Eurographics Symposium on Parallel Graphics and Visualization (2015)
C. Dachsbacher, P. Navrátil (Editors)

Memory-Efficient On-The-Fly Voxelization of Particle Data

Tobias Zirr[1] and Carsten Dachsbacher[1]

[1]Karlsruhe Institute of Technology

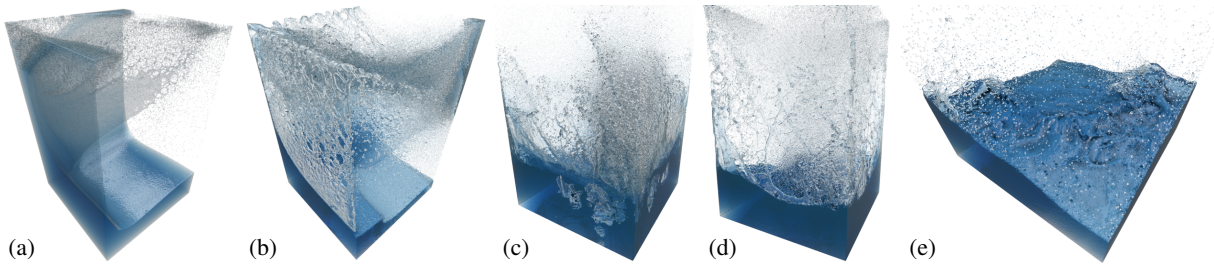

Figure 1: *Our novel voxelization and rendering method used with a Smoothed Particle Hydrodynamics simulation computed using 20 million particles. The view-adaptive voxelization comprises about 30 billion voxels but can be carried out and used for interactive rendering on a single GPU due to our tiled-based voxelization and on-the-fly compression.*

Abstract

In this paper we present a novel GPU-friendly real-time voxelization technique for rendering homogeneous media that is defined by particles, e.g. fluids obtained from particle-based simulations such as Smoothed Particle Hydrodynamics (SPH). Our method computes view-adaptive binary voxelizations with on-the-fly compression of a tiled perspective voxel grid, achieving higher resolutions than previous approaches. It allows for interactive generation of realistic images, enabling advanced rendering techniques such as ray casting-based refraction and reflection, light scattering and absorption, and ambient occlusion. In contrast to previous methods, it does not rely on preprocessing such as expensive, and often coarse, scalar field conversion or mesh generation steps. Our method directly takes unsorted particle data as input. It can be further accelerated by identifying fully populated simulation cells during simulation. The extracted surface can be filtered to achieve smooth surface appearance.

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

1. Introduction

Particle-based simulation is a versatile tool with numerous applications. In computer graphics, for example, Smoothed Particle Hydrodynamics (SPH) is used for simulating fluids in movie productions and games; in molecular dynamics, particle-based simulations investigate molecular structure, dynamics, and thermodynamical properties. In all these cases, the simulation results in a (typically large) number of particles which represent physical matter, e.g. a fluid. For rendering, the surface of this body has to be extracted either as a polygonal mesh or another representation that can be used with image generation methods. In this paper we focus primarily on SPH simulation data from which we extract surfaces. Nevertheless, our method is likewise applicable to rendering of other particle-based simulation data.

SPH simulations nowadays comprise a large number (often tens of millions [IABT11]) of particles, and interestingly, the rendering, in particular the extraction of a high-quality surface, becomes the bottleneck. The classical approach to this is using Marching Cubes (MC) [LC87] to obtain a polygonal surface. Various approaches have been presented to obtain the therefor required scalar field that implicitly describes the surface [ZB05, APKG07, SSP07, YT13, AIAT12]. This step takes up most of the computation time, but MC-based approaches also struggle with mesh quality which is strongly dependent on the MC grid size and the particles' influence radii in the scalar field. Implicit surfaces can also be rendered directly using ray casting [KSN08], splatting [vdLGS09] and volumetric rendering [FAW10], however, this typically either limits rendering to only basic techniques or requires additional preprocessing.

DOI: 10.2312/pgv.20151150

In this paper we present a method which enables view-adaptive high-resolution voxelization of SPH particle data, i.e. the surfaces discriminating the fluid and the surrounding medium. It achieves interactive speed for millions of particles on a single GPU and enables rendering with complex illumination effects such as reflection and refraction or translucency. We show results for voxelizations at $1920 \times 1080 \times 16000$ (for interactive preview rendering) and at up to $3840 \times 2160 \times 16000$ (which would amount to 13.8 gigabytes uncompressed) for fast supersampled preview rendering, both on a single GPU. We address the problem of high-resolution volumetric data exceeding the available memory with a tiled voxelization that is instantaneously compressed for later use in rendering and ray casting.

2. Related Work

Rendering the results of SPH simulations requires the reconstruction of surfaces from the set of particles. The existing techniques can be roughly classified into the following categories: (1) polygonization of an isosurface of a scalar field defined by the particles, (2) mesh advection, (3) direct and screen space rendering of the isosurface, and (4) volume rendering approaches. A recent state-of-the-art report [IOS*14] provides a comprehensive overview of this topic and we thus restrict ourselves to a brief discussion to provide context.

