COORDINATE SYSTEM

FRAME OF REFERENCE

Right hand <> left hand

$$\begin{cases}\theta_x^L = -\theta_x^R, d_x^L = d_x^R \\ \theta_y^L = -\theta_y^R, d_y^L = d_y^R\text{, plug in to get } R^L, d^L \\ \theta_z^L = \theta_z^R, d_z^L = -d_z^R\end{cases}$$

ROTATION

Rotation Matrix

Orthonormal, invertible, $R^T = R^{-1}$

Euler Angles

With respect to local frame, post multiplication

Euler Angles <> Rotation Matrix

See appendix 1 for more

Six configurations of Euler angles

As for the inverse, take y-x-z for example:

For generally case ($\cos\theta_x \neq 0$), there are 2 solutions:

$$\begin{cases}\theta_x = atan2(-r_{23}, \sqrt{1-r_{23}^2}) \\ \theta_y = atan2(r_{12}, r_{33}) \\ \theta_z = atan2(r_{21}, r_{22})\end{cases}, \begin{cases}\theta_x = atan2(-r_{23}, -\sqrt{1-r_{23}^2}) \\ \theta_y = atan2(-r_{12}, -r_{33}) \\ \theta_z = atan2(-r_{21}, -r_{22})\end{cases}$$

Axis-Angle

Axis k and angle θ are defined in the frame before rotation. Though k has the same coordinates in both frames.

Axis-Angle <> Rotation Matrix

$$\left[\begin{array}{c|c|c} k_x^2 v_\theta + c_\theta & k_x k_y v_\theta - k_z s_\theta & k_x k_z v_\theta + k_y s_\theta \\ k_x k_y v_\theta + k_z s_\theta & k_y^2 v_\theta + c_\theta & k_y k_z v_\theta - k_x s_\theta \\ k_x k_z v_\theta - k_y s_\theta & k_y k_z v_\theta + k_x s_\theta & k_z^2 v_\theta + c_\theta \end{array}\right]$$

where $v_\theta = 1 - \cos\theta$, and the inverse is:

$$\theta = \cos^{-1}\left(\frac{r_{11} + r_{22} + r_{33} - 1}{2}\right), k = \frac{1}{2\sin\theta}\begin{bmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12}\end{bmatrix}$$

Quaternions

Quaternion <> Axis Angle

$axis: \hat{n} = [n_x, n_y, n_z]^T, angle: \theta$

$Q = (\cos\left(\frac{\theta}{2}\right), n_x \sin\left(\frac{\theta}{2}\right), n_y \sin(\frac{\theta}{2}), n_z \sin\left(\frac{\theta}{2}\right))$

Quaternion <> Rotation Matrix

1 - 2qy^2 - 2qz^2	2qxqy - 2qzqw	2qxqz + 2qyqw
2qxqy + 2qzqw	1 - 2qx^2 - 2qz^2	2qyqz - 2qxqw
2qxqz - 2qyqw	2qyqz + 2qxqw	1 - 2qx^2 - 2qy^2

```
trace = m00 + m11 + m22
if trace>0
    S = sqrt(tr+1.0) * 2; // S=4*qw
    qw = 0.25 * S;
    qx = (m21 - m12) / S;
    qy = (m02 - m20) / S;
    qz = (m10 - m01) / S;
else if m00 > m11 and m00 > m22
    S = sqrt(1.0 + m00 - m11 - m22) * 2; // S=4*qx
    qw = (m21 - m12) / S;
    qx = 0.25 * S;
    qy = (m01 + m10) / S;
    qz = (m02 + m20) / S;
else if m11 > m22
    S = sqrt(1.0 + m11 - m00 - m22) * 2; // S=4*qy
    qw = (m02 - m20) / S;
    qx = (m01 + m10) / S;
    qy = 0.25 * S;
    qz = (m12 + m21) / S;
else
    S = sqrt(1.0 + m22 - m00 - m11) * 2; // S=4*qz
    qw = (m10 - m01) / S;
    qx = (m02 + m20) / S;
    qy = (m12 + m21) / S;
    qz = 0.25 * S;
```

More on Quaternions

Identity quaternion: $Q_I = (1,0,0,0)$

Conjugate quaternion for Q: $Q^* = (q_0, -q_1, -q_2, -q_3)$

Multiplying two quaternions:

$X = (x_0, \vec{x}), Y = (y_0, \vec{y})$

$XY = x_0 y_0 - \vec{x}^T \vec{y} + x_0 \vec{y} + y_0 \vec{x} + \vec{x} \times \vec{y}$

Apply a unit quaternion's rotation to a vector:

Vector $\vec{v} = (v_x, v_y, v_z)$ in quaternions: $Q_v = (0, v_x, v_y, v_z)$; Rotated $Q_{v'} = Q Q_v Q^*$

INTERPOLATION

GENERAL PROBLEM

Local fit – splines

Global fit – e.g. regression

Explicit vs implicit vs parametric

Polynomials

Represent using monomials

$f(u) = (x(u), y(u), z(u))^T = a_0 + a_1 u + \cdots + a_n u^n$

degree, coefficients

Represent using basis functions

$f(u) = \sum_{i=0}^{n} b_i B_i^n(u)$

Local fit Interpolation Problem – General Solution

Input: a set of key point $(p_0, t_0), \ldots (p_{m-1}, t_{m-1})$

Output: interpolated spline

Step1: compute control points for each segment

Step2: compute curves using control points

BEZIER CURVE

Using Bernstein polynomials as basis functions

$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i}$

Derivative of Bezier curves:

$\frac{df(u)}{du} = n \sum_{i=0}^{n-1} (b_{i+1} - b_i) B_i^{n-1}(u)$

Cubic Bezier curve: n=3

Matrix Form

Rewrite the Bezier curve function as:

$f(u) = \sum_{i=0}^{n} b_i B_i^n(u) = b_0 + 3(b_1 - b_0)u + 3(b_2 - 2b_1 + b_0)u^2 + (b_3 - 3b_2 + 3b_1 - b_0)u^3 =$

$$[b_0\ b_1\ b_2\ b_3]\begin{bmatrix} 1 & -3 & 3 & -1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix} 1 \\ u \\ u^2 \\ u^3\end{bmatrix}$$

