7. Physically Based Animation

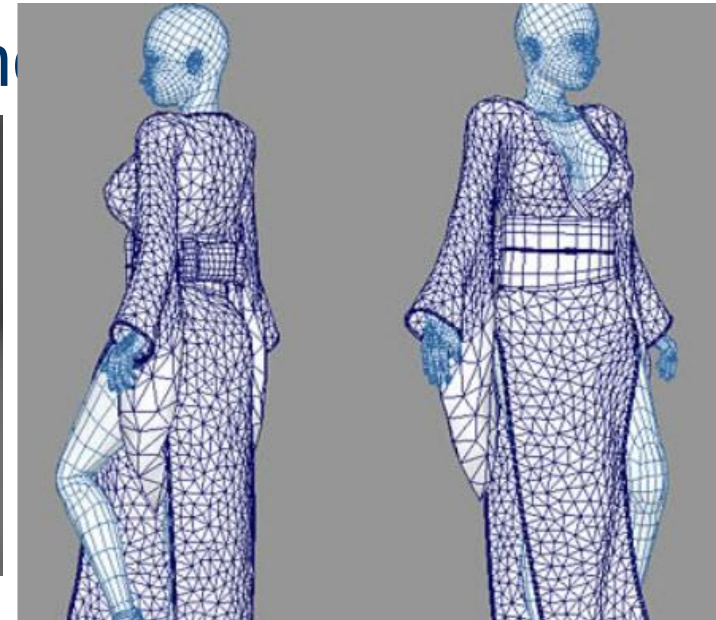
- Concerned with quality of motion than with precisely controlling the position and orientation
- Animation is not necessarily concerned with accuracy, but is concerned with *believability* (sometimes referred to as being *physically realistic*)
- Forces used to maintain relationships among geometric elements might not be physically correct

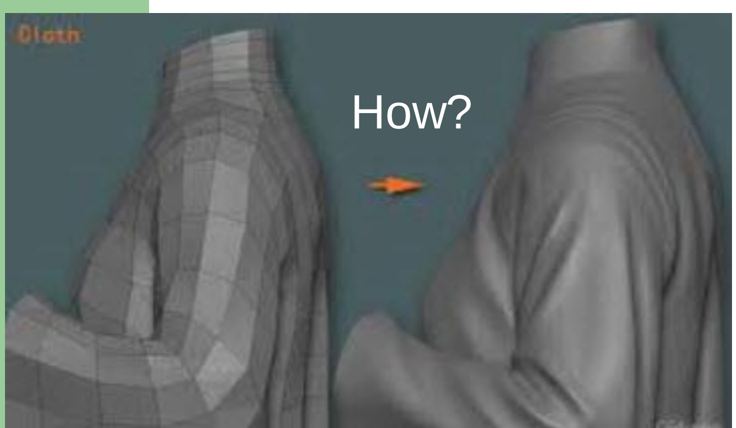
Physically based Animation

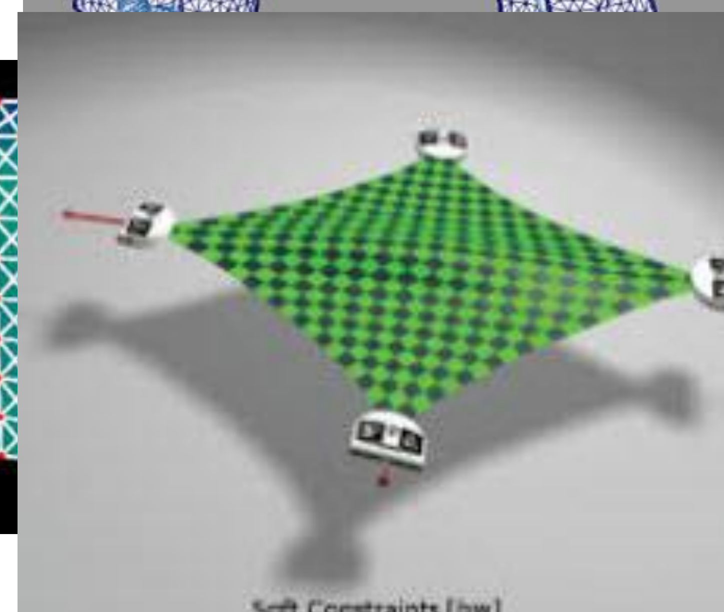
- There are often several levels at which a process can be modeled
  e.g., cloth modeling
  surface level – less expensive, less flexible
  thread level - more expensive, more flexible
- We hope to relieve the animators of low-level specifications of motions, and only be concerned with specifying high-level relationships or qualities of the motion
- Mainly concerned with *dynamic control, basic physics, spring meshes, particle systems, rigid body dynamics, use of constraints*

Cloth Modeling – surface level

- less expensive, less flexibleh







Cloth Modeling – thread level

- more expensive, more flexible

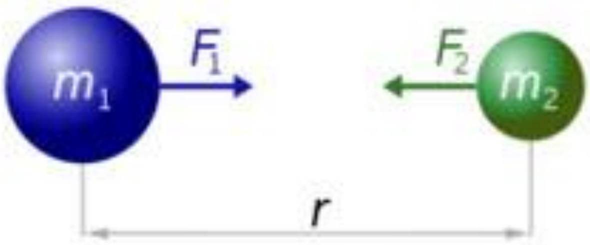
Basic Physics

- Newton's 2nd law of motion: $f = m * a$

- object's new velocity: $v' = v + a * \Delta t$

- object's new location: $p' = p + \frac{1}{2} * (v + v') * \Delta t$

- gravitational force:



$$F_1 = F_2 = G\frac{m_1 \times m_2}{r^2}$$

where $G = 6.67384 \times 10^{-11}\ \mathrm{m^3\,kg^{-1}\,s^{-2}}$

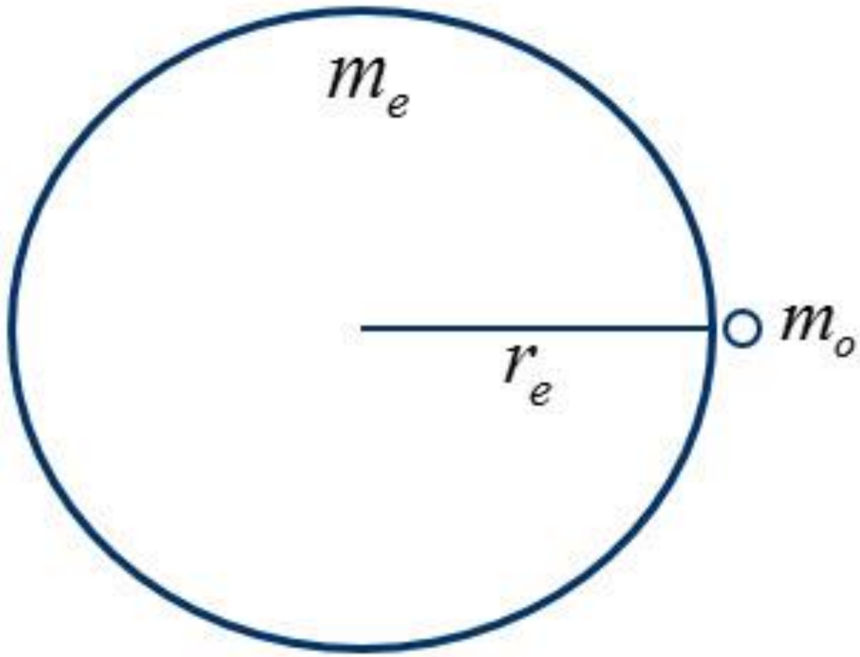
and is called the gravitational constant

Basic Physics

- **gravity model for earth**:

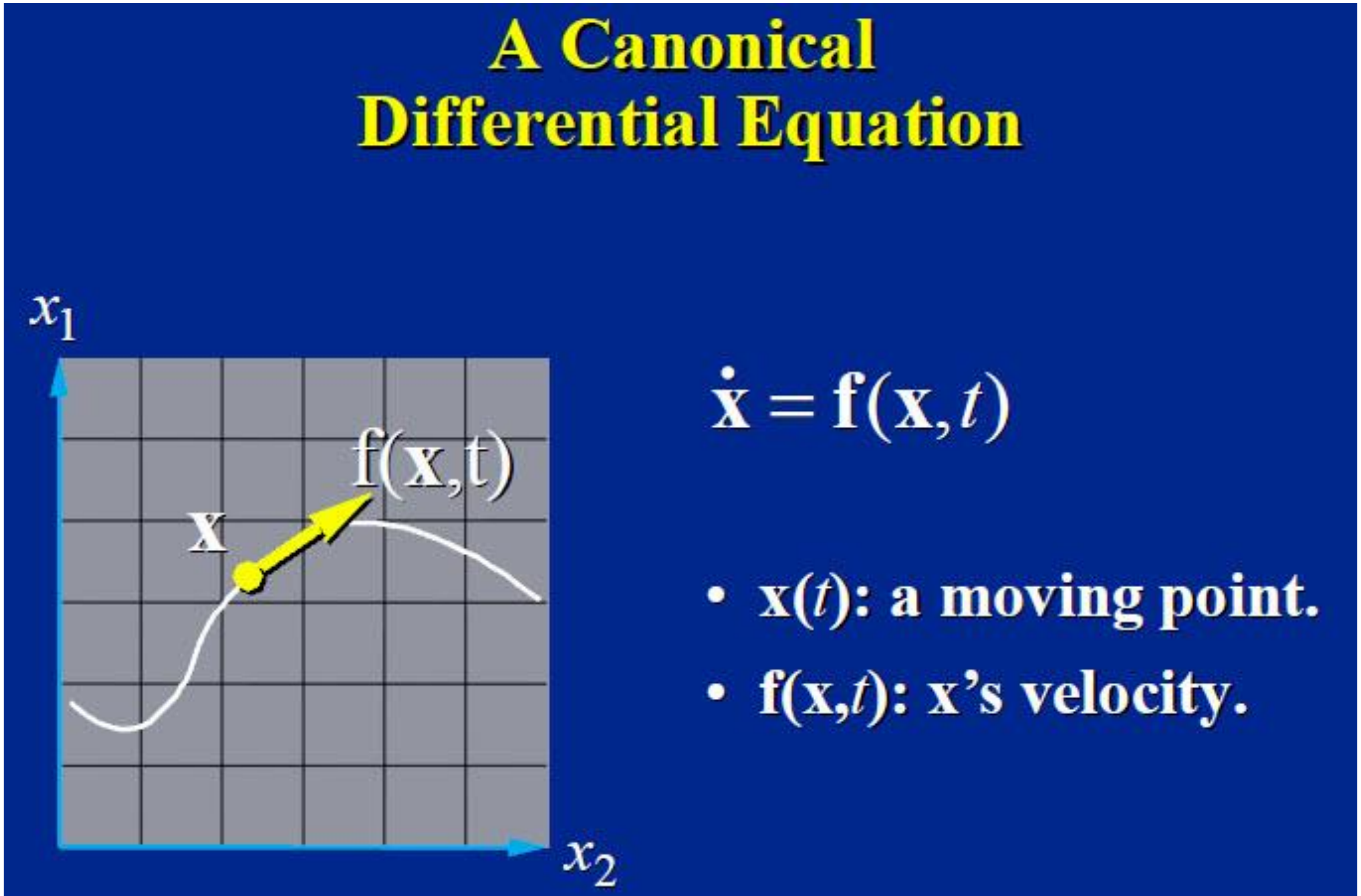
gravitational acceleration a_e is computed as follows:

$$a_e = \frac{f}{m_o} = G\frac{m_e}{r_e^2} = 9.8 \ \frac{\text{meter}}{\text{sec}^2}$$



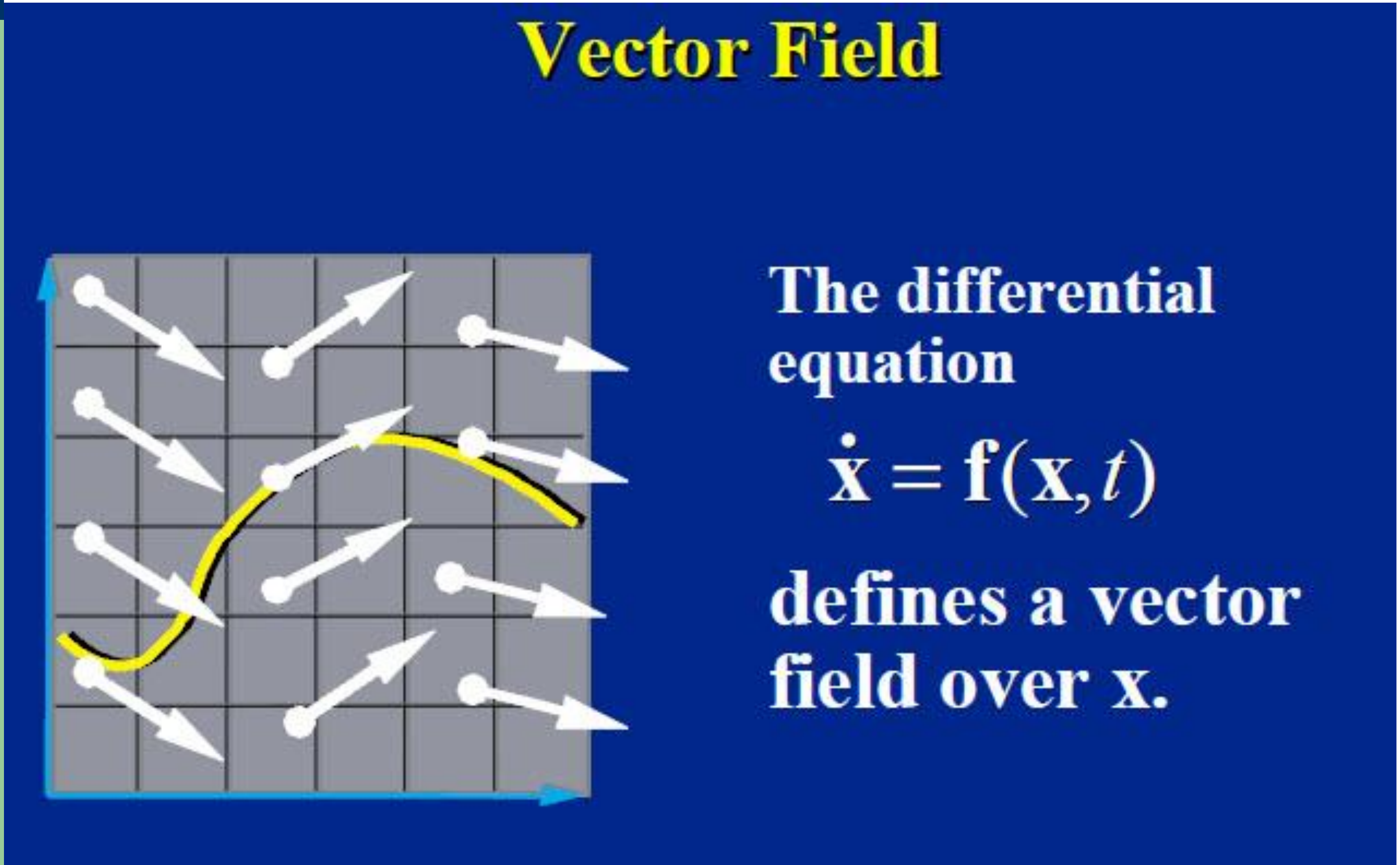
Differential Equation Basics

Initial value problem:



