Fast Octree Neighborhood Search for SPH Simulations

JOSÉ ANTONIO FERNÁNDEZ-FERNÁNDEZ, RWTH Aachen University, Germany
LUKAS WESTHOFEN, RWTH Aachen University, Germany
FABIAN LÖSCHNER, RWTH Aachen University, Germany
STEFAN RHYS JESKE, RWTH Aachen University, Germany
ANDREAS LONGVA, RWTH Aachen University, Germany
JAN BENDER, RWTH Aachen University, Germany

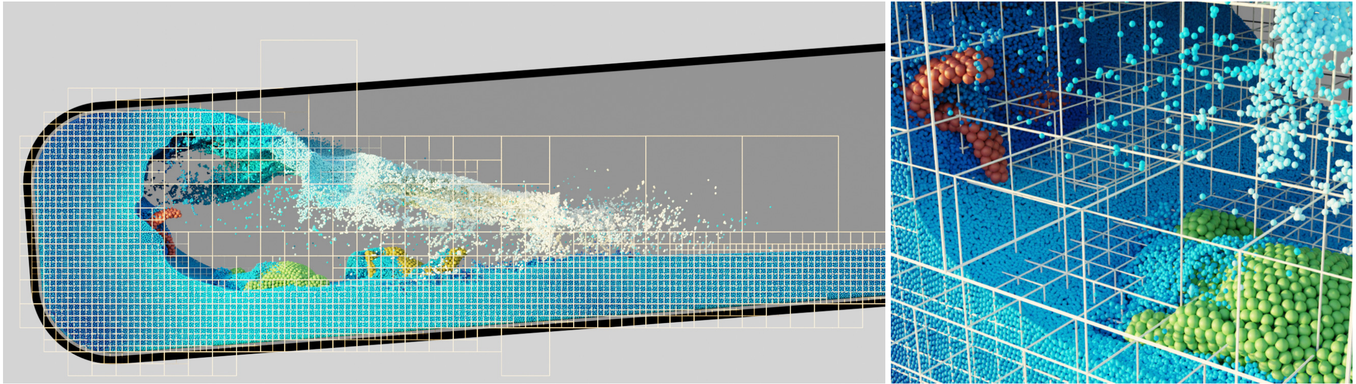


Fig. 1. Multi-resolution Smoothed Particle Hydrodynamics simulation of a swinging box that contains fluid and deformable objects. 5.6 million particles (from blue to white) are used to discretize the fluid and a total of 12 thousand particles (in red, green and yellow) are used to discretize the three deformable objects. The fluid particles have 2.5 times smaller radius than the ones used for the solids. The octree used to accelerate the neighborhood search is shown as a white wireframe. On the left, a far view shows the adaptivity of our acceleration structure at a global scale. On the right, a close up picture better shows the difference in particle sizes and how the octree behaves around that area of interest. For better visibility on the right picture, we show the octree uniformly coarsened by one level.

We present a new octree-based neighborhood search method for SPH simulation. A speedup of up to 1.9x is observed in comparison to state-of-the-art methods which rely on uniform grids. While our method focuses on maximizing performance in fixed-radius SPH simulations, we show that it can also be used in scenarios where the particle support radius is not constant thanks to the adaptive nature of the octree acceleration structure.

Neighborhood search methods typically consist of an acceleration structure that prunes the space of possible particle neighbor pairs, followed by direct distance comparisons between the remaining particle pairs. Previous works have focused on minimizing the number of comparisons. However, in an effort to minimize the actual computation time, we find that distance comparisons exhibit very high throughput on modern CPUs. By permitting more comparisons than strictly necessary, the time spent on preparing and searching the acceleration structure can be reduced, yielding a net positive speedup. The choice of an octree acceleration structure, instead of the uniform grid typically used in fixed-radius methods, ensures balanced computational tasks. This benefits both parallelism and provides consistently high computational intensity for the distance comparisons. We present a detailed account of high-level considerations that, together with low-level decisions, enable high throughput for performance-critical parts of the algorithm.

Finally, we demonstrate the high performance of our algorithm on a number of large-scale fixed-radius SPH benchmarks and show in experiments with a support radius ratio up to 3 that our method is also effective in multi-resolution SPH simulations.

CCS Concepts: • **Computing methodologies → Physical simulation**.

Additional Key Words and Phrases: Neighborhood Search, Smoothed Particle Hydrodynamics

ACM Reference Format:
José Antonio Fernández-Fernández, Lukas Westhofen, Fabian Löschner, Stefan Rhys Jeske, Andreas Longva, and Jan Bender. 2022. Fast Octree Neighborhood Search for SPH Simulations. *ACM Trans. Graph.* 41, 6, Article 242 (December 2022), 13 pages. https://doi.org/10.1145/3550454.3555523

Authors' addresses: José Antonio Fernández-Fernández, RWTH Aachen University, Aachen, Germany, fernandez@cs.rwth-aachen.de; Lukas Westhofen, RWTH Aachen University, Aachen, Germany, l.westhofen@cs.rwth-aachen.de; Fabian Löschner, RWTH Aachen University, Aachen, Germany, loeschner@cs.rwth-aachen.de; Stefan Rhys Jeske, RWTH Aachen University, Aachen, Germany, jeske@cs.rwth-aachen.de; Andreas Longva, RWTH Aachen University, Aachen, Germany, longva@cs.rwth-aachen.de; Jan Bender, RWTH Aachen University, Aachen, Germany, bender@cs.rwth-aachen.de.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Association for Computing Machinery.
0730-0301/2022/12-ART242 $15.00
https://doi.org/10.1145/3550454.3555523

1 INTRODUCTION

Smoothed Particle Hydrodynamics (SPH) is an established meshless method for simulating fluids, elastic solids and granular material. SPH expresses quantities of interest for a given particle — such as density or pressure forces — as a sum of contributions from *neighboring* particles in close proximity. The identification of neighboring particles — the neighborhood search — is therefore an integral component of all SPH simulators. Two particles are neighbors if the distance between them is smaller than the support radius of the SPH kernel. Historically, most SPH simulators in computer graphics have used constant support radii. This assumption has been exploited by specialized fixed-radius neighborhood search algorithms.

A key idea often used in fixed-radius neighborhood search methods is the placement of each particle in a sparse uniform grid with cell size equal to the support radius. Then all neighbors of a particle are guaranteed to be located in the cell of the particle or in any of the adjacent neighbor cells. This way, the fixed-radius approach performs an almost minimal number of distance comparisons to determine if two particles are truly neighbors. On the other hand, the small number of particles considered for each grid cell makes it difficult to fully exploit vectorization and pipelining in modern CPU hardware — key enablers for high performance. Furthermore, the small size of the cells necessitates a larger number of cells, leading to noticeable computational costs for managing the acceleration structure itself.

We find that if we use appropriately sized, larger — and therefore fewer — cells, the computational cost of the acceleration structure is significantly reduced, but the overall cost of the brute-force distance comparisons is only marginally increased. Therefore, the overall cost is lower.

By aggregating particles into these larger cells, we find that it is favorable to exchange the sparse uniform grid acceleration structure with a specialized octree algorithm where the tree leaf nodes consist of clusters of cells. The top-down octree construction clusters cells so that each leaf node has approximately the same number of particles, regardless of the number of cells, which benefits parallelism and ensures that the brute force stage always works on a near-optimal number of particles, thereby maximizing throughput. Contrary to conventional wisdom, which suggests using the support radius as cell size is essential to performance, we demonstrate through numerous examples that our method is consistently faster than state-of-the-art fixed-radius methods.

A seemingly limitless number of choices can be made in the design of a tailor-made acceleration structure. In this paper we present high-level decisions in detail that enable high throughput for performance-critical parts of the algorithm. In this context, we put special emphasis on vectorization and a branchless implementation.

Some state-of-the-art methods *z-sort* the particle data and then rely on this ordering for gathering particle data for distance comparisons. Sorting particle data in this way is in any case beneficial to SPH methods, as it improves cache locality for the SPH loops [Ihmsen et al. 2011]. However, it is generally not necessary to do this for every time step of the simulation; amortizing this cost over several time steps is usually preferable. Our method does not require z-sorted particle data to produce correct output. Instead, we propose a technique in which we take advantage of *almost sorted* data for a more compact representation of particle data in our octree.

Although the fixed-radius assumption has proved to be effective for simulating single-phase fluids, it nonetheless imposes severe limitations on complex multi-physics simulations, such as multi-phase fluids or fluid-solid interaction. In these settings, it is natural to want to use different support radii for different objects. Here, slow-moving, viscous fluids or elastic solids generally require fewer particles for high-fidelity results compared to the intricate patterns formed by swirling water.

To ensure efficient simulations in these settings, we must do away with the fixed-radius assumption. Our proposed octree-based method is also capable of handling SPH simulations where particles have different support radii. Our results therefore challenge the prevalent notion that the fixed-radius assumption is the key to fast neighborhood search in SPH simulators.

