Vis Comput (2017) 33:1429–1442
DOI 10.1007/s00371-016-1289-x

ORIGINAL ARTICLE

GPU-accelerated SPH fluids surface reconstruction using two-level spatial uniform grids

Wei Wu[1] · Hongping Li[1] · Tianyun Su[2] · Haixing Liu[2] · Zhihan Lv[3]

Published online: 11 July 2016

Abstract An efficient two-level spatial uniform grid structure-based high-quality surface reconstruction method with Marching Cubes (MC) for smoothed particle hydrodynamics (SPH) fluids was presented in this paper. Compared with the traditional way that dividing the simulation domain with uniform grid directly, an enhanced narrow-band approach using the parallel cuckoo hashing method was taken to index the coarse-level surface vertices, hence decrease the memory consumption. Moreover, a two-level spatial uniform grid structure was employed with a scheme of arranging the fine surface vertices, which could preserve the spatial locality property to facilitate the coalesced memory access on the GPU. Our algorithm was designed for parallel architectures, based on which a parallel version of the optimized surface reconstruction was performed on the CUDA platform. In the experiment of comparison to traditional approaches, the results indicated that our surface reconstruction method was more efficient at the same level of quality of the reconstructed surfaces.

Keywords Smoothed particle hydrodynamics · Fluids simulation · Surface reconstruction · Cuckoo hashing

✉ Hongping Li
lhp@ouc.edu.cn

Wei Wu
oucws2011@163.com

Tianyun Su
sutiany@fio.org.cn

1 College of Information Science and Engineering, Ocean University of China, Qingdao 266100, China

2 The First Institute of Oceanography, SOA, Qingdao 266061, China

3 Shenzhen Research Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen 518055, China

1 Introduction

In recent years, smoothed particle hydrodynamics (SPH) is becoming increasingly popular in computer graphics. It has been successfully used for the simulation of various fluid phenomena, such as multiphase fluids, rigid and elastic solids, fluid features, such as spray, foam and tiny air bubbles, granular materials, and other complex scenes that use multi millions of sampling points. The SPH method works by dividing the fluid into a set of discrete particles. The physical quantity of any particle can be obtained by summing the relevant properties of all the neighbor particles within the influence radius. Even though the scenarios simulated using SPH method can be computed at a reasonable expense nowadays, reconstructing high-quality fluid surfaces with high computational efficiency and optimized memory consumption is still a challenge especially for interactive dynamic scenes. A widely used method for the surface reconstruction is representing the surfaces by defining a scalar density field [1–8]; therefore, the isosurface of this 3D field is identified and polygonized with Marching Cubes (MC) [9] to generate the surface triangle meshes. Apart from an appropriate definition of the implicit function, the grid resolution also has a significant influence on the computation time, memory consumption, and the quality of the surface.

The extracted surfaces are commonly thin layers compared to the whole simulation domain. Many researchers have been trying to reduce the space and scalar field computation complexity from $O(n^3)$ to near $O(n^2)$. In summary, there are two different ways to divide grid, adaptive grid division, and uniform grid division. For the adaptive grid division method, an octree structure is usually used to reduce storage space which can achieve a space complexity of $O(n^2)$ in the theory [10–13]. However, it mainly focuses on static scenes due to the expensive construction cost. In addition, the implementation is really tough especially for parallelization. In contrast, the uniform grid division method is characterized by its simplicity, and it can be applied with an efficient parallelization technique that runs on the narrowband around the surface [5]. Narrowband means the scalar field computation will only focus on the surface region instead of the simulation volume. The surface region makes the computational complexity and memory consumption scale with the fluid surface. However, the spatial query for surface grid vertices still requires the full grid representation, which causes the storage for grid vertices highly redundant especially in a high-resolution grid.

Our contribution In this paper, we present a novel two-level spatial uniform grid structure that can improve both the time and space efficiency for fluids surface reconstruction without sacrificing the quality of the reconstructed surfaces. In our method, the scalar field is only computed in the narrow-band region around the fluid surface. By employing an enhanced coarse-to-fine scheme, the two-level grid structure can reconstruct surfaces in the narrow-band region faster. Every step of the algorithm is specially designed for parallel architectures on many-core GPUs. The main contributions of this paper are described as follows:

1. The uniform grid is replaced with a hash table which is established with an efficient parallel cuckoo hashing method [14]. Only the coarse surface cells of the two-level grid structure are hashed into the hash table, which makes the memory consumption entirely irrelevant with the simulation domain. Moreover, the fluid particles are also managed using the parallel cuckoo hashing method to further improve the memory efficiency.
2. The spatial locality is further improved by continuously arranging the fine-grid vertices of the two-level grid structure in memory, which decreases the memory transfer times hence reduces the coalesced memory access time on the GPU.

The final experiments indicate that our method using the two-level spatial uniform grid structure can reconstruct high-quality surfaces with more efficient performance compared to the narrow-band method [5] and traditional uniform grid method.

2 Related work

In recent years, many techniques have been presented for representing surfaces from a set of discrete particles based on the Marching Cubes method. Within the context of these techniques, researchers mainly aimed at achieving an optional balance among surface quality, computational efficiency, and memory consumption. To generate a high-quality surface, an important aspect is the definition of the implicit function of the isosurface. Blinn [15] proposed one of the earliest approaches by introducing blobbies. While this method is comparably simple, the bumpy appearance is visible. Later, Müller et al. [1] presented an algorithm by weighting the contribution of each particle by its volume to alleviate the surface bumps. However, bumpiness still exists. Zhu and Bridson [2] proposed a signed distance field computation method, where the scalar field is computed based on the weighted average values of neighbor particle positions, but the artifacts in concave regions are still observed. This can be addressed by considering the movement of the neighbor particles' center of mass [4] or by a position-based decay function [7]. The other approaches that use anisotropic kernels [8] are also employed, which yields high-quality fluid surfaces but at the expense of high computational complexity.

There exist various techniques to address the performance of surface reconstruction. One way to save on memory consumption is using an adaptive scheme instead of one single uniform grid. Its main idea is taking a coarse-to-fine step to reconstruct surfaces. There are many data structures which can be used to generate an adaptive mesh, e.g., using octrees [10–13]. While being very efficient for memory consumption improvements, the construction of octrees is really time consuming. Therefore, rebuilding octrees in every step are too expensive in dynamic scenes and they mainly focus on static applications or offline rendering. Later, an efficient parallel implementation of octree structures for particle-based fluids is presented with the work of Zhou et al. [16]. A three-level grid structure which adapts its cells according to the curvature of the fluid surfaces has been used for surface reconstruction on the CPU [17], and more recently on the GPU, where the performance is vastly accelerated due to the parallelization [18]. One of the problems of using adaptive grid is that cracks may arise between two grid cells of two different levels; therefore, techniques need to detect and fill them to produce smooth surfaces using one single-level uniform grid structure.

Another useful technique to reduce both the computational complexity and the memory consumption is reconstructing the scalar field only on the narrowband around the surface. Müller et al. [1] suggested applying the Marching Cubes only on the identified surface cells. Bridson [19] proposed a sparse block grids structure which divides the large grid into blocks, while only blocks near the level set surface actually exist. Houston et al. introduced a novel scalable-level set representation and the RLE sparse-level sets [20] to encode the regions with respect to their distance to the narrowband. A dynamic tabular grid (DT-grid) structure is introduced by Nielsen and Museth [21], where the narrowband is constructed without the limitation of any computational box. Later, a different approach using out-of-core techniques together with compression strategies is presented to handle very high grid resolutions [22]. Although these methods can reduce the memory footprint efficiently, they are usually rather tough to implement and none of them are designed for parallel architectures. Within the context of parallel surface reconstruction, Akinci et al. [5] proposed an efficient parallelization method for high-quality surface generations, where the scalar field computation only focuses on the narrowband. However, it still requires the whole grid to mark the narrow-band region, which causes unnecessary allocation for the unused cells.

