Particle Systems

Brian Curless
CSEP 557
Spring 2019

Reading

Required:

- Witkin, *Particle System Dynamics*, SIGGRAPH '01 course notes on Physically Based Modeling. (online handout)
- Witkin and Baraff, *Differential Equation Basics*, SIGGRAPH '01 course notes on Physically Based Modeling. (online handout)

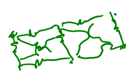
Optional

- Hockney 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.

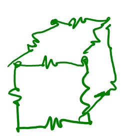
What are particle systems?

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:



Snow
rain
fire
volumetric light
cloth
dust
astronomical
bees
smoke
jello



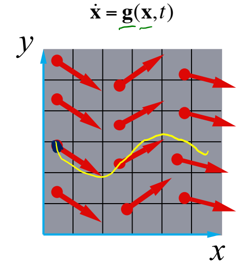
Particle in a flow field

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 a driving function, a vector flow field, **g**:

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



If a particle starts at some point in that flow field, how should it move?

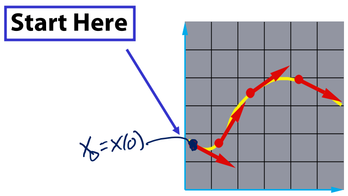
Diff eqs and integral curves

The equation

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

is actually a **first order differential equation**.

We can solve for $\mathbf{x}$ 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

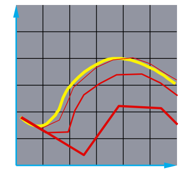
One simple approach is to choose a time step, Δt, and take linear steps along the flow:

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

Writing as a time iteration:

$$\mathbf{x}^{i+1} = \mathbf{x}^i + \Delta t \cdot \mathbf{g}^i \quad \text{with} \quad \mathbf{g}^i \equiv \mathbf{g}(\mathbf{x}^i, t = i\Delta t)$$

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., adaptive timesteps, 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, $\mathbf{a} \equiv \ddot{\mathbf{x}} = \dot{\mathbf{v}} = \dfrac{d\mathbf{v}}{dt} = \dfrac{d^2\mathbf{x}}{dt^2}$

The particle obeys Newton's law:

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

So, given a force, we can solve for the acceleration:

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

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:

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

Second order equations

This equation:

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

is a **second order differential equation**.

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

We can rewrite the second order equation as:

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

where we substitute in $\mathbf{v}$ and its derivative to get a pair of **coupled first order equations**.

Phase space

$\begin{bmatrix} \mathbf{x} \\ \mathbf{v} \end{bmatrix}$ Concatenate $\mathbf{x}$ and $\mathbf{v}$ 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:

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

Applying Euler's method:

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

And making substitutions:

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

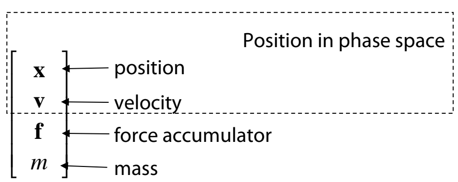
Writing this as an iteration, we have:

$$\mathbf{x}^{i+1} = \mathbf{x}^i + \Delta t \cdot \mathbf{v}^i$$
$$\mathbf{v}^{i+1} = \mathbf{v}^i + \Delta t \cdot \frac{\mathbf{f}^i}{m} \qquad \text{with} \quad \mathbf{f}^i \equiv \mathbf{f}(\mathbf{x}^i, \mathbf{v}^i, t)$$

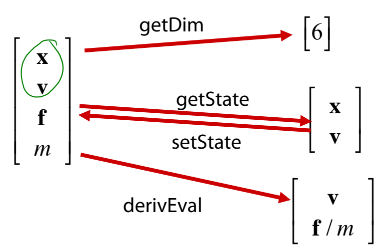
Again, performs poorly for large Δt.

Particle structure

How do we represent a particle?