We demonstrate the performance of our method across both fixed-radius and multi-resolution examples. Our fast neighborhood search requires only roughly 5%-10% of the total simulation time across all of our examples, and consistently outperforms a representative state-of-the-art fixed-radius method across all of our fixed-radius benchmarks, with a speedup of up to 1.9x. Fig. 1 illustrates the distribution of particles in our octree in a large-scale multi-physics scene.

2 RELATED WORK

In this work we present a neighborhood search approach that is specifically tailored for SPH. SPH is a popular method in computer graphics for the simulation of a wide range of materials and effects. First, we will show an overview of relevant publications about SPH in general followed by a discussion of works on neighborhood search. Finally, we will classify and summarize the most prevalent approaches for neighborhood search and elaborate on their limitations that inspired our work.

2.1 Smoothed particle hydrodynamics

While the SPH formulation was originally introduced in the field of astrophysics [Gingold and Monaghan 1977], today's application areas also include the field of computer graphics. Recent advancements involve the simulations of incompressible fluids [Bender and Koschier 2017; Ihmsen et al. 2014a], highly viscous materials [Takahashi et al. 2015; Weiler et al. 2018], deformable solids [Kugelstadt et al. 2021; Peer et al. 2017] and snow [Gissler et al. 2020]. An overview over current research can be found in the latest surveys by Ihmsen et al. [2014b] and Koschier et al. [2022]. For simulating the aforementioned effects, the SPH method relies on locally computing the required physical quantities by interpolation using the discrete particles inside a given support radius, which is typically dependent on the particle radius. In computer graphics it is common practice to use the same support radius for all particles.

However, to improve runtime, incorporating different particle resolutions may be considered to significantly reduce the amount of particles without sacrificing visual quality. For example, in simulations between a low and highly viscous fluid a fine resolution is

needed to capture the high frequency motions of the former material, while larger particles can be used for the latter. Thus, choosing an independent, appropriate resolution may help to speed up the simulation significantly.

Incorporating this concept, Desbrun and Cani [1999] spatially adapt the particle resolution based on the density deviation of neighboring particles. The idea of particle splitting and merging was also adopted in several recent publications [Adams et al. 2007; Winchenbach et al. 2017; Winchenbach and Kolb 2021]. Another approach is to utilize different, distinct particle resolutions in desired regions of interest [Horvath and Solenthaler 2013; Solenthaler and Gross 2011]. While all of the aforementioned methods show a significant improvement in simulation quality or efficiency, special attention must be given to computing the neighborhood of each particle. As denoted by Xia and Liang [2016], incorporating a single, standard fixed-radius neighborhood search algorithm like compact hashing [Ihmsen et al. 2011] must use the largest support radius of all particles to avoid artifacts. This introduces significant computational overhead for smaller particles since their associated computations will include several particles outside their support radius. Hoverath and Solenthaler [2013] as well as Solenthaler and Gross [2011] execute one neighborhood search for each level of detail in their multi-resolution approaches. This leads either to an increased memory footprint at the interface of different resolutions or requires an unphysical boundary region. In summary, efficiently determining the neighborhood in spatially adaptive SPH simulations is a challenging task.

2.2 Neighborhood search

In SPH-based simulations the neighborhood search step is an integral component since a field quantity for a fluid particle is computed by interpolating over its neighboring particles within the support radius of the SPH kernel. Therefore, different approaches have been proposed in the computer graphics community. An overview of CPU- and GPU-based algorithms can be found in the survey of Ihmsen et al. [2014b]. We focus on CPU-based algorithms, which, despite the advent of efficient GPU simulators, remain relevant especially for very large simulations that are limited by the amount of available memory, such as those shown by Band et al. [2020].

State-of-the-art neighborhood search methods are mainly based on either spatial grids, Verlet lists [Verlet 1967] or hierarchical data structures. Nowadays, the computer graphics community tends towards uniform grid-based methods. These store the particles inside a uniform background grid and calculate the neighbors of a particle by querying the adjacent cells. Most grid-based approaches are either based on Cell-Linked Lists (CLL) [Band et al. 2020; Domínguez et al. 2010; Green 2010; Pelfrey and House 2010] or hash maps [Ihmsen et al. 2011; Tang et al. 2018; Winchenbach and Kolb 2020]. While offering fast construction and query times from the standpoint of computational complexity, grid-based methods may suffer from high cache miss rates on the CPU if the particles are not spatially indexed and sorted. Regardless of the spatial sorting, the resulting neighbor lists are recomputed in each time step.

Leveraging the temporal coherence of neighboring particles, the idea of Verlet lists is to extend the amount of potential neighbors in relation to a particle's velocity. Thus, these predicted neighbors may help to speed up the neighborhood search [Viccione et al. 2008; Willis et al. 2018]. The main disadvantages of using Verlet lists is the computational overhead if the prediction fails to capture the movement of particles and the increased memory footprint. The latter also results in them being unfavorable for large-scale and GPU-based simulations.

Both of the aforementioned concepts can be modified to suit adaptive particle radii. Winchenbach and Kolb [2020] use a hashed, multi-resolution grid, which exploits the self-similarity of Morton codes for the particle sorting. This way, it is possible to query the acceleration structure at different resolution levels depending on the support radius of any individual particle. Regarding Verlet lists, Winchenbach et al. [2016] compute memory constrained neighbor lists in a predictor-corrector fashion which allows for variable support radii. In this method, the support radius of a particle is adjusted so that a maximum fixed number of neighbors is attained.

Tree-based neighborhood search methods have also been used to tackle the neighborhood search in spatially adaptive SPH simulations [Harada et al. 2007; Hernquist and Katz 1989; Xia and Liang 2016]. Since this acceleration structure yields a multi-resolution view of the simulation domain, neighboring particles can easily be determined by a traversal of the tree.

The previously denoted categories of neighborhood search algorithms have also been adapted to better suit the strengths of the used underlying hardware. Examples leveraging the streaming architecture of GPUs include grid-based [Green 2010; Gross et al. 2019], Verlet list-based [Willis et al. 2018] and tree-based methods [Xia and Liang 2016].

Our novel approach can be classified as a tree-based method. However, instead of using the acceleration structure for tree-traversal based neighbor checks, the employed octree functions as a domain decomposition. In contrast to traditional grid-based methods, the resulting task size for the brute force stage adapts to the underlying fluid geometry.

3 METHOD

In the following, we define the neighborhood search problem in the context of SPH (Section 3.1), and give a high-level overview of how our method solves the problem (Section 3.2). We provide a detailed account of the implementation details of our method in Section 3.3. In Section 3.4, we discuss the required extensions that enable our method to be used for variable support radii. Finally, we discuss the differences to other fast fixed-radius methods in Section 3.5.

3.1 Problem description

In the standard fixed-radius SPH formulation, a scalar field f is approximated by interpolating values from a discrete set of particles as

$$f(\mathbf{x}_i) = \sum_{j \in \mathcal{N}_i} \frac{m_j}{\rho_j} f(\mathbf{x}_j) W(\mathbf{x}_i - \mathbf{x}_j, h), \tag{1}$$

where m_j, ρ_j and $\mathbf{x}_j$ are the mass, the density and the position of particle j. W is a compactly supported kernel with smoothing length h and support radius r. This means that only the particles within the support radius r of $\mathbf{x}_i$ have a non-zero contribution to the value of $f(\mathbf{x}_i)$. Hence, we only sum up contributions of particles j that

are in the neighborhood $\mathcal{N}_i = \{j \mid \|\mathbf{x}_i - \mathbf{x}_j\|_2 \leq r\}$ of particle i. Therefore, SPH simulators require efficient neighborhood search methods to determine the neighborhood $\mathcal{N}_i$ for each particle i.

3.2 Method overview

A global brute force approach to the neighborhood search problem has a complexity of $O(n^2)$, which is prohibitive even for medium sized simulations. To address this, our method, like most other approaches, uses an acceleration structure to decompose the global problem into many self-contained subproblems so that checking particle pairs with a large distance is avoided.

Our method employs a novel two stage acceleration structure: In a problem reduction preprocessing step, we assign all particles to the cells of an encapsulating uniform grid. Then, we use an octree to adaptively find clusters of non-empty grid cells which in total approximately contain a specified number of particles. Finally, we use a brute force approach on the particles of each cluster to generate the corresponding neighbor lists. All stages are performed using optimized SIMD code which is discussed in Section 3.3. An overview of our approach is shown in Fig. 2.

