UberFlow: A GPU-Based Particle Engine

Peter Kipfer
Technische Universität München

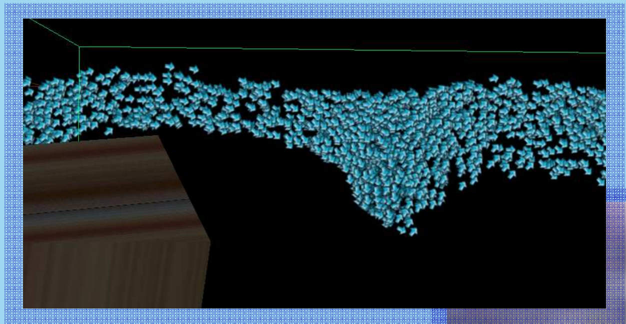
Mark Segal
ATI Research

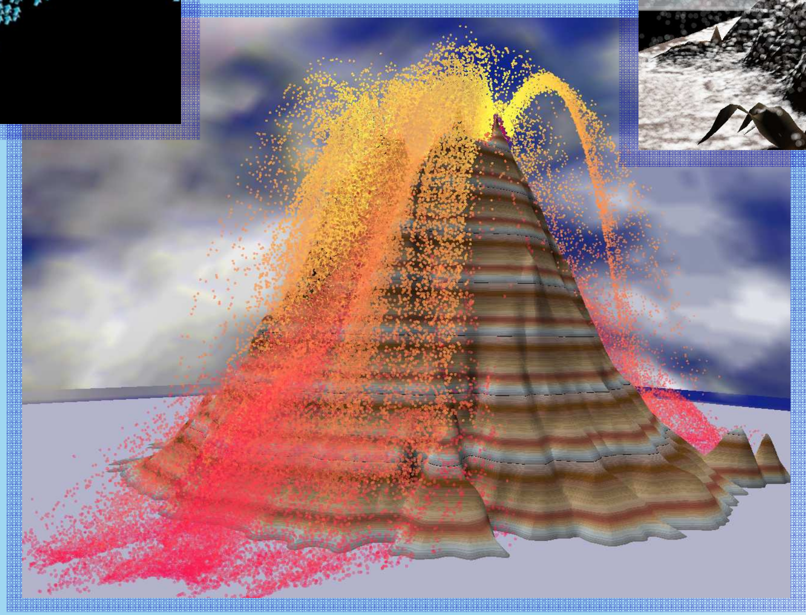
Rüdiger Westermann
Technische Universität München

tum.3D
computer graphics & visualization

Motivation

Want to create, modify and render large geometric models





Important example:

Particle system

Motivation

Major bottleneck

- Transfer of geometry to graphics card

Process on GPU if transfer is to be avoided

- Need to avoid intermediate read-back also

Requires dedicated GPU implementations

→ Perform geometry handling for rendering on the GPU

Bus transfer

- Send geometry for every frame
  - because simulation or visualization is time-dependent
  - the user changed some parameter
- Render performance: 12.6 mega points/sec

- Make the geometry reside on the GPU
  - need to create/manipulate/remove vertices without read-back
- Render performance: 114.5 mega points/sec

ATI Radeon 9800Pro, AGP 8x, **`GL_POINTS`** with individual color

Motivation

Previous work

- GPU used for large variety of applications
  - local / global illumination [Purcell2003]
  - volume rendering [Kniss2002]
  - image-based rendering [Li2003]
  - numerical simulation [Krüger2003]
- GPU can outperform CPU for both compute-bound and memory-bound applications

→ Geometry handling on GPU potentially faster

GPU Geometry Processing

Simple copy-existing-code-to-shader solutions will not be efficient

Need to re-invent algorithms, because

- different processing model (stream)
- different key features (memory bandwidth)
- different instruction set (no binary ops)

GPU Geometry Processing

Need shader access to vertex data

- OpenGL SuperBuffer
  - Memory access in fragment shader
  - Directly attach to compliant OpenGL object
- VertexShader 3.0
  - Memory access in vertex shader
  - Use as displacement map
- Both offer similar functionality

