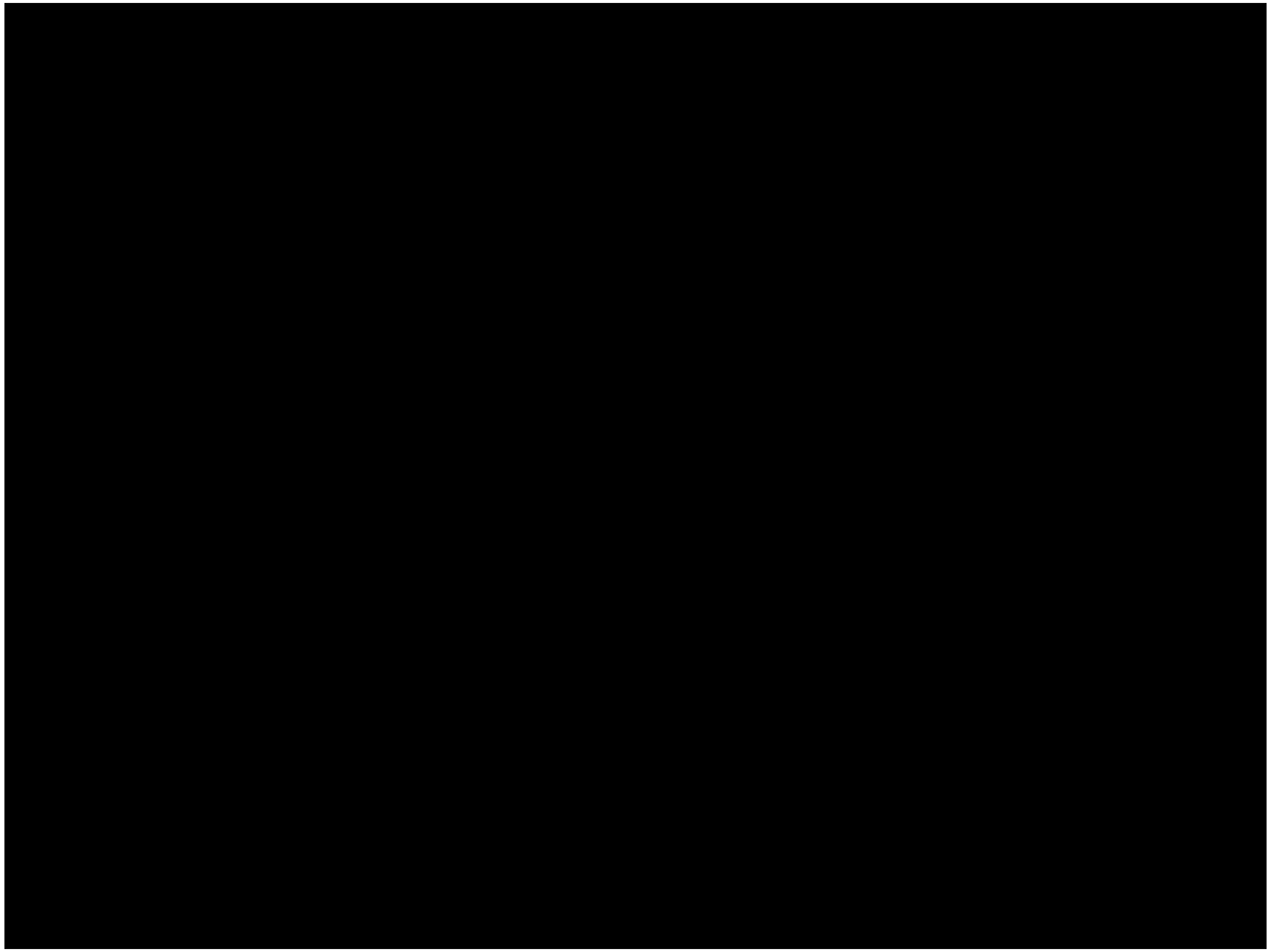
SPH Neighborhood Search (and Time Step)

Matthias Teschner

Computer Science Department
University of Freiburg

Motivation



1.7 million fluid particles

341 million particle pairs are processed per simulation step

Motivation

12 million fluid particles, 5 million boundary particles

2.3 billion particle pairs are processed per simulation step

5.2 s for neighborhood search

Outline

- neighborhood search in SPH
- uniform grid
- index sort
- z-index sort
- spatial hashing
- compact hashing
- results

SPH Simulation Step Using a State Equation

- foreach particle do
  - compute density
  - compute pressure
- foreach particle do
  - compute forces
  - integrate

- density and force computation process all neighbors of a particle

Neighbor Search Characteristics

- efficient construction and processing of dynamically changing neighbor sets is essential
- neighbor search requires fast access
  - to the cell of a particle
  - to all adjacent cells of a particle's cell
- temporal coherence should be employed
- spatial locality should be preserved

- hierarchical data structures are less efficient in this context
  - construction in O (n log n), access in O (log n)
- uniform grid is generally preferred
  - construction in O (n), access in O (1)

Uniform Grid Implementations

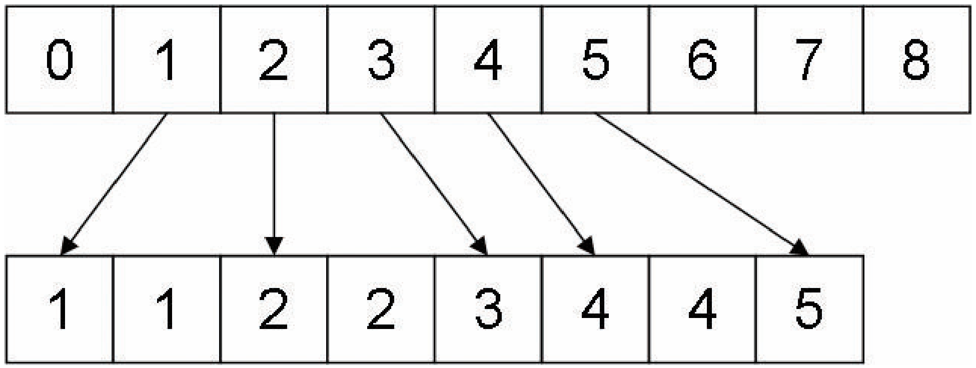
- basic grid
- index sort
- z-index sort
- spatial hashing
- compact hashing

