Building a Million Particle System

Lutz Latta
Massive Development GmbH

Overview

- History of particle systems
- Basics of particle system simulation
- Particle systems on the GPU
  - Stateless simulation and rendering
  - State-preserving physical simulation
  - Sorting algorithm
  - Rendering algorithm

History of Particle Systems:

"Spacewar!"

- 1962
- Second video game ever!
- Uses pixel clouds as explosions
- Random motion

Image from online, public domain version of Spacewar!
http://lcs.www.media.mit.edu/groups/el/projects/spacewar/

History of Particle Systems:

"Asteroids"

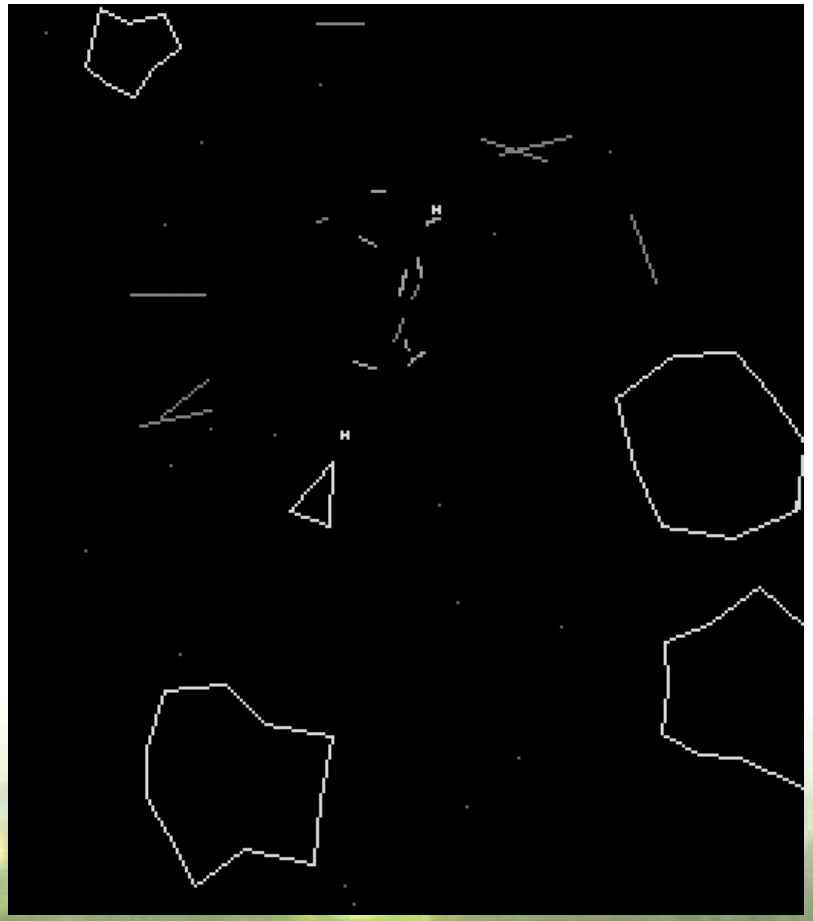


Image from open source Asteroids clone by Justin Irwin
http://www.mirwin.net/

- 1978
- Uses short moving vectors for explosions
- Probably first "physical" particle simulation in CG/games

History of Particle Systems:

"Star Trek II: The Wrath of Kahn"

- 1983
- Movie Visual FX: Planetary fire wall
- First CG paper about particle systems by William T. Reeves
- Today concept still unaltered!

What is a Particle System (PS)?

- Individual mass points moving in 3D space
- Forces and constraints define movement
- Randomness or structure in some start values (e.g. positions)
- Often rendered as individual primitive geometry (e.g. point sprites)

Uses of Particle Systems

- Explosions

- Smoke
- Fog

Uses of Particle Systems (cont.)

- Weapon FX

- Impact FX

Basic Particle System Physics

- Particle is a point in 3D space
- Forces (e.g. gravity or wind) accelerate a particle
- Acceleration changes velocity
- Velocity changes position

Euler Integration

- Integrate acceleration to velocity:

$$v = \bar{v} + a \cdot \Delta t$$

- Integrate velocity to position:

$$p = \bar{p} + v \cdot \Delta t$$

Δt	time step
a	acceleration
v	velocity
$\bar{v}$	prev. veloc.
p	position
$\bar{p}$	prev. pos.

