Vision, Modeling, and Visualization (2019)
H.-J. Schulz, M. Teschner, and M. Wimmer (Eds.)

Multi-Level-Memory structures for adaptive SPH simulations

Rene Winchenbach[1], Andreas Kolb[1]

[1]University of Siegen

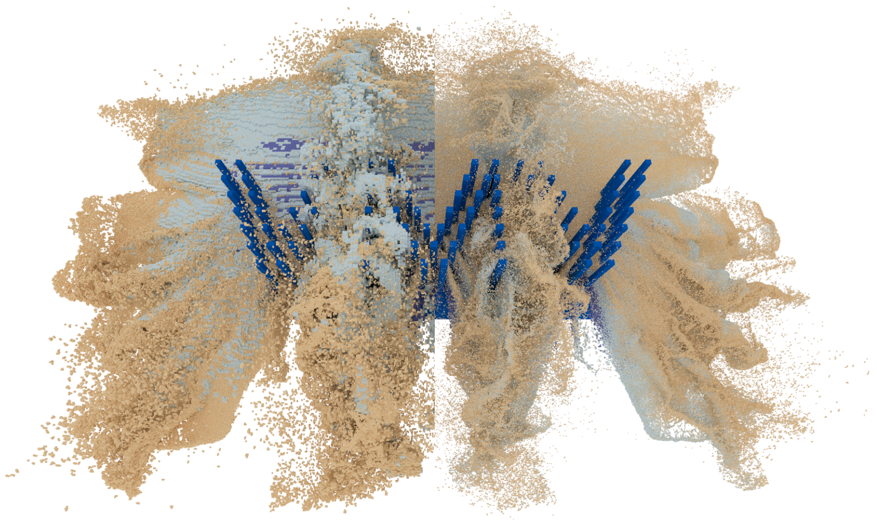

Figure 1: *Using our multi level memory structure (left image half), we can efficiently simulate a highly adaptive (1000:1) SPH simulation, for up to 8 million particles (right image half), using 4 memory levels. At the pictured timepoint the level 0 domain spans* $189 \times 124 \times 84$ *cells* $(1,968,624)$*, whereas the level 3 domain spans* 8^3 *as many cells . Color coding indicates memory level from* 0 *(blue) to* 3 *(red).*

Abstract

In this paper we introduce a novel hash map-based sparse data structure for highly adaptive Smoothed Particle Hydrodynamics (SPH) simulations on GPUs. Our multi-level-memory structure is based on stacking multiple independent data structures, which can be created efficiently from the same particle data by utilizing self-similar particle orderings. Furthermore, we propose three neighbor list algorithms that improve performance, or significantly reduce memory requirements, when compared to Verlet-lists for the overall simulation. Overall, our proposed method significantly improves the performance of spatially adaptive methods, allows for the simulation of unbounded domains and reduces memory requirements without interfering with the simulation.

CCS Concepts

• ***Computing methodologies*** $\rightarrow$ ***Massively parallel and high-performance simulations;*** *Physical simulation;*

1. Introduction

Vivid and highly detailed fluid simulations have become an essential part of modern computer graphics, due to ever increasing demands for more realism. Smoothed Particle Hydrodynamics (SPH) [GM77] is a simulation technique for fluid systems, which has been extended recently to allow for highly adaptive incompressible fluid simulations [WHK17]. Spatially adaptive methods dedicate computational resources where they are most beneficial to the desired outcome. However, adaptive simulations with adaptivity ratios of $1000:1$ and higher suffer from significant performance drops due to limitations in the underlying data structures [WHK17]. For CPU based, single resolution SPH simulation methods compact hash maps are commonly used [IABT11]. GPU based methods cannot easily utilize these approaches and instead often rely on dense cell structures [Gre10; GSSP10] or linked list based structures [DCV*13; WRR18].

DOI: 10.2312/vmv.20191323

In this paper, we present a hash map based data structure, which is specifically designed to handle the requirements of highly adaptive SPH methods simulated on a GPU. 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-neighbor particle accesses by providing an appropriate data structure for different particle resolutions. Furthermore, we present a corrective algorithm that guarantees symmetric particle neighborhoods, which are essential for spatially adaptive incompressible fluid simulations. Additionally, we propose a set of novel neighbor-list algorithms, which are applicable to adaptive and non-adaptive simulations, by either improving performance or memory consumption. Our proposed method significantly improves the practical applicability of adaptive simulations, and substantially reduces the data structure overhead. Our proposed method provides better memory scaling and allows for the simulation of unbounded domains.

2. Related Work

SPH has been a very active field of research since its introduction by Gingold and Monaghan [GM77]. Initially, stiff equations of state were employed to achieve simulations of weakly compressible fluids [MCG03; BT07]. Later techniques are based on prediction-correction [SP09], iterative [MM13], and implicit [ICS*13] methods in order to enforce incompressibility. In addition to solving incompressibility, divergence-free SPH simulations (DFSPH) have been demonstrated [BK15], which significantly stabilize the overall simulation and improve visual fidelity. Recent research has also made large improvements to boundary handling, either by utilizing particles [BGPT18; BGI*18; GPB*19], analytical [FM15] or numerical [KB17] boundary models.

Adaptive simulations using splitting and merging processes were introduced by Desbrun and Cani [DC99]. This work was extended by adjusting particle positions after splitting, in order to reduce the error in the pressure term [APKG07]. To further stabilize the interaction between particles with different resolutions, Keiser et al. [KAD*06] used virtual link particles of neighboring resolutions. To realize adaptivity in incompressible methods, Winchenbach et al. [WHK17] introduce an adaptive method, which works with estimates of original particle positions, a temporal blending process, similar to that proposed by Orthmann and Kolb [OK12], and a process of mass redistribution. This allows for much larger adaptivity ratios than previously possible, in excess of $10,000:1$. However, the performance benefits of higher adaptivity ratios are significantly hampered by the limitations of existing data structures.

