MARCH 5-9
2007
SAN FRANCISCO
MOSCONE CENTER
TAKE CONTROL
www.gdconf.com
CMP

Game Developers Conference 07

TAKE CONTROL

March 5-9, 2007 in San Francisco

Everything about Particle Effects

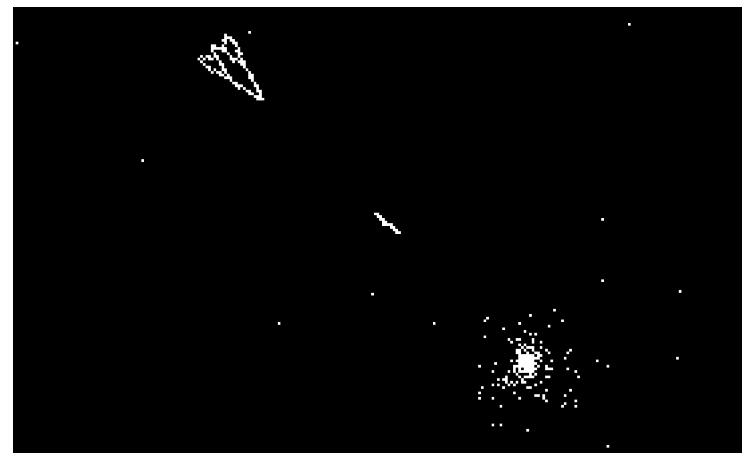
Lutz Latta
Electronic Arts Los Angeles

Overview

- History
- Definition
- Simulation basics
- Where to simulate
- Particle operations
- High quality rendering
- Performance tips

History of Particle Systems

- 1962: Pixel clouds in "Spacewar!" (2nd video game ever)
- 1978: Explosion physics simulation in "Asteroids"
- 1983: First CG paper about particle systems in "Star Trek II: The Wrath of Kahn" by William T. Reeves



Images: (top) Public domain version of Spacewar! http://spacewar.oversigma.com/
(bottom) ©ACM, used by permission of Association of Computing Machinery
Reeves1983, Particle Systems – Technique for Modeling a Class of Fuzzy Objects

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)

Basic Particle System Physics

- Particle is a point in 3D space

Forces (eg. gravity or wind) **accelerate** a particle

↓

Acceleration changes **velocity**

↓

Velocity changes **position**

Particle Simulation Options

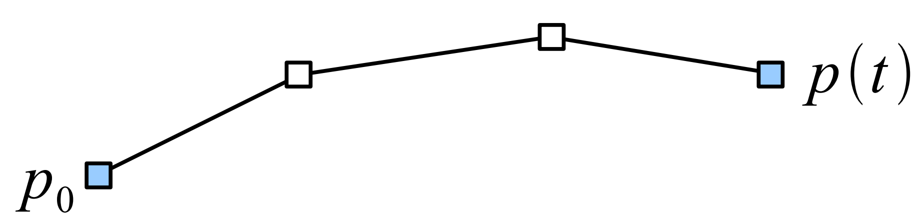
- Evaluating closed-form functions
  - stateless simulation
- Iterative integration
  - updates previous state of system
    - Euler integration
    - Verlet integration
    - Higher (eg 4th) order Runge-Kutta integration

Closed-Form Function

- Parametric equations describe current position
- Position depends on initial position p_0, initial velocity v_0 and fixed acceleration (eg gravity g)

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

- No storage of intermediate values (stateless)



Euler Integration

- Integrate acceleration to velocity:

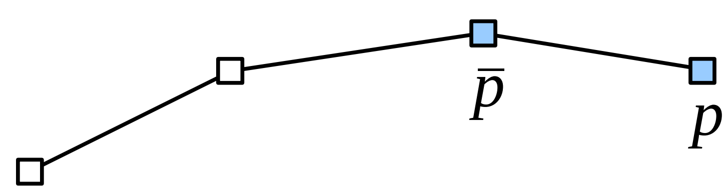
$$v = \bar{v} + a \cdot \Delta t$$

Integrate velocity to position:

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

- Computationally simple
- Needs storage of particle position and velocity

Δt	time step
a	acceleration
v	velocity
$\bar{v}$	previous velocity
p	position
$\bar{p}$	previous position



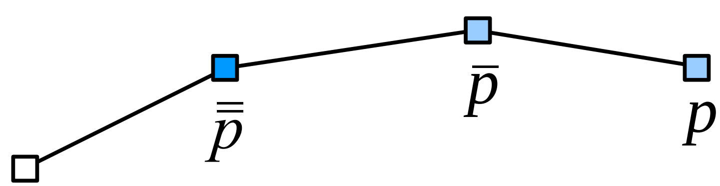
Verlet Integration

- Integrate acceleration to position:

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

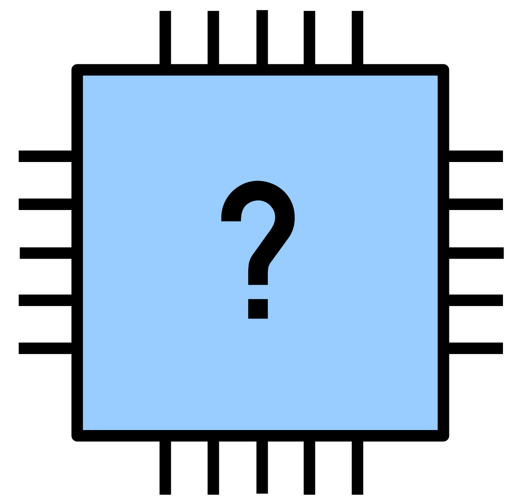
$\bar{\bar{p}}$ position two time steps before

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



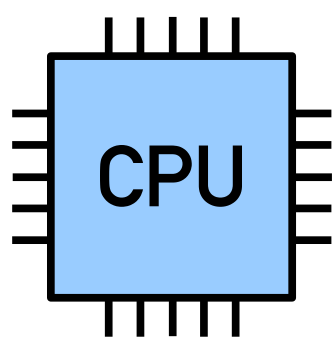
Where to Simulate?

