Particle Systems

Reading

- Required:
  - Witkin, *Particle System Dynamics*, SIGGRAPH ’97 course notes on Physically Based Modeling.
  - Witkin and Baraff, *Differential Equation Basics*, SIGGRAPH ’01 course notes on Physically Based Modeling.

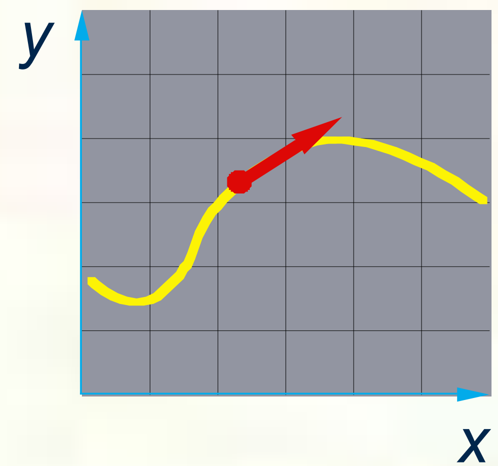
- Optional
  - Hocknew and Eastwood. *Computer simulation using particles*. Adam Hilger, New York, 1988.
  - Gavin Miller. “The motion dynamics of snakes and worms.” *Computer Graphics* 22:169-178, 1988.

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:

Particle in a flow field

- We begin with a single particle with:
  - Position, $\vec{\mathbf{x}} = \begin{bmatrix} x \\ y \end{bmatrix}$
  - Velocity, $\vec{\mathbf{v}} = \dot{\mathbf{x}} = \frac{d\vec{\mathbf{x}}}{dt} = \begin{bmatrix} dx/dt \\ dy/dt \end{bmatrix}$

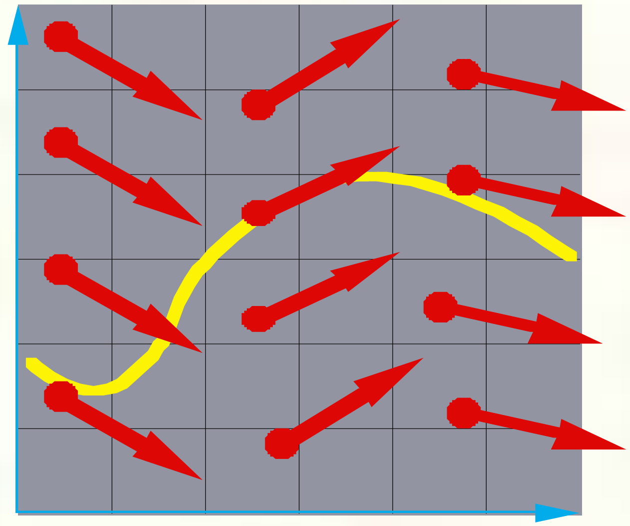


- Suppose the velocity is actually dictated by some driving function **g**:

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

Vector fields

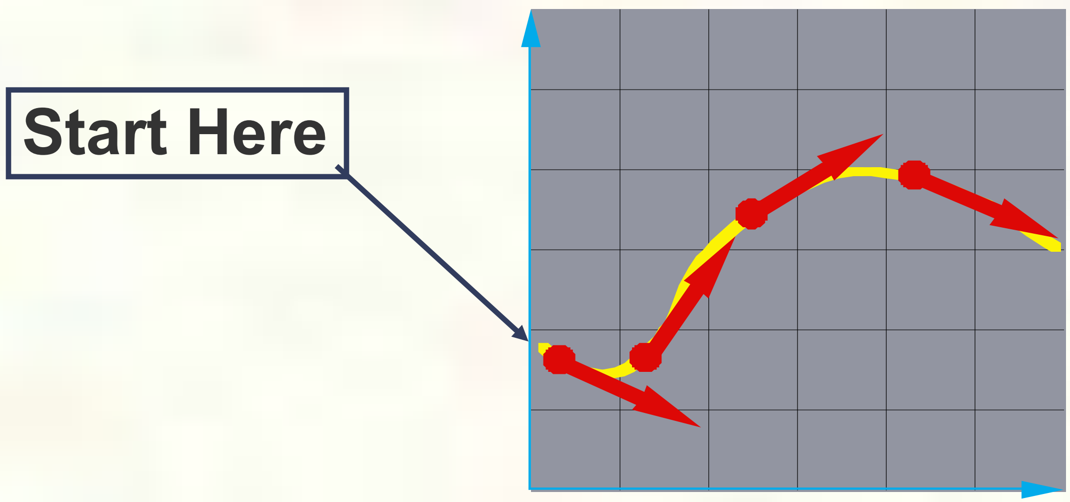
- At any moment in time, the function **g** defines a vector field over **x**:



