

Computer Animation Particle systems

- Some slides courtesy of Jovan Popovic & Ronen Barzel



MIT EECS 6.837

What is a particle system?

- Collection of many small simple particles
- Particle motion influenced by force fields
- Particles created by *generators*
- Particles often have *lifetimes*
- Used for, e.g:
 - sand, dust, smoke, sparks, flame, water, ...

MIT EECS 6.837

ODEs and numerical integration

$$\frac{d\mathbf{X}(t)}{dt} = f(\mathbf{X}(t), t)$$

- Given a function $f(\mathbf{X}, t)$ compute $\mathbf{X}(t)$
- Typically, *initial value problems*:
 - Given values $\mathbf{X}(t_0) = \mathbf{X}_0$
 - Find values $\mathbf{X}(t)$ for $t > t_0$
- We can use lots of standard tools

MIT EECS 6.837

Turn higher order ODE into 1st order

- E.g., Mechanics has 2nd order ODE:

$$\frac{d^2}{dt^2} x = \frac{1}{m} F$$

- Express as 1st order ODE by defining $v(t)$:

$$\begin{aligned} \frac{dx}{dt} x(t) &= v(t) \\ \frac{dv}{dt} v(t) &= \frac{1}{m} F(x, v, t) \end{aligned}$$

$$\mathbf{x} = \begin{pmatrix} x \\ v \end{pmatrix} \quad f(\mathbf{x}, t) = \begin{pmatrix} v \\ \frac{1}{m} F(x, v, t) \end{pmatrix}$$

- Reduces our numerical problem to 1st order, no need for complicated second-order solver
 - But would be useless for analytical derivation

MIT EECS 6.837

For a collection of 3D particles...

- Again, reduces everything to a simple 1-vector ODE solver

$$\mathbf{x} = \begin{pmatrix} p_x^{(1)} \\ p_y^{(1)} \\ p_z^{(1)} \\ v_x^{(1)} \\ v_y^{(1)} \\ v_z^{(1)} \\ p_x^{(2)} \\ p_y^{(2)} \\ p_z^{(2)} \\ v_x^{(2)} \\ v_y^{(2)} \\ v_z^{(2)} \\ \vdots \end{pmatrix} \quad f(\mathbf{x}, t) = \begin{pmatrix} v_x^{(1)} \\ v_y^{(1)} \\ v_z^{(1)} \\ \frac{1}{m} F_x^{(1)}(\mathbf{X}, t) \\ \frac{1}{m} F_y^{(1)}(\mathbf{X}, t) \\ \frac{1}{m} F_z^{(1)}(\mathbf{X}, t) \\ v_x^{(2)} \\ v_y^{(2)} \\ v_z^{(2)} \\ \frac{1}{m} F_x^{(2)}(\mathbf{X}, t) \\ \frac{1}{m} F_y^{(2)}(\mathbf{X}, t) \\ \frac{1}{m} F_z^{(2)}(\mathbf{X}, t) \\ \vdots \end{pmatrix}$$

MIT EECS 6.837

Still, a path through a field:

- $\mathbf{X}(t)$: path in multidimensional state space



- $\mathbf{X}(t)$ is an array/vector of numbers.

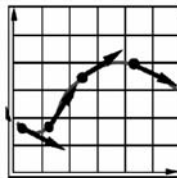
MIT EECS 6.837

Questions?

MIT EECS 6.837

Intuitive solution: take steps

- Current state X
- Examine $f(X,t)$ at (or near) current state
- Take a step to new value of X
- Most solvers do some form of this



MIT EECS 6.837

Euler's method

- Simplest and most intuitive.
- Define **step size** h
- Given $\mathbf{X}_0 = \mathbf{X}(t_0)$, take step:

$$t_1 = t_0 + h$$

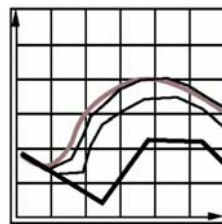
$$\mathbf{X}_1 = \mathbf{X}_0 + h f(\mathbf{X}_0, t_0)$$