An effective method to represent spatial grid with low memory consumption is replacing the uniform grid with a spatial hashing table. For SPH fluids simulation, a memory-efficient data structure for the spatial hashing method called compact hashing is proposed [23]. It allows for larger tables and faster queries. In contrast to the uniform grid method, spatial hashing can represent infinite simulation volume. However, the overhead of building and accessing for the hash table should not be negligible. Alcantara et al. [14] presented a parallel hashing method which is a hybrid approach that uses both a classical sparse perfect hashing and a cuckoo hashing. Their work achieves good tradeoff among construction time, access time, and memory requirements. However, few researchers have ever tried to apply this parallel hashing method into the field of surface reconstruction for particle-based fluids.

Spatial locality is another strategy that influences the performance of fluids simulation or surface reconstruction. A space-filling Z curve is usually used to improve the efficiency of fluids simulation [23,24]. In these works, particles are sorted according to the Z curve order, which helps to decrease the memory transfer times. However, the research on the spatial locality of the MC grid vertices in surface reconstruction is still necessary.

There are also other approaches that address the visualization of fluids surfaces efficiently, e.g., explicit meshes [25,26], direct rendering [24,27–29], screen space meshes [30,31], or volume rendering [32–34]. In this paper, we contribute to present a more efficient parallelization technique to reconstruct high-quality fluid surfaces using the Marching Cubes approach by incorporating the scheme of spatial hashing and spatial locality.

3 Fluid surface reconstruction

As mentioned earlier, the definition of implicit function of isosurface has great influence on the surface quality and performance of the reconstruction. To generate a high-quality surface efficiently, we employ the improved signed distance field approach [4] to compute the scalar value for each MC grid vertex. The implicit surface function is defined as

$$\Phi(\mathbf{x}) = |\mathbf{x} - \bar{\mathbf{x}}(\mathbf{x})| - rf, \tag{1}$$

where r is the particle radius, $\bar{\mathbf{x}} = \frac{\sum_j \mathbf{x}_j W(|\mathbf{x}-\mathbf{x}_j|, R)}{\sum_j W(|\mathbf{x}-\mathbf{x}_j|, R)}$ with $R = 4r$, and W is the density kernel function. The factor f is computed as

$$f = \begin{cases} 1 & \mathrm{EV}_{\max} < t_{\mathrm{low}} \\ \gamma^3 - 3\gamma^2 + 3\gamma & \text{otherwise} \end{cases}, \tag{2}$$

with $\gamma = \frac{t_{\mathrm{high}} - \mathrm{EV}_{\max}}{t_{\mathrm{high}} - t_{\mathrm{low}}}$, where $\mathrm{EV}_{\max}$ is the largest eigenvalue of $\nabla_x(\bar{\mathbf{x}}(\mathbf{x}))$. Another important aspect that influences the surface quality is the selection of grid resolution, and three different MC grid cell sizes are generally used for fluids surface reconstruction: $2r$, r and $r/2$, with r being the equilibrium distance of the SPH particles. For quick overview purpose, MC cell size of $2r$ is usually used for a summary preview of fluid animation. For a detailed view of fluid animation, $r/2$ is used to preserve enough and high-quality surface details with expensive computational complexity. r may be selected in the tradeoff between performance and quality case. In this paper, we focus on high-quality surface generation; therefore, our method is specially designed for a finer MC grid resolution.

Our surface reconstruction algorithm is initialized by extracting the surface region based on our two-level spatial uniform grid structure (Sect. 3.1). Then, for the scalar field computation, we focus only on the narrowband around the fluid surfaces (Sect. 3.2). Unnecessary computation and memory allocation for grid vertices far from the fluid surfaces are avoided. In addition, the performance of scalar field computation can be improved by observing optimization strategy of coalesced global memory access on the GPU. Finally, the fluid surfaces are triangulated efficiently with the help of shared memory (Sect. 3.3). The overview of the optimized surface reconstruction algorithm is shown in Fig. 1.

In our algorithm, the surface region is represented by surface vertices. This can be determined using surface particles. Any surface particle can easily define an AABB (axis-aligned bounding box) around the particle which spans $4r$ length on each axis, and each MC grid vertex that resides in this bounding box is marked as surface vertex. We determine the surface particles in a preprocessing step by employing the smoothed color field method [1]. In the following subsections, details of our surface reconstruction algorithm with a grid cell size of $r/2$ are described.

3.1 Extracting surface vertices

Eliminating the memory dependency on the simulation domain is necessary in a large scene with high MC grid resolution. Spatial hashing is a useful approach for this purpose. However, it is indirect to hash surface vertices of MC grid

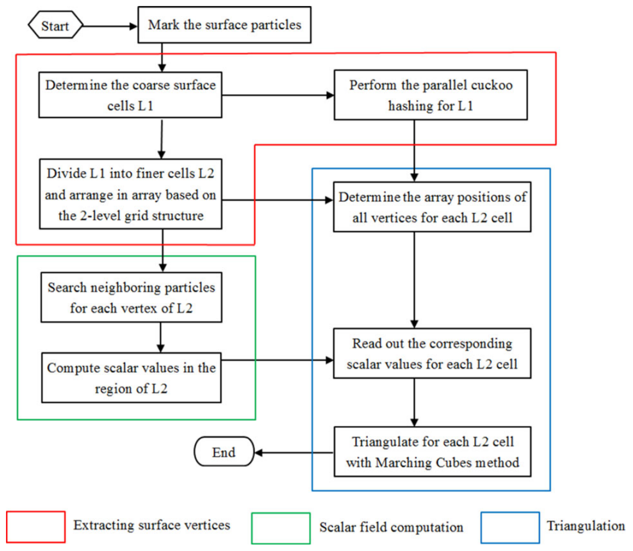

Start
Mark the surface particles
Determine the coarse surface cells L1
Perform the parallel cuckoo hashing for L1
Divide L1 into finer cells L2 and arrange in array based on the 2-level grid structure
Determine the array positions of all vertices for each L2 cell
Search neighboring particles for each vertex of L2
Compute scalar values in the region of L2
Read out the corresponding scalar values for each L2 cell
Triangulate for each L2 cell with Marching Cubes method
End
Extracting surface vertices
Scalar field computation
Triangulation

Fig. 1 Overview of optimized surface reconstruction

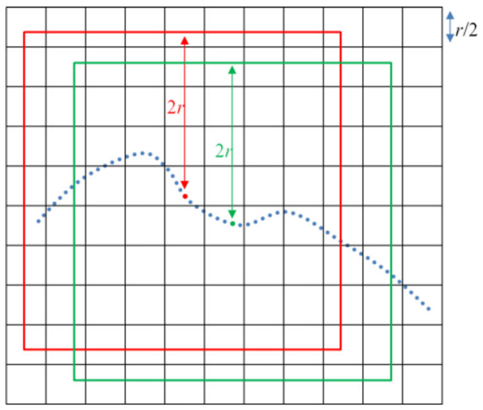


Fig. 2 Memory allocation problem directly using the spatial hashing method in one single uniform grid. Surface particles are shown by *small circle dots*. For parallelization, each thread corresponds to a surface particle and requires pre-allocating a piece of fixed-size memory to store the influenced grid vertices within this surface particle's AABB, which spans a $2r$ length in each direction. The situation of redundant memory allocation is illustrated by *red*- and *green*-colored *dots* and bounding *boxes*

using only one single uniform grid in parallel. First, if we use a hash table instead of the whole MC grid to mark whether it is surface vertex or not, we need to pre-allocate a piece of fixed-size memory to store the influenced grid vertices for each surface particle. For an AABB with 4r length of a side, the fixed-size memory requires preparation for at least 8^3 grid vertices with respect to the MC grid cell size of $r/2$. Allocating memory for each surface particle in such a way results in large amounts of redundancy especially in a large scene. Figure 2 explains this problem with a two-dimensional illustration. Furthermore, even if we determine the surface vertices successfully, hashing all these data in a hash table makes the access for them less efficient, since hash function is not designed to maintain spatial locality, but rather abolish it [23].

To solve these problems, a two-level spatial uniform grid structure integrated with the parallel cuckoo hashing method is designed to extract the surface vertices. In contrast to dividing the uniform grid with a cell size of $r/2$ directly, we first initialize the MC grid cell size with $2r$ as the first level. Each grid cell in this level is called coarse cell. For each surface particle, the neighboring contiguous 3^3 coarse cells are defined as coarse surface cells. They represent the surface region, in which the scalar field is computed. This approach guarantees the fluid surfaces reconstructed with hole-free quality. Then, each coarse surface cell is further divided into finer cells as the second level. Finally, a hash table is built only for the coarse surface cells using the parallel cuckoo hashing method [14].

