High Performance Graphics (2014)
Jonathan Ragan-Kelley and Ingo Wald (Editors)

Interactive Rendering of Giga-Particle Fluid Simulations

F. Reichl[1 †] and M. G. Chajdas[1 ‡] and J. Schneider[2 §] and R. Westermann[1 ¶]

[1]Technische Universität München
[2]King Abdullah University of Science and Technology

Abstract

We describe the design of an interactive rendering system for particle-based fluid simulations comprising hundreds of millions of particles per time step. We present a novel binary voxel representation for particle positions in combination with random jitter to drastically reduce memory and bandwidth requirements. To avoid a time-consuming preprocess and restrict the workload to what is seen, the construction of this representation is embedded into front-to-back GPU ray-casting. For high speed rendering, we ray-cast spheres and extend on total-variation-based image de-noising models to smooth the fluid surface according to data specific boundary conditions. The regular voxel structure permits highly efficient ray-sphere intersection testing as well as classification of foam particles at runtime on the GPU. Foam particles are rendered volumetrically by reconstructing densities from the binary representation on-the-fly. The particular design of our system allows scrubbing through high-resolution animated fluids at interactive rates.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.7]: Three-Dimensional Graphics and Realism—Rendering, Animation

1. Introduction

Particle-based methods are increasingly used for simulating high-resolution violent fluid effects with possible fragmentation, splashing, and large deformations. In particular the Fluid-Implicit-Particle (FLIP) method can easily use hundreds of millions of particles and has led to a tremendous increase in the number of particles used in visual effects simulations.

A problem that has not gained much attention so far is how to render efficiently the gigantic particle sets resulting from large-scale particle simulations. In particular previewing tools for giga-particle-scale simulations are not existing, yet they are important to analyze and control the visual effects created by such simulations. For instance, to detect failures early, to investigate specific fluid details, or to look at a scene from different viewpoints to find the best angle. Previewing tools do not need to show the fluid effects in final-film-quality, but they should show a plausible rendering that reveals the fluid surface and splashes, and support interactive scrubbing through the animation to analyze the dynamics over time.

Some visual effects companies have started to engage in the issue of rendering very large particle simulations. See, for instance, the 2013 release of Houdini by Side Effects. However, the classical approach, which first computes a 3D scalar field by summing up the particle smoothing kernels and then reconstructs a polygonal isosurface from this field, is not suited for previewing. For instance, to capture all fluid details in the FLIP simulation shown in Fig. 1, a regular grid of size $4K^3$ is required to discretize the scalar field. This results in several gigabytes of data per time step and many minutes of preprocessing until the fluid surface is available.

Current real-time rendering approaches for particle-based fluid simulations can handle a few million particles [MSD07, vdLGS09, GSSP10], but they do not scale well with the number of particles and time steps. This is due to bandwidth and memory constraints which limit the amount of particles that can be streamed and processed at reasonable rates. Another constraint is often caused by the use of a rasterization-based rendering pipeline. Especially when many isolated splashes occur and occlusion culling cannot be performed

† e-mail: reichlf@in.tum.de
‡ e-mail: chajdas@tum.de
§ e-mail:jens.schneider@KAUST.EDU.SA
¶ e-mail:westermann@tum.de

DOI: 10.2312/hpg.20141099

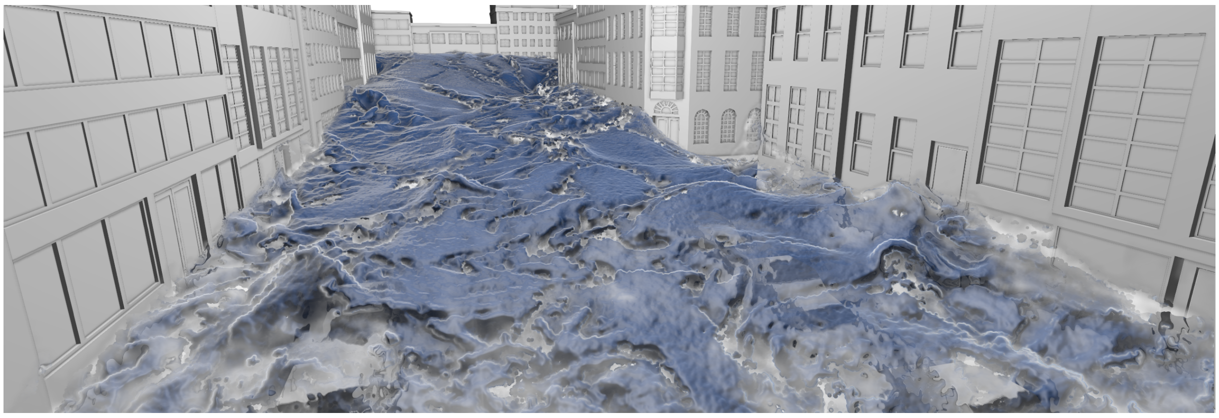


Figure 1: Rendering a 500 million particle simulation, preprocessed in 12 seconds, at 72 ms per frame on a 1280×720 viewport using ray-sphere intersections, image-based surface smoothing, volumetric foam and approximative refractions on a single PC.

effectively, the required rasterization capabilities go beyond what is available today. Real-time approaches which can classify and render volumetrically splashes out of the box at runtime have not yet been proposed.

Our Contribution. We present an interactive rendering system for particle-based fluid simulations reaching the giga-particle scale. Our system design builds upon a novel binary voxel representation for particle positions. It relaxes bandwidth limitations throughout the entire previewing pipeline and enables parallel GPU processing for high performance rendering. It further reduces the memory requirements significantly and shows no qualitative difference to the initial data representation. The regular voxel structure permits highly efficient data access operations, which we exploit to classify foam particles and perform ray-surface intersection testing and volumetric compositing with early ray-termination.

All data processing tasks are embedded into front-to-back ray-casting. Based on spatial particle binning, which is the only required preprocess, all other processes are carried out on request during rendering. In this way we can restrict the workload, including memory transfer, to the fluid parts which are seen.

Our rendering system displays the iso-surface by rendering each particle as a sphere and post-processing the resulting surface. Therefore, we have adapted and extended the total-variation-based image de-noising model of Rudin, Osher, and Fatemi [ROF92] to work on oddly-shaped image domains and allow depth-dependent smoothing strengths. We demonstrate superior quality of this model over other available models, at similar speeds.

For the 500 million particle simulation shown in Fig. 1, our system shows the following performance features when rendering onto a 1280×720 viewport:

- A very low preprocessing time of less than 12 seconds.
- A time to first frame of less than 1 second.
- Rendering at 50 milliseconds per frame.
- Interactive scrubbing through the animated fluid.

Note, in particular, that the very low processing time distinguishes our approach from existing point based rendering approaches, for instance [KSW05, GEM*13]. Such approaches typically build upon sophisticated compression/acceleration structures and require a pre-process of the order of many minutes for the data sets we address.

Our paper is structured as follows: First, we review previous work that is related to ours. We then describe the architecture of the proposed previewing system. Here we introduce the system's functionality and how this is achieved. In the next section we motivate our design decisions and discuss the trade-offs involved in making our system practical for rendering extremely large particle simulations. Finally, we analyze the processing and rendering performance of our system.

2. Related Work

Due to page restrictions, we do not attempt here to survey the vast body of literature related to particle-based fluid simulations, for which our rendering system is designed. However, Müller et al. [MCG03], Ihmsen et al. [IOS*14], and Zhu and Bridson [ZB05, Bri08] discuss the basic principles and algorithms used in such simulations.