- Piecewise-linear approximation to the curve

MIT EECS 6.837

Effect of step size

- Step size controls accuracy
- Smaller steps more closely follow curve
- For animation, may need to take many small steps per frame



Euler's method: inaccurate

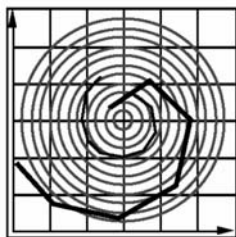
- Moves along tangent; can leave curve, e.g.:

$$f(\mathbf{X}, t) = \begin{pmatrix} -y \\ x \end{pmatrix}$$

- Exact solution is circle:

$$\mathbf{X}(t) = \begin{pmatrix} r \cos(t+k) \\ r \sin(t+k) \end{pmatrix}$$

- Euler's spirals outward
- no matter how small h is
 - will just diverge more slowly



MIT EECS 6.837

Euler's method: unstable

$$f(x, t) = -kx$$

- Exact solution is decaying exponential:

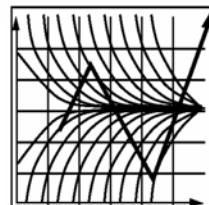
$$x(t) = x_0 e^{-kt}$$

- Limited step size:

$$x_1 = x_0(1 - hk)$$

$$\begin{cases} h \leq 1/k & \text{ok} \\ h > 1/k & \text{oscillates } \pm \\ h > 2/k & \text{explodes} \end{cases}$$

- If k is big, h must be small



MIT EECS 6.837

Analysis: Taylor series

- Expand exact solution $\mathbf{X}(t)$

$$\mathbf{X}(t_0 + h) = \mathbf{X}(t_0) + h \left(\frac{d}{dt} \mathbf{X}(t) \right) \Big|_{t_0} + \frac{h^2}{2!} \left(\frac{d^2}{dt^2} \mathbf{X}(t) \right) \Big|_{t_0} + \frac{h^3}{3!} (\dots) + \dots$$

- Euler's method approximates:

$$\mathbf{X}(t_0 + h) = \mathbf{X}_0 + h f(\mathbf{X}_0, t_0) \dots + O(h^2) \text{ error}$$

$$h \rightarrow h/2 \Rightarrow \text{error} \rightarrow \text{error}/4 \text{ per step} \times \text{twice as many steps} \\ \rightarrow \text{error}/2$$

- First-order method: Accuracy varies with h
- To get 100x better accuracy need 100x more steps

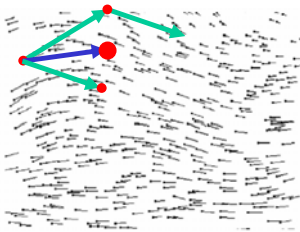
MIT EECS 6.837

Questions?

MIT EECS 6.837

Can we do better?

- Problem: f has varied along the step
- Idea: look at f at the arrival of the step and compensate for variation



MIT EECS 6.837

2nd order methods

- Let

$$\begin{aligned} f_0 &= f(\mathbf{X}_0, t_0) \\ f_1 &= f(\mathbf{X}_0 + h f_0, t_0 + h) \end{aligned}$$

- Then $\mathbf{X}(t_0 + h) = \mathbf{X}_0 + \frac{h}{2}(f_0 + f_1) + O(h^3)$

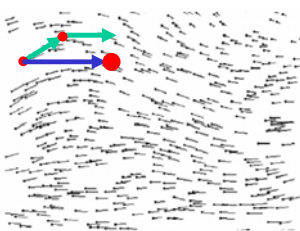
- This is the *trapezoid method*,
- AKA *improved Euler's method*

- Analysis omitted (see 6.839)

MIT EECS 6.837

Can we do better?

- Problem: f has varied along the step
- Idea: look at f at the arrival of the step and compensate for variation



MIT EECS 6.837

2nd-order methods continued...

