DCGrid: An Adaptive Grid Structure for Memory-Constrained Fluid Simulation on the GPU

WOUTER RAATELAND, TU Delft, Netherlands
TORSTEN HÄDRICH, KAUST, KSA
JORGE ALEJANDRO AMADOR HERRERA, KAUST, KSA
DANIEL T. BANUTI, UNM, USA
WOJCIECH PAŁUBICKI, AMU, Poland
SÖREN PIRK, Google Research, USA
KLAUS HILDEBRANDT, TU Delft, Netherlands
DOMINIK L. MICHELS, KAUST, KSA

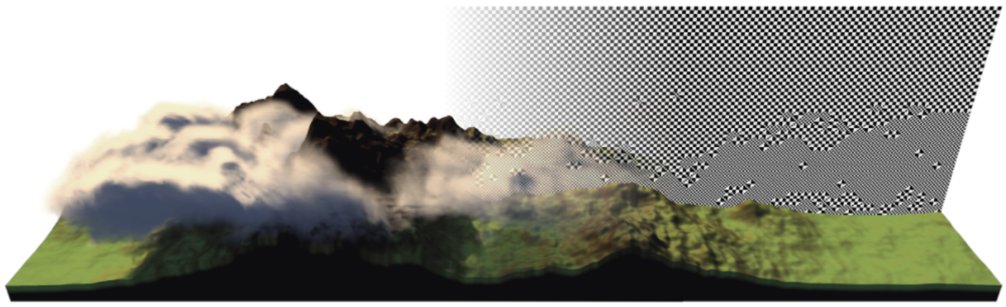


Fig. 1. Clouds simulation using DCGrid. The checkerboard pattern shows the automatically adapting grid resolution.

We introduce Dynamic Constrained Grid (DCGrid), a hierarchical and adaptive grid structure for fluid simulation combined with a scheme for effectively managing the grid adaptations. DCGrid is designed to be implemented on the GPU and used in high-performance simulations. Specifically, it allows us to efficiently vary and adjust the grid resolution across the spatial domain and to rapidly evaluate local stencils and individual cells in a GPU implementation. A special feature of DCGrid is that the control of the grid adaption is modeled as an optimization under a constraint on the maximum available memory, which addresses the memory limitations in GPU-based simulation. To further advance the use of DCGrid in high-performance simulations, we complement DCGrid with an efficient scheme for approximating collisions between fluids and static solids on cells with different resolutions. We demonstrate the effectiveness of DCGrid for smoke flows and complex cloud simulations in which terrain-atmosphere interaction requires working with cells of varying resolution and rapidly changing conditions. Finally, we compare the performance of DCGrid to that of alternative adaptive grid structures.

Authors' addresses: Wouter Raateland, TU Delft, Netherlands, wouterraateland@gmail.com; Torsten Hädrich, KAUST, KSA, torsten.haedrich@gmail.com; Jorge Alejandro Amador Herrera, KAUST, KSA, jorge.amadorherrere@kaust.edu.sa; Daniel T. Banuti, UNM, USA, daniel@banuti.de; Wojciech Pałubicki, AMU, Poland, wp06@amu.edu.pl; Sören Pirk, Google Research, USA, soeren.pirk@gmail.com; Klaus Hildebrandt, TU Delft, Netherlands, k.a.hildebrandt@tudelft.nl; Dominik L. Michels, KAUST, KSA, dominik.michels@kaust.edu.sa.

CCS Concepts: • **Computing methodologies → Animation**; • **Information systems → Record and block layout**.

Additional Key Words and Phrases: Fluid simulation, Adaptive grid, Hierarchical solver, Real-time simulation

ACM Reference Format:
Wouter Raateland, Torsten Hädrich, Jorge Alejandro Amador Herrera, Daniel T. Banuti, Wojciech Pałubicki, Sören Pirk, Klaus Hildebrandt, and Dominik L. Michels. 2022. DCGrid: An Adaptive Grid Structure for Memory-Constrained Fluid Simulation on the GPU. *Proc. ACM Comput. Graph. Interact. Tech.* 5, 1, Article 3 (May 2022), 14 pages. https://doi.org/10.1145/3522608

1 INTRODUCTION

Grid-based fluid simulations are widely used in the entertainment and advertising industries to create detailed and physically plausible visual effects such as splashes, sprays, smoke, and fire. These simulations require finely resolved grids to achieve the desired level of detail. In other areas, such as the simulation of weather phenomena, high-resolution grids are needed to adequately resolve the interaction with geometrically modeled environments in the simulation. The grid resolutions needed for these simulations lead to high-dimensional and complex computations and therefore place high demands on the performance of the simulation frameworks.

In this paper, we introduce Dynamic Constrained Grid (DCGrid), a novel GPU-based multigrid data structure for high-performance fluid simulation that supports adaptive topology. Our approach increases computational and memory savings compared to regular static grids, and requires less a priori knowledge with respect to the temporal evolution of the solution. It respects key defining properties and characteristic global features of the fluid flow resulting in physically plausible results. Supporting adaptive topology, maintaining a multigrid structure, and being GPU-based sets DCGrid apart from alternative grid data structures such as SPGrid [Setaluri et al. 2014], which is an adaptive multigrid but not GPU-based, and GDVB [Hoetzlein 2016; Wu et al. 2018], which is GPU-based and adaptive but does not maintain a multigrid structure.

DCGrid stores the whole simulation domain on the GPU and is designed to allow for fast execution of operations important for fluid simulation, such as accessing values at individual cells and on local stencils. In our comparisons to CPU-based data structures, we observe a speedup of more than an order of magnitude. As a multigrid structure, DCGrid manages a hierarchy of uniform grids and allows for efficient access to values and stencils at different levels of the hierarchy. The hierarchical structure of DCGrid facilitates the use of multigrid solvers for the pressure projection in fluids simulations. Since the memory required to store grids grows quickly with their resolution, large-scale simulations can easily exceed memory limits on the GPU. To use memory efficiently, DCGrid is an adaptive grid structure where the grid resolution varies spatially. The uniform grids in DCGrid's hierarchy are all sparsely populated, so that storage demand depends on the number of populated cells rather than the resolution of the grid. During simulations, grid topology is adapted at each timestep in an efficient way. The grid adaption is controlled by an algorithm that optimizes the grid topology based on the maximum available memory.

Comparisons to the alternative adaptive grid structures SPGrid and GDVB demonstrate that DCGrid can execute stencil and streaming operations substantially faster. We extend this comparison by benchmarking the performance of the grids on smoke simulations. Furthermore, we include a supplementary material in which we extend the suitability of DCGrid in high-performance simulation, especially real-time simulation, by proposing a scheme for the approximation of collisions between fluids and static solids on cells with different resolutions. This scheme does not enforce high-resolution cells at boundaries so that more cells can be used elsewhere. As an application, we use DCGrid in complex cloud simulations where the interaction between terrain and atmosphere requires working with cells of varying resolution and quickly changing conditions.

