Particle Systems

CSE 457, Autumn 2003
Graphics

http://www.cs.washington.edu/education/courses/457/03au/

Reading

- Readings (also linked on our otherlinks page)
  » SIGGRAPH 2001 course notes on Physically Based Modeling, Witkin and Baraff, http://www.pixar.com/companyinfo/research/pbm2001/
    - *Particle System Dynamics*
    - *Differential Equation Basics*
- Other References
  » 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 or spring behaviors).
- Particle systems can be used to simulate all sorts of physical phenomena:

Overview

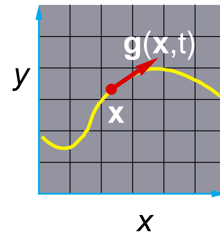
» A single particle
» Particle systems
» Forces: gravity, springs
» Collision detection

Particle in a flow field

We begin with a single particle with:

» Position $\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}$

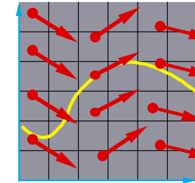
» Velocity $\mathbf{v} \equiv \dot{\mathbf{x}} = \frac{d\mathbf{x}}{dt} = \begin{bmatrix} dx/dt \\ dy/dt \end{bmatrix}$



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**:

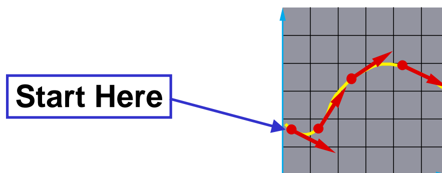


How does our particle move through the vector field?

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:



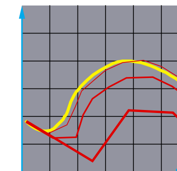
This is called an **initial value problem** and the solution is called an **integral curve**.

Euler's method

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

$$\begin{aligned} \mathbf{x}(t + \Delta t) &= \mathbf{x}(t) + \Delta t \cdot \dot{\mathbf{x}}(t) \\ &= \mathbf{x}(t) + \Delta t \cdot \mathbf{g}(\mathbf{x}, t) \end{aligned}$$

This approach is called **Euler's method** and the solutions it generates look like this:



Properties:

Simplest numerical method

Bigger steps, bigger errors. Error ~ $O(\Delta t^2)$.

Need to take pretty small steps, so not very efficient.

Better methods exist, e.g., Runge-Kutta

Particle in a force field

Now consider a particle in a force field **f**.

In this case, the particle has:

Mass, m

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

The particle obeys Newton's law:

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

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

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:

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

where we have used 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 1st-order differential equation.

Differential equation solver

Starting with $\begin{bmatrix} \mathbf{x} \\ \mathbf{v} \end{bmatrix}$ and $\begin{bmatrix} \dot{\mathbf{x}} \\ \dot{\mathbf{v}} \end{bmatrix} = \begin{bmatrix} \mathbf{v} \\ \mathbf{f}/m \end{bmatrix}$ find the next $\begin{bmatrix} \mathbf{x} \\ \mathbf{v} \end{bmatrix}$

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:

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

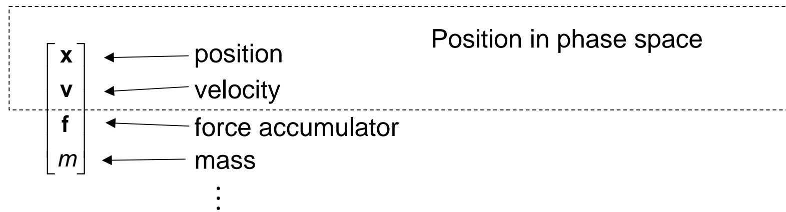
Writing this as an iteration, we have:

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

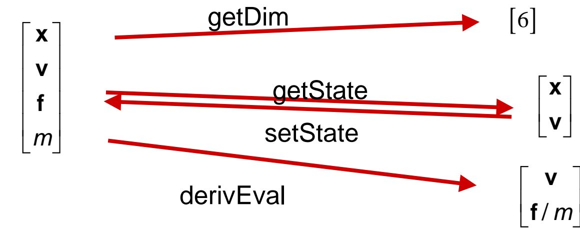
(Again, has problems with large Δt.)

Particle structure

How do we represent a particle?