Rendering a fluid's surface from particle-based simulation data is a well-studied problem. Most commonly it is performed by using Marching-Cubes-style algorithms [LC87] to reconstruct a polygonal isosurface in the density field that is derived from the discrete particle set, or by rendering the surface directly via volume ray-casting in this field. Usually, the density is first resampled to a uniform grid, so that the surface location can be determined efficiently and cell-wise interpolation schemes can be used. The effect of the resampling scheme on the quality of the reconstructed surfaces has been studied intensively [MCG03, ZB05, SSP07, APKG07]. In the most recent approach by Yu and Turk [YT13] the resampling kernels are aligned locally with the distribution of nearby particles via principal component analysis. Especially when using GPU resampling techniques [ZSP08, FAW10], the fluid surface can be rendered at high speed using GPU-based ray-casting. Resampling can be further accelerated by considering only those grid vertices close to the surface boundary [YHK09, AIAT12].

To avoid the memory consumption of a uniform grid, isosurfaces can be rendered directly from the particle set. Kanamori et al. [KSN08] evaluate ray-particle intersections on the GPU and use depth peeling to determine their front-to-back order. Zhang et al. [ZSP08] and Goswami et al. [GSSP10] use GPU particle binning and neighbor search to determine ray-particle overlaps efficiently. Gribble et al. [GIK*07] perform direct ray-sphere intersections on the CPU using a grid-based acceleration structure. An isosurface extraction technique that works directly on the particle set was proposed by Rosenberg and Birdwell [RB08].

While adaptive data structures can exploit the advantages of uniform grids, they restrict the workload to a narrow band around the fluid surface. The method of Bridson [Bri03] uses sparse grids, which store adaptively refined bricks in this band. Dynamic tubular grids and RLE encoded levelsets [NM06, HNB*06] do not resort to any brick representation and encode the voxels in the dynamically evolving narrow band. An out-of-core variant of dynamic tubular grids was presented by Nielsen et al. [NNSM07]. The work on dynamic sparse grids in the context of numerical simulations has grown into the open source C++ library OpenVDB [MLJ*13]. It supports hierarchical adaptive data structures and a number of processing operators working on these structure. Fraedrich et al. [FAW10] use a perspective grid, which discretizes the view frustum to restrict resampling to the visible space, at the cost of recurring resampling for every new view.

Especially in computer games, screen-space approaches for rendering iso-surfaces in SPH data have gained attention due to their real-time capability. Adams et al. [ALD06] render particles as spheres and blend the contributions in the overlapping regions. Müller et al. [MSD07] reconstruct a triangle mesh in screen space from visible surface fragments generated via rasterization, and further improve the surface via mesh smoothing. Smoothing operators that work directly on the depth imprint of the rendered surface particles have been proposed by Coords and Staadt [CS09] and van der Laan et al. [vdLGS09]. In this work we improve on these methods by extending the image de-noising model by Rudin, Osher, and Fatemi [ROF92] (ROF) to work effectively on sphere-based surface representations. We demonstrate the advantages of this approach by comparing results

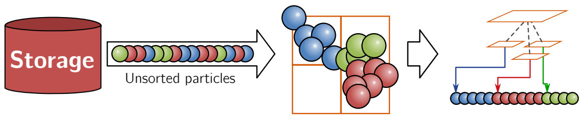


Figure 2: The preprocess: Particles are binned into bricks and an octree is built to encode empty space hierarchically.

to bilateral filtering [TM98, PM90] and screen-space curvature flow [vdLGS09]. To make fluid effects look more realistic, dedicated rendering techniques for pre-classified foam particles have been presented in [IAAT12]. It is one concern of our paper to demonstrate how foam particles can be classified and rendered efficiently for very large particle sets.

3. Data Structures and Algorithms

Our approach begins with an unordered set of particle positions. Additional attributes can be associated to each particle, such as a color index, a velocity index, a size indicator, or any other classifier resulting from the application that generates the particles.

In a preprocess, illustrated in Fig. 2, the particle set is read and binned into a set of equally sized cubical domain partitions, so-called *bricks*. Their size depends on the user-selected resolution of the domain discretization. It generates an array in which particles falling into the same brick are stored in consecutive locations. For every brick, a pointer to its first element in this array is stored. The CPU also builds a *presence-octree* with non-empty bricks at its leaf nodes. The current scene setup is transferred to the GPU, including for every brick its worldspace position and extent, and the octree, the latter enabling empty space skipping during rendering.

All the datasets we use in this work contain particle positions as 32-bit floating point values per component initially. Particle quantization and brick binning throughput is only limited by disk I/O, which is about 40 million particles per second on our test system.

3.1. Binary Voxel Representation

After particle binning, rendering starts instantly without any further preprocessing of the particles. The CPU and GPU work asynchronously during rendering: The CPU traverses all non-empty bricks in front-to-back order and converts the bricks lying inside the view frustum into a *binary voxel representation*, as described below. The specific traversal order mimics the order in which the bricks are rendered on the GPU using parallel ray-casting, so that bricks to be rendered first are converted early.

Our binary voxel representation uses a 3D Cartesian grid to discretize the fluid domain, i.e., every brick is discretized by a grid comprising $n_x \times n_y \times n_z$ cubical cells. The size of a cell in the domain space is set by the user. For instance, in FLIP simulations it is set to the minimum distance below which two particles are rendered as one single particle. In our test scenario shown in Fig. 1, the domain was discretized using a 4096^3 grid and partitioned into bricks of size 64^3. In SPH simulations, the cell size is chosen to be a fraction of the particle smoothing length, assuring that no particle contributions get lost during conversion.

When a brick is selected for conversion, the CPU generates a 3D array that stores one bit (initially set to 0) for every cell of this brick. Then the CPU loops over the set of particles which was associated to the brick in the preprocess. For every particle, it is computed into which cell this particle falls, and the bit of this cell is set. Finally, the bits of contiguous *sub-bricks* of size $4 \times 4 \times 8$ are stored in 4 32-bit integer values. In this way, every sub-brick can be stored in one single element of a texture on the GPU. For instance, when bricks of size 64^3 are used, a 3D texture of size $16 \times 16 \times 8$ stores all bits of one brick, and one texture lookup operation retrieves all bits of one sub-brick. It should be noted that, internally, the conversion process works solely on 3D arrays of integers, using bit operations and address arithmetic.

3.2. Attribute Storage

To handle additional particle attributes as well, we store these attributes compactly and provide a highly efficient access operation to this encoding. The attributes of *non-empty* cells in a sub-brick are written sequentially to a 1D attribute buffer, where the order is determined by the linearized cell indices. A 16-bit pointer to the first entry in this buffer is stored for each sub-brick.

During rendering, the offset of a cell's attribute is determined by counting the number of non-empty cells with lower linearized index inside a sub-brick: For an integer value v encoding one sub-brick, the offset of a cell with linearized index i in this sub-brick is computed as

$$\texttt{baseOffset} + \texttt{countbits}(v \& ((1 << i) - 1)),$$

where `baseOffset` is the pointer to the sub-brick's first attribute, and the function `countbits` counts the number of set bits in an integer value. The function is available on current GPUs, so that, in combination with the binary voxel representation, a highly efficient and bandwidth-oblivious access operation is given.

