DOI: 10.1111/cgf.14090

COMPUTER GRAPHICS *forum*
Volume 00 (2020), number 00 pp. 1–15

Multi-Level Memory Structures for Simulating and Rendering Smoothed Particle Hydrodynamics

R. Winchenbach and A. Kolb

University of Siegen, Siegen, Germany
andreas.kolb@uni-siegen.de

Abstract

In this paper, we present a novel hash map-based sparse data structure for Smoothed Particle Hydrodynamics, which allows for efficient neighbourhood queries in spatially adaptive simulations as well as direct ray tracing of fluid surfaces. Neighbourhood queries for adaptive simulations are improved by using multiple independent data structures utilizing the same underlying self-similar particle ordering, to significantly reduce non-neighbourhood particle accesses. Direct ray tracing is performed using an auxiliary data structure, with constant memory consumption, which allows for efficient traversal of the hash map-based data structure as well as efficient intersection tests. Overall, our proposed method significantly improves the performance of spatially adaptive fluid simulations and allows for direct ray tracing of the fluid surface with little memory overhead.

Keywords: fluid modelling, animation, surface reconstruction, modelling, ray tracing, rendering

ACM CCS: • Computing methodologies → Massively parallel and high-performance simulations; Ray tracing; Physical simulation

1. Introduction

Highly detailed and realistic fluid simulations have become an essential part of modern computer graphics, where Smoothed Particle Hydrodynamics (SPH) [GM77] provides a good balance of visual quality and computational cost [UHT17]. Recent advances enable highly adaptive incompressible fluid simulations [WHK17], which dedicate computational resources where they are most beneficial to the desired outcome, that is, at the fluid surface. However, adaptive simulations with adaptivity ratios of 1000:1 and higher suffer from significant performance drops due to limitations in existing underlying data structures. CPU-based SPH simulations commonly use compact hash maps [IABT11], which are difficult to apply on GPUs. GPU-based SPH simulations commonly use dense cell structures [Gre10, GSSP10] or linked list based structures [DCVB*13, WRR18], which suffer from high memory usage and less than ideal particle orderings. Additionally, rendering the resulting fluid surfaces often involves either expensive explicit surface extraction methods [AAOT13, WLS*17] using marching cubes [LC87], which cannot readily be done on the fly, or screen space-based approaches [vdLGS09, XZY17], which result in lower quality visual results and cannot readily handle refraction effects. Furthermore, many rendering methods have varying memory requirements, for example depending on particle resolution and not particle count, making them impractical to use on the fly on a GPU as this requires an overly conservative maximum number of particles to not run out of memory during a simulation.

In this paper, we present a hash map-based data structure, which is specifically designed to handle the requirements of highly adaptive SPH methods, on GPUs and CPUs, and is readily extended to enable direct ray tracing of the fluid surface. Our proposed data structure works by utilizing a hash map to efficiently access a compact cell list, which refers to particles sorted by a self-similar ordering. We extend this method by efficiently creating multiple distinct data structures, based on different cell sizes, by utilizing the self-similarity. Our method allows us to significantly reduce the number of non-neighbour particle accesses by providing an appropriate data structure for different particle resolutions. Furthermore, we add an auxiliary data structure to facilitate traversal of our data structure during rendering, which also enables efficient ray-fluid intersection tests. Our proposed method significantly improves the practical

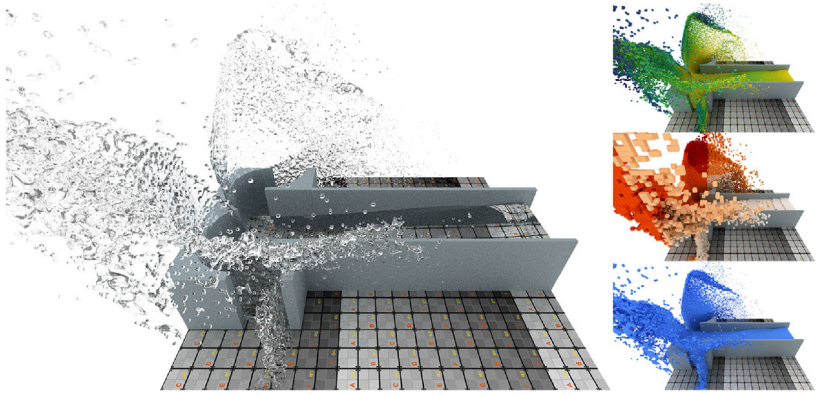

Figure 1: *Using our rendering approach, we can render an SPH fluid simulation with only a small, constant, memory overhead. Pictured here is an inlet stream colliding with an obstacle using an anisotropic surface rendering. The images to the right show the underlying particles, colour coded for their velocity (top), the underlying cells, colour coded for their Morton code (middle) and an opaque fluid surface (bottom).*

applicability of adaptive simulations, and substantially reduces the data structure overhead, as well as enabling direct ray tracing of fluid simulations, with constant memory overhead. Finally, our method requires a constant amount of memory, regardless of simulation domain size, particle resolution, or particle distribution, allowing for unbounded simulation domains.

2. Related Work

SPH has been an active field of research since its introduction by Gingold and Monaghan [GM77]. Although initially only stiff equations of state were used to simulate weakly compressible fluids [MCG03, BT07], recent advances allow both divergence-free and incompressible fluids [BK15, GPB*19], allowing for highly detailed and realistic fluid simulations. For a general overview of SPH methods, we refer the reader to [KBST19].

Spatially adaptive SPH methods using splitting and merging were initially introduced by Desbrun and Cani [DC99], however, directly changing particle resolutions causes instabilities. To reduce these instabilities Adams *et al*. [APKG07] adjusted particle positions after splitting, Keiser *et al*. [KAD*06] used virtual link particles of neighbouring resolutions, Orthmann and Kolb [OK12] used temporal blending, Horvath and Solenthaler [HS13] used multiple simultaneous simulations and Winchenbach *et al*. [WHK17] used an additional process of mass redistribution. However, even though recent work enables adaptive ratios in excess of 10 000:1, these methods are constrained by significant performance limitations due to existing data structures [WHK17].

To render fluid simulations there are three commonly used approaches: explicit surface extraction, ray casting or splatting. Explicit surface extraction methods are commonly based on marching cubes [LC87] or metaballs [Bli82], which have been adapted over time for GPUs [WLS*17, SI12], for varying spatial surface resolutions [AAOT13] or for direct rendering [KSN08], however, these methods often have highly varying memory requirements making them difficult to use on the fly. Ray casting methods are commonly found in volume rendering approaches [DCH88] where the fluid volume is commonly resampled into a 3D uniform grid [NJB07] or a perspective aligned grid [FAW10] and then rendered using standard volume ray casting [KW03]. However, these methods commonly require resampling of the full simulation data into memory intensive grids. Splatting methods [ZPVBG01] used in real-time applications, due to their computational simplicity, render particles as simple geometry in screen space and then smooth the resulting depth image [MSD07, vdLGS09, XZY17], however, these methods cannot handle refractions and reflections properly. Outside of fluid simulations, various other rendering approaches exist, that is, for point clouds [BTS*17], however, they are usually not directly applicable.

For SPH-based simulations, Ihmsen *et al*. [IOS*14] give a good overview of existing data structure methods for CPUs, and identify a hash map-based method [IABT11] as the most efficient data structure. This approach is, however, not directly applicable to GPUs due to the way in which the hash map is constructed. For GPU-based simulations, Green [Gre10] introduced a method utilizing a fixed domain with linearly indexed cell lists. A similar approach was used by Dominguez *et al*. [DCG11], optimized for multiple GPUs. Goswami *et al*. [GSSP10] used Morton codes, however, their approach introduces a complex scheme to balance workloads on GPUs, making it difficult to implement and utilize. In order to limit memory usage on GPUs, Winchenbach *et al*. [WHK16] introduced an iterative process to constrain the size of so-called Verlet-lists, which are used to store references to neighbouring particles. However, all of these methods suffer from scaling and performance problems for adaptive simulations due to excessive non-neighbour particle accesses.

Many generic data structures have been developed for computer animation, that is, perfect hash maps to store sparse voxel data [LH06, GLHL11], which are not easily scalable to multiple resolutions, or approximate nearest neighbour methods from machine learning [AI08], which are only approximate and designed for high-dimensional data. To improve the rendering speed of explicit surface methods, many methods have been developed, that is, bounding volume hierarchies (BVH) [GPSS07, LGS*09] and kd-trees [FS05]; however, these structures are varying in their memory requirements making them difficult to apply on the fly. Furthermore, various

CPU-based approaches exist, for example, OpenVDB [MLJ*13], but they often require significant changes to be realized on GPU-based systems. OpenVDB was realized for GPUs as GVDB, where recently, Wu *et al.* [WTYH18] introduced a GVDB-based data structure for FLIP-based simulations that significantly improves performance, but which is not directly applicable to SPH, as FLIP imposes significantly different requirements on the data structure, which is an integral part of the simulation itself. Overall, none of the existing data structures can be applied directly both for rendering and simulation without significant overhead.

3. Foundations of Isotropic and Anisotropic SPH

In standard SPH, a quantity A at a position $\mathbf{x}$ is interpolated using a weighted average of spatially close particles j as [KBST19]

$$\langle A(\mathbf{x})\rangle = \sum_j A_j \frac{m_j}{\rho_j} W(\mathbf{x}-\mathbf{x}_j, h), \tag{1}$$

where m denotes the mass and ρ the density of a particle and W is a kernel function. Evaluating Equation (1) at the position of a particle i then results in

$$A_i = \sum_j A_j \frac{m_j}{\rho_j} W(\mathbf{x}_{ij}, h) = \sum_j A_j \frac{m_j}{\rho_j} W_{ij}, \tag{2}$$

where $\mathbf{x}_{ij} = \boldsymbol{x}_i - \boldsymbol{x}_j$. The support radius h in standard SPH formulations describes an isotropic sphere with radius h, which allows one to define the kernel function W as [DA12]

$$W_{ij} = W(\mathbf{x}_{ij}, h) = C_d \frac{1}{h^d} \hat{W}(q), \tag{3}$$

where $q = \|\mathbf{x}_{ij}\|/h$, d being the dimensionality of the simulation and C_d a normalization factor. For the interaction of two particles with different support radii, that is, due to spatil adaptivity, h can be determined as either h_i, resulting in a gather formulation, h_j, resulting in a scatter formulation, or $\frac{h_i+h_j}{2}$, resulting in a symmetric formulation. When not evaluating an SPH quantity at a particle, but at an arbitrary position $\mathbf{x}$, symmetric or gather formulations cannot be directly used and, thus, the scatter formulation is commonly utilized. Here $\hat{W}$ denotes the actual kernel function, where for an SPH simulation the cubic spline kernel [Mon05]

$$\hat{W}(q) = \left[(1-q)^3\right]_+ + \left[4(0.5-q)^3\right]_+ \tag{4}$$

is a common choice, with $C_3 = \frac{14}{\pi}$ and $[\cdot]_+$ being $\max(\cdot, 0)$. For rendering a more simple kernel is commonly chosen, that is, [YT13]

$$\hat{W}(q) = \left[(1-q^2)^3\right]_+, \tag{5}$$

with $C_3 = \frac{315}{64\pi}$. An isotropic support radius results in an equal extent of the support domain in all directions, that is, $h = h^x = h^y = h^z$, which leads to issues on the surface of the fluid [YT13]. Instead of this isotropic formulation, one can determine an anisotropic formulation of SPH [OVSM98]

$$W_{ij} = C_d \frac{1}{|H|} \hat{W}(||H \cdot \boldsymbol{x}_{ij}||), \tag{6}$$

where H is the anisotropy matrix and |H| being the determinant of H. Here the isotropic variant of SPH is equivalent to $H = \frac{1}{h}\mathbb{1}$. This anisotropic formulation, however, also means that the support domain of a particle is ellipsoidal, instead of spherical, meaning the extent of the domain in all directions is not equal, that is, $h^x \neq h^y \neq h^z$, resulting in additional challenges for neighbourhood queries as this requires non-cubic cells in a data structure. Determining a gather and scatter formulation for anisotropic SPH can be done directly using the anisotropic matrix of either particle; however, to yield a symmetric formulations, we use [HK89]

$$\hat{W}(||H \cdot \boldsymbol{x}_{ij}||) = \frac{1}{2}\left[\hat{W}(||H_i \cdot \boldsymbol{x}_{ij}||) + \hat{W}(||H_j \cdot \boldsymbol{x}_{ij}||)\right]. \tag{7}$$

