Eurographics/ ACM SIGGRAPH Symposium on Computer Animation (2010)
M. Otaduy and Z. Popovic (Editors)

Interactive SPH Simulation and Rendering on the GPU

Prashant Goswami[†], Philipp Schlegel[‡], Barbara Solenthaler[§] and Renato Pajarola[¶]

Visualization and MultiMedia Lab, Department of Informatics, University of Zurich

Abstract

In this paper we introduce a novel parallel and interactive SPH simulation and rendering method on the GPU using CUDA which allows for high quality visualization. The crucial particle neighborhood search is based on Z-indexing *and parallel sorting which eliminates GPU memory overhead due to grid or hierarchical data structures. Furthermore, it overcomes limitations imposed by shading languages allowing it to be very flexible and approaching the practical limits of modern graphics hardware. For visualizing the SPH simulation we introduce a new rendering pipeline. In the first step, all surface particles are efficiently extracted from the SPH particle cloud exploiting the simulation data. Subsequently, a partial and therefore fast distance field volume is rasterized from the surface particles. In the last step, the distance field volume is directly rendered using state-of-the-art GPU raycasting. This rendering pipeline allows for high quality visualization at very high frame rates.*

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—SPH Simulation, Raycasting, Surface Reconstruction

1. Introduction

Physically-based real-time simulations of fluids have a wide range of applications in computer graphics like computer games, medical simulators, and virtual reality applications. Real-time constraints, however, required in the past low fluid resolutions resulting in poor physical and visual results. The particle-based fluid solver *Smoothed Particle Hydrodynamics* (SPH), needs a large number of particles to achieve smooth surfaces and to resolve fine-scale surface details. To accelerate the simulation, enabling real-time simulation with a higher particle resolution, we have implemented the SPH fluid solver and particle rendering on the GPU. Although executing the SPH physics on the GPU accelerates the simulation compared to a CPU implementation [AIY*04, HKK07b, ZSP08], previous solutions come with a number of limitations. The main disadvantage is that the common grid based approach overestimates the memory consumption per grid cell a priori, thus excess use of GPU memory cannot be avoided. Furthermore, these approaches are highly constrained in their choice and usage of problem attributes with respect to functionalities allowed by shader languages.

To cope with these issues, we present a novel CUDA based parallel SPH implementation. The approach relies only on basic CUDA structures like textures and arrays and hence is very flexible and generic and can therefore accommodate any extra attributes. Spatial indexing and search is built upon *Z-indexing* that eliminates use of buckets and allows to determine the neighborhood set of a particle in constant time without wasting space. All other computations, which include sorting particles, are done on the GPU avoiding any CPU-GPU transfer overhead. As a result, the approach produces more efficient results for a similar particle count than state-of-the-art real-time SPH simulation methods. Also our solution can be used for offline SPH simulation of larger particle counts than existing GPU based methods. Alternately, the available free graphics memory can be used for visualization purposes together with simulation.

Once the physical simulation is completed, the resulting particle cloud needs to be visualized. However, visualizing the particles comprises the explicit or implicit reconstruction of the surface of the particle cloud. Many of the pro-

† goswami@ifi.uzh.ch
‡ schlegel@ifi.uzh.ch
§ solenthaler@ifi.uzh.ch
¶ pajarola@acm.org

posed methods for explicit surface reconstruction include expensive calculations and preprocessing and are therefore not suitable for visualization at interactive frame rates. Furthermore, many existing methods cannot be applied to a particle fluid which consists of multiple surfaces, splashes and individual droplets. For that reason, one of the most popular methods to render particles are metaballs [Bli82] which, however, suffers from overly bumpy surfaces.

In this paper we present a new, efficient rendering pipeline that allows for high quality visualization at very high frame rates. The rendering pipeline consists of several steps including the interactive generation of a distance field volume as core step. First, the surface particles have to be extracted from the simulation particle cloud though. The criterion whether a particle is a surface particle is its deviation from the center of mass of the local neighborhood in combination with the overall number of particles in the local neighborhood. The optimal reuse of the simulation data makes this selection very fast. After the surface particles are extracted, a distance field volume is generated by rasterizing the surface point cloud into a volume with scalar values. The values are the minimal square distance to the nearest particle by rasterizing each particle at its position and updating the neighborhood within a predefined radius for the minimal square distance. Thereafter, the normals for lighting are estimated by calculating the gradient based on central differences of the distance field volume. All this is done entirely in parallel on the GPU using CUDA as well. Once the distance field is completed, we render the distance field volume using current state of the art GPU raycasting. The actual color of a pixel depends upon the transfer function. The transfer function is a function of the distance to the nearest particle in the distance field volume. This allows for rendering of soft surfaces and easy visualization of bubbles inside the particle cloud.

The contributions of our visualization are manifold. We introduce a new rendering pipeline consisting of fast surface particle extraction, partial distance field generation and direct volume raycasting. This voxel based approach is not only very fast but opens a whole range of opportunities like transfer function application, easy refractions and Fresnel effects as shown in Figure 7. Constructing the distance field is a fast way to extract a volume from the particle cloud and has, to our knowledge, not been done for SPH visualization so far. The advantage of a distance field is that a voxel is only dependent on the nearest particle. Therefore, it is possible to construct the distance field only where really needed to achieve a better performance.

The main contributions of our paper can be summarized as follows. We present

1. a novel, memory-optimized SPH implementation using *Z-indexing*, and
2. a rendering pipeline based on a minimal distance field volume.

All steps of the simulation and visualization are fully executed on the GPU avoiding any CPU-GPU transfer overhead. We show that with our system we are able to simulate and render 255K particles at interactive frame rates. The performance data of our implementation shows that our system is faster than previous GPU implementations claiming the same simulation quality and stability as ours. In [ZSP08] a higher simulation performance is reached but only at the cost of excessive memory consumption and by introducing simplifications that can lead to stability problems [SP08]. Our approach facilitates a much higher particle resolution, produces better simulation results and incorporates a fast rendering pipeline with an inspiring image quality.

