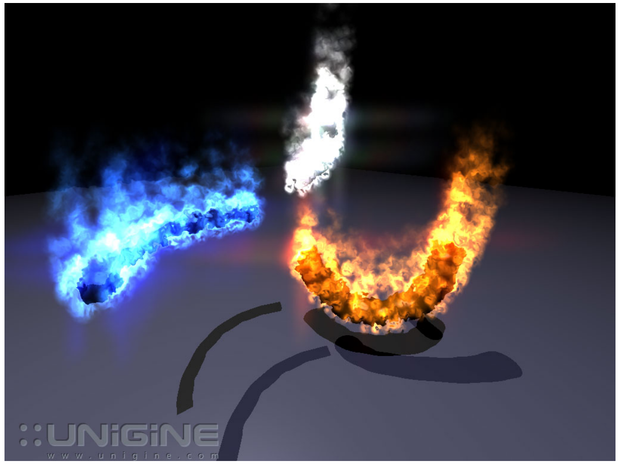
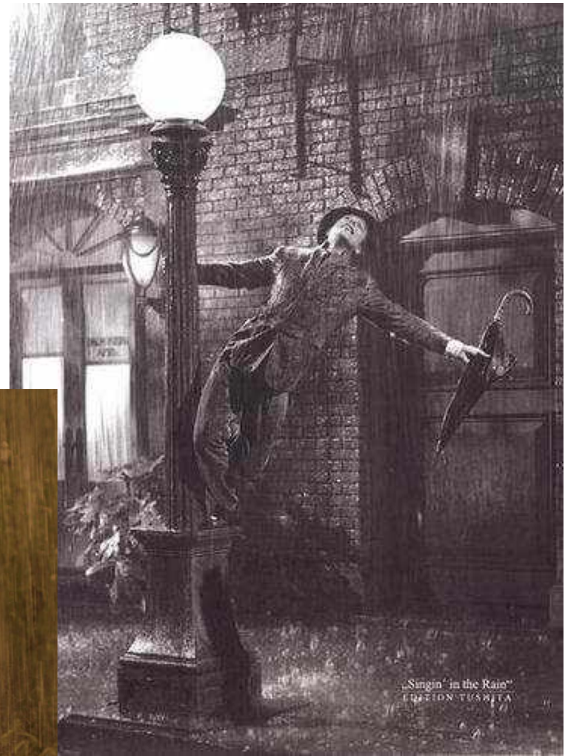
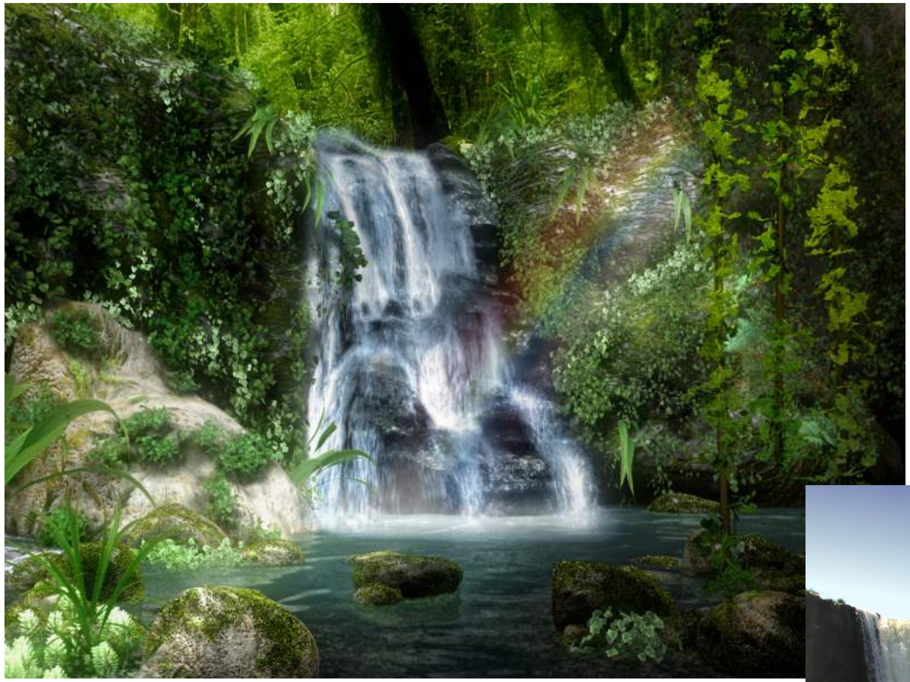
Virtual Reality & Physically-Based Simulation
Particle Systems

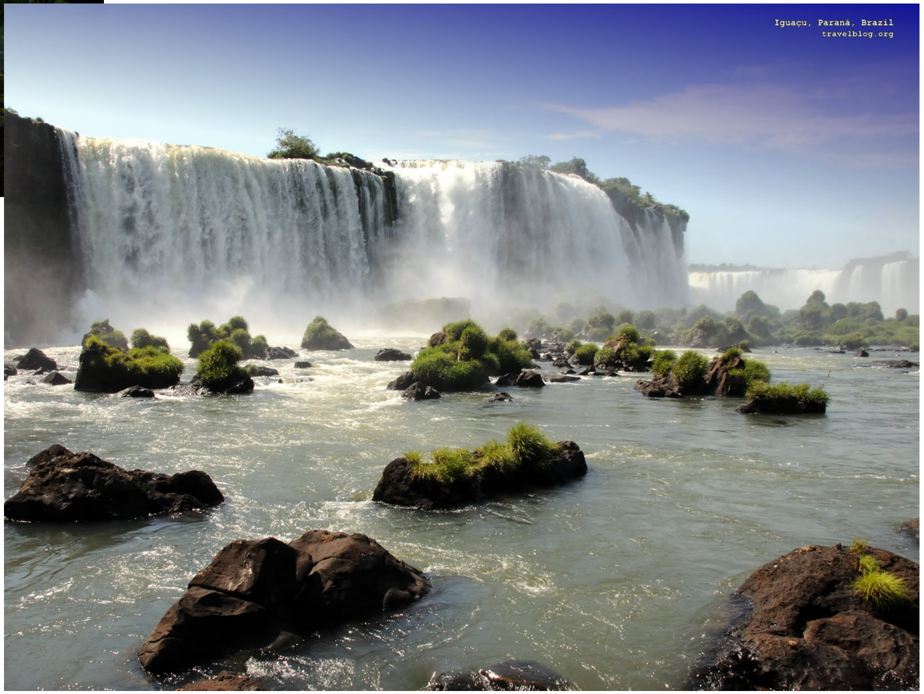


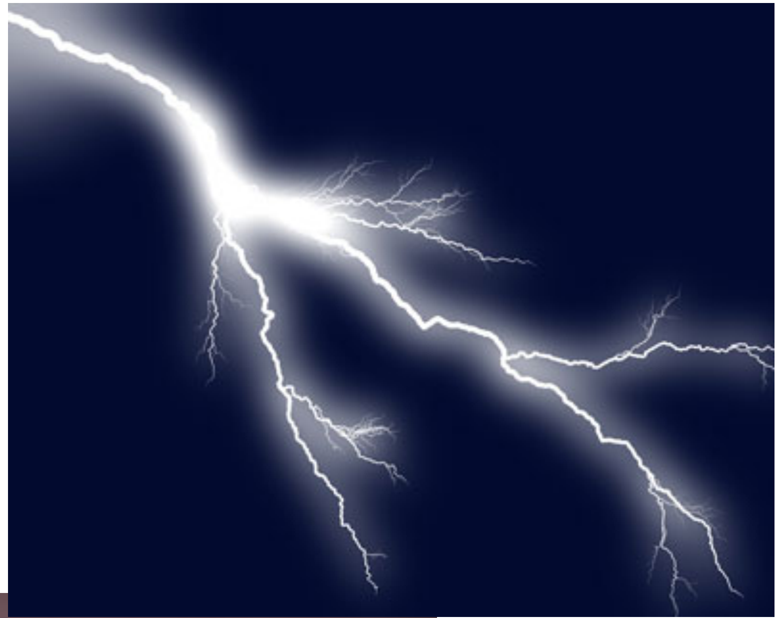
G. Zachmann
University of Bremen, Germany
cgvr.cs.uni-bremen.de

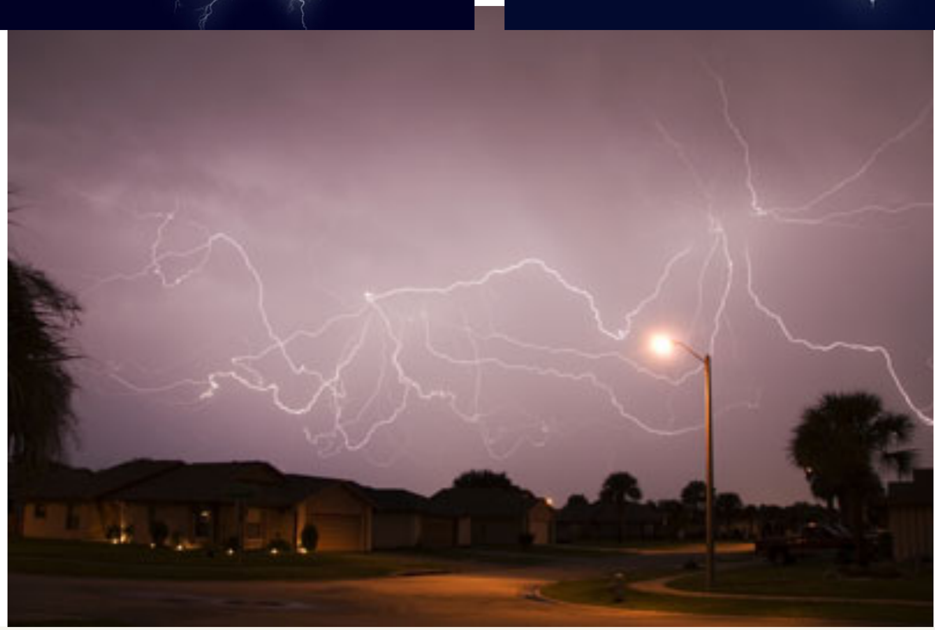
How to Model/Simulate/Render Natural Phenomena?






Iguaçu, Paraná, Brazil
travelblog.org





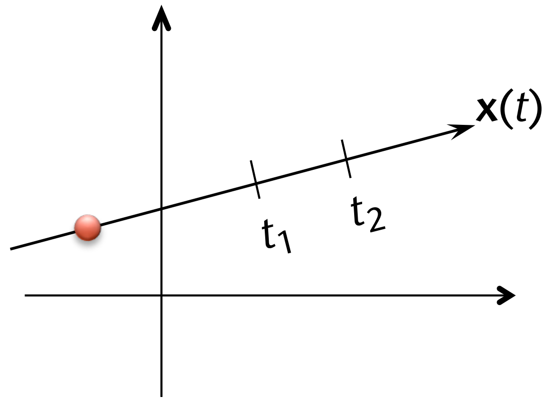
Dynamics of a Point Mass

- Definition Particle:

  A particle is ideal point with a mass m and a velocity $\mathbf{v}$.

  It does not process an orientation.

- Path of a particle = $\mathbf{x}(t)$



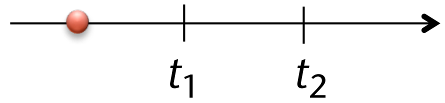
- Velocity:

$$\mathbf{v} = \frac{\text{distance}}{\text{time}} = \frac{\mathbf{x}(t_2) - \mathbf{x}(t_1)}{t_2 - t_1}$$

  - Unit: $^m/_s$
  - Note: velocity of particle = vector

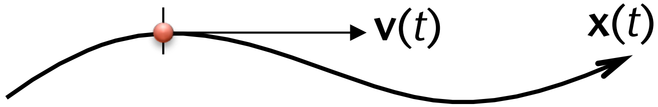
    position of particle = point!

- The momentary velocity:



$$\mathbf{v}(t_1) = \lim_{t_2 \to t_1} \frac{\mathbf{x}(t_2) - \mathbf{x}(t_1)}{t_2 - t_1}$$

$$= \frac{d}{dt}\mathbf{x}(t_1) = \dot{\mathbf{x}}(t_1)$$



- Examples:
  - Point moves on a circular path $\rightarrow \|\dot{\mathbf{x}}\|$ is constant
  - Point accelerates on a straight line $\rightarrow \dfrac{\dot{\mathbf{x}}}{\|\dot{\mathbf{x}}\|}$ is constant
- Acceleration at some point in time :

$$\mathbf{a}(t) = \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{v}(t) = \dot{\mathbf{v}}(t) = \frac{\mathbf{F}(t)}{m}$$

Newtons 2. Law

Euler Integration

- Given: a particle of mass m; a force $\mathbf{F}(t)$ that acts on the particle over time
- Wanted: the path $\mathbf{x}(t)$ of the particle
- The analytical approach:

$$\mathbf{v}(t) = \mathbf{v}_0 + \int_{t_0}^{t} \mathbf{a}(t)\,\mathrm{d}t$$

$$\mathbf{x}(t) = \mathbf{x}_0 + \int_{t_0}^{t} \mathbf{v}(t)\,\mathrm{d}t$$

- Discretization and linearization yields:

$$v^{t+1} = v^t + a^t \cdot \Delta t$$

$$x^{t+1} = x^t + v^t \cdot \Delta t$$

or

$$x^{t+1} = x^t + \frac{v^t + v^{t+1}}{2} \Delta t$$

(*approx. midpoint method*)

The Phase Space

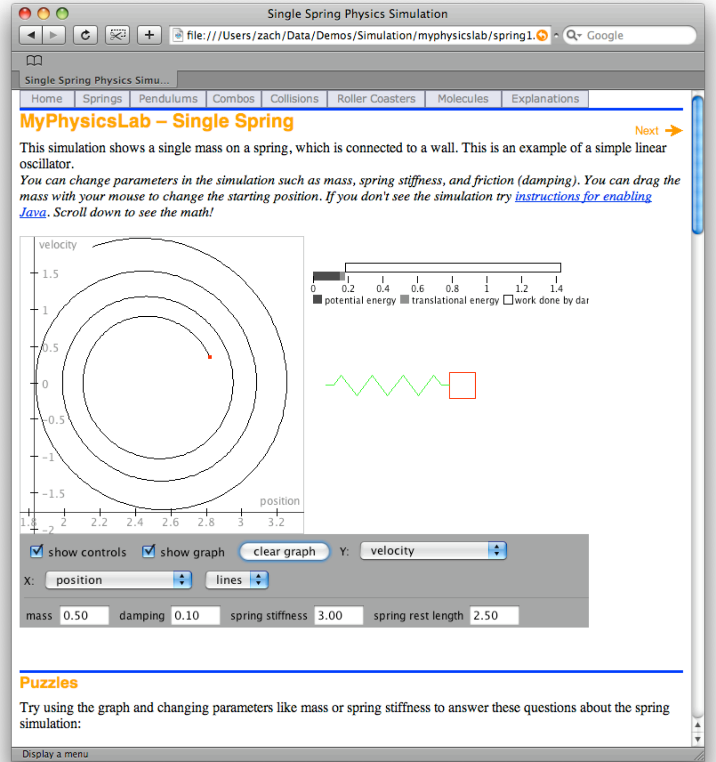
- The (physical) momentary state of a particle is described completely by

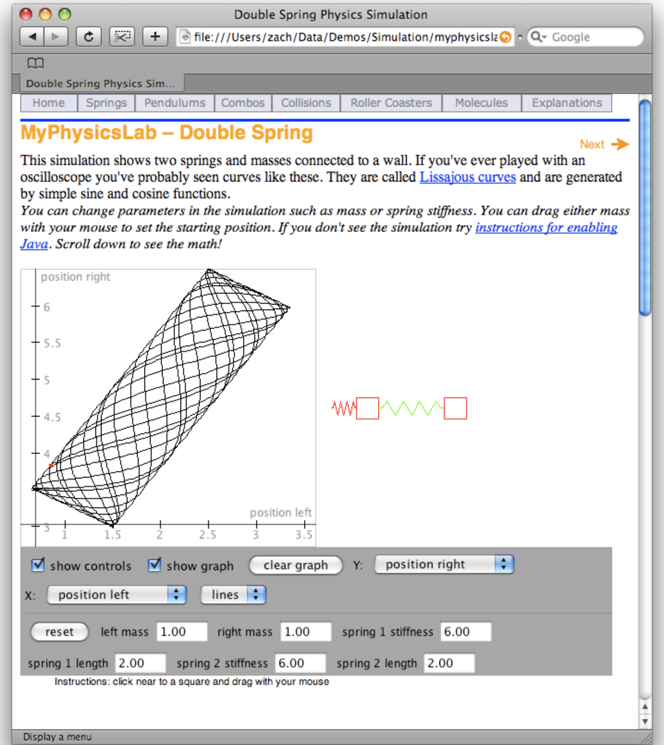
$$\begin{aligned}\mathbf{q} &= (\mathbf{x}, \mathbf{v}) = (x_1, x_2, x_3, v_1, v_2, v_3) \\ &= (x_1, x_2, x_3, \dot{x}_1, \dot{x}_2, \dot{x}_3) \in \mathbb{R}^6\end{aligned}$$

