• Surface is deformed by a set of bones
• Bones are in turn controlled by a smaller set of controls
• The controls are useful, intuitive DOFs for an animator to use
Forward vs. Inverse Kinematics

Inverse Kinematics Solves for all Intermediate Constraints

Skinning

• After solving for the skeleton, one still needs to update and deform the surface

Mesh skinning math: setup

• Surface has control points $p_i$
  – Triangle vertices, spline control points, subdiv base vertices

• Each bone has a transformation matrix $M_j$
  – Normally a rigid motion

• Every point–bone pair has a weight $w_{ij}$
  – In practice only nonzero for small # of nearby bones
  – The weights are provided by the user
Mesh skinning math

- Deformed position of a point is a weighted sum
  - of the positions determined by each bone's transform alone
  - weighted by that vertex's weight for that bone

\[ p'_i = \sum_j w_{ij} M_j p_i \]

Skinning Mesh Animations

Doug L. James
Christopher D. Twigg
Carnegie Mellon University

Physics-Based Animation

Motion Capture Can Be Used for Data-Driven Methods

https://youtu.be/_QZN2lC0vOo

http://graphics.cs.cmu.edu/projects/sma/, 2005
Animation vs. Simulation

- Animation methods use scripted actions to make objects change
- Simulation: simulate physical laws by associating physical properties to objects
- Solve for physics to achieve (predict) realistic effects

Using Particle Systems

- Idea: Represent the physics on the simplest possible entity: particles
- Used for effects like smoke, fire, water, sparks, and more
- Plenty of other approaches, this is just one family

Unified Particle Physics for Real-Time Applications
Miles Macklin  Matthias Müller  Nuttapong Chentanez  Tae-Yong Kim
NVIDIA

http://blog.mmacklin.com/flex/, 2014

Used in Games Physics

Particle System Setup

```cpp
class Particle {
  Vector3 position;
  Vector3 velocity;
};
```

Moving Particles

Position is a function of time
  • i.e., \( \vec{x} \equiv \vec{x}(t) \)
  • Note that \( \vec{v}(t) \equiv \frac{\partial \vec{x}}{\partial t} \)

Use a function to control the particle's velocity
  • \( \vec{v}(t) = f(\vec{x}(t)) \)

This is an Ordinary Differential Equation (ODE)
Solve this ODE at every frame
  • i.e., solve for \( \vec{x}(t_0), \vec{x}(t_1), \vec{x}(t_2), \ldots \)
  • Then we can draw each of these positions to the screen
A Simple Example

Let \( \vec{v} \) be constant

- e.g., \( \vec{v} = f(x) = (0, 0, 1)^T \)

Then we can solve for the position at any time:

- \( \vec{x}(t) = \vec{x}(0) + t \vec{v} \)

Not always so easy

- \( f(\vec{v}) \) can be anything!
- Might be unknown until runtime (e.g., user interaction)
- Often times, not solved exactly

Moving Particles, Revisited

Now, acceleration is in the mix

- \( a(t) = \frac{\partial^2 \vec{x}}{\partial t^2} \)

Use a function to control the particle’s acceleration

- \( a(t) = f(\vec{x}(t)) \)

This is a Second Order ODE

Solve this ODE at every frame, same as before

- Can sometimes be reduced to a first order ODE
- Calculate position and velocity together

Physically-based Motion

Acceleration based on Newton’s laws

- \( \vec{f}(t) = m \vec{a}(t) \) ... or, equivalently ... \( \vec{a}(t) = \frac{\vec{f}(t)}{m} \)
- i.e., force is mass times acceleration

Forces are known beforehand

- e.g., gravity, springs, others....
- Multiple forces sum together
- These often depend on the position, i.e., \( \vec{f}(t) \equiv \vec{F}(\vec{x}(t)) \)
- Sometimes velocity, too