2. Related Work

Simulating fluid motion using SPH was introduced in astrophysics [Mon92] and successfully used in computer graphics to simulate a few thousand particles at interactive frame rates [MCG03]. Performance improvements were obtained by using adaptive particle sizes [APKG07], and by enforcing incompressibility by applying a local prediction-correction scheme to determine the particle pressures [SP09]. While these implementations run completely on the CPU, the GPU was used in addition in [AIY*04] to speed-up the execution. In their work, the neighbor search is computed on the CPU while the standard SPH physics computation is done on the GPU. Similar to our work, [HKK07b, ZSP08] execute all steps of the computation on the GPU. In these methods, a grid-based structure is used to simplify the shader based neighbor search. The drawback of grid structures, however, is that they use too much memory as buckets are allocated with predefined capacity. In a typical simulation, many of these buckets are not used hence the occupied GPU memory cannot be used for any other purpose. Another issue with these approaches is that shader languages make an efficient mapping of the problem difficult and are limited in their usage of problem attributes.

While [ZHWG08] present a method to efficiently construct a real-time KD-Tree on the GPU, it is well known that hierarchical data structures are not the best for SPH like computations [HKK07a]. On the other hand, the sliced data structure proposed by [HKK07a] might not only end up using as much memory as the grid volume itself in the worst case, but it might make direct mapping of SPH onto CUDA more complicated. Though the simulation of simple particle interactions has been done on CUDA [NVI09] using a uniform grid structure with an upper bound of 8 neighboring particles, there is no other work besides [HKK07b, ZSP08] leveraging its huge computation power for SPH simulation.

Visualizing a particle cloud can be done in many different ways but few of them are suitable for rendering at interactive frame rates. A classical method is to extract a polygonal mesh of an iso-surface using marching cubes [LC87]. The particle cloud has to be transformed into a 3D scalar field

that is suitable for marching cubes and marching cubes itself has to be performed fast enough. There are GPU implementations of marching cubes like [DZTS07] to speed it up. The MLS surface introduced in [Lev03] can be used for reconstructing a surface without a polygonal mesh. The projection procedure of MLS is quite expensive and not suitable for interactive applications. Many improvements have been proposed like point set surfaces [AK04] or algebraic point set surfaces [GG07, GGG08], which greatly enhance the performance and enable interactive applications. However, most of them still presume particle normals to be pre-calculated. Even though there are fast methods to calculate particle normals like the color field approach presented in [Mor00], the resulting normals are of low quality in regions with a weak particle density. Nevertheless, the biggest issue with point set surfaces surfaces is that in many cases the particle cloud does not form a smooth, manifold surface or no surface at all and not all particles may lie on the surface. According to our experiments, this causes bad results in these critical areas. The point set surface operator often needs many time consuming iterations or does not converge at all and even if it converges, the reconstructed surface doesn't form a plausible liquid surface because of the wrong input points.

Other method for visualizing a particle cloud include metaballs [Bli82]. The concept of metaballs is quite closely related to the concept of SPH and therefore well suitable. In [ZSP08] an approach is presented for rendering a SPH simulation using metaballs on the GPU. Kanamori et al. [KSN08] showed how a large number of metaballs can be rendered using raycasting. Unfortunately, metaballs cannot hide their structure even if several 10k particles are used for the simulation. Points as a rendering primitives have been successfully used to render high quality surfaces and could also be employed for rendering particle clouds, see [GP07]. In [IDYN08] point primitives are used for rendering of water surfaces. Unfortunately, point-based methods also rely on accurate particle normals and effects like the surface of a transparent liquid with (multiple) refractions are difficult to implement. In [YHK09] a voxel based method has been presented recently. They construct a density field in two stages. First they construct a rough approximation which they refine later on. The finished density field is then rendered using standard GPU raycasting. With their approach they can render liquids with multiple refractions at interactive frame rates.

3. Physical Simulation

The computationally most expensive part in SPH simulation is the neighborhood search, performed on each particle for every time step. Fixed grids have commonly been used to allocate particles to buckets for fast spatial range queries. However, the number of buckets grows too quickly with the count of particles and extent of the simulation domain to maintain this concept on the GPU with its limited memory.

We present a method which performs the entire SPH computation on the GPU using CUDA, not only optimizing the neighborhood search, both in terms of space and time overhead, but also accelerating the expensive SPH computations. The efficient neighborhood search also supports subsequent surface particle extraction used in the visualization part.

3.1. Neighbor Search

Our approach introduces an efficient *Z-indexing* [Mor66] in the context of range queries which enables obtaining a neighborhood set for a particle without any space overhead. The simulation domain is divided into a virtual indexing grid in X, Y, Z along each of the dimensions, and the grid location of a particle is used to determine its bit-interleaved Z-index, see also Figure 1. The Z-index can be computed very efficiently using a table lookup approach. We can observe that all particles lying within any power-of-two sized aligned block have contiguous Z-indices.

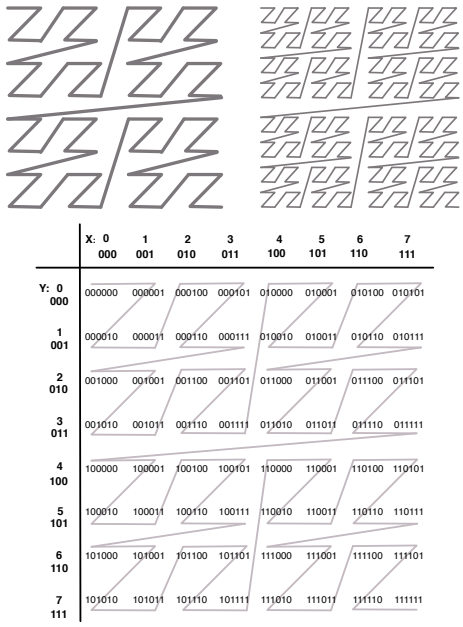

Figure 1: *Z-indices of particles falling within an aligned block of some power of 2 are contiguous, and can be constructed using bit-interleaving from grid locations.*

For range queries given a radius R, the global support radius of the SPH simulation, we determine the nearest power-of-two block size S in the indexing grid domain. The starting Z-index s of any block of size S can easily be determined and particles falling into that block form a sequence between s and $s+S^3$.