State-of-the-art solutions for the neighborhood search problem that solely use uniform grids rely on two operations where modern computer architectures are slow: random memory reads and hard-to-predict branching. Our method, on the other hand, is designed with modern CPU architectures in mind. Almost for the entirety of the execution time, the CPU is executing branchless code on data that is contiguous in memory and which has an optimized layout. This is further enhanced by our octree being able to generate clusters of a specific number of particles, which can be processed with higher computational efficiency in comparison to the clusters generated by uniform grid methods, which are typically very small. These are the main reasons why our method achieves significant runtime improvements when compared with uniform grid methods.

3.3 Implementation

In this section we discuss the three main stages (see Fig. 2) of our method in more detail.

3.3.1 *Particles to Cells.*

We employ a uniform grid to classify the particles into cells in order to speed up the construction of the octree, which is the main component of the acceleration structure. In contrast to uniform grid methods, we do not use the grid for queries of neighboring cells. Its only purpose is to reduce the number of entities to be processed in the octree.

To classify the particles into cells, first we compute a bounding box containing all particles. Then we define a uniform background grid with cell size $d \in \mathbb{R}$, which encloses the bounding box of the particles and use the minimum point of the box as origin $\mathbf{o} \in \mathbb{R}^3$. Our results show that a good choice for the cell size is 1.5 times the support radius (Section 4). The integer coordinates $(k, l, m)_c \in \mathbb{N}_0^3$ of a cell c that contains a particle i is given by the component-wise floor function

$$(k, l, m)_c = \left\lfloor \frac{\mathbf{x}_i - \mathbf{o}}{d} \right\rfloor. \tag{2}$$

By subtracting the origin, all cell coordinates are non-negative. We denote by $\mathcal{P}_c$ the set of particle indices assigned to c and by $\Omega_c \subset \mathbb{R}^3$ the physical domain of the cell.

The input to our octree construction algorithm (Section 3.3.2) is an array of cells. Conceptually, each cell contains a set of particles. One way to compute the cells is to first sort all the particles according to their Morton code [Morton 1966] with respect to the uniform grid. We refer to this process as *z-sorting*. Then all the particles inside a given cell will share the same Morton code, and therefore will be next to each other in the sorted particle array. This is the approach used to construct the CLL data structure employed by Band et al. [2020]. The array of cells can then be constructed by identifying sequences of particles that share the same Morton code in the sorted particle array. Since the particles for a given cell are contiguous in the particle array, the particle range associated with a cell can be compactly represented by the *begin* and *end* offsets to the particle array. This idea is illustrated in Fig. 3.

Since we tend to assign approximately 30 particles to each cell (Section 4), this means that the octree, which only works with cells and not individual particles, only processes about 1/30 the number of entities compared to the total particle count, which significantly reduces the computational cost of the acceleration structure. The actual particle data is only gathered in the brute force stage.

A problem with this approach is that a full z-sort on the list of particles can add significant computation time to the neighborhood search. A key component of our approach is the realization that the particle list does not need to be *perfectly* z-sorted before handing off the resulting cells to the octree construction algorithm. Nothing changes in the octree algorithm if two ranges of particles share the same cell in the uniform grid but are not contiguous in the particle array. Since SPH methods anyway tend to occasionally z-sort the particle data for improving cache behavior [Ihmsen et al. 2011], we can therefore exploit the fact that the particle data is expected to be *almost z-sorted* anyway. In the absence of a perfectly z-sorted sequence of particles, our method will still greedily cluster them as described above. In the worst case, if the particles are not close to be z-sorted, our method will still produce the correct result, but performance will degrade: the number of cells in the octree will get closer to the number of particles. To avoid this, we force a global z-sort of the particles before the first time step of the simulation. Our results show that it is sufficient to z-sort the particles every few time steps (Section 4).

3.3.2 *Octree.*

We use an octree to divide the global problem into smaller, self-contained subproblems by clustering cells at the leaves of the octree. The octree is aligned to the uniform grid so that any grid cell is geometrically contained in exactly one leaf node. The uniform grid has a power of two number of cells in each direction to ensure that splits in half do not invalidate that restriction. The grid cells that are geometrically contained in an octree leaf node are the *interior* cells for the node. In addition, each leaf node has a set of *exterior* cells. These are the cells that are not geometrically contained inside the leaf node, but have particles that are potential neighbors with the particles in the interior cells.

By construction, each cell is therefore interior in exactly one leaf node, and the complete neighborhood $\mathcal{N}_i$ for any particle i in that

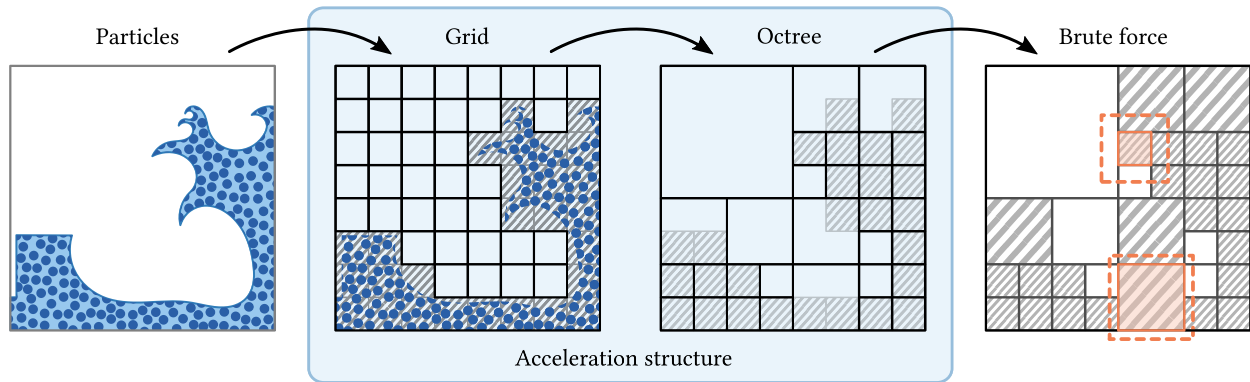


Fig. 2. Overview of our method. The input to our method is a list of particles from an SPH simulation. To build an acceleration structure for our method we first map particles to cells of a background grid using Morton z-indices. The set of all non-empty cells is then geometrically subdivided using an octree until all octree leaves approximately contain a specified number of particles. As the octree only works on cells we can avoid looping over any particles in this stage. Finally, for each octree leaf a brute force neighborhood search is performed.

cell is a subset of the particles in the leaf node cells (interior and exterior). This implies that the neighborhood search can be broken down into independent subproblems: the neighborhoods of interior particles in each octree leaf node can be computed independently in parallel.

The octree is constructed in a top-down manner. Consider an octree node $\mathcal{T}$ with domain $\Omega \subset \mathbb{R}^3$. We define $\widetilde{\Omega}$ as the extended domain obtained by enlarging Ω by the support radius in all coordinate directions.

Starting with the root node of the octree, each octree node $\mathcal{T}$ is recursively split into eight equally sized child nodes with non-overlapping domains (see Fig. 4), in the usual octree fashion. A cell c in $\mathcal{T}$ is assigned to a child node $\mathcal{T}_{\text{child}}$ if the cell domain Ω_c overlaps with octree node domain $\widetilde{\Omega}_{\text{child}}$.

If $\mathcal{T}$ contains only a single interior cell, or the total number of particles across all cells in $\mathcal{T}$ is smaller than a constant threshold, we do not subdivide it further, which means that $\mathcal{T}$ is a leaf node. We study the optimal number of particles per leaf node in Section 4.

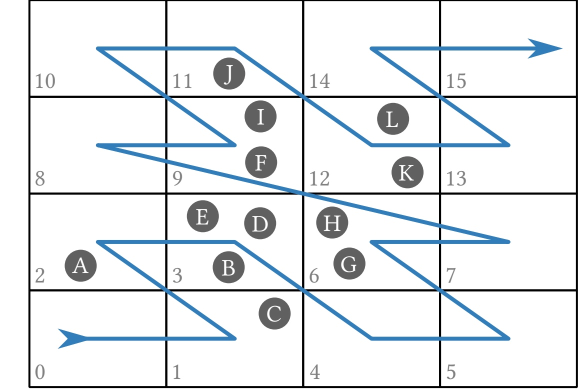
3.3.3 Brute force stage. After the octree construction, we are left with a collection of individual subproblems, one per leaf node of the octree. The final task is to compute the neighbor lists of the *interior particles* of each leaf node $\mathcal{T}$, defined as the particles that belong to the interior cells of $\mathcal{T}$. The remaining *exterior* particles are the particles in the exterior cells of $\mathcal{T}$. All neighbors of any interior particle are by construction either internal or external particles of the node.

At this point, only the leaf and cell data for a task is known, therefore, prior to actually computing the pair-wise distances between the particles, the particle data must be gathered. To provide optimal operational conditions for the core of the brute force procedure (computing inter-particle distances), we store the gathered particle coordinates and indices directly in SIMD-friendly data structures. Finally, we calculate the distance from each interior particle i in Ω to each particle in the extended leaf domain $\widetilde{\Omega}$. The neighboring particles indices are finally written into the neighbor list of particle i.