Polygonization In SPH (and other particle-based) simulations a scalar field can be obtained by superimposing kernel functions of the individual particles. An isosurface of this scalar field can be polygonized using Marching Cubes (MC) [LC87] (or related approaches). The mesh quality and smoothness depends on various aspects, such as the MC grid resolution [AIAT12] and the computation of the scalar field itself. The simplest way for the latter is to superimpose Gaussian potentials [Bli82] which, however, may result in distracting bumps with irregular particle distributions. More elaborate approaches obtain smoother surfaces by computing the scalar field based on the weighted average of nearby particles [ZB05, APKG07, SSP07, AIAT12], possibly using anisotropic kernels [YT13]. The computation of the scalar field is crucial for the performance of the overall rendering and thus several optimizations have been developed, e.g. reconstructing the scalar field only in the proximity of the fluid surface [AIAT12], mesh decimation [AAIT12], and adaptivity using hierarchical grids [AAOT13]. Polygonization approaches typically require significant computation time and memory, and often involve subsequent mesh decimation to reduce the number of primitives and adapt the mesh resolution to the viewpoint.

Mesh Advection Instead of extracting meshes for isosurfaces in each frame, another approach is to use explicit meshes which are advected (updated) over time [PTB*03, HH09, YWTY12]. Note that these methods introduce significant overhead for maintaining the polygonal meshes.

Direct and Screen Space Rendering All aforementioned approaches are typically not feasible for interactive rendering (without significant loss of quality). Screen space methods operate in 2D image space with the primary goal of extracting a depth map of the frontmost surface by rendering the particles (e.g. as spheres), possibly followed by smoothing using a (separated) bilateral filter [Gre10], according to mean curvature [vdLGS09, BSW10], or using local fitting [GRDE10]. Accounting for further surfaces (behind the frontmost) requires depth peeling [KSN08]. Szécsi and Illés raycast metaballs by storing them in per-pixel fragment lists and computing the intersection afterwards [SI12]. Polygonization can also be avoided when directly computing the intersection of (view) rays and the isosurface [GSSP10] or particle spheres [GIK*07]. Our approach is related to screen space approaches as we also compute a view-dependent representation. However, we obtain a truly volumetric representation which is able to handle large numbers of particles without costly depth peeling.

Volume Rendering In order to use more elaborate lighting techniques (e.g. transmittance along light or shadow rays), or to render mass flow inside fluids, volume rendering techniques are required. SPH particle data is often transformed into a volumetric representation using splatting-slicing approaches [NMM*06, vdLGS09, FAW10]. Similar to our method, Fraedrich et al. [FAW10] use a view-dependent perspective grid for computing a voxelization of SPH data. In contrast to our approach, they generate and immediately render slabs of the grid, which enables storing scalar data (while we assume homogeneous media). However, the resolution of their method is severely limited by the available GPU memory and multi-layer information does not remain accessible, disallowing secondary ray casting.

Compression This work makes use of a kind of run-length encoding that is somewhat similar to the multi-layer runlength-encoded orthogonal framebuffer used by Reichl et al. [RCBW12] to enhance a coarse voxelization of polygonal meshes with detail surface samples for accelerated hybrid rendering. However, their encoding runs at preprocessing time and uses depth peeling, which is impracticable for large quantities of overlapping particles. With respect to SPH visualization, Reichl et al. [RTW13] describe an out-of-core octree data structure enhanced with wavelet compression to allow for visualization of very large data sets (billions of particles). However, they require hours of preprocessing and cast only primary rays.

3. Algorithm

Our voxelization and rendering algorithms comprise distinct stages which are all executed on the GPU and depicted in Fig. 2. The first two steps operate in a tile-based mode which enables us to voxelize at resolutions which would otherwise not be possible with the nowadays available GPU memory. The algorithm stages are:

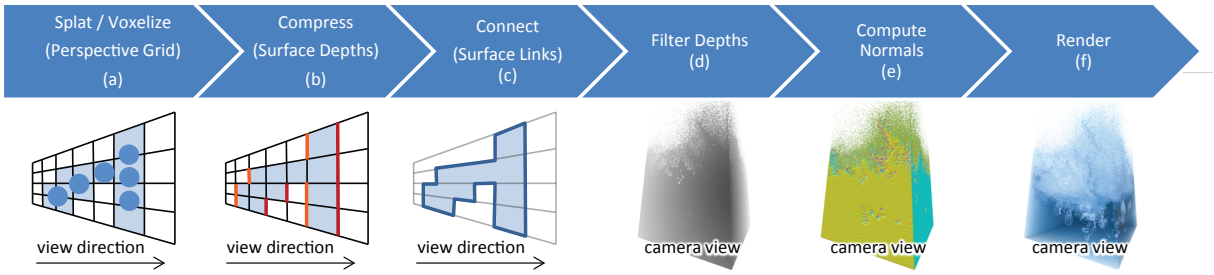

Figure 2: *Our voxelization operates in three stages (from left to right): a) voxelization of particles from the simulation, b) compression of contiguous voxels in* voxel columns*, c) determination of surfaces across voxel columns. Prior to rendering, d) we apply an iterative filter to the interval depths and e) compute normals.*