For CPU based simulations, Ihmsen et al. [IOS*14] give a good overview of existing data structure methods, 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], which was optimized for multiple GPUs. Goswami et al. [GSSP10] used Morton codes , however, their approach introduces a complex scheme to balance workloads on the GPU, 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 neighboring particles. However, all of these methods suffer from scaling and performance problems for adaptive simulations.

Many generic data structures and methods have been developed, for computer animation, where some notable examples include perfect hash maps to store sparse voxel data [LH06; GLHL11], which are not easily scalable to multiple resolutions or approximate nearest neighbors from machine learning aspects [AI08], which are only approximate and designed for high dimensional data. Furthermore, various CPU based approaches exist, e.g. 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 different requirements on the data structure, which is an integral part of the simulation itself.

3. Basics of Smoothed Particle Hydrodynamics

In general, quantities for a particle i are interpolated from a weighted average using neighboring particles j as [Mon05]

$$A_i = \sum_j A_j \frac{m_j}{\rho_j} W_{ij}, \tag{1}$$

where the interpolated quantity is denoted as $A_i = A(\boldsymbol{x}_i)$, which depends on the mass m_j and density ρ_j of neighboring particles within a compact support radius. For further details refer to [Pri12]. The contributions from these neighboring particles are then weighted based on a kernel function $W_{ij} = W(x_{ij}, h_{ij})$, $x_{ij} = \left\|\boldsymbol{x}_i - \boldsymbol{x}_j\right\|$, which in turn is based on the support radius of an interaction h_{ij}. For adaptive incompressible methods, $h_{ij} = \frac{h_i + h_j}{2}$ is used in order to avoid instabilities. The support radius of a particle can be calculated as

$$h_i = \eta \sqrt[3]{\frac{m_i}{\rho_i}}. \tag{2}$$

Here, η is a configuration parameter set based on the chosen kernel function [DA12; WHK16], which determines the number of neighboring particles in a resting state N_h and as such can be found by refactoring of $\frac{4}{3}\pi h^3 = N_h \frac{m_i}{\rho_0}$ [WHK16]. Other kernel functions, e.g. those of the Wendland family, have much larger N_h values, and therefore require different neighbor list algorithms to prevent excessive memory usage. In computer animation the support radius is often calculated as

$$h_i^0 = \eta \sqrt[3]{V_i^0}, \tag{3}$$

which is based on the rest volume of a particle V_i^0 instead. The rest volume of a particle solely relies on the particles physical size, i.e. $V_i^0 = \frac{4}{3}\pi r^3$ for some radius r, and does not change based on changes in density. As such we denote this support radius as h_i^0. This is equivalent to assuming that $\rho_i = \rho_0$ in (2).

4. Data Structures

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 $\mathcal{O}(n \cdot m)$. One possible approach is to divide the simulation domain into uniform cells of size h [Gre10; IOS*14]. 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) neighboring particles. The sphere described by the support radius of a particle will, on average, contain N_h neighbors [DA12] within a volume of $\frac{4}{3}\pi h^3$, whereas the sub-grid of all potential neighbors has a volume of $27h^3$. This means that the sub-grid will contain, on average, $\frac{81}{4\pi}N_h \approx 6.5N_h$ particles, i.e. 15.5% of all potential neighbors are actual neighbors. For an adaptive ratio of $1000:1$, however, only 0.016% of all considered particles are neighbors as a cell of the same size would now contain $\frac{81000}{4\pi}N_h$ particles, causing significant performance problems [WHK17]. We first introduce our general data structure for non-adaptive simulations in Sec. 4.1, and then the changes required for adaptive simulations in Sec. 4.2.

4.1. Single-Level Data Structure

We chose the cell size $C_{\max}$ to be the same as the largest support radius of any particle as this ensures that all neighbors of all particles are contained within a $3 \times 3 \times 3$ sub-grid, which would not be possible for an arbitrary cell size. We can calculate $C_{\max}$ efficiently by using a reduction operation over all particle support radii h_i as

$$C_{\max} = \max\{h_0, ..., h_{n-1}\}. \tag{4}$$

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

$$\boldsymbol{D}_{\min} = \min\{\boldsymbol{x}_0, ..., \boldsymbol{x}_{n-1}\}, \boldsymbol{D}_{\max} = \max\{\boldsymbol{x}_0, ..., \boldsymbol{x}_{n-1}\}. \tag{5}$$

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

$$\boldsymbol{D} = \left\lceil \frac{\boldsymbol{D}_{\max} - \boldsymbol{D}_{\min}}{C_{\max}} \right\rceil. \tag{6}$$

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 can 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 $C_{\max}$ as

$$\bar{\boldsymbol{x}} = \left\lfloor \frac{\boldsymbol{x} - \boldsymbol{D}_{\min}}{C_{\max}} \right\rfloor. \tag{7}$$

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

$$\mathcal{L}(\bar{\boldsymbol{x}}) = \bar{\boldsymbol{x}}_x + \boldsymbol{D}_x\left(\bar{\boldsymbol{x}}_y + \boldsymbol{D}_y\left(\bar{\boldsymbol{x}}_z\right)\right), \tag{8}$$

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, also sometimes referred to as the Z-ordering, is an alternative indexing scheme, which describes a self-similar space-filling curve. We can efficiently determine $\mathcal{Z}(\bar{\boldsymbol{x}})$ by interleaving the binary representation of an integer coordinates as

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

where the superscript denotes a specific bit. Using a 32 bit Morton code results in 10 bit per dimension, meaning each dimension can contain a maximum of $\#K = 1024$ cells. A 64 bit Morton code results in 21 bit per dimension, meaning a maximum of $\#K = 2097152$ 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. For SPH simulations many nodes of an octree, e.g. the root node, do not contain any useful information and furthermore, traversing an octree is computationally 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.

