Vision, Modeling, and Visualization (2010)

Topology-Caching for Dynamic Particle Volume Raycasting

Jens Orthmann, Maik Keller and Andreas Kolb

Computer Graphics Group, Institute for Vision and Graphics (IVG), University of Siegen, Germany

Abstract

*In this paper we present a volume rendering technique for the ad-hoc visualization of interactive particle systems. We focus on methods for an efficient spatial caching (*topology caching*) of particles when applying a raycasting approach. Thus, we get a fast reconstruction of the scalar field which is defined by the particles' entities.*
The node-cache *allows for efficient caching and pre-fetching of a subset of the octree nodes. The* influence-cache *provides fast access to all particles which contribute to a specific node including level-of-detail particles. Finally, the introduced* slab-cache *allows for efficient volume rendering and gradient computation. Our algorithms are completely built and managed on the GPU and interactive frame rates for up to several 10^5 particles are achieved.*

Categories and Subject Descriptors (according to ACM CCS): Volume Raycasting [I.3.7]: Particle Visualization—

1. Introduction

Traditionally, there are two different approaches to represent spatial information in fluid simulations: grid-based (*Eulerian*) or particle-based (*Lagrangian*). Having a fixed spatial relation, Eulerian approaches have advantages regarding spatial resolution and topology. In Lagrangian techniques, however, the "data elements" do not have a fixed spatial relation, and so these approaches are spatially more flexible. The advantage of self-adaption comes at the expense of an irregular particle distribution which makes any kind of further processing more complex.

In this paper we aim at interactive, high quality volume visualizations for large dynamic particle sets, as they arise from *Smoothed Particle Hydrodynamics (SPH)* [GM77], for example. In Lagrangian simulations, particles are the carriers of physical flow properties which are advected with the flow field (see Koumoutsakos et al. [KCR08] for an overview). Current ad-hoc rendering techniques mainly concentrate on the rendering of the fluid surface only. However, the flux profile of fluid quantities applies to the whole particle set (see Figure 1 for the visualization of advection-diffusion equations). Such distributed quantities require volume rendering techniques in order to be visualized correctly.

For raycasting, just as for any other kind of processing, the continuous scalar field over the particle set N_x of all discrete

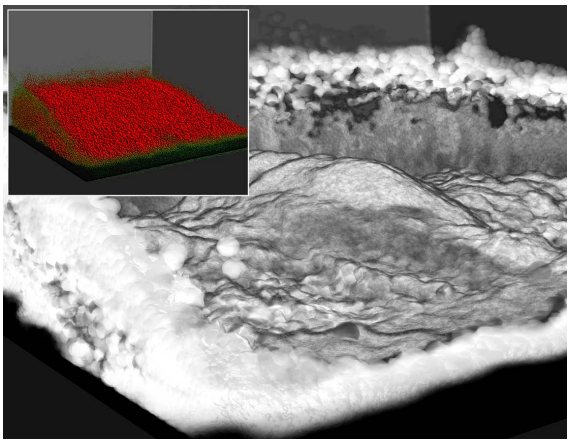

Figure 1: *Our GPU-based visualization algorithm generates high quality images of dynamic particle simulations at interactive frame rates. The visualization is directly applied after each simulation step. The result shows the particle diffusion layer within an SPH dam break simulation:* 130*k particles,* 25*k nodes, rendering at* 800*x*600*pixels at* $\sim$ 10*fps. The small picture shows a point sprite rendering of the simulation.*

DOI: 10.2312/PE/VMV/VMV10/147-154

field quantities q_i is evaluated at a specified ray position $\mathbf{x}$ in space:

$$q(\mathbf{x}) = \frac{\sum_{\mathbf{p}_i \in N_x} q_i K(\|\mathbf{x}-\mathbf{p}_i\|, h)}{\sum_{\mathbf{p}_i \in N_x} K(\|\mathbf{x}-\mathbf{p}_i\|, h)}, \quad (1)$$

where K is a radial symmetric kernel function centered at the particle positions $\mathbf{p}_i$ with finite support radius h. There are two computational strategies for the reconstruction of $q(\mathbf{x})$ as each particle has only a finite influence radius: With *scattering* [ZSP08] the contributions of a particle are pushed to each point of evaluation. Since these splats in general have a non-distinct support, this introduces memory collisions which require expensive atomic operations. *Gathering approaches* [SDG08], on the other hand, collect the contributions from neighboring particles for a given point of evaluation. This approach fits better to the newly introduced GPU streaming architecture but it requires a fast access mechanism to neighboring particles.