- The space of all possible states is called *phase space*
- The dimension is $6n$, n = number of particles
- The motion of a particular in phase space:

$$\dot{\mathbf{q}} = (\dot{x}_1, \dot{x}_2, \dot{x}_3, \dot{v}_1, \dot{v}_2, \dot{v}_3) = \left(v_1, v_2, v_3, \frac{\mathbf{f}_1}{m}, \frac{\mathbf{f}_2}{m}, \frac{\mathbf{f}_3}{m} \right)$$

- Example for a particle that can move only along the X axis and that is held in a resting position by a spring:





www.myphysicslab.com

Kinematics vs Dynamics

- Technical terms:

  kinematics = motion of bodies without simulation of forces

  dynamics = simulation/computation of forces and the motions of the objects resulting from them

- In computer graphics we always move within a continuum:

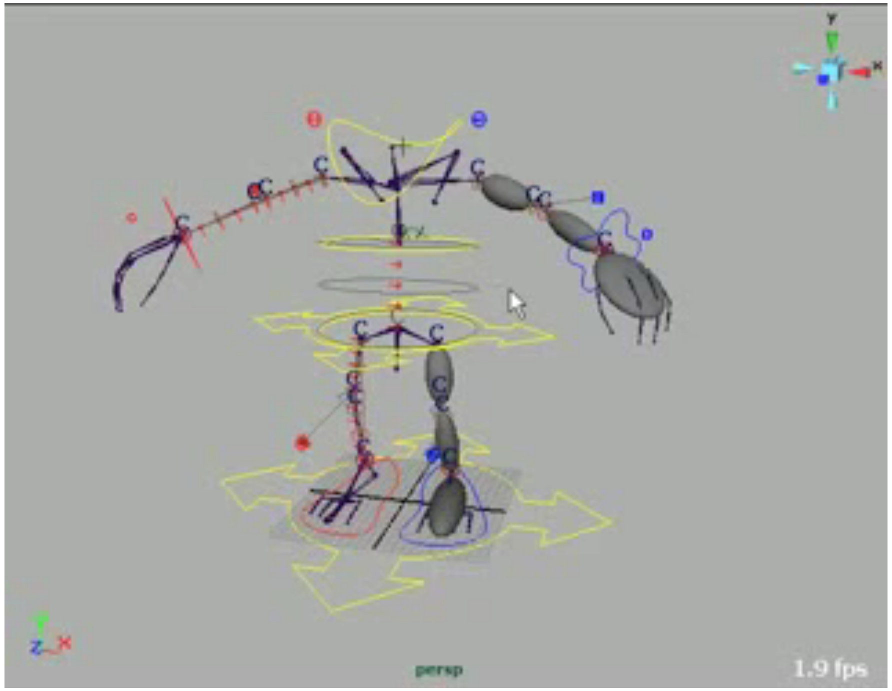
kinematics ⟷ dynamics

Direct control over place and velocity of individual objects

All objects of a physical system are represented and initialized completely; thereafter all forces are computed exactly and the whole physical system is being simulated (" with integration schemes as good as possible")

Particle Systems

- Example of pure kinematics: inverse kinematics



Particle Systems

- Definition: a particle system is comprised of
  1. A set of particles; each particle i has, at least, the following attributes:
     - Mass, position, velocity (m_i, $\mathbf{x}_i$, $\mathbf{v}_i$)
     - Age a_i
     - Force accumulator $\mathbf{F}_i$
     - Optional: color, transparency, optical size, lifespan, type, …
  2. A set of particle sources; each one is described by
     - Form of the particle source
     - Stochastic processes that determine the initial attributes of the particles, e.g., velocity, direction, etc.
     - Stochastic processes that determine the number of particles created per frame
  3. Other (global) parameters, e.g.
     - TTL (time to live) = max. lifespan of particles
     - Global forces, e.g. gravitation, wind, …
     - The Algorithms, that move and renderer of particles

- Stochastic process =
  - Simplest case: average + variance; process outputs random value according to uniform distribution
  - A bit more complicated: average and variance functions over time

- Remarked on the form of a particle source:
  - Just an intuitive way to describe the stochastic process for the initial position of particles
  - Frequent forms: disk, cube, cone, etc.

The Execution Model

- The "main loop" of a particle system:

```
loop forever:
  render all particles
  Δt := rendering time
  kill all particles with age > TTL (max. life-span)
  create new particles at particle source
  reset all force accumulators
  compute all forces on each particle (accumulate them)
  compute new velocities (one Euler step with Δt)
  optionally modify velocities (*)
  compute new positions (another Euler step)
  optionally modify positions (e.g. b/c of constraints)
  sort all particles by depth (for alpha rendering)
```

Remarks

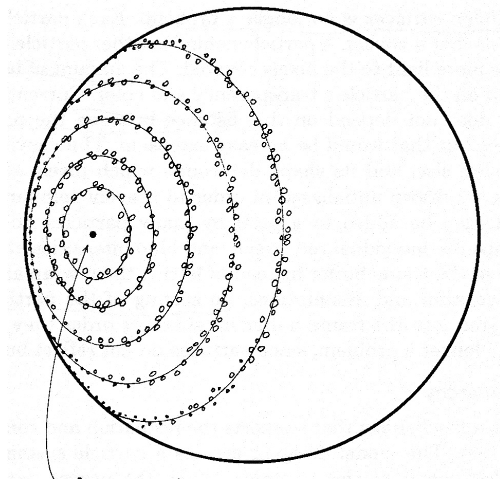
- There is lots of space for optimizations, e.g.
  - Initialize force accumulators with gravitational force
  - don't increment the age of each particle "by hand"; instead, save the time of their creation in t_{gen} , then just test $t_{current} - t_{gen} > TTL$
    - Will be important for parallel implementation later
- On (∗) in the algorithm:
  - This is "non-physical", but allows for better kinematic control by the programmer/animator
  - This is also necessary in case of collisions
- Often, we store a small history of the positions of particles, in order to create simple "motion blur" effect
- Particulates can be killed by other constraints, too, e.g. distance from the source, entrance into a specific region, etc.
- For an efficient implementation, a "struct-of-array" data structure can be better! (SoA instead of AoS)

Example of a particle system

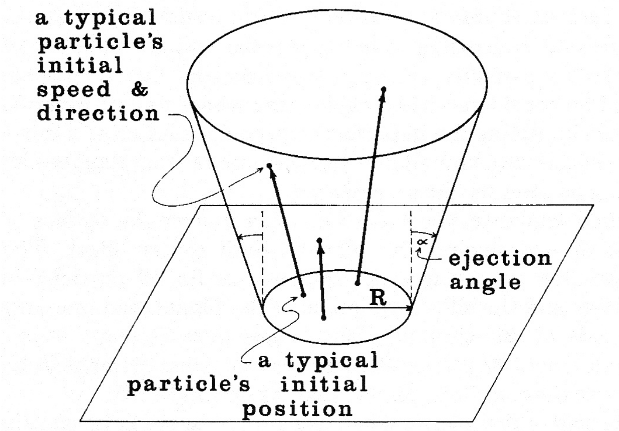
- Excerpt of "Wrath of Khan":

(Loren Carpenter, William Reeves, Alvy Ray Smith, et al., 1982)

- Particle source = circles on a sphere around the *point of impact*, which increase over time

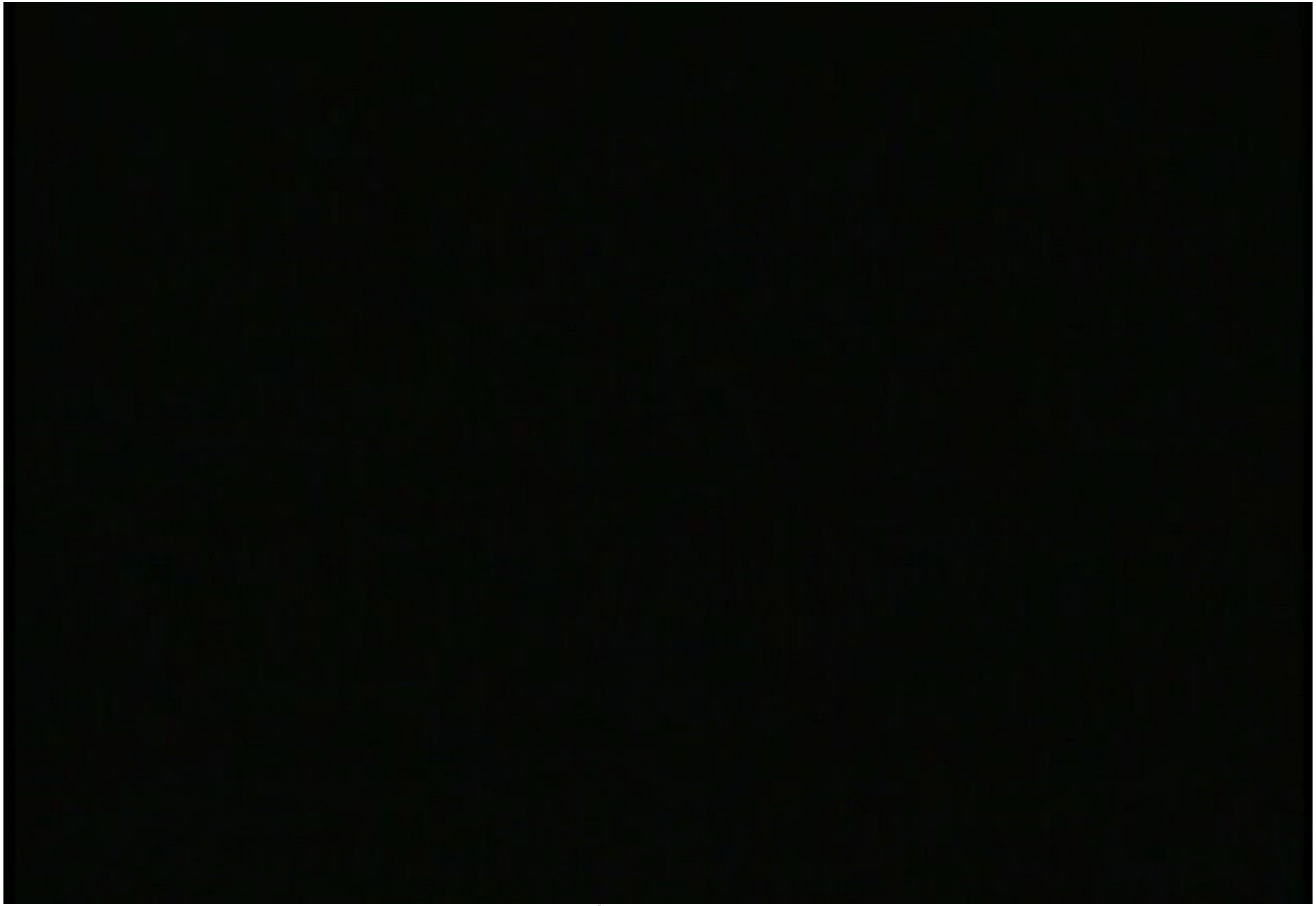


- Stochastic processes for particle creation:
  - Capped cone normal to surface of sphere
  - Some variance of each particles lifespan



- Color = $f(age)$

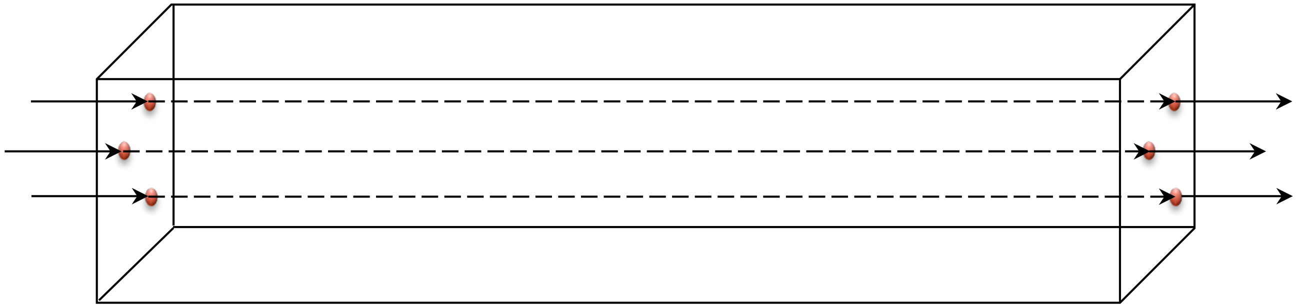
Digression: the Panspermia Hypothesis



Karl Sims, 1990

Operations on Particles

- Position operations:
  - Rather rare, e.g. “tunneling”

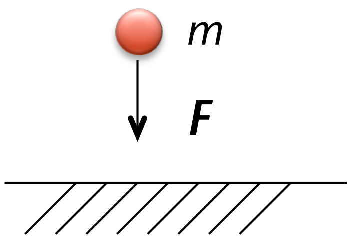


  - Mostly for done collision handling

Physical Effects

- Gravity:

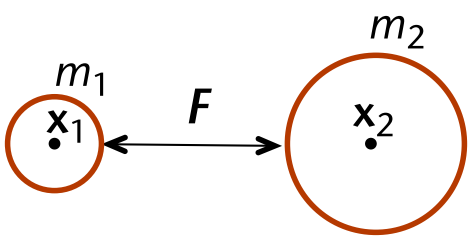
$$\mathbf{F} = m \cdot \mathbf{g} \quad , \qquad g = 9.81 \frac{\mathrm{m}}{\mathrm{s}^2}$$



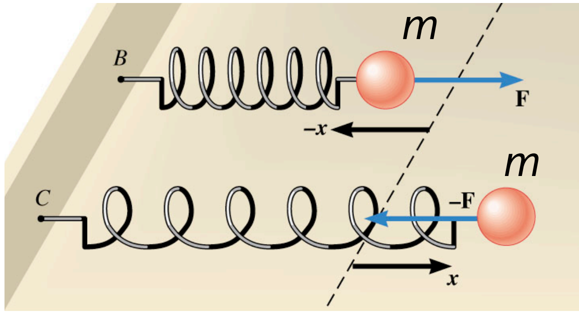
- Gravitation:

$$\mathbf{F} = G \cdot \frac{m_1 m_2}{r^2} \cdot \frac{\mathbf{x}_1 - \mathbf{x}_2}{|\mathbf{x}_1 - \mathbf{x}_2|}$$

$$G = 6,67 \cdot 10^{-11}$$



- Spring force: later



- Viscous drag (viskose Hemmung/Dämpfung):

$$\mathbf{F} = -b\ \mathbf{v}$$

in a stationary fluid/gas;

or, sometimes,

$$\mathbf{F} = 6\pi\eta r(\mathbf{v} - \mathbf{v}_{fl})$$

in fluids with velocity $\mathbf{v}_{fl}$, particles with radius r, viscosity η;

or, sometimes,

$$\mathbf{F} = -\frac{1}{2} c\rho A\mathbf{v}^2$$

with high velocities; ρ = density, A = size of cross-sectional area,

c = viscosity constant

- Electromagnetic force (Lorentz force):

