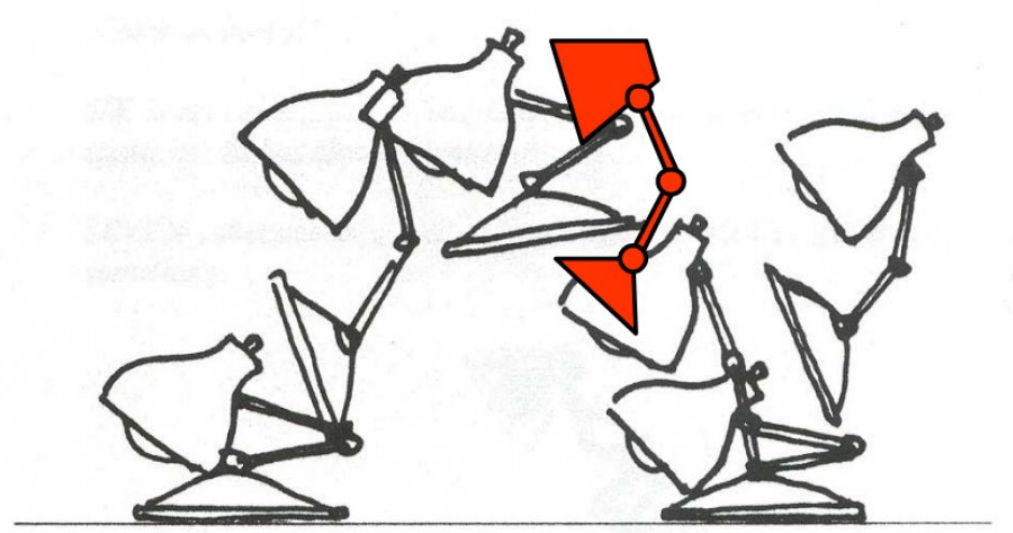
Keyframe interpolation

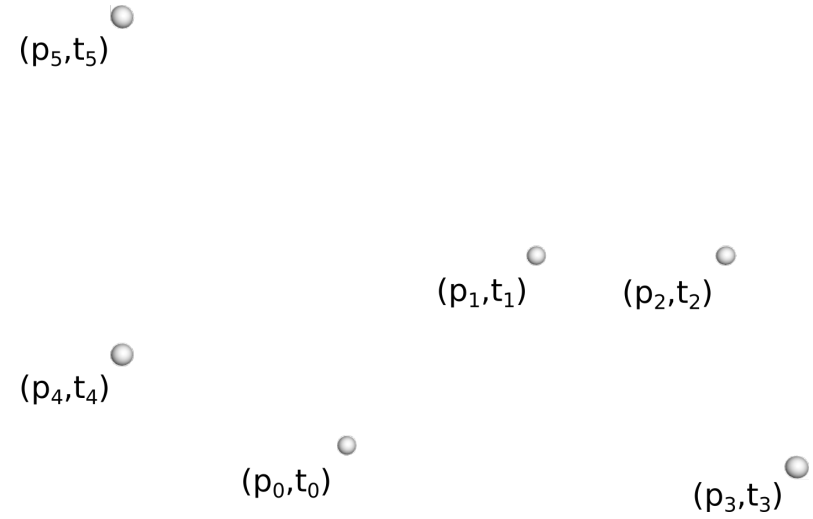
- **Interpolate positions**
- Interpolate rotations



Interpolate positions

Objective

- Given a set of key-positions (positions+time) we want to find an interpolating space-time curve



- **Input** $(p_i, t_i), i \in [0, N-1]$
- **Output**
  - Space time curve $p(t), t \in [t_0, t_{N-1}]$
  - Space time curve $p(t_i) = p_i$

Linear Interpolation

- Simplest solution: linear interpolation between each sample pairs

- $\forall t \in [t_i, t_{i+1}]$, $\begin{cases} p(t) = (1 - \alpha(t))\, p_i + \alpha(t)\, p_{i+1} \\ \alpha(t) = \dfrac{t - t_i}{t_{i+1} - t_i} \end{cases}$

Pros

- Simple
- Constant speed between keyframes

  Easy to ajust

Cons

- Non smooth trajectory
- Generates straight segments

  Artists prefer non straight trajectories for "real" looking motion

Smooth curve

Objective: Generate a **smooth** interpolating space-time curve

Classical Idea Use polynomials $p(t) = \sum_{i=0}^{d} \left(\sum_{j=0}^{N-1} c_i^j \, p_j \right) t^i$

Or formulated using basis functions $p(t) = \sum_{i=0}^{N-1} \alpha_i(t) \, p_i$

(α_i) polynomial basis function of degree d

- *Which polynomials/degree choose* ?

Lagrange polynomial interpolation

Naive idea: Interpolate all points at once

$$\forall t \in [t_0, t_{N-1}]\ ,\ \ p(t) = \sum_{i=0}^{N-1} \alpha_i(t)\ p_i \qquad \forall i \in [0, N-1]\ \ p(t_i) = p_i$$

Degree of polynomial : $N-1$

- Known solution: **Lagrange polynomial**

$$p(t) = \sum_{i=0}^{N-1} \alpha_i(t)\ p_i \qquad \alpha_i(t) = \prod_{k=0, k\neq i}^{N-1} \frac{t - t_k}{t_i - t_k}$$

- Explanation: By construction $\alpha_i(t_i) = 1$ and $\alpha_i(t_k) = 0$

(+) Interpolate all points
(-) Large oscillations between samples for large degree.
(-) Non local influence

=> Not used in practice

Lagrange polynomial interpolation : Comparison

Ex. Global effect on curve when two samples are added.

10 samples

12 samples

Spline

Idea

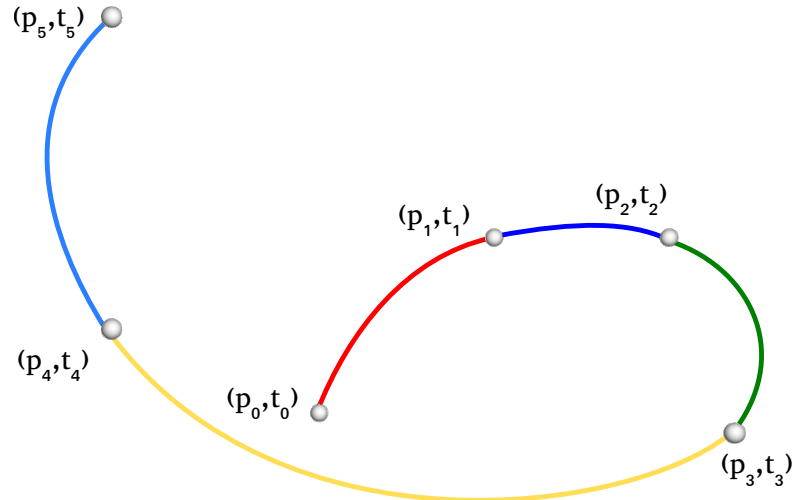
- Define on each part a polynomial
- Smooth junctions between them.