DE CASTELJAU ALGORITHM

Take cubic Bezier curve for example:
Given four coefficient vectors/control points: b_0, b_1, b_2, b_3 and u

$b_0^1 = LERP(b_0, b_1, u), b_1^1 = LERP(b_1, b_2, u), b_2^1 = LERP(b_2, b_3, u)$

$b_0^2 = LERP(b_0^1, b_1^1, u), b_1^2 = LERP(b_1^1, b_2^1, u)$

$b_0^3 = LERP(b_0^2, b_1^2, u)$

SPLINE

A spline is a curve comprised of a collection of piece-wise polynomials of arbitrary degree tied together at knot points with certain continuity conditions.
Continuity: 0, 1, 2 …

CATMUL-ROM SPLINE

Each segment is a Bezier curve

$f_j(t) = \sum_{i=0}^{n} b_{ji} B_i^n(u)$, where $u = \frac{t-t_j}{t_{j+1}-t_j}$

Usually, collect all the control points into a n*m length vector, while n is the degree of the curve, m is the number of curve segments.
Catmul-Rom Spline is C1 continuous
Generally, we have $f'(t_i) = (p_{i+1} - p_{i-1})/(t_{i+1} - t_{i-1})$
To compute control points for Catmul-Rom spline, for segment i, we have:

$$\begin{cases} b_0 = p_i \\ b_3 = p_{i+1} \\ b_1 = b_0 + s_0/3 \\ b_2 = b_3 - s_1/3 \end{cases} \quad \begin{cases} s_0 = (p_{i+1} - p_{i-1})/(t_{i+1} - t_{i-1}) \\ s_1 = (p_{i+2} - p_i)/(t_{i+2} - t_i) \end{cases}$$

Special case for endpoints
    Option 1: $s_0 = p_1 - p_0$
    Option 2: introduce phantom point p_{-1} and p_{m+1}
We can use the matrix form to convert between curve types (Bezier curve and monomial curve)

HERMITE CURVE

$h(u) = p_0 H_0^3(u) + p_1 H_3^3(u) + p_0' H_1^3(u) + p_1' H_2^3(u)$
Let $h(u) = f(u)$, we have:

$$H_0^3 = B_0^3 + B_1^3, H_3^3 = B_2^3 + B_3^3, H_1^3 = B_1^3/3, H_2^3 = B_2^3/3$$

$$h(u) = (1 - 3u^2 + 2u^3)p_0 + (3u^2 - 2u^3)p_1 + (u - 2u^2 + u^3)p_0' + (-u^2 + u^3)p_1'$$

Matrix Form

$$h(u) = [p_0\ p_1\ p_0'\ p_1'] \begin{bmatrix} 1 & 0 & -3 & 2 \\ 0 & 0 & 3 & -2 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ u \\ u^2 \\ u^3 \end{bmatrix}$$

HERMITE SPLINE

C2 continuity - $p_j' + 4p_{j+1}' + p_{j+2}' = -3p_j + 3p_{j+2}$

Special case for endpoints
    Option 1: clamped, assign value for endpoint slope
    Option 2: natural, let second order derivative at endpoints be 0, so that:

$$2p_0' + p_1' = -3p_0 + 3p_1, p_{n-1}' + 2p_n' = 3p_n - 3p_{n-1}$$

B-SPLINE

B-spline is C2 continuous
See handout 'B-spline Construction Summary' for detail.

Catmul-Rom Spline vs B-spline

A Catmul-Rom spline is constructed from individual Bezier curves, while a B-spline uses overlapping basis functions. Both provide local function approximation. Catmul-Rom spline is easy to compute and computationally efficient, but has discontinuous second derivatives at the knot points, while B-spline provides an optimal interpolation to the control points and it has continuous second derivatives at the knot points, but is more computationally expensive.

QUATERNION INTERPOLATION

$Slerp(a, b, u) = a(\sin(1-u)\Omega/\sin\Omega) + b(\sin u\Omega/\sin\Omega)$, where $\cos\Omega = a \cdot b$
Why use quaternion?
Euler angles: discontinuous
Rotation matrix: has to be orthonormal
See handout 'Cubic Quaternion Spline Summary' for detail

2D SURFACES

2D polynomials

$$f(u,v) = \left(x(u,v), y(u,v), z(u,v)\right)^T = \sum_{i=0}^{n} a_i u^i \sum_{i=0}^{n} b_i v^i = \sum_{i,j} c_{ij} u^i v^j$$

Manifold

A lower dimensional surface embedded in a higher dimensional space that is a local deformation of Euclidean space without any tears of intersections
Local coordinate system on surface

$\frac{\partial f}{\partial v}, \frac{\partial f}{\partial u}$ and $\hat{n} = \frac{\frac{\partial f}{\partial u} \times \frac{\partial f}{\partial v}}{\left\|\frac{\partial f}{\partial u} \times \frac{\partial f}{\partial v}\right\|}$

2D Interpolation

Given 4 points $p_{00}, p_{10}, p_{11}, p_{01}$, find $p = f(u,v)$ for $u, v \in [0,1]$
2D linear interpolation – start with Lerp in terms of v (for u=0 and u=1), then interpolate the two values in terms of u

$$f(u,v) = p_{00}(1-v)(1-u) + p_{01}v(1-u) + p_{10}(1-v)u + p_{11}vu = \sum_{i=0}^{1}\sum_{j=0}^{1} p_{ij} B_i^1(u) B_j^1(v)$$

(Cubic) Bezier Surface Interpolation – start with Bezier curves in terms of v, then then interpolate in terms of u

$f(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} b_{ij} B_i^3(u) B_j^3(v)$ (16 control pts)

