GPU-based Adaptive Surface Reconstruction for Real-time SPH Fluids

Shuchen Du
The University of Tokyo
shuchen@graco.c.u-tokyo.ac.jp

Takashi Kanai
The University of Tokyo
kanait@acm.org

ABSTRACT

We propose a GPU-based adaptive surface reconstruction algorithm for Smoothed-Particle Hydrodynamics (SPH) fluids. The adaptive surface is reconstructed from 3-level grids as proposed by [Akinci13]. The novel part of our algorithm is a pattern based approach for crack filling, which is recognized as the most challengeable part of building adaptive surfaces. Unlike prior CPU-based approaches [Shu95, Shekhar96, Westermann99, Akinci13] that detect and fill cracks according to some criteria during program running that were slow and unrobust, all the possible crack patterns are analyzed and defined in advance and later, during program running, the cracks are detected and filled according to the patterns. Our approach is thus robust, GPU-friendly, and easy to implement. Results obtained show that our algorithm can produce surface meshes of almost the same quality as those produced by the conventional Marching Cubes method, with significantly reduced computation time and memory usage.

Keywords

GPU Computing, Surface Reconstruction, Computer Animation, Fluid Simulation, Particle-based Simulation, Smoothed-Particle Hydrodynamics, Adaptive Marching Cubes

1 INTRODUCTION

In SPH-based fluid simulation, especially for water simulation, carrying out only particle representation is not enough. A surface built upon the particles is desired. For real-time or interactive applications, not only must particle simulation be carried out, but surface reconstruction also need to be computed at high speed. Surface reconstruction for SPH fluids is well studied in Computer Graphics research. With these methods, generally an isosurface of some implicit functions is computed and a triangle mesh is constructed using the Marching Cubes (MC) [Lorensen87] method. Also, for real-time applications, GPU-based versions of MC are utilized such as [Dyken08]. However, with conventional MC methods, because the triangle resolution is uniform, the construction of high-resolution models of large data set requires considerable computation time and a large amount of memory. In order to reduce the cost of uniform MC, Adaptive Marching Cubes (AMC) algorithms such as [Shu95, Shekhar96, Westermann99, Akinci13] have been proposed, in which high resolution triangles are used to capture the thin features of highly deformable surface parts while low resolution triangles are used for flat surface areas. AMC requires less memory and time for generating triangle meshes of similar quality as these uniform meshes. However, approaches proposed to date are all CPU-based, which are very slow and more or less cannot be used in real-time or interactive applications.

We propose a fast GPU-based adaptive surface reconstruction algorithm for Smoothed-Particle Hydrodynamics (SPH) fluids. Our method has several characteristics: First, the algorithm is totally implemented on GPU and it is faster than the CPU implementation. Secondly, we analyze all the possible cracks that may occur and define them as patterns and later, during program running, the cracks are simply detected and filled by referring to the pre-defined crack patterns.

Our work is categorized as a novel extension of [Akinci13] with an acceleration of performance by GPU implementation. Unlike [Akinci13] which is implemented on CPU, our algorithm is designed from scratch for execution on GPU, thus making it totally different from that of [Akinci13]. Specifically, in [Akinci13], the existence of surface cracks is checked during the algorithm running, and if a crack exists, the crack is filled on-the-fly. In contrast, our algorithm is completely procedural; cracks are simply detected and filled by referring to the pre-defined crack patterns. Our method is robust enough to deal with all the surface cracks.

Our proposed algorithm is a totally GPU-based adaptive surface reconstruction one. Specifically, it con-

sists of two parts: First, for each level grid, we propose array-based data structures to store the cell vertices, edge vertices and the numbers of triangles generated in the tesselated cells according to their local indices. This enables parallel accessing among different grid properties for MC implementation without the need to maintain the relationship among vertices, edges and tesselated cells in memory. Secondly, we propose a GPU-friendly algorithm for detecting and filling surface cracks based on pre-defined crack patterns.

2 RELATED WORK

2.1 SPH Fluid Simulation

Smoothed Particle Hydrodynamics (SPH) is a Lagrangian particle framework originally presented by [Lucy77, Gingold77] for astrophysical simulations. Desbrun and Gascuel [Desbrun96] introduced the method to the CG community to simulate highly deformable objects and Müller et al. [Müller03] extended it to the simulation of fluids in real-time for interactive applications. On the other hand, Adams et al. [Adams07] extended the uniform SPH model to an adaptive one to simulate large scale fluids with reduced number of particles. Harada et al. [Harada07] improved the performance on GPUs. A comprehensive survey on recent state-of-the-art SPH developments was conducted by [Ihmsen14].

2.2 Marching Cubes/Adaptive Marching Cubes

Marching Cubes (MC) [Lorensen87] is the dominant method for contouring an implicit field using triangle meshes. Several useful methods of building a surface for SPH fluids based on MC are proposed [Zhu05, Solenthaler07, Yu10, Akinci12].

In the typical MC, a mesh with a large number of triangles is extracted from a uniform high-resolution grid. However, a large number of redundant surface triangles are also generated. In order to reduce such triangles while maintaining their resolution, Adaptive Marching Cubes (AMC) methods [Shu95, Shekhar96, Westermann99] were proposed which adaptively extract triangles using octrees. Recently, Akinci et al. [Akinci13] use grids of three levels to construct an adaptive surface mesh. The above methods reduce redundant triangles and memory used to reconstruct large scale meshes efficiently. However, one problem with such methods is that they generally produce cracks between different levels of grids. Thus, with real-time applications, obtaining high quality surface mesh with no cracks is as important as accelerating their performance.

