Liquid Simulation with Mesh-based Surface Tracking

Siggraph 2011 Course Notes
Half Day Course

Course organizer:
Chris Wojtan
Institute of Science and Technology Austria

Lecturers:
Matthias Müller-Fischer
NVIDIA

Chris Wojtan
Institute of Science and Technology Austria

Tyson Brochu
University of British Columbia

Course Description

Animating detailed liquid surfaces has continually been a challenge for computer graphics researchers and visual effects artists. Over the past few years, a strong trend has emerged among researchers in this field towards mesh-based surface tracking in order to synthesize extremely detailed liquid surfaces as efficiently as possible. This course will provide attendees with a solid understanding of the steps necessary to create a fluid simulator with a mesh-based liquid surface.

The course will begin with an overview of several existing liquid surface tracking techniques, discussing the pros and cons of each method. We will then provide instructions and a simple demonstration on how to embed a triangle mesh into a finite-difference-based fluid simulator. Once this groundwork has been laid, the next section of the course will stress the importance of surface quality and review techniques for maintaining a high quality triangle mesh. Afterward, we will describe several methods for allowing the liquid surface to merge together or break apart. The final section of this course showcase the benefits and further applications of a mesh-based liquid surface, highlighting state-of-the-art methods for tracking colors and textures, maintaining liquid volume, preserving small surface features, and simulating realistic surface tension waves.

Level of Difficulty: Advanced.

Intended Audience

This course is intended for both researchers and developers in industry who want to implement and have a solid understanding of the state of the art in fluid simulation for computer animation.

Preprequisites

A familiarity with Eulerian fluid simulation techniques for computer animation. The necessary background material can be found in the book Fluid Simulation for Computer Graphics by Robert Bridson (available from A K Peters), or the SIGGRAPH 2007 course notes on Fluid Simulation by Robert Bridson and Matthias Mller-Fischer. In addition, a passing knowledge of basic triangle mesh algorithms like subdivision and edge collapses will be useful.

About the Lecturers

Chris Wojtan
Institute of Science and Technology Austria
wojtan@ist.ac.at
http://pub.ist.ac.at/group_wojtan/

Dr. Chris Wojtan is an assistant professor at the Institute of Science of Technology Austria (IST Austria), where he is establishing a computer graphics lab with a research focus on physically-based animation, geometric modelling, and numerical techniques. His computer graphics contributions include methods for animating detailed viscoplastic materials, several techniques for controlling physics simulations, and an algorithm for efficiently computing topological changes in deforming triangle meshes. His research into mesh-based fluid surface tracking has helped produce extremely detailed liquid surface animations, allowing arbitrarily thin features and detailed crown splashes. Prior to his work at IST Austria, Chris received his Ph.D. in Computer Science from the Georgia Institute of Technology in 2010, and he worked as a visiting scientist at Carnegie Mellon University and ETH Zürich.

Matthias Müller-Fischer
NVIDIA
matthias@mueller-fischer.com
http://www.matthiasmueller.info/

Matthias Müller-Fischer is a principal software engineer at nvidia and head of the PhysX research team. He received his PhD in atomistic simulation of polymers in 1999 from ETH Zürich. During his post-doc with the MIT Computer Graphics Group (1999-2001) he changed fields to macroscopic real-time simulations. Since then, his main research field has been the development of fast and robust physically based simulation methods for computer games. He has published key papers in computer graphics on real-time particle based water simulation and visualization as well as finite element and geometry based soft body, cloth and fracture simulation. In 2002, he co-founded the game middleware company NovodeX (acquired in 2004 by AGEIA Technologies, Inc.), where he was head of research and responsible for the extension of the physics simulation library PhysX by innovative new features. He has been with nvidia since the acquisition of AGEIA in early 2008.

Tyson Brochu
University of British Columbia
tbrochu@cs.ubc.ca
www.cs.ubc.ca/ tbrochu/

Tyson Brochu is a PhD candidate in the department of Computer Science at the University of British Columbia. His research focuses on purely mesh-based representations of surfaces undergoing extreme deformations and changes in topology, with a focus on tracking liquid surfaces. Additionally, he developed a novel approach to constructing fluid simulation elements, which captures fine surface details present in mesh-based surface tracking. His work has been published in scientific computing and computer graphics journals. Collaboration with VFX studios such as Weta Digital and Digital Domain informs his research and provides industry practitioners with access to cutting-edge applied research.

Course Overview

5 minutes: Introduction and welcome
Chris Wojtan

25 minutes: Liquid surface tracking review
Matthias Müller-Fischer
- Review of Eulerian surface tracking methods
- Review of point-based Lagrangian surface tracking methods
- Problems with previous methods that can be overcome with a Lagrangian surface mesh
- Benefits of a mesh-based Lagrangian representation (Examples with fixed topology)

30 minutes: Embedding a surface mesh into an Eulerian fluid simulation
Matthias Müller-Fischer
- Mesh-based surface tracking in a simulation using regular cubic grid cells
- Boundary conditions for solids and the free surface
- Live demonstration
- Problems with differing surface and simulation resolutions
- Adapting Eulerian grid geometry to better fit the surface mesh

15 minute break

15 minutes: Maintaining surface mesh quality
Chris Wojtan
- Why? Visual artifacts, memory limitations, and stable computation
- Quality measures
- Operations: Edge splits, edge flips, edge collapses, and null-space smoothing

50 minutes: Topology changes
Chris Wojtan
- Why do we need topology changes?
- Topological operations on the mesh itself
- Global grid-based re-meshing of the surface
- Local grid-based re-meshing of the surface (marching cubes & convex hulls)

50 minutes: Advantages of a mesh surface
Tyson Brochu
- Color and texture tracking
- Volume preservation (both global and local)
- Preserved surface details
- Thin features
- Surface tension (several methods)
- The future of mesh-based surface tracking

5 minutes: Conclusion
Chris Wojtan

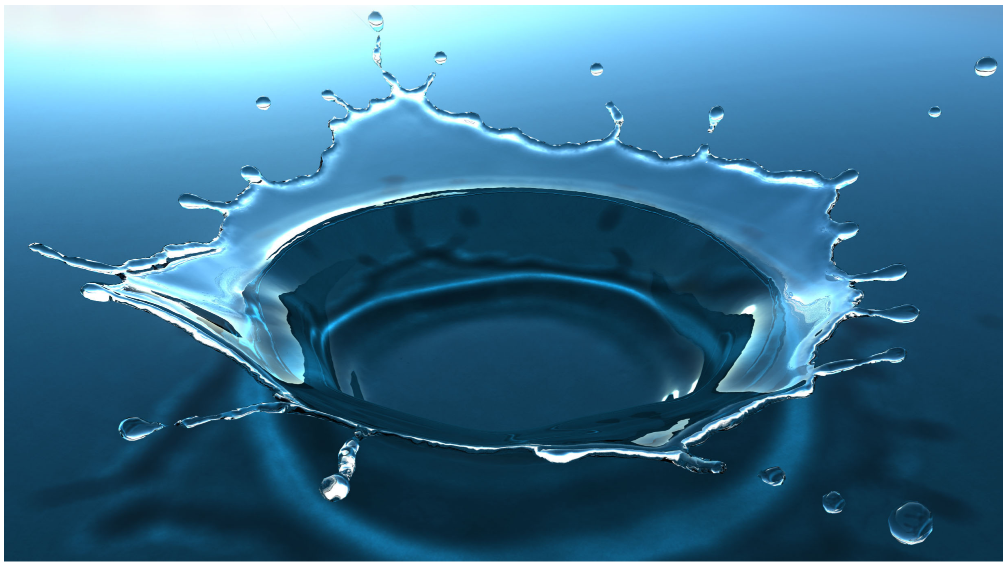


Figure 1: This course will provide all of the imlpementation details necessary to implement a fluid simulator capable of preserving thin liquid sheets and efficiently exhibiting detailed surface tension behavior.

1 Introduction

Animating detailed liquid surfaces is continually an important challenge for computer graphics researchers and visual effects artists. Until recently, purely Lagrangian techniques (like triangle meshes or particles) for simulating the liquid surface had been limited by several challenging problems. The difficulties associated with tangling surfaces were quite unattractive, and the complicated mesh surgery techniques necessary to split and merge surfaces were often plagued with robustness problems and promptly dismissed. On the other hand, Eulerian techniques (such as the level set method) were practically bulletproof and quite simple to implement. As a result, surface tracking techniques based on dynamic implicit surfaces became extremely popular and helped revolutionize the field of physics-based computer animation.

Over the past few years, however, several of the seemingly insurmountable problems associated with dynamic explicit surfaces have become more tractable — steady research in engineering, applied mathematics, computer vision, computational geometry, and computer graphics has slowly chipped away at the problem. Methods now exist for efficiently manipulating complex meshes while preventing self-collisions, and recent advances have allowed for the fast and robust computation of topological changes on deforming meshes. As a result of this progress, dynamic explicit meshes can now be integrated in a fluid simulation environment.

While explicit meshes may require more care for certain operations than standard implicit surface methods, dynamic meshes have the potential to become the new standard tool for

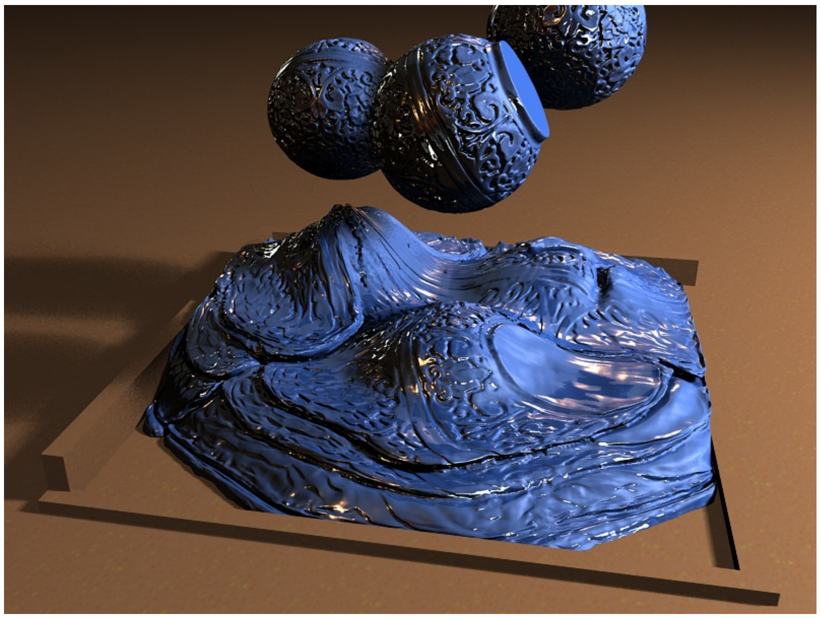


Figure 2: This course will explain how to couple a grid-basd fluid solver with a mesh-based surface in order to simulate materials with extremely detailed surface features.

creating beautiful fluid simulations. By default, mesh-based dynamic surfaces preserve an exquisite amount of visual detail, and they naturally provide a straightforward computational environment for simulating differential equations on a moving surface. In addition, although care is required to robustly handle topological changes, we now have the ability to *control* the topology of a dynamic surface, allowing for the persistence of thin liquid features and the efficient formation of tiny water droplets.

As a result of these compelling advantages of dynamic explicit surfaces, a trend has emerged among computer graphics researchers towards simulating liquids with mesh-based surface tracking. In the interest of keeping researchers and special effects developers up to date with the latest technology, this course aims to provide attendees with a solid understanding of the steps necessary to create their own fluid simulator with a mesh-based liquid surface.

The course will begin with an overview of several existing liquid surface tracking techniques, discussing the pros and cons of each method. We will then provide instructions and a simple demonstration on how to embed a triangle mesh into a finite-difference-based fluid simulator. Once this groundwork has been laid, the next section of the course will stress the importance of surface quality and review techniques for maintaining a high quality triangle mesh. Afterward, we will describe several methods for allowing the liquid surface to merge together or break apart. The final section of this course showcase the benefits and further applications of a mesh-based liquid surface, highlighting state-of-the-art methods for tracking colors and textures, maintaining liquid volume, preserving small surface features, and simulating realistic surface tension waves.

2 Surface tracking review

The main visual feature of a liquid is its surface. The sharp boundary distinguishes it from gases and creates all the fascinating visual effects that make liquids popular entities in computer graphics. Therefore, a large body of work on tracking liquid surfaces exists both in the computational fluid dynamics and in the computer graphics literature.

The problem of tracking a surface does not appear in fluid simulations only. It arises whenever a surface follows an underlying deformation or velocity field. In fluid simulations, this field is the solution of the Navier-Stokes equations but it can also be procedurally generated or created by a mesh deformation tool.

As in the case of fluid simulation approaches, there are Eulerian (grid based) and Lagrangian (particle based) surface tracking methods. In computer graphics, Eulerian tracking methods are often used with Eulerian simulations and Lagrangian tracking methods with Lagrangian simulations. Nonetheless, simulation and surface tracking are two independent problems. All the surface tracker needs from the simulator is the velocity vector at an arbitrary location in space. An Eulerian simulator provides this information by interpolating the discretized velocity field, while particle based simulators extrapolate the velocities of the particles. In turn, the tracker needs to provide a function that tells the simulator whether an arbitrary point in space is inside or outside the volume enclosed by the surface.

In this course we focus on surface tracking of liquids simulated with an Eulerian solver. However, as mentioned above, the methods we are going to present might as well be used in connection with particle based fluid simulations or mesh deformation tools. They could even be applied to solid simulations where the surface mesh is driven by a surrounding tetrahedral mesh as in [46, 71]. The mesh based surface tracking methods presented in this course belong to the class of Lagrangian methods because the vertices of the mesh are treated as particles that are advected with the fluid flow. The main difference to previous particle tracking methods is that mesh based methods maintain connectivity information during the simulation.

2.1 Previous work on Eulerian surface tracking

Grid based tracking methods operate on a scalar field that defines the surface implicitly. Typically, this field is discretized using a regular hexaderal grid. Instead of moving the surface explicitly, the tracker updates the field via Eulerian advection along the velocities provided by the simulator. The scalar field is used to answer the inside/outside query and by the simulation framework to render the surface.

We will discuss three popular grid based surface tracking approaches. They differ mainly in what they store as the scalar field and the advection method used.

- The Level Set method (LSM)
- The Volume Of Fluid (VOF) method
- Density based approaches.

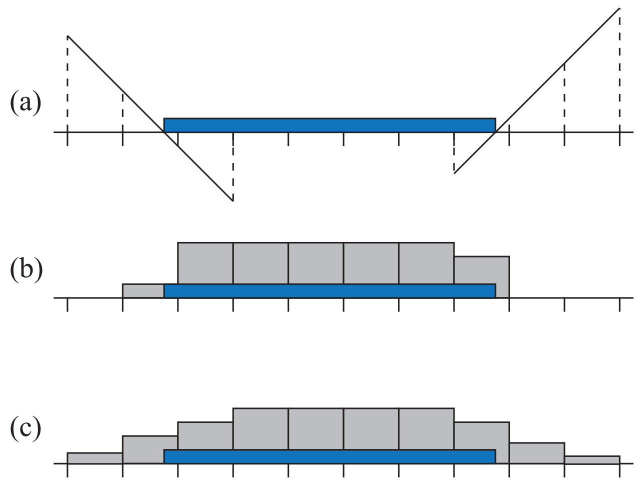


Figure 3: Representation of a 1D-liquid (blue bar) using (a) level sets [56], (b) Volume Of Fluid (VOF) [54] and (c) a mass conserving density field [44].

The Level Set method is the most popular Eulerian approach to surface tracking in computer graphics because it handles topological changes of the surface automatically, as all grid based tracking methods do, and is easy to implement, at least in its basic form. Unfortunately, the method suffers volume loss in underresolved, high-curvature regions due to numerical diffusion. In practice, this means that in the course of a simulation thin features dissolve and, eventually, the entire liquid disappears. One way to solve this problem is to introduce a quantity that is conserved by the operations that evolve the surface. In VOF methods, this quantity is the total volume of the liquid, while in density based approaches, it is the total mass. Another idea to reduce volume loss is to extend the original LSM itself using Lagrangian elements such as particles.

2.1.1 The Level Set method

The basic LSM, originally developed by [49], is described in detail in the two excellent and comprehensive books [56] and [48]. Here we only give a brief introduction. The LSM defines the surface via its signed distance field ϕ. Each grid cell stores the distance to the surface as a positive or negative scalar depending on whether the cell is outside or inside the liquid respectively. Hence, the surface itself is defined implicitly as the zero level-set of this scalar field (see Figure 3(a)).

The inside/outside query can be answered by interpolating the distance field and checking its sign. To advance the surface, ϕ is passively advected along the velocity field $\mathbf{u}$ of the fluid via

$$\phi_t = -\mathbf{u} \cdot \nabla\phi. \tag{1}$$

To save computation time, ϕ is typically tracked only in a narrow band near the surface. With the advection along the fluid flow, ϕ loses the property of being a distance field, i.e.

$|\nabla\phi| \neq 1$. Therefore, it has to be reinitialized periodically. In [56], the authors use Fast Marching to do this. First, the values of ϕ are explicitly re-computed near the surface. Then, the equation $|\nabla\phi| = 1$ is solved in a process that propagates known values of ϕ away from the surface (see [56] for the details of this elegant method).

Surface tracking itself is performed by solving Equation 1 either using finite differences or semi-Lagrangian advection [61]. The advantage of the latter method is its unconditional stability. It updates the value of ϕ at location $\mathbf{x}$ by sampling ϕ where the fluid at $\mathbf{x}$ was located in the previous time step. In the simplest version, ϕ is updated using linear backward path tracing

$$\phi(\mathbf{x}, t_{n+1}) = \phi(\mathbf{x} - \Delta t\mathbf{u}(\mathbf{x}, t_{n+1}), t_n) \tag{2}$$

and tri-linear interpolation to evaluate the right hand side. Interpolation from the surrounding grid values is necessary because the backtraced location is, in general, not aligned with the grid. In this basic form, semi-Lagrangian advection introduces substantial damping, meaning that details of the shape of the surface get lost quickly. There are several ways to improve the situation.

To increase accuracy of the backward tracing step to second order in time, linear backward path tracing can be replaced by a second-order Runge-Kutta scheme. Here, the current position is first linearly backtraced for half a time step only. The fluid velocity at this intermediate position is then used to perform the original linear backtracing step. The superiority of the Runge-Kutta scheme over linear backtracing is especially apparent in regions where the fluid rotates. A way to lift the backtracking step to second order accuracy in space is to use the MacCormack method. It performs a forward tracing step after backward tracing which yields an estimate of the numerical error introduced by using a linear path. This error estimate is then used to correct the original result [55].

Damping in the interpolation step can be reduced by replacing simple tri-linear interpolation with the Cubic Interpolated Propagation (CIP) method [35]. Here, the spatial derivatives of ϕ are advected along with ϕ and used to replace tri-linear interpolation by higher order polynomial interpolation. Yet another idea was introduced recently by Heo et al.[26]. Instead of sampling ϕ regularly in space, they store the discrete values of ϕ at the Gauss quadrature locations within each cell and use multi-dimensional Lagrange interpolation to evaluate ϕ at arbitrary locations within the cell.

All these methods are purely Eulerian. Volume and feature loss can also be reduced by introducing Lagrangian elements. Enright et al. proposed the particle level set method in [19, 20] and refined it in [17]. They place spherical particles randomly near the surface with a sign indicating which side of the surface they lie on. These particles are advected passively with the fluid flow along with the level set function ϕ. An advection error in ϕ can now be detected when the sign of a particle does not correspond to the sign of ϕ at its location. In this case, the level set values near the particle are corrected using the sign and radius of the escaped particle. To make sure that the particles remain evenly distributed near the surface, they are reseeded periodically after a fixed number of frames.

Semi-Lagrangian Contouring (SLC) [3] is another method that combines Lagrangian elements with Eulerian surface tracking. In this method, the surface is represented by an explicit triangle mesh. In contrast to mesh-based surface tracking, as described in this course, the explicit mesh used in SLC is not persistent but is recreated at every time step as the zero iso-surface of the signed distance field using the Marching Cubes method [38]. The main advantage of having an explicit mesh is that in the advection step distances to the surface – the triangle mesh – can be measured exactly so no interpolation is needed. In contrast to mesh based surface tracking, however, subgrid detail gets lost in the resampling step.

To counteract volume loss, Kim et al.[33] devised a method for conserving the volume of a liquid defined by a level set. They use a controller that generates an artificial pressure term based on the volume deviation. However, while the method reduces volume loss globally, it does not preserve small features. The problem of small feature loss in connection with the LSM was addressed by Losasso et al.[39]. They proposed to use an oct-tree data structure instead of a regular grid to discretize the signed distance function. This way, subgrid features can be kept alive by subdividing the surrounding cells to the necessary resolution. Instead of modifying the grid, Chentanez et al.[14] proposed to thicken features that are about to shrink to a size smaller than the grid spacing.

2.1.2 The Volume Of Fluid method

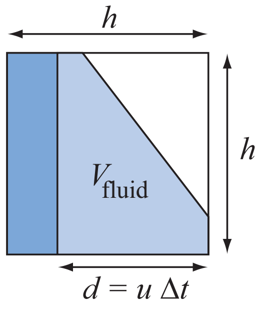


Figure 4: To compute the volume of fluid crossing a cell through the right face in one time step, the rectangle $d \times h$ is intersected with the fluid fraction, given by the surface plane (see [54]).

As we have seen in the previous section, one of the main disadvantages of the LSM is volume loss. Since the LSM uses the signed distance field ϕ to define the liquid surface and most often in a narrow band only, there is no direct way to derive the total volume or mass of the liquid from the scalar field of the tracker. This problem is depicted in 1D in Figure 3(a). The signed distance field represented by the dashed bars is not immediately aware of the amount of liquid between the two bands.

A fundamentally new idea to solve the volume loss problem is to replace the signed distance field by another scalar quantity. The Volume Of Fluid (VOF) method [54] uses the *volume fraction* (see Figure 3 (b)). Let us consider a 2D staggered grid with cell size h and the velocity components u and v in x- and y-direction respectively. The volume fraction $f_{i,j}$

is a discretized quantity defined per grid cell, not a continuous field. It is derived from the continuous characteristic function χ as

$$f_{i,j} = \frac{1}{h^2} \iint\limits_{\text{cell } i,j} \chi(x,y)\, dxdy, \tag{3}$$

where

$$\chi(x,y) = \begin{cases} 1 & \text{if } (x,y) \text{ is covered by liquid} \\ 0 & \text{otherwise} \end{cases}. \tag{4}$$

The total volume of the liquid is now given by

$$V_{\text{liquid}} = \sum_{i,j} h^2 f_{i,j}. \tag{5}$$

To conserve the total volume we simply have to make sure that whenever the advection operation subtracts a certain amount from $f_{i,j}$ it distributes the exact same amount somewhere else on the grid. In the continuous case, the advection equation that conserves a quantity f is given by

$$\frac{\partial f}{\partial t} = -\nabla \cdot (\mathbf{u} f) = -\mathbf{u} \cdot \nabla f - (\nabla \cdot \mathbf{u}) f. \tag{6}$$

Note that this equation is a generalization of Equation 1 with the extra term on the far right which vanishes for divergence free velocity fields. This more general form conserves the advected quantity even if the velocity field is not divergence free. Discretizing this advection equation using first order finite differences makes it more intuitive:

$$f_{i,j}^{n+1} = f_{i,j}^{n} + \frac{\Delta t}{h}(F_{i-\frac{1}{2},j} - F_{i+\frac{1}{2},j} + G_{i,j-\frac{1}{2}} - G_{i,j+\frac{1}{2}}), \tag{7}$$

where $F_{i\pm\frac{1}{2},j} = (fu)_{i\pm\frac{1}{2},j}$ denotes the flux of f across the right/left edge of cell (i,j), and $G_{i,j\pm\frac{1}{2}} = (fu)_{i,j\pm\frac{1}{2}}$ denotes the flux of f across the top/bottom edge of cell (i,j). This update scheme conserves the quantity $f_{i,j}$ because first, the difference of the fluxes into and out of the cell is accumulated in $f_{i,j}$ and and second, the fluxes are defined on the cell faces so the same quantity that leaves cell (i,j) on the right, i.e. $\frac{\Delta t}{h} F_{i+\frac{1}{2},j}$, enters cell $(i+1,j)$ on the left.

For the evaluation of the fluxes we need to know the shape of the surface first. From the definition of $f_{i,j}$ it is apparent that $0 \leq f_{i,j} \leq 1$. Also, a cell contains the surface if and only if $0 < f_{i,j} < 1$. The surface itself can be extracted using the least squares volume-of-fluid interface reconstruction algorithm (LVIRA) [54]. This algorithm computes a plane in each surface cell (i,j) as follows. A given normal vector $\mathbf{n}$ together with the volume fraction $f_{i,j}$ uniquely define a plane through cell (i,j) with normal $\mathbf{n}$ that divides it accord to $f_{i,j}$. This plane also cuts cells in the neighborhood of cell (i,j) but in general not according to their stored volume fractions. LIVRA picks the normal, and with it the plane, that minimizes this error in the least squares sense in the 3x3 neighborhood of cell (i,j).

Once we have computed the planes in all cells, we are ready to compute the fluxes needed to advect the surface. Let us consider $F_{i+\frac{1}{2},j}$. The distance the fluid travels through the right side of cell (i,j) in one time step is $d = u_{i+\frac{1}{2},j}\Delta t$, and the volume that leaves the cell is $V = dh$. Since cell (i,j) is only partially filled, we need to know the volume of fluid V_{fluid} inside V. To compute this, we intersect the rectangle of size $d \times h$ inside cell (i,j) aligned to the right with the fluid volume defined by the surface plane inside that cell (see Figure 4). The volume fraction that leaves the cell and must be subtracted from $f_{i,j}$ is, thus, V_{fluid}/h^2, so $\frac{\Delta t}{h}F_{i+\frac{1}{2},j} = V_{\text{fluid}}/h^2$ and

$$F_{i+\frac{1}{2},j} = \frac{V_{\text{fluid}}}{h\Delta t}. \tag{8}$$

The construction shown in Figure 4 is only valid if $d < h$ which means that the CFL condition must hold. This makes VOF only conditionally stable in contrast to the LSM. An additional drawback of VOF is that LVIRA is obviously not the only way to determine surface normals and that the VOF representation of the surface does not permit accurate curvature estimates, which are essential to surface tension computations. In contrast, normals and curvature are easily and uniquely extracted from a signed distance field as $\mathbf{n} = \frac{\nabla\phi}{|\nabla\phi|}$ and $\kappa = \nabla \cdot \frac{\nabla\phi}{|\nabla\phi|}$ (although this Laplacian operator $\nabla \cdot \nabla$ becomes inaccurate and unstable for small surface features only one or two grid cells wide).