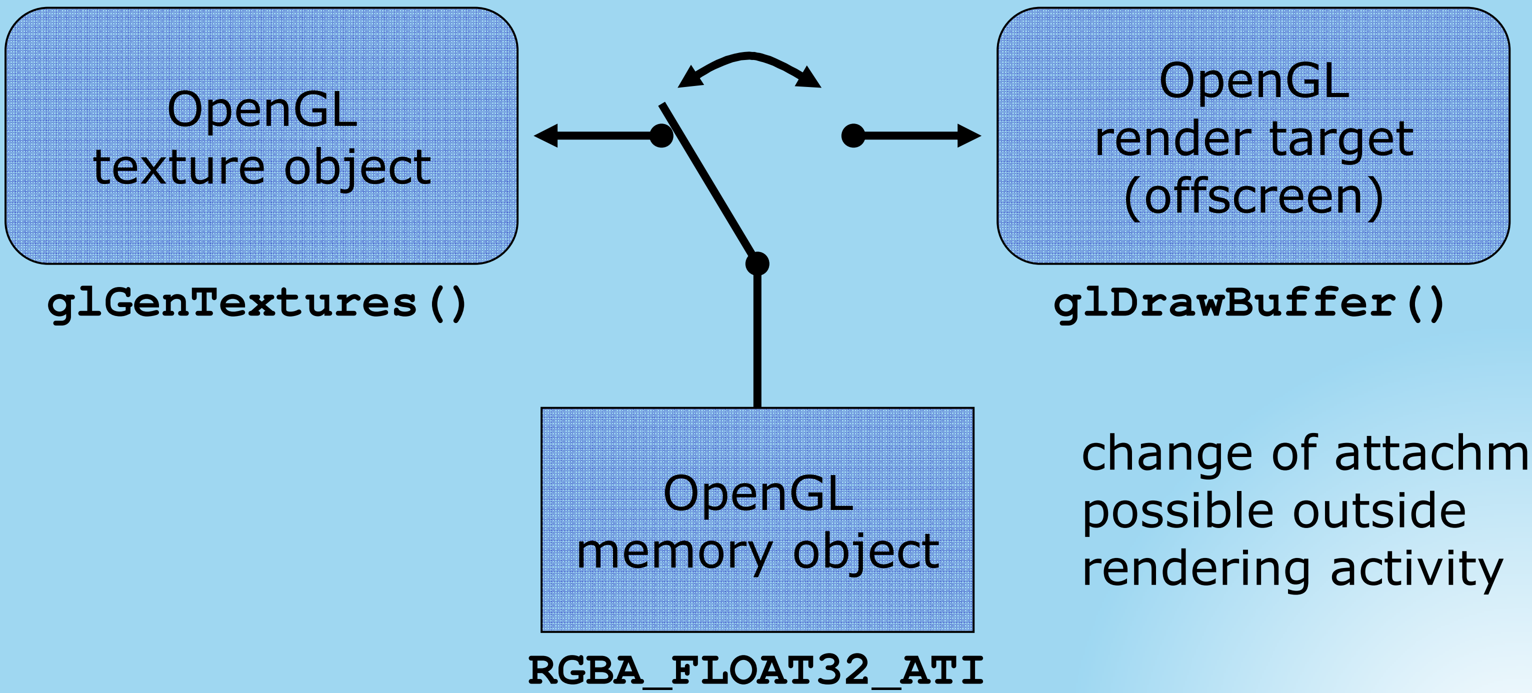
OpenGL SuperBuffer

Separate semantic of data from it's storage

- Allocate buffer with a specified size and data layout
- Create OpenGL objects
  - Colors: texture, color array, render target
  - Vectors: vertex array, texcoord array
- If data layout is compatible with semantic, the buffer can be attached to / detached from the object
  - Zero-copy operation in GPU memory
  - Render-to-vertex array possible by using floating-point textures and render targets

OpenGL SuperBuffer

- Example: floating point array that can be read and written (not at the same time)