In this paper, we propose a volume raycasting approach which utilizes *data-parallel octrees* proposed by Zhou et al. [ZGHG10]. We enhance this acceleration data structure for the use of various caching mechanisms in order to gather the information of neighboring tree nodes efficiently. With this new, fully GPU-based approach we add the following contributions:

- The *node cache* allows for caching and pre-fetching of a local subset of octree nodes for an accelerated ray traversal during the volume raycasting. A multi-level particle hierarchy is set up during the octree construction by aggregating particle data to coarser octree levels. This approach allows for efficient level-of-detail raycasting and anti-aliasing.
- The *gathering*-principle is efficiently realized by the redundant neighborhood assignment of particles for a fast access to all particles that contribute to a specific node; this structure is called *influence cache*. The approach is also capable of handling adaptive particle support sizes.
- In order to reduce the large number of memory read-operations even more we employ a local *slab cache* following Mensmann et al. [MRH10]. With this cache particles can contribute to more than one ray position within each lookup. Additionally, gradients can be computed efficiently.

This paper is structured as follows: We start with an overview of the previous work in Sec. 2, followed by a conceptual overview of the raycasting system (Sec. 3). A detailed description of the respective caching mechanisms is presented in Sec. 4, 5 and 6. In Sec. 7 the results are evaluated. Finally, Sec. 8 concludes this paper and comments on future work.

2. Related work

The ad-hoc visualization of particle simulations mainly focuses on the reconstruction of the particle surface. Zhang et al. [ZSP08] solve the equation for metaballs for close particles. They seperate the metaballs to non-overlapping groups. This allows them to apply the smooth kernel interpolation in the image space as a post processing pass [ZP07]. However, this grouping mechanism cannot be applied to volume rendering as inner particles are clipped away. Van der Laan et al. [vdLGS09] splat particles without explicitly sorting the input stream. Being related to the volume rendering integral they track the overdraw at each pixel position as an approximation for the volume's thickness. Zhou et al. [ZGHG10] apply a marching cube based method after spatial organization of the particles.

State-of-the-art volume raycasting techiques, as summarized in Hadwiger et al. [HLSR08], usually implement the well-known volume rendering integral along rays in order to accurately visualize semi-transparent datasets. These techniques, which are constrained by the rasterization pipeline, have been translated to programming interfaces for stream processing [MHS08] which are better suited for raycasting. Mensmann et al. [MRH10] utilize the on chip memory to cache adjacent values of a slab along a small group of rays. With shared samples they are able to compute gradients without re-reading isovalues which have already been calculated. However, their methods deal only with Eulerian grids. Parallel to our work Fraedrich et al. [FAW10] have introduced a volume raycasting approach for SPH-based particle data. However, our technique is applied directly as an ad-hoc visualization of a running particle simulation.

Splatting techniques as first introduced by Westover [Wes90] handle an unstructured point data set directly. By sheet splatting, particles are projected onto view aligned [NMM*06] or axis aligned slices [SP09], before the final composition. The resulting footprint images are then composited front to back to the final image. This requires to sort the particles whenever the ordering of splats becomes invalid. Aliasing artefacts may occure due to the undersampling of the respective particle data over diverging rays which then makes prefiltering necessary [ZPvBG02]. Splatting is the state of the art technique to visualize particles very efficiently. However, in contrast to GPU-based splatting techniques, where texture-based implementations are used, we perform a direct raycasting approach.

Kd-trees are usually the preferred spatial data structure for raytracing of point-based models [ZHWG08]. Popov et al. [PGSS07] have introduced a stackless method for static scenes in order to eliminate the node traversal along ray-packets. However, octrees are better suitable for neighborhood queries because of their simple structure [ZGHG10]. This is important for the direct volume raycasting of unstructured points. Linsen et al. [LLRR08] spread the particle data over the octree nodes for faster access of neighbors at each ray position. Guennebaud et al. [GGG08] build a redundant octree on the GPU in order to efficiently handle dynamic point set sur-

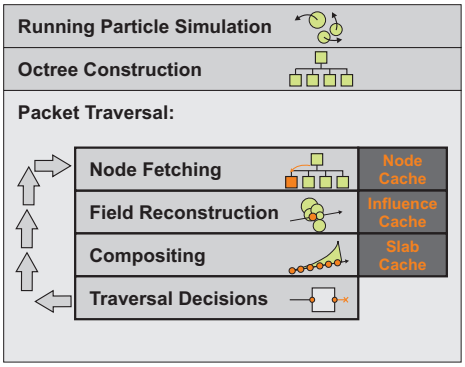

Figure 2: *Overview: the basic components of the volume raycasting algorithm. The visualization is applied immediately after the update of the unstructured particle set.*

