Symposium On Computer Animation (2009)
Eitan Grinspun and Jessica Hodgins (Editors)

Fast and Robust Tracking of Fluid Surfaces

Matthias Müller

NVIDIA

Abstract

Surface tracking is an important problem with applications in many research fields. Among the most famous examples in computer graphics is the simulation and rendering of liquids with free surfaces. A surface that is advected by a general velocity field constantly changes its topology. This is the main reason why moving surfaces are typically defined implicitly as the zero set of a scalar field rather than by an explicit representation such as a mesh for instance.
In this paper we present a method for tracking fluid surfaces using triangle meshes. This is done in two steps. First, the vertices are advected by the velocity field of the fluid. Second, self-penetrations are fixed using marching cubes triangle templates. The technique is efficient in terms of computation and memory consumption, it is simple to implement and allows for direct control of volume and feature preservation.

1. Introduction

In computer graphics, surfaces of rigid and deformable objects are often represented by explicit triangle meshes. Triangle meshes are simple and efficient data structures. They can be sent directly to GPUs for rendering for instance. In addition, the explicit representation makes direct Lagrangian simulation of cloth or soft bodies straight forward. One simply modifies the vertex positions. Under large deformations, local re-sampling might be necessary as well. Meshes are well suited as long as the topology of the surface does not change. In the case of deformable solids this is guaranteed in most scenarios (except when the bodies are allowed to merge or tear). For solids, the self collision handling process should make sure that surfaces do not self-intersect.

In contrast, the free surface of a liquid constantly changes its topology by splitting and merging with itself. In this scenario, working with explicit representations of the surface gets tricky. A triangle mesh would need to be repaired whenever a self intersection occurs. The common way to circumvent this problem is to use the level set method [Set99]. The main idea is to define the surface implicitly as the zero set of a scalar field – typically its signed distance field. During simulation, the scalar field is advected along a velocity field. In case of fluid simulation, the velocity field is given as the solution of the Navier-Stokes equations. A typical algorithm to simulate free surface fluids in an Eulerian, grid-based setting comprises the following steps:

1. Update the velocity field by solving the Navier Stokes equations on the fluid grid
2. Solve the advection equation on the level set
3. Update the structure of the narrow band grid and extrapolate the level set values
4. Re-normalize the level set
5. Extract a triangle mesh from the level set for rendering using Marching Cubes [LC87]

High resolution simulations of this type are still too expensive to run in real time. By reducing the resolution and using non-physics tricks for preventing volume loss, Crane *et al.*produced impressive results at interactive frame rates though [CLT07].

If the resolution of the level set is higher than the one of the fluid grid, steps 2 - 4 become the bottleneck in both

performance and memory consumption. Also, many people have implemented Stam's stable fluid method [Sta99] but most often without free liquid-air interfaces. We believe that implementing the (particle) level set method is still a hurdle. The goal of our work is to remove steps 2 - 4 by replacing the level set with an explicit triangle mesh which significantly reduces the complexity of the algorithm. Our new method looks like this:

1. Update the velocity field by solving the Navier Stokes equations on the fluid grid
2. Advect the vertices of the surface mesh
3. Make the surface mesh self-collision free

Step 2 is so fast, it can be neglected in a performance analysis. Our method to fix the potentially self-intersecting surface is simple and fast as well.

- Determine the intersections of the mesh with the edges of an implicit regular grid
- For the corners of each intersected cell, determine whether they are inside or outside the surface
- Based on the locations of intersections and inside-outside information, create the triangles of the new mesh using marching cubes templates.

Our method has several advantages over the traditional implicit approach. Even in the narrow band approach, the explicit level set grid stretches several cells away from the surface using considerable amount of memory which we save in our approach. The algorithm is easy to implement and well suited for parallelization. Also, arbitrary speedup is possible by only fixing the mesh every n^{th} frame. In addition, with a surface represented as a triangle mesh, rendering is fast and we can use a simple method to prevent volume loss.

1.1. Related Work

There is a large body of work in the field of surface tracking and the level set method. We refer the reader to the two well known and comprehensive books [Set99] and [OF03] and only discuss a few selected papers here. Surface tracking plays an important role in many fields of research. A few recent examples are modeling of soft tissue deformation in medical simulation [HBAD07], tomography [MM07], molecular dynamics [CDML07] and feature profile evolution in nano technology [KH07].

The level set method was originally developed by [OS88]. An important application of the level set approach in computer graphics is the tracking of the free surface of a liquid simulated on an Eulerian grid [FF01]. [DC98] deployed an implicit representation for the surface for deformable objects.

When used to track the free surface of liquids, the basic level set method suffers from volume loss and blurring of features. To alleviate this problem, Enright *et al.*proposed the particle level set method in [EFFM02, EMF02] and refined it in [ELF05]. Here, particles that are advected explicitly along with the signed distance function are used to counteract numerical dissipation. The method is therefore a combination of explicit and implicit surface tracking. [BGOS06] proposed the semi-Lagrangian contouring method which goes back and forth between an implicit and explicit surface representation to reduce the volume loss problem. In contrast to these methods, our algorithm is purely Lagrangian without the need of alternating between two different representations.

Recently [KLL*07] devised a method for controlling the volume of a liquid defined by a level set. They use a controller which generates an artificial pressure term based on the volume deviation. In contrast, with the explicit mesh, we enforce volume conservation geometrically.

A scalar field defined on a regular grid cannot capture features that are smaller than the spacing of the grid. The main problem in connection with liquid simulation is that thin sheets and puddles disappear unnaturally. [LGF04] addressed this problem by using an oct-tree data structure. In the second part of the paper, we propose a method that can capture subgrid detail without the overhead of dynamically maintaining an oct-tree.

There are two main classes of fully explicit surface tracking methods, grid-free and grid based methods. Both have their advantages and disadvantages.

Brochu *et al.*published the most recent grid free method [BB06, BB09]. The main and important advantage of working without a grid is that features are preserved and the mesh is not altered when translated or rotated. Under large distortions such as stretch, compression and self-intersections, maintaining a good quality mesh becomes tricky though. Brochu *et al.*perform up to 10 sweeps over all triangles per time step. Also, their method only works if the mesh is kept intersection free at all times.

In grid based approaches, a new mesh is generated at each time step by creating new vertices and triangles inside cells of an implicit grid. Here, the mesh structure is altered even under pure translation or rotation. The advantage however is that this approach performs all fixing steps such as resampling and self-intersection removal in one single sweep. Also, the operations within all cells are similar making the method ideal for parallelization on a SIMD machine such as a GPU. Glimm *et al.*propose a hybrid approach [GGL98, GGLT99]. They use a grid free method as long as the mesh is not severely distorted and fall back to a grid based step otherwise.

Since we aim at real-time simulation of free surface liquids, we propose a purely grid-based algorithm because of the simplicity and parallel nature of the technique. As Glimm *et al.*we create new vertices where the mesh intersects the implicit grid. The way we create triangles is more

robust though because we do not need the mesh of the previous step and can handle defect meshes with holes as well.

Recently [LZ09] used a combination of Lagrangian particles instead of triangles in connection with a background grid to track moving interfaces. This is an interesting alternative but it has grid based time stepping restrictions.

Most closely related to our method is the work of [WTGT09]. They also use an explicit triangle mesh to represent the surface and fix topological changes using marching cubes templates on a regular background grid. In contrast to our approach, they only do this in regions where self intersections occur with the advantage of preventing feature loss due to constant re-sampling. However, as in [BB09], explicit mesh maintenance is needed. Explicitly keeping a consistent triangulation between fixed and non-fixed parts adds additional complexity to the method. The goal of our research was to provide a method suitable for real-time use. At the price of constant re-sampling, we can keep the algorithm very simple (at least the basic version). An additional advantage is that the same steps are executed for all cells which simplifies parallelization significantly. An advantage of our extended version is that it can handle self intersections within thin sheets or shallow puddles. Self-intersecting thin features are automatically fixed because our extended template set contains configurations to capture them (see Fig. 4). In [WTGT09] a piece of the mesh is either kept as is or replaced by regular marching cube templates. So self intersecting thin features are either removed completely or remain self intersecting.

2. Core Algorithm

In this section we describe the core algorithm introduced in the previous section in more detail. The surface to be tracked is represented at all times by a closed manifold triangle mesh. At each time step, the vertices of the mesh are first advected. Advection potentially introduces self intersections. Therefore, in a second stage, the mesh is fixed before the next time step starts. This stage does not need to be executed at each time step necessarily. It might be feasible to run and render few advection steps before the self-intersections are resolved. In any case, the problem we have to solve could be stated as follows:

- *Given a closed potentially self-intersecting manifold mesh*
- *Create a closed non-self-intersecting manifold mesh which approximates the outside of the input mesh.*

To solve this problem, a regular grid is used. The size of the cubical cells h is a user-specified parameter. We use a sparse data structure based on spatial hashing [THM*03] and only store the cells that are intersected by the input mesh (see Fig. 2).

First we determine the intersections of the input triangles with the edges of the grid cells. Each intersection has a type

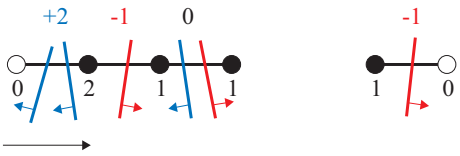

Figure 1: *Entry and exit intersections are summed up in the state change variable of each cell edge to determine whether the grid nodes are inside our outside the surface.*

depending on the triangle normal. If the component of the normal along the edge is positive, the intersection is of type *exit*, otherwise it is of type *entry*. Note that there is potentially more than one intersection per cell edge. Therefore, with each cell edge we store a state change counter which is initialized with zero. For all intersections of that particular edge, the state change variable is increased or decreased by one dependent on whether the intersection is of type *entry* or *exit*, respectively.

With this information we can now determine the states of the nodes of the grid as either *inside* or *outside*. To do this, only the edges pointing in x-direction are necessary. For each pair of y- and z-coordinates present in at least one grid cell we follow the edges in x-direction summing up the state difference counters (see Fig. 1). Each grid node for which the sum is greater zero is marked as *inside*. All others are *outside* nodes. Interpreting all marks greater than zero as *inside* is our trick to get rid of all self-intersecting parts of the input mesh (see Fig. 2). It is important that singular cases are handled properly. If the cell edge runs through edges or vertices of the input mesh, one has to make sure that only one intersection is counted. We do this by lumping together cuts that are closer than an ε. This simple approach is prone to numerical errors though. To make the process more robust, we not only use the x-direction as just described but all three principal edge directions in positive and negative direction and mark a node as *inside* if more than 3 out of the 6 tests vote for *inside*. This way, we can also handle defect meshes with holes robustly. A similar approach was used in [HBW03] for modeling occlusions. In our tests, the method was robust enough to handle the potentially ill conditioned normals produced by the marching cubes method.

Finally we apply the marching cubes templates [LC87] to each cell to create the triangles of the new mesh. These templates require a vertex on each cell edge for which the states of the adjacent grid nodes differ. We create these vertices uniquely for all the cells adjacent to the particular edge to make sure that the resulting mesh is connected. In order to handle multiple intersections per edge, we choose the positions of these vertices to be the average of the positions of all intersections of the edge Note that, by construction, the resulting mesh is manifold, closed and non-self-intersecting as requited.

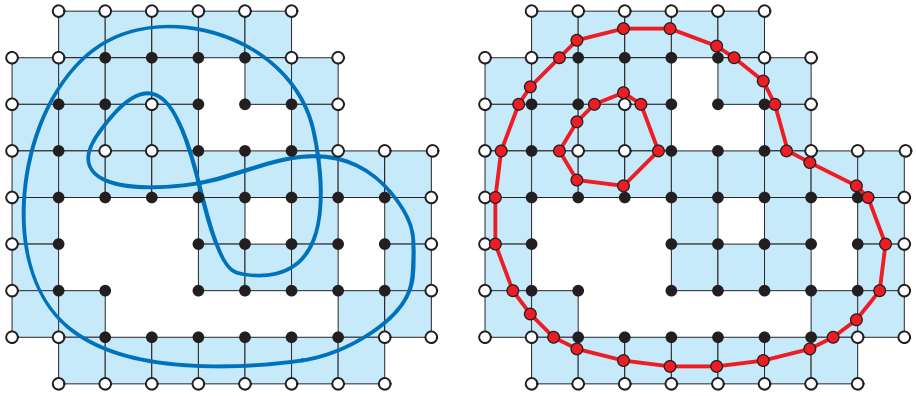

Figure 2: *Two dimensional case. Left: The input mesh and the sparse grid with nodes marked as inside or outside. Right: The new mesh created using the marching squares templates.*

2.1. Volume Conservation

As with Level Sets, our approach suffers from volume loss. A simple approach to control the volume exploits the fact that we are working with an explicit representation of the surface. The volume enclosed by the surface is

$$V = \frac{1}{6} \sum_{i=1}^{n_{\text{triangles}}} (\mathbf{p}_{t_1^i} \times \mathbf{p}_{t_2^i}) \cdot \mathbf{p}_{t_3^i}, \tag{1}$$

where t_1^i, t_2^i and t_3^i are the three indices of the vertices belonging to triangle i and $\mathbf{p}_i$ the positions of the vertices. The volume gain ΔV when moving all vertices by a distance d along their outward normal can be approximated as

$$\Delta V \approx Ad, \tag{2}$$

where

$$A = \frac{1}{2} \sum_{i=1}^{n_{\text{triangles}}} |\mathbf{p}_{t_1^i} \times \mathbf{p}_{t_2^i}|. \tag{3}$$

is the area of the surface. So one way to keep the volume at a desired value V_0 is to move all vertices by the distance

$$d = (V_0 - V)/A \tag{4}$$

along their outward normal vector. To make sure that the liquid does not get extended into solid objects, only the triangles and vertices not in contact with the boundary should be considered in Eq. (3) and Eq. (4) while the entire surface is used for the computation of the volume.

Our simple tracking method together with this straight forward way of preserving volume make possible fast and robust simulation of free surface liquids with very limited memory consumption. Fig. 10 shows a simulation sequence created with the basic algorithm described so far. We also used a simple way of preserving texture coordinates. Each vertex of the new mesh is created by an intersection of a grid edge with a triangle. We compute the texture coordinates of the new vertex as the barycentric sum of the texture coordinates of the vertices adjacent to that intersecting triangle. The barycentric weights are given by the location where the edge hits the triangle. If multiple triangles intersect an edge, texture advection is not well defined anymore. In that case we choose and arbitrary triangle.

3. Subgrid Feature Preservation

Using a grid for mesh creation has the effect that subgrid features disappear. This is true for the level set approach as well. For cases where this is problematic, we propose an extension of our basic method.

One of our applications is an unbounded Eulerian liquid simulation using a sparse dynamically changing simulation grid. In contrast to the easier case where the fluid is confined to a box, the unbounded liquid typically spreads on the floor and turns into a thin layer. The thickness of the layer decreases rather evenly so as soon as it goes below the grid spacing the entire layer disappears more or less at once.

To alleviate such problems, we now present a technique that can track arbitrarily thin structures on the uniform grid used so far without the need of local subdivision. To bound the complexity and the number of vertices created at each time step, we restrict the type of subgrid geometry a single cell can hold to one arbitrarily thin layer. In our tests, the ability to handle this type of detail resolves a majority of the problematic cases. The technique allows us to handle thin sheets of water (separated by at least one grid cell) as well as shallow puddles for instance (see Fig. 3).

3.1. The 2d case

Let us first look at the 2d case. The top row in Fig. 5 shows the standard marching squares templates modulo rotation. On each cell edge with adjacent nodes of different type a

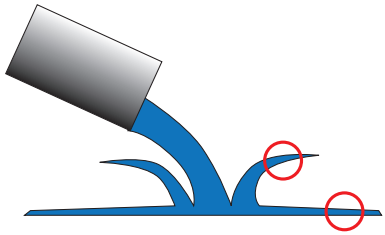

Figure 3: *Our method preserves thin layers such as sheets and shallow puddles.*

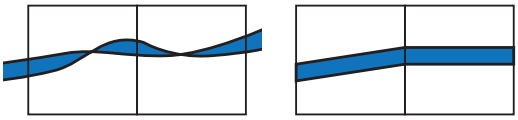

Figure 4: *Explicit advection of the two surfaces of thin features often causes self-intersections. Our method resolves these automatically.*

vertex is created (shown in green). As Fig. 1 shows, it is possible that edges are cut and still end up having adjacent nodes of the same type. This is detail that is lost in the standard algorithm. We preserve this detail as follows: For each edge with adjacent nodes of the same type for which we have registered at least two cuts, we create two additional vertices (shown in red in Fig. 5). As their positions we choose the minimum and maximum locations of all registered cuts on the edge.

An enlarged set of templates is necessary to account for the additional vertices (Fig. 5). In addition to the two states of the cell nodes, cell edges have two states as well – they can contain no or two red vertices. This enlarges the number of templates from 2^4 to 2^8. Not all of the 2^8 configurations are valid though because red vertices are potentially created only on cell edges with adjacent nodes of equal type.

There are certain cases where additional vertices are necessary. Let us have a look at template 1-b for instance. Here, a thin layer ends inside the cell so we need the yellow vertex for not losing the front. The case can only occur if the input 2d surface takes a turn inside the cell which is only possible if there is an input vertex inside the cell. We choose this vertex of the input surface directly for the construction of the new surface. There might be more than one vertex of the input surface in the cell. In order to control the complexity of the new surface, we don't want to use all of them. Instead, the user can specify an upper bound k for how many internal vertices should be used (we chose $k = 2$ in the samples). Fig. 7 shows the cases k = 1 (a), k = 2 (b) and k = 3 (c). The vertices are chosen such that the enclosed region has maximal area. We use internal vertices of the input mesh in other templates as well as image (d) shows. This is an effec-

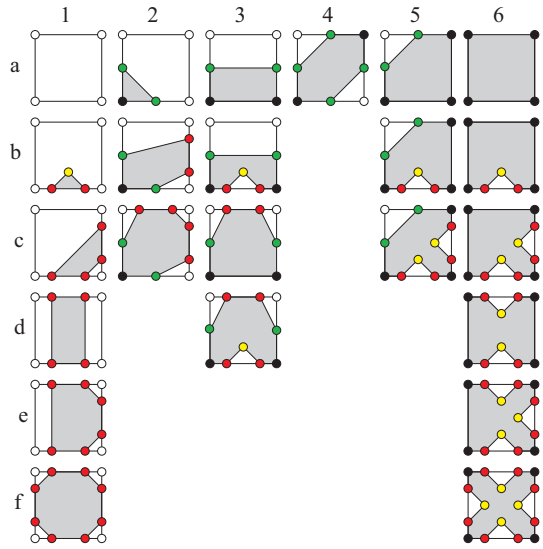

Figure 5: *Top row: Standard marching squares templates. Green vertices exist between* inside *(black) and* outside *(white) nodes. Below: Each edge connecting nodes of the same type can contain one interval of the opposite type bounded by two additional (red) vertices. Some templates require inner vertices (yellow) as well.*

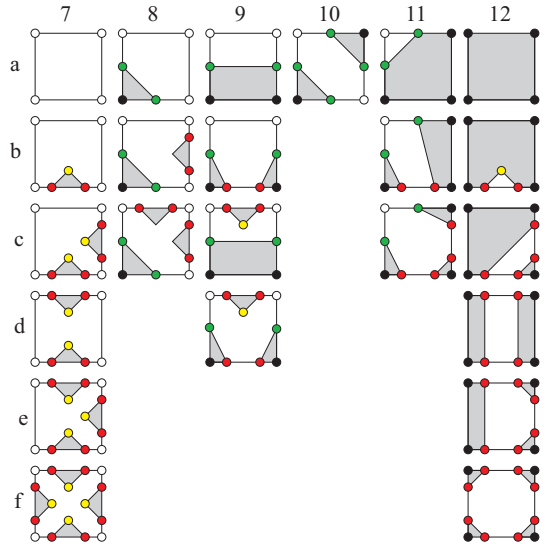

Figure 6: *For some configurations there exist dual templates. Cases 4a and 10a are the pair known from regular marching squares.*

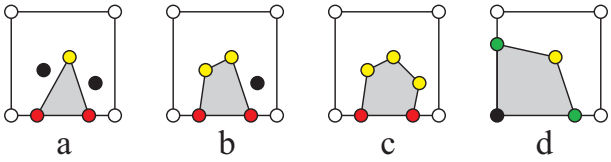

Figure 7: *Fronts and edges are preserved by considering vertices of the input mesh inside cells.*

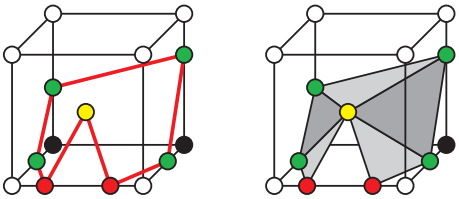

Figure 8: *Triangles of a grid cell are created by applying the 2d templates to all 6 faces and triangulating the resulting closed loops of segments.*

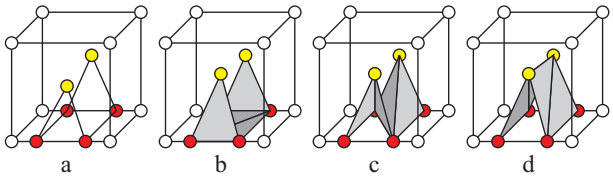

Figure 9: *Triangulation of a boundary (a) is not unique. The correct one is (d). Triangles on faces are not allowed (b) and (c) is an undesired configuration.*

tive way to preserve sharp edges and conserve volume (see Fig. 11 - Fig. 13).

Some of the templates have dual configurations which are shown in Fig. 6. While there is only one ambiguity in standard marching squares (templates 4-a and 10-a), a few more are present in the extended set. The ambiguity is solved in the standard approach by testing the value of the center of the cell. If it is inside the surface template 4-a is chosen, otherwise template 10-a. The same strategy could be used with the extended template set because the templates in Fig. 6 tend to cover the cell centers while their dual counterparts tend to leave the center open. We use columns $1,2,3,4$ and $11,12$ independent of the state of the center. This way we reduce the number of yellow vertices needed and even out the bias of inside / outside area in the set of templates of Fig. 5.

Our method does not recover all subgrid features. Sharp corners inside grid cells are lost for instance. It does, however, preserve arbitrarily thin layers using a bounded number of additional vertices. Keeping arbitrarily thin layers is not always desirable. We introduce an additional level of control for the user. If the distance of the two red vertices goes below a threshold, they are discarded. Also, in the case of liquid simulation, it is feasible to ignore thin layers of void altogether by only creating red vertices if both ends of an edge are marked as *outside*.

3.2. From 2d to 3d

In the 3d case we need a way to create triangles for each grid cell. We do this in three steps (see Fig. 8). First, we use the 2d templates on all 6 faces of the cell independently to create the green, red and yellow vertices plus line segments between them. In the 3d case, the yellow vertices are located where edges of the input mesh intersect the faces of the cells. In order for the triangulation to be compatible across grid cells, the configuration on a cell face must not depend on features of the cell not contained in that face. Therefore it is correct to handle the sides independently reducing the problem to 2d. Second, we connect the line segments across all faces of the cell to form closed segment chains.

Finally each of those chains is triangulated separately using ear clipping [Mei75]. This has to be done carefully. Fig. 9 shows three ways of triangulating the boundary (a). To avoid case (b) we never cut an ear that lies completely inside one face or a cut that leaves triangles in one side only. We do this by using a bit mask per vertex that stores one bit for each face the vertex belongs to (yellow vertices have one, non-yellow vertices have two bits set). If the bit-wise AND of the vertex masks of an ear is non zero, the cut is illegal. Avoiding case (c) is a bit trickier. When selecting the next ear, we choose the one for which a maximal number of vertices is below the ear's plane, thereby maximizing the enclosed volume.

3.3. Template Tables

The original marching cubes algorithm uses only $2^8 = 256$ templates. These can be stored in a table to prevent creating them during runtime. Unfortunately, this is not possible in our case. Our method uses on the order of $2^8 \cdot 2^{12}$ templates (yellow vertices not considered). Also, the triangulation depends on the actual positions of the vertices in space. One way to improve performance is to check whether there are any red or yellow vertices and to fall back to the standard marching cube template table if this is not the case. Parallelization is another effective way to speed up the process because triangulation can be done independently for each cell.

4. Results

So far we only have a sequential implementation of the algorithm. All the examples were run on a single core of a 2.4 GHz quad core Pentium.

The images shown in Fig. 10 are screen shots of our interactive environment. Here we used the basic surface tracking method without feature preservation but with geometric volume conservation. On average, there are about 40K triangles in the scene. The simulation runs at about 6 frames per second where surface tracking takes 70ms per time step on average.

Fig. 11 shows how features are preserved by the extended method. A plate is advected along a rotational velocity field. It's thickness is smaller than the grid spacing. Tracking the yellow face vertices makes sure that edges are preserved while storing red interval cuts prevents subdgrid features from disappearing. We also ran the standard notched cylinder benchmark. Typically, this test is run on a 100×100 grid. We used a resolution of only 16×16 cells to emphasize potential problems. Convex edges are preserved. In the concave region there is some volume gain however. This happens because in Fig. 7 we choose face vertices such that the area they enclose is maximized. The motivation for this is that errors in concave regions are not as obvious in fluid simulations as errors in convex fronts of thin sheets.

The sequence in Fig. 12 shows a fluid duck as it falls into tank built of four dynamic rigid bodies. The sheets during the first splash are tracked accurately. After the fluid pressure pushes the walls apart, the liquid spills out and forms shallow puddles. Using sparse grids for both surface tracking and Eulerian fluid simulation enables efficient handling of this unbounded scene. The grid resolution for both the fluid and the surface is such that there are $35 \times 35 \times 12$ cells inside the enclosure. Surface tracking takes 150 ms per time step at the beginning and about 700 ms at the end of the sequence while the number of surface cells increases from 14K to 50K. The time used by the Eulerian fluid simulator increases from 50 ms to 600 ms in the course of the simulation.

A similar scenario is shown in Fig. 13. Here a cube is thrown into a pool of water. Again, the walls tumble and the liquid spills out. In this scene, we have $40 \times 40 \times 24$ cells inside the pool. During the simulation, computation time per step increases from 100 ms to 800 ms for surface tracking and from 150 ms to 800 ms for fluid simulation. The number of surface cells starts at 15K and ends at 55K.

5. Conclusions and Future Work

We have presented a novel method for tracking the surface of liquids. It uses an explicit representation of the surface, namely a triangle mesh. This way, there is no need for storing and maintaining a scalar field in the neighborhood of the interface. Also, with the feature preserving method, important detail such as shallow puddles or thin sheets do not get lost between grid cells.

It took us only two days to code a working prototype of the simple core version. For games and other real-time applications we recommend to start with the simple approach and to see whether the feature-loss artifacts are acceptable.

Our way to preserve volume is global and allows droplets to disappear while larger bodies of water grow. As future work we plan to remove this limitation. One idea is to use flood fill on the triangles to identify connected submeshes and preserve the volumes of each submesh individually while merging and splitting have to be handled properly. Another challenge would be to fix the mesh only locally where needed and make the interface between fixed and non-fixed parts watertight as in [WTGT09] while working with the extended template set.

We hope that the improvements of our method in memory consumption, speed and the ability to handle unbounded scenes brings Eulerian fluid simulation within the reach of interactive 3d computer games. So far we only have a sequential version of the algorithm. The next step will be to parallelize our code using CUDA and render the results directly on the GPU.

References

[BB06] BROCHU T., BRIDSON R.: Fluid animation with explicit surface meshes. In *Symposium on Computer Animation 2006, poster session* (2006).

[BB09] BROCHU T., BRIDSON R.: Robust computational algorithms for dynamic interface tracking in three dimensions. *SIAM Journal on Scientific Computing* (2009).

[BGOS06] BARGTEIL A. W., GOKTEKIN T. G., O'BRIEN J. F., STRAIN J. A.: A semi-lagrangian contouring method for fluid simulation. *ACM Transactions on Graphics 25*, 1 (2006).

[CDML07] CHENG L. T., DZUBIELLA J., MCCAMMON J., LI B.: Application of the level-set method to the implicit solvation of nonpolar molecules. *Journal of Chemical Physics 127*, 8 (2007).

[CLT07] CRANE K., LLAMAS L., TARIQ S.: Real-time simulation and rendering of 3d fluids. *GPU Gems 3* (2007), 633–673.

[DC98] DESBRUN M., CANI M.: Active implicit surface for animation. In *In proceedings of Graphics Interface 1998* (1998).

[EFFM02] ENRIGHT D., FEDKIW R., FERZIGER J., MITCHELL I.: A hybrid particle level set method for improved interface capturing. In *J. Comput. Phys* (2002), pp. 83–116.

[ELF05] ENRIGHT D., LOSASSO F., FEDKIW R.: A fast and accurate semi-lagrangian particle level set method. In *Computers and Structures* (2005), pp. 479–490.

[EMF02] ENRIGHT D., MARSCHNER S., FEDKIW R.: Animation and rendering of complex water surfaces. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (2002), ACM Press, pp. 736–744.

[FF01] FOSTER N., FEDKIW R.: Practical animation of liquids. In *In proceedings of Siggraph 2001* (2001), pp. 23–30.

[GGL98] GLIMM J., GROVE J. W., LI X. L.: Three dimensional front tracking. *SIAM Journal on Scientific Computing 19* (1998), 703–727.

[GGLT99] GLIMM J., GROVE J. W., LI X. L., TAN D. C.: Robust computational algorithms for dynamic interface tracking in three dimensions. *SIAM Journal on Scientific Computing 21* (1999), 2240Ű2256.

[HBAD07] HOGEA C., BIROSA G., ABRAHAM F., DAVATZIKOS C.: A robust framework for soft tissue simulations with application to modeling brain tumor mass effect in 3d mr images. *Phys. Med. Biol. 52* (2007), 6893–6908.

[HBW03] HOUSTON B., BOND C., WIEBE M.: A unified approach for modeling complex occlusions in fluid simulations. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Sketches & Applications* (2003), ACM.

[KH07] KENNEY J., HWANG G.: Electrochemical machining with ultrashort voltage pulses: modelling of charging dynamics and feature profile evolution. *Nanotechnology 16*, 7 (2007), 309–313.

[KLL*07] KIM B., LIU Y., LLAMAS I., JIAO X., ROSSIGNAC J.: Simulation of bubbles in foam with the volume control method. *ACM Trans. Graph. 26*, 3 (2007), 98pp.

[LC87] LORENSEN W., CLINE H.: Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of ACM SIGGRAPH 87* (1987), pp. 163–169.

[LGF04] LOSASSO F., GIBOU F., FEDKIW R.: Simulating water and smoke with an octree data structure. In *Proceedings of SIGGRAPH 2004, ACM TOG 23* (2004), pp. 457–462.

[LZ09] LEUNG S., ZHAO H.: A grid based particle method for moving interface problems. *Journal of Computational Physics 228 (8)* (2009), 2993–3024.

[Mei75] MEISTERS G. H.: Polygons have ears,. *Amer. Math. Monthly 82* (1975), 648–651.

[MM07] MILED M. B. H., MILLER E. L.: A projection-based level-set approach to enhance conductivity anomaly reconstruction in electrical resistance tomography. *Inverse Problems 23*, 6 (2007), 2375–2400.

[OF03] OSHER S., FEDKIW R.: *Level Set Methods and Dynamic Implicit Surfaces*. Springer 2003, ISBN 0387954821, 2003.

[OS88] OSHER S., SETHIAN J. A.: Fronts propagating with curvature dependent speed: Algorithms based on hamilton–jacobi formulations. *J. Comput. Phys. 79* (1988), 12–49.

[Set99] SETHIAN J.: *Level Set Methods and Fast Marching Methods*. Addison-Weslay Publishing Company, ISBN 0521645573, 1999.

[Sta99] STAM J.: Stable fluids. In *in Proceedings of ACM Siggraph 99* (1999), pp. 121–128.

[THM*03] TESCHNER M., HEIDELBERGER B., MÜLLER M., POMERANERTS D., GROSS M.: Optimized spatial hashing for collision detection of deformable objects. In *in Proceedings of Vision, Modeling, Visualization VMV 2003* (2003), pp. 19–21.

[WTGT09] WOJTAN C., THÜREY N., GROSS M., TURK G.: Deforming meshes that split and merge. *ACM SIGGRAPH 2009 Papers 28 (3)* (August 2009).

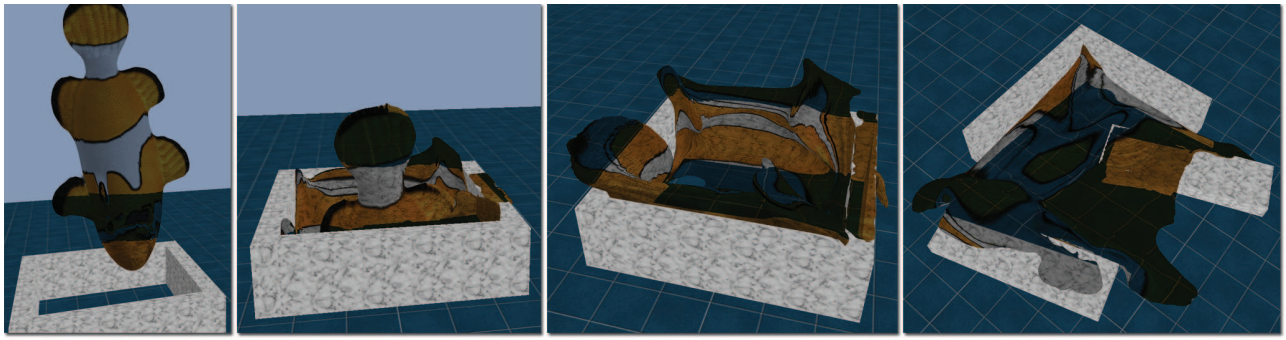

Figure 10: *Example of an interactive simulation using the basic method with texture coordinate preservation.*

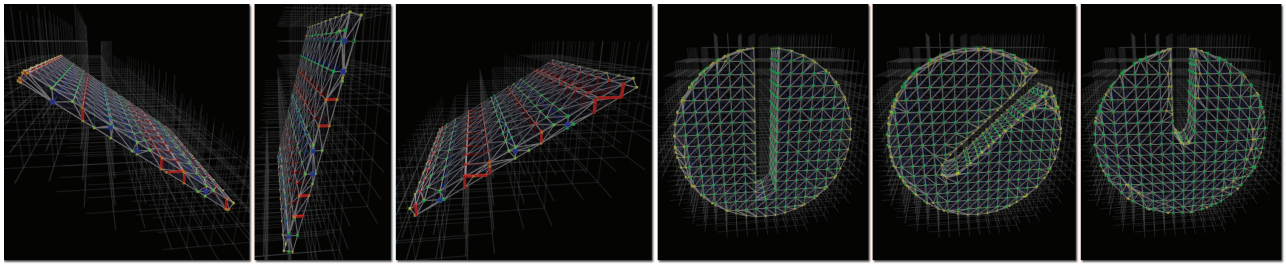

Figure 11: *Left: Rotation of a plate thinner than the grid resolution without loss of subgrid features. Right: Notched cylinder rotation on a low resolution grid. Edges are preserved. There is some volume loss/gain in convex/concave regions.*

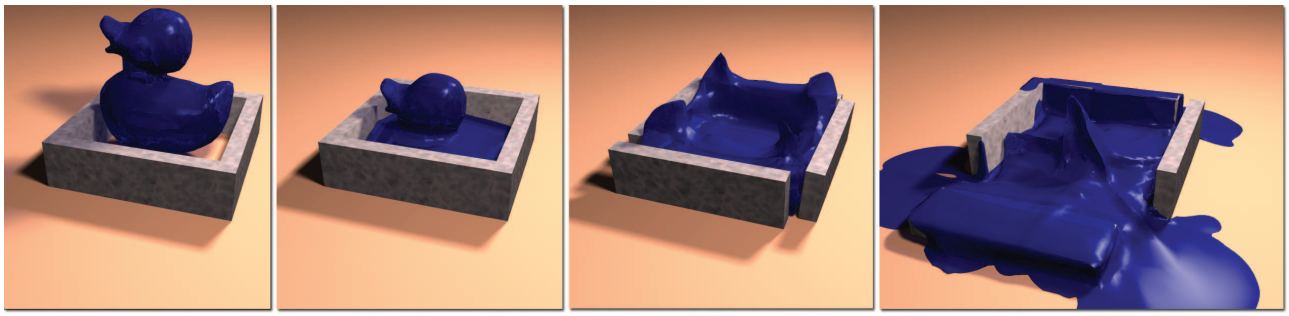

Figure 12: *Unbounded Eulerian fluid simulation showing thin sheets, sharp features and shallow puddles rendered off-line.*

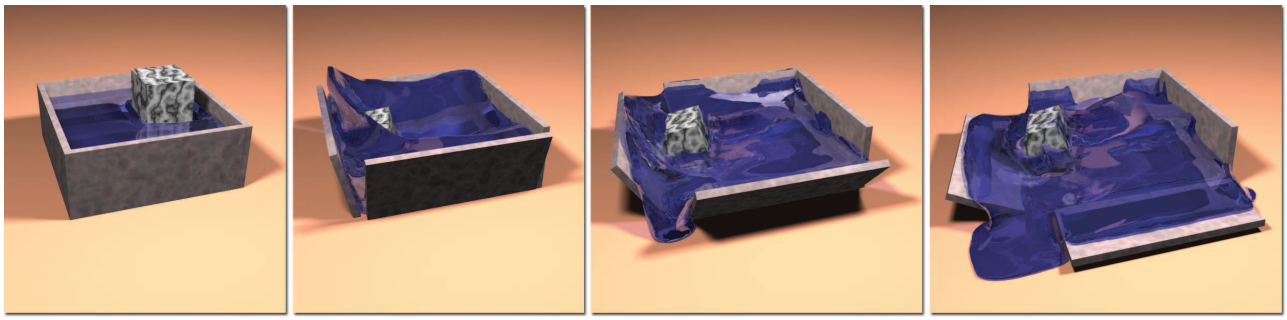

Figure 13: *Two way interaction with rigid bodies.*