- Could also have chosen

$$\Delta_{\mathbf{X}} = \frac{h}{2} f(\mathbf{X}_0, t_0) \text{ and } \Delta_t = \frac{h}{2}$$

- then rearrange the same way, let

$$\begin{aligned} f_0 &= f(\mathbf{X}_0, t_0) \\ f_m &= f(\mathbf{X}_0 + \frac{h}{2} f_0, t_0 + \frac{h}{2}) \end{aligned}$$

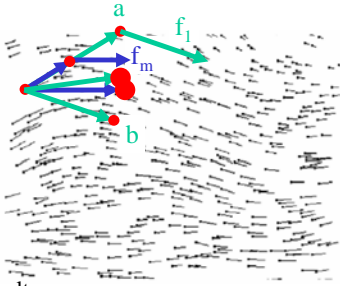
- and get $\mathbf{X}(t_0 + h) = \mathbf{X}_0 + h f_m + O(h^3)$

- This is the *midpoint method*

MIT EECS 6.837

Comparison

- **Midpoint:**
 - ½ Euler step
 - evaluate f_m
 - full step using f_m
- **Trapezoid:**
 - Euler step (a)
 - evaluate f_i
 - full step using f_i (b)
 - average (a) and (b)
- Not exactly same result
- Same order of accuracy



MIT EECS 6.837

Can we do even better?

- You bet!
- You will implement Runge Kutta for assignment 3
- See e.g. the Physically-Based Modeling Siggraph course notes, e.g. at <http://www.cs.berkeley.edu/~daf/games/webpage/SimList/Papers/physicallyBasedModeling-sg2002.pdf#search=%22physically-based%20modeling%22>
THE reference for anything related to physics and computer animation

MIT EECS 6.837

Questions?

MIT EECS 6.837

Particle Animation

```

AnimateParticles(n,  $\mathbf{y}_0$ ,  $t_0$ ,  $t_f$ )
{
     $\mathbf{y} = \mathbf{y}_0$ 
     $t = t_0$ 
    DrawParticles(n,  $\mathbf{y}$ )
    while( $t \neq t_f$ ) {
         $\mathbf{f} = \text{ComputeForces}(\mathbf{y}, t)$ 
         $d\mathbf{y}/dt = \text{AssembleDerivative}(\mathbf{y}, \mathbf{f})$ 
        //there could be multiple force fields
         $\{\mathbf{y}, t\} = \text{ODESolverStep}(6n, \mathbf{y}, d\mathbf{y}/dt)$ 
        DrawParticles(n,  $\mathbf{y}$ )
    }
}
    
```

MIT EECS 6.837

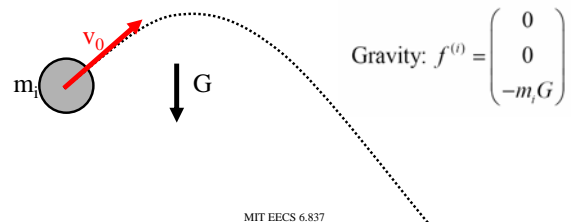
What is a force?

- Forces can depend on location, time, velocity
- Implementation:
 - **Force** is a (super)class
 - Computes force function for each particle \mathbf{p}
 - Adds computed force to total in $\mathbf{p.f}$
 - There can be multiple force sources

MIT EECS 6.837

Forces: gravity on Earth

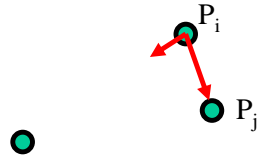
- depends only on particle mass:
- $f(\mathbf{X}, t) = \text{constant}$
- for smoke, flame: make gravity point up!



