CS559: Computer Graphics

Lecture 26: Animation

Li Zhang

Spring 2010

Slides from Brian Curless at U of Washington

Animation

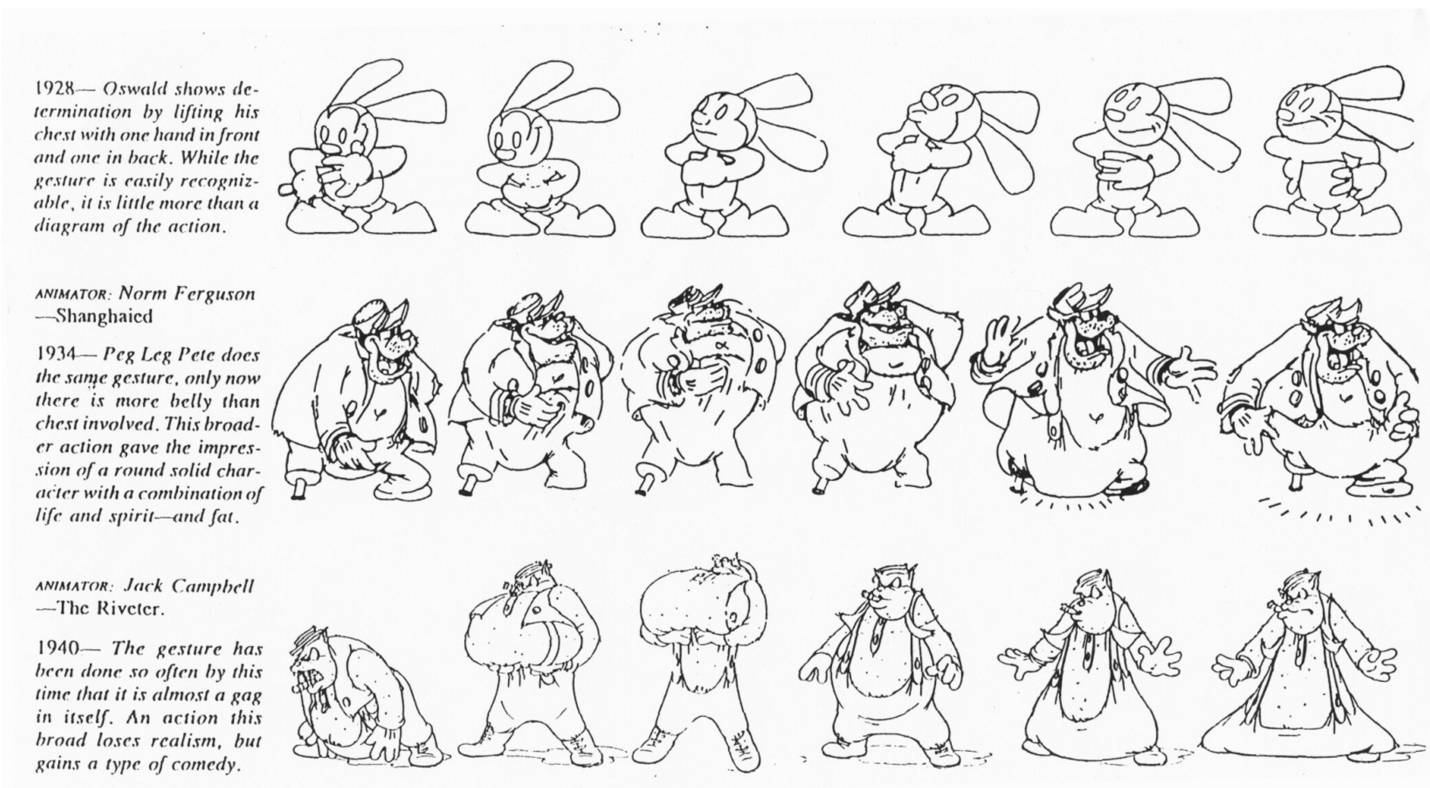
- Traditional Animation – without using a computer

Animation

- Computer Animation

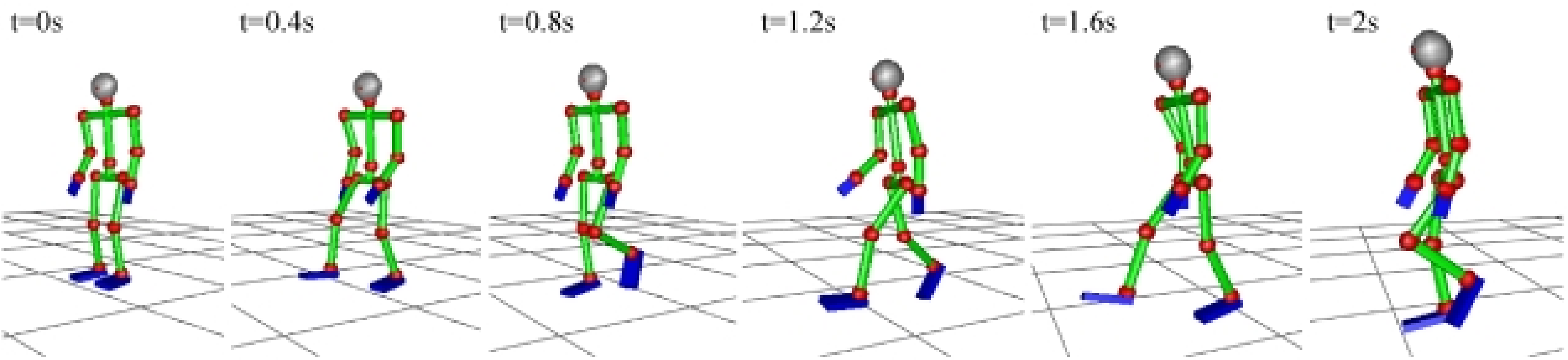
Types of Animation

- Cartoon Animation



Types of Animation

- Cartoon Animation
- Key Frame Animation



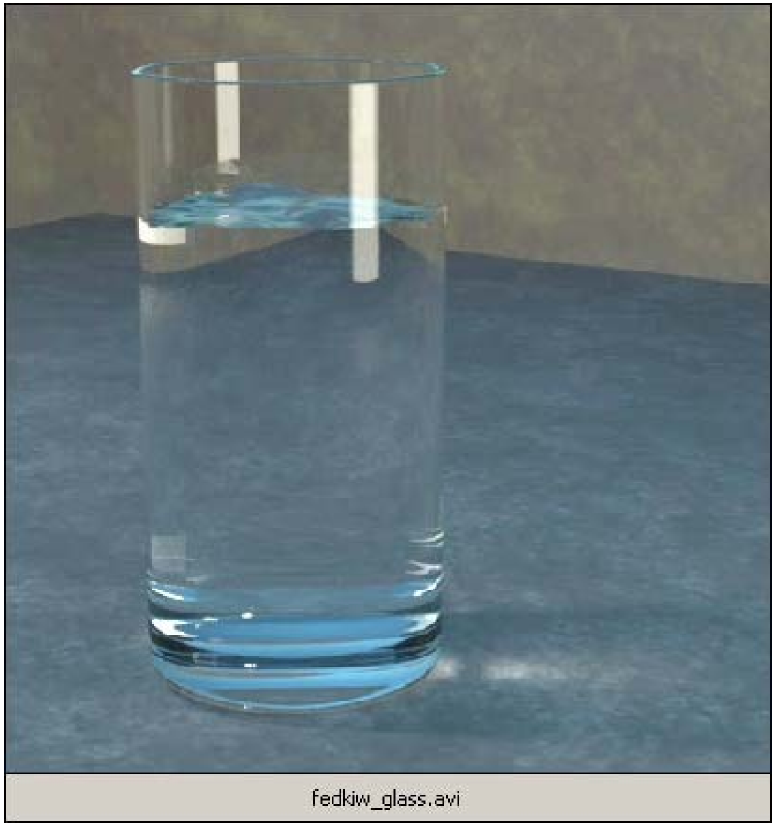
Types of Animation

- Cartoon Animation
- Key Frame Animation
- Physics based animation

Nguyen, D., Fedkiw, R. and Jensen, H., "Physically Based Modeling and Animation of Fire", SIGGRAPH 2002

Types of Animation

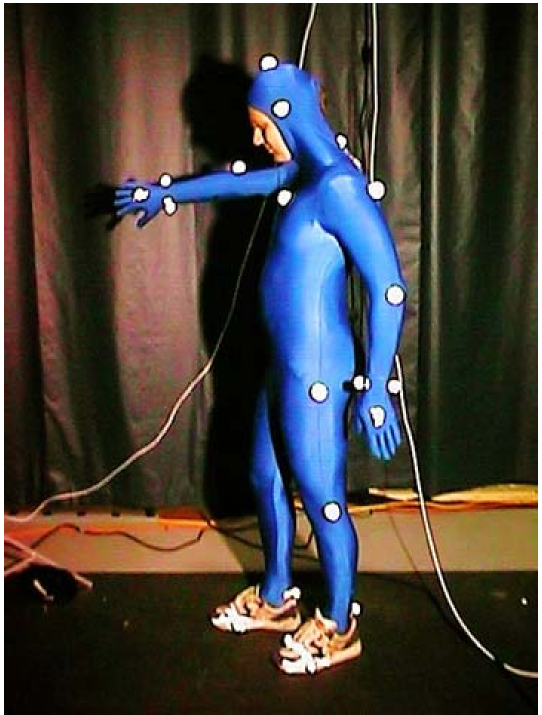
- Cartoon Animation
- Key Frame Animation
- Physics based animation



Enright, D., Marschner, S. and Fedkiw, R., "Animation and Rendering of Complex Water Surfaces", SIGGRAPH 2002

Types of Animation

- Cartoon Animation
- Key Frame Animation
- Physics based animation
- Data driven animation



Types of Animation

- Cartoon Animation
- Key Frame Animation
- Physics based animation
- Data driven animation

Types of Animation

- Cartoon Animation
- Key Frame Animation
- Physics based animation
- Data driven animation

Types of Animation

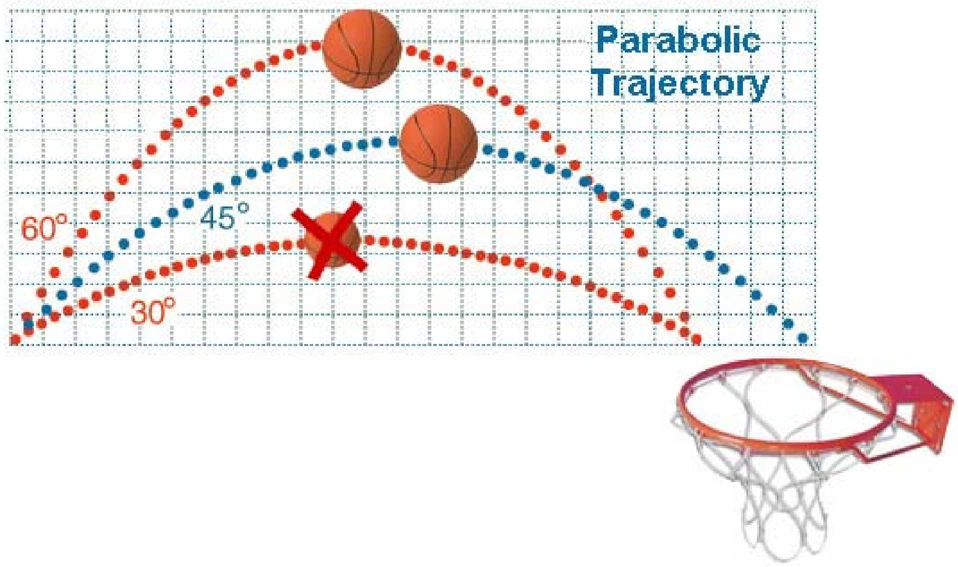
- Cartoon Animation
- Key Frame Animation
- Physics based animation
- Data driven animation

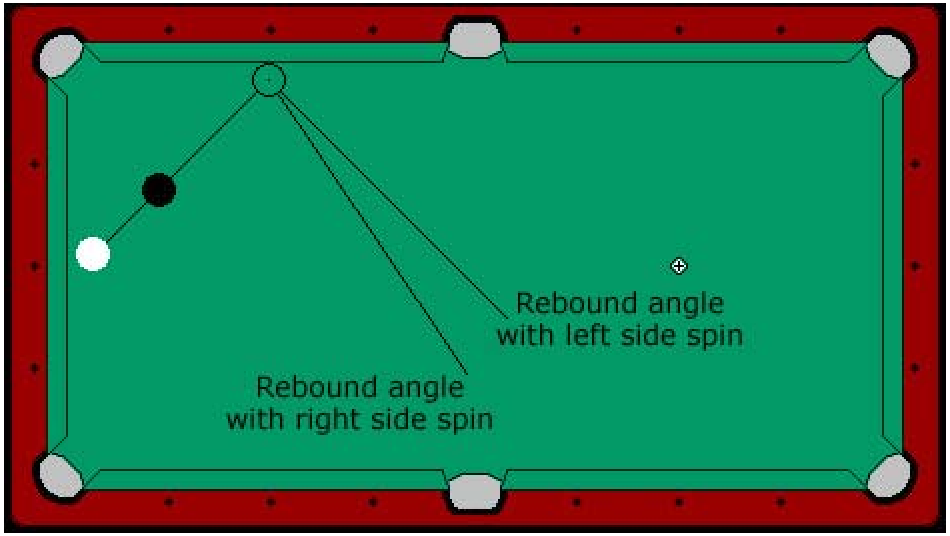
Particle Systems

- What are particle systems?
  - A **particle system** is a collection of point masses that obeys some physical laws (e.g, gravity, heat convection, spring behaviors, …).