3.3. Rendering

Rendering on the GPU is performed via single-pass ray-casting inspired by [CNLE09]. The ray-caster writes out surface depth and color information, as well as an image of the volumetric foam in front of the surface. Normals are reconstructed from the surface depth and used in deferred shading. The different images are then composited is illustrated in Fig. 3.

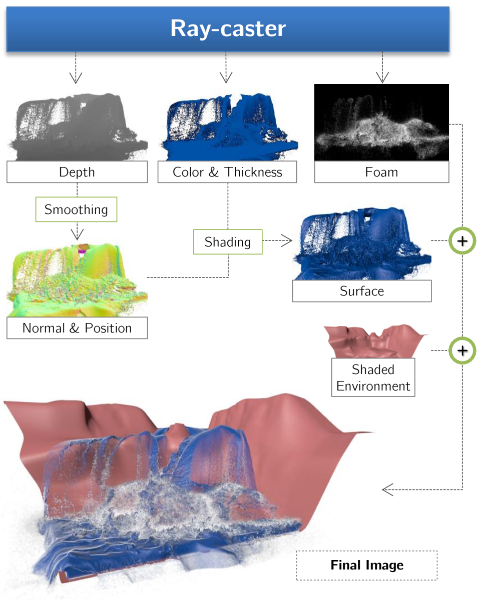


Figure 3: Surface depth, color and foam are output by the ray-caster. Normals and positions are reconstructed from smoothed depth values and used for surface shading. The environment is then blended with the semi-transparent fluid surface and foam.

For every view ray, the ray-caster traverses the presence-octree and determines the non-empty bricks in front-to-back order, rendering them as described below. If a brick's binary voxel representation is not available in GPU memory, the brick index is stored in a request buffer and the ray terminates. At the end of each frame, the request buffer is read to the CPU, and the requested bricks are converted (if not yet done) and sent to the GPU for use in the next frame.

Since the GPU renders in front-to-back order including occlusion culling, only non-occluded bricks are streamed to the GPU. Bricks that are processed on the GPU are cached in video memory using a FIFO strategy.

3.3.1. Sphere Ray-Casting

The ray-caster renders each bit in the binary voxel structure as a sphere of user-defined radius. Since spheres in one brick can overlap adjacent bricks, an n-voxel-wide overlap is stored around each brick. The width of the overlap regions considers the maximum possible sphere radius in units of cell size. This allows for an independent traversal of each brick.

The rays are marched through the bricks in a DDA-like manner, yielding all cells that are hit along their ways. For every cell, its bit is fetched from the binary voxel representation, which also brings the bits of an entire sub-brick containing many adjacent cells into registers. When the bit is set, indicating the presence of a particle, the GPU computes the intersection between the ray and a sphere located at the cell center. To account for the sphere radius, the bits of cells in adjacent sub-bricks need to be fetched, too, and the corresponding particles are tested for intersections. If no intersection occurred, the ray steps to the next cell. Otherwise, the particle color (if available) and the distance of the intersection point to the viewer is written to the color and depth buffer, respectively, and the ray terminates. Figure 4(a) shows a fluid surface rendered as a set of (illuminated) spheres at original particle positions.

3.3.2. Random Jittering

Figure 4(b) shows the same surface as in (a), but particle positions are quantized to the bricks' cell centers. To avoid the visible regular structure introduced by quantization, particle positions are randomly jittered inside the grid cells: When a particle in the cell with index (i,j,k) at time step t is rendered, it is offset by a random vector.The random vector is generated via three TEA hash [ZOC10] operations on the Morton ordered cell index and the current time step, yielding three random offsets for the x, y and z vector components. The jitter is calculated for each particle during rendering, and no additional memory is required. The maximum amount of jitter is limited by the width of the overlap region.

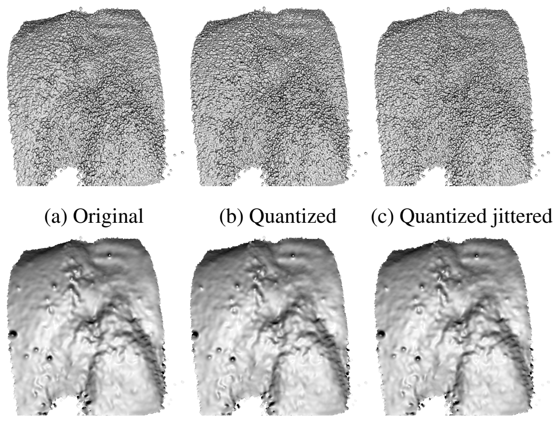


Figure 4: Top row: Direct rendering as spheres. Bottom row: Same as top, but with image-based ROF smoothing applied.

Figure 4(c) shows sphere rendering using position quantization and jittering. Compared to quantization only in (b), the regular structure is broken up entirely, and compared to rendering particles at their original positions in (a), the qualitative surface appearance can be judged as equal. Especially when image-based smoothing and deferred shading is performed, as described next, only very subtle differences in the renderings can be observed.

3.4. Image-based Surface Smoothing

To further enhance the surface's image obtained via sphere rendering, we apply a screen-space post-smoothing filter to the depth buffer. Resulting images are shown in the 2nd row of Fig. 4.

An effective filter has to fulfill the following goals: 1) stronger smoothing closer to the camera and weaker smoothing further away, mimicking object-space filtering in the presence of perspective projections, 2) preservation of both features and depth discontinuities, and 3) reconstruction of a "smooth" surface. While goals 2) and 3) can be achieved with the ROF de-noising model, major modifications are required to achieve the first goal. In the following we will first review the concept underlying ROF following the exposition and solver presented by Chambolle [Cha04], and we will then introduce the specific adaptations to reach our goals.

Total-variation-based image denoising. For a piecewise smooth but unknown image $u : \Omega \to \mathbb{R}$, ROF filtering assumes an additive, stationary Gaussian noise $G(0,\sigma)$, such that an image $v = u + G(0,\sigma)$ is observed. To make the problem tractable, a soft-constrained formulation is generally used to reconstruct u on a continuous domain Ω:

$$\arg\min_{x} \int_{\Omega} \left(\|\nabla u(\mathbf{x})\|_2 + \frac{1}{2\lambda}(u(\mathbf{x}) - v(\mathbf{x}))^2 \right) d\mathbf{x}, \quad (1)$$

where $\int_{\Omega} \|\nabla u(\mathbf{x})\|_2 \, d\mathbf{x}$ denotes the *total variation* of u, and $\lambda \in \mathbb{R}^+$ is an inverse data fidelity parameter dictating the amount of smoothing. Note that, to improve readability, we shorten $u(\mathbf{x})$ to u etc.

To solve Eq. (1), Chambolle presented a fast primal-dual solver that iteratively computes a non-linear projection $\pi_{\lambda K}(v)$ to obtain the smoothed image $u = v - \pi_{\lambda K}(v)$. On 2D domains, two dual variables $\mathbf{p} : \Omega \to \mathbb{R}^2$ are introduced and a gradient descent on the Euler equation yields the following fixpoint iteration:

$$\begin{aligned} \mathbf{p}^{(0)} &:= 0 \\ \mathbf{p}^{(n+1)} &= \frac{\mathbf{p}^{(n)} + \tau(\nabla \text{div } \mathbf{p} - \nabla v/\lambda)}{1 + \|\tau(\nabla \text{div } \mathbf{p} - \nabla v/\lambda)\|_2} \end{aligned} \quad (2)$$

Each iteration thus performs one update on $\mathbf{p}$ for the full domain Ω and $\|\cdot\|_2$ is a 2D vector 2-norm evaluated for each position $\mathbf{x}$ separately. Chambolle shows this to converge for certain values of τ to

$$\pi_{\lambda K} := \lim_{n \to \infty} \lambda \text{div } \mathbf{p}^{(n)} \quad (3)$$

Equation (2) is then discretized and, since ∇ and div are adjoint operators, forward and backward differences are respectively used. Finally, Neumann boundary conditions $(\nabla u)\nu = 0$ are imposed on $\partial\Omega$. Furthermore, Chambolle notes that $\tau = 0.25$ shows the best convergence in practice, although the theoretical reason is unknown.