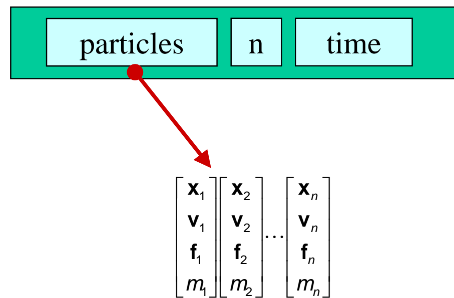
Single particle solver interface



update forces
D = calculate new derivatives with derivEval()
S = getState()
S_{new} = S+ΔtD
setState(S_{new})

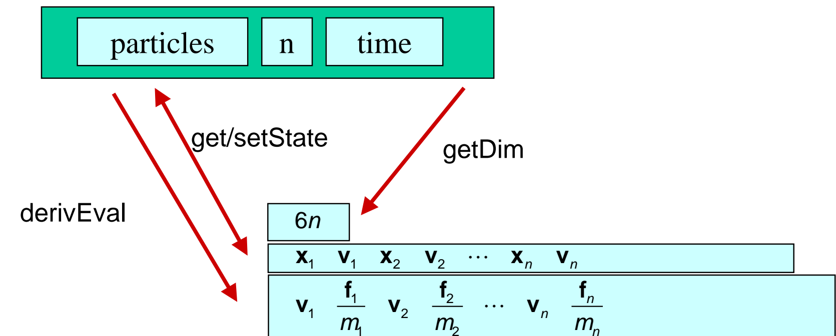
Particle systems

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



Particle system solver interface

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



Particle system diff. eq. solver

Thus, we start with:

$$\begin{bmatrix} \mathbf{x}_1^i \\ \mathbf{v}_1^i \\ \vdots \\ \mathbf{x}_n^i \\ \mathbf{v}_n^i \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} \dot{\mathbf{x}}_1 \\ \dot{\mathbf{v}}_1 \\ \vdots \\ \dot{\mathbf{x}}_n \\ \dot{\mathbf{v}}_n \end{bmatrix} = \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{f}_1 / m_1 \\ \vdots \\ \mathbf{v}_n \\ \mathbf{f}_n / m_n \end{bmatrix}$$

And can solve, using the Euler method:

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

Forces

Each particle can experience a force which sends it on its merry way.

Where do these forces come from? Some examples:

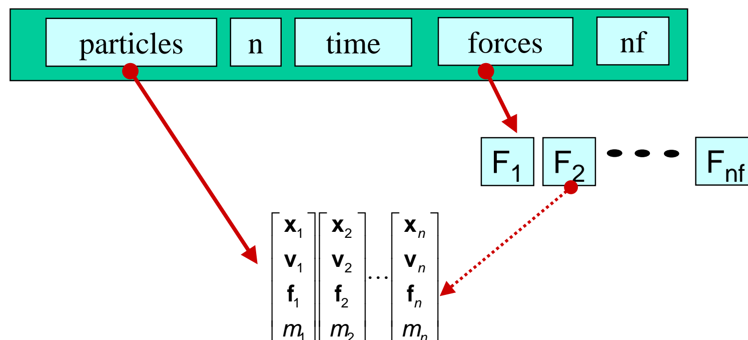
- Constant (gravity)
- Position/time dependent (force fields)
- Velocity-dependent (drag)
- N-ary (springs)

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:

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

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

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

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

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

These are examples of unary forces.

Damped spring

A spring is a simple examples of an "N-ary" force.   Recall the equation for the force due to a spring:

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

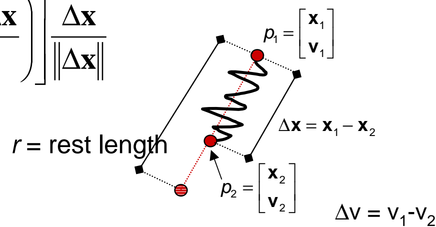
We can augment this with damping:

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

The resulting force equations become:

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

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



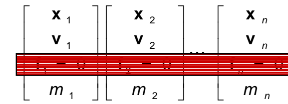
r = rest length

Store spring info in a force object.