- Particle systems can be used to simulate all sorts of physical phenomena:

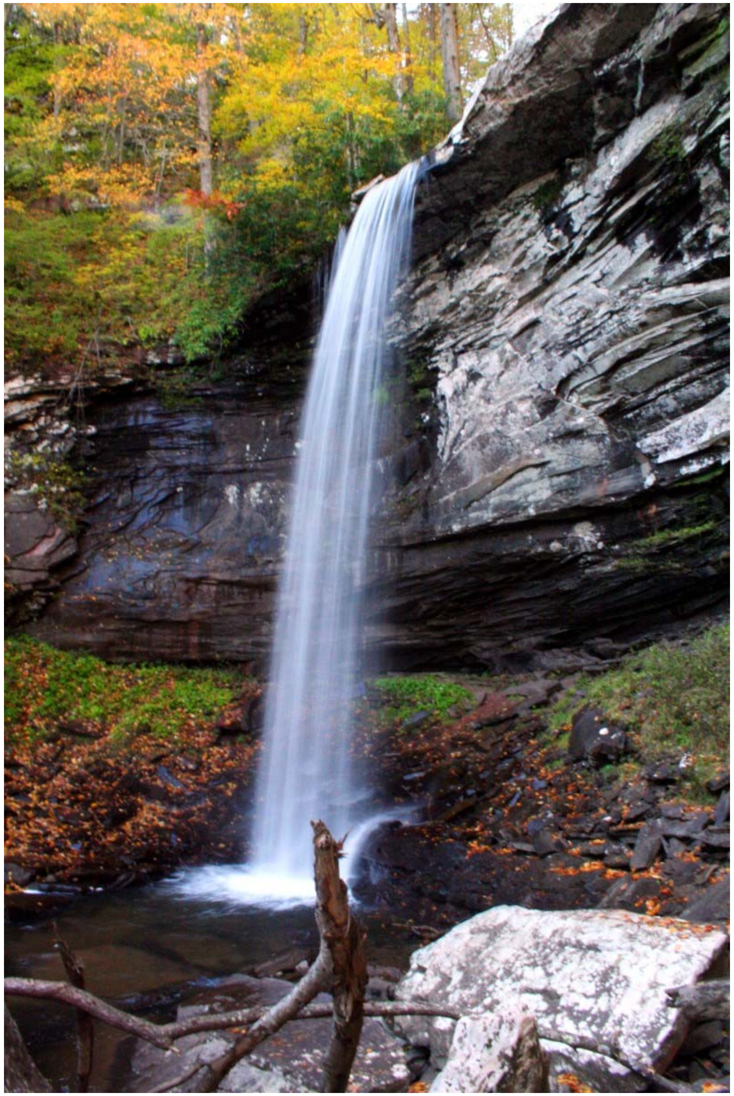
Balls in Sports



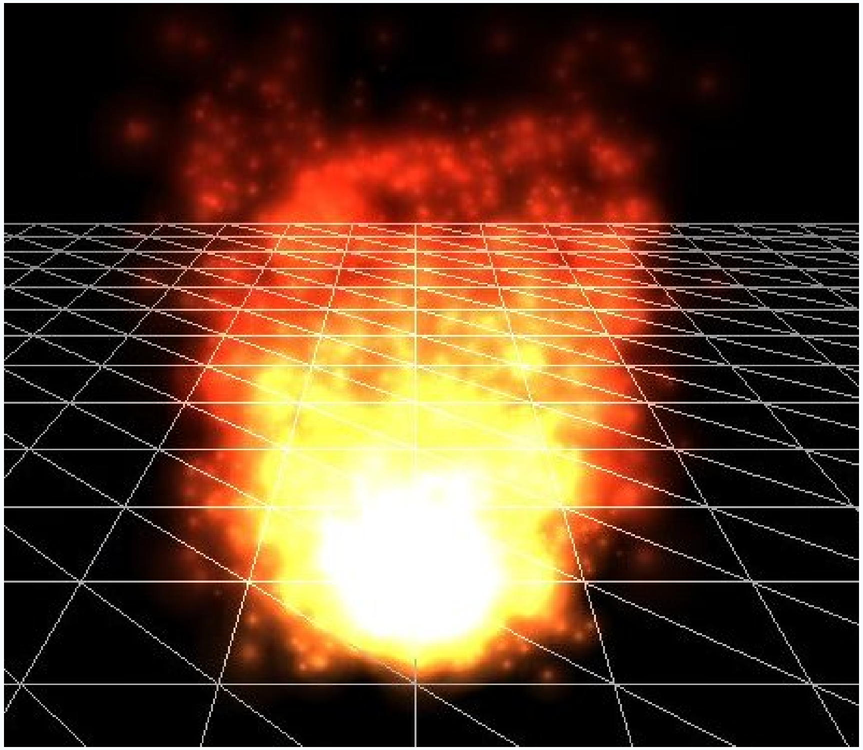


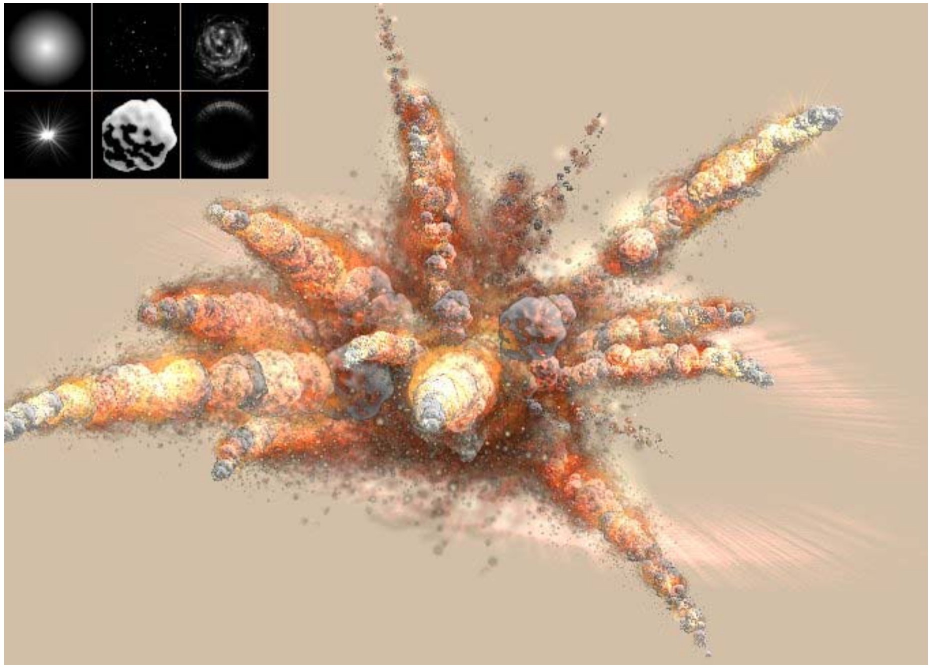
Fireworks

Water



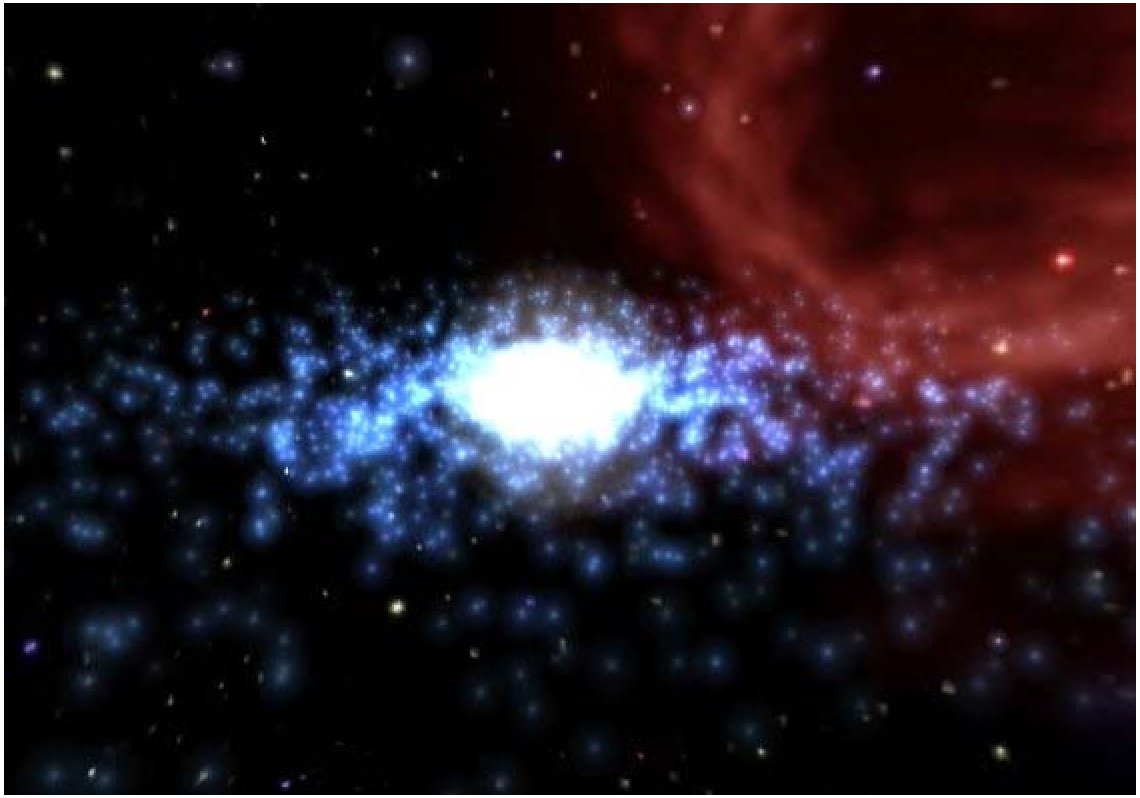
Fire and Explosion





http://en.wikipedia.org/wiki/Particle_system

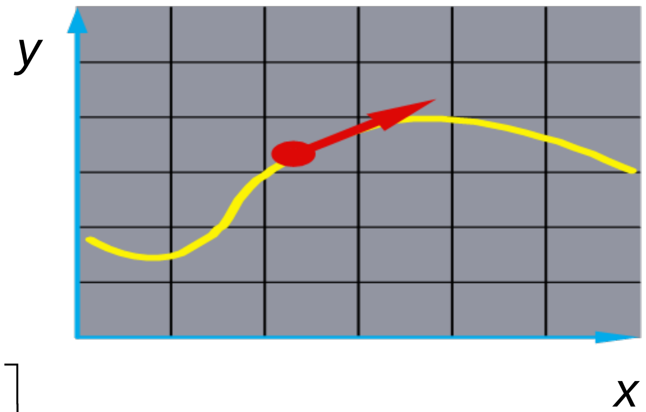
Galaxy



http://en.wikipedia.org/wiki/Particle_system

Particle in a flow field

- We begin with a single particle with:
  - Position, $\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}$



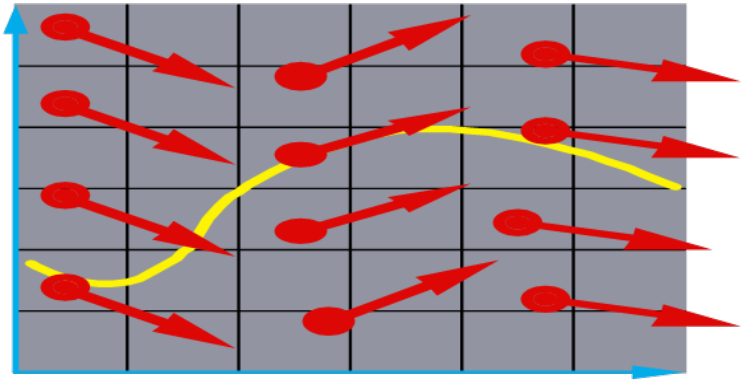
  - Velocity, $\mathbf{v} \equiv \dot{\mathbf{x}} = \frac{d\mathbf{x}}{dt} = \begin{bmatrix} dx/dt \\ dy/dt \end{bmatrix}$

- Suppose the velocity is actually dictated by some driving function **g**:

$$\dot{\mathbf{x}} = \mathbf{g}(\mathbf{x}, t)$$

Vector fields

- At any moment in time, the function **g** defines a vector field over **x**:
  - Wind
  - River



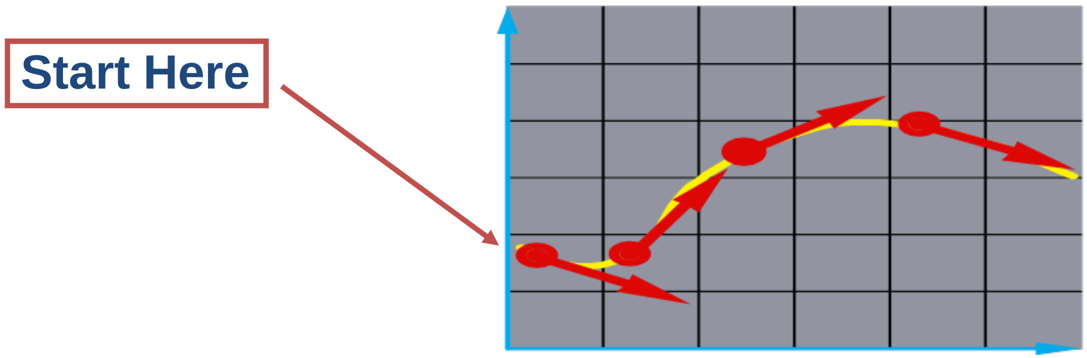
- How does our particle move through the vector field?