In the following, we suppose L and M being the length in x- and y-directions separately in the whole MC grid.

1. *Identification of coarse surface cells* Surface particles are traversed in parallel to compute their related coarse cells. The index of the coarse cell for any surface particle with position $\mathbf{x} = (x, y, z)$ is calculated as

$$\mathbf{C3} = \left(\left\lfloor \frac{x - x_{\min}}{d} \right\rfloor, \left\lfloor \frac{y - y_{\min}}{d} \right\rfloor, \left\lfloor \frac{z - z_{\min}}{d} \right\rfloor\right) \quad (3)$$

$$C = \mathbf{C3}.x + \mathbf{C3}.y.\frac{L}{d} + \mathbf{C3}.z.\frac{L}{d}.\frac{M}{d}, \quad (4)$$

where $d = 2r$. Then, the determined coarse cells are expanded in parallel by storing each one's 3^3 neighboring coarse cells into a new array. In such a way, the memory allocation is significantly smaller in contrast to allocating memory for the influenced 8^3 grid vertices of each surface particle, as shown in Fig. 2. A data deduplication step is required for the expanded coarse cells, since the neighbors of a coarse cell may overlap with another coarse cell's neighbors. Thus, we obtain the final coarse surface cells, and the corresponding index values are stored in an array called *coarseSurfaceCells*.
A problem of using coarse cells is that the extracted surface region is slightly larger than directly using one single uniform grid [5] (see Sect. 5 for the comparison). This means more scalar values need to be computed for the surface vertices. However, the efficiency of scalar field computation can be improved by arranging the surface vertices in memory according to step (2) based on the spatial locality principle.
2. *Coarse surface cells subdivision* Each coarse surface cell is subdivided into 64 fine cells with the cell size of r/2, which correspond to 64 fine-grid vertices as the second level. A two-dimensional subdivision illustration is shown in Fig. 3. Each fine cell is represented in memory by the index of the vertex located in the lower left corner of this cell. The position of this vertex lies within the coarse cell C can be represented as

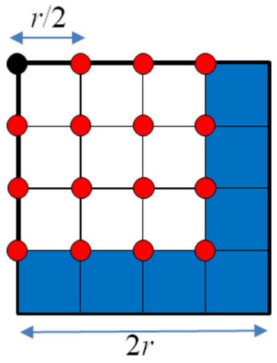


Fig. 3 A two-dimensional subdivision illustration. The *thick black line* represents the outline of the coarse cell, whose initial grid vertex is colored in *black*. Both *black*- and *red*-colored grid vertices are the fine-grid vertices in the second level. The region of *blue* shading denotes the fine cells, whose corner points belong to different coarse cells

$$\mathbf{FineVertexPos} = 4.\mathbf{CPos} + \mathbf{IPos}, \tag{5}$$

with **CPos** being the position of the initial grid vertex of C in the coarse grid and **IPos** being the inside position in C. Each component of **IPos** is defined within the range of [0, 4). Then, the index of this fine-grid vertex with position **FineVertexPos** is defined as

$$P1 = \mathbf{CPos}.x + \mathbf{CPos}.y.\frac{L}{d} + \mathbf{CPos}.z.\frac{L}{d}.\frac{M}{d} \tag{6}$$

$$P2 = \mathbf{IPos}.x + 4.\mathbf{IPos}.y + 16.\mathbf{IPos}.z \tag{7}$$

$$P = 64.P1 + P2, \tag{8}$$

with $d = 2r$. To keep all these fine-grid vertices, we need a new array called *FineSurfaceVertices*, whose size is 64 times larger than array *CoarseSurfaceCells*. Each element of this array stores the index of a fine-grid vertex and its corresponding scalar value.

In specific implementation, a parallelization process starts with each thread corresponding to a coarse surface cell and writes the index values of all 64 fine-grid vertices within this coarse surface cell into a continuous memory. In other words, let $V^i = \{V^i_j | j = 0, 1, \ldots 63\}$, $P^i = \{P^i_j | j = 0, 1, \ldots 63\}$ and suppose the value in array *CoarseSurfaceCells* with array position i is C_i, then the ith thread writes the value V^i to position P^i in array *FineSurfaceVertices* according to the rules as

$$V^i_j = 64.C_i + j \tag{9}$$

$$P^i_j = 64.i + j. \tag{10}$$

Now, we obtain an array that the adjacent fine-grid vertices within a coarse cell are continuous in memory but those belong to different coarse cells are not necessarily close. To triangulate a fine cell, all its eight vertices' array positions need to be determined. For fine cells, whose vertices belong to different coarse cells (see the blue shading in Fig. 3), their vertices' array positions in *FineSurfaceVertices* need to be identified with the aid of a hash table which has been built at stage (3).

3. *Building hash table* Again, we emphasize that we did not hash all extracted fine-grid vertices into a hash table because of its violation to the spatial locality. Instead, we only hash the index values of coarse surface cells in the first level. Key-value pairs are built with each element in array *CoarseSurfaceCells* as the key and the corresponding array position as the value. Then, the parallel cuckoo hashing method is used to determine, where a key-value pair is pushed into the hash table.

Thus, the vertices of a fine cell that belongs to different coarse cells can be obtained with the help of the hash table. Every process in the stage of extracting surface vertices is entirely paralleled.

3.2 Scalar field computation

After extracting all the fine surface vertices, the scalar field computation is performed. This stage consists of two parts, computing coordinates of surface vertices based on

Algorithm 1: Optimized Particles Neighborhood Search

```
FOREACH particle DO
    compute Z-ordered cell ID;
    sort to the new position based on the corresponding cell ID;
    atomic add on the counter of the corresponding cell;
FOREACH particle DO
    IF the first inserted particle THEN
        record the corresponding cell ID as key;
        create a handler that holds the array index of the particle and the counter of the corresponding cell as value;
    ENDIF
perform parallel cuckoo hashing for key-value pairs;
FOREACH fine surface vertex DO
    compute coordinates;
    compute AABB that spans 4r distance on each axis;
    find cells in AABB;
    FOREACH cell DO
        perform the hash table lookup with the cell ID as key;
        find particles inside
```

their indexes, and contributing the neighboring particles for the scalar field computation.

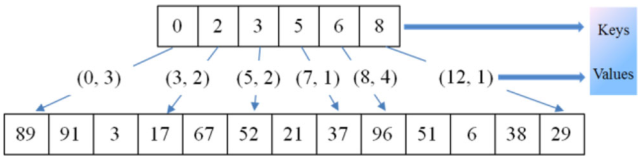


Fig. 4 Illustration of the combination of Z indexing method and parallel cuckoo hashing method. The *top row* illustrates the used cells, where the numbers refer to the corresponding Z ordered cell indexes. Each used cell points to the first inserted particle in the sorted particle array illustrated in the *bottom row*, where the numbers refer to the particle ids. The pair of numbers in *brackets* lie in each *arrow line* indicates the handler of the corresponding cell. Cell indexes as keys together with corresponding handlers as values constitute key-value pairs, which are prepared for hashing

1. *Computing coordinates of surface vertices* We implement this process by traversing along the array *FineSurfaceVertices* in parallel. Since each array element is computed according to formula (8), we divide it by 64 to obtain the quotient $P1$ and the remainder $P2$. Using $P1$ and $P2$, we can reversely compute the vector **CPos** based on formula (6) and vector **IPos** based on formula (7) separately. The position **FineVertexPos** of each surface vertex can be obtained according to formula (5). Then, the corresponding coordinates can be easily computed with the cell size of $r/2$ and the predefined minimum coordinates of the grid domain.
2. *Contributing neighboring particles* After computing the coordinates of each surface vertex, we search their neighboring particles in the influence radius of $4r$. In traditional ways, particles are usually managed using a uniform grid which scales with the simulation domain. Each fluid particle is related to a single cell. For a surface vertex, all particles in cells which overlap the influence range will be collected to compute its scalar value. To eliminate the memory dependency on the simulation domain, an optimization algorithm of identifying particles in a cell is described in the following.