MIT EECS 6.837

Forces gravity for N-body problem

- Depends on all other particles
- Opposite for pairs of particles
- Force in the direction of $p_i p_j$ with magnitude inversely proportional to square distance
- $F_{ij} = G m_i m_j / r^2$



MIT EECS 6.837

Forces: damping

$$f^{(i)} = -d v^{(i)}$$

- force on particle i depends only on velocity of i
- force opposes motion
- removes energy, so system can settle
- small amount of damping can stabilize solver
- too much damping makes motion like in glue

MIT EECS 6.837

Forces: spatial fields

Spatial fields: $f^{(i)} = f(x^{(i)}, t)$

- force on particle i depends only on position of i
- arbitrary functions:
 - wind
 - attractors
 - repulsers
 - vortexes
- can depend on time
- note: these add energy, may need damping

MIT EECS 6.837

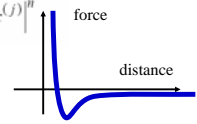
Forces: spatial interaction

Spatial interaction: $f^{(i)} = \sum_j f(x^{(i)}, x^{(j)})$

- e.g., approximate fluid: Lennard-Jones force:

$$f(x^{(i)}, x^{(j)}) = \frac{k_1}{|x^{(i)} - x^{(j)}|^m} - \frac{k_2}{|x^{(i)} - x^{(j)}|^n}$$

- Repulsive + attractive force
- $O(N^2)$ to test all pairs
 - usually only local
 - Use buckets to optimize. Cf. 6.839



MIT EECS 6.837

Questions?

MIT EECS 6.837

Implementation: Particle

- Particles p have attributes that specify mass $p.m$, position $p.x$, and velocity $p.v$.
- Other attributes such as color, particle age, and other can also be included for more advanced effects.
- For implementation convenience, each particle will also have a force accumulator $p.f$ that sums up all forces acting on the particle.

MIT EECS 6.837

Implementation: Particle Systems

- Particle system is an array of particles.
- Decouple system from solver:
 - We need to translate into a generic position vector and acceleration vector
 - 1. Set current state (positions and velocities)
 - 2. Get current state (positions and velocities)
 - 3. Compute accelerations $f(\mathbf{X}, t)$
- Integration uses only these methods to simulate evolution of a particle system

MIT EECS 6.837

Implementation: Accelerations

```
/* compute accelerations and place in f[] */
int ComputeAccelerations(ParticleSystem ps, double f[]) {
    int i;          /* particle counter */
    int j;          /* force dimension counter */
    ClearForces(ps); /* zero force accumulators */
    ComputeForces(ps); /* accumulate forces */
    for(i=0, j=0; i < ps.n; i++){
        f[j++] = ps.p[i].v[0]; /* xdot = v */
        f[j++] = ps.p[i].v[1];
        f[j++] = ps.p[i].v[2];
        f[j++] = ps.p[i].f[0]/m; /* vdot = f/m */
        f[j++] = ps.p[i].f[1]/m;
        f[j++] = ps.p[i].f[2]/m;
    }
}
```

MIT EECS 6.837

Implementation: Get State

```
/* gather state from the particles into X */
int GetState(ParticleSystem ps, double X[]) {
    int i, j;
    for(i=0, j=0; i < ps.n; i++){
        X[j++] = ps.p[i].x[0];
        X[j++] = ps.p[i].x[1];
        X[j++] = ps.p[i].x[2];
        X[j++] = ps.p[i].v[0];
        X[j++] = ps.p[i].v[1];
        X[j++] = ps.p[i].v[2];
    }
}
```

MIT EECS 6.837

Implementation: Set State

```
/* scatter state from X into particles */
int SetState(ParticleSystem ps, double X[]){
    int i, j;
    for(i=0, j=0; i < ps.n; i++){
        ps.p[i].x[0] = X[j++];
        ps.p[i].x[1] = X[j++];
        ps.p[i].x[2] = X[j++];
        ps.p[i].v[0] = X[j++];
        ps.p[i].v[1] = X[j++];
        ps.p[i].v[2] = X[j++];
    }
}
```

MIT EECS 6.837

Implementation: Euler Integration

```
void EulerStep(ParticleSystem ps, double dt){
    ComputeAccelerations(ps, f);
    GetState(ps, X0);
    X1 = X0 + dt * f; /* Euler step */
    SetState(ps, X1);
    ps.t += dt; /* advance time */
}
```

MIT EECS 6.837

Implementation: Trapezoid Integration

```
void EulerStep(ParticleSystem ps, double dt){
    ComputeAccelerations(ps, f0);
    GetState(ps, X0);
    X1 = X0 + dt * f0; /* Euler step */
    SetState(ps1, X1);
    ComputeAccelerations(ps1, f1);
    X'1 = X0 + dt * 0.5(f0+f1); /* trapezoid */
    SetState(ps, X'1);
    ps.t += dt; /* advance time */
}
```

Note the modularity enable by the general formulation of our problem. Could work for any dimensionality of the particle state, for any order of ODE as long as it's reduced to our 1st order formulation

MIT EECS 6.837

Questions?