(a) **Splat:** Determine occupied voxels of a view-aligned perspective grid by projecting particle spheres onto the screen tile, optionally accelerated with additional information from the simulation grid.
(b) **Compress:** Scan each pixel's voxels along the view direction for occupied voxels, store distance to surfaces (entries and exits).
(c) **Surface Connection:** Identify and connect overlapping depth intervals in neighboring pixels to surfaces.
(d) **Filter:** Apply smoothing on connected surfaces' depth.
(e) **Normal generation:** Apply Sobel operator on connected surface depths to obtain surface normals.
(f) **Render:** Render images (optionally with ray casting for secondary rays) using the multi-layer depth and normal information available in the voxelization.

3.1. Tiled Voxelization and Compression

We construct our view-adaptive perspective voxel grid at full render resolution, typically using $\sim$16.000 depth layers ($\sim$30 billion voxels at 1080p). To achieve a resolution this high within GPU memory bounds, we split the screen into smaller tiles whose subgrids comfortably fit into video memory. We denote the voxels along the depth dimension that belong to the same pixel as *voxel column* or simply *column*. For each voxel we store 1 bit (empty or occupied).

Splatting (Fig. 2a) takes a list of particles as input (in any order) and renders one sphere for each particle, marking the overlapped voxels in the columns of overlapped pixels. When one tile has been voxelized, we extract the entry and exit surface depths of an imaginary eye ray through each pixel, by iterating over all columns in parallel. We store the resulting list of surface depths for every pixel (Fig. 2b). Afterwards, the voxel subgrid is discarded and we continue voxelization with the next tile. Note that this scheme would trivially allow for work distribution across multiple GPUs.

If the number of layers exceeds the allocated memory, we continuously update the last exit depth and additionally keep track of the amount of empty space between encountered layers. Thus, we obtain surfaces for the first k layers while still allowing for correct volumetric absorption.

Additionally, we can take advantage of simulation constraints such as in the simulation of incompressible fluids: Since particle simulations are typically grid-based, a maximum number of particles per grid cell allows us to quickly determine *full* inner cells whose particles do not add information. We leave these out of the input data, drastically reducing the number of particles to be voxelized. As many simulations keep their particles sorted, this is easily achieved in a fast reduction and compaction step before voxelization.

3.2. Surface Extraction

In principle, whenever two successive bits in a voxel column differ, a boundary surface of the voxelized volume has been found. However, there is a trade-off between minimizing particle size and minimizing gaps in the volume. To allow for smaller particles, we always keep track of the last encountered set bit in each iterated column, and only insert exit surface depths when a minimum (world-space) distance between the last and the next set bit has been exceeded.

3.3. Connecting Depth Lists to Surfaces

After compression, we obtain a list of depth values that mark the entry and exit points of the fluid along a voxel column. We will subsequently refer to the depth intervals formed by pairs of successive entry and exit depths in each column as *spans*. The number of spans can be different for neighboring columns and at first, it is unclear which depth values in neighboring columns belong to the same surfaces. To extract connected surfaces, we start by identifying spans in neighboring columns that overlap in depth, and which are therefore connected to the same part of the volume.

Figure 3 illustrates the possible configurations that need to be considered. We connect the frontmost entry depths and the backmost exit depths of *overlapping* neighbor spans to form one surface (a). *Connect* means to store pointers for the

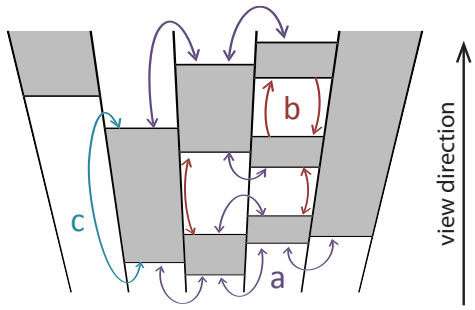

Figure 3: *Gray bars represent contiguous set voxels within a voxel column. In order to extract surfaces, we search for overlapping spans and store links connecting (a) entry and exit locations of neighboring spans belonging together, and exit and entry points of empty space (b) (e.g. bubbles in a fluid) and (c) at boundary spans.*

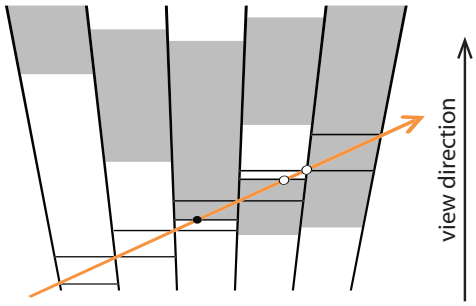

Figure 4: *Ray casting is performed by marching along the ray's projection onto the image plane. For every pixel we determine the intervals (gray) from its voxel column whose boundaries potentially intersect the ray (orange).*

two connected entry or exit points to the respective other. Inner exit depths are connected to the subsequent entry depth *in their own column* and inner entry depths to the preceding exit depth (b), which effectively accounts for bubbles inside the volume. When there is no overlapping neighbor span, entry and exit depths are linked to each other (c). See Listing 1 for the exact procedure of linking of a given ('current') span.