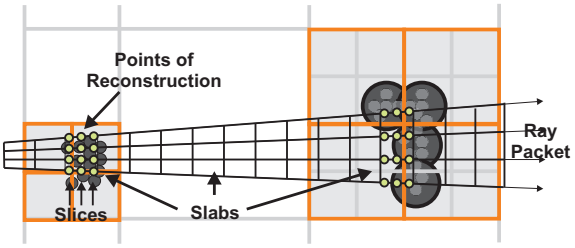

Figure 3: *The slab based ray casting with four rays per packet and three slices per slab. The image shows the fetched octree nodes for two specific slabs. At distance, particles on the finest level are replaced by coarser particles.*

faces. Besides the memory consumption of their full octree they have to use mutex strategies to write the point updates which in case of surface aligned points cause only a small number of conflicts. However, for non-surface point clouds this has a strong impact on the final performance. Zhou et al. [ZGHG10] introduced a fast GPU octree algorithm for point sets with direct access to neighbor nodes. We build upon this system and extend it in various ways.

3. Volume Raycasting

This section introduces our approach to the raycasting algorithm and its various improvements. The basic steps are illustrated in Fig. 2. Our system aims at a very fast and efficient reconstruction of the unstructured particle set which is dynamically updated, e.g. by lagrangian simulation. This means that the visualization of the particle set is performed directly after each simulation step. For this reason we utilize several caching strategies which are explained in detail in the following sections.

Octree Construction: According to Equation 1, the reconstruction of a particle set requires fast access to neighboring particles. Thus, the unstructured particle set needs a spatial organization which is usually done by the utilization of a fast spatial data structure. This data structure has to be rebuild in each frame, since the particle set is dynamically updated. Our specific implementation of the data structure follows the data parallel octrees introduced by Zhou et al. [ZGHG10]. They developed a technique for octree construction on the GPU which builds octrees in real-time and uses level-order traversals to exploit the parallelism of the GPU. We refer to Zhou et al. for implementation details concerning the basic octree construction.

Packet Traversal: The reconstruction of the particle set needs many memory read operations in order to access all relevant particles which contribute to the final field reconstruction. The limitation by memory read operations can be avoided by the use of coherent memory access and coherent branch decisions of adjacent rays. This is achieved by a packet traversal which consists of a bunch of several adjacent rays which access nearly the same particle subset for the field reconstruction. All rays of a packet are traversed with the same step size through the particle set. This means, the viewing plane is subdivided into rectangular regions which define the origin of the packet's rays.

Node fetching: In the first step of the algorithm's main loop we advance the rays which defines the discrete points of the field reconstruction. Then we need to identify all relevant octree nodes which contain the particle subset for the whole ray packet. Naive implementations would result in redundant tree traversals within each packet. We solve this problem by introducing the node-cache (see Sec. 4) which supports a node pre-fetching mechanism for all rays of a packet.

Field reconstruction: The reconstruction of the particle field is done by following the gathering-principle: all contributions from neighboring particles are collected and evaluated for a specific position along the ray. The fast access to the particles is efficiently realized by providing a redundant neighborhood assignment of particles. This assignment is called influence-cache (see Sec. 5).

Compositing: The Rays accumulate field values during the traversal through the scene. In fact, the traversal is done in a slab-based manner instead of a simple packet traversal. Each slab consists of several slices along the viewing direction for a ray packet (see Mensmann et al. in [MRH10]). Fig. 3 shows an illustration of a slab. Similar to Mensmann et al. we take care of the starting points for each ray. We use a proxy geometry which indicates the entry point for the first slab into the particle set. Thus, all rays of a specific ray packet start their traversal with the smallest distance to the proxy geometry. The intermediate reconstruction results at the slab's positions are stored in a slab-cache (see Sec. 6). The usage of this cache results in the further reduction of memory read operations as particles can contribute to more than one ray position at once. Finally, the values of the slab-slices are composited in front-to-back order. Lighting tech-

niques can easily extend the standard composition algorithm by using the slab-cache.

Ray traversal decisions: Optimization strategies such as empty space skipping and tests for early ray termination are applied at the last stage of the main loop. Since the rays of a packet advance simultaneously through the particle set, all rays need to be synchronized. For early ray termination this is achieved by comparing the individual results of the compositing step among the rays. The packet traversal is stopped if all of the composited values are above a certain threshold. In the case of empty space skipping the empty nodes are skipped for the whole ray packet and the packet traversal is advanced to the next node intersection.