Diff eqs and integral curves

- The equation

$$\dot{\mathbf{x}} = \mathbf{g}(\mathbf{x}, t)$$

- is actually a **first order differential equation.**
- We can solve for **x** through time by starting at an initial point and stepping along the vector field:



- This is called an **initial value problem** and the solution is called an **integral curve**.
  - Why do we need initial value?

Euler's method

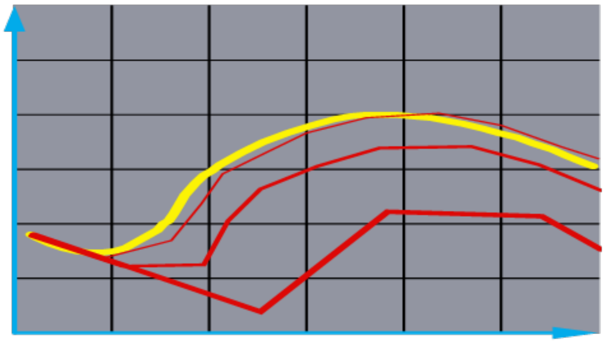
- One simple approach is to choose a time step, Δt, and take linear steps along the flow:

$$\mathbf{x}(t+\Delta t) \approx \mathbf{x}(t) + \Delta t \cdot \dot{\mathbf{x}}(t)$$
$$\approx \mathbf{x}(t) + \Delta t \cdot \mathbf{g}(\mathbf{x}, t)$$

- Writing as a time iteration:

$$\mathbf{x}^{i+1} = \mathbf{x}^{i} + \Delta t \cdot \mathbf{v}^{i}$$

- This approach is called **Euler's method** and looks like:



- Properties:
  - Simplest numerical method
  - Bigger steps, bigger errors. Error ~ O(Δt^2).
- Need to take pretty small steps, so not very efficient. Better (more complicated) methods exist, e.g., "Runge-Kutta" and "implicit integration."

Particle in a force field

- Now consider a particle in a force field **f**.
- In this case, the particle has:
  - Mass, *m*
  - Acceleration,

$$\mathbf{a} \equiv \ddot{\mathbf{x}} = \dot{\mathbf{v}} = \frac{d\mathbf{v}}{dt} = \frac{d^2\mathbf{x}}{dt^2}$$

$$\mathbf{f} = m\mathbf{a} = m\ddot{\mathbf{x}}$$

- The particle obeys Newton's law:

$$\ddot{\mathbf{x}} = \frac{\mathbf{f}(\mathbf{x}, \dot{\mathbf{x}}, t)}{m}$$

- The force field **f** can in general depend on the position and velocity of the particle as well as time.
- Thus, with some rearrangement, we end up with:

Second order equations

This equation:

$$\ddot{\mathbf{x}} = \frac{\mathbf{f}(\mathbf{x}, \mathbf{v}, t)}{m}$$

is a **second order differential equation**.

Our solution method, though, worked on first order differential equations.

We can rewrite this as:

$$\begin{bmatrix} \dot{\mathbf{x}} = \mathbf{v} \\ \dot{\mathbf{v}} = \dfrac{\mathbf{f}(\mathbf{x}, \mathbf{v}, t)}{m} \end{bmatrix}$$

where we have added a new variable **v** to get a pair of coupled first order equations.

Phase space

$$\begin{bmatrix} \mathbf{x} \\ \mathbf{v} \end{bmatrix}$$

- Concatenate **x** and **v** to make a 6-vector: position in **phase space**.

$$\begin{bmatrix} \dot{\mathbf{x}} \\ \dot{\mathbf{v}} \end{bmatrix}$$

- Taking the time derivative: another 6-vector.

$$\begin{bmatrix} \dot{\mathbf{x}} \\ \dot{\mathbf{v}} \end{bmatrix} = \begin{bmatrix} \mathbf{v} \\ \mathbf{f} \;/\; m \end{bmatrix}$$

- A vanilla 1^{st}-order differential equation.

Differential equation solver

Starting with:

$$\begin{bmatrix} \dot{\mathbf{x}} \\ \dot{\mathbf{v}} \end{bmatrix} = \begin{bmatrix} \mathbf{v} \\ \mathbf{f} / m \end{bmatrix}$$

Applying Euler's method:

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t \cdot \dot{\mathbf{x}}(t)$$
$$\dot{\mathbf{x}}(t + \Delta t) = \dot{\mathbf{x}}(t) + \Delta t \cdot \ddot{\mathbf{x}}(t)$$

And making substitutions:

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t \cdot \mathbf{v}(t)$$
$$\mathbf{v}(t + \Delta t) = \dot{\mathbf{x}}(t) + \Delta t \cdot \frac{\mathbf{f}(\mathbf{x}, \dot{\mathbf{x}}, t)}{m}$$

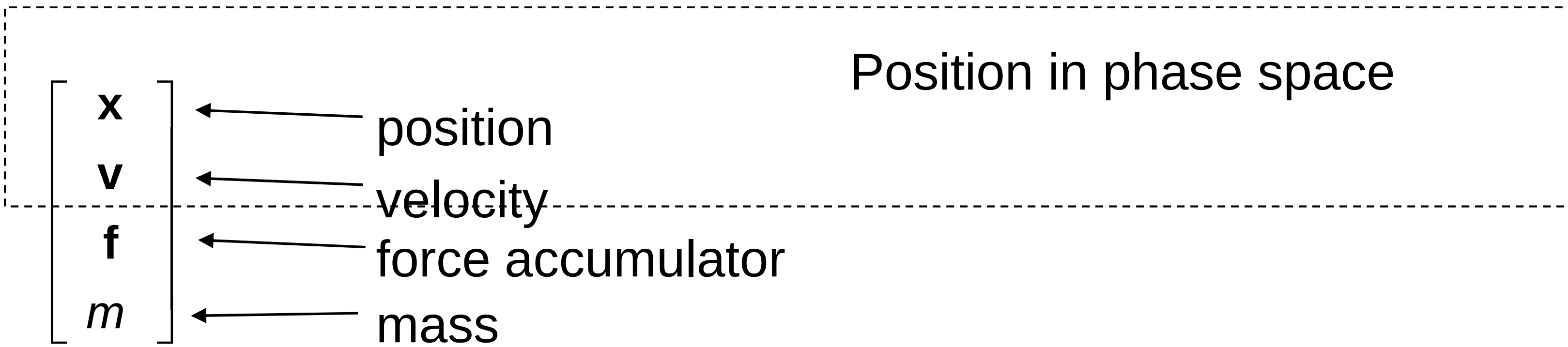
Writing this as an iteration, we have:

$$\mathbf{x}^{i+1} = \mathbf{x}^{i} + \Delta t \cdot \mathbf{v}^{i}$$
$$\mathbf{v}^{i+1} = \mathbf{v}^{i} + \Delta t \cdot \frac{\mathbf{f}^{i}}{m}$$

Again, performs poorly for large Δt.

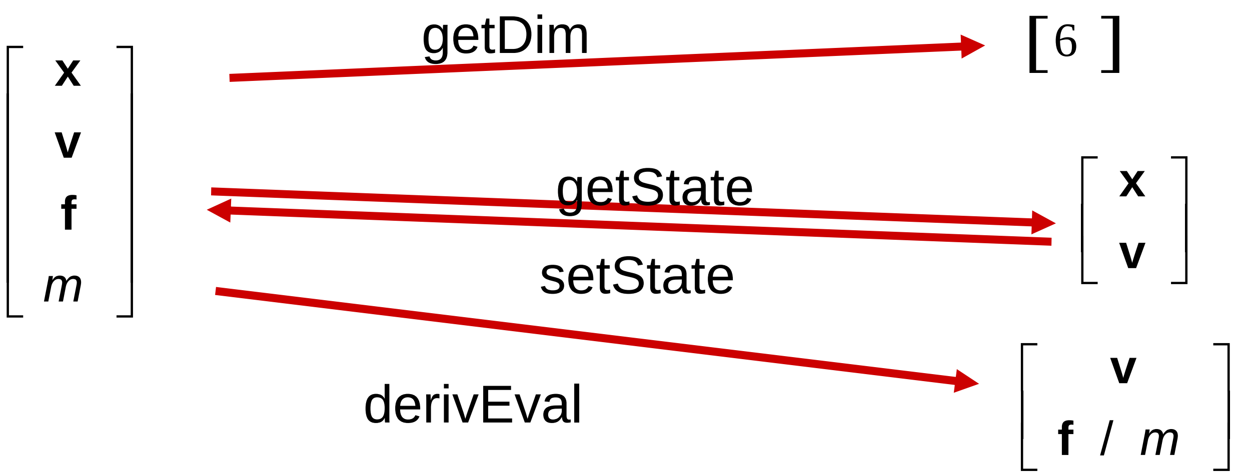
Particle structure

How do we represent a particle?




```
typedef struct{
float m; /* mass */
float *x; /* position vector */
float *v; /* velocity vector */
float *f; /* force accumulator */
} *Particle;
```

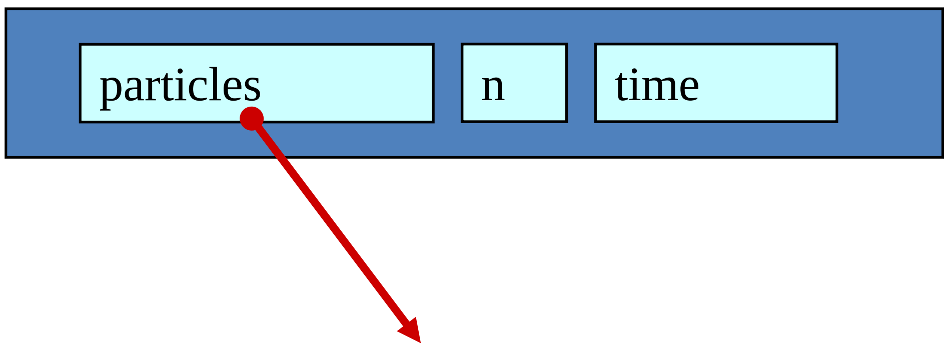
Single particle solver interface




```
typedef struct{
float m; /* mass */
float *x; /* position vector */
float *v; /* velocity vector */
float *f; /* force accumulator */
} *Particle;
```

Particle systems

In general, we have a particle system consisting of *n* particles to be managed over time:



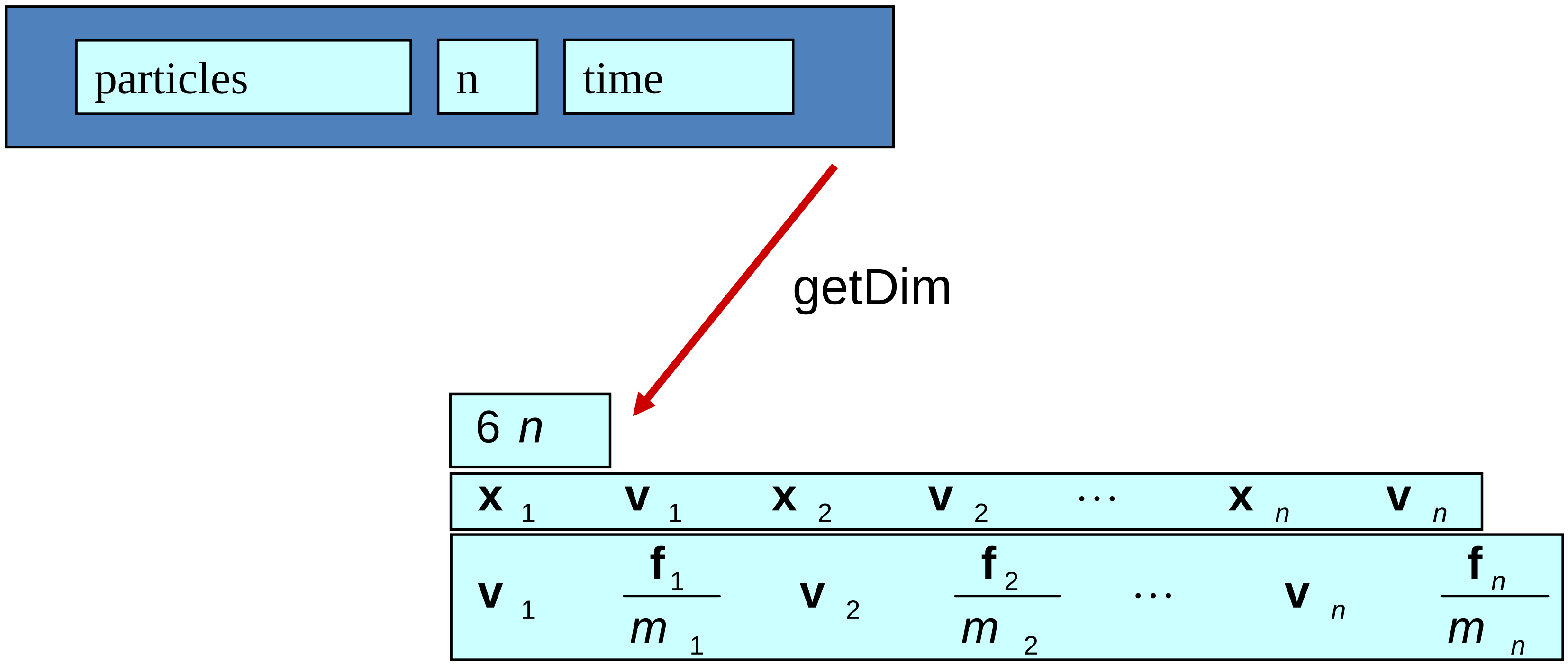
$$\begin{bmatrix} \mathbf{x}_1 \\ \mathbf{v}_1 \\ \mathbf{f}_1 \\ m_1 \end{bmatrix} \begin{bmatrix} \mathbf{x}_2 \\ \mathbf{v}_2 \\ \mathbf{f}_2 \\ m_2 \end{bmatrix} \cdots \begin{bmatrix} \mathbf{x}_n \\ \mathbf{v}_n \\ \mathbf{f}_n \\ m_n \end{bmatrix}$$

```
typedef struct{
float m; /* mass */
float *x; /* position vector */
float *v; /* velocity vector */
float *f; /* force accumulator */
} *Particle;
```

```
typedef struct{
Particle *p; /* array of pointers to particles */
int n; /* number of particles */
float t; /* simulation clock */
} *ParticleSystem
```

Particle system solver interface

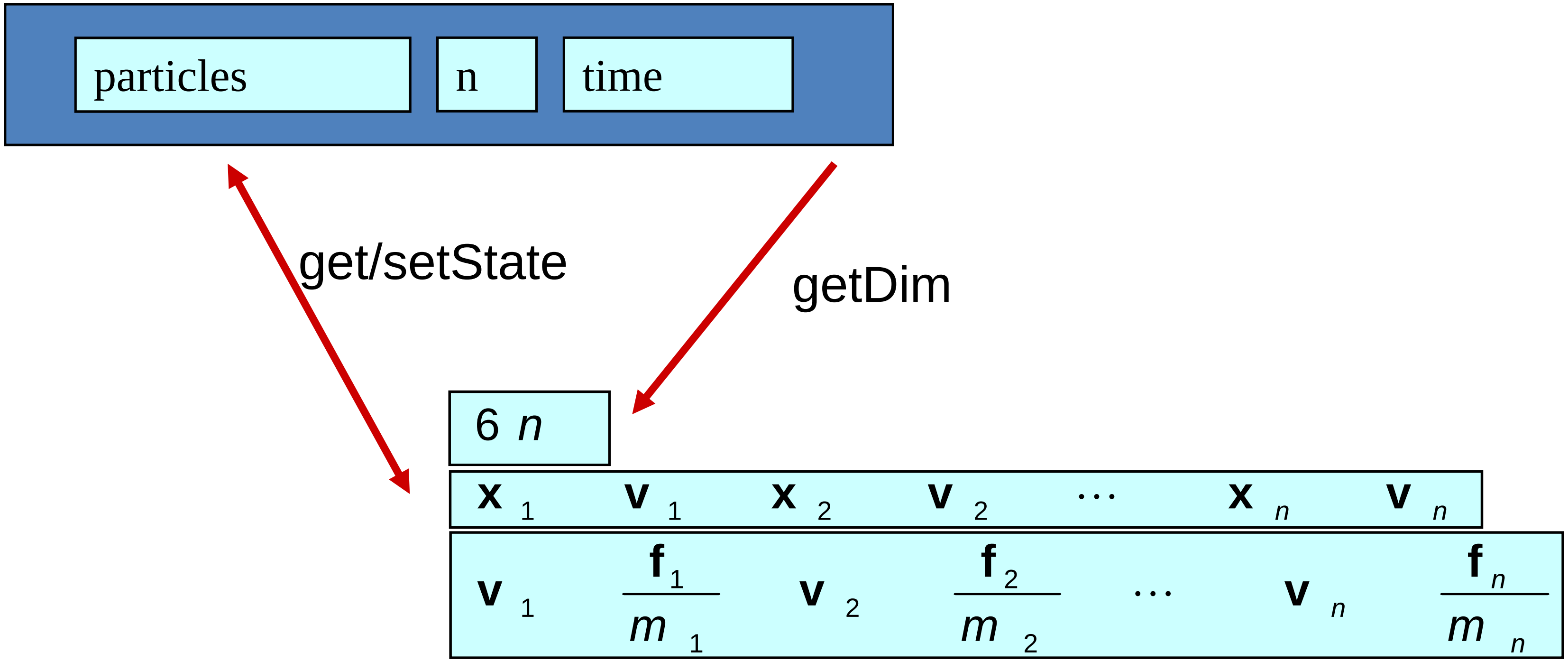
For *n* particles, the solver interface now looks like:




```
int ParticleDims(ParticleSystem p){
return(6 * p->n);
};
```

Particle system solver interface

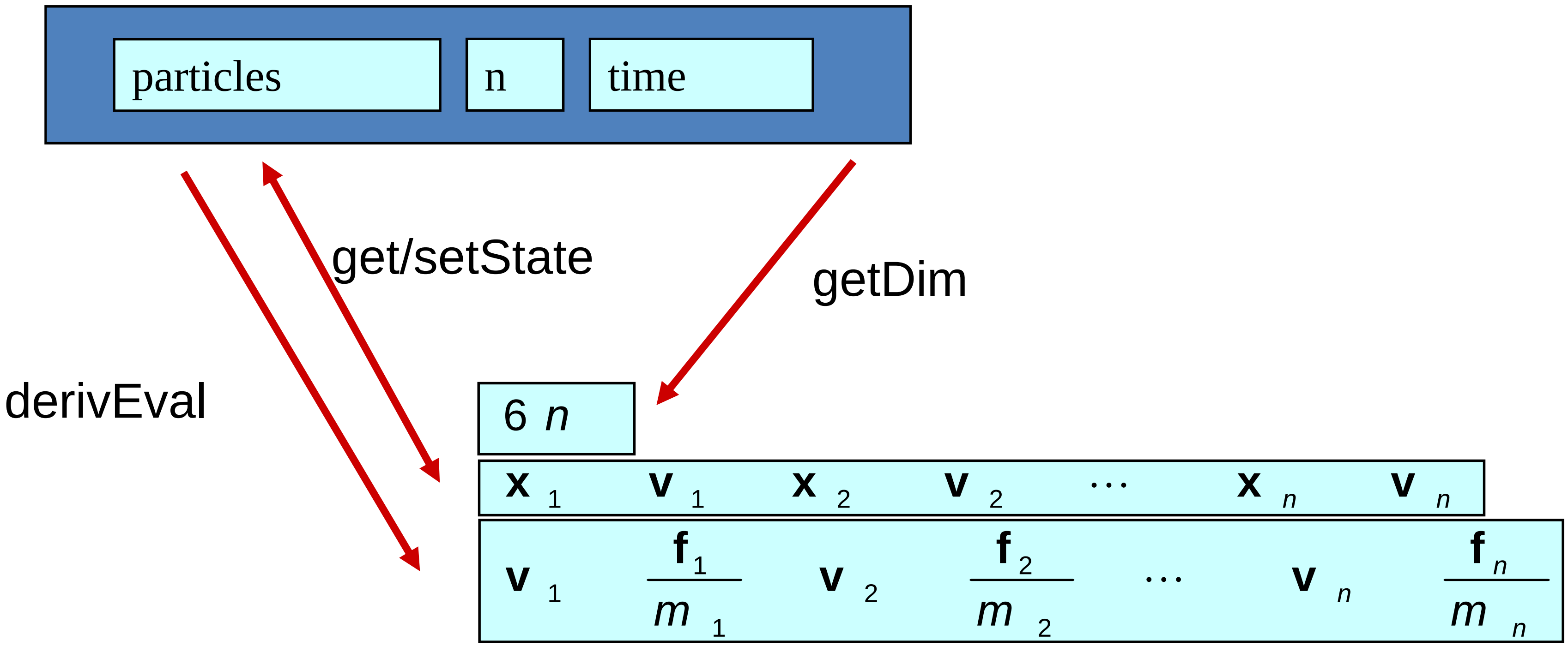
For *n* particles, the solver interface now looks like:




```
int ParticleGetState(ParticleSystem p, float *dst){
for(int i=0; i < p->n; i++){
*(dst++) = p->p[i]->x[0];      *(dst++) = p->p[i]->x[1];      *(dst++) = p->p[i]->x[2];
*(dst++) = p->p[i]->v[0];      *(dst++) = p->p[i]->v[1];      *(dst++) = p->p[i]->v[2];
}
}
```

Particle system solver interface

For *n* particles, the solver interface now looks like:



Particle system diff. eq. solver

We can solve the evolution of a particle system again using the Euler method:

$$\begin{bmatrix} \mathbf{x}_1^{i+1} \\ \mathbf{v}_1^{i+1} \\ \vdots \\ \mathbf{x}_n^{i+1} \\ \mathbf{v}_n^{i+1} \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1^{i} \\ \mathbf{v}_1^{i} \\ \vdots \\ \mathbf{x}_n^{i} \\ \mathbf{v}_n^{i} \end{bmatrix} + \Delta t \begin{bmatrix} \mathbf{v}_1^{i} \\ \mathbf{f}_1^{i} / m_1 \\ \vdots \\ \mathbf{v}_n^{i} \\ \mathbf{f}_n^{i} / m_n \end{bmatrix}$$

```
void EulerStep(ParticleSystem p, float DeltaT){
	ParticleDeriv(p,temp1); /* get deriv */
	ScaleVector(temp1,DeltaT) /* scale it */
	ParticleGetState(p,temp2); /* get state */
	AddVectors(temp1,temp2,temp2); /* add -> temp2 */
	ParticleSetState(p,temp2); /* update state */
	p->t += DeltaT; /* update time */
}
```