- Computationally simple
- Needs storage of particle position and velocity

Verlet Integration

- Integrate acceleration to position:

$$p = 2\,\bar{p} - \bar{\bar{p}} + a \cdot \Delta t^2$$

$\bar{\bar{p}}$ position two time steps before

- Needs no storage of particle velocity
- Time step needs to be (almost) constant
- Explicit manipulations of velocity (e.g. for collision) impossible

Particle Simulation on the GPU

- Stateless simulation
  - Only simple effects
  - Simulation in vertex shader
  - Possible on first generation programmable GPUs
- State-preserving (iterative) simulation
  - Simulation with textures and pixel shaders
  - Only on recent generation GPUs

NEW!

Stateless Particle Systems

- No storage of varying particle data
- Evaluation of closed form function describing movement/attribute changes
- Computed data depends only on initial values and static environment description

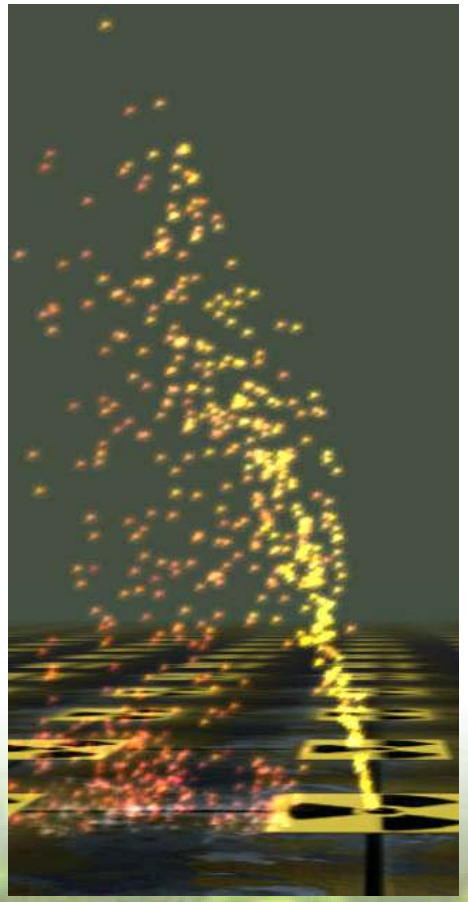


Image from NVIDIA particle system demo

Example of Stateless PS

- Position depends on initial position p_0, initial velocity v_0 and gravity g

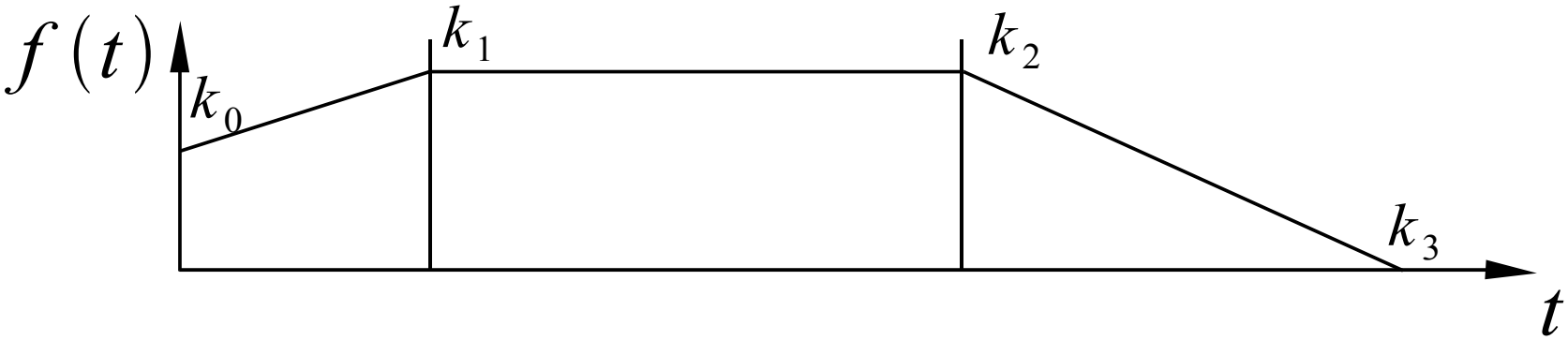
$$p(t) = p_0 + v_0 t + \frac{1}{2} g t^2$$

- Orientation depends on initial orientation ω_0 and rotation velocity φ

$$\omega(t) = \omega_0 + \varphi t$$

Example of Stateless PS (cont.)