4. Node-cache

The rays of a packet define the discrete points for the field reconstruction. We need to traverse the tree in order to obtain all relevant tree nodes which contain the particle-subset at these positions. Data structures which implement stackless traversals are well suited for GPUs as shown by Popov et al. [PGSS07]. These trees are characterized by nodes which store pointers to their neighboring nodes. Thus, the nodes are directly linked to each other and the number of traversal steps is reduced. In particular, we organize the unstructured particle set in an octree which is based on the idea of data parallel octrees by Zhou et al. [ZGHG10]. This data structure follows the stackless implementation since all neighbors of a node are accessible by using the relative position of the nodes with respect to their parents and their parents' neighbors.

4.1. Stackless Traversal

The tree traversal of the raycasting algorithm extends the mechanisms introduced by Samet [Sam89], see Fig. 4 for an example. Since the rays of a packet are organized in a slab, the mid-point of the slab is calculated in order to determine the current position for the tree traversal. If this position requires a traversal to a new node we distinguish three cases then:

1. If the previously required node is at leaf-level and if a valid pointer to the neighboring node exists then the neighboring node is loaded (*c*).
2. If the pointer to the neighboring node is invalid (*a*), i.e. no neighboring node exists at leaf-level, then the tree is traversed in the upward direction following the parent links until a valid neighboring node at another tree-level has been found. Since this node must be an empty-node, i.e. it contains no particles, the node is perfectly suited for an empty space skipping algorithm.
3. If the previous node is at intermediate-level and if a valid pointer to the neighboring node exists (*b*) then the neighboring node is traversed following the child-pointers until a node at leaf-level has been reached (see Appendix A for details about a node's spatial correlation and the traversal of its children). If there is not any node at leaf-level available then the node closest to the leaf-level is loaded which is also suited for an empty space skipping algorithm.

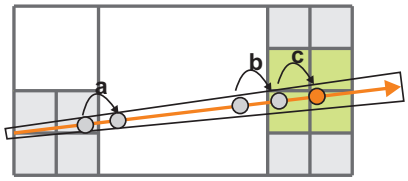

Figure 4: *Node traversal along different tree levels. If no direct neighbor address exists the tree hierarchy is traversed until the required node-level is reached. For each slab the relevant nodes (green) are stored in the node-cache.*

4.2. Pre-fetching

Each single ray would need to traverse its own local subset of nodes in order to perform a proper reconstruction along its positions. But this would result in a negative impact on the overall performance because an enormous number of memory read-operations is caused by multiple tree traversals. Since the tree traversal is done in a slab-based manner, all points for the field reconstruction are determined by the size of the slab and the number of its slices. This means, the spatial extend of the slab covers all the nodes which may contain relevant particles (green in Fig. 4). Therefore, these nodes are pre-fetched, and thus available to all rays of a slab for the succeeding field reconstruction (see Sec. 6 for more information about the field reconstruction). All pre-fetched nodes are loaded into the node-cache. In detail, the node-cache is filled according to the following steps:

1. The position of the slab's mid-point determines the *current leaf-node of interest*. Almost all rays of a slab would need exactly the particles assigned to this node for the field reconstruction.
2. However, some of the slab's border points for the field reconstruction may not been covered by the current leaf-node of interest. These positions need the particles which are assigned to a neighboring node in order to perform a proper field reconstruction. Therefore, each point of a slab needs to determine its respective node.
3. We restrict a slab's spatial extent to be smaller than the size of a leaf-node. Hence, a slab covers a maximum of eight nodes in total (in three-dimensional space). The current leaf-node of interest as well as all relevant neighboring nodes are loaded into the node-cache. The use of stackless tree traversals results in a very efficient pre-fetching of the nodes which are covered by the slab's extent.

As a result, each ray of a slab utilizes the node-cache for the field reconstruction.

4.3. Level-of-Detail

The spatial extend of a slab may cover more than the size of a leaf-node, especially at large distances to the view plane. Then, the tree traversals need to stop at a specific level-of-detail (see the slab on the right side in Fig. 3). Additionally, the level-of-detail may be defined by the user due to performance considerations. Therefore, we perform a bottom-up construction of a particle level-of-detail hierarchy where particles are added to the next coarser tree-level and merged into larger particles according to Hong et al. [HHK08] (see also Fig. 5). Thus, aliasing artifacts [ZPvBG02] are avoided when nodes are pre-fetched from the various tree-levels with respect to their distance to the view plane. Our implementation of the data parallel octrees has been enhanced with the particle level-of-detail hierarchy. The particle quantities are also accumulated according to Hong. The level-of-detail particles are propagated through the tree hierarchy.