In order to fill the cracks and obtain watertight surface mesh, Shu et al. [Shu95] proposed the detection of crack regions and filling them using polygons with the same boundary shape as these regions. Their method produces surface meshes with triangles and non-triangle polygons, which causes inefficiency in subsequent processes such as rendering and other geometry processing tasks that require triangle meshes. Shekhar et al. [Shekhar96] first identified cracks surrounded by iso-curves of different level grids and projected the fine iso-curves to coarse ones. There are two problems with their method: First, T-vertices are generated and these are bad for rendering; secondly, not all the cracks are surrounded by iso-curves of different level grids. There can be cracks that are only surrounded by fine iso-curves. Westermann et al. [Westermann99] uses triangle fans to fill the cracks. However, the triangle fans are constructed by traversing the neighbor grid cells that are difficult for parallel implementation. Recently, Akinci et al. [Akinci13] proposed a method for detecting cracks surrounded by at least one intersection at the inner edges and filling them using triangles. The main issue with their approach is that not all cracks contain inner edge intersections. There can be cracks which only contain intersections of outer edges.

[Akinci13] is the most relevant research to ours. In order to reconstruct an adaptive surface from 3-level grids, [Akinci13] uses a map-based data structure whose key is the position of a grid vertex and value is the corresponding implicit function value. However, their approach is difficult to implement on GPU in parallel, because all the grid vertices of different levels are mixed together in the data structure. In our GPU-based parallel implementation, we use three arrays to store the implicit values of grid vertices level by level. The corresponding adaptive surface of each level is reconstructed by invoking a kernel function of CUDA. In order to fill the surface cracks, [Akinci13] checks whether a crack occurs using information on cell neighborhood and intersection of inner edge. Then, they fill the cracks by traversing outer edges and pushing an edge to a crack array if it has an intersection. However, their CPU-based approach is not robust enough as discussed above and difficult to extend to GPU.

As for other adaptive surface reconstruction methods, Ju et al. [Ju02] proposed Dual Contouring (DC) which is a feature-preserving isosurfacing method that extracts crack-free surfaces from both uniform and adaptive octree grids whose edges contain Hermite data. Schaefer et al. [Schaefer07] proposed the Manifold Dual Contouring method for correcting topology errors on surfaces generated by DC to guarantee manifold surfaces. However, mapping DC to GPU is not straightforward for its octree data structure and QEF based vertex positioning which requires considerable memory [Schmitz09]. In addition, Ho et al. [Ho05] also

proposed a method called Cubical Marching Squares (CMS) that can extract crack-free adaptive surfaces from Hermite data in octrees. However, like DC, mapping CMS to GPU is not straightforward for its octree data structure.

With parallel adaptive surface reconstruction, Zhou et al. [Zhou11] introduced a Look Up Tables (LUT) based technique for efficiently computing the neighborhood information of every octree node and utilized such data structure to construct AMCs on GPU. The main problem with their node based approach is that all the computations have to refer to the data of parent, child and neighboring nodes, which makes the data structure complicated and uses a large amount of memory.

3 PARALLEL SURFACE RECONSTRUCTION ON 3-LEVEL GRIDS

In this section, we present our parallel algorithm for the 3-level surface reconstruction on SPH fluids before crack filling. Here, the SPH fluid models can be uniform [Müller03] or adaptive [Adams07]. It is noted that, in order to approximate the surface mesh accurately, the criterion for determining the subdivision level of cells is different. The level difference of two adjacent cells that have a common face should be less than 1 for the convenience of the next crack filling process.

We designed our algorithm based on 3-level grids like [Akinci13] rather than octrees because in an octree, the neighborhood conditions are usually different from cell to cell and have to be recorded using specially designed complex data structures as in [Zhou11]. However, 3-level grids are simple and locally uniform, which is important for designing parallel algorithms. For real-time fluid surface extraction, we believe that surfaces of three levels are enough to reduce the memory and time needed for construction while still capturing thin features in the surfaces. The flexibility here is that the users can add the number of subdivision levels for specific applications by adding the corresponding data arrays and applying an algorithm similar to existing ones.

For uniform SPH fluids, the input is a particle position array whose elements are the coordinates of particles and a cell level array discretizing the simulation domain whose elements indicate which level the cells are on. For adaptive SPH fluids, besides the two arrays above, a particle level array whose elements are particle level numbers is also required. The outputs are reconstructed adaptive surface meshes that are stored in six arrays. For each level grid, we use two arrays to store the corresponding surface mesh; a surface vertex array whose elements are the coordinates of surface mesh vertices and a surface triangle array whose elements are triangles formed by the indices of the corresponding vertex array.

Our algorithm for adaptive surface reconstruction performed before crack filling for uniform SPH fluids is described in the following, in which arrays are largely used as data structures. The algorithm is further divided into two parts, **Algorithm 1** and **2**. **Algorithm 1** comprises of three steps, Step 1 to Step 3, which make up the preprocessing part preparing for later operations. **Algorithm 2** comprises of four steps, Step 4 to Step 7, which describe details about MC surface reconstruction for each level.

Here, for array $\mathscr{A}$, we represent the value of an array element as $\mathscr{A}[i]$ and the CUDA thread that manipulates the element as a_i. Then we explain the algorithm in detail step by step. For simplicity, we illustrate it in 2D while a 3D version can be derived similarly. Also, we only take level-2 cells as an example. Cells of other levels can be computed similarly.

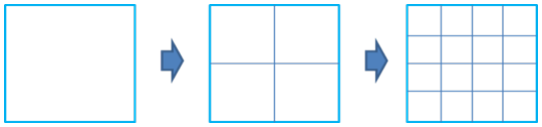


Figure 1: 3-level grid cells. From left to right: level-1 grid cell, level-2 grid cell, level-3 grid cell.

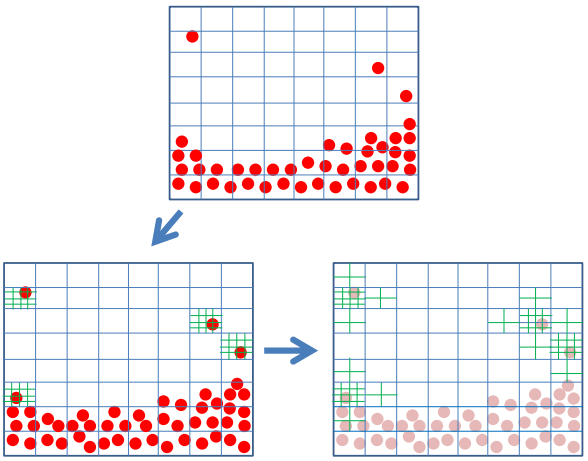


Figure 2: Cell splitting on 8×8 grid. Top: All grid cells are set to level-1 initially. Bottom left: A level-1 grid cell is split to level-3 if it only contains one fluid particle. Bottom right: The adjacent level-1 cells are split into level-2 cells. Finally, there are four level-3 cells, 11 level-2 cells, and 50 level-1 cells in the grid.