- Color and opacity depend on linear function segments with four keyframes



First segment:

$$f_0(t) = \underbrace{\frac{k_1 - k_0}{t_1 - t_0}}_{m} t + \underbrace{k_0 - \frac{k_1 - k_0}{t_1 - t_0} t_0}_{b} = m \cdot t + b$$

Algorithm of Stateless PS

At particle birth

Upload time of birth and initial values to dynamic vertex buffer

In extreme cases only a “random seed” needs to be uploaded as initial value

At rendering time

Set global function parameters as vertex shader constants

Render point sprites/triangles/quads with particle system vertex shader

Now...

...let's talk about some hot new stuff!

State-Preserving Simulation

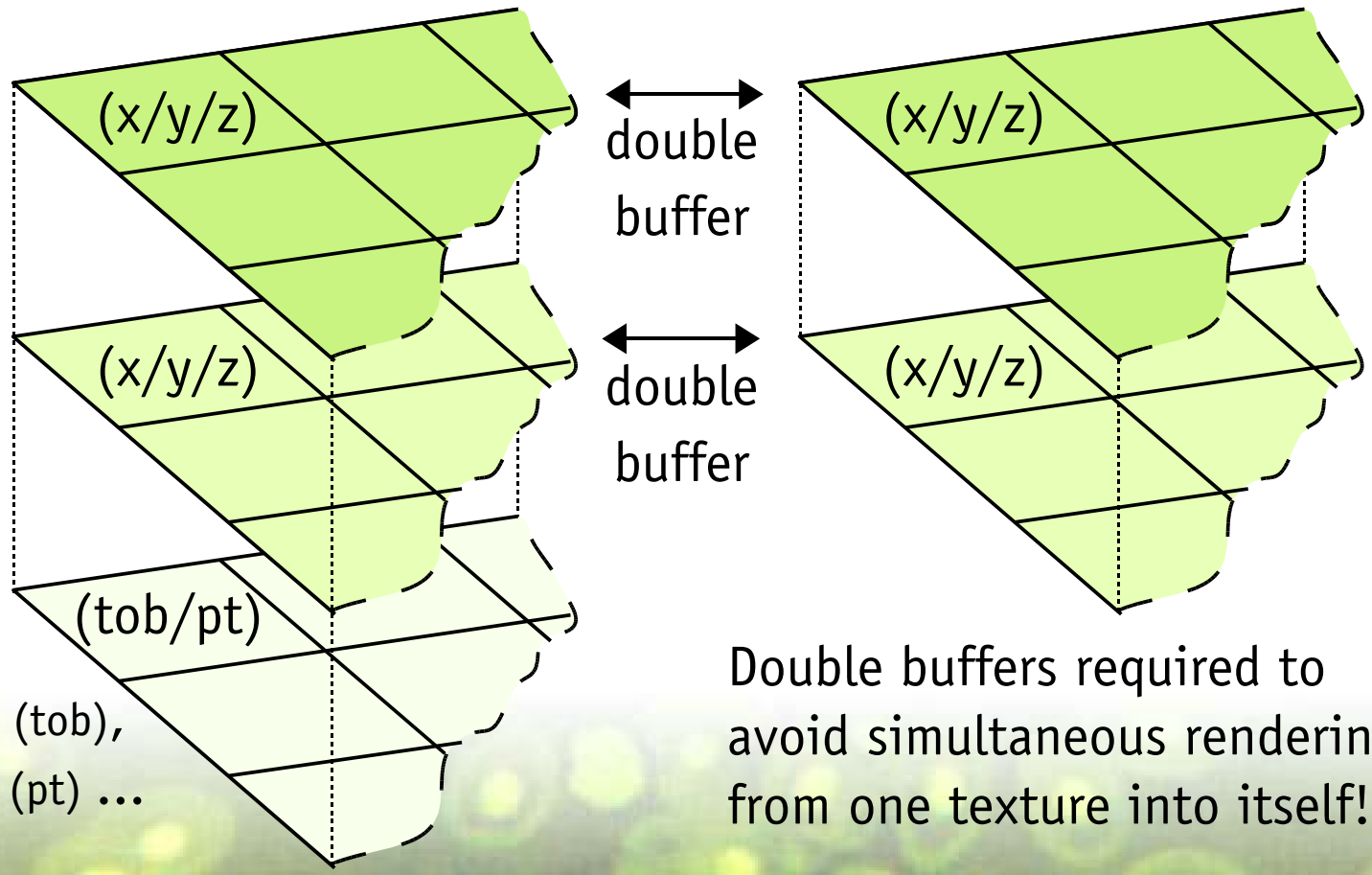
- Position and velocity stored in textures
- From these textures each simulation step renders into equally sized other textures
- Pixel shader performs iterative integration
- Position textures are “re-interpreted” as vertex data
- Rendering of point sprites/triangles/quads