We save the resulting surface connection links for the four direct neighbors of each pixel, packed into one machine word and stored in parallel to the surface depth values. We use two special link values to indicate links to subsequent or preceding entries in the same column.

3.4. Filtering and Normal Generation

Given the surface links, we can now easily traverse the surface from any given point. This allows for trivial application of smoothing filters such as blurring kernels or iterative curvature flow [vdLGS09,BSW10]. Due to the potentially complex shape of the surface, iterative filters are a good choice to ensure relatively even filter spread in all directions.

After filtering, we generate normals using the Sobel operator to estimate derivatives in two directions. Just like the surface links, we store the normals in parallel to the surface depth array to allow easy access during rendering. In fact, we reuse surface link storage and compute surface connectivity on the fly, using the same logic as during surface connection.

3.5. Rendering and Ray Casting

Rendering the frontmost surface is now as easy as reading the first depth and normal entry of every voxel column (i.e. pixel). For translucency effects, it is sufficient to iterate over all depth and normal entries and to apply a scattering and absorption model using ray segment lengths computed from the depth differences of adjacent entry and exit depths.

To implement more complex illumination effects such as refractions, ray casting can be started after reading the first surface entry, perturbing the ray direction accordingly. Ray casting is performed in screen space: starting at the pixel that contains the ray origin, we march pixel by pixel along the projected ray direction. For each pixel we identify all entry and exit points that lie in-between the depths where our ray enters and leaves the corresponding voxel column (illustrated in Figure 4). When entering the next pixel, the same depth layer is usually a good guess where to start looking for the next hit surface. Note that hits may also occur on voxel column boundaries in-between stored surface values (e.g. the third intersection in Figure 4). However, these cases are easily detected by checking whether the ray changes its state between inside and outside when entering a new column (LSB of layer index). Listing 2 provides pseudocode.

```
front_nb = first_neighbor_span(where .exit >= current_span.entry)
if (front_nb.entry <= current_span.exit) {
  //if neighbor not connected to previous span, link entries
  if (front_nb.entry > previous_span.exit)
       link(current_span.entry, front_nb.entry)
  //else: same neighbor as last time, forming a bubble wall
  else link(current_span.entry, previous_span.exit)

  back_nb = last_neighbor_span(where .entry <= current_span.exit)
  //if neighbor not connected to next span, link exits
  if (next_span.entry > back_nb.exit)
       link(current_span.exit, back_nb.exit)
  //else: neighbor recurs next time, forming a bubble wall
  else link(current_span.exit, next_span.entry)
}
```

Listing 1: *Pseudocode for connected surface construction.*

```
while (column = next_intersected(ray, column)) {
  [entry, exit] = intersection_depths(ray, column)
  layer = first_layer_after(entry, z_sgn, start_hint = layer)
  layer_found = (z_sgn * layer <= exit * z_sgn)
  entering = ((layer & 1) != (z_sgn > 0)) == layer_found
  do {
    if (entering != was_inside) handle_intersection(...)
    if (layer_found) layer += z_sgn
    entering = !entering
  } while(z_sgn * layer <= exit * z_sgn)
} // z_sgn = -1 if ray marching against view dir
```

Listing 2: *Pseudocode for ray marching through columns.*

As we want to trace multiple refractions, we abort marching whenever a surface is intersected. We compute the ray's contribution and then restart ray casting with a new perturbed ray. We can terminate ray marching when we have reached the voxel grid boundary.

3.6. Ambient Occlusion

As the data structure stores multiple layers, it can also be used to implement deep ambient occlusion. In our results, we use a simple screen-space ambient occlusion approach that computes the amount of occlusion for a given point by sampling the occlusion from neighboring surface points inside a given radius [Mit07]. In contrast to traditional screen-space ambient occlusion, we do not suffer from missing information behind the front-most surfaces, since we have full knowledge of many layers of entry and exit surfaces.

4. Implementation

We have implemented our algorithm using CUDA 7.0 and OpenGL. Using volume textures to store all our data allows us to take advantage of GPU-internal memory layouts optimized for cache-locality of spatially adjacent data.

4.1. Splatting and Voxelization

To render particles into the voxel grid, we render screen-aligned bounding quads using an OpenGL geometry shader. This shader projects particle spheres onto the screen. For each generated fragment, the particle sphere is intersected with the corresponding camera ray. All overlapped voxels in the corresponding column are then marked, the atomic bitwise operations offered by modern GPUs make it easy to set the bits of overlapped voxels concurrently.