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. To generate this list, 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

$$\mathbb{C}[i] = \begin{cases} i & \text{, if } i = 0 \vee \mathcal{Z}_i \neq \mathcal{Z}_{i-1} \\ -1 & \text{, if } \mathcal{Z}_i = \mathcal{Z}_{i-1} \\ n_{\text{particles}} & \text{, else.} \end{cases} \tag{9}$$

$\mathbb{C}$ 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 $\mathbb{C}_{\text{compact}}^{\text{begin}}$ 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

$$\mathbb{C}_{\text{compact}}^{\text{length}}[i] = \mathbb{C}_{\text{compact}}^{\text{begin}}[i+1] - \mathbb{C}_{\text{compact}}^{\text{begin}}[i]. \tag{10}$$

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 $\mathbb{C}_{\text{compact}}^{\text{begin}}$ and $\mathbb{C}_{\text{compact}}^{\text{length}}$. Following Ihmsen et al. [IABT11], we determine the hash of an integer coordinate by using three large prime numbers $p_1 = 73856093$, $p_2 = 19349663$, $p_3 = 83492791$ and the size of the hash table n_{hash} as

$$\mathcal{H}(\bar{\boldsymbol{x}}) = \left(p_1\bar{\boldsymbol{x}}_x + p_2\bar{\boldsymbol{x}}_y + p_3\bar{\boldsymbol{x}}_z\right) \% n_{\text{hash}}, \tag{11}$$

where we choose n_{hash} as the smallest prime number larger than the maximum number of particles in a simulation, as this gives a relatively sparse hash map with few collisions, in general.

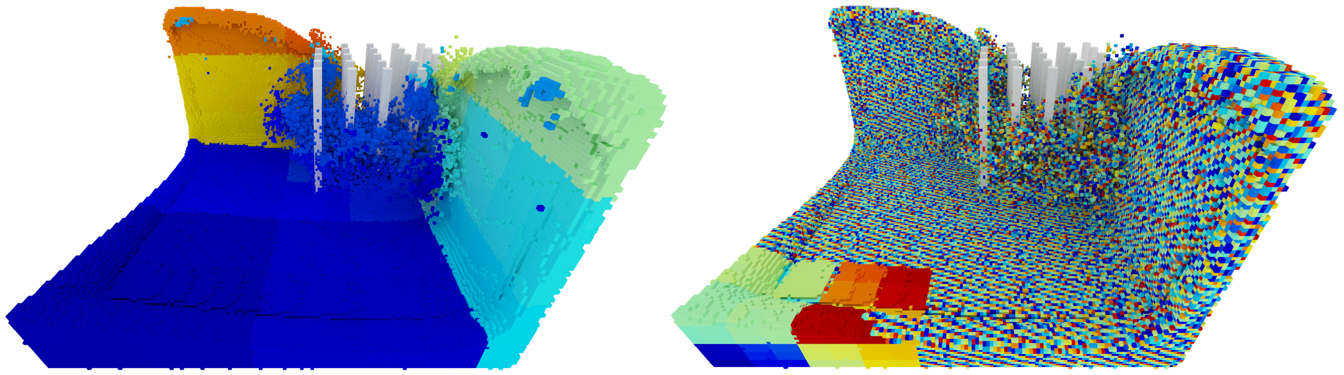

Figure 2: *These two images show the Morton code $\mathcal{Z}$ on the left and the hashed indices $\mathcal{H}$ on the right for every occupied cell, with color coding indicating indices. The Morton code gives much greater spatial locality but would lead to a significant number of collisions.*

However, this hash function leads to low coherency for nearby particles, meaning that particles which are spatially close, might be assigned to very distant memory locations, see Fig. 2, right. If we can avoid hash collisions, we can embed the cell information directly into the hash map, which removes a level of indirection. However, if we chose a simple spatially coherent hash function, e.g. $\mathcal{H}(\bar{\mathbf{x}}) = \mathcal{Z}(\bar{\mathbf{x}})\% n_{\text{hash}}$, we would find significantly more hash collisions, which outweighs the benefit of more coherence. Contrarily, choosing a perfect hash function, e.g. [LH06], reduces the number of collisions, but at the cost of a significant overhead for its creation. Therefore, we opt for the hash function of Ihmsen et al. [IABT11] as it provides a good balance between complexity and collisions. Furthermore, as we utilize per-particle neighbor-lists, we need to access the cells only once during each timestep, as all particle interactions afterwards are calculated using these neighbor-lists avoiding the spatial incoherence of the function.

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 $\mathcal{H}(\bar{\mathbf{x}})$. The process required to find a specific cell c based on the cells integer coordinates $\bar{\mathbf{x}}_c$ is described in Algorithm 1.

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

$$\mathbb{H}^{\text{begin}}[\mathcal{H}_i] = i, \text{ if } i = 0 \vee \mathcal{H}_i \neq \mathcal{H}_{i-1}, \tag{12}$$

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

$$\mathbb{H}^{\text{length}}[\mathcal{H}_i] = i - \mathbb{H}^{\text{begin}}[\mathcal{H}_i] - 1, \text{ if } i = n_{\text{occupied}} \vee \mathcal{H}_i \neq \mathcal{H}_{i+1} \tag{13}$$