We utilize a standard isotropic formulation for the simulation, and the anisotropic formulation only for a fluid surface rendering using [YT13]. For an isotropic formulation, the support radius h_i for particle i can be determined as $h_i = \sqrt[3]{N_h V_i}$ [DA12, WHK16], with V_i being the rest volume of a particle, that is, $\frac{4}{3}\pi r^3$ for a particle of radius r, and $N_h = 50$ for the cubic spline kernel.

4. Simulation Data Structure

The main purpose of a data structure for SPH is to relate the spatial position of a particle with its location in memory in order to reduce the number of particle accesses from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \cdot c)$, where c is the number of particles accessed for each particle. Therefore, with $c \ll n$ this results in an asympotically linear scaling instead of quadratic scaling. One possible approach is to divide the simulation domain into uniform cells of size h [Gre10, IOS*14], with h being the particle support radius. Note that this notion of a *cell* does not introduce any grid-based methodology into SPH and is solely for data handling. Owing to this, a particle only needs to consider at most 27 cells (a $3 \times 3 \times 3$ sub-grid) for accessing (potentially) neighbouring particles. The sphere described by the support radius of a particle will, on average, contain N_h neighbours [DA12] within a volume of $\frac{4}{3}\pi h^3$, whereas the sub-grid of all potential neighbours has a volume of $27h^3$. This means that the sub-grid will contain, on average, $\frac{81}{4\pi} N_h \approx 6.5 N_h$ particles, that is, 15.5% of all potential neighbours are actual neighbours, that is, the factor m in $\mathcal{O}(n \cdot m)$ becomes 325. For an adaptive ratio of 1000:1, however, only 0.016% of all considered particles are neighbours as a cell of the same size would now contain $\frac{81\,000}{4\pi} N_h$ particles, causing significant performance problems [WHK17]. We are going to introduce our general data structure for non-adaptive simulations in Section 4.1, and the changes required for adaptive simulations in Section 4.2. Section 4.3 introduces our data structure for rendering and Section 4.4 discusses how we can optimize the memory layout of our data structure.

4.1. Single-level data structure

Isotropic SPH methods commonly use cubic cells for data handling [Gre10, IABT11], as the support domain of a particle extents equally along all axes. Anisotropic SPH methods require non-cubic cells for data handling, as the support domain of a particle might extend differently along all axes. As our overall method uses both formulations, we determine the effective support extent $\mathbf{h}$ for each particle, based on the isotropic support radius h_i^i and the anisotropic support extent along all axes $[h_i^x, h_i^y, h_i^z]$, as

$$\mathbf{h}_i = \left[\max\left(h_i^i, h_i^x\right), \max\left(h_i^i, h_i^y\right), \max\left(h_i^i, h_i^z\right)\right]. \tag{8}$$

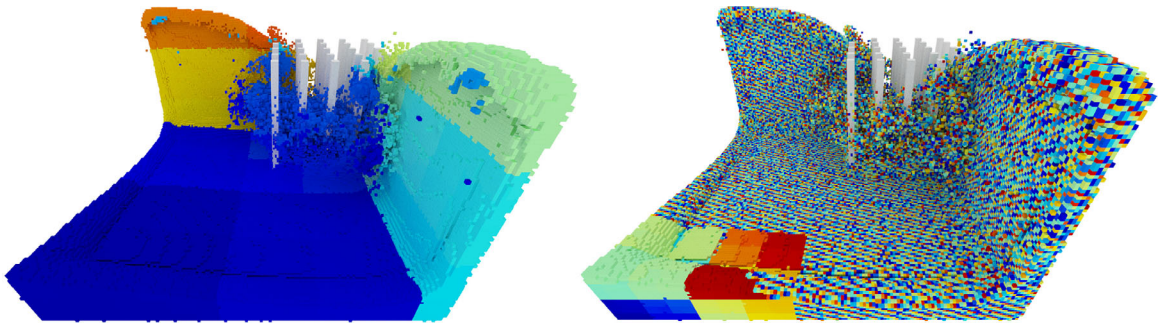

Figure 2: *These two images show the Morton code $\mathcal{Z}$ on the left and the hashed indices $\mathcal{H}$ on the right for every occupied cell, with colour coding indicating indices. The Morton code yields indices that are very similar, for spatially nearby cells, but results in significant amounts of collisions. Using a Hash function (right) yields no relation between spatial position and index, causing a low amount of collisions.*

Alternatively, anisotropic SPH methods can be treated as isotropic, for data handling, using $h_i^i = \max(h_i^i, h_i^x, h_i^y, h_i^z)$, however, this places more particles in each cell, causing more non-neighbour accesses in total. The cell size $\mathbf{C}_{\max}$ is set to the same value as the largest support extent of all particles. This ensures that all neighbours for a particle are contained in a $3 \times 3 \times 3$ sub-grid, which would not be possible for an arbitrary cell size. As such, we calculate $\mathbf{C}_{\max}$ as

$$\mathbf{C}_{\max} = \max\{\mathbf{h}_0, \dots, \mathbf{h}_{n-1}\}. \tag{9}$$

The simulation domain itself is similarly determined as the axis aligned bounding box, from $\boldsymbol{D}_{\min}$ to $\boldsymbol{D}_{\max}$, surrounding the positions of all particles. We determine these bounds by using reduction operations over all particle positions $\boldsymbol{x}_i$

$$\boldsymbol{D}_{\min} = \min\{\boldsymbol{x}_0, \dots, \boldsymbol{x}_{n-1}\}, \boldsymbol{D}_{\max} = \max\{\boldsymbol{x}_0, \dots, \boldsymbol{x}_{n-1}\}. \tag{10}$$

These bounds are used to calculate the size of the simulation domain in cells as

$$\boldsymbol{D} = \left\lceil \frac{\boldsymbol{D}_{\max} - \boldsymbol{D}_{\min}}{\mathbf{C}_{\max}} \right\rceil. \tag{11}$$

When using dense data structures, $\boldsymbol{D}$ needs to be kept constant to avoid reallocating memory when particles move outside the current simulation domain. This, in turn, limits the scene's extend as it needs to be known *a priori*. We then calculate the integer coordinates $\bar{\boldsymbol{x}}$ for any position $\boldsymbol{x}$ based on the lower simulation bound $\boldsymbol{D}_{\min}$ and the cell size $\mathbf{C}_{\max}$ as

$$\bar{\boldsymbol{x}} = \left\lfloor \frac{\boldsymbol{x} - \boldsymbol{D}_{\min}}{\mathbf{C}_{\max}} \right\rfloor. \tag{12}$$

This can be used to determine a linear index $\mathcal{L}$ as

$$\mathcal{L}(\bar{\boldsymbol{x}}) = \bar{\boldsymbol{x}}_x + \boldsymbol{D}_x\big(\bar{\boldsymbol{x}}_y + \boldsymbol{D}_y(\bar{\boldsymbol{x}}_z)\big), \tag{13}$$

where the subscript denotes the dimension. In a dense cell grid, we can utilize $\mathcal{L}(\bar{\boldsymbol{x}})$ to find the memory location of any position in space. Dense data structures, however, are not desirable as their memory consumption scales with both the simulation domain $\boldsymbol{D}$ and the cell size $C_{\max}$, instead of scaling with the particle count $n_{\text{particles}}$. The Morton code [Mor66], also sometimes referred to as the Z-ordering, is an alternative indexing scheme, which describes a self-similar space-filling curve. We can determine $\mathcal{Z}(\bar{\boldsymbol{x}})$ by interleaving the binary representation of an integer coordinates as

$$\bar{\boldsymbol{x}} = \begin{pmatrix} \dots \bar{\boldsymbol{x}}_x^3 \bar{\boldsymbol{x}}_x^2 \bar{\boldsymbol{x}}_x^1 \bar{\boldsymbol{x}}_x^0 \\ \dots \bar{\boldsymbol{x}}_y^3 \bar{\boldsymbol{x}}_y^2 \bar{\boldsymbol{x}}_y^1 \bar{\boldsymbol{x}}_y^0 \\ \dots \bar{\boldsymbol{x}}_z^3 \bar{\boldsymbol{x}}_z^2 \bar{\boldsymbol{x}}_z^1 \bar{\boldsymbol{x}}_z^0 \end{pmatrix} \rightarrow \mathcal{Z}(\bar{\boldsymbol{x}}) = \dots \bar{\boldsymbol{x}}_z^3 \bar{\boldsymbol{x}}_y^3 \bar{\boldsymbol{x}}_x^3 \bar{\boldsymbol{x}}_z^2 \bar{\boldsymbol{x}}_y^2 \bar{\boldsymbol{x}}_x^2 \bar{\boldsymbol{x}}_z^1 \bar{\boldsymbol{x}}_y^1 \bar{\boldsymbol{x}}_x^1 \bar{\boldsymbol{x}}_z^0 \bar{\boldsymbol{x}}_y^0 \bar{\boldsymbol{x}}_x^0,$$

where the superscript denotes a specific bit. Using a 32 bit Morton code results in 10 bit per dimension, meaning each dimension contains a maximum of $\#K = 1024$ cells. A 64 bit Morton code results in 21 bit per dimension, meaning a maximum of $\#K = 2\,097\,152$ cells per dimension. On one hand, it would be possible to create an octree directly from Morton codes [Kar12], as this code represents the ordering of an octree. To find neighbouring particles, in SPH, we only have to consider a small spatial region and, as such, many octree notes, for example, the root node, contain no useful information but still require memory. Furthermore, traversing an octree is computationally relatively expensive and the memory consumption of an octree is not independent of the content. On the other hand, a dense data structure using a Morton code would require excessive amounts of memory, especially as a 64 bit Code would require 2^{60} entries.