We employ the combination of Z indexing method as discussed in [23] and the parallel cuckoo hashing method to perform the particles neighborhood search. After sorting particles based on their Z ordered cell indexes, we compute a handler only for each used cell. Each handler holds the array index of the first inserted particle in the sorted particle array and the total number of inserted particles in the corresponding cell. Then, the index values of the used cells and their corresponding handlers constitute the key-value pairs which are hashed into a table using the parallel cuckoo hashing method. This method makes the memory allocated for searching neighboring particles scale with the number of particles instead of the whole simulation domain. The data structure is illustrated in Fig. 4. Thus, giving a cell index as the key, we can take out the corresponding handler of this cell in the hash table. The handler offers the information about where to start and stop reading particles from the sorted particles array. The pseudocode of the particles neighborhood search is summarized in Algorithm 1.

Reading particles means the data access from global memory on the graphics card, which is the most exhausting process in the stage of scalar field computation. This process will achieve an efficiency improvement due to the arrangement mode of surface vertices that already stored in array *FineSurfaceVertices*. Before the explanation for this result, we give a brief introduction of the optimization mechanism of coalesced global memory access on the GPU. On NVIDIA GPU architecture, hundreds of threads are divided into groups of 32 parallel threads called warps [35]. Global memory is accessed via a fixed-size (e.g., 32-, 64-, or 128-byte) address-aligned segments. Each time a data access for the memory can only acquire a fixed-size segment of continuous data. If the requested data for all threads within a warp falls into the same address-aligned memory segment, a single memory transaction will happen, namely, coalesced memory access. Otherwise, more than one memory transactions will result, namely, the non-coalesced memory access. Generally speaking, the more transactions are necessary, the more unused data are transferred in addition to the data required by the threads, which decreases the memory transfer efficiency. In our algorithm, the MC fine-grid vertices within a coarse cell have been continuously arranged in memory as presented in Sect. 3.1. Therefore, for threads within a warp, the requested neighboring particles data are more likely to fall into a single memory segment, which complies with the coalesced memory access principle and increases the memory efficiency greatly.

Finally, the scalar value of each surface vertex is computed by contributing all its neighboring particles and stored in the corresponding entry of array *FineSurfaceVertices*.

3.3 Triangulation

In this stage, we triangulate each fine cell in parallel with an optimization method. Before querying MC look-up tables and applying MC algorithm for each fine cell, all its eight corresponding scalar values need to be prepared in advance. Suppose the index of a fine surface vertex in array *FineSurfaceVertices* is *FineVertIdx*, we take the same process as Sect. 3.2 to determine $P1$ and **IPos**. For the corresponding fine cell, its eight vertices' inside positions within the coarse cell are represented as

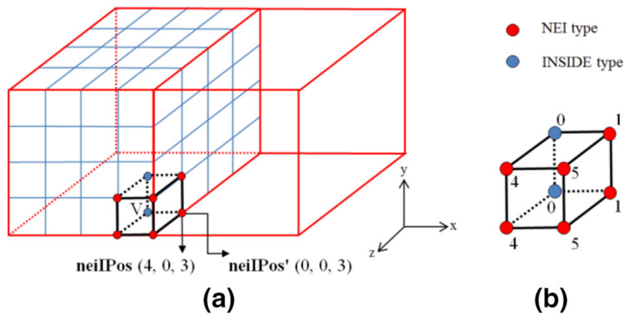


Fig. 5 **a** Scalar values determination for vertices of a fine cell with initial vertex *V*. **b** The vertex type is illustrated with the value *Pflag* computed by the inside position **neiIPos**

$$\mathbf{A} = (i, j, k)$$
$$\mathbf{neiIPos} = \mathbf{IPos} + \mathbf{A}, \quad (11)$$

where i, j, k are the component of **A** and each is assigned as 0 or 1. If any component of **neiIPos** is larger than 3, then we mark this vertex as type *NEI* which indicates that it belongs to a different coarse cell. Otherwise, we mark it as type *INSIDE* which indicates that it lies in the same coarse cell. For the convenience of implementation, the type of this vertex can be represented by an integer flag *Pflag*, which is computed using shift operations as

$$Pflag = flagX + (flagY \ll 1) + (flagZ \ll 2)\,, \quad (12)$$

with *flagX*, *flagY*, and *flagZ* corresponding to each component of **neiIPos** in separate and being assigned as 1 if the component is larger than 3, or else being assigned as 0. The value of *Pflag* is within the range of [0, 8), with value 0 representing the type *INSIDE* and the other values which correspond to the seven possible neighbor coarse cells representing the type *NEI*.
Type INSIDE For a vertex with type *INSIDE*, we determine its array index in *FineSurfaceVertices* by adding an offset from the initial vertex index as

$$ConstantOffset = \mathbf{A}.x + 4.\mathbf{A}.y + 16.\mathbf{A}.z$$
$$neiPIdx = FineVertIdx + ConstantOffset. \quad (13)$$

Type NEI For a vertex with type *NEI*, it belongs to the neighbor coarse cell and may not be continuously stored in memory with the initial vertex, which needs the help of the hash table to determine the final array index. First, we compute the index of the neighbor coarse cell it lies in as

$$ConstantOffset = flagX + flagY.\frac{L}{d} + flagZ.\frac{L}{d}.\frac{M}{d}$$
$$neiCCellIdx = P1 + ConstantOffset, \quad (14)$$

Algorithm 2: Optimized Triangulation

```
FOREACH block (512 threads) in CUDA DO
    allocate shared memory with the size of 64;
    FOREACH coarse surface cell (64 threads) DO
        compute the index values of eight neighboring coarse surface cells (14);
        perform the hash table lookup and store in shared memory;
FOREACH fine surface vertex DO
    create a fine cell;
    FOREACH vertex of the fine cell DO
        compute the value of PFlag (12);
        IF Type INSIDE THEN
            determine the array index (13);
        ELSE IF Type NEI THEN
            retrieve the hash value from shared memory as the base position;
            compute the inside position in the neighboring coarse cell (15);
            determine the array index (16);
        ENDIF
        read out the corresponding scalar value;
    triangulate the fine cell
```

where $d = 2r$. Using *neiCCellIdx* as a key to retrieve the entry in the hash table which has been built in Sect. 3.1 and read out the corresponding value i as the base position. Using **neiIPos**, we can compute the inside position of this vertex in the neighbor coarse cell as

$$\mathbf{neiIPos}' = (\mathbf{neiIPos}.x \,\%\, 4, \mathbf{neiIPos}.y \,\%\, 4, \mathbf{neiIpos}.z \,\%\, 4)\,. \quad (15)$$

Then, the inside index of this vertex in the neighboring coarse cell is computed similar to formula (7), and marked as j. Thus, the array index of this vertex is represented as

$$neiPIdx = 64.i + j. \quad (16)$$

After computing the array index of this vertex, its corresponding scalar value can be read out immediately (See an example in Fig. 5 for a better understanding).

For an optimized implementation, the triangulation stage is started by dividing the extracted fine surface vertices into a set of buckets, each of which contains 512 items. Each bucket corresponds to a block with 512 threads. Apparently, each block manages at most eight coarse cells with 512 fine

Table 1 General information of the presented scenes

Scene	#FP	$\frac{\#FP_{surf}}{\#FP}$	Cell size = r			Cell size = $r/2$			
			$\frac{\#Vert_{surf}}{\#Vert}$	#Triangles (M)	Grid res.	$\frac{\#Vert_{surf}}{\#Vert}$		#Triangles (M)	Grid res.
						NB	Ours		
Drop	0.37 M	0.13	0.31	0.18	164 × 132 × 164	0.11	0.15	0.71	328 × 264 × 328
CBD	1.7 M	0.13	0.27	0.89	284 × 240 × 284	0.1	0.137	3.65	568 × 480 × 568
Landslide	73 K	0.61	0.1	0.27	300 × 176 × 360	0.029	0.043	1.12	600 × 344 × 716
Buddha	0.2 M	0.3	0.053	0.42	259 × 672 × 259	0.023	0.025	1.69	516 × 1342 × 516

All the data are averaged per frame

Table 2 Performance of the presented scenes with a cell size of r

Scene	t_{es} (ms)	t_{sf} (ms)	t_{tri} (ms)	t_{tol} (ms)	MEM (GB)
Drop	11.7	262.7	6.84	288	0.26
CBD	37.5	1120.3	36	1225.5	0.64
Landslide	9.4	189	11.5	221	0.22
Buddha	20.7	537	20.7	580.7	0.21

All the data are averaged per frame

Table 3 Performance of the presented scenes with a cell size of $r/2$

Scene	t_{es} (ms)	t_{sf} (ms)	t_{tri} (ms)	t_{tol} (ms)	MEM (GB)
Drop	18.9	874.3	24.63	943	0.31
CBD	67.9	3819	464	4479.7	1.09
Landslide	18.6	663	75.8	798	0.37
Buddha	50.4	1846	142	2044	0.53

All the data are averaged per frame

vertices. Instead of performing the hash table lookup from global memory for each *NEI* type vertex, we compute the index of each coarse surface cell's seven neighbors as keys and take out the corresponding values from the hash table in advance. All these values are stored in shared memory for fast access. In this case, for the 64 fine vertices in a coarse cell, the hash table lookup from global memory is only performed seven times, which reduces unnecessary data transmission. The whole optimized triangulation stage is summarized in Algorithm 2.