Properties:
~Endpoint interpolation $f(i,j) = b_{ij}$
~Cotangents ~Twist

$\frac{\partial f}{\partial u}(0,0) = 3(b_{10} - b_{00})$	$\frac{\partial f}{\partial v}(0,0) = 3(b_{01} - b_{00})$	$\frac{\partial^2 f}{\partial u \partial v}(0,0) = 9(b_{00} - b_{01} - b_{10} + b_{11})$
$\frac{\partial f}{\partial u}(1,0) = 3(b_{30} - b_{20})$	$\frac{\partial f}{\partial v}(0,1) = 3(b_{03} - b_{02})$	$\frac{\partial^2 f}{\partial u \partial v}(0,1) = 9(b_{03} - b_{02} - b_{31} + b_{12})$
$\frac{\partial f}{\partial u}(0,1) = 3(b_{13} - b_{03})$	$\frac{\partial f}{\partial v}(1,0) = 3(b_{31} - b_{30})$	$\frac{\partial^2 f}{\partial u \partial v}(1,0) = 9(b_{30} - b_{20} - b_{31} + b_{21})$
$\frac{\partial f}{\partial u}(1,1) = 3(b_{33} - b_{23})$	$\frac{\partial f}{\partial v}(1,1) = 3(b_{33} - b_{32})$	$\frac{\partial^2 f}{\partial u \partial v}(1,1) = 9(b_{33} - b_{32} - b_{23} + b_{22})$

Spline Cages

~Use 2D parametric surfaces (e.g. Bezier, B-spline)
~Control points used to deform model (surface path subdivision for local control
~Convert parametric form to polygons

3D: FREE FORM DEFORMATION

Cubic Bezier Volume Patch

$f(u,v,w) = \sum_{i=0}^{3}\sum_{j=0}^{3}\sum_{k=0}^{3} b_{ijk} B_i^3(u) B_j^3(v) B_k^3(w)$

FFD

~Set up lattice

~Polygon embedding: express local coordinates of vertex point p in a lattice $p = p_0 + u\hat{u} + v\hat{v} + w\hat{w}$

~Initialize control points b_{ijk} to corners of the cube

~Deform lattice by moving the control points of the lattice

KINEMATICS

FORWARD KINEMATICS

See 'Forward Kinematics Recap'

JACOBIAN MATRIX

See Jacobian Matrix Computation

EULER ANGLE RATE TO ANGULAR VELOCITY

See 'Euler Angle Rate to Angular Velocity Conversion' and Appendix 2

INVERSE KINEMATIC

Before everything: create IK chain

Pseudo Inverse Method

Theory

$$\dot{x} = J\dot{\theta} \implies \dot{\theta} = J^{\dagger}\dot{x}$$

Left pseudo inverse: more DoF in x than θ

$$J^{\dagger} = (J^T J)^{-1} J^T$$

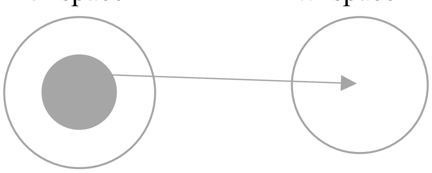
Right pseudo inverse: otherwise

$$J^{\dagger} = J^T (J J^T)^{-1}$$

Integrate $\Delta\theta$ to get new joint angle

- Null space: $0 = J\,\dot{\theta} \implies \dot{\theta} = \left(J^{\dagger}J - I\right)c$

$\dot{\theta}$ space $\dot{x}$ space



- Spring like behavior: $\dot{\theta} = k(\theta - \theta_0)$

$$\dot{\theta} = J^{\dagger}\dot{x} + \left(J^{\dagger}J - I\right)(\theta - \theta_0)$$

- Damped Pseudo Inverse:

  To avoid singular case in $(JJ^T)^{-1}$, let

$$J^{\dagger} = J^T(\lambda I + JJ^T)^{-1}, \quad \lambda \ll 1$$

Implementation Steps

1. Find current and desired position and orientation of the end joint
2. Path planning, design a trajectory and break into small segments (check 'arc length parameterization')
3. Compute current Jacobian matrix, we can use $J_i = B_i L_i$ (check *'Jacobian Matrix Computation'* for detail)
4. For each small Δx, compute $\Delta\theta$ using $\Delta\theta = J^{\dagger}\Delta x$

Limb IK

Notation

$p_{end}, p_{mid}, p_{base}$ - three joint position

p_{target} - target position

l_1, l_2 - length of the upper and lower 'arm'

Implementation Steps

$e \leftarrow p_{target} - p_{end}$

$r \leftarrow p_{end} - p_{base}$

$r_d \leftarrow p_{target} - p_{base}$

$\theta_{mid} \leftarrow \mathrm{acos}(l_1^2 + l_2^2 - \|r_d\|^2)/2l_1 l_2$ //mid joint angle

$\Delta\theta_{base} \leftarrow \mathrm{acos}(\|r\|^2 + \|r_d\|^2 - \|e\|^2)/2\|r\|\,\|r_d\|$

Update the transformations on mid joint and base joint, pay attention to local and global frame

CCD

Implementation Steps

Repeat:

For each joint p_{curr} in IK chain (distal to proximal):

//compute axis and angle

$e \leftarrow p_{end} - p_{base}$

$r_{c2e} \leftarrow p_{end} - p_{curr}$

$angle \leftarrow \|r_{c2e} \times e\| / (\|r_{c2e}\|^2 + \|r_{c2e}\|\|e\|)$

(when angle is small, we can omit the 'atan')

$axis \leftarrow r_{c2e} \times e / \|r_{c2e} \times e\|$

convert axis to local coordinate

$\Delta\theta_{curr} = c_{curr} * angle$

$\Delta R = AxisAngle2Rot(axis, \Delta\theta_{curr})$

Update joint transformation

BODY ANIMATION

BASICS

- Animation = pose(time)
- Collection of motion curves (6DOF for a single joint)
- Motion curve representation (e.g. cubic splines)

MAIN APPROACHES

- Key frame
- Motion Capture
- Procedural
- Physically-based

MOTION CAPTURE

With markers/sensors placed on subject, record motion from real world objects, used recorded motion to animate virtual objects