On an NVIDIA GTX Titan, speedups can be achieved by rendering smaller particles (radius < 7 pixels) using a simple splatting CUDA kernel for all particles in parallel. This kernel also projects particles onto the screen and then loops over all pixels in an axis-aligned bounding quad sequentially. (While CUDA does not yet expose atomics on textures directly, they are accessible via the 'sured' instruction using inline PTX code). Like most simulation data, our particles were partially sorted by the simulation. Thus, splatting happens in groups of approximately equal distance and size, explaining SIMT efficiency. However, on a GTX 970, pure OpenGL-based rendering is always faster.

4.2. Voxelization Layer Distribution

To ensure a continuous appearance in the final renders generated from the extracted boundary surface depths, we need to enforce a minimum angle resolution for the resulting surface normals. For this, we need more depth precision near the camera than farther away during voxelization. The minimum angle step α between a reconstructed surface that is coplanar to the viewer and an adjacent surface bent towards the viewer is determined by the minimum depth step that can be taken towards the viewer. In-between columns of width c_w, that minimum depth step is one cell depth c_d:

$$\alpha = \tan \frac{c_d}{c_w}$$

For perspective grids, the vertical cell width c_w can be computed as $c_w = \frac{2\tan(\theta_y/2)}{R_y} d$ [OBA12], where d denotes the depth of the cell, the vertical grid resolution is R_y and the vertical field of view is θ_y. Setting $c_d = c_w \arctan\alpha =: K_d d$, we can compute the depth d_i of layer i and vice versa as:

$$d_i = d_{i-1} + c_d^{(i-1)} = d_{i-1} + K_d d_{i-1} = (1+K_d)^i d_0$$
$$i = \left\lfloor \frac{\log d_i - \log d_0}{\log(1+K_d)} \right\rfloor$$

As we can see, the minimum angle α and the near plane at d_0 fully determine a logarithmic depth partitioning that keeps cell extent ratios and thus α constant throughout the perspective grid. Note that all but one logarithm can be precomputed. Most GPUs offer fast binary logarithm intrinsics.

4.3. Surface Extraction with the Simulation Grid

To accelerate the iteration over voxel columns, we iterate over the packed machine words rather than single bits, using bit counting intrinsics to quickly determine the next relevant bits to be considered.

SPH simulations typically use regular grids to accelerate the neighbor search [IOS*14]. Furthermore, it is possible to efficiently determine during the simulation whether a simulation grid cell is empty, full (e.g. filled entirely with particles of an incompressible fluid) or partly filled with particles. This information can be used in our method to skip over empty and fully occupied regions during surface extraction.

To implement cell skipping, we employ standard ray marching through the simulation grid. During extraction, we march along the view ray corresponding to each voxel column. We keep marching through full or empty simulation grid cells until we encounter a succession of partially filled cells, after which we interrupt ray marching to iterate over the corresponding voxels. When the iteration reaches the end of the current set of connected partially filled cells, we resume ray marching in the simulation grid. Note that in order not to cut off detailed boundary surfaces potentially formed by particles overlapping cell boundaries, we may not skip cells that have partially filled neighbors, i.e., only *inner* full and empty cells may be skipped.

4.4. Surface Extraction with Gap Bridging

To allow for smaller particle radii, we only assume two distinct surfaces (and insert separating entry and exit points) when we detect contiguous empty voxels which span a greater distance in world space than a threshold g. As the

mapping of depth to voxels in a column is logarithmic, we want to avoid computing the distances repetitively during surface extraction. Instead we compute a lower bound of the number of voxels that corresponds to g for each part of a voxel column in a boundary region. Only when this lower bound is exceeded by two successive set voxels we need to compute their actual world-space distance to decide whether to insert separating surfaces or to connect voxels.

Note that this *gap bridging* allows for further optimization during splatting: instead of setting bits for all voxels inside the particle spheres, it is now sufficient to mark only the voxels that overlap the surface of each particle sphere.

4.5. Data Structure and Compression

As outlined in Section 3, we need to store multiple layers of depth, surface links, and normals for each pixel. We store depth in 32-bit floating-point format. The links to the four direct neighbors of each depth value are stored in one byte each, adding up to another 32-bit word. We compress normals to another 32-bit word using an octahedron normal encoding [MSS*10] with two 16-bit normalized integers.

4.6. Surface Smoothing

To obtain a smooth surface from voxelized particle spheres, we iteratively apply a 3×3 smoothing kernel to the reciprocal of the extracted depth values. Like Bagar et al. [BSW10], we let the number of iterations depend on the distance of the processed surface point to the camera. As a consequence, the world-space radius of our filter is approximately constant. Since the surface connection step already identified the right connected surface points for each pixel's neighborhood, we can simply follow the stored surface links.

5. Results

We tested our algorithm using an SPH simulation data set computed using the method described by Ihmsen et al. [IABT11]. The data set features both relatively calm and even water surfaces as well as highly turbulent water flow with a lot of spray. The simulation was done with 20 million particles, of which we extracted up to 7 million particles in boundary cells for voxelization. To obtain boundary cells, we excluded empty and full *inner* cells, i.e. cells that do not have partially filled neighbors. The number of particles in full cells is determined from simulation constraints.