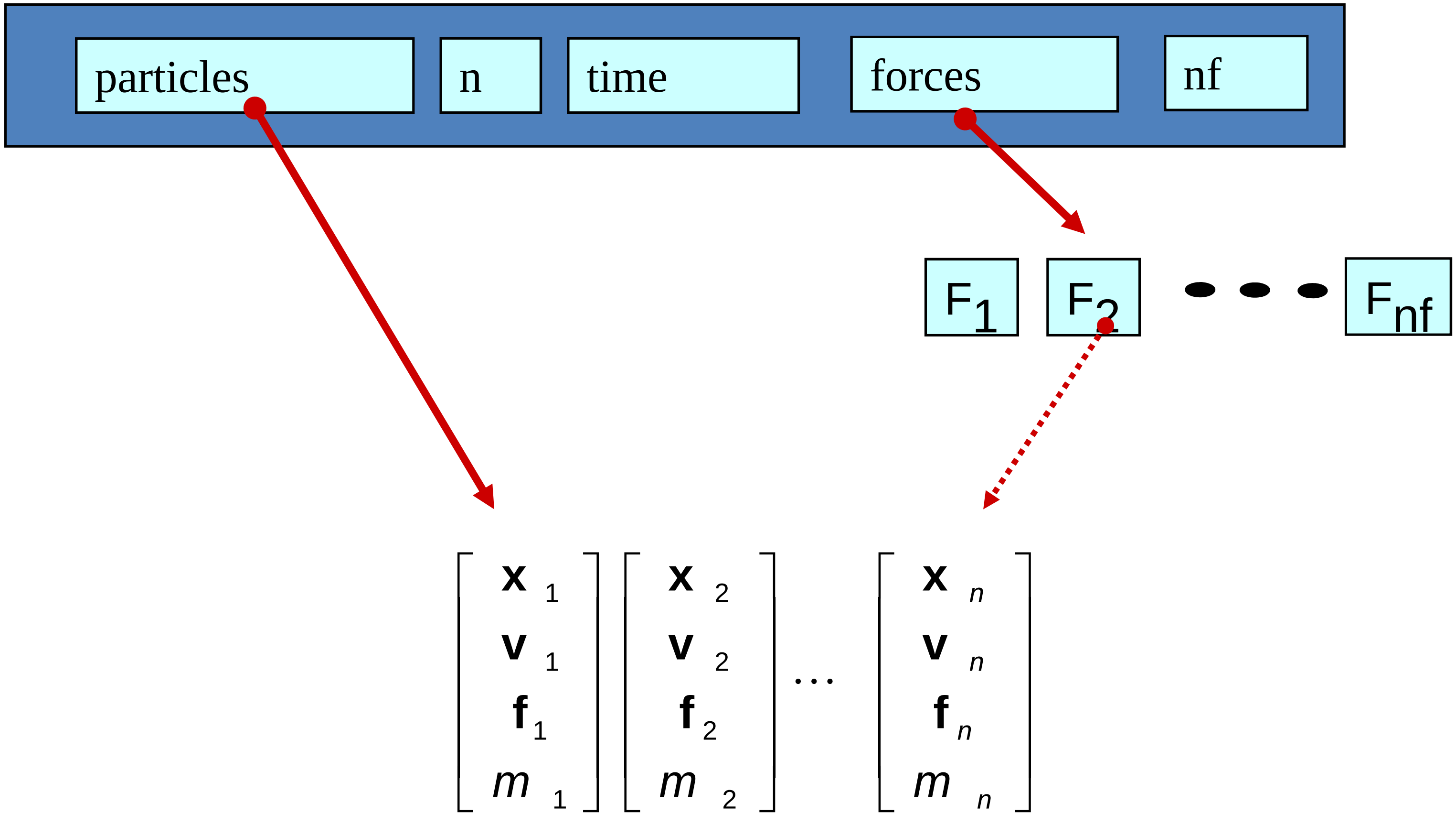
Forces

- Each particle can experience a force which sends it on its merry way.
- Where do these forces come from? Some examples:
  - Constant (gravity)
  - Position/time dependent (force fields)
  - Velocity-dependent (drag)
  - N-ary (springs)

- How do we compute the net force on a particle?

Particle systems with forces

- Force objects are black boxes that point to the particles they influence and add in their contributions.
- We can now visualize the particle system with force objects:



Gravity and viscous drag

The force due to **gravity** is simply:

$$\mathbf{f}_{grav} = m\,\mathbf{G}$$

Often, we want to slow things down with **viscous drag**:

$$\mathbf{f}_{drag} = -k_{drag}\,\mathbf{v}$$

Damped spring

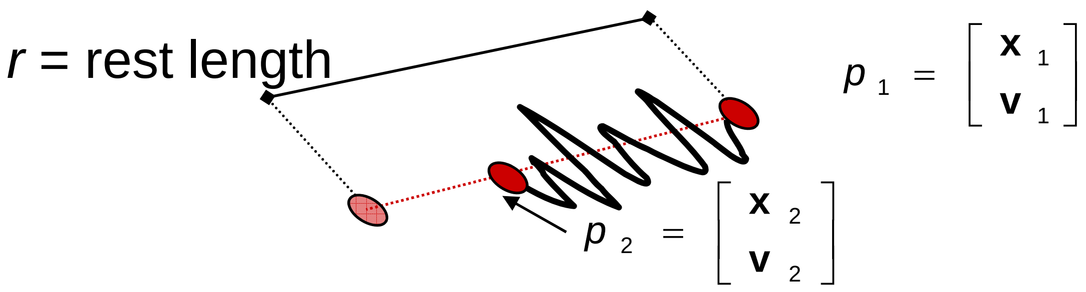
A spring is a simple examples of an "N-ary" force.

Recall the equation for the force due to a spring:

$$f = -k_{spring}(x - r)$$

We can augment this with damping:

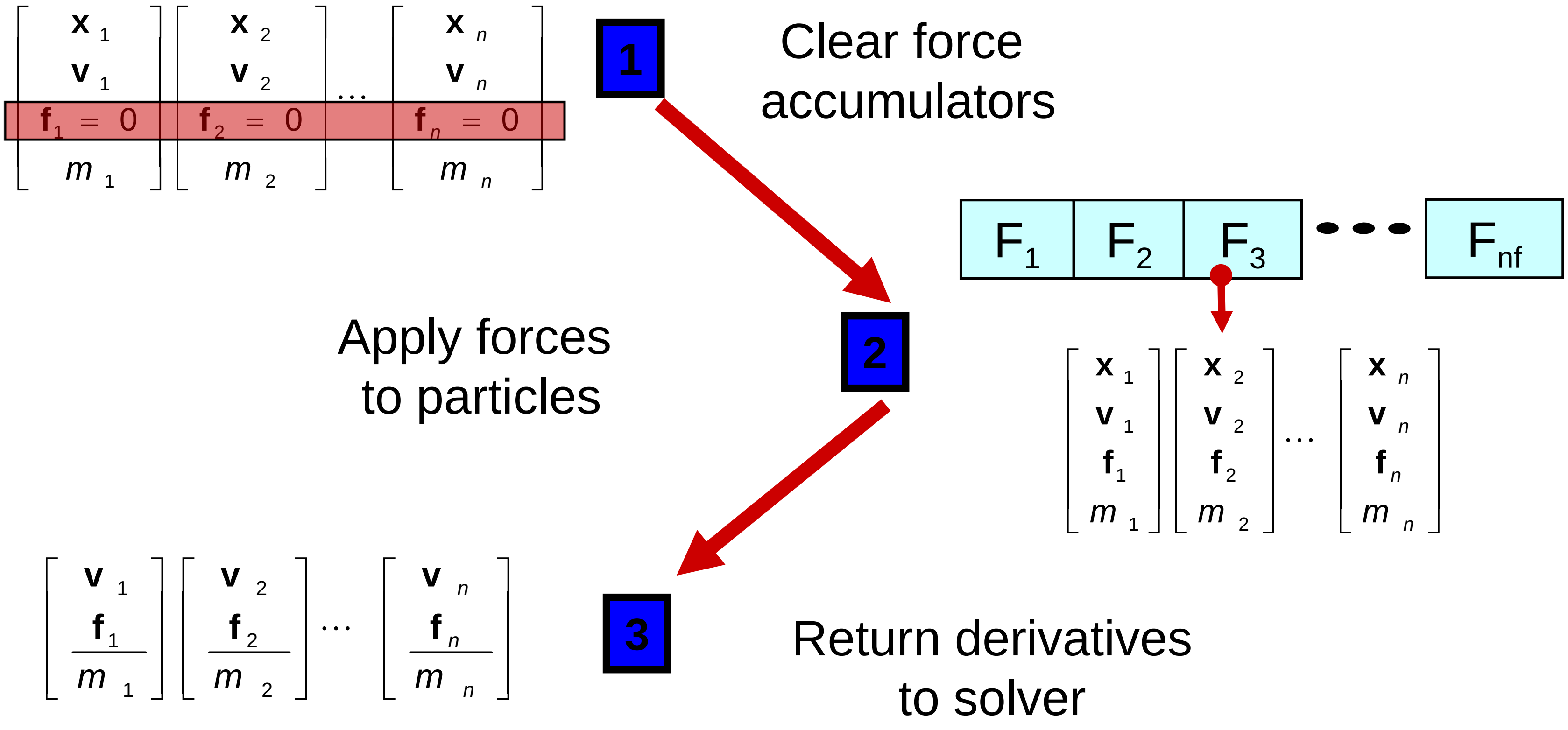
$$f = -[k_{spring}(x - r) + k_{damp} v]$$



Note: stiff spring systems can be very unstable under Euler integration. Simple solutions include heavy damping (may not look good), tiny time steps (slow), or better integration (Runge-Kutta is straightforward).

derivEval

1. Clear forces
   - Loop over particles, zero force accumulators
2. Calculate forces
   - Sum all forces into accumulators
3. Return derivatives
   - Loop over particles, return **v** and **f**/*m*



Particle system solver interface

```
int ParticleDerivative(ParticleSystem p, float *dst){
	Clear_Forces(p); /* zero the force accumulators */
	Compute_Forces(p); /* magic force function */
	for(int i=0; i < p->n; i++){
		*(dst++) = p->p[i]->v[0]; /* xdot = v */
		*(dst++) = p->p[i]->v[1];
		*(dst++) = p->p[i]->v[2];
		*(dst++) = p->p[i]->f[0]/m; /* vdot = f/m */
		*(dst++) = p->p[i]->f[1]/m;
		*(dst++) = p->p[i]->f[2]/m;
	}
}
```

Particle system diff. eq. solver

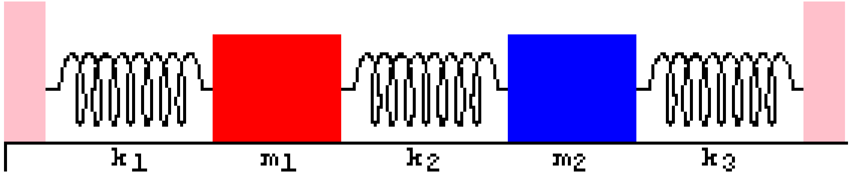
We can solve the evolution of a particle system again using the Euler method:

$$\begin{bmatrix} \mathbf{x}_1^{i+1} \\ \mathbf{v}_1^{i+1} \\ \vdots \\ \mathbf{x}_n^{i+1} \\ \mathbf{v}_n^{i+1} \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1^{i} \\ \mathbf{v}_1^{i} \\ \vdots \\ \mathbf{x}_n^{i} \\ \mathbf{v}_n^{i} \end{bmatrix} + \Delta t \begin{bmatrix} \mathbf{v}_1^{i} \\ \mathbf{f}_1^{i} / m_1 \\ \vdots \\ \mathbf{v}_n^{i} \\ \mathbf{f}_n^{i} / m_n \end{bmatrix}$$

```
void EulerStep(ParticleSystem p, float DeltaT){
        ParticleDeriv(p,temp1); /* get deriv */
        ScaleVector(temp1,DeltaT) /* scale it */
        ParticleGetState(p,temp2); /* get state */
        AddVectors(temp1,temp2,temp2); /* add -> temp2 */
        ParticleSetState(p,temp2); /* update state */
        p->t += DeltaT; /* update time */
}
```

Particle system diff. eq. solver

We can solve the evolution of a particle system again using the Euler method:

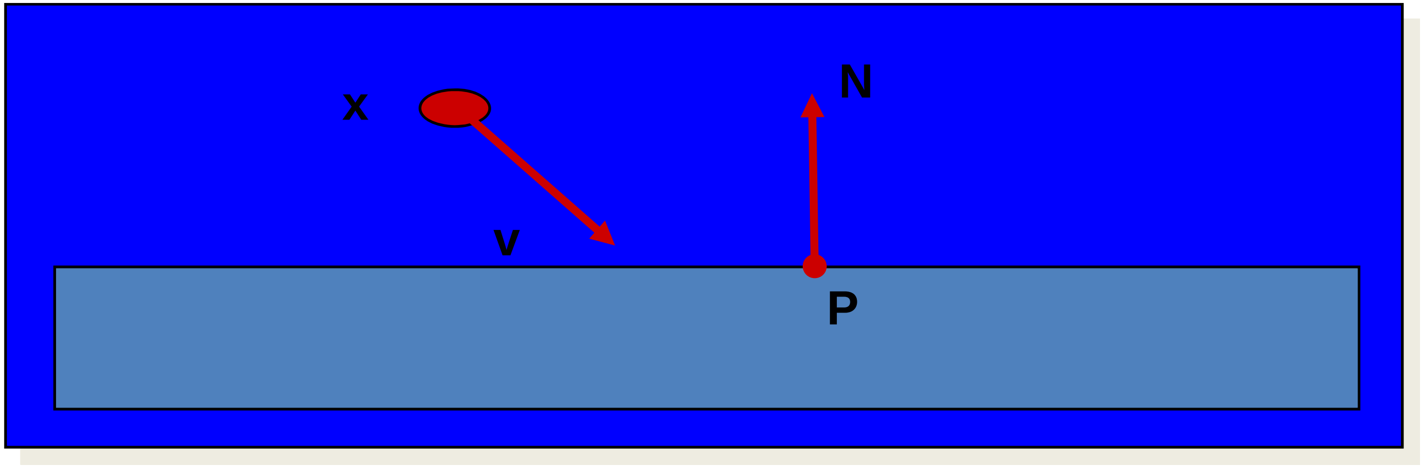


$$\begin{bmatrix} \mathbf{x}_1^{i+1} \\ \mathbf{v}_1^{i+1} \\ \vdots \\ \mathbf{x}_n^{i+1} \\ \mathbf{v}_n^{i+1} \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1^{i} \\ \mathbf{v}_1^{i} \\ \vdots \\ \mathbf{x}_n^{i} \\ \mathbf{v}_n^{i} \end{bmatrix} + \Delta t \begin{bmatrix} \mathbf{v}_1^{i} \\ \mathbf{f}_1^{i} / m_1 \\ \vdots \\ \mathbf{v}_n^{i} \\ \mathbf{f}_n^{i} / m_n \end{bmatrix}$$

```
void EulerStep(ParticleSystem p, float DeltaT){
	ParticleDeriv(p,temp1); /* get deriv */
	ScaleVector(temp1,DeltaT) /* scale it */
	ParticleGetState(p,temp2); /* get state */
	AddVectors(temp1,temp2,temp2); /* add -> temp2 */
	ParticleSetState(p,temp2); /* update state */
	p->t += DeltaT; /* update time */
}
```