Basic Grid

- particle is stored in a cell with coordinates (k, l, m)
- 27 cells are queried in the neighborhood search (k±1, l±1, m±1)
- cell size equals the influence radius of a particle
  - larger cells increase the number of tested particles
  - smaller cells increase the number of tested cells
- parallel construction suffers from race conditions
  - insertion of particles from different threads in the same cell

Index Sort Construction

- cell index $c = k + l \cdot K + m \cdot K \cdot L$ is computed for a particle
  - K and L denote the number of cells in x and y direction
- particles are sorted with respect to their cell index
  - radix sort, O(n)
- each grid cell (k, l, m) stores a reference to the first particle in the sorted list



uniform grid

sorted particles with their cell indices

Index Sort
Construction

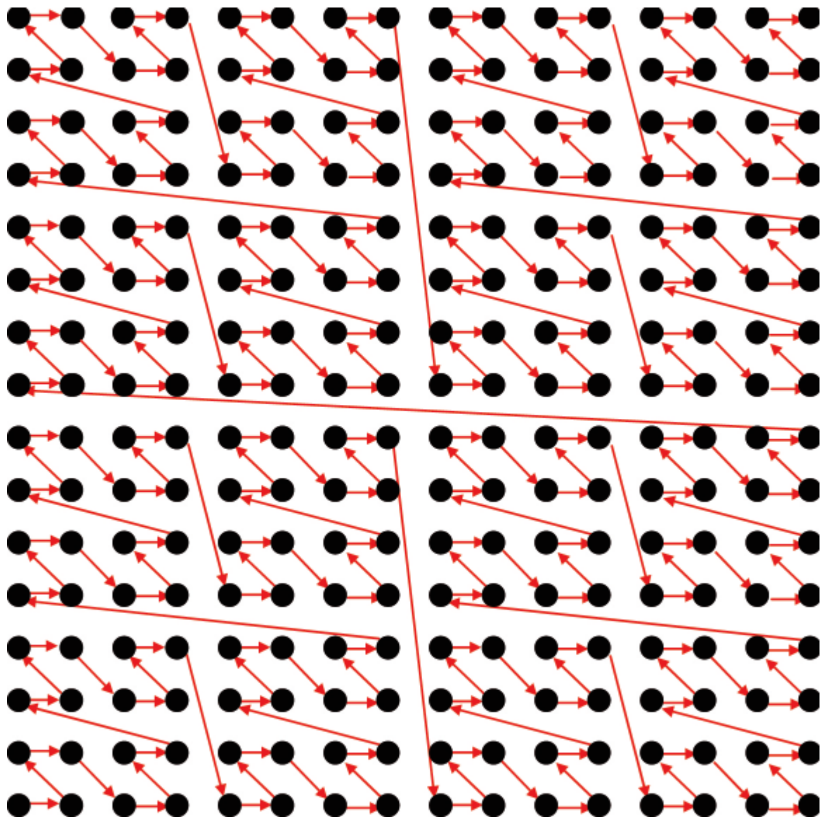
- parallelizable
- memory allocations are avoided
- constant memory consumption
- entire spatial grid has to be represented to find neighboring cells

Index Sort
Query

- sorted particle array is queried (parallelizable)
- particles in the same cell are queried
- references to particles of adjacent cells are obtained from the references stored in the uniform grid
- improved cache-hit rate
  - particles in the same cell are close in memory
  - particles of neighboring cells are not necessarily close in memory

Z-Index Sort

- particles are sorted with respect to a z-curve index
- improved cache-hit rate
  - particles in adjacent cells are close in memory
- efficient computation of z-curve indices possible