Single particle solver interface



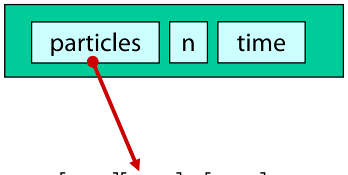
$S_{old} = \text{getState}(\,)$

$S_{new} = S_{old} + \Delta t \cdot \text{derivEval}(\,)$

$\text{setState}(S_{new})$

Particle systems

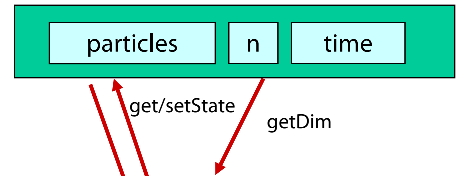
In general, we have a particle system consisting of n particles to be managed over time:



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

Particle system solver interface

For n particles, the solver interface now looks like:



$$\begin{bmatrix} \mathbf{x}_1 & \mathbf{v}_1 & \mathbf{x}_2 & \mathbf{v}_2 & \cdots & \mathbf{x}_n & \mathbf{v}_n \end{bmatrix}$$

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

Particle system diff. eq. solver

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

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

setState(getState() + Δt · derivEval())

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?

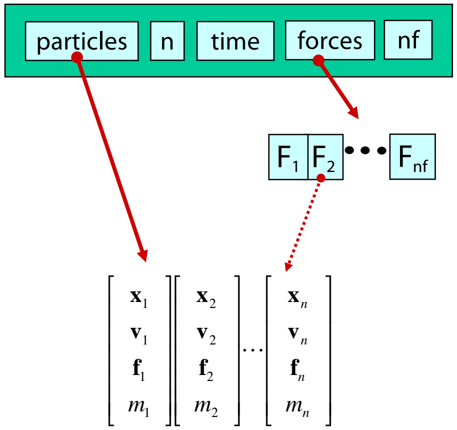
Sum them up

"principle of superposition"