change of attachment possible outside rendering activity

GPU Particle Engine

cool demo

Overview

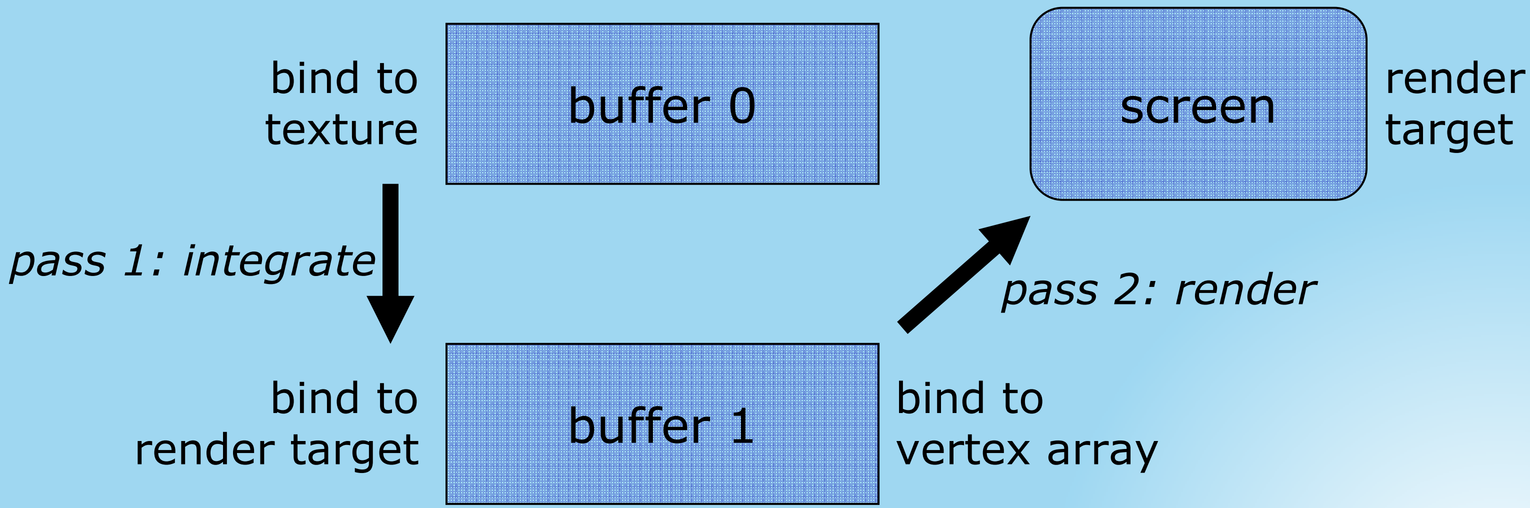
GPU particle engine features

- Particle advection
  - Motion according to external forces and 3D force field
- Sorting
  - Depth-test and transparent rendering
  - Spatial relations for collision detection
- Rendering
  - Individually colored points
  - Point sprites

Particle Advection

Simple two-pass method using two vertex arrays in double-buffer mode

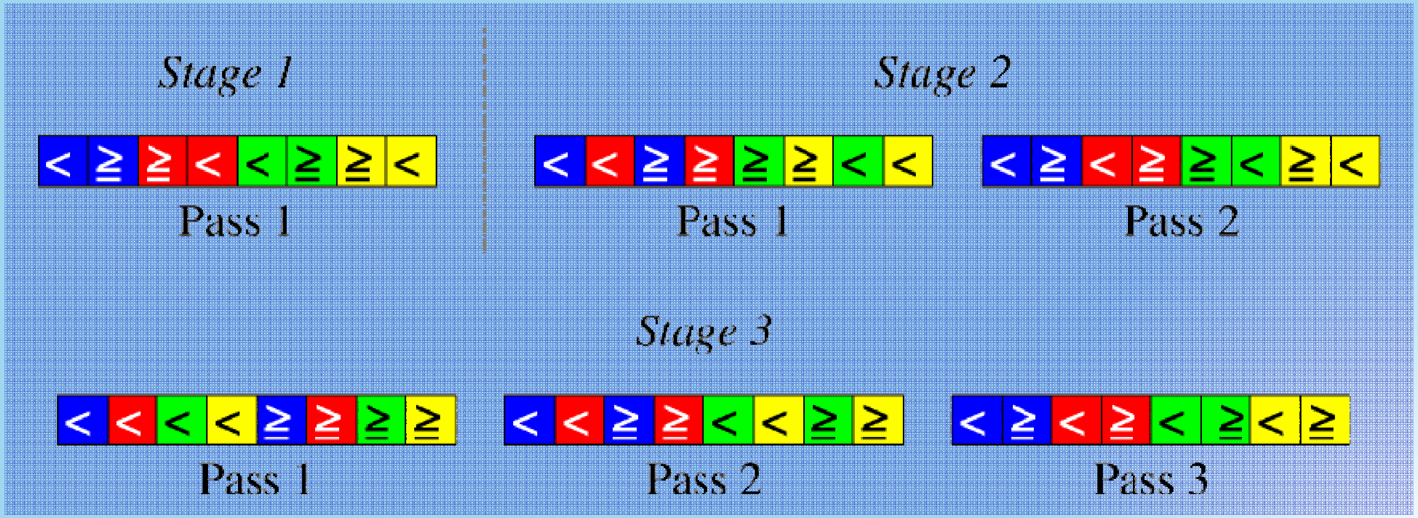
- Render quad covering entire buffer
- Apply forces in fragment shader