Storage of Particle Data

Position texture

Velocity texture

Static info per particle: time of birth (tob), particle type (pt) ...



Double buffers required to avoid simultaneous rendering from one texture into itself!

Storage of Particle Data (cont.)

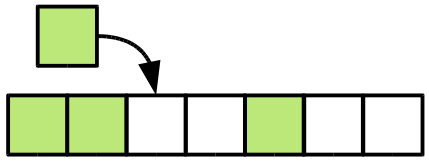
- Position and velocity stored in 2D textures
- Textures treated as 1D array
- Precision: position 32bit FP, velocity 16bit FP
- Static per particle information:
  - Time of birth, type ID, random seed
  - Not necessarily in a texture, but indices are the same as in particle textures

Algorithm for One Time Step

1. Process birth and death
2. Velocity operations (forces, collisions)
3. Position operations
4. Sorting for alpha blending (optional)
5. Transfer pixel to vertex data
6. Rendering

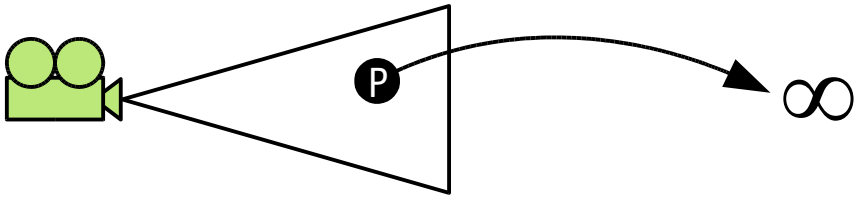
Birth of a Particle/Allocation

- Allocation is a deeply serial problem; it cannot be done efficiently on parallel GPU
- Use fast, problem-specific allocator
- Important to get compact index range
- Perform allocation on the CPU:
  - Determine next free index
  - Determine initial particle values
  - Render initial values as pixel-size points



Death of Particles

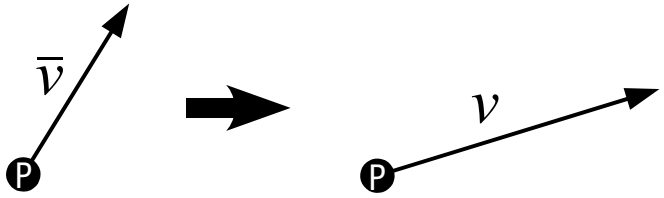
- Processed independently on CPU and GPU
- CPU: Free particle index, re-add it to allocator
- GPU: Move particles to infinity (or very far away)



- If particles fade out or “fall” out of view, this clean-up rarely really needs to be done

Velocity Operations

- Update velocity textures with various operations:
  - Global forces
  - Local forces
  - Dampening
  - Collisions
  - Flow field
  - [insert artists' dream operation here]



