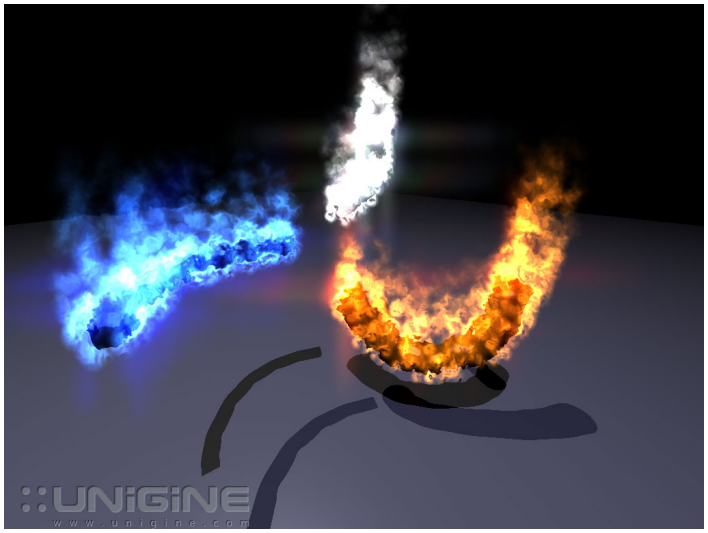
Virtuelle Realität
Partikelsysteme

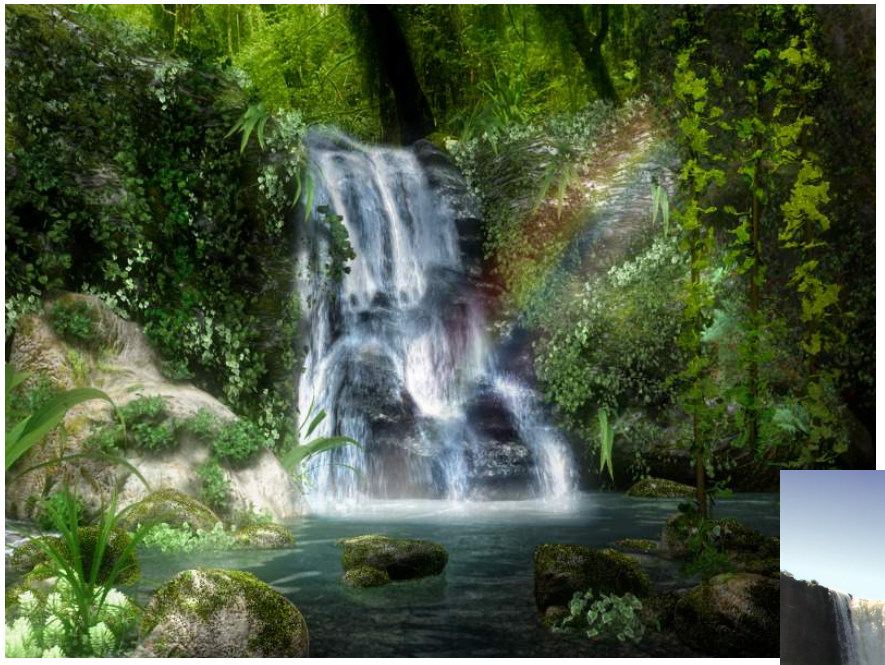


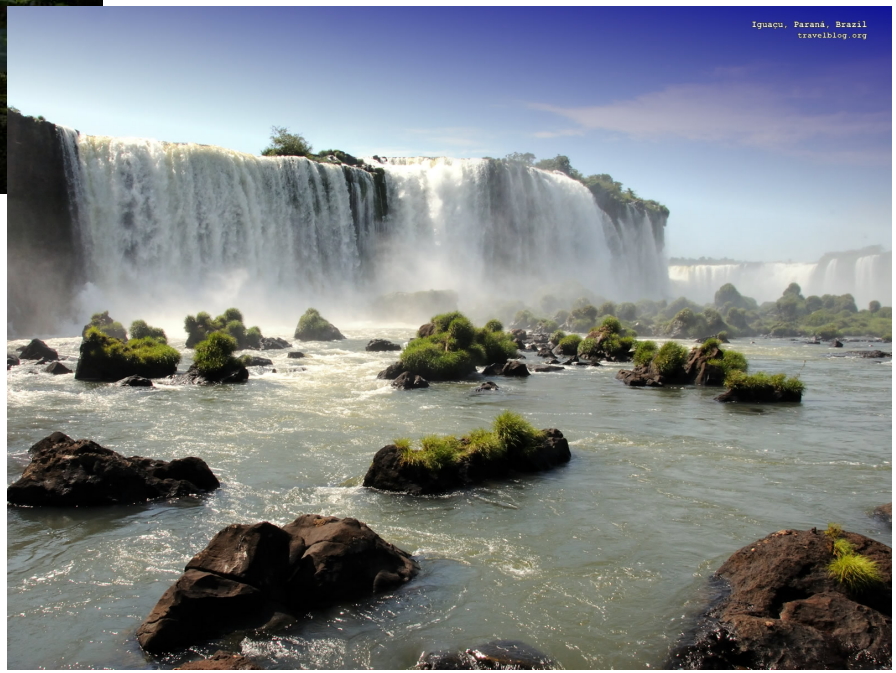
G. Zachmann

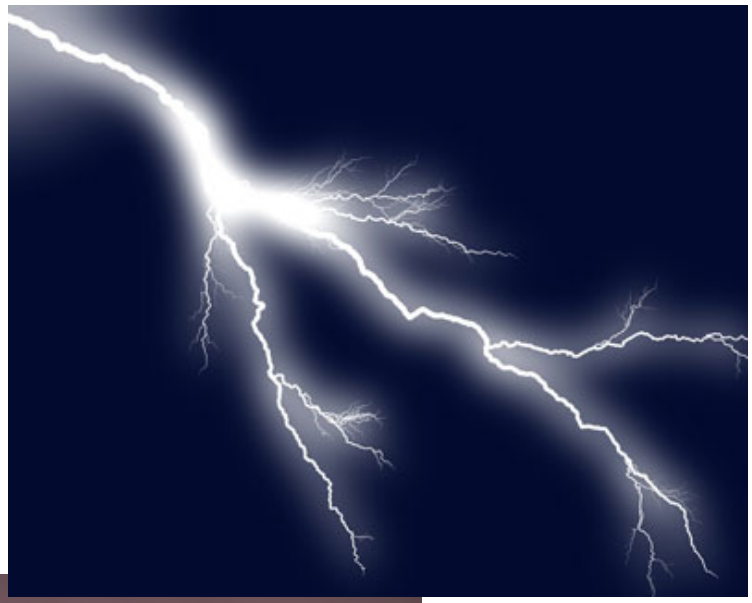
University of Bremen, Germany

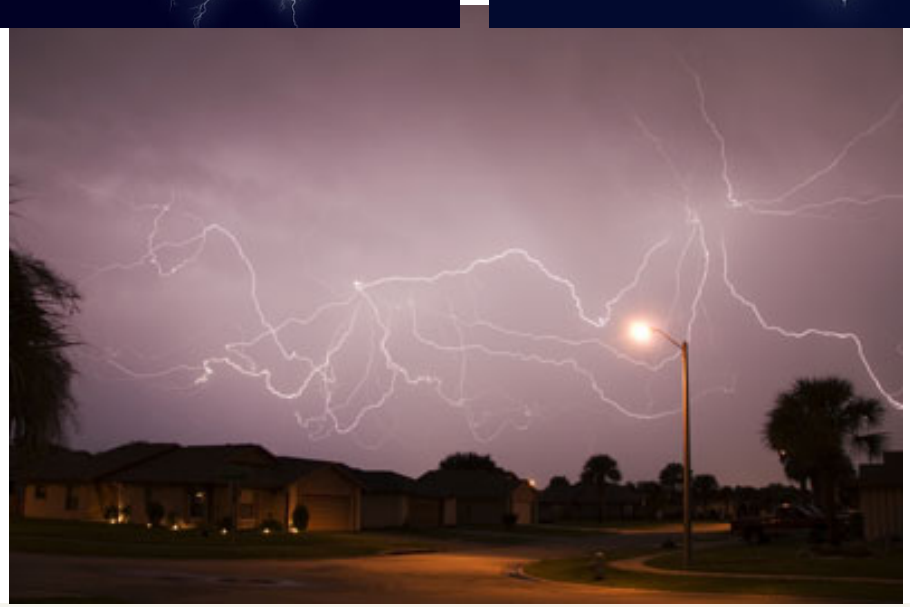
cgvr.cs.uni-bremen.de

Modellierung/Simulation/Rendering natürlicher Phänomene




Iguaçu, Paraná, Brazil
travelblog.org



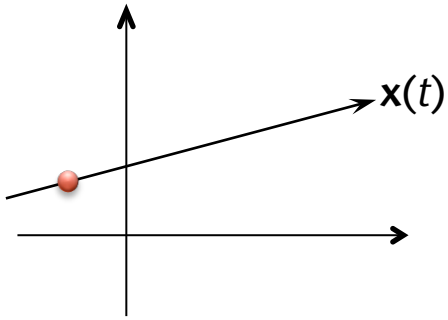


Dynamik eines Massenpunktes

- Definition Partikel:

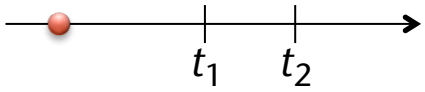
  Ein Partikel ist ein ideeller Punkt mit einer Masse m und einer Geschwindigkeit $\mathbf{v}$.

  →Die Orientierung ist irrelevant



- Bahn eines Partikels: $\mathbf{x}(t)$
- Geschwindigkeit:

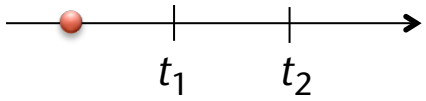
$$\mathbf{v} = \frac{\text{Weg}}{\text{Zeit}} = \frac{\mathbf{x}(t_2) - \mathbf{x}(t_1)}{t_2 - t_1}$$



  - Einheit: m/s
  - Beachte: Geschwindigkeit = Vektor

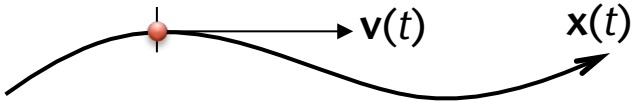
    Ort des Partikels = Punkt!

- Momentangeschwindigkeit:



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

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



- Beispiele:
  - Punkt bewegt sich auf Kreisbahn → $\|\dot{\mathbf{x}}\|$ ist konstant
  - Punkt beschleunigt auf Gerade → $\frac{\dot{\mathbf{x}}}{\|\dot{\mathbf{x}}\|}$ ist konstant
- Beschleunigung :

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

Newtons 2. Gesetz

Euler-Integration

- Gegeben: ein Partikel der Masse m; eine Kraft $\mathbf{F}(t)$, die auf das Partikel über die Zeit wirkt
- Gesucht: die Bahn $\mathbf{x}(t)$ des Partikels
- Analytischer Ansatz:

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

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

- Diskretisieren und Linearisieren:

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

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

oder

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

(*approx. midpoint method*)

Der Phasenraum

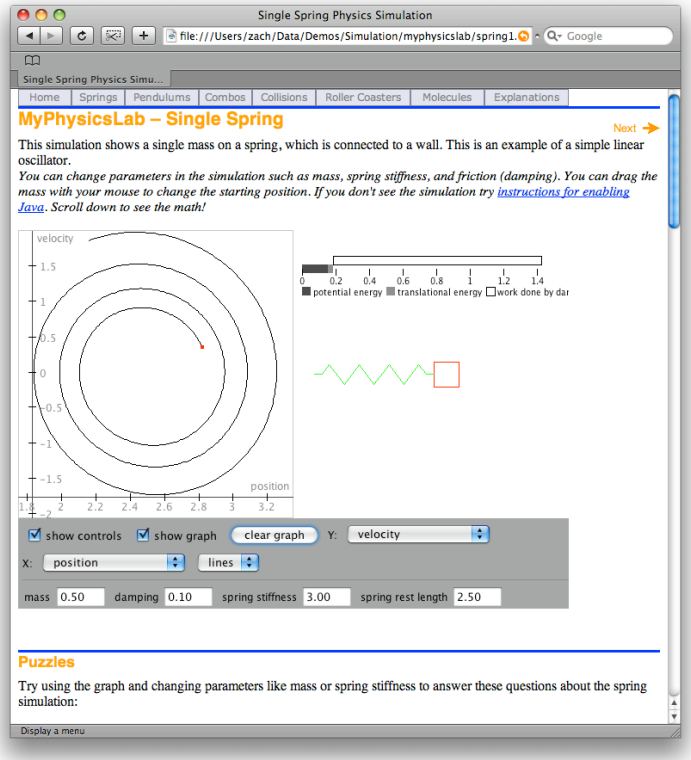
- Der (physikalische) momentane Zustand eines Partikels ist vollständig beschrieben durch

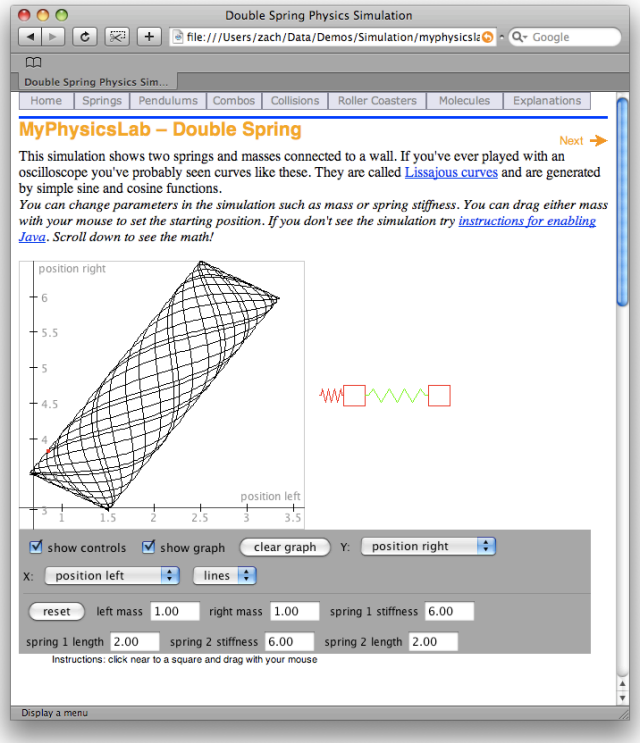
$$\begin{aligned}\mathbf{q} &= (\mathbf{x}, \mathbf{v}) = (x_1, x_2, x_3, v_1, v_2, v_3) \\ &= (x_1, x_2, x_3, \dot{x}_1, \dot{x}_2, \dot{x}_3) \in \mathbb{R}^6\end{aligned}$$

- Der Raum aller möglischer Zustände heißt Phasenraum (*phase space*)
- Die Dimension ist $6n$, n = Anzahl Partikel
- Bewegungsgleichungen im Phasenraum:

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

- Beispiel für ein Partikel, das sich nur auf der x-Achse bewegen kann und durch eine Feder in einer Ruhelage gehalten wird:





www.myphysicslab.com

Der Laplace'sche Dämon

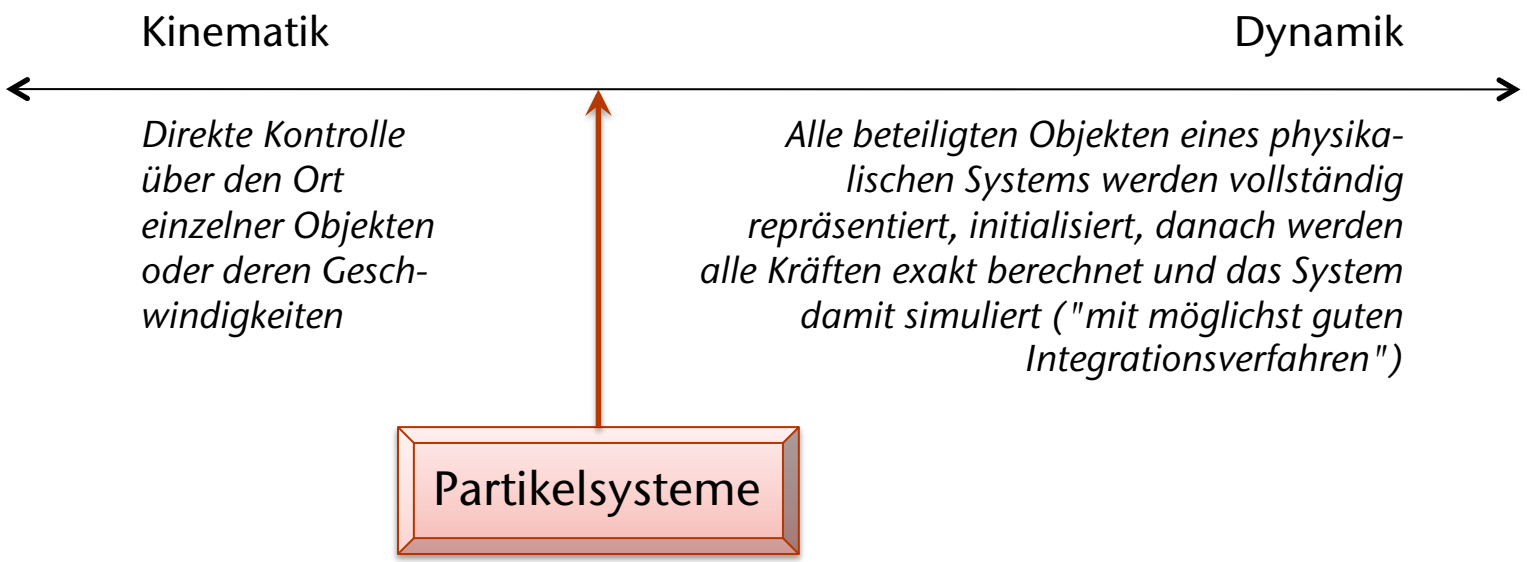
Kinematik versus Dynamik

- Begriffe:

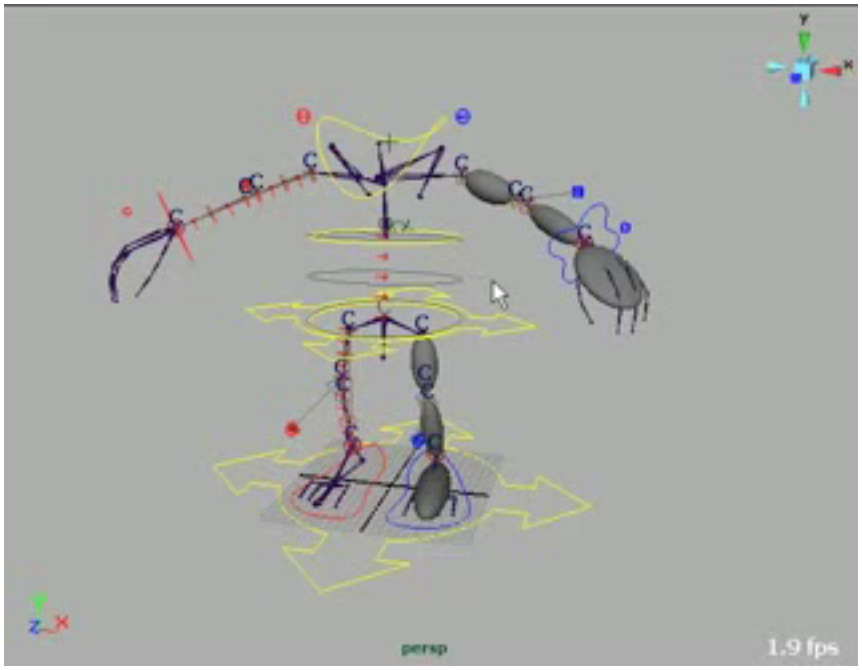
  Kinematik = Bewegung von Körpern ohne Simulation von Kräften

  Dynamik = Simulation / Berechnung von Kräften und die daraus resultierende Bewegungen der Objekte

- In der Computergraphik bewegt man sich in einem Kontinuum:



- Beispiel für reine Kinematik: inverse Kinematik



Partikelsysteme

- Definition: Ein Partikelsystem besteht aus
  1. Einer Menge von Partikeln; jedes Partikel i hat (mindestens) folgende Attribute:
     - Masse, Position, Geschwindigkeit (m_i , $\mathbf{x}_i$, $\mathbf{v}_i$)
     - Alter a_i
     - Kräfteakkumulator $\mathbf{F}_i$
     - evtl.: wie z.B. Farbe, Transparez, Optische Größe, Lebensdauer, Typ/Art ...
  2. Einer Menge Partikelquellen; jede ist beschrieben durch
     - Form der Partikelquelle
     - Stochastische Prozesse, die die initialen Attribute der Partikel festlegen (Geschwindigkeit, Richtung, etc.)
     - Stochastische Prozesse, die die Anzahl der erzeugten Partikel pro Frame festlegen
  3. Weitere (globale) Parameter, z.B.
     - TTL (time to live) = max. Lebensdauer eines Partikels
     - Globale Kräfte (z.B. Gravitation, Wind, ...)
     - die Algorithmen, die die Partikel bewegen und rendern

- Stochastischer Prozeß =
  - Im einfachsten Fall: Mittelwert + Varianz; Prozeß liefert zufälligen Wert gemäß Gleichverteilung
  - Etwas komplizierter: Mittelwert und Varianz sind Funktionen der Zeit

- Form der Partikelquelle:
  - Ist intutive Art, den stochastischen Prozess für die initiale Position von Partikeln zu beschreiben
  - Häufig: Kreisscheibe, Würfel, Kegel, etc.

Das Execution Model

- Der Ablauf eines Partikelsystems:

```
loop forever:
  rendere alle Partikel
  Δt := Rendering-Zeit
  kille alle Partikel mit Alter > TTL (max. Lebensdauer)
  erzeuge neue Partikel an der Quelle
  lösche alle Kräfteakkumulatoren
  berechne alle Kräfte auf jedes Partikel (akkumuliere diese)
  aktualisiere Geschwindigkeit (ein Eulerschritt mit Δt)
  modifiziere gegebenenfalls Geschwindigkeiten (*)
  aktualisiere Positionen (ein weiterer Eulerschritt)
  modifiziere eventuell Positionen (z.B. wg. Constraints)
  sortiere Partikel nach Tiefe (für Alpha-Rendering)
```

Bemerkungen

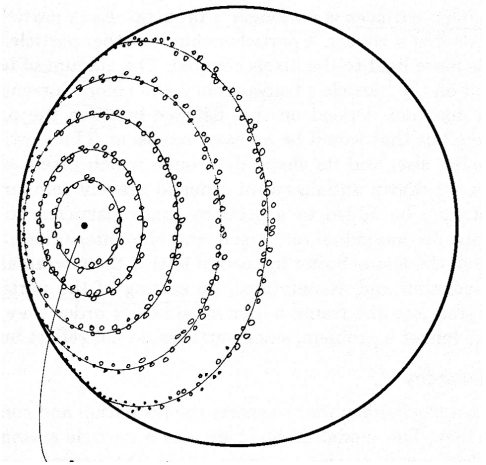
- Hier gibt es viel Raum für Optimierungen, z. B.
  - Gravitationskraft gleich beim Löschen des F-Akkumulators setzen
  - Nicht bei jedem Partikel händisch das Alter inkrementieren, sondern Zeit t_{gen} der Entstehung speichern, dann nur noch $t_{current} - t_{gen} >$ TTL testen
    - Wird später bei paralleler Implementierung wichtig
- Zu (*) im Algorithmus:
  - Ist "un-physikalisch", erlaubt aber bessere kinematische Kontrolle durch den Programmierer / Animateur
  - Ist auch bei Kollisionen nötig
- Der Rest ist Intuition und Kreativität ...
- Oft speichert man eine kleine Historie der Positionen der Partikel, um einen einfachen "motation blur"-Effekt zu erhalten
- Partikel können auch auf Grund anderer Bedingungen gekillt werden, z.B. Entfernung von der Quelle, Eintritt in einer bestimmter Region, etc.
- Achtung, für eine effiziente Implementierung kann eine "Struct-of-Array"-Datenstruktur besser sein! (SoA statt AoS)

Beispiel eines Partikelsystems

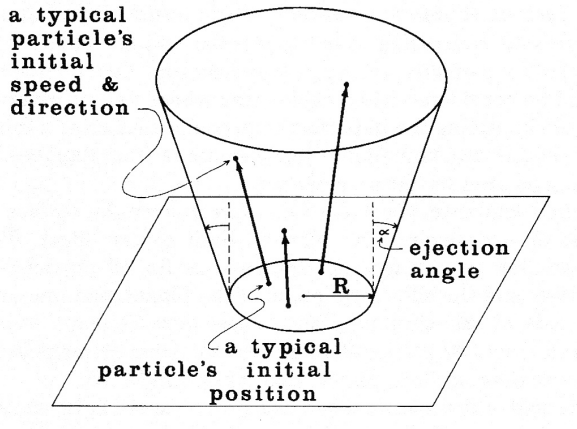
- Ausschnitt aus "Wrath of Khan":

(William Reeves, 1984)

- Partikelquelle = Kreis auf der Kugel um den *impact point*, der sich vergrößert

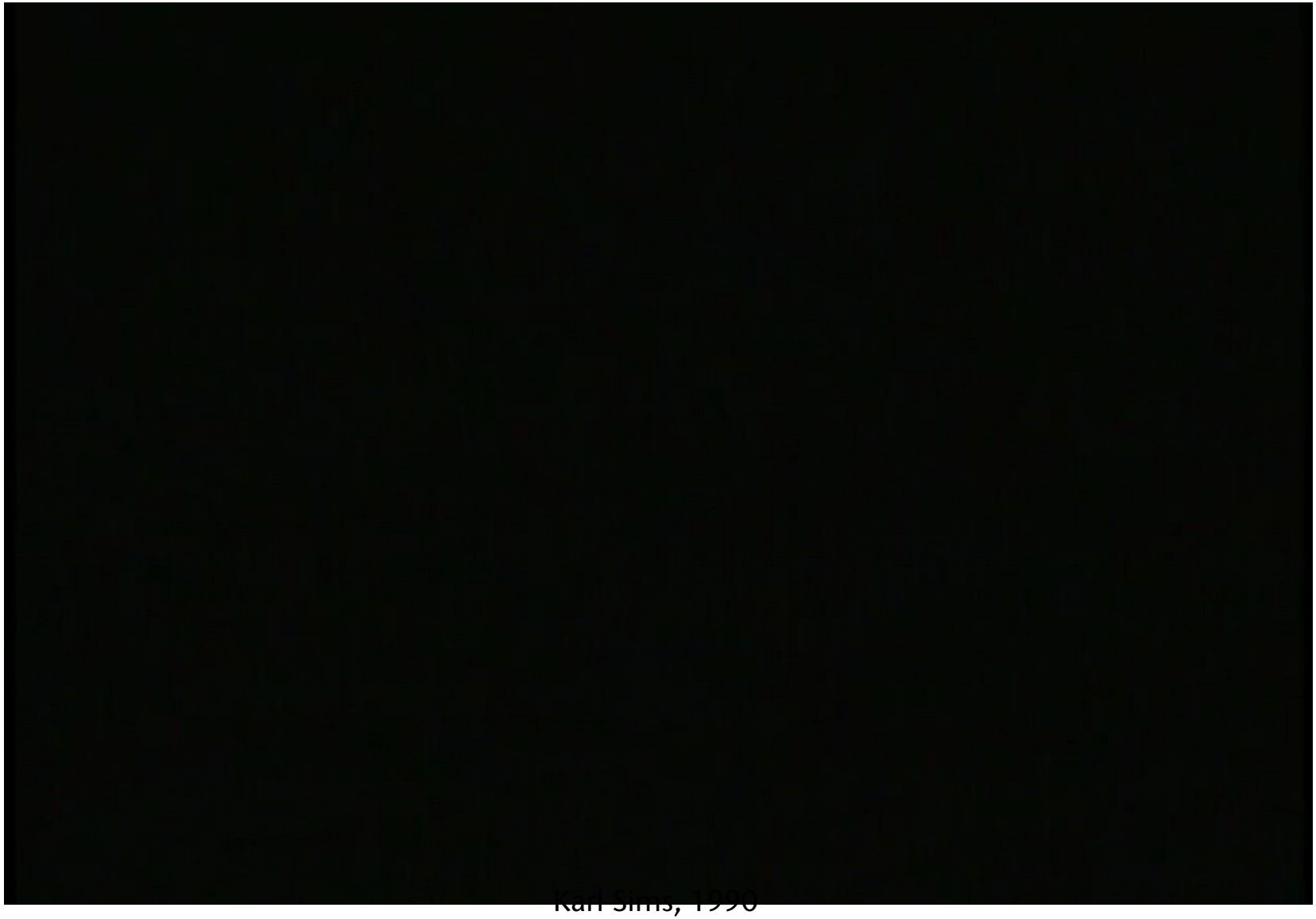


- Stochastische Prozesse für Partikel-generierung:
  - Kegelstumpf senkrecht zu Kugeloberfläche
  - Varianz für Lebendauer



- Farbe = *f*(*Alter*)

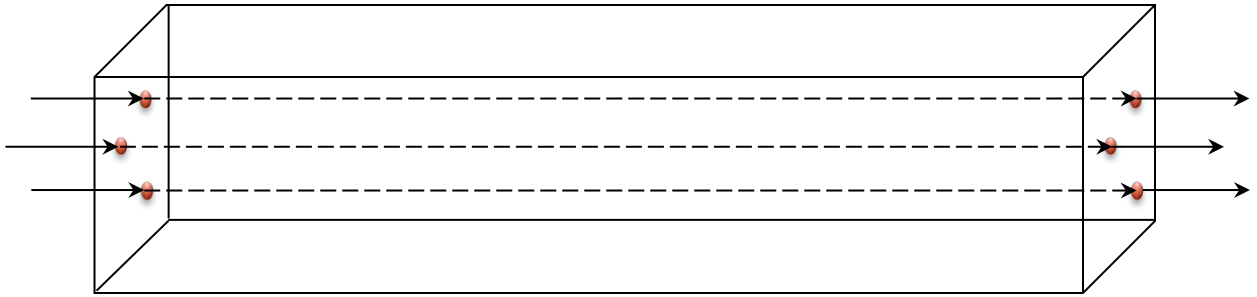
Exkurs: die Panspermie-Hypothese



Karl Sims, 1990

Operationen auf Partikeln

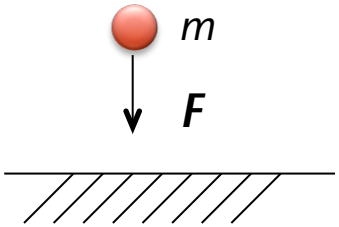
- Positionsoperationen:
  - Eher selten
  - z.B. “Tunneln”



Physikalische Effekte

- Schwerkraft:

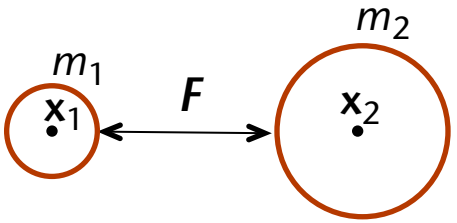
$$\mathbf{F} = m \cdot \mathbf{g} \quad , \qquad g = 9.81 \frac{\mathrm{m}}{\mathrm{s}^2}$$



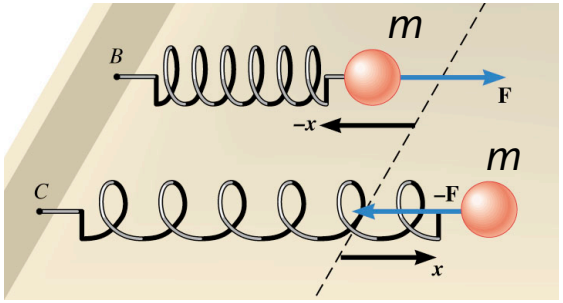
- Gravitation:

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

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



- Federkraft: später



- Viskose Hemmung/Dämpfung (viscous drag):

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

in einem ruhenden Fluid/Gas;

oder auch

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

bei Fluid mit Geschwindigkeit $\mathbf{v}_{fl}$, Partikel mit Radius r, Viskosität η;

oder auch

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

bei hoher Geschwindigkeit; ρ = Dichte, A = Querschnittfläche des Körpers, c = Viskositätskonstante

- Elektromagnetische Kraft (Lorentz-Kraft):

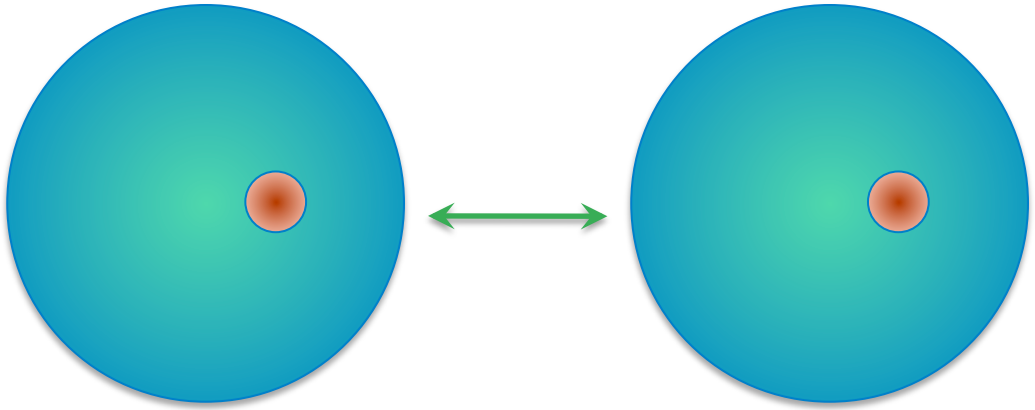
$$\mathbf{F} = q \cdot \mathbf{v} \times \mathbf{B}$$

wobei q die Ladung des Partikels , $\mathbf{v}$ dessen Geschwindigkeit, und $\mathbf{B}$ das magnetische Feld ist.

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

Die Lennard-Jones-Kraft

- Zwischen neutralen Atomen gibt es zwei Arten von Kräften:
  - Eine abstoßende Kraft auf kurze Distanzen
  - Eine anziehende Kraft auf größere Entfernung (van der Waals-Kräfte oder Dispersionskraft)



instantaneous dipole induced dipole

- Eine (willkürliche) Approximation ist das Lennard-Jones-Potential bzw. Lennard-Jones-Kraft:

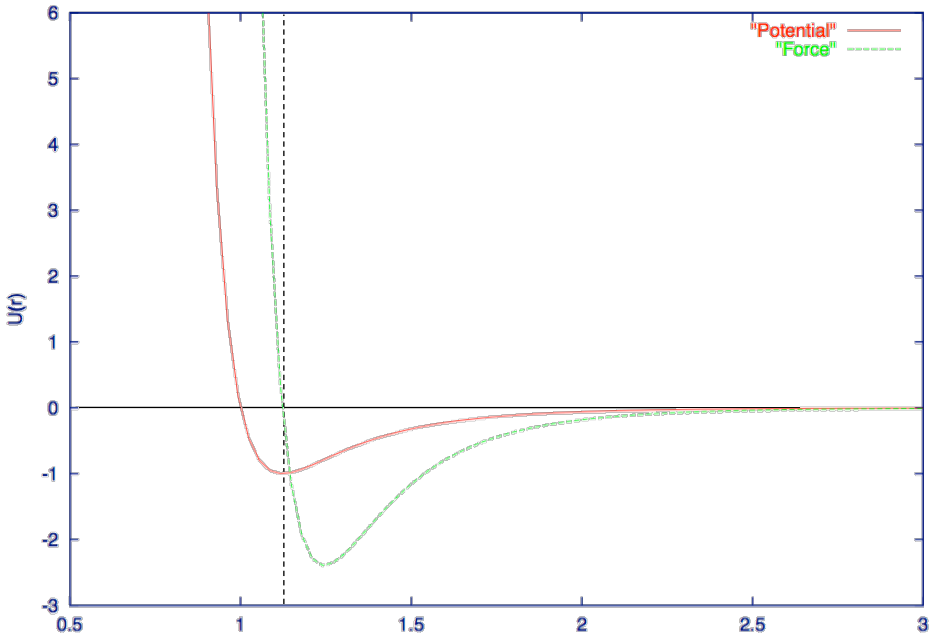
$$\mathbf{F} = \varepsilon \cdot \left(c \left(\frac{\sigma}{d} \right)^m - \left(\frac{\sigma}{d} \right)^n \right) \cdot \frac{\mathbf{x}_1 - \mathbf{x}_2}{\|\mathbf{x}_1 - \mathbf{x}_2\|}$$

wobei

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

und

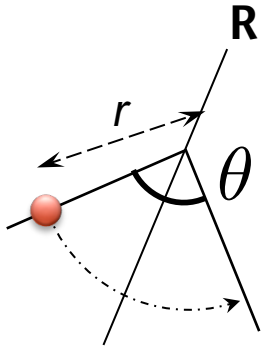
ε , c, m, n (für unsere Zwecke) beliebige Konstanten sind



Nicht-physikalische Effekte

- Strudel (*vortex*): rotiere Position eines Partikels um Achse **R** und Winkel

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



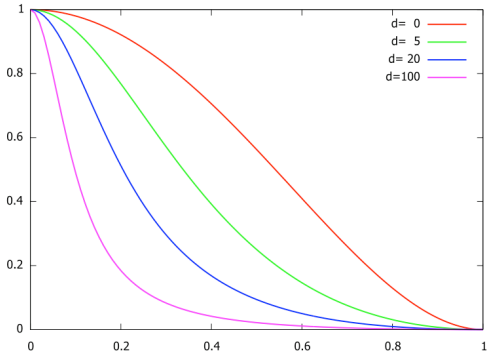
wobei a = "Stärke" des Vortex,

r = Abstand Partikel – Achse, und

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

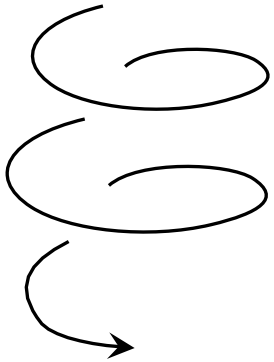
oder

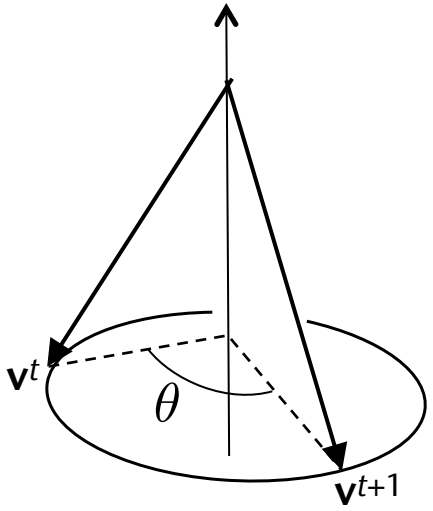
$$f(r) = \begin{cases} \frac{r^4 - 2r^2 + 1}{1 + dr^2} & , \; r \leq 1 \\ 0 & , \; r > 1 \end{cases}$$



- Erweiterungen:
  - Masse des Partikels einbeziehen
  - B-Spline als Achse des Vortex (für Tornados z.B.)
  - Achse des Vortex animieren

- Winkel:
  - Oftmals bewegt sich jedes einzelne Partikel auf einer spiralförmigen Bahn (z.B. in Feuer, oder Schneeflocken)
  - Idee:
    Rotiere $\mathbf{v}$ um eine Achse mit Winkel
    $$\theta = \sigma \cdot \Delta t$$
  - σ kann wieder leicht zufällig variieren, ebenso die Achse
  - Die Achse und σ können über die Zeit animiert werden

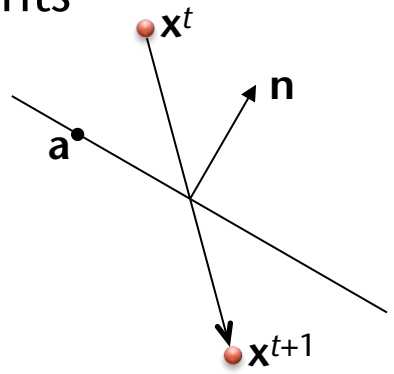




Kollisionen

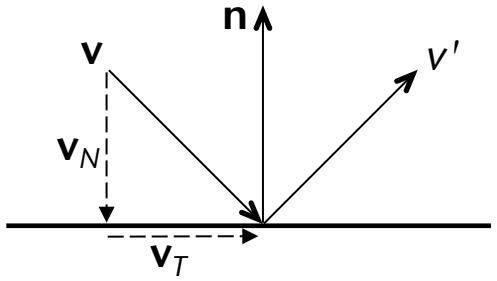
- Die wichtigste Form von geometrischen Constraints
- Zunächst: Kollision mit einer Ebene
- Test:

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



- Koll.-Behandlung: reflektiere $\mathbf{v}$

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



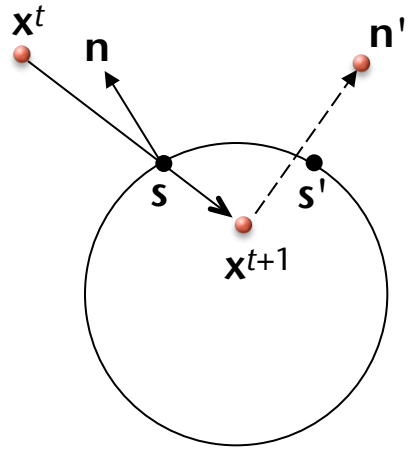
- Erweiterung um Reibung und elastischer/inelastischer Stoß:

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

mit μ = Reibungskoeffizient (friction parameter) und

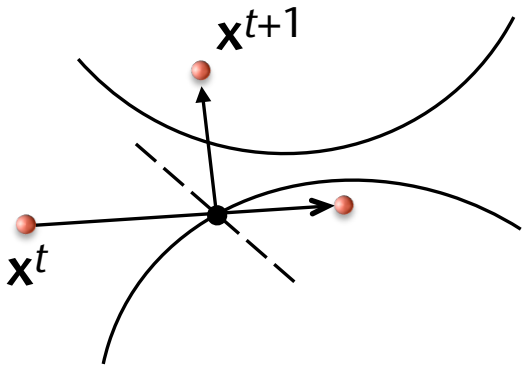
ε = Federung / Elastizität (resilience)

- Fazit: Kollisionserkennung für Partikel = "Punkt-in-Geometrie-Test" bzw. Schnitttest zwischen Geradensegment und Geometrie
- Analog für Kugeln:
  - Exakten Schnittpunkt **s** und Normale **n** bestimmen
  - Dann weiter wie eben

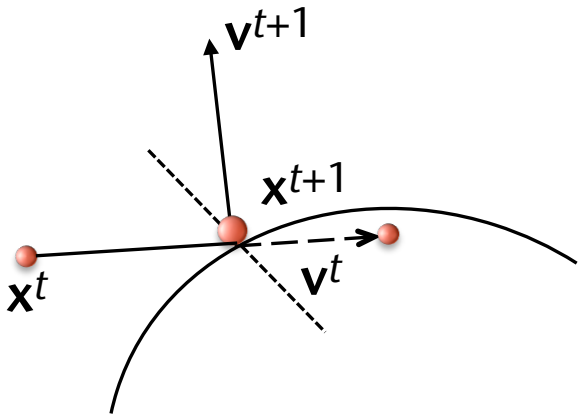


- Für Polyeder und implizite Flächen: siehe CG1
- Für Höhenfeld (Terrain): siehe CG2 (Raytracing)

- Achtung: stelle konsistenten Zustand nach der Kollisionsbehandlung her!
  - Problem: "Doppelkollisionen" an engen Stellen
  - Beispiel:



  - Korrekte Behandlung:



  - Gibt noch weitere Möglichkeiten

Karl Sims: *Particle Dreams*

Hierarchische Partikelsysteme

- Idee:
  - Ein Partikel ist seinerseits wieder ein Partikelsystem
  - Transformation des Vater-”Partikels” wirkt sich auf dessen Kind-Partikel aus (analog zu Scenengraph)

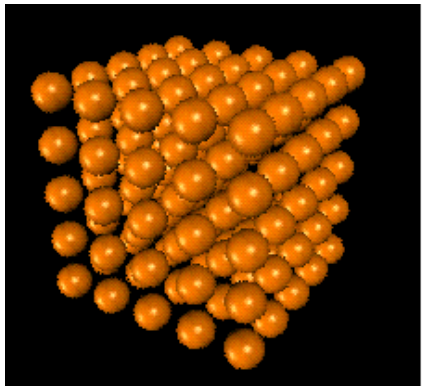
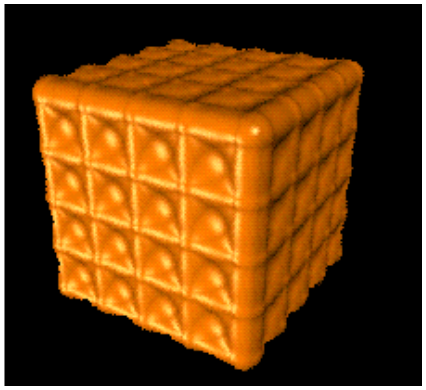
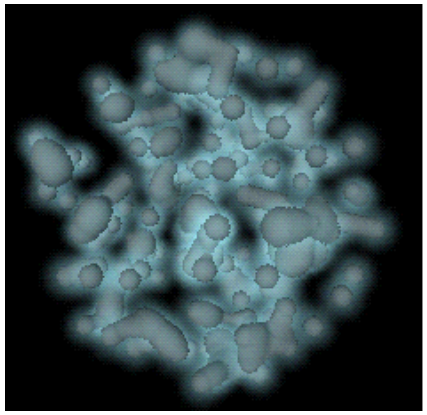
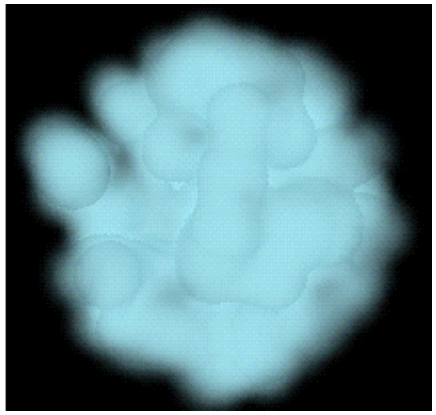
- Second-Order-Partikelsysteme:
  - Auch alle Kräfte werden durch Partikel repräsentiert
  - Diese können wechselwirken, werden an Partikelquellen geboren, sterben, etc.