How to choose the polynomial

- Sufficiently high degree to be smooth
- Sufficiently low degree to avoid oscillations

=> In Graphics cubic polynomials are often used

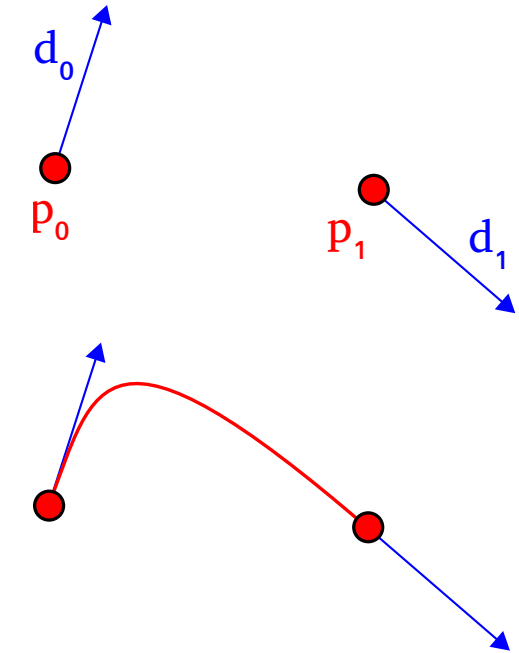
Allows up to C^2 junctions



Each color is another polynomial

Hermite interpolation

Hermite interpolation : cubic curve interpolating points and derivatives at extremities



Consider the following constraints

$$\begin{cases} p(s) = c_3 s^3 + c_2 s^2 + c_1 s + c_0 \\ p(0) = p_0,\ p(1) = p_1,\ p'(0) = d_0,\ p'(1) = d_1 \end{cases}$$

$\Rightarrow$ System of equations

$$\begin{cases} c_3 + c_2 + c_1 + c_0 = p_0 \\ c_3 + c_2 + c_1 + c_0 = p_1 \\ c_3 + c_2 + c_1 + c_0 = d_0 \\ 3\,c_3 + 2\,c_2 + c_1 + c_0 = d_1 \end{cases} \Rightarrow \begin{cases} c_0 = p_0 \\ c_1 = d_0 \\ c_2 = -3p_0 + 3p_1 - 2d_0 - d_1 \\ c_3 = 2p_0 - 2p_1 + d_0 + d_1 \end{cases}$$

$$\forall s \in [0,1]\ ,\ \ p(s) = (2s^3 - 3s^2 + 1)\, p_0 + (s^3 - 2s^2 + s)\, d_0 + (-2s^3 + 3s^2)\, p_1 + (s^3 - s^2)\, d_1$$

For arbitrary $t \in [t_i, t_{i+1}]$, we set $s = \dfrac{t - t_i}{t_{i+1} - t_i}$

Interpolating curve

Our initial problem: set of multiple keyframes position+time

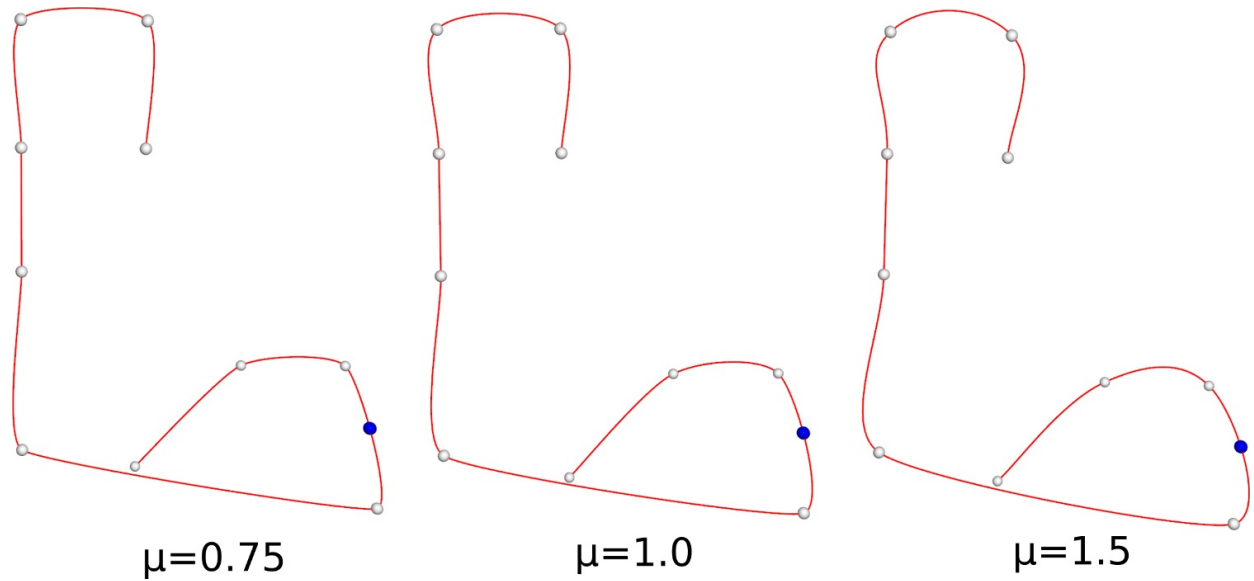
Two solutions
- Set explicitely derivatives for each keyframe - *teadious*
- Compute automatically plausible derivatives from surrounding samples - *often used*

Cardinal spline

Set $d_i = \mu \dfrac{p_{i+1} - p_{i-1}}{t_{i+1} - t_{i-1}}$

- μ curve tension $\in [0, 2]$
- $\mu = 1$ is commonly used

Catmull Rom Spline



Wrap-up Algorithm

Compute $p(t)$ as a cubic spline interpolation

- Given keyframes $(p_i, t_i)_{i \in [0, N-1]}$
- Given time $t \in [t_1, t_{N-2}]$

1. Find i such that $t \in [t_i, t_{i+1}]$

2. Compute $d_i = \mu \dfrac{p_{i+1} - p_{i-1}}{t_{i+1} - t_{i-1}}$, and $d_{i+1} = \mu \dfrac{p_{i+2} - p_i}{t_{i+2} - t_i}$

3. Compute $p(t) = (2s^3 - 3s^2 + 1)\, p_i + (s^3 - 2s^2 + s)\, d_i + (-2s^3 + 3s^2)\, p_{i+1} + (s^3 - s^2)\, d_{i+1}$

   with $s = \dfrac{t - t_i}{t_{i+1} - t_i}$.