The neighbor list data structure is a collection of fixed size blocks of memory. The size of the blocks is chosen large enough to hold a few thousands of single particle neighbor lists. To avoid data races and false sharing, each thread has its own list of these blocks to which it has exclusive access. When a neighbor list cannot be appended to the current active block of a thread, a new block exclusive to that thread is allocated. No memory reallocations of previously written neighbor lists are needed. We use a separate structure for the unlikely cases of a single neighbor list being larger than a block. To access the neighborhood data, a pointer to the beginning of each neighbor list is stored together with the number of neighbors. Particles with consecutive indices are likely to have their neighbor lists stored consecutively in memory, which improves cache coherency during the SPH loops.

3.3.4 Branchless Conditional Pushbacks. A very common operation in our method is to append a value to a list if a certain criterion is met. This *conditional pushback* is a crucial operation during the octree construction and the computation of the neighbor lists. It has three stages: 1. evaluate the condition, 2. if necessary, write the value to the destination pointer, and 3. increment the pointer. Due to branch mispredictions, conditional pushbacks can significantly slow down routines that are otherwise well optimized.

To avoid this issue, we design our solution so that all performance critical conditional pushbacks can be carried out using SIMD permutations, which are instructions that can move multiple numbers inside a special small array (SIMD register) without using branches in a single instruction. The new order must be provided using a index map. We can use these permutations for branchless conditional pushbacks by selectively moving the correct items in the SIMD register to the front, using the appropriate index map, and then appending the contents of the register to the destination list. Finally, we increment the destination pointer by the number of conditionals evaluated to true.



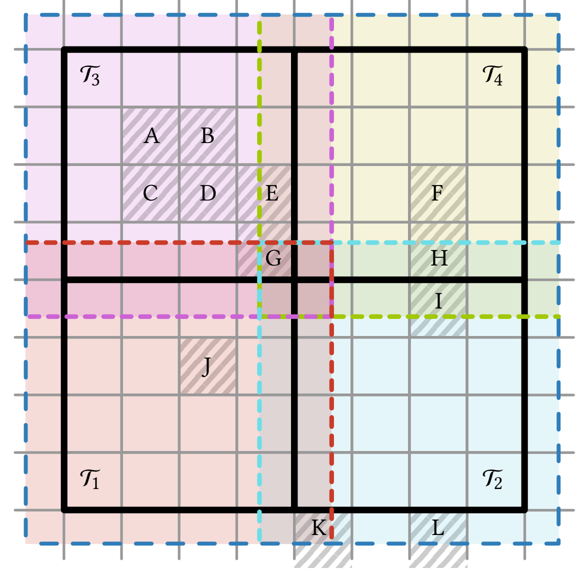
Particles sorted by z-index

Order	C	A	B	D	E	G	H	F	I	J	K	L
Z-index	1	2	3	3	3	6	6	9	9	11	12	12
Cell offsets	0	1	2	5	7	9	10	12	(7 cells)			

Particles almost sorted

Order	A	B	C	D	E	F	G	H	I	J	K	L
Z-index	2	3	1	3	3	9	6	6	9	11	12	12
Cell offsets	0	1	2	3	5	6	8	9	10	12	(9 cells)	

Fig. 3. Mapping particles to the background grid. The particle's z-index identifies the cell of a particle. The list of non-empty cells is then given by the offset into the particle array each time the z-index changes between particles adjacent in memory. For perfectly sorted particles the cell offsets are monotonically increasing. This is not the case for our method where we assume only *almost sorted* particles.



$\mathcal{T}_0$:	A	B	C	D	E	F	G	H	I	J	K	L
$\mathcal{T}_1$:	G	J	K									
$\mathcal{T}_2$:	G	H	I	K	L							
$\mathcal{T}_3$:	A	B	C	D	E	G						
$\mathcal{T}_4$:	E	F	G	H	I							

Fig. 4. Subdivision of an octree cell $\mathcal{T}_0$. The blue dashed line indicates the parent cell $\mathcal{T}_0$'s enlarged domain $\widetilde{\Omega}_0$ containing 12 non-empty cells A to L (shaded). The thick black lines and colored backgrounds indicate the non-overlapping sub-domains $\Omega_{1...4}$ of the child nodes $\mathcal{T}_{1...4}$ dividing Ω_0 into quadrants (octants in 3D). The non-empty cells are then assigned to all child nodes whose extended subdomains $\widetilde{\Omega}_{1...4}$ (colored dashed lines) overlap with the respective cell.

In Listing 1, we show the actual instructions we use for the case of conditionally appending integer values using the AVX2 instruction set, which gives us access to permutations of up to eight 32-bit integers. Let's assume we have eight integer values (`src`) that we want to selectively copy over to a different list (`dst`) depending whether eight conditional values indicate true or false (`cond`). First, we need to generate a index map from the eight conditional values. While the index map can be generated programmatically, we found that using a precomputed lookup table is faster (line 4). To index the lookup table we need an index between 0 and 255 (all possible eight true/false combinations) from the conditional array. The instruction `movemask` takes the conditional array and returns an integer with its lower bits as one if true and zero if false which we can use as index (line 3). Then, we can use the `permute` SIMD instruction with the source values and the index map to get the values corresponding to positive conditionals in front of an 8 value array (line 5). This shuffled array is then copied to the destination pointer (line 6). Finally, the destination pointer must be incremented by the exact number of bits equals to one in the mask, corresponding to the number of elements appended to the array, which is determined with the `popcnt` instruction (line 7). See Fig. 5 for a diagram of the process.

```
1 void pushback_8int(__m256i& src, __m256& cond, int*& dst)
2 {
3   int mask = _mm256_movemask_ps(cond);
4   __m256i index_map = lookup_table[mask];
5   __m256i shuffled_values = _mm256_permutevar8x32_epi32(
         src, index_map);
6   _mm256_storeu_si256((__m256i*)dst, shuffled_values);
7   dst += _mm_popcnt_u32(mask);
8 }
```

Listing 1. Branchless conditional pushbacks with SIMD. Eight integers are conditionally appended to a list without the use of branches. `__m256i& src` is a SIMD type that holds the eight source integers, `__m256& cond` is a SIMD type that holds the result of eight floating point comparisons, `int* dst` is the destination pointer where the values in `src` corresponding to true values in cond will be copied into.

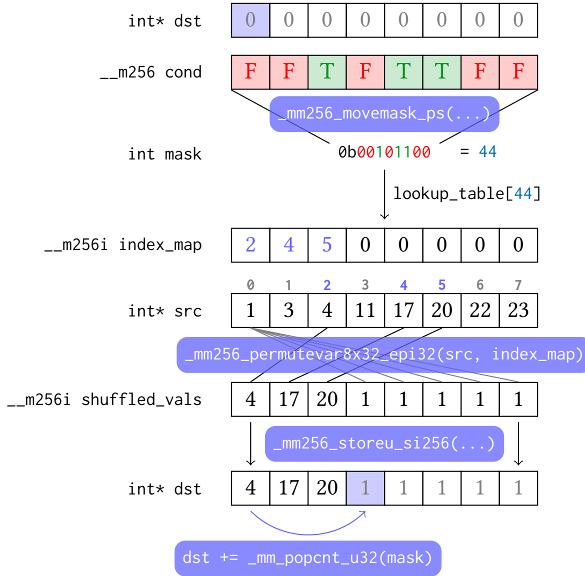


Fig. 5. A diagram of the branchless SIMD pushback.

In the end, a procedure that would have taken 8 branches and up to 8 writes and pointer increments, now takes 5 instructions without any branches. In Section 4, we present benchmarks that show the improvements this procedure brings to our algorithm in several of its stages.

3.3.5 Z-sort. As discussed in Section 3.3.1, our method takes advantage from the particles being almost z-sorted. We do not need to explicitly sort the particles for our method before every neighborhood search, however, we z-sort them occasionally to keep them approximately sorted at all times.

Since the cells that we use for the octree are effectively a coarse representation of the particles, we can use them to speed up sorting the particles. By sorting the cells instead of the particles we can very significantly decrease the computational time needed for the z-sort operation. Since the particles are almost z-sorted, the cells will also be almost z-sorted, therefore sorting them with a suitable sorting algorithm is very fast. The updated particle order is found by concatenating all the particle indices belonging to the cell in the new cell order. As shown in Section 4, this approach can be significantly faster than explicitly sorting the whole particle array directly.

3.4 Variable support radius

While the main objective of our octree design is to maximize performance for fixed-radius neighborhood search problems, we extend our method utilizing the adaptive nature of the octree to also support variable support radii.