z-curve

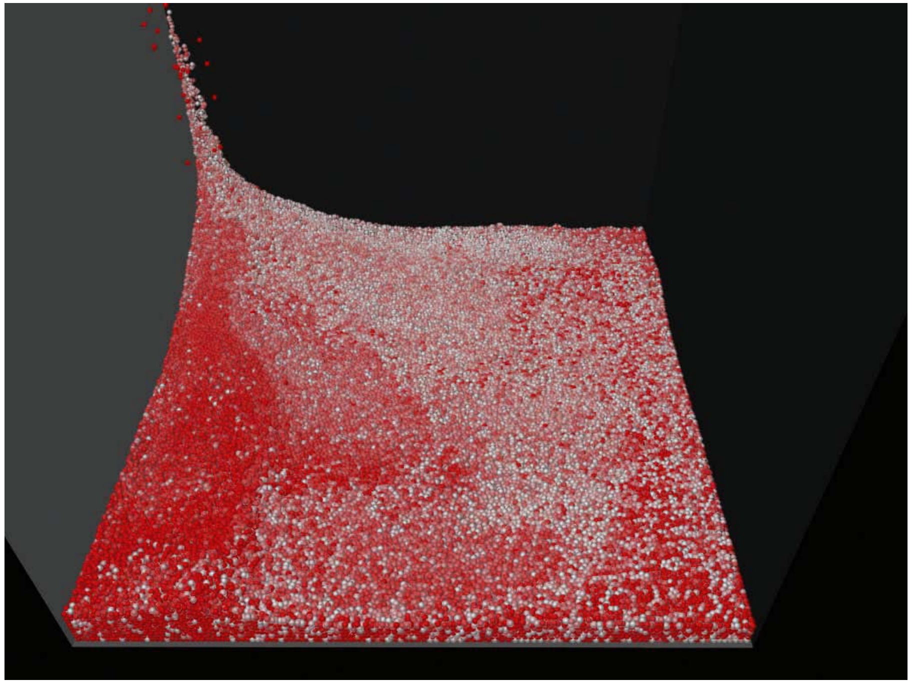
Z-Index Sort
Sorting

- particle attributes and z-curve indices are processed separately
- handles (particle identifier, z-curve index) are sorted in each time step
  - reduces memory transfer
  - spatial locality is only marginally influenced due to temporal coherence
- attribute sets are sorted every 100th simulation step
  - restores spatial locality

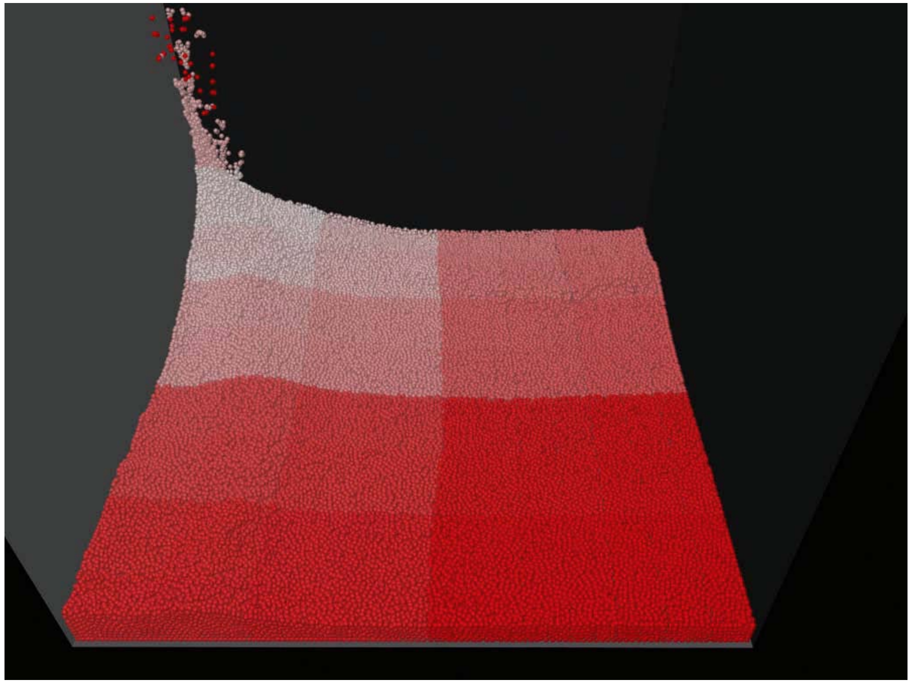
Z-Index Sort
Sorting

- instead of radix sort, insertion sort is employed
  - O (n) for almost sorted arrays
  - due to temporal coherence, only 2% of all particles change their cell, i. e. z-curve index, in each time step

Z-Index Sort Reordering



particles colored according to their location in memory



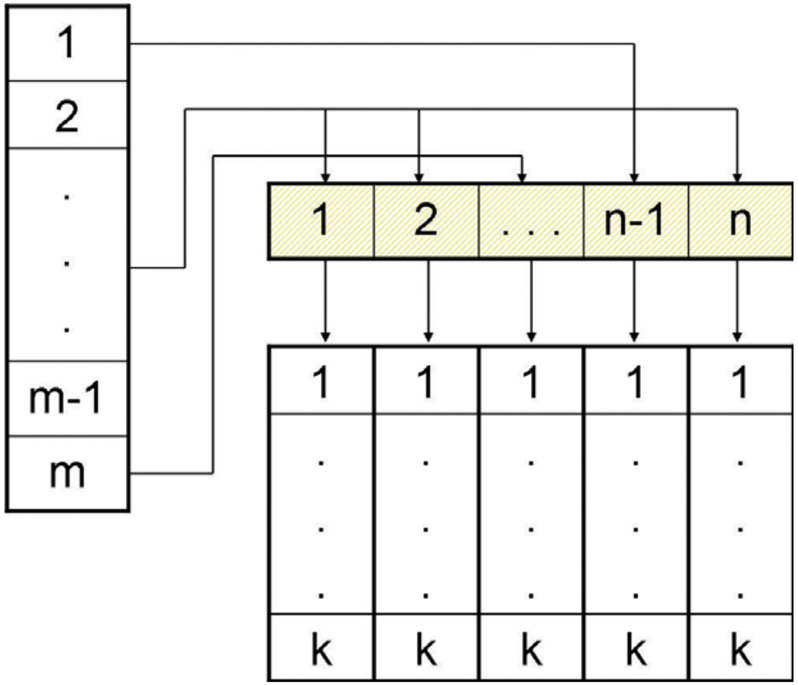
spatial compactness is enforced using a z-curve

Spatial Hashing

- hash function maps a grid cell to a hash cell
  - infinite domain is mapped to a finite list
  - in contrast to index sort, infinite domains can be handled
- large hash tables reduce number of hash collisions
  - hash collisions occur, if different spatial cells are mapped to the same hash cell
  - hash collisions slow down the query
- reduced memory allocations
  - memory for a certain number of entries is allocated for each hash cell
- reduced cache-hit rate
  - hash table is sparsely filled
  - filled and empty cells are alternating