The array data structures of level-2 cells in 2D are explained in detail in Figure 3. There are five arrays corresponding to level-2 cells: $\mathscr{V}^2$ (vertex coordinate index array), $\mathscr{I}^2$ (cell vertex implicit value array), $\mathscr{E}^2$ (cell edge vertex array), $\mathscr{H}^2$ (cell edge vertex normal vector array) and $\mathscr{C}^2$ (triangle number array), whose lengths and element types are illustrated in Figure 3 where N_2 is the number of level-2 cells.

We store the different data of each cell in the corresponding arrays according to its level and "n^{th}" cell in

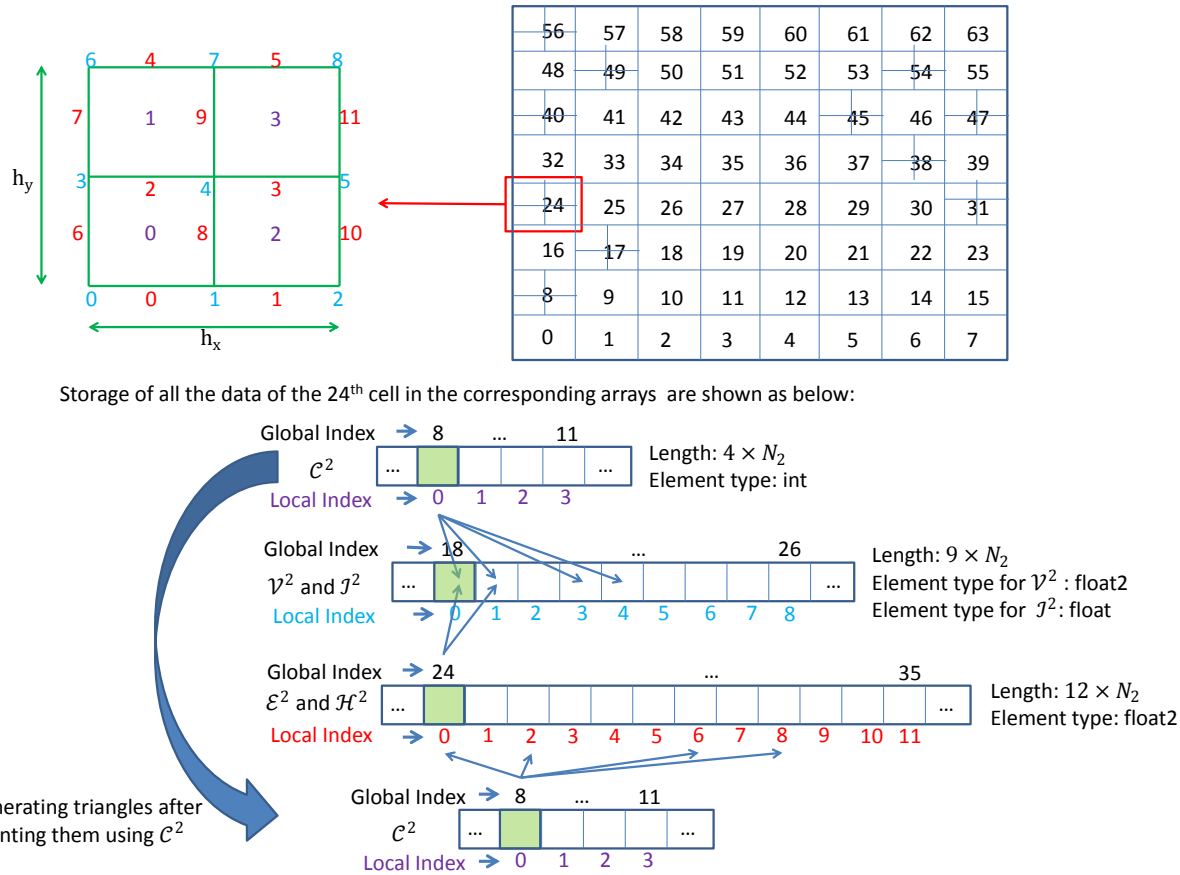


Figure 3: Storage of all data for 24^{th} cell. The top-left shows the local indices and coordinates of vertices, edges and tessellated cells of a level-2 cell. The top-right shows the global cell indices and all the level-2 cells extracted from Figure 2. Bottom: Array structures $\mathscr{C}^2$, $\mathscr{V}^2$, $\mathscr{I}^2$, $\mathscr{E}^2$ and mutual relationships for the 24^{th} cell.

the corresponding level. For example, as shown in Figure 3, the 24^{th} cell is a level-2 cell and also the 3^{rd} level-2 cell globally. In order to store the data of the 24^{th} cell, we should find the start and end global indices in the corresponding arrays concerning the 3^{rd} level-2 cell. We also use local indices shown in different colors in Figure 3 to assist our parallel access algorithm. For $\mathscr{V}^2$ and $\mathscr{I}^2$, the start and end global indices are 18 and 26. In the elements prior to index 18, we store the corresponding vertex information of the first two level-2 cells. The local indices are from 0 to 8 and each index points to a vertex with the same index shown in the top-left of Figure 3. Similarly, the start and end global indices and the corresponding local indices for $\mathscr{E}^2$ and $\mathscr{C}^2$ are shown in Figure 3.

Since the manipulations of all threads in a kernel function are the same, we illustrate the first corresponding elements of the 24^{th} cell that are shaded in light green, with all the local indices 0, as shown in Figure 3.

We developed our parallel algorithms based on the two key observations:

- All the elements (vertex, edge and tessellated cell) of a level-2 cell are compactly stored in the corresponding arrays sequentially according to the global indices shown in the top-right of Figure 3.