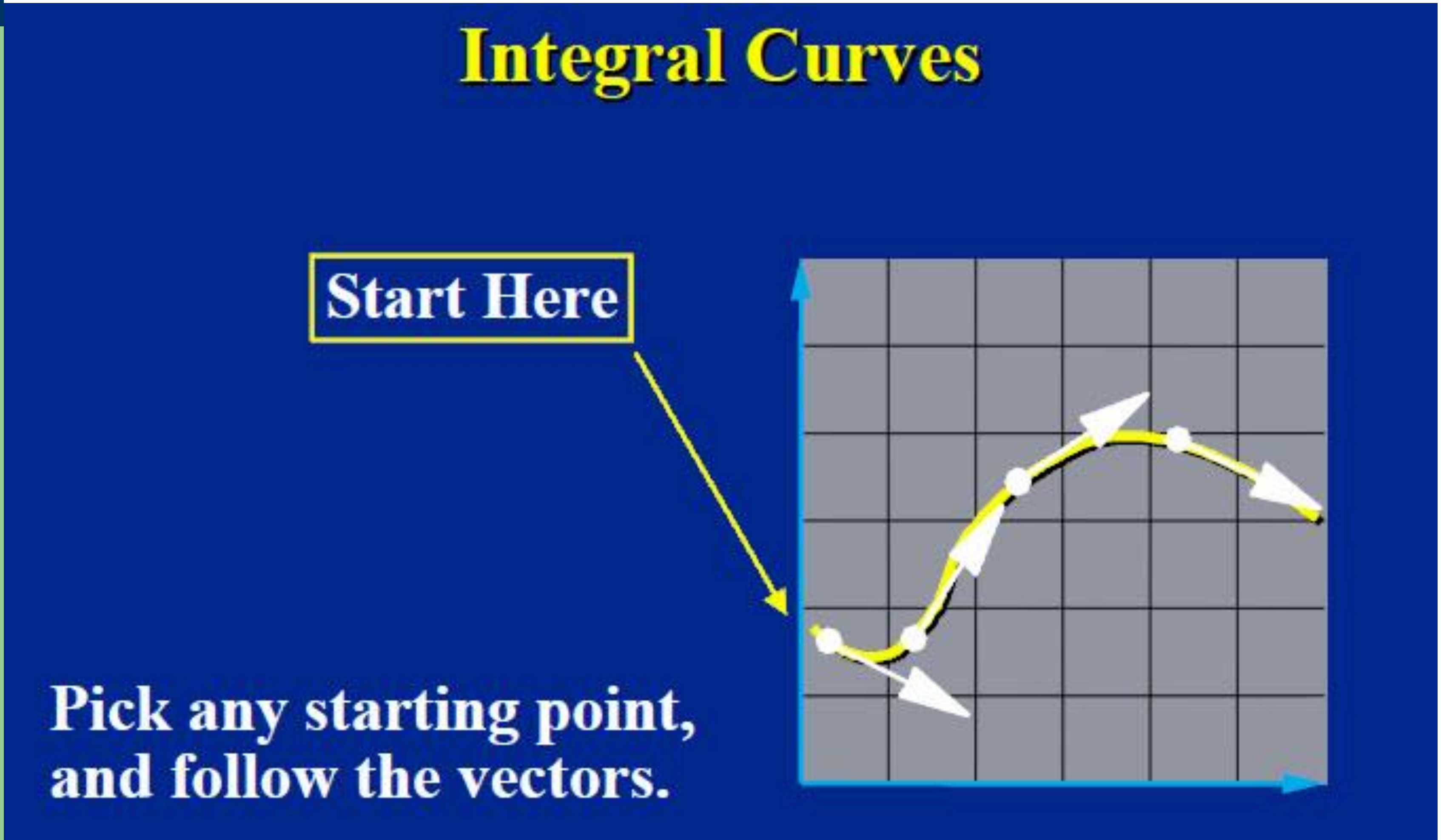
Differential Equation Basics

Initial value problem:



Differential Equation Basics

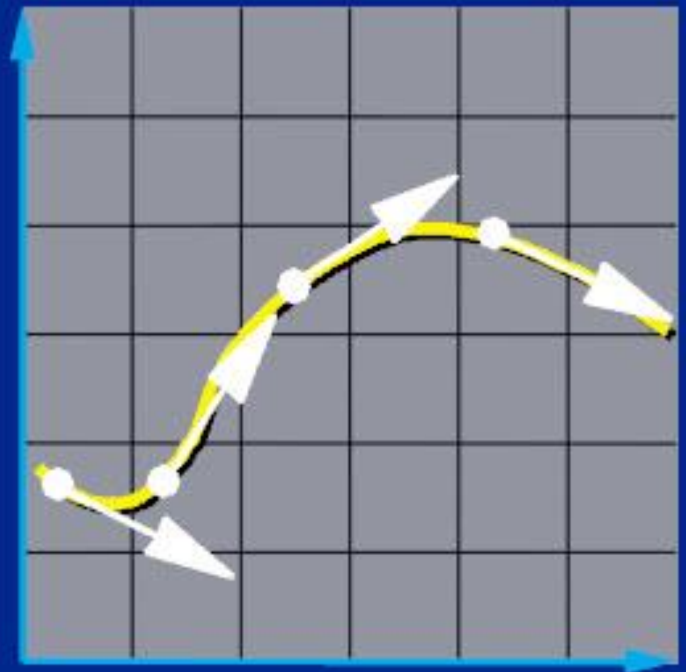
Initial value problem:



Differential Equation Basics

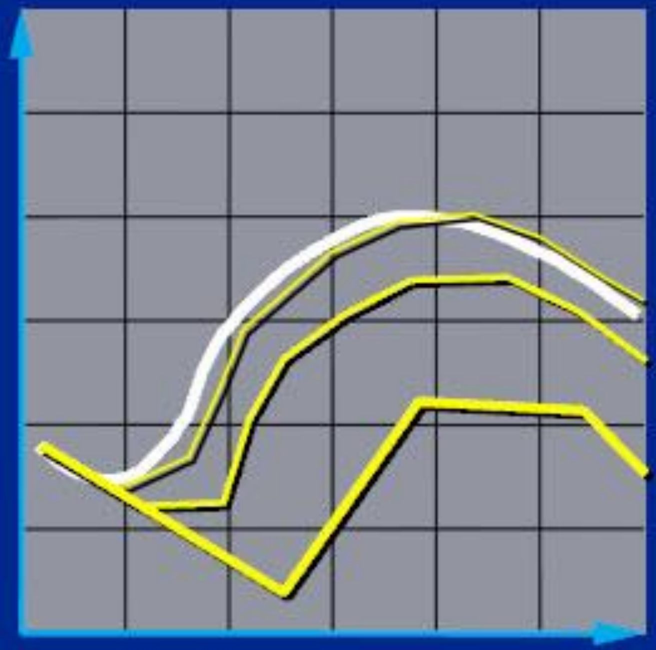
Initial Value Problems

Given the starting point, follow the integral curve.



Differential Equation Basics

Euler's Method



- **Simplest numerical solution method**
- **Discrete time steps**
- **Bigger steps, bigger errors.**

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t\, \mathbf{f}(\mathbf{x}, t)$$

Differential Equation Basics

Problem I: Inaccuracy

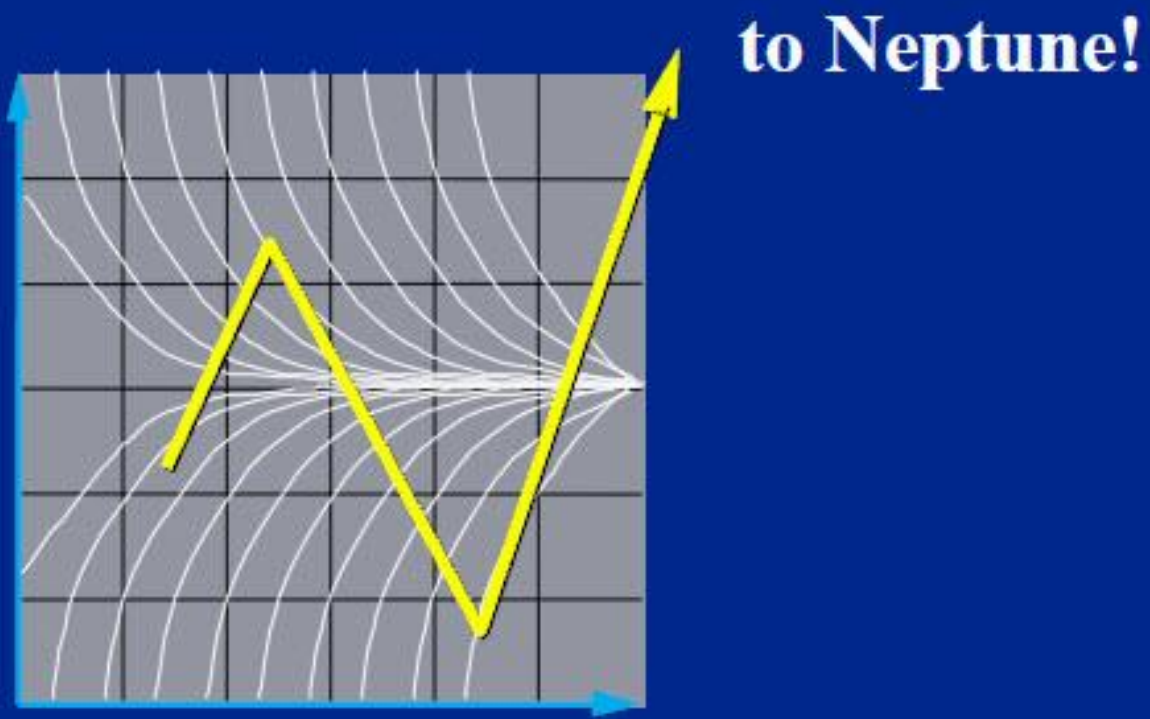
(Shrinking the step size doesn't cure the problem, but only reduces the rate at which the error accumulates.)

Error turns x(t) from a circle into the spiral of your choice.

Differential Equation Basics

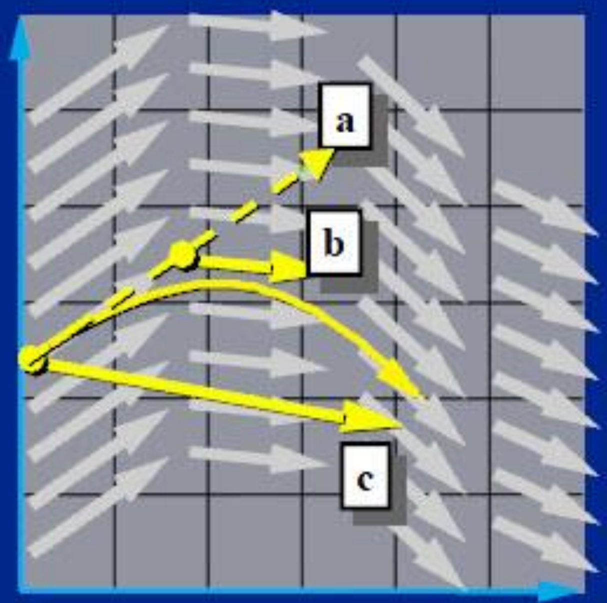
Problem II: Instability

(Too large a step size can make Euler's method diverge)



Differential Equation Basics

The Midpoint Method



a. Compute an Euler step

$$\Delta \mathbf{x} = \Delta t\, \mathbf{f}(\mathbf{x}, t)$$

b. Evaluate f at the midpoint

$$\mathbf{f}_{mid} = \mathbf{f}\left(\frac{\mathbf{x} + \Delta \mathbf{x}}{2}, \frac{t + \Delta t}{2}\right)$$

c. Take a step using the midpoint value

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t\, \mathbf{f}_{mid}$$

Differential Equation Basics

More methods…

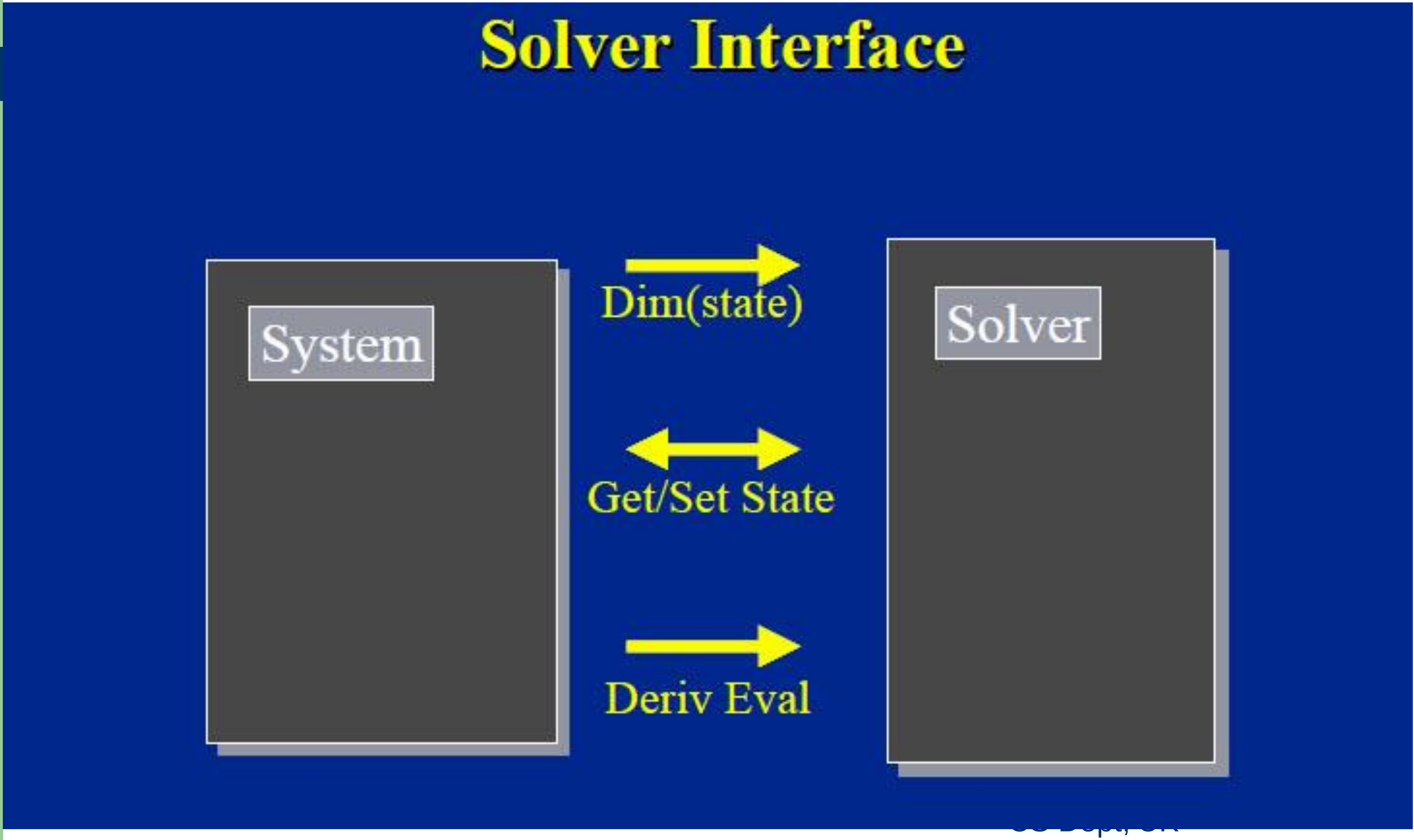
- Euler's method is *1st Order.*
- The midpoint method is *2nd Order.*
- Just the tip of the iceberg. See *Numerical Recipes* for more.
- Helpful hints:
  - *Don't* use Euler's method (you will anyway.)
  - *Do* use adaptive step size.