Local estimate of λ. To mimic object space filtering (goal

1), we assume an *object space* stationary Gaussian density distribution $G_{obj}(0,\sigma)$ around each particle. This, if unfiltered, results in an instationary Gaussian noise $G_{scr}(0,\sigma_{ij})$ after projection, where ij are pixel positions. Non-uniform weights λ_{ij} related to G_{scr} therefore allow us to achieve high-speed previews by representing particles as spheres with $r_{obj} = \sigma = \text{const}$ in object space.

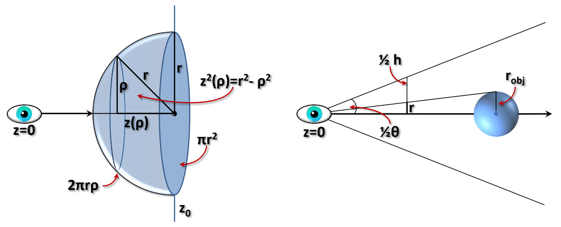


Figure 5: Quantities used in the estimation of the local filter weights λ_{ij}. Left: Estimating Var$[z]$. Right: Estimating the screen space radius r from r_{obj}.

Assuming the viewing ray to hit the sphere at its frontmost point (Fig. 5 left), we estimate the expected value E$[z]$ and variance $\sigma^2 = \text{Var}[z]$ of the depth buffer by integrating over concentric rings:

$$\begin{aligned} \mathrm{E}[z] &= z_0 - \frac{1}{\pi r^2}\int_0^r 2\pi\rho z(\rho)\,d\rho &&= z_0 - \frac{2}{3}r, \\ \mathrm{Var}[z] &= \frac{1}{2\pi r^2}\int_0^r 2\pi\rho\,(z(\rho)-\mathrm{E}[z])^2\,d\rho &&= \frac{1}{18}r^2, \end{aligned} \quad (4)$$

where $r \in \mathbb{R}^+$ is the sphere's radius in screen space. For a perspective projection with field-of-view θ and image height h, r is then estimated using the theorem of parallel lines (Fig. 5 right) as

$$r = r_{obj}\frac{h}{2\tan\left(\frac{\theta}{2}\right)z} \quad = \quad \text{const}\cdot z^{-1}. \quad (5)$$

Therefore, we estimate the screen space noise to $G_{scr}\left(0,\sigma_0/z_{ij}\right)$.

By combining $(v-u)^2 = \int_\Omega d\mathbf{x}G^2(0,\sigma) = |\Omega|\sigma^2$ and $v-u = \pi_{\lambda K} = \lambda\text{div}\,\mathbf{p}$ (Eq. (3)), an optimal λ can be estimated for a known σ using the method of alternating variables [Ber04]:

$$\lambda^{(n+1)} = \frac{\sqrt{|\Omega|}\sigma}{\int_\Omega \|\text{div}\,\mathbf{p}^{(n)}\|_2\,d\mathbf{x}}. \quad (6)$$

In order to avoid the costly per-iteration log-reduce required to compute the norm of div **p**, we assume this norm to be constant for coherent views and time steps. After combining all constants into a new global parameter λ_0, we arrive at the following update at pixel position ij:

$$\begin{aligned} \lambda_{ij}^{(0)} &= \lambda_0\left(z_{ij}+\varepsilon\right)^{-1} \\ \lambda_{ij}^{(n+1)} &= \lambda_0\left(z_{ij}-\lambda_{ij}^{(n)}\text{div}\,\mathbf{p}_{ij}^{(n)}+\varepsilon\right)^{-1}. \end{aligned} \quad (7)$$

Boundary conditions. To apply our filter only to those pixels for which the view ray intersected a sphere (foreground), we keep track of a binary foreground mask. To ensure proper support and boundary conditions for our filter on this oddly shaped domain Ω, we further extrude depth values by one pixel using a 3×3 mask that interpolates from valid pixels.

Implementation. The updates given by Eq. (2) and Eq. (7) can be mapped to a straightforward GPU implementation using compute shaders with one thread per pixel of the viewport. In addition to a linear depth buffer containing z and the foreground mask—which we obtain from one of our G-buffers—we store the two dual variables **p** as well as div **p** and the local λ for each pixel in four floating-point textures.

After fluid rendering in each frame, we initialize **p** and div **p** to 0 and λ to $\lambda^{(0)}$ given by Eq. (7) for each pixel. The depth buffer is extruded by one pixel without modifying the foreground mask. We then repeatedly perform two passes for a user-defined number of iterations: First, both **p** are updated according to Eq. (2), where the required derivatives are approximated using forward differences. Afterwards, div **p** is updated from backward differences of **p**, and the new local lambda is calculated following Eq. (7).

After these iterations, we obtain the final depth value for foreground pixels by subtracting λ div **p** from the input depth value according to Eq. (3). Invisible non-foreground pixels are not updated. The final depth is stored in the depth buffer, which is then used in deferred shading for normal and position calculation.

3.5. Foam Classification and Accumulation

Once a requested brick has been converted and uploaded to the GPU, further processing of the quantized particles can be performed. Here, the binary voxel representation allows highly efficient operations, such as neighbor searches, which are very well suited to the GPU's design for massively parallel workloads. In the following we describe how to use this operation for classifying isolated particles. They can either be removed to expose the surface or remove noise, or they can be rendered as semi-transparent foam atop of the surface to give the fluid a more realistic appearance.

We propose to classify particles as foam depending on the number of neighboring particles in a certain surrounding. Isolated particles having few neighbors are selected as foam, whereas particle having more than a user-defined threshold of neighbors are selected as surface particles. We found that classifying particles as foam when less than 20% of the cells inside the particle radius are occupied yields visually plausible results in our test cases. Figure 6 shows the result of the classification (and foam rendering) for one time step of a FLIP fluid simulation.

When foam rendering is enabled, every visible brick is

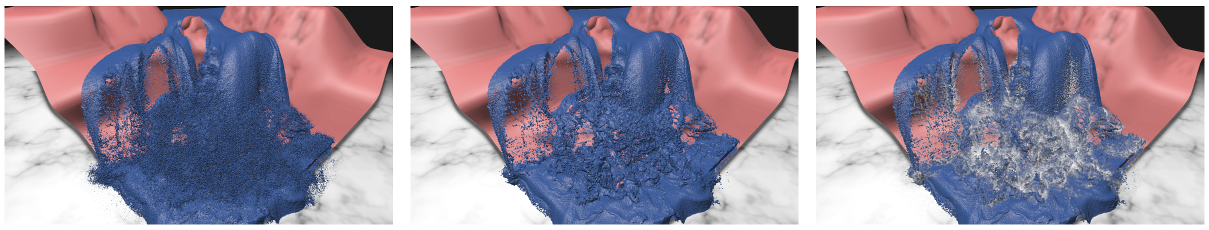


Figure 6: Isolated particles (left) may clutter the view on the actual fluid surface. These particles can be determined and either removed (middle) or optionally rendered as volumetric foam (right).

processed on the GPU after uploading to classify foam particles. Batches of 256 bricks are processed in a single kernel with one thread per sub-brick. Each thread gathers the neighboring sub-bricks, counting, for each cell, the amount of set bits inside the selected search radius. For each classified foam cell, a single bit is set in a separate *foam brick* using the same storage format. Since a sub-brick is exclusively processed by a single thread, no atomic operations are required and the result can be written in a single texture access once all cells have been classified.