We instead propose to create a list of all occupied cells, as the number of occupied cells n_{occupied} is bound by the number of particles $n_{\text{particles}}$, as the worst case would be every particle occupying a different cell. Moreover, the lower bound of occupied cells is based on the number of particles per cell, which is bounded by incompressibility, of $\frac{3}{4\pi} N_h$, see Section 4, resulting in approximately 12 particles per cell for the cubic spline kernel, that is, $n_{\text{occupied}} = \frac{1}{12} n_{\text{particles}}$. However, during a simulation many cells contain less particles, that is, particles flying through the air after an impact, resulting in an average ratio of approximately 1:6 over the course of a simulation.

To generate the list of occupied cells, we first re-sort all particles according to their Morton code $\mathcal{Z}_i = \mathcal{Z}(\bar{\boldsymbol{x}}_i)$. Using this ordering we create a list $\mathbb{C}$ of length $n_{\text{particles}} + 1$, where each element is determined as

$$\mathbb{C}[i] = \begin{cases} i & , \text{if } i = 0 \vee \mathcal{Z}_i \neq \mathcal{Z}_{i-1} \\ -1 & , \text{if } \mathcal{Z}_i = \mathcal{Z}_{i-1} \\ n_{\text{particles}} & , \text{else.} \end{cases} \tag{14}$$

$\mathbb{C}$ now contains either a marker entry (-1 or $n_{\text{particles}}$), or the first index of a particle in an occupied cell, which is similar to the approach by Green [Gre10]. We can now compact $\mathbb{C}$, by removing all invalid entries, which gives us a list $\mathbb{C}^{\text{begin}}_{\text{compact}}$ of length $n_{\text{occupied}}+1$. Using this list of occupied cell beginnings, we can calculate the number of particles in each occupied cell as

$$\mathbb{C}^{\text{length}}_{\text{compact}}[i] = \mathbb{C}^{\text{begin}}_{\text{compact}}[i+1] - \mathbb{C}^{\text{begin}}_{\text{compact}}[i]. \tag{15}$$

This compact list, however, does not yield any way to find the memory location for a particle based on its spatial location. To resolve this, we propose to apply a hash map on top of $\mathbb{C}^{\text{begin}}_{\text{compact}}$ and $\mathbb{C}^{\text{length}}_{\text{compact}}$. Depending on the intended purpose of the data structure, that is, solely simulation or simulation and rendering, different hash functions should be used. The cell information is only accessed once during the simulation, as it is only required for the creation of a neighbour list, but accessed many times during rendering, as rendering requires evaluating SPH estimates at arbitrary positions, which have no neighbour list associated. As such, for the simulation a hash function with few collisions is preferable, but for rendering a hash function with good spatial locality is preferable. If the hash map is only required for the simulation we use the hash function of Teschner *et al.* [THM*03], which is determined using three large prime numbers $p_1 = 73\,856\,093$, $p_2 = 19\,349\,663$, $p_3 = 83\,492\,791$ and the size of the hash table n_{hash} as

$$\mathcal{H}(\bar{\boldsymbol{x}}) = \left(p_1\bar{\boldsymbol{x}}_x + p_2\bar{\boldsymbol{x}}_y + p_3\bar{\boldsymbol{x}}_z\right)\%n_{\text{hash}}. \tag{16}$$

We choose n_{hash} as the smallest prime number larger than the maximum number of particles in a simulation, as this results in a relatively sparse hash map with few collisions, in general. Figure 2 shows a comparison of this hash function with the Morton code, which demonstrates the lack of spatial locality. If the hash map is also used for rendering, we use a more simple hash function, directly based on the Morton code, as

$$\mathcal{H}(\bar{\boldsymbol{x}}) = \mathcal{Z}(\bar{\boldsymbol{x}})\%n_{\text{hash}}, \tag{17}$$

which will result in more hash collisions, but also much greater cache locality. Alternatively other hash functions could be used, that is, perfect hash functions [LH06], but they are more expensive to create and especially more computationally expensive to evaluate, making them unattractive for rendering. In general, we place the cell information in the hash map, if there were no collisions in this hash map entry, as this avoids one level of indirection.

The hash map itself is similar to the cell list in that it contains a begin entry and a length entry, where the begin entry now points to the first cell mapped to a hash table entry and the length entry indicates how many cells map to this hash table entry. If there is no cell, then the length is 0, if there is a single cell occupying this hash map the length is 1 and a length > 1 indicates a hash collision. The hash map, contrary to the cell list, is not compacted and as such allows us access via the hash index of an integer coordinate $\mathcal{H}(\bar{\boldsymbol{x}})$. The process required to find a specific cell c based on the cells integer coordinates $\bar{\boldsymbol{x}}_c$ is described in Algorithm 1.

To create the hash table $\mathbb{H}$, we first start by initializing all hash table entries as invalid, that is 0 length, and re-sort the list of occupied cells according to the hashed index of the first particle in this cell. We then, for each occupied cell i, set

$$\mathbb{H}^{\text{begin}}[\mathcal{H}_i] = i, \text{ if } i = 0 \vee \mathcal{H}_i \neq \mathcal{H}_{i-1}, \tag{18}$$

where we then set the length entry, for each occupied cell i, as

$$\mathbb{H}^{\text{length}}[\mathcal{H}_i] = i - \mathbb{H}^{\text{begin}}[\mathcal{H}_i] - 1, \text{ if } i = n_{\text{occupied}} \vee \mathcal{H}_i \neq \mathcal{H}_{i+1}, \tag{19}$$

which naturally handles hash collisions as the predicate is based on $\mathcal{H}_i \neq \mathcal{H}_{i+1}$ which is only true for the last cell associated with a certain hash value. The overall algorithm to create our hash map–based data structure, for a single level, is described in Algorithm 2.

Algorithm 1. The algorithm to access the cell associated with an arbitrary integer coordinate using our proposed sparse data structure without embedding $\mathbb{C}$ into $\mathbb{H}$. Note that an empty cell can map to the same hash map entry as an occupied cell, without causing a collision, and as such we always check $\boldsymbol{\mathcal{Z}_c = \mathcal{Z}_j}$ to avoid returning a wrong cell

Calculate $\bar{\boldsymbol{x}}_c$ for the cell we are looking for
Calculate $\mathcal{Z}_c$ and $\mathcal{H}_c$ for $\bar{\boldsymbol{x}}_c$
Look-up $b = \mathbb{H}^{\text{begin}}[\mathcal{H}_c]$ and $l = \mathbb{H}^{\text{length}}[\mathcal{H}_c]$
If $l \neq 0$
  For $h \in [b, b+l)$
    Look-up Particle $j = \mathbb{C}^{\text{begin}}_{\text{compact}}[h]$
    Calculate $\bar{\boldsymbol{x}}_j$ and $\mathcal{Z}_j$
    If $\mathcal{Z}_c = \mathcal{Z}_j$
      Return $\mathbb{C}_{\text{compact}}[b]$
Return not found

Algorithm 2. Our proposed single resolution data-structure algorithm. This algorithm first re-sorts all particles and then creates a compact cell table followed by the creation of our hash map as described in Section 4.1

Initialize
  Calculate $\mathbf{C}_{\max}$, $\mathbf{D}_{\min}$ and $\mathbf{D}_{\max}$ using reductions
  Calculate P^2 based on $\boldsymbol{D}$
  Re-sort particles using $\mathcal{Z}^{\text{fine}}$

Cell table creation
  Initialize $\mathbb{C} = -1$ and $\mathbb{C}^{\text{length}}_{\text{compact}} = 0$
  Create $\mathbb{C}$ based on Morton codes of consecutive particles
  Compact $\mathbb{C}$ into $\mathbb{C}^{\text{begin}}_{\text{compact}}$ and determine $\mathbb{C}^{\text{length}}_{\text{compact}}$
  Calculate $\mathcal{H}_i$ for all particles
  Re-sort $\mathbb{C}^{\text{begin}}_{\text{compact}}$ and $\mathbb{C}^{\text{length}}_{\text{compact}}$ based on the hash index of the first contained particle.

Hash map creation
  Initialize $\mathbb{H}^{\text{begin}} = -1$ and $\mathbb{H}^{\text{length}} = 0$
  Create $\mathbb{H}^{\text{begin}}$ based on compacted cell list
  Calculate $\mathbb{H}^{\text{length}}$
  Embed $\mathbb{C}_{\text{compact}}$ into $\mathbb{H}$ if $\mathbb{H}^{\text{length}} = 1$

4.2. Multi-level data structures

The prior section described our approach for a single fixed cell size, which would suffer from the same problems for adaptive

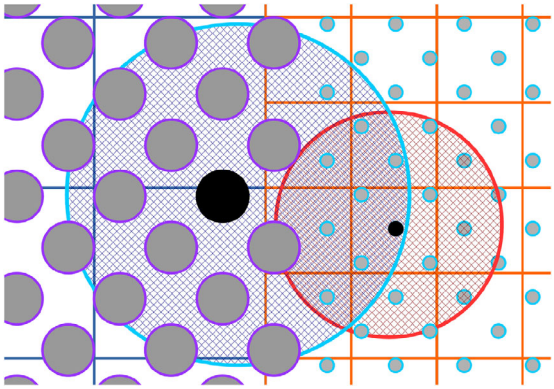

Figure 3: *Asymmetry interaction: Two particles at different levels may not mutually see each other due to the different cell size. Here, the lower level particle to the left sees the higher level particle to the right, but not vice versa.*

simulations as prior methods, due to a mismatch of cell size and particle resolution. Utilizing the self-similarity of the underlying Morton code, however, we can efficiently create multiple, distinct, data structures for different cell sizes on the same underlying particle data. All power of 2 times coarser resolutions follow the same underlying ordering, due the octree-like structure of Morton codes. The lowest required resolution for the data structure is still $\mathbf{C}_{\max}$, resulting in a coarse grid size of $\mathbf{D}_c$; see Equation (11). Using the largest component of the grid, $P = \max(\boldsymbol{D}_c^x, \boldsymbol{D}_c^y, \boldsymbol{D}_c^z)$, we determine the smallest cell size possible, as the bit length of the Morton code used limits the number of cells per dimension to $\#K$; see Section 4.1, as

$$C_{\text{fine}} = C_{\max} \frac{2^{\lceil \log_2(P) \rceil}}{\#K}. \tag{20}$$

Here $2^{\lceil \log_2(P) \rceil} / \#K = 2^{-L_{\text{fine}}}$, $L_{\text{fine}} \in N$ is the refinement factor. The algorithm described in Section 4.1 can now be extended by creating the cell list and hash map for a Morton code $\mathcal{Z}^{\max}$ based on $C_{\max}$ and additional finer levels $0 < L \leq L_{\text{fine}}$ using the integer coordinates

$$\bar{\mathbf{x}}^L = \left\lfloor \frac{\mathbf{x} - \mathbf{D}_{\min}}{C_{\max} \cdot 2^{-L}} \right\rfloor. \tag{21}$$