Rendering

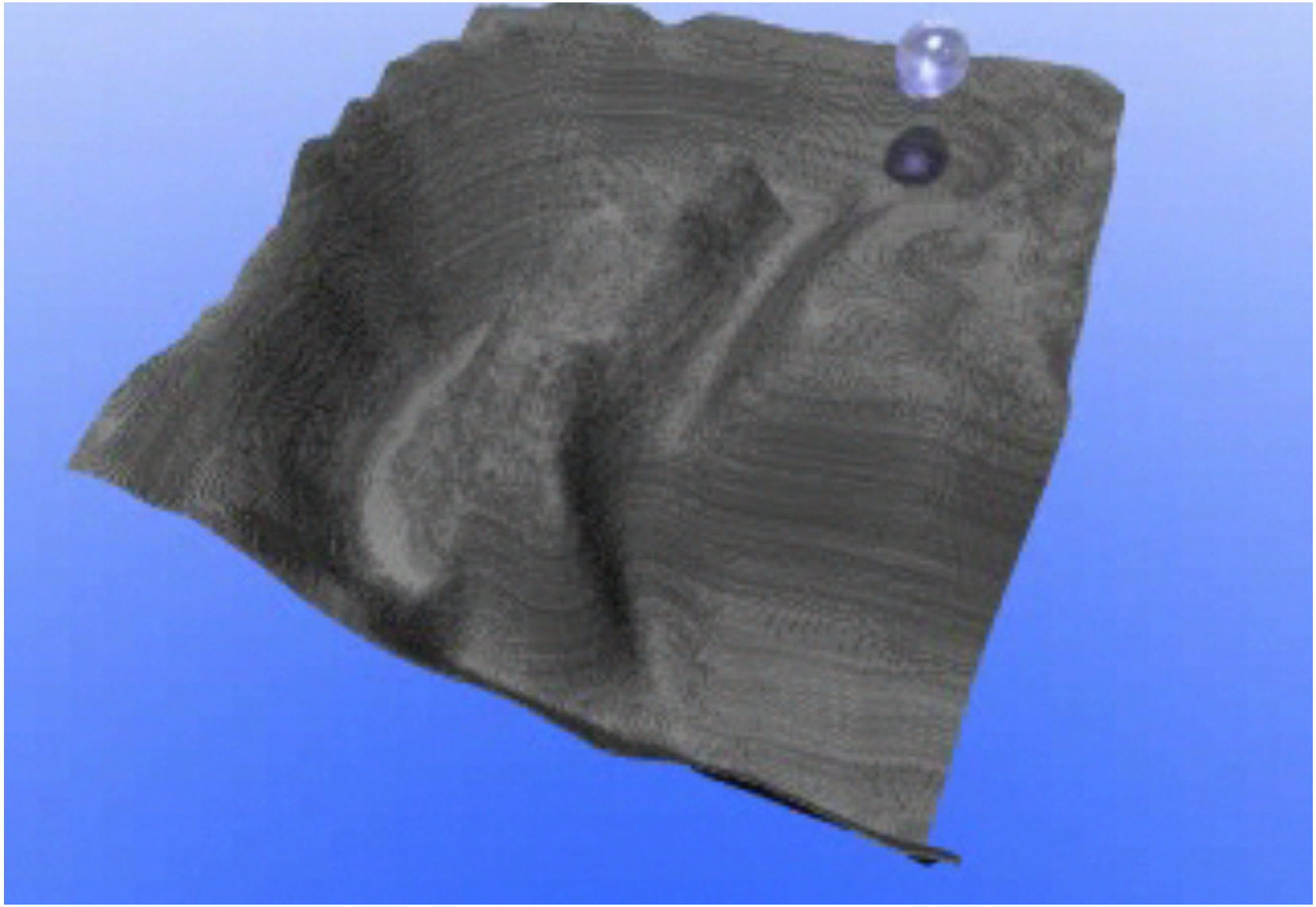
- Es gibt kein Standardverfahren
- Häufig:
  - Partikel als kleine Kreisscheibe (Splat, Sprite, Billboard) rendern
  - Meist mit Transparenz, die zum Rand abfällt
  - Benötigt Alpha-Blending!
- Alternative:
  - Farbe aller Partikel im Framebuffer akkumulieren (z.B. für Feuer)
  - Benötigt ca. 10 Partikel/Pixel

Rendering von "blobby objects"

- Betrachte Partikel als Metaballs
  - Aus CG 2: Metaballs = spheres that blend together to form (implicit) surfaces
  - Rendering mittels Ray-Casting
  - Entweder: Nullstelle der impliziten Fläche suchen
  - Oder: "Dichte" entlang des gesamten Strahls aufsummieren und als Opazität (*opacity*) oder Leuchtdichte interpretieren

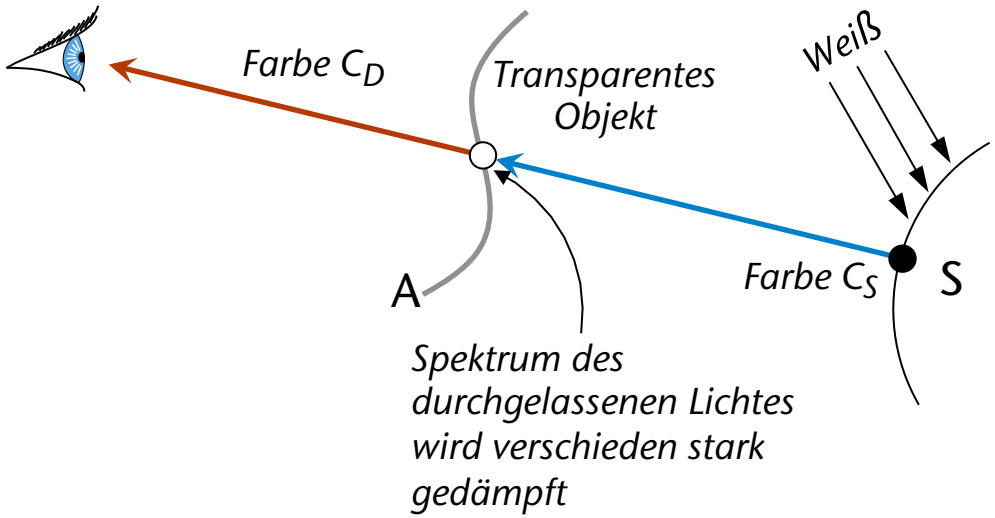





Beispiel



Rendering von transparenten Objekten

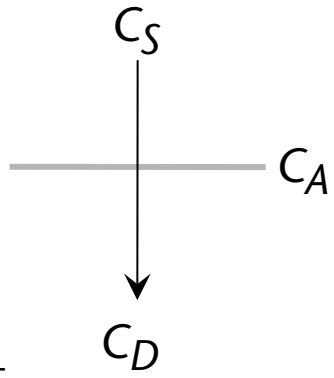
- Transparenz ≈ Licht wird von einem Material teilweise durchgelassen, wobei verschiedene Wellenlängen verschieden stark gedämpft werden



  - Extremfall: Farbfilter

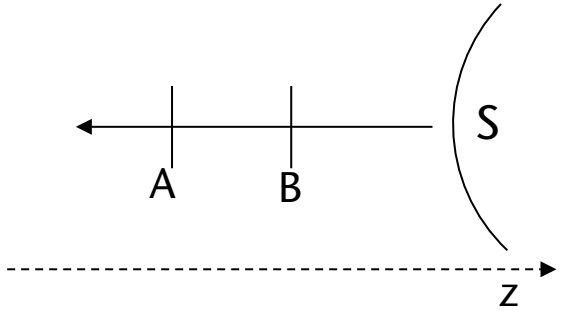
- Approximation: Alpha-Blending
  - $\alpha \in [0, 1]$ = Transparenz / Opacity
    - $\alpha = 0$ → komplett durchsichtig,
      $\alpha = 1$ → komplett opak (opaque)
  - Objekt A bekommt eine transparente "Farbe" C_A
  - Resultat:

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



  - α wird als 4-te Komponente in Farbvektoren gespeichert
- Beim Rendern führt die Graphikkarte folgende Operationen aus:
  - Color aus Framebuffer lesen → C_S
  - Formel auswerten → C_D
  - C_D in Framebuffer schreiben

- Achtung bei mehreren transparenten Objekten hintereinander!
  - Erst *A*, dann *B* → *B* wird durch z-Test gekillt
  - Naive Idee: Z-Buffer abschalten
  - Erst *A* dann *B* (ohne z-Test) ergibt:



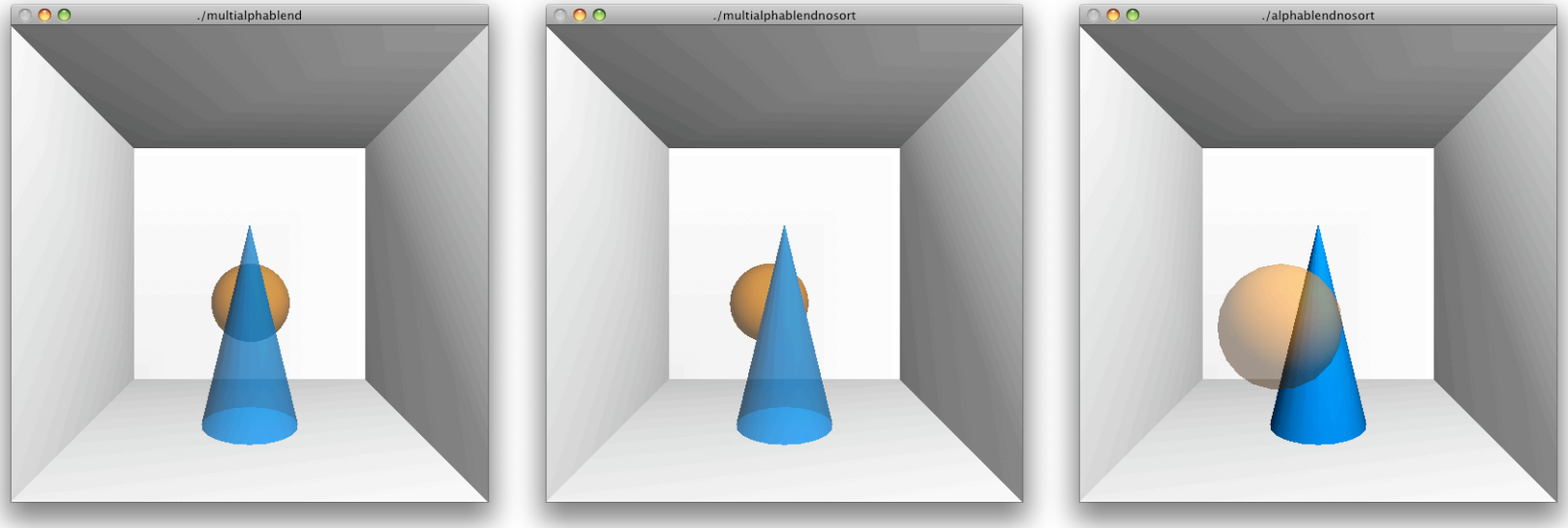
$$C'_D = \alpha_A C_A + (1 - \alpha_A)\, C_S$$
$$C_D = \alpha_B C_B + (1 - \alpha_B)\, C'_D$$
$$= \alpha_B C_B + (1 - \alpha_B)\, \alpha_A C_A + (1 - \alpha_B)\,(1 - \alpha_A)\, C_S$$

  - Erst *B*, dann *A* (ohne z-Test) ergibt:

$$C'_D = \alpha_B C_B + (1 - \alpha_B)\, C_S$$
$$C_D = (1 - \alpha_A)\, \alpha_B C_B + \alpha_A C_A + (1 - \alpha_B)\,(1 - \alpha_A)\, C_S$$

- Fazit: man muss die Polygone/Partikel von hinten nach vorne rendern, selbst dann, wenn der Z-Buffer abgeschaltet wird!

- Beispiele:




```
% cd VR/demos/alphablending; ./multialphablend; ./multialphablendnosort; ./alphanosortblend
```

- In Open GL:
  - Einschalten mit:

```
glEnable( GL_BLEND );
```

  - Blending-Funktion festlegen:

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

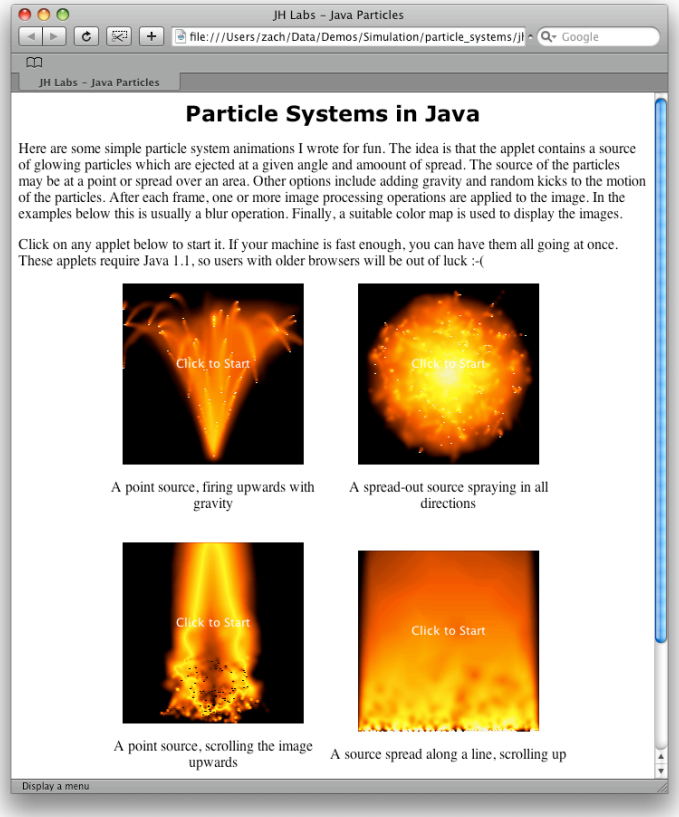
GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA →

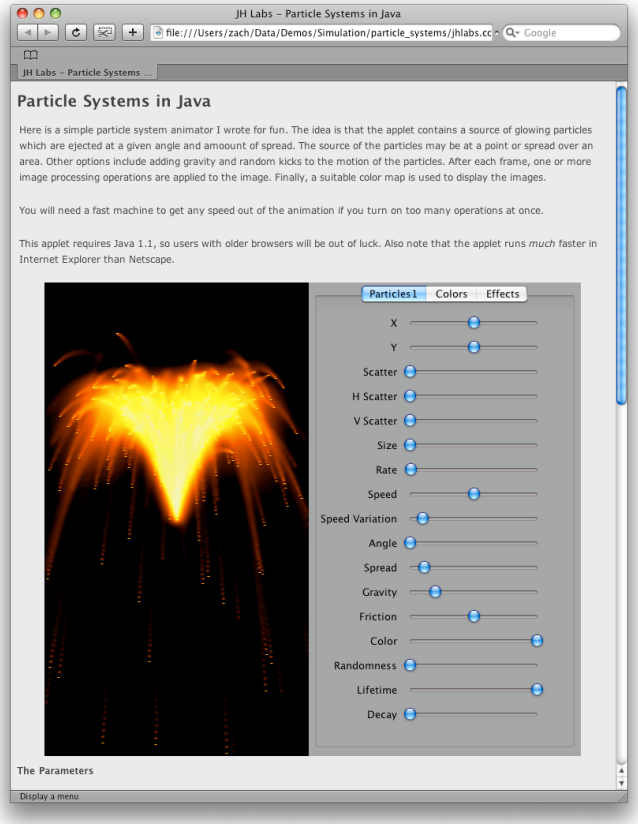
$$C'_D = \alpha C_A + (1 - \alpha)\, C_B$$

wobei C_D die Farbe aus dem Framebuffer ist;