Sorting

Required for correct transparency and collision detection

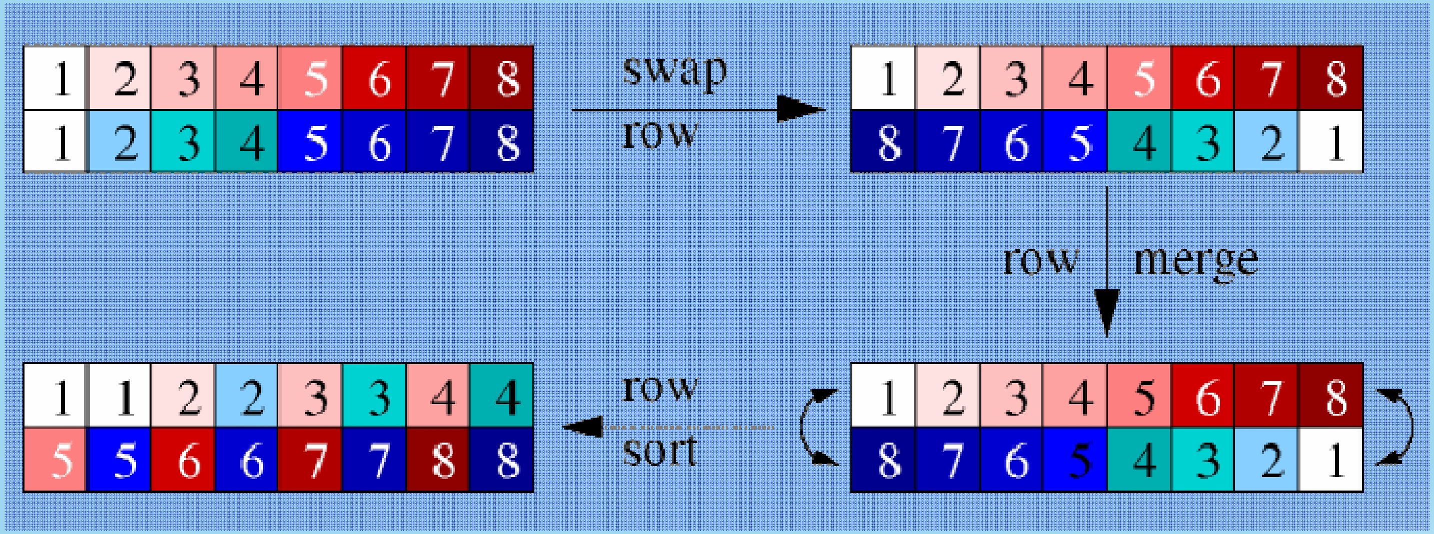
- Bitonic merge sort (sorting network) [Batcher1968]
- Sorting n items needs (log n) stages
- Overall number of passes ½ ($\log^2 n$ + log n)