Compact Hashing

- hash cells store handles to a compact list of used cells
  - k entries are pre-allocated for each element in the list of used cells
  - elements in the used-cell list are generated if a particle is placed in a new cell
  - elements are deleted, if a cell gets empty
- memory consumption is reduced from O (m · k) to O (m + n · k) with m » n
- list of used cells is queried in the neighbor search



Compact Hashing Construction

- not parallelizable
  - particles from different threads might be inserted in the same cell
- larger hash table compared to spatial hashing to reduce hash collisions
- temporal coherence is employed
  - list of used cells is not rebuilt, but updated
  - set of particles with changed cell index is estimated (about 2% of all particles)
  - particle is removed from the old cell and added to the new cell (again not parallelizable)

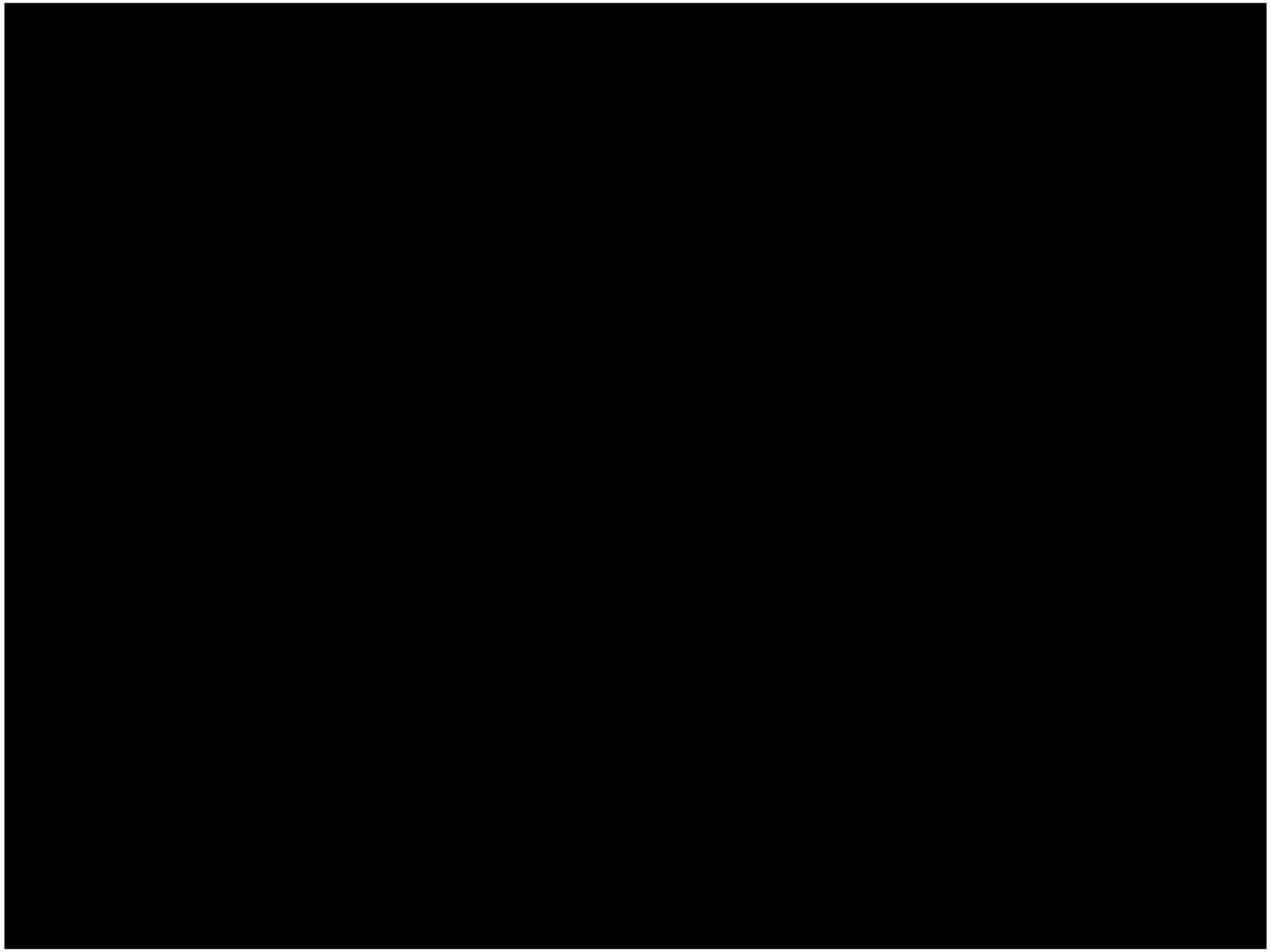
Compact Hashing Query

- processing of used cells
  - bad spatial locality
  - used cells close in memory are not close in space
- hash-collision flag
  - if there is no hash collision in a cell, hash indices of adjacent cells have to be computed only once for all particles in this cell
  - large hash table results in 2% cells with hash collisions

Compact Hashing Query

- particles are sorted with respect to a z-curve every 100th step
- after sorting, the list of used cells has to be rebuilt
- as particles are serially inserted into the list of used cells, the list is consistent with the z-curve
  - improved cache hit rate during the traversal of the list of used cells