Differential Equation Basics

Modular Implementation

- Generic operations:
  - Get dim(x)
  - Get/set x and t
  - Deriv Eval at current (x,t)
- Write solvers in terms of these.
  - Re-usable solver code.
  - Simplifies model implementation.

Differential Equation Basics



Differential Equation Basics

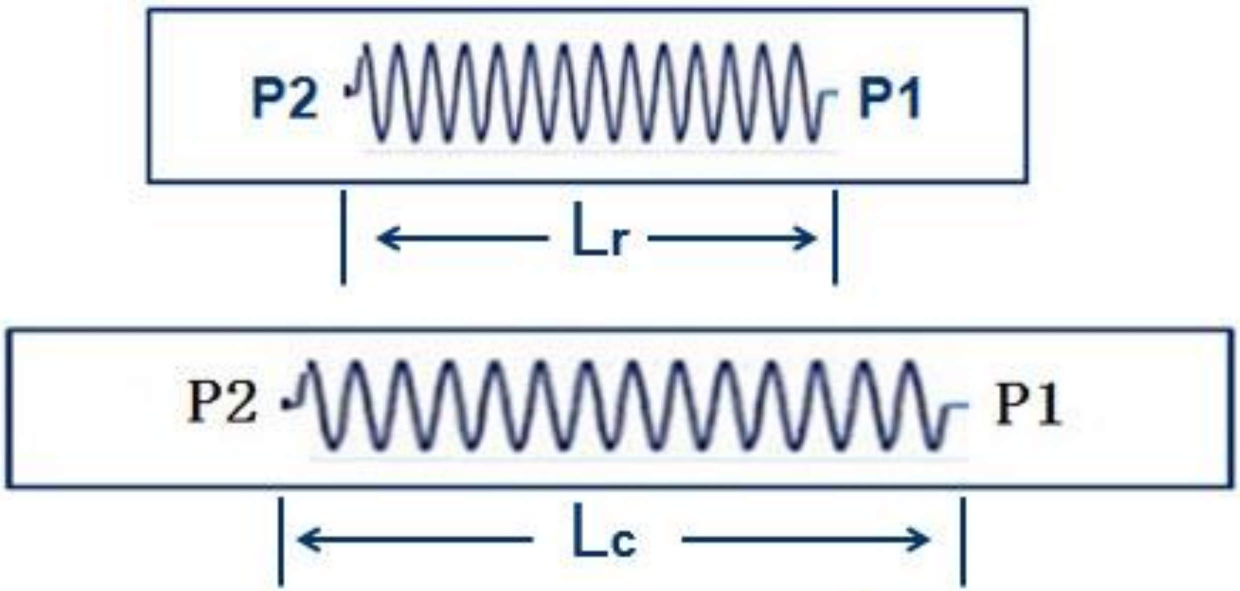
A Code Fragment

```
void eulerStep(Sys sys, float h) {
   float t = getTime(sys);
   vector<float> x0, deltaX;

   t = getTime(sys);
   x0 = getState(sys);
   deltaX = derivEval(sys,x0, t);
   setState(sys, x0 + h*deltaX, t+h);
}
```

Spring Model:

- common tool for modeling flexible objects
- to keep 2 objects at a prescribed distance
- to insert temporary control forces into an environment

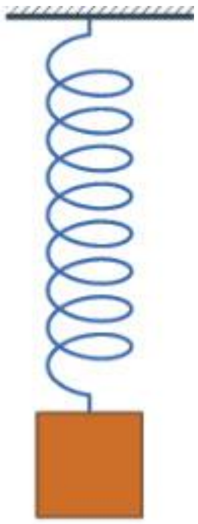


$$f_s = -k_s(L_c - L_r)\left(\frac{p_2 - p_1}{\|p_2 - p_1\|}\right)$$

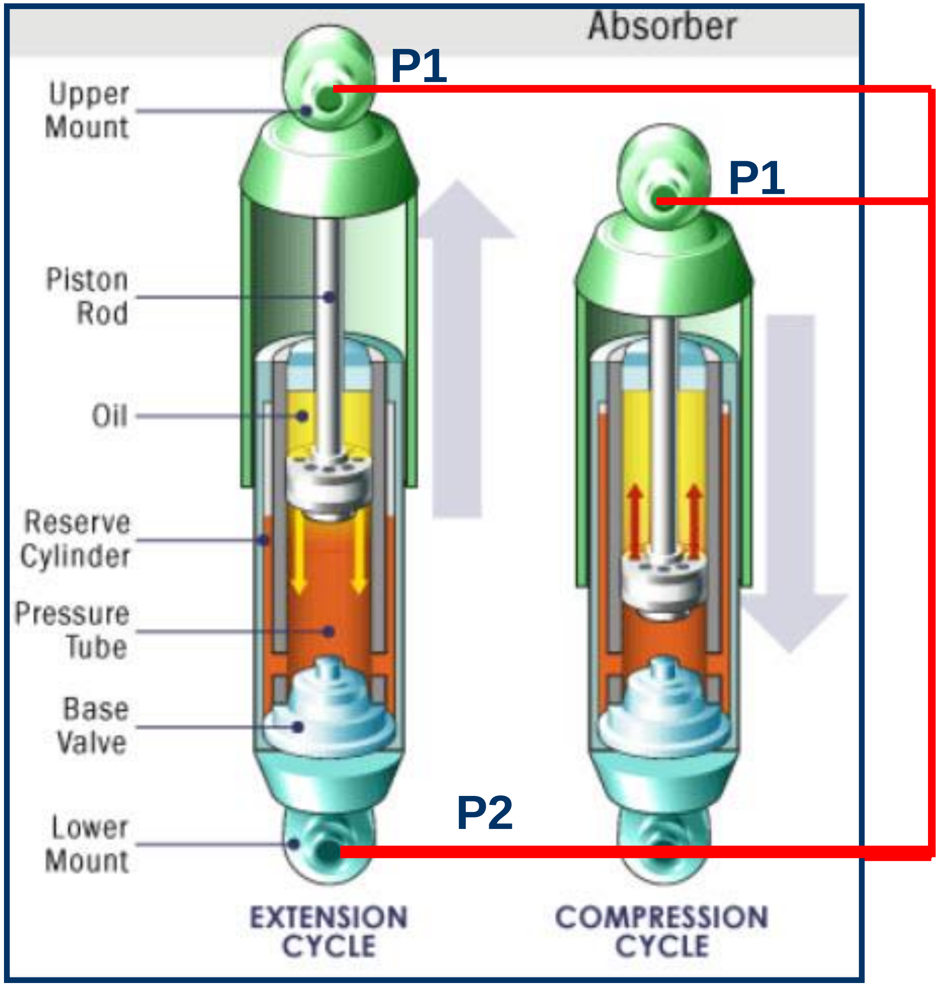
k_s : spring constant (stiffness)

Damper:

Damping is an influence within or upon an oscillatory system that has the effect of reducing, restricting or preventing its oscillations

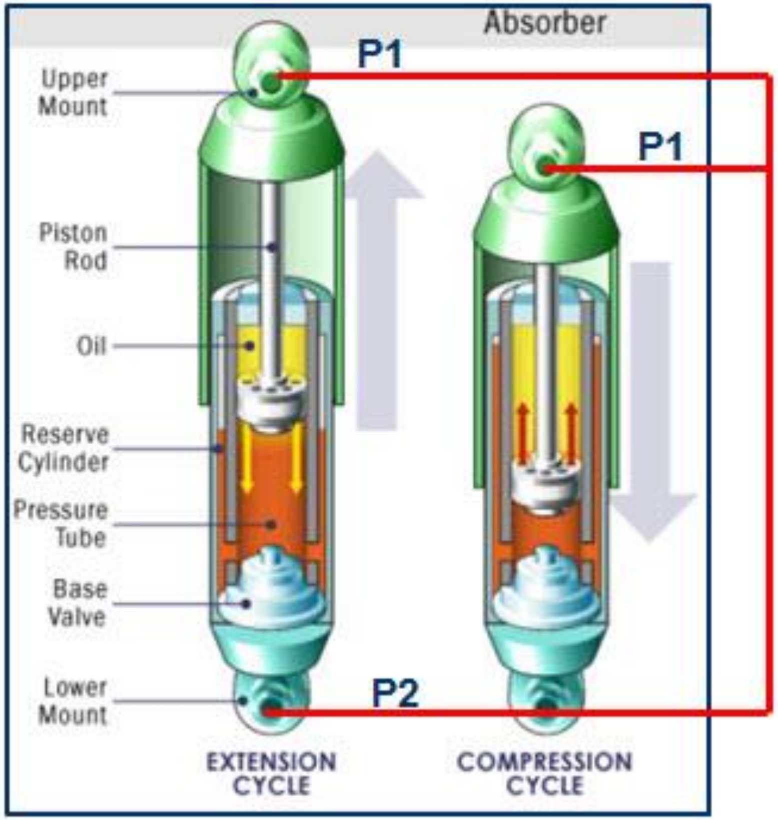


Un-damped spring



Damper:

- **Damper** works against its relative velocity
- the force of a damper is negatively proportional to the velocity of spring length (v_s)



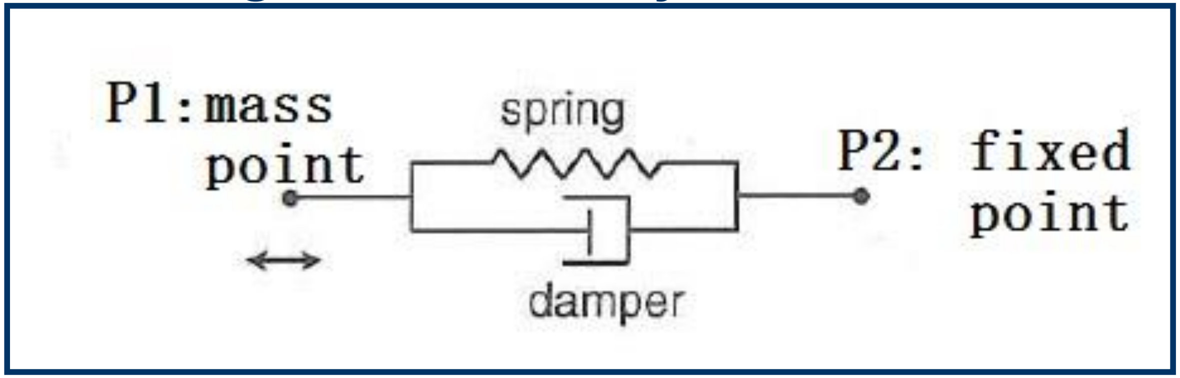
$$f_d = -k_d(\dot{p}_2 - \dot{p}_1) \cdot \left(\frac{p_2 - p_1}{\|p_2 - p_1\|}\right)\left(\frac{p_2 - p_1}{\|p_2 - p_1\|}\right)$$

k_d : damper constant (resistance to spring length change)

works in two cycles: compression cycle ; extension cycle

Spring-Damper Pair:

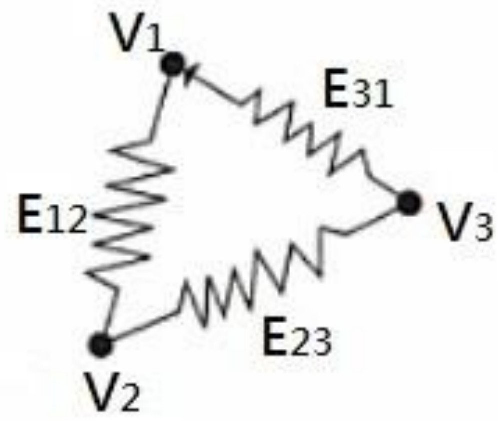
- one of the most useful tools to incorporate some use of forces into an animation
- the spring represents a force to maintain a relationship between two points, the damper is used to restrict the motion and keep the system from reacting too violently



$$f = \left(k_s \left(L_c - L_r \right) - k_d \left(\dot{p}_2 - \dot{p}_1 \right) \cdot \left(\frac{p_2 - p_1}{\| p_2 - p_1 \|} \right) \right) \left(\frac{p_2 - p_1}{\| p_2 - p_1 \|} \right)$$

7.2 Spring Animation Examples

Mass-spring-damper modeling of flexible objects:

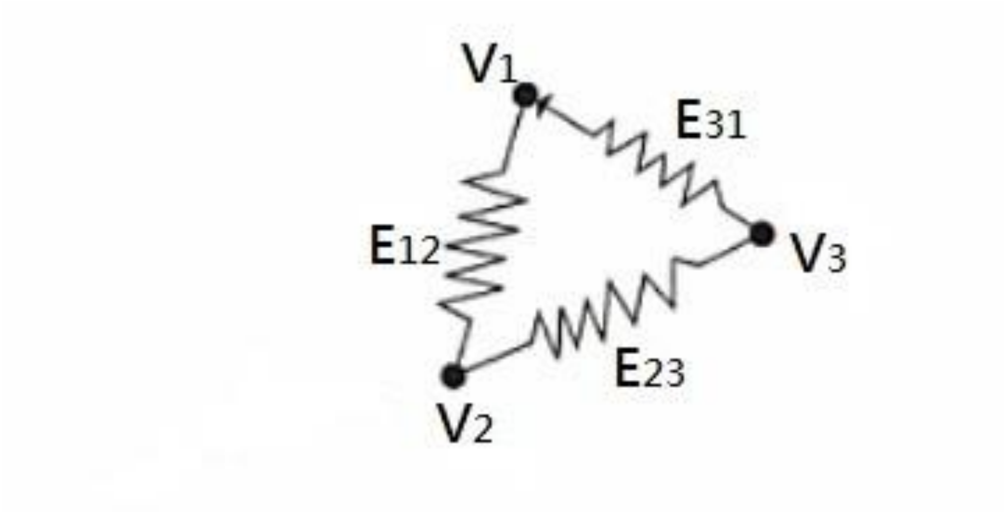


Dampers not shown

- model each vertex as a point mass
- model each edge as a spring with a paired damper (not shown)
- each spring's rest length is set equal to the original length of the edge
- a mass is assigned to the object by the animator and the mass is evenly distributed among the object's vertices
- spring constants are assigned uniformly throughout object

7.2 Spring Animation Examples

Mass-spring-damper modeling of flexible objects:

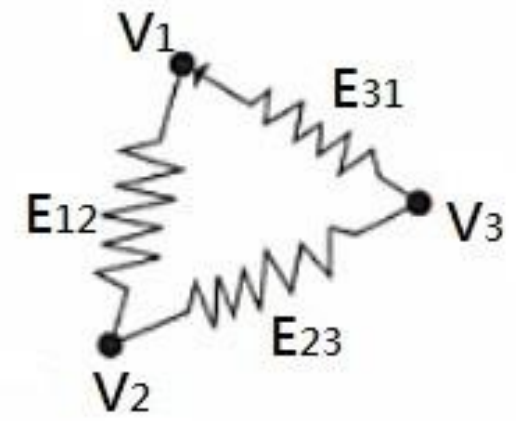


Dampers not shown

- as external forces are applied to specific vertices, vertices will be displaced relative to other vertices of the object
- this displacement will induce spring forces, which will impart forces to the adjacent vertices as well as reactive forces back to the initial vertex
- these forces will result in further displacements, which will induce more spring forces throughout the object, result in more displacements, and so on.

