Pacific Graphics 2015
N. J. Mitra, J. Stam, and K. Xu
(Guest Editors)

Volume 34 (2015), Number 7

Grid-Free Surface Tracking on the GPU

1060

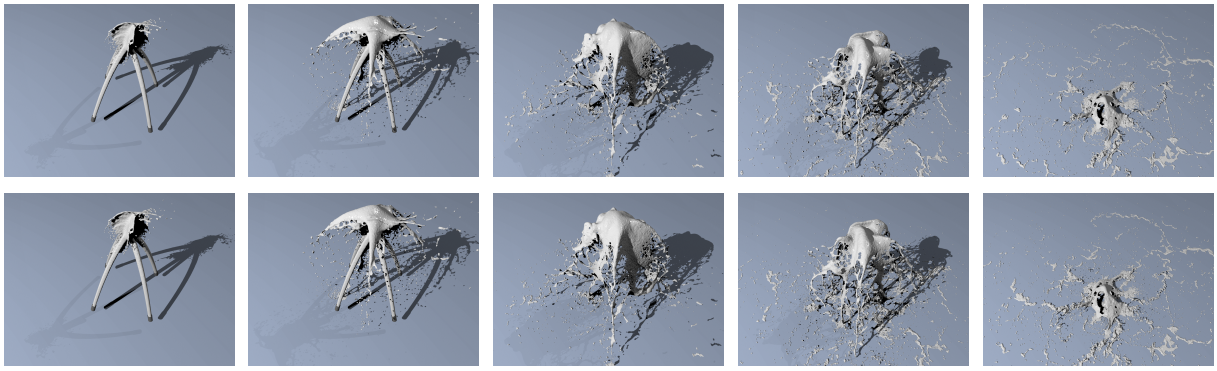

Figure 1: *Four jets of liquid simulated with a particle-based simulation. Top: Computed with our GPU implementation. Bottom: The result of the CPU implementation of Chentanez et al. [CM]. The methods produce similar meshes throughout the simulation. Our parallel self-intersection removal step is 75 times faster and the mesh improvement step 20 times faster than the sequential version. The overall speedup for this example is 50 times.*

Abstract

We present the first mesh-based surface tracker that runs entirely on the GPU. The surface tracker is both completely grid-free and fast which makes it suitable for the use in a large, unbounded domain. The key idea for handling topological changes is to detect and delete overlapping triangles as well as triangles that lie inside the volume. The holes are then joined or closed in a robust and efficient manner. Good mesh quality is maintained by a mesh improvement algorithm. In this paper we describe how all these steps can be parallelized to run efficiently on a GPU. The surface tracker is guaranteed to produce a manifold mesh without boundary. Our results show the quality and efficiency of the method in both Eulerian and Lagrangian liquid simulations. Our parallel implementation runs more than an order of magnitude faster than the CPU version.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: —Computational Geometry and Object Modeling Physically Based Modeling I.3.7 [Computer Graphics]: —Three-Dimensional Graphics and Realism Animation and Virtual Reality

1. Introduction

In the past years, mesh based surface tracking has become a relevant alternative to implicit methods in computer graphics. Most recently, Chentanez et al. [CM] proposed a method that is completely grid-free. To fix a mesh after the advection of the vertices, they simply remove all intersecting triangles and the triangles that are inside the liquid volume. After this they triangulate the resulting holes to get a new closed and manifold mesh.

The goal of our project was to parallelize this approach to make it run entirely on a GPU. For this we had to efficiently parallelize each individual step. For some of the steps, efficient parallel methods were available. In most cases, however, we had to devise new parallel algorithms which we will present in this paper. Most of them are not limited to the special case of surface tracking but are useful in other related areas as well. Our main contributions include

- A parallel hole pairing algorithm

- A parallel hole filling algorithm
- A parallel algorithm for mesh improvement

To combine these components, our method makes intensive use of dynamic parallelism [S.12], [HM14], i.e. the possibility of kernels to launch other kernels directly on the GPU. This removes the need of expensive memory transfers between kernel calls.

2. Related Work

Various approaches for mesh based surface tracking have been proposed so far. Müller [Mül09] move triangle mesh vertices along the velocity field and handle topological changes using a regular background grid. The intersections of the mesh and the edges of the grid are computed and then the marching cubes [LC87] method with an extended stencil set is used for extracting the surface. Wojtan et al. [WTGT09] also use a background grid. They identify cells where the topology of the iso-surface defined by the grid differs from the triangle mesh. In those cells they use the marching cubes mesh and stitch it together with the surrounding triangle mesh. Yu et al. [YWTY12] utilize this method to track the liquid surface in particle-based simulations. Wojtan et al. [WTGT10] improve the quality of the their method by using the convex hulls of the vertices instead of the marching cubes mesh in cells with overlap. They also propose subdivision stitching which better preserves features. Brochu and Bridson [BB09] use continuous collision detection (CCD) to detect intersecting triangles during vertex advection and move the vertices to avoid intersections. Their method guarantees an intersection free result, albeit at a high computational cost. Topological merges and splits are handled explicitly by surgical mesh operations. Da et al. [DBG14] extend this approach to handle interfaces between more than two materials. They also introduce a more efficient merging strategy based on vertex snapping. Bernstein et al. [BW13] propose a method for handling topological changes for non-closed, non-manifold meshes. After moving mesh vertices, they compute exact triangle-triangle intersections. For each input triangle, they use constrained Delaunay triangulation [PC89] on the segments created by intersecting triangles. The resulting triangles that are classified as being inside are deleted. Stanculescu et al. [SCC11] move a vertex at a time and incrementally handle topological changes. The distance a vertex can move in one time step is bounded based on the maximum edge length allowed, so as to simplify collision detection. Most recently, Chentanez et al. [CM] propose to delete all intersecting triangles and triangles that are classified as being inside the volume. Topological merges or splits are handled by joining or filling all holes to obtain a manifold mesh. Our method is based on this work with a number of important changes to make it more GPU friendly.