Rendering of foam particles is performed via fixed-step volume ray-casting, by extracting, at every sampling point, the local particle density and accumulating these densities using a simple emission-absorption model. The process works by dividing every sub-brick into two 4^3 blocks, and by computing their density as the number of bits set (see Fig. 7). Since sub-bricks are encoded as 4 32-bit integer values, we can use a simple popcount on two integer values accessed in one texture fetch operation. The density at the sampling point is then determined by trilinear interpolation between adjacent cells. When volumetric foam is rendered, early ray-termination is performed when the fluid surface is reached or the opacity along a ray has reached a value of 0.95.

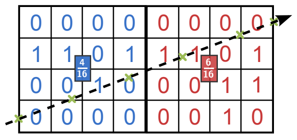


Figure 7: Foam density calculation: In 2D, two 4^2 bit-fields of a (4×8) sub-brick are queried fetched, and the number of set bits in either field is used as density value.

3.6. Transparencies

To further enhance the visual appearance of the fluid, translucent materials can be approximated with a fast screen-space approach [CLT07]. First, the shaded scene geometry is rendered into a separate background buffer (see Fig. 3). Then, when shading the fluid surface, we blend the contents of this buffer with the fluid color. The texture coordinates used in the background buffer lookup are given by the position of the currently shaded pixel. We distort these coordinates by offsetting them depending on the fluid's smooth surface normal to approximate refractions.

In addition, absorption is calculated by tracing each ray through the fluid body, using our proposed density calculation (see Fig. 7) until maximum absorption is reached. The resulting absorption value determines the amount of background color blended with the fluid surface color. Fig. 8 illustrates the effect.

It should be clear that, since we are using front to back ray-casting and can reconstruct accurate fluid surface normals by evaluating the particles' smoothing kernels, our approach can also simulate correct multi-bounce refractions and reflections. However, due to efficiency reasons we have not considered this option in the current work.

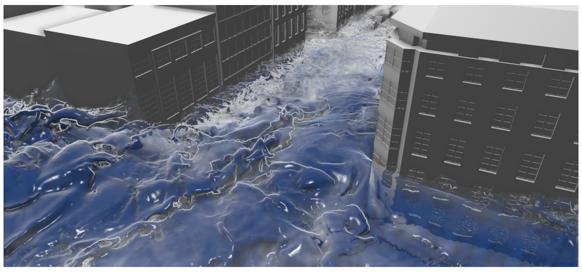


Figure 8: Transparency is determined by the absorption along each ray and combined with screen-space refractions.

4. Design Decisions and Tradeoffs

We now consider some of the decisions made in the design of our system to make it suitable for rendering very large particle data, including a discussion and comparison of alternative design choices. In particular, we want to emphasize the possible tradeoffs that allow the user to choose between high quality and speed.

4.1. Binary Voxel Representation

Our binary voxel representation consumes one bit per cell of the adaptive spatial grid. Thus, if the fill rate of a brick is below $1/24$, in terms of memory requirements it would be more efficient to encode particle positions explicitly using 24 bits, that is, via three block-relative 8-bit coordinates. Our statistics demonstrate, however, that in practice only a small number of the non-empty bricks show this property, and that even compared to a 24-bit position encoding we achieve very good compression rates. Furthermore, in contrast to an explicit encoding, our approach enables efficient particle processing and parallel rendering algorithms, and provides coherent memory access operations due to the specific sub-brick encoding.

For our statistics we use four FLIP fluid simulations. They are described in Table 1 and Fig. 11. The `Dam` dataset contains particle positions and velocities. The velocity magnitude is mapped to colors during brick conversion and stored as a particle attribute. Thus, we include one 16-bit pointer to the color array for every sub-brick in the memory requirements for `Dam`. All other datasets contain only positions. The compression ratio includes foam classification which is calculated upon GPU upload and not cached on the CPU, which effectively yields an additional compression factor of 1:2 for timesteps residing in CPU main memory.

Table 1: Dataset statistics. 'Frm' and 'Part' give the number of simulation frames and maximum particles per frame. 'Res' is our selected grid resolution, 'Mem' gives the memory of the original sequence, 'Ratio' is the compression ratio we achieve, and 'Bpp' gives the number of bits per particles our binary voxel representation requires.

Name	Frm	Part	Res	Mem	Ratio	Bpp
Step	330	430M	4096^3	840 GB	1:18	5.85
Fall	366	180M	2048^3	700 GB	1:10	10.78
Dam	100	180M	1024^3	192 GB	1:20	4.55
City	223	500M	4096^3	588 GB	1:29	2.46

In particular, this shows that even the largest fluid simulation we preview can be stored in CPU RAM entirely. For browsing through the animation it is important that a huge fraction of all time steps can be stored in GPU video memory at once.

4.2. Bricked Representation

The bricked data representation is necessary to restrict data processing and rendering to those parts of the data which are in the view frustum and not occluded by others. Larger bricks cannot adapt so well to the areas where the fluid surface or splashes occur, and therefore encode larger empty areas. On the other hand, our performance analysis in Section 5 indicates that larger bricks result in a better throughput in cells per second. This is confirmed in Fig. 9, which shows for the views in Fig. 11 decreasing rendering times, but increasing memory requirements with increasing brick size. According to these tests we have chosen a brick size of 64^3 as a good balance between rendering and processing times, and memory consumption.

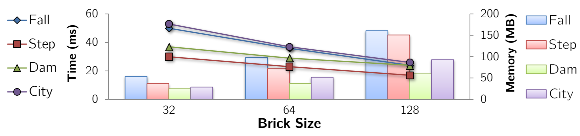


Figure 9: Influence of the brick size on rendering time and memory requirement.

An important extension is the use of an additional mipmap-structure for every converted brick on the GPU. For a particular brick, it stores at every coarse level cell a bit indicating the empty sub-space in this brick. Mipmap generation works similar to foam classification on the GPU. Especially when large scale splashes are simulated, performance speedups of up to a factor of 15 can be achieved by using mipmaps. The reason for this extreme gain is the significantly reduced number of neighbor search operations that have to be performed along the view rays.

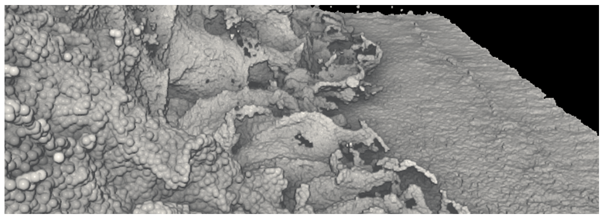
Note that our approach does not make use of a level-of-detail particle representation to reduce the number or rendered particles in every frame. This is possible in general, for instance, by using particle merging strategies as proposed in [FSW09], yet the particular strategies for fluid and foam particles in combination with sphere rendering need some further investigation.