At the start of each time step, the Z-indices of all particles are calculated in parallel. Furthermore, the particles are then sorted using parallel radix-sort in CUDA [LG07]. Hence for each block we just need to determine the index

of its first particle and the number of particles it contains. This is accomplished by launching as many CUDA threads as there are number of particles wherein each particle determines its block. Whereas the first particle in a block can be determined using the *atomicMin* operation in CUDA, the number of particles is found by incrementing a particle count with *atomicInc*. Thus each particle updates both the starting index and particle count of its block in the list B, which is of size $|B| = (\frac{X_{max}}{S})^3$ (assuming a simulation domain grid dimension X_{max}).

For the subsequent steps, we have the information of the starting index of every block in B and the number of particles in it. Populated blocks are split if holding more than some N particles and compacted to a set of non-empty CUDA blocks B' in parallel also using *atomicMin* and *atomicInc*, see also Figure 2. Instead of directly launching a pre-decided number of threads, we launch only as many CUDA kernel blocks $|B'|$ as necessary. Each of these blocks has at most N CUDA threads which is the maximum number of particles per block. Each thread is responsible to copy particles iteratively from one of its 27 neighboring blocks in B, and including itself, into the shared memory of its own CUDA block in B' as illustrated in Figure 3. Alternating with this copy process, each CUDA thread computes the physical attributes for one particle in its block in B', see also the algorithm in Figure 4. Therefore, each particle effectively ends up interacting with slightly more others than its actual SPH neighbors. But this way we avoid duplicate copying and hence expensive global memory accesses as well as duplicate particle interactions since:

1. Each particle is copied into a block's shared memory exactly once.
2. Each copied particle in shared memory is used by all other threads or particles in that block once.

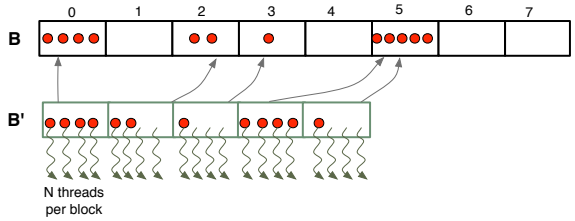

Figure 2: *For each non-empty block in B, a CUDA block is generated in B' and launched with N threads ($N = 4$ here).*

Many blocks may contain fewer than N particles but still run N threads. However, a thread with thread-id tid will not be completely idle as long as the particle count in the current block is equal to greater than tid, or $tid < 27$ in which case the thread has to copy particles from neighboring blocks.

Thus as mentioned above in order to keep track of a block, we just need two attributes and therefore the space required to maintain all blocks is only $2 * B$ for $|B|$ grid blocks in the simulation domain. Note that blocks refer to their particles only by index, using the starting particle and number of particles in a block, and that particles are maintained in a CUDA attributes array. Moreover, as we show in Section 3.3 block computations can be carried out within the memory used for particle attributes and we do not need any extra memory allocated for it.

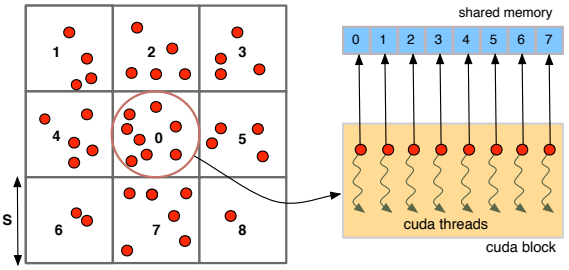

Figure 3: *Each CUDA thread in a block computes attributes for one particle and at the same time copies particles from a neighboring block into its shared memory.*

3.2. Density and Force Computation

The first step in SPH simulation is the density computation of Equation 1 (see also [Mon92]) where m_j refers to the mass of particle of particle at position $\mathbf{r}_j$ and $W(\mathbf{r},h)$ is the smoothing kernel with core radius h.

$$\rho_i = \sum_j m_j W(\mathbf{r}_i - \mathbf{r}_j, h) \qquad (1)$$

Since our simulation domain is divided into blocks of size equal to or greater than the global support radius, the neighbors to any particle in a block for density or force computation, lie no farther than the particles in its immediate neighbor blocks, see also Figure 3. Each of the N threads in a CUDA kernel block copies one particle at a time from a neighboring block to its shared memory and at the same time computes physical attributes for one particle in the current block. In each iteration, every particle in the current block thus updates its density with all the particles copied into shared memory which are at a distance less than or equal to global support radius.

The computed densities from the last step are made available as CUDA textures for force computation. Each particle repeats the same procedure as above for accessing neighbors and their new densities, calculates the force using Equation 2 where ρ_j refers to density of a particle, p_i to pressure and $\nabla W(\mathbf{r},h)$ to the gradient of the smoothing kernel. Thereafter it updates its own position and Z-index. In a CUDA kernel block, each thread also computes attributes like inverse density once and stores them in the shared memory. This way we optimize by reducing expensive operations like division, since similar to density its inverse value is also used by all the threads of block. Finally, each thread writes the computed attributes for its particle to the global memory.

$$\mathbf{f}_i^{pressure} = -\sum_j \frac{m_j}{\rho_j} \frac{(p_i + p_j)}{2} \nabla W(\mathbf{r} - \mathbf{r}_j, h) \qquad (2)$$

Since the updated density values are required for force kernels, density and force computation cannot be clubbed together in a single CUDA procedure and therefore, have to be launched separately as different kernels. This necessitates neighbor finding and copying into the shared memory separately in each procedure. However, in our case neighbor finding is inexpensive and does not hurt the overall performance.

3.3. CUDA Computation

Figure 4 outlines the basic steps of our CUDA SPH algorithm. All the global look-up tables for SPH kernel computations are generated beforehand and kept in constant memory. We avoid bit operations in Z-indexing (Figure 1) by computing interleaved bitwise representations of all possible grid values along any x, y and z dimension and storing them as CUDA texture. The Z-index value for a given position $(\mathbf{r}_x, \mathbf{r}_y, \mathbf{r}_z)$ on the grid can be obtained by bitwise OR of these texture look-ups.