Tile Size	Splat	Compress	# Tiles
128^2	546 ms	47 ms	135
256^2	182 ms	32 ms	40
512^2	80 ms	28 ms	12
1024×512	58 ms	28 ms	6
1024^2	76 ms	28 ms	4

Table 1: *Voxelization splatting and compression performance with different tile sizes on an NVIDIA GTX 970.*

We ran tests on an NVIDIA GeForce GTX 970 and a GTX Titan. We provide results for voxelization at 1920×1080 and at 3840×2160, and for two types of visualization: simple fully translucent surface rendering without refraction and more elaborate translucent rendering with refraction and total internal reflection ray casting (two full paths per pixel).

5.1. Angle Resolution

With an angle resolution of 14°, discontinuities in the normals become barely noticeable even without smoothing (compare Figure 6). This corresponds to $\sim$16.000 layers between near and far plane; we allocate 16384 layers.

5.2. Memory Requirements

The memory taken up by the input data set is dominated by the particle data, ranging from 40 to 110 megabytes comprised of 12-byte floating-point triplets that define the world-space position of each particle.

The bit-packed perspective voxel grid used for a screen tile of the voxelization has a resolution of $1024 \times 512 \times 16384/32$, temporarily taking up 1024 megabytes. Note that the temporary storage can be easily adapted by adjusting the tile size. The surface information extracted from the high-resolution voxelization is stored at full screen resolution in up to 7 spans, requiring 126 megabytes for depths and normals; and totalling 252 megabytes at 1920×1080 for this setting. Surface links are replaced by normals.

5.3. Performance

In Table 2 we provide detailed timings for the steps of our algorithm that were measured in six different viewpoints and with two types of rendering, as shown in Figures 1, 5(f). Translucency-only rendering simply iterates over all surfaces extracted from one voxel column and is therefore much more efficient than casting perturbed refraction rays that potentially traverse many columns.

Apart from complex ray casting-based rendering, most of the time is taken up by surface extraction, comprised of splatting and RLE compression. This is to be expected, since up to 7 million particles need to be voxelized during splatting and many layers of depth need to be scanned during compression. Afterwards, surface connection, filtering and normal generation are all rather simple operations.

The iterative filtering is dependent on the viewer distance, as the projected filter radius and therefore the number of necessary iterations (up to 20) both depend on this distance.

Table 1 shows that it is essential to choose a large enough tile size during splatting and compression. With smaller tiles, the overhead of repeated splatting grows, while too few threads get launched during compression, leaving the GPU under-occupied. On the GTX 970, a tile size of 1024×512 is optimal, but within 20 ms of 512^2 timings.

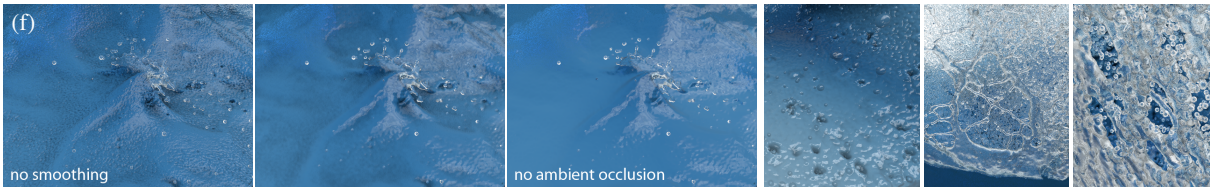

Figure 5: *Close-up renderings of an SPH simulation computed with the method by Ihmsen et al. [IABT11] (20 Mio particles).*

	# Particles	Splat	RLE	Connect	Filter	Normals	Translucent (∑)	Refract (∑)
(a)	6.7 Mio	70	52	1.2	14	8	10 (158)	359 (507)
(b)	7.1 Mio	76	49	2.0	16	11	15 (174)	800 (640)
(c)	5.9 Mio	70	44	1.9	16	11	13 (160)	552 (700)
(d)	4.7 Mio	60	28	1.7	15	8	12 (131)	710 (832)
(e)	3.4 Mio	56	57	0.7	13	6	10 (146)	452 (589)
(f)	3.4 Mio	62	38	0.9	21	7	58 (190)	2195 (2326)
(a)	NV Titan	78	18	1.5	42	13	10 (165)	753 (907)
(b)	NV Titan	82	23	2.6	48	17	14 (187)	1303 (1477)
(a)	at 4K	232	138	4.5	207	31	66 (686)	2316 (2935)
(b)	at 4K	239	128	7.8	377	43	105 (910)	3948 (4752)

Table 2: *Timings (in ms) of voxelization and rendering for the views in Figs. 1, 5(f), measured on a GeForce GTX 970 (top) and a GTX Titan (middle). The last two columns denote rendering with translucency only versus refraction and reflection tracing (compare Fig. 7). The resolution is* $1920 \times 1080 \times 16384$*, except 4K rows.*