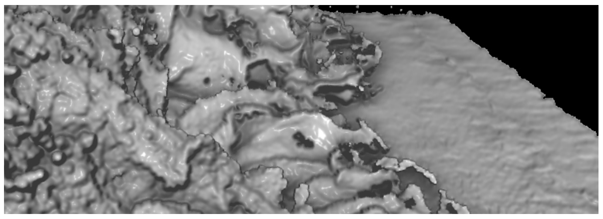
4.3. Position Quantization

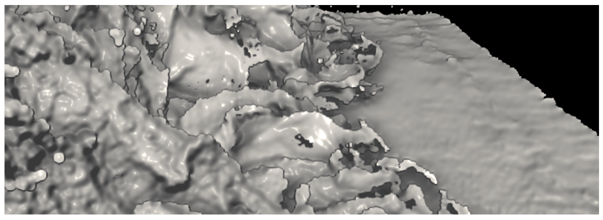
The conversion process quantizes particle positions to the cell centers of a regular 3D grid. Even though we can avoid the regular structure in the particle distribution by adding a random jitter to the quantized positions, quantization always introduces some loss of positional accuracy. However, this does not introduce any significant artifacts due to the following reasons. Firstly, since the binary voxel representation is very compact, a high resolution discretization of the domain can be chosen to make the quantization error very small. Secondly, when rendering spheres and image-based post-smoothing, only an approximation of the fluid surface is displayed, rendering quantization errors negligible (see Fig. 4).

4.4. Volume Ray-Casting

There are a number of reasons for using volume ray-casting in our previewing system: Firstly, the ray-caster can render from the compact binary voxel representation directly, without having to convert the binary data into a renderable vertex representation as required by rasterization. Secondly, compared to rasterization-based particle rendering, much better rendering rates can be achieved. When rasterizing particles


(a) No smoothing


(b) Curvature flow, 75 iterations, 18 ms


(c) Bilateral, 15 ms

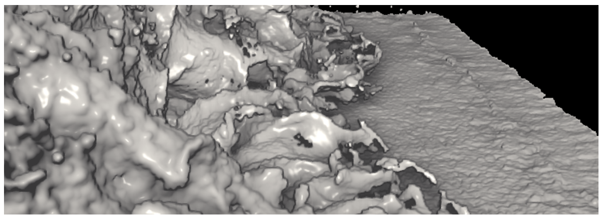

(d) Our method, 75 iterations, 13 ms

Figure 10: Screen-space smoothing comparison

on recent GPUs as screen-space aligned quads consisting of two triangles, about 250M particles can be rendered in roughly 250 ms, not including the time required to realize the spheres' perspectively correct depth footprints in a pixel shader. Using the same particle size, the ray-caster renders 3D spheres about a factor of 10 faster. Thirdly, the ray-caster can exploit occlusion culling at the pixel level, which is problematic when using rasterization. Occlusion queries in rasterization can only be effective at a rather coarse granularity, i.e., if parts consisting of many polygons can be culled at once. Finally, due to the front-to-back traversal order, the ray-caster can render semi-transparent particles like foam without any modifications. The rasterizer, in contrast, either needs to sort the particles or the generated fragments per frame to ensure correct front-to-back blending.

4.5. Image-based Smoothing

We compared our proposed smoothing filter to two popular screen-space filters: bilateral filtering [PM90] and curvature flow [vdLGS09]. Figure 10 shows the same data set and view processed with the three different filters to demonstrate the differences.

Our filter has the distinct advantage of having a fixed and very compact stencil size. This stencil size is independent of the desired amount of smoothing. Thus, the filter has very light memory bandwidth requirements and outperforms even simple (i.e., non-depth-adaptive) bilateral filtering. While the bilateral filter can also be extended to locally adapt the radius to a desired world-space filter size, this imposes severe performance constraints as the filter is not serperable. Furthermore, iterative application of smaller bilateral filters as well as separable bilateral filters [PV05] are approximations that may give suboptimal and undesirable results, such as overly flat surfaces and overly sharp depth discontinuities.

Compared to curvature flow, our filter is better at preserving sharp features and depth discontinuities, and the amount of smoothing does not come from adding iterations, but from altering the smoothing parameter λ_0.

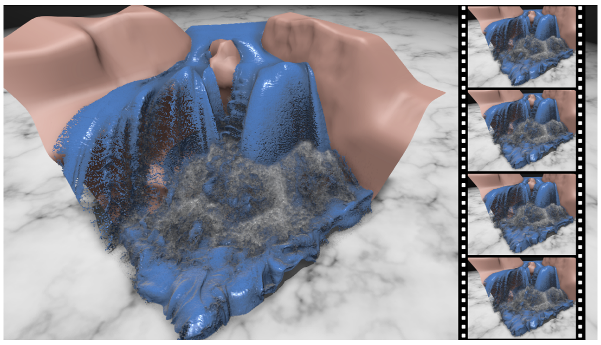
5. Performance

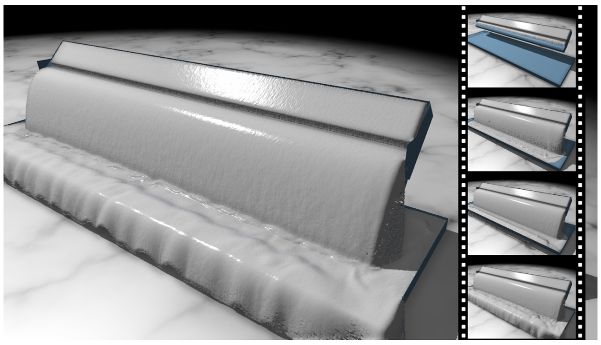
Here we evaluate the performance of all components of our system and provide accumulated timings for the most time-consuming operations. All timings were performed on a dual quadcore Intel Xeon X5560 with 48 GB main memory, and an NVIDIA GTX Titan with 6 GB video memory. Data was accessed from a solid state drive delivering up to 500 MB/s read peak-performance. We used a viewport resolution of 1280 $\times$ 720 pixels for all rendering passes. All times given in this section are in milliseconds.

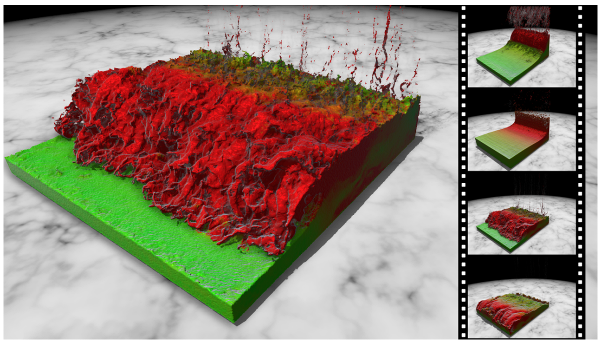
Table 2 gives average times for the operations performed during previewing. The times for brick sizes 32^3, 64^3, and 128^3 are compared. The particle radius was set to 2 cells.

Table 2: Per brick processing statistics for different brick sizes. 'Conv', 'Foam', and 'Mip' are the times for CPU conversion, foam classification, and mipmap generation, respectively.