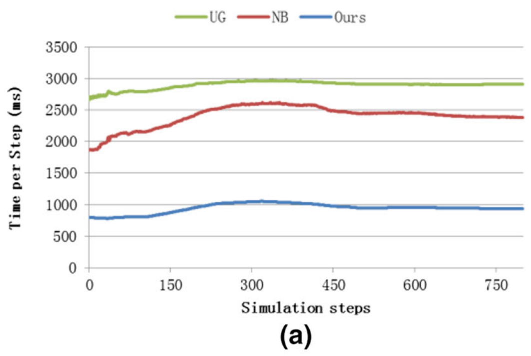
4 Experiment and analysis

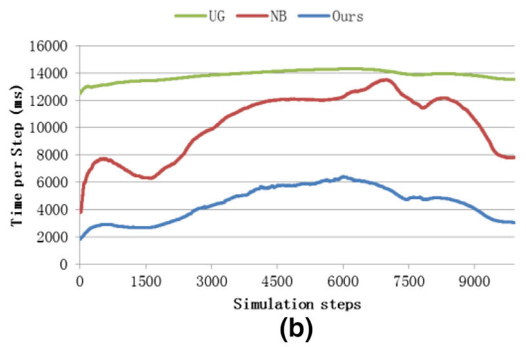
In this section, we use four scenes to demonstrate the utility of our approach with cell sizes of r and $r/2$. All experiments have been carried out via an Intel Core i7-4910MQ CPU @2.9 GHz and NVIDIA Quadro K5100M running on Windows Professional 7 64 bit using VS2010. PCISPH [36] is employed as the basic fluid simulation model. CUDA 5.5 is used for GPU parallel computing and OpenGL is used for rendering.

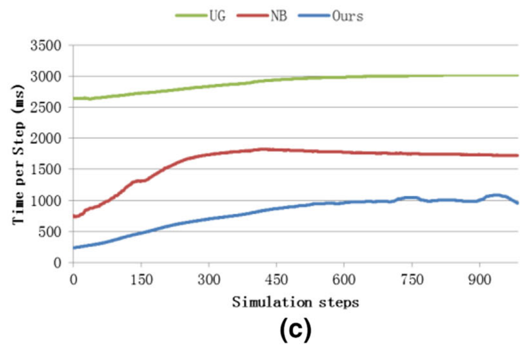
The detailed information of our presented examples is given in Table 1 #FP, $\#FP_{surf}$, #Triangles represent the number of fluid particles, extracted surface particles and the reconstructed triangles, respectively. $\#Vert_{surf}$ and #Vert denote the number of extracted surface vertices and uniform grid vertices, respectively. $\frac{\#Vert_{surf}}{\#Vert}$ is used to define the ratio of the determined surface region.

The computation time and memory consumption with cell sizes of r and $r/2$ are presented in Tables 2 and 3, respectively. t_{es} denotes the time of extracting surface vertices, t_{sf} denotes the time of scalar field computation, t_{tri} denotes the triangulation time, t_{tol} denotes the total running time, and MEM represents the memory usage on the graphics card.

To demonstrate the advantage of our method more clearly, the performance between the traditional uniform grid method [1,4] (UG), the narrow-band method of Akinci et al. [5] (NB), and our method are compared for all scenes using two different grid resolutions: r and $r/2$. The comparison with the UG method is aimed at showing the efficiency difference between the narrow-band method and the non-narrow-band method. We implement the UG method by computing and storing the scalar values for all grid vertices in parallel. The comparison with the NB method is aimed at showing the enhanced effect of the narrow-band method after using our two-level spatial uniform grid structure. The running time per frame over simulation steps with a cell size of $r/2$ between different methods is shown in Fig. 6, and the corresponding average running time and memory usage per frame are shown in Tables 4, 5, 6, and 7. Since the memory difference between our method and the other two methods is only on the memory of graphics card side, we did not present the host side memory size. The average running time per frame with cell sizes of r and $r/2$ between different methods is also compared as in Fig. 11.







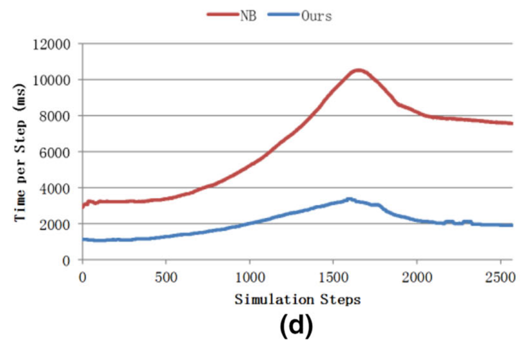


Fig. 6 Surface reconstruction time over simulation steps with a cell size of $r/2$ in presented scenes. **a** In the drop scene, **b** in the corner breaking dam scene, **c** in the submarine landslide scene, and **d** in the Buddha scene

Our first scene simulates a Stanford bunny falls into a mud-filled pool (see Fig. 7). The simulation domain is limited into a box and the spreading splashes rush at the boundary, then bounce back to center to form a spout. In this scene, we have used up to 0.37 M particles. The average surface reconstruction time per frame using our method is 288 and 943 ms with cell sizes of r and $r/2$, respectively. When compared with the UG method and the NB method using a finer cell size of $r/2$ (Table 4), our method is three times and two times faster, respectively. The memory consumption using our method is 0.31 GB and achieves up to four times and two times better in contrast to the other two methods. In Fig. 6a, we can see a suddenly increased surface reconstruction time in the 35th simulation step, as the bunny contacts the mud surface and causes a circle of splashes instantaneously. Then, as more mud spatters towards the air, the curve in this figure changes with an increase tendency. Until about 320th simulation step, the reconstruction time decreases gradually, which indicates the splashes begin to fall back to the pool. From the 600th simulation step, the mud that bounced back by the box boundary flows to center and converges into a slowly rising mud spout, but this phenomenon causes only

Table 4 Comparison between different surface reconstruction methods in the drop scene with a cell size of $r/2$

Method	t_{es} (ms)	t_{sf} (ms)	t_{tri} (ms)	t_{tol} (ms)	MEM (GB)
UG	–	2817	52	2894	1.18
NB	68	2283	26	2404	0.52
Ours	18.9	874.3	24.63	943	0.31

Table 5 Comparison between different surface reconstruction methods in the corner breaking dam scene with a cell size of $r/2$

Method	t_{es} (ms)	t_{sf} (ms)	t_{tri} (ms)	t_{tol} (ms)	MEM (GB)
UG	–	13,424.8	275	13,823	5.75
NB	317.9	9697.9	130	10,274	2.29
Ours	67.9	3819	464	4479.7	1.09

Table 6 Comparison between different surface reconstruction methods in the submarine landslide scene with a cell size of $r/2$

Method	t_{es} (ms)	t_{sf} (ms)	t_{tri} (ms)	t_{tol} (ms)	MEM (GB)
UG	–	2650	209.6	2899	4.9
NB	131.6	1407.6	41.3	1622	1.43
Ours	18.6	663	75.8	798	0.37

Table 7 Comparison between different surface reconstruction methods in the Buddha scene with a cell size of $r/2$