Different SPH methods use different definitions for the particle neighborhood in the context of variable support radii. We use the definition proposed by Adams et al. [2007], which states that two particles i and j with positions $\mathbf{x}_i$ and $\mathbf{x}_j$ and support radii r_i and r_j are neighbors if

$$\left\|\mathbf{x}_i - \mathbf{x}_j\right\|_2 \leq \max(r_i, r_j). \tag{3}$$

This definition leads to symmetric neighbor lists where a particle inside the support radius of another will always appear in the neighborlist of that second particle, regardless of its support radius.

In the following, we enumerate the changes required for our method to effectively handle variable particle support radii:

(1) We define the *octree node support radius* $r_{\mathcal{T}} \in \mathbb{R}$ to be the maximum support radius of all the particles an octree node $\mathcal{T}$ contains. Then, the extended octree node domain $\widetilde{\Omega}_{\mathcal{T}} \subset \mathbb{R}^3$ is the extension of $\Omega_{\mathcal{T}}$ by the parent node $r_{\mathcal{T}}$ in all coordinate directions.
(2) A cell c is assigned *exterior cell* of an octree node $\mathcal{T}$ if its domain $\Omega_c \subset \mathbb{R}^3$ intersects the extended octree node search domain $\widetilde{\Omega}_{\mathcal{T}}$.
(3) In the brute force stage, the particle support radii are fetched along with their positions and Eq. 3 is used to find neighbors.

3.5 Discussion

Up to this point our method has been presented in detail. In the next section we will validate it with experiments and compare it with the CLL method introduced by Band et al. [2020], which, to the best of our knowledge, is the fastest solution to the fixed-radius neighborhood search problem for CPUs. However, to fully understand those results we first need to discuss the conceptual differences between the CLL method and our approach.

We claim, and show in our experiments, that our method is significantly faster than the CLL method. This may seem counter-intuitive since the CLL method does perform fewer pair-wise comparisons, yet the reason is straightforward: using an octree to decompose the problem into smaller brute force subproblems (clusters of particles) results in more balanced brute force tasks which contain the target amount of particles that benefits computational throughput.

By construction, uniform grid methods lock themselves into a fairly fixed amount of work per brute force task: a center cell, and its 26 neighbor cells. Since these methods set the cell size equal to the support radius, and for a typical SPH support radius of twice the particle sampling distance, each cell has approximately 8 particles, which makes a total of 216 particles for the 27 cells in each brute force task. This is assuming perfectly packed particle distributions which is not always the case in SPH simulations, e.g., when having thin fluid layers or spray particles. The problem is that, besides not being well balanced, these tasks do not necessarily have the optimal size to solve the problem in the most efficient way possible. All the brute force preparation work, mainly searching the cells and fetching the particle data, has to be carried out regardless of the amount of particles in the task. However, there is an optimal number of particles for a brute force task which balances its quadratic complexity in regards of the pair-wise comparisons with the overhead from setting up the brute force itself. As shown in Section 4, we find the optimal task size to be 1000 particles in our benchmark conditions, which is much more than the 216 particles that uniform grid methods operate on per task. The benefits of larger (or rather not very small) tasks go beyond reducing the setup overhead of

the brute force: better cache and SIMD utilization being the most important ones.

Another important idea we introduce is challenging the concept that the best cell size for the background uniform grid is the support radius. As described in the previous section, we use a uniform grid in our method as a problem reduction step. Since we want to generate larger brute force tasks in comparison to uniform grid methods, we do not need such small cells anymore. Larger cells, means fewer cells and more particles per cell. Having fewer cells reduces the time to build the octree and the amount of cells at the octree leaves. More particles per cell means that we load particles in larger batches, which better amortizes memory accesses. In fact, our experiments show that an optimal cell size in our benchmark conditions is one that results in 30 particles per cell on average.

4 RESULTS

In this section we discuss several experiments for our neighborhood search approach. Moreover, we compare our method with the state-of-the-art Cell-Linked-Lists method (CLL) [Band et al. 2020] which is currently one of the fastest fixed-radius neighborhood search solutions. To ensure a fair comparison, we included our optimizations (bruteforce, branchless SIMD pushbacks, neighbor lists layout etc.) into our CLL implementation. The CLL method presented by Band et al. [2020] lacks a description of low level optimizations. Not including our optimizations would have resulted in significantly worse results for the CLL method. However, we did not use variable cell sizes in our CLL solution because using the support radius as cell size is one of, if not the most, fundamental design choices of that method. Since our focus is fixed-radius neighborhood search and CLL only supports a constant radius for all particles, we perform the runtime comparisons only for fixed-radius simulations. Additionally, we show that it is possible to use our modified method in multi-resolution SPH simulations while keeping the neighborhood search runtime significantly lower than the other components of the SPH solver.

All benchmarks were performed on an Intel Core i9-9900K processor with 8 physical cores at 3.60 GHz. We used AVX2 instructions to implement the SIMD vectorization. For the simulations we integrated our neighborhood search method in the open-source SPH framework SPlisHSPlasH [Bender et al. 2022]. In all benchmarks we used the pressure solver *Divergence-Free SPH* [Bender and Koschier 2017] and an implicit boundary representation [Bender et al. 2020]. For the Beach Scene (see Fig. 6), we employ the method by Bender at al. [2018] to enhance the turbulent behavior of the water. For the multi-resolution experiments we extended the pressure solver following the approach of Adams et al. [2007] using averaged kernel values for particles at the interface of the different resolutions. In all experiments we used the cubic spline kernel [Monaghan 1992] and the support radius was twice the initial inter-particle sampling distance.

4.1 Fixed-Radius Neighborhood Search

We will use two scenes for these experiments. The first one, the Beach Scene (see Fig. 6), consists of 9 million particles pushed by a wave generator to an inclined shore. The goal of this scene is to represent high particle counts in a relatively dense distribution. In the second scene for the fixed radius experiments, the Fountain Scene (see Fig. 7), up to 3.5 million particles are emitted into a fountain with pools and waterfalls. This scene represents simulations with increasing amounts of particles over time where the particles are more distributed over the simulation domain.

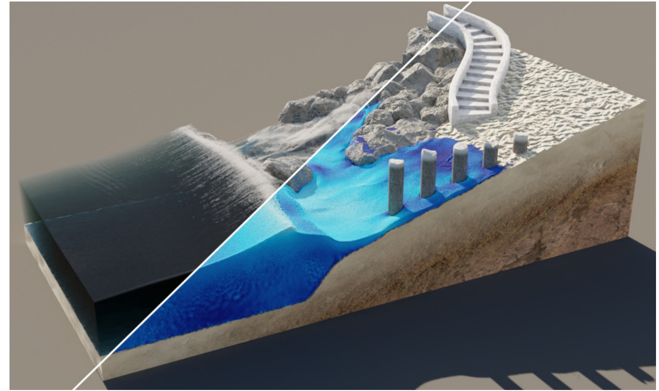


Fig. 6. Beach Scene. A wave generator (not rendered) pushes 9 million particles to a sloped beach shore with obstacles. Surface reconstruction on the left. Particle view on the right.

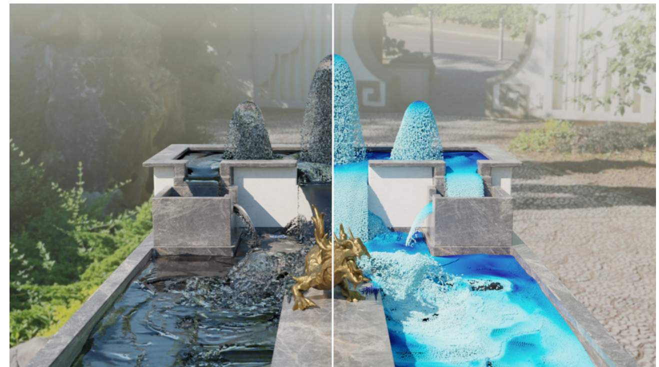


Fig. 7. Fountain Scene. Fluid particles are emitted upwards at the top of a fountain with pools and waterfalls. The number of particles increases up to 3.5 million. Surface reconstruction on the left. Particle view on the right.

4.1.1 Octree Calibration. As discussed in Section 3.5, our method can generate brute force tasks with an approximate target number of particles, which makes it possible for it to be tuned to specific hardware conditions. For this experiment we take a representative frame of the Beach Scene and one from the Fountain Scene and study the effect of the uniform grid *cell size* and the number of particles for which we stop the recursion and generate a brute force task, which we call *cap*. We benchmark each parameter combination for each simulation, normalize the respective results so the fastest combination is 1.0 and finally add and renormalize both benchmarks into a final result in Fig. 8. The results for the individual simulations are in any case very similar.