- How does our particle move through the vector field?

Diff eqs and integral curves

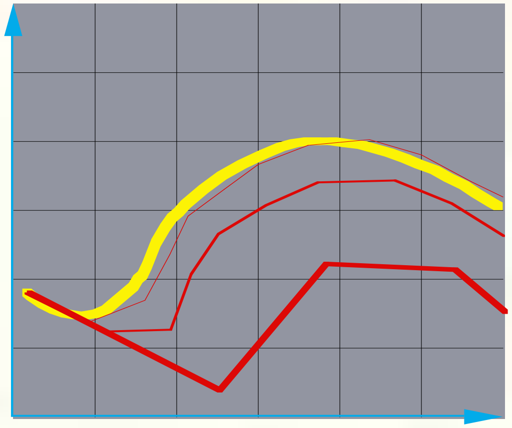
- The equation $$\dot{\mathbf{x}} = g(\vec{\mathbf{x}}, t)$$
  is actually a **first order differential equation**.
- We can solve for **x** through time by starting at an initial point and stepping along the vector field:



- This is called an **initial value problem** and the solution is called an **integral curve**.

Euler's method

- One simple approach is to choose a time step, Δt, and take linear steps along the flow: $$\vec{\mathbf{x}}(t+\Delta t) = \vec{\mathbf{x}}(t) + \Delta t \cdot \dot{\vec{\mathbf{x}}}(t) = \vec{\mathbf{x}}(t) + \Delta t \cdot g(\vec{\mathbf{x}}, t)$$
- Writing as a time iteration: $$\vec{\mathbf{x}}^{i+1} = \vec{x}^{i} + \Delta t \cdot \vec{\mathbf{v}}^{i}$$
- This approach is called **Euler's method** and looks like:



- Properties:
  - Simplest numerical method
  - Bigger steps, bigger errors. Error ~ $O(\Delta t^2)$.
- 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, $\vec{\mathbf{a}} \equiv \ddot{\mathbf{x}} = \frac{d\vec{\mathbf{v}}}{dt} = \frac{d^2\vec{\mathbf{x}}}{dt^2}$
- The particle obeys Newton's law: $\vec{\mathbf{f}} = m\vec{\mathbf{a}} = m\ddot{\mathbf{x}}$
- 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:

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

Second order equations

This equation:

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

is a **second order differential equation**.

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

We can rewrite this as:

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

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

Phase space

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

- 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.

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

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

Differential equation solver

Starting with:

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

Applying Euler's method:

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

And making substitutions:

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

Writing this as an iteration, we have:

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

Again, performs poorly for large Δt.

Verlet Integration

- Also called Størmer's Method
  - Invented by Delambre (1791), Størmer (1907), Cowell and Crommelin (1909), Verlet (1960) and probably others
- More stable than Euler's method (time-reversible as well)

Particle structure

How do we represent a particle?