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 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 $\mathcal{Z}_c = \mathcal{Z}_j$ to avoid returning a wrong cell.

```
Calculate $\bar{\mathbf{x}}_c$ for the cell we are looking for
Calculate $\mathcal{Z}_c$ and $\mathcal{H}_c$ for $\bar{\mathbf{x}}_c$
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 $\bar{\mathbf{x}}_j$ 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 $C_{\max}$, $\mathbf{D}_{\min}$ 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 uniform cell sizes, which would suffer from the same problems for adaptive simulations as prior methods, due to a mismatch of cell size and particle resolution. However, as we based our method on a Morton code, we can utilize the self-similarity to efficiently create multiple, distinct, data structures for different cell sizes on the same underlying particle data. This is possible for, coarser, power of 2 multiple cell sizes of the cell size used for re-sorting the data.

We start with an initially much finer particle sorting $\mathcal{Z}^{\text{fine}}$, from which we can generate the desired coarser resolutions. To determine $\mathcal{Z}^{\text{fine}}$ we calculate the corresponding cell size C_{fine}, based on the largest dimension $P = \max(\boldsymbol{D}_x, \boldsymbol{D}_y, \boldsymbol{D}_z)$, as

$$C_{\text{fine}} = C_{\max} \frac{2^{\lceil \log_2(P) \rceil}}{\#K}. \tag{14}$$

Here, $\#K$ depends on the size of the Morton code used (see Sec. 4.1), C_{fine} is the smallest cell size that can be represented using this code length, and $2^{\lceil \log_2(P) \rceil}/\#K = 2^{-L_{\text{fine}}}$, $L_{\text{fine}} \in \mathbb{N}$ is the refinement factor. The algorithm described in the Sec. 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

$$\bar{\boldsymbol{x}}^L = \left\lfloor \frac{\boldsymbol{x} - \boldsymbol{x}_{\min}}{C_{\max} \cdot 2^{-L}} \right\rfloor. \tag{15}$$

$L = 0$ 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

$$L_{\max} = \min\left\{ \left\lceil log_2 \sqrt[3]{\alpha} \right\rceil, L_{\text{fine}} \right\}, \tag{16}$$

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 (11) as

$$\mathcal{Z}^L(\bar{\boldsymbol{x}}) = L n_{\text{hash}} + \left(p_1 \bar{\boldsymbol{x}}_x^L + p_2 \bar{\boldsymbol{x}}_y^L + p_3 \bar{\boldsymbol{x}}_z^L \right) \% n_{\text{hash}}. \tag{17}$$

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

$$L_i = \text{clamp}\left(\left\lfloor -\log_2 \frac{h_i}{C_{\max}} \right\rfloor, 0, L_{\text{fine}} - 1 \right). \tag{18}$$

Thus, every particle can easily access all neighbors at any scale $L \in [0, L_{\max}]$ using the data structure for L. However, when a particle i only looks for neighbors at its level L_i, we may encounter asymmetric interactions, as neighbor searches are limited by the cell size for level L_i; see Fig. 3. This could be avoided entirely by utilizing a gather-formulation of SPH, but this formulation is not stable for adaptive incompressible SPH [WHK17]. Therefore, we explicitly need to handle this case.

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, i.e. if $L_k > L_i \wedge x_{ik} < h_{ik}$. In order to resolve the asymmetry, we iterate over all neighboring 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

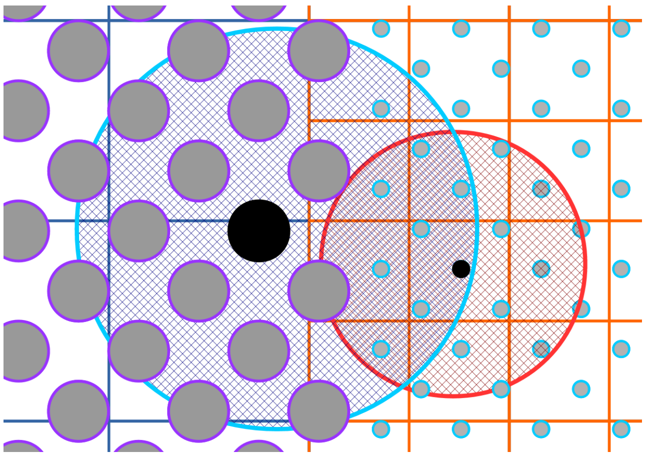

Figure 3: *Asymmetry interaction: Two particles at different levels may not mutually see each other due to the different cell size. Here, the lower level particle to the left sees the higher level particle to the right, but not vice versa.*

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