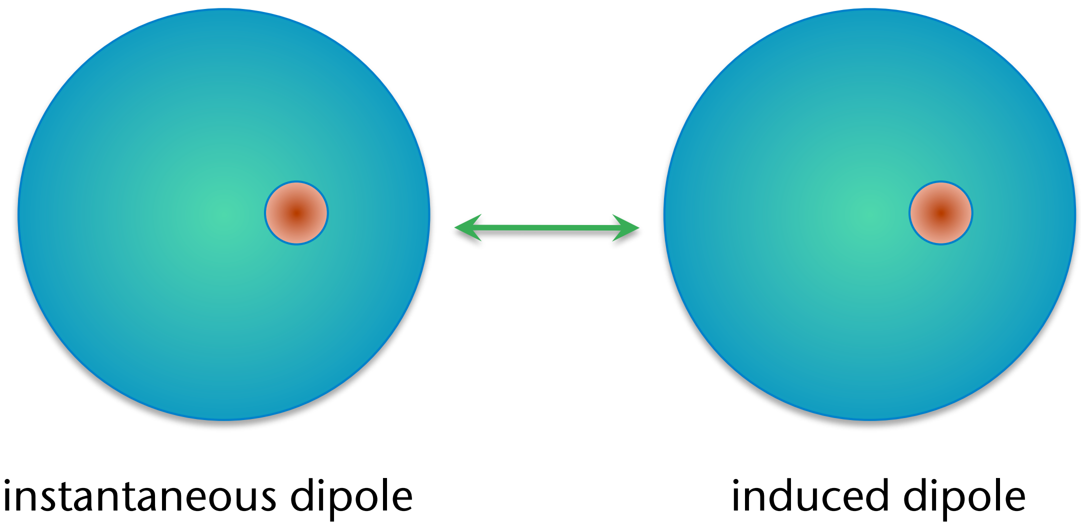
$$\mathbf{F} = q \cdot \mathbf{v} \times \mathbf{B}$$

where q = charge of particle , $\mathbf{v}$ = velocity of particle,
$\mathbf{B}$ = magnetic field

https://elearning.mat.univie.ac.at/physikwiki/index.php/LV002:LV-Uebersicht/Videos/Lorentzkraft_1

The Lennard-Jones Force

- There are two kinds of forces between atoms:
  - A repelling force (abstoßend) on short distances
  - An attracting force on longer distances (called van-der-Waals force or dispersion force)



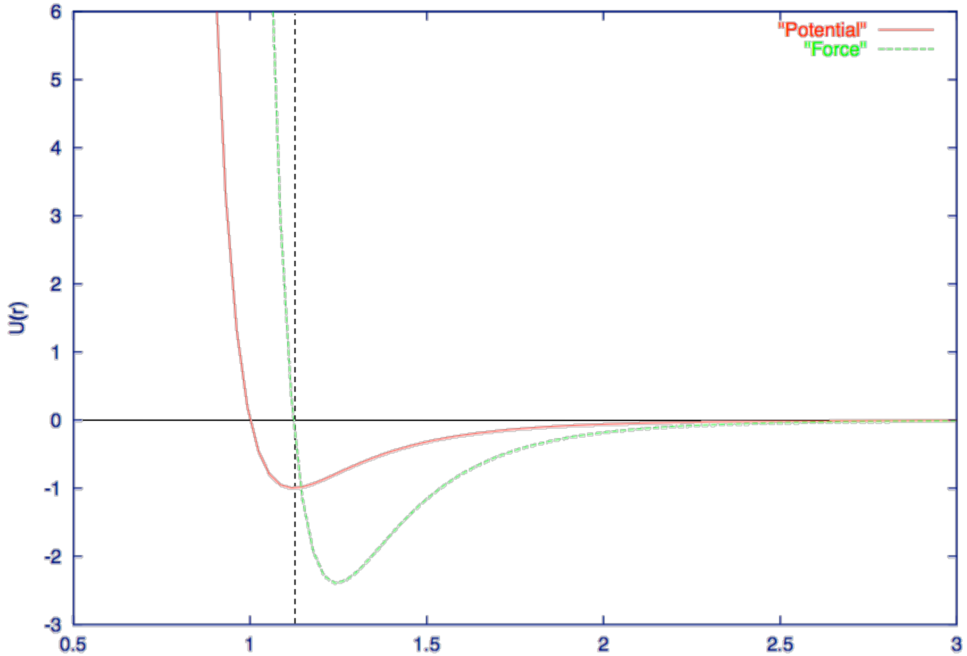
- One (arbitrary) approximation of the Lennard-Jones force:

$$\mathbf{F} = \varepsilon \cdot \left(c \left(\frac{\sigma}{d} \right)^m - \left(\frac{\sigma}{d} \right)^n \right) \cdot \frac{\mathbf{x}_1 - \mathbf{x}_2}{\|\mathbf{x}_1 - \mathbf{x}_2\|}$$

where

$$d = \|\mathbf{x}_1 - \mathbf{x}_2\|$$

and ε , c, m, n are arbitrary constants (for our purposes)



Non-Physical Effects

- Vortex: rotate particle about axis **R** with angle

$$\theta = a \cdot f(r)$$

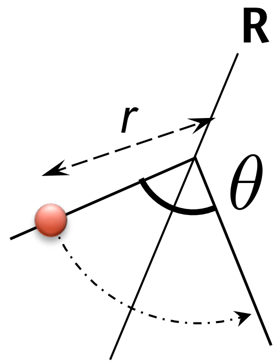
where a = "force" of the vortex,

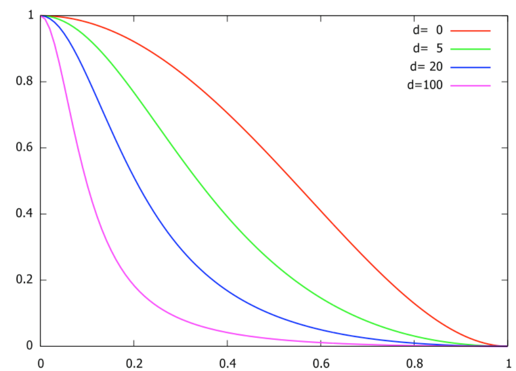
r = distance particle – axis, and

$$f(r) = \frac{1}{r^{\alpha}}$$

or

$$f(r) = \begin{cases} \frac{r^4 - 2r^2 + 1}{1 + dr^2} & , \; r \leq 1 \\ 0 & , \; r > 1 \end{cases}$$

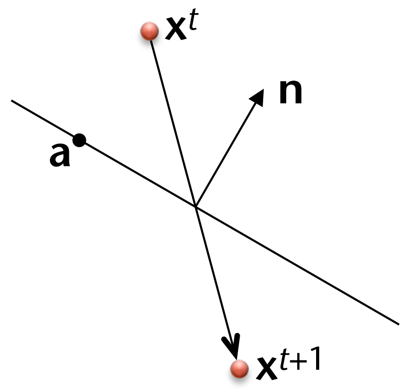




- Extensions:
  - Take mass of particle into account
  - Use B-spline as axis of the vortex (e.g., for tornado)
  - Animate the axis of the vortex

Collisions

- Most important kind of geometric constraint
- First of all: collision with a plane



- Collision check:

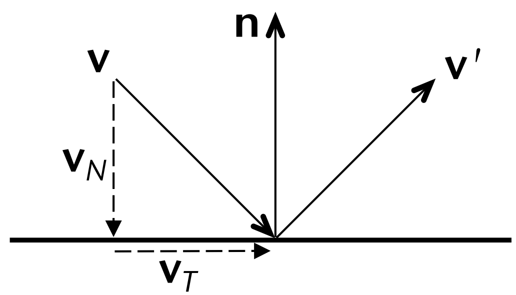
$$(\mathbf{x}^t - a)\,\mathbf{n} > 0 \;\wedge\; \left(\mathbf{x}^{t+1} - \mathbf{a}\right)\mathbf{n} < 0$$

- Collision handling: reflect $\mathbf{v}$

$$\mathbf{v}_N = (\mathbf{v}\cdot\mathbf{n})\,\mathbf{n}$$

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

$$\mathbf{v}' = \mathbf{v}_T - \mathbf{v}_N = \mathbf{v} - 2(\mathbf{v}\cdot\mathbf{n})\,\mathbf{n}$$

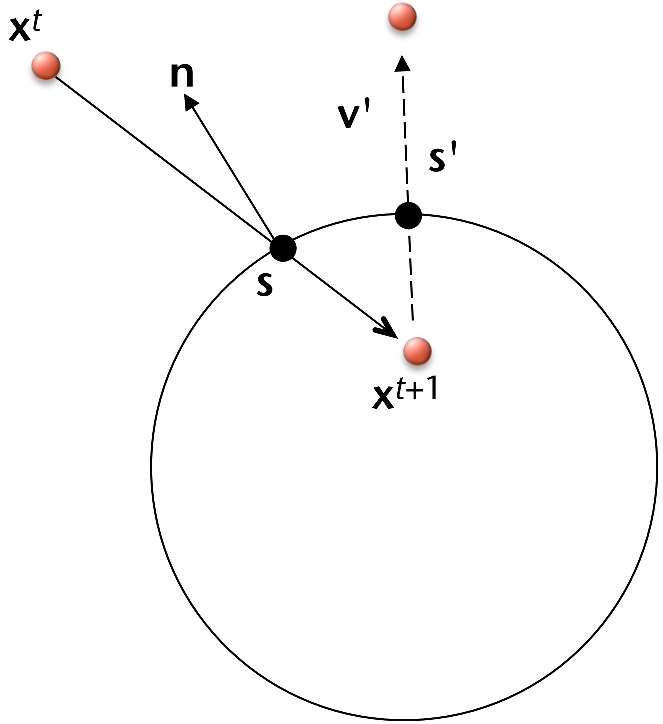


- Extensions to friction and elastic/inelastic impact:

$$\mathbf{v}' = (1-\mu)\,\mathbf{v}_T - \varepsilon\mathbf{v}_N$$

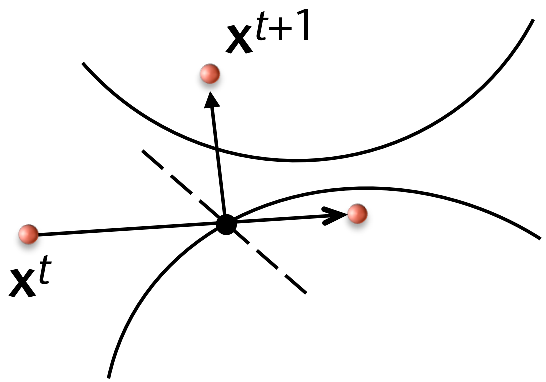
with μ = friction parameter and
ε = resilience (Federung / Elastizität)

- Collision with a sphere:
  - Compute exact intersection of $\mathbf{x}^t\mathbf{x}^{t+1}$ with $\mathbf{s}$
  - Determine normal $\mathbf{n}$ at point $\mathbf{s}$
  - Then continue as before
- Conclusion:
  collision detection for particles = "point inside geoetry test", or more precisely: intersection tests between line segment and geometry

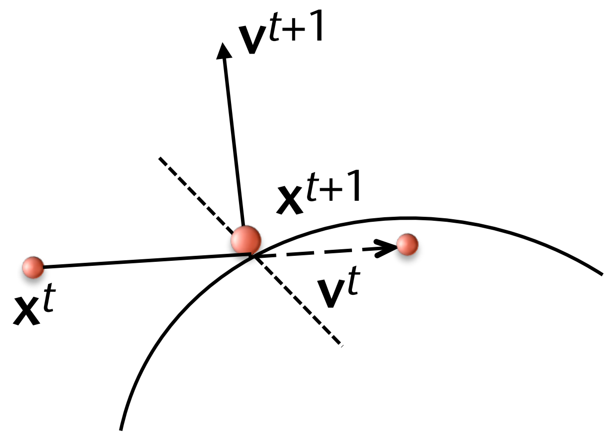


- For polyhedra and terrain: see "Computer Graphics 1"
- For implicit surfaces: see "Advanced Computer Graphics"

- Challenge: always create a consistent system after the collision handling!
  - Problem: "double collisions" at narrow places
  - Example:



  - Correct handling:



  - There are more ways to handle these kinds of situations …

Karl Sims: *Particle Dreams*

Bremen

Hierarchical Particle Systems

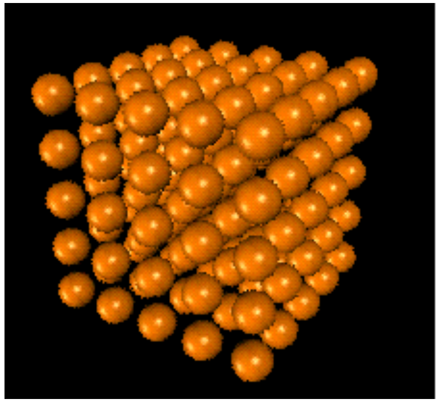
- Idea:
  - A particlel represents a whole (lower level) particle system
  - Transformation of the parent particle moves the local coord frame of its ancillary particle system (just like with scenen graphs)
- Second-order particle systems:
  - All forces are being represented by particles
  - Forces can, thus, interact with each other, they can die, get born, etc.

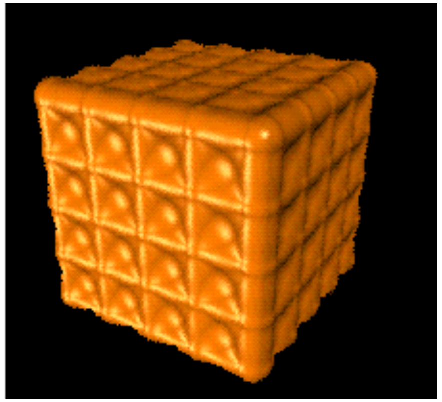
Rendering

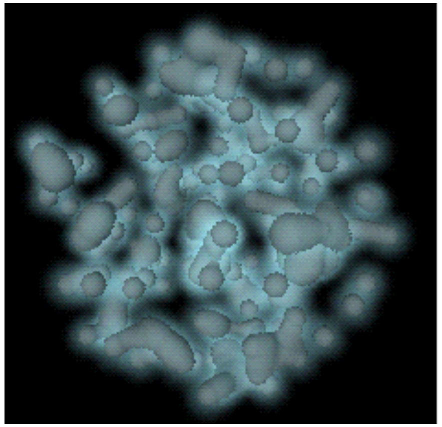
- Tehre is no standard method for that
- Common method:
  - Render small discs for particles (splat, sprite, billboard)
  - Often with transparency that decreases toward the rim
  - Needs alpha-blending!
- Alternative:
  - Accumulate the color of all particles in frame buffer (e.g., fire)
  - Needs about 10 Particlels/pixel to look good

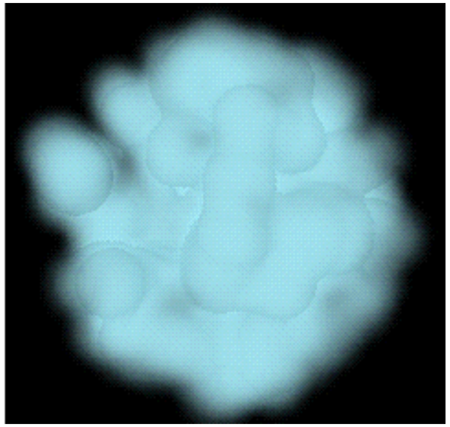
Rendering of "Blobby Objects"

- Regard particle as metaballs
  - In CG 2: Metaballs = spheres that blend together to form (implicit) surfaces
  - Render using ray-casting
  - Either: find root of implicit surface
  - Or: accumulate the "density" along ray and interpret this as opacity or as luminance