$L = 0$ results in the same data structures as the single-level version of this algorithm and $L = L_{\text{fine}}$ the finest possible data structure, with the given Morton code. The corresponding Morton code is determined as $\mathcal{Z}^L(\boldsymbol{x}) = \mathcal{Z}(\bar{\boldsymbol{x}}^L)$. We relate the maximum level $L_{\max}$ to the maximal adaptivity ratio α of the simulation as

$$L_{\max} = \min\left\{ \left\lceil \log_2 \sqrt[3]{\alpha} \right\rceil, L_{\text{fine}} \right\}, \tag{22}$$

and generate the data structure for all levels $0 \leq L \leq L_{\max}$. We store all data structures within single continuous arrays, which allows us to calculate the hashed indices by simply adding an offset based on the level to any hash function

$$\mathcal{Z}^L(\bar{\boldsymbol{x}}) = L n_{\text{hash}} + \mathcal{Z}(\bar{\boldsymbol{x}}). \tag{23}$$

Furthermore, we determine the appropriate level for a particle i according to its support radius as

Algorithm 3. Changes required to create multiple levels of our data structure

Initialize
- **Calculate** $\mathbf{C}_{\max}$, $\mathbf{D}_{\min}$ and $\mathbf{D}_{\max}$ using reductions
- **Calculate** P^2 based on $\boldsymbol{D}$
- **Re-sort** particles using $\mathcal{Z}^{\text{fine}}$
- **Calculate** Ideal level L_i for every particle

For every Multi-Level-Memory level L
- **Execute** Cell table creation and Hash map creation using $\mathcal{Z}^L$ and $\mathcal{H}^L$ respectively.

Finalize
- **Enforce** symmetric interactions

$$L_i = clamp\left(\left\lfloor -\log_2 \frac{h_i}{C_{\max}} \right\rfloor, 0, L_{\text{fine}} - 1 \right). \tag{24}$$

Thus, every particle can easily access all neighbours at any scale $L \in [0, L_{\max}]$ using the data structure for L. However, when a particle i only looks for neighbours at its level L_i, we may encounter asymmetric interactions, as neighbour searches are limited by the cell size for level L_i; see Figure 3.

More formally, asymmetric interactions occur when a particle i of lower level L_i is interacting with a particle k of a higher level L_k, that is, if $L_k > L_i \wedge x_{ik} < h_{ik}$. In order to resolve the asymmetry, we iterate over all neighbouring particles k of i and determine their integer coordinate distance, for the cell size associated with L_k as $\bar{\boldsymbol{x}}_{ik}^{L_k} = \bar{\boldsymbol{x}}_i^{L_k} - \bar{\boldsymbol{x}}_k^{L_k}$. We use $\bar{\boldsymbol{x}}_{ik}^{L_k}$ to identify the problem case that k does not see particle i by checking

$$\left\| \bar{\boldsymbol{x}}_{ik}^{L_k} \right\|_1 = \max(|\bar{\boldsymbol{x}}_{ik,x}^{L_k}|, |\bar{\boldsymbol{x}}_{ik,y}^{L_k}|, |\bar{\boldsymbol{x}}_{ik,z}^{L_k}|) > 1. \tag{25}$$

We then resolve this issue by atomically updating L_k to be at least L_i, as this ensures that k will search distant enough cells to find i. The overall changes to the single resolution algorithm are relatively minor but are outlined in Algorithm 3.

4.3. Rendering data structure

A naïve way to render the fluid surface using ray-casting would be to treat the underlying cell structure as a grid and use traditional ray-casting approaches, that is, [KW03]; however, this would re-

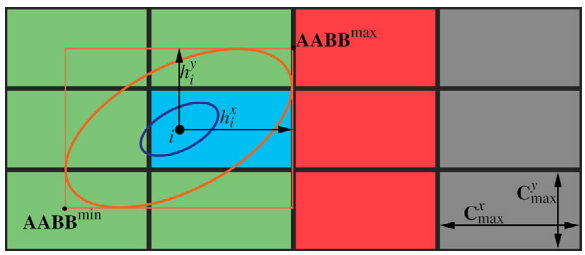

Figure 4: *An anisotropic particle in a non-cubic grid. The blue cell contains the actual particle position, red cells are neighbouring but cannot contain fluid surface, green cells are neighbouring and can contain fluid surface and grey cells are not neighbouring. The fluid surface due to the particle is coloured dark blue.*

quire checking every cell inside of the simulation domain along a ray for potential ray-fluid intersections, even though most cells contain no actual fluid surface. Alternatively, simply checking for fluid-ray intersections within occupied cells does not work as the fluid surface of particles within one cell extends into other cells; see Figure 4. At worst, the fluid surface for every single particle could be distributed amongst 27 cells, which would require a total of $27 \cdot n_{\text{particles}}$ cells; however, explicitly storing these cells is not practical, due to limited memory resources on GPUs. Instead, we propose to store this information only implicitly using a hash map, as it suffices to mark whether any cell, that potentially contains fluid surface, maps to a certain hash map entry. Moreover, cells that contain no fluid particles, themselves, can still contain fluid surfaces. Thus using the data-structure directly requires checking every cell intersected by a ray, instead of only checking cells that can contain fluid surfaces.

The main purpose of this data structure is not to find particles close to a spatial location, as was the case for the simulation data structure, but instead to check whether a cell at a spatial location could contain fluid surface. Instead of storing references to particle ranges contained in a cell, we only store the Morton code of a cell, as many cells that could contain fluid surface do not contain any particles. The hash map used here uses the same size as the one for the simulation, which, due to the low number of cells containing fluid surface, results in only a moderate number of collisions, even when using non-ideal hash functions.

In order to find cells containing particles at the fluid surface, we first calculate the signed surface distance ϕ for all particles, using the method of Horvath and Solenthaler [HS13]. A particle that is close enough to the surface, that is, $\phi_i < r_i$, is marked as a surface particle. We then iterate over all cells in the compact list of occupied cells $\mathbb{C}_{\text{compact}}$ and create a list of surface cells

$$\mathbb{R}[i] = \begin{cases} \mathcal{Z}(\mathbf{x}_{c_i^0}), & \text{if } \exists c \text{ marked as surface } : c \in \mathcal{C}_i, \\ -1, & \text{else,} \end{cases} \tag{26}$$

where $\mathcal{C}_i$ is the set of particles contained in cell i and c_i^0 is the first particle contained in a cell, that is, $c_i^0 = \mathbb{C}^{\text{begin}}_{\text{compact}}[i]$. Next, we use a compact operation on this list to yield a compacted list of cells containing surface particles, denoted as $\mathbb{F}$. Next, similar to the simulation data structure, we sort this list according to the hash function and create a hash table $\mathbb{H}_R$, as before, using this sorted compact list. Next, we embed the cell information in the hash table, if there was no collision for a hash entry, as this avoids one level of indirection. At the end of this process, a hash table entry either contains a cell, represented as a Morton code, or a range of cells that map to this hash entry. We then calculate an AABB for every cell $i \in \mathbb{F}$, based on the contained particles, as

$$\mathbf{AABB}_i^{\min} = \min_{c \in \mathcal{C}_i} \mathbf{x}_c - \mathbf{h}_c; \mathbf{AABB}_i^{\max} = \max_{c \in \mathcal{C}_i} \mathbf{x}_c + \mathbf{h}_c, \tag{27}$$

where $\mathbf{h}$ is the extent of support along the simulation axes. Finally, for every cell $i \in \mathbb{F}$, we check if any of the surrounding cells N_i overlap the bounding box of i. If we find an overlap with cell $n \in N_i$, we check if $n \in \mathbb{F}$ using $\mathbb{H}^R$, in which case nothing needs to be done. If $n \notin \mathbb{F}$ and the corresponding hash map entry is empty, we set the entry to indicate n as potentially having fluid surface. If the hash map entry was already occupied, we mark the entry as having a collision, which means that any cell mapping to this hash entry is assumed to have fluid surface in it. This process is also described in Algorithm 1.

Algorithm 4. Changes required to create multiple levels of our data structure

Initialize
- **Determine** signed surface distance ϕ for all particles [HS13]
- **Set** $s[i] = 1$ if particle i close to surface, otherwise $s[i] = 0$

Find surface particle cells
- **Determine** cells containing surface particles as $\mathbb{R}$; see Eqn. 26
- **Compact** cells containing surface particles into $\mathbb{F}$
- **Sort** $\mathbb{F}$ based on hash function using stored Morton codes
- **Create** $\mathbb{H}_R^{\text{begin}}$ based on sorted list
- **Calculate** $\mathbb{H}_R^{\text{length}}$
- **For all** hash entries h with $\mathbb{H}_R^{\text{length}}[h] = 1$
  - **Embed** Store Morton code of corresponding cell directly in $\mathbb{H}_R[h]$
  - **Mark** entry h as *particle cell*
- **For all** hash entries h with $\mathbb{H}_R^{\text{length}}[h] \geq 1$
  - **Mark** entry h as *particle cell collision*

Spread surface cells
- **Calculate** $\mathbf{AABB}_c^{\min}$ and $\mathbf{AABB}_c^{\max}$ for all cells $c \in \mathbb{F}$
- **For each** neighboring cell n of all cells $c \in \mathbb{F}$
  - **If** n overlaps AABB of c
    - **Calculate** Morton code $\mathcal{Z}_n$ and hash key $\mathcal{H}_n$ for cell n
    - **If** $\mathbb{H}_R[\mathcal{H}_n]$ empty
      - **Store** $\mathcal{Z}_n$ in $\mathbb{H}_R[\mathcal{H}_n]$
      - **Mark** $\mathbb{H}_R[\mathcal{H}_n]$ as *potential surface cell*
    - **Else If** $\mathbb{H}_R[\mathcal{H}_n]$ does not contain $\mathcal{Z}_n$ already
      - **Mark** $\mathbb{H}_R[\mathcal{H}_n]$ as *collision*

Given a position $\mathbf{x}$, with a corresponding cell q, we can query this data structure using the corresponding Morton code $\mathcal{Z}_q$ and hash key $\mathcal{H}_q$. If $\mathbb{H}_R[\mathcal{H}_q]$ is empty, no fluid surface can be contained in q. If $\mathbb{H}_R[\mathcal{H}_q]$ is marked as *collision*, q potentially contains fluid surface. Otherwise, if $\mathbb{H}_R[\mathcal{H}_q]$ contains $\mathcal{Z}_q$, then q contains particles at the fluid surface. This rendering data structure is also described in Algorithm 4.