- CPU
  - Main core
  - Other core
- GPU
  - Vertex shader
  - Pixel shader
  - Geometry shader
- Other
  - PS2 VU, PS3 SPU
  - Physics processor



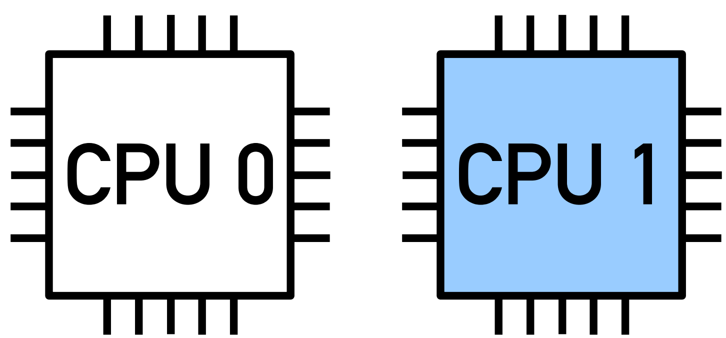
CPU Simulation

- + Simple, straight forward
- + Everything possible
- - General purpose processor, not optimized for this
- - Uses cycles that could be used for more complex algorithms, eg gameplay, AI
- - Requires upload of resulting simulation data for rendering every frame



CPU Simulation: Multi-core

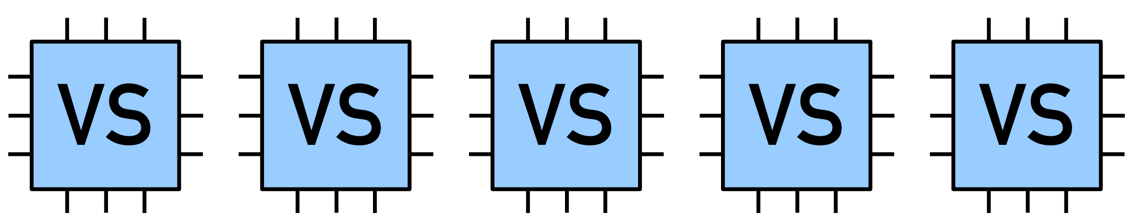
- If other CPU cores are available (multi-core PC, Xbox360), use their power
- PS are usually a quite isolated system, ie relatively easy to move to separate processor
- Individual particles typically independent from each other ▸ distribute updates over many threads/processors



Vertex Shader Simulation

- Vertex shaders cannot store simulation state (data only passes through to next stage)
- Can only simulate with „closed form function" methods above
- Limits use to simple „fire and forget" effects

- DX10 can store vertex/geometry data - discussed later



Vertex Shader Simulation: Data Flow

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

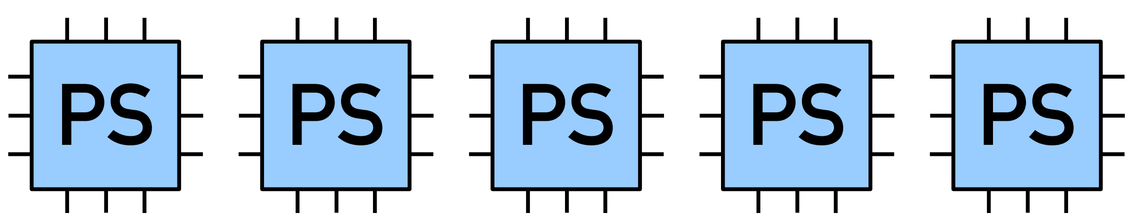
Set global function parameters as vertex shader constants

