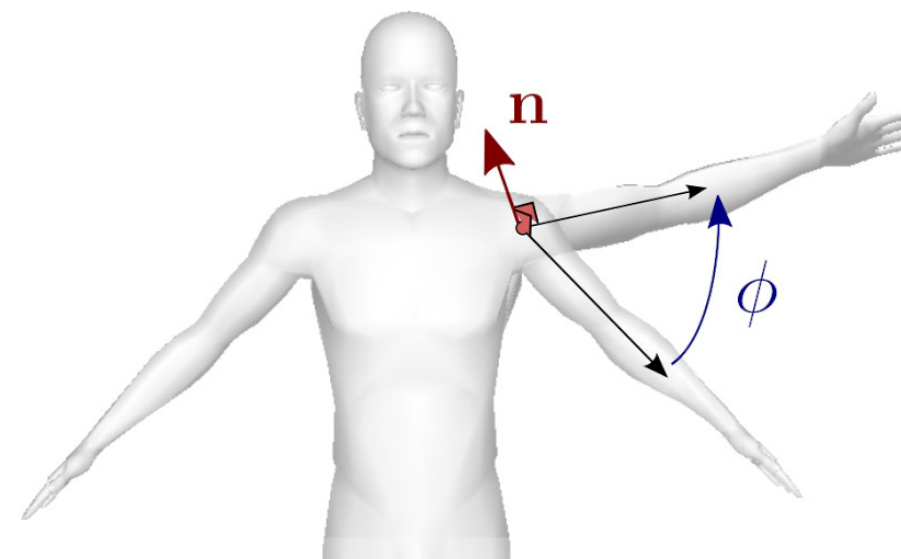


Axis angle

$(\mathbf{n}, \theta)$

(+) Meaningfull parameters

(-) No direct composition



Quaternion

$$q = (x, y, z, w) \\ = \left(\mathbf{n} \sin \left(\frac{\theta}{2} \right), \cos \left(\frac{\theta}{2} \right) \right)$$

(+) Composition and interpolation

(-) Less intuitive components

Cautions with transformations order

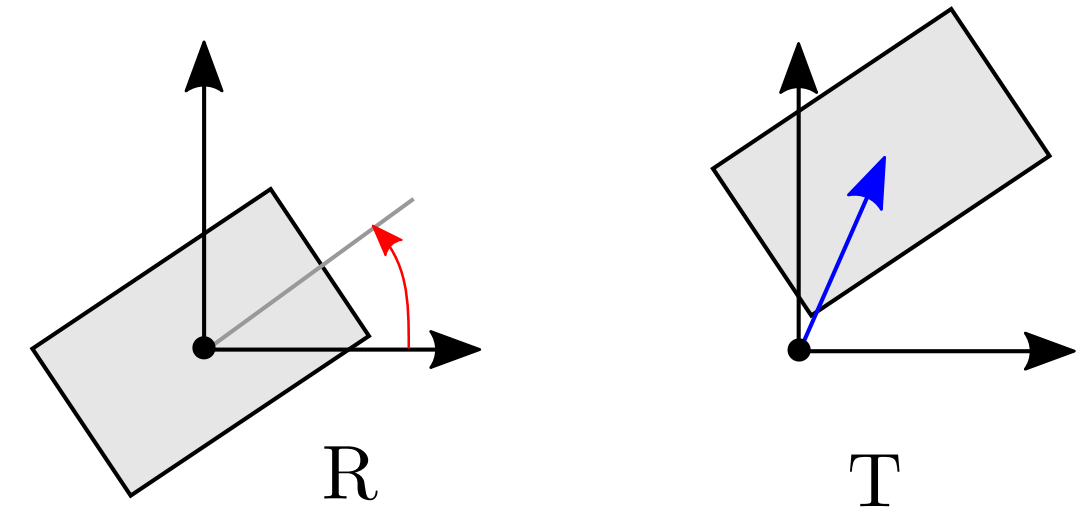
Take care, order of operation does matter !

Rotation r , Translation t : $r \circ t \neq t \circ r \Rightarrow M_1 = T R \neq R T = M_2$

Take care (2): transformation matrices applied to coordinates from right to left.

$$M_1 = TR = \left(\begin{array}{c|c} 1 & t \\ \hline 0 & 1 \end{array} \right) \left(\begin{array}{c|c} R & 0 \\ \hline 0 & 1 \end{array} \right) = \left(\begin{array}{c|c} R & t \\ \hline 0 & 1 \end{array} \right)$$

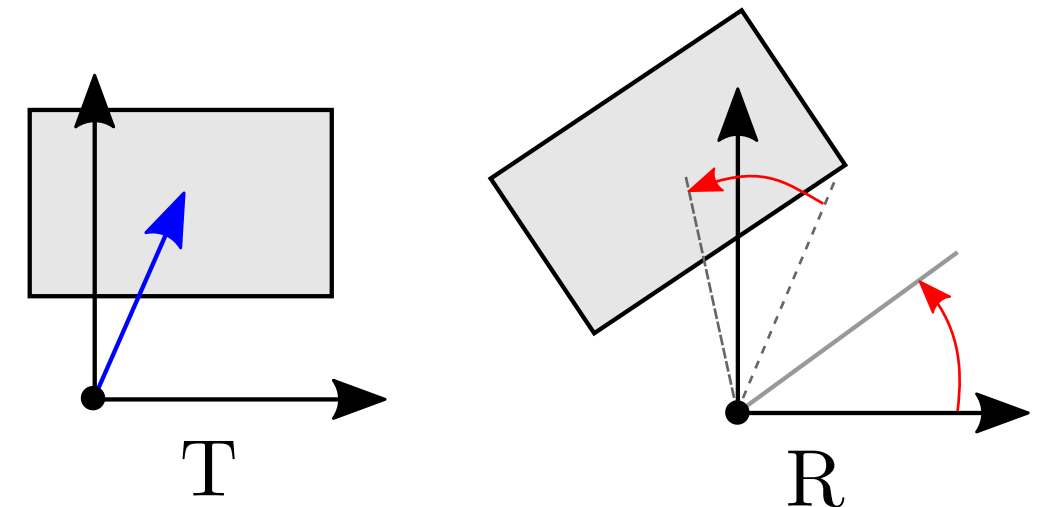
First rotates, then translates



$$M_2 = RT = \left(\begin{array}{c|c} R & 0 \\ \hline 0 & 1 \end{array} \right) \left(\begin{array}{c|c} 1 & t \\ \hline 0 & 1 \end{array} \right) = \left(\begin{array}{c|c} R & Rt \\ \hline 0 & 1 \end{array} \right)$$

First translates, then rotates

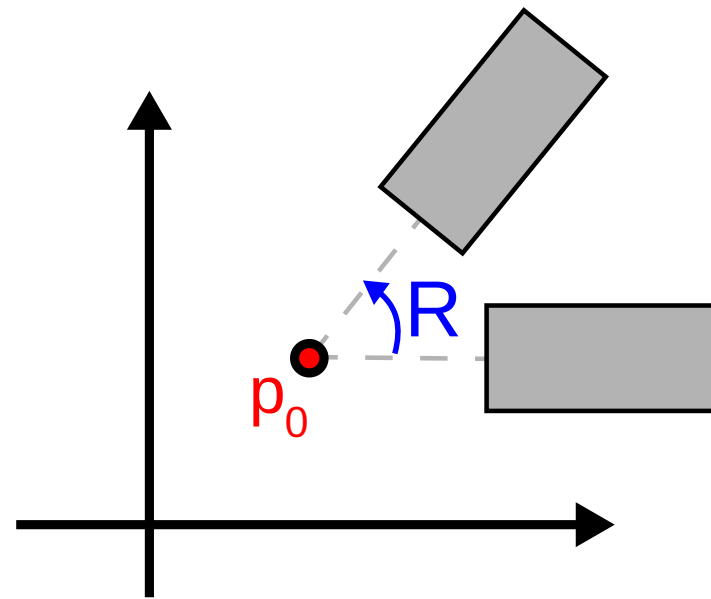
Rotation always happens around the origin.



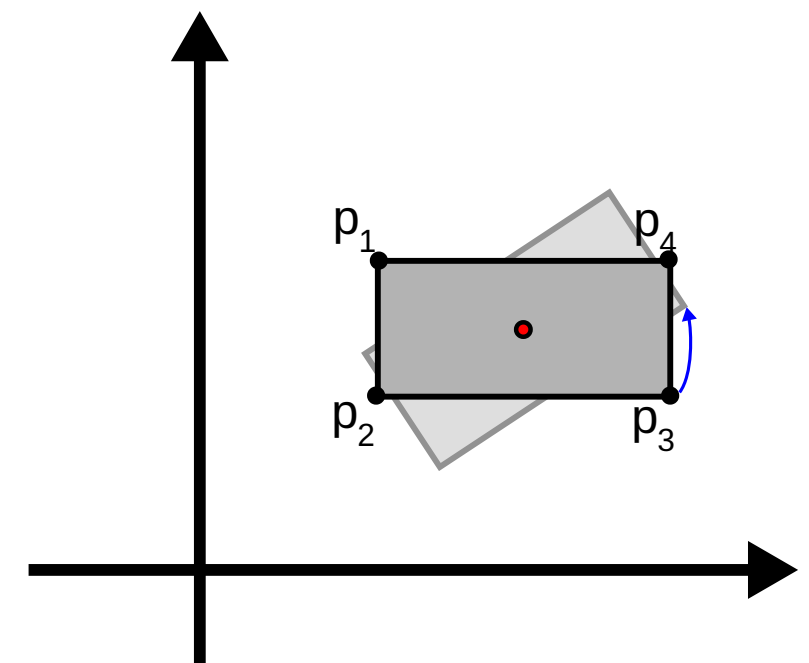
Affine Transformation Exercises

Q. Express the affine transform (as a 4×4 block matrix) corresponding to a rotation R applied around an arbitrary position p_0 in space.