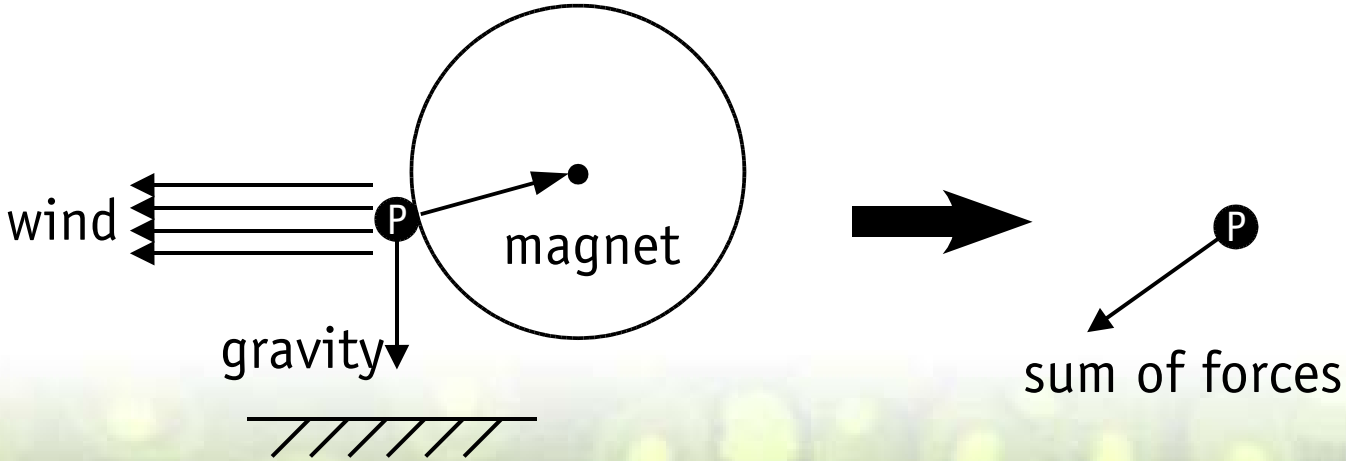
Velocity Ops: Global and Local Forces

- Global forces are position-invariant:
  Gravity, wind
- Local forces fall off based on distance:
  Magnet, orbiting, turbulence, vortex
- Fall off with $\frac{1}{r^2+\varepsilon}$ or at hard boundary
- Individual particles might scale effect based on mass, air resistance etc.

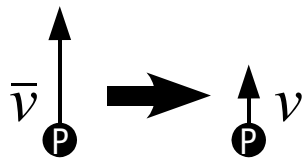
r	distance
ε	small epsilon

V. Ops: Global and Local Forces (cont.)

- Add all forces to one force vector
- Convert force to acceleration ($F = m \cdot a$)
  - identical if all particles have unit mass



Velocity Ops: Dampening



$$v = c \cdot \bar{v}$$

c constant scale factor

- Dampening:
  - Scale down velocity vector
  - Simulates slow down in viscous materials
- Un-dampening:
  - Scale up velocity vector
  - Simulates self-moving objects, e.g. bee swarm

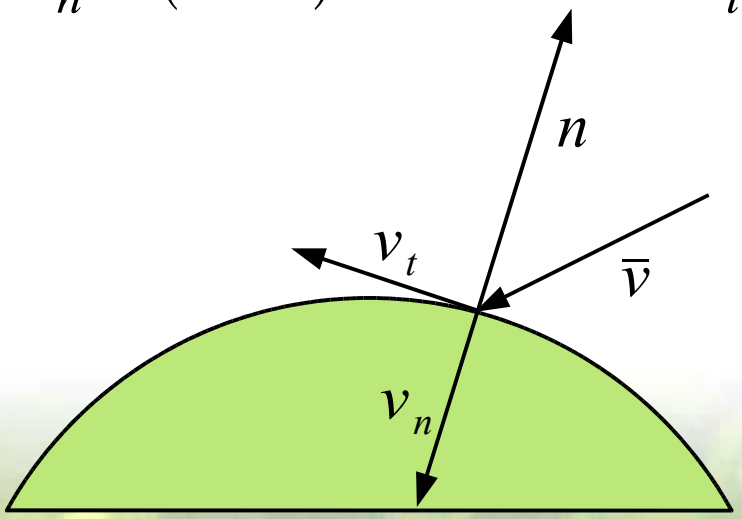
Velocity Ops: Collision

- Collision on GPU limited for efficiency:
  - Primitive objects: sphere, plane, box
  - Texture-based height fields, i.e. terrain (might be dynamic!)
- Algorithm:
  1. Detect collision with expected new position
  2. Determine surface normal at approximate penetration point
  3. React on collision, i.e. alter velocity