In summary, we present a hierarchical, adaptive, and GPU-based grid structure for high-performance fluid simulations and make the following main contributions:

- a memory-efficient data structure based on a hierarchy of sparsely populated uniform grids, highly suitable for fluid simulation;
- an efficient, optimization-based, local topology adaptation method that respects memory constraints.

In addition, we demonstrate in the supplementary material the applicability of DCGrid to complex simulation scenarios by integrating DCGrid into a framework for meteorological simulations.

2 BACKGROUND

In this section, we review closely related work and discuss the SPGrid structure, which is the basis for DCGrid.

2.1 Related Work

Fluid simulations solve a discretization of the Navier–Stokes equations, usually by finite differences [Foster and Metaxas 1996], and need to deal with velocity advection and pressure projection. An unconditionally stable semi-Lagrangian scheme for solving velocity advection was introduced in by Stam [1999] and has been the basis for grid-based fluid solvers for phenomena such as smoke [Fedkiw et al. 2001] and water [Foster and Fedkiw 2001]. Fluid Implicit Particle (FLIP) and Point in Cell (PIC) methods combine particle and grid representations of the fluids and solve advection using the particles and the pressure projection on the grid [Boyd and Bridson 2012; Ferstl et al. 2016; Zhu and Bridson 2005]. The pressure projection is in many cases the bottleneck of fluid solvers since a large linear system needs to be solved at every time step. These systems can be solved by the preconditioned conjugate gradient method [Bridson 2008; Foster and Fedkiw 2001]. Due to the regular structure of the lattices, multigrid methods, which are particularly effective for large systems, can be used to directly solve the pressure projection [Molemaker et al. 2008] or to build effective preconditioners for conjugate gradients schemes [McAdams et al. 2010]. Multigrid solvers are also effective for real-time fluid simulation [Chentanez and Müller 2011] and the simulation of viscous liquids [Aanjaneya et al. 2019]. An alternative to grid-based simulations are purely particle-based methods such as Smoothed Particle Hydrodynamics [Cornelis et al. 2014; Gissler et al. 2019].

The structured nature of uniform grids offers benefits when used in fluid simulations, such as fast stencil access and simple memory management. On the other hand, their uniform nature dictates a uniform distribution of computational resources over the fluid domain. This uniform distribution is often an inaccurate representation of the distribution of interesting features in a fluid domain (e.g. areas with high vorticity). To simulate features of high interest in the most accurate way possible, we would like to focus computational resources on these features. Multiple works have presented different alternations to the uniform grid that achieve this. Losasso et al. [2004] was the first to introduce a fluid simulation on an octree structure. Museth [2013] introduced OpenVDB, a sparse data structure organized as a tree with a high branching factor. OpenVDB is especially suited for simulating values at a uniform resolution. A GPU implementation of OpenVDB, GVDB has been developed by Hoetzlein [2016]. Wu et al. [2018] developed a FLIP simulation on GVDB, capable of simulating and rendering scenes with tens of millions of particles, where the scene topology automatically adapts to the FLIP particles. Recently, Museth [2021] introduced NanoVDB, a port of OpenVDB that can be instantiated in both CPU and GPU contexts. It currently only supports static topology and is able to perform tasks like rendering and collision handling very efficiently. Setaluri et al. [2014] introduced SPGrid a sparse paged grid structure. They also introduce the SPGrid pyramid, which achieves spatial adaptivity by combining multiple SPGrids in a hierarchy. SPGrid has been used in multiple large-scale fluid simulations [Aanjaneya et al. 2017; Liu et al. 2016] and also in an MPM context [Hu et al. 2018]. Gao et al. [2019] used a non-hierarchical, modified version of SPGrid in a GPU context. Recently, Xiao et al. [2020] has designed an alternative approach to adaptivity based on an Adaptive Staggered-Tilted (AST) grid. This method augments a primary grid with a secondary, overlapping grid with tilted cells (i.e., rotated by 45° in 2D). By scaling each tilted cell individually, they achieve fine-grained adaptation on a uniform grid. The secondary grid typically imposes a runtime penalty of only a few percent. Nielsen et al. [2020] proposed an automatic optimization-based grid refinement algorithm for smoke simulations. Their algorithm uses a global voxel limit and runs in $O(n)$ time, with respect to the number of allocated voxels. Because we focus on a high-performance grid structure to be implemented on the GPU, we have to be more restrictive and use voxel limits per mipmap level, requiring a different adaptation algorithm. The widely used application OpenFOAM includes an Adaptive Mesh Refinement (AMR) module. The module is not yet optimization-based. Nevertheless, it has enabled simulations with improved accuracy using less computational resources [Cooke et al. 2014; Lapointe et al. 2020].

2.2 SPGrid

SPGrid is a data structure for sparse Cartesian grids. A SPGrid pyramid uses a hierarchy of SPGrids to model a grid with cells of varying resolution. It supports voxels at each hierarchy level and thereby enables different levels of detail (LOD). SPGrid allocates all grids in the hierarchy in virtual memory. It achieves adaptivity by only allocating the active parts of the grids in physical memory. To do so, SPGrid relies on specific properties of Haswell processors.

SPGrid structures each grid in a hierarchy in blocks of cells such that one block occupies exactly one 4KB memory page. This results in blocks of 4^3 or $4^2 \times 8$ cells, depending on the number of values simulated per cell. SPGrid allocates blocks in virtual memory following a Morton encoding (Z-order curve) for optimal data locality. This structure enables constant time translations between a block's memory address and its location in the grid. These constant time translations allow for the efficient execution of random and stencil accesses. To efficiently compute gradients and Laplacians, SPGrid introduces ghost cells. Let cell C_I^l be a cell with index I located at grid l in the hierarchy. Then C_I^l is a ghost cell when:

- C_I^l is not active at level l,
- C_I^l neighbors a cell that is active at level $s \leq l$,
- and there exists a coarse parent of C_I^l at level $l^* > l$ that is active.

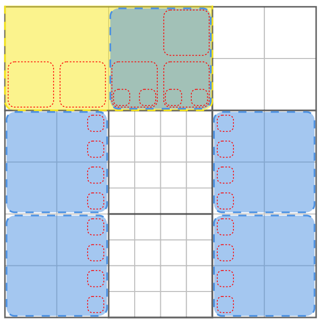


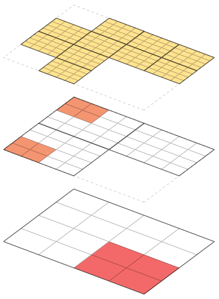
Fig. 2. Example of ghost cells and additional blocks allocated in a SPGrid hierarchy with an irregular domain. Red dashed squares indicate required ghost cells. Blue, green and yellow rectangles indicate allocated blocks on different levels of the hierarchy.

