Texturing Fluids

Vivek Kwatra, David Adalsteinsson, Theodore Kim, Nipun Kwatra[1], Mark Carlson[2], and Ming Lin
UNC Chapel Hill [1]Georgia Institute of Technology [2]DNA Productions

***Abstract*— We present a novel technique for synthesizing textures over dynamically changing fluid surfaces. We use both image textures as well as bump maps as example inputs. Image textures can enhance the rendering of the fluid by either imparting realistic appearance to it or by stylizing it, whereas bump maps enable the generation of complex micro-structures on the surface of the fluid that may be very difficult to synthesize using simulation. To generate temporally coherent textures over a fluid sequence, we transport texture information, i.e. color and local orientation, between free surfaces of the fluid 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 along the fluid velocity field for one time step, and interpolating it back onto the second surface – this operation, in part, uses a novel vector advection technique for transporting orientation vectors. 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 demonstrate our novel algorithm for texture synthesis on dynamically evolving fluid surfaces in several challenging scenarios.**

***Index Terms*— Texture Synthesis, Fluid Simulation, Surfaces, Vector Advection.**

I. INTRODUCTION

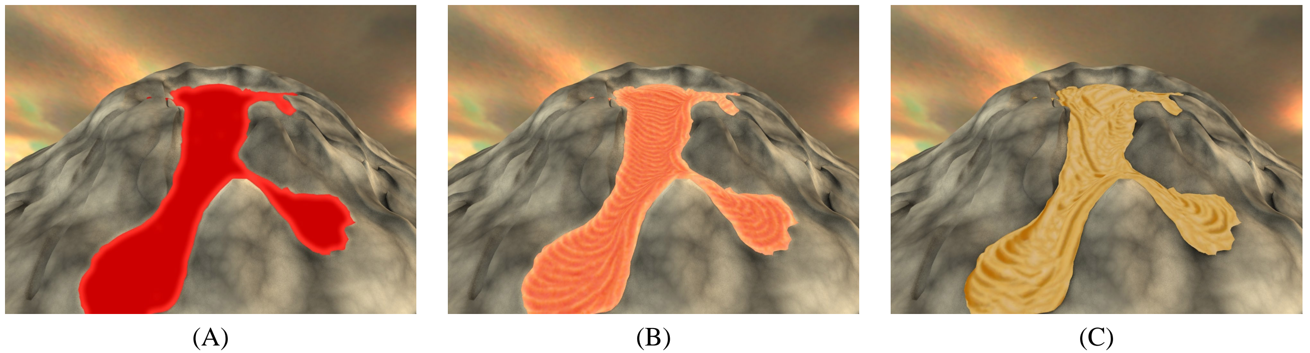
REALISTIC modeling, simulation, and rendering of fluid media have applications in various domains, including special effects, computer animation, electronic games, engineering visualization, and medical simulation. Often the computational expense involved in simulating complex fluid phenomena limit the 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 and visualization 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. Typically, these material properties behave like a texture, *i*.e., the fluid appearance consists of local features that are statistically self-similar over space and time. Even though these features diffuse as they move along with the fluid, they also re-generate over time, ensuring that, statistically, the appearance of the fluid features remains the same. We exploit this property of fluid behavior by obtaining example textural images of the fluid of interest, and using them to render the appearance of the fluid. Figure 1 demonstrates an example where the appearance of lava flowing over a mountain is made much more interesting by rendering it with texture. Our technique also provides a new way of visualizing surface properties of the fluid like flow and shape by exposing them using the appearance and evolution of the synthesized texture.

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 or features 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 (as shown in Figure 9). 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.

A. Main Results

In this paper, we present a novel texture synthesis algorithm for fluid flows. We assume that we have available a fluid simulator, which is capable of generating free surfaces of the simulated fluid medium as well as providing the 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 [1]–[3]. However, these approaches typically grow the texture point-by-point over the surface. We extend and generalize the idea of texture optimization [4] 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.



(A) (B) (C)

Fig. 1

LAVA SCENE RENDERED WITH AND WITHOUT FLUID TEXTURING. (A) SHOWS A FRAME FROM A LAVA ANIMATION RENDERED WITHOUT ANY TEXTURE SYNTHESIZED OVER IT, WHILE (B) AND (C) SHOW THE SAME FRAME RENDERED AFTER TEXTURING USING TWO DIFFERENT LAVA TEXTURES.

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 texture 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 the 3D velocity fields and free surfaces of the fluid at each iteration as output.
- It can take image textures, bump/displacement maps, as well as alpha maps as input.
- It performs texture synthesis on *dynamically evolving 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*[1] of the synthesized texture to the input texture, even while advecting both scalar (e.g. color) and vector quantities (local orientations) describing 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 have other applications as well. It takes into account rotation undergone by the vector when traveling through the velocity field in addition to the translation. 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 9 – 14.

[1] Visual similarity refers to the spatial continuity and the resemblance of visual appearance between the input and synthesized textures. See Section V-B for more details.

B. Organization

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

II. RELATED WORK

In this section, we briefly summarize recent advances in the research areas relevant to our work.

A. Example-based Texture Synthesis

Texture synthesis has been widely investigated in computer graphics. Various approaches are known, including pixel-based [5], [6], patch-based [7]–[9], and global synthesis [4], [10]–[12] techniques. Patch-based techniques usually obtain higher synthesis quality than pixel-based methods. Global techniques provide the most control, especially when coupled with an intuitive cost metric, and are therefore most desirable for fluid texturing. Our synthesis algorithm is based on a global texture optimization technique [4] which achieves a nice blend of quality and flexibility by working with patch sizes of varying degree from large to small.

An important class of texture synthesis techniques relevant to our work is that concerned with surface texture synthesis, where texture is synthesized directly over a 3D surface. The primary issues that arise here include representation of the texture and neighborhood construction and parameterization for performing search in the input texture. Turk [1] and Wei and Levoy [2] represent texture by storing color and local surface orientation on points uniformly distributed over the surface. Ying et al. [3] parameterize the surface locally onto

the plane to compute a texture atlas, which is then used to perform synthesis in the plane. Praun et al. [13] also perform local patch parameterizations for generating lapped textures. The technique of Maillot et al. [14] is significant in the context of surface parameterization for texture mapping. For point based representations, texture neighborhoods need to be constructed on the fly. While in [1], surface marching is used to compute the neighborhood, in [2], the mesh vertices are locally parameterized onto the plane to form each neighborhood. We use both kinds of neighborhoods in our work (see Section V-A for more details).

B. Flow-Guided Texturing and Visualization

Kwatra et al. [4] 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 triangulated surface in 3D – the motion of the surface being guided by a 3D fluid simulation as opposed to a planar flow field. Recently, Lefebvre and Hoppe [15] have also demonstrated texture advection on the plane and on static surfaces using appearance-space texture synthesis.

Bhat et al. [16] 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.

Wiebe and Houston [17] and Rasmussen et al. [18] perform fluid texturing by advecting texture coordinates along the flow field using level sets and particles, respectively. However, they do not address the issue of regeneration of texture at places of excessive stretch or compression. Neyret [19] proposed a method for applying stochastic textures to fluid flows that avoids a variety of visual artifacts, and demonstrated interesting 2D and 3D animations produced by coherent advection of the applied texture. This approach works in regular domains (2D or 3D) and the textures employed are primarily stochastic or procedural in nature to avoid blending artifacts. Our technique, on the other hand, is concerned with synthesis on the *free surface* of the fluid, and can handle a wider variety of textures.

There has been work in the scientific visualization community that makes use of texture for visualization and representation of vector fields [20] as well as shape [21]. We observe that, in a similar spirit, our technique can also be used for visualization of surface velocity fields as well as motion of deformable bodies, using *arbitrary* textures.

C. Fluid Simulation and Flows on Arbitrary Surfaces

Simulation of fluids and various related natural phenomena have received much recent attention. Foster and Metaxas [22] and Stam [23] were among the pioneers in using full 3D Navier-Stokes differential equations for generating fluid animations in computer graphics. Level set methods [24], [25] have been developed for tracking and rendering the free surface of the fluid. Specialized techniques for synthesizing detailed fluid phenomena like drops, bubbles, and foam etc, directly through simulation, have also been researched [26]–[28]. In the context of our work, fluid simulation is treated as a black box, where its outputs, namely the 3D velocity field and the free surface, are used by our algorithm to transport texture information between successive frames and synthesize the texture on the fluid surface, respectively.