Size	Conv	Foam	Mip
32	.06 $\pm$ .02	.02 $\pm$ .01	.01
64	.22 $\pm$ .06	.15 $\pm$ .08	.04
128	.91 $\pm$ .37	1.06 $\pm$ .41	.27


(a) Fall, 180M particles


(b) Step, 430M particles


(c) Dam, 180M particles

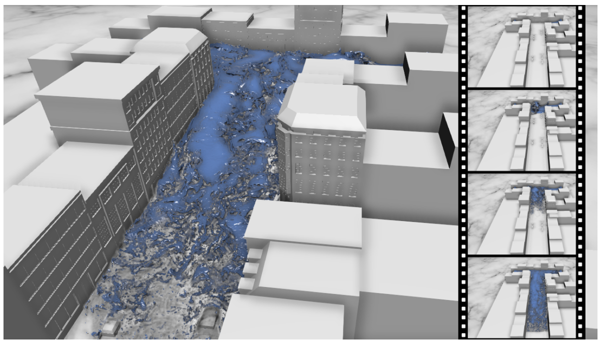

(d) City, 500M particles

Figure 11: Test datasets.

Table 3: Memory and performance statistics for the views in Fig. 11. Listed are consumed GPU memory ('Mem') for surface and foam, the times for sphere ('Sphere') and foam ('Foam') rendering, absorption estimation for transparencies ('Trans'), and post-smoothing ('Smooth').

Data	Mem	#Bricks	#SurfParticles	#FoamParticles	Sphere	Foam	Trans	Smooth
Fall	210 MB	3175	46.6M	4.9M	27 ms	34 ms	14 ms	13 ms
Step	157 MB	2366	254.2M	27.8M	20 ms	12 ms	12 ms	13 ms
Dam	83 MB	1162	95.1M	6.8M	23 ms	22 ms	13 ms	16 ms
City	127 MB	1919	213.3M	45.6M	24 ms	16 ms	12 ms	13 ms

The statistics were performed using the bricks in all datasets and computing the times required for processing each of them. We give the average times and the interval in which the times vary. Only mipmap generation does not depend on the brick's fill rate and is constant for a fixed size. The times of all other operations vary depending on a brick's fill rate.

Another exemplary experience shows the impact of the particle influence radius on the rendering and processing times. Here we used bricks of size 64^3, and we analyzed sphere rendering and foam classification as the representative operations. Figure 12 shows exponentially increasing times due to the increasing number of neighbor particles to be considered in both operations.

To analyze the concrete previewing times for our test datasets, Table 3 summarizes statistics for average views where the whole dataset is visible on screen. Experiments were only performed once for the given viewport size, because all rendering times scale linearly in the number of pixels. We assume that all converted bricks, foam bricks and brick mipmaps are residing in GPU cache. Density bricks are resampled on the fly.

The first observation is that only a small fraction of all bricks need to be accessed to generate the views. This is because occluded bricks do not have to be touched. Another observation is that the rendering of volumetric foam has a significant effect on rendering performance, as seen for the turbulent datasets exhibiting violent splashes and foam. In

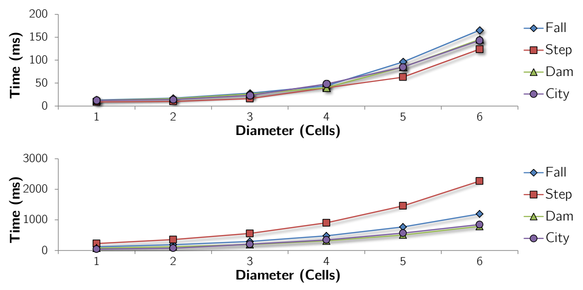


Figure 12: Influence of particle radius on rendering (top) and foam classification (bottom) performance.

this case the view-rays have to be traversed through the volume for a much longer distance compared to the rendering of an opaque surface. Post-smoothing using 75 iterations on the GPU requires about half the time sphere rendering does.

System Behavior. The responsiveness of our system was analyzed in a previewing session, where a single time step is explored by the user interactively, performing a number of view changes and zooms. For each frame we measured the *frame time* until all required bricks are converted, processed, and rendered. Furthermore, we counted the number of requested bricks per frame. Disk I/O is *not* included in the timings, and the system started from "cold caches".

The performance graphs for the four data sets in Fig. 13 show the measured frame times, the required GPU memory, and the number of requested bricks. A peak in the frame time is observed at the beginning. This peak results from the particular strategy we use on the GPU to request bricks not yet residing in video memory. After each frame, the request buffer is read back to the CPU and requested bricks are converted and uploaded to the GPU. Since a ray terminates as soon as a brick not yet available on the GPU should be rendered, this rendering can only be performed in the next frame. Thus, at the beginning, when no bricks are cached in GPU memory, our strategy results in a delay of several frames. The number of frames is equal to the maximum number of bricks that are requested by a ray. Nonetheless, one can see that the first frame requires only roughly 700 ms. Note that this time also gives the rate by which one can scrub through an entire fluid sequence, where in every frame the GPU cache is clear and bricks are newly requested.

The graphs showing for each frame the number of requested bricks and the memory requirements reveal the advantages of embedding the compact particle representation into front-to-back ray-casting. Since only non-empty and non-occluded bricks are rendered, a rather small sub-set of all bricks needs to be uploaded and processed on the GPU. Furthermore, because the ray-caster can render directly from the binary particle representation, avoiding any data conversion on the GPU, the memory consumption on the GPU is kept very small.

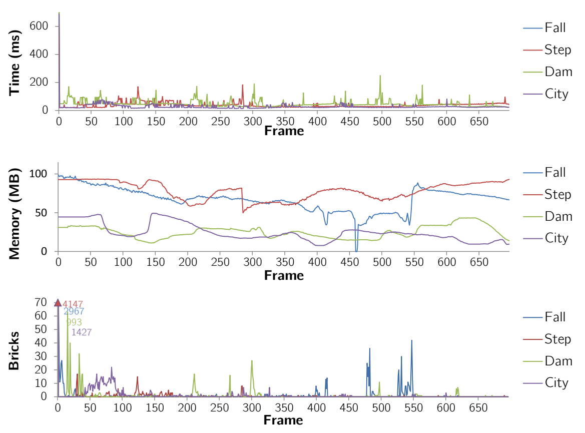


Figure 13: System behavior during data exploration of a single time step with different views and zooms. Note that the graph for uploaded bricks in the left column is capped at 50 for better visualization, with the number of bricks in the first frame explicitly labeled.

6. Conclusion

We have presented a rendering system for very large particle-based fluid simulations. Due to the compact particle representation and the intertwining of particle processing and rendering we could demonstrate interactive previews of entire fluid sequences. Our approach is able to display a smooth fluid surface, in particular due to the use of an improved image post-smoothing method. We could demonstrate interactive previews of simulation data comprising up to 500 millions of particles per time step.

In the future we plan to extend our system towards a high-quality rendering system including advanced fluid and foam illumination effects. This is possible in principle, because our implemented neighbor search operations can be used to evaluate particle kernels and compute accurate density volumes and surfaces therein. To avoid performing huge numbers of operations in the fluid body, we will further investigate the construction of effective space-leaping techniques on the fly during rendering.