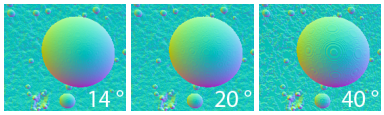

Figure 6: *The impact of angle resolution with no smoothing applied.*

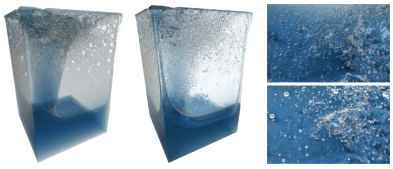

Figure 7: *Translucency vs. refraction.*

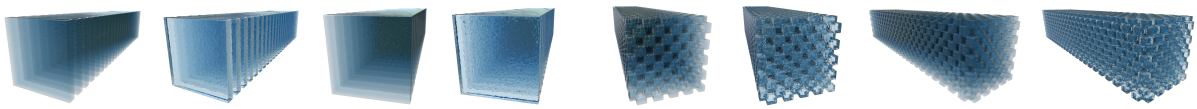

Figure 8: *Synthetic data sets featuring more layers than can be stored: 24 Layers (left) and a 3D checkerboard pattern (right).*

In all cases, even with unaccelerated ray casting-based refraction rendering, our total render time stays well below the time it currently takes to simulate one step in simulations of comparable particle count, as presented in [IABT11]. Thus, rendering is no longer a bottleneck.

5.4. Synthetic Data Sets

Figure 8 shows renderings of two synthetic data sets with 10 million particles each. The layer data set features 24 layers of particles. While we only store 7 spans for each pixel, the translucency-only renderings still fully reproduce all layers, since we always keep track of the actual number of layers as well as the amount of empty space inside the extended last span. In renderings with refraction ray casting, on the other hand, refraction usually makes it impossible to clearly discern even the first four layers. In the refracting 3D checkerboard data set, even less layers are discernible.

5.5. Comparison to Related Work

Using a factor of 3 [RTW13] to compare [FAW10] to our Titan experiments, we compare viewport (b) with its 7.1 Mio particles and a total frame time of 187 ms for translucency rendering to their SNIaEjecta dataset with its 8.7 Mio particles and an upscaled, compensated frame time of 7543 ms / 3 = 2514 ms, suggesting our method is 10 times faster. Moreover, their method requires a preprocessing phase of unspecified time, whereas our method directly renders simulation data and is therefore suitable for ad-hoc visualization.

Compared to [RTW13], we are about two times faster. However, the focus of their work is out-of-core rendering of billions of particles, and requires 35 hours of preprocessing, making it a rather asymmetric comparison.

6. Discussion, Limitations and Future Work

The view-adaptivity of our voxelization has pros and cons. The advantages of this approach are an exact resolution match for the actual viewport and thus an optimal utilization of memory. Conversely, casting rays whose direction deviates from the camera rays is more expensive than with non-view-dependent data structures. Note that for shadow mapping or other approaches where a further from-point view is required, we can still compute a second voxelization.

Currently our approach is limited to homogeneous media as we only detect entry and exit points. An extension to heterogeneous media with particles of different types or with scalar attributes would be interesting future work. Without noteworthy modifications, our method could still detect boundaries between different regions without large changes, but it would also need to store scalar values either explicitly or compressed, e.g. akin to Salvi et al. [SVLL10].

Lastly, ray casting can be accelerated using a hierarchical representation of the spans akin to mipmapping. For example, the boundary surfaces of four neighboring columns can be conservatively bounded in one column of a coarser grid.

7. Conclusion

In this paper we presented a view-adaptive voxelization for homogeneous particle-based (in particular SPH) simulation data. Our method efficiently handles millions of particles and computes voxelizations – compressed during construction – at resolutions which would exceed available GPU memory otherwise. The view-dependent voxelization represents a compromise between truly volumetric representation of the particle data (with all the benefits such as ray casting, rendering with translucency etc.) and the efficiency of image space techniques in terms of memory and performance. We believe that our method serves as a basis for rendering and visualization of diverse particle-based simulations.

Acknowledgements

We thank the Computer Graphics group at the University of Freiburg for providing us with the dambreak dataset.

References

[AAIT12] AKINCI G., AKINCI N., IHMSEN M., TESCHNER M.: An efficient surface reconstruction pipeline for particle-based fluids. In *Proc. of VRIPHYS* (2012), pp. 61–68. 2

[AAOT13] AKINCI G., AKINCI N., OSWALD E., TESCHNER M.: Adaptive surface reconstruction for SPH using 3-level uniform grids. In *Proc. of Winter School of Computer Graphics* (2013), pp. 195–204. 2

[AIAT12] AKINCI G., IHMSEN M., AKINCI N., TESCHNER M.: Parallel surface reconstruction for particle-based fluids. *Computer Graphics Forum 31*, 6 (2012), 1797–1809. 1, 2

[APKG07] ADAMS B., PAULY M., KEISER R., GUIBAS L. J.: Adaptively sampled particle fluids. *ACM Transactions on Graphics 26*, 3 (2007). 1, 2