1. Track motions of actor body and/ or face
2. Convert to skeleton description and joint angle data
3. Use skeleton and joint angle data to animate characters

Different types of Motion Capture Technology

- Optical: passive marker, active marker, marker-less (only track position of markers)
- Magnetic
- Inertial Systems
- Exoskeleton

File format

BVH: Joint-based skeleton

AMC: bone-based skeleton

NOTATIONS

- Rigging system (FK and IK)
- Pose space

  A point in ~: $\Theta = [\theta_1, \dots, \theta_n]^T$

  An animation can be thought of as a point moving through pose space, or alternately as a curve or spline in pose space : $\Theta = \Theta(t)$
- Channels: 1-dimensional curves (one for each DOF)

  $\theta_i = \theta_i(t)$

  - Can be a joint angle or arbitrary parameter value

- Represents pre-recorded data
- Channels can be discontinuous in value, but not in time

- Array of channels (flexible, less memory) vs. array of poses (faster) (numDoFs*numFrames)

MOTION EDITING

Applications

Interactive Posing

Adding constrains

Optimizing motion over a sequence of poses

Main Editing Techniques

- Time warping

$$m'(t) = m(wt)$$

  $w > 1$ – speed up

  $0 < w < 1$ - slow down

- Blending

  Frame level: $m(t) = (1-\alpha)m_1(t) + \alpha m_2(t)$

  Ctrl pts level: $c_j = (1-\alpha)c_{1j} + \alpha c_{2j}$ (when both motion curves have same knot points)

  Knot pts level: $t_j = (1-\alpha)t_{1j} + \alpha t_{2j}$ (when both motion curves have knot points at different points in time)

- Layering

$$m(t) = (1-\alpha(t))m_1(t) + \alpha(t)m_2(t)$$

ARC LENGTH PARAMETERIZATION

Motivation

Given a 3D path representation $p = f(u) = [x(u)\ \ y(u)\ z(u)]^T$, the x, y and z path coordinates are functions of independent parameter; We want to move along the path in constant speed (or following a given time-velocity curve), but $s = g(u)$ is usually not linear.

Solution:

Use $u = g^{-1}(s)$ to choose Δu to get uniform Δs

Non-analytical; Need a list of u-s correspondence table for approximation (see slide for detail)

BEHAVIOR ANIMATION

Contents in this section are covered in 'Lecture Notes on Interactive Animation and Control Architectures'. See notes for details

OPTIMIZATION APPROACHES

Contents in this section are covered in slide 'Animation through Optimization'. See notes for details

LEAST SQUARES OPTIMIZATION

Example: fit a set of points to a cubic curve

$$\begin{bmatrix} 1 & u_1 & u_1^2 & u_1^3 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & u_m & u_m^2 & u_m^3 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}$$

A x = b

Let $e = Ax - b$

The problem transforms to a optimization problem:

$\min_c \frac{1}{2}\|e\|_2^2 = \frac{1}{2}(Ax-b)^T(Ax-b)$

take derivative, we have

$A^T(Ax-b) = 0 \implies x = (A^TA)^{-1}A^Tb$ (MLE solution)

GRADIENT DESCENT

Sequential quadratic Programming (SQP) /Newton's method or Newton Lagrangian

2 steps (See slide for more)

PCA

Eigenvector and Eigenvalue

$$Ae = \lambda e \ \Rightarrow (A - \lambda I)e = 0$$

calculate eigenvalue $\det(A - \lambda I) = 0$

PCA Algorithm

Given a dataset X, assume X is centered, let $A = XX^T$

Let $E = [e_1\ e_2\ \dots e_n]$ be the set of all eigenvectors of A, i.e. $AE = E\Lambda$, where $\Lambda = diag([\lambda_1\ \lambda_2 \dots \lambda_n])$, therefore:

$$A = E\Lambda E^{-1} = E\Lambda E^T$$

Let $Y = PX$, where $P = E^T$, the covariance of Y is

$$Cov_Y = \frac{1}{m}PXX^TP = \frac{1}{m}PAP = \frac{1}{m}E^TE\Lambda E^TE = \frac{1}{m}\Lambda$$

In many cases, the first k eigenvectors account for most of the variance, we can use a reduced form

$$\tilde{P}^T = [e_1\ e_2\ \dots e_k]$$

and $\tilde{y}_i = \tilde{P}x_i$, where $\tilde{y}_i$ is $k \times 1$, x_i is $n \times 1$

PCA Algorithm

Input: $D\{\mathrm{x}^1, \dots, \mathrm{x}^n\}$

Output: principle components $z^1, \dots, z^k$

Compute mean and covariance of data $\bar{\mathrm{x}}$, Σ

Find k eigenvectors of Σ with largest eigenvalues $\mathrm{u}_1, \dots, \mathrm{u}_k$ (loadings)

Get principle components: $z^i = ((x^i - \bar{x})^T u_1, \dots, (x^i - \bar{x})^T u_k)$

EigenFace

Implementation steps for calculate Eigenfaces

1. Vectorize and centerize image
2. Calculate covariance matrix
3. Obtain eigenvectors and eigenvalues
4. Choose 16 largest eigenvectors as loadings
5. Re-represent faces using principle components

SPACETIME CONSTRAINTS

Solve for the object motion by varying the force over the entire time

Problem Statement

Governing Equation (dynamics)

Boundary Conditions (position constraint)

Objective function (minimize energy consumption)

Implementation process

- Set up motion equations
- Define objective function
- Evaluate the derivatives
- Pass into SQP solver

Pros: 'optimal' motion sequence; don't have to understand control ***Cons:*** Not real time control (lags); local minima; hard to define appropriate objective functions

Applications: optimal transitions, optimal gait and form, sybthesis of motion, motion retargeting (see slide for more)

DYNAMICS

NOTATION					
position	x	velocity	v	acceleration	a
mass	m	momentum	p (mv)	force	f
Moment of Inertia	I	Angular momentum	L	torque	τ

LINEAR DYNAMICS (TRIVIAL)