Collision Reaction

- Split velocity (relative to collider) into normal v_n and tangential v_t component:

$$v_n = (\bar{v} \cdot n)\bar{v} \qquad v_t = \bar{v} - v_n$$

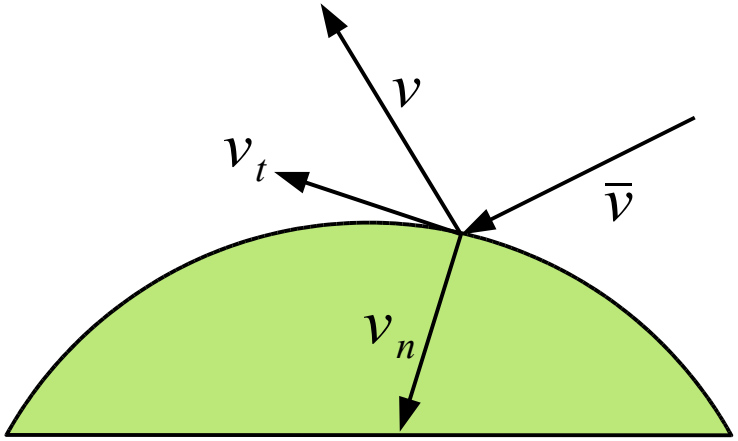


Collision Reaction (cont.)

- Friction μ reduces tangential component
- Resilience ϵ scales reflected normal comp.

Resulting velocity:

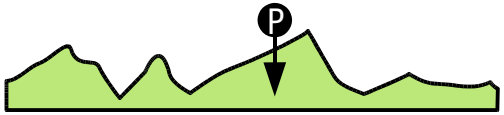
$$v = (1-\mu)v_t - \epsilon v_n$$



Shows some artifacts (see paper for fix-ups)

Collision with Height-Field

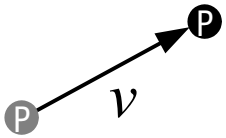
- Quite efficient on GPU, only 3 shader ops:
  - Multiply-add particle position into tex. coords.
  - Look-up height in texture
  - Compare particle height against texture height



- Reaction requires surface normal
  - Look-up from normal map
  - Or compute from three height samples

Position Update

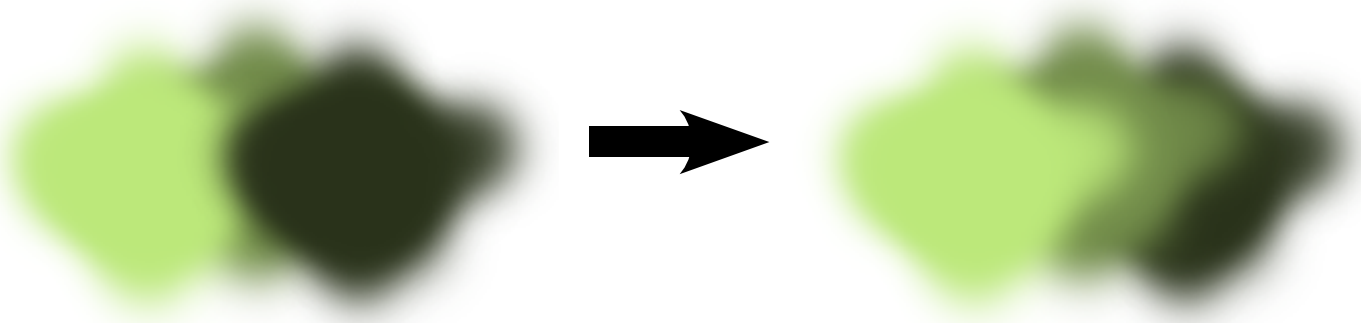
- Euler integration:



  – Simply apply velocity (see previous slides)
- Verlet integration:
  – Apply acceleration caused by forces
  – Apply other simple effects like dampening etc.
  – Collision is best handled with position constraints, i.e. move particle outside a sphere
  – Collision reaction is then stored implicitly in constraint-based position shift