7.2 Spring Animation Examples

Mass-spring-damper modeling of flexible objects:



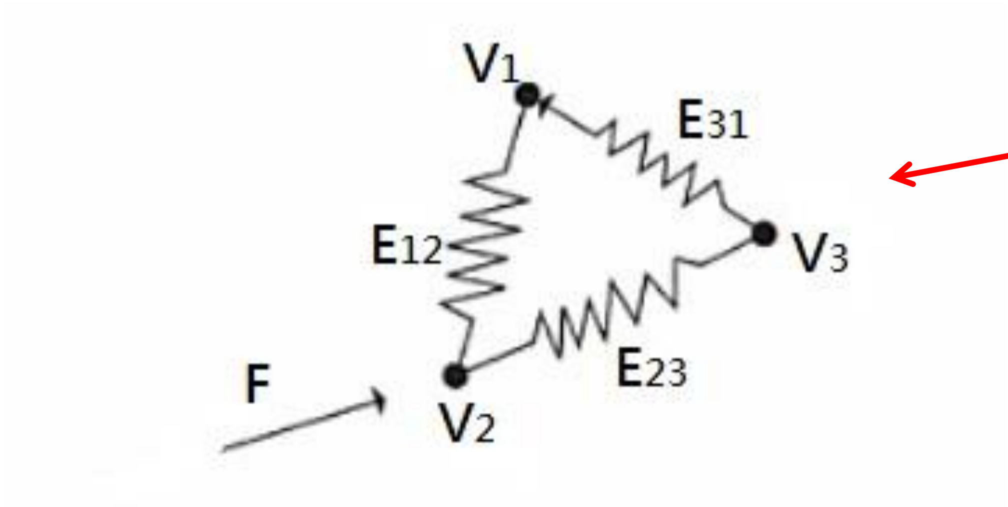
Dampers not shown

- the result will be an object that is wriggling and jiggling as a result of the forces propagating along the edge springs and producing constant relative displacements of vertices

Drawback: the effect has to propagate through the object, one time step at a time. This means that the object's reaction to forces depends on the representation of the object (not unique)

7.2 Spring Animation Examples

A simple example:



Dampers not shown

- an external force is applied to vertex V_2 of an equilateral triangle for one time step
- acceleration

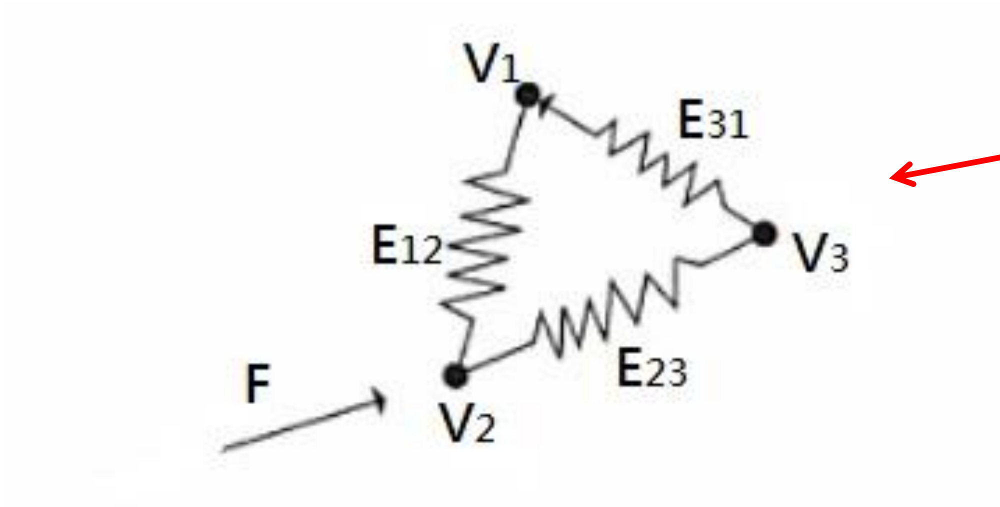
$$a_2 = F / m_2$$

- velocity

$$v_2' = v_2 + a_2 \Delta t = a_2 \Delta t$$

7.2 Spring Animation Examples

A simple example:



Dampers not shown

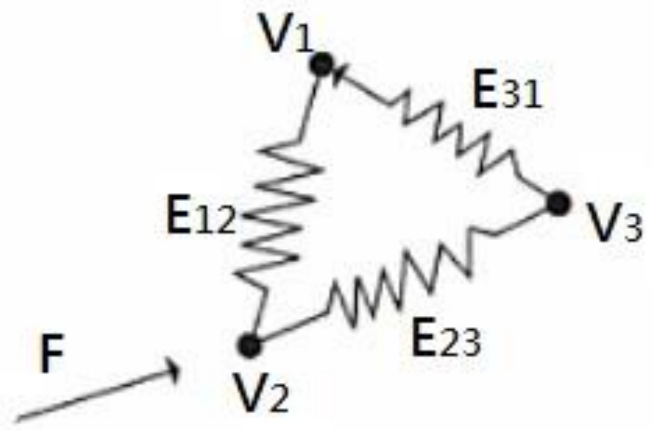
- position

$$p_2' = p_2 + \frac{1}{2}(v_2 + v_2')\Delta t = p_2 + \frac{1}{2}v_2'\Delta t = p_2 + \frac{1}{2}a_2(\Delta t)^2$$

consequently, the lengths of edges E_{12} and E_{23} are changed, and the spring force is created along the two edges

7.2 Spring Animation Examples

A simple example:

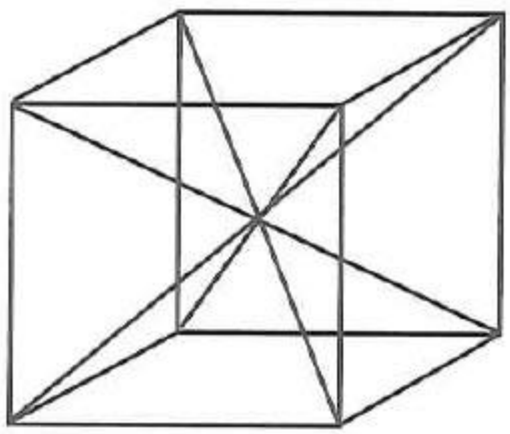


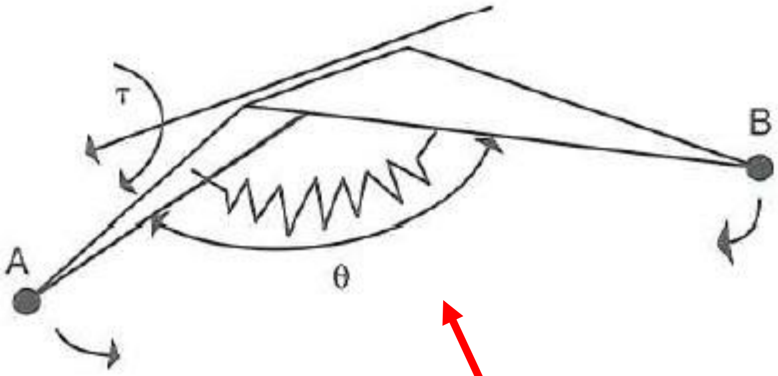
Dampers not shown

- next time step, the spring that models edge E_{12} imparts a restoring force to vertices V_1 and V_2, while the spring that models edge E_{23} imparts a restoring force to vertices V_2 and V_3

- needs to consider **stable configuration**

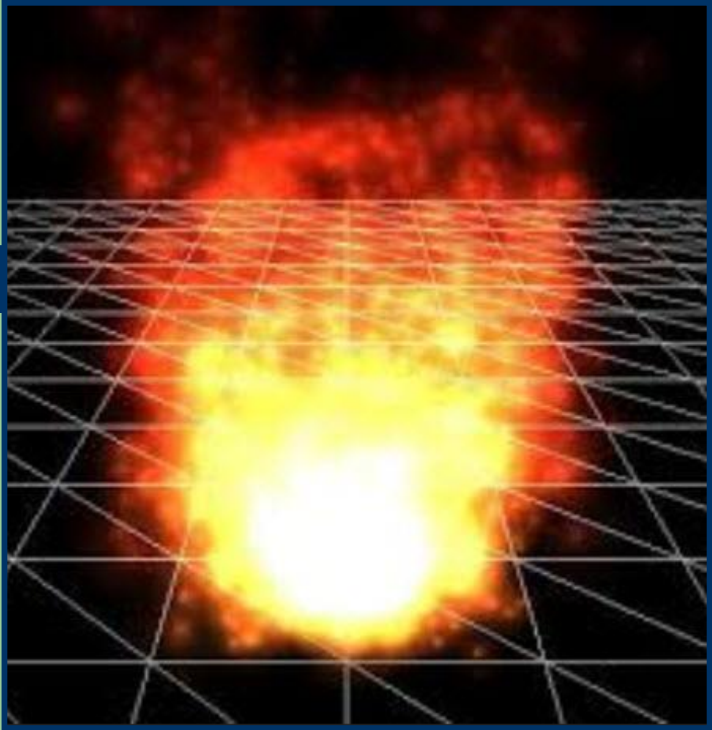
7.2 Spring Animation Examples





- if a tube’s edges are modeled with springs, during applications of external forces, the cube can turn inside out (why?)
- to stable the shape of an object, additional springs can be added across the object’s faces and its volume

7.3 Particle System Dynamics:



7.3 Particle systems:

- Collection of large number of point-like elements to simulate certain kinds of "fuzzy" phenomena
- Often animated as a simple physical simulation (show a video here)
- Assumptions used in rendering and calculation:
  - particles do not collide with other particles
  - particles do not cast shadows, except in an aggregate sense
  - particles only cast shadows on the rest of the environment
  - particles do not reflect lights – they are treated as point light sources

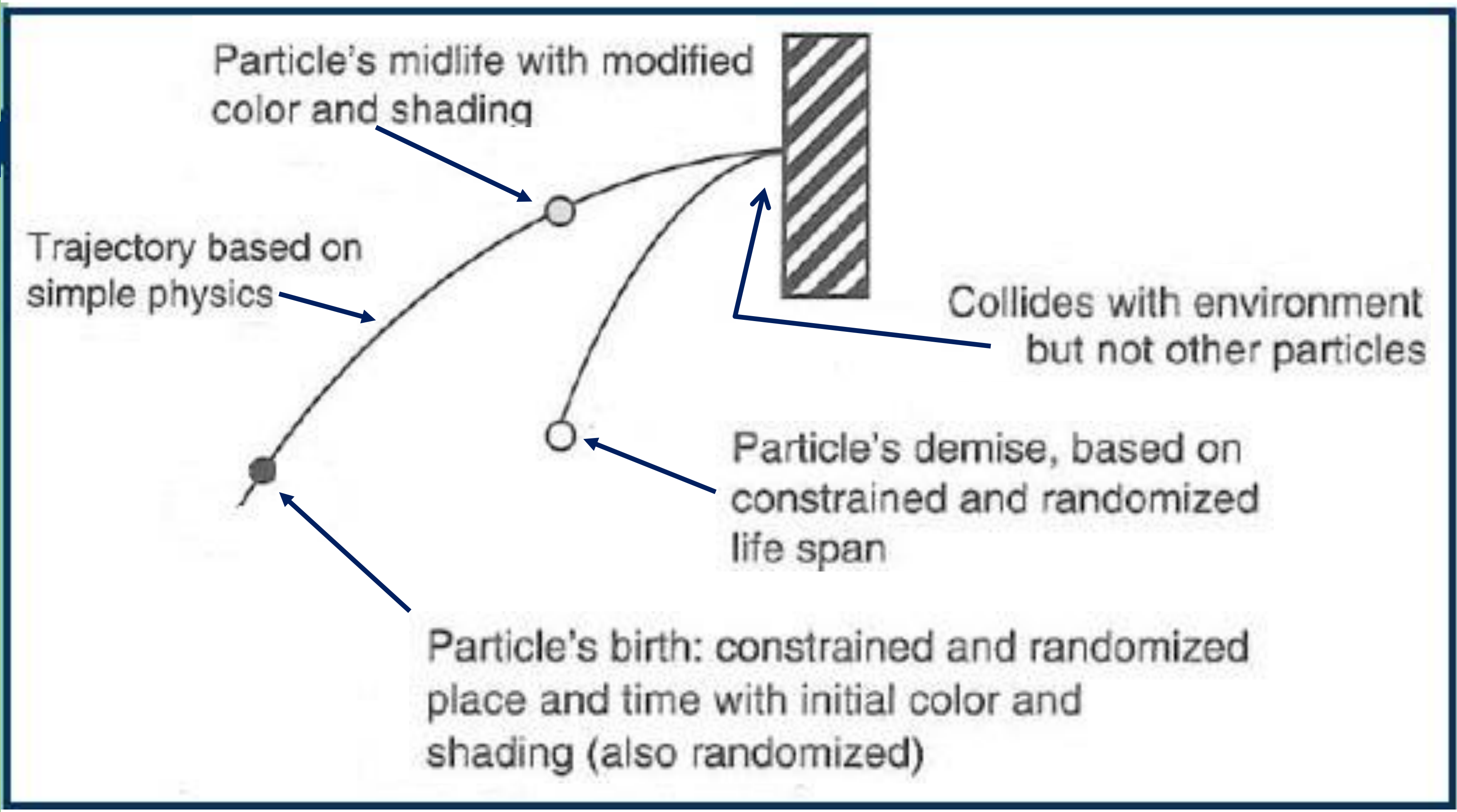
7.3 Particle systems:

- a particle system's position and motion are controlled by an **emitter** (a regular 3D mesh object, such as a cube or a plane)
- the emitter acts as the source of the particles, and its location in 3D space determines where they are generated and whence they proceed.
- a set of “fuzzy” particle behavior parameters are attached to the emitter, including: spawning rate, particles' initial velocity vector, particle lifetime, particle color, …
- the particles are usually appear to “spray” directly from faces of the emitter (the initial velocity vector is set to be normal to the individual face(s) of the object)
- a typical particle system's update loop (performed for each frame of animation) can be separated into two distinct stages, the **parameter update/simulation** stage and the **rendering** stage.