FORCE TYPE

Gravity $\mathrm{f} = mg$

Springs $\mathrm{f}_s = -K_s\mathrm{x}$

Dampers $\mathrm{f}_d = -K_d\mathrm{v}$

Friction $\mathrm{f} = \mathrm{f_n}\mu_d$ (dynamic friction) $\mathrm{f} \le \mathrm{f_n}\mu_s$

Aerodynamic Drag $\mathrm{f_{drag}} = -0.5\rho\|\mathrm{v}\|^2 c_d A\hat{\mathrm{v}}$

Force field $\mathrm{f_{field}} = \mathrm{f(x)}$

PARTICLE SYSTEM DYNAMICS

Newtonian approach

Simulation Steps:

1. Compute all forces acting on each particle in current configuration
2. Compute the resulting acceleration for each particle
3. Integrate over some small time step to update state
4. Repeat 1~3

Particle systems

2nd Order ODE

$$1^{st} order: a = \ddot{\mathrm{x}} = \frac{f(\mathrm{x},\dot{\mathrm{x}},t)}{m}, 2^{nd} order: \begin{cases} \dot{\mathrm{x}} = \mathrm{v} \\ \dot{\mathrm{v}} = f(\mathrm{x},\dot{\mathrm{x}},t)/m \end{cases}$$

Phase Space

$$State\ \ s = \begin{bmatrix} x \\ v \end{bmatrix}, \dot{s} = \begin{bmatrix} \dot{x} \\ \dot{v} \end{bmatrix}$$

$$State\ dynamics\ \begin{bmatrix} \dot{x} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} \dot{v} \\ f/m \end{bmatrix}\ or\ \dot{s} = f(s,t)$$

See more in 'Physically Based Modeling: Particle System Dynamics'

Solve Particle System Dynamics

Derive Evaluation Loop

- Clear forces
- Calculate forces
- Gather (computer dynamics)
- Update state

Rendering

Points, lines, sprites, geometry, …

Rotational dynamics for particles

$$L = r \times p$$
$$\tau = dL/dt = r \times f$$

velocity vs angular velocity: $v = dr/dt = \omega \times r + v_r$

centripetal acc.: $a_{cen} = dv/dt = \dot{\omega} \times r + \omega \times (\omega \times r)$

Derivative of Rotation Matrix $dR/dt = \dot{R} = \omega \times R$

Rotational Inertia (assume constant r and m)

$$\begin{aligned} L = r \times p = r \times (mv) = mr \times v = mr \times (\omega \times r) \\ = -mr \times r \times \omega = -m\Gamma \cdot \Gamma \cdot \omega = I \cdot \omega \end{aligned}$$

where Γ is the skew symmetric matrix of r, $I = -m\Gamma \cdot \Gamma$

$$I = -m\begin{bmatrix} -r_y^2 - r_z^2 & r_x r_y & r_x r_z \\ r_x r_y & -r_x^2 - r_z^2 & r_y r_z \\ r_x r_z & r_y r_z & -r_x^2 - r_y^2 \end{bmatrix}$$

System of Particles

$m_{total} = \sum m_i$

$x_{cm} = \sum m_i x_i / \sum m_i$

$p_{cm} = \sum p_i = \sum m_i v_i$

$$\begin{aligned} v_{cm} &= dx_{cm}/dt = (1/\textstyle\sum m_i)\, d(\textstyle\sum m_i x_i)/dt \\ &= \textstyle\sum m_i v_i / \textstyle\sum m_i = p_{cm}/m_{total} \end{aligned}$$

$p_{cm} = m_{total} v_{cm}$

$f_{cm} = dp_{cm}/dt = \sum f_i$

$L_{cm} = \sum r_i \times p_i = \sum (x_i - x_{cm}) \times p_i$

$\tau_{cm} = dL_{cm}/dt = \sum r_i \times f_i$

RIGID BODY DYNAMICS

$x_{cm} = \int \rho x d\Omega / \int \rho d\Omega$

$$I = \begin{bmatrix} \int \rho(r_y^2 + r_z^2) d\Omega & -\int \rho r_x r_y d\Omega & -\int \rho r_x r_z d\Omega \\ -\int \rho r_x r_y d\Omega & \int \rho(r_x^2 + r_z^2) d\Omega & -\int \rho r_y r_z d\Omega \\ -\int \rho r_x r_z d\Omega & -\int \rho r_y r_z d\Omega & \int \rho(r_x^2 + r_y^2) d\Omega \end{bmatrix}$$

$L = I \cdot \omega$

$\tau = dL/dt = \omega \times I \cdot \omega + I \cdot \dot{\omega}$

Equation of motions

$a = f_m/m$

$\dot{\omega} = I^{-1}(\tau - \omega \times I \cdot \omega)$

Rigid body simulation

- Translation
  Velocity & position update $s(t_{k+1}) = s(t_k) + \dot{s}(t_k)\Delta t$
- Orientation
  Angular velocity update $\omega(t_{k+1}) = \omega(t_k) + \dot{\omega}(t_k)\Delta t$
  Orientation update:
  $Axis = \frac{\omega(t_k)}{\|\omega(t_k)\|}, Angle = \|\omega(t_k)\|\Delta t$
  $R(t_{k+1}) = R(t_k)\Delta R$
  where $\Delta R = AxisAngle2Rot(Axis, Angle)$

SUMMARY: FORWARD AND INVERSE DYNAMICS

Forward dynamics Compute motion resulting from applied forces and torques

$\ddot{x} = f/m$

$\dot{\omega} = I^{-1}(\tau - \omega \times I \cdot \omega)$

Inverse dynamics Compute forces and torques required to generate desired motion

$f = m\ddot{x}$

$\tau = \omega \times I \cdot \omega + I \cdot \dot{\omega}$

DIFFERENTIAL EQUATIONS

General form: $\dot{x} = f(x,t)$

Vector field $\dot{x} = f(x)$

Taylor expansion

$$x_{t+\Delta t} = x_t + \dot{x}_t \Delta t + \ddot{x}_t \Delta t/2 + \dddot{x}_t \Delta t/6 + \cdots$$