> $M = \dots$



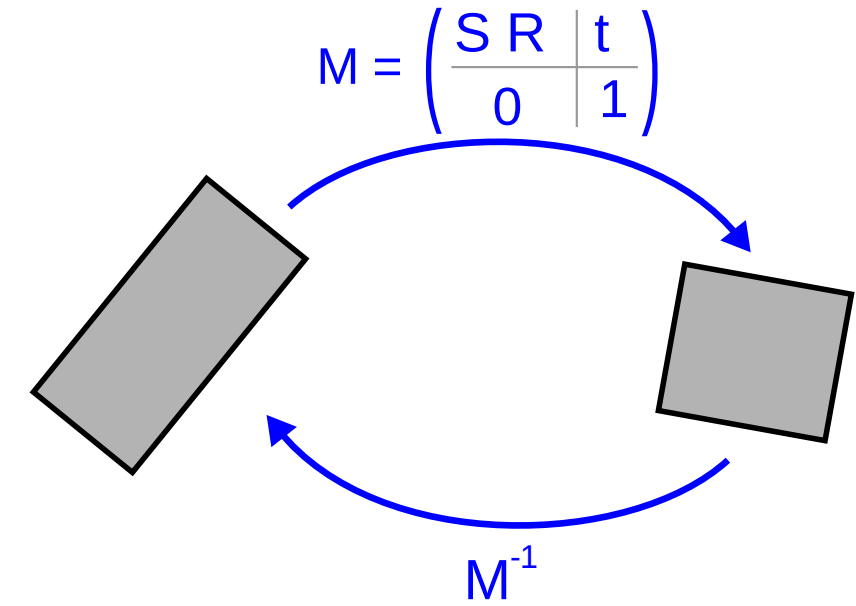
Q. Consider a shape defined by a triangular mesh with vertex positions $(p_i)_{i \in [1, N]}$.
> Express the affine transform allowing to rotate the shape around its barycenter.



Affine Transformation Exercises

Q. Consider the affine transform M parameterized by a scaling s , a rotation R , and a translation t .

> Express the inverse matrix M^{-1} with respect to s , R and t



Interpolating rotation

Do not use componentwise-interpolation on rotation matrix
 $\Rightarrow$ interpolate in **quaternion** space

Can use either:

- **SLERP** - Spherical Linear Interpolation

$$q(t) = \frac{\sin((1-t)\Omega)}{\sin(\Omega)} q_1 + \frac{\sin(t\Omega)}{\sin(\Omega)} q_2, \quad \text{with } \cos(\Omega) = q_1 \cdot q_2$$

Between two unit quaternions q_1, q_2

- **LERP** - Linear Interpolation

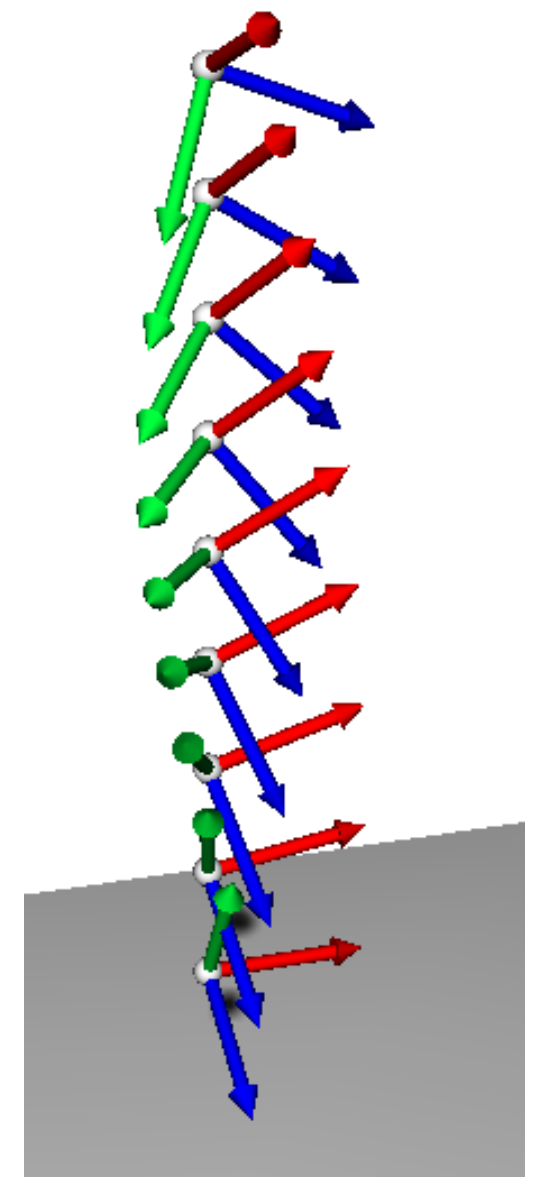
$$q(t) = \frac{\sum_j \alpha_j q_j}{\|\sum_j \alpha_j q_j\|}$$

When blending multiple quaternions q_j with weights α_j

Rem.

When interpolating b/w rotations and positions:

Use quaternion with rotation, componentwise-interpolation on position



Handling affine transformation : Polar Decomposition

Key-pose transforms are often given as 4×4 matrices.

How can we interpolate b/w affine transforms?

- Splitting linear M /translation part is easy.
- *Problem*: interpolating the linear part M mixes rotation, scaling, shearing

⇒ Split M into: rotation part, and scaling/shearing.

- Interpolate rotation with quaternion (ex. SLERP/LERP)
- Scaling/shearing using componentwise interpolation (ex. linear).

- Polar decomposition: $M = R D$

- R : Rotation matrix
- D : Positive semi-definite matrix

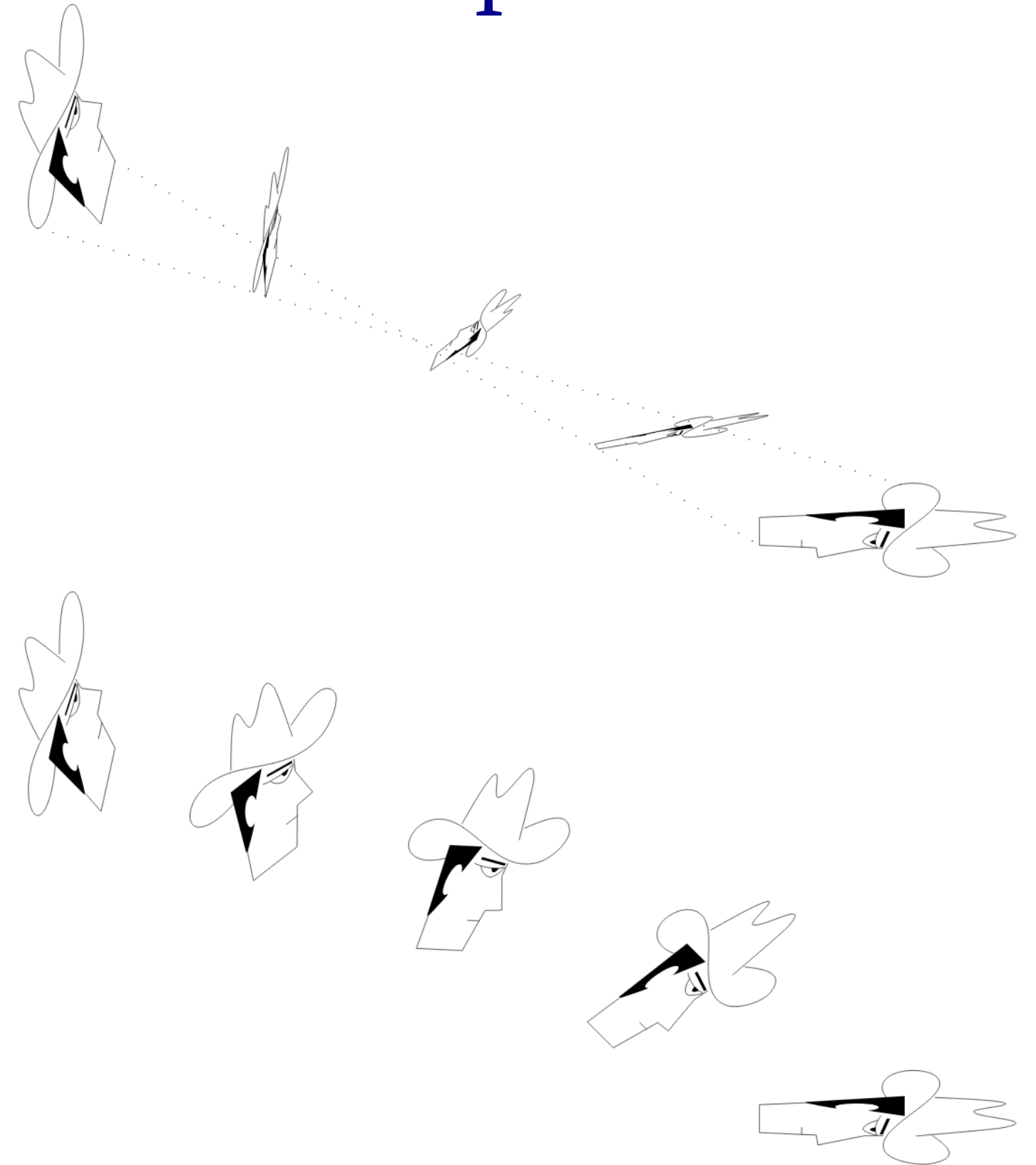
[K. Shoemake and T. Duff, Matrix Animation and Polar Decomposition. Graphics Interface 92.]

- Polar decomposition is obtained from SVD

$$\text{SVD}(M) = W \Sigma V^T \text{ with } R = W V^T, \quad D = V \Sigma V^T$$

- Or numerically, R can be computed using the following iterative scheme

$$R_0 = M, \quad R_{i+1} = 0.5 (R_i + (R_i^{-1})^T)$$



Handling affine transformation

Algorithm to interpolate between two general 4×4 matrices M_1, M_2

1. Extract translation p_1, p_2 from M_1, M_2 .
2. Compute R_1, R_2 (3×3 rotation matrices), and D_1, D_2 (3×3 scaling/shearing matrices) from M_1, M_2
3. Interpolate linearly position and scaling/shearing $p(t) = (1 - t) p_1 + t p_2, D(t) = (1 - t) D_1 + t D_2$
4. Compute quaternions q_1, q_2 from R_1, R_2

Note $M \rightarrow q$ with $q = \left(\frac{M_{zy} - M_{yz}}{2r}, \frac{M_{xz} - M_{zx}}{2r}, \frac{M_{yx} - M_{xy}}{2r}, \frac{r}{2} \right), r = \sqrt{1 + M_{xx} + M_{yy} + M_{zz}}$

5. Compute $q(t) = \text{SLERP}(q_1, q_2, t)$
6. Convert back to matrix $q(t) \rightarrow R(t)$
7. Compute final matrix $M(t) = R(t) D(t)$ with translation $p(t)$.

Camera and OpenGL

In OpenGL: no real notion of "camera"

Unique viewpoint - cube $(x, y, z) \in [-1, 1]^3$ - Normalized Device Coordinate

Left-right/x, Bottom-up/y, Front-back/z (toward negative z)

No perspective (displays (x,y) for the smallest z)

Any camera effect (position, orientation, perspective)
must be coded "manually" in the shader.

All these effects can be represented as matrices.

Three common matrices

Model matrix

Affine transform - specific to a shape.

Local object coordinates to world space.