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
```

$f_{grav} + f_{drag} = 0$

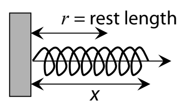
$mG - k_{drag} v = 0$

$v_{termin} = \frac{mG}{k_{drag}}$

Damped spring

A spring is a simple examples of an "N-ary" force.
Recall the equation for the force due to a 1D spring:

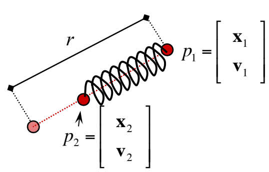
$$f = -k_{spring}(x - r)$$



With damping:

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

In 2D or 3D, we get:



$$p_1 = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{v}_1 \end{bmatrix} \qquad p_2 = \begin{bmatrix} \mathbf{x}_2 \\ \mathbf{v}_2 \end{bmatrix}$$

$$\Delta\mathbf{x} = \mathbf{x}_1 - \mathbf{x}_2$$

$$\Delta\hat{\mathbf{x}} = \frac{\Delta\mathbf{x}}{\|\Delta\mathbf{x}\|}$$

$$\Delta\mathbf{v} = \mathbf{v}_1 - \mathbf{v}_2$$

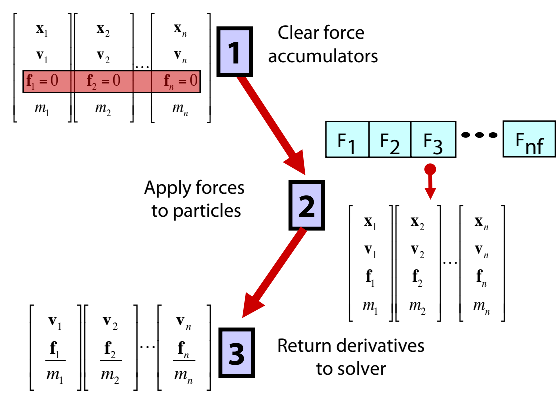
$$f_1 = -\left[k_{spring}(\|\Delta\mathbf{x}\| - r) + k_{damp}(\Delta\mathbf{v}\cdot\Delta\hat{\mathbf{x}})\right]\Delta\hat{\mathbf{x}}$$

$$f_2 = -f_1$$

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

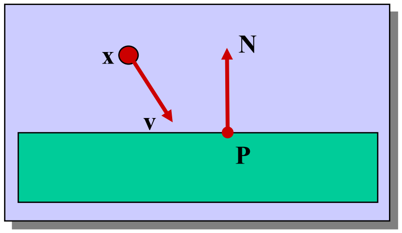
1. Clear forces
   - Loop over particles, zero force accumulators
2. Calculate forces
   - Sum all forces into accumulators
3. Return derivatives
   - Loop over particles, return $\mathbf{v}$ and $\mathbf{f}/m$



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.

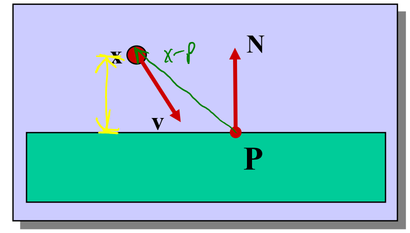


A plane is fully specified by any point $\mathbf{P}$ on the plane and its normal $\mathbf{N}$.

(Let's assume $\mathbf{N}$ is normalized, $||\mathbf{N}|| = 1$.)

Collision Detection

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

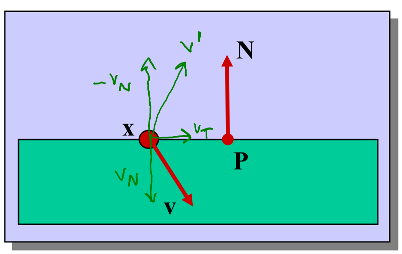


$(x-p)\cdot N = 0 \Rightarrow$ exact collision

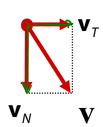
$(x-p)\cdot N =$ signed distance of x from plane

Normal and tangential velocity

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



$v' = v_T - v_N$

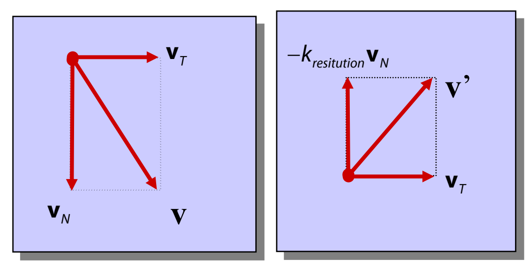


$v = v_N + v_T$

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

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

Collision Response



before

after

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

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

where $k_{\text{restitution}} \in [0,1]$.

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.

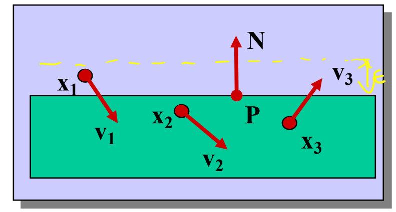
The most expensive is **backtracking**: determine if a collision must have occurred, and then roll back the simulation to the moment of contact.

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 during a timestep?

$(x-p)\cdot N \leq 0$



A problem with this approach is that particles will disappear under the surface. We can reduce this problem by essentially offsetting the surface:

$(x-p)\cdot N \leq \epsilon$

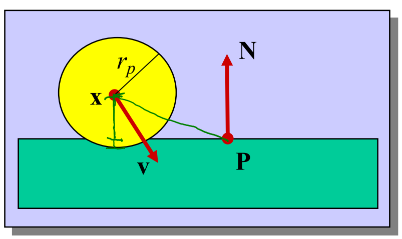
both true $\Rightarrow$ collision

Also, 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:

$(v\cdot N) < 0$

Spherical particles

We will allow our particles to have finite radius:



The basic particle-inside-surface test then becomes:

$(x-p)\cdot N \leq r_p$

We still pad this test to limit how much particles cross the surface before being reflected:

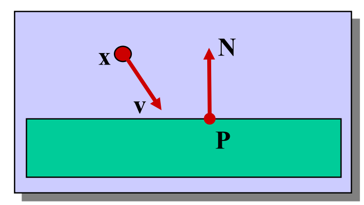
$(x-p)\cdot N \leq r_p + \epsilon$

The velocity test is unchanged.

More complicated collision response

Another solution is to modify the update scheme to:

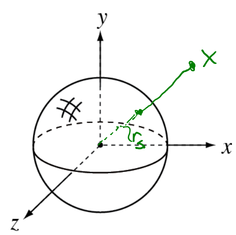
- detect the future time and point of collision
- reflect the particle within the time-step



Particle-sphere collision

Suppose a particle collides with a sphere of radius r_s sitting at the origin:

- How would we detect this collision?
- What normal should we use for collision response?



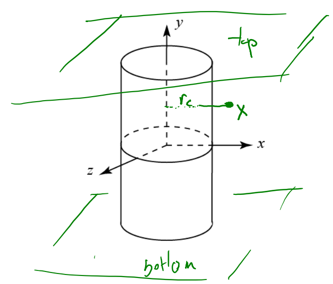
$\|x\| \le r_s + \epsilon + r_p$

$N = \dfrac{x}{\|x\|}$

Particle-cylinder collision

Suppose a particle collides with a cylinder of height h and radius r_c sitting at the origin:

- How would we detect this collision?
- What normal should we use for collision response?



x is:

below top

above bottom

and w/in r_c of the y-axis

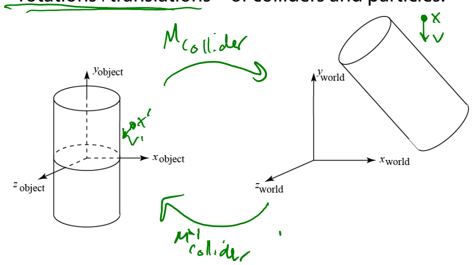
"pad" this w/ $r_p + \epsilon$

Use normal of the closest surface

Collision in model coordinates

As with ray tracing, it convenient to handle collisions in canonical object coordinates.

For the project, we will assume only rigid transformations – rotations+translations – of colliders and particles.



$x' = M^{-1}_{collider}\, x$

$v' = M^{-1}_{collider}\, v$

Do collision analysis in object coords.

Then $v'_{update} \longrightarrow v_{update} = M_{collider}\, v'_{update}$

Collision in model coordinates (cont'd)

Note: we assume for the project that scaling will be handled only through the provided dimensions of colliders and particles (height, radius, etc.).

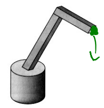
If uniform scale is additionally applied, then it must be extracted from the collider transformation and applied to the epsilons and particle sphere radius when operating in the collider's object coordinates.

If non-uniform scale is applied, then things get trickier, since, e.g., spherical particles may become oriented (non-axis-aligned) ellipsoids, requiring more complex collision analysis.

Extra credit… ☺

Particle frame of reference

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

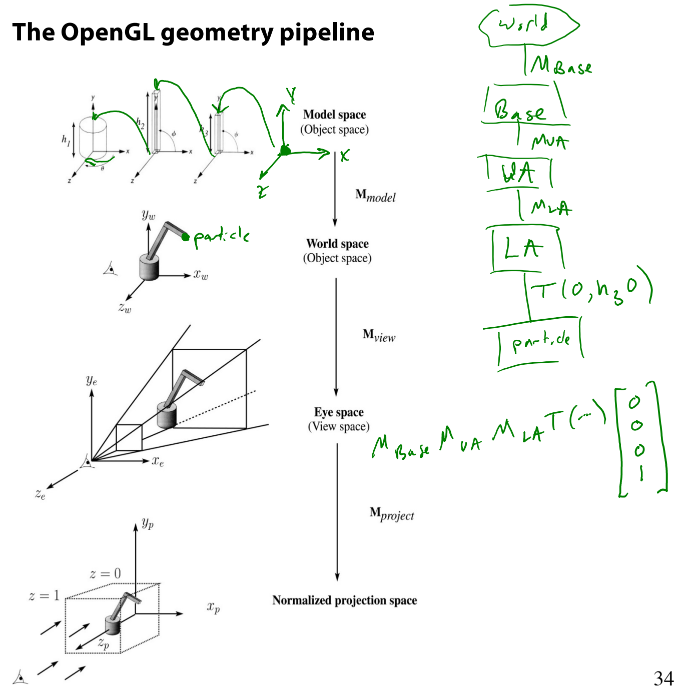


How would we go about starting the particles from the right place?

First, we have to look at the coordinate systems in the OpenGL pipeline…

33

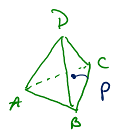
The OpenGL geometry pipeline



$M_{Base} M_{UA} M_{LA} T(\ldots) \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$

34

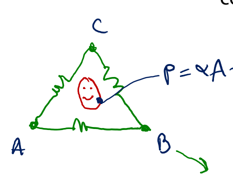
Summary

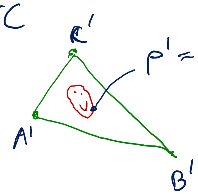


$P = \alpha A + \beta B + \gamma C + \delta D$

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 with a plane and a sphere
- How to hook your particle system into the coordinate frame of your model





35