There are a number of GPU based mesh processing works. DeCoro and Tatarchuk [DT07] simplify a mesh on the GPU by binning vertices to grid cells, compute a representative vertex for each cell and re-map triangle indices to use the representative vertices. Degenerate triangles are removed. Navarro et al. [NHS11] represent triangle meshes with triangle-vertex, edge-vertex, and edge-triangle adjacency information. They perform parallel edge flips on the GPU to construct a Delaunay mesh. The first thread that works on an edge performs atomic operations to mark the triangles involved. Other threads that would need to work on a marked triangle are terminated to avoid race conditions. Shontz and Nistor [SN13] simplify meshes with edge collapses by dividing the work between the CPU and the GPU. They lock vertices using test-and-set [AGTV92] to avoid multiple threads working on the same vertex. Nasre et al. [NBP13] propose several GPU algorithms for problems that involve modifying the graph topology including Delaunay mesh refinement. The refinement algorithm deletes triangles nearby a bad triangle, adds a new vertex and triangulates the cavity. They propose a three-stage locking mechanism to prevent GPU threads from fixing overlapping neighborhoods. Gao et al. [GCTH13] proposed a flip-flop algorithm for transforming star-shaped polyhedra to their convex hulls and use them in convex hull construction algorithm. They also ported their algorithm to the GPU and use a locking mechanism similar to Navarro et al. [NHS11] to avoid flipping dependent triangles in the same pass. Papageorgiou and Platis [PP15] simplify a mesh on the GPU using edge collapses. They split the mesh into parts. Each GPU thread block is responsible for identifying non-overlapping neighbors within a part. Edge collapses are then performed on the independent neighbors. The computation is repeated until the target vertex count is reached. Our work utilizes edge collapses in the mesh improvement step to get rid of short edges or edges opposite from small dihedral angles.

Another important component used in our work are GPU hash tables. Lefebvre and Hoppe [LH06] propose the use of a perfect hash function without collisions for a given data set. This results in a very fast lookup, but has an expensive construction time. Alcantara et al. [ASA*09] use a combination of perfect hashing and cuckoo hashing [PR04] to reduce the hash table construction time on the GPU. Garcia et al. [GLHL11] propose a GPU parallel hashing scheme based on Robin Hood hashing [CLM85] which results in lowered construction times and more coherent memory accesses. Moazeni and Sarrafzadeh [MS12] implement a separate chaining hash table on the GPU. They use a lock free mechanism [Mic02] using atomic compare and swap operations. Our hash table algorithms build on ideas from these works.

3. Method

Our algorithm is based on the method of Chentanez et al. [CM]. Its core steps are self-intersection removal and mesh improvement. In this section we describe the changes we made to these steps to make them run efficiently on a GPU. The changes are shown on a high level in Algorithm 1.

Algorithm 1: Overview of the method.

1: Build a list D of intersecting triangles and triangles with bad adjacent dihedral angles (§3.2)
2: **if** *num_relax_iterations* > 0 **then**
3: **for** $i = 1$ to *num_relax_iterations* **do**
4: Remove topological noise via a position based relaxation of all the vertices adjacent to triangles in D (§3.3)
5: **end for**
6: Rebuild D (§3.3)
7: **end if**
8: Append triangles that either are inside the volume or have no valid velocity values to D (§3.4)
9: Delete all triangles in D and generate a list of boundary vertices, B
10: **loop**
11: Ensure that the mesh is manifold (§3.5)
12: Identify holes (§3.6)
13: Pair and fill holes (§3.7 and §3.8)
14: **if** all holes are filled **then**
15: break
16: **else**
17: Delete triangles with edges adjacent to holes that cannot be filled and replace D with the list of these triangles and update B
18: **end if**
19: **end loop**
20: Improve the mesh quality via vertex smoothing, edge splits and edge collapses while making sure that the mesh remains manifold (§3.9)
21: Remove triangles that are marked for deletion, with a parallel compaction
22: Remove vertices that are not referenced by any triangle, using parallel compaction and re-map the vertex index.

3.1. Parallel Lock-Free Hash Tables

We made use of two kinds of hash table in our method. The first kind is a spatial hash table [THM*03] which takes as an input a spatial location and returns objects such as triangles or points in a close neighborhood. The second type implements a key-value map which we use to store attributes on edges or pairs of holes. Both implementations allow parallel accesses except when a hash collision occurs for a write access. In that case, a GPU friendly lock-free mechanism is used to resolve the collision. The read accesses can always be done in parallel.

For a spatial hash table with up to n cells and m slots per cell, we allocate $n(1+m)$ entries. The first n entries store the number of slots used per cell while the rest store the actual data. For a hash table insertion at cell (i,j,k), we first compute the hash value h and then increment the number of slots used in cell h atomically. If the number of slots used is less than m we store the data in a new slot. Otherwise we go to the next cell and retry. For a hash table lookup, we first compute the hash location. Then, we search all slots of that cell and all subsequent cells until we find a cell that uses less than m slots.

To implement a key-value map, we allocate two arrays, one to store keys and one to store the corresponding values. Initially, the entries of the key array are marked as invalid. To insert a new key k, we first compute its hash value h. Then we perform an atomic compare exchange operation with k and the value at position h in the key array. If the result is invalid, location h was not used. In this case we store the value to the value array at position h. If the result is k which means key has been inserted before, we modify the value array at position h. Otherwise we have a hash collision, and we move on to the next entry of the key array. To store a value on an edge (a,b) we use the key $k = a << 32 + b$, where k is a 64bit unsigned integer.

3.2. Parallel Identification of Intersecting Triangles

We use a boolean array to represent the deletion list D with one entry per triangle which is true iff the triangle belongs to D. To identify intersecting triangles we first initialize a spatial hash table. For this we launch a kernel with one thread per triangle. Each thread inserts a triangle reference to all cells that intersect the triangle. We then launch a second kernel, again with one thread per triangle which uses the hash grid to identify intersecting triangles. As soon as a thread identifies an intersection or a bad dihedral angle, it adds the triangle to D and terminates. For the narrow phase we use the same overlap tests as Chentanez et al. [CM].

3.3. Parallel Topological Noise Removal

As Chentanez et al. [CM], we perform a position based relaxation on the mesh when tracking the surface of a particle based liquid simulation. We use averaged Jacobi iterations (5 in our examples) to modify the positions v_i of the vertices. Specifically we store a temporary sum of positions s_i and a weight w_i, both initialized to zero with each vertex. For each triangle (v_i, v_j, v_k) belonging to D, we then compute

$$s_i \leftarrow s_i + \alpha v_i + (1-\alpha)\frac{v_j + v_k}{2}, \qquad w_i \leftarrow w_i + 1, \qquad (1)$$

$$s_j \leftarrow s_j + \alpha v_j + (1-\alpha)\frac{v_i + v_k}{2}, \qquad w_j \leftarrow w_j + 1, \qquad (2)$$

$$s_k \leftarrow s_k + \alpha v_k + (1-\alpha)\frac{v_i + v_j}{2}, \qquad w_k \leftarrow w_k + 1, \qquad (3)$$

in parallel using atomic operations with $\alpha = 0.5$. The vertex positions are then updated in parallel

$$v_i \leftarrow s_i / w_i \qquad (4)$$

if $w_i > 0$. After the smoothing, the spatial hash is updated for all triangles that have at least one adjacent vertex with $w_i > 0$. We then repeat the intersection test described in the previous section but only for the modified triangles against all other triangles. This whole topological noise removal step can be skipped for grid based simulations as the velocity field tends to be smoother.

3.4. Parallel Triangle-Inside-Volume Test

For grid based liquid simulations, we determine whether a location is inside the liquid volume by casting a ray and testing the parity of the intersections with the surface as in Müller [Mül09]. First, three 2D hash tables are created for the triangles projected onto the xy-, xz- and yz- planes by iterating through all triangles in parallel. These hash tables are used to speed up ray - triangle mesh intersections. We then perform inside/outside tests for triangles not in D. If D is determined correctly, it is sufficient to perform one ray cast per cluster of connected triangles not in D instead of checking all triangles individually. To make the method robust against numerical errors however, we perform individual tests for triangles in the q-ring neighborhood of D and perform a fixed number of ray-casts for the clusters outside this region as Chentanez et al. [CM].

To perform these two steps in parallel we first identify the set of vertices V_q in a q-ring neighborhood of D. For this we use a kernel that expands the set from V_i to V_{i+1} and call it q times, re-using the arrays V_i in a ping-pong manner. We then define D_q to be the set of triangles adjacent to V_q.

To find clusters for triangles not in D_q we use an iterative method. First we initialize a cluster index $c[i]$ of vertex i to $c[i] = i$. Then we loop through each triangle (i, j, k) in parallel. If it is not in D_q, we compute $m = \min(c[i], c[j], c[k])$ and $m' = c[c[m]]$. Then we perform $\text{atomicMin}(c[i], m'), \text{atomicMin}(c[j], m')$, and $\text{atomicMin}(c[k], m')$ on the adjacent vertices. Using m' instead of m accelerates cluster expansion. We use 10 iterations in our examples to get clusters of a size that substantially accelerates the inside/outside test. By storing the cluster indices on the vertices instead of the triangles, no triangle-triangle adjacency information is needed.

To test whether an entire cluster is inside the volume we perform ray casts from w of its triangles as follows. Let *num_cast* and *num_inside* be counters for each vertex. We go through each triangle (i, j, k) in parallel. If the triangle is not in D_q, we check if $\textit{num\_cast}[c[i]] < w$. If so, we atomically increment $\textit{num\_cast}[c[i]]$ and recheck if it is still $< w$. If so, we perform ray-casting from its barycenter to a main axis direction that most align with normal. We then atomically increment or decrement $\textit{num\_inside}[c[i]]$ if ray-casting indicates that it is inside or outside, respectively.

Next we add all triangles that are inside the volume to D. For this we can now use the cluster information. Specifically, we add a triangle not in D_q if $\textit{num\_inside}[c[i]] > 0$, i.e. if its cluster is marked to be inside. For the triangles in D_q we have to perform individual ray casts from their barycenter. We use $q = 2$ and $w = 9$ in the Eulerian liquid simulation examples.

When using our surface tracker with particle based liquid simulations the surface removal algorithm is simpler. In this case we delete triangles that have one or more vertices without valid velocity information or fail to be projected onto surface as Chentanez et al. [CM].

In both simulation types we finally delete the triangles in D in parallel by changing their vertex indices to $(-1, -1, -1)$. During this step, we also generate a boolean array B indicating whether a vertex belongs to a deleted triangle. The vertices in B that are still used by an undeleted triangle are boundary vertices. B will allow us to skip a various computations in later steps.

3.5. Parallel Mesh Manifoldness Enforcement

From this step onwards we make extensive use of modern GPU's dynamic parallelism capability which allows kernels to launch other kernels directly on the GPU. There are two main reasons why this feature is required for an efficient implementation. First, some parts of the algorithm require conditional looping the termination conditions of which are computed on the GPU. Second, the number of threads some kernels need to launch is not known in advance but depends on the results of other GPU computations. Without dynamic parallelism, results would need to be read back to the CPU for processing before launching the next kernel, which is inefficient. Instead we launch one initial kernel with one block and one thread that launches all kernels of the remaining steps.

We detect non-manifold vertices after triangle deletion by counting adjacent boundary edges as Chentanez et al. [CM]. To do this in parallel we first construct a lock free unordered edge hash table, H_e, by going through each triangle edge in parallel. If both vertices are in B, we atomically increment the edge count in H_e. We then go through each triangle edge again in parallel and check if the count is 1 which indicates that the edge is a boundary edge. In that case, we store the ID of the adjacent triangle on the boundary edge and atomically increase the boundary edge count of the two adjacent vertices. We also store vertex $\rightarrow$ edge adjacency information. Storing the IDs of the first two boundary edges for each vertex is sufficient because if a vertex is adjacent to more than two boundary edges, it is non-manifold and will be deleted in the next step anyway.

To ensure manifoldness, we go through each triangle in parallel and delete it if it is adjacent to a vertex with a boundary edge count greater than 2. D and B are updated incrementally as well. We use a global boolean variable indicating whether any triangle deletion happened in this step. If so, we loop back to the construction of the unordered edge hash table, edge counting and triangle deletion steps. The loop is repeated until no triangle deletion happens. The entire loop is executed on the GPU without CPU intervention with the use of dynamic parallelism.

3.6. Parallel Hole Identification

In this step, we identify holes which are loops of boundary edges. To do so, we perform parallel clustering of the vertices using the boundary edges as connectivity in a similar manner to the parallel clustering done for ray casting described in Section 3.4. As a stop criterion we check whether

the hole index $c[i]$ of all boundary vertices have not changed in one iteration. This criterion is only tested every 10 iterations.

We then compute the sizes of the holes by going through each vertex i and if it is in B, atomically increment the counter of the hole $c[i]$. In a further pass through all the vertices we check if a vertex is a seed vertex, i.e. has $i = c[i]$ we check if the corresponding loop size is 3. Such a loop can either be a dangling triangle which we delete by adding it to D or a trivial hole which we close by adding a triangle. This can be checked via the ID of the adjacent triangle of the boundary edges of the hole. If the size is greater than 3, we append the seed vertex to a list of holes, and store the hole's sizes in the list as well.

We then launch a kernel with a number of blocks equal to the number of the remaining holes. Each block has only one thread to collect all vertices in the hole using the edge $\rightarrow$ vertex and vertex $\rightarrow$ edge adjacency information starting from the seed vertex. After this step, we have a list consisting of all vertices of all loops which we call L. We also store the ID of the loop each vertex in L belongs to.

3.7. Parallel Hole Pairing

We now consider pairing the remaining holes that represent topological merge or split events. Chentanez et al. [CM] use two scoring criteria to decide whether a pair of holes (h_a, h_b) should be joined or not. The first criterion considers the number of intersecting pairs of triangles that use vertices from h_b and h_a. The second criterion considers the number of vertex pairs (v_i, v_j) with $v_i \in h_a, v_j \in h_b$ and $||v_i - v_j||_2 \leq g_{\max}$. We omit the first criterion because an efficient parallel implementation would be quite involved and we found that the second criteria alone is sufficient if the parameters are chosen properly.

We compute the score of the second criterion by first populating a 3D spatial hash table with a cell size of $g_{\max}$ by all vertices in L in parallel. We then go through each vertex in L in parallel and look up the spatial hash table for nearby boundary vertices. For each close vertex pair we atomically increment the score of the corresponding pair of holes S_i which is stored in a pair hash table. During this step, every time a new pair of holes is encountered, we append it to a hole pair list, $P_i = (a_i, b_i)$.

We sort P with a parallel bitonic merge sort [KW05] using S as the key in ascending order. For each hole h_j, let $h^p[j]$ be the ID of hole pair in P to be used for pairing or -1 if there is none. Let $h^g[j]$ be the temporary guess. We pair up holes in a way that prioritizes the pairs with higher indices and therefore larger scores using the atomic maximization operations in Algorithm 2.

We use $g_{\max} = 3l_{\max}$ and $\beta = 0.25$ in all of our particle based examples and $g_{\max} = 2l_{\max}$ and $\beta = 0.25$ for our grid based example as opposed to $g_{\max} = 2l_{\max}$ and $\beta = 1$ used by Chentanez et al. [CM]. The lower β is used because we do not use the first scoring criterion. We chose a larger $g_{\max}$ in the particle-based example because global vertex smoothing, to be described Section 3.9, can move vertices by a small distance. A larger $g_{\max}$ allows correctly pairing even if vertices move slightly relative to each other.

Algorithm 2: Hole Pairing Algorithm

```
h^p[j] ← −1, in parallel
loop
    h^g[j] ← −1, in parallel
    if h^p[a_i] = −1 and h^p[b_i] = −1 and
    S_i ≥ β max(|h_{a_i}|, |h_{b_i}|) then
        atomicMax(h^g[a_i], i)
        atomicMax(h^g[b_i], i)
    end if, in parallel
    if h^g[a_i] = i and h^g[b_i] = i then
        h^p[a_i] ← i
        h^p[b_i] ← i
    end if, in parallel
    break if nothing was changed
end loop
```

We are now ready to merge the pairs of holes. To merge a pair, we identify the closest legal pair of edges e_a and e_b, one from each hole. A pair of edges is legal if the quad formed by the two edges can be triangulated with manifold edges. Whether an edge is manifold can be checked by looking up H_e to see if it exists in the mesh no more than once.

We identify the closest legal pairs of edges for all pairs of holes in parallel by launching a kernel with the number of blocks equal to the number of holes, each with $t = 512$ threads. For a hole h_a, the first thread checks whether h_a appears first in the pair $P[h^p[a]]$ to make sure that only one thread works on the pair. In this case, let h_b be the hole h_a is to be joined with. Each thread of the block is responsible for up to $\lceil |h_a|/t \rceil$ edges of h_a. Each thread searches for the closest legal edge in e_b by trying all possible choices in $O(|h_b|)$ time. We then perform parallel reduction within each block to obtain the closest legal pair. If the closest legal pair does not exist, i. e. all pairs of edges searched yield non-manifold triangulations, we skip merging this pair of holes. Otherwise, we generate a new hole with a single boundary by adding a quad between the closest edges and append the new hole vertices to L. This is done in parallel by utilizing the t threads. We then mark the holes h_a and h_b to be ignored for the remaining computation. Finally we update H_e to include the edges of the quad.

3.8. Parallel Hole Filling

At this point, all the holes, both the ones that are the result of joining two holes and the unjoined holes need to be triangulated. The hole filling kernel creates one block per hole and the threads of a block work together in parallel to fill each hole.

As in Chentanez et al. [CM], our hole filling algorithm is conceptually based on a recursive procedure. To compute the

optimal triangulation and cost $W(i,j)$ of a polygon with vertices $i..j$, it finds the vertex $k = K(i,j) \in \{i..j\}$ such that the combination of the optimal triangulations of polygon $(i..k)$, the triangle (i,k,j) and the optimal triangulation of the polygon $(k..j)$ forms the optimal triangulation (see Figure 2). The exponential running time of this algorithm can be reduced to $O(n^3)$ by dynamic programming, i.e. storing intermediate results in a table, where n is the number of vertices in the hole. Chentanez et al. use heuristics to reduce the time complexity further to $O(n^2)$ computing an approximation of the optimal result. Together with their optimality criterion they find a triangulation that yields a manifold mesh and minimizes the sum of squared edges lengths.

A first approach to construct a parallel hole filling algorithm would be to parallelize the original $O(n^3)$ algorithm which computes the optimal result as shown in Algorithm 3. Parallelization reduces its time complexity from $O(n^3)$ to $O(n^2)$ given an unbounded number of threads. However, in practice, holes with several thousands vertices can appear in which case even a quadratic algorithm would be too slow and kill the speedup of the GPU based surface tracker. We still state the algorithm for readers with less time critical applications but who need the true optimum.

Algorithm 3: An $O(n^2)$ parallel bottom-up dynamic programming algorithm for hole filling

```
1:  for i = 1 to n in parallel do
2:    for j = 1 to n do
3:      legal(i,j) ← edge (i,j) exists at most once in the
        surrounding mesh (accelerated using H_e)
4:      W(i,j) ← ∞, K(i,j) ← −1
5:    end for
6:    synchronize
7:    W(i,i) ← 0, K(i,i) ← i, count ← 0
8:    for j = 1 to n do
9:      if legal(i,j) then
10:       W(i,i+2) ← F(i,i+1,i+2)
11:       K(i,i+2) ← i
12:     end if
13:   end for
14:   synchronize
15:   for m = 3 to n do
16:     j ← i+m
17:     if j ≤ n then
18:       for k = i+1 to j−1 do
19:         if legal(i,k) and legal(k,j) and
            W(i,j) > W(i,k)+W(k,j)+F(i,k,j) then
20:           W(i,j) ← W(i,k)+W(k,j)+F(i,k,j)
21:           K(i,j) ← k
22:         end if
23:       end for
24:     end if
25:     synchronize
26:   end for
27: end for
```

Another approach would be to parallelize the algorithm of Chentanez et al. They suggest to use top-down dynamic programming to compute an approximation to the optimal triangulation with a serial time complexity of $O(n^2)$ in practice. Directly parallelizing their solution to obtain an $O(n)$ parallel algorithm would not be suitable for a GPU implementation though because of the need for dynamic threads creation or a centralized work queue. We therefore propose the GPU friendly parallel bottom-up approach stated in Algorithm 4. In the pseudo code,

$$F(i,k,j) = |(i,k)|^2 + |(k,j)|^2 + |(j,i)|^2,$$

max_check_near is the maximum number of candidates for vertex k to check for partition, and *large_prime* is a large prime number. We use *max_check_near* = 30. The algorithm runs independently for each hole. Each kernel block is responsible for filling one hole. The i^{th} thread of each block is responsible for the i^{th} vertex of that hole. The threads in a given block work together to compute all $W(i,j)$ and $K(i,j)$. The hash table H_h, with cell size $h_{\max} = 3l_{\max}$ is built at the beginning and contains all vertices belonging to holes, which are those in L.

We will now discuss the basic ideas behind our new approach. Three observations were made in Chentanez et al. [CM]. The first observation is that $W(i,j)$ tends to be minimized by a vertex $k = K(i,j)$ that is spatially near both vertices i and j. The second observation is that for nearby choices of k, the cost does not differ much, so skipping some of the choices of k most often leads to a nearly optimal choice. The third observation is that if the vertices i and j are far apart, this particular polygon is likely not the optimal choice for the bigger sub-problem. These observations lead to their proposed top-down algorithm.

Our algorithm also utilize these observations, albeit in a bottom-up fashion so as to be more GPU friendly. First, we only consider at most *max_check_near* geometrically close by vertices. Only when this does not lead to a legal triangulation, we try choices of k in a pseudo random order, utilizing *large_prime*, until the first one that yields a legal triangulation is encountered. We then use that legal choice right away. If the number of possible k is $\leq$ *max_check_near* then all choices are considered.

If there are enough threads, namely, one for each i, this parallel algorithm will run in $O(n)$. In practice, however, we can only use up to $t = 512$ threads per block due to hardware register limitation. In that case each thread is responsible for n/t vertices, so the running time becomes $O(n^2/t)$. In all of our examples, most holes have $n < t$ vertices. For the remaining few large holes, n is also never larger than several thousands, so this parallel algorithm has a time complexity close to $O(n)$ in practice. We note again that we compute $W(i,j)$ for all the holes in parallel so this step is substantially faster than the CPU version.

After the $W(i,j)$ of all holes are computed, we generate the triangulation for each hole by tracing the $K(i,j)$. This step can be implemented using a stack data structure and runs

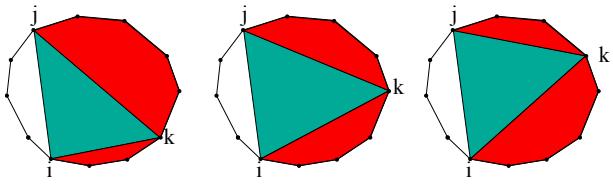

Figure 2: *The optimal triangulation of the polygon $(i..j)$ consists of the optimal triangulation of the polygons $(i..k)$ and $(k..j)$ and the triangle (i,k,j), where $k = K(i,j)$ is the vertex that minimizes the cost function, $W(i,j)$.*

in $O(n)$. We generate triangulations for all holes in parallel using one thread block per hole, each with one thread.

To ensure that the new triangles have no edge longer than $l_{\max}$, edge splits are performed in parallel on the newly generated triangles until all new edge lengths are below $l_{\max}$. This step is the same as what is used for improving mesh quality and is discussed in detail in Section 3.9 except that we allow only edges between two new triangles to be split. Next we perform parallel position based smoothing on the newly generated vertices with $\alpha = 0.7$ for 10 iterations. Finally, the newly generated vertices are projected onto the closest points on the old mesh, if the distance is smaller than $l_{\max}$, by looking up the nearby triangles in the 3D spatial hash table. These steps are similar to Chentanez et al. [CM] but run in parallel.

3.9. Parallel Mesh Improvement

The running time of the algorithm so far is roughly proportional to the number of triangles. Therefore, it is crucial to ensure that the triangles are reasonably well shaped, so that the number of triangles is proportional to surface area. Moreover, having well shape triangles reduces numerical problems in intersection tests and results in less topological noise, both of which slow down the algorithm. We employ three mesh improvement strategies: vertex smoothing, edge splits and edge collapses.

Applying vertex smoothing globally to improve a triangle mesh smoothes out small scale features. Therefore, Chentanez et al. [CM] and Wojtan et al. [WTGT09, WTGT10] do not use it for mesh improvement. However, when using our algorithm with particle based simulations, the vertices are projected onto the iso-surface defined by the particles every time step anyway, so a limited amount of smoothing does not cause a visually significant loss of features. As will be discussed in more details in Section 3.9.2, vertex smoothing can reduce the number of required edge collapse passes significantly. Therefore, for the particles simulation examples, we apply parallel position based relaxation on all vertices with $\alpha = 0.9$ for 5 iterations. As grid based simulations do not cause the mesh quality to degrade as quickly, we do not need to apply vertex smoothing in this case.

Algorithm 4: Our $O(n)$ parallel bottom-up dynamic programming algorithm for computing approximately optimal triangulation.

```
1: for i = 1 to n in parallel do
2:     for j = 1 to n do
3:         legal(i, j) ← edge (i, j) exists at most once in the
           surrounding mesh (accelerated using H_e).
4:         W(i, j) ← ∞, K(i, j) ← −1
5:     end for
6:     synchronize
7:     W(i,i) ← 0, K(i,i) ← i, count ← 0
8:     for j = 1 to n do
9:         if legal(i, j) then
10:            W(i,i+2) ← F(i,i+1,i+2), K(i,i+2) ← i
11:        end if
12:    end for
13:    synchronize
14:    for m = 3 to n do
15:        j ← i+m, all ← false
16:        if (j < n) and |(i, j)| < h_max then
17:            if m ≤ max_check_near then
18:                all ← true
19:            else
20:                for each vertex i < k < j of the same hole within
                   distance h_max to vertex i in H_h do
21:                    if legal(i,k) and legal(k, j) and
                       W(i, j) > W(i,k)+W(k, j)+F(i,k, j) then
22:                        W(i, j) ← W(i,k)+W(k, j)+F(i,k, j)
23:                        K(i, j) ← k
24:                    end if
25:                    count ← count + 1
26:                    if (count > max_check_near) break;
27:                end for
28:            end if
29:        end if
30:        if K(i, j) ← −1 then
31:            l ← m−1
32:            q ← large_prime mod l
33:            for s = i+1 to j−1 do
34:                k ← i+1+q, q ← (q+large_prime) mod l
35:                if legal(i,k) and legal(k, j) and
                   W(i, j) > W(i,k)+W(k, j)+F(i,k, j) then
36:                    W(i, j) ← W(i,k)+W(k, j)+F(i,k, j)
37:                    K(i, j) ← k
38:                    if not all break
39:                end if
40:            end for
41:        end if
42:        synchronize
43:    end for
44: end for
```

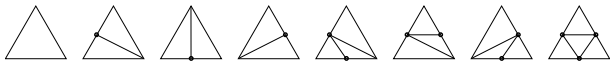

Figure 3: *Stencil table for splitting a triangle.*

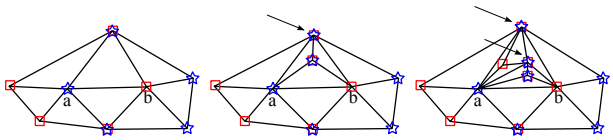

Figure 4: *Collapsing edge (a,b). One-ring neighbors of a and b are marked by red squares and blue stars respectively. Left most) An example of a legal edge collapse. Middle and Right) Examples of illegal edge collapses that result in a non-manifold mesh. Vertices other than the ones opposite from the edge (a,b) that are both neighbor to a and b are pointed by arrows.*

3.9.1. Parallel edge split

Edge splits are usually done by adding a vertex and dividing the two adjacent triangles into four triangles. This approach cannot be parallelized efficiently because adjacent edges are not independent and cannot be split in the same parallel pass. Many passes are therefore required for splitting all long edges in the mesh.

Therefore we use an alternative approach. In a first pass, we go through all edges in parallel and put those that should be split into an edge hash table. An edge should be split if it is longer than $l_{\max}$ and the opposite angle is greater than 10°. The first time an edge is put into the hash table, we create a new vertex at the mid point. In the second pass, we go through all triangles in parallel. We then check which edges of the triangle are split. We then replace the triangle with up to four new triangles by looking up a table as shown in Figure 3. Instead of replacing existing triangles, we set their indices to -1. The new triangles are appended to the triangles indices list with atomic operations. In this way, we can be sure that all new triangles are at the end of the indices list. This is useful in case we want to run further smoothing as done in hole filling. We repeat the two passes to further split edges that are still long, until no new edge is split. In our examples, no more than four iterations are needed, as edges do not become longer very fast in a single time step.

3.9.2. Parallel edge collapse

We consider an edge for potential collapse if it is either shorter than $l_{\min}$ or it is opposite from an angle smaller than 5°. We also need to ensure that collapsing this edge does not create non-manifold edges or non-manifold vertices. This can be checked by ensuring that the one ring neighbors of the two end points of the edge overlap only at the two vertices opposite from the edge, as shown in Figure 4.

To be able to collapse edges independently in parallel, we need to ensure that edges that will be collapsed in the same parallel pass are not within 2-ring neighbors of others. To do so, we have lock_i for each vertex which are initialized to -1. We go through each edge in parallel and check if it should be collapsed, if so, we atomicMax the edge ID on the locks of the two endpoints. We then propagate the lock value to adjacent vertex with atomicMax, so that if two IDs are to be written to the same lock, the one with higher value would win. This can be done for each triangle in parallel. We do the propagation twice. Now, if an edge (i,j) still have lock_i and lock_j equal to its ID, we can be sure that no other edge within 2-ring neighbor would be collapsed in this pass. We mark all such vertices i and j to indicate that they participate in edge collapse in this pass, by checking each edge in parallel.

We now need to ensure that collapsing the edges does not cause non-manifoldness. We check this by going through each triangle (i,j,k) in parallel. If exactly one of its vertices is marked, we atomically increment counters on the other two vertices by one. If the value of a counter after the increment is two or more, the one ring neighbors overlap and hence the edge involving the marked vertices should not be collapsed. In this case, we unmark the previously marked vertex. Now, all the edges (i,j) whose lock_i and lock_j match with the edge ID and whose endpoints are marked can be safely collapsed. We collapse them in parallel by re-mapping all triangles that use the endpoint with the smaller index to the other endpoint and mark triangles with duplicated indices for deletion by changing the indices to -1. We also reposition the vertex with the smaller index to the midpoint.

Not all possible edge collapses are performed in one parallel pass, however, because only edges that are two rings apart from each other are collapsed. In our particle based simulation examples with highly turbulent velocity fields, hundreds of passes are sometimes needed in order to collapse all possible edges. In this case, the edge collapsing step becomes the bottle neck of the entire mesh tracker pipeline. On the other hand, limiting the number of passes per time step to a fixed number like 30 causes the mesh quality to degrade over time, causing numerical problems, a slowdown of the other steps and poor visual quality.

We fix this problem by performing a limited amount of vertex smoothing prior to the edge split and edge collapse steps as already mentioned. This reduces the number of required edge collapse passes from hundreds to tens. In all our examples, we clamp the number of passes to 30 per time step. In grid based simulations, short edges appear more slowly and the smoothing step can be skipped.

Another important check we also do is that we store the edges that we know to cause non-manifoldness in an edge hash table. These edges are not considered for collapses in later passes, by skipping the their locking step. This allow other nearby edges with lower edge ID to be considered instead in later passes.

4. Results

We implemented our algorithm using CUDA and used it for tracking surfaces in a number of examples. For all the tim-

	Self Intersection Removal												Mesh Improvement									
	3DHash		Intersect		Relax&Update		2DHash		DeleteTris		Manifold&Fill		VS		ES		EC		RemUnused		NumTris	
	avg	max	avg	max	avg	max	avg	max	avg	max	avg	max	avg	max	avg	max	avg	max	avg	max	avg	max
Jets	2.46	3.59	27.48	38.87	-	-	1.96	2.80	24.95	46.63	11.30	16.72	-	-	0.84	1.22	31.57	51.92	0.90	1.63	520K	734K
BallSplash	0.92	2.06	8.94	39.48	2.72	12.91	-	-	0.07	0.14	13.82	119.45	0.46	1.94	0.50	1.16	15.84	39.14	0.55	1.31	228K	527K
Fountain	4.58	9.51	43.68	106.26	7.63	20.89	-	-	0.28	0.66	33.33	141.47	1.29	3.72	1.83	4.09	73.05	165.00	1.24	2.42	927K	1733K
Balloon	9.88	14.49	136.99	246.55	50.41	77.08	-	-	0.70	1.13	41.44	120.77	9.34	14.57	3.96	6.08	170.34	272.94	2.13	3.33	1832K	2742K

Table 1: *Breakdown of the running times per step of our method in the examples. All times are in milliseconds. "3DHash" stands for the 3D hash table construction, "Intersect" for computing the initial triangle intersections, "Relax&Update" for topological noise removal, hash table updates and intersections, "2D Hash" for the construction of the 2D hash tables for fast ray casting, "DeleteTris" for deleting intersecting triangles and triangles inside the liquid volume, "Manifold&Fill" for ensuring that the mesh is manifold, join holes and fill holes, "VS" for global vertex smoothing, "ES" for edge splitting, "EC" for edge collapsing, "RemUnused" for removing unused vertices and triangles and "NumTris" for the number of triangles.*

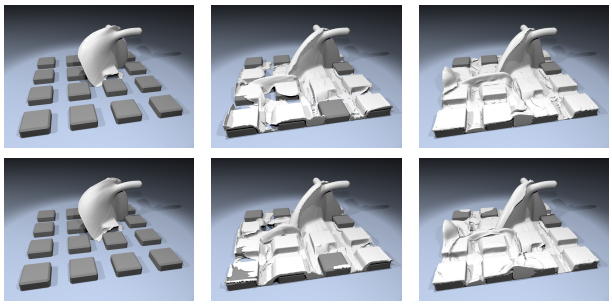

Figure 5: *Two streams of liquid collide in mid-air to form thin sheets before falling on the floor. The scene is simulated with a grid based fluid solver. Top: Our method. Bottom: Chentanez et al. The results are visually similar. Our method is about 28 times faster overall.*

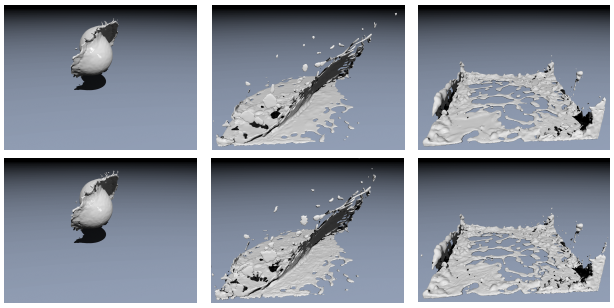

Figure 6: *Two liquid balls colliding in mid-air simulated with a particle-based liquid simulation. Top: Our method. Bottom: Chentanez et al. Again, the results are visually similar. In this case, our method is about 21 times faster overall.*

ings we used an NVIDIA GTX TitanX GPU. Table 1 shows the running times of the steps of our algorithm. We used the same examples and setup as Chentanez et al. [CM] to be able to do a direct comparison.

For a grid based simulation test, our benchmark scene is the two colliding jets example shown in Figure 5. To test our method with a particle-based simulation, we used the scene of two colliding balls shown in Figure 6, the fountain scene shown in Figure 1 and the balloon burst simulation of Figure 7. Table 2 summarizes the speed up of our method relative to the sequential CPU version of Chentanez et al. which was run on a 3.40GHz Intel Core i7-4930K with 16GB of RAM using a single thread. The self-intersection removal step combined with mesh improvement step is between 21 to 50 times faster than the CPU version in our examples. As the figures and the accompanying video show, both works produce meshes of similar quality.

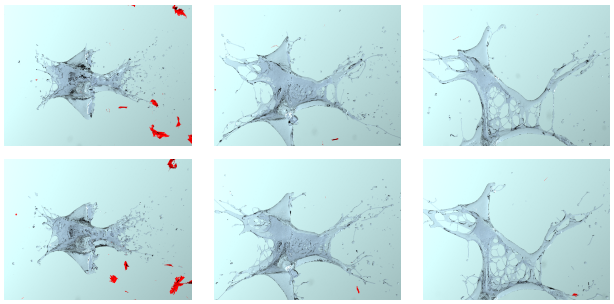

Figure 7: *A balloon filled with liquid is shot by a fast moving projectile simulated with a particle-based based solver. Top: Our method. Bottom: Chentanez et al. Here we achieve a speedup of 29 times.*

5. Conclusion and Discussion

We have presented the first explicit surface tracking method that runs completely on a GPU more than an order of magnitude faster than the serial CPU implementation. For this we had to parallelize all steps individually resulting in new algorithms that can potentially be used in other applications as well. As future work we plan to improve the most criti-

cal part of our pipeline, namely the edge collapse step. The mesh improvement step is currently "only" between 16 and 20 times faster than the CPU counter part reducing the overall speedup. A fast solution would also remove the need of a global smoothing step that could potentially remove some small scale details.

References

[AGTV92] AFEK Y., GAFNI E., TROMP J., VITANYI P.: Wait-free test-and-set. In *Distributed Algorithms*, Segall A., Zaks S., (Eds.), vol. 647 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1992, pp. 85–94. 2

[ASA*09] ALCANTARA D. A., SHARF A., ABBASINEJAD F., SENGUPTA S., MITZENMACHER M., OWENS J. D., AMENTA N.: Real-time parallel hashing on the gpu. In *ACM SIGGRAPH Asia 2009 Papers* (New York, NY, USA, 2009), SIGGRAPH Asia '09, ACM, pp. 154:1–154:9. 2

[BB09] BROCHU T., BRIDSON R.: Robust topological operations for dynamic explicit surfaces. *SIAM Journal on Scientific Computing 31*, 4 (2009), 2472–2493. 2

[BW13] BERNSTEIN G. L., WOJTAN C.: Putting holes in holey geometry: Topology change for arbitrary surfaces. *ACM Trans. Graph. 32*, 4 (July 2013), 34:1–34:12. 2

[CLM85] CELIS P., LARSON P.-Å., MUNRO J. I.: Robin Hood hashing. pp. 281–288. 2

[CM] CHENTANEZ N., M M.:. 1, 2, 3, 4, 5, 6, 7, 9, 10

[DBG14] DA F., BATTY C., GRINSPUN E.: Multimaterial mesh-based surface tracking. *ACM Trans. on Graphics (SIGGRAPH 2014)* (2014). 2

[DT07] DECORO C., TATARCHUK N.: Real-time mesh simplification using the GPU. In *Symposium on Interactive 3D Graphics (I3D)* (Apr. 2007), vol. 2007, p. 6. 2

[GCTH13] GAO M., CAO T.-T., TAN T.-S., HUANG Z.: Flip-flop: Convex hull construction via star-shaped polyhedron in 3d. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2013), I3D '13, ACM, pp. 45–54. 2

[GLHL11] GARCÍA I., LEFEBVRE S., HORNUS S., LASRAM A.: Coherent parallel hashing. *ACM Trans. Graph. 30*, 6 (Dec. 2011), 161:1–161:8. 2

[HM14] HOWES L., MUNSHI A.: The opencl specification. *Khronos OpenCL Working Group* (2014). 2

[KW05] KIPFER P., WESTERMANN R.: Improved GPU sorting. In *GPUGems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation* (2005), Pharr M., (Ed.), Addison-Wesley, pp. 733–746. 5

	CS	CM	GS	GM	SpS	SpM	SpO
Jets	2338	572	70	33	33X	17X	28X
BallSplash	628	319	28	17	22X	19X	21X
Fountain	6880	1468	91	77	75X	19X	50X
Balloon	9646	2767	241	184	40X	15X	29X

Table 2: *Average timing of Chentanez et al. [CM] compared to ours. The C- and G-prefixes indicate times in ms for Chentanez et al. [CM] and our GPU implementation. The S- and M-suffixes indicate the self intersection removal step and the mesh improvement steps respectively. The O-suffix indicates self-intersection removal plus mesh improvement. Sp stands for the speedup of the GPU over the CPU implementation.*

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph. 21*, 4 (Aug. 1987), 163–169. 2

[LH06] LEFEBVRE S., HOPPE H.: Perfect spatial hashing. In *ACM SIGGRAPH 2006 Papers* (New York, NY, USA, 2006), SIGGRAPH '06, ACM, pp. 579–588. 2

[Mic02] MICHAEL M. M.: High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures* (New York, NY, USA, 2002), SPAA '02, ACM, pp. 73–82. 2

[MS12] MOAZENI M., SARRAFZADEH M.: Lock-free hash table on graphics processors. In *Application Accelerators in High Performance Computing (SAAHPC), 2012 Symposium on* (July 2012), pp. 133–136. 2

[Mül09] MÜLLER M.: Fast and robust tracking of fluid surfaces. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (New York, NY, USA, 2009), SCA '09, ACM, pp. 237–245. 2, 4

[NBP13] NASRE R., BURTSCHER M., PINGALI K.: Morph algorithms on gpus. *SIGPLAN Not. 48*, 8 (Feb. 2013), 147–156. 2

[NHS11] NAVARRO C., HITSCHFELD N., SCHEIHING E.: A parallel gpu-based algorithm for delaunay edge-flips. In *27th European Workshop on Computational Geometry (EuroCG)* (Morschach, Switzerland, Mar 2011), Hoffmann M., (Ed.), pp. 75–78. 2

[PC89] PAUL CHEW L.: Constrained delaunay triangulations. *Algorithmica 4*, 1-4 (1989), 97–108. 2

[PP15] PAPAGEORGIOU A., PLATIS N.: Triangular mesh simplification on the gpu. *The Visual Computer 31*, 2 (2015), 235–244. 2

[PR04] PAGH R., RODLER F. F.: Cuckoo hashing. *J. Algorithms 51*, 2 (May 2004), 122–144. 2

[S.12] S. J.: Introduction to dynamic parallelism. *GPU Technology Conference* (2012). 2

[SCC11] STANCULESCU L., CHAINE R., CANI M.-P.: Freestyle: Sculpting meshes with self-adaptive topology. *Comput. Graph.-UK 35*, 3 (June 2011), 614–622. Special Issue: Shape Modeling International (SMI) Conference 2011. 2

[SN13] SHONTZ S., NISTOR D.: Cpu-gpu algorithms for triangular surface mesh simplification. In *Proceedings of the 21st International Meshing Roundtable*, Jiao X., Weill J.-C., (Eds.). Springer Berlin Heidelberg, 2013, pp. 475–492. 2

[THM*03] TESCHNER M., HEIDELBERGER B., MUELLER M., POMERANETS D., GROSS M.: Optimized spatial hashing for collision detection of deformable objects. pp. 47–54. 3

[WTGT09] WOJTAN C., THÜREY N., GROSS M., TURK G.: Deforming meshes that split and merge. *ACM Trans. Graph. 28*, 3 (July 2009), 76:1–76:10. 2, 7

[WTGT10] WOJTAN C., THÜREY N., GROSS M., TURK G.: Physics-inspired topology changes for thin fluid features. *ACM Trans. on Graphics (Proc. SIGGRAPH) 29*, 3 (2010). 2, 7

[YWTY12] YU J., WOJTAN C., TURK G., YAP C.: Explicit mesh surfaces for particle based fluids. *EUROGRAPHICS 2012 30* (2012), 41–48. 2