View matrix

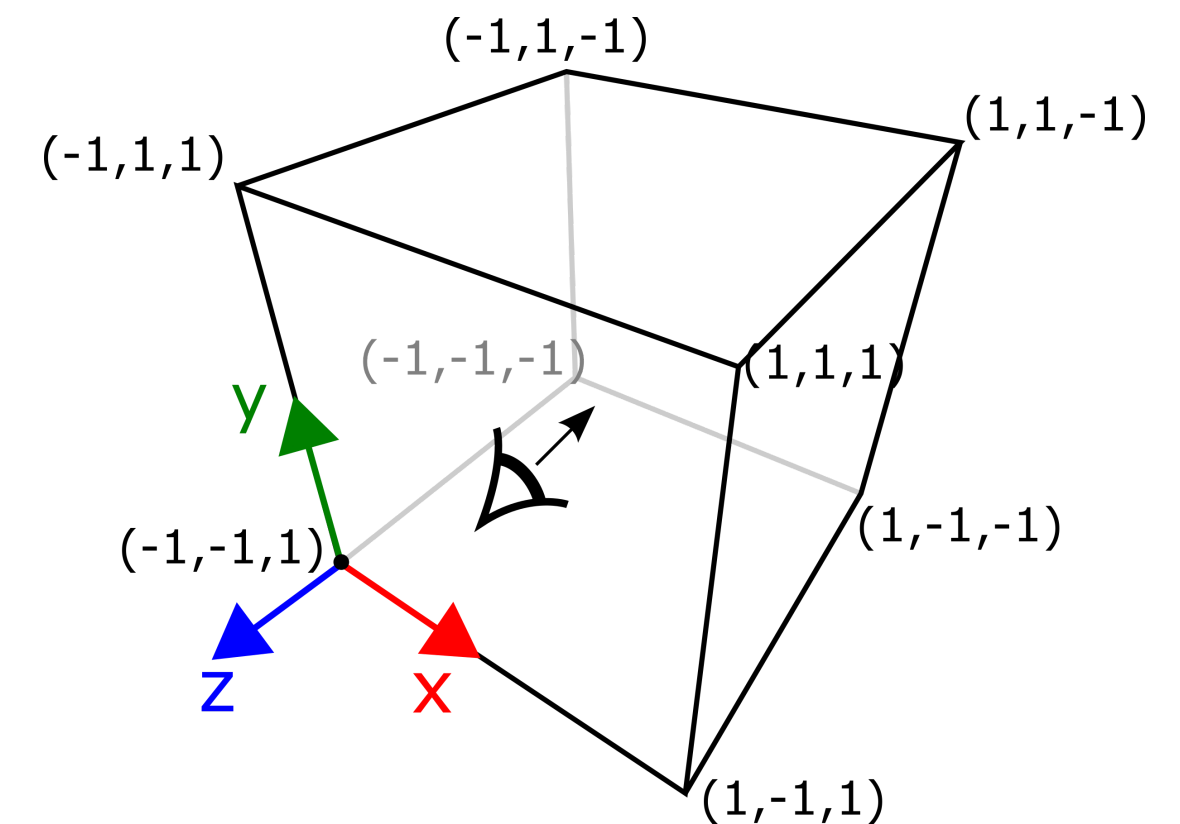
Rigid transform - position/orientation of the camera in the scene.

World space to camera/view/eye space

Projection matrix

Perspective matrix - map frustum to normalized cube. (/or ortho projection)

Camera space to Normalized Device Coordinate (clip/screen space)

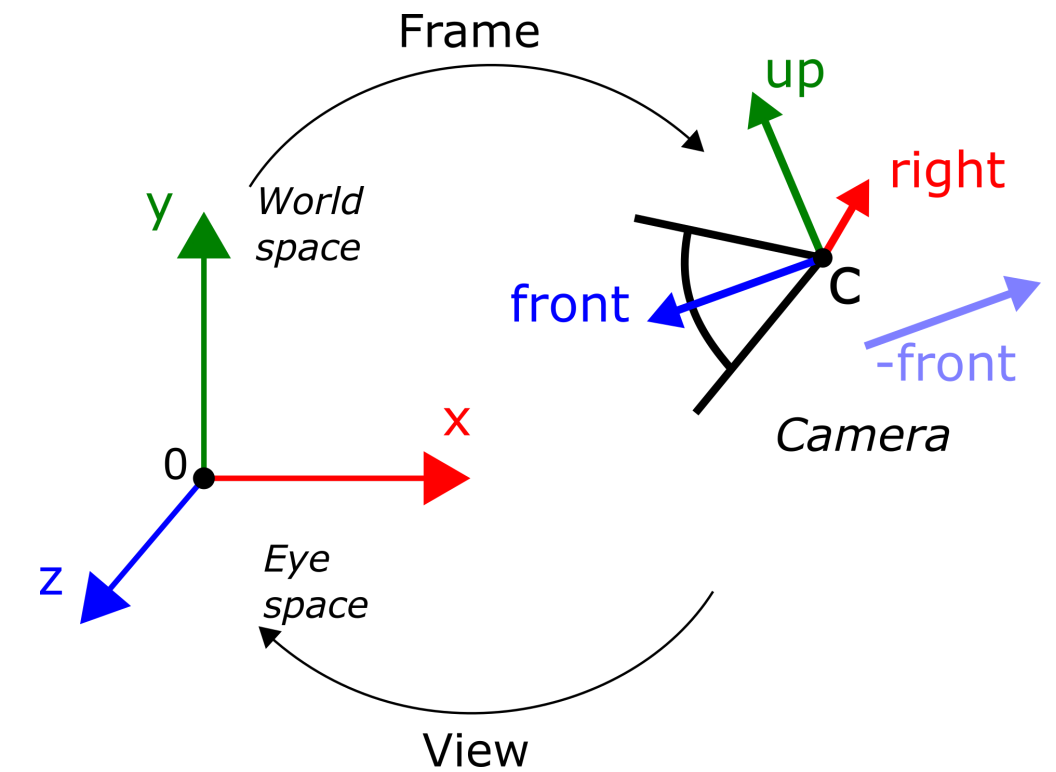


$$p = \text{Projection} \times \underbrace{\text{View} \times \text{Model}}_{\text{ModelView}} \times p_0$$

View representation as matrix

Camera parameterized by:

A center c ; right, up, front unit vectors
expressed in global coordinates



Frame matrix:

Transform unit coordinates into camera vectors

$$\text{Frame} \times (1, 0, 0, 0) = \text{right}$$

$$\text{Frame} \times (0, 1, 0, 0) = \text{up}$$

$$\text{Frame} \times (0, 0, 1, 0) = -\text{front}$$

$$\text{Frame} \times (0, 0, 0, 1) = c$$

Matrix of column-oriented vectors of camera directions

$$\text{Frame} = \begin{pmatrix} | & | & | & | \\ \text{right} & \text{up} & -\text{front} & c \\ | & | & | & | \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} O & c \\ 0 & 1 \end{pmatrix}$$

View matrix: inverse of the Frame matrix

Transforms the camera vectors into the unit coordinates

$$\text{View} \times \text{right} = (1, 0, 0, 0)$$

$$\text{View} \times \text{up} = (0, 1, 0, 0)$$

$$\text{View} \times -\text{front} = (0, 0, 1, 0)$$

$$\text{View} \times c = (0, 0, 0, 1)$$

Orientation O can be read on the rows

$$\text{View} = \begin{pmatrix} (\dots & \text{right} & \dots) & | \\ (\dots & \text{up} & \dots) & | \\ (\dots & -\text{front} & \dots) & | \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} O & -O^T c \\ 0 & 1 \end{pmatrix}$$

Camera in shader - Question

Q. In a vertex shader the View matrix is passed as uniform parameter
How can you compute the position of the camera ?

A usefull function to set a point of view is the standard *look_at(eye, center, up)*

- eye: position of the viewer
- center: position that the viewer is looking at
- up: a direction representing the "vertical direction" of the viewer.

Q. How to compute the *View* matrix from the *look_at* parameters ?

Interacting with camera

Camera orientation has 3 degrees of freedom (+ 3 dof in position)

How to handle a camera orientation using a mouse (2-dof) ?

Assuming a freely oriented camera

Common approaches

"Spherical camera"/coordinates (θ, ϕ)

(+) Good to easily "turn around" oriented structure

(-) Lacks a dof: No "twist": position on the unit sphere force the *angle*

Not adapted to turn around non-oriented/badly-oriented objects.

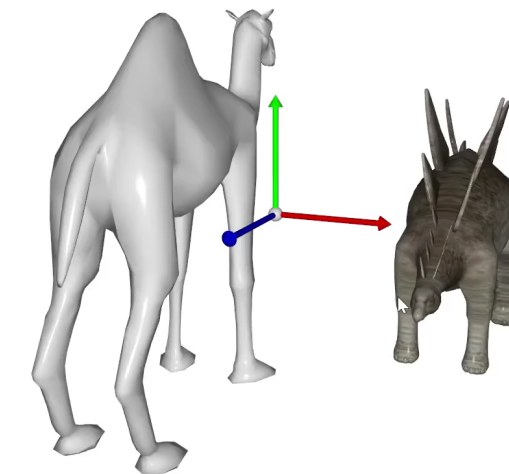
Spherical coordinates + twist (ex. roll, pitch, yaw)

(+) Precise and complete control

(-) Require an additional key/controller to the mouse to control the 3 dof

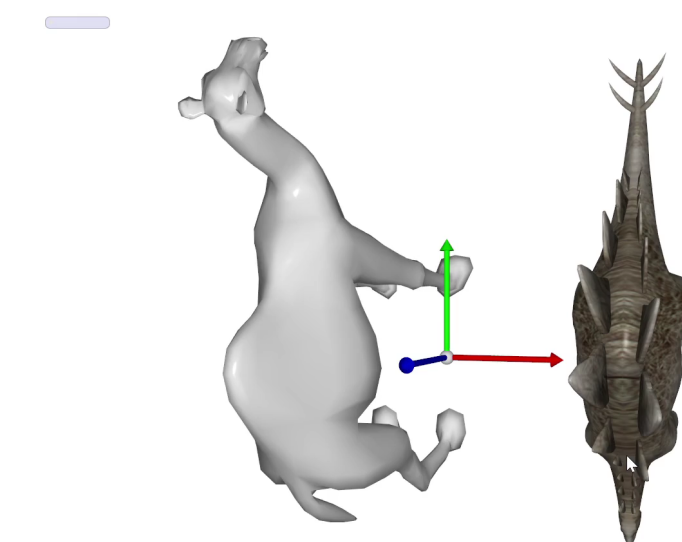
Example of "Spherical camera"

Correct orientation



Bad orientation of the shape

spherical camera is "stuck" in this orientation



Interacting with camera - ArcBall/Trackball metaphore

Other approach: **ArcBall/Trackball behavior**

3 dof: left/right, top/down, but also twist

Idea: Use the 2D cursor position as "3D point on the trackball"

Motion between 2 cursor positions = 3D Rotation applied to the ball

[K. Shoemake. A User Interface for Specifying Three-Dimensional Orientation Using a Mouse. Graphics Interface, 1992.]