MIT EECS 6.837

Where do particles come from?

- Often created by *generators* (or “emitters”)
 - can be attached to objects in the model
- Given rate of creation: particles/second
 - record t_{last} of last particle created

$$n = \lfloor (t - t_{last}) rate \rfloor$$

- create n particles.
 - update t_{last} if $n > 0$
- Create with (random) distribution of initial x and v
 - if creating $n > 1$ particles at once, spread out on path

MIT EECS 6.837

Particle lifetimes

- Record time of “birth” for each particle
- Specify lifetime
- Use particle age to:
 - remove particles from system when too old
 - often, change color
 - often, change transparency (old particles fade)
- Sometimes also remove particles that are offscreen

MIT EECS 6.837

Rendering and motion blur

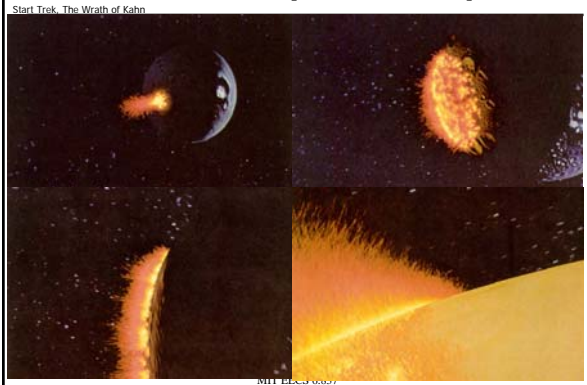
- Particles are usually not shaded (just emission)
- Often, they don’t contribute to the z-buffer (rendered last with z-buffer disabled)
 - That is, they don’t occlude one another
- Draw a line for motion blur
 - $(x, x+vd)$
- Sometimes use texture maps (fire, clouds)

MIT EECS 6.837

Questions?

MIT EECS 6.837

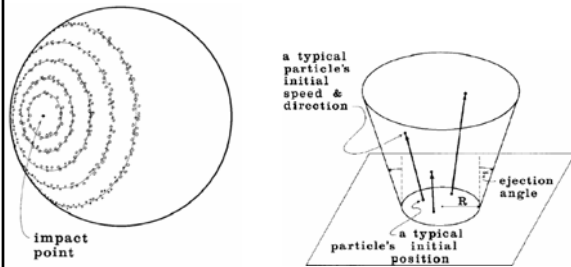
Particle Animation [Reeves et al. 1983]



MIT EECS 6.837

How they did it?

- One big particle system at impact
- Secondary systems for rings of fire.



Particle Modeling [Reeves et al. 1983]

- The grass is made of particles



MIT EECS 6.837

Other uses of particles

- Water
 - E.g. splashes in lord of the ring river scene
- Explosions in games
- Grass modeling
- Snow, rain
- Screen savers

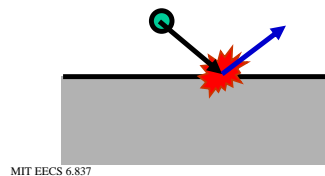
MIT EECS 6.837

Questions?