computer graphics & visualization

Sorting a 2D field

- Merge rows to get a completely sorted field

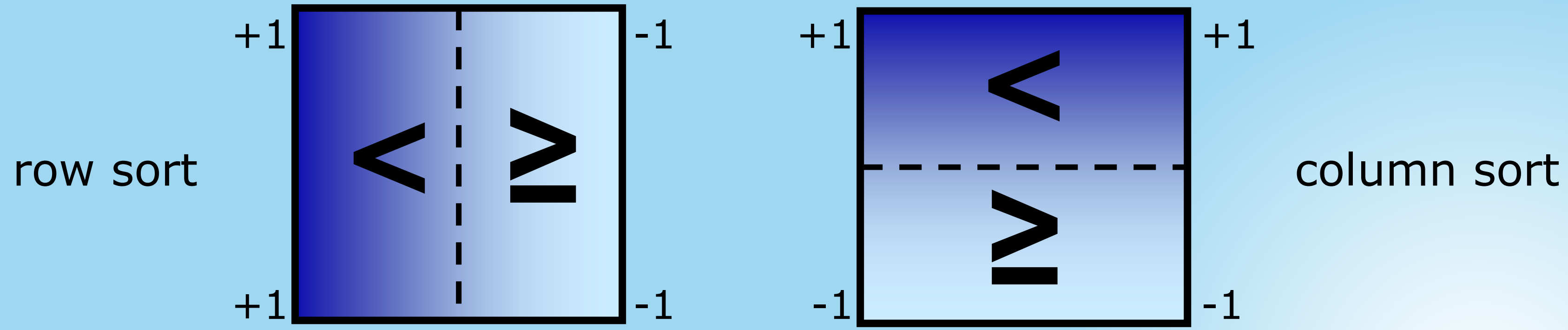


- Implement in fragment shader [Purcell2003]
  - A lot of arithmetic necessary
  - Binary operations not available in shader

Fast sorting

Make use of all GPU resources

- Calculate constant and linear varying values in vertex shader and let raster engine interpolate
- Render quad size according to compare distance
- Modify compare operation and distance by multiplying with interpolated value



Fast sorting

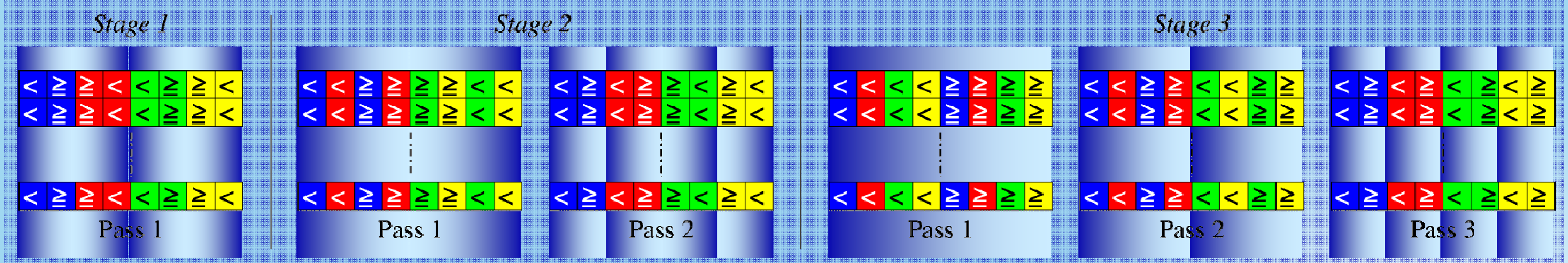
- Perform mass operations (texture fetches) in fragment shader

  `t0` = fragment position
  `t1` = parameters from vertex shader (interpolated)

```
OP1 = TEX[t0]
sign = (t1.x < 0) ? -1 : 1
OP2 = TEX[t0.x + sign*dx, t0.y]
return (OP1 * t1.y < OP2 * t1.y) ? OP1 : OP2
```

Fast sorting

- Final optimization: sort [index, key] pairs
  - pack 2 pairs into one fragment
  - lowest sorting pass runs internal in fragment shader
- Generate keys according to distance to viewer or use cell identifier of space partitioning scheme



initial pass | collapse into single pass | third pass | collapse into single pass

Fast sorting

- Same approach for column sort, just rotate the quads
- Benefits for full sort of n items
  - 2*log(n) less passes (because of collapse and packing)
  - n/2 fragments processed each pass (because of packing)
  - workload balanced between vertex and fragment units (because of rendering quads and interpolation)