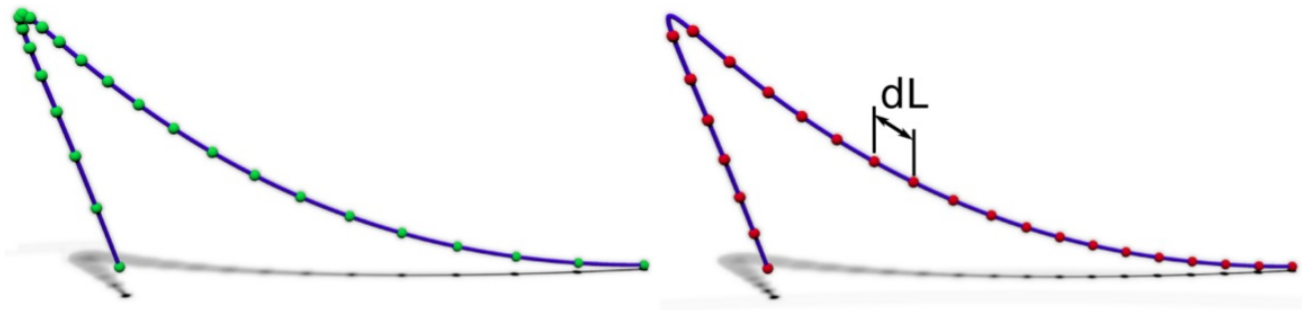
Limitation of cubic curve interpolation

- Only C^1, but not C^2 at junctions : Curvature/acceleration discontinuity

  - *Force C^2 continuity for cubic polynomial (global linear system to solve - loose local structure)*
  - *Consider higher degree polynomial*

- Non constant speed along each polynomial

  - *Reparametrization with curvilinear length*



Usage of keyframes interpolation

Interpolate every vertex of multiple meshes

Multi-target blending and blend shapes

The standard for facial animation

Full blend shape

- Per-vertex formulation $p^i = \sum_k \omega_k \, b_k^i$
- Matrix formulation $\mathbf{p} = \mathrm{B}\,\omega$

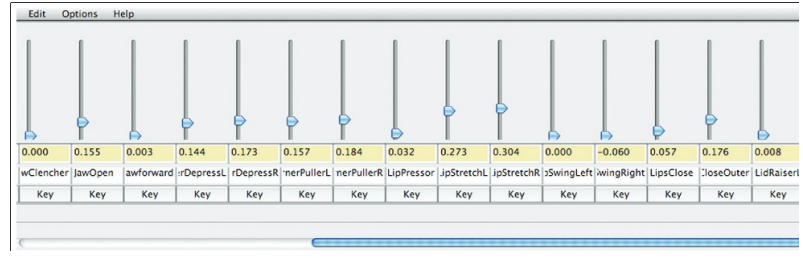
Differential representation

- Per-vertex formulation $p = b_0 + \sum_k (\underbrace{b_k - b_0}_{d_k})$
- Matrix formulation $\mathbf{p} = \mathbf{b_0} + \mathrm{D}\,\omega$

$\Rightarrow$ Allows expression transfert.

Problems:

- D is not an orthogonal matrix (non unique combination, redundancies) $\Rightarrow$ decomposition PCA, etc.
- How to directly manipulate blend shapes



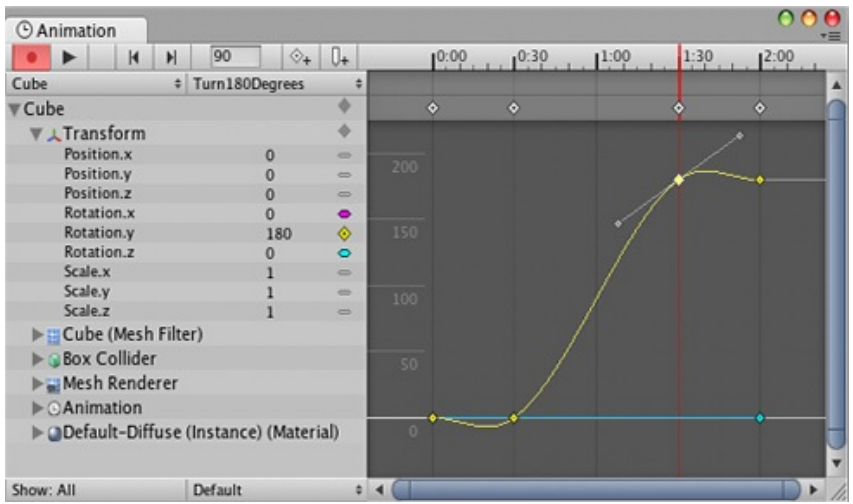
[J.P. Lewis et al. Practice and Theory of Blendshape Facial Models. EG STAR 2014]

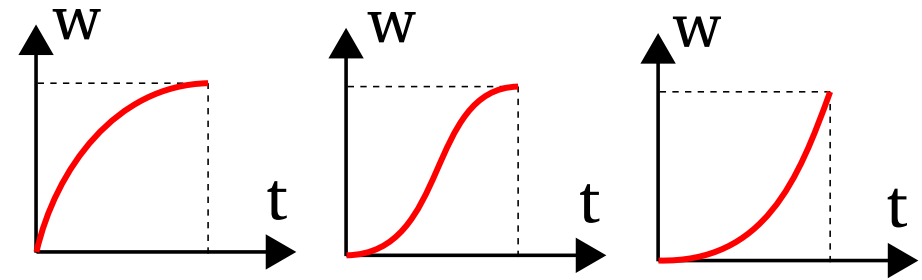
Curve editing

- Animation software (Maya, 3DSMax, Blender, etc) always come with a **curve editor**.

- Artists can manually ajust their position, time, and **derivatives** on curve editor.
  - One curve for each scalar parameter
    - position (x, y, z)
    - scaling (s_x, s_y, s_z)
    - rotation/quaternion (q_x, q_y, q_z, q_w)

- Can also use a wrapper function w to change time $p(t) = f(w(t))$



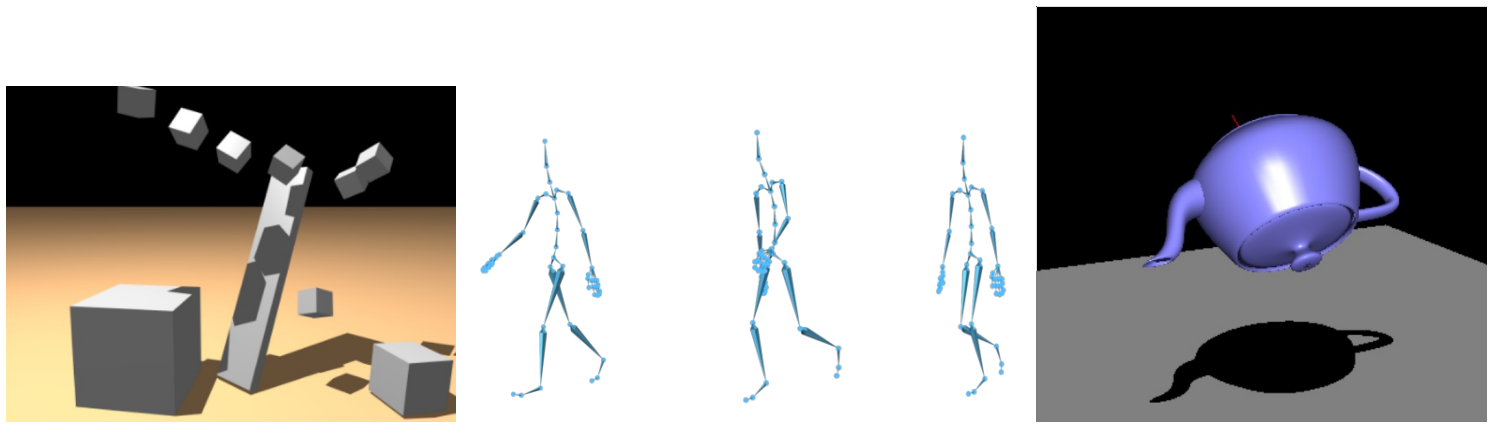


Interpolate rotations

and rigid transformations

Rotation

When dealing with rigid objects (ex. Character skeleton, oriented rigid objects, etc) orientations (thus rotations) are involved.



