Dynamic explicit surface meshes and applications

by

Tyson Brochu

BSc, University of Regina, 2004

MSc, The University of British Columbia, 2006

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES
(Computer Science)

The University Of British Columbia
(Vancouver)

July 2012

Abstract

Explicit surface tracking encompasses the discretization of moving surfaces in 3D with triangle meshes. This thesis presents key contributions towards making explicit surface tracking tractable. I first deal with the topology change problem (the merging and splitting of mesh surfaces), introducing a framework for guaranteeing intersection-free surfaces while handling these topological changes. I then introduce new methods for continuous collision detection which are “exact” in the sense of returning the same results as they would if computed with symbolic or exact arithmetic, but which are implemented using faster, floating-point arithmetic.

The thesis also showcases several application domains in which explicit surface tracking can offer improvements over traditional methods: geometric flows, adaptive cloth simulation, and passive visualization of smoke simulation. It also presents two simulation techniques which take advantage of the explicit surface mesh representation and would not be possible with traditional methods: vortex sheet smoke and adaptive liquid simulation with high-resolution surface tension.

Preface

All work in this thesis was done under the supervision of Dr. Robert Bridson, and work in Chapter 3 and Chapter 5 was done without other collaborators besides Dr. Bridson.

Chapter 3 was published as: Tyson Brochu and Robert Bridson. Robust topological operations for dynamic explicit surfaces. *SIAM Journal on Scientific Computing*, 31(4):2472-2493, 2009.

Work in Chapter 4 was a collaboration with Essex Edwards, one of Dr. Bridson's PhD students. Essex's main contribution was the proof of the Root Parity Lemma, included as an appendix in this thesis for completeness. Some work from this chapter has been accepted to ACM SIGGRAPH 2012 as: Tyson Brochu, Essex Edwards, and Robert Bridson. Efficient Geometrically Exact Continuous Collision Detection. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 2012. Other sections were published as a technical report: Tyson Brochu and Robert Bridson. Numerically robust continuous collision detection for dynamic explicit surfaces. Technical Report TR–2009–03, University of British Columbia, 2009.

Some of the work in Chapter 5 was presented as a poster at SCA 2009, published as: Tyson Brochu and Robert Bridson. Animating smoke as a surface. *Eurographics/ACM SIGGRAPH Symposium on Computer Animation (posters and demos)*, 2009.

Chapter 6 was written with Todd Keeler, also one of Dr. Bridson's PhD students, who provided an implementation of the Fast Multipole Method (FMM), as well as suggesting the form of the integral equations to solve for solid boundaries. I developed the vortex sheet framework, including the linear solver for solid boundaries, and we collaborated on integrating Todd's FMM code into both the velocity and solid boundary computations. Since the FMM implementation is not my contribution, its description in the thesis is deliberately brief. The surface tracking component, including redistribution of edge circulations during remeshing, was my work, as was the interactive rendering. The offline, self-shadowed rendering was developed with Dr. Bridson. Work in Chapter 6 has been accepted for publication as: Tyson Brochu, Todd Keeler, and Robert Bridson. Linear-Time Smoke Animation with Vortex Sheet Meshes. *Proceedings of Eurographics/ACM SIGGRAPH Symposium on Computer Animation*, 2012.

Chapter 7 was a joint project with Dr. Christopher Batty, who was then a PhD candidate also under the supervision of Dr. Bridson, and the research effort was divided relatively evenly between us. Christopher proposed the initial idea of using Voronoi-based simulation in concert with my prior surface tracking research in order to capture thin features. I worked out the specifics of the intelligent sampling scheme that appeared in the paper and coded the majority of the full 3D liquid simulator. Christopher incorporated surface tension into the pressure projection, with my implementations of a few mean curvature estimates. We collaborated on several ideas for interpolation, and ended up using a scheme developed chiefly by Christopher. This chapter was co-presented at SIGGRAPH 2010 by Christopher and myself, and published as: Tyson Brochu, Christopher Batty, and Robert Bridson. Matching fluid simulation elements to surface geometry and topology. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 29:47:1-47:9, July 2010.

Table of Contents

List of Tables

List of Figures

Notation

$\mathbf{x}(t)$: mesh vertex positions
$\mathbf{T}(t)$: mesh triangles
$\vec{u}(\vec{x},t)$: velocity field
$\vec{x}_i$: position of vertex i at time t
$\vec{x}_i^*$: predicted position of vertex i at future time $t+\Delta t$
L_{max}: maximum edge length
L_{min}: minimum edge length
ΔV_{max}: maximum volume change
ξ: initial average edge length
ε_p: proximity threshold

$\vec{F}$: a vector valued function
Ω: function domain
$\partial\Omega$: boundary of a function domain

$\alpha\vec{g}$: smoke buoyancy force
$\vec{\omega}$: vorticity
Γ: circulation
K: fundamental solution to Laplace's equation, or a regularized approximation thereof
Φ: a scalar boundary potential function
σ: the layer density of a single-layer potential

γ: surface tension coefficient
κ_s: mean curvature of surface s

Acknowledgments

Since I started graduate school in 2004, my supervisor, Dr. Robert Bridson, has provided me with guidance, assistance, and positive motivation. Not only is he an incredibly gifted and talented researcher, but also an outstandingly kind and empathetic human being. His curiosity, ambition, and superhuman productivity have been an inspiration from day one.

I am deeply indebted to the experience of my other paper co-authors, Dr. Christopher Batty, Dr. Eric Brochu, Dr. Nando de Freitas, Essex Edwards, and Todd Keeler, as well as SIGGRAPH course collaborators Dr. Chris Wojtan and Dr. Matthias Müller.

During my PhD I was fortunate enough to collaborate with outstanding industrial partners Weta Digital and Digital Domain. I would particularly like to thank Simon Clutterbuck, Richard Dorling, and Sebastian Sylwan at Weta Digital, and James Jacobs at Weta Digital and Digital Domain for their time and support.

Thank you to my committee members, Dr. Alla Sheffer and Dr. Ian Mitchell, for advice, guidance, and constructive criticism throughout.

Over the past eight years, I've had too many friends, lab mates, and SIGGRAPH buddies to name. I am grateful to those of you who have made my time in the lab enjoyable, and my time at home relaxing and entertaining. Long-distance friends, thank you for giving me something extra to look forward to at the too-infrequent annual conferences.

Thanks to my parents Bernard and Sheila Joan for all their support over the years, and my brothers Eric, Lee, and Luc.

Finally, a big thank you to Gillian. Your support and patience through the late-night SIGGRAPH deadlines, exam stress, and time spent away at conferences and internships have kept me going over the years. (And thanks for being the literal voice of my research!)

Chapter 1

Introduction

Discretizing and maintaining surfaces are common tasks in physical simulation and computer graphics. In computer graphics, virtual artists such as modelers and animators often work directly with surfaces. In the simulation of physical phenomena, tracking a moving surface is often essential to capturing accurate dynamics if the surface defines regions of materials with different physical properties. For example, in a fluid simulation the surface can define the boundary between a liquid and surrounding air, or between regions of differing temperatures or densities in the same fluid. Combining simulation and graphics, *physics-based animation* aims to use concepts and techniques from physical simulation to capture the visual characteristics of various phenomena which can be difficult to animate by hand: clothing, elastic solids, rigid bodies, smoke, and liquids. The quality of the animation in many of these areas can be significantly affected by the quality of surface discretization and tracking. Furthermore, surfaces which undergo extreme deformations and changes in topology, such as merging and splitting, can pose a significant challenge.

Dynamic surfaces are surfaces that move over time. Surface motion can be specified by a number of different means, depending on the application. For example, geometric flows, such as motion normal to the surface or mean-curvature-minimizing motion are often used for volumetric segmentation. Another common domain where surface motion is encountered is physical simulation, such as fluid simulation, where the surface defines a material interface. *Explicit surfaces* are surfaces constructively defined by their geometry. With dynamic explicit surfaces, the surface geometry itself is updated, or tracked, as it moves. In this thesis, I will use the term “explicit surfaces” to refer almost exclusively to triangulated meshes in 3D, and line segment meshes in 2D.

Thesis outline

In the remainder of this chapter, I will introduce the main contributions of this thesis. In Chapter 2, I will review relevant previous work, in order to place the current work in the proper context of related research topics. Work in the first part of this thesis focuses on overcoming two of the main challenges in explicit surface tracking: handling changes in topology, and robustly detecting and preventing self-intersections in the surface mesh. As we shall see, these two problems can be closely related. Once these pieces are in place, I will show that our surface tracking method compares favourably to the level-set method, and demonstrate a few applications where the straightforward use of dynamic explicit surfaces can be beneficial: cloth simulation and the animation of smoke. Finally, I present simulation methods which can take advantage of the unique nature of explicit surface tracking: smoke simulation using vortex sheets, and liquid simulation with an adaptive volumetric simulation mesh.

Terminology

The term “topology” is somewhat overloaded. In this thesis, I will use the word *topology* to refer to the connectedness and genus of the surface, and *connectivity* to refer to the particular structure of the mesh’s graph. In other words, I will use topology to indicate global shape irrespective of mesh connections (e.g. sphere vs. torus), and connectivity to refer to the mesh itself (e.g. which vertices are connected by edges).

1.1 Robust topological operations for dynamic explicit surfaces

Implicit surface methods do not track the location of points on the surface, but rather use a collection of data points, usually located at fixed, regular intervals throughout the computational domain, to reconstruct the surface when needed. A popular example of an implicit method is the level set method [OF03], in which the zero-isocontour of a scalar function—usually a numerically approximated signed distance field—defines a surface.

Implicit surface functions are commonly used to represent dynamic surfaces because they easily handle topological changes, such as merging and pinching-off: no special effort is required. These changes are notoriously hard to handle with explicit surfaces — meshes often become "tangled" if they are advected into a self-intersecting state or if "mesh surgery" introduces holes that must be remeshed, and it can then be difficult to reconstruct a consistent, intersection-free surface.

Implicit methods, however, do suffer drawbacks such as numerical dissipation and the inability to reliably capture detail near or beyond the resolution of the underlying grid. 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. In fact, very thin yet smooth (low curvature) surfaces require excessively refined grids to avoid this problem. In addition, the user has no control to *prevent* topology change if the surface physics of the problem dictate a more subtle behaviour. A further drawback of implicit surface tracking is that the extraction of an explicit surface from an implicit representation (for example, for rendering) is still a non-trivial operation. Many of the challenges that face explicit surface tracking methods also apply here, and thus the complexity is shifted from surface tracking to surface extraction.

In Chapter 3 we present a framework for robustly handling topological changes in explicit surfaces, in a step towards tractable explicit surface tracking that does not suffer from the drawbacks of an implicit method. We consider two-dimensional surfaces embedded in $\mathbb{R}^3$, discretized as triangulated meshes.

The key idea 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. We thus first 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, and otherwise minimally perturb mesh positions to avoid the problem (patterned after frictionless inelastic collision response in a physical contact problem).

We present numerical experiments verifying convergence of our method for geometric flows with topology change, highlighting its ability to robustly and efficiently capture extremely thin and delicate details. This chapter is based on our SISC paper [BB09c]: Tyson Brochu and Robert Bridson. Robust topological operations for dynamic explicit surfaces. *SIAM Journal on Scientific Computing*, 31(4):2472-2493, 2009.

1.2 Efficient geometrically exact continuous collision detection

Maintaining the intersection-free invariant for a moving explicit surface (necessary for robust topological operations described above) requires robust and efficient collision and intersection detection. This thesis brings the paradigm of "Exact Geometric Computation" (EGC) to continuous collision detection. As defined by Yap [Yap04]:

> [EGC is the] preferred name for the general approach of "exact computation," as it accurately identifies the goal of determining geometric relations exactly. The exactness of the computed numbers is either unnecessary, or should be avoided if possible.

In other words, we are concerned with returning the correct Boolean result of a geometric test for given coordinates as if computed without rounding error. In intersection and collision testing, it is often of vital importance to get the correct Boolean result, even if there may be error in the estimate of where exactly an intersection occurs.

In Chapter 4, we introduce two new continuous collision detection approaches which build on a set of existing and new geometric predicates, and so can be made fully robust, in the EGC sense. Our methods do not require parameter tuning by the user, as they have no parameters which affect correctness.

The use of parameter-free, geometrically exact continuous collision detection makes it easier to maintain the intersection-free invariant for a moving mesh surface undergoing remeshing and changes in topology, and provides a key piece of our general-purpose explicit surface tracking package. We demonstrate the robustness of the proposed method with an adaptive cloth simulation.

Some work from this chapter has been conditionally accepted for publication at SIGGRAPH 2012 as: Tyson Brochu, Essex Edwards, and Robert Bridson. Efficient Geometrically Exact Continuous Collision Detection. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 2012. Other sections were published as a technical report [BB09b]: Tyson Brochu and Robert Bridson. Numerically robust continuous collision detection for dynamic explicit surfaces. Technical Report TR-2009-03, University of British Columbia, 2009.

1.3 Animating smoke as a surface

In visual effects, fluids such as smoke are often simulated with a grid-based fluid solver or particle system. The visible density of soot is usually tracked using either a set of passive marker particles, or a scalar density function on an Eulerian grid. If a marker particle system is used, tens of millions of particles may be required to prevent a grainy, bumpy, or blobby look, which can occupy considerable storage and network resources. Furthermore, if the smoke effect is particularly dense, only the set of particles or grid cells near the surface may be of interest, while the interior is almost completely occluded, thus using valuable storage and computational effort for regions which are not visible.

In Chapter 5 we embed a dynamic explicit surface in procedural and volumetric fluid simulations, visualizing the volume of smokey soot with surfaces. With a continuous mesh surface, we can achieve smooth, non-diffusive, high-quality renders of smoke and dye for a fraction of the storage necessary with a particle or grid representation. Some of the work in this chapter was presented as a poster at SCA [BB09a]: Tyson Brochu and Robert Bridson. Animating smoke as a surface. Eurographics/ACM SIGGRAPH Symposium on Computer Animation (posters and demos), 2009.

1.4 Linear-time smoke animation with vortex sheet meshes

In Chapter 6, we take the surface representation of smoke further, presenting a simulation method which uses the mesh primitives themselves as simulation elements, with no need for a volumetric simulation. We develop a vortex sheet model of smoke simulation, and argue that in the most useful asymptotic limit it captures all essential dynamic and visual detail in a closed 2D surface. We discretize the model with an adaptive dynamic triangle mesh, and model smoke dynamics using a vortex sheet method — a powerful technique where the velocity field can be evaluated explicitly using an integral over the surface.

Directly computing this integral for every point on the surface requires $O(N^2)$ operations to evaluate the velocity on N vertices: to overcome this limit we use a Fast Multipole Method (FMM) to approximate the velocity to arbitrary precision in optimal $O(N)$ time. Additionally, our method handles arbitrary no-stick moving solid boundaries. We demonstrate efficient smoke rendering techniques which are also linear in the number of mesh elements, providing complete $O(N)$ solutions for simulation and rendering.

Work in this chapter has been accepted for publication at SCA 2012 as: Tyson Brochu, Todd Keeler, and Robert Bridson. Linear-Time Smoke Animation with Vortex Sheet Meshes. *Proceedings of Eurographics/ACM SIGGRAPH Symposium on Computer Animation*, 2012.

1.5 Matching fluid simulation elements to surface geometry and topology

Finally, we focus on the simulation of liquid with explicit surface tracking. We demonstrate that naïvely plugging an explicit surface tracker into a volumetric liquid simulation leads to undesirable results, as the simulation and surface are often operating at different resolutions. The solution is either to simplify the surface mesh until it can be adequately resolved by the simulation, or to add degrees of freedom in the simulation to account for the higher-resolution surface mesh. In Chapter 7 we introduce a method for accomplishing the latter, using an adaptive simulation mesh based on the Voronoi diagram of an unorganized collection of simulation nodes. We also show that high-frequency, accurate surface tension can be achieved using mean curvature estimates from the triangle mesh itself. This chapter was presented at SIGGRAPH 2010, and published as [BBB10]: Tyson Brochu, Christopher Batty, and Robert Bridson. Matching fluid simulation elements to surface geometry and topology. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 29:47:1-47:9, July 2010.

Chapter 2

Related work

2.1 Surface tracking

As discussed in the previous chapter, methods for discretizing and evolving a surface embedded in Euclidean space can be lumped into two broad categories: implicit and explicit.

Implicit surface methods do not track the location of points on the surface, but instead use a collection of data points, usually located at fixed, regular intervals throughout the computational domain to reconstruct the surface when needed. They are sometimes called "front-capturing" methods since they do not explicitly track points on the surface, but rather contain the information needed to reconstruct the surface. Discretizing a signed distance function on a volumetric grid and defining the surface as the zero-isocontour of the function yields the level set method [OF03].

Explicit surface methods, by contrast, discretize the surface using a set of connected points. This is sometimes called "front tracking", as points on the surface are followed as the surface evolves, and these points entirely define the surface. Perhaps the best-known general purpose front tracking method is that developed by Glimm and collaborators [GGL+98]. They discretize the surface as a simplex mesh (line segments in two dimensions, triangles in three dimensions).

2.1.1 Implicit surface tracking

Since its introduction [OS88], the Level-Set Method has become very popular for implicit surface capturing, and has seen use in computer graphics, as well as image processing, computer vision, and scientific applications [Set99, OF03].

This method requires a volumetric grid of samples over the entire domain, even though usually only the surface is of interest. As such, many subsequent improvements to the method have aimed at focusing computational effort on capturing the surface. For example, the "narrow band" level set method [AS95] updates sample points only within a few grid cells around the zero level set of the implicit surface function, aiming to reduce the dimensionality of the problem. Intelligent storage of narrow bands, combined with hierarchical grids, has allowed for efficient storage and processing of implicit surface functions with extremely high effective resolutions [HNB$^+$06].

The "particle level set" method [EFFM02], which supplements the volumetrically discretized implicit surface function with a set of marker particles, has proven especially accurate in some applications. Particles are initialized in a narrow band around the zero level set, and each particle is given a sign depending on which side of the initial surface it initially lies. These particles are advected with the underlying velocity field, and, at each step, their signs are used to reconstruct the surface. Placing particles exactly *on* the initial surface results in the "marker particle level set method" [MMS07]. This particular method has the added benefit that additional surface data can be stored on the particles (for example, texture coordinates for graphics applications). Additional accuracy has also been achieved through the use of an octree grid structure [LGF04], which can efficiently increase the resolution of the level set function discretization around the interface.

Volume-Of-Fluid (VOF) methods were introduced by DeBar [DeB74] and Noh and Woodward [NW76] for the simulation of fluids. These methods operate on a fixed volumetric voxel grid, maintaining the fraction of volume occupied by fluid at each voxel. At the beginning of the simulation, these volume fractions are initialized using the known geometric interface. At subsequent steps in the simulation, the volume fractions are evolved according to the advection equation. The actual interface must then be reconstructed from these volume fractions when required.

See work by Rider and Kothe [RK98] for more details and an overview of some advancements in this technique over the past several decades.

A method similar to VOF was introduced to the computer graphics community by Mullen et al. [MMTD07], who store a "mass density" function of an object at regular grid cells. This density is advected using a finite-volume approach which conserves total mass, in effect perfectly conserving the volume defined by the mass density function.

The Coupled Level Set and Volume Of Fluid method was introduced by Sussman [Sus03] and subsequently applied to the animation of boiling liquids in graphics [MUM$^+$06]. This approach maintains a signed distance field, as in the level set method, as well as a volume fraction per cell, as in VOF. The signed distance field is used to cull spurious volume fractions which might occur outside and away from the actual surface (due to numerical error). The volume fractions, in turn, are used to correct volume loss in the level set function.

The use of passive marker particles to track a surface is common in applications such as fluid simulation. For example, the Marker-And-Cell method [HW65] uses an Eulerian grid to drive the fluid simulation, and passive marker particles to indicate the interior of the fluid. An explicit surface can then in principle be reconstructed as needed, though we note determining an accurate smooth surface from the marker particles remains an open research problem. We might classify the marker particle approach as both Langrangian *and* implicit since it uses the Lagrangian frame of reference but still only *captures* information necessary to reconstruct a surface. Using particles to track the location of fluid is common in computer graphics [ZB05], and the extraction of a smooth, temporally coherent surface from a set of particles is an ongoing area of research [APKG07, Wil08, YT10, BGB11]. Using the alpha-shape [EM94] of a set of particles to extract a surface, as is done in the Particle Finite Element Method [OIPA04], is also a promising direction.

Torres and Brackbill [TB00] introduced the point-set method, arguing that surface tracking does not necessarily require connectivity information between surface particles. To compute normals and curvatures, they first construct an indicator function on a regular grid (similar to a signed distance field, but only containing inside-outside information). This function is smoothed around the interface so that the isocontour representing the interface passes through the surface particles. Normals and curvatures are then defined using the partial derivatives of the indicator function.

2.1.2 Explicit surface tracking

In our discussion of previous explicit surface tracking work, we will distinguish between methods which do not address changes in topology, and methods which handle these changes to varying degrees.

Fixed topology

In the scientific computing literature, explicitly-discretized surfaces are often used to track, for example, the interface between two volumes of fluid, or the exterior surface of a solid. Often these surfaces can contain the simulation elements themselves, and much work on surface tracking has been done in the context of these "boundary methods".

Boundary Element Methods [BTW84] are a general class of physical simulation approaches where the simulation elements are located on a domain boundary. Generally, a fundamental solution of the partial differential equation being studied is employed, the boundary conditions are partially specified, and the remaining boundary conditions are solved for. The appeal of such approaches is that the dimensionality of the problem is reduced, since we are concerned with elements on a surface, rather than throughout a volume. Problems studied with this approach include elasticity, Stokes flow, and electromagnetism. In computer graphics, BEM has been used to model linear elasticity for interactive applications [JP99, PDJ$^+$01].

In fluid dynamics, vortex sheets model layers of fluid where a discontinuous jump in velocity occurs. Often this layer corresponds to the interface between two different fluids. These layers are often discretized as surface meshes (although level-set methods have also been used). This framework has been used to study inviscid and incompressible multifluid flows with surface tension [HLS01, Poz00], rising bubbles with viscosity and surface tension [UT92], the Rayleigh-Taylor Instability [GMMS86], and many other phenomena in fluid dynamics. (One literature review of vortex methods contains over 450 references [Sto07].) More recently, adaptive vortex sheets have been introduced, where vertices are added to and removed from the surface during simulation [SDT08]. To our knowledge, none of these methods address "inter-sheet" refinement which would result in changes in topology.

General purpose surface tracking methods proposed by Jiao and collaborators [JCNH06, Jia07] can handle highly-deforming surfaces with excellent fidelity, but they do not address surface merging or separation.

Changing topology with grids

Augmenting a surface mesh with a grid-based implicit representation transfers some of the benefits of implicit methods to explicit surface tracking. In particular, globally or locally constructing an implicit surface from a mesh, then extracting a new explicit surface, can automatically handle some changes in topology.

Researchers in medical imaging have long used deformable contours in 2D ("snakes") , to segment images [KWT88], and deformable surfaces for volume segmentation [MT96]. McInerney & Terzopoulos [MT00] introduced an approach in which a deformable surface is augmented with a regular volumetric grid. Intersection points between the grid edges and the surface are used as points for mesh subdivision, and the grid nodes can maintain an implicit representation of the surface which aids in performing robust topology changes.

Another hybrid implicit/explicit surface tracking method called "grid-based front tracking" was introduced by Glimm et al. [GGLT99]. In this method, explicit surfaces that have been advected into an intersecting state are treated in one of two ways. First, a grid-free untangling is attempted, which cuts the triangle mesh along intersection contours and re-triangulates. If this fails, a global, grid-based reconstruction is performed. (A similar effect is achieved by Bargteil et al. [BGOS06], where the explicit surface is regenerated from a level set discretization at each step.) Du et al. [DFG$^+$06] refined this method, performing only local grid-based reconstruction around the intersecting mesh components. Wojtan et al. introduced a similar approach to computer graphics [WTGT09, WTGT10], extending this idea by identifying thin structures which should not be removed by local reconstruction, thus preserving small features in their fluid simulations. Alternatively, construction of the implicit surface function may be skipped and a new, consistent mesh may be constructed directly from the old mesh using marching-cubes-style stencils on the volumetric grid [Mül09].

With all of these approaches, regular grid nodes are used to determine when the surface is in an intersecting state, and thus sub-grid-scale intersections may be missed; the grid reconstruction similarly eliminates any details at or below the grid resolution, including smooth but thin parts of the surface.

Changing topology, grid-free

Changing mesh topology without the use of a grid is somewhat rarer in the literature, although a few papers deal with topology changes to varying degrees. Brakke's Surface Evolver [Bra92] implements an operation called "vertex popping" in which vertices with disconnected neighbourhoods are identified, then duplicated and pulled apart, achieving a topological splitting event. Garland and Heckbert [GH97] allowed surfaces to merge during simplification by collapsing pseudo-edges between two vertices on different surface patches. Our method differs in that we use robust geometric predicates for interference detection, allowing for non-regular, anisotropic triangulations while guaranteeing intersection-free meshes. Our method also permits surfaces to approach much closer than the length-scale of a triangle, critical for handling thin yet smooth geometry efficiently.

Tryggvason et al. [TBE$^+$01] combine a front tracking approach with a grid-based fluid solver. In two dimensions, they model merging and splitting of droplets by detecting surface elements which are near to one another and performing the appropriate change in surface connectivity. However, the authors acknowledge that their extension of this method to 3D is not as robust, and has not thus far been demonstrated for a wider variety of scenarios.

Lachaud and Taton [LT03] use dynamic, purely explicit surfaces with interference detection to determine when topological changes should occur. Their method of interference detection relies on maintaining a regular triangulation of the surface mesh and detecting when any two vertices are close to each other. When such a pair of vertices are detected, a remeshing operation is triggered, which may change the surface topology. The downside of this approach is that the minimum vertex spacing on the mesh is the same as the minimum vertex spacing between vertices on disjoint surface patches, implying that two separate surfaces can be no closer together than the minimum edge length.

Pons and Boissonnat [PB07] handle topological changes in explicit surfaces by embedding the surface mesh in a tetrahedralization of the surface vertices. The tetrahedralization is updated at each step, rejecting those tetrahedra whose circumcenters lie outside the surface mesh. The exterior triangles of the remaining tetrahedra form a new triangulation of the surface. This tetrahedralization is shown to be a good approximation of the input surface mesh, but is not guaranteed to conform to the surface triangles used in the previous step.

Similarly, Misztal et al. [MBE$^+$10] have developed a fluid simulation system based on a tetrahedralization of space. The faces of the fluid surface are a restricted set of tetrahedral faces, and as the tetrahedron vertices move through space and the volumetric mesh is modified, the surface is automatically updated. When tetrahedra are collapsed down to zero volume, they are removed in another volumetric remeshing operation, which can also modify the surface topology.

Recent advances in exact mesh Boolean operations [PCK10, CK10] offer promising paths for future development of surface tracking algorithms that robustly handle changes in topology. Although constructive solid geometry (CSG) operations have been shown to work with these approaches, a fully robust, topology-changing dynamic explicit surface tracking package has not yet been demonstrated.

2.1.3 Adaptive surface remeshing

One important component of explicit surface tracking is remeshing. Several undesirable mesh characteristics can be avoided by changing mesh connectivity or triangle shape:

- Overly refined meshes may cause unnecessary computational expense, so we may wish to minimize the number of elements while achieving some measure of accuracy in approximating the original surface [HDD$^+$93, GH97].

- Large angles may result in bad gradient interpolation, affecting, for example, rendering. As pointed out by Shewchuk, small angles negatively affect the conditioning of stiffness matrices if the mesh is used for FEM simulation [She02].

- Extreme deformation may result in a high variance of vertex density on the surface [JCNH06].

- If the surface is driven by a grid-based simulation, it is desirable to maintain a surface resolution comparable to that of the underlying grid so that information on the grid is adequately captured by the surface [TBE$^+$01].

Here we discuss the remeshing problem, very broadly defined as: given an input mesh, produce an output mesh which is in some sense “improved”. There are many possible desirable characteristics including: triangle quality, closeness to input mesh geometry, regular connectivity, denoising, compatible meshing, feature restoration, and self-intersection repair.

Mesh optimization

Several static remeshing papers employ a mesh optimization framework [SZL92, HDD$^+$93, GH97, FB98], where a compromise is sought between triangle quality and faithfulness to the input surface. Mesh quality can be measured in several ways: minimum and maximum edge length and angle, aspect ratio, etc. The optimal mesh can be approached discretely, via changes in connectivity such as edge collapse, splitting or flipping, or continuously, by moving mesh vertices.

During the vertex moving phase, vertices are often moved according to a Laplacian-like scheme, since this tends to produce isotropic triangles and regular triangulations. However, naïvely applying Laplacian smoothing or an umbrella operator can lead to volume loss and erosion of features such as edges and corners. Several modifications have been proposed to counter this loss of volume [DMSB99, VRS03, Jia06].

A map over the mesh vertices can also be employed to adjust vertex density. For example, more vertices are required to resolve regions of high curvature, so scaling distance computations by a curvature map can result in more vertices being created in high-curvature regions [FB98].

Complete remeshing

The methods mentioned above incrementally optimize the surface by changing connectivity locally and adjusting vertex positions. An alternative is to create a new set of vertices which lie on the surface, then construct a new triangle mesh from these points, and discard the old mesh.

Early work by Turk [Tur92] used an attraction-repulsion scheme to evenly redistribute a set of points on the surface, while keeping them on the input surface. Once an optimal distribution is found, the points are meshed, resulting in a regular, isotropic mesh.

Another family of remeshing algorithms is based on *Centroidal Voronoi Tessellation* [DFG99, AVDI03]. Here a set of points are seeded on the surface, then redistributed by moving each seed point towards the centroid of its associated Voronoi cell. Constructing the Delaunay triangulation of the points results in a highly regular, isotropic triangle mesh. Computing the Voronoi diagram can be costly, since it must be done with respect to the geodesic distance, rather than Euclidean distance. Recent approximation approaches have increased efficiency of these methods [YLL$^+$09], but it has not seen widespread use for applications requiring per-frame remeshing.

Remeshing for dynamic surfaces

In this work, we wish to maintain good quality surface mesh undergoing deformation. We can see this as static remeshing at each time step. However, not all methods for static meshing are appropriate. For example, many approaches compute a Delaunay triangulation of the surface vertices, which can be quite expensive to run at each time step (complexity is $O(N \log N)$). Instead, we will look at local, optimization-type algorithms, which have linear complexity in the number of mesh vertices. Further, many of these approaches take advantage of time coherence — if the mesh vertex positions don't change much between time steps, then the mesh quality doesn't change much, and so the optimization is fast. For these reasons, many researchers in dynamic explicit surface tracking use some combination of the familiar edge-based local remeshing operators: edge flip, edge split, and edge collapse [GGL$^+$98, TBE$^+$01, LT03, JCNH06, WT08].

Other researchers working with dynamic surfaces use some form of regular grid to improve surface regularity. The surface is embedded in a grid, and points where the grid intersects the surface become candidate points for new vertices [MT00]. This ensures that large, flat elements are subdivided regularly. Müller takes this approach further by discarding the original mesh at each step, and reconstructing a new mesh from these intersection points [Mül09]. This complete remeshing is fast, and can also account for changes in topology. However, the complexity of the topology representable on the grid is limited by the number of marching cubes-like templates we wish to consider, as well as the resolution of the grid.

2.2 Collision detection

Chapter 4 deals with *continuous collision detection*, defined as the process of detecting if a mesh moving between initial and final configurations over a time step comes into contact with itself during that time step. A more complete review of previous methods is in that chapter, but we will briefly mention that the predominant method, developed and popularized by Provot [Pro97] and Bridson et al. [BFA02], is based on solving a cubic equation for each pair of mesh elements to determine if and when the elements are coplanar, then running a distance or intersection query at these *coplanarity* times to determine if there is a collision. This method relies on user-set parameters to minimize the number of false positives, while ensuring no false negatives. Our new approach is quite different, and brings the paradigm of *exact geometric computation* [Yap04] to continuous collision detection, computing the results of collision queries as if they had been computed with exact arithmetic, without going to a symbolic representation. Our method makes use of floating-point expansion predicates [Dek71, She96] with interval arithmetic filters [BBP01] for exactness and efficiency.

2.3 Fluid simulation for animation and visual effects

Fluid simulation for computer graphics has become a broad topic in recent years. A thorough survey of all techniques is beyond the scope of this thesis, so we refer the reader to the textbook by Bridson [Bri08] for an extensive literature review. Chapter 5 and Chapter 6 both deal with the simulation of smoke, and previous work related to this sub-field is discussed in these chapters. Chapter 7 is about the simulation of liquids with free surfaces for visual effects, and contains a discussion of previous work in this area, with a focus on adaptive methods and surface tension.

Chapter 3

Robust topological operations for dynamic explicit surfaces

3.1 Introduction

In this chapter we describe the main operations used in our surface tracking library, El Topo. This library is used in all of the applications presented in this thesis. We then present convergence tests for a few geometric flow problems which illustrate the accuracy of our method.

3.1.1 Related work

Most of the literature relevant to this chapter is discussed in Chapter 2. We provide references to specific concepts as they appear in the text.

3.2 Our approach

3.2.1 Framework

Our method operates on triangle meshes. Let the mesh configuration at time t, be defined by the pair $(\mathbf{x}(t), \mathbf{T}(t))$, a set of vertex positions and a set of triangles defining the mesh connectivity. Each triangle is a triple of vertex indices, oriented consistently over all triangles.

$\mathbf{T}$ is a function of time, indicating that connectivity can change over time, but note that connectivity changes occur at discrete events, for instance when a vertex is added or removed. In contrast, $\mathbf{x}(t)$ is usually an approximation of a continuous trajectory of vertex positions.

Our method operates in two phases, which we call the "static operations" phase, and the "integration" phase. The first phase operates on a mesh at an instant in time, not considering its motion. We take as input a mesh configuration $(\mathbf{x}(t_n), \mathbf{T}(t_n))$ and produce a modified mesh configuration $(\mathbf{x}'(t_n), \mathbf{T}'(t_n))$. We perform mesh optimization as well as changes in topology during this phase, and we describe these operations in detail in this chapter. In general, the output mesh will not share the same connectivity as the input mesh and may have a different number of vertices.

In the second phase, the user provides a set of "current" and "predicted" vertex positions, $\mathbf{x}'(t_n)$, and $\mathbf{x}^*(t_n + \Delta t)$ as well as the connectivity information, $\mathbf{T}'(t_n)$. The choice of input vertex locations is determined by the user, usually the result of a physical simulation or geometric flow. Our system produces as output a "final" set of vertex positions, $\mathbf{x}(t_n + \Delta t)$, which is as close as possible to the predicted positions, but guaranteed to define an intersection-free mesh. We ensure that as long as the current mesh is intersection-free, the final mesh will be intersection-free as well, even if the given predicted mesh is not. The mesh connectivity does not change in this phase.

We note that the underlying simulation may use any time integration scheme to get the predicted vertex positions. For example, the simulation might advect a surface vertex using a high-order, multi-step time integration scheme (in a collision-naive way) from time t_n to $t_n + \Delta t$ and pass in the resulting $\mathbf{x}^*(t_n + \Delta t)$. If this trajectory is indeed free of interference, the resulting integration will retain the accuracy of the high-order scheme.

There are a few inadmissible mesh triangulations which we will not accept as input, and will not produce as output. For simplicity of presentation, we do not allow boundary edges (edges incident on fewer than two triangles), but return to this later in Section 3.3.4 when extending the method to handle open surfaces. We do not allow two triangles to share the same three vertices (creating a zero-volume tetrahedron). We **do** allow some non-manifold surfaces. In particular, we allow more than two triangles to be incident on an edge. However, each triangle incident on an edge must have another triangle with a consistent orientation incident on the same edge. We also only allow an even number of triangles to be incident on a single edge. Loosely speaking, we allow only meshes that partition the computational domain into an interior and an exterior, i.e. we ensure the mesh is the boundary of an open set.

This restriction allows us to generalize mesh operations originally intended only for manifold surfaces in a consistent way. For example, an edge flipping operation (see Section 3.2.4) usually assumes that the edge is incident on two triangles with consistent orientation. Our requirement above ensures that there are *at least* two triangles with consistent orientation incident on the edge, so we can find two such triangles and apply the edge flip operation.

3.2.2 Algorithm overview

Here we provide a high-level outline of our algorithm. Each non-trivial step is explained in detail in the sequel. The first and fourth steps, splitting long edges and null-space smoothing, are mesh maintenance steps and can be omitted at the user's discretion. The second and third steps, edge flipping and short edge collapsing, are mainly used for mesh maintenance, but are also key to allowing surface separation, as described in Section 3.2.5, and thus omitting these steps will prevent separation events (which may or not be desirable, depending on the application). We also allow the option of using these steps but preventing topology changes by adding extra checks to prevent surface separation. The zippering step can also be omitted if the user wishes to avoid topological changes altogether, as might be appropriate in some applications.

Our software implementation allows the user to toggle three Boolean settings: whether to perform mesh improvement operations, whether to allow topological changes, and whether to enforce intersection-free surfaces. Algorithms 1 and 2 show the main two phases of our system, and Algorithm 3 shows an example, Forward Euler-based simulation loop with an external velocity field. The static and integration operations are as follows:

Split long edges (Section 3.2.4): Subdivide all edges with length greater than the user-defined maximum edge length.

Flip sub-optimal edges (Section 3.2.4): Repeatedly search the mesh for edges which are "sub-optimal" (defined below) and replace them.

Collapse short edges (Section 3.2.4): Delete all edges with length less than the user-defined minimum edge length, replacing the edge with a single vertex.

Null-space smoothing (Section 3.2.4): Apply a Laplacian-type filter to the vertex positions, moving each vertex only in the null space of its local quadric metric tensor.

Merging/zippering (Section 3.2.6): Detect edges that are near to each other and attempt to merge surfaces by deleting their incident triangles and zippering the resulting holes.

Proximity detection and repulsion forces (Section 3.2.7): Detect elements that are near to each other and apply a repulsion force between them by adjusting their predicted final positions.

Impulse-based collision resolution (Section 3.2.7): Detect individual collisions using continuous collision detection and apply impulses which will prevent an intersection.

Impact zones (Section 3.2.7): Detect remaining collisions, group colliding elements into impact zones, and solve for a set of impulses which will prevent intersection. If this fails, compute a uniform rigid motion for the group of colliding vertices.

3.2.3 Interference detection

Before diving into the details of our approach, we briefly discuss techniques for 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 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

Algorithm 1 Static operations

```
Given: mesh $(\mathbf{x}(t_n), \mathbf{T}(t_n))$
Split long edges
Flip sub-optimal edges
Collapse short edges
Null-space smoothing
Merge close edges
return mesh $(\mathbf{x}'(t_n), \mathbf{T}'(t_n))$
```

Algorithm 2 Integration

```
Given: mesh $(\mathbf{x}'(t_n), \mathbf{T}'(t_n))$, predicted positions $\mathbf{x}^*(t_n + \Delta t)$
Proximity detection and repulsion forces
Impulse-based collision resolution
if collisions persist then
   Impact zones
end if
return final mesh $(\mathbf{x}(t_n + \Delta t), \mathbf{T}(t_n + \Delta t))$
```

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. We also use a static point-in-tetrahedron test during mesh maintenance (described below).

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 denote this proximity tolerance ε_p.

Algorithm 3 Sample simulation loop

```
Given: mesh $(\mathbf{x}(t_0), \mathbf{T}(t_0))$, velocity function $F(\mathbf{x}, t)$, time step $\Delta t$.
for $n = 0 \to N$ do
   $(\mathbf{x}'(t_n), \mathbf{T}'(t_n)) =$ static operations $((\mathbf{x}(t_n), \mathbf{T}(t_n)))$
   Evaluate velocities $\mathbf{v} = F(\mathbf{x}'(t_n), t)$
   $\mathbf{x}^*(t_n + \Delta t) = \mathbf{x}'(t_n) + \Delta t \mathbf{v}$
   $(\mathbf{x}(t_{n+1}), \mathbf{T}(t_{n+1})) =$ integration $((\mathbf{x}'(t_n), \mathbf{T}'(t_n)), \mathbf{x}^*(t_n + \Delta t))$
end for
return final mesh $(\mathbf{x}(t_N), \mathbf{T}(t_N))$
```

Proximity detection also finds the two points on the mesh elements that are closest to each other. If we denote the set of four barycentric coordinates of these two points as **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 **p** are the indices of the vertices involved, then the shortest distance is given by the length of this vector:

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

Our proximity detection function can also return a "collision normal", $\vec{n}$, which, when an impulse is applied along it, will increase the distance between elements.

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 given: 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 $t_n + \Delta t$. Given these two configurations, continuous collision detection will return any point-triangle and edge-edge collisions, as well as a collision normal, the set of barycentric coordinates describing the point of contact, and the computed relative displacement. The output of this collision detection routine is used in the *collision resolution* routines, which will be described in Section 3.2.7. If the time of collision is known from the collision detection function, the collision normal is often computed by interpolating the geometry at that time and taking either the triangle normal, or the normal orthogonal to both edge lines. If the time of contact is not known, we generally take the normal at the mid point of the time step (see Chapter 4), though a more sophisticated estimate of the time of contact (achieved through root finding, for example), could also be used.

In Chapter 4, we will recap the current state-of-the-art methods for continuous collision detection, and introduce novel, exact methods for detecting collisions.

3.2.4 Mesh quality improvement

Triangles with small areas or poor aspect ratios can adversely affect collision detection, topological operations and any simulation running on the mesh. To improve the quality of our surface discretization, we use a few common operations. The need for these operations and their effectiveness has been argued by others [JCNH06] and is orthogonal to the main contribution of the chapter, robust topological changes. However, we include this section for completeness, and highlight our approach to ensuring they do not introduce any intersections in the mesh.

Edge split

If an edge is longer than a user-defined maximum edge length (denoted L_{max}), we subdivide it by introducing a new vertex (see Figure 3.1). 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. To ensure intersection safety in this case, we can make use of the continuous collision detection framework introduced earlier. We begin by introducing the new vertex at the edge midpoint. We then compute the "predicted" location of the new vertex via the subdivision scheme. These two points define a *pseudo-motion*. 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 doesn't collide with any existing mesh elements (which do not move during this pseudo-motion). If a collision does occur we revert to using the edge midpoint, which is guaranteed to not introduce any new intersections.

We do not attempt to subdivide non-manifold edges (edges incident on more than two triangles), although if handled carefully these edges could be treated as well. In our experience, these edges are rare enough that failing to subdivide them does not introduce significant error.

Edge flip

We employ an edge flip operation as a way of helping to keep edge lengths uniform. 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

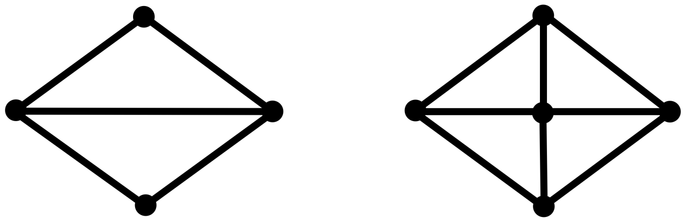


Figure 3.1: Edge split operation on the mesh graph

the edge and create a new edge between these two points (see Figure 3.2). This is a useful heuristic to keep edges all roughly the same length. With better criteria, these edge flips could help improve triangle aspect ratios (see Shewchuk for applications in FEM meshing [She02]).

Again, we must ensure this operation does not introduce any intersections. A simple way of doing this 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 denote this maximum volume change ΔV_{max}, and usually set it to be $0.1\xi^3$, where ξ is the average edge length at the beginning of the simulation; for simulations involving extremely thin surfaces such as our curl noise example later, this may need to be further reduced). We extend this operation to handle non-manifold edges by choosing *any pair* of incident triangles with consistent orientation, and applying these steps to the edge and the chosen pair of triangles.

Flipping a single edge may introduce new triangles with sub-optimal edge lengths, so we iteratively sweep over all edges in the mesh until no flip is performed, or until we reach a maximum number of sweeps (in our implementation, we set this maximum to five, which is almost never reached). 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.

Edge collapse

If an edge is shorter than a user-defined minimum edge length (denoted L_{min}), we attempt to collapse it by replacing it with a single vertex, as shown in Figure 3.3.

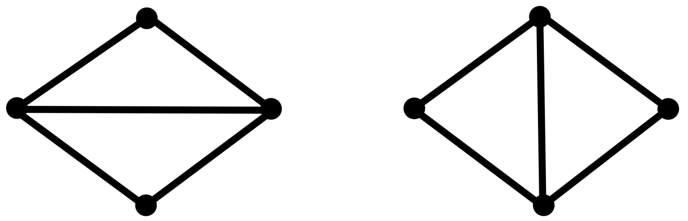


Figure 3.2: Edge flip operation on the mesh graph

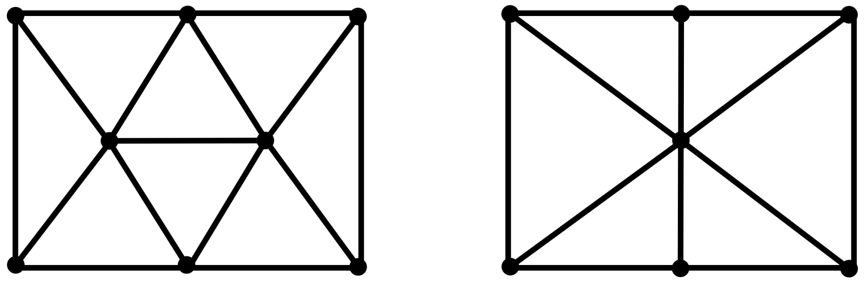


Figure 3.3: Edge collapse operation on the mesh graph

As with edge splitting, we treat only manifold edges, skipping edges incident on more than two triangles. We again use a subdivision scheme to choose the location of the new single vertex in the general case. However, we also use an eigendecomposition of the quadric metric tensor to detect vertices that lie on ridges or creases [Jia07]. 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 pseudo-motion collision detection described in Section 3.2.4, 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 pseudo-motion, 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 cannot find a collision-free trajectory, 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 ξ. This has the effect of keeping the edge lengths within some range of the initial average using split and collapse operations. In our examples in this chapter, 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 (see, for example, work by Jiao [Jia07]).

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* [DLG90] due to the simplicity of its implementation and because it is free of parameters. Quadric error minimization schemes [GH97] 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, $\vec{p}_{new}$ as:

$$\vec{p}_{new} = \frac{1}{16}(8(\vec{p}_1 + \vec{p}_2) + 2(\vec{q}_1 + \vec{q}_2) - (\vec{r}_1 + \vec{r}_2 + \vec{r}_3 + \vec{r}_4))$$

Where $\vec{p}_1$ and $\vec{p}_2$ are end points of the edge, $\vec{q}_1$ and $\vec{q}_2$ are the vertices on the two triangles incident on the edge which are not the edge end points, and $\vec{r}_1 ... \vec{r}_4$ are the vertices on the four triangles adjacent to the triangles incident on the edge (see Figure 3.4).

Null-space smoothing

A powerful mesh improvement technique was recently introduced by Jiao [Jia07]. Applying a Laplacian filter to the vertex locations would move each vertex to the average of its neighbours' 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, as was shown to be crucial by Jiao. If the vertex is on a flat or smoothly curved patch of surface, the null space

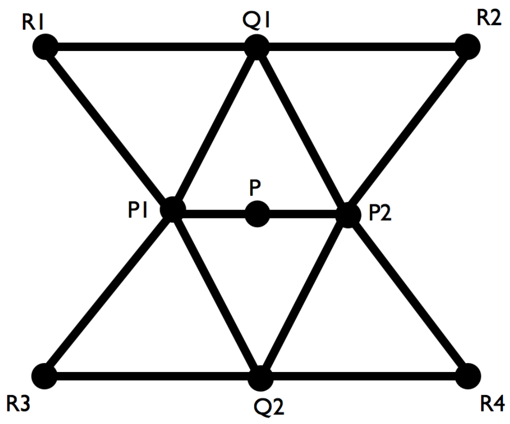


Figure 3.4: Butterfly subdivision

will correspond to the plane tangental 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 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 pseudo-trajectory on all vertices, and apply collision resolution as if the surface was moving under the influence of an external velocity field (see Section 3.2.7).

3.2.5 Mesh separation

As mentioned in Section 3.2.1, we do allow surfaces which are not strictly 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, which would define a zero-volume tetrahedron. An edge collapse or flip operation may introduce such 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, surfaces 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 con-

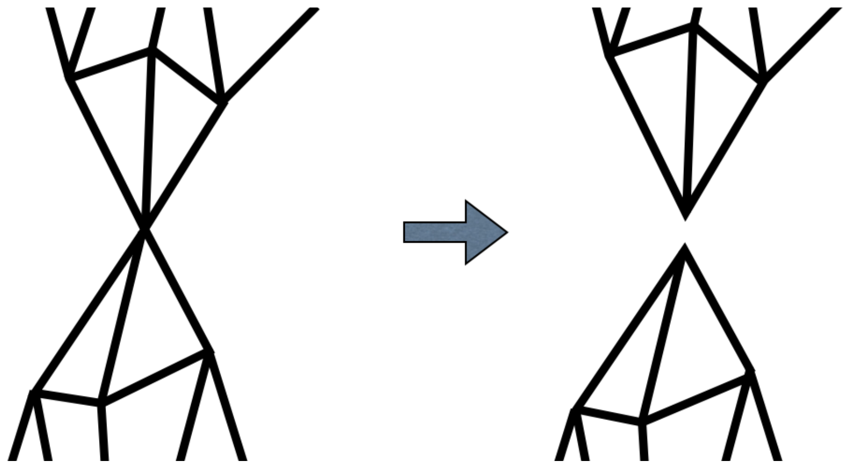


Figure 3.5: Mesh separation

nected components. For each component, we create a duplicate vertex and map all triangles in the component to this new vertex (see Figure 3.5). A similar procedure is described by Guéziec et al. [GTLH01]. 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. This combination of removing degenerate tetrahedrons and the “duplicate-and-separate” operation on singular vertices allows for mesh separation. If we wish to avoid topological changes, we modify the edge collapse and edge flip operations to check if any degeneracies or singular vertices would result and, if so, abort the operation.

An interesting alternative to this separation method is to detect when an edge collapse is occurring on a closed path of just three edges. Such an edge collapse would introduce a non-manifold edge if completed. Wojtan et al. [WTGT10] detect this scenario and, rather than allowing the non-manifold edge, instead duplicate the three-edge ring, and add a new triangle on each ring. This creates a clean splitting of surface meshes. Incorporating this into our library is an area of future work.

3.2.6 Mesh merging

We have described several mesh quality maintenance operations that can be performed without introducing geometric intersections. To make our general surface tracking algorithm useful for a wider range of applications (e.g. fluid simulation), we must allow surfaces to change topology. We have seen in Section 3.2.5 how our method allows meshes to separate when they become too thin. We now describe a method for allowing surface patches which are close to each other to merge without introducing any intersections.

To achieve this, we seek out edges that are close together and attempt to merge the surface. We use proximity detection to search for edges that are closer than a specified tolerance (considering only edges that are incident on two triangles). We 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. We then use intersection testing to determine if these 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.

Similar to edge collapsing and flipping above, this merge operation may introduce degenerate tetrahedra and singular vertices which must be handled as described in Section 3.2.5.

3.2.7 Collision resolution

After performing mesh improvement and any topological operations, all that remains is determining the surface velocity and integrating the vertex positions forward in a collision-free manner. We allow the user to specify predicted end positions for each vertex, and the goal of our time integration scheme is to produce a final mesh configuration that is as close to the specified state as possible, while being free of intersections.

Our collision resolution procedure is based on the *velocity filtering* approach for handling collisions [BFA02], and operates in three phases. For convenience, we compute a temporary velocity for each vertex as $\vec{u}_i = (\vec{x}_i^* - \vec{x}_i)/\Delta t$ and apply impulses to these velocities. (The collision resolution routines could operate only on the predicted vertex locations, but it is conceptually simpler to think in terms of velocities.)

In the first phase, we run proximity detection as described in Section 3.2.3 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 3.2.3, 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:

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

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

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

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:

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

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:

$$\delta = -\frac{\vec{n} \cdot \Delta\vec{x}_{\mathrm{rel}}}{\Delta t}$$

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 necessarily 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. [HVTG08]. 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:

$$\begin{aligned} &\min \left|\left|\mathbf{u}' - \mathbf{u}\right|\right|_M^2 \\ &\text{subject to } \mathbf{n} \cdot \mathbf{u}'_{\mathrm{rel}} = 0 \ \text{ for all collisions} \end{aligned}$$

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 $C_{i,\mathbf{j}} = \bar{a}_v \vec{n}^T$, our constrained optimization problem becomes:

$$\min \left|\left|\mathbf{u}' - \mathbf{u}\right|\right|_M^2$$
$$\text{subject to } C\mathbf{u} = \mathbf{0}$$

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.

If this system is degenerate, we compute a single rigid motion for the vertices of the offending impact zone, ensuring no collision. This was referred to as "rigid impact zones" in Bridson's original paper [BFA02].

At the end of each time step, we verify that no tangling has occurred by running intersection tests on all edge-triangle pairs; while not necessary for the method as presented, this is a useful practice for detecting programming errors during software development.

Error introduced by collision resolution

Each individual collision response and repulsion force perturbs the vertex location by $O(\Delta t)$. In applications such as cloth simulation (see Chapter 4), collision resolution forms an important part of the dynamics. However, for other surface tracking applications such as geometric flows and liquid simulation, which involve changes in topology, this vertex perturbation is not specified by the surface dynamics, so can be seen as error introduced by our method.

We posit that the number of collision events for a given vertex in any such numerically-resolved simulation should be small and finite for a fixed end-time, with the collisions in the limit becoming a set of measure zero. Since each collision perturbation has magnitude at most $O(\Delta t)$, this implies the collision resolution should introduce at worst a global $O(\Delta t)$ error, so we should achieve at least first order convergence. When surface elements collide or merge, we introduce an error similar to that introduced into the level set method by the kink in the signed distance field; at a fundamental level topological changes are non-smooth and unlikely to permit greater than first order accuracy in any numerical method.

As a caveat, we note that the $O(\Delta t)$ error produced by the impact zone solver could potentially involve a very large constant, since it will perturb the velocity field at many points. However, we have found that impact zones are seldom used in practice, after one pass of repulsion forces and three passes of impulse-based collision resolution. Adaptively cutting the time step size has been a successful strategy for reducing the number of collisions [BFA02], and could also be used in the case where impact zones grew too large. However, we shall see in the next section that our method shows convergence under mesh refinement, even without cutting the time step size: this error does not appear to be a concern in practice.

3.3 Numerical examples

Armed with techniques for guaranteeing intersection-free meshes, changing mesh topology, and maintaining mesh quality, we attack some dynamic surface problems traditionally handled by implicit surfaces.

3.3.1 Motion in the normal direction

We compare motion in the normal direction using an explicit method with a level set method. Points on the surface are advected according to:

$$\frac{d\vec{x}}{dt} = c(t)\vec{n}(\vec{x}), \tag{3.1}$$

for speed $c(t)$. We ran motion in the normal direction on two disjoint spheres with speed of 0.2 for $t = [0,1)$, then with speed of -0.2 for $t = [1,2]$. The spheres initially have centers at $(-0.25,0,0)$ and $(0.25,0,0)$, and radius of 0.2. We used Marching Tiles [Wil08] to generate a mesh with an initial average edge length of approximately 0.02. Let L_{max} be the maximum edge length, L_{min} be the minimum edge length, ΔV_{max} be the maximum volume change per remeshing operation, and ξ be the initial average edge length. We set $L_{max} = 1.5\xi$, $L_{min} = 0.5\xi$, and $\Delta V_{max} = 0.1\xi^3$. The time step Δt was generally set to 0.005, but was shortened for some time steps to avoid inverting triangles, as described below.

A first instinct may be to use a simple scheme for specifying normal motion, namely using vertex normals (averaged from incident triangle normals) to specify the direction of motion, then simply advecting the vertices according to this direction. However, this scheme does not produce the entropy solution, as it does not correctly handle flow fields with merging characteristics (as argued by Enright et al. [EFFM02], for example). To confirm this, we computed vertex normals as the area-weighted average of normals of incident triangles. We then ran our test, advecting the vertices according to these computed normals. Comparing against the analytic entropy solution revealed that this method does not converge as the mesh is refined. Figure 3.6 shows the results of this test on a high-resolution mesh.

Our explicit method instead uses the entropy solution of the *face offsetting* algorithm [Jia07] to achieve normal motion. For this advection scheme to function properly, we must guard against operations that will invert a patch on the surface. Thus we reject any edge collapse or edge flip operation that results in a triangle with a normal that is too different from the original triangle normals. We also adjust the time step to avoid inverting any triangles over a single face offsetting step, following the method described in the original face offsetting paper [Jia07].

Figure 3.7 shows our method at $t = 0, 1,$ and 2. Note that our method cleanly

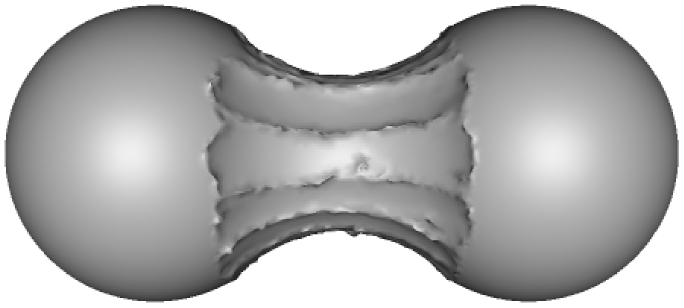


Figure 3.6: Motion in the normal direction using vertex normals

Initial average edge length	L_∞ error at $t = 2$	L_1 error at $t = 2$
0.04	0.00811073	0.00114493
0.02	0.00236906	0.00042705
0.01	0.00129011	0.000174606

Table 3.1: Motion in the normal direction: error measures for the new method.

handles merging of the two spheres. In this figure we compare our method against the level set method, using the Toolbox of Level Set Methods [Mit08]. The grid resolution used was $100 \times 50 \times 50$, resulting in a grid spacing of $\Delta x = 0.02$, similar to the average initial edge length of the triangle mesh. We used a WENO5 spatial derivative approximation, and a third-order accurate TVD RK time integrator with the same time step size of $\Delta t = 0.005$.

We also ran this example on lower- and higher-resolution initial meshes to determine convergence. Table 3.1 compares the initial average edge length to the L_∞ and L_1 error, obtained by comparing against the analytic exact solution for time $t = 2$. The results are consistent with first-order convergence, as shown in the log-log plot in Figure 3.8. We used the same initialization values for Δt, and set L_{max},

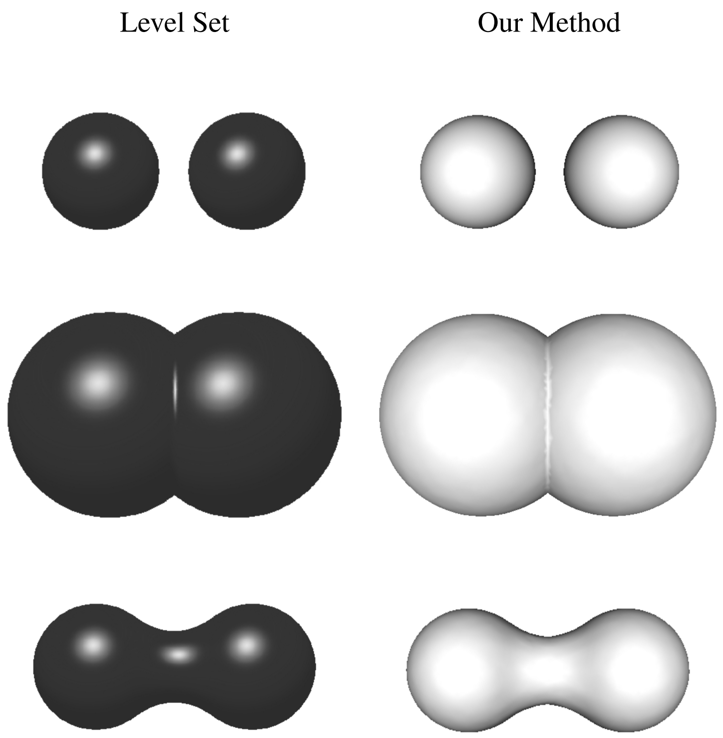


Figure 3.7: Motion in the normal direction

Grid spacing (Δx)	L_∞ error at $t = 2$
0.04	0.0118
0.02	0.0053
0.01	0.0031
0.005	0.0017

Table 3.2: Motion in the normal direction: error measures for the Level Set method.

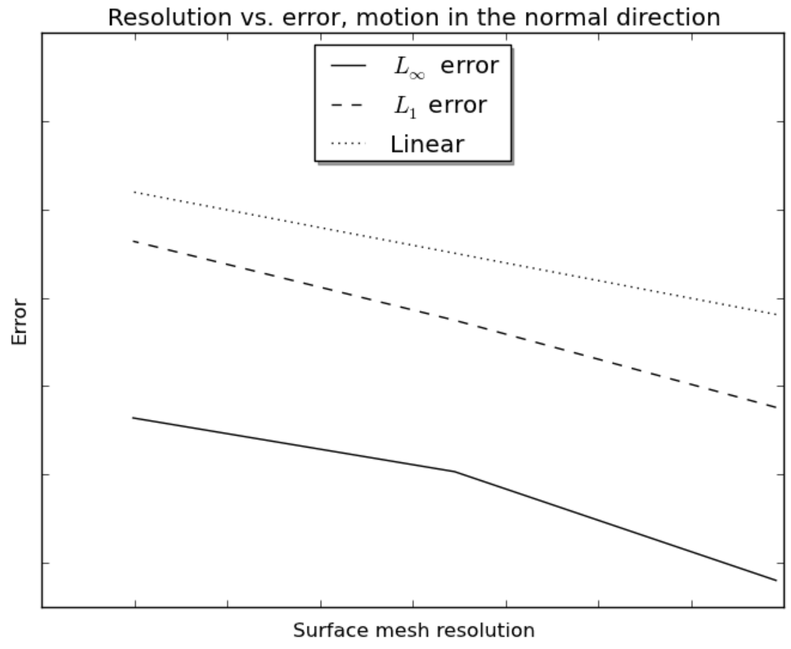


Figure 3.8: Log-log plot of error for motion in the normal direction

L_{min}, and ΔV_{max} according to the same fraction of ξ throughout. Table 3.2 shows a comparable rate of convergence when using the Level Set method with RK3 time integration and upwind WENO5 spatial derivatives. Note that despite the use of high-resolution numerical methods, the topology change reduces the level set method to first-order accuracy, due to the kink in the signed distance field at the merging event.

3.3.2 Motion by mean curvature

Motion in the normal direction with speed proportional to mean curvature is a classic geometric flow used for testing surface tracking methods, and is also useful in volume segmentation applications. Points on the surface move according to:

$$\frac{d\vec{x}}{dt} = \kappa(\vec{x})\vec{n}(\vec{x}), \tag{3.2}$$

Initial edge length	L_∞ error at $t = 0.025$	L_1 error at $t = 0.025$
0.04	6.18499×10^{-4}	2.62302×10^{-4}
0.02	1.44355×10^{-4}	7.02612×10^{-5}
0.01	1.47687×10^{-5}	1.54722×10^{-5}

Table 3.3: Motion by mean curvature: error measures.

for mean curvature κ.

We ran motion by mean curvature on a dumbbell-shaped surface, as described by Sethian [Set99]. This causes surface separation, as the dumbbell "handle" shrinks faster due to its higher mean curvature. We estimated mean curvature multiplied by the surface normal at each vertex using the scheme introduced by Meyer et al. [MDSB02], then moved vertices using simple Forward Euler time integration, with the maximum time step size limited for stability as $t_{\max} = \frac{1}{AW}$, where A is the speed of mean curvature flow, and W is the maximum area-weighting specified by the mean curvature normal computation over all vertices.

Initial conditions were generated by running Marching Tiles on a $28 \times 14 \times 14$ grid, generating an average initial edge length, ξ, of approximately 0.02. We again set $L_{max} = 1.5\xi$, $L_{min} = 0.5\xi$, and $\Delta V_{max} = 0.1\xi^3$.

Figure 3.9 shows a comparison with the level set method. For the level set example, we again use the Toolbox of Level Set Methods with a $100 \times 50 \times 50$ grid with $\Delta x = 0.02$, a second-order accurate spatial derivative scheme for estimating curvature, and a third-order accurate TVD RK time integration scheme with time step size dictated by the CFL condition.

To determine convergence in this case, we compare our solution against a high-resolution ($200 \times 100 \times 100$) solution produced by the Toolbox of Level Set Methods. Table 3.3 compare the initial average edge length to the error after integration. The results are consistent with first-order convergence (as shown in the log-log plot in Figure 3.10), which again is probably very hard to improve upon in the presence of topological change.

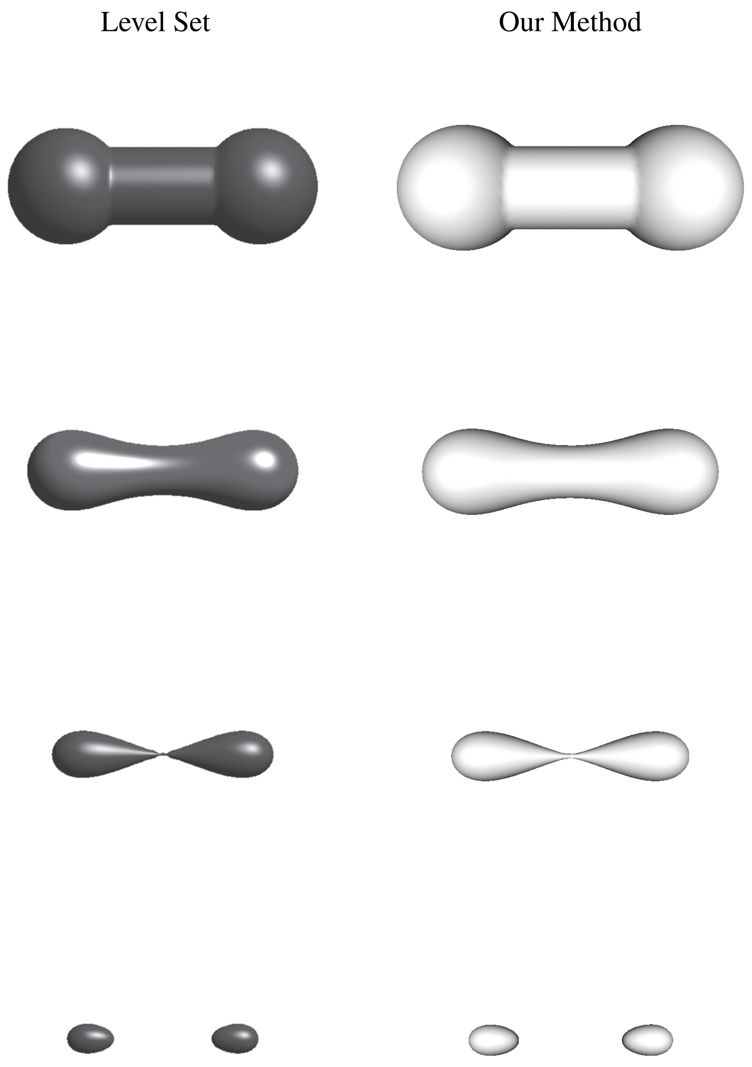


Figure 3.9: Motion by mean curvature at $t = 0$, $t = 0.0125$, $t = 0.0219$ and $t = 0.025$

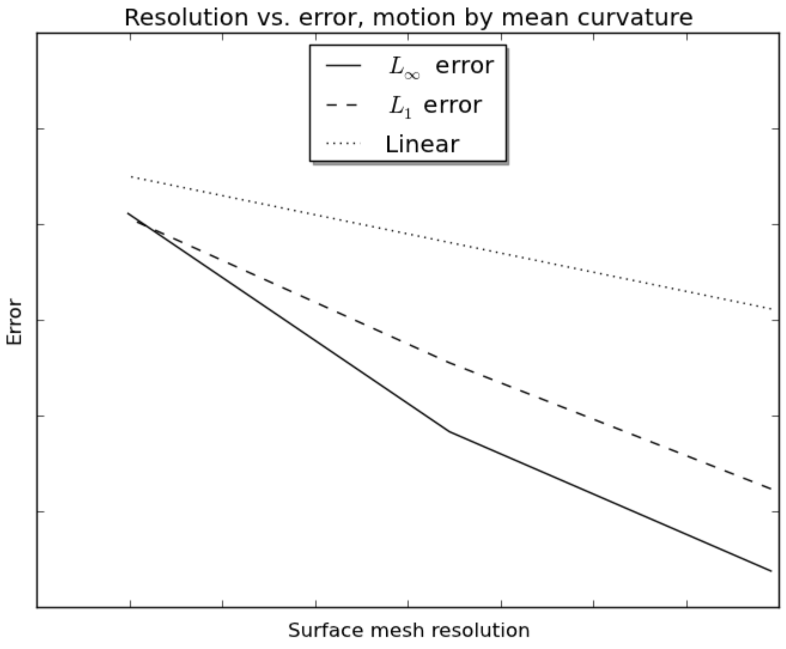


Figure 3.10: Log-log plot of error for motion by mean curvature

3.3.3 Motion by external flows

The Enright Test

We subjected our method to the *Enright test*, which was developed to test the accuracy of surface tracking methods [EFFM02] by measuring volume change. The initial surface is a sphere centred at $(0.35, 0.35, 0.35)$ with radius 0.15. The mesh is advected by the velocity field given by:

$$\begin{aligned} u(x,y,z) &= 2\sin^2(\pi x)\sin(2\pi y)\sin(2\pi z) \\ v(x,y,z) &= -\sin(2\pi x)\sin^2(\pi y)\sin(2\pi z) \\ w(x,y,z) &= -\sin(2\pi x)\sin(2\pi y)\sin^2(\pi z) \end{aligned}$$

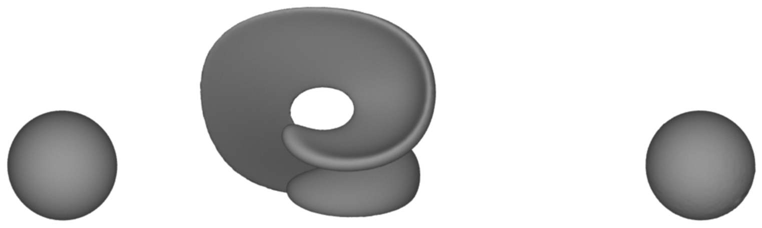


Figure 3.11: The "Enright test"

This velocity field is modulated by the term $\sin(\frac{2}{3}\pi t)$ to achieve a smooth, periodic motion. The initial mesh is generated by Marching Tiles from a $14 \times 14 \times 14$ grid, generating an average initial edge length of $\xi = 0.01$. We initialize the mesh maintenance parameters as $L_{max} = 1.5\xi$, $L_{min} = 0.5\xi$ and $\Delta V_{max} = 0.1\xi^3$. We use a 4$^{\text{th}}$-order accurate Runge-Kutta scheme to advect the mesh vertices, with a time step of 0.01. After one period of motion (see Figure 3.11), the volume enclosed by the surface has changed by just 2.48899×10^{-5}, resulting in a relative error of 0.1764 percent. (Note that standard level set methods fail completely on this test, and require additional effort such as the particle level-set method to resolve the surface [EFFM02].)

To illustrate the effects of our mesh adaptivity operations, we also ran the Enright test with various mesh maintenance operations turned off. Figure 3.12a shows a mesh where no edge splitting had been performed. Note that the resulting large triangles poorly capture the curvature of the surface. Turning off edge collapse generates high triangle density when the surface is contracting, as in figure 3.12b, resulting in wasted computational effort. Figure 3.13 shows the non-uniform triangulations resulting when edge flipping and null-space smoothing operations are turned off. Since the patch shown has relatively low curvature, a uniform triangulation is desirable, but without flipping and null-space smoothing, the resulting triangulation is irregular.

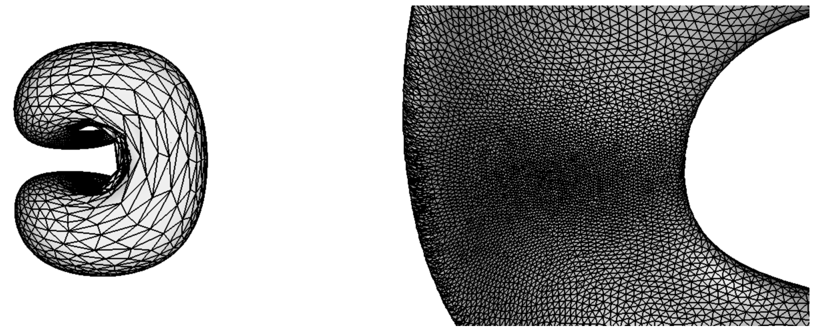

(a) No edge split (b) No edge collapse

Figure 3.12: Effect of edge splitting and collapsing

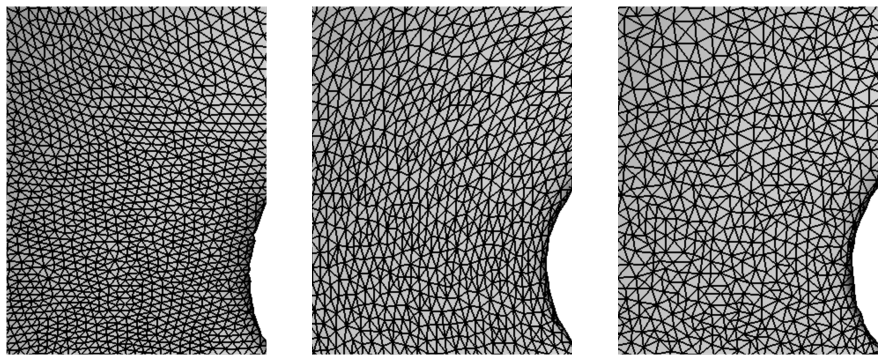
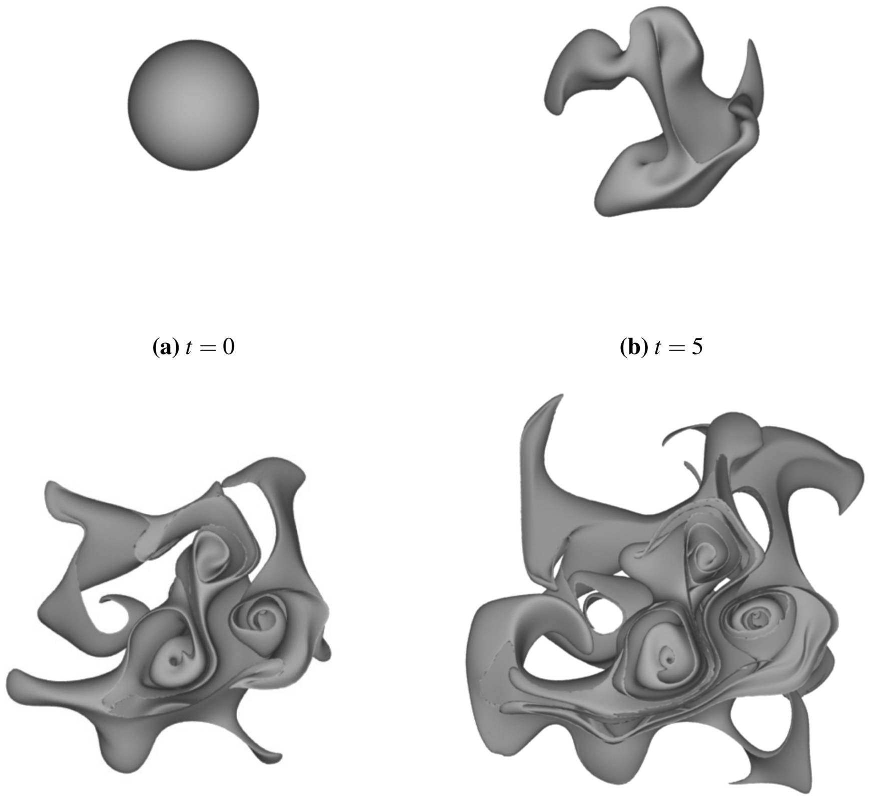

(a) All operations (b) No edge flip (c) No smoothing

Figure 3.13: Effect of edge flipping and vertex smoothing

Curl Noise

We also advect, with RK4 time integration, an initially spherical surface with a smooth, pseudo-random, divergence-free velocity field using curl noise [BHN07] (see Section 5.4.2). It would be very challenging for a grid-based, implicit method with similar resolution to resolve the extremely thin structures which are shown in Figure 3.14. We restrict the rotational motion to be planar by generating a random spline potential with zero x- and y-components and taking the curl of this potential. We do this only to aid visualization, eliminating the occlusion that occurs when



(a) $t = 0$ **(b)** $t = 5$

(c) $t = 15$ **(d)** $t = 30$

Figure 3.14: Motion by curl noise

using a fully three-dimensional rotation field. We disallowed topological changes and set the maximum allowed change in volume, ΔV_{max}, to be very small ($5 \times 10^{-4}\xi^3$) to faithfully capture the thin structures. At time $t = 30$, the total volume enclosed by the surface has changed by 0.9211 percent.

We ran this test a second time with collision detection turned off, allowing mesh elements to intersect. Notice that with a smooth velocity field defined everywhere

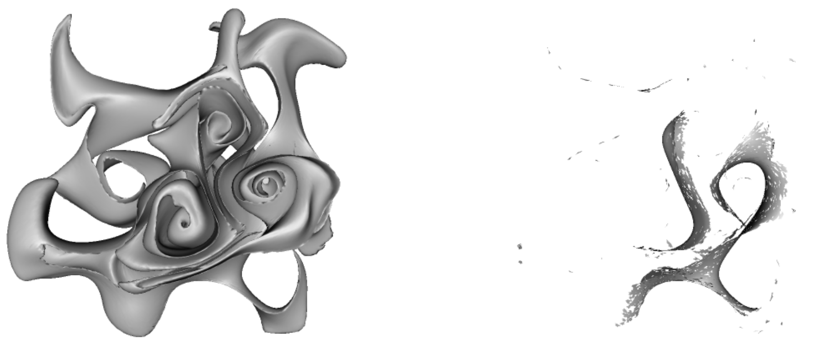

(a) Surface **(b)** Intersecting triangles

Figure 3.15: Motion by curl noise with no collision resolution

in space, the surface should never self-intersect, but—even with a high-order time integration scheme—self-intersections do occur due to discretization into triangles, numerical error and collision-oblivious mesh improvement operations. Figure 3.15 shows one screen capture showing the entire mesh, and one showing only the set of intersecting triangles.

3.3.4 Extension to open surfaces

With some modification, we can extend our algorithm to handle open surfaces. Figure 3.16 shows a curl noise velocity field advecting an open surface. In such scenarios, we must be careful when dealing with boundary edges, as they are incident on only one triangle and our remeshing operations assume edges are incident on two or more triangles. In our example we handle these edges in the simplest way possible: we disallow flipping, collapsing and splitting of any such edges, as well as topology changes. The remaining triangles and edges on the surface are unchanged. (We note that with minor implementation effort, the boundary edges could be refined and coarsened as usual.)

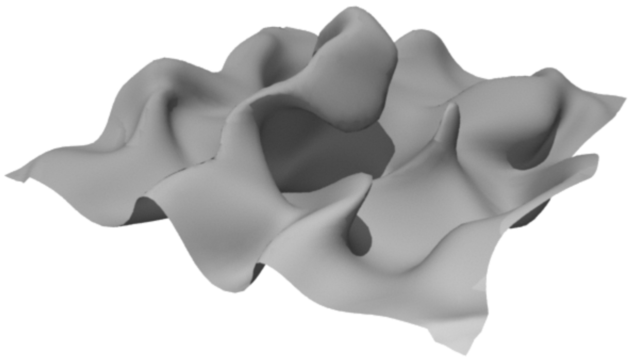


Figure 3.16: Motion by curl noise on an open surface

The ability to handle open surfaces suggests a possible further extension to periodic surfaces, although we have not yet attempted to implement this. Another extension would be allowing an odd number of triangles incident on an edge in order to represent, for example, a solid-fluid-air triple point in a fluid simulation.

3.3.5 Performance

Table 3.4 shows timings for our method running the motion in the normal direction example. We list the time for setting predicted vertex positions, detecting and handling collisions, performing topological operations and improving mesh quality per time step. All computations were performed on a single core of a 2.4 GHz Intel Core2 Duo with 4 GB main memory. We have not attempted a parallel implementation, but as the majority of our operations are local in nature, it should be possible to spread the work over several processors.

The timings for topology change and mesh improvement operations include the time taken for intersection and collision queries to ensure collision safety. The speed of such interference detection depends greatly on the broad-phase collision culling strategy. We use a simple regular grid of bounding boxes for each mesh

ξ	Triangles	Improvement	Topology	Set velocity	Collisions
0.04	1476–3512	0.1104	0.04171	0.04031	0.06529
0.02	5896–14222	0.5203	0.2115	0.1602	0.3123
0.01	23372–56390	2.07845	0.845781	0.617641	1.41106

Table 3.4: Execution time per step (in seconds), for given initial average edge lengths, ξ

element type (vertex, edge, and triangle), which theoretically scales linearly with the number of elements. We do achieve linear scaling in the number of objects tested after broad-phase culling but only near-linear scaling in execution time. Investigating and optimizing our broad-phase algorithm to achieve linear scaling in actual execution time is an area of future work.

Table 3.5 lists the minimum and maximum numbers of triangles for each example, as well as the total run time. Where appropriate, we also list the grid resolution of the level set grid used for comparison. We did not run the Enright test using the level set method, but we include the resolution of the grid used in the original particle level set paper [EFFM02] for comparison. Note that in the particle level set method, the grid is augmented with marker particles (64 particles in each grid cell located within 3 cells of the initial interface), effectively increasing the resolution of their method even further.

Since we are using a MATLAB implementation of the level set algorithms, direct comparison of timings against our un-optimized C++ implementation is difficult, but we include run times for a very general idea of how our method compares. We ran the level set examples for motion in the normal direction and motion by mean curvature on a Sun x4600 M2 with 4 dual core Intel x64 processors (at 2.8 GHz) and 128 GB shared main memory.

3.4 Conclusion

In this chapter, we presented a method for robustly handling topological changes in surfaces represented as triangle meshes, addressing one of the major obstacles in using such explicit surface tracking methods. The use of robust interference detec-

Example	Triangles	Run time	LS resolution	LS run time
Normal direction	5896–14251	617s	$100 \times 50 \times 50$	1435s
Mean curvature	3354–15468	1537s	$100 \times 50 \times 50$	666s
Enright test	6614–25176	597s	$100 \times 100 \times 100$	N/A
Curl noise	8798–381092	608m	N/A	N/A

Table 3.5: Mesh resolution and run time

tion, topological operations, and failsafe collision handling provides the framework for guaranteeing intersection-free surfaces while still allowing merging and separation. We presented a collection of mesh maintenance operations to improve the quality of the surface discretization. Finally, we presented results of numerical experiments, comparing our method to the level set method for geometric flows.

Chapter 4

Efficient geometrically exact continuous collision detection

4.1 Introduction

For the purpose of this thesis, we define continuous collision detection (CCD) as the process of detecting if a mesh moving between initial and final configurations comes into contact with itself at any point in time. CCD is a critical element of many algorithms for physical simulation, when a non-intersecting mesh invariant must be maintained, and for path planning, when the feasibility of a path must be guaranteed. Beyond efficiency, a good CCD algorithm should therefore be *safe*: no false negatives, i.e. missed collisions which can lead to self-intersection, can be tolerated. It should also be *accurate* in the sense of minimizing the number of false positives, i.e. non-collisions being flagged as collisions, for the effectiveness and efficiency of algorithms using CCD.

As discussed in the introduction, a *geometrically exact* test returns the correct result as if computed with exact arithmetic. The typical route to geometrically exact methods involves *predicates*, a set of simple Boolean tests on the input geometry whose answers determine the final outcome. With floating-point filters and, if necessary, exact arithmetic, some predicates can be computed exactly. This has been done for the exact evaluation of intersection testing in static geometry—for example, detecting if an edge intersects a triangle in a triangle mesh.

Predicates stand in contrast to *constructive* methods, where intermediate real quantities are computed—for example, where along an edge the intersection with a triangle's plane occurs—and subsequently used. Given that intermediate quantities may not even be representable in floating-point arithmetic, let alone easily computed without error, constructive methods are extremely difficult to make robust without resorting to expensive symbolic computation.

In this chapter, we move from *intersection* to *collision* detection, where the problem becomes detecting whether moving objects intersect at any point over some time interval. We will limit our discussion to moving triangle meshes, and assume that mesh vertices move on straight-line, constant-velocity paths. In this chapter we will introduce two CCD methods based on existing and new geometrically exact predicates. Contributions for this chapter include:

1. a fully robust CCD approach based on intersection testing of space-time simplices, which can be computed with exact arithmetic;

2. a linear solver which, given floating point inputs, can determine exactly if the linear system has a unique solution, and provide an estimate for the solution if it exists;

3. a second CCD algorithm using the same geometric model as the classic cubic solver approach, but which transforms the problem into a series of 3D predicates which can also be evaluated with exact geometric computation;

4. a new, efficient, geometrically exact ray vs. bilinear patch intersection parity test, which can be used to precisely determine if a point is inside a quad-mesh-bounded volume or not;

5. a new adaptive cloth simulation method which maintains intersection-free meshes despite remeshing, as an example of the practical advantage of geometrically exact CCD, and show there is no performance penalty for using geometrically exact CCD.

We restrict our attention to triangle meshes in 3D, with an intersection-free initial configuration, so CCD can be reduced to two primitive tests: does a moving point hit a moving triangle, or does a moving edge hit another moving edge?

Note that we ignore the connectedness of the mesh: multiple meshes are treated as lumped together, so there is no difference between inter-object collision and self-collision.

4.2 Related work

4.2.1 The cubic solver approach

The most popular current method for continuous collision detection for triangle meshes was introduced by Provot [Pro97]. First a cubic equation is solved to determine coplanarity times, then the interpolated geometry is checked for interference at these times to determine if a collision actually occurs. Bridson et al. [BFA02] significantly reduced the number of false negatives due to floating-point error by introducing error tolerances in the root-finding algorithm used to solve the cubic equation and using a static distance query at the coplanarity times: collisions are reported if the mesh elements are within some small distance of each other.

However, the minimum error tolerances required for safe CCD are difficult to predict in advance. Especially in cases where the primitives remain nearly coplanar for the entire step, such as hair segments [SFK$^+$08] sliding against each other on skin, cancellation error in simply computing the coefficients of the cubic can eliminate almost all precision in the rest of the calculation. (Of course, in constantly coplanar cases, the method breaks down entirely.) Even if the cubic is represented exactly, its roots are in general irrational and must be rounded to floating-point numbers. Completing the error analysis with further bounds on the construction of the intermediate geometry at the rounded coplanarity time, bounds on the calculated barycentric coordinates of closest points, and then bounds on the distance appears intractable. In practice, a usable tolerance can typically be found by trial and error for a large class of similar simulations (e.g. cloth animations), but different applications such as adaptive cloth, hair, or liquid surface tracking can require enervating per-simulation adjustment, which makes writing a general purpose library especially tricky.

The cubic solver approach naturally gives false positives if the tolerance is high enough to work. If the tolerance is too high (a definite possibility if restricted to single precision arithmetic, for example) this can seriously slow down or even completely stymie collision resolution: tuning the tolerance for a new simulation isn't always easy.

A fully symbolic implementation could in principle resolve the above problems, apart from the degenerate constantly coplanar case, but the computational overhead would be drastic. In this chapter we show a different approach to CCD can be fully safe and accurate without need for tuning, yet run just as fast.

4.2.2 Other work in continuous collision detection

Stam [Sta09] extended the cubic solver approach to explicitly test if two mesh elements approach closer than a given distance during the time step, resulting in a sixth degree polynomial to solve for potential collision times. This helps to resolve the coplanar-motion degeneracy mentioned above, but poses an even less tractable rounding error analysis problem for safe CCD, suffers from the same false-positive issues, and is a heavier burden computationally.

Alternative methods for computing the time of possible collisions, such as conservative local advancement [TKM10] offer potential speed-ups over the cubic solver approach, but do not robustly deal with rounding error, and must rely on user-set tolerances to account for slight non-planarities in intersection/proximity testing.

The constant vertex velocity model underlying this work and the cubic solver approach is perhaps the most natural for general deformable motions. However, for rigid bodies, constant linear and angular velocity of the entire model makes more sense — although the helical trajectories of vertices are somewhat more difficult to handle. Zhang et al. [ZRLK07] demonstrate significant acceleration of conservative advancement using the Taylor model generalization of interval arithmetic, but again rely on user-set tolerances to cope with the inexact solve and rounding error.

Raytracing can be seen as a special case of CCD, generally easier since the geometry is static relative to the "motion" of the light ray. We highlight Ramsey et al.'s ray-bilinear patch test [RPH04] as particularly relevant, as our 3D CCD test in fact relies on ray-bilinear patch intersection parity tests. However, our new approach is geometrically exact and fully robust, unlike Ramsey et al.'s constructive approach, which is vulnerable to rounding error; on the other hand, our test only provides the parity of the number of intersections, not their location.

The related problem of culling collision tests is very well studied in computer graphics. Several approaches using bounding volume hierarchies have been proposed, as well as culling using bounding boxes with regular grids, sweep-and-prune testing, and sweeping-plane testing: see Ericson's book for example [Eri04]. We observe that except for axis-aligned methods which only use comparisons (no arithmetic), the culling literature generally does not worry about verifiably handling rounding error. We briefly address this issue later, but emphasize our focus is the correctness of the core element vs. element test, not the efficiency of broader culling methods.

4.2.3 Exact geometric computation

An alternative to constructive geometric algorithms discussed so far is to decompose a geometric test into a set of simpler *predicates*, providing discrete answers such as “does a point lie to the left, to the right or on a line?” rather than continuous values. Approximate continuous values may be computed alongside, of course, but the discrete correctness of the algorithm as a whole relies only on the correctness of the discrete answers from the predicates. Several approaches to defining and implementing correct predicates exist; the most successful is the paradigm of Exact Geometric Computation (EGC). We recommend Yap's article as an excellent review of the topic [Yap04]. In brief, a *geometrically exact* predicate must return the same discrete answer as if computed with exact arithmetic (from the floating point input) even if under the hood it takes a faster approach. Our method is the first geometrically exact CCD test for general CCD, but exact predicates for simpler problems have long been used in graphics and elsewhere.

Building on previous work by Dekker [Dek71] and Priest [Pri91], Shewchuk presented practical systems for exactly evaluating a number of geometric predicates needed for Delaunay mesh generation [She96]. These sign-of-determinant predicates are equally useful for detecting self-intersections for triangle meshes. When higher precision than provided by floating point hardware is required, the system uses *floating-point expansions* (the sum of a sequence of floats) leveraging fast floating-point hardware even for exact arithmetic.

In Section 4.5 we decompose CCD into a set of simplex intersection tests, based on the same standard sign-of-determinant tests, together with the evaluation of the sign of a simple polynomial function. No radicals or even divisions are required, making it straightforward to implement exactly using expansion arithmetic like Priest and Shewchuk. Furthermore, through the use of fast interval arithmetic filters, we can rapidly find the provably correct signs without need for high precision expansions in all but the most extreme cases, leading to highly efficient execution on average.

4.2.4 Adaptive cloth simulation

In section Section 4.5.6, we present an adaptive FEM cloth simulation to demonstrate the robustness of our CCD approach. Although mass-spring systems are popular for cloth simulation due to their simplicity and ease of implementation, implementers of cloth animation systems have been turning to increasingly sophisticated models in recent years. For example, the Finite Element Method (FEM) can achieve accurate results and is less dependent on the mesh structure than using edge-based springs [EKS03].

In the related sub-field of simulating volumetric elastic solids for graphics, it is becoming increasingly popular to perform on-the-fly optimization of the volumetric simulation mesh [BWHT07, WT08, WRK$^+$10]. The benefits of remeshing include reducing error for highly-sheared elements and concentrating computational effort where it is needed to resolve small-scale details. For cloth, an additional important benefit of remeshing is to increase vertex density in regions of high curvature so that curved regions can be accurately represented without having to globally refine the surface mesh.

A few authors have suggested refining the simulation elements for cloth [LV05, VB05], however, the idea has not been as popular for cloth as it has for solid elasticity. One reason for the lack of uptake is the difficulty in dealing with collisions. For example, adding and removing vertices without introducing self-intersections is a major concern if continuous collision detection assumes that the mesh is free of self-intersections at the beginning of each time step. Adding and removing vertices and altering the triangulation at discrete times compounds the difficulty in choosing suitable collision and intersection error tolerances.

4.3 Space-time simplex continuous collision detection

In this section, we propose a collision detection approach which considers time as an additional spatial variable. Discretizing the motion of mesh elements in space time with simplices, the continuous collision test becomes a $d+1$ simplex intersection test, where d is the dimensionality of the original problem. As we are now dealing with simplices, intersection testing is a linear problem, which can be solved in a geometrically exact fashion either using matrix determinants (Cramer's rule), or with an exact linear solver.

First consider CCD in two spatial dimensions. The fundamental collision test is point-vs-segment. We are given the vertex and segment endpoint locations at the beginning and end of the time step, and must determine if the vertex lies on the segment at any time in the time step. In the following we will index the moving vertex with 0, and the segment end vertices as 1 and 2. We denote the location in space of vertex i at the beginning of the time step as $\vec{x}_i$, and its location at the end of the time step as $\vec{x}_i^*$. The input to our continuous collision detection is then the set of six 2D points: $\vec{x}_0$, $\vec{x}_1$, $\vec{x}_2$, $\vec{x}_0^*$, $\vec{x}_1^*$, and $\vec{x}_2^*$ (see Figure 4.1).

The moving vertex then sweeps out a path in space-time which is a static 3D line segment, and the moving line segment sweeps out a bilinear patch in 3D (see Figure 4.2). The 2D continuous collision test then becomes a 3D *intersection* test, between the line segment and bilinear patch.

We further propose approximating the bilinear patch with two flat triangles (see Figure 4.3). This further reduces the CCD problem to testing a specially structured line segment against two separate triangles in three dimensions, which is much more tractable for rounding error analysis. This discretization is water-tight, and can be made conservative by computing a bound on the floating-point error incurred, or by evaluating exactly with extended precision arithmetic if necessary.

Let (x_0, y_0) and (x_0^*, y_0^*) be the coordinates of the point at the start ($t=0$) and end ($t=1$) of the time step, or in our 3D view $(x_0, y_0, 0)$ and $(x_0^*, y_0^*, 1)$. Let the edge's endpoints be (x_1, y_1) and (x_2, y_2) at $t=0$, and (x_1^*, y_1^*) and (x_2^*, y_2^*) at $t=1$.

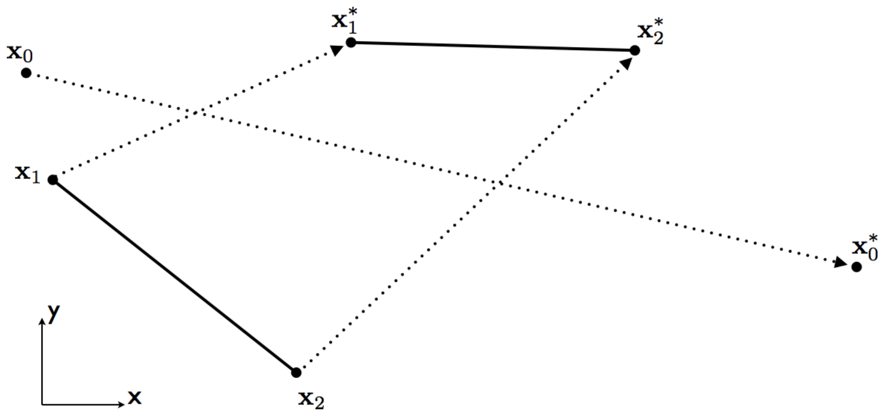


Figure 4.1: Discretization of 2D continuous collision detection problem.

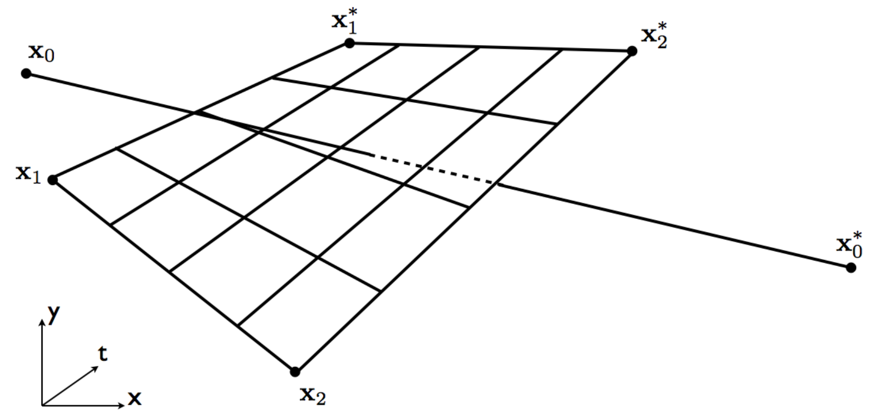


Figure 4.2: Space-time trajectories, a line segment and a bilinear patch.

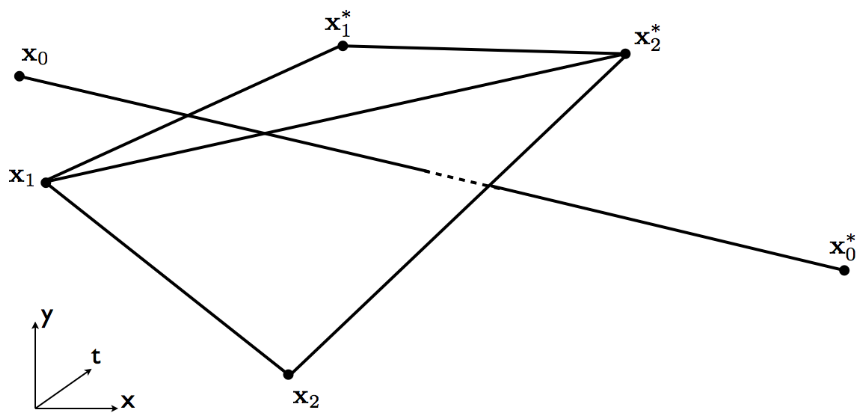


Figure 4.3: Space-time trajectories discretized as a line segment and pair of triangles.

Picking one diagonal arbitrarily, the two triangles representing the edge's swept trajectory are $(x_1,y_1,0):(x_2,y_2,0):(x_2^*,y_2^*,1)$ and $(x_1,y_1,0):(x_1^*,y_1^*,1):(x_2^*,y_2^*,1)$.

Focus on the first triangle. We set up the intersection problem as a linear system, looking for $(s,t,\alpha,\beta,\gamma)$ satisfying:

$$\begin{aligned} sx_0+tx_0^*+\alpha x_1+\beta x_2+\gamma x_2^* =&\ \ 0 \\ sy_0+ty_0^*+\alpha y_1+\beta y_2+\gamma y_2^* =&\ \ 0 \\ t+\gamma =&\ \ 0 \\ s+t+\alpha+\beta+\gamma =&\ \ 0 \\ s+t =&\ \ 1 \end{aligned} \tag{4.1}$$

The last two equations define the pair (s,t) as barycentric coordinates along the line through the point's trajectory, and the triple $(-\alpha,-\beta,-\gamma)$ as barycentric coordinates in the plane of the triangle. The first three equations then express the intersection of the point's trajectory with the triangle, in x, y, and t in order. There is an intersection if and only if there is a solution to this linear system where s and t are both non-negative, and α, β, and γ are all non-positive—i.e. the intersection lies within the triangle and in the time interval $[0,1]$.

In Section 4.4, we discuss methods for exactly determining if there is a unique solution to this linear system, and providing an estimate if there is. If there is no solution, the segment misses the triangle, and if there are an infinite number of solutions, the segment lies coplanar to the surface, and intersects it.

Additional information about the collision is easily collected from the barycentric coordinates, once computed. For example, the time of collision (between 0 and 1) is simply the coordinate t; the normal vector at the collision is either the $t=0$ edge's normal or the $t=1$ edge's normal depending on which triangle was hit; the relative normal velocity at the collision is defined from the point's velocity and the velocity of the endpoint of the edge that is entirely contained in the colliding triangle. (Special cases must be made for degenerate situations, sometimes leading to an ambiguity which we arbitrarily resolve: e.g. a degenerate edge has no defined normal.)

4.3.1 Space-time simplex CCD in three dimensions

To handle CCD in two dimensions, we essentially built a triangle mesh stretching between the initial and final configurations in 3D space-time, and then ran robust self-intersection specialized to this type of mesh construction. We do exactly the same in three dimensions, building a tetrahedral mesh stretching between the initial and final configurations in 4D space-time. We model a moving point by sweeping out a straight line segment from $t=0$ to $t=1$ and a moving edge with two triangles as before. A moving triangle with vertices labeled 1, 2, and 3 sweeps out a triangular prism discretized into three tetrahedra, as in Figure 4.4:

$$\vec{x}_1 : \vec{x}_2 : \vec{x}_3 : \vec{x}_3^*$$
$$\vec{x}_1 : \vec{x}_2 : \vec{x}_2^* : \vec{x}_3^*$$
$$\vec{x}_1 : \vec{x}_1^* : \vec{x}_2^* : \vec{x}_3^*$$

where $\vec{x}_i = (x_i, y_i, z_i, 0)$ and $\vec{x}_i^* = (x_i^*, y_i^*, z_i^*, 1)$.

Note that to avoid cracks in the tetrahedral space-time mesh, a consistent choice of diagonals for the triangles must be made: in our code, we first sort the vertices

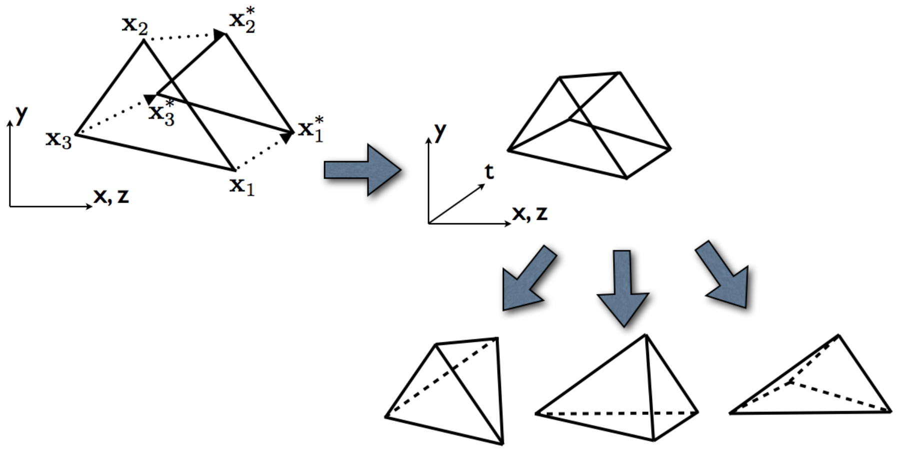


Figure 4.4: For collisions with a triangle in 3D, we discretize the triangular space-time prism of the triangle's swept trajectory with three tetrahedra as shown (with x and z axes projected together).

by index before constructing the triangles or tetrahedra in a call to make sure this happens.

With this discretization, the moving edge vs. moving edge tests is reduced to four triangle vs. triangle intersection tests in 4D (each edge sweeps out two triangles), and the moving point vs. triangle test is reduced to three segment vs. tetrahedron intersection tests in 4D. The intersection of two such 4D simplex primitives can be described with six barycentric coordinates that solve a linear system analagous to the one presented above, robustly evaluated in exactly the same fashion.

4.3.2 Discussion

Our approximation of higher-dimension objects with simplices leads to a somewhat unintuitive model for the mesh geometry at intermediate times: mesh edges develop kinks and triangles develop folds; normals do not vary continuously over time. This in turn raises problems for standard collision resolution methods, and precludes the use of CCD culling techniques which assume the geometry is linearly interpolated at intermediate times [TMT10]. In Section 4.5, we present a CCD method which is also geometrically exact, but maintains the same model of intermediate geometry as the traditional cubic-solver approach.

4.4 Exact solution of linear systems

Here we discuss the solution of the linear system (4.1). This system may have a unique solution, or be over- or under-determined.

4.4.1 Cramer's rule

The usual Computational Geometry approach to determining the signs of components of the solution is to use Cramer's rule. Assuming the matrix is invertible for now, Cramer's rule expresses the components of the solution as ratios of determinants formed from the matrix and right-hand side. Since we only care about signs, we can take out the common denominator to get the solution scaled by the determinant of the matrix. For example, the scaled s is the determinant of the matrix with its first column replaced by the right hand side:

$$\hat{s} = \det \left(\begin{array}{c|cccc} 0 & x_0^* & x_1 & x_2 & x_2^* \\ 0 & y_0^* & y_1 & y_2 & y_2^* \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{array} \right) \tag{4.2}$$

Similarly $\hat{t}$, $\hat{\alpha}$, $\hat{\beta}$, and $\hat{\gamma}$ can be expressed as 5×5 determinants. These determinants can be substantially simplified by cofactor expansion. (We note these determinants are often given geometric meaning in terms of signed volumes of tetrahedra, or triple products of certain vectors, but extending this to higher dimensions taxes the intuition.) These can be computed using only multiplication, addition and subtraction—no square or cube roots or even divisions are needed. If using fixed-point or integer coordinates with p bits, the exact answers only require $2p+4$ bits. We instead use floating-point expansion arithmetic to get an exact solution, similar to Shewchuk's method [She96].

It is a common occurrence in a typical simulation that we will end up a singular matrix, where both $\hat{s}$ and $\hat{t}$, or all of $\hat{\alpha}$, $\hat{\beta}$, and $\hat{\gamma}$ evaluate to zero: geometrically this corresponds to a degenerate configuration, such as parallel motion or a zero-length edge. We handle the degenerate case robustly by checking the moving point's path against both "time" edges of each triangle. These smaller segment vs. segment checks begin by checking coplanarity with the appropriate already-evaluated determinant. If the two edges are not coplanar in 3D they cannot collide, and otherwise we check for collision by projecting the segments onto the $x-t$ and the $y-t$ planes and running similarly robust 2D segment vs. segment tests. There is a collision if and only if both 2D tests report a collision. If either of the 2D tests hits a degeneracy, we again reduce the dimension by checking the endpoints against the other segment, and so on.

Since the use of floating-point expansions can be much slower than hardware-supported floating-point arithmetic, we use a "filter" approach to avoid using expansions unless they are necessary to achieve the required degree of precision. Our filter is a simple interval arithmetic filter: we evaluate the determinants using interval arithmetic first (roughly twice as expensive as straightforward floating-point arithmetic), and go to expansion arithmetic only when the interval contains zero (i.e. the sign cannot be accurately determined).

4.4.2 Gaussian elimination

An interesting alternative to using Cramer's rule is to perform Gaussian Elimination on this linear system. Performing division on floating-point expansions, however, is not possible in general, since not all quotients of floating point numbers can be represented as expansions (for example, $1/3$). This suggests that we cannot straightforwardly implement a Gaussian Elimination linear solver using expansion arithmetic in place of floating-point.

However, there is a class of direct linear solvers, developed for integer-valued problems, that do not perform any division until the very last step (called Division-Free Gaussian Elimination). There is also another class of algorithms for integer problems that *do* perform division, but every division (up to a final scaling of the solution) is known to produce an integer result. These are termed Fraction-Free

Gaussian Elimination (FFGE) [Bar68, NTW97]. By using floating-point expansions in place of integers in these algorithms, we can perform exact division, since we know beforehand that the results will be representable as expansions. Although the final results are not fully accurate due to the final floating-point division, the *sign* of each solution component is known with certainty, which is all we require for fully robust CCD.

The linear system (4.1) may be singular, but nonetheless may be consistent. To detect this, we allow the linear solver to row-reduce the matrix to upper-triangular form. If the $n \times m$ matrix has rank $n-k$, and if the last k entries of the solution are zero, then the system is consistent. If it is inconsistent, we quit and report no intersection (e.g. in our segment-triangle case, the segment and triangle would be in distinct, parallel planes). If it is consistent then we iteratively remove columns from A, checking if there is a solution for each reduced system. Geometrically, this corresponds to a degenerate configuration, for example a line segment and triangle being coplanar. Removing columns from A corresponds to projecting out one coordinate at a time, and checking for a separating line in each projection. If there is a separating line in any of the projections, we know the segment and triangle do not intersect in the original space, since an intersection would be present in any projected space.

We have implemented FFGE using floating-point expansions and have found it to be very robust and quite a bit easier to debug than manipulating the long expressions usually present in computing large determinants. However, designing floating-point filters is more challenging in this case than when using determinants, and we have not been able to achieve the same computational efficiency. We plan to investigate better filter design and other optimizations to make this solver more tractable in the future.

4.5 Root-parity based CCD

Recall that in 3D CCD, there are two fundamental collision tests: point-triangle and segment-segment, and the inputs to each test are the locations of the vertices at the beginning and end of the time step. We denote the location in space of vertex i at the beginning of the time step as $\vec{x}_i$, and its location at the end of the time step as $\vec{x}_i^*$. Then for each test, we are given 8 points: $\vec{x}_0$, $\vec{x}_1$, $\vec{x}_2$, $\vec{x}_3$, $\vec{x}_0^*$, $\vec{x}_1^*$, $\vec{x}_2^*$, and $\vec{x}_3^*$. For convenience, we will normalize the time step so that $t \in [0,1]$. The constant velocity model of motion gives the location of a vertex at an intermediate time as $\vec{x}_i(t) = (1-t)\vec{x}_i + t\vec{x}_i^*$.

First consider the point-triangle test. In the following we will index the moving vertex with 0, and the triangle vertices as 1, 2, and 3. Any point on the triangle can be written as: $\vec{x}(u,v) = (1-u-v)\vec{x}_1 + u\vec{x}_2 + v\vec{x}_3$, where $\vec{x}_1$, $\vec{x}_2$, and $\vec{x}_3$ are the triangle corners, and $u, v \in [0,1]$ with $u+v \leq 1$. The vector between a point on the moving triangle defined by the coordinates (u,v) and the other vertex, at time t, can then be written as:

$$\begin{aligned}\vec{F}(t,u,v) =& \vec{x}_0(t) - \big[(1-u-v)\vec{x}_1(t) + u\vec{x}_2(t) + v\vec{x}_3(t)\big] \\ =& (1-t)\vec{x}_0 + t\vec{x}_0^* \\ & - (1-u-v)\big((1-t)\vec{x}_1 + t\vec{x}_1^*\big) \\ & - u\big((1-t)\vec{x}_2 + t\vec{x}_2^*\big) \\ & - v\big((1-t)\vec{x}_3 + t\vec{x}_3^*\big).\end{aligned}$$

This is a tri-affine function which is zero precisely when the vertex lies on the triangle. The domain for the point-triangle test is $\Omega = [0,1] \times \{u, v \geq 0 \mid u+v \leq 1\}$, a triangular prism.

A similar tri-affine function can be defined for the segment-segment collision test, the vector between the point at fraction $u \in [0,1]$ along one segment and the point at fraction $v \in [0,1]$ along the other, at time $t \in [0,1]$. The domain is then $[0,1]^3$, the unit cube.

CCD then amounts to discovering if such a function has a root in the domain, a point in Ω which $\vec{F}$ maps to $\vec{0}$. We make an important simplification: we report a collision if there is any zero on the boundary of the domain (i.e. at the initial or final time, or at any edge or endpoint of the geometry) or if there is an *odd* number of roots in the interior. We justify ignoring the case of a nonzero even number of interior roots by noting that, in the reference frame of the triangle or one of the segments, an even number of roots indicates the other element crosses from one side to the other then back again — so a small perturbation of the trajectory (within the convex hull of the elements) would suffer no collisions at all. Important invariants, such as freedom from self-intersection and meshes outside of each other remaining outside, are still maintained using this slight relaxation of CCD.

4.5.1 Determining root parity

Write the image of Ω under $\vec{F}$ as $\vec{F}(\Omega) = \left\{ \vec{y} \mid \vec{y} = \vec{F}(\vec{x}), \vec{x} \in \Omega \right\}$, and similarly $\vec{F}(\partial\Omega) = \left\{ \vec{y} \mid \vec{y} = \vec{F}(\vec{x}), \vec{x} \in \partial\Omega \right\}$ for the image of the domain boundary $\partial\Omega$.

If $\vec{F}$ was smooth and one-to-one, determining if $\vec{0} \in \vec{F}(\Omega)$ could be done by counting the number of crossings of a ray from $\vec{0}$ to infinity through the boundary image $\vec{F}(\partial\Omega)$: an odd number of crossings indicates a root by the usual Jordan-Brouwer Theorem argument. However, our $\vec{F}$ may not be one-to-one: the image of Ω can "fold over itself". However, the parity of the ray crossings with $\vec{F}(\partial\Omega)$ intuitively still gives us the parity of the number of roots: the sum of an odd number of intersections for each separate root leads to an odd total if and only if there is an odd number of roots — with the proviso that if $\vec{0} \in \vec{F}(\partial\Omega)$, i.e. we have a root on the domain boundary, we always report a collision. More formally:

Root Parity Lemma. *Suppose $\Omega \subset \mathbb{R}^n$ is an n-polytope.*

Suppose $\vec{F} : \Omega \mapsto \mathbb{R}^n$ is C^2, has $p < \infty$ roots in Ω, has no roots on $\partial\Omega$, and has non-singular Jacobian at each root.

Suppose R is a ray from $\vec{0}$ to infinity. Call any point $\vec{x} \in \partial\Omega$ such that $\vec{F}(\vec{x}) \in R$ a crossing point*, then the* crossing number *q is the number of crossing points. Suppose that $\vec{F}(\partial\Omega)$ is smooth at the image of any crossing points, that the ray is not tangent to $\vec{F}(\partial\Omega)$ at any these points, and that $q < \infty$.*

Then, $p \equiv q \pmod 2$.

We offer a sketch of a proof of the lemma, and that it applies to the particular functions we need for CCD, in Appendix A.

This lemma also describes our algorithm. We report a collision if the image of the boundary $\vec{F}(\partial\Omega)$ passes through $\vec{0}$, and otherwise cast a ray from $\vec{0}$ in an arbitrary direction, and then count the number of crossings of the ray though $\vec{F}(\partial\Omega)$, choosing a different direction and trying again if any crossings are tangent or lie

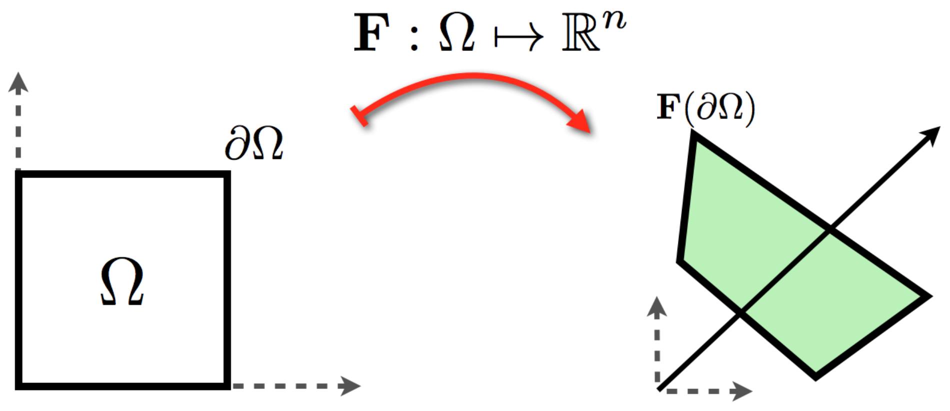

$\mathbf{F} : \Omega \mapsto \mathbb{R}^n$
$\partial\Omega$
Ω
$\mathbf{F}(\partial\Omega)$

Figure 4.5: Root parity test. In this case there are no roots in the domain, so the origin is outside of $\vec{F}(\Omega)$.

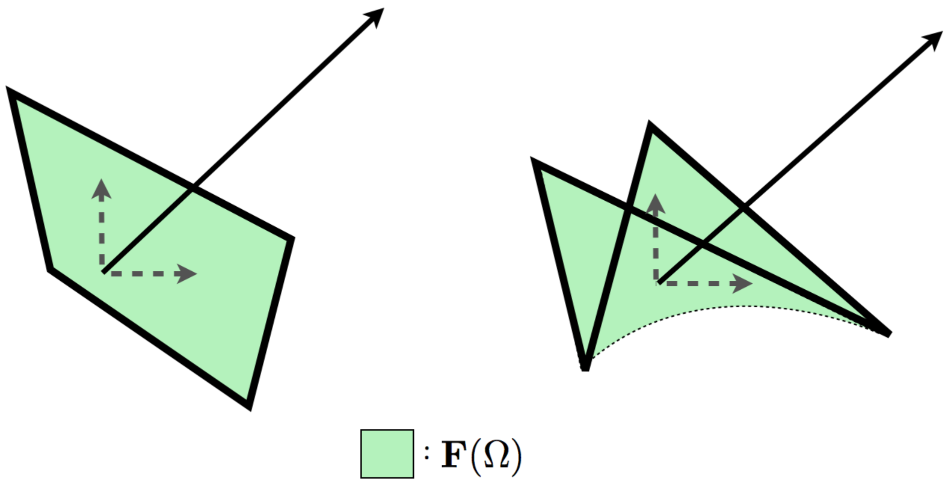


Figure 4.6: One and two roots in the domain. A ray cast from the origin will have odd and even parity, respectively.

on corners. Figures 4.5 and 4.6 illustrate this approach for the 2D case, showing cases where we have zero, one, and two roots in the domain.

We transform the boundary of these domains (cube or triangular prism) by the corresponding function $\vec{F}$ to get a *generalized* hexahedron or prism, and test for ray crossings on each of their faces. The hexahedron has potentially non-planar bilinear patches for faces (the restriction of the tri-affine function to a face of the domain is bi-affine), and the prism is composed of three bilinear patches and two triangles. Computing ray-triangle crossings can be done with exact arithmetic — however, we know of no prior practical method for quickly and exactly computing the crossings of a ray through a bilinear patch, or even the *parity* of the number of crossings, which is all we need. We thus introduce an efficient method for exactly computing this parity.

4.5.2 Ray-bilinear-patch crossing parity testing

We first define a continuous scalar function $\phi(\vec{x})$ which is positive if $\vec{x}$ is on one side of the patch and negative on the other side, and, to permit exact evaluation with floating-point expansions, define it using only multiplication, addition, and subtraction.

We define the following multivariate polynomial $\phi(\vec{x})$ where the indices 0 to 3 refer to the patch corner vertices:

$$\phi(\vec{x}) = h_{12}(\vec{x}) - h_{03}(\vec{x}).$$

The two h-functions are designed to be zero on the straight edges of the patch via products of plane g-functions for the various subsets of three vertices:

$$\begin{aligned} h_{12}(\vec{x}) &= g_{012}(\vec{x})\, g_{132}(\vec{x}) \\ h_{03}(\vec{x}) &= g_{013}(\vec{x})\, g_{032}(\vec{x}) \\ g_{pqr}(\vec{x}) &= (\vec{x}-\vec{x}_p)\cdot(\vec{x}_q-\vec{x}_p)\times(\vec{x}_r-\vec{x}_p). \end{aligned}$$

We claim that the zero level set of $\phi(\vec{x})$ contains the bilinear patch. To confirm this is indeed the function we seek, take an arbitrary point $\vec{x}$ with barycentric coordinates α, β, γ, and δ w.r.t. the corners of the patch:

$$\vec{x} = \alpha\vec{x}_0 + \beta\vec{x}_1 + \gamma\vec{x}_2 + \delta\vec{x}_3.$$

Recall that the barycentric coordinates of a point with respect to a tetrahedron are proportional to the signed volumes of the tetrahedra formed by the point and each of the triangular faces. Letting V be six times the signed volume of the tetrahedron spanning the corners of the patch, observe that:

$$\begin{aligned} g_{132}(\vec{x}) &= \alpha V & g_{032}(\vec{x}) &= \beta V \\ g_{013}(\vec{x}) &= \gamma V & g_{012}(\vec{x}) &= \delta V \end{aligned}$$

Therefore our function evaluates to:

$$\phi(\vec{x}) = \delta \alpha V^2 - \gamma \beta V^2.$$

Assuming $V \neq 0$, this is zero if and only if $\alpha\delta = \beta\gamma$, which occurs precisely for the parameterized bilinear surface:

$$\begin{aligned} \alpha &= (1-s)(1-t) & \beta &= (1-s)t \\ \gamma &= s(1-t) & \delta &= st. \end{aligned}$$

Moreover, it is clear that $\phi(\vec{x})$ changes sign across the zero level set, dividing space into positive and negative regions separated by the conic containing the bilinear patch.

This construction breaks down if $V = 0$. However this is the case when the patch is perfectly flat, where we can simply replace the entire ray-patch intersection test with two ray-triangle tests.

Next consider the tetrahedron spanned by the four corners of the bilinear patch. It is composed of two pairs of triangles, one pair corresponding to each side of the bilinear patch. For the "positive" triangle pair, any point $\vec{x}$ on either triangle has the property that $\phi(\vec{x}) \geq 0$, and vice versa for the "negative" pair of triangles. For the test, we consider two cases depending on whether the ray origin $\vec{0}$ lies inside the tetrahedron or not — which can be determined directly from standard sign-of-determinant "orientation" predicates with the tetrahedron's triangular faces.

If the ray origin $\vec{0}$ lies inside the tetrahedron, we can determine the sign of $\phi(\vec{0})$ and *replace* the ray-patch test with two ray-triangle tests, using the triangles corresponding to the opposite sign of $\phi(\vec{0})$. Ray-triangle intersection can also be broken down into determinant predicates [GD03]. If there is an intersection between the ray and either triangle, then the ray must also pass once through the bilinear patch.

If instead the ray origin lies outside of the tetrahedron, we can use either set of triangles as an equivalent proxy for the bilinear patch. The parity of the number of intersections between the ray and the triangle pair matches the parity of the number of (non-tangential) intersections between the ray and the bilinear patch.

Pseudocode for the test is given in Algorithm 4; Figure 4.7 illustrates the 2D analog. Since it relies only on determinants and the evaluation of ϕ, i.e. just multiplication and addition/subtraction, the test can be evaluated using floating-point expansions to give the geometrically exact result.

Algorithm 4 Ray-bilinear-patch crossing parity

```
Given: Ray origin $\vec{0}$, direction $\vec{R}$, and a bilinear patch.
Form the tetrahedron from the bilinear patch corner vertices.
Let $F_1^+$, $F_2^+$ be the tetrahedron faces where $\phi \geq 0$.
Let $F_1^-$, $F_2^-$ be the tetrahedron faces where $\phi \leq 0$.
if $\vec{0}$ is inside the tetrahedron then
   if $\phi(\vec{0}) > 0$ then
      return intersect( $\vec{0}$, $\vec{R}$, $F_1^-$ ) $\vee$ intersect( $\vec{0}$, $\vec{R}$, $F_2^-$ )
   else
      return intersect( $\vec{0}$, $\vec{R}$, $F_1^+$ ) $\vee$ intersect( $\vec{0}$, $\vec{R}$, $F_2^+$ )
   end if
else
   {Use either pair of triangles}
   return intersect( $\vec{0}$, $\vec{R}$, $F_1^+$ ) XOR intersect( $\vec{0}$, $\vec{R}$, $F_2^+$ )
end if
```

4.5.3 Putting it together

We now have the tools we need for determining the intersection parity of a ray versus a set of bilinear patches and triangles. For segment-segment CCD, our algorithm runs 6 ray-vs-patch tests for the faces of the hexahedron, and for point-triangle CCD, we run 3 ray-vs-patch and 2 ray-vs-triangle tests for the faces of the triangular prism, and determine the parity of the total number of intersections. If we have an odd parity, we know there is an odd number of roots in the domain, and so we must flag this as a collision. Algorithm 5 shows the point-vs-triangle test (the segment-vs-segment test is analogous).

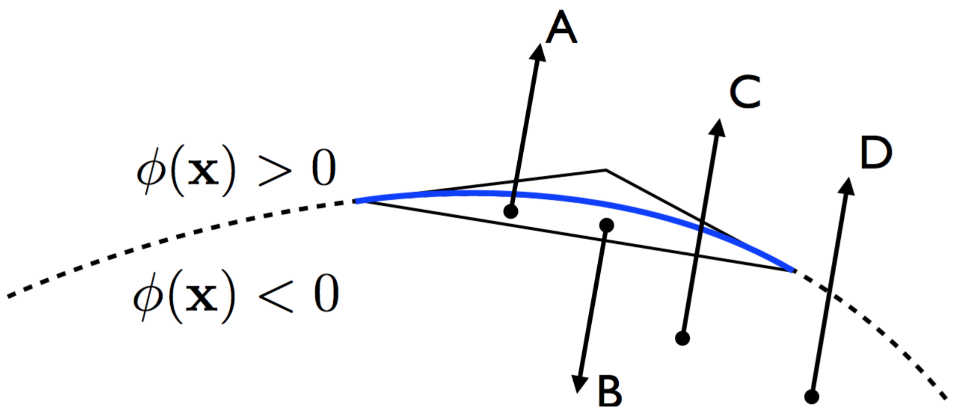


Figure 4.7: A 2D analog of the ray-vs-bilinear-patch parity test. Rays A and B have origins on the "negative" side of the patch, and so we test against the proxy geometry on the "positive" side. Ray A intersects both the patch and the proxy geometry, while B intersects neither. Rays C and D have origins outside the bounding simplex, and so can be tested with either proxy geometry.

There are a few special cases we must watch for. If the origin lies exactly on a patch or a triangle (i.e. there is a root on the boundary of the domain), then we report the collision and skip the ray testing. This includes, for example, the fully degenerate cases of exactly planar motion (such as two edges sliding into each other along a flat surface) that entirely defeats the cubic solver. In our simulations this type of collision is vanishingly rare unless artificially induced.

We must also take care if the ray hits an edge shared between two patches, between two triangles (acting as proxy geometry in the ray-vs-patch test), or between a patch and a triangle. If this occurs, we will see two positive ray intersection tests. In the context of inside-outside testing, this may or may not be correct (see Figure 4.8). Fortunately, since we are using exact arithmetic, we can precisely detect these degenerate cases (one barycentric coordinate will be *exactly* zero). In such a case, we simply choose a new ray direction at random and run the test again. Again, in our testing this happens only extraordinarily rarely.

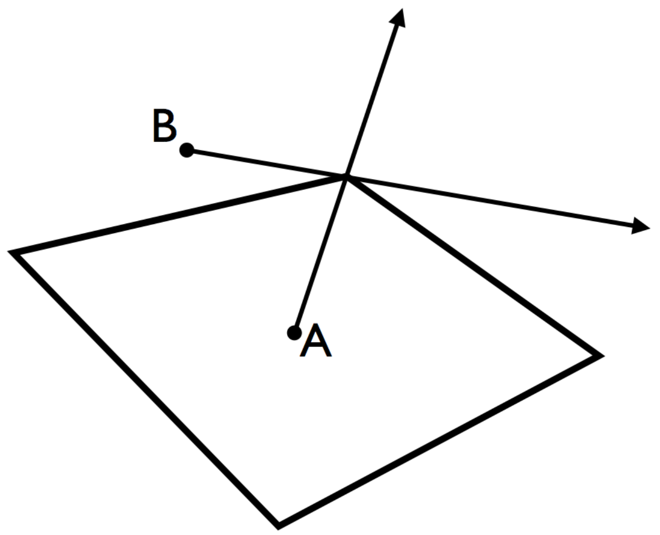


Figure 4.8: Two rays, each hitting two segments at their common endpoint. If we are testing each segment individually, then in both cases the parity of ray intersections is even. Here the parity of intersection count cannot determine whether points A and B are inside the quadrilateral. Perturbing the rays slightly would produce the correct results in both cases.

4.5.4 Implementation

Fast implementation of exact intersection testing is crucial for making our approach practical. Computing intersections with expansion arithmetic is expensive, so we use a filter: we evaluate the determinants and ϕ first with interval arithmetic, only switching to exact expansions when the sign is indeterminate (the final interval contains zero). See Brönnimann et al. for the case of determinants [BBP01].

To avoid repeatedly switching the rounding mode during interval arithmetic, we use the standard "opposite trick", storing the negative of the lower bound and defining new operations that rely on one rounding direction only. We have also experimented with using SIMD vectors to store the intervals and using vector intrinsics for arithmetic operations [Lam06, Gou10], but found that our implementation of this strategy was not significantly faster in practice than simply operating on two doubles.

Algorithm 5 Point-triangle collision test

```
Given: corner vertices of the domain, $X$
Create ray $(\vec{0}, \vec{R})$ with arbitrary direction $\vec{R}$
$S \leftarrow 0$
**for** $i = 1 \rightarrow 3$ **do**
    Form bilinear patch $i$ with appropriate $\vec{F}(X)$
    $p_i \leftarrow$ intersection parity of ray $(\vec{0}, \vec{R})$ vs bilinear patch $i$
    $S \leftarrow S + p_i$
**end for**
**for** $j = 1 \rightarrow 2$ **do**
    Form triangle $j$ with appropriate $\vec{F}(X)$
    **if** Ray $(\vec{0}, \vec{R})$ intersects triangle $j$ **then**
        $S \leftarrow S + 1$
    **end if**
**end for**
**return** $S \equiv 1 \pmod 2$
```

Collision test culling is also critical for efficiency. We compute the axis-aligned bounding box (AABB) of each moving edge, triangle and vertex and only run collision detection when AABBs overlap, accelerating this test with a regular background grid. We further cull tests by checking if, for any of several non-axis normal directions, there is a plane separating the origin from the transformed hexahedron or prism. Our implementation of this plane test uses interval arithmetic for robustness: only when all vertices are definitely on the negative or positive side of a plane (no interval contains zero), do we consider the plane to be a separating plane. This relatively inexpensive plane-based testing eliminates 99% of the tests, considerably improving performance.

4.5.5 Resolving collisions

We implemented our new algorithm using interval filtered floating-point expansion arithmetic, providing practical, provably robust CCD code for deforming meshes without any user-tuned tolerances. However, at this point we can only guarantee collision detection: this says nothing about *resolving* these collisions, i.e. finding a physically consistent adjustment to the final configuration of the mesh that eliminates all collisions. Indeed, taking into account that the final positions are quantized to a finite number of bits of precision, provably robust but ideally physical collision resolution may involve the solution of a rather daunting large-scale integer programming problem. As an example of the complications involved, even just transformation of an intersection-free mesh with a rotation matrix can potentially create self-intersection once the results are rounded again.

To resolve collisions, we use the velocity filtering approach discussed in Chapter 3. While previous works used the normal at the time of collision (typically from the triangle or from the cross-product of edges), we have found this is not a crucial choice. Interpolating the geometry at $t = 0.5$ and computing the normal from the vector between the closest points on the two mesh elements has worked equally well in our experiments, and is more computationally efficient.

4.5.6 Examples

We tested our new CCD routines in a standard mass-spring cloth simulator with an initially curved sheet of cloth of resolution of 40×400 vertices, dropped on a solid ground plane. As shown in Figure 4.9, this results in a large number of collisions as the cloth stacks up on itself.

Armed with our parameterless collision detection system, we also demonstrate a complete FEM simulator with on-the-fly, adaptive remeshing. We use linear elasticity with rotated finite elements, as described by Etzmuß et al. [EKS03], and simple edge crossover springs for bending forces. The equations of motion for the vertices are:

$$\frac{d\vec{x}}{dt} = \vec{v} \tag{4.3}$$

$$\frac{d\vec{v}}{dt} = M^{-1}\vec{F}, \tag{4.4}$$

where M is the diagonal mass matrix with mass concentrated on the vertices, and $\vec{F}$ is the force due to in-plane linear elasticity and springs across edges.

We choose simple edge splitting, flipping, and collapsing as our mesh optimization operations, and make them all collision safe, using CCD on "pseudo-trajectories", as described in Chapter 3. To increase the vertex density in high-curvature areas, we scale the measured edge lengths by local curvature estimates when deciding to collapse or split edges. (We note that these operations are perhaps not as suitable for cloth simulation as a regular subdivision scheme, but it is a reasonable proxy for examining the challenges faced when maintaining intersection-free adaptive surfaces.) Figure 4.10 shows a frame from a simulation with a single piece of cloth, and the underlying rest-state mesh. We also show a more challenging CCD scenario, with several layers of cloth draped over a solid sphere (Figure 4.11).

All of our tests were performed on a single core of a 2.7 GHz Intel i5 processor with 4GB of RAM. We integrated our new CCD algorithm into the open-source El Topo surface tracking library developed in Chapter 3, as it provides an intersection-free remeshing framework suitable for adaptive cloth simulation, and provides an implementation of the cubic-solver based CCD approach for comparison. (Our cloth simulation is only client code of this library – it was not published as part of El Topo.)

The average time spent per call to CCD, not including culling based on AABB comparisons, but including plane-based culling, was approximately 614 ns for segment-segment testing, and 439 ns for point-triangle testing. By comparison, the average time in El Topo's cubic solver CCD implementation (again not including AABB culling) was 649 ns and 659 ns.

Counting only tests which were positive (exercising the entire path of our code), the average time per call was 19 µs for segment-segment, and 15 µs for point-triangle, compared to 1.2 µs for both tests with the cubic solver CCD. This indicates that without culling, our new algorithm is more expensive than the cubic-solver version (as expected) but also that our culling is effective enough to cancel out any efficiency penalty.

4.6 Conclusions

Section 4.3 introduced a novel CCD method, constructed from a set of predicates which can be evaluated exactly. By considering time as an additional spatial variable, we rephrased the continuous collision detection problem as a set of higher-dimension simplex intersection problems. These intersection tests can be evaluated exactly, either with determinant tests, or with our new exact linear solver, introduced in Section 4.4.2. The new linear solver is based on fraction-free Gaussian Elimination algorithms for integer arithmetic, which we extended to work with floating-point expansions for geometrically exact predicate testing.

In Section 4.5, we presented a second approach to continuous collision detection. In this section, we showed that the collision detection problem can be rephrased as determining whether a function has an odd number of roots in a given domain. We then showed how this problem can be reduced to testing a ray from the origin against the image of the domain boundary and counting the parity of the crossings. This in turn reduces to a set of ray-vs-triangle and ray-vs-bilinear-patch tests, built from determinant and simple function sign evaluations.

Our implementation uses a floating-point filter approach for efficiency — first checking if the correct Boolean result can be determined using interval arithmetic, and only using floating-point expansions for exact evaluation if required. We demonstrated the utility of our approach with a challenging test case: simulation of cloth undergoing on-the-fly remeshing with a large amount of contact. To our knowledge, this is the first time an adaptive cloth simulation scheme has been presented which explicitly deals with the challenges of continuous collision detection.

There are several avenues of future work. While irrelevant for the simulations we presented, being able to distinguish zero from a positive even number of roots could be critical for other applications. Our approach should extend naturally from multi-affine functions to testing intersections with higher-degree polynomial patches. Rigid body motion is more naturally expressed in terms of screw motions; though the intermediate positions involve trigonometric functions of time, reparameterization similar to the NURBS approach to conic sections would lead to a multivariate polynomial problem amenable to this attack.

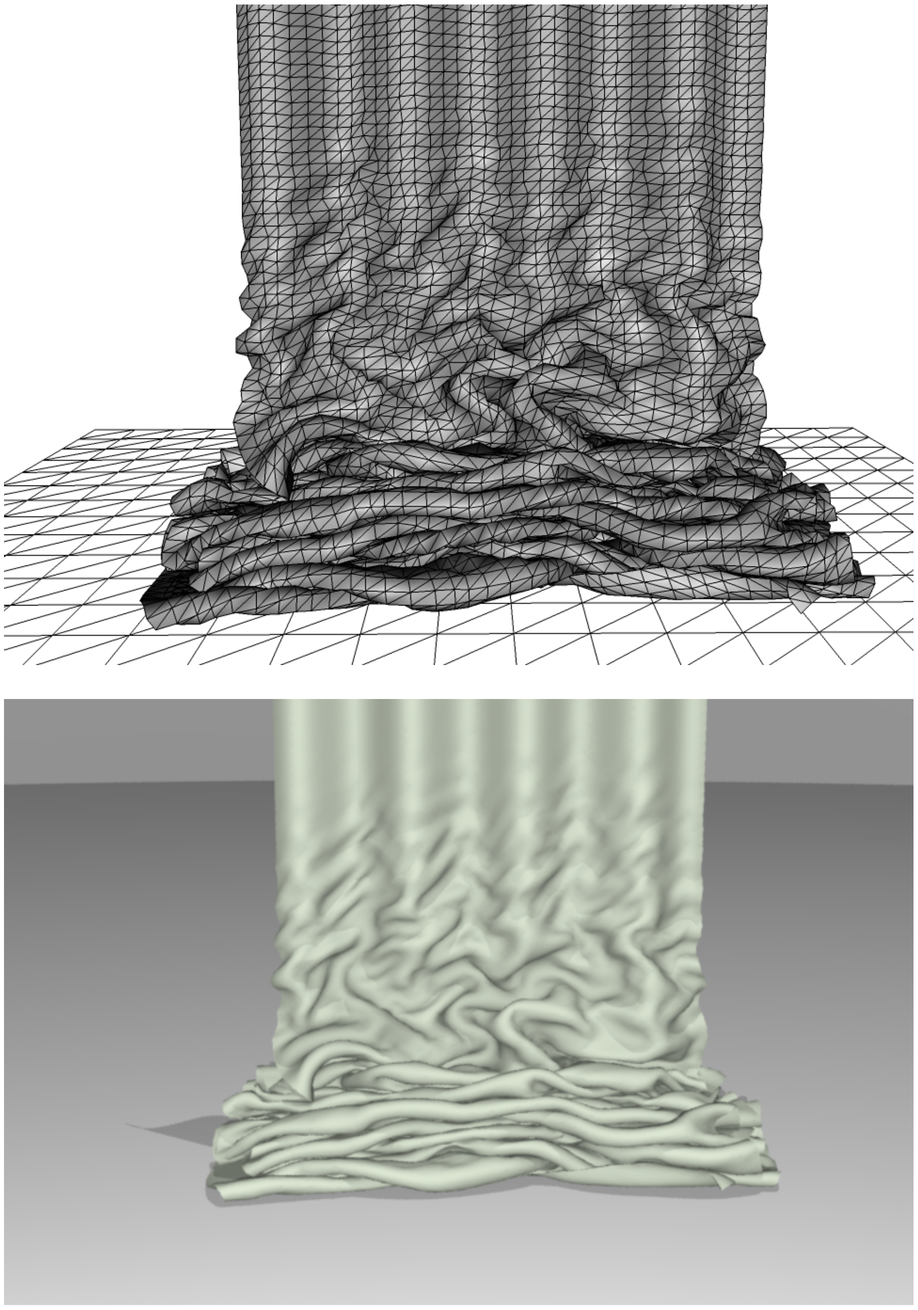
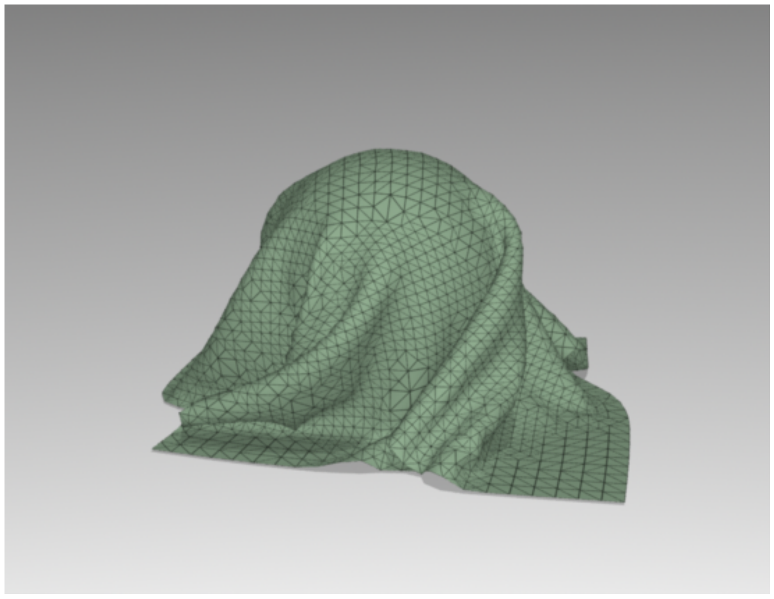


Figure 4.9: CCD stress test with wireframe and smooth rendering.



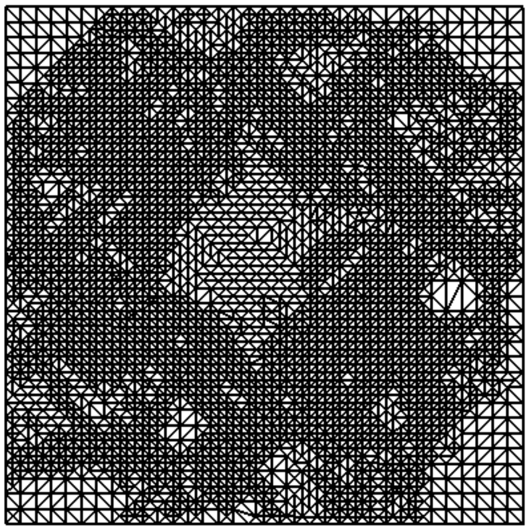


Figure 4.10: Cloth with an adaptive simulation mesh

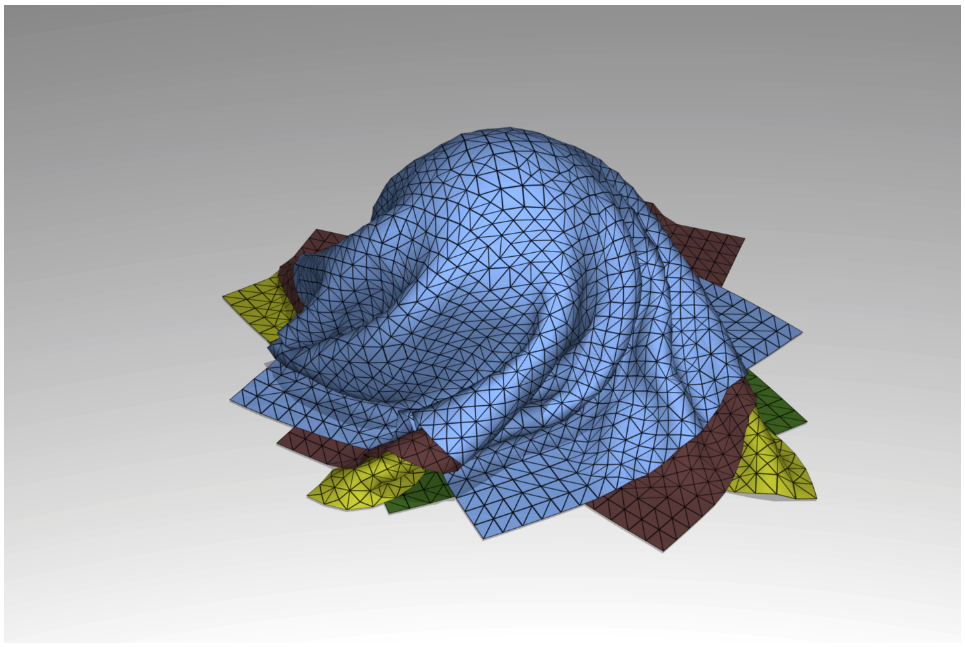


Figure 4.11: Four layers of adaptive cloth

Chapter 5

Animating smoke as a surface

5.1 Introduction

In visual effects, fluids such as smoke are often simulated with a grid-based fluid solver or particle system. For smoke, the visible density of soot is usually tracked using either a set of passive marker particles, or a scalar density function on an Eulerian grid. If a marker particle system is used, tens of millions of particles may be required to prevent a grainy, bumpy, or blobby look, which can occupy considerable storage and network resources. Furthermore, if the smoke effect is particularly dense, only the set of particles or grid cells near the surface may be of interest, while the interior is almost completely occluded. Grid-based density fields are susceptible to diffusion, and may also require very high resolutions to capture thin, wispy sheets of smoke. Similarly, to simulate liquid dye in a tank of water, one might consider advecting a density field or set of marker particles to track the dye location, incurring exactly the same downsides.

We propose to instead define a surface mesh which encloses a volume of dye or soot. For this chapter, we consider the surface to be embedded in a fluid, with each vertex moving passively, similar to a set of marker particles. (In the next chapter we will present a method in which the surface mesh also defines the dynamics.) Implicit in this model is that the concentration of soot or dye is uniform within the volume, and as such, we do not require density samples in the interior. The concentration s evolves in a divergence-free velocity field, according to:

$$\frac{\partial s}{\partial t} + \vec{u} \cdot \nabla s = \varepsilon \nabla^2 s, \tag{5.1}$$

where ε is the rate of soot or dye diffusion. Just as the viscosity in air is virtually negligible for most practical scenarios, hence the inviscid equations are an excellent model, the rate of molecular diffusion of temperature and soot particles in air is vanishingly small, so they are just advected through the velocity field, and the equation becomes:

$$\frac{\partial s}{\partial t} + \vec{u} \cdot \nabla s = 0. \tag{5.2}$$

Close-up renders of smoke using particles often show artifacts such as graininess or blobbiness. By comparison, our triangle mesh surface shows fewer artifacts, appearing as a flat surface under extreme close-up. Our method is also resistant to numerical dissipation present in grid-based density tracking methods.

5.2 Related work

Foster and Metaxas animated smoke using a one-phase, grid-based Navier-Stokes solver [FM97]. They used marker particles to track smoke through the simulation, transferring particle density onto a grid at render time. Stam introduced an unconditionally stable time integration scheme to computer graphics [Sta99], and his method was later adapted to use a staggered grid with vorticity confinement [FSJ01]. His method uses a grid-based scalar density field to track the concentration of smoke. Recent work has focused on reducing the damping effect of numerical diffusion by introducing improved integration schemes [ZB05, KLLR07, SFK$^+$08], adding sub-grid-scale turbulence [KTJG08, SB08], or through the use of procedural methods [SRF05, BHN07].

The use of triangulated surface meshes in fluid animation has traditionally been eschewed in favour of grid- or particle-based surface tracking methods, such as the particle level set method [EFFM02]. Recently, however, some researchers in the computer graphics community have begun turning to triangle mesh-based surface tracking for liquid simulation [WTGT09, Mül09, BBB10, WTGT10]. These authors generally use explicit surfaces to track the fluid interface in a free-surface simulation, whereas in this thesis, we use surfaces to visualize (and later simulate) volumes of smoke in a one-phase fluid simulation.

Although grid-based methods are very popular and the subject of much recent research effort, procedural methods for animating smoke and other fluids have been widely used in practice for many years, starting with "flow primitives" for particle systems, introduced by Sims [Sim90] and Wejchert and Haumann [WH91]. In this chapter, we simulate smoke using primitives based on vortex filaments [AN05], flow noise [PN01] and divergence-free curl noise [BHN07].

In somewhat related work, mesh surfaces have been used for visualizing 3D vector fields. Hultquist generalized streamlines used in analysis of airflows to *stream surfaces* [Hul92]. More recently, *streak surfaces* (moving open surfaces seeded continuously from a curve), and *time surfaces* (seeded instantaneously from a surface) have been treated using methods similar to the one presented in this chapter. Krishnan et al. [KGJ09] recently introduced an algorithm which advects an adaptive mesh through a 3D vector field to treat both of these categories of surface. A similar visualization method which explicitly uses a smoke metaphor was introduced by von Funck et al. [vFWTS08]. Their method does not use adaptivity, but instead gradually reduces the opacity of poor quality triangles, giving the appearance of dissipating smoke or steam. Park et al. [PSCN10] constructed NURBS surfaces from particle streamlines to achieve thin, sheet-like surface renderings of smoke simulations. Most of these methods use open surfaces to visualize wispy, thin sheets, whereas our approach can use closed meshes with arbitrary thickness, so that thick smoke and fog can also be represented efficiently (see Figure 5.4).

5.3 Surface tracking

In order to use a triangulated surface to define a volume of fluid, we require a surface tracking system with a certain set of features. Obviously an adaptive surface is required to generate highly detailed results as the surface undergoes extreme deformations. We would also like to maintain good mesh quality in terms of aspect ratios and surface smoothness. The surface tracking library described Chapter 3 is a clear candidate.

In our examples we disabled collision detection and resolution, allowing the meshes to self-intersect, as the simulation was unaffected and collision detection is the bottleneck of the implementation. With a divergence-free velocity field, self intersections are due only to numerical error and are rare enough in practice to not be noticeable, as long as the renderer can handle inside-out volumes in some plausible way. We also disabled topology changes to simplify the algorithm; they could, however, be introduced to reduce the number of triangles.

5.4 Generating fluid motion

Our surface tracking method can be used with any underlying vector field. We use two separate methods for generating fluid motion: a grid-based fluid simulator using FLIP advection, and a procedural animation system.

5.4.1 Eulerian fluid simulation

Our first method for generating fluid motion uses a staggered-grid fluid simulator with FLIP advection as introduced to the graphics community by Zhu and Bridson [ZB05]. We numerically solve the incompressible, inviscid Euler equations:

$$\frac{\partial \vec{u}}{\partial t} + (\vec{u} \cdot \nabla)\vec{u} = -\frac{\nabla p}{\rho} + \frac{\vec{F}}{\rho} \tag{5.3}$$

$$\nabla \cdot \vec{u} = 0 \tag{5.4}$$

for velocity $\vec{u}$, and pressure p, where ρ is the density. Fluid motion is introduced either by adding a buoyancy force $\vec{F}$ to grid nodes inside the smoke volume, or by explicitly setting velocity values before pressure projection. The embedded surface is treated as strictly passive, and is advected along with the grid velocities. We outline the major steps of the algorithm, emphasizing in bold the steps involving the triangle mesh surface.

1. **Perform mesh improvement operations.**
2. **Handle mesh topology changes (optional).**
3. Apply buoyancy or scripted forces to the grid.
4. Perform divergence-free velocity projection with appropriate boundary conditions.
5. **Interpolate velocity from the grid faces to the surface vertices.**

6. **Integrate mesh vertices (with optional collision detection and resolution).**
7. Advect grid velocity using FLIP.

We use this framework to simulate both smoke and dye, changing only the forces introduced in step 3, and the final rendering set up.

5.4.2 Procedural fluids

Since the main contribution of this chapter is independent of the underlying dynamics, we also demonstrate our idea using a small toolkit of procedural methods. We use the linear superposition of several fluid flow primitives to generate plausible smoke without an Eulerian fluid simulator. We use *vortex rings* to generate upward motion, *divergence sources* to simulate smoke dispersion, and *curl noise* [BHN07] to add extra small-scale detail.

Vortex ring

We use a set of vortex rings to generate general upward motion. The potential function of a vortex ring perpendicular to the y axis with center $\vec{c}$, radius r, at a point $\vec{x}$ is given by

$$\vec{\psi}(\vec{x}) = \frac{1}{(r-d)^2 + 2rd} < x_3, 0, -x_1 >,$$

where $d = \|x - c\|$. The velocity at $\vec{x}$ due to the ring is then the curl of this potential:

$$\vec{v_r}(\vec{x}) = \nabla \times \vec{\psi}(\vec{x})$$

Source

We add a divergent source with center $\vec{c}$ and radius r. If d is the distance between $\vec{x}$ and $\vec{c}$, then the velocity imparted by this source is given by:

$$\vec{v_s}(\vec{x}) = \max(0, r-d) n(\vec{x}) \frac{\vec{x} - \vec{c}}{d}$$

where $n(\vec{x})$ is a spatially continuous scalar noise function such as Flow Noise [PN01].

Curl noise

Curl noise uses a spatially and temporally continuous noise function to generate a vector-valued potential function. We take the curl of this potential to generate a velocity field:

$$\vec{v}_n(\vec{x}) = \nabla \times \vec{n}(\vec{x})$$

Total flow

We allow for arbitrarily many instances of each type of primitive, and assign a user-defined weight to each instance. The velocity at any point in space is then the weighted sum of all instances of all primitives:

$$\vec{v}(\vec{x}) = \sum_i \alpha_i \vec{v}_{ri}(\vec{x}) + \sum_j \alpha_j \vec{v}_{sj}(\vec{x}) + \sum_k \alpha_k \vec{v}_{nk}(\vec{x})$$

5.5 Results

We compare our surface tracking approach against the standard marker-particle-based approach for tracking smoke. As mentioned above, the sheer number of particles required for high-resolution rendering can occupy considerable storage and network resources. Using our method, we can generate improved results for the same data storage requirements. All simulations in this section were executed on a single core of a PowerMac G5 2.0 GHz PPC with 2 GB of RAM.

5.5.1 Eulerian fluid simulation

We first compare results generated using a grid-based fluid simulation, as described in section Section 5.4.1. Figure 5.1 shows a frame from an animation generated using marker particles, and a frame generated using our surface tracking method. For rendering, we use a simple ray tracer, assuming an exponential decay of light transmittance inside the smoke volume, with no self-shadowing.

Assuming a triangle and a vertex require the same storage (a triple of 32-bit

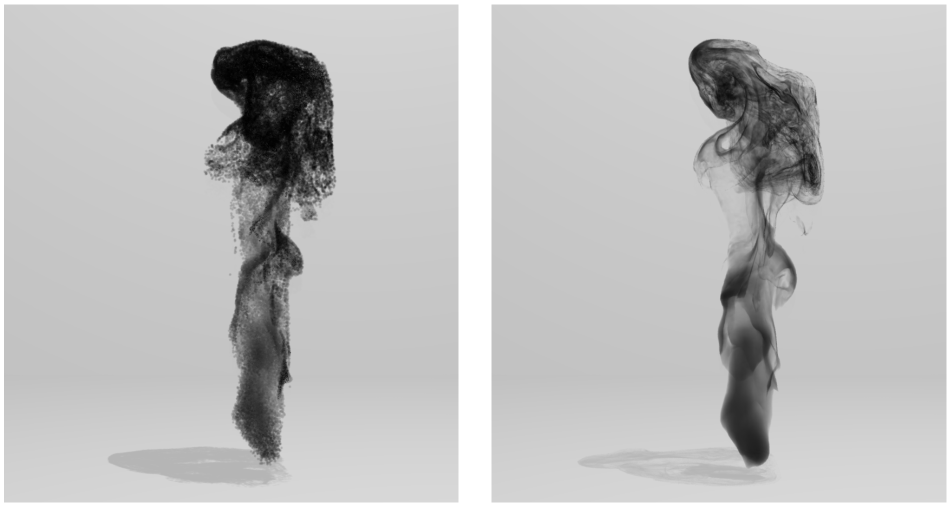


Figure 5.1: Frames from a smoke animation. The volume of smoke is tracked with marker particles (left) and an explicit surface (right)

integers for triangles and a triple of single-precision floats for vertices), the total animation of 500 frames was stored using just under 54 million of these 3D vectors. To match this data storage requirement, we generated an animation using a total of 54 million marker particles (summed over all frames). It took 78 minutes to run the simulation using surface mesh tracking, compared to 25 minutes using marker particles (not including render times). For a small factor longer running time, we achieve significantly higher quality results with the surface approach. We again stress that storage is often the critical limiting factor for film simulation and rendering pipelines, thus we compare the two methods at roughly equivalent amounts of storage.

Since our method uses a surface, we can achieve a smooth look even at very high resolutions. Figure 5.2 shows an image cropped from a large, high-resolution render of the surface, compared against the same frame rendered using marker particles. To achieve comparable quality results that hold up at film resolutions with marker particles, we would need vastly more data.

Our surface tracking method can also be used for modeling marker dye suspended in a fluid. In this example, we introduce a spherical dye source with an

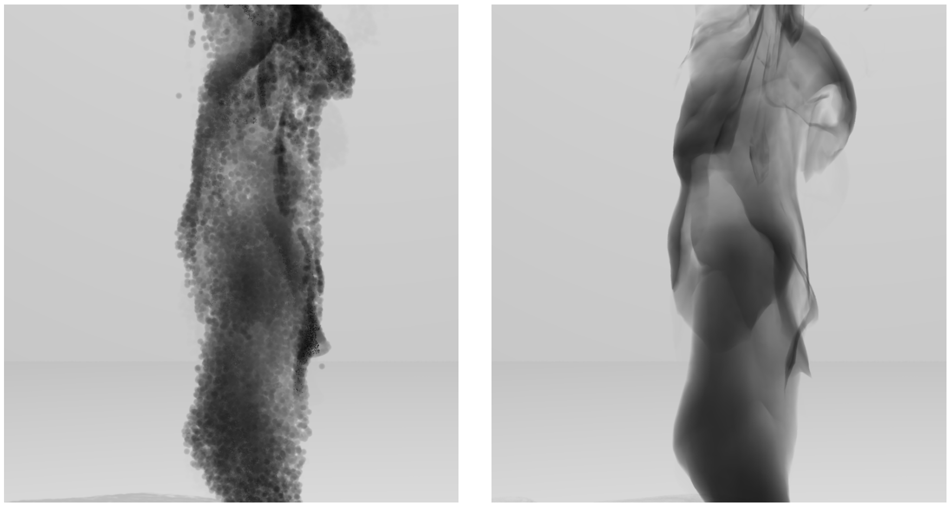


Figure 5.2: A close-up image reveals the difference in quality between surface meshes and marker particles.

associated downward force. No other forces are present. The simulation and surface tracking ran in 35 minutes, and produced a total of 32 million triangles and 16 million vertices over 500 frames (48 million vectors). We ran the same simulation using marker particles, which ran in 14 minutes and produced 47 million marker particles over 500 frames, roughly equivalent to the amount of data produced using our surface tracking method. Figure 5.3 shows a side-by-side comparison of the two methods. Again, for the same amount of storage and only a small factor longer running time, the surface mesh approach produces much higher quality results.

5.5.2 Procedural fluids

The procedural animation system described in Section 5.4.2 allows us to efficiently generate a thick plume of smoke. A plume with 25k triangles is shown in Figure 5.4. We used a single-scatter model with self-shadowing in our ray tracer to render this plume. Since the plume is nearly opaque, we can optimize the ray tracer by taking only a few samples near the surface, and consider the volume deep inside the plume as completely occluded. (In the next chapter, we demonstrate a surface shader approach to rendering that achieves much of the same visual appearance for

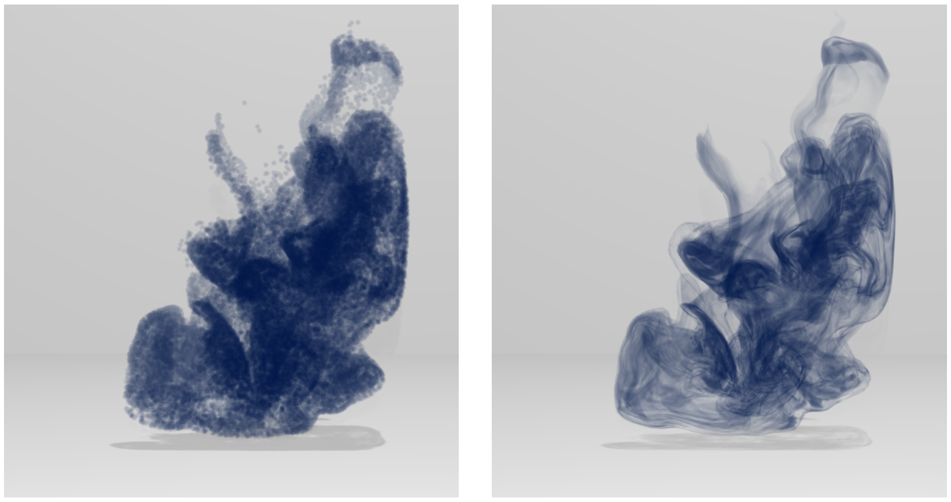


Figure 5.3: Dye in a volume of fluid modeled as a collection of particles (left), and as a surface (right)

a fraction of the computational effort.)

We also use curl noise to generate motion on an open rectangular surface, which represents the top surface of a volume of smoke. At render time, this surface is extruded downward to generate a thick layer of foggy smoke (Figure 5.4). Since we are only defining motion on the top surface, which contains just 7200 triangles, the animation can be generated very efficiently.

5.6 Conclusion

We presented explicit surface meshes as an alternative to marker particles or grids for simulating effects such as smoke and dye. We demonstrated that, per unit of storage, this method offers an improvement in visual quality over using marker particles. The continuous nature of our surface mesh eliminates graininess and blur from rendering, while still resolving very thin sheets and strands. We tested our method with a grid-based fluid simulator and a procedural animation system, rendering with a ray-marching approach to volume rendering.

Since we do not model diffusion, the animation produced using our method will have a characteristic cohesion. In particular, dye dropped in water will not

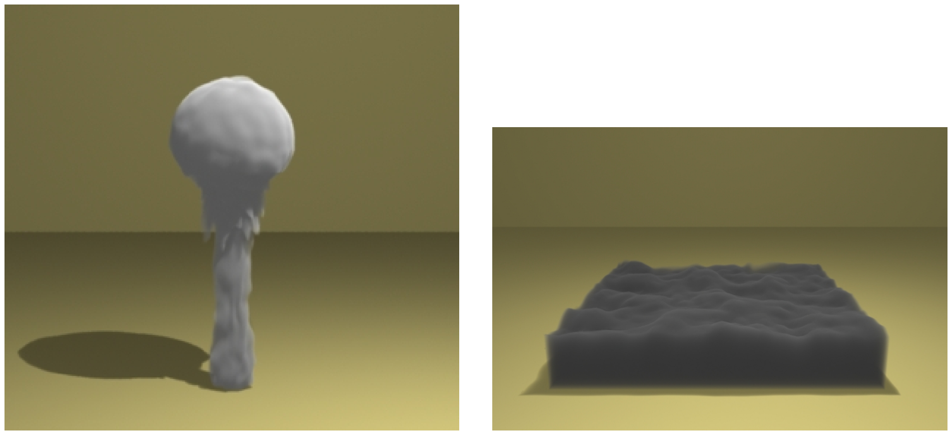


Figure 5.4: A thick plume of smoke generated using an explicit surface embedded in a procedural flow field (left). A thick layer of fog is modeled using a single, open surface mesh (right).

appear to dissolve as it would if modeled using a density grid, nor will smoke dissipate into the surrounding air. One could argue that in the absence of an actual model for molecular diffusion or sub-grid-scale turbulence, this is a sign that our method actually tracks the marker substance *more* accurately. However, in an artistic context diffusion may be desirable, physical model or no. In Chapter 6, we will introduce age-based blurring in the renderer to partially account for diffusion at render time in a parallel, per-frame operation. We have also experimented with activating topology changes in the surface tracking code, as well as varying the aggressiveness of the mesh simplification operations, which tends to delete thin volumes of dye or smoke. Investigating the effect of these parameters and translating them into meaningful, user-tunable values is another potential area of future investigation.

Chapter 6

Linear-time smoke animation with vortex sheet meshes

6.1 Introduction

Tantum videt, quantum computato: as much as one sees, that much one should compute. While the last decade and a half has seen a revolution in the quality of cinematic smoke animation through the use of physical simulation, scalability remains a serious impediment to high quality real-time smoke effects and interactive artistic design. To achieve $\sim n \times n$ visual detail in 2D renders, all prior methods have needed at least $O(n^3)$ time per step, due to the simulation or the rendering or both. In this chapter we develop a purely Lagrangian vortex sheet model of smoke simulation, and argue that in the most useful asymptotic limit it captures all essential dynamic and visual detail in a closed 2D surface. We discretize the model with an adaptive dynamic triangle mesh, implement time integration using an optimal linear-time Fast Multipole Method (FMM), and render the result at interactive rates — computing only what is seen. Additionally, our method handles arbitrary no-stick moving solid boundaries and provides output for more sophisticated offline rendering.

Previous smoke simulation methods in graphics can loosely be divided between velocity-pressure formulations and vorticity approaches. The classic example of a velocity-pressure solver is Stam's Stable Fluids approach [Sta99]: this requires an n^3 grid of the volume, and even with an optimal linear-time multigrid solver for the pressure projection, is an order of magnitude more costly than the desired $O(n^2)$ rendered output. Vortex methods use a much richer representation whereby in principle many fewer elements (such as our vortex sheet of $O(n^2)$ triangles) can replicate the same detailed dynamics. However, so far in graphics the solvers hit a simulation bottleneck when calculating velocity from the vorticity and solid boundary conditions, resulting in a dense $n^2 \times n^2$ matrix solve which can be as bad as $O(n^4)$ storage and $O(n^6)$ run-time using *LU* factorization. Prior vortex methods also have used simpler volumetric soot particle tracking for providing output to the renderer, which introduces another expensive $O(n^3)$ bottleneck. For film visual effects, this strains the file system and network capacity in particular, as simulation is generally run separately from rendering, and so all particle data must be stored and transferred to the renderer.

Our basic model in this chapter treats air as an incompressible inviscid fluid with the Boussinesq buoyancy approximation,

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = \alpha \vec{g} \tag{6.1}$$

$$\nabla \cdot \vec{u} = 0 \tag{6.2}$$

where $\alpha \vec{g}$ is the buoyancy force, with $\alpha = 0$ in the ambient clear air, and α in the smoky region depending on the ratio of smoke temperature to ambient temperature as well as the density of soot particles. At solid boundaries we take the no-stick boundary condition,

$$\vec{u} \cdot \hat{n} = \vec{u}_{\text{solid}} \cdot \hat{n}, \tag{6.3}$$

and at infinity the far-field condition $\vec{u} = 0$. We assume $\vec{u} = 0$ initially. Without viscosity, the vorticity $\vec{\omega} = \nabla \times \vec{u}$ remains zero apart from where buoyancy creates it; vorticity is advected and stretched by the flow, but doesn't diffuse away from smoke/temperature gradients.

As in the previous chapter, we ignore molecular diffusion. Under the assumption that smoke sources have uniform temperature and soot concentration, and also

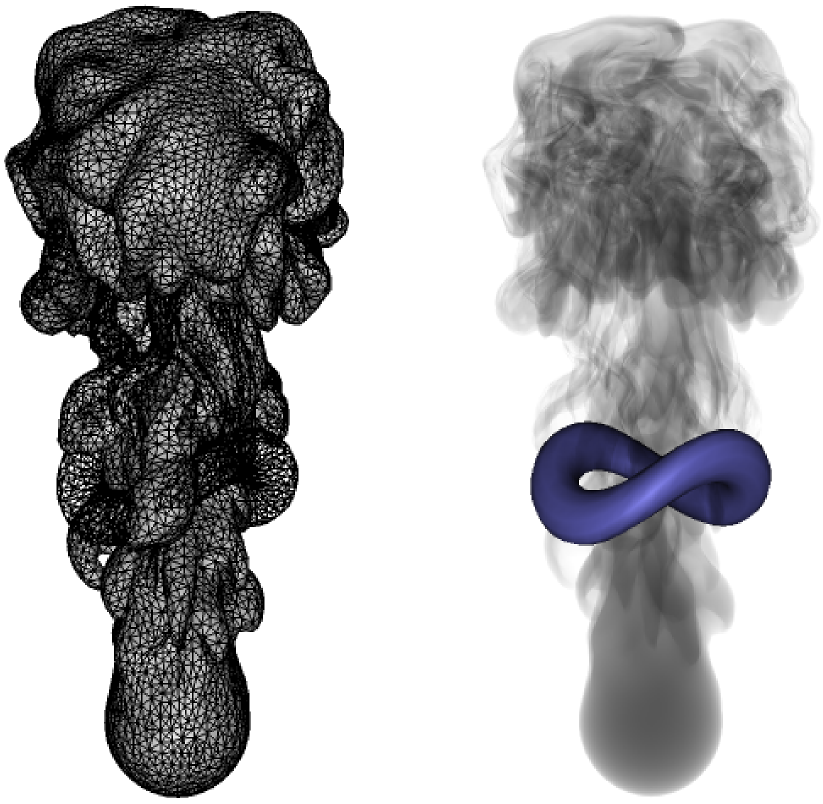


Figure 6.1: We represent smoke with an adaptive triangle mesh both for linear-time simulation, as a vortex sheet, and linear-time interactive rendering as the boundary of the smoky region. Here a deforming torus influences the buoyant smoke flowing around and through it.

using the incompressibility condition, the air is then precisely divided into uniformly clear and uniformly smoky regions. This in turn implies buoyant vorticity is only generated at the surface separating the two, where α jumps in the normal direction, hence all vorticity is concentrated in that surface, making a *vortex sheet*. As Stock et al. show [SDT08], the vorticity is further constrained to remain tangent to this sheet — in section 6.3, we show that our discretization identically enforces this constraint.

6.2 Related work

Much of the previous work related to smoke animation was covered in the previous chapter (see Section 5.2). The important summary is that many contemporary smoke simulation methods use a volumetric grid or tetrahedral mesh, which have n^3 complexity and storage requirements. Losasso et al. improved the scalability somewhat with the replacement of the uniform grid by an octree structure, but adaptivity in velocity-pressure formulation is difficult as much of the volume must still be finely gridded for adequate behaviour [LGF04]. Other fluid solvers (e.g. [Exo11]) only run a solve on a sparse tiled grid instantiated in a bounded envelope around the smoke, but need a substantial envelope to approximate an unbounded domain, and run at full volumetric resolution inside the smoke, likewise limiting scalability.

Vortex methods for fluid dynamics were developed during the 1980s as an efficient and high-order, purely Lagrangian method of numerically solving the Euler equations for fluid flow [BM82, AG85]. Subsequent work was done on developing viscous vortex methods for the Navier-Stokes equations and semi-Lagrangian contexts [CK00]. Though the grid distortion in fully Lagrangian vortex methods makes them less useful for scientific application, their efficiency in preserving rotational flow makes them appealing for fluid animation.

In computer graphics, early 2D simulations discretized vorticity on a grid or used vortex-in-cell methods, advecting Lagrangian parcels of vorticity through the domain but relying on a grid-based Eulerian solver [YUM86, CMTM94, GLG95]. Park and Kim [PK05] introduced a fully Langrangian vortex particle method, using the Biot-Savart formula to compute velocity on the particles without a grid. Recently, Weißmann and Pinkall [WP10] have demonstrated a complete vortex filament method, with adaptive filaments. Both approaches discretize solid boundaries with vortex sheets and rely on static geometry to enable precomputation for efficiently handling boundary conditions. In comparison, our linear-time approach can handle arbitrary moving solid boundaries, as demonstrated in section 6.4.

Several papers have used vortex primitives to augment Eulerian fluid simulations, including vortex particles [SRF05] and Eulerian vortex sheets [KSK09]. Concurrent with this work, Pfaff et al. have introduced a Lagrangian vortex sheet method for augmenting an Eulerian fluid solver [PTG12]. Their vortex sheet discretization is similar to the one presented in this chapter; however their vortex sheet interactions are purely local, with global flow handled by an underlying grid-based, Eulerian fluid solver. Their use of compact kernels avoids the need for FMM, but the reliance on the grid-based solver limits scalability and complicates the simulation process.

Boundary integral equations were initially developed a century ago as a tool for proving the uniqueness and existence for solutions to PDEs, particularly Laplace's equation. They have had a modern renaissance as effective numerical methods for solving a variety of PDEs [GGM93]. This has been primarily motivated by the construction of effective summation methods such as the FMM which allows rapid solution of the dense linear equations resulting from the numerical treatment of many integral equations.

6.3 The vortex sheet mesh

Our discretization of the vortex sheet closely follows Stock et al. [SDT08] and Pfaff et al. [PTG12] (though unlike their methods which rely on a 3D background grid for bulk velocity computation, we directly compute it in linear time on the surface itself with FMM, including arbitrary solid boundaries, as we will see in section 6.5). In particular, we discretize the vortex sheet surface with a triangle mesh, and store circulation values, Γ_i, on each mesh edge i. The per-triangle vortex sheet strength is then:

$$\vec{\gamma}_p = \frac{1}{a_p}\sum_j \vec{e}_j \Gamma_j, \tag{6.4}$$

for triangle edges j, where $\vec{e}_j$ is the vector between the end points of edge j, and where a_p is the area of triangle p.

The vorticity on a triangle is then computed as required with the formula:

$$\vec{\omega}_p = a_p \vec{\gamma}_p = \sum_j \vec{e}_j \Gamma_j. \tag{6.5}$$

for triangle edges j, where $\vec{e}_j$ is the vector between the end points of edge j. The main advantage of using these scalar circulations as our fundamental variable is that they are conserved along particle paths. The vorticity stretching is thus implicitly handled by the stretching of the vortex sheet mesh. Recomputing the vortex sheet strength when needed for the velocity update is done according to (6.5). Using (6.5) also guarantees that $\vec{\omega}$ is tangent to the triangle plane.

We apply a buoyancy force $\alpha\vec{g} = \langle 0, \alpha, 0\rangle$ to each triangle as $\Delta\vec{\omega}_i = \alpha\vec{g} \times \vec{n}_i A_i$ for triangle normal $\vec{n}_i$ and area A_i. We then compute the change in circulation for each triangle edge by solving the overdetermined system of equations 6.5 for Γ as suggested by Stock et al. [SDT08]. We solve this in the least-squares sense. We note that recomputing the *total* edge circulations in this way would introduce numerical damping manifesting as undesirable loss of vorticity, so we only solve for the *change* in edge vorticity due to the buoyancy force.

With no solid boundaries, the velocity at any point in the domain can be computed explicitly with the Biot-Savart law, with S being the vortex sheet surface, and K being a (mollified) fundamental solution to Laplace's equation:

$$\vec{u}_{\text{fluid}}(\vec{x}) = \int_S \nabla K(\vec{x} - \vec{y}) \times \vec{\omega}(\vec{y}) dy \tag{6.6}$$

This is discretized on a mesh surface using midpoint quadrature as:

$$\vec{u}_{\text{fluid}}(\vec{x}) \approx \sum_j \nabla K(\vec{x} - \vec{c}_j) \times \vec{\omega}_j \tag{6.7}$$

where $\vec{c}_j$ is the barycentre of triangle j, and $\vec{\omega}_j$ is the constant vorticity on triangle j.

Since we require the velocity at each vertex, and every velocity evaluation depends on the entire surface, this direct summation has computational complexity $O(M^2)$ in the number of mesh vertices. Later in this chapter we will discuss using the Fast Multipole Method to achieve an approximation which runs in $O(M)$ time.

The gradient of the fundamental solution $\nabla K(\vec{x} - \vec{y})$ is singular when $\vec{x} = \vec{y}$ and becomes large as $\vec{x}$ approaches $\vec{y}$, therefore a mollified approximation is often used. Our implementation follows Beale and Majda's form [BM85]:

$$\nabla \tilde{K}(\vec{x} - \vec{y}) = -(\vec{x} - \vec{y}) \frac{1 - e^{(r/\beta)^3}}{r^3}, \tag{6.8}$$

where $r = ||\vec{x} - \vec{y}||$, and β is a mollification parameter.

As our mesh evolves due to this self-imparted velocity, it naturally expands and curls up. Without some form of surface mesh refinement, the simulation would quickly lose accuracy and visual appeal. On the other hand, we wish to keep the number of mesh elements as low as possible, to keep the computational cost down. Therefore, removing small elements (recoarsening) is also desirable. For mesh refinement and recoarsening operations, we use the open-source El Topo surface tracking library [BB09c]. This library also has the ability to perform changes in topology, such as merging and pinching. By merging nearby surface patches, we can eliminate very thin sheets, thereby further reducing the total number of mesh triangles while still maintaining a good quality simulation.

6.3.1 Circulation redistribution

El Topo's surface remeshing operations add and remove edges. As we are storing scalar circulation values on the edges, we must update these values as the mesh connectivity changes. One way of doing so is to compute per-triangle vorticity values using equation 6.5, performing the remeshing operations, then solving equation 6.5 for Γ in a least-squares sense. However, we have found that repeatedly solving this overdetermined system leads to diffusion of vorticity.

An alternative is to explicitly update the edge circulation values using some rules of thumb when a remeshing operation is performed. Though heuristic, we have found these rules reduce vorticity diffusion as compared to repeatedly projecting back and forth between edge and triangle values. Of the local remeshing operations available in El Topo, we use only edge splitting, edge collapsing, and mesh merging via deletion and creation of new triangles.

Edge split

As shown in figure 6.2, splitting edge AB with opposite vertices C and D introduces a new vertex E. We store the circulation value on the original edge AB, and assign it to both of the new edges:

$$\Gamma_{\mathrm{AE}} := \Gamma_{\mathrm{EB}} := \Gamma_{\mathrm{AB}}. \tag{6.9}$$

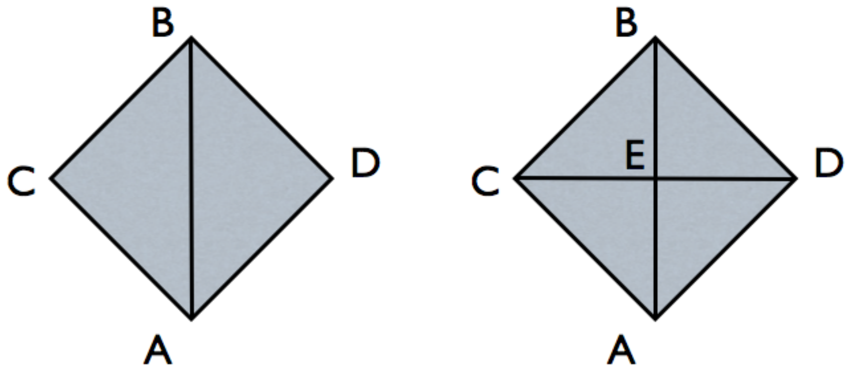


Figure 6.2: Edge split operation. New edges AE and EB are assigned circulation from original edge AB.

The two other new edges, CE and ED, receive no circulation. This ensures that the sum of vorticities over the new triangles (by equation 6.5) equals the sum of vorticities on the new triangles:

$$\vec{\omega}_{\text{ABC}} + \vec{\omega}_{\text{DBA}} = \vec{\omega}_{\text{AEC}} + \vec{\omega}_{\text{CEB}} + \vec{\omega}_{\text{DBE}} + \vec{\omega}_{\text{DEA}} \tag{6.10}$$

Edge collapse

When collapsing an edge, we simply allow the circulation on that edge to disappear from the system. Since we collapse only very short edges, the loss in circulation is generally minimal.

Mesh merge

El Topo merges surface meshes by removing two nearby edges and their incident triangles, and introducing new triangles, forming a neck between the surfaces [BB09c]. From the new edges introduced, we pick the two edges whose associated vectors (difference of end points) best match the removed edges, i.e. finding new edges *i* and *j* such that

$$||\vec{e}_i - \vec{e}_0|| + ||\vec{e}_j - \vec{e}_1||$$

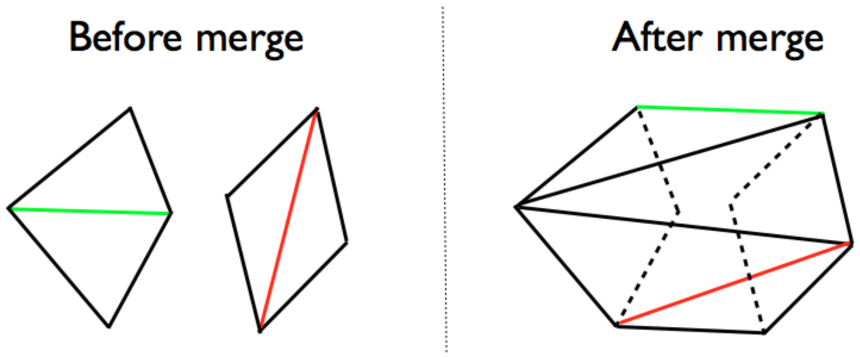


Figure 6.3: Edge merge operation. Left: The red and green edges and their incident triangles are removed from the mesh, leaving two quad-shaped holes. Right: The resulting holes are zippered with new triangles. Suppose the red and green edges in the right diagram are the closest matches to the corresponding red and green edges in the left diagram (comparing Euclidean distances of associated edge vectors). These new edges are then assigned the circulation values from the deleted edges.

is minimized, where edges 0 and 1 are the removed edges. We assign the original edge circulations to these "best match" new edges (see figure 6.3).

6.4 Solid boundaries

Boundary conditions for Lagrangian vorticity-based fluid animation have generally been constructed by defining a vortex sheet on the surface of the solid boundaries. Park and Kim [PK05] implemented boundary conditions in 3D by enforcing the tangential and normal components of the fluid velocity to be equal to those of the boundary using a vortex sheet, which resulted in an over-determined matrix solve. To avoid finding an optimal solution each time-step, they restricted themselves to a simple static boundary for which they precomputed a pseudo-inverse, requiring only a matrix-vector multiplication instead of a matrix solve. Weißmann and Pinkall [WP10] defined their vortex sheet in terms of a scalar potential, avoiding the extra dimensionality of the tangential vorticity field. They regularized the resulting matrix and also precomputed a factorization, again limiting their domain to a single boundary with static geometry.

We incorporate solid boundaries by adding an irrotational, divergence-free vector field that corrects the boundary velocity induced by the vorticity to account for solids. To do so we define the irrotational flow by the gradient of a scalar potential function Φ that satisfies Laplace's equation $\nabla^2\Phi = 0$ in the domain. The potential due to a single point charge with value σ, at point $\vec{y}$, is:

$$\frac{\sigma}{4\pi|\vec{x}-\vec{y}|}. \tag{6.11}$$

We represent Φ by a continuous distribution of point charges $\{\sigma(\vec{y})|y \in S\}$ where S is the boundary of our domain:

$$\Phi(\vec{x}) = \int_S \frac{\sigma(\vec{y})}{4\pi|\vec{x}-\vec{y}|} dy. \tag{6.12}$$

This representation of Φ is called the single layer potential. The normal derivative of the single layer potential has a discontinuity across the domain boundary. When satisfying boundary conditions, this leads to the following Fredholm integral equation of the second kind for σ [Kel29, Hes72]. Given the normal derivative f of Φ on the boundary we have:

$$f = -\sigma(\vec{x})/2 + \int_S \sigma(\vec{y}) \frac{\partial}{\partial n_x} \frac{1}{4\pi|\vec{x}-\vec{y}|} dy, \quad \vec{x},\vec{y} \in S. \tag{6.13}$$

Fredholm equations of the second kind are integral equations consisting of the identity operator plus an integral operator that is "compact". While the definition of compact requires an understanding of functional analysis and is therefore beyond the scope of this chapter, an integral operator is compact if the multiplicative terms in the integral are themselves square integrable, which is the case in (6.13). The solvability of Fredholm equations of the second kind can be determined by examining the null-space of the equation, similar to determining the solvability of linear systems. For smooth boundaries, (6.13) has only a trivial null-space and thus its solution exists and is unique for any right hand side f. In addition, solvable Fredholm equations of the second kind tend to lead to well conditioned numerical approximations. While our description of integral equations is cursory and incomplete, there exists ample mathematical literature for the interested. Solving (6.13) for σ and evaluating Φ from equation (6.12) satisfies the exterior Neumann problem for Laplace's equation,

$$\begin{aligned} \nabla^2\Phi &= 0 && (6.14) \\ \frac{\partial}{\partial n}\Phi(\vec{x}) &= f, \quad \vec{x}\in S. && (6.15) \end{aligned}$$

Let $\vec{u}_{\text{fluid}}$ be the velocity due to the fluid, and $\vec{u}_{\text{solid}}$ be the velocity of the solid object. Setting $f = (\vec{u}_{\text{solid}} - \vec{u}_{\text{fluid}})\cdot\vec{n}$, solving for potential Φ, and calculating the gradient yields a potential flow which will cancel the normal component of velocity at the solid boundary, effecting a free-slip, no-entry boundary condition.

The advantages of this scalar formulation over the previously described vortex sheet boundary implementations are that it is solvable for arbitrary smooth solid boundaries and is particularly amenable to numerical solutions based on iterative matrix solvers and fast summation methods. Integral equations for implementing boundaries based on vortex sheets are more difficult to implement, and though one can derive Fredholm integral equations of the second kind for finding a boundary vorticity [CK00], these formulations become singular if the domain is multiply-connected.

Discretizing (6.13) with midpoint quadrature yields the following equation for triangle i:

$$f_i \approx -\sigma_i/2 + \sum_j \sigma_j \frac{\partial}{\partial n_i}\frac{1}{4\pi|\vec{c}_i - \vec{c}_j|}A_j \qquad (6.16)$$

Directly evaluating this summation for all triangles yields an $M\times\ M$ linear system where M is the number of triangles on the solid boundary. In practice, this matrix is very well-conditioned; for such problems, iterative Krylov solvers such as BiCGSTAB have been observed to converge in $O(1)$ iterations, resulting in total complexity of $O(M^2)$ for the solid solve. We can additionally accelerate this to $O(M)$ by using the FMM to compute the matrix-vector product in the iterative solve rather than explicitly constructing the system.

Once we have solved for a density σ, we can compute Φ as in equation (6.12). The gradient of Φ is then evaluated as:

$$\nabla_x\Phi(\vec{x}) = \int_S \frac{-\sigma(\vec{y})}{4\pi|\vec{x}-\vec{y}|^3}(\vec{x}-\vec{y})dy \qquad (6.17)$$

The midpoint quadrature rule for this integral yields:

$$\nabla_x \Phi(\vec{x}) \approx \sum_j \frac{-\sigma_j A_j}{4\pi|\vec{x}-\vec{c}_j|^3}(\vec{x}-\vec{c}_j) \tag{6.18}$$

We use this to modify our fluid surface vertex velocities:

$$\vec{u}_{\text{final}}(\vec{x}) = \vec{u}_{\text{fluid}}(\vec{x}) + \nabla\Phi(\vec{x}) \tag{6.19}$$

6.5 Fast Multipole Method

The Fast Multipole Method (FMM), initially introduced by Greengard and Rhoklin [GR87], reduces the asymptotic complexity of the N-body problem from $O(N^2)$ to $O(N)$. The FMM is used in two different areas of our fluid simulation. Computing the velocity from the vorticity using the Biot-Savart law can be accomplished using an FMM for each component of the vorticity. In addition, computing the matrix multiply in the iterative solver for the solid-boundary potential flow can be reduced from $O(M^2)$ to $O(M)$ by using the FMM to approximate Φ:

$$\Phi(\vec{x}_i) = \sum_j \frac{\sigma_j}{|\vec{x}_i - \vec{x}_j|}, \quad i,j = 1..N, \tag{6.20}$$

and its gradient,

$$\nabla\Phi(\vec{x}_i) = \sum_j \frac{-\sigma_j(\vec{x}_i - \vec{x}_j)}{|\vec{x}_i - \vec{x}_j|^3} \tag{6.21}$$

for a set of points $\vec{x}_i$ and "charges" σ_i. Our 3D FMM implementation follows that described by Cheng et al. [CGR99].

Mild controversy surrounds the FMM's claim to linear complexity, since a straightforward computation of the octree data structure required by the FMM is $O(N \log N)$. We compute our octree in $O(N)$ complexity. To do so, we specify beforehand the maximum depth of the octree. We then compute unsigned integer coordinates for each particle that correspond to the grid cell of the finest octree subdivision that contains the particle. A single key is then created by interleaving the bits of each of the three coordinates creating a 3D Morton ordering of the occupied grid cells. We sort the particles by these interleaved keys using a radix sort

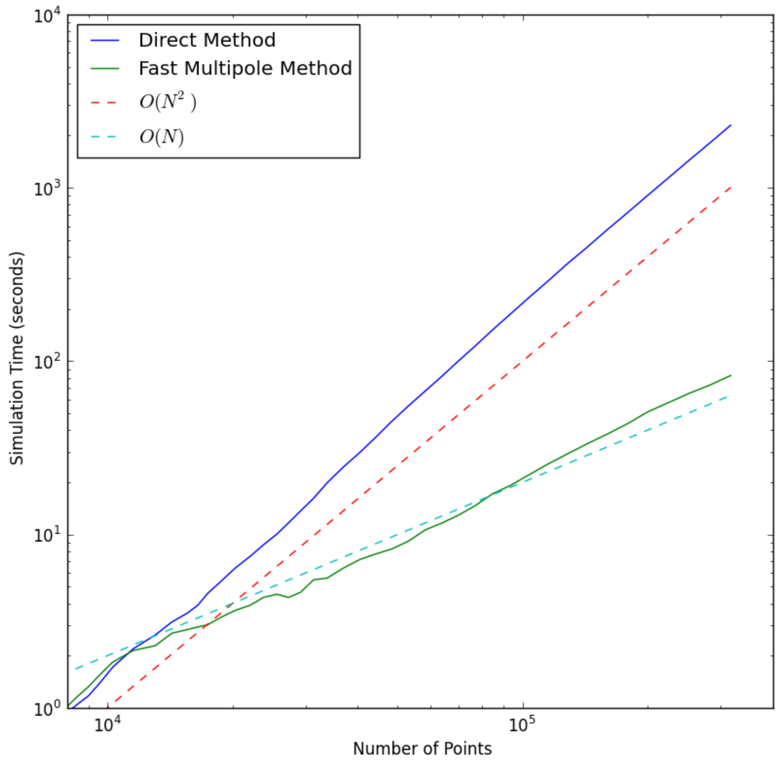


Figure 6.4: Simulation time per frame for computing velocity via the Biot-Savart law, for both the direct method and Fast Multipole Method

in $O(N)$. The result of the sort is that the particles are arranged in a linear octree, from which the hierarchical octree structure can be constructed in $O(N)$.

The log-log plot in figure 6.4 shows the order of complexity for computing velocity due to vorticity using the direct summation method vs. the FMM. Performance plots for the solid solver are shown in figure 6.5.

6.6 Rendering

Our interactive-rate smoke rendering is accomplished through a set of OpenGL shaders, which compute the optical depth of the smoke volume. We then apply

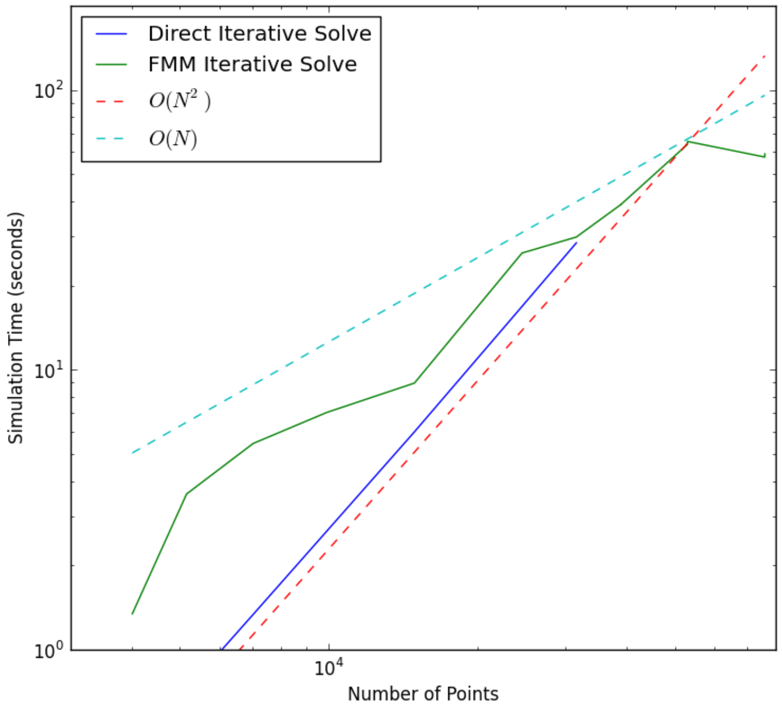


Figure 6.5: Simulation time per frame for computing the single layer potential on a sphere using BiCGSTAB, computing the matrix multiply with both the direct method and Fast Multipole Method. Note that the direct method is nearly exactly $O(N^2)$ showing the $O(1)$ convergence of BiCGSTAB

a zero-albedo absorption model based on this depth, taking into account the solid objects in the scene. More sophisticated real-time smoke rendering methods do exist (cf. [ZRL$^+$08]), and exploring self-shadowing and light scatter for triangle mesh models in real time is an area of potential future work.

As Brochu and Bridson [BB09c] point out, for high quality cinematic smoke rendering in a typical film pipeline, the critical bottleneck is often the load on the file system and network from tracking smoke per frame with hundreds of millions of soot particles, or similarly high resolution grids. The surface mesh representation is vastly more efficient for capturing the volumetric bulk of the smoke, and as Brochu and Bridson note can be directly ray-traced for self-shadowing smoke rendering. We extend their idea noting that the interior of the surface mesh, at render time, can easily be rasterized into a lower resolution 3D texture for scattering computation, at least using a box filter via polygon clipping; if this grid is aligned with the light source or the view, especially efficient calculation of single-scattering is possible. The direct render can use triangle rasterization of the mesh, storing depths in an A-buffer, looking up into the 3D texture for smooth lighting calculations. Going even further, we can track the age of mesh elements since their emission from a source, and splat this into another render channel to provide age-proportional blurring, thus giving the visual effect of smoke dissipation from turbulent mixing or diffusion at render time, for every frame in parallel. Figure 6.6 illustrates the effect of an image-space age-dependent blur.

6.7 Results

Our FMM-accelerated vortex sheet solver was implemented in C++, using El Topo for surface tracking, and tested on several Linux and Mac OS X computers. Figures 6.4 and 6.5 show that our tests have empirically near-linear time complexity. Since we do not rely on precomputing the solution to the exterior Neumann problem, we can easily handle non-rigidly deforming solid objects, as shown in figure 6.1.

Our interactive smoke renderer implemented in GLSL runs at 4 FPS for 168K triangles on an AMD Radeon 6770M with 512 MB of memory (shown in figure 6.1). Our proof-of-concept software renderer uses CPU-based volumetric rasterization and self-shadowing to produce the images shown in figure 6.6. These images were produced from 126K triangles at 1600×1200 resolution with a $167 \times 125 \times 82$ shadow texture in around 5.5 s on a single thread of a 2.3GHz Intel Core i7 CPU, including all file I/O.

6.7.1 Mesh simplification

As mentioned in section 6.3, remeshing operations are crucial for allowing the surface mesh to move freely while keeping the number of triangles manageable. The effectiveness of edge splitting and collapsing to control the explosion of mesh el-

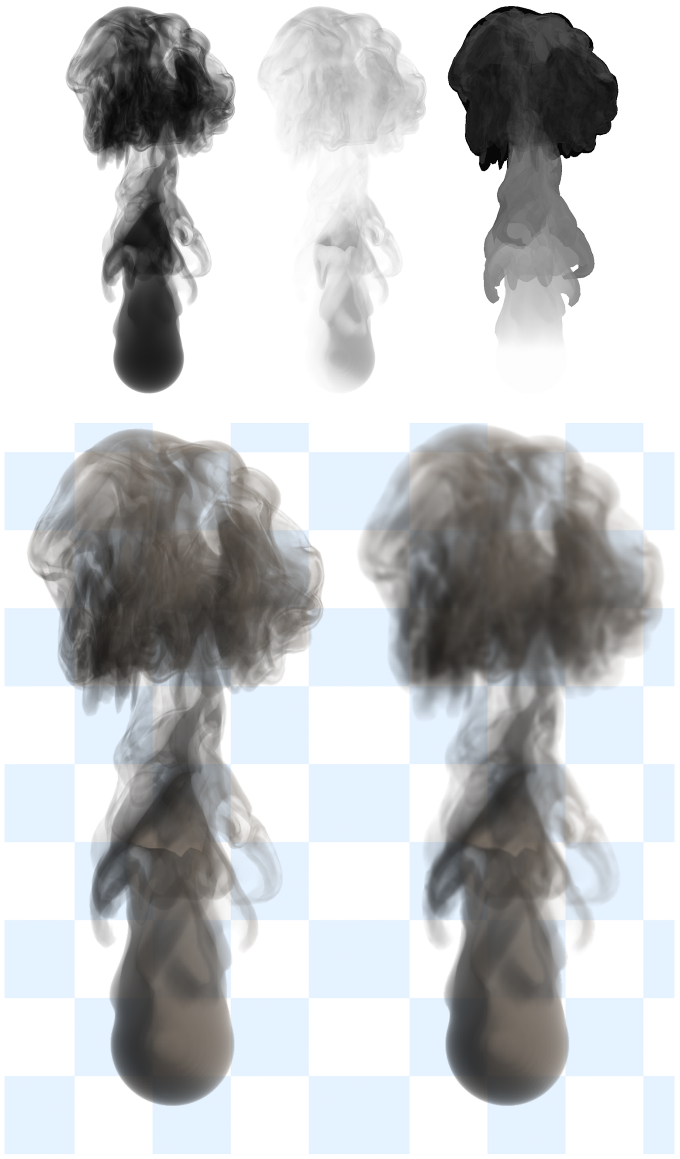


Figure 6.6: Our proof-of-concept volume renderer produces, from left to right on the top row, an alpha channel from the smoke mesh, a scattering channel with self-shadowing, and an image-space age channel. Below on the left is a sharp composite of the first two, and below on the right includes an age-dependent blur, simulating diffusion in image-space at render-time.

ements has been shown by Stock et al. [SDT08] and others, but we also make use of El Topo's topology change operations — specifically the merging and pinching of mesh surfaces. We have found this to be an effective tool in mitigating the explosion of surface area (and number of mesh elements), which is a known problem in vortex sheet simulations.

As an example, we ran a simulation twice, once with topology changes enabled, and once with these changes disabled. The number of triangles per time step is shown in figure 6.7. Note that without topology changes enabled, the number of triangles grows exponentially, but with aggressive topological merging and splitting, the number of triangles remains bounded (see the accompanying video).

Pfaff et al. [PTG12] also address this explosion in the number of mesh elements. Their approach identifies triangles which are deep within the volume of smoke (using a grid-based signed distance field), or otherwise occluded from the camera view, and marks them for deletion. By contrast, our approach uses only Lagrangian mesh-based operations, without additional structures or heuristics.

6.8 Future work

We have demonstrated that interactive-rate, near-cinematic-quality smoke simulation *and* rendering is within reach, but there are many avenues to explore further:

- Code optimization and parallelization of the FMM solver to achieve interactive simulation rates.
- Porting our software A-buffer rasterizer to run self shadowing and diffusion in GPU hardware.
- View-dependent, level-of-detail mesh refinement, coarsening, and topology changes.
- Incorporating the creation of vorticity from solid interactions (vortex shedding).
- Simulation of flame front propagation to achieve vortex sheet based fire simulation, combined with real-time lighting techniques on the rendering side.

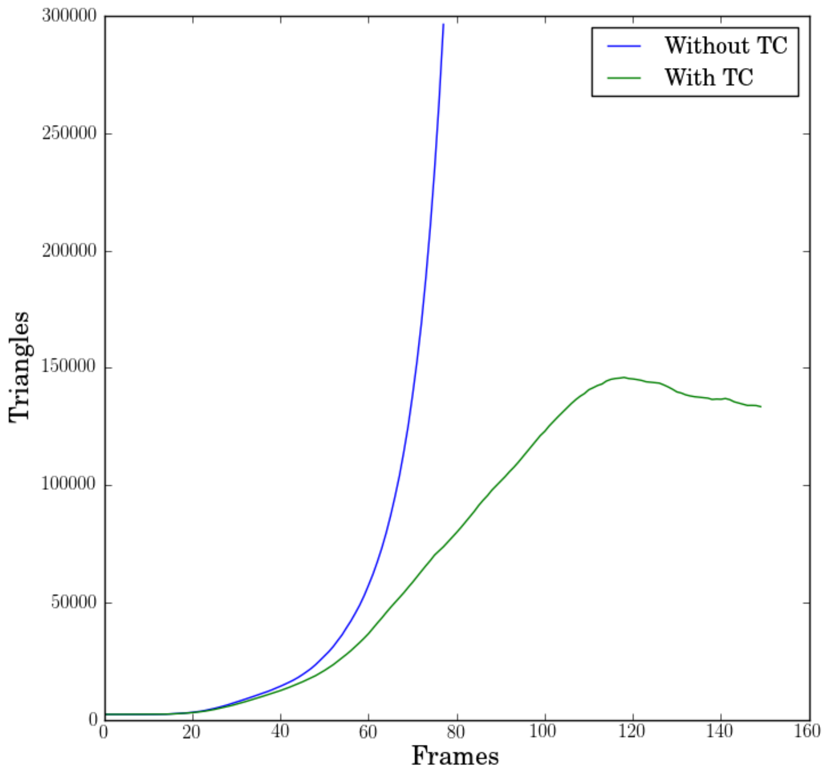


Figure 6.7: Geometry creation when simulating a smoke plume without topological changes (Without TC) and with topological changes (With TC).

- Simulation of ocean wave free surfaces — much of our vortex sheet work could apply to rough and vast simulations, which are currently infeasible with volumetric simulation methods.

Chapter 7

Matching fluid simulation elements to surface geometry and topology

7.1 Introduction

One of the most visually compelling aspects of liquids is the variety of complex thin sheets and droplets that arise during splashing. However, these remain among the most difficult features to simulate plausibly and accurately with existing techniques. Such detailed behaviour is extremely computationally expensive to resolve because of the tremendous grid resolution required for both the fluid solver and the surface tracking mechanism.

Recent advances in explicit surface tracking with triangle meshes [WTGT09, BB09c, Mül09] have made feasible the geometric representation and manipulation of small features, without the loss of detail exhibited by implicit surface methods. However, 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 sur-

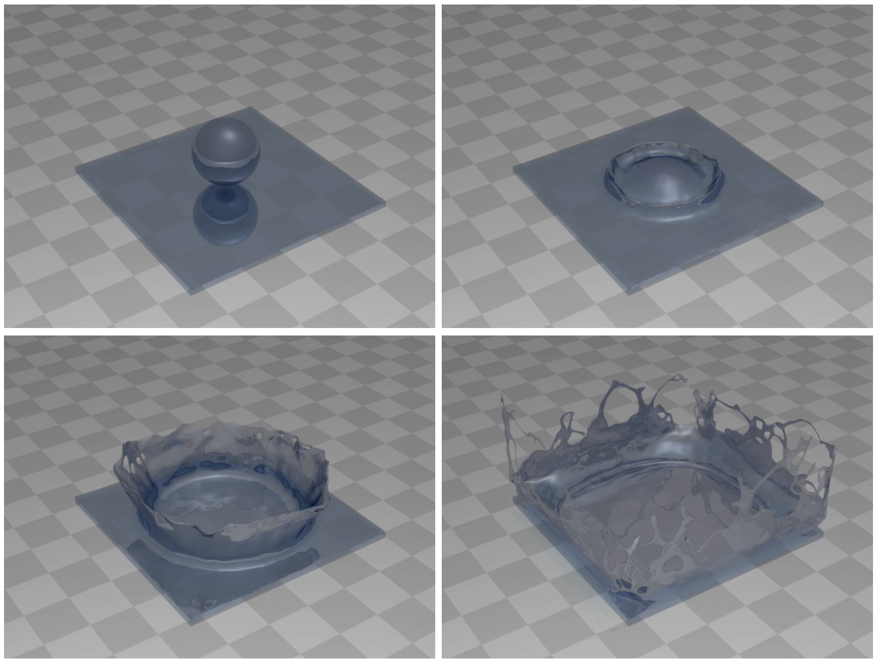


Figure 7.1: Coupling an explicit surface tracker to a Voronoi simulation mesh built from pressure points sampled in a geometry-aware fashion lets us capture very fine details in this sphere splash animation that uses only 314K tetrahedra.

face 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 [BGOS06, KSK09]. When combined with surface tension forces, noisy sub-mesh details can also severely hamper stability if they are not artificially smoothed out.

We will address these problems by constructing a simulator that "sees" every detail in the explicit liquid surface. We carefully generate pressure sample points near the liquid surface, 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. We couple this with a semi-Lagrangian advection scheme and a new approach to surface tension, arriving at a complete liquid simulator.

In summary, our key contribution is coupling an explicit surface tracker to a Voronoi-based liquid simulator with:

1. a pressure sample placement strategy that captures the complete liquid surface geometry,

2. an accurate surface tension model combining mesh-based curvature estimates and ghost fluid boundary conditions,

3. embedded free surface and solid boundary conditions adapted to Voronoi cells, avoiding the need for more onerous conforming tetrahedral mesh generation,

4. and a new velocity interpolant over unstructured meshes.

The practical benefits of such a system include:

1. improved animation of detailed liquid features, including very thin sheets, tendrils and droplets,

2. elimination of surface noise in explicit surface tracking without non-physical smoothing,

3. more detailed and less damped surface tension effects,

4. and faster semi-Lagrangian advection on unstructured meshes without increased dissipation.

7.2 Related work

7.2.1 Fluid animation on unstructured meshes

The use of unstructured and semi-structured meshes has a long history in computational fluid dynamics, and has gained traction in computer animation as well. An important reason for their popularity is that careful control of the mesh geometry can often simplify the discretization or improve accuracy. For example, matching the simulation mesh to solid walls makes the no-flow boundary condition trivial: just set the normal velocity on the boundary faces to zero. Adaptivity can also be easily introduced by grading mesh elements as desired. These ideas have been studied extensively in the context of finite volume methods for tetrahedral meshes

[FOK05, FOKG05, KFCO06, CGFO06, ETK$^+$07, WBOL07, CFL$^+$07], and as a result, many of the features present in standard grid-based solvers are now supported on tetrahedra, including free surfaces and implicit coupling to dynamic rigid and deformable objects. Work by Batty et al. [BXH10] augmented this approach with embedded boundaries [ENGF03, BBB07], to improve free surface accuracy and reduce the complexity of remeshing. Our method extends these advantages to Voronoi meshes.

In a related approach, Sin et al. [SBH09] developed a particle-based method that solves a finite-volume pressure projection on a Voronoi diagram generated from the set of liquid particles. An advantage of this approach is that the pressure degrees of freedom are directly tied to the number of particles, so there can never be a resolution mismatch between the surface geometry and the simulator. This idea motivates the work in this chapter.

7.2.2 Surface resolution vs. simulation resolution

Most of the earliest level-set-based liquid animation papers [FF01, EMF02] used one level set sample per pressure sample (i.e., per grid cell), presumably to avoid resolution inconsistencies. Goktekin et al. [GBO04] experimented with using a double-resolution level set, which improved volume conservation while introducing slight artifacts. Bargteil et al. [BGOS06] similarly coupled their octree-based semi-Lagrangian contouring surface tracking method to a regular grid fluid simulator. Moreover, they explicitly discussed potential artifacts due to this resolution mismatch, including surface details or noise being erroneously maintained and the simulator interpreting disconnected fluid regions as connected. Similarly, Kim et al. [KSK09] coupled a high resolution particle level set surface to a low resolution ghost-fluid-based liquid simulator. To ensure that all liquid geometry is captured by the pressure projection, their method resamples the high resolution level set onto a new level set at the grid resolution, after conceptually inflating it. This ensures that the resampled surface completely encloses the original so that no liquid components are missed. However, this can exacerbate other resolution mismatch artifacts, since liquid components behave as though they are half a cell-width larger than they appear visually. To reduce the retention of sub-grid surface noise in the high resolution surface, Kim et al. also smooth the surface by applying artificial diffusion.

Mismatched resolutions have also been applied in embedded deformable object simulation, such as in work by Wojtan & Turk [WT08]. They used a high resolution mesh surface coupled to a low resolution finite element solid simulator. This approach is often quite useful for solids, because solids are generally expected to maintain their surface details over time. Forcing the simulation mesh to have the same topology as the high resolution embedded surface mesh can also improve realism [TSB$^+$05, NKJF09].

Terashima and Tryggvason [TT09] introduced a method for coupling the ghost-fluid method [FAMO99] on a Cartesian grid with an explicit surface tracker, describing an approach for extrapolating surface normals from the surface mesh onto the grid.

Müller [Mül09] introduced a liquid simulation coupled with an explicit surface tracking method, using an enriched set of marching cubes templates on a regular grid to rebuild the surface mesh at each time step. This ensures the simulation grid has the same level of detail as the surface. Wojtan et al. [WTGT09] use a similar approach, but reconstruct the surface only on grid cells where the surface topology is more complex than the implicit grid representation. This approach maintains surface detail smaller than the scale of a grid cell; they do not specifically handle these small details with simulation, but since their simulation treats a cell with any geometry as liquid, the surface is advected in a plausible way.

In work developed concurrently with the publication of this chapter, Wojtan et al. [WTGT10] have also addressed the sub-grid-scale detail problem with a different, somewhat complementary approach: simplifying the surface to match the resolution of the simulation, rather than adapting the simulation to match the surface.

7.2.3 Surface tension

Approaches to surface tension generally fall into two categories. The first comprises methods that apply surface tension as a body force restricted to a region around the interface through the use of a smeared delta function [BKZ92, HK03, ZYP06, WTGT09]. Methods in the second category apply the surface tension force *discontinuously* at the interface, typically by integrating it as a boundary condition on the pressure projection step. This approach is exemplified by the ghost fluid method and related approaches [ENGF03, HK05, HSKF07], and has been shown to provide more realistic results.

Surface tension models can also be compared in terms of how they actually compute the surface tension force itself. In level set-based schemes, finite differences on the level set values are often used to estimate mean curvature: however, this has been shown to be rather inaccurate without careful modification (e.g., [Shi07]) and cannot capture small details. If a surface mesh is available, a more accurate approach is to compute surface tension forces using either mesh-based curvature operators (e.g., [MDSB02]), or, as proposed recently, as a physical tension on the surface mesh geometry [TBE$^+$01, PN03, Bro06, WT08].

Concurrent with the work in this chapter, Thürey et al. [TWGT10] introduced an approach to simulating surface tension which uses explicit surface meshes. They set boundary conditions for the pressure projection by fictitiously advecting the surface mesh forward according to volume-preserving mean curvature flow, following ideas by Sussman and Ohta [SO09].

7.3 Algorithm outline

We simulate incompressible, inviscid liquids according to the Euler equations:

$$\frac{d\vec{u}}{dt} = -(\vec{u}\cdot\nabla)\vec{u} - \frac{\nabla p}{\rho} + \frac{\vec{F}}{\rho} \tag{7.1}$$

$$\nabla\cdot\vec{u} = 0. \tag{7.2}$$

Our numerical solution employs semi-Lagrangian advection and an embedded-boundary finite volume pressure projection. We generally follow the tetrahedral scheme of Batty et al. [BXH10] with modifications to use Voronoi meshes instead. Like Sin et al. [SBH09], we place pressure samples on the vertices of a Delaunay tetrahedral mesh, corresponding to the sites of the dual Voronoi diagram (Figure 7.7a and Figure 7.7b). Normal components of velocity lie on the faces of the Voronoi cells, so that the velocity sample is parallel to the line segment connecting the pressure samples in the Delaunay mesh. This configuration requires a slightly different velocity reconstruction compared to previous methods, but semi-Lagrangian advection is otherwise straightforward.

For front tracking, we used the El Topo code developed in Chapter 3, in particular using its triangle mesh surface to determine the location of pressure samples for our Voronoi simulation mesh.

Purely explicit front tracking algorithms generally use mesh refinement and coarsening to maintain a high quality discretization as the surface deforms. As described in Chapter 3, El Topo uses a sequence of edge subdivision, collapse and flipping operations, combined with null-space Laplacian smoothing. While these operations change mesh connectivity, they are designed to be geometry-preserving. For example, the smoothing moves vertices only in the null space of the local quadric metric tensor [GH97], as suggested by Jiao [Jia07]. If the vertex lies on a locally smooth patch it is moved in the plane tangent to the surface, but if on a ridge or corner it is moved only along this line. Therefore, sharp features are preserved, allowing the present algorithm to handle them physically.

The solver runs through the following stages each time step:

1. Advect the explicit surface with El Topo.
2. Generate a new simulation mesh as the Voronoi diagram of a lattice with extra samples near the liquid surface (Section 7.4).
3. Advect velocities onto the new mesh with semi-Lagrangian advection (Section 7.6).
4. Add external forces—typically just gravity.
5. Solve for the embedded-boundary pressure projection on the Voronoi mesh, including surface tension forces (Section 7.5).

7.4 Mesh generation

An advantage of a Voronoi-based discretization is the freedom to explicitly choose pressure sample locations, which is critical for accurate ghost fluid free surface

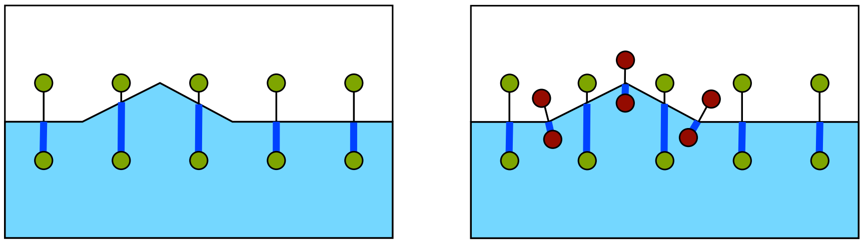


Figure 7.2: Left: Even with the ghost fluid method, regular sampling may miss surface details which do not align with the simulation mesh, such as this wave crest. Right: Adaptive samples (shown in red) placed on either side of each mesh vertex ensure that all geometric detail is resolved by the simulation.

conditions as the signed distance at these samples communicate the surface geometry to the solver. We can visualize the solver's "knowledge" by contouring this level set: Figure 7.2 and Figure 7.3 illustrate how uniform sampling may fail.

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 surface tension where spurious noise affects the curvature estimates and induces disastrously large yet futile compensating velocities that destabilize the simulation. 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. 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.

Comparison to Adaptive Lattices: The brute-force approach to these issues is to locally refine using octree grids or graded BCC lattice tetrahedra to capture smaller features. However, this scales poorly since many of the extra samples yield little benefit, while incurring memory and computational overhead. Furthermore, there remains no guarantee that features below the smallest grid cell size will be

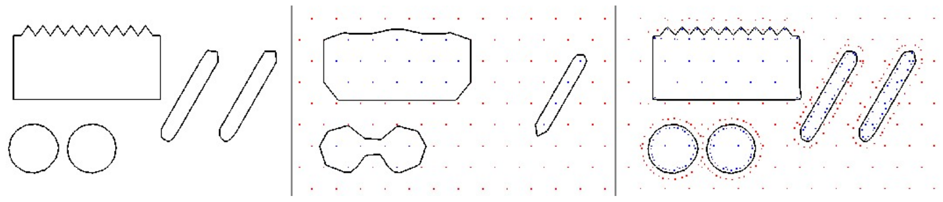


Figure 7.3: Left: The input surface geometry. Centre: The resulting surface after resampling onto a regular lattice simulation mesh. Note the spurious topology change, rounding of sharp features, aliasing of high frequency details, and the complete disappearance of one small fluid component due to poor placement relative to the mesh samples. Right: The resampled surface after adding geometry-aware sample points to the simulation mesh; the result is much more consistent with the input. (Mesh sample locations are indicated by points, colored blue when inside, red when outside.)

captured. By choosing sample points to precisely capture the geometry rather than naïvely increasing sample density, we can guarantee sampling of features which would require potentially orders of magnitude more samples with pure adaptive lattices.

Comparison to Conforming Tetrahedra: While the tetrahedral method of Chentanez et al. [CFL$^+$07] also builds a volumetric mesh that attempts to respect the liquid surface, it matches boundary faces rather than positioning pressure samples. This is considerably more difficult than non-conforming Delaunay tetrahedralization, and generally requires more Steiner points, worse-shaped tetrahedra, and/or the loss of the Delaunay property. Since our method uses embedded boundary conditions (described in Section 7.5), we do not require conforming elements. (Note that this advantage is shared by the method of Batty et al. [BXH10].) Moreover, the position of pressure samples plays a more important role in free surface conditions than the position of element faces. As accuracy requires that tetrahedral schemes store pressures at circumcenters [KFCO06, BXH10], and since circumcenters often lie outside their associated tetrahedra, even filling a thin feature with conforming tetrahedra provides no guarantee that its interior will be sampled at all.

7.4.1 Pressure sample placement strategy

We begin by choosing a characteristic length scale for the simulation, Δx, and configure El Topo 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 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. 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 7.4). 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 [LSSF06] 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,

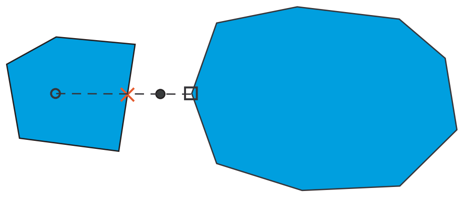


Figure 7.4: A pressure sample is seeded along the outward normal direction from a surface vertex (black square). The initial proposed pressure location (empty black circle) would land in the wrong component and potentially fail to resolve the intervening air gap. We instead place the final pressure sample (filled black circle) midway between the starting vertex and the first intersection point (red X).

by contrast, 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 7.6. 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 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* [Si06]. Figure 7.5 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.

7.5 Embedded boundaries on Voronoi meshes

We use finite volumes on a Voronoi mesh for the pressure projection step, similar to Sin et al. [SBH09]. However, rather than applying boundary conditions as they describe, we adapt the embedded boundary methods of Batty et al. [BXH10] to Voronoi meshes. Conveniently, the duality/orthogonality relationship between Voronoi and Delaunay meshes lets the accuracy benefits of the method carry over. Figure 7.7 illustrates our mesh configuration, and the computation of the required weights, as discussed below. We solve the resulting symmetric positive definite

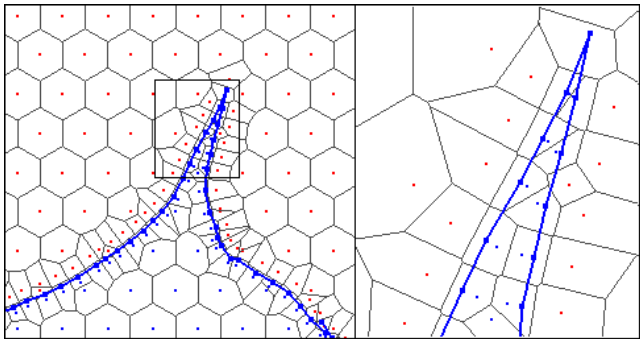


Figure 7.5: A 2D example of a thin feature simulated with our method. The zoom on the right illustrates the sample placement with respect to surface vertices, and the resulting Voronoi mesh. Notice that even the very sharp tip contains a pressure sample, as indicated by the surrounding Voronoi cell.

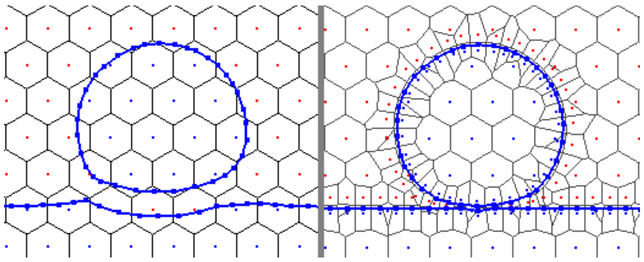


Figure 7.6: Left: Regular sampling erroneously identifies a topology change, causing a premature reaction in both liquid bodies. Right: Geometry-aware sampling responds correctly.

linear system using an incomplete Cholesky-preconditioned conjugate gradient solver.

To enforce embedded solid boundary conditions, we need to estimate the partial unobstructed area of each element face (Figure 7.7d). Batty et al. [BXH10] 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.

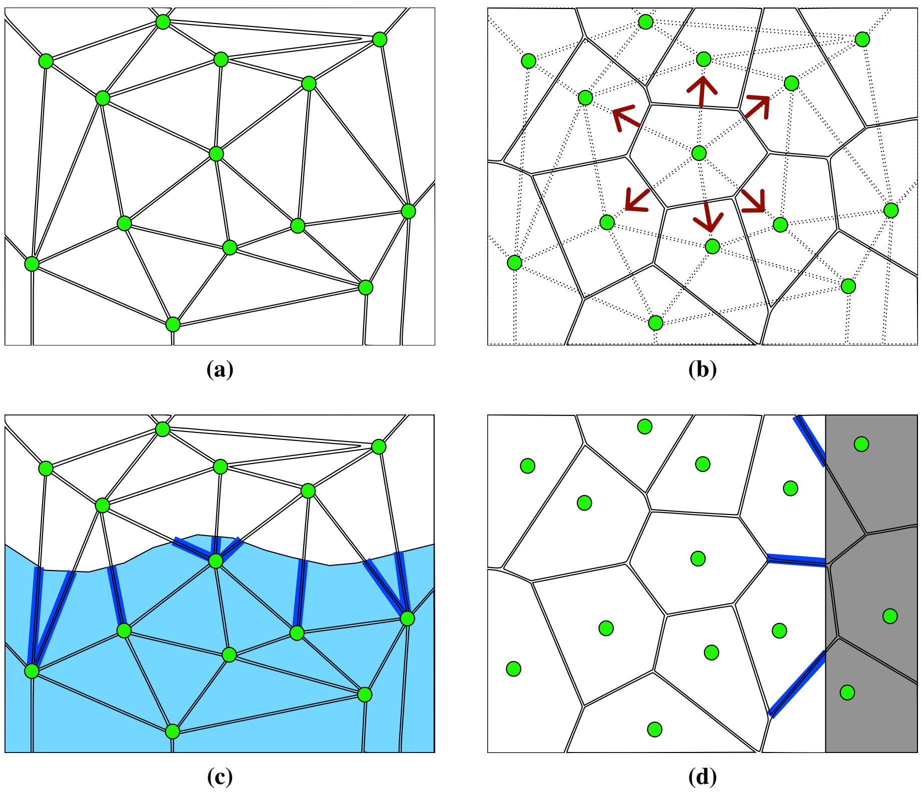


Figure 7.7: Pressure samples are shown as green circles. (a) Delaunay triangulation. (b) Voronoi diagram dual to the Delaunay triangulation (velocity components for the central cell are shown as red arrows). (c) Computation of ghost fluid weights on the edges of the triangulation. (d) Computation of 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.

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 7.7c). A slight improvement can be achieved by casting rays to find the exact position of the surface 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 [Bri08], and is easily adapted to tetrahedra.

7.5.1 Surface tension

To incorporate surface tension, we follow Enright et al. [ENGF03] in setting the free surface pressure $p_{\text{fs}} = p_{\text{air}} + \gamma\kappa_{\text{fs}}$, where p_{air} is the constant air pressure, γ is the surface tension coefficient and κ_{fs} is the mean curvature of the surface.

Rather than using level set finite differences, we compute curvature directly from the surface mesh to accurately capture high-frequency features. We chose the operator of Meyer et al. [MBLD02] because it provides high quality estimates using just the one-ring of triangles surrounding each vertex, but others could work too.

Curvature is evaluated at the intersection point between the triangle mesh surface and the line joining an interior pressure sample to an exterior one. Often this intersection point will coincide with a surface mesh vertex due to our choice of sampling scheme; where it does not, we use simple linear interpolation between the vertices of the surface triangle mesh. This method appears highly accurate, and leads to much less damping than that of Wojtan et al. [WTGT09].

7.6 Interpolation and advection

Velocity interpolation methods for unstructured meshes typically proceed in two steps [KFCO06, ETK+07, BXH10]. First, a full velocity vector is reconstructed at

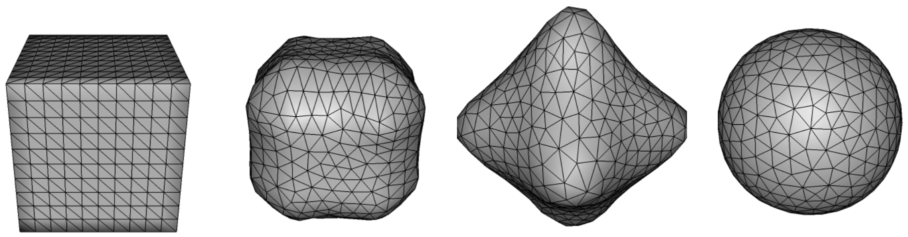


Figure 7.8: Our accurate surface tension model captures capillary waves even on relatively low resolution meshes. From left to right: A cube in zero gravity begins to collapse due to surface tension, inverts to become an octahedron, and continues to oscillate rapidly before settling down to a sphere.

selected mesh locations using a least-squares fit to the nearby velocity components. Then barycentric or generalized barycentric interpolation between those locations interpolates velocity over the full domain. Given such an interpolant, advection of velocities and geometry is straightforward. We follow this general framework, with two modifications.

In previous work, face normal components on tetrahedra were used to reconstruct velocities at circumcenters (Voronoi vertices). In our configuration, velocity components instead lie along the tetrahedra edges (Voronoi faces) so we perform the least squares fit on this data instead. We could then apply the usual generalized barycentric interpolant over Voronoi cells, but this is expensive [CFL$^+$07] and requires special case handling to avoid degeneracies [MDSB02]. A simple and fast alternative discussed by Klingner et al. and Chentanez et al. is to first interpolate velocities to Voronoi sites (tetrahedra vertices) and apply standard (and fast) barycentric interpolation over each tetrahedron. However, the interpolation onto tetrahedra vertices discards any local extrema at the Voronoi vertices, thereby severely over-smoothing the velocity field in practice, damping out interesting flow behavior.

Rather than discard extrema at Voronoi vertices, we use a slightly refined tetrahedral mesh that *includes* them. We conceptually tetrahedralize the Voronoi cells themselves by placing additional vertices at Voronoi face centroids and Voronoi sites (see Figure 7.9). Velocities for each of these new points need to be computed;

while previous work used the generalized barycentric interpolant for this transfer step, we found that simply averaging the velocities of the surrounding ring or cell of Voronoi vertices is quicker and equally effective. For maximum fidelity at the face centroids, we also replace the normal component of the averaged full velocity with the exact normal component already stored at the face. Simple and efficient barycentric interpolations can then be applied on the resulting smaller tetrahedra. Because the sharper, more accurate velocities at the Voronoi vertices are retained and merely augmented with additional data, this is far less dissipative, yielding results that closely match generalized barycentric interpolation (see Figure 7.10).

Lastly, note that reconstructions should only use face velocities which were assigned valid data by the pressure projection, and thus we can only reconstruct reasonable velocities inside the fluid. We therefore extrapolate velocities outwards from the fluid using a breadth-first graph propagation: each unknown point in a layer is set by averaging all adjacent known points from previous layers, repeating until we have a sufficiently large band of velocities surrounding the surface. This simple method, suggested in the context of cloth-fluid coupling by [GSLF05], sufficed for all our animations.

In summary, the steps of our interpolation scheme are:

1. Reconstruct full velocity vectors at Voronoi vertices using least squares.
2. Assign full velocity vectors to Voronoi sites and faces using simple averaging from neighboring vertices.
3. Subdivide the Voronoi cells into sub-tetrahedra using the sites and face centroids (see Figure 7.9).
4. Apply a simple graph-based extrapolation of velocities to fill in velocities near the liquid.
5. To interpolate at a point, locate the sub-tetrahedron containing the point and apply basic barycentric interpolation from its four associated data points (i.e. one site, one face centroid, and two Voronoi vertices).

One potential issue, not unique to our method, is that despite enforcing a lower bound on the distance between pressure samples, our unstructured sampling can

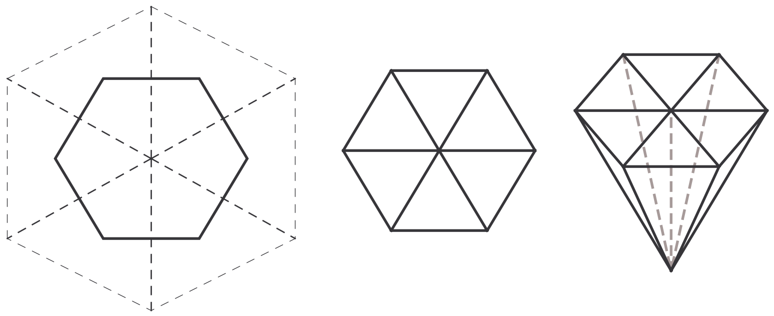


Figure 7.9: Rather than interpolating velocity over Voronoi regions directly, we tetrahedralize them and use simple barycentric interpolation. Left: A 2D Voronoi cell with standard dual Delaunay mesh overlaid. Centre: The same cell subdivided into smaller triangles that include the Voronoi vertices. Right: In 3D, each Voronoi face is triangulated using its centroid, and joined to its Voronoi site to build a tetrahedralization.

cause sliver tetrahedra in the unmodified Delaunay tetrahedralization. While we found this posed little problem for the pressure projection, it can cause the least squares velocity reconstructions to be ill-conditioned due to nearly coplanar faces. This can be readily resolved by requesting that the mesh generator add Steiner points to enforce fairly lax quality bounds; because our embedded pressure projection does not require the mesh generator to match boundaries, this is relatively inexpensive and effective. If mesh quality cannot be improved sufficiently, using additional nearby velocity samples in the reconstruction can ameliorate this at the cost of a smoother result.

7.7 Results

7.7.1 Sampling

The issues that arise from regular, non-geometry-aware pressure sampling are common and consistent across Cartesian grids, octrees, Voronoi meshes, and tetrahedral meshes. We will therefore use Voronoi meshes throughout, and simply compare our geometry-aware sampling against naïve regular sampling.

Surface Noise: As discussed above, regularly-spaced pressure samples can

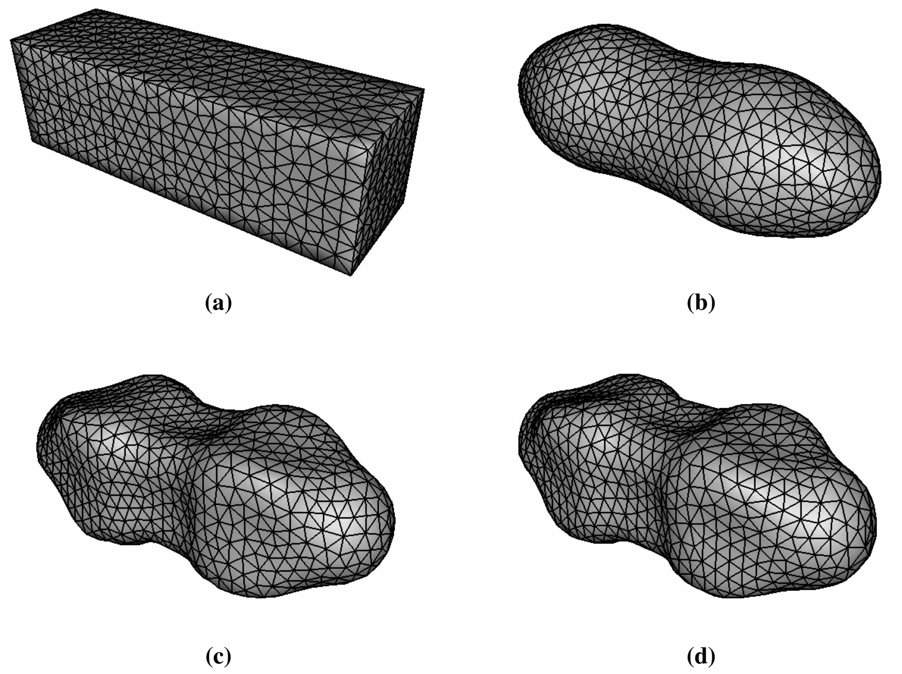


Figure 7.10: (a) Initial conditions for the collapse of a liquid block due to surface tension in zero gravity. (b) Naïve barycentric interpolation on tetrahedra generates very little detail. (c) Generalized barycentric interpolation over Voronoi cells retains interesting small scale structure. (d) Applying simpler barycentric interpolation over our refined tetrahedra produces qualitatively consistent results.

miss fine surface details, resulting in surface noise which is never smoothed out. Figure 7.11 illustrates that our sampling approach successfully resolves and corrects such small surface details. In contrast, regular samples cannot fully capture the initial surface perturbation, so it cannot be rectified. Though the ghost fluid method on regular samples does detect some differences in surface height, this actually exacerbates the problem because noisy sub-mesh details will appear to the simulator as rapid discontinuous changes in surface position over time, inducing noisy responses in the fluid velocity.

Topology Mismatch: Another visible artifact of using mismatched surface and simulation resolutions is topological inconsistencies. For example, a surface

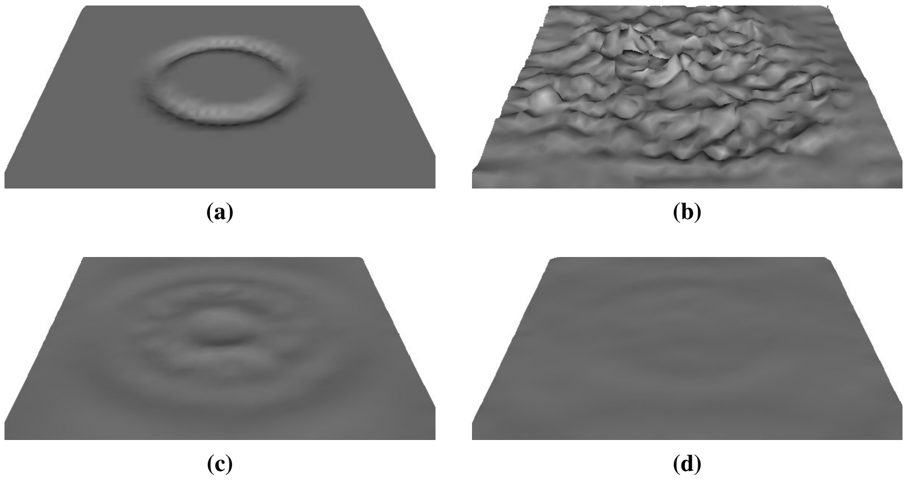


Figure 7.11: (a) A perturbation is introduced into a smooth surface. (b) On a regular tetrahedral mesh, the sub-mesh-resolution noise causes instability. (c, d) With adaptively-placed samples, the surface noise is accurately captured by the fluid solver and initially causes ripples before steadily settling.

with two disjoint volumes of liquid may appear to the solver as one volume, resulting in a premature response. Figure 7.6 shows a liquid drop impacting a still surface. With regular sampling, the droplet begins to influence the static liquid before the surfaces are actually joined. Because our adaptively-placed samples match the topology of the surface tracker, they easily correct this spurious motion. Figure 7.1 also features such a topological merge, along with many splitting and tearing operations, with timings listed in Table 7.1.

Thin Features: To illustrate our method's ability to animate thin features, Figure 7.12 shows a scene in which we drop a small sphere of liquid onto the ground. Thin sheets rapidly develop as the fluid spreads out across the floor. With regular pressure samples, sheets of this kind often end up between samples, effectively disappearing from the solver. Our sampling ensures that almost arbitrarily thin sheets of liquid remain visible to the solver, and as such, interesting rippling and splashing motion still occurs.

Our method also resolves thin sheets and small surface details generated by

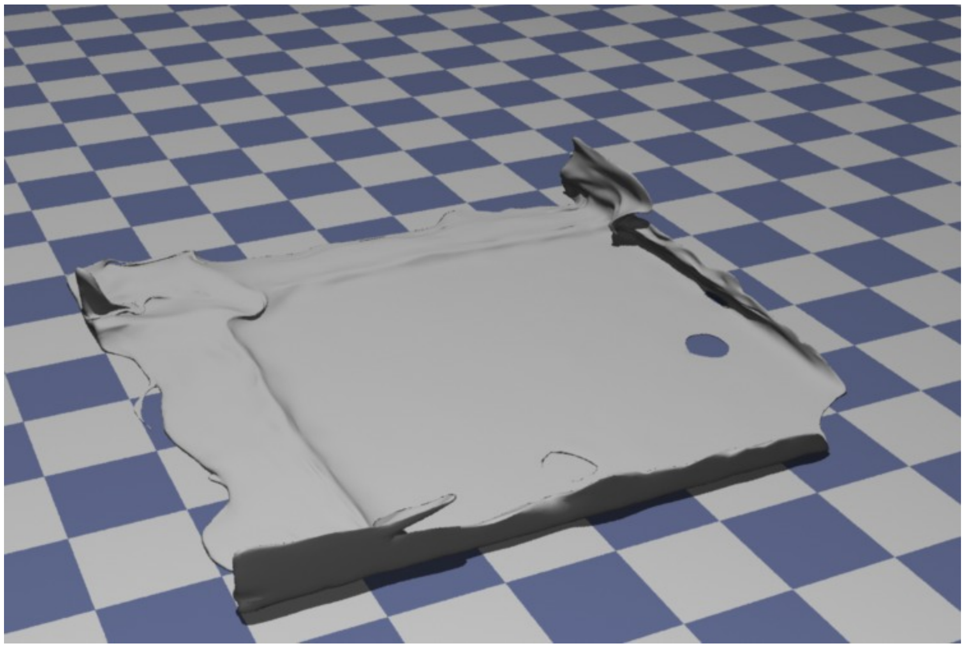


Figure 7.12: Seeding pressure samples directly inside the fluid volume allows us to capture thin sheets.

large splashes, as shown in Figure 7.1. To counteract gradual volume drift, we do add a corrective motion in the normal direction [Bro06, Mül09], which further aids in preserving thin sheets. Our video also includes an example of a column of liquid being released into a still pool. Although we are using only first-order accurate semi-Lagrangian advection, the liquid motion remains lively and active throughout. We suspect that because our method retains sharp wave peaks and splashes rather than continually eroding them, their extra kinetic and gravitational potential energy is retained in the simulation, accounting for this reduced dissipation.

Table 7.1 gives timings for our 3D examples. All figures are averages per frame and all timings are in seconds. These simulations used no more than 320K tetrahedra each, whereas recent tetrahedra-based free surface methods used up to 4 times more tetrahedra to achieve a similar level of detail.

Statistic	Thin sheet	Liquid column	Sphere Splash
# tetrahedra	141,701	197,911	313,587
Velocity reconstruction (s)	3	8	18
Surface tracking (s)	7	37	26
Volumetric remeshing (s)	15	39	69
Velocity advection (s)	7	18	15
Redistancing (s)	5	22	42
Pressure solve (s)	0.29	1.8	0.45
Total simulation time (s)	37	127	171

Table 7.1: Simulation statistics for 3D examples (all statistics are per-frame values, averaged over all frames).

7.7.2 Surface tension

Figure 7.8 illustrates the action of our surface tension model on a low resolution cube in zero gravity. Rather than quickly collapsing into a sphere, a cascade of detailed capillary waves propagate along the surface, causing it to oscillate rapidly. It initially inverts almost completely into an octahedron (the geometric dual of a cube), and continues to oscillate for many subsequent frames. To illustrate the benefits of our sampling approach in the context of surface tension, we launch an identical simulation using the same time steps on a regular mesh. Because this mesh cannot respond and correct high frequency sub-mesh details present in the curvature estimates, the simulation becomes unstable almost immediately. Applying an excessively strict timestep restriction only brings the simulation to a halt as the surface noise introduces increasingly sharp features.

Inspired by an example from the work of Wojtan & Turk [WT08], we run another zero gravity simulation on a rectangular block. Because our simulation does not use diffusive Laplacian mesh smoothing and applies accurate mesh-based surface tension forces discontinuously at the interface, we retain substantially greater detail in the resulting capillary wave motion.

7.7.3 Interpolation

We revisit our surface tension block example to compare different interpolation schemes. As seen in Figure 7.10, our barycentric method is substantially less damped than the naïve barycentric interpolation approach, and matches the more complex generalized barycentric interpolant.

7.8 Discussion and limitations

Our implementation is not heavily optimized, and we defer various potential performance gains to future work. Obvious optimizations include: reducing the number of tetrahedra through smarter sampling, improving the algorithm for point-location queries, and streamlining the construction of mesh data structures. More fundamentally, our Voronoi simulator is in many ways dual to a tetrahedral scheme, and for a given mesh the number of velocity samples is identical; we believe that approximately comparable costs are therefore reasonable to expect.

The main contribution of this chapter is the coupling of simulation elements to an existing explicit surface tracking method, and not the explicit surface tracking itself. Therefore, not all artifacts due to surface tracking are addressed. For example, El Topo delays handling some very difficult collisions for a few timesteps until the topological operations can be safely processed, which occasionally yields visible lingering surface noise. (Reducing the time step size can help by introducing fewer and simpler collisions, and more aggressive simplification can also be enabled by tuning the volume change tolerance that El Topo uses to decide whether to accept a given simplification.) Likewise, despite the use of feature-preserving mesh improvement, some popping artifacts due to on-the-fly remeshing are still visible in our animations. We chose El Topo because its resolution is not constrained to a regular grid and it is therefore able to showcase very thin features; nevertheless our method could adapt to any of the front tracking methods mentioned in Section 2.1.

Surface tension was only used for examples in Section 7.7.2 and Section 7.7.3. Our goal in many of the other examples was to highlight the ability to track thin sheets, whereas surface tension would break these sheets into droplets. Moreover, explicit surface tension schemes, such as the ghost-fluid-based method used in this chapter, suffer from a stringent $O(\Delta x^{\frac{3}{2}})$ time step restriction for stability, which is particularly costly when small scale capillary waves are not erroneously damped out. Pursuing a more efficient, fully implicit surface tension model is a promising future direction.

7.9 Conclusions and future work

We have shown that with careful placement of pressure samples, our Voronoi mesh-based fluid solver makes it possible for explicit surface tracking to achieve its full potential in capturing small scale liquid features. In addition, we adapted embedded boundary pressure projection techniques to Voronoi meshes, introduced a simple improvement to barycentric velocity interpolation for Voronoi/Delaunay meshes, and extended the ghost fluid surface tension model with mesh-based curvature in order to capture complex capillary waves with minimal damping.

Several directions for future work remain. For example, it may be possible to enhance our sampling scheme in various ways, perhaps by exploiting curvature adaptivity, topological information, or measures of vorticity and velocity variation. Likewise, improvements to front tracking would be welcome, such as curvature-driven adaptivity, or greater robustness and efficiency. Lastly, many common extensions to basic inviscid liquid simulation rely on regular grids, and would need to be adapted to accomodate our approach.

Chapter 8

Conclusion

Explicit surface tracking is a potentially powerful tool for physics-based animation and physically accurate simulation. Although at first it may appear overly complicated and challenging to deal with the geometric and topological problems that naturally arise in dealing with moving discrete meshes, in this thesis I have presented a few tools to make such an approach tractable.

Each self-contained chapter of this thesis contains conclusions specific to that particular branch of research; here we present an overall summary and some more general future research topics.

8.1 Summary

In Chapter 3, we presented a framework for intersection-free remeshing and topology changes on dynamic triangle meshes, and showed convergence in the presence of topology changes. The benefits of this approach were illustrated with a practical implementation of grid-free surface tracking which robustly handles changes in topology without mesh tangling or self-intersections, and tested with geometric flow applications.

Chapter 4 introduced two new approaches to continuous collision detection which are geometrically exact. We also presented a new, exact method for computing intersection parity between a ray and a bilinear patch. These approaches allow for parameter-free continuous collision detection implementations which are nearly optimal in the number of false positives, while incurring no false negatives. The run-time computational cost for these new collision queries is competitive with current state-of-the-art methods.

Chapters 5 and 6 applied the explicit surface tracking techniques described in the previous chapters by treating smoke as a surface. We first introduced a novel combination of explicit surface tracking and smoke simulation techniques, resulting in a method capable of high-resolution renders for a fraction of the storage cost of particle or grid-based smoke simulations. We next integrated the Fast Multipole Method into a working vortex sheet simulation, and introduced a fast smoke rendering technique. The result is a comprehensive method for smoke simulation and visualization that scales with the visual detail of the scene.

Finally, in Chapter 7 we combined explicit surface tracking with an adaptive fluid simulation which captures the complete surface geometry of the liquid. We used embedded free surface and solid boundary conditions, adapted to Voronoi cells, avoiding the need for more onerous conforming tetrahedral mesh generation. We also introduced an accurate surface tension model combining mesh-based curvature estimates and ghost fluid boundary conditions. The tangible benefits of our approach include improved animation of detailed liquid features, elimination of noise in explicit surface tracking without non-physical smoothing, and more detailed, less damped surface tension effects.

8.2 Future directions

Concurrent work on explicit surface tracking for physics-based animation by others [WTGT09, Mül09, WTGT10, TWGT10, YWTY12] hints at a shift in the way researchers and practitioners think of fluid surfaces. In the conclusions of previous chapters, we have suggested various future directions for this research, specific to the topics covered in this thesis, and so we will make broader suggestions here.

Several directions to improving the quality of explicit surface tracking methods remain. Improvements in remeshing which are well-suited to moving triangle mesh surfaces would benefit a wide range of methods. For example, in Chapter 4 we used a simple curvature-based metric to increase triangle density in some regions. Further development of this idea could allow for high-quality surfaces without the computational expense of employing a uniformly high-resolution mesh.

Our surface tracking approach relies on collision detection and resolution to prevent the mesh from becoming self-intersecting and tangled. Although we introduced a fully robust collision detection scheme, the collision *resolution* problem remains an open area of research. Developing a completely robust collision response system which is guaranteed to minimally perturb the surface vertex trajectories appears to be a difficult problem, but would be extremely beneficial for both surface tracking and the simulation of cloth and solids.

Promising future applications include boundary integral equations for elastic solids, fracture, and deep water simulation. Outside of visual effects, other applications for explicit surface tracking include the segmentation of volumetric medical data and the physical simulation of free surfaces.

Bibliography

[AG85] Christopher Anderson and Claude Greengard. On vortex methods. *SIAM Journal on Numerical Analysis*, 22(3):413–440, June 1985. → pages 92

[AN05] Alexis Angelidis and Fabrice Neyret. Simulation of smoke based on vortex filament primitives. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer animation*, SCA '05, pages 87–96, New York, NY, USA, 2005. ACM. → pages 81

[APKG07] Bart Adams, Mark Pauly, Richard Keiser, and Leonidas J. Guibas. Adaptively sampled particle fluids. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 26, July 2007. → pages 10

[AS95] David Adalsteinsson and James A. Sethian. A fast level set method for propagating interfaces. *Journal of Computational Physics*, 118(2):269 – 277, 1995. → pages 9

[AVDI03] Pierre Alliez, Éric Colin de Verdière, Olivier Devillers, and Martin Isenburg. Isotropic surface remeshing. In *Proceedings of the Shape Modeling International 2003*, SMI '03, pages 49–, Washington, DC, USA, 2003. IEEE Computer Society. → pages 16

[Bar68] Erwin H. Bareiss. Sylvester's identity and multistep integer-preserving Gaussian elimination. *Mathematics of Computation*, 22(103):pp. 565–578, 1968. → pages 63

[BB09a] Tyson Brochu and Robert Bridson. Animating smoke as a surface. Eurographics/ACM SIGGRAPH Symposium on Computer Animation (posters and demos), 2009. → pages 5

[BB09b] Tyson Brochu and Robert Bridson. Numerically robust continuous collision detection for dynamic explicit surfaces. Technical Report TR-2009-03, University of British Columbia, 2009. → pages 5

[BB09c] Tyson Brochu and Robert Bridson. Robust topological operations for dynamic explicit surfaces. *SIAM Journal on Scientific Computing*, 31(4):2472–2493, 2009. → pages 4, 95, 96, 102, 107

[BBB07] Christopher Batty, Florence Bertails, and Robert Bridson. A fast variational framework for accurate solid-fluid coupling. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 26(3), July 2007. → pages 110

[BBB10] Tyson Brochu, Christopher Batty, and Robert Bridson. Matching fluid simulation elements to surface geometry and topology. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 29:47:1–47:9, July 2010. → pages 7, 80

[BBP01] Hervé Brönnimann, Christoph Burnikel, and Sylvain Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. *Discrete Applied Mathematics*, 109:25–47, 2001. → pages 17, 71

[BFA02] Robert Bridson, Ronald Fedkiw, and John Anderson. Robust treatment of collisions, contact and friction for cloth animation. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 21(3):594–603, 2002. → pages 17, 32, 34, 35, 52

[BGB11] Haimasree Bhatacharya, Yue Gao, and Adam Bargteil. A level-set method for skinning animated particle data. In *Proceedings of the 2011 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '11, pages 17–24, New York, NY, USA, 2011. ACM. → pages 10

[BGOS06] Adam W. Bargteil, Tolga G. Goktekin, James F. O'Brien, and John A. Strain. A semi-Lagrangian contouring method for fluid simulation. *ACM Transactions on Graphics*, 25(1):19–38, 2006. → pages 12, 108, 110

[BHN07] Robert Bridson, Jim Houriham, and Marcus Nordenstam. Curl-noise for procedural fluid flow. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 26(3):46, 2007. → pages 44, 80, 81, 83

[BKZ92] Jeremiah Brackbill, Douglas Kothe, and Charles Zemach. A continuum method for modeling surface tension. *Journal of Computational Physics*, 100(2):335–354, 1992. → pages 111

[BM82] J. Thomas Beale and Andrew Majda. Vortex methods I: Convergence in three dimensions. *Mathematics of Computation*, 39(159):1–27, July 1982. → pages 92

[BM85] J. Thomas Beale and Andrew Majda. High order accurate vortex methods with explicit velocity kernels. *Journal of Computational Physics*, 58(2):188–208, 1985. → pages 94

[Bra92] Ken Brakke. The surface evolver. *Experimental Mathematics*, 1(2):141–165, 1992. → pages 13

[Bri08] Robert Bridson. *Fluid Simulation for Computer Graphics*. A K Peters, 2008. → pages 17, 120

[Bro06] Tyson Brochu. Fluid animation with explicit surface meshes and boundary-only dynamics. Master's thesis, University of British Columbia, 2006. → pages 112, 126

[BTW84] Carlos A. Brebbia, José Claudio Faria Telles, and Luiz C. Wrobel. *Boundary element techniques : theory and applications in engineering*. Springer-Verlag, 1984. → pages 11

[BWHT07] Adam W. Bargteil, Chris Wojtan, Jessica K. Hodgins, and Greg Turk. A finite element method for animating large viscoplastic flow. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 26(3), 2007. → pages 55

[BXH10] Christopher Batty, Stefan Xenos, and Ben Houston. Tetrahedral embedded boundary methods for accurate and flexible adaptive fluids. *Computer Graphics Forum (Proc. Eurographics)*, 29(2):695–704, 2010. → pages 110, 112, 115, 117, 118, 120

[CFL$^+$07] Nuttapong Chentanez, Bryan E. Feldman, François Labelle, James F. O'Brien, and Jonathan R. Shewchuk. Liquid simulation on lattice-based tetrahedral meshes. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer animation*, SCA '07, pages 219–228, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association. → pages 110, 115, 121

[CGFO06] Nuttapong Chentanez, Tolga G. Goktekin, Bryan E. Feldman, and James F. O'Brien. Simultaneous coupling of fluids and deformable bodies. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics Symposium on Computer animation*, SCA '06, pages 83–89, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association. → pages 110

[CGR99] Hongwei Cheng, Leslie Greengard, and Vladimir Rokhlin. A fast adaptive multipole algorithm in three dimensions. *Journal of Computational Physics*, 155(2):468–498, 1999. → pages 100

[CK00] Georges-Henri Cottet and Petros D. Koumoutsakos. *Vortex Methods: Theory and practice*. Cambridge University Press, 2000. → pages 92, 99

[CK10] Marcel Campen and Leif Kobbelt. Exact and robust (self-)intersections for polygonal meshes. *Computer Graphics Forum (Proc. Eurographics)*, 29(2):397–406, 2010. → pages 14

[CMTM94] Norishige Chiba, Kazunobu Muraoka, Hiromichi Takahashi, and Mamoru Miura. Two-dimensional visual simulation of flames, smoke and the spread of fire. *The Journal of Visualization and Computer Animation*, 5(1):37–53, 1994. → pages 92

[DeB74] Roger DeBar. Fundamentals of the KRAKEN code. Technical Report UCID-17366, California Univ., Livermore (USA). Lawrence Livermore Lab, 1974. → pages 9

[Dek71] Theodorus J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18:224–242, 1971. 10.1007/BF01397083. → pages 17, 54

[DFG99] Qiang Du, Vance Faber, and Max Gunzburger. Centroidal Voronoi tessellations: Applications and algorithms. *SIAM Review*, 41(4):637–676, 1999. → pages 16

[DFG$^+$06] Jian Du, Brian Fix, James Glimm, Xicheng Jia, Xiaolin Li, Yuanhua Li, and Lingling Wu. A simple package for front tracking. *Journal of Computational Physics*, 213(2):613 – 628, 2006. → pages 12

[DLG90] Nira Dyn, David Levine, and John A. Gregory. A butterfly subdivision scheme for surface interpolation with tension control. *ACM Transactions on Graphics*, 9(2):160–169, 1990. → pages 28

[DMSB99] Mathieu Desbrun, Mark Meyer, Peter Schröder, and Alan H. Barr. Implicit fairing of irregular meshes using diffusion and curvature flow. In *Proceedings of the 26th annual conference on Computer* *graphics and interactive techniques*, SIGGRAPH ’99, pages 317–324, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co. → pages 15

[EFFM02] Doug Enright, Ronald Fedkiw, Joel Ferziger, and Ian Mitchell. A hybrid particle level set method for improved interface capturing. *Journal of Computational Physics*, 183(1):83–116, 2002. → pages 9, 36, 42, 43, 48, 80

[EKS03] Olaf Etzmuß, Michael Keckeisen, and Wolfgang Straßer. A fast finite element solution for cloth modelling. In *Proceedings. 11th Pacific Conference on Computer Graphics and Applications, 2003.*, pages 244 – 251, 2003. → pages 55, 73

[EM90] Herbert Edelsbrunner and Ernst Peter Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9(1):66–104, January 1990. → pages 153

[EM94] Herbert Edelsbrunner and Ernst P. Mücke. Three-dimensional alpha shapes. *ACM Transactions on Graphics*, 13(1):43–72, 1994. → pages 10

[EMF02] Douglas Enright, Stephen Marschner, and Ronald Fedkiw. Animation and rendering of complex water surfaces. In *ACM Transactions on Graphics (Proc. SIGGRAPH)*, pages 736–744, 2002. → pages 110

[ENGF03] Douglas Enright, Duc Nguyen, Frederic Gibou, and Ronald Fedkiw. Using the particle level set method and a second order accurate pressure boundary condition for free surface flows. In *Proc. of the 4th ASME-JSME Joint Fluids Engineering Conference*, 2003. → pages 110, 112, 120

[Eri04] Christer Ericson. *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004. → pages 54

[ETK$^+$07] Sharif Elcott, Yiying Tong, Eva Kanso, Peter Schröder, and Mathieu Desbrun. Stable, circulation-preserving, simplicial fluids. *ACM Transactions on Graphics*, 26(1):4, 2007. → pages 110, 120

[Exo11] Exotic Matter. Naiad 0.6. 2011. → pages 92

[FAMO99] Ronald P. Fedkiw, Tariq Aslam, Barry Merriman, and Stanley Osher. A non-oscillatory Eulerian approach to interfaces in multimaterial flows (the ghost fluid method). *Journal of Computational Physics*, 152(2):457–492, 1999. → pages 111

[FB98] Pascal J. Frey and Houman Borouchaki. Geometric surface mesh optimization. *Computing and Visualization in Science*, 1:113–121, 1998. 10.1007/s007910050011. → pages 15

[FF01] Nick Foster and Ronald Fedkiw. Practical animation of liquids. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 23–30, New York, NY, USA, 2001. ACM. → pages 110

[FM97] Nick Foster and Dimitris Metaxas. Modeling the motion of a hot, turbulent gas. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, pages 181–188, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co. → pages 80

[FOK05] Bryan E. Feldman, James F. O'Brien, and Bryan M. Klingner. Animating gases with hybrid meshes. In *ACM Trans. Graph. (Proc. SIGGRAPH)*, pages 904–909, 2005. → pages 110

[FOKG05] Bryan E. Feldman, James F. O'Brien, Bryan M. Klingner, and Tolga G. Goktekin. Fluids in deforming meshes. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer animation*, SCA '05, pages 255–259, New York, NY, USA, 2005. ACM. → pages 110

[FSJ01] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. Visual simulation of smoke. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 15–22, New York, NY, USA, 2001. ACM. → pages 80

[GBO04] Tolga G. Goktekin, Adam W. Bargteil, and James F. O'Brien. A method for animating viscoelastic fluids. In *ACM Transactions on Graphics (Proc. SIGGRAPH)*, pages 463–468, 2004. → pages 110

[GD03] Philippe Guigue and Oliver Devillers. Fast ray-triangle intersection test using orientation predicates. *Journal of Graphics Tools*, 8(1):addendum, 2003. → pages 68

[GGL$^+$98] James Glimm, John W. Grove, Xiao Lin Li, Keh-Ming Shyue, Yanni Zeng, and Qiang Zhang. Three-dimensional front tracking. *SIAM Journal on Scientific Computing*, 19(3):703–727, 1998. → pages 8, 16

[GGLT99] James Glimm, John W. Grove, Xiao Lin Li, and Daoliang C. Tan. Robust computational algorithms for dynamic interface tracking in three dimensions. *SIAM Journal on Scientific Computing*, 21(6):2240–2256, 1999. → pages 12

[GGM93] Anne Greenbaum, Leslie Greengard, and Geoffrey B. McFadden. Laplace's equation and the Dirichlet-Neumann map in multiply connected domains. *Journal of Computational Physics*, 105(2):267–348, 1993. → pages 93

[GH97] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, pages 209–216, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co. → pages 13, 14, 15, 28, 113

[GLG95] Manuel Noronha Gamito, Pedro Faria Lopes, and Mario Rui Gomes. Two-dimensional simulation of gaseous phenomena using vortex particles. In *Proceedings of the 6th Eurographics Workshop on Computer Animation and Simulation*, pages 3–15. Springer-Verlag, 1995. → pages 92

[GMMS86] James Glimm, Oliver McBryan, Ralph Menikoff, and David H. Sharp. Front tracking applied to Rayleigh–Taylor instability. *SIAM Journal on Scientific and Statistical Computing*, 7(1):230–251, 1986. → pages 11

[Gou10] Frédéric Goualard. Fast and correct SIMD algorithms for interval arithmetic. In *Proceedings of PARA '08*, Lecture Notes in Computer Science, Trondheim, 2010. Springer. 10 pages. → pages 71

[GR87] Leslie Greengard and Vladimir Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73:325–348, December 1987. → pages 100

[GSLF05] Eran Guendelman, Andrew Selle, Frank Losasso, and Ronald Fedkiw. Coupling water and smoke to thin deformable and rigid shells. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, pages 973–981, 2005. → pages 122

[GTLH01] Andre Guéziec, Gabriel Taubin, Francis Lazarus, and William Horn. Cutting and stitching: Converting sets of polygons to manifold surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 7(2):136–151, 2001. → pages 30

[HDD⁺93] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '93, pages 19–26, New York, NY, USA, 1993. ACM. → pages 14, 15

[Hes72] John L. Hess. Calculation of potential flow about arbitrary three-dimensional lifting bodies. Technical report, DTIC Document, 1972. → pages 98

[HK03] Jeong-Mo Hong and Chang-Hun Kim. Animation of bubbles in liquid. *Computer Graphics Forum (Proc. Eurographics)*, 22:253–262, 2003. → pages 111

[HK05] Jeong-Mo Hong and Chang-Hun Kim. Discontinuous fluids. In *ACM Transactions on Graphics (Proc. SIGGRAPH)*, pages 915–920, 2005. → pages 112

[HLS01] Thomas Y. Hou, John S. Lowengrub, and Michael J. Shelley. Boundary integral methods for multicomponent fluids and multiphase materials. *Journal of Computational Physics*, 169(2):302 – 362, 2001. → pages 11

[HNB⁺06] Ben Houston, Michael B. Nielsen, Christopher Batty, Ola Nilsson, and Ken Museth. Hierarchical RLE level set: A compact and versatile deformable surface representation. *ACM Transactions on Graphics*, 25(1):151–175, 2006. → pages 9

[HSKF07] Jeong-Mo Hong, Tamar Shinar, Myungjoo Kang, and Ronald Fedkiw. On boundary condition capturing for multiphase interfaces. *SIAM Journal on Scientific Computing*, 31(1-2):99–125, 2007. → pages 112

[Hul92] Jeffrey P. M. Hultquist. Constructing stream surfaces in steady 3D vector fields. In *VIS '92: Proceedings of the 3rd conference on*

Visualization '92, pages 171–178, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press. → pages 81

[HVTG08] David Harmon, Etienne Vouga, Rasmus Tamstorf, and Eitan Grinspun. Robust treatment of simultaneous collisions. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 27(3):1–4, 2008. → pages 33

[HW65] Francis H. Harlow and J. Eddie Welch. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *Physics of Fluids*, 8(12):2182–2189, 1965. → pages 10

[JCNH06] Xiangmin Jiao, Andrew Colombi, Xinlai Ni, and John C. Hart. Anisotropic mesh adaptation for evolving triangulated surfaces. In *Proc. Meshing Roundtable*, pages 173–190, September 2006. → pages 12, 15, 16, 25

[Jia06] Xiangmin Jiao. Volume and feature preservation in surface mesh optimization. In *Proc. Meshing Roundtable*, pages 62–69, 2006. → pages 15

[Jia07] Xiangmin Jiao. Face offsetting: A unified approach for explicit moving interfaces. *Journal of Computational Physics*, 220(2):612–625, 2007. → pages 12, 27, 28, 36, 113

[JP99] Doug L. James and Dinesh K. Pai. ArtDefo: accurate real time deformable objects. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '99, pages 65–72, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co. → pages 11

[Kel29] Oliver Dimon Kellogg. *Foundations of Potential Theory*. Dover Publications, 1929. → pages 98

[KFCO06] Bryan M. Klingner, Bryan E. Feldman, Nuttapong Chentanez, and James F. O'Brien. Fluid animation with dynamic meshes. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, pages 820–825, 2006. → pages 110, 115, 120

[KGJ09] Hari Krishnan, Christoph Garth, and Kenneth I. Joy. Time and streak surfaces for flow visualization in large time-varying data sets. *Proceedings of IEEE Visualization '09*, October 2009. → pages 81

[KLLR07] 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. → pages 80

[KSK09] Doyub Kim, Oh-young Song, and Hyeong-Seok Ko. Stretching and wiggling liquids. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)*, page 120, 2009. → pages 92, 108, 110

[KTJG08] Theodore Kim, Nils Thürey, Doug James, and Markus Gross. Wavelet turbulence for fluid simulation. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, pages 1–6, 2008. → pages 80

[KWT88] Michael Kass, Andrew Witkin, and Demetri Terzopoulos. Snakes: Active contour models. *International Journal of Computer Vision*, 1:321–331, 1988. 10.1007/BF00133570. → pages 12

[Lam06] Branimir Lambov. Interval arithmetic using SSE-2. In P. Hertling, C. M. Hoffmann, W. Luther, and N. Revol, editors, *Reliable Implementation of Real Number Algorithms: Theory and Practice*, volume 5045 of *Lecture Notes in Computer Science*, pages 102–113, 2006. → pages 71

[LGF04] Frank Losasso, Frederic Gibou, and Ronald Fedkiw. Simulating water and smoke with an octree data structure. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 23(3):457–462, 2004. → pages 9, 92

[LSSF06] Frank Losasso, Tamar Shinar, Andrew Selle, and Ronald Fedkiw. Multiple interacting liquids. In *ACM Transactions on Graphics (Proc. SIGGRAPH)*, pages 812–819, 2006. → pages 116

[LT03] Jacques-Olivier Lachaud and Benjamin Taton. Deformable model with adaptive mesh and automated topology changes. In M. Rioux, P. Boulanger, and G. Godin, editors, *Proc. 4th int. Conf. 3-D Digital Imaging and Modeling (3DIM'2003), Banff, Alberta, Canada*. IEEE Computer Society Press, 2003. → pages 13, 16

[LV05] Ling Li and Vasily Volkov. Cloth animation with adaptively refined meshes. In *Proceedings of the Twenty-eighth Australasian conference on Computer Science - Volume 38*, ACSC '05, pages 107–113, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc. → pages 55

[MBE+10] Marek Krzysztof Misztal, Robert Bridson, Kenny Erleben, Jabob Andreas Baerentzen, and François Anton. Optimization-based fluid simulation on unstructured meshes. In *Proceedings of the 7th Workshop on Virtual Reality Interaction and Physical Simulation, VRIPHYS*, 2010. → pages 14

[MBLD02] Mark Meyer, Alan Barr, Haeyoung Lee, and Mathieu Desbrun. Generalized barycentric coordinates on irregular polygons. *Journal of Graphics Tools*, 7(1):13–22, 2002. → pages 120

[MDSB02] Mark Meyer, Mathieu Desbrun, Peter Schroder, and Alan H. Barr. Discrete differential-geometry operators for triangulated 2-manifolds. In *VisMath*, 2002. → pages 40, 112, 121

[Mit08] Ian M. Mitchell. The flexible, extensible and efficient toolbox of level set methods. *SIAM Journal on Scientific Computing*, 35(2-3):300–329, June 2008. → pages 37

[MMS07] Viorel Mihalef, Dimitris Metaxas, and Mark Sussman. Textured liquids based on the marker level set. *Computer Graphics Forum (Proc. Eurographics)*, 26(3):457–466, 2007. → pages 9

[MMTD07] Patrick Mullen, Alexander McKenzie, Yiying Tong, and Mathieu Desbrun. A variational approach to Eulerian geometry processing. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 26(3), July 2007. → pages 10

[MT96] Tim McInerney and Demetri Terzopoulos. Deformable models in medical image analysis: a survey. *Medical Image Analysis*, 1(2):91 – 108, 1996. → pages 12

[MT00] Tim McInerney and Demetri Terzopoulos. T-snakes: Topology adaptive snakes. *Medical Image Analysis*, 4(2):73 – 91, 2000. → pages 12, 17

[Mül09] Matthias Müller. Fast and robust tracking of fluid surfaces. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 237–245, New York, NY, USA, 2009. ACM. → pages 13, 17, 80, 107, 111, 126, 131

[MUM+06] Viorel Mihalef, Betül Ünlüsü, Dimitris Metaxas, Mark Sussman, and M. Yousuff Hussaini. Physics based boiling simulation. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics Symposium on Computer animation*, pages 317–324, 2006. → pages 10

[NKJF09] Matthieu Nesme, Paul G. Kry, Lenka Jeřábková, and François Faure. Preserving topology and elasticity for embedded deformable models. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 28(3):52:1–52:9, July 2009. → pages 111

[NTW97] George C. Nakos, Peter R. Turner, and Robert M. Williams. Fraction-free algorithms for linear and polynomial equations. *SIGSAM Bulletin*, 31:11–19, September 1997. → pages 63

[NW76] William Noh and Paul Woodward. SLIC (Simple Line Interface Calculation). In A. I. van de Vooren and P. J. Zandbergen, editors, *5th International Conference on Numerical Methods in Fluid Dynamics*, volume 59 of *Lecture Notes in Physics, Berlin Springer Verlag*, pages 330–340, 1976. → pages 9

[OF03] Stanley Osher and Ronald Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*, volume 153 of *Applied Mathematical Sciences*. Springer-Verlag New York, Inc., 2003. → pages 3, 8, 9

[OIPA04] Eugenio Oñate, Sergio Rodolfo Idelsohn, Facundo Del Pin, and Romain Aubry. The particle finite element method. an overview. *International Journal of Computational Methods*, 1(2):267–307, 2004. → pages 10

[OS88] Stanley Osher and James A. Sethian. Fronts propagating with curvature dependent speed: algorithms based on Hamilton-Jacobi formulations. *Journal of Computational Physics*, 79:12–49, 1988. → pages 9

[PB07] Jean-Philippe Pons and Jean-Daniel Boissonnat. Delaunay deformable models: Topology-adaptive meshes based on the restricted Delaunay triangulation. In *IEEE Conference on Computer Vision and Pattern Recognition CVPR 2007*, 2007. → pages 14

[PCK10] Darko Pavić, Marcel Campen, and Leif Kobbelt. Hybrid booleans. *Computer Graphics Forum*, 29(1):75–87, 2010. → pages 14

[PDJ$^+$01] Dinesh K. Pai, Kees van den Doel, Doug L. James, Jochen Lang, John E. Lloyd, Joshua L. Richmond, and Som H. Yau. Scanning physical interaction behavior of 3D objects. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 87–96, New York, NY, USA, 2001. ACM. → pages 11

[PK05] Sang Il Park and Myoung Jun Kim. Vortex fluid for gaseous phenomena. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer animation*, SCA '05, pages 261–270, New York, NY, USA, 2005. ACM. → pages 92, 97

[PN01] Ken Perlin and Fabrice Neyret. Flow noise. In *Siggraph Technical Sketches and Applications*, page 187, Aug 2001. → pages 81, 84

[PN03] Blair Perot and Ramesh Nallapati. A moving unstructured staggered mesh method for the simulation of incompressible free-surface flows. *Journal of Computational Physics*, 184(1):192–214, 2003. → pages 112

[Poz00] Constantine Pozrikidis. Theoretical and computational aspects of the self-induced motion of three-dimensional vortex sheets. *Journal of Fluid Mechanics*, 425:335–366, 2000. → pages 11

[Pri91] Douglas M. Priest. Algorithms for arbitrary precision floating point arithmetic. In *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 132–145. IEEE Computer Society Press, 1991. → pages 54

[Pro97] Xavier Provot. Collision and self-collision handling in cloth model dedicated to design garment. *Graphics Interface*, pages 177–89, 1997. → pages 17, 52

[PSCN10] Jinho Park, Yeongho Seol, Frederic Cordier, and Junyong Noh. A smoke visualization model for capturing surface-like features. *Computer Graphics Forum (Proc. Eurographics)*, 29(8):2352–2362, 2010. → pages 81

[PTG12] Tobias Pfaff, Nils Thuerey, and Markus Gross. Lagrangian vortex sheets for animating fluids. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 31(4):112:1–112:8, 2012. → pages 93, 105

[RK98] William J. Rider and Douglas B. Kothe. Reconstructing volume tracking. *Journal of Computational Physics*, 141:141–112, 1998. → pages 10

[RPH04] Shaun D. Ramsey, Kristin Potter, and Charles Hansen. Ray bilinear patch intersections. *Journal of Graphics Tools*, 9(3):41–47, 2004. → pages 53

[SB08] Hagit Schechter and Robert Bridson. Evolving sub-grid turbulence for smoke animation. In *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2008. → pages 80

[SBH09] 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. → pages 110, 112, 117

[SDT08] Mark J. Stock, Werner J.A. Dahm, and Grétar Tryggvason. Impact of a vortex ring on a density interface using a regularized inviscid vortex sheet method. *Journal of Computational Physics*, 227(21):9021 – 9043, 2008. Special Issue Celebrating Tony Leonard's 70th Birthday. → pages 12, 91, 93, 94, 105

[Set99] James A. Sethian. *Level Set Methods and Fast Marching Methods*. Cambridge University Press, June 1999. → pages 9, 40

[SFK$^+$08] Andrew Selle, Ronald Fedkiw, Byungmoon Kim, Yingjie Liu, and Jarek Rossignac. An unconditionally stable MacCormack method. *SIAM Journal on Scientific Computing*, 35(2-3):350–371, 2008. → pages 52, 80

[She96] Jonathan Richard Shewchuk. Robust Adaptive Floating-Point Geometric Predicates. In *Proceedings of the Twelfth Annual Symposium on Computational Geometry*, pages 141–150. Association for Computing Machinery, May 1996. → pages 17, 54, 61

[She02] Jonathan Richard Shewchuk. What is a good linear element? interpolation, conditioning, and quality measures. In *Proc. Meshing Roundtable*, pages 115–126, 2002. → pages 15, 26, 155

[Shi07] Seungwon Shin. Computation of the curvature field in numerical simulation of multiphase flow. *Journal of Computational Physics*, 222(2):872–878, 2007. → pages 112

[Si06] Hang Si. *TetGen: A Quality Tetrahedral Mesh Generator and Three-Dimensional Delaunay Triangulator*, 2006. → pages 117

[Sim90] Karl Sims. Particle animation and rendering using data parallel computation. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '90, pages 405–413, New York, NY, USA, 1990. ACM. → pages 81

[SO09] 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. → pages 112

[SRF05] Andrew Selle, Nick Rasmussen, and Ronald Fedkiw. A vortex particle method for smoke, water and explosions. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 24(3):910–914, 2005. → pages 80, 92

[Sta99] Jos Stam. Stable fluids. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '99, pages 121–128, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co. → pages 80, 89

[Sta09] Jos Stam. Nucleus: Towards a unified dynamics solver for computer graphics. In *11th IEEE International Conference on Computer-Aided Design and Computer Graphics*, pages 1–11, 2009. → pages 53

[Sto07] Mark J. Stock. Summary of vortex methods literature. unpublished, 2007. → pages 12

[Sus03] Mark Sussman. A second order coupled level set and volume-of-fluid method for computing growth and collapse of vapor bubbles. *Journal of Computational Physics*, 187(1):110–136, 2003. → pages 10

[SZL92] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '92, pages 65–70, New York, NY, USA, 1992. ACM. → pages 15

[TB00] David Torres and Jeremiah Brackbill. The point-set method: front-tracking without connectivity. *Journal of Computational Physics*, 165:620–644, December 2000. → pages 10

[TBE+01] Grétar Tryggvason, Bernard Bunner, Asghar Esmaeeli, Damir Juric, Nebeel Al-Rawahi, Warren Tauber, Jaehoon Han, Selman Nas, and Yi-Jou Jan. A front-tracking method for the computations of multiphase flow. *Journal of Computational Physics*, 169(2):708 – 759, 2001. → pages 13, 15, 16, 112

[TKM10] Min Tang, Young J. Kim, and Dinesh Manocha. Continuous collision detection for non-rigid contact computations using local advancement. *Proceedings of International Conference on Robotics and Automation*, 2010. → pages 53

[TMT10] Min Tang, Dinesh Manocha, and Ruofeng Tong. Fast continuous collision detection using deforming non-penetration filters. In *I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 7–13, New York, NY, USA, 2010. ACM. → pages 61

[TSB+05] Joseph Teran, Eftychios Sifakis, Silvia S. Blemker, Victor Ng-Thow-Hing, Cynthia Lau, and Ronald Fedkiw. Creating and simulating skeletal muscle from the visible human data set. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):317–328, 2005. → pages 111

[TT09] Hiroshi Terashima and Grétar Tryggvason. A front-tracking/ghost-fluid method for fluid interfaces in compressible flows. *Journal of Computational Physics*, 228(11):4012–4037, 2009. → pages 111

[Tur92] Greg Turk. Re-tiling polygonal surfaces. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '92, pages 55–64, New York, NY, USA, 1992. ACM. → pages 16

[TWGT10] Nils Thürey, Chris Wojtan, Markus Gross, and Greg Turk. A Multiscale Approach to Mesh-based Surface Tension Flows. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 29(4):48:1–48:10, July 2010. → pages 112, 131

[UT92] Salih Ozen Unverdi and Grétar Tryggvason. A front-tracking method for viscous, incompressible, multi-fluid flows. *Journal of Computational Physics*, 100(1):25 – 37, 1992. → pages 11

[VB05] Julien Villard and Houman Borouchaki. Adaptive meshing for cloth animation. *Engineering with Computers*, 20:333–341, August 2005. → pages 55

[vFWTS08] Wolfram von Funck, Tino Weinkauf, Holger Theisel, and Hans-Peter Seidel. Smoke surfaces: An interactive flow visualization technique inspired by real-world flow experiments. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1396–1403, Nov/Dec 2008. → pages 81

[VRS03] Jens Vorsatz, Christian Rössl, and Hans-Peter Seidel. Dynamic remeshing and applications. In *Proceedings of the eighth ACM symposium on Solid modeling and applications*, SM '03, pages 167–175, New York, NY, USA, 2003. ACM. → pages 15

[WBOL07] Jeremy D. Wendt, William Baxter, Ipek Oguz, and Ming C. Lin. Finite volume flow simulations on arbitrary domains. *Graphical Models*, 69(1):19–32, 2007. → pages 110

[WH91] Jakub Wejchert and David Haumann. Animation aerodynamics. In *Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '91, pages 19–22, New York, NY, USA, 1991. ACM. → pages 81

[Wil08] Brent Williams. Fluid surface reconstruction from particles. Master's thesis, University of British Columbia, 2008. → pages 10, 36

[WP10] Steffen Weißmann and Ulrich Pinkall. Filament-based smoke with vortex shedding and variational reconnection. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 29:115:1–115:12, July 2010. → pages 92, 97

[WRK$^+$10] Martin Wicke, Daniel Ritchie, Bryan Klingner, Sebastian Burke, Jonathan Shewchuk, and James O'Brien. Dynamic local remeshing for elastoplastic simulation. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 29(3), 2010. → pages 55

[WT08] Chris Wojtan and Greg Turk. Fast viscoelastic behavior with thin features. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 27(3):47:1–47:8, August 2008. → pages 16, 55, 111, 112, 127

[WTGT09] Chris Wojtan, Nils Thürey, Markus Gross, and Greg Turk. Deforming meshes that split and merge. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 28(3):1–10, 2009. → pages 12, 80, 107, 111, 112, 120, 131

[WTGT10] Chris Wojtan, Nils Thürey, Markus Gross, and Greg Turk. Physics-inspired topology changes for thin fluid features. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 29:50:1–50:8, July 2010. → pages 12, 30, 80, 111, 131

[Yap04] Chee Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *CRC Handbook of Computational and Discrete Geometry*, chapter 41. CRC Press LLC, 2 edition, 2004. → pages 4, 17, 54

[YLL+09] Dong-Ming Yan, Bruno Lévy, Yang Liu, Feng Sun, and Wenping Wang. Isotropic remeshing with fast and exact computation of restricted Voronoi diagram. In *ACM/EG Symposium on Geometry Processing / Computer Graphics Forum*, 2009. → pages 16

[YT10] 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. → pages 10

[YUM86] Larry Yaeger, Craig Upson, and Robert Myers. Combining physical and visual simulation–creation of the planet jupiter for the film "2010". *SIGGRAPH Comput. Graph.*, 20(4):85–93, August 1986. → pages 92

[YWTY12] Jihun Yu, Chris Wojtan, Greg Turk, and Chee Yap. Explicit mesh surfaces for particle based fluids. *Computer Graphics Forum (Proc. Eurographics)*, 31(2), 2012. → pages 131

[ZB05] Yongning Zhu and Robert Bridson. Animating sand as a fluid. In *ACM Trans. Graph. (Proc. SIGGRAPH)*, pages 965–972, New York, NY, USA, 2005. ACM. → pages 10, 80, 82

[ZRL+08] Kun Zhou, Zhong Ren, Stephen Lin, Hujun Bao, Baining Guo, and Heung-Yeung Shum. Real-time smoke rendering using compensated ray marching. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 27:36:1–36:12, August 2008. → pages 102

[ZRLK07] Xinyu Zhang, Stephane Redon, Minkyoung Lee, and Young J. Kim. Continuous collision detection for articulated models using Taylor models and temporal culling. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 26(3):15, 2007. → pages 53

[ZYP06] Wen Zheng, Jun-Hai Yong, and Jean-Claude Paul. Simulation of bubbles. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics Symposium on Computer animation*, SCA '06, pages 325–333, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association. → pages 112

Appendix A

The root parity lemma and proof of correctness for collision detection

This appendix provides a proof of the root parity lemma, introduced in Chapter 4. This proof is mostly due to Essex Edwards, and is included in this thesis for completeness.

A.1 Outline

We describe the root parity lemma, which relates the roots of a function to the action of that function on the domain boundary. First, we show it to be true for piecewise linear functions defined on a simplicial mesh. Second, we use a piecewise linear interpolant to approximate any sufficiently well behaved C^2 function. We show that this approximation is sufficiently good that the root parity lemma is also true for these functions. The intersection-function used in collision detection falls into this class of functions, so we argue the correctness of our algorithm on top of the lemma.

A.2 The Root Parity Lemma

Suppose $\Omega \subset \mathbb{R}^n$ is a non-degenerate n-polytope.

Suppose $\vec{F} : \Omega \mapsto \mathbb{R}^n$ is continuous, has $p < \infty$ roots in Ω, has no roots on $\Gamma = \partial\Omega$, and is invertible in a ball around each root. Any such function is called *admissible*.

Suppose R is a ray from $\vec{0}$ to infinity. Call any point $x \in \Gamma$ such that $\vec{F}(x) \in R$ a *crossing point*, then the *crossing number* q is the number of crossing points. Suppose that $\vec{F}(\Gamma)$ is smooth at the image of any crossing points, that the ray is not tangent to $\vec{F}(\Gamma)$ at any these points, and that $q < \infty$.

Then $p \equiv q \bmod 2$.

A.3 Proof for Piecewise Linear Functions on a Simplex Mesh

Consider a simplicial mesh M discretizing the domain Ω. Let $\vec{F}$ be an admissible function which is linear within each simplex. The image under $\vec{F}$ of any simplex in this mesh is also a simplex. So, the image of the entire mesh is also a simplicial mesh, M', though it may be self-intersecting. The function $\vec{F}$ is uniquely defined by the position of the vertices of M', and roots of $\vec{F}$ are given by simplicies in M' that contain the origin.

We may make various simplifying assumptions about M'. Such an assumption will be without loss of generality if, for any mesh M', a perturbation exists that modifies it to satisfy these conditions without changing the parity of the number of roots or the parity of the crossing number for the ray.

Assume the origin does not lie on any facets of M'. This is possible because $\vec{F}$ is invertible around each root. So a small perturbation will push the origin from being on a facet to being in only one of the adjacent simplices; therefore not changing the number of roots.

Assume also that M' has no degenerate simplices. That is, each simplex has volume. Simulation of Simplicity [EM90] addresses almost exactly this problem. A similar argument applies here. The only additional concern is that we do not modify the number of roots. Let $\varepsilon > 0$ be the distance from the origin to the nearest facet. Since the perturbation can be infinitesimal, no vertex needs to be moved more than ε, so the number of roots does not change.

Now, consider a ray R from $\vec{0}$ to infinity satisfying the conditions of the root parity lemma. Such a ray exists because $\vec{F}(\Gamma)$ is smooth almost everywhere and does not include $\vec{0}$. Define a *hit* to be an intersection of the ray with the boundary of a simplex. If the ray intersects a facet shared by two simplices, then this is two hits. Each simplex can contribute hits.

First, consider a simplex from M with no roots in it. The image of this simplex does not contain the origin. The ray crosses its boundary either zero or two times. This simplex contributes an even number of hits.

Second, consider a simplex from M containing a root. By hypothesis, the image of this simplex contains the origin in its interior. Consequently, the ray intersects its boundary once, contributing an odd number of hits.

Summing up all the hits, only simplices with roots contribute odd parity to the sum, so the parity of the number of hits equals the parity of the number of roots. Likewise, the parity of the number of hits equals the parity of the crossing number. This is because any intersection of the ray with an interior facet contributes two hits. Only the boundary facets, coincident with Γ, contribute odd parity. So the parity of the number of roots equals the parity of the number of crossings. Consequently, the Root Parity Lemma is true for this class of functions.

A.4 Proof for C^2 Functions

Let $\vec{F}$ be an admissible C^2 function with curvature bounded everywhere. In particular, the magnitude of the directional second derivative is bounded by some constant $0 < c_t < \infty$. And let R be a ray consistent with the hypotheses of the root parity lemma. Also, suppose $\vec{F}$ has non-singular Jacobian at all the roots, which is a stronger than the local invertibility requirement givn by admissility.

Let the entire domain Ω be tessellated with a simplicial mesh M. Let each simplex have circumradius less than δ_{out}, and let each root of $\vec{F}$ be at the centroid of a regular simplex. We take the existence of such a mesh to be trivial. Let $\vec{F}_L$ be the piecewise linear interpolant of $\vec{F}$ on a the mesh M. We argue that there exists such a mesh for which $\vec{F}_L$ and $\vec{F}$ have the same number of roots, the same crossing number, and $\vec{F}$ is admissible. From this it follows that the root parity lemma is true for $\vec{F}$, and for the entire class of functions.

A.4.1 Roots

In this section, we show that if the mesh is sufficiently fine, then in any simplex, either $\vec{F}$ and $\vec{F}_L$ both have roots, or neither $\vec{F}$ or $\vec{F}_L$ have roots.

Let $\vec{x}^\star$, be an arbitrary root of $\vec{F}$, and let Σ be the simplex containing it. Since this simplex is regular, it has inradius $\delta_{\text{out}} = \kappa\delta_{\text{in}}$ with constant κ. In addition to the functions $\vec{F}$ and $\vec{F}_L$, we introduce $\vec{F}_A = J(\vec{x}-\vec{x}_i)$ which is the linear approximation to $\vec{F}$ about the root (i.e. $J = \nabla\vec{F}(\vec{x}^\star)$).

Clearly $\vec{F}_A(\vec{x}^\star) = \vec{0}$, so $\vec{F}_A$ has a root in Σ. Now, consider a point $\vec{q}$ on the surface of Σ.

$$\begin{aligned} \|\vec{F}_A(\vec{q})\|_2 &= \|J(\vec{x}-\vec{x}^\star)\| \\ &\geq \|J^{-1}\|_2^{-1}\|\vec{x}-\vec{x}^\star\| \\ &\geq \delta_{\text{in}}\|J^{-1}\|_2^{-1} \end{aligned}$$

So, the image of Σ under $\vec{F}_A$, which is also a simplex, has its surface at least $\delta_{\text{in}}\|J_i^{-1}\|_2^{-1}$ away from the origin.

By Taylor's Theorem, we have,

$$\|\vec{F}(\vec{q}) - \vec{F}_A(\vec{q})\| \leq c_1\|\vec{q}-\vec{q}_i\|^2 \leq c_1\delta_{\text{out}}^2$$

where $c_1 < \infty$ is a constant related to c_t. Similarly, by the approximation quality of linear interpolants (for proof, see e.g. [She02]), we have

$$\|\vec{F}(\vec{q}) - \vec{F}_L(\vec{q})\| \leq c_2\delta_{\text{out}}^2$$

where $c_2 < \infty$ is another constant related to c_t. These can be combined to get the relationship,

$$\|\vec{F}_A(\vec{q}) - \vec{F}_L(\vec{q})\| \leq c_3\delta_{\text{out}}^2.$$

The origin is at least $\delta_{\text{in}}\|J_i^{-1}\|_2^{-1}$ away from the surface of $\vec{F}_A(\Sigma)$, and $\vec{F}_A(\Sigma)$ is no more than $c_3\delta_{\text{out}}^2$ away from $\vec{F}_L(\Sigma)$. Since $\vec{F}_L(\Sigma)$ is also a simplex, it follows that if

$$\delta_{\text{in}}\|J_i^{-1}\|_2^{-1} > c_3{\delta_{\text{out}}}^2 \Leftrightarrow 1/(\kappa c_3\|J^{-1}\|_2) > \delta_{\text{out}},$$

then $\vec{F}_L(\Sigma)$ will also contain the origin, and $\vec{F}_L$ will also have a root in Σ. Since $\kappa c_3 \|J^{-1}\|_2$ is some constant bounded away from zero, we can choose such a δ_{out}.

It follows that the image Σ under $\vec{F}_L$ will contain the origin if $\delta_{\text{in}} \|J_i^{-1}\|_2^{-1} > c_3 {\delta_{\text{out}}}^2$. For the regular simplex used in this case, $\delta_{\text{out}} = \kappa \delta_{\text{in}}$ with some constant $\kappa < \infty$. So, choose a sufficiently small simplex such that $1/(\|J_i^{-1}\|_2^{-1} c_3 \kappa) > \delta_{\text{out}}$. Consequently, both $\vec{F}$ and $\vec{F}_L$ have a single root in Σ, and $\vec{F}_A$ is locally invertible there. Looking at the whole set of roots, since there are finitely many of them, this constraint on δ_{out} can be summarized as $\delta_{\text{out}} < \delta_{\text{out}}^{\text{roots}}$, where $0 < \delta_{\text{out}}^{\text{roots}}$ is bounded away from zero.

Now we show that the other simplices of the mesh, $\vec{F}_L$ has no roots. Continuing with the point $\vec{q}$ on the surface of the simplex around a root, we find that

$$\begin{aligned}
\|\vec{F}(\vec{q})\| &\geq \|\vec{F}_A(\vec{q})\| - c_1 \|\vec{q} - \vec{x}\|^2 \\
&\geq \|J(\vec{q} - \vec{x})\| - c_1 {\delta_{\text{out}}}^2 \\
&\geq \|J^{-1}\|^{-1} \|\vec{q} - \vec{x}\| - c_1 {\delta_{\text{out}}}^2 \\
&\geq \|J^{-1}\|^{-1} \delta_{\text{in}} - c_1 {\delta_{\text{out}}}^2 \\
&\geq \|J^{-1}\|^{-1} \delta_{\text{in}} - c_1 \kappa^2 {\delta_{\text{in}}}^2 \\
&\geq \|J^{-1}\|^{-1} \kappa^{-1} \delta_{\text{out}} - c_1 {\delta_{\text{out}}}^2 \\
&\geq \alpha \delta_{\text{out}} - c_1 {\delta_{\text{out}}}^2
\end{aligned}$$

where $\alpha > 0$ is the minimum value over all roots of $\|J^{-1}\|^{-1} \kappa^{-1}$.

Let S be the union of the simplicies which contain roots of $\vec{F}$, as have already been addressed. Then, $\Omega_0 = \overline{\Omega \setminus S}$ is the remainder of the domain, including the boundaries. In Ω_0, $\|\vec{F}\| > F_{\text{min}} > 0$. For sufficiently small simplices, the minimum value of $\|\vec{F}\|$ will occur on the surface of one of the simplices surrounding a root. For simplicity, assume that this is the case. So, $F_{\text{min}} > \alpha \delta_{\text{out}} - c_1 {\delta_{\text{out}}}^2$ everywhere outside the simplicies surrounding the roots. Then at all points $\vec{x}$ outside the root-simplices $\|\vec{F}_L(\vec{x}) - \vec{F}(\vec{(x)})\| \leq c_2\delta_{\text{out}}^2$, and so

$$\begin{aligned}
\|\vec{F}_L(\vec{x})\| &\geq \|\vec{F}(\vec{x})\| - c_2\delta_{\text{out}}^2 \\
&\geq F_{\min} - c_2\delta_{\text{out}}^2 \\
&> \alpha\delta_{\text{out}} - c_1{\delta_{\text{out}}}^2 - c_2\delta_{\text{out}}^2 \\
&> \alpha\delta_{\text{out}} - {\delta_{\text{out}}}^2(c_1 + c_2)
\end{aligned}$$

For sufficiently small δ_{out}, we have $\|\vec{F}_L(\vec{x})\| > 0$ because $\alpha > 0$. So, outside of the simplices surrounding the roots, $\vec{F}_L$ has no roots.

It follows that $\vec{F}(x)$ and $\vec{F}_L(x)$ have the same number of roots.

A.4.2 Crossing Points

To show that $\vec{F}_L$ also has the same crossing number as $\vec{F}(\Gamma)$, we construct two new functions $H : \Gamma \mapsto \mathbb{R}$ and $\vec{G} : \Gamma \mapsto \mathbb{R}^{n-1}$. Without loss of generality, by a simple rotation, let the ray be the positive x_1 axis. Then, let $H(x) = \vec{F}_1(x)$ be the first component of $\vec{F}(x)$. It measures the distance along the ray of the closest point to $\vec{F}(x)$. Second, let $\vec{G}(x)$ be components 2 through n of $\vec{F}(x)$. So, it measures a vector-distance from $\vec{F}(x)$ to the closest point on the ray. Consequently, a point $\vec{x}$ is a crossing point of R and $\vec{F}(\Gamma)$ iff $H(x) > 0$ and $\vec{G}(x) = \vec{0}$.

Now, consider the functions $\bar{\vec{G}}$ and $\bar{H}$ defined as above, but using $\vec{F}_L$ instead of $\vec{F}$. As before, $\vec{x}$ is a crossing point of R and $\vec{F}_L(\Gamma)$ iff $\bar{H}(x) > 0$ and $\bar{\vec{G}}(x) = \vec{0}$. Notice also, $\bar{H}$ and $\bar{\vec{G}}$ are the linear interpolants of H and $\vec{G}$ on the boundary elements of the mesh, which form an $n-1$ dimensional simplex mesh embedded in n-space. So we can use similar arguments as above to establish an exact correspondence between the crossing points of R and $\vec{F}(\Gamma)$ and the crossing points of R and $\vec{F}_L(\Gamma)$ by matching the roots of $\vec{G}$ and $\bar{\vec{G}}$ and the signs of H at those roots.

$\vec{G}$ may not be locally invertible at all of its roots, so we take a different approach than above for $\vec{F}$. Add all of the crossing points of R and $\vec{F}(\Gamma)$ as vertices to the mesh, by hypothesis there are a finite number of them. This does not contradict the earlier construction of a single simplex around each root of $\vec{F}$, because $\vec{F}$ has no roots on Γ. With these vertices in the mesh, $H(\vec{x}) > 0$ and $\vec{G})(\vec{x}) = \vec{0}$ implies

$\bar{\vec{G}}(\vec{x}) = \vec{0}$. Let any simplex containing a root of $\vec{G}$ be small enough that it contains only one. By construction, it will be at a vertex, the *root-vertex*. All the other vertices of this simplex form an $(n-2)$-face, the *far-face*. By using anisotropic simplices around the roots, the far-face can always be distance $\delta > 0$ from the root, but fit in an arbitrarily small ball of radius r. Consequently, $\|\vec{G}\|_\infty > \varepsilon > 0$ on the far-face. A described earlier, a sufficiently fine mesh will have no roots on the far-face and consequently no roots anywhere in the simplex, except at the root-vertex.

Let ε be half the minimum magnitude of H at any root of $\vec{G}$. When $-\varepsilon < H < \varepsilon$, by definition of ε, $\|\vec{G}\|_\infty > \delta > 0$. So, in a sufficiently fine mesh, $\bar{\vec{G}} \neq \vec{0}$. This follows from the same bound used above to analyze $\vec{F}_L$. When $H < -\varepsilon$, in a sufficiently fine mesh we get $\bar{H} < 0$. Finally, when $H > \varepsilon$, a sufficiently fine mesh will have $\bar{H} > 0$ and, outside of the simplices constructed above, $\bar{\vec{G}} \neq \vec{0}$. Combined with the results above, we conclude that $\vec{F}$ and $\vec{F}_L$ have the same crossing points.

Combined with the earlier result, we have that $\vec{F}$ and $\vec{F}_L$ have the same number of roots, and exactly the same crossing points. So we have shown that the root parity lemma holds for the entire class of C^2 functions with bounded second derivative and non-singular jacobian at each root.

A.5 Proof of the Collision Algorithm

The function that the algorithm uses is C^2 with bounded curvature, but isn't always admissible and with invertible Jacboian at the roots. We argue here that the algorithm still does the right thing for collision detection, even with inadmissible functions.

When the input geometry to the collision detection algorithm produces an admissible function $\vec{F}$, the correctness of the algorithm follows trivially from the root parity lemma. The construction of a ray which is consistent with the lemma is done by generating random rays and rejecting ones that intersect at non-smooth parts of $\vec{F}(\Gamma)$. This rejection is extremely rare.

If $\vec{F}(u) = 0$ on Γ, the algorithm identifies this as a collision. This may be a false positive for a 'significant' collision, but it is never a false positive that a collision between these elements has occurred.

Otherwise, if $\vec{F}$ has an infinite number of roots, then because of the multi-affine form of $\vec{F}$, it has a root on Γ, and this case is detected as above.

Otherwise, if $\nabla\vec{F}$ is not invertible at a root, then there exists a small perturbation to $\vec{F}$ such it is not singular at any roots and is unchanged on Γ. This perturbation need not affect the initial and final configuration of the mesh, it only causes an infinitesimal adjustment to their intermediate trajectories. Thus, it will not change whether or not a significant collision has occurred, and the algorithm returns the correct result.

The root parity lemma also has conditions on the ray that the algorithm must meet. The only non-smooth regions of Γ are the edges. The algorithm traces a new ray when an intersection with an edge is detected. If the crossing number is infinite, then the ray must hit the edges, which is detected as above. The requirement that the ray not be tangent is handled by the ray-patch parity algorithm, which returns the correct (even) parity in that case.

We note that while this proof contains several cases involving perturbing $\vec{F}$ or the ray, the algorithm does not need to do this. This was only a tool for use in the proof.