For the physics part, our implementation requires four ping-pong CUDA arrays: first for radix-sort, second for position and Z-index, third for velocity and pressure, fourth for density. Since it could be the case that global memory accesses are not coalesced, we obtain the old attribute values from CUDA texture arrays (i.e. particle positions, densities) while we write updated values into the ping-pong arrays. These two sets of CUDA arrays reverse their role every frame as readable textures and writable global memory. Since force values are not required outside the force kernel, we do not need any global memory for them. Also, we avoid allocating separate memory for block computations by doing it in the same CUDA array as we use for radix-sort. Once updated particle positions are copied to the position array, the radix-sort array is free and can be used for block computations. Hence, no extra space is in fact needed for blocks maintenance.

4. Rendering

For visualizing the simulated particle data we introduce a new, voxel-based rendering pipeline. Voxel based approaches have the advantage that no explicit surface reconstruction is required and that estimating surface normals is quite easy. Furthermore, there is no limitation to a single, closed surface, e.g. bubbles in the liquid can easily be rendered. The rendering pipeline consists of three parts: surface particle extraction, distance field volume generation and GPU raycasting. Our approach is very flexible in terms of trading performance for quality and vice versa. If a higher rendering quality is desired, a larger distance field can be

```
ComputePhysics()
{
  Copy all particles from CPU to GPU memory
  foreach frame {
    /*—— Z-index and Sorting ——*/
    Calculate Z-indices for particles
    Sort them using radix-sort
    Copy sorted particles in the ping pong array
    Make CUDA texture of sorted particle positions

    /*—— Block Generation ——*/
    Create blocks from sorted particles by determining
    foreach block (using positions from texture):
      – Starting index or index of first particle in array
      – Number of particles in it
    B0 : Number of blocks identified
    Split all blocks containing more than N particles each
    B' : Number of compacted blocks after splitting

    /*—— Density Computation ——*/
    Launch B' CUDA kernels with N threads each
    foreach CUDA block
      Determine M: max particles in neighbors
      N: particles in current block
      Copy its own N particles into its shared memory
      for i = 0 to M
        – Copy a particle from neighboring blocks 0 to 26
        (one per thread) to its own shared memory
        – syncthreads()
        – for j = 0 to N
            Compute new densities from new copied
            neighbors in shared memory
      for j = 0 to N
        – Write updated densities to global memory
    Make CUDA texture of newly computed densities

    /*—— Force Computation ——*/
    Launch B' CUDA kernels with N threads each
    foreach CUDA block
      Determine M: max particles in neighbors
      N: particles in current block
      Copy its own N particles into its shared memory
      for i = 0 to M
        – Copy a particle from neighboring blocks 0 to 26
        (one per thread) to its own shared memory
        – syncthreads()
        – for j = 0 to N
            Compute new forces using texture densities
            and neighbors copied in shared memory
      for j = 0 to N
        – Handle collisions and boundary forces
        – Update particle positions
        – Write updated positions to global memory
  }
}
```

Figure 4: *Algorithm for CUDA SPH Physics computation*

used and the simple raycaster can be enhanced by features known from raytracers like support for multiple refractions and reflections. In fact, our raycaster does support multiple refractions, Fresnel effects and environment mapping and still achieves very good frame rates (see also Figure 7).

The core part of our rendering pipeline is the construction of the distance field volume. Our distance field approach allows us to construct the distance field volume only partially making it cheap compared to other methods like density fields. Since the cost is directly related to the number of particles, we reduce them by extracting only the surface particles once the simulation step is done. Only these remaining surface particles are used to generate the distance field. The distance field volume is then directly rendered using raycasting (Figure 6). Samples along the ray are classified by applying a transfer function, illuminated and composited together resulting in the final color value.

4.1. Surface Particle Extraction

After the physical simulation step, the surface particles are extracted. The particles that do not lie on the surface are not required for the visualization and can thus be omitted. It makes the visualization much faster because the generation cost of the distance field is closely related to the number of particles. The extraction of the surface particles follows the method presented in [ZSP08]. A particle i is considered to be a surface particle if its distance to the center of mass $\mathbf{r}_{CM_i}$ of its neighborhood is larger than a certain threshold. The center of mass can be determined by summing up the positions $\mathbf{r}_j$ in i's neighborhood weighted by their mass m_j.

$$\mathbf{r}_{CM_i} = \frac{\sum_j m_j \mathbf{r}_j}{\sum_j m_j} \tag{3}$$

However, Equation 3 fails to detect surface particles in areas where only a few particles are present and that is why an additional constraint is added. If for a particle i the number of neighbors is below a user defined threshold, i is always considered to be a surface particle. Because the neighborhood has already to be calculated in the physics simulation, this method for extracting surface particles is quite efficient. It employs the same pattern as for the density and force computation. However, the result depends heavily on the defined thresholds and is not always perfect. This is not a problem for our rendering method, but it could be an issue for other methods which rely on the invariant that particles correctly belong to the surface. Our surface particle extraction is implemented entirely on the GPU using CUDA. For each particle and its neighbors, the number of particles in the neighborhood and the center of mass are determined respectively. If the particle belongs to the surface it is written to an output array and omitted otherwise.

4.2. Distance Field Generation

A distance field is a scalar field where the values represent the distance to the nearest surface, with a distinction between signed and unsigned distance fields. Because the relation if a particular spatial position lies inside or outside the fluid cannot easily be derived, we use an unsigned distance field. We note that constructing an entire distance field is prohibitively expensive. However, since for a distance field a voxel value is in fact only dependent on the nearest particles, we can afford to only rasterize the distance field in the areas required for rendering, i.e. the fluid surface.

Our distance field consists of a uniformly spaced grid. The grid size has a major impact not only on performance but also on image quality. A larger grid results in a better image quality but requires much more time to construct. In fact, the number of operations raises cubically with the grid dimension. But the grid size is not the only factor regarding the time needed for constructing the distance field. Also the number of particles heavily influences the required time. For practical reasons we are typically using a 64^3 grid, as it has shown to be the best compromise in most cases.