- How to **encode and interpolate rotations** ?

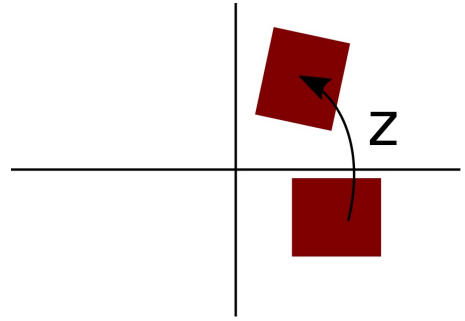
Rotation in 2D: Easy

1 - Model rotation by an angle

2 - Interpolate the angle

Can be formalized using complex numbers representation

- $R_1 \rightarrow z_1 = e^{i\,\theta_1}$
- $R_2 \rightarrow z_2 = e^{i\,\theta_2}$
- $R^{\alpha}_{1\rightarrow 2} \rightarrow z = e^{i\,(\alpha\theta_1 + (1-\alpha)\theta_2)}$



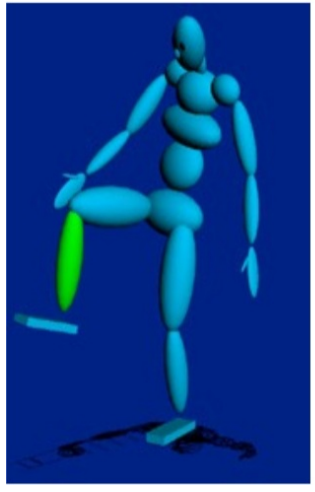
In 3D: not so easy ...

How many angles do I need to represent a 3D rotation?

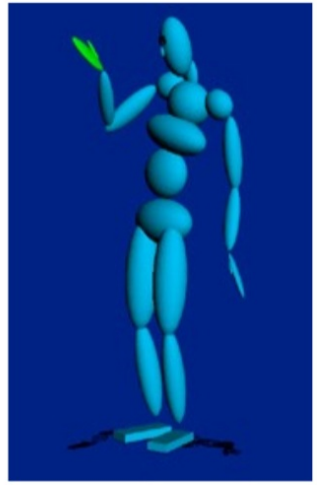
Rotation in 3D - DOF

- 3 Degrees of Freedom (dof)

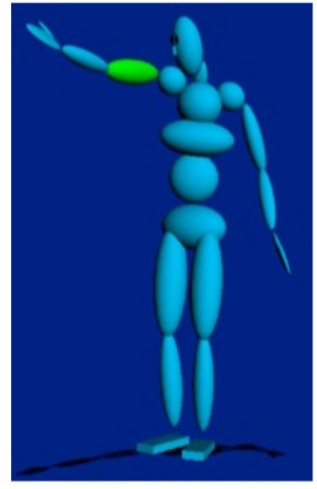
1 DOF: knee



2 DOF: wrist



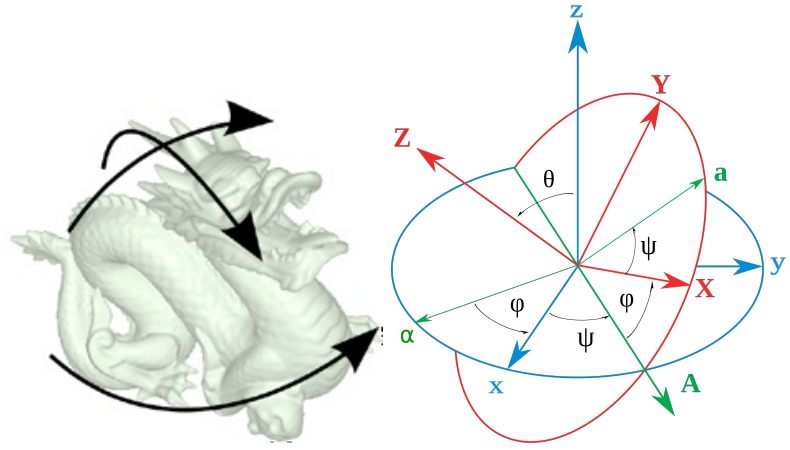
3 DOF: arm



Representing 3D Rotations

Multiple ways to represent rotations

1 - Matrices
2 - Euler (/Trait-Bryan) angles
3 - Axis-Angle
4 - Quaternion



Rotation representation: Matrix

$$\begin{cases} \mathrm{R} = \begin{pmatrix} r_{00} & r_{01} & r_{02} \\ r_{10} & r_{11} & r_{12} \\ r_{20} & r_{21} & r_{22} \end{pmatrix} \\ \\ \mathrm{R}^T \, \mathrm{R} = \mathrm{Id} \end{cases}$$

Pro.

- Standard structure (storage,manipulate)
- Simple application to vector: $v' = \mathrm{R}v$
- Composition b/w rotations: $\mathrm{R} = \mathrm{R}_1 \, \mathrm{R}_2$

Cons.

- Large redundancy (*9 coefficients for 3 dof*).
- Rotation parameters don't appear explicitely
- Unclear interpolation
  - Linear interpolation doesn't work well for large angles

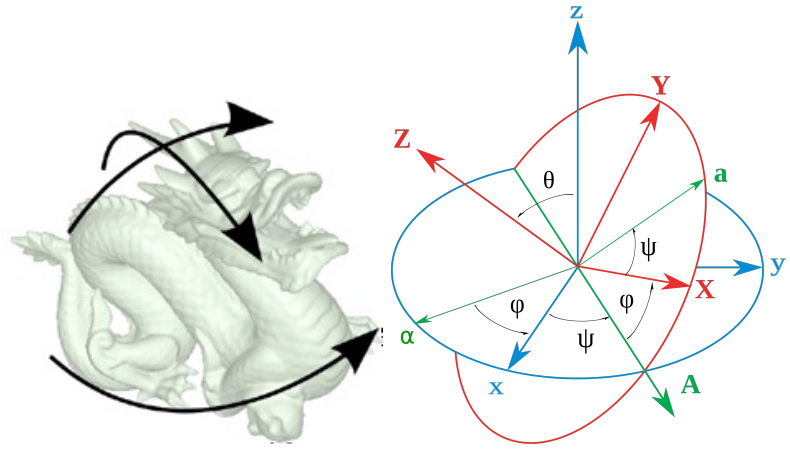
  ex. $\mathrm{M} = (1-\alpha) \, \mathrm{R}_1 + \alpha \mathrm{R}_2$