Simulation stage:

- calculate the number of new particles that must be created
- each particle is spawned in a specific position in 3D space (based on the emitter's position and the spawning area specified) with initialized parameters (velocity, color, etc.)
- at each update, all existing particles are checked to see if they have exceeded their lifetime (YES: removed from the simulation; NO: particles' position and other characteristics are advanced based on a physical simulation, which can be as simple as translating their current position, or as complicated as performing physically accurate trajectory calculations which take into account external forces (gravity, friction, wind, etc.))
- perform collision detection between particles and specified 3D objects in the scene to make the particles bounce off of or otherwise interact with obstacles in the environment
- collisions between particles are rarely used, as they are computationally expensive and not visually relevant

Life of a particle:

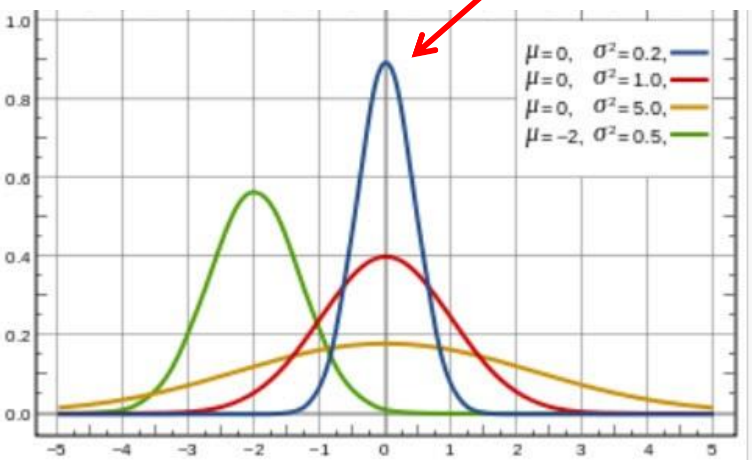


Particle generation:

-Generated according to a controlled stochastic process

$$\#\ \text{of particles} = n + Rand(\,) * r$$

$$\#\ \text{of particles} = n(A) + Rand(\,) * r$$



Particle attributes:

-Position
-Velocity
-Shape parameters
-Color
- Transparency
- Lifetime

Randomized in some controlled way

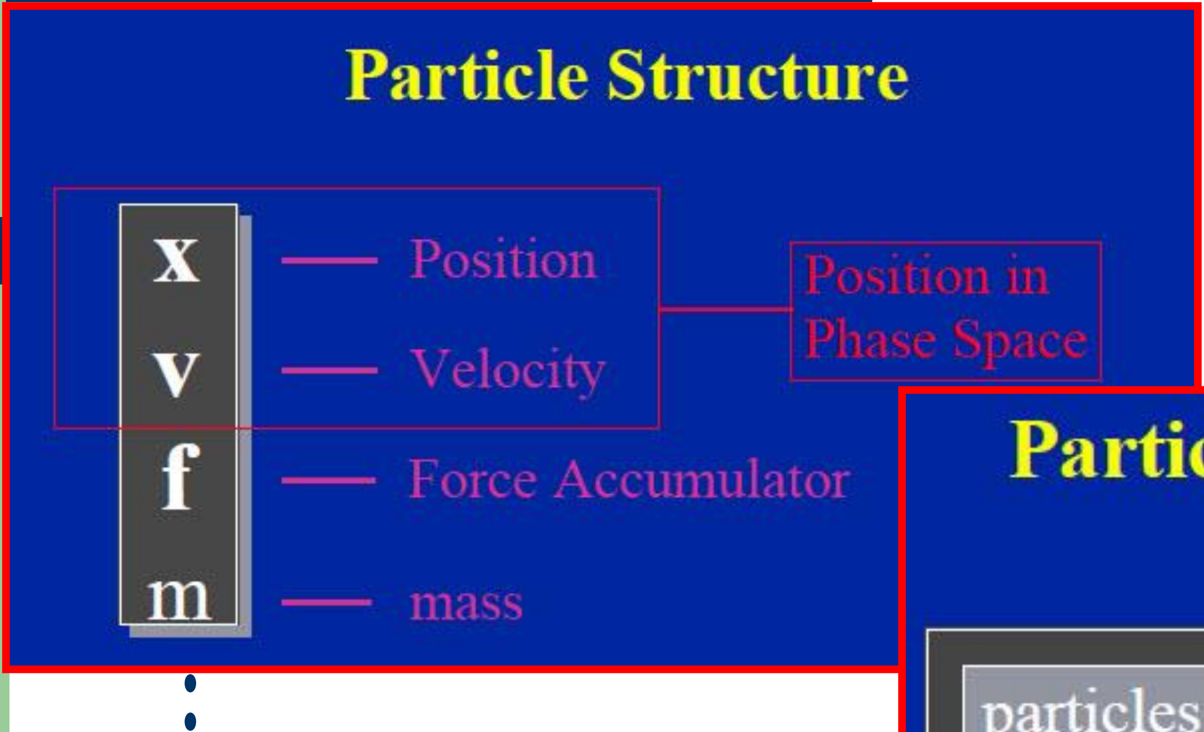
Particle termination:

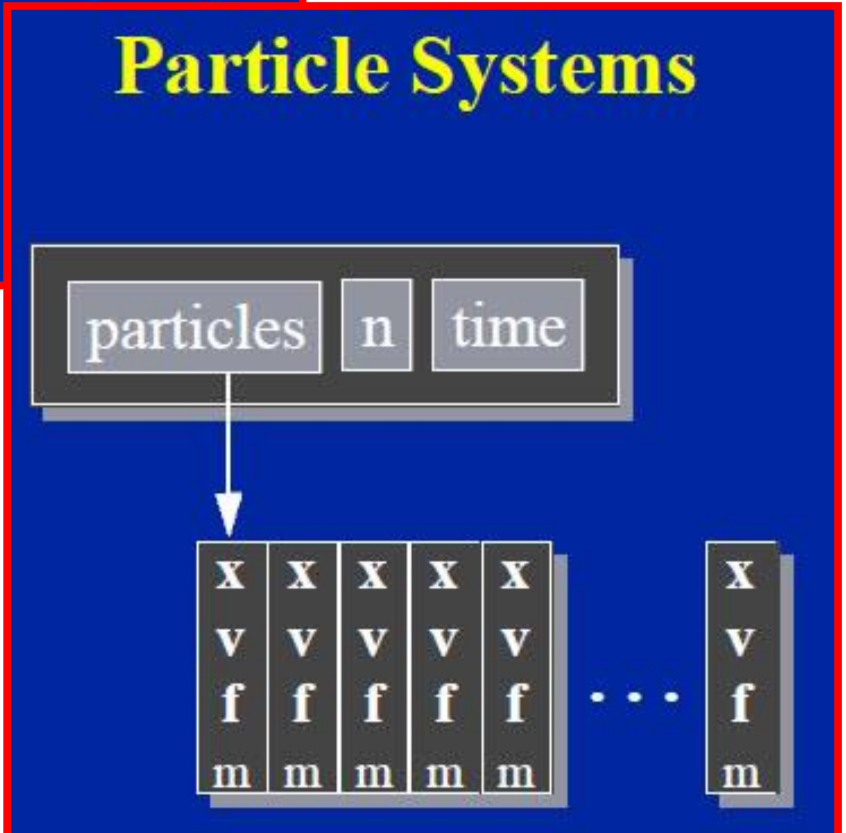
- Lifetime attribute is decremented by 1 at each new frame
- Particle is removed when attribute reaches 0

Particle animation:

- each active particle is animated throughout its life
- the activation includes:
  - position: considers forces and computes resultant particle acceleration
  - velocity: updated from its acceleration, then average of its old velocity and newly updated velocity
  - forces modeled in the environment: global force fields (gravity, wind), local force fields (vertices), collisions with objects in the environment
  - mass: mass of the particle
  - color and transparency: function of global time, its own life span remaining, its height
  - shape: function of velocity (use ellipsoid to represent a particle)

Representation:





Forces

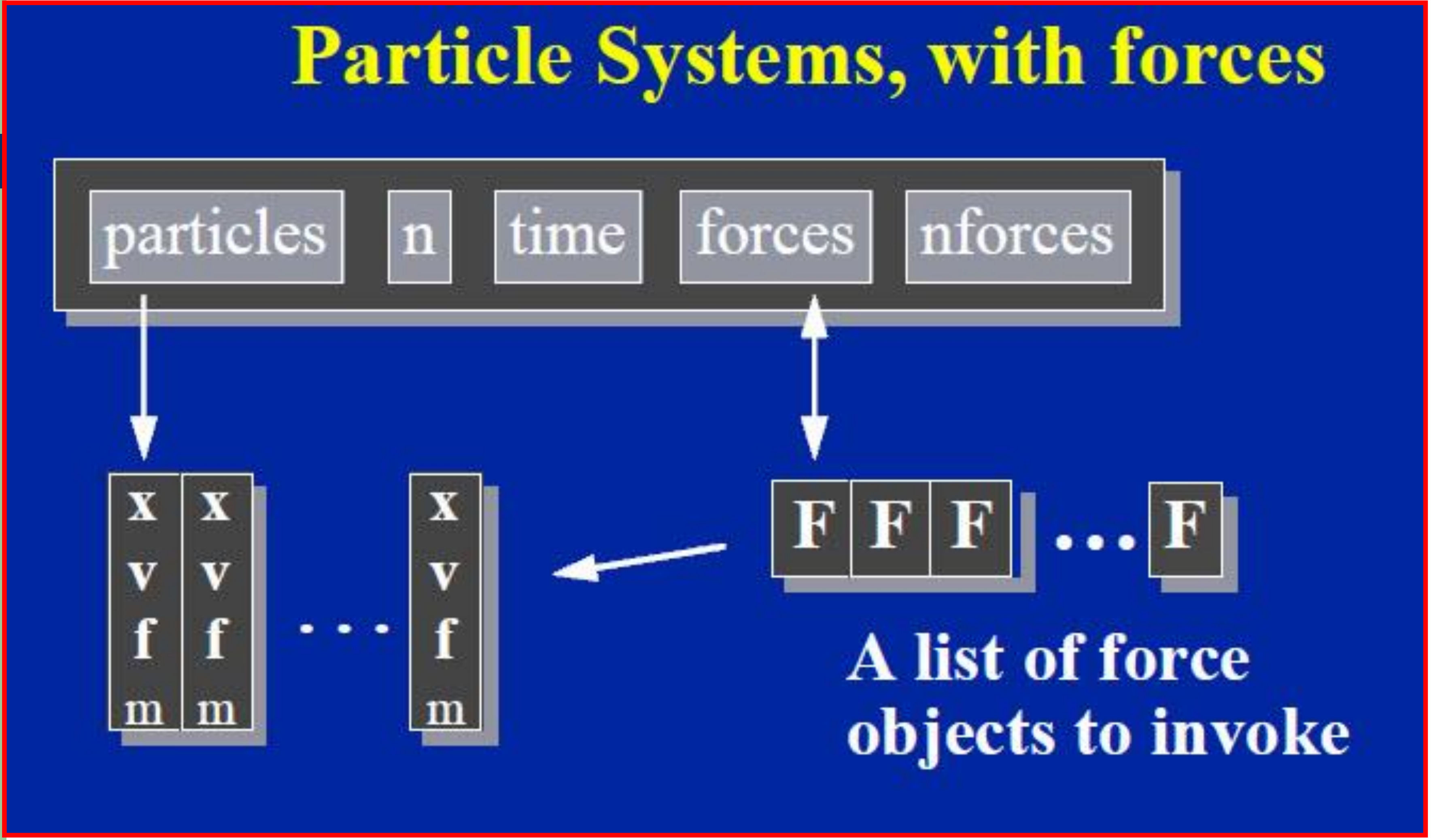
- **Constant** — gravity
- **Position/time dependent** — force fields
- **Velocity-Dependent** — drag
- **n-ary** — springs

unary

Force Structures

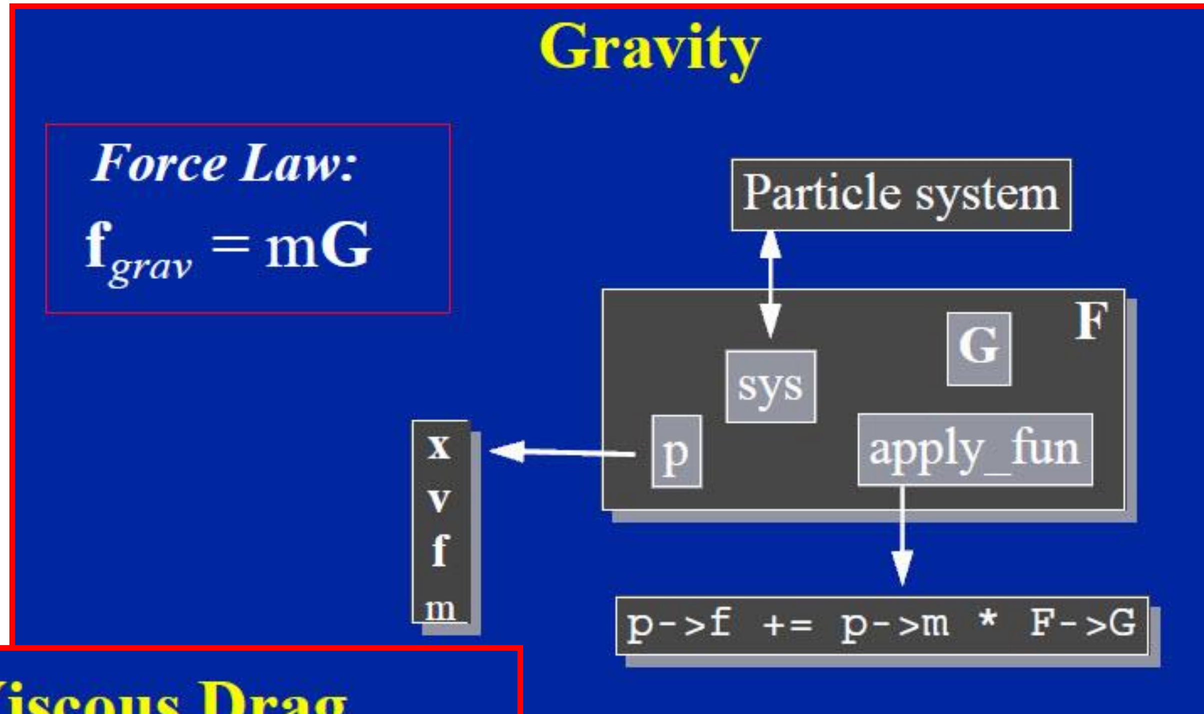
- **Unlike particles, forces are heterogeneous.**
- **Force Objects:**
  - black boxes
  - point to the particles they influence
  - add in their own forces (type dependent)
- **Global force calculation:**
  - loop, invoking force objects