↓

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

Pixel Shader Simulation

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

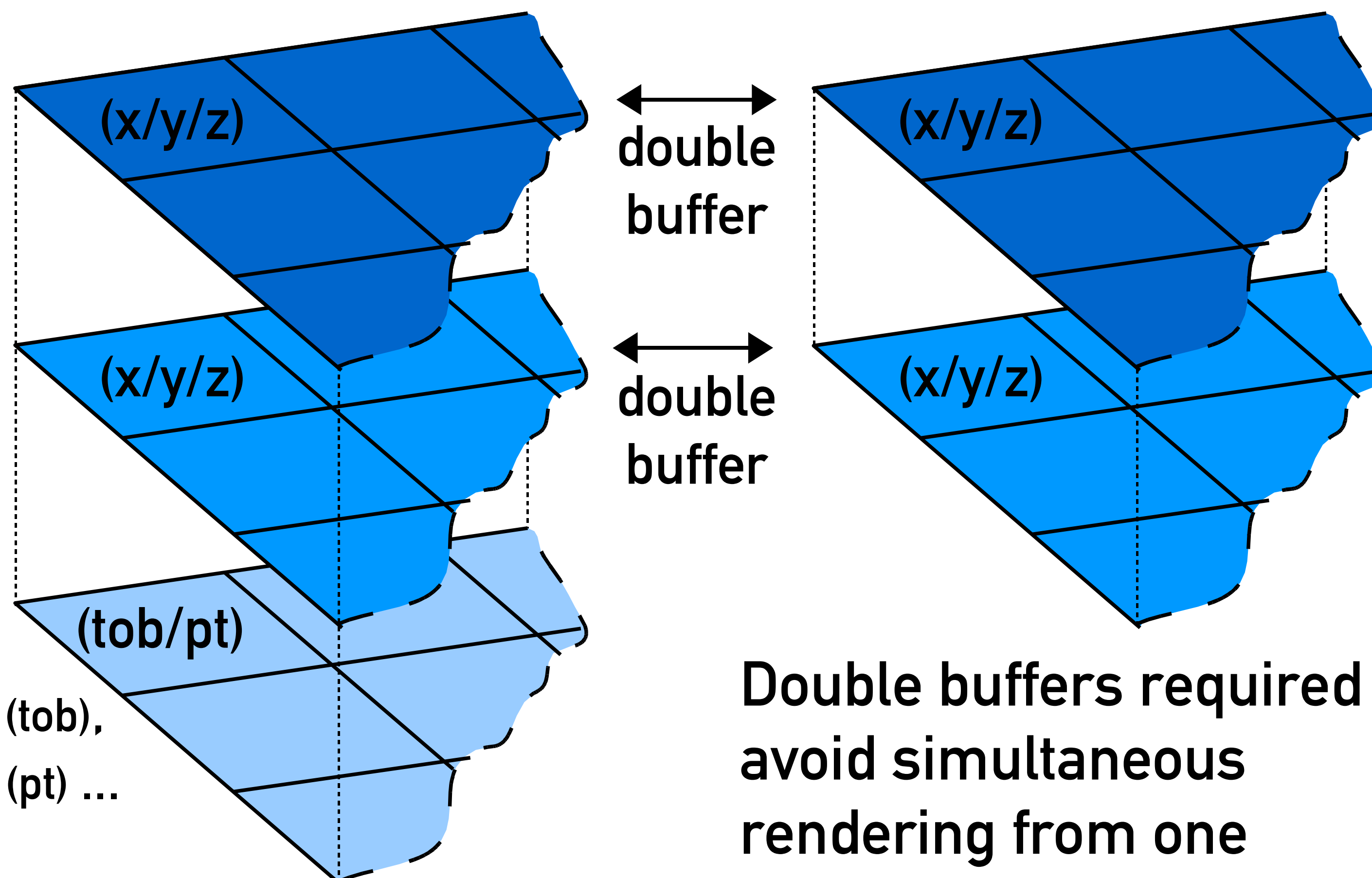


Pixel Shader Simulation: Data Storage

Position texture

Velocity texture

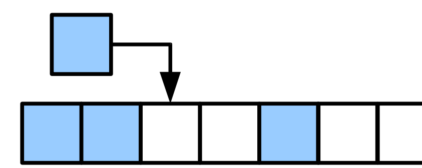
Static info per particle: time of birth (tob), particle type (pt) ...



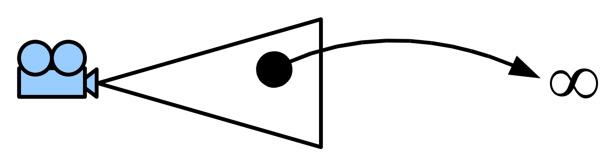
Double buffers required to avoid simultaneous rendering from one texture into itself!

Pixel Shader Simulation: Allocation

- Position/velocity textures are treated as 1D array
- Array index (ie texture coordinate) for new particles determined on CPU
- Use fast, problem-specific allocator
- Important to get compact index range



- Render start values for new particles as points into textures
- At death of a particle
  - GPU: Move to infinity
  - CPU: Return free index to allocator

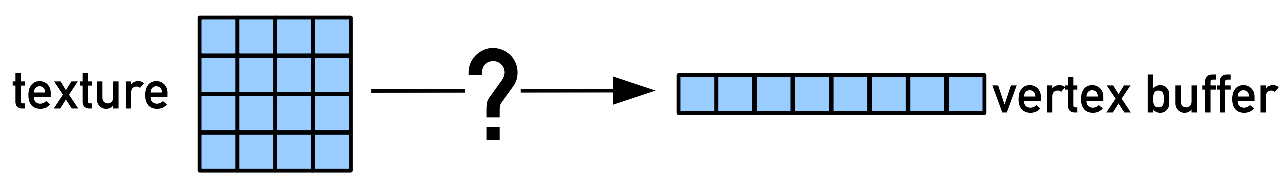


Pixel Shader Simulation: Updates

- Velocity update
  - Set one texture of the double buffer as render target
  - Set up other texture for sampling
  - Draw full-screen quad (or smaller sub-rectangle)
  - Use pixel shader to do one iterative integration step
- Position update
  - Do the same on position textures
  - Use pixel shader to update positions, also sampling from current velocity texture
- With MRT (Multiple Render Targets) can do both in one step

Pixel Shader Simulation: Pixel to Vertex Data Transfer

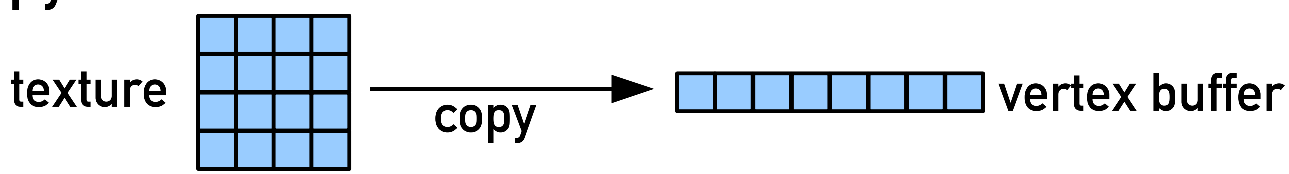
- For final rendering position texture needs to be used for generating vertices at the particle positions



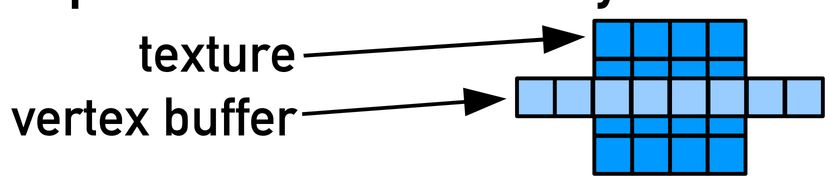
- Two conceptual options:
  - Render-to-vertex-buffer
  - Vertex textures

Pixel Shader Simulation: Render to Vertex Buffer

- Two options:
  - Copy texture to vertex buffer



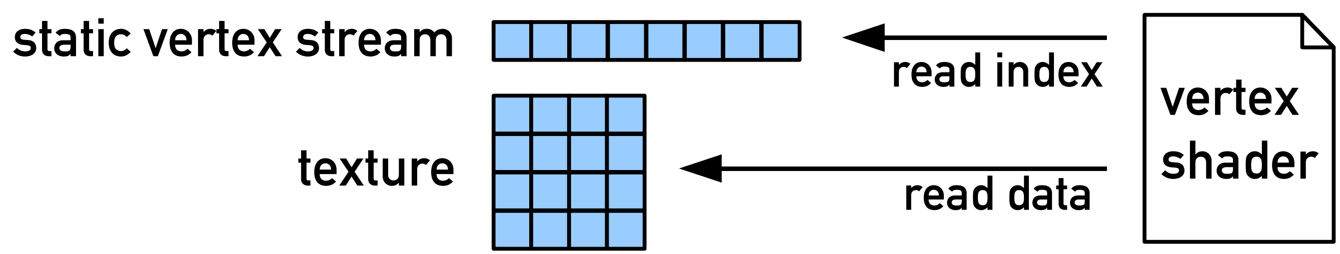
  - Re-interpret texture memory as vertex memory