5. Influence-cache

In Sec. 4.2, we explain the principles of finding the minimal subset of nodes for a slab-based field reconstruction in order to fill the node-cache. The octree implementation by Zhou et al. would exclusively assign a single particle to one specific octree node at the finest-level. But up to now we did not take the particle's finite support radius h into account (see Equation 1). This means if particles are close to the boundary of the node they are assigned to, then their support radius may overlap into neighboring nodes. Instead of pre-fetching only the nodes which are covered by the spatial extend of a slab we actually would have to load their neighboring nodes into the node-cache, too. This would result in a much larger node-cache referencing many particles $\mathbf{p}_i$ which do not contribute to the points of evaluation for the field reconstruction $\mathbf{x}$ as they are outside of the points' support radius: $\|\mathbf{x}-\mathbf{p}_i\| > h$.

We avoid this disadvantage by using an extended algorithm of the octree construction: we introduce a redundant particle assignment to multiple nodes according to the particle's support radius. This technique leads to an algebraic expansion of each node. Now the node addresses its relevant particle subset directly. Particles are spread out over a maximum of eight neighboring nodes as the particle's maximum support radius is smaller than the node's size. For each node we utilize a reference array which stores pointers to the particles (red arrows in Fig.5) if the support radius of the particles is overlapping the node. We refer to this reference array as the influence-cache of a node. This spreading mechanism is applied to all the particles including the merged particles which are generated on coarser tree-levels of the particle level-of-detail hierarchy. The extensions of the octree are described in Appendix B.

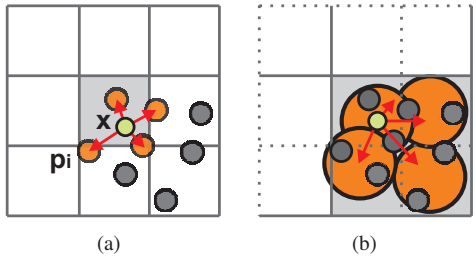

Figure 5: *The field reconstruction. The field is reconstructed at each point of evaluation (green). With the redundant particle assignment (red arrows), a single node (light grey) includes all relevant particles (orange). The particles of a tree-level (see 5(a)) are merged to larger particles at the next coarser tree-level (see 5(b))*

6. Slab-cache

The intermediate results of the field reconstruction for a single slab are stored in the so-called slab-cache. Therefore, we evaluate Equation 1 at each point of a slab. In detail, each point defined by the slices along a ray gathers the respective particle quantities q_i towards its position $\mathbf{x}$. The node-cache is used in order to access the particles for the evaluation of this position. Note, the influence-cache ensures that all the required particles N_x are referenced by the respective nodes. The algorithm can be optimized for adjacent points along a ray which require the same node. Then, the node's particles contribute directly to multiple points at a time. This reduces the number of particle read operations from global memory. Memory collisions are avoided since rays compute their own contributions without interfering adjacent rays. Finally, the resulting image is computed by compositing the slices of a slab in front-to-back order over all slabs.

With a slab-cache we are able to combine direct volume rendering with an isosurface rendering as described by Mensmann et al. [MRH10]. The gradient computation takes place in eye space and utilizes the slab-cache. During gradient computation the step-size Δx of the central differences must be adjusted for each slice of a slab. This is necessary because adjacent rays are not parallel. Taking the current distance α to the view plane into account, this leads to: $\Delta x = \alpha \cdot \frac{w}{n}$ with the pixel-size w and the distance to the near plane n. The computation of gradients would lead to discontinuities at the borders of a slab due to the missing neighborhood samples. However, this problem is solved by overlapping the slabs by border-rays.

7. Results and Analysis

In this section we demonstrate the application of topology-caching for dynamic particle volume raycasting. A table with running-times of various simulation examples shows

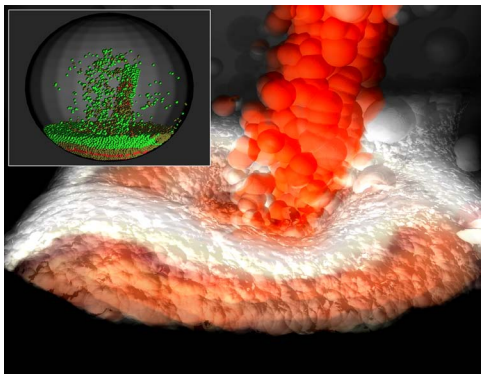

Figure 6: *Fountain simulation. The diffusion of two different fluids is visualized.*

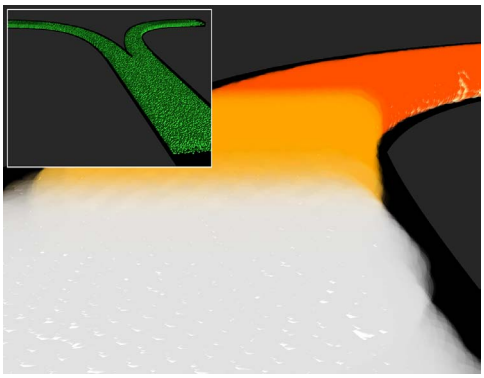

Figure 7: *T-Sensor simulation. The diffusion in a micro-chemical device is simulated and visualized.*

the speed-up which is achieved by using our enhanced octree data structure for raycasting purpose. The image quality is compared to a basic GPU raycaster with no explicit performance strategies. Additionally, the scalability of our algorithm is analyzed and discussed.