  - Es gibt noch viele andere Varianten, z.B. kann man damit die Farben auch einfach aufakkumulieren (**GL_ONE, GL_ONE**)

Partikel-Systeme-Demos





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

Flammen [2002]

- Ziel:
  1. Glaubhaft aussehende Flammen
  2. Möglichst volle Kontrolle über die Flammen
- Das Modell:
  1. Einzelne Flammen (-elemente) durch parametrische Kurven modellieren → “spine” der Flamme
  2. Kontrollpunkte als Partikel simulieren
  3. Zylindrisches Profil um den Spine ergibt Oberfläche der Flamme (wo Oxidation = Verbrennen stattfindet)
  4. Der Raum in der Nähe der Oberfläche wird mit Partikeln gesampelt
  5. Rendern der Partikel (entweder volumetrisch, oder mit Alpha-Blending)
- Kontrollelemente:
  - Länge der Spines
  - Lebensdauer der Partikel
  - Intensität (=Anzahl Partikel), Quellen, Richtung, Wind, etc
  - Farbe und Größe der Partikel

- Generierung des *Spines*:
  - Spine-Partikel P im ersten Frame generieren
  - Dieses aufsteigen lassen (Auftrieb) und durch Wind bewegen:

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

    wobei

    w = Windfeld

    b = Auftrieb

    d = Diffusion = Rauschen;

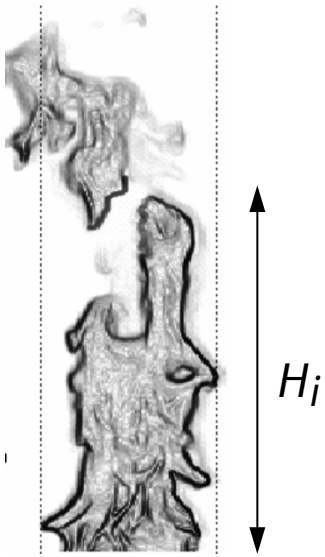
    T_P = Temperatur des Partikels = Alter

    (Vereinfachung hier: Partikel haben keine Masse)
  - In Folge-Frames weitere solche Partikel generieren, bis Max.-Anzahl für ein Spine erreicht
  - Spine-Partikel durch B-Spline verbinden

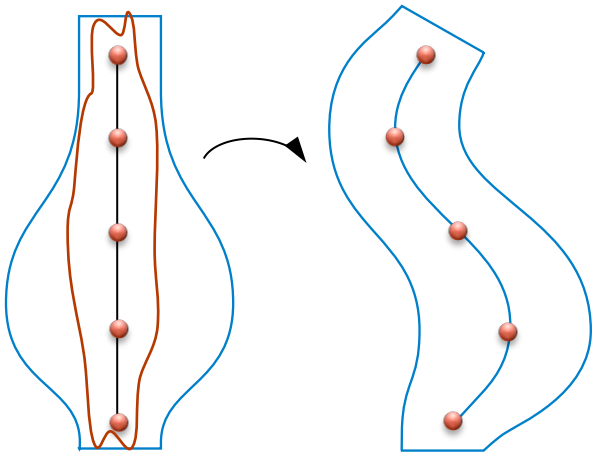
- Aufbrechen von Flammen-Elementen:
  - Das obere Stück des Spines wird zu einem zufälligen Zeitpunkt abgetrennt, wenn Höhe > H_i
  - Lebensdauer nach der Abtrennung:

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

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



- Das Profil der Flamme:
  - Rotationssymmetrisch um den Spine herum



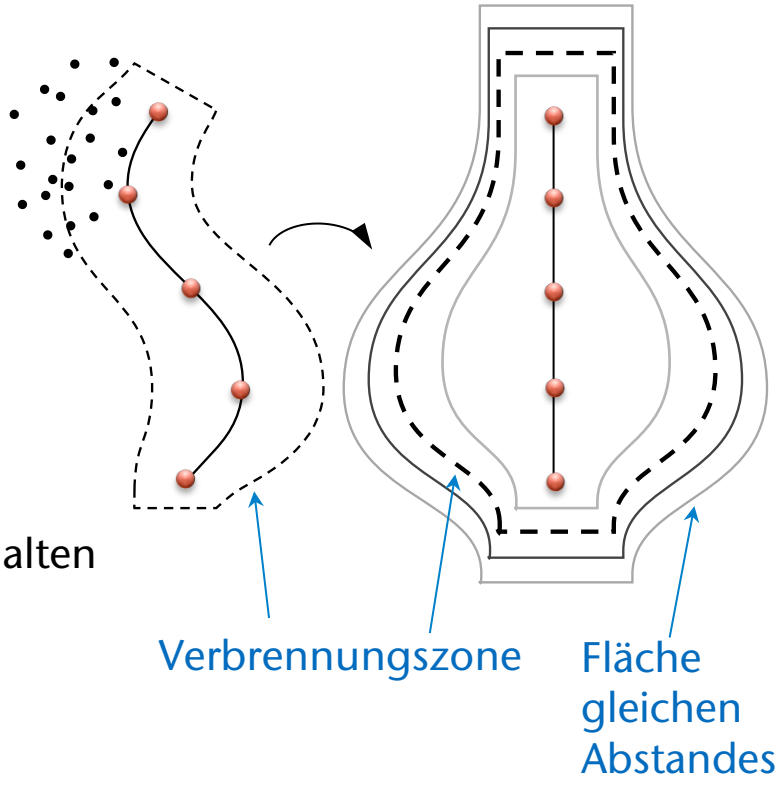
- Rendering:
  - Raum um die Flamme durch eine große Menge von Punkten sampeln gemäß der Dichtefunktion

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

wobei **x**' der Punkt auf der (deformierten) Profilfläche ist, der zu **x** am nächsten ist:

- Zufälliges **x** erzeugen
- Transformieren in Modellraum
- **x**' bestimmen
- *D* auswerten
- Falls $D(\mathbf{x})$ > Zufallszahl → Sample **x** behalten

  - Lege Referenzfoto als Textur auf die Profilfläche → Basisfarbe für **x**



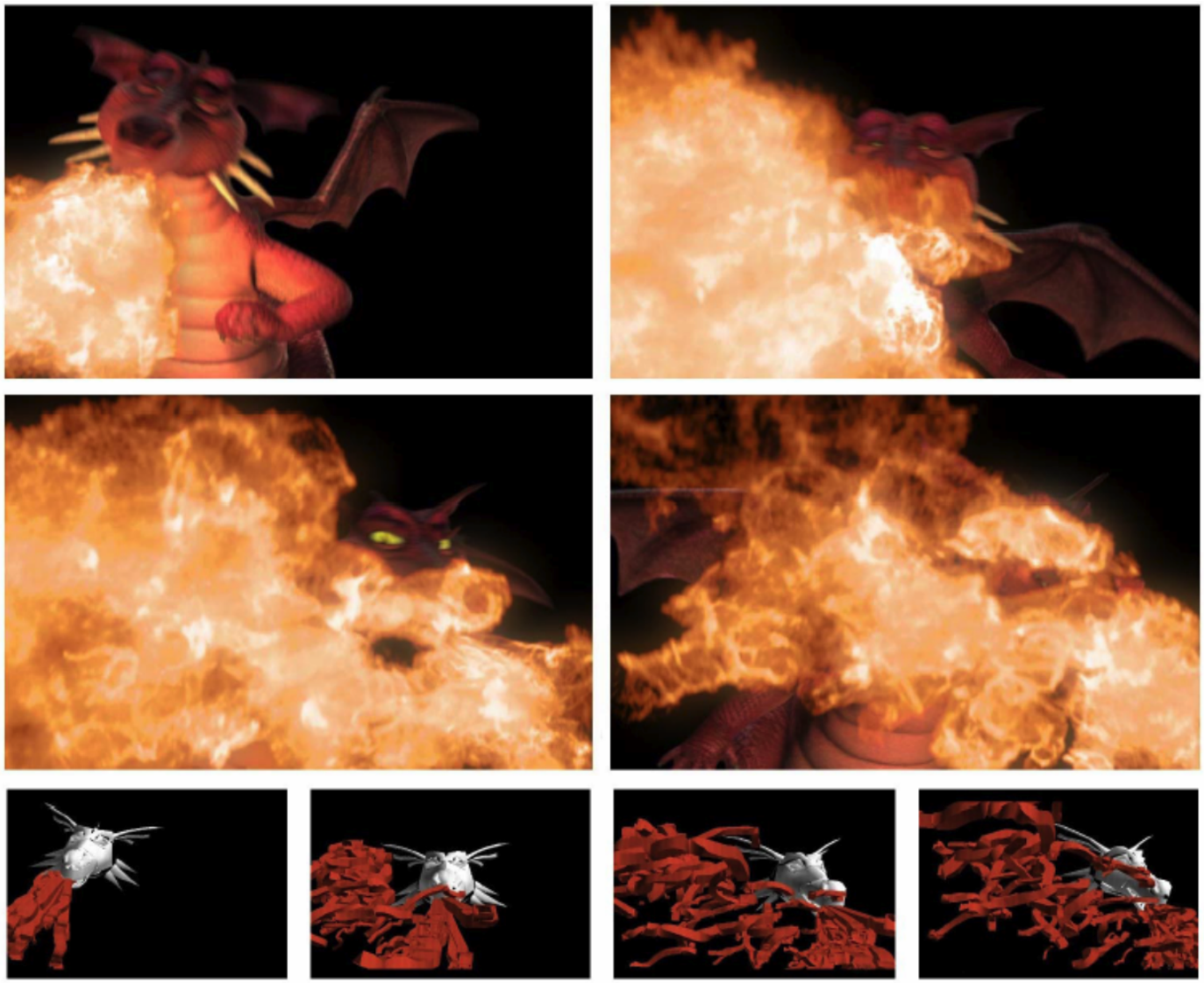
- Helligkeit eines Samples an Position **x**:

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

  wobei k = Faktor zur Kontrolle, n = Anzahl Samples

- Anzahl: ca. 10 Samples pro Pixel, ca 10,000 Samples pro Flamme
- Samples im Inneren von anderen Objekten werden verworfen
- Rauch: Samples > "Rauchhöhe" werden grau/schwarz gerendert

Results



Arnauld Lamorlette and Nick Foster, PDI/DreamWorks

Exkurs: Procedural Modeling of Plants with Particles

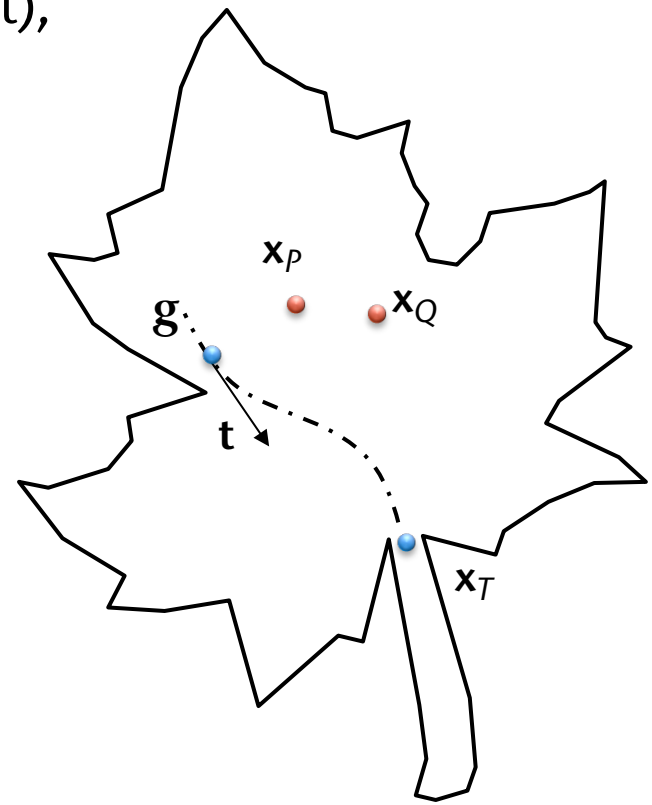
- Idee: verwende Partikel, um den Transport von Flüssigkeit in einem Blatt zu simulieren
  - Bahnen der Partikel ergeben die Adern
- Axiome:
  1. Die Natur versucht, die Länge der Bahnen zu minimieren
     → Partikel versuchen, sich auf gemeinsamen Bahnen zu bewegen
  2. Es geht keine Flüssigkeit verloren oder kommt hinzu
     → Wenn 2 Partikel eine gemeinsame Bahn verfolgen, muss die Ader dort doppelt so dick sein
  3. Alle Bahnen gehen vom Blattstiel aus

- Übersicht des Algorithmus:

```
plaziere Partikel zufällig auf der Oberfläche des Blattes
loop bis kein Partikel übrig:
    bewege jedes Partikel in Richtung seines nächsten
        Nachbarn oder der nächsten schon existierenden Bahn,
        und in Richtung des Blattstiels
    falls Partikel bei Blattstiel angekommen ist:
        lösche dieses Partikel
    falls zwei Partikel einander "nahe genug" sind:
        verschmelze beide zu einem Partikel
```

Zur Bewegung der Partikel

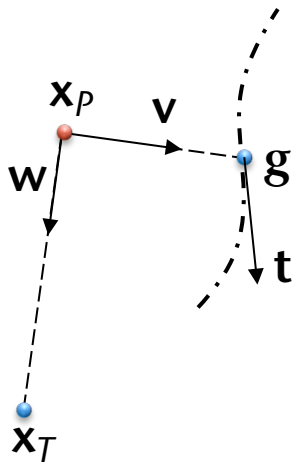
- Seien
  - $\mathbf{x}_P$ = aktuelle Position des Partikels P,
  - $\mathbf{x}_T$ = Position des Ziels (Blattstiel),
  - $\mathbf{g}$ = nächster Punkt zu $\mathbf{x}_P$ auf einer Bahn,
  - $\mathbf{t}$ = Tangente in $\mathbf{g}$ (normiert),
  - $\mathbf{x}_Q$ = nächstes Partikel zu P



- Falls $\|\mathbf{x}_P - \mathbf{g}\| < \|\mathbf{x}_P - \mathbf{x}_Q\|$:
  - Setze:

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

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



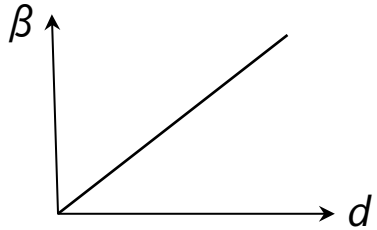
  - Neue Position:

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

wobei

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

- Ein (ungefähr) lineares β ergibt z.B. Partikelbahnen, die in der Nähe der bestehenden Bahn tangential dazu verlaufen, weiter weg senkrecht darauf zu

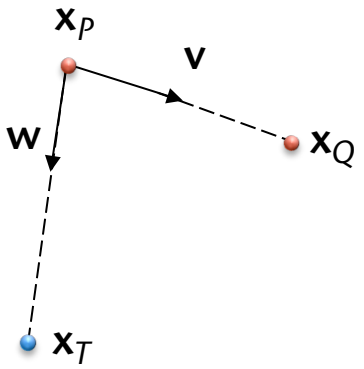


- Sonst ($||\mathbf{x}_P - \mathbf{x}_Q|| < ||\mathbf{x}_P - \mathbf{g}||$):
  - Setze