Algorithm:

Inputs $p_1 = (p_{1x}, p_{1y})$, $p_2 = (p_{2x}, p_{2y})$ in screen coord.

$q_{1/2} = \text{ArcBallProjection}(p_{1/2})$

R = rotation between vectors (q_1, q_2)

$\text{ArcBallProjection}(p)$

$d = \text{norm}(p)$

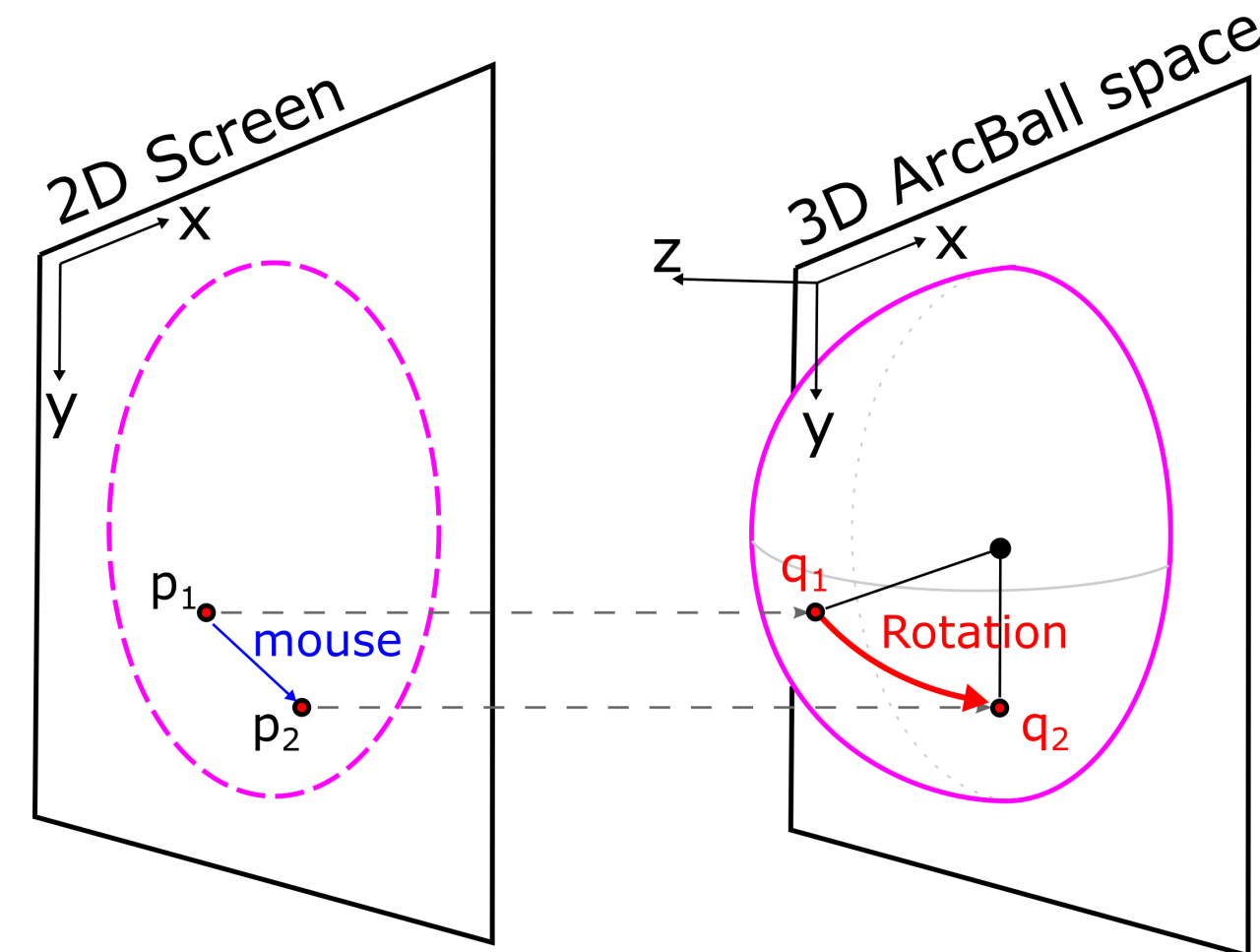
if ($d < 1/\sqrt{2}$)

$q = (p, \sqrt{1 - d^2})$

else // hyperbola

$q = (p, 1/(2d))$

return q



Note: the hyperbola smoothly extends the sphere in the entire screen.

Interacting with camera - ArcBall/Trackball metaphors

Natural way to rotate around a shape

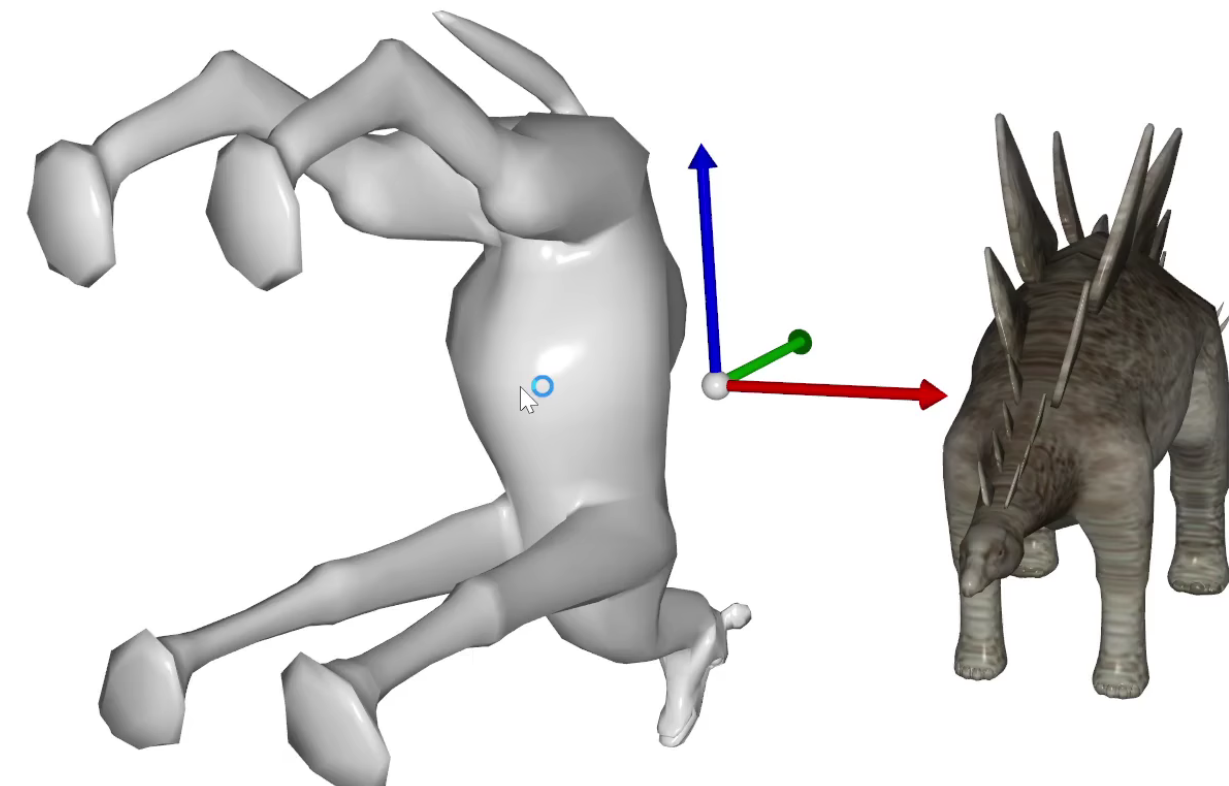
(+) No privileged axis

No reference configuration, works in incrementing rotations

Behavior is orientation invariant

(-) Less precise than individual dof control

Drift of twist when moving back and forth



Rotations

Rotations

3D rotations have 3 dof, No unique representation

Matrix

$$R = \begin{pmatrix} R_{xx} & R_{xy} & R_{xz} \\ R_{yx} & R_{yy} & R_{yz} \\ R_{zx} & R_{zy} & R_{zz} \end{pmatrix}$$

$$R^T R = I$$

$$\det(R) = 1$$

Euler Angles

3 angles: (α, β, γ)

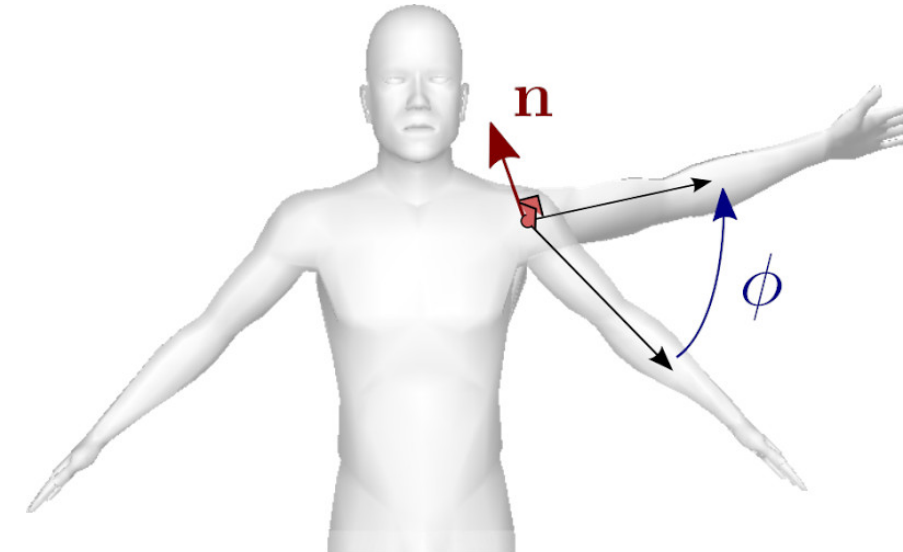
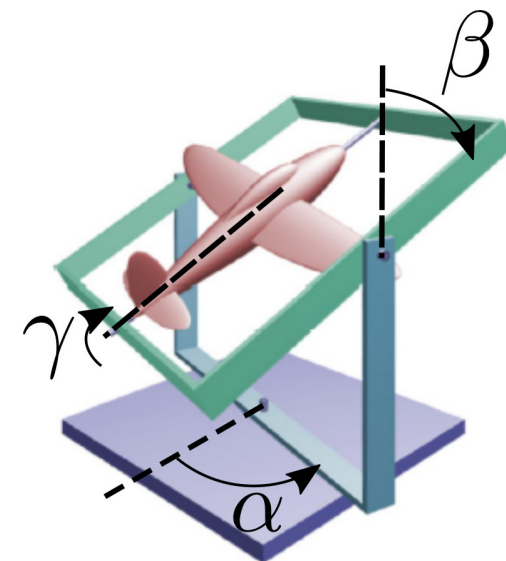
Composition of rotation
around basic axes

Axis angle

$(\mathbf{n}, \theta)$

Quaternion

$$q = (x, y, z, w) \\ = \left(\mathbf{n} \sin \left(\frac{\theta}{2} \right), \cos \left(\frac{\theta}{2} \right) \right)$$



Rotation: Focus Euler angles

Three consecutive rotations along fixed axis along (x, y, z) coordinates.