4.4. Data structure memory layout

To reduce the memory consumption, and improve the performance, of our data structure, several optimizations can be made to the memory layout of the data structure. These optimizations are based on a word size of 4 Byte, which reduces the number of required atomic operations significantly. For the Morton code, we use 30 bits in non-adaptive simulations and 60 bits in adaptive simulations.

A hash table and cell list entry of the simulation data structure contain both a beginning and length entry, using a 4 Byte integer each. To embed a cell list entry in the hash map, we reserve a single bit of the beginning entry, as an indicator for the kind of data, which limits the maximum number of particles to at most $2^{31} - 1$, which should not cause problems on single GPUs.

As stated at the beginning of Section 4, an average cell contains only $\frac{3}{4\pi} N_h \approx 11.94$ particles, when using a cubic spline kernel, in non-adaptive simulations. Furthermore, the maximum number of

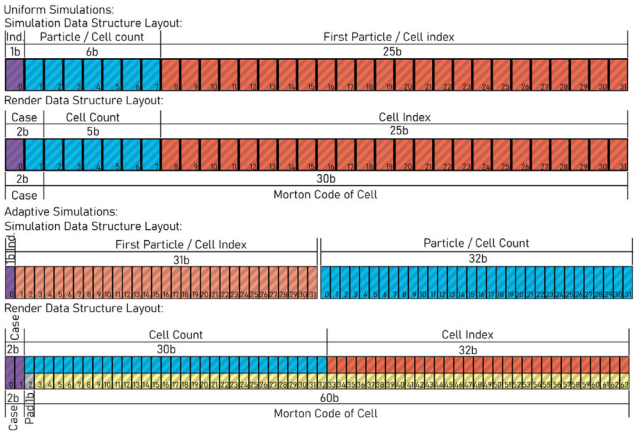

Figure 5: *Memory layout of the data structures for simulation and rendering for uniform and adaptive simulations. The render data layout uses a union indicated by the split fields.*

particles we can simulate on a single GPU is below 32 M. As such, we only require 25 bits to represent the cell begin entry and approximately 3.6 bits to represent the number of particles per cell. As we still require a single bit to indicate if a cell entry is embedded, we chose a 25 bit integer for the cell begin entry and a 6 bit integer for the cell length entry; see Figure 5. To account for larger support radii for rendering, which might increase the number of particles per cell significantly, we keep track of the actual number of particles in each cell, in a separate list, where a length entry of 63 in a cell indicates a required additional lookup. The data structure for rendering needs to indicate four cases, requiring 2 bits, as well as store the Morton code, requiring 30 bits in non-adaptive simulations. As such, we can store the Morton code for a cell, and the case indicator, within a single $4B$ entry. To account for the 2 bits for the cases a hash entry in this data structure consists of a 25 bit integer for the begin entry and a 5 bit integer for the length entry, limiting the number of cells mapped to the same hash entry to 32, which should not be exceeded in practice. To embed cell information into the hash map, we utilize a union, with a case field indicating the kind of entry; see Figure 5

5. Rendering

Intersecting a ray with the fluid surface could, naïvely, be done by simply evaluating a function at every point along the ray and finding the closest value that matches the desired iso value; however, this is not practically possible. Instead, we propose to first find rays that can potentially intersect the fluid, that is, rays intersecting the simulation domain. Next, we iterate along those rays, over the cells in the data structure, to find cells that can potentially contain fluid surfaces. Finally, we evaluate the ray-fluid intersection, but only within an intersected cell; see Figure 6. Given a ray

$$\text{ray}_i(\lambda) = \mathbf{x}_i^{\text{o}} + \lambda \cdot \mathbf{d}_i, \lambda \in \mathbb{R}_0^+, \tag{28}$$

starting at $\mathbf{x}_i^{\text{o}}$ with direction $\mathbf{d}_i$, we check for an intersection of a ray with the simulation domain using a standard AABB intersection test. Next, in case of an intersection, we calculate the first cell hit by the ray and traverse the domain, along the ray; see Section 5.1, until

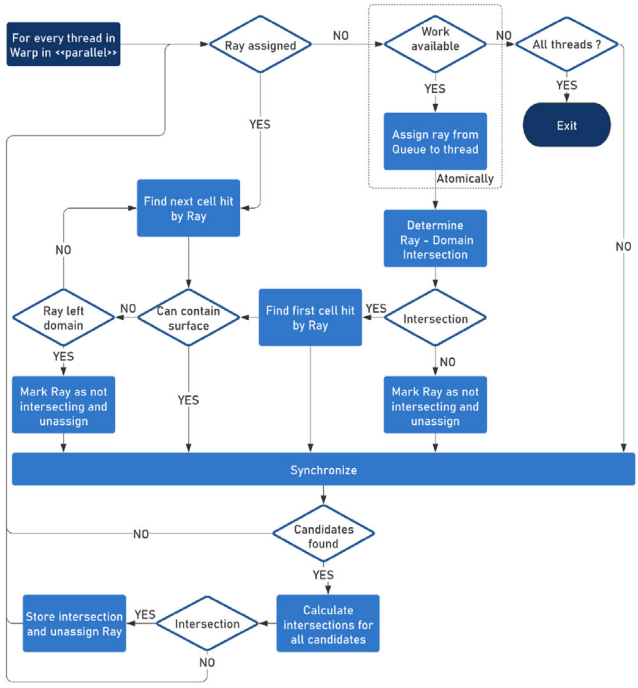

Figure 6: *The warp program used to calculate ray fluid intersections. Warp here refers to a group of threads, that is, 32, executing in parallel on either a GPU or CPU.*

either the ray exits the simulation domain, or we find a cell that can potentially contain fluid surfaces, see Section 4.3. For a found cell, we then calculate an intersection using either the cells directly; see Section 5.2, treating particles as spheres; see Section 5.3, or using a surface function to determine an iso surface; see Section 5.4.

To implement this process on a GPU, all rays are stored in a queue and we process one ray at a time in a thread. The simulation domain intersection and traversal of the data structure is executed independently, on all threads in a warp, as these do not require any inter-thread communication. Next, after synchronizing the threads in a warp, we check if any thread has found a cell that can potentially contain fluid surfaces. For these potential intersections, we then calculate the actual ray-fluid intersection and store the results, if an intersection is found.

5.1. Data structure traversal

In case an intersection of a ray with the simulation domain AABB exists, this calculation yields a close distance λ^{min} and far distance λ^{max} to the domain intersections along the ray, where we clamp λ^{min} to positive values, which yields the distance along the ray, within the domain, as $\lambda_l = \lambda^{\text{max}} - \lambda_+^{\text{min}}$. Using λ_+^{min}, we determine $\mathbf{x}_{\text{min}} = \text{ray}(\lambda_+^{\text{min}})$, which, in turn, yields the integer coordinates of the first cell $\bar{\mathbf{x}}_{\text{min}}$ hit by the ray, within the simulation domain. Similar to Amanatides and Woo [AW87], we traverse the simulation domain by first calculating the integer ray direction as

$$\bar{\mathbf{d}} = \left[\text{sign}(\mathbf{d}^x), \text{sign}(\mathbf{d}^y), \text{sign}(\mathbf{d}^z)\right]. \tag{29}$$

Algorithm 5. Domain traversal algorithm; based on [AW87]

Intersect ray with simulation domain AABB, yielding $\lambda^{\min}$ and $\lambda^{\max}$
$\lambda^{\min}_{+} \leftarrow \max(0, \lambda^{\min})$
$\mathbf{x}_{\min} \leftarrow \text{ray}(\lambda^{\min}_{+})$
Determine $\bar{\mathbf{x}}_{\min}$ using Eqn. 12
$\bar{\mathbf{d}} \leftarrow [\text{sign}(\mathbf{d}^x), \text{sign}(\mathbf{d}^y), \text{sign}(\mathbf{d}^z)]$
$\mathbf{x}_{\text{next}} \leftarrow \mathbf{D}_{\min} + \left(\bar{\mathbf{x}}_{\min} + \max(\bar{\mathbf{d}}, \mathbf{0})\right) \circ \mathbf{C}_{\max}$
$\lambda_n \leftarrow \mathbf{x}_{\text{next}} - \mathbf{x}_{\min} \oslash \mathbf{d}$
$\lambda_{inc} \leftarrow \bar{\mathbf{d}} \oslash \mathbf{d} \circ \mathbf{C}_{\max}$
$\lambda_l \leftarrow \mathbf{x}_{\min} - \text{ray}(\lambda^{\min}_{+})$
Initialize $\bar{\mathbf{c}} = \bar{\mathbf{x}}_{\min}$
Iterate
  If $\bar{\mathbf{c}}$ can contain fluid surfaces; see Sec. 4.3
    Return Cell found.
  If $\lambda^x_n < \lambda^y_n$
    If $\lambda^x_n < \lambda^z_n$
      If $\lambda^x_n > \lambda_l$: **Return** No cell found.
      $\bar{\mathbf{c}}^x \leftarrow \bar{\mathbf{c}}^x + \bar{\mathbf{d}}^x \qquad \lambda^x_n \leftarrow \lambda^x_n \lambda^x_{\text{inc}}$
    Else
      If $\lambda^z_n > \lambda_l$: **Return** No cell found.
      $\bar{\mathbf{c}}^z \leftarrow \bar{\mathbf{c}}^z + \bar{\mathbf{d}}^z \qquad \lambda^z_n \leftarrow \lambda^z_n + \lambda^z_{\text{inc}}$
  Else
    If $\lambda^y_n < \lambda^z_n$
      If $\lambda^y_n > \lambda_l$: **Return** No cell found.
      $\bar{\mathbf{c}}^y \leftarrow \bar{\mathbf{c}}^y + \bar{\mathbf{d}}^y \qquad \lambda^y_n \leftarrow \lambda^y_n + \lambda^y_{\text{inc}}$
    Else
      If $\lambda^z_n > \lambda_l$: **Return** No cell found.
      $\bar{\mathbf{c}}^z \leftarrow \bar{\mathbf{c}}^z + \bar{\mathbf{d}}^z \qquad \lambda^z_n \leftarrow \lambda^z_n + \lambda^z_{\text{inc}}$

We then determine the next cell boundary, intersected by the ray, as

$$\mathbf{x}_{\text{next}} = \mathbf{D}_{\min} + \left(\bar{\mathbf{x}}_{\min} + \max(\bar{\mathbf{d}}, \mathbf{0})\right) \circ \mathbf{C}_{\max}, \tag{30}$$

where $\circ$ denotes the component-wise multiplication (Hadamard product) and the maximum is applied component-wise. Next, we determine the distance along the ray, until the next cell boundary is intersected, as

$$\lambda_n = \mathbf{x}_{\text{next}} - \mathbf{x}_{\min} \oslash \mathbf{d}, \tag{31}$$

where $\oslash$ is the component-wise division (Hadamard division). For small components of $\mathbf{d}$, that is, $|\mathbf{d}^x| < 10^{-6}$, we set the corresponding component of λ_n to infinite. Finally, we start the traversal in the cell $\bar{\mathbf{c}} = \bar{\mathbf{x}}_{\min}$.