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

  - Neue Position:

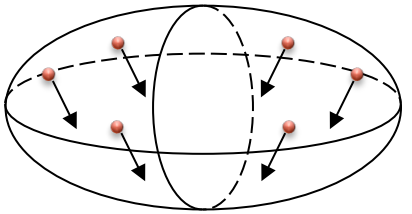
$$\mathbf{x}'_P = \mathbf{x}_P + \gamma\mathbf{v} + (1-\gamma)\mathbf{w}$$

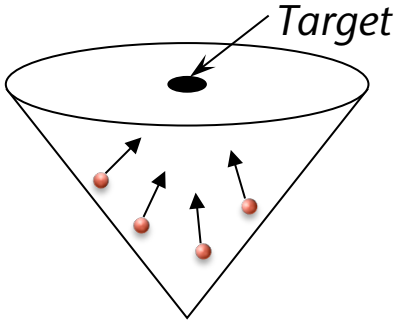


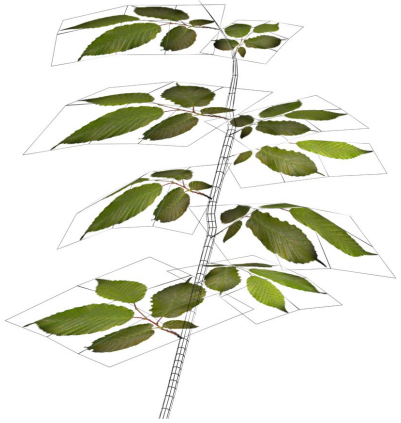
- Dicke der Adern:
  - Jedes Partikel hat Größe = Betrag der Querschnittsfläche der Ader
  - Zu Beginn: alle Partikel haben Einheitsgröße
  - Bei Verschmelzen: Größen addieren
  - Bei Auftreffen auf bestehende Bahn: Größe des Partikels zu Größe des Querschnitts der Bahn ab dem Auftreffpunkt bis zum Ziel dazuaddieren

Modellierung von Bäumen

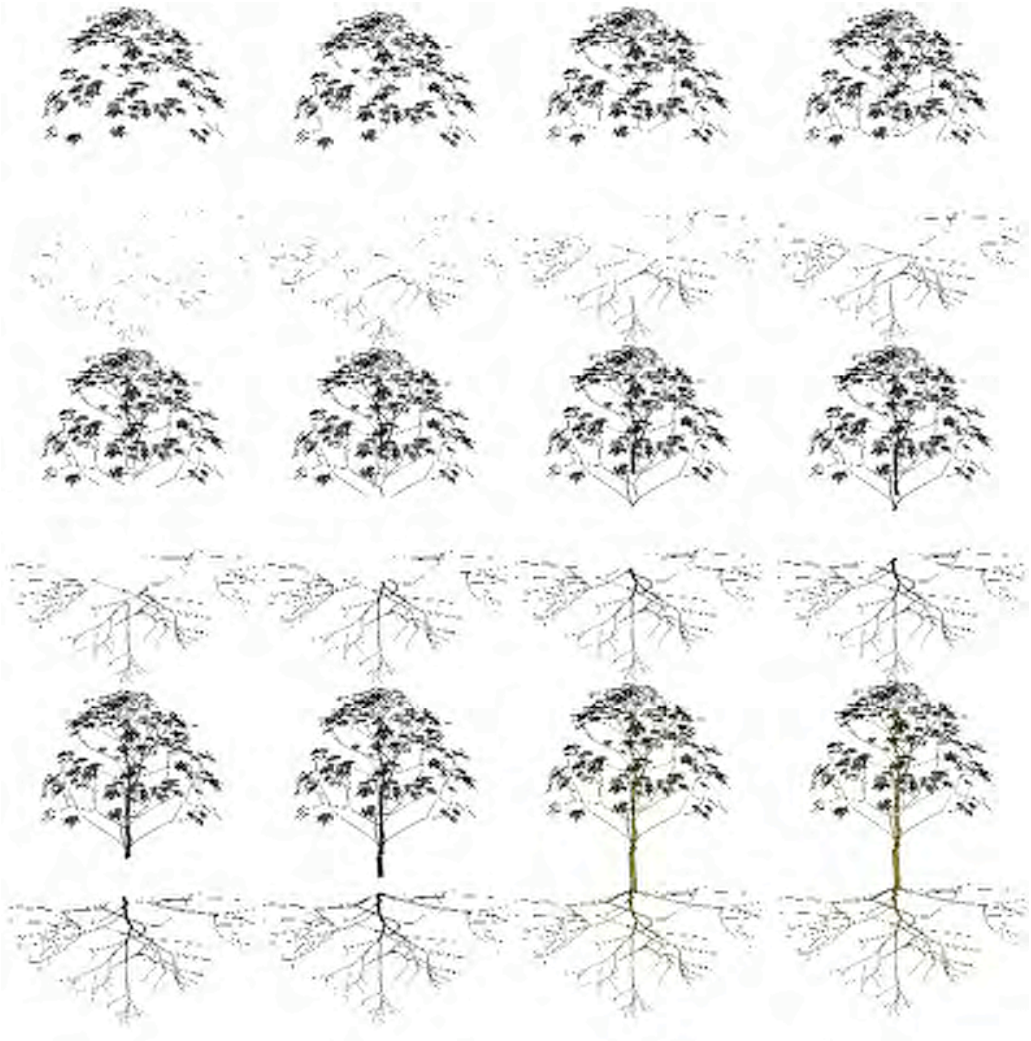
- Funktioniert genau gleich
- Vorgabe: Geometrie für die initialen Positionen der Partikel
  - Nur Hüllgeometrie
  - Erzeuge Partikel darin mittels stochastischem Prozeß
- Geometrie der Zweige: verbinde Kreisscheiben, die senkrecht zur Bahn entlang der Bahn plaziert werden
  - "sweep a disk along the path"
- Zweig-Primitive an die Äste setzen:





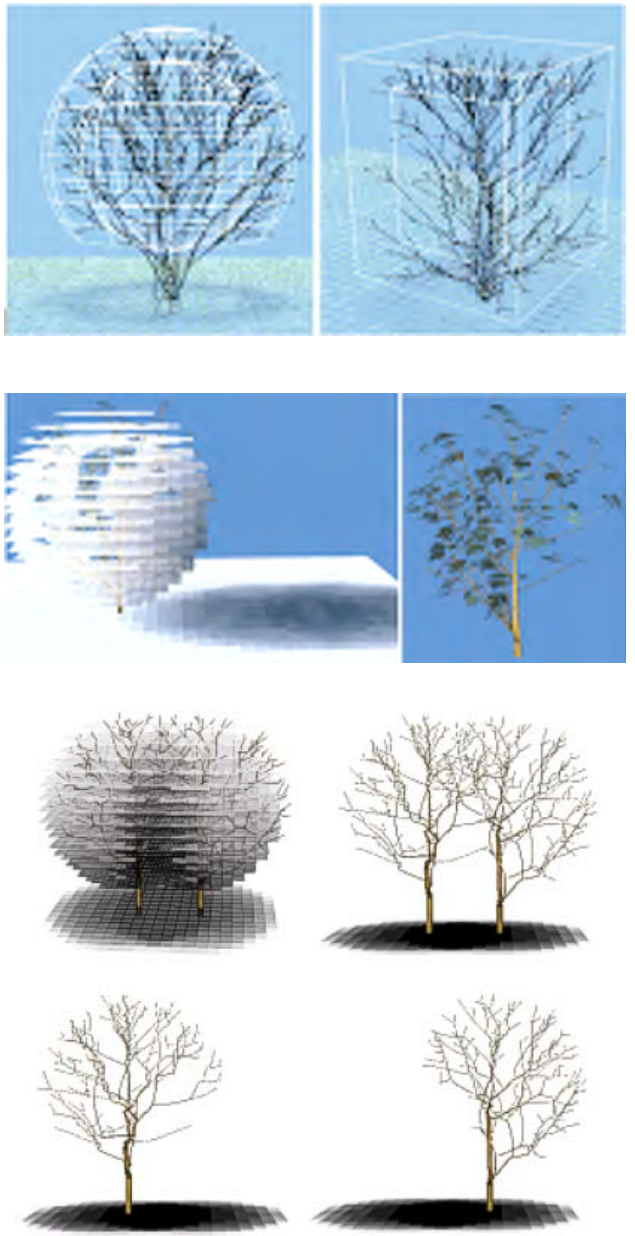


- Beispiel-Ablauf:

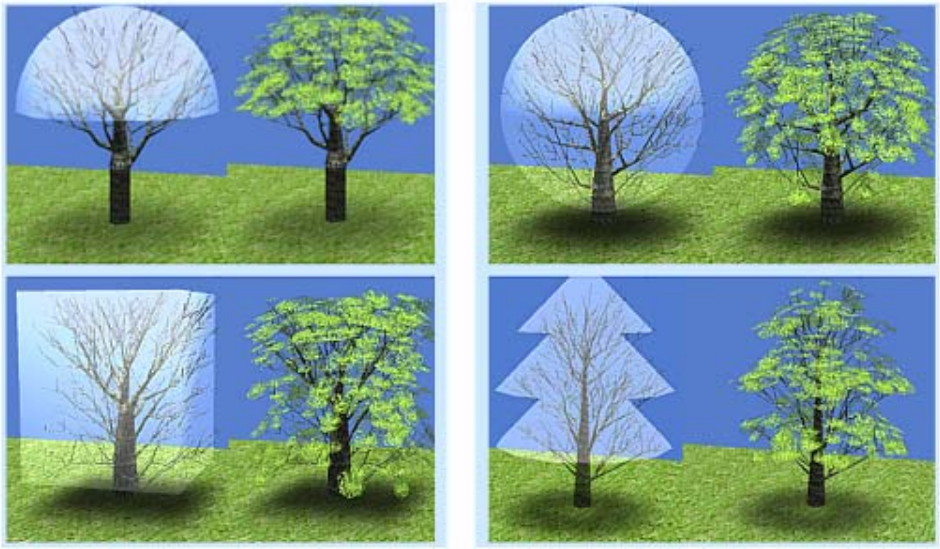


Berücksichtigung der Lichtverhältnisse

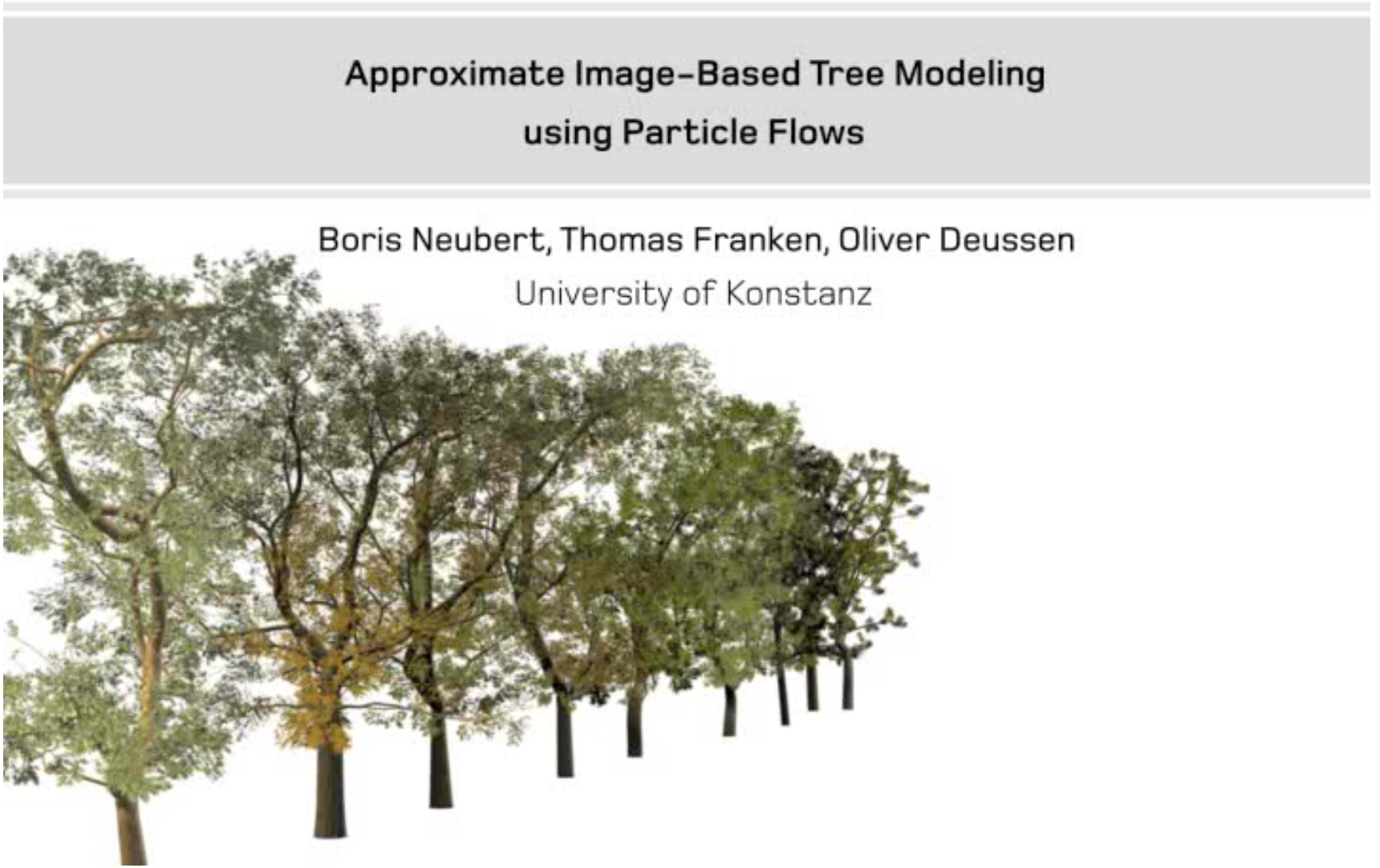
- Beobachtung: Stellen mit weniger Licht haben weniger Zweige / Blätter
- Lässt sich relativ einfach modellieren:
  - Lege den Baum in ein Gitter
  - Approximiere die (noch nicht existierende) Blätterschicht durch eine Kugel- oder Würfelschale
  - Berechne Lichteinfall für jeden Gitterknoten durch die Schale hindurch (ray casting)
  - Bei der Partikelerzeugung: passe Wahrscheinlichkeit einer Erzeugung dem Lichteinfall an (trilinear interpolieren)