Method	t_{es} (ms)	t_{sf} (ms)	t_{tri} (ms)	t_{tol} (ms)	MEM (GB)
UG	–	–	–	–	–
NB	330.7	5983	89.7	6408	3.16
Ours	50.4	1846	142	2044	0.53

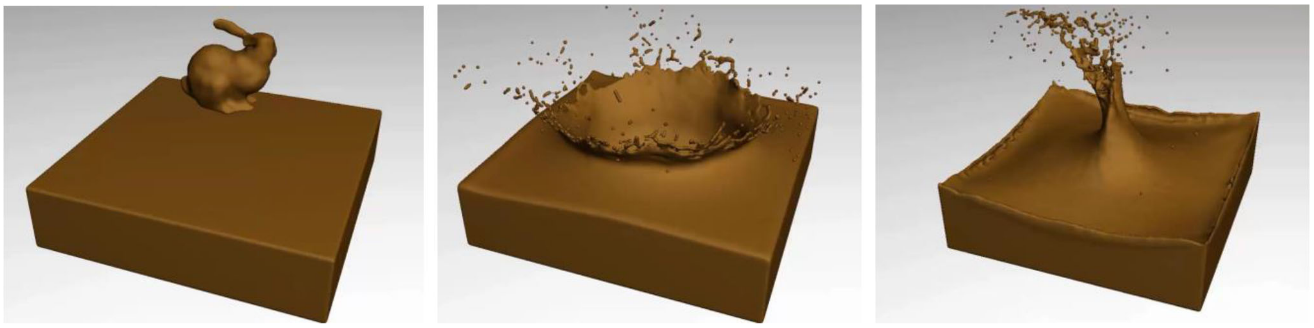


Fig. 7 Drop scene with 0.37 M particles and average 0.71M triangles per frame

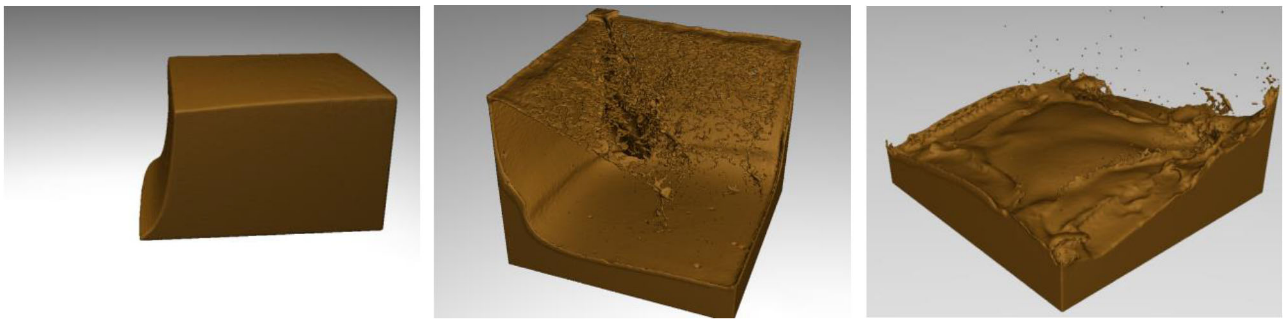


Fig. 8 corner breaking dam scene with 1.7 M particles and average 3.65M triangles per frame

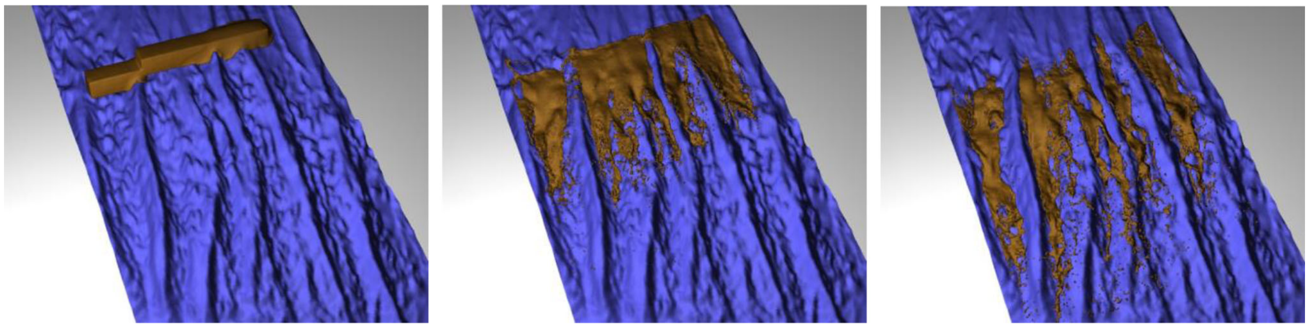


Fig. 9 Submarine landslide scene with 73K particles and average 1.12M triangles per frame

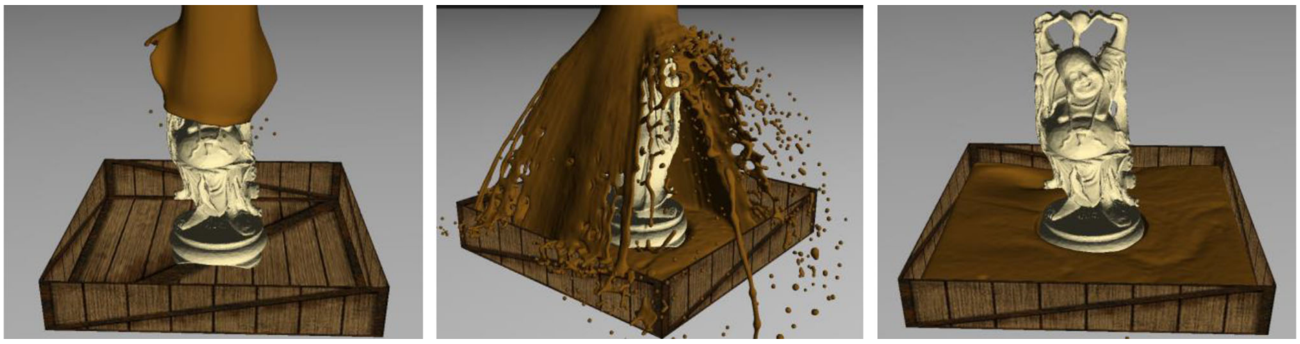


Fig. 10 Buddha scene with 0.2 M particles and average 1.69M triangles per frame

little influence on the surface reconstruction time, presenting in the curve with a steady form. The next example is a corner breaking dam scene (see Fig. 8), in which a huge block of water at the corner of a tank is suddenly released. In this scene, the fluids are simulated with 1.7 M particles. The average surface reconstruction time per frame is 1225.5

and 4479.7 ms with cell sizes of r and r/2, respectively. With a cell size of *r*/2, our method can achieve a speed-up of three times and two times compared with the UG method and the NB method (Table 5). The memory consumption is 1.09 GB, gaining five times and two times better memory consumption, respectively. As shown in Fig. 6b, we can observe a gradually rising tendency until about the 7000th simulation step, during which period, the particles strike the box boundary hence generate more and more surface triangles. Then, as particles fall back from the box boundary, the curve in this figure changes with a downtrend.

In the third scene, we simulate a landslide slipping process along a submarine topography model with about 73K particles (see Fig. 9). Due to gravity, the initial intact slipping body slips down and is gradually separated along the submarine ravine. The average surface reconstruction time per frame using our method is 221 and 798 ms with cell sizes of *r* and *r*/2, respectively. The ratio between surface/total vertices is 0.043 with up to 13 times and 4 times better memory consumption compared with the other two methods using a cell size of *r*/2 (Table 6). The reason of this strength is that the slipping body only occupies a thin layer in comparison to the whole simulation domain. Generally speaking, the smaller the ratio of surface vertices to total grid vertices, the higher the speed-up of memory consumption can achieve. Although some methods compute the minimum bounding box dynamically in every frame and the memory efficiency can be improved to a certain extent, the extra computation time for the bounding box on the GPU cannot be ignored. In addition, as the initial slide block tiles gradually along the oblique landslide, its minimum bounding box becomes larger accordingly, which still causes the wasted memory allocation.