→ Speedup factor of 10 compared to previous solutions

Fast sorting

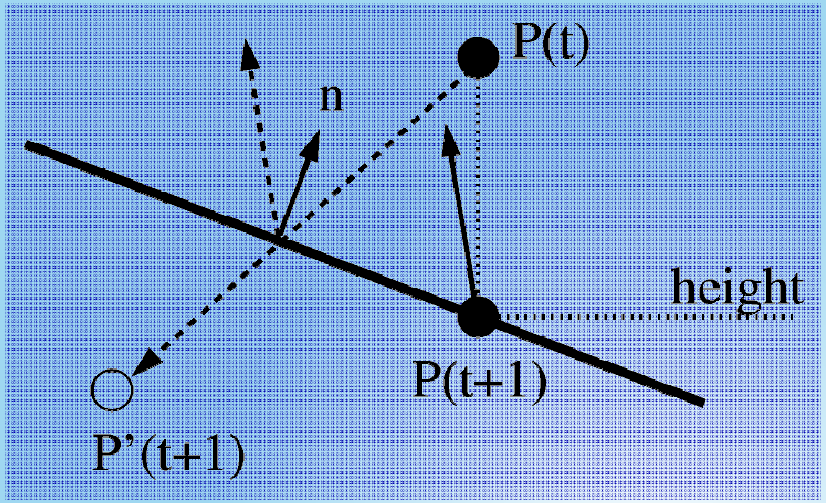
- Performance: full sort

n	*sorts/sec*	*mega items/sec*	*mega frag/sec*	
128^2	175.0	2.8	130	ATI Radeon 9800Pro
256^2	43.6	2.8	171	
512^2	9.3	2.4	186	
1024^2	1.94	2.0	193	
128^2	238.0	3.9	177	ATI Radeon X800 XT
256^2	110.0	7.2	433	
512^2	24.4	6.4	489	
1024^2	4.85	5.1	483	

Particle – Scene Collision

Additional buffers for state-full particles

- Store velocity per particle (Euler integration)
- Keep last two positions (Verlet integration)
- Simple: Collision with height-field stored as 2D texture
  - RGB = [x,y,z] surface normal
  - A = [w] height
  - Compute reflection vector
  - Force particle to field height

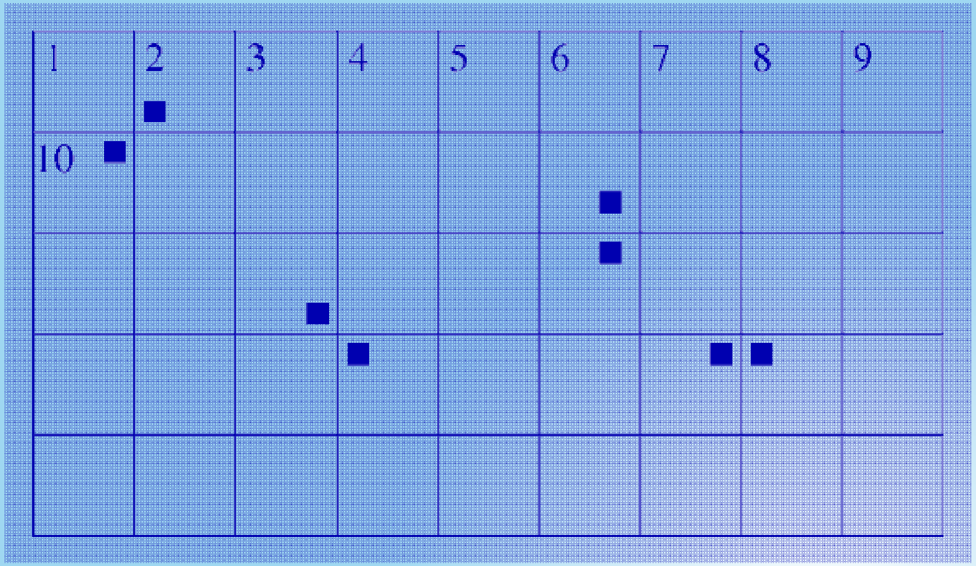


Particle – Particle Collision

Essential for natural behavior

- Full search is O(n²), not practicable
- Approximate solution by considering only neighbors
- Sort particles into spatial structure
  - Staggered grid misses only few combinations