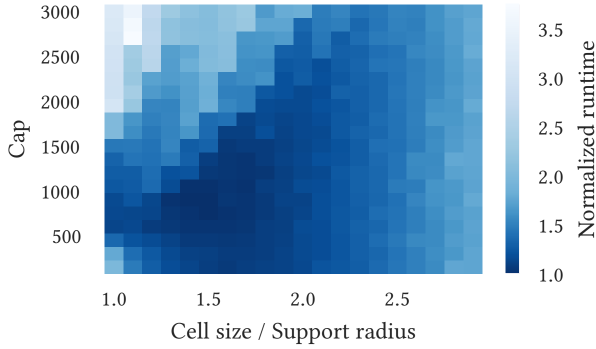


Fig. 8. Normalized timings for a combination of cap and cell size parameters of the octree for the Beach and Fountain scenes. The experiment shows that the optimal values for our benchmark conditions are a cap of 1000 particles and a cell size of 1.5 times the particles support radius.

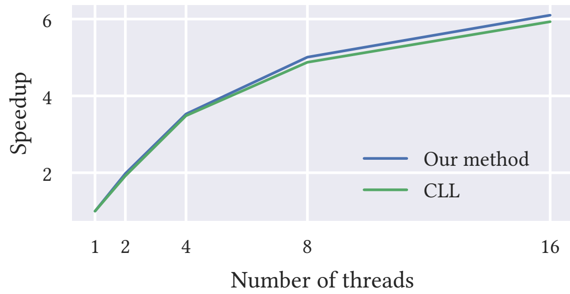


Fig. 9. Parallel scalability comparison between our method and the CLL method for the Beach Scene.

We can see that the region of optimal values lies for caps between 500 and 1500 and cell sizes between 1.25 and 1.75 times the particles support radius. For the rest of the experiments, we use a cap value of 1000 and a cell size of 1.5 times the particle support radius. In the Beach Scene, cells contain 30.59 particles on average.

4.1.2 Parallel scalability. We use the Beach Scene to investigate the parallel scalability of our method in comparison to the CLL method. The results in Figure 9 show that our method scales slightly better in comparison to our implementation of the CLL method.

4.1.3 SIMD and parallelism. In this experiment, we compare our method and the CLL method on the Beach Scene in relation to their CPU usage.

In Table 1 we can see CPU metrics gathered with the performance analyzing tool `perf` for both methods. In general we can see the large impact SIMD optimized routines have in both methods: Much fewer total instructions, better cache utilization and greatly reduced branch mispredictions. However, we can observe that our method experiences larger benefits than the CLL method, which is a direct consequence of our acceleration structure being better suited to SIMD optimizations and also to being able to adapt the brute force tasks to optimal sizes for the hardware.

In Table 2 we can see runtimes for sequential, parallel, scalar and SIMD modes of both methods on the Beach Scene. While both

Table 1. CPU metrics generated with `perf` for our method and the CLL method on the Beach Scene. The table shows the averaged values of 100 benchmark runs.

	Our method		CLL	
	scalar	SIMD	scalar	SIMD
Instructions (mil.)	8143	1772	4345	2715
Cache misses (mil.)	10	7	12	11
Branch misses (mil.)	64	2	72	18

Table 2. Scalar/SIMD and sequential/parallel runtime comparison between Our method and the CLL method. Speedup factor with respect to the respective sequential scalar timing in parentheses.

	Our method	CLL
Sequential Scalar	12.87 s *(1.0x)*	8.47 s *(1.0x)*
Sequential SIMD	1.64 s *(7.9x)*	3.15 s *(2.7x)*
Parallel Scalar	1.55 s *(8.3x)*	1.02 s *(8.3x)*
Parallel SIMD	0.26 s *(49.0x)*	0.52 s *(16.2x)*

Table 3. Timings for the different stages of our method in scalar and SIMD mode on the Beach Scene averaged to a single time step.

	Scalar		SIMD		
Stage	Time	%	Time	%	Speedup
Particles to cells	0.007 s	0.4%	0.003 s	1.3%	2.0x
Octree	0.060 s	3.9%	0.010 s	4.0%	5.7x
Brute force	1.480 s	95.7%	0.248 s	94.7%	6.0x
Total	1.547 s	100.0%	0.262 s	100.0%	5.9x

algorithms scale almost perfectly with the number of cores up to 8, we can see again how our octree takes much better advantage of SIMD instructions. As it is shown in the table, our method is actually slower in scalar mode. This is due to our brute force tasks being larger, and therefore more pair-wise comparisons must be done than in the CLL method. However, when we remove the branches and vectorize the loops, we can make much better use of the hardware, resulting in a net improvement over the CLL method.

4.1.4 Stages of our method. Table 3, shows the runtime for the main stages of our method in scalar and SIMD mode. A remarkable result is that our acceleration structure, constituted by the *particles to cells* and the *octree* stages, takes up only 5.3% of the total neighborhood search runtime. The rest, 94.7%, of the execution time is spent in the highly optimized brute force stage, 28.26% of which is gathering the relevant particle data.

4.1.5 Z-sort. As discussed in Section 3.3.5, every few time steps we z-sort the particles with the help of the last updated cell list. For the Beach Scene, sorting the cells to generate the new particle order takes 4.94 ms on average while actually sorting the particles directly takes 83.0 ms. We found that updating the particle data order every 10 steps is frequent enough while being unnoticeable in relation to the whole operation of the SPH solver.

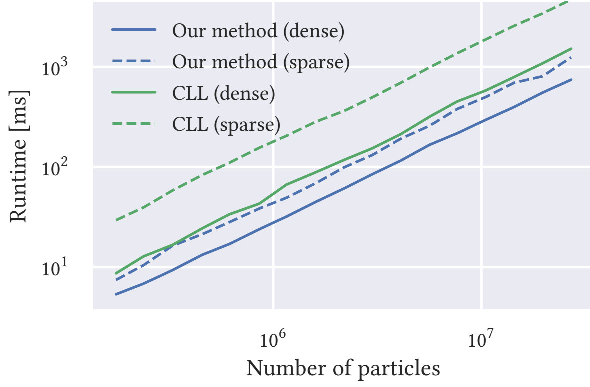


Fig. 10. Problem size scalability comparison for a dense (solid lines) and a sparse (dashed lines) particle distributions.

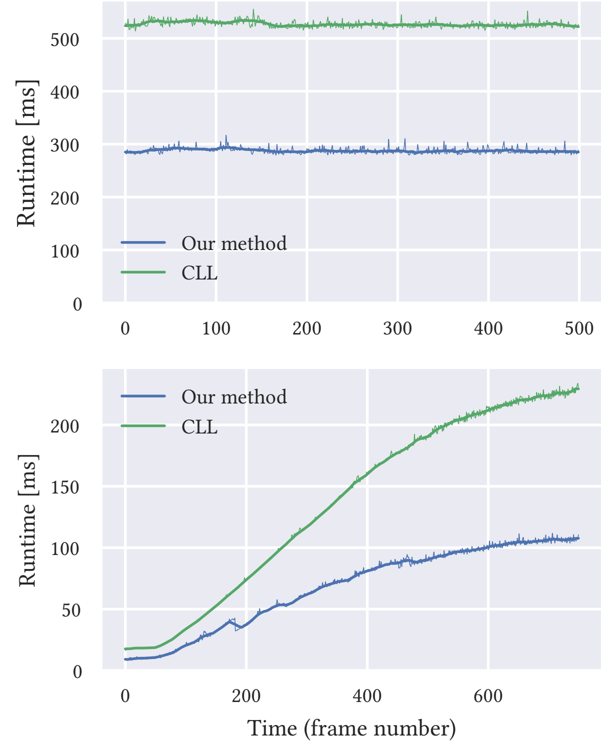


Fig. 11. Comparison between the total runtime of our method and the CLL method for the Beach (on top) and Fountain (below) scenes. The moving average has been overlayed to the original data for clarity.

4.1.6 Dense vs. Sparse particle distributions. In this experiment we compare our method and the CLL method in two synthetic scenes, one with a dense particle distribution and one with a sparse particle distribution. In the first case, particles are sampled with the SPH sampling distance used in the other experiments which results in 30 neighbors on average. In the second case, particles will be sampled slightly further apart than support radius so particles do not have neighbors. In the latter setting, the cell size in both methods is chosen to be the support radius so that each cell contains just one particle. The results can be seen in Fig. 10.

In general, we can see quite consistent runtimes when scaling up the number of particles, from 175 thousand to up to 27 million, for both methods. Our method solves the dense problem 1.65 times faster than the sparse one. The CLL method suffers much more since it fails to adapt the task sizes and it can solve the dense problem 3.24 times faster than the sparse one. Our method was on average 1.95 times faster than CLL for the dense problem.

4.1.7 Time series. In Fig. (11) we compare the total neighborhood search runtime per iteration between our method and the CLL method for the Beach and Fountain scenes. In the Beach Scene, both methods have very consistent execution times throughout the simulation. Our method presents a speedup of 1.87 over the CLL method. On the other hand, in the Fountain Scene, the speedup ranges from 1.6 to 2.2, with an average of 1.9. This evolution is due to the increasing number of particles and their changing distribution in space over time. To demonstrate that the larger cell size alone is not responsible for the observed speedup, we additionally run the experiment setting the cell size to the support radius. Our method is still 1.6 times faster on average in this case.