2.1.3 Density based approaches

The problem of defining smooth normals and curvature present in the VOF method are removed by replacing the volume fraction field by a density field, as proposed by Mullen et al.[44] (see Figure 3 (c)). In their approach, the liquid surface is defined as the 0.5 - iso-surface of the density field ρ. This density is not to be confused with the fluid density. When tracking the surface of an incompressible fluid, the fluid density ρ_{fluid} is one everywhere while the surface density ρ can show a smooth transition from one to zero near the surface as in Figure 3. The advantage of this definition is that no special treatment is necessary near the surface as in the VOF method. The density field ρ is simply advected with the underlying fluid flow with the conserving advection equation

$$\frac{\partial\rho}{\partial t} = -\nabla \cdot (\rho\mathbf{u}). \tag{9}$$

Analogous to the VOF method, the value that is stored on the discrete grid is the averaged density of the cell

$$d_{i,j} = \frac{1}{h^2}\iint\limits_{\text{cell i,j}} \rho \, dx \, dy \tag{10}$$

The flux through the right side is given by $\frac{\Delta t}{h}F_{i+\frac{1}{2},j} = \rho_{i,j}u_{i+\frac{1}{2},j}\Delta t\, h/h^2$ or

$$F_{i+\frac{1}{2},j} = \rho_{i,j}u_{i+\frac{1}{2},j} \tag{11}$$

if $u_{i+\frac{1}{2},j}$ is positive and $\rho_{i+1,j}u_{i+\frac{1}{2},j}$ if $u_{i+\frac{1}{2},j}$ is negative.

This way, the total mass of the fluid is conserved exactly over time. As in the VOF method, and in contrast to the LSM, the density approach is only stable when the CFL condition holds.

An issue of the density approach not present in the VOF method is that the density profile near the liquid surface gets blurred over time due to numerical diffusion. This is problematic because cells with densities less then 0.5 are considered to be air cells. This way, the volume enclosed by the iso-surface can shrink even though the total mass is conserved. Mullen et al.[44] solve this problem by introducing a sharpening flow.

2.2 Previous work on Lagrangian surface tracking

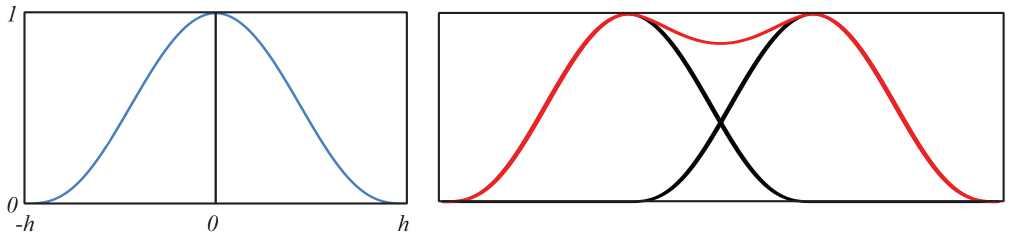


Figure 5: Left: A density kernel with finite support radius h. Right: Sum of two mutually shifted kernels.

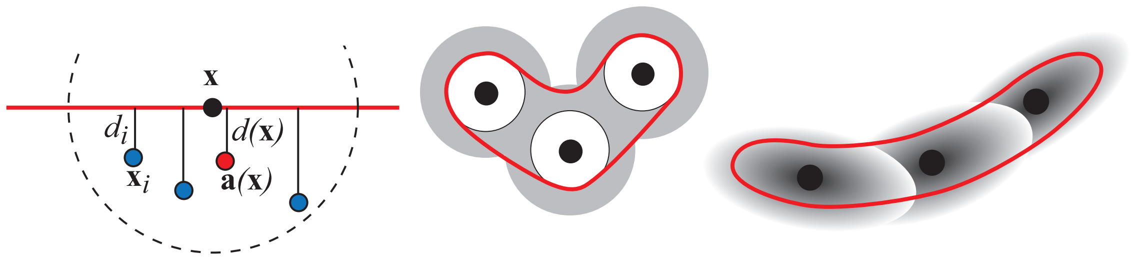


Figure 6: Methods to create smoother surfaces. Left: Adams et al.[1] average particle positions $\mathbf{x}_i$ and distance-to-surface estimates d_i to derive the red surface plane. Middle: Williams [66] smoothes a surface estimate while restricting the vertices to stay inside the gray area. Right: Yu et al.[72] use anisotropic kernels.

Lagrangian tracking methods typically represent the surface by a set of particles. The advection step is therefore simpler and without stability problems because the particles are moved passively with the fluid flow as in the Markers and Cells (MAC) method (see [41] for an excellent review). In particle based simulations, the tracking particles are often chosen to be a subset of the simulation particles so advection is not needed because it is performed by the simulator.

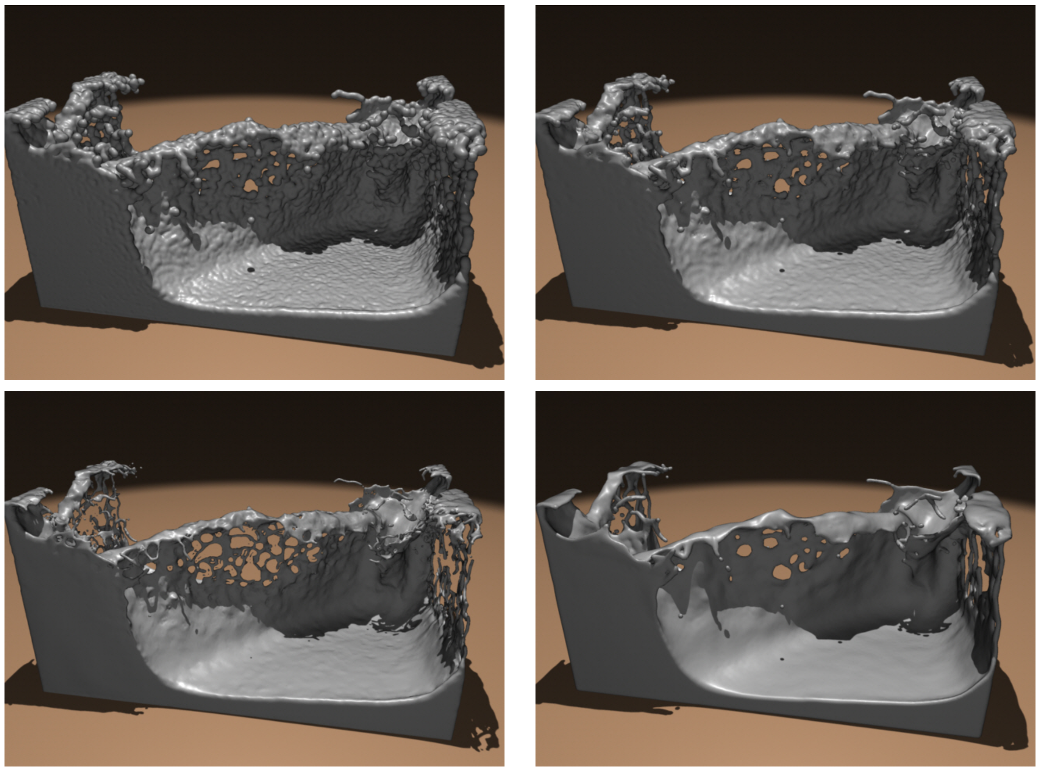


Figure 7: Comparison between different surface reconstruction approaches for particle based liquids. From top left to bottom right: the basic isotropic kernel method used by Müller et al.[47], the method of averaged kernel centers proposed by Zhu et al.[74], the method by Adams et al.[1] and the anisotropic kernel method of Yu et al.[72]. Images curtesy of Yu et al.[72].

However, constructing a closed surface from the point cloud is challenging. In Eulerian tracking methods, the surface is uniquely defined as an iso-surface and can be easily extracted using Marching Cubes. In contrast, *reconstructing* a surface from a point cloud is an entire research field not restricted to surface tracking and the shape of the surface is not uniquely defined by the points.

Blinn [7] introduced a popular method to create a surface from a set of points. This is at the heart of most Lagrangian liquid surface tracking methods in computer graphics. The main building block is a *kernel.* A kernel is a scalar function that describes the influence of a single particle i on its neighborhood. A popular example taken from [47] is

$$w_i(\mathbf{x}) = \frac{1}{h^6}\begin{cases}(h^2 - ||\mathbf{x} - \mathbf{x}_i||^2)^3 & ||\mathbf{x} - \mathbf{x}_i|| \le h \\ 0 & \text{otherwise,}\end{cases} \tag{12}$$

where $\mathbf{x}_i$ is the position of particle i. This function has finite support, meaning it is non-zero only within distance h from the particle (see Figure 5). The fact that a particle only influences a bounded neighborhood of fixed size reduces the complexity of kernel-based algorithms from $O(n^2)$ to $O(n)$. Two other nice properties of this specific kernel are that it is C^2-continuous at the boundary and that it depends on the square of the distance, not the distance, which avoids an expensive square root operation.

We can now define a density field for a set of particles using the kernels w_i as

$$\rho(\mathbf{x}) = \sum_i w_i(\mathbf{x}) \tag{13}$$

and the implicit surface

$$S = \{\mathbf{x} : \rho(\mathbf{x}) = 0.5\}. \tag{14}$$

This definition handles topological changes automatically as before. Müller et al. used it in [47] to extract the liquid surface from an SPH-based particle simulation and created a triangle mesh for rendering using Marching Cubes [38]. Unfortunately, this simple and straight forward density field yields blobby surfaces especially in regions where the fluid sampling is low as the top left image in Figure 7 shows.

Zhu et al.[74] addressed this problem. For a given position $\mathbf{x}$ in space, they first compute a normalized weighted average of the nearby particle positions as

$$\mathbf{a}(\mathbf{x}) = \sum_i w_i(\mathbf{x})\mathbf{x}_i / \sum_i w_i(\mathbf{x}) \tag{15}$$

and then evaluate the kernel centered at this position. This modification results in far smoother surfaces (see top right image in Figure 7).

Adams et al.[1] further improved the method of Zhu et al.. They maintain an additional scalar attribute d_i per particle storing an approximate particle-to-surface distance. With this additional information, they define the liquid surface as the zero-level set of

$$\phi(\mathbf{x}) = d(\mathbf{x}) - ||\mathbf{a}(\mathbf{x}) - \mathbf{x}|| \tag{16}$$

$$d(\mathbf{x}) = \sum_i w_i(\mathbf{x}) d_i / \sum_i w_i(\mathbf{x}) \tag{17}$$

(see Figure 6 left). As the bottom left image in Figure 7 shows, the resulting surface is further improved but at the price of higher computation complexity because updating the distance estimates d_i involves a propagation process that has to be performed at each time step.

Williams [66] first creates an explicit triangle mesh similar to [47] using Marching Tiles which is an extension of Marching Cubes. He then iterates through the vertices multiple times and applies a Laplacian smoothing operator while making sure that each vertex stays within $r_{\min}$ and $r_{\max}$ to the closest particle (see Figure 6 middle). The optimization process slows down surface extraction but creates perfectly flat surfaces for a liquid at rest.

Recently, Yu et al.[72] presented another method that yields high quality results. The basic idea is to use anisotropic kernels that have the following form:

$$w_i(\mathbf{x}, G_i) = \frac{1}{h^6} \begin{cases} (h^2 - G_i\, ||\mathbf{x} - \mathbf{x}_i||^2)^3 & ||G_i\,(\mathbf{x} - \mathbf{x}_i)|| \le h \\ 0 & \text{otherwise,} \end{cases} \tag{18}$$

where G_i is a symmetric 3×3 matrix. The application of G_i stretches the kernels along the Eigenvectors G_i by the inverse of the respective eigenvalues (see Figure 6 right). G_i is determined by applying Principal Component Analysis to the nearby particle positions. The resulting elliptic kernels reduce the bumpiness of the resulting surface, as the bottom right image in Figure 7 shows.

In this course we are going to study mesh based surface tracking methods. Here, the surface is represented as an explicit, closed triangle mesh. These methods are Lagrangian because the vertices of the mesh are treated as particles, passively advected with the fluid. The main difference to the methods above is the reconstruction step. This step is trivial in mesh based tracking because connectivity information is available as a triangle mesh. The mesh can be used directly for rendering and to answer the inside/outside-query. The difficult part is the maintenance of the connectivity structure, which is the primary subject of this course. As we will see, handling topological changes, e.g. self intersections and keeping well shaped triangles are the most challenging problems.

2.3 Conclusion

In this chapter we have reviewed Eulerian and Lagrangian methods to track the surface of a liquid. Eulerian methods define the surface implicitly as the iso-surface of a scalar function. The Lagrangian methods that were used in computer graphics before the introduction of mesh based tracking used the same approach in the reconstruction step. So the core difference between previous tracking methods and mesh based tracking is that the former represent the closed surface *implicitly* as an iso-surface, while mesh based tracking uses an *explicit* triangle mesh. The advantage of implicit surface representations are

- Topology changes are handled automatically. No special operations are needed when the liquid merges with itself.
- The quality of the representation stays the same throughout the simulation. Even though volume or small scale features might get lost, the underlying grid does not change its shape.
- For these reasons implicit methods are relatively simple to implement, at least in their basic formulation.
- Implicit methods also have a parallel structure because they operate on regular grids. This makes them good candidates for the implementation on a GPU.

What plagues implicit surface representations is feature loss. Working with a grid restricts the smallest features to have the size of the underlying grid sampling. It was the feature loss problem that originally initiated research in explicit surface tracking. This relatively new approach has several advantages that make it an attractive alternative to implicit surface tracking:

- Detail is perfectly preserved. It is not feasible to keep features alive throughout the simulation but if they are removed it is done deliberately by the algorithm.
- The same is true for topological changes. Handling topological changes explicitly is challenging but has the advantage that merges and splits can be performed in a controlled way.
- Mesh based methods simplify tracking of colors and texture coordinates.
- Explicit surface meshes allow the computation of the enclosed volume exactly, which is the basic building block of direct volume conservation methods.
- Triangle meshes allow an easy way to compute PDE's on the surface. This feature was used by Thuerey et al.[64] to add simulated small scale detail on liquid surfaces.

Mesh based surface tracking also poses some challenging problems.

- One of the main challenges is to catch and resolve all possible cases of mesh self intersections and splits.
- The solver has to be modified to make sure it is aware of the detail preserved in the mesh.
- The quality of the surface mesh degrades over time so the triangulation has to be constantly improved.
- Most operations on the mesh are geometric in nature. Special care has to be taken w.r.t. numerical robustness in degenerate and singular cases.

In this class we will address these topics and propose various solutions. We will present the state of the art in explicit surface tracking and show the fascinating effects and level of detail it can produce; a level of detail not seen in computer graphics' fluid simulations before.

3 Embedding a surface mesh into an Eulerian fluid simulation

This section lays the groundwork for a liquid simulation with mesh-based surface tracking. We will begin with a standard fluid simulation framework used frequently throughout computer animation [61, 21, 22, 8], and then we will explain how to embed a triangle mesh inside of this fluid simulation. The velocity of the fluid simulation is used to advect the vertices of the triangle mesh, and the way that the mesh partitions space into "inside" and "outside" regions will be used to define the domain of the fluid simulation. The result is a mesh that splashes and sloshes around under the influence of fluid forces.

3.1 Physical model

We start by building upon a common technique for simulating fluid dynamics in computer graphics: We aim to simulate the partial differential equations which govern subsonic fluid flow, known as the incompressible Navier-Stokes equations:

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{g} \tag{19}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{20}$$

where $\mathbf{u}$ is the velocity of the fluid, t is time, ρ is the density of the fluid, p is the fluid's pressure, ν is the fluid's viscosity, and $\mathbf{g}$ is the acceleration due to gravity. We store these variables on a regular hexahedral grid, with the pressure values stored at the centers of each grid cell and the velocity components stored on the faces of each grid cell. The details of storing these variables are discussed further in [8].

We can think of the Equation 19 as a collection of competing accelerations that push around the fluid. This equation governs the momentum of the fluid, and we will refer to it as the *momentum equation*. Equation 20 is a constraint on the momentum equation that forces the fluid's velocity field to have a divergence of zero ("divergence-free"). When the fluid's velocity has no divergence, it is considered *incompressible*, and there are no sources or sinks of momentum in this system. Additionally, as long as the density of the fluid is constant, this divergence-free constraint implies that the fluid must conserve mass and volume.

To simulate the fluid, we can break up the momentum equation into a series of terms using a technique called *operator-splitting*. This effectively means that each term on the right hand side of Equation 19 can be applied independently, one after the other. To account for the gravity term, we simply add a constant downward acceleration to our velocity field. To account for the viscosity term, we solve the diffusion equation $\frac{\partial \mathbf{u}}{\partial t} = \nu \Delta \mathbf{u}$. The advection term $-(\mathbf{u} \cdot \nabla)\mathbf{u}$, has been studied extensively in the applied mathematics and computer graphics literature, and we can use any of several techniques ([61, 21, 34, 43]). We used a higher-order variant on semi-Lagrangian advection known as unconditionally stable MacCormack advection [55] for most of our simulations.

Because the pressure field p is unknown at the beginning of each time step, we can use it as a Lagrange multiplier to satisfy the the divergence-free constraint (Equation 20). We solve the remaining system of equations using the Conjugate Gradient method, giving us the final velocity field $\mathbf{u}$. See [8] for details.

3.2 Embedded surface mesh

In order to simulate a liquid, we also need to simulate the motion of the the visible surface at the fluid boundary. This is where our mesh-based surface tracking comes in — we choose to use an explicit triangle mesh for this fluid surface. The mesh is simply a collection of vertices connected by triangles. This surface mesh partitions space into two regions: a portion of space that is "inside" the simulated liquid, and a vast region of space that is "outside" of the liquid. Because of this clean partition, we use this surface mesh to mark the simulation domain in our fluid solver. In other words, the surface mesh tells the fluid simulation which cells should be simulated as fluid and which should not, depending on whether the cells lie inside or outside of the mesh surface. As such, we insist that this surface is manifold and closed, in order to have a clear definition of what regions are "inside" of the simulated fluid, and which regions are "outside."

3.2.1 Updating the surface mesh

Once we have solved for the velocity field $\mathbf{u}$ at the end of each time step, use it to move the fluid surface. Each surface vertex is implicitly associated with a velocity, because it has a unique position $\mathbf{x}_i$ and the velocity field is a function of space, $\mathbf{u}(\mathbf{x})$. In practice, $\mathbf{u}$ is only defined at discrete regular intervals in our fluid grid, but we can use simple interpolation techniques to fit a smoothly-varying function to this data. In practice, we use tri-linear interpolation to find a velocity for a given position in space.

At this point, we assume that no topological changes occur during the motion of the surface, so we do not need to update any triangle connectivity; we simply need to solve the ordinary differential equation

$$\frac{d\mathbf{x}_i}{dt} = \mathbf{u}_i \tag{21}$$

for each surface vertex i with position $\mathbf{x}_i$ and velocity $\mathbf{u}_i$. We can rearrange this equation to get the integral

$$\mathbf{x}_i^{\text{new}} = \mathbf{x}_i^{\text{old}} + \int \mathbf{u}_i dt \tag{22}$$

The simplest technique for performing this numerical integration for each surface vertex is to use the forward Euler method:

$$\mathbf{x}_i^{\text{new}} = \mathbf{x}_i^{\text{old}} + \mathbf{u}_i \Delta t \tag{23}$$

The forward Euler method works well enough for most purposes, however we choose to use a fourth order Runge Kutta method (RK4) [52] because it performed best in analytical tests

(Figure 8, for example) and it is relatively inexpensive to compute. RK4 can be written as:

$$\mathbf{x}_i^{\text{new}} = \mathbf{x}_i^{\text{old}} + \frac{1}{6}(\mathbf{k}_1 + \mathbf{2}k_2 + \mathbf{2}k_3 + \mathbf{k}_4) \tag{24}$$

with:

$$\begin{aligned}
\mathbf{k}_1 &= \mathbf{u}(\mathbf{x}_i^{\text{old}}) \\
\mathbf{k}_2 &= \mathbf{u}(\mathbf{x}_i^{\text{old}} + \frac{1}{2}\Delta t\mathbf{k}_1) \\
\mathbf{k}_3 &= \mathbf{u}(\mathbf{x}_i^{\text{old}} + \frac{1}{2}\Delta t\mathbf{k}_2) \\
\mathbf{k}_4 &= \mathbf{u}(\mathbf{x}_i^{\text{old}} + \Delta t\mathbf{k}_3)
\end{aligned}$$

where $u(\mathbf{x})$ is the velocity field $\mathbf{u}$ evaluated at the position $\mathbf{x}$.

Because we make extensive use of velocity interpolation when integrating the surface vertices, we need to be able to access the velocity field $\mathbf{u}$ in many regions just outside of the boundary of the fluid simulation. To make sure that our velocity field $\mathbf{u}$ adequately samples space, we extrapolate $\mathbf{u}$ outward using a fast marching method [8].

Lastly, when we update the positions of the surface vertices, some vertices may lie within solid obstacles. One way to address this problem is to resolve the collision by projecting the vertices onto the surface of the obstacle. This works well in most cases, though it can cause problems when an entire thin sheet of liquid lies inside an obstacle. In such cases, simply projecting vertices onto the surface of the obstacle tends to create infinitely thin sheets of liquid. These degenerate cases can cause problems later (during volume or convex hull calculations, for example). In practice, we used free-slip boundary conditions at these solid obstacles and ignored any resulting collisions. While this approach worked surprisingly well for us, a more principled approach to collision handling may work better.

3.2.2 Updating the finite difference grid

As mentioned in the previous section, the fluid simulation uses the surface mesh to determine which cells will be active fluid cells, and which cells will be inactive "air" cells. This is essentially the same problem as *voxelizing* a surface mesh, or returning a set of grid cells that overlap the volume contained within a surface mesh. However, we can take this method a step further and actually compute a signed distance function in order to give ourselves some more information about the surface geometry. The scalar values of this signed distance function (negative values inside, and positive values outside) are collocated with the cell-centered pressure values in the fluid grid.

We employ a voxelization-style method [45] to compute the signed distance function. We first voxelize the triangle mesh onto a grid by computing intersections with a rays in the x, y, and z directions. For each ray, we keep track of its inside/outside status with a counter. At the start of the ray (which is guaranteed to be outside of the mesh), the counter has a value of zero. For each intersection with the triangle mesh, we increment the counter

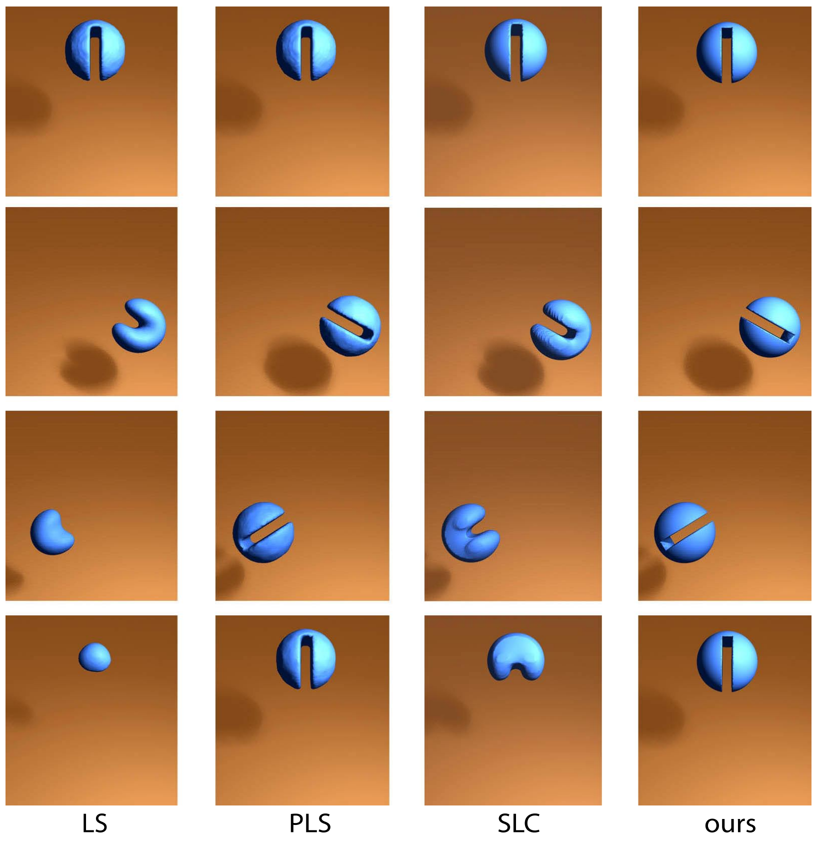


Figure 8: How does our mesh-based surface tracking scheme compare with other common surface trackers? This figure compares our mesh-based surface (far right column) with three other schemes (from left to right: level set method, particle level set method, and semi-Lagrangian contouring). The top row shows the initial conditions of the Zalesak sphere test. As time goes on (subsequent lower rows), the notched sphere rotates. The analytical solution to this problem is that the final condition after one rotation should be idential to its initial shape and position. The mesh-based surface tracker described in this section behaves significantly better than other methods.

if it intersects a triangle whose normal is facing the opposite direction of the ray, and we decrement it if it intersects a triangle with a normal facing the same direction of the ray. This way, all regions outside of the mesh will have a counter value of zero, regions inside the mesh will have a value of one, regions where multiple surfaces overlap each other will have a value greater than one, and regions that are inside-out will be negative. We store this counter value at each grid point to assign an inside/outside status, and then we compute the distance from the point to the nearest surface triangle in order to complete the signed distance calculation. Because our algorithms only need an accurate signed distance in the thin band surrounding the surface, we save time by only performing this distance calculation for grid cells that overlap surface geometry.

At this point, we can use this signed distance function to determine which fluid cells are inside of the surface (treating all "inside" cells as active fluid cells in the next time step).

However, some regions of the surface may have been too thin to be adequately sampled by the fluid grid, so these thin regions would not be surrounded by any active fluid cells with this method.

We use an explicit geometric technique to ensure that all thin features will be represented by negative values on the signed distance function grid. First, we find the set of all fluid cells that intersect or completely envelop any surface triangles — this can be done efficiently by looking at triangle vertex locations and seeing which grid cells they lie within. We then take the union of that set of cells and the original set of cells associated with negative values in the signed distance function. This final set of cells will adequately sample all thin features.

3.3 Free-surface boundary conditions

Like any Eulerian grid-based fluid simulation, we need to handle boundary conditions for the free surface. This can be done with several different techniques, and we will discuss two specific methods here.

3.3.1 Constant cell weights

The easiest way to implement the boundary conditions at the liquid surface is to simply assume that every fluid cell represents a unit of mass $\rho\Delta x^3$, where Δx is the width of a fluid cell. Unfortunately, the embedded surface mesh will not necessarily occupy the exact same volume as the simulation elements in general, so the simulation will overestimate the total mass in the system. This is noticeable in thin regions of liquid, when a surface sheet splashes into a larger body of water. The artificially large mass injects additional momentum into the system and creates a larger-than-expected splash. Although this behavior may actually be desirable in a special effects environment and more pleasing to watch, it is nevertheless inaccurate.