single grid

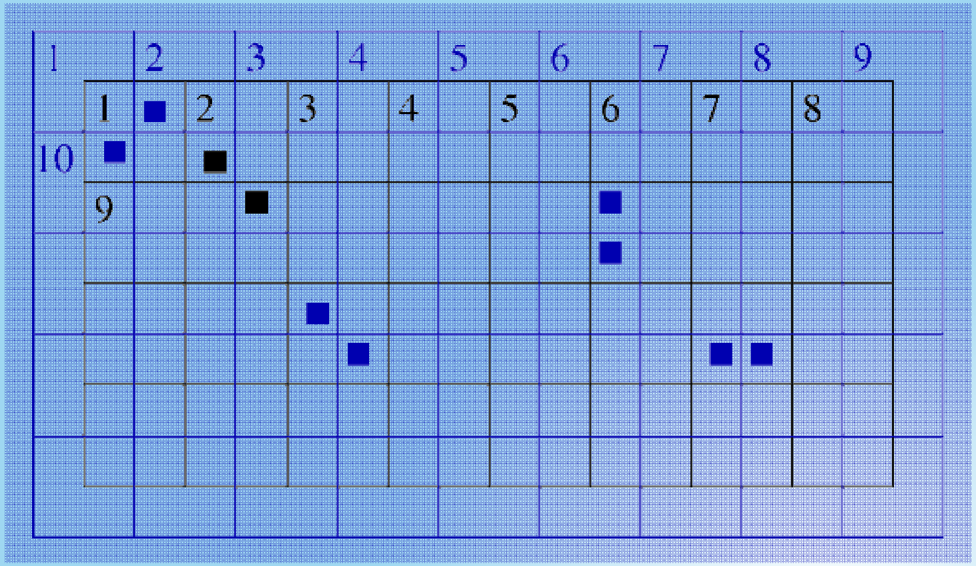


Particle – Particle Collision

Essential for natural behavior

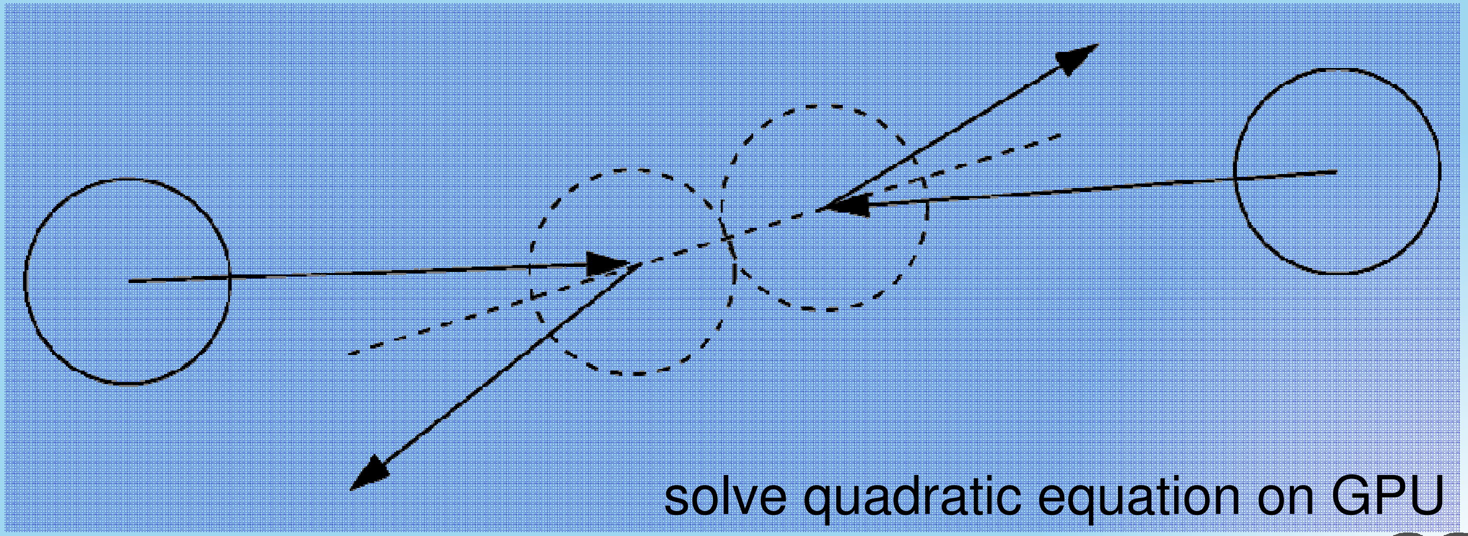
- Full search is O(n²), not practicable
- Approximate solution by considering only neighbors
- Sort particles into spatial structure
  - Staggered grid misses only few combinations

staggered grid



Particle – Particle Collision

- Check m neighbors to the left/right
- Collision resolution with first collider (time sequential)
- Only if velocity is not excessively larger than integration step size



GPU Particle Engine

more cool demos

GPU Particle Engine

Acknowledgements

- ATI Research for providing hardware
- Jens Krüger for insight on shader programming

http://wwwcg.in.tum.de/GPU