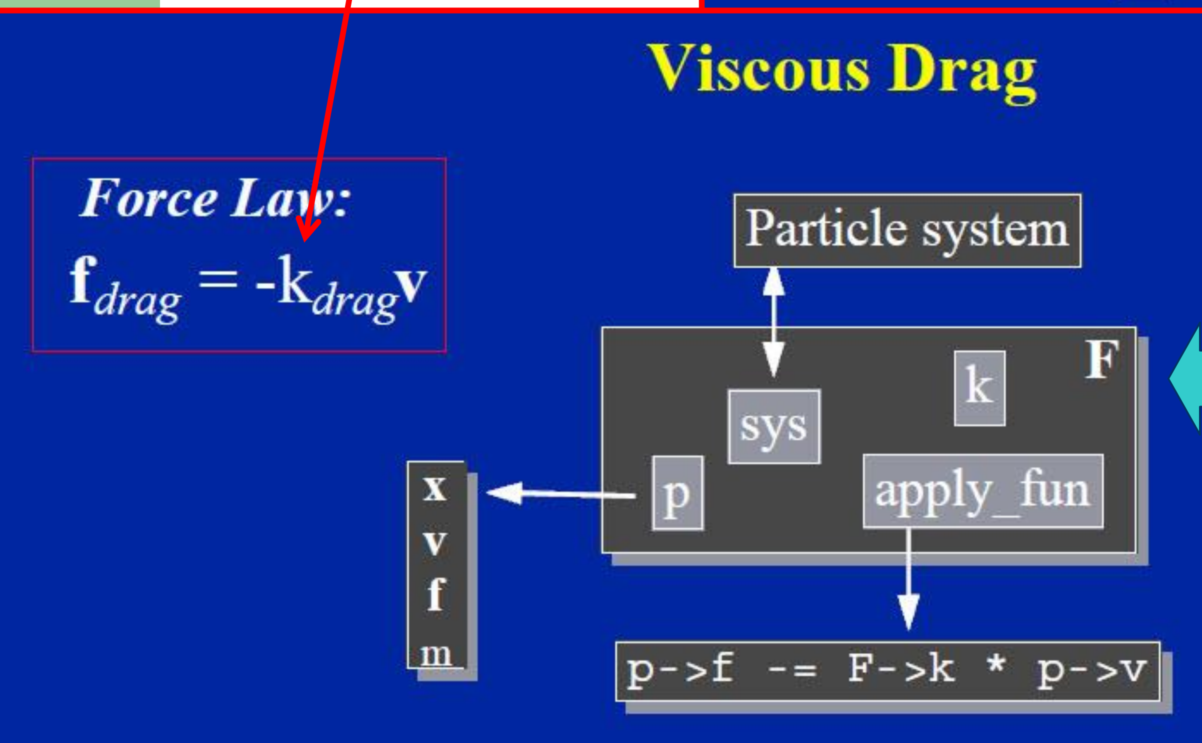
Representation:



Represen-tation:

Coefficient of drag





The effect of this is to resist motion, making a particle gradually come to rest in the absence of other influences.

Highly Recommended!

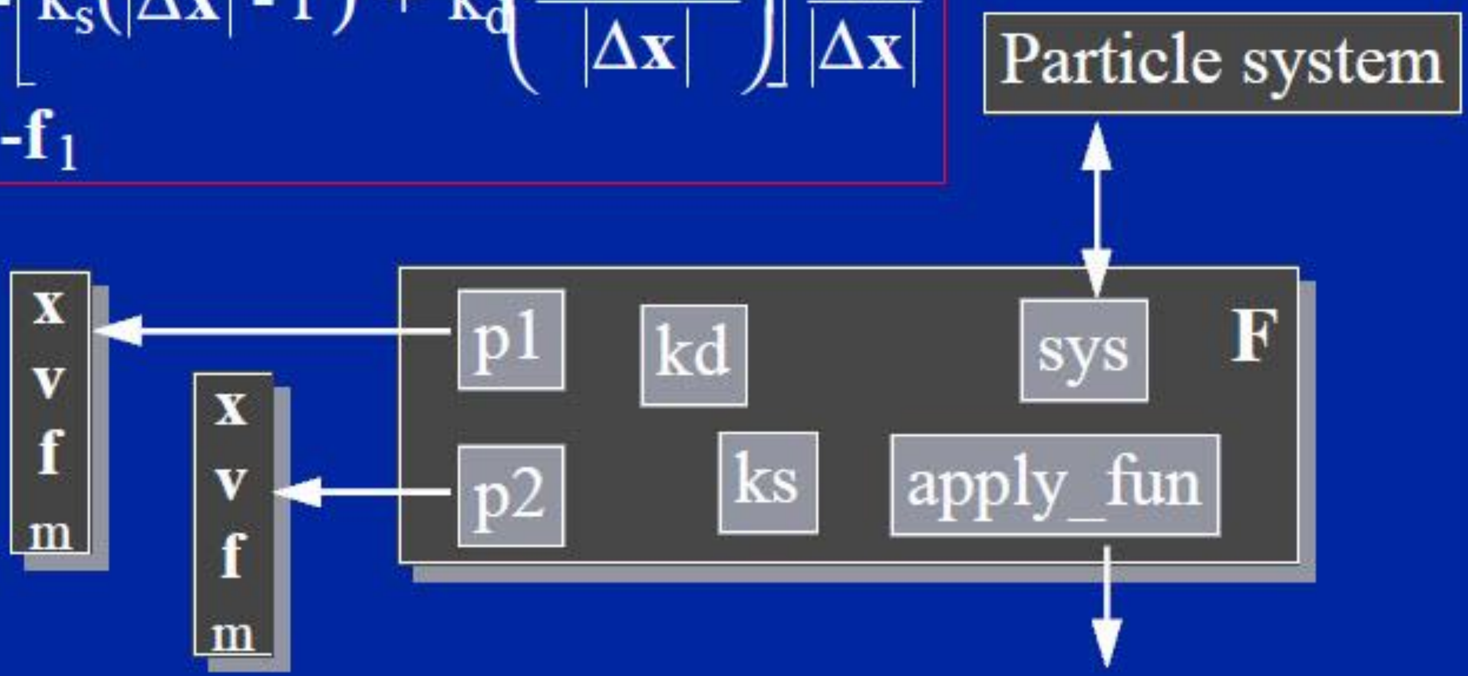
Representation:

Damped Spring

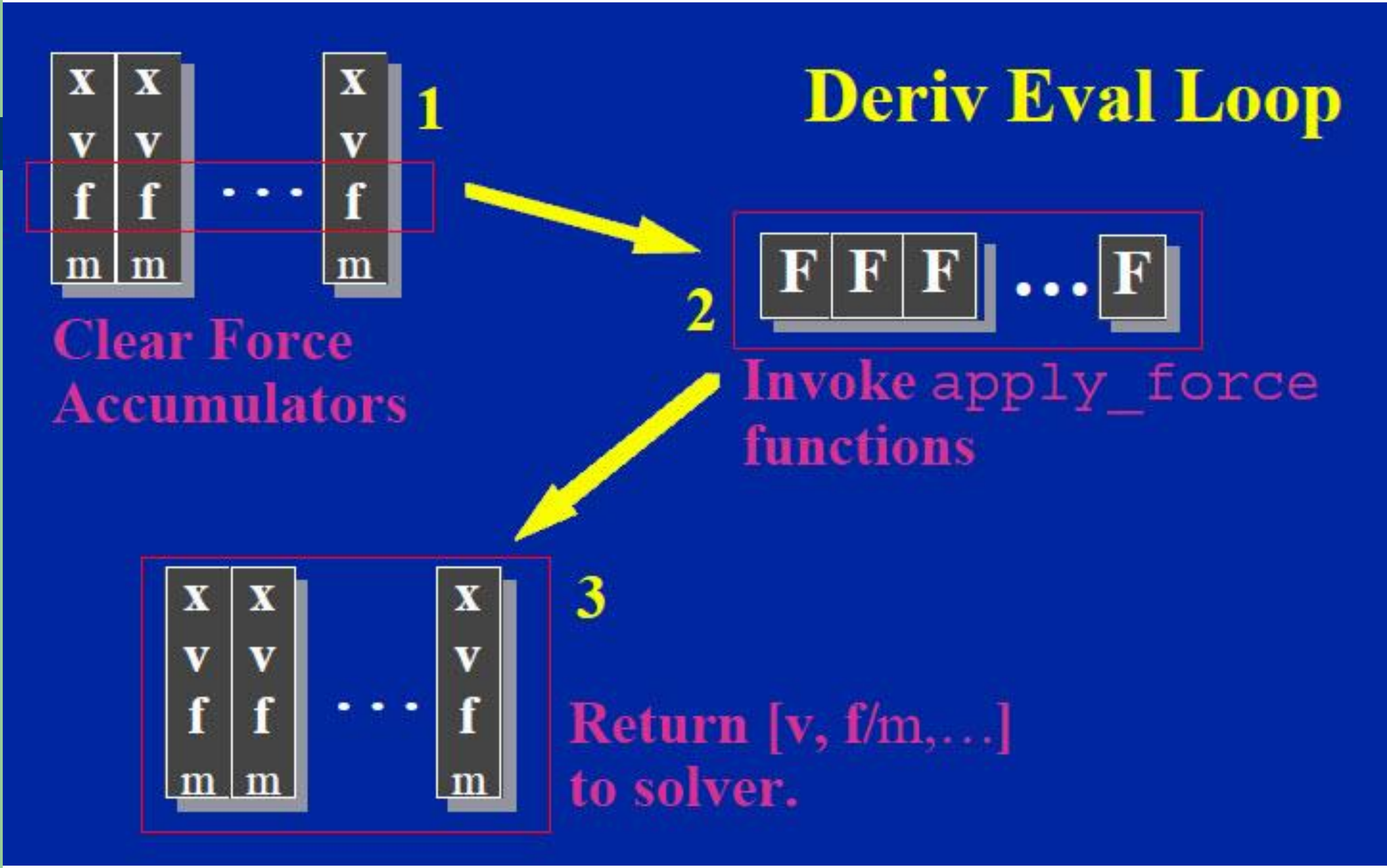
Force Law:

$$\mathbf{f}_1 = -\left[k_s(|\Delta\mathbf{x}| - r) + k_d\left(\frac{\Delta\mathbf{v}\cdot\Delta\mathbf{x}}{|\Delta\mathbf{x}|}\right)\right]\frac{\Delta\mathbf{x}}{|\Delta\mathbf{x}|}$$

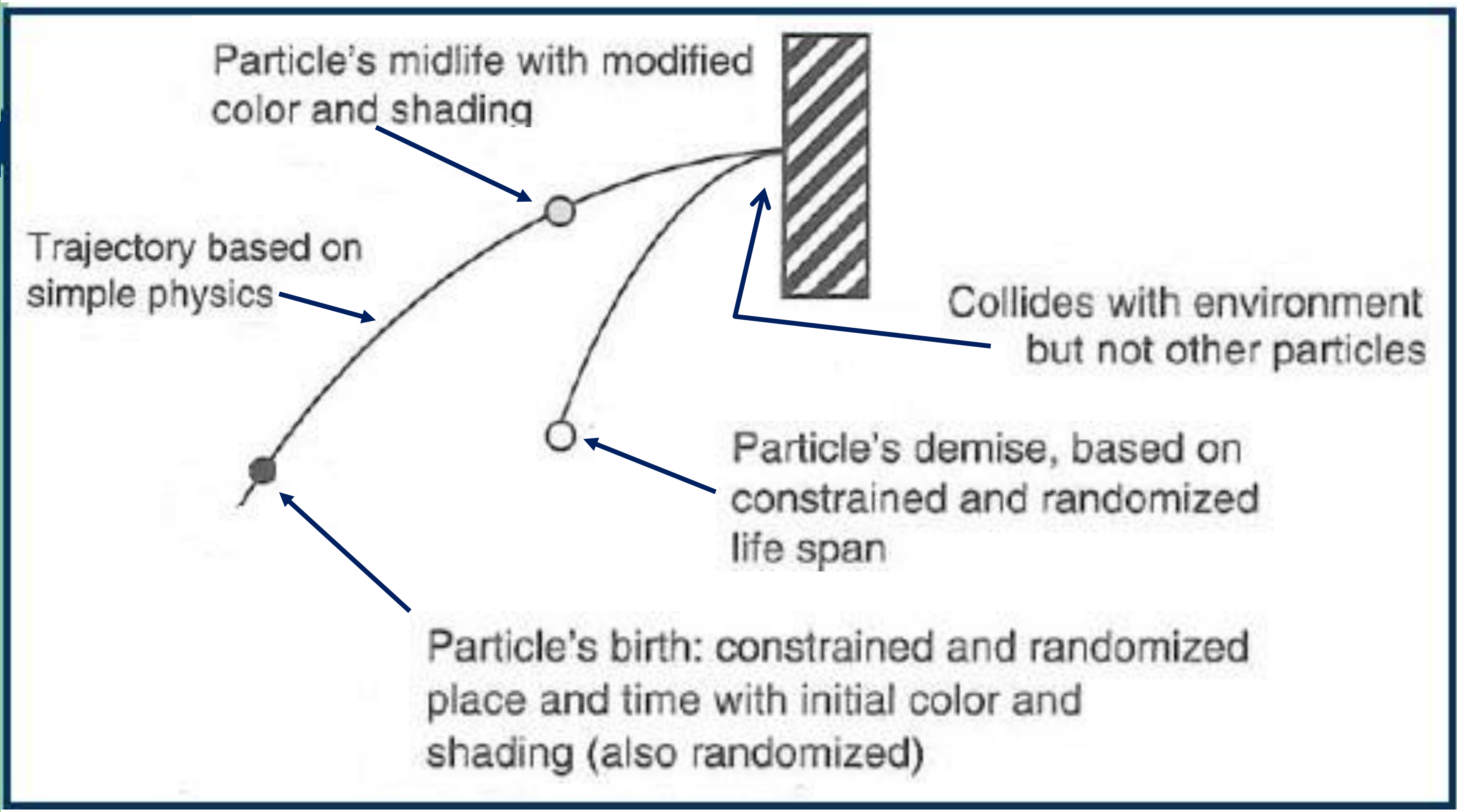
$$\mathbf{f}_2 = -\mathbf{f}_1$$



Representation:

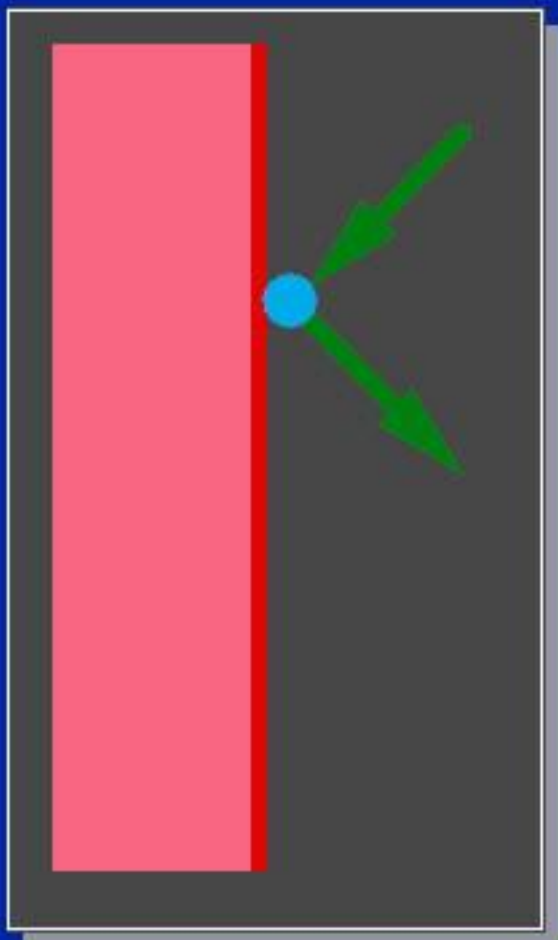


Life of a particle:



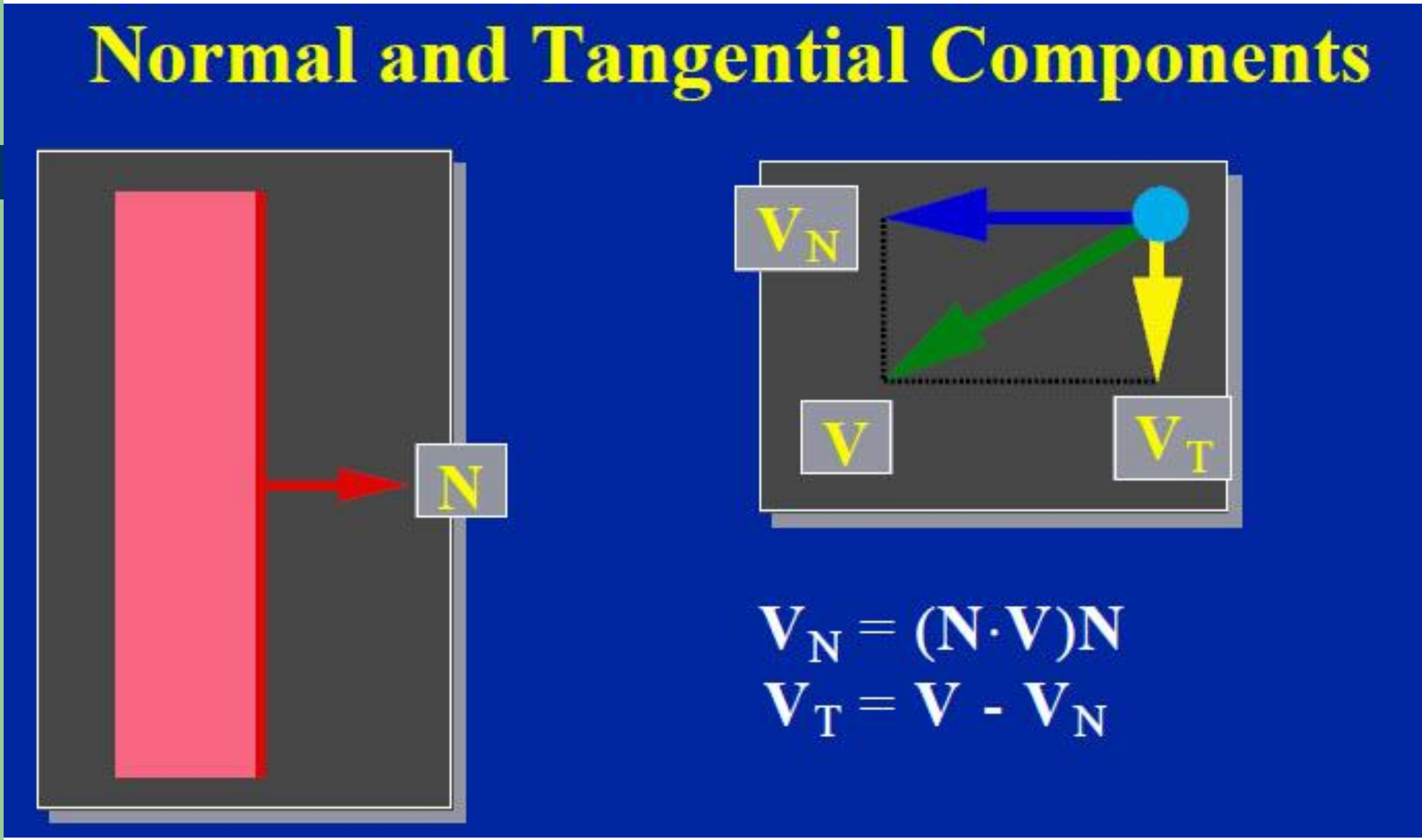
Simulation:

Bouncing off the Walls



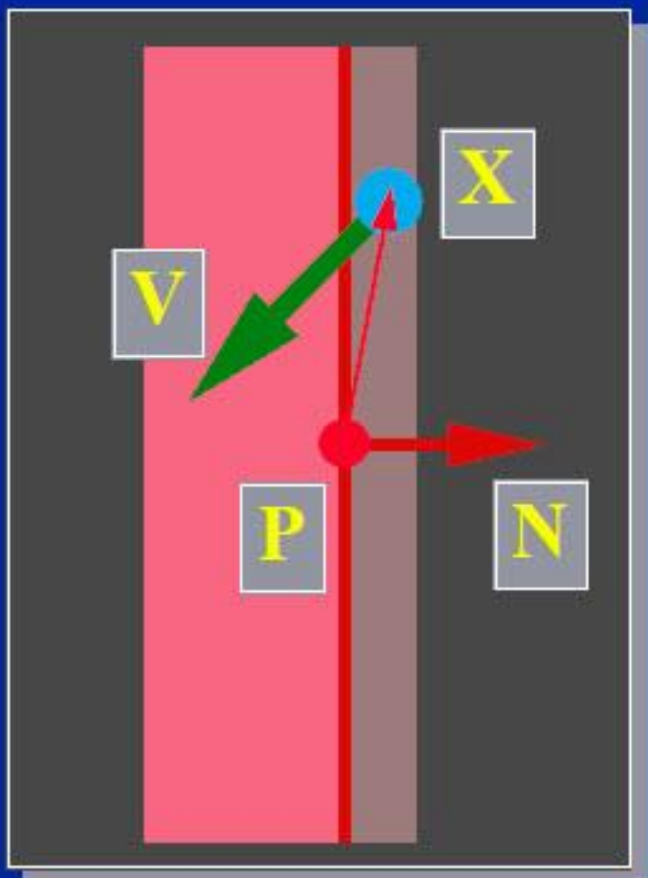
- Later: rigid body collision and contact.
- For now, just simple point-plane collisions.
- Add-ons for a particle simulator.

Simulation:



Simulation:

Collision Detection



$$(X - P)\cdot N < \varepsilon$$

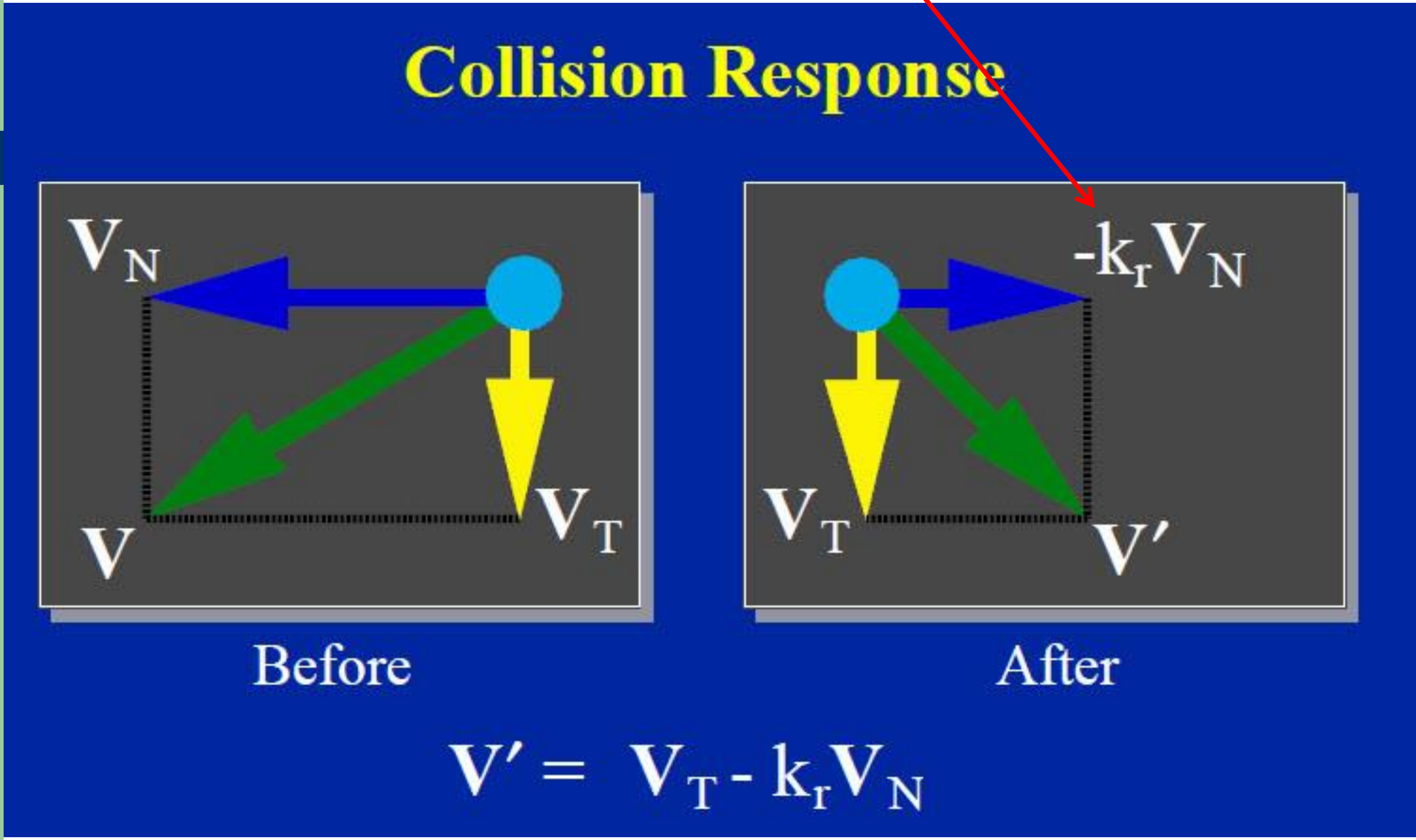
$$N\cdot V < 0$$

Theoretically, zero

- Within ε of the wall.
- Heading in.

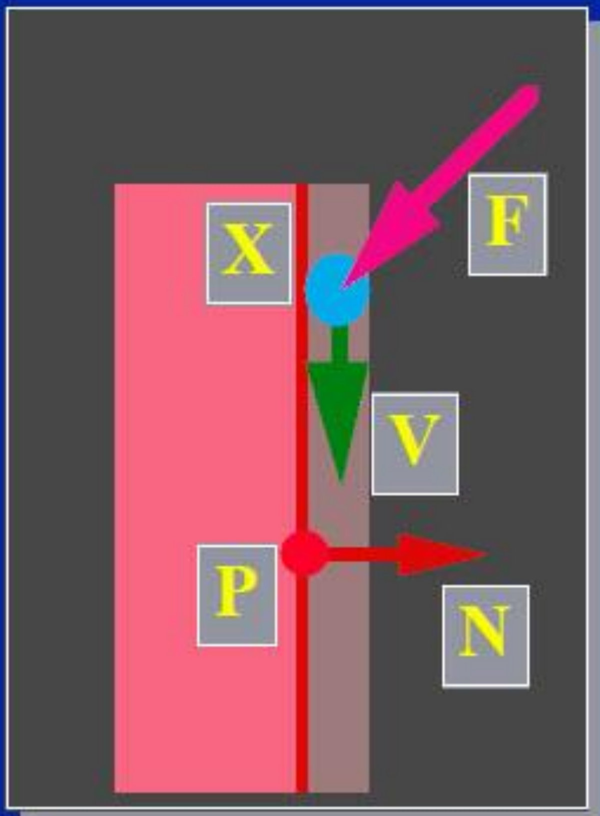
Simulation:

coefficient of restitution



Simulation:

Conditions for Contact

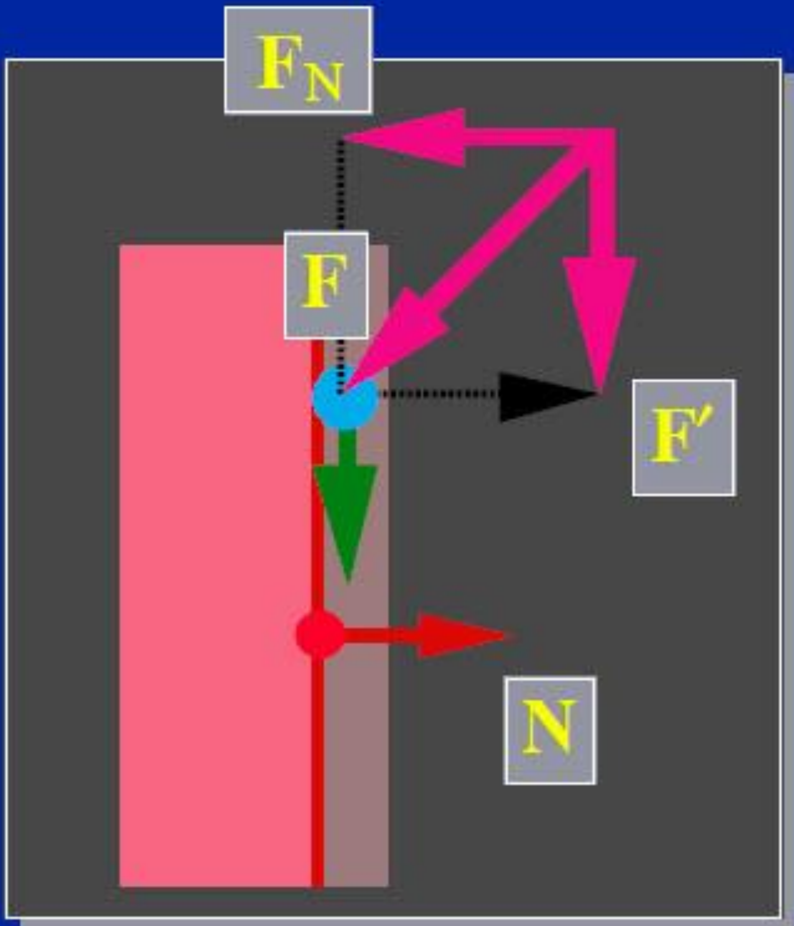


$$|(X - P)\cdot N| < \varepsilon$$

$$|N\cdot V| < \varepsilon$$

- On the wall
- Moving along the wall
- Pushing against the wall

Simulation:



Contact Force

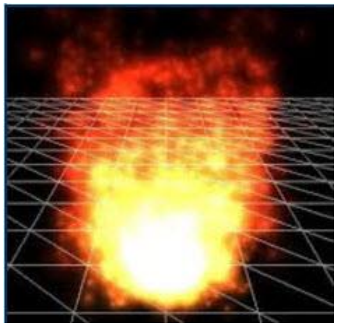
$$F' = F_T$$

The wall pushes back, cancelling the normal component of F.