7.1. Examples

The algorithms described in the previous sections have been tested on an Intel Dual Core 2.67GHz CPU with a NVIDIA GeForce GTX 280 (1024MB) graphics card. For comparison reasons and scalability issues we have also tested our algorithms on a GeForce GTX 480 (1.5GB) graphics card. All algorithms are implemented using CUDA SDK 3.2. The raycaster has been configured with a slab-size of 8*x*8pixels and three slices for each slab. Note that the visualization is directly applied after each simulation step.

We tested our algorithms with a variety of dynamic SPH simulations as summarized in Table 1, see Figures 1, 6 and 7 for visual results. In the presented scenes the raycaster does not benefit much from the level-of-detail approach. But even at short distances to the viewport the influence on the performance is noticeable. If no influence-cache is applied then the number of read operations for particles increases significantly. Thus, the influence-cache has the highest impact on the performance. Furthermore, the node-cache as well as the slab-cache strongly increase the frame rate. Specifically, the slab-cache can be used efficiently for isosurface rendering. The small building-times of the octree are willingly accepted. This allows us to use the topology caches which enormously speed-up the whole raycasting process.

7.2. Image Quality

The topology-caching rendering approach results in a very efficient and fast rendering algorithm. However, the usage of ray-packets and also the utilization of the level-of-detail particles lead to an approximation of the final image. True,

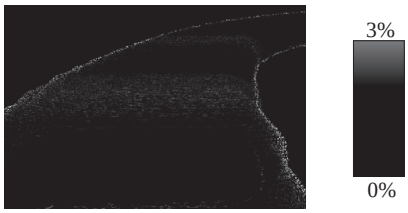

Figure 8: *Difference-image of the scene which is shown in Fig. 7: optimized raycaster vs. basic raycaster.*

this speeds up the rendering time. But this is at the cost of image quality. Fig. 8 displays the error image showing the differences of the optimized raycaster and the basic raycaster (ground truth) both rendering the scene of Fig. 7. The error is hardly noticeable and gray-scale coded. The white regions differ less than 3% from the dark regions. This means that the resulting error is almost zero and thus our optimized raycasting approach produces nearly the same results.

7.3. Discussion

The raycaster's slab-size has an enormous impact on the performance and efficiency of the visualization algorithm. The exact size of 8*x*8pixels has been chosen with respect to the size of the shared memory which is limited to 16KB for all threads residing in a streaming multi-processor. With our slab-based raycaster we reserve approximately 1900 bytes per packet as we have to store the results of the field reconstruction for 8*x*8*x*3 slab positions. As the results are weighted we have to reserve two times the memory of 768 bytes which results in 1536 bytes. Additionally, the node-cache requires 300 Byte of shared memory.

With this enormous shared memory consumptions the compiler cannot outsource registers to shared memory. This leaves us with 45 registers for each thread. With this high number of registers we can get an occupancy of only 31.3%.

Simulation	#steps	#ptcls	#nodes	*tree*	*tc_cast*	*no_nc*	*no_ic*	*no_sc*	*no_lod*	*basic*
T-Sensor	2048	50k	10k	11(7)	25(4)	132(27)	204(57)	103(16)	50(11)	634(201)
Fountain	512	60k	6k	15(5)	21(4)	251(31)	550(51)	230(9)	90(9)	971(205)
Dam	512	130k	25k	20(12)	45(7)	274(89)	510(109)	256(38)	88(19)	1115(530)

Table 1: *Running-time in milliseconds for the simulation examples. Please note that the simulations are dynamic and that the timings and numbers above are only valid for a specific frame (as shown in Figures 1, 6 and 7). #steps is the number of reconstruction steps per ray, #ptcls is the number of particles of the simulation, and #nodes is the number of octree nodes. The table shows the timings for building the tree, and the timings for the topology cached raycaster (tc_cast). Then, the timings are shown with various caches disabled: no node-cache (no_nc), no influence-cache (no_ic), no slab-cache (no_sc), and no application of the level-of-detail approach (no_lod). All the timings are taken with a NVidia GTX 280 (GTX 480). All examples use an octree of depth 8. The raycaster's screen resolution is* $800x600$*pixels with empty space-skipping and early ray-termination (*$\alpha = 0.99$*), and number of slices-per-slab=* 3*. For comparison reasons, the last column shows the timings for the basic raycaster (no tree and no caches).*

However, even with this low occupancy our topology cached raycaster is much faster then the basic raycaster (no caches activated) which has an occupancy of 50%.