- Available on consoles and in DX10
- Available in DX9 as unofficial ATI extension (R2VB)
- Not generally available in DX9!
- Available in OpenGL through extensions

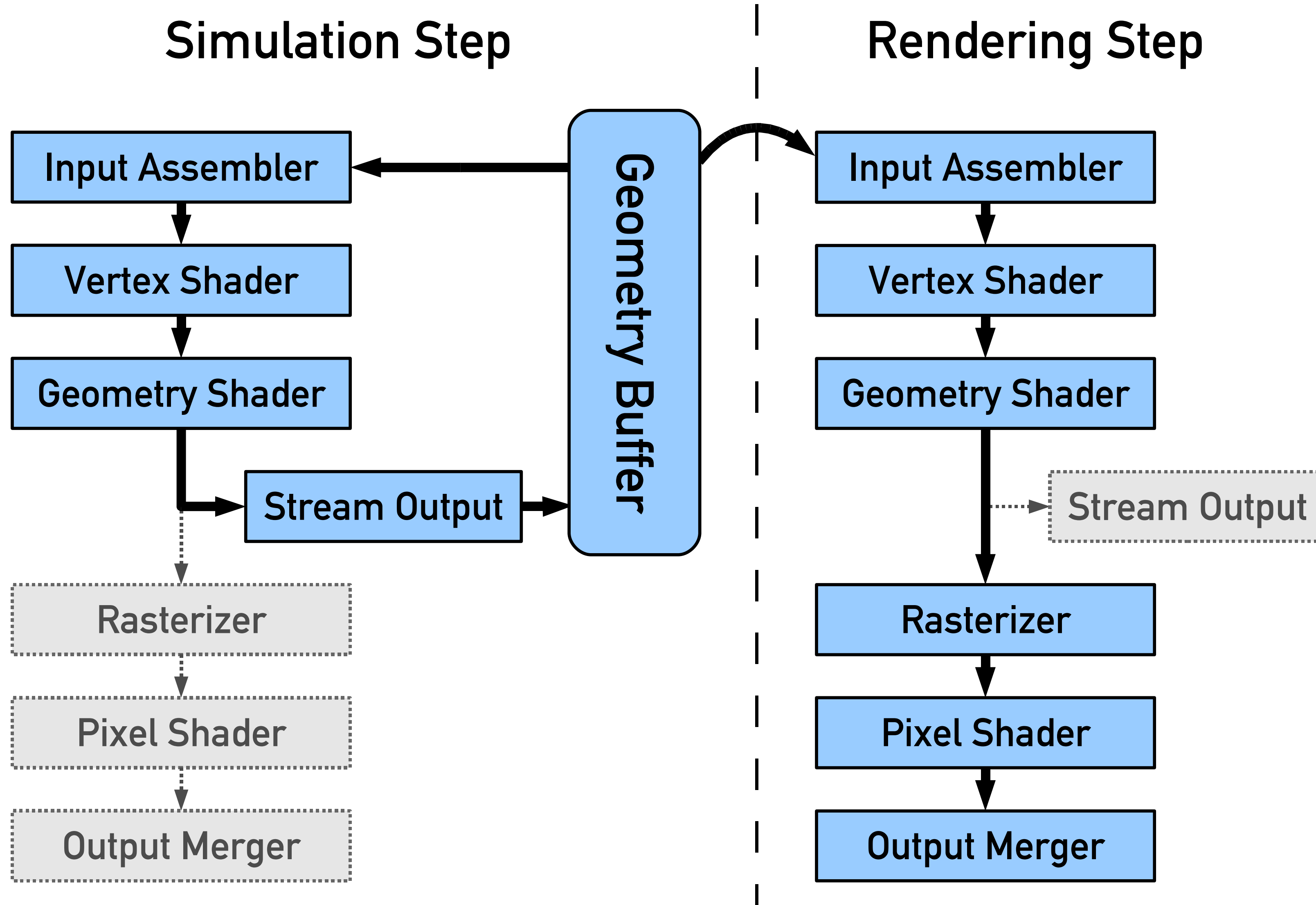
Pixel Shader Simulation: Vertex Textures

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



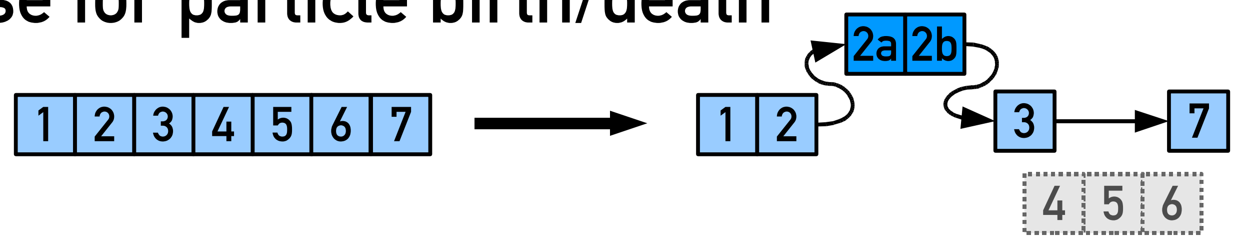
- Available in DX9 (VS3.0, except ATI X1xxx)
- Available in OpenGL on VS3.0 hardware
- High latency on early VS3.0 hardware
- Render-to-VB has usually better performance