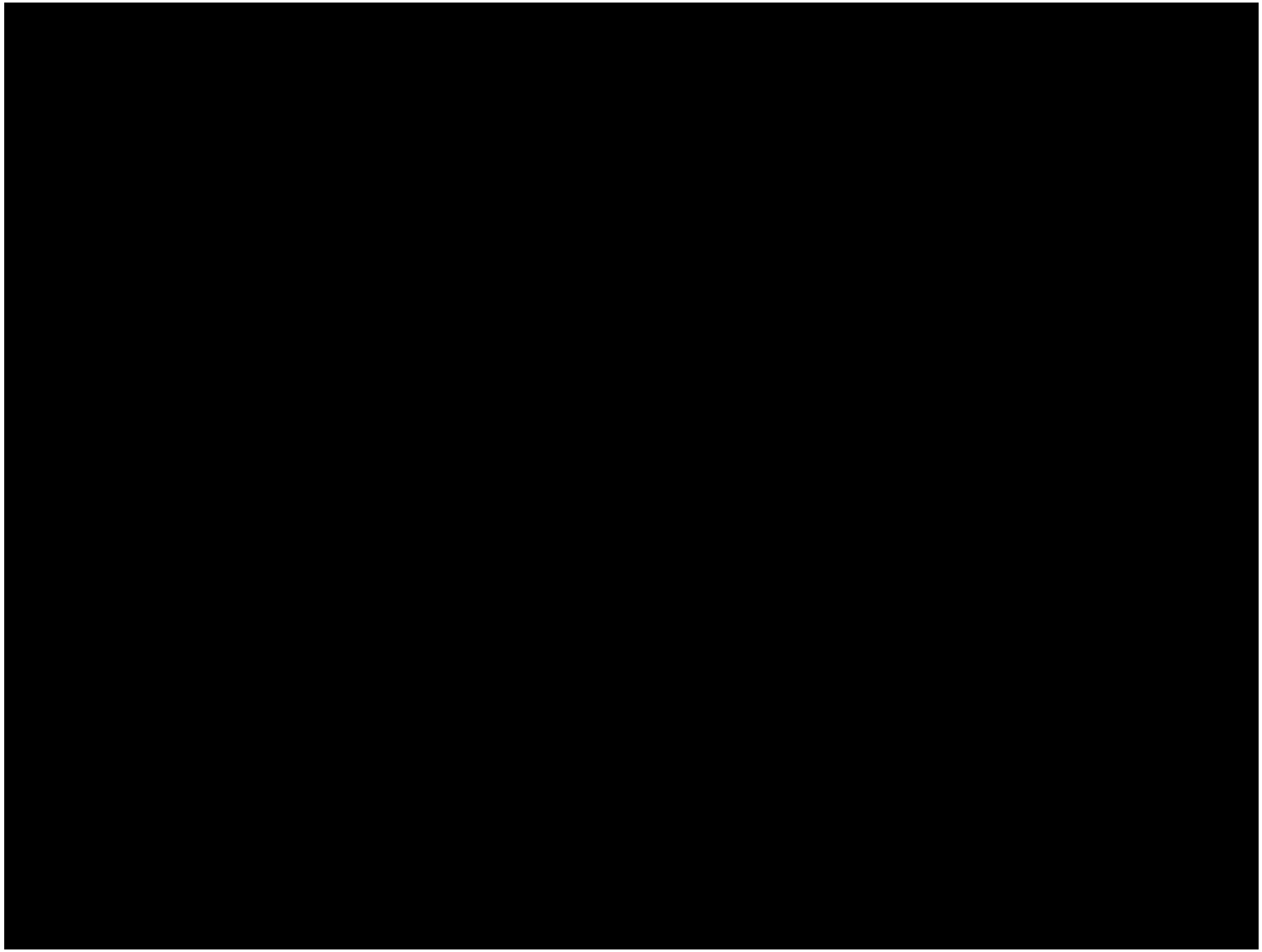
Beispiele



Videos


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

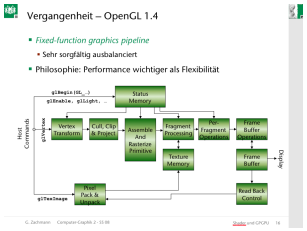
Historischer Video

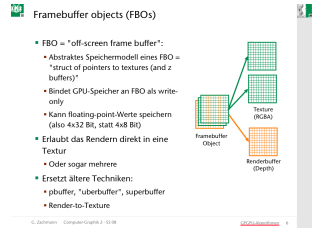


Andre and Wally B (Pixar)

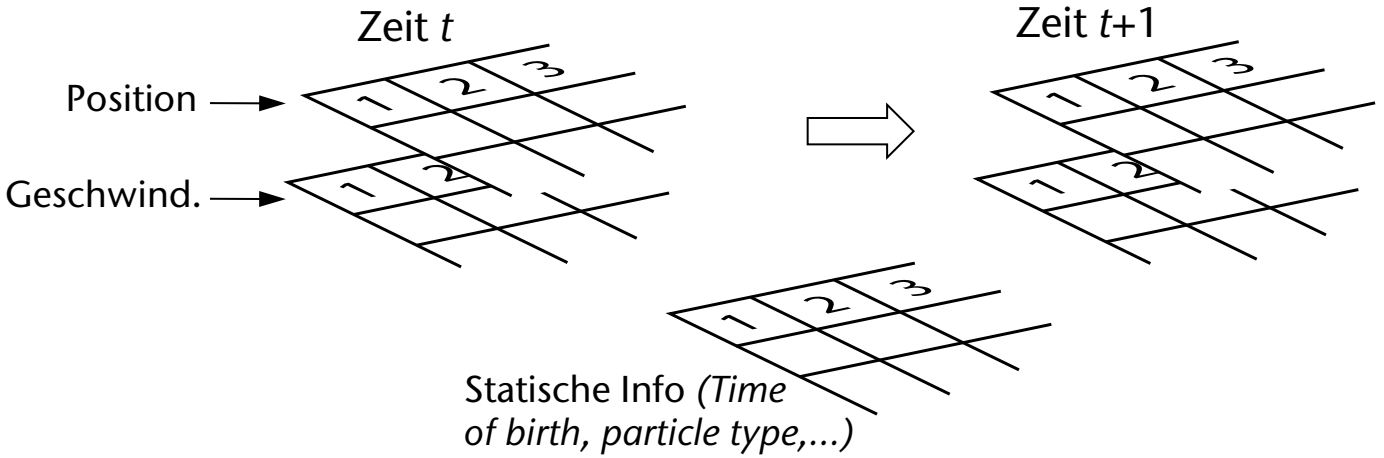
Massiv-parallele Simulation

- Exkurs / Erinnerung: die GPU als massiv-parallele general-purpose Architektur





- Speicherung der Daten in Texturen:

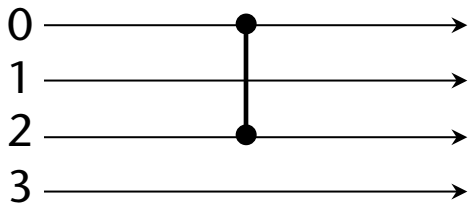


  - Verwende 2D-Textur, da so mehr Partikel gespeichert werden können (reines Impl.-Detail)
    - Indizes nach 2D-Indizes umrechnen, oder gleich mit 2D-Indizes arbeiten

- Verwaltung freier Speicherplätze (memory management):
  - Wenn Partikel stirbt, trage Textur-Index in Liste ein
  - Bei Partikel-Generierung: hole freie Indizes aus Liste
  - Eventuell besser: Queue statt Liste, sortiert nach Index
    - Vorteil: keine Fragmentierung (keine "Löcher")
    - Nachteil: man kann nicht en bloc/parallel neue Partikel generieren und allozieren

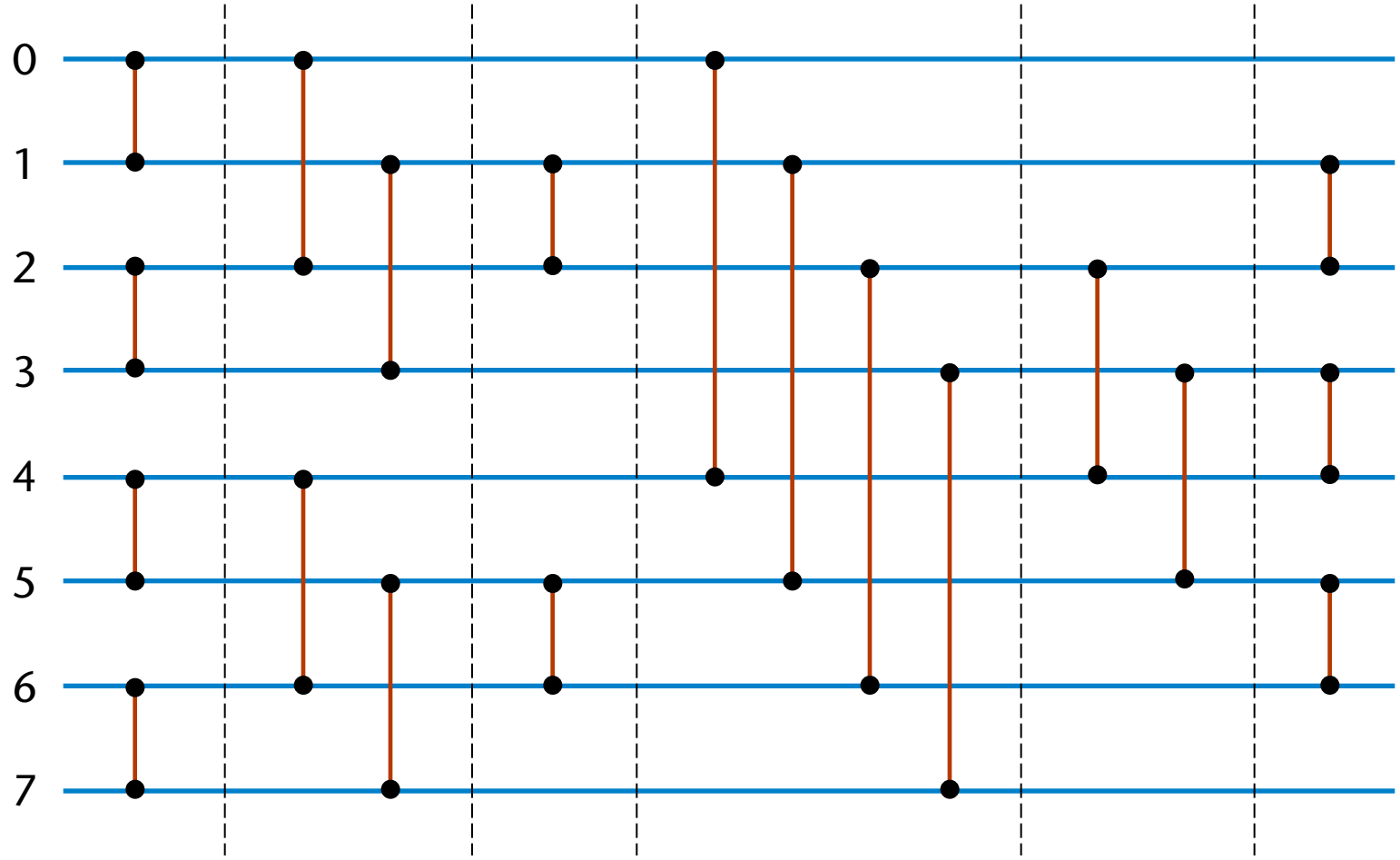
Paralleles Sortieren

- Erinnerung: Sortierung wird für Alpha-Blending benötigt
- Lösung: Sortiernetzwerke
- Informelle Definition:
  - Bestehen aus einer Menge von "Leitungen"
  - Daten D_i laufen von links nach rechts durch die Leitungen i
  - Zwei Leitungen können vertikal durch einen Komparator verbunden werden
  - Falls $D_i > D_j \wedge i < j$, dann werden die beiden Daten durch den Komparator vertauscht



- Eigenschaft: Ein Sortiernetzwerk ist datenunabhängig, d.h., die Laufzeit ist unabhängig von der "Sortiertheit" der Eingabe!

Beispiel



Das 0-1-Prinzip

- Definition (*monoton*):
  Seien A, B zwei Mengen mit Ordnungsrelation,
  und $f : A \to B$ eine Abbildung.
  f heißt monoton genau dann, wenn
  $$\forall a_1 a_2 \in A : \; a_1 \leq a_2 \Rightarrow f(a_1) \leq f(a_2)$$

- Lemma:
  Sei $f : A \to B$ monoton. Dann gilt
  $$\forall a_1, a_2 \in A : \; f(\min(a_1, a_2)) = \min(f(a_1), f(a_2))$$
  Analoges gilt für max.

- Beweis:
  Fall 1: $a_1 \leq a_2 \Rightarrow f(a_1) \leq f(a_2)$
  $\min(a_1, a_2) = a_1$, $\min(f(a_1), f(a_2)) = f(a_1)$
  $f(\min(a_1, a_2)) = f(a_1) = \min(f(a_1), f(a_2))$
  Fall 2: $a_2 < a_1 \to$ analog

- Erweiterung von $f : A \rightarrow B$ auf Folgen über A bzw. B:

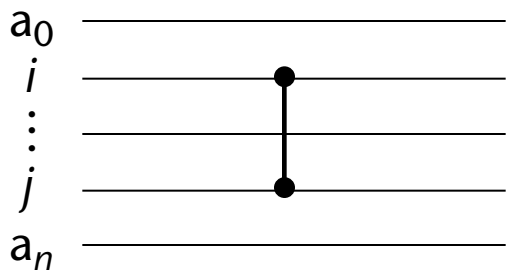
$$f\,(a_0, \ldots, a_n) = f(a_0), \ldots, f(a_n)$$

- Lemma:
  Sei f eine monotone Abbildung und $\mathcal{N}$ ein Komparatornetzwerk.
  Dann kommutieren $\mathcal{N}$ und f, d.h.

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

- Beweis:
  - Sei $a = (a_0, \ldots, a_n)$ eine Folge
  - Notation: schreibe einen Komparator zwischen Leitung i und j so:

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



  - Es gilt:

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

- Satz (0-1-Prinzip):
  Sei $\mathcal{N}$ ein Komparatornetzwerk.
  Falls $\mathcal{N}$ jede 0-1-Folge sortiert, dann sortiert es auch jede beliebige Folge!

- Beweis (durch Wiederspruch):
  - Annahme: Folge a wird nicht durch $\mathcal{N}$ sortiert
  - Dann ist $\mathcal{N}(a) = b$ nicht korrekt sortiert, d.h. $\exists k : b_k > b_{k+1}$
  - Definiere $f : A \rightarrow \{0,1\}$ wie folgt:

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

  - Nun gilt:

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

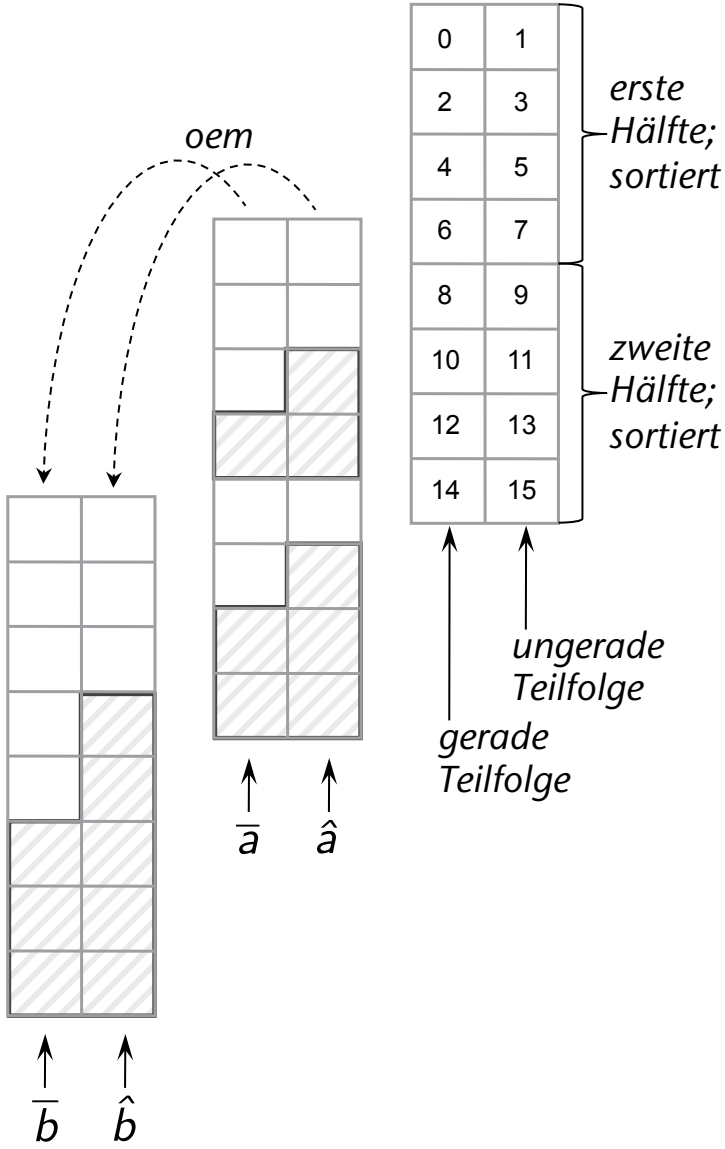
    wobei a' eine 0-1-Folge ist.
  - Aber: $f(b)$ ist unsortiert, denn $f(b_k) = 1$ und $f(b_{k+1}) = 0$
  - Also ist auch $\mathcal{N}(a')$ unsortiert, d.h., wir haben aus a eine 0-1-Folge konstruiert, die von $\mathcal{N}$ nicht sortiert wird.