Several methods have been proposed for fast distance fields construction on the GPU such as [SOM04, FL08]. Though, most of the proposed methods assume construction of a distance field from a triangle mesh. In contrast, we construct a distance field from the surface particle cloud and further, we do not require a full-fledged distance field, which allows for optimizations. The construction of the distance field depends on the transfer function used during raycasting in the final rendering step. Not all distance values are of interest since values for which the transfer function maps to zero opacity are not significant. Moreover, the actual transfer function mapping of the distance values itself is not important as long as it is consistent. Consequently only those distance field areas have to be rasterized where they map to relevant transfer function values. For a typical Gaussian, or otherwise narrow-band transfer function for displaying a surface, the relevant transfer function values may only occur in a small band r_{min} to r_{max} around the surface particles as indicated in Figure 5.

In our approach the distance field volume is initialized with a maximum value. An imaginary box with radius r around each extracted surface particle is then rasterized into the distance field. The radius r of this box depends on the particle density and transfer function, and has to prevent any preliminary clipping. Within a particle's box, for each voxel the distance d to the particle is calculated. If d lies in the interval $[r_{min}, r_{max}]$ the voxel value d_v is updated according the following equation:

$$d_v = \min\left(d, d_v^{old}\right) \tag{4}$$

If $d > r_{max}$ nothing is done and if $d < r_{min}$ then d_v is set to the minimal distance. $r_{max} = r\sqrt{3}$, whereas r_{min} corresponds to the maximal value for which the transfer function maps to

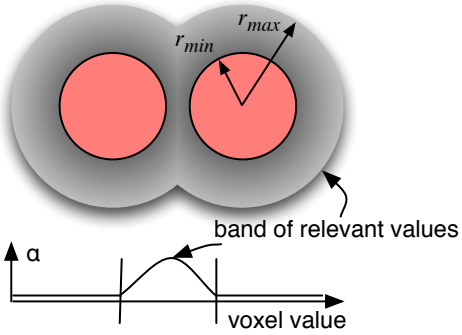

Figure 5: *The grey gradient represents the distance to the two particles. For a given transfer function, only values in the transition band have effectively to be rasterized for the distance field.*

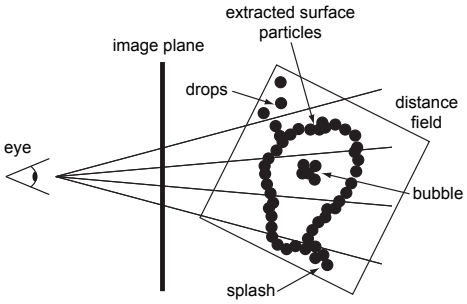

Figure 6: *Rays travelling through the rasterized distance field have to be intersected with surface particles only.*

the same as every value $< r_{min}$ minus the radius used for gradient estimation. Otherwise the gradients may be impaired depending on the particular application.

The distance field generation is implemented entirely in CUDA on the GPU. It is very challenging to optimize this process for memory access patterns in CUDA. Hence we rasterize only the minimal set of surface particles required for rendering as described above keeping overall memory access low. Rasterization takes place into a mapped pixel buffer object, which can later be bound as an OpenGL 3D texture. In principal, all data can stay in GPU memory and no copying has to be performed. Unfortunately on few hardware platforms this does not seem to be the case and we suspect unoptimized or faulty graphics drivers.

4.3. GPU Raycasting

Eventually the distance field is rendered using GPU raycasting. GPU raycasting [SSKE05] has become the preferred way for direct volume rendering and can also be employed for iso-surface rendering [HSS*05]. Raycasting is based on the light emission and absorption model [Max95] and the volume rendering integral [Mor04] respectively. For each pixel of the image plane a ray from the viewpoint is shot through the volume and samples are taken along the ray, see also Figure 6.

Each sample on a ray is mapped from the scalar voxel field value to an (r,g,b,a) tuple according to the transfer function. The necessary normal for lighting calculations can be approximated by the volume gradient using central differences where ρ denotes the voxel value:

$$\nabla \rho_{i,j,k} = \begin{pmatrix} \frac{\partial \rho}{\partial x} \\ \frac{\partial \rho}{\partial y} \\ \frac{\partial \rho}{\partial z} \end{pmatrix} \approx \begin{pmatrix} \rho_{i+1,j,k} - \rho_{i-1,j,k} \\ \rho_{i,j+1,k} - \rho_{i,j-1,k} \\ \rho_{i,j,k+1} - \rho_{i,j,k-1} \end{pmatrix} \qquad (5)$$

The final color for each pixel is derived by compositing the samples together along the ray.

Raycasting on the GPU has become very efficient with the advent of GPU support for real loops. For each pixel a shader is executed which basically consists of a loop where samples along the ray are taken, mapped, shaded and composited together. The rays are commonly initialized by rasterizing the front and the back of a bounding geometry into two textures respectively. Its color is defined by the interpolated 3D coordinates of the bounding geometry. Thus the direction vectors of rays can directly be extracted by taking the difference between the front and back colors for each pixel. The simplest bounding geometry is a cube enclosing the distance field volume. Such a bounding box is easy and fast to rasterize but contains the inherent disadvantage of potentially covering large empty regions. This can be avoided by rasterizing a more complex bounding geometry. For each extracted surface particle a small bounding box can be rasterized resulting in a relatively tight overall bounding geometry. We implement this approach efficiently by binding the buffer containing the extracted surface particles as a vertex buffer object and render the particles as simple points, and a geometry shader generates the small bounding boxes on-the-fly. The rasterization overhead is typically outweighed by the gainings in raycasting.

Sampling along the rays using 3D texture lookups is the core part of GPU raycasting. Since we use an unsigned distance field no distinction between inside/outside is needed. The applied sampling frequency has a direct impact on the performance as well as the image quality. 250 samples per unit distance turned out to be sufficient for most transfer functions. Sampling can be terminated once a pixel becomes opaque. We implemented GPU raycasting using regular GLSL shaders and a Phong like lighting model. We also implemented multiple refractions and Fresnel effects based on the gradient. Additionally we use environment mapping and partly mix the environment to the color of the samples defined by the transfer function.