Errors in this constant-cell-weight strategy are clearly visible when thin surface regions stretch out into even thinner surfaces. Here, the volume of these surface features is preserved, so the volume per unit length drops as the length increases. However, the mass per unit length stays the same with this strategy, so a large amount of mass and momentum is added into the system whenever a small region thins out.

The constant cell weight strategy also causes problems when the liquid settles down and creates a flat surface. When one fluid cell has small amount of liquid, the mass gets rounded up to $\rho\Delta x^3$. This overestimated mass causes extra pressure at random spots along the settling surface and prevent a flat surface from forming.

Although the constant cell weight method is less accurate, it is quite simple to implement. We used this method in the simulations by Wojtan et al. [69, 70] and Thürey et al. [64].

3.3.2 Ghost fluid method

Alternatively, we can use a more accurate fluid discretization, called the *ghost fluid method* [18, 5]. This is a second-order approximation that essentially uses a linear extrapolation to as-

```
Compute inside/outside classification of mesh
Assign new fluid cells
Calculate fluid velocity
Advect surface vertex positions
If mesh collides with solid boundary
  Update vertex positions
```

Figure 9: Pseudo-code for one timestep of Eulerian fluid simulation

sign pressure values outside of the free surface. Because the fluid pressure should drop to zero exactly at the surface, an extrapolation will give negative pressures outside of the fluid surface. Combining the ghost fluid method with a surface mesh has yielded high quality visual results in the simulations of [11].

3.4 Algorithm overview

Figure 9 shows pseudo-code for a single time step of an Eulerian fluid simulation with a mesh-based embedded surface. The simulation first computes an inside/outside test for each fluid cell. Next, each simulation cell that is located inside of the mesh is considered a "fluid" cell, while the cells outside of the mesh are ignored. Next, we solve one time step of the incompressible Navier-Stokes equations to get a new fluid velocity for each fluid cell. Finally, we advect the surface vertices through the Eulerian fluid cell velocities using standard techniques (e.g. 4th order Runge-Kutta), and then adjust these vertex positions in the case of any collisions with solid obstacles.

3.5 Problems with differing surface and simulation resolutions

When a very high resolution surface is advected through a comparatively low resolution physics simulation, interesting visual artifacts can occur. In particular, the high resolution features captured by the surface (a high resolution mesh or high resolution level set) cannot be communicated to the physics simulation. As a result, the physics simulation does nothing to correct unphysical high resolution features, such as surface kinks and small voids. Some examples of this type of phenomena can be seen in the right column of Figure 10. One way to solve this problem is to adapt the Eulerian simulation grid to match up with the surface geometry. Later in this course, we will describe a very effective method for achieving this matching between the surface and grid resolutions using Voronoi regions for the Eulerian fluid elements [11].

3.6 Limitations

In this section, we introduced a straightforward method for coupling a grid-based Eulerian fluid simulation to a mesh-based Lagrangian surface. As such, it inherits the benefits of both

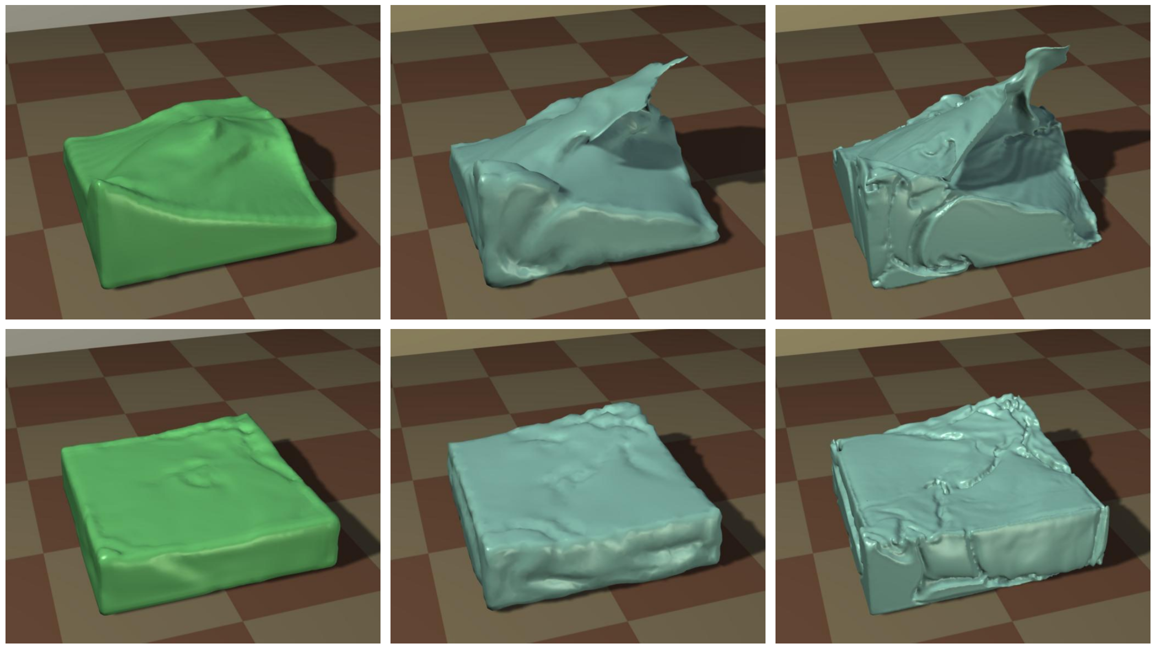


Figure 10: Different examples of surface tracking, from left to right: Level set, low-resolution triangle mesh, and high-resolution triangle mesh. The top row shows how mesh-based surface tracking can better capture thin surface structures, and higher-resolution meshes capture extremely detailed surface features. The bottom row shows how surface artifacts can accumulate when the surface is much higher resolution than the physics simulation (far right).

the Eulerian and Lagrangian simulation techniques. By using a semi-Lagrangian advection scheme, we can guarantee that the fluid simulation will be unconditionally stable for large time steps. The Lagrangian nature of the surface mesh also avoids the accumulation of re-sampling errors that are common in Eulerian surface tracking schemes.

Although it is quite basic, the surface tracking method described in this section is already quite good. Unlike many Eulerian surface tracking schemes, this method has no smoothing errors caused by continually re-sampling the surface from a grid. The only numerical errors in this scheme are due to the fluid simulation or the time integration. Figure 8 shows how this simple mesh-based surface tracker compares to other common surface-tracking schemes in the Zalesak sphere test.

One major problem with this approach as presented so far is that it does nothing to address topology changes in the surface geometry. Lively fluid animations can have drastic changes in the connectivity of surface geometry, so it is important that we address this issue. We will discuss this problem and present some solutions in more detail later in the course. Another problem with this approach is that surface triangles can become very distorted, causing many computational problems. We will discuss some ways to maintain a high triangle mesh in the next section of these course notes.

4 Maintaining surface mesh quality

As we update the position of surface vertices during a fluid simulation, the surface mesh will deform. If we do not change the connectivity of the triangle mesh, then triangles in the surface mesh can become severely distorted — some triangles may become much too large, while other triangles may have near-zero area. To address this problem, we perform various "mesh-maintenance" operations at the end of each advection step. This section of the course will elaborate on this idea and provide some tools to improve the quality of the surface mesh.

4.1 Why do we care about mesh quality?

The importance of mesh quality has been addressed thoroughly by others [57, 31], so we will only lightly touch on the concept here. Triangles with poor quality metrics can adversely affect a simulation in many ways:

- **Visual artifacts:** If we allow triangles to become arbitrarily large, then giant jagged spikes may pervade the visible surface in our fluid animations.
- **Memory and time limitations:** Triangles with small areas use space very inefficiently. Without any restrictions on triangle area, our simulations can get bogged down by millions of invisible triangles. This is bad for memory usage, and it is a waste of time to run computations on so many vertices.
- **Geometric computations:** We may decide to use different forms of geometric computation, such as collision detection, ray tracing, topology changes, or boundary integration. Triangles with poor quality can create serious robustness problems for these operations.
- **Numerical stability:** If we wish to use any numerical integration techniques on our surface mesh, such as surface smoothing or surface tension calculations, then we need to maintain a high quality triangle mesh.

4.2 Measuring triangle quality

A very simple way to detect troublesome triangles is by measuring the lengths of all edges in the mesh. If any edge is smaller than some minimum edge length, then it may create problems for our simulation. Similarly, if any edge is larger than some maximum length, then it may be too bloated and distorted to accurately sample the surface. Some triangles may have perfectly acceptable edge lengths but have very small areas (see Figure 11), so a simple edge-length measurement is not enough to find poorly-shaped triangles. In addition to measuring edge length, we can also measure the area and penalize very large or very small area measurements, or we can measure triangle angles and penalize extreme angles that are too large or too small.

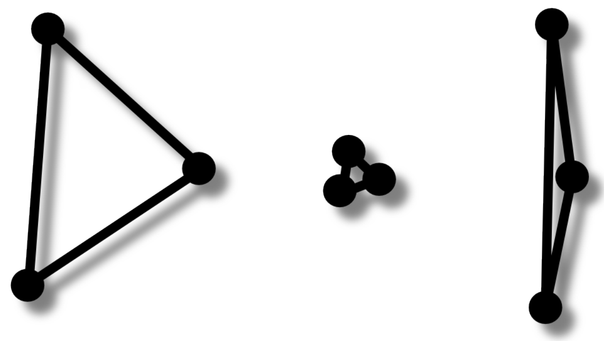


Figure 11: The leftmost triangle is of acceptable quality: it has large area, and its edge lengths are nearly equal. The middle triangle has a very small area, and its edge lengths are too short. The rightmost triangle has good-sized edge lengths, but its area is still too small.

The problem of finding a metric for triangle quality in a way that is meaningful and useful is very well studied. This section only briefly touches on some of the simplest measurements — just enough to detect the problematic triangles in our mesh so we can get rid of them. For a much more thorough discussion of triangle quality measures (and linear element quality in general), see [57].

4.3 Mesh quality operations

Once we have determined which triangles are unacceptable, we need to actually remove them from our triangle mesh. This section explains how to perform several different mesh operations in order to locally improve triangle quality. The operations covered in this section are the *edge split*, the *edge flip*, the *edge collapse*, and *null-space smoothing*. These operations are relatively independent from each other and can be performed in any order, though certain orders are more sensible than others. For a point of reference, the mesh-based surface tracking software *El Topo* [12] uses these operations in the same order that we present them in this chapter. The simulations in [71, 69, 64, 70] use only the *edge split*, followed by the *edge collapse* operations.

Each of these operations may be performed in a way that guarantees the resulting triangle mesh to be collision-free, and we include those details here these details here for completeness. These collision-free constraints can be safely disregarded if you don't care whether your surface has self-intersections, or if those intersections will be fixed by some topology-changing operation later.

4.3.1 Edge split

If an edge is longer than a user-defined maximum edge length, we subdivide it by introducing a new vertex (see Figure 12). The new vertex can be placed at the edge midpoint, which will not introduce any new intersections. However, we may wish to offset the new vertex from the current surface using a subdivision scheme to maintain curvature. We begin by introducing

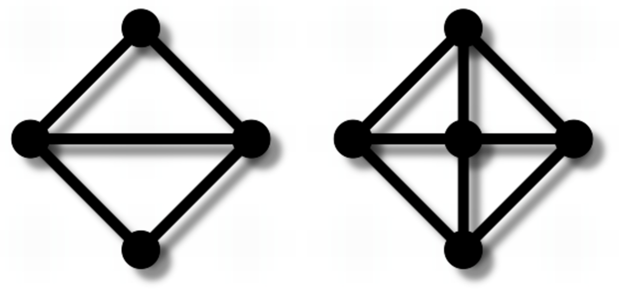


Figure 12: Edge split operation

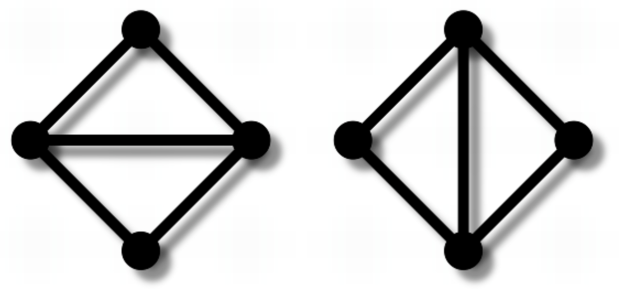


Figure 13: Edge flip operation

the new vertex at the edge midpoint, and we then compute the predicted location of the new vertex via the subdivision scheme. These two points define a *pseudomotion*. We check the new vertex and its incident triangles and edges as it moves from the edge midpoint to its predicted point to ensure that it does not collide with any existing mesh elements (which do not move during this pseudomotion). If a collision does occur, we can revert to using the edge midpoint, which is guaranteed to not introduce any new intersections.

4.3.2 Edge flip

We employ an edge flip operation as a way of maintaining good triangle aspect ratios. For each edge incident on two triangles, we check whether the distance between the two points *not* on the edge is less than the length of the edge. If so, we remove the edge and create a new edge between these two points (see Figure 13). A simple way of ensuring that this operation does not introduce any intersections is to check that no existing edge intersects the two new triangles and that no point lies inside the tetrahedron formed by the two new and two old triangles. We also reject the edge flip if it introduces a change in volume greater than a user-defined maximum volume change (we usually set this maximum volume change to be $0.1\xi^3$, where ξ is the average desired edge length; for simulations involving extremely thin surfaces, this may need to be further reduced).

Flipping a single edge may introduce new triangles with poor aspect ratios, so we iteratively sweep over all edges in the mesh until no flip is performed or until we reach a

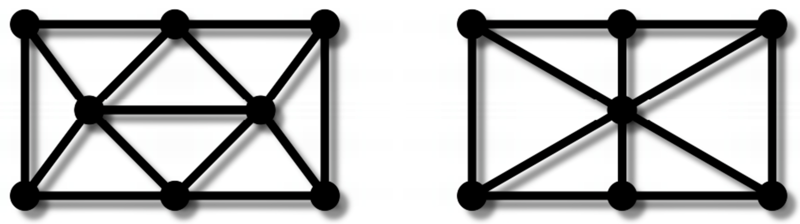


Figure 14: Edge collapse operation

maximum number of sweeps (in [12], we set this maximum to five). We also require that the new edge length decrease by a minimum amount to prevent the same edge from flipping back and forth on subsequent sweeps.

4.3.3 Edge collapse

If an edge is shorter than a user-defined minimum edge length, we attempt to collapse it by replacing it with a single vertex as shown in Figure 14. As with edge splitting, we treat only manifold edges, skipping edges incident on more than two triangles. We use a subdivision scheme to choose the location of the new single vertex in the general case. However, we also use an eigen-decomposition of the quadric metric tensor to detect vertices that lie on ridges or creases [32]. If one edge end point lies on a ridge and the other lies on a smooth patch of the surface, we set the new vertex position to be the position of the existing vertex on the ridge. In other words, we wish to prevent vertices moving off of the ridge, which tends to introduce bumps or jagged edges.

To ensure collision safety, we can use the same pseudomotion collision detection described during our *edge split* explanation earlier, this time with two vertices in motion: the end points of the edge. These end vertices will have the same predicted location: the location chosen by the subdivision algorithm. If a collision is detected during this pseudomotion, we can try again, this time moving the vertices towards the edge midpoint. Unlike edge splitting, however, we have no safe fallback vertex location. If we insist on having a collision-free mesh and cannot find a collision-free trajectory, then the edge collapse must be abandoned.

We use simple minimum and maximum edge lengths for determining when to split and collapse edges. In practice, we compute the average edge length when the mesh is initialized and set the minimum and maximum edge length parameters to be some fractions of the initial average length. This has the effect of keeping the edge lengths within some range of the initial average using split and collapse operations. In our examples, we allow edge lengths to vary between 0.5 and 1.5 of the initial average edge length; however, these parameters did not require tuning and our system remains stable for other values. More sophisticated criteria for triggering a split or collapse exist, such as detecting triangles whose areas are too small or too large, or aspect ratios that are too far from unity (c.f. [32]).

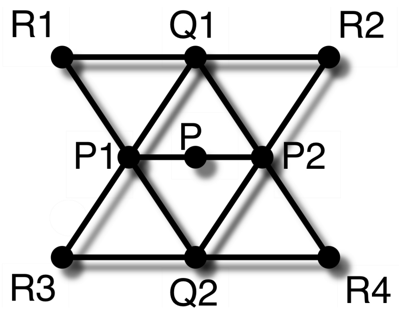


Figure 15: Butterfly subdivision

When choosing locations for new vertices during an edge collapse or split operation, there are a number of schemes that can be used. We use traditional butterfly subdivision [16] due to the simplicity of its implementation and because it is free of parameters. Quadric error minimization schemes [23] are promising, but in our experience the simplicity and quality of butterfly subdivision made it more attractive.

Butterfly subdivision determines the location of a new edge midpoint P_{new} as

$$P_{\text{new}} = \frac{1}{16}(8(P_1 + P_2) + 2(Q_1 + Q_2) - (R_1 + R_2 + R_3 + R_4)),$$

where P_1 and P_2 are end points of the edge, Q_1 and Q_2 are the vertices on the two triangles incident on the edge which are not the edge end points, and $R_1 \ldots R_4$ are the vertices on the four triangles adjacent to the triangles incident on the edge (see Figure 15). We have also found that the adaptive butterfly subdivision scheme of [75] worked very well, because it does not assume that all vertices have a valence of six. This adaptive subdivision scheme helps ensure high degrees of smoothness even for abnormal triangle configurations.

One potential danger to keep in mind is that non-manifold geometry can be created by collapsing an edge with a valence-3 vertex (see Figure 16. The edge collapse will result in two non-manifold triangles (each sharing the same three vertices). We must ensure that our surface geometry is manifold, so we can do one of two things: we can either forbid this edge collapse, or we can allow the collapse and delete any resulting non-manifold geometry. The first option is far simpler to implement.

4.3.4 Null-space smoothing

A powerful mesh improvement technique was recently introduced by Jiao [32]. Applying a Laplacian filter to the vertex locations would move each vertex to the average of its neighbors locations. This usually has the desirable effect of equalizing edge lengths. However, it will also shrink the volume enclosed by the surface and smooth away sharp features. We instead move the vertices only in the null-space of their associated quadric metric tensors. If the vertex is on a flat or smoothly curved patch of surface, the null space will correspond to the plane tangential to the surface at the vertex. If the vertex is on a ridge, the null space will be the infinite line defined by the ridge, and the smoothing operation preserves the ridge

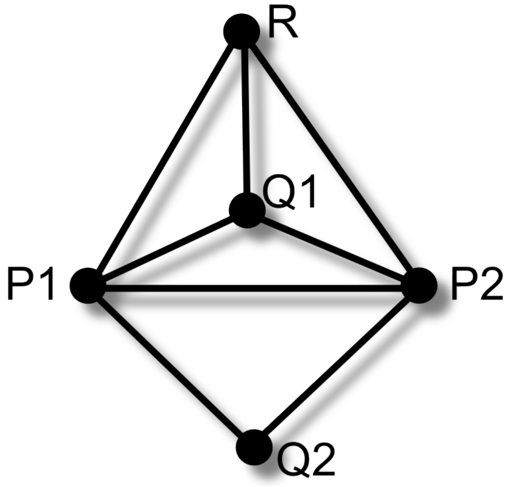


Figure 16: Example of an edge collapse that will result in non-manifold geometry. If P_1 and P_2 collapse into a single point P, then we are left with two triangles sharing the exact same 3 vertices (P, R, and Q_1).

feature. If the vertex is at a corner, the null space will be empty and the vertex will not move, preserving the corner. To ensure no mesh intersection, we treat the global smoothing operation as a pseudotrajectory on all vertices and apply collision resolution as if the surface was moving under the influence of an external velocity field.

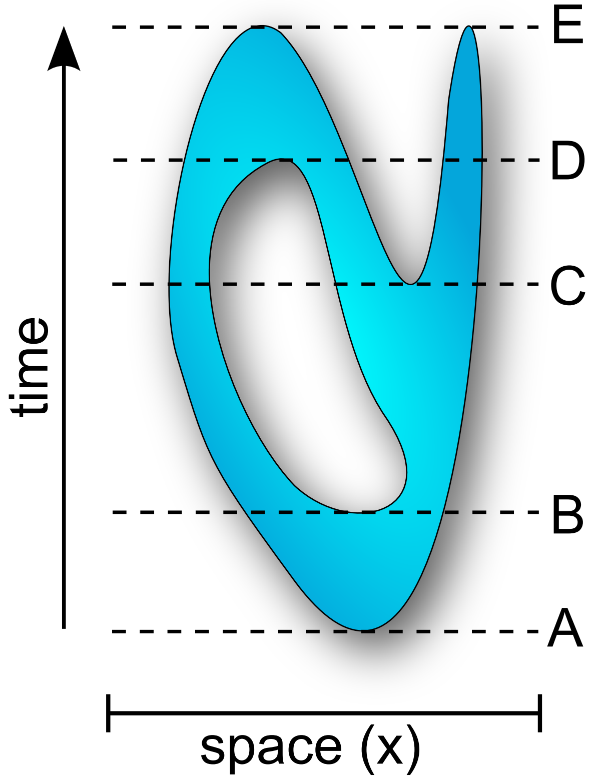


Figure 17: This figure shows a one-dimensional dynamic surface swept through time. Topological changes to dynamic surfaces can be represented as critical points in space-time, and these critical moments are marked by dashed horizontal lines.

5 Topology changes

Up to this point in the course, we have shown some methods for tracking a simulated liquid surface by deforming a triangle through time. As these surfaces evolve, they not only change shape and move around, but they can also merge together and split apart. Such events are called changes in the surface's *topology*.

To learn more about the behavior of a surface through time, we can plot the its position at every point in space and in time on the same graph. In Figure 17, we visualize a one-dimensional surface swept through time, with its position in space represented on the horizontal axis and time on the vertical axis. This time-swept figure creates a polygon in space-time. A vertical line in this graph represents a fixed point in space, and horizontal line represents a fixed instant in time. Whenever a topological change occurs, parts of the surface will either appear, disappear, split apart, or merge together. These instances can be identified as critical points in space-time. In Figure 17, a dashed line is drawn at each instant a topological change occurs. Line **A** marks the global minimum in time, signifying the initial creation of the surface. As time advances, the surface gets wider (occupies more space) until it splits into two pieces at instant **B**. In the period of time between **B** and **C**, the two new surfaces drift away from each other, as the right surface inflates. Instant **C** represents another local minimum, as the right surface splits apart once again. Line **D** marks our first local maximum, as the leftmost two surfaces merge together. Finally, the global maximum is found at time **E**, where the remaining surfaces shrink until they disappear.

5.1 Why do we need topological changes?

In order to faithfully simulate interesting natural phenomena, we must allow our dynamic surfaces to change their topology. These topological changes are not only necessary to capture realistic physical effects, but they are also a useful tool for simplifying overwhelmingly complicated systems for the purposes of computer animation.

5.1.1 Importance in nature

Topological changes occur naturally in several settings. If we are concerned with the dynamics of fluids and liquid surfaces, then topological changes occur whenever the surface breaks into droplets, or whenever different surfaces merge together upon contact. Cohesive forces between liquid molecules cause surfaces to merge together the instant they touch each other, and surface tension forces cause surfaces to rip apart whenever they form thin tendrils (this phenomena is known as a Rayleigh-Plateau instability, and it is primarily responsible for the degeneration of a splashing surface of water into a mist filled with tiny droplets). If our simulations do not allow topological changes such as merging, then surfaces will build up several layers of water separated by nothing but a vacuum. Similarly, if we do not allow surfaces to tear apart, then surface tension effects will routinely create infinitely thin tendrils of liquid. This type of behavior is distractingly unnatural, and it can also cause numerical stability problems in our simulations.

5.1.2 Importance in animation

Topological changes are also important for ensuring that our simulations have a manageable workload. In particular, if we neglect to merge together any surfaces, then only a few seconds of a dynamic splashing liquid will generate an overwhelming number of folded surfaces that should have naturally merged together. Aside from this behavior being unnatural, the computational load necessary to store and operate on each of these surface primitives will skyrocket. As mentioned above, we can also run into numerical problems if we allow surfaces to become arbitrarily thin without imposing topological changes. These problems result from the inevitable division by very small numbers as the angles and edge lengths of surface triangles shrink. In order to allow for fast and stable computer animations of physical phenomena with dynamic surfaces, we must properly handle their changing topology.

5.1.3 Dynamic topology changes in practice

In practice, because we use a triangle mesh that evolves through time as our dynamic surface, we are interested in methods for explicitly enforcing topological changes on a triangle mesh. Unfortunately, the solution is not straightforward. Many arbitrarily complicated shape configurations can arise, and our code must handle every one of them robustly. It is not always as simple as identifying two surfaces that are close to each other and then sewing them together — self-intersecting surfaces can form extremely complex shapes. We must also guarantee that the surface is closed, manifold, and consistently oriented after every one of our topological operations — otherwise, many important assumptions will be violated and the physics simulation will fail.

Within this course, we will discuss three practical solutions to this problem of handling topological changes with a triangle mesh. Section 5.2 will explain how to enforce topological changes by reconnecting triangles when they come close together and guarantee a collision-free state of the mesh. Section 5.3 will explain how to leverage our fluid simulation grid in order to replace the surface mesh with a topologically corrected one. Finally, Sections 5.4 and 5.5 will explain how to only locally re-sample the surface in order to limit the amount of surface re-sampling.

5.2 Topological operations on the mesh itself

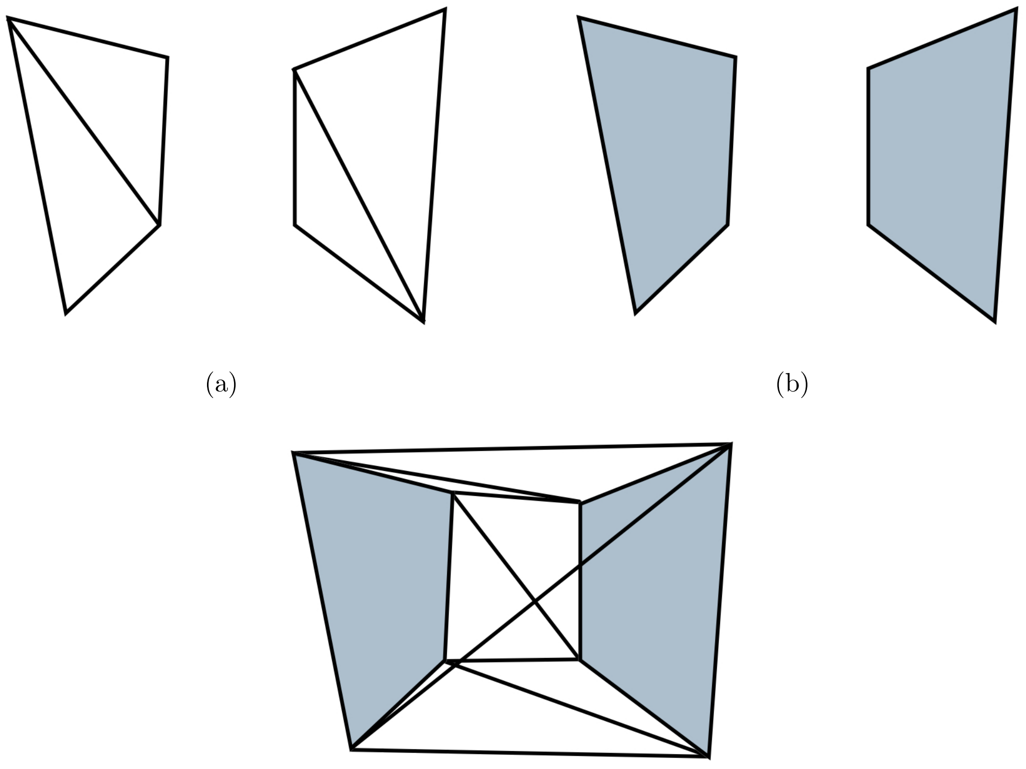
In this section we will describe a few operations that will locally modify the mesh connectivity to result in a change in topology. The key idea to make this tractable is to require that every mesh operation should leave the mesh in a consistent, non-intersecting state—as opposed to attempting to recover such a state after the fact. Therefore, in this section we will show how to use robust collision detection methods to ensure we detect every possible violation. Once a collision is detected, we either roll back the operation if it is deemed non-critical and may be delayed to a subsequent time step when it may succeed, or otherwise minimally perturb mesh positions to guaranteeably avoid the problem. Our mesh perturbation is patterned after frictionless inelastic collision response in a physical contact problem.