Bouncing off the walls

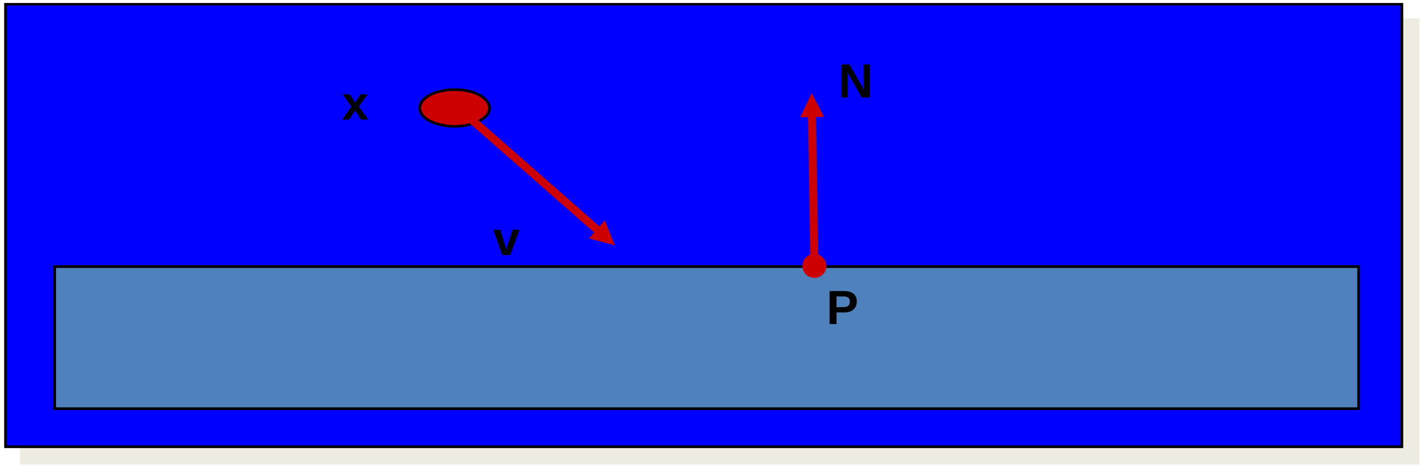
- Handling collisions is a useful add-on for a particle simulator.
- For now, we'll just consider simple point-plane collisions.



A plane is fully specified by any point **P** on the plane and its normal **N**.

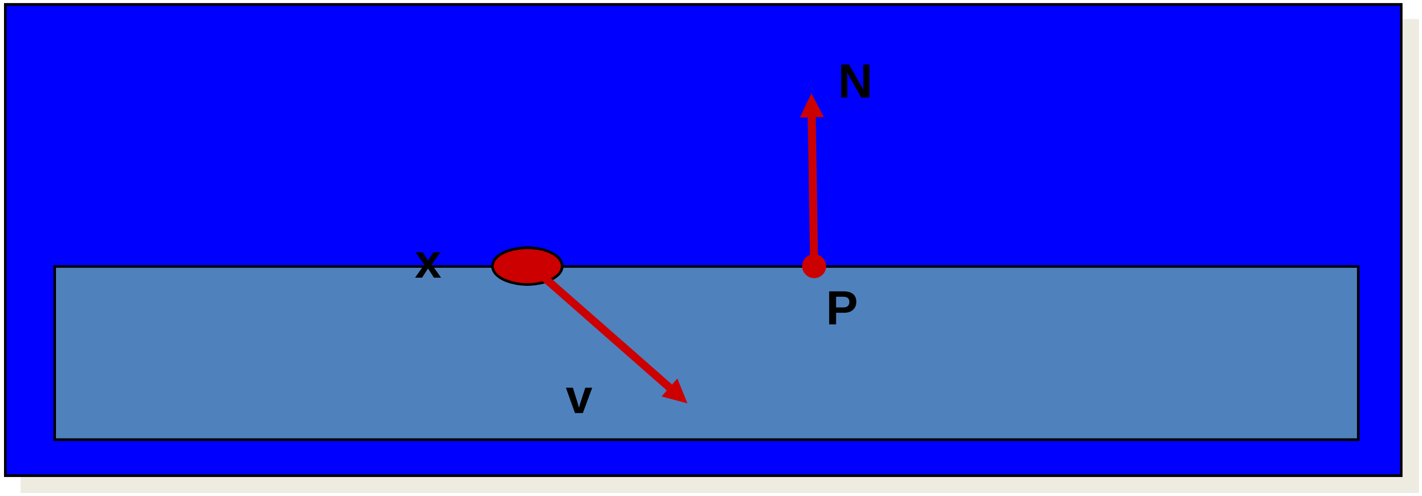
Collision Detection

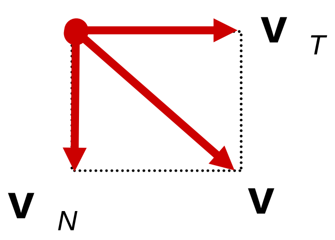
How do you decide when you've made **exact** contact with the plane?



Normal and tangential velocity

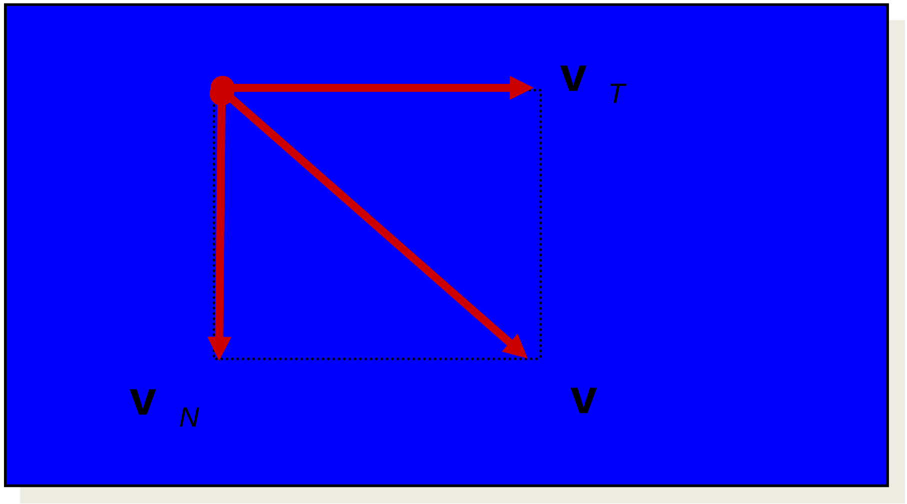
To compute the collision response, we need to consider the normal and tangential components of a particle's velocity.



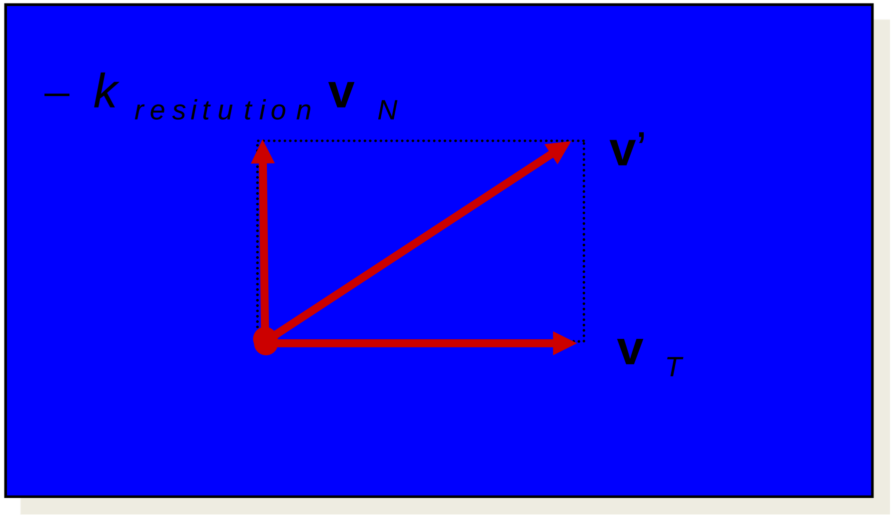


$$\mathbf{v}_N = (\mathbf{N} \cdot \mathbf{v})\mathbf{N}$$
$$\mathbf{v}_T = \mathbf{v} - \mathbf{v}_N$$

Collision Response



before



after

The response to collision is then to immediately replace the current velocity with a new velocity:

$$\mathbf{v}' = \mathbf{v}_T - k_{restitution}\mathbf{v}_N$$

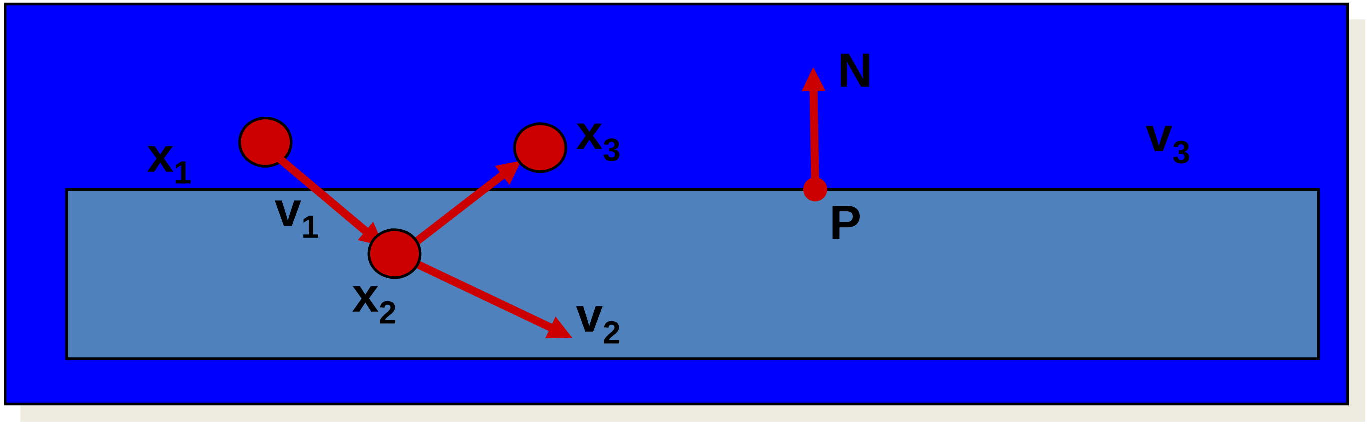
The particle will then move according to this velocity in the next timestep.

Collision without contact

- In general, we don’t sample moments in time when particles are in *exact* contact with the surface.
- There are a variety of ways to deal with this problem.
- A simple alternative is to determine if a collision must have occurred in the past, and then pretend that you’re currently in exact contact.

Very simple collision response

- How do you decide when you've had a collision?

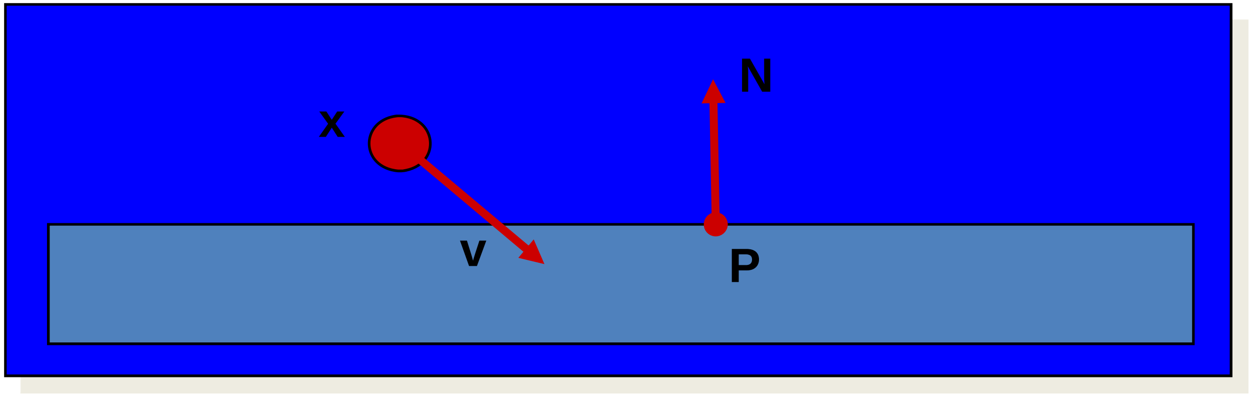


A problem with this approach is that particles will disappear under the surface.