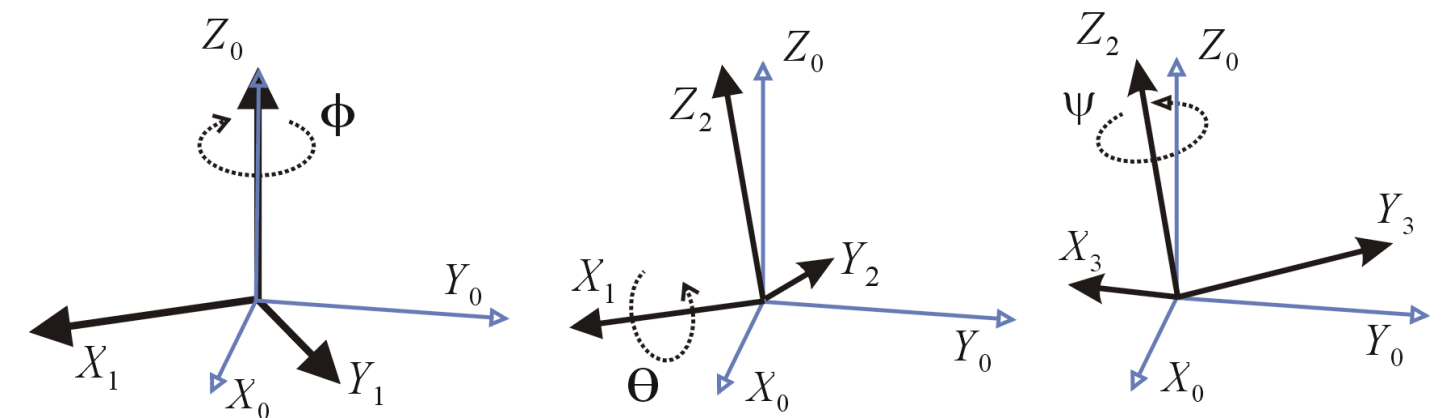
Can use basic rotation matrices composition

$$R_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & \sin(\alpha) \\ 0 & -\sin(\alpha) & \cos(\alpha) \end{pmatrix} \quad R_y = \begin{pmatrix} \cos(\beta) & 0 & \sin(\beta) \\ 0 & 1 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) \end{pmatrix} \quad R_z = \begin{pmatrix} \cos(\gamma) & \sin(\gamma) & 0 \\ -\sin(\gamma) & \cos(\gamma) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Multiple Euler angles conventions

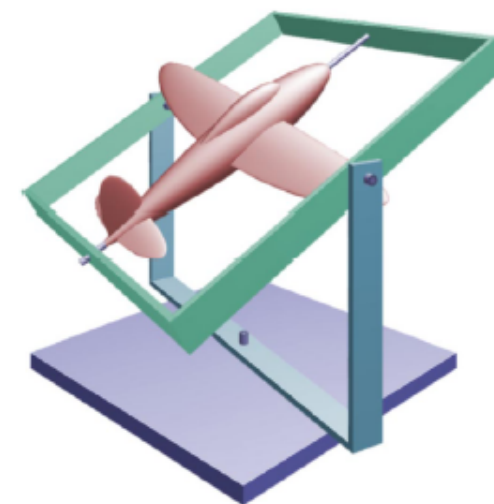
- *Proper Euler* : $z-x-z'$, $x-y-x'$, $y-z-y'$, ...
- *Tait-Bryan* : $x-y-z$, $z-y-x$, $x-z-y$, ...

Take care when exporting/importing/parsing between Softwares



Pro

- Combination of rotation around known axis
- Comprehensive parameters (3 dof)
- Animators can interact with angular curves
- *Widely used in robotics*



Rotation: Focus Euler angles

Limitations of Euler Angle

- *Gimbal Lock* when composing b/w some rotations

Loose one degree of freedom in specific configuration

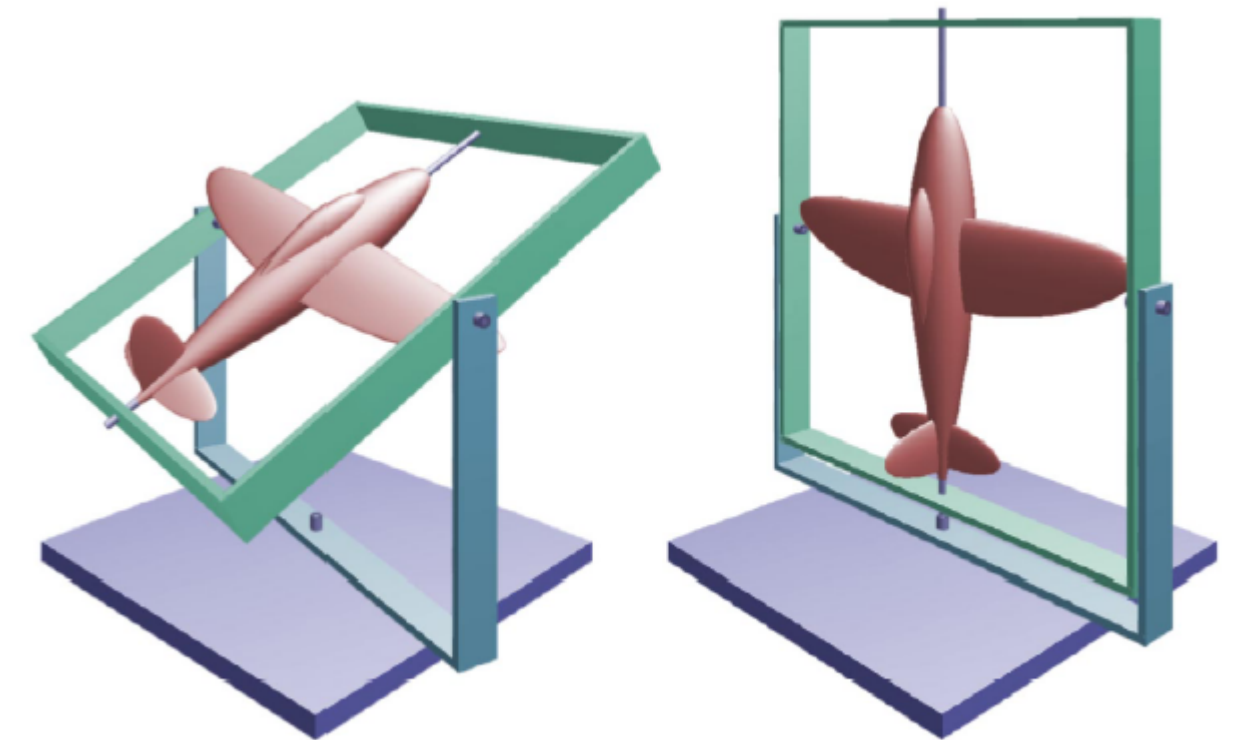
$$\text{ex. } R_x(\alpha) R_y(\pi/2) R_z(\gamma) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{pmatrix} \begin{pmatrix} \cos(\gamma) & -\sin(\gamma) & 0 \\ \sin(\gamma) & \cos(\gamma) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} 0 & 0 & 1 \\ \sin(\alpha) \cos(\gamma) + \cos(\alpha) \sin(\gamma) & -\sin(\alpha) \sin(\gamma) + \cos(\alpha) \cos(\gamma) & 0 \\ -\cos(\alpha) \cos(\gamma) + \sin(\alpha) \sin(\gamma) & \cos(\alpha) \sin(\gamma) + \sin(\alpha) \cos(\gamma) & 0 \end{pmatrix}$$

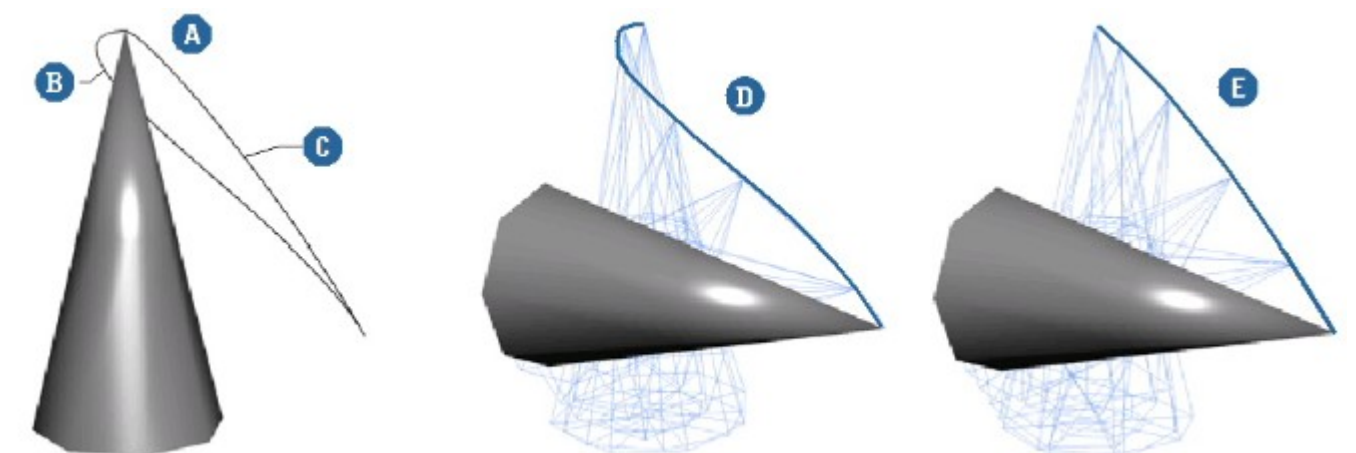
$$= \begin{pmatrix} 0 & 0 & 1 \\ \sin(\alpha + \gamma) & \cos(\alpha + \gamma) & 0 \\ -\cos(\alpha + \gamma) & \sin(\alpha + \gamma) & 0 \end{pmatrix}$$

Expect 2-dof, but get 1-dof

- Interpolation of 3 angles possible but not necessarily with the simplest trajectory.