MIT EECS 6.837

Collisions

- Detection
- Response
- Overshooting problem (when we enter the solid)



MIT EECS 6.837

Detecting collisions

- Easy with implicit equations of surfaces
- $H(x,y,z)=0$ at surface
- $H(x,y,z)<0$ inside surface
- So just compute H and you know that you're inside if it's negative
- More complex with other surface definitions

MIT EECS 6.837

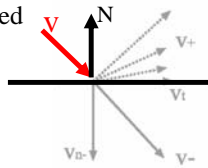
Collision response

- tangential velocity v_t unchanged
- normal velocity v_n reflects:

$$v = v_t + v_n$$

$$v \leftarrow v_t - \epsilon v_n$$

- coefficient of restitution
- change of velocity $= -(1+\epsilon)v$
- change of momentum *Impulse* $= -m(1+\epsilon)v$
- Remember mirror reflection?
Can be seen as photon particles



MIT EECS 6.837

Collisions - overshooting

- Usually, we detect collision when it's too late: we're already inside

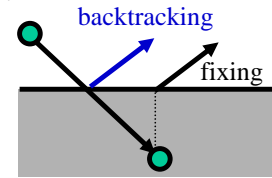
- Solutions: back up

- Compute intersection point
- Ray-object intersection!
- Compute response there
- Advance for remaining fractional time step

- Other solution:

Quick and dirty fixup

- Just project back to object closest point



MIT EECS 6.837

Questions?

MIT EECS 6.837

More advanced version

- Flocking birds, fish school
– <http://www.red3d.com/cwr/boids/>
- Crowds



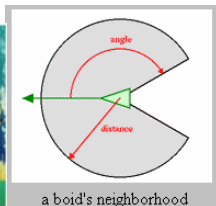
Battle of Helm's Deep, LOTR



MIT EECS 6.837

Flocks

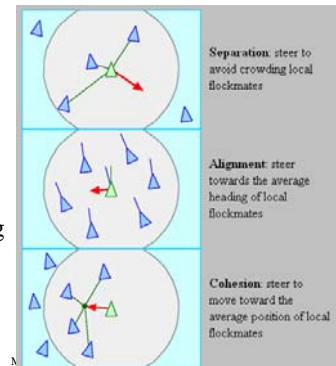
- From Craig Reynolds
- Each bird modeled as a complex particle ("boid")
- A set of forces control its behavior
- Based on location of other birds and control forces



a boid's neighborhood

Flocks

- From Craig Reynolds
- "Boid" was an abbreviation of "birdoid", as his rules applied equally to simulated flocking birds, and schooling fish.



Questions?

MIT EECS 6.837

Additional references

- <http://www.cse.ohio-state.edu/~parent/book/outline.html>
- <http://www.pixar.com/companyinfo/research/pbm2001/>
- <http://www.cs.unc.edu/~davemc/Particle/>

MIT EECS 6.837

Crowds in Cars – Marco da Silva



MIT EECS 6.837

Crowds in Cars

- Break down the crowd problem into two sub problems
 - Generating lots of animation
 - Rendering lots of cars

MIT EECS 6.837

Crowds in Cars

- Animation also broken into two problems
 - Generating the driving path of a car, also known as root animation
 - Generating root relative animation such as facial animation and rocking



MIT EECS 6.837

Crowds in Cars

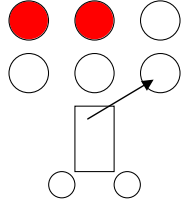
- Used a 3rd party system to model a driver's "brain"
 - Different heuristics for a freeway driver versus cars milling around outside of a stadium



MIT EECS 6.837

Crowds in Cars

- Milling cars tried to get to a goal point
- Steered to avoid oncoming traffic
- Tried to back up once in a while to clear some space



MIT EECS 6.837

Crowds in Cars



- Driving motion exported to in-house anim system
- Cars assigned event driven finite state machines
- FSMs cycled hand animated clips of cheering and other root relative actions

MIT EECS 6.837

Crowds in Cars

- When animation is nearly finished, it's time to start lighting and rendering
- 100,000+ cars in the final race scenes
- Crowd cars had special procedural shader



MIT EECS 6.837

Cars in Crowds

- Part of achieving look was to insure a proper distribution of fans
- McQueen fans had special colors and gear
- Variety was key



MIT EECS 6.837

Crowds in Cars

- Very simple geometry
- At most 25 polygons depending on accessories
- See Fredo's billboard clouds paper for an idea of how this was done



MIT EECS 6.837