Recently Stam [29], Shi and Yu [30] 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 [19].

Bargteil et al. [31] present a semi-Lagrangian surface tracking method for tracking surface characteristics of a fluid, such as color or texture coordinates. In a similar manner, our work also relies on fluid surface transport to advect color and other texture properties. However, in addition to these scalar quantities, we also track the orientation vectors on the fluid surface through the velocity field. These vectors are tracked to ensure that the synthesized texture has consistent orientation across (temporally) nearby free surfaces.

In work concurrent to ours, Bargteil et al. [32], [33] have also developed a similar technique for texturing liquid animations. Our neighborhood construction and search techniques as well as our orientation advection method are different from their work. Our work was also presented as a *technical sketch* in SIGGRAPH 2006 [34].

III. 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 temporally neighboring surfaces, and (iii) a method for transporting texture information from one surface to the other. Figure 2 shows a flow chart of how these three components interact with each other for fluid texturing. The surface texture synthesis module hands the textured surface over to the texture transporter, which in turn, transports texture information along the velocity field for a single time step, and hands this information back to the synthesis module.

The only requirements for a fluid simulator to work with our system are that it should be able to output the 3D fluid

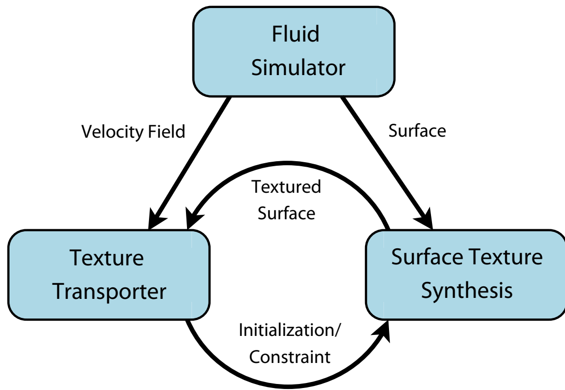


Fig. 2
OVERVIEW OF OUR FLUID TEXTURE SYNTHESIS SYSTEM.

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 V). We then transport the texture to the fluid surface for the second time step using our texture transport technique. The transported quantities include the texture colors (and any other associated properties like surface displacement, transparency, etc.) 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 surface for the second time step. 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.

IV. 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. These points are connected to form an auxiliary triangle mesh that aids in interpolation and neighborhood construction. Secondly, a smooth vector field is computed over this mesh that defines the local orientation (2D coordinate system) at each point on the surface[2]. The orientation field is used to map 3D points on the surface onto 2D points in a plane. This mapping is later used for comparing a surface patch against a pixel neighborhood in the input image. These operations are mostly similar to previous work, but we describe them here briefly for completeness.

A. Point Placement and Mesh Hierarchy

As discussed above, we store texture information (color and local orientation) in points placed on the surface being textured. Hence, the number of points will determine the *resolution* of the synthesized texture. 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. 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 explicitly.

We generate the points in a hierarchical fashion to represent the texture at varying resolutions. We follow the procedure of Turk [1]. At each level, we initialize the points by placing them randomly over the surface mesh, and then use the surface-restricted point repulsion procedure of Turk [35] to achieve uniform spacing between these points. 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 [36] for re-triangulating the original surface mesh using the new points. We use the Triangle library [37] for triangulation at each intermediate step.

B. 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. [38]. 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

[2]The curved nature of the surface implies that a unique vector cannot be used to define the 2D coordinate system at each point on the surface – unlike the case with a plane. This is due to the fact the coordinate system needs to lie in the tangent plane of the surface which itself changes from point to point. Consequently, we need to define an orientation vector *field* spread over the entire surface.

space angle (shown in the Figure 3A). The orientation vector at each point can now be represented 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 their orientations to make the same polar angle with the common edge between them, as shown in Figure 3B. 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. The orientation field is initialized to be the zero polar angle everywhere, after which multiple iterations of smoothing are performed. 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. The 3D orientation vector at the point is then obtained 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.

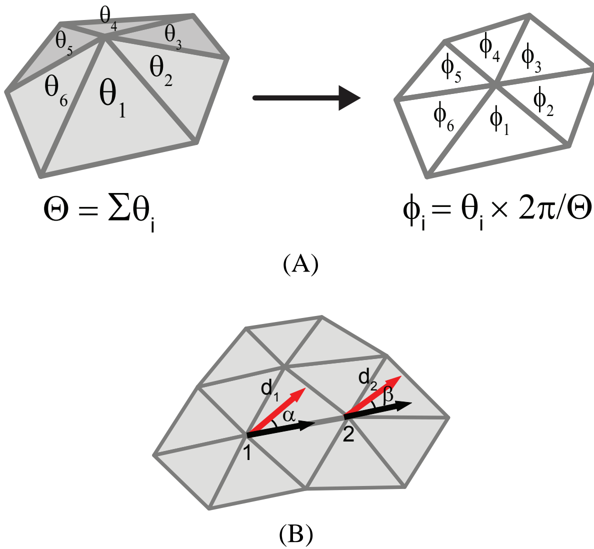


Fig. 3
(A) MAPPING ANGLES FROM MESH SPACE TO POLAR SPACE.
(B) ORIENTATIONS $\mathbf{d}_1$ AND $\mathbf{d}_2$ SHOULD MAKE SIMILAR *polar* ANGLES (α AND β RESPECTIVELY) WITH THE COMMON EDGE BETWEEN THEM.

V. SURFACE TEXTURE SYNTHESIS

Once we have constructed the mesh hierarchy and the orientation field, 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.

A. 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 [3], or by constructing local neighborhoods over the surface on the fly throuh either surface marching [1] or construction of local parameterizations [2]. Pre-computing the mapping from surface to plane gives nicer neighborhoods, but is tedious to compute, especially for a sequence of surfaces. We favor an *on the fly* approach because of its simplicity and scalability in handling a sequence of meshes.

We construct two types of neighborhoods on the mesh, which we refer to as *pixel neighborhoods* and *vertex neighborhoods* respectively. These two types of neighborhoods are used to interpolate color information back and forth between vertices[3] on the mesh and pixels in the (image) plane. Pixel neighborhoods are used to transfer information from mesh space to image space, while vertex neighborhoods perform the reverse operation, transferring information from image space to mesh space.

1) Pixel Neighborhood: A pixel neighborhood is defined as a set of points on the mesh whose 2D coordinates in the local orientation space around a central point map to integer locations in the plane; also, the neighborhood is bounded by a width w, which implies that if (i,j) are the 2D coordinates of a point in the neighborhood, then $-w/2 \leq i,j \leq w/2$. Given the orientation field, we have a coordinate system in the tangent plane of the surface mesh that we can use to march along the surface. The orientation direction $\mathbf{d}$ defines one axis of the coordinate system while the second axis $\mathbf{t}$ (orthonormal to $\mathbf{d}$) is obtained as the cross product between the normal at the surface point and $\mathbf{d}$[4]. We use this coordinate system to march along the surface in a fashion similar to [1], adding all surface points corresponding to valid integer pixel locations to the pixel neighborhood. Once a pixel neighborhood is constructed, it can be used to interpolate colors from mesh space to image space. For example, the color of the pixel at (i,j) is computed using barycentric interpolation of colors associated with the corners of the triangle in which the corresponding point lies. Figure 4 shows a pixel neighborhood mapped onto the surface of a triangle mesh.

2) Vertex Neighborhood: A vertex neighborhood in a mesh is defined as the set of vertices connected to each other and lying within a certain *geodeisc distance* – distance measured in the local orientation space of the mesh – to a central vertex. Given the vertex c as the center, the 2D location of a vertex in its neighborhood is computed as its displacement from c along the orientation field on the mesh. For a given *pixel* neighborhood width w, we include only those vertices in c's

[3]we use the term *vertex* here to distinguish it from the term *point* used earlier. A *point* can be any point on the surface, where as *vertices* refer to the set of points that are part of a triangle mesh that is being used to sample and represent the surface.