Before computing a gradient or a Laplacian, SPGrid upsamples values from coarse parent cells into their corresponding ghost cells. With these values in place, it computes gradients and stencils as if the grid were uniform. After performing the computations, values in the ghost cells are copied back into their coarse parent cells. This way, all grid cells are updated without computing values multiple times. While ghost cells are an effective means for fast computations, they do come with a problem. The problem is that SPGrid allocates cells in blocks. Consequently, allocating only a few ghost cells may require the allocation of multiple complete blocks. This problem is most visible in irregular domains. Figure 2 shows a possible SPGrid hierarchy. In this configuration, SPGrid allocates 6 blocks for active cells. It also allocates 6 extra blocks just for ghost cells. Adapting topology in a SPGrid hierarchy works by instantiating a new instance with the desired topology and copying values accordingly. This workflow is necessary because some architectures are unable to forget that memory pages have been touched.

3 DATA STRUCTURE

Grid-based fluid simulations rely on random and stencil access for most of their operations, including Semi-Lagrangian advection, diffusion, and projection. Therefore, to achieve high-performance simulations, our data structure should perform well on these types of access. The regularity of Cartesian grids gives us the potential to perform fluid operations in parallel. To utilize this potential, our structure should be usable in a GPU context. Additionally, fluid simulations often operate on multiple data channels, such as densities, velocities, and auxiliaries and so our structure should support this.

We propose DCGrid, a data structure based on a hierarchy of L sparsely populated uniform grids $G_0, \ldots, G_{L-1}$ (Figure 3). To achieve cache coherency, we store cell data in blocks. Each block consists of 2^3 subblocks, which in turn consist of 2^3 cells each. The resolution between two adjacent levels in the grid hierarchy differs by a factor of 2. Thus, a block allocated on grid G_l spans the same volume as a subblock on grid G_{l+1}. As this scaling factor resembles a mipmap, we call grid G_l mipmap level l.


(a) Hierarchy of sparse uniform grids. Colored cells are active.

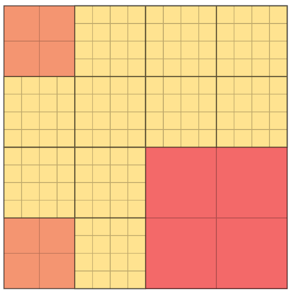

(b) Flattened grid hierarchy into one dense multi-resolution grid, showing only active cells.

Fig. 3. Two-dimensional slice of the same hierarchy of sparsely populated uniform grids. The thicker lines indicate block boundaries.

To satisfy memory constraints, we define a global limit on the number of blocks that can be allocated simultaneously $B_{\mathtt{max}}$. To improve read performance, we also impose a similar limits $B_{\mathtt{max},l}$ on each mipmap level $0 \leq l < L$, such that $\sum_{l \leq 0 < L} B_{\mathtt{max},l} = B_{\mathtt{max}}$.

Random Access. DCGrid allocates blocks of cells in a linear span of memory. A simple GPU hash table `h` links the position of blocks in the grid to their index in this linear memory span. Let b be a block located at $\mathbf{p}_b$, then the hash used by `h` is the 32 bit Morton encoding [Morton 1966] of $\mathbf{p}_b = \lfloor \mathbf{p}/4 \rfloor$. This way of hashing means that a DCGrid instance can have a maximum resolution of $2^{11} \times 2^{11} \times 2^{10}$ blocks, which is equivalent to $8192 \times 8192 \times 4096$ cells. We use a key space of `h.size` $= 4B_{\mathtt{max},l}$ entries for each grid G_l. With this size, we can find the index of the block in the linear memory span I_b in $O(1)$ time using a hash table lookup.

DCGrid stores data for subblocks and cells in separate linear memory spans. These memory spans are ordered equivalently to the memory span used for block data. To illustrate this, let b be the block stored at index I_b. DCGrid stores data for the 8 subblocks contained in b at indices

$\{2^3 I_b, 2^3 I_b + 1, \ldots, 2^3 I_b + (2^3 - 1)\}$ in the following order:

$$I_s = 2^3 I_b + \texttt{0x100}\left(\left\lfloor \frac{\mathbf{p}.x}{2} \right\rfloor \mod 2\right) + \texttt{0x010}\left(\left\lfloor \frac{\mathbf{p}.y}{2} \right\rfloor \mod 2\right) + \texttt{0x001}\left(\left\lfloor \frac{\mathbf{p}.z}{2} \right\rfloor \mod 2\right). \quad (1)$$

DCGrid stores data for the cells contained in subblock s at indices $\{2^3 I_s, 2^3 I_s + 1, \ldots, 2^3 I_s + (2^3 - 1)\}$ in a similar order:

$$I_c = 2^3 I_s + \texttt{0x100}(\mathbf{p}.x \mod 2) + \texttt{0x010}(\mathbf{p}.y \mod 2) + \texttt{0x001}(\mathbf{p}.z \mod 2). \quad (2)$$

In summary, to find the index I_c of cell c located at position $\mathbf{p}$ in grid G_l, we perform three transformations. We first find the index I_b of the block containing c using a hash table lookup. Then, we find the index I_s of the subblock containing c using a simple transformation. Finally, we find the index I_c of cell c using a similar transformation.

Stencil Access. The stencil operations that we focus on are stencils with dimension 3^3. The logical ordering of cells within blocks makes it trivial to compute stencils for cells in the interior of blocks. Computing stencils for cells on the border of blocks, however, requires accessing neighboring blocks. Naively, this would require multiple random access operations and thus multiple hash table lookup operations for each stencil operation.

To more efficiently access adjacent cells in different blocks, we pre-compute an apron of cell indices directly neighboring each block (Figure 4). Pre-computing of apron cell indices is performed once, during grid initialization. As the higher resolution mipmap levels will be sparsely occupied, not all blocks will have direct neighbors at the same mipmap level. We could directly include neighbors on other mipmap levels in the apron. Doing so would lead to a different amount of cells neighboring each block. Consequently, we get in non-uniform stencils. Non-uniform stencil computations on a GPU might lead to thread divergence, which reduces their efficiency. Therefore, we want to avoid non-uniform stencils. Instead, we implement an approach that retains the benefit of uniform stencils and works with sparsely occupied grids. To do so, we restrict the layout of the grids in the hierarchy using two rules:

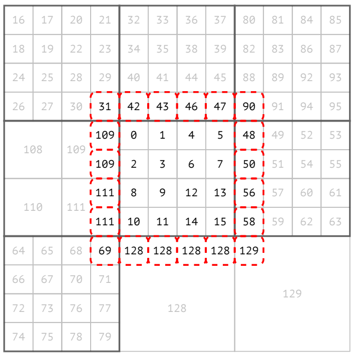


Fig. 4. Apron cell indices as calculated for the central block.