```
Initialize
    Calculate $C_{\max}$, $\mathbf{D}_{\min}$ 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
```

$$\left\| \bar{\boldsymbol{x}}_{ik}^{L_k} \right\|_1 = \max(|\bar{\boldsymbol{x}}_{ik,x}^{L_k}|, |\bar{\boldsymbol{x}}_{ik,y}^{L_k}|, |\bar{\boldsymbol{x}}_{ik,z}^{L_k}|) > 1. \tag{19}$$

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.

5. Neighbor-lists

As noted in Section 4, only about 15.5% of all potentially neighboring particles are actual neighbors. To avoid the repeated access to non-neighboring particles, neighbor-lists are a common solution, which store a reduced set of potential neighbors. Verlet-lists store references to every actual neighbor, but for adaptive simulations where the number of neighbors often exceeds $2N_h$ this would lead to excessive memory usage. In order to avoid this, we first introduce a novel histogram based constrained neighbor-list in Section 5.1, followed by a span based neighbor-list in Section 5.2 and a bitmask based neighbor-list in Section 5.3.

5.1. Histogram Based Neighbor-lists

A constrained neighbor list, e.g. [WHK16], limits the number of neighbors N_i for a particle i to an upper bound N_c, where the main motivation comes from reducing memory requirements and optimizing access patterns and is not based around a change of neighborhood size due to a changing support radius, e.g. by using (2).

Naïvely, it would be possible to simply exclude actual neighbors from this list, however this would lead to asymmetries and thus to instabilities. To avoid this, a constrained neighbor-list method reduces the support radius of a particle h_i until $N_i < N_c$. The constrained neighbor-list approach of Winchenbach et al. [WHK16] implemented this in an iterative process, however, due to the cost of an iteration over all potential neighbors, this method became computationally expensive for adaptive methods. In order to realize this in a single step, we first consider the support radius for i in an interaction with another particle j that would result in $|x_{ij}| = h_{ij}$, and as such $W_{ij} = 0$. We can determine this support radius for each interacting pair of particles as

$$k_{ij} = 2|\mathbf{x}_{ij}| - h_j, \tag{20}$$

where the constrained support radius h_i^c would trivially be the N_c-th smallest value. However, calculating k_{ij} for all potentially neighboring particles and storing this list to find the N_c-th smallest value is not practical and we instead propose an alternative histogram-based approach. We create a histogram, whose bins evenly segment the range $[0.5h_i, h_i)$ and store the number of particles with

$$\frac{1}{2}h_i + \frac{B}{2\#B}h_i \leq k_{ij} < \frac{1}{2}h_i + \frac{B+1}{2\#B}h_i \tag{21}$$

in each bin, where B denotes the bin index and $\#B$ the number of bins for the histogram. This allows us to calculate the bin a particle j belongs to in the histogram for particle i, as

$$\#b_{ij} = \left\lfloor \text{clamp}\left(\frac{k_{ij} - 0.5h_i^{\text{new}}}{2\#B \cdot h_i^{\text{new}}}, 0, \#B - 1\right)\right\rfloor. \tag{22}$$

We can store this histogram within the shared memory of a GPU by choosing a small enough bin size and number of bins, e.g. 32 bins with an 8 bit bin size on modern GPUs. Each bin contains the number of particles associated with this range of values and as such, once the histogram is completed, we can sum up the counters, starting with the lowest bin, until the sum is larger than the upper bound of neighbors N_c at some bin index b. The final constrained support radius can then be calculated as

$$h_i^c = \frac{h_i}{2}\left[1 + \frac{1}{b-1}\right]. \tag{23}$$

In order to avoid an ever decreasing support radius, as constraining can only reduce h_i, we propose to update h_i at the end of every timestep based on the rest support radius (3) as

$$h_i(t + \Delta t) = \alpha h_i(t) + (1 - \alpha)h_i^0, \tag{24}$$

where α is a linear blend weight, usually chosen as 0.95. In general, this neighbor-list is still, conceptually, a Verlet-list and requires more than N_h entries as we cannot reduce N_i below N_h without causing instabilities. For larger kernel functions, i.e. Wendland kernels, which have very large neighborhoods and adaptive simulations this becomes quite memory consuming.

5.2. Span Based Neighbor-Lists

Instead of storing explicit references to all actual neighbor particles, we store appropriate index-spans in order to be more memory efficient at the cost of covering more non-interacting neighbor particles. Our general approach is to store one index span for each of the 27 neighboring cells per particle.

For any neighboring cell c we iterate over the contained particles $j \in c$ in ascending order, and store the first index b where $|x_{ib}| < h_{ib}$. We then keep iterating until we find the last index l where $|x_{il}| < h_{il}$, which gives the span of particle indices $j \in [b, l]$ that contains all neighbors of i in c. We store b as well as the length of this span $s = l - b + 1$.

The memory requirement for storing a span is $\text{size}(b) = \lceil \log_2 n_{\text{particles}} \rceil$ and $\text{size}(s) \propto \left\lceil \log_2 N_h \frac{3}{4\pi} \right\rceil$. For non-adaptive simulations $\text{size}(b) + \text{size}(s)$ is almost always less than 4 byte, however for adaptive simulations the number of particles in a cell can become much larger and we require 8 byte in this case. This is a significant improvement in memory efficiency for non-adaptive and adaptive simulations utilizing kernel functions with $N_h > 27$, as the memory consumption results to $27 \cdot 4$ or $27 \cdot 8$ byte instead of $N_h \cdot 4$ and $2N_h \cdot 4$ byte for the non-adaptive and the adaptive case, respectively.

5.3. Bitmask Based Neighbor Lists

The previously described span based neighbor list requires 8 bytes for adaptive simulations, even though 4 bytes would be sufficient in regions with rather homogeneous support radii. Therefore, we propose a third approach that stores bitmask indicating neighboring particles for cells that have interacting neighbors using 4 byte only, accepting that particles in regions with rather inhomogeneous support radii need additional handling.