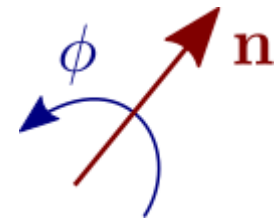
<http://www.fho-enden.de/~hoffmann/gimbal09082002.pdf>



Rotation: Focus Axis Angle

Any 3D rotation can be represented by

- A unit axis n
- An angle θ

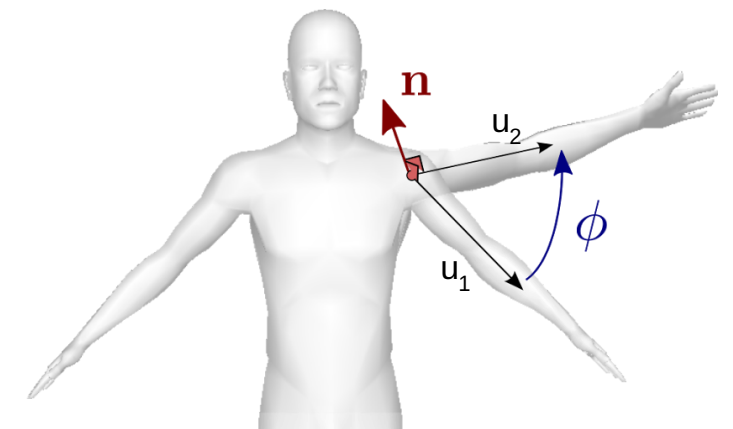
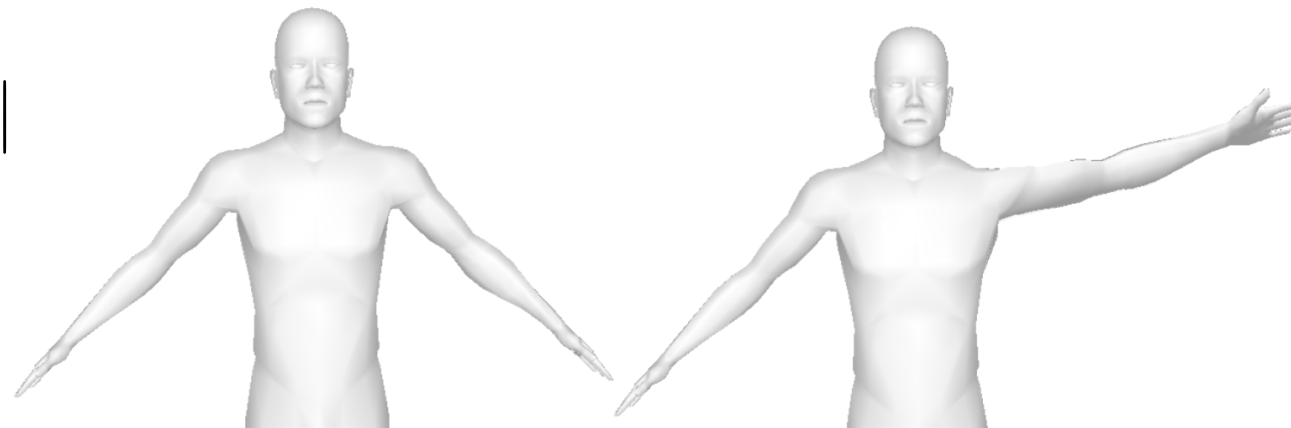


Pro.

- Concise representation: 3 DOF
- Meaningfull parameters
- Describes well the rotation between two vectors

Ex. Rotation between u_1 and u_2

- Axis of rotation $n = (u_1 \times u_2) / \|u_1 \times u_2\|$
- Angle of rotation $\theta = \arccos(u_1 \cdot u_2)$
- Twist around u_2 can be arbitrarily chosen



Rotation: Focus Axis Angle - Rodrigues

Applying a rotation (n, θ) to a vector v

$$v = v_{\parallel} + v_{\perp}$$

$$v' = v'_{\parallel} + v'_{\perp}$$

$$\Rightarrow v' = v_{\parallel} + (\cos(\theta) v_{\perp} + \sin(\theta) n \times v_{\perp})$$

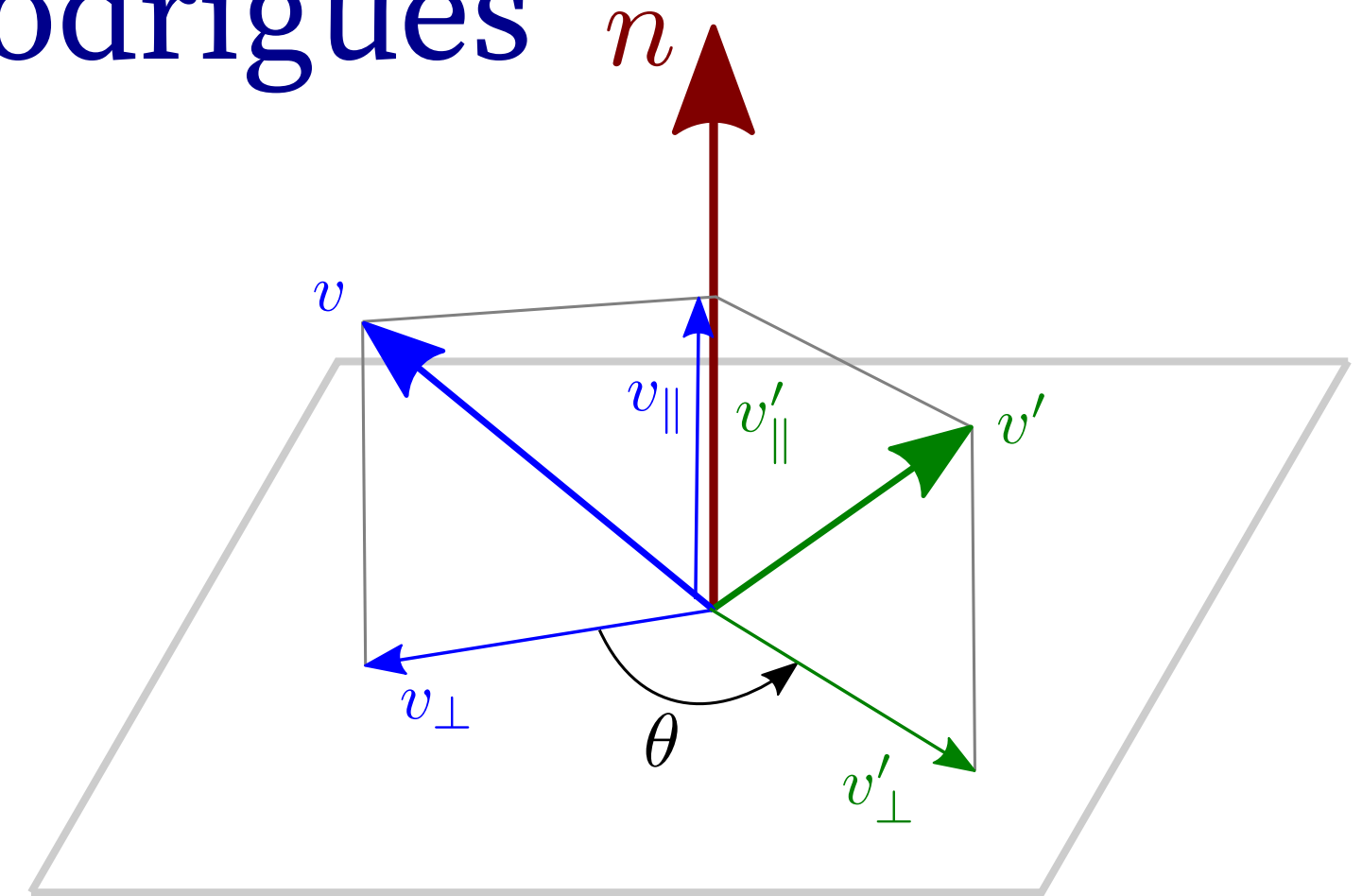
$$v_{\parallel} = (v \cdot n) n$$

$$v_{\perp} = v - (v \cdot n) n$$

$$v' = (v \cdot n) n + \cos(\theta)(v - (v \cdot n) n) + \sin(\theta) n \times (v - (v \cdot n) n)$$

$$\Rightarrow v' = \cos(\theta) v + \sin(\theta) n \times v + (1 - \cos(\theta)) (v \cdot n) n$$

Rodrigues' rotation Formula



Rotation: Focus Axis Angle as Matrix

$$v' = \cos(\theta) v + \sin(\theta) n \times v + (1 - \cos(\theta)) (v \cdot n) n = R(n, \theta) v$$

$$v' = \cos(\theta) v + \sin(\theta) \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix} \times v + (1 - \cos(\theta)) \begin{pmatrix} n_x & n_y & n_z \end{pmatrix} v \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix}$$

$$v' = \cos(\theta) v + \sin(\theta) \underbrace{\begin{pmatrix} 0 & -n_z & n_y \\ n_z & 0 & -n_x \\ -n_y & n_x & 0 \end{pmatrix}}_K v + (1 - \cos(\theta)) \underbrace{\begin{pmatrix} n_x^2 & n_x n_y & n_x n_z \\ n_x n_y & n_y^2 & n_y n_z \\ n_x n_z & n_y n_z & n_z^2 \end{pmatrix}}_{K^2} v$$

$$v' = \begin{pmatrix} \cos(\theta) + n_x^2(1 - \cos(\theta)) & n_x n_y(1 - \cos(\theta)) - n_z \sin(\theta) & n_x n_z(1 - \cos(\theta)) + n_y \sin(\theta) \\ n_x n_y(1 - \cos(\theta)) + n_z \sin(\theta) & \cos(\theta) + n_y^2(1 - \cos(\theta)) & n_y n_z(1 - \cos(\theta)) - n_x \sin(\theta) \\ n_x n_z(1 - \cos(\theta)) - n_y \sin(\theta) & n_y n_z(1 - \cos(\theta)) + n_x \sin(\theta) & \cos(\theta) + n_z^2(1 - \cos(\theta)) \end{pmatrix} v$$

Rotation: Focus Axis Angle - Summary

Given a rotation (n, θ) - Corresponding rotation matrix

$$R(n, \theta) = I + \sin(\theta) K + (1 - \cos(\theta)) K^2$$

$$R(n, \theta) = \begin{pmatrix} \cos(\theta) + n_x^2(1 - \cos(\theta)) & n_x n_y(1 - \cos(\theta)) - n_z \sin(\theta) & n_x n_z(1 - \cos(\theta)) + n_y \sin(\theta) \\ n_x n_y(1 - \cos(\theta)) + n_z \sin(\theta) & \cos(\theta) + n_y^2(1 - \cos(\theta)) & n_y n_z(1 - \cos(\theta)) - n_x \sin(\theta) \\ n_x n_z(1 - \cos(\theta)) - n_y \sin(\theta) & n_y n_z(1 - \cos(\theta)) + n_x \sin(\theta) & \cos(\theta) + n_z^2(1 - \cos(\theta)) \end{pmatrix}$$

Pro

- Concise and general representation
- Expressive parameters in 3D space (axe, angles)
- Efficient rotation and correspondance with matrix

Cons

- No simple composition expression between two rotations.
- No direct interpolation

(well handled by quaternion representation)

Rotation: Focus on Quaternions

Quaternions: generalization of complex numbers.