Geometry Shader Simulation

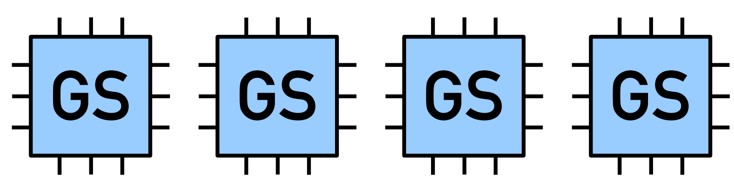


Geometry Shader Simulation

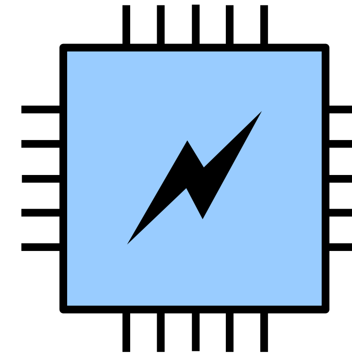
- Geometry shader can create new or destroy old data ▸ use for particle birth/death



- Simulation step reads and writes point primitives to/from geometry buffer
- Render geometry shader creates quad per point
- Available in DX10 and OpenGL on SM4.0 hardware
- Check out sample in DirectX SDK



Other Processors



- Playstation 2 Vector Unit
  - Similar to vertex and geometry shader
  - Can run closed form function simulation
- Playstation 3 Cell SPUs
  - Intended for high-volume vector arithmetic, like particle simulation
  - Can do iterative or closed form function simulation
- Custom physics processors
  - Install-base limited

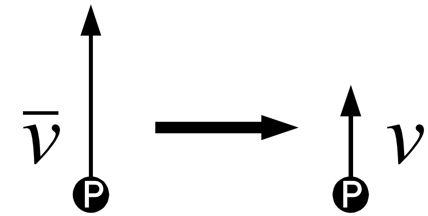
So many choices... What to do?

- Number one rule:
  What processor is most under-used in your game?
- Have a CPU core running idle?
  - Move particle simulation onto it
- GPU upload too expensive? Or shader bandwidth left, GPU running idle?
  - Use pixel or geometry shader simulation
- (On PC) Vertex shader often not a bottleneck
  - Move simple fire-and-forget effects to vertex shaders

Particle Operations

- We have focused so far only on simple velocity and position updates
- Further operations:
  - Velocity dampening
  - Rotation and scaling
  - Color and opacity animation
  - Collision

Particle Operations: Velocity Dampening



- Scale down (or up) velocity vector
- Simulates slow down in viscous materials or acceleration of self-propelled objects (bee swarm)
- Iterative simulation trivial:

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

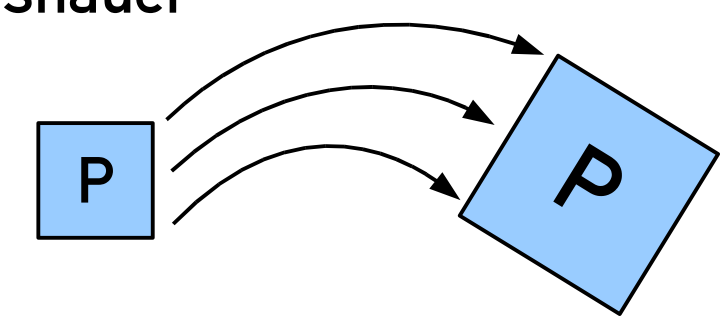
c constant scale factor

- Closed form simulation requires solving integral:

$$p(t) = p_0 + v_0 \int_0^t c^u \, du \; = \; p_0 + v_0 \cdot \begin{cases} t & \text{for } c = 1 \\ \dfrac{c^t - 1}{\ln(c)} & \text{for } c \neq 1 \end{cases}$$

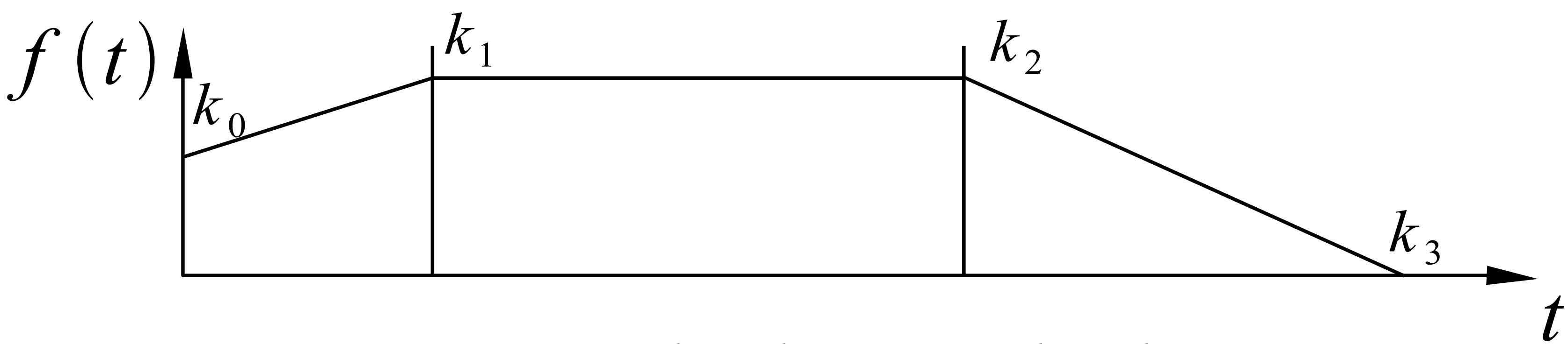
Particle Operations: Rotation and Scaling

- Typically simple animation: $x(t) = x_0 + dx\, t$
  - Start value x_0: angle/scale factor
  - Velocity dx: angular rate/scale shift
- Dampening of initial velocity useful
  - Use same formulas as position dampening
- Randomize start parameters
  - Simple random number generator enough
  - Can be done in shader