Position in phase space

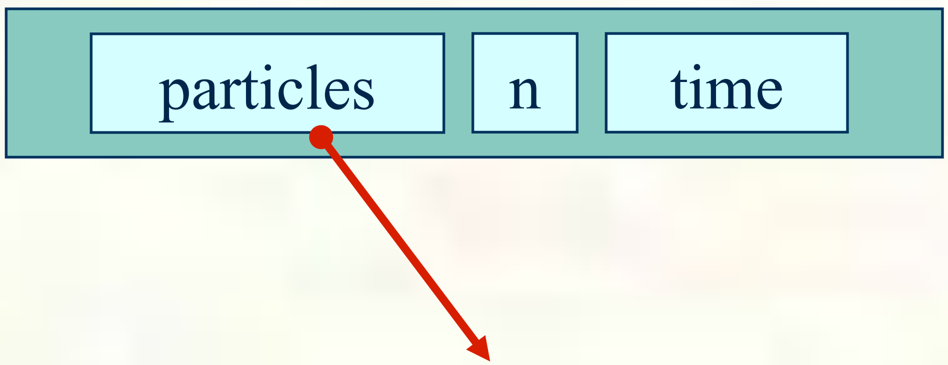
$$\begin{bmatrix} \mathbf{x} \\ \mathbf{v} \\ \mathbf{f} \\ m \end{bmatrix}$$

- $\mathbf{x}$ ← position
- $\mathbf{v}$ ← velocity
- $\mathbf{f}$ ← force accumulator
- m ← mass

Single particle solver interface

Particle systems

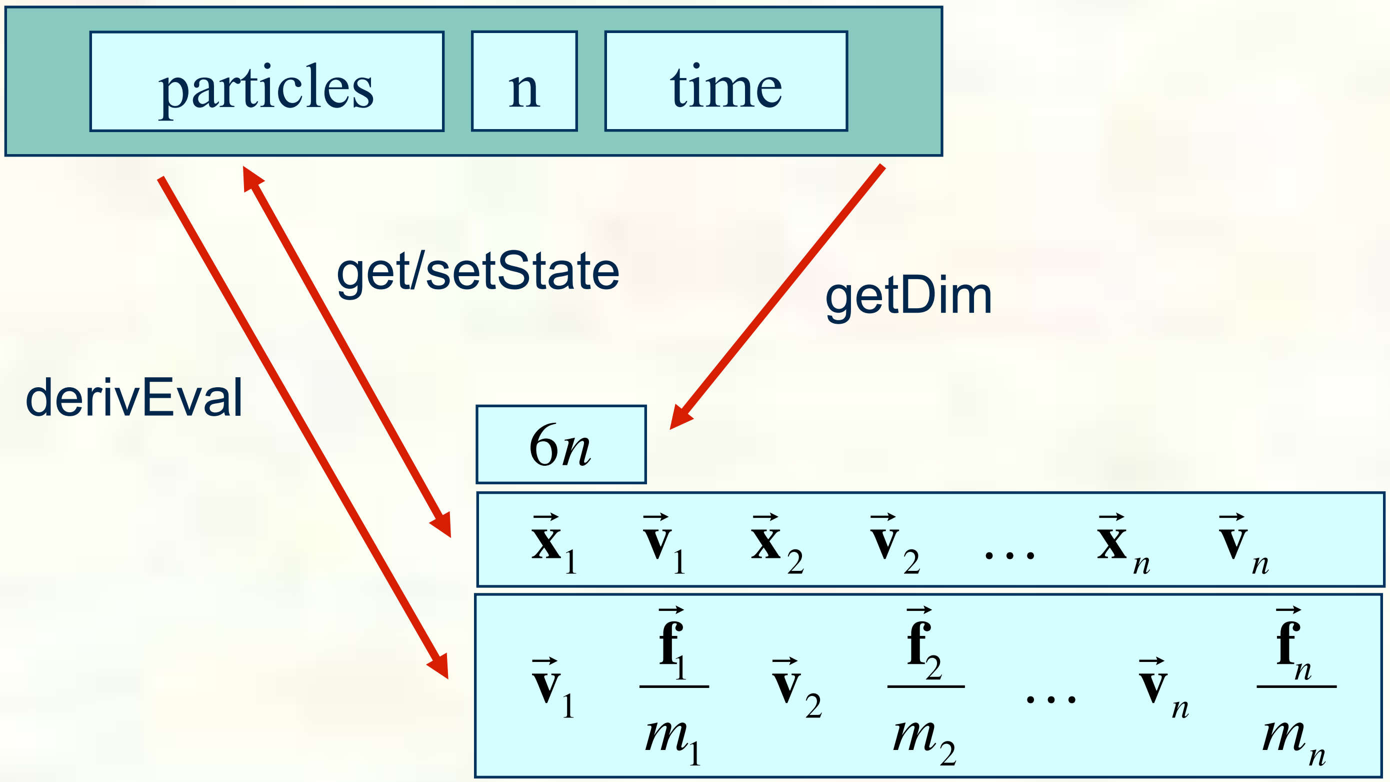
In general, we have a particle system consisting of *n* particles to be managed over time:



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

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:

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

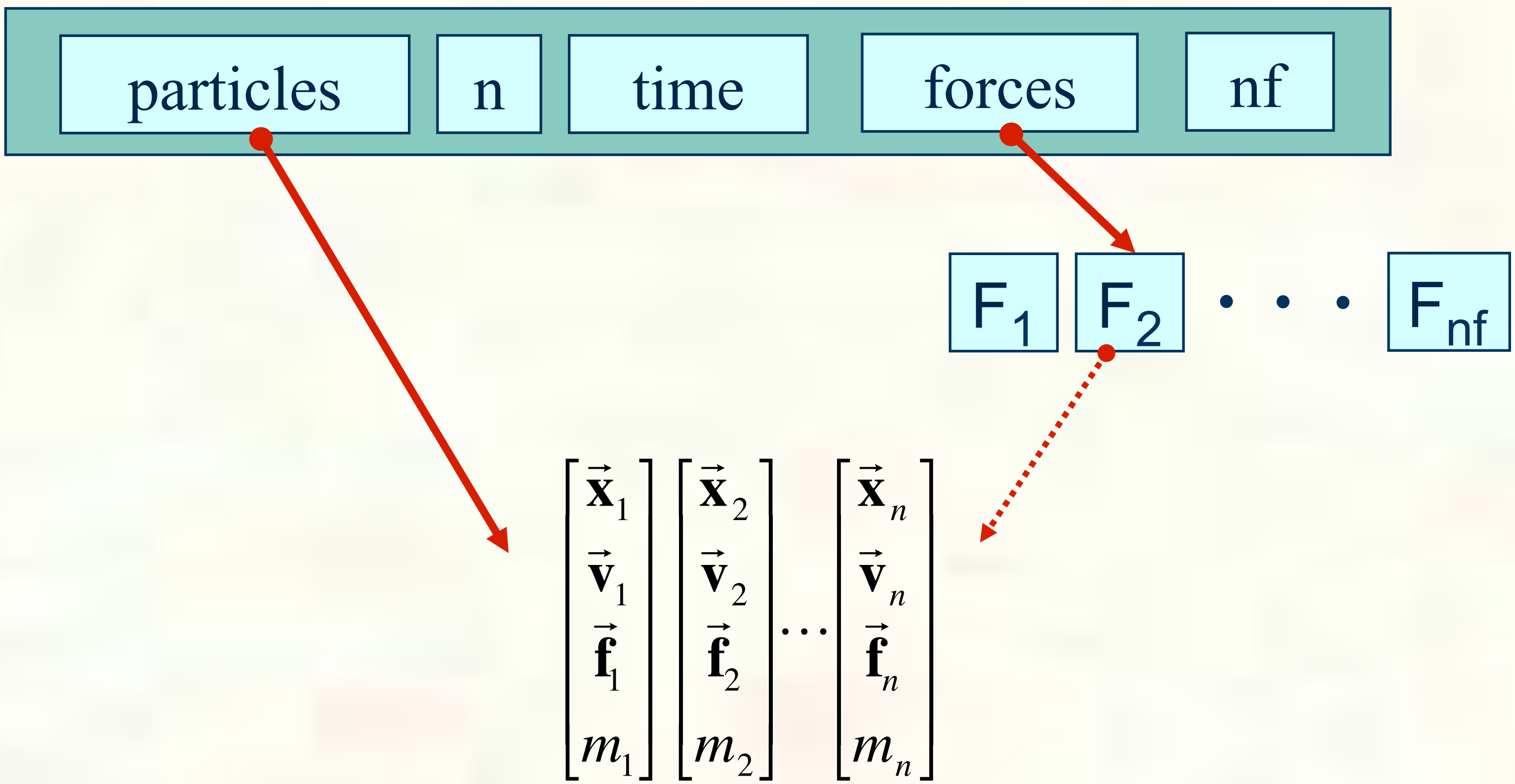
Forces

- 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)
  - Combinations (Damped 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:



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

Gravity and viscous drag

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

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

```
p->f += p->m * F->G
```

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

$$\vec{\mathbf{f}}_{drag} = -k\vec{\mathbf{v}}$$