- In a level-2 cell, the numbers of vertices, edges and tessellated cells are constants, which are equal to 9, 12 and 4, respectively. We use them to construct local indices, from which we can obtain the relative position for a specific element in a level-2 cell as shown in Figure 3. We can also obtain the relationship between different elements whose indices point to different arrays.

3.1 Step 1: Particle Registration

Since we split cells according to the number of particles contained, it is necessary to count the number of particles inside each cell. We use a CUDA kernel function here to do this in parallel. The total number of threads for the function is the same as the number of particles and each thread manipulates a particle. If particle i is computed as in cell j, we increase $C[j]$ (a j^{th} element of cell level array in **Algorithm 1**) by one using CUDA *atomicAdd*() function, which guarantees thread synchronization.

Algorithm 1 Preprocessing

Input:
P: particle position array;
C: cell level array, initialized with 0;
Output:
C: cell level array, containing the level number of each cell;
// *Step 1: Particle Registration*
for all p_i of P in parallel, **do**
  compute which cell p_i is in and increase the corresponding $C[j]$ by 1 using *atomicAdd*();
end for
// *Step 2: Cell Splitting*
for all c_i of C in parallel, **do**
  if $C[i]$ is 1 **then**
    $C[i] \Leftarrow 3$;
  else
    $C[i] \Leftarrow 1$;
  end if
end for
// *Step 3: Cell Level Adjusting*
for all c_i of C in parallel, **do**
  if $C[i]$ is 3 **then**
    for all C_j that has a common face shared with c_i in serial, **do**
      if $C[j]$ is 1 **then**
        $C[j] \Leftarrow 2$;
      end if
    end for
  end if
end for

3.2 Step 2: Cell Splitting

Here we just split a cell into level-3 if the number of particles inside is one as shown in Figure 2 (bottom left). Otherwise the cell is assigned to level-1. We do this because we are interested in surface areas of fluids, especially where splash occurs. A common observation is that splash areas are usually formed by isolated particles, and the corresponding surfaces need to be constructed with finest level cells to fully catch the thin features. On the contrary, in other areas without splash, we just use the coarsest level cells to approximate the corresponding surface. The criterion here is flexible and users can tune it to obtain surfaces of different adaptive extent. For example, in Section 5, we apply our algorithm to a 2-scale adaptive SPH fluids, where the criterion of cell splitting is based on the number of fine particles inside each cell and users can tune the number to adjust the adaptive extent of the generated surface.

3.3 Step 3: Cell Level Adjusting

For all level-3 cells, we adjust all the neighbor cells with which a common face is shared to level-2 as shown in Figure 2 (bottom right). Consequently, the level difference between two cells is less than 1. We do this for two reasons: First, a smoother surface will be gained by splitting the former level-1 cells to level-2; second, it makes the later crack filling process simpler.

Algorithm 2 MC Surface Reconstruction for Level-2 Cells

Input:
$\mathscr{V}^2$: vertex coordinate index array;
$\mathscr{I}^2$: cell vertex implicit value array;
$\mathscr{E}^2$: cell edge vertex array;
$\mathscr{H}^2$: cell edge vertex normal vector array;
$\mathscr{C}^2$: triangle number array;
Output:
$\mathscr{E}^2$: cell edge vertex array;
$\mathscr{T}^2$: triangle array;
// *Step 4: Cell Vertex Coordinates and Implicit Field Computation*
for all v_i of $\mathscr{V}^2$ in parallel, **do**
  compute the coordinate $\mathscr{V}^2[i]$;
  compute implicit value $\mathscr{I}^2[i]$;
end for
// *Step 5: Triangle Number Counting of Tessellated Cells According to MC Patterns*
for all c_i of $\mathscr{C}^2$ in parallel, **do**
  find the coordinates of end vertices of tessellated cell c_i from $\mathscr{V}^2$;
  count the number of triangles generated by MC patterns and store it in $\mathscr{C}^2[i]$;
end for
scan and compact $\mathscr{C}^2$;
// *Step 6: Edge Vertex Computation*
for all e_i of $\mathscr{E}^2$ in parallel, **do**
  access the two vertices of edge e_i from $\mathscr{V}^2$;
  compute the coordinate and normal vector of edge vertex and store them in $\mathscr{E}^2[i]$ and $\mathscr{H}^2[i]$;
end for
scan and compact $\mathscr{E}^2$;
// *Step 7: Triangle Generation According to MC Patterns*
for all c_i of $\mathscr{C}^2$ in parallel, **do**
  access the edge vertices of tessellated cell c_i from $\mathscr{E}^2$;
  generate triangles inside it and store them in $\mathscr{T}^2$;
end for

3.4 Step 4: Cell Vertex Coordinates and Implicit Field Computation

In this step, the coordinates and implicit values of cell vertices are computed and stored in array $\mathscr{V}^2$ and $\mathscr{I}^2$, respectively. We initialize $\mathscr{V}^2$ with global cell indices as shown in the top-right of Figure 3. For each $v_i \in \mathscr{V}^2$ in parallel, we first compute its local index from its global index and find its relative position in a level-

2 cell. Then its coordinate and a value of an implicit function are computed and stored in $\mathscr{V}^2[i]$ and $\mathscr{I}^2[i]$, respectively. Here, for simplicity, we just use the SPH density function [Müller03] as shown in Equation (1) to compute the implicit values on cell vertices as,

$$\mathscr{I}^2[i] = \sum_j m_j W(\mathscr{V}^2[i] - \mathbf{p}_j, h), \quad (1)$$

where m_j is the mass of neighboring fluid particles within supporting radius h, $\mathbf{p}_j$ is the position of neighboring fluid particles, and W is the kernel function. The implicit function used here is just a simple choice to test our crack-filling algorithm, and other choices to build smooth surfaces of high quality [Zhu05, Yu10, Akinci13] are recommendable.