We first check if $\bar{\mathbf{c}}$ can contain fluid surfaces; see Section 4.3, and in this case stop the traversal, otherwise we move to the next cell. This increment, using $\bar{\mathbf{d}}$, is based on the smallest component in λ_n; however, if the smallest component λ_n was larger than λ_l, we stop the traversal. Otherwise we increment the smallest component in $\lambda_{\mathbf{n}}$, using $\bar{\mathbf{d}} \oslash \mathbf{d} \circ \mathbf{C}_{\max}$, and repeat this process; see Algorithm 5.

5.2. Data structure visualization

For simulations with sparse data structures, visualizing the occupied cells is a useful, and fairly straight forward, visualization. To visualize the data structure, after the traversal algorithm yields a cell $\mathbf{c}$ potentially containing fluid surface, we check if there is a cell at location $\mathbf{c}$ in the simulation data structure, which is only the case for cells containing particles, not just fluid surface; see Section 4.1.

Algorithm 6. Intersection calculation after all threads in a warp have evaluated $\boldsymbol{\phi}$. ***ballot*(*p*)** is an intra-warp voting function of the predicate $\boldsymbol{p}$, ***shuffle*(*t*, *i*)** returns the value of $\boldsymbol{t}$ for thread $\boldsymbol{i}$, ***active*** is a mask indicating active threads and ***ffs*(*b*)** returns the index of the first set bit in $\boldsymbol{b}$. The result of this algorithm is a depth $\boldsymbol{t}$

In parallel for each thread w in warp
  $\phi_w \leftarrow \phi(\mathbf{x}_w)$
  $\text{mask}_L \leftarrow ballot(\phi_s \geq \phi^{\text{iso}})$
  $\text{mask}_H \leftarrow ballot(\phi_s < \phi^{\text{iso}})$
  $\text{mask}_F \leftarrow (\text{mask}_L << 1) \,\&\, \text{mask}_H$
  $\text{mask}_R \leftarrow (\text{mask}_L << 1) \,\&\, \text{mask}_H$
  If $\text{mask}_L \& 0x1$
    $i \leftarrow \text{ffs}(\text{mask}_L) - 1$
  Else
    $i \leftarrow \text{ffs}(\text{mask}_L) - 1$
  If $\text{mask}_L \neq 0 \wedge \text{mask}_L \neq active$
    $x_0 \leftarrow i - 1$
    $y_0 \leftarrow \text{shuffle}(\phi_w, i - 1)$
    $x_1 \leftarrow i$
    $y_1 \leftarrow \text{shuffle}(\phi_w, i)$
    $dy \leftarrow= y_1 - y_0$
    $\alpha \leftarrow \phi^{\text{iso}} - y_0 / dy$
    $t_0 \leftarrow \text{shuffle}(\lambda_w, i - 1)$
    $t_1 \leftarrow \text{shuffle}(\lambda_w, i)$
    $t \leftarrow (1 - \alpha) t_0 + \alpha t_1$

If there is no cell, containing particles, we continue the traversal; otherwise, we calculate the centre of the found cell as

$$\mathbf{x}_c = \mathbf{D}_{\min} + \mathbf{c} \circ \mathbf{C}_{\max}, \tag{32}$$

with a half cell size $r_c = 0.5\mathbf{C}_{\max}$. Calculating the intersection of the ray with this cell yields $\lambda^{\min}_c$ and $\lambda^{\max}_c$ and the point of ray-cell intersection $\mathbf{x}_i = \text{ray}(\lambda^{\min}_c)$. Using $\mathbf{d} = \mathbf{x}_i - \mathbf{x}_c$, we determine a normal, where a cell is represented as a cuboid, as

$$\mathbf{n}_c = [(1 + \epsilon)\mathbf{d} \oslash r_c], \tag{33}$$

where ϵ is a small positive value and $[\cdot]$ denotes rounding to the nearest integer. As the colour for the intersected cell, we use the colour of the first particle in $\mathbf{c}$. This process yields all required information, that is, depth, normal, and colour, required to render the data structure.

5.3. Particle visualization

Rendering particles as spheres, overlapping the cell $\mathbf{c}$ first involves finding the intersection of the current ray with the cell, similar to the cell visualization, which yields $\lambda^{\max}_c$ and $\lambda^{\min}_c$. Using the simulation data structure, we iterate over all particles that can potentially overlap $\mathbf{c}$, that is, all those contained within neighbouring cells of $\mathbf{c}$ and $\mathbf{c}$. For each of these particles j, we can calculate a ray sphere intersection, which, if there was an intersection, yields an intersection distance λ_j. If $\lambda_j \in [\lambda^{\min}_c, \lambda^{\max}_c]$, we store j and λ_j, if this was the closest intersection found so far, and keep iterating over the remaining particles. If there was an intersection with any particle, we can then determine the intersection point $\mathbf{x}_i = \text{ray}(\lambda_j)$ and the intersection normal $\mathbf{n}_i = \mathbf{x}_i - \mathbf{x}_j$. We use the colour associated with the particle, that is, using colour mapping of some quantity, as the colour for the intersection.

5.4. Surface rendering

Rendering the fluid surface directly requires extracting an iso surface of some field quantity ϕ. There are many different ways to determine this quantity; however, common choices include simply using the density ρ, the surface distance function of Zhu and Bridson [ZB05], or the anisotropic surface distance function of Yu and Turk [YT13]. For our approach, the choice of function does not influence the process beyond potentially increasing the computational workload. Therefore, we assume an arbitrary field quantity $\phi(\mathbf{x}) : \mathbb{R}^3 \to \mathbb{R}$, with an iso value of ϕ^{iso}. To calculate the ray-surface intersection, we evaluate a single intersection of a ray with a corresponding found cell $\bar{\mathbf{c}}$ in parallel, using all threads in a warp, where we sequentially process the overall set of potential intersections.

To evaluate a single ray-surface intersection we first calculate the intersection of the ray with the found the cell $\bar{\mathbf{c}}$, yielding $\lambda_c^{\min}$ and $\lambda_c^{\max}$. For a warp size of w, we then subdivide the interval $[\max\{\lambda_c^{\min}, 0\}, \lambda_c^{\max})$ into $w-1$ segments, where each segment $s \in [0, w-1)$ is assigned a starting position

$$\mathbf{x}_s = \text{ray}\left(\max\{\lambda_c^{\min}, 0\} + \frac{s}{w-1}\left(\lambda_c^{\max} - \max\{\lambda_c^{\min}, 0\}\right)\right), \quad (34)$$

where each thread in a warp is assigned one position, with the last thread being assigned $\mathbf{x} = \text{ray}(\lambda_c^{\max})$, and each ray is assigned an according distance value λ_s.

To evaluate ϕ_s, at each assigned position $\mathbf{x}_s$, we iterate over all particles in all neighbouring cells, where the cell entries are stored in shared memory. Afterwards, utilizing Algorithm 6, we find an intersection depth λ_f and, if there was an intersection, store λ_f and mark the ray as processed; otherwise, we keep traversing the simulation domain for this ray. Calculating the intersection normal can be done directly by evaluating $\nabla\phi(\mathbf{x})$. Note the synchronization required here can be implemented implicitly by checking after each cell traversal if any thread in a warp has found a cell that can contain fluid surfaces and performing the check immediately. This avoids threads within a warp waiting for other threads and reduces warp divergence. We refer the reader to the supplementary materials for the implementation of this optimization.

6. Results and Discussion

All results were simulated and rendered using a single Nvidia RTX 2080 Ti GPU with 11 GiB of VRAM and an AMD Ryzen 3970x CPU with 64 GB of RAM. We used DFSPH [BK15], with a density error limit of 0.01% and a divergence error limit of 0.1%, and density maps [KB17] to represent rigid objects. Artificial viscosity was modelled based on XSPH [Mon05], surface tension was modelled based on [AAT13] and fluid air phase interactions were modelled based on [GBP*17]. We used the vorticity refinement method of [BKKW18] and the spatially adaptive method of [WHK17]. Our overall uni-directional ray-tracing algorithm is implemented in CUDA and inspired by [PJH16], where we used a simple bounding volume hierarchy for rigid objects, and calculate each bounce of a ray using a loop on the CPU to avoid recursions on the GPU. We implemented our data structure and rendering approach in the open source SPH framework openMaelstrom [Win]. For an in-depth discussion of the simulation data structure, we refer the reader to [WK19]. For all renderings we used a fixed resolution of 1920×1080 and a frame rate of 60 fps, with 35 primary rays cast per pixel with 6 and 16 bounces per primary ray for opaque and transparent renderings, respectively. .

Test Scenes: We evaluated our approach using three test scenes. The *Dam break* test scene involves the collision of two initial fluid volumes in a cubic domain; see Figure 8, with a particle resolution of $r = 0.175$ m and 3.9 million particles. The *Inlet* test scene involves a stream of liquid flowing along a channel and colliding with an obstacle; see Figure 10, with a particle resolution of $r = 0.35$ m and up to 2.2 million particles. The *Drop* test scene involves dropping a sphere of liquid into a basin; see Figure 7, with a particle resolution of $r = 0.2$ m and 1.5 million particles. For simulation performance see Table 1, and for rendering performance see Table 2.

Ray-casting performance: To evaluate the performance of our method, we first consider the performance cost of the primary ray intersection. These rays are cast directly from the camera into the scene and traverse similar cells. Primary ray intersections can be used to implement a deferred shading approach, as the intersection yields both a depth and normal, or using a simple direct lighting model. When using an isotropic surface function [ZB05], we find good performance in all scenes, that is, primary intersection tests requiring less than 50 ms. For an anisotropic surface function [YT13], the overall computational cost is significantly higher, due to a more expensive surface function and larger cell non-cubic cells. The cost of the primary intersection, regardless of which surfacing method specific is chosen, changes significantly depending on the relation of camera and fluid geometry. This involves many aspects, that is, many rays parallel to a flat surface require many unnecessary intersection tests, whereas rays orthogonal to a flat surface require few unnecessary intersection tests. Additionally the performance changes depending on how many rays can even intersect the fluid, that is, the fluid surface might be partially occluded by some other geometry, and also depends on the screen-space size of the fluid simulation. Finally, the overhead for the construction of the render data structure is fairly low and is only required once per frame.

Opaque fluid rendering: Rendering an opaque surface does not just require the primary intersection, but a full path trace per ray. The directions of bounced rays show very little spatial coherency, which causes the rays to not traverse similar cells, as was the case for the primary intersection. In all scenes we tested (see Table 2), we found that the performance of the first bounced ray is always lower than the primary ray, and the cost of following bounces decreases with each bounce. However, if the number of rays intersecting the fluid domain falls below the number of threads, we can executed in parallel, that is, after a significant number of bounces, the cost stays relatively fixed. Additionally, the performance depends on the geometry of the surface, that is, any bounced ray from an intersection with a cube shared surface is guaranteed to not intersect the surface directly. However, more complex geometries that, for instance, contain cavities where rays enter but rarely exit, significantly increase computational cost. Furthermore, for an opaque rendering, we require multiple samples per pixel, which further increases the computational cost.