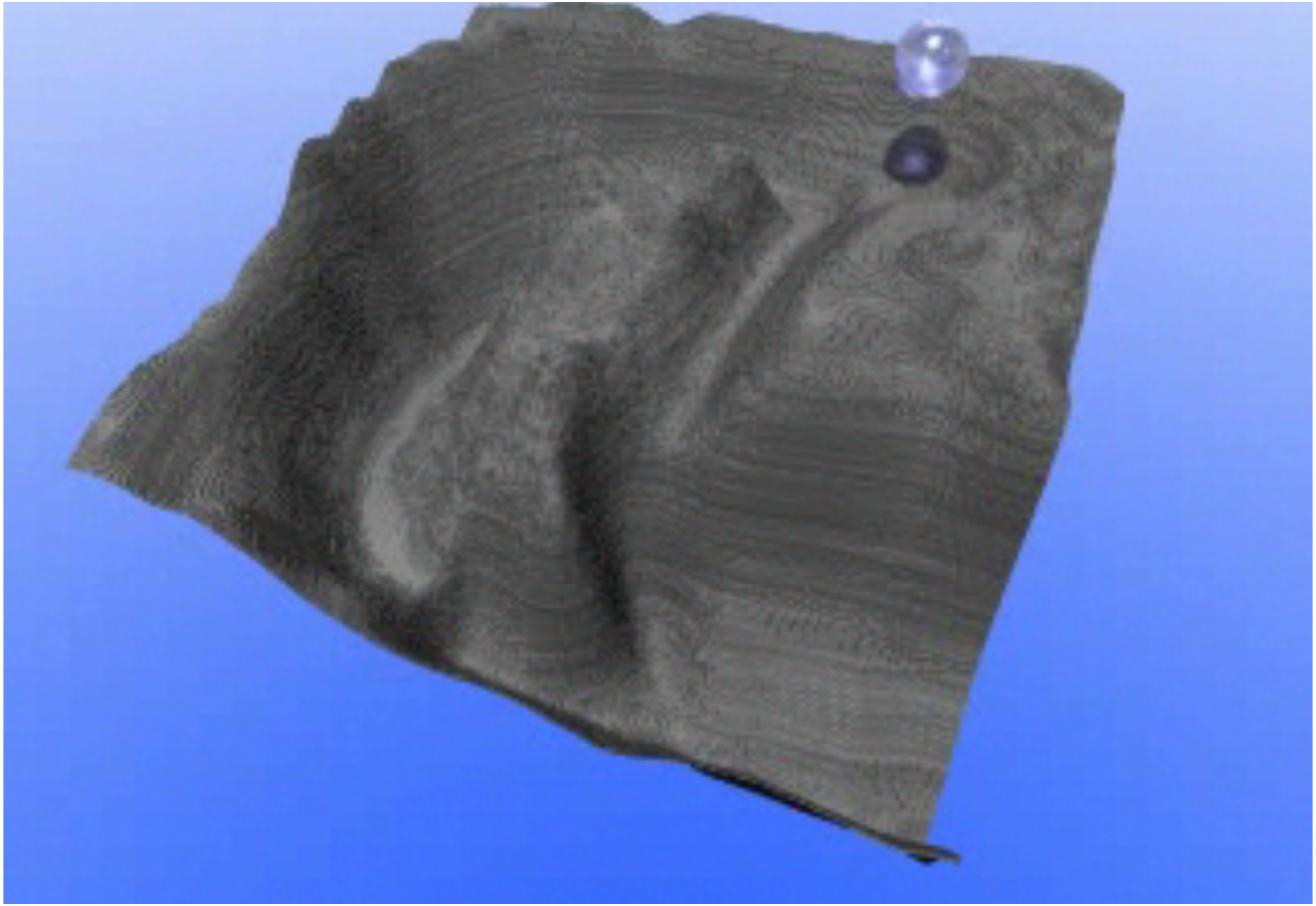






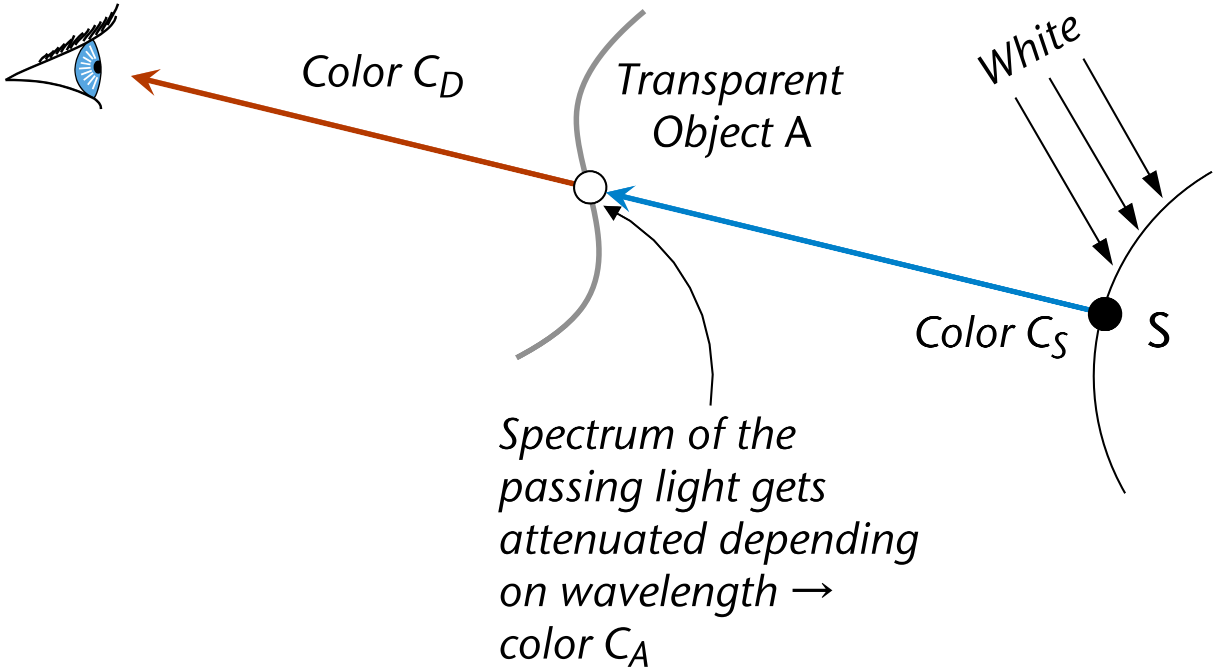


Example



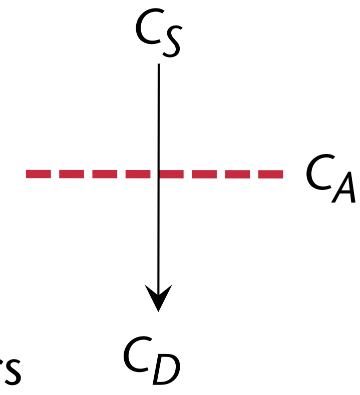
Rendering of Transparent Objects

- Transparency ≈ material that lets light pass partially
- Often, some wavelengths are attenuated more than others → colored transparency
  - Extreme case: color filter (photography)



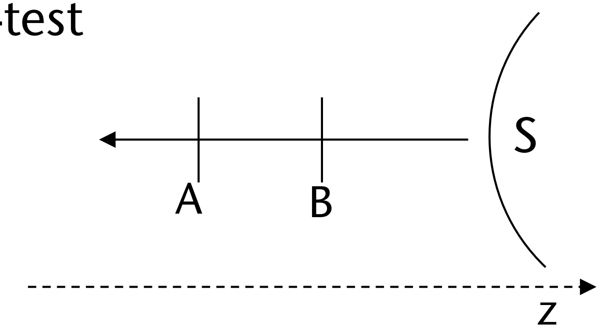
- Approximation: Alpha Blending
  - $\alpha \in [0, 1]$ = opacity (= opposite of transparency)
    - $\alpha = 0 \rightarrow$ completely transparent,
      $\alpha = 1 \rightarrow$ completely opaque
  - "Color" C_A of object A = transmission spectrum
    (similar to reflectance spectrum of opaque objects, see CG1)
  - Outgoing color:

$$C_D = \alpha C_A + (1 - \alpha) C_S$$



  - Practical implementation: α = 4th component in color vectors
- During rendering, the graphics card performs these operations:
  1. Read color from frame buffer $\rightarrow C_S$
  2. Compute C_D by above equation
  3. Write C_D in framebuffer

- Problem: several transparent objects behind each other!
  - Solution: first A, then B $\rightarrow$ B gets killed by Z-test
- Naïve idea: switch Z-buffer off
  - First A then B (w/o Z-test) results in:



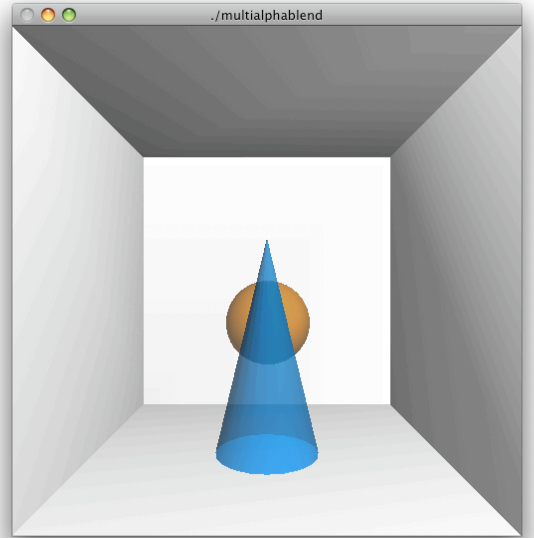
$$
\begin{aligned}
C'_D &= \alpha_A C_A + (1-\alpha_A)\, C_S \\
C_D &= \alpha_B C_B + (1-\alpha_B)\, C'_D \\
&= \alpha_B C_B + (1-\alpha_B)\,\alpha_A C_A + (1-\alpha_B)\,(1-\alpha_A)\, C_S
\end{aligned}
$$

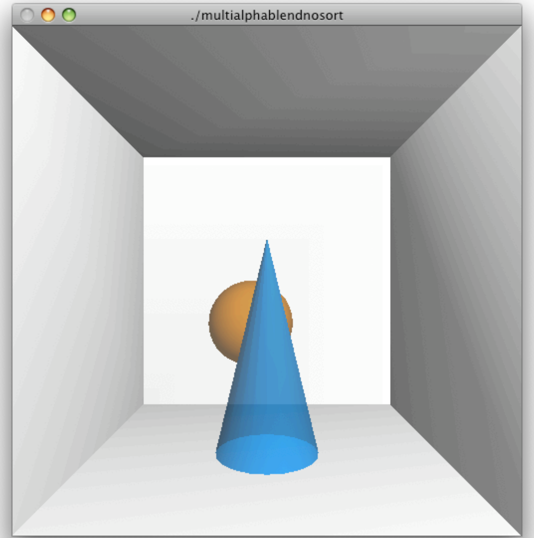
  - First B then A (w/o Z-test) results in:

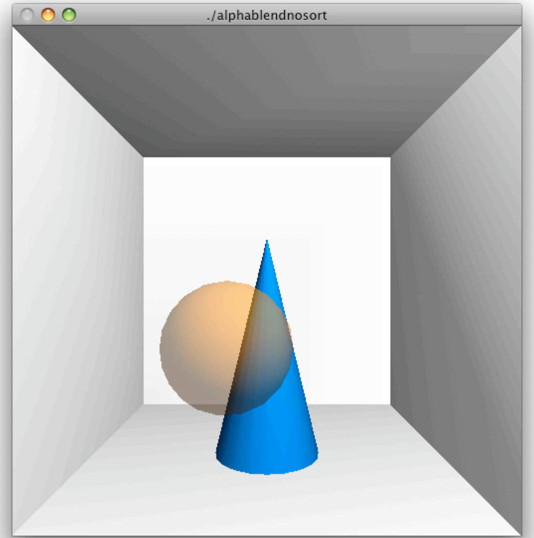
$$
\begin{aligned}
C'_D &= \alpha_B C_B + (1-\alpha_B)\, C_S \\
C_D &= (1-\alpha_A)\,\alpha_B C_B + \alpha_A C_A + (1-\alpha_B)\,(1-\alpha_A)\, C_S
\end{aligned}
$$

- Conclusion: you must render transparent polygons/particles from back to front, even if the Z-buffer is switched off!

- Examples (1 is correct, 2 with artifacts):







- In Open GL:
  - Switch blending on:

```
glEnable( GL_BLEND );
```

  - Determine blending function:

```
glBlendFrame( Glenum s, Glenum d );
```

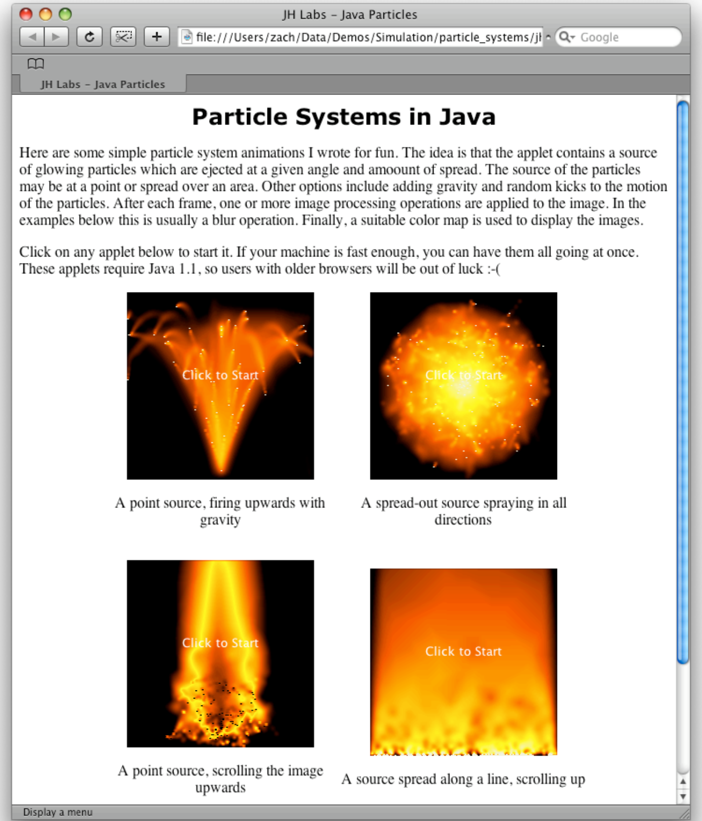
GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA $\rightarrow$

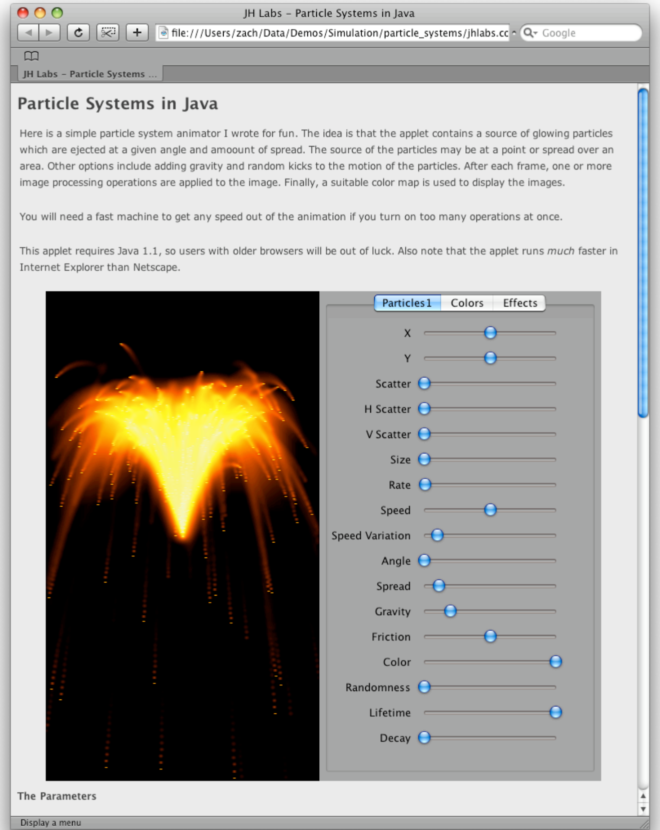
$$C'_D = \alpha C_A + (1 - \alpha)\, C_B$$

where C_D = color from frame buffer;

  - There are many more variants, e.g., you can just accumulate colors (**GL_ONE, GL_ONE**)

Particle System Demos





http://www.jhlabs.com/java/particles.html

Flames & Fire [2002]

- Goals:
  1. Flames that look convincing
  2. Complete control over the flames
- The model:
  1. Represent individual flame (elements) by parametric curves → "spine" of a flame
  2. Regard the control points of the spine as particles
  3. Create surface around the spinewhere the burning happens
  4. Sample space in the proximity of the surface by "five" particles
  5. Render these particles (either volumetrically, or with alpha-blending)
- Controls for animators:
  - Length of spines (average & variance)
  - Lifespan of spine particles
  - Intensity of fire (=number of fire particles; particle sources, wind, etc
  - Color and size of fire particles

- Generation of the *spines*:
  - Create a spine particlel P in first frame
  - Simulate P: let it move upwards (buoyancy) and sideways (wind)

$$\mathbf{v}_P^{t+1} = \mathbf{v}_P^t + w\left(\mathbf{x}_P, t\right) + b(T_P) + d(T_P)$$

    where

    w = wind field

    b = buoyancy

    d = diffusion = noise;