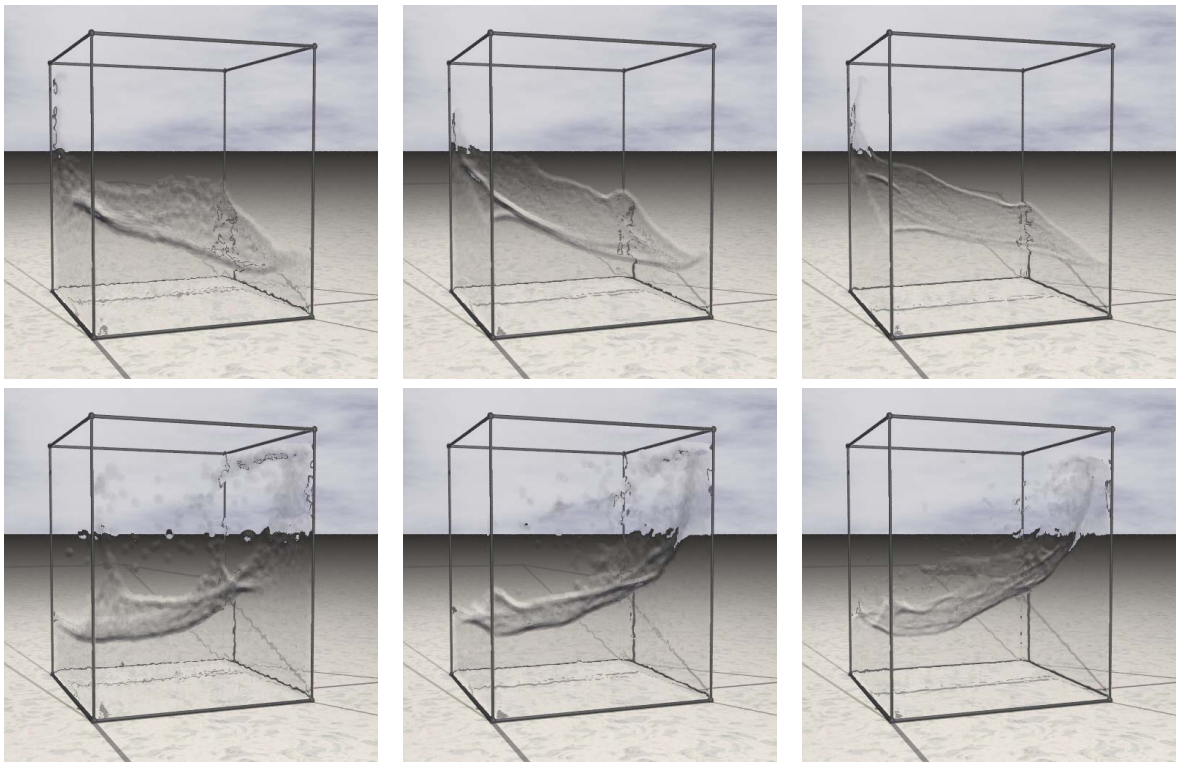

Figure 7: *Rendering of a collapsing water cube with 16'128 particles (left), 75'200 particles (middle) and 129'024 particles (right) respectively. The rendering rates are 126fps (left), 88fps (middle) and 70fps (right). The overall frame rates including physical simulation, surface particle extraction and rendering are 52fps (left), 15fps (middle) and 11fps (right).*

5. Results

We have implemented and tested both SPH simulation and rendering using OpenGL, GLSL and CUDA 2.3 on two different platforms:

1. MAC OS X 10.5.8, 2 X 2.66 GHz Dual-Core Intel Xeon and NVIDIA GeForce 8800 GT with 512 MB VRAM
2. Linux, 2.93 GHz Core i7 and NVIDIA GeForce GTX 280 with 1 GB VRAM

The virtual grid size for Z-indexing is kept at the same fine resolution of 1024^3 throughout the experiments. Since we need only 10 bits to represent a particle's grid position along any direction, all bits of a Z-index can be packed into a single 32-bit integer. Throughout our experiments, all CUDA blocks for SPH computation have a block dimension of 64 which we have verified to be a good size experimentally. Also since the block size S is dependent on global support radius R, it is important to choose parameters such that the difference between the projection of global support radius on grid and its closest power of 2 is minimal.

For testing the SPH simulation and rendering we have used two different setups. The first scene consists of a water block collapsing due to gravity as in Figure 7. The particle count of the water block can be chosen initially and stays constant during the simulation. This does not apply for the number of extracted surface particles, which depends on the actual surface and changes every frame. The second scene consists of a water-jet filling a basin as in Figures 8 and 9. There the number of particles changes during the simulation but is limited by an upper bound. Scene 1 is more suitable for performance measurements. We have measured the performance for different particle counts as well as for the different stages of simulation and rendering as can be seen in Tables 1 and 2. For rendering, we extended our raycaster to include multiple refractions, Fresnel effects and environment mapping as shown in Figures 7 and 8.

With our simulation and rendering solutions we are able to achieve high frame rates as well as great image quality at the same time. Effectively we can reach interactive frame rates with a quarter million of particles on Platform 2 while enjoying an excellent image quality.