  => M is not a rotation anymore (not a vectorial space)

Representing 3D Rotations

Multiple ways to represent rotations

1 - Matrices
2 - Euler (/Trait-Bryan) angles
3 - Axis-Angle
4 - Quaternion



Rotation representation: Euler angles

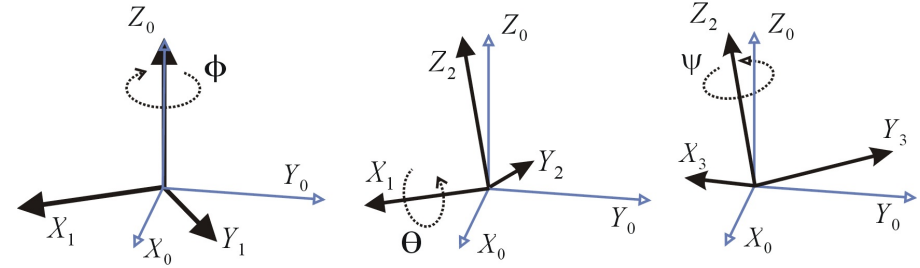
Three consecutive rotations along (x, y, z) coordinates.

Can use basic rotation matrices

$$R_x = \begin{pmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} R_y = \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{pmatrix} R_z = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) \\ 0 & -\sin\theta & \cos(\theta) \end{pmatrix}$$

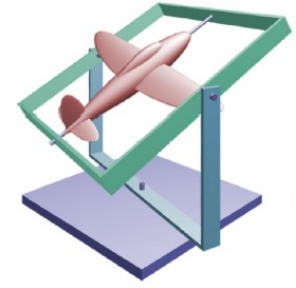
Multiple Euler angles conventions

- *Proper Euler : z-x-z, x-y-x, y-z-y, ...*
- *Trait-Bryan : x-y-z, y-z-x, z-x-y-, ...*



Pro

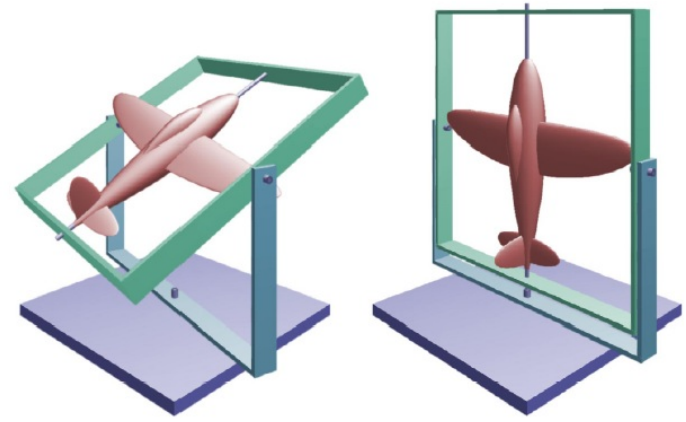
- Combination of rotation around known axis
- Comprehensive parameters (3 dof)
- Animators can interact with angular curves
- Easy conversion to matrix
- *Widely used in robotics*



Rotation representation: Euler angles

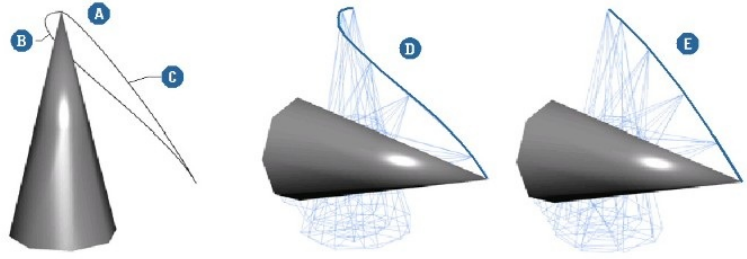
Limitations of Euler Angle

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



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

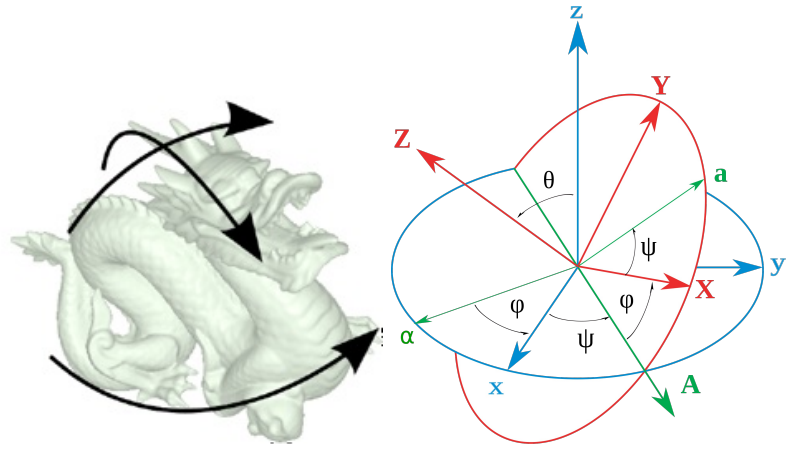
- Interpolation of 3 angles leads to non-intuitive trajectory



Representing 3D Rotations

Multiple ways to represent rotations

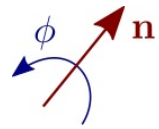
1 - Matrices
2 - Euler (/Trait-Bryan) angles
3 - Axis-Angle
4 - Quaternion



Rotation representation: Axis-Angle

Any 3D rotation can be represented by

- A unit axis n
- An angle θ

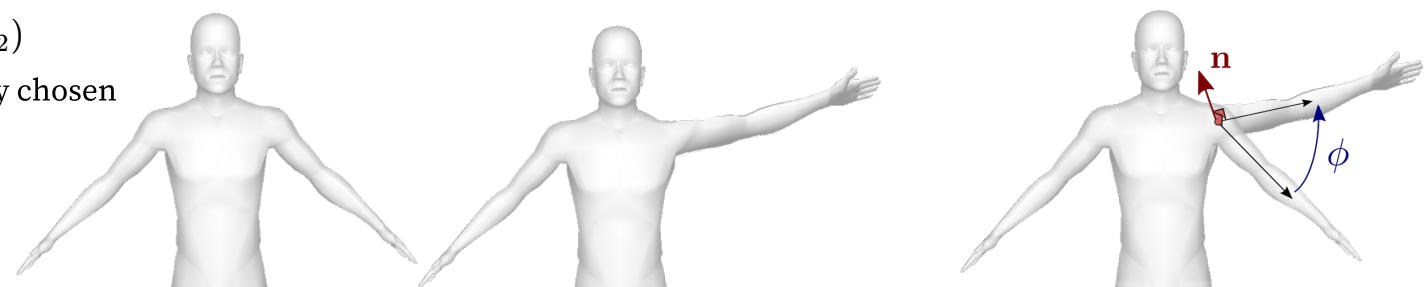


Pro.

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

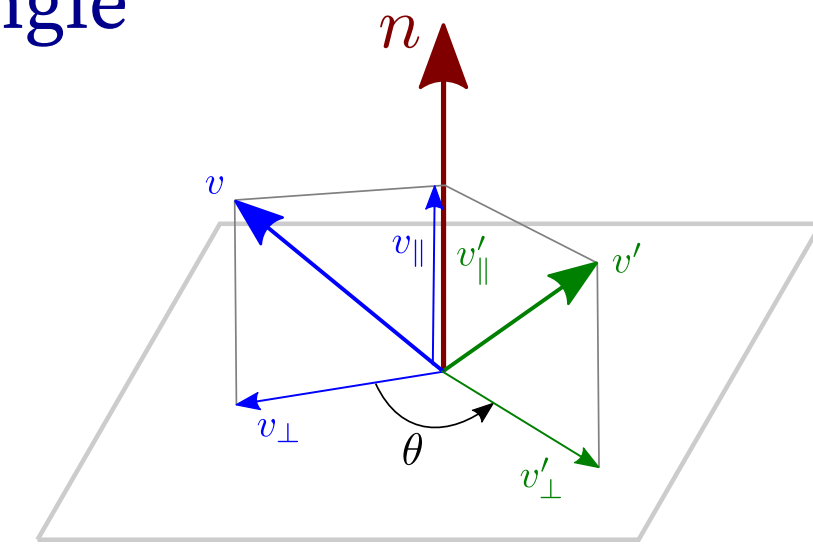
Ex. Rotation between u_1 and u_2

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



Rotation representation: Axis-Angle

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

Axis-Angle as matrix

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

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

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

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

Rotation representation: Axis-Angle

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

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

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

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

Composition between axis angle representation

Given two rotations (n_1, θ_1), (n_2, θ_2)

The composition $(n_3, \theta_3) = (n_1, \theta_1) \circ (n_2, \theta_2)$ is expressed by

- $$\cos\left(\frac{\theta_3}{2}\right) = \cos\left(\frac{\theta_1}{2}\right)\cos\left(\frac{\theta_2}{2}\right) - \sin\left(\frac{\theta_1}{2}\right)\sin\left(\frac{\theta_2}{2}\right) n_1 \cdot n_2$$

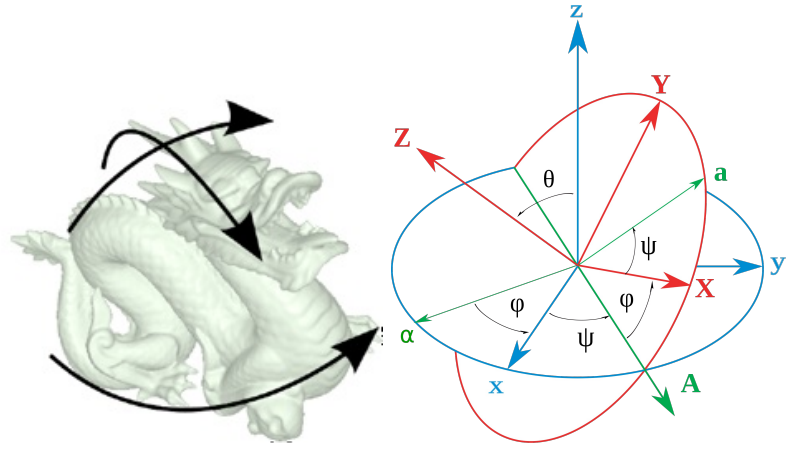
- $$n_3 = \frac{1}{\tan\left(\frac{\theta_3}{2}\right)} \frac{\tan\left(\frac{\theta_2}{2}\right) n_2 + \tan\left(\frac{\theta_1}{2}\right) n_1 - \tan\left(\frac{\theta_1}{2}\right)\tan\left(\frac{\theta_2}{2}\right) n_1 \times n_2}{1 - \tan\left(\frac{\theta_1}{2}\right)\tan\left(\frac{\theta_2}{2}\right) n_1 \cdot n_2}$$

- Can be derived using quaternions
- Rare to use the formula as it is

Representing 3D Rotations

Multiple ways to represent rotations

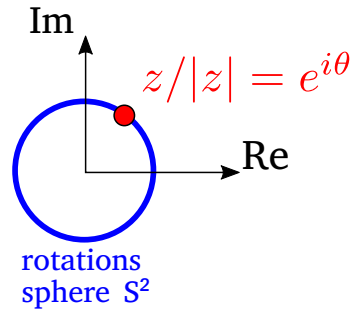
1 - Matrices
2 - Euler (/Trait-Bryan) angles
3 - Axis-Angle
4 - Quaternion



Intuition for quaternion

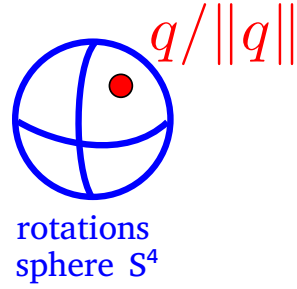
Going back to 2D and complex space

- 2D rotations have 1 dof. (rotation angle)
- 2D rotation represented as a point on a unit circle (S^1).
  - *1D structure embedded in a 2D space.*
  - *Complex space has algebraic structure adapted to model 2D rotation.*



Moving to 3D

- 3D rotations have 3 dof.
  - *(point on a unit 3D sphere allows only 2 dof : not sufficient ex. orientation that can twist)*
  - point on a unit 4D sphere allows 3 dof
- 3D rotation are represented as **point on the unit sphere in 4D** (S^3).
  - *3D structure embedded in a 4D space.*
  - *Complex quaternion space has algebraic structure adapted to model 3D rotation.*



Big picture of quaternions

Share similarities with 2D complex

- Complex value object with 4 components $q = (x, y, z, w)$
- Unit quaternions represent rotations $q/\|q\|$.
- Quaternion product represent rotation composition $q_1\, q_2$

Some differences

- 3D vectors are points in pure quaternion subspace $q_v = (x, y, z, 0)$
- Applying quaternion to points $q_{v'} = q\, q_v\, q^\star$

Advantage of quaternion

- Interpolation works well in quaternion space (interpolation of points on sphere).

Rotation representation: 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

$$\begin{cases} \mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = -1 \\ \mathbf{ij} = -\mathbf{ji} = \mathbf{k} \\ \mathbf{jk} = -\mathbf{kj} = \mathbf{i} \\ \mathbf{ki} = -\mathbf{ik} = \mathbf{j} \\ \mathbf{ijk} = -1 \end{cases}$$

- Give the algebraic properties

Basics operations on quaternions

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

- Quaternion norm $\|q\| = \sqrt{q\,q^\star} = \sqrt{x^2 + y^2 + z^2 + w^2}$. 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 $q = (s, v)$
- 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

- $q_{v'} = \mathcal{R}_q(v) = q\, q_v\, q^\star$ 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 $2\ \mathrm{acos}(w)$.

Demonstration

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

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

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 $\mathcal{R}_{q_1\, q_2}(v) = \mathcal{R}_{q_1} \circ \mathcal{R}_{q_2}(v)$.

$\mathcal{R}_{q_1\, q_2}(v) = (q_1 q_2)\; v\; (q_1 q_2)^\star$

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

$\mathcal{R}_{q_1\, q_2}(v) = q_1\; (q_2\; v\; q_2^\star)\, q_1^\star$

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

Correspondance quaternion to rotation matrix

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

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

Demonstration

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

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

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

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

Summary - Correspondance quaternion / matrix-vector

Representation and Operations

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

Rotation to Quaternion

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

Quaternion to Rotation

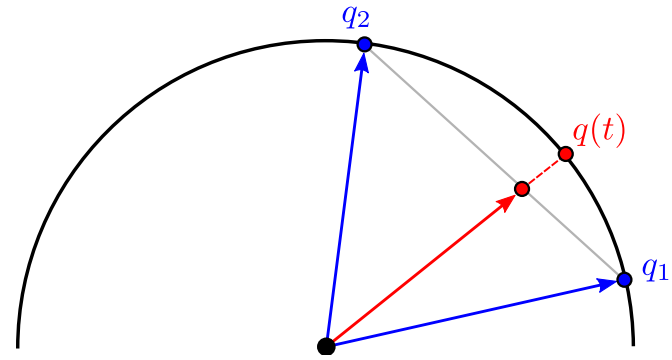
Unit quaternion $q = (x, y, z, w) \Rightarrow R = \begin{pmatrix} 1 - 2(y^2 + z^2) & 2(xy - wz) & 2(xz + wy) \\ 2(xy + wy) & 1 - 2(x^2 + z^2) & 2(yz - wx) \\ 2(xz - wy) & 2(yz + wx) & 1 - 2(x^2 + y^2) \end{pmatrix}$

Interpolation of Quaternion - LERP

Linear interpolation (with normalization) in quaternion space (**LERP**)

- Consider two rotations with corresponding quaternion q_1, q_2.
- The *linearly interpolated* quaternion at parameter $t \in [0, 1]$ is

$$q(t) = \frac{(1-t)\,q_1 + t\,q_2}{\|(1-t)\,q_1 + t\,q_2\|}$$



(+) Follows a shortest path (great circle on 4D sphere)

(+) Can be generalized to more general parametric curves (Splines, etc)

(-) Angular speed of orientation is not-constant

ex. extreme case of two opposite quaternions

Interpolation of Quaternion - SLERP

Spherical Linear interpolation (**SLERP**)

- Consider two rotations with corresponding quaternion q_1, q_2.
- The *spherical linear interpolated* quaternion at parameter $t \in [0, 1]$ is

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

Demonstration

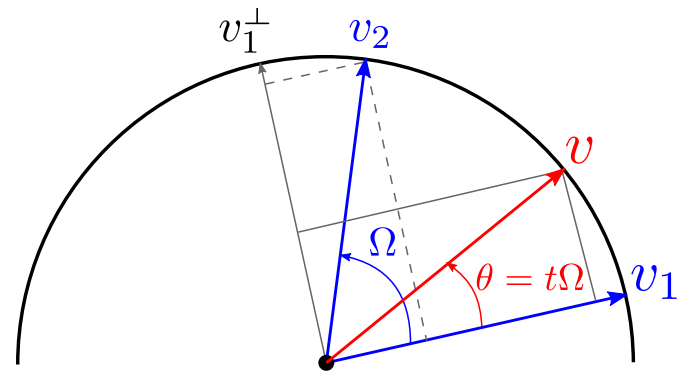
Consider

- Two unit vectors (arbitrary dimensions) v_1, v_2.
- Ω: angle b/w v_1 and v_2
- The interpolated vector v at angle $\theta = \Omega t, t \in [0, 1]$.

$v = v_1 \cos(\theta) + v_1^{\perp} \sin(\theta)$, and $v_1^{\perp} = \frac{v_2 - \cos(\Omega)\, v_1}{\sin(\Omega)}$

$\Rightarrow v = \left(\frac{\sin(\Omega)\, \cos(\theta) - \cos(\Omega)\, \sin(\theta)}{\sin(\Omega)} \right) v_1 + \frac{\sin(\theta)}{\sin(\Omega)} v_2$

$\Rightarrow v = \frac{\sin(\Omega - \theta)}{\sin(\Omega)} v_1 + \frac{\sin(\theta)}{\sin(\Omega)}$

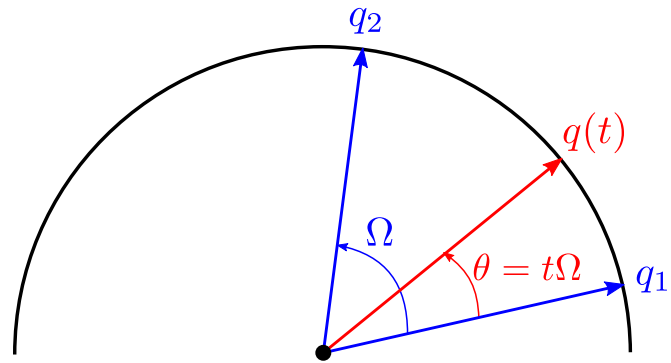


Interpolation of Quaternion - SLERP

Spherical Linear interpolation (**SLERP**)

- Consider two rotations with corresponding quaternion q_1, q_2.
- The *spherical linear interpolated* quaternion at parameter $t \in [0, 1]$ is

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



(+) Follows a shortest path (great circle on 4D sphere)

(+) Constant angular speed

(-) Cannot be generalized to more general curves.

Cannot interpolates between more than two quaternions

Care with quaternion negation

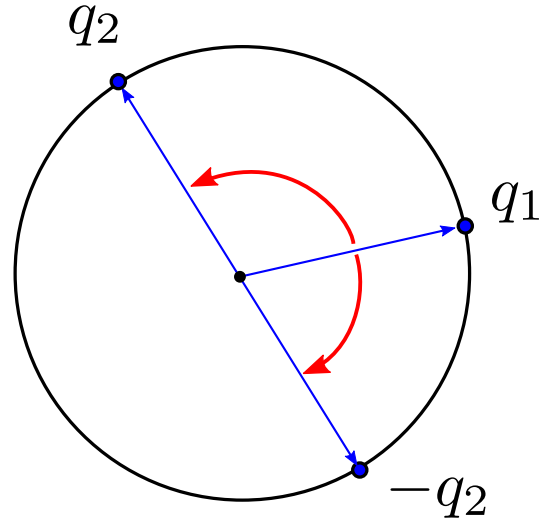
$+q$ and $-q$ correspond to the same rotation ($n \to -n, \theta \to 2\pi - \theta$)

Warning $-q$ ***does not*** *corresponds to the rotation matrix $-R$.*

But to a **different path** when interpolated in the 4D quaternion space.

$\Rightarrow$ path $q_1 \to -q_2$ is shorter than $q_1 \to q_2$ when $q_1 \cdot q_2 < 0$.

In practice we check for the shorter path before applying SLERP.



Algorithm

```
if( dot(q1,q2)<0 )
    q2 = -q2
q(t) = SLERP(q1,q2,t)
```

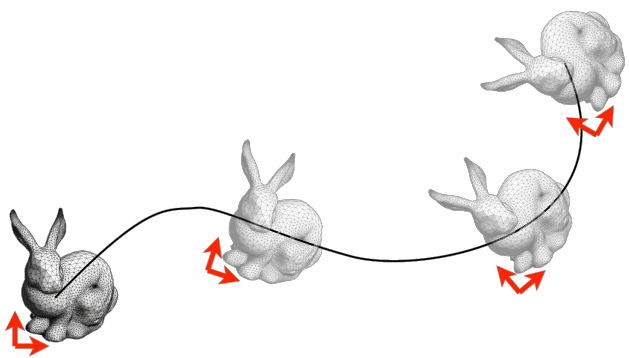
Interpolating rigid motion

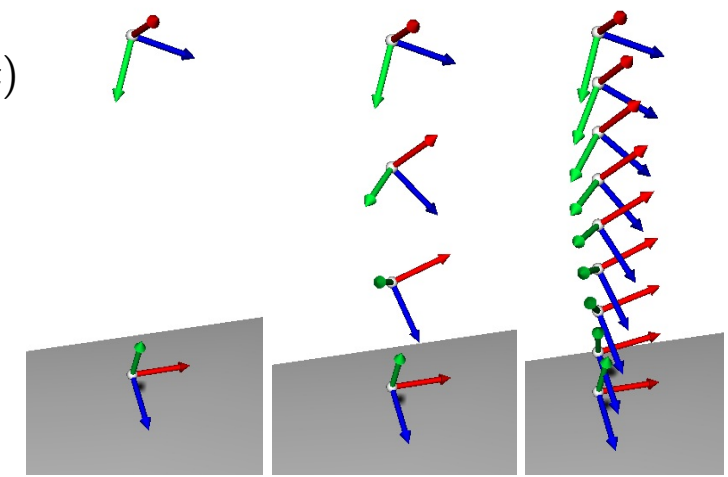
3D objects have orientation r and position p.

$\Rightarrow$ 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 $p(t) = (1 - t)\, p_1 + t\, p_2$
- Interpolate rotation with SLERP on quaternions
  - Convert $(r_1, r_2) \rightarrow (q_1, q_2)$
  - Compute $q(t) = SLERP(q_1, q_2, t)$
  - Convert back $q(t) \rightarrow r(t)$





Interpolating rigid motion - Comparison

Matrix interpolation

Euler angle interpolation

Quaternion interpolation

Handling affine transformation : Polar Decomposition

Key-pose can be given as matrix M containing rotation, but also scaling and shearing.

$\Rightarrow$ Cannot convert M to quaternion (not a rotation).

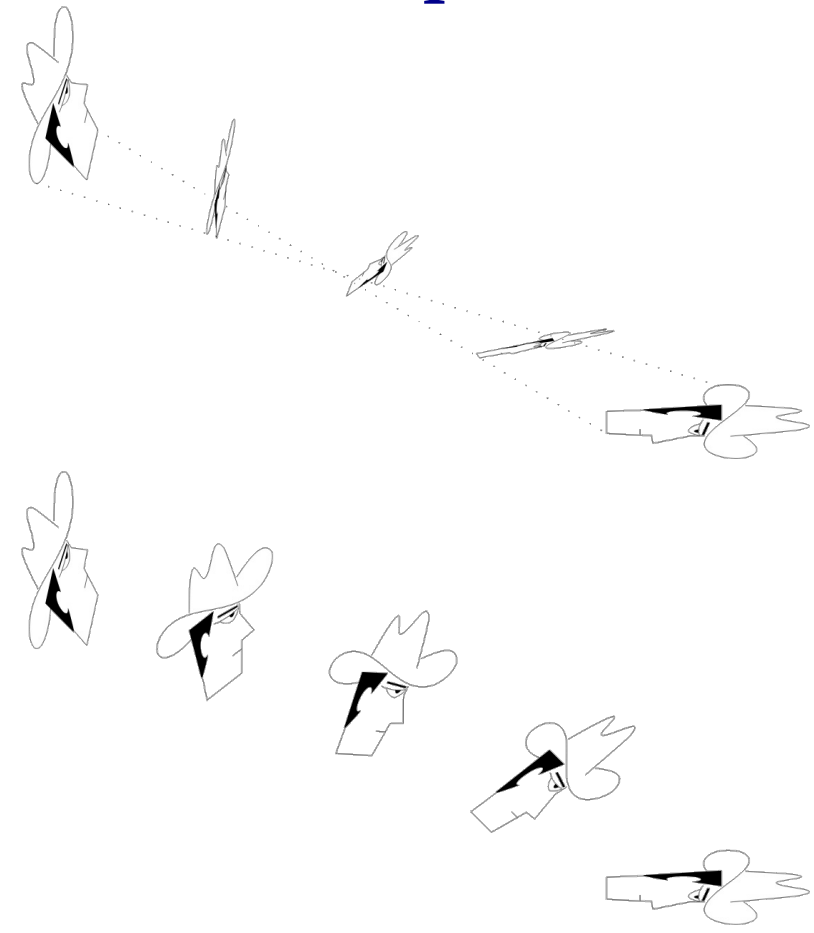
$\Rightarrow$ Separate *rotation* component from shearing and scaling component using **polar decomposition**.

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

- Polar decomposition: $\mathrm{M} = \mathrm{R}\,\mathrm{D}$
  - R: Rotation matrix
  - D: Positive semi-definite matrix

- Polar decomposition is obtained from SVD
  $\mathrm{SVD}(\mathrm{M}) = W\Sigma V^T$ with $\mathrm{R} = \mathrm{W}\mathrm{V}^T$, $\mathrm{D} = \mathrm{V}\Sigma\mathrm{V}^T$

- Numerically, R can be computed using the following iterative scheme
  $R_0 = \mathrm{M}$, $R_{i+1} = 0.5\,(R_i + R_i^{-T})$



Handling affine transformation

Algorithm to interpolate between two general 4×4 matrix 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 $M \to q$ 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 $q(t) = \mathrm{SLERP}(q_1, q_2, t)$
6. Convert back to matrix $q(t) \to R(t)$
7. Compute final matrix $M(t) = R(t)\,D(t)$ with translation $p(t)$.