3.5 Step 5: Triangle Number Counting of Tessellated Cells According to MC Patterns

In this step, the number of triangles generated in each tessellated level-2 cell is stored in array $\mathscr{C}^2$. For each $c_i \in \mathscr{C}^2$ in parallel, we first compute its local index from its global index and find its relative position in a level-2 cell. Then, its four vertices are accessed and triangles generated inside a cell are counted according to MC patterns.

After the kernel function, we scan and compact $\mathscr{C}^2$ using [Harris07] for triangle generation described later.

3.6 Step 6: Edge Vertex Computation

In this step, the edge vertices are computed and stored in array $\mathscr{E}^2$. For each $e_i \in \mathscr{E}^2$ in parallel, we first compute its local index from its global index and find its relative position in a level-2 cell. The corresponding coordinates and implicit values of its two vertices are accessed from $\mathscr{V}^2$ and $\mathscr{I}^2$ and then, an edge vertex whose iso-value is zero is computed.

With conventional methods such as [Lorensen87], the edge vertex is computed using linear interpolation for uniform grids. However, linear interpolation does not perform well in adaptive grids because for the common edges between two cells of different levels, the generated edge vertices from two sides do not coincide, which causes trouble in the later crack filling process. In order to make the edge vertices coincide, we use bi-section interpolation [Press07] which is an asymptotic iterative approximation method. On our 3-level grids, the number of iteration in bi-section is set to $n, n+1, n+2$ for a level-3, level-2 and level-1 edges, respectively, where n is a user-defined parameter. Here we set $n = 5$.

After the coordinate of edge vertex is computed, we compute its normal vector for rendering. With conventional methods, the normal vector of an edge vertex is computed by first computing the normal vectors of triangles that are adjacent to it and then taking the average of those vectors. However, that method is not straightforward to be used here because the triangles adjacent to an edge vertex may belong to different levels, which are stored in different arrays and cannot be accessed straightforwardly. In order to resolve this problem, for an edge vertex i, we just use Equation (2) derived in the method as [Müller03],

$$\mathscr{H}^2[i] = \sum_j m_j \nabla W(\mathscr{V}^2[i] - \mathbf{p}_j, h), \quad (2)$$

to compute the gradient of Equation (1) as its normal vector and store it in $\mathscr{H}^2[i]$. Then $\mathscr{H}^2[i]$ is normalized.

Finally, we scan and compact $\mathscr{E}^2$ and $\mathscr{H}^2$ using [Harris07] for the triangle generation described later.

3.7 Step 7: Triangle Generation According to MC Patterns

In this step, the triangles are generated and stored in array $\mathscr{T}^2$. The $\mathscr{C}^2$ here is after compaction. For each $c_i \in \mathscr{C}^2$ in parallel, we first obtain its global index from the corresponding scanned array and access the vertices generated on its four edges from compacted $\mathscr{E}^2$. Finally, the triangles are generated according to MC patterns.

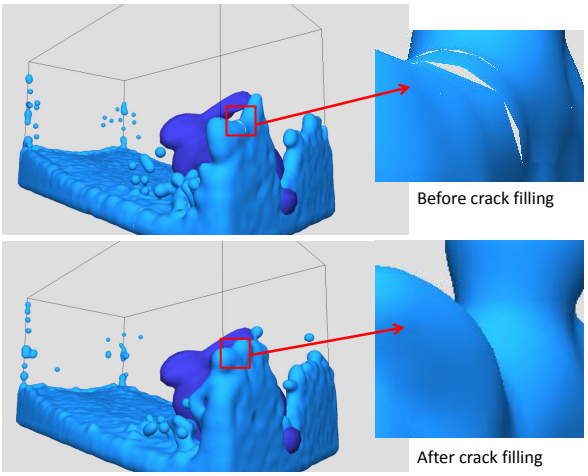


Figure 4: Top: Fluid surface before crack filling. Bottom: Fluid surface after crack filling.

4 PARALLEL CRACK FILLING USING PRE-DEFINED PATTERNS

The surfaces reconstructed in Section 3 contain cracks and the biggest challenge for our adaptive surface reconstruction algorithm is crack filling as shown in Figure 4. Unlike prior MC based methods detecting and filling cracks during program running [Shu95, Shekhar96, Westermann99, Akinci13], our algorithm has two distinguished features for parallel

implementation. First, we present an intuitive and robust method to detect all the possible cracks, including their shapes, constructions and triangulations. Secondly, we pre-define all the cracks as patterns before the running phase. If cracks occur during running, they are filled using the corresponding pre-defined patterns.

In this section, we first explain the reason why cracks occur. Then we propose our method to analyze and define all the possible crack patterns. Finally, we propose the parallel implementation of crack filling on GPUs.

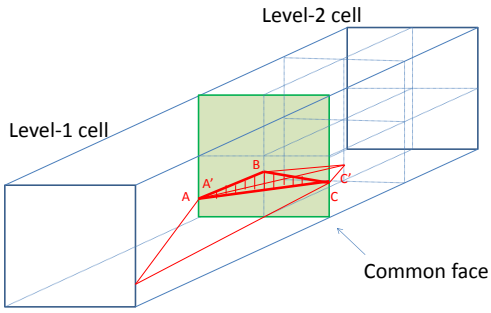


Figure 5: Crack (triangle ABC) generated on common face (green) between level-1 cell and level-2 cell.

4.1 Causes of Cracks

As described in prior works such as [Akinci13, Shu95], cracks occur in the common faces between two grid cells of different levels if different edge points are generated on two sides. As shown in Figure 5, on the common face between a level-1 and a level-2 cell, the edge points generated from the level-1 side are points A and C while the edge points generated from the level-2 side are point A', B and C'. As discussed in Section 3.6, A coincides with A' and C coincides with C' so that there are only three distinctive points. As a result of MC algorithm, the iso-curve of the level-1 side is line segment AC while the iso-curves of the level-2 side are line segments AB and BC. Most of the time, AB and BC are not in the same line so that they cannot coincide with AC, which generates a crack as triangle ABC.