Particle Operations: Color and Opacity

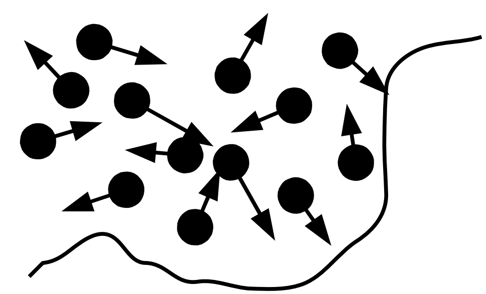
- Typically animated by keyframes
- Linear interpolation sufficient
- Can be done efficiently with fixed number (eg 4) keyframes in vertex shader



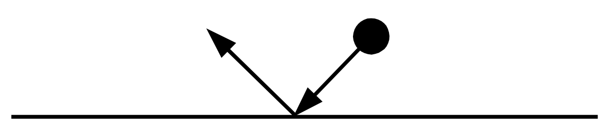
First segment: $f_0(t) = \underbrace{\frac{k_1 - k_0}{t_1 - t_0}}_{m} t + \underbrace{k_0 - \frac{k_1 - k_0}{t_1 - t_0} t_0}_{b} = m \cdot t + b$

Particle Operations: Collision

- Generic collision (every particle against every particle and object in the scene) usually prohibitively expensive

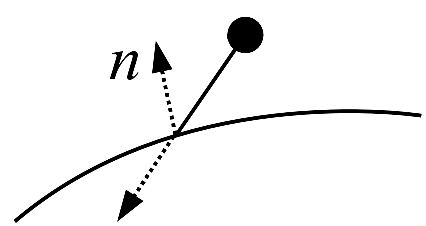


- Restrict to „important“ particles
- Simplify collisions:
  - Primitives: Plane, box, sphere
  - Height fields: Terrain, depth maps of main objects



Particle Operations: Collision Detection

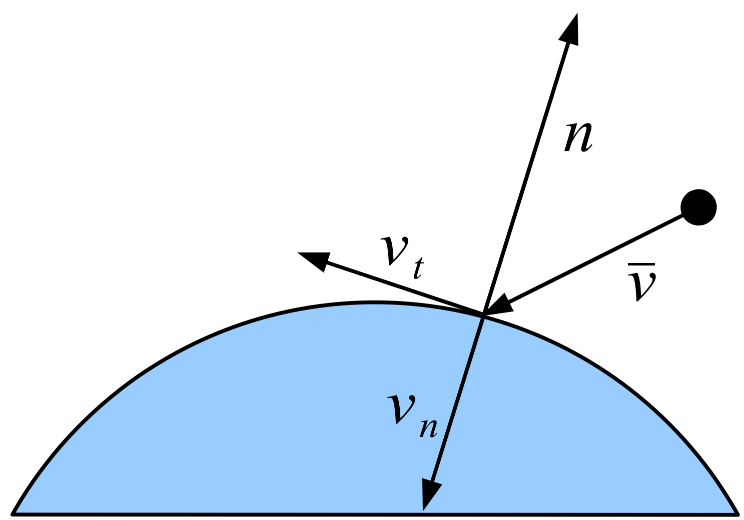
- Detect collision ie if position is inside collider body
  - Primitives:
    - Test implicit surface formula (eg point below plane)
  - Height field:
    - Simple 2D test of particle position vs height value
    - Similar to shadow map depth test ▸ can be done in pixel shader simulation
    - Can also use depth cube maps to approximate convex objects
- Also determine surface normal at approximate penetration point (implicitly or via normal map)



Particle Operations: Collision Reaction

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

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

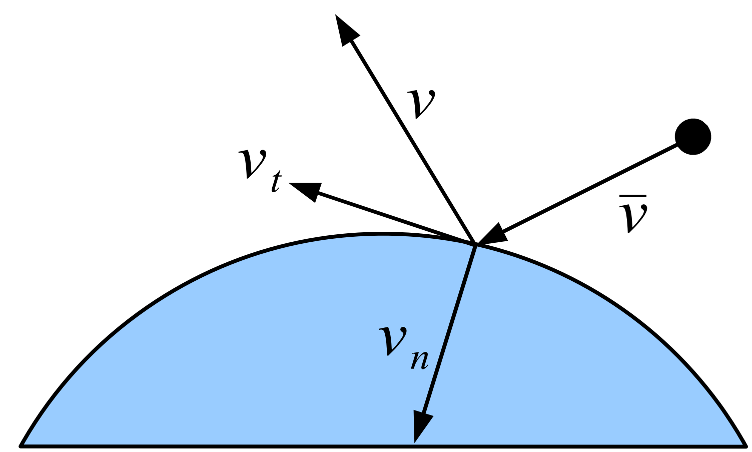


Particle Operations: Collision Reaction (cont.)

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

- Resulting velocity:

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



Shows some artifacts (see references for fixes)

Particle Sorting

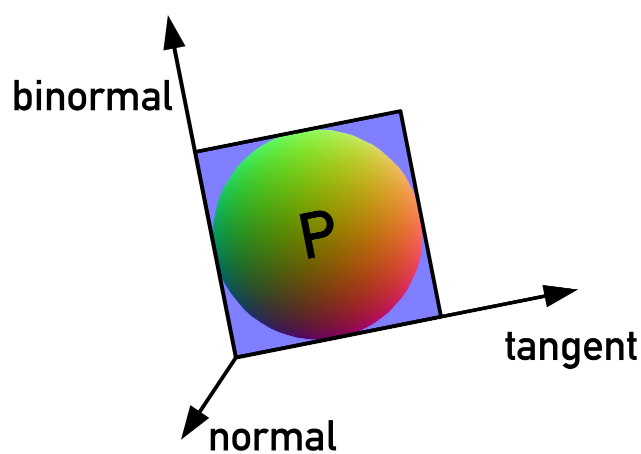
- When rendering with alpha-blending, particles should be sorted
- Sorting is expensive. Make sure you need it!
- Not necessary when a commutative blend operation (add or multiply) is used
- Ordering issues might be hardly noticeable, eg
    - Low contrast particles like middle-gray smoke
    - Small particles
    - Roughly ordered particles, eg emitted in sequence

Particle Sorting Options

- CPU simulation: Use your favorite sort algorithm
  - Potentially exploit frame-to-frame coherence (order does not change much):
  - Sort algorithm with good optimal case performance might be more important than good average case performance
- Vertex shader simulation: Can't sort properly, only by emission position on the CPU
- Pixel or geometry shader simulation: Can sort in pixel shader! See references [Latta2004]