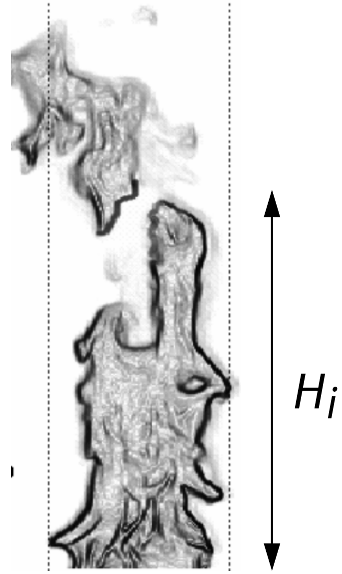
    T_P = temperature of particlel = age

    (Simplification here: particles don't have a mass)
  - In subsequent frames: create more particles; until max. number per flame is reached
  - Connect all spine particles by B-spline

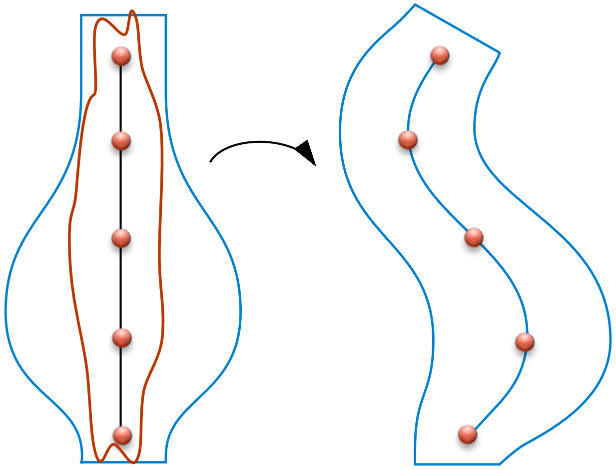
- At top of flames: break flame apart
  - Top part of spine is separated fom rest at a random point in time, if height > H_i
  - Lifespan after the split:

$$A \cdot \alpha^3 \;, \quad \alpha \in [0,1] \text{ zufällig}$$

$$A = 0.1 \ldots 2 \text{ sec}$$



- The profile of a flame:
  - Rotationally symmetric around spine (generalized cylinder)

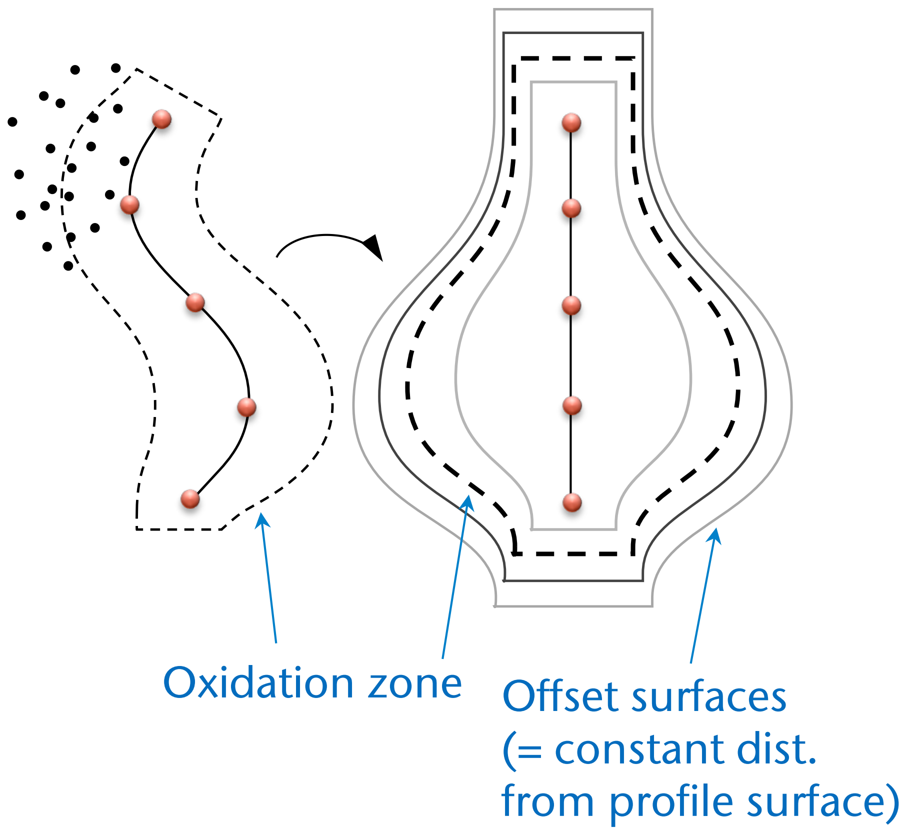


- Rendering:
  - Sample space around flame by a large number of "fire" particles according to this density function

$$D(\mathbf{x}) = \frac{1}{1 + ||\mathbf{x} - \mathbf{x}'||}$$

where **x**' = point on the (deformed) profilesurface which is closest to **x** :

- Create random **x**
- Transform into model space
- Compute **x**'
- Evaluate *D*
- If $D(\mathbf{x})$ > random number → keep fire particle **x**
- Put reference photo of real flame on profile surface → basis color for **x**

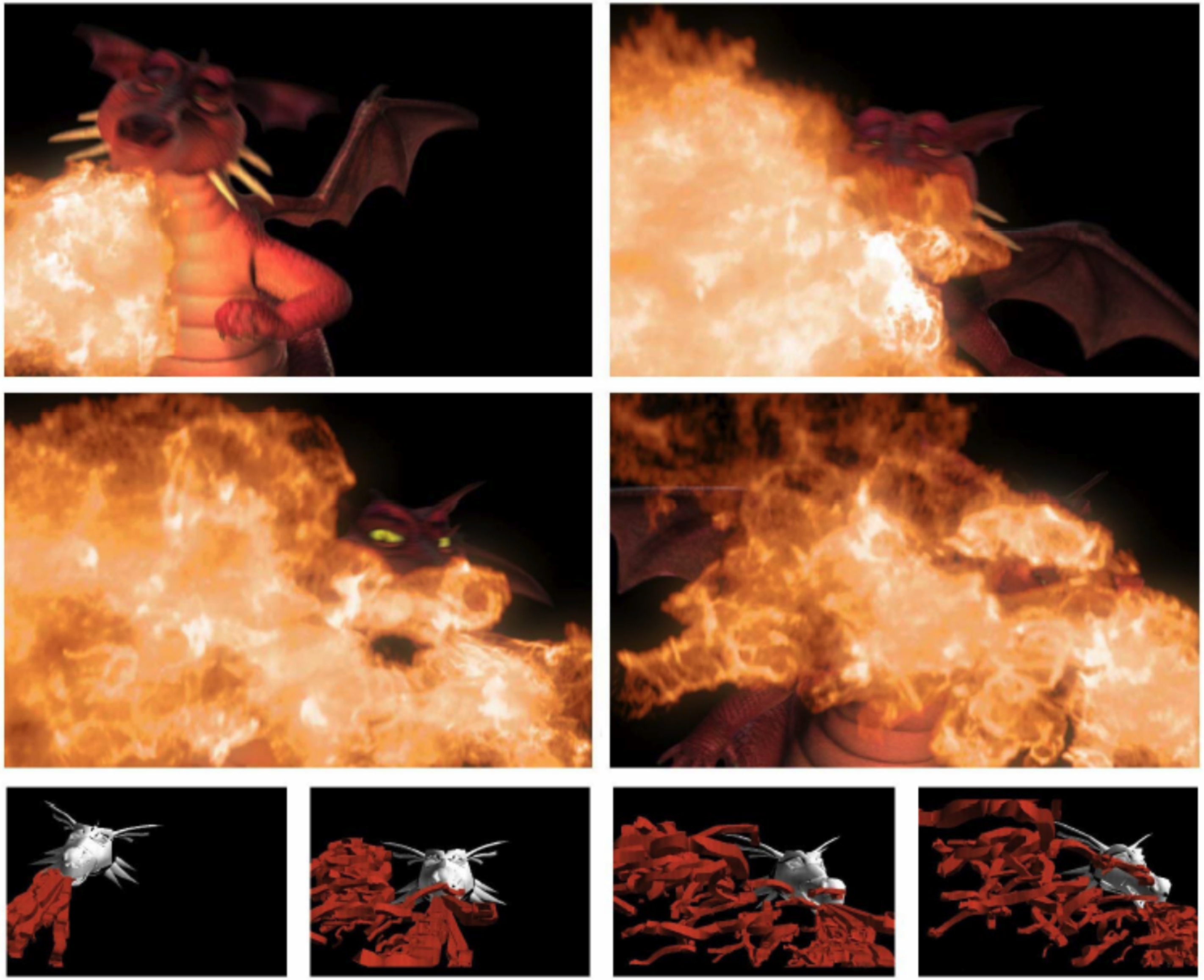


- Brightness of a fire particle at position **x**:

$$E(\mathbf{x}) = k\frac{D(\mathbf{x})}{n}$$

  where k = faktor for animator's control, n = number of samples
- Rule fo thumb: ca. 10 samples per pixel, ca 10,000 samples per flame
- Discard samples on the inside of obstacles
- Smoke: render fire particles with height > "smoke height" in grey/black

Results



Arnauld Lamorlette and Nick Foster, PDI/DreamWorks

Procedural Modeling of Plants with Particles

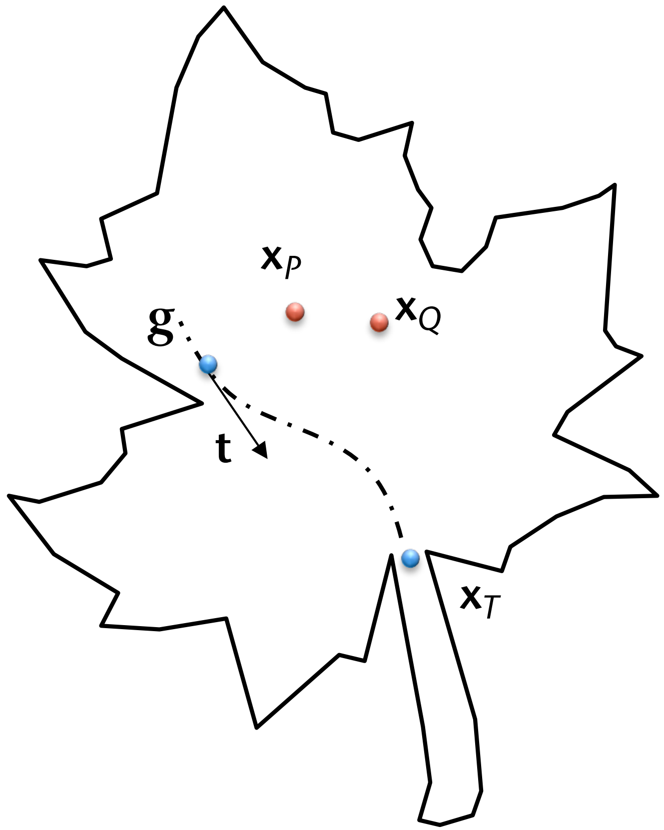
- Idea: use particles to simulate the transportation of water inside a leaf
  - Paths of particles constitute the vessels/"arteries"in the leaf
- Axioms:
  1. Nature always tries to minimize the total length of all arteries → particles will try to merge
  2. No water gets lost or gets added within the arteries →
     if 2 particles merge their paths, the resulting artery must have twice the cross-sectional area
  3. All arteries/paths emanate from the stem of the leaf

- Overview of the algorithm:

```
Initialize particles randomly on surface/rim of the leaf
loop until no particle is left:
    move each particle closer towards its nearest neighbor
        or towards an existig path,
        and in the direction of the stem
    if particle has reached the stem:
        kill it
    if two particles are "close enogh" to each other:
        merge both particles
```

On the Motion/Simulation of Particles

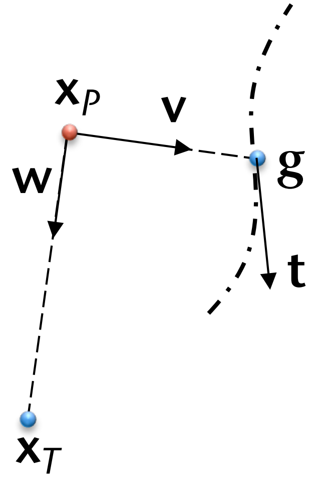
- Let $\mathbf{x}_P$ = current position of particle P
  $\mathbf{x}_T$ = target position (stem of leaf)
  $\mathbf{g}$ = point on an existing path closest to $\mathbf{x}_P$
  $\mathbf{t}$ = tangent in $\mathbf{g}$ (normalized)
  $\mathbf{x}_Q$ = particle closest to P



- If $||\mathbf{x}_P - \mathbf{g}|| < ||\mathbf{x}_P - \mathbf{x}_Q||$:
  - Let:

$$\mathbf{v} = \frac{\mathbf{g} - \mathbf{x}_P}{||\mathbf{g} - \mathbf{x}_P||}$$

$$\mathbf{w} = \frac{\mathbf{x}_T - \mathbf{x}_P}{||\mathbf{x}_T - \mathbf{x}_P||}$$



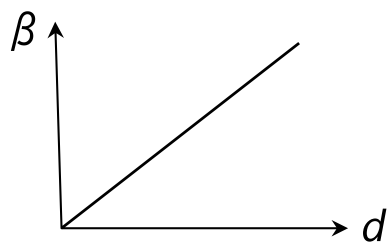
  - New position:

$$\mathbf{x}'_P = \mathbf{x}_P + \alpha\mathbf{w} + (1-\alpha)\left(\beta\mathbf{v} + (1-\beta)\,\mathbf{t}\right)$$

  with

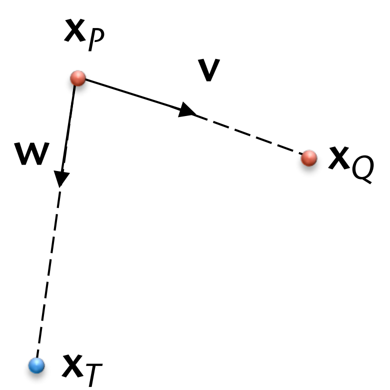
$$\beta = \beta\left(||\mathbf{x}_P - \mathbf{g}||\right)$$

- If β is (approximately) linear, this will yield particle paths, that are tangential to existing paths, and perpendicular to them when further away



- Else (i.e., $\|\mathbf{x}_P - \mathbf{x}_Q\| < \|\mathbf{x}_P - g\|$) :
  - Let

$$\mathbf{v} = \frac{\mathbf{x}_Q - \mathbf{x}_P}{\|\mathbf{x}_Q - \mathbf{x}_P\|}$$



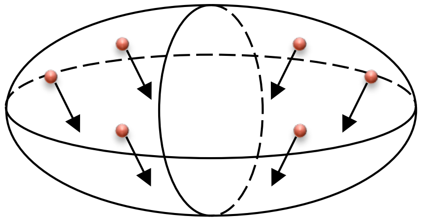
  - New position:

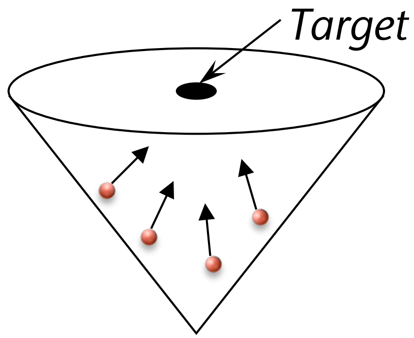
$$\mathbf{x}'_P = \mathbf{x}_P + \gamma\mathbf{v} + (1-\gamma)\mathbf{w}$$

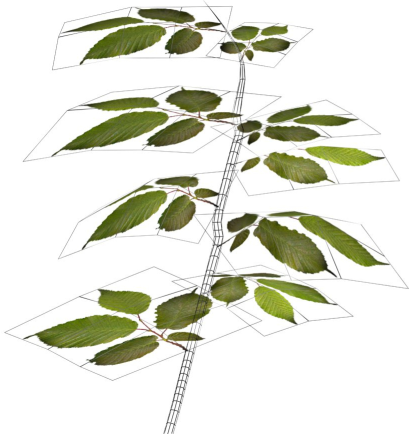
- About the thickness of the arteries:
  - Each particlke has a size = size of cross-sectional area of artery
  - At beginning: each particle has unit size
  - In case of merging: add sizes
  - In case of particle hitting existing path: add size of particle from there on until the stem (target position)