Considering the sub-grid of $3 \times 3 \times 3$ cells, we can find a unique mapping from this sub-grid to a linear index $\mathcal{L} \in [0, 27)$, which can be stored in 5 bits, leaving 27 bits for representing the bitmask. In homogeneous regions, a cell contains $N_h \frac{3}{4\pi}$ particles, i.e. 12 for the cubic spline kernel. The 26 bits indicate, if the corresponding particle is an interacting particle. To handle cells with more than 26 neighbors in regions with strongly varying support radii, we have to process all particles in the neighboring cell. We indicate this by setting the full bitmask to 1.

Using this masked neighbor list guarantees a memory consumption of $27 \cdot 4$ byte. The drawback is that cells with more than 26 particles, e.g. in rather turbulent regions with strongly varying support radii, are handled rather inefficiently. However, even in the worst case this is not worse than using no neighbor list at all.

6. Results and Discussion

All simulations were run using a single Nvidia RTX 2080 Ti GPU with 11 GiB of VRAM, an Intel i7-4790 and 16 GiB of RAM. We used DFSPH [BK15] with a density error limit of 0.01% and a divergence error limit of 0.1%, with rigid objects represented as density maps [KB17]. Artificial viscosity was modeled based on XSPH [Mon05], Surface tension was modeled based on [AAT13],

Methods		Overall /ms		Re-sorting /ms		Neighbor-list /ms		Density /ms		DFSPH /ms		Memory /GiB	
Structure	Neigh-list	single	adapt	single	adapt	single	adapt	single	adapt	single	adapt	single	adapt
						Pillar scene							
[Gre10]	[WHK16]	278	2557	2.70	7.67	13.54	2067.16	1.80	1.89	216.8	316.8	1.13	6.96
Ours	[WHK16]	274	572	7.74	19.71	17.89	45.72	1.74	1.78	196.0	279.5	1.20	7.54
Embed	[WHK16]	271	543	9.76	24.78	15.67	38.77	1.72	1.76	195.2	276.9	1.20	7.54
Embed	<none>	412	1782	10.32	21.52	–	–	1.85	2.24	328.9	596.4	0.68	3.59
Embed	Histogram	253	527	9.74	22.06	12.35	26.73	2.01	1.83	196.4	285.1	1.20	7.54
Embed	Spans	291	596	10.62	21.80	7.48	28.12	0.59	3.28	222.5	302.9	1.13	5.25
Embed	Bitmask	274	866	9.97	22.39	7.80	36.41	0.67	10.64	212.9	436.9	0.91	4.42
						Dragon scene							
[Gre10]	[WHK16]	86	3185	2.40	4.34	4.71	1637.74	2.17	2.48	64.0	1342.4	0.27	7.24
Ours	[WHK16]	85	1365	7.03	11.02	6.19	35.86	2.11	2.34	57.6	1187.9	0.29	7.84
Embed	[WHK16]	84	1340	8.74	13.86	5.34	30.74	2.08	2.31	57.4	1181.8	0.29	7.84
Embed	<none>	129	4402	9.24	12.04	–	–	2.24	2.95	96.4	2547.9	0.16	3.73
Embed	Histogram	84	1302	8.72	12.43	4.37	21.93	2.43	2.43	57.7	1217.3	0.29	7.84
Embed	Spans	93	1472	9.42	12.81	2.53	22.58	0.72	4.31	65.2	1294.6	0.27	5.46
Embed	Bitmask	88	2136	8.90	12.58	2.65	28.88	0.81	14.01	62.6	1867.6	0.22	4.59

Table 1: *The values shown here are given as the average value over 30 simulated seconds. Ours refers to the data structure presented in Sec. 4, with embed referring to the optimization of embedding $\mathbb{C}$ into $\mathbb{H}$. For the neighbor lists see Sec. 5.1 for the histogram based variant, Sec. 5.2 for the span based variant and Sec. 5.3 for the bitmask based variant. Single refers to a non-adaptive simulation and adapt refers to a simulation with an adaptive ratio of* $1000:1$.

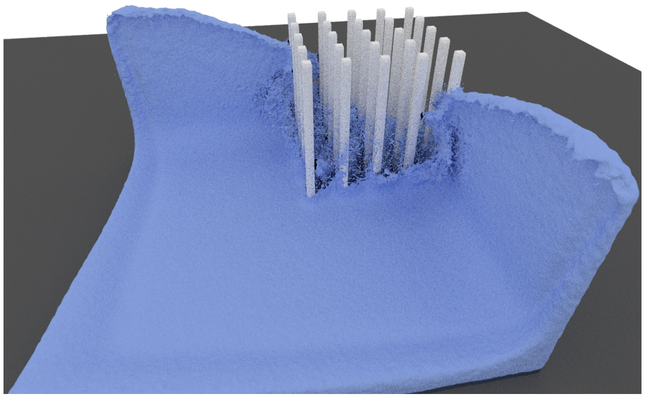

Figure 4: *A dam break scenario where the fluid was initialized opposite of the rigid obstacles.*

fluid air phase interactions were modeled based on [GBP*17]. For adaptivity we utilize [WHK17]. Rendering was done using a proprietary renderer, with surface extraction based on [YT13]. We used the cubic spline kernel for all tests. The full source code of our simulation, and renderer, can be found `www.cg.informatik.uni-siegen.de/openMaelstrom`.

Test Scenes: We evaluated our approach using two text scenes. The *Pillars* test scene is a dambreak, depicted in Figure 4, where an initial fluid volume interacts with many small rigid obstacles on impact. Here, the simulation domain spans 153^3 cells. In this scene we used 2 million particles for the non adaptive tests and up to 8 million particles in the adaptive tests. The *Dragon* test scene is a dambreak scenario, depicted in Figure 5, where an initial fluid volume interacts with a single complex rigid object. Here, the simulation domain spans $27 \times 49 \times 55$ cells. In this scene we used 400 thousand particles for the non adaptive tests and up to 8 million particles in the adaptive tests. For numerical stability we set $N_c = 1.2N_h$ for non-adaptive scenes and $N_c = 2.3N_h$ for adaptive scenes. We intentionally selected the adaptivity ratio to be rather moderate, i.e. we used 1,000:1, since we observed an extreme performance drop of several orders of magnitude for Green's method [Gre10] for higher adaptivity ratios such as 100,000:1, since this method was not designed for adaptive simulations. Additionally, Figure 5 (bottom right) shows a uniform simulation of comparable computational cost to the highly adaptive simulation with significantly lower visual fidelity, i.e. the larger particles cannot move in-between the body parts of the dragon resulting in lower visual fidelity.

Non-adaptive data-structure performance: Comparing our proposed sparse data structure with a dense data structure based on [Gre10] we can see a slight overall increase in performance (see Tab. 6, structure "Ours" and structure "[Gre10]" for single). Due to the more ideal Morton code for particle ordering, instead of a linear ordering, we can observe an increase in performance for SPH operations. However, the re-sorting process is slowed down, as well as accessing the data structure for the construction of the neighbor-list. Embedding $\mathbb{C}$ into $\mathbb{H}$ when no hash collision occurred reduced this overhead slightly (see Tab. 6, structure "Embed"). The memory consumption is slightly higher as the dense structure requires $2 \cdot 153^3$ entries, whereas our structure required $2 \cdot 10^6$ entries and additional temporary arrays for construction.

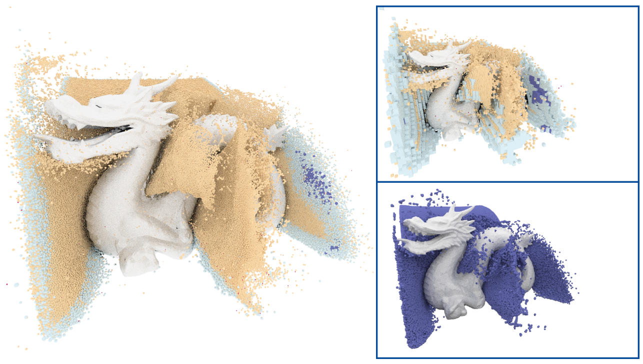