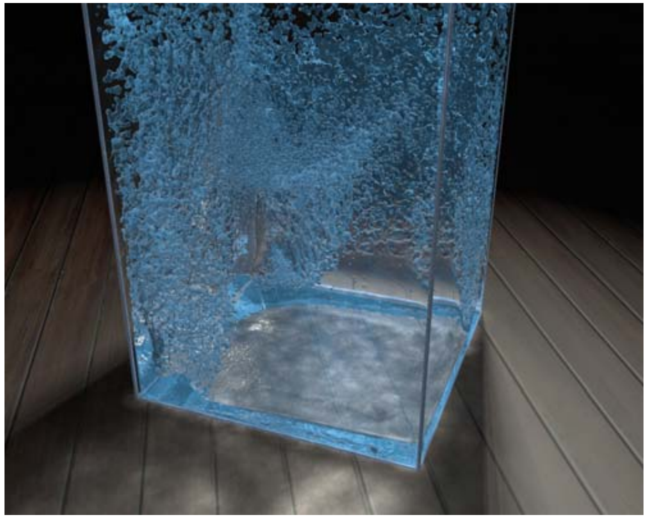
Compact Hashing Reordering



Comparison

method	construction	query	total
basic grid	26 (27)	38 (106)	64 (133)
index sort	36 (38)	29 (30)	65 (68)
z-index sort	16 (20)	27 (30)	43 (50)
spatial hashing	42 (44)	86 (90)	128 (134)
compact hashing	8 (9)	32 (55)	40 (64)

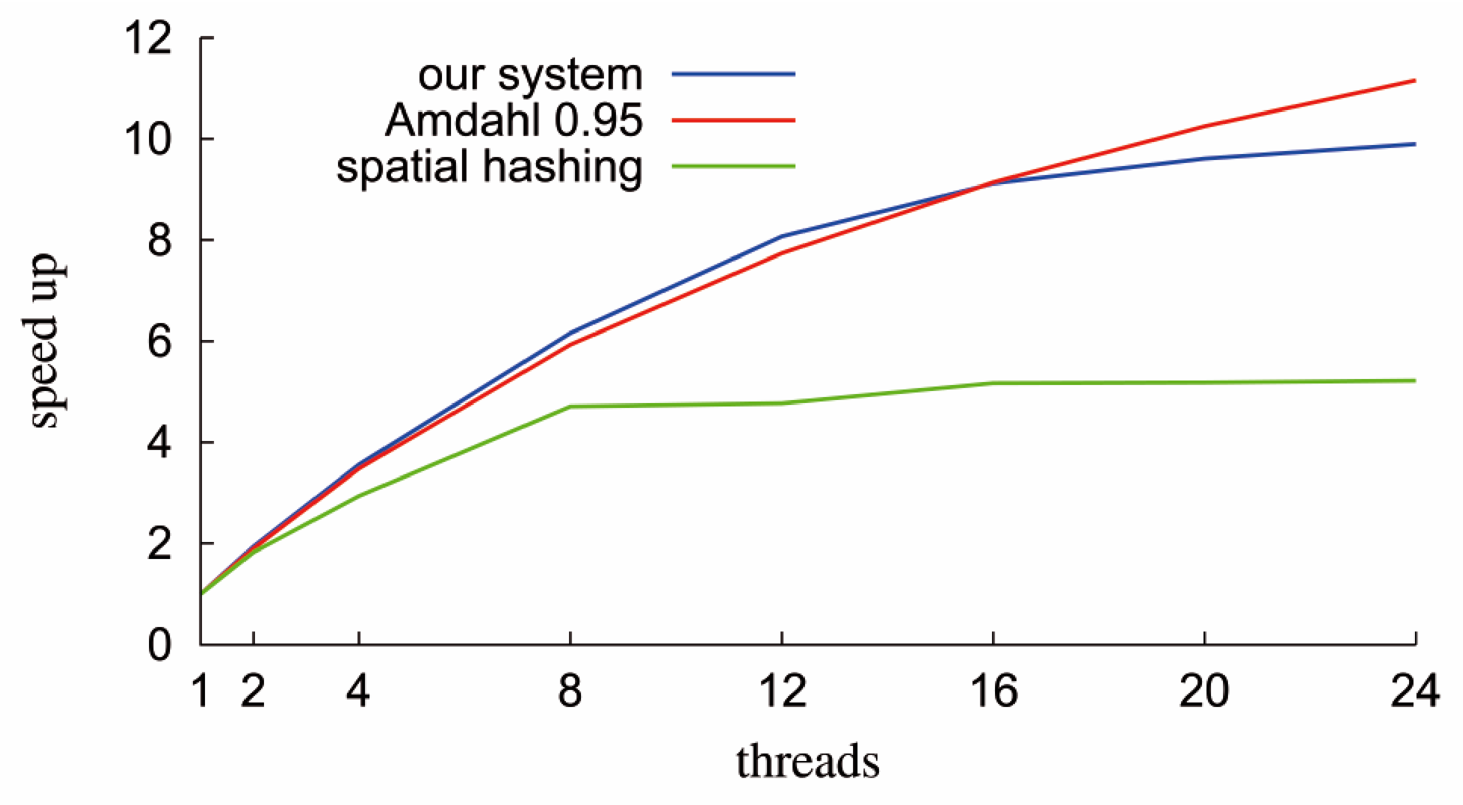
- measurements in ms for 130K particles on a 24-core computer with 128 GB RAM
- with reordering and (without reordering)