[4]vertex normals are *interpolated* from incident faces and the normal at a point is interpolated from vertex normals of the containing face.

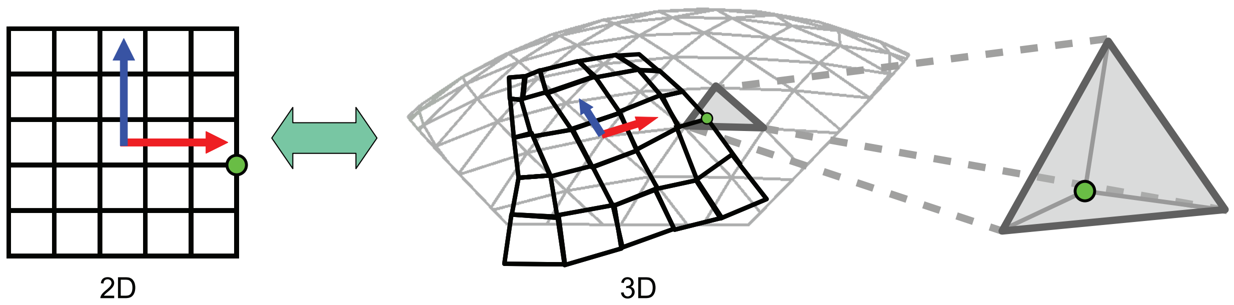


Fig. 4
PIXEL NEIGHBORHOOD CONSTRUCTION: A 2D PIXEL NEIGHBORHOOD IS MAPPED ONTO POINTS ON THE TRIANGLE MESH. THE RED ARROW IS THE ORIENTATION VECTOR, WHILE THE BLUE ARROW IS ITS ORTHONORMAL COMPLEMENT. THE POINT CORRESPONDING TO ANY GIVEN PIXEL IN THE NEIGHBORHOOD, LIKE THE ONE SHOWN AS A GREEN CIRCLE, LIES ON A SINGLE TRIANGLE OF THE MESH AS SHOWN ON THE RIGHT. THIS TRIANGLE IS USED TO DETERMINE TEXTURE INFORMATION AT THE GIVEN PIXEL THROUGH BARYCENTRIC INTERPOLATION.

neighborhood whose displacement vector (l_1, l_2), measured starting at c, is such that $-w/2 < l_1, l_2 < w/2$. To compute displacements of vertices in the neighborhood of c, we employ a local mesh flattening procedure similar to the one used in [2], [13]. We first consider all vertices in the 1-ring of c, *i*.e., vertices that are directly connected to c. If $\mathbf{d}$ represents the orientation vector at c and $\mathbf{t}$ represent its orthonormal complement, then the displacement vector of a vertex v in the 1-ring of c is $(\mathbf{v}\cdot\mathbf{d}, \mathbf{v}\cdot\mathbf{t})$, where $\mathbf{v} = (v-c)/\sigma$. Here, σ is the scaling factor between mesh space and image space. We then keep moving outwards to the 2-ring and so on until all neighborhood vertices are exhausted. Generally, on a curved surface, the displacement vector computed for a vertex will depend on the path one takes to reach that vertex. Hence, for each vertex, we compute its displacement as the average of those predicted by its neighbors, and also run a relaxation procedure to ensure an even better fit for the displacements.

Once a point neighborhood is constructed, it can be used to interpolate colors from image space to mesh space. Since the displacement of each vertex in the neighborhood corresponds to real-valued 2D coordinates, we use bilinear interpolation of colors at nearby integer locations in the image space to determine the color of the vertex. Figure 5 shows a vertex neighborhood where a set of connected points on the mesh are mapped onto the 2D plane.

B. Surface Texture Optimization

Our approach for synthesizing texture on a 3D surface is motivated by the texture optimization algorithm proposed by Kwatra et al. [4]. Their formulation is valid only on a 2D plane. We extend that formulation to handle synthesis on a arbitrary 3D 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 *vertex* neighborhoods of the textured surface and *image-space pixel* neighborhoods of the input texture example. To compare these two incongruous neighborhoods, we first interpolate colors from the image-space pixel neighborhood onto real-valued 2D locations corresponding to the vertex neighborhood, as described in Section V-A.2 . We then define the texture energy for a single vertex neighborhood to be the sum of squared differences between the colors of mesh vertices and the interpolated input colors at the vertex locations. The total energy of the textured surface is equal to the sum of energies over individual neighborhoods of the surface. If S denotes the 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 vertices of the mesh. $\mathbf{s}_p$ is the set of colors associated with vertices in a vertex neighborhood around the vertex p. $\mathbf{z}_p$ contains colors from a pixel neighborhood in Z, interpolated at 2D locations corresponding to the vertex neighborhood around p. The input texture neighborhood from which $\mathbf{z}_p$ is interpolated is the one that appears most similar to the *pixel neighborhood around* p – which is constructed as explained in Section V-A.1 – under the sum of squared differences. It should be noted that even though our energy formulation is described completely in terms of vertex neighborhoods, we need to resort to pixel neighborhoods during search for efficiency.

The set of vertices, $S^\dagger$, around which neighborhoods are constructed is a subset of the set of all vertices 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. See Figure 6 for a schematic explanation of how the texture energy is computed.

The energy function defined above measures the spatial consistency of the synthesized surface texture with respect to the input texture example. To ensure that the synthesized texture is temporally coherent as well, we add another term

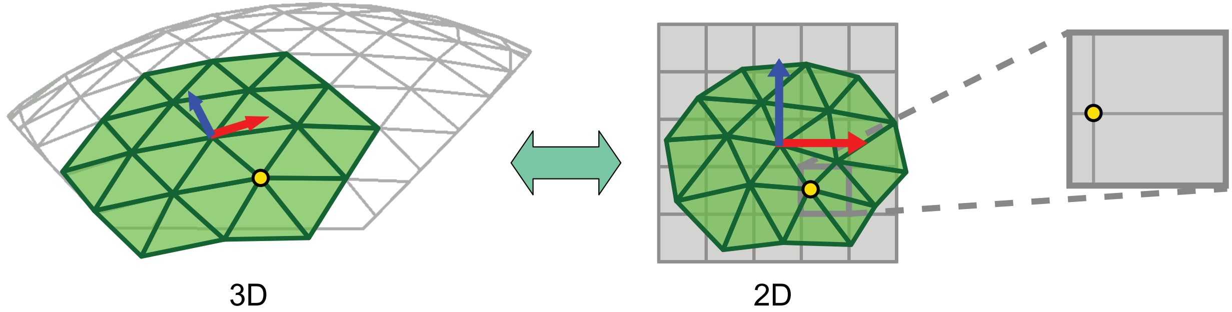


Fig. 5
VERTEX NEIGHBORHOOD CONSTRUCTION: A SET OF CONNECTED POINTS ON THE MESH ARE MAPPED ONTO THE 2D PLANE. THE ORIENTATION VECTOR AT THE CENTRAL VERTEX ALIGNS ITSELF WITH THE PRIMARY AXIS OF THE PLANE. EACH VERTEX MAPS TO A REAL-VALUED 2D LOCATION, AS SHOWN EXPLICITLY FOR THE YELLOW CIRCLED VERTEX. GIVEN A PLANAR PIXEL NEIGHBORHOOD, THE COLOR AT A MESH VERTEX IS DETERMINED THROUGH BILINEAR INTERPOLATION OF THE FOUR PIXELS THAT ITS 2D MAPPED LOCATION LIES BETWEEN.

which measures the similarity of the textured surface to the texture transported from the previous time step. The transported texture already specifies a color value vector, denoted as $\mathbf{c}$, corresponding to each vertex location in S. We treat $\mathbf{c}$ as a soft constraint on $\mathbf{s}$, *i*.e., we want the color of the transported texture and synthesized texture to be close to each other. The corresponding energy function is