If we know the values of the forces, we can solve for particle’s state

Unary Forces

Constant

- Gravity

Position/Time-Dependent

- Force fields, e.g. wind

Velocity-Dependent

- Drag
Ordinary Differential Equations

\[ \frac{d\mathbf{X}(t)}{dt} = f(\mathbf{X}(t), t) \]

- Given a function \( f(\mathbf{X}, t) \) compute \( \mathbf{X}(t) \)
- Typically, initial value problems:
  - Given values \( \mathbf{X}(t_0) = \mathbf{X}_0 \)
  - Find values \( \mathbf{X}(t) \) for \( t > t_0 \)
- We can use lots of standard tools

Newtonian Mechanics

- Point mass: 2nd order ODE
  \[ \vec{F} = m\vec{a} \quad \text{or} \quad \vec{F} = m\frac{d^2\vec{x}}{dt^2} \]
- Position \( x \) and force \( F \) are vector quantities
  - We know \( F \) and \( m \), want to solve for \( x \)
- You have all seen this a million times before

Reduction to 1st Order

- Point mass: 2nd order ODE
  \[ \vec{F} = m\vec{a} \quad \text{or} \quad \vec{F} = m\frac{d^2\vec{x}}{dt^2} \]
- Corresponds to system of first order ODEs
  \[ \begin{align*}
  \frac{d\vec{x}}{dt} &= \vec{v} \\
  \frac{d\vec{v}}{dt} &= \vec{F}/m
  \end{align*} \]
- Why reduce?
  2 unknowns \((\mathbf{x}, \mathbf{v})\) instead of just \( \mathbf{x} \)
Reduction to 1st Order

\[
\begin{align*}
\frac{d}{dt} \vec{x} &= \vec{v} \\
\frac{d}{dt} \vec{v} &= \vec{F} / m
\end{align*}
\]

2 variables \((\vec{x}, \vec{v})\) instead of just one

- Why reduce?
  - Numerical solvers grow more complicated with increasing order, can just write one 1st order solver and use it
  - Note that this doesn’t mean it would always be easy :-)

Notation

- Let’s stack the pair \((\vec{x}, \vec{v})\) into a bigger state vector \(\vec{X}\)
  \[ \begin{pmatrix} \vec{x} \\ \vec{v} \end{pmatrix} \]

For a particle in 3D, state vector \(\vec{X}\) has 6 numbers

\[
\begin{aligned}
\frac{d}{dt} \vec{X} &= f(\vec{X}, t) = \begin{pmatrix} \vec{v} \\ \vec{F}(x, v)/m \end{pmatrix}
\end{aligned}
\]

Now, Many Particles

- We have \(N\) point masses
  - Let’s just stack all \(x\)s and \(v\)s in a big vector of length \(6N\)

\[
\begin{pmatrix} x_1 \\ v_1 \\ \vdots \\ x_N \\ v_N \end{pmatrix} \quad f(\vec{X}, t) = \begin{pmatrix} v_1 \\ F^1(X, t) \\ \vdots \\ v_N \\ F^N(X, t) \end{pmatrix}
\]

Now, Many Particles

- We have \(N\) point masses
  - Let’s just stack all \(x\)s and \(v\)s in a big vector of length \(6N\)
  - \(F^i\) denotes the force on particle \(i\)
    - When particles don’t interact, \(F^i\) only depends on \(x_i\) and \(v_i\).

\[
\begin{pmatrix} x_1 \\ v_1 \\ \vdots \\ x_N \\ v_N \end{pmatrix} \quad f(\vec{X}, t) = \begin{pmatrix} v_1 \\ F^1(X, t) \\ \vdots \\ v_N \\ F^N(X, t) \end{pmatrix}
\]

\(f\) gives \(d/dt \vec{X}\), remember!
Path through a Vector Field

- \( X(t) \): path in multidimensional phase space

\[
\frac{d}{dt} X = f(X, t)
\]

“When we are at state \( X \) at time \( t \), where will \( X \) be after an infinitely small time interval \( dt \)?”