Modeling of Trees

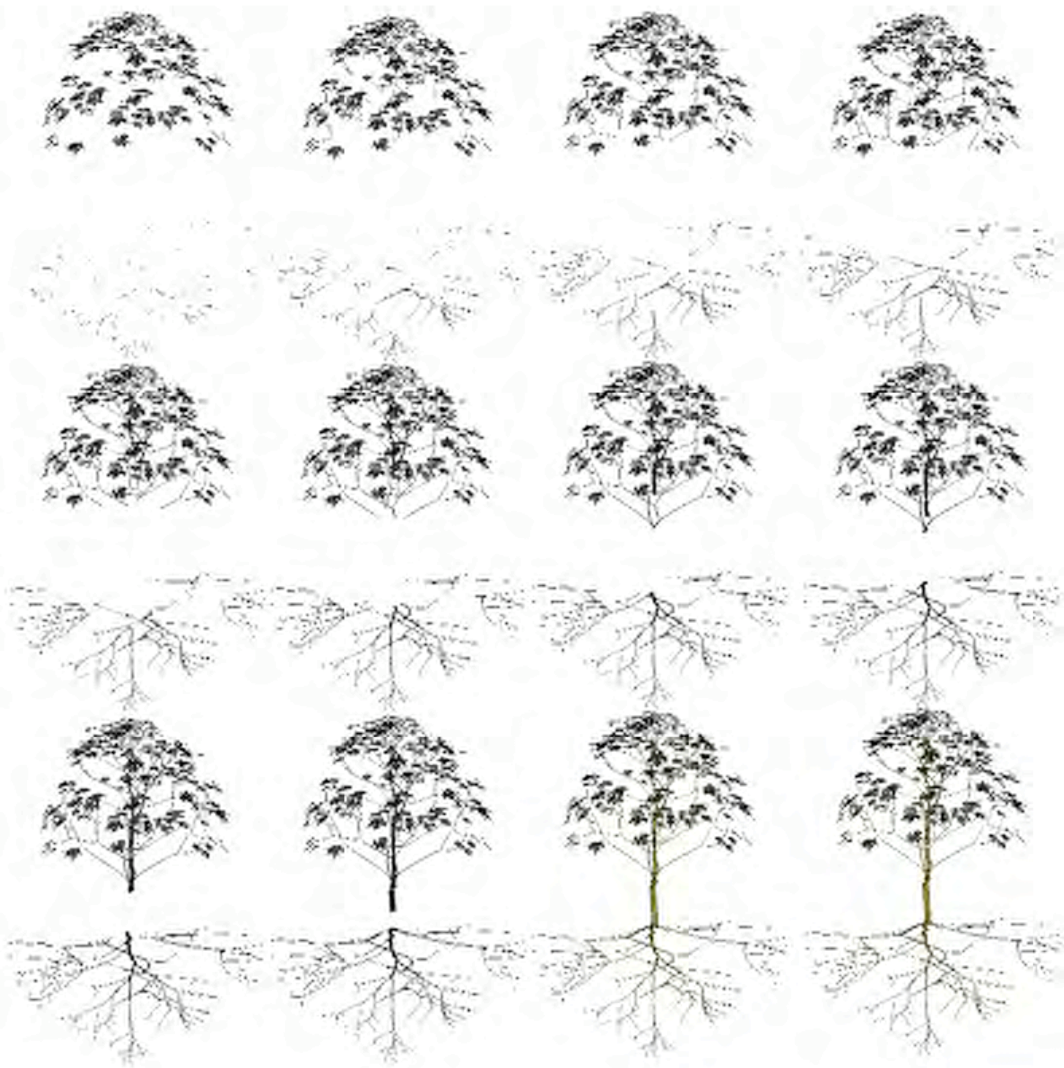
- Works exactly the same
- Input from the animator: geometry of crown (= particle source)
  - Create particles within the volume by stochastisc process
- Create geometry of branches & twigs by sweeping a disk along the path
- Place leaf primitives at end of twigs





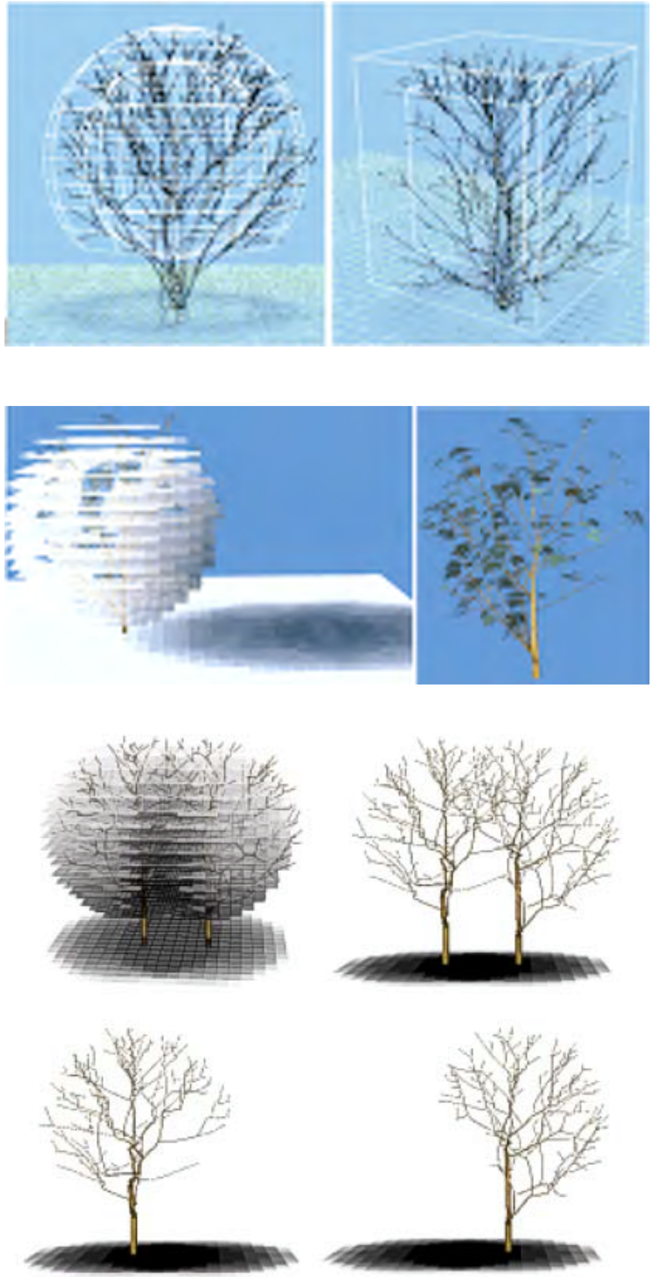


- Example of the procedural modeling process:

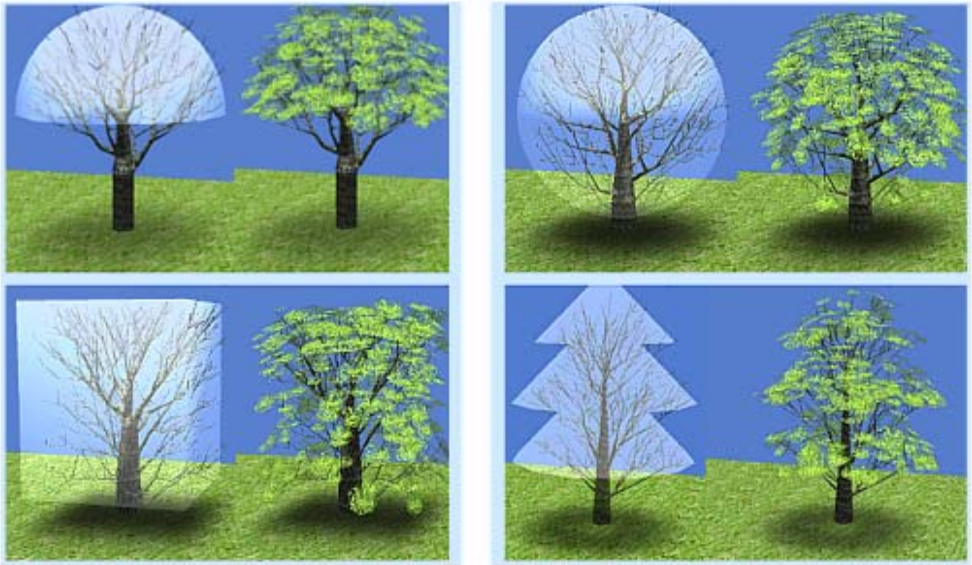


Incorporation of Lighting Conditions

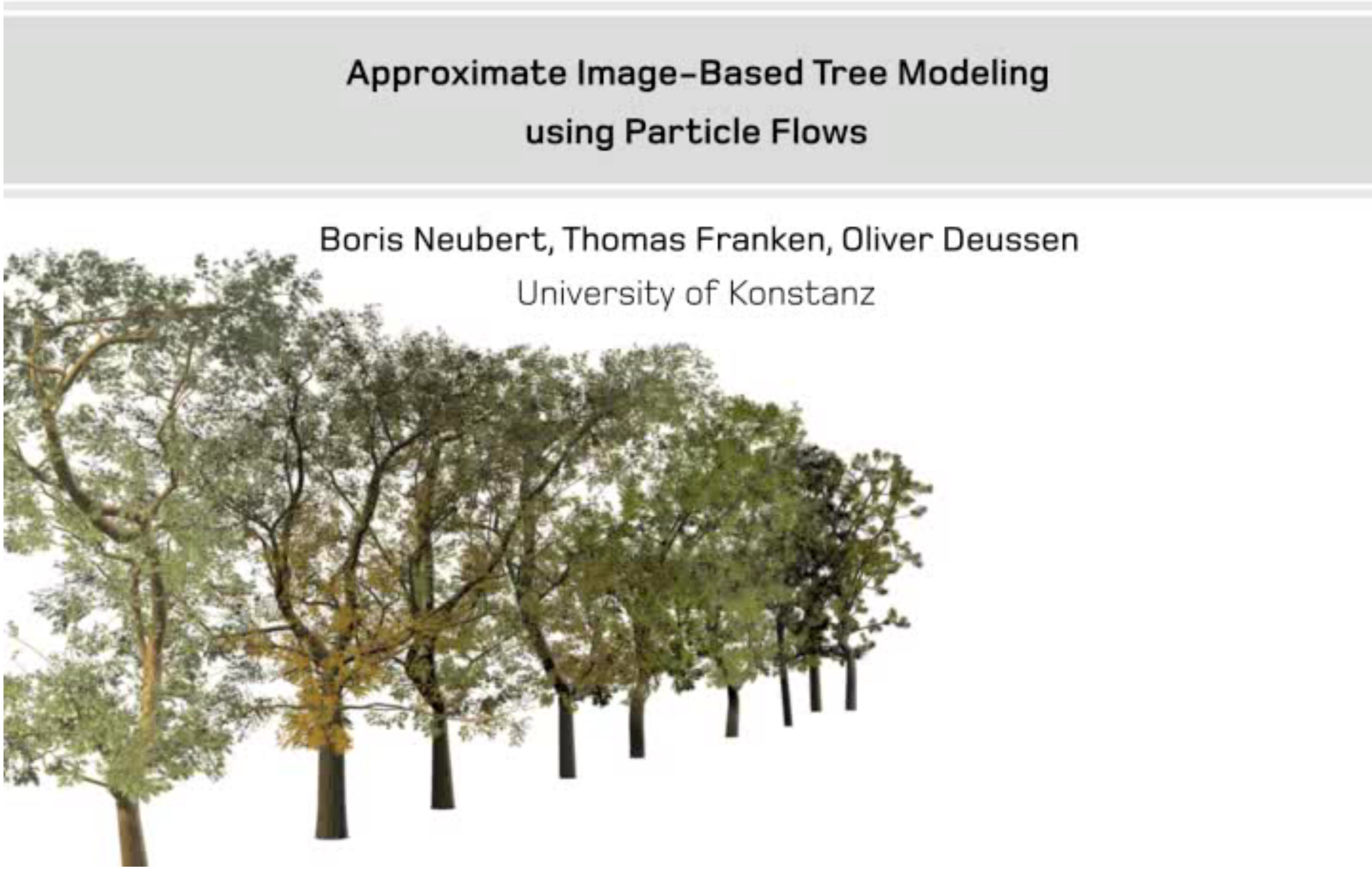
- Observation: regions with less light irradiation have less branches/leaves
- Can be modeled relatively easy:
  - Put tree inside 3D grid
  - Approximate the (not yet existing) foliage by a spherical or cubical shell
  - Compute light irradiation for each grid node by casting a ray outward
  - During particle creation: modify probability of creation according to irradiation (obtained by trilinear interpolation of grid nodes)



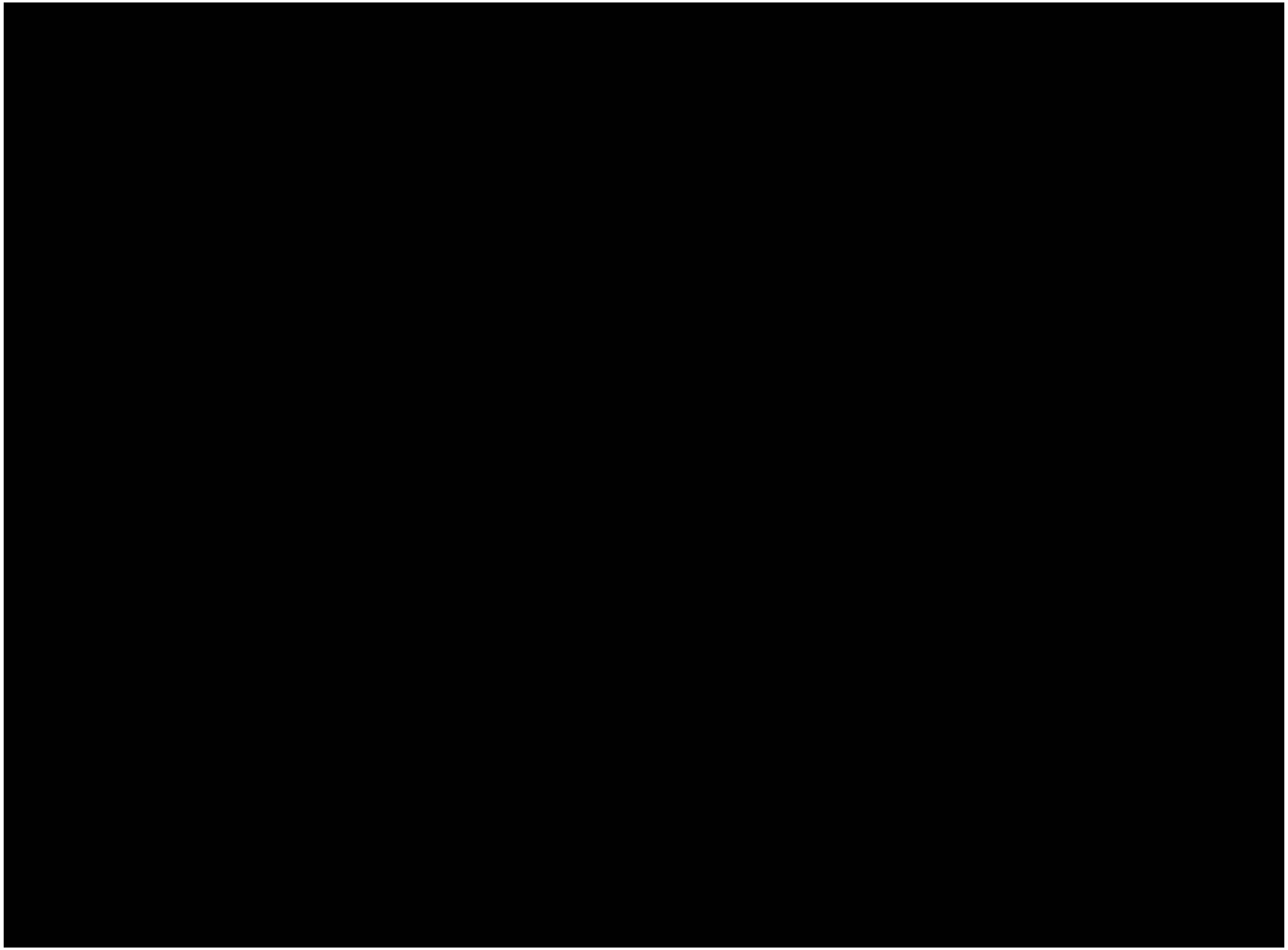
Examples



Videos


Approximate Image-Based Tree Modeling
using Particle Flows
Boris Neubert, Thomas Franken, Oliver Deussen
University of Konstanz