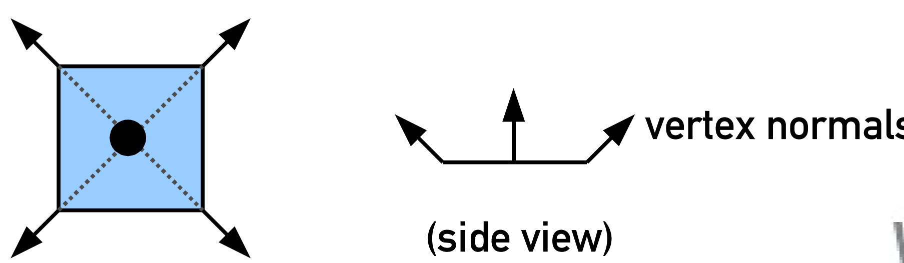
Normal Mapping

- Traditionally particles don't have a surface normal
  - cannot take lighting
- Normal can be read from texture
- Basically tangent-based normal mapping
- Tangent space based on edges of particle



Alternative Lighting

- Normal mapping is still expensive, esp with high overdraw of particles
- Simpler solutions:
  - Average light source colors. Tints particles to color scheme of scene
  - Use particle velocity (normalized) as surface normal. Totally fake, but "sort-of works"
  - Use vertex normals approximating a (squished) sphere. Improve by adding vertices in the middle of the quad



Soft Particles

- Particles have ugly hard edges where they intersect with opaque scene geometry (eg terrain)

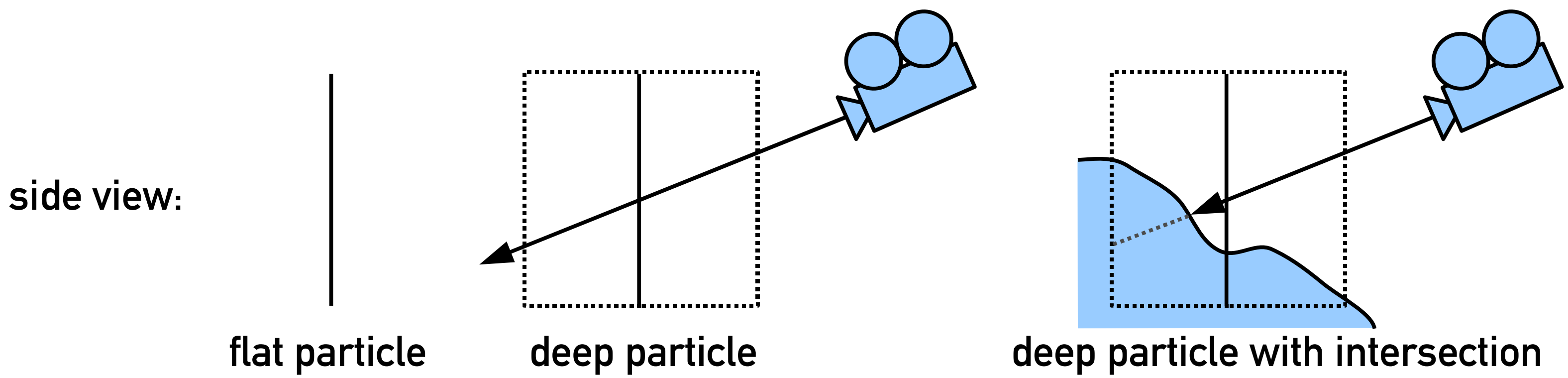
normal ("hard") particles

soft particles

- Can be avoided with blending them out softly at intersection edges

Soft Particles Algorithm

- Treat particle conceptually as a screen-aligned box, not a flat billboard



- Compute how much the view ray travels trough the box before hitting the depth in the depth map
- Use the ratio of the view ray length vs the total depth to blend out the particle opacity (multiply with original opacity)

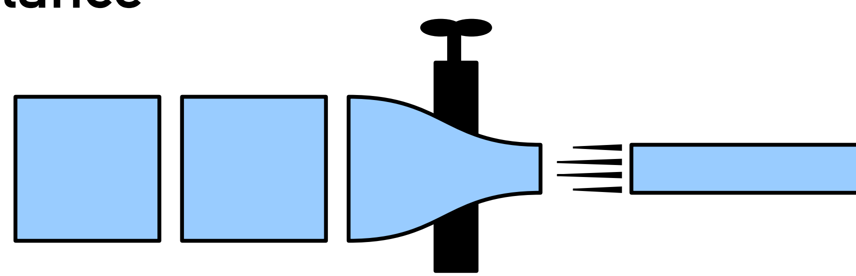
Soft Particles Requirements

- How to detect intersection edges?
- Special case: Height field ‣ Can lookup/encode approximate terrain height into particle info
- General case: Need the depth values of scene objects as a texture.
  - DX9: Depth texture needs to be rendered separately (extra pass over whole scene or with multiple-RT)
    - ‣ expensive, if you don't do it for other effects already
  - DX10: Can use current depth buffer as texture
    Can't use it as depth buffer at the same time though
    - ‣ either copy it, or don't test z, as it is not needed here

Programming for Performance

- Remember:
- Updating particles is your „inner loop"
  - Code executed in high frequency, many per frame
  - Relatively simple behaviors
- Particles are often "fluff"
  - Game logic does not depend on them
  - Accuracy non-critical
  - Determinism of low importance

- Optimize aggressively!