Figure 5: *This image shows the dragon test scene where a fluid volume collides with a complex rigid object. Color coding indicates the memory levels (blue to red). The top right view visualizes the data structure cells. The bottom right view is a uniform resolution simulation with a comparable time per timestep as the adaptive variant at this timepoint.*

Adaptive data-structure performance: Looking at the adaptive results (see Tab. 6, structure "Ours" and structure "[Gre10]" for adapt), we can now see a significant overall difference in performance where our data structure reduces the simulation time by nearly 60%. This is mainly due to the neighbor list construction that takes about 50 times longer, requiring about half of the overall computation time per frame when using Green's method [Gre10]. We can also observe increased performance of the SPH operations, where the largest improvement is for DFSPH with a speed up of about 12% due to the improved particle ordering. However, our multiple data structure requires more memory than a dense linear structure, due to now having to store multiple data structures, however the increase is moderate at some 6-8%.

Non-adaptive Neighbor list performance: When comparing the neighbor-lists presented in Sec. 5, we observe significantly different performance between the different neighbor-list approaches for the non adaptive *Dragon* scene, if compared to the prior constrained list approach of [WHK17]. Our proposed constrained list performs almost identical with identical memory consumption. Utilizing our proposed bit mask approach, we observe a slightly slower simulation as DFSPH takes slightly longer. However the density approximation was significantly faster as the neighbor-list is significantly smaller, which benefits a simple operation more than a complex one. We see a similar effect for the span based approach, but an overall slightly lower performance. Not using a neighbor-list requires significantly less memory, and slows the overall simulation down by about 50%, with much slower complex operations.

Adaptive Neighbor list performance: Considering the adaptive *Pillars* scene, we see a different ordering of the methods. Our proposed constrained list has a slightly improved performance compared to the prior approach, which is mostly due to the faster neighbor list construction. The bitmask based approach is now significantly slower (64% overall) but requires only 59% of the memory. The span based approach performs somewhat better, i.e. it is 13% slower but requires only 70% of the memory. Compared to [WHK17], not using a neighbor list requires only 47% of memory, but it is 237% slower.

Memory consumption: Overall, non-adaptive simulations require significantly less memory, as they require less information per particle, and as such we can simulate up to 35.5 million particles, without utilizing a neighbor-list. Using our bitmask approach, we can still simulate about 25 million particles, and using the span based and constrained methods we are able to simulate about 19 million particles. Considering the relatively low impact of the bitmask approach for non-adaptive methods, this results in an increase of 32% for the maximum number of particles, when compared to prior constrained neighbor-list approach [WHK16] without a significant drop in performance. For adaptive simulations we can simulate up to 23.5 million particles, without utilizing a neighbor-list, but the performance of this approach is too low. Using our bitmask approach we can simulate about 19 million particles, and using the span based approach about 16 million particles. Using a constrained list would only allow us to simulate about 10 million particles, which, due to the relatively low overhead of the span based approach, means that we can increase the maximum number of particles by about 60% by using our span based neighbor list, when compared to the prior constrained neighbor-list approach [WHK16]. In summary, none of the neighbor list approaches is superior to the others, i.e. none offers the highest performance at the lowest memory consumption.

Limitations: For non-adaptive simulations our proposed data-structure only offers a very minor increase in performance, due to the better memory layout, at the cost of a slightly higher memory consumption. This increased cost, however, is only required for relatively small and bounded domains and as such not a problem in general. For adaptive-methods, we often have to adjust the resolution of particles to avoid asymmetries and, thus, severe instabilities, which reduces the overall potential performance gain. In general, smoother resolution gradients are less affected by asymmetries and allow for larger speed-ups.

7. Conclusions

Our contributions allow us to efficiently simulate highly adaptive simulations, with adaptive ratios beyond $1,000 : 1$, without causing performance limitations due to the underlying data structuring. Using our data structure allows us to simulate unbounded domains, where the memory consumption only scales with particle count, not resolution. In addition, by using our propose neighbor list methods we can further improve performance, or significantly reduce memory consumption allowing for higher particle counts. In the future we would like to expand our work on data structures to multi GPU systems for even larger simulations.

References

[AAT13] AKINCI, NADIR, AKINCI, GIZEM, and TESCHNER, MATTHIAS. "Versatile surface tension and adhesion for SPH fluids". *ACM Transactions on Graphics (TOG)* 32.6 (2013), 182 6.

[AI08] ANDONI, ALEXANDR and INDYK, PIOTR. "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions". *Communications of the ACM* 51.1 (2008), 117 2.

[APKG07] Adams, Bart, Pauly, Mark, Keiser, Richard, and Guibas, Leonidas J. "Adaptively sampled particle fluids". *ACM Transactions on Graphics (TOG)*. Vol. 26. 3. Acm. 2007, 48 2.

[BGI*18] Band, Stefan, Gissler, Christoph, Ihmsen, Markus, et al. "Pressure boundaries for implicit incompressible SPH". *ACM Transactions on Graphics (TOG)* 37.2 (2018), 14 2.