```
p->f -= F->k * p->v
```

Damped spring

Recall the equation for the force due to a spring: $f = -k_{spring}(|\Delta\vec{\mathbf{x}}| - r)$

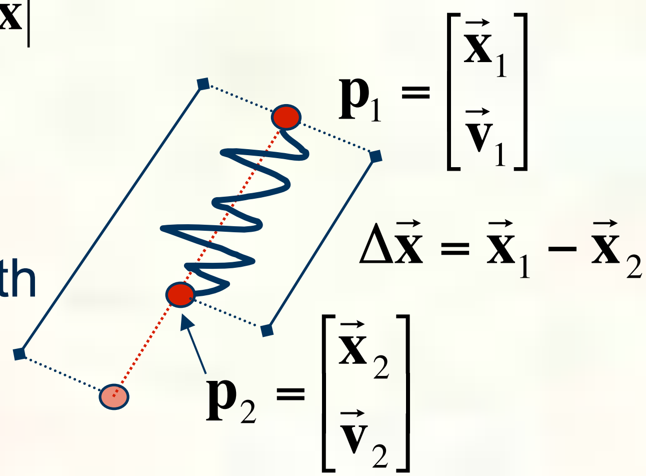
We can augment this with damping: $f = -\left[k_{spring}(|\Delta\vec{\mathbf{x}}| - r) + k_{damp}|\vec{\mathbf{v}}|\right]$

The resulting force equations for a spring between two particles become:

$$\vec{\mathbf{f}}_1 = -\left[k_{spring}\left(|\Delta\vec{\mathbf{x}}| - r\right) + k_{damp}\left(\frac{\Delta\vec{\mathbf{v}} \bullet \Delta\vec{\mathbf{x}}}{|\Delta\vec{\mathbf{x}}|}\right)\right]\frac{\Delta\vec{\mathbf{x}}}{|\Delta\vec{\mathbf{x}}|}$$