(In upcoming sections we will see alternative approaches that *do* allow the mesh to self intersect, then reconstruct a consistent surface. Each approach has its own set of trade-offs.)

We will first describe a method for allowing surface patches which are close to each other to safely *merge* without introducing any intersections, then introduce two methods allowing meshes to *separate* when they become too thin. We will finish this section by describing how to keep the mesh intersection-free even if topology changes fail.

5.2.1 Mesh merge

To achieve a surface merge operation, we seek out edges that are close in space but are on distinct surface patches, and attempt to merge the surface. We first search for edges that are closer than a specified tolerance, then sort the pairs of edges in order of increasing separation distance so that the nearest edges are merged first. For each pair in the sorted list, we first remove the triangles incident on each edge. This introduces two temporary "holes" in the mesh, each hole consisting of a loop of four boundary edges. We create eight new triangles between the two holes, using a closed-form triangulation. (This sequence is shown in figure 18.) We then use intersection testing to determine if these new triangles



(a) (b) (c)

Figure 18: **Merge operation.** (a) Two edges are in close proximity to each other. (b) The edges are deleted, temporarily creating two holes in the mesh. (c) The holes are closed using eight new triangles.

intersect any existing mesh elements or each other (treating degenerate cases as intersections for safety). If an intersection is found, we discard the new triangles and replace the original triangles incident on the proximal edges, abandoning the topology change.

This merge operation may introduce degenerate tetrahedra which must be detected and deleted. The operation can also be aborted if the edge neighbourhoods are not distinct or otherwise degenerate in such a way that we cannot perform our closed-form remeshing. If we must abandon a topology change, we will need to rely on collision detection and resolution in order to maintain the intersection-free invariant (described below).

5.2.2 Mesh separation

The mesh “zippering” operation just described cannot separate or pinch a mesh to create two disjoint volumes, so we must treat this with a separate operation. Two alternative approaches to cutting the mesh at thin necks or spindles are detailed here.

The first approach, as described by Wojtan et al. [70] is to detect thin spindles of liquid while performing the local surface maintenance operations. These thin spindles are detected by flagging edges that will produce non-manifold geometry if collapsed. Of these non-manifold cases, we can easily identify any thin spindles by their triangular cross-section. To perform the mesh surgery, we cut the mesh at the triangular cross-section, seal each end with a new triangle, and perturb the two new strands away from each other by a small offset to avoid intersection. A similar operation is explained in detail by Lachaud et al. [37].

An alternative approach to separating meshes was introduced by Brochu et al. [12]. We must first relax our requirement that the surface meshes must be manifold. In particular, we allow more than two triangles to be incident on an edge. We do not, however, allow two triangles to share the same three vertices, thus creating a zero-volume tetrahedron, and we do not allow open surfaces, since we must still have an "inside" and an "outside" for the fluid simulation to work. An edge collapse or merge operation may introduce degenerate tetrahedra, so after performing either of these operations, we search the surface meshes for degenerate tetrahedra, and delete the two offending triangles. We also delete triangles that may have repeated vertices ("collapsed" triangles).

After this sweep, we deal with surfaces which may be connected only at a single vertex. These so-called "singular" vertices can be detected if their incident triangles are not all connected. If this is the case, we partition the set of incident triangles into connected components. For each component, we create a duplicate vertex and map all triangles in the component to this new vertex. A similar procedure is described by Guéziec et al. [24]. We also move the duplicate vertices very slightly towards the centroid of their associated triangles to avoid problems with collision detection and resolution which may occur when two vertices occupy exactly the same point in space.

5.2.3 Maintaining the intersection-free invariant

Up to this point, we have discussed how to locally improve mesh quality and change surface topology without introducing self-intersections in the surface. However, since we have taken the stance that we will not attempt to fix self-intersections after they occur, we must still deal with collisions that may occur when the mesh is advected according to the fluid simulation velocity field. The topology change operations are designed to prevent this from happening by merging surfaces that are close to each other before they collide. However, there may be topological operations that must be prevented because they would introduce self-intersections, or degenerate configurations. Also, since the the mesh moves in discrete steps, it may move from a non-intersecting state into an intersecting state, without first entering into a "proximal" state if the fluid velocities are great enough or the interval between discrete states is large enough.

To deal with these problems, we use continuous collision detection and resolution, adapted from the cloth simulation literature.

5.2.4 Interference detection

We differentiate between three types of geometric interference detection: *intersection* detection, *proximity* detection, and *collision* detection. We use all three of these types at different steps in the collision processing algorithm.

Static *intersection detection* detects if and where a mesh intersects itself for a given mesh configuration (i.e. at one instant in time). This can generally be decomposed into primitive tests discovering where an edge is penetrating a triangle, but we must take care to identify degenerate cases, such as an edge penetrating a surface only at an edge or at a vertex.

Static *proximity detection* detects when mesh elements are closer than a specified tolerance (in particular, when a vertex is close to a triangle or when two edges are close to each other). We will name this tolerance ϵ_p. Our proximity detection function can also return a “collision normal”, $\mathbf{n}$, which, when an impulse is applied along it, will increase the distance between elements.

Proximity detection finds the two points on the pair of mesh elements that are closest to each other. If we denote the set of four barycentric coordinates of these two points as $\mathbf{a}$ (setting $a_i = 1$ if i is the vertex in a vertex–triangle collision), then to find the distance between the mesh elements, we multiply the barycentric coordinates by -1 if they refer to a point on the triangle or on the second edge in an edge-edge proximity, to get a new set of coordinates, $\bar{\mathbf{a}}$. Then taking the sum of vertex locations weighted by these scaled barycentric coordinates yields a vector between these closest points. If $\mathbf{p}$ are the indices of the vertices involved, then the shortest distance is given by the length of this vector:

$$d = \left\| \sum_{i=1}^{4} \bar{a}_i \mathbf{x}_{p_i} \right\|$$

Continuous collision detection (CCD) detects whether a collision between a moving vertex and a moving triangle or between two moving edges will occur during some specified time span.

In our framework, two mesh configurations are passed into the collision detection routines: one at time t_n and one at time $t_n + \Delta t$. We assume vertices move in a linear trajectory from their positions at time t_n to their positions at time $t_n + \Delta t$, and that the mesh connectivity does not change over this time step. Given these two configurations, continuous collision detection will return any point-triangle and edge-edge collisions, as well as the time that the collision occurs (sometime between t_n and $t_n+\Delta t$), the collision normal, the set of barycentric coordinates describing the point of contact, and the computed relative displacement.

The El Topo surface tracking library currently uses the collision detection approach introduced by Provot [53]. This method makes the simplifying assumption that, within time steps, mesh vertices move on constant, straight-line velocity paths, and the edges and triangles between them are linearly interpolated at every intermediate time. The times at which a point and a triangle or two edges become coplanar (a necessary condition for collision) are the roots of a simple cubic equation in this model, which is simple enough to solve. At these times, the proximity of the elements can be evaluated to determine if a collision occurs. Bridson et al. [9] introduced error tolerances in the cubic solver, and an error tolerance on proximity testing at the coplanarity times, to account for rounding errors in the process, effectively eliminating any false negatives (undetected collisions) in the CCD process.

5.2.5 Collision resolution

Detecting collisions is only half the story — we must also perturb the mesh to avoid intersections. Our collision resolution procedure is based on the filtering approach for handling collisions [9], and operates in three phases. First, we run proximity detection as described in section 5.2.4 to obtain pairs of elements that are closer to each other than ϵ_p. For each pair of proximal elements, we compute the relative normal velocity of the elements. We then perturb the vertex velocities so that the new relative normal velocity is positive, and large enough to carry the vertices at least ϵ_p away from each other if they were integrated forward for Δt without further interference. Attempting to maintain this small minimum separation significantly helps in avoiding degenerate geometric cases which would otherwise slow subsequent floating-point-based collision detection and resolution.

As described in section 5.2.4, for a pair of elements, proximity detection returns a distance d and a set of scaled barycentric coordinates $\bar{\mathbf{a}}$. If $\mathbf{p}$ are the indices of the element vertices and $\mathbf{u}$ are the vertex velocities, then the relative velocity is:

$$\mathbf{u}_{\text{rel}} = \sum_{i=1}^{4} \bar{a}_i \mathbf{u}[p_i]$$

If $\mathbf{n}$ is the unit-length collision normal, the impulse J we apply is computed as:

$$\delta = \frac{\epsilon_p - d}{\Delta t} - \mathbf{n} \cdot \mathbf{u}_{\text{rel}}$$
$$J = \frac{\delta}{\langle \bar{\mathbf{a}}, \bar{\mathbf{a}} \rangle_{M^{-1}}}$$

where M is the diagonal matrix of vertex masses. (In problems where there is no natural mass for a surface vertex, we simply use a unit weighting: $M = I$.)

Then for each vertex in proximity, we distribute the impulse J to perturb the predicted velocity field:

$$\mathbf{u}_{p_i} = \mathbf{u}_{p_i} + J \frac{\bar{a}}{M_{p_i}} \mathbf{n}$$

Note that this will not immediately resolve any of the proximities detected, as the "current" vertex positions are left untouched; it aims to resolve the proximity at the next time step. More importantly, it tends to dramatically reduce the number of collisions that must be dealt with in the next phase.

In the second phase of collision resolution, we use continuous collision detection to determine pairs of colliding elements. Our CCD function returns the relative displacement of the elements in the direction of the collision normal (which we can scale by $1/\Delta t$ to compute the relative normal velocity), as well as the barycentric coordinates that should be used to distribute the corrective impulse. For each pair of colliding elements we encounter, we apply an impulse that sets the relative normal velocity between the elements to zero, thus preventing the collision from occurring. This is similar to the repulsion impulses applied in the previous phase, except that the impulse magnitude is:

$$\delta = -\frac{\mathbf{n} \cdot \boldsymbol{\Delta}\mathbf{x}_{\text{rel}}}{\Delta t}$$

This is equivalent to introducing an impulse that instantaneously changes the velocity, while minimizing the velocity change in the normal direction in a least-squares sense. (Minimizing the normal velocity change in this way ensures that momentum is conserved, if the least-squares metric is kinetic energy.) One sweep through all mesh elements will not prevent all collisions, as resolving one collision may introduce a new collision between a pair of elements that was already checked. We have found that applying three sweeps of this individual collision resolution handles most collisions: however we must use a fail-safe to ensure that all collisions are handled.

For our fail-safe, we use the simultaneous treatment of collisions developed by Harmon et al. [25]. After three passes of individual collision resolution, we detect all pairs of elements that are still colliding. We lump colliding pairs of elements into "impact zones" based on adjacency and resolve all collisions in each zone simultaneously using one linear solve. We can think of our desired new velocities $\mathbf{u}'$ as being the solution to a constrained minimization problem:

$$\begin{array}{l} \min ||\mathbf{u}' - \mathbf{u}||_M^2 \\ \text{subject to } \mathbf{n} \cdot \mathbf{u}'_{\text{rel}} = 0 \;\; \text{for all collisions} \end{array}$$

We can re-write the constraint as a linear operator on the vertex velocities by building a matrix C, where each row, C_i, corresponds to one collision, and has non-zero entries in block columns $\mathbf{j} = [3v, 3v+1, 3v+2]$, where v is one of the four vertices involved in collision i. Setting $C_{i,\mathbf{j}} = \bar{a}_v \mathbf{n}^T$, our constrained optimization problem becomes:

$$\begin{array}{l} \min ||\mathbf{u}' - \mathbf{u}||_M^2 \\ \text{subject to } C\mathbf{u} = \mathbf{0} \end{array}$$

Solving this using the method of Lagrange multipliers yields the system:

$$CM^{-1}C^T\lambda = C\mathbf{u}$$

We can think of λ as the set of impulses which, when applied, yields zero relative normal velocities for all collisions. We update the vertex velocities according to:

$$\mathbf{u}' = \mathbf{u} + M^{-1}C^T\lambda$$

The application of these impulses may result in new collisions, so we run collision detection again and add any additional collisions to the set of impact zones, repeating the process until no new collisions are detected. This is guaranteed to terminate, assuming adequately accurate linear solves, since each additional constraint reduces the finite dimension of the solution space; in practice it proves to be very efficient.

5.3 Global grid-based re-meshing of the surface

In this section, we will outline a very different method for changing the topology of a mesh-based fluid surface. Here, we start with the mesh-based fluid simulation outlined in Section 3, and then use the Eulerian grid to globally reconstruct the surface mesh. The basic strategy for globally re-sampling this triangle mesh looks like this:

- 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.

5.3.1 Core algorithm

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, so, in a second stage, we fix the mesh 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.

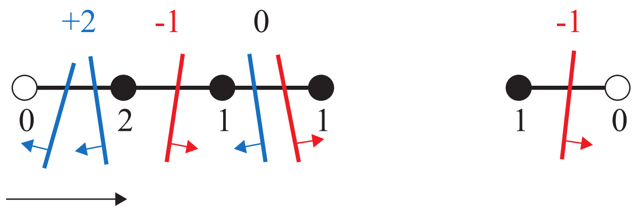


Figure 19: Entry and exit intersections are summed up in the state change variable of each cell edge to determine whether the grid nodes are inside our outside the surface.

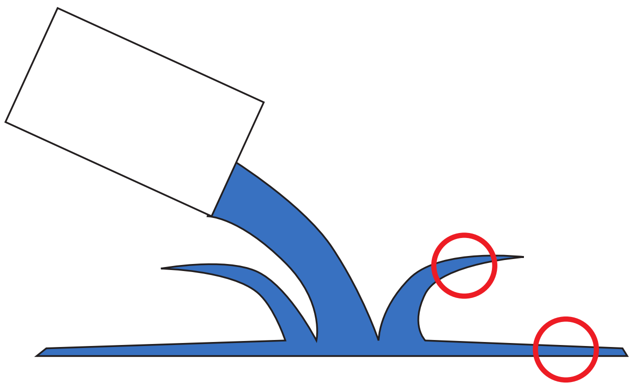


Figure 20: Using special marching cubes templates, this method can preserve thin layers such as sheets and shallow puddles.

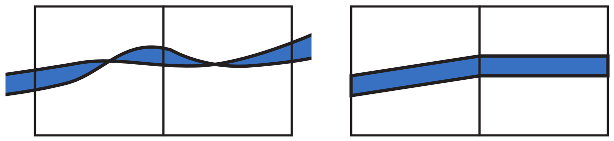


Figure 21: Explicit advection of the two surfaces of thin features often causes self-intersections. This method resolves these automatically.

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 [63] and only store the cells that are intersected by the input mesh (see Figure 22).

First we determine the intersections of the input triangles with the edges of the grid cells. Each intersection has a type 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. 19). 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 Figure 22). 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

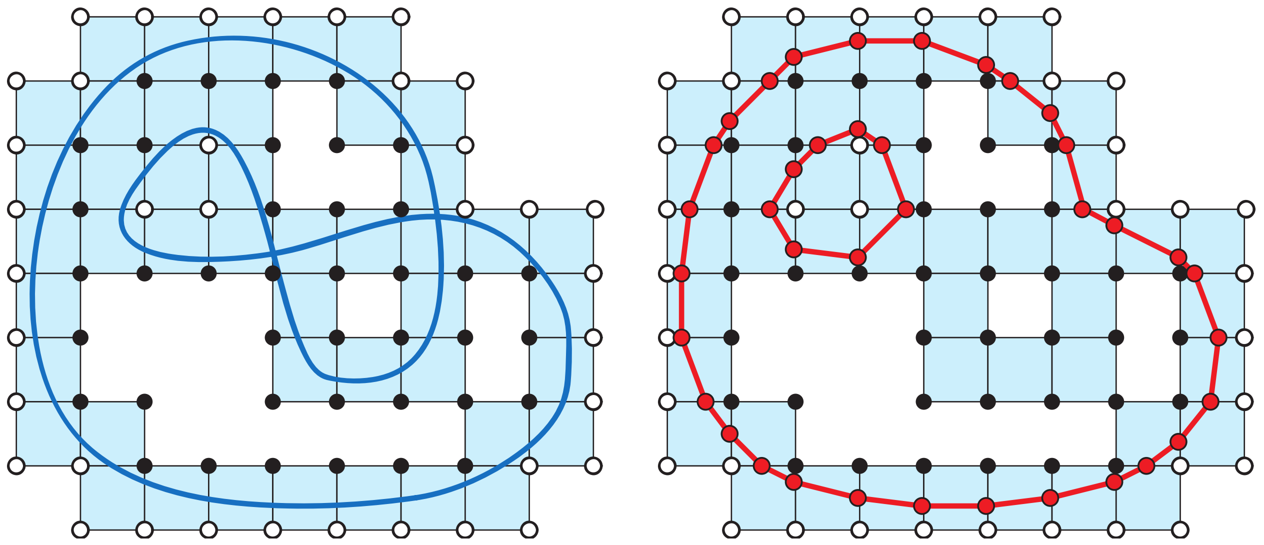


Figure 22: Two dimensional case. Left: The input mesh and the sparse grid with nodes marked as inside or outside. Right: The new mesh created using the marching squares templates.

inside. This way, we can also handle defect meshes with holes robustly. A similar approach was used in [30] 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 [38] 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 required.

5.3.2 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

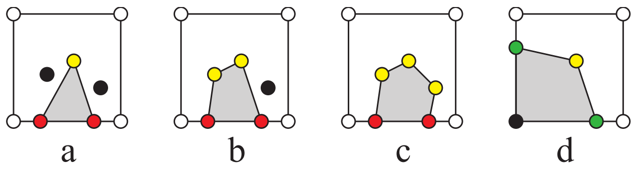


Figure 23: Fronts and edges are preserved by considering vertices of the input mesh inside cells.

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 Figure 20).

The 2d case

Let us first look at the 2d case. The top row in Figure 24 shows the standard marching squares templates modulo rotation. On each cell edge with adjacent nodes of different type a vertex is created (shown in green). As Figure 19 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 Figure 24). 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 (Figure 24). 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 do not 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). Figure 23 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 effective way to preserve sharp edges and conserve volume (see Figures 27 and 28).

Some of the templates have dual configurations which are shown on the right side of Figure 24. 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

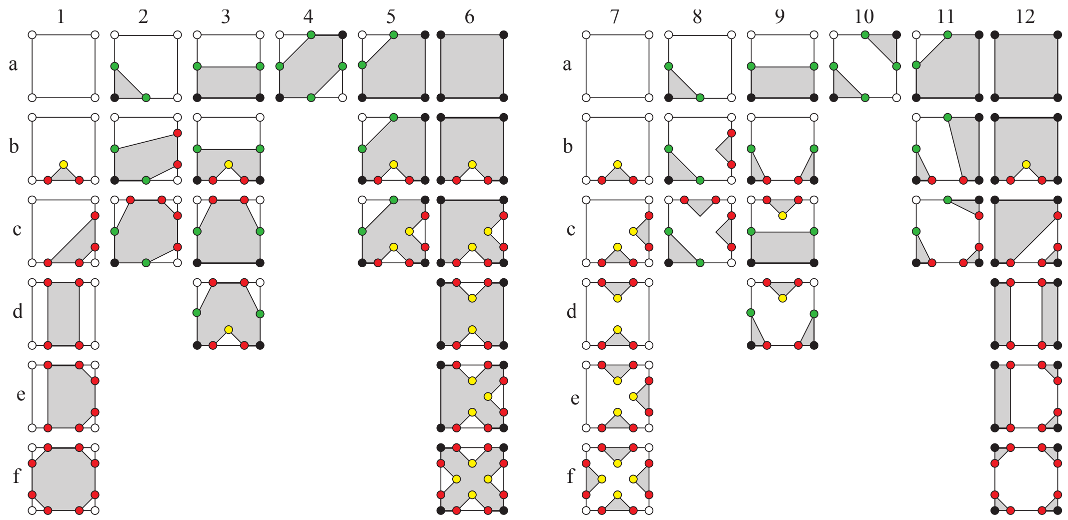


Figure 24: Top row: Standard marching squares templates. Green vertices exist between inside (black) and outside (white) nodes. Below: Each edge connecting nodes of the same type can contain one interval of the opposite type bounded by two additional (red) vertices. Some templates require inner vertices (yellow) as well. Columns 7-12 are dual templates of those in columns 1-6. Cases 4a and 10a are the pair known from regular marching squares.

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 on the left of Figure 24 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 Figure 24.

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.

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 Figure 25). 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

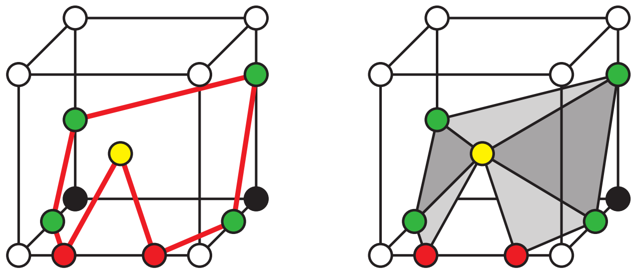


Figure 25: Triangles of a grid cell are created by applying the 2d templates to all 6 faces and triangulating the resulting closed loops of segments.

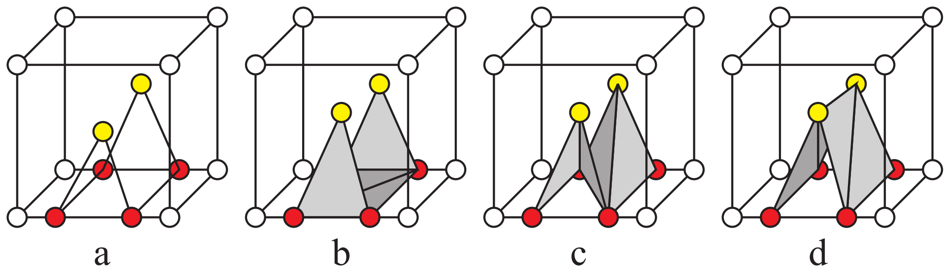


Figure 26: Triangulation of a boundary (a) is not unique. The correct one is (d). Triangles on faces are not allowed (b) and (c) is an undesired configuration.

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 [42]. This has to be done carefully. Figure 26 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 bitmask 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 ears plane, thereby maximizing the enclosed volume.

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.

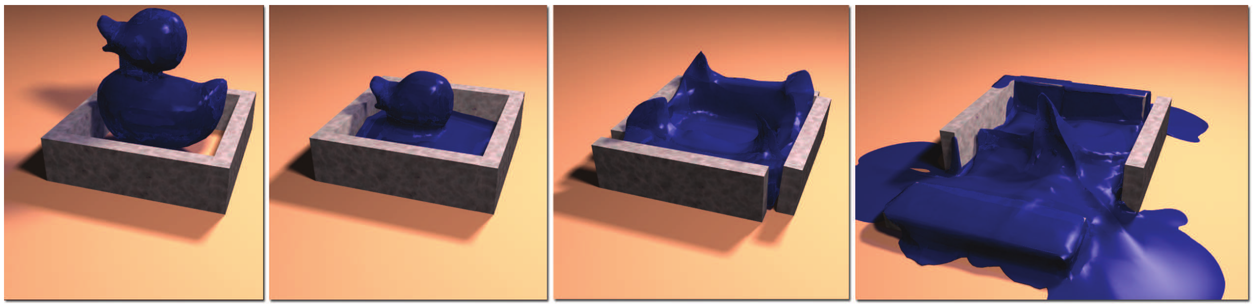


Figure 27: Handling topological changes with global grid-based re-meshing: This simulation shows an unbounded Eulerian fluid simulation showing thin sheets, sharp features and shallow puddles.

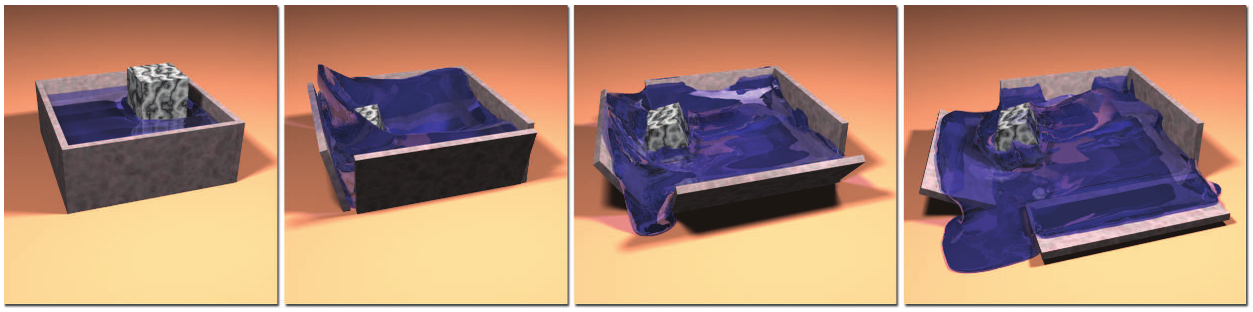


Figure 28: Handling topological changes with global grid-based re-meshing: Two way interaction with rigid bodies.

5.4 Local grid-based re-meshing of the surface using marching cubes

In the previous section, we introduced a method for handling topological changes to the deforming fluid surface by replacing the entire surface mesh with a new one derived from marching cubes templates. While this technique is extremely fast, effective, and easy to implement, it unnecessarily re-samples the surface mesh where no topology changes are needed.

This section is concerned with only locally re-meshing the surface in order to handle topology changes, while leaving the rest of the surface untouched. This local handling of topological changes has the advantage of never re-sampling the surface unless it is involved in a topology change, which can limit numerical smoothing errors. However, this local re-sampling algorithm requires us to define exactly which cells need to be re-sampled, and we need to guarantee that we don't leave any holes in the mesh when we sew the new mesh together with the old one.

Much like the previous section used local marching cubes templates to replace the mesh surface in each grid cell, this section will explain how to use marching cubes templates for local surface replacement. However, because the algorithm in this section can not retain thin surface sheets, we introduce a more general method based on local convex hulls that reproduces thin surface sheets and strands in Section 5.5.

5.4.1 Overview of approach

This method begins very similarly to the method described in Section 5.3. We first deform the fluid surface mesh by advecting it through the fluid's velocity field. Next, similar to the *inside / outside* classification in Section 5.3.1, we compute a signed distance field from the mesh. We then compare the surface mesh to its grid-sampled signed distance field in order to detect where to locally change the topology of the surface. We then locally recompute the surface mesh in these areas and connect the new surface patches to the original surface. We are then free to repeat this process for the next step of the fluid simulation. Figure 29 shows a diagram of this algorithm.

5.4.2 Detection of topological events

After advecting the fluid surface mesh as explained in Section 3, we then decide whether we should split any surfaces apart or merge them together. We first calculate a signed distance field D from our surface mesh, and then we examine the structure of this field to help decide where any topological events should take place.

Signed Distance Field Calculation

We choose to place our signed distance function on a regular grid that encloses our surface mesh. Note that the only places where topological changes can occur are at grid cells that intersect the mesh (the surface cells). Thus we calculate the distance to the triangle mesh at

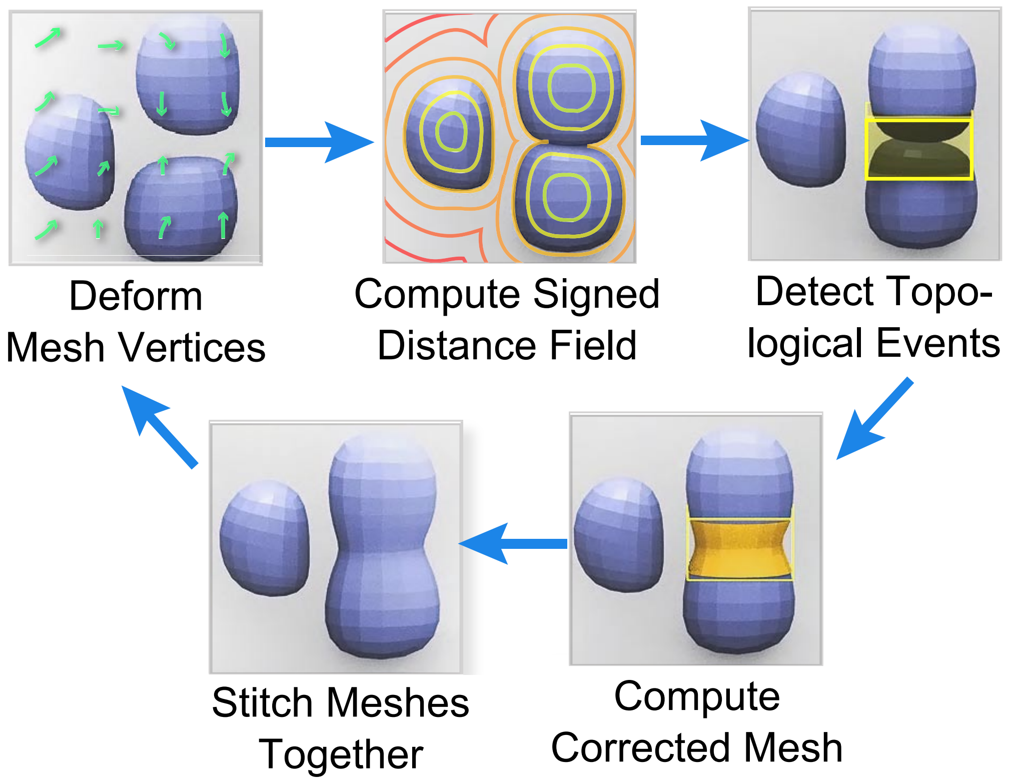


Figure 29: Overview of the topology modification pipeline using local grid-based marching cubes templates, beginning in the upper left by deforming the input mesh. The explicit mesh surface is given as input, and its signed distance function on a grid implicitly represents a similar surface with simpler topology.

each grid point that touches a surface cell by calculating the exact distance to each nearby triangle and taking the minimum. We then use the voxelization method described in Section 5.3.1 to determine which grid points lie *inside* of the mesh, and which lie *outside*. We assign a positive signed distance to the grid points outside of the mesh and a negative sign to the points inside of the mesh. Because we only calculate the signed distance at grid cells touching the surface, and because we only sample the nearest triangles for each distance query, this calculation of the signed distance function can be made quite efficient.

Complex cell test

At this point, we have two surface representations: an explicit surface mesh M, and a signed distance function D that implicitly defines a surface wherever the distance is zero. We can contrast these two surface representations to give us an idea of where the surface should be re-sampled. Any grid cell where the explicit mesh surface is connected significantly differently than the implicit grid surface will be referred to as a *complex cell*. More specifically,

- A *complex edge* is an edge in the grid that intersects the mesh more than once.

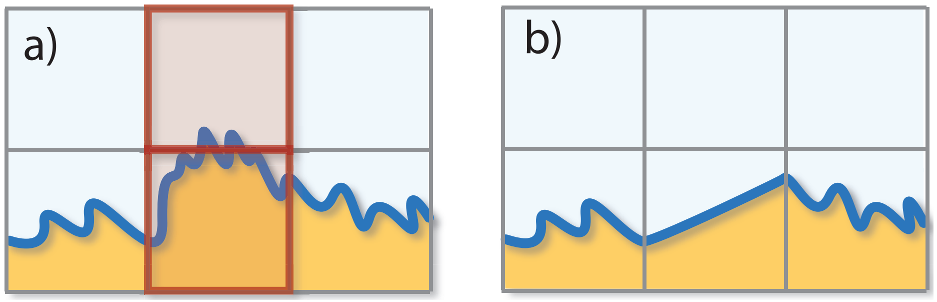


Figure 30: This two-dimensional example shows how the complex cell test will aggressively re-sample detailed surface features, even in the absence of topological changes. In (a), we show cells that the complex cell test will mark due to multiple edge intersections, and we show the re-sampled surface in (b).

- A *complex face* is a square face in the grid that intersects the mesh in the shape of a closed loop or touches a complex edge.
- A *complex cell* is a cubic grid cell that has any complex edges or complex faces, or any cell that has the same sign of the signed distance function at all of its corners while also having explicit geometry from the mesh embedded inside of it. We will refer to the act of testing a cell for complexity as the *complex cell test.*

This complex cell test can be used to mark a region in space where any topological changes occur in our surface. However, a straightforward application of this test will also mark detailed surfaces as topologically complex (see Figure 30). In order to preserve surface details like sharp corners, we must significantly modify this test. We will now describe our modified version of this test, the *deep cell test.*

Deep cell test
Because we are primarily interested in the fidelity of the visual surface, we would prefer that most surface re-sampling occurs only in invisible regions or in the presence of major topological changes. To avoid the excessive re-sampling of highly detailed surfaces, we do not wish to mark all complex cells as topological events. Instead, we start by contrasting the interface of our explicit surface M with the interface of our implicit surface D. In the interest of surface detail preservation, we ignore any complex cells that are sufficiently close to both the interfaces. We only mark a complex cell that is at least one cell away from the isosurface interface, as illustrated in Figure 31. This *deep cell test* is necessary because subtle bumps in surface geometry can still trigger any complex cell test, but only cells with geometry fluctuations larger than a cell length will be marked by our deep cell test.

Self-intersection tests
In addition to topological changes triggered by surface proximity, we also choose to mark cells that indicate significantly large self-intersections in the mesh. This test is performed while

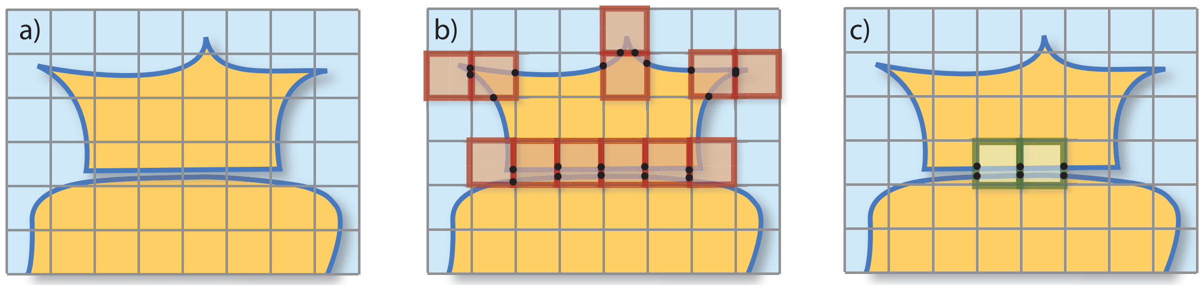


Figure 31: A two-dimensional illustration of our deep cell test. Figure (a) shows an input surface mesh M (dark blue line) with a visualization of the corresponding signed distance field D, where orange points are inside of the surface, and light blue points are outside. Next is a figure showing all complex cells (b). The rightmost figure shows all deep cells (c). Note that the deep cells only label geometry necessary for a topological change, while the complex cells aggressively label important surface details.

marching rays through the mesh in the voxelization phase of our signed distance calculation (Section 5.3.1). Whenever a ray (grid edge) intersects the surface, we determine whether the ray is entering or leaving the surface by calculating the dot product of the ray direction with the triangle normal: a negative dot product indicates that the ray is entering the surface, while a positive dot product indicates an exit ray. In our implementation, we initialize an integer variable to zero at the start, and then we increment the value if the ray enters the surface and decrement the value if the ray leaves. As the ray passes through each grid node, every node that does not have an integer value of zero or one necessarily is in a region of multiple overlapping or inside-out surfaces. We mark any cells touching these grid nodes as topologically complex. An example of such "inside-out" or self-intersecting surfaces can be seen in Figure 22.

Once we have marked every cell that we wish to re-sample, we want to replace M in these cells with a topologically-simple isosurface extraction. However, we need to ensure that every cell on the boundary of our marked cell region provides a topologically simple interface with marching cubes. In other words, the boundary must contain no complex edges and no complex faces. We execute a flood fill algorithm, starting with the initially marked cells and marching outward across complex faces, until the entire region of marked cells is bound by topologically simple cube faces (see Figure 32). Once this region has a clean interface with marching cubes, we can perform surgery on the mesh.

5.4.3 Altering the mesh topology

At this stage in the algorithm, we have a mesh M, a distance field D, and a list of marked grid cells. Each of these marked cells describes a region of space in which we will locally remove the topologically complex explicit surface M and replace it with the topologically simple extracted isosurface of D. The surface inside of these cells is computed with a marching cubes algorithm and connected to the original mesh at the cell boundaries. To maintain a

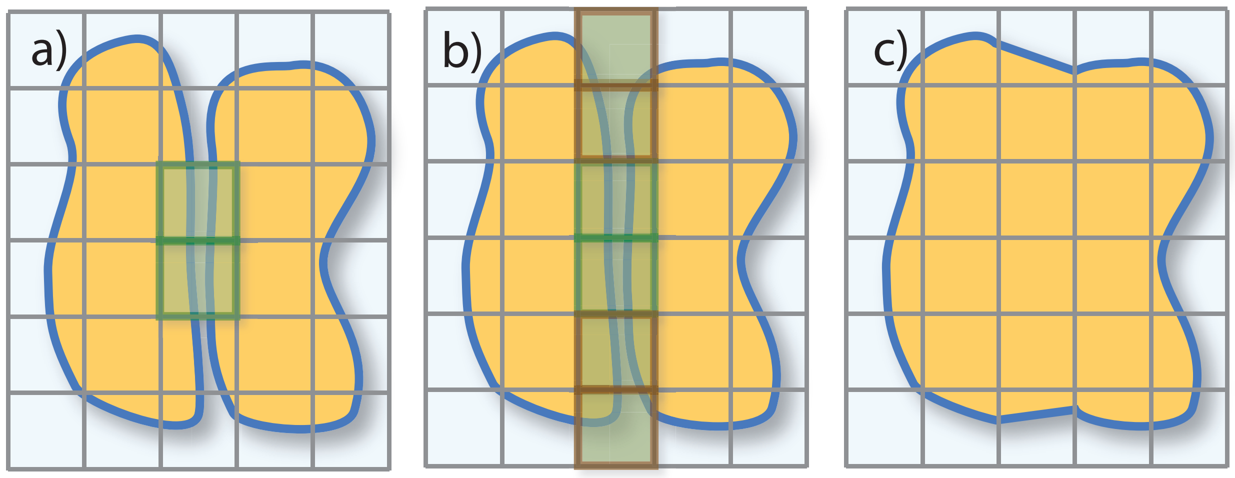


Figure 32: A two-dimensional illustration of the cell-marching step. In (a), all deep cells are marked. We march outward along complex edges and faces until the marked region is topologically simple (b). The right figure (c) shows a topological change after re-sampling the marked cells.

manifold surface mesh, it is important to ensure that the interface between the isosurface and the explicit surface matches up perfectly. We spend the rest of this section explaining how to compute this interface efficiently and robustly.

Sewing meshes together

Here, we describe the basic method for matching the interface between an explicit triangle mesh and an extracted isosurface. We also explain this algorithm graphically in Figures 33 and 34.

1. After marking topologically complex cells, we find each triangle that intersects a cell edge on the boundary of the marked region. We then calculate the intersection point between the triangle and edge, and we subdivide the triangle into three new triangles that share a vertex at the intersection point. We call this newly created vertex a *type 1* vertex.

2. Next, we find all triangle edges that intersect a cell face on the boundary of the marked region. We split each triangle edge at the point where it intersects the face, subdividing the two original triangles into four and inserting a new vertex on the face. We call this face vertex a *type 2* vertex.

3. After all subdivisions have been performed, no triangles will cross the faces of any marked cells. That is, each triangle will lie completely inside or completely outside of the marked region. We delete all triangles that lie completely inside of the marked region.

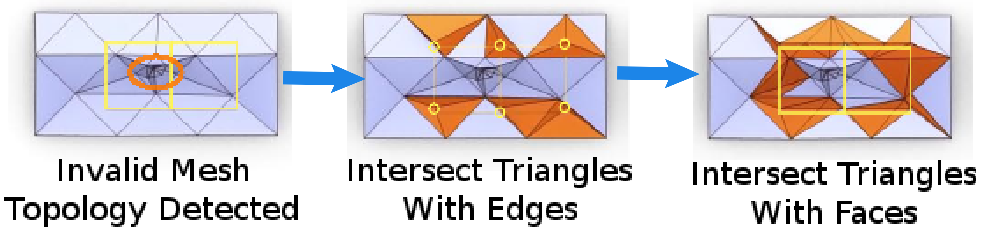

Invalid Mesh Topology Detected
Intersect Triangles With Edges
Intersect Triangles With Faces

Figure 33: We first create *complex cells* (yellow, left) wherever invalid topology is detected. We create type 1 vertices by subdividing triangles where they intersect the edges of a complex cell (middle), and then we create type 2 vertices by subdividing triangles where they intersect the faces of a complex cell (right). After these steps, the mesh triangles are either completely inside or completely outside of the complex cell region. We will delete the triangles inside of these complex cells and replace them with new triangles from marching cubes templates.

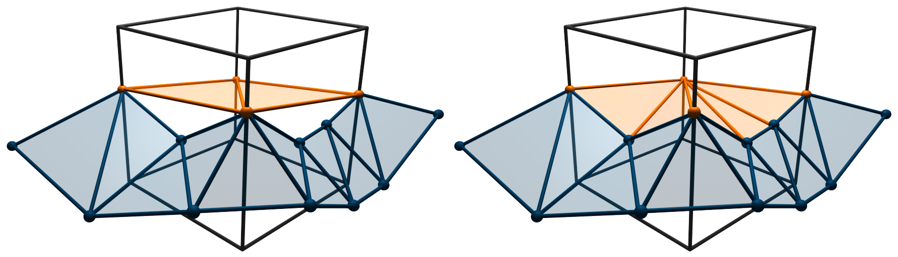


Figure 34: After deleting the triangles inside of the complex cells and replacing them with triangles from marching cubes templates (left), we have to sew the surfaces together along the boundary of complex cells (right). We do this by subdividing the new triangles (orange) and connecting these newly-created vertices to the type 2 vertices of the original surface mesh (blue).

4. We use marching cubes to generate a triangle mesh in the marked region, and we connect the meshes together at the type 1 vertices.

5. Now we want to sew the surfaces together along the faces of each complex cell. We first ensure that each new triangle shares an edge with at most one triangle outside of the complex cell region — for each new triangle that shares edges with two or more triangles outside of the complex region, we place a vertex at its barycenter and split it into three new triangles.

6. For each new triangle that does not perfectly match up with a segment of the boundary curve, we subdivide it in two by adding a point on its boundary edge and snapping it to one of the vertices on the boundary curve. We then recursively subdivide each of these newly-created triangles in the same way until the curves perfectly match (each curve eventually reduces to a single line segment in the base case). See Figure 34 for a visual aid.

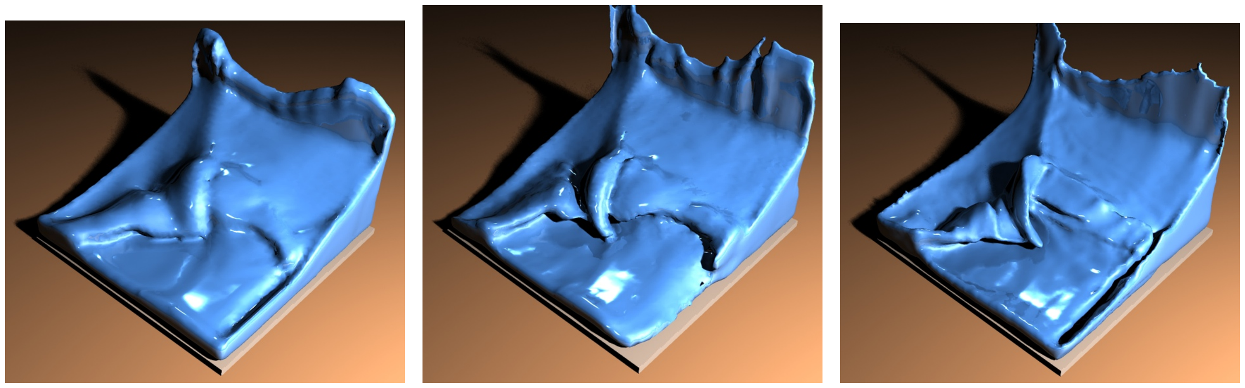


Figure 35: Comparison between different surface tracking methods. Left: Level set. Middle: Particle level set. Right: Mesh-based tracker with local grid-based topology changes.

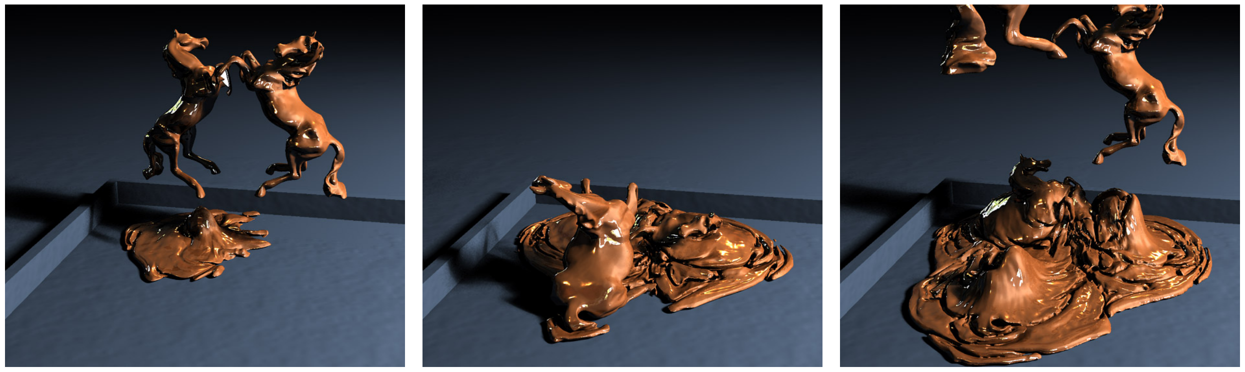


Figure 36: These images from an animation show viscoelastic horses being dropped onto one another. Many topological merges occur, yet details of the surfaces are kept.

Robustness

On rare occasions, numerical errors due to inexact arithmetic can prevent us from cleanly sewing together surfaces and producing a watertight surface mesh. In such situations, we simply add the surrounding cells to the existing list of marked cells and repeat the flood fill steps in Section 5.4.2, replacing the incorrect geometry with the topologically simple isosurface. This correction step typically adds one or two complex cells and re-samples slightly more surface geometry as a result.

5.5 Preserving thin sheets with local convex hulls

As mentioned in Section 5.3, marching cubes templates are not enough to preserve thin features in a fluid simulation. As a result, the method described in the previous section cannot preserve thin sheets and strands of liquid. However, we can fix this by using more general local grid connectivity, similar to the modified marching cubes tables described in Section 5.3.2. In this section, we generalize these marching cubes tables by using a convex

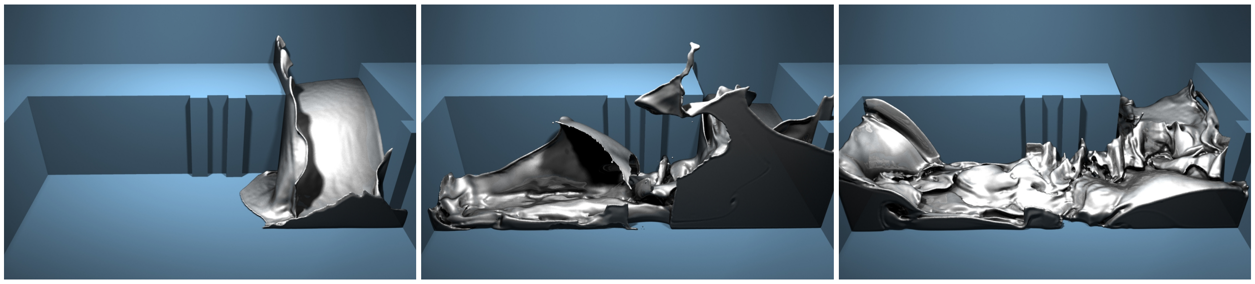


Figure 37: This algorithm efficiently produces detailed thin sheets and liquid droplets, even with low-resolution fluid simulations — The main corridor in this example is only 30 fluid cells wide.

hull algorithm to automatically compute new surfaces on the fly, and we use a different topological argument to ensure that we can sew together our surfaces at the cell boundaries. Essentially, we use the same algorithm as defined in Section 5.4, but we redefine how to detect complex cells and reconnect the surface.

In this section, the main goal of changing the topology of the mesh is to ensure that the liquid surface is connected to other regions of fluid in the same way as the pressure values. If these topologies differ, then distracting visual artifacts will occur. For example, if two disconnected surface components lie within the same cell in the fluid grid, then the fluid pressure values will be unable to distinguish between these independent components. The low resolution fluid velocities will then move both surface pieces together in the same direction, creating an invisible link between them that tends to persist for the entirety of the simulation. We want to detect these situations and get rid of them by changing surface topology.

5.5.1 Defining valid topology

We detect disagreements between the explicit surface mesh and the fluid grid by locally contrasting the topology of the explicit surface with a surface that possesses the same topology as the fluid simulation. We define any region where these topologies disagree as *topologically invalid* (note we are creating a direct analogy to the term "topologically complex" from the Section 5.4.2).

Before specifying what it means for surface geometry to be topologically valid, we first define a topological cell as a cube with its eight corners located at sample points in the signed distance function (the corners are co-located with the fluid pressure values). The cube is bounded by six faces, twelve edges, and eight corners. We can better understand the topology of the explicit surface mesh by examining the intersection between the the solid geometries of the triangle mesh and each topological cell. This intersection is empty if the cell is completely outside of the surface, the intersection is identical to the original cube if the cell is completely inside the surface, and the intersection is more complicated if the cell overlaps the surface.

In order for the cell to be topologically valid, the surface of this intersection should

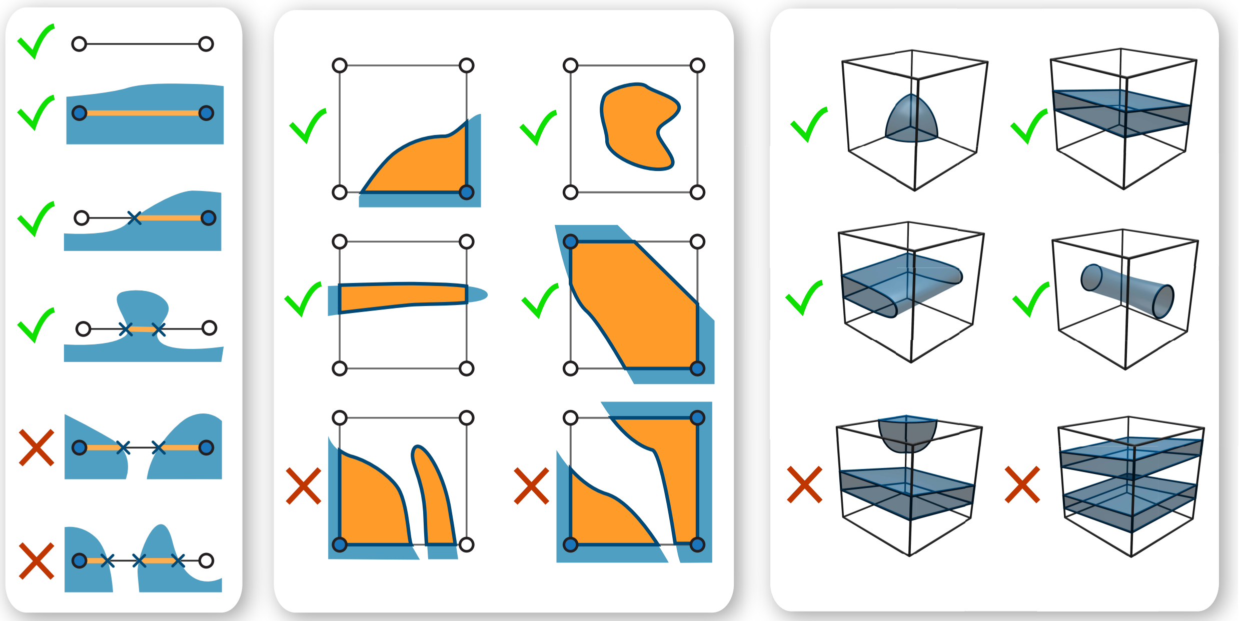


Figure 38: Examples of valid (green check mark) and invalid (red X) topology for edges, faces, and cells.

have the same connectivity with its neighbors as the fluid cell does, and it must not have any more topological features than a single fluid cell. That is, it must contain at most one surface component, with no holes or voids — the surface of this intersection should be homeomorphic to a sphere (more formally, a 2-sphere). Similarly, the transitions from this topological cell to its neighbors must be topologically valid, so the surface intersections with faces and edges of this cell should be homeomorphic to 1-spheres (circles) and 0-spheres (intervals), respectively. Lastly, a corner of the cell is topologically valid if it is not located in an inside-out region of the surface. Figure 38 shows some examples of valid and invalid geometry.

We can efficiently detect the topological validity of a cell corner by checking its counter from the signed distance function calculation (Figure 19). The counter of a valid corner must be either zero or one, otherwise it means the surface is inside out or self-intersecting. Next, we determine the validity of a cell edge by counting the number and orientation of its intersections with triangles in the surface mesh and comparing the result with a single line segment with outward-oriented endpoints (0-sphere). If there are too many components, or if the surface is oriented the wrong way at any of the intersection points, then the edge is invalid. This step is performed at the same time as the corner validity test (during the creation of the signed distance field). We can check the validity of a face by computing its intersection with the mesh and counting the number of components, and we can test the topological status of a cell by similarly counting connected components and ensuring that the Euler characteristic detects no holes.

The corner validity test samples the signed distance function at several regularly-spaced sample points, and it is guaranteed to identify any self-intersections larger than the grid spacing. This test will catch any topological flaws that are well-resolved in the x, y, and z dimensions, just like marching cubes will faithfully reproduce a surface as long as there are no features smaller than the grid size. The edge validity test checks all of the edges in the grid, so it is guaranteed to identify any topological flaws that are well-resolved in at least two dimensions. This means that self-intersections and pockets of air that look like thin sheets will be identified by the edge validity test. This test can also catch thin spindles and voids if they happen to intersect one of the grid edges. The face validity test checks for intersections with all faces in the grid, so it is guaranteed to catch topological flaws that look like thin spindles (which span more than one cell in a single dimension, but are very thin in the other two). However, the face validity test cannot catch topological flaws smaller than a single grid cell unless they happen to intersect the face. Finally, the cell validity test is guaranteed to identify all topological flaws, because it checks within every cell.

5.5.2 Topological detection in practice

In practice, such topological problems smaller than a grid cell rarely ever occur, because we start with a well-resolved surface and then smoothly deform it according to a low resolution fluid velocity field. This means that large features morph into small features through a gradual process, by first becoming thin sheets and then thin spindles. Because topological inconsistencies are caught by lower-dimensional validity tests before they have time to shrink smaller than a grid cell, we have not found it necessary to perform the full cell validity test.

Face validity tests are mostly redundant for our purposes as well, because thin structures eventually intersect cell edges after some perturbations from the fluid simulation — the edge and corner validity tests tend to quickly catch all problems. As a result, we have found it practical to bypass the testing of any cells and faces unless we specifically have to guarantee valid topology in a particular region, as we will describe shortly.

To summarize the work done in practice by our implementation, we only perform the topological validity tests on the corners and edges for every corner and edge in the fluid grid.

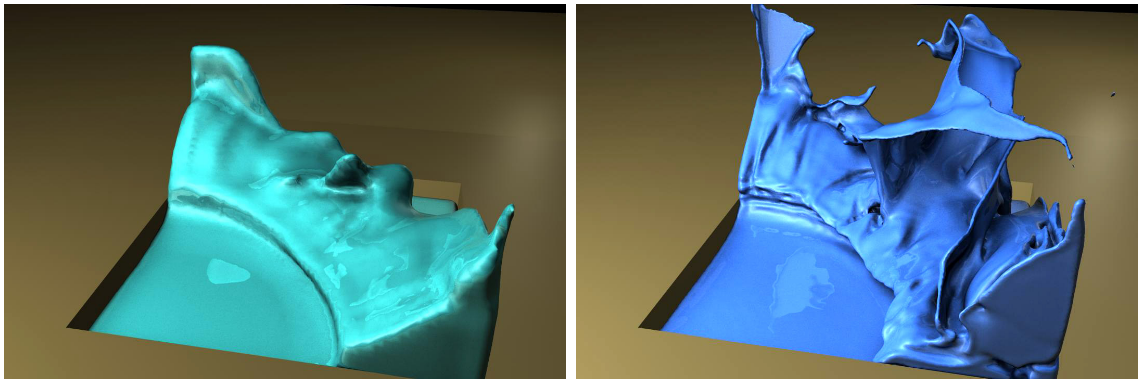


Figure 39: Comparison between a level set surface tracker with 2nd order advection (left) and a mesh-based surface tracker that preserves thin sheets (right) on a 60^3 grid.

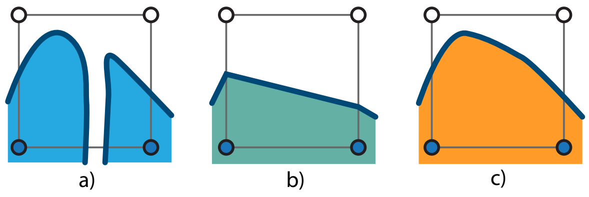


Figure 40: Given a surface with invalid topology (a), methods based on marching cubes-style lookup tables will ignore interior vertices (b), while our method preserves many of the original details (c).

That is, we check all corners to see whether they represent inside-out geometry, and we check all edges to see if the surface of their solid intersection is homeomorphic to a 0-sphere. To optimize our implementation at the expense of missing some small geometry within a single cell, we do *not* perform topological validity tests *within any cells*. The test to guarantee that the surface of the solid geometry is homeomorphic to a 2-sphere requires computing the boolean intersection between a cell and the mesh, counting the number of holes, and then counting the number of components. Although this test would help guarantee that all of our geometry is simple enough to be represented by a fluid cell, it does not catastrophically break our algorithm if we neglect it either. As a further optimization, we do *not* test every *face* for topological validity either. Because this test is not trivial (it requires intersecting the mesh with a face and counting the number of connected components in this intersection), we only perform it at the boundary between topologically valid and topologically invalid cells, as explained in the next paragraph.

After an invalid cell is detected, we will soon replace its intersecting surface with a similar surface that has topologically valid geometry. Because this cell shares its boundaries with other cells that may not have been classified as topologically invalid, we have to specifically guarantee the validity of its faces. In this case, we count the face components and pass topological validity information to neighboring cells, similar to the complex cell propagation strategy in Section 5.4.2 [69]. To summarize the frequency of topological tests in our implementation: we exhaustively test every corner and edge in the signed distance grid, we never check cells, and we only check faces when it is absolutely necessary to ensure valid connectivity between an invalid cell and its topologically valid neighbors.

5.5.3 Correcting invalid topology

After deciding that a cell has invalid topology, we must replace the surface/cube intersection in that cell with a new one. We require that this new intersection surface meets two constraints: it should be topologically valid, and it should cleanly connect with the original surface. In addition, because each vertex of the original surface represents a valuable piece of Lagrangian simulation data, we also desire that the new surface preserves as many of the original surface vertices as possible.

According to our definition of topological validity, a valid intersection must have a surface which is either either empty or homeomorphic to a 2-sphere, its intersection with each

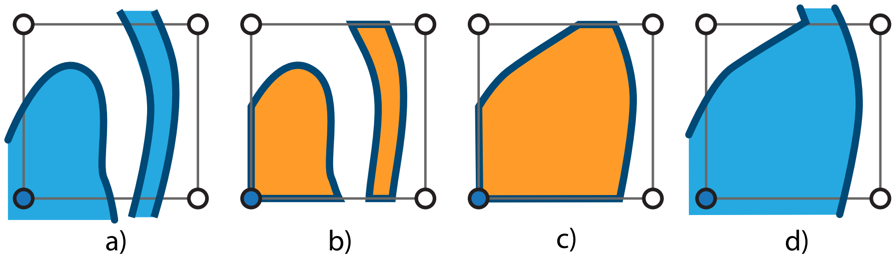


Figure 41: Given a triangle mesh and topological cell (a), we examine the intersection between the the solid geometries of the mesh and the cell (b). If the intersection has invalid topology, we replace it with its convex hull (c). Finally, we remove facets belonging to the boundary of the cell and reconnect the surface (d). Note that the cell below this one will also be re-sampled, because the bottom edge in (a) is topologically invalid.

cell face must either be empty or have a boundary that is homeomorphic to a 1-sphere, and its intersection with each cell edge must be either empty or have a boundary which is homeomorphic to a 0-sphere. A straightforward strategy for reducing the geometric complexity of a surface/cube intersection is to wrap a single surface around all of the original surface components. Fortunately, this new surface can be created efficiently by replacing the original surface/cube intersection with its convex hull. If the input surface intersected the cell boundary, then the convex hull preserves this connectivity. Furthermore, the convex hull's intersections with faces and edges are lower dimensional convex hulls, so they are also topologically valid. In addition, all of the vertices on this convex hull are preserved from the input surface, so we reduce re-sampling errors during this process, as illustrated in Figure 40.

In practice, we use the qhull library [2] to compute the convex hull of the following points: all original surface vertices that lie within that cell, the vertices created by intersecting the original mesh with the edges and faces of the cell (the *type 1* and *type 2* vertices described in Section 5.4.3), and all "inside" corners of the cell. Finally, because the convex hull represents the intersection between a cube and the new surface, we extract the final surface by deleting any facets that are co-planar with the cell boundary. See Figure 41 for an example.

This strategy of reducing invalid surface regions to topological spheres has physical implications when used in a fluid simulation: (1) thin sheets of air will be detected as topologically invalid and destroyed, and (2) thin sheets and strands of liquid will persist throughout the simulation. We will discuss how to further control the final surface topology in the next section.

5.5.4 Topological control

The method presented in Section 5.5.3 will preserve all thin sheets and spindles unless there are significant self-intersections. In addition, we found it useful to manually perform topological changes to the surface by explicitly cutting off exceptionally thin spindles of liquid. We perform the explicit mesh separation operation explained in Section 5.2.2 from

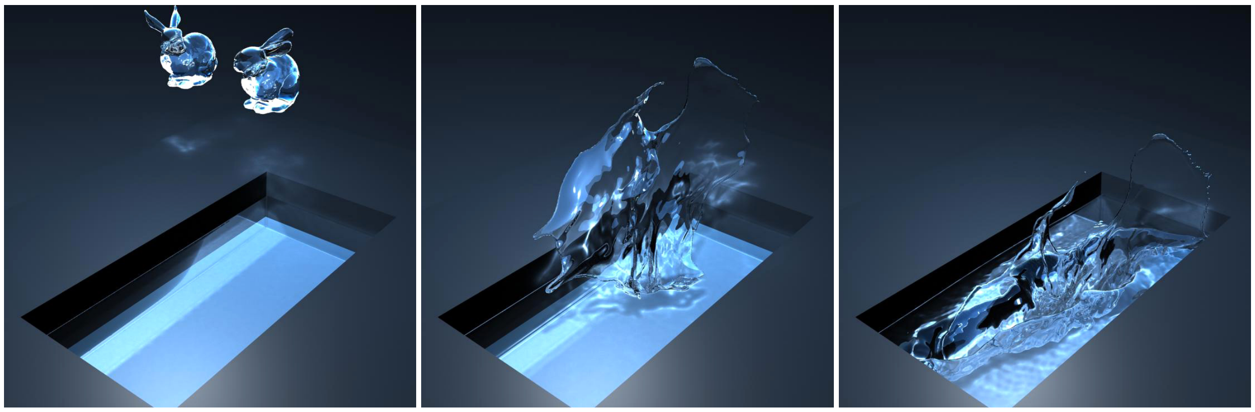


Figure 42: Two bunnies splatter into each other, producing a thin liquid sheet that merges with the pool of water below.

Wojtan et al. 2010 [70].

As explained in [67], thin spindles of liquid consistently lead to topological changes in real-world liquids. Due to a surface-tension-based Rayleigh-Plateau instability, the liquid spindle rapidly collapses until it is extremely thin and then breaks in two. This explicit cutting operation precisely mimics this behavior in the discrete setting by separating the thinnest possible unit of a discrete surface. This cutting method is also quite efficient — it is detected for free during standard surface maintenance operations, and it requires about as much work as a single edge collapse.

5.5.5 Sewing meshes together

In order to sew together the convex hull surface with the original surface outside of the topologically invalid region, we follow the exact same steps in Section 5.4.3. However, instead of replacing the surface with marching cubes templates in step 4, we replace the surface with a convex hull as described in Section 5.5.3.

5.5.6 Smooth surface interpolation

Both marching cubes-style lookup tables as well as our convex hull algorithm in Section 5.5.3 use piecewise linear surfaces to connect vertices together. These low-order connecting surfaces are indistinguishable from the original surface mesh when the triangles in the surface mesh are about the same size as a fluid grid cell. However, our method allows the surface mesh to have a much higher resolution than the simulation grid, so the triangles output by our convex hull will often seem disproportionately large. To maintain a detailed surface with many small triangles, we repeatedly subdivide these large triangles until their edges are as short as the maximum edge length allowed in our surface mesh. These subdivisions create new vertices, and we are free to place them wherever we like.

We used butterfly subdivision when placing our new vertices, similar to [12]. We found that the modified scheme of Zorin et al. [75] worked best, because a significant portion of the

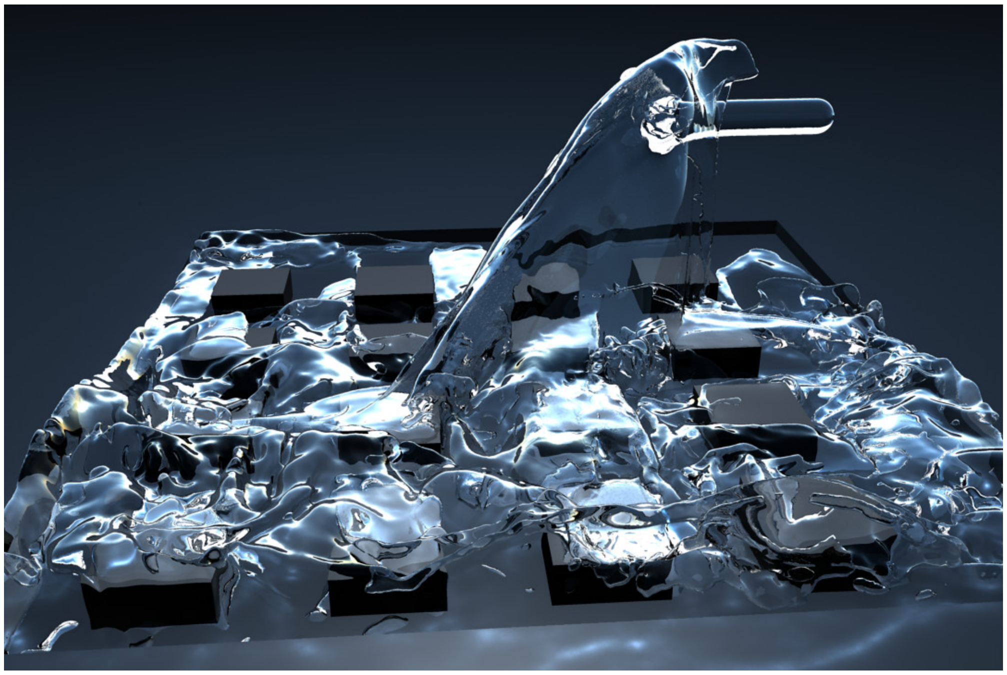


Figure 43: Two streams of water collide, filling a box with liquid.

vertices in our simulations did not have a valence of six. After applying this interpolating subdivision to the triangles output by our topological correction algorithm, the surfaces exhibit both correct topology and high degrees of smoothness.

5.5.7 Input parameters

This algorithm has relatively few parameters to set. The topological detection depends on the grid size, which is identical to the fluid grid resolution in this section. We are free to change the topological resolution if we like, but then we lose any guarantees about topological validity if the topological grid is not the dual of the fluid simulation grid. The main tool for fixing topology in this section is the convex hull, which does not depend on any input parameters.

The topological algorithm is unconditionally stable, so it does not depend on any minimum time step size. Some degenerate configurations like completely flat shapes can cause errors when sewing together the convex hull to the rest of the surface, but this problem can be solved by jittering vertices or by more intelligent collision resolution.

The explicit mesh cutting procedure described in Section 5.5.4 requires the animator to specify the maximum allowed edge length in order to force the breakup of thin liquid spindles.

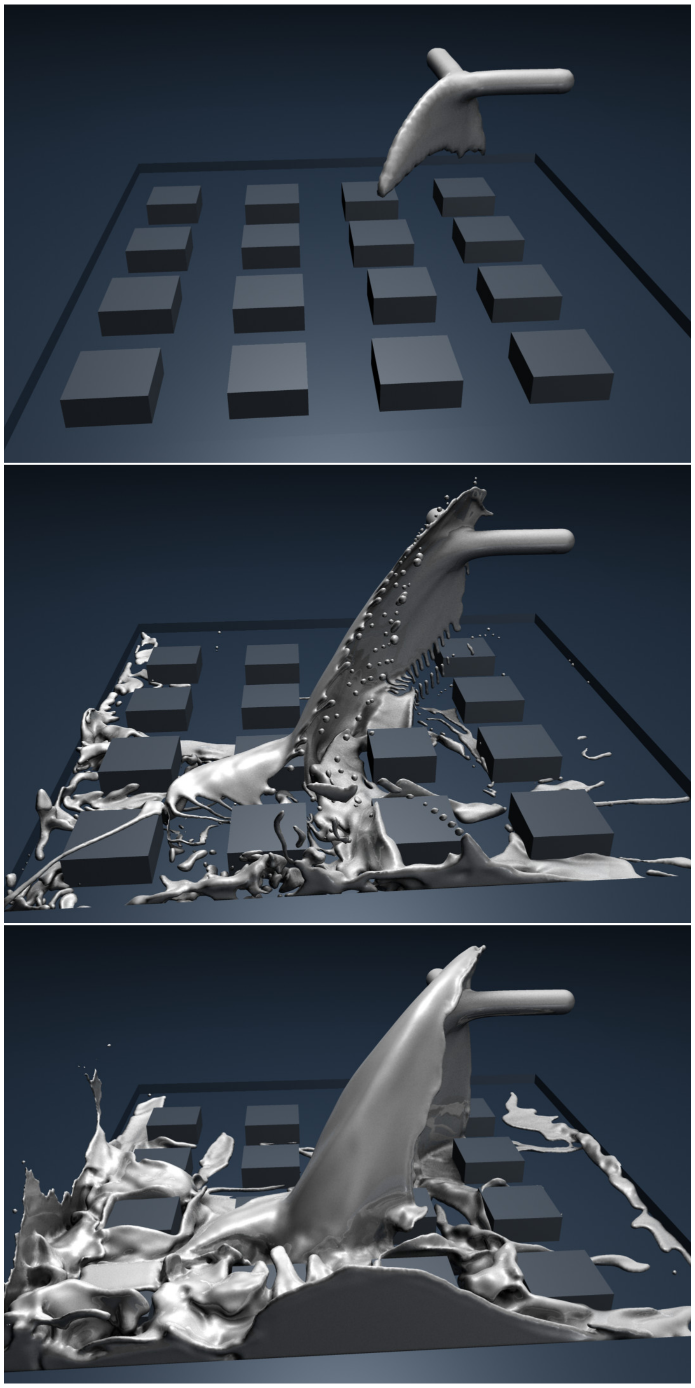


Figure 44: In this example, the level set method (left) deletes thin sheets before they even touch the ground. The method from Section 5.4 (middle) re-samples the surface from a grid during topological changes, so it cannot merge together multiple thin sheets without deleting large portions of the surface. The method of Section 5.5 (right) merges together thin sheets without rampant thin feature deletion.

6 Advantages of a mesh surface

In the previous sections, we explained how to integrate a deforming surface mesh into a fluid simulation while robustly handling changes in the surface topology. This mesh-based surface tracking strategy gives us a great deal of advantages over previous methods. In this section, we will discuss these benefits in detail.

6.1 Color and texture tracking

When using fluid simulations for visual effects and animation, it may be desirable to attach a renderable texture to the surface mesh. This is a difficult problem when using implicit surfaces, as the texture information must be carried on the grid and interpolated onto the surface at render time. This approach incurs smoothing due to numerical diffusion and interpolation (although work by Bargteil et al. has mitigated these issues for grid-based surfaces [3]).

For deforming mesh surfaces with unchanging connectivity, maintaining texture information per vertex is trivial. For surfaces undergoing remeshing and changes in topology, we require a method of assigning texture information to new vertices. For an edge split operation, we could simply interpolate texture values from nearby vertices for the newly created vertex (for example, taking the average of the values on the two edge end points, or the four new vertex neighbours).

A more complicated problem occurs when using a surface reconstruction method for dealing with topology changes [45, 69], since vertices may be added without any neighbours immediately available. Müller proposes a solution to this, noting that each vertex of the new mesh is created by an intersection of a grid edge with a triangle [45]. We can 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 any more. In that case we choose an arbitrary triangle. Figure 45 shows the result of applying this approach in a practical system.

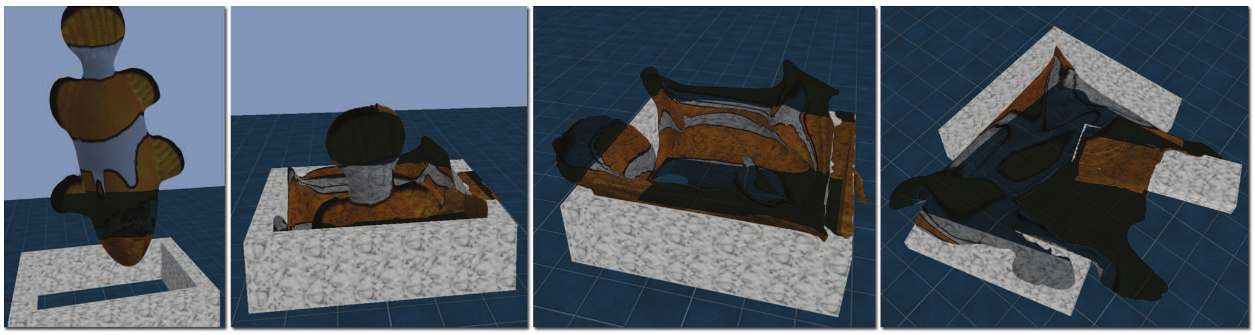


Figure 45: Advecting surface texture information

6.2 Preserved surface details

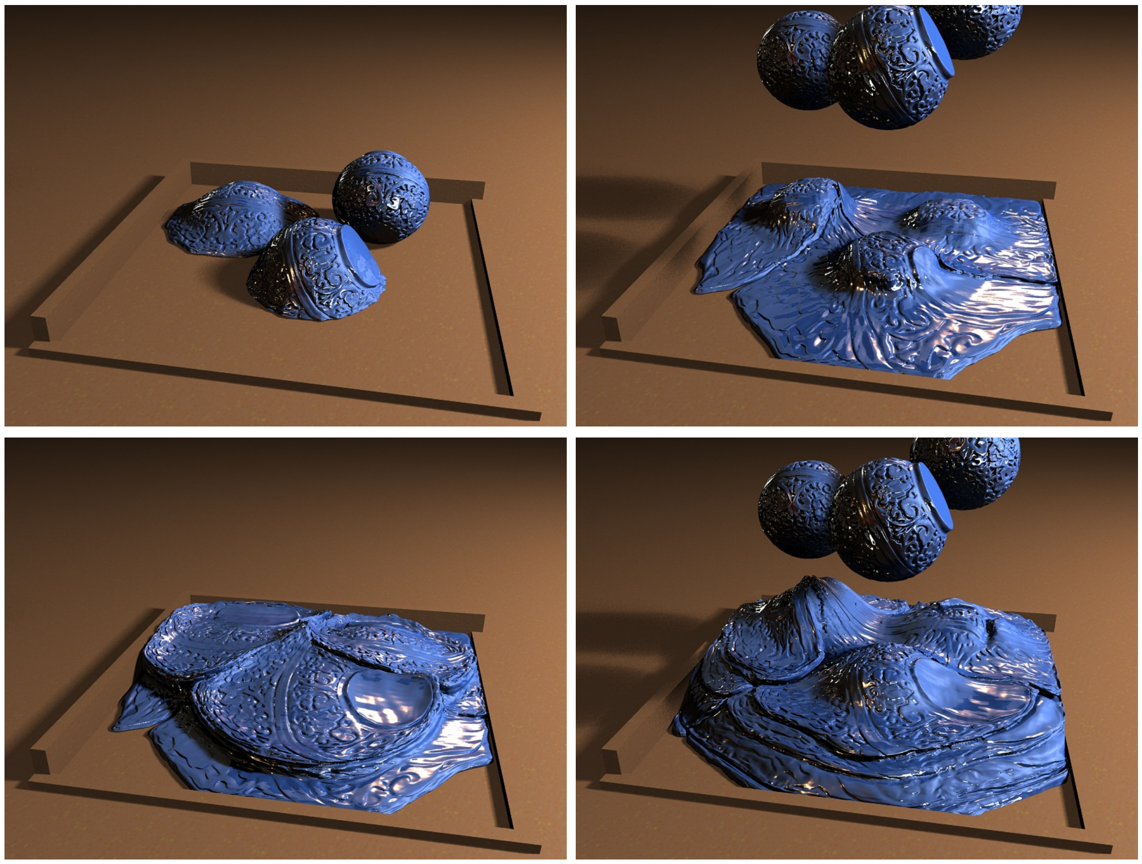


Figure 46: Dropping viscoelastic balls in an Eulerian fluid simulation. Invisible geometry is quickly deleted, while the visible surfaces retain their details even after translating through the air and splashing on the ground.

Besides maintaining renderable texture information, high-resolution explicit surfaces can also maintain geometric detail that would quickly be smoothed out by an implicit surface representation. Figure 46 shows several frames from an animation produced by using explicit surface tracking in a grid-based fluid simulation. In this particular surface tracking approach (due to Wojtan et al. [69]), the surface is only regularized by reconstruction where topology changes occur. Surface patches which do not merge are left alone, allowing the rest of the surface to maintain these fine-scale features.

6.3 Thin features

One of the most compelling reasons for choosing an explicit triangle mesh over a traditional level set is that triangle meshes can capture thin surface features much more easily. For example, numerical dissipation in level set methods smooths out high curvature regions and can cause parts of the surface to vanish, even under rigid motion (recall the Zalesak sphere test in figure 8). In fact, very thin yet smooth (low curvature) surfaces, such as sheets of liquid, require excessively refined grids to avoid this problem.

As a quick example, consider the *Enright test*, which is often used to test the accuracy of surface tracking methods [19]. A sphere is advected through a smooth, periodic velocity field, which stretches the volume to a thin sheet and back into a sphere (see figure 47). The volume is measured at the start and end of one period, and the difference in volume is taken as a rough estimate of the surface tracking quality. When using a regular grid, the surface begins to disappear as its thickness falls below the grid resolution. Using an unmodified level set, the original paper by Enright et al. reported a volume loss of 80%, even using a moderately high-resolution grid (100^3). By contrast, an implementation of explicit, mesh-based surface tracking undergoing the same test reports a volume loss of less than 1% [12].

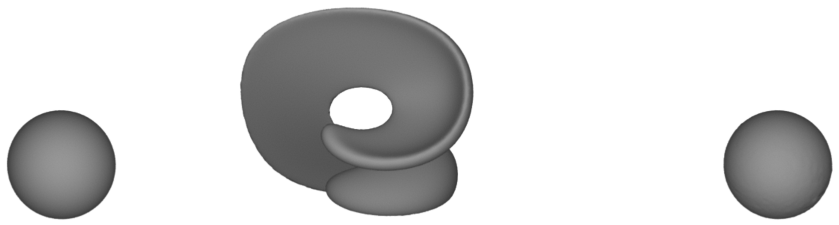


Figure 47: Running the Enright test on an adaptive triangle mesh surface. Volume is preserved even when stretching into a very thin sheet.

A more extreme extreme example of thin feature preservation in a practical fluid animation system is shown in figure 48.

6.3.1 Thin features and robust topology changes

Though it offers many benefits, the presence of thin features also presents a set of new challenges. To deal with topological changes, several methods will temporarily construct an implicit surface representation on a regular grid, then extract a new mesh surface from this implicit surface [15, 3, 45, 69]. This can be done globally, or only locally around inconsistent surface patches. This approach results in consistent surfaces and clean topology changes, however it carries some of the disadvantages of implicit surface tracking. In particular, thin features in the surface may not be registered on the grid, and so will be lost in reconstruction if care is not taken. Much research effort over the past few years has been devoted to preserving thin features while using these approaches [45, 69, 70].

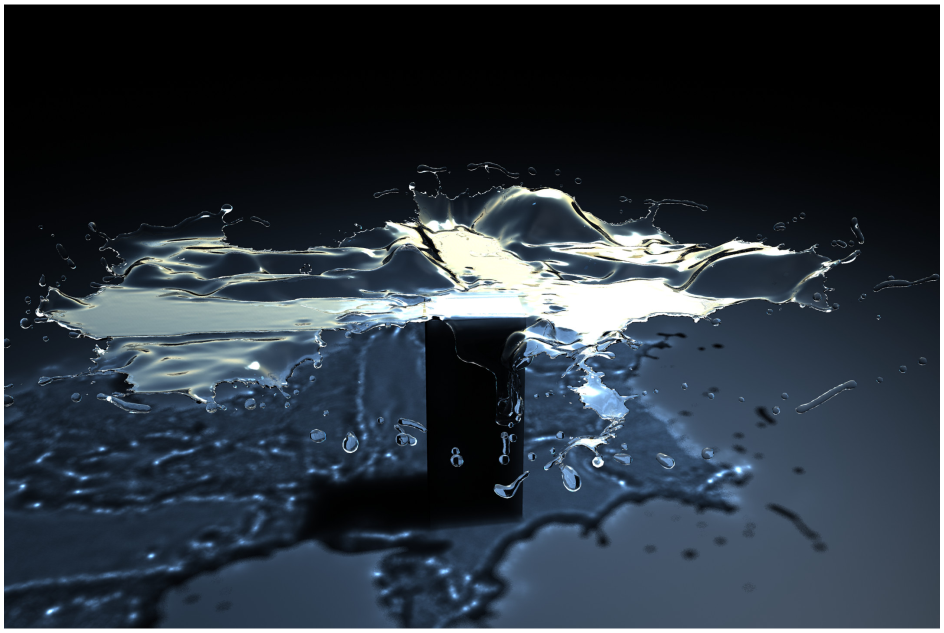


Figure 48: Thin sheets, tendrils, and droplets are challenging to maintain with a grid-based surface representation, but can be easily captured with explicit surfaces.

6.3.2 Resolution mismatch

When integrating an explicit surface tracking method into a fluid simulation, another difficulty arises: the mismatch between the resolution of the fluid simulation and that of the surface. When the surface is coupled to a standard Eulerian simulator, the liquid volume must first be resampled onto the simulation mesh or grid to provide geometric information for boundary conditions. As this resampling process typically destroys small details, they are invisible to the fluid solver and cannot be advanced appropriately. This can lead to a variety of visible artifacts including lingering surface noise, liquid behaving as if it were connected when it is not (and vice versa), and thin features simply halting in mid-air because the simulator fails to see them. When combined with surface tension forces, noisy sub-mesh details can also severely hamper stability if they are not artificially smoothed out (see section 6.6).

Dealing with sub-grid geometry is one of the most critical issues when using explicit surface tracking in fluid simulation. Different researchers have approached this problem in different ways.

Make the surface match the simulation: To handle changes in topology, Wojtan et al. [69] use local implicit reconstruction, which tends to regularize the surface mesh and smooth out features below the scale of the grid. Similarly, Müller [45] uses a regular grid to construct a new mesh from a given mesh at each time step, using a Marching Cubes-like alogrithm. In these approaches, if the implicit surface grid matches the simulation grid, then the output of the reconstruction will be resolvable by the simulation. Extensions of this idea allow for more sub-grid features than would be allowed by applying traditional Marching Cubes during reconstruction [45, 70], but there is always some limit on the complexity of the topology within each cell. This complexity limit is what allows the regular grid simulation to still be so effective, as demonstrated in these papers. This idea has been explored in the course up to this point.

Make the simulation match the surface: Brochu et al. [11] come at the problem from another direction, and instead attempt to capture the surface geometry directly by adaptively placing simulation nodes around the surface vertices. The main advantage of this approach is that it treats all surface geometry with the same physical rules, and that it can resolve fine details and complex topologies. The big disadvantages are that it requires an irregular simulation mesh, so algorithms designed for regular grids need to be re-thought, and it involves expensive Delaunay meshing of the simulation domain at each time step. This approach will be described in more detail in section 6.4.

Run another simulation on the sub-grid geometry: An idea which could complement either of these approaches is to separate grid-scale physics from the sub-grid-scale geometry. The usual Eulerian fluid simulation runs on the simulation grid, while a small-scale surface tension or wave simulation runs using the surface polygons as simulation elements. This approach was most fully realized in work by Thürey et al. [64]. This has the effect of regularizing surface noise, as well as producing plausible motion for fine surface features which do not show up on the grid. This separation becomes crucial when dealing with surface tension, as we will soon see.

6.4 Matching the simulation to the surface

In this course we have discussed how to incorporate explicit surface tracking into an Eulerian, regular-grid fluid simulation. Although regular grid sims are very popular, unstructured and semi-structured meshes have a long history in computational fluid dynamics, and have gained traction in computer animation as well. In this section we will see how to construct a simulation that captures every detail of an explicit surface using an unstructured mesh.

The key idea of this scheme is to carefully generate sample points near the liquid surface. These points will be the locations where pressure values are stored during the projection step. We then build a Voronoi simulation mesh from these points and a background lattice, and apply a ghost fluid/finite volume pressure discretization which captures the precise position of the liquid interface. Coupling this with a standard semi-Lagrangian velocity advection scheme and surface tension model completes the fluid simulator. We will discuss the adaptive pressure sampling and pressure projection step using Voronoi elements in this section. More details are available in the original paper [11].

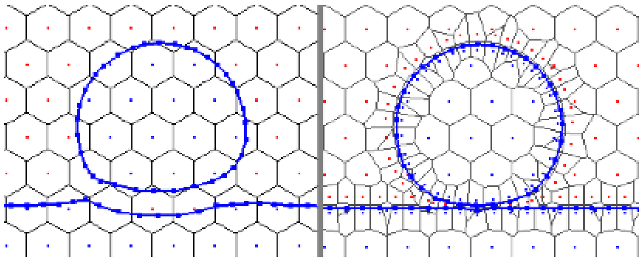


Figure 49: **Merging.** Left: Regular sampling erroneously identifies a topology change, causing a premature reaction in both liquid bodies. Right: Geometry-aware sampling responds correctly.

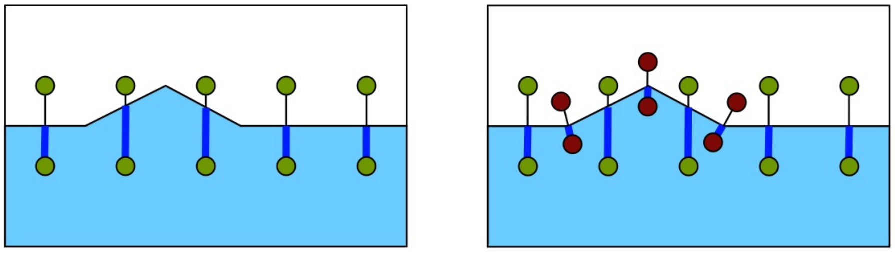


Figure 50: Left: Even with the ghost fluid method, regular sampling may miss surface details which do not align with the simulation mesh, such as this wave crest. Right: Adaptive samples (shown in red) placed on either side of each mesh vertex ensure that all geometric detail is resolved by the simulation.

6.4.1 Adaptive pressure sampling

Careful pressure sample placement with respect to the surface helps in three important ways. First, we can inform the solver of all local geometric extrema, allowing the physics to act upon them correctly. This eliminates the accumulation of erroneous surface noise without requiring non-physical smoothing; this is especially vital for incorporating surface tension. Second, we can ensure that the solver sees the correct surface *topology* so that the physics responds to merging or splitting only when the surface mesh itself merges or splits (see figure 49). Lastly, grid-scale features often disappear and reappear in regular grid sampling, from the perspective of the solver, as the surface translates through the grid. By specifically placing points inside such small features, we ensure they cannot be missed.

We begin by choosing a characteristic length scale for the simulation, Δx, and configure our explicit surface tracking library to try to maintain triangle edge lengths in the range $[\frac{1}{2}\Delta x, \frac{3}{2}\Delta x]$. To resolve all surface details with our volumetric mesh, we need to place pressure samples so that they capture the surface's local geometric extrema, i.e. around surface mesh vertices. In particular, we try to ensure that one edge of the Delaunay triangulation passes through each surface vertex, with one sample inside and one outside. Therefore we take

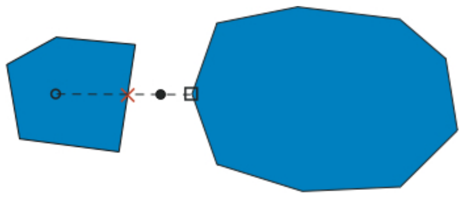


Figure 51: **Sampling Thin Features.** A pressure sample is seeded along the outward normal direction from a surface vertex (black square). The initial proposed pressure location (empty black circle) would land in the wrong component and potentially fail to resolve the intervening air gap. We instead place the final pressure sample (filled black circle) midway between the starting vertex and the first intersection point (red X).

the inward and outward normal at each surface vertex (averaged from the incident surface triangles), and attempt to place a pressure sample a short distance along each (see figure 50). We placed outward samples at $\frac{1}{2}\Delta x$ and inner samples at $\frac{1}{4}\Delta x$, though other ratios would work as well. As a result, surface mesh normal directions will often align exactly with a velocity sample in the simulation mesh; this lends additional accuracy to the vertex's normal motion, and to the incorporation of the normal force due to surface tension calculated at the vertex.

This placement may miss very thin sheets or other fine structures: to robustly sample such features, we check line segments of length Δx from each surface vertex in both offset directions for intersection with the rest of the surface mesh. If we find any triangle closer than Δx, we store the distance d to the closest intersection, and use d in place of Δx in the offset distance calculations above (see figure 51). We further reject new pressure samples which are too close to an existing sample by some epsilon, which would cause a very short edge in the final mesh.

If the distance between the surface vertex and the first intersection is below some threshold (e.g. $\frac{1}{20}\Delta x$) at which we consider the two surfaces to have effectively collided, and the proposed sample is an air sample, we also discard it. This is necessary because the divergence constraint is not enforced on air cells, so they can act as liquid sinks [40] and destroy liquid volume until the geometry finally merges. Unfortunately, merging in this scenario can often take several time steps to resolve because the interpolated velocity in the air gap still averages to zero, thereby preventing surface geometry from actually intersecting and flagging a collision. By not placing a sample point in these very small gaps, our simulator treats the two liquid bodies as merged and prevents volume loss; the geometric merge is usually then processed within a few timesteps. (With regular sampling, merging will depend on where grid points happen to fall with respect to the surface; hence the physics can respond as if merged when the surfaces are still as much as Δx apart, as in figure 49. This generates non-physical air bubbles which linger for many timesteps before they self-collide and are eliminated.)

After placing the surface-adapted pressure samples, we complete the sampling of the domain by adding regularly-spaced points from a BCC lattice with cell size $2\Delta x$, again

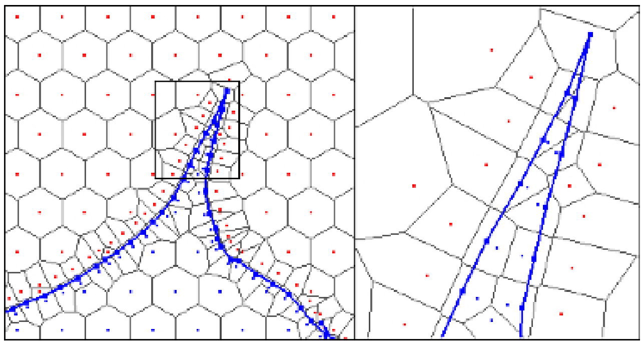


Figure 52: **Simulating Thin Features.** A 2D example of a thin feature simulated with our method. The zoom on the right illustrates the sample placement with respect to surface vertices, and the resulting Voronoi mesh. Notice that even the very sharp tip contains a pressure sample, as indicated by the surrounding Voronoi cell.

rejecting samples which fall too near existing samples—of course, a graded octree or any other strategy could also be used to fill the domain. All samples are then run through a Delaunay mesh generator such as *TetGen* [59]. Figure 52 illustrates in 2D how this sampling approach is able to capture thin features such as splashes. Further experimentation with relative mesh spacing parameters could yield improved results.

6.4.2 Pressure projection with embedded boundaries

We use a finite volume method on a Voronoi mesh for the pressure projection step, similar to Sin et al. [60]. However, rather than applying boundary conditions as they describe, we adapt the embedded boundary methods of Batty et al. [5] to Voronoi meshes. Conveniently, the duality/orthogonality relationship between Voronoi and Delaunay meshes lets the accuracy benefits of the method carry over. Figure 53 illustrates our mesh configuration, and the computation of the required weights, as discussed below. We solve the resulting symmetric positive definite linear system using incomplete Cholesky-preconditioned Conjugate Gradient.

To enforce embedded solid boundary conditions, we need to estimate the partial unobstructed area of each element face (figure 53(d)). Batty et al. [5] used marching triangles cases for computing tetrahedra face fractions from signed distance values on the vertices. However, in the Voronoi setting, the faces are arbitrary convex planar polygons rather than triangles. To handle this, we temporarily place an extra vertex at the face centroid, and use it to triangulate the face. We then use signed distance estimates at the vertices to compute each sub-triangle's partial area, and sum them to determine the partial area for the complete face.

The embedded (ghost fluid) free surface condition uses signed distance estimates at pressure samples to estimate the surface position; these are now located at Voronoi sites rather than tetrahedra circumcenters, but the method is otherwise unchanged (figure 53(c)). A slight improvement can be achieved by casting rays to find the exact position of the surface

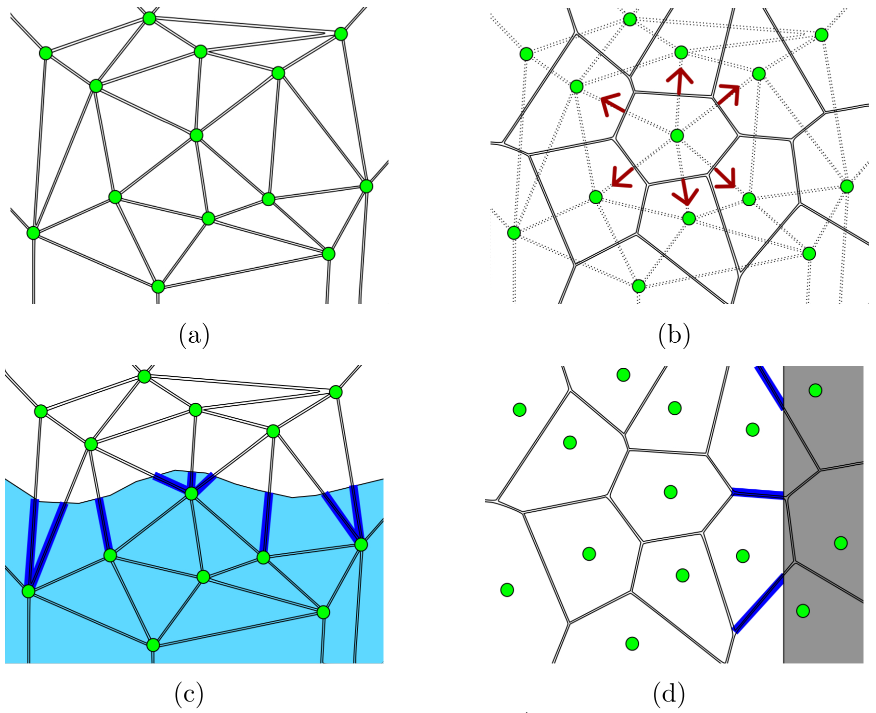


Figure 53: **Embedded boundaries on Voronoi/Delaunay meshes.** Pressure samples are shown as green circles. (a) Delaunay triangulation. (b) Voronoi diagram dual to the Delaunay triangulation (velocity components for the central cell are shown as red arrows). (c) Computation of ghost fluid weights on the edges of the triangulation. (d) Computation of non-solid weights on the faces of the Voronoi diagram. In 2D, Voronoi faces are simply line segments, so solid weights are just fractions of segment lengths. In 3D, Voronoi faces are convex polygons, so determining non-solid weights involves computing polygon areas.

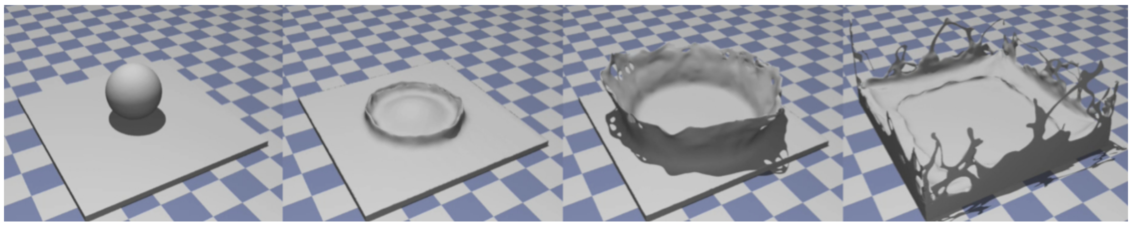


Figure 54: Coupling an explicit surface tracker to a Voronoi simulation mesh built from pressure points sampled in a geometry-aware fashion lets us capture very fine details in this "sphere splash" animation.

mesh between pressure samples. In some cases this is much more accurate than the estimate derived from signed distances, but in practice we found it made minimal visual difference. To actually compute the liquid signed distance field on the tetrahedral mesh, we compute exact geometric distance for a narrow band of tetrahedra near the surface, then use a graph-based propagation of closest triangle indices to roughly fill in the rest of the mesh. This family of redistancing schemes is described by Bridson [8], and is easily adapted to tetrahedra.

Several frames from a complete implementation of this approach is shown in figure 54.

6.5 Volume correction

As mentioned above, an important characteristic of any good surface tracking package is its ability to preserve volume. An additional side benefit of using an explicit surface is that the volume enclosed by a surface can be computed accurately and easily. To take advantage of this, a frequently-used correction is to maintain a target volume for each volume of fluid, and as the system gradually loses or gains volume, perturb the surface slightly to drive it towards this target volume.

To compute the volume of fluid, first notice that the integral of 1 over a closed domain is equal to the total volume of that domain:

$$\text{volume}(\Omega) = \int_\Omega 1 \; dx$$

We transform this volume integral into a surface integral using the Gauss divergence theorem, which can be written as:

$$\int_\Omega \nabla \cdot \vec{F} \; dV = \int_{\partial\Omega} \vec{F} \cdot \hat{n} \; dS$$

To apply this theorem, we specify a function whose divergence equals 1, for example:

$$\begin{aligned} \vec{F}(x) &= \frac{\vec{x}}{3} \\ \nabla \cdot \vec{F}(x) &= 1 \end{aligned}$$

We can now apply the divergence theorem to get a formula for volume in terms of a surface integral:

$$\text{volume}(\Omega) = \int_\Omega 1 \; dV = \int_\Omega \nabla \cdot \frac{\vec{x}}{3} \; dV = \int_{\partial\Omega} \frac{\vec{x}}{3} \cdot \hat{n} \; dS$$

Thus we have a method for computing volume using only surface positions and normals. Since we are dealing with a triangulated surface, we can compute this integral exactly, as:

$$\text{volume}(\Omega) = \frac{1}{6} \sum_{t}^{n_{\text{triangles}}} (\mathbf{x}_{t1} \times \mathbf{x}_{t2}) \cdot \mathbf{x}_{t3},$$

where $\mathbf{x}_{tj}$ is vertex j in triangle t. To drive the surface from its current volume V to a target volume V_0, we can apply an offset in the normal direction at each vertex, $\mathbf{x}_i$:

$$\begin{aligned} d &= (V_0 - V)/A \\ \mathbf{x}_i &= \mathbf{x}_i + d\mathbf{n}_i \end{aligned}$$

where A is the surface area [10, 45]. This is admittedly non-physical, but is a convenient way to counter drift in volume due to numerical error.

Thürey et al. use this idea in an iterative approach, recomputing V and updating the mesh vertex locations until the relative error $\varepsilon = (V_0 - V)/V_0$ is less than some user-defined threshold [64].

6.6 Surface tension

Recall the Navier-Stokes equations, as introduced previously:

$$\begin{aligned} \frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \nu\nabla^2\mathbf{u} + \mathbf{g} \\ \nabla \cdot \mathbf{u} = 0 \end{aligned}$$

Surface tension is incorporated into this mathematical model as an additional force, $\sigma\boldsymbol{\kappa}$, where σ is the surface tension coefficient, and $\boldsymbol{\kappa}$ is the mean curvature normal at the surface:

$$\begin{aligned} \frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \nu\nabla^2\mathbf{u} + \mathbf{g} + \sigma\boldsymbol{\kappa} \\ \nabla \cdot \mathbf{u} = 0 \end{aligned}$$

Approaches to surface tension generally fall into two categories: those which apply surface tension as a body force in a region around the interface via smeared delta functions [27, 73, 71], and those which apply surface tension *discontinuously* at the interface, typically as a boundary condition in the pressure projection step. The latter is exemplified by the ghost fluid method and related approaches [18, 28, 29], and has been shown to provide more realistic results.

Surface tension models can also be compared in terms of how the force itself is approximated. In level set schemes, finite differences are often used to estimate mean curvature, though this can be quite inaccurate without careful modification [58], and cannot capture small details. If a surface mesh is available, a more accurate approach is either to use mesh-based curvature [11, 64], or to model a physical tension directly in the surface mesh geometry [51, 10, 71].

6.6.1 Body force approach

We will focus on methods which are uniquely suited to take advantage of explicit surface meshes. One idea is to treat surface tension as an actual *tension* per unit length. To accomplish this, we add three forces to a given triangle, corresponding to all neighboring triangles. The force is proportional to the surface tension coefficient times the edge length, in the direction normal to the edge and coplanar to the neighboring face (see figure 55) [51]. (In a two dimensional fluid sim with polygons acting as surfaces, we would add two forces to each surface edge, proportional to a surface tension coefficient, parallel to the directions of the two neighboring edges, as in figure 56.) We note this exactly conserves the total momentum of the volume of fluid, since the sum of all applied forces is identically zero.

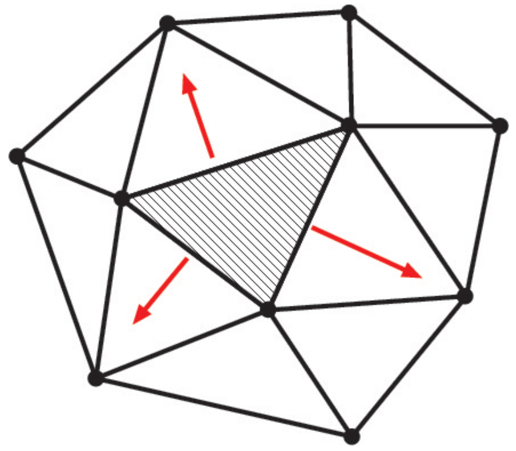


Figure 55: 3D surface tension forces acting on the shaded triangle

If the triangles themselves are simulation elements, the forces can be directly applied [10]. If the surface is embedded in a volumetric simulation, the forces can be summed up over each surface mesh vertex, and then added into the simulation as a body force [71]. Figure 57 shows a viscoelastic simulation incorporating this approach.

6.6.2 Boundary condition approach with an adaptive simulation mesh

Eulerian fluid simulations can incorporate surface tension by modifying the boundary conditions of the pressure projection problem. The projection step involves solving a Poisson

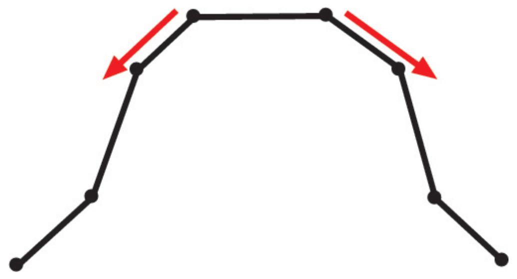


Figure 56: 2D surface tension forces acting on the top edge

problem for pressure, p, subject to a boundary condition on the free surface:

$$\nabla^2 p = \nabla \cdot \mathbf{u}$$
$$\text{subject to:} \quad p_{\text{fs}} = p_{\text{air}}$$

where p_{fs} is the pressure at the free surface and p_{air} is the constant air pressure. To incorporate surface tension forces, we instead set the free surface pressure $p_{\text{fs}} = p_{\text{air}} + \sigma\kappa_{\text{fs}}$, where κ_{fs} is the (scalar) mean curvature of the surface [18]. This ensures that the forces due to surface tension and the internal pressure forces are balanced, so no net volume loss occurs. If we are using a level set based surface, the curvature can be computed using finite differences. However, we would like to take advantage of the triangle surface mesh at our disposal.

Brochu et al. used surface-based curvature to enforce the pressure boundary condition while using a Ghost Fluid method. They first compute a curvature value for each vertex on the surface mesh using a local curvature estimate. To transfer these values to the simulation mesh where the pressure values are stored, they evaluate curvature at the intersection point between the the triangle mesh surface and the line joining an interior pressure sample to an exterior one. If this intersection point does not coincide with a surface mesh vertex, they use simple linear interpolation between the vertices of the surface triangle mesh.

Note that this approach can run into difficulties when using a regular Eulerian grid as the underlying simulation mesh. This is another artifact of the resolution mismatch problem described earlier. Brochu et al. show disaster results when using curvature measured on a high-res surface inside a low-res fluid sim. Because this low-res simulation cannot respond and correct high frequency sub-grid details present in the curvature estimates, the simulation becomes unstable almost immediately. Spurious noise affects the curvature estimates and induces increasingly large yet futile compensating velocities that destabilize the simulation. To prevent problems due to spurious surface noise, Brochu et al. rely on their adaptive simulation mesh to account for all surface details. Figure 58 shows this approach results in accurate, undamped surface tension behaviour.

6.6.3 Boundary condition approach with multiscale surface tension

An alternative approach to computing the pressure boundary condition was introduced by Sussman & Ohta [62] for implicit surfaces, and extended for use on explicit surfaces by

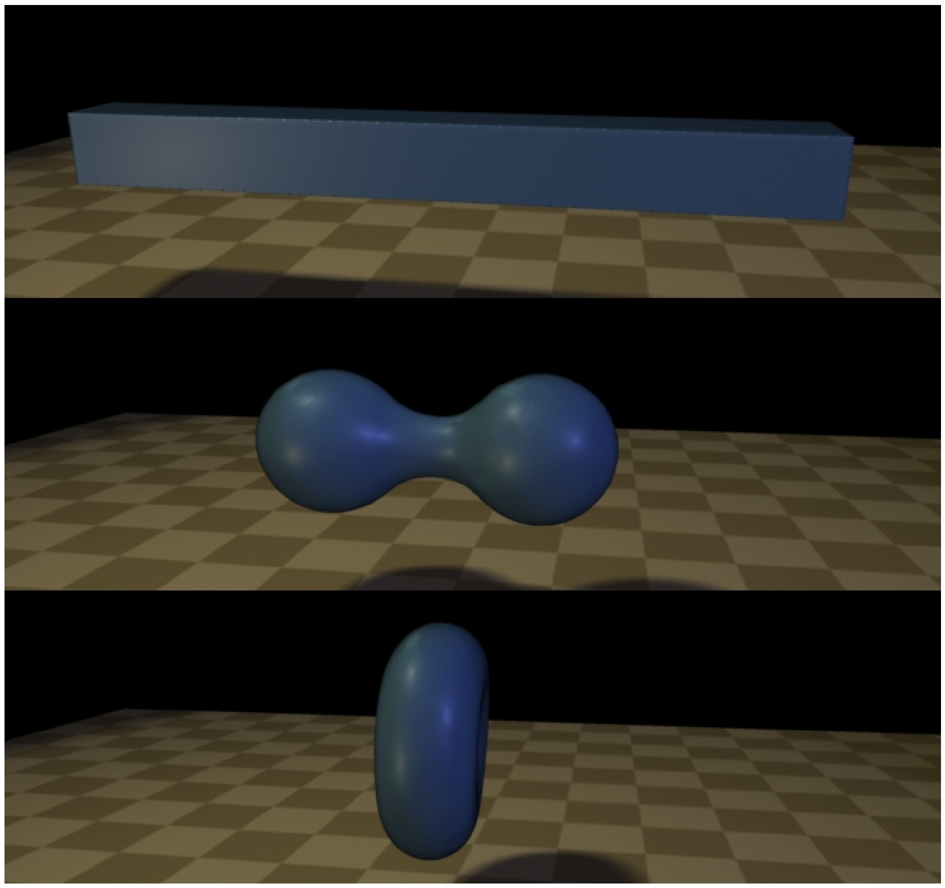


Figure 57: Surface tension computed on a surface and added into the simulation as a body force.

Thürey et al. [64]. This approach evolves the surface forward by one time step using a volume-preserving mean curvature flow of the liquid interface. The distance of the original towards the evolved surface is then used to compute the pressure boundary conditions. This results in better stability than applying a force proportional to the curvature measured from the surface.

This approach also suffers from artifacts due to the high-resolution surface when attempting to use it directly on a regular grid. Thürey et al. report that their regular grid Eulerian fluid solver has trouble resolving features on the order of one or two grid cells. Rather than taking the adaptive simulation approach of Brochu et al., they deal with this using a multiscale approach to surface tension.

We will now briefly outline the different steps of this algorithm, which are also illustrated in figure 60. For simulating the fluid, we use the standard Eulerian fluid solver with an embedded surface mesh. The input to our algorithm is a triangle mesh F representing the liquid interface. First, we advect F through the velocity field given by the previous step in the fluid simulation, using any of the advection methods described in previous chapters.

Next, we compute sub-grid cell surface tension dynamics by solving a wave equation on

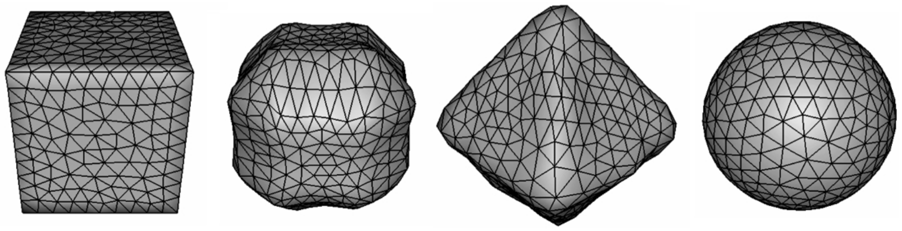


Figure 58: **Surface Tension.** From left to right: A cube in zero gravity begins to collapse due to surface tension, inverts to become an octahedron, and continues to oscillate rapidly before settling down to a sphere.

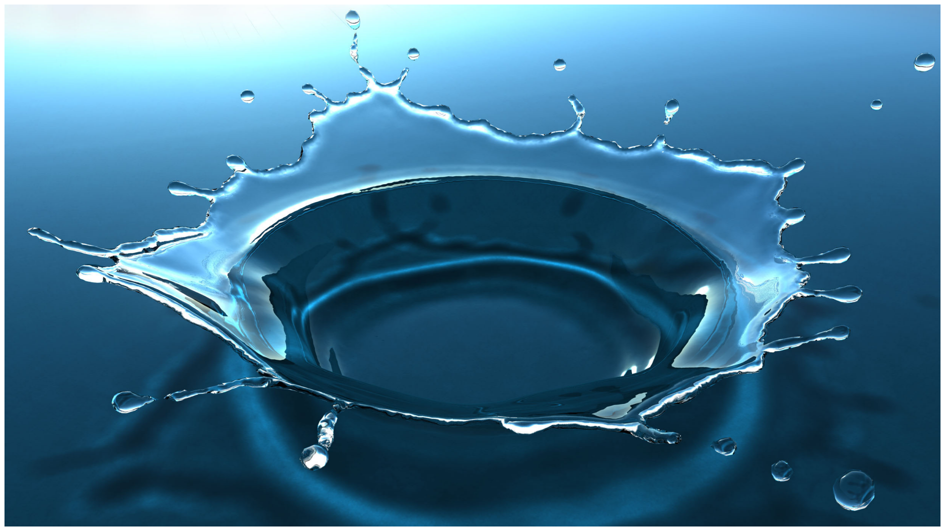


Figure 59: Separating grid-scale physics from a sub-grid surface simulation allows us to capture phenomena such as this water crown.

the surface. To set up the wave equation, we first filter out the surface features of F that are too high resolution to be captured by the fluid grid, in order to create a low-resolution smooth surface, S. We then initialize the wave equation problem with wave heights equal to the difference vectors between points on F and S. We solve the wave equation on F using an implicit Newmark scheme, resulting in surface dynamics that make F ripple and oscillate about the smoother surface S.

After that, because S has a low enough curvature to be fully resolved on the fluid grid, we use S to compute Eulerian surface tension forces in the fluid simulation. We show that surface tension forces can be treated as the time derivative of volume-preserving mean curvature flows in order to allow for much larger time steps than with other explicit schemes. We run a volume-preserving mean curvature flow on S proportional to the time step size and surface tension strength, producing another new surface, T. The difference vectors between T and S give us the surface tension forces that can be included as boundary conditions for the Eulerian pressure solve. Finally, we solve this system to get our final divergence-free

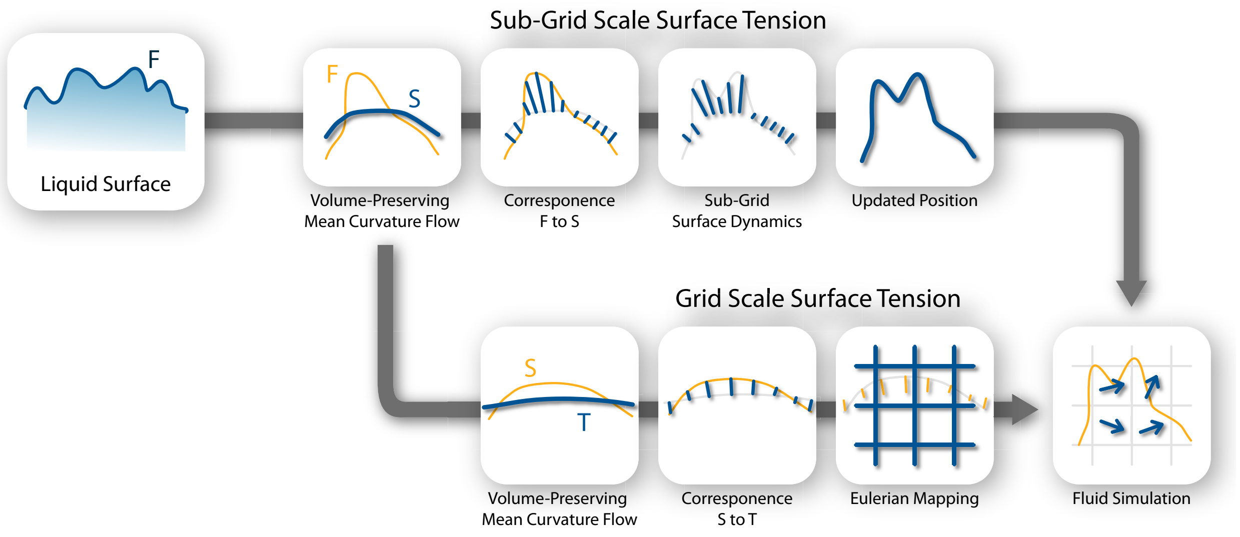

Sub-Grid Scale Surface Tension
F
Liquid Surface
F
S
Volume-Preserving Mean Curvature Flow
Corresponence F to S
Sub-Grid Surface Dynamics
Updated Position
Grid Scale Surface Tension
S
T
Volume-Preserving Mean Curvature Flow
Corresponence S to T
Eulerian Mapping
Fluid Simulation

Figure 60: Overview of our surface tension approach. The two rows illustrate the steps performed for the sub-grid surface tension dynamics in the top row, and the Eulerian surface tension in the bottom row.

velocity field and then move onto the next time step.

Figure 59 shows an implementation of this approach. Please see the paper by Thürey et al. [64] for implementation details.

6.7 The future of mesh-based surface tracking

Incorporating triangulated surface meshes into fluid simulation is an exciting area of research, since it crosses several areas of interest in the graphics community. The literature on mesh optimization alone is very deep, and there are many ideas that could be incorporated into future work. We end this section with a few suggested topics for future work.

Mesh surfaces for particle simulation: This course has focused on incorporating triangle mesh surfaces into Eulerian fluid simulations. The simulations we have discussed have been mainly on regular grids, or on unstructured simulation meshes (tetrahedral or Voronoi). However, there are many popular particle-based methods, such as SPH [47], which could benefit from using an explicit surface. Possible uses could include more accurate surface tension and other boundary conditions, or reseeding simulation particles to match the mesh surface.

Texture synthesis: Beyond advecting texture coordinates as discussed in section 6.1, another possibility is to generate textures as the surface deforms and moves. Work by Bargteil et al. [4] and Kwatra et al. [36] could be used to generate texture information on a liquid surface.

Robust boolean operations: A possible alternative approach to handling changes in topology is to perform boolean operations on the surface meshes. Although this has traditionally been quite difficult due to the complexity in handling degenerate geometry and robustness in the face of numerical error, some recent work has made this approach a real possibility in the near future [6, 65, 50, 13].

Parallel algorithms and real-time applications: Some methods discussed in this course lend themselves well to parallel implementions. An avenue of research in the near future could be exploring the power offered by GPUs and multi-core CPUs to move some of these ideas into the real-time domain.

Surface-based simulation: Although fairly uncommon in computer graphics, there is extensive work in scientific computing on using mesh surfaces as simulation elements. Thürey et al. [64] showed one approach, running a wave simulation on the surface mesh. Combining the techniques for fast and robust remeshing and topology changes we have discussed here with a surface-only simulation is another promising area of research.

7 Conclusion

This course has outlined the essential steps necessary to implement a state-of-the-art Eulerian fluid simulator with mesh-based surface tracking. These course notes have been assembled primarily from the lecturers' personal experiences while performing related research [10, 12, 11, 45, 71, 69, 64, 70, 68]. Section 3 of these notes discusses effective strategies for embedding a deforming surface mesh into a grid-based Eulerian simulator, Section 4 describes the key algorithms for maintaining a high quality surface mesh as it deforms, and Section 5 discusses four major strategies for allowing the deforming surface mesh to change topology. Each of these strategies has proven effective for advancing the state of the art in computer animation research, and we hope that these notes will help any course attendees implement and experiment with their own mesh-based fluid simulator.

While implementing a mesh-based fluid surface tracker may require more work than more elementary techniques, the benefits are enormous. The final section of these course notes outlines a great list of advantages to using a mesh-based fluid surface tracker. As explained in Section 6, explicit surface tracking allows for better preservation of visual detail, guaranteed volume conservation, and a rich class of topological features like thin sheets and strands that were not possible with any other simulation method. The goal of this course is that these notes are presented in enough detail to help developers implement mesh-based fluid surface tracking algorithms for use in motion pictures and video games, and that the notes are thorough enough to aid new researchers in building upon the presented techniques.

References

[1] Bart Adams, Mark Pauly, Richard Keiser, and Leonidas J. Guibas. Adaptively sampled particle fluids. *ACM Trans. Graph.*, 26, July 2007.

[2] C.B. Barber and H. Huhdanpaa. Qhull Software Package, 1995.

[3] Adam W. Bargteil, Tolga G. Goktekin, James F. O'Brien, and John A. Strain. A semi-Lagrangian contouring method for fluid simulation. *ACM Trans. Graph.*, 25(1):19–38, 2006.

[4] Adam W. Bargteil, Funshing Sin, Jonathan E. Michaels, Tolga G. Goktekin, and James F. O'Brien. A texture synthesis method for liquid animations. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, Sept 2006.

[5] Christopher Batty, Stefan Xenos, and Ben Houston. Tetrahedral embedded boundary methods for accurate and flexible adaptive fluids. In *Proceedings of Eurographics*, 2010.

[6] G. Bernstein and D. Fussell. Fast, exact, linear booleans. In *Proceedings of the Symposium on Geometry Processing*, pages 1269–1278. Eurographics Association, 2009.

[7] James F. Blinn. A generalization of algebraic surface drawing. *ACM Trans. Graph.*, 1:235–256, July 1982.

[8] Robert Bridson. *Fluid Simulation for Computer Graphics.* A K Peters, 2008.

[9] Robert Bridson, Ronald Fedkiw, and John Anderson. Robust treatment of collisions, contact and friction for cloth animation. *ACM Trans. Graph.*, 21(3):594–603, 2002.

[10] Tyson Brochu. Fluid animation with explicit surface meshes and boundary-only dynamics. Master's thesis, University of British Columbia, 2006.

[11] Tyson Brochu, Christopher Batty, and Robert Bridson. Matching fluid simulation elements to surface geometry and topology. *ACM Trans. Graph.*, 29(4):1–9, 2010.

[12] Tyson Brochu and Robert Bridson. Robust topological operations for dynamic explicit surfaces. *SIAM Journal on Scientific Computing*, 31(4):2472–2493, 2009.

[13] M. Campen and L. Kobbelt. Exact and robust (self-) intersections for polygonal meshes. In *Computer Graphics Forum*, volume 29, pages 397–406. John Wiley & Sons, 2010.

[14] Nuttapong Chentanez, Bryan E. Feldman, François Labelle, James F. O'Brien, and Jonathan R. Shewchuk. Liquid simulation on lattice-based tetrahedral meshes. In *SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 219–228, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[15] J. Du, B. Fix, J. Glimm, X. Jia, X. Li, Y. Li, and L. Wu. A simple package for front tracking. *Journal of Computational Physics*, 213(2):613–628, 2006.

[16] N. Dyn, D. Levine, and J.A. Gregory. A butterfly subdivision scheme for surface interpolation with tension control. *ACM transactions on Graphics (TOG)*, 9(2):160–169, 1990.

[17] D. Enright, F. Losasso, and R. Fedkiw. A fast and accurate semi-Lagrangian particle level set method. *Computers and Structures*, 83(6-7):479–490, 2005.

[18] D. Enright, D. Nguyen, F. Gibou, and R. Fedkiw. Using the particle level set method and a second order accurate pressure boundary condition for free surface flows. In *Proc. 4th ASME-JSME Joint Fluids Eng. Conf., number FEDSM2003–45144. ASME.* Citeseer, 2003.

[19] Douglas Enright, Ronald Fedkiw, Joel Ferziger, and Ian Mitchell. A hybrid particle level set method for improved interface capturing. *J. Comput. Phys.*, 183(1):83–116, 2002.

[20] Douglas P. Enright, Stephen R. Marschner, and Ronald P. Fedkiw. Animation and rendering of complex water surfaces. In *Proceedings of ACM SIGGRAPH 2002*, volume 21, pages 736–744, July 2002.

[21] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. Visual simulation of smoke. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 15–22, New York, NY, USA, 2001. ACM.

[22] Nick Foster and Ronald Fedkiw. Practical animation of liquids. In *SIGGRAPH '01*, pages 23–30, New York, NY, USA, 2001. ACM.

[23] M. Garland and P.S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 209–216. ACM Press/Addison-Wesley Publishing Co., 1997.

[24] A. Guéziec, G. Taubin, F. Lazarus, and B. Horn. Cutting and stitching: Converting sets of polygons to manifold surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 7(2):136–151, 2001.

[25] D. Harmon, E. Vouga, R. Tamstorf, and E. Grinspun. Robust treatment of simultaneous collisions. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 27(3):1–4, 2008.

[26] Nambin Heo and Hyeong-Seok Ko. Detail-preserving fully-eulerian interface tracking framework. In *ACM SIGGRAPH Asia 2010 papers*, SIGGRAPH ASIA '10, pages 176:1–176:8, New York, NY, USA, 2010. ACM.

[27] Jeong-Mo Hong and Chang-Hun Kim. Animation of bubbles in liquid. *Comput. Graph. Forum*, 22(3):253–262, 2003.

[28] Jeong-Mo Hong and Chang-Hun Kim. Discontinuous fluids. *ACM Trans. Graph.*, 24(3):915–920, 2005.

[29] Jeong-Mo Hong, Tamar Shinar, Myungjoo Kang, and Ronald Fedkiw. On boundary condition capturing for multiphase interfaces. *J. Sci. Comput.*, 31(1-2):99–125, 2007.

[30] B. Houston, C. Bond, and M. Wiebe. A unified approach for modeling complex occlusions in fluid simulations. In *ACM SIGGRAPH 2003 Sketches & Applications*, pages 1–1. ACM, 2003.

[31] X. Jiao, A. Colombi, X. Ni, and J. Hart. Anisotropic mesh adaptation for evolving triangulated surfaces. In *Proceedings of the 15th international meshing roundtable*, pages 173–190. Springer, 2006.

[32] Xiangmin Jiao. Face offsetting: A unified approach for explicit moving interfaces. *J. Comput. Phys.*, 220(2):612–625, 2007.

[33] Byungmoon Kim, Yingjie Liu, Ignacio Llamas, Xiangmin Jiao, and Jarek Rossignac. Simulation of bubbles in foam with the volume control method. *ACM Trans. Graph.*, 26(3):98, 2007.

[34] ByungMoon Kim, Yingjie Liu, Ignacio Llamas, and Jarek Rossignac. Advections with significantly reduced dissipation and diffusion. *IEEE Transactions on Visualization and Computer Graphics*, 13(1):135–144, 2007.

[35] Doyub Kim, Oh young Song, and Hyeong-Seok Ko. A semi-lagrangian cip fluid solver without dimensional splitting. *Computer Graphics Forum (Proc. Eurographics)*, 27:467–475, 2008.

[36] Vivek Kwatra, David Adalsteinsson, Theodore Kim, Nipun Kwatra, Mark Carlson, and Ming Lin. Texturing fluids. *IEEE Trans. Visualization and Computer Graphics*, 13(5):939–952, 2007.

[37] J.O. Lachaud and B. Taton. Deformable model with adaptive mesh and automated topology changes. In *Proc. 4th Int. Conference on 3D Digital Imaging and Modeling, Banff, Canada, IEEE*, pages 12–19, 2003.

[38] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87*, pages 163–169, New York, NY, USA, 1987. ACM.

[39] Frank Losasso, Frederic Gibou, and Ron Fedkiw. Simulating water and smoke with an octree data structure. In *Proceedings of ACM SIGGRAPH 2004*, pages 457–462. ACM Press, August 2004.

[40] Frank Losasso, Tamar Shinar, Andrew Selle, and Ronald Fedkiw. Multiple interacting liquids. *ACM Trans. Graph.*, 25(3):812–819, 2006.

[41] S. McKee, M.F. Tome, V.G. Ferreira, J.A. Cuminato, A. Castelo, Sousa F.S., and N. Mangiavacchi. The mac method. *Computers & Fluids*, 37(8):907–930, 2008.

[42] G.H. Meisters. Polygons have ears. *American Mathematical Monthly*, pages 648–651, 1975.

[43] Jeroen Molemaker, Jonathan M. Cohen, Sanjit Patel, and Jonyong Noh. Low viscosity flow simulations for animation. In *SCA '08: Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 9–18, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[44] Patrick Mullen, Alexander McKenzie, Yiying Tong, and Mathieu Desbrun. A variational approach to eulerian geometry processing. *ACM Trans. Graph.*, 26(3):66, 2007.

[45] M. Müller. Fast and robust tracking of fluid surfaces. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 237–245. ACM, 2009.

[46] M. Müller and M. Gross. Interactive virtual materials. In *the Proccedings of Graphics Interface*, pages 239–246, 2004.

[47] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. *Proc. of the ACM Siggraph/Eurographics Symposium on Computer Animation*, pages 154–159, 2003.

[48] Stanley Osher and Ronald Fedkiw. *The Level Set Method and Dynamic Implicit Surfaces.* Springer-Verlag, New York, 2003.

[49] Stanley Osher and James Sethian. Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations. *Journal of Computational Physics*, 79:12–49, 1988.

[50] Darko Pavić, Marcel Campen, and Leif Kobbelt. Hybrid booleans. *Computer Graphics Forum*, 29(1):75–87, 2010.

[51] Blair Perot and Ramesh Nallapati. A moving unstructured staggered mesh method for the simulation of incompressible free-surface flows. *J. Comput. Phys.*, 184(1):192–214, 2003.

[52] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, second edition, 1994.

[53] X. Provot. Collision and self-collision handling in cloth model dedicated to design garment. *Graphics Interface*, pages 177–89, 1997.

[54] Elbridge Gerry Puckett, Ann S. Almgren, John B. Bell, Daniel L. Marcus, and William J. Rider. A high-order projection method for tracking fluid interfaces in variable density incompressible flows. *J. Comput. Phys.*, 130:269–282, January 1997.

[55] Andrew Selle, Ronald Fedkiw, Byungmoon Kim, Yingjie Liu, and Jarek Rossignac. An unconditionally stable maccormack method. *J. Sci. Comput.*, 35(2-3):350–371, 2008.

[56] James Albert Sethian. *Level Set Methods and Fast Marching Methods.* Cambridge Monograph on Applies and Computational Mathematics. Cambridge University Press, Cambridge, U.K., 2^{nd} edition, 1999.

[57] Jonathan R. Shewchuk. What is a good linear element? interpolation, conditioning, and quality measures. In 11^{th} *Int. Meshing Roundtable*, pages 115–126, 2002.

[58] Seungwon Shin. Computation of the curvature field in numerical simulation of multiphase flow. *J. Comput. Phys.*, 222(2):872–878, 2007.

[59] Hang Si. *TetGen: A Quality Tetrahedral Mesh Generator and Three-Dimensional Delaunay Triangulator*, 2006.

[60] Funshing Sin, Adam W. Bargteil, and Jessica K. Hodgins. A point-based method for animating incompressible flow. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '09, pages 247–255, New York, NY, USA, 2009. ACM.

[61] Jos Stam. Stable fluids. In *SIGGRAPH '99*, pages 121–128, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[62] Mark Sussman and Mitsuhiro Ohta. A stable and efficient method for treating surface tension in incompressible two-phase flow. *SIAM Journal on Scientific Computing*, 31(4):2447–2471, 2009.

[63] M. Teschner, B. Heidelberger, M. Müller, D. Pomeranets, and M. Gross. Optimized spatial hashing for collision detection of deformable objects. In *Proceedings of Vision, Modeling, Visualization VMV03*, pages 47–54, 2003.

[64] Nils Thürey, Chris Wojtan, Markus Gross, and Greg Turk. A multiscale approach to mesh-based surface tension flows. *ACM Trans. Graph.*, 29(4):1–10, 2010.

[65] C.C.L. Wang. Approximate Boolean Operations on Large Polyhedral Solids with Partial Mesh Reconstruction. *IEEE transactions on visualization and computer graphics*, 2010.

[66] Brent Williams. Fluid surface reconstruction from particles. Master's thesis, University of Waterloo, 2008.

[67] Chris Wojtan. *Animating Physical Phenomena with Embedded Surface Meshes.* PhD thesis, Georgia Institute of Technology, 2010.

[68] Chris Wojtan. *Animating physical phenomena with embedded surface meshes.* PhD thesis, Georgia Institute of Technology, 2010.

[69] Chris Wojtan, Nils Thürey, Markus Gross, and Greg Turk. Deforming meshes that split and merge. *ACM Trans. Graph.*, 28(3):1–10, 2009.

[70] Chris Wojtan, Nils Thürey, Markus Gross, and Greg Turk. Physics-inspired topology changes for thin fluid features. *ACM Trans. Graph.*, 29(4):1–8, 2010.

[71] Chris Wojtan and Greg Turk. Fast viscoelastic behavior with thin features. *ACM Trans. Graph.*, 27(3):47, 2008.

[72] Jihun Yu and Greg Turk. Reconstructing surfaces of particle-based fluids using anisotropic kernels. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '10, pages 217–225, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[73] Wen Zheng, Jun-Hai Yong, and Jean-Claude Paul. Simulation of bubbles. In *Proc. of the ACM Siggraph/Eurographics Symposium on Computer Animation*, pages 325–333, 2006.

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

[75] D. Zorin, P. Schröder, and W. Sweldens. Interpolating subdivision for meshes with arbitrary topology. In *ACM SIGGRAPH 1996 papers*, page 192. ACM, 1996.