Table 1: *The timing values shown here are given as the average time, required to simulate 1/60 s, over an entire simulation as measurements in milliseconds. DFSPH refers to the combined time required to solve for both incompressibility and divergence-freedom. The timing for the data structure here refers to all steps in Algorithm 2, that is, including resorting and data structure creation. Neighbour-list timings refer to the timing of the construction of a constrained neighbour-list [WK19].*

Scene	Figure	Particles	Radius/m	Δt/ms	Domain	Structure/ms	Neighbour-list/ms	DFSPH/ms	Overall/ms
Drop	7	1.5M	0.2	4.0	$68x68x204$	30.5	77.3	1150.6	2079
Dam break	8	3.9M	0.175	5.6	156^3	41.5	174.9	2312	2919
Inlet	9	2.2M	0.35	6.0	$156x156x76$	22.1	66.8	316.2	517

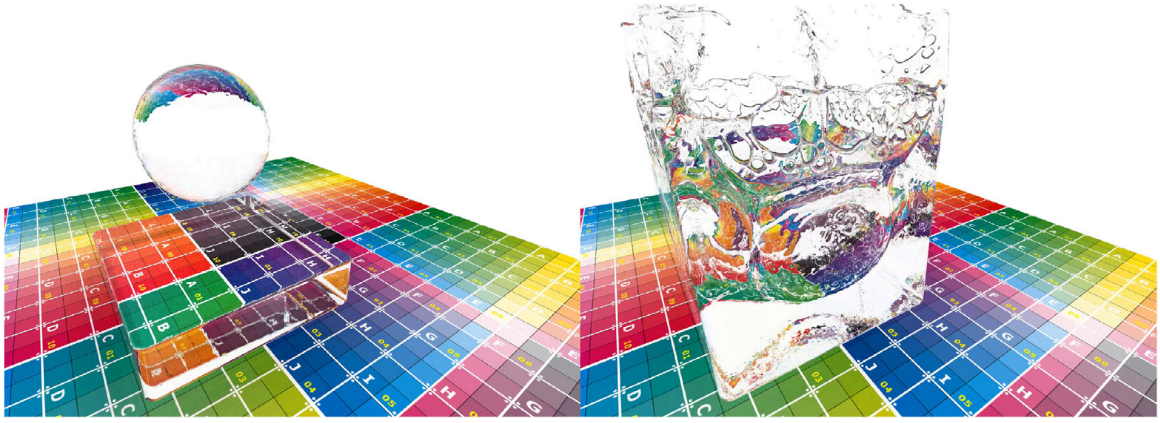

Figure 7: *Rendering of the* drop *scene. A spherical fluid volume is dropped into a basin and rendered using an anisotropic surface function as a transparent fluid. The left part of the image shows the initial configuration, whereas the right part shows a later point in time.*

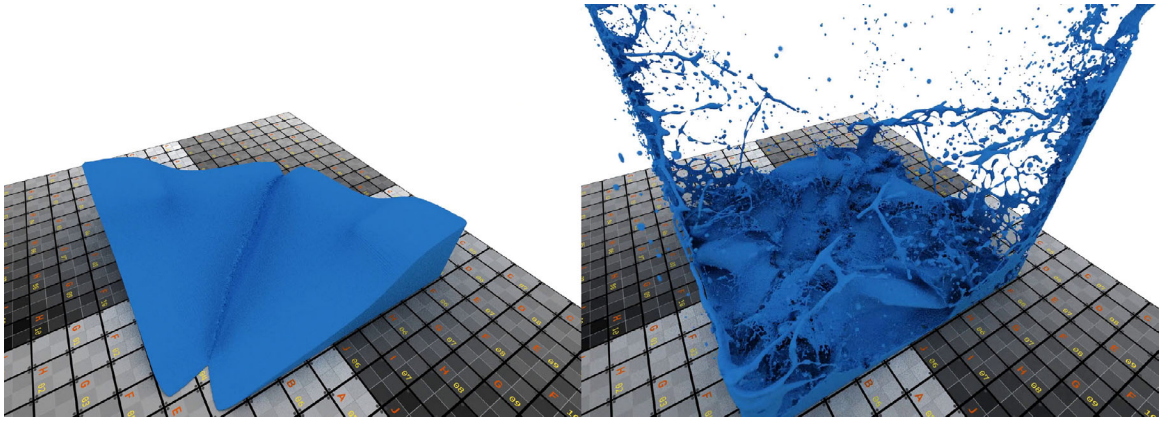

Figure 8: *Rendering of the* dam break *scene. Two initial fluid volumes collide in this scene and are rendered using an isotropic surface function as an opaque fluid. The left part of the image shows the simulation immediately after the collision, whereas the right part shows a much later point in time.*

Transparent fluid rendering: Although an opaque surface rendering does not cause rays to transmit into the fluid, a transparent surface rendering will cause rays to either reflect or refract into the fluid. This means that the computational requirements are significantly higher, as rays traversing the interior of the fluid require many checks for potential fluid intersections, which are mostly negative as a significant number of internal cells could contain fluid surface. Additionally, rays moving on the interior of a fluid volume might be reflected multiple times before they exit the fluid, requiring significantly more bounces per ray than an opaque surface rendering. These additional computational costs increase the time per sample-per-pixel by about 3 times. Although there are options to improve the computational performance, such as lowering the number of samples per pixels, utilizing a screen-space

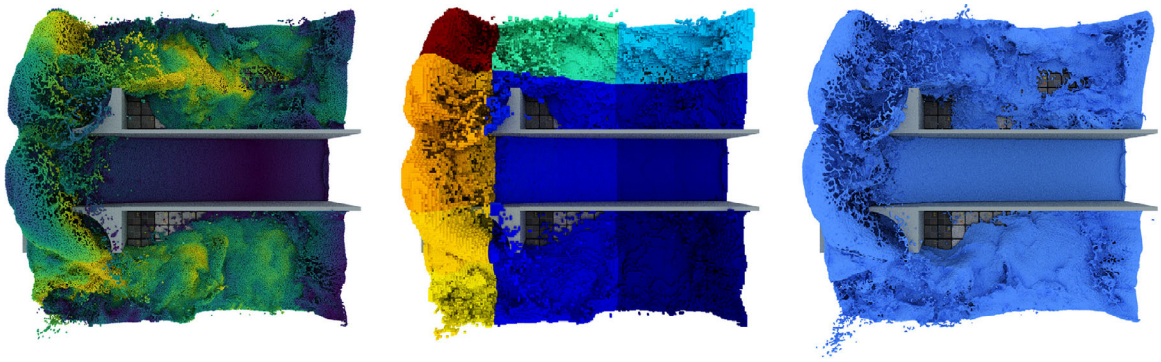

Figure 9: *Rendering of the* inlet *scene. A fluid inlet stream impacts a rigid obstacle visualized as particle spheres, with particle velocities colour coded from purple (0 m/s) to yellow (30 m/s) (left), a visualization of the data structure, with the Morton code of the first particle in each cell colour coded from blue to red (middle) and an isotropic surface extraction (right).*

Table 2: *Timing for intersection tests. This table states the average computation time over the entire simulation in milliseconds per sample per pixel, separately for the first four intersection and the total rendering time including illumination calculations per sample per pixel.* Grid and *Particles stands for visualizing the underlying data structure and particles as spheres, respectively.*

		Intersection tests				Render
Method	Init	First	Second	Third	Fourth	Overall
		Drop scene, Figure 7				
[YT13]	12.1	90.1	113.4	103.5	84.8	838.7
		Dam break scene, Figure 8				
[ZB05]	23	37.5	49.4	17.7	10.5	132.8
Particles	22.2	14.3	23.5	12.6	9.5	69.2
		Inlet scene top down, Figure 9				
[ZB05]	12.7	30.4	41.8	19.1	14.6	136.7
[YT13]	14.1	149.8	204.2	60.6	46.9	503.6
Grid	12.7	2.3	3.4	1.7	1.3	27.86
Particles	12.6	12.9	19.9	13.7	10.9	79.8
		Inlet scene side view, Figure 10				
[ZB05]	12.8	46.9	52.7	25.3	18.2	180.2
[YT13]	14.0	237.5	273.3	124.7	79.5	785.3

based refraction model [vdLGS09] or considering refraction on the first bounce only, these methods have a large and complex impact on visual quality, the study of which is beyond the scope of this paper. Although our method is slowed down for a transparent rendering, we can still achieve good results, especially using anisotropic surface functions, just not at interactive frame rates; see Table 2.

Simulation data structure memory usage: Our data structure requires a hash map and a cell list, both containing a begin and length entry; see Section 4.4. The length of the cell list is determined by the number of particles, that is $\mathcal{O}(n_{\text{particles}})$, whereas the hash table size is determined as the smallest prime number larger than the number of particles. As this value is very close to the number of particles, we can assume that the hash map scales with the number of particles. Using bit-fields we require 4 Bytes per hash map entry and 8 Bytes per cell table entry. In contrast to many prior methods, the memory requirements only depend on the particle count, and not the simulation domain or particle distribution.

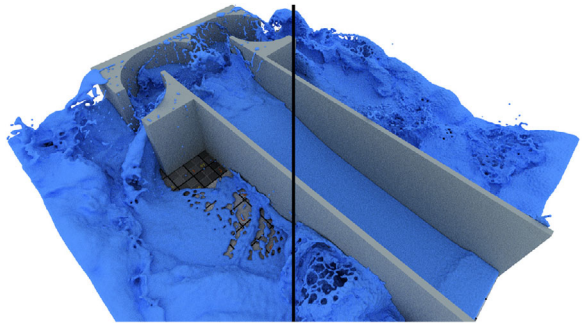

Figure 10: *Rendering of the* inlet *scene using an anisotropic (left) and an isotropic surface function (right).*

Rendering data structure memory usage: Our rendering data structure is similar to the simulation data structure and, as such, also requires 12 Bytes per particle, for a 32 Bit Morton Code. Overall, as the simulation data structure is required for rendering, the total cost per particle is 24 Bytes. However, ignoring memory requirements for the overall rendering algorithm, that is, for textures, these memory requirements only scale with the particle count, that is, $\mathcal{O}(n_{\text{particles}})$, and are independent of domain size, particle distribution or particle resolution, similar to the simulation data structure. Furthermore, we do not require any amount of memory to store an extracted mesh, nor an acceleration structure for the mesh, nor an intermediate grid used for an explicit surface extraction.

Limitations: Although our method yields very high quality surfaces at a low, and constant, memory cost, the computational requirements for the surface extraction can become very high, especially when using complex surface functions, that is, [YT13]. This performance cost becomes especially noticeable in transparent surface renderings, making them comparably slow. Although our approach is ready to render adaptive fluid surfaces [WHK17], the performance quickly degrades with higher adaptivity ratios, as applying the same approach as was done for the simulation (see Section 4.2) is not directly possible.