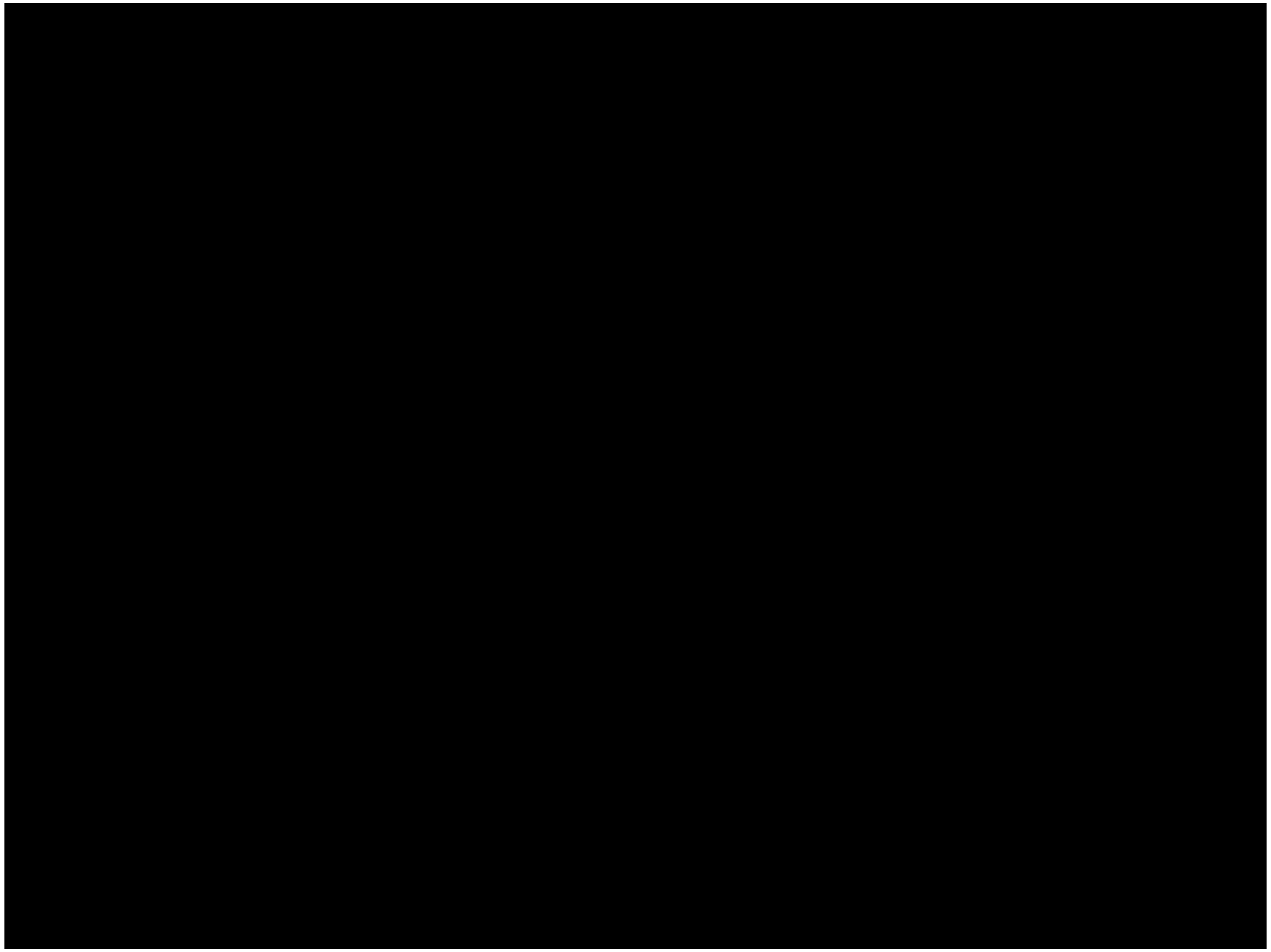
Discussion

- index sort
  - fast query as particles are processed in the order of cell indices
  - slow construction due to sorting
- z-index sort
  - fast construction due to insertion sort of an almost sorted list
  - sorting with respect to the z-curve improves cache-hit rate
- spatial hashing
  - slow query due to hash collisions and due to the traversal of the sparsely filled hash table
- compact hashing
  - fast construction due to temporal coherence
  - fast query due to the compact list of used cells and due to the hash-collision flag