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

where k iterates over all vertices in the mesh and λ_k is a weighting factor that controls the influence of transported texture color at a vertex 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 vertex that has greater color gradient in its vicinity. Note that we also use $\mathbf{c}$ as an initialization for $\mathbf{s}$. 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 surface being textured, 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 vertex $p \in S^{\dagger}$, construct a vertex neighborhood $\mathbf{s}_p$ and a pixel neighborhood $\mathbf{x}_p$ in the vicinity of p from the current surface texture $\mathbf{s}$. Assign colors to the pixels in $\mathbf{x}_p$ through barycentric interpolation on the mesh (Section V-A).
2) Find the closest input texture neighborhood, say $\mathbf{x}'_p$, for each pixel neighborhood $\mathbf{x}_p$ constructed above, and interpolate $\mathbf{x}'_p$ at real-valued 2D locations corresponding to the vertices in $\mathbf{s}_p$ to obtain the interpolated input neighborhood $\mathbf{z}_p$.
3) Re-compute the surface texture colors $\mathbf{s}$ by minimizing the total energy $E(\mathbf{s};\{\mathbf{z}_p\})$ with respect to $\mathbf{s}$, keeping the set of interpolated input neighborhoods $\{\mathbf{z}_p\}$ fixed. For the squared energy function used here, this corresponds to simply taking a weighted average of the colors predicted by 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, 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 run three optimization passes that use neighborhoods of size 33×33, 17×17, and 9×9 pixels respectively, in that order.

VI. 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, transparency, etc.) and (ii) the orientation vector 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 difficult to find neighborhoods in the input texture that match the transported texture.

A. 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,

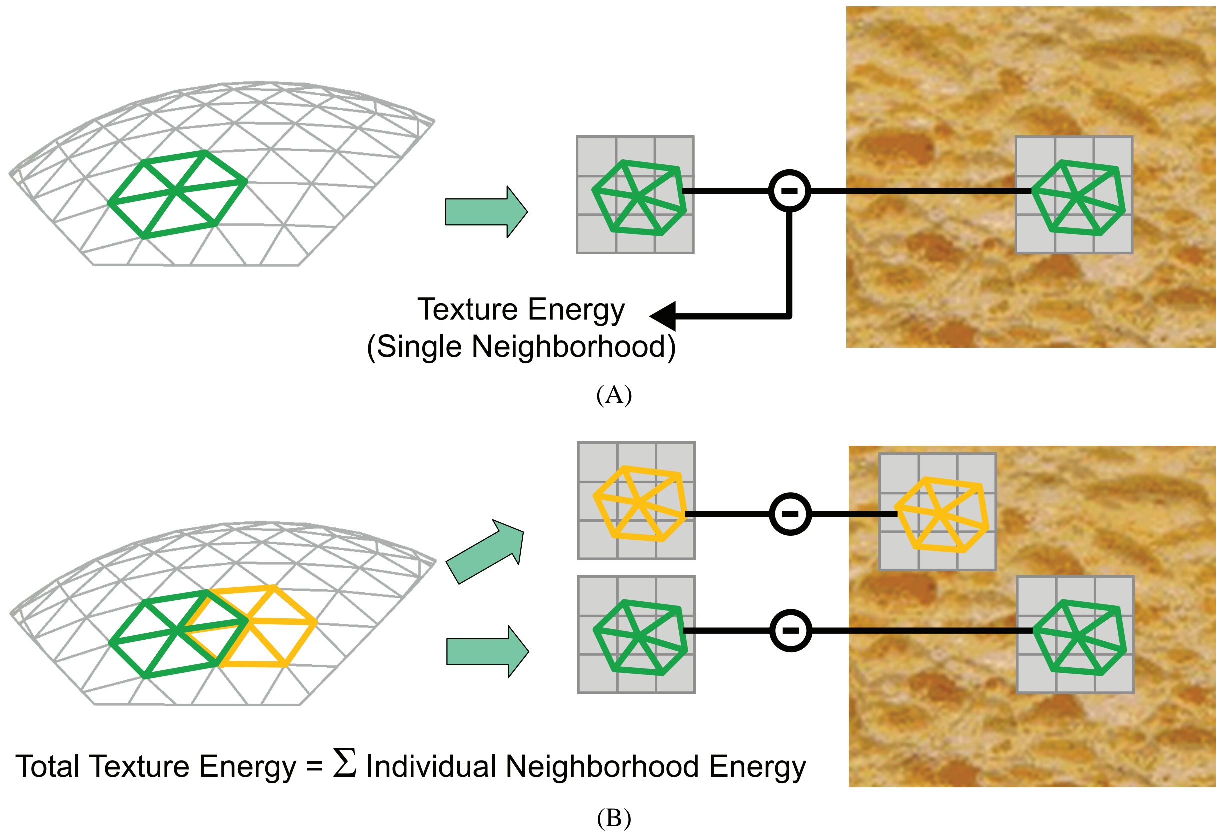


Fig. 6

TEXTURE ENERGY: (A) SHOWS THE TEXTURE ENERGY FOR A SINGLE VERTEX NEIGHBORHOOD. IT IS THE SQUARED COLOR-SPACE DISTANCE BETWEEN THE GIVEN VERTEX NEIGHBORHOOD AND ITS CLOSEST MATCH IN THE INPUT IMAGE TEXTURE. (B) SHOWS THE TOTAL TEXTURE ENERGY FOR THE SURFACE TEXTURE. IT IS SIMPLY THE SUM OF ENERGIES OF INDIVIDUAL NEIGHBORHOODS.

we advect 3D vertex coordinates instead of texture color because vertex coordinates can usually be safely interpolated without loss of resolution. Advection of 3D coordinates is conceptually equivalent to back-tracking each point in the volumetric grid to its source location in the previous time-step. For a given vertex in the new mesh, one can obtain its back-tracked location by interpolating from nearby grid points. Its texture color is then obtained through barycentric interpolation over the triangle closest to this back-tracked location. The various steps are enumerated in order below:

1) The first step is to determine the 3D coordinates at grid locations. This is quite 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:

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

where φ is the scalar being advected, $\mathbf{u}$ is the velocity field obtained from the fluid simulation, and $\nabla\varphi$ is the spatial gradient of the scalar field. We save the fluid velocity field at all intermediate time steps that the simulation may have stepped through, and use those steps while performing the advection update as well. The update is performed using a first order upwind scheme [39].
3) After the advection is complete, we have the advected coordinates at each grid point. These coordinates are interpolated onto the vertices of the new surface corresponding to the next time step. Each new surface vertex 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 vertex as the color of the back-tracked location. Let $\mathbf{x}$ be the back-tracked location. The color of $\mathbf{x}$ is obtained by first finding the point, $\mathbf{p}$, on the previous surface that is nearest to $\mathbf{x}$. We compute the color of $\mathbf{p}$ through barycentric interpolation of the colors at the corners of the triangle on which $\mathbf{p}$ lies. This color is then also used as the color for the back-tracked location $\mathbf{x}$. To speed up the search for the nearest surface points, we use a hash table to store the triangles of the mesh.

It should be noted that the backtracked coordinates may not always be close to the surface, in which case the projection onto the surface may return undesirable texture color values. However, our texture synthesis algorithm automatically handles this by *pasting* over the bad regions with new input patches that are visually consistent with the surrounding regions.

B. Orientation Advection

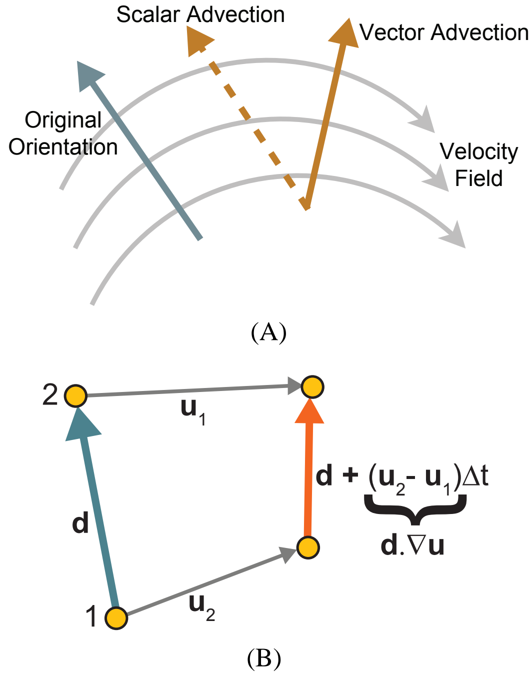


Fig. 7
(A) Transporting an orientation vector by only advecting its scalar components will (incorrectly) only translate the vector. Our vector advection technique correctly transports the vector by translating as well as rotating it. (B) The orientation vector **d** between points 1 and 2 gets distorted as a function of the gradient of the velocity field along **d**.