As can be seen from Tables 1 and 2, our CUDA based SPH implementation achieves excellent simulation and rendering rates. Table 1 lists the statistics for Platform 1, which notably on a slower GPU in comparison to [HKK07b] still demonstrates improved simulation performance. While for the simulation part [HKK07b] report 17fps for 60K and 1fps for 1M particles on an NVIDIA 8800 GTX, we achieve 16fps

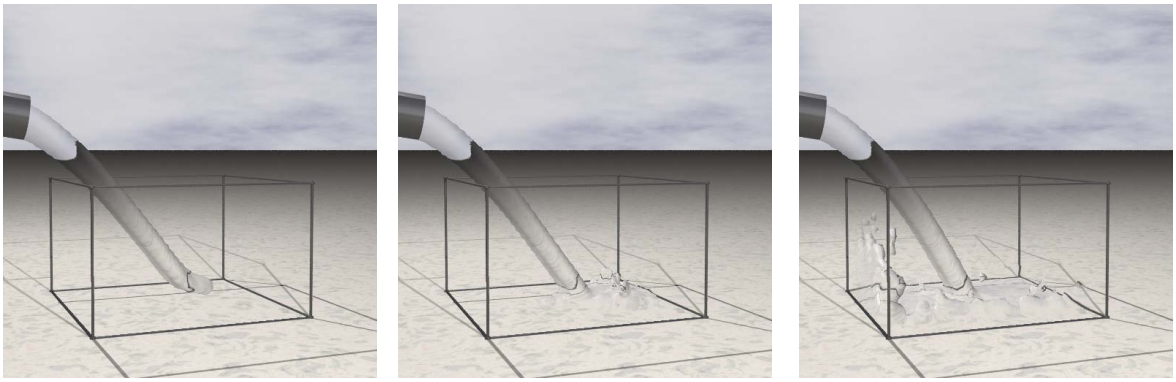

Figure 8: *Simulation of a water-jet with variable particle count up to 102'000. The average rendering rate is 75fps and 18fps overall, including the physical simulation, surface particle extraction and rendering.*

for 75K and 1.25fps for 1M particles on the much slower NVIDIA 8800 GT.

On the faster 8800 GTX, [ZSP08] report faster times than [HKK07b], although not only at the expense of extensive memory consumption but also at a loss of quality and accuracy since the density computation is combined with that of the pressure force, which is then updated with a delay of one frame. Though this combination about doubles the performance, the simulation can become unstable [SP08]. If we compare the simulation together with visualization we are distinctly faster than [ZSP08]: they achieve 8fps for 53k particles (8800 GTX) where we have 8fps for 75k particles (8800 GT) on a GPU with 30% less memory bandwidth and 12.5% less CUDA cores but nevertheless at a much higher image quality. In addition, considering the combined density and pressure force computation as in [ZSP08], we estimate similar physics simulation speed as [ZSP08], but even on our slower GT hardware.

Along with the results in Table 2 we can show that our approach is applicable to a few hundred thousand particles on current graphics hardware, matching or outperforming other state-of-the-art methods at a reduced GPU memory overhead.

Particle Count	Physical Simulation	Surface Particle Extraction	Rendering	Overall
16'128	69fps	213fps	71fps	28fps
75'200	**16fps**	42fps	37fps	**8fps**
129'024	9fps	26fps	27fps	5fps
255'600	5fps	15fps	26fps	3fps

Table 1: *Simulation and rendering performance results for a collapsing water column on Platform 1.*

In comparison to the fastest recent CPU based SPH simulation [SP09], our offline simulation for 250k particles is about 5 times faster at 5.6 physics time steps per second on an NVIDIA 8800 GT, against 1.176 on a 2.66 GHz, 4 cores processor.

Particle Count	Physical Simulation	Surface Particle Extraction	Rendering	Overall
16'128	123fps	413fps	126fps	52fps
75'200	26fps	74fps	88fps	15fps
129'024	17fps	51fps	70fps	11fps
255'600	10fps	28fps	49fps	6fps

Table 2: *Simulation and rendering performance results for a collapsing water column on Platform 2.*

6. Conclusions

In this paper we have presented a novel GPU accelerated parallel SPH simulation and rendering pipeline that achieves interactive simulation and rendering rates for particle counts of up to a quarter million on current consumer graphics hardware. We have introduced a new particle neighbor search approach in CUDA based on efficient Z-indexing which is not only computationally but also memory efficient. The spatial indexing and search method is flexible and generic, and as well can be used for fast surface particle extraction. Additionally, we have introduced a new rendering pipeline on the GPU based on the online construction of a distance field volume from the extracted surface particles, which subsequently is rendered using state-of-the-art raycasting.

The main advantages of our parallel SPH simulation are its low memory consumption compared to other grid based approaches and that it outperforms prior state-of-the-art solutions in terms of performance. Our SPH fluid rendering introduces a new fast distance field generation method for particles combined with efficient particle raycasting that supports a wide range of volume rendering and raytracing options such as flexible transfer functions, illumination and shading effects.

References

[AIY*04] AMADA T., IMURA M., YASUMORO Y., MANABE Y., CHIHARA K.: Particle-based fluid simulation on GPU. In *ACM Workshop on General-Purpose Computing on Graphics Processors* (2004).

Figure 9: *Offline rendered images using POVRAY of a simulation with 250'000 particles using our CUDA based SPH solution. On average 5.6 physics time steps per second are reached on Platform 1.*

[AK04] AMENTA N., KIL Y. J.: Defining point-set surfaces. *ACM Transactions on Graphics 23*, 3 (2004), 264–270.

[APKG07] ADAMS B., PAULY M., KEISER R., GUIBAS L. J.: Adaptively sampled particle fluids. *ACM Transactions on Graphics 26*, 3 (July 2007), 48–54.

[Bli82] BLINN J. F.: A generalization of algebraic surface drawing. In *ACM Transactions on Graphics* (1982), pp. 235–256.

[DZTS07] DYKEN C., ZIEGLER G., THEOBALT C., SEIDEL H.-P.: *GPU marching cubes on shader model 3.0 and 4.0*. Research Report MPI-I-2007-4-006, Max-Planck-Institut für Informatik, August 2007.

[FL08] FAGERJORD K. R., LOCHEHINA T.: *GPGPU: Fast and easy 3D Distance Field computation on GPU*. Research report, Narvik University College, 2008.

[GG07] GUENNEBAUD G., GROSS M.: Algebraic point set surfaces. *ACM Transactions on Graphics 26*, 3 (2007), 23.

[GGG08] GUENNEBAUD G., GERMANN M., GROSS M.: Dynamic sampling and rendering of algebraic point set surfaces. *Computer Graphics Forum 27*, 2 (April 2008), 653–662.