Parallel Scaling



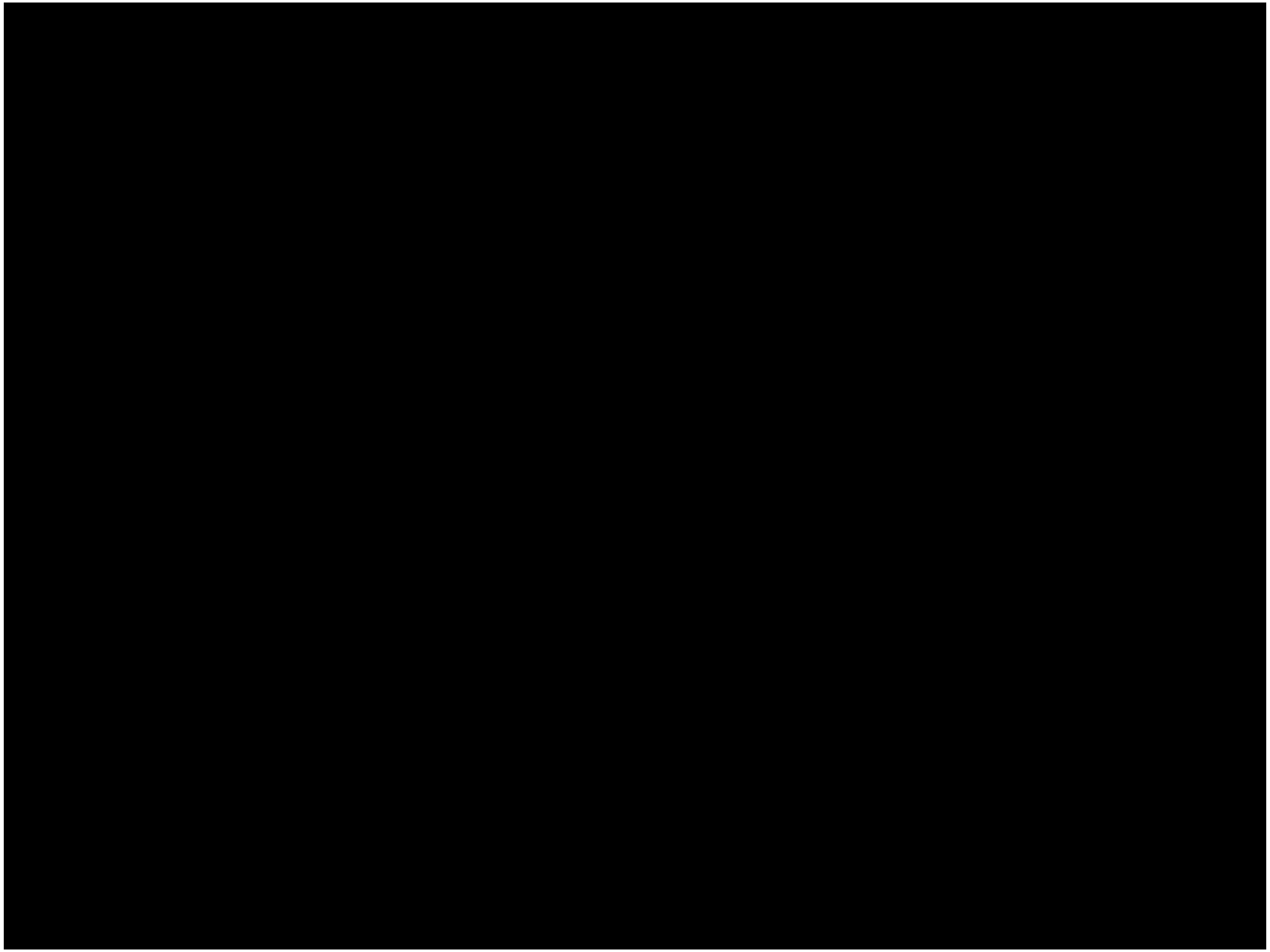
Result



75k fluid particles

4 min computation time

Result



Summary

- neighborhood search in SPH
- uniform grid
- index sort
- z-index sort
- spatial hashing
- compact hashing
- results

References

- index sort
  - PURCELL T. J., DONNER C., CAMMARANO M., JENSEN H. W., HANRAHAN P.: Photon Mapping on Programmable Graphics Hardware. *ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware,* 2003.
- spatial hashing
  - TESCHNER M., HEIDELBERGER B., MÜLLER M., POMERANETS D., GROSS M.: Optimized Spatial Hashing for Collision Detection of Deformable Objects. *Vision, Modeling, Visualization* 2003.
- z-index sort, compact hashing
  - IHMSEN M., AKINCI N., BECKER M., TESCHNER M.: A Parallel SPH Implementation on Multi-core CPUs. *Computer Graphics Forum*, accepted.

SPH
Time Step

Matthias Teschner

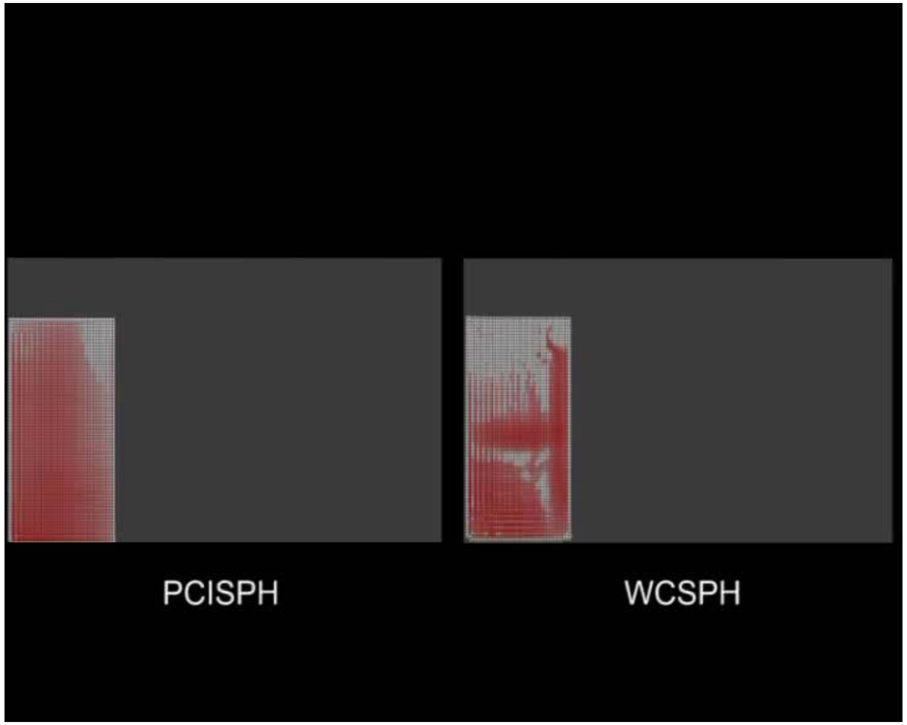
Computer Science Department
University of Freiburg

Outline

- pressure computation
- boundary handling
- adaptive time stepping

Pressure Computation

- Predictor-corrector (PCISPH)
  - [Solenthaler 2009]
  - iterative pressure computation
  - large time step
- Tait equation (WCSPH)
  - [Becker and Teschner 2007]
  - efficient to compute
  - small time step



- computation time for the PCISPH scenario is 20 times shorter than WCSPH

SESPH

- foreach particle do
  - compute density
  - compute pressure
- foreach particle do
  - compute forces
  - integrate