Vintage Video

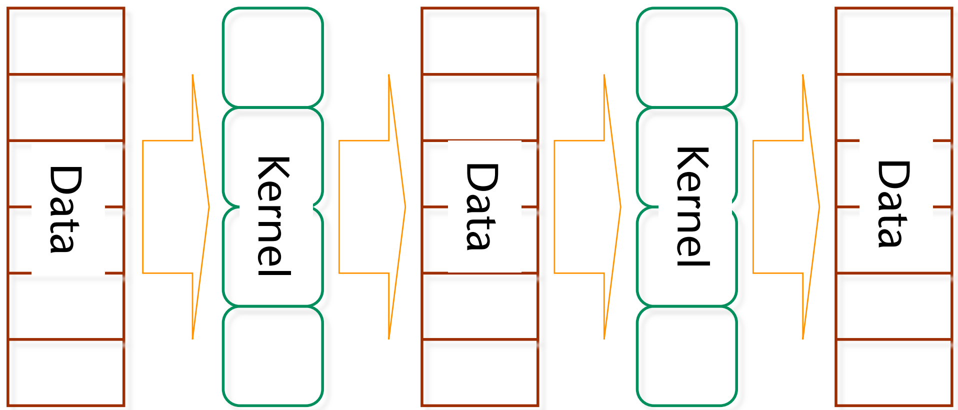


The Adventures of André and Wally B. (Pixar, 1984)

Massivle-Parallel Simulation on Stream Architectures

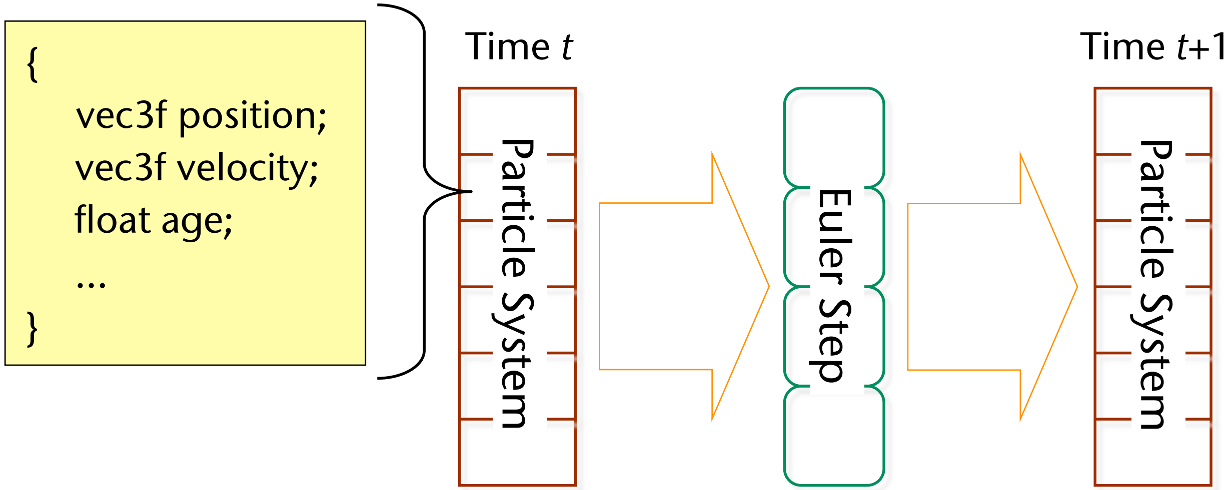
- Background on streaming architectures (and GPUs):
  - Stream Programming Model =
    "Streams of data passing through computation kernels."
  - Stream = ordered, homogenous set of data of arbitrary type
  - Kernel = program to be performed on *each* element of the input stream
- Sample stream program:

```
{
    stream A, B, C;
    ...
    kernelfunc1( input: A,
                 output: B);
    kernelfunc2( input: B,
                 output: C);
  ...
}
```



- Today's GPU's are streaming architectures, i.e., massively-parallel , general purpose computing architectures

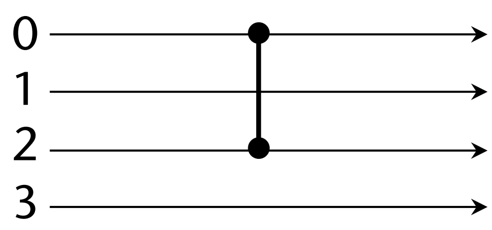
- Today's GPU's have — at least conceptually — 1000's of processors
- Each processor (kernel) can read several (a few) elements from the input stream, but it can/should write only one output element!
- Particle Simulation on GPU's:



- Managing (free) memory places:
  - When a particlel dies, record its stream index in a list
  - During particle creation: fill free positions in stream
  - Better perhaps: use p-queue instead of list, sorted by index
    - Advantage: less fragmentation (fewer "holes")
    - Disadvantage: probably impossible to create particles in parallel

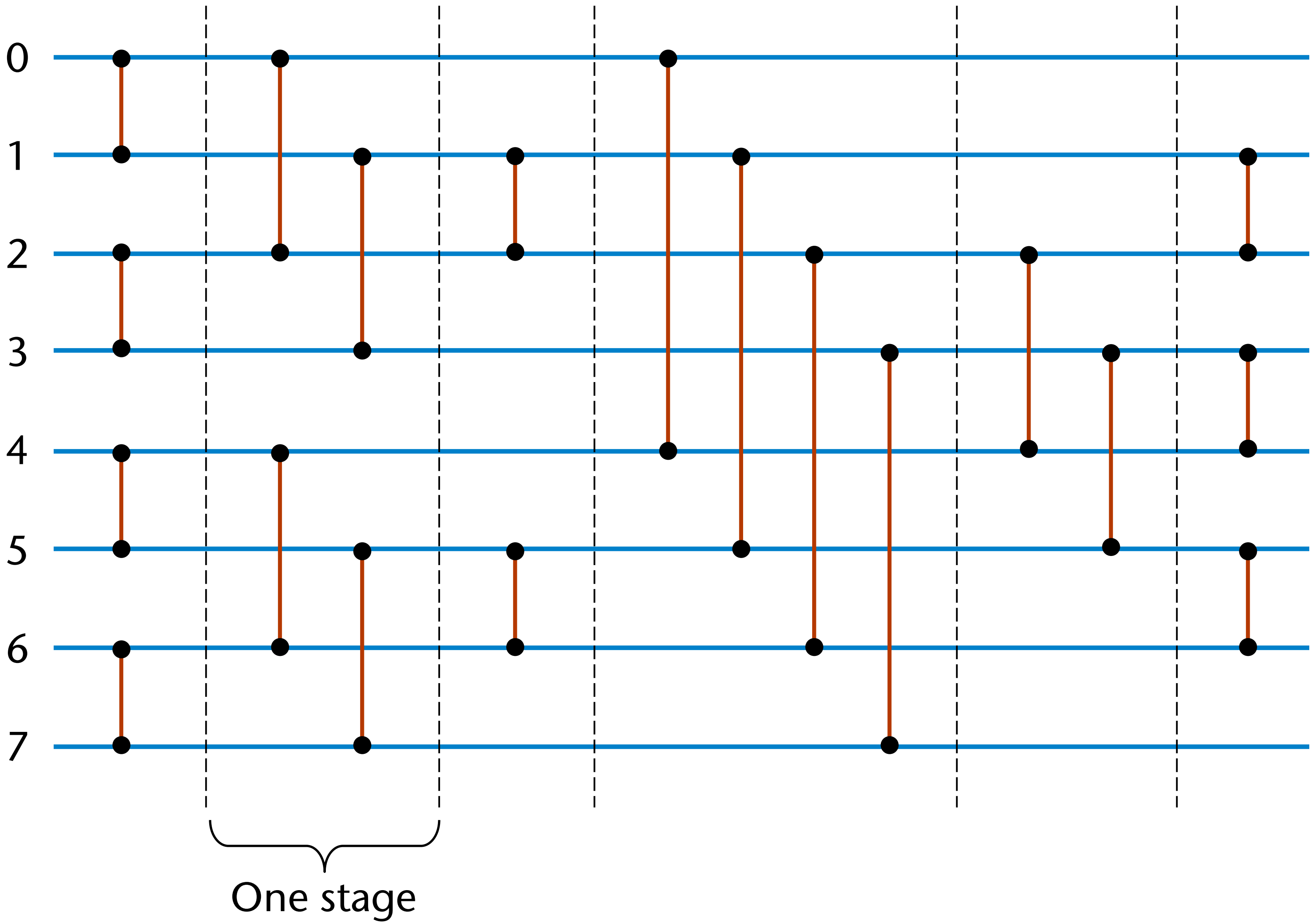
Parallel Sorting

- Reminder: sorting is needed for alpha-blending
- Solution: sorting networks
- Informal definition:
  - Consist of a bundle of "wires"
  - Each wire i carries a data element D_i (e.g., float) from left to right
  - Two wires can be connected vertically by a comparator
  - If $D_i > D_j \wedge i < j$ (i.e., wrong order), then D_i and D_j are swapped by the comparator before they move on along the wires



- Observation: every comparator network is *data independent*, i.e., the arrangement of comparators and the running time are always the same!
- Goal: find a (small) set of comparators that performs sorting for any input → sorting network

Example



The 0-1 Principle

- Definition (*monotone function*):
  Let A, B be two sets with a total ordering relation,
  and let $f : A \rightarrow B$ be a mapping.
  f is called monotone iff
  $$\forall a_1, a_2 \in A : \; a_1 \leq a_2 \Rightarrow f(a_1) \leq f(a_2)$$
- Lemma:
  Let $f : A \rightarrow B$ be monotone. Then the following holds:
  $$\forall a_1, a_2 \in A : \; f(\min(a_1, a_2)) = \min(f(a_1), f(a_2))$$
  Analogously for the max.
- Proof:
  Case 1: $a_1 \leq a_2 \Rightarrow f(a_1) \leq f(a_2)$
  $\min(a_1, a_2) = a_1 \;,\; \min(f(a_1), f(a_2)) = f(a_1)$
  $f(\min(a_1, a_2)) = f(a_1) = \min(f(a_1), f(a_2))$
  Case 2: $a_2 < a_1 \longrightarrow$ analog

- Extension of $f: A \rightarrow B$ to sequences over A and B, resp.:

$$f(a_0, \ldots, a_n) = f(a_0), \ldots, f(a_n)$$

- Lemma:

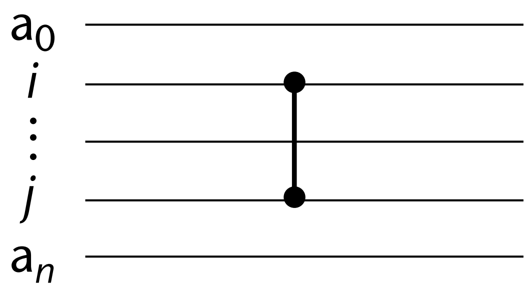
  Let f be a monotone mapping and $\mathcal{N}$ a comparator network.

  Then $\mathcal{N}$ and f commute, i.e.

$$\forall n \, \forall a_0, \ldots, a_n : \; \mathcal{N}\big(f(a)\big) = f\big(\mathcal{N}(a)\big)$$

- Proof:
  - Let $a = (a_0, \ldots, a_n)$ be a sequence
  - Notation: we write a comparator connecting wire i and j like so:

$$[i : j](a)$$



  - Now the following is true:

$$
\begin{aligned}
[i : j](f(a)) &= [i : j](f(a_0), \ldots, f(a_n)) \\
&= (f(a_0), \ldots, \underbrace{\min(f(a_i), f(a_j))}_{i}, \ldots, \underbrace{\max(f(a_i), f(a_j))}_{j}, \ldots, f(a_n)) \\
&= (f(a_0), \ldots, f(\min(a_i, a_j)), \ldots, f(\max(a_i, a_j)), \ldots, f(a_n)) \\
&= f(a_0, \ldots, \min(a_i, a_j), \ldots, \max(a_i, a_j), \ldots, a_n) \\
&= f([i : j](a))
\end{aligned}
$$

- Theorem (the 0-1 principle):
  Let $\mathcal{N}$ be a comparator network.
  Now, if $\mathcal{N}$ sorts every sequence of 0's and 1's, then it also sorts every sequence of elements!

- Proof (by contradiction):
  - Assumption: $\mathcal{N}$ sorts all 0-1 sequences, but does not sort sequence a
  - Then $\mathcal{N}(a) = b$ is not sorted correctly, i.e. $\exists k : b_k > b_{k+1}$
  - Define $f : A \rightarrow \{0,1\}$ as follows:

$$f(c) = \begin{cases} 0, & c < b_k \\ 1, & c \geq b_k \end{cases}$$

  - Now, the following holds:

$$f(b) = f\left(\mathcal{N}(a)\right) \underset{\uparrow \atop f \text{ monotone}}{=} \mathcal{N}\left(f(a)\right) = \mathcal{N}(a')$$

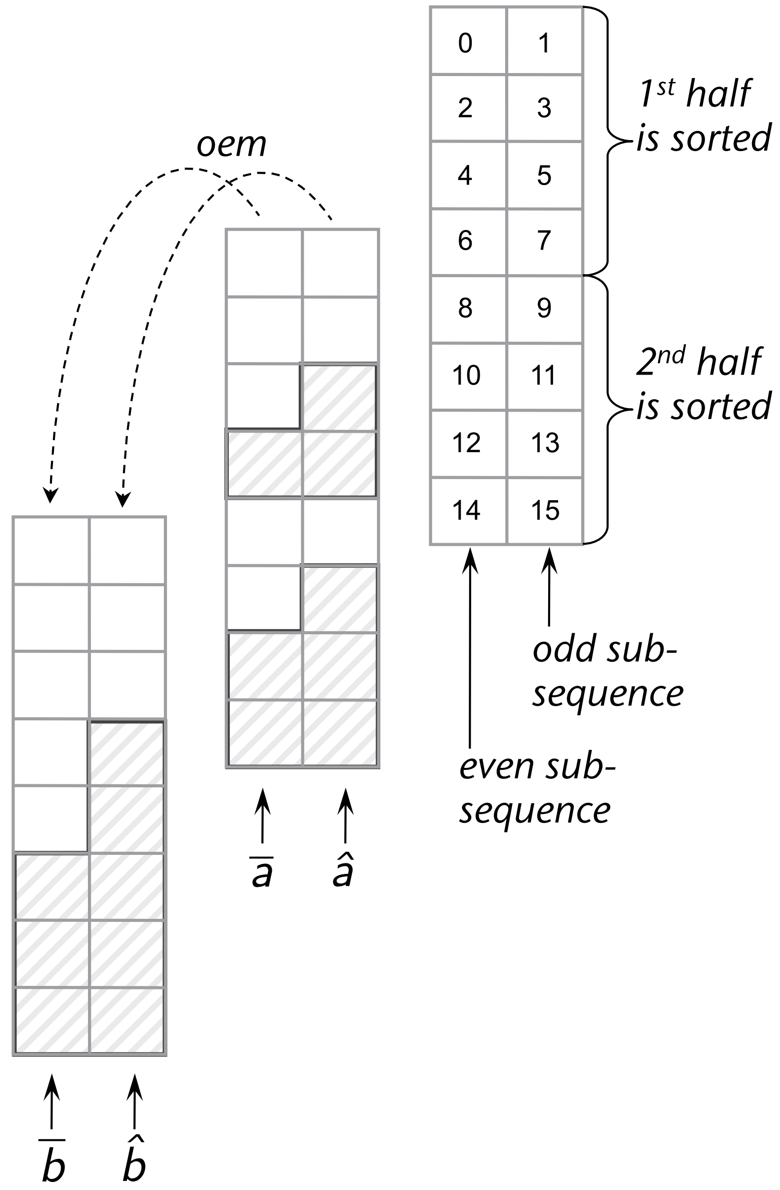
    where a' is a 0-1 sequence.
  - But: $f(b)$ is not sorted, because $f(b_k) = 1$ and $f(b_{k+1}) = 0$
  - Therefore, $\mathcal{N}(a')$ is not sorted as well, in other words, we have constructed a 0-1 sequence that is not sorted correctly by $\mathcal{N}$.