We also tested our algorithm on a GTX 480 in order to show the scalability of the proposed methods. The shared memory has been enlarged to 48KB for a streaming multiprocessor. Thus, with the GTX480 we have a much better occupancy resulting in a better performance as shown in parentheses of Table 1.

8. Conclusion

We have presented a new acceleration structure for dynamic particle volume rendering. The resulting speed-up enables the visualization of a large number of particles at interactive frame rates on the GPU. The topology caching algorithms provide a consistent speedup compared to a standard tree structure due to the various caching abilities. The system is capable to handle the visualization of dynamic particle simulations out of core. No preprocessing is required and no assumptions about the particle simulation are necessary. There are several directions for future investigation: additional flow visualization may improve the visual results. The design of (semi-)automatic transfer functions for a dynamic particle simulation is a challenging topic, too. Todays current graphics hardware (GTX 480) is worth further investigation due to the scalability of our rendering approach in order to exploit the GPU's parallel architecture.

Acknowledgments

This work is partially funded by the Siegener Graduate School "Development of Integral Heterosensor Architectures for the n-Dimensional (Bio)chemical Analysis".

References

[FAW10] FRAEDRICH R., AUER S., WESTERMANN R.: Efficient high-quality volume rendering of sph data. *IEEE Transactions on Visualization and Computer Graphics (Proceedings Visualization / Information Visualization 2010) 16*, 6 (2010), to appear. 2

[GGG08] GUENNEBAUD G., GERMANN M., GROSS M. H.: Dynamic sampling and rendering of algebraic point set surfaces. *J. Computer Graphics Forum27*, 2 (2008), 653–662. 2

[GM77] GINGOLD R., MONAGHAN J.: Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Notices of the Royal Astronomical Society 181* (1977), 375–389. 1

[HHK08] HONG W., HOUSE D. H., KEYSER J.: Adaptive particles for incompressible fluid simulation. *Vis. Comput. 24*, 7 (2008), 535–543. 5

[HLSR08] HADWIGER M., LJUNG P., SALAMA C. R., ROPINSKI T.: Advanced illumination techniques for GPU volume raycasting. In *Proc. ACM SIGGRAPH Asia 2008* (2008). 2

[KCR08] KOUMOUTSAKOS P., COTTET G.-H., ROSSINELLI D.: Flow simulations using particles - bridging computer graphics and CFD, Sept. 01 2008. 1

[LLRR08] LINSEN L., LONG T. V., ROSENTHAL P., ROSSWOG S.: Surface extraction from multi-field particle volume data using multi-dimensional cluster visualization. *IEEE Trans. Vis. Comput. Graph 14*, 6 (2008), 1483–1490. 2

[MHS08] MARSALEK L., HAUBER A., SLUSALLEK P.: High-speed volume ray casting with cuda. In *Proceedings of IEEE Symposium on Interactive Ray Tracing* (2008), pp. 185–185. 2

[MRH10] MENSMANN J., ROPINSKI T., HINRICHS K. H.: An advanced volume raycasting technique using gpu stream processing. In *GRAPP: International Conference on Computer Graphics Theory and Applications* (Angers, 2010), INSTICC Press, pp. 190–198. 2, 3, 5

[NMM*06] NEOPHYTOU N., MUELLER K., MCDONNELL K. T., HONG W., GUAN X., QIN H., KAUFMAN A. E.: GPU-accelerated volume splatting with elliptical RBFs. In *EuroVis* (2006), Eurographics Association, pp. 13–20. 2

[PGSS07] POPOV S., GÜNTHER J., SEIDEL H.-P., SLUSALLEK P.: Stackless KD-tree traversal for high performance GPU ray tracing. *J. Computer Graphics Forum26*, 3 (2007), 415–424. 2, 4

[Sam89] SAMET H.: Implementing ray tracing with octrees and neighbor finding. *Computers And Graphics 13* (1989), 445–460. 4

[SDG08] STANTCHEV G., DORLAND W., GUMEROV N. A.: Fast parallel particle-to-grid interpolation for plasma PIC simulations on the GPU. *J. Parallel Distrib. Comput 68*, 10 (2008), 1339–1349. 2

[SP09] SCHLEGEL P., PAJAROLA R.: Layered volume splatting. In *Proc. Int. Symp. on Visual Computing (ISVC)* (2009), vol. 5876 of *Lecture Notes in Computer Science*, Springer, pp. 1–12. 2