(1) For mipmap levels $0 \le l < L - 1$ and for each cell $c \in G_l$, there must exist a parent cell c_p at position $2\lfloor \mathbf{p}/2 \rfloor$ on grid G_{l+1}. We also call c a child cell of c_p.
(2) The lowest resolution grid, G_{L-1}, should be densely allocated.

These restrictions ensure that $G_0 \subseteq G_1 \subseteq \cdots \subseteq G_{L-2} \subseteq G_{L-1}$. To compute the apron cell indices for a block at grid G_l, we iterate over all positions directly adjacent to that block. At each position $\mathbf{p}$, if $\mathbf{p}$ lies inside the domain, we search for a cell in G_l first. If this cell does not exist, we search for a cell at position $\mathbf{p}$ in increasingly coarser mipmap levels. As G_{L-1} covers the complete domain, this process always finds a cell at some level. We now treat all cell indices in aprons as if they originate from the same mipmap level. This process yields a uniform, and thus a performant stencil operation.

Restriction and Prolongation. By restricting the layout of the grid hierarchy, most of the domain will be covered by multiple cells. Specifically, if cell c covers point $\mathbf{p}$ in the domain, then its parent cell also covers that point. Performing fluid simulation operations on multiple cells covering the same point would be a waste of computational resources. Therefore operations are only performed on the highest resolution cell that covers any point. This cell is called active.

Any cell that is the parent cell of another cell is called inactive. To get the correct values in the inactive cells, we perform a simple down-sampling routine called restriction (Figure 5a). Starting at mipmap level G_0, we average the values of cells sharing the same coarse parent cell. We store this average in the parent cell and repeat on each coarser grid until all cells have values. The inverse operation of restriction is prolongation (Figure 5b). This operation traverses the grid hierarchy in the other direction, from low to high resolution, and transfers values from parent cells into their children.

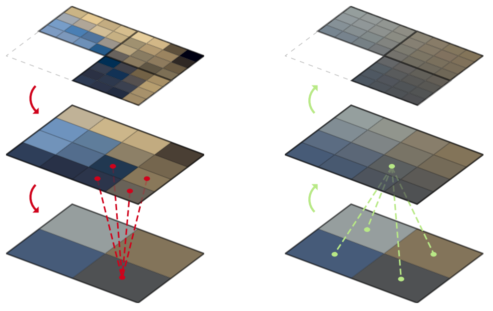

(a) Restriction operation updating cell values by averaging the values of their fine child cells. (b) Prolongation of restricted values back to high-resolution grid.

Fig. 5. Restriction and prolongation operations performed after each other on the same data.

4 TOPOLOGY ADAPTATION

We developed an optimization-based local topology adaptation method. Our method distributes computing power and memory usage over the simulation domain by distributing active cells over mipmap levels. In particular, our method distributes active cells according to each cell's priority score $p(c)$, such that active cells on higher-resolution mipmap levels have higher priority scores than active cells on lower-resolution levels. In other words:

$$\forall c_1, c_2 : (\texttt{is\_active}(c_1) \land \texttt{is\_active}(c_2) \land \texttt{mipmap\_level}(c_1) < \texttt{mipmap\_level}(c_2)) \\ \Leftrightarrow p(c_1) > p(c_2)\,. \tag{3}$$

When distributing cells, we need to account for the block limit $B_{\texttt{max},l}$ per mipmap level and the restricted grid layout. A cell's priority score $p(c)$ can be any user-defined function. Potentially useful priority scores include the velocity gradient, proximity to the camera, or proximity to a boundary. Unless stated otherwise, we use the vorticity magnitude of a cell as its priority score.

Basic Operations. Before introducing our algorithm for topology adaptation, we need to establish the basic operations that constitute the algorithm: subblock refinement and block coarsening. To

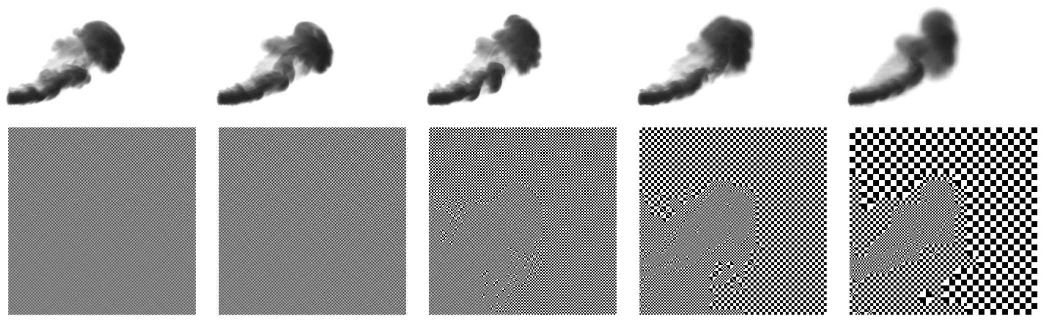

(a) Uniform, 16.8M cells. (b) DCGrid, 16.8M active cells. (c) DCGrid, 4.2M active cells. (d) DCGrid, 840K active cells. (e) DCGrid, 168K active cells.

Fig. 6. Smoke simulation run for 150 timesteps under different memory limits. Effective resolution is 256^3. Each checkerboard square represents one cell.

refine subblock s located at mipmap level G_l, we insert a new block $b_{\texttt{refined}}$ at mipmap level G_{l-1} spanning the same volume as s. This insertion uses CUDA atomic operations (`atomicCAS`) to allow for parallel insertions. When a block already exists at position $\mathbf{p}$, or if there is no space left at mipmap level G_{l-1}, the operation is canceled. Otherwise, subblock s is marked as inactive, and block $b_{\texttt{refined}}$ is marked as active and positioned in the grid. Finally, the values in block $b_{\texttt{refined}}$ are initialized by prolonging the values from subblock s.

To coarsen a block b located at mipmap level G_l, we first check if block b is active. If not, then there exists a higher-resolution block at the same position. In this case, we cannot delete b, as that would leave a gap in the grid hierarchy. If b is active, we look up the subblock s that resides at mipmap level $l+1$ at the same position. Because of our restricted grid hierarchy layout, s always exists when $l < L-1$. We mark s as active and delete b from the hash table. Again, this hash table deletion uses CUDA atomic operations and can be run in parallel.

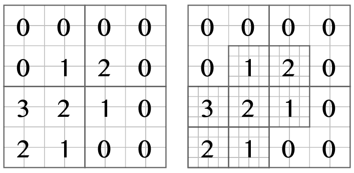


Fig. 7. Adaptation setup. First, a coarse grid is initialized. Then after each timestep, the subblocks with the highest priority scores are refined, until the finest grid is allocated.