[GP07] GROSS M. H., PFISTER H. (Eds.): *Point-Based Graphics*. Series in Computer Graphics. Morgan Kaufmann Publishers, 2007.

[HKK07a] HARADA T., KOSHIZUKA S., KAWAGUCHI Y.: Sliced data structure for particle-based simulations on GPUs. In *Proceedings 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia* (2007), pp. 55–62.

[HKK07b] HARADA T., KOSHIZUKA S., KAWAGUCHI Y.: Smoothed particle hydrodynamics on GPUs. In *Proceedings Computer Graphics International* (2007), pp. 63–70.

[HSS*05] HADWIGER M., SIGG C., SCHARSACH H., BÜHLER K., GROSS M. H.: Real-time ray-casting and advanced shading of discrete isosurfaces. *Computer Graphics Forum 24*, 3 (2005), 303–312.

[IDYN08] IWASAKI K., DOBASHI Y., YOSHIMOTO F., NISHITA T.: GPU-based rendering of point-sampled water surfaces. *The Visual Computer 24*, 2 (2008), 77–84.

[KSN08] KANAMORI Y., SZEGO Z., NISHITA T.: GPU-based fast ray casting for a large number of metaballs. *Computer Graphics Forum 27*, 3 (2008), 351–360.

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3D surface construction algorithm. In *Proceedings ACM SIGGRAPH* (1987), pp. 163–169.

[Lev03] LEVIN D.: Mesh-independent surface interpolation. In *Geometric Modeling for Scientific Visualization* (2003), pp. 37–49.

[LG07] LE GRAND S.: *Broad Phase Collision Detection with CUDA. GPU Gems*. Addison-Wesley, 2007.

[Max95] MAX N.: Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics 1*, 2 (June 1995), 99–108.

[MCG03] MÜLLER M., CHARYPAR D., GROSS M.: Particle-based fuid simulation for interactive applications. In *Proceedings Eurographics/ACM Symposium on Computer Animation* (2003), pp. 154–159.

[Mon92] MONAGHAN J.: Smoothed particle hydrodynamics. *Annual Review of Astronomy and Astrophysics 30* (1992), 543–574.

[Mor66] MORTON G.: A computer oriented geodetic data base and a new technique in file sequencing. IBM, Ottawa, Canada, 1966.

[Mor00] MORRIS J. P.: Simulating surface tension with smoothed particle hydrodynamics. *International Journal of Numerical Methods in Fluids 33* (2000), 333–353.

[Mor04] MORELAND K. D.: *Fast High Accuracy Volume Rendering*. PhD thesis, The University of New Mexico, 2004.

[NVI09] NVIDIA: CUDA SDK Samples. *http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html* (2009).

[SOM04] SUD A., OTADUY M. A., MANOCHA D.: DiFi: Fast 3D distance field computation using graphics hardware. *Computer Graphics Forum 23*, 3 (2004), 557–566.

[SP08] SOLENTHALER B., PAJAROLA R.: Density contrast SPH interfaces. In *Proceedings ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2008), pp. 211–218.

[SP09] SOLENTHALER B., PAJAROLA R.: Predictive-corrective incompressible SPH. *ACM Transactions on Graphics 28*, 3 (2009), 40:1–6.

[SSKE05] STEGMAIER S., STRENGERT M., KLEIN T., ERTL T.: A simple and flexible volume rendering framework for graphics-hardware–based raycasting. In *Proceedings International Workshop on Volume Graphics* (2005), pp. 187–195.

[YHK09] YASUDA R., HARADA T., KAWAGUCHI Y.: Fast rendering of particle-based fluid by utilizing simulation data. In *Proceedings EUROGRAPHICS Short Papers* (2009), pp. 61–64.

[ZHWG08] ZHOU K., HOU Q., WANG R., GUO B.: *Real-Time KD-Tree Construction on Graphics Hardware*. Technical report, Microsoft Research, 2008.

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

Figure 8: *In this production example from the movie Avatar, branches are again simulated as tetrahedral meshes and glued together with point cloud glue. At appropriate frames, the glue is keyed to release to produce the desired result. Simulation by Seunghun Lee. Image TM and ©2009 Twentieth Century Fox Film Corp. All rights reserved.*

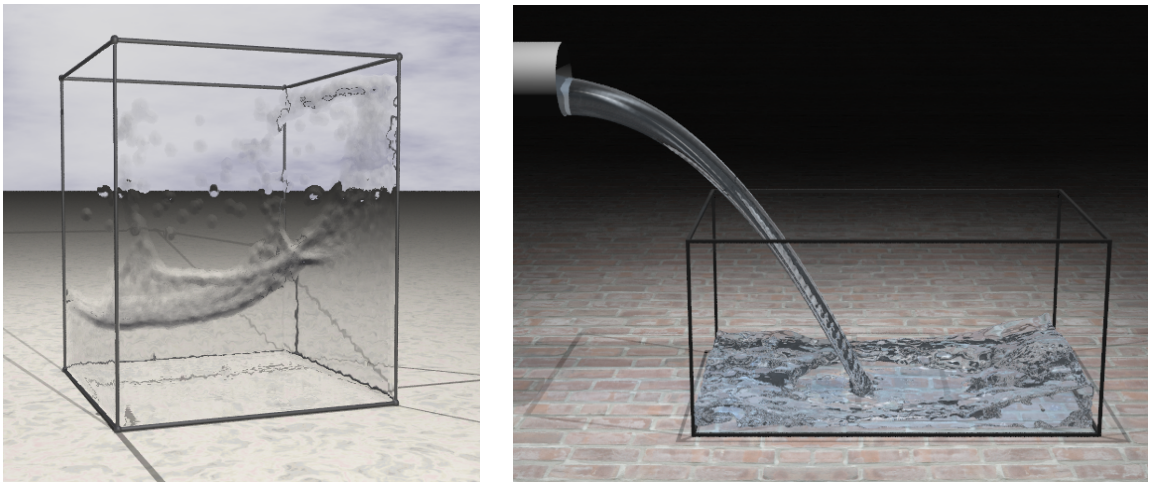

Figure 10: *Poster Snapshots*