Euler's Method (1st order) $x_{t_{k+1}} = x_{t_k} + \dot{x}_{t_k}\Delta t$

Drawbacks: drift off, error accumulate by time, oscillate at large step

Runge Kutta Method (2nd and above)

- Use weighted average of slopes across interval
- Order determines approximation error

2nd order RK method

- $x^p_{t_{k+1}} = x_{t_k} + \dot{x_{t_k}}\Delta t$

- $\dot{x}^p_{t_{k+1}} = f(x^p_{t_{k+1}})$
- $x_{t_{k+1}} = x_{t_k} + (\dot{x}^p_{t_{k+1}} + \dot{x}_{t_k})\Delta t/2$

4th order RK method – better fit, expensive

$x_{t_{k+1}} = x_{t_k} + (d_1 + 2d_2 + 2d_3 + d_4)\Delta t/6$

where

$d_1 = f(t_k, x(t_k))$

$d_2 = f(t_{k+1} + \Delta t/2, x(t_k) + d_1/2)$

$d_3 = f(t_{k+1} + \Delta t/2, x(t_k) + d_2/2)$

$d_4 = f(t_{k+1} + \Delta t, x(t_k) + d_3)$

Implicit Method

Explicit Euler method add energy in the form of errors, which is bad for stiff systems (may result in instability)

Backward Euler Method $x_{t_{k+1}} = x_{t_k} + \dot{x}_{t_{k+1}}\Delta t$

Converges much slower

How to compute $\dot{x}_{t_{k+1}}$:

- derive from formula
- predictor-corrector (explicit method + plug in, e.g. RK2)
- linear system

$$x_{t_{k+1}} = x_{t_k} + \dot{x}_{t_{k+1}}\Delta t = x_{t_k} + \Delta x_{t_k}$$
$$\dot{x}_{t_{k+1}} = f(x_{t_{k+1}}) = f(x_{tk} + \Delta x_{t_k})$$
$$x_{t_k} + \Delta x_{t_k} = x_{t_k} + \Delta t\, f(x_{tk} + \Delta x_{t_k})$$
$$\Delta x_{t_k} \approx \Delta t(f(x_{t_k} + (\partial f/\partial x_{t_k})\Delta x_{t_k})$$
$$\Delta x_{t_k} \approx \left(\frac{1}{\Delta t} I - \frac{\partial f}{\partial x_{t_k}}\right)^{-1} f(x_{t_k})$$

Multi-step Methods

Use values from previous time steps to calculate next one Anchors approximation with more accurate data

Adams Bashforth

$x_{t_{k+1}} = x_{t_k} + \dot{x}_{t_k}\Delta t + \ddot{x}_{t_k}\Delta t/2$

$\ddot{x}_{t_k} \approx (\dot{x}_{t_k} - \dot{x}_{t_{k-1}})/\Delta t$

$x_{t_{k+1}} + (3\dot{x}_{t_k} - \dot{x}_{t_{k-1}})\Delta t/2$

Verlet Integration

$x_{t_{k+1}} = x_{t_k} + \dot{x}_{t_k}\Delta t + \ddot{x}_{t_k}\Delta t/2$

$x_{t_{k-1}} = x_{t_k} - \dot{x}_{t_k}\Delta t + \ddot{x}_{t_k}\Delta t/2$

$x_{t_{k+1}} = 2x_{t_k} - x_{t_{k-1}} + \ddot{x}_{t_k}\Delta t$

Which to use

In practice, Euler Method or 2nd Order Runge Kutta is usually good enough for real-time apps (60 frames/sec)

If simulation becomes unstable even though system dynamics are stable implies that errors are being introduced by the numerical integration.

- Reduce the step size
- Try using the implicit Euler method
- Try higher order integration scheme

FEEDBACK CONTROL

<u>SYSTEM DYNAMICS</u>

Input: $u = \begin{bmatrix} f \\ \tau \end{bmatrix}$, State: $x = \begin{bmatrix} p \\ \theta \\ v \\ \omega \end{bmatrix}$

Dynamics: $\dot{x} = f(x, u)$

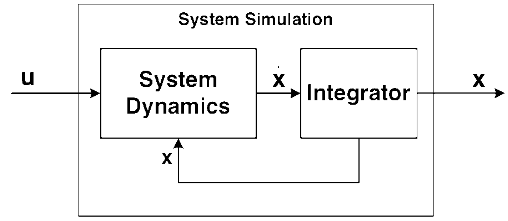
E.g.

2nd order system $\ddot{x} + a_1\dot{x} + a_0 x = bu$

3rd order system $\dddot{x} + a_2\ddot{x} + a_1\dot{x} + a_0 x = bu$

<u>SYSTEM SIMULATION</u>

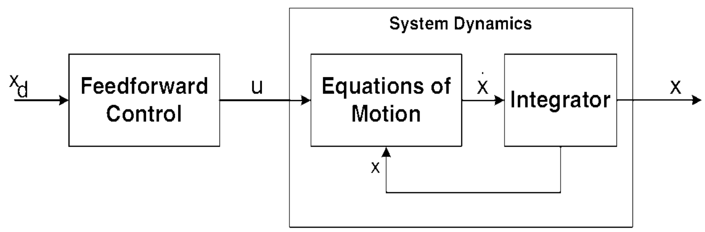
Objective: compute input u such that state x moves to desired state x_d over time



<u>FEEDFORWARD CONTROL</u>

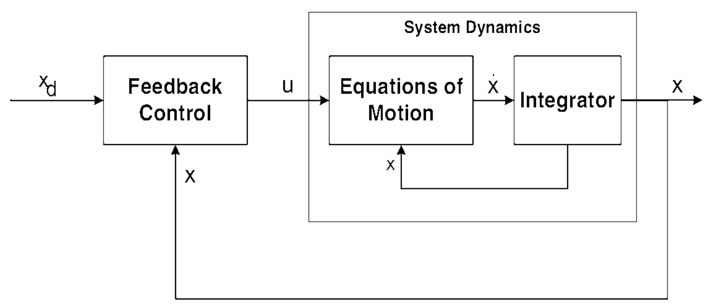
- The control input (u) is determined as a function of the desired state (x_d) and/or time (t) without considering the actual value of the system state (x).
- This type of control is only effective if the system can be accurately modeled and there are no disturbances in the environment that can affect the system state.