$q = x \mathbf{i} + y \mathbf{j} + z \mathbf{k} + w$ w real part, (x, y, z) imaginary (or pure quaternion) part.

We write in short $q = (x, y, z, w)$

(don't confound with 4D vectors in homogeneous coordinates)

Properties of imaginary basis vectors

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = -1$$

$$\mathbf{ij} = -\mathbf{ji} = \mathbf{k}$$

$$\mathbf{jk} = -\mathbf{kj} = \mathbf{i}$$

$$\mathbf{ki} = -\mathbf{ik} = \mathbf{j}$$

$$\mathbf{ijk} = -1$$

- Provides the algebraic properties

Basics operations on quaternions

- Conjugated quaternion $q^* = (-x, -y, -z, w)$

- Quaternion norm $\|q\| = \sqrt{q q^*} = \sqrt{x^2 + y^2 + z^2 + w^2}$. Unit quaternion satisfies $\|q\| = 1$.

- Quaternion product

$$q_1 q_2 = (x_1 \mathbf{i} + y_1 \mathbf{j} + z_1 \mathbf{k} + w_1) (x_2 \mathbf{i} + y_2 \mathbf{j} + z_2 \mathbf{k} + w_2) = \dots$$

$$q_1 q_2 = \begin{pmatrix} x_1 w_2 + w_1 x_2 + y_1 z_2 - z_1 y_2 \\ y_1 w_2 + w_1 y_2 + z_1 x_2 - x_1 z_2 \\ z_1 w_2 + w_1 z_2 + x_1 y_2 - y_1 x_2 \\ w_1 w_2 - x_1 x_2 - y_1 y_2 - z_1 z_2 \end{pmatrix}$$

Note: Quaternion product is

- associative: $(q_1 q_2) q_3 = q_1 (q_2 q_3) = q_1 q_2 q_3$
- **non-commutative**: $q_1 q_2 \neq q_2 q_1$

Sometimes interesting to separate *real part* w from *pure quaternion part* $s = (x, y, z)$.

- Shorthand vector form $q = (s, w)$

- Quaternion product in vector form $q_1 q_2 = (s_1 w_2 + s_2 w_1 + s_1 \times s_2, w_1 w_2 - s_1 \cdot s_2)$

Relation between quaternion and rotation

Consider

- a quaternion $q = (s, w)$ of unit norm $\|q\| = 1$.
- a vector $v = (v_x, v_y, v_z)$ assimilated to the pure quaternion $q_v = (v, 0) = (v_x, v_y, v_z, 0)$.

Then

- $q_{v'} = \mathcal{R}_q(v) = q q_v q^*$ is a pure quaternion $q_{v'} = (v'_x, v'_y, v'_z, 0)$
- And $v' = (v'_x, v'_y, v'_z)$ is the rotation of vector v around the axis $n = s/\|s\|$, with angle $2 \arccos(w)$.

Demonstration

$$\mathcal{R}_q(v) = (s, w) (v, 0) (-s, w) = \dots = ((w^2 - s^2) v + 2(s \cdot v) s + 2w (s \times v), 0)$$

As $\|q\| = 1$, we can write $q = (s, w) = (n \sin(\phi), \cos(\phi))$, where $\|n\| = 1$

$$\text{Then } \mathcal{R}_q(v) = (\underbrace{(\cos^2(\phi) - \sin^2(\phi))}_{\cos(2\phi)} v + \underbrace{2 \sin^2(\phi)}_{1 - \cos(2\phi)} (n \cdot v) n + \underbrace{2 \cos(\phi) \sin(\phi)}_{\sin(2\phi)} n \times v, 0)$$

$\Rightarrow$ Rodrigues formula for axis n and angle 2ϕ .

The unit quaternion $q = (n \sin(\theta/2), \cos(\theta/2))$ represents the rotation of angle θ around the axis n .

Composition of rotations

Consider two rotation (R_1, R_2) associated to their unit quaternions (q_1, q_2) .

The product $q_1 q_2$ represents the composition $R_1 \circ R_2$.

Demonstration

We show that $\mathcal{R}_{q_1 q_2}(v) = \mathcal{R}_{q_1} \circ \mathcal{R}_{q_2}(v)$.

$$\mathcal{R}_{q_1 q_2}(v) = (q_1 q_2) v (q_1 q_2)^*$$

$$\mathcal{R}_{q_1 q_2}(v) = (q_1 q_2) v (q_2^* q_1^*), \text{ as } (q_1 q_2)^* = q_2^* q_1^*$$

$$\mathcal{R}_{q_1 q_2}(v) = q_1 (q_2 v q_2^*) q_1^*$$

$$\mathcal{R}_{q_1 q_2}(v) = q_1 \mathcal{R}_{q_2}(v) q_1^* = \mathcal{R}_{q_1} \circ \mathcal{R}_{q_2}(v)$$

Correspondance quaternion to rotation matrix

The unit quaternion $q = (x, y, z, w)$ represents the rotation given by the matrix

$$R = \begin{pmatrix} 1 - 2(y^2 + z^2) & 2(xy - wz) & 2(xz + wy) \\ 2(xy + wz) & 1 - 2(x^2 + z^2) & 2(yz - wx) \\ 2(xz - wy) & 2(yz + wx) & 1 - 2(x^2 + y^2) \end{pmatrix}$$

Demonstration

$$v' = \mathcal{R}_q(v) = q q_v q^* = ((w^2 - s^2) v + 2(s \cdot v) s + 2w (s \times v), 0) \quad \text{with } s = (x, y, z)$$

$$v' = (w^2 - x^2 - y^2 - z^2)v + 2 \begin{pmatrix} x & y & z \end{pmatrix} v \begin{pmatrix} x & y & z \end{pmatrix}^T + 2w \begin{pmatrix} x & y & z \end{pmatrix} \times v$$

$$v' = \left((w^2 - x^2 - y^2 - z^2) \mathbf{I} + 2 \begin{pmatrix} x^2 & xy & xz \\ xy & y^2 & yz \\ xz & yz & z^2 \end{pmatrix} + 2w \begin{pmatrix} 0 & -z & y \\ z & 0 & -x \\ -y & x & 0 \end{pmatrix} \right) v$$

$$v' = R v, \text{ with } x^2 + y^2 + z^2 + w^2 = 1$$

Summary - Correspondance quaternion / matrix-vector

Representation and Operations

	3D space	Quaternion space
<i>Vector</i>	$v = (v_x, v_y, v_z)$	$q_v = (v, 0) = (v_x, v_y, v_z, 0)$
<i>Rotation</i>	R (3×3 matrix)	$q = (x, y, z, w), \ q\ = 1$
<i>Apply rotation to vector</i>	Rv	$q q_v q^\star$
<i>Rotation composition</i>	$R_1 R_2$	$q_1 q_2$

Rotation to Quaternion

Rotation of axis n and angle $\theta \Rightarrow q = (n \sin(\theta/2), \cos(\theta/2))$

Quaternion to Rotation

$$\text{Unit quaternion } q = (x, y, z, w) \Rightarrow R = \begin{pmatrix} 1 - 2(y^2 + z^2) & 2(xy - wz) & 2(xz + wy) \\ 2(xy + wz) & 1 - 2(x^2 + z^2) & 2(yz - wx) \\ 2(xz - wy) & 2(yz + wx) & 1 - 2(x^2 + y^2) \end{pmatrix}$$

Interpolation

Rotations

Interpolating rotation

Problem: Given 2 rotations r_1, r_2

- Find rotation $r(t)$ st $r(0) = r_1, r(1) = r_2$, and varies smoothly along t

Matrix representation

Componentwise interpolation not adapted

ex. $t R_1 + (1 - t) R_2$ is not a rotation: introduce scaling/shearing

The correct formulation to interpolate on the manifold would be $R_1 \exp(t \log(R_1^T R_2))$

But matrix exponential is complex to compute

Euler angle

Can interpolate the 3 angles separately

(+) Leads to a rotation

(-) Doesn't necessarily follows simplest trajectory

Axis-Angle

No trivial interpolation scheme with different axes

Quaternion

Well adapted (see later)