For reference, in the Beach Scene our neighborhood search took on average 5.6% (258 ms) of the total simulation time (4587 ms) per simulation step. Also in the Beach Scene, the extra memory that our method needed over the CLL approach, the space required to allocate the octree, was less than 40 MB. The particle positions themselves took 103 MB.

4.2 Multi-Resolution Neighborhood Search

We demonstrate the applicability of our neighborhood search method in the context of multi-resolution SPH simulations.

4.2.1 Comparison with CLL for variable support radii. To show the need for special handling when the support radius is not globally constant, we compare the CLL method against our method which can handle different support radii. This experiment consists of two large blocks of fluid particles side by side sampled regularly on a grid. One of the blocks contains 1 million small particles, with sampling distance of d_0 and a corresponding support radius of $r_0 = 2d_0$. For the other coarser grid, we use a sampling distance $d_1 = \alpha d_0$ with a corresponding support radius of $r_1 = 2\alpha d_0$, where α is the scale ratio for the particle sampling and support radius between the two grids. Particles fully inside their own grids have, on average, approximately 30 neighbors.

Since the CLL method does not support different support radii, the largest radius must be used globally. As α grows, the small particles in the fine grid will find more and more neighbors. However, these extra neighbors are not needed for the small particles since their support radius is smaller than the one used by the CLL method. This increase in the number of unwanted neighbors is cubic in the support radius and, as we can see in Table 4, it slows down the neighborhood search very significantly as the support radius ratio increases.

Table 4. Runtime comparison between the CLL method and our method for multi-resolution simulations. Since the CLL method has to globally use the largest support radius, its runtime scales very poorly in multi-resolution simulations in comparison to our method.

Scale ratio (α)	x1.0	x1.5	x2.0	x2.5	x3.0
Fine particles	1.00 M	1.00 M	1.00 M	1.00 M	1.00 M
Coarse particles	1.00 M	0.29 M	0.12 M	0.06 M	0.04 M
CLL	0.109 s	0.111 s	0.232 s	1.328 s	3.961 s
Ours	0.065 s	0.051 s	0.047 s	0.046 s	0.049 s

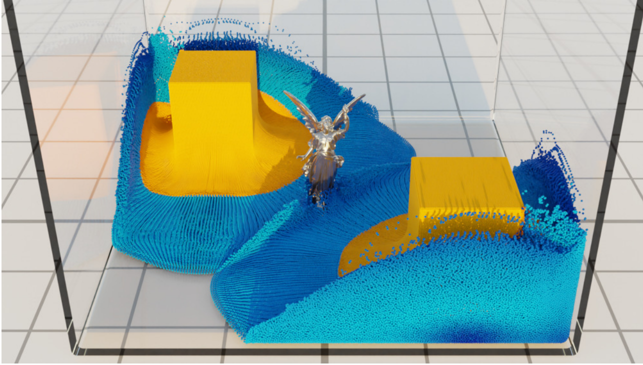


Fig. 12. Multi-resolution single-phase double dam break scene with the Stanford Lucy statue as obstacle in the center. Two fluid particle sets with different radii are used: 450 thousand coarse particles (blue color map) and 3.6 million fine particles (yellow color map). The small particles have half the radius of the large ones.

4.2.2 Multi-Resolution, Single Phase. In Fig. 12, we show the classic double dam break simulation, in this case with a center obstacle, but in the multi-resolution setting. We simulate a single fluid phase, the liquid, but we employ two different particle resolutions, each of them with a constant particle radius. The fine particle set has a slightly lower density so it settles on top after the first splash to provide high detail fluid motions while the coarse particle set stays below. The fine particle set contains 3.6 million particles of 5 mm particle radius. The coarse set has 450 thousand particles of 10 mm particle radius, twice the radius of the fine set.

4.2.3 Multi-Resolution, Fluid-Solid. In Fig. 13, we show an example of a multi-resolution simulation with one fluid phase and three elastic objects solved with the method by Kugelstadt et al. [2021]. By using fewer particles in the elastic bodies, we can alleviate the main drawback of the elastic solver, the quadratic memory complexity and initialization times. Having a coarser discretization for the elastic objects with respect to the fluid does not have a noticeable impact in the result after the surface reconstruction. There are 5.6 million particles of 10 mm particle radius in the fluid phase and a total of 12 thousand particles of 25 mm in the solid phase, a factor of 2.5 larger.

4.2.4 Multi-Resolution, Two Phase. Another use case for SPH simulations with different resolution particles is the simulation of fluids with different viscosities, modeled using the method by Weiler et al. [2018], in different resolutions. In Fig. 14, we show a two-phase

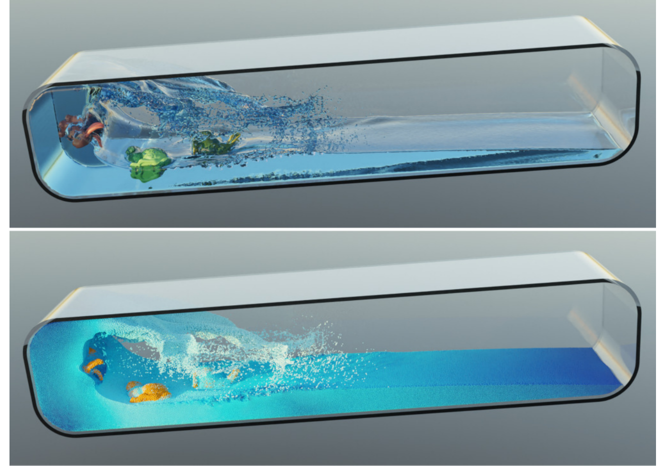


Fig. 13. Multi-resolution fluid-solid simulation of a high-resolution fluid and low-resolution deformable Stanford bunnies. A rigid box swings with 5.6 million fluid particles and a total of 12 thousand solid particles inside. Particles belonging to the deformable bodies have a 2.5 times larger radius than the ones of the fluid phase. Surface view on the top, particle view on the bottom.

Fig. 14. Multi-resolution two-phase simulation of high- and low-viscosity fluids. Six emitters pour up to 5 million fluid particles into a pool with a static rigid body (Stanford Lucy) and three highly viscous Stanford Armadillos. The 90 thousand particles of the viscous material have a 2.5 times larger radius than the fluid particles.

simulation where a highly viscous material experiences deformations at a much lower frequency than a low viscous fluid. In this setting, we can simulate the low viscous fluid with much higher particle resolution and save computational effort on the highly viscous one. The low viscous fluid is discretized with up to 5 million particles of 8.4 mm particle radius and the highly viscous one with 90 thousand particles of a particle radius of 21 mm, a factor of 2.5 larger radius.

In all non-fixed radius simulations, the relative runtime of our neighborhood search method ranged between 5% and 10% of the total simulation time.

5 LIMITATIONS

Our method inherits the usual corner cases of octrees. The recursive uniform subdivision of the octree can lead to linear chains in the tree where all particles repeatedly end up in the same child node. In simulation settings, this realistically only happens in cases where there is a very large distance between groups of particles. This would usually only cost a modest number of traversals over the particle data. However, if there are local instabilities in the simulation, it is possible for single particles to rapidly travel large distances away from the intended simulation domain. This is easily remedied by ignoring particles far outside of some predefined region of interest during the uniform grid construction.

It is also possible to detect and skip linear chains during the octree construction by jumping ahead to the smallest octree node that is large enough to contain the extended domains of all cells of the current octree node. We did not find it necessary to explicitly handle this problem for any of our simulations.

Our method may also be suboptimal for large ratios in support radii in certain configurations. For example, if most particles have a much greater radius than the particle with the smallest radius, then many grid cells will only contain a single particle with a large radius, reducing the effectiveness of the particle-to-cell aggregation. However, this is not the case in a typical SPH simulation, and our focus was in maximizing the performance on the fixed-radius case.

6 CONCLUSION AND FUTURE WORK

Neighborhood search is a critical component in SPH simulations. We present a new approach to the neighborhood search problem that departs from the long-standing assumption that the optimal cell size to use in the neighborhood search is the kernel support radius. We show that, due to the ability to generate more balanced tasks and to better utilize the CPU, octrees are better suited than uniform grids, even for fixed-radius neighborhood search problems. In addition, we show that uniform grids, which we also use as a problem reduction step, are more effective when the cell size is not chosen to be the kernel support radius, but it is instead tuned to maximize computation throughput during the brute force stage. We included discussions and extensive experimentation to show the behavior of our method in comparison to a state-of-the-art method for fixed-radius neighborhood search. Our method achieved up to 1.9 speedup in real world large SPH simulations in comparison to the well-established Cell-Linked-List method, taking up around 5% of the total SPH simulation runtime.