A hash table deletion does not clear the previously occupied key in the hash table, leaving the key unusable. After many insertions and deletions, this will slow down lookups in the hash table. Eventually, the hash table will run out of space. To ensure that the hash table performs well, we can refill it periodically. The time required for refilling is insignificant compared to the total time topology adaptation takes.

Adaptation Setup. To initialize a DCGrid instance with L mipmap levels, we define a global block limit B_{max} and use that to determine the block limits $B_{\mathsf{max},l}$ for each level $l \in \{0, \ldots, L-1\}$. In our experiments we used the following division:

(1) Set the remaining block budget B to be B_{max}.
(2) For each hierarchy level grid $l = L-1$ down to 0:
(3) Set $B_{\mathsf{max},l}$ to the smallest value of $B/(l+1)$ or the number of blocks required to densely allocate G_l.
(4) Subtract $B_{\mathsf{max},l}$ from the remaining budget B.

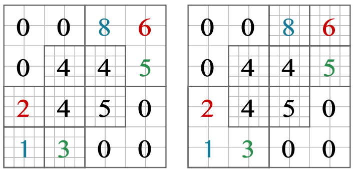


Fig. 8. Block re-arrangement. After each timestep, for each adjacent pair of grids in the hierarchy is considered. Fine blocks with low priority scores are matched with coarse subblocks with high priority scores. Then the fine blocks with low scores moved to the locations of the coarse subblocks with high scores.

With this division, some mipmap levels are always densely allocated. In particular, let $G_0, \ldots, G_{L-1}$ be a DCGrid instance with L mipmap levels, then there is a mipmap level l, so that for each $l \le l' < L$, block limit $B_{\mathsf{max},l'}$ is such that $G_{l'}$ can be densely allocated. When initializing a DCGrid instance, we immediately allocate these mipmap levels densely.

Cells on the other mipmap levels $(G_0, \ldots, G_{l-1})$ are allocated iteratively by refining subblocks from $G_1, \ldots, G_l$ respectively. Let G_l, $1 \le l < L-1$ be a mipmap level on which some blocks are allocated already, i.e., $B_l > 0$. If $B_{l-1} < B_{\mathsf{max},l-1}$, then there still space for $N_{l-1} = B_{\mathsf{max},l-1} - B_{l-1} > 0$ blocks on mipmap level G_{l-1}. We can now allocate $R_l = \min\{N_{l-1}, 2^3 B_l\}$ blocks on G_{l-1} be refining R_l subblocks from G_l. To select subblocks from G_l, we perform this refinement step only once after each timestep. We first calculate the priority score of each subblock on G_l using a simple CUDA kernel that averages the priority scores of each cell in the subblock. We select the R_l subblocks with the highest priority scores for refinement. Selecting these subblocks is an instance of the k-selection problem. For this problem, an $O(n)$ algorithms exists [Blum et al. 1973]. We solve this problem using the Quick Select algorithm, as this has expected runtime $O(n)$ and is often faster in practice.

Re-arrangement. In a typical fluid simulation, the distribution of values throughout the domain changes over time. Hence, the parts of the domain that deserve the most attention also change over time. To account for these changes, we allow for re-arrangement of blocks via Algorithm 1.

We determine which blocks should be rearranged per mipmap level G_l, starting at the highest resolution mipmap level G_0. First, we calculate the priority scores of each block in G_l and each subblock in G_{l+1} using a single CUDA kernel. We then find the set of active blocks $B_{\texttt{coarsen}} \subseteq G_l$ and the set of active subblocks $S_{\texttt{refine}} \subseteq G_{l+1}$. On the CPU, we sort $B_{\texttt{coarsen}}$ ascending and $S_{\texttt{refine}}$ descending by their priority scores, and iterate over each pair in order. For each pair, we check if the priority score of the block is lower than the priority score of the subblock. If this is the case, then this pair violates our adaptation objective (Equation 3). To move closer towards fulfilling the adaptation objective, we now coarsen the block and refine the subblock. The actual coarsening and refinement happen in parallel via a CUDA kernel.

Sorting all blocks and subblocks is an expensive operation, especially as the number of blocks grows. Optimally, we would only sort the v_l objective violating pairs. To reduce the number of blocks and subblocks to sort, introduce a move limit m_l, per mipmap level. Move limit m_l indicates that at most m_l subblocks can be refined and m_l blocks can be coarsened at mipmap level l in one execution of the re-arrangement algorithm. To find the blocks and subblocks that have to be coarsened and refined, we now only need to find the m_l blocks with the lowest priority scores and the m_l subblocks with the highest priority scores. This only requires partial sorting of $B_{\texttt{coarsen}}$ and $S_{\texttt{refine}}$ up to m_l elements, which can be performed in $O(\max\{|B_{\texttt{coarsen}}|, |S_{\texttt{refine}}|\} \log m_l)$ time.

ALGORITHM 1: Block re-arrangement.

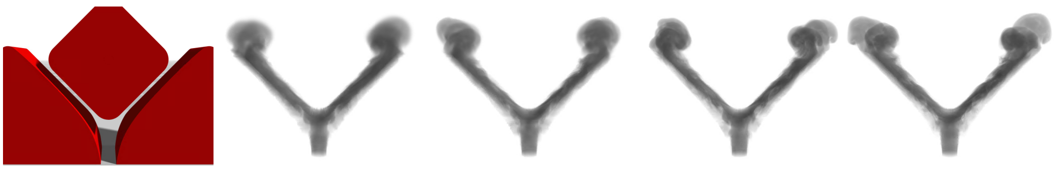
```
Input: Grids $G_0, \dots, G_{L-1}$, move limit per grid $m_l$.
for $l = 0 \dots L-2$ do
    $\forall s \in G_{l+1}, p_s \leftarrow$ calc_subblock_score($s$)
    $\forall b \in G_l, p_b \leftarrow$ calc_block_score($b$)
    $B_{\texttt{coarsen}} \leftarrow \{b \in G_l | \texttt{is\_active}(b)\}$
    $S_{\texttt{refine}} \leftarrow \{s \in G_{l+1} | \texttt{is\_active}(s)\}$
    partial_sort$_{\texttt{ascending}}(B_{\texttt{un\_refine}}, m_l)$
    partial_sort$_{\texttt{descending}}(S_{\texttt{refine}}, m_l)$
    $v_l \leftarrow 0$
    for $(b, s) \in$ zip($B_{\texttt{coarsen}}, S_{\texttt{refine}}$) do
        if $p_b < p_s$ then
            coarsen($b$)
            refine($s$)
            $v_l \leftarrow v_l + 1$
    $m_l \leftarrow$ next_move_limit($m_l, v_l$)
```

To find reasonable limits m_l, we make three observations:

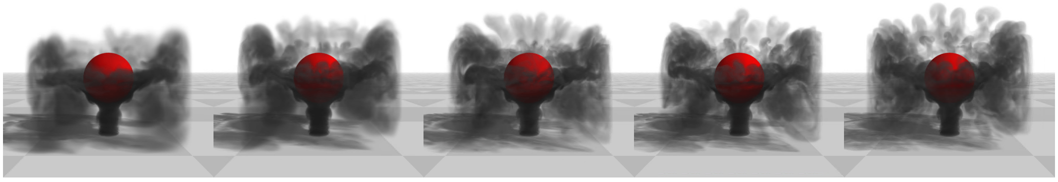
- If $m_l > v_l$, we sort more blocks and subblocks than necessary, which is a waste of time.
- If $m_l < v_l$, we sort fewer blocks and subblocks than there are objective violating pairs. Doing so leaves some objective violating pairs unchanged, which is undesirable.
- The number of objective violating pairs v_l in one timestep is often similar to v_l in the next timestep.