Interpolation

Affine Transforms

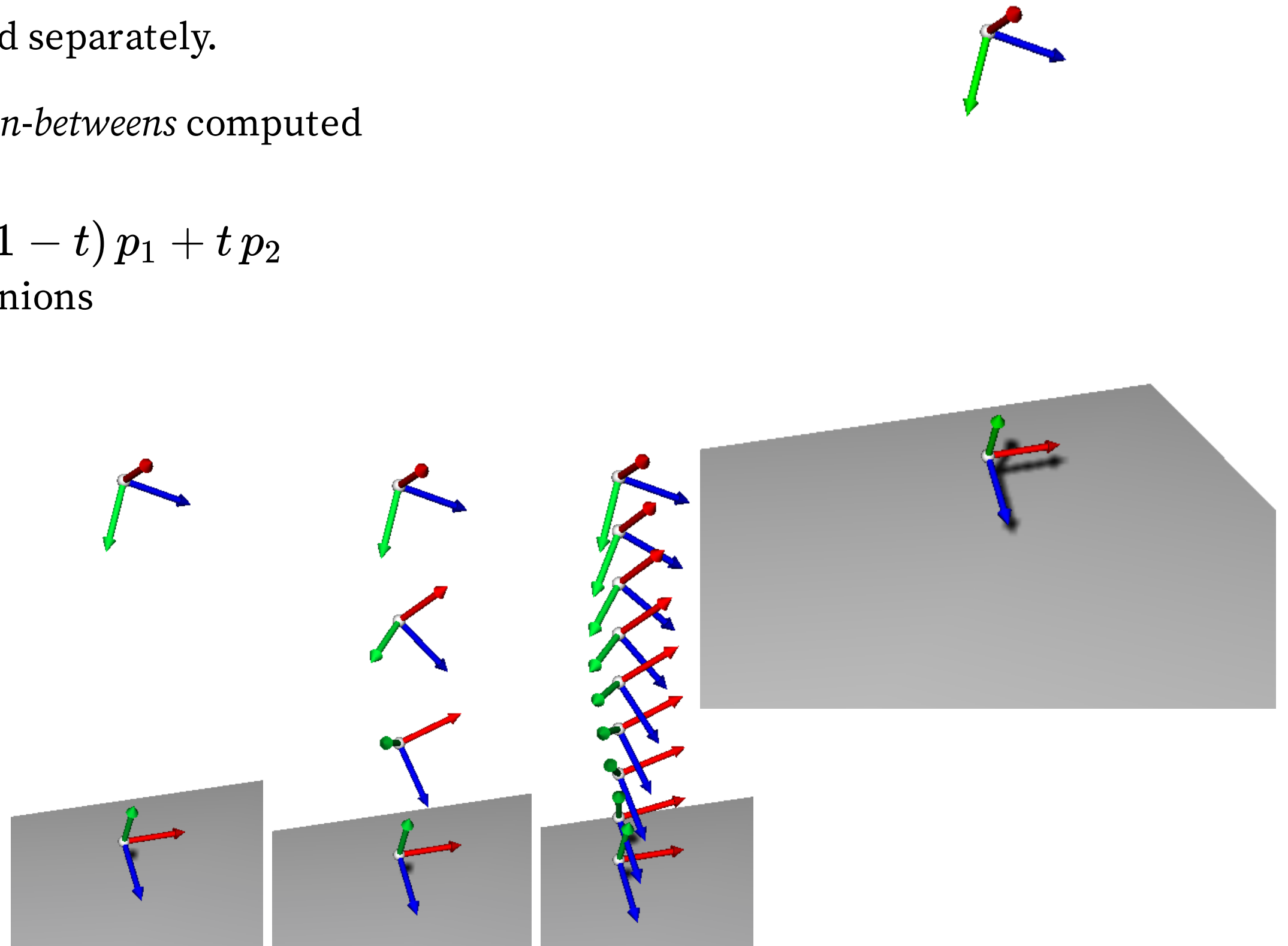
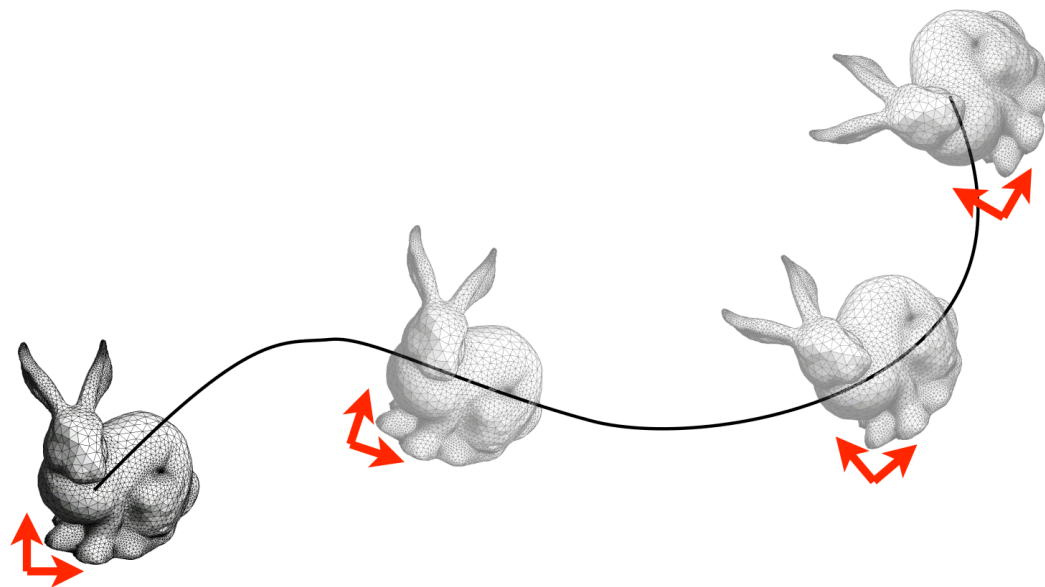
Interpolating rigid motion

3D objects have orientation r and position p .

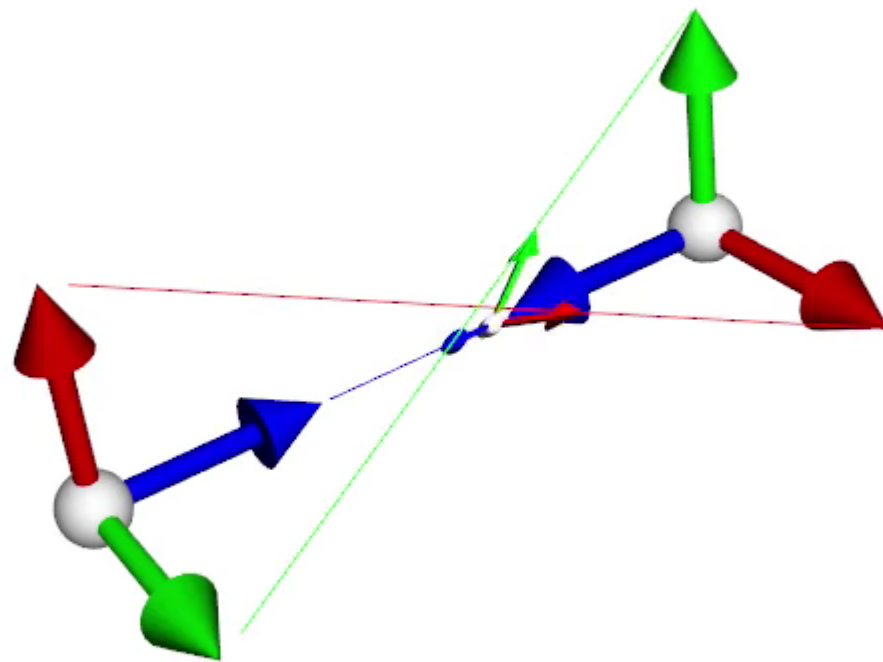
⇒ Need to interpolate both, usually handled separately.

Given two key-positions (p_1, r_1) and (p_2, r_2) *in-betweens* computed at time $t \in [0, 1]$ as

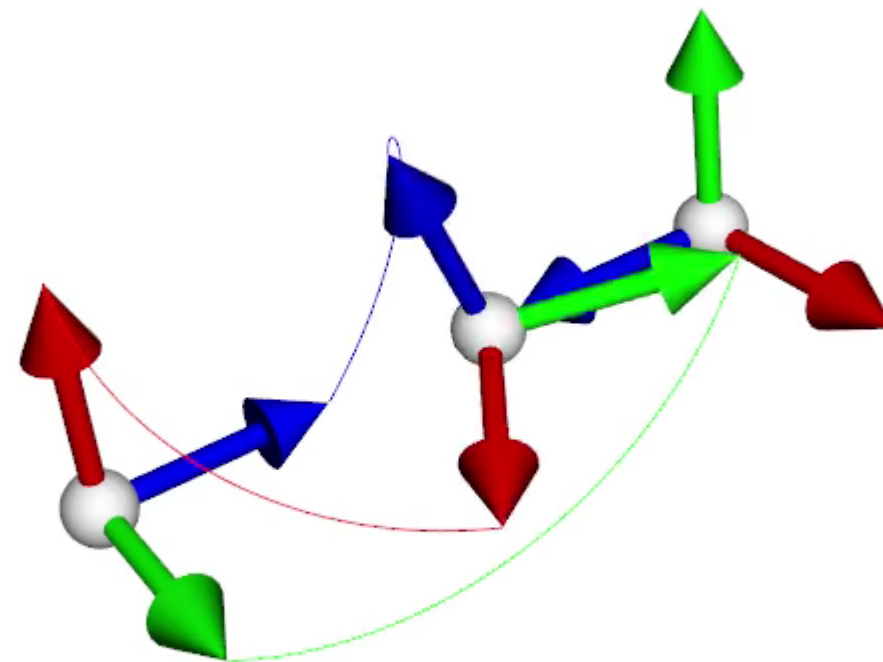
- Linear interpolation of positions $p(t) = (1 - t)p_1 + tp_2$
- Interpolate rotation with SLERP on quaternions
 - Convert $(r_1, r_2) \rightarrow (q_1, q_2)$
 - Compute $q(t) = \text{SLERP}(q_1, q_2, t)$
 - Convert back $q(t) \rightarrow r(t)$



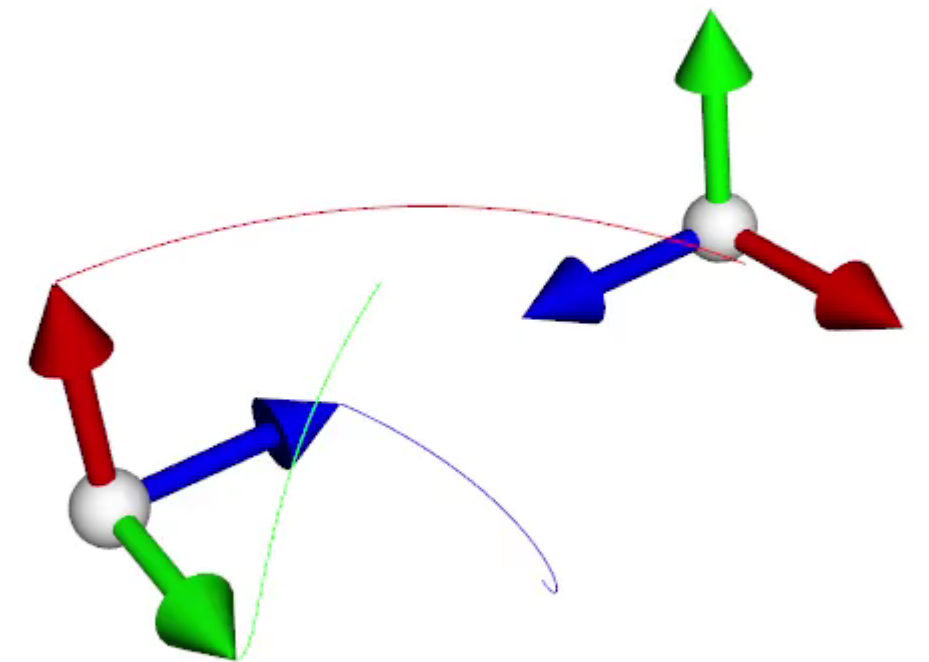
Interpolating rigid motion - Comparison



Matrix interpolation



Euler angle interpolation



Quaternion interpolation