References

[AIAT12] Akinci G., Ihmsen M., Akinci N., Teschner M.: Parallel surface reconstruction for particle-based fluids. *Comp. Graph. Forum 31*, 6 (2012), 1797–1809. 3

[ALD06] Adams B., Lenaerts T., Dutre P.: *Particle Splatting: Interactive Rendering of Particle-Based Simulation Data*. Technical report cw453, Katholieke Universiteit Leuven, 2006. 3

[APKG07] Adams B., Pauly M., Keiser R., Guibas L. J.: Adaptively sampled particle fluids. *ACM Trans. Graph. 26*, 3 (2007), 48. 3

[Ber04] Bertsekas D. P.: *Nonlinear Programming*, 2nd ed. Athena Scientific, 2004. 6

[Bri03] BRIDSON R. E.: *Computational Aspects of Dynamic Surfaces*. PhD thesis, 2003. 3

[Bri08] BRIDSON R.: *Fluid Simulation for Computer Graphics*. A K Peters, 2008. 2

[Cha04] CHAMBOLLE A.: An algorithm for total variation minimization and applications. *J. Math. Imaging Vis. 20*, 1-2 (2004), 89–97. 5

[CLT07] CRANE K., LLAMAS I., TARIQ S.: *GPU Gems*, vol. 3. Addison-Wesley Professional, 2007, ch. Real-Time Simulation and Rendering of 3D Fluids, pp. 633–675. 7

[CNLE09] CRASSIN C., NEYRET F., LEFEBVRE S., EISEMANN E.: Gigavoxels: ray-guided streaming for efficient and detailed voxel rendering. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games* (2009), pp. 15–22. 4

[CS09] CORDS H., STAADT O. G.: Interactive screen-space surface rendering of dynamic particle clouds. *J. Graphics, GPU, &Game Tools 14*, 3 (2009), 1–19. 3

[FAW10] FRAEDRICH R., AUER S., WESTERMANN R.: Efficient high-quality volume rendering of sph data. *IEEE Transactions on Visualization and Computer Graphics 16*, 6 (2010), 1533–1540. 3

[FSW09] FRAEDRICH R., SCHNEIDER J., WESTERMANN R.: Exploring the Millenium Run - scalable rendering of large-scale cosmological datasets. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1251–1258. 8

[GEM*13] GOSWAMI P., EROL F., MUKHI R., PAJAROLA R., GOBBETTI E.: An efficient multi-resolution framework for high quality interactive rendering of massive point clouds using multi-way kd-trees. *The Visual Computer 29*, 1 (2013), 69–83. 2

[GIK*07] GRIBBLE C. P., IZE T., KENSLER A., WALD I., PARKER S. G.: A coherent grid traversal approach to visualizing particle-based simulation data. *IEEE Transactions on Visualization and Computer Graphics 13*, 4 (2007), 758–768. 3

[GSSP10] GOSWAMI P., SCHLEGEL P., SOLENTHALER B., PAJAROLA R.: Interactive sph simulation and rendering on the gpu. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2010), pp. 55–64. 1, 3

[HNB*06] HOUSTON B., NIELSEN M. B., BATTY C., NILSSON O., MUSETH K.: Hierarchical RLE level set: A compact and versatile deformable surface representation. *ACM TOG 25*, 1 (2006), 151–175. 3

[IAAT12] IHMSEN M., AKINCI N., AKINCI G., TESCHNER M.: Unified spray, foam and air bubbles for particle-based fluids. *The Visual Computer 28*, 6-8 (2012), 1–9. 3

[IOS*14] IHMSEN M., ORTHMANN J., SOLENTHALER B., KOLB A., TESCHNER M.: Sph fluids in computer graphics. In *Eurographics 2014 State-of-the-Art Report* (2014), Eurographics Association, pp. 1–22. 2

[KSN08] KANAMORI Y., SZEGO Z., NISHITA T.: Gpu-based fast ray casting for a large number of metaballs. *Computer Graphics Forum 27*, 2 (2008), 351–360. 3

[KSW05] KRÜGER J., SCHNEIDER J., WESTERMANN R.: Duodecim - a structure for point scan compression and rendering. In *Proceedings of the Symposium on Point-Based Graphics 2005* (2005). 2

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph. 21*, 4 (1987), 163–169. 2

[MCG03] MÜLLER M., CHARYPAR D., GROSS M.: Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2003), SCA '03, Eurographics Association, pp. 154–159. 2, 3

[MLJ*13] MUSETH K., LAIT J., JOHANSON J., BUDSBERG J., HENDERSON R., ALDEN M., CUCKA P., HILL D., PEARCE A.: OpenVDB: an open-source data structure and toolkit for high-resolution volumes. In *ACM SIGGRAPH Courses* (2013), pp. 19:1–19:1. 3

[MSD07] MÜLLER M., SCHIRM S., DUTHALER S.: Screen space meshes. In *SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2007), pp. 9–15. 1, 3

[NM06] NIELSEN M. B., MUSETH K.: Dynamic tubular grid: An efficient data structure and algorithms for high resolution level sets. *J. Sci. Comput. 26*, 3 (2006), 261–299. 3

[NNSM07] NIELSEN M. B., NILSSON O., SÖDERSTRÖM A., MUSETH K.: Out-of-core and compressed level set methods. *ACM Trans. Graph. 26*, 4 (2007). 3

[PM90] PERONA P., MALIK J.: Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence 12* (1990), 629–639. 3, 9

[PV05] PHAM T. Q., VLIET L. J. V.: Separable bilateral filtering for fast video preprocessing. In *IEEE Intl. Conf. on Multimedia and Expo (ICME)* (2005), pp. 4pp.–. 9

[RB08] ROSENBERG I. D., BIRDWELL K.: Real-time particle isosurface extraction. In *Proceedings of the 2008 symposium on interactive 3D graphics and games* (2008), pp. 35–43. 3

[ROF92] RUDIN L. I., OSHER S., FATEMI E.: Nonlinear total variation based noise removal algorithms. *Phys. D 60*, 1-4 (1992), 259–268. 2, 3

[SSP07] SOLENTHALER B., SCHLÄFLI J., PAJAROLA R.: A unified particle model for fluid-solid interactions: Research articles. *Comput. Animat. Virtual Worlds 18*, 1 (2007), 69–82. 3

[TM98] TOMASI C., MANDUCHI R.: Bilateral filtering for gray and color images. In *Proc. Intl. Conf. on Computer Vision* (1998), IEEE, pp. 839–846. 3

[vdLGS09] VAN DER LAAN W. J., GREEN S., SAINZ M.: Screen space fluid rendering with curvature flow. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games* (2009), pp. 91–98. 1, 3, 9

[YHK09] YASUDA R., HARADA T., KAWAGUCHI Y.: Fast Rendering of Particle-Based Fluid by Utilizing Simulation Data. In *Proceedings of Eurographics 2009 - Short Papers* (2009), Alliez P., Magnor M., (Eds.), Eurographics Association, pp. 61–64. 3

[YT13] YU J., TURK G.: Reconstructing surfaces of particle-based fluids using anisotropic kernels. *ACM Trans. Graph. 32*, 1 (2013), 5:1–5:12. 3

[ZB05] ZHU Y., BRIDSON R.: Animating sand as a fluid. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers* (2005), pp. 965–972. 2, 3

[ZOC10] ZAFAR F., OLANO M., CURTIS A.: Gpu random numbers via the tiny encryption algorithm. In *Proceedings of the Conference on High Performance Graphics* (Aire-la-Ville, Switzerland, Switzerland, 2010), HPG '10, Eurographics Association, pp. 133–141. 5

[ZSP08] ZHANG Y., SOLENTHALER B., PAJAROLA R.: Adaptive sampling and rendering of fluids on the gpu. In *Symposium on Point-Based Graphics* (2008), pp. 137–146. 3