[vdLGS09] VAN DER LAAN W. J., GREEN S., SAINZ M.: Screen space fluid rendering with curvature flow. In *SI3D* (2009), ACM, pp. 91–98. 2

[Wes90] WESTOVER L.: Footprint evaluation for volume rendering. In *Computer Graphics (SIGGRAPH '90 Proceedings)* (Aug. 1990), Baskett F., (Ed.), vol. 24, pp. 367–376. 2

[WG92] WILHELMS J., GELDER A. V.: Octrees for faster isosurface generation. *ACM Trans. Graph. (Proc. SIGGRAPH)11*, 3 (1992), 201–227. 8

[ZGHG10] ZHOU K., GONG M., HUANG X., GUO B.: Data-parallel octrees for surface reconstruction. *IEEE Trans. on Visualization and Computer Graphics* (2010). 2, 3, 4, 8

[ZHWG08] ZHOU K., HOU Q., WANG R., GUO B.: Real-time KD-tree construction on graphics hardware. *ACM Transactions on Graphics 27*, 5 (2008), 126ff. 2

[ZP07] ZHANG Y., PAJAROLA R.: Deferred blending: Image composition for single-pass point rendering. *Computers & Graphics 31*, 2 (2007), 175–189. 2

[ZPvBG02] ZWICKER M., PFISTER H., VAN BAAR J., GROSS M.: EWA splatting. *IEEE Trans. on Visualization and Computer Graphics8*, 3 (2002), 223–238. 2, 5

[ZSP08] ZHANG Y., SOLENTHALER B., PAJAROLA R.: Adaptive sampling and rendering of fluids on the gpu. In *Proceedings Symposium on Point-Based Graphics* (2008), pp. 137–146. 2

Appendix A: Parallel Shuffled Keys

During raycasting we address a node array in order to obtain the particle subset located in the neighborhood of a sampling position. However, it is important to mention that each node is identified by its shuffled *xyz* key [WG92]. In particular, shuffled *xyz* keys encode the subregion of an octree node's eight children by using a 3-bit code, ranging from zero to seven. The shuffled *xyz* key of a node at tree depth D is defined as a tuple of 3-bit child-ids

$$x_1y_1z_1\, x_2y_2z_2\, \ldots\, x_Dy_Dz_D\,, \tag{2}$$

which encodes the path from the root node to the respective octree node. With a packet-based traversal mechanism these shuffled keys can be computed in parallel as shown for quadtrees in Fig. 9. Firstly, each thread computes a shuffled *xyz* key for a specific tree depth only and stores the result in shared memory. Secondly, a prefix sum over the shared values is executed in parallel in order to receive a stack of shuffled keys. As a result we have computed the shuffled keys of all possibly required nodes along the tree hierachy. With this location code at an intermediate node we can traverse the child-nodes down to a leaf-node.

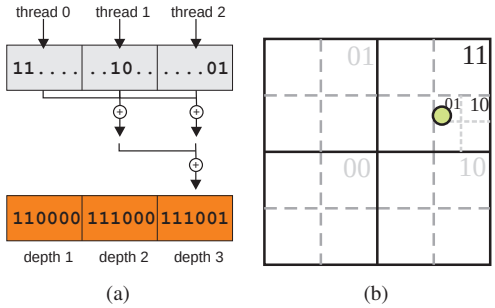

Figure 9: *The parallel computation of the shuffled key stack 9(a) and the corresponding point in the scene 9(b).*

Appendix B: Octree Construction

Listing 1 shows the extensions to the octree of Zhou et al. [ZGHG10]. We refer to their paper for the general octree construction algorithm. Two additional steps are introduced in order to propagate level-of-detail (LOD) particles over the tree. In step 2, particles create references in the influence-cache (infcache) of their neighboring nodes. Together with the particles the influence-cache is sorted according to the shuffled keys (code). In step 7 the particles of each node are merged to coarser particles. These LOD-particles are appended to the particle array.

Listing 1 Construction of Depth D

```
1  // Step 1: compute bounding box ...
2  // Step 2: spread particles
3  code <- new array
4  infcache <- new array
5  for each i=0 to N-1 in parallel
6    for each neighboring node n=0 to 8
7      if particle[i] overlapping n
8        code[i*8 + n] = key << 32 + n
9        infcache[i*8 + n] = i
10 // Step 3: sort all sample points
11 sortCode <- new array
12 Sort( sortCode, code, infcache )
13 Generate the point array according to sortCode
14 // Step 4: find the unqiue nodes ...
15 // Step 5: augment uniqueNode ...
16 // Step 6: create node array ...
17 // Step 7: insert LoD particles
18 for each node n
19   particles[offset[D] + n] =
20       Merge( particles[offset[D-1]] + infcache[n] )
```