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

r = rest length



derivEval

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

1 Clear force accumulators

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

2 Apply forces to particles

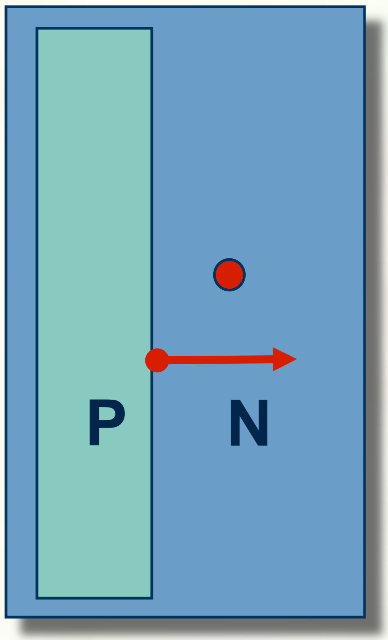
F_1 F_2 F_3 ••• F_{nf}

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

3 Return derivatives to solver

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

Bouncing off the walls

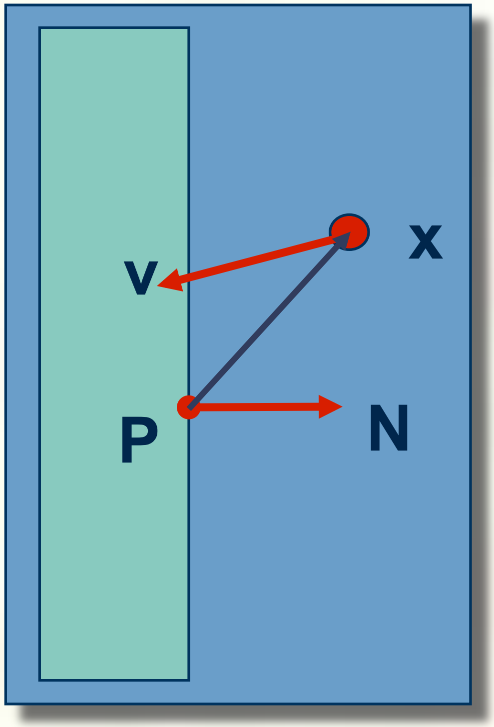


- Add-on for a particle simulator
- For now, just simple point-plane collisions

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

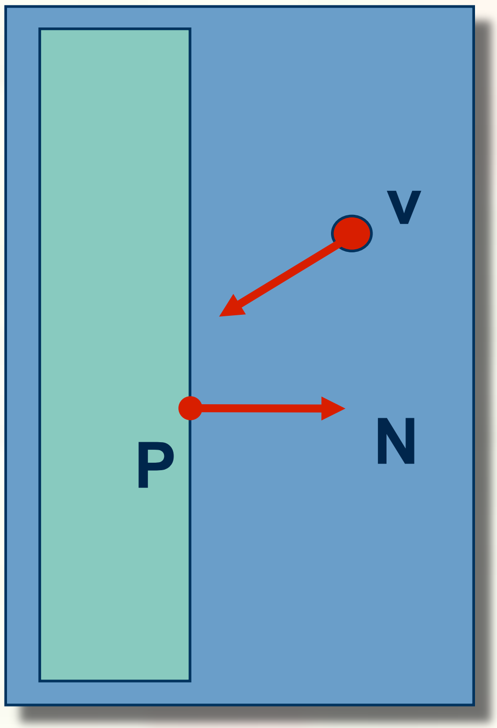
Collision Detection

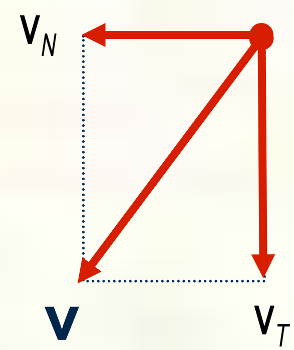
How do you decide when you've crossed a plane?