4.2 Definition of All Possible Crack Patterns

As discussed above, cracks occur on the common face between different level grid cells and are defined by the edge points from two sides. Also, as shown in Section 3.6, the edge points coincide with the common edges between different level grid cells. That is to say, if an edge point is generated in the low level edge side, there must be an identical edge point generated in the corresponding high level edge side. So we only care about the edge points generated on the high level side when analyzing crack patterns. Since the level difference between two different level cells is exactly one, there are only two conditions: A level-1 cell is adjacent to a level-2 cell; a level-2 cell is adjacent to a level-3 cell. We treat the common face between a level-2 and a level-3 grid cell as four small faces between a level-1 cell and a level-2 cell, so it is sufficient to analyze only the first condition.

Due to the paper length limit, the method of determining all the possible crack patterns is described in the supplemental material in detail.

Algorithm 3 Crack Detection and Filling for level-1 and level-2 cells in x direction.

Input:
C: cell level array;
$\mathscr{E}^2$: edge point array of level-2 cells after compaction;
Output:
$\mathscr{F}^2_{12x}$: array to store triangles for crack filling for level-1 and level-2 cells in x direction;

1: **for all** c_i of C in parallel, **do**
2: find the neighboring level-1 and level-2 cells in x direction;
3: find edge points from $\mathscr{E}^2$ on the level-2 cell side;
4: compute crack filling triangles and store them in $\mathscr{F}^2_{12x}$ according to crack patterns;
5: **end for**

4.3 Storage of Crack Patterns

As defined in our patterns, there are at most 12 triangles necessary to fill a crack. Since each triangle is represented by the indices of its three vertices and we also need an integer to store the number of triangles, $3 \times 12 + 1 = 37$ integers are needed to represent a pattern as an element in the pattern array, whose length is 2^{12}. The storage method is shown in Figure 6. The first number shows that three triangles are needed to fill the crack and the following nine numbers represent the local indices of three triangles.

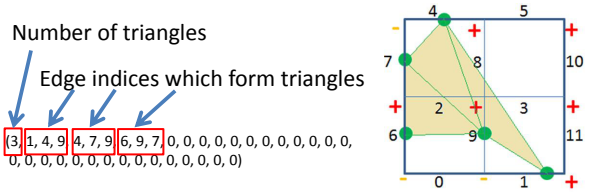


Figure 6: Storage of crack information.

4.4 Parallel Implementation of Crack Detection and Filling on GPUs

In 2D case for our implementation, we use four arrays: $\mathscr{F}^2_{12x}$, $\mathscr{F}^2_{12y}$, $\mathscr{F}^2_{23x}$ and $\mathscr{F}^2_{23y}$, to store the crack filling triangles. They fill the cracks between level-1 and level-2 cells, and the cracks between level-2 and level-3 cracks, in x and y directions, respectively. We use four CUDA

kernel functions for crack filling. Each function manipulates an array. The algorithm of the functions is described in **Algorithm 3**. Here $\mathscr{F}^2_{12x}$ is used as an example and a triangle is stored as three indices to the compacted $\mathscr{E}^2$ array.

In the parallel checking of common faces between two different level cells, we also define the local indices on the common face as patterns in advance. In 2D, there are four conditions: level-1 cell on the left and level-2 cell on the right and vice versa, and level-2 cell on the left and level-3 cell on the right and vice versa.

After obtaining the edge indices, we check whether there are edge points in the edges or not and, if there are, we find the corresponding crack pattern. Finally, we use triangles to fill the crack according to the pattern.

5 RESULTS AND DISCUSSION

We implemented our algorithms on an Intel® Core™ i7-2600 CPU with 24GB RAM and an NVIDIA® GeForce® GTX TITAN GPU with 6GB VRAM. The program was written in C++ and NVIDIA® CUDA [CUDA07].

To optimize our implementation, the data to be rendered like triangles, edge points and normal vectors are all stored using OpenGL VBOs. Manipulations on the VBOs can be switched between CUDA kernel functions and graphics render pipeline using CUDA Resource APIs without data transfer between CPU and GPU.

Due to the limit of GPU VRAM, we only demonstrate our algorithm on small examples, with the number of triangles up to 100k and the number of SPH particles up to 16k. Large-scale examples can be implemented on multi-GPU environment if the readers are interested. The kernel radius for implicit function sampling equals the kernel radius of SPH particles, and the time step is fixed to 0.005.

The results of applying our GPU adaptive surface reconstruction algorithm to dam breaking simulation are shown in Figure 7. The left and right columns show the wireframe and shaded versions of adaptive surfaces reconstructed by our algorithm. The three images from top to bottom of each column are screenshots of 100^{th}, 200^{th}, and 500^{th} frame of the corresponding simulation.

The grid used in the simulation starts with uniform level-1 cells with a resolution of $81 \times 43 \times 43$. During simulation, some level-1 cells are split into level-2 and level-3 cells and then adaptive surfaces are generated. We also compare the adaptive surfaces with their corresponding uniform ones generated from a uniform grid whose resolution is $321 \times 171 \times 171$.

The comparison of their computation time is shown in Figure 8(a) and the comparison of their number of triangles is shown in Figure 8(b). As shown in Figure 8(a), the time required for surface construction is shorter in our approach, and also as shown in Figure 8(b), the number of triangles in our approach is considerably less than the uniform MC algorithm.

In Figure 8(c), different parts of computation time for adaptive surface reconstruction are shown. It can be seen that the parts for computing scalar values on implicit field are increasing. However, the crack filling part that does not require implicit field calculation remains nearly unchanged. Figure 8(d) shows the number of triangles in different levels. The biggest number comes from level-3 triangles, which are generated from level-3 grid cells. Other three kinds of triangles are nearly equal.

Figure 9 is another example of our approach. The left is adaptive surfaces displayed in wireframe and the right is the same surface but shaded. It can be seen that the splash areas of the adaptive surface are all preserved.

6 CONCLUSION AND FUTURE WORK

We propose a novel GPU-based adaptive surface reconstruction algorithm for SPH fluids which can produce surface meshes similar to those generated by the conventional MC method with significantly reduced computation time and memory usage. Our algorithm has the following unique features: (1) MC algorithm is implemented on a 3-level grid level by level in parallel; (2) All the possible cracks, including their shapes, construction vertices, and triangulations, are analyzed in advance and stored as crack patterns; (3) The cracks are detected and filled according to crack patterns in parallel. These features make our algorithm robust in crack filling, simple and easy to implement on GPU, and also fast and effective in performance.