Additionally, we show that it is possible to exploit the adaptive nature of our octree acceleration structure to enhance the method to support variable support radii. We use this version of our neighborhood search in multi-resolution SPH simulations, including fluid-solid interaction and highly viscous fluids, with radius ratios of up to 3. The runtime of the neighborhood search was 10% or less of the total simulation runtime.

We have evaluated our method for representative fixed-radius and multi-resolution use cases. Fully adaptive SPH simulations, however, can feature particle radius ratios beyond what was shown in this paper [Winchenbach and Kolb 2020]. We believe that our method will also work well in such settings, but a comprehensive comparison is left for future work.

ACKNOWLEDGMENTS

The Armadillo, Bunny, Lucy and Asian Dragon models are courtesy of the Stanford Computer Graphics Laboratory. The presented investigations were carried out at RWTH Aachen University within the framework of the Collaborative Research Centre SFB1120-236616214 "Bauteilpräzision durch Beherrschung von Schmelze und Erstarrung in Produktionsprozessen" and funded by the Deutsche Forschungsgemeinschaft e.V. (DFG, German Research Foundation). The sponsorship and support is gratefully acknowledged.

REFERENCES

Bart Adams, Mark Pauly, Richard Keiser, and Leonidas J. Guibas. 2007. Adaptively Sampled Particle Fluids. *ACM Transactions on Graphics* 26, 3 (2007), 48.

Stefan Band, Christoph Gissler, and Matthias Teschner. 2020. Compressed Neighbour Lists for SPH. *Computer Graphics Forum* 39, 1 (2020), 531–542.

Jan Bender et al. 2022. SPlisHSPlasH Library. https://github.com/InteractiveComputerGraphics/SPlisHSPlasH.

Jan Bender and Dan Koschier. 2017. Divergence-Free SPH for Incompressible and Viscous Fluids. *IEEE Transactions on Visualization and Computer Graphics* 23, 3 (2017), 1193–1206.

Jan Bender, Dan Koschier, Tassilo Kugelstadt, and Marcel Weiler. 2018. Turbulent micropolar SPH fluids with foam. *IEEE transactions on visualization and computer graphics* 25, 6 (2018), 2284–2295.

Jan Bender, Tassilo Kugelstadt, Marcel Weiler, and Dan Koschier. 2020. Implicit Frictional Boundary Handling for SPH. *IEEE Transactions on Visualization and Computer Graphics* 26, 10 (2020), 2982–2993.

Mathieu Desbrun and Marie-Paule Cani. 1999. *Space-time adaptive simulation of highly deformable substances.* Ph. D. Dissertation. INRIA.

J. M. Domínguez, A. J. C. Crespo, M. Gómez-Gesteira, and J. C. Marongiu. 2010. Neighbour lists in smoothed particle hydrodynamics. *International Journal for Numerical Methods in Fluids* 67, 12 (nov 2010), 2026–2042.

Robert A Gingold and Joseph J Monaghan. 1977. Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly notices of the royal astronomical society* 181, 3 (1977), 375–389.

Christoph Gissler, Andreas Henne, Stefan Band, Andreas Peer, and Matthias Teschner. 2020. An Implicit Compressible SPH Solver for Snow Simulation. *ACM Transactions on Graphics* 39, 4 (Aug. 2020), 1–16.

Simon Green. 2010. Particle simulation using CUDA. *NVIDIA whitepaper* 6 (2010), 121–128.

Julian Gross, Marcel Köster, and Antonio Krüger. 2019. Fast and Efficient Nearest Neighbor Search for Particle Simulations. In *Computer Graphics and Visual Computing (CGVC)*. The Eurographics Association.

Takahiro Harada, Seiichi Koshizuka, and Yoichiro Kawaguchi. 2007. Sliced data structure for particle-based simulations on GPUs. In *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia - GRAPHITE '07*. ACM Press.

Lars Hernquist and Neal Katz. 1989. TREESPH-A unification of SPH with the hierarchical tree method. *The Astrophysical Journal Supplement Series* 70 (1989), 419–446.

Christopher Jon Horvath and Barbara Solenthaler. 2013. *Mass Preserving Multi-Scale SPH.* Technical Report.

Markus Ihmsen, Nadir Akinci, Markus Becker, and Matthias Teschner. 2011. A parallel SPH implementation on multi-core CPUs. *Comput. Graph. Forum* 30 (03 2011), 99–112.

Markus Ihmsen, Jens Cornelis, Barbara Solenthaler, Christopher Horvath, and Matthias Teschner. 2014a. Implicit Incompressible SPH. *IEEE Transactions on Visualization and Computer Graphics* 20, 3 (mar 2014), 426–435.

Markus Ihmsen, Jens Orthmann, Barbara Solenthaler, Andreas Kolb, and Matthias Teschner. 2014b. SPH Fluids in Computer Graphics.

Dan Koschier, Jan Bender, Barbara Solenthaler, and Matthias Teschner. 2022. A Survey on SPH Methods in Computer Graphics. *Computer Graphics Forum* 41, 2 (2022).

Tassilo Kugelstadt, Jan Bender, José Antonio Fernández-Fernández, Stefan Rhys Jeske, Fabian Löschner, and Andreas Longva. 2021. Fast Corotated Elastic SPH Solids with Implicit Zero-Energy Mode Control. *Proc. ACM Comput. Graph. Interact. Tech.* 4, 3, Article 33 (Sept. 2021), 21 pages.

JJ Monaghan. 1992. Smoothed Particle Hydrodynamics. *Annual Review of Astronomy and Astrophysics* 30, 1 (1992), 543–574. arXiv:arXiv:1007.1245v2

Guy M Morton. 1966. A computer oriented geodetic data base and a new technique in file sequencing. (1966).

Andreas Peer, Christoph Gissler, Stefan Band, and Matthias Teschner. 2017. An Implicit SPH Formulation for Incompressible Linearly Elastic Solids. *Computer Graphics Forum* 37, 6 (dec 2017), 135–148.

Brandon Pelfrey and Donald House. 2010. Adaptive Neighbor Pairing for Smoothed Particle Hydrodynamics. In *Advances in Visual Computing*. Springer Berlin Heidelberg, 192–201.

Barbara Solenthaler and Markus Gross. 2011. Two-Scale Particle Simulation. *ACM Transactions on Graphics* 30, 4, Article 81 (jul 2011), 8 pages.

Tetsuya Takahashi, Yoshinori Dobashi, Issei Fujishiro, Tomoyuki Nishita, and Ming C. Lin. 2015. Implicit Formulation for SPH-based Viscous Fluids. *Computer Graphics Forum* 34, 2 (may 2015), 493–502.

Min Tang, Zhongyuan Liu, Ruofeng Tong, and Dinesh Manocha. 2018. PSCC: Parallel Self-Collision Culling with Spatial Hashing on GPUs. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1, 1 (jul 2018), 1–18.

Loup Verlet. 1967. Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Physical Review* 159, 1 (jul 1967), 98–103.

G. Viccione, V. Bovolin, and E. Pugliese Carratelli. 2008. Defining and optimizing algorithms for neighbouring particle identification in SPH fluid simulations. *International Journal for Numerical Methods in Fluids* 58, 6 (oct 2008), 625–638.

Marcel Weiler, Dan Koschier, Magnus Brand, and Jan Bender. 2018. A Physically Consistent Implicit Viscosity Solver for SPH Fluids. *Computer Graphics Forum* 37, 2 (2018), 145–155.

James S. Willis, Matthieu Schaller, Pedro Gonnet, Richard G. Bower, and Peter W. Draper. 2018. An Efficient SIMD Implementation of Pseudo-Verlet Lists for Neighbour Interactions in Particle-Based Codes. *Advances in Parallel Computing* 32 (2018).

Rene Winchenbach, Hendrik Hochstetter, and Andreas Kolb. 2016. Constrained Neighbor Lists for SPH-based Fluid Simulations.

Rene Winchenbach, Hendrik Hochstetter, and Andreas Kolb. 2017. Infinite Continuous Adaptivity for Incompressible SPH. *ACM Transactions on Graphics* 36, 4 (2017).

Rene Winchenbach and Andreas Kolb. 2020. Multi-Level Memory Structures for Simulating and Rendering Smoothed Particle Hydrodynamics. *Computer Graphics Forum* 39, 6 (2020), 527–541.

Rene Winchenbach and Andreas Kolb. 2021. Optimized Refinement for Spatially Adaptive SPH. *ACM Transactions on Graphics* 40, 1 (feb 2021), 1–15.

Xilin Xia and Qiuhua Liang. 2016. A GPU-accelerated smoothed particle hydrodynamics (SPH) model for the shallow water equations. *Environmental Modelling & Software* 75 (jan 2016), 28–43.