Batcher's Odd-Even-Mergesort [1968]

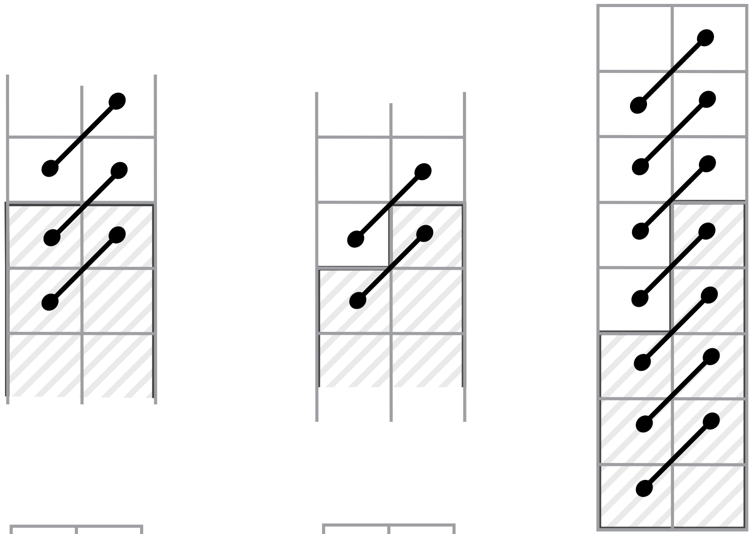
- In the following, we'll always assume that the length n of a sequence $a_0,\ldots,a_{n-1}$ is a power of 2, i.e., $n = 2^k$
- First of all, we define the sub-routine "odd-even merge":

```
oem( a_0,…,a_{n-1} ):
precondition:  a_0,…,a_{n/2-1}  and  a_{n/2},…,a_{n-1}  are both sorted
postcondition: a_0,…,a_{n-1}  is sorted
if  n = 2:
     compare [a_0:a_1]                                        (1)
if  n > 2:
     ā ← a_0,a_2,…,a_{n-2}    (even sub-sequence)
     â ← a_1,a_3,…,a_{n-1}    (odd sub-sequence)
     b̄ ← oem( ā )
     b̂ ← oem( â )                                             (*)
     copy b̄ → a_0,a_2,…,a_{n-2}
     copy b̂ → a_1,a_3,…,a_{n-1}                               (**)
     for i ∈ {1,3,5,…,n-3}
          compare [a_i : a_{i+1}]                             (2)
```

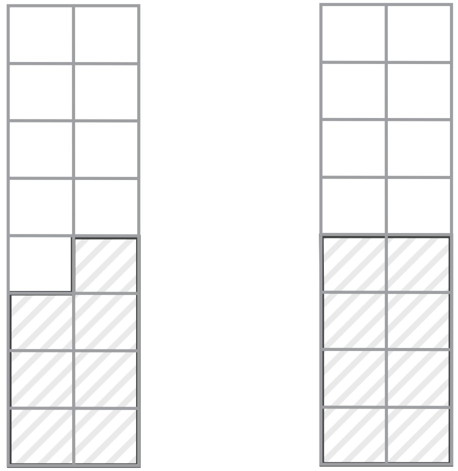
- Proof of correctness:
  - By induction and the 0-1-principle
  - Base case: $n = 2$
  - Induction step: $n = 2^k$, $k > 1$
  - Consider a 0-1-sequence $a_0,\ldots,a_{n-1}$
  - Write it in two columns
  - Visualize 0 = white, 1 = grey
  - Obviously: both $\bar{a}$ and $\hat{a}$ consist of two sorted halves → preconditon of *oem* is met
  - After line (*) we have this situation (the odd sub-sequence can have at most two 1's more than the even sub-sequence)



- After line (**), these comparisons are made, and there can be only 3 cases:



- Afterwards, one of these two situations has been established:
- Result: the output sequence is sorted



- Conclusion:
  each 0-1-sequence (meeting the preconditons) is sorted correctly
- Running time: $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{2} - 1 \in O\left(n \log n\right)$

- The complete general sorting-algorithm:

```
oemSort(a_0,…,a_{n-1}):
if n = 1:
   return
a_0,…,a_{n/2-1} ← oemSort(a_0,…,a_{n/2-1})
a_{n/2},…,a_{n-1} ← oemSort(a_{n/2},…,a_{n-1})
oem(a_0,…,a_{n-1})
```

- Running time: $T(n) \in O\left(n \log^2 n\right)$

The Mapping on a Streaming Architecture

- Load data into a stream on the GPU (here, a global variable)
- The CPU executes the following program:

```
oemSort(n):
if n = 1 → return
oemSort( n/2 )
oem( n, 1 )
```

```
oem( n, stride ):
if n = 2:
    execute oemEndKernel
    // launches n parallel exec's
else:
    oem( n/2, stride*2 )
    execute oemRecursionKernel
```

- With the stride parameter,we can achieve sorting "in situ"

- Kernel for base case of recursion:

```
oemEndKernel ( i, stride ):
// are we on the even or the odd side?
if i/stride is even:
    div = 1
else:
    div = -1
a0 ← SortData[i]                // SortData = stream =
a1 ← SortData[ i+div+stride ]   // globales "array"
if div > 0:
    output max(a0,a1)           // write output
else:
    output min(a0,a1)           // in output stream
```

- The oemEndKernel implements line (1) of the algorithm
- Reminder: a kernel is executed in parallel for each index $i = 0, \ldots, n$-1 in a stream; i is provided by the GPU, not the CPU!

- The kernel for finishing up a recursion:

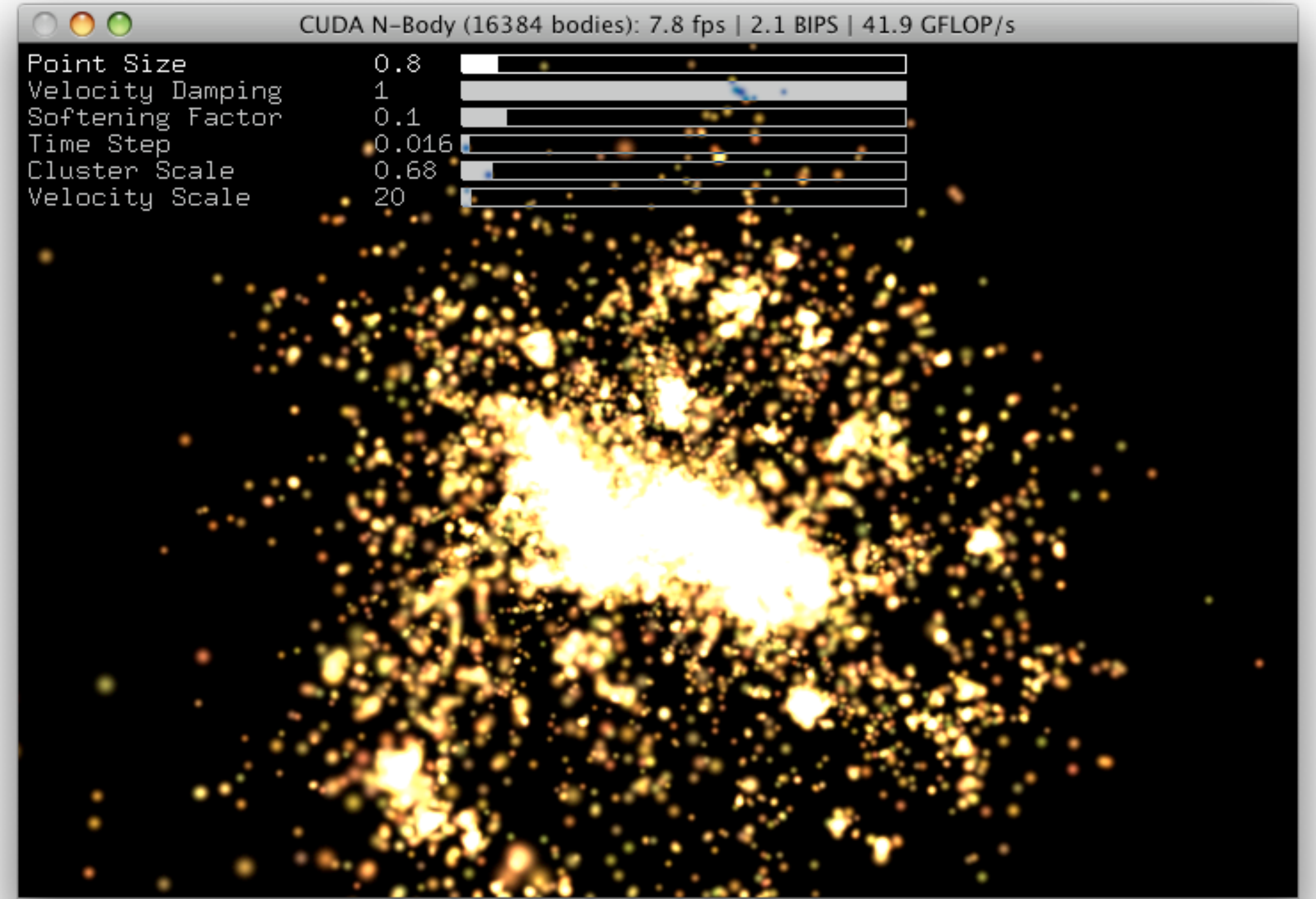
```
oemRecursionKernel( i, stride ):
if i < stride || i ≥ n-stride:
    output SortData[i]
else:
    a_i ← SortData[i]
    a_i_plus_1 ← SortData[ i+stride ]
    if i/stride is even:
        output max( a_i, a_i_plus_1 )
    else:
        output min( a_i, a_i_plus_1 )
```

- The oemRecursionKerneler implements line (2) of the algorithm
- Again, index $i = 0, \ldots, n\text{-}1$

- Running time: $\frac{1}{2}\log^2 n + \frac{1}{2}\log n$ rendering passes

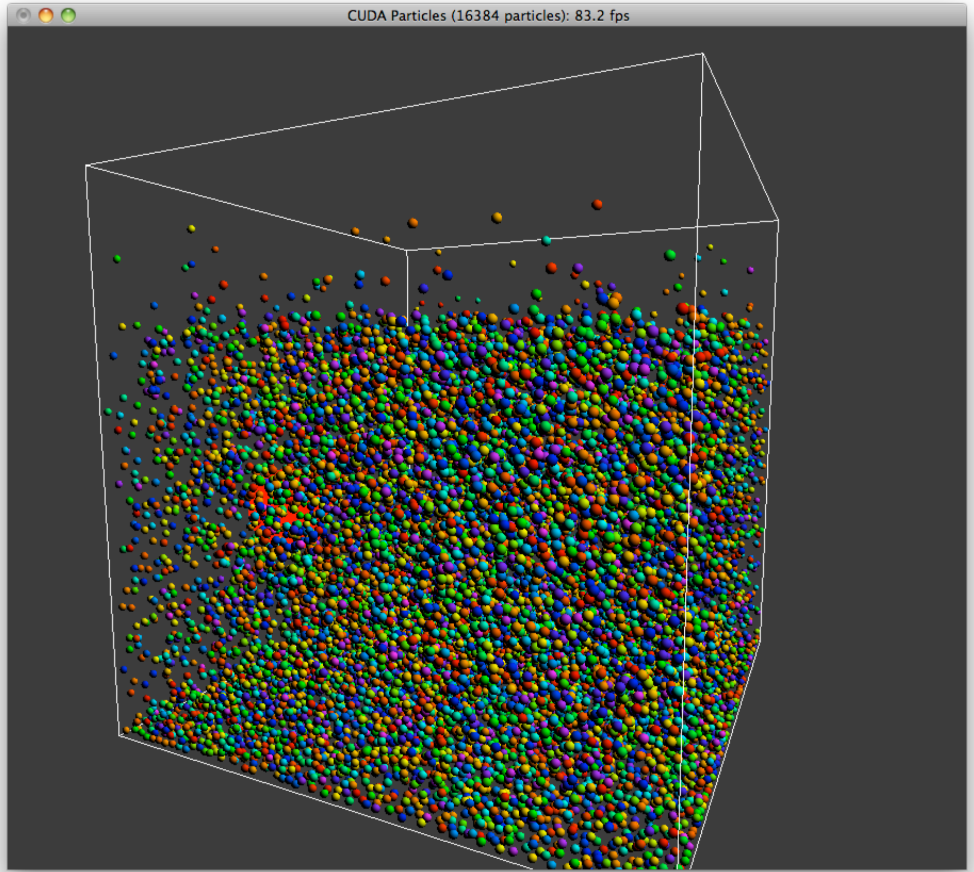
- This are 210 passes for 2^{20} particles
  - For particle systems, this can be done incrementally, i.e. only a few sorting passes per frame (might return "not quite" sorted particles, which is sometimes OK, e.g. for fire)

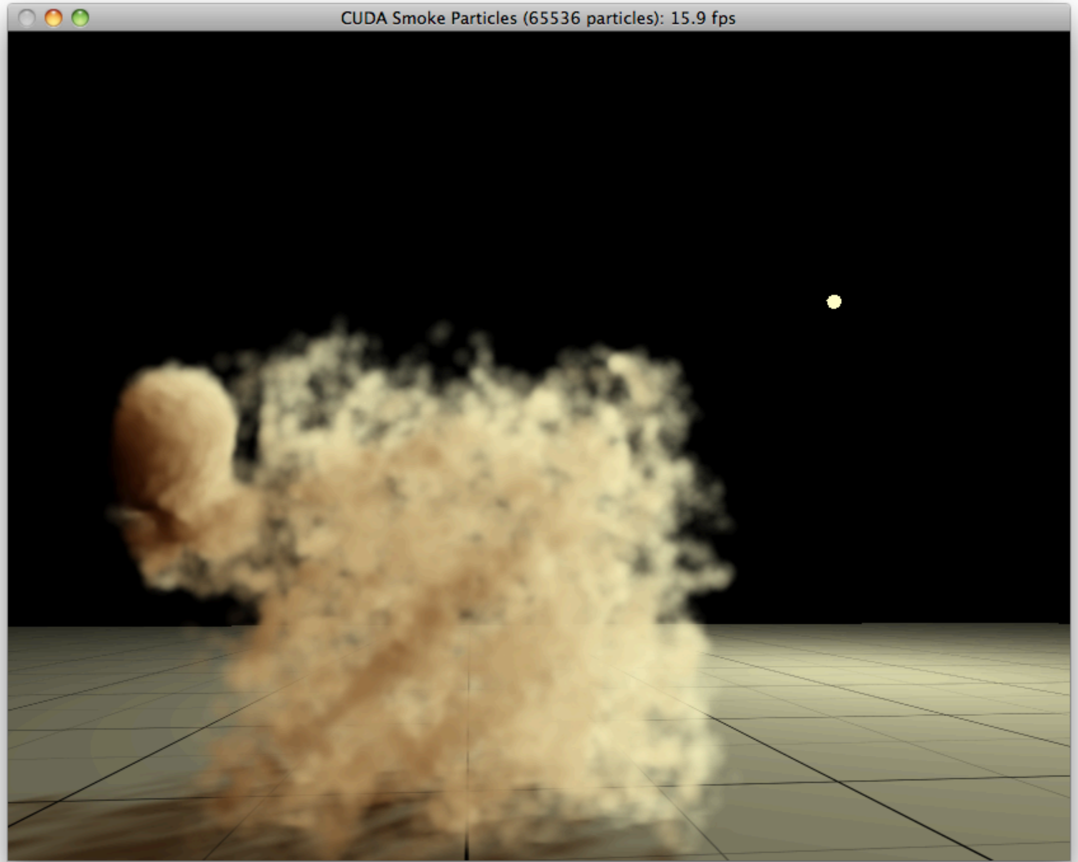
Demos



N-body simulation

http://www.nvidia.com/cuda


CUDA Particles (16384 particles): 83.2 fps


CUDA Smoke Particles (65536 particles): 15.9 fps