In the future, we think our algorithm can be used in GPU-based parallel surface tracking and topology fixing for adaptive triangle meshes. This is done by extending and optimizing some previous works which apply pre-defined patterns for surface topology fixing, such as [Müller09]. In [Müller09], the surface mesh is tracked on uniform grids and the implementation is on CPU, which is too slow to run in real-time. It can be extended to adaptive grids and optimized by GPU implementation based on our algorithm, which generates real-time performance and interactive applications for surface tracking.

7 ACKNOWLEDGEMENTS

We would like to thank all the reviewers who provided valuable comments. We would also like to thank Prof. Makoto Fujisawa for disclosing the source code for GPU-based SPH simulation to the public.

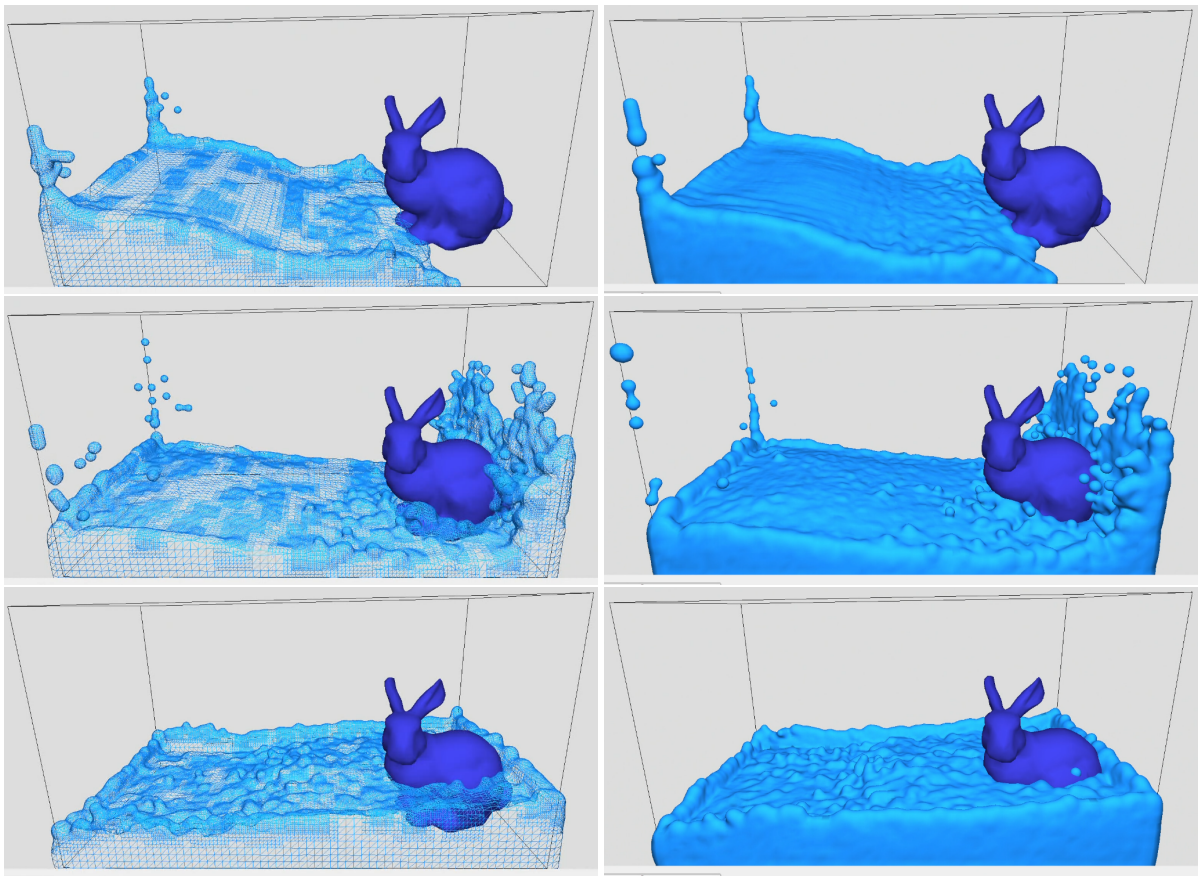
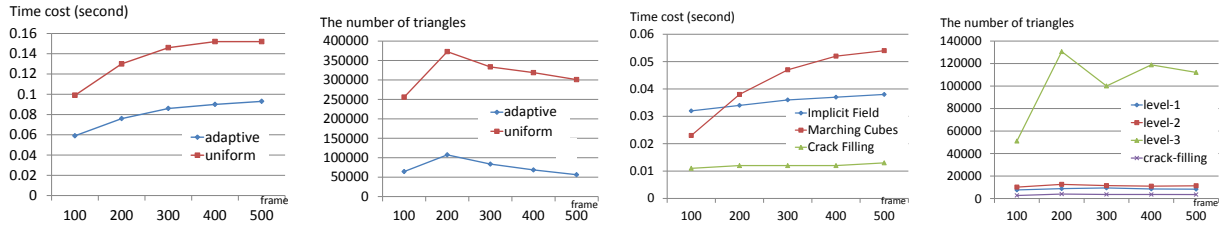


Figure 7: The result of surface reconstruction on adaptive SPH water.



(a) (b) (c) (d)

Figure 8: Statistics for Figure 7. (a) Computation time for uniform and adaptive surface reconstruction. (b) The number of triangles for uniform and adaptive surface reconstruction. (c) Computation time for each process. (d) The number of triangles in different levels of cells.