From these observations, we conclude that the move limit m_l should be larger than the number of objective violating pairs v_l, but the difference $m_l - v_l$ should be as small as possible. We also see that we can predict reasonable limits m_l for the next timestep using the number of objective violating pairs v_l in the current timestep. In our experiments, we found that the simple prediction `next_move_limit`$(m_l, v_l) = \max\{.8m_l, 1.5v_l\}$ gave good results.

After the re-arrangement step, we ensure that all apron cell indices point to the correct cells again. To do so, we calculate the apron cell indices for each newly inserted block and refresh the apron cell indices that pointed to refined or coarsened blocks.



(a) Domain setup. (b) DCGrid (5%). (c) DCGrid (10%). (d) DCGrid (25%). (e) Uniform.



(f) DCGrid (10%). (g) DCGrid (30%). (h) DCGrid (50%). (i) DCGrid (80%). (j) Uniform grid.

Fig. 9. Smoke flow performed on a uniform grid and on DCGrid with various block limits and thus different amounts of active cells. All percentages are relative to their respective values in the simulation on the uniform grid.

5 RESULTS

In this section, we present results evaluating our simulation framework and comparing it to other approaches. We evaluate the distinct features of DCGrid, compare DCGrid against a uniform grid and benchmark the performance of DCGrid against SPGrid and GVDB on single kernels and using end-to-end tests. We implemented DCGrid and our simulation framework on the GPU using CUDA. Unless stated otherwise, we performed all simulations on an NVIDIA® GeForce GTX 1070.

Table 1. Timings and memory usage of experiments included in this work.

Scene	Structure	Domain	Memory	Floats			Avg. Time / Timestep (ms)			
			(MB)	/ Cell	Advect	Project	Topology	Render	Rest	Total
Fig. 6a	Uniform	256^3	537	8	34	24	–	2.0	16	76
Fig. 6b	DCGrid	256^3	1177	8	50	39	8.6	9.5	22	130
Fig. 6c	DCGrid	256^3	294	8	13	9.7	7.3	12	5.7	47
Fig. 6d	DCGrid	256^3	59	8	3.9	2.5	2.8	10	1.8	21
Fig. 6e	DCGrid	256^3	12	8	1.4	0.9	1.6	7.1	1.4	12
Fig. 1	DCGrid	512×128^2	193	17	11	4.8	6.4	25	13	60
N/A	SPGrid*	$1024^2 \times 2048$	23 GB	16	26 s	525 s	–	?	24	575 s
Fig. 12	DCGrid	$1024^2 \times 2048$	6872	8	507	430	303	41	–	1280
Fig. 13a	GVDB	256^3	97	8	15	102	9.3	1.4	15	142
Fig. 13b	DCGrid	256^3	97	8	5.2	9.1	3.5	9.2	2.2	29

*Data for the SPGrid result is sourced from [Setaluri et al. 2014]. Rendering time is not included. Experiments run on an Intel Xeon E5-2670.

Topology Adaptation. To evaluate the topology adaption scheme, we compare results of four runs of the same simulation with different memory bounds on adaptive grids of effective resolution 256^3 (Figure 6). As a baseline, we also show the results obtained on the corresponding uniform grid. In all cases, the high-resolution areas in the adaptive grids follow the smoke flow. The simulation with the lowest cell limit (Figure 6e) allocates less than 1.5% of the cells used by the simulation on a uniform domain (Figure 6a). Still, it faithfully captures the global features of the smoke flow. The highest resolution adaptive simulation (Figure 6c) uses one quarter of the cells that the non-adaptive simulations use (Figure 6a, 6b). This adaptive simulation reproduces almost all the details visible in the non-adaptive simulations.

Comparison to a Uniform Grid. To evaluate the performance of the adaptivity of DCGrid, we compare memory usage and average access times against a uniform grid by performing similar simulations in both settings. In general, fluid operations on DCGrid require extra time, because they require pointer indirection. Also, after each update, DCGrid requires some extra time to accumulate changed values. The apron cell indices are the main contributor to the memory overhead. Since their size only depends on the number of cells allocated, not on the amount of data stored per cell, the memory overhead is lower in more complex simulations.
Figure 10 shows plots of memory usage and the average computation time per frame over the number of active cells in the grid. We report results for two types of simulations: smoke simulation with 8 floating-point channels per cell (blue graphs) and simulations of weatherscapes with 17 floating-point channels (red graphs). While when densely allocated, DCGrid requires more memory and yields slower runtimes than a uniform grid, this changes when not all cells are allocated. The break-even is reached when about 50% of the cells are used for 8 channels and about 60% for 17 channels. When 25% of the cells are used, DCGrid is about twice as fast and requires only half of the memory space.

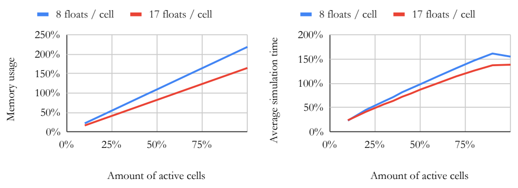
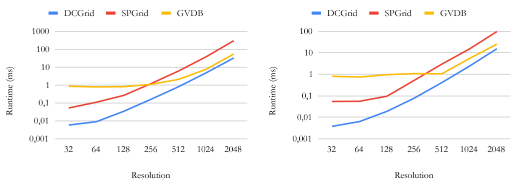


Fig. 10. Performance of DCGrid relative to a uniform grid under different configurations.