Normal and tangential velocity

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

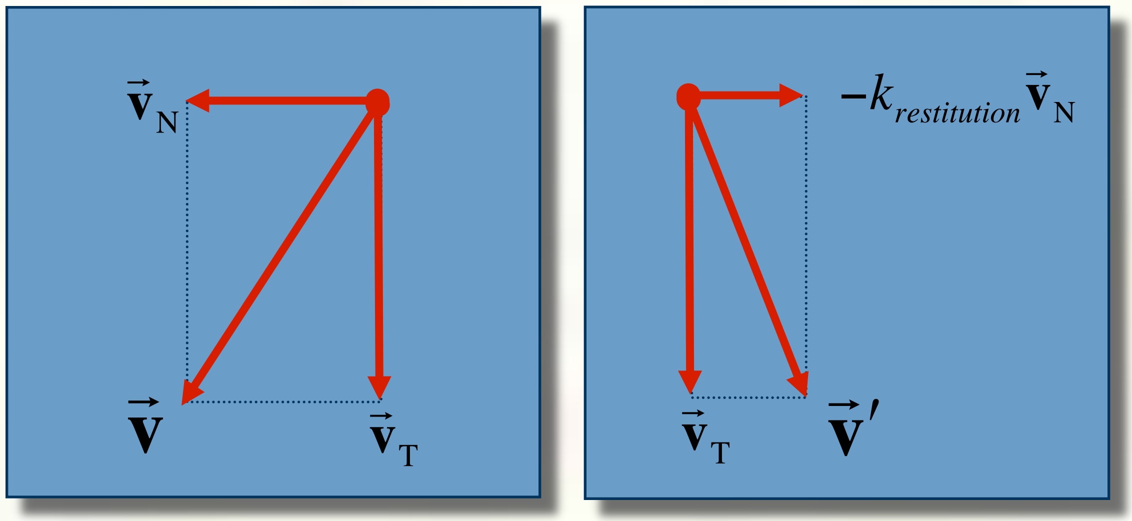




$$\vec{\mathbf{v}}_N = \left(\vec{\mathbf{N}} \bullet \vec{\mathbf{v}}\right)\vec{\mathbf{N}}$$

$$\vec{\mathbf{v}}_T = \vec{\mathbf{v}} - \vec{\mathbf{v}}_N$$

Collision Response



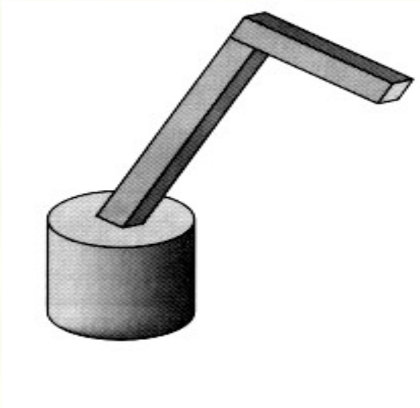
before after

$$\vec{\mathbf{v}}' = \vec{\mathbf{v}}_\mathrm{T} - k_{restitution}\vec{\mathbf{v}}_\mathrm{N}$$

Without backtracking, the response may not be enough to bring a particle to the other side of a wall.
In that case, detection should include a velocity check:

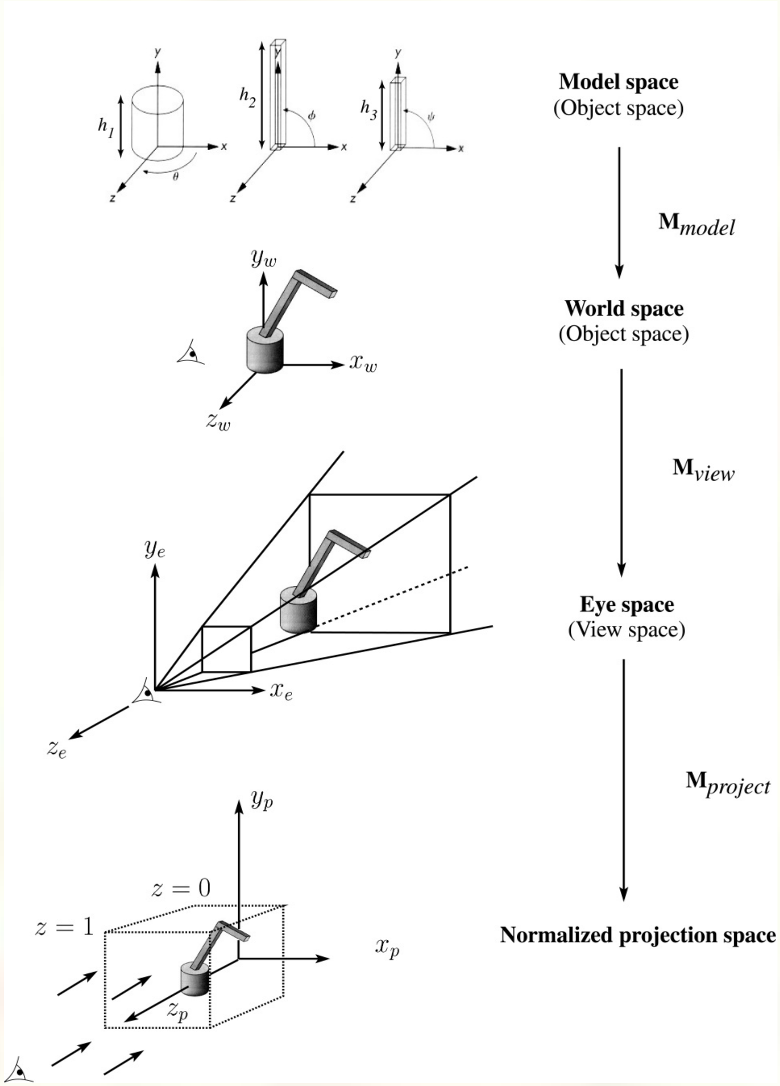
Particle frame of reference

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



- How would we go about starting the particles from the right place?
- First, we have to look at the coordinate systems in the OpenGL pipeline…

The OpenGL geometry pipeline



Projection and modelview matrices

- Any piece of geometry will get transformed by a sequence of matrices before drawing:

$$\mathbf{p}' = \mathbf{M}_{\text{project}}\, \mathbf{M}_{\text{view}}\, \mathbf{M}_{\text{model}}\, \mathbf{p}$$

- The first matrix is OpenGL's GL_PROJECTION matrix.
- The second two matrices, taken as a product, are maintained on OpenGL's GL_MODELVIEW stack:

$$\mathbf{M}_{\text{modelview}} = \mathbf{M}_{\text{view}}\, \mathbf{M}_{\text{model}}$$

Robot arm code, revisited

- Recall that the code for the robot arm looked something like:

```
glRotatef( theta, 0.0, 1.0, 0.0 );
base(h1);
glTranslatef( 0.0, h1, 0.0 );
glRotatef( phi, 0.0, 0.0, 1.0 );
upper_arm(h2);
glTranslatef( 0.0, h2, 0.0 );
glRotatef( psi, 0.0, 0.0, 1.0 );
lower_arm(h3);
```

- All of the GL calls here modify the modelview matrix.
- Note that even before these calls are made, the modelview matrix has been modified by the viewing transformation, $\mathbf{M}_{view}$.

Computing particle launch point

To find the world coordinate position of the end of the robot arm, you need to follow a series of steps:

1. Figure out what $\mathbf{M}_{view}$ before drawing your model.

```
Mat4f matCam = ps>glGetMatrix(GL_MODELVIEW_MATRIX);
```

2. Draw your model and add one more transformation to the tip of the robot arm.

```
glTranslatef( 0.0, h3, 0.0 );
```

3. Compute $\mathbf{M}_{model} = \mathbf{M}_{view}^{-1}\mathbf{M}_{modelview}$

```
Mat4f particleXform = ps->getWorldXform( matCam);
```

4. Transform a point at the origin by the resulting matrix.

```
Vec4f particleOrigin = particleXform * Vec4f(0,0,0,1);
// 4th coordinate should be 1.0 -- ignore
```

Now you're ready to launch a particle from that last computed point!

Next lecture

- Topic:

  Parametric Curves:
  C2 interpolating curves.

  How can we make splines that interpolate the control points, and have C2 continuity everywhere?

- Reading:

  • Bartels, Beatty, and Barsky. An Introduction to Splines for use in Computer Graphics and Geometric Modeling, 1987.
  [Course reader, pp. 239-247]