$$\ddot{x} = \ddot{x}_d \Longrightarrow \dot{x} = \dot{x}_d \Longrightarrow x = x_d$$



<u>FEEDBACK CONTROL</u>

- The objective of the feedback controller is to get the system to achieve and maintain the desired state (x_d).
- The control input (u) is determined based on the error between the actual state (x) and desired state (xd) of the dynamic system



Types of feedback control

- Proportional (P)

$$u_{fb} = K_p(x_d - x)$$

- Proportional Derivative (PD)

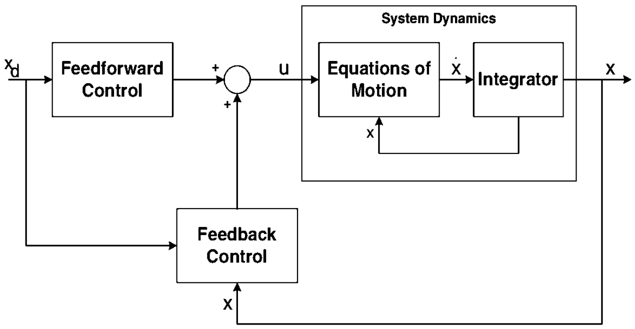
$$u_{fb} = K_p(x_d - x) + K_d(v_d - v)$$

- Proportional Integral Derivative (PID)

$$u_{fb} = K_p(x_d - x) + K_d(v_d - v) + K_I \int (x_d - x)dt$$

Feedforward + feedback

$$u = u_{ff} + u_{fb}$$



MASS SPRING DAMPER SYSTEM

Translation (1D)

Equation of motion: $\ddot{x} + \left(\frac{c}{m}\right)\dot{x} + \left(\frac{k}{m}\right)x = \frac{f}{m}$

Desired dynamics: $\ddot{x} + 2\zeta\omega_n\dot{x} + \omega_n^2 x = \omega_n^2 x_d$

Solution:

$$x(t) = \left(1 - e^{-\zeta\omega_n t}\left(\cos(\omega_n t) + \frac{\zeta}{\sqrt{1-\zeta^2}}\sin(\omega_n t)\right)\right)x_d$$

Where ζ is damping ratio and ω_n is natural frequency
$\zeta = 0$ - no damping
$0 \leq \zeta \leq 1$ - oscillatory
$\zeta = 1$ - critically damped
$\zeta > 1$ - no oscillation
System time constant $T_{TC} = 1/\zeta\omega_n$
Settling time $T_{settle} = 4T_{TC}$ (98% steady)

Rotation (1D)

Equation of motion: $\ddot{\theta} + \left(\frac{c}{I_{zz}}\right)\dot{\theta} + \frac{k}{I_{zz}}\theta = \frac{\tau}{I_{zz}}$

CONTROLLER DESIGN

Compute f as a function of x_d, x, v such that the actual dynamics behaves like the desired system dynamics (e.g. settle in certain time, exhibits desired amount of oscillation, etc.)
Take the mass spring damper system for example:

Position Controller Design

Desired dynamics: $\ddot{x} + 2\zeta\omega_n\dot{x} + \omega_n^2 x = \omega_n^2 x_d$
Solution: $f = (c - 2m\zeta\omega_n)\dot{x} + (k - m\omega_n^2)x + m\omega_n^2 x_d = K_P x + K_D\dot{x} + K_0 x_d$

Velocity Controller Design

Desired dynamics: $\ddot{x} + \alpha\dot{x} = \alpha\dot{x}_d \ or \ (\dot{v} + \alpha v = \alpha v_d)$
Solution: $f = (c - m\alpha)\dot{x} + kx + m\alpha\dot{x}_d = K_P x + K_D v + K_1 v_d$

Tracking Controller Design

Desired dynamics:
$(\ddot{x} - \ddot{x}_d) + 2\zeta\omega_n(\dot{x} - \dot{x}_d) + \omega_n^2(x - x_d) = 0$
Solution: $f = (c - 2m\zeta\omega_n)\dot{x} + (k - m\omega_n^2)x + m\omega_n^2 x_d + 2m\zeta\omega_n\dot{x}_d + m\ddot{x}_d = K_P x + K_D\dot{x} + K_0 x_d + K_1\dot{x}_d + K_2\ddot{x}_d$

VIHECLE DYNAMICS

Equations of Motion (world frame)

$f^0 = m\dot{V}^0)$ - translation
$\tau^0 = \omega^0 \times I^0 \cdot \omega^0 + I^0 \cdot \dot{\omega}^0$ - rotation

Body frame to world frame $V^0 = R_B^0 V^B$

Equations of Motion (body frame)

$f^B = m\dot{V}^B + m\omega^B \times V^B$
$\tau^B = I^B \cdot \dot{\omega}^B + \omega^B \times I^B \cdot \omega^B$

2D planar case (world frame)

$$m\dot{V}_x^0 = f_x^0, \quad m\dot{V}_y^0 = f_y^0, \quad I_{zz}\dot{\omega}_z = \tau_z^0$$

2D planar case (body frame)

$$V^B = \left[V_x^B, V_y^B, 0\right]^T, \omega^B = \left[0, 0, \dot{\theta}\right]^T$$
$$m\dot{V}_x^B - m\dot{\theta}V_y^B = f_x^B, m\dot{V}_y^B + m\dot{\theta}V_x^B = f_y^B, I_{zz}\dot{\omega}_z = \tau_z^0$$

3D case (world frame)

Desired value: V_d, R_d
Velocity controller:
$f^0 = m\left(K_p(V_d - V^0)\right)$ (world frame)
$f^B = m\left(K_p(V_d - V^B) + \omega^B \times V^B\right)$ (body frame)