Batcher's Odd-Even-Mergesort [1968]

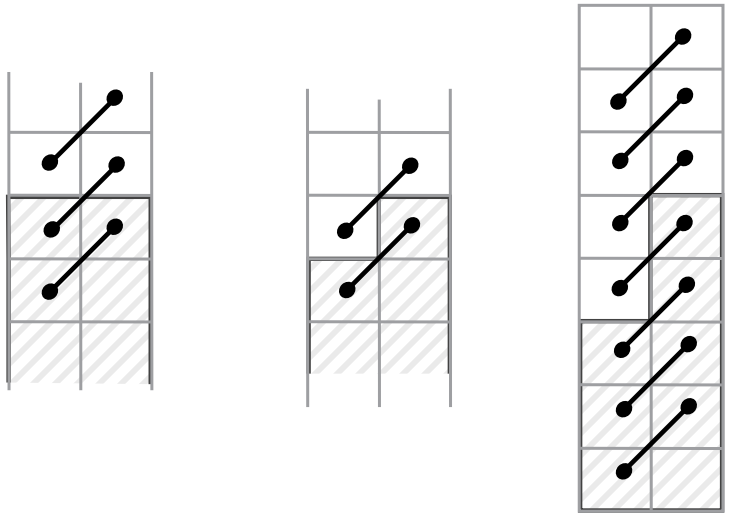
- Im Folgenden sei die Länge n einer Folge $a_0,\ldots,a_{n-1}$ immer eine Zweier-Potenz, d.h. $n = 2^k$
- Zunächst die Sub-Routine "odd-even merge":

```
oem( a_0,…,a_{n-1} ):
precondition:  a_0,…,a_{n/2-1}  und  a_{n/2},…,a_{n-1}  sind beide sortiert
postcondition:  a_0,…,a_{n-1}  ist sortiert
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)
```

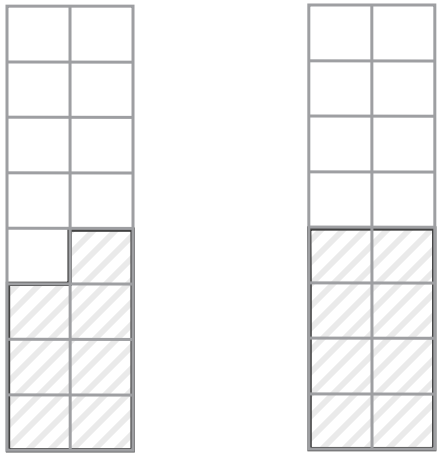
- Korrektheit:
  - Mittels Induktion und dem 0-1-Prinzip
  - Induktionsanfang: $n = 2$
  - Induktionsschritt: $n = 2^k$, $k > 1$
  - Betrachte 0-1-Folge $a_0,\ldots,a_{n-1}$
  - Schreibe diese in 2 Spalten
  - Markiere 0 = weiß, 1 = grau
  - Offensichtlich: ā und â bestehen beide aus zwei sortierten Hälften, d.h., Vorbedingung von *oem* ist erfüllt
  - Nach Zeile (*) haben wir diese Situation (die ungerade Teilfolge kann höchtens zwei 1-en mehr enthalten)



- Nach Zeile (**) werden diese Vergleiche vorgenommen und es gibt nur diese 3 Fälle:



- Danach ist in jedem Fall eine dieser beiden Situation hergestellt:
- D.h., die Ausgabefolge ist sortiert



- Fazit: jede 0-1-Folge wird sortiert
- Laufzeit : $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{2} - 1 \in O(n \log n)$

- Der Sortier-Algorithmus:

```
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})
```

- Laufzeit:

$$T(n) \in O\left(n \log^2 n\right)$$

Die Abbildung auf Shader (Stream Programming Model)

- Daten werden zu Beginn in einer Textur auf die GPU geladen
- Auf der CPU läuft folgendes Programm:

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

```
oem( n, step ):
if n = 2:
    oemEndShader ausführen
else:
    oem( n/2, step*2 )
    oemRekursionShader ausführen
```

- Mit dem Step-Parameter erreicht man eine Sortierung “in situ”

- Der Shader (stream kernel) für das Ende einer Rekursion:

```
oemEndShader( i, step ):
// are we on the even or the odd side?
if i/step is even:
    div = 1
else:
    div = -1
a0 ← SortData[i]              // SortData = Textur =
a1 ← SortData[ i+div+step ] // globales "Array"
if div > 0:
    output max(a0,a1)          // schreibe ins
else:
    output min(a0,a1)          // Ausgabe-Array
```

  - Der oemEndShader bildet Zeile (1) aus dem Algorithmus ab
  - Erinnerung: ein Shader wir für jeden Index *i* einmal (parallel) aufgerufen

- Der Shader für die Rekursion selbst:

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

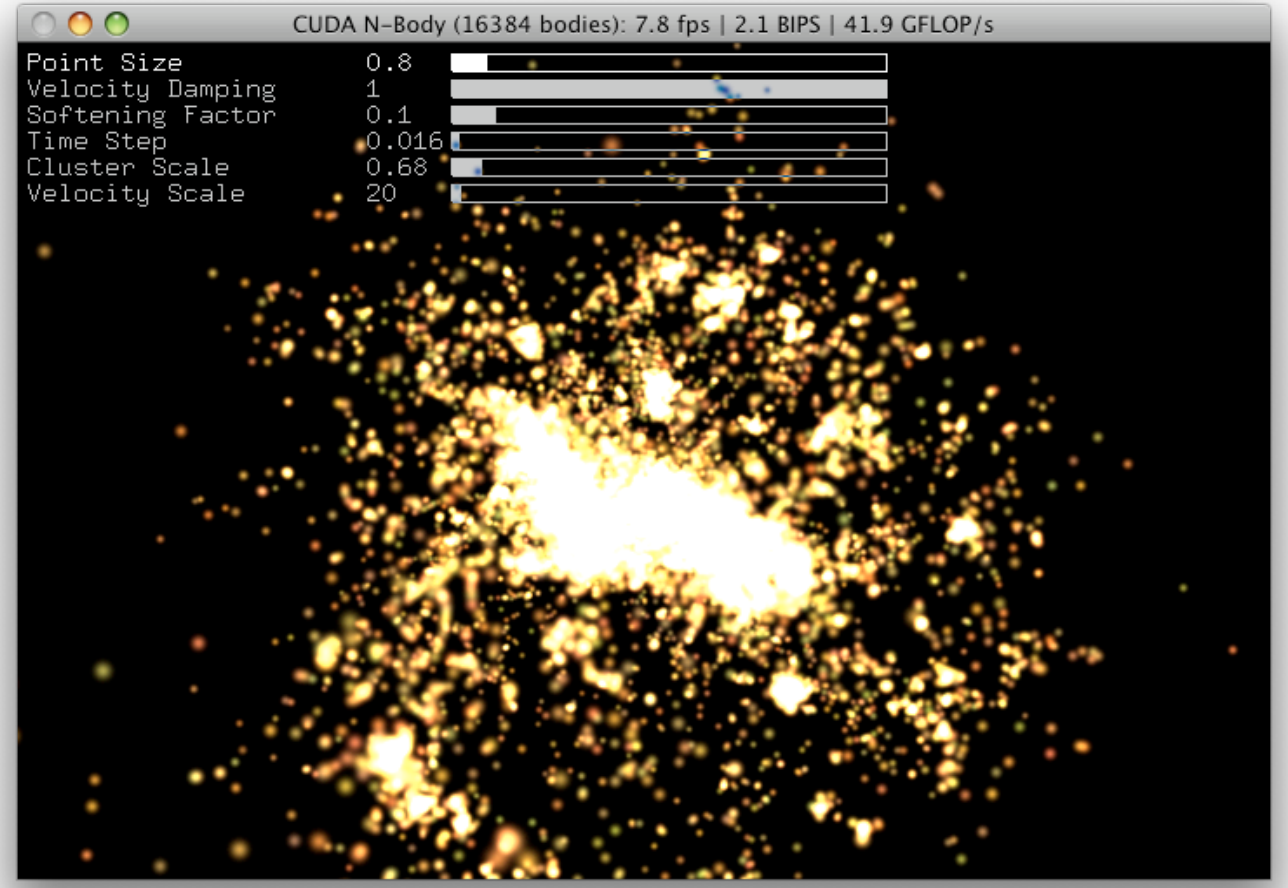
  - Der oemRecursionShader bildet die Zeile (2) aus dem Algorithmus ab

- Laufzeit:

$$\frac{1}{2}\log^2 n + \frac{1}{2}\log n \qquad \text{rendering passes}$$

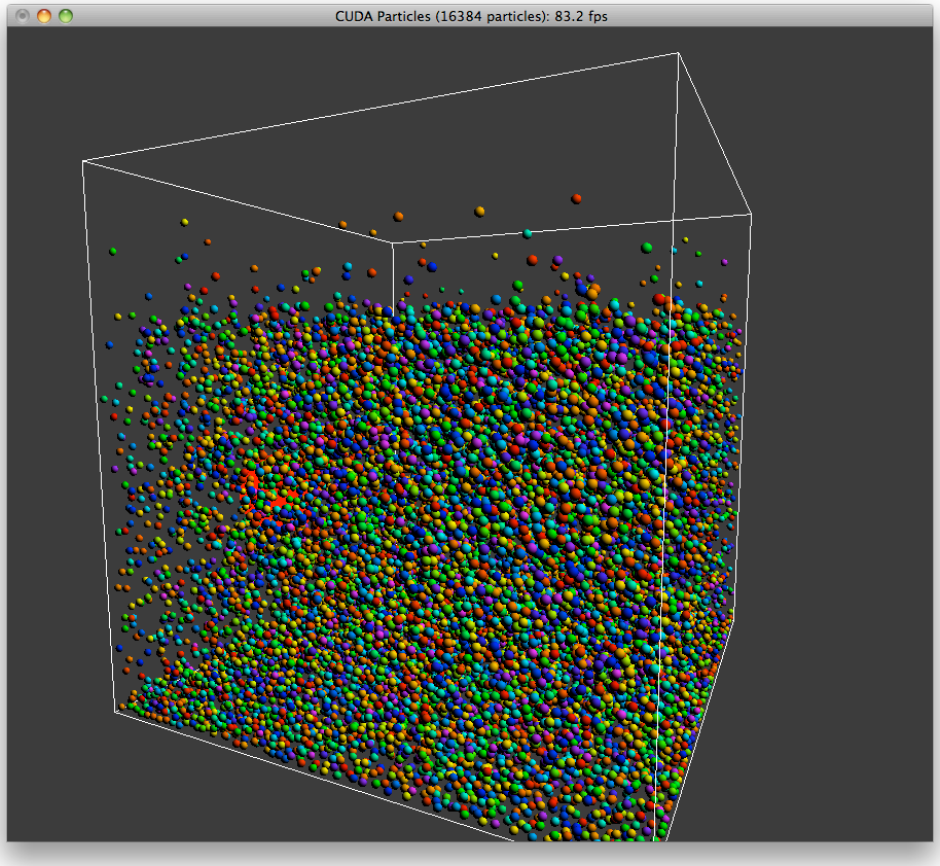
- Ergibt 210 Passes für 1024 x 1024 Partikel
  - Kann man inkrementell machen, also eine kleine Anzahl Sortier-Passes pro Frame

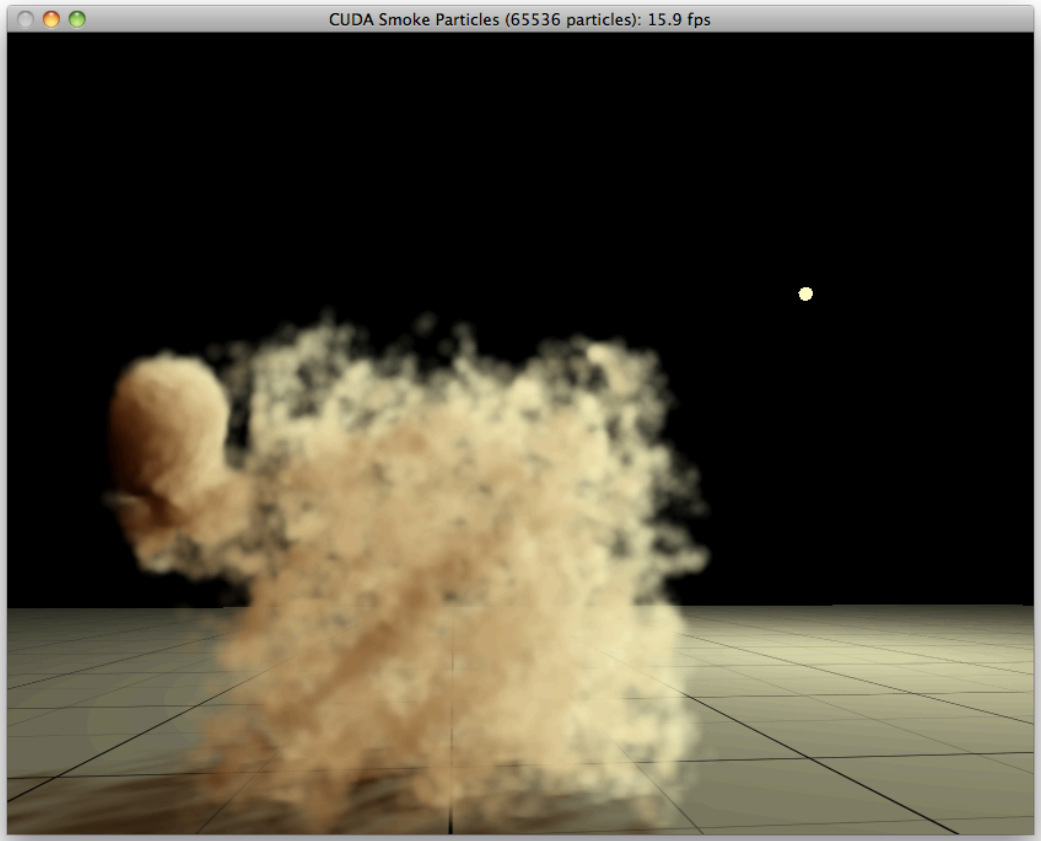
Demos



N-body simulation

http://www.nvidia.com/cuda


CUDA Particles (16384 particles): 83.2 fps


CUDA Smoke Particles (65536 particles): 15.9 fps