[Bli82] BLINN J. F.: A generalization of algebraic surface drawing. *ACM Transactions on Graphics 1*, 3 (1982), 235–256. 2

[BSW10] BAGAR F., SCHERZER D., WIMMER M.: A layered particle-based fluid model for real-time rendering of water. *Computer Graphics Forum (Proc. of EGSR) 29*, 4 (2010), 1383–1389. 2, 4, 6

[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. 1, 2, 7

[GIK*07] GRIBBLE C., IZE T., KENSLER A., WALD I., PARKER S.: A coherent grid traversal approach to visualizing particle-based simulation data. *IEEE Transactions on Visualization and Computer Graphics 13*, 4 (2007), 758–768. 2

[GRDE10] GROTTEL S., REINA G., DACHSBACHER C., ERTL T.: Coherent culling and shading for large molecular dynamics visualization. *Computer Graphics Forum (Proc. of EuroVis) 29*, 3 (2010), 953–962. 2

[Gre10] GREEN S.: Screen space fluid rendering for games. http://developer.download.nvidia.com/presentations/2010/gdc/Direct3D_Effects.pdf, 2010. 2

[GSSP10] GOSWAMI P., SCHLEGEL P., SOLENTHALER B., PAJAROLA R.: Interactive SPH simulation and rendering on the GPU. In *Proc. of ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2010), pp. 55–64. 2

[HH09] HOETZLEIN R., HÖLLERER T.: Interactive water streams with sphere scan conversion. In *Proc. of ACM SIGGRAPH I3D* (2009), pp. 107–114. 2

[IABT11] IHMSEN M., AKINCI N., BECKER M., TESCHNER M.: A parallel SPH implementation on multi-core CPUs. *Computer Graphics Forum 30*, 1 (2011), 99–112. 1, 6, 7

[IOS*14] IHMSEN M., ORTHMANN J., SOLENTHALER B., KOLB A., TESCHNER M.: SPH fluids in computer graphics. In *Eurographics State of the Art Reports* (2014), pp. 21–42. 2, 5

[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. 1, 2

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics (Proc. SIGGRAPH) 21*, 4 (1987), 163–169. 1, 2

[Mit07] MITTRING M.: Finding next gen: Cryengine 2. In *ACM SIGGRAPH Courses* (2007), pp. 97–121. 5

[MSS*10] MEYER Q., SÜSSMUTH J., SUSSNER G., STAMMINGER M., GREINER G.: On floating-point normal vectors. *Computer Graphics Forum (Proc. of EGSR) 29*, 4 (2010), 1405–1409. 6

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

[OBA12] OLSSON O., BILLETER M., ASSARSSON U.: Clustered deferred and forward shading. In *Proc. of High Performance Graphics* (2012), pp. 87–96. 5

[PTB*03] PREMOZE S., TASDIZEN T., BIGLER J., LEFOHN A., WHITAKER R. T.: Particle-based simulation of fluids. *Computer Graphics Forum 22*, 3 (2003), 401–410. 2

[RCBW12] REICHL F., CHAJDAS M. G., BÜRGER K., WESTERMANN R.: Hybrid Sample-based Surface Rendering. In *Proc. of VMV* (2012), pp. 47–54. 2

[RTW13] REICHL F., TREIB M., WESTERMANN R.: Visualization of big SPH simulations via compressed octree grids. In *Proc. of IEEE Conference on Big Data* (2013), pp. 71–78. 2, 7

[SI12] SZÉCSI L., ILLÉS D.: Real-time metaball ray casting with fragment lists. In *Eurographics Short Papers* (2012), pp. 93–96. 2

[SSP07] SOLENTHALER B., SCHLÄFLI J., PAJAROLA R.: A unified particle model for fluid-solid interactions. *Computer Animation and Virtual Worlds 18*, 1 (2007), 69–82. 1, 2

[SVLL10] SALVI M., VIDIMČE K., LAURITZEN A., LEFOHN A.: Adaptive volumetric shadow maps. *Computer Graphics Forum (Proc. of EGSR) 29*, 4 (2010), 1289–1296. 7

[vdLGS09] VAN DER LAAN W. J., GREEN S., SAINZ M.: Screen space fluid rendering with curvature flow. In *Proc. of ACM SIGGRAPH I3D* (2009), pp. 91–98. 1, 2, 4

[YT13] YU J., TURK G.: Reconstructing surfaces of particle-based fluids using anisotropic kernels. *ACM Transactions on Graphics 32*, 1 (2013), 5:1–5:12. 1, 2

[YWTY12] YU J., WOJTAN C., TURK G., YAP C.: Explicit mesh surfaces for particle based fluids. *Computer Graphics Forum (Proc. of Eurographics) 31*, 2 (2012), 815–824. 2

[ZB05] ZHU Y., BRIDSON R.: Animating sand as a fluid. *ACM Transactions on Graphics 24*, 3 (2005), 965–972. 1, 2