Tracking and Texturing Liquid Surfaces

Adam Wade Bargteil

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-196
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-196.html

December 31, 2006

Copyright © 2006, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Tracking and Texturing Liquid Surfaces

by

Adam Wade Bargteil

B.S. (University of Maryland, College Park) 2000

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor James F. O'Brien, Chair
Professor Jonathan R. Shewchuk
Professor John A. Strain

Fall 2006

The dissertation of Adam Wade Bargteil is approved:

Chair Date

Date

Date

University of California, Berkeley

Fall 2006

Tracking and Texturing Liquid Surfaces

Copyright 2006

by

Adam Wade Bargteil

Abstract

Tracking and Texturing Liquid Surfaces

by

Adam Wade Bargteil

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor James F. O'Brien, Chair

This thesis addresses the problems of tracking and texturing liquid surfaces in computer graphics fluid simulations. The proposed surface tracking method, known as semi-Lagrangian contouring, takes the unusual approach of representing the liquid surface explicitly with a closed, manifold triangle mesh. However, rather than attempting to track the triangle mesh through time, a new triangle mesh is built at each timestep by contouring the zero-set of an advected signed-distance function. Thus, while we represent the surface explicitly, we update the surface through time using an implicit representation. One of the primary advantages of this formulation is that it enables tracking of surface characteristics, such as color or texture coordinates, at negligible additional cost. These advected surface characteristics can then be used in a variety of ways to generate time-coherent textures on the liquid surfaces. After considering a variety of simple texturing techniques, I propose an example-based texture synthesis method designed specifically for liquid animations. This example-based texture synthesis method first advects color values on the surface and then uses an optimization process to force the surface texture to more closely match a user-

input example texture. This approach creates textures which resemble the example texture even in the precense of complicated topological changes and significant surface stretching/compression. I include a variety of examples demonstrating that these methods can be effectively used as part of a fluid simulation system to animate and texture complex and interesting liquid behaviors.

Professor James F. O'Brien
Dissertation Committee Chair

To my fiancé, Laura,

for your companionship during the California years.

Contents

List of Figures **iv**

1 Introduction **1**

2 Previous Work **4**
2.1 Previous Surface Tracking Methods 4
2.1.1 Level-Set Methods 5
2.1.2 Particle-Based Methods 6
2.1.3 Particle Level-Set Methods 7
2.1.4 Semi-Lagrangian Contouring 9
2.2 Implicit Representations 9
2.3 Contouring 10
2.4 Semi-Lagrangian Methods 11
2.5 Texturing Fluids 13
2.6 Example-Based Texture Synthesis 14
2.7 Mesh Parameterization 15

3 Semi-Lagrangian Contouring **17**
3.1 Method Overview 19
3.2 Hybrid Representation 20
3.2.1 Explicit Representation 20
3.2.2 Implicit Representation 20
3.3 The Distance Tree 24
3.3.1 Approximating the Signed-Distance Function 25
3.3.2 General Splitting Criterion 25
3.3.3 Building a Distance Tree to Resolve ψ 26
3.3.4 Building a Distance Tree from a Triangle Mesh 27
3.4 Contouring 28
3.5 Redistancing 30
3.6 Tracking Surface Properties 31

4 Texturing Liquid Surfaces **33**
4.1 Advected Colors 34
4.2 Advected Parameterizations 35
4.3 Reaction-Diffusion Texture Synthesis for Liquids 36

4.4 Example-Based Texture Synthesis for Liquids 37
4.4.1 Surface Tracking 37
4.4.2 Surface Parameterization 38
4.4.3 Texture Synthesis 47

5 Results and Discussion **52**
5.1 Surface Tracking 52
5.2 Surface Texturing 59

6 Conclusions **67**

List of Figures

3.1 Method overview 19
3.2 Handling topological changes 21
3.3 Interpolation gets the wrong answer 24
3.4 A two-dimensional distance tree 24

4.1 The advected parametric directions 38
4.2 Patches used in optimization 39
4.3 A surface patch using the popular heuristic of Maillot *et al.* 40
4.4 A surface patch using the bounded-distortion method of Sorkine *et al.* . . . 41
4.5 A surface patch after optimization 42
4.6 Incremental patch construction 43
4.7 Planar view of one step of texture optimization 48
4.8 The parameter, w, allows us to trade off temporal coherence and matching the input texture 48
4.9 Texture optimization progression 51

5.1 Multicolored balls falling in a tank 53
5.2 Merging balls 54
5.3 The "Enright" test 54
5.4 Error plot for the last frame of the "Enright" test 54
5.5 The "Enright" test continued 55
5.6 Viscoelastic fluid sliding off a shelf with a checkerboard texture 57
5.7 Viscoelastic fluid with tree-bark and reaction-diffusion textures 57
5.8 Two streams of oscillating fluid merging and separating 58
5.9 Different approaches to texturing liquid animations 60
5.10 Textured splash 61
5.11 Textured melting bunny 62
5.12 Textured merging balls of goop 63
5.13 Textured viscous flow 64
5.14 Several examples with multiple textures 65

Acknowledgments

I am deeply indebted to my skillful advisor, James O'Brien, for guiding me through graduate school, spending hours reading early (and poorly written) drafts of papers, teaching me the importance of presentation and attention to detail, and telling me straight when I screwed up. I am also grateful to the other members of my committee, Jonathan Shewchuk and John Strain, whose exceptional lectures and writings have inspired my own work.

I would also like to thank the other members of the Berkeley graphics community, faculty and students, past and present, for their insightful criticism and helpful comments. I am especially grateful to the other students in the group who were not only the best of colleagues, but also great friends. I will always remember fondly the video games we've played and the pints we've shared. In particular, I would like to thank Tolga Goktekin, my longtime co-author and good friend, for his contributions to the work described in this thesis and Funshing Sin for his invaluable contributions to the texture synthesis project. I would also like to thank my friends and colleagues at PDI/DreamWorks, for teaching me about the "real world" of computer graphics.

Finally, I would like to thank my family and friends, for their love and support and the welcome distractions from paper deadlines. I owe my greatest thanks to my fiancé, Laura, whose love and companionship have made every day in California a good one.

Chapter 1

Introduction

This thesis considers two fundamental problems in physically based computer animation of liquids: tracking the liquid's free surface and texturing this free surface. Surface tracking is an essential component of any liquid simulator—the location of the free surface must be known in order to apply boundary conditions. Thus, the surface tracking method directly affects the simulation and inaccuracies in the surface location lead to inaccuracies in the resulting simulation. While this point alone is a strong argument for the importance of surface tracking in liquid simulations, in the context of computer graphics there is an even more significant argument—the surface is what we actually see. Regardless of how we render the results of our simulation, be it with a clear water shader or as textured matte surfaces, we always display the surface. Thus, even if we have an ideal method for solving the Navier-Stokes equations, a poor surface tracking method will cause our final results, computer animations, to be unimpressive.

In addition to being a very important problem, surface tracking is also an extremely difficult problem. Liquid surfaces are characterized by frequent and complicated topological changes as well as significant stretching/compression of the surface. These characteristics

make many methods, which are quite adequate for tracking a variety of deforming surfaces, untenable in the context of liquid surfaces. For example, methods which seek to maintain an explicit surface representation incur significant complexity performing the *mesh surgery* necessary to deal with the frequent topological changes of liquid surfaces. Additionally, methods which sample the surface with points will have to address resampling the surface as it stretches and compresses. While the field of computational fluid mechanics has developed many methods and strong theoretical frameworks for numerically solving and analyzing the equations of fluid motion, the problem of tracking the free surface has received surprisingly little attention. Additionally, many surface tracking methods which are quite acceptable for a variety of engineering applications produce visual artifacts, such as flickering, which make them unusable in computer graphics, where the goal is often to fool the audience into believing that what they see is real; visual artifacts, such as flickering or significant volume loss, can destroy this illusion of realism.

This thesis proposes a solution to the surface tracking problem for use in computer graphics. Our method explicitly represents the surface as a closed, manifold triangle mesh. However, rather than attempting to advect this mesh forward with the flow, we update the surface in time with an implicit representation: an advected signed-distance function, ψ, whose zero set defines the surface. A new polygonal surface is generated by *contouring* or extracting the zero set of ψ. The value of ψ at a point $\boldsymbol{x}$, at current time t, is obtained by first tracing backward through the flow field to find the previous location $\boldsymbol{x'}$ at time $t - \Delta t$, and then returning the signed distance of $\boldsymbol{x'}$ from the previous surface. Using adaptive octree data structures, we can efficiently and reliably construct the new surface and corresponding signed-distance function. One of the primary advantages of this formulation is that it enables tracking of surface characteristics, such as color or texture coordinates,

at negligible additional cost. These surface characteristics can then be used in a variety of ways to texture the liquid surface, which is the other problem considered in this thesis.

Surface texturing is an essential computer graphics tool, which gives artists additional control over their results by allowing them to stylize surfaces or add detail to low-resolution simulations. For example, an artist could use texturing techniques to add the appearance of foam to a wave, bubbles to beer, or fat globules to soup. Texturing liquid surfaces is difficult for many of the same reasons as tracking liquid surfaces. Complex and frequent topological changes lead to discontinuities in advected parameterizations, surface distortions can lead to loss of detail or aliasing in noise-based procedural textures, and surface re-sampling issues can lead to blurring or aliasing of textures. Unlike B-spline patches, liquid surfaces lack any inherent parameterization and their complex motion makes it extremely difficult to build a temporally coherent parameterization. After considering a variety of simple texturing techniques, including advected colors, advected parameterizations, and reaction-diffusion texture synthesis, I propose an example-based texture synthesis method designed specifically for animations of liquids. This example-based synthesis method begins by advecting color values between frames using the mapping provided by the semi-Lagrangian contouring method used to track the surface. For every frame an optimization procedure attempts to force these distorted color values to more closely match the input example texture. This approach creates textures which resemble the example texture even in the precense of complicated topological changes and significant surface stretching/compression. I include a variety examples demonstrating that these methods can be effectively used as part of a fluid simulation system to animate and texture complex and interesting fluid behaviors.

Chapter 2

Previous Work

Our work pulls together solutions to a number of well-studied problems to arrive at methods for tracking and texturing liquid surfaces. In this section I will first discuss other surface tracking methods and then discuss related work and the mathematical foundations for several of the individual components of our surface tracking method. Next, I will discuss previous fluid texturing approaches and conclude by discussing previous work which developed the components of our example-based texture synthesis method.

2.1 Previous Surface Tracking Methods

Because surface tracking arises in a variety of contexts, the topic has received a significant amount of attention. Even in the limited context of fluid animation, there has been a great deal of excellent work on simulating fluids with free surfaces, including Foster and Metaxas [1996], Foster and Fedkiw [2001], Enright *et al.* [2002b], Carlson *et al.* [2002; 2004], Losasso *et al.* [2004], Goktekin *et al.* [2004], Hong and Kim [2005], Wang *et al.* [2005], Guendelman *et al.* [2005], and Zhu and Bridson [2005]. The methods available for tracking free surfaces of liquids can be roughly sorted into four categories: level-set methods,

particle-based methods, particle level-set methods, and semi-Lagrangian contouring.

2.1.1 Level-Set Methods

Many of the most successful solutions to the surface tracking problem are based on level-set methods, which were originally introduced by Osher and Sethian [1988]. A complete review of level-set methods is beyond the scope of this thesis, and I recommend the excellent surveys by Sethian [1999] and Osher and Fedkiw [2003]. Level-set methods represent a surface as the zero set of a scalar function which is updated over time by solving a partial differential equation, known as the *level-set equation*. This equation relates change of the scalar function to an underlying velocity field. By using this implicit representation, level-set methods avoid dealing with complex topological changes. However, the scalar function is defined and maintained in the embedding three-dimensional space, rather than just on the two-dimensional surface. In practice, scalar function values need only be accurately maintained very near the surface, resulting in a cost that is roughly linear in the complexity of the surface. One difficulty with level-set methods is that they generally require very high-order conservation-law solvers, though fast semi-Lagrangian methods have been shown to work in some cases [Strain, 1999b; Enright et al., 2005]. The most significant drawback to using level-set methods to track liquid surfaces is their tendency to lose volume in underresolved, high-curvature regions. See Enright *et al.* [2002a] for an excellent discussion of the reasons for this volume loss.

Bærentzen and Christensen [2002] built a sculpting system using a level-set surface representation which could be manipulated by a user with a variety of sculpting tools. Like us, they used adaptive grid structures to store the scalar field. However, they used a two-level structure rather than a full octree. They also used semi-Lagrangian methods to

update their level-set function. However, when evaluating the distance function after the semi-Lagrangian path tracing, they interpolated distance values stored on a regular grid, while our explicit surface representation allows us to compute exact distances near the surface.

Sussman and Puckett [2000] coupled volume-of-fluid and level-set methods to model droplet dynamics in ink-jet devices. Volume-of-fluid [Hirt and Nichols, 1981] techniques represent the surface by storing, in each voxel, a *volume fraction*—the proportion of the voxel filled with liquid. Any cell whose fraction is not one or zero contains surface. Unfortunately, this representation does not admit accurate curvature estimates, which are essential to surface tension computations. However, accurate curvature estimates are easily computed from level-set representations. Thus, the authors combined volume-of-fluid and level-set representations to model surface tension in ink droplets. Some volume-of-fluid methods build an explicit surface representation from the volume fractions stored in each voxel. The key difference between our method and volume-of-fluid methods is that we never compute volume fractions. Instead, our explicit representation is generated by contouring an advected signed-distance function.

2.1.2 Particle-Based Methods

A number of researchers [Terzopoulos et al., 1989; Desbrun and Gascuel, 1995; Foster and Metaxas, 1996; Desbrun and Cani, 1996; Cani and Desbrun, 1997; Stora et al., 1999; Müller et al., 2003; Premože et al., 2003; Müller et al., 2004; Zhu and Bridson, 2005; Pauly et al., 2005] have used particles to track surfaces. In many of these methods, the simulation elements are particles, which are already being tracked throughout the volume of the deforming liquid or solid. The surface can then be implicitly defined as the boundary

between where the particles are and where they aren't. The particles can be visualized directly, or can be used to define an implicit representation using metaballs [Blinn, 1982] or moving least-squares methods [Kolluri, 2005]. The moving least-squares approach was successfully used by Zhu and Bridson [2005] to construct liquid and sand surfaces from marker particles placed throughout the fluid volume. Such approaches have the significant advantage that the surfaces at each frame are independent and can be constructed in parallel as a post-processing step. Premože *et al.* [2003] took a different approach and used particle positions and velocities to guide a level-set solution. Mueller *et al.* [2004] and Pauly *et al.* [2005] used special particles, called *surfels* [Szeliski and Tonnesen, 1992], to represent the surface. Surfels store a surface normal as well as position and there are generally many more surfels than simulation particles. The principal drawback of these methods is that generating high-quality time-coherent surfaces can be difficult: directly visualizing the particles is insufficient for high-quality animations, methods which convert the particles to some other representation on a per-frame basis often lack temporal coherence, and methods which must run sequentially through the frames or run during the simulation are often quite costly. Additional difficulties arise when trying to ensure a good sampling of the surface.

2.1.3 Particle Level-Set Methods

To address the volume loss of level-set methods, Enright and his colleagues [Enright et al., 2002a; Enright et al., 2002b; Enright et al., 2005] built on the work of Foster and Fedkiw [2001] to develop particle level-set methods. These methods track the characteristics of the fluid flow with Lagrangian particles, which are then used to fix the level-set solution, essentially increasing the effective resolution of the method. Recently, these methods have been extended to work with octrees [Enright et al., 2005; Losasso et al., 2004], allowing for

very high-resolution surface tracking. These methods represent the current state of the art on tracking liquid surfaces for animation, but do have some drawbacks. In particular, the published particle correction rules choose a single particle to provide the signed-distance value. Since there is no guarantee that the same particle will be chosen at subsequent timesteps, the method is extremely susceptible to high-frequency temporally incoherent perturbations of the surface. The artifacts are most noticeable when the surface thins out below the grid resolution and particles happen to be near some of the sample points, but not others. Also, the method has a large number of parameters and rules, such as the number of particles per cell and the reseeding strategy, which need to be decided, often in an application-specific way. Finally, the method tends to produce very smooth surfaces with very little detail, which is desirable in some, but not all, applications. Despite these drawbacks, the particle level-set methods have been very successful and represent a significant step forward in the area of surface tracking for liquid simulations.

More recently Hieber and Koumoutsakos [2005] introduced a Lagrangian particle level-set method which overcomes many of these drawbacks. Instead of using a hybrid representation, they represent the level-set function solely with particles (though their resampling strategy does use a Cartesian grid). Thus, there is no correction step and all queries of the level-set function and its derivatives are handled with suitable mollification kernels, alleviating the flickering problems present in the work of Enright and his colleagues [Enright et al., 2002a; Enright et al., 2002b; Enright et al., 2005]. Moreover, they present a general resampling (reseeding) strategy which removes the guesswork from previous approaches and, additionally, regularizes the level-set values stored at the particle locations, addressing any distortions which may have developed as the particles were advected through the flow field.

2.1.4 Semi-Lagrangian Contouring

Recently, Strain [1999b; 1999c; 1999a; 2000; 2001] has written a series of articles building a theoretical framework culminating in the formulation of surface tracking as a contouring problem. He demonstrated his semi-Lagrangian contouring method on a variety of two-dimensional examples. Our method is based on the method presented by Strain [2001], but with variations and extensions to deal with problems that arise in three-dimensional computer animation. While our method bears a number of similarities to level-set methods and takes advantage of many techniques developed for those methods, we are not directly solving the level-set equation. By formulating surface tracking as a contouring problem, we avoid many of the issues that complicate level-set methods. In particular, we do not have the same volume loss issues which prompted the particle level-set methods: while we do not explicitly conserve volume, our semi-Lagrangian path tracing tends to conserve volume in the same way as the Lagrangian particles in the particle level-set method.

2.2 Implicit Representations

The octree structure we use to build and index the polygonal mesh is quite similar to adaptively sampled distance fields [Frisken et al., 2000]. These structures adaptively sample distance fields according to local detail and store samples in a spatial hierarchy. The key difference between adaptively sampled distance fields and our surface representation is that we store a polygon mesh in addition to distance samples. This polygon mesh is used for exact evaluation of the distance function near the surface. Additionally, our splitting criterion is different from that presented by Frisken *et al.* [2000].

An alternative structure for storing narrow-band level-set functions is the dynamic

tubular grid of Nielsen and Museth [2006]. This structure can be combined with run-length encoding schemes [Houston et al., 2006] providing extremely compact, high-resolution representations of level-set functions. While the asymptotic times for their structure match ours, they are able to exploit cache coherence to provide extremely fast run times for most level-set operations. Integrating the methods presented here with this data structure is a promising area for future work.

2.3 Contouring

Contouring the advected signed-distance function, ψ, is a fundamental step in our surface tracking method. The contouring problem has been well studied in computer graphics and a number of approaches have been suggested. The oldest and most widely used is marching cubes, which was first presented by Wyvill *et al.* [1986], and later named and popularized by Lorensen and Cline [1987]. Marching cubes suffers from a tendency to create ill-shaped triangles. This problem is fixed to some degree by dual contouring [Ju et al., 2002], which also provides adaptive contouring and an elegant means of preserving sharp boundaries. Dual contouring depends on normal estimates at edge crossings and is very sensitive to inaccuracies in these normal estimates. Unfortunately, in our method we do not have accurate normal information until after the contouring step, when normals can be computed on the triangle mesh. More recently, Boissonnat and Oudot [2003] presented a contouring technique which uses Delaunay triangulation methods to generate provably good triangulations. However, this method appears to be prohibitively expensive for something which must run at every timestep. Yet another alternative is marching triangles [Hilton et al., 1996], which takes a surface-based rather than volume-based approach to contouring. Marching triangles requires significantly less computation time, produces fewer triangles,

and creates higher-quality triangles than marching cubes. Unfortunately, marching triangles is not guaranteed to produce closed, manifold meshes in the presence of sharp or thin features.

2.4 Semi-Lagrangian Methods

Semi-Lagrangian methods have been widely used in computer graphics since they were introduced by Stam [1999] to solve the nonlinear advection term of the Navier-Stokes equations. These methods provide the foundation for our surface tracking method. Consequently, I briefly discuss the mathematical foundation of semi-Lagrangian methods. The discussion follows that of Strain [1999b].

Consider the simplest linear hyperbolic PDE

$$\phi_t + \boldsymbol{v}(\boldsymbol{x}, t) \cdot \nabla \phi = 0, \tag{2.1}$$

where ϕ is a scalar field and $\boldsymbol{v}(\boldsymbol{x}, t)$ is a velocity function. Equation (2.1) passively advects ϕ through the velocity field $\boldsymbol{v}$. Semi-Lagrangian methods are based on the observation that Equation (2.1) propagates ϕ values along characteristic curves $\boldsymbol{x} = \boldsymbol{s}(t)$ defined by

$$\dot{\boldsymbol{s}}(t) = \boldsymbol{v}(\boldsymbol{s}(t), t), \qquad \boldsymbol{s}(0) = \boldsymbol{x_0}. \tag{2.2}$$

Thus we can find ϕ values at any time t by finding the characteristic curve passing through $(\boldsymbol{x}, t)$, following it backward to some previous point $(\boldsymbol{x_0}, t_0)$ where the value of ϕ is known, and setting $\phi(\boldsymbol{x}, t) = \phi(\boldsymbol{x_0}, t_0)$. This observation forms the basis of the *backward characteristic* or *CIR* scheme developed by Courant, Isaacson and Rees [1952], which is the simplest semi-Lagrangian scheme. Given ϕ at time t_n, CIR approximates $\phi(\boldsymbol{x}, t_{n+1})$ at any point $\boldsymbol{x}$ at time $t_{n+1} = t_n + \Delta t$ by evaluating the previous speed $\boldsymbol{v}(\boldsymbol{x}, t_n)$, approximating the

backward characteristic through $\boldsymbol{x}$ by a straight line

$$\boldsymbol{s}(t) \approx \boldsymbol{x} - (t_{n+1} - t)\boldsymbol{v}(\boldsymbol{x}, t_n), \tag{2.3}$$

and interpolating ϕ at time t_n to the point

$$\boldsymbol{s}(t_n) \approx \boldsymbol{x} - (\Delta t)\boldsymbol{v}(\boldsymbol{x}, t_n). \tag{2.4}$$

Then $\phi(\boldsymbol{x}, t_{n+1})$ is set equal to the interpolated value, $\phi(\boldsymbol{s}(t_n), t_n)$.

For linear PDEs, such as Equation (2.1), the Lax-Richtmyer equivalence theorem [LeVeque, 1990] guarantees that CIR will converge to the exact solution as $\Delta t, \Delta x \to 0$ if it is stable and consistent.

The stability properties of the CIR scheme are excellent. Each new value $\phi(\boldsymbol{x}, t_{n+1})$ is a single interpolated value of ϕ at time t_n, so unconditional stability is guaranteed in any norm where the interpolation does not increase norms. For example, CIR with linear interpolation is unconditionally stable in the 2-norm. In general, semi-Lagrangian schemes satisfy the CFL condition by shifting the stencil, rather than restricting the timestep. Thus information propagates over long distances in one timestep.

Consistency (loosely speaking, the local accuracy of the method), however, is conditional. The global error of CIR is

$$O\left(\frac{(\Delta x)^2}{\Delta t}\right) + O(\Delta t), \tag{2.5}$$

due to the $O((\Delta x)^2)$ error in linear interpolation accumulated over $O(1/(\Delta t))$ timesteps, plus the $O(\Delta t)$ error due to freezing F and approximating the characteristics by straight lines. Thus CIR is consistent to $O(\Delta t)$ if a condition $\Delta t \geq O(\Delta x)$ is satisfied, contrary to the usual hyperbolic condition $\Delta t \leq C\Delta x$. This condition is extremely convenient, because $\Delta t = O(\Delta x)$ balances time and space resolution in this first-order accurate scheme.

For nonlinear PDEs, CIR still converges when the solution is smooth. But non-smooth shock solutions of conservation laws move at the wrong speed because CIR is not in conservative form. Since level-set solutions have no shocks, CIR is a natural scheme for moving interfaces.

2.5 Texturing Fluids

Soon after the introduction of fluid simulation techniques to computer graphics, researchers began experimenting with texturing these simulations. The simplest approach, demonstrated by Witting [1999] advects texture coordinates through the flow field and uses these texture coordinates to lookup color in a texture map. Unfortunately, over time, the texture becomes progressively more distorted. To address this distortion, Stam [1999] advects three separate layers of texture coordinates, each of which is periodically reset. The final texture map is then a superposition of these three texture maps. Neyret [2003] built on this approach and also advects several layers of textures. Additionally, he computes and advects the local accumulated deformation for each texture layer. Using this deformation measure, he combines the various texture layers to arrive at a final texture, which is well adapted to the local deformation. When using procedural noise-based textures, he combines the layers in frequency space to avoid ghosting effects and contrast fading.

While these techniques work relatively well for advecting textures through general fluid simulations, they are not directly applicable in the case of free-surface liquid simulation. In this case, we wish to texture the liquid surface rather than the fluid volume. To address the particular context of liquids, Rasmussen *et al.* [2004] describes a method that advects texture particles, initialized near the free surface, through the fluid flow field. During rendering, when a ray intersects the surface the texture coordinates from the nearest 64

particles are interpolated to provide a texture coordinate for the surface point being shaded. They have used their technique in production at Industrial Light and Magic, including a shot in *Terminator 3*. In a similar approach, Wiebe and Houston [2004] and Houston *et al.* [2006] stored three-dimensional texture coordinates in a grid structure and advected them like any other scalar field. To avoid artifacts resulting from volumetric advection the authors used extrapolation techniques to force the gradient of the texture field to be perpendicular to the free surface normal. This approach has also been successfully used in production, in particular for texturing the tar monster in *Scooby Doo 2*. Unfortunately, both these approaches suffer from problems with discontinuous and distorted parameterizations and can only be used for very short sequences before the distortions become too great.

2.6 Example-Based Texture Synthesis

Example-based texture synthesis has been a popular research area in computer graphics and vision. Heeger and Bergen [1995] analyzed input textures by computing filter response histograms at different spatial scales and then synthesized new textures which matched these histograms. De Bonet [1997] also used a multi-resolution filter-based approach, but linked the various spatial frequencies by conditioning finer scales on decisions made at coarser scales. More recently, Efros and Leung [1999], developed a very simple and elegant texture synthesis method, which "grows" textures, pixel by pixel, outward from an initial seed. They model texture as a Markov Random Field, which implies that the color of each individual pixel in the texture depends only on the colors of pixels in its spatial neighborhood and is independent of the rest of the image. Wei and Levoy [2000] developed a very similar technique, but used multiresolution synthesis and tree-structured vector quantization to speed the generation of new textures. Efros and Freeman [2001] took a patch-based

approach—instead of choosing each individual pixel, they choose entire patches of texture at a time. The patches overlap slightly and a minimum error boundary cut is found in this overlap region to stitch the various patches together.

Our texture synthesis method is based on the flexible optimization approach developed by Kwatra *et al.* [2005]. This approach divides the output texture into a number of overlapping, square patches. Each of these patches is mapped to some region of the example texture. Each pixel, p, in the output texture will be covered by a nonempty subset of these patches. Each patch, P_i, in this subset provides a mapping from p to some pixel, q_i, in the example texture. The color of p can then be computed as a weighted average of the colors of the q_i. The texture is optimized using an two-step expectation-maximization approach:

1. Holding the output texture constant, optimize the mapping.

2. Holding the mapping constant, optimize the output texture.

The mapping is optimized by finding, for each patch, P_i, the region in the example texture which most closely matches P_i. The quality of a match is determined by comparing the region of the output texture covered by P_i to the region of the example texture. The output texture is optimized by computing new pixel colors using the updated mapping. These steps are then repeated until convergence.

2.7 Mesh Parameterization

In order to compare surface textures to the two-dimensional input example texture, a necessary step in our texture optimization, we must construct some parameterization of the surface. Numerous methods for the automatic generation of parameterizations of arbitrary surfaces exist. These methods can be roughly divided into two categories: methods

that parameterize a set of (potentially overlapping) small patches and methods that attempt to find a globally-optimal parameterization. Our work belongs to the first category. The pioneering work of Bennis *et al.* [1991] introduced the idea of using piecewise parameterizations of surfaces for texture mapping. Later, Maillot *et al.* [1993] introduced the concept of texture atlases, which allow the surface to be broken up into patches where each patch has its own parameterization and texture. They also introduced a widely used surface-flattening heuristic. The lapped textures technique of Praun *et al.* [2000] places overlapping, irregularly shaped texture patches on the surface. This approach works quite well for many textures and is very similar to our approach, the primary differences being that we optimize the mapping of texture patches onto the surface and that our patches have substantially more overlap. Concurrently, Wei and Levoy [2001] and Turk [2001] introduced texture synthesis methods which create local parameterizations of the surface and then synthesize textures directly on the surface. However, their greedy texture synthesis approach differs from the optimization approach presented here. More recently, Sorkine *et al.* [2002] introduced a greedy method for creating bounded-distortion local surface parameterizations based on a simple distortion metric, which was introduced by Sander *et al.* [2001]. We make use of this distortion metric when building our surface patches.

Chapter 3

Semi-Lagrangian Contouring

The fundamental problem of tracking a surface as it is advected by some velocity field arises frequently in applications such as surface reconstruction, image segmentation, and fluid simulation. Unfortunately, the naïve approach of simply advecting the vertices of a polygonal mesh, or other explicit representation of the surface, quickly encounters problems such as tangling and self-intersection. Instead, a family of methods, known as level-set methods, has been developed for surface tracking. These methods represent the surface implicitly as the zero set of a scalar field defined over the problem domain. The methods are widely used, and the texts by Sethian [1999] and Osher and Fedkiw [2003], and Osher and Sethian's [1988] seminal article provide an excellent introduction to the topic. One of the key issues that distinguishes various level-set and similar approaches is the representation of the scalar field, which must capture whatever surface properties are important to a given application.

Our surface tracking method represents the surface explicitly with a closed, manifold polygon mesh. However, rather than attempting to advect these polygons forward with the flow, we update the surface in time with an implicit representation: an advected

signed-distance function, ψ, whose zero set defines the surface. A new polygonal surface is generated by *contouring* or extracting the zero set of ψ. The value of ψ at a point $\boldsymbol{x}$, at current time t, is obtained by first tracing backward through the flow field to find the previous location $\boldsymbol{x'}$ at time $t - \Delta t$, and then returning the signed distance of $\boldsymbol{x'}$ from the previous surface. Using adaptive octree data structures, we can efficiently and reliably construct the new surface and corresponding signed-distance function.

The theoretical framework for this method comes from a series of articles by Strain [1999b; 1999c; 1999a; 2000; 2001] that described and analyzed a method for contour tracking in two dimensions. While the semi-Lagrangian procedure for backward advection does not change significantly when going from two- to three-dimensional problems, significant surface tracking issues arise when moving to three dimensions. This thesis discusses these issues, as well as the general method, and demonstrates how semi-Lagrangian surface contouring can be useful for animating the complex and interesting behavior of fluids.

One of the primary advantages of this method is that it enables tracking surface characteristics, such as color or texture coordinates on the actual surface. These properties can be easily stored directly on the polygonal mesh and efficiently mapped onto the new surface during semi-Lagrangian advection. The explicit surface representation also facilitates other common operations, such as rendering, while reconstruction from a scalar function allows operations that rely on an implicit representation. Finally, the method produces detailed, well-defined surfaces that are suitable for realistic animation and that do not jitter or exhibit other undesirable behaviors.

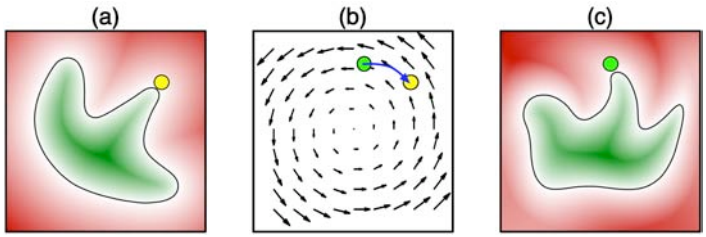

Figure 3.1: An overview of our method. At each timestep we begin with an explicit surface representation, from which we can build a signed-distance function (a) and a velocity field given by the fluid simulator (b). We then define a field function, the zero set of which will be our new surface. To get the value of the field function at the green point (c), we trace backward through the flow field to find the yellow point (b), which is the image of the green point at the previous timestep. We then evaluate the signed distance of the yellow point to the previous surface (a) and copy this value to the green point (c). We can evaluate this field function at every point in the domain and extract the zero set (c).

3.1 Method Overview

The surface tracking problem can be phrased as: given a surface representation and a velocity field at time t, build a representation of the surface at time $t + \Delta t$. We begin with a triangle mesh and an octree annotated with signed-distance field samples. We could try to advect the mesh points through the flow field, but would quickly encounter significant topological difficulties. Instead, we avoid topological issues by updating the surface using an implicit representation. The implicit representation is then used to construct a new mesh at the current timestep. More specifically, we define a scalar-valued function which relates the surface at the current timestep to the surface at the previous timestep. Next, we extract the zero set of this function using a contouring algorithm. Finally, a new signed-distance field is computed through a process known as *redistancing* (see Figure 3.1).

3.2 Hybrid Representation

3.2.1 Explicit Representation

One of the key differences between our method and other surface tracking methods is that we build an explicit representation of the surface at every timestep. This explicit representation is a closed, manifold triangle mesh, which is stored as an array of vertices and an array of triangles. The vertices are shared between triangles, allowing for easy computation of smooth vertex normals and other common mesh operations. The distance tree (see Section 3.3) provides a spatial index for the mesh. The explicit representation provides our method with several advantages. First, it allows us to compute *exact* signed-distance values near the mesh. Second, it allows us to store properties on mesh vertices, rather than at points near the mesh. Finally, it allows us to take advantage of the many tools and algorithms which have been developed in computer graphics for manipulating and rendering triangle meshes.

3.2.2 Implicit Representation

To avoid the topological difficulties of directly updating an explicit surface representation, we update the surface in time through an implicit representation (see Figure 3.2). We define a scalar-valued field function, $\psi(\boldsymbol{x})$, which relates the surface at the current timestep to the surface at the previous timestep. The surface at the current timestep will be the zero set of this function,

$$S_n = \{\boldsymbol{x} : \psi(\boldsymbol{x}) = 0\}. \tag{3.1}$$

For a point $\boldsymbol{x}$ at the current timestep, the function, ψ, first uses backward path tracing, a semi-Lagrangian integration technique, to find the point $\boldsymbol{x'}$ at the previous timestep which

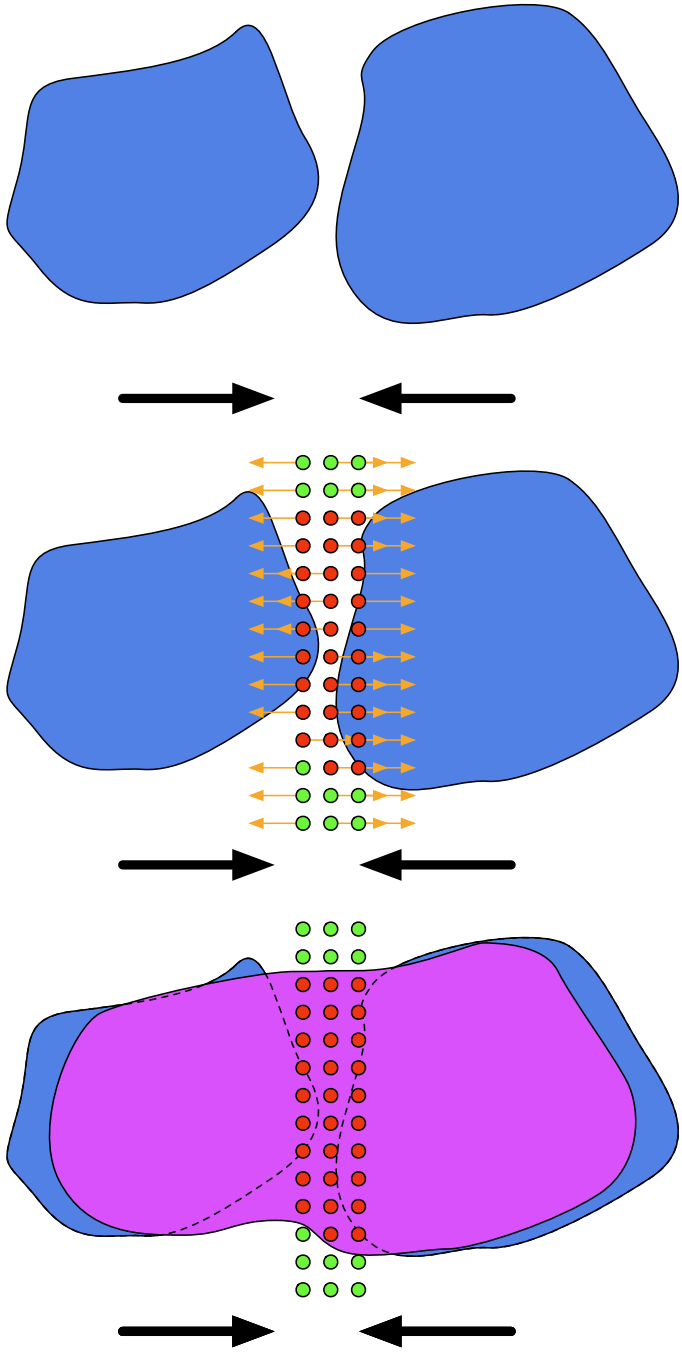

Figure 3.2: An example of how our implicit representation accommodates merging surfaces. The red grid points trace back through the velocity field to points inside the surface. The green grid points trace back to points outside the surface. When the contouring algorithm runs, it will look for zero crossings only between positive and negative (green and red) grid points and create a surface that does not pass between two grid points of the same color. Thus, without even explicitly determining that a topological change has occurred, the method handles the change.

flows to $\boldsymbol{x}$. It then returns the distance from $\boldsymbol{x'}$ to the surface, S_{n-1}, at the previous timestep. If we denote the backward path tracing as $\boldsymbol{b}(\boldsymbol{x}) : \mathcal{R}^3 \rightarrow \mathcal{R}^3$ and let $\phi_n(\boldsymbol{x})$ be the signed distance from $\boldsymbol{x}$ to the surface S_n,

$$\psi_n(\boldsymbol{x}) = \phi_{n-1}(\boldsymbol{b}(\boldsymbol{x})) = \phi_{n-1}(\boldsymbol{x'}). \tag{3.2}$$

Essentially, we are advecting the signed-distance function through the velocity field given by the fluid simulator. In solving this advection term, our method differs from the simple CIR scheme discussed earlier in two ways. First, instead of the simple linear backward path tracing, we use a second-order Runge-Kutta scheme (also known as the midpoint method with an Euler predictor)

$$\boldsymbol{x_{n-1/2}} = \boldsymbol{x}(t_{n-1/2}) = \boldsymbol{x_n} - \frac{\Delta t}{2}\boldsymbol{v}(\boldsymbol{x_n}, t_n), \tag{3.3}$$

$$\boldsymbol{x_{n-1}} = \boldsymbol{x}(t_{n-1}) = \boldsymbol{x_n} - (\Delta t)\boldsymbol{v}(\boldsymbol{x_{n-1/2}}, t_n), \tag{3.4}$$

where $\boldsymbol{v}(\boldsymbol{x}, t)$ is the velocity function. It is important to note that, while this method traces back through the velocity field with second-order accuracy, the velocity field is frozen over the course of the timestep, leading to first-order accuracy in time. The second difference is that, when evaluating ϕ at points near the surface, we do *not* interpolate values stored on a grid. Instead, we compute *exact* distance values. These changes only improve the accuracy (consistency) of our method and do not affect the unconditional stability.

To compute the exact distance from a point $\boldsymbol{x'}$ to the surface, we compute the distances d_i to all the nearby triangles. The distance to the surface is then computed as $\min_i d_i$. Schneider and Eberly [2002] detailed a method for computing the distance from a point to a triangle. This operation is relatively expensive, but many triangles can be pruned, especially when $\boldsymbol{x'}$ is very close to the surface, by using standard bounding-box techniques and our octree data structure (see Section 3.3).

Signing the distance values turns out to be somewhat difficult near sharp corners. Let $\boldsymbol{y}$ and $\boldsymbol{n}(\boldsymbol{y})$ denote the closest point on the surface to $\boldsymbol{x'}$ and its normal, respectively. When $\boldsymbol{y}$ lies strictly inside a triangle then the sign can be easily computed as

$$s = sign((\boldsymbol{x'} - \boldsymbol{y}) \cdot \boldsymbol{n}(\boldsymbol{y})), \tag{3.5}$$

where $\boldsymbol{n}(\boldsymbol{y})$ is the normal of the triangle containing $\boldsymbol{y}$. However, if the nearest point in the mesh lies on more than one triangle (i.e., on an edge or vertex of the mesh), the triangles do not always agree on the sign. These situations can be resolved by computing an angle-weighted pseudonormal for each edge and vertex of the mesh and using these pseudonormals to determine the sign when the nearest point is on an edge or vertex of the mesh. Bærentzen and Aanæs [2005] provided a proof that this procedure results in accurate signing (in exact arithmetic).

The ability to compute exact distances is one of the chief advantages of having an explicit surface representation. Interpolation can produce substantial errors (see Figure 3.3) which are compounded over time. In fact, this interpolation error is one of the most significant drawbacks to semi-Lagrangian methods in general. When used for velocity advection, interpolation produces such significant smoothing that researchers have proposed a number of methods to add detail back to the flow [Fedkiw et al., 2001] or avoid semi-Lagrangian advection altogether [Zhu and Bridson, 2005]. In this work, we are able to leverage the advantages of semi-Lagrangian advection, without incurring the interpolation error that would otherwise undesireably smooth surface detail.

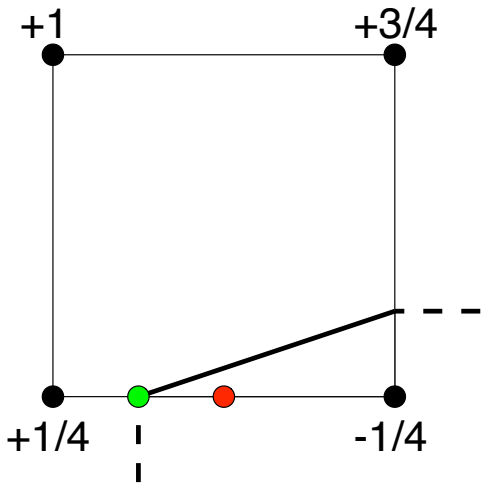

Figure 3.3: The figure shows a part of the surface passing through a grid cell. The cell's vertices have been annotated with signed-distance values. Linear interpolation of these values incorrectly chooses the red point as the zero crossing along the bottom edge. The green point is the actual zero crossing, which will be found with exact evaluation.

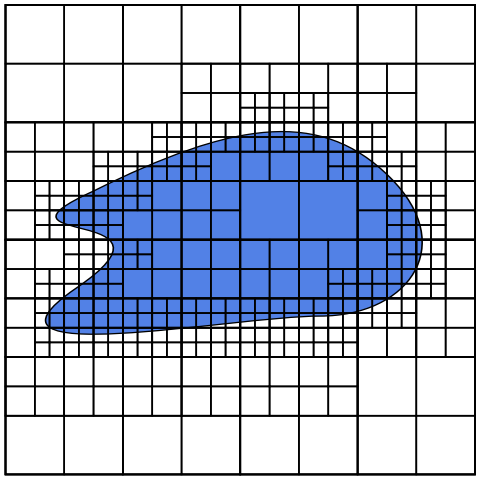

Figure 3.4: A two-dimensional distance tree. Distance samples are stored at the octree vertices and triangle lists are stored in cells which intersect the surface. This distance tree could be generated using our implementation of Criterion (3.8), which considers ψ only at cell centers.

3.3 The Distance Tree

Our implementation makes heavy use of a structure we call the *distance tree*. The distance tree is a balanced octree subdivision of the spatial domain. The octree vertices are annotated with signed-distance values and each cell of the octree contains a list of the triangles with which it intersects. The distance tree serves three purposes:

1. It provides a fast spatial index for the mesh so that nearby triangles can be found quickly.

2. It provides a fast, approximate signed-distance function, which is sufficient when evaluating the signed distance far from the surface.

3. It guides the contouring algorithm, quickly identifying cells which have vertices of different sign and, thus, contain triangles.

3.3.1 Approximating the Signed-Distance Function

When computing the signed distance from a point $\boldsymbol{x'}$ to a surface, S, we first find the smallest octree cell, $\boldsymbol{C}$, containing $\boldsymbol{x'}$. If $\boldsymbol{C}$ is at the finest level of the octree, then $\boldsymbol{x'}$ may be near the surface and all the triangles in the up to 27 cells in the concentric triple[1] of $\boldsymbol{C}$ are considered when computing the minimum distance to the surface. By storing the nearest distance seen so far and using standard bounding-box techniques, many of these triangles can be pruned before computing distances, especially when $\boldsymbol{x'}$ is very near the surface. If the computed distance is less than $\boldsymbol{C}$'s edge length, then the distance is guaranteed to be exact. Otherwise, the computed distance is a very good estimate but may be slightly larger than the actual distance. Contrariwise, if $\boldsymbol{C}$ is not at the finest level of the octree or if there are no triangles in the concentric triple of $\boldsymbol{C}$, then $\boldsymbol{x'}$ is not near the surface and we do not require an exact distance. An approximation with the correct sign is sufficient. In this case, we use trilinear interpolation of the distance values stored at the vertices of $\boldsymbol{C}$.

3.3.2 General Splitting Criterion

We make use of two different methods for building distance trees in this work. Most often, we wish to build a distance tree to resolve the zero set of our field function ψ. However, it is sometimes useful to build a distance tree from an existing triangle mesh. Our octrees are always built in a top-down manner where each cell is split based on some

[1] If cell $\boldsymbol{C} = \{\boldsymbol{x} : \|\boldsymbol{x} - \boldsymbol{c}\|_\infty \leq r\}$ has center $\boldsymbol{c}$ and edge length $2r$ then its concentric triple T is given by $T = \{x : \|\boldsymbol{x} - \boldsymbol{c}\|_\infty \leq 3r\}$.

variation of the following splitting criterion:

Split any cell whose edge length exceeds its minimum distance to the surface. (3.6)

Splitting ends when the tree reaches a predetermined maximum depth. Criterion (3.6) results in a three-color octree, as described by Samet [1990], where each cell of the octree has one of three types: interior, exterior, or boundary (see Figure 3.4).

In general, Criterion (3.6) builds octrees with several useful properties:

- Adjacent cells differ in size by no more than a factor of 2, producing a smooth mesh and simplifying procedures such as neighbor finding and triangulation of the vertices.
- A cell's size is proportional to its distance to the surface.
- If ϕ is the signed distance to the surface at vertices and we extend ϕ into each cell by trilinear interpolation, then, because cells vary in size, ϕ will be discontinuous. However, the jumps in ϕ decrease in size in cells near the surface because of the triangle inequality. Thus the interpolated ϕ is nearly continuous near the surface.
- Cells coarsen very rapidly away from the surface: if there are N childless cells touching the surface, then the entire tree contains only $O(N \log N)$ cells. Hence the surface is resolved accurately at minimal cost.

3.3.3 Building a Distance Tree to Resolve ψ

When building a new octree at the beginning of each timestep, we are essentially trying to resolve our approximation

$$\psi_{n+1}(\boldsymbol{x}) = \phi_n(\boldsymbol{x} - (\Delta t)\boldsymbol{v}(\boldsymbol{x_{n-1/2}}, t_n)) \tag{3.7}$$

to the signed-distance function $\phi_{n+1}(\boldsymbol{x})$. The octree is built recursively from the root cell C_0 using the following splitting criterion:

$$\textit{Split every cell where } |\psi_{n+1}| \textit{ is smaller than the edge length.} \tag{3.8}$$

Thus we apply Criterion (3.8) as if ψ_{n+1} were a distance function. Redistancing every timestep keeps

$$\psi_{n+1} = \phi_n + (\Delta t)\boldsymbol{v} \cdot \nabla\phi_n + O(\Delta t) = \phi_n + O(\Delta t) \tag{3.9}$$

within $O(\Delta t)$ of the signed-distance function ϕ_n. Thus in the limit, $\Delta t = O(\Delta x) \to 0$, Criterion (3.8) reduces to (3.6), yielding the properties noted above. In practice, we use the value of ϕ at the cell's center to determine whether we should split the cell. To deal with the fact that ψ_{n+1} is not a distance function and that the value at the cell's center may not be the minimum over the cell, we multiply the edge length by some constant before doing the comparison. We have found that 3 works well in practice—always dividing near the surface, without spuriously dividing too many cells. Notice that we can vary this constant to achieve high-resolution bands of varying width around the surface.

3.3.4 Building a Distance Tree from a Triangle Mesh

When building an octree from a triangle mesh (either in initialization, or after some geometric operation has been applied to the triangle mesh) we use the following splitting criterion:

$$\textit{Split every cell whose concentric triple intersects the surface.} \tag{3.10}$$

This test is efficiently implemented using Green and Hatch's [1995] cube/triangle intersection test. Notice that we need not check every cell in the concentric triple of C individually,

but can just increase the edge length passed to the intersection test, effectively increasing the size of C. In practice we have found it sufficient to increase the edge length by a factor of 2, rather than 3, but such trees may not satisfy all the properties listed above.

3.4 Contouring

Once we have resolved ψ on our distance tree, we need to create an explicit representation of our surface at the new timestep. Creating this explicit representation amounts to extracting the zero set of ψ and is an instance of the contouring problem, which has been well studied in computer graphics. For its simplicity, robustness, and speed, we choose to use a marching-cubes method in our implementation. Our implementation is based on Bloomenthal's [1994]. Our cubes are the leaf cells in the distance tree which have vertices of differing sign. We divide each cube into six tetrahedra to simplify the implementation. Additionally, when finding the zero crossing along any edge (which will eventually be a vertex in the triangle mesh), we use a secant method to speed up convergence and evaluate our full composite field function, including exact evaluation of the previous signed-distance function. Consequently, the vertices of our polygon mesh are guaranteed to lie on the implicit surface (within an ϵ tolerance). In fact, each vertex in our polygon mesh can be mapped to some point on some triangle in the mesh at the previous timestep. We take advantage of this fact when advecting surface properties. The marching-cubes algorithm works well for our purposes because each triangle generated by marching cubes sits strictly inside a single cell of the distance tree, making the distance tree an especially effective spatial index. Furthermore, we use the distance tree we have already built to guide the marching cubes, avoiding the need to build a second structure to determine the topology of the new mesh. Near the surface, our distance tree is refined to the maximum level and looks like a uniform

grid. Consequently, we need not worry about patching the marching-cubes solution.

Our choice of contouring algorithm does result in some limitations. In addition to creating poorly shaped triangles, marching cubes is nonadaptive. That is, the sampling is as dense in flat regions as in regions of high curvature. Unfortunately, the nonadaptive nature of marching cubes limits the resolution we can achieve in high-curvature areas, but is necessary to ensure compatibility. To address this lack of resolution in high-curvature areas, Strain [2001] split line segments whose centers were far from the surface, yielding arbitrarily high accuracy. Unfortunately, this splitting technique is not easily extended to three dimensions as splitting a triangle either creates an incompatible triangulation or produces even more poorly shaped triangles. It is also very difficult to guarantee that we will still have a manifold when the inserted vertices are moved to the surface. Alternatively, several adaptive contouring methods [Shu et al., 1995; Shekhar et al., 1996; Poston et al., 1998] seek to use adaptive grids and regain compatibility through various crack-patching techniques. Such methods could easily be used here and we plan to explore adaptive methods in future work.

Although we did not find it necessary, after the contouring step the mesh can be processed in any way that preserves the closed-manifold invariant. This optional processing might include smoothing the surface, improving the shape of the triangles, or any other operation that returns a closed manifold. A new distance tree can then be built from this modified mesh using Criterion (3.10). A new distance must be built only if the mesh is modified.

By taking advantage of the details of our method, we can very efficiently achieve limited smoothing in two ways. First, we can define a second scalar function to be the combination of path tracing backward in time followed by the evaluation of a high-order

polynomial interpolant of the distances at the vertices of the octree. This function is quite similar to the functions used in semi-Lagrangian level-set methods [Strain, 1999b; Enright et al., 2005]. When marching cubes encounters an edge whose vertices have different signs, we find a point which evaluates to zero for both scalar functions. We can then average these two zero-crossing to compute the final mesh vertex. By constraining the mesh vertex to be on the edge of the marching-cubes grid, we still guarantee a consistent, closed, manifold triangulation. While this smoothing technique may be quite useful in some applications, we did not use this method for any of the results in this thesis. Second, repeatedly using the same grid for contouring can produce grid artifacts. For example, a sphere of fluid falling under gravity will develop creases along the coordinate axes. Such artifacts are a form of aliasing and can be reduced by jittering the grid each timestep. Most of the examples in this thesis used grids which were slightly larger than the simulation domain. These grids were then randomly perturbed so that grids at adjacent timesteps were slightly offset from one another. This jittering limits the reusability of our octrees, but since we build new octrees every timestep, this limitation is not significant.

3.5 Redistancing

After the triangle mesh at the current timestep has been extracted, we must assign true distance values to the vertices of our octree. This problem, referred to as *redistancing*, has been well studied by the level-set community and a number of methods have been suggested. Strain [1999a] suggested redistancing by performing an exact evaluation at every vertex of the octree. This method is relatively efficient since the tree coarsens rapidly away from the surface and works well in two dimensions. However, in three dimensions, we have found it to be prohibitively expensive and unnecessary. Instead, we perform exact

evaluation at all vertices of the cells that contain triangles, but then run a fast marching method [Sethian, 1996; Losasso et al., 2004] over the remaining vertices. In our method, there may be some parts of the domain where the octree was refined but that did not result in any triangles, such as when the surface becomes thinner than the resolution of the tree. Consequently, our octree, unlike those used by Losasso *et al.* [2004], does not necessarily coarsen away from the surface. To address this problem, we coarsen parts of the tree which have been refined but did not generate surface. We do this coarsening in two steps. First, we propagate the triangle lists up the tree so that the triangle list of a cell is the union of the triangle lists of a cell's descendants. Second, we remove all the children of any cell whose concentric triple does not contain any triangles.

Our redistancing method comprises three steps:

- coarsen the octree;
- compute exact distances at vertices of cells which contain triangles;
- run a fast marching method over the remaining vertices.

3.6 Tracking Surface Properties

One of the primary advantages of our method is the ability to track surface properties, such as color, texture coordinates, or even simulation variables, accurately at negligible additional cost. As pointed out earlier, every vertex in a polygon mesh corresponds to some point on some triangle in the previous mesh. Thus, semi-Lagrangian advection provides a mapping between surfaces at adjacent timesteps. If vertex $\boldsymbol{v}$ in the current mesh maps to point $\boldsymbol{p}$ in the old mesh and some surface property was stored at $\boldsymbol{p}$, this property can be copied to $\boldsymbol{v}$. In this way we can track surface properties on the actual surface as we build

the surface, so we do not incur any significant additional cost. Previous methods, such as the ones proposed by Rasmussen *et al.* [2004] and Wiebe and Houston [2004], have been limited to tracking properties in the volume near the surface and interpolating them to the surface. Such methods incur significant cost, introduce substantial smoothing, and blur properties between nearby surfaces.

In many applications there is no value actually stored at $\boldsymbol{p}$. Instead, the properties are stored at the vertices of the triangle containing $\boldsymbol{p}$. In these cases the problem is slightly more involved. In many cases it is sufficient to use barycentric interpolation to compute a value at $\boldsymbol{p}$ and copy this interpolated value to $\boldsymbol{v}$. However, for some applications this interpolation can produce unwanted smoothing. A simple alternative is to set the value at $\boldsymbol{v}$ to the value stored at the vertex nearest $\boldsymbol{p}$. Unfortunately, this approach may introduce unwanted aliasing. Essentially, we are having trouble because we are resampling the surface at every timestep. However, if we know something about the property we are tracking, we may be able to "clean up" the blurred signal. For example, in some cases we wish to track reference coordinates, which can later be used as texture coordinates, passed to procedural shaders, or for other purposes. Since we know that the tracked value should always be a point on the initial surface we can find the point on the initial mesh which is nearest to the point the tracking method supplied. In this way, we can ensure that, at every timestep, every vertex in the mesh maps back to some point on the initial surface. Once we have this mapping we can copy any property stored on the initial surface, whether it be the reference coordinates, texture coordinates, or color values. Texturing liquids will be discussed in more detail in the next chapter.

Chapter 4

Texturing Liquid Surfaces

Liquid simulation techniques have become a standard tool in production environments, producing extremely realistic liquid motion in a variety of films, commercials, and video games. Surface texturing is an essential computer graphics tool, which gives artists additional control over their results by allowing them to stylize surfaces or add detail to low-resolution simulations. For example, an artist could use texturing techniques to add the appearance of foam to a wave, bubbles to beer, or fat globules to soup. Unfortunately, texturing liquid surfaces is difficult because the surfaces have no inherent parameterization.

Creating a temporally consistent parameterization is extremely difficult for two primary reasons. First, liquid simulations are characterized by their complex and frequent topological changes. These topological changes result in significant discontinuities in any parameter tracked on the surface. Second, liquid surfaces tend to stretch and compress dramatically over the course of a simulation. Similarly, an advected parameterization will also stretch and compress. While it may be appropriate to squash and stretch some textures with the motion of the liquid surface, many textures, such as fat globules on the surface of soup, should maintain a particular scale even as the liquid surface deforms. For these

reasons, advected texture coordinates are often unsuitable for texturing liquid surfaces.

To address these problems, we have designed a new example-based texture synthesis method specifically for liquid animations. Rather than advecting texture coordinates on the surface, we synthesize a new texture for every frame. We initialize the texture with color values advected from the surface at the previous frame. We then run an optimization procedure which attempts to match the surface texture to an input example texture and, for temporal coherence, the advected colors.

By synthesizing a new texture for every frame, our method is able to overcome the discontinuities and distortions of an advected parameterization. We avoid discontinuities in the parameterization due to topological changes by building a new parameterization of the surface for each frame. Discontinuities in advected colors are removed during the optimization procedure. Similarly, we avoid stretched and compressed parameterizations; because we optimize the surface texture for every frame, it maintains a consistent level of detail throughout the animation. We ensure temporal coherence by initializing the optimization with the advected colors and including a coherence term in the energy function used during optimization. As a result, our method is able to produce textures with excellent temporal coherence, while still matching the input example texture. Before discussing the details of our example-based texture synthesis method, I will first discuss alternative approaches to texturing animated liquid surfaces.

4.1 Advected Colors

Perhaps the simplest method for texturing liquids is to advect color values on the surface. This is easily done using the mapping provided by the semi-Lagrangian contouring surface tracking method. For each vertex, $\boldsymbol{v}$, in our mesh, we find the corresponding point,

$\boldsymbol{p}$ on the previous mesh, use barycentric interpolation to compute a color value at $\boldsymbol{p}$, and copy this value to $\boldsymbol{v}$. If multiple colors were stored at the vertices of the triangle containing $\boldsymbol{p}$, these colors will be blurred together. In many cases this behaviour is acceptable, even desirable, as it appears that the various colors are mixing as the fluid flows, much like mixing paint. One drawback of this approach is that this blurring is tied to the timestep and mesh resolution and is not a parameter which can be tuned by an artist.

4.2 Advected Parameterizations

Another simple approach to texturing liquid surfaces would be to advect some parameterization (for example, texture or reference coordinates) and use standard texture mapping or procedural shading techniques to generate a texture. This approach works reasonably well and has been used successfully in production for very short sequences. In longer sequences, the textures become too distorted. There are several sources of this distortion. First, topological changes lead to discontinuities in the advected parameterization which manifest as seams in the texture. Second, liquid surfaces tend to stretch and compress, leading to loss of detail and aliasing when using noise-based procedural textures. These problems can likely be addressed by adapting the work of Neyret [2003] or Cook and DeRose [2005]. However, these techniques essentially push the problem onto the shader writer who must now make certain that the shader behaves properly at a variety of frequency bandwidths. Furthermore, these approaches do not offer solutions to discontinuities arising from topological changes, or distortions to the texture due to a swirling surface.

4.3 Reaction-Diffusion Texture Synthesis for Liquids

Reaction-diffusion [Turk, 1991; Witkin and Kass, 1991] systems provide a more sophisticated approach to generating textures. Motivated by the chemical reactions that generate patterns on animals, reaction-diffusion is a process in which two or more chemicals, or *morphogens*, diffuse at unequal rates over a surface and react with one another to form stable patterns such as spots and stripes. An example of a reaction-diffusion system, which generates surface spots, due to Meinhardt [1982; 1992b] is:

$$\begin{aligned} \frac{\partial a}{\partial t} &= s(ap_1 + \frac{0.01a_i a^2}{b} + p_3) + D_a \nabla^2 a \\ \frac{\partial b}{\partial t} &= s(bp_2 + 0.01a_i a^2) + D_b \nabla^2 b, \end{aligned} \tag{4.1}$$

where a and b are scalar fields stored on the surface which are updated in fictitious time according to Equation (4.1), p_1, p_2, and p_3 are parameters of the system, D_a and D_b control the rates of diffusion for the different morphogens and s controls the reaction speed. When generating the actual surface color, we compare the values of a and b. If a is larger we choose one color, if b is larger we choose another color.

Reaction-diffusion texturing methods are attractive in general because they avoid the often difficult task of assigning texture coordinates to a complex surface. In the context of texturing liquid animations they are even more attractive because they obviate the need to advect a parameterization. Instead, we simply advect simulation variables, morphogens, which are used to initialize the simulation at the next frame. Thus, reaction-diffusion texture synthesis is able to deal with topological changes and surface distortions. Unfortunately, they only admit limited types of textures and suffer from the fact that very small perturbations of the surface can substantially change the resulting texture. Consequently, small surface motion can cause large changes in the texture.

4.4 Example-Based Texture Synthesis for Liquids

Similar to reaction-diffusion texture synthesis, our example-based texture synthesis method can handle the topological changes and surface distortions characteristic of liquids. Our example-based texture synthesis method for liquid animations is built from three relatively new computer graphics technologies: the ability to track surface properties in liquid simulations (see Chapter 3), techniques to parameterize overlapping patches of surface [Praun et al., 2000; Sorkine et al., 2002], and an optimization-based technique for texture synthesis [Kwatra et al., 2005]. By combining these three methods we have developed a new algorithm that generates coherent, undistorted textures on liquid surfaces based on example textures.

4.4.1 Surface Tracking

The motion in our examples is generated using a state-of-the-art physically based liquid simulator. More specifically, we use the staggered-grid data structure of Foster and Metaxas [1996], the semi-Lagrangian advection method introduced by Stam [1999], the extrapolation boundary condition of Enright *et al.* [2002b], the viscoelasticity model of Goktekin *et al.* [2004], and the surface tracking method described in Chapter 3.

A necessary feature of any liquid simulation system is the ability to track the liquid's free surface. While several surface tracking techniques exist, the semi-Lagrangian contouring method presented in Chapter 3 also provides a mapping between liquid surfaces at adjacent timesteps. This mapping can be used to accurately track arbitrary surface properties on the actual liquid surface at negligible additional cost. We use this feature to advect colors and parametric directions (see Figure 4.1) on the surface through time. If a different surface tracking method is preferred, the texture particle interpolation method developed

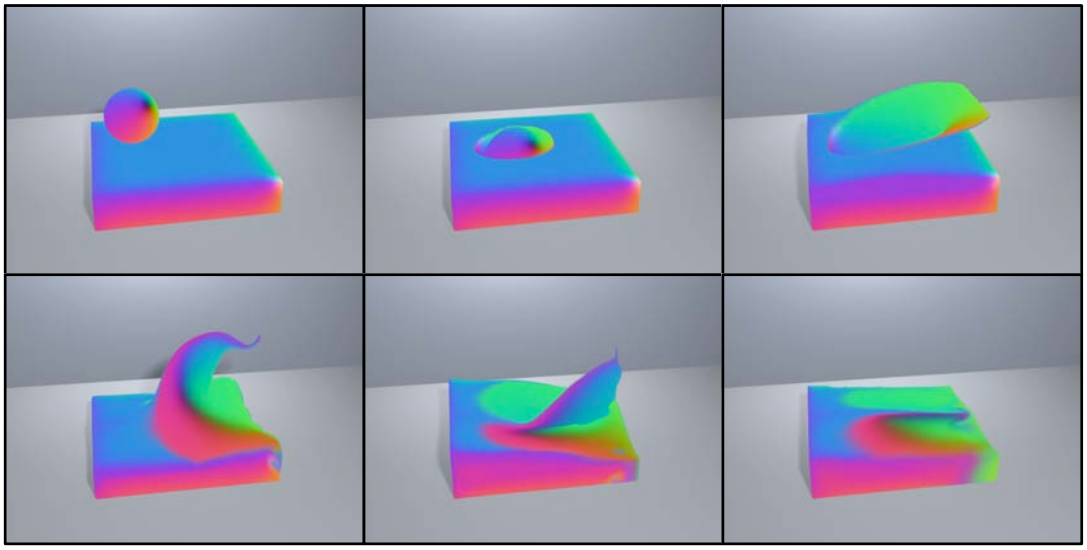

Figure 4.1: These images show the parametric directions which are advected on the surface and used to orient triangles in texture space. One of the parametric direction has been used to color the surface (the x-component in the red channel, the y-component in the green channel and the z-component in the blue channel).

by Rasmussen *et al.* [2004] or the grid-based approach of Wiebe and Houston [2004] and Houston *et al.* [2006] could be used to advect colors, though these approachs would introduce significant computational expense and may cause unwanted blurring between nearby surfaces.

4.4.2 Surface Parameterization

To apply our optimization-based texture synthesis method (see Section 4.4.3), we must construct some parameterization of the surface. In particular, we create local parameterizations of a set of overlapping patches on the surface (see Figure 4.2). For each patch, the parameterization allows us to map colors on the surface to two-dimensional texture space and vice versa.

The surface meshes generated by the liquid simulation system, which uses a marching cubes method, contain many poorly shaped triangles and large dihedral angles. Unfor-

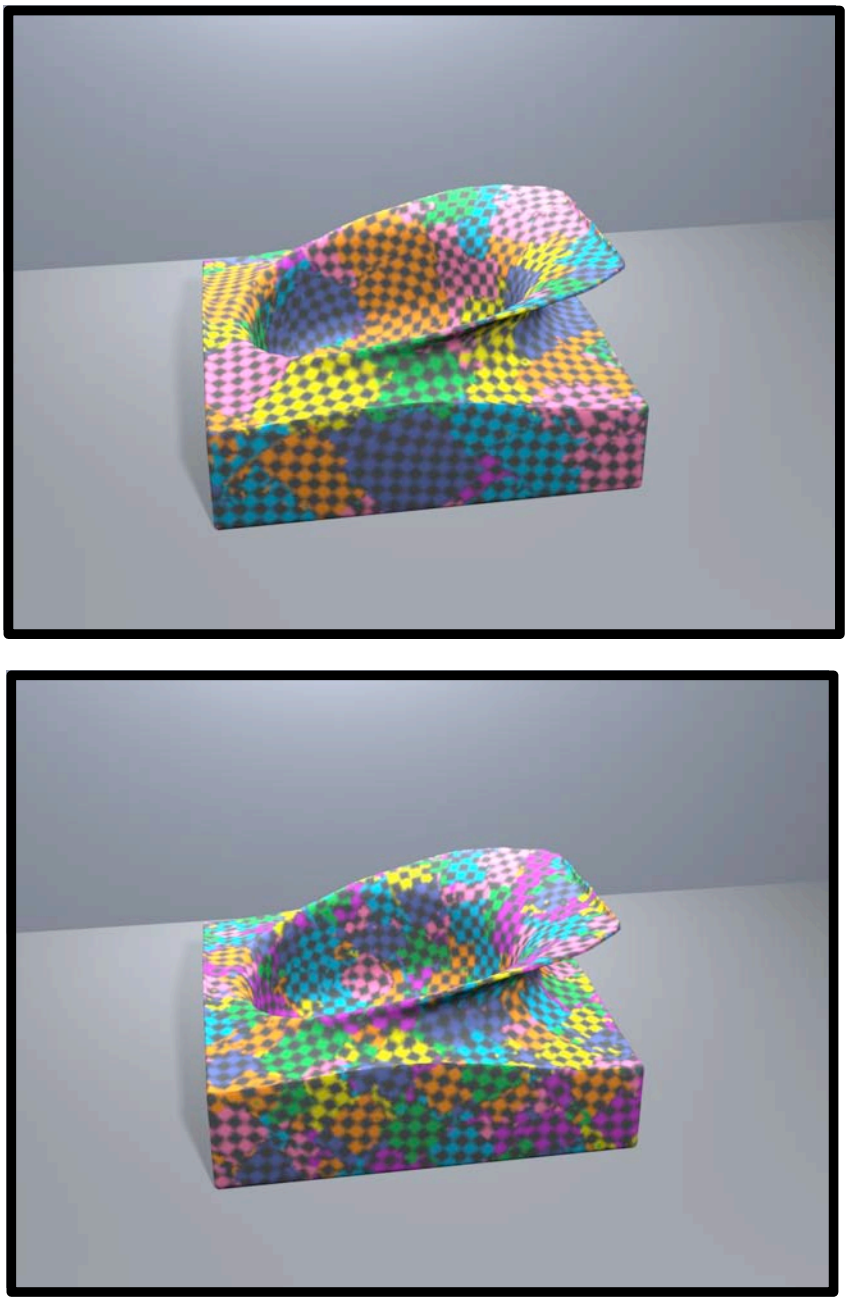

Figure 4.2: These images show the patches used in our optimization process. Each patch will be assigned colors from one region of the input texture and overlapping patches will have their colors blended together. It is interesting to note that we cannot always construct perfect patches and this leads to small holes in some of the patches.

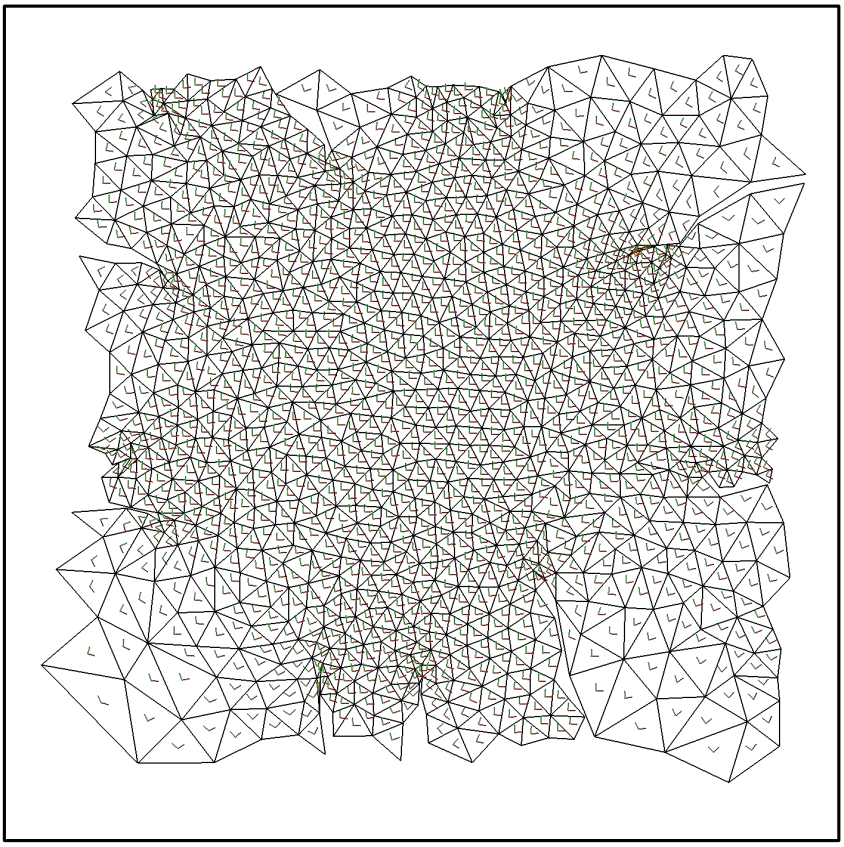

Figure 4.3: This is the two-dimensional parameterization of a surface patch using the popular heuristic of Maillot *et al.* [1993].

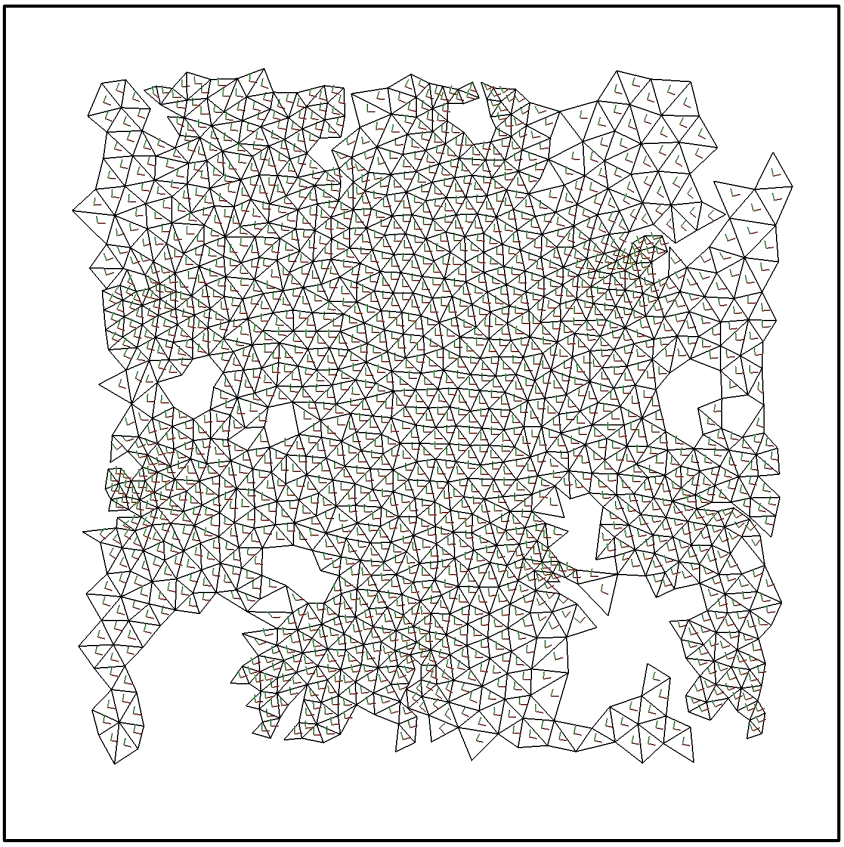

Figure 4.4: This is the two-dimensional parameterization of a surface patch when we do not add triangles which fail the distortion test of Sorkine *et al.* [2002]. The triangles near the boundary are decidedly less distorted than those in Figure 4.3.

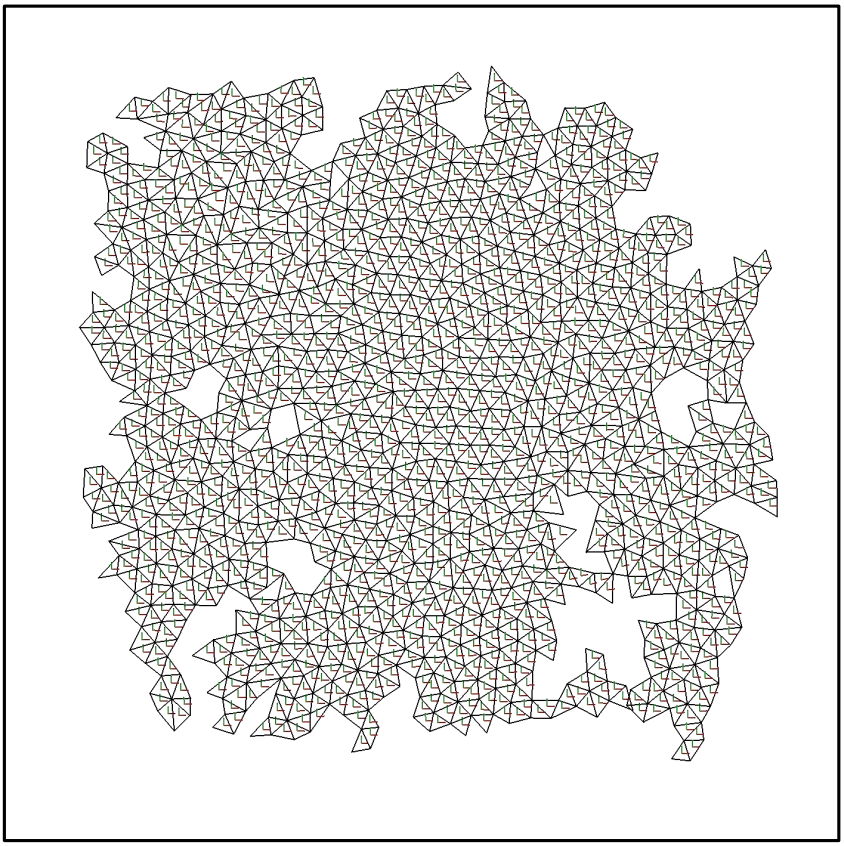

Figure 4.5: This is the two-dimensional parameterization of a surface patch after running the optimization introduced by Praun et al [2000]. The optimization was initialized with a patch constructed using the bounded-distortion approach of Sorkine *et al.* [2002] (see Figure 4.4).

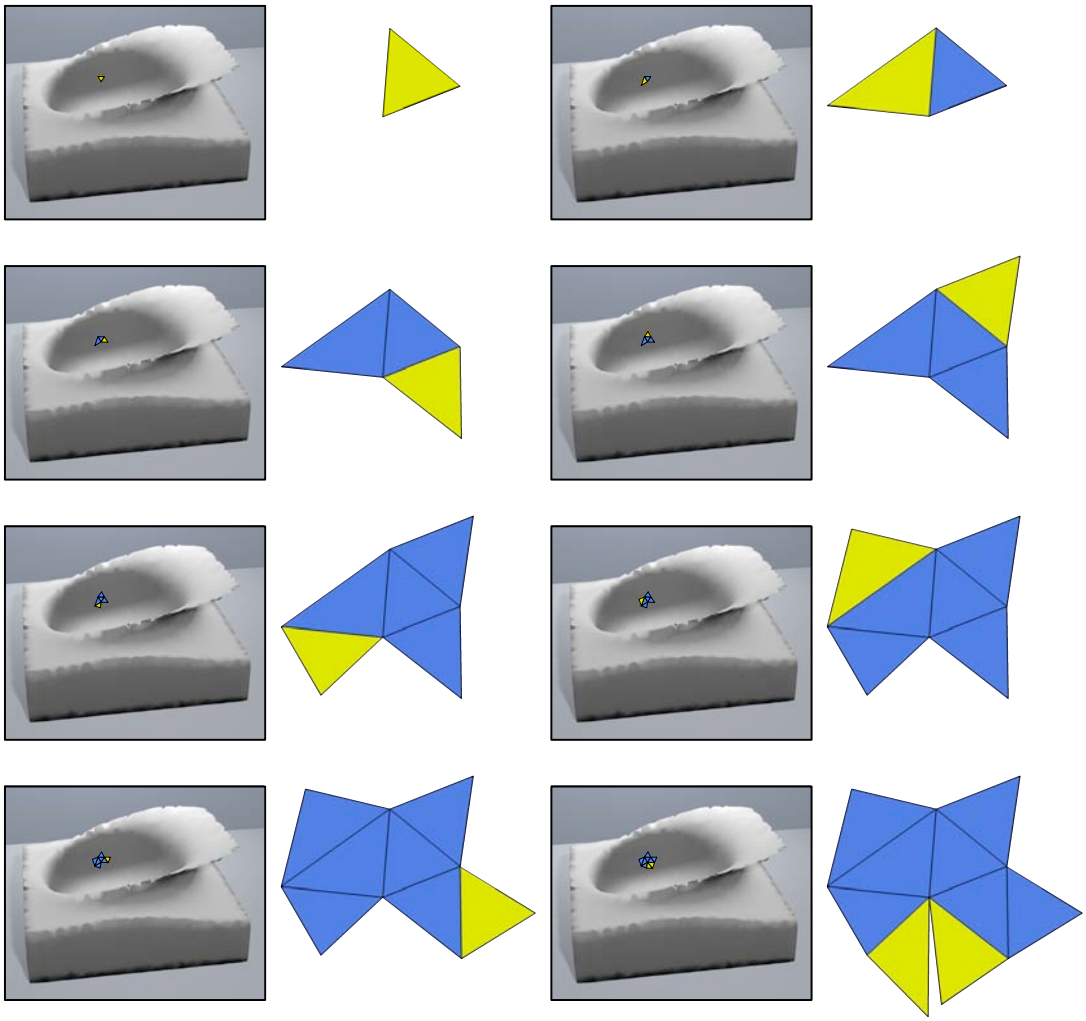

Figure 4.6: These images show the incremental construction of a texture patch. When inserting the final vertex, more than one triangle is created and the actual vertex is inserted at the average of the two predicted locations.

tunately, these meshes do not admit even local parameterizations without significant distortions. Consequently, as a pre-processing step, we re-tile the surfaces using the method presented by Turk [1992a]. This re-tiling step also allows us to control the resolution of the texture on the surface [Wei and Levoy, 2000]. We then generate a set of points which uniformly sample the surface using the repulsion method described by Turk [1992a]. For each point, p_i, we grow a surface patch by iteratively mapping neighboring vertices to the plane (see Figure 4.6). The target size of the patches is a user-specified parameter, which is chosen based on the scale of the features in the example texture we wish to preserve.

We grow our patches by first mapping the triangle, T, containing p_i to a isometric triangle, T', in two-dimensional texture space. We orient T' based on the parametric directions advected during surface tracking (Section 4.4.1) and require that p_i maps to the origin of texture space. Following Wei and Levoy [2001], to avoid aliasing or other artifacts we uniformly scale T' (and hence the entire patch) by a factor of $1/\sqrt{2 \times A}$ (where A is the average triangle area) so that, in texture space, we have roughly the same number of pixels and triangle vertices. After flattening the first triangle, we iteratively add vertices adjacent to the patch. The addition of each vertex, v, will create at least one triangle in texture space. If adding v creates exactly one triangle, then we position v so that the newly created triangle is isometric to its corresponding triangle on the three-dimensional surface. If, however, more than one triangle is created, computing the position of v is more involved. For each triangle created, we compute a candidate position for v based on isometric triangles as in the previous case. The final location of v is then the average of all these candidate locations. This is essentially the same heuristic proposed by Maillot *et al.* [1993] and used by Praun *et al.* [2000] and Wei and Levoy [2001]. However, we do not add any vertex which falls outside the user-specified target patch size, that causes any self-intersections of

the patch in texture space, or that creates any overly distorted triangles. Rejecting overly distorted triangles creates far more uniform patches as can be seen by comparing Figure 4.3 and Figure 4.4. In determining whether a triangle is overly-distorted, we use the distortion metric used by Sorkine *et al.* [2002] and originally introduced by Sander *et al.* [2001]. In describing our distortion metric we follow Sorkine *et al.* [2002].

Let $T = \triangle \boldsymbol{q_1 q_2 q_3}$ be the triangle on the three-dimensional surface and $T' = \triangle \boldsymbol{p_1 p_2 p_3}$ be the triangle in two-dimensional texture space, where $\boldsymbol{p_i} = (s_i, t_i)$. Let $S : R^2 \rightarrow R^3$ be the (unique) affine mapping from the two-dimensional texture space to three-dimensional surface, such that $S(\boldsymbol{p_i}) = \boldsymbol{q_i}$. Let $\langle \boldsymbol{p_1}, \boldsymbol{p_2}, \boldsymbol{p_3} \rangle = ((s_2 - s_1)(t_3 - t_1) - (s_3 - s_1)(t_2 - t_1))/2$ be the area of $\triangle \boldsymbol{p_1 p_2 p_3}$. Then, S is given by

$$S(\boldsymbol{p}) = \frac{\langle \boldsymbol{p}, \boldsymbol{p_2}, \boldsymbol{p_3} \rangle \boldsymbol{q_1} + \langle \boldsymbol{p}, \boldsymbol{p_3}, \boldsymbol{p_1} \rangle \boldsymbol{q_2} + \langle \boldsymbol{p}, \boldsymbol{p_1}, \boldsymbol{p_2} \rangle \boldsymbol{q_3}}{\langle \boldsymbol{p_1}, \boldsymbol{p_2}, \boldsymbol{p_3} \rangle}$$

and the partial derivatives of S are:

$$S_s = \frac{\partial S}{\partial s} = \frac{\boldsymbol{q_1}(t_2 - t_3) + \boldsymbol{q_2}(t_3 - t_1) + \boldsymbol{q_3}(t_1 - t_2)}{2\langle \boldsymbol{p_1}, \boldsymbol{p_2}, \boldsymbol{p_3} \rangle}$$

$$S_t = \frac{\partial S}{\partial t} = \frac{\boldsymbol{q_1}(s_3 - s_2) + \boldsymbol{q_2}(s_1 - s_3) + \boldsymbol{q_3}(s_2 - s_1)}{2\langle \boldsymbol{p_1}, \boldsymbol{p_2}, \boldsymbol{p_3} \rangle}.$$

The singular values of the 3×2 Jacobian matrix $[S_s S_t]$ are:

$$\gamma_{max} = \sqrt{\frac{1}{2}\left((a + c) + \sqrt{(a - c)^2 + 4b^2}\right)}$$

$$\gamma_{min} = \sqrt{\frac{1}{2}\left((a + c) - \sqrt{(a - c)^2 + 4b^2}\right)}$$

where $a = S_s \cdot S_s$, $b = S_s \cdot S_t$, $c = S_t \cdot S_t$.

The values γ_{max} and γ_{min} represent the largest and the smallest scaling caused by the mapping S. Since both stretching and compression lead to geometric distortions, we use the following expression as our distortion metric:

$$D(T, T') = \max\left\{\gamma_{max}, 1/\gamma_{min}\right\}.$$

Note that $D(T, T') \geq 1$, and the equality holds if and only if T and T' are isometric.

At this point we have constructed a parameterization of the surface. Unfortunately, as can be seen in Figure 4.4, far from the center of the patch, this parameterization does not closely match the vector field stored on the surface. During texture synthesis this distortion can cause overlapping patches to find non-overlapping regions of the example texture, degrading the results. Overlapping patches are more likely to find overlapping regions of the example texture if they have consistent parameterizations. To this end, we apply the patch optimization procedure described by Praun *et al.* [2000]. This optimization involves solving a sparse linear system and seeks to align all of the triangles in the patch with the advected parametric directions (see Figure 4.5).

Our discussion follows that of Praun *et al.* [2000]. Let $(\boldsymbol{u}, \boldsymbol{v})$ be the target parametric directions for a given triangle, $\triangle \boldsymbol{q_1 q_2 q_3}$ and $(\hat{\boldsymbol{u}} = (1,0), \hat{\boldsymbol{v}} = (0,1))$ be the parametric directions in texture space. We compute $(\boldsymbol{u}, \boldsymbol{v})$ for a given triangle by averaging the advected parametric directions stored at the triangle's vertices, projecting these directions to the plane of the triangle, and, finally, forcing them to be orthogonal. Now, $(\boldsymbol{u}, \boldsymbol{v})$ lie in the plane of $\triangle \boldsymbol{q_1 q_2 q_3}$ and we therefore can express them using barycentric coordinates:

$$\boldsymbol{u} = \alpha_{\boldsymbol{u}} \boldsymbol{q_1} + \beta_{\boldsymbol{u}} \boldsymbol{q_2} + \gamma_{\boldsymbol{u}} \boldsymbol{q_3}$$

$$\boldsymbol{v} = \alpha_{\boldsymbol{v}} \boldsymbol{q_1} + \beta_{\boldsymbol{v}} \boldsymbol{q_2} + \gamma_{\boldsymbol{v}} \boldsymbol{q_3},$$

where $\alpha + \beta + \gamma = 0$. Since the map S is linear (and invertible) over the face, the image $S^{-1}(\boldsymbol{u})$ is therefore a linear function of the vertex parameterizations $S^{-1}(\boldsymbol{q_1})$, $S^{-1}(\boldsymbol{q_2})$, $S^{-1}(\boldsymbol{q_3})$. We can then define vectors

$$\boldsymbol{d_u} = \alpha_{\boldsymbol{u}} S^{-1}(\boldsymbol{q_1}) + \beta_{\boldsymbol{u}} S^{-1}(\boldsymbol{q_2}) + \gamma_{\boldsymbol{u}} S^{-1}(\boldsymbol{q_3}) - \hat{\boldsymbol{u}}$$

$$\boldsymbol{d_v} = \alpha_{\boldsymbol{v}} S^{-1}(\boldsymbol{q_1}) + \beta_{\boldsymbol{v}} S^{-1}(\boldsymbol{q_2}) + \gamma_{\boldsymbol{v}} S^{-1}(\boldsymbol{q_3}) - \hat{\boldsymbol{v}}.$$

Our optimization problem is then to minimize the least squares functional

$$\sum_f \|\boldsymbol{d_u}\|^2 + \|\boldsymbol{d_v}\|^2.$$

The minimum of this function is unique up to a translation, so we add the further constraint that the initial point which seeded the patch remains at the origin. Solving this optimization problem involves solving a sparse linear system $\boldsymbol{Ax} = \boldsymbol{b}$ where $\boldsymbol{A}$ has four rows for every triangle, one for each of the two coordinates of $\boldsymbol{u}$ and $\boldsymbol{v}$ and a final row for the barycentric coordinates of our seed point. $\boldsymbol{x}$ is the vector of texture coordinates of the vertices in the patch, and $\boldsymbol{b}$ is a vector of ones and zeros expressing the texture coordinate directions.

4.4.3 Texture Synthesis

Due to discontinuities, surface stretching/compression, or blurring of the surface signal, the distorted pattern of colors generated by the semi-Lagrangian mapping typically will not be a good match to the original texture pattern. To force the surface colors to more closely match the input example texture, we employ an optimization-based texture synthesis method based on the one presented by Kwatra *et al.* [2005].

Throughout the optimization we store three types of color for each vertex of each surface patch (see Figure 4.7): the *current colors*, the *advected colors*, and the *best-match colors*. The *current colors* refer to the colors currently stored on the mesh—a blend of all the best-match colors of all the patches overlapping a given vertex. The current colors represent the current state of the optimization and change with each optimization step. When optimization is complete the current colors will define the final texture on the surface. The *advected colors* refer to the distorted colors generated through the semi-Lagrangian mapping. The advected colors are used to initialize the current colors, but remain constant

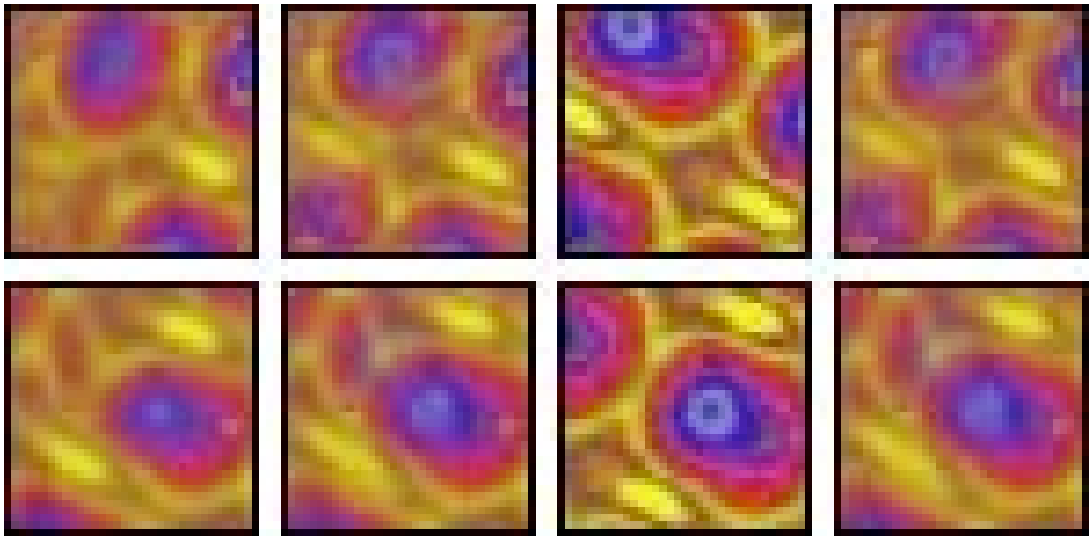

Figure 4.7: This figure shows a planar view of one step of texture optimization for two surface patches. The leftmost images are the distorted advected colors generated through the semi-Lagrangian mapping. Second from the left are the current colors at the beginning of the optimization step. Third from the left are the best-match colors, the region from the example texture which most closely matches the advected and current colors. On the right are the current colors at the end of the step. Since this step is late in the optimization process the current colors at the end of the step are indistinguishable from the current colors at the beginning of the step.

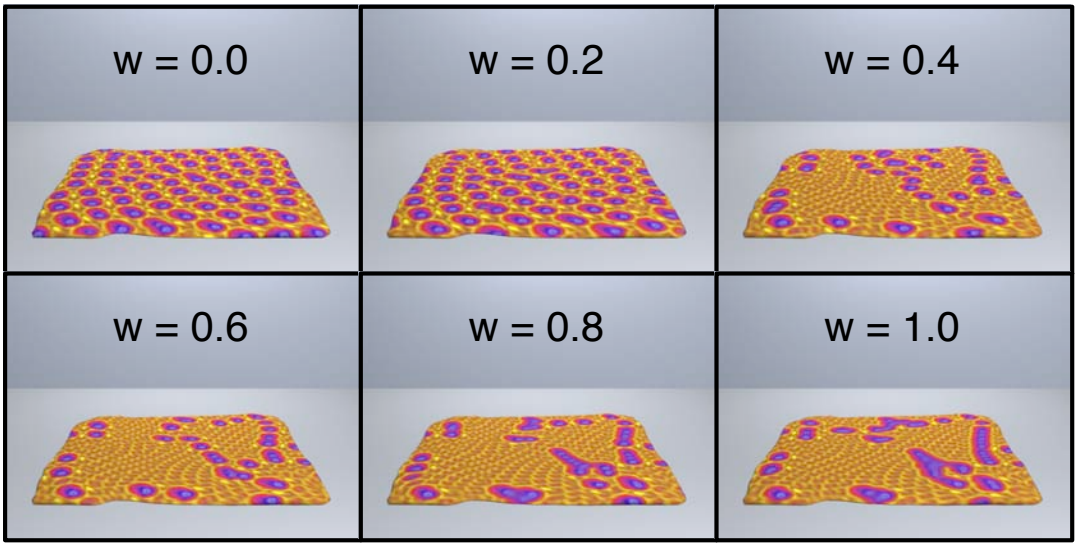

Figure 4.8: The parameter, w, allows us to trade off temporal coherence and matching the input texture. This figure shows the last frame of an animation similar to those in Figure 5.11 for a variety of w values. For low values of w, the final texture more closely matches the input texture. As w increases there is more temporal coherence between the frames, but the optimization is less able to match the details of the example texture.

during optimization. The *best-match colors* are the colors, chosen from the input example texture, that most closely match the current and advected colors in a given patch.

Each iteration of the optimization process comprises four steps for each patch:

1. Map the current and advected colors from the surface to texture space.
2. Find the best-match to the current colors and advected colors in the example texture.
3. Map these best-match colors to the surface.
4. Update the current colors on the surface.

Step (1) uses the parameterization described in Section 4.4.2 to map the current and advected colors from the surface to texture space. Of course, since the advected colors do not change during optimization, these can be mapped to texture space once and cached. Step (2) finds the best match in the input example texture to both the advected and current colors by finding the region in the input example texture which minimizes the energy function

$$E(\boldsymbol{c}, \boldsymbol{a}, \boldsymbol{b}) = \sum_{i,j} g(i,j) \left\| (1-w)(c_{ij} - b_{ij}) + w(a_{ij} - b_{ij}) \right\|^2,$$

where $\boldsymbol{c}$ are the current colors for the patch (mapped to texture space), $\boldsymbol{a}$ are the advected colors, $\boldsymbol{b}$ are the best-match colors (which is the variable we are minimizing over), i and j vary over the two-dimensional texture region, $g(\cdot)$ is a Gaussian weighting function which ensures that colors near the center of the patch have more weight, and w is a weighting parameter that trades off temporal coherence and matching the example texture (see Figure 4.8). In our implementation, the best-match colors are found through a brute-force search over all regions of the example texture. More sophisticated search techniques do exist [Wei and Levoy, 2000; Kwatra et al., 2005] and could provide significant speedups. Step (3) maps these best-match colors from texture space to the surface mesh. Finally, step (4),

removes the contribution of the previous best-match colors from the current colors stored at the mesh vertices and blends in the new best-match colors. Following Kwatra *et al.* [2005], we perform the optimization at several mesh resolutions and for several patch sizes at each resolution, though for most of the examples in this thesis we found one resolution with patch sizes of 33x33 pixels and 17x17 pixels to be sufficient. Several optimization steps for one frame can be seen in Figure 4.9.

This optimization approach is particularly appealing in our context. In many parts of the surface that have experienced minimal distortion, the advected colors may quite closely match the example texture. Consequently, the optimization makes only minor changes and converges quite quickly. Additionally, we achieve temporal coherence by initializing the optimization with the advected colors and including a term in the energy function which attempts to match these advected colors. This temporal coherence is demonstrated in Figure 5.10.

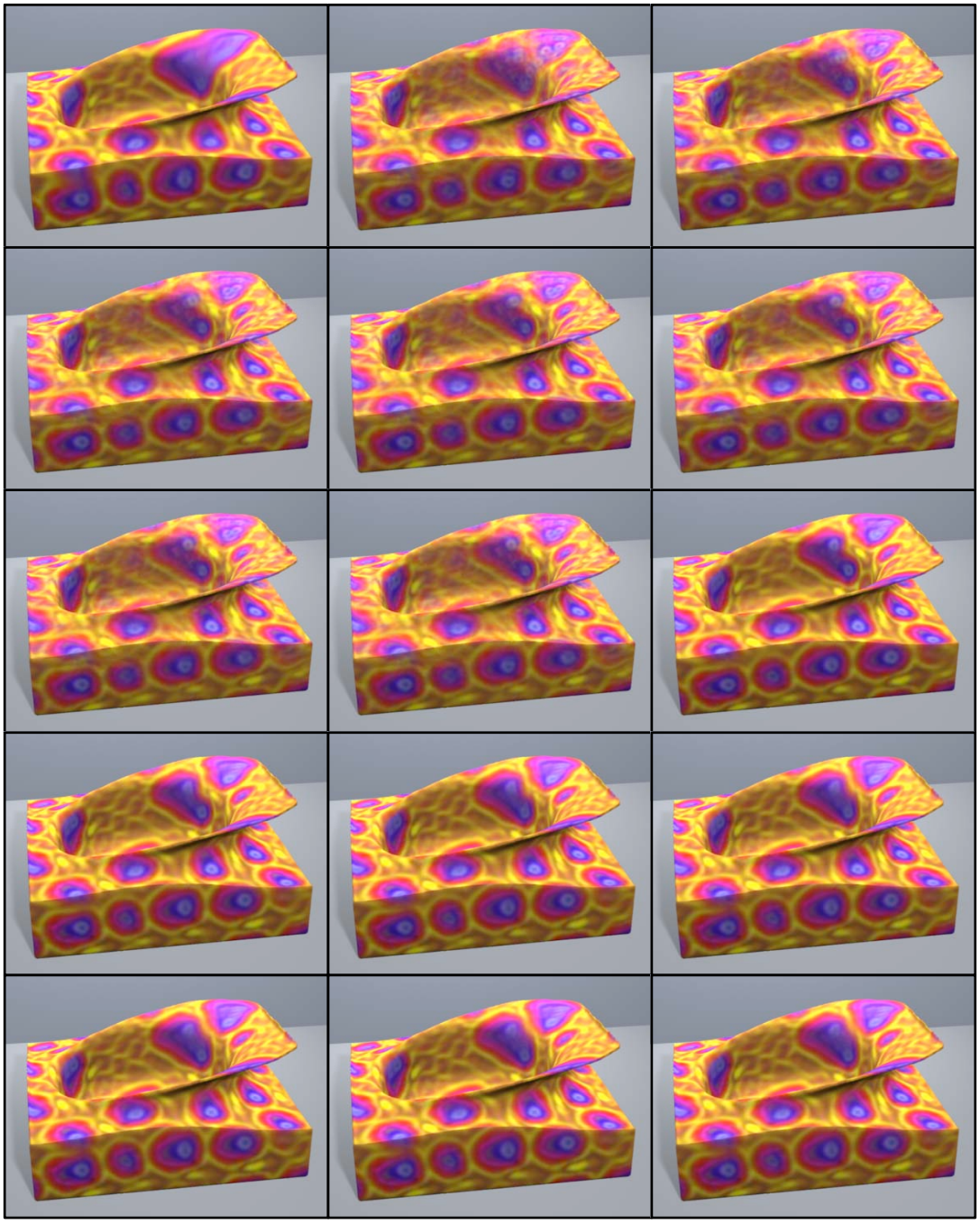

Figure 4.9: These images show the progress of the texture optimization. The image in the upper left hand corner is the initialization for the optimization (i.e. the *advected colors*). The next image is after the first optimization step, and so on. In the last image of the third row, the optimizer begins working with smaller patches and the bottom right corner shows the final output texture for this frame.

Chapter 5

Results and Discussion

5.1 Surface Tracking

We have tested our semi-Lagrangian contouring surface tracking method coupled with a fluid simulation on several examples such as the ones shown in Figures 5.1 and 5.2. We have also tested it in the spiraling analytical test field proposed by LeVeque [1996] and used by Enright *et al.* [2002a] to test their particle level-set method:

$$\begin{aligned} \boldsymbol{v_t}(x,y,z) =& (2\sin(\pi t)\sin^2(\pi x)\sin(2\pi y)\sin(2\pi z), \\ & -\sin(\pi t)\sin(2\pi x)\sin^2(\pi y)\sin(2\pi z), \\ & -\sin(\pi t)\sin(2\pi x)\sin(2\pi y)\sin^2(\pi z)) \end{aligned} \tag{5.1}$$

Figure 5.3 shows two objects being advected in this divergence-free velocity field to a mid-point after which the field reverses. The sphere of radius 0.15, centered at $(0.35, 0.35, 0.35)$ was restored to a nearly identical shape (see Figure 5.4), while the bunny exhibited a small amount of smoothing. The surface of the bunny was textured by a spot-generating reaction-diffusion system [Turk, 1991; Witkin and Kass, 1991] that ran on the surface as the object

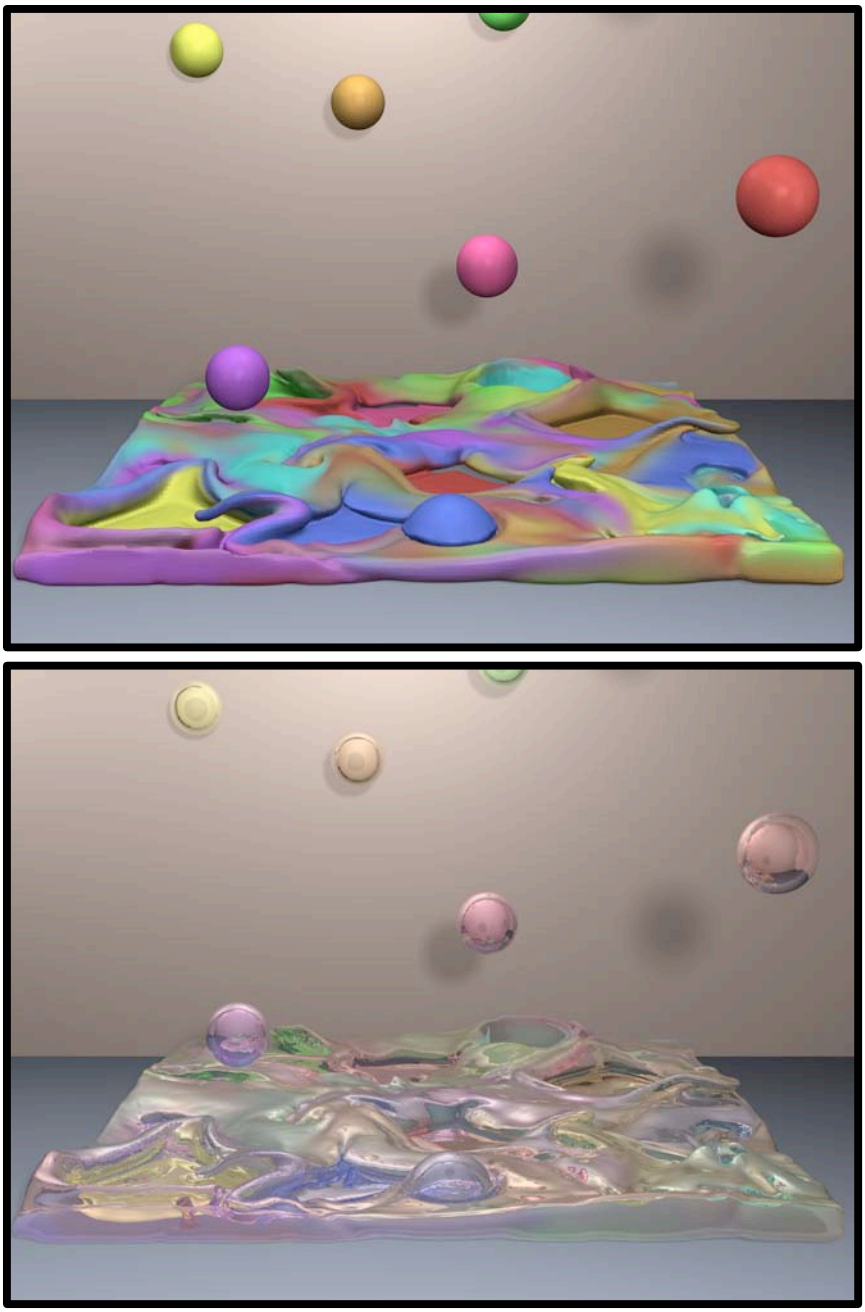

Figure 5.1: These images shows an invisible tank being filled as multicolored balls of fluid fall into it. The resulting surface contains complex geometric details which retain the different colors of the balls. The top image was rendered with a matte shader, while the bottom image was rendered with a colored glass shader.

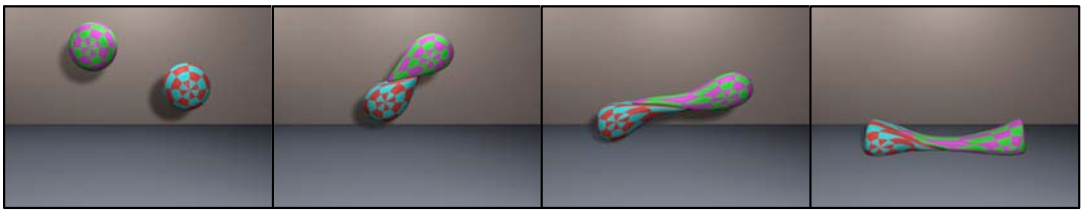

Figure 5.2: Two balls of visco-elastic fluid are thrown at each other and merge.

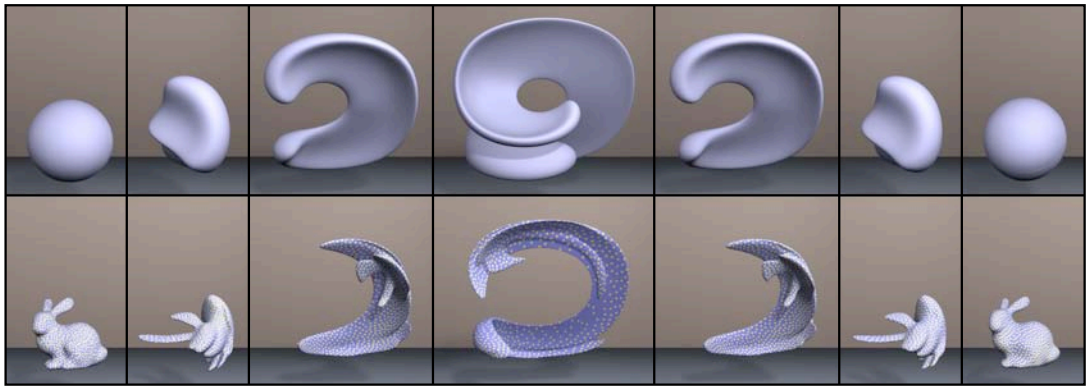

Figure 5.3: This figure shows the behavior generated when two different surfaces are passed through an analytical flow field that stretches and distorts them. The first three images show the object flowing along the field, the last three show the behavior when the distorted object then flows back along the reverse field. The bunny is textured using a reaction-diffusion system that is running on the surface during the sequence.

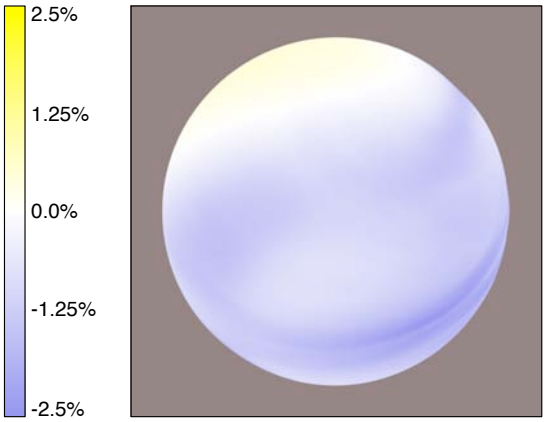

Figure 5.4: This figure shows the error in the final frame of the sphere example in Figure 5.3. The color maps to the error as a percentage of the sphere's radius, with blue points slightly inside and yellow points slightly outside. Over most of the surface the error is quite small, though a small crease has formed where the sphere has undergone substantial distortion.

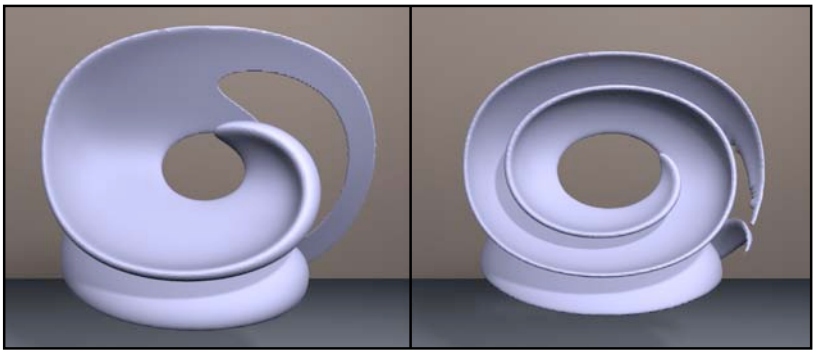

Figure 5.5: In this figure we show the result generated when we continue to distort and stretch an object past the point where it thins out and tears.

was being advected. At each timestep, the morphogens were advected along with the surface and then allowed to react. In Figure 5.5 we show the result of running the sphere through the flow field for several revolutions to highlight the behavior generated when the surface thins below the resolution of the octree's finest level.

All of our fluid examples used a standard regular-grid Eulerian fluid simulator. More specifically, we use the staggered-grid data structure of Foster and Metaxas [1996], the semi-Lagrangian advection method introduced by Stam [1999], the extrapolation boundary condition of Enright *et al.* [2002b], and the viscoelasticity model of Goktekin *et al.* [2004]. The fluid simulator and the surface tracking module were only very loosely coupled: the fluid simulator provided the surface tracker with a velocity function and, in turn, the surface tracker provided the simulator with the signed-distance function. Because our fluid simulator has a regular grid its resolution is notably coarser than the surface tracker, which uses an octree. The idea of using different resolutions for the fluid and surface is not new; Foster and Fedkiw [2001] used different timesteps for their fluid and surface calculations and Goktekin *et al.* [2004] found that increasing the spatial resolution of the surface tracking grid dramatically reduced volume loss. As noted by Losasso *et al.* [2004], using different spatial resolutions can produce artifacts. For example, pieces of surface could appear connected

when the simulator thinks they are disconnected and vice versa. Additionally, surface features may be maintained when a more detailed fluid simulator would smooth them away. In general, we found the increased surface resolution to be worth these artifacts. Ideally we would use a multiresolution fluid simulation, like the octree method of Losasso *et al.* [2004]. We plan to incorporate a multiresolution fluid simulator as part of our future work.

For most of our examples the surface tracking module took roughly 1 min/timestep at an effective resolution of 512^3. The fluid simulation also required about 1 min/timestep. Both the fluid simulator and the surface tracking module took 11 timesteps per frame. Thus it took about 2 days to simulate 10 s of animation, with roughly half the time spent solving for the velocity field and half the time updating the surface. It is important to note that, given a perfect semi-Lagrangian path tracer, the surface tracking method could take arbitrarily large timesteps. Decoupling the timesteps of the fluid simulator and surface tracker, so that the surface tracker runs only once per frame, is an interesting area of future work.

In Figure 5.6 we show the behavior when a thick viscoelastic fluid is allowed to flow off a shelf into a basin. This surface is textured by advecting reference coordinates along with the flow and applying a procedural checkerboard texture. Figure 5.7 shows beginning and ending frames using both an off-the-shelf procedural shader, which includes a displacement map, and a reaction-diffusion system. The motion of the spots on the surface occurs both from the motion of the surface and from the reaction-diffusion system seeking equilibrium on the moving surface. Thus, even after the fluid motion has mostly stopped the surface spots continue to move over the surface.

Figure 5.8 show two streams of liquid that are being sprayed toward each other. As the streams oscillate from side-to-side, they collide and produce a thin, web-like surface

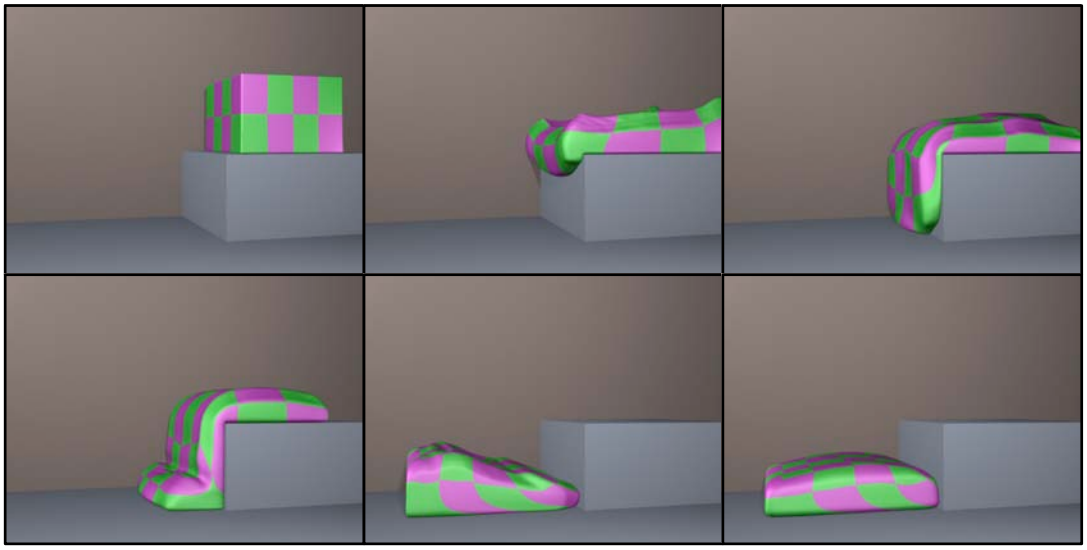

Figure 5.6: This sequence shows a thick viscoelastic fluid sliding off of a shelf. A checkerboard texture is mapped onto the surface. It is interesting to notice that the corners of the checkerboard texture stay sharp, despite the significant deformation. However, it is also evident that the surface deformation has caused the texture to distort significantly.

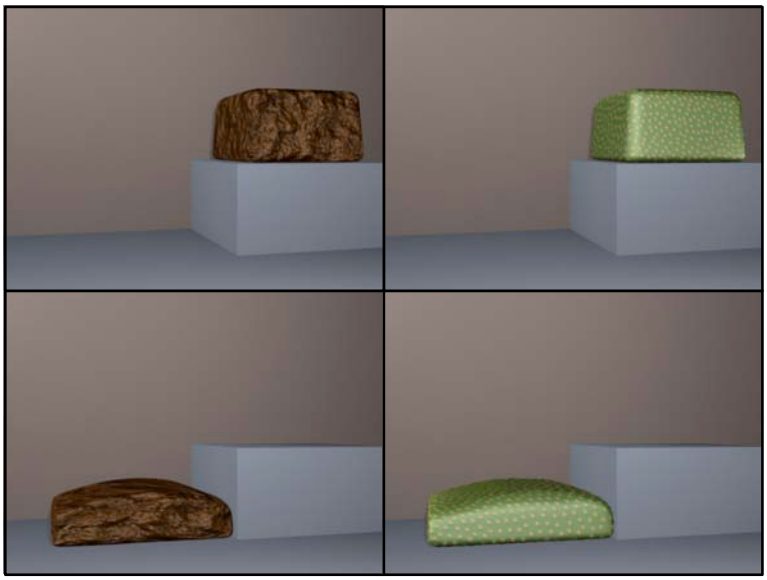

Figure 5.7: This figure shows the beginning and ending frames of an animation similar to that shown in Figure 5.6. The left images were rendered with an off-the-shelf procedural shader which includes a displacement map, while the images on the right were generated with a reaction-diffusion texture.

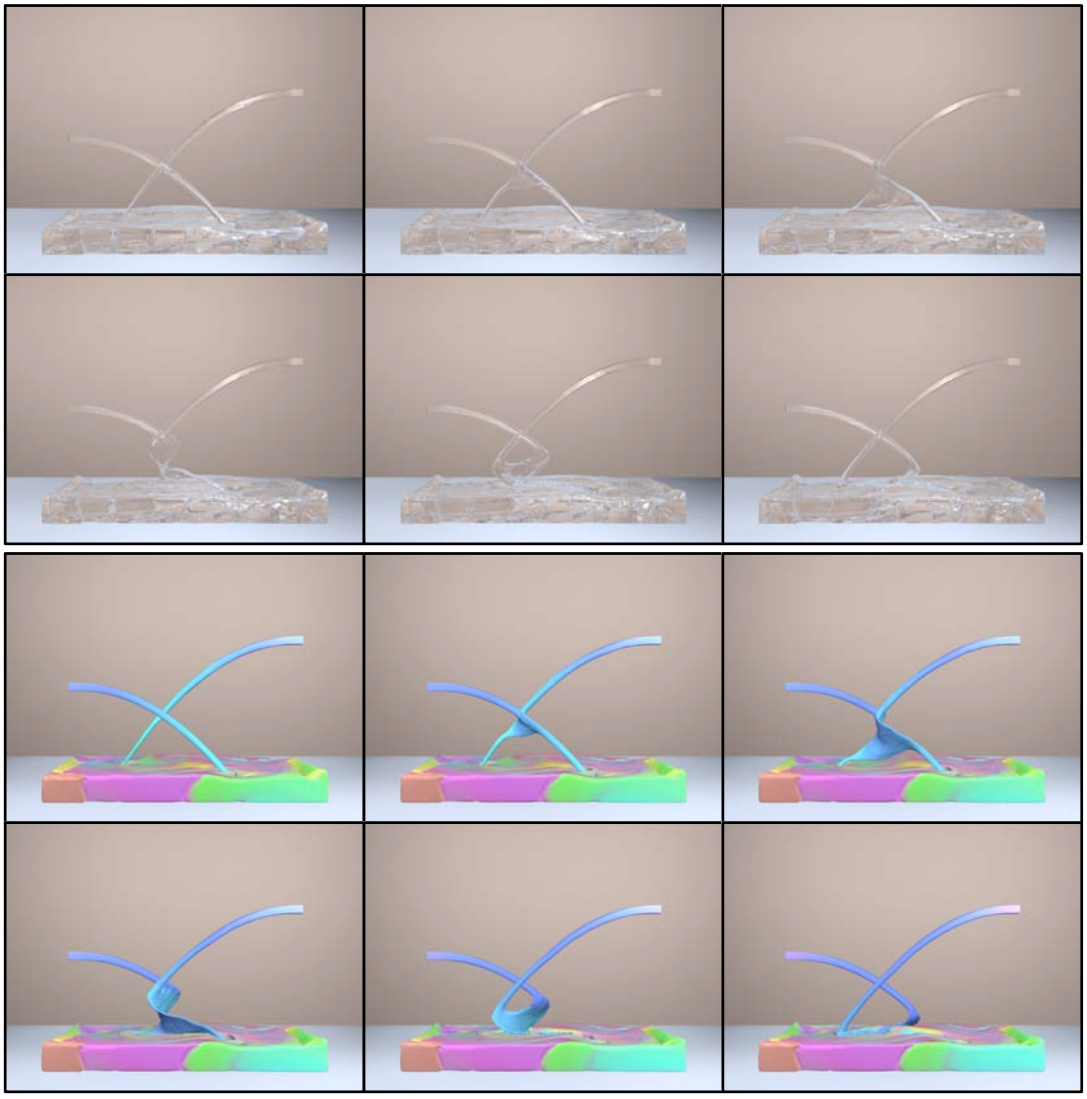

Figure 5.8: This sequence shows realistic (top) and matte (bottom) renderings of two liquid sprays. As the sprays move from side to side, they periodically intersect and create a web-like spiral pattern.

between them. The motion of the two streams causes this thin surface to form a spiral shape as the streams separate. Similar effects can be seen in real-world footage.

All of our images were rendered with the open-source renderer Pixie [Arikan, 2005]. Since we generated a polygonal mesh for each frame, we could take advantage of standard rendering techniques, allowing for very fast rendering times; most of our renderings took

less than 3 min/frame. Many of our examples were rendered with a matte shader so that the surface detail can be seen. A number of our examples were also rendered with a glass shader (using water's index of refraction) for comparison to previous methods and real fluids, and to demonstrate how the method can be used to generate realistic results. Our colored and textured examples illustrate how easily a variety of properties may be attached to the surface. In practice, we believe that advected properties could be used effectively with standard shading techniques to generate a wide range of interesting effects.

5.2 Surface Texturing

We have implemented the example-based texture synthesis method described in this thesis and demonstrated it with a variety of fluid motions and example textures. The fluid motions demonstrate significant squashing and stretching of the surface as well as a variety of topological changes. Our method generates surface textures which match the input example texture while remaining temporally coherent.

Figure 5.9 shows a comparison between three texturing approaches. Both the advected parameterization and advected colors approaches suffer from substantial distortion of the texture over time. The advected parameterization additionally contains discontinuities where topological changes have occurred. The advected colors handle these topological changes but suffer from significant blurring of the surface texture. In contrast, our optimization-based texture synthesis approach is able to closely match the example texture while maintaining temporal coherence. Figure 5.10 shows an animation of a splash created when a ball of fluid is thrown into a shallow pool of fluid. The resulting motion demonstrates significant stretching of the surface, but the surface texture does not become overly distorted and always provides a good match to the input example texture.

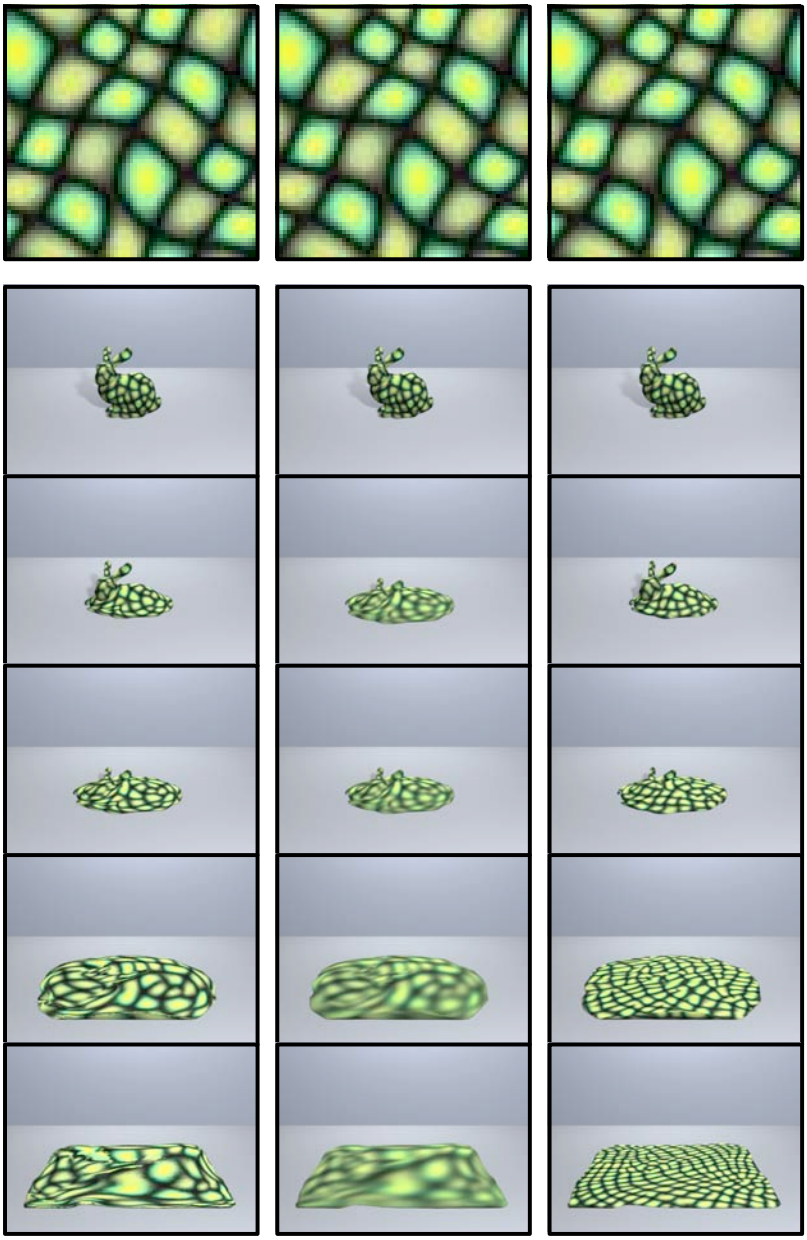

Figure 5.9: This figure shows three approaches to texturing liquid animations. The animation on the left was textured using an advected parameterization. This technique results in significant distortions of the texture and discontinuities where the surface has undergone topological changes. The middle column was textured with advected colors. Over time, the colors distort and blur. The right column was textured with our optimization-based technique. The texture avoids distortion while maintaining temporal coherence.

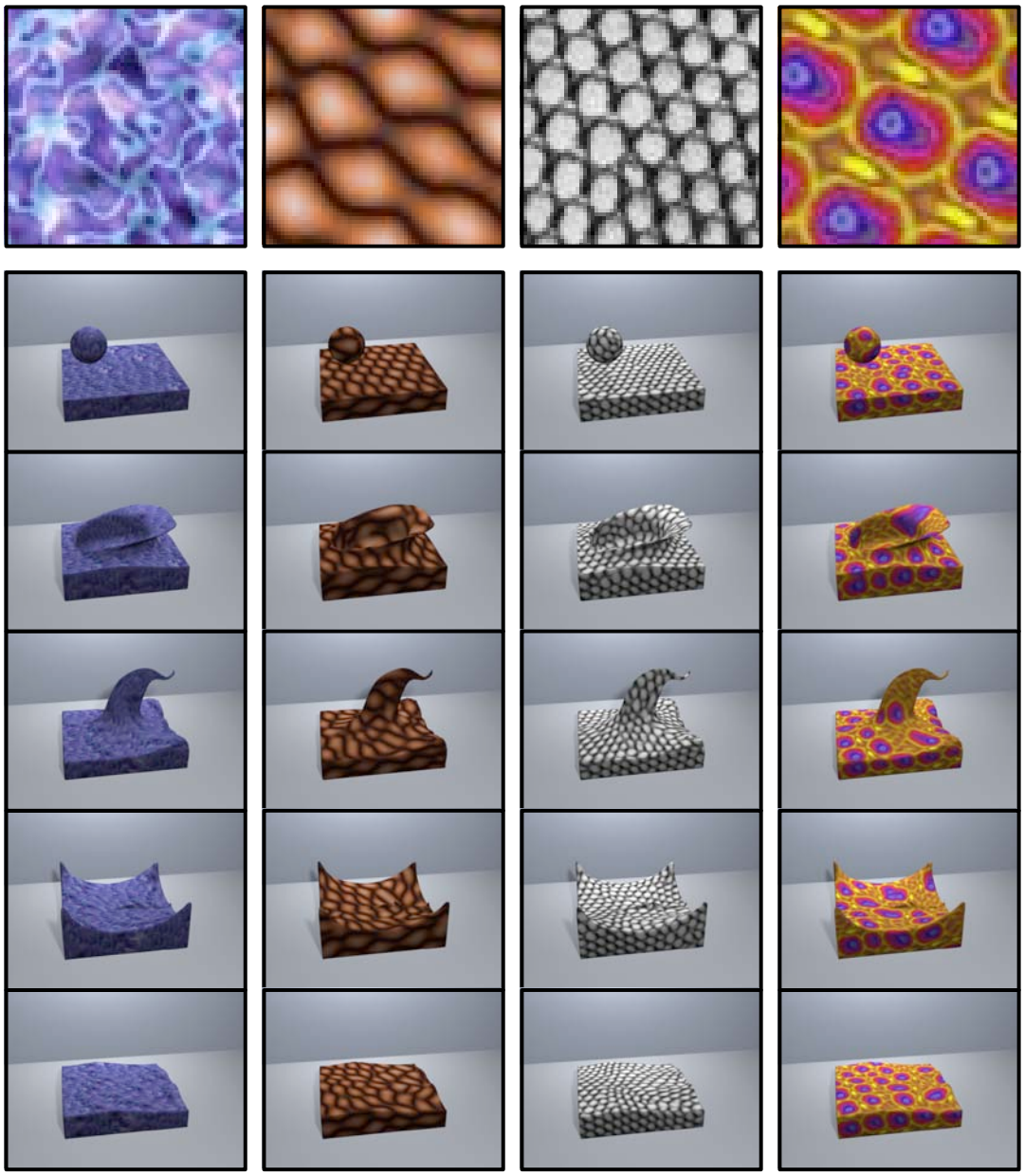

Figure 5.10: This splashing motion was textured using our example-based texture synthesis technique for liquid animations. Despite topological changes and significant surface distortions, the salient characteristics of the synthesized texture remain constant.

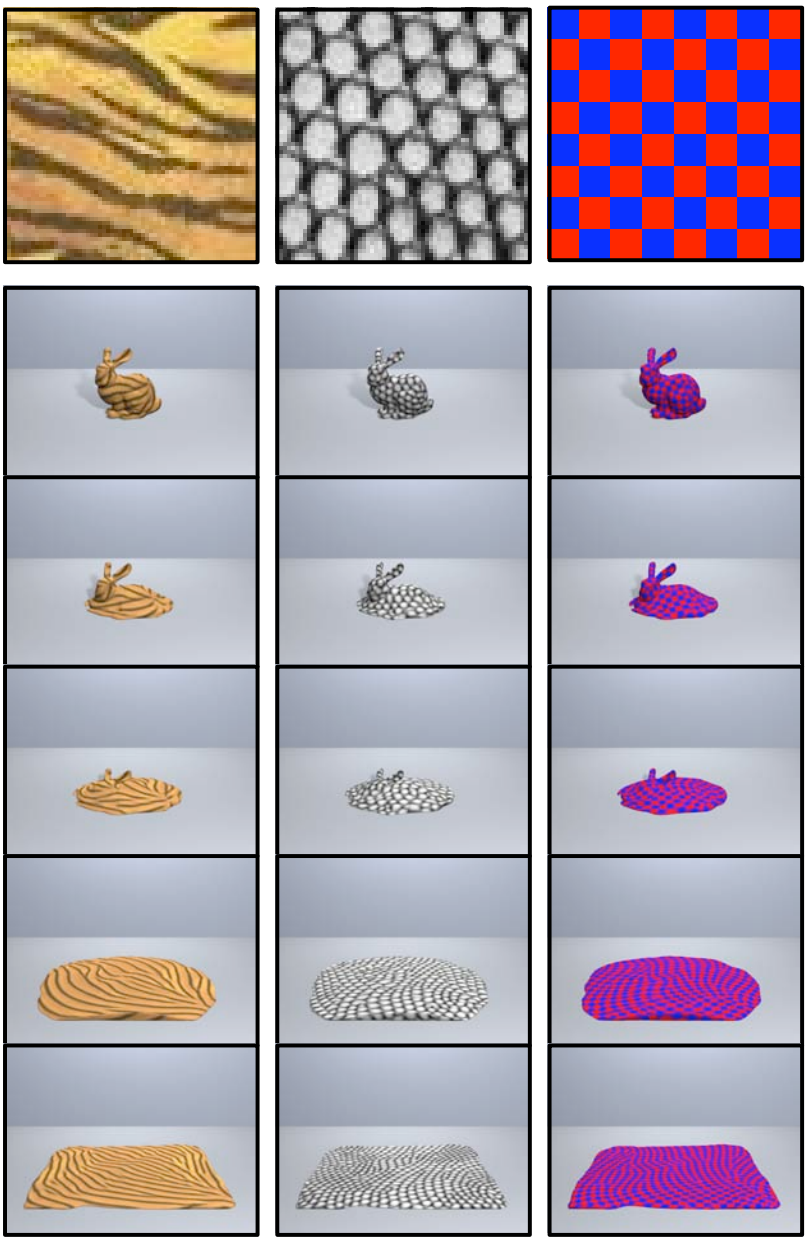

Figure 5.11: This figure shows several textures applied to a simulation of a melting bunny.

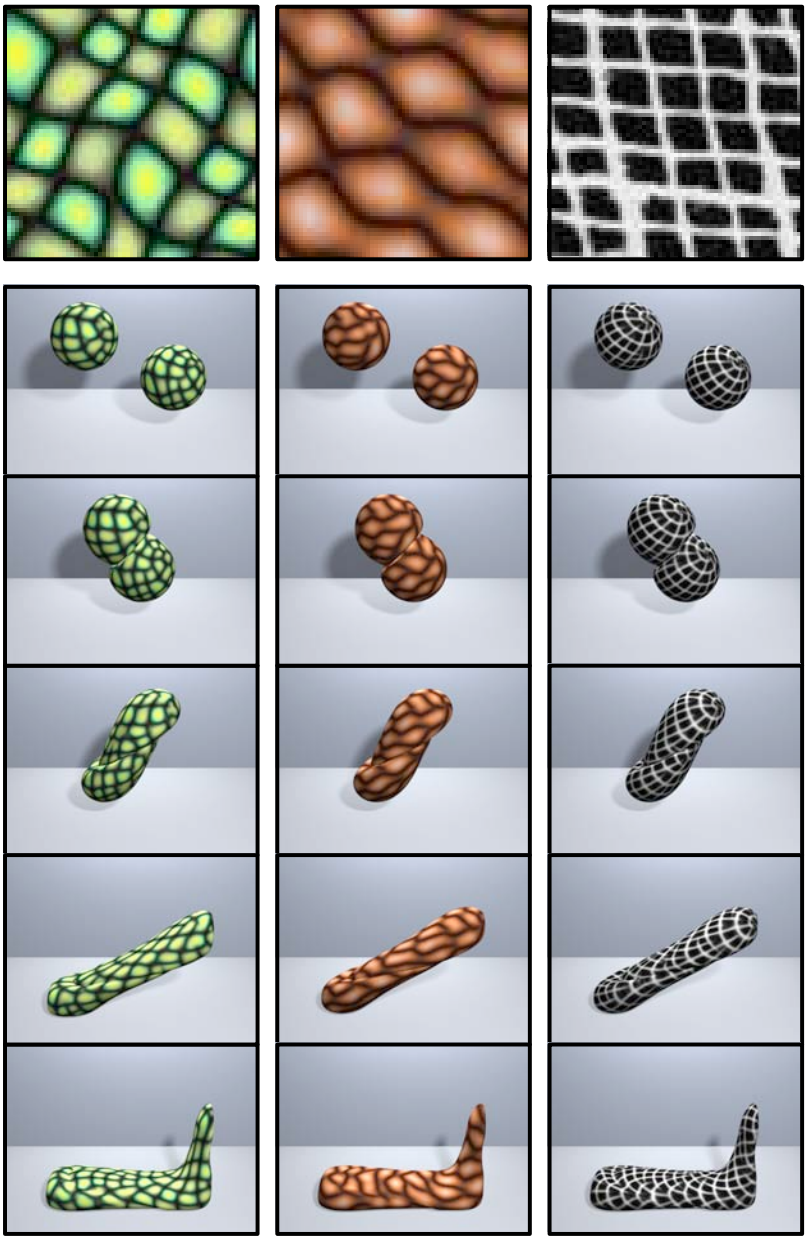

Figure 5.12: In this simulation, two balls of viscoelastic fluid are thrown at each other and merge. The texturing method handles this topological change without introducing any objectionable "pops."

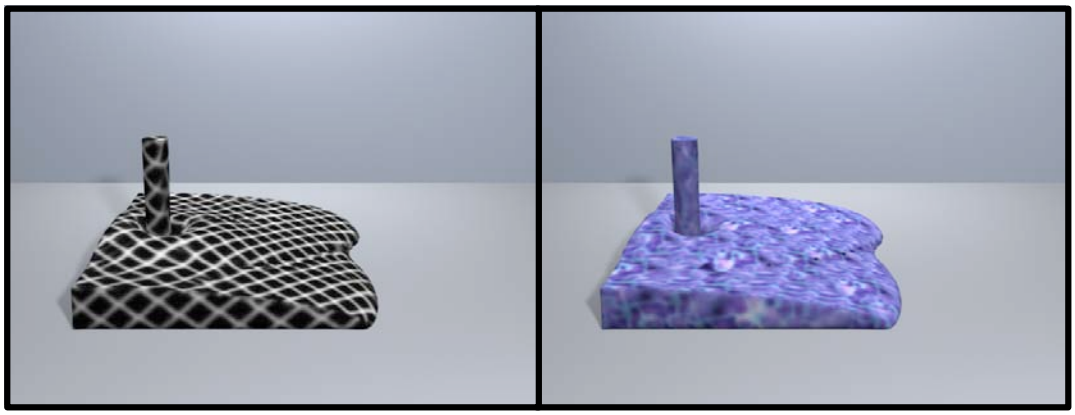

Figure 5.13: This figure shows two textures applied to an animation of a tank filling with viscous fluid.

Figure 5.11 shows an animation of a melting bunny with a checkerboard texture. Though the resulting texture is not a perfect checkerboard, which is impossible because the surface is not developable, locally the texture quite closely matches the checkerboard example, and globally the texture does resemble a checkerboard.

Figure 5.2 shows an animation of two balls of fluid thrown at each other. Because we explicitly include temporal coherence in the energy function there is no noticeable "pop" when the two spheres merge, rather they gradually move toward a continuous texture which matches the example. Figure 5.13 shows an additional example of viscous fluid filling a tank with two different textures. Figure 5.14 shows some examples with multiple textures. We took two approaches to handling multiple textures. The simpler approach initializes different objects with different textures. Then, for each optimization step, all the textures are searched when looking for the best match. Unfortunately, this approach fails when the similarity metric (in our case, sum of squared differences) is unable to distinguish between the various textures. In this case, we advect an additional scalar field on the surface which determines, individually for each patch, which example texture is searched for the best match.

Figure 5.14: This figure shows some examples with multiple textures. In the examples on the left, the objects were initialized with different textures. After initialization the optimization searched both textures for the best match. In the examples on the right, an additional scalar field was advected on the surface and used to determine which texture to search for the best match.

Unfortunately, our implementation is not particularly fast. Re-tiling a surface mesh takes about one minute, generating and optimizing the surface patches takes between fifteen and thirty minutes for a single frame, and the texture optimization takes between one and fifteen minutes per frame, depending on the amount of distortion of the texture and the number of vertices in the mesh. Fortunately, the re-tiling and patch generation can be done in parallel, so they do not create a significant bottleneck in a traditional rendering pipeline. Additionally, in this work we were more concerned with developing a method which produces high quality results rather than one optimized for speed. We believe the general method could be made much faster, perhaps using ideas developed by Magda and Kriegman [2003].

Concurrently with our work, Kwatra *et al.* [2006a; 2006b] have developed a very similar example-based texture synthesis method for fluids. However, they do not build and optimize patches as a precomputation, but rather construct color neighborhoods on the fly using the method presented by Turk [2001]. They also have a more developed texture synthesis module which uses K-means trees to find the best match in the example texture rather than our brute force approach.

Chapter 6

Conclusions

Semi-Lagrangian contouring offers an elegant and effective means for surface tracking and has a number of advantages over competing methods. First, we have an explicit representation. In addition to enabling exact evaluation, this explicit representation also allows us to leverage 30 years of computer graphics technology which has been optimized for polygonal meshes. Rendering, texture mapping, and a variety of other applications are all very straightforward. Second, we have an implicit representation. This implicit representation allows us to update the surface without explicitly addressing any of the difficult topological issues which plague other approaches. Third, semi-Lagrangian advection gives us a mapping between surfaces at adjacent timesteps. This mapping allows us to accurately track surface properties on the actual surface at negligible complexity and cost. Fourth, our method does not have any ad hoc rules or parameters to tune. In fact, the only parameters to our system are the upper and lower corners of the domain, the maximum depth of the octree (a resolution parameter), and some resolution tolerances. Finally, and most importantly, we are able to produce detailed, flicker-free animations of complex fluid motions.

Example-based texture synthesis methods are in many ways ideal for liquid surfaces because they are able to overcome the discontinuities and distortions of an advected parameterization while maintaining excellent temporal coherence. Our method is able to handle a wide variety of input example textures and should prove to be a useful tool for artists, complementing existing texturing techniques such as procedural texturing [Ebert et al., 2002], and advected texture maps. Of course, our example-based texturing method is not limited to the domain of liquid animations—the method requires only the ability to map colors from the surface at one frame to the surface at the next frame and could prove quite useful for texturing a variety of deforming surfaces. Furthermore, our methods are not limited to image textures, but could also be used for bump and displacement maps or any other method of stylizing surfaces or adding surface detail, as long as an example texture can be supplied.

Bibliography

[Arikan, 2005] Arikan, O., 2005. Pixie: Photorealistic renderer.

[Bærentzen and Aanæs, 2005] Bærentzen, J. A., and Aanæs, H. 2005. Signed distance computation using the angle weighted pseudonormal. *IEEE Transactions on Visualization and Computer Graphics 11*, 3, 243–253.

[Bærentzen and Christensen, 2002] Bærentzen, J. A., and Christensen, N. J. 2002. Interactive modelling of shapes using the level-set method. *International Journal of Shape Modelling 8*, 2, 79–97.

[Bennis et al., 1991] Bennis, C., Vézien, J.-M., and Iglésias, G. 1991. Piecewise surface flattening for non-distorted texture mapping. In *the Proceedings of ACM SIGGRAPH 1991*, 237–246.

[Blinn, 1982] Blinn, J. F. 1982. A generalization of algebraic surface drawing. *ACM Transactions on Graphics 1*, 3, 235–256.

[Bloomenthal, 1994] Bloomenthal, J. 1994. An implicit surface polygonizer. In *Graphics Gems IV*. Academic Press Professional, Inc., 324–349.

[Boissonnat and Oudot, 2003] Boissonnat, J. D., and Oudot, S. 2003. Provably good surface sampling and approximation. In *the Proceedings of the Eurographics/ACM SIGGRAPH Symposium on Geometry Processing 2003*, 9–18.

[Bonet, 1997] Bonet, J. S. D. 1997. Multiresolution sampling procedure for analysis and synthesis of texture images. In *the Proceedings of ACM SIGGRAPH 1997*, 361–368.

[Cani and Desbrun, 1997] Cani, M.-P., and Desbrun, M. 1997. Animation of deformable models using implicit surfaces. *IEEE Transactions on Visualization and Computer Graphics 3*, 1, 39–50.

[Carlson et al., 2002] Carlson, M., Mucha, P. J., R. Brooks Van Horn, I., and Turk, G. 2002. Melting and flowing. In *the Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation 2002*, 167–174.

[Carlson et al., 2004] Carlson, M., Mucha, P. J., and Turk, G. 2004. Rigid fluid: Animating the interplay between rigid bodies and fluid. *ACM Transactions on Graphics 23*, 3, 377–384.

[Cook and DeRose, 2005] Cook, R. L., and DeRose, T. 2005. Wavelet noise. *ACM Transactions on Graphics 24*, 3, 803–811.

[Courant et al., 1952] Courant, R., Isaacson, E., and Rees, M. 1952. On the solution of nonlinear hyperbolic differential equations by finite differences. *Communications on Pure and Applied Mathematics 5*, 243–249.

[Desbrun and Cani, 1996] Desbrun, M., and Cani, M.-P. 1996. Smoothed particles: A new paradigm for animating highly deformable bodies. In *Computer Animation and Simulation*, 61–76.

[Desbrun and Gascuel, 1995] Desbrun, M., and Gascuel, M.-P. 1995. Animating soft substances with implicit surfaces. In *the Proceedings of ACM SIGGRAPH 1995*, 287–290.

[Ebert et al., 2002] Ebert, D. S., Musgrave, K. F., Peachey, D., Perlin, K., and Worley, S. 2002. *Texturing & Modeling: A Procedural Approach*, third ed. Morgan Kaufmann.

[Efros and Freeman, 2001] Efros, A. A., and Freeman, W. T. 2001. Image quilting for texture synthesis and transfer. In *the Proceedings of ACM SIGGRAPH 2001*, 341–346.

[Efros and Leung, 1999] Efros, A. A., and Leung, T. K. 1999. Texture synthesis by non-parametric sampling. In *the Proceedings of the International Conference on Computer Vision 1999*, 1033–1038.

[Enright et al., 2002a] Enright, D., Fedkiw, R., Ferziger, J., and Mitchell, I. 2002. A hybrid particle level set method for improved interface capturing. *Journal of Computational Physics 183*, 1, 83–116.

[Enright et al., 2002b] Enright, D. P., Marschner, S. R., and Fedkiw, R. P. 2002. Animation and rendering of complex water surfaces. *ACM Transactions on Graphics 21*, 3, 736–744.

[Enright et al., 2005] Enright, D., Losasso, F., and Fedkiw, R. 2005. A fast and accurate semi-Lagrangian particle level set method. *Computers and Structures 83*, 479–490.

[Fedkiw et al., 2001] Fedkiw, R., Stam, J., and Jensen, H. W. 2001. Visual simulation of smoke. In *the Proceedings of ACM SIGGRAPH 2001*, 15–22.

[Foster and Fedkiw, 2001] Foster, N., and Fedkiw, R. 2001. Practical animation of liquids. In *the Proceedings of ACM SIGGRAPH 2001*, 23–30.

[Foster and Metaxas, 1996] Foster, N., and Metaxas, D. 1996. Realistic animation of liquids. In *the Proceedings of Graphics Interface 1996*, 204–212.

[Frisken et al., 2000] Frisken, S. F., Perry, R. N., Rockwood, A. P., and Jones, T. R. 2000. Adaptively sampled distance fields: a general representation of shape for computer graphics. In *the Proceedings of ACM SIGGRAPH 2000*, 249–254.

[Goktekin et al., 2004] Goktekin, T. G., Bargteil, A. W., and O'Brien, J. F. 2004. A method for animating viscoelastic fluids. *ACM Transactions on Graphics 23*, 3, 463–468.

[Green and Hatch, 1995] Green, D., and Hatch, D. 1995. Fast polygon-cube intersection testing. In *Graphics Gems V*. Academic Press Professional, Inc., 375–379.

[Guendelman et al., 2005] Guendelman, E., Selle, A., Losasso, F., and Fedkiw, R. 2005. Coupling water and smoke to thin deformable and rigid shells. *ACM Transactions on Graphics 24*, 3, 973–981.

[Heeger and Bergen, 1995] Heeger, D. J., and Bergen, J. R. 1995. Pyramid-based texture analysis/synthesis. In *the Proceedings of ACM SIGGRAPH 1995*, 229–238.

[Hieber and Koumoutsakos, 2005] Hieber, S. E., and Koumoutsakos, P. 2005. A lagrangian particle level set method. *Journal of Computational Physics 210*, 1, 342–367.

[Hilton et al., 1996] Hilton, A., Stoddart, A. J., Illingworth, J., and Windeatt, T. 1996. Marching triangles: Range image fusion for complex object modelling. In *International Conference on Image Processing*, 381–384.

[Hirt and Nichols, 1981] Hirt, C. W., and Nichols, B. D. 1981. Volume of fluid (vof) method for the dynamics of free boundaries. *Journal of Computational Physics 39*, 201–225.

[Hong and Kim, 2005] Hong, J.-M., and Kim, C.-H. 2005. Discontinuous fluids. *ACM Transactions on Graphics 24*, 3, 915–920.

[Houston et al., 2006] Houston, B., Nielsen, M. B., Batty, C., Nilsson, O., and Museth, K. 2006. Hierarchical RLE level set: A compact and versatile deformable surface representation. *ACM Transactions on Graphics 25*, 1, 151–175.

[Ju et al., 2002] Ju, T., Losasso, F., Schaefer, S., and Warren, J. 2002. Dual contouring of hermite data. *ACM Transactions on Graphics 21*, 3, 339–346.

[Kolluri, 2005] Kolluri, R. 2005. Provably good moving least squares. In *the Proceedings of the ACM-SIAM Symposium on Discrete Algorithms 2005*, 1008–1017.

[Kwatra et al., 2005] Kwatra, V., Essa, I., Bobick, A., and Kwatra, N. 2005. Texture optimization for example-based synthesis. *ACM Transactions on Graphics 24*, 3, 795–802.

[Kwatra et al., 2006a] Kwatra, V., Adalsteinsson, D., Kwatra, N., Carlson, M., and Lin, M. 2006. Texturing fluids. Tech. rep., University of North Carolina at Chapel Hill.

[Kwatra et al., 2006b] Kwatra, V., Adalsteinsson, D., Kwatra, N., Carlson, M., and Lin, M. 2006. Texturing fluids. In *the Proceedings of ACM SIGGRAPH 2006 Sketches & Applications*.

[LeVeque, 1990] LeVeque, R. J. 1990. *Numerical Methods for Conservation Laws*. Birkhauser-Verlag, Basel.

[LeVeque, 1996] LeVeque, R. J. 1996. High-resolution conservative algorithms for advection in incompressible flow. *SIAM Journal on Numericeal Analysis 33*, 2, 627–665.

[Lorensen and Cline, 1987] Lorensen, W. E., and Cline, H. E. 1987. Marching cubes: A high resolution 3d surface construction algorithm. In *the Proceedings of ACM SIGGRAPH 1987*, 163–169.

[Losasso et al., 2004] Losasso, F., Gibou, F., and Fedkiw, R. 2004. Simulating water and smoke with an octree data structure. *ACM Transactions on Graphics 23*, 3, 457–462.

[Magda and Kriegman, 2003] Magda, S., and Kriegman, D. 2003. Fast texture synthesis on arbitrary meshes. In *the Proceedings of the Eurographics workshop on Rendering 2003*, 82–89.

[Maillot et al., 1993] Maillot, J., Yahia, H., and Verroust, A. 1993. Interactive texture mapping. In *the Proceedings of ACM SIGGRAPH 1993*, 27–34.

[Meinhardt, 1982] Meinhardt, H. 1982. *Models of Biological Pattern Formation*. Academic Press, London.

[Müller et al., 2003] Müller, M., Charypar, D., and Gross, M. 2003. Particle-based fluid simulation for interactive applications. In *the Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation 2003*, 154–159.

[Müller et al., 2004] Müller, M., Keiser, R., Nealen, A., Pauly, M., Gross, M., and Alexa, M. 2004. Point based animation of elastic, plastic and melting objects. In *the Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation 2004*.

[Neyret, 2003] Neyret, F. 2003. Advected textures. In *the Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation 2003*, 147–153.

[Nielsen and Museth, 2006] Nielsen, M. B., and Museth, K. 2006. Dynamic tubular grid: An efficient data structure and and algorithms for high resolution level sets. *Journal of Scientific Computing 26*, 1, 1–39.

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

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

[Pauly et al., 2005] Pauly, M., Keiser, R., Adams, B., Dutré;, P., Gross, M., and Guibas, L. J. 2005. Meshless animation of fracturing solids. *ACM Transactions on Graphics 24*, 3, 957–964.

[Poston et al., 1998] Poston, T., Wong, T.-T., and Heng, P.-A. 1998. Multiresolution isosurface extraction with adaptive skeleton climbing. *Computer Graphics Forum 17*, 3, 137–148.

[Praun et al., 2000] Praun, E., Finkelstein, A., and Hoppe, H. 2000. Lapped textures. In *the Proceedings of ACM SIGGRAPH 2000*, 465–470.

[Premože et al., 2003] Premože, S., Tasdizen, T., Bigler, J., Lefohn, A., and Whitaker, R. 2003. Particle-based simulation of fluids. *Computer Graphics Forum 22*, 3, 401–410.

[Rasmussen et al., 2004] Rasmussen, N., Enright, D., Nguyen, D., Marino, S., Sumner, N., Geiger, W., Hoon, S., and Fedkiw, R. 2004. Directable photorealistic liquids. In *the Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation 2004*, 193–202.

[Samet, 1990] Samet, H. 1990. *The design and analysis of spatial data structures.* Addison-Wesley Longman Publishing Co., Inc.

[Sander et al., 2001] Sander, P. V., Snyder, J., Gortler, S. J., and Hoppe, H. 2001. Texture mapping progressive meshes. In *the Proceedings of ACM SIGGRAPH 2001*, 409–416.

[Schneider and Eberly, 2002] Schneider, P. J., and Eberly, D. H. 2002. *Geometric Tools for Computer Graphics*, first ed. Morgan Kaufmann, San Francisco.

[Sethian, 1996] Sethian, J. A. 1996. A fast marching level set method for monotonically advancing fronts. *Proc. of the National Academy of Sciences of the USA 93*, 4, 1591–1595.

[Sethian, 1999] Sethian, J. A. 1999. *Level Set Methods and Fast Marching Methods*, 2^{nd} ed. Cambridge Monograph on Applied and Computational Mathematics. Cambridge University Press, Cambridge, U.K.

[Shekhar et al., 1996] Shekhar, R., Fayyad, E., Yagel, R., and Cornhill, J. F. 1996. Octree-based decimation of marching cubes surfaces. In *the Proceedings of IEEE Visualization 1996*, 335–ff.

[Shu et al., 1995] Shu, R., Chen, Z., and Kankanhalli, M. S. 1995. Adaptive marching cubes. *The Visual Computer 11*, 4, 202–217.

[Sorkine et al., 2002] Sorkine, O., Cohen-Or, D., Goldenthal, R., and Lischinski, D. 2002. Bounded-distortion piecewise mesh parameterization. In *the Proceedings of IEEE Visualization 2002*, 355–362.

[Stam, 1999] Stam, J. 1999. Stable fluids. In *the Proceedings of ACM SIGGRAPH 1999*, 121–128.

[Stora et al., 1999] Stora, D., Agliati, P.-O., Cani, M.-P., Neyret, F., and Gascuel, J.-D. 1999. Animating lava flows. In *the Proceedings of Graphics Interface 1999*, 203–210.

[Strain, 1999a] Strain, J. A. 1999. Fast tree-based redistancing for level set computations. *Journal of Computational Physics 152*, 2, 648–666.

[Strain, 1999b] Strain, J. A. 1999. Semi-Lagrangian methods for level set equations. *Journal of Computational Physics 151*, 2, 498–533.

[Strain, 1999c] Strain, J. A. 1999. Tree methods for moving interfaces. *Journal of Computational Physics 151*, 2, 616–648.

[Strain, 2000] Strain, J. A. 2000. A fast modular semi-Lagrangian method for moving interfaces. *Journal of Computational Physics 161*, 2, 512–536.

[Strain, 2001] Strain, J. A. 2001. A fast semi-Lagrangian contouring method for moving interfaces. *Journal of Computational Physics 169*, 1, 1–22.

[Sussman and Puckett, 2000] Sussman, M., and Puckett, E. G. 2000. A coupled level set and volume-of-fluid method for computing 3d and axisymmetric incompressible two-phase flows. *Journal of Computational Physics 162*, 2, 301–337.

[Szeliski and Tonnesen, 1992] Szeliski, R., and Tonnesen, D. 1992. Surface modeling with oriented particle systems. In *the Proceedings of ACM SIGGRAPH 1992*, 185–194.

[Terzopoulos et al., 1989] Terzopoulos, D., Platt, J., and Fleischer, K. 1989. Heating and melting deformable models (from goop to glop). In *the Proceedings of Graphics Interface 1989*, 219–226.

[Turk, 1991] Turk, G. 1991. Generating textures on arbitrary surfaces using reaction-diffusion. In *the Proceedings of ACM SIGRAPH 1991*, 289–298.

[Turk, 1992a] Turk, G. 1992. Re-tiling polygonal surfaces. In *the Proceedings of ACM SIGGRAPH 1992*, 55–64.

[Turk, 1992b] Turk, G. 1992. *Texturing Surfaces Using Reaction-Diffusion.* PhD thesis, University of North Carolina, Chapel Hill.

[Turk, 2001] Turk, G. 2001. Texture synthesis on surfaces. In *the Proceedings of ACM SIGGRAPH 2001*, 347–354.

[Wang et al., 2005] Wang, H., Mucha, P. J., and Turk, G. 2005. Water drops on surfaces. *ACM Transactions on Graphics 24*, 3, 921–929.

[Wei and Levoy, 2000] Wei, L.-Y., and Levoy, M. 2000. Fast texture synthesis using tree-structured vector quantization. In *the Proceedings of ACM SIGGRAPH 2000*, 479–488.

[Wei and Levoy, 2001] Wei, L.-Y., and Levoy, M. 2001. Texture synthesis over arbitrary manifold surfaces. In *the Proceedings of ACM SIGGRAPH 2001*, 355–360.

[Wiebe and Houston, 2004] Wiebe, M., and Houston, B. 2004. The tar monster: Creating a character with fluid simulation. In *the Proceedings of ACM SIGGRAPH 2004 Sketches & Applications.*

[Witkin and Kass, 1991] Witkin, A., and Kass, M. 1991. Reaction-diffusion textures. In *the Proceedings of ACM SIGGRAPH 1991*, 299–308.

[Witting, 1999] Witting, P. 1999. Computational fluid dynamics in a traditional animation environment. In *the Proceedings of ACM SIGGRAPH 1999*, 129–136.

[Wyvill et al., 1986] Wyvill, G., McPheeters, C., and Wyvill, B. 1986. Data structure for *soft* objects. *The Visual Computer 2*, 4, 227–234.

[Zhu and Bridson, 2005] Zhu, Y., and Bridson, R. 2005. Animating sand as a fluid. *ACM Transactions on Graphics 24*, 3, 965–972.