\( f = \frac{d}{dt} X \) is a vector that sits at each point in phase space, pointing the direction.

Integration Algorithm 1

Calculating Particle State from Forces: First attempt
- Use forces to update velocity: \( \vec{v}(t + h) = \vec{v}(t) + \frac{h}{m} \vec{f}(t) \)
- Use old velocity to update position: \( \vec{x}(t + h) = \vec{x}(t) + h \vec{v}(t) \)

Issues
- Unstable in certain cases!
- Reducing time step can help, but this becomes computationally expensive
- Error is \( O(h^2) \) per step (and accumulates). Error is \( O(h) \) globally.

This technique is called Forward (Explicit) Euler Integration

Example: circle

Comparison Euler, Step Sizes

Euler quality is proportional to \( dt \)
Intuitive Solution: Take Steps

- Current state \( X \)
- Examine \( f(X,t) \) at (or near) current state
- Take a step to new value of \( X \)
  \[
  \frac{d}{dt} X = f(X,t)
  \]
  \[\Rightarrow \text{"} \frac{dX}{dt} = dt \cdot f(X,t) \text{"} \]

\( f = \frac{d}{dt} X \) is a vector that sits at each point in phase space, pointing the direction.

Euler’s Method

- Simplest and most intuitive
- Pick a step size \( h \)
- Given \( X_0 = X(t_0) \), take step:
  \[
  t_1 = t_0 + h
  \]
  \[
  X_1 = X_0 + h \cdot f(X_0,t_0)
  \]

- Piecewise-linear approximation to the path
- **Basically, just replace \( dr \) by a small but finite number**

Euler, Visually

\[
\frac{d}{dt} X = f(X,t)
\]

Image by MIT OpenCourseWare.

Euler, Visually

\[
\frac{d}{dt} X = f(X,t)
\]

Image by MIT OpenCourseWare.
Euler, Visually

\[ \frac{d}{dt} \mathbf{X} = f(\mathbf{X}, t) \]

Another Simple Example: Sprinkler

```cpp
list<Particle> PL;
spread = 0.1; //how random the velocity is

//add k particles to the list
for (int i=0; i<k; i++) {
    Particle p;
    p->position = Vec3(0,0,0);
    p->velocity = Vec3(0,0,1) + spread*Vec3(rand(), rand(), rand());
    PL->add(p);
}

for (each time step) {
    for (each particle p in PL) {
        p->position += p->velocity*dt; //dt: time step
        p->velocity -= g*dt; //g: gravitation constant
    }
}
```

Binary, \( n \)-ary Forces

Much more interesting behaviors to be had from particles that interact

Simplest: binary forces, e.g. springs

\[
\mathbf{f}_i(\mathbf{x}_i, \mathbf{x}_j) = -k_s(\|\mathbf{x}_i - \mathbf{x}_j\| - r_{ij}) \frac{\mathbf{x}_i - \mathbf{x}_j}{\|\mathbf{x}_i - \mathbf{x}_j\|}
\]

Nice example project with mass-spring systems:

- https://vimeo.com/73188339

More sophisticated models for deformable things use forces relating 3 or more particles

Particle System Setup, Revisited

```java
class Particle {
    float mass;
    Vector3 position;
    Vector3 velocity;
    Vector3 force;
};
```

Basic Algorithm

1) Clear forces from previous calculations
2) Calculate/accumulate forces for each particle
3) Solve for particle’s state (position, velocity) for the next time step \( h \)
Generalizations

• It’s not all hacks: Smoothed Particle Hydrodynamics (SPH)
  – A family of “real” particle-based fluid simulation techniques.
  – Fluid flow is described by the Navier-Stokes Equations, a nonlinear partial differential equation (PDE)
    • SPH discretizes the fluid as small packets (particles!), and evaluates pressures and forces based on them.

Jos Stam