Also, the response may not be enough to bring a particle to the other side of a wall.

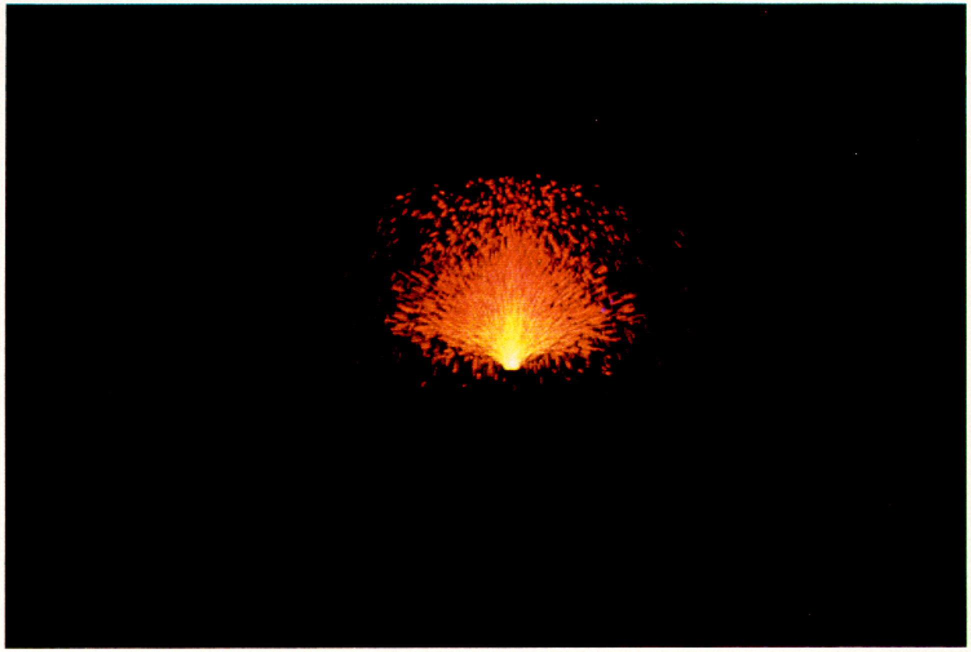
More complicated collision response

- Another solution is to modify the update scheme to:
  - detect the future time and point of collision
  - reflect the particle within the time-step



Generate Particles

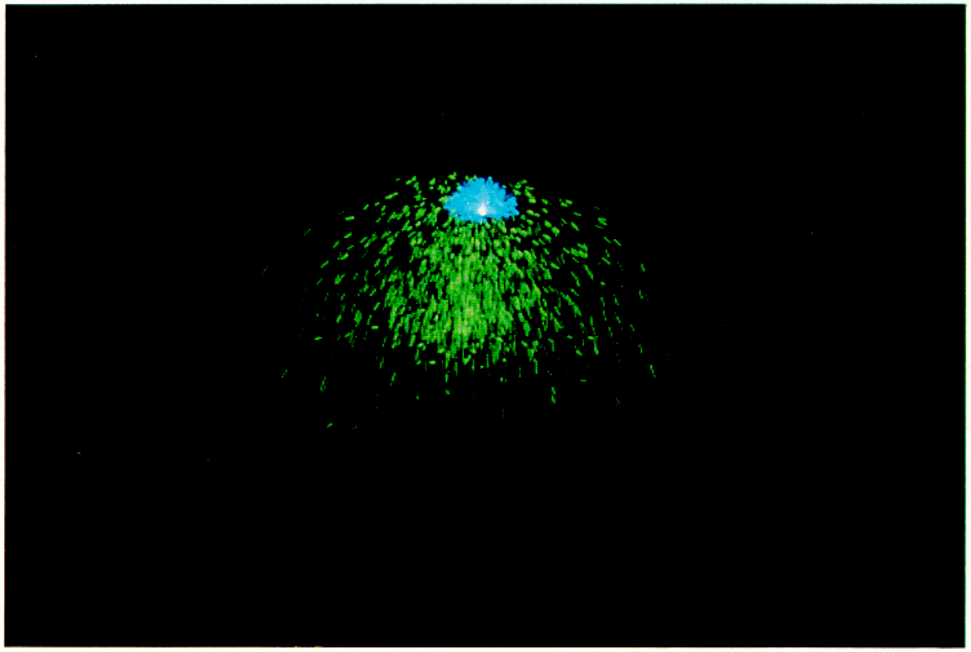
- Particle Attributes
  - initial position,
  - initial velocity (both speed and direction),
  - initial size,
  - initial color,
  - initial transparency,
  - shape,
  - lifetime.



WILLIAM T. REEVES, ACM Transactions on Graphics, Vol. 2, No. 2, April 1983

Generate Particles

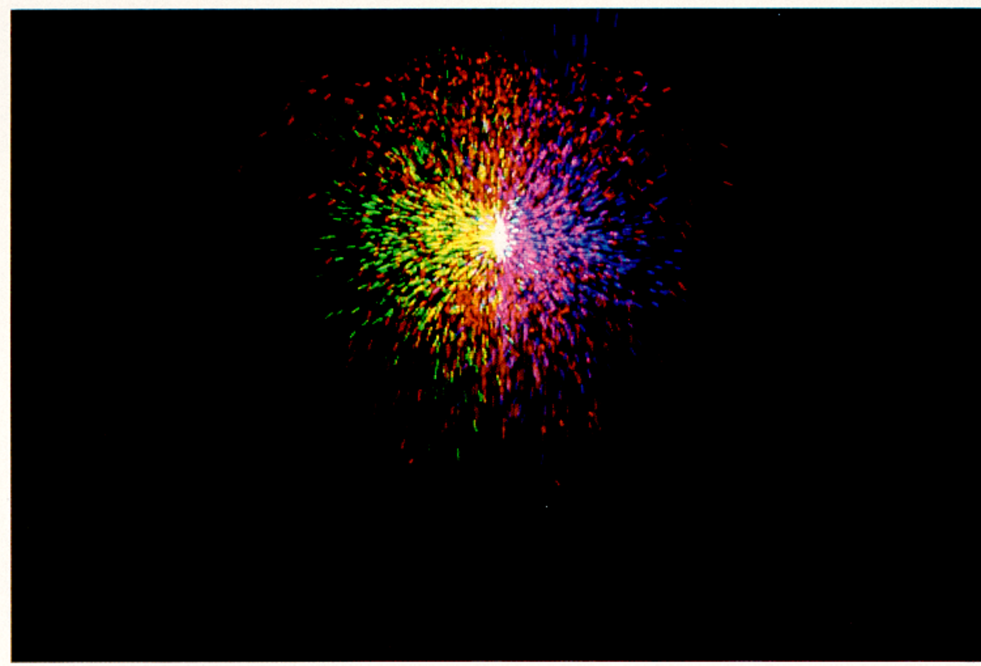
- Particle Attributes
  - initial position,
  - initial velocity (both speed and direction),
  - initial size,
  - initial color,
  - initial transparency,
  - shape,
  - lifetime.



WILLIAM T. REEVES, ACM Transactions on Graphics, Vol. 2, No. 2, April 1983

Generate Particles

- Particle Attributes
  - initial position,
  - initial velocity (both speed and direction),
  - initial size,
  - initial color,
  - initial transparency,
  - shape,
  - lifetime.



WILLIAM T. REEVES, ACM Transactions on Graphics, Vol. 2, No. 2, April 1983

Generate Particles

- Initial Particle Distribution

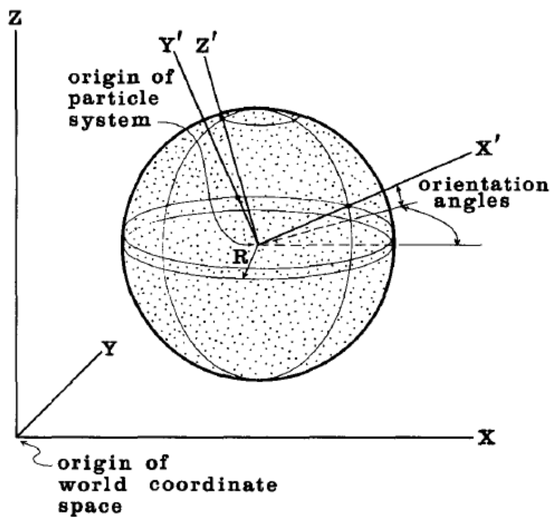
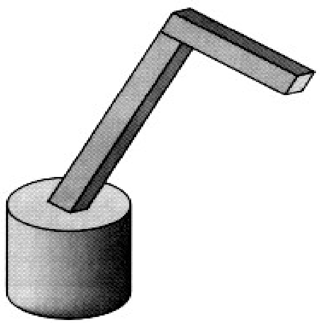


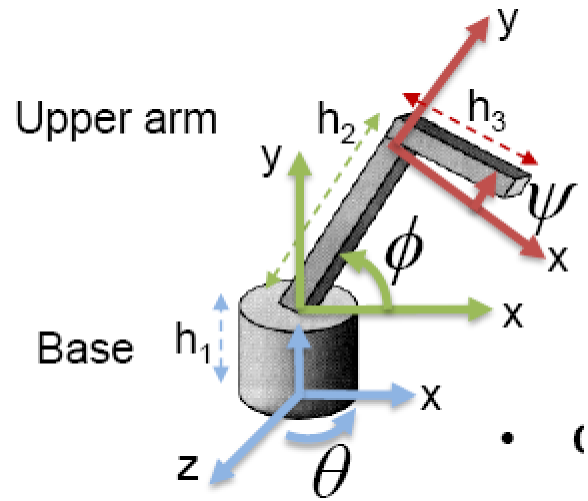
Fig. 1. Typical particle system with spherical generation shape.

- Particle hierarchy, for example
  - Skyrocket : firework
  - Clouds : water drops

Throwing a ball from a robot arm

- Let's say we had our robot arm example and we wanted to launch particles from its tip.





- How would we calculate initial speed?

  Q=R(theta)*T1*R(phi)*T2*R(psi)*P

  We want dQ/dt

Principles of Animation

- **Goal**: make characters that move in a convincing way to communicate personality and mood.

- Walt Disney developed a number of principles.
  - ~1930

- Computer graphics animators have adapted them to 3D animation.

John Lasseter. Principles of traditional animation applied to 3D computer animation. Proceedings of SIGGRAPH (Computer Graphics) 21(4): 35-44, July 1987.

Principles of Animation

- The following are a set of principles to keep in mind:
  1. Squash and stretch
  2. Staging
  3. Timing
  4. Anticipation
  5. Follow through
  6. Secondary action
  7. Straight-ahead vs. pose-to-pose vs. blocking
  8. Arcs
  9. Slow in, slow out
  10. Exaggeration
  11. Appeal

Squash and stretch

- **Squash**: flatten an object or character by pressure or by its own power.
- **Stretch**: used to increase the sense of speed and emphasize the squash by contrast.
- Note: keep volume constant!

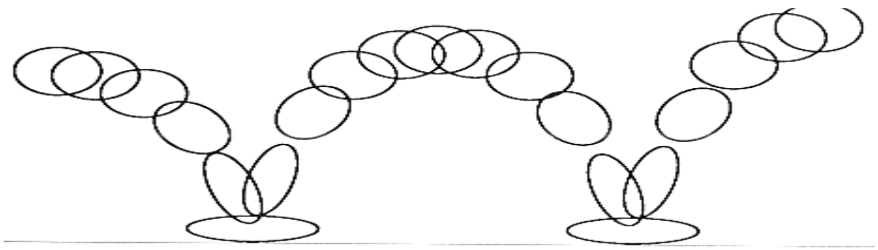


FIGURE 2. Squash & stretch in bouncing ball.

- http://www.siggraph.org/education/materials/HyperGraph/animation/character_animation/principles/squash_and_stretch.htm
- http://www.siggraph.org/education/materials/HyperGraph/animation/character_animation/principles/bouncing_ball_example_of_slow_in_out.htm

Squash and stretch (cont’d)

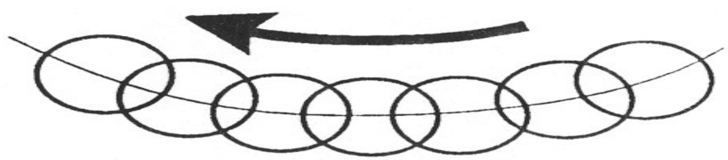


FIGURE 4a. In slow action, an object's position overlaps from frame to frame which gives the action a smooth appearance to the eye.