Angle controller:

$$\tau = I(K_P\Delta\theta - K_D\omega^B) + \omega^B \times I \cdot \omega^B$$

where $\Delta R = R^T R_d, \Delta\theta = AxisAngle(\Delta R)$

Plug in, we got the close loop vehicle dynamics:

$$\dot{V} + K_P V = K_P V_d, \quad \dot{\omega} + K_V\omega = K_P\Delta\theta$$

Integrate to update state R

FACIAL ANIMATION

FACE MODELING METHODS

3D modeling
photograph & digitize
sculpt & digitize
scanning
computer vision

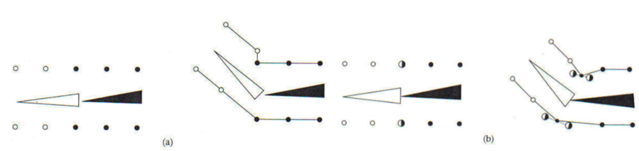
FACE ANIMATION METHODS

Blend shapes

Require several key expressions to be modeled ahead of time. The key expressions are then blended on the fly to create a new expression, either locally or globally.
You can use either linear (LERP) or non-linear interpolation (B-spline, Bazier, etc)
LERP $v^* = v_{base} + \sum w_i(v_i - v_{base}) = (1 - \sum w_i)v_{base} + \sum w_i v_i$
Also for the normal, we have
$n^* = n_{base} + \sum w_i(v_i - v_{base}), n^* = n^*/\|n^*\|$
If multiple targets affect the same vertex, their results combine in a reasonable way

Example: Skinning

- Transform blend



center vertices attached to both bones, perform weighted sum to determine actual position of center vertices

$$\mathbf{x}_{VT}^0(\theta) = w\mathbf{H}_{i+1}^0(\theta)\mathbf{x}_V^{i+1}(\theta_0) + (1-w)\mathbf{H}_i^0(\theta)\mathbf{x}_V^i(\theta_0)$$

$$\text{since} \quad \mathbf{x}_V^{i+1}(\theta_0) = \mathbf{H}_i^{i+1}(\theta_0)\mathbf{x}_V^i(\theta_0)$$

- Shape blend

$$\mathbf{x}^i_{VS}(\beta) = \beta \mathbf{x}^i_{VS}(\theta_{\max}) + (1-\beta)\mathbf{x}^i_{VS}(\theta_0)$$

$$\beta = \frac{(\theta - \theta_0)}{(\theta_{\max} - \theta_0)} \quad \in [0,1]$$

$$\mathbf{x}^0_{VST}(\theta) = [w\mathbf{H}^0_{i+1}(\theta)\mathbf{H}^{i+1}_i(\theta) + (1-w)\mathbf{H}^0_i(\theta)\ [\beta \mathbf{x}_{VS}(\theta_{\max}) + (1-\beta)\mathbf{x}_{VS}(\theta_0)]$$

- smooth skinning
  add intermediate joint-points

Example: Facial Animation

As opposed to just two joints, in facial animation, the deformed vertex position is a weighted average over many joints to which the vertex is attached

$$\mathbf{v}'' = \sum w_i \mathbf{H}^0_i \cdot (\mathbf{B}^0_i)^{-1}\mathbf{v}' \quad \text{where} \quad \sum w_i = 1$$

- v' is the untransformed vertex position in world coordinates from blend shapes calculation
- Bi is the binding matrix (global transform of joint i when the skin was initially attached)
- v" is the final vertex position in world space
- w_i is the weight of joint i
- H_i is the current world matrix of joint i after running the skeleton forward kinematics

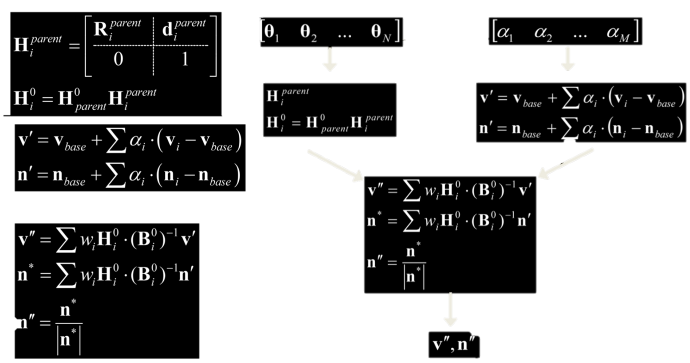
Note:

- B remains constant, so B^{-1} can be computed at load time, H B^{-1}can be computed for each joint before skinning

Blending normals is essentially the same, except we transform them as directions (x,y,z,0) and then renormalize the results

$$\mathbf{n}^* = \sum w_i \mathbf{H}^0_i \cdot (\mathbf{B}^0_i)^{-1}\mathbf{n}' \qquad \mathbf{n}'' = \frac{\mathbf{n}^*}{|\mathbf{n}^*|}$$

Skeleton, Morph, & Skin Data Flow



Muscle-based

Each muscle has a zone of influence

Performance-based

Performance based methods capture the motion of live performers and translate the actions into facial animation control parameters

Performance-driven facial animation system that maps points in images of a face to a 3D Model

Input: Real-time video capture from a single camera (e.g. webcam)

Output:

- Head movement and rotation
- 64 facial points and 100 expressions per frame
- Face texture extraction

Parametric

FACS (Facial Action Coding System) describes all visually distinguishable facial movements based on an anatomical analysis of facial actions

MPEG-4 based

Efficient encoding scheme of the human face intended for multi-media scenes/video conferencing

- Promotes visual speech intelligibility and the recognition of the mood of the speaker with low bandwidth requirements
- Implemented as a collection of nodes and parameters in a scene graph which are animated

Facial Definition Parameters (FDP)

- 84 feature points used to transform a generic face model into a particular face of specific shape and texture

Facial Animation Parameters (FAP)

- 68 parameters used to animate expressions and speech through translation of FDPs and rotation of head, eyes, eyelids and jaw

LIPSYNCHING

Speech Synchronization

First, generate a sequence of (phoneme, duration) pairs from voice recognition or text-to-speech

Next, generate the associated mouth deformation (i.e. Viseme)