- neighbor sets are processed two times

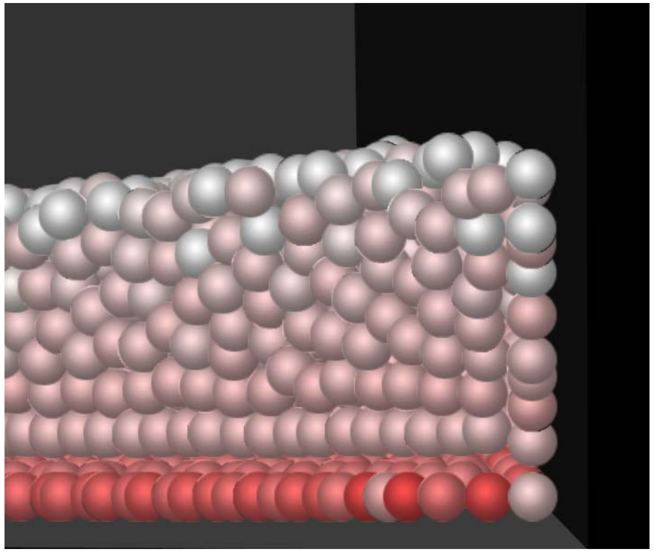
PCISPH

- foreach particle do
  - compute forces
  - set pressure and pressure force to zero
- while (max(ρ_{err}) > η) or number of iterations < 3) do
  - foreach particle do
    - predict velocity and position
  - foreach particle do
    - update distances to neighbors
    - predict density variation
    - update pressure
  - foreach particle do
    - compute pressure force
- foreach particle do
  - update position and velocity

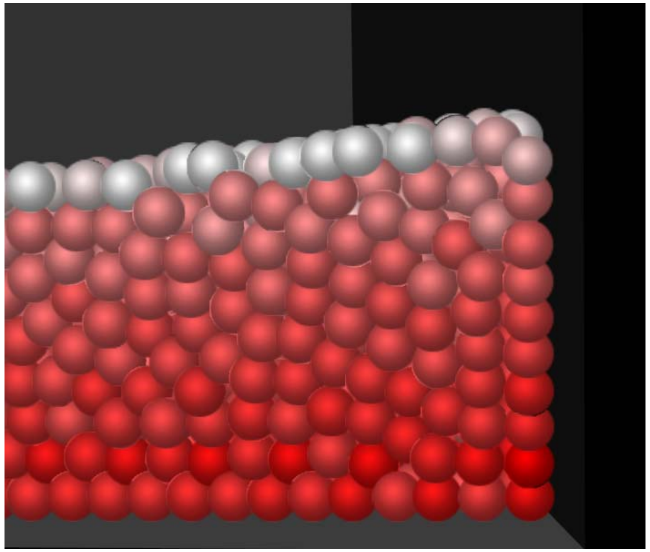
- neighbor sets are processed at least seven times

Boundary Handling

- is a limiting factor for the time step
  due to a potentially non-homogenous pressure distribution



[Becker et al., IEEE TVCG 2009]

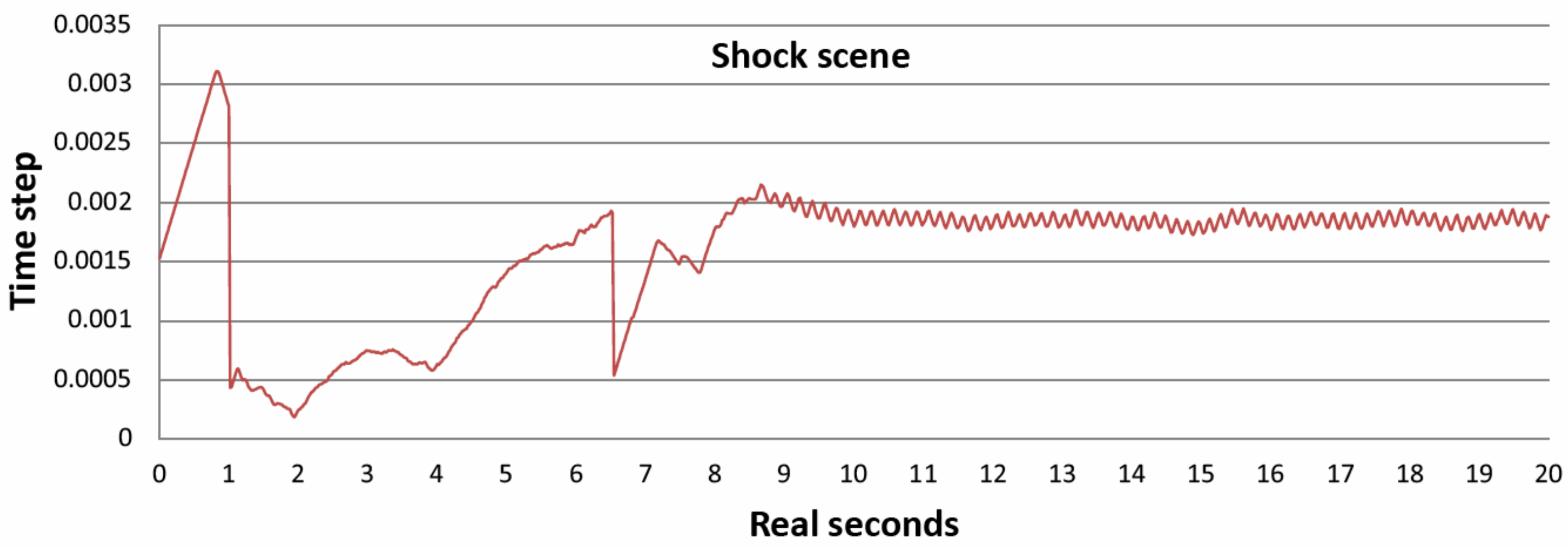


[Ihmsen et al., VRIPHYS 2010]

color indicates pressure

Adaptive Time Stepping

- small time step is required only for short time periods
- difficult to pre-estimate the time step
- significant speed-up of the overall computation time due to adaptive time-stepping



Adaptive Time Stepping