$\Delta v = v_1 - v_2$

derivEval

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

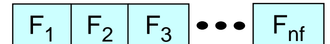


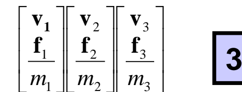
1

Clear force accumulators

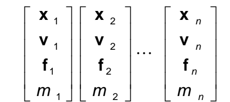
Calculate particle forces

2



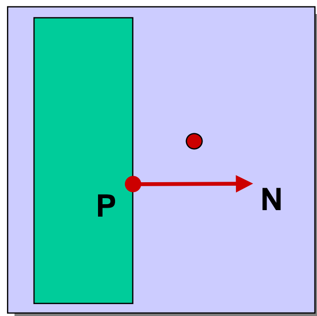


3



Return [**v**,**f**/*m*,…] to solver

Bouncing off the walls

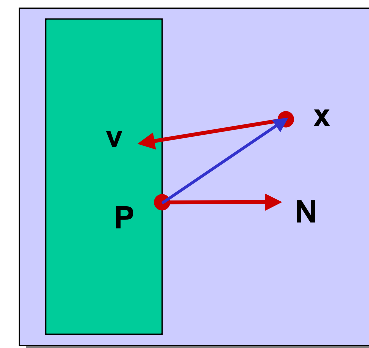


» Add-on for a particle simulator
» For now, just simple point-plane collisions

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

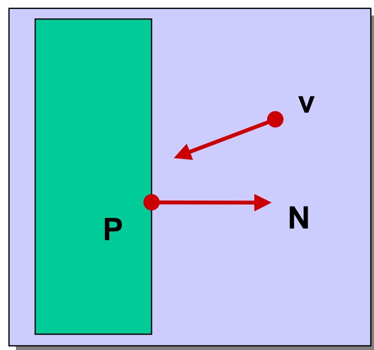
Collision Detection

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



Normal and tangential velocity

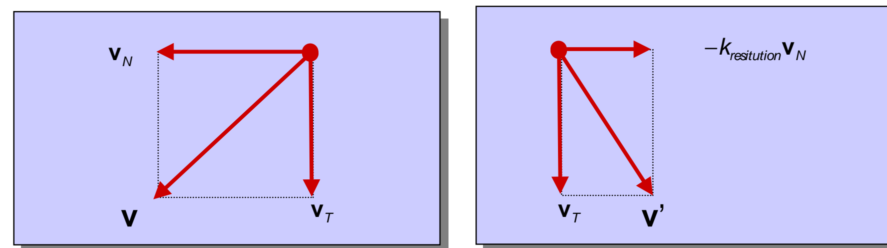
To compute the collision response, we need to consider the normal and tangential components of a particle's velocity.



$\mathbf{v}_N$ $\mathbf{v}$ $\mathbf{v}_T$

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

Collision Response



before after

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

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

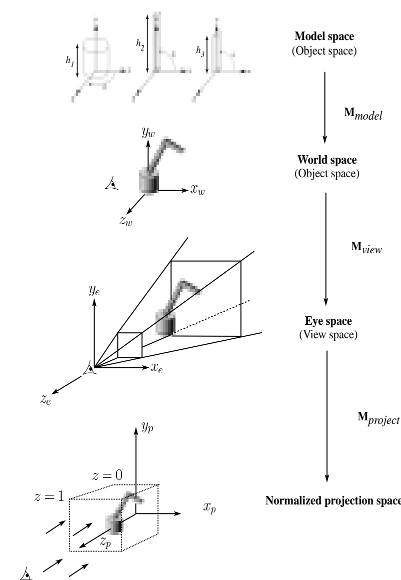
Particle frame of reference

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:

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

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:

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

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 the particle launch point

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

1. Remember what $\mathbf{M}_{view}$ is before drawing your model. (glGetMatrix uses glGetFloatv to get the modelview matrix and returns the transpose as Mat4f.)

```
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 $\mathbf{M}_{model} = \mathbf{M}^{-1}_{view}\mathbf{M}_{modelview}$

```
Mat4f matModelView = glGetMatrix(GL_MODELVIEW_MATRIX);
Mat4f matModel = matCam.inverse() * matModelView;
```

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

```
Vec3f particleOrigin = matModel * Vec3f(0,0,0);
```

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

Summary

- What you should take away from this lecture:
  - » The meanings of all the **boldfaced** terms
  - » Euler method for solving differential equations
  - » Combining particles into a particle system
  - » Physics of a particle system
  - » Various forces acting on a particle
  - » Simple collision detection
  - » How to hook your particle system into the coordinate frame of your model