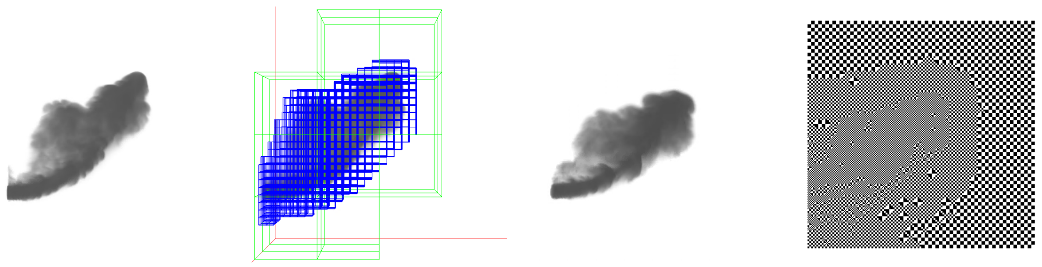
The number of active cells used in a simulation varies for different simulations. In Figure 9, we show two examples that illustrate how simulations can benefit from the adaptivity of DCGrid. The upper row shows a simulation that is localized in the simulation domain. In this case, DCGrid reaches a similar quality simulation using only 0.25 times as many cells as a uniform grid would use, requiring only half the computation time and the memory that the uniform grid requires. The lower row shows a setting in which most of the scene is occupied by some fluid. Even in this case, DCGrid can reach a simulation of similar quality using fewer cells. DCGrid reaches a similar quality simulation as the simulation in the uniform grid using 0.5 times as many cells as the uniform grid uses, requiring only slightly more memory and computation time compared to the uniform simulation. This indicates that DCGrid can be used even in non-sparse settings without sacrificing performance or quality.



(a) Stencil operation. (b) Streaming operation.

Fig. 11. Performance of DCGrid compared to SPGrid and GVDB on a stencil and streaming kernel. The kernels are performed on a narrow-band domain consisting of a spherical shell. SPGrid operations are performed on an Intel i7-3820 (4 cores).

Benchmark Comparisons. We performed two benchmark tests, streaming and stencil operations, to compare the performance of DCGrid with SPGrid [Setaluri et al. 2014] and GVDB [Hoetzlein 2016; Wu et al. 2018]. A Saxpby operation was computed for the streaming kernel and a Laplacian for the stencil kernel. The benchmarks were performed on the same sparse domain for each data



(a) GVDB, brick size 8^3. (b) DCGrid.

Fig. 13. Smoke simulation run in GVDB and DCGrid on a 256^3 grid. Middle and right scene both use 97MB memory at most.

structure. This sparse domain consists of a narrow band set of a spherical shell. Results are plotted in Figure 11. DCGrid is faster than GVDB and SPGrid at all resolutions tested.

Comparison to SPGrid. To compare DCGrid with SPGrid, we recreated one of the scenes showcased in [Setaluri et al. 2014] in a best-effort manner as not all input parameters, such as emission velocity, were available. Also, we used our own rendering engine, because SPGrid did not specify exactly how they rendered the original scene. Our results are shown in Figure 12. Due to limited GPU memory, we can only allocate 112M cells in the DCGrid instance compared to the 135M used by SPGrid. Even with fewer cells, our automatic topology adaptation scheme produces a more detailed simulation. Because DCGrid uses cell-centric advection, reuses temporary channels, and does not require ghost cells, DCGrid requires only about one-third of the memory compared to SPGrid for a similar simulation. Even when rendering time is included, the experiment with DCGrid runs about 449 times faster than the original experiment with SPGrid (Table 1). Since DCGrid and SPGrid run on different systems, and our DCGrid experiment only uses about 82% of the cells that the SPGrid experiment used, the performance difference might be less significant in other scenarios.

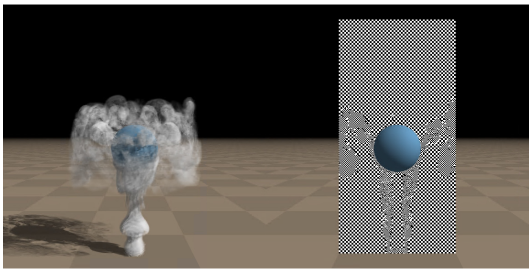


Fig. 12. Smoke flow along sphere, 112M cells allocated. Domain size is $1024^2 \times 2048$, each checkerboard square represents a 4^3 cells.

Comparison to GVDB. To compare DCGrid to GVDB end-to-end, we set up a smoke simulation in GVDB, shown in Figure 13. Performance data is included to Table 1. In the experiment, we used cell-centered semi-Lagrangian advection, vorticity confinement, and non-multigrid projection using 10 Jacobi iterations for both DCGrid and GVDB. We set up GVDB such that the domain automatically expands around the smoke, using bricks of 8^3 cells. To get a fair comparison between GVDB and DCGrid, we configured the memory limit for the DCGrid instance to be equal to the memory used by the GVDB instance. Using the same amount of memory, DCGrid allocated 1.57M cells (of which 1.38M active cells), while GVDB allocated only 1.18M cells. Also, DCGrid performed the fluid simulation about 4.8 times as fast as GVDB overall. We attribute a large part of the performance difference between GVDB and DCGrid to the projection operation, where GVDB has to synchronize its apron cells after each iteration. On advection, DCGrid performed faster than

GVDB. We expect that DCGrid performed faster here because it samples from plain arrays, whereas GVDB samples from textures and surfaces, which may cause overhead.

A major difference between DCGrid and GVDB is that GVDB only allocates cells on the highest resolution. In the simulation shown, this benefits the amount of detail visible in the result. GVDB only has to use memory for the high-resolution cells and can thus allocate more of them than DCGrid, using the same amount of memory. As a result, it preserves some more details in the simulation. In general, only allowing cells on the highest resolution can be a limitation for GVDB. Due to this restriction, for example, we have to resort to a non-multigrid solver for velocity projection.

6 CONCLUSION AND FUTURE WORK

We present a memory-efficient adaptive multigrid structure tailored for fluid simulations on the GPU that features an optimization-based algorithm for automatic local grid refinement. Our experiments demonstrate that compared to SPGrid and GVDB, DCGrid performs important stencil and streaming operations faster. Moreover, performance comparisons on smoke simulation indicate the benefits of DCGrid over SPGrid and GDVB. In supplementary material, we develop a scheme for handling solid-fluid interactions approximately for cells of different resolutions and integrate DCGrid into a framework for terrain atmosphere interaction.

For the design of DCGrid, we chose to manage memory explicitly. For a simpler implementation, one could explore CUDA's virtual memory management APIs. Using these APIs, it might be possible to implement random access using the native memory page table instead of a GPU hash table. The drawback of this approach could be that memory needs to be fetched more often during a simulation. Additionally, one could explore the approach used in [Wu et al. 2018] and store cell data in textures. Using textures could potentially improve the performance of advection and rendering, as it would enable hardware-accelerated interpolation.

Our structure is currently limited in two ways. First, we use fixed-size blocks of 4^3 cells. Allowing users to vary the block size could be beneficial to simulations. Larger blocks, for example, would require fewer apron cell indices per cell, which would reduce memory overhead. Second, we limit the maximum number of blocks per mipmap level. This limit does not appear to restrict the scenarios we explored. However, one could explore using an overall block limit instead. Such a limit would allow for more flexible grid layouts, which could help in highly dynamic scenarios.