7. Conclusions

In this paper, we presented a data structure that can be used to efficiently simulate and render SPH fluid simulations using a combination of a cell table and hash map. The memory requirements of our approach do not scale with the domain size, particle resolution or particle distribution, but instead solely depend on the number of particles, making our approach well suited for usage on GPUs. Using our simulation data structure [WK19], we can efficiently simulate even highly adaptive SPH simulations in unbounded simulation domains. Using our rendering data structure, we can efficiently render an iso surface, without requiring an explicit surface extraction, using arbitrary surface metrics, that is, isotropic and anisotropic, using opaque or transparent shading models.

Acknowledgements

Open access funding enabled and organized by Projekt DEAL.

References

[AAOT13] Akinci G., Akinci N., Oswald E., Teschner M.: Adaptive surface reconstruction for sph using 3-level uniform grids. In *WSCG* (2013), Václav Skala-UNION Agency, pp. 195–204.

[AAT13] Akinci N., Akinci G., Teschner M.: Versatile surface tension and adhesion for SPH fluids. *ACM Transactions on Graphics 32*, *6* (2013), 182.

[AI08] Andoni A., Indyk P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM 51*, *1* (2008), 117.

[APKG07] Adams B., Pauly M., Keiser R., Guibas L. J.: Adaptively sampled particle fluids. In *SIGGRAPH '07: ACM SIGGRAPH 2007 Papers* (New York, NY, 2007), ACM Press, pp. 48–es.

[AW87] Amanatides, J., Woo, A.: A fast voxel traversal algorithm for ray tracing. *Eurographics 87*, (1987), 3–10.

[BK15] Bender J., Koschier D.: Divergence-free smoothed particle hydrodynamics. In *Proceedings of the 14th ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Los Angeles, CA, 2015), ACM Press, pp. 147–155.

[BKKW18] Bender J., Koschier D., Kugelstadt T., Weiler M.: Turbulent micropolar SPH fluids with foam. *IEEE Transactions on Visualization and Computer Graphics 25*, *6* (2018), 2284–2295.

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

[BT07] Becker M., Teschner M.: Weakly compressible SPH for free surface flows. *Proceedings of the 14th ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2007), pp. 209–217.

[BTS*17] Berger M., Tagliasacchi A., Seversky L. M., Alliez P., Guennebaud G., Levine J. A., Sharf A., Silva C. T.: A survey of surface reconstruction from point clouds. *Computer Graphics Forum 36* (2017), 301–329.

[DA12] Dehnen W., Aly H.: Improving convergence in smoothed particle hydrodynamics simulations without pairing instability. *Monthly Notices of the Royal Astronomical Society 425*, *2* (2012), 1068–1082.

[DC99] Desbrun M., Cani M.-P.: *Space-Time Adaptive Simulation of Highly Deformable Substances*. Research Report RR-3829, INRIA, 1999.

[DCG11] Domínguez J. M., Crespo A. J. C., Gómez-Gesteira M.: Optimization strategies for parallel CPU and GPU implementations of a meshfree particle method. Preprint, 2011, http://arxiv.org/abs/1110.3711.

[DCH88] Drebin R. A., Carpenter L., Hanrahan P.: Volume rendering. *ACM Siggraph Computer Graphics 22*, *4* (1988), 65–74.

[DCVB*13] Dominguez J. M., Crespo A. J., Valdez-Balderas D., Rogers B. D., Gomez-Gesteira M.: New multi-GPU implementation for smoothed particle hydrodynamics on heterogeneous clusters. *Computer Physics Communications 184*, *8* (2013), 1848–1860.

[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.

[FS05] Foley T., Sugerman J.: Kd-tree acceleration structures for a gpu raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics hardware* (New York, NY, 2005), ACM Press, pp. 15–22.

[GBP*17] Gissler C., Band S., Peer A., Ihmsen M., Teschner M.: Generalized drag force for particle-based simulations. *Computers & Graphics 69* (2017), 1–11.

[GLHL11] Garcia I., Lefebvre S., Hornus S., Lasram A.: Coherent parallel hashing. *ACM Transactions on Graphics 30*, (2011), 161.

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

[GPB*19] Gissler C., Peer A., Band S., Bender J., Teschner M.: Interlinked SPH pressure solvers for strong fluid-rigid coupling. *ACM Transactions on Graphics 38*, *1* (2019), 5.

[GPSS07] Gunther J., Popov S., Seidel H.-P., Slusallek P.: Realtime ray tracing on GPU with BVH-based packet traversal. In *2007 IEEE Symposium on Interactive Ray Tracing* (Piscataway, NJ, 2007), IEEE, pp. 113–118.

[Gre10] Green S.: Particle Simulation using CUDA. *Cuda 4.0 Sdk*, May (2010).

[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* (Madrid, Spain, 2010), Eurographics Association, pp. 55–64.

[HK89] Hernquist L., Katz N.: TREESPH - A unification of SPH with the hierarchical tree method. *Astrophysical Journal Supplement Series 70* (1989), 419.

[HS13] Horvath C. J., Solenthaler B.: *Mass Preserving Multi-Scale SPH*. Tech. Rep., Emeryville, CA, 2013.

[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.

[IOS*14] Ihmsen M., Orthmann J., Solenthaler B., Kolb A., Teschner M.: SPH Fluids in Computer Graphics. *Eurographics STARS*, *2* (2014), 21–42.

[KAD*06] Keiser R., Adams B., Dutré P., Guibas L., Pauly M.: *Multiresolution Particle-Based Fluids*. Tech. Rep. 520, Department of Computer Science, ETH Zurich, 2006.

[Kar12] Karras T.: Thinking parallel, part iii: Tree construction on the GPU, 2012. https://devblogs.nvidia.com/thinking-parallel-part-iii-tree-construction-gpu/. Accessed: 2019-06-17.

[KB17] Koschier D., Bender J.: Density maps for improved SPH boundary handling. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2017), ACM Press, pp. 1–10.

[KBST19] Koschier D., Bender J., Solenthaler B., Teschner M.: Smoothed particle hydrodynamics techniques for the physics based simulation of fluids and solids. In *Eurographics 2019 - Tutorials*. W. Jakob W. and E. Puppo (Eds.), The Eurographics Association, Aire-la-Ville, Switzerland, pp. 1–41.

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

[KW03] Kruger J., Westermann R.: Acceleration techniques for gpu-based volume rendering. In *IEEE Visualization 2003* (Piscataway, NJ, 2003), IEEE, pp. 287–292.

[LC87] Lorensen W. E., Cline H. E.: Marching cubes: A high resolution 3d surface construction algorithm. *ACM Siggraph Computer Graphics 21*, *4* (1987), 163–169.

[LGS*09] Lauterbach C., Garland M., Sengupta S., Luebke D., Manocha D.: Fast BVH construction on GPUS. *Computer Graphics Forum 28* (2009), 375–384.

[LH06] Lefebvre S., Hoppe H.: Perfect spatial hashing. *ACM Transactions on Graphics 25* (2006), 579–588.

[MCG03] Müller M., Charypar D., Gross M.: Particle-based fluid simulation for interactive applications. *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation 5*, (2003), 154–159.

[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 2013 Courses* (2013), ACM Press, p. 19.

[Mon05] Monaghan J. J.: Smoothed particle hydrodynamics. *Reports on Progress in Physics 68*, *8* (2005), 1703.

[Mor66] Morton G. M.: *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company, Armonk, New York, 1966.

[MSD07] Müller M., Schirm S., Duthaler S.: Screen space meshes. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2007), pp. 9–15.

[NJB07] Navratil P., Johnson J., Bromm V.: Visualization of cosmological particle-based datasets. *IEEE Transactions on Visualization and Computer Graphics 13*, *6* (2007), 1712–1718.

[OK12] Orthmann J., Kolb A.: Temporal blending for adaptive SPH. *Computer Graphics Forum 31* (2012), 2436–2449.

[OVSM98] Owen J. M., Villumsen J. V., Shapiro P. R., Martel H.: Adaptive smoothed particle hydrodynamics: Methodology. ii. *The Astrophysical Journal Supplement Series 116*, *2* (1998), 155.

[PJH16] Pharr M., Jakob W., Humphreys G.: *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, Burlington, MA, 2016.

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

[THM*03] Teschner M., Heidelberger B., Müller M., Pomerantes D., Gross M. H.: Optimized spatial hashing for collision detection of deformable objects. *Vmv 3*, (2003), 47–54.

[UHT17] Um K., Hu X., Thuerey N.: Perceptual evaluation of liquid simulation methods. *ACM Transactions on Graphics 36*, *4* (2017), 143.

[vdLGS09] van der Laan W. J., Green S., Sainz M.: Screen space fluid rendering with curvature flow. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games* (2009), pp. 91–98.

[WHK16] Winchenbach R., Hochstetter H., Kolb A.: Constrained neighbor lists for SPH-based fluid simulations. In *Proceedings of the 15th ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (July 2016).

[WHK17] Winchenbach R., Hochstetter H., Kolb A.: Infinite continuous adaptivity for incompressible SPH. *ACM Transactions on Graphics 36*, *4* (2017), 102:1–102:10.

[Win] Winchenbach R.: openmaelstrom, 2019–. URL: http://www.cg.informatik.uni-siegen.de/openMaelstrom. Accessed: 2020-02-28.

[WK19] Winchenbach R., Kolb A.: Multi-level-memory structures for adaptive SPH simulations. In *Vision, Modeling and Visualization* (2019), Schulz, H.-J., Teschner, M., Wimmer, M., (Eds.), The Eurographics Association, Aire-la-Ville, Switzerland. 99–107.

[WLS*17] Wu W., Li H., Su T., Liu H., Lv Z.: Gpu-accelerated SPH fluids surface reconstruction using two-level spatial uniform grids. *The Visual Computer 33*, *11* (2017), 1429–1442.

[WRR18] Winkler D., Rezavand M., Rauch W.: Neighbour lists for smoothed particle hydrodynamics on GPUS. *Computer Physics Communications 225* (2018), 140–148.

[WTYH18] Wu K., Truong N. P., Yuksel C., Hoetzlein R.: Fast fluid simulations with sparse volumes on the GPU. *Computer Graphics Forum 37*, *2* (2018), 157–167.

[XZY17] Xiao X., Zhang S., Yang X.: Real-time high-quality surface rendering for large scale particle-based fluids. In *Proceedings of the 21st ACM Siggraph Symposium on Interactive 3D Graphics and Games* (2017), pp. 1–8.

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

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

[ZPVBG01] Zwicker M., Pfister H., Van Baar J., Gross M.: Surface splatting. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques* (2001), pp. 371–378.

Supporting Information

Additional supporting information may be found online in the Supporting Information section at the end of the article.

Data Video S1

Data S2