Sorting for Alpha-Blending

- Alpha-blended particles show artifacts when rendered unsorted



Unsorted, wrong order Sorted, right order

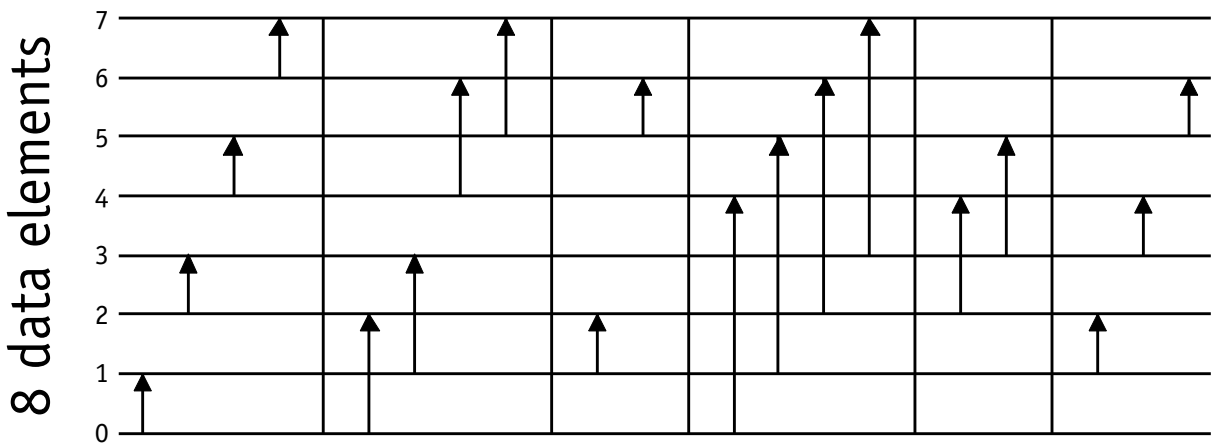
Sorting for Alpha-Blending (cont.)

- Try first:
  Check whether commutative blending (add, multiply) can be used instead!
- Otherwise:
  The GPU can sort nowadays...
  - Put viewer-distance and index into a texture
  - Sort this texture (at least partially)
  - Render with indirection over sorted indices

Sorting Networks

- GPU needs parallel sorting algorithm
- Check for total order is serial, i.e. expensive
- Best to use a data-independent algorithm
- Sorting networks always have fixed number of comparisons
- Efficient algorithm: Odd-Even Merge Sort

Odd-Even Merge Sort



Arrows mark one comparison, maybe with swapping.

Vertical lines separate rendering passes.

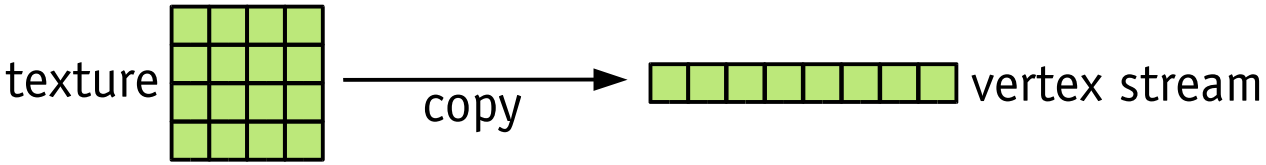
- Every step increases or at least keeps order!
- Allows distributing the sorting over several frames (high frame-to-frame coherence)

Transfer Pixel to Vertex Data

- After simulation (and sorting) we need to transfer particle positions to geometry data
- Point sprites allow most efficient transfer: only one vertex per particle
- Triangles or quads need replicated vertices
- Several methods to transfer texture data to vertices exist:

Über-Buffer (also called Super Buffer)

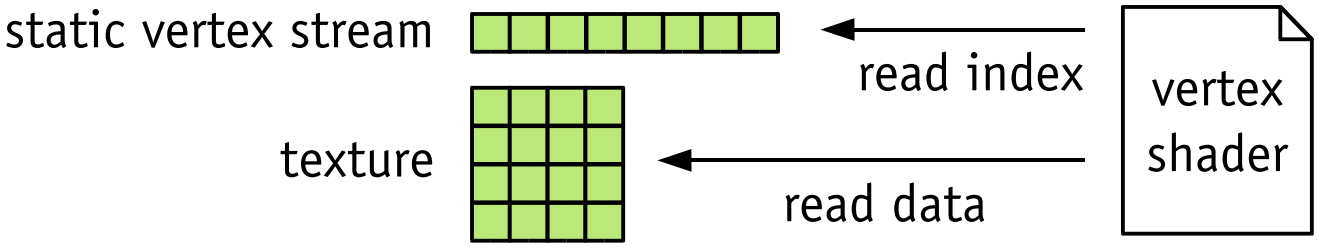
- Store any data in graphics card memory
- Copy from one buffer (here texture) to another buffer (here vertex buffer)