8 REFERENCES

[Adams07] Bart Adams, Mark Pauly, Richard Keiser, Leonidas J. Guibas: Adaptively sampled particle fluids. ACM Trans. Graph. 26(3): 48 (2007)

[Akinci12] Gizem Akinci, Markus Ihmsen, Nadir Akinci, Matthias Teschner: Parallel Surface Reconstruction for Particle-Based Fluids. Comput. Graph. Forum 31(6): 1797-1809 (2012)

[Akinci13] Gizem Akinci, Nadir Akinci, Edgar Oswald, Matthias Teschner: Adaptive Surface Reconstruction for SPH using 3-Level Uniform Grids. Proc. WSCG 2013: 195-204 (2013)

[Ando13] Ryoichi Ando, Nils Thürey, Chris Wojtan: Highly adaptive liquid simulations on tetrahedral meshes. ACM Trans. Graph. 32(4): 103 (2013)

[CUDA07] nVIDIA CUDA Compute Unified Device Architecture - Programming Guide, nVIDIA Corporation, (2007)

[Desbrun96] Mathieu Desbrun, Nicolas Tsingos, Marie-Paule Gascuel: Adaptive Sampling of Implicit Surfaces for Interactive Modelling and Animation. Comput. Graph. Forum 15(5): 319-325 (1996)

[Dyken08] Christopher Dyken, Gernot Ziegler, Christian Theobalt, Hans-Peter Seidel: High-speed Marching Cubes using HistoPyramids. Comput. Graph. Forum 27(8): 2028-2039 (2008)

[Gingold77] R. A. Gingold and Joe Monaghan,

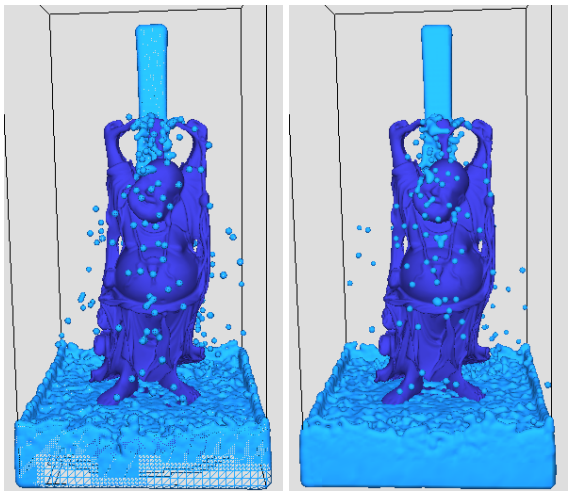


Figure 9: Water is poured on Happy Buddha.

Smoothed Particle Hydrodynamics - Theory and Application to Non-spherical Stars, Monthly Notices of the Royal Astronomical Society, 181, 375-389 (1977)

[Harada07] Takahiro Harada, Seiichi Koshizuka, Yoichiro Kawaguchi, Smoothed Particle Hydrodynamics on GPUs, Proc. of Computer Graphics International, 63-70 (2007)

[Harris07] Mark Harris and Shubhabrata Sengupta and John D. Owens, Parallel Prefix Sum (Scan) with CUDA, GPU Gems 3, (2007)

[Ho05] Chien-Chang Ho, Fu-Che Wu, Bing-Yu Chen, Yung-Yu Chuang, Ming Ouhyoung: Cubical Marching Squares: Adaptive Feature Preserving Surface Extraction from Volume Data. Comput. Graph. Forum 24(3): 537-545 (2005)

[Ihmsen14] M. Ihmsen, J. Orthmann, B. Solenthaler, A. Kolb, M. Teschner, "SPH Fluids in Computer Graphics," State-of-the-Art Report, Eurographics, 2014.

[Ju02] Tao Ju, Frank Losasso, Scott Schaefer, Joe D. Warren: Dual contouring of hermite data. ACM Trans. Graph. 21(3): 339-346 (2002)

[Lorensen87] William E. Lorensen, Harvey E. Cline: Marching cubes: A high resolution 3D surface construction algorithm. SIGGRAPH (1987): 163-169

[Lucy77] L. B. Lucy. A numerical approach to the testing of the fission hypothesis. The Astronomical Journal, 82:10131024, (1977).

[Müller03] Matthias Müller, Barbara Solenthaler, Richard Keiser, and Markus Gross. 2005. Particle-based fluid-fluid interaction. In Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation: 237-244 (2003)

[Müller09] Matthias Müller. 2009. Fast and robust tracking of fluid surfaces. In Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation: 237-245 (2009)

[Press07] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery: Numerical Recipes: The Art of Scientific Computing (3rd Edition), Cambridge University Press (2007)

[Schaefer07] Scott Schaefer, Tao Ju, Joe D. Warren: Manifold Dual Contouring. IEEE Trans. Vis. Comput. Graph. 13(3): 610-619 (2007)

[Schmitz09] Leonardo A. Schmitz, Carlos A. Dietrich, Joäo Luiz Dihl Comba: Efficient and High Quality Contouring of Isosurfaces on Uniform Grids. SIBGRAPI (2009): 64-71

[Shekhar96] Raj Shekhar, Elias Fayyad, Roni Yagel, J. Fredrick Cornhill: Octree-Based Decimation of Marching Cubes Surfaces. IEEE Visualization (1996): 335-342

[Shu95] Renben Shu, Chen Zhou, Mohan S. Kankanhalli: Adaptive marching cubes. The Visual Computer 11(4): 202-217 (1995)

[Solenthaler07] Barbara Solenthaler, Jürg Schläfli, Renato Pajarola: A unified particle model for fluid-solid interactions. Journal of Visualization and Computer Animation 18(1): 69-82 (2007)

[Westermann99] Rüdiger Westermann, Leif Kobbelt, Thomas Ertl: Real-time exploration of regular volume data by adaptive reconstruction of iso-surfaces. The Visual Computer 15(2): 100-111 (1999)

[Yu10] Jihun Yu, Greg Turk: Reconstructing Surfaces of Particle-Based Fluids Using Anisotropic Kernels. Symposium on Computer Animation (2010): 217-225

[Zhou11] Kun Zhou, Minmin Gong, Xin Huang, Baining Guo: Data-Parallel Octrees for Surface Reconstruction. IEEE Trans. Vis. Comput. Graph. 17(5): 669-681 (2011)

[Zhu05] Yongning Zhu, Robert Bridson: Animating sand as a fluid. ACM Trans. Graph. 24(3): 965-972 (2005)