The last example is a Buddha scene (see Fig. 10), where a viscous fluid is poured onto a Buddha statue and collected into a boundary box. The falling fluids strike the top of the statue and gradually form a splash towards the side of the statue, which can reveal the fine details of the reconstructed surfaces. The initial fluids is set to a slender column, and hence, a high bounding box is needed. We use 0.2 M particles to reconstruct fluid surfaces in this scene. The average surface reconstruction time is 580.7 and 2044 ms with cell sizes of r and *r*/2, respectively. Compared with the NB method with cell size of *r*/2, our method can achieve a speed-up of three times in the running time and six times in the memory consumption (Table 7). When using UG method to reconstruct surfaces, it runs out of memory due to the large bounding box.

Throughout the average running time per frame of all the above four scenes in Fig. 11, results indicate that our method is the most optimal using both cell sizes of *r* and *r*/2. In addition, the benefit of our method is more obvious when choosing a finer cell size. This is because the spatial locality can be well preserved using a high grid resolution, while the benefit of arranging surface vertices is limited and may suppressed by the extra overhead for hashing in a coarser grid. With a cell size of *r*/2, our method runs at least two times faster even if the ratio of $\frac{\#\text{Vert}_{\text{surf}}}{\#\text{Vert}}$ is larger (Table 1) when compared with the NB method. The mainly improvement is at the stage of scalar field computation, which can be explained as: using our two-level spatial uniform grid structure, the required data for the continuous threads are more likely to be located in a continuous memory, which conduces to the reduction of memory transfer times. Another speed-up is at the stage of extracting surface vertices. It is noted that even if our method requires the procedure of building hash table and more steps are needed for the surface vertices extraction, the maximum speed-up in this phase can still achieve 7 times in the landslide scene. The reason is that every step of our method is designed for parallel architectures, yet a serial part is unavoidable in the NB method due to the possible race conditions. The triangulation process only accounts for a rather small proportion in comparison with the scalar field computation stage. However, with the help of shared memory, the data access by the way of hashing in our method did not cause too much influence on the performance, presenting in Tables 4, 5, 6, and 7 with an approximately similar triangulation time in contrast to the NB method.

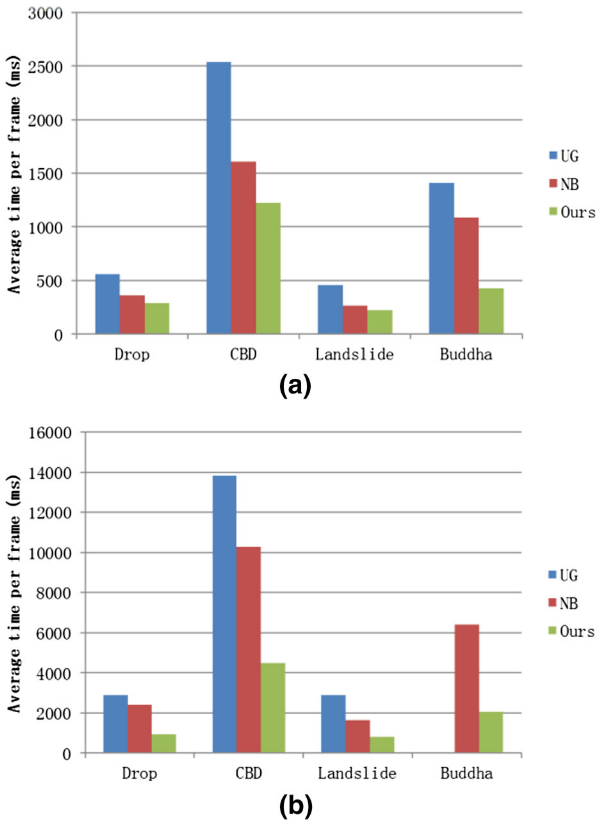


Fig. 11 Average running time per frame between different methods. **a** With a cell size of *r* and **b** with a cell size of *r*/2

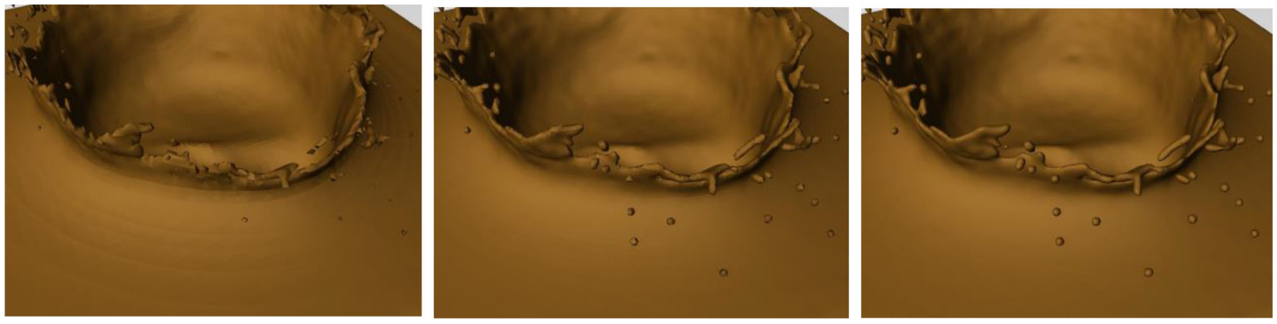


Fig. 12 Enlarged view of the reconstructed surfaces for the drop scene with cell sizes of $2r$, r, and $r/2$, respectively

With respect to the visual quality of our reconstructed surfaces, the scalar field computation method and the grid resolutions are the two most important factors as stated before. In this section, we take the scalar field computation method in [4] for all experiments because of its benefits in both performance and quality, as discussed in [5]. Our method aims at improving the computation time and memory efficiency from the perspective of spatial data structure; therefore, the effects of applying different scalar field computation method are beyond the scope of this paper. However, it is important to note that any scalar field computation method can be integrated into the designed two-level spatial uniform grid structure. Different grid resolutions have an obvious influence on the visual quality. The magnified parts of the surfaces in the drop scene with cell sizes of $2r$, r, and $r/2$ are presented in Fig. 12, since the splashes can demonstrate the visual difference more clearly. This difference can also be verified by the numbers of reconstructed triangles, which is 43K, 0.18M, and 0.71M, respectively, almost four times relationship between them.

We have compared the quality of the reconstructed surfaces with the UG method and the NB method using cell sizes of r and $r/2$ for all presented scenes. The results indicate that the visual quality is almost no difference when taking the same scalar field computation method and the same grid resolution. Especially with a finer cell size of $r/2$, both methods yield smooth high-quality surfaces as in Figs. 7, 8, 9, and 10.

5 Conclusion

In this paper, we present a memory-efficient and performance-friendly surface reconstruction method for particle-based fluids using a two-level spatial uniform grid structure. Although our method extracts a larger surface region when compared to the traditional narrow-band surface reconstruction method and thus more scalar values need to be computed for the extracted surface grid vertices, we make up this defect by improving the spatial locality using a specially designed data structure. Our approach runs especially well with a high-resolution grid in a large volume, in which situation, the savable memory consumption is more obvious. In the future, we expect to explore an adaptive surface reconstruction method and attempt the integration with the method in this paper for further performance improvement.

Acknowledgements This work was jointly supported by the National Natural Science Foundation of China (Grant No. 41275013) and the National High-Tech Research and Development Program (863) (Grant No. 2013AA09A506-4).