- Available on current hardware (GFFX, R9xxx)
- OpenGL-only, currently not in DirectX
- Various OpenGL extensions: ARB_super_buffer, NV_pixel_buffer_object, NV_pixel_data_range

Vertex Textures

- Access textures from vertex shaders
- Vertex shader actively reads particle positions



- Available in DirectX (VS3.0) and OpenGL (ARB_vertex_shader/GLSL)
- Hardware support right around the corner

Rendering

- Render as point sprites or triangles/quads?

- Point sprites reduce work for vertex unit
- Rotation of point sprites tricky:
  requires pixel shader texture rotation
- Triangles/quads cause less per-pixel work

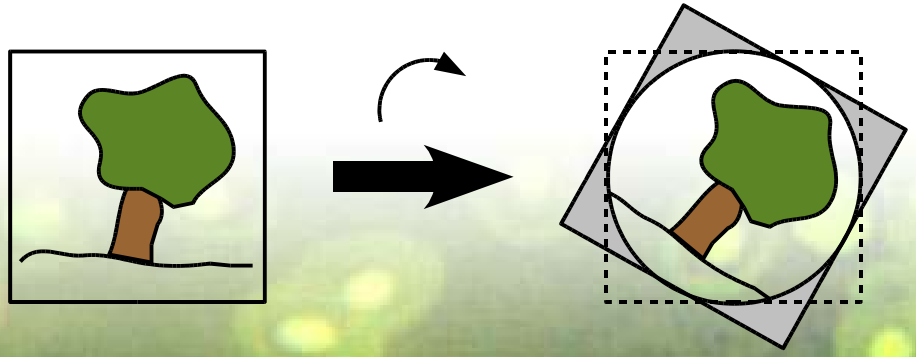
➔ You decide!

Handle Other Particle Attributes

- Up to here we've only discussed the particle position...
- How do we get color, opacity, size, orientation, texture coords. of particle?
- ➔ Mix with stateless particle simulation!
  - Functions of these values have lower complexity
  - Use closed form functions as in example at the beginning

Point Sprite Rotation

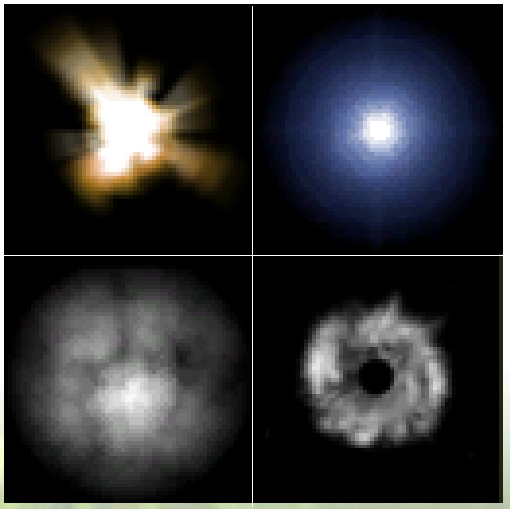
- You can't rotate point sprites, as they are axis-aligned quads
- You can however rotate texture coords.
- Rasterizer generates tex. coords. [0..1]×[0..1]
- Rotate in pixel shader with 2×2 matrix



Reduces usable texture area to inner circle!

Multiple Particle Textures

- Problem: You cannot change textures while drawing a sequence of particles
- ➔ Combine several textures as sub-textures of one larger texture
  - Adjust texture coords. accordingly
  - For point sprites: Transform tex. coords. in pixel shader (comes free with rotation)



Demo

• • •

Conclusion

- Stateless simulation is easily set-up and has broad hardware support
- State-preserving simulation allows more complex FX, increasing player immersion
- Use sorting when necessary – it's efficient!
- Make particles smaller – it's more realistic!
- Explore new gameplay uses – it's more fun!

Questions

?

Thanks for their support:

Ingo Frick, Prof. Andreas Kolb,
Sieggi Fleder, Matthias Wloka, Simon Green
Dr. Christoph Luerig, Mark Novozhilov

More info: www.2ld.de/gdc2004/