Performance: Batching

- Operate on large batches, not individual particles

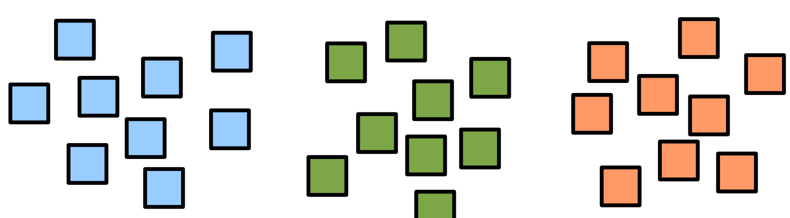
  No:
  ```
  class Particle { void update(); }
  ```
  Better:
  ```
  void updateParticles(Particle* begin, Particle* end);
  ```

- Group as-large-as-possible (or -sensible)

  Group at least all particles of one system/emitter

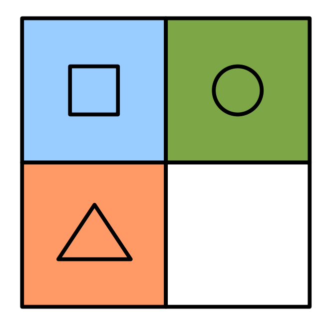
  Group all particles of one type/set of configuration parameters

  But don't group too much, forcing to add branches



Performance: Batch Rendering

- Batching even more important for rendering than simulation
- Draw calls are expensive!
- Batch at least all particles with the same configuration parameters
- Maybe batch all particles with the same render states (eg blend mode)
- Texture changes often break batches
  - put them together in a texture atlas



Performance: Instruction cache misses

- Especially important on Xbox360/PS3 CPUs
- Avoid virtual functions:

No:
```
class PhysicsModule { virtual void simulate() = 0; }
```

- Avoid branches:

No:
```
void update()
{
    if (hasRotation) { updateRotation(); }
    if (hasScaling) { updateScaling(); }
    if (hasColorAnimation) { updateColorAnimation(); }
    if (hasAlphaAnimation) { updateAlphaAnimation(); }
}
```

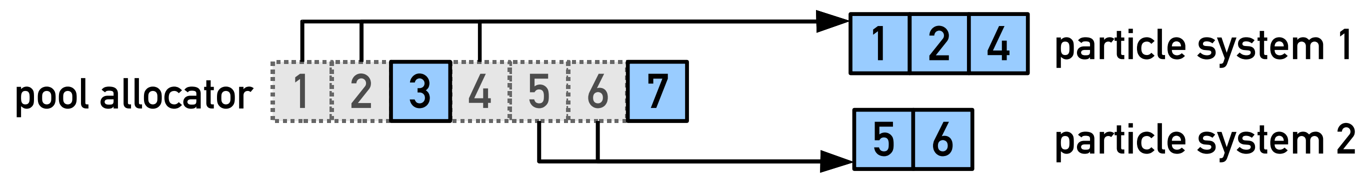
- Maybe use generic programming (templates) to compile variations taking/skipping a branch

Performance: Vector Arithmetic

- If you can, use processor specific vector instructions: SSE, Altivec, ...
- On GPU you have to use them anyway
- Try compiler intrinsics, if you are no assembler expert
- Or just use your super-optimized math library...
- On PC:
  - Can use different code paths depending on processor feature support
  - Slightly different results usually not problematic here

Performance: Memory

- Avoid using your standard allocation for particles
- Pre-allocate a pool of particles and just hand out elements from the pool (fixed-size pool allocator)



- Keep particles close together in memory to avoid data cache misses
- Avoid cache unfriendly structures, eg linked lists
- When using GPU particles, use these allocation schemes to determine the „address“ of the data in vertex buffers/textures

Performance: Scalability

- Particles often need level-of-detail (LOD) reductions
  - Too many particle systems due to long view distance
  - On PC: machine specific performance differences
- Typically a priority level is necessary
  - Some particles are game-play critical, ie convey important information about some event or state of an object ▸ don't cut them, at most reduce them
  - Other particles will be more or less important to overall visual quality ▸ usually requires artists' judgment

Summary

- So many options to beef up your old particle system code!
- Find your optimal processor (mix)!
- Make it fast!
- Make it spit out millions of particles!
- Make them look great!

Questions

?

More info: www.2ld.de/gdc2007

Thanks:
Ofer Estline, Mike Jones, John Versluis and the amazing Command and Conquer 3 team at EALA

Wolfgang Engel and my co-presenters

References: Particle system basics

- Reeves1983: Reeves, William T.; Particle Systems – Technique for Modeling a Class of Fuzzy Objects. In SIGGRAPH Proceedings 1983, http://portal.acm.org/citation.cfm?id=357320
- Sims1990: Sims, Karl; Particle Animation and Rendering Using Data Parallel Computation. In SIGGRAPH Proceedings 1990, http://portal.acm.org/citation.cfm?id=97923
- McAllister2000: McAllister, David K.; The Design of an API for Particle Systems, Technical Report, Department of Computer Science, University of North Carolina at Chapel Hill, 2000, ftp://ftp.cs.unc.edu/pub/publications/techreports/00-007.pdf
- Burg2000: van der Burg, John; Building an Advanced Particle System, Game Developer Magazine, 03/2000, http://www.gamasutra.com/features/20000623/vanderburg_01.htm

References: Particle systems on the GPU

- Latta2004: Latta, Lutz; Building a Million Particle System. In GDC 2004 Proceedings, http://www.2ld.de/gdc2004/
- Kolb2004: Kolb, Andreas; Latta, Lutz; Rezk-Salama, Christof; Hardware-based Simulation and Collision Detection for Large Particle Systems. In Graphics Hardware 2004 Proceedings, http://www.2ld.de/gh2004/
- Green2003: Green, Simon; Stupid OpenGL Shader Tricks, 2003, http://developer.nvidia.com/docs/IO/8230/GDC2003_OpenGLShaderTricks.pdf
- Kipfer2004: Kipfer, Peter; Segal, Mark; Westermann, Rüdiger; UberFlow: A GPU-Based Particle Engine. In Graphics Hardware 2004 Proceedings, http://wwwcg.in.tum.de/Research/Publications/UberFLOW

References: Example code

- Pixel shader simulation:
  http://www.2ld.de/gdc2004/
- Vertex shader simulation:
  NVIDIA SDK, http://developer.nvidia.com/
- Geometry shader simulation, soft particles:
  DirectX SDK, http://msdn.microsoft.com/directx/
- Particle System API, McAllister, David K.:
  http://www.cs.unc.edu/~davemc/Particle/