References

1. Müller, M., Charypar, D., Gross, M.: Particle-based fluid simulation for interactive applications. In: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation, pp. 154–159. Eurographics Association, UK (2003)
2. Zhu, Y., Bridson, R.: Animating sand as a fluid. ACM Trans. Graphics **24**(3), 965–972 (2005)
3. Adams, B., Pauly, M., Keiser, R., et al.: Adaptively sampled particle fluids. In: ACM Transactions on Graphics (TOG), vol. 26, no. 3, pp. 48. ACM, New York (2007)
4. Solenthaler, B., Schläfli, J., Pajarola, R.: A unified particle model for fluid-solid interactions. Comput. Anim. Virtual Worlds **18**(1), 69–82 (2007)
5. Akinci, G., Ihmsen, M., Akinci, N., et al.: Parallel surface reconstruction for particle-based fluids. In: Computer Graphics Forum, vol. 31, no. 6, pp. 1797–1809. Blackwell Publishing Ltd, Hoboken (2012)
6. Akinci, G., Akinci, N., Ihmsen, M.,Teschner, M.: An efficient surface reconstruction pipeline for particle-based fluids, pp. 61–68. In: Proc. VRIPHYS. Darmstadt, Germany (2012)
7. Onderik, J., Chládek, M., Durikovic, R.: SPH with small scale details and improved surface reconstruction. In: Proceedings of the 27th Spring Conference on Computer Graphics, SCCG '11, pp. 29–36(2013)
8. Yu, J., Turk, G.: Reconstructing surfaces of particle-based fluids using anisotropic kernels. ACM Trans. Graphics **32**(1), 5 (2013)
9. Lorensen, W.E., Cline, H.E.: Marching cubes: A high resolution 3D surface construction algorithm. In: ACM siggraph computer graphics, vol. 21, no. 4, pp. 163–169. ACM, New York (1987)
10. Velasco, F., Torres, J.C.: Cell Octrees: A new data structure for volume modeling and visualization. In: Ertl, T., Girod, B., Niemann, H., Seidel, H.P. (eds.) Proceedings of the Vision Modeling and Visualization Conference 2001 (VMV'01), pp. 151–158. Aka GmbH, Germany (2001)

11. Lee, H., Yang, H.S.: Real-time Marching-cube-based LOD Surface Modeling of 3D Objects. ICAT (2004)
12. Ju, T., Udeshi, T.: Intersection-free contouring on an octree grid. In: Proceedings of 14th Pacific Conference. Computer Graphics and Applications (PG '06). IEEE Computer Society Press, Los Alamitos (2006)
13. Manson, J., Schaefer, S.: Isosurfaces over simplicial partitions of multiresolution grids. Comput. Graphics Forum **29**(2), 377–385 (2010)
14. Alcantara, D.A., Sharf, A., Abbasinejad, F., et al.: Real-time parallel hashing on the GPU. ACM Trans. Graphics **28**(5), 154 (2009)
15. Blinn, J.F.: A generalization of algebraic surface drawing. ACM Trans. Graphics **1**(3), 235–256 (1982)
16. Zhou, K., Gong, M., Huang, X., Guo, B.: Data-parallel octrees for surface reconstruction. IEEE Trans. Visual Comput. Graphics **17**(5), 669–681 (2011)
17. Akinci, G., Akinci, N., Oswald, E., Teschner, M.: Adaptive surface reconstruction for SPH using 3-Level Uniform Grids. In: WSCG proceedings, pp. 195–204. Union Agency (2013)
18. Du, S., Kanai, T.: GPU-based Adaptive Surface Reconstruction for Real-time SPH Fluids. In: WSCG proceedings, pp. 141–150 (2014)
19. Bridson, R.E.: Computational aspects of dynamic surfaces. Stanford University, Stanford (2003)
20. Houston, B., Wiebe, M., Batty, C.C.: RLE sparse level sets. In: Proceedings of the SIGGRAPH 2004 conference on sketches & applications. New York (2004)
21. Nielsen, M.B., Museth, K.: Dynamic tubular grid: an efficient data structure and algorithms for high resolution level sets. J. Sci. Comput. **26**(3), 261–299 (2006)
22. Nielsen, M. B., Nilsson, O., Söderström, A., Museth, K.: Out-of-core and compressed level set methods. ACM Trans. Graphics **26**(4), 16 (2007)
23. Ihmsen, M., Akinci, N., Becker, M., et al.: A Parallel SPH Implementation on Multi -Core CPUs. In: Computer Graphics Forum, vol. 30, no. 1, pp. 99–112. Blackwell Publishing Ltd, Hoboken (2011)
24. Goswami, P., Schlegel, P., Solenthaler, B., et al.: Interactive SPH simulation and rendering on the GPU. In: Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, pp. 55–64. Eurographics Association, UK (2010)
25. Premoze, S., Tasdizen, T., Bigler, J., Lefohn, A., Whitaker, R.: Particle-based simulation of fluids. In: Computer Graphics Forum (Proceedings of Eurographics), vol. 22, pp. 401–410 (2003)
26. Yu, J., Wojtan, C., Turk, G., Yap, C.: Explicit mesh surfaces for particle based fluids. Eurographics **2012**(30), 41–48 (2012)
27. Gribble, C.P., Ize, T., Kensler, A., Wald, I., Parker, S.G.: A coherent grid traversal approach to visualizing particle-based simulation data. IEEE Trans. Visual Comput. Graphics **13**(4), 758–768 (2007)
28. Kanamori, Y., Szego, Z., Nishita, T.: GPU-based fast ray casting for a large number of metaballs. Comput. Graphics Forum **27**(2), 351–360 (2008)
29. Zhang, Y., Solenthaler, B., Pajarola, R.: Adaptive sampling and rendering of fluids on the GPU. In: Proceedings Symposium on Point-Based Graphics, pp. 137–146 (2008)
30. Müller, M., Heidelberger, B., Hennix, M., Ratcliff, J.: Position based dynamics. J. Vis. Commun. **18**(2), 109–118 (2007)
31. Bagar, F., Scherzer, D., Wimmer, M.: A layered particle-based fluid model for real-time rendering of water. In: Computer Graphics Forum (Proceedings EGSR 2010), vol. 29, no. 4, pp. 1383–1389 (2010)
32. Orthmann, J., Keller, M., Kolb, A.: Topologycaching for dynamic particle, volume raycasting. In: Proceedings Vision, Modeling and Visualization (VMV), pp. 147–154 (2010)
33. Jang, Y., Fuchs, R., Schindler, B., Peikert, R.: Volumetric evaluation of meshless data from smoothed particle hydrodynamics simulations. In: Proceedings of the 8th IEEE EG International Conference on volume graphics, VG'10, pp. 45–52. Eurographics Association, UK (2010)
34. Ihmsen, M., Akinci, N., Akinci, G., Teschner, M.: Unified spray, foam and bubbles for particle-based fluids. Vis. Comput. **30**(1), 99–112 (2012)
35. Nvidia: CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#hardware-implementation (2015). Accessed 10 Nov 2015
36. Solenthaler, B., Pajarola, R.: Predictive-corrective incompressible SPH. In: ACM transactions on graphics (TOG), vol. 28, no. 3, p. 40. ACM, New York (2009)

Wei Wu is taking a continuous academic project that involves postgraduate and doctoral study for doctoral degree in the College of Information Science and Engineering at Ocean University of China, Qingdao, China. His current research interests include computer graphics, physically-based simulation, and GPU computing.

Hongping Li received his B.S. and M.S. degree from Tianjin University, Tianjin, China in 1984 and 1988, respectively. In 2003, he received his Ph.D degree in computer science from the University of Oklahoma, Norman, OK, USA. He severed as a lecturer in Tsinghua University, Beijing, China from 1991 to 1997. In 2004, he joined the faculty of Ocean University of China, Qingdao, China, and served as a professor in the Department of Marine Technology. His research interests include ocean remote sensing and parallel computing.

Tianyun Su received his bachelor's degree in electrical engineering from Ocean University of China, China, in 2000 and his M.S. degree and Ph.D. in marine geosciences from the First Institute of Oceanography, SOA, China, in 2003 and Ocean University of China, China, in 2006, respectively. Currently, he is an associate professor at Marine Information and Computation Center, the First Institute of Oceanography, SOA, China. His research interests include spatial database, marine GIS, and multi-view rendering of marine information. He has published nearly 50 papers in international and domestic journals and conferences.

Haixing Liu received bachelor’s degree in mathmatics at Peking University in 1983. He is a Research Professor at Marine Information and Computation Center, the First Institute of Oceanography, SOA, China. His research interests include Operational Oceanography System, Marine Environment Information System, Database, Visualization, Software Engineering. He has published nearly 20 papers in journals and conferences.

Zhihan Lv is an Engineer and a Researcher of virtual/augmented reality and multimedia major in mathematics and computer science, having plenty of work experience in virtual reality and augmented reality projects, engage in the application of computer visualization and computer vision. His research application fields widely range from everyday life to traditional research fields (i.e., geography, biology, and medicine). During the past years, he has completed several projects successfully on PC, Website, smartphone, and smartglasses.