[BGPT18] Band, Stefan, Gissler, Christoph, Peer, Andreas, and Teschner, Matthias. "MLS pressure boundaries for divergence-free and viscous SPH fluids". *Computers & Graphics* 76 (2018), 37–46 2.

[BK15] Bender, Jan and Koschier, Dan. "Divergence-free smoothed particle hydrodynamics". *Proceedings of the 14th ACM SIGGRAPH/Eurographics symposium on computer animation*. ACM. 2015, 147–155 2, 6.

[BT07] Becker, Markus and Teschner, Matthias. "Weakly compressible SPH for free surface flows". *Proceedings of the 14th ACM SIGGRAPH/Eurographics symposium on computer animation* (2007), 209–217 2.

[DA12] Dehnen, Walter and Aly, Hossam. "Improving convergence in smoothed particle hydrodynamics simulations without pairing instability". *Monthly Notices of the Royal Astronomical Society* 425.2 (2012), 1068–1082 2, 3.

[DC99] Desbrun, Mathieu and Cani, Marie-Paule. *Space-Time Adaptive Simulation of Highly Deformable Substances*. Research Report RR-3829. INRIA, 1999. URL: https://hal.inria.fr/inria-00072829 2.

[DCG11] Domínguez, J.M., Crespo, Alex, and Gómez-Gesteira, Moncho. "Optimization strategies for parallel CPU and GPU implementations of a meshfree particle method". (Oct. 2011) 2.

[DCV*13] Dominguez, Jose M, Crespo, Alejandro JC, Valdez-Balderas, Daniel, et al. "New multi-GPU implementation for smoothed particle hydrodynamics on heterogeneous clusters". *Computer Physics Communications* 184.8 (2013), 1848–1860 1.

[FM15] Fujisawa, Makoto and Miura, Kenjiro T. "An efficient boundary handling with a modified density calculation for SPH". *Computer Graphics Forum*. Vol. 34. 7. Wiley Online Library. 2015, 155–162 2.

[GBP*17] Gissler, Christoph, Band, Stefan, Peer, Andreas, et al. "Generalized drag force for particle-based simulations". *Computers & Graphics* 69 (2017), 1–11 7.

[GLHL11] Garcia, Ismael, Lefebvre, Sylvain, Hornus, Samuel, and Lasram, Anass. "Coherent parallel hashing". *ACM Transactions on Graphics (TOG)*. Vol. 30. 6. ACM. 2011, 161 2.

[GM77] Gingold, R. A. and Monaghan, J. J. "Smoothed particle hydrodynamics-theory and application to non-spherical stars". *Monthly Notices of the Roy. Astronomical Soc.* 181 (1977), 375–389. ISSN: 0035-8711 1, 2.

[GPB*19] Gissler, Christoph, Peer, Andreas, Band, Stefan, et al. "Interlinked SPH Pressure Solvers for Strong Fluid-Rigid Coupling". *ACM Transactions on Graphics (TOG)* 38.1 (2019), 5 2.

[Gre10] Green, Simon. "Particle Simulation using CUDA". *Cuda 4.0 Sdk* May (2010) 1–3, 7, 8.

[GSSP10] Goswami, Prashant, Schlegel, Philipp, Solenthaler, Barbara, and Pajarola, Renato. "Interactive SPH simulation and rendering on the GPU". *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. Eurographics Association. 2010, 55–64 1, 2.

[IABT11] Ihmsen, Markus, Akinci, Nadir, Becker, Markus, and Teschner, Matthias. "A parallel SPH implementation on multi-core CPUs". *Comput. Graph. Forum* 30.1 (2011), 99–112. ISSN: 01677055. eprint: 1110.3711 1–4.

[ICS*13] Ihmsen, Markus, Cornelis, Jens, Solenthaler, Barbara, et al. "Implicit incompressible SPH". *IEEE transactions on visualization and computer graphics* 20.3 (2013), 426–435 2.

[IOS*14] Ihmsen, Markus, Orthmann, Jens, Solenthaler, Barbara, et al. "SPH Fluids in Computer Graphics". *Eurographics STARS* 2 (2014), 21–42. ISSN: 1017-4656 2, 3.

[KAD*06] Keiser, Richard, Adams, Bart, Dutré, Philip, et al. *Multiresolution particle-based fluids*. Technical Report 520. Department of Computer Science, ETH Zurich, 2006, 10 2.

[Kar12] Karras, Tero. *Thinking Parallel, Part III: Tree Construction on the GPU*. Accessed: 2019-06-17. 2012. URL: https://devblogs.nvidia.com/thinking-parallel-part-iii-tree-construction-gpu/ 3.

[KB17] Koschier, Dan and Bender, Jan. "Density maps for improved SPH boundary handling". *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. ACM. 2017, 1 2, 6.

[LH06] Lefebvre, Sylvain and Hoppe, Hugues. "Perfect spatial hashing". *ACM Transactions on Graphics (TOG)*. Vol. 25. 3. ACM. 2006, 579–588 2, 4.

[MCG03] Müller, Matthias, Charypar, David, and Gross, Markus. "Particle-Based Fluid Simulation for Interactive Applications". *Proc. ACM SIGGRAPH/Eurographics Symp. Comput. Animat.* 5 (2003), 154–159. ISSN: 17275288 2.

[MLJ*13] Museth, Ken, Lait, Jeff, Johanson, John, et al. "OpenVDB: an open-source data structure and toolkit for high-resolution volumes". *Acm siggraph 2013 courses*. ACM. 2013, 19 2.

[MM13] Macklin, Miles and Müller, Matthias. "Position based fluids". *ACM Trans. Graph.* 32.4 (2013), 1. ISSN: 07300301. DOI: 10.1145/2461912.2461984 2.

[Mon05] Monaghan, Joseph J. "Smoothed particle hydrodynamics". *Reports on progress in physics* 68.8 (2005), 1703 2, 6.

[OK12] Orthmann, Jens and Kolb, Andreas. "Temporal blending for adaptive SPH". *Computer Graphics Forum*. Vol. 31. 8. Wiley Online Library. 2012, 2436–2449 2.

[Pri12] Price, Daniel J. "Smoothed particle hydrodynamics and magnetohydrodynamics". *Journal of Computational Physics* 231.3 (2012), 759–794 2.

[SP09] Solenthaler, Barbara and Pajarola, Renato. "Predictive-corrective incompressible SPH". *ACM transactions on graphics (TOG)*. Vol. 28. 3. ACM. 2009, 40 2.

[WHK16] Winchenbach, Rene, Hochstetter, Hendrik, and Kolb, Andreas. "Constrained Neighbor Lists for SPH-based Fluid Simulations". *Proceedings of the 15th ACM SIGGRAPH/Eurographics symposium on computer animation*. July 2016 2, 6–8.

[WHK17] Winchenbach, Rene, Hochstetter, Hendrik, and Kolb, Andreas. "Infinite Continuous Adaptivity for Incompressible SPH". *ACM Transactions on Graphics (TOG)* 36.4 (2017), 102:1–102:10 1–3, 5, 7, 8.

[WRR18] Winkler, Daniel, Rezavand, Massoud, and Rauch, Wolfgang. "Neighbour lists for smoothed particle hydrodynamics on GPUs". *Computer Physics Communications* 225 (2018), 140–148 1.

[WTYH18] Wu, Kui, Truong, Nghia P, Yuksel, Cem, and Hoetzlein, Rama. "Fast Fluid Simulations with Sparse Volumes on the GPU". 2018 2.

[YT13] Yu, Jihun and Turk, Greg. "Reconstructing surfaces of particle-based fluids using anisotropic kernels". *ACM Transactions on Graphics (TOG)* 32.1 (2013), 5 7.