The transport of orientations is not as straightforward as the transport of color. This is because orientations are vectors which may rotate 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 7A 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, $-\mathbf{u} \cdot \nabla \mathbf{d}$, is the scalar advection term as applied to each component of the orientation vector **d**. The term $\mathbf{d} \cdot \nabla \mathbf{u}$ computes the derivative of velocity **u** along the orientation **d**. As shown in Figure 7B, **d** gets *distorted* by the velocity field, because its head and tail undergo different displacements. This difference in displacements is governed by the difference in the velocity values at the head and the tail, which is simply the velocity field gradient $\nabla \mathbf{u}$ along the orientation **d**. Finally, since we are only interested in the direction of the **d** and not its magnitude, we need to normalize the distortion so that it reduces to a rotation. This is achieved through the term $(\mathbf{I} - \mathbf{d}\mathbf{d}^{\mathbf{T}})$, which is a projection operator that projects $\mathbf{d} \cdot \nabla \mathbf{u}$ to the plane perpendicular to **d**. This projection essentially ensures that, in differential terms, **d** always stays a unit vector.

To perform advection of **d**, we follow similar steps as for scalar advection, with the following changes. In step 1, we *extend* the orientation field from the surface onto the grid using the technique of [40]. 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 IV. We typically, re-run the diffusion algorithm to smoothen the orientation field. Note that unlike texture color, the orientation field is smooth to begin with, hence loss of resolution is not a concern. Therefore, we interpolate it directly without any need to go through step 4.

VII. Results

We have implemented the techniques presented here in C++ and rendered the results using 3Delight®. We use a grid-based method for fluid simulation [23], [41], [42]. However, our technique is general enough to work with other types of simulators like finite volume based techniques [43]. We applied our implementation on several challenging scenarios with several different textures (shown in Figure 8).

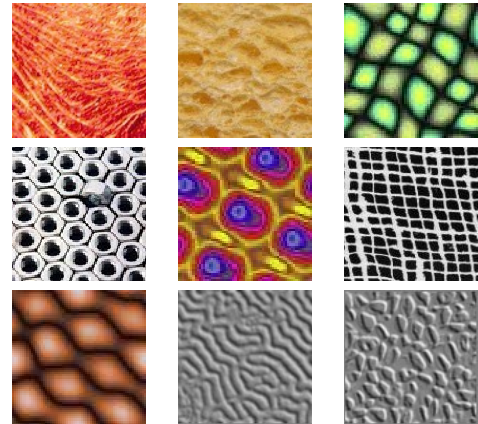


Fig. 8
Texture exemplars used in our results. Last two are bump-map textures used with the river.

The first scene is a broken-dam simulation as shown in Figure 9 and Figure 10, 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. The water surface is stylized using various different textures in the shown examples. In the accompanied video, note that the texture on the surface is split and merged seamlessly, as the surface undergoes substantial topological changes. Figure 11A shows a comparison of our result for this simulation with the result obtained through pure advection (*i*.e., no texture synthesis after the first frame).

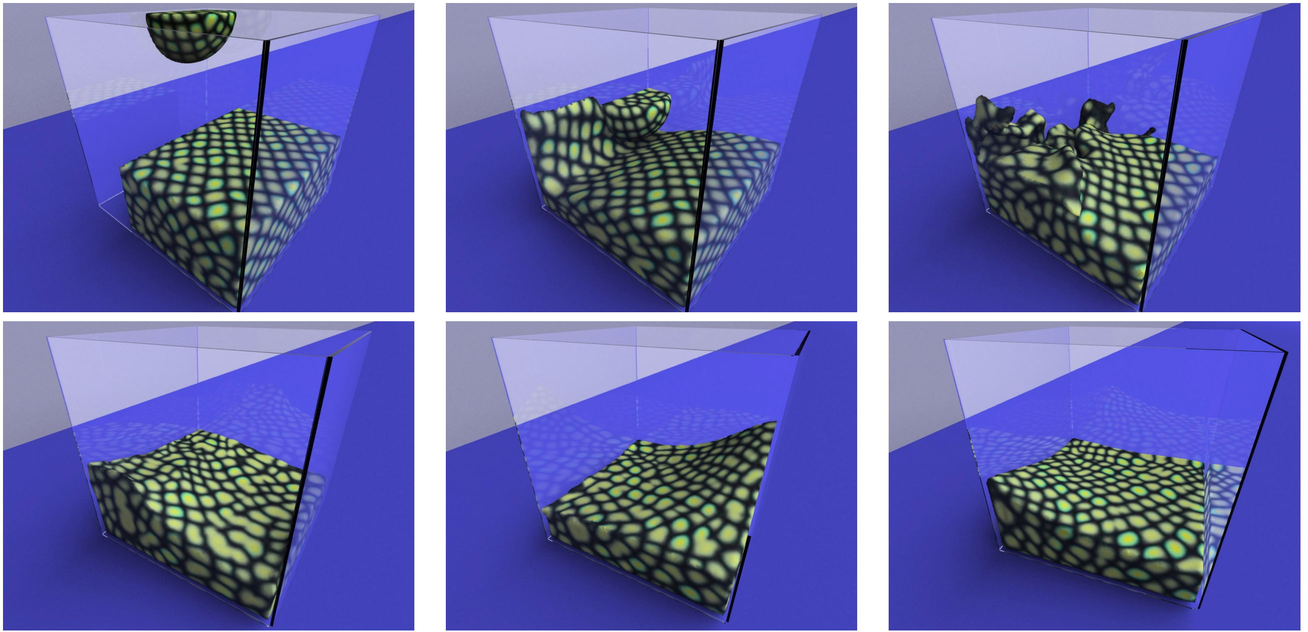


Fig. 9
BROKEN-DAM: A LARGE VOLUME OF WATER IS DROPPED INTO A BROKEN DAM, CAUSING THE WATER TO SPLASH. THE WATER SURFACE IS STYLIZED USING A GREEN SCALES TEXTURE. THE TEXTURE FEATURES ON THE SURFACE SPLIT AND MERGE NICELY AS THE SURFACE UNDULATES. THE VIEWPOINT IS ROTATING AROUND THE SCENE.

Even though advecting the texture ensures perfect temporal coherence, the visual quality of the texture is lost due to diffusion.

The second simulation is a river scene shown in Figure 12 and Figure 13, where the water flows down the stream and carries the surface texture with it. In Figure 12A, we show the river with a bump mapped texture which creates the impression of small scale waves on the surface of the river. Figure 12B shows the same scene with a different bump texture that gives the impression of rocks floating in the river. Figure 12C uses the same texture as Figure 12B but treats it as a transparency texture coupled with color to give the impression of floating leaves on the river surface. Figure 11B compares the rendering of a frame of the bump mapped river with the rendering of the original fluid surface generated by the simulation. It should be easy to see that the texture adds a lot of detail and micro-structure to the river, which is missing from the original surface.

The third simulation is that of lava flow, shown in Figure 14 in which lava flows from the top of the mountain downwards onto the walls. Results are shown for four different textures. The first two examples are more realistic while the last two are stylistic.

The computational complexity of our algorithm is dominated by nearest neighbor search during texture synthesis, and mesh hierarchy construction during surface preparation. Depending upon the number of points used for the mesh, hierarchy construction takes between $3-15$ minutes per frame[5]. However, the mesh hierarchy is computed offline for each 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 results for river, lava and broken dam examples. We also show a comparison between the results of our technique and the results obtained by using pure advection, as well as comparisons with results obtained without using any texture synthesis at all. All our results are also available online at `http://gamma.cs.unc.edu/TexturingFluids/`.

VIII. DISCUSSION & FUTURE WORK

We have presented a novel synthesis algorithm for advecting textures on dynamically changing fluid surfaces over time. Our work successfully demonstrates transport of textures along 3D fluid flows, which 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 surface texture optimization process with an advection scheme based on a level-set method for tracking the surface characteristics. We also explicitly model and solve for the advection of vector quantities like surface orientation through the velocity field.

[5]Only one level of hierarchy is constructed for all frames of the animation subsequent to the first one, which uses three levels.

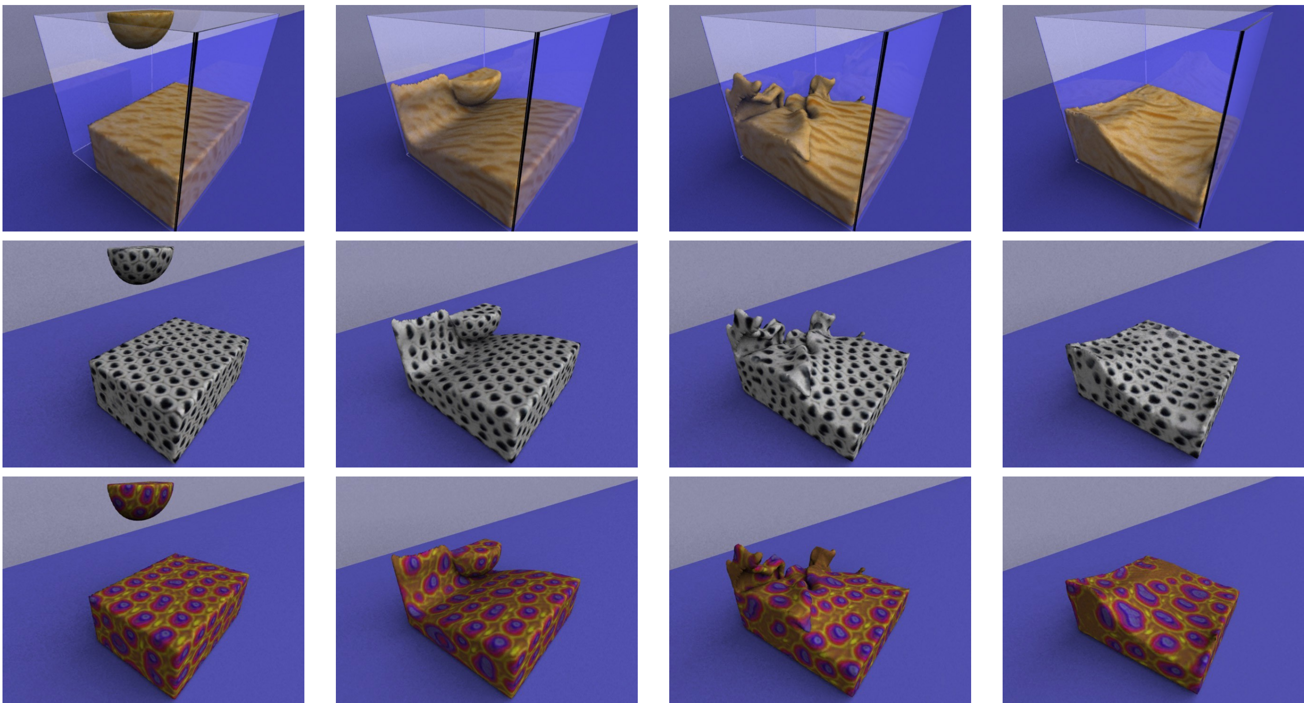


Fig. 10
BROKEN-DAM RENDERED USING DIFFERENT TEXTURE EXEMPLARS.

One can envision using our technique as a rendering mechanism in conjunction with fluid simulation techniques. It can be used to enhance the complexity of the synthesized fluid phenomena by adding surface micro-structures such as waves, and eliminating the need to manually model photometric properties of materials such as lava. Another interesting application is flow visualization. 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.

A limitation of our technique, which is relevant to surface texture synthesis in general, is that it is often impossible to define an orientation vector field on the surface that is smooth everywhere. Overlapping neighborhoods near singularity points like sources, sinks and vortices of the field, may not align well with each other, because computation of the 2D mapping of vertices becomes unstable in these regions. This can sometimes cause excessive blurring near singularity points.

Currently, the texture synthesis process is agnostic of any properties of the fluid surface that may affect its appearance. For example, one might want the appearance of the texture to be affected in a certain way by vortices and high curvature regions on the fluid surface. As part of future research, we would like to add to our technique, the ability to incorporate such relationships into the synthesis process. Among other future directions, we would like to extend our approach to handle 3D volume textures as well as video textures.

REFERENCES

[1] G. Turk, “Texture synthesis on surfaces,” in *SIGGRAPH ’01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM Press, 2001, pp. 347–354.

[2] L.-Y. Wei and M. Levoy, “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: ACM Press, 2001, pp. 355–360.

[3] L. Ying, A. Hertzmann, H. Biermann, and D. Zorin, “Texture and shape synthesis on surfaces,” in *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*. London, UK: Springer-Verlag, 2001, pp. 301–312.

[4] V. Kwatra, I. Essa, A. Bobick, and N. Kwatra, “Texture optimization for example-based synthesis,” *ACM SIGGRAPH*, vol. 24, no. 3, pp. 795–802, August 2005, appears in ACM Transactions on Graphics (TOG). [Online]. Available: http://www.cc.gatech.edu/cpl/projects/textureoptimization/

[5] A. Efros and T. Leung, “Texture synthesis by non-parametric sampling,” in *International Conference on Computer Vision*, 1999, pp. 1033–1038.

[6] L.-Y. Wei and M. Levoy, “Fast texture synthesis using tree-structured vector quantization,” *Proceedings of ACM SIGGRAPH 2000*, pp. 479–488, July 2000, iSBN 1-58113-208-5.

[7] A. A. Efros and W. T. Freeman, “Image quilting for texture synthesis and transfer,” *Proceedings of SIGGRAPH 2001*, pp. 341–346, 2001.

[8] L. Liang, C. Liu, Y.-Q. Xu, B. Guo, and H.-Y. Shum, “Real-time texture synthesis by patch-based sampling,” *ACM Transactions on Graphics*, vol. Vol. 20, No. 3, pp. 127–150, July 2001.

[9] V. Kwatra, A. Schödl, I. Essa, G. Turk, and A. Bobick, “Graphcut textures: Image and video synthesis using graph cuts,” *ACM SIGGRAPH*, vol. 22, no. 3, pp. 277–286, July 2003, appears in ACM Transactions on Graphics (TOG). [Online]. Available: http://www.cc.gatech.edu/cpl/projects/graphcuttextures/

[10] R. Paget and I. D. Longstaff, “Texture synthesis via a noncausal nonparametric multiscale markov random field,” *IEEE Transactions on Image Processing*, vol. 7, no. 6, pp. 925–931, June 1998.

[11] W. T. Freeman, T. R. Jones, and E. C. Pasztor, “Example-based super-resolution,” *IEEE Comput. Graph. Appl.*, vol. 22, no. 2, pp. 56–65, 2002.

[12] Y. Wexler, E. Shechtman, and M. Irani, “Space-time video completion,” in *CVPR 2004*, 2004, pp. 120–127.

[13] E. Praun, A. Finkelstein, and H. Hoppe, “Lapped textures,” *Proc. of ACM SIGGRAPH*, pp. 465–470, 2000.

[14] J. Maillot, H. Yahia, and A. Verroust, “Interactive texture mapping,” in *SIGGRAPH ’93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM Press, 1993, pp. 27–34.

[15] S. Lefebvre and H. Hoppe, “Appearance-space texture synthesis,” in

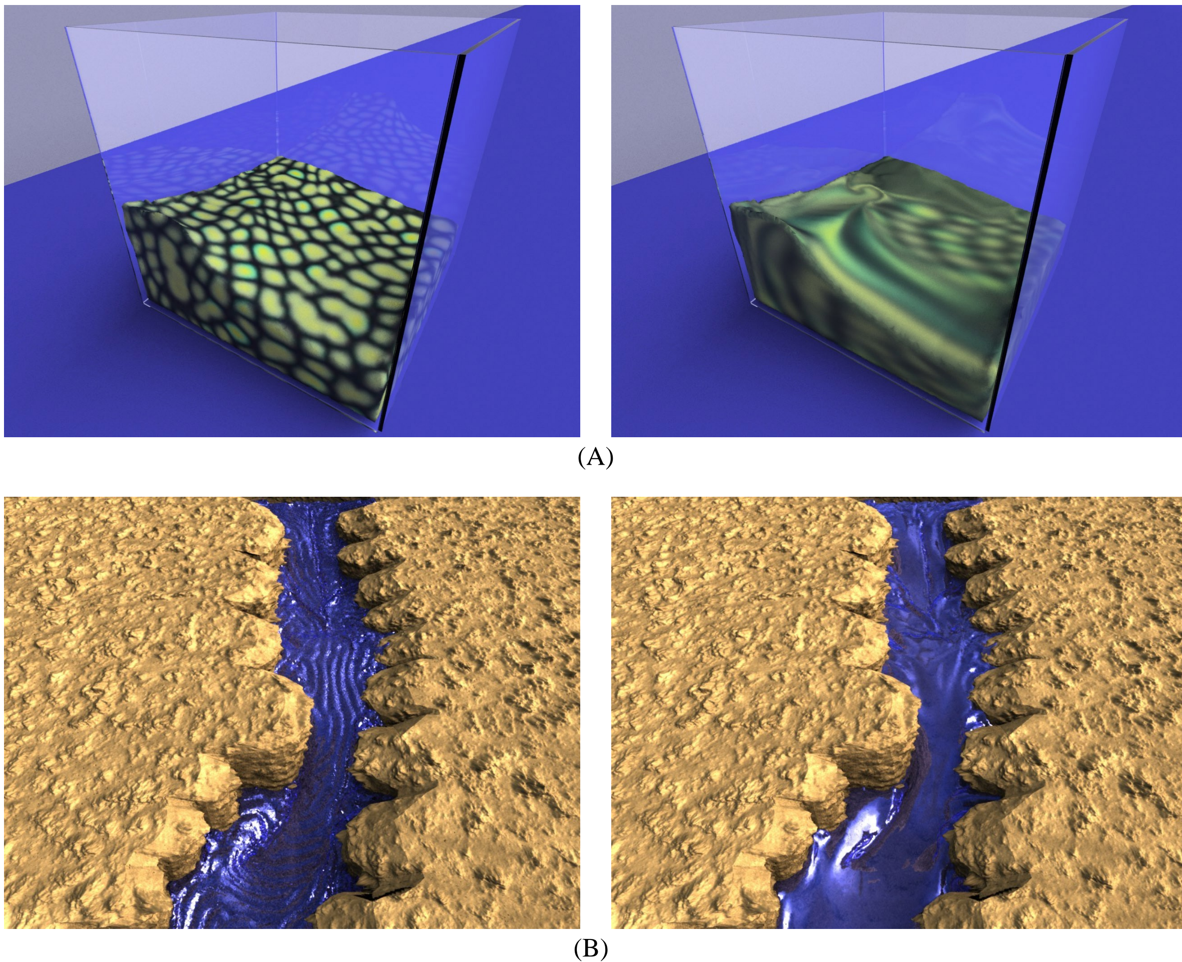


Fig. 11

(A) COMPARISON WITH PURE ADVECTION: OUR RESULT IS SHOWN ON THE LEFT, WHILE THE RESULT OF ADVECTION IS SHOWN ON THE RIGHT.
(B) COMPARING THE RIVER SCENE: RENDERED WITH FLUID TEXTURING ON THE LEFT AND WITHOUT ANY TEXTURE ON THE RIGHT.

SIGGRAPH '06: ACM SIGGRAPH 2006 Papers. New York, NY, USA: ACM Press, 2006, pp. 541–548.

[16] K. S. Bhat, S. M. Seitz, J. K. Hodgins, and P. K. Khosla, "Flow-based video synthesis and editing," *ACM Transactions on Graphics (SIGGRAPH 2004)*, vol. 23, no. 3, August 2004.

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

[18] N. Rasmussen, D. Enright, D. Nguyen, S. Marino, N. Sumner, W. Geiger, S. Hoon, and R. Fedkiw, "Directable photorealistic liquids," in *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, 2004, pp. 193–202.

[19] F. Neyret, "Advected textures," *Proc. of ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, pp. 147–153, 2003.

[20] F. Taponecco and M. Alexa, "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: Eurographics Association, 2003, pp. 195–202.

[21] G. Gorla, V. Interrante, and G. Sapiro, "Texture synthesis for 3d shape representation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 9, no. 4, pp. 512–524, 2003.

[22] N. Foster and D. Metaxas, "Realistic animation of liquids," *Graphical models and image processing: GMIP*, vol. 58, no. 5, pp. 471–483, 1996. [Online]. Available: citeseer.nj.nec.com/foster95realistic.html

[23] J. Stam, "Stable fluids," in *Siggraph 1999, Computer Graphics Proceedings*, A. Rockwood, Ed. Los Angeles: Addison Wesley Longman, 1999, pp. 121–128. [Online]. Available: citeseer.nj.nec.com/stam99stable.html

[24] N. Foster and R. Fedkiw, "Practical animations of liquids," in *SIGGRAPH 2001, Computer Graphics Proceedings*, E. Fiume, Ed. ACM Press / ACM SIGGRAPH, 2001, pp. 23–30. [Online]. Available: citeseer.nj.nec.com/foster01practical.html

[25] D. Enright, S. Marschner, and R. Fedkiw, "Animation and rendering of complex water surfaces," in *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. ACM Press, 2002, pp. 736–744.

[26] J.-M. Hong and C.-H. Kim, "Animation of bubbles in liquid," in *Comp. Graph. Forum*, vol. 22, no. 3, 2003, pp. 253–263.

[27] T. Takahashi, H. Fujii, A. Kunimatsu, K. Hiwada, T. Saito, K. Tanaka, and H. Ueki, "Realistic animation of fluid with splash and foam," in *Comp. Graph. Forum*, vol. 22, no. 3, 2003, pp. 391–401.

[28] H. Wang, P. J. Mucha, and G. Turk, "Water drops on surfaces," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 921–929, 2005.

[29] J. Stam, "Flows on surfaces of arbitrary topology," *ACM Trans. on Graphics (Proc. of ACM SIGGRAPH)*, 2003.

[30] L. Shi and Y. Yu, "Inviscid and incompressible fluid simulation on triangle meshes," *Journal of Computer Animation and Virtual Worlds*, vol. 15, no. 3–4, pp. 173–181, 2004.

[31] A. W. Bargteil, T. G. Goktekin, J. F. O'Brien, and J. A. Strain, "A semi-lagrangian contouring method for fluid simulation," *ACM Transactions on Graphics*, vol. 25, no. 1, 2006.

[32] A. W. Bargteil, F. Sin, J. E. Michaels, T. G. Goktekin, and J. F. O'Brien, "A texture synthesis method for liquid animations," *ACM SIGGRAPH 2006 Technical Sketches*, July 2006.

[33] ——, "A texture synthesis method for liquid animations," *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, September 2006.

[34] V. Kwatra, D. Adalsteinsson, N. Kwatra, M. Carlson, and M. Lin, "Texturing fluids," *ACM SIGGRAPH 2006 Technical Sketches*, July 2006.

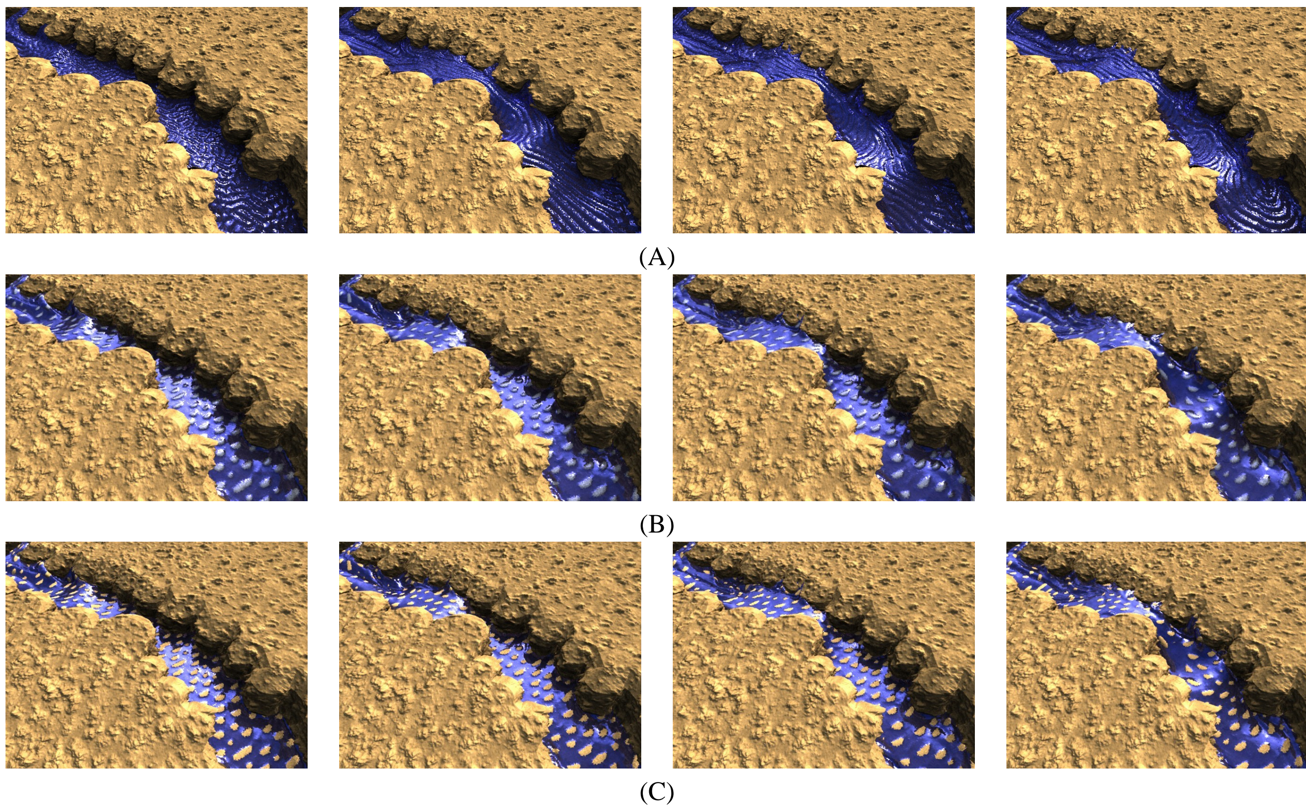


Fig. 12

RIVER SCENE (SIDE-ON VIEW): RIVER SCENE WITH BUMP-MAP AND TRANSPARENCY TEXTURES CREATING THE IMPRESSION OF (A) SMALL SCALE WAVES, (B) FLOATING ROCKS AND (C) FLOATING LEAVES ON THE SURFACE OF THE RIVER.

[35] G. Turk, "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: ACM Press, 1991, pp. 289–298.

[36] ——, "Re-tiling polygonal surfaces," in *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM Press, 1992, pp. 55–64.

[37] J. R. Shewchuk, "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: Springer-Verlag, 1996, pp. 203–222.

[38] E. Zhang, K. Mischaikow, and G. Turk, "Vector field design on surfaces," Georgia Institute of Technology, Tech. Rep. 04-16, 2004. [Online]. Available: http://www.cc.gatech.edu/grads/z/Eugene.Zhang/vecfld_design.html

[39] J. Sethian, "Level set methods and fast marching methods: Evolving interfaces in computational geometry," 1998. [Online]. Available: citeseer.ist.psu.edu/sethian99level.html

[40] D. Adalsteinsson and J. Sethian, "The fast construction of extension velocities in level set methods," *Journal of Computational Physics*, vol. 148, pp. 2–22, 1999.

[41] M. Griebel, T. Dornseifer, and T. Neunhoeffer, *Numerical Simulation in Fluid Dynamics: A Practical Introduction*, ser. SIAM Monographcs on Mathematical Modeling and Computation. SIAM, 1990.

[42] M. Carlson, P. Mucha, and G. Turk, "Rigid fluid: Animating the interplay between rigid bodies and fluid," in *Proc. of the ACM SIGGRAPH*. ACM Press, 2004.

[43] R. J. LeVeque and D. G. Crighton, *Finite Volume Methods for Hyperbolic Problems*, ser. Cambridge Texts in Applied Mathematics. Cambridge University Press, 2002.

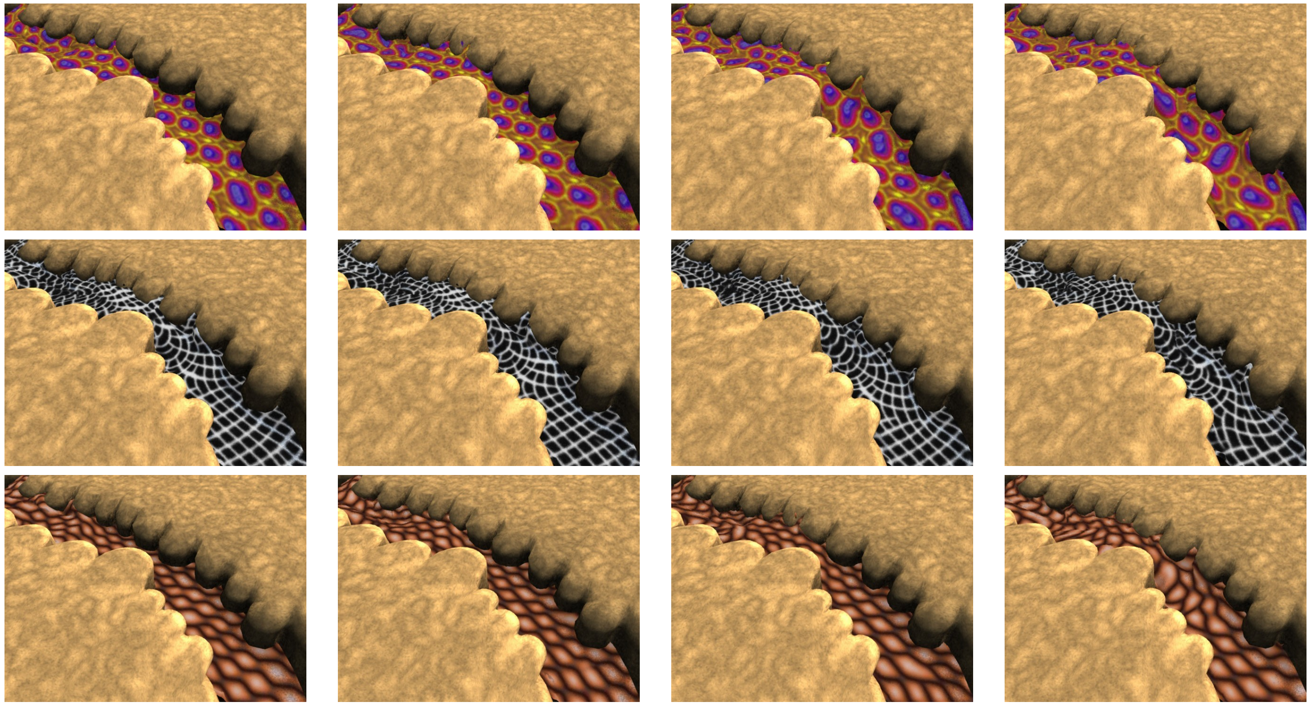


Fig. 13
RIVER SCENE RENDERED WITH VARIOUS IMAGE TEXTURES.

Fig. 14
LAVA SCENE: LAVA FLOWING ALONG A MOUNTAIN, SHOWN WITH DIFFERENT TEXTURES.