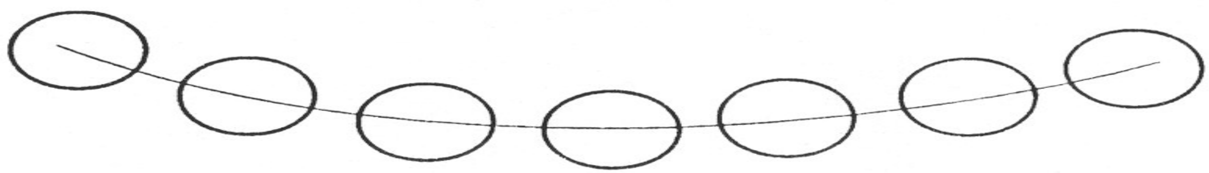


FIGURE 4b. Strobing occurs in a faster action when the object's positions do not overlap and the eye perceives seperate images.

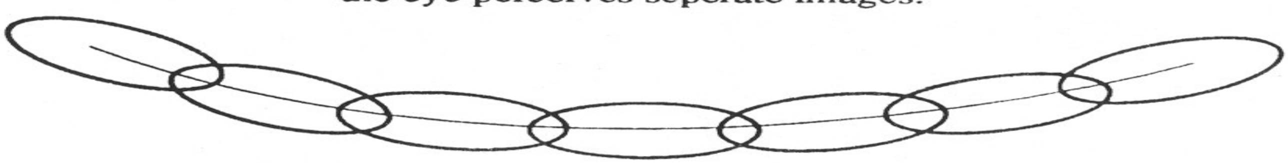
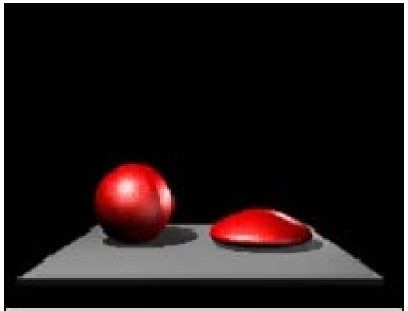
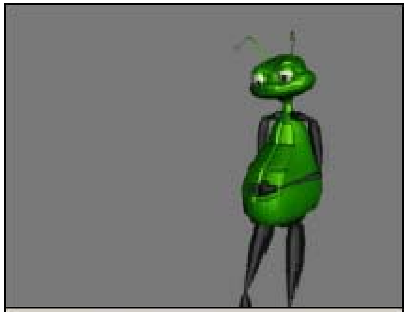


FIGURE 4c. Stretching the object so that it's positions overlap again will relieve the strobing effect.

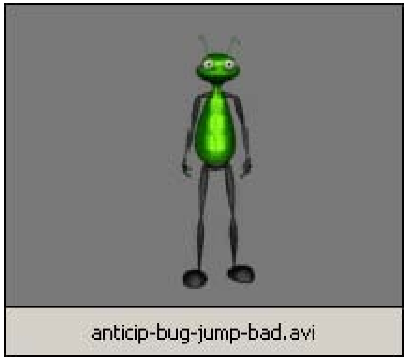
Squash and stretch (cont'd)

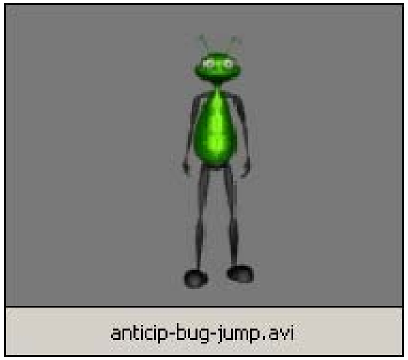

squash-ball.avi


bug-squash-tummy.avi

Anticipation

- An action has three parts: anticipation, action, reaction.
- Anatomical motivation: a muscle must extend before it can contract.

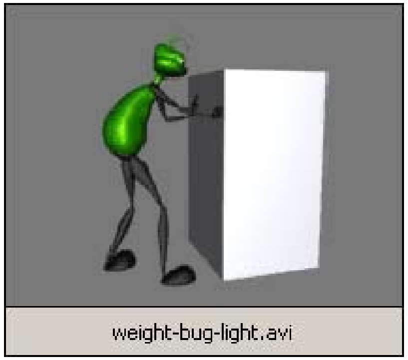


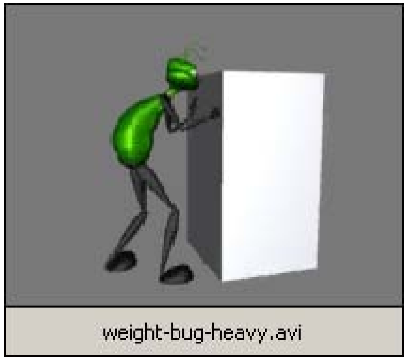


- Watch: bugs-bunny.virtualdub.new.mpg
- Prepares audience for action so they know what to expect.
- Directs audience's attention.

Anticipation (cont’d)

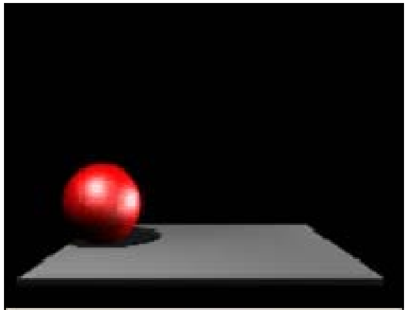
- Amount of anticipation (combined with timing) can affect perception of speed or weight.

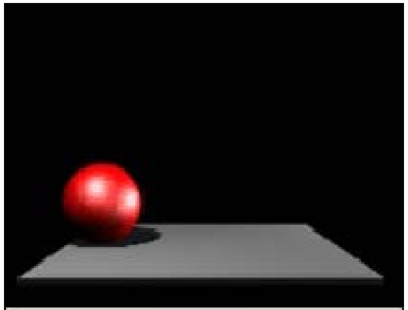




Arcs

- Avoid straight lines since most things in nature move in arcs.


arc-ball-bad.avi

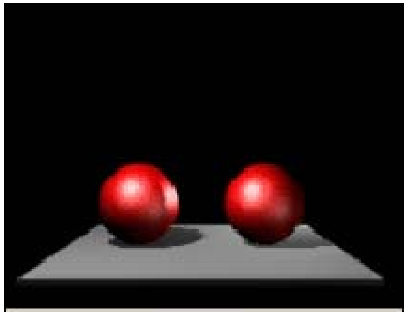

arc-ball-good.avi

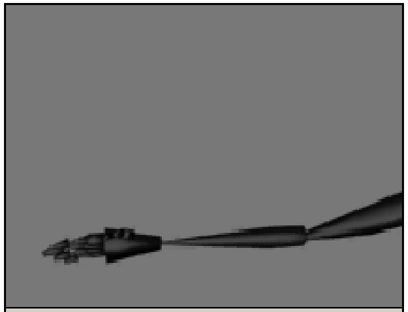

arc-headturn-bad.avi

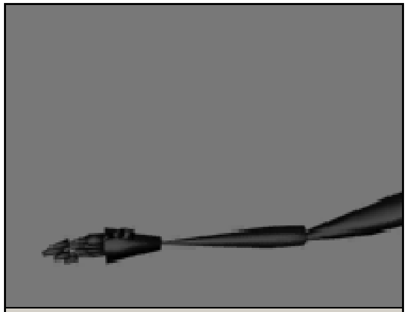

arc-headturn-good.avi

Slow in and slow out

- An extreme pose can be emphasized by slowing down as you get to it (and as you leave it).
- In practice, many things do not move abruptly but start and stop gradually.

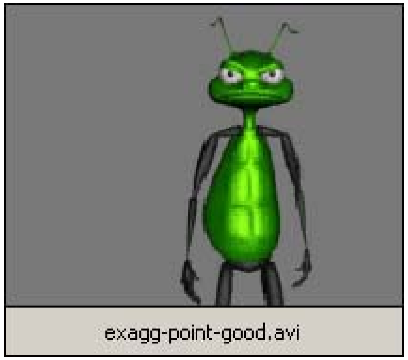

ease-ball.avi


ease-arm-bad.avi


ease-arm-good.avi

Exaggeration

- Get to the heart of the idea and emphasize it so the audience can see it.



Exaggeration

- Get to the heart of the idea and emphasize it so the audience can see it.

Appeal

- The character must interest the viewer.
- It doesn't have to be cute and cuddly.
- Design, simplicity, behavior all affect appeal.
- Example: Luxo, Jr. is made to appear childlike.

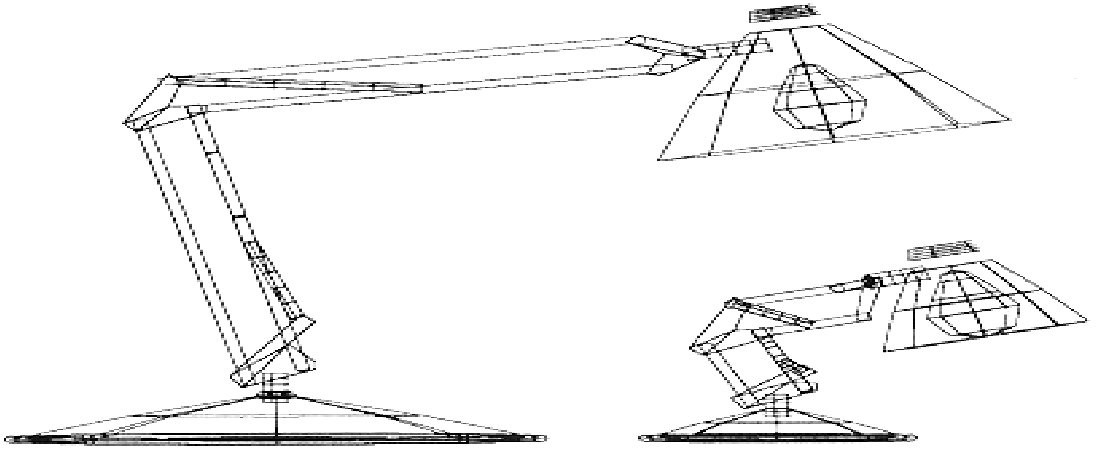
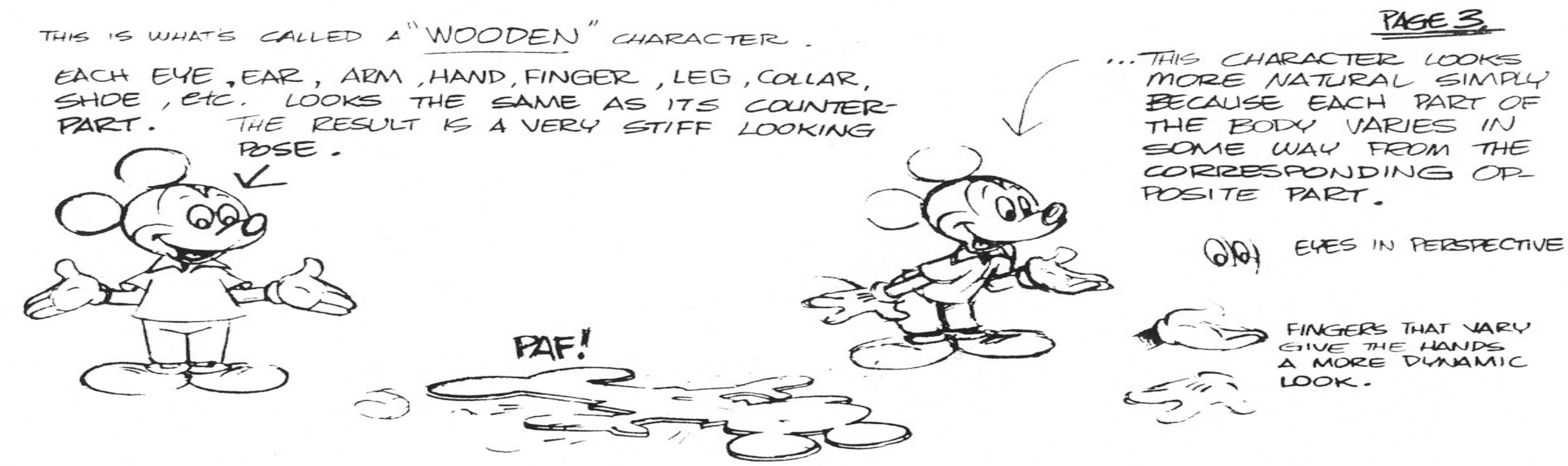


FIGURE 11. Varying the scale of different parts of Dad created the child-like proportions of Luxo Jr.

http://www.youtube.com/watch?v=HDuRXvtImQ0&feature=related

Appeal (cont'd)

- Note: avoid perfect symmetries.



Appeal (cont'd)

- Note: avoid perfect symmetries.