(An example of a *constraint force.*)

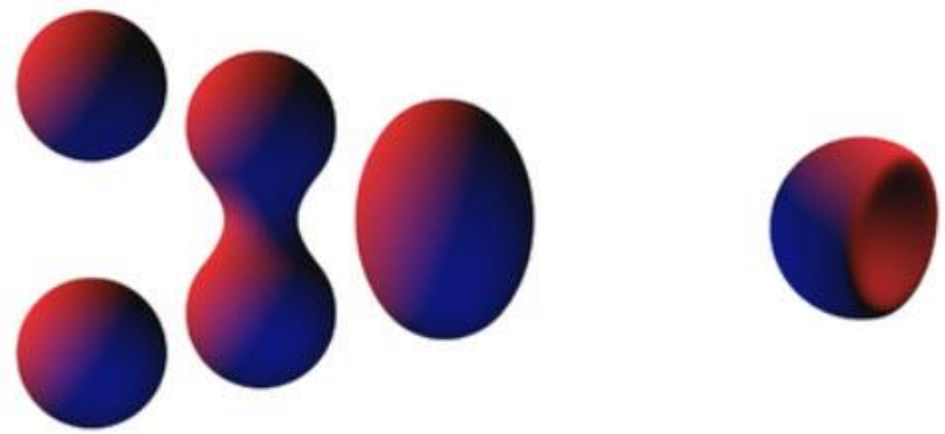
Rendering stage:

- after the update is complete, each particle is rendered, usually in the form of a textured billboarded quad (i.e. a quadrilateral that is always facing the viewer).
- However, this is not necessary; a particle may be rendered as a point light source
- it adds color to the pixel(s) it covers, but is not involved in the display pipeline (except to be hidden) or shadowing
- density of particles between a position in space and a light source can be used to estimate the amount of shadowing

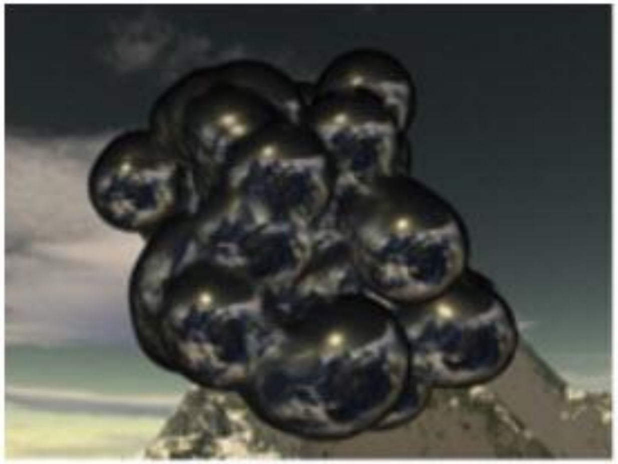


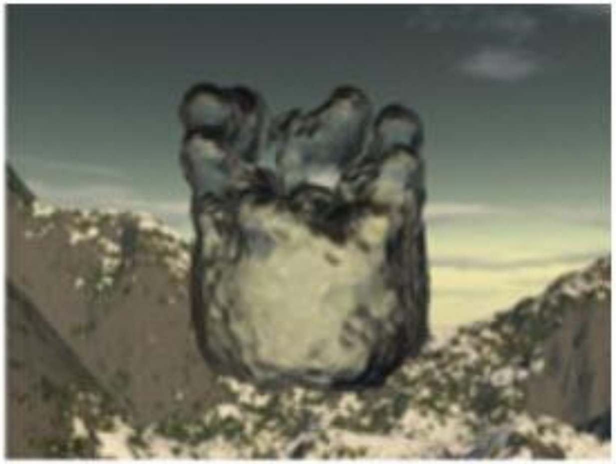
Rendering stage:

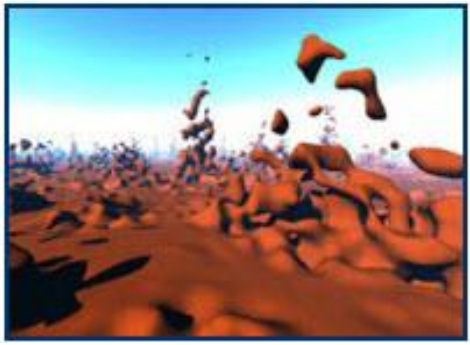
- particles can be rendered as metaballs in off-line rendering



- isosurfaces computed from particle-metaballs make quite convincing liquids.





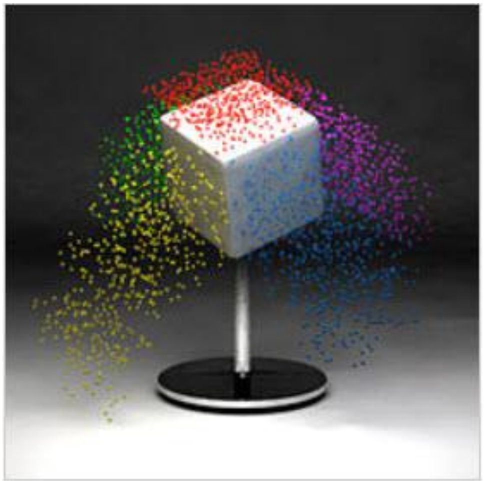


Rendering stage:

- 3D mesh objects can "stand in" for the particles — a snowstorm might consist of a single 3D snowflake mesh being duplicated and rotated to match the positions of thousands or millions of particles.

Snowflakes vs Hair

- particle systems can be either **animated** or **static**; that is, the lifetime of each particle can either be distributed over time or rendered all at once
- consequence of this distinction is similar to the difference between snowflakes and hair - animated particles are akin to snowflakes, which move around as distinct points in space, and static particles are akin to hair, which consists of a distinct number of curves



A cube emitting 5000 animated particles, obeying a "gravitational" force in -Y direction

The same cube emitter rendered using static particles, or strands.

Snowflakes vs Hair

- the term "particle system" itself often brings to mind only the **animated** aspect, which is commonly used to create moving particle simulations — sparks, rain, fire, etc.
- In these implementations, each frame of the animation contains each particle at a specific position in its life cycle, and each particle occupies a single point position in space.
- For effects such as fire or smoke that dissipate, each particle is given a fade out time or fixed lifetime; effects such as snowstorms or rain instead usually terminate the lifetime of the particle once it passes out of a particular field of view

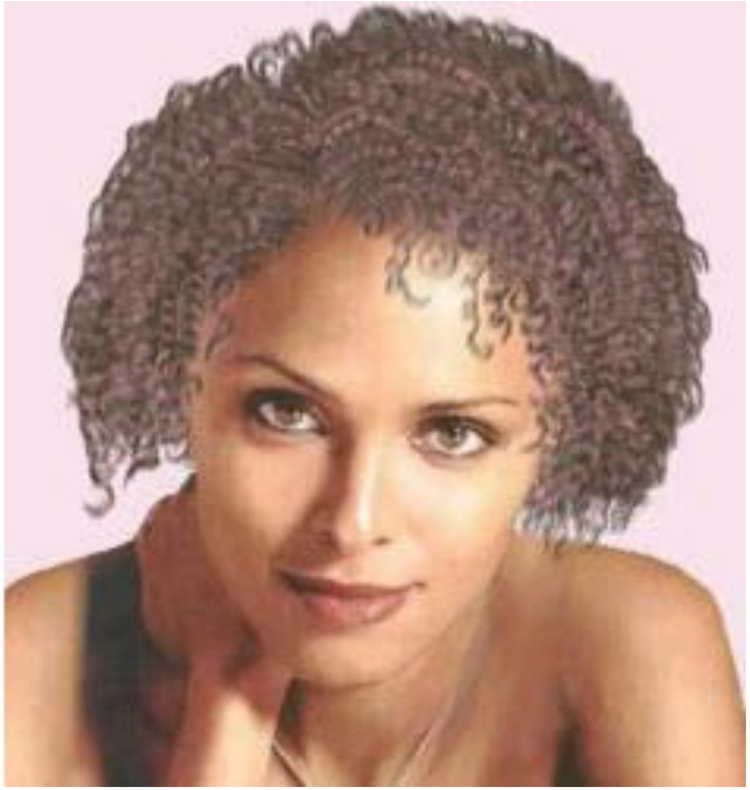
Snowflakes vs Hair

- However, if the entire life cycle of each particle is rendered simultaneously, the result is **static** particles — strands of material that show the particles' overall trajectory, rather than point particles. These strands can be used to simulate hair, fur, grass, and similar materials.

Alice Lin and Fuhua (Frank) Cheng

Snowflakes vs Hair

- However, if the entire life cycle of each particle is rendered simultaneously, the result is **static** particles — strands of material that show the particles' overall trajectory, rather than point particles. These strands can be used to simulate hair, fur, grass, and similar materials.



Alice Lin and Fuhua (Frank) Cheng

Snowflakes vs Hair

- However, if the entire life cycle of each particle is rendered simultaneously, the result is **static** particles — strands of material that show the particles' overall trajectory, rather than point particles. These strands can be used to simulate hair, fur, grass, and similar materials.

Alice Lin and Fuhua (Frank) Cheng

Snowflakes vs Hair

- The strands can be controlled with the same velocity vectors, force fields, spawning rates, and deflection parameters that animated particles obey.
- In addition, the rendered thickness of the strands can be controlled and in some implementations may be varied along the length of the strand.
- Different combinations of parameters can impart stiffness, limpness, heaviness, bristliness, or any number of other properties. The strands may also use texture mapping to vary the strands' color, length, or other properties across the emitter surface.

References:

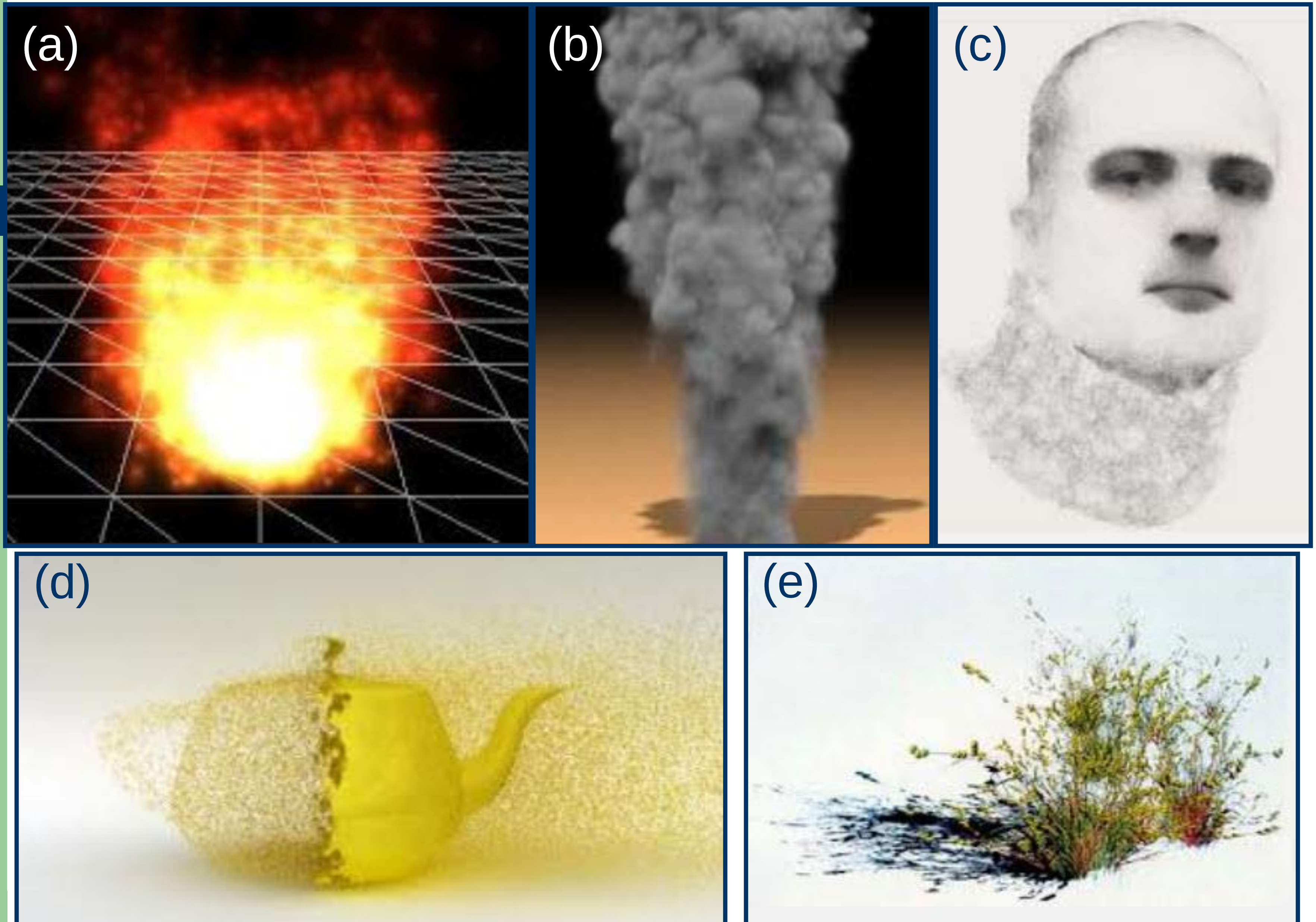
- A. Witkin, Physically Based Modeling: Principles and Practice, http://www.cs.cmu.edu/~baraff/ sigcourse/
- A. Witkin, D. Baraff, M. Kass, An Introduction to Physically Based Modeling, http://www.cs.cmu.edu/~baraff/pbm/pbm.html
- D. Baraff, Physically based modeling, http://www.pixar.com/*companyinfo/research/pbm2001/*
- *D. James,* Physically Based Animation for Computer Graphics, http://www.cs.cornell.edu/***courses****/cs5643/2010sp/*

Acknowledgement:

Some materials are taken from the above references.

End of Physically Based Animation I

7.3 Particle System Dynamics:



Life of a particle: