Technical Report TR06-016 (June 2006)
Department of Computer Science, UNC Chapel Hill

Texturing Fluids

Vivek Kwatra[1] David Adalsteinsson[1] Nipun Kwatra[2] Mark Carlson[3] Ming Lin[1]

[1]University of North Carolina, Chapel Hill, NC
[2]Georgia Institute of Technology, Atlanta, GA
[3]DNA Productions, Irving, TX

Abstract

We present a novel technique for synthesizing textures over dynamically changing fluid surfaces. We transport texture information, i.e. color and local orientation, between fluid free surfaces from one time step to the next. This is accomplished by extending the texture information from the first fluid surface to the 3D fluid domain, advecting this information within the fluid domain using fluid velocity for one time step, and interpolating it back onto the second surface. We then refine the transported texture by performing texture synthesis over the second surface using our "surface texture optimization" algorithm, which keeps the synthesized texture visually similar to the input texture and temporally coherent with the transported one. We use both image textures as well as displacement textures as example inputs. Image textures can enhance rendering of the fluid by imparting novel realistic appearance to it; whereas displacement textures enable the generation of complex micro-structures on the surface of the fluid that may be very difficult to synthesize using simulation. We demonstrate our novel texture synthesis algorithm on dynamically moving fluid surfaces in several challenging scenarios.

1. Introduction

Realistic modeling, simulation, and rendering of fluid media have applications in various domains, including special effects, computer animation, electronic games, engineering visualization, medical simulation, and many more. Often the computational expense involved in simulating complex fluid phenomena limit the spatio-temporal resolution at which these simulations can be performed. This limitation makes it extremely difficult to synthesize complex fine-resolution micro-structures on the free surface of the fluid, that are usually present in many commonly occurring fluid phenomena. Examples of such micro-structures include small-scale waves in a river stream, foam and bubbles in turbulent water, patterns in lava flow, etc. Even with a highly robust and sophisticated fluid simulation system capable of modeling such structures, it is quite difficult to control the shape and appearance of these structures within the simulation. We explore an alternative approach which makes use of examples or samples of fluid shape and appearance to aid and complement the fluid simulation process.

An important aspect of synthesizing fluid animations is the rendering of the fluid. In order to render the desired appearance of the fluid, one needs to accurately model the material properties of the fluid. In general, it is non-trivial to model these properties, since they depend on multiple factors like density, temperature, viscosity, turbulence, etc. Additionally, these properties tend to vary across the surface of the fluid as well as over time. One way of circumventing these problems is to obtain example images of the fluid of interest and use them as texture to render the fluid. We explore such an approach in this paper. Textures have been extensively used to impart novel appearance to static surfaces, either by synthesizing texture over a plane and wrapping it over the surface, or by directly synthesizing the texture over surfaces. However, extending these techniques for texturing a surface that is dynamically changing over time is a non-trivial problem. This is so because one needs to maintain temporal coherence and spatial continuity of the texture over time, while making sure that the textural elements that compose the texture maintain their visual appearance even as the entire texture evolves. Such a general technique would also aid in creation of special effects with fluid phenomena, where the effect involves texturing a fluid surface with arbitrary textures. Additionally, it has potential applications in the visualization of vector fields and motion of deformable bodies, *e*.g., the velocity field of the fluid near its free surface is made apparent by the continuously evolving texture.

1.1. Main Results

In this paper, we present a novel texture synthesis algorithm for fluid flows. We assume that a fluid simulator is given, which is capable of generating free surfaces of the simulated fluid medium as well as providing a fluid velocity field. We develop a technique for performing texture synthesis on the free surface of the fluid by synthesizing texture colors on *points* placed on the surface. This is motivated by previous methods for synthesizing texture directly on surfaces [Tur01, WL01, YHBZ01]. However, these approaches typically grow the texture point-by-point over the surface. We extend and generalize the idea of texture optimization [KEBK05] to handle synthesis over 3D surfaces. Consequently, ours is a global technique, where the texture over the entire surface is evolved simultaneously across multiple iterations.

In order to maintain temporal coherence between moving surfaces in different time steps, we need to ensure that the texture synthesized on consecutive surfaces is similar to each other. We achieve this goal by first transporting the texture on the surface in the first time step to the next using the velocity field of the fluid that was responsible for the transport of the surface in the first place. The transported surface is then used as a *soft constraint* for the texture optimization algorithm when synthesizing texture over the second surface. The transport of texture across 3D surfaces is not as straightforward as advecting pixels using 2D flow fields in the planar case, since there is no obvious correspondence between points on the two surfaces. We establish the correspondence by first transferring texture information (color and *local orientation*) from the first surface onto a uniform 3D grid, followed by advection of texture information on this grid using the velocity field of the fluid. The advected texture is then interpolated back on the second surface to complete the transport.

Our approach has the following characteristics:

- It can work with any fluid simulator that provides 3D fluid velocity fields and fluid free surfaces at each iteration.
- It can take image textures, displacement textures, as well as alpha maps as input.
- It performs texture synthesis on *dynamically moving 3D surfaces*, as opposed to just 2D flow fields.
- It can handle significant topological changes in the simulated fluids, including merge and separation of multiple fluid volumes.
- It preserves the *visual similarity*[†] of the synthesized texture to the input texture, even while advecting both scalar (e.g. color) and vector quantities (local orientations over the free surface) relevant to the texture, to maintain temporal coherence with respect to the motion of the 3D fluid.

Our technique for advection of vector quantities through a velocity field is a novel contribution which may be useful in other applications as well. We demonstrate our algorithm using a variety of textures on several scenes, including a broken dam, a river scene, and lava flow, as shown in Figures 5 – 7.

1.2. Organization

The rest of the paper is organized as follows. In Section 2, we briefly summarize the related work in relevant areas. In Section 3, we present an overview of our approach. We describe the pre-computation required to construct the necessary data structures in Section 4 and our generalized texture optimization technique on 3D surfaces in Section 5. We then explain how we maintain temporal coherence of the resulting texture sequence by transporting texture information between successive time steps in Section 6. We show the results of our system in Section 7. Finally, we conclude with some possible future research directions.

2. Related Work

In this section, we briefly summarize recent work on texture synthesis and flows on surfaces. For a comprehensive overview of techniques for procedural texturing and modeling, we refer the readers to [EMP*98]. Below we describe recent work on texture synthesis using image textures as input.

2.1. Texture and Video Synthesis

Texture synthesis has been widely investigated in computer graphics. Various approaches are known, including pixel-based methods [DeB97, EL99, WL00], patch-based techniques [EF01, LLX*01, CSHD03, KSE*03, WY04], and global synthesis [HB95, PS00, PL98, FJP02, JFK03, WSI04]. The existing work focuses mostly on synthesis over static 2D planar or 3D surfaces. While some of the global methods (e.g. Markov Random Fields) could be considered for our problem, they are fully discrete optimizations, which are less flexible and suitable than semi-discrete-continuous techniques (e.g. [KEBK05]).

[†] Visual similarity refers to the spatial continuity and the resemblance of visual appearance between the input and synthesized textures. We measure visual similarity in terms of the similarity between local neighborhoods of the input and the output, which is quantified by our cost function in performing surface texture optimization.

Kwatra et al. [KEBK05] recently introduced a new technique for 2D texture synthesis based on iterative optimization. They also demonstrate how the same technique can be used for flow-guided texture animation, where a planar flow field is used to guide the motion of texture elements in a synthesized 2D texture sequence. We solve the fluid texturing problem by adapting ideas from the texture optimization technique to perform texture synthesis directly on a dynamically changing triangular surface in 3D – the motion of the surface being guided by a 3D fluid simulation as opposed to a planar flow field.

Other researchers have also extended texture synthesis techniques to video editing. Schödl et al. [SSSE00] concatenating multiple videos to synthesize a longer sequence. Doretto and Soatto [DS03] edit videos of complex fluid motions by modeling them as Linear Dynamical Systems. Graph cut algorithms have also been used to create novel images and video sequences by minimizing errors along the seams. Bhat et al. [BSHK04] presented a flow-based video synthesis technique by enforcing temporal continuity along a set of user-specific flow lines. While this method focus on stationary flow fields with focuses on video editing, our algorithm is applicable to any time-varying *dynamic* flow fields generated by fluid simulators and use image textures as input. In addition, we use the simulated flow fields as a mechanism to automatically control and guide constrained texture synthesis, while theirs requires user input to specify the flow lines to edit the video sequences.

2.2. Flows on Arbitrary Surfaces

Fluid simulations and various related natural phenomena have received much recent attention. Foster and Metaxas [FM96] were among the first to present the use of the full 3D Navier-Stokes differential equations for generating fluid animations. The "stable fluids" method of Stam [Sta99] introduced stable semi-Lagrangian advection combined with an implicit viscosity solver to arrive at a completely stable method, more amenable to use in animation. Level set methods [FSJ01, FF01, EMF02, LGF04, CMT04] and other meshfree methods [SF95, DG96, MCG03, Liu02, LL03] have also been proposed to simulate fluid dynamics. We refer the readers for more detailed survey in [LY05, SSK05]. Specialized techniques for synthesizing detailed fluid phenomena like drops, bubbles, and foam etc, directly through simulation, have also been developed [HK03, TFK*, WMT05].

Neyret [Ney03] proposed a method of stylizing fluid flows using stochastic textures that avoids a variety of visual artifacts, and demonstrated interesting 2D animations produced by advection. Recently Stam [Sta03], Shi and Yu [SY04] have proposed methods to simulate Navier-Stokes flows on 2D meshes. Stam's method requires the surface to be a regular quadrilateral mesh, while Shi and Yu's technique works on any triangulated mesh. Both focused on the goal of generating plausible 2D flows on surfaces embedded in 3D space.

In contrast, we present techniques for performing *texture synthesis* on dynamically moving 3D surfaces. Our approach can alleviate common artifacts that occur in simple passive advection of texture coordinates and color as detailed in [Ney03].

Bargteil et al. [BGOS06] present a semi-Lagrangian surface tracking method for tracking surface characteristics of a fluid, such as color or texture coordinates. However, they do not perform texture synthesis on the fluid free surface. In a similar manner, our work also relies on the fluid surface transport to advect color and textures. In addition to these scalar quantities, we also track orientation vectors over the fluid free surface through the velocity field. These vectors are tracked to ensure that the synthesized texture has consistent orientation across (temporally) nearby free surfaces.

Finally, there has been work in the scientific visualization community that makes use of texture for visualization and representation of vector fields [TA03] as well as shape [GIS03]. We observe that, in a similar spirit, our technique can also be used for visualization of surface velocity fields as well as deformable bodies, using *arbitrary* textures.

3. Overview

We couple controllable texture synthesis with fluid simulation to perform spatio-temporally coherent fluid texturing. The main elements of our system include (i) a fluid simulator for generating the dynamic surface with velocity information, (ii) a technique for performing texture synthesis on the fluid surface coherent with neighboring surfaces, and (iii) a method for transporting texture information from one surface to the other. Figure 1 shows a flow chart of how these three components interact with each other for fluid texturing.

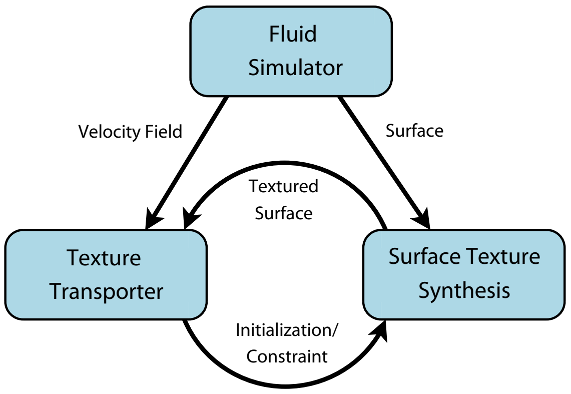

Figure 1: *An overview of our fluid texture synthesis system.*

The only requirements for a fluid simulator to work with our system are that it should be able to output the 3D fluid velocity field at each iteration, and that the surfaces generated during each iteration should be a consequence of transporting the surface at the previous iteration through the fluid velocity field over a single time step. In our simulator, the surfaces are generated as the level set of an advected distance function.

We start by obtaining the free surface of the fluid for the first time step and then texture this surface using our surface texture optimization algorithm (explained in Section 5). We then transport the texture to the fluid free surface for the second time step using our texture transport technique. The transported quantities include the texture colors (and any other associated properties like alpha mask or displacement values) as well as local orientation vectors that are needed to synthesize the texture on the 3D surface.

This transported texture serves two purposes. Firstly, it acts as an initialization for the texture on the second surface. Secondly, it is treated as a *soft constraint* which specifies that the synthesized texture on the second surface should stay as close as possible to this initialized texture. Our surface texture optimization technique can naturally handle this constraint by plugging it into a texture cost function. These two operations of transport and synthesis are then repeated for each time step of the simulation.

4. Surface Preparation

To perform texture synthesis on a 3D surface, one needs to take into account the fact that there is no regular grid of pixels available as is the case in an image. Hence, we represent the texture on a surface by assigning color values to *points* placed on the surface. These points serve the same purpose on a surface as pixels do in an image. However, in the case of an image, pixels lie on a uniform grid. On the other hand, it is impossible to specify a single uniform grid of points on an arbitrary surface. Even so, we want the points to be as uniformly spaced as possible to ensure uniform texture sampling on the surface.

Before we begin synthesis, we prepare the surface for synthesis by having the following constructs in place. Firstly, we place the desired number of points on the surface in a way that they sample the surface uniformly. We then connect these points to form an auxiliary triangle mesh that aids in interpolation between points and neighborhood construction. A smooth vector field is then computed over this mesh that defines the local orientation (2D coordinate system) at each point. This orientation field is used to map points in a neighborhood onto a plane. This mapping is then used for comparing a surface patch against a pixel neighborhood in the input image. These operations are now described in more detail.

4.1. Point Placement

As discussed above, we store texture information in points placed on the surface being textured. These points will store the texture color at the location represented by the point as well as the local orientation over the surface at that point. Hence, the number of points will determine the *resolution* of the synthesized texture. The more the number of points, the higher the resolution. For example, a surface with $10,000$ points will be equivalent to an image of size 100×100 pixels. An important thing to note is that the resolution of the surface changes from frame to frame. This happens because, we want the inter-point distance on the surface to roughly stay equivalent throughout the sequence. Hence, if the area of the surface increases, the points also increases in number proportionally and vice-versa. The starting number of points is a user-defined parameter, but it is computed automatically for subsequent frames. We want the points to be spaced as uniformly as possible over the surface so that the synthesized texture also has a uniform quality throughout. A consequence of this need for uniformity (and the non-constant nature of the number of points over time) is that the points for each free surface (in time) are generated independently. Fortunately, our grid-based texture color and orientation advection techniques obviate the need to track the points anyways.

Note that the points being generated *sit* on the surface; hence, even though the points are in 3D, the distance between two points is not the straight line distance between them. Instead, we need to march along the surface to move from one point to another. This is where the local surface orientation is required in order to define displacements when moving on the surface (Section 5.1 explains this step in more detail). It also requires us to maintain neighborhood/connectivity information at each point for interpolating orientations (and later on, color and point locations). We maintain this information by building an auxiliary mesh hierarchy, that connects these points to form a set of triangulated meshes. The hierarchy is obtained by using different number of points to represent different resolutions which is helpful in multi-resolution texture synthesis.

We start by generating the points hierarchically. We follow the same procedure as adopted by Turk [Tur01]. We first place n points at the coarsest level of the hierarchy. Then, at the next finer level, we add $3n$ more points, also keeping the n points from coarser level, and repeat this procedure up the hierarchy. At each level, we initialize the points by placing them randomly over surface mesh. We then use the surface-restricted point repulsion procedure of Turk [Tur91] to achieve uniform spacing between these points. Note that the n points from the coarser level are copied over and stay fixed when applying point repulsion at a finer level.

4.2. Mesh Hierarchy

Once we have placed the points, we connect them to generate a triangulated mesh for each level of the hierarchy. We use the mesh re-tiling procedure of [Tur92] for re-triangulating the original surface mesh using the new points. This time, we start at the finest level of the hierarchy and work our way downward. We use the Triangle library [She96] for triangulation at each intermediate step. Each point in the hierarchy maintains a reference to its corresponding point in the coarser level (if one exists). For synthesizing texture on a stand-alone surface, we use three levels of hierarchy. However, when texturing a fluid sequence, only the first frame requires three levels. Subsequent frames only use one level of hierarchy because a good starting texture is easily obtained by advecting the texture from the previous frame.

4.3. Local Orientation

The next step is the computation of a local orientation at each point placed on the surface. We want these orientations to vary smoothly over each mesh in the hierarchy. We use a polar space representation of the orientation field, as proposed by Zhang et al. [ZMT04]. Firstly, a polar map is computed for each point on the mesh. A polar map linearly transforms angles defined between vectors on the surface of the mesh to angles between vectors in a 2D polar space. The transformation is simply $\phi = \theta \times 2\pi/\Theta$, where θ is the angle in mesh space, Θ is the total face angle around a point, and ϕ is the polar space angle. An orientation at each point is now defined as an angle in polar space. This representation allows us to easily smooth the orientation field by diffusion across adjacent points: for two points on the mesh connected by an edge, we want them to make the same polar angle with the common edge between them. Thus, each diffusion operation averages the current orientation angle of a point with the angles determined through the points connected to it. In a mesh hierarchy, this diffusion is performed at the coarsest level first and then propagated up the hierarchy. Note that an orientation angle can be converted into a 3D orientation vector by first applying the reverse transformation (of the one described above) to obtain a mesh space angle. We then compute the final orientation at a point by rotating a pre-defined *reference* vector stored at that point by this mesh space angle. This reference vector sits in the tangent plane of the point, *i*.e., lies perpendicular to the *normal* at the point, and is designated as having a *zero* polar angle.

5. Surface Texture Synthesis

Once we have constructed the requisite data elements, we are ready to perform synthesis. In this section, we will describe the two essential steps in performing surface texture synthesis: neighborhood construction and surface texture optimization.

5.1. Neighborhood Construction

Texture synthesis operations primarily require matching input and output pixel neighborhoods with each other. As we discussed in the previous section, a uniform grid of pixels is not always available on the surface. All we have is a unstructured mesh of points. Therefore, we need to match unstructured point neighborhoods against gridded pixel neighborhoods.

In earlier approaches for surface texture synthesis, this problem is solved by either pre-computing a mapping from the surface to a plane using texture atlases [YHBZ01] or by constructing pixel neighborhoods over the surface by marching over it on the fly [Tur01, WL01]. Pre-computing the mapping from surface to plane gives nicer neighborhoods, but is tedious to compute, especially for a sequence of surfaces. We favor a marching based approach because of its simplicity and scalability in handling a sequence of meshes. However, marching can lead to inconsistencies among adjacent neighborhoods due to curved nature of the surface and non-uniform spacing between points. Note that previous approaches use point-by-point synthesis algorithms where each point is synthesized one at a time: the neighborhood around each point is constructed once to determine its color and then discarded before moving on the next point. Our synthesis technique, on the other hand, uses a global optimization algorithm that requires construction of all neighborhoods on the mesh simultaneously. Consequently, inconsistencies between neighborhoods are of greater concern for us.

Our neighborhood construction alleviates this inconsistency problem by working with points directly, instead of computing exact pixel locations on the surface. Recall that in Section 4, we described how a mesh hierarchy is constructed after distributing points over the free surface. We will concentrate on a single mesh of the hierarchy. A point neighborhood in a mesh is defined as a set of points connected to each other and lying within a certain *distance* to a central point (also called center). This distance is measured in the local orientation space of the mesh and is *not* the same as the Euclidean distance between these points.

Treating the center as the origin, we first define a 2D location for each point in its neighborhood. Given the point p as the center, the 2D location of a point in its neighborhood is computed as its displacement from p along the orientation field on the mesh. For a given *pixel* neighborhood width w, we include only those points in p's neighborhood whose displacement vector (d_1, d_2), measured starting at p, is such that $-w/2 < d_1, d_2 < w/2$. To compute displacements of points in the neighborhood of p, we first consider all points in the 1-ring of p, *i*.e., points that are directly connected to p. If $\mathbf{o}$ represents the orientation vector at p and $\mathbf{t}$ represent its orthonormal complement, then the displacement vector of a point q in the 1-ring of p is $(\mathbf{v} \cdot \mathbf{o}, \mathbf{v} \cdot \mathbf{t})$, where $\mathbf{v} = q - p$. Moving outwards, we next consider points in the 2-ring of p, *i*.e., points that are in the 1-ring of immediate neighbors of p, but have not been considered already. To compute the displacement of a point r in the 1-ring of q (which in turn is in the 1-ring of p), we first compute the displacement of r with respect to q and then add to it the displacement of q with respect to p. We keep moving outwards in this manner, until we exhaust all points that satisfy the criteria of being part of the neighborhood around p.

We eventually want to match this point neighborhood against the pixel neighborhoods of the input image. We use the 2D displacement vectors of the points in the neighborhood to *flatten* the neighborhood, with the displacement vectors acting as the points' 2D coordinates on the flattened plane, which may be real-valued. When comparing this flattened point neighborhood against a pixel neighborhood, we first construct a new pixel neighborhood by interpolating color values from the real-valued point locations to integer-valued pixel locations, and then compare this newly constructed neighborhood against the pixel neighborhoods from the image. Figure 3 shows a schematic of the flattening process.

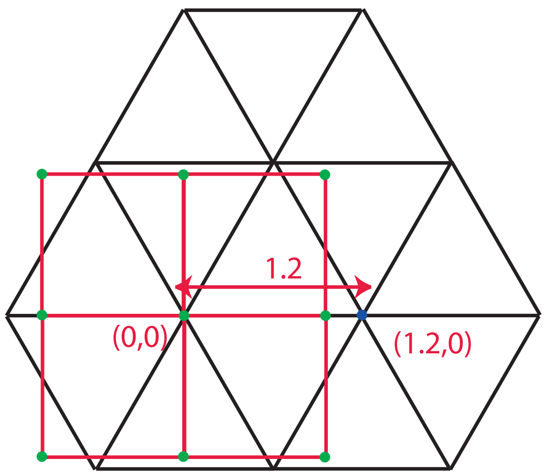

Figure 2: *Centers of point neighborhoods are shifted from origin because they may not perfectly align with pixel boundaries. Here, the right (blue) pixel is 0.2 units away from the pixel boundary. It will be assigned a shift of (0.2,0) when a neighborhood is constructed around it.*

As we will describe below, our synthesis algorithm constructs multiple neighborhoods simultaneously using multiple points as centers. That has a potential pitfall due to the fact that the input texture is available only as a discrete sampled image, *i*.e., we do not have access to a continuous texture function. This can cause problems because until now, we have decided to treat the central point of the neighborhood as the origin of the neighborhood, *i*.e., it is assigned the displacement vector $(0,0)$. Consider two nearby points that are both centers for different neighborhoods. For the sake of clarity, lets assume that this point neighborhood is flat (as shown in Figure 2). The figure shows that the the two central points are not an integer displacement apart from each other. This means that the *pixel* grid will not align with one of the centers if the other center is used as origin. Consequently, there will always be disagreement between the pixel neighborhoods that we would want to match with these point neighborhoods. We alleviate this problem by pre-computing a *shift* for each point that is used as the center of a point neighborhood. This shift implies that instead of treating the point as the origin of the pixel neighborhood, we consider it to be slightly shifted from the origin. We determine this shift by first computing displacements as we did for points inside a neighborhood above and then taking the fractional part of this displacement. For example, in Figure 2, the shift is $(0.2,0)$. Of course, there may always be disagreement between some points due to the curved nature of a surface. Hence, we diffuse these shifts over the points of the mesh to mitigate the problem as much as possible.

5.2. Surface Texture Optimization

Our approach for synthesizing texture on a 3D surface is motivated by the texture optimization algorithm proposed by Kwatra et al. [KEBK05]. Their formulation is valid only on a 2D plane. We extend that formulation to handle synthesis on a surface. The reason for using such an optimization algorithm for synthesis is that we want it to naturally handle synthesis on a dynamic surface, maintaining temporal coherence between consecutive surfaces and spatial coherence with the texture for each individual surface. We can incorporate the transported texture from the previous surface as a soft constraint with the optimization approach.

The optimization proceeds by minimizing an energy function that determines the quality of the synthesized texture with respect to the input texture example as well as the transported texture from previous surface. We first consider the energy with respect to just the input example. This energy is defined in terms of the similarity between local neighborhoods of the textured output surface and input texture example. To compare output point and input pixel neighborhoods, we first sample colors from the pixel neighborhood at real-valued locations corresponding to the point displacements (computed earlier as described above). We then define the texture energy for a single output neighborhood to be the sum of squared differences between the colors of output points and the input colors at the sampled locations. The total energy of the output surface is equal to the sum of energies over individual neighborhoods of the surface. If S denotes the output surface being textured and Z denotes the input texture sample, then the texture energy over S is defined as

$$E_t(\mathbf{s};\{\mathbf{z}_p\}) = \sum_{p\in S^\dagger} \|\mathbf{s}_p - \mathbf{z}_p\|^2. \quad (1)$$

Here $\mathbf{s}$ is the vectorized set of color values for all points on the mesh. $\mathbf{s}_p$ is the set of colors associated with points in a neighborhood around the point p. $\mathbf{z}_p$ contains colors sampled from a pixel neighborhood in Z, at 2D locations corresponding to the point neighborhood around p. The pixel neighborhood from which $\mathbf{z}_p$ is sampled is the one that appears most similar to $\mathbf{s}_p$ under the sum of squared differences. For searching this neighborhood in Z , we apply the opposite sampling operation to the point neighborhood: we construct a query pixel neighborhood by interpolating colors from the point neighborhood at integer pixel locations and then search for the input neighborhood closest to this query neighborhood. The set of points, $S^\dagger$, around which neighborhoods are constructed is a subset of the set of all points in S. $S^\dagger$ is chosen such that there is a significant overlap between all neighborhoods, *i*.e., a single point occurs within multiple neighborhoods. Figure 3 shows how point neighborhoods are flattened and matched against pixel neighborhoods.

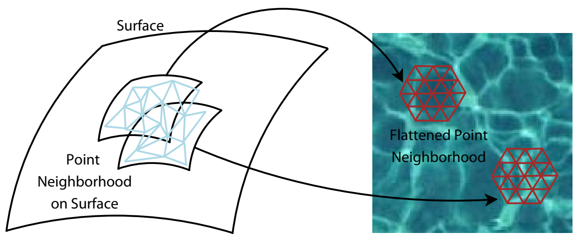

Figure 3: *Point neighborhoods on the surface are mapped onto a 2D plane in order to compare against pixel neighborhoods.*

The energy function defined above measures the spatial consistency of the synthesized surface texture with the input texture example. To ensure that the synthesized texture is temporally coherent as well, we add another term which measures the proximity of the output texture with the texture transported from the previous time step. The transported texture already specifies a color value vector, denoted as **c**, corresponding to each point location in S. We treat **c** as a soft constraint on **s**, *i*.e., we want the color of the transported texture and synthesized texture to be close to each other. The corresponding energy function is

$$E_c(\mathbf{s};\mathbf{c}) = \sum_{k \in S} \lambda_k (\mathbf{s}(k) - \mathbf{c}(k))^2, \qquad (2)$$

where k iterates over all points on the surface and λ_k is a weighting factor that controls the influence of transported texture color at a point over its synthesized value. We typically use a gradient based weighting scheme, where a larger weight is given to the transported texture color at a point that has greater color gradient in its vicinity. Note that we also use **c** as an initialization for **s**. The total energy of the textured fluid surface is

$$E(\mathbf{x}) = E_t(\mathbf{x};\{\mathbf{z}_p\}) + E_c(\mathbf{x};\mathbf{c}).$$

The algorithm proceeds by optimizing this total energy of the output surface in an iterative fashion. Given an initialization of the texture – random for the first frame, and transported from the previous surface for the remaining – the following steps are repeated until convergence:

1. For each surface point $p \in S^\dagger$, construct a point neighborhood in the vicinity of p from the current surface texture **s**, and sample it at integer pixel locations by interpolation (as described earlier). Lets call this constructed pixel neighborhood $\mathbf{x}_p$.
2. Find the closest input neighborhood for each output neighborhood $\mathbf{x}_p$ constructed above, and sample it at real-valued point locations to obtain the resampled neighborhood $\mathbf{z}_p$.
3. Re-compute the surface texture colors **s** by minimizing the total energy $E(\mathbf{s};\{\mathbf{z}_p\})$ with respect to **s**, keeping the set of input point neighborhoods $\{\mathbf{z}_p\}$ fixed. For the squared energy function used here, this corresponds to simply taking a weighted average between the colors coming from the different neighborhoods affecting a point, as well as the colors obtained from the transported texture.

Note that when performing synthesis in a multi-resolution fashion – this is done either for stand-alone surfaces or for the first frame of a fluid sequence – the optimization first proceeds at the coarsest level of the mesh hierarchy. This is followed by an up-sampling step, in which the finer level mesh vertices copy the color values from their coarser level counterparts, followed by diffusion of these color values at the finer level. This acts as the initialization for the finer mesh, after which optimization proceeds as earlier. Also, we typically use more than one neighborhood size at each level. In our experiments, we first run an optimization pass using 17×17 pixels as the neighborhood size followed by a pass with 9×9 pixels as the neighborhood size.

6. Texture Transport

We now describe the texture transport procedure which transports texture information from the currently synthesized surface to the next surface in the time sequence. The texture information being transported includes two quantities: (i) the texture color (or any other properties being synthesized, like displacement, alpha etc.) and (ii) the orientation field on the surface. While the texture color is what we really want to transport, it is necessary to transport the orientation field also because that determines how the texture neighborhoods are oriented on the surface when synthesis is performed. If the orientation fields on two consecutive surfaces are not consistent, it would be very diffult to find neighborhoods in the input texture that match the transported texture.

6.1. Color Advection

Our approach for transporting the texture color is based on standard advection techniques that have been well studied in the level set literature. The basic idea is to perform the transfer on a volumetric grid as opposed to directly between surfaces. This volumetric grid is the same as that used by the fluid simulation to represent the velocity field. One might be tempted to simply advect the texture colors from the first surface to the next. However, since the resolution of the grid is typically much smaller than the number of points used in synthesis, this will result in loss of texture resolution. Hence, we advect point coordinates instead of texture color because point coordinates can usually be safely interpolated without loss of resolution. The texture color is then obtained by interpolating from the closest point on the back-tracked point coordinate. The various steps are enumerated in order below:

1. The first step is to determine point coordinates at grid locations. This is relatively simple: the coordinates are the location of the grid point itself, since the surface and the grid sit in the same space.
2. Next, we treat each component of the coordinate field as a scalar field. Each scalar field is advected along the velocity field for a single time step. This step is done by solving the following advection update equation:

$$\frac{\partial \varphi}{\partial t} = -\mathbf{u} \cdot \nabla \varphi, \qquad (3)$$

   where φ is the scalar being advected and $\mathbf{u}$ is the velocity field obtained from the fluid simulation. We save the fluid velocity field at any intermediate time steps that the simulation may have generated and also use that while performing the advection update. The update is performed using a first order upwind scheme [Set98].
3. After the advection is complete, we have the advected coordinates at each grid point. These coordinates are interpolated onto the points on the new surface corresponding to the next time step. Each new surface point now knows which location it came from in the previous time step, through these back-tracked coordinates.
4. Finally, we assign a color (or other property) value to each new point as the color of the back-tracked point location on the previous surface. The color of the back tracked point location is obtained by finding the nearest point to that location on the previous surface and using its color value. To speed up this search process we use a hash table to store the points of the previous mesh.

6.2. Orientation Advection

The transport of orientations is not as straightforward as the transport of color. This is because orientations are vectors which may rotate (change orientation) as they move along the velocity field. If we simply advect each scalar components of the orientation vector independently, then it would only translate the vector. Figure 4 shows the difference in the result obtained by scalar vs. vector advection. Conceptually, to preform vector advection, we can advect the tail and head of the orientation vector separately through the field, and use the normalized vector from the advected tail to the advected head as the new orientation vector. This operation needs to be performed in the limit that the orientation is of infinitesimal length, *i*.e., the head tends to the tail. This results in a modified analytical advection equation of the form

$$\frac{\partial \mathbf{d}}{\partial t} = -\mathbf{u} \cdot \nabla \mathbf{d} + (\mathbf{I} - \mathbf{d}\mathbf{d}^{\mathbf{T}})\mathbf{d} \cdot \nabla \mathbf{u}. \qquad (4)$$

Here, the first term is the scalar advection term as applied to each component of the orientation vector $\mathbf{d}$. The term $\mathbf{d} \cdot \nabla \mathbf{u}$ computes the derivative of velocity $\mathbf{u}$ along the orientation $\mathbf{d}$ and $(\mathbf{I} - \mathbf{d}\mathbf{d}^{\mathbf{T}})$ is a projection operator which projects $\mathbf{d} \cdot \nabla \mathbf{u}$ to the plane perpendicular to $\mathbf{d}$. This projection essentially ensures that, in differential terms, $\mathbf{d}$ always stays a unit vector.

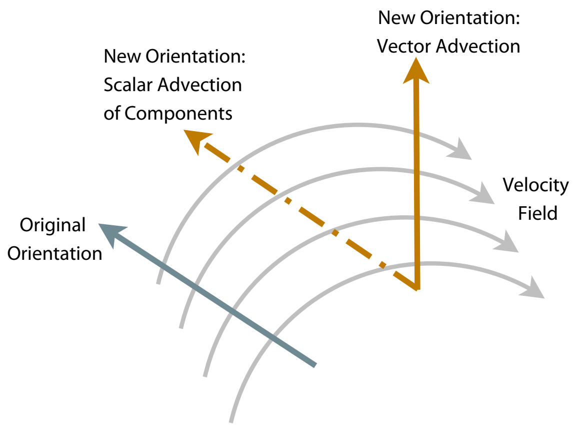

Figure 4: *Transporting an orientation vector by only advecting its scalar components will (incorrectly) only translate the orientation vector. Our vector advection technique, on the other hand, correctly transports the vector by translating as well as rotating it.*

To perform advection of $\mathbf{d}$, we follow similar steps as above with the following changes. In step 1, we *extend* the orientation field from the surface onto the grid using the technique of [AS99]. The orientation vectors are propagated outwards from the surface along the surface normal onto the grid. The advection update step 2 solves (4), again using upwind differencing, this time alternating between the two terms in the equation. For step 3, we directly interpolate the advected orientations on new surface points and normalize them. These vectors are then converted into the polar angle representation described in Section 4. We typically, rerun the diffusion algorithm to create a smooth orientation field.

7. Results

We have implemented the techniques presented here in C++ and rendered the results using POVRay and OpenGL. We use a grid-based method for fluid simulation [GDN90,Sta99, CMT04]. However, our technique is general enough to work with other types of simulators like finite volume based techniques [LC02]. We applied our implementation on several challenging scenarios with several different textures.

The first scene is a broken-dam simulation as shown in

Figure 5, where a large volume of water is dropped into a tank of water causing the invisible wall to break and the simulated water to splash. In the accompanied video, note that the texture on the water is split and merged naturally, as the water surfaces undergo substantial topological changes. The second simulation is a river scene as in Figure 6, where the water flows down the stream and carries the surface texture with it. We show the river with a displacement mapped texture in chocolate color. The third simulation is that of lava flow, shown in Figure 7 in which lava flows from the top of the mountain downwards onto the walls.

The computational complexity of our algorithm is dominated by nearest neighbor search during texture synthesis and mesh hierarchy construction which is used for generating points on the surface that contain texture color information. The mesh hierarchy is computed offline for a given simulation, so that runtime costs include only surface texture synthesis and texture transport. We accelerate the nearest neighbor search by using a hierarchical clustering of the image neighborhoods. The synthesis times depend on the simulation, because the number of points on the mesh change linearly with the area of the mesh as it evolves. Our average synthesis times are approximately 60 seconds per frame for the broken dam, 90 seconds per frame for the river scene, and 200 seconds per frame for the lava flow. The number of points used for storing texture pixels varied in the range of $100,000$ points and $500,000$ points over the course of the simulation.

The supplementary videos include the files for river, lava and broken dam examples. It also includes flowing displacement texture examples for the river and the broken dam. We also show a comparison between the results of our technique and the results obtained by using only texture transport (and no texture synthesis) at each frame.

8. Discussion & Future Work

We have presented a novel synthesis algorithm for advecting textures on dynamically changing fluid surfaces over time. Our work is perhaps the first successful demonstration of transporting textures along 3D fluid flows, that undergo complex topological changes between successive frames, while preserving visual similarity between the input and the output textures. We define visual similarity through a cost function that then drives a surface texture optimization process. We achieve fluid texturing by combining this novel surface texture optimization process with an advection scheme based on a level-set method for tracking the surface characteristics.

One can envision using our technique as a rendering mechanism in conjunction with fluid simulation techniques. Another interesting application is flow visualization using arbitrary textures. Our technique synthesizes texture sequences that animate the input texture as controlled by the fluid flows obtained from a fluid simulator. Hence, it can facilitate flow visualization using a rich variety of textures.

As for future research directions, we would like to extend our approach to incorporate 3D volume textures and to handle video textures. We would also like to explore other type of control mechanisms besides motion, in guiding texture synthesis.

References

[AS99] ADALSTEINSSON D., SETHIAN J.: The fast construction of extension velocities in level set methods. *Journal of Computational Physics 148* (1999), 2–22.

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

[BSHK04] BHAT K. S., SEITZ S. M., HODGINS J. K., KHOSLA P. K.: Flow-based video synthesis and editing. *ACM Transactions on Graphics (SIGGRAPH 2004) 23*, 3 (August 2004).

[CMT04] CARLSON M., MUCHA P., TURK G.: Rigid fluid: Animating the interplay between rigid bodies and fluid. In *Proc. of the ACM SIGGRAPH* (2004), ACM Press.

[CSHD03] COHEN M. F., SHADE J., HILLER S., DEUSSEN O.: Wang tiles for image and texture generation. *ACM Transactions on Graphics, SIGGRAPH 2003 22*, 3 (2003), 287–294.

[DeB97] DEBONET J. S.: Multiresolution sampling procedure for analysis and synthesis of texture images. *Proceedings of ACM SIGGRAPH 97* (August 1997), 361–368.

[DG96] DESBRUN M., GASCUEL M. P.: Smoothed particles: A new paradigm for animating highly deformable bodies. In *Computer Animation and Simulation '96 (Proceedings of EG Workshop on Animation and Simulation)* (Aug. 1996), Springer-Verlag, pp. 61–76.

[DS03] DORETTO G., SOATTO S.: Editable dynamic textures. In *IEEE Computer Vision and Pattern Recognition* (2003), pp. II: 137–142.

[EF01] EFROS A. A., FREEMAN W. T.: Image quilting for texture synthesis and transfer. *Proceedings of SIGGRAPH 2001* (2001), 341–346.

[EL99] EFROS A., LEUNG T.: Texture synthesis by non-parametric sampling. In *International Conference on Computer Vision* (1999), pp. 1033–1038.

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

[EMP*98] EBERT D., MUSGRAVE F., PEACHEY D., PERLIN K., WORLEY S.: *Texturing and Modeling: A Procedural Approach*. AP Professional, 1998.

[FF01] FOSTER N., FEDKIW R.: Practical animations of liquids. In *SIGGRAPH 2001, Computer Graphics Proceedings* (2001), Fiume E., (Ed.), ACM Press / ACM SIGGRAPH, pp. 23–30.

[FJP02] FREEMAN W. T., JONES T. R., PASZTOR E. C.: Example-based super-resolution. *IEEE Comput. Graph. Appl. 22*, 2 (2002), 56–65.

[FM96] FOSTER N., METAXAS D.: Realistic animation of liquids. *Graphical models and image processing: GMIP 58*, 5 (1996), 471–483.

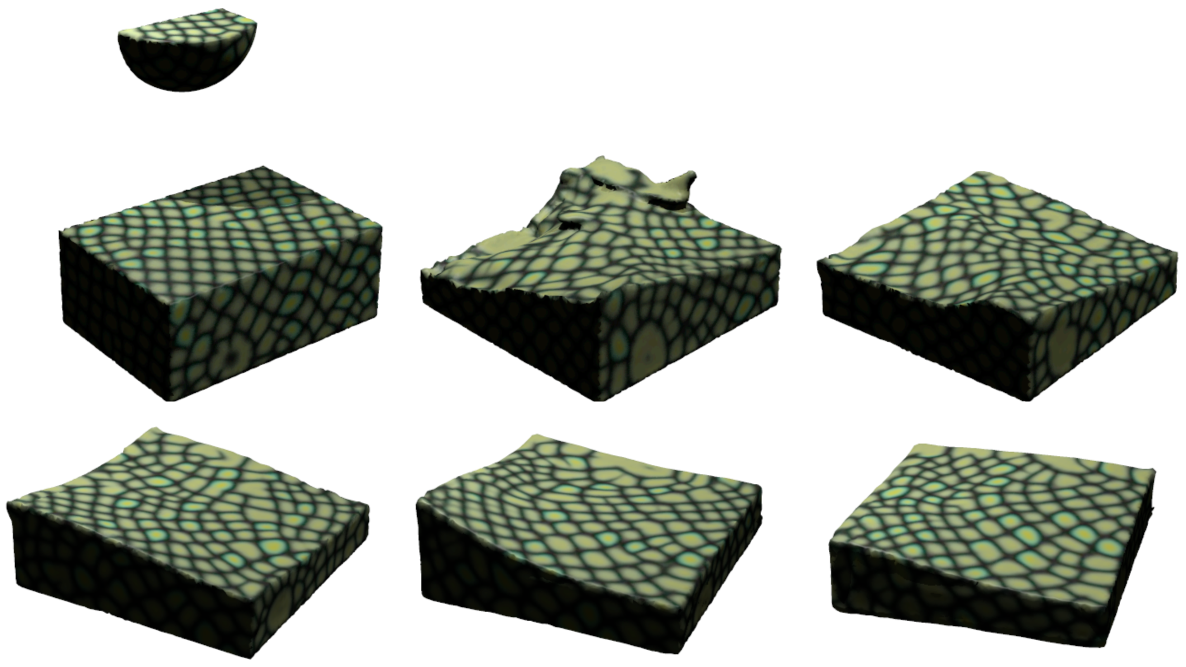

Figure 5: ***B****roken-Dam: A large volume of water is dropped into a broken dam, causing the water to splash. The textures on the water surfaces are split and merged nicely as the water surfaces undergo significant topological changes.*

[FSJ01] FEDKIW R., STAM J., JENSEN H. W.: Visual simulation of smoke. In *SIGGRAPH 2001, Computer Graphics Proceedings* (2001), Fiume E., (Ed.), ACM Press / ACM SIGGRAPH, pp. 15–22.

[GDN90] GRIEBEL M., DORNSEIFER T., NEUNHOEFFER T.: *Numerical Simulation in Fluid Dynamics: A Practical Introduction*. SIAM Monographcs on Mathematical Modeling and Computation. SIAM, 1990.

[GIS03] GORLA G., INTERRANTE V., SAPIRO G.: Texture synthesis for 3d shape representation. *IEEE Transactions on Visualization and Computer Graphics 9*, 4 (2003), 512–524.

[HB95] HEEGER D. J., BERGEN J. R.: Pyramid-based texture analysis/synthesis. *Proceedings of ACM SIGGRAPH 95* (August 1995), 229–238.

[HK03] HONG J.-M., KIM C.-H.: Animation of bubbles in liquid. In *Comp. Graph. Forum* (2003), vol. 22, pp. 253–263.

[JFK03] JOJIC N., FREY B., KANNAN A.: Epitomic analysis of appearance and shape. In *International Conference on Computer Vision* (2003).

[KEBK05] KWATRA V., ESSA I., BOBICK A., KWATRA N.: Texture optimization for example-based synthesis. *ACM Transactions on Graphics, SIGGRAPH 2005 24*, 3 (August 2005), 795–802.

[KSE*03] KWATRA V., SCHÖDL A., ESSA I., TURK G., BOBICK A.: Graphcut textures: Image and video synthesis using graph cuts. *ACM Transactions on Graphics, SIGGRAPH 2003 22*, 3 (July 2003), 277–286.

[LC02] LEVEQUE R. J., CRIGHTON D. G.: *Finite Volume Methods for Hyperbolic Problems*. Cambridge Texts in Applied Mathematics. Cambridge University Press, 2002.

[LGF04] LOSASSO F., GIBOU F., FEDKIW R.: Simulating water and smoke with an octree data structure. *ACM Trans. Graph. 23*, 3 (2004), 457–462.

[Liu02] LIU G. R.: *Mesh Free Methods: Moving Beyond the Finite Element Method*, 1st ed. CRC Press, 2002.

[LL03] LIU G. R., LIU M. B.: *Smoothed Particle Hydrodynamics: A Meshfree Particle Method*. World Scientific Pub. Co., Inc., 2003.

[LLX*01] LIANG L., LIU C., XU Y.-Q., GUO B., SHUM H.-Y.: Real-time texture synthesis by patch-based sampling. *ACM Transactions on Graphics Vol. 20, No. 3* (July 2001), 127–150.

[LY05] LIN S., YU Y.: Controllable smoke animation with guiding objects. *ACM Transactions on Graphics 24*, 1 (2005), 140–164.

[MCG03] MULLER M., CHARYPAR D., GROSS M.: Particle-based fluid simulation for interactive applications. *Proc. of ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2003).

[Ney03] NEYRET F.: Advected textures. *Proc. of ACM SIGGRAPH / Eurographics Symposium on Computer Animation* (2003), 147–153.

[PL98] PAGET R., LONGSTAFF I. D.: Texture synthesis via a noncausal nonparametric multiscale markov random field. *IEEE Transactions on Image Processing 7*, 6 (June 1998), 925–931.

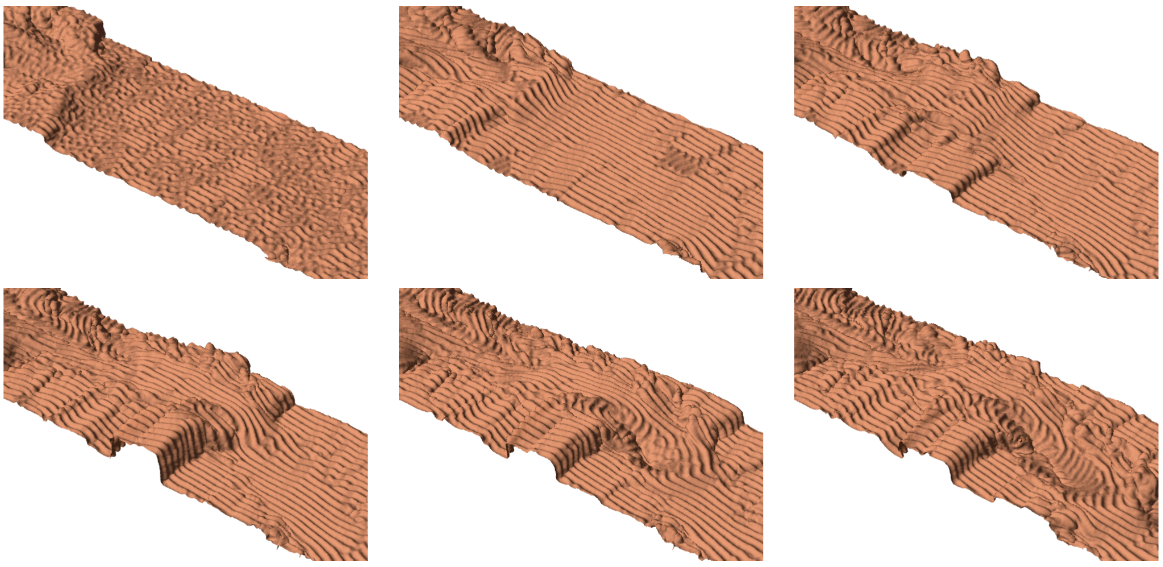

Figure 6: ***R****iver Scene: Chocolate river with a displacement texture map*

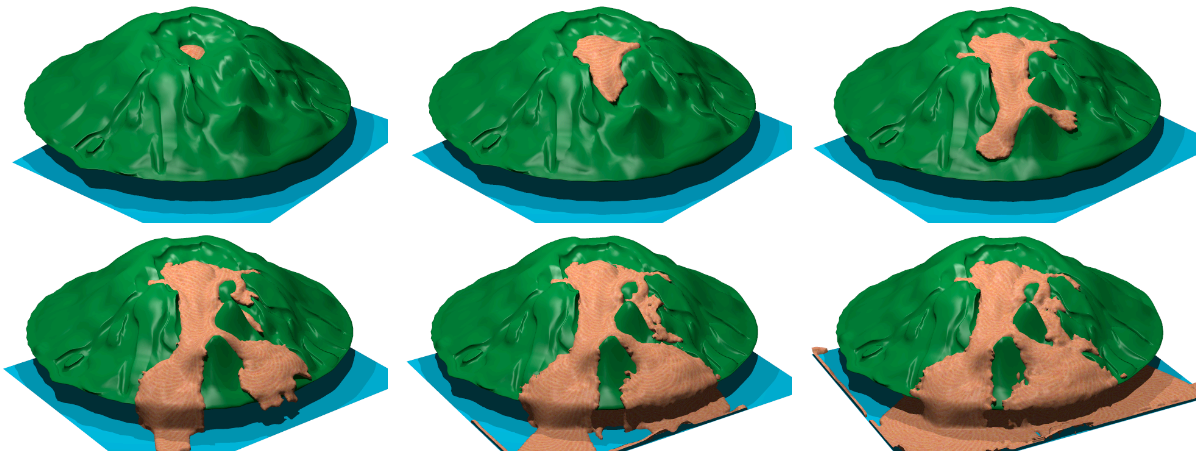

Figure 7: *Lava Scene: Lava flowing along a mountain.*

[PS00] PORTILLA J., SIMONCELLI E. P.: A parametric texture model based on joint statistics of complex wavelet coefficients. *International Journal of Computer Vision 40*, 1 (October 2000), 49–70.

[Set98] SETHIAN J.: Level set methods and fast marching methods: Evolving interfaces in computational geometry, 1998.

[SF95] STAM J., FIUME E.: Depicting fire and other gaseous phenomena using diffusion processes. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques* (1995), ACM Press, pp. 129–136.

[She96] SHEWCHUK J. R.: Triangle: Engineering a 2d quality mesh generator and delaunay triangulator. In *FCRC '96/WACG '96: Selected papers from the Workshop on Applied Computational Geormetry, Towards Geometric Engineering* (London, UK, 1996), Springer-Verlag, pp. 203–222.

[SSK05] SONG O., SHIN H., KO H.: Stable but nondissipative water. *ACM Transactions on Graphics 24*, 1 (2005).

[SSSE00] SCHÖDL A., SZELISKI R., SALESIN D. H., ESSA I.: Video textures. *Proceedings of ACM SIGGRAPH 2000* (July 2000), 489–498. ISBN 1-58113-208-5.

[Sta99] STAM J.: Stable fluids. In *Siggraph 1999, Computer Graphics Proceedings* (Los Angeles, 1999), Rockwood A., (Ed.), Addison Wesley Longman, pp. 121–128.

[Sta03] STAM J.: Flows on surfaces of arbitrary topology. *ACM Trans. on Graphics (Proc. of ACM SIGGRAPH)* (2003).

[SY04] SHI L., YU Y.: Inviscid and incompressible fluid simulation on triangle meshes. *Journal of Computer Animation and Virtual Worlds 15*, 3–4 (2004), 173–181.

[TA03] TAPONECCO F., ALEXA M.: Vector field visualization using markov random field texture synthesis. In *VISSYM '03: Proceedings of the symposium on Data visualisation 2003* (Aire-la-Ville, Switzerland, Switzerland, 2003), Eurographics Association, pp. 195–202.

[TFK*] TAKAHASHI T., FUJII H., KUNIMATSU A., HIWADA K., SAITO T., TANAKA K., UEKI H.: Realistic animation of fluid with splash and foam. In *Comp. Graph. Forum*, vol. 22.

[Tur91] TURK G.: Generating textures on arbitrary surfaces using reaction-diffusion. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1991), ACM Press, pp. 289–298.

[Tur92] TURK G.: Re-tiling polygonal surfaces. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1992), ACM Press, pp. 55–64.

[Tur01] TURK G.: Texture synthesis on surfaces. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2001), ACM Press, pp. 347–354.

[WL00] WEI L.-Y., LEVOY M.: Fast texture synthesis using tree-structured vector quantization. *Proceedings of ACM SIGGRAPH 2000* (July 2000), 479–488. ISBN 1-58113-208-5.

[WL01] WEI L.-Y., LEVOY M.: Texture synthesis over arbitrary manifold surfaces. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2001), ACM Press, pp. 355–360.

[WMT05] WANG H., MUCHA P. J., TURK G.: Water drops on surfaces. *ACM Trans. Graph. 24*, 3 (2005), 921–929.

[WSI04] WEXLER Y., SHECHTMAN E., IRANI M.: Space-time video completion. In *CVPR 2004* (2004), pp. 120–127.

[WY04] WU Q., YU Y.: Feature matching and deformation for texture synthesis. *ACM Transactions on Graphics (SIGGRAPH 2004)* (August 2004).

[YHBZ01] YING L., HERTZMANN A., BIERMANN H., ZORIN D.: Texture and shape synthesis on surfaces. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques* (London, UK, 2001), Springer-Verlag, pp. 301–312.

[ZMT04] ZHANG E., MISCHAIKOW K., TURK G.: *Vector Field Design on Surfaces*. Tech. Rep. 04-16, Georgia Institute of Technology, 2004.