ACKNOWLEDGEMENTS

We thank the authors of SPGrid and GVDB for partially making their code publicly available, and the anonymous reviewers for their constructive feedback and suggestions that improved the manuscript. The helpful discussions with Miłosz Makowski are gratefully acknowledged. This work has been supported by KAUST (individual baseline funding).

REFERENCES

Mridul Aanjaneya, Ming Gao, Haixiang Liu, Christopher Batty, and Eftychios Sifakis. 2017. Power diagrams and sparse paged grids for high resolution adaptive liquids. *ACM Transactions on Graphics* 36, 4 (2017), 1–12.

Mridul Aanjaneya, Chengguizi Han, Ryan Goldade, and Christopher Batty. 2019. An Efficient Geometric Multigrid Solver for Viscous Liquids. *ACM Computer Graphics and Interactive Techniques* 2, 2 (2019), 1–21.

Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, Robert Endre Tarjan, et al. 1973. Time bounds for selection. *J. Comput. Syst. Sci.* 7, 4 (1973), 448–461.

Landon Boyd and Robert Bridson. 2012. MultiFLIP for energetic two-phase fluid simulation. *ACM Trans. Graph.* 31, 2 (2012), 16:1–16:12.

Robert Bridson. 2008. *Fluid Simulation for Computer Graphics.* Taylor & Francis.

Nuttapong Chentanez and Matthias Müller. 2011. Real-time Eulerian Water Simulation Using a Restricted Tall Cell Grid. *ACM Trans. Graph.* 30, 4 (2011), 82:1–82:10.

J. J. Cooke, L. M. Armstrong, K. H. Luo, and S. Gu. 2014. Adaptive mesh refinement of gas–liquid flow on an inclined plane. *Computers & Chemical Engineering* 60 (2014), 297–306.

Jens Cornelis, Markus Ihmsen, Andreas Peer, and Matthias Teschner. 2014. IISPH-FLIP for incompressible fluids. *Computer Graphics Forum* 33, 2 (2014), 255–262.

Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. 2001. Visual simulation of smoke. *ACM SIGGRAPH* (2001).

Florian Ferstl, Ryoichi Ando, Chris Wojtan, Rüdiger Westermann, and Nils Thuerey. 2016. Narrow Band FLIP for Liquid Simulations. *Computer Graphics Forum* 35, 2 (2016), 225–232.

Nick Foster and Ronald Fedkiw. 2001. Practical animation of liquids. In *ACM SIGGRAPH*. 23–30.

Nick Foster and Dimitris N. Metaxas. 1996. Realistic Animation of Liquids. In *Proceedings of the Graphics Interface*. 204–212.

Ming Gao, Xinlei Wang, Kui Wu, Andre Pradhana, Eftychios Sifakis, Cem Yuksel, and Chenfanfu Jiang. 2019. GPU optimization of material point methods. *ACM Transactions on Graphics* 37, 6 (2019), 1–12.

Christoph Gissler, Andreas Peer, Stefan Band, Jan Bender, and Matthias Teschner. 2019. Interlinked SPH Pressure Solvers for Strong Fluid-Rigid Coupling. *ACM Transactions on Graphics* 38, 1 (2019), 1–13.

Rama Karl Hoetzlein. 2016. GVDB: Raytracing Sparse Voxel Database Structures on the GPU. In *High Performance Graphics*.

Yuanming Hu, Yu Fang, Ziheng Ge, Ziyin Qu, Yixin Zhu, Andre Pradhana, and Chenfanfu Jiang. 2018. A moving least squares material point method with displacement discontinuity and two-way rigid body coupling. *ACM Transactions on Graphics* 37, 4 (2018), 1–14.

Caelan Lapointe, Nicholas T. Wimer, Jeffrey F. Glusman, Amanda S. Makowiecki, John W. Daily, Gregory B. Rieker, and Peter E. Hamlington. 2020. Efficient simulation of turbulent diffusion flames in OpenFOAM using adaptive mesh refinement. *Fire Safety Journal* 111 (2020), 102934.

Haixiang Liu, Nathan Mitchell, Mridul Aanjaneya, and Eftychios Sifakis. 2016. A Scalable Schur-Complement Fluids Solver for Heterogeneous Compute Platforms. *ACM Trans. Graph.* 35, 6, Article 201 (Nov. 2016), 12 pages.

Frank Losasso, Frédéric Gibou, and Ron Fedkiw. 2004. Simulating water and smoke with an octree data structure. *ACM Transactions on Graphics* 23, 3 (2004), 457–462.

Aleka McAdams, Eftychios Sifakis, and Joseph Teran. 2010. A Parallel Multigrid Poisson Solver for Fluids Simulation on Large Grids. In *Symposium on Computer Animation*. 65–73.

Jeroen Molemaker, Jonathan M. Cohen, Sanjit Patel, and Jonyong Noh. 2008. Low Viscosity Flow Simulations for Animation. In *Symposium on Computer Animation*. 9–18.

G. M. Morton. 1966. A computer oriented geodetic data base and a new technique in file sequencing. *International Business Machines Co.* (1966).

Ken Museth. 2013. VDB: High-Resolution Sparse Volumes with Dynamic Topology. *ACM Transactions on Graphics* 32, 3 (2013), 1–22.

Ken Museth. 2021. NanoVDB: A GPU-friendly and portable VDB data structure for real-time rendering and simulation. In *ACM SIGGRAPH 2021 Talks*. 1–2.

Michael B. Nielsen, Konstantinos Stamatelos, Morten Bojsen-Hansen, and Robert Bridson. 2020. Auto-Adaptivity: An Optimization-Based Approach to Spatial Adaptivity for Smoke Simulations. *SIGGRAPH Talks* (2020).

Rajsekhar Setaluri, Mridul Aanjaneya, Sean Bauer, and Eftychios Sifakis. 2014. SPGrid: A Sparse Paged Grid structure applied to adaptive smoke simulation. *ACM Transactions on Graphics* 33, 6 (2014), 1–12.

Jos Stam. 1999. Stable Fluids. In *ACM SIGGRAPH*. 121–128.

Kui Wu, Nghia Truong, Cem Yuksel, and Rama Hoetzlein. 2018. Fast Fluid Simulations with Sparse Volumes on the GPU. *Computer Graphics Forum* 37, 2 (2018), 157–167.

Yuwei Xiao, Szeyu Chan, Siqi Wang, Bo Zhu, and Xubo Yang. 2020. An adaptive staggered-tilted grid for incompressible flow simulation. *ACM Transactions on Graphics* 39, 6 (2020), 1–15.

Yongning Zhu and Robert Bridson. 2005. Animating sand as a fluid